Department of Mathematics and Geosciences
Master Degree in Data Science and Scientific Computing

The Dynamics of Learning Beyond Stochastic Gradient Descent

Supervisor:
Prof. Sebastian Goldt

Candidate:
Doğan Can Demirbilek

Academic Year 2020/2021

To my family

Acknowledgements

This is where you will normally thank your advisor, colleagues, family and friends, as well as funding and institutional support. In our case, we will give our praises to the people who developed the ideas and tools that allow us to push open science a little step forward by writing plain-text, transparent, and reproducible theses in R Markdown.

We must be grateful to John Gruber for inventing the original version of Markdown, to John MacFarlane for creating Pandoc (`http://pandoc.org`) which converts Markdown to a large number of output formats, and to Yihui Xie for creating `knitr` which introduced R Markdown as a way of embedding code in Markdown documents, and `bookdown` which added tools for technical and longer-form writing.

Special thanks to Chester Ismay, who created the `thesisdown` package that helped many a PhD student write their theses in R Markdown. And a very special thanks to John McManigle, whose adaption of Sam Evans' adaptation of Keith Gillow's original maths template for writing an Oxford University DPhil thesis in LaTeX provided the template that I in turn adapted for R Markdown.

Finally, profuse thanks to JJ Allaire, the founder and CEO of RStudio, and Hadley Wickham, the mastermind of the tidyverse without whom we'd all just given up and done data science in Python instead. Thanks for making data science easier, more accessible, and more fun for us all.

Ulrik Lyngs
Linacre College, Oxford
2 December 2018

Abstract

This *R Markdown* template is for writing an Oxford University thesis. The template is built using Yihui Xie's `bookdown` package, with heavy inspiration from Chester Ismay's `thesisdown` and the `OxThesis` LaTeX template (most recently adapted by John McManigle).

This template's sample content include illustrations of how to write a thesis in R Markdown, and largely follows the structure from this R Markdown workshop.

Congratulations for taking a step further into the lands of open, reproducible science by writing your thesis using a tool that allows you to transparently include tables and dynamically generated plots directly from the underlying data. Hip hooray!

Contents

List of Figures

List of Abbreviations

ANN . . . . . Artificial Neural Network.

BP . . . . . Backpropagation.

DFA . . . . . Direct Feedback Alignment.

BCE . . . . . Binary Cross Entropy.

FA . . . . . Feedback Alignment.

NTK . . . . . Neural Tangent Kernel.

t-SNE . . . . . t-Distributed Stochastic Neighbor Embedding.

SGD . . . . . Stochastic Gradient Descent.

NAG . . . . . Nesterov Accelerated Gradient.

Introduction

Artificial neural networks (ANNs) are a collection of connected computational nodes inspired by biological neural networks. Each connection can transmit a helpful signal to another computational node like synapses in a brain. ANNs demonstrated colossal advancements in the last decades. Thanks to these advancements, it is possible to solve complex problems in computer vision, speech recognition, and natural language processing within a reasonable amount of time and with satisfactory performance. These advancements were actualized through an old but robust algorithm called backpropagation (BP). BP is a training algorithm for ANNs based on repeatedly adjusting network weights to minimize the difference (loss) between the output of the network and the ground truth [1].

Although nowadays BP is the workhorse algorithm for training ANNs, it has some drawbacks, and it is not the only alternative. Recent studies offered different algorithms to train ANNs by addressing these drawbacks. These algorithms have other properties and principles than BP. Some of them are competitive with BP, or they even outperform the BP in terms of performance or convergence speed for specific problems.

This thesis investigates the learning structures through BP and one of the alternative algorithm called direct feedback alignment (DFA) on the particular problem. Unlike BP, the error is propagated through a fixed random matrix instead of the layers' weights in DFA. Then network learns how to make this feedback useful [2]. Due to this error propagation mechanism, DFA is considered more biologically plausible than BP, and it opens the gate of parallelism in the training phase of ANNs.

The main problem at hand is known as the parity learning problem. Previous results showed that these parities are learnable by BP and lazy methods in a more simple

setting, whereas it is only learnable by BP in a more complex setting [3]. That is why it is intriguing to test alternative algorithms on this problem to understand their learning dynamics and capabilities.

The experiment results might lead us to three possible outcomes. First, we might acquire a similar performance as BP. If it is the case, it would be beneficial to test DFA and BP on a more challenging problem. Second, there might be a gap between BP and DFA then it would be intriguing to understand where the difference is coming from and how we can close this gap. Third, the alternative algorithm might not even learn, and in this case, it is interesting to ask what makes a problem learnable by BP but not DFA. In all cases, results should help to understand the dynamics of learning of both methods.

For applying BP and DFA in a more realistic setting, experiments are performed on the MNIST dataset by imitating the parity learning problem as it is described in [3]. After putting DFA to this frame, the reason behind the results is interpreted, and possible improvements are motivated and implemented.

Chapter 1 constructs the theoretical bases of the algorithms that are used for the experiments. These bases are composed of simple definitions, mathematical foundations, and the drawbacks of the algorithms. They help to dig deeper into the learning structures of the training algorithms. It is expected to have more control over their learning behaviors by adjusting components of these foundations. Also, it is beneficial to have these theoretical bases for acquiring a better understanding of the further interventions. Moreover, these theoretical foundations are used to implement the algorithms from scratch to use in experiments.

Chapter 2 introduces the parity learning problem at hand. First, the formal definition of the problem is demonstrated then how the problem is imitated by using the MNIST dataset is explained in detail. This part is also highly correlated with the training phase of the algorithms. Later experiment results from [3] are replicated to have a concrete picture of previous studies.

Chapter 3 presents the results of the experiments. This chapter is the main contribution of this study. The first experiment is testing DFA on the same problem

and observing the difference with BP. As it is specified before, depending on the experiment outcome, different further experiments are performed. Such as closing the gap between BP and DFA, if any, and trying harder problems if we acquire similar results or explaining why DFA cannot learn the problem if we get a similar behavior as lazy methods. In addition to the main experiment, we performed side experiments because these side experiments might help us to improve the performance of the algorithms or they can be helpful to understand the learning dynamics of them. For instance, using different random matrices for DFA, trying adaptive optimization algorithms with BP and DFA, and observing hidden representation of BP and DFA to understand if the networks learn the digits individually to calculate the parities or memorizes the data without knowing the digits.

