

IMPROVING EFFICACY AND EFFICIENCY OF HYPOTHESIS
ADAPTATION AND FEDERATED LEARNING

by

Farshid Varno

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

at

Dalhousie University
Halifax, Nova Scotia
July 2023

© Copyright by Farshid Varno, 2023

Dedicated to my mother, who has always put my happiness before her own and sacrificed so much for my well-being.

Table of Contents

List of Tables	vii
List of Figures	viii
Abstract	xii
List of Abbreviations	xiii
Glossary	xviii
Acknowledgements	xxvi
Chapter 1 Introduction	1
1.1 Motivation	1
1.2 Background	4
1.3 Research Objectives	6
1.3.1 Task Adaptation	7
1.3.2 Federated Learning	8
1.4 Contributions	9
1.5 Outline	9
Chapter 2 Background to Deep Learning	12
2.1 Supervised Deep Learning	12
2.2 Gradient Descent	15
2.3 Stochastic Gradient Descent	16
2.4 Mini-batch Stochastic Gradient Descent	20
2.5 Resilient Gradient-based Methods	20
Chapter 3 Background to Transfer Learning	22
3.1 Content-based Categorization of Transfer Learning	23
3.1.1 Data Transfer Learning	23
3.1.2 Hypothesis Transfer Learning	24
3.1.3 Information Transfer Learning	27

3.2	Federated Learning	28
Chapter 4	Related Work	31
4.1	Deep Learning and Convolutional Neural Networks	31
4.2	Hypothesis Transfer Learning	32
4.2.1	The First HTL Attempts	32
4.2.2	Linking ANNs and Symbolic Representation	33
4.2.3	Task Decomposition	34
4.3	Task Adaptation	34
4.3.1	Unsupervised Pretraining	34
4.3.2	Pretraining CNNs	35
4.3.3	Supervised Pretraining for CNNs	35
4.4	Continual Learning	35
4.5	Distributed Machine Learning	36
4.5.1	Federated Learning	37
4.5.2	Client Drift	38
Chapter 5	Incremental Tuning with Normalized Features	40
5.1	Background	41
5.2	Problem Statement	42
5.3	Noise Reduction in Feature-tuning	44
5.4	ENTAME	45
5.4.1	Feature Normalization	46
5.4.2	Maximum Entropy Initialization	46
5.5	Discussion	47
5.5.1	Maximum Entropy Predicted Labels	47
5.5.2	Feature Normalization	48
5.5.3	Generalized Maximum Entropy Initialization	48
5.6	Experimental Results	49
5.6.1	Feature-tuning with ENTAME	49
5.6.2	Gradual Increase of the Norm of Head's Weights	55
5.6.3	Domain Similarity	55
5.6.4	Continuous Hypothesis Transfers	61
5.7	Conclusion	65

Chapter 6	Incremental Tuning with Decoupled Step Sizes	69
6.1	Problem statement	69
6.2	FAST	70
6.3	Discussion	71
6.3.1	Geometric Interpretation	71
6.3.2	Velocity Analysis	75
6.3.3	Optimization Algorithm	76
6.4	Experiments	78
6.4.1	Catastrophic Forgetting	78
6.4.2	Quick Head Learning	81
6.4.3	Head Warmup	83
6.4.4	Decoupled Learning Rates	84
6.4.5	Optimization	86
6.4.6	Convergence Performance	91
6.4.7	FAST Compared to ENTAME	93
6.5	Conclusion	94
Chapter 7	Adaptive Bias Estimation for Federated Learning	98
7.1	Introduction	98
7.2	Problem Statement	100
7.3	Existing Solutions	100
7.4	Adaptive Bias Estimation	103
7.4.1	Setup	103
7.4.2	Method	103
7.4.3	Relation to FL Baselines	105
7.5	Experiments	109
7.5.1	Setup	109
7.5.2	Model Architecture	111
7.5.3	Baselines	111
7.5.4	Evaluation	111
7.6	Conclusions	112
Chapter 8	Conclusion and Future Research	115
8.1	Limitations and future research	117

Appendix A	Supplementary material for Chapter 2	119
A.1	Proofs	119
Appendix B	Supplementary material for Chapter 5	123
B.1	Proofs	123
B.2	Complementary Experiments	129
B.2.1	task adaptation Performance	129
B.2.2	domain adaptation Performance	131
Appendix C	Supplementary material for Chapter 6	145
C.1	Proofs	145
Appendix D	Supplementary material for Chapter 7	146
D.1	Proofs	146
D.2	Algorithm Details	149
D.2.1	Notation	149
D.2.2	Algorithmic Costs	149
D.2.3	Experiments Details	153
D.2.4	Stability and norm of parameters	154
D.2.5	Overfitting Analysis	155
D.2.6	Local regularization sensitivity	156
D.2.7	Discount factor sensitivity	157
Bibliography		159

List of Tables

2.1	Summary of variables, sets, and functions	13
2.2	Summary of operators' notation used in this thesis.	14
5.1	Initial performance improvement using ENTAME	54
5.2	Converged perf. of ENTAME and baselines on CIFAR-10 . . .	54
5.3	Converged perf. of ENTAME and baselines on CIFAR-100 . . .	55
5.4	Converged perf. of ENTAME and baselines on Caltech-101 . .	55
6.1	Convergence performance of FAST and baselines	92
7.1	Notation for FL.	103
7.2	Performance of AdaBest and baselines on various settings . . .	113
D.1	Summary of notion used to formulate the algorithm costs . . .	150
D.2	Comparing FL algorithms in compute cost of local ops.	150
D.3	Comparing FL algorithms in compute cost of global ops.	150

List of Figures

1.1	The concept of task adaptation	3
1.2	The concept of domain adaptation	3
1.3	The concept of parallel task adaptation	5
1.4	Basic components of a DNN	6
3.1	DTL and HTL.	24
3.2	Feature-extraction vs. feature-tuning	25
3.3	Feature-tuning after feature-extraction	27
3.4	Flowchart of iterative optimization for training a DNN	29
3.5	A round of communication hypotheses in LocalSGD	30
4.1	Transferring knowledge between Symbolic models and ANNs	33
4.2	Task decomposition	33
5.1	Norm of backpropagated gradient towards pretrained params	43
5.2	Conventional feature-tuning vs. ENTAME	45
5.3	ENTAME & baselines, ResNet-50 on CIFAR-10	52
5.4	ENTAME & baselines, ResNet-50 on CIFAR-100	52
5.5	ENTAME & baselines, ResNet-50 on Caltech-101	53
5.6	Frobenius norm of \mathbf{W} , Adam with $\eta = 10^{(-3)}$	56
5.7	Frobenius norm of \mathbf{W}	56
5.8	Frobenius norm of \mathbf{W} , Adam with $\eta = 10^{(-5)}$	57
5.9	Class-partition distribution with 2 partitions	58
5.10	The effect of task similarity on methods under the study	59
5.11	Class-partition distribution with 10 partitions	60
5.12	Test acc. on all data domains per episode using Xav	64

5.13	Test acc. on all data domains per episode using Xav + FN . . .	65
5.14	Test acc. on all data domains per episode using SW + FN . . .	66
5.15	Test acc. on all data domains per episode using SR + FN . . .	67
5.16	Mean test acc. of episodes of hypothesis transfer	68
6.1	Frobenius norm of \mathbf{W} with Gaussian distribution initialization	71
6.2	Conceptual comparison between conventional task adaptation methods	73
6.3	Conceptual comparison between feature-tuning with head warmup and FAST	73
6.4	Re-initializing head’s params and forgetting the same task . .	80
6.5	The development of $\log(\ W\ _F)$ during feature-tuning	80
6.6	Initializing head’s params and forgetting a different task . . .	81
6.7	The effect of preventing the loss landscape to deform in $\mathfrak{S}(\phi)$	82
6.8	Head warmup and minimum overshooting	84
6.9	Symmetric vs. asymmetrical scaling lr, ResNet-18	86
6.10	Symmetric vs. asymmetrical scaling lr, VGG-19	87
6.11	FAST vs. baseline optimized using mini-batch SGD	88
6.12	FAST vs. baseline optimized using Adam	89
6.13	FAST vs. baseline optimized using Rectified Adam (RAdam) .	95
6.14	Comparing FAST optimized different optimization algorithms	96
6.15	FAST vs. baselines convergence performance on CIFAR-100 .	97
6.16	FAST vs. baselines convergence performance on Caltech-256 .	97
6.17	FAST vs. ENTAME vs. baseline	97
7.1	Interest trend for “machine learning” and “statistical analysis”	99
7.2	Interest trend for “federated learning” and “distributed learning”	99
7.3	Asymptotic instability of FedDyn and the norm of parameters	102
7.4	Geometric interpretation of AdaBest’s correction on the server	104

7.5	AdaBest vs. baselines under balance and scale tests	112
B.1	Comparing ENTAME for ResNet-152, CIFAR-10	129
B.2	Comparing ENTAME for ResNet-152, CIFAR-100	130
B.3	Comparing ENTAME for ResNet-152, Caltech-101	130
B.4	Comparing ENTAME for DenseNet-121, CIFAR-10	131
B.5	Comparing ENTAME for DenseNet-121, CIFAR-100	131
B.6	Comparing ENTAME for DenseNet-121, Caltech-101	132
B.7	Comparing ENTAME for DenseNet-201, CIFAR-10	132
B.8	Comparing ENTAME for DenseNet-201, CIFAR-100	133
B.9	Comparing ENTAME for DenseNet-201, Caltech-101	133
B.10	Comparing ENTAME for VGG-16, CIFAR-10	134
B.11	Comparing ENTAME for VGG-16, CIFAR-100	134
B.12	Comparing ENTAME for VGG-16, Caltech-101	135
B.13	Comparing ENTAME for VGG-19, CIFAR-10	135
B.14	Comparing ENTAME for VGG-19, CIFAR-100	136
B.15	Comparing ENTAME for VGG-19, Caltech-101	136
B.16	Comparing ENTAME for Inception-V3, CIFAR-10	137
B.17	Comparing ENTAME for Inception-V3, CIFAR-100	137
B.18	Comparing ENTAME for Inception-V3, Caltech-101	138
B.19	Test acc. on all data domains per episode using Kin + FN	139
B.20	Test acc. on all data domains per episode using Kout + FN	140
B.21	Test acc. on all data domains per episode using SW	141
B.22	Test acc. on all data domains per episode using SR	142
B.23	Test acc. on all data domains per episode using Kin	143
B.24	Test acc. on all data domains per episode using Kout	144
D.1	Instability of FedDyn and norm of cloud parameters.	155

D.2	Test accuracy score of AdaBest and its baselines	156
D.3	Sensitivity to local regularization factor	157
D.4	Sensitivity to client rate and global regularization factor	158

Abstract

Small computing devices, such as smartphones, constantly collect and store data that can potentially help machines learn complex tasks. However, the data contained on a single device is relatively small and biased, making [Machine Learning](#) difficult. Although it is possible to transfer data from other devices, doing so poses storage, communication, and privacy concerns. [Hypothesis Transfer Learning \(HTL\)](#) provides a solution to this dilemma by transferring and importing knowledge learned from data on other devices in form of pretrained [Machine Learning](#) models (hypotheses). In its simplest form, a hypothesis can be transferred from one task or domain to another. Training a model on the target task can benefit from the knowledge embedded in the transferred hypothesis with no direct access to the source data. This often leads to significantly lower storage and communication costs as well as fewer privacy concerns. [HTL](#) can be extended to a chain of transfers and adaptations in the context of [Continual Learning](#). [Federated Learning \(FL\)](#) further extends this idea by coordinating concurrent transfers from multiple sources at each transfer iteration. It enables a large number of remote devices to train a model collaboratively without sharing their data. In this thesis, we study some challenges associated with transferring and adapting pretrained models in the context of [task adaptation](#) and [FL](#). We discuss an often-overlooked source of inefficiency in [feature-tuning](#), the most popular method of [task adaptation](#). Accordingly, we propose two novel methods for improving the efficiency of [feature-tuning](#) which work by gradually increasing the magnitude of updates to pretrained feature-extractors. In addition, we present a new algorithm that addresses the issue of *client drift* which is known to make existing [FL](#) algorithms sub-optimal in heterogeneous settings. The thesis includes several experiments with image classification benchmarks for each of the learning settings under the study. Our findings show that the methods we propose can significantly improve their baselines in terms of accuracy and efficiency. Although our methods provide practical improvements over existing baselines, there is still room for further improvement as well as better understanding of the underlying mechanisms, which we leave to future research.

List of Abbreviations

AdaBest *ADaptive Bias ESTimation for Federated Learning (AdaBest)* is an [FL](#) algorithm (introduced by us in this thesis). [vii](#), [ix](#), [x](#), [xi](#), [9](#), [11](#), [39](#), [102](#), [104](#), [105](#), [106](#), [107](#), [108](#), [109](#), [110](#), [111](#), [112](#), [113](#), [116](#), [117](#), [118](#), [147](#), [148](#), [149](#), [150](#), [151](#), [152](#), [154](#), [155](#), [156](#), [157](#)

Adagrad *Adaptive Subgradient (Adagrad)* is a gradient-based optimization algorithm that adaptively adjusts the step size at each dimension by dividing the corresponding gradient on a its running magnitude since the beginning of the optimization. [21](#)

AI *Artificial Intelligence (AI)* is the field of study that aims to imitate the behaviour of human or other living things. [31](#)

ANN An *Artificial Neural Network (ANN)* is a [Machine Learning](#) model built of interconnected nodes that collectively aim to approximate a function. [viii](#), [xiii](#), [xiv](#), [xvi](#), [xviii](#), [xix](#), [xxi](#), [xxiv](#)

CL *Continual Learning (CL)* is an [ML](#) concept where a model is continually trained on new tasks or domains and aims to perform well on all the tasks or domains exposed to the model. [xii](#), [2](#), [3](#), [4](#), [6](#), [35](#), [36](#), [62](#)

CNN A *Convolutional Neural Network (CNN)* is a [DNN](#) that uses parameteric layers applying (simplified) convoltion oprtation. [xviii](#), [xx](#), [xxi](#), [xxii](#), [xxiii](#), [xxiv](#), [10](#), [31](#), [32](#), [35](#), [59](#), [94](#)

CPU *Central Processing Unit (CPU)* is a a computer's component responsible for executing the instructions. . [37](#)

DBN A *Deep Belief Network (DBN)* is a class of generative [DNNs](#) traditionally trained a layer at a time. [xx](#), [31](#), [32](#)

DNN A *Deep Neural Network (DNN)* is an [ANN](#) with multiple layers. The interest in the depth stems from the learning ability of a model that applies several

transformations on the input. [viii](#), [xiii](#), [xvi](#), [xx](#), [xxiv](#), [5](#), [6](#), [7](#), [12](#), [13](#), [20](#), [25](#), [29](#), [30](#), [31](#), [34](#), [36](#), [43](#), [71](#), [115](#), [117](#)

DTL *Data Transfer Learning (DTL)* is a [TL](#) in which data is directly transferred in learning the target task(s). The transferred data is usually not a direct observation for the target task(s). [viii](#), [xxi](#), [xxiii](#), [23](#), [24](#)

ENTAME *Efficient Neural Task Adaptation via Maximum Entropy Initialization (ENTAME)* is a [task adaptation](#) algorithm (introduced by us in this thesis). [vii](#), [viii](#), [ix](#), [x](#), [9](#), [10](#), [40](#), [41](#), [45](#), [46](#), [47](#), [48](#), [49](#), [51](#), [53](#), [54](#), [55](#), [61](#), [63](#), [66](#), [67](#), [69](#), [70](#), [93](#), [97](#), [115](#), [116](#), [117](#), [125](#), [126](#), [127](#), [129](#)

FAST *Fast And Stable Task-adaptation (FAST)* is a [task adaptation](#) algorithm (introduced by us in this thesis). [vii](#), [ix](#), [9](#), [10](#), [11](#), [69](#), [70](#), [73](#), [75](#), [77](#), [78](#), [82](#), [86](#), [88](#), [89](#), [90](#), [91](#), [92](#), [93](#), [94](#), [95](#), [96](#), [97](#), [115](#), [116](#), [117](#)

FC *Fully Connected (FC)* or *Linear layer* or *Dense Layer* is a parameterized layer used in [ANNs](#) that operates on its input by applying a dot product with its parameters 2D matrix. The term “Fully” is an emphasize that all outputs are connected to all inputs. [6](#), [25](#), [26](#)

FedAvg *Federated Averaging (FedAvg)* or [LocalSGD](#) is an [FL](#) algorithm. [xv](#), [29](#), [38](#), [39](#), [100](#), [101](#), [109](#), [111](#), [112](#), [113](#), [116](#), [148](#), [149](#), [150](#), [153](#)

FedDyn *Federated Learning with Dynamic Regularization (FedDyn)* is an [FL](#) algorithm. [ix](#), [x](#), [39](#), [101](#), [102](#), [105](#), [106](#), [107](#), [108](#), [109](#), [110](#), [111](#), [112](#), [113](#), [116](#), [146](#), [148](#), [149](#), [150](#), [151](#), [152](#), [153](#), [154](#), [155](#), [156](#), [157](#)

FL *Federated Learning (FL)* is an [ML](#) concept where a group of devices collaborate to train a model by communicating parameters through a central hub which also coordinates the training. [vii](#), [xii](#), [xiii](#), [xiv](#), [xv](#), [xvii](#), [xviii](#), [xix](#), [xxi](#), [xxii](#), [xxiv](#), [4](#), [6](#), [8](#), [9](#), [10](#), [11](#), [19](#), [27](#), [28](#), [34](#), [37](#), [38](#), [39](#), [59](#), [98](#), [99](#), [100](#), [101](#), [102](#), [103](#), [105](#), [109](#), [115](#), [116](#), [117](#), [153](#), [154](#), [155](#)

- GD** *Gradient Descend (GD)* is an optimization algorithm that calculates the gradient of the loss function for the entire training set to iteratively adjust the model parameters towards the optimal values. [xxi](#), [10](#), [13](#), [15](#), [16](#), [17](#), [18](#), [20](#), [21](#), [75](#), [76](#), [119](#), [120](#), [121](#), [145](#)
- GPU** *Graphical Processing Unit (GPU)* is a type of processor with many simple processing cores. [50](#)
- HTL** *Hypothesis Transfer Learning (HTL)* is a [TL](#) that involves transferring knowledge through trained models or hypotheses for the source task(s). [viii](#), [xii](#), [xxi](#), [xxiii](#), [2](#), [6](#), [10](#), [22](#), [23](#), [24](#), [28](#), [33](#), [34](#), [35](#), [44](#), [62](#), [115](#)
- IID** A collection of random variables is *Independent and Identically Distributed (IID)* if they share a common probability distribution. [xix](#), [100](#), [110](#), [113](#), [117](#)
- ILSVRC** *ImageNet Large Scale Visual Recognition Challenge (ILSVRC)* is an image classification and localization challenge. [32](#)
- ITL** *Information Transfer Learning (ITL)* is a [TL](#) in which some information about some data or some information about hypotheses learned for a task defined on that data is transferred to help learning target task(s).. [24](#), [27](#)
- KL divergence** *Kullback–Leibler (KL) divergence* is an asymmetric distance metric to determine dissimilarity between two probability distributions. [14](#), [45](#)
- LocalSGD** *Local Stochastic Gradient Descent (LocalSGD)* or [FedAvg](#) is an [FL](#) algorithm. [viii](#), [xiv](#), [xvi](#), [xix](#), [10](#), [29](#), [30](#), [37](#), [38](#), [100](#), [101](#)
- mini-batch SGD** *Mini-batch Stochastic Gradient Descend (GD)* is a modified [SGD](#) in which the cost is still computed stochastically computed but for a randomly selected subset training instances. [ix](#), [10](#), [20](#), [21](#), [42](#), [46](#), [49](#), [70](#), [76](#), [77](#), [81](#), [83](#), [85](#), [86](#), [88](#), [90](#), [92](#), [94](#), [103](#), [127](#), [128](#)

ML *Machine Learning (ML)* is the process in which the machines learn to perform tasks from the data without explicit assertive rules. [xii](#), [xiii](#), [xiv](#), [xvii](#), [xviii](#), [xx](#), [xxii](#), [xxiii](#), [xxiv](#), [4](#), [6](#), [10](#), [22](#), [23](#), [26](#), [31](#), [35](#), [37](#), [98](#), [99](#)

RAdam *Rectified Adam (RAdam)* is a gradient-based optimization algorithm that improves the high variance initial steps of [Adam](#).. [ix](#), [21](#), [90](#), [91](#), [95](#)

RBM A *Restricted Boltzmann Machine (RBM)* is a class of generative [ANNs](#) that is able to learn a probability distribution over its inputs. [26](#), [31](#)

ReLU *Rectified Linear Unit (ReLU)* is an activation function typically used in [DNNs](#). As a function, its a unary operator that outputs zero if the input is negative, otherwise, outputs the input identically. [32](#)

RMSProp *Root Mean Square Propagation (RMSProp)* is a gradient-based optimization algorithm that adaptively adjusts the step size at each dimension by dividing the corresponding gradient on its running average.. [xviii](#), [21](#)

RProp *Resilience backpropagation (RProp)* is a gradient-based optimization algorithm that adaptively adjusts the step size at each dimension based on consecutive changes in the sign of the gradient for that dimension. It only works well with full-gradients algorithm. [20](#), [21](#)

RV-LSGD *Reduced Variance LocalSGD (RV-LSGD)* is adaptation of [RV-SGD](#) to [Local Stochastic Gradient Descent \(LocalSGD\)](#). [102](#), [107](#), [116](#)

RV-SGD *Reduced Variance Stochastic Gradient Descent (RV-SGD)* is a group of algorithms that aims to reduce the variance of gradients introduced by the stochasticity of [SGD](#). [xvi](#), [xvii](#), [17](#), [20](#), [39](#), [100](#), [101](#)

SAG *Stochastic Average Gradient (SAG)* is an [RV-SGD](#) algorithm. [17](#), [18](#)

SARAH *StochAstic Recursive grAdient algorithM (SARAH)* is an [RV-SGD](#) algorithm. [19](#)

SCAFFOLD *Stochastic Controlled Averaging for Federated Learning (SCAFFOLD)*

is an [FL](#) algorithm. [39](#), [101](#), [102](#), [105](#), [106](#), [108](#), [109](#), [111](#), [112](#), [113](#), [116](#), [148](#), [149](#), [150](#), [151](#), [152](#), [154](#), [155](#), [156](#)

SGD *Stochastic gradient descent (SGD)* is an optimization technique used in machine learning that calculates the gradient of the loss function for a randomly

selected instance from the training set to adjust the model parameters iteratively. [xv](#), [xvi](#), [10](#), [17](#), [18](#), [19](#), [20](#), [21](#), [37](#), [38](#), [44](#), [61](#), [63](#), [71](#), [75](#), [76](#), [78](#), [80](#), [81](#), [84](#), [88](#), [96](#), [97](#), [100](#), [120](#), [121](#), [128](#), [145](#), [153](#)

SVRG *Stochastic Variance Reduction Gradients (SVRG)* is an [RV-SGD](#) algorithm.

[18](#), [19](#), [20](#), [38](#), [39](#), [121](#)

TL *Transfer Learning (TL)* is a class of [ML](#) that involves transferring data, information or knowledge from some sources tasks towards learning some target ones

layers applying (simplified) convolution oprtation. [xiv](#), [xv](#), [2](#), [7](#), [10](#), [22](#), [27](#), [28](#), [32](#), [46](#), [72](#)

Glossary

***L*-smooth** A function is *L*-smooth if the slope of the straight line that connects any two points on the function curve is equal or smaller than *L*. 18

activation function A function that applies non-linearity to its inputs. 32

Adam An optimization algorithm that combine the [momentum](#) method with [RM-SPop](#). viii, ix, xvi, 21, 51, 55, 56, 57, 76, 77, 86, 89, 90, 91, 94, 96

admissible In optimization literature, an admissible set refers to a sub-set of possible solutions that includes acceptable solutions.. 119

AIDE An [FL](#) algorithm. 39

AlexNet A [CNN](#) architecture. 32

backpropagation A method that uses chain rule for gradients to find the gradient of a calculated loss (found in forward pass) with respect to all of its parameters (or if desired even the input) for a [ANN](#). xx, 16, 26, 35

Caltech-256 A labeled image dataset. ix, 92, 97

Caltech-101 A labeled image dataset. vii, viii, x, 50, 51, 53, 54, 55, 91, 130, 132, 133, 135, 136, 138

catastrophic forgetting Abrupt forgetting of the knowledge that an [ANN](#) has gained often triggered by acquiring new but knowledge. 36, 42, 69, 80, 84, 93, 94

centralized learning *Centralized learning* is a [ML](#) sub-category in which a learning machine has a direct local access to the training data because it is stored on the same device. In contrast, in [decentralized learning](#) is stored on remote devices. xix, 8, 28

CIFAR-100 A labeled image dataset. [vii](#), [viii](#), [ix](#), [x](#), [43](#), [50](#), [52](#), [54](#), [55](#), [58](#), [60](#), [63](#), [70](#), [79](#), [80](#), [81](#), [82](#), [85](#), [87](#), [91](#), [92](#), [93](#), [97](#), [109](#), [110](#), [111](#), [112](#), [113](#), [130](#), [131](#), [133](#), [134](#), [136](#), [137](#), [154](#), [158](#)

CIFAR-10 A labeled image dataset. [vii](#), [viii](#), [x](#), [50](#), [52](#), [54](#), [109](#), [110](#), [111](#), [113](#), [129](#), [131](#), [132](#), [134](#), [135](#), [137](#), [154](#)

classification The [supervised learning](#) task of categorizing data into classes. [6](#), [9](#), [13](#)

client drift The discrepancy in optimization trajectory between a model updated on the whole or an [IID](#) selection of data and a model trained with biased draws of data (e.g., in case of [LocalSGD](#)). [10](#)

client-server model A distributed application structure where nodes are assigned to be client or servers based on whether they request a service or provide it. [1](#), [98](#)

connectionism A school of thought that looks for answering certain unresolved questions in the field of cognitive science through connecting the concepts with [ANNs](#). [31](#)

cross-device FL [FL](#) setup with massive number of clients (100s to billions). [38](#), [39](#)

cross-silo FL [FL](#) setup in which the number of clients are small (typically between 2 to 100s). [37](#), [38](#)

DANE An [FL](#) algorithm.. [39](#)

data centralization Stacking data from different devices on one device and integrate in addition to prepare the stacked data (possibly for learning from). [1](#), [2](#), [23](#)

decentralized learning See [centralized learning](#). [xviii](#), [28](#), [36](#)

Deep Learning Methods that enable learning “representations of data with multiple levels of abstraction” for computational models with the help of [backpropagation](#). [9](#), [10](#), [35](#)

DenseNet A [CNN](#) architecture. [x](#), [50](#), [54](#), [55](#), [87](#), [88](#), [89](#), [95](#), [96](#), [131](#), [132](#), [133](#)

device In this study, a device is a physical computer with storage and network connection. [xx](#), [28](#)

discrete uniform distribution A distribution in which possible outcomes have equal probability. [17](#)

domain An input space and a marginal distribution defined over it. [10](#), [12](#), [41](#)

domain adaptation Adapting a model trained on one domain to learn another domain, given (usually) the same task defined over both domain. [viii](#), [2](#), [3](#), [4](#), [5](#), [35](#)

edge computing Distributed computing paradigm that emphasizes on moving computation from servers to [edge devices](#) or the leaf nodes in a network. [1](#), [98](#)

edge device A [device](#) at the edge of the Internet; very close to the end user nodes. [xx](#), [1](#), [2](#)

EMNIST A labeled image dataset. [108](#), [109](#), [110](#), [111](#), [113](#), [154](#), [156](#)

energy-based models Generative [ML](#) models in which a computed level of energy determines the probability of each model state. See [DBNs](#) as an exsmple. [31](#)

entropy A probabilistic measure of certainty in information theory. [47](#), [124](#), [125](#)

feature-extractor Part of a [Deep Neural Network \(DNN\)](#) that extracts features. see [Figure 1.4](#). [45](#)

feature-tuning A [task adaptation](#) method in which pretrained and randomly initialized parameters are all trained and tuned together. [viii](#), [ix](#), [xii](#), [25](#), [26](#), [27](#), [35](#), [36](#), [40](#), [41](#), [42](#), [43](#), [44](#), [45](#), [49](#), [52](#), [53](#), [54](#), [55](#), [57](#), [59](#), [61](#), [65](#), [66](#), [67](#), [69](#), [70](#),

72, 73, 74, 75, 76, 77, 78, 79, 80, 82, 83, 86, 87, 90, 91, 92, 94, 115, 116, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138

feature-extraction A [task adaptation](#) method in which pretrained parameters are frozen and only randomly initialized parameters are trained. [viii](#), [25](#), [26](#), [27](#), [35](#), [41](#), [43](#), [73](#), [74](#), [79](#), [115](#)

FedDANE An [FL](#) algorithm.. [39](#)

FedPD An [FL](#) algorithm.. [39](#)

FedProx An [FL](#) algorithm.. [38](#), [111](#)

FedSplit An [FL](#) algorithm.. [39](#)

feedforward A *feedforward* [ANN](#) is an [ANN](#) in which the connection do not form a loop. [35](#)

FSVRG An [FL](#) algorithm.. [39](#)

full gradient Gradient calculated based on full batch of data (all data). [GD](#) works with *full gradient*. [15](#)

hash table A data structure that implements mapping from keys to values. [25](#)

head See [Figure 1.4](#) for a visual explanation. [ix](#), [6](#), [7](#), [8](#), [10](#), [13](#), [25](#), [26](#), [27](#), [35](#), [40](#), [41](#), [42](#), [43](#), [45](#), [46](#), [47](#), [51](#), [55](#), [57](#), [61](#), [62](#), [65](#), [69](#), [70](#), [71](#), [72](#), [73](#), [75](#), [76](#), [77](#), [78](#), [79](#), [83](#), [84](#), [85](#), [92](#), [94](#), [115](#), [123](#), [126](#)

ImageNet A large-scale hierarchical image dataset. [43](#), [50](#), [52](#), [53](#), [78](#), [82](#), [85](#), [87](#), [91](#), [129](#), [130](#), [131](#), [132](#), [133](#), [134](#), [135](#), [136](#), [137](#), [138](#)

Inception A [CNN](#) architecture. [x](#), [50](#), [51](#), [54](#), [55](#), [137](#), [138](#)

instance weighting A [DTL](#) approach. [23](#)

knowledge distillation An [HTL](#) approach in which a pretrained model supervises another model that is under training. [38](#)

LeNet A [CNN](#) architecture. [31](#)

loss A value that is aimed to be minimized in an optimization problem. Also known as *cost* or *risk* . [xxii](#), [14](#), [15](#), [18](#), [42](#)

loss function A [scalar-valued function](#) that outputs a [loss](#) value. [14](#)

machine A program running on a device. [28](#)

MNIST A labeled image dataset. [50](#), [51](#), [54](#), [79](#), [81](#)

momentum It is a discount factor that is used to smooth out the values of a sequence based on the weighted average of the earlier values in that sequence. [xviii](#), [19](#), [21](#), [38](#), [76](#), [77](#), [78](#), [88](#), [90](#), [94](#), [96](#)

noise A random quantity is considered *noise* with regard to one or multiple [signals](#) if it is not objectively related to them. See Definition [6](#). [44](#), [45](#)

off-the-shelf model Pretrained [ML](#) models that are shared for [task adaptation](#). The core idea is to pretrain a model only once but tune it numerous times for different applications. [2](#), [9](#), [50](#), [82](#)

one-hot A sparse vector encoding in which only one elements of the vector is one and the rest are zero. The one-hot vector space for a vector spans over a number of points equal to the length of that vector. [13](#), [14](#), [123](#)

oracle dataset A dataset that includes all training examples. In context of [FL](#), it is a hypothetical set created by stacking up data of all the clients. [28](#)

paired t-test A statistics hypothesis test to determine if a statistics significantly follow a hypothesis or not. In the paired version of the t-test, the hypothesis is usually about determining if a pair of statistics are from the same distribution or not. [53](#)

prediction error The difference between true and predicted labels. See Definition [7](#). [44](#), [45](#), [76](#), [123](#)

pretrained model A model trained on an upstream machine and transferred to a downstream machine. [2](#), [7](#), [24](#), [41](#), [72](#)

probability simplex A vector of non-negative values that sums to one. [xxiv](#), [13](#), [47](#)

pseudo-gradient See Definition [3](#). [17](#), [76](#)

query-less HTL A sub-category of [HTL](#) in which the downstream does not make a specific request for the way upstream machine trains and prepares its hypotheses to transfer. [28](#), [29](#)

query-based HTL A sub-category of [HTL](#) in which the downstream machine requests for certain characteristics of the hypotheses to be learned on the upstream machine. [28](#), [29](#)

reinforcement learning A sub-category of [ML](#) in which machines learn from the outcomes of their interactions with the environment. [4](#), [5](#)

ResNet A [CNN](#) architecture. [viii](#), [ix](#), [x](#), [43](#), [50](#), [52](#), [53](#), [54](#), [55](#), [70](#), [78](#), [85](#), [86](#), [93](#), [129](#), [130](#)

ResNeXt A [CNN](#) architecture. [91](#)

RGB A color model in which each color is made up of a mixture of three main colors: red, green, and blue. In digital imaging, RGB format consists of three channels each of which representing the intensity of the pixels in one of the main colors.. [50](#), [92](#)

scalar-valued function A function with one dimensional output scalar. [xxii](#), [14](#)

self-taught learning A form of [DTL](#) in that there is unlabeled supplementary data available but if labeled, the labels would be different from the ones for the objective task. [23](#), [24](#)

semi-supervised learning A sub-category of [ML](#) in which machines learn from a dataset in which only a portion of instances are labeled. [23](#), [24](#)

sigmoid A mathematical function sometimes used as activation function in [DNNs](#). [32](#), [34](#)

signal A quantity that withholds useful information. See Definition [5](#). [xxii](#), [44](#)

softmax A function that normalizes its inputs to output a [probability simplex](#). See Definition [4](#). [13](#), [41](#), [42](#), [45](#), [75](#), [127](#)

supervised learning A sub-category of [ML](#) in which machines learn from labeled data. [xix](#), [4](#), [5](#), [6](#), [10](#), [12](#), [32](#), [34](#)

Symbolic Machine Learning A type of [ML](#) based on human readable representations. [33](#)

Tanh A mathematical function sometimes used as an activation function in [DNNs](#). [32](#), [34](#)

task An output space and a desired mapping from an input space to the output space. [10](#), [12](#), [41](#)

task decomposition An [ANN](#) learning technique in which a task is decomposed into sub-tasks for which finding solution is easier. [viii](#), [33](#), [34](#)

task adaptation Adapting a model trained on one task to learn another task. [viii](#), [xii](#), [xiv](#), [xx](#), [xxi](#), [xxii](#), [2](#), [3](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [24](#), [25](#), [26](#), [35](#), [41](#), [45](#), [62](#), [65](#), [66](#), [71](#), [73](#), [74](#), [78](#), [94](#), [115](#), [117](#)

unsupervised learning A sub-category of [ML](#) in which machines learn from unlabeled data. [4](#)

VGG A [CNN](#) architecture. [ix](#), [x](#), [50](#), [54](#), [55](#), [85](#), [87](#), [88](#), [89](#), [95](#), [96](#), [134](#), [135](#), [136](#)

VRL-SGD An [FL](#) algorithm.. [39](#)

z-normalization A type of normalization applied on each member of a set of numbers that makes the set of results have mean zero and standard deviation one.
[45](#), [46](#)

Acknowledgements

I wish to express my deepest gratitude to my PhD supervisor , Stan Matwin, for his unwavering support, guidance and encouragement throughout my research journey. His expertise and insights have been invaluable to me, and I am grateful for the time and effort he invested in my research. I would also like to extend my appreciation to Thomas Trappenberg for providing valuable advice in the latter stages of my thesis. His insights and support truly helped me refine my work. I'm also grateful to my committee members for their guidance, feedback, and support throughout my research, and to the external examiner for accepting the role. I especially want to thank Lisa Di Jorio for supporting my thesis as supervisor during my internships at Imagia, and Marzie Saghayi, Mohammad Havaei, Laya Rafiee Sevyeri, Sharut Gupta, Lucas May Petry, Xiang Jinag and William Taylor-Melanson for their help and support during my research. I acknowledge and appreciate the assistance provided by Adam Auch in reviewing and making linguistic corrections to this thesis. This research has been made possible thanks to the funding and support provided by Imagia, Mitacs, and Research Nova Scotia. Finally, I would like to express my appreciation to my family, friends, and colleagues for their encouragement, support, and understanding throughout this journey. Their unwavering love and support has been essential in helping me achieve my research goals.

Chapter 1

Introduction

1.1 Motivation

In 1997, *Deep Blue*, a supercomputer taking up two cubic meters of space, defeated Grand Master Garry Kasparov, the world chess champion at the time [Kasparov, 2021]. Today, we can install chess engines that are far more powerful than Deep Blue on our palm-sized smartphones [Sedice, 2020]. These small devices have not only transformed the world of entertainment but also a plethora of other industries. These devices have made human life more convenient and efficient, enabling us to be more productive and informed than ever before. As a result, there has been a natural shift in data collection and analysis from a small number of large devices to a large number of small ones.

Consequently, the number of data collection devices has increased to the point where network bandwidth has become a bottleneck for numerous remote applications [Hecht et al., 2016]. These applications are typically built using the [client-server model](#), where clients request services from servers, which in turn process and reply the requests. This has inspired a new paradigm in which the data collection devices that interact with clients, perform as much required computation as possible locally (in contrast with doing it mostly on the server) in order to reduce the amount of communication. This paradigm is sometimes referred to as [edge computing](#) because the majority of computation is performed by network nodes at the Internet's edge¹.

The data on [edge devices](#) can give machines the opportunity to learn and perform complex tasks. However, using this data effectively can be challenging. A major issue is that the data stored on individual [edge devices](#) is often limited in quantity and may be biased. This can make it difficult for machines to learn from this data alone. In order to learn complex tasks, machines typically need a large amount of unbiased data [Roh et al., 2019]. One solution is [data centralization](#) which is to simply share

¹These devices are much closer to end users than they are to service providers.

the data stored on [edge devices](#) (also known as supplementary data) with the target learning machine. This can supplement the machine’s own data and provide it with a more diverse and comprehensive dataset to learn from.

[Data centralization](#) can be difficult or even impossible due to privacy, storage and communication concerns. Moreover, even if transferred, the supplementary data may provide a different input-output mapping (task) than the one the target machine aims to learn. One way to deal with these difficulties is through [Transfer Learning \(TL\)](#).

[TL](#) alleviates the aforementioned difficulties by encoding the knowledge of the supplementary data into a more portable form. This knowledge can then be transferred and eventually be incorporated into the target learning process. Perhaps the most intuitive form of encoded knowledge is a trained model itself. A model can be trained on the supplementary data and then be transferred to the target machine. This sub-category of [TL](#) is referred to as [Hypothesis Transfer Learning \(HTL\)](#) because the transferred models are hypotheses learned for solving learning tasks on the source machines. The shared hypotheses are sometimes referred to as [pretrained models](#).

Hypothesis sharing can better protect the privacy of the data compared to [data centralization](#), especially when the supplementary data is stored on personal remote devices. Moreover, the re-usability of the [pretrained models](#) is computationally much more appealing. The same encoded knowledge can be transferred to many devices to help them to learn new tasks. Widely shared and used [pretrained models](#) are sometimes called [off-the-shelf models](#).

On the target machine, a [pretrained model](#) can be directly adapted to perform a different task than the one it is pretrained on, or it can be tuned for a different data domain—data domain is simply characterized by the input space and its distribution (see domain definition in Section 2.1). The former is referred to as [task adaptation](#) while the latter is known as [domain adaptation](#). These concepts are respectively shown in Figures 1.1 and 1.2. In these figures, the gray, yellow and orange trapezoids represent randomly initialized, pretrained and adapted (tuned) models, respectively. The solid arrows show training while the dotted elbowed ones express hypothesis transfer.

A generalization of these hypothesis adaptation concepts, sometimes known as [Continual Learning \(CL\)](#) is to train a model to perform multiple tasks or perform

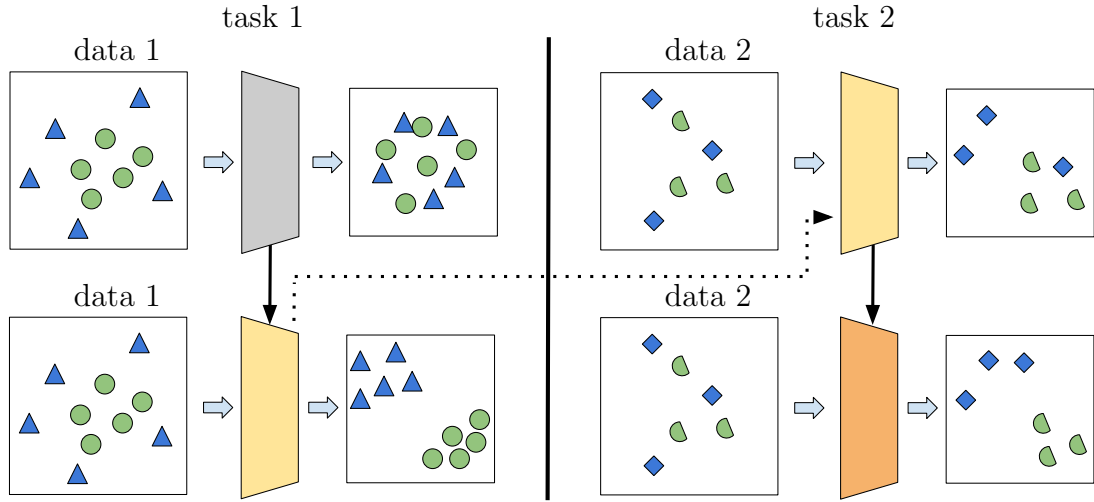


Figure 1.1: The concept of **task adaptation**. Task 1 (on the left) is to classify between triangles and circles while task 2 (on the right) is to classify between diamonds and semi-circles. Solid and dotted arrows represent training and hypothesis transfer, respectively.

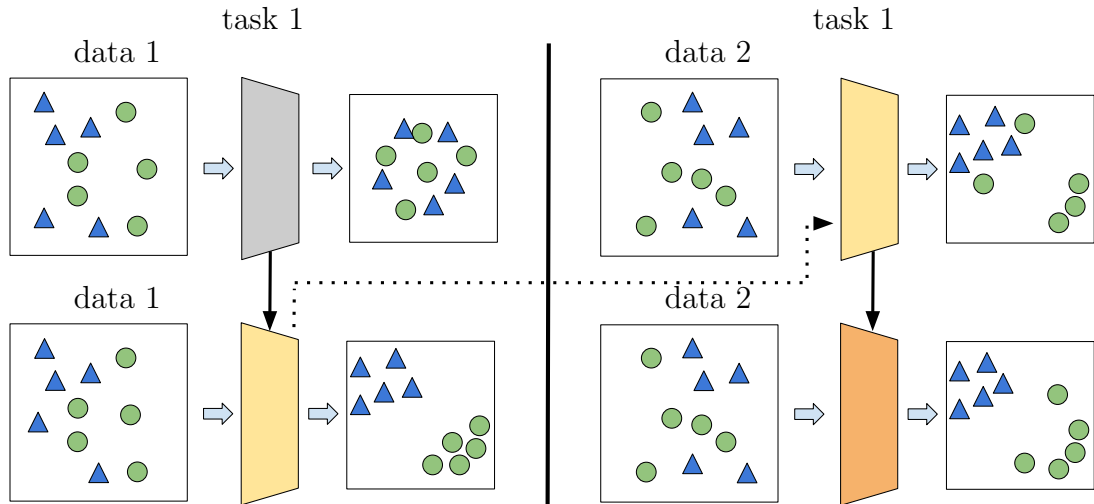


Figure 1.2: The concept of **domain adaptation**. The task (task 1) is to classify between triangles and circles. Data 1 (on the left) has a different distribution than that of data 2 (on the right). Solid and dotted arrows represent training and hypothesis transfer, respectively.

a single task but on multiple data domains using a chain of hypothesis transfers. Accordingly, such generalized **task adaptation** and **domain adaptation** are referred to as task incremental and domain incremental **CL**, respectively². From the learning

²According to [van de Ven and Tolias, 2019], there is a third category for **CL** referred to as class-incremental which we consider as a sub-category as task-incremental.

perspective, the domain incremental **CL** can be seen as a task virtually defined on centralized data, except that the data is partitioned into subsets that can only be accessed one partition at a time.

A new paradigm has emerged that further generalizes the domain incremental **CL** by permitting simultaneous access to more than one remote partition of the data; this is motivated by the proliferation of data collection/storage devices, their increasing compute capabilities, and the need for parallelization. In this paradigm known as **Federated Learning (FL)**, at each iteration of hypothesis transfer, a server sends a model to a random selection of the remote devices. After training the local copies of the model, selected devices send them back to the server where they are aggregated into a model that can be sent to the next set of selected devices.

Figure 1.3 reflects the idea of paralleled **domain adaptation**. It also depicts one iteration of hypothesis transfer in **FL** known as “a round of communication”³. In this iteration, two devices are selected, one hosting a data partition tagged with “data 1” and the other one hosting a data partition tagged with “data 2”.

In contrast to the sequential domain incremental **CL**—for which an iteration is shown in Figure 1.2, **FL** requires device selection, hypothesis communication and hypothesis aggregation. These operations are orchestrated by a separate device called the *server*. Accordingly, the devices where the models are sent for local training are sometimes called *clients*.

1.2 Background

An **ML** task is often modeled by approximating the mapping from the inputs to the expected outputs or measures given some observation of such association (think of these observations as experience). In the field of **Machine Learning (ML)**, the observations for such approximations are called training data⁴.

Generally, there are three different types of tasks in **ML**. They are **supervised learning**, **unsupervised learning**, and **reinforcement learning**. In **supervised learning**, examples of input-output mapping are given to the learning machine. In contrast, **unsupervised learning** is to learn from unlabeled data, meaning that examples of

³Sometimes, we simply refer to it as a *round*.

⁴“Observation” is a more common term in Estimation Theory; in this thesis, we use observations and training data interchangeably

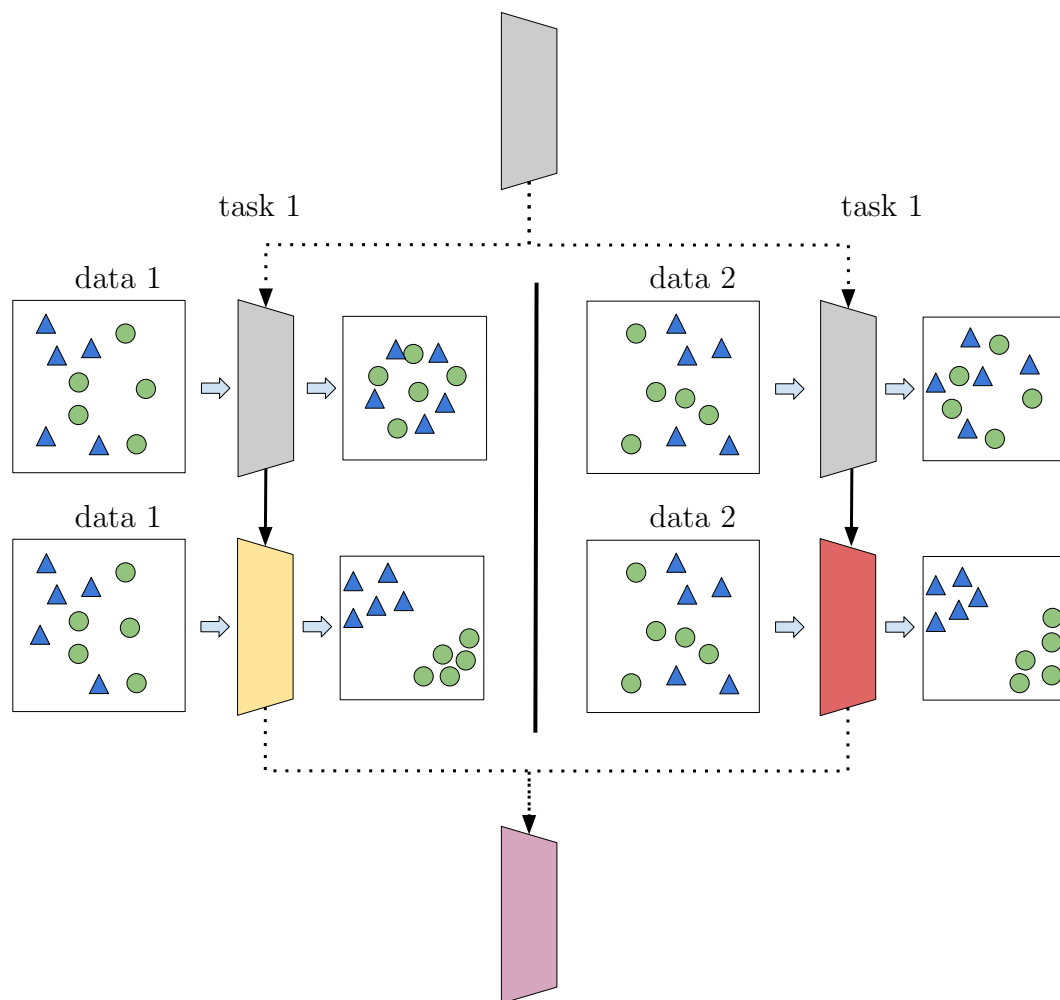


Figure 1.3: The concept of parallel [domain adaptation](#). The task (task 1) is to classify between triangles and circles. Data 1 (on the left) has a different distribution than that of data 2 (on the right). Solid and dotted arrows represent training and hypothesis transfer, respectively. The hypothesis transfer is done in parallel orchestrated by a server.

expected outputs are not provided. [Reinforcement learning](#) takes a different approach from these two in that the machines learn through interacting with an environment and receiving rewards and penalties for the actions that they take. Although some of our research questions and answers in this thesis can be applied to all of these kinds of tasks, we restrain our focus to [supervised learning](#).

The objective mapping or function from inputs to outputs can be modeled in various ways. In particular, [Deep Neural Networks \(DNNs\)](#) provide function approximators that have demonstrated a high capacity for learning complex tasks and have

emerged as the [ML](#) community’s first contender for solving a wide range of problems.

[DNNs](#) apply a series of non-linear operations on the input data so it is transformed into a space where performing the task is much simpler. We refer to such a space as the feature space and the inputs transformed into it as the extracted features. These features can then be linearly mapped to the output space where the model’s approximations (predictions) are represented. The former non-linear transformation is often referred to as feature-extractor while the latter is called the [head](#). A [Fully Connected \(FC\)](#) layer—which applies a matrix multiplication on its input—is often chosen as the [head](#). Components of a typical [DNN](#) are shown in Figure 1.4 for a model with six outputs.

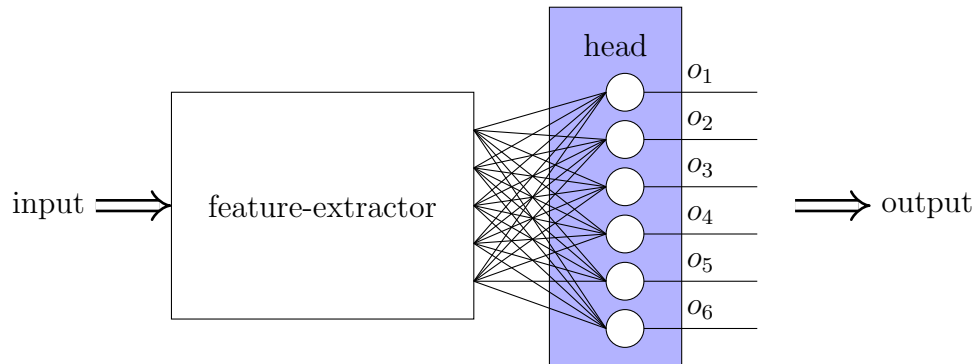


Figure 1.4: Basic components of a [DNN](#) consisting of a feature-extractor (the white colored box) and a [head](#) (the blue colored box). The [head](#) is considered to be an [FC](#) layer.

The operations that a [DNN](#) applies on the inputs to predict the outputs are parameterized. Training a [DNN](#) for a [supervised learning](#) task is equivalent to finding a set of such parameters that minimizes the prediction error. It is often done by iteratively modifying the parameters using gradient based optimization algorithms. Chapter 2 gives an introduction to these algorithms.

1.3 Research Objectives

This thesis is about the efficacy and efficiency of [HTL](#). We aim to resolve a number of existing problems in [task adaptation](#), domain incremental [CL](#), and [FL](#). Some of the questions asked and solutions provided are applicable to various types of tasks. However, we focus on [classification](#) tasks in which the goal is to categorize each input

instance into one of a pre-defined set of categories (classes).

Throughout this thesis, we prioritize the practical applications of our results rather than limiting ourselves to abstract concepts. We strive to maintain an experimental-oriented mindset rather than a merely theoretical one. Everywhere the application of the newly introduced methods is illustrated, we consistently concentrate on straightforward image classification task, so that the results can be easily interpreted.

1.3.1 Task Adaptation

The most common way of adapting a pretrained [DNN](#) to a new task and/or domain is by simply using the [pretrained model](#) as the initial state in training process of the target learning machine. However, the target learning task dictates the dimensions of the model’s output. Therefore, a [pretrained model](#) needs to be modified for [task adaptation](#). The most common practice is to simply replace the pretrained model’s [head](#) with one that matches the target task’s output.

The model’s [head](#), if replaced, is often initialized at random⁵ with no apparent recipe [[He et al., 2016](#), [Huang et al., 2017](#), [Ma et al., 2018](#), [Sandler et al., 2018](#), [Xie et al., 2017](#), [Tan and Le, 2019](#)]. This randomness can lead to large and noisy initial updates to the feature-extractor. These updates easily distort the pretrained features [[Varno et al., 2020](#), [Kumar et al., 2021](#)], thus, losing useful transferred knowledge. In turn, this can lead to slow convergence and so to undermine the merits of [TL](#).

Existing solutions include slowing down the learning process and including an extra warmup training phase in which only the [head](#) is updated [[Li and Hoiem, 2017](#), [Kumar et al., 2021](#)]. The former is in contrast to a major goal of [TL](#) which is to accelerate the learning process. The latter requires an extra validation set which is not always an option given that [TL](#) is usually motivated by the small size of datasets on the target machines (from which sometimes is hard to separate a statistically meaningful subset as a validation dataset).

In this thesis, we propose to start the training on the target task (tuning) with a gradual increase in the magnitude of updates to the feature-extractor starting from zero. Such a scheme avoids large noisy updates at the beginning of the training, yet lets the feature-extractor to eventually be adapted to the target task. As it leads to

⁵In this context, by [head](#) we virtually refer to the “parameters” of the model’s [head](#).

preserving the transferred knowledge from random perturbation at the beginning, we hypothesize that the efficiency of [task adaptation](#) can be improved.

To achieve gradual increase in the feature-extractor updates, we propose to initialize the replaced [head](#) with zero weights and biases. Regardless of the input, such initialization leads to same prediction for all classes at the beginning; therefore, we refer to it as the maximum entropy initialization.

Initializing parameters of the replaced [head](#) with zeros poses a new challenge which is to make sure that feature-extractor parameters are eventually adapted with an adequate rate. That is to make sure the gradual increase in the magnitude of the updates happens as desired. We overcome this challenge via two novel approaches. First, by normalizing extracted features and second, by adjusting [head](#)'s parameters with a different rate than those of the feature-extractor. These approaches are detailed in [Chapters 5 and 6](#), respectively.

1.3.2 Federated Learning

[FL](#) is a promising alternative to [centralized learning](#). It consumes reasonable amount of communication bandwidth and to some extent can preserve the privacy of the data/device owners. However, [FL](#) optimization is challenging especially when there is a large distribution mismatch among clients' data.

Heterogeneity among data of different clients can cause the local optimization on clients to “drift” away from the global objective—which is to learn from data of all clients as if they were combined. In order to estimate and therefore remove this drift, variance reduction techniques have been recently incorporated into [FL](#) optimization as in [Karimireddy et al. \[2020\]](#) and [Acar et al. \[2020\]](#). However, these approaches inaccurately estimate the clients' drift and ultimately fail to remove it properly.

In [Chapter 7](#), we propose an adaptive algorithm that accurately estimates drift across clients. In comparison to previous work, our approach necessitates less or equal storage and communication bandwidth, as well as lower computation costs. Additionally, our proposed methodology induces stability by constraining the norm of estimates for client drift, making it more practical for large scale [FL](#). Experimental findings demonstrate that the proposed algorithm converges significantly faster and achieves higher accuracy than the baselines across various [FL](#) benchmarks.

1.4 Contributions

The outcome of this research is a collection of methods to improve the efficiency and efficacy of hypotheses adaptation and [Federated Learning \(FL\)](#). The contributions of this thesis are⁶:

- We empirically investigate the speed of convergence of applying [task adaptation](#) using different existing initialization methods. This is done across an extensive set of [off-the-shelf models](#) tuned on a number of well-known image [classification](#) tasks.
- We introduce two novel methods for [task adaptation](#) which cause a gradual increase in the magnitude of the updates on the feature extractor. These methods which we refer to as [Efficient Neural Task Adaptation via Maximum Entropy Initialization \(ENTAME\)](#) and [Fast And Stable Task-adaptation \(FAST\)](#), help to avoid the initially large and noisy updates on pretrained parameters, so compared to the conventional practice they better maintain the transferred knowledge.
- We empirically show that [ENTAME](#) and [FAST](#) not only significantly accelerate the convergence on almost all the benchmarks but also eventually lead to a better performance on many of them.
- We propose [ADAPtive Bias ESTimation for Federated Learning \(AdaBest\)](#), the state-of-the-art in [FL](#) optimization.
- We empirically compare [AdaBest](#) and the existing baselines on several [FL](#) settings with different levels of data heterogeneity, and sample balance for clients.

1.5 Outline

Chapter 2 In this chapter, the notation used in this thesis is established. Some of the most prominent optimization algorithm for [Deep Learning](#) are overviewed. The

⁶Parts of this thesis is published in our research articles [Varno et al. \[2020\]](#) and [Varno et al. \[2022a\]](#), world-wide patent [Varno et al. \[2022b\]](#)

concepts such as [domain](#), [task](#), [supervised learning](#), convergence, and [Gradient Descent](#) (GD) algorithm are defined. We detail [Stochastic Gradient Descent](#) (SGD) and [mini-batch Stochastic Gradient Descent](#) ([mini-batch SGD](#)) algorithms, and enumerate several existing attempts to reducing their variance. [Appendix A](#) contains several proofs for theorems and lemmas included in [Chapter 2](#).

Chapter 3 This chapter gives background information on [TL](#). It elaborates on different categories of [TL](#) based on the transferred content. It also pin-points the conventional [task adaptation](#) approaches and [head](#) initialization issues. Furthermore, [FL](#) is described as a query-based [HTL](#) so it is better contrasted with the typical hypothesis adaptation approaches. Finally, in this chapter, [LocalSGD](#), as a seminal [FL](#) algorithm, is introduced.

Chapter 4 In this chapter we go over the related work. We highlight the research work leading to what is known as [Deep Learning](#) with an emphasize on [Convolutional Neural Networks](#) (CNNs). Additionally, we investigate the first attempts to [HTL](#) and then more precisely to [task adaptation](#). This includes unsupervised and supervised pretraining approaches especially for [CNNs](#). An overview of primitive research work in [Distributed ML](#) is also covered in this chapter. It is followed by an overview of [FL](#) algorithms, when they started to appear in [Deep Learning](#) literature, their categorizations and attempts to solve the well-known [client drift](#) issue.

Chapter 5 In this chapter, we propose [Efficient Neural Task Adaptation via Maximum Entropy Initialization](#) ([ENTAME](#)) to improve the efficiency of knowledge transfer in [task adaptation](#). We show how it makes updates on the pretrained parameters initially small and eventually larger. The chapter also includes several experiments to compare performance of [ENTAME](#) with the baselines. Proofs and additional experiments for this chapter are extended in [Appendix B](#).

Chapter 6 In this chapter, we propose [FAST](#) which shows similar learning acceleration as of [ENTAME](#) but without its feature normalization operation. Additionally, we examine [FAST](#) in preserving the source task’s knowledge and show that it forgets

much slower than the conventional [task adaptation](#). An extensive set of experiments is conducted to compare [FAST](#) with the baselines from various perspectives.

Chapter 7 In this chapter, we present a novel [FL](#) algorithm, [ADaptive Bias ESTimation for Federated Learning \(AdaBest\)](#). We further illustrate how it addresses the stability issue of the existing solutions. Finally, we empirically show that [AdaBest](#) outperforms the baselines by a high margin across several benchmarks.

Chapter 8 This chapter summarizes our findings and offers concluding remarks. We also outline our plans for future research, which are inspired by the work presented in this thesis.

Chapter 2

Background to Deep Learning

*If learning is what you would hold most dear,
With wisdom you will stride the turning sphere.*

- Ferdowsi, *Shahname*

In supervised learning, training a **DNN** involves finding the set of parameters that result in low prediction error on the training data. This search can be challenging because **DNNs** often have a large number of parameters, far more than the number of training examples. Due to the overparameterized nature of **DNNs**, a random search or even Bayesian optimization with random samples may be computationally infeasible. Gradient-based algorithms, on the other hand, can effectively navigate a large parameter space by marching in the direction that most reduces the error at each step. As a result, they can often find a better solution in a much shorter amount of time. In this chapter, we will first introduce some key concepts and notations that will be used to express the problems and ideas in the rest of this thesis; then, we will look at the most popular gradient-based algorithms, how they are used to train **DNNs**, and the mathematics behind them.