TODO: Try to fix the conclusion reference

Conclusion wraps up the findings from experiments by summarizing the key findings. It creates a path for future studies that are not covered in this study.

1 Theoretical Foundations

Contents

1.1 Backpropagation

BP is one of the first algorithms that show ANNs could learn hidden representations well. Numerous studies showed that ANNs trained with BP could capture similar information as biological neural networks (e.g., specific nodes learn the edges, corners). We need three components for BP, a dataset composed of input-output pairs, a network consisting of parameters (weights and biases), and it allows the input to flow through the network to have output. We need a loss function to measure the difference between the output of the network and the ground truth that we have from the dataset.

The main goal of BP is computing the gradients of the loss function (a measure of difference) concerning the parameters of neural networks by using the chain rule. These gradients show how much the parameter needs to change (positively or negatively) to minimize the loss function. After efficiently calculating the gradients, we can adjust the network parameters using gradient descent or its variants.

Although BP is an older idea, it earned popularity with [1] because it presented how BP can make a network to learn the representations. After this popularity, the community published numerous practical and theoretical papers that investigated the dynamics of BP. It would be repeat and infeasible to show all the aspects again. However, for completeness and a smoother transition from BP to DFA, it is beneficial to have visual and mathematical explanations showing how the algorithms propagate the errors and the weights. For the mathematical foundations, a binary classification task will be demonstrated with binary cross-entropy (BCE) loss as an example in appendix A. This example is not chosen arbitrarily. Indeed the parity problem that MNIST imitates is a binary classification problem. In addition to this, equations from appendix A are used to implement BP from scratch to have more control over the process. Then the exact implementation is modified to obtain DFA. The same set of steps are valid for different loss functions and activation functions. Only the calculations will be slightly different. The general idea is the same: obtaining the gradients by calculating the derivative of the loss function concerning the parameters.

In figure 1.1 we have a simple network with only a hidden layer that shows the error transportation configuration in BP. W_i are the weights, h_i are the output of the hidden layers that is denoted as i, $\hat{y}$ is the output of the network, and y is the ground truth, for the sake of simplicity, biases are not showed in this figure. It is important to note that in BP, the transpose of weight is propagated to calculate the gradients. In literature, this issue is known as the weight transport problem, and it is one of the most criticized disadvantages of BP.

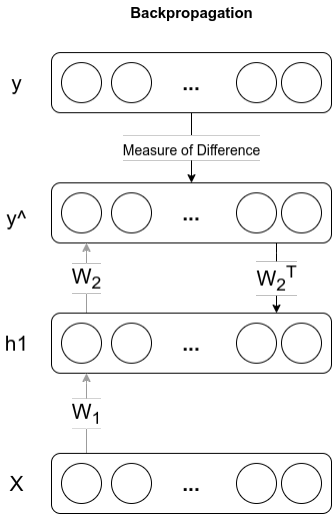

Figure 1.1: Error Transportation in BP

1.1.1 Drawbacks of BP

We know that biological neurons inspired ANNs. However, recent studies showed that BP is not precisely how biological neurons learn [4]. This brings a term called biological plausibility of an algorithm that indicates the algorithm's consistency with existing biological, medical, and neuroscientific knowledge. That is why the community proposed many alternative algorithms by addressing these limitations of BP. In the light of this term, we can put in order the drawbacks of BP as the following:

- **Biological implausibility:**
  - The BP computation is purely linear whereas biological neurons interleave linear and non-linear operations.
  - BP needs precise knowledge of derivatives of the non-linearities at the operating point used in the corresponding feedforward computation on the feedforward path.
  - BP has to use exact symmetric weights of the feedforward connections.
  - Real neurons communicate by binary values (spikes), not by clean continuous values.

  - The computation has to be precisely clocked to alternate between feedforward and BP phases.
  - It is not clear where the output targets would come from. [4, 5]

- **Vanishing or Exploding Gradients**
- **Lack of Parallel Processing** [6]

Simple interventions may handle some of these drawbacks. For instance, implementing gradient clipping or different activation functions might solve exploding gradients and vanishing gradients. However, they may frequently happen in deeper networks, and they must be considered while training ANNs. On the other hand, some of the drawbacks can not be handled with superficial modifications. For instance, BP is a sequential process, and there are locking mechanisms (forward, backward and update) that ensure none of the processes is executed before its preceding completion. This makes BP infeasible for parallel processing because each execution has to wait for its preceding process. Hence deeper and larger networks' training can be computationally expensive.

Biological plausibility is significant because of a couple of reasons. We know that biological neurons inspired ANNs, and biological plausibility refers to consistency between BP and biological knowledge about the neurons of a brain. Hence, it is interesting to examine the dissimilarity or similarity among them. Besides, there is a field that is the intersection of neuroscience and deep learning, so it is natural to investigate the biological plausibility feature of the algorithms, especially for this field. Furthermore, even though nowadays ANNs might outperform the human brain in a specific task, we are still distant from fully mimicking it. In other words, most of the time, ANNs are very good on a task that they are trained in, but they are not diverse, and some kinds of attacks like adversarial ones can easily trick them. Investigating the learning dynamics of these algorithms may open the doors of diverse ANNs that are not specialized in a single task or make them more robust to attacks.

Alternative algorithms address some of the drawbacks of BP, and they propose a solution to them, but they also demonstrate a couple of them. However, these algorithms can be considered one or more steps closer to more biologically plausible and robust algorithms.

1.2 Direct Feedback Alignment

So far, we have seen how the error is propagated in BP sequentially through a network with the backward pass. Unlike BP, DFA uses a different way to propagate the error. This way uses a random matrix instead of the transpose of the weight matrix, which brings a solution to the weight transport problem. Before explaining how DFA works, it is better to investigate the feedback alignment (FA) algorithm since DFA is the extension of FA.

In [7], authors proved that precise symmetric weights are not required to obtain learning in ANNs. Without these matrices, BP-like learning can be obtained. Any random matrix under some conditions can provide the learning. Implicit dynamics in the standard forward weight updates encourage an alignment between weights and the random matrix. In other words, a random matrix pushes the network in roughly the same direction as BP would. They supported this hypothesis with some experiments on a linear problem and MNIST classification task. The empirical results demonstrate that FA successfully trains the network and has similar performance results as BP on these tasks.