2.1 Supervised Deep Learning

Let characterize each dataset by a **domain** which consists of the raw data space \mathcal{X} and the marginal probability distribution $P(X)$ where $X \in \mathcal{X}$. Furthermore, let define a **task** $\mathcal{T} = \{\mathcal{Y}, f(\cdot)\}$ in relation with a domain $\mathcal{D} = \{\mathcal{X}, P(X)\}$ where \mathcal{Y} is the output space and $f : \mathcal{X} \rightarrow \mathcal{Y}$ is the true function which we aim to approximate using a **DNN**. In this thesis, we focus on **supervised learning** in which expected outputs for each training data instance are given. In other words, training data includes pairs of $(\mathbf{x} \in \mathcal{X}, \mathbf{y} \in \mathcal{Y})$ such that $\mathbf{y} = f(\mathbf{x})$. An iterative optimization algorithm

Table 2.1: Summary of variables, sets, and functions used in this thesis.

†: $\mathfrak{S}(\mathbf{v})$ gives the Euclidean space that has $|\mathbf{v}|$ dimensions.

$\mathcal{X}, \mathcal{A}, \mathcal{Y}$	{ input, feature, label } space
f, \hat{f}	{ true, approximate } objective function
Ω, \mathcal{H}	{ feature-extractor, head } function
\mathfrak{S}	Euclidean space function†
\mathcal{D}, \mathcal{T}	domain, task
ℓ, \mathcal{L}	{ instance, average } loss
$\mathbf{x}, \mathbf{o}, \mathbf{y}$	sample { input, output, predicted label }
\mathbf{a}, \mathcal{A}	sample { extracted, normalized } features
$\mathcal{G}, \mathbf{g}, \hat{\mathbf{g}}$	{ full-batch, mini-batch, pseudo } gradient
\mathbf{h}	gradient estimate
$\boldsymbol{\theta}, \boldsymbol{\phi}, \mathbf{W}$	{ model, feature-extractor } parameters
\mathbf{W}, \mathbf{b}	head’s { weights, biases }
$\mathcal{U}, \mathcal{N}, P$	{ uniform, normal, probability } distribution
η	learning rate (step size)
ϵ	small scalar
M, C, Q	number of { samples, classes, extracted features }
β	discount factor
\mathcal{S}, \mathcal{P}	set of { all, mini-batch } indices

(such as [Gradient Descend](#)) uses the labels to update the parameters of a [DNN](#) as an approximator for $f(\cdot)$.

We represent [DNN](#) parameterized by $\boldsymbol{\theta}$ with $\hat{f}(\cdot; \boldsymbol{\theta})$. In particular, for [classification](#) tasks, one can use [one-hot](#) encoding to represent the true labels. For example, suppose we have a categorical variable with three categories: “cat”, “dog”, and “rabbit”. We can use one-hot encoding to represent these variable as follows:

- “cat” would be represented as $[1, 0, 0]$,
- “dog” would be represented as $[0, 1, 0]$,
- “rabbit” would be represented as $[0, 0, 1]$.

This is the labeling convention we use in this thesis. Furthermore, we assume that \hat{f} outputs a [probability simplex](#) of the predicted class probabilities. This is often done by applying a [softmax](#) layer on top of the model’s [head](#) shown in [Figure 1.4](#). Note that a [probability simplex](#) is a vector of non-negative values that sums to one.

Table 2.2: Summary of operators' notation used in this thesis.

$u^t, \mathbf{v}^t, \mathbf{M}^t$	a { scalar, vector, matrix } at iteration t
$u^{(n)}, (\mathbf{v})^{\text{Tr}}$	power, Transpose
$ \mathbf{v} , \ \mathbf{v}\ , \ \mathbf{M}\ _F$	cardinality, 2-norm, Frobenius norm
$\langle \mathbf{v}, \tilde{\mathbf{v}} \rangle, \angle(\mathbf{v}, \tilde{\mathbf{v}})$	inner product, angle
\gg	much larger
$\frac{\partial v}{\partial \tilde{v}}, \nabla u$	partial derivative, gradient
$\mathbf{0}, \mathbf{1}$	{ all-zero, all-one } vector
$u\mathbf{1}, \mathbf{v}\mathbf{1}$	{ all-u, sum of } vector
\in, \cup	set { membership, union }
$\rightarrow, \Rightarrow, \mapsto$	space mapping, implication, transfer
\sum	summation
\triangleq	equality by definition
$\frac{\partial}{\partial \mathbf{v}}$	gradient with respect to \mathbf{v}
\mathbb{E}	expected value
$\mathbb{E}_{\text{batch}}$	arithmetic mean across batch
\sim	sampled from distribution
\ln	natural logarithm function
softmax	softmax probability estimator function
exp	exponential function

In such a case, for an input-label pair (\mathbf{x}, \mathbf{y}) the cost or **loss** of the approximation can be found using the negative log-likelihood of the predicted labels $\hat{\mathbf{y}} = \hat{f}(\mathbf{x}; \boldsymbol{\theta})$ and the true labels \mathbf{y} . This negative log-likelihood is mathematically equal to the **Kullback–Leibler divergence** (**KL divergence**), i.e.,

$$\ell(\mathbf{x}, \mathbf{y}; \boldsymbol{\theta}) = -\langle \mathbf{y}, \ln \hat{\mathbf{y}} \rangle; \quad (2.1)$$

and, as previously stated, \mathbf{y} is **one-hot** encoded. A **scalar-valued function** that outputs a **loss**, such as $\ell(\mathbf{x}, \mathbf{y}; \boldsymbol{\theta})$ is called a **loss function**. \ln and $\langle \cdot, \cdot \rangle$ represent the natural logarithm—operating elements-wise on its input vector in this case—and inner product, respectively. Note that a transpose operation is hidden in $\langle \cdot, \cdot \rangle$. For instance, $\langle \mathbf{y}, \ln \hat{\mathbf{y}} \rangle$ is equivalent to $\mathbf{y} (\ln \hat{\mathbf{y}})^{\text{Tr}}$; where \mathbf{y} is a row vector and $(\ln \hat{\mathbf{y}})^{\text{Tr}}$ is the transpose of $\ln \hat{\mathbf{y}}$. A list of symbols used in this thesis is presented in Tables 2.2 and 2.1.

For a training dataset with m data instances $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^m$, the average loss for the

whole training dataset is obtained as

$$\mathcal{L}(\boldsymbol{\theta}) \triangleq \frac{1}{m} \sum_{i=1}^m \ell(\mathbf{x}_i, \mathbf{y}_i; \boldsymbol{\theta}). \quad (2.2)$$

For the sake of simplicity, we sometimes omit the input arguments of the loss functions in our formulations. In this thesis, we assume that the datasets are static in the sense that over the time, the data distribution does not evolved or shifted. In other words, all pairs of data in Equation (2.2) are observations of the same input-output joint distribution, even though they may have been collected at different times.

2.2 Gradient Descent

Gradient Descent (GD)—dating back to [Cauchy et al. \[1847\]](#)—is a gradient-based first-order optimization algorithm that iteratively updates parameters in the direction of reducing the **loss**. To explain how **GD** uses gradient to update the parameters, we first introduce some related concepts and rationals as follows.

Let \mathcal{G} be the **full gradient** of the average loss with respect to the parameters. That is

$$\mathcal{G} \triangleq \nabla^{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^m \ell(\mathbf{x}_i, \mathbf{y}_i; \boldsymbol{\theta}) \right); \quad (2.3)$$

where $\nabla^{\boldsymbol{\theta}}$ is the gradient operator with respect to parameters $\boldsymbol{\theta}$. For the sake of the simplicity, wherever we deal with only one average **loss** and only one set of parameters, we refer to “the gradient of average loss with respect to the parameters” simply as the “gradient”.

GD updates the parameters towards the steepest direction (in the parameter space) of decreasing the **loss** or the steepest descent direction in the loss landscape (see Definition 1). Based on Theorem 1, at each parameters state, this direction is the negative of the gradient of loss with respect to the parameters, at that state. Let $\boldsymbol{\theta}^{t-1}$ and \mathcal{G}^{t-1} be the state of $\boldsymbol{\theta}$ and \mathcal{G} at iteration $t - 1$ of the optimization algorithm. **GD** finds the state of the parameters at iteration t as

$$\boldsymbol{\theta}^t = \boldsymbol{\theta}^{t-1} - \eta \mathcal{G}^{t-1}; \quad (2.4)$$

where η is the step size which is also called the learning rate.

Definition 1 (Loss landscape) *The loss landscape is a space with one more dimension than the parameter space to representing the loss, corresponding to a determined set of parameters and data instances. For each point in the parameter space, the data instances result in a loss that is reflected in the loss dimension.*

Theorem 1 (Steepest descent direction, proof in Appendix A) *For the differentiable loss function $\mathcal{L}(\boldsymbol{\theta})$ the negative of the gradient is the steepest direction of descend in the loss landscape.*

There are various criteria for stopping the iterations of gradient-based algorithms (end of training). In this work, we use convergence as defined in Definition 2 as the stopping criteria, unless otherwise stated. Accordingly, the optimization steps are stopped whenever the gradient becomes very small (see Corollary 1).

Definition 2 (Convergence) *A gradient based optimization algorithm with parameters $\boldsymbol{\theta}$ is said to be converged when $\|\boldsymbol{\theta}^t - \boldsymbol{\theta}^{t-1}\| < \epsilon$; where ϵ is a tolerance.*

Corollary 1 (Condition for GD convergence, proof in Appendix A) *In GD, the convergence is achieved when $\|\mathcal{G}^{t-1}\| < \frac{\epsilon}{\eta}$.*

2.3 Stochastic Gradient Descent

Applying GD on large datasets can be computationally challenging. One can accumulate gradient for each training datum instead of taking gradient of their sum—meaning $\frac{1}{m} \sum_{i=1}^m \nabla_{\boldsymbol{\theta}} \ell(\mathbf{x}_i, \mathbf{y}_i; \boldsymbol{\theta})$ instead of $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$ (see Equation (2.3))¹. Applying a single step after visiting all the data instances one by one can help with using less storage in [backpropagation](#). That is to accumulate gradient corresponding to each data instance without taking any steps. When all instances are visited, a single step is taken in the direction of the accumulated gradient. As a result, less memory is needed to cache signals of the forward pass. However, because parameters are updated after visiting all data points it can still take a long time especially for large datasets (even never happening for a never-ending stream of data).

¹The outcome may not be exactly the same if additional operations relies on the statistics calculated across the batch of the data in forward pass; e.g., batch normalization layers are used.

Stochastic Gradient Descend (SGD) [Robbins and Monro, 1951] updates the parameters immediately after visiting individual data instances. The “stochasticity” comes from the fact that these instances are drawn at random. Stochastic Gradient Descend (SGD) updates the parameters as follows:

$$\boldsymbol{\theta}^t = \boldsymbol{\theta}^{t-1} - \eta \nabla_{\boldsymbol{\theta}} \ell(\mathbf{x}_i, \mathbf{y}_i; \boldsymbol{\theta}^{t-1}); \quad i \sim \mathcal{U}(1, m); \quad (2.5)$$

where $\mathcal{U}(1, m)$ is the discrete uniform distribution from 1 to m .

In Theorem 2, we show that SGD is an unbiased estimate for GD. Note that in the proof of this Theorem, we use the concept of pseudo-gradient which we define as follows:

Definition 3 (Pseudo-gradient) For $t' > t$, the difference $\boldsymbol{\theta}^{t'} - \boldsymbol{\theta}^t$ is called a pseudo-gradient vector at $\boldsymbol{\theta}^t$.

As is implied by the term “pseudo”, there can be different ways of defining a pseudo-gradient based on how the definition may help introducing other concepts. In our definition, a pseudo-gradient vector is described using two points in the parameter space. Unlike the true gradient, a pseudo-gradient is not necessarily tied to a loss function. Pseudo-gradient helps us to estimate true gradients (given the same origin in the parameter space) mainly in this chapter and Chapter 7.

Theorem 2 (SGD unbiased, proof in Appendix A) SGD is an unbiased estimator for GD.

Although SGD estimates GD with no bias, it can have a high variance. The convergence can be significantly slowed down by a very noisy gradient estimate (of the full-gradient). As a result, a much smaller learning rate has to be chosen for SGD compared to that for GD [Schmidt et al., 2017, Johnson and Zhang, 2013].

A line of research work tries to reduce the gradient variance in SGD; thereby, a larger learning rates can be chosen and convergence be accelerated. We refer to these algorithms generally as Reduced Variance Stochastic Gradient Descent (RV-SGD). Among these, Stochastic Average Gradient (SAG)[Schmidt et al., 2017] reduces the variance of each step by adding the average of the gradients from recent optimization steps to the gradient found for the drawn instance. It finds the average of gradients

by storing the gradients in memory. **Stochastic Average Gradient (SAG)** is not a scalable algorithm because training a large model would require too much memory.

Stochastic Variance Reduction Gradients (SVRG) [Johnson and Zhang, 2013] and its variants [Nguyen et al., 2017, Bi and Gunn, 2021, Konečný et al., 2015, Babanezhad Harikandeh et al., 2015, Xiao and Zhang, 2014] are among the most recent and popular memory-less alternatives. **SVRG** modifies **SGD**'s update (see Equation (2.5)) as

$$\boldsymbol{\theta}^t = \boldsymbol{\theta}^{t-1} - \eta \left(\nabla \ell(\mathbf{x}_i, \mathbf{y}_i; \boldsymbol{\theta}^{t-1}) + \mathbf{h}^t - \mathbf{h}_i^t \right); \quad i \sim \mathcal{U}(1, m); \quad (2.6)$$

$$\text{SVRG} \quad \mathbf{h}^t = \nabla \mathcal{L}(\tilde{\boldsymbol{\theta}}), \quad (2.7)$$

$$\text{SVRG} \quad \mathbf{h}_i^t = \nabla \ell(\mathbf{x}_i, \mathbf{y}_i; \tilde{\boldsymbol{\theta}}); \quad (2.8)$$

where $\tilde{\boldsymbol{\theta}}$ is a snapshot of the parameters state $\boldsymbol{\theta}$ taken from previous updates. \mathbf{h} and \mathbf{h}_i can be considered estimates of the full-batch and instance gradients at the current step, respectively (here, they are the true full-batch and instance gradients at $\tilde{\boldsymbol{\theta}}$). The intuition is to regularize the parameter update with an estimate of the gradients accumulated for all *other* data instances. Johnson and Zhang [2013] suggested $\tilde{\boldsymbol{\theta}} \triangleq \boldsymbol{\theta}^{\lfloor \frac{t-1}{\nu} \rfloor}$ as a practical option in which ν is the frequency of taking snapshots.

The analytical result of this unbiased modification (see Theorem 3) is that if the empirical **loss** is strongly convex and the **loss** function over individual samples is both convex and **L -smooth**—meaning that the slope of the straight line that connects any two points on the function curve is equal or smaller than L —then the error in estimating gradient of $\mathcal{L}(\boldsymbol{\theta})$ is not only bounded but it also linearly converges to zero (refer to Johnson and Zhang [2013] for proof). Bi and Gunn [2021] investigated applying **SVRG** to non-convex problems.

Theorem 3 (SVRG unbiased, proof in Appendix A) *SVRG is an unbiased estimator for GD.*

Calculating full-batch gradients at the snapshots is computationally expensive. Towards more efficiency and under certain conditions, Babanezhad Harikandeh et al.

[2015] showed that this convergence rate is not largely impacted if a noisier estimate than the original $\mathcal{L}(\tilde{\boldsymbol{\theta}})$ —proposed by SVRG—is chosen for full-batch gradients. Nguyen et al. [2017] proposed StochAstic Recursive grAdient algorithM (SARAH), a biased version of SVRG that progressively updates the estimate for full-gradients for optimization steps applied in between taking two snapshots. Compared to SVRG, it uses different \mathbf{h} and \mathbf{h}_i as follows:

$$\text{SARAH } \mathbf{h}^t = \frac{1}{\eta}(\boldsymbol{\theta}^{t-3} - \boldsymbol{\theta}^{t-2}), \quad (2.9)$$

$$\text{SARAH } \mathbf{h}_i^t = \frac{\theta}{\eta} \nabla \ell(\mathbf{x}_i, \mathbf{y}_i; \boldsymbol{\theta}^{t-2}). \quad (2.10)$$

In Chapter 7, we propose an algorithm for FL which is partly inspired by SARAH.

Another traditional way to decrease variance of gradients in SGD is to use momentum (see Tseng [1998]). We can write SGD with momentum update as in Equation (2.6) such that \mathbf{h}^t and \mathbf{h}_i^t are defined as follows:

$$\text{SGD with momentum } \mathbf{h}^t = \frac{\beta}{\eta}(\boldsymbol{\theta}^{t-2} - \boldsymbol{\theta}^{t-1}), \quad (2.11)$$

$$\text{SGD with momentum } \mathbf{h}_i^t = \mathbf{0}. \quad (2.12)$$

SGD with momentum is sometimes formulated with an accumulative recursive variable. Theorem 4 presents this more common representation and shows its equality with the format we just defined.

Theorem 4 (velocity form of SGD with momentum, proof in Appendix A) *SGD with momentum updates described by Equation (2.6), Equation (2.11), and Equation (2.12) is equivalent to applying*

$$\boldsymbol{\theta}^t = \boldsymbol{\theta}^{t-1} - \eta \mathbf{v}^t; \quad (2.13)$$

where

$$\mathbf{v}^t = \beta \mathbf{v}^{t-1} + \frac{\theta}{\eta} \nabla \ell(\mathbf{x}_i, \mathbf{y}_i; \boldsymbol{\theta}^{t-1}); \quad i \sim \mathcal{U}(1, m). \quad (2.14)$$

The scalar β in Equation (2.14) and Equation (2.11) is sometimes called **momentum** and it determines the weight of the previous gradient estimate in calculating the new one. β can also be thought as a discount factor in these formulas.

2.4 Mini-batch Stochastic Gradient Descent

Given m as the number of training examples, the computing cost of a [GD](#) step is $O(m)$. The same cost is $O(1)$ for a [SGD](#) step. However, as discussed in section 2.3, high variance of the full-gradient estimates causes [SGD](#) to converge relatively slowly per step. A more popular approach especially used for training [DNNs](#) is [mini-batch SGD](#) which is a trade-off between [GD](#) and [SGD](#). [Mini-batch SGD](#) randomly draws a determined number of samples from the data (compared to only one sample from the data in [SGD](#)) and use them to apply an optimization step. This number is often referred to as *mini-batch size*. Let the set of mini-batch indices drawn at step t be \mathcal{P}^t . We often consider the mini-batch size $|\mathcal{P}^t|$ to be a fixed scalar for all t ². The update rule for mini-batch [SGD](#) on parameters at step t can be written as

$$\boldsymbol{\theta}^t = \boldsymbol{\theta}^{t-1} - \eta \nabla \left(\frac{1}{|\mathcal{P}^t|} \sum_{i \in \mathcal{P}^t} \ell(\mathbf{x}_i, \mathbf{y}_i; \boldsymbol{\theta}^{t-1}) \right), \quad (2.15)$$

where the operator $|\cdot|$ represents cardinality.

The mini-batch version of [SGD](#) can be combined with other [RV-SGD](#) methods. For example, [Konečný et al. \[2015\]](#) uses it with [SVRG](#)-like algorithms. Furthermore, applying momentum on top of mini-batch [SGD](#) is a common practice for training [DNNs](#).

2.5 Resilient Gradient-based Methods

The gradient based optimization algorithms that we have discussed up to here directly use point gradients in taking their steps. A major problem in optimizing highly non-convex functions using these methods is stepping through a saddle point of plateau region in the loss landscape. In this situation, the magnitude of the updates may differ extremely from one another across different dimensions (corresponding to different parameters).

In order to address this issue, [Resilience backpropagation \(RProp\)](#) [[Riedmiller and Braun, 1993](#)] escapes plateaus and saddle points by disentangling the magnitude of the gradient across different dimensions. For each dimension, it exponentially boosts

²See [Qian and Klabjan \[2020\]](#) as an example discussion on the relation between mini-batch size and the stability of [SGD](#).

its step size (learning rate) as long as the sign of the gradient remains unchanged. Whenever the sign of gradient flips in a dimension, the step size of that dimension is scaled down. As a result, [RProp](#) moves slower across sharp descending dimensions and faster across flattened ones. Eventually, the update for each dimension uses only the sign of the gradient in that dimension and its adaptive step size. [RProp](#) is a full-batch algorithm so it does not work well with high variance stochastic gradients as in [SGD](#) and [mini-batch SGD](#). Applying it on [SGD](#) results in an estimator of [GD](#) that not only has high variance but also is largely biased.

A more popular approach is to ignore the magnitude of gradient simply by dividing the gradient on its magnitude in each dimension. [Adaptive Subgradient \(Adagrad\)](#) [[Duchi et al., 2011](#)] takes this approach; however, in order to decrease the bias and variance and to become applicable to work with stochastic gradients, it divides the gradient on a discounted running sum of its magnitude instead of the instant one. [Root Mean Square Propagation \(RMSProp\)](#) [[Tieleman et al., 2012](#)] is similar to [Adagrad](#) but uses [momentum](#) method to find a running *average* instead of a discounted running sum. [Adam](#) [[Kingma and Ba, 2015](#)] combines [RMSProp](#) with the [momentum](#) method for [mini-batch SGD](#). There exist several practical reports that [Adam](#) requires a few warm-up steps (with no parameter updates, only calculating the first and second moments) before the actual updates on parameters begin. [Liu et al. \[2019\]](#) found that this issue stems from [Adam](#) having large variance updates at the beginning of the optimization. They addressed it with a method that they named [Rectified Adam \(RAdam\)](#).

Chapter 3

Background to Transfer Learning

To learn is to acquire knowledge or skills through experience or by acquiring knowledge from elsewhere. Learning by experience can be difficult and costly at times [Kang et al., 2010, Miśkiewicz, 2018]. Perhaps this is why “reinventing the wheel” is not always a good idea. As a result, effective transfer of information (others’ experience) and knowledge (what others’ learned from their experience) is essential to an efficient learning process [Khamseh and Jolly, 2008]. In human cognition sciences, transferring information¹ (data) and knowledge from one person or entity to another [Subedi, 2004] or even within an entity (e.g., from a concept to another in a person’s mind) [Perkins et al., 1992] is often referred to as *Transfer of Learning*.

In the field of ML, the learning entities are machines and Transfer of Learning is simply called **Transfer Learning (TL)**. The term **Transfer Learning** can be misleading because it can refer to either the transfer of learning methodology or a learning process that involves the transfer of knowledge or information. The latter target-oriented concept is referred to as **TL** in this work. Accordingly, the goal of **TL** is to improve **ML** outcomes by incorporating data, information or knowledge from the source machines into the target machines’ learning processes. The machines from which the transfer is performed are sometimes referred to as upstream. The target machines that receive content from upstream machines are sometimes called downstream machines. It is worth noting that we distinguish between machines and devices. The former are computer programs, whereas the latter are physical computers. A device, according to this definition, can host one or more machines.

In the remainder of this chapter, we will first iterate over different types of **TL** from one machine to another in terms of the transferred content. We then enumerate other aspects of **TL**, as well as topics related to **TL** and finally we will get into more details on **HTL**. To establish some common language in order to describe each

¹knowledge is said to be processed information.

concept, let the downstream (target) domain and task be $\mathcal{D}_t = \{\mathcal{X}_t, P_t(X)\}$ and $\mathcal{T}_t = \{\mathcal{Y}_t, f_t(\cdot)\}$, respectively. Furthermore, assume there exists an upstream (source) machine that has data and correspondingly provides domain $\mathcal{D}_s = \{\mathcal{X}_s, P_s(X)\}$ and task $\mathcal{T}_s = \{\mathcal{Y}_s, f_s(\cdot)\}$.

3.1 Content-based Categorization of Transfer Learning

3.1.1 Data Transfer Learning

Assume $\mathcal{T}_t = \mathcal{T}_s$ and $\mathcal{X}_s = \mathcal{X}_t$ (or feasible with a transformation such as scaling for image data) and the cost of transferring the supplementary data itself to the target machine is insignificant. Then, if the goal is to train a model that can generalize to both domains, the data can be directly combined. We already defined this type of transfer learning as [data centralization](#) which is not usually even considered a part of [ML process](#)² but rather a matter of data collection/integration [[Roh et al., 2019](#)].

Under the same conditions, direct [data centralization](#) is not a viable option if $\mathcal{T}_t \neq \mathcal{T}_s$. However, transferred supplementary data may still help learning the target task if there are similarities between the source and target domains. [Fernandes and Cardoso \[2019\]](#) and [Lao et al. \[2021\]](#) refer to this concept as *data-driven Transfer Learning*. However, we find the term [Data Transfer Learning \(DTL\)](#) more appealing because it explicitly resembles the transfer of data. Similarly, [Hypothesis Transfer Learning \(HTL\)](#) is [ML](#) that involves transferring hypotheses. [Figure 3.1a](#), depicts [DTL](#) conceptually and compares it to [HTL](#). An example of [DTL](#) is [instance weighting](#) [[Shimodaira, 2000](#), [Lienen and Hüllermeier, 2021](#)] in which the influence of each training datum is determined by matching it against the supplementary data—or a distribution derived from it.

When the supplementary data is unlabeled, [DTL](#) can be performed in a [semi-supervised learning](#) setting. The literature on [semi-supervised learning](#) has a long tail with many sub-branches [[Van Engelen and Hoos, 2020](#)]. [Self-taught learning](#) [[Raina et al., 2007](#)] can be another form of [DTL](#). It is different from [semi-supervised learning](#) in that if the unlabeled data is labeled, its label set can be different from that

²It may be arguable that [data centralization](#) is part of the [ML](#) or not in active learning. We only consider passive learning scenarios in this thesis. That is when learning and collecting further data are not performed simultaneously on a device.

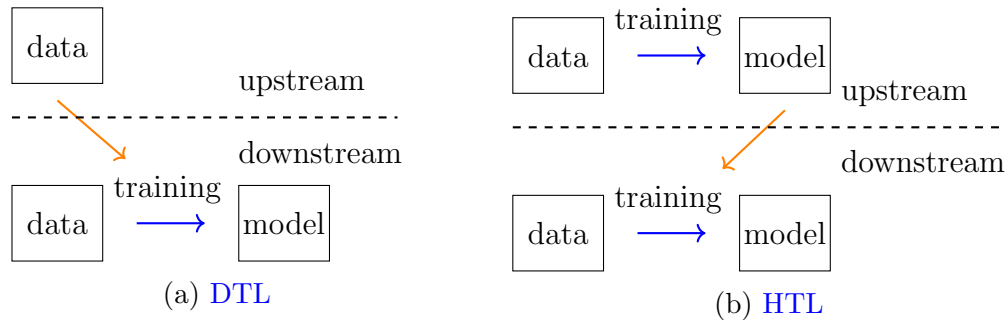


Figure 3.1: **DTL** versus **HTL**. Blue horizontal arrows represent **training** while orange vertical arrows indicate **transferring**. Transferring anything else about the data which is not a hypothesis for a task defined on the source or target machines is referred to as **ITL**.

of the labeled data³. For example, if the objective is to classify images of apples and oranges, in **semi-supervised learning** the supplementary unlabeled data also includes only images of apples and oranges while in **self-taught learning**, they can be images of other things such as fruit baskets and knives.

3.1.2 Hypothesis Transfer Learning

A model that is trained to perform a task on an upstream machine is referred to as a hypothesis for that task. Upon being transferred to downstream machine it is also called a **pretrained model**. Transferring knowledge through hypotheses is known as **Hypothesis Transfer Learning (HTL)**. The corresponding concept is shown in Figure 3.1b. There are various sub-categories of **HTL** based on the mechanism for incorporating knowledge embedded in pretrained models into downstream task learning [Sermanet et al., 2013, Hinton et al., 2015, Li and Hoiem, 2017]. **HTL** is called **task adaptation** when \mathcal{T}_t and \mathcal{T}_s are unequal or when on the target machine we do not have the knowledge about their equality (e.g., not knowing which class corresponds to the model’s output at index 0).

Task Adaptation

A popular **task adaptation** practice is to infer the features for the downstream task directly by using the feature-extractor part of a pretrained model. In this context,

³This definition is according to Raina et al. [2007].

inferring features means that the parameters of the feature-extractor are not trained during the downstream training or they are “frozen”. The extracted features can be mapped to the output space required by the target task using a **Fully Connected (FC)** layer (also called linear layer) which we refer to as the model’s **head**. Because only the **head** is trained in this arrangement, this type of **task adaptation** is often referred to as **feature-extraction** [Li and Hoiem, 2017]. Figure 3.2a color codes the frozen and trainable components in a **feature-extraction**.

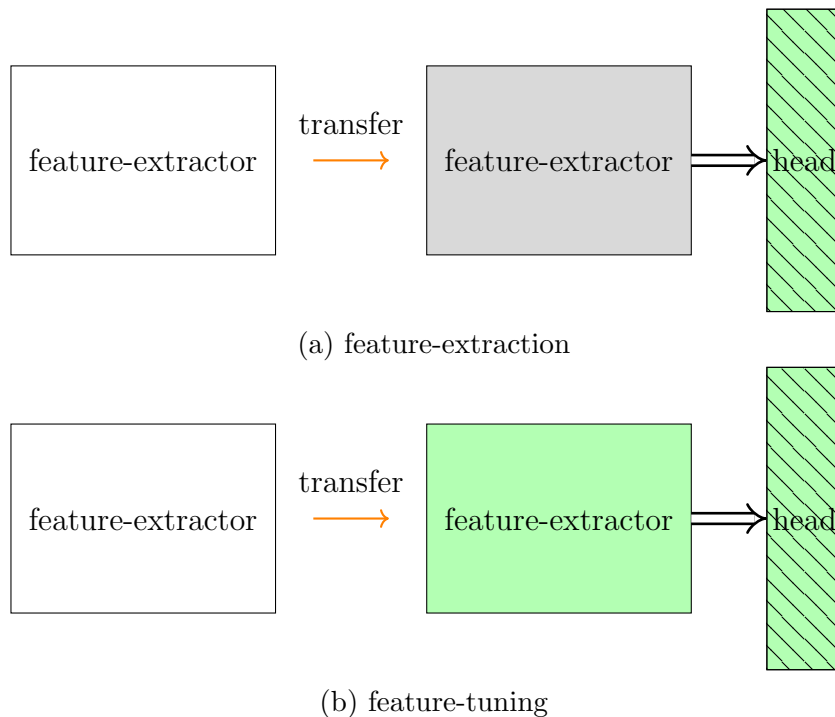


Figure 3.2: (a) **feature-extraction** versus (b) **feature-tuning**. **green** boxes represent components that are trained to learn the downstream task where as the **gray** box, only works in inference mode. The hatch patterns on the head boxes indicate random initialization.

Some research work refer to this approach as *Linear Probing* [Kumar et al., 2021, Chen et al., 2021]. This term can be confused with a method for resolving collisions in **hash tables**. This terminology may be borrowed from research studies that use **FC** layers to understand intermediate representation of **DNNs**⁴ [Alain and Bengio, 2016, Liang et al., 2022, Laina et al., 2021]. Since there is no actual probing involved in

⁴In these studies, linear probes are referred to the **FC** layers that are trained while the rest of the layers are kept unchanged.

[feature-extraction](#), we avoid using the alternative term Linear Probing throughout the rest of this thesis.

Only training an [FC](#) layer, while a pretrained feature-extractor works in inference mode, as in [feature-extraction](#), provides a quick and easy to optimize solution. However, features that are tailored specifically for the downstream task can provide much better results [[Kornblith et al., 2019](#), [He et al., 2020](#)]. Another popular [task adaptation](#) method motivated by this argument is to train the pretrained feature-extractor along with the appended model’s [head](#). This method is conceptually shown in [Figure 3.2b](#) and is usually called *fine-tuning*. To refer to this method, we use the term [feature-tuning](#) in this thesis⁵.

Head Initialization

Initializing the [head](#)’s parameters with large values leads to larger distortion in the pretrained parameters which eventually results in unnecessarily slow convergence in cases where the upstream and downstream data are similar. Additionally, we show in [Chapter 5](#) that doing so can accelerate forgetting the already learned upstream task. It means that in case careful [head](#) initialization is undermined, the upstream features can be forgotten even before anything is learned about the downstream domain and task, making the knowledge transfer pointless.

One solution to this problem is to first apply [feature-extraction](#) and then [feature-tuning](#) [[Li and Hoiem, 2017](#), [Kanavati and Tsuneki, 2021](#)]. This method is depicted in [Figure 3.3](#) where each phase of the training is shown in one row and the transition between [feature-extraction](#) and [feature-tuning](#) is marked with black arrows. In [Chapter 5](#) and [6](#), we introduce more elegant solutions which do not require manual switching between the two phases of the training (also see [[Varno et al., 2019, 2020](#)]).

⁵In ML literature, fine-tuning dates back to the late 1980s [[Waibel et al., 1989](#)]. However, more related to “only training the head” is the seminal work of [Hinton and Salakhutdinov \[2006\]](#) in which the first relatively *deep* model (in terms of number of layers) is constructed by stacking [Restricted Boltzmann Machines \(RBMs\)](#) trained on unlabeled data. In that approach, a [head](#) appended on the top of the model is “fine-tuned” using [backpropagation](#) on the same data but in the supervised fashion. Fine-tuning also refers to a concept in theoretical physics which indicates very careful and precise adjustment of parameters of a theoretical model that describes a phenomenon in the universe (e.g., as fine-tuning is used in [Wetterich \[1984\]](#)). In contrast, we show in [Chapter 5](#) that the updates on the feature-extractor parameters can have large magnitude and impact. Unlike the term “fine-tuning”, [feature-tuning](#) reflects a better contrast with [feature-extraction](#) and causes less confusion with the aforementioned unrelated concepts to the [task adaptation](#) method of our interest.

In our methods, we initialize the model’s head to small numbers. Doing so makes the updates on the pretrained feature-extractor small at the beginning of the training which leads to much better preservation of the transferred knowledge. The way each of our methods controls the increase in the magnitude of updates to the feature-extractor parameters is different. One uses an extra normalization layer between the feature-extractor and the model’s head, while the other adjusts a learning rate for the [head](#) that is independent from the learning rate of the feature-extractor.

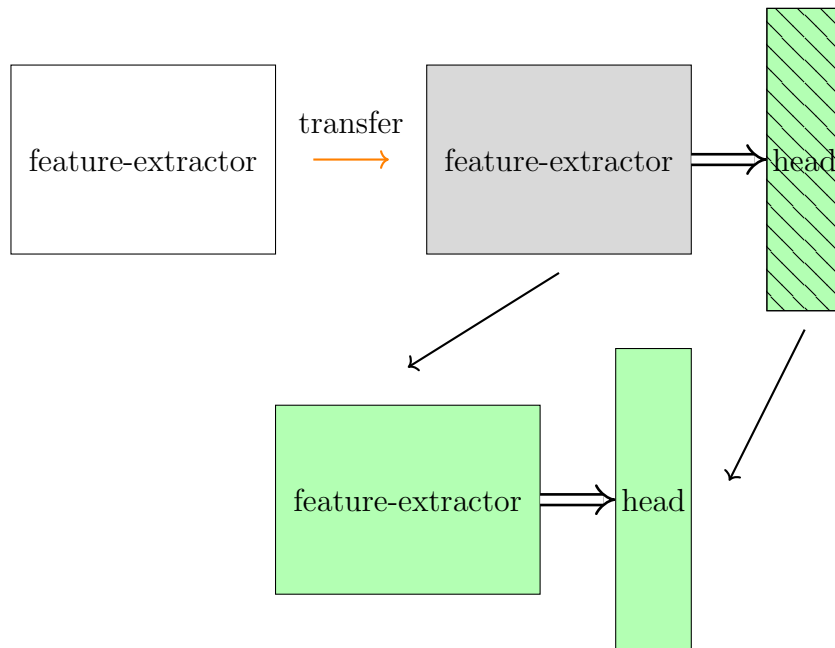


Figure 3.3: [Feature-tuning](#) after [feature-extraction](#). Boxes that are filled with [green](#) color are trained whereas those filled with [gray](#) color only work in inference mode. The patterns on the head boxes indicate random initialization.

3.1.3 Information Transfer Learning

Perhaps the least frequently used form of getting help from supplementary data in learning a task is [Information Transfer Learning](#) (ITL). This involves transferring only some statistics about the data on upstream machine or some information about a model trained on it. An example of this category of [TL](#) is the method [Raghu et al. \[2019\]](#) named “Mean Var init”. In this technique, the model’s parameters are initialized with mean and variance transferred from a hypothesis for the upstream task. As we will see in [Chapter 7](#), it is common in [FL](#) that a mixture of information

and hypothesis is transferred from upstream machines to downstream machines.

3.2 Federated Learning

Federated Learning (FL) is a promising alternative to **centralized learning** that potentially can scale to countless continuous transfers and preserve the privacy of data/device owners. In **FL**, several devices with data known as clients iteratively collaborate to train a model by communicating knowledge through a central hub called server. At each round, the server broadcasts the cloud's knowledge to a random selection of clients. Given this transferred knowledge, recipient devices locally extract knowledge from their data and transfer their results back to the server where the cloud's model is updated by aggregating the transferred knowledge from the clients. This cycle is called a round of communication and goes on until the training converges to a stationary point.

FL is different from ordinary **HTL** in two important ways:

1. Typically, the downstream machine has no direct local access to any data by itself. In most cases, it only seeks learning through aggregating the knowledge of the upstream machines.
2. The downstream machine iteratively queries the upstream machines to align the returned hypotheses with the overall learning goal. Such **TL** is characterized with the concept we described as **query-based HTL**. See Figure 3.4 for comparing this concept with a an ordinary **HTL** (**query-less HTL**).

Additionally, in the context of **FL** we mostly refer to learning parties as **devices** rather than **machines** mainly to emphasize the fact that the involved parties are physically distinct entities (i.e., not machines running on the same computer).

FL is a broad research area and it has various categories to address different **decentralized learning** scenarios. However, the core research on this realm is focused on the settings in which there is a single task to learn and local datasets are possibly distributed heterogeneously with respect to each other. If the data can be centralized (hypothetically stacked up), the resulting dataset is referred to as the **oracle dataset**. Often the goal of **FL** is to train a model that performs on the same level as one that is well trained on the **oracle dataset**.

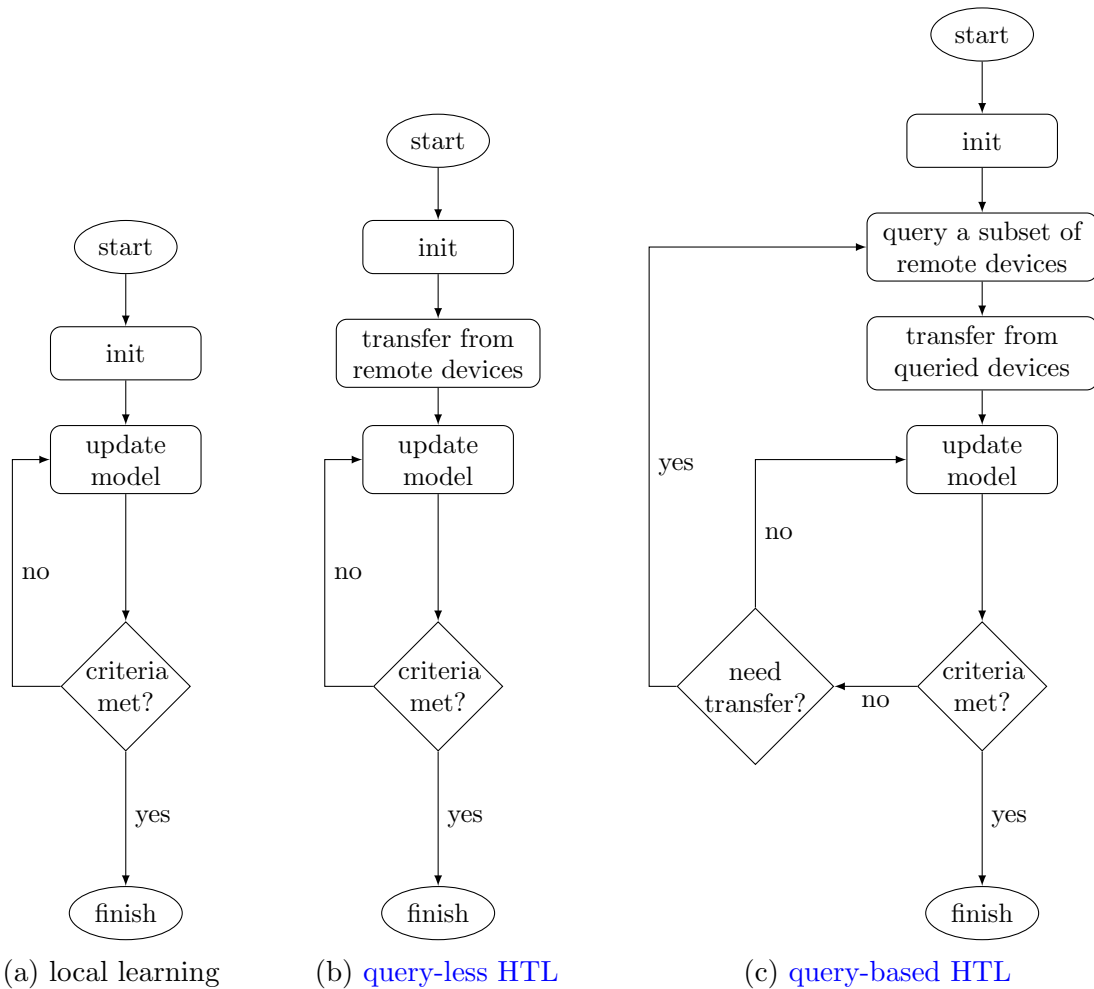
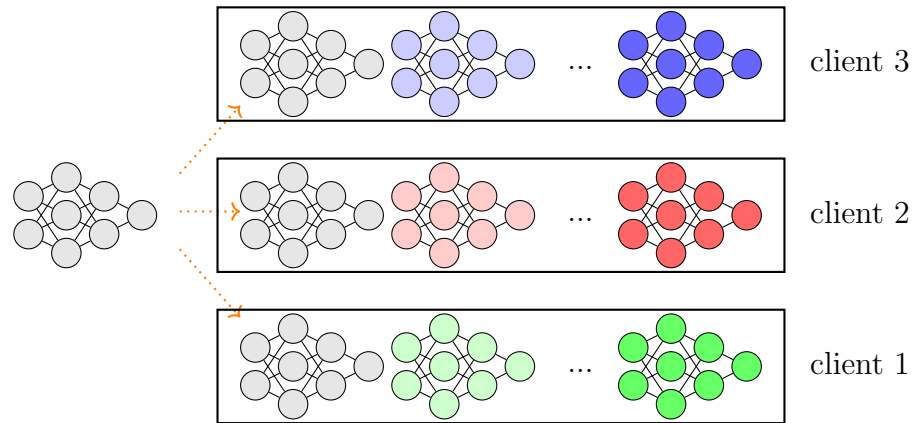
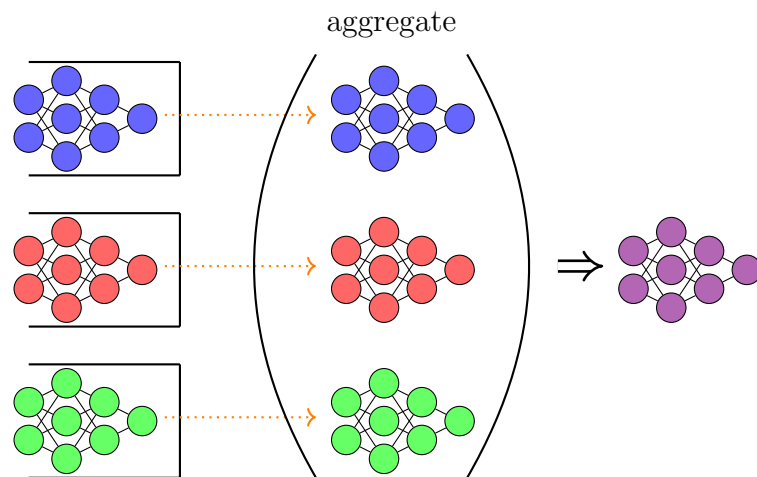


Figure 3.4: Flowchart of iterative optimization for training a DNN. Comparing (a). local training, (b). [query-less HTL](#) , and (c). [query-based HTL](#).


The simplest known FL algorithm is referred to as [Federated Averaging \(FedAvg\)](#) [McMahan et al., 2017] or [Local Stochastic Gradient Descent \(LocalSGD\)](#) [Stich, 2018]. In [LocalSGD](#), the server, initiates a round of communication by sending its *cloud model* (hypothesis for the oracle dataset) to a random selection of the clients. The selected clients train these copies (same initial state) on their local data and send the trained models (hypotheses) back to the server where they are averaged to update the next round’s *cloud model*. Transferring hypotheses to/from clients in [Local Stochastic Gradient Descent](#) is shown in Figure 3.5a/3.5b.



(a) server transfers cloud hypothesis and clients train locally.



(b) clients transfer local hypothesis and server aggregate.

Figure 3.5: A round of communicating hypotheses in **LocalSGD**. The symbol  represents a **DNN**. Changes in model parameters is shown with color changes.

Chapter 4

Related Work

The advancements in deep learning and [Convolutional Neural Networks \(CNNs\)](#) have completely transformed the landscape of [ML](#). With their exceptional abilities in performing machine vision tasks, these concepts have opened new horizons in the field of [AI](#). However, as the field has matured, novel challenges have emerged, and researchers are grappling with issues such as how to transfer knowledge between different models, how to continually learn and adapt to new data, and how to train models with data that cannot be centralized. This related work chapter delves into these developments from where they began to their current state.

4.1 Deep Learning and Convolutional Neural Networks

[Connectionism](#) research work of [Fukushima and Miyake \[1982\]](#) and [Mozer \[1987\]](#) on hierarchical representations and valuing the approximate positioning of patterns over their absolute positioning inspired the introduction of [CNNs](#) by [LeCun et al. \[1989\]](#). As stated by [LeCun et al. \[1989\]](#), the idea is also based on the “weight sharing” concept, already indicated in original back-propagation paper [[Rumelhart et al., 1986](#)]. They used this approach to train [LeNet](#), a seven layer [CNN](#) that recognized hand-written digits as well as ASCII characters [[LeCun et al., 1998](#)].

Seven years later, [Hinton et al. \[2006\]](#) introduced a novel way to progressively train [DNNs](#). In the same year, they used this approach to train a model [[Hinton and Salakhutdinov, 2006](#)], a [Deep Belief Network \(DBN\)](#) constructed from [Restricted Boltzmann Machines \(RBMs\)](#). The idea was to first learn a good representation of the inputs in an unsupervised fashion, making optimization for the subsequent supervised learning stage easier. This was a beginning of a line of research in representation learning using [energy-based models](#) [[Ranzato et al., 2006](#), [Bengio et al., 2006](#), [Lee et al., 2009](#), [Salakhutdinov and Hinton, 2009](#), [Vincent et al., 2010](#), [Bengio, 2012](#)].

In 2012, University of Toronto marked another important milestone in the history

of Computer Vision. Their CNN, AlexNet [Krizhevsky et al., 2017], won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) by a large margin compared to the other competitors. AlexNet is not a DBN by definition as it does not use components that are pretrained in an unsupervised fashion. Moreover, despite the popularity of sigmoid and Tanh by the time, AlexNet followed Jarrett et al. [2009] and Nair and Hinton [2010]¹ in employing Rectified Linear Units (ReLUs) as activation function.

Since then, various strategies have been proposed to ease the burden of training very deep CNNs which along with advancement in parallel processing made these models capable of undertaking the most complex vision tasks. These strategies include initialization methods [Glorot and Bengio, 2010, He et al., 2015], optimization methods [Nesterov, 1983, Tieleman et al., 2012, Kingma and Ba, 2015, Liu et al., 2019, Foret et al., 2020], and normalization methods [Ioffe and Szegedy, 2015, Ulyanov et al., 2016, Ba et al., 2016, Wu and He, 2018] as well as architectural modifications [Simonyan and Zisserman, 2014, Szegedy et al., 2015, He et al., 2016, Szegedy et al., 2016, Zagoruyko and Komodakis, 2016, Huang et al., 2017, Xie et al., 2017, Tan and Le, 2019, Tan et al., 2019, Radosavovic et al., 2020] and many other techniques and tricks [He et al., 2019b].

4.2 Hypothesis Transfer Learning

4.2.1 The First HTL Attempts

Recently, Stevo Bozinovski published an overview paper about his own research work on TL during 1970s and early 1980s [Bozinovski, 2020]. This body of work which is stated to be mostly written in Croatian language, investigates the effect of pre-training a neural network for a supervised learning task, on what he refers to as the “learning trajectory” of another supervised learning task. This is probably the earliest appearance of TL in Neural Networks literature.

¹Jarrett et al. [2009] refers to rectified units as the “positive part” function.

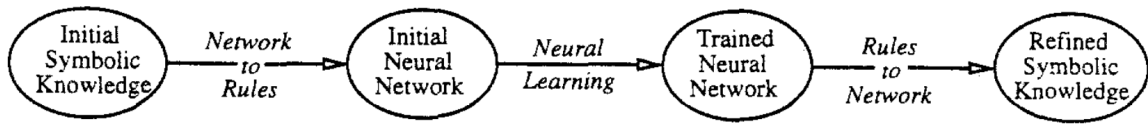


Figure 4.1: Symbolic reasoning refinement through exporting to and then importing from neural networks. Figure taken from [Towell and Shavlik \[1993\]](#).

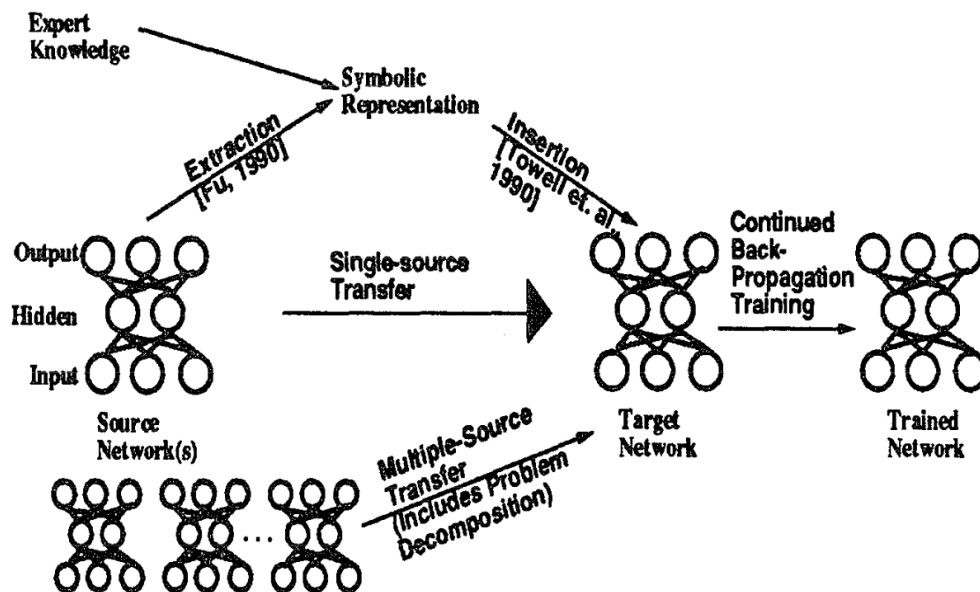


Figure 4.2: HTL methods introduced by [Pratt et al. \[1991\]](#). Problem decomposition refers to the concept we named [task decomposition](#). Figure directly taken from [Pratt et al. \[1991\]](#).

4.2.2 Linking ANNs and Symbolic Representation

Fast-forwarding to late 1980s, the appealing idea of transferring domain knowledge in [Symbolic Machine Learning](#) [[Callan and Utgoff, 1991](#), [Fawcett and Utgoff, 1991](#), [Utgoff, 1984](#)] inspired a group of research studies to import knowledge into neural networks from symbolic representations or vice versa [[Towell et al., 1990](#), [Sayegh, 1992](#), [Towell and Shavlik, 1993](#)] (see Figure 4.1). [Pratt et al. \[1991\]](#), proposed directly transferring a single upstream hypothesis to a downstream task (see Figure 4.2). The authors further added to this line of work in their follow-up research studies that appeared in [Pratt and Kamm \[1991\]](#) and [Pratt \[1993\]](#).

4.2.3 Task Decomposition

Waibel et al. [1989] proposed [task decomposition](#)² for neural networks. It was to break a classification task into multiple smaller tasks, pretrain a hypothesis for each, and then “glue” them together to construct a larger model for the original task. In this setting, the knowledge is transferred through hypothesis from multiple source tasks $\mathcal{T}_{s_1}, \mathcal{T}_{s_2}, \dots, \mathcal{T}_{s_n}$ to the target task $\mathcal{T}_t = \bigcup_{i=1}^{i=n} \mathcal{T}_{s_i}$. The motivation was to speed up the training by better initialization. [Task decomposition](#) is sometimes also referred to as problem decomposition (see Figure 4.2). Waibel [1988] was followed by a number of other studies (e.g., Waibel et al. [1989], De Bollivier et al. [1991] and Hong et al. [1996]).

4.3 Task Adaptation

4.3.1 Unsupervised Pretraining

Primitive DNNs were difficult to train. In those models, [sigmoid](#) or [Tanh](#) activation functions were typically used, which can result in gradient vanishing. Additionally, large labeled datasets for complex tasks (such as face recognition) were rare. To overcome these shortcomings, a common training strategy for a DNN was to progressively pretrain one layer after another on unlabeled data in an unsupervised fashion [Hinton et al., 2006, Hinton and Salakhutdinov, 2006, Bengio et al., 2006, Le et al., 2011]. Note that such layer-wise training was already introduced for speeding up training shallow neural networks, however, earlier research work applied pretraining in a supervised fashion [Fahlman and Lebiere, 1989, Lengellé and Denoëux, 1996]. In almost all these works, pretraining was often performed on the same data as the target supervised learning. We still recognize this strategy as [HTL](#) because the target [supervised learning](#) is boosted by a hypothesis. Le et al. [2011] practically kept the supplementary unsupervised data on several remote machines and pretrained a model in a distributed fashion. Their work in aggregating the gradients is reminiscent of [FL](#).

²We borrow the terminology from research work like Sharkey [1997] and [Sharkey, 1999].

4.3.2 Pretraining CNNs

LeCun et al. [2015] summarize [Deep Learning](#) as methods that enable learning “representations of data with multiple levels of abstraction” for computational models with the help of [backpropagation](#) [Rumelhart et al., 1986]. Visually demonstrating the features at different layers of [CNNs](#), Zeiler and Fergus [2014] further supported the idea of “multiple levels of representations” [Bengio, 2012]. According to this concept, in a [feedforward CNN](#)³, deeper representations of data correspond to higher-level features [Bengio et al., 2006, 2011, LeCun et al., 2015]. In other words, the outputs of layers that are closer to the input provide more abstract representations which consequently are more transferable. In particular, the model’s [head](#) has strong ties to the learning task [Neyshabur et al., 2020] and can be viewed as a transformation from high-level extracted features to the output space defined by the task.

4.3.3 Supervised Pretraining for CNNs

Donahue et al. [2014] employed [HTL](#) on image classification in a fully supervised fashion. That is supervised pretraining on upstream and supervised training on down-stream which we refer to as [task adaptation](#) if the source and target tasks are different (otherwise, we refer to it as [domain adaptation](#)). They applied [feature-extraction](#), meaning that their pretrained feature-extractor remained frozen during training the [head](#) on the target task (see Figure 3.2). Girshick et al. [2014] took the same supervised pretraining approach except that they applied [feature-tuning](#), meaning that feature-extractor and head are optimized towards learning the target task together.

4.4 Continual Learning

[Continual Learning \(CL\)](#) [van de Ven and Tolias, 2019, Hsu et al., 2018], also known as “lifelong learning” or “incremental learning”, refers to an [ML](#) system’s ability to continuously learn and adapt from new data. It consists of several learning sub-scenarios with real life applications that take into account incrementally learning new

³In a [feedforward](#) model, the output of a layer cannot be input to itself or another layer that is closer to the input.

tasks (e.g., [Li and Hoiem \[2017\]](#)) or domains (e.g., [Churamani et al. \[2022\]](#)) or classes (e.g., [Rebuffi et al. \[2017\]](#) and [Liu et al. \[2020\]](#)). CL involves knowledge transfer from previously learned tasks or domains (forward knowledge transfer) to new ones or vice versa (backward knowledge transfer). For instance, [feature-tuning](#) a pretrained DNN on a target task, can largely degrade the performance of the model on the source task. This phenomenon is widely known as [catastrophic forgetting](#) [[McCloskey and Cohen, 1989](#)]. There have been several studies to alleviate forgetting in DNNs. To name a few, [Goodfellow et al. \[2013\]](#) empirically found that employing dropout [[Srivastava et al., 2014](#)] results in less catastrophic forgetting though it may come at the cost of performing sub-optimally on the new task. [Kirkpatrick et al. \[2017\]](#) introduced a way to find parameters that are more important for the source task and to assign a smaller learning rate to them. Due to its practicality, CL has recently become more popular and has even been studied in combination with several other learning scenarios (e.g., [Lao et al. \[2020\]](#)).

4.5 Distributed Machine Learning

Primitive Distributed ML

In 1983, [Davis and Smith \[1983\]](#) introduced “distributed problems solving” framework as follow

In our view, three defining characteristics of distributed problem solving are that it is a cooperative activity of a group of decentralized and loosely coupled knowledge-sources (KSs), each of which may reside in a distinct processor node. The KSs cooperate in the sense that no one of them has sufficient information to solve the entire problem, so a mutual sharing of information is necessary to allow the group as a whole to produce an answer. By decentralized we mean that both control and data are logically and often geographically distributed; there is neither global control nor global data storage. Loosely coupled means that individual KSs spend most of their time in computation rather than communication.

To a large extent, this pictures a contemporary fully-decentralized learning setting except that it rather describes a collaborative system for solving any kind of problem,

not just ML. Soon after, some adaptations of primitive distributed systems were introduced for ML [Brazdil et al., 1991, Dowell and Bonnell, 1991]. The processing time and amount of memory required for learning from large datasets even motivated to partition already centralized datasets into smaller sets and distribute them to separate processing units [Provost and Hennessy, 1994].

4.5.1 Federated Learning

In 2011, Le et al. [2011] employed unsupervised pretraining to improve the image classification performance of a model with one billion parameters. For the pretraining stage, they divided the training data into multiple partitions and stored each of them on a separate device. Each device trained their own copy of the model on its local data partition. The models and gradients were periodically synchronized through “parameter servers”. By definition, Le et al. [2011] applied LocalSGD. However, they had a different motivation than FL which was to accelerate the training process of their “large model”⁴ on Central Processing Unit (CPU) cores via parallelism.

As the number of devices connected to the Internet increased and the volume of data grew, the privacy and communication costs of sharing data for ML became a more compelling reason for implementing a distributed version of SGD. Therefore, the focus, and so the terminology, gradually shifted from so called “parallel” SGD [Zinkevich et al., 2010, Recht et al., 2011, Zhang et al., 2016] to “distributed” or “privacy-preserving” SGD [Noel and Osindero, 2014, Shokri and Shmatikov, 2015]. McMahan et al. [2017] used the term Federated Learning for the first time and enumerated a number of characteristics such as large number of devices, their unreliable access to a network connection, and data heterogeneity, to distinguish it from other optimization problems.

Some follow up research work, explored FL in settings describable with different characteristics than those determined by McMahan et al. [2017]. However, they still refer to their work as FL. Consequently, several FL categorizations have emerged. Kairouz et al. [2021, Section 2] provides a taxonomy of these categories. For example, there are research work focusing on learning across a few number of clients which are referred to as cross-silo FL. In contrast, the original FL which expects “the number

⁴by the time, their model was considered very large in terms of the number of parameters.

of clients participating in an optimization to be much larger than the average number of examples per client” [McMahan et al., 2017] is considered as [cross-device FL](#).

This type of categorization allows some practical assumptions to be made about the learning setting. If you deal with [cross-silo FL](#) setting, the clients are probably some large entities like organizations. Therefore, improving privacy and decreasing computing costs matter much more than communication consumption or bias caused by stochastic client participation at different rounds of communication. In this thesis, we do not deviate from the original [FL](#) definition provided by McMahan et al. [2017].

4.5.2 Client Drift

A major challenge in [FL](#) is to deal with data heterogeneity among clients’ data which results in discrepancy between local and global objective. It can cause a *drift* in the local parameter updates with respect to the server aggregated parameters. Recent research has shown that in such heterogeneous settings, [LocalSGD](#) or [FedAvg](#) is highly pruned to client drift [Zhao et al., 2018] which can result in sub-optimal solutions.

To improve the performance of [FL](#) with heterogeneous data, some previous work use [knowledge distillation](#) to learn the cloud model from an ensemble of client models. This approach has been shown to be more effective than simple parameter averaging in reducing bias of the local gradients [Lin et al., 2020, Li and Wang, 2019, Zhu et al., 2021].