Even though learning still occurs with random matrix and FA offers the solution to the weight transport problem, it does not provide any computational advantage. To extend DFA, we need to change the error propagation mechanism of FA slightly. In FA, the error is propagated through a random matrix, but the backward process is still sequential. DFA extends this idea and propagates the random matrix in parallel to each layer. In other words, DFA takes the loss and distribute it globally to all layers without requiring sequential step. It also creates an opportunity to parallelize the computation that might speed up the training process.

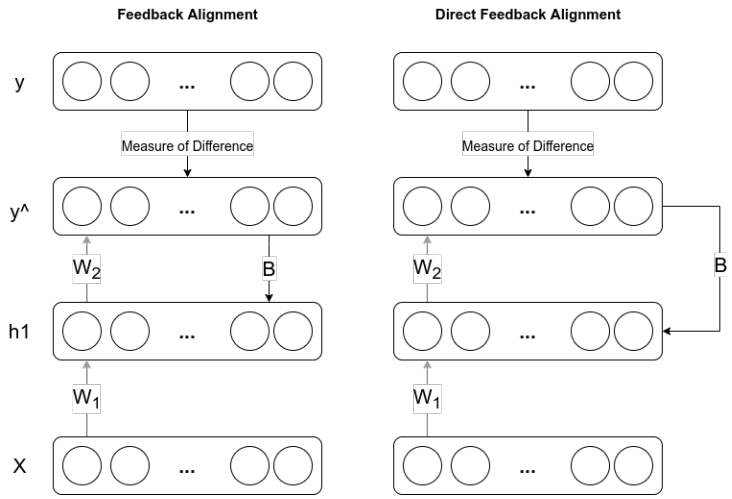

Figure 1.2: Error Transportation in DFA

In figure 1.2 we can see the error transportation configurations for FA and DFA. This figure is the same as the one in [2] but shows only one hidden layer. In fact, with only one hidden layer, FA and DFA are identical.

It is crucial to point out, BP and DFA have different learning dynamics. BP calculates the gradients that point to the steepest descent in the loss function space. On the other hand, FA and DFA provide a different update direction but still descending. It is still descending because empirical and theoretical results proved that the networks' weights align with the random matrix that leads to gradients alignment. Therefore the more alignment we have, the same direction would FA and DFA point as BP. Even though they have different update directions, since they are both descending, the results from [2, 7] showed that FA and DFA are as good as BP in terms of performance for specified tasks in these papers. In addition to this, ANNs trained with DFA show decent separation for labels as in BP's hidden representations of the layers. We can observe this from the t-distributed stochastic neighbor embedding (t-SNE) visualizations of the hidden layers' representations [2]. t-SNE is a method visualizing high-dimensional data which tries to keep the neighbor property in lower dimensions [8].

Recently, a new study has been published which tests the applicability of DFA on modern deep learning tasks and architectures such as neural view synthesis, recommender systems, geometric learning, and natural language processing [9].

Because even though some of the alternative methods are competitive with BP in simple tasks like MNIST, they are not competitive or trainable on more complex tasks. Results showed that DFA successfully trains all these complex architectures with performance close to BP. This study supports that complex tasks can be solved without symmetric weight transport, proving that DFA is suitable for more challenging problems.

Let us use the same example as A to present how gradients are calculated in DFA. After having the mathematical foundations of BP, transition to DFA is relatively easy. The forward pass is the same as BP, whereas, in the backward pass, we need to replace the transpose of the weight matrix, which is used to calculate the gradients with the random matrix. Considering the same example, we have a simple binary classification task with binary cross-entropy loss, and our network has only one hidden layer. In this setting, gradients of the weights can be calculated as the following:

$$\frac{\partial BCE}{\partial w_2} = h_1^T \left(\hat{y} - y\right)$$

There is no change in the calculations of gradients of the last layer, whereas, for the hidden layer, we have:

$$\frac{\partial BCE}{\partial w_1} = (X)^T \left(\hat{y} - y\right)(B) \odot f'(a_1)$$

Please pay attention that w_2^T is replaced with the random matrix B. This means that we can obtain learning by changing either the random matrix or weight matrix. We know that in DFA, B is fixed, so the feedforward weights of the network will learn to make these signals useful by aligning with the BP's teaching signal.

Update rules are the same as BP, which means that gradient descent and its variants can be used.

$$\text{parameter} = \text{parameter} - \text{step size} \times \frac{\partial BCE}{\partial (parameter)}$$

With this tiny modification, DFA brings a solution to some of the drawbacks of BP. Such as using exact symmetric weights of the feedforward connections (weight

transport problem), lack of parallel processing (random matrix can be propagated in parallel), and it is less likely to suffer from vanishing or exploding gradients than BP. Eventually, it proposes a more biologically plausible training method. However, it is not the perfect solution either. Because it assumes there is a global feedback path to propagate the error that might be biologically implausible because feedback has to travel a long physical distance. It also suffers some of the drawbacks of BP. For instance, computation is still purely linear. We still need precise knowledge of derivatives of non-linearities. We still communicate by clean, continuous values, and it is unclear where the output targets would come from. Besides, DFA has an extra task to accomplish while training the ANN that aligns with BP's weights, and a layer can not learn before its preceding layers are aligned. This might spawn performance concerns, and DFA might lag behind BP. Furthermore, DFA fails to train convolutional neural networks which dominate the computer vision tasks [10, 11]. Finally, unlike BP, DFA was not investigated on particular subjects like adversarial attacks and interpretability by the community. This leaves some question marks about the robustness of DFA.

1.3 Lazy Methods

Theoretical results present that especially over-parameterized ANNs (not limited to these networks) trained with gradient-based methods can reach zero training loss with their parameters barely changing. The term lazy does not refer to the poor property of methods, whereas they are called lazy methods because their parameters hardly move during training [12].

Lazy methods are not at the center of the experiments. Hence, detailed explanations of these methods are out of scope in this study, but they have been presented in [3], and they fail to learn the parities in a more complex setting. Hence, they are implemented too for completeness, and it is essential to embody at least a simple definition of them and how they are practically implemented.

1.3.1 Neural Tangent Kernel

Studies showed that neural networks under some conditions are equivalent to gaussian process, and they mathematically approximate the kernel machines if they are trained with gradient descent [13, 14]. Authors of [15] proved that during the training phase, ANNs follow the kernel gradient of the functional loss concerning a new kernel. They named this kernel a Neural Tangent Kernel (NTK). In other words, NTK is a kernel that describes the evolution of an ANN during the learning phase, and it is beneficial to explain the training of ANNs in function space rather than parameters space. It allows us to work with infinite width neural networks using the kernel trick, and it helps us understand the dynamics of learning and inference.

Empirical results demonstrated that the NTK regime performs worse than BP on standard tasks like MNIST. However, NTK is still worth investigating further to understand ANNs' training dynamics since it brings a new perspective on the training phase.

Basic practical implementation of NTK is obtained with three steps. Initially, an extra layer is created with the exact dimensions of the first layer. The second in the forward pass concatenation of these two layers' parameters are given as input to the gated linear unit with 1. Lastly, in the parameters update phase, the extra layer is not considered. With these adjustments, we decoupled the gating from the linearity of the ReLU, and we kept the gates fixed during training.

1.3.2 Random features

Standard random features are where first layer weights are initialized randomly by following a distribution and the train only the second layer. These mechanisms are particularly good at approximating kernels. They are preferred because kernel machines might take too much time to train if the data size is big. In **gaussian features** case, we initialize the first layer weights using gaussian distribution. In contrast, in **ReLU features** and **linear features**, we initialize the first layer

weights uniformly but, for linear features, non-linear activation functions are not used in the forward pass.

1.4 Optimizers

Up to this point, we only mentioned how we could use gradient descent and its variants to update the weights of a network superficially. This part is worth further investigation because many variants provide better convergence properties to find the minimum of the loss function. We may take advantage of these methods to have better performance or faster convergence for BP and DFA. These methods may spawn a significant impact on convergence speed and overall performance. As a reference to the following methods, mostly [16] is used, also the structure of this part and mathematical notations are adapted from the same paper, it is an excellent overview for the optimizers, and it reviews their advantages as well as drawbacks.

1.4.1 Gradient Descent

Gradient descent is a first-order iterative optimization algorithm. It is the most used algorithm to optimize neural networks. It has three variants that depend on how much data we use to compute the gradients. **Batch gradient descent** computes the gradients for the entire dataset and performs only one update. **Stochastic gradient descent** (SGD), in contrast, calculates gradients for each training example and performs parameter update for each of them. Lastly, **mini-batch gradient descent** calculates the gradients of mini-batches and performs updates for each mini-batches. Gradient descent is infeasible to implement for the datasets that do not fit in the memory. In contrast, SGD performs too frequent updates, spawning high variance in parameters that cause fluctuation in the loss function. SGD provides the same convergence properties as batch gradient descent if the learning rate periodically decreases through iterations. For our experiments, we used mini-batch gradient descent, which takes the best of two methods. Most of the implementations use SGD

term instead of mini-batch gradient descent. The same tradition will be followed in this study too. Update rule of mini-batch gradient descent is the following:

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta J\left(\theta; x^{(i:i+n)}; y^{(i:i+n)}\right)$$

where θ is the parameters of the network, η is the learning rate or step size, ∇_θ is the gradients of the parameters and $J\left(\theta; x^{(i:i+n)}; y^{(i:i+n)}\right)$ is the loss function for mini-batch i to $i+n$.
There are a couple of challenges in SGD because it doesn't always guarantee good convergence:

- Choosing a proper learning rate is intricate. Small learning rates may take too long to converge, whereas big learning rates may spawn loss function fluctuations and even diverge.
- SGD does not guarantee the global minimum. It can easily be stuck in the local minimum for highly non-convex loss functions that are standard for deep learning tasks.
- Same learning rate is applied to all parameters, but we may want to update the parameter by their frequencies.
- Convergence is strongly dependent on where the initial step starts. Unfortunate initializations may never reach the global minimum.

Momentum

SGD has difficulties finding the direction in valleys because the gradients on these areas will be either zero or very close to zero, so it will slow down and make hesitant progress. These areas are prevalent around the local minimum. Momentum is an idea that dampens the oscillations in the relevant direction. It is accomplished by adding a fraction γ of the update vector of the past time step. This fraction is usually set to 0.9. This term usually leads to faster convergence and speeds up the iterations.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

$$\theta_{t+1} = \theta_t - v_t$$

However, momentum follows the direction of the gradients blindly, **nesterov accelerated gradient** (NAG) is a way of giving our method to intuition by approximating the next position of the parameters with $\theta - \gamma v_{t-1}$, with this we hope to slow down before the hill slopes up. In other words, first, as in the momentum method, we make a big jump in the direction of previous gradients, then we measure the gradients where we end up and make a correction. The new update rule becomes:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J\left(\theta - \gamma v_{t-1}\right)$$

$$\theta_{t+1} = \theta_t - v_t$$

1.4.2 Adaptive Methods

Two main drawbacks of SGD are; tuning the learning rate is complex, and we use the same learning rate for each parameter. Adaptive methods offer solutions to these problems. They use intelligent ways to modify the learning rate that may differ from parameter to parameter, and some of them even remove the need to set the learning rate. However, they are still gradient-based algorithms with some modifications, and they do not always guarantee convergence.

Adagrad

In vanilla SGD and SGD with momentum, we used the same learning rate for each parameter. On the contrary, adagrad adapts the learning rates for each parameter. It performs larger updates for infrequent parameters and smaller updates for frequent parameters. To do this, it updates the learning rate at each time step t for each parameter based on their past gradients.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

With this update rule, the learning rate is modified at each time step. G_t contains the sum of squares of the past gradients for all parameters. g_t is the gradients of all parameters at time step t, and ϵ is the smoothing constant to avoid zero division, and it is usually set to 10^{-8}. G_t is getting larger with each step since we only add positive terms that make the learning rate very small, and the algorithm cannot learn any more in advancing time steps.