Another group of methods can be categorized as *gradient based* in which the gradients are explicitly constrained on the clients or server for bias removal. [FedProx](#) Li et al. [2020] penalizes the distance between the local and cloud parameters whereas Wang et al. [2020] normalizes the client gradients prior to aggregation. Inspired by [SGD](#) with [momentum](#), Yu et al. [2019] use a local buffer to accumulate gradients from previous rounds at each client and communicate the [momentum](#) buffer with the server as well as the local parameters which doubles the consumption of communication bandwidth. Instead of applying [momentum](#) method on the client level, Hsu et al. [2019] and Wang et al. [2019] implement a server [momentum](#) approach which avoids increasing communication costs.

Inspired by [SVRG](#) [Johnson and Zhang, 2013], some work incorporate variance reduction into [LocalSGD](#) [Acar et al., 2020, Li et al., 2019, Karimireddy et al., 2020,

Liang et al., 2019, Zhang et al., 2020, Konečný et al., 2016, Murata and Suzuki, 2021, Nguyen et al., 2017]. DANE [Shamir et al., 2014], AIDE [Reddi et al., 2016], and VRL-SGD [Liang et al., 2019] incorporated RV-SGD in distributed learning for full client participation. FedDANE [Li et al., 2019] is an attempt to adapt DANE to the FL setting⁵, but because it communicates full batch gradients, it still undermines the privacy concerns such as attacks to retrieve data from true gradients [Zhu and Han, 2020]. Most methods in this category such as VRL-SGD [Liang et al., 2019], FSVRG [Konečný et al., 2016], FedSplit [Pathak and Wainwright, 2020] and FedPD [Zhang et al., 2020] require full participation of clients which makes them less suitable for cross-device FL where only a fraction of clients participate in training at each round.

While FedDANE [Li et al., 2019] works in partial participation, empirical results show it performs worse than FedAvg [Acar et al., 2020]. In this thesis, we focus on methods that are capable of learning in partial participation settings. In particular, SCAFFOLD Karimireddy et al. [2020] uses control variates on both the server and clients to reduce the variance in local updates. In addition to the model parameters, the control variates are also learned and are communicated between the server and the clients which take up additional bandwidth. Murata and Suzuki [2021] also reduces local variance by estimating the local bias at each client and using an SVRG-like approach to reduce the drift but it requires to communicate models corresponding to all intermediate local steps. While SCAFFOLD applied variance reduction on clients, FedDyn [Acar et al., 2020] applies it partly on the server and partly on the clients. The method we propose in Chapter 7, is probably closer to FedDyn than to others; however, they differ in the way gradients are estimated. See section 7.4.3 for detailed comparison of AdaBest with FedDyn and SCAFFOLD.

⁵Recall that FL is a sub-branch of distributed learning with specific characteristics geared towards practicality [McMahan et al., 2017].

Chapter 5

Incremental Tuning with Normalized Features

“... and there is the neural knowledge mobilization phase which is about how do I use this large neural net that has implicit knowledge to extract the right thing that I need to solve, a particular task as one user; and there might be many downstream users so many downstream tasks to be solved.”

- Hugo Larochelle, *Twiml Podcast (May 2023)*

Conventional [feature-tuning](#) methods lead to an initial perturbation of the pretrained parameters with large and noisy gradients. This problem is addressed in this chapter by making the initial updates to the pretrained parameters smaller and less noisy. Throughout the chapter, we focus on a setting where a single “well-trained” hypothesis is transferred from an upstream machine to help with learning a task on the downstream machine. In this context, “well-trained” means that the optimization algorithm is converged up to a small tolerance (see [Definition 2](#)). Furthermore, we assume that the size of the upstream training data is large enough to avoid the hypothesis to severely over-fit. Even though the preceding assumptions are weak, we will only use them in this chapter to argue about intuitions and motivations rather than proofs.

The [feature-tuning](#) algorithm we propose in this chapter, [Efficient Neural Task Adaptation via Maximum Entropy Initialization \(ENTAME\)](#) is designed to prevent back-propagating large and noisy gradients towards the pretrained feature-extractor at the beginning of the training. [ENTAME](#) differs from conventional [feature-tuning](#) simply in that it normalizes the extracted features and initializes the parameters of the model’s [head](#) to zeros.

We show empirically that, unlike baselines, [ENTAME](#) causes a gradual increase in the magnitude of gradients to the feature-extractor at the start of tuning. Therefore,

large noisy initial feature perturbations are avoided. Across several benchmarks, we show that our [task adaptation](#) method converges faster and performs better when converged than the conventional [feature-tuning](#) methods.

The rest of this chapter is organized as follows. Section 5.1 gives some background on our setup and introduces some notation. The research motivations are expressed in Section 5.2. Section 5.4 introduces our method, [ENTAME](#), which is followed by a discussion on the details of our method’s components in Section 5.5. Finally, the experimental results are provided in Section 5.6.

5.1 Background

Let \mathcal{D}_s and \mathcal{T}_s be the upstream’s [domain](#) and the upstream’s [task](#), respectively¹. Likewise, consider \mathcal{D}_t and \mathcal{T}_t be downstream’s [domain](#) and [task](#) ². A hypothesis trained on upstream is transferred to downstream, where it is called the [pretrained model](#). In [feature-extraction](#) and [feature-tuning](#), the feature-extractor part of the the [pretrained model](#) is directly used to initialize the model that is trained on downstream.

For [task adaptation](#), we consider $\mathcal{T}_s \neq \mathcal{T}_t$. Therefore, the extracted features need to be mapped into the output space \mathcal{Y} such that $\mathcal{T}_t = \{\mathcal{Y}, f(\cdot)\}$. The most common practice for applying such a mapping is to use a new [head](#) for the model. To put this arrangement into mathematical context, let us represent the feature-extractor with the function $\Omega : \mathcal{X} \rightarrow \mathcal{A}$, parameterized by ϕ . Further, let the extracted features be transformed to the output space by the [head](#) function $\mathcal{H} : \mathcal{A} \rightarrow \mathcal{Y}$ which is parameterized by \mathbf{w} . Therefore, for an input \mathbf{x} , the features are inferred as

$$\mathbf{a} = \Omega(\mathbf{x}; \phi). \tag{5.1}$$

Then, they are mapped into the output space through the head as

$$\hat{\mathbf{y}} = \mathcal{H}(\mathbf{a}; \mathbf{W}; \mathbf{b}). \tag{5.2}$$

For classification, this function is often characterized by a [softmax](#) probability estimator (see Definition 4) in the following form

$$\mathcal{H}(\mathbf{a}; \mathbf{W}; \mathbf{b}) \triangleq \text{softmax}(\mathbf{o}), \tag{5.3}$$

¹Their sub-index stands for “source”.

²Their sub-index, t stands for “target” but this t is different from superscript t used to show the number of steps.

where

$$\mathbf{o} = \mathbf{a}(\mathbf{W})^T + \mathbf{b}, \quad (5.4)$$

$\mathbf{W} \in \mathbb{R}^{(C \times Q)}$ and $(\mathbf{W})^T$ is the transpose of \mathbf{W} . Recall from Table 2.2 that C and Q denote the number of classes and extracted features, respectively; that is $C = |\mathbf{y}|$ and $Q = |\mathbf{a}|$.

Definition 4 (Softmax) *Softmax* is defined as

$$\text{softmax}(\mathbf{u}) \triangleq \frac{\exp(\mathbf{u})}{\langle \exp(\mathbf{u}), \mathbf{1} \rangle}, \quad (5.5)$$

where $\exp(\cdot)$ is a function that applies element-wise exponential operation on its input vector and $\mathbf{1}$ is a vector of all ones.

The forward flow of data through the entire model can be expressed as

$$\hat{f}(\mathbf{x}; \boldsymbol{\theta}) = \mathcal{H}(\Omega(\mathbf{x}; \boldsymbol{\phi}); \mathbf{W}; \mathbf{b}); \quad (5.6)$$

where $\hat{f}(\cdot; \boldsymbol{\theta})$ is an approximator for $f(\cdot)$ and $\{\boldsymbol{\theta}\} = \{\boldsymbol{\phi} \cup \mathbf{W} \cup \mathbf{b}\}$. As mentioned in Chapter 1, we focus on classification and assume that the popular cross-entropy loss function is employed (see equation (2.1)).

5.2 Problem Statement

Assume that the optimization algorithm converges at the end of pretraining. Based on Definition 2, the gradient of the upstream loss with respect to feature-extractor’s parameters should have a small norm (assume ϵ is very small) for the last optimization steps. For example, for mini-batch SGD that is

$$\lim_{t \rightarrow \infty} \|\nabla_{\mathbf{s}}^{\phi} \mathcal{L}(\boldsymbol{\phi}^t)\| = \frac{\epsilon}{\eta} \quad (5.7)$$

where $\mathcal{L}(\boldsymbol{\phi}^t)$ is the upstream loss at optimization step t .

Switching the task and introducing a randomly initialized layer as the head, results in a large $\|\nabla_{\mathbf{t}}^{\theta} \mathcal{L}(\boldsymbol{\phi}^0)\|$, in which $\mathcal{L}_{\mathbf{t}}(\boldsymbol{\phi}^0)$ is the downstream loss at the first step of feature-tuning. Carelessly applying back-propagation can perturb the transferred features before anything is learned, resulting in catastrophic forgetting which in turn can also slow down the convergence on the target task. Additionally, if a learning rate

decay is employed during pretraining, there would be a large difference between the magnitude of the updates applied at the end of pretraining and those at the beginning of feature-tuning.

Xavier [Glorot and Bengio, 2010] and *Kaiming* [He et al., 2015]³ are by far the most commonly used methods for initializing parameters of DNNs. The model’s *head* is often treated the same way as other layers in terms of parameter initialization. Figure 5.1 quantitatively shows how large the norm of backpropagated gradients are when Xavier and Kaiming initialization methods are used (in log scale) in an example *feature-tuning*. In this experiment, a ResNet-50 model pretrained on ImageNet is tuned on the classification task defined by CIFAR-100 dataset.

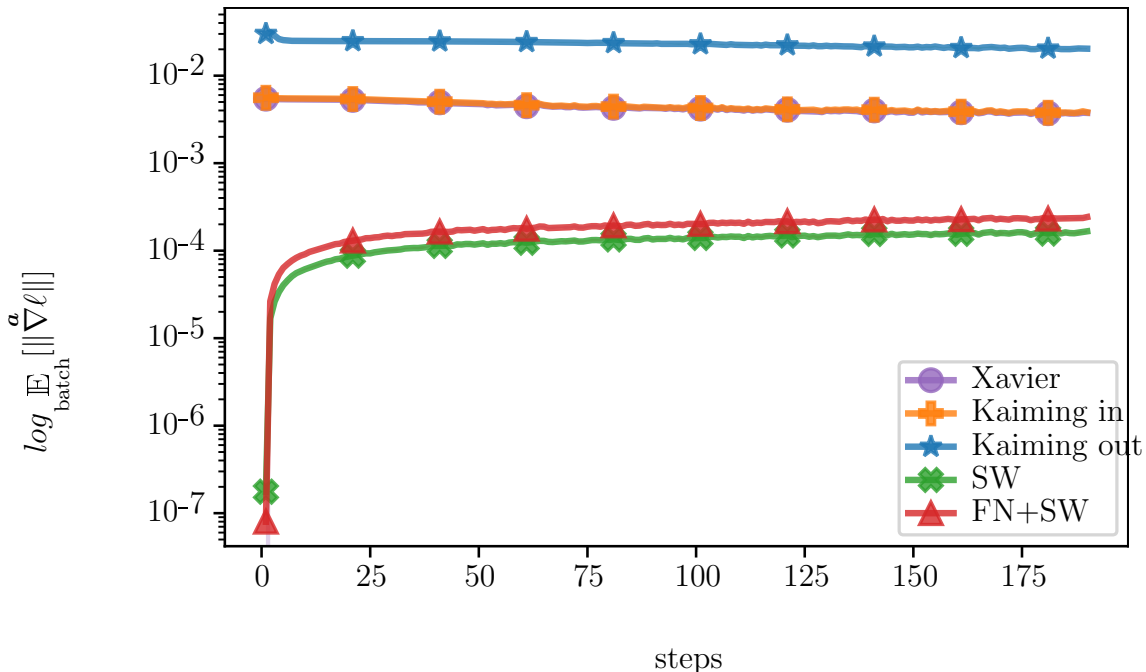


Figure 5.1: Norm of backpropagated gradient towards pretrained parameters for different initialization methods in *log* scale. See details of this experiment in Section 5.6.1.

One solution is to include an initial warmup phase in which only the *head*’s parameters are updated for a number of steps, before the feature extractor parameters are unfrozen. In other words, *feature-tuning* is applied after a phase of *feature-extraction*. The model can perform poorly during the warmup phase as the features are inferred from the upstream task.

³These methods are known by the names of the first authors of their corresponding papers.

Li and Hoiem [2017] use a validation set to determine when is a good time to switch between the warmup phase and the main one. However, a statistically large enough validation set can not be always used for the downstream task, given that a major motivation for employing HTL is the small number of training instances. Moreover, the effective number of required training steps in the warmup phase can be large, depending on the learning rate, initial values of augmented parameters and the size of the dataset. Our goal is to find a more elegant way to perform feature-tuning in a single stage without interfering with the learning process.

5.3 Noise Reduction in Feature-tuning

As mentioned in Section 5.2, the initial backpropagated gradient with respect to the feature-extractor parameters is not only large but is also mixed with irrelevant values, or noise (see Definition 6) which can perturb the pretrained features. To decrease feature perturbation, we aim to reduce both the level of noise and the magnitude of the mentioned gradients in the beginning of feature-tuning. Note that the noise that we are targeting is one that is rooted from inefficient model reconstruction. It is inherently different from the noise that may exist in data instances and their labels; therefore, sometimes for clarification we refer to it as the training noise⁴.

Definition 5 (Signal) *Signal is a quantity that contains useful information.*

Definition 6 (Noise) *A random quantity is considered noise with regard to one (multiple) signal(s) if it is not objectively related to that (those) signal(s).*

According to the chain-rule for gradients, the gradient of the loss with respect to extracted features can be expressed as

$$\frac{\partial \ell}{\partial \mathbf{a}} = \frac{\partial \ell}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial \mathbf{a}}. \quad (5.8)$$

where $\frac{\partial \ell}{\partial \mathbf{a}} = \nabla^{\mathbf{a}} \ell$ and $\frac{\partial \mathbf{o}}{\partial \mathbf{a}}$ is a Jacobian. Decreasing the training noise in $\frac{\partial \ell}{\partial \mathbf{a}}$ is equal to decreasing noise in $\frac{\partial \ell}{\partial \mathbf{o}}$ and each column of $\frac{\partial \mathbf{o}}{\partial \mathbf{a}}$. In Theorem 5 we show that $\frac{\partial \ell}{\partial \mathbf{o}}$ is equal to the prediction error $\delta = \hat{\mathbf{y}} - \mathbf{y}$ (see Definition 7). On the other hand, from Equation (5.4) we can conclude that

$$\frac{\partial \mathbf{o}}{\partial \mathbf{a}} = \mathbf{W}. \quad (5.9)$$

⁴This noise is also different from the noise related to the stochasticity of SGD.

Therefore

$$\frac{\partial \ell}{\partial \mathbf{a}} = \boldsymbol{\delta} \mathbf{W}. \quad (5.10)$$

In this equation, $\boldsymbol{\delta}$ and \mathbf{W} can include training [noise](#), impacted by careless initialization. as well as carelessly using \mathbf{W} and \mathbf{b} through the forwards and then backward passes.

Definition 7 (Prediction error) For predicted labels $\hat{\mathbf{y}}$ and true labels \mathbf{y} , the [prediction error](#) $\boldsymbol{\delta}$ is defined as

$$\boldsymbol{\delta} \triangleq \hat{\mathbf{y}} - \mathbf{y}. \quad (5.11)$$

Note that it may make more sense to call $\boldsymbol{\delta}$ “negative prediction error”; however, we drop the negative word from the term just for the sake of simplicity.

Theorem 5 (Gradient w.r.t. [head’s output](#), proof in [Appendix B](#)) The gradient of cross-entropy loss with respect to [head’s output](#) is equal to the [prediction error](#).

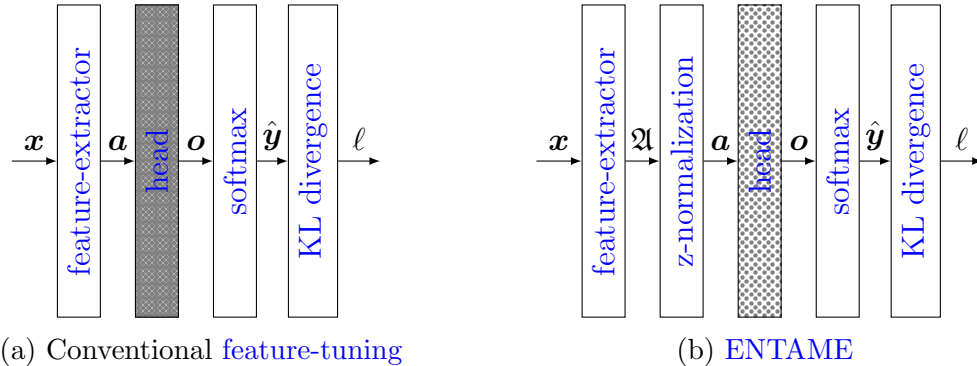


Figure 5.2: Conventional [feature-tuning](#) versus [ENTAME](#). The magnitude of the initial values of [head’s](#) parameters is symbolically shown with density of dot patterns.

5.4 ENTAME

In this section, we describe how our solution is applied to [task adaptation](#). Then in [Section 5.5](#), we discuss how it helps preserve transferred knowledge from an abrupt perturbation. Compared to the conventional [feature-tuning](#), [ENTAME](#) has two special characteristics: 1- extracted features are normalized, 2- [head](#) parameters are initialized to zeros. These characteristics are depicted in [Figure 5.2](#).

5.4.1 Feature Normalization

ENTAME is defined for working with mini-batch SGD. Before the extracted features are fed to the model’s head, z-normalization is applied on them across the batch dimension. This normalization operation can be expressed as

$$\mathfrak{A} = \frac{\mathbf{a} - \bar{\mathbf{a}}}{\sqrt{\mathbf{s}^{(2)} + \epsilon \mathbf{1}}}; \quad (5.12)$$

where $\bar{\mathbf{a}}$ and $\mathbf{s}^{(2)}$ are the mean and variance of features across the batch dimension, respectively. The subtraction, addition and division are element-wise between vectors in this equation. The scalar ϵ is a small number to avoid numerical problems. For clarity, let $\mathbb{E}_{\text{batch}}$ be the averaging across the batch dimension (**sample mean**). Then we can represent the statistics in this z-normalization operation as

$$\bar{\mathbf{a}} = \mathbb{E}_{\text{batch}} [\mathbf{a}] \quad (5.13)$$

and

$$\mathbf{s}^{(2)} = \mathbb{E}_{\text{batch}} [\langle \mathbf{a} - \bar{\mathbf{a}}, \mathbf{a} - \bar{\mathbf{a}} \rangle]. \quad (5.14)$$

We assume that the probability of the special case of $\mathbf{a} = \mathbf{0}$ is near zero as such a condition is a strong indication of domain dissimilarity, hence TL would be unlikely to outperform training from scratch.

In general, this normalization is similar to batch-normalization [Ioffe and Szegedy, 2015], except that it does not need any learnable parameters. To avoid transduction, the statistics used in inference mode of our simple z-normalization are detached (from the computational graph) version of the ones obtained in the latest training step. We discuss the role of this normalization in Section 5.5.

5.4.2 Maximum Entropy Initialization

Let $\mathbf{W}_{i,j}$ denote the element on the i -th row and j -th column of \mathbf{W} . ENTAME initializes elements of \mathbf{b} and \mathbf{W} to zero. To account for precision of the hosting machine and numeric errors, as well to qualitatively measure the importance of the head’s initialization, let us instead assume weights are drawn from a normal distribution as follows:

$$\forall i \in \{1, 2, \dots, C\}, j \in \{1, 2, \dots, Q\} : \mathbf{W}_{i,j}^0 \sim \mathcal{N}(0, \sigma^{(2)}); \quad (5.15)$$

where $\sigma^{(2)}$ is the variance of each initial value in \mathbf{W} and is chosen to be very small. Assume the same initialization for the elements of \mathbf{b} . A small $\sigma^{(2)}$ results in a near all-zero \mathbf{o} at the beginning of the training (see Equation (5.4)). The exponential function is linear around zero and hence, the softmax outputs approximately $\frac{1}{C}$ for each class regardless of the input. This is further discussed in Section 5.5 where it is also shown that as a result, the initial **entropy** of the predicted labels is maximized.

5.5 Discussion

5.5.1 Maximum Entropy Predicted Labels

Near zero initialization of model's **head** is achieved when $\sigma^{(2)}$ in Equation (5.15) is chosen to be small. Theorem 7 states that such an initialization, despite being stochastic, approximately maximizes the **entropy** of predicted labels. This is concluded from the condition for the maximum **entropy** of the predicted labels determined in Theorem 6. The variance of elements of resulting $\hat{\mathbf{y}}$, depends on the number of classes. For a large C variance of $\hat{\mathbf{y}}^0$ and so that variance of δ^0 converges to 1 (see Corollary 3).

Definition 8 (Entropy of probability simplex) *Entropy of vector \mathbf{v} subjected to $\sum_{i=1}^{i=C} v_i = 1$ and $\forall i \in \{1, 2, \dots, C\} : v_i \geq 0$, is defined as*

$$H(\mathbf{v}) := - \sum_{i=1}^{i=C} v_i \ln v_i \quad (5.16)$$

Theorem 6 (Maximum entropy of predicted labels, proof in Appendix B) *Maximum **entropy** of predicted class labels is achieved when all classes are predicted equally.*

Theorem 7 (proof in Appendix B) *ENTAME makes the expected **entropy** of predicted labels initially maximized.*

Corollary 2 (proof in Appendix B) *If $\forall i, j \in \{1, 2, \dots, C\} : \hat{\mathbf{y}}_i = \hat{\mathbf{y}}_j$ then $\langle \delta, \delta \rangle = \frac{C-1}{C}$.*

Corollary 3 (proof in Appendix B) *If $\forall i, j \in \{1, 2, \dots, C\} : \hat{\mathbf{y}}_i = \hat{\mathbf{y}}_j$, a large C implies $\langle \delta, \delta \rangle \approx 1$.*

5.5.2 Feature Normalization

Since $\sigma^{(2)}$ is very small, $\mathbb{E}_{\text{batch}} [\|\nabla^a \ell\|]$ becomes very small too (see Equation (5.10)). This is practically depicted in Figure 5.1 which reflects the magnitude of the difference in $\log \mathbb{E}_{\text{batch}} [\|\nabla^a \ell\|]$ between ENTAME and the baselines. In this Figure, **SW** stands for “small weights” and **FN** for “feature normalization”.

As we eventually aim to update the feature-extractor’s parameters, $\|\mathbf{W}\|_F$ is desired to gradually increase. When the magnitude of the features is small, increase of $\|\mathbf{W}\|_F$ is also hindered. This condition does not result in a quick transitioning from maximum entropy prediction nor the feature-extractor gets updated. As a result, the training converges slowly. On the other hand, large magnitude features can cause rapid increase in $\|\mathbf{W}\|_F$ after just a few optimization steps, which in turn results in undesired larger updates to the feature-extractor.

To alleviate this dilemma, we use a feature normalizer (as described in Section 5.4) so $\|\mathbf{W}\|_F$ is appropriately increased right after the first steps. In Theorem 8 and Corollary 4, we analytically show that except in the case where first mini-batch only contains instances of a single class, $\|\mathbf{W}\|_F^1 > \|\mathbf{W}\|_F^0$. This implies that after the first step, typically larger updates are applied to the feature-extractor parameters. Figure 6.1, demonstrates how fast $\|\mathbf{W}\|_F$ is increased for different choices of σ^2 . Besides the initialization, the learning rate plays a role in the amount and speed of $\|\mathbf{W}\|_F$ increase at the beginning of tuning. We briefly overview this effect in Section 5.6.2 of this chapter. Chapter 6 expands further on the relation between learning rate and $\|\mathbf{W}\|_F$.

Theorem 8 (proof in Appendix B) *ENTAME guarantees*

$$\|\mathbf{W}^1\|_F \geq \|\mathbf{W}^0\|_F \tag{5.17}$$

Corollary 4 (proof in Appendix B) *When all instances from the first mini-batch are from the same class, ENTAME gives*

$$\|\mathbf{W}^1\|_F = \|\mathbf{W}^0\|_F \tag{5.18}$$

5.5.3 Generalized Maximum Entropy Initialization

As shown earlier in this section, initializing \mathbf{b} and \mathbf{W} to all zeros results in maximum entropy predictions as well as no updates to the feature-extractor’s parameters at the

beginning of [feature-tuning](#). In this section, we generalize this initialization scheme and show that the mentioned properties are maintained as long as the rows of \mathbf{W}^0 stay the same and \mathbf{b}^0 has the same elements. In other words, one can choose an ordered sequence of Q random values of any magnitude, set it as initial values of each row of \mathbf{W} (see [Algorithm 1](#)) and not only get maximum entropy predictions but also zero gradient to the feature-extractor.

Algorithm 1 Generalized maximum entropy initialization

Input: \mathbf{W}

draw \mathbf{v} from any arbitrary distribution such that $|\mathbf{v}| = Q$

for $c = 1$ to C **do**

$\mathbf{W}_c \leftarrow \mathbf{v}$

end for

We express these properties in [Theorems 9](#) and [11](#), and analytically prove them in [Appendix B](#). Furthermore, in [Section 5.6](#), we experimentally compare this scheme to the one presented in [Section 5.4](#) as well as to the baselines.

Theorem 9 (generalized maximum entropy initialization, proof in [Appendix B](#)) *Maximum entropy prediction labels is achieved when $\forall c, c' \in \{1, 2, \dots, C\} : \mathbf{W}_c = \mathbf{W}_{c'}$ and $b_c = b_{c'}$.*

Theorem 10 (rows stays equal, proof in [Appendix B](#)) *Let \mathcal{C}' be the set of classes that were never observed by the model during training. If [mini-batch SGD](#) is used as the optimization algorithm, then $\forall c, c' \in \mathcal{C}' : \mathbf{W}_c = \mathbf{W}_{c'}$ and $b_c = b_{c'}$ stays true as far as no training instance from classes c and c' is observed.*

Theorem 11 (proof in [Appendix B](#)) *Given condition expressed in [Theorem 9](#) we have $\mathbb{E}_{\text{batch}} [\overset{\mathbf{a}}{\nabla} \ell] = \mathbf{0}$ for the first optimization step using [mini-batch Stochastic Gradient Descend](#) ([mini-batch SGD](#)).*

5.6 Experimental Results

5.6.1 Feature-tuning with ENTAME

The section aims to experimentally analyze the impact of [ENTAME](#)'s components, maximum entropy initialization and feature-normalization. For maximum entropy

initialization, we use the method described in Section 6.3.

Data In this Chapter, we conduct experiments on the classification tasks defined by the following datasets:

- **MNIST** [LeCun, 1998]: It includes 60,000 training and 10,000 test examples of hand-written digits. These images are 24×24 and in gray-scale. **MNIST** defines a comparably easy to classify task. The classification task is *roughly* balanced with about 6,000 and 1,000 training and test images per class, respectively.
- **CIFAR-10** [Krizhevsky et al., 2009]: It provides 5,000 training and 1,000 test images of each of the following 10 classes: “airplane”, “automobile”, “bird”, “cat”, “deer”, “dog”, “frog”, “horse”, “ship”, and “truck”. Images are sized 32 by 32 and are in **RGB** format.
- **CIFAR-100** [Krizhevsky et al., 2009]: It is similar to **CIFAR-10** in size and dimension except that its 50,000 training and 10,000 test images are equally categorized into 100 classes.
- **Caltech-101** [Li et al., 2022]: Consists of 9,146 images labeled as exclusive instances of 101 objects. Each image is roughly 300 by 300 pixels. The classification task is imbalanced.

Caltech-101 is not originally separated into train and test splits nor is balanced in contrast to the other datasets listed above. Therefore, we randomly split each of its categories into train and test subsets with 15 percent chance of drawing each image for the test subset.

Model Architectures **Off-the-shelf models** pretrained on **ImageNet** are used. The model architectures used in this experiment include **ResNet-50**, **ResNet-152** He et al. [2016], **DenseNet-121**, **DenseNet-201** Huang et al. [2017], **VGG-16**, **VGG-19** Simonyan and Zisserman [2014] and **Inception-V3** Szegedy et al. [2016].

Batch Size Among these models, **Inception-V3** requires all images to be scaled up to 229×229 . Due to limitations in our **GPUs**’ memory, a batch size of 64 is chosen for experiments corresponding to **Inception-V3**. In addition, the other models that

are trained on the [Caltech-101](#) dataset are also fed 64 images per batch owing to large image sizes in this dataset. All other models and dataset use the training batch size of 256.

Data Augmentation For images in the target tasks, each input channel is normalized with its mean and standard deviation obtained from all pixels in that channel throughout the corresponding training subset before being fed into the models. Random horizontal flipping is also applied to training images. As already indicated, in experiments where [Inception-V3](#) is used as the model architecture, images are scaled to 229×229 . For all other architectures, only [MNIST](#), and [Caltech-101](#) images are scaled to 32×32 and 200×200 , respectively. No scaling is applied to images in the rest of the settings.

Baselines The fan-in mode of the method recommended by [He et al. \[2015\]](#) is used for initializing parameters of the appended [head](#) in the base models. In contrast to fan-out mode, the fan-in mode of this method tries to preserve the variance of the data in the forward pass. In this section, we tried to unify the problem by applying similar conditions for training different models as much as possible. This by itself would show the impact of the proposed method and how universally it could help the task adaptation, even without considering micro tuning for each method.

Optimization It is important to note that the reported performance for the experiments of this chapter is not intended to be compared with the best performance one has ever come up with for the tasks under this study but rather it should be viewed as a measures to compare the methods with each other⁵. We used [Adam](#) as the optimization algorithm in the experiments of this section. For each model architecture and dataset, the best learning rate is selected between $\{10^{(-5)}, 10^{(-4)}, 10^{(-3)}\}$. In most cases, $10^{(-4)}$ showed the best overall performance in terms of the area under the test accuracy’s curve. For [ENTAME](#), the value of $\sigma^{(2)}$ is chosen to be $10^{(-12)}$ everywhere.

⁵For example, as it can be implied by comparing to experiments in [Chapter 6](#), scaling the training images up can lead to much better performance in most of the cases; however, the performance gap between [ENTAME](#) and the baselines is retained.

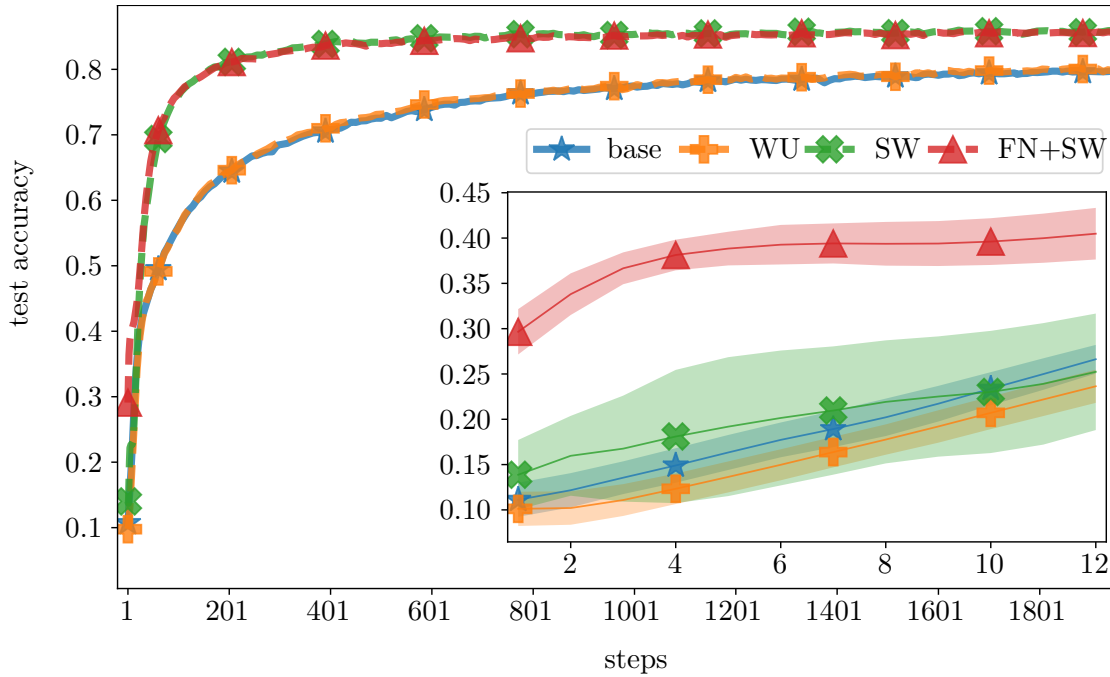


Figure 5.3: Feature-tuning ResNet-50, ImageNet \mapsto CIFAR-10.

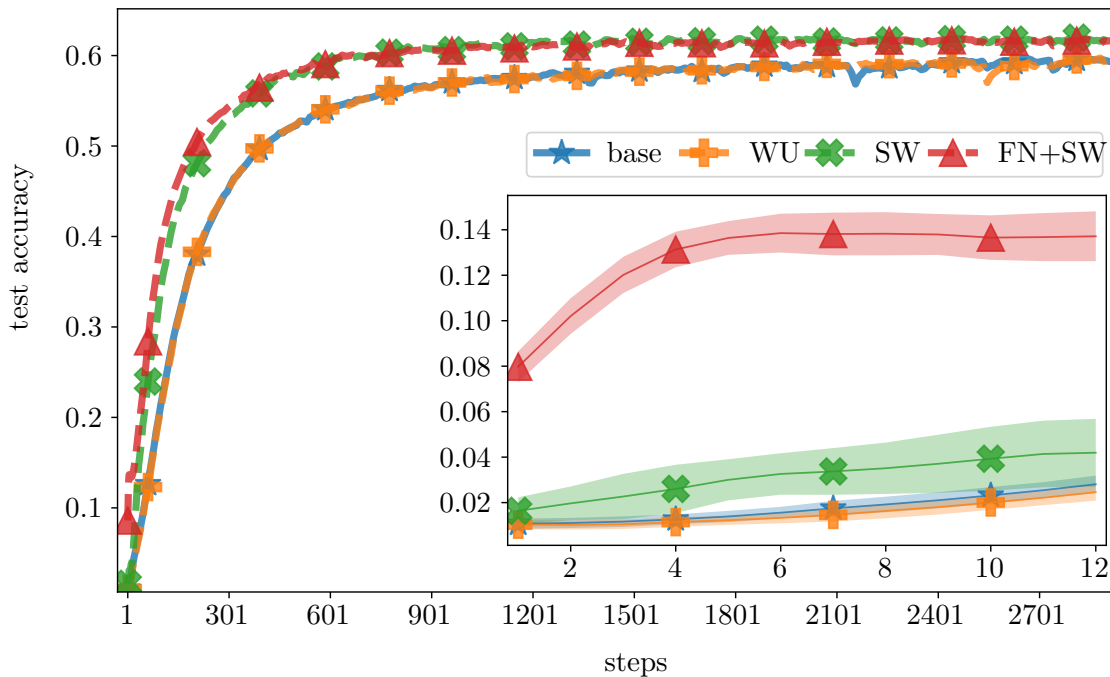


Figure 5.4: Feature-tuning ResNet-50, ImageNet \mapsto CIFAR-100.

Overall Performance Comparison Figures 5.3, 5.4 and 5.5 show the progress of test accuracy of pretrained ResNet-50 tuned on different datasets. We provide similar plots for other model architectures in Appendix B. The smaller plot inside

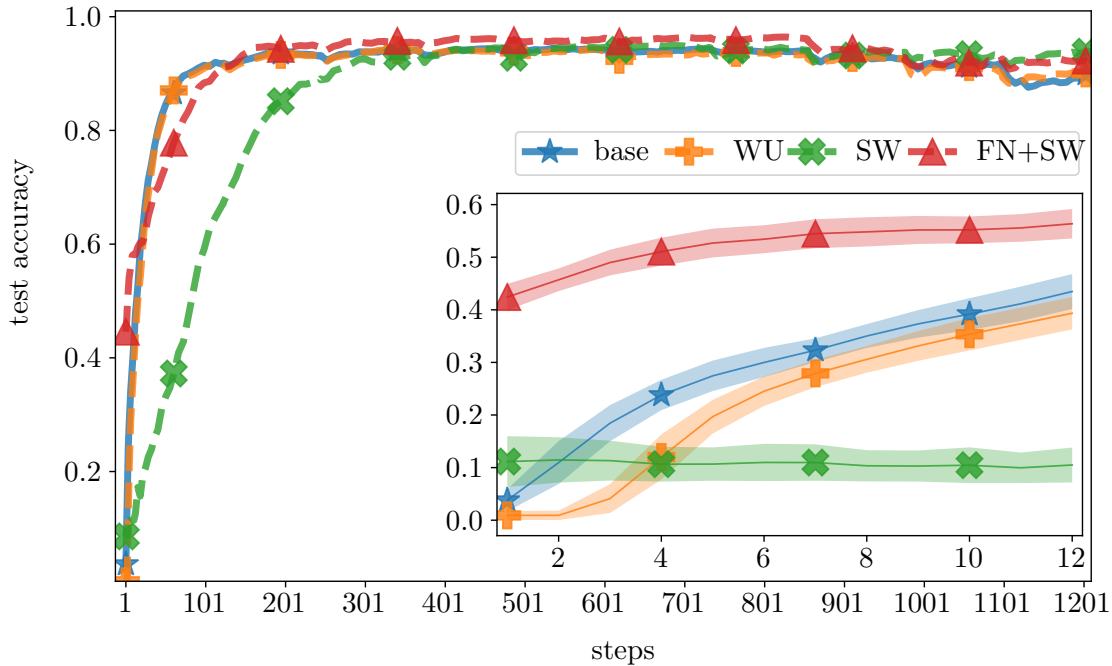


Figure 5.5: Feature-tuning ResNet-50, ImageNet \mapsto Caltech-101.

each larger one shows the same curves zoomed-in to better present the first steps of training. The shade around each curve shows the standard deviation across 24 different random seeds. Each plot includes 4 curves color mapped as follows: blue is for the base (fan-in mode of the recommended initialization in [He et al., 2015]), orange is for the base with a single Warm Up (WU) step, green is for **Small Weight (SW)** initialization for the head, red is for the full **ENTAME** or SW + **Feature Normalization (FN)**. We also conducted experiments only applying FN, but they mostly performed worse than all other cases, so they are not included for the sake of saving space and increasing the readability of the plots. *In almost all the cases, feature normalization (FN) used along with small weight (SW) initialization results in the best performance.* For many of the cases, the zero initialization is the main factor for the improvement and the features do not need to be normalized. However, *for some cases applying the feature normalization has a significant impact* (for example, see Figures 5.5).

Initial Speedup To measure how the convergence is sped up initially, we compare average test accuracy over first ten training steps. **Paired t-test** suggests that **ENTAME** significantly enhances the test accuracy compared to the base method for all

Table 5.1: Average initial test accuracy (in percent) improvement by using [ENTAME](#) instead of the base method. The entries show increase in the mean of test accuracy over first 10 steps of training with 95% confidence interval calculated over 24 random seeds.

	MNIST	CIFAR-10	CIFAR-100	Caltech-101
ResNet-50	10.86 \pm 2.97	21.81 \pm 1.10	10.19 \pm 0.32	31.29 \pm 1.21
ResNet-152	4.52 \pm 1.90	18.94 \pm 1.22	9.74 \pm 0.41	30.75 \pm 1.09
DenseNet-121	12.61 \pm 1.55	28.38 \pm 0.98	13.21 \pm 0.26	43.95 \pm 1.25
DenseNet-201	17.90 \pm 1.85	26.10 \pm 1.08	11.99 \pm 0.26	39.09 \pm 1.95
VGG-16	35.29 \pm 2.40	29.14 \pm 0.78	13.95 \pm 0.38	25.86 \pm 0.90
VGG-19	33.04 \pm 1.69	28.37 \pm 1.30	13.38 \pm 0.40	25.58 \pm 1.31
Inception-V3	9.17 \pm 1.38	33.21 \pm 2.02	8.90 \pm 0.38	31.94 \pm 1.48

Table 5.2: Convergence test accuracy of models trained on [CIFAR-10](#) dataset with 95% confidence.

	<i>Base</i>	<i>Base+WU</i>	<i>SW</i>	<i>SW+FN</i>
ResNet-50	79.73 \pm 0.43	79.74 \pm 0.76	85.80 \pm 0.35	85.54 \pm 0.20
ResNet-152	79.18 \pm 1.07	78.70 \pm 1.16	86.02 \pm 0.32	86.01 \pm 0.14
DenseNet-121	80.21 \pm 0.23	80.39 \pm 0.31	86.20 \pm 0.23	86.32 \pm 0.27
DenseNet-201	81.11 \pm 0.35	80.92 \pm 0.46	86.27 \pm 0.20	86.38 \pm 0.29
VGG-16	87.50 \pm 0.37	87.70 \pm 0.47	88.79 \pm 0.61	89.19 \pm 0.25
VGG-19	87.94 \pm 0.60	88.12 \pm 0.25	88.77 \pm 0.22	89.12 \pm 0.19
Inception-V3	95.58 \pm 0.47	95.50 \pm 0.60	95.93 \pm 0.27	95.91 \pm 0.21

architectures and datasets mentioned in this document. Table 5.1 reports the average increase in the test accuracy of the first 10 training steps with a 95% confidence interval. As implied from almost all of the cases, *careful initialization makes the feature-tuning process converge much faster*. The case of [Inception-V3](#) is an exception but there is no surprise in this difference as it converges much faster compared to other models in all the cases⁶. We have observed further improvements by adjusting $\sigma^{(2)}$, mini-batch size, and learning rate; however, to show the robustness of our model we tried to keep a unified setup as much as possible.

Converged Performance Finally, the converged accuracy of employed models is listed in Tables 5.2, 5.3 and 5.4. The convergence test accuracy is recorded after training models for 10 epochs if target dataset is [CIFAR-10](#) or [Caltech-101](#) and 15 epochs if target dataset is [CIFAR-100](#).

⁶Recall that the inputs images are scaled to 229×229 in experiments on [Inception-V3](#).

Table 5.3: Converged test accuracy of models trained on [CIFAR-100](#) dataset with 95% confidence interval.

	<i>Base</i>	<i>Base+WU</i>	<i>SW</i>	<i>SW+FN</i>
ResNet-50	59.38 ± 0.39	59.44 ± 0.41	61.50 ± 0.46	61.54 ± 0.35
ResNet-152	58.71 ± 1.37	58.50 ± 0.89	61.91 ± 0.63	61.85 ± 0.77
DenseNet-121	56.52 ± 0.46	56.70 ± 0.26	62.52 ± 0.41	62.90 ± 0.27
DenseNet-201	58.27 ± 0.62	57.98 ± 0.58	63.36 ± 0.15	63.64 ± 0.59
VGG-11	60.67 ± 0.29	60.64 ± 0.51	63.49 ± 0.34	62.88 ± 0.56
VGG-16	63.94 ± 0.24	63.77 ± 0.22	65.19 ± 0.58	64.99 ± 0.40
VGG-19	64.30 ± 0.42	64.47 ± 0.54	65.11 ± 0.28	65.02 ± 0.21
Inception-V3	82.25 ± 0.37	82.19 ± 0.21	82.17 ± 0.32	81.75 ± 0.64

Table 5.4: Converged test accuracy of models trained on [Caltech-101](#) dataset with 95% confidence.

	<i>Base</i>	<i>Base+WU</i>	<i>SW</i>	<i>SW+FN</i>
ResNet-50	89.69 ± 3.30	90.69 ± 1.12	93.87 ± 0.61	92.15 ± 1.61
ResNet-152	93.12 ± 1.01	92.98 ± 1.04	93.19 ± 0.84	93.71 ± 1.34
DenseNet-121	92.36 ± 0.67	92.41 ± 0.91	95.13 ± 1.03	95.96 ± 0.19
DenseNet-201	93.95 ± 0.53	94.01 ± 0.50	94.87 ± 1.90	96.50 ± 0.71
VGG-16	89.35 ± 1.75	91.02 ± 0.95	90.07 ± 0.68	94.69 ± 1.19
VGG-19	90.50 ± 1.42	89.96 ± 0.90	89.70 ± 1.28	90.53 ± 4.15
Inception-V3	94.98 ± 0.54	95.07 ± 0.67	95.70 ± 0.41	95.50 ± 0.89

5.6.2 Gradual Increase of the Norm of Head’s Weights

Learning rate is an important factor to make sure $\|\mathbf{W}\|_F$ properly increases and in turn feature-extractor parameters get adapted to the target task. Figures 5.7 show how [ENTAME](#) results in gradual increase in $\|\mathbf{W}\|_F$ for when [Adam](#) with learning rates of respectively, $10^{(-4)}$ is used. This corresponds to the same setup as the one used for Figure 5.4. Figures 5.6 and 5.8 provide further insight on the increase of $\|\mathbf{W}\|_F$ for when respectively smaller and larger learning rates are employed. Comparing these figures suggests that *a larger learning rate leads to saturating $\|\mathbf{W}\|_F$ to a larger value.*

5.6.3 Domain Similarity

In this section, we investigate how the level of similarity between the source and target domains impacts [feature-tuning](#) for different [head](#) initialization schemes. To control the domain similarity, we further partition the training split of a dataset into two

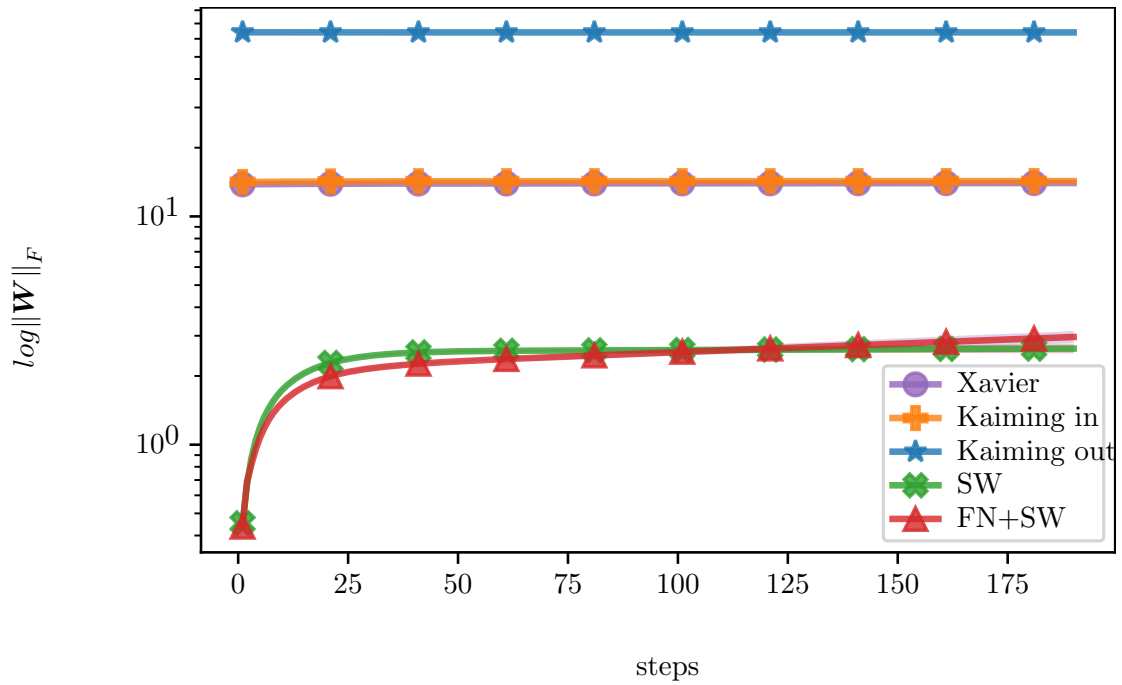


Figure 5.6: Frobenius norm of \mathbf{W} for different initialization methods in \log scale. [Adam](#) with $\eta = 10^{(-3)}$ is used. See details of this experiment in Section 5.6.2.

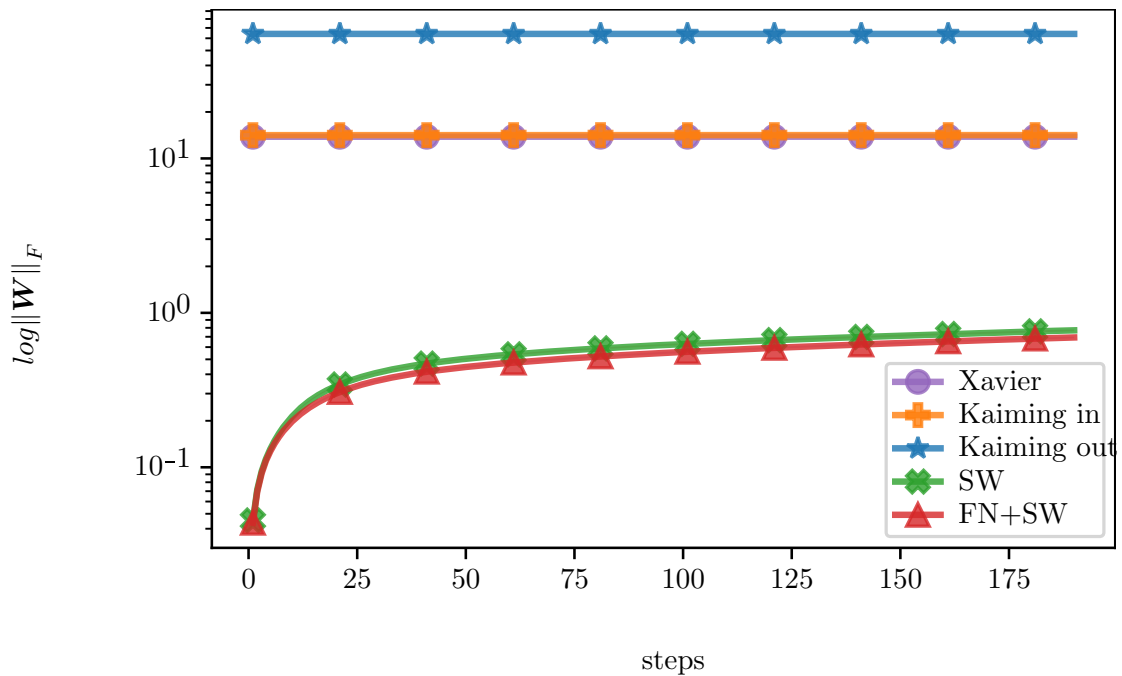


Figure 5.7: Frobenius norm of \mathbf{W} for different conventional initialization methods in \log scale. [Adam](#) with $\eta = 10^{(-4)}$ is used. See details of this experiment in Section 5.6.2.

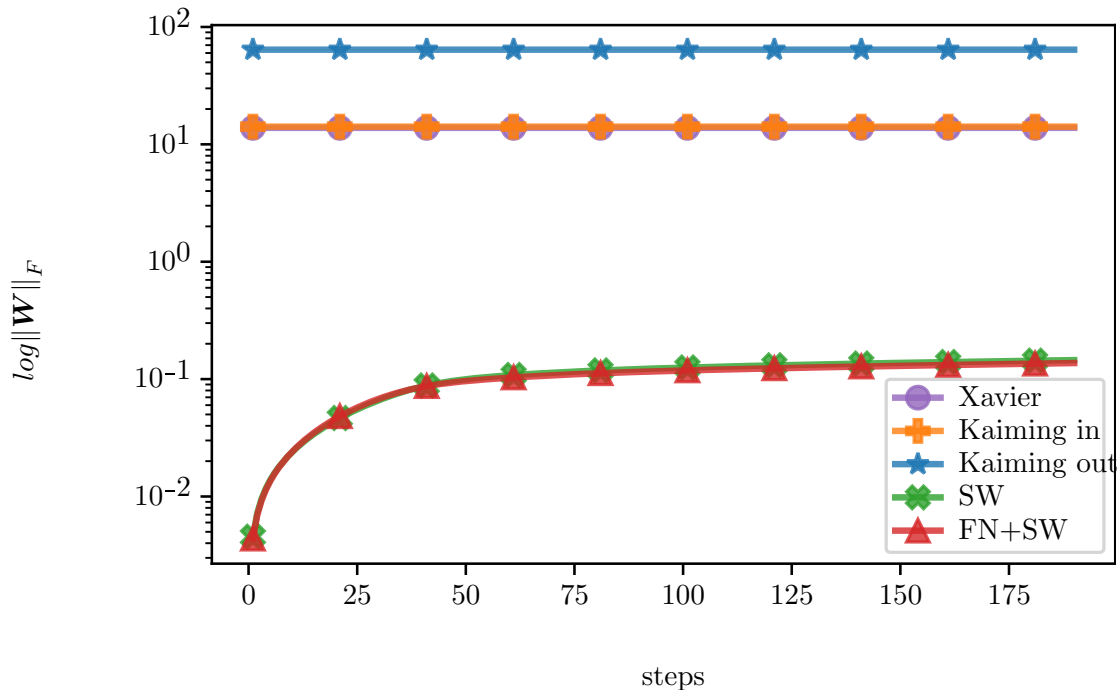


Figure 5.8: Frobenius norm of \mathbf{W} for different initialization methods in \log scale. [Adam](#) with $\eta = 10^{(-5)}$ is used. See details of this experiment in Section 5.6.2.

disjoint sets. We use one of these new partitions as the training data for pretraining and the other for [feature-tuning](#). In this way we can impose the level of domain similarity through our method of partitioning. Although the source and target tasks have the same number of classes in this setting, we still replace the transferred [head](#) before training on the target task. This is because our goal is to see the impact of head initialization and feature normalization on the target performance. Notice that source and target tasks can still be distinct if the order of classes in the target task is perturbed, yet the training is not effected mathematically.

Data In order to make sure that the data distribution varies from source domain to the target domain, we partition a dataset into two disjoint sets with an equal number of instances but different label distributions. One of these sets is used for pretraining and the other one for [feature-tuning](#). The label distribution for each partition is determined by sampling from a Dirichlet distribution with C dimensions and concentration factor α . A smaller α results in more heterogeneous samples, meaning that it is less probable for the sampled vectors to have equal elements. Each partition is filled with half of the original training data samples drawn based on its

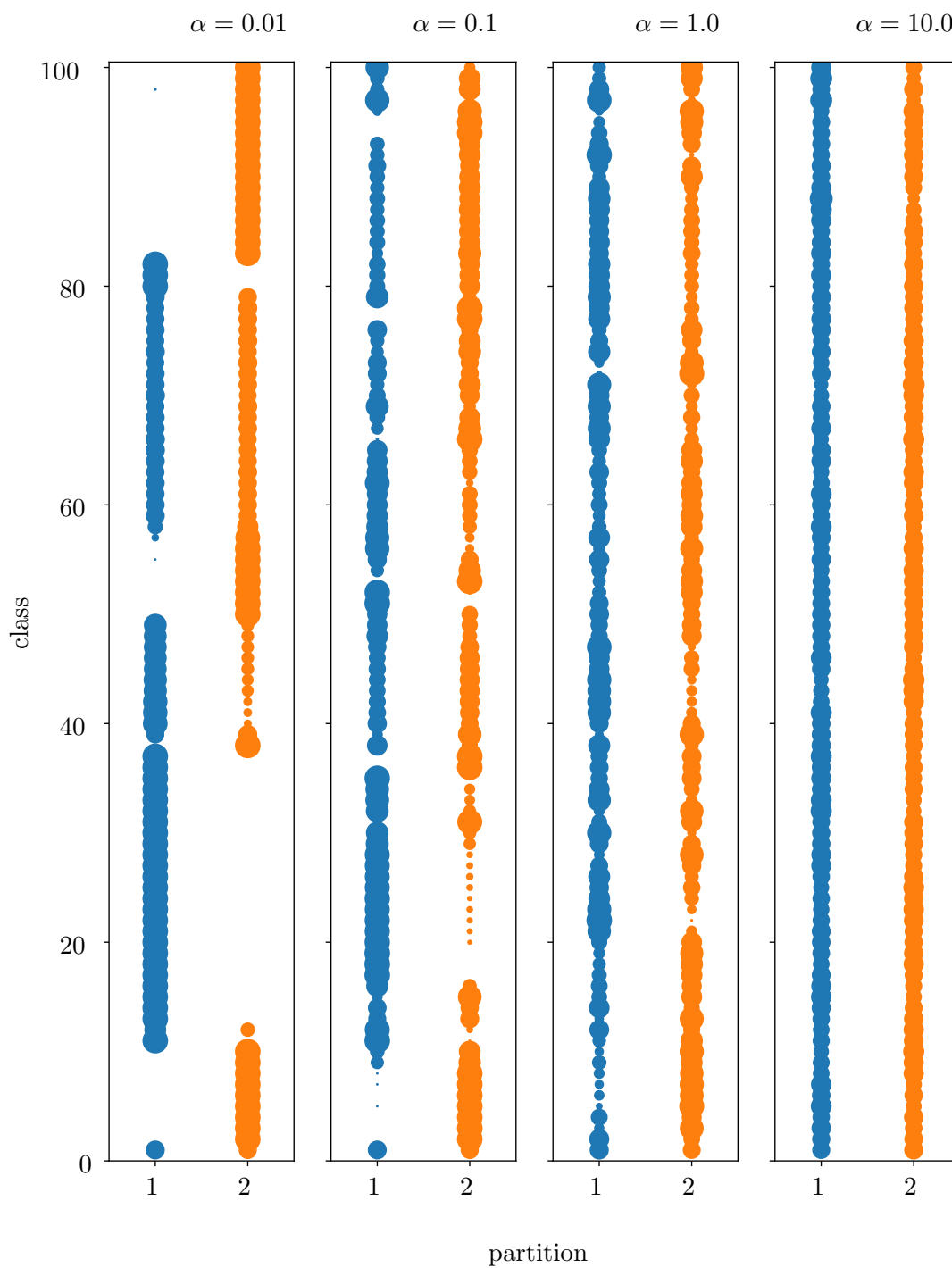


Figure 5.9: The partition distribution of each class of the training examples after partitioning [CIFAR-100](#) dataset. From left to right the concentration factor α is increased. The horizontal and vertical axes show partition numbers and class numbers, respectively. The size of the circles represents the relative number of data instances.

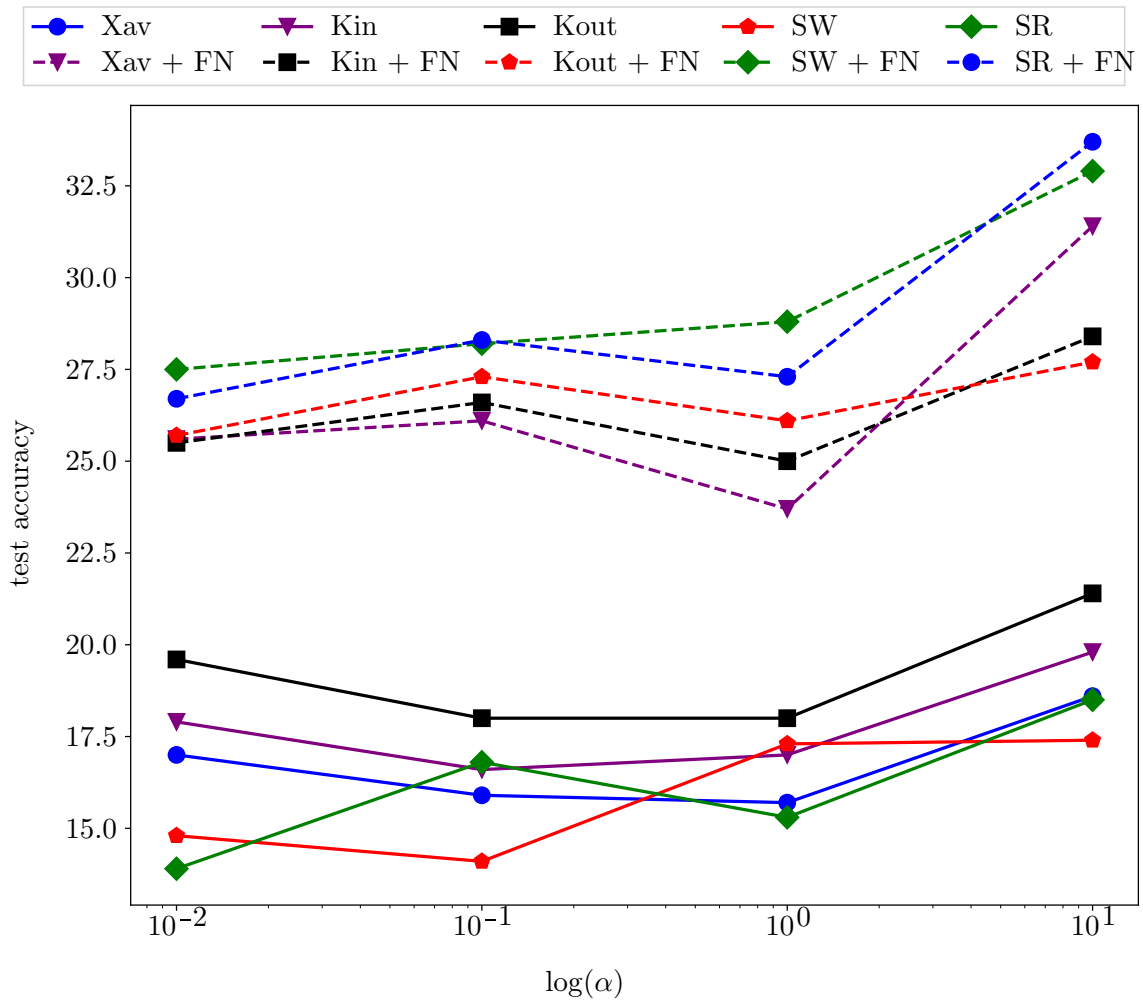


Figure 5.10: The effect of task similarity on methods under the study for [feature-tuning](#).

determined label distribution, one after another without replacement. Figure 5.11 compares the result of this partitioning scheme for different values of α . In this experiment, the size of the training batch is set at 256. The images are scaled down to 24×24 , and no data augmentation is applied.