Adadelta

Adadelta is an extension of Adagrad, which tries to solve the decreasing learning rate problem and tries to remove the need for tuning the learning rate manually [17]. Instead of using the squares of all past gradients, Adadelta sets a moving window of gradient updates, and by doing so, it continues learning even after many iterations. It does by storing the exponentially decaying average of the squared gradients.

$$E\left[g^2\right]_t = \rho E\left[g^2\right]_{t-1} + (1-\rho)g_t^2$$

$E\left[g^2\right]_t$ is the running average, ρ is the decay constant which is similar to momentum term (it is usually set to around 0.9 like momentum). The demonitor of the update rule of adadelta is very similar to adagrad, only difference is G_t is replaced with $E\left[g^2\right]_t$. The term $\sqrt{E\left[g^2\right]_t + \epsilon}$ can be rephrased as root mean squares of the previous gradients up to time t.

$$\text{RMS}[g]_t = \sqrt{E\left[g^2\right]_t + \epsilon}$$

Where ϵ is a smoothing constant for avoiding any problem in the denominator, by using this term, we can change the update rule of Adagrad to the following:

$$\theta_{t+1} = \theta_t - \frac{\eta}{RMS[g]_t} \odot g_t$$

For clarity, we can rephrase the update rule as follows:

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

where;

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} \odot g_t$$

Authors of [17] pointed out that parameters updates in SGD, momentum and Adagrad doesn't match with the units of the parameters. The units relate the gradients, not the parameters. To overcome this issue they defined exponentially decaying average of parameters instead of gradients.

$$E\left[\Delta\theta^2\right]_t = \rho E\left[\Delta\theta^2\right]_{t-1} + (1-\rho)\Delta\theta_t^2$$

The root means squared error of the parameters is:

$$\text{RMS}[\Delta\theta]_t = \sqrt{E\left[\Delta\theta^2\right]_t + \epsilon}$$

Since $\text{RMS}[\Delta\theta]_t$ is unknown at time step t, it is approximated with previous time step.

$$\theta_{t+1} = \theta_t - \frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$

Normally this final update would remove the need of learning rate but we follow the same approach as in Pytorch implementation [18] so last term is scaled with learning rate which finally yields to update rule of Adadelta:

$$\theta_{t+1} = \theta_t - \eta \frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$

RMSProp

RMSProp is another method that is offered to solve the decreasing learning rate problem of adagrad. Geoffrey Hinton proposed it in his neural networks for machine learning class[1]. It is identical to the first update rule of Adadelta that is:

$$E\left[g^2\right]_t = \rho E\left[g^2\right]_{t-1} + (1-\rho) g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E\left[g^2\right]_t + \epsilon}} \odot g_t$$

Similar to momentum constant, it is suggested to set ρ to 0.9, and ϵ is the smoothing constant similar to previous methods' update rules.

ADAM

Adam is another adaptive method that adjusts the learning rates for each parameter. It also stores an exponentially decaying average of the past gradients and past squared gradients similar to momentum. It combines the best properties of adagrad and RMSProp algorithms.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\, g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\, g_t^2$$

[1] http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

Where m_t is the estimate of the first moment of the gradients and v_t is the estimate of the second moment. However, the authors noticed that with zero initialization, these two terms are biased towards zero. Therefore they proposed bias-corrected forms of these terms to overcome this problem. It is suggested to set defaults values for β_1 and β_2 as 0.9 and 0.999.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Then the update rule is very similar to Adadelta and RMSProp that is:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

With ADAM, we completed the theoretical overview of the methods that are used for the experiments. We started with the simple definition of BP, we explained how the error is propagated in BP and how gradients are calculated mathematically. We also mentioned the drawbacks of BP. Then, we moved to DFA by slightly changing the error propagation mechanism of BP, and we adapted this change to our mathematical foundations. We specified which drawbacks of BP were addressed by DFA, and we also mentioned the limitations of DFA. After that, lazy methods are summarized since they are implemented and used in experiments of previous studies. Finally, optimizers are described in general. We mentioned why they are important and how they might increase the convergence property, and lastly, we presented their update rules and drawbacks.

2
Learning Problems

Contents

The success of neural networks spawned a great interest in the field of learnability of the various models. This involves testing different models on the same problem, usually difficult to learn, and observing the results. It is particularly beneficial to understand the learning dynamics of the models, which helps to find out their limitations. These studies achieved striking success in understanding neural networks.

2.1 Parity Learning Problem

In [3], authors questioned how far neural networks could go beyond the linear models. They did this by focusing on parities that have a complex family of target functions. They demonstrated that this family could be approximated by a two-layer network trained with Adadelta, but not by lazy methods. This study brings an explanation of why neural networks' performance is better than linear methods, and it proves neural networks' learning capacities are beyond lazy methods.

Experiments are performed on the MNIST dataset by imitating the parity problem.

The task is: given the parameter k (defines the number of digits to be stacked together that is chosen uniformly from the dataset), determine if the sum of the digits is odd or even. When $k = 1$, it is a simplified version of the standard MNIST task to find if a digit is even or odd. Experiment results showed that all models, including the lazy ones, reached a similar performance in $k = 1$ case where the neural network slightly outperformed others. On the other hand, the problem becomes more difficult for the case $k = 3$ because models need to compute the parity of the digits' sum. In this case, there is a drastic gap between the neural network and lazy methods. Because the predictions of lazy methods did not go beyond the random guess.

Since our goal is comparing DFA and BP on this particular problem, reproducing the results from [3] is unavoidable. The same configurations are used with minor differences and, they are implemented in Pytorch [18]. The network has only a hidden layer with 512 neurons. For the last layer, sigmoid is used as a non-linear activation function. For the hidden layer, reLU is used. BCE is preferred as a loss function, and 10^{-3} is set to weight decay. In addition to previous settings, we have also used SGD to observe how much Adadelta improves. For the case $k = 3$, we performed a simple hyper-parameter tuning process to get a decent learning rate for each method. Same learning rate values are used for the case $k = 1$. The hyperparameter tuning process is the following. First, we define the parameter space, later we run with all different learning rates, and we compare these runs by the average of test accuracy of the last ten epochs, and we choose the highest one. It is also crucial to mention that, at each epoch, train data is recreated to boost the available data for the models. The same is also performed for test data to have an unbiased estimation of test accuracy. It increases the available data because the creation of the data is stochastic. The process is the following: random images are sampled uniformly from the available dataset, then according to given parameter k, these random images are horizontally stacked. Hence we have different dataset at each epoch that helps networks to learn and perform better. Finally, these images are normalized before training the networks, which is necessary for deep learning tasks.