Model architecture We use a comparably shallow [CNN](#) for this experiment. Our choice is motivated by maintaining consistency with the experiments of Chapter 7 and [FL](#) baselines. The model architecture consists of two convolutional layers with 5×5 kernel size and 64 kernels each, followed by two fully-connected layers with 394 and 192 hidden units, respectively.

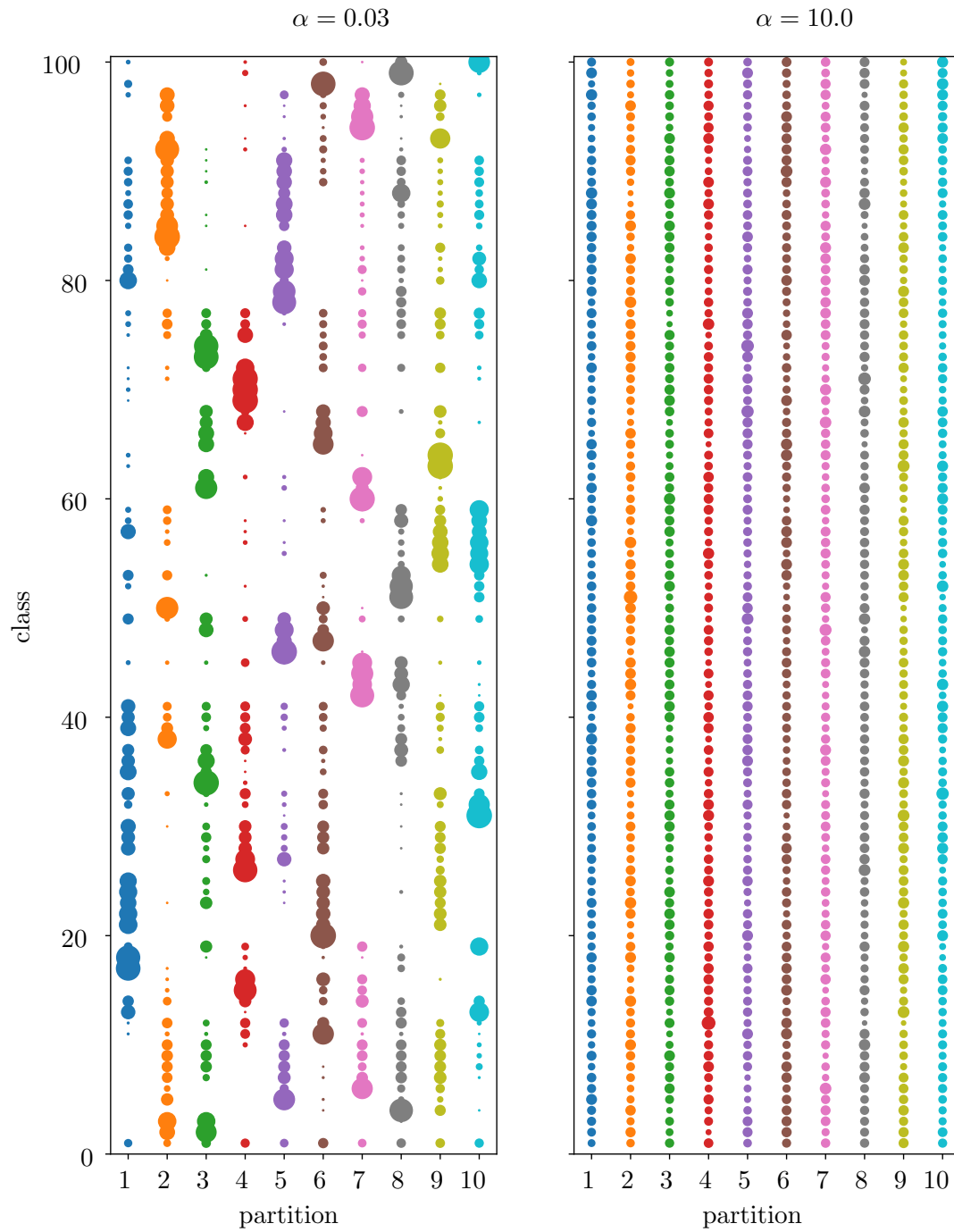


Figure 5.11: The partition distribution of each class of the training examples after partitioning [CIFAR-100](#) dataset. On the left subplot $\alpha = 0.03$ is used while on the right subplot, $\alpha = 10.0$. The horizontal and vertical axes show partition number and class number, respectively. The size of the circles represents the relative number of data instances.

Other settings For training, we use [SGD](#) with 0.9 as momentum both in pretraining and [feature-tuning](#) stages. The pretraining uses 0.001 as the learning rate. In tuning stage, we try all the learning rates in $\{0.0001, 0.001, 0.01, 0.1, 0.2, 0.5, 1.0\}$ for each method and select the setting that yields the best result. We made sure that the optimizer is re-initialized at the beginning of [feature-tuning](#) to prevent propagating extra information through momentum statistics. The pretraining is performed for 10 epochs while each training sample is visited only once during [feature-tuning](#) (single epoch). We use Xavier to initial the [head](#) for the pretraining stage.

Methods We consider the following methods for re-initializing \mathbf{W} before [feature-tuning](#): Xavier initialization (Xav) [[Glorot and Bengio, 2010](#)], the fan-in mode of Kaiming initialization (Kin) [[He et al., 2015](#)], as well as its output mode (Kout), small weights (SW) initialization as described in Section 5.4, and the generalized maximum entropy initialization introduced in Algorithm 1 or in short “Same Row” (SR) initialization. To be fair, for each case, we measure the performance with and without feature normalization (FN) layer we used in [ENTAME](#). For SR, the vector used to initialize rows of \mathbf{W} is drawn from a normal distribution $\mathcal{N}(0, 1)$.

Performance Figure 5.10 compares the test accuracy of different methods on the target domain. The dashed lines correspond to the same methods represented by solid lines with the same color and marker except that feature-normalization (FN) is applied on top of them. As seen in this figure, for all the levels of domain similarity, the best results are acquired when maximum entropy initialization (SW or SR) is used along with feature normalization. **The feature normalization significantly helps with the performance on the target task/domain.** Furthermore, the performance gap with average of the baselines gets larger when the source and target domains are almost identical (case of $\alpha = 10$).

5.6.4 Continuous Hypothesis Transfers

So far, we have discussed the efficiency of forwarding knowledge from one task or domain to another through employing pretrained models. We introduced a method that helps training a target task by adapting the forwarded knowledge more efficiently.

Given the large impact of this method on the performance and speed of convergence on the target task, some interesting questions may naturally emerge. These include:

- what is the impact of the introduced method when applied on the source learning procedure?
- how does the method work for a chain of hypothesis transfers?

This section looks for some possible answers to these questions.

To make quantitative measures possible, we consider a different setup than the one studied so far (i.e., [task adaptation](#)). In particular, we consider an episodic HTL setting where each episode involves both hypothesis transfer and training. Additionally, from one episode to another, the data distribution may significantly change. This can be considered a form of CL where the goal is to maximize the efficiency in adapting to each new distribution. Note that another popular desire in CL is to also maintain the previously learned knowledge which is not subject of this study. The task learned at each episode is the same by definition (see task definition at [2.1](#)). Therefore, the transferred [head](#) is maintained and tuned along with the feature-extractor.

Algorithm 2 An episodic training and evaluation procedure.

```

Input: model and pairs of training and test splits with domains  $\{\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_T\}$ 
/* loop of  $T$  episodes */
for  $t = 1$  to  $T$  do
  train the model on training split with domain  $\mathcal{D}_t$ 
  for  $t' = 1$  to  $T$  do
    evaluate the model on test split with domain  $\mathcal{D}_{t'}$ 
  end for
end for

```

Algorithm 2, shows our episodic train and evaluation procedure used in this section. At each episode t , we train the model on a training split with domain \mathcal{D}_t , then, evaluate it on all of the test partitions. In this setting, among the evaluation acquired at the end of episode t , we care the most about the one performed on test split with domain \mathcal{D}_t which is related to the adaptability of the model. Moreover, the evaluations on the test splits with domains in $\{\forall s < t : \mathcal{D}_s\}$ can reflect the ability of the model to maintain its previous learned knowledge.

Data In order to make sure that the data distribution varies from one learning episode to another, we partition a dataset into disjoint sets with an equal number of instances but different label distributions. We then use each of these sets in a separate episode. Similar to Section 5.6.3, the label distribution for each partition is determined by sampling from a Dirichlet distribution with C dimensions and concentration factor α . A smaller α results in more heterogeneous samples, meaning that it is less probable for the sampled vectors to have equal elements. With m total sample and n partitions, each partition is filled with $\frac{m}{n}$ data samples drawn based on the label distribution determined for that partition, one after another without replacement from the original dataset. The same sampled label distribution is employed to partition the training split of the dataset as well as its test split. Figure 5.11 compares the result of this partitioning scheme for $\alpha = 0.03$ (on the left) and $\alpha = 10.0$ (on the right). We use [CIFAR-100](#) that is partitioned on 10 sets with Dirichlet concentration factor, $\alpha = 0.03$ for the experiments of this section. The size of the training batch is set at 256. The images are scaled down to 24×24 , and no data augmentation is applied.

Other settings We use the same model architecture as in Section 5.6.3. For training, we use [SGD](#) with 0.9 as momentum for each episode keeping the learning rate identical across episodes. We made sure that the optimizer is initialized at the beginning of each episode to prevent the momentum statistics from propagating extra information. Each training sample is visited only once. In other words, each training episode is performed for a single epoch.

Methods We consider the same set of methods as of Section 5.6.3. To be fair, for each case, we measure the performance with and without feature normalization (FN) layer we used in [ENTAME](#). Additionally, we scale features to get the best we can out of each method. Whenever FN is employed too, the scalar is applied after it. For each method, we try all the feature scalar values in $\{0.025, 0.5, 1, 2, 4\}$, and all the learning rates in $\{0.0001, 0.001, 0.01, 0.1, 0.2, 0.5, 1.0\}$. For SR, the vector used to initialize rows of \mathbf{W} is drawn from a normal distribution $\mathcal{N}(0, 1)$.

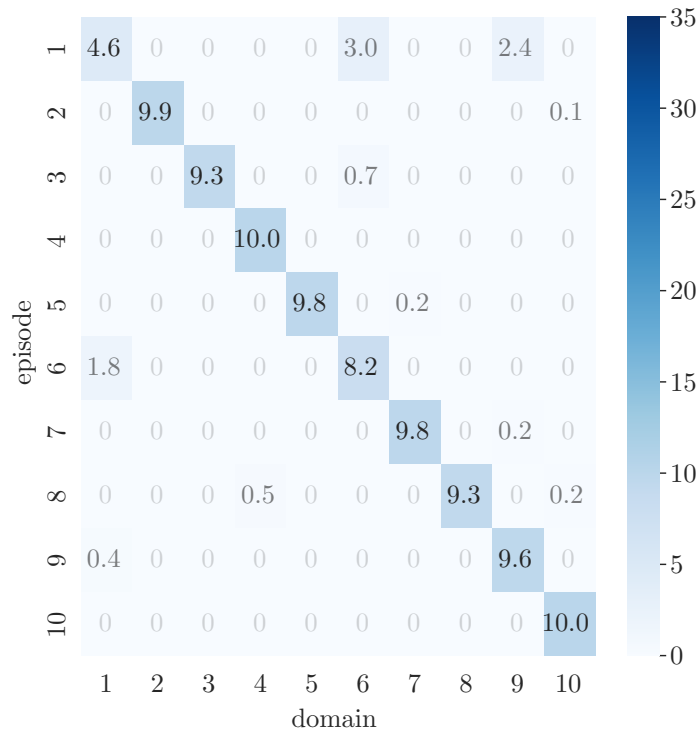


Figure 5.12: Test accuracy measured for all domain per training episode using best configuration for Xav ($\eta = 0.1$, features scaled by 1.0).

Learning and adaptability performance Figures 5.12 to 5.15 show the best evaluation outcome for Xav + FN, SW + FN, and SR + FN respectively. In these figures, rows from top to bottom reflect the sequence of training the model while the columns show evaluation result on test split of corresponding partitions. Therefore, numbers on and below diagonal are directly related to the ability of the model to learn new domains and preserve knowledge of previous domains, respectively. All other baselines (with and without feature normalization) show much poorer performance in terms of the average test accuracy on the diagonal; however, we include similar figures for them in Appendix B. An interesting observation is that in Figures 5.13 to 5.15 there is significant drop in performance in episodes 5 and 6. This can be justified by looking at the extent of label distribution shifts for these episodes correspondingly visualized on the left subplot of Figure 5.11. Figure 5.16 shows the average test accuracy on diagonal (as on diagonal of matrices in Figures 5.12 to 5.15) for all of the methods used in this experiment. Overall, from the results quantitatively presented in

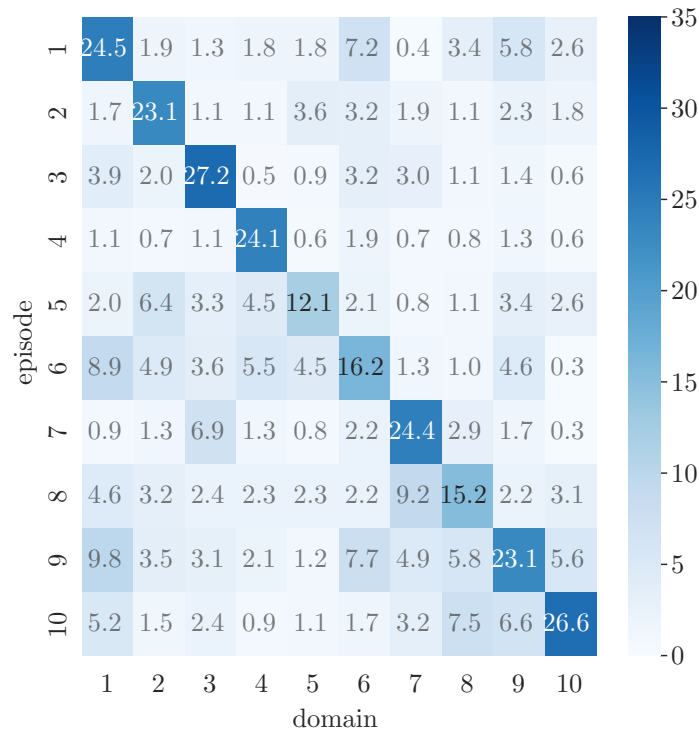


Figure 5.13: Test accuracy measured for all domain per training episode using best configuration for $\mathbf{Xav} + \mathbf{FN}$ ($\eta = 0.001$, features scaled by 2).

this section and related ones in Appendix B, the following conclusions can be inferred:

- Feature normalization (FN) can have a surprisingly large impact on the convergence speed.
- Maximum entropy initialization can further help the convergence speed.
- The generalized maximum entropy initialization has the same learning capabilities as near zero initialization, even with large magnitude of drawn values (We used $\mathcal{N}(0, 1)$).

5.7 Conclusion

In this chapter, we showed the significance of a careless initialization of model’s [head](#) for [feature-tuning](#) in [task adaptation](#). We illustrated how an incremental increase

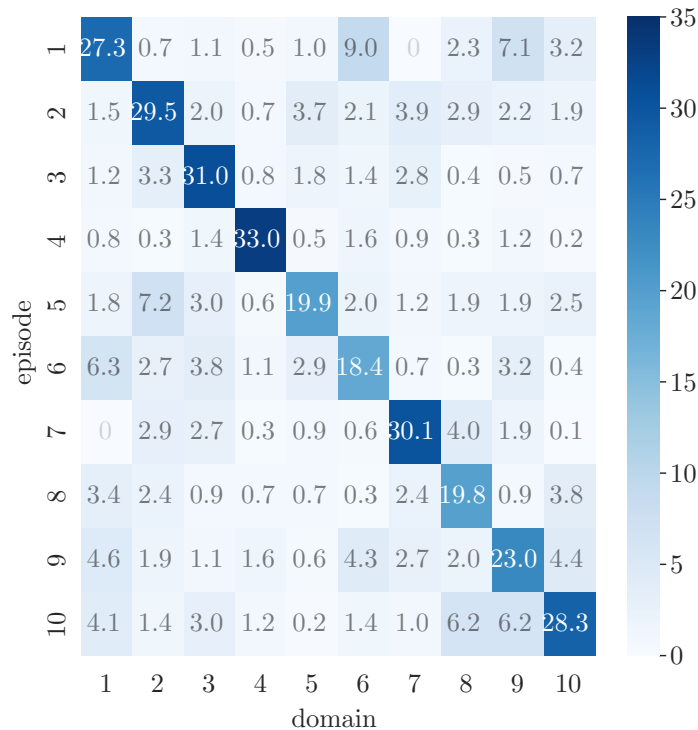


Figure 5.14: Test accuracy measured for all domain per training episode using best configuration for **SW + FN** ($\eta = 0.01$, features scaled by 2).

in the norm of gradients to a pretrained feature-extractor can improve the outcome of the training on the target task. We proposed **ENTAME**, a **task adaptation** technique to improve the speed of convergence and generalization performance through preserving the transferred features from large and noisy updates at the starting steps of **feature-tuning**. We compared our proposed method with the most conventional practices for **feature-tuning** through an extensive set of experiments. Our experiments suggest that compared to the baselines, **ENTAME** can significantly improve the speed of convergence and even the converged performance across various task and architectures. Additionally, from our theoretical and experimental results, we can conclude that

- the maximum prediction entropy can be achieved either by initializing the head’s weight matrix to (close to) zero values or the same row values.
- given maximum entropy initialization, the initial norm of the perdition error

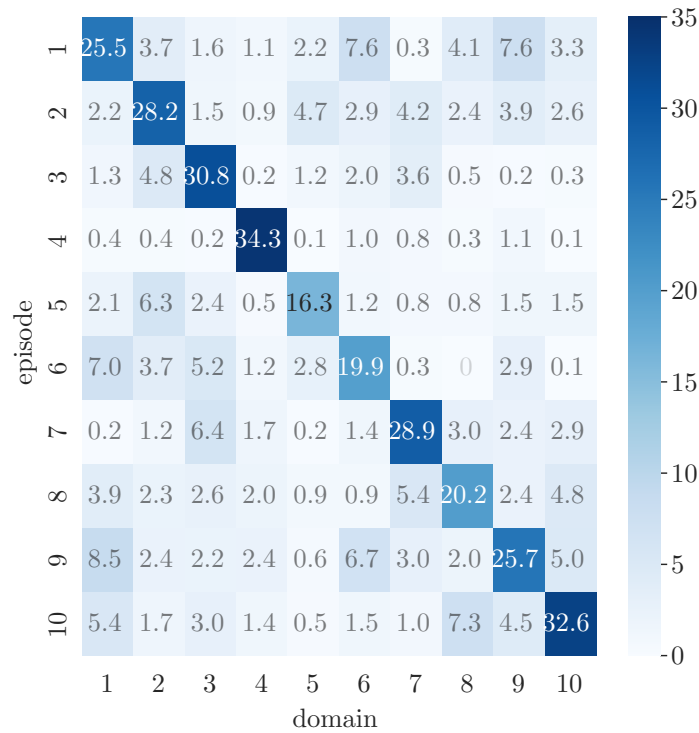


Figure 5.15: Test accuracy measured for all domain per training episode using best configuration for **SR + FN** ($\eta = 0.01$, features scaled by 4).

converges to 1 for a classification task with large number of classes;

- **ENTAME** guarantees increasing the Frobenius norm of heads weights, except when all samples are from the same class.
- maximum entropy initialization prevents the transferred features from being initially perturbed (because the gradient to feature extractor become zero mathematically).
- our proposed method works better when the source and target domains are more similar to each other.
- feature normalization can give a surprisingly large boost to **feature-tuning**.

Nevertheless, our method, **ENTAME** comes with a few caveats for certain training settings. These shortcomings stem from requiring the normalization across the batch

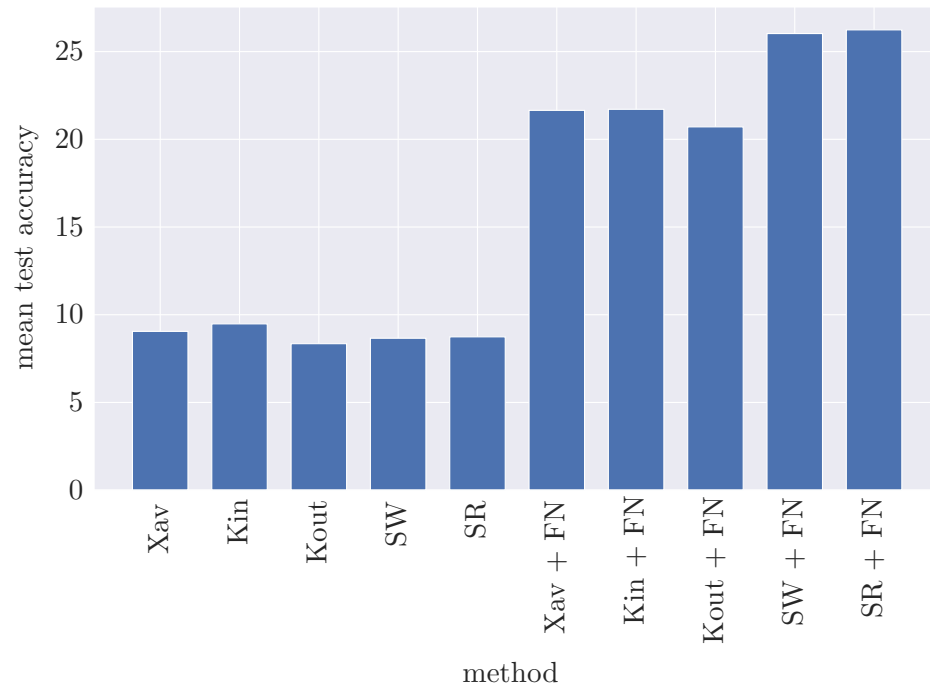


Figure 5.16: Mean test accuracy of episodes of hypothesis transfer for baselines and proposed methods. The model is trained for one epoch in each episode and learning rate is carefully tuned for each method.

dimension. We further discuss these shortcomings and try to address them in the next chapter.

Chapter 6

Incremental Tuning with Decoupled Step Sizes

”The rapidity with which we forget is astonishing.”

- Dale Carnegie, *How to Win Friends and Influence People*

In Chapter 5, we showed that initializing the parameters of the model’s [head](#) with zero or even near-zero values causes the feature-extractor to stay away from changes at the beginning of the training. We also discussed why it is desired to gradually increase the magnitude of the gradients backpropagated towards feature-extractor’s parameters throughout the first steps of [feature-tuning](#). Additionally, it was shown that, these can be achieved by increasing the Frobenius norm of the weights of the [head](#), starting from almost zero. For ensuring such increase in the norm, our proposed method, [ENTAME](#) incorporated a feature normalizer.

In this chapter, we introduce another approach to gradually increase the norm of [head](#)’s parameters. We propose [Fast And Stable Task-adaptation \(FAST\)](#) which instead of the feature-normalizer in [ENTAME](#), considers tuning a separate learning rates for the parameters of the model’s [head](#). Moreover, we take a closer look into the impact of the gradual tuning strategy used by [FAST](#) on [catastrophic forgetting](#).

6.1 Problem statement

It was shown in Chapter 5 that [ENTAME](#) significantly boosts convergence with [feature-tuning](#) across various benchmarks. However, it has a few drawbacks including

- *mini-batch size dependency*: the feature-normalization in [ENTAME](#) is performed across the batch dimension. What if only a single instance could be drawn in each batch for training?
- *output size dependency*: [ENTAME](#) ensures effective updates on feature-extractor through the increase of $\|\mathbf{W}\|_F$. This norm depends on the output size (C).

Given the same pretrained feature-extractor, a classification task with a larger number of classes may result in larger update to the feature-extractor parameters. This undermines the feature-normalization attempt to unify the rate of increase in update magnitudes. Thus, ENTAME’s impact on performance is not completely model agnostic.

- *memory efficiency*: ENTAME uses an extra normalization layer which in turn requires memory to store the mean and variance of the extracted features. The amount of required memory depends on the dimensions of the feature space (Q). This overhead can be important for edge model training where usually there are resources constraints.

All of these ENTAME’s limitations stem from normalizing the features. Can we find an alternative approach to gradually increase $\|\mathbf{W}\|_F$ but avoid the extra memory and dimensional dependencies of ENTAME?

6.2 FAST

According to Theorem 8 and Corollary 4, if the first training mini-batch contains instances of more than one class, ENTAME guarantees $\|\mathbf{W}^1\|_F > \|\mathbf{W}^0\|_F$. Interestingly, none of these are concluded from the feature-normalization in ENTAME (see Proof of Theorem 8 and Corollary 4 in Appendix B). However, normalizing \mathbf{a} , helps controlling how fast the norm of \mathbf{W} grows initially. Motivated by the shortcomings stated in Section 6.1 and the results of Section 5.6.2, we propose to independently control this growth by using separate learning rates for the head and the feature-extractor. We refer to this method as FAST.

Formally, FAST is different from the conventional feature-tuning in two ways: (i). it applies maximum entropy initialization (similar to ENTAME), (ii). it tunes head with a separate learning rate than that of the feature-extractor. Figure 6.1 shows the impact of initializing the head’s weights to small values in gradually increasing $\|\mathbf{W}\|_F$. In the corresponding experiment, using mini-batch SGD with momentum a pretrained ResNet-18 is tuned on CIFAR-100.

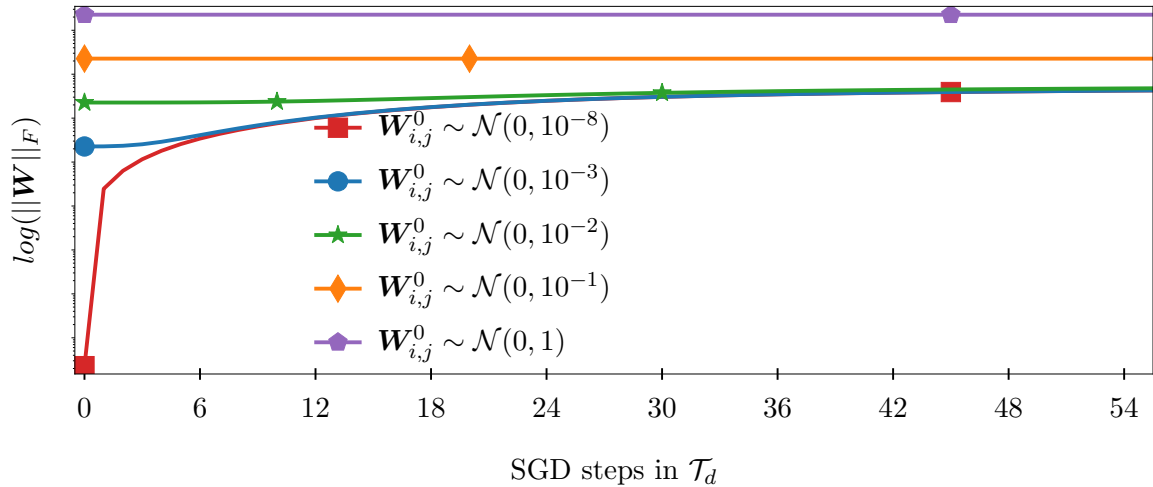


Figure 6.1: Frobenius norm of \mathbf{W} for drawing initial [head](#)’s weights from Gaussian distributions with different variances. The setup is different from the plots in [5.6.2](#). See details of this experiment in [Section 6.4](#). Note that the vertical axis is in log scale.

6.3 Discussion

6.3.1 Geometric Interpretation

To justify the impact of adopting a pretrained model, most research works rely on the concept of learning *multiple levels of representations* [[Bengio, 2012](#), [Zeiler and Fergus, 2014](#), [Goodfellow et al., 2016](#)]. According to this concept, higher-level representations contain more *abstract features* [[Bengio, 2012](#)]. These representations correspond to the layers that are closer to the input of the model [[Zhong et al., 2016](#)].

Another popular geometric view of [DNNs](#), depicts the loss landscape (see [Definition 1](#)) in the space formed by all the model’s parameters, regardless of the layers they belong to [[Li et al., 2018](#)]. This perspective has provided researchers with the intuitions behind many optimization algorithms [[Tseng, 1998](#), [Duchi et al., 2011](#), [Tieleman et al., 2012](#), [Kingma and Ba, 2015](#)].

[SGD](#) walks the parameters through the parameters’ space under the light of the gradients corresponding to the training data. It settle parameters in a minimum point of the loss landscape, and hopefully this point also is a minimum for the loss landscape corresponding to the true distribution of the data.

In order to employ the latter geometric view for understanding the concept of [task](#)

adaptation, answering a fundamental question seems to be necessary: *in the space formed by the model’s parameters, how much and in what direction should SGD drive the parameters to quickly adapt a pretrained model to a target task?* In other words, in the aforementioned space, *how much does the minimum of the target task’s loss deviate from that of the source task and what is a good trajectory to quickly approach it?*

On our way to search for an answer to this question, let the Euclidean space formed by the parameter set \mathbf{v} be $\mathfrak{S}(\mathbf{v})$. Notice that, each point on the landscape of the loss that is depicted in parameter space $\mathfrak{S}(\mathbf{v})$ is in fact, a point in $\mathfrak{S}(\{\mathbf{v}, \mathcal{L}\})$ with the loss (\mathcal{L}) being the only non-parametric dimension in this space. As mentioned in Section 5.1, to adapt a pretrained model for a target classification task, the space of all model’s parameters are manually changed to $\mathfrak{S}(\{\phi, \mathbf{W}, \mathbf{b}\}) = \mathfrak{S}(\theta)$. So, during the model reconstruction, not only the number of dimensions in the parameter space may change, but also the appended parameters are generally not relevant to either of the tasks (the source task and the target task) before feature-tuning starts. In fact, right after appending new head’s parameters, spotting the minimum of the source task’s loss—where the pretraining has landed in—becomes difficult.

To make it possible to exploit the geometric view of the loss landscape for TL, we only consider loss landscape corresponding to the feature-extractor’s parameters, $\mathfrak{S}(\phi)$. This relaxed view of the loss landscape enables our analysis to link the loss landscapes of the two tasks at the expense of having more complicated view of the loss landscape of the target task. In this scheme, the modifications made in the appended head’s parameters are reflected as deformation of the loss landscape in the space of the pretrained parameters. More formally, moving in $\mathfrak{S}(\mathbf{W}, \mathbf{b})$ is reflected as changes along the dimension \mathcal{L} of $\mathfrak{S}(\phi, \mathcal{L})$. This perspective is flexible enough to symbolically describe the effect of different TL strategies as it is shown in Figure 6.2 and Figure 6.3. The intuition provided in these figures is novel in the sense that it can reflect updates on θ in $\mathfrak{S}(\phi)$. It relaxes the complexity in depicting a heterogeneous space formed by both the pretrained feature-extractor’s and appended head’s parameters, and opens up the opportunity to geometrically link the source task’s and target task’s loss landscapes.

In each one of the subplots in Figure 6.2 and Figure 6.3, a model pretrained

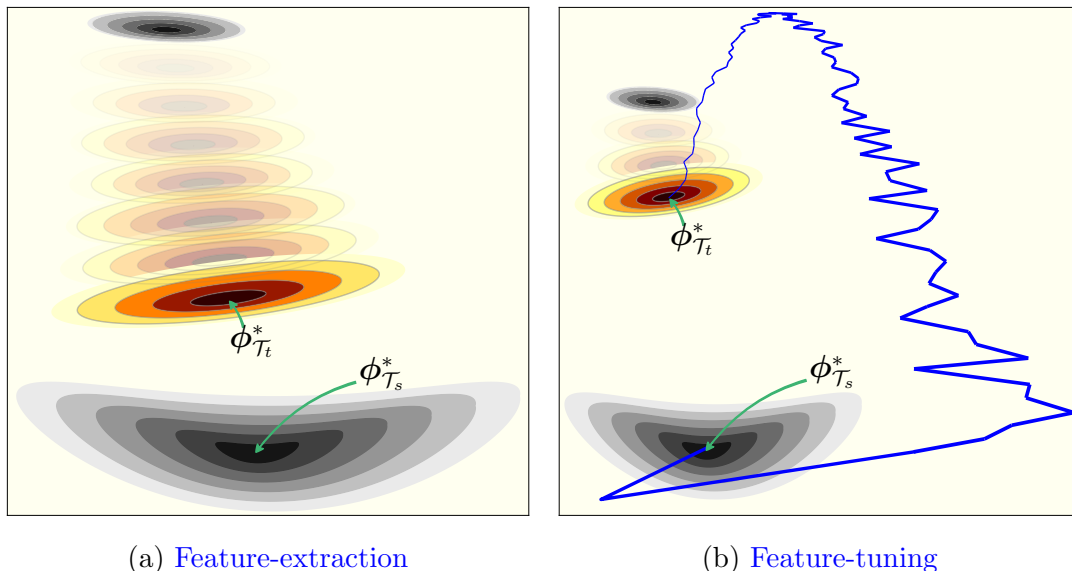


Figure 6.2: Conceptual comparison between conventional **task adaptation** methods in the loss landscape of feature-extractor parameters. Updating appended **head** parameters is depicted as moving the landscape of the target loss. The blue lines represent modifying ϕ .

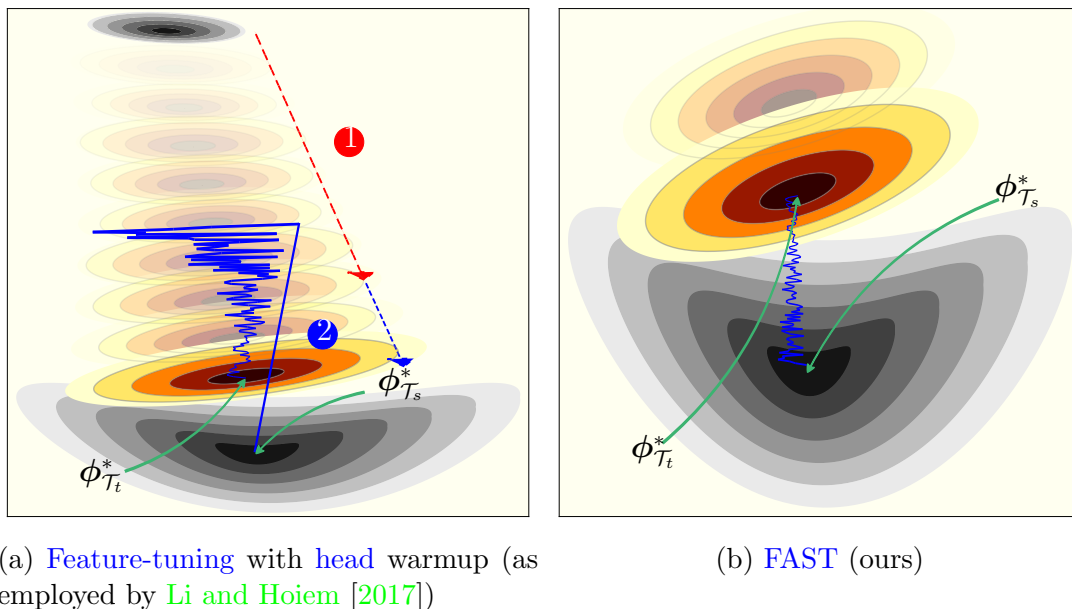


Figure 6.3: Conceptual comparison between **feature-tuning** with **head** warmup and **FAST** in the loss landscape of feature-extractor parameters. Updating appended **head** parameters is depicted as moving the landscape of the target loss. The blue lines represent modifying ϕ .

on task \mathcal{T}_s is adopted to accelerate learning the target task \mathcal{T}_t . The center of the gray contours at the bottom, marked with $\phi_{\mathcal{T}_s}^*$, represents the minimum in the loss landscape of \mathcal{T}_s where ϕ settles in, at the end of the pretraining. Similarly, the aiming minimum in the loss landscape of \mathcal{T}_t is shown with $\phi_{\mathcal{T}_t}^*$, though its location in the space is subjected to change because the appended parameters are modified by the optimization algorithm. Thus, to make a clearer view, changes in the location of $\phi_{\mathcal{T}_t}^*$ is shown by multiple sets of contours with a gray-level set at the top to indicate the initial location and is following by several color-tempered sets which represent the deformations made in the loss landscape.

We simplified the landscape deformations with simple affine transformations (shift, scale and rotation) applied on the minimum; however, many other kinds of transformations are also possible. The levels of the loss landscapes are shown with differently tempered colors. The black is the coldest color used to show the minima whereas ivory is used to describe the high altitude surface (with large loss values). Notice that, the chosen shapes are symbolic to simply show the effect of each method; otherwise, the minima valleys can be of any arbitrary shape [Skorokhodov and Burtsev, 2019, Czarnnecki et al., 2019].

Starting from $\phi = \phi_{\mathcal{T}_s}^*$, the goal of the **task adaptation** is to make ϕ get as close as possible to $\phi_{\mathcal{T}_t}^*$. Figure 6.2a describes **feature-extraction** (e.g., employed by Donahue et al. [2014]) in which only the appended parameters are trained and thus $\phi = \phi_{\mathcal{T}_s}^*$ stays hold during the **task adaptation**. On the other hand in **feature-tuning** (e.g., employed by Girshick et al. [2014]) shown in Figure 6.2b, both transferred and appended parameters are jointly trained. The blue curve represents the updates on ϕ in $\mathfrak{S}(\phi)$ corresponding to the steps taken by the optimization algorithm.

As shown in Figure 6.2, compared to **feature-extraction**, **feature-tuning** can potentially guide ϕ closer to $\phi_{\mathcal{T}_t}^*$. However, as depicted in Figure 6.2b, carelessly initializing \mathbf{W} and \mathbf{b} can mislead the optimization algorithm to guide ϕ in a wrong direction at the beginning of **feature-tuning**. If the initial steps be large, they take ϕ far away from $\phi_{\mathcal{T}_t}^*$ and so postpone the convergence. One possible solution employed by Li and Hoiem [2017] is to start with a warmup phase within which only \mathbf{W} and \mathbf{b} are modified and ϕ is kept unchanged before they jointly are trained as in the typical **feature-tuning**. This approach is depicted in Figure 6.3a where the arrows and circled

numbers determine the sequence of the applied changes.

The inclusion of a [head](#) warmup phase prevents \mathbf{W} from randomly distorting the back-propagating gradients and, therefore, the initial modifications on ϕ become more likely to be in a path that leads to $\phi_{\mathcal{T}_t}^*$. However, since the distance from $\phi_{\mathcal{T}_t}^*$ is not predetermined by the first-order [GD](#) algorithms, it is also likely that the initial steps for ϕ overshoot the minimum. Clearly, it is not easy to find an optimal step size for these updates considering the commonly large number of dimensions in $\mathfrak{S}(\phi, \mathcal{L})$. [FAST](#), our proposed method for [feature-tuning](#) builds $\phi_{\mathcal{T}_t}^*$ close to $\phi_{\mathcal{T}_s}^*$ without notably misdirecting ϕ at anytime from beginning until the convergence. The geometrically interpretation of [FAST](#) method is shown in [Figure 6.3b](#).

The [softmax](#) function normalizes its inputs such that they sum up to one. The magnitude of its inputs is exponentially reflected in the discrepancy among its outputs. The negative log-likelihood loss that is applied on top of [softmax](#), is notably affected by this characteristic. By initializing the [head](#)'s weights to values that are closer to zero, the loss becomes further independent from the extracted features. Equivalently, as much as the entries of \mathbf{W}^0 and \mathbf{b}^0 are selected closer to zero, the landscape of the loss in ϕ becomes flatter, until a point where it becomes level and loss becomes equal to $\ln(C)$ everywhere (recall that C is the number of classes). In this situation the loss becomes almost independent of the extracted features. That is, for all possible values of ϕ in $\mathfrak{S}(\phi, \mathcal{L})$ we have $\mathcal{L} = \ln(C)$. For this reason, the initial state of the minima in the target task's loss landscape is not shown with gray colors in [Figure 6.3b](#) unlike other plots in [Figures 6.2](#) and [6.3](#).

A flat loss landscape in the space of ϕ is much easier to deform. If done carefully, it can be immediately deformed such that a minimum is placed close to the current state of ϕ (this can be implied from the arguments in [Section 6.3.2](#)). In fact, instead of finding a solution to increase the relative speed between ϕ and $\phi_{\mathcal{T}_t}^*$, this strategy makes it possible to initially find a minima close to where ϕ is located in $\mathfrak{S}(\phi)$.

6.3.2 Velocity Analysis

Ideally, [SGD](#) drives the feature-extractor parameters (ϕ) toward a good minimum in the target task's loss landscape located at $\phi_{\mathcal{T}_t}^*$, and at the same time, through

modifying \mathbf{W} it guides $\phi_{\mathcal{T}_t}^*$ toward ϕ in the opposite direction.

$$V(\phi^t) = -\eta_\phi \mathbb{E}_{\text{batch}} \left[\frac{\partial \ell}{\partial \phi} \right]. \quad (6.1)$$

This leads to Theorem 12 based on which we can express the following statements.

1. Both the learning rate of the feature-extractor and the learning rate of the [head](#) linearly scale the velocity of changing the feature-extractor’s parameters during the optimization. Therefore, in conventional [feature-tuning](#) where these learning rates are set to the same values, feature-extractor parameters are updated with a quadratic order of the learning rate.
2. The noise in \mathbf{W}^0 remains in all of the updates to the feature-extractor parameters until its effect is canceled out by the head’s updates.

By choosing small values for elements of \mathbf{W}^0 and choosing a proper η_w , **(i)**. the noise in feature-extractor updates is smaller, **(ii)**. the [prediction error](#) (δ) is more informative, **(iii)**. and, the [head](#)’s updates can faster cancel out the noise in \mathbf{W}^0 (by learning).

Definition 9 (Optimization Velocity) *For a set of parameters, at a given optimization step, the optimization velocity is defined as the negative of the [pseudo-gradient](#) for those parameters at that step (see Definition 3).*

Theorem 12 (proof in Appendix C) *If [GD](#) or [SGD](#) is used to optimize the head parameters, the optimization velocity of the feature-extractor parameters is*

$$V(\phi^{t+1}) = \eta_\phi \eta_w \sum_{\tau=1}^{\tau=t-1} \mathbb{E}_{\text{batch}} \left[\delta^\tau \langle \mathbf{a}^\tau, \delta^\tau \rangle \frac{\partial \mathbf{a}^\tau}{\partial \phi^\tau} \right] - \eta_\phi \mathbb{E}_{\text{batch}} \left[\delta^t \mathbf{W}^0 \frac{\partial \mathbf{a}^t}{\partial \phi^t} \right]; \quad (6.2)$$

where η_ϕ and η_w are learning rates of the feature-extractor and the [head](#), respectively.

6.3.3 Optimization Algorithm

Despite its merits, [Adam](#) has been sometimes criticized for asymptotically being outperformed by [mini-batch SGD](#) with [momentum](#) [[Keskar and Socher, 2017](#), [Reddi et al., 2018](#)]. Dampening the gradients along the dimensions with a high frequency [Zeiler \[2012\]](#), [Tieleman et al. \[2012\]](#), [Kingma and Ba \[2015\]](#) has been sometimes found

to gain over [mini-batch SGD](#) only at the beginning of the training [Keskar and Socher \[2017\]](#) which corresponds to moving across regions of the loss landscape with high altitudes. These regions are often more chaotic compared to regions closer to the minima [Li et al. \[2018\]](#). Motivated by these facts, we will empirically show that in the case of [feature-tuning](#) on a similar task, the gap between the performance of [Adam](#) and [mini-batch SGD](#) with [momentum](#) can be significantly reduced when using [FAST](#). This mainly comes from the fact that, unlike the traditional [feature-tuning](#), optimizing the classification objective with [FAST](#) does not make parameters to step out of the proximity of $\phi_{\mathcal{T}_s}^*$ (the minimum in the loss landscape of the target task) far onto the high altitudes of the loss landscape

[Vaswani et al. \[2017\]](#) and [Popel and Bojar \[2018\]](#) showed that a warmup phase for the adaptive learning rate can significantly accelerate the training convergence¹. This phase is applied at the beginning of the training during which a very small learning rate is applied. [Liu et al. \[2019\]](#) found the source of this phenomenon in the large initial variance of the gradients. They demonstrate this by showing that the distribution of the gradients has momentous changes in the first few optimization steps and they addressed the issue by an optimization algorithm that dampens the variance accordingly. Similarly, [Luo et al. \[2018\]](#) and [Zhang et al. \[2019\]](#) proposed to set dynamic boundaries on the magnitude of the updates. [Keskar and Socher \[2017\]](#) proposed starting training with [Adam](#) to take advantage of the convergence speed and then switching to [mini-batch SGD](#) with [momentum](#) to better generalize when converged. Our proposed optimization algorithm for [feature-tuning](#) also decreases the initial variance of the gradients and similarly accelerates the training convergence.

Using [FAST](#), neither the speed of convergence nor the generalizable performance when converged are sacrificed. [FAST](#) finds its merits in controlling the velocity of applying updates only via the first gate that gradients back-propagate through, the [head](#). Although our work focuses only on classification, the analysis we provide suggests that in an already stable model (e.g., pretrained on a similar task, so at comparably low altitude regions of the loss landscape), [mini-batch SGD](#) with [momentum](#) is able to show a competent convergence speed as long as the pretrained parameters are not disrupted by sudden and large-variance updates. Such updates can cause

¹This is different from the [head](#)'s warmup phase in which only the [head](#) is updated.

overshooting the minimum of the loss landscape that they aim to converge. It is worth mentioning that label smoothing [Szegedy et al., 2016] is also relevant in the sense that it reduces the initial variance of the gradients by decreasing the 2-norm of the true labels. However, our method focuses on the norm of the predicted labels for which the magnitude could be automatically adapted during the course of training (unlike the norm of the true labels).

6.4 Experiments

6.4.1 Catastrophic Forgetting

In this chapter we restrain our focus to classic [task adaptation](#); therefore, we only concern about forgetting in a single pass hypothesis transfer (and not a chain of transfers). Although by proposing [FAST](#), we mainly aim to accelerate the [feature-tuning](#) process on the target task without compromising its convergence performance, as a by-product the source task is also forgotten slower than in atypical [feature-tuning](#) procedure. Interestingly, in contrast to the previous studies, [FAST](#) does not make an explicit effort to retain the performance of the source task and does not compromise the performance on the target task. The aim of this section is to quantitatively measure forgetting that is caused by a careless initialization of the [head](#)'s parameters and its connection to the speed of converging to the target task.

To cover engagement of features in different representation levels, we consider two scenarios: first, choosing identical domain and task for the source and destination, and second, choosing them to be unrelated and distinct. A [ResNet-18](#) [He et al., 2016] that is already pretrained on [ImageNet](#) [Deng et al., 2009] is used for this experiment. Five percent of the training split of each task is separated for validation in a stratified order. Training, validation, and test images are resized to 128×128 . Random horizontal flip is applied on images of the training mini-batches. The size of each training mini-batch is 100 with an equal number of images per label. [SGD](#) is used with a learning rate of 0.01 and a gradient [momentum](#) equal to 0.9. The number of gradient updates (steps) between two consecutive validations is increased gradually starting from 1 and saturating at 10 steps. Test checkpoints are set to the update numbers corresponding to the *Fibonacci sequence*, except for the third number which

is skipped because it is equal to the second number in the sequence. The validation is used to store the best performing model up to the step that test performance is measured. We refer to the test performance measured in this fashion as *progressive test accuracy*.

The source task is chosen to be classification defined over the [CIFAR-100](#) dataset [Krizhevsky et al. \[2009\]](#) (see details of [CIFAR-100](#) in Section 5.6). \mathbf{W} is initialized according to the fan-in mode of Kaiming’s method [He et al. \[2015\]](#) and trained for 120 epochs. In the first scenario, we choose $\mathcal{T}_s = \mathcal{T}_t$ and $\mathcal{D}_s = \mathcal{D}_t$; however, the model’s [head](#) is randomly initialized for the target task. The second scenario, chooses \mathcal{T}_t to be the classification defined over the [MNIST](#) dataset [LeCun \[1998\]](#) (refer to Section 5.6 for detail of [MNIST](#)). Multiple sets of models are tuned, initializing elements of \mathbf{W} by drawing from zero-centered normal distributions with different standard deviations ranging from 1 down to 10^{-8} for each set of models. Each experiment is repeated using a set of 15 different random seeds. In our setup, applying different random seeds not only initializes \mathbf{W} differently but also exposes dissimilar sequences and combinations of training mini-batches to the model².

During [feature-tuning](#), the model is tested on both the source and target tasks at determined checkpoints (the mentioned pseudo-Fibonacci sequence). To test the model on the source task in this experiment, the test split of the source task is first fed to the feature-extractor in its inference mode, then it is passed through the original [head](#) of the model that was trained on the source task.

Scaling down the magnitude of the initial values of \mathbf{W} not only increases the speed of convergence to the target task, but also retains more knowledge from the source task. This is concluded from the progressive performance measurement on these tasks shown in Figure 6.4. Although the rising $\|\mathbf{W}\|_F$ in some curves shown in Figure 6.5 indicates that ϕ is considerably modified (unlike [feature-extraction](#) where $\eta_\phi = 0$) after just a few steps, the corresponding performance of the source task plotted in the second row of Figure 6.4 suggests that almost no forgetting happens when source and target domains and tasks are identical, and $\mathbf{W}_{i,j}^0 \rightarrow 0$. Even though catastrophic forgetting occurs for when the source and target domain are significantly dissimilar (see Figure 6.6), it is less severe if \mathbf{W} is initialized with small numbers.

²To study further about the influence of initialization and the order of mini-batches see [McCoy et al. \[2020\]](#).

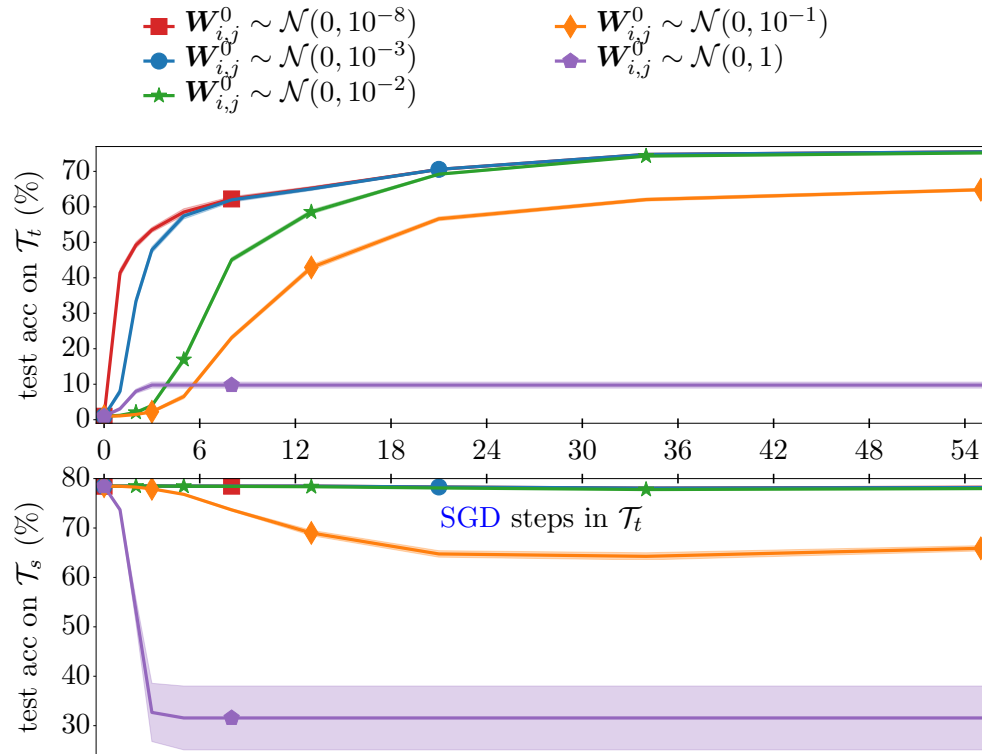


Figure 6.4: Understanding the effect of initializing \mathbf{W} through *catastrophic forgetting*. Progressive test accuracy accuracy on \mathcal{T}_t at the top and \mathcal{T}_s at the bottom. Both tasks are identical and defined by classification on **CIFAR-100** dataset. The horizontal axis shows the optimization steps on the target task which is shared among the two plots.

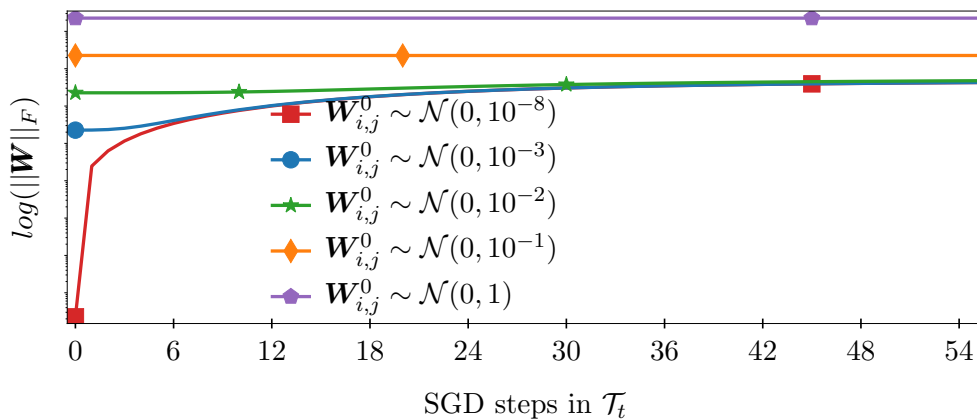


Figure 6.5: The development of $\log(\|\mathbf{W}\|_F)$ corresponding to *feature-tuning* on \mathcal{T}_t shown in Figure 6.4.

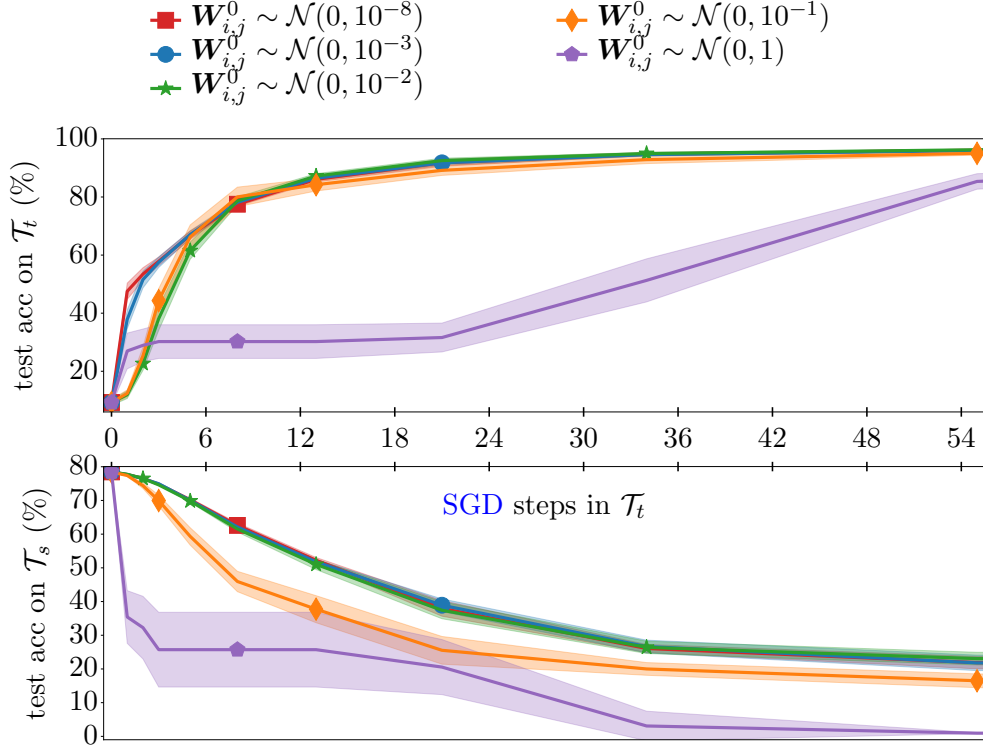


Figure 6.6: Understanding the effect of initializing \mathbf{W} through catastrophic forgetting. progressive test accuracy on \mathcal{T}_t : [MNIST](#) at the top and \mathcal{T}_s : [CIFAR-100](#) at the bottom. The horizontal axis shows the optimization steps on the target task which is shared.

6.4.2 Quick Head Learning

Geometrically, until any step t that

$$\forall i \in \{1, 2, \dots, C\} : \|\mathbf{W}_i^0\| \gg \eta_\phi \left\| \sum_{\tau=1}^{\tau=t} \mathbb{E}_{\text{batch}} [\delta_i^\tau \mathbf{a}^\tau] \right\| \quad (6.3)$$

holds, the landscape of the target task’s loss does not deform much in $\mathfrak{S}(\phi)$ (the target minimum in Figure 6.2 stays almost held for that period). This is because using [mini-batch SGD](#), we have

$$\forall i \in \{1, 2, \dots, C\} : \mathbf{W}_i^1 = \mathbf{W}_i^0 - \eta \sum_{\tau=1}^{\tau=t} \eta_\phi \mathbb{E}_{\text{batch}} [\delta_i^\tau \mathbf{a}^\tau]. \quad (6.4)$$

In this situation, if elements of \mathbf{W}^0 are carelessly drawn, it is very likely that the trajectory through which SGD guides ϕ , becomes long and hard to settle. In this experiment, we extremely slow down $V(\mathbf{W})$ by choosing a near zero η_w . This makes

us able to abstractly perceive the effect of the fast geometric deformation which FAST provides.

We compare the progressive test accuracy when randomly-initialized \mathbf{W} is kept unchanged (i.e., $\eta_w = 0$) during tuning compared to when it is updated with the same rate as ϕ (i.e., $\eta_w = \eta_\phi$). For both cases we choose $\mathbf{W}_{i,j}^0 \sim \mathcal{N}(0, 10^{-2})$. The rest of the setup is identical to the first scenario of the previous experiment except that feature-tuning is done on CIFAR-100 as the target task by adapting an off-the-shelf model pretrained on ImageNet’s [Deng et al., 2009].

Figure 6.7 suggests that when the loss landscape deforms as discussed in Section 6.3.1, the convergence is significantly accelerated. The shadows representing 95% confidence intervals are barely visible indicating that the results shown in this figure are highly consistent. The results in this experiment strongly support our hypothesis that *the deformation of the loss landscape in the space of the transferred parameters largely influences the convergence speed.*

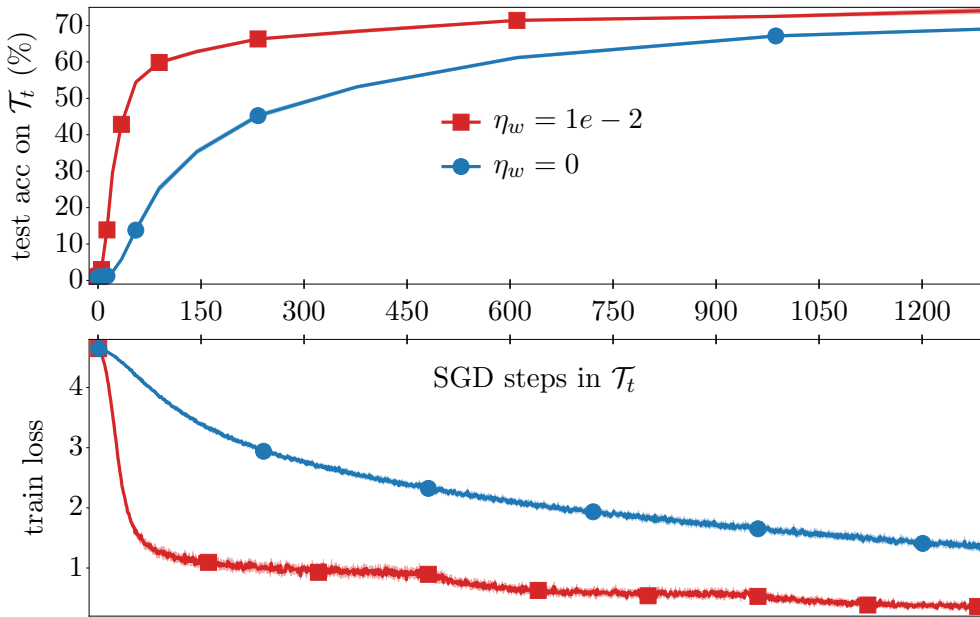


Figure 6.7: The effect of preventing the loss landscape to deform in $\mathfrak{S}(\phi)$ during feature-tuning.

6.4.3 Head Warmup

As shown by Li and Hoiem [2017] and [Kumar et al., 2021], feature-tuning can be done after a warmup phase during which only the head is modified. Li and Hoiem [2017] suggests that the transition between the two training phases (head warmup and feature-tuning) is committed whenever the validation performance does not improve for a number of training steps. In this experiment we show how using head warmup effects the generalizable performance while training.

In this experiment, the warmup phase is stopped whenever the validation accuracy has not improved at least one percent for 10 consecutive mini-batch SGD steps. The rest of the setup is akin to that of Section 6.4.2.

Large leaps of validation accuracy pointed out with arrows in the first row of Figure 6.8 show the step at which the transition from head warmup phase to jointly training ϕ and \mathbf{W} is committed. At these points, the back-propagate gradients have comparably large magnitude which makes the optimization algorithm take a large (but unnecessary) step in $\mathfrak{S}(\phi)$. The effective size of this step depends on $\|\mathbf{W}\|_F$ which in turn depends on \mathbf{W}^0 and η_w . In this experiment, we use $\eta_w = 10^{-2}$ across all cases but initialize the head in different ways. The orange and green curves correspond to the fan-out and fan-in modes of Kaiming initialization He et al. [2015], respectively. The blue curves shows the effect of letting the running statistics—used in batch normalization layers of feature extractor—to be updated during the training of the head. Similarly, the running statistics are updated for the case shown with red curves but the head is initialized with small-variance values in this case. The reason for including cases with updating running statistics is to opt-out its influence factor and make more certain conclusions.

Unlike most of experiments in this chapter, the visualization made for this experiment only reflects a single random seed (no shadow is plotted to indicate confidence interval) and the outcomes of runs with other random seeds are not shown so as to let the minimum overshooting leaps be visually clear. The optimization steps where sudden leaps in the performance of different curves takes place in Figure 6.8, supports our geometric hypothesis visualized in Figure 6.3a. In summary, the results presented in this experiment suggest that *although including a head warmup phase can help the optimization algorithm to take its initial steps in a correct direction, it still is prone to*

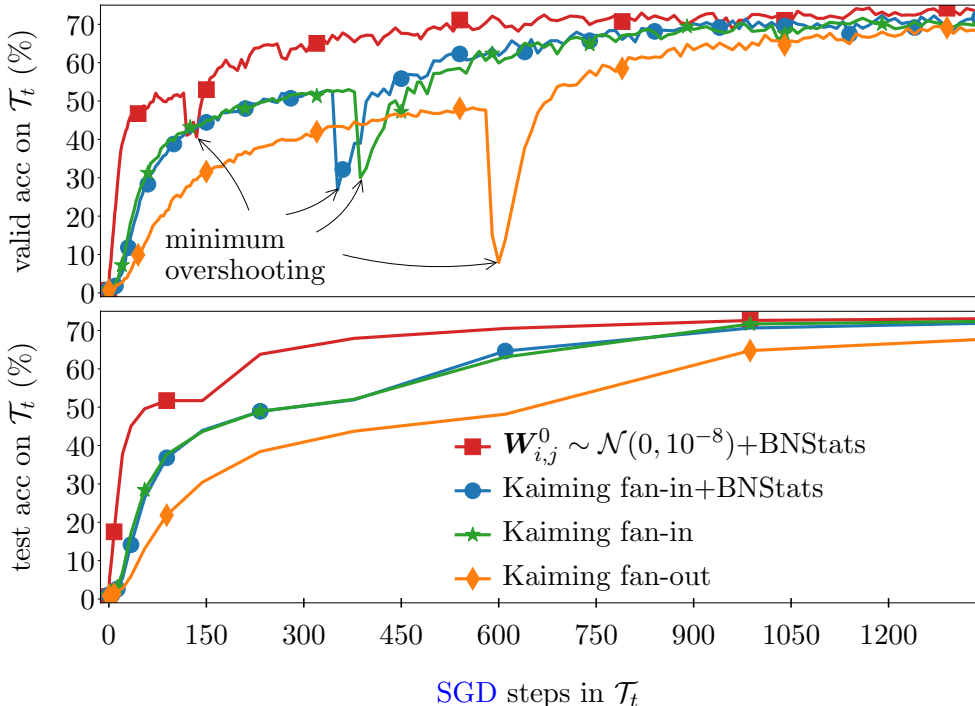


Figure 6.8: [Head](#) warmup is still prone to overshooting the minimum. The plots show the validation performance and progressive test performance from top to bottom.

minimum overshooting. In the severe case of minimum overshooting, ϕ may step into high-altitude regions of the loss landscape which makes the convergence non-efficient and can cause [catastrophic forgetting](#).

6.4.4 Decoupled Learning Rates

As suggested by Theorem 12, the effect of η_w on the relative velocity of ϕ and $\phi_{\mathcal{T}_t}^*$ is much larger than that of η_ϕ (because it affects the both ends). This implies that for an efficient convergence, the learning rates of the feature extractor and the [head](#) should be adjusted independently or at least should not necessarily be set equally as is traditionally done. In this experiment we compare the outcome of scaling these learning rates equally, as traditionally is set and decoupled from each other, according to our proposed method. To do so, starting from a baseline, we scale up and scale down η_w and η_ϕ both equally and unequally, and inspect the outcomes.

In this experiment, we use ResNet-18 [He et al., 2016] and VGG-19 [Simonyan and Zisserman, 2014] which are pretrained on ImageNet [Deng et al., 2009] to tune on the classification task defined over the CIFAR-100 dataset [Krizhevsky et al., 2009]. For all the cases, elements of \mathbf{W} are initialized randomly from a zero-centered normal distribution with a close to zero standard deviation (10^{-8}). The rest of the setup is akin to that of Section 6.4.2 except that the learning rates for mini-batch SGD are explicitly expressed in plots on each curve.

Figure 6.9 compares the top-1 progressive accuracy (top row) and the Frobenius norm of \mathbf{W} (bottom row) when the learning rates are scaled equally (Figure 6.9a) versus they are scaled in a decoupled fashion (Figure 6.9b) for the ResNet-18 [He et al., 2016]. The blue curves are identical in this Figure and are just repeated so can easily be compared to the other curves. Symmetrically scaling down the learning rates shown by the red curve on Figure 6.9a improves the test accuracy eventually, though it degrades the performance in the first 100 optimization steps. On the other hand, as shown by the red curve in Figure 6.9b, if only η_ϕ is scaled down, the best performance is similarly improved but is not compromised on the earlier steps at all.

For VGG-19 [Simonyan and Zisserman, 2014] which compared to ResNet-18 [He et al., 2016] is a more challenging model to train [Li et al., 2018], the difference between the aforementioned learning-rate scaling strategies seems to be bolder. This is depicted in Figure 6.10 where the asymmetric scaling more significantly speeds-up the progressive test performance.

The results in this experiment indicate that in order to have an efficient progressive performance, *the learning rates of the pretrained feature extractor and appended head are better to be tuned separately*. This statement does not intend to convey that these learning rates do not influence the optimal value of each other, but rather it emphasizes that the prevailing practice of setting them equal could compromise the performance either at the beginning of the training or for a long time afterward. Furthermore, the compromise seems to be more significant as the loss landscape is more chaotic (e.g., VGG-19 compared to ResNet-18 as explained by Li et al. [2018]).

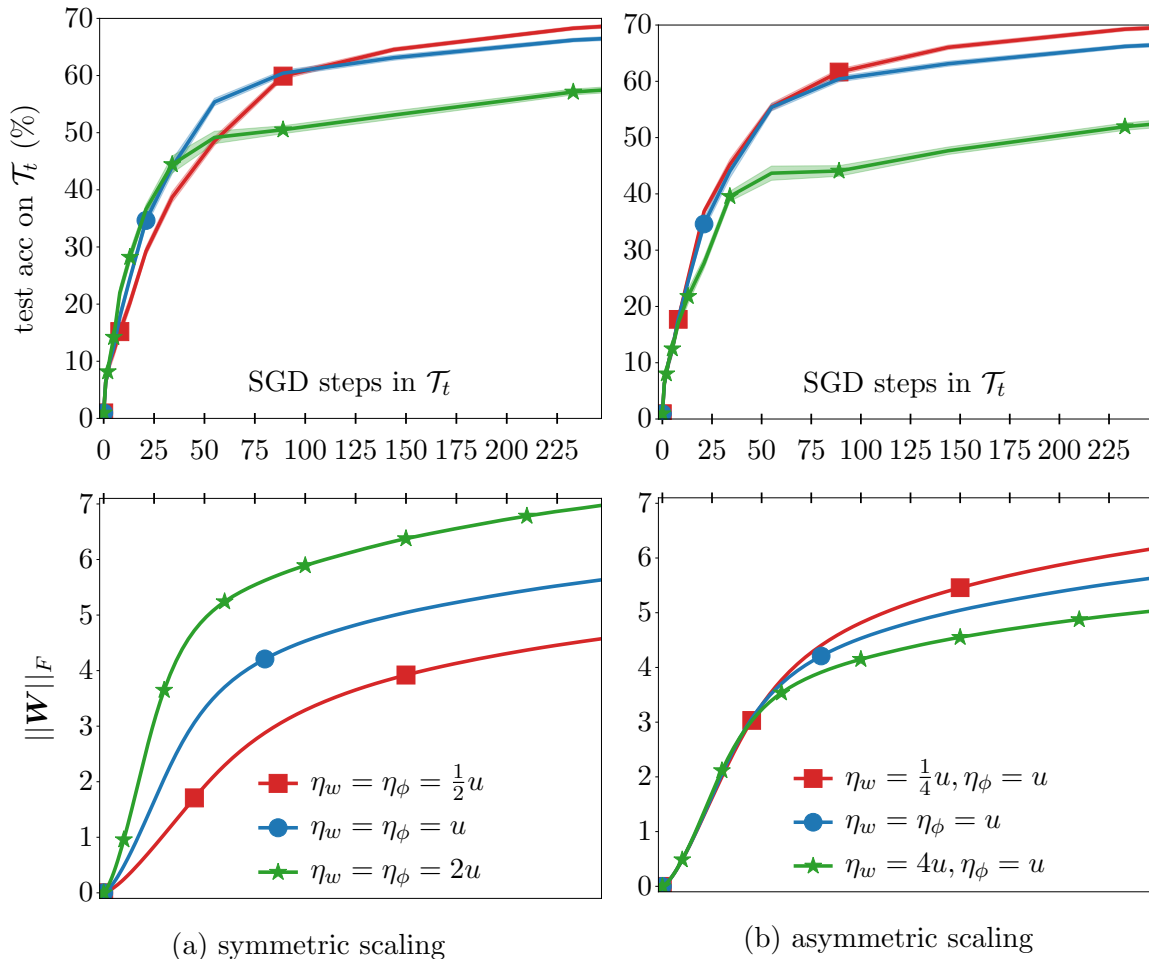


Figure 6.9: Comparing the effect of scaling the learning rate in (a) symmetrically versus (b) asymmetrically for feature-tuning on ResNet-18 [He et al., 2016]. In all cases $u = 10^{-2}$.

6.4.5 Optimization

Putting the outcomes of Sections 6.4.1 and 6.4.4 together, we desire to compare the complete FAST method with a feature-tuning baseline. However, we first need to answer two fundamental questions.

- First, are the outcomes of employing FAST essentially different from those of the baseline or could similar convergence behavior be obtained by tuning the learning in the baseline?
- Second, does FAST reduce the performance gap between mini-batch SGD with momentum and Adam (In the following we explain why this is expected)?

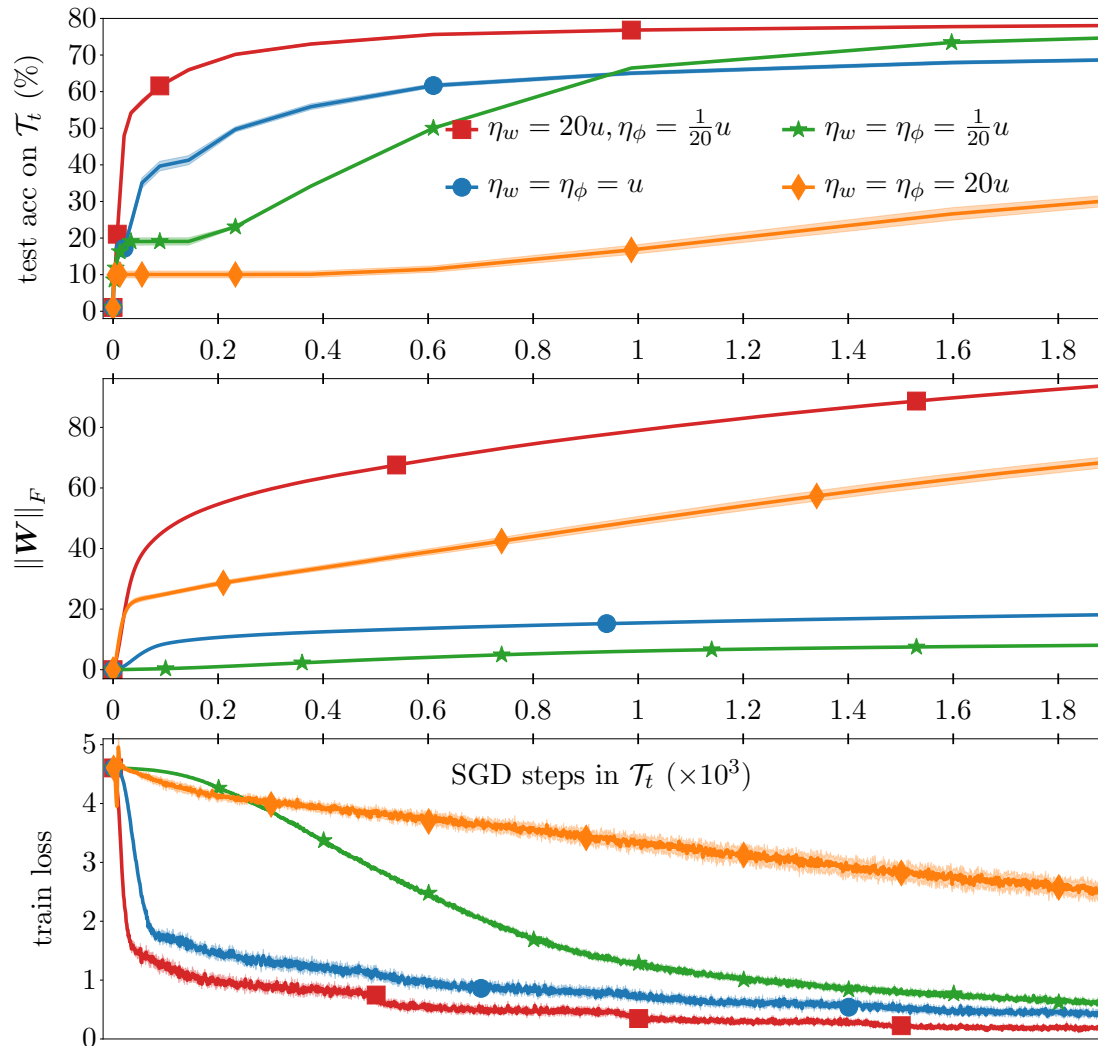


Figure 6.10: Comparing the effect of scaling the learning rate in (a) symmetrically versus (b) asymmetrically for [feature-tuning](#) on [VGG-19](#) [Simonyan and Zisserman \[2014\]](#). In all cases $u = 0.05$ and $\mathbf{W}_{i,j}^0 \sim \mathcal{N}(0, 10^{-8})$.

We look for answers to these question from an empirical point of view.

In this experiment, we use [VGG-19](#) [\[Simonyan and Zisserman, 2014\]](#) as our challenging optimization case, in addition to [DenseNet-201](#) [\[Huang et al., 2017\]](#) which according to what [Li et al. \[2018\]](#) states about the skip connections, is characterized with a relatively smooth loss landscape despite having a lot of layers. Both models are pretrained on [ImageNet](#) [\[Deng et al., 2009\]](#) and the goal is to tune them on the classification task defined over the [CIFAR-100](#) dataset [\[Krizhevsky et al., 2009\]](#).

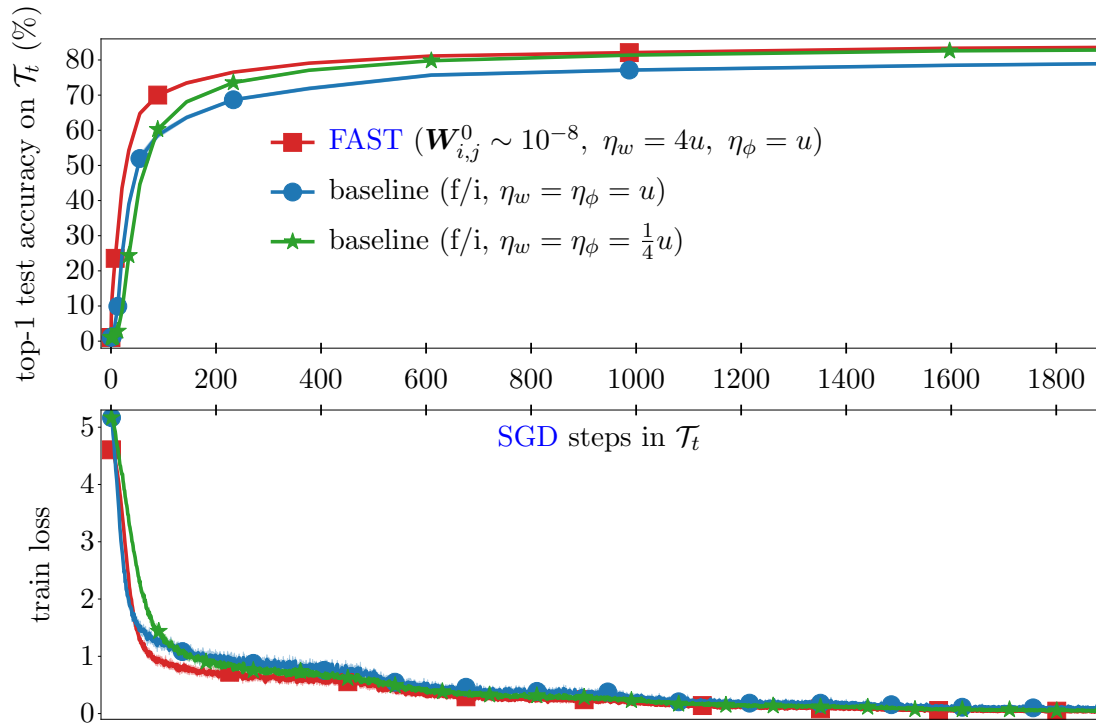
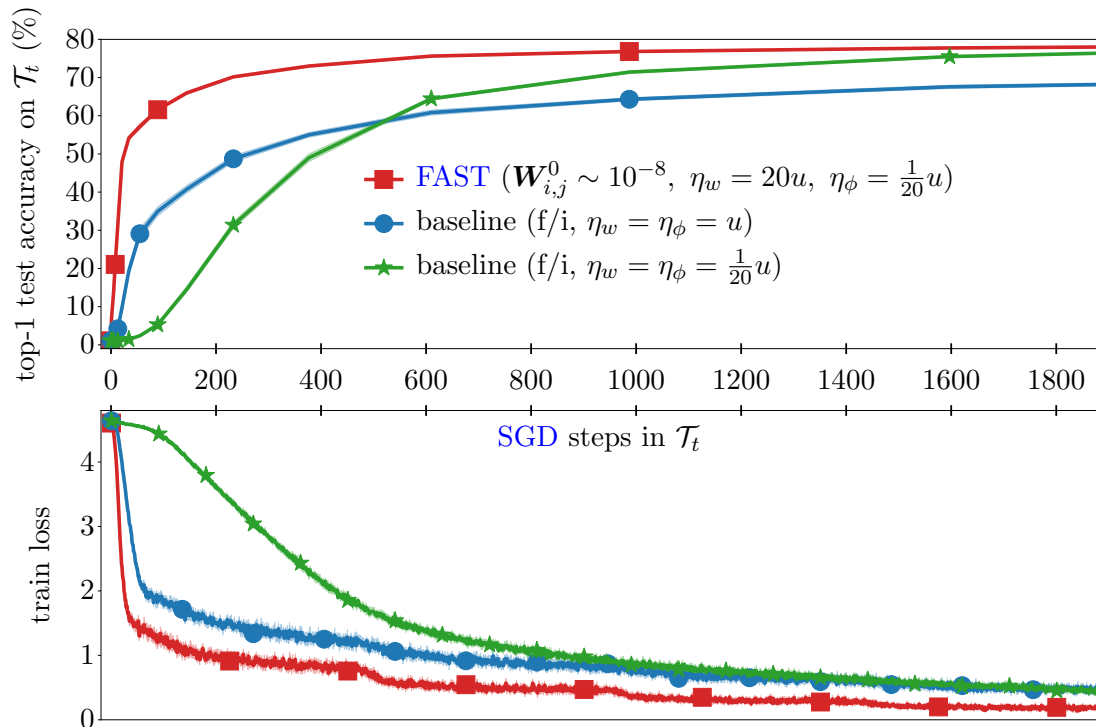
(a) DenseNet-201, $u = 1 \times 10^{-2}$ (b) VGG-19, $u = 5 \times 10^{-2}$

Figure 6.11: Comparison between FAST and the baselines in their learning progress when mini-batch SGD with momentum is used as the optimization algorithm.

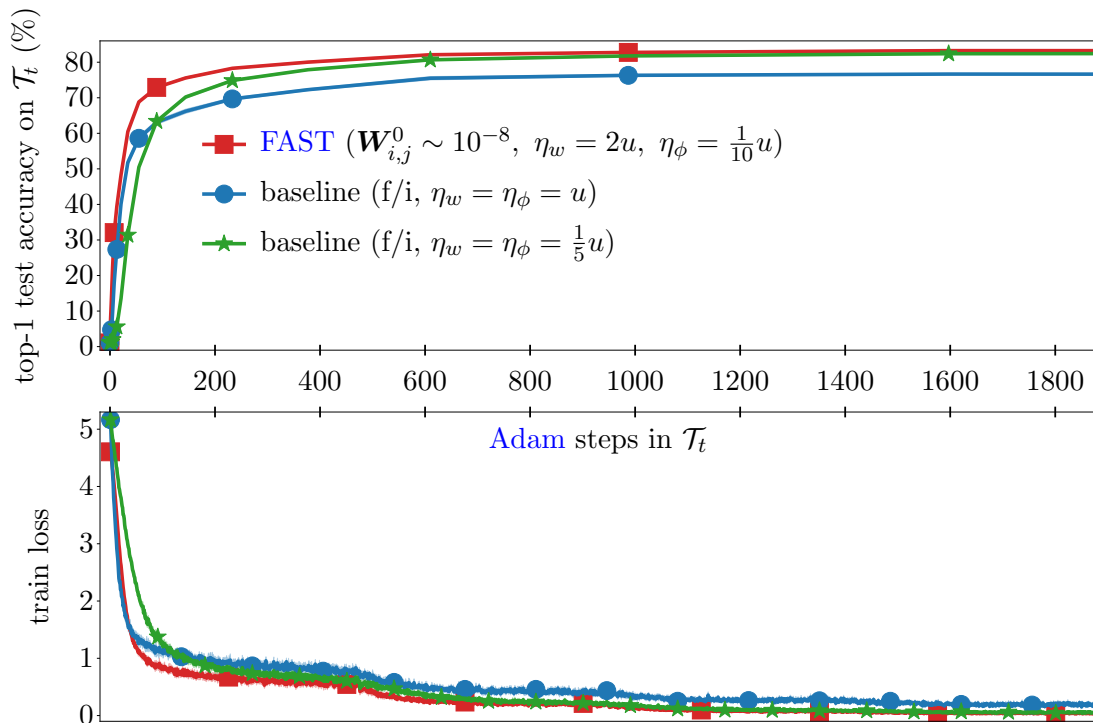
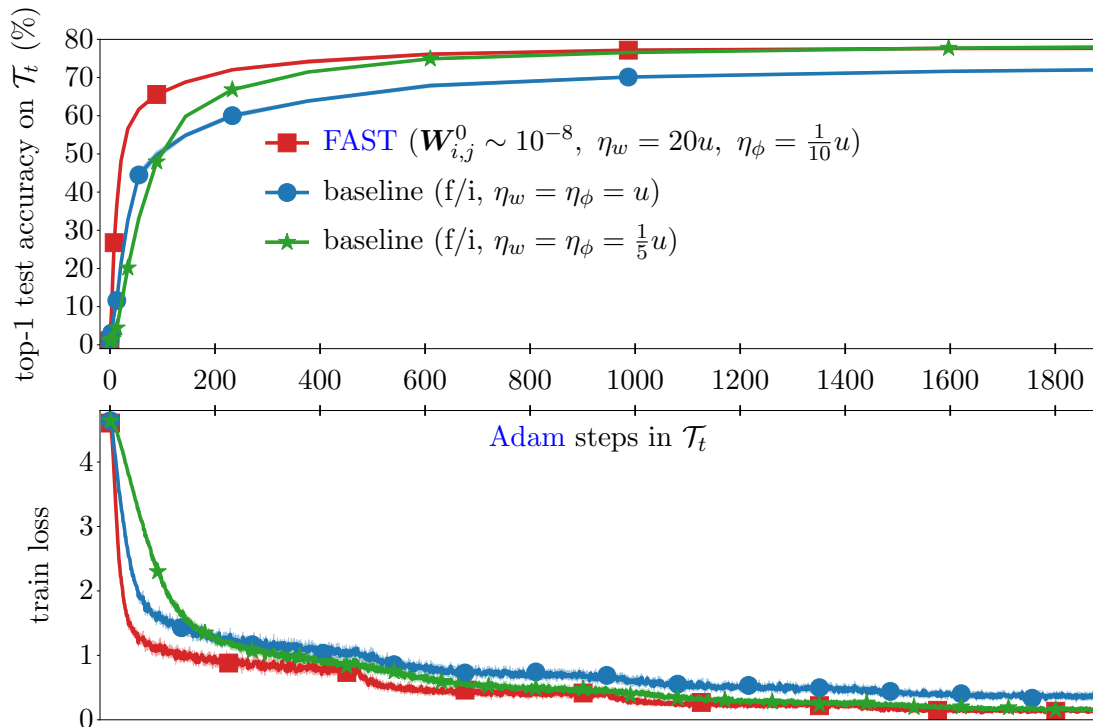
(a) DenseNet-201, $u = 5 \times 10^{-4}$ (b) VGG-19, $u = 5 \times 10^{-4}$

Figure 6.12: Comparison between FAST and the baselines in their learning progress Adam is used as the optimization algorithm.

The learning rates and initialization setup for [FAST](#) and the baselines are explicitly marked on this section’s the plots. Wherever the [Adam](#) optimization algorithm [[Kingma and Ba, 2015](#)] is used, its exponential decay rates for estimating the first and the second moments are set to 0.9 and 0.999, respectively (the defaults recommended by the original paper). The results are also verified for the [RAdam](#) algorithm which is introduced more recently by [Liu et al. \[2019\]](#). The definition of [RAdam](#) is directly borrowed from the implementation that [Liu et al. \[2019\]](#) provided and the default hyper-parameters are preserved.

Similar to Section 6.4.4, when the unified learning rate is scaled down or scaled up for the baselines, in this experiment, the progressive performance is either compromised at the beginning of [feature-tuning](#) or for a large number of steps afterward. In other words, optimizing the baselines with larger learning rates improves the performance at the beginning of [feature-tuning](#) but holds it back from reaching its potential performance thereafter. However, as it is shown in Figures 6.11b and 6.11a, employing [FAST](#) provides for achieving a non-compromised progressive performance. Figures 6.12b and 6.12a, have the same message as that of Figures 6.11b and 6.11a except that all the models corresponding to the curves in these Figures use the [Adam](#) optimization algorithm. The baseline curves are marked with f/i which refers to the fan-in mode of the Kaiming [He et al. \[2015\]](#) method used for initializing the appended parameters³.

A comparison between Figures 6.11b and 6.12b, clearly reveals that [Adam](#) substantially speeds-up the convergence of the baselines compared to [mini-batch SGD](#) with [momentum](#). However, applying [Adam](#) on top of [FAST](#) does not make a huge difference. To make it easier to compare the cases that use [FAST](#) in Figures 6.11b and 6.12b, we plotted the same corresponding curves again in Figure 6.14b. We also compared the aforementioned optimization algorithms that use [FAST](#) with a case in which *Nesterov momentum* [Nesterov \[1983\]](#) is used in [mini-batch SGD](#), instead of the simple [momentum](#). Also notice that as expected, [Adam](#) fails to get on par with [mini-batch SGD](#) with [momentum](#) after a long period of training but generally the performance gap comparably looks minor during the whole training period.

³In these cases, since $\|\mathbf{a}\| > C$, choosing the fan-out mode of the Kaiming method to initialize \mathbf{W} results in worse performance than that of the fan-in mode

Finally, Figure 6.13a compares FAST with the baselines when the RAdam optimization algorithm [Liu et al., 2019] is employed. Unlike in the original setup used in Liu et al. [2019], in our feature-tuning setting the convergence rate of RAdam seems not to be less sensitive to the choice of the learning rate than Adam. Moreover, comparing Figure 6.12 with Figure 6.13 suggests that Adam converges faster than RAdam for the settings used in this experiment.

In summary, tuning the learning rate results in a compromise of the baselines’ performance either at the beginning of the training or after a large number of optimization steps. FAST solves this issue by decoupling the learning rates for the feature extractor and the head. Moreover, FAST could be used along with advance optimization algorithms to improve the progressive performance. However, *when FAST is used for feature-tuning a model pretrained on a source task on a similar target task, the gap between performance obtained from different optimization algorithms is notably decreased.* This supports our hypothesis that FAST prevents the feature-extractor’s parameters from suddenly stepping out of the proximity of $\phi_{\mathcal{T}_s}^*$ and $\phi_{\mathcal{T}_t}^*$ —Supposing $\phi_{\mathcal{T}_s}^*$ and $\phi_{\mathcal{T}_t}^*$ lay close to each other in a smooth region on $\mathfrak{S}(\phi, \ell)$ due to similarity of \mathcal{T}_s and \mathcal{T}_t .

6.4.6 Convergence Performance

In this experiment we compare FAST and the baselines with a focus on convergence accuracy. The main goal is to confirm that using FAST would not have an adverse effect on convergence performance; therefore, the models in this experiment are fine-tuned for a longer period than those in the previous experiments. Another goal of this experiment is to quantitatively confirm that feature-tuning with FAST keeps ϕ closer to $\phi_{\mathcal{T}_s}^*$ compared to when the baselines are employed. This is similar to the experiment conducted in Section 6.4.1 in that they both indicate how much our method and the baselines are resistant to forgetting. However, the one in Section 6.4.1 focuses on generalizable accuracy while in this experiment the geometric perspective is highlighted.

For this experiment, we employ ResNeXt-101-32x8d introduced by Xie et al. [2017], which is already pretrained on ImageNet [Deng et al., 2009]. The feature-tuning is done on CIFAR-100 [Krizhevsky et al., 2009], Caltech-101 [Li et al., 2022]

Table 6.1: Comparing **FAST** with baselines in their convergence performance and traveled distance in $\mathfrak{S}(\phi)$ from $\phi_{\mathcal{T}_s}^*$. In all baselines, the **head** is initialized using the fan-in mode of Kaiming method [He et al., 2015] and $\eta_w = \eta_\phi$.

dataset	method	top-1 error	top-5 error	$\ \phi_{\mathcal{T}_t}^* - \phi_{\mathcal{T}_s}^*\ $
CIFAR-100	baseline ($\eta_\phi = 5 * 10^{-2}$)	22.64 ± 0.26	5.91 ± 0.14	119.51 ± 1.38
	baseline ($\eta_\phi = 10^{-2}$)	16.80 ± 0.12	3.46 ± 0.06	34.58 ± 0.43
	baseline ($\eta_\phi = 5 * 10^{-3}$)	13.90 ± 0.13	2.22 ± 0.08	14.32 ± 0.12
	FAST	12.63 ± 0.09	1.62 ± 0.04	7.19 ± 0.04
Caltech-256	baseline ($\eta_\phi = 5 * 10^{-2}$)	18.45 ± 0.14	7.14 ± 0.01	37.2 ± 0.23
	baseline ($\eta_\phi = 10^{-2}$)	13.55 ± 0.19	4.38 ± 0.09	7.70 ± 0.07
	baseline ($\eta_\phi = 5 * 10^{-3}$)	14.39 ± 0.14	4.73 ± 0.10	5.93 ± 0.17
	FAST	12.81 ± 0.10	3.96 ± 0.08	5.28 ± 0.06

and **Caltech-256** Griffin et al. [2007] classification tasks. **Caltech-256** contains 74×74 labeled natural **RGB** images from 256 classes. It is originally published with no train-test split; therefore, we made a random but stratified split with 20% of the images as test. In the training split, 5 images are randomly selected from each class to form the validation set. This quantity is aligned with the size of validation set we chose for the other datasets (5% of the whole training) employed in this study. The images in **Caltech-256** dataset are resized to twice their original size in both width and height. **Feature-tuning** on **CIFAR-100** and **Caltech-256** continued for 250 and 330 epochs respectively. In all the cases, the batch size is set equal to the number of classes, and the training batches are sampled in a stratified order.

Figures 6.15 and 6.16 compare the progressive test accuracy obtained from **FAST** and the baselines. The wide plots on the left of each Figure show the progressive test accuracy of the first **mini-batch SGD** steps while the narrow one on the right side magnify the same quantity but for a larger number of steps taken afterward. Neither the magnitudes nor the ratios of the axes are preserved among the plots in each Figure. The confidence interval is shown with transparent shadows but are barely visible since there is a high consistency among the results from different random seeds. The convergence results are also restated in numbers in Table 6.1. *The performance on test dataset suggest not only that **FAST** learns faster than the baselines but it also helps reaching a higher convergence performance.* The latter is not a very strong argument in the sense that the training time is subjective and the point of convergence has

no formal meaning in this context—i.e., one may claim that after an infinite number of training steps the difference between progressive results may become insignificant (see He et al. [2019a]).

Assume that after the chosen number of training epochs, the convergence is achieved (in its broad definition: no improvement for a fairly large number of epochs). In the last column of Table 6.1 we present the Euclidean distance traveled by the feature extractor’s parameters from aiming minimum of the source task’s loss to that of the target task’s loss. The results confirm that compared to the baselines, *FAST* preserves $\phi_{T_t}^*$ in a closer proximity to $\phi_{T_s}^*$. This implies that *FAST reaches the objective of the target task from a direction that is more aligned with the objective of the source task compared to the baselines.* This characteristic makes *FAST* more resistant to catastrophic forgetting.

6.4.7 FAST Compared to ENTAME

In this section we compare *FAST* and *ENTAME* in terms of the empirical speed of convergence. For this experiment we tune a pretrained *ResNet-18* on *CIFAR-100*. The fan-in mode of Kaiming’s method He et al. [2015] is chosen as the baseline. The learning rates are tuned for the largest area under test accuracy curve for the first 2000 training steps using a validation set of 25 examples per class. The learning rate values are selected from $0.01 * \{\frac{1}{4}, \frac{1}{2}, 1, 2\}$. The best learning rate for the baseline and *ENTAME* was 0.005. For *FAST*, the best learning rates were $\eta_w = 0.01$ and $\eta_\phi = 0.1 * \frac{1}{4}$.

Figure 6.17 shows the outcome of this experiment. The mean and 95% confidence are shown with solid line and color shades, respectively. Unlike other experiments in this chapter, each run in this experiment is repeated for 5 different seeds. As seen in this Figure, *ENTAME* and *FAST* show roughly the same convergence rate and final performance while the baseline falls behind both. Note that compared to the experiments presented in Chapter 5, we report higher performance for both the baselines and *ENTAME* in this Chapter. The major reason for this performance gap is efficiently scaling up the size of the input images (e.g., 128×128 for *CIFAR-100*). To learn more about the impact of scaling the input images, see Touvron et al. [2019].

6.5 Conclusion

In this study we pinpointed a major problem of traditional [feature-tuning](#) that not only leads to [catastrophic forgetting](#), but also slows down the convergence to the target task. To clarify this problem, we provided a novel perspective of the loss landscape which makes it possible to connect the loss landscapes of the source and target tasks. Using this perspective, we compared different [task adaptation](#) techniques and proposed [FAST](#) for [feature-tuning](#) pretrained neural networks on classification tasks. With a focus on machine vision and [Convolutional Neural Network \(CNN\)](#) models, we provided an empirical analysis with the following outcomes:

- Initializing the [head](#) for the target task in [task adaptation](#) with close-to-zero values, not only better preserves the transferred knowledge but also accelerates the training procedure on the target task (Section [6.4.1](#)).
- The convergence speed is largely influenced by the randomness of the parameters of the [head](#) and how quickly they are adapted (Section [6.4.2](#)).
- Although including a [head](#) warmup phase (as suggested by [Li and Hoiem \[2017\]](#) and [Kumar et al. \[2021\]](#)) can help the optimization algorithm to take its initial steps in a correct direction, it still is prone to minimum overshooting (Section [6.4.3](#)).
- For an efficient convergence, the learning rates chosen for the pretrained feature-extractor and the appended [head](#) are better to be tuned separately. Otherwise, if they are chosen equally as in the traditional [feature-tuning](#) the progressive test performance can be compromised either in the beginning of the [feature-tuning](#) or during a large number of optimization steps afterward (Section [6.4.4](#)).
- Compared to the traditional [feature-tuning](#), using [FAST](#) remarkably decreases the gap between the performance obtained from [mini-batch SGD](#) with [momentum](#) and [Adam](#) optimization algorithms.

Generally, our proposed method learns the target task faster than traditional [feature-tuning](#), besides preserving more knowledge about the source task. We showed the superiority of our method from different analytical and empirical viewpoints, though yet more aspects remain to be explored.

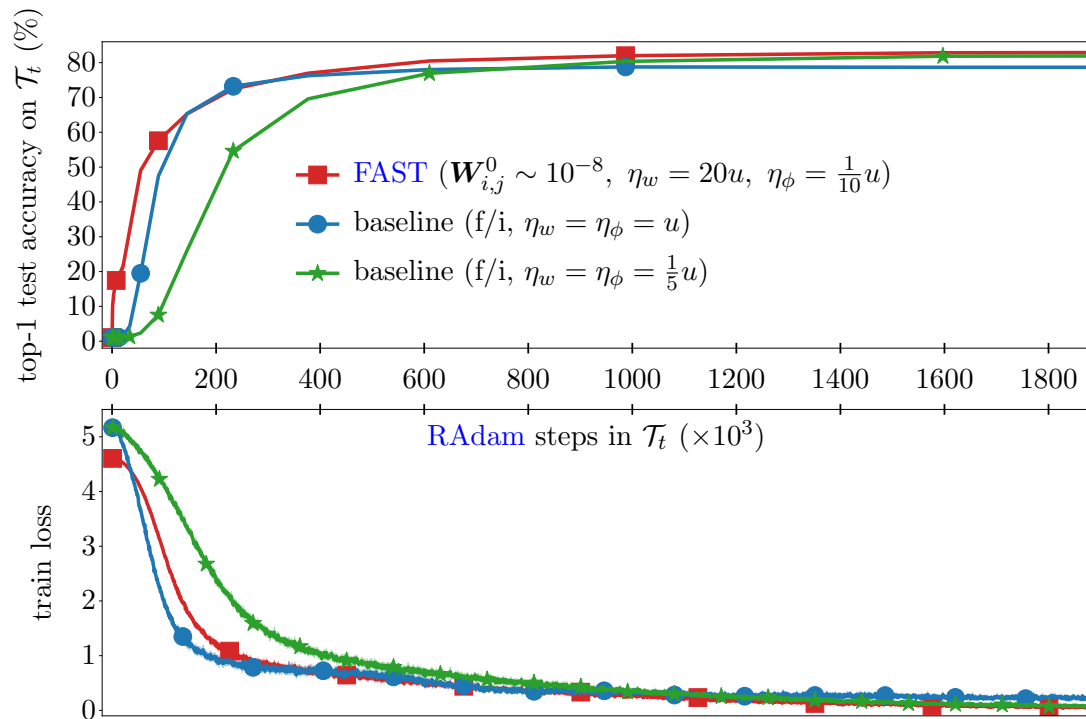
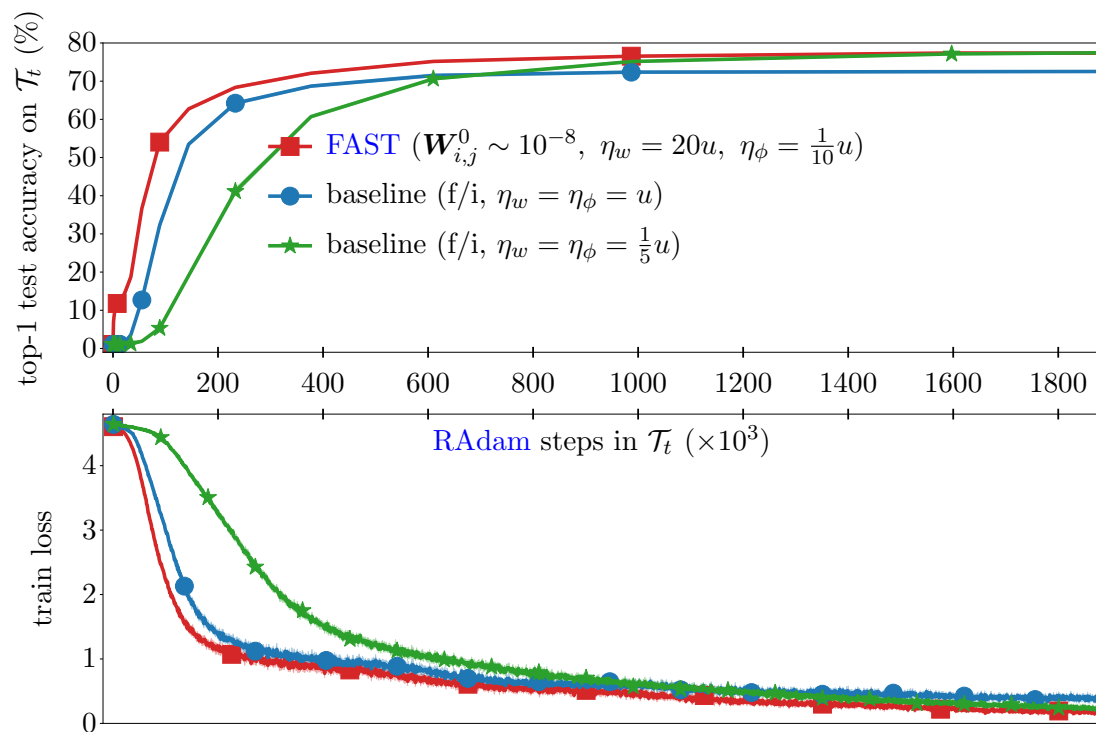
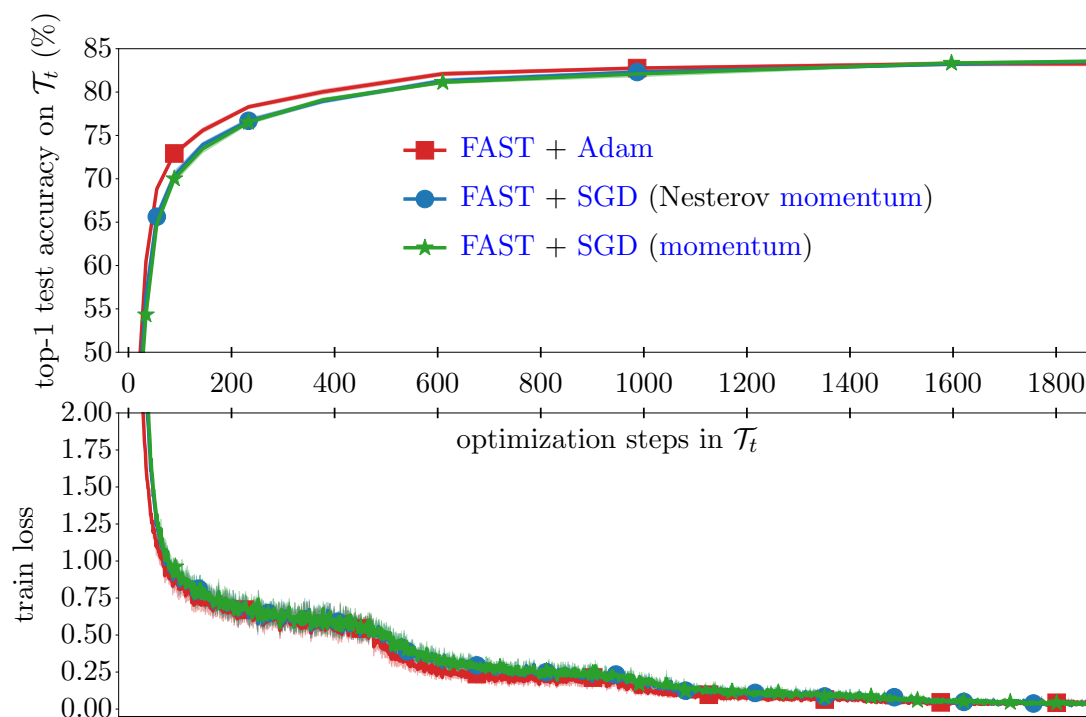
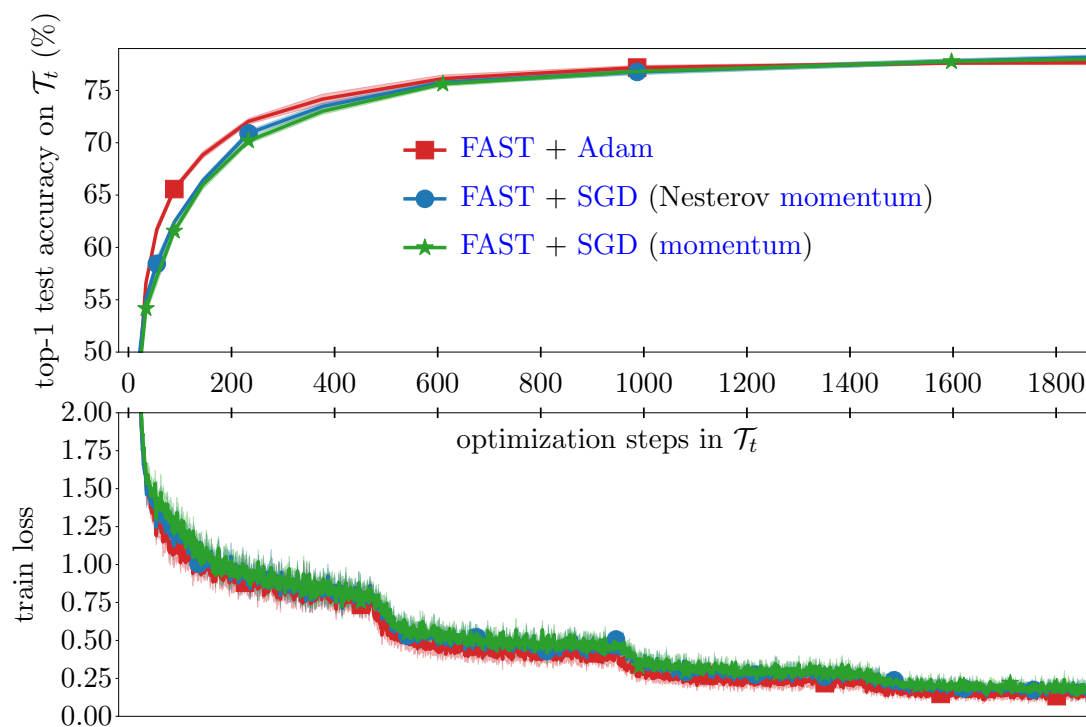
(a) DenseNet-201, $u = 5 \times 10^{-4}$ (b) VGG-19, $u = 5 \times 10^{-4}$

Figure 6.13: Comparison between FAST and the baselines in their learning progress when RAdam is used as the optimization algorithm.



(a) DenseNet-201



(b) VGG-19

Figure 6.14: Comparison between the learning progresses corresponding to different optimization algorithms used with FAST.

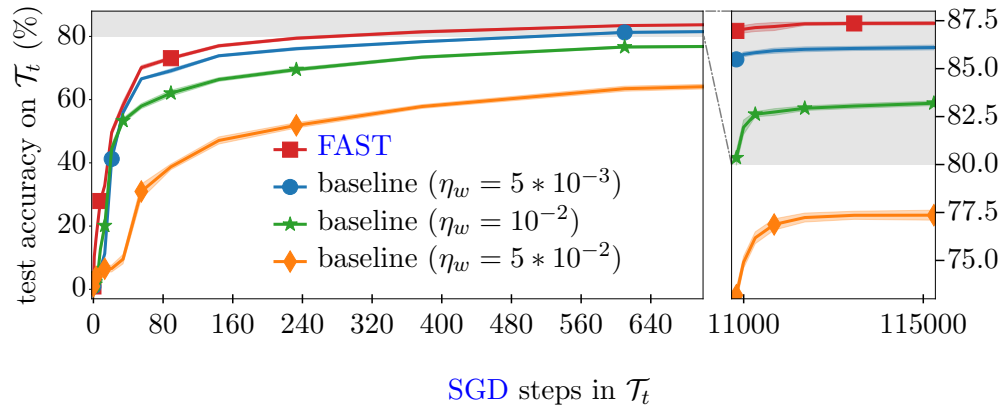


Figure 6.15: Comparing the convergence progressive test accuracy of **FAST** and the baselines on **CIFAR-100** dataset.

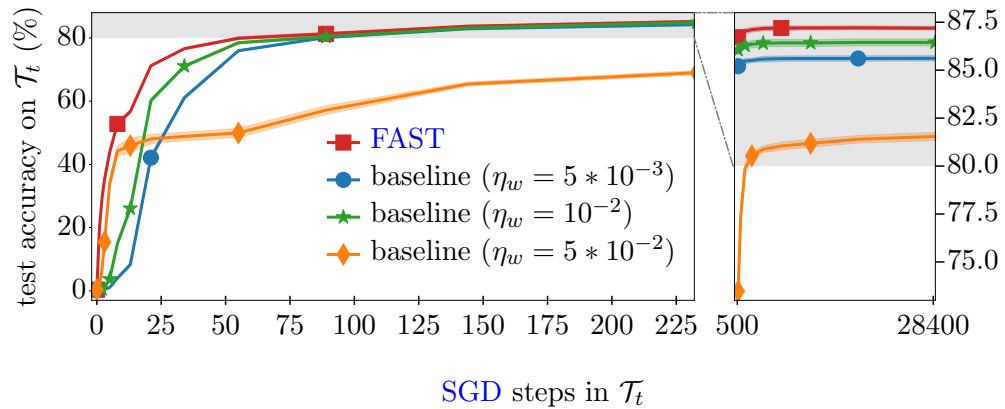


Figure 6.16: Comparing the convergence progressive test accuracy of **FAST** and the baselines on **Caltech-256** dataset.

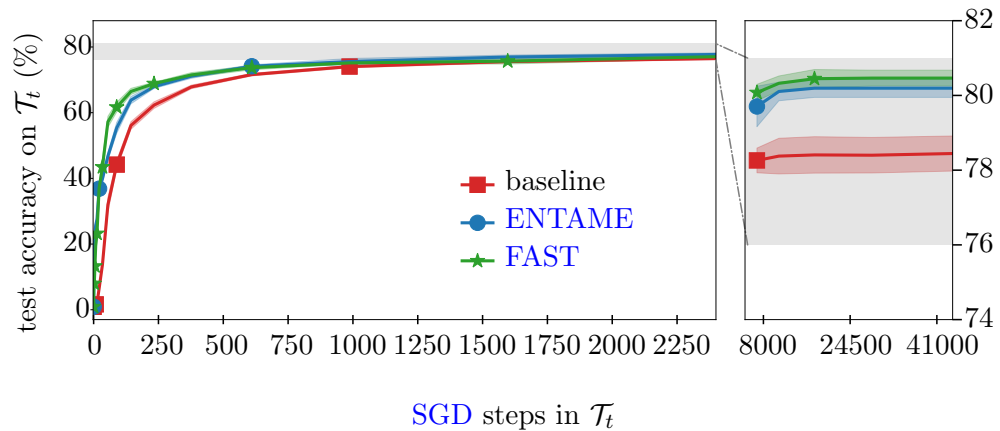


Figure 6.17: Comparing baseline, **ENTAME**, and **FAST** in their speed of convergence and convergence accuracy on **CIFAR-100** dataset.

Chapter 7

Adaptive Bias Estimation for Federated Learning

7.1 Introduction

About half a century ago, a few giant computers constituted the first general-purpose computer network [Kaminow and Li, 2002]. Soon after, military incentives and economic opportunities led to the expansion of computer networks, and as a result, “the Internet” was established. As technology advanced, the number of devices connected to the Internet continued to grow at an unprecedented rate. With this growth came an explosion of remote applications, from simple websites to complex data-driven systems. These applications were typically built using the [client-server model](#), where clients requested services from servers, which in turn processed and replied to these requests.

With the increasing demand for data-driven applications, the amount of data being generated and transmitted over the Internet grew exponentially. This put a tremendous strain on the network [Hecht et al., 2016], as well as on the servers that were responsible for processing and storing all of this data. [Edge computing](#) has emerged to address this issue. It involves processing data as close to where it is generated as possible, in order to reduce the amount of data that needs to be transmitted over the network and the amount of processing required on the servers.

[ML](#) has been the most popular and impactful data-driven approach in recent years. [Figure 7.1](#) shows the significance of [ML](#)’s popularity, comparing the interest trends in “machine learning” and “statistical analysis” terms queried from the Google search engine between January 2004 to January 2023. For many industries and applications, however, [ML](#) has not even come close to reaching its true potential. A major reason is that [ML](#) models are primarily designed to digest centralized datasets which is by and large in conflict with the new [edge computing](#) paradigm and also with the increasing demand for data privacy. [Federated Learning \(FL\)](#) is the most promising paradigm that has emerged to make [ML](#) compatible with these new desires (see [Figure 7.2](#) for

a measure of the popularity).

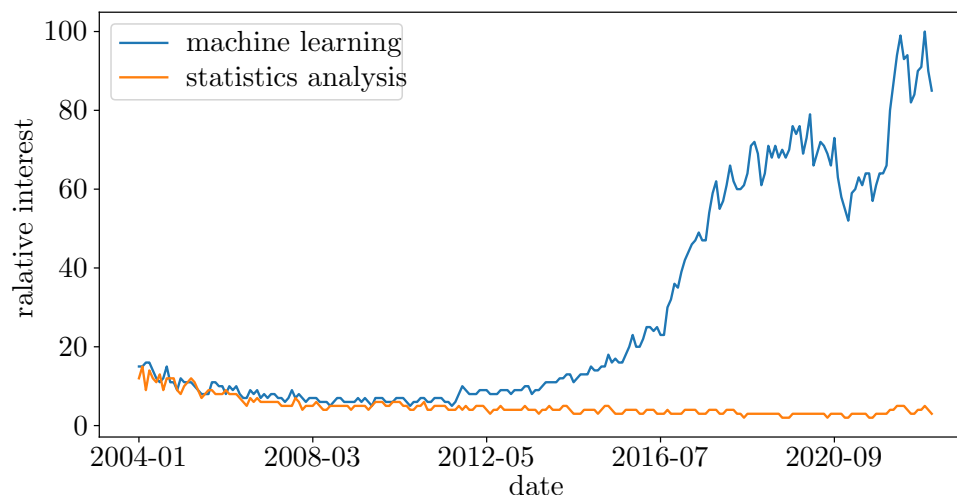


Figure 7.1: Relative interest trend for terms “machine learning” and “statistical analysis” in Google searches from January 2004 to January 2023.

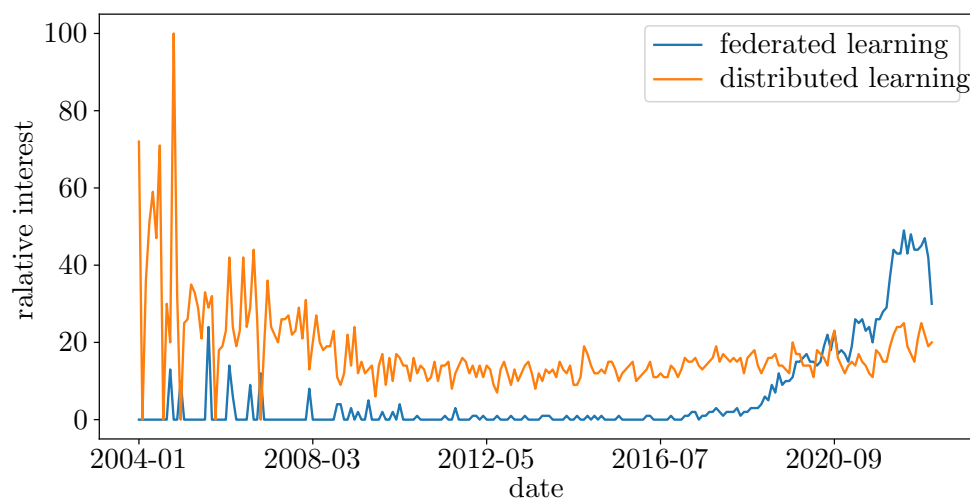


Figure 7.2: Relative interest trend for terms “federated learning” and “distributed learning” in Google searches from January 2004 to January 2023.

FL is a distributed **ML** technique that allows for the training of models on multiple devices or edge nodes without compromising the privacy of the data. With **FL**, devices such as smartphones and IoT devices can contribute their data to the training process without the need to transmit sensitive data to a central location. By training models locally on each device and only transmitting model updates to a central server, **FL**

not only improves data privacy but also provides a more robust and accurate model, as it is trained on a diverse set of data.

7.2 Problem Statement

In **FL**, multiple sites with data often known as *clients* collaborate to train a model by communicating parameters through a central hub called *server*. At each round, the server broadcasts a set of model parameters to a number of clients. Selected clients separately optimize towards their local objective. The locally trained parameters are sent back to the server, where they are aggregated to form a new set of parameters for the next round. A well-known aggregation is to simply average the parameters received from the participating clients in each round. This method is known as **Federated Averaging (FedAvg)** [McMahan et al., 2017] or **LocalSGD** [Stich, 2018].

In order to reduce the communication costs as well as privacy concerns (for example possible leakage of data from gradients [Zhu and Han, 2020]), multiple local optimization steps are often preferable and sometimes inevitable [McMahan et al., 2017]. Unfortunately, multiple local updates subject the parameters to *client drift* [Karimireddy et al., 2020]. While **SGD** is an unbiased gradient descent estimator, **LocalSGD** is biased due to the existence of client drift. As a result, **LocalSGD** converges to a neighborhood around the optimal solution with a distance proportionate to the magnitude of the bias [Ajalloeian and Stich, 2020]. The amount of this bias itself depends on the heterogeneity among the clients' data distribution, causing **LocalSGD** to perform poorly on non-**Independent and Identically Distributed (IID)** benchmarks [Zhao et al., 2018].

7.3 Existing Solutions

One effective way of reducing client drift is by adapting **Reduced Variance Stochastic Gradient Descent (RV-SGD)** methods (introduced in Chapter 2) to **LocalSGD**. The general strategy is to regularize the local updates with an estimate of gradients of inaccessible training samples (i.e., data of other clients). In other words, the optimization direction of a client is modified using the estimated optimization direction of other clients. These complementary gradients could be found for each client i by

subtracting an estimate of the local gradients from an estimate of the oracle’s¹ full gradients. In this chapter, we refer to these two estimates with \mathbf{h}_i and \mathbf{h} , respectively. Therefore, a reduced variance local gradient for mini-batches on client i would be in general form of $\mathbb{E}_{\text{batch}} [\nabla_{\boldsymbol{\theta}} \ell_i] + (\mathbf{h} - \mathbf{h}_i)$ where $\nabla_{\boldsymbol{\theta}} \ell_i$ corresponds to the true gradient of the local objective for client i .

The majority of existing research work on adapting **RV-SGD** to distributed learning, do not meet the requirement to be applied to **FL**. Some proposed algorithms require full participation of clients [Shamir et al., 2014, Reddi et al., 2016, Liang et al., 2019]; thus, they are not scalable to *cross-device FL*². Another group of algorithms require communicating the true gradients [Li et al., 2019, Murata and Suzuki, 2021] and, as a result, completely undermine the **FL** privacy concerns such as attacks to retrieve data from true gradients [Zhu and Han, 2020].

Stochastic Controlled Averaging for Federated Learning (SCAFFOLD) [Karimireddy et al., 2020] is an algorithm that supports partial participation of clients and does not require the true gradients at the server. While **SCAFFOLD** shows superiority in performance and convergence rate compared to its baselines, it consumes twice as much bandwidth. To construct the complementary gradients, it computes and communicates \mathbf{h} as an extra set of parameters to each client along with the model parameters. **Federated Learning with Dynamic Regularization (FedDyn)** [Acar et al., 2020] proposed to apply \mathbf{h} in a single step prior to applying any local update, and practically found better performance and convergence speed compared to **SCAFFOLD**. Since applying \mathbf{h} uses the same operation in all participating clients, Acar et al. [2020] moved it to the server instead of applying on each client. This led to large savings of local computation, and more importantly to use the same amount of communication bandwidth as vanilla **LocalSGD** (i.e., **FedAvg**), which is half of what **SCAFFOLD** uses.