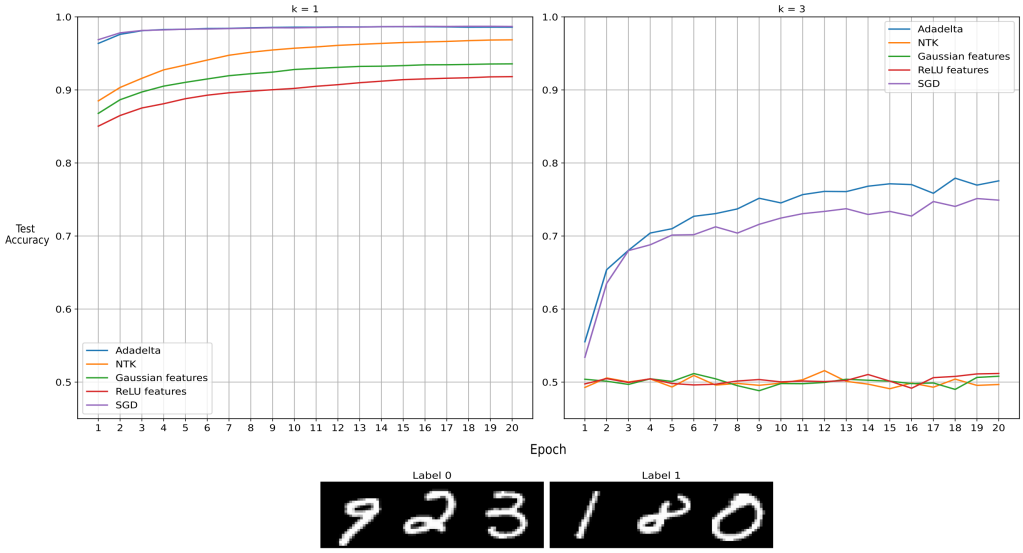

Figure 2.1: Reproduced MNIST Parity Experiment [3]

The reproduced result can be observed in figure 2.1. Similar to results from [3], all the methods succeed learning for $k = 1$ case. However, adadelta and SGD slightly outperformed lazy methods in this setting. In the case of $k = 3$, adadelta and SGD almost reach 80%, but the performance of lazy methods doesn't go beyond a random guess. After having the concrete picture from the previous study, it is intriguing to see how DFA would perform with SGD and adaptive methods on this particular problem. We will investigate it in chapter 3 with other experiments.

2.2 Random Data Problem

TODO: Explain why we need another problem (To test DFA capabilities on a different and challenging problem which has similar behaviour as parity - not learnable by some models-). Write how this data is generated.

3
Experiments

Contents

After having the previous study results, we can continue to test DFA on the parity learning problem. Train phase and the hyperparameter tuning process are explained in chapter 2. These processes are the same for the following experiments. For all experiments, scratch implementations are used with minimal Pytorch functionalities. They are performed three times and plotted with their mean and standard deviation or with their confidence interval.

The same architectures are used as the previous study for BP and DFA, meaning that we have only a hidden layer with 512 neurons, and reLU is used as a non-linear function for the hidden layer. BCE is chosen as a loss function and, sigmoid is preferred for the non-linearity of the last layer. Networks are trained for twenty epochs unless others are specified, and at each epoch, train and test datasets are recreated as it is explained in chapter 2. Weights of the networks are initialized uniformly with $\frac{1}{\sqrt{inputdim}}$ as in default Pytorch weight initialization. Moreover,

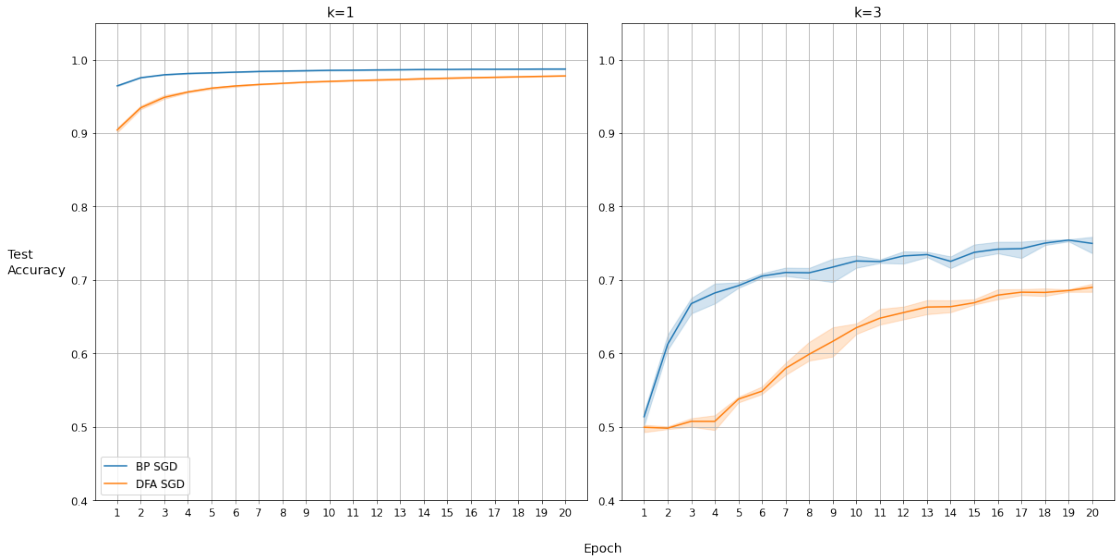

Figure 3.1: BP and DFA on Parity Problem with SGD

the random matrix B is initialized with the same way to have similar behaviors as the weight matrix unless other specified.

3.1 Parity Learning Experiments

Since the experiment details are clear now, it is time to test DFA on parity learning problem with BP by using SGD.

We can observe the results in 3.1 with 95% confidence interval. In the case $k = 1$, although DFA outperforms the lazy methods, it is behind the BP. In the case $k = 3$, it is obvious that DFA performs much better than lazy methods. However, the gap between BP and DFA is a bit higher than $k = 1$ case. It seems like there is a limit for DFA to reach with SGD that is %70. The reason is that DFA has an additional task to accomplish, which is aligning with BP's teaching signals. In other words, the network loses time while trying to make teaching signals useful. This delays the convergence and causes performance lag. We can see that during the first iterations, DFA does not converge fast enough to catch up with BP, and it always stays behind the BP.

The thrilling question is, is there a performance limit for DFA to reach, and can we get similar performance as BP by making some changes? For answering the first

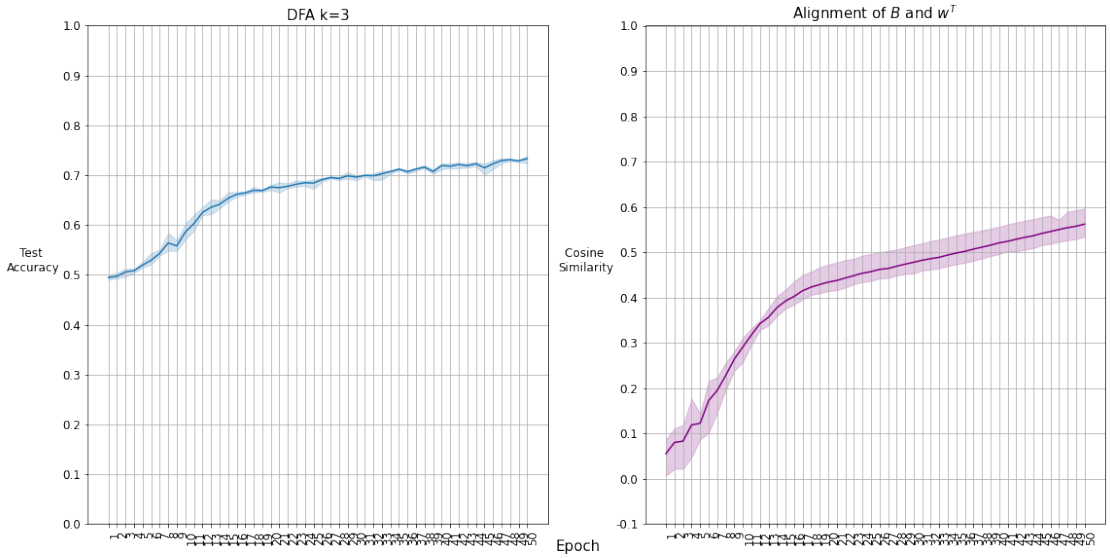

Figure 3.2: DFA on Parity Problem with Alignment

question, it is better to run DFA for more epochs to see if it can reach a similar performance as BP. Because with longer training, DFA will have time to align and converge. It would be convenient to test DFA with different random matrices to observe any improvement for the second question. Because it is clear that learning in DFA is strongly dependent on random matrix. Besides, it is interesting to test if DFA can learn with a different types of random matrices. During the learning rate phase of DFA, we noticed that it is susceptible to the learning rate. Having a small learning rate caused no convergence, and larger learning rates showed over-fitting within specified epoch numbers. Therefore we can use adaptive methods to have better convergence properties both in BP and DFA. These methods are specifically good at adjusting the learning rate, which is more difficult to tune for DFA than BP.

We trained DFA for 50 epochs with a tuned learning rate to observe if it can reach a similar performance as BP. At the same time, alignment between the random matrix and the transpose of the weight matrix is plotted. This alignment is measured by using the cosine similarity.

From figure 3.2, we can see that DFA can reach a similar performance as BP trained with SGD. This result approves our comments about the additional task DFA has

and why it takes longer to achieve the same performance. On the right side of the plot, we can examine the alignment between the random matrix and the transpose of the weight matrix. At the beginning of training, the similarity is low. However, with advancing steps, we can see that alignment becomes higher, similar to the performance. It shows that the network aligns with the BP's teaching signals. In other words, the network learns how to learn by using the random matrix. After having a similar performance from DFA with SGD, it is intriguing to test if we can achieve similar performance within the same epoch number. For this purpose, the first improvement attempt will be related to random matrices. Using different random matrices may influence the performance of DFA. Some of them might align better with BP's teaching signals. On the other hand, it is interesting to observe if we can learn with any random matrix.

Among uniform random matrix, three different random matrices are tested. They are initialized as the following: **standard uniform** is default Pytorch initialization that is uniformly distributed from 0 to 1. **Gaussian** is initialized normally with μ and σ are equal to each other that is $\frac{1}{\sqrt{inputdim}}$. Lastly, **standard gaussian** is initialized with $\mu = 0$ and $\sigma = 1$.

From figure 3.3, we can observe that DFA can learn with any random matrices. However, it is essential to specify that learning rates for each random matrix are tuned and drastically different. Apart from standard uniform, the rest of the random matrices achieved similar performances, but they are still behind the BP. However, thanks to these results, we can see that DFA is highly sensitive to the learning rate. Because during the tuning phase, small learning rates did not converge within the specified epoch number. On the other hand, high learning rates demonstrated overfitting. Therefore, since adaptive methods have better convergence properties, they may increase the performance of DFA as they did in BP. For the rest of the DFA experiments, the random matrix is uniformly initialized since there is no significant improvement with other initializations.

Following the previous deduction, various adaptive methods are tested on the parity learning problem for BP and DFA. Their learning rates are tuned, as it is explained