FedDyn makes several assumptions that are often not satisfied in large-scale **FL**. These assumptions include having prior knowledge about the total number of clients, a high rate of re-sampling clients, and drawing clients uniformly from a stationary set.

¹Oracle dataset refers to the hypothetical dataset formed by stacking all clients’ data. Oracle gradients are the full-batch gradients of the Oracle dataset.

²In contrast to *cross-silo FL*, *cross-device FL* is referred to a large-scale (in terms of number of clients) setting in which clients are devices such as smart-phones.

Even with these assumptions, we show that \mathbf{h} in FedDyn is pruned to explosion, especially in large-scale setting. This hurts the performance and holds the optimization back from converging to a stationary point.

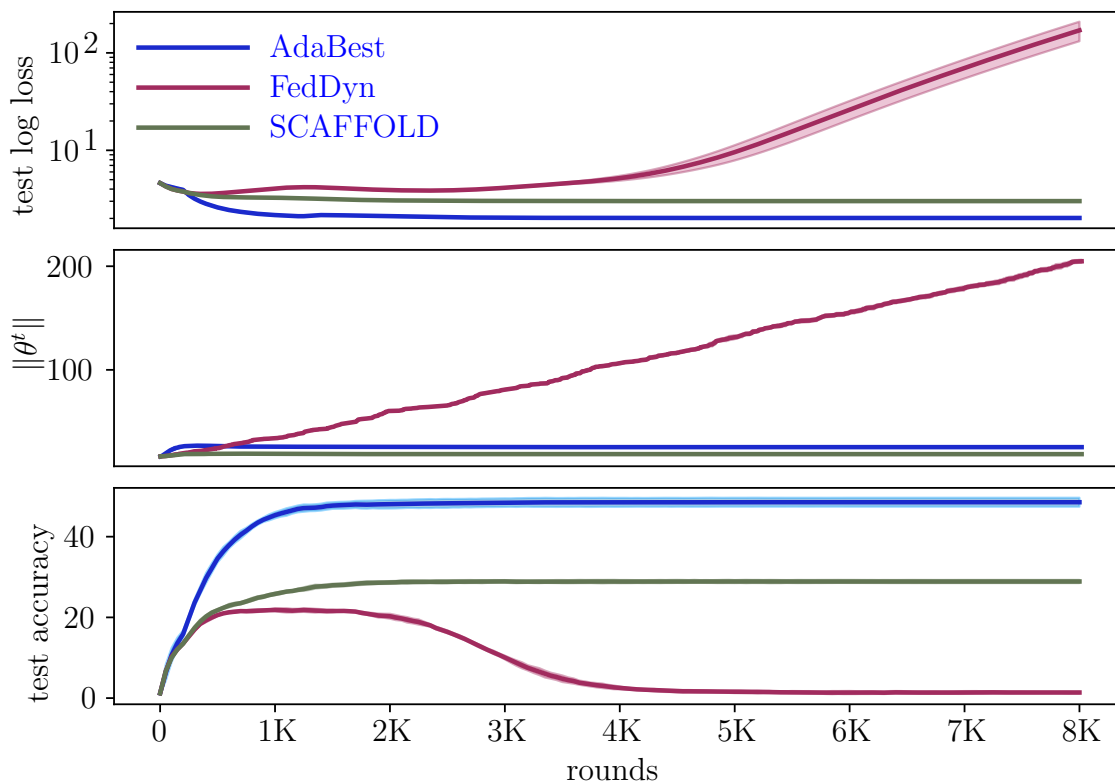


Figure 7.3: Asymptotic instability of FedDyn as a results of unbounded increase of $\|\mathbf{h}^t\|$. From top to bottom test loss (log scale), norm of cloud parameters, and test accuracy are shown in subplots. The shared horizontal axis shows the number of communication rounds. Each experiment is repeated 5 times with different random seed of data partitioning. Solid lines and shades represent mean and standard deviation respectively

This Chapter proposes [AdaBest](#), a [Reduced Variance LocalSGD \(RV-LSGD\)](#) solution to minimize the client drift in FL. Compared to the baselines, we define a simpler yet more elegant method of incorporating previous observations into estimates of complementary gradients. Our solution alleviates the instability of the norm of \mathbf{h} in FedDyn (see Figure 7.3 for empirical evaluation) while consuming the same order of storage and communication bandwidth, and even reducing the compute cost (see Appendix D for quantitative comparison). Unlike previous works, our algorithm provides a mechanism for adapting to changes in the distribution of client sampling and

does not require prior knowledge of the entire set of clients.

7.4 Adaptive Bias Estimation

7.4.1 Setup

We assume a standard FL setup in which a central server communicates parameters to a number of clients. The goal is to find an optimal point in the parameter space that solves a particular task while clients keep their data privately on their devices during the whole learning process. Let S^t be the set of all registered clients at round t and \mathcal{P}^t be a subset of it drawn from a distribution $P(S^\tau; \tau = t)$. The server broadcasts the *cloud model* θ^{t-1} to all the selected clients. Each client $i \in \mathcal{P}^t$, uses *mini-batch SGD* to optimize the cloud model based on its local objective and transmits the optimized *client model*, θ_i^t back to the server. The server aggregates the received parameters and prepares a new cloud model for the next round.

We take a slight leap from the already introduced notation in Table 2.2 in that each client can be considered like a data instance from server’s perspective. In this view, server’s optimization can be processed as it did in centralized case except that gradients corresponding to sampled data instances are replaced with pseudo-gradients corresponding to sampled clients. Table 7.1 lists the most frequently used symbols in this chapter along with their meanings. Note that the *aggregate model* (client gradients) is an average of *client model* (client gradients) over values of $i \in \mathcal{P}^t$.

$u^t, u_i^t, u_i^{t,\tau}$	variable u at {round t , and client i , and local step τ }
S^t, \mathcal{P}^t	set of { all, round } clients
$\theta^t, \bar{\theta}^t, \theta_i^t, \theta_i^{t,\tau}$	{ cloud, aggregate, client, local } model
$\mathcal{G}^t, \bar{g}^t, \hat{g}_i^t, g_i^{t,\tau}$	{ oracle, aggregate, client, local } gradients
h^t, h_i^t	{ full, client } gradients estimates

Table 7.1: Notation for FL.

7.4.2 Method

Upon receiving the client models of round t ($\{\forall i \in \mathcal{P}^t : \theta_i^t\}$) on the server, the aggregate model, $\bar{\theta}^t$ is computed by averaging them out. Next, the server finds the

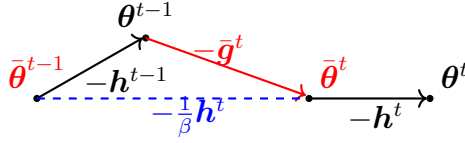


Figure 7.4: Geometric interpretation of [AdaBest](#)'s correction applied to the server updates. Server moves the aggregate parameters in the direction of $\bar{\theta}^{t-1} - \bar{\theta}^t$ before sending the models to the next round's clients.

cloud model θ^t by applying the estimate of the oracle gradients \mathbf{h}^t ; that is

$$\theta^t \leftarrow \bar{\theta}^t - \mathbf{h}^t, \quad (7.1)$$

where \mathbf{h}^t is found as follows

$$\mathbf{h}^t = \beta(\bar{\theta}^{t-1} - \bar{\theta}^t). \quad (7.2)$$

In section [7.4.3](#), we further discuss the criteria for chosen β which leads to a fast convergence. The described cycle continues by sending the cloud model to the clients sampled for the next round ($t + 1$) while the aggregate model ($\bar{\theta}^t$) is retained on the server to be used in calculation of \mathbf{h}^{t+1} or deployment. A schematic of the geometric interpretation of the additional drift removal step taken at the server is shown in [Figure 7.4](#).

Remark 1 *Aggregating client models by averaging is equivalent to applying a gradient step of size 1 from the previous round's cloud model using average of client pseudo-gradients or mathematically it is $\bar{\theta}^t \leftarrow \frac{1}{|S^t|} \sum_{i \in S^t} \theta_i^t = \theta^{t-1} - \bar{\mathbf{g}}^t$.*

After receiving the cloud model, each client $i \in \mathcal{P}^t$, optimizes its own copy of the model towards its local objective, during which the drift in the local optimization steps is reduced using the client's pseudo-gradients stored from the previous rounds (see [Algorithm 3](#)).

The modified client objective is $\arg \min_{\theta} \mathfrak{R}_i(\theta^t)$ such that

$$\mathfrak{R}_i(\theta^t) = \mathcal{L}_i(\theta^t) - \mu \langle \theta^t, \mathbf{h}_i^{t'_i} \rangle, \quad (7.3)$$

where \mathcal{L}_i is the local empirical risk defined by the task and data accessible by client i , and t'_i is the last round that client i participated in the training. Accordingly, the

local updates with step size η becomes

$$\boldsymbol{\theta}_i^{t,\tau} \leftarrow \boldsymbol{\theta}_i^{t,\tau-1} - \eta \left(\mathbb{E}_{\text{batch}}^{\boldsymbol{\theta}} [\nabla \ell_i(\boldsymbol{\theta}_i^{t,\tau-1})] - \mu \mathbf{h}_i^{t'} \right), \quad (7.4)$$

where $\mathbf{g}_i^{t,\tau-1} = \ell_i(\boldsymbol{\theta}_i^{t,\tau-1})$ and μ is the *regularization factor* (FedDyn has a similar factor; see Appendix D for further discussion on the choice of optimal value for μ). After the last local optimization step, each sampled client updates the estimate for its own local gradients and stores it locally to be used in the future rounds that the client participate in the training. This update is equivalent to $\mathbf{h}_i^t = \frac{1}{t-t'_i} \mathbf{h}_i^{t'} + \mathbf{g}_i^t$ where $\mathbf{g}_i^t = \boldsymbol{\theta}^{t-1} - \boldsymbol{\theta}_i^t$. Finally, the participating clients send the optimized model $\boldsymbol{\theta}_i^t$ back to the server. Our method along with SCAFFOLD and FedDyn is presented in Algorithm 3.

7.4.3 Relation to FL Baselines

Algorithm 3 demonstrates where our method differs from the baselines by color codes. Compared to the original SCAFFOLD, we made a slight modification in the way communication to the server occurs, preserving a quarter of the communication bandwidth usage. We refer to this modified version as SCAFFOLD/m. In the rest of this section, we will discuss the key similarities and differences between our algorithm, FedDyn and SCAFFOLD in terms of cost, robustness and functionality.

Cost

SCAFFOLD consumes twice as much communication bandwidth as FedDyn and AdaBest. This should be taken into account when comparing the experimental performance and convergence rate of these algorithms. All of these three algorithms require the same amount of storage on the server and on each client. Finally, AdaBest has a lower compute cost compared to FedDyn, SCAFFOLD both locally (on clients) and on the server. We provide quantitative comparison of these costs in Appendix D.

Robustness

According to the definition of cross-device FL, the number of devices could even exceed the number of examples per each device [McMahan et al., 2017]. In such a massive pool of devices, if the participating devices are drawn randomly at uniform

Algorithm 3 SCAFFOLD/m, FedDyn, and AdaBest

Input: T, θ^0, μ, β

for $t = 1$ **to** T **do**

Sample clients $\mathcal{P}^t \subseteq S^t$.

Transmit θ^{t-1} to each client in \mathcal{P}^t

Transmit h^{t-1} to each client in \mathcal{P}^t (SCAFFOLD/m)

for each client $i \in \mathcal{P}^t$ **in parallel do**

/ receive cloud model */*

$$\theta_i^{t,0} \leftarrow \theta^{t-1}$$

/ locally optimize for K local steps */*

for $k = 1$ **to** K **do**

Compute mini-batch gradients $\mathbf{g}_i^{t,k-1} = \mathbb{E}_{\text{batch}} [\nabla \ell_i(\theta_i^{t,\tau-1})]$

$$\mathcal{Q} \leftarrow \mathbf{g}_i^{t,k-1} - \mathbf{h}_i^{t'} + \mathbf{h}^t \text{ (SCAFFOLD/m)}$$

$$\mathcal{Q} \leftarrow \mathbf{g}_i^{t,k-1} - \mathbf{h}_i^{t'} - \mu(\theta^{t-1} - \theta_i^{t,k-1}) \text{ (FedDyn)} \quad \mathcal{Q} \leftarrow \mathbf{g}_i^{t,k-1} - \mathbf{h}_i^{t'} \text{ (AdaBest)}$$

$$\theta_i^{t,k} \leftarrow \theta_i^{t,k-1} - \eta \mathcal{Q}$$

end for

/ update local gradient estimates */*

$$\mathbf{g}_i^t \leftarrow \theta^{t-1} - \theta_i^{t,K}$$

$$\mathbf{h}_i^t \leftarrow \mathbf{h}_i^{t'} - \mathbf{h}^{t-1} + \frac{1}{K\eta} \mathbf{g}_i^t \text{ (SCAFFOLD/m)}$$

$$\mathbf{h}_i^t \leftarrow \mathbf{h}_i^{t'} + \mu \mathbf{g}_i^t \text{ (FedDyn)} \quad \mathbf{h}_i^t \leftarrow \frac{1}{t-t'} \mathbf{h}_i^{t'} + \mu \mathbf{g}_i^t \text{ (AdaBest)}$$

$$t'_i \leftarrow t$$

Transmit client model $\theta_i^t := \theta_i^{t,K}$.

end for

/ aggregate received models */*

$$\bar{\theta}^t \leftarrow \frac{1}{|\mathcal{P}^t|} \sum_{i \in \mathcal{P}^t} \theta_i^t$$

/ update oracle gradient estimates */*

$$\mathbf{h}^t \leftarrow \frac{|S^t| - |\mathcal{P}^t|}{|S^t|} \mathbf{h}^{t-1} + \frac{|\mathcal{P}^t|}{K\eta|S^t|} (\theta^{t-1} - \bar{\theta}^t) \text{ (SCAFFOLD/m)}$$

$$\mathbf{h}^t \leftarrow \mathbf{h}^{t-1} + \frac{|\mathcal{P}^t|}{|S^t|} (\theta^{t-1} - \bar{\theta}^t) \text{ (FedDyn)} \quad \mathbf{h}^t \leftarrow \beta(\bar{\theta}^{t-1} - \bar{\theta}^t) \text{ (AdaBest)}$$

/ update cloud model */*

$$\theta^t \leftarrow \bar{\theta}^t \text{ (SCAFFOLD/m)}$$

$$\theta^t \leftarrow \bar{\theta}^t - \mathbf{h}^t \text{ (FedDyn)} \quad \theta^t \leftarrow \bar{\theta}^t - \mathbf{h}^t \text{ (AdaBest)}$$

end for

(which our baselines premised upon), there is a small chance for a client to be sampled multiple times in a short period of time. In *FedDyn*, however, $\mathbf{h}^t = \sum_{\tau=1}^t \frac{|\mathcal{P}^\tau|}{|S^\tau|} \bar{\mathbf{g}}^\tau$, making it difficult for the norm of \mathbf{h} to decrease if pseudo-gradients in different rounds are not negatively correlated with each other (see Theorem 13 and its proof in Appendix D). In case clients are not re-sampled with a high rate then this negative correlation is unlikely to occur due to changes made to the state of the parameters and so the followup pseudo-gradients (see Section 7.4.3 for detailed discussion). A large norm of \mathbf{h}^t leads to a large norm of $\boldsymbol{\theta}^t$ and in turn a large $\|\bar{\boldsymbol{\theta}}^{t+1}\|^{(2)}$. This process is exacerbated during training and eventually leads to exploding norm of the parameters (see Figure 7.3). In Section 7.4.3, we intuitively justify the robustness of *AdaBest* for various scale and distribution of client sampling.

Theorem 13 (proof in Appendix D) *In FedDyn, $\|\mathbf{h}^t\|^2 \leq \|\mathbf{h}^{t-1}\|^2$ requires*

$$\cos(\angle(\mathbf{h}^{t-1}, \bar{\mathbf{g}}^t)) \leq -\frac{|\mathcal{P}^t|}{2|S^t|} \frac{\|\bar{\mathbf{g}}^t\|}{\|\mathbf{h}^{t-1}\|}. \quad (7.5)$$

Functionality

AdaBest allows to control how far to look back through the previous rounds for effective estimation of full and local gradients compared to existing *Reduced Variance LocalSGD (RV-LSGD)* baselines. To update the local gradient estimates, we dynamically scale the previous values down because the period between computing and using $\mathbf{h}_i^{t'}$ on client i (the period between two participation, i.e., $t - t'$) can be long during which the error of estimation may notable increase. See Algorithm 3 for comparing our updates on local gradients estimation compared to that of the baselines. Furthermore, at the server, \mathbf{h}^t is calculated as the weighted difference of two consecutive aggregate models. Note that, if expanded as a power series, this difference by itself is equivalent to accumulating pseudo-gradients across previous rounds with an exponentially weighted factor. This series is presented in Remark 14 for which the proof is provided in Appendix D. Unlike previous works, proposed pseudo-gradients' accumulation does not necessitate any additional effort to explicitly maintain information about the previous rounds. Additionally, it reduces the compute cost as quantitatively shown in Appendix D. It is a general arithmetic; therefore, could be adapted to work with our baselines as well.

Theorem 14 (Proof in Appendix D) *Cloud pseudo-gradients of [AdaBest](#) form a power series of $\mathbf{h}^t = \sum_{\tau=1}^t \beta^{(t-\tau)} \bar{\mathbf{g}}^t$, given that superscript in parenthesis means power.*

Adaptability

As indicated earlier, the error in estimation of oracle full gradients in [FedDyn](#) is only supposed to be eliminated by using pseudo-gradients. A difficult learning task, both in terms of optimization and heterogeneity results in a higher variance of pseudo-gradients when accompanies with a low rate of client participation. The outcome of constructing a naive estimator by accumulating these pseudo-gradients is severe. This is shown in [Figure 7.3](#), where on average, there is a long wait time between client re-samples due to the large number of participating clients. The results of this experiment empirically validates that $\|\boldsymbol{\theta}^t\|^{(2)}$ in [FedDyn](#) grows more rapidly and to a much higher value than [AdaBest](#). [SCAFFOLD](#) is prone to the same type of failure; however, because it scales down previous values of \mathbf{h} in its accumulation, the outcomes are less severe than that of [FedDyn](#). In [Appendix D](#), we present, similar analysis, for a much simpler task (classification on [EMNIST](#)). It is important not to confuse the source of [FedDyn](#)'s instability with overfitting (see supplementary material for overfitting analysis). However, our observations imply that the stability of [FedDyn](#) decreases with the difficulty of the task.

Our parameter β solves the previously mentioned problem with norm of \mathbf{h} . It is a scalar values between 0 and 1 that acts as a slider knob to determine the trade-off between the amount of information maintained from the previous estimate of full gradient and the estimation that a new round provides. On an intuitive level, a smaller β is better to be chosen for a more difficult task (both in terms of optimization and heterogeneity) and lower level participation—and correspondingly higher round to round variance among pseudo-gradients and vice versa. We provide an experimental analysis of β in [Appendix D](#); however, in a practical engineering level, β could be dynamically adjusted based on the variance of the pseudo-gradients. The goal of this chapter is rather showing the impact of using β . Therefore, we tune it like other hyper-parameters in our experiments. We leave further automation for finding an optimal β to be an open question for the future works.

Theorem 15 *FedAvg* is a special case of *AdaBest* where $\beta = \mu = 0$.

Theorem 16 *Server update of FedDyn* is a special case of *AdaBest* where $\beta = 1$ except that an extra $\frac{|\mathcal{P}|}{|\mathcal{S}|}$ scalar is applied which also adversely makes *FedDyn* require prior knowledge about the number of clients.

Theorem 17 *If S be a fixed set of clients, $\bar{\theta}$ does not converge to a stationary point unless $\mathbf{h} \rightarrow 0$.*

As mentioned in Section 7.4.3 and more particular with Theorem 13, *FedDyn* is only able to decrease norm of \mathbf{h} if pseudo-gradients are negatively correlated with oracle gradient estimates which could be likely only if the rate of client re-sampling is high. Therefore, with these conditions often being not true in large-scale FL and partial-participation, it struggles to converge to an optimal point. *SCAFFOLD* has a weighting factor that eventually could decrease $\|\mathbf{h}\|$ but it is not controllable. Our algorithm enables a direct decay of \mathbf{h} through decaying β . We apply this decay in our experiments when norm of \mathbf{h} plateaus (see Section 7.5.4). This is consistent with Theorem 17 which states that converging to a stationary point require $\mathbf{h} \rightarrow 0$.

7.5 Experiments

7.5.1 Setup

We evaluate performance and speed of convergence of our algorithm against state-of-the-art baselines. We concentrate on FL classification tasks defined using three well-known datasets. These datasets are, the letters classification task of *EMNIST* [LeCun et al., 1998] for an easy task, *CIFAR-10* [Krizhevsky et al., 2009] for a moderate task and *CIFAR-100* [Krizhevsky et al., 2009] for a challenging task. The training split of the dataset is partitioned randomly into a predetermined number of clients, for each task. 10% of these clients are set aside as validation clients and only used for evaluating the performance of the models during hyper-parameter tuning. The remaining 90% is dedicated to training. The entire test split of each dataset is used to evaluate and compare the performance of each model. Our assumption throughout the experiments is that, test dataset, oracle dataset, and collective data of validations clients have the same underlying distribution.

To ensure consistency with previous works, we follow [Acar et al. \[2020\]](#) to control heterogeneity and sample balance among client data splits. For heterogeneity, we evaluate algorithms in three modes: **IID**, $\alpha = 0.3$ and $\alpha = 0.03$. The first mode corresponds to data partitions (clients’ data) with equal class probabilities. For the second and third modes, we draw the skew in each client’s labels from a Dirichlet distribution with a concentration parameter α (see Section 5.6.3 for more details on how this draw is made). For testing against balance of sample number, we have two modes: *balanced* and *unbalanced* such that in the latter, the number of samples for each client is sampled from a log-normal distribution with concentration parameter equal to 0.3.

Throughout the experiments, we consistently keep the local learning rate, number of local epochs, and batch size as 0.1, 5 and 45 respectively. The local learning rate is decayed with a factor of 0.998 at each round. As tuned by [Acar et al. \[2020\]](#), the local optimizer uses a weight decay of 10^{-4} for the experiments on **EMNIST** and 10^{-3} for the experiment on **CIFAR-10** and **CIFAR-100**. Further details about the optimization is provided in supplementary material.

To tune the hyper-parameters we first launch each experiment for 500 rounds. μ of **FedDyn** is chosen from [0.002, 0.02, 0.2], with 0.02 performing best in all cases except **EMNIST**, where 0.2 also worked well. For the sake of consistency, we kept $\mu = 0.02$ for **AdaBest** as well. We found the rate of client participation to be an important factor for choosing a good value for β . Therefore, for 1% client participation experiments we search β in [0.2, 0.4, 0.6]. For higher rates of client participation, we use the search range of [0.94, 0.96, 0.98, 1.0]. For all these cases, 0.96 and 0.98 are selected for 10% and 100% client participation rates, respectively (both balanced and unbalanced). We follow [Acar et al. \[2020\]](#) for choosing the inference model by averaging client models though the rounds. Experiments are repeated 5 times, each with a different random seed of data partitioning. The reported mean and standard deviation of the performance are calculated based on the the last round of these 5 instance for each setting.

7.5.2 Model Architecture

We use the same model architectures as McMahan et al. [2017] and Acar et al. [2020]. For EMNIST, the architecture comprises of two fully-connected layers, each with 100 hidden units. For CIFAR-10 and CIFAR-100, there are two convolutional layers with 5×5 kernel size and 64 kernels each, followed by two fully-connected layers with 394 and 192 hidden units, respectively.

7.5.3 Baselines

We compare the performance of AdaBest against FedAvg [McMahan et al., 2017], SCAFFOLD [Karimireddy et al., 2020] and FedDyn [Acar et al., 2020]. These baselines are consistent with the ones that the closest work to us [Acar et al., 2020], has experimented with³. However, we avoided their choice of tuning the number of local epochs since we believe it does not comply with a fair comparison in terms of computation complexity and privacy loss.

7.5.4 Evaluation

Table 7.2 compare the performance of our model to all the baselines in various settings with 100 clients. The results show that our algorithm is effective in all settings. The 1000-device experiments confirm our arguments about the large-scale cross-device setting and practicality of AdaBest in comparison to the baselines. Our algorithm has notable gain both in the speed of convergence and the generalization performance. This gain is only overtaken for some benchmarks in full client participation settings (CP=100%) where the best β is chosen close to one. According to Remark 16, and the fact that in full participation $\frac{|\mathcal{P}^t|}{|S^t|} = 1$ and $t_i = t'_i + 1$ for all feasible i and t , FedDyn and AdaBest become nearly identical in these settings. In Figure 7.5, we show the impact of scaling the number of clients in both balanced and imbalanced settings for the same dataset and the same number of clients sampled per round (10 clients). During the hyper-parameter tuning we noticed that the sensitivity of FedDyn and AdaBest to their μ is small specially for the cases with larger number of clients.

³FedDyn additionally compares with FedProx [Li et al., 2020]; however, as shown in their benchmarks it performs closer to FedAvg than the other baselines.

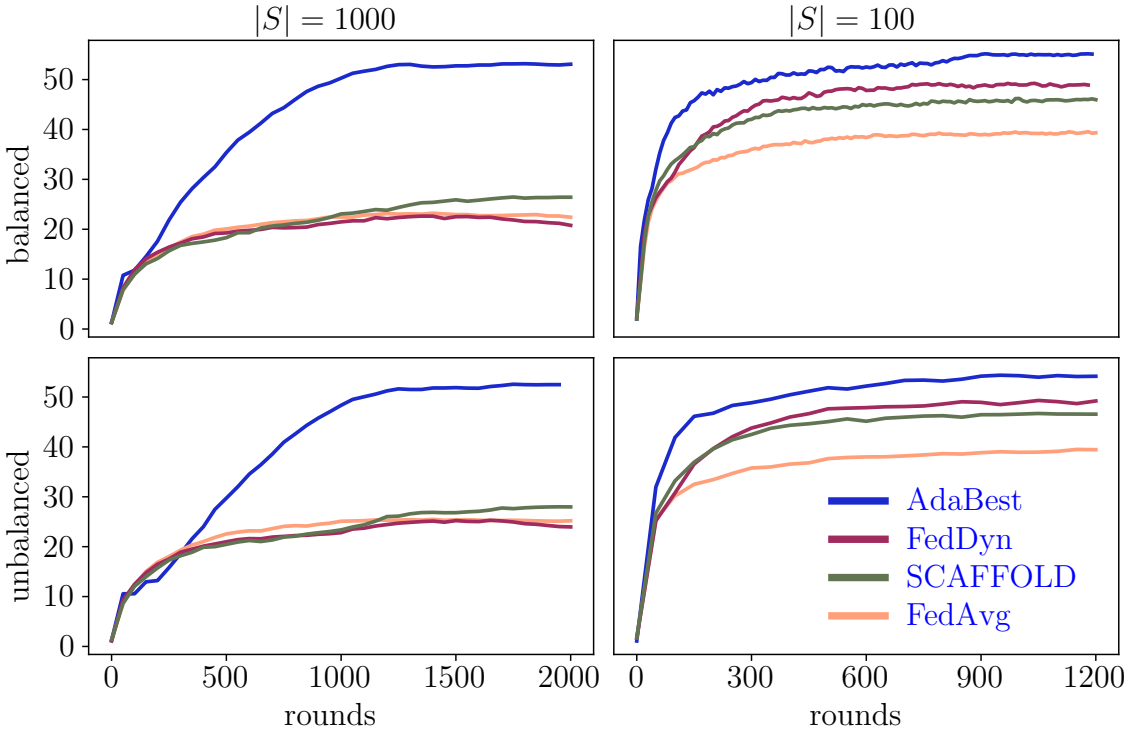


Figure 7.5: Test accuracy on balanced (top) and unbalanced (bottom) settings for training on 1000 (left) and 100 (right) clients. The training dataset is [CIFAR-100](#) and $|\mathcal{P}|=10$.

7.6 Conclusions

In this chapter, we introduce [AdaBest](#), an adaptive approach for tackling client drift in Federated Learning. Some of our most important contributions and findings in this chapter are as follows

- Unlike the existing solutions, our approach is robust to low rates of clients re-sampling, which makes it practical for large-scale cross-device Federated Learning.
- The performance and empirical convergence rates of our method demonstrate the efficacy of our technique compared to all the baselines across various benchmarks. In benchmarks with a large number of clients, our method outperforms the baselines by a wide margin. The results show nearly a two-fold improvement in test accuracy over the second-best candidate in some cases.
- Our algorithm consumes no more communication bandwidth or storage than

Table 7.2: Mean and standard deviation of test accuracy for various settings. The results are based on 5 random data partitioning seeds. Models are trained for 1k, 1.2k, and 2k rounds, for 1%, 10% and 100% client participation settings, respectively. A *smaller* α indicates *higher* heterogeneity. *CP stands for rate of client participation $\left(\frac{|P|}{|S|}\right)$

			top-1 test accuracy			
CP*	Dataset	Setting	FedAvg	FedDyn	SCAFFOLD	AdaBest
1%	EMNIST	$\alpha=0.03$	94.28±0.07	92.42±0.14	93.99±0.16	94.49±0.07
		$\alpha=0.3$	94.47±0.10	92.64±0.31	94.34±0.23	94.72±0.22
		IID	94.04±1.37	92.89±0.14	94.48±0.11	94.81±0.08
	CIFAR-10	$\alpha=0.03$	78.18±0.80	77.91±0.79	75.83±2.36	78.44±1.12
		$\alpha=0.3$	82.21±0.36	82.06±0.17	82.96±0.42	83.09±0.76
		IID	83.84±0.17	83.36±0.39	84.18±0.26	85.05±0.31
	CIFAR-100	$\alpha=0.03$	47.56±0.59	46.27±0.65	47.29±0.95	47.91±0.83
		$\alpha=0.3$	49.63±0.47	50.53±0.36	52.87±0.61	53.62±0.23
		IID	49.93±0.36	50.85±0.38	53.43±0.44	55.33±0.44
10%	EMNIST	$\alpha=0.03$	93.58±0.25	93.57±0.20	94.29±0.11	94.62±0.17
		$\alpha=0.3$	94.04±0.04	93.54±0.22	94.54±0.11	94.64±0.11
		IID	94.32±0.10	93.60±0.35	94.62±0.16	94.70±0.24
	CIFAR-10	$\alpha=0.03$	74.04±0.88	76.85±0.91	77.19±1.10	79.64±0.58
		$\alpha=0.3$	79.74±0.07	81.91±0.19	82.26±0.38	84.15±0.36
		IID	81.35±0.23	83.56±0.31	83.50±0.15	85.78±0.14
	CIFAR-100	$\alpha=0.03$	39.18±0.56	44.24±0.66	45.80±0.36	48.56±0.45
		$\alpha=0.3$	38.78±0.35	48.92±0.37	46.34±0.43	54.51±0.35
		IID	37.45±0.57	49.60±0.24	44.30±0.22	55.58±0.14
100%	EMNIST	$\alpha=0.03$	93.36±0.15	94.18±0.21	94.38±0.20	94.06±0.11
		$\alpha=0.3$	93.99±0.19	94.23±0.14	94.53±0.16	94.40±0.21
		IID	94.06±0.33	94.37±0.15	94.63±0.10	94.69±0.14
	CIFAR-10	$\alpha=0.03$	72.97±1.09	78.24±0.77	77.64±0.25	78.07±0.71
		$\alpha=0.3$	79.12±0.15	83.19±0.18	82.26±0.23	83.20±0.25
		IID	80.72±0.33	84.39±0.20	83.55±0.25	84.75±0.17
	CIFAR-100	$\alpha=0.03$	38.24±0.63	46.00±0.42	46.51±0.50	46.16±0.79
		$\alpha=0.3$	37.03±0.35	50.42±0.29	45.48±0.38	50.90±0.42
		IID	35.92±0.48	50.61±0.25	43.73±0.23	51.33±0.41

the baselines, and it even has a lower compute cost.

- [AdaBest](#) addresses the instability of norm of gradient estimates used in [FedDyn](#) by adapting to the most relevant information about the direction of the client drift.

- we formulated the general estimate of oracle gradients in a much elegant arithmetic that eliminates the need for the explicit, recursive form used in the previous algorithms.

Chapter 8

Conclusion and Future Research

In this thesis, we studied the efficiency and efficacy of [HTL](#) in the contexts of [task adaptation](#) and [FL](#). We first, reviewed popular gradient based algorithms for training [DNNs](#) and how they are used in the context of [HTL](#). We then went over the fundamental concepts that define [HTL](#) and [FL](#), as well as how these two fields are related. We continued by iterating over the existing methods for [HTL](#) and [FL](#) and discussed their limitations.

We further studied [feature-tuning](#) as the most popular [task adaptation](#) method and pronounced a major issue with the way it is conventionally applied. We hypothesized that this issue can be alleviated if starting from about zero, the magnitude of the updates to a pretrained feature-extractor gradually increases during the tuning. We proposed two methods, [ENTAME](#) and [FAST](#), to achieve such a gradual increase. Both of these methods initialize the weights of the [head](#) to small numbers so that the initial update to the pretrained feature-extractor becomes near zero. To make sure that parameters of the feature extractor are updated with a proper magnitude in the follow-up optimization steps, [ENTAME](#) uses a feature normalizer while [FAST](#) tunes a separate learning rate for the model’s [head](#). Based on various experiments conducted in this thesis about the speed of convergence, performance at the converged state and the amount of knowledge preserved from the source task, we make the following conclusions

- [feature-tuning](#) can converge to a significantly better solution compared to [feature-extraction](#);
- careful initialization of the [head](#) is crucial to an efficient [feature-tuning](#);
- initial updates to pretrained feature-extractors are noisy and can perturb the learned features resulting in loss of transferred knowledge;

- gradual increase in the magnitude of the updates to a pretrained feature-extractor not only better preserves the transferred knowledge but also improves the tuned model’s performance;
- [ENTAME](#) and [FAST](#) both achieve a gradual increase in the magnitude of the updates to a pretrained feature-extractor and as a result they both can accelerate [feature-tuning](#) and even sometimes converge to a better solution. [FAST](#) requires slightly less computation and storage overhead but has an extra hyperparameter to be tuned compared to [ENTAME](#). Therefore, depending on an application’s constraints, one of these methods can be preferred over the other.

On the subject of [FL](#), we targeted the issue of client drift and showed how existing [RV-LSGD](#) algorithms tackle it. We pronounced the bandwidth overhead and stability problem with the previous state-of-the-art algorithms, [SCAFFOLD](#) and [FedDyn](#). We provided a novel algorithm that not only elegantly solves these shortcoming but also significantly outperforms the existing methods in several cross-device settings. In summary, our contributions in the realm of heterogeneous [FL](#) are as follows.

- We showed that the existing [RV-LSGD](#) approaches for cross-device [FL](#) fail to efficiently converge to a stationary point. In particular, the norm of the parameters in [FedDyn](#) is pruned to explosion.
- We formulated a novel arithmetic approach for implicit accumulation of previous observations into estimates of the oracle full gradients (\mathbf{h}).
- We presented [AdaBest](#), a novel algorithm that can be thought as a generalization of both [FedAvg](#) and [FedDyn](#). We introduced a new factor β that stabilizes our algorithm through controlling the norm of \mathbf{h} . As a result, the optimization algorithm converges to a stationary point (see Sections [7.4.3](#) for detailed discussion). Unlike baselines, [AdaBest](#) does not assume that the set of training clients are stationary nor it requires a prior knowledge about its cardinality.
- We conducted various experiments under different settings of number of clients, balance among partitions, and data heterogeneity. Our results indicated superior performance of [AdaBest](#) in nearly all benchmarks (up to 94% improvement

in test accuracy compared to the second best candidate; almost twice better), in addition to significant improvements in stability and convergence rate.

8.1 Limitations and future research

The efficiency and efficacy of our proposed [task adaptation](#) methods [ENTAME](#) and [FAST](#), as well as our [FL](#) algorithm [AdaBest](#) has been shown in many cases in this thesis. However, there are several limitations to our proposed methods that should be acknowledged before they can be used in production.

[ENTAME](#) has a small storage and computation overhead; especially if the number of features is very large, a slightly larger memory is expected to be consumed, both in training and inference modes. Additionally, it uses the training batch statistics, which can be unreliable if batches are not drawn [IID](#) or if training with moderately large training batches is impossible.

Furthermore, while [FAST](#) does not share the same limitation as [ENTAME](#), its reliance on an additional learning rate can lead to a more time-consuming process of tuning hyperparameters. Addressing this requirement may necessitate further exploration to optimize the tuning process and minimize computation costs.

Finally, despite its effectiveness, [AdaBest](#) has certain limitations that require further exploration. Firstly, similar to its precedents, our method addresses the issue of client drift abstractly without taking into account privacy concerns which presents an open research question. Secondly, [AdaBest](#) ignores the heterogeneity that results from the presence of stragglers, an issue typically addressed in asynchronous [FL](#). Finally, our method relies on an additional hyperparameter β which is not present in previous methods, and it can add some burden to the training process in some specific settings.

In terms of theoretical analysis, some previous [FL](#) work investigated their bounds for convergence rate given convex and strongly convex loss landscapes. Although such analysis can help comparing different algorithms, it does not provide a certain lead to practical outcomes. Perhaps, this is mainly because, when dealing with [DNNs](#), we often anticipate a highly non-convex loss landscape. Even for convex and strongly convex settings, we found it challenging to adapt the existing analysis to [AdaBest](#). As much as the existence of β makes our algorithm practically appealing, it makes

many proofs provided by [Acar et al. \[2020\]](#) inapplicable to [AdaBest](#). For instance, [Acar et al. \[2020\]](#) simplify several telescope series in proving their theorems, whereas adapting the same theorems to [AdaBest](#) fails because given our discounting factor, those series do not appear in many formula. We leave further theoretical analysis to the future work.

In summary, our proposed methods have shown outstanding results, but there are several limitations that need to be addressed in future research. These limitations are important and may affect the applicability of our methods.

Appendix A

Supplementary material for Chapter 2

“To define is to limit.”

- Oscar Wilde, *The Picture of Dorian Gray*

A.1 Proofs

Lemma 1 For an unconstrained optimization problem (i.e., all directions be *admissible* in parameter space), if $\boldsymbol{\theta}$ is not a minimizer then \mathbf{d} is an ascend direction of the differentiable loss function $\mathcal{L}(\boldsymbol{\theta})$ if $\langle \mathcal{G}, \mathbf{d} \rangle > 0$.

Proof Let r be a small positive scalar. By gradient definition $\mathcal{G} = \lim_{r \rightarrow 0} \frac{\mathcal{L}(\boldsymbol{\theta} + r\mathbf{d}) - \mathcal{L}(\boldsymbol{\theta})}{rd}$. If $\langle \mathcal{G}, \mathbf{d} \rangle > 0$, then $\lim_{r \rightarrow 0} (\mathcal{L}(\boldsymbol{\theta} + r\mathbf{d}) - \mathcal{L}(\boldsymbol{\theta})) > 0$. As a result, $\mathcal{L}(\boldsymbol{\theta} + r\mathbf{d}) > \mathcal{L}(\boldsymbol{\theta})$ which means that \mathbf{d} is a ascend direction. ■

Lemma 2 For the differentiable loss function $\mathcal{L}(\boldsymbol{\theta})$ the gradient is the steepest ascend direction.

Proof Let $\|\mathbf{d}\|_2 = 1$. Using the assumptions in Lemma 1 and by Cauchy-Swcharz inequality, we have $\langle \mathcal{G}, \mathbf{d} \rangle \leq \|\mathcal{G}\| \|\mathbf{d}\|$ which means $\langle \mathcal{G}, \mathbf{d} \rangle \leq \|\mathcal{G}\|$. Therefore, $\langle \mathcal{G}, \mathbf{d} \rangle$ gets to its maximum at the direction of $\mathbf{d} = \frac{\mathcal{G}}{\|\mathcal{G}\|}$. ■

Theorem 1 (Steepest descent direction) For the differentiable loss function $\mathcal{L}(\boldsymbol{\theta})$ the negative of the gradient is the steepest direction of descending.

Proof Given Lemma 2, we have $\overset{\boldsymbol{\theta}}{\nabla}(-\mathcal{L}(\boldsymbol{\theta})) = -\overset{\boldsymbol{\theta}}{\nabla}\mathcal{L}(\boldsymbol{\theta})$. ■

Corollary 1 In GD, the convergence is achieved when $\|\mathcal{G}^{t-1}\| < \frac{\epsilon}{\eta}$.

Proof The proposition can be directly found given Definition 2 and GD's update rule in Equation (2.4). ■

Theorem 2 (SGD unbiased) *Stochastic Gradient Descend (SGD) is an unbiased estimator for Gradient Descend (GD).*

Proof based on Equation (2.5) we can consider the pseudo-gradient at θ^0 to θ^m as $\hat{\mathbf{g}} \triangleq \theta^0 - \theta^m = \eta \sum_{t=1}^m \overset{\theta}{\nabla} \ell(\mathbf{x}, \mathbf{y}; \theta^{t-1})$. To prove the theorem, we show that starting from the same state of the parameters θ^0 , expectation of average of m consecutive steps of SGD equals average of m consecutive steps of GD. This is equivalent to showing $\mathbb{E}[\hat{\mathbf{g}}] - \eta \sum_{j=1}^m \mathcal{G}^j = 0$ where the expectation is over all possible orders (random shuffles) of data instances and \mathcal{G}^j is full-batch gradient of the j -th GD step. We have

$$\begin{aligned} \mathbb{E}[\hat{\mathbf{g}}] &= \mathbb{E}\left[\eta \sum_{t=1}^m \overset{\theta}{\nabla} \ell(\mathbf{x}_i, \mathbf{y}_i; \theta^{t-1})\right] \\ &= \eta \sum_{t=1}^m \mathbb{E}_i\left[\overset{\theta}{\nabla} \ell(\mathbf{x}_i, \mathbf{y}_i; \theta^{t-1})\right]. \end{aligned} \quad (\text{A.1})$$

Because i is sampled at uniform, probability of each instance drawn is equally $\frac{1}{m}$ so at $t = j$,

$$\mathbb{E}_i\left[\overset{\theta}{\nabla} \ell(\mathbf{x}_i, \mathbf{y}_i; \theta^{t-1}) \mid t = j\right] = \frac{1}{m} \sum_{i=1}^m \overset{\theta}{\nabla} \ell(\mathbf{x}_j, \mathbf{y}_j; \theta^{j-1}) = \mathcal{G}^j. \quad (\text{A.2})$$

On the other hand, the marginal expectation gives

$$\mathbb{E}_i\left[\overset{\theta}{\nabla} \ell(\mathbf{x}_i, \mathbf{y}_i; \theta^{t-1})\right] = \frac{1}{m} \sum_{j=1}^m \mathbb{E}_i\left[\overset{\theta}{\nabla} \ell(\mathbf{x}_i, \mathbf{y}_i; \theta^{t-1}) \mid t = j\right] \quad (\text{A.3})$$

From Equation (A.2) and Equation (A.3) we have

$$\mathbb{E}_i\left[\overset{\theta}{\nabla} \ell(\mathbf{x}_i, \mathbf{y}_i; \theta^{t-1})\right] = \frac{1}{m} \sum_{j=1}^m \mathcal{G}^j. \quad (\text{A.4})$$

Plugging this into Equation (A.1) gives

$$\mathbb{E}[\hat{\mathbf{g}}] = \eta \sum_{t=1}^m \frac{1}{m} \sum_{j=1}^m \mathcal{G}^j = \eta \sum_{j=1}^m \mathcal{G}^j. \quad (\text{A.5})$$

■

Theorem 3 *SVRG is an unbiased estimate of GD.*

Proof Let $\nu = m$ and similar to theorem 2 but starting from m -th update, $\hat{\mathbf{g}} \triangleq \boldsymbol{\theta}^m - \boldsymbol{\theta}^{2m}$ or

$$\hat{\mathbf{g}} \triangleq \eta \sum_{t=m}^{2m} \left(\nabla \ell(\mathbf{x}_i, \mathbf{y}_i; \boldsymbol{\theta}^{t-1}) + \nabla \mathcal{L}(\tilde{\boldsymbol{\theta}}) - \nabla \ell(\mathbf{x}_i, \mathbf{y}_i; \tilde{\boldsymbol{\theta}}) \right). \quad (\text{A.6})$$

We need to show that $\mathbb{E}[\hat{\mathbf{g}}] = \eta \sum_{j=1}^m \mathcal{G}^j$; starting from the left side we have

$$\begin{aligned} \mathbb{E}[\hat{\mathbf{g}}] &= \mathbb{E}_i \left[\eta \sum_{t=m}^{2m} \left(\nabla \ell(\mathbf{x}_i, \mathbf{y}_i; \boldsymbol{\theta}^{t-1}) + \nabla \mathcal{L}(\tilde{\boldsymbol{\theta}}) - \nabla \ell(\mathbf{x}_i, \mathbf{y}_i; \tilde{\boldsymbol{\theta}}) \right) \right] \\ &= \eta \sum_{t=m}^{2m} \mathbb{E}_i \left[\nabla \ell(\mathbf{x}_i, \mathbf{y}_i; \boldsymbol{\theta}^{t-1}) \right] + \eta \sum_{t=m}^{2m} \mathbb{E}_i \left[\nabla \mathcal{L}(\tilde{\boldsymbol{\theta}}) \right] - \eta \sum_{t=m}^{2m} \mathbb{E}_i \left[\nabla \ell(\mathbf{x}_i, \mathbf{y}_i; \tilde{\boldsymbol{\theta}}) \right]. \end{aligned} \quad (\text{A.7})$$

Getting help from theorem 2, we can simplify this equation to

$$\mathbb{E}[\hat{\mathbf{g}}] = \eta \sum_{t=m}^{2m} \mathbb{E}_i \left[\nabla \ell(\mathbf{x}_i, \mathbf{y}_i; \boldsymbol{\theta}^{t-1}) \right] + \nabla \mathcal{L}(\tilde{\boldsymbol{\theta}}) - \nabla \mathcal{L}(\tilde{\boldsymbol{\theta}}). \quad (\text{A.8})$$

This takes us to the expectation of $\hat{\mathbf{g}}$ in theorem 2 which follows the same result. ■

Theorem 4 (velocity form of SGD with momentum) *SGD with momentum updates described by Equation (2.6), Equation (2.11), and Equation (2.12) is equivalent to applying*

$$\boldsymbol{\theta}^t = \boldsymbol{\theta}^{t-1} - \eta \mathbf{v}^t; \quad (\text{A.9})$$

where

$$\mathbf{v}^t = \beta \mathbf{v}^{t-1} + \nabla \ell(\mathbf{x}_i, \mathbf{y}_i; \boldsymbol{\theta}^{t-1}); \quad i \sim \mathcal{U}(1, m). \quad (\text{A.10})$$

Proof From Equation (2.13),

$$\mathbf{v}^{t-1} = \frac{1}{\eta} (\boldsymbol{\theta}^{t-2} - \boldsymbol{\theta}^{t-1}). \quad (\text{A.11})$$

Replacing it in Equation (2.14) gives

$$\mathbf{v}^t = \frac{\beta}{\eta} (\boldsymbol{\theta}^{t-2} - \boldsymbol{\theta}^{t-1}) + \nabla \ell(\mathbf{x}_i, \mathbf{y}_i; \boldsymbol{\theta}^{t-1}). \quad (\text{A.12})$$

Plugging this into Equation (2.13) yields

$$\boldsymbol{\theta}^t = \boldsymbol{\theta}^{t-1} - \eta \left(\frac{\beta}{\eta} (\boldsymbol{\theta}^{t-2} - \boldsymbol{\theta}^{t-1}) + \nabla \ell(\mathbf{x}_i, \mathbf{y}_i; \boldsymbol{\theta}^{t-1}) \right), \quad (\text{A.13})$$

in which setting $\mathbf{h}^t = \frac{\beta}{\eta}(\boldsymbol{\theta}^{t-2} - \boldsymbol{\theta}^{t-1})$ and adding a zero vector \mathbf{h}_i^t gives what we are looking for. ■

Appendix B

Supplementary material for Chapter 5

B.1 Proofs

Theorem 5 (Gradient w.r.t. head's output) *The gradient of cross-entropy loss with respect to head's output is equal to the prediction error.*

Proof Equation (2.1) can equally be written as

$$\ell(\mathbf{y}, \hat{\mathbf{y}}) = - \sum_{i=1}^{i=C} y_i \ln \hat{y}_i. \quad (\text{B.1})$$

Replacing \hat{y}_i from Equation (5.5) gives

$$\begin{aligned} \ell(\mathbf{o}, \mathbf{y}) &= - \sum_{i=1}^{i=|y|} y_i \ln \frac{\exp(o_i)}{\sum_{j=1}^{j=|y|} \exp(o_j)} \\ &= - \sum_{i=1}^{i=C} y_i \ln(\exp(o_i)) + \sum_{i=1}^{i=C} y_i \ln\left(\sum_{j=1}^{j=|y|} \exp(o_j)\right) \\ &= - \sum_{i=1}^{i=C} y_i o_i + \sum_{i=1}^{i=C} y_i \ln\left(\sum_{j=1}^{j=C} \exp(o_j)\right) \end{aligned} \quad (\text{B.2})$$

The gradient of loss with respect to k -th head's output is

$$\frac{\partial \ell}{\partial o_k} = -y_k + \sum_{i=1}^{i=|y|} y_i \frac{\exp(o_k)}{\sum_{j=1}^{j=C} \exp(o_j)}. \quad (\text{B.3})$$

The last fraction redefines k -th outputs of softmax(\mathbf{o}); therefore,

$$\frac{\partial \ell}{\partial o_k} = -\mathbf{y}_k + \sum_{i=1}^{i=C} y_i \hat{y}_k. \quad (\text{B.4})$$

However, since \mathbf{y} is one-hot encoded

$$\frac{\partial \ell}{\partial o_k} = \hat{y}_k - y_k. \quad (\text{B.5})$$

Thus, according to Definition 7 we have

$$\frac{\partial \ell}{\partial \mathbf{o}} = \boldsymbol{\delta}. \quad (\text{B.6})$$

■

Theorem 6 (Maximum entropy of Predicted Labels) *Maximum entropy of predicted class labels is achieved when all classes are predicted equally.*

Proof Based on Definition 8, we have

$$H(\mathbf{y}) := \sum_{i=1}^{i=C} \hat{y}_i \ln \frac{1}{\hat{y}_i}. \quad (\text{B.7})$$

\ln is a concave function; therefore, based on Jensen's inequality

$$H(\mathbf{y}) \leq \ln \sum_{i=1}^{i=C} \hat{y}_i \frac{1}{\hat{y}_i} = \ln C. \quad (\text{B.8})$$

This maximum is achieved only if $\forall i, j \in \{1, 2, \dots, C\} : \hat{y}_i = \hat{y}_j$.

■

Axiom 1 $\forall u, v \in \mathbb{R}$, if $u \geq 0$ and $v \geq 0$ then $uv \geq 0$.

Lemma 3 $\forall u \in \mathbb{R}$, if $0 \leq u \leq 1$ then $u^{(2)} \leq u$.

Proof We have

$$1 - u \geq 0. \quad (\text{B.9})$$

Axiom 1 gives

$$u(1 - u) \geq 0; \quad (\text{B.10})$$

which means

$$u^{(2)} \geq u; \quad (\text{B.11})$$

■

Axiom 2 $\forall \mathbf{u}, \mathbf{v} \in \mathbb{R}^{(c)}$, $\langle \mathbf{u}, \mathbf{v} \rangle \leq \langle \mathbf{u}, \mathbf{u} \rangle \langle \mathbf{v}, \mathbf{v} \rangle$ (Cauchy-Schwarz)

Axiom 3 *Taylor series approximation of exponential function around zero is*

$$\exp(u) \approx 1 + u. \quad (\text{B.12})$$

Theorem 7 (Maximum initial expected entropy in ENTAME) *ENTAME makes the expected entropy of predicted labels initially maximized.*

Proof *Using Axiom 3 and Definition 4, we have*

$$\hat{y}_i = \frac{1 + o_i}{C + \sum_{j=1}^{j=C} o_j}. \quad (\text{B.13})$$

Replacing the normalized features in Equation (5.4) give

$$o_i = \langle \mathfrak{A}, \mathbf{W}_i \rangle + b_i \quad (\text{B.14})$$

for the i -th output. Additionally, given the mean and variance of elements in \mathbf{W}^0 and the normalized features we can conclude $\mathbb{E}[o_i] = 0$ so

$$\mathbb{E}[\hat{y}_i] = \frac{1}{C}, \quad (\text{B.15})$$

which according to Theorem 6 is expectantly maximum prediction entropy. ■

Lemma 4 (Initial output mean and variance in ENTAME) *ENTAME gives $\forall i \in \{1, 2, \dots, C\} : \mathbb{E}[o_i^0] = 0, \text{Var}(o_i^0) = Q\sigma^{(2)}$.*

Proof \mathbf{W}_i^0 *contains all zero-centered elements, so $o_i = \langle \mathfrak{A}^0, \mathbf{W}_i^0 \rangle$ is sum of all zero-centered elements which is zero-centered. Therefore, $\mathbb{E}[o_i^0] = 0$. As for the Variance, because \mathbf{W}_i^0 and \mathfrak{A}^0 are independent*

$$\begin{aligned} \text{Var}(o_i^0) &= \text{Var}(\langle \mathfrak{A}^0, \mathbf{W}_i^0 \rangle) \\ &= Q\text{Var}(\mathfrak{A}^0)\text{Var}(\mathbf{W}_i^0) \\ &= Q\sigma^{(2)}. \end{aligned} \quad (\text{B.16})$$

■

Corollary 2 *If $\forall i, j \in \{1, 2, \dots, C\} : \hat{y}_i = \hat{y}_j$ then $\langle \boldsymbol{\delta}, \boldsymbol{\delta} \rangle = \frac{C-1}{C}$.*

Proof Because predicted labels have to sum to one, then based on their equality proposition, we have $\forall i \in \{1, 2, \dots, C\} : \hat{y}_i = \frac{1}{C}$. Let k be the index of the true class; then, based on the Definition 7 we have

$$\delta_i = \begin{cases} \frac{1}{C} - 1 & \text{if } i=k; \\ \frac{1}{C} & \text{otherwise.} \end{cases} \quad (\text{B.17})$$

Accordingly, we have

$$\begin{aligned} \langle \boldsymbol{\delta}, \boldsymbol{\delta} \rangle &= \sum_{i=1}^{i=C} \delta_i^{(2)} \\ &= (C-1) \frac{1}{C^{(2)}} + \left(\frac{1}{C} - 1\right)^{(2)} \\ &= \frac{1}{C} - \frac{1}{C^{(2)}} + \frac{1}{C^{(2)}} - 2\frac{1}{C} + 1 \\ &= \frac{C-1}{C}. \end{aligned} \quad (\text{B.18})$$

■

Corollary 3 If $\forall i, j \in \{1, 2, \dots, C\} : \hat{y}_i = \hat{y}_j$, a large C implies $\langle \boldsymbol{\delta}, \boldsymbol{\delta} \rangle \approx 1$.

Proof If $C \gg 1$ then $C \approx C - 1$. Given the result of Corollary 3, this means $\langle \boldsymbol{\delta}, \boldsymbol{\delta} \rangle \approx 1$. ■

Lemma 5 The first update to *head's* weights in *ENTAME* is such that

$$\mathbf{W}_i^1 \approx \mathbf{W}_i^0 + \eta \mathbb{E}_{\text{batch}} [y_i^0 \boldsymbol{\alpha}^0]. \quad (\text{B.19})$$

Proof Given the aforementioned feature normalizer, the first forwarded mini-batch of training data, for the i -th row of \mathbf{W} the gradient is

$$\frac{\partial \mathcal{L}^0}{\partial \mathbf{W}_i^0} = \mathbb{E}_{\text{batch}} \left[\frac{\partial \ell^0}{\partial \mathbf{W}_i^0} \right] = \mathbb{E}_{\text{batch}} [\delta_i^0 \boldsymbol{\alpha}^0], \quad (\text{B.20})$$

or given Theorem 7

$$\frac{\partial \mathcal{L}^0}{\partial \mathbf{W}_i^0} \approx \frac{1}{C} \mathbb{E}_{\text{batch}} [\boldsymbol{\alpha}^0] - \mathbb{E}_{\text{batch}} [y_i^0 \boldsymbol{\alpha}^0]. \quad (\text{B.21})$$

Accordingly, using *mini-batch SGD* and $\mathbb{E}_{\text{batch}} [\mathfrak{A}^0] = \mathbf{0}$ we have

$$\mathbf{W}_i^1 \approx \mathbf{W}_i^0 + \eta \mathbb{E}_{\text{batch}} [y_i^0 \mathfrak{A}^0]. \quad (\text{B.22})$$

■

Theorem 8 *ENTAME* guarantees

$$\|\mathbf{W}^1\|_F \geq \|\mathbf{W}^0\|_F \quad (\text{B.23})$$

Proof The minimum of $\|\mathbf{W}^1\|_F^{(2)}$ is found when all instances of the batch are from the same class. If that is the k -th class then according to Lemma 5, we have

$$\mathbf{W}_k^1 = \mathbf{W}_k^0 + \eta \mathbb{E}_{\text{batch}} [\mathfrak{A}^0] = \mathbf{W}_k^0 \quad (\text{B.24})$$

which is resulted from $\mathbb{E}_{\text{batch}} [\mathfrak{A}^0] = 0$. Therefore, $\|\mathbf{W}^1\|_F^{(2)}$ remains at $\|\mathbf{W}^1\|_F^{(2)} = QC\sigma^2$. Based on triangle inequality, any other condition results in larger norm square across mini-batch dimension and so larger Frobenius norm of \mathbf{W}^1 . ■

Corollary 4 When all instances from the first mini-batch are from the same class, *ENTAME* gives

$$\|\mathbf{W}^1\|_F = \|\mathbf{W}^0\|_F \quad (\text{B.25})$$

Proof This can be directly concluded from the proof of Theorem 8 and in particular Equation (B.24). ■

Theorem 9 Maximum entropy prediction labels is achieved when $\forall c, c' \in \{1, 2, \dots, C\} : \mathbf{W}_c = \mathbf{W}_{c'}$ and $b_c = b_{c'}$.

Proof Given an extracted feature vector \mathbf{a} , each outputs o_i is found as

$$o_c = \langle \mathbf{a}, \mathbf{W}_c \rangle + b_c. \quad (\text{B.26})$$

Therefore, from the proposition we can conclude that $\forall c, c' \in \{1, 2, \dots, C\} : o_c = o_{c'}$. Replacing this into *softmax* (Definition 4) gives

$$\forall c \in \{1, 2, \dots, C\} : \hat{y}_c = \frac{\exp(o_c)}{C \exp(o_c)} = \frac{1}{C}, \quad (\text{B.27})$$

which is the maximum entropy prediction labels according to Theorem 6. ■

Theorem 10 (rows stays equal) *Let \mathcal{C}' be the set of classes that were never observed by the model during training. If *mini-batch SGD* is used as the optimization algorithm, then $\forall c, c' \in \mathcal{C}' : \mathbf{W}_c = \mathbf{W}_{c'}$ and $b_c = b_{c'}$ stays true as far as no training instance from classes c and c' is observed.*

Proof According to Theorem 9, stated conditions for \mathbf{W} and \mathbf{b} at time t gives $\hat{y}_c^t = \hat{y}_{c'}^t$; therefore, $\delta_c^t = \delta_{c'}^t$ (see Theorem 5). We have

$$\frac{\partial \ell^t}{\partial \mathbf{W}_c^t} = \frac{\partial \ell^t}{\partial \mathbf{W}_{c'}^t}. \quad (\text{B.28})$$

The *Stochastic Gradient Descend (SGD)* update results in $\mathbf{W}_c^{t+1} = \mathbf{W}_{c'}^{t+1}$ and $b_c^{t+1} = b_{c'}^{t+1}$. Therefore, by induction the condition stays true as far as training instances of c and c' are not observed. ■

Theorem 11 *Given condition expressed in Theorem 9 we have $\mathbb{E}_{\text{batch}} [\overset{\mathbf{a}}{\nabla} \ell] = \mathbf{0}$ for the first optimization step using *mini-batch SGD*.*

Proof From Equation 5.10 and the proposition, we have

$$\overset{\mathbf{a}_i}{\nabla} \ell = \sum_{c=1}^{c=C} \delta_c \mathbf{W}_{c,i}. \quad (\text{B.29})$$

At the beginning of training we have

$$\overset{\mathbf{a}_i}{\nabla} \ell = u_i \sum_{c=1}^{c=C} \delta_c. \quad (\text{B.30})$$

given that $\{1, 2, \dots, C\} \in: \mathbf{u} = \mathbf{W}_c$. This in turn gives

$$\overset{\mathbf{a}_i}{\nabla} \ell = u_i \left(\sum_{c=1}^{c=C} \hat{y}_c - \sum_{c=1}^{c=C} y_c \right) = 0. \quad (\text{B.31})$$

■

B.2 Complementary Experiments

B.2.1 task adaptation Performance

In Section 5.6.1, we compared the ENTAME with the baselines for ResNet-50 as the model. In this section we provide results for many more model architectures in Figures B.2 to B.18.

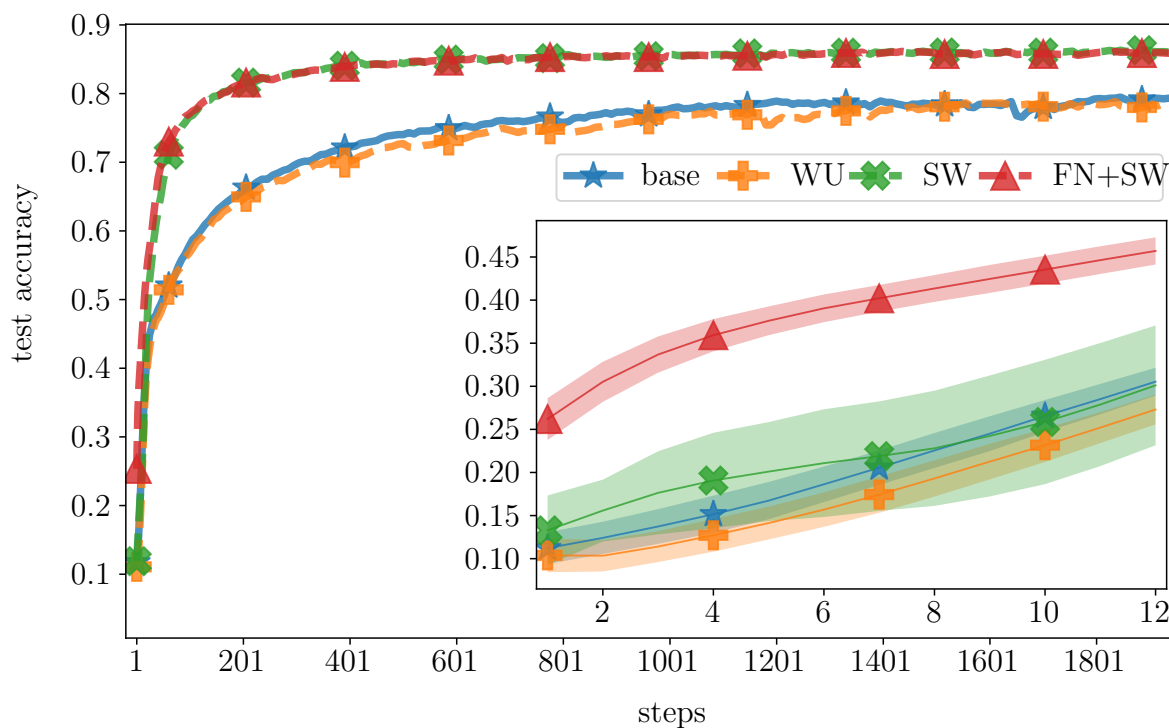


Figure B.1: Feature-tuning ResNet-152, ImageNet \mapsto CIFAR-10.

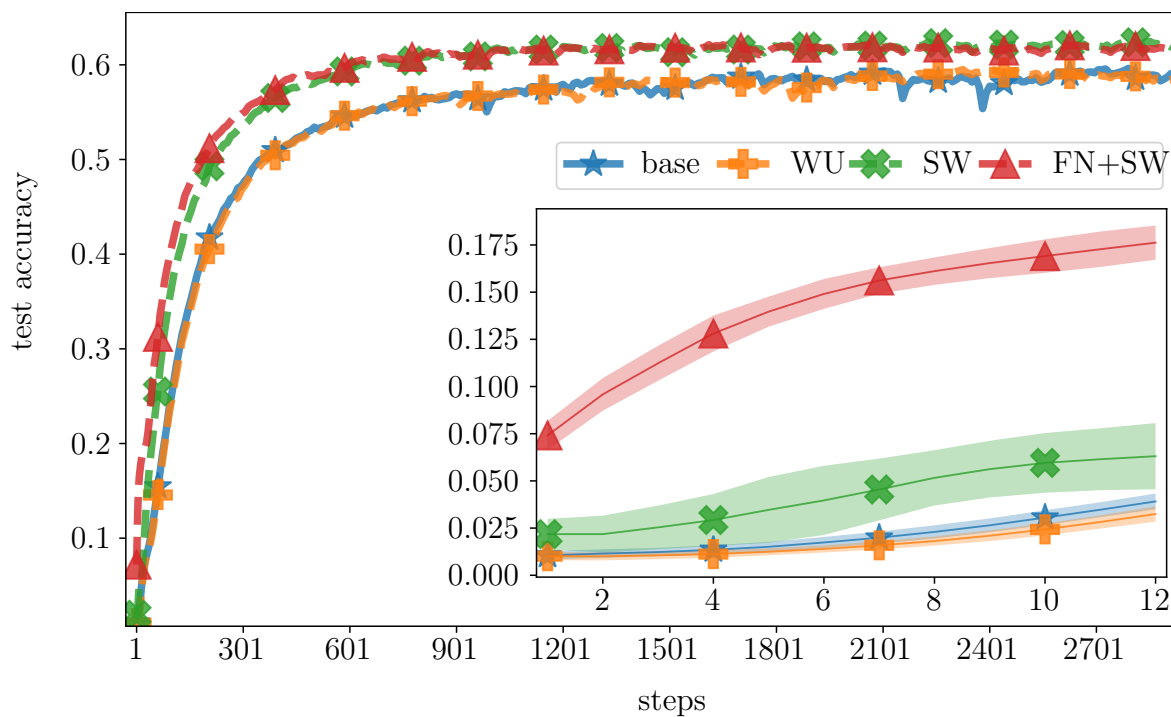


Figure B.2: Feature-tuning ResNet-152, ImageNet \mapsto CIFAR-100.

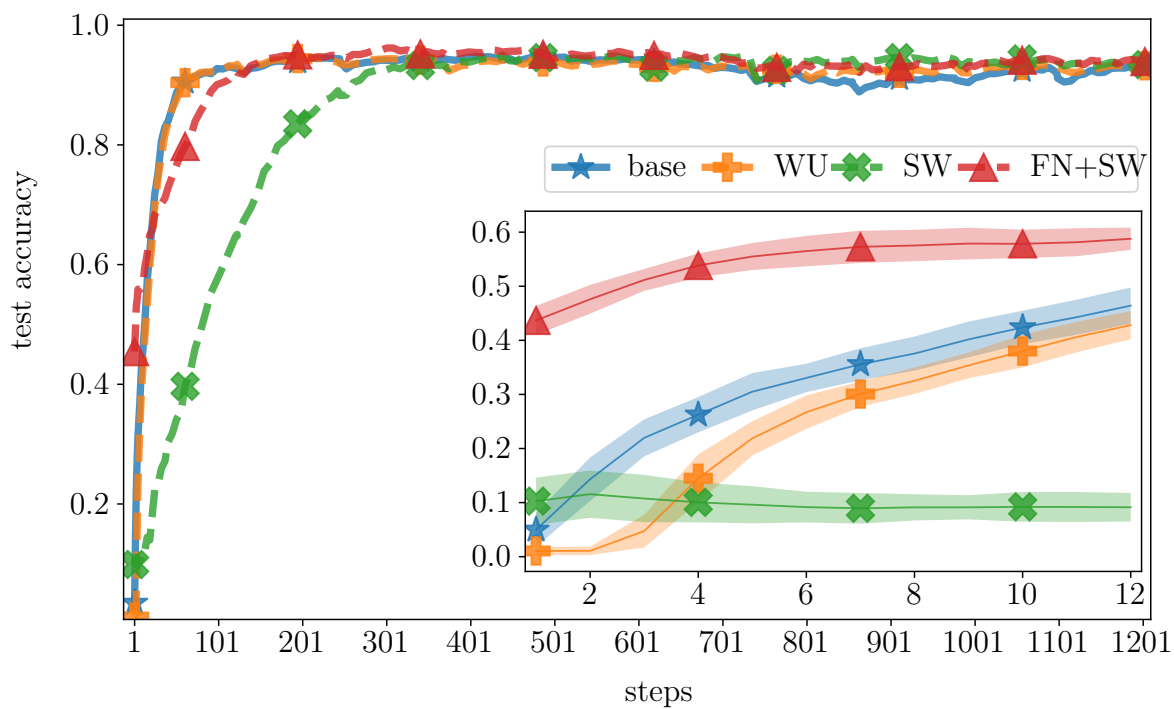


Figure B.3: Feature-tuning ResNet-152, ImageNet \mapsto Caltech-101.

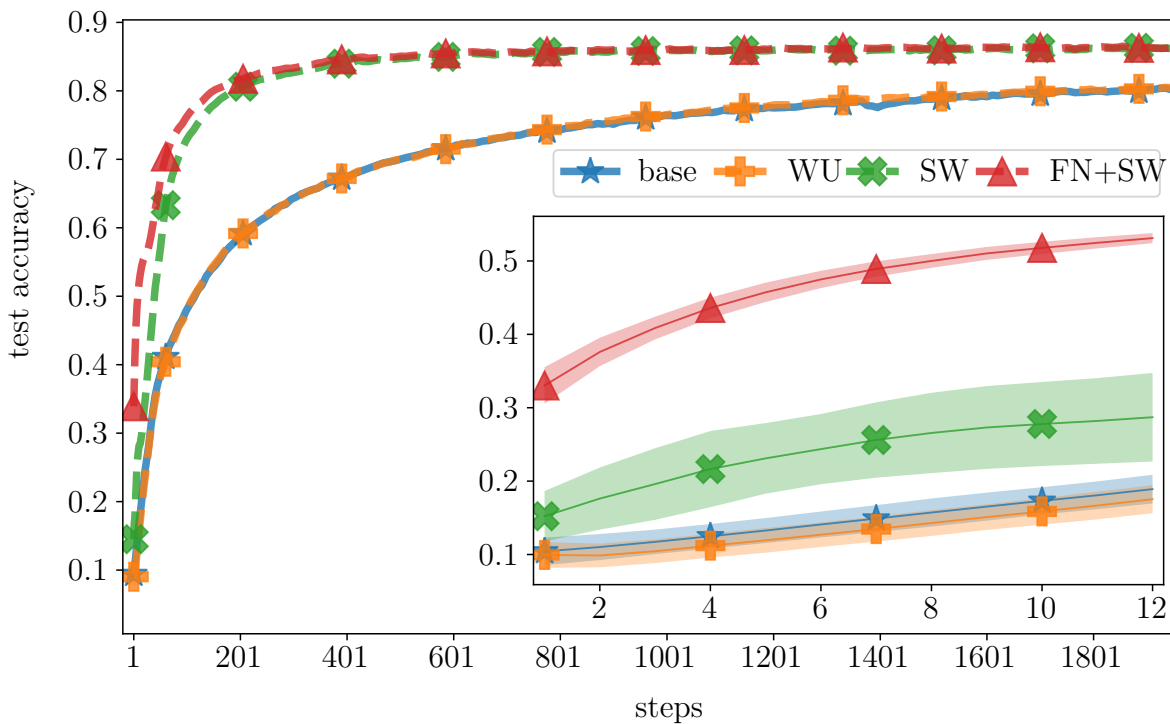


Figure B.4: Feature-tuning DenseNet-121, ImageNet \mapsto CIFAR-10.

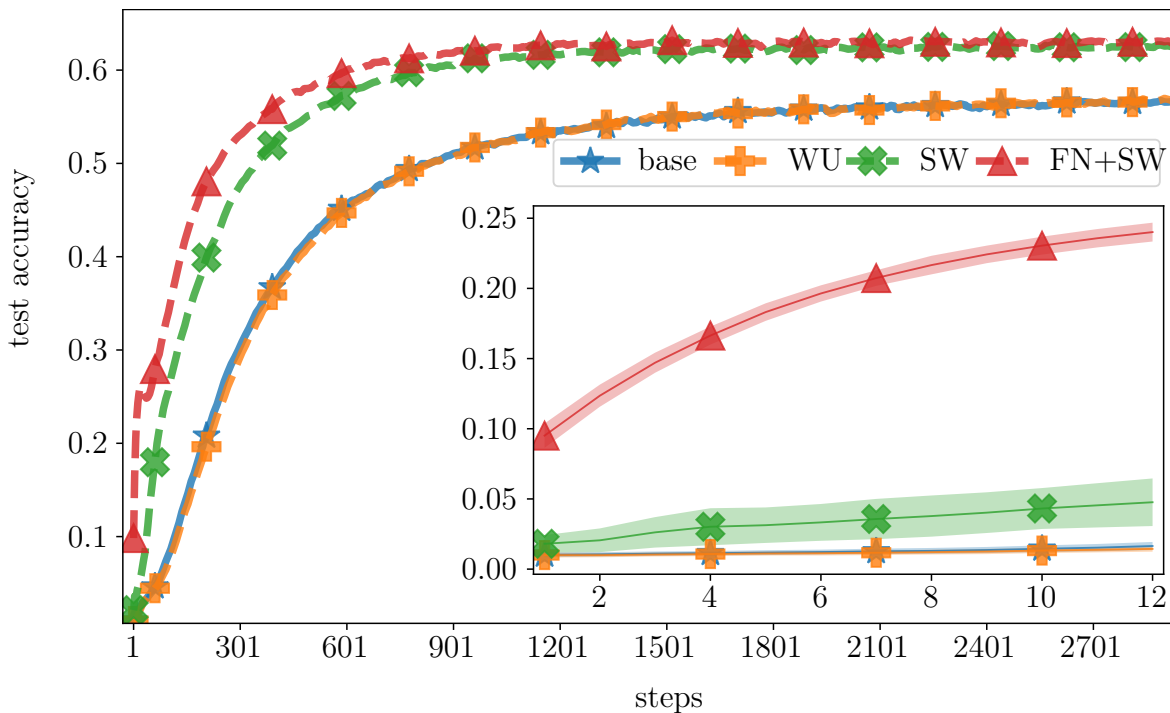


Figure B.5: Feature-tuning DenseNet-121, ImageNet \mapsto CIFAR-100.

B.2.2 domain adaptation Performance

In Section 5.6.4, we compared best performing baseline model without feature normalization (Figure 5.12) to our proposed methods with feature normalization (Figures

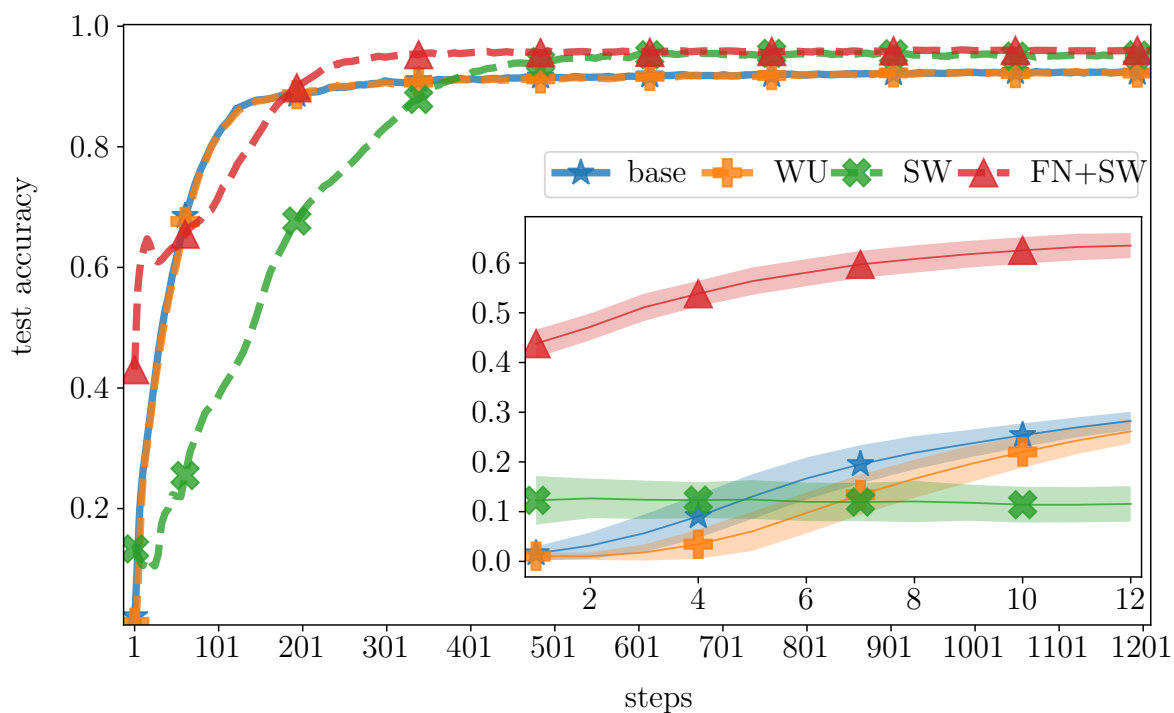


Figure B.6: Feature-tuning DenseNet-121, ImageNet \mapsto Caltech-101.

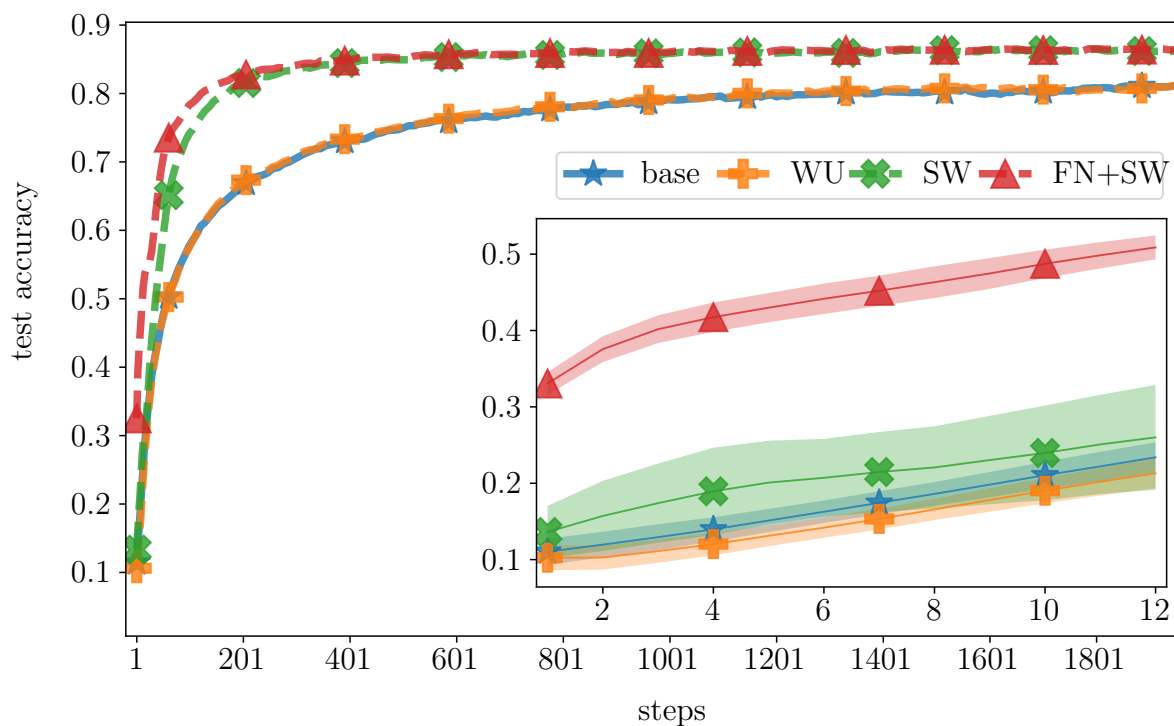


Figure B.7: Feature-tuning DenseNet-201, ImageNet \mapsto CIFAR-10.

5.14 and 5.15). In this section, we provide more results, including baselines and our methods with and without feature normalization for the same settings discussed in

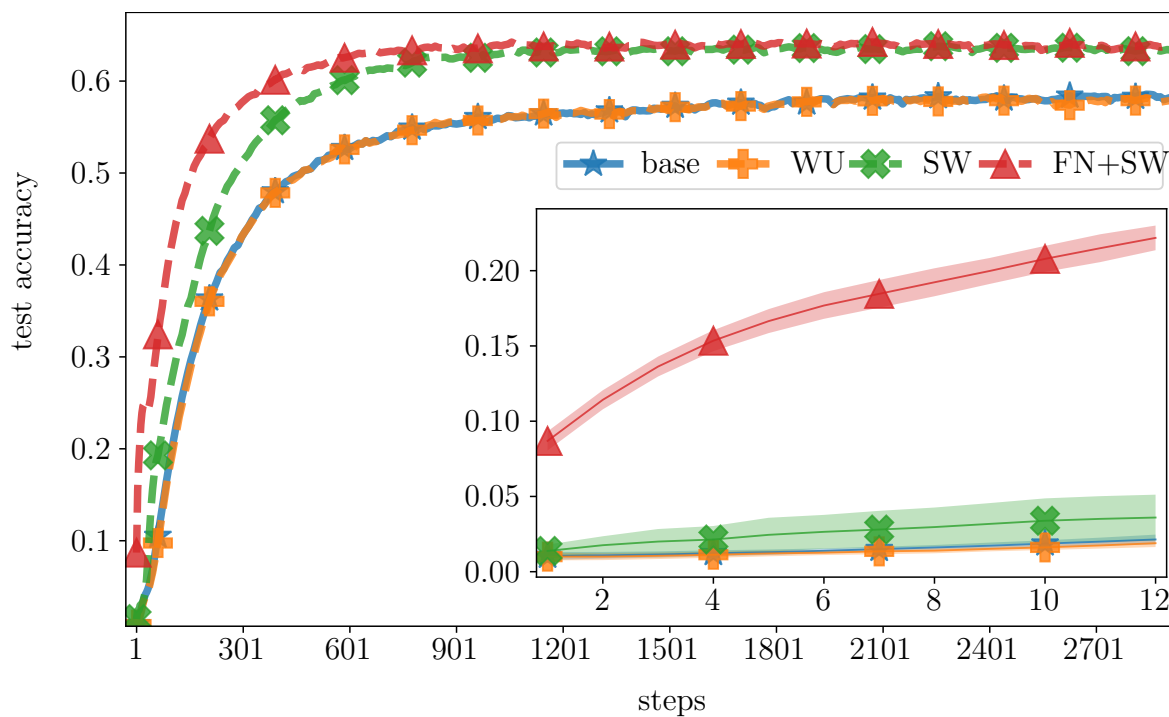


Figure B.8: Feature-tuning DenseNet-201, ImageNet \mapsto CIFAR-100.

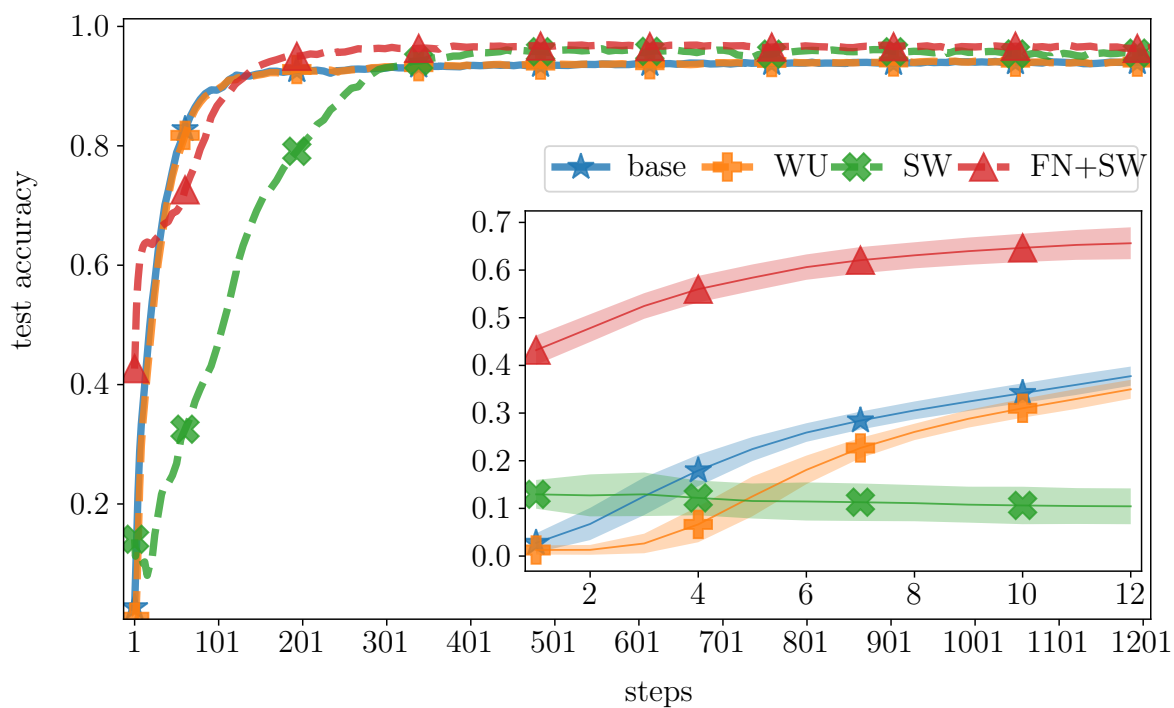
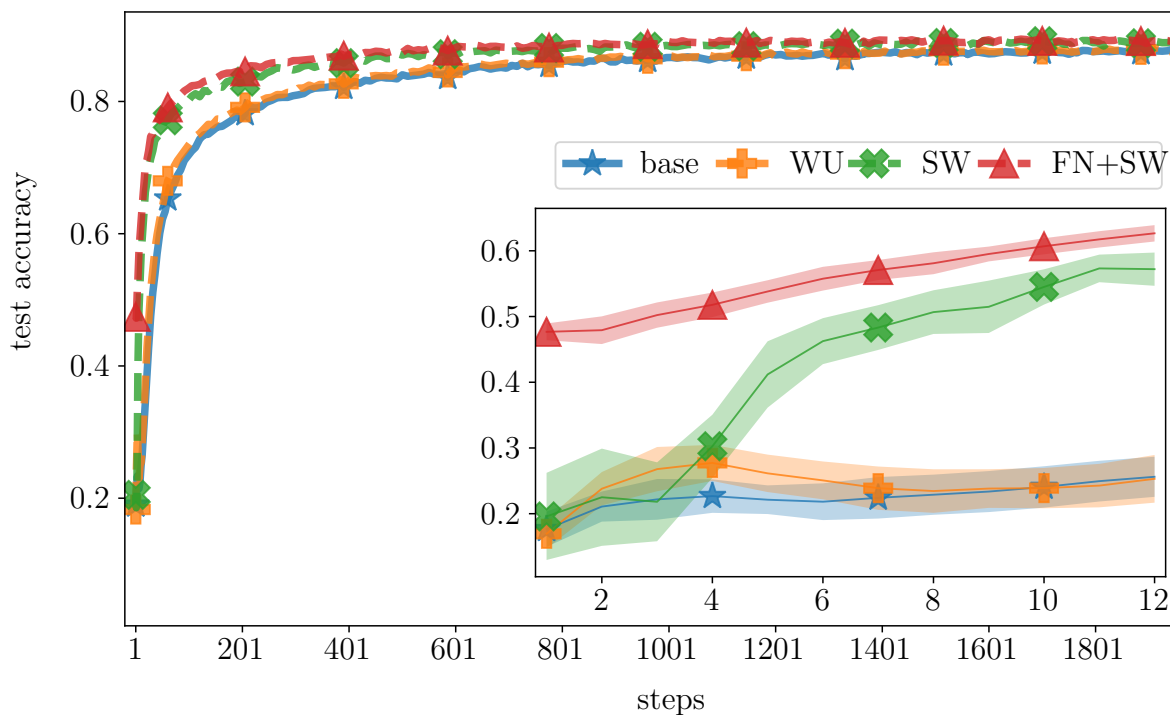
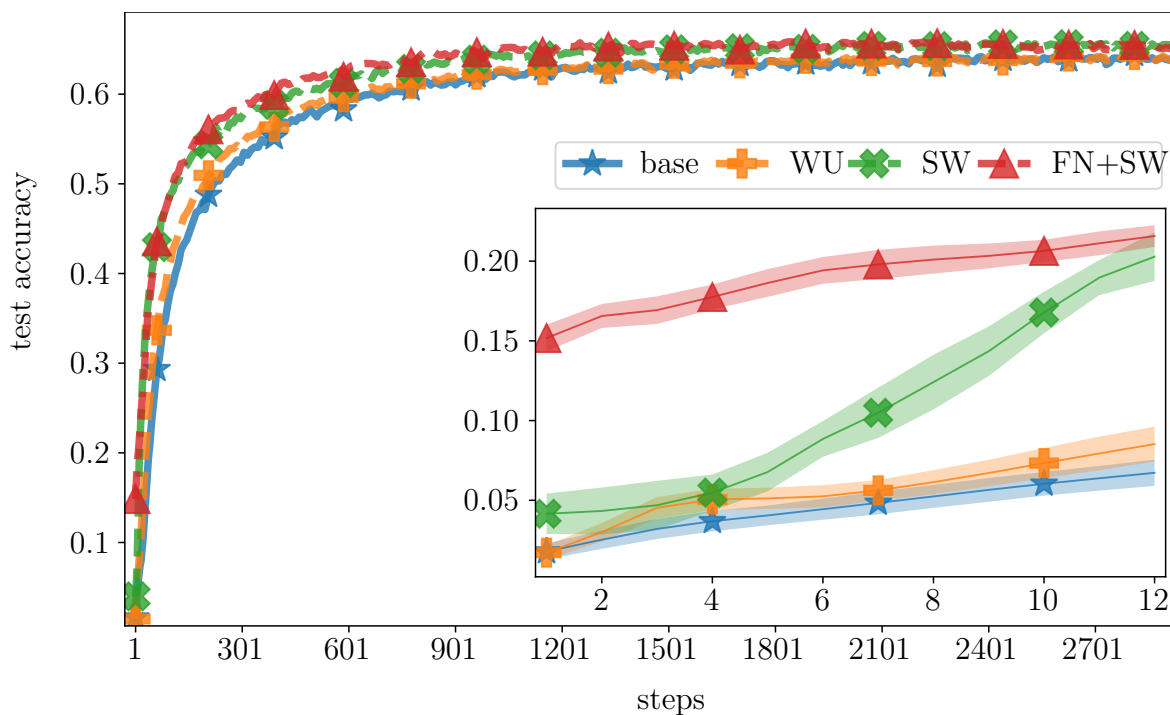
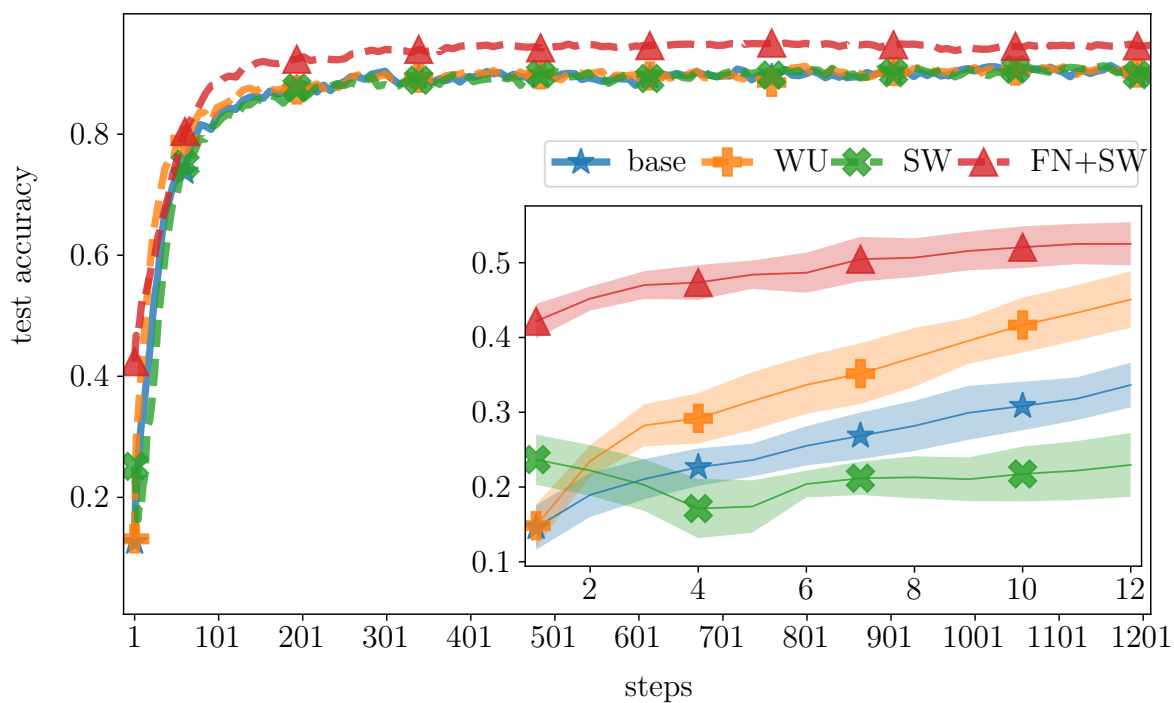
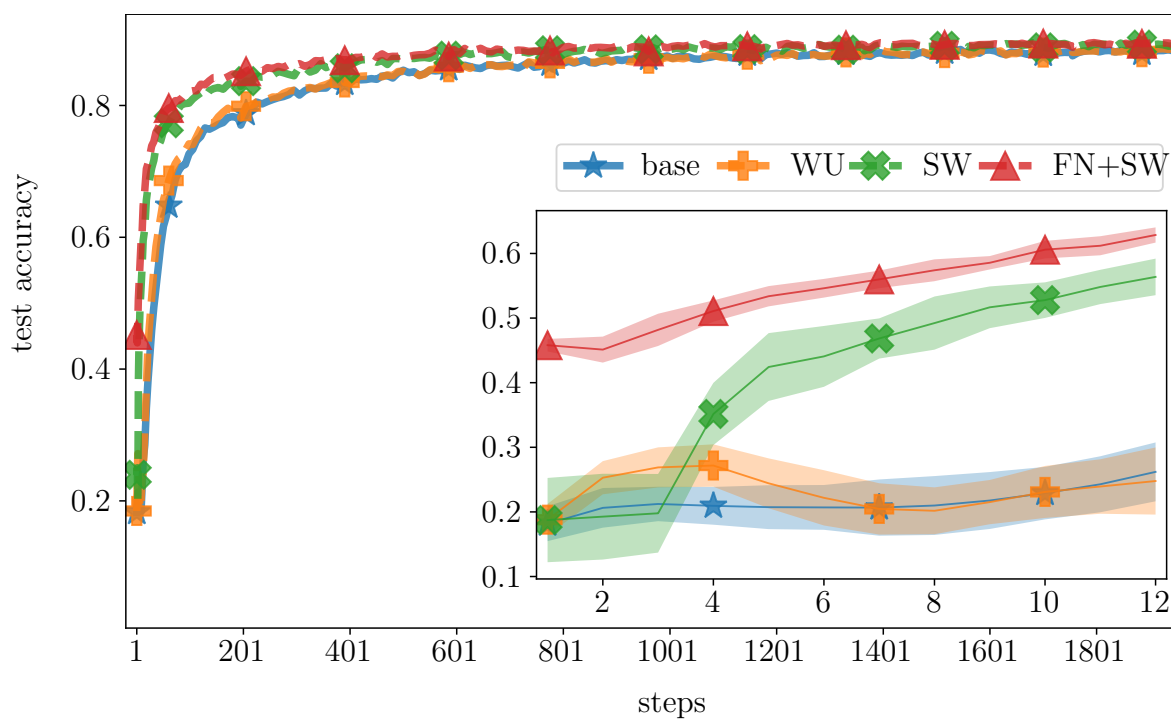


Figure B.9: Feature-tuning DenseNet-201, ImageNet \mapsto Caltech-101.

Section 5.6.4. These are presented in Figures B.19 to B.24.

Figure B.10: Feature-tuning VGG-16, ImageNet \mapsto CIFAR-10.Figure B.11: Feature-tuning VGG-16, ImageNet \mapsto CIFAR-100.

Figure B.12: Feature-tuning VGG-16, ImageNet \mapsto Caltech-101.Figure B.13: Feature-tuning VGG-19, ImageNet \mapsto CIFAR-10.

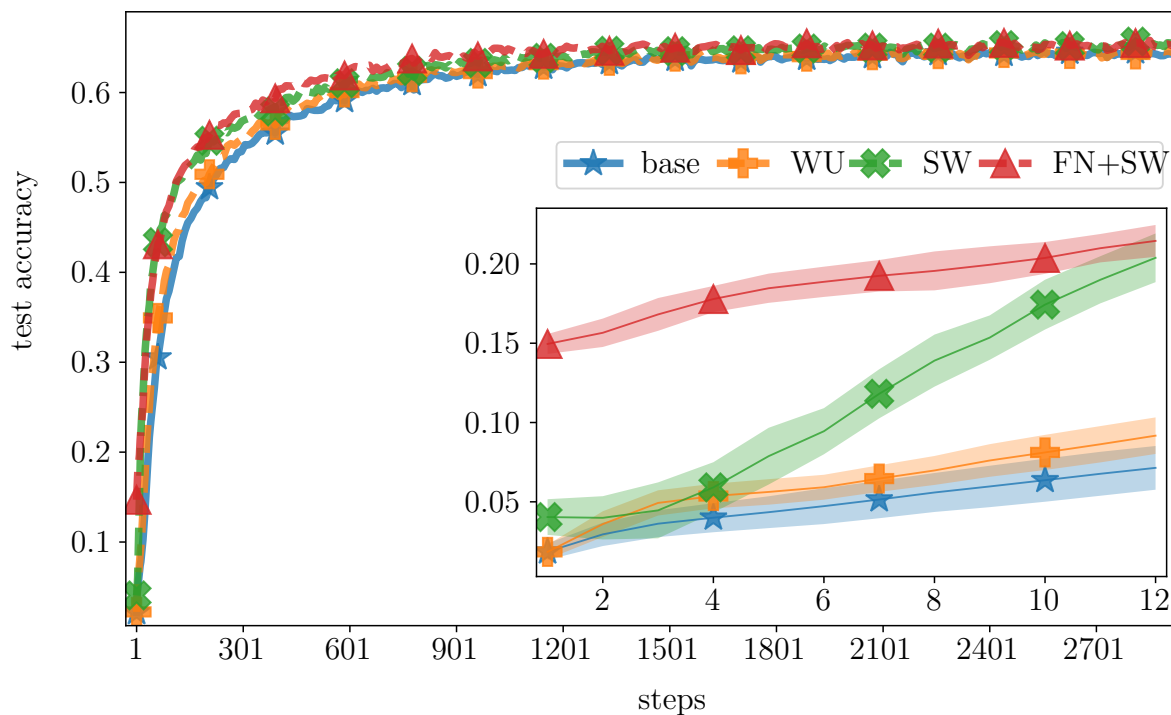


Figure B.14: Feature-tuning VGG-19, ImageNet \mapsto CIFAR-100.

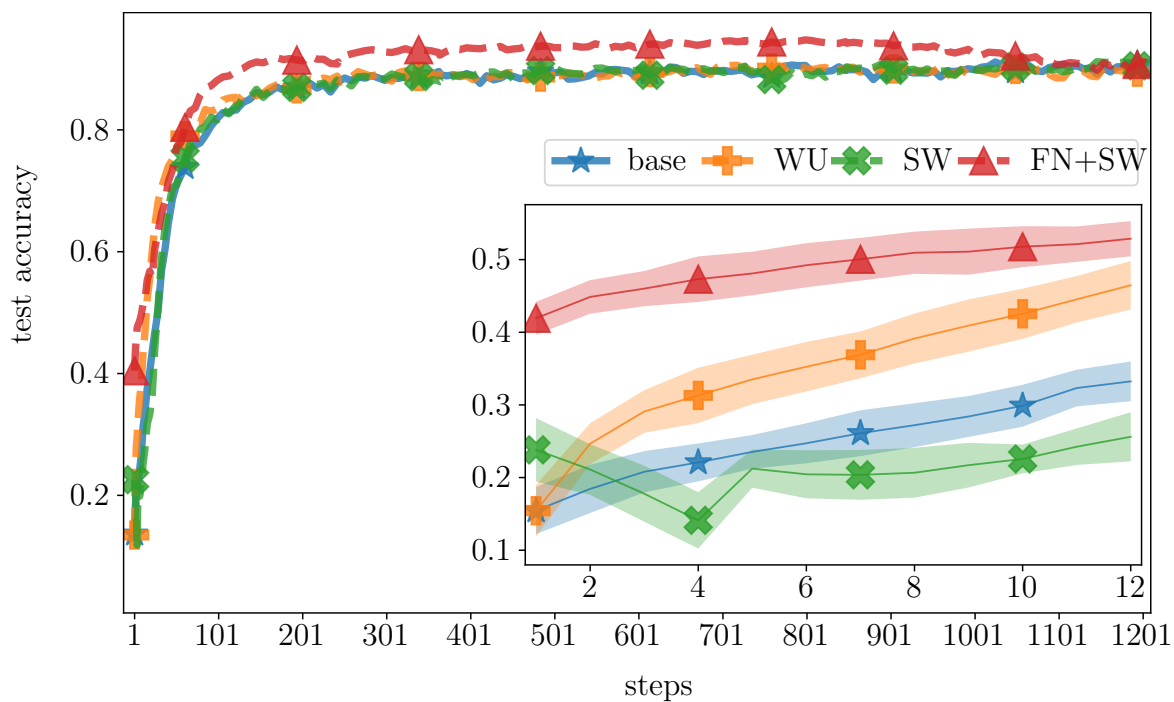


Figure B.15: Feature-tuning VGG-19, ImageNet \mapsto Caltech-101.

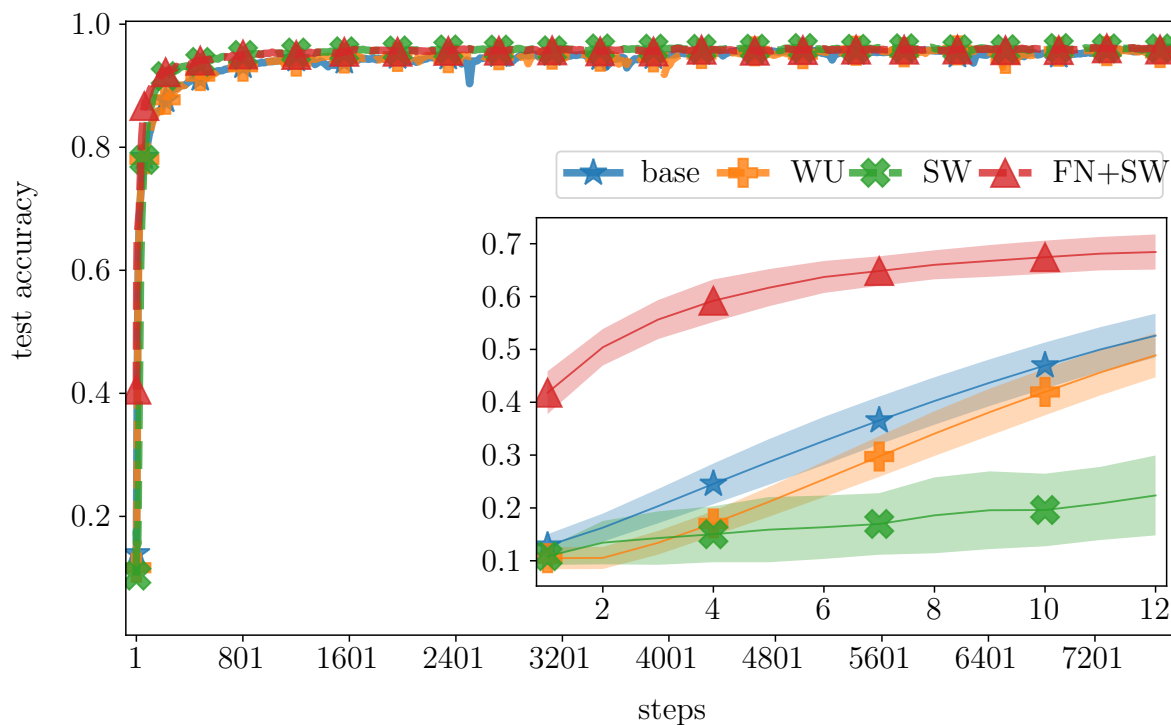


Figure B.16: Feature-tuning Inception-V3, ImageNet \mapsto CIFAR-10.

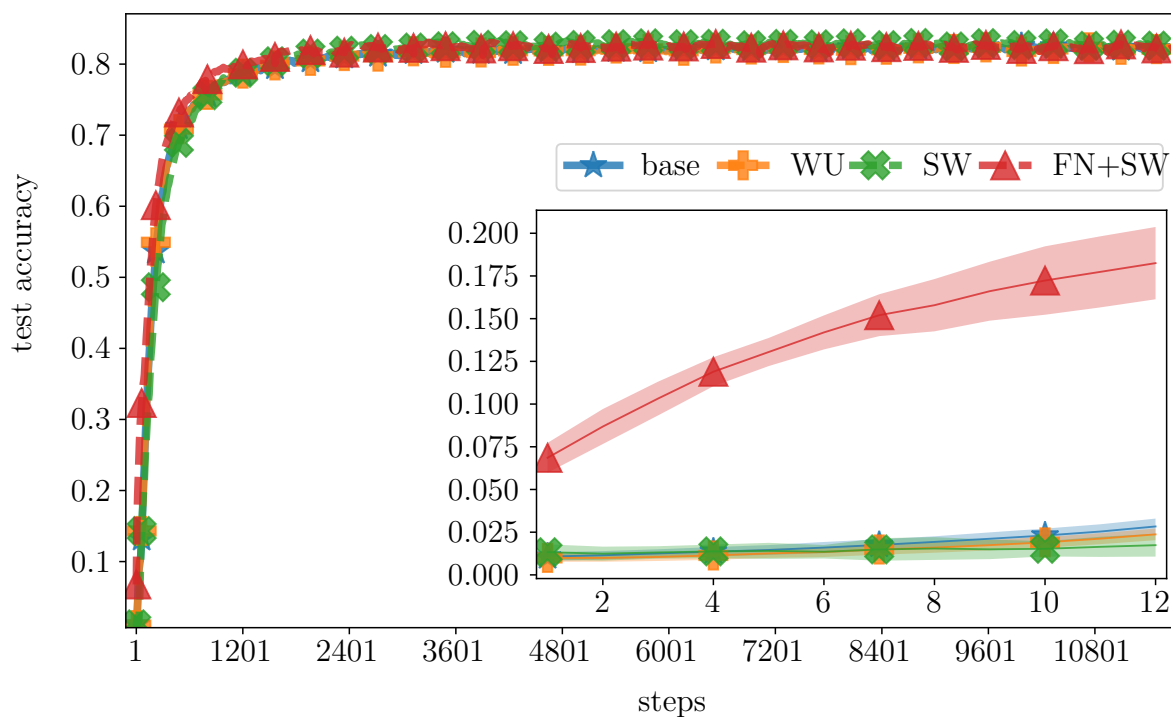


Figure B.17: Feature-tuning Inception-V3, ImageNet \mapsto CIFAR-100.

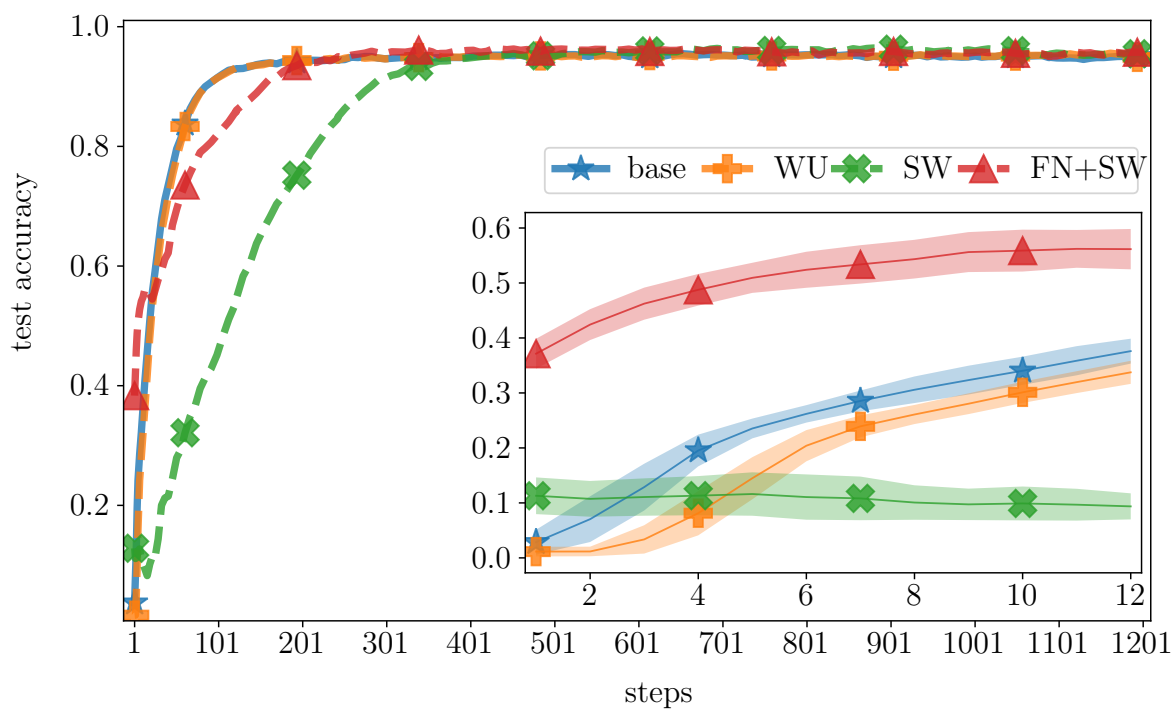


Figure B.18: Feature-tuning Inception-V3, ImageNet \mapsto Caltech-101.

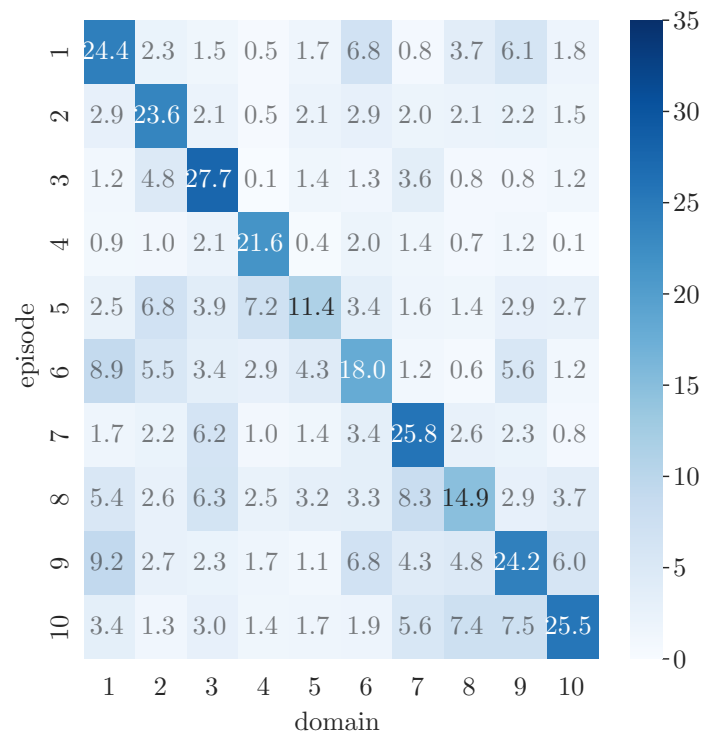


Figure B.19: Test accuracy measured for all domain per training episode using best configuration for Kin + FN ($\eta = 0.001$, features not scaled).

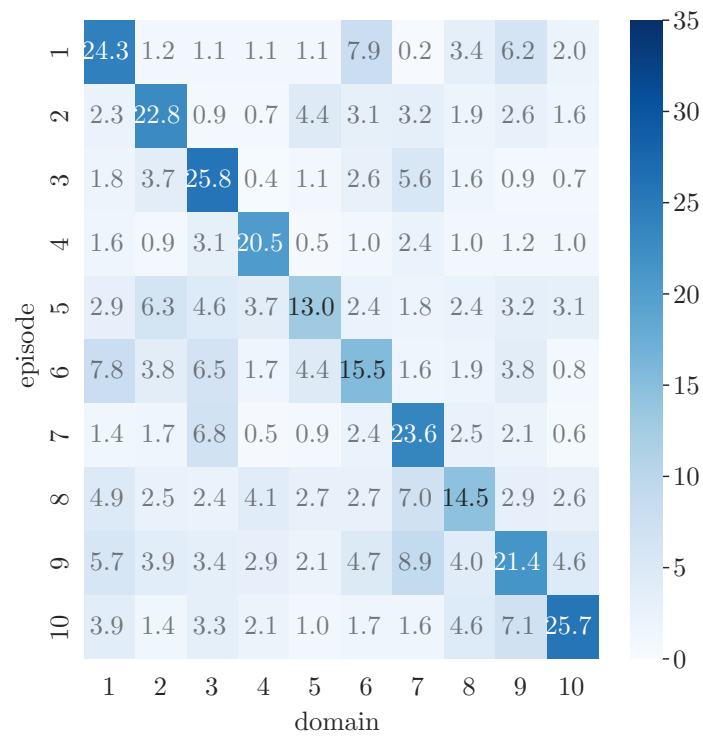


Figure B.20: Test accuracy measured for all domain per training episode using best configuration for Kout + FN ($\eta = 0.001$, features not scaled).

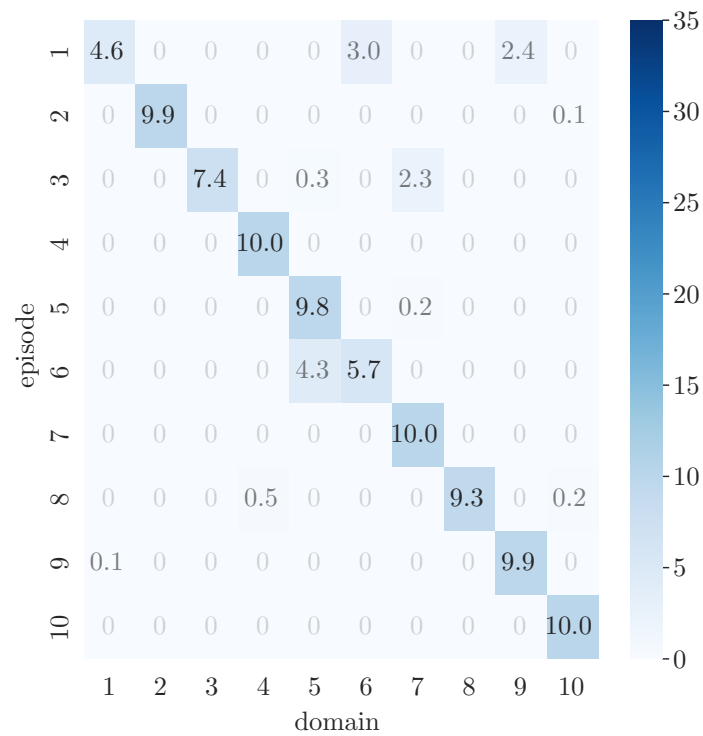


Figure B.21: Test accuracy measured for all domain per training episode using best configuration for SW ($\eta = 0.1$, features scaled by 1.0).

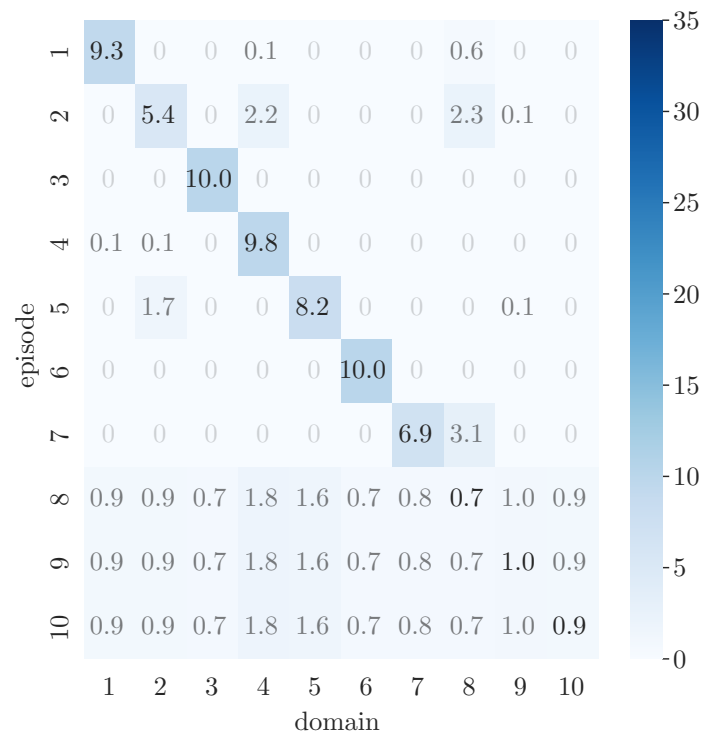


Figure B.22: Test accuracy measured for all domain per training episode using best configuration for SR ($\eta = 0.2$, features not scaled).

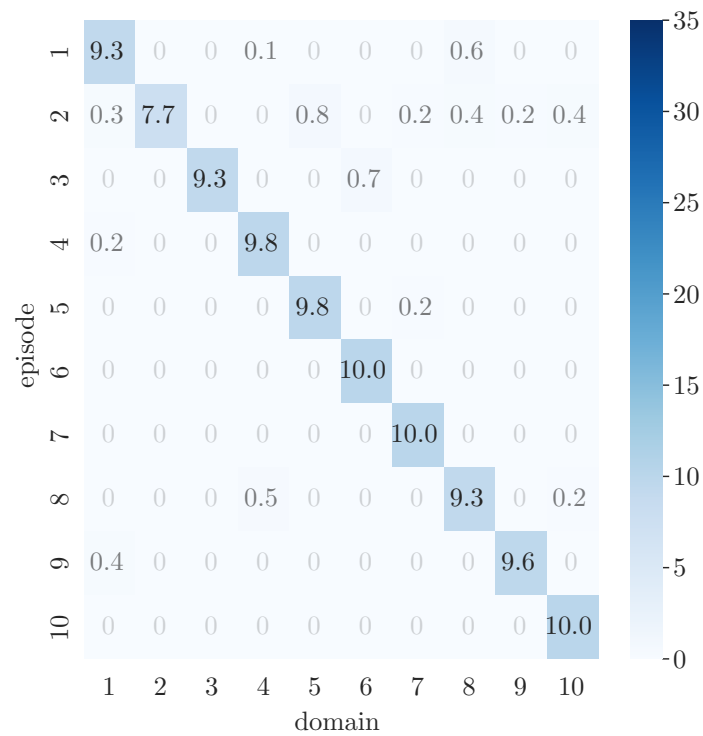


Figure B.23: Test accuracy measured for all domain per training episode using best configuration for Kin ($\eta = 0.2$, features scaled by 0.5).

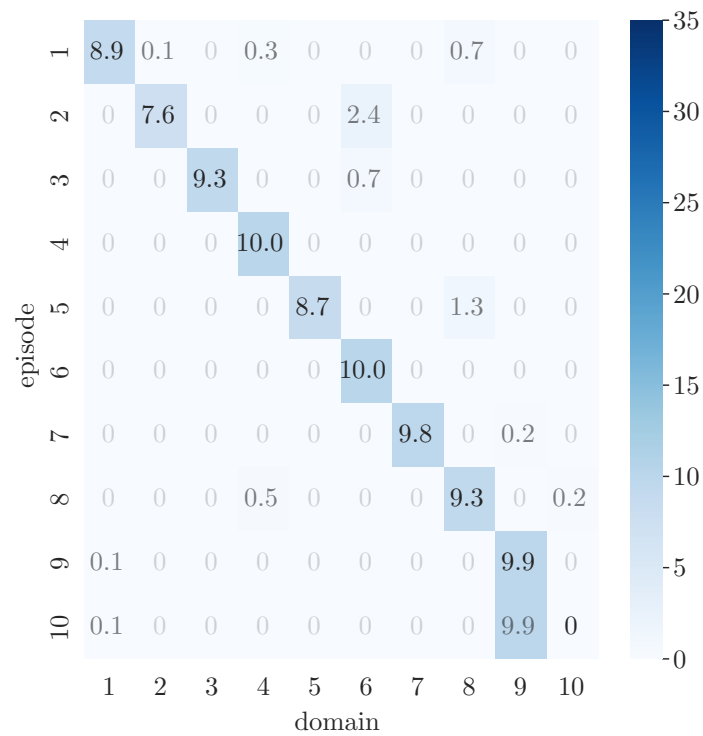


Figure B.24: Test accuracy measured for all domain per training episode using best configuration for Kout ($\eta = 0.2$, features scaled by 0.5).

Appendix C

Supplementary material for Chapter 6

C.1 Proofs

Theorem 12 *If GD or SGD is used to optimize the head parameters, the optimization velocity of the feature-extractor parameters is*

$$V(\phi^{t+1}) = \eta_\phi \eta_w \sum_{\tau=1}^{\tau=t-1} \mathbb{E}_{batch} [\delta^t \langle \mathbf{a}^\tau, \delta^\tau \rangle \frac{\partial \mathbf{a}^t}{\partial \phi^t}] - \eta_\phi \mathbb{E}_{batch} [\delta^t \mathbf{W}^0 \frac{\partial \mathbf{a}^t}{\partial \phi^t}] \quad (\text{C.1})$$

Proof *Given Equation (6.1), we have*

$$V(\phi^{t+1}) = -\eta_\phi \mathbb{E}_{batch} \left[\frac{\partial \ell^t}{\partial \mathbf{a}^t} \frac{\partial \mathbf{a}^t}{\partial \phi^t} \right] = -\eta_\phi \mathbb{E}_{batch} [\delta^t \mathbf{W}^t \frac{\partial \mathbf{a}^t}{\partial \phi^t}] \quad (\text{C.2})$$

Because GD or SGD is used

$$\mathbf{W}^t = \mathbf{W}^0 - \eta_w \sum_{\tau=1}^{\tau=t-1} \mathbb{E}_{batch} [\langle \mathbf{a}^\tau, \delta^\tau \rangle]; \quad (\text{C.3})$$

which gives

$$V(\phi^{t+1}) = -\eta_\phi \mathbb{E}_{batch} \left[\delta^t \left(\mathbf{W}^0 - \eta_w \sum_{\tau=1}^{\tau=t-1} \mathbb{E}_{batch} [\langle \mathbf{a}^\tau, \delta^\tau \rangle] \right) \frac{\partial \mathbf{a}^t}{\partial \phi^t} \right]. \quad (\text{C.4})$$

That is

$$V(\phi^{t+1}) = \eta_\phi \eta_w \mathbb{E}_{batch} \left[\delta^t \left(\sum_{\tau=1}^{\tau=t-1} \mathbb{E}_{batch} [\langle \mathbf{a}^\tau, \delta^\tau \rangle] \right) \frac{\partial \mathbf{a}^t}{\partial \phi^t} \right] - \eta_\phi \mathbb{E}_{batch} [\delta^t \mathbf{W}^0 \frac{\partial \mathbf{a}^t}{\partial \phi^t}]. \quad (\text{C.5})$$

Finally,

$$V(\phi^{t+1}) = \eta_\phi \eta_w \sum_{\tau=1}^{\tau=t-1} \mathbb{E}_{batch} [\delta^t \mathbb{E}_{batch} [\langle \mathbf{a}^\tau, \delta^\tau \rangle] \frac{\partial \mathbf{a}^t}{\partial \phi^t}] - \eta_\phi \mathbb{E}_{batch} [\delta^t \mathbf{W}^0 \frac{\partial \mathbf{a}^t}{\partial \phi^t}]. \quad (\text{C.6})$$

■

Appendix D

Supplementary material for Chapter 7

D.1 Proofs

Remark 2 *Aggregating client models by averaging is equivalent to applying a gradient step of size 1 from the previous round's cloud model using average of client pseudo-gradients or mathematically it is $\bar{\theta}^t \leftarrow \frac{1}{|S^t|} \sum_{i \in S^t} \theta_i^t = \theta^{t-1} - \bar{g}^t$.*

Proof

$$\begin{aligned} \bar{\theta}^t &\leftarrow \frac{1}{|S^t|} \sum_{i \in S^t} \theta_i^t = \theta^{t-1} - \frac{1}{|S^t|} \sum_{i \in S^t} \theta^{t-1} - \theta_i^t \\ &= \theta^{t-1} - \frac{1}{|S^t|} \sum_{i \in S^t} g_i^t \\ &= \theta^{t-1} - \bar{g}^t . \end{aligned}$$

■

Theorem 13 *In FedDyn, $\|\mathbf{h}^t\|^2 \leq \|\mathbf{h}^{t-1}\|^2$ requires*

$$\cos(\angle(\mathbf{h}^{t-1}, \bar{\mathbf{g}}^t)) \leq -\frac{|\mathcal{P}^t|}{2|S^t|} \frac{\|\bar{\mathbf{g}}^t\|}{\|\mathbf{h}^{t-1}\|}$$

Proof According to algorithm 3,

$$\mathbf{h}^t \leftarrow \mathbf{h}^{t-1} + \frac{|\mathcal{P}^t|}{|S^t|} \bar{\mathbf{g}}^t.$$

Applying 2-norm squared on both sides gives

$$\|\mathbf{h}^t\|^2 = \|\mathbf{h}^{t-1}\|^2 + \left(\frac{|\mathcal{P}^t|}{|S^t|}\right)^2 \|\bar{\mathbf{g}}^t\|^2 + 2\frac{|\mathcal{P}^t|}{|S^t|} \langle \mathbf{h}^{t-1}, \bar{\mathbf{g}}^t \rangle.$$

Considering the proposition, we have

$$\therefore \|\mathbf{h}^t\|^2 \leq \|\mathbf{h}^{t-1}\|^2 \implies 2\langle \mathbf{h}^{t-1}, \bar{\mathbf{g}}^t \rangle \leq -\frac{|\mathcal{P}^t|}{|S^t|} \|\bar{\mathbf{g}}^t\|^2.$$

with dividing both sides on some positive values, we get

$$\frac{\langle \mathbf{h}^{t-1}, \bar{\mathbf{g}}^t \rangle}{\|\bar{\mathbf{g}}^t\| \|\mathbf{h}^{t-1}\|} \leq -\frac{|\mathcal{P}^t|}{2|\mathcal{S}^t|} \frac{\|\bar{\mathbf{g}}^t\|}{\|\mathbf{h}^{t-1}\|},$$

which is equivalent to the

$$\cos(\angle(\mathbf{h}^{t-1}, \bar{\mathbf{g}}^t)) \leq -\frac{|\mathcal{P}^t|}{2|\mathcal{S}^t|} \frac{\|\bar{\mathbf{g}}^t\|}{\|\mathbf{h}^{t-1}\|}.$$

■

Remark 3 $\bar{\boldsymbol{\theta}}^{t-1} - \bar{\boldsymbol{\theta}}^t$ is equivalent to $\mathbf{h}^{t-1} + \bar{\mathbf{g}}^t$ in *AdaBest*.

We first add and remove $\boldsymbol{\theta}^{t-1}$ from the first side of the equation,

$$\begin{aligned} \bar{\boldsymbol{\theta}}^{t-1} - \bar{\boldsymbol{\theta}}^t &= \bar{\boldsymbol{\theta}}^{t-1} - \bar{\boldsymbol{\theta}}^t + \boldsymbol{\theta}^{t-1} - \boldsymbol{\theta}^{t-1} \\ &= (\bar{\boldsymbol{\theta}}^{t-1} - \boldsymbol{\theta}^{t-1}) + (\boldsymbol{\theta}^{t-1} - \bar{\boldsymbol{\theta}}^t). \end{aligned}$$

Then, we replace some terms using equation (7.1) and Remark 1 to get

$$\bar{\boldsymbol{\theta}}^{t-1} - \bar{\boldsymbol{\theta}}^t = \mathbf{h}^{t-1} + \bar{\mathbf{g}}^t.$$

Remark 4 *Cloud pseudo-gradients of AdaBest form a power series of $\mathbf{h}^t = \sum_{\tau=0}^t \beta^{(t-\tau+1)} \bar{\mathbf{g}}^\tau$, given that superscript in parenthesis means power.*

Proof We make the induction hypothesis $\mathbf{h}^{t-1} = \sum_{\tau=0}^{t-1} \beta^{(t-\tau)} \bar{\mathbf{g}}^\tau$. We need to prove that $\mathbf{h}^t = \sum_{\tau=0}^t \beta^{(t-\tau+1)} \bar{\mathbf{g}}^\tau$. From algorithm 3 we have

$$\mathbf{h}^t = \beta(\bar{\boldsymbol{\theta}}^{t-1} - \bar{\boldsymbol{\theta}}^t).$$

Additionally, using Remark 3 it could be rewritten as

$$\mathbf{h}^t = \beta(\mathbf{h}^{t-1} + \beta \bar{\mathbf{g}}^t).$$

Replacing the induction hypothesis changes it to

$$\begin{aligned} \mathbf{h}^t &= \beta \left(\sum_{\tau=0}^{t-1} \beta^{(t-\tau)} \bar{\mathbf{g}}^\tau + \beta \bar{\mathbf{g}}^t \right) \\ &= \sum_{\tau=0}^{t-1} \beta^{(t-\tau)} \bar{\mathbf{g}}^{\tau+1} + \beta^{(2)} \bar{\mathbf{g}}^t \\ &= \sum_{\tau=0}^t \beta^{(t-\tau+1)} \bar{\mathbf{g}}^\tau. \end{aligned}$$

■

Remark 5 *FedAvg* is a special case of *AdaBest* where $\beta = \mu = 0$.

Proof In *AdaBest*, $\mu = 0$ makes \mathbf{h}_i^t zero for all feasible i and t . The resulting local update is identical to that of *FedAvg*. Similarly, \mathbf{h}^t becomes zero at all rounds if $\beta = 0$. So *AdaBest* would also have the same server updates as of *FedAvg*. ■

Remark 6 Server update of *FedDyn* is a special case of *AdaBest* where $\beta = 1$ except that an extra $\frac{|P|}{|S|}$ scalar is applied which also adversely makes *FedDyn* require prior knowledge about the number of clients.

Proof According to algorithm 3, on the server side *AdaBest* and *FedDyn* are different in their update of \mathbf{h}^t . Based on Remark 3, for $\beta = 1$ *AdaBest* update is $\mathbf{h}^t \leftarrow \mathbf{h}^{t-1} + \bar{\mathbf{g}}^t$. Comparably the same update in *FedDyn* is $\mathbf{h}^t \leftarrow \mathbf{h}^{t-1} + \frac{|P|}{|S^t|} \bar{\mathbf{g}}^t$. Involving $\|S^t\|$ in the update means assuming that prior knowledge on number of total clients is available from the beginning of the training. On the other hand, $\beta = 0$ leads to $\mathbf{h}^t = 0$ and consequently the update on the cloud model become $\boldsymbol{\theta}^t \leftarrow \bar{\boldsymbol{\theta}}^t$ which is identical to the server update of *FedAvg*. ■

Theorem 14 If S be a fixed set of clients, $\bar{\boldsymbol{\theta}}$ does not converge to a stationary point unless $\mathbf{h} \rightarrow \mathbf{0}$.

Proof With a minor abuse in our notation for the case of SCAFFOLD/m (the difference only is applying \mathbf{h} on the clients after $\boldsymbol{\theta}$ is sent to them), we can generally state that

$$\bar{\boldsymbol{\theta}}^t \leftarrow \boldsymbol{\theta}^{t-1} - \bar{\mathbf{g}}^t = \bar{\boldsymbol{\theta}}^{t-1} - (\mathbf{h}^{t-1} + \bar{\mathbf{g}}^t).$$

With S being fixed, upon $t \rightarrow \infty$, and convergence of $\bar{\boldsymbol{\theta}}$, we expect that $\mathbf{g}^t \rightarrow \mathbf{0}$, so the optimization does not step out of the minima. In that case, we also expect $\bar{\boldsymbol{\theta}}^t \approx \bar{\boldsymbol{\theta}}^{t-1}$. On the other hand, above formula results in $\bar{\boldsymbol{\theta}}^t \approx \bar{\boldsymbol{\theta}}^{t-1} - \mathbf{h}^{t-1}$ which holds if $\mathbf{h} \rightarrow \mathbf{0}$. ■

D.2 Algorithm Details

D.2.1 Notation

Following [Karimireddy et al. \[2020\]](#) and [Acar et al. \[2020\]](#), for the sake of simplicity and readability, we only presented algorithm 3 for the balanced case (in terms of number of data samples per client). According to [Acar \[2021\]](#), [SCAFFOLD](#) and [FedDyn](#) also need the prior knowledge about the number of clients in order to properly weight their \mathbf{h}^t accumulation in the case of unbalance data samples. These weights are used in the form of average samples per client in [Acar \[2021\]](#). We eliminate such dependency in our algorithm by progressively calculating this average throughout the training. We also confirm that applying the same modification on [SCAFFOLD](#) and [FedDyn](#) does not impede their performance nor their stability (experiments on [FedDyn](#) and [SCAFFOLD](#) still are done with their original form). For the experiments, we implemented [SCAFFOLD](#) as introduced in the original paper [Karimireddy et al. \[2020\]](#); However, for more clarity a modification of it ([SCAFFOLD/m](#)) is contrasted to other methods in algorithm 3 in which only the model parameters are sent back to the server (compare it to Algorithm 1 in [Karimireddy et al. \[2020\]](#)). Note that this difference in presentation is irrelevant to our arguments in this paper since the more important factor for scalability in the recursion is $\frac{|S^t| - |\mathcal{P}^t|}{|S^t|}$.

D.2.2 Algorithmic Costs

In this section, we compare compute, storage and bandwidth costs of [AdaBest](#) to that of [FedAvg](#), [SCAFFOLD/m](#) and [FedDyn](#).

Compute Cost

Table D.1 shows the notation we use for formulating the *costs* of these algorithms. Algorithm 4 is an exact repetition of Algorithm 3 except that the compute cost of operations of interest are included as comments. These costs are summed in tables Table D.2 and Table D.3 for the client and server sides, respectively. According to these tables, [AdaBest](#) has lower client-side and server-side compute costs than [SCAFFOLD/m](#) and [FedDyn](#).

Table D.1: Summary of notion used to formulate the algorithm costs

Notation	Meaning
n	Number of parameters of the model $:= \boldsymbol{\theta} $
g	Cost of computing local mini-batch gradients
s	Cost of summing two floating point numbers
m	Cost of multiplying two floating point numbers

Table D.2: Comparing [AdaBest](#) to [FedAvg](#), [SCAFFOLD/m](#), [FedDyn](#) in their compute cost of local (client side) operations. See Algorithm 4 for more detailed comparison

Algorithm	Client side compute cost
FedAvg	$K(g + ns + nm)$
SCAFFOLD/m	$K(g + ns + nm) + 2Kns + 2n(s + m)$
FedDyn	$K(g + ns + nm) + 3Kns + Knm + n(s + m) + ns$
AdaBest	$K(g + ns + nm) + Kns + n(s + m) + ns$

Table D.3: Comparing [AdaBest](#) to [FedAvg](#), [SCAFFOLD/m](#), [FedDyn](#) in their compute cost of global (server side) operations. See Algorithm 4 for more detailed comparison

Algorithm	Server side compute cost
FedAvg	$ \mathcal{P}^t ns$
SCAFFOLD/m	$ \mathcal{P}^t ns + 2ns + 2nm$
FedDyn	$ \mathcal{P}^t ns + 3ns + nm$
AdaBest	$ \mathcal{P}^t ns + 2ns + nm$

Storage Cost

All the three algorithms require the same amount of storage, on the clients and the server. Each client is supposed to store a set of local gradient estimates with size n (see Table D.1), noted as \mathbf{h}_i^t . Likewise, each algorithm stores the same number of variables on the server so that estimates are forwarded to the next rounds. These variables are introduced with \mathbf{h}^t in [SCAFFOLD/m](#) and [FedDyn](#) but with $\bar{\boldsymbol{\theta}}^{t-1}$ in [AdaBest](#).

Communication Cost

[AdaBest](#) and [FedDyn](#) are not different in the way information is communicated between the server and clients; thus, they do not differ in terms of costs of communication bandwidth. That is sending n parameters from server to each selected client at each round and receiving the same amount in the other direction (from each client to the server). The original [SCAFFOLD](#) needs doubling the amount of information communicated in each direction ($2n$). However, [SCAFFOLD/m](#) reduces this overhead to 1.5 times (of that of [AdaBest](#)) by avoiding to send the extra variables from clients' to the server ($1.5 n$).

More accurate costs requires exact specifications of the system design. For example, the aggregation operation on the server if done in a batch (from a buffer of client delivered parameters) requires more storage but can decrease the compute cost using multi-operand adders on floating-point mantissas such as *Wallace* or *Dadda* tree adders. However, these design choices do not appear to make a difference in the ranking of the costs for algorithms compared in this paper.

For cost estimates to be more precise, system design specifications must be considered. For instance, using multi-operand adders on floating-point mantissas, such as *Wallace* or *Dadda* tree adders, can reduce the compute cost of the aggregation operation on the server if it is performed in a batch (from a buffer of client-delivered parameters) but requires more storage. However, it does not appear that these design decisions affect the ranking of the costs for the algorithms compared in this paper.

Algorithm 4 Compute cost of **SCAFFOLD/m**, **FedDyn**, **AdaBest**. Operations that are common among all three algorithms are grayed out. The compute cost of other operations are shown with a comment in front of each line. The variables used to represent the cost of each micro-operation are introduced in Table D.1

Input: T, θ^0, μ, β

for $t = 1$ **to** T **do**

Sample clients $\mathcal{P}^t \subseteq S^t$.

Transmit θ^{t-1} to each client in \mathcal{P}^t

Transmit h^{t-1} to each client in \mathcal{P}^t (**SCAFFOLD/m**)

for each client $i \in \mathcal{P}^t$ **in parallel do**

$\theta_i^{t,0} \leftarrow \theta^{t-1}$

for $k = 1$ **to** K **do**

Compute mini-batch gradients $\mathbf{g}_i^{t,k-1} = \mathbb{E}_{\text{batch}} [\nabla \ell_i(\theta_i^{t,\tau-1})]$ /* g^* */

$\mathcal{Q} \leftarrow \mathbf{g}_i^{t,k-1} - \mathbf{h}_i^{t'} + \mathbf{h}^t$ (**SCAFFOLD/m**) /* $2ns^*$ */

$\mathcal{Q} \leftarrow \mathbf{g}_i^{t,k-1} - \mathbf{h}_i^{t'} - \mu(\theta^{t-1} - \theta_i^{t,k-1})$ (**FedDyn**) /* $3ns + nm^*$ */

$\mathcal{Q} \leftarrow \mathbf{g}_i^{t,k-1} - \mathbf{h}_i^{t'}$ (**AdaBest**) /* ns^* */

$\theta_i^{t,k} \leftarrow \theta_i^{t,k-1} - \eta \mathcal{Q}$ /* $ns + nm^*$ */

end for

$\mathbf{g}_i^t \leftarrow \theta^{t-1} - \theta_i^{t,K}$ /* ns^* */

$\mathbf{h}_i^t \leftarrow \mathbf{h}_i^{t'} - \mathbf{h}^{t-1} + \frac{1}{K\eta} \mathbf{g}_i^t$ (**SCAFFOLD/m**) /* $2ns + 2nm^*$ */

$\mathbf{h}_i^t \leftarrow \mathbf{h}_i^{t'} + \mu \mathbf{g}_i^t$ (**FedDyn**) $\mathbf{h}_i^t \leftarrow \frac{1}{t-t'} \mathbf{h}_i^{t'} + \mu \mathbf{g}_i^t$ (**AdaBest**) /* $ns + nm^*$ */

$t'_i \leftarrow t$

Transmit client model $\theta_i^t := \theta_i^{t,K}$.

end for

$\bar{\theta}^t \leftarrow \frac{1}{|\mathcal{P}^t|} \sum_{i \in \mathcal{P}^t} \theta_i^t$ /* $|\mathcal{P}^t|ns^*$ */

$\mathbf{h}^t \leftarrow \frac{|S^t| - |\mathcal{P}^t|}{|S^t|} \mathbf{h}^{t-1} + \frac{|\mathcal{P}^t|}{K\eta|S^t|} (\theta^{t-1} - \bar{\theta}^t)$ (**SCAFFOLD/m**) /* $2ns + 2nm^*$ */

$\mathbf{h}^t \leftarrow \mathbf{h}^{t-1} + \frac{|\mathcal{P}^t|}{|S^t|} (\theta^{t-1} - \bar{\theta}^t)$ (**FedDyn**) /* $2ns + nm^*$ */

$\mathbf{h}^t \leftarrow \beta(\bar{\theta}^{t-1} - \bar{\theta}^t)$ (**AdaBest**) /* $ns + nm^*$ */

$\theta^t \leftarrow \bar{\theta}^t$ (**SCAFFOLD/m**)

$\theta^t \leftarrow \bar{\theta}^t - \mathbf{h}^t$ (**FedDyn**) $\theta^t \leftarrow \bar{\theta}^t - \mathbf{h}^t$ (**AdaBest**) /* ns^* */

end for

D.2.3 Experiments Details

Evaluation

There are two major differences between our evaluation and those of prior works.

1. **We do not consider number of epochs to be a hyper-parameter.** Comparing two FL algorithms with different number of local epochs, is unfair in terms of the amount of local compute costs. Additionally, it makes it difficult to justify the impact of each algorithm on preserving the privacy of clients' data. This is because the privacy cost is found to be associated with the level of random noise in the pseudo-gradients. This randomness in turn is impacted by the number of epochs (see page 8 of Fowl et al. [2021]). For an example of comparing algorithms after tuning the number of epochs, refer to Appendix 1 of Acar et al. [2020] where 20 and 50 local epochs are chosen respectively for FedAvg and FedDyn in order to compare their performance on MNIST dataset.
2. **We consider a hold-out set of clients for hyper-parameter tuning.** Although, an *on the fly hyper-parameter tuning* is much more appealing in FL setting, for the purpose of studying and comparing FL algorithms, it is reasonable to consider hyper-parameters are tuned prior to the main training phase. However, using the performance of the test dataset in order to search for hyper-parameters makes the generalization capability of the algorithms that use more hyper-parameters questionable. Therefore, we set aside a separate set of training clients to tune the hyper-parameters for each algorithm individually. This may make our reported results on the baselines not exactly matching that of their original papers (different size of total training samples).

In addition, we use five distinct random seeds for data partitioning to better justify our reported performance. Throughout all the experiments, SGD with a learning rate of 0.1 is used¹ with a round to round decay of 0.998. Batch size of 45 is selected for all datasets and experiments. Whenever the last batch of each epoch is less than this number, it is capped with bootstrapping from all local training examples (which can further enhances the privacy especially for imbalance settings). We follow Acar et al.

¹We followed Acar et al. [2020] in choosing this optimization algorithm and learning rate.

[2020] in data augmentation and transformation. Local optimization on [CIFAR-10](#) and [CIFAR-100](#) involves random horizontal flip and cropping on the training samples both with probability of 0.5. No data augmentation is applied for experiments on [EMNIST](#).

Implementation

We used `PYTORCH` to implement our [FL](#) simulator. For data partitioning and implementation of the baseline algorithms, our simulator is inspired from [Acar \[2021\]](#) which is publicly shared by the authors of [Acar et al. \[2020\]](#). To further validate the correctness of [SCAFFOLD](#) implementation, we consulted the first author of [Karimireddy et al. \[2020\]](#). Additionally, we cross-checked most of the results our simulator yielded to the ones made by [Acar \[2021\]](#). We have not publicly shared our simulator due to intellectual property considerations; however, we have provided an implementation of [AdaBest](#) in our other simulator which is open-sourced. This simulator is accessible at <https://fedsim.varnio.com>.

D.2.4 Stability and norm of parameters

In figure [7.3](#), we showed an experimental case of low client participation to demonstrate how $\|\theta^t\|$ is associated with instability of [FedDyn](#). In this experiment, the training split of [CIFAR-100](#) is partitioned over 1100 clients from which 1000 is used for training. The partitioning is balanced in terms of number of examples per client and the labels are skewed according to our $\alpha = 0.3$ heterogeneity setup (see Section [7.5.4](#) for detailed explanation). In each round, 5 clients are drawn uniformly at random. This low client participation rate is more likely to occur in a large-scale (in terms of number of clients) cross-device [FL](#) compared to the setting used for reporting the performances in Table [7.2](#) and most of those of our prior works. In figure [D.1](#) we repeat the same experiment; except that a much simpler [FL](#) task is defined. In this experiment the training split of [EMNIST](#) dataset is partitioned into 110 clients, 100 of which are used for training. Partitions are IID (labels are not skewed). Even though this task is much simpler than the previous one, still [FedDyn](#) fails to converge when the training continues for a large number of rounds.

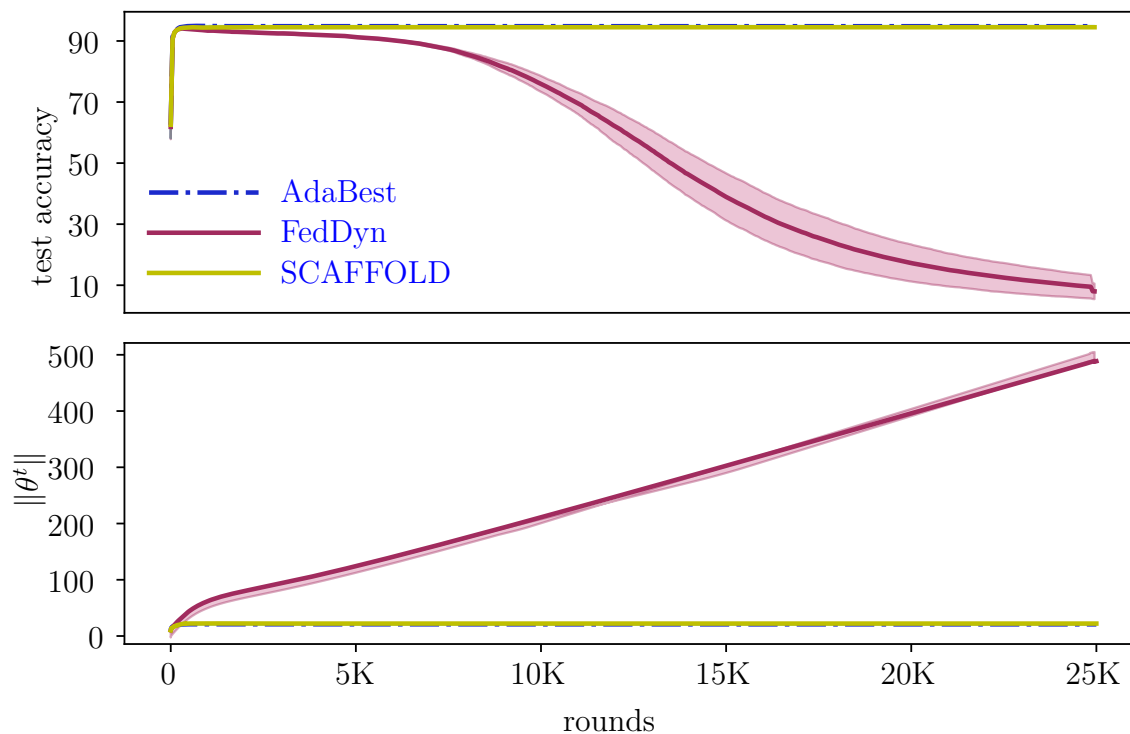


Figure D.1: Highlighting the instability of FedDyn and its association with the norm of cloud parameters. The training is performed on a comparably easy FL task but for a large number of communication rounds. Top and bottom subplots show test accuracy (in percentage) and norm of cloud parameters respectively. The horizontal axis which shows number of communication rounds is shared among subplot

Overfitting Analysis

D.2.5 Overfitting Analysis

It is important not to confuse FedDyn’s source of instability with overfitting. To confirm this, we can compare the average train and test accuracy of the same rounds while the model is being trained. figure D.2 compares SCAFFOLD, FedDyn, and AdaBest in this manner. The configuration used in this experiment is identical to the configuration of the experiment in figure 7.3, with the exception that it corresponds to a single data partitioning random seed for clarity. The train accuracy is calculated by averaging the train accuracy of participating clients. The results suggest that train accuracy of the baselines is severely declined as the training continues while AdaBest is much more stable in this regard.

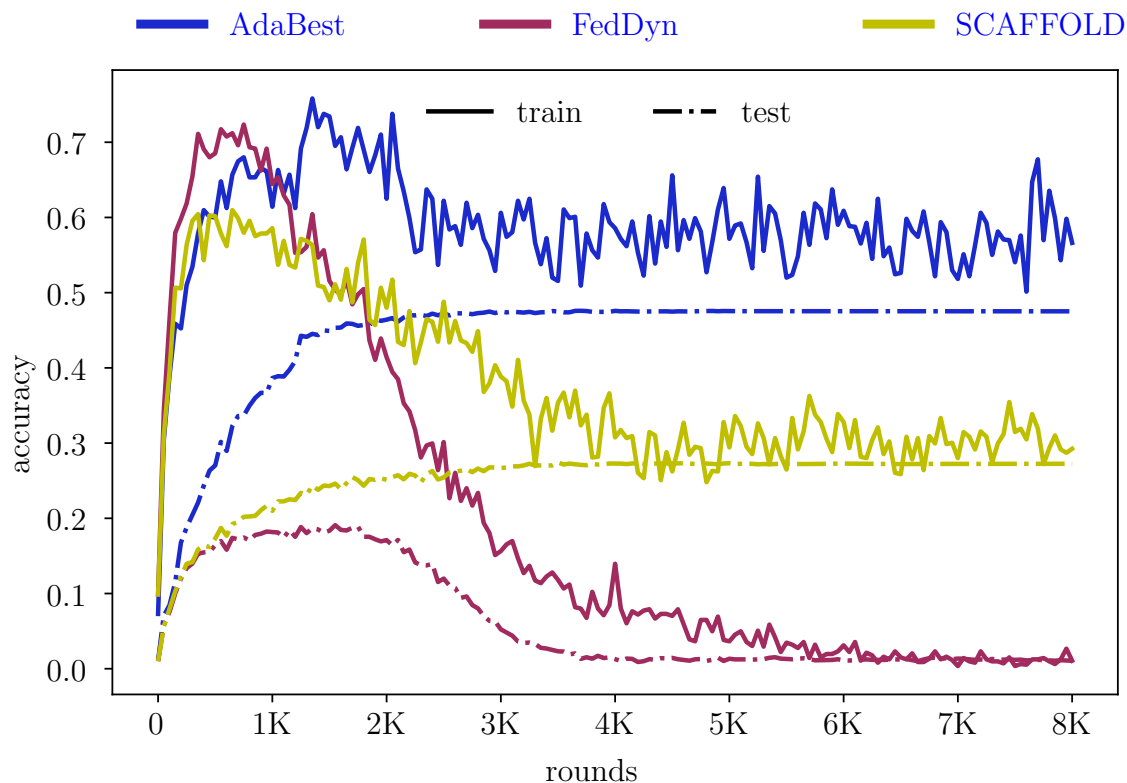


Figure D.2: Top-1 train and test accuracy score of [AdaBest](#) and the baselines. The solid and dash-dotted lines represent train and test, respectively

D.2.6 Local regularization sensitivity

During the hyper-parameter tuning phase, we only tune β of [AdaBest](#) and set its μ constantly to 0.02. This is done throughout all the experiments except the experiment presented in this section which especially investigates the sensitivity of [AdaBest](#) to varying μ . Therefore, in practice we have not treated μ as a hyper-parameter but rather chose the value that works best for [FedDyn](#). For this experiment, we use the same setup as the one presented in Section D.2.4 on [EMNIST](#) dataset. The only difference is that we vary μ in the range of $\{0.02 \times 2^{(k)}\}_{k=1}^{k=3}$. figure D.3 depicts the outcome of this experiment along with the result of the same analysis on [FedDyn](#) so it would be easy to compare the sensitivity of each of these algorithms to their local factor μ . As suggested by the top left subplot in this Figure, [AdaBest](#) can even achieve higher numbers in terms of the test accuracy than what is reported in Table 7.2. The scales on the vertical axis of the subplots related to [AdaBest](#) (on the

left column) are zoomed-in to better show the stability of our algorithm throughout the training. On the other hand, the performance and stability of FedDyn shows to be heavily relied on the choice of μ when the training continues for a large number of rounds.

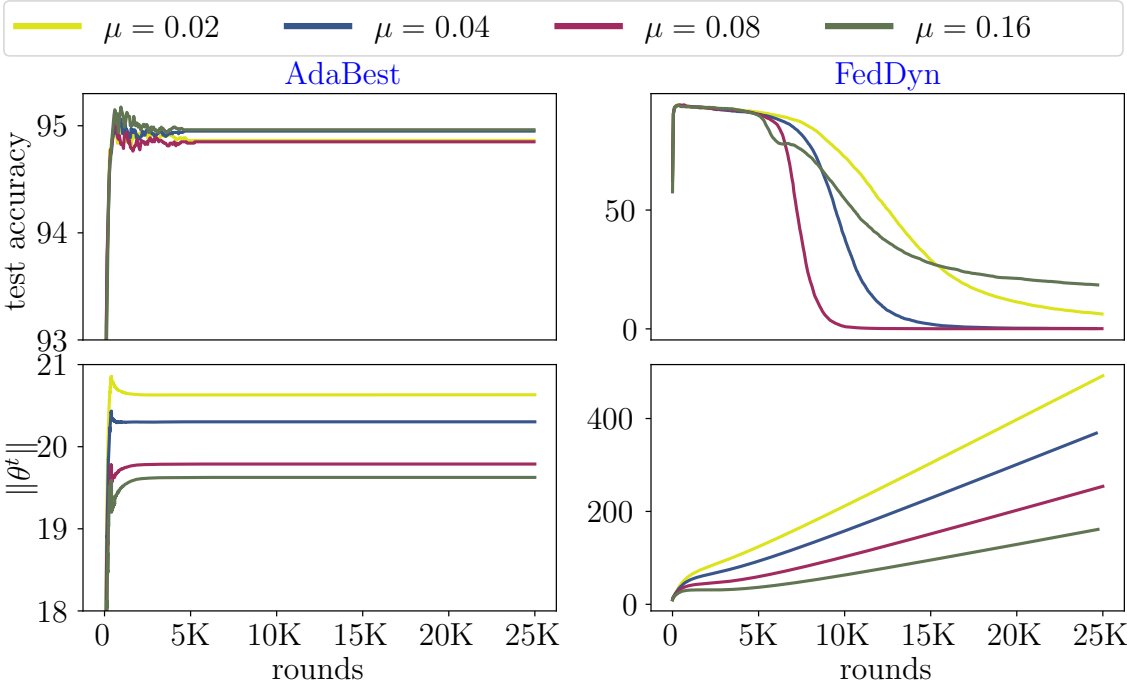


Figure D.3: μ sensitivity of [AdaBest](#) (on the left) and [FedDyn](#) (on the right). The horizontal axis which shows the number of communication rounds is shared for each column of subplots. Top and bottom row show the test accuracy (in percentage) and norm of cloud parameters respectively. Note that the vertical axis is not shared as for clarity. This means that the scale of difference between converging point of $\|\theta^t\|$ in [AdaBest](#) is largely different from the divergence scale of the same quantity for [FedDyn](#)

D.2.7 Discount factor sensitivity

To investigate how the choice of β impacts the generalization performance, we conduct an experiment with varying β and the rate of client participation. The relation comes from the fact that in lower rates of client participation, the variance of the pseudo-gradients is higher and so a lower β is required both in order to avoid explosion of $\|\theta^t\|$ and also to propagate estimation error from the previous rounds as explained in Section 7.4.3. For this experiment we use the same setup as the one used in

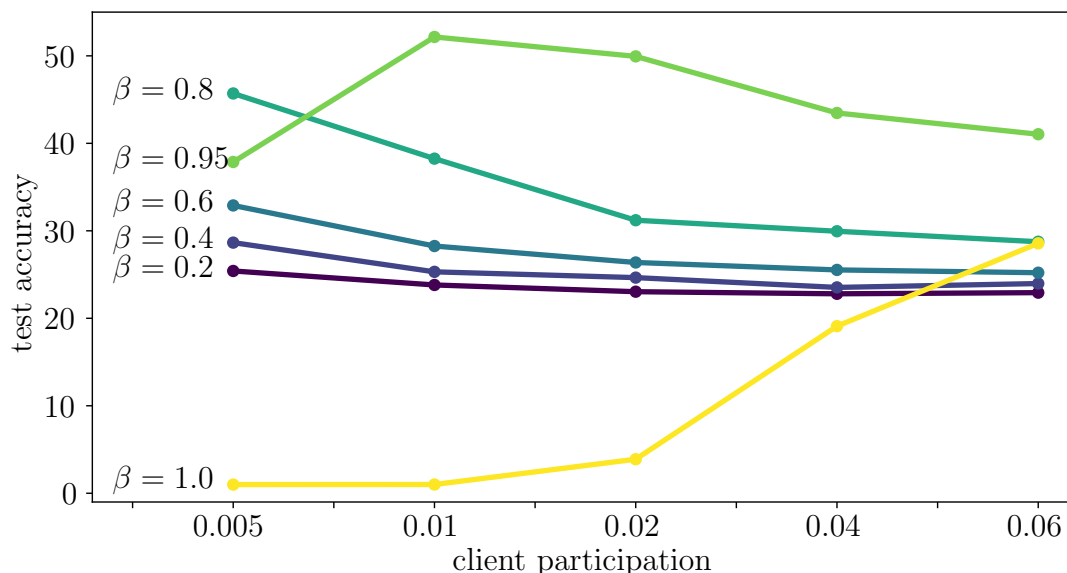


Figure D.4: The sensitivity of the test accuracy (in percentage) to different rates of client participation and values of β . The training is done on a partitioning of [CIFAR-100](#) with 1000 training clients. The numbers on the horizontal axis show the fractions of the total clients sampled at each round

figure 7.3 (see Section D.2.4 for details). figure D.4 implies that when the rate of client participation is 0.005 (5 clients participating in each round out of 1000 training clients) the optimal β is between 0.8 and 0.95. With a larger rate of client participation, the optimal value moves away from 0.8 towards between 0.95 and 1.0;

This observation is aligned with our hypothesis on the impact of variance of pseudo-gradients on the estimation error and norm of the cloud parameters. Another interesting point is that for a wide range of β values, the performance remains almost stable regardless of the rate of client participation. Additionally, β values closer to 1 seem to be suitable for higher rates of client participation, which is suggested by the curve corresponding to $\beta = 1.0$ rising as the rate of client participation increases. This case ($\beta = 1.0$) is the most similar to formulation of our baselines where the estimations of oracle gradients are not scaled properly from one round to another.

Bibliography

- D. A. E. Acar. Feddyn. <https://github.com/alpemreacar/FedDyn>, 2021. [Accessed 12-Mar-2022].
- D. A. E. Acar, Y. Zhao, R. Matas, M. Mattina, P. Whatmough, and V. Saligrama. Federated learning based on dynamic regularization. In *International Conference on Learning Representations*, 2020.
- A. Ajalloeian and S. U. Stich. On the convergence of sgd with biased gradients. *arXiv preprint arXiv:2008.00051*, 2020.
- G. Alain and Y. Bengio. Understanding intermediate layers using linear classifier probes. *arXiv preprint arXiv:1610.01644*, 2016.
- J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.
- R. Babanezhad Harikandeh, M. O. Ahmed, A. Virani, M. Schmidt, J. Konečný, and S. Sallinen. Stopwasting my gradients: Practical svrg. *Advances in Neural Information Processing Systems*, 28, 2015.
- Y. Bengio. Deep learning of representations for unsupervised and transfer learning. In *Proceedings of ICML workshop on unsupervised and transfer learning*, pages 17–36. JMLR Workshop and Conference Proceedings, 2012.
- Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19, 2006.
- Y. Bengio, F. Bastien, A. Bergeron, N. Boulanger-Lewandowski, T. Breuel, Y. Chherawala, M. Cisse, M. Côté, D. Erhan, J. Eustache, et al. Deep learners benefit more from out-of-distribution examples. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 164–172. JMLR Workshop and Conference Proceedings, 2011.
- J. Bi and S. R. Gunn. A variance controlled stochastic method with biased estimation for faster non-convex optimization. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 135–150. Springer, 2021.
- S. Bozinovski. Reminder of the first paper on transfer learning in neural networks, 1976. *Informatica*, 44(3), 2020.
- P. Brazdil, M. Gams, S. Sian, L. Torgo, and W. Velde. Learning in distributed systems and multi-agent environments. In *European Working Session on Learning*, pages 412–423. Springer, 1991.

- J. P. Callan and P. E. Utgoff. Constructive induction on domain information. In *Proceedings of the ninth National conference on Artificial intelligence-Volume 2*, pages 614–619, 1991.
- A. Cauchy et al. Méthode générale pour la résolution des systemes d'équations simultanées. *Comp. Rend. Sci. Paris*, 25(1847):536–538, 1847.
- X. Chen, S. Xie, and K. He. An empirical study of training self-supervised vision transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9640–9649, 2021.
- N. Churamani, O. Kara, and H. Gunes. Domain-incremental continual learning for mitigating bias in facial expression and action unit recognition. *IEEE Transactions on Affective Computing*, 2022.
- W. M. Czarnecki, S. Osindero, R. Pascanu, and M. Jaderberg. A deep neural network's loss surface contains every low-dimensional pattern. *arXiv preprint arXiv:1912.07559*, 2019.
- R. Davis and R. G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial intelligence*, 20(1):63–109, 1983.
- M. De Bollivier, P. Gallinari, and S. Thiria. Cooperation of neural nets and task decomposition. In *IJCNN-91-Seattle International Joint Conference on Neural Networks*, volume 2, pages 573–576. IEEE, 1991.
- J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.
- J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. In *International conference on machine learning*, pages 647–655. PMLR, 2014.
- M. L. Dowell and R. D. Bonnell. Learning for distributed artificial intelligence systems. In *Proceedings of the Twenty-Third Southeastern Symposium on System Theory, Robert Werner (Ed.)*, pages 218–221, 1991.
- J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.
- S. Fahlman and C. Lebiere. The cascade-correlation learning architecture. *Advances in neural information processing systems*, 2, 1989.
- T. E. Fawcett and P. E. Utgoff. A hybrid method for feature generation. In *Machine Learning Proceedings 1991*, pages 137–141. Elsevier, 1991.
- K. Fernandes and J. S. Cardoso. Hypothesis transfer learning based on structural model similarity. *Neural Computing and Applications*, 31(8):3417–3430, 2019.

- P. Foret, A. Kleiner, H. Mobahi, and B. Neyshabur. Sharpness-aware minimization for efficiently improving generalization. In *International Conference on Learning Representations*, 2020.
- L. H. Fowl, J. Geiping, W. Czaja, M. Goldblum, and T. Goldstein. Robbing the fed: Directly obtaining private data in federated learning with modified models. In *International Conference on Learning Representations*, 2021.
- K. Fukushima and S. Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer, 1982.
- R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.
- X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.
- I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.
- I. J. Goodfellow, M. Mirza, D. Xiao, A. Courville, and Y. Bengio. An empirical investigation of catastrophic forgetting in gradient-based neural networks, 2013.
- G. Griffin, A. Holub, and P. Perona. Caltech-256 object category dataset, 2007.
- K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.
- K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- K. He, R. Girshick, and P. Dollár. Rethinking imagenet pre-training. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4918–4927, 2019a.
- K. He, H. Fan, Y. Wu, S. Xie, and R. Girshick. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9729–9738, 2020.
- T. He, Z. Zhang, H. Zhang, Z. Zhang, J. Xie, and M. Li. Bag of tricks for image classification with convolutional neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 558–567, 2019b.
- J. Hecht et al. The bandwidth bottleneck. *Nature*, 536(7615):139–142, 2016.

- G. Hinton, O. Vinyals, J. Dean, et al. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2(7), 2015.
- G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.
- G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.
- T. Hong, M. Fang, and D. Hilder. Pd classification by a modular neural network based on task decomposition. *IEEE transactions on dielectrics and electrical insulation*, 3(2):207–212, 1996.
- T.-M. H. Hsu, H. Qi, and M. Brown. Measuring the effects of non-identical data distribution for federated visual classification. *arXiv preprint arXiv:1909.06335*, 2019.
- Y.-C. Hsu, Y.-C. Liu, A. Ramasamy, and Z. Kira. Re-evaluating continual learning scenarios: A categorization and case for strong baselines. *arXiv preprint arXiv:1810.12488*, 2018.
- G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.
- S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.
- K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun. What is the best multi-stage architecture for object recognition? In *2009 IEEE 12th international conference on computer vision*, pages 2146–2153. IEEE, 2009.
- R. Johnson and T. Zhang. Accelerating stochastic gradient descent using predictive variance reduction. *Advances in neural information processing systems*, 26:315–323, 2013.
- P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings, et al. Advances and open problems in federated learning. *Foundations and Trends® in Machine Learning*, 14(1–2):1–210, 2021.
- I. P. Kaminow and T. Li. *Optical fiber telecommunications IV-B: systems and impairments*, volume 2. Elsevier, 2002.
- F. Kanavati and M. Tsuneki. Partial transfusion: on the expressive influence of trainable batch norm parameters for transfer learning. In *Medical Imaging with Deep Learning*, pages 338–353. PMLR, 2021.

- J. Kang, M. Rhee, and K. H. Kang. Revisiting knowledge transfer: Effects of knowledge characteristics on organizational effort for knowledge transfer. *Expert Systems with Applications*, 37(12):8155–8160, 2010.
- S. P. Karimireddy, S. Kale, M. Mohri, S. Reddi, S. Stich, and A. T. Suresh. Scaffold: Stochastic controlled averaging for federated learning. In *International Conference on Machine Learning*, pages 5132–5143. PMLR, 2020.
- G. Kasparov. Worry about human (not machine) intelligence, 2021. URL <https://www.britannica.com/topic/Worry-About-Human-Not-Machine-Intelligence-2119055>.
- N. S. Keskar and R. Socher. Improving generalization performance by switching from adam to sgd. *arXiv preprint arXiv:1712.07628*, 2017.
- H. M. Khamseh and D. R. Jolly. Knowledge transfer in alliances: determinant factors. *Journal of Knowledge Management*, 2008.
- D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *ICLR (Poster)*, 2015.
- J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.
- J. Konečný, J. Liu, P. Richtárik, and M. Takáč. Mini-batch semi-stochastic gradient descent in the proximal setting. *IEEE Journal of Selected Topics in Signal Processing*, 10(2):242–255, 2015.
- J. Konečný, H. B. McMahan, D. Ramage, and P. Richtárik. Federated optimization: Distributed machine learning for on-device intelligence. *arXiv preprint arXiv:1610.02527*, 2016.
- S. Kornblith, J. Shlens, and Q. V. Le. Do better imagenet models transfer better? In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2661–2671, 2019.
- A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.
- A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.
- A. Kumar, A. Raghunathan, R. M. Jones, T. Ma, and P. Liang. Fine-tuning can distort pretrained features and underperform out-of-distribution. In *International Conference on Learning Representations*, 2021.

- I. Laina, Y. M. Asano, and A. Vedaldi. Measuring the interpretability of unsupervised representations via quantized reversed probing. In *International Conference on Learning Representations*, 2021.
- Q. Lao, X. Jiang, M. Havaei, and Y. Bengio. Continuous domain adaptation with variational domain-agnostic feature replay. *arXiv preprint arXiv:2003.04382*, 2020.
- Q. Lao, X. Jiang, and M. Havaei. Hypothesis disparity regularized mutual information maximization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 8243–8251, 2021.
- Q. V. Le, R. Monga, M. Devin, G. Corrado, K. Chen, M. Ranzato, J. Dean, and A. Y. Ng. Building high-level features using large scale unsupervised learning. *CoRR*, abs/1112.6209, 2011. URL <http://arxiv.org/abs/1112.6209>.
- Y. LeCun. The mnist database of handwritten digits. <http://yann.lecun.com/exdb/mnist/>, 1998.
- Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.
- Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.
- H. Lee, R. Grosse, R. Ranganath, and A. Y. Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th annual international conference on machine learning*, pages 609–616, 2009.
- R. Lengellé and T. Denooux. Training mlps layer by layer using an objective function for internal representations. *Neural Networks*, 9(1):83–97, 1996.
- D. Li and J. Wang. Fedmd: Heterogenous federated learning via model distillation. *arXiv preprint arXiv:1910.03581*, 2019.
- F.-F. Li, M. Andreto, M. Ranzato, and P. Perona. Caltech 101, Apr 2022.
- H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein. Visualizing the loss landscape of neural nets. In *Advances in Neural Information Processing Systems*, pages 6389–6399, 2018.
- T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smithy. Feddane: A federated newton-type method. In *2019 53rd Asilomar Conference on Signals, Systems, and Computers*, pages 1227–1231. IEEE, 2019.

- T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith. Federated optimization in heterogeneous networks. *Proceedings of Machine Learning and Systems*, 2:429–450, 2020.
- Z. Li and D. Hoiem. Learning without forgetting. *IEEE transactions on pattern analysis and machine intelligence*, 40(12):2935–2947, 2017.
- X. Liang, S. Shen, J. Liu, Z. Pan, E. Chen, and Y. Cheng. Variance reduced local sgd with lower communication complexity. *arXiv preprint arXiv:1912.12844*, 2019.
- Y. Liang, L. Zhu, X. Wang, and Y. Yang. A simple episodic linear probe improves visual recognition in the wild. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9559–9569, 2022.
- J. Lienen and E. Hüllermeier. Instance weighting through data imprecisation. *International Journal of Approximate Reasoning*, 134:1–14, 2021.
- T. Lin, L. Kong, S. U. Stich, and M. Jaggi. Ensemble distillation for robust model fusion in federated learning. *Advances in Neural Information Processing Systems*, 33:2351–2363, 2020.
- L. Liu, H. Jiang, P. He, W. Chen, X. Liu, J. Gao, and J. Han. On the variance of the adaptive learning rate and beyond. In *International Conference on Learning Representations*, 2019.
- X. Liu, C. Wu, M. Menta, L. Herranz, B. Raducanu, A. D. Bagdanov, S. Jui, and J. v. de Weijer. Generative feature replay for class-incremental learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 226–227, 2020.
- L. Luo, Y. Xiong, Y. Liu, and X. Sun. Adaptive gradient methods with dynamic bound of learning rate. In *International Conference on Learning Representations*, 2018.
- N. Ma, X. Zhang, H.-T. Zheng, and J. Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *European Conference on Computer Vision*, pages 122–138. Springer, 2018.
- M. McCloskey and N. J. Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pages 109–165. Elsevier, 1989.
- R. T. McCoy, R. Frank, and T. Linzen. Does syntax need to grow on trees? sources of hierarchical inductive bias in sequence-to-sequence networks. *Transactions of the Association for Computational Linguistics*, 8:125–140, 2020.
- B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. PMLR, 2017.

- R. Miśkiewicz. The importance of knowledge transfer on the energy market. *Polityka Energetyczna-Energy Policy Journal*, pages 49–62, 2018.
- M. C. Mozer. *Early parallel processing in reading: A connectionist approach*. Lawrence Erlbaum Associates, Inc, 1987.
- T. Murata and T. Suzuki. Bias-variance reduced local sgd for less heterogeneous federated learning. In *International Conference on Machine Learning*, pages 7872–7881. PMLR, 2021.
- V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Icml*, 2010.
- Y. E. Nesterov. A method for solving the convex programming problem with convergence rate $o(1/k^2)$. In *Dokl. akad. nauk Sssr*, volume 269, pages 543–547, 1983.
- B. Neyshabur, H. Sedghi, and C. Zhang. What is being transferred in transfer learning? *Advances in neural information processing systems*, 33:512–523, 2020.
- L. M. Nguyen, J. Liu, K. Scheinberg, and M. Takáč. Sarah: A novel method for machine learning problems using stochastic recursive gradient. In *International Conference on Machine Learning*, pages 2613–2621. PMLR, 2017.
- C. Noel and S. Osindero. Dogwild!-distributed hogwild for cpu & gpu. In *NIPS Workshop on Distributed Machine Learning and Matrix Computations*, pages 693–701, 2014.
- R. Pathak and M. J. Wainwright. Fedsplit: An algorithmic framework for fast federated optimization. *Advances in Neural Information Processing Systems*, 33:7057–7066, 2020.
- D. N. Perkins, G. Salomon, et al. Transfer of learning. *International encyclopedia of education*, 2:6452–6457, 1992.
- M. Popel and O. Bojar. Training tips for the transformer model. *The Prague Bulletin of Mathematical Linguistics*, 110(1):43–70, 2018.
- L. Y. Pratt. *Transferring previously learned backpropagation neural networks to new learning tasks*. Rutgers The State University of New Jersey-New Brunswick, 1993.
- L. Y. Pratt and C. A. Kamm. Improving a phoneme classification neural network through problem decomposition. In *IJCNN-91-Seattle international joint conference on neural networks*, volume 2, pages 821–826. IEEE, 1991.
- L. Y. Pratt, J. Mostow, C. A. Kamm, A. A. Kamm, et al. Direct transfer of learned information among neural networks. In *Aaai*, volume 91, pages 584–589, 1991.

- F. J. Provost and D. N. Hennessy. Distributed machine learning: Scaling up with coarse-grained parallelism. In *ISMB*, pages 340–347, 1994.
- X. Qian and D. Klabjan. The impact of the mini-batch size on the variance of gradients in stochastic gradient descent. *arXiv preprint arXiv:2004.13146*, 2020.
- I. Radosavovic, R. P. Kosaraju, R. Girshick, K. He, and P. Dollár. Designing network design spaces. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10428–10436, 2020.
- M. Raghu, C. Zhang, J. Kleinberg, and S. Bengio. Transfusion: Understanding transfer learning for medical imaging. *Advances in neural information processing systems*, 32, 2019.
- R. Raina, A. Battle, H. Lee, B. Packer, and A. Y. Ng. Self-taught learning: transfer learning from unlabeled data. In *Proceedings of the 24th international conference on Machine learning*, pages 759–766, 2007.
- M. Ranzato, C. Poultney, S. Chopra, and Y. Cun. Efficient learning of sparse representations with an energy-based model. *Advances in neural information processing systems*, 19, 2006.
- S.-A. Rebuffi, A. Kolesnikov, G. Sperl, and C. H. Lampert. icarl: Incremental classifier and representation learning. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 2001–2010, 2017.
- B. Recht, C. Re, S. Wright, and F. Niu. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Advances in neural information processing systems*, 24, 2011.
- S. J. Reddi, J. Konečný, P. Richtárik, B. Póczós, and A. Smola. Aide: Fast and communication efficient distributed optimization. *arXiv preprint arXiv:1608.06879*, 2016.
- S. J. Reddi, S. Kale, and S. Kumar. On the convergence of adam and beyond. In *International Conference on Learning Representations*, 2018. URL <https://openreview.net/forum?id=ryQu7f-RZ>.
- M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *IEEE international conference on neural networks*, pages 586–591. IEEE, 1993.
- H. Robbins and S. Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.
- Y. Roh, G. Heo, and S. E. Whang. A survey on data collection for machine learning: a big data-ai integration perspective. *IEEE Transactions on Knowledge and Data Engineering*, 33(4):1328–1347, 2019.

- D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.
- R. Salakhutdinov and G. Hinton. Deep boltzmann machines. In D. van Dyk and M. Welling, editors, *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics*, volume 5 of *Proceedings of Machine Learning Research*, pages 448–455, Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA, 16–18 Apr 2009. PMLR.
- M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4510–4520. IEEE, 2018.
- S. Sayegh. Inheriting knowledge in neural networks. In *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*, volume 1, pages 841–846. IEEE, 1992.
- M. Schmidt, N. Le Roux, and F. Bach. Minimizing finite sums with the stochastic average gradient. *Mathematical Programming*, 162(1):83–112, 2017.
- S. Sedice. 5 strong chess engines and the best ways to train with them, 2020. URL <https://www.houseofstaunton.com/blog/5-strong-chess-engines-and-the-best-ways-to-train-with-them.html>.
- P. Sermanet, K. Kavukcuoglu, S. Chintala, and Y. LeCun. Pedestrian detection with unsupervised multi-stage feature learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3626–3633, 2013.
- O. Shamir, N. Srebro, and T. Zhang. Communication-efficient distributed optimization using an approximate newton-type method. In *International conference on machine learning*, pages 1000–1008. PMLR, 2014.
- A. J. Sharkey. Multi-net systems. In *Combining artificial neural nets*, pages 1–30. Springer, 1999.
- A. J. C. Sharkey. Modularity, combining and artificial neural nets. *Connection Science*, 9(1):3–10, 1997.
- H. Shimodaira. Improving predictive inference under covariate shift by weighting the log-likelihood function. *Journal of statistical planning and inference*, 90(2): 227–244, 2000.
- R. Shokri and V. Shmatikov. Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 1310–1321, 2015.
- K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

- I. Skorokhodov and M. Burtsev. Loss surface sightseeing by multi-point optimization. *arXiv preprint arXiv:1910.03867*, 2019.
- N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014. URL <http://jmlr.org/papers/v15/srivastava14a.html>.
- S. U. Stich. Local sgd converges fast and communicates little. In *International Conference on Learning Representations*, 2018.
- B. S. Subedi. Emerging trends of research on transfer of learning. *International education journal*, 5(4):591–599, 2004.
- C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.
- C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.
- M. Tan and Q. V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *ICML*, 2019.
- M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.
- T. Tieleman, G. Hinton, et al. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.
- H. Touvron, A. Vedaldi, M. Douze, and H. Jégou. Fixing the train-test resolution discrepancy. *Advances in neural information processing systems*, 32, 2019.
- G. G. Towell and J. W. Shavlik. Extracting refined rules from knowledge-based neural networks. *Machine learning*, 13(1):71–101, 1993.
- G. G. Towell, J. W. Shavlik, M. O. Noordewier, et al. Refinement of approximate domain theories by knowledge-based neural networks. In *Proceedings of the eighth National conference on Artificial intelligence*, volume 2, pages 861–866. Boston, MA, 1990.
- P. Tseng. An incremental gradient (-projection) method with momentum term and adaptive stepsize rule. *SIAM Journal on Optimization*, 8(2):506–531, 1998.

- D. Ulyanov, A. Vedaldi, and V. Lempitsky. Instance normalization: The missing ingredient for fast stylization. *arXiv preprint arXiv:1607.08022*, 2016.
- P. E. Utgoff. *Shift of bias for inductive concept learning*. Rutgers The State University of New Jersey-New Brunswick, 1984.
- G. M. van de Ven and A. S. Tolias. Three scenarios for continual learning, 2019.
- J. E. Van Engelen and H. H. Hoos. A survey on semi-supervised learning. *Machine Learning*, 109(2):373–440, 2020.
- F. Varno, B. H. Soleimani, M. Saghayei, L. Di Jorio, and S. Matwin. Efficient neural task adaptation by maximum entropy initialization. *arXiv preprint arXiv:1905.10698*, 2019.
- F. Varno, L. M. Petry, L. Di Jorio, and S. Matwin. Learn faster and forget slower via fast and stable task adaptation. *arXiv preprint arXiv:2007.01388*, 2020.
- F. Varno, M. Saghayei, L. Rafiee Sevyeri, S. Gupta, S. Matwin, and M. Havaei. Adabest: Minimizing client drift in federated learning via adaptive bias estimation. In *European Conference on Computer Vision*, pages 710–726. Springer, 2022a.
- F. Varno, B. H. Soleimani, M. Saghayei, L. Di Jorio, and S. Matwin. Method and system for initializing a neural network, July 7 2022b. US Patent App. 17/609,296.
- A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.
- P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, P.-A. Manzagol, and L. Bottou. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of machine learning research*, 11(12), 2010.
- A. Waibel. Consonant recognition by modular construction of large phonemic time-delay neural networks. *Advances in neural information processing systems*, 1, 1988.
- A. Waibel, H. Sawai, and K. Shikano. Modularity and scaling in large phonemic neural networks. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(12):1888–1898, 1989.
- J. Wang, V. Tantia, N. Ballas, and M. Rabbat. Slowmo: Improving communication-efficient distributed sgd with slow momentum. *arXiv preprint arXiv:1910.00643*, 2019.
- J. Wang, Q. Liu, H. Liang, G. Joshi, and H. V. Poor. Tackling the objective inconsistency problem in heterogeneous federated optimization. *Advances in neural information processing systems*, 33:7611–7623, 2020.

- C. Wetterich. Fine-tuning problem and the renormalization group. *Physics Letters B*, 140(3-4):215–222, 1984.
- Y. Wu and K. He. Group normalization. In *Proceedings of the European conference on computer vision (ECCV)*, pages 3–19, 2018.
- L. Xiao and T. Zhang. A proximal stochastic gradient method with progressive variance reduction. *SIAM Journal on Optimization*, 24(4):2057–2075, 2014.
- S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500, 2017.
- H. Yu, R. Jin, and S. Yang. On the linear speedup analysis of communication efficient momentum sgd for distributed non-convex optimization. In *International Conference on Machine Learning*, pages 7184–7193. PMLR, 2019.
- S. Zagoruyko and N. Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.
- M. D. Zeiler. Adadelata: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.
- M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.
- J. Zhang, C. De Sa, I. Mitliagkas, and C. Ré. Parallel sgd: When does averaging help? *arXiv preprint arXiv:1606.07365*, 2016.
- J. Zhang, S. P. Karimireddy, A. Veit, S. Kim, S. J. Reddi, S. Kumar, and S. Sra. Why adam beats sgd for attention models. *arXiv preprint arXiv:1912.03194*, 2019.
- X. Zhang, M. Hong, S. Dhople, W. Yin, and Y. Liu. Fedpd: A federated learning framework with optimal rates and adaptivity to non-iid data. *arXiv preprint arXiv:2005.11418*, 2020.
- Y. Zhao, M. Li, L. Lai, N. Suda, D. Civin, and V. Chandra. Federated learning with non-iid data. *arXiv preprint arXiv:1806.00582*, 2018.
- Y. Zhong, J. Sullivan, and H. Li. Face attribute prediction using off-the-shelf cnn features. In *2016 International Conference on Biometrics (ICB)*, pages 1–7. IEEE, 2016.
- L. Zhu and S. Han. Deep leakage from gradients. In *Federated learning*, pages 17–31. Springer, 2020.
- Z. Zhu, J. Hong, and J. Zhou. Data-free knowledge distillation for heterogeneous federated learning. In *International Conference on Machine Learning*, pages 12878–12889. PMLR, 2021.

M. Zinkevich, M. Weimer, L. Li, and A. Smola. Parallelized stochastic gradient descent. *Advances in neural information processing systems*, 23, 2010.