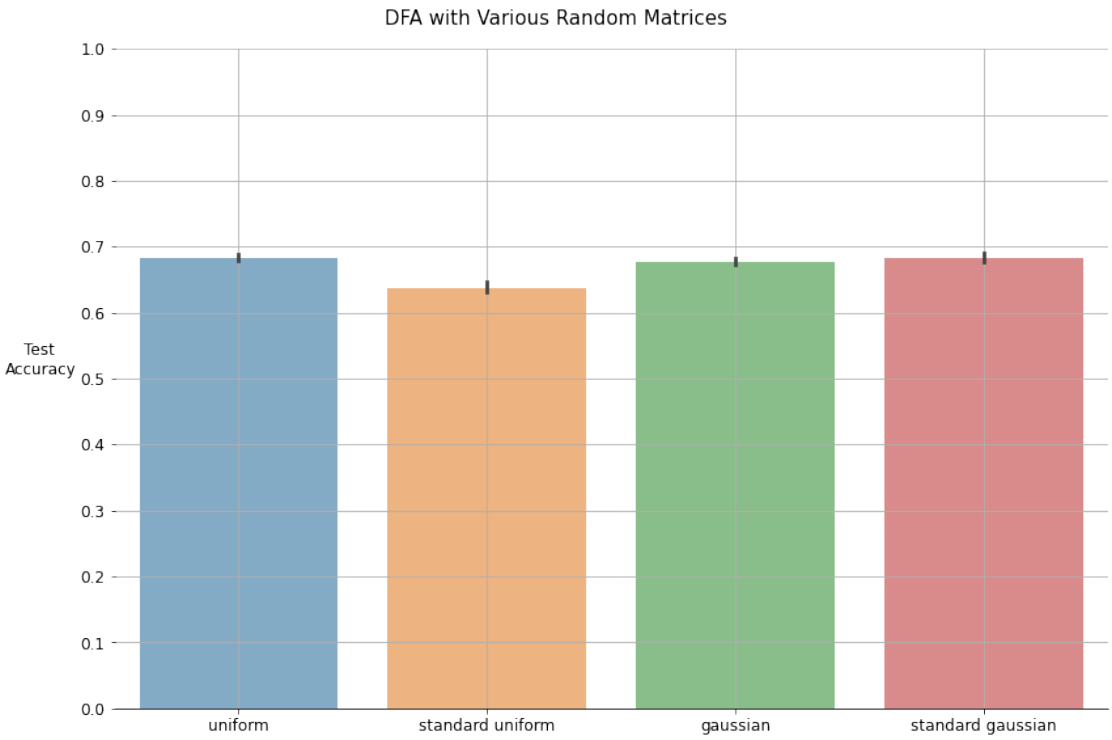

Figure 3.3: DFA on Parity Problem with Various Random Matrices

in the previous chapter. For the experiments, they are run three times, and their final test accuracies are plotted. The results are presented in figure **??**.

As expected, adaptive methods improve the final test accuracy significantly for both BP and DFA. On average, DFA is still behind the BP, but with RMSProp and Adadelta, the gap is much smaller than with plain SGD, sometimes DFA's final test accuracy even exceeds BP. In other words, we can say that some adaptive methods help DFA more than BP. However, we should not ignore that DFA has larger fluctuations for the final test accuracy than BP. With the last experiment, we could close the gap between BP and DFA on the parity learning problem thanks to adaptive methods.

3.2 Random Data Experiments

TODO: Report the random data experiments under this subtitle.

Conclusion

TODO: Start summarizing all the process theoretical foundations, problems and wrap up the results of the experiments. Write some possible future studies. Write the take-home messages.

Appendices

A

Backpropagation with Binary Cross-Entropy

Let's consider a simple binary classification task. It is common to use a network with a single logistic output with the binary cross-entropy (BCE) loss function and for the sake of simplicity, let's assume that there is only one hidden layer.

$$BCE = -\sum_{i=1}^{nout} \left(y_i \log\left(\hat{y}_i\right) + \left(1 - y_i\right) \log\left(1 - \hat{y}_i\right)\right)$$

Where y is the ground truth and $\hat{y}$ is the output of the network. After having the loss function, let's continue with the forward pass.

$$a_k = h_{k-1} w_k + b_k$$

$$h_k = f(a_k)$$

Where, w_k is the weight, b_k is the bias term, h_k is the output of the layer (which means that $h_0 = X$ and $h_2 = \hat{y}$) and f is the non linear function. Please note that for last layer logistic function is used whereas for hidden layer reLU is used as non linear functions.

We can compute the derivative of the weights by using the chain rule.

$$\frac{\partial BCE}{\partial w_2} = \frac{\partial BCE}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_2} \frac{\partial a_2}{\partial w_2}$$

A. Backpropagation with Binary Cross-Entropy

Computing each factor in the term, we have:

$$\begin{aligned}\frac{\partial BCE}{\partial \hat{y}} &= \frac{-y}{\hat{y}} + \frac{1-y}{1-\hat{y}} \\ &= \frac{\hat{y}-y}{\hat{y}\,(1-\hat{y})} \\ \frac{\partial \hat{y}}{\partial a_2} &= \hat{y}\,(1-\hat{y}) \\ \frac{\partial a_2}{\partial w_2} &= h_1^T\end{aligned}$$

This gives us:

$$\frac{\partial BCE}{\partial w_2} = h_1^T\,(\hat{y}-y)$$

We can calculate the derivative of the w_1 concerning loss function as the following:

$$\frac{\partial BCE}{\partial w_1} = \frac{\partial BCE}{\partial h_1}\frac{\partial h_1}{\partial a_1}\frac{\partial a_1}{\partial w_1}$$

Compute each factor in the term again, we have:

$$\begin{aligned}\frac{\partial BCE}{\partial h_1} &= \frac{\partial BCE}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial a_2}\frac{\partial a_2}{\partial h_1} \\ &= (\hat{y}-y)\,w_2^T \\ \frac{\partial h_1}{\partial a_1} &= f'(a_1) \\ \frac{\partial a_1}{\partial h_1} &= X^T\end{aligned}$$

This gives us:

$$\frac{\partial BCE}{\partial w_1} = (X)^T\,(\hat{y}-y)\left(w_2^T\right) \odot f'(a_1)$$

Where $\odot$ is element-wise multiplication, similarly, bias terms can be calculated by following:

$$\begin{aligned}\frac{\partial BCE}{\partial b_2} &= \frac{\partial BCE}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial a_2}\frac{\partial a_2}{\partial b_2} \\ &= (\hat{y}-y) \\ \frac{\partial BCE}{\partial b_1} &= \frac{\partial BCE}{\partial h_1}\frac{\partial h_1}{\partial a_1}\frac{\partial a_1}{\partial b_1} \\ &= (\hat{y}-y)\left(w_2^T\right) \odot f'(a_1)\end{aligned}$$

After having all these results, we can update the parameters (weights and biases) using gradient descent and its variants.

B

Hidden Representation of Digits in Parity Problem

One of the interesting questions is: does the network learn the digits individually when we train it and make the necessary processes (summation, division), or does it memorize the way data is fed? We can answer this question by observing the hidden representation. If the network learns the digits well, we need to observe a good separation like the MNIST task. For this purpose, hidden representations of the networks trained with BP and DFA are plotted in two-dimensional space by using t-SNE [8] implementation of sklearn [19].

The hidden representation of a single image in $k = 3$ case is obtained in the following way; we know that after we flatten the images, particular parts of each image are multiplied by corresponding parts of the weight matrix. Getting these parts and performing multiplication for each digit will give us the hidden representation of the individual digit trained in $k = 3$ case. Process is visualized in B.1. Since the t-SNE is a computationally expensive process, only 7500 random samples are plotted from the dataset.

With this matrix multiplication, we have the hidden representation of each image. The rest is visualizing them by using the t-SNE. In figure B.2 we can observe the

B. Hidden Representation of Digits in Parity Problem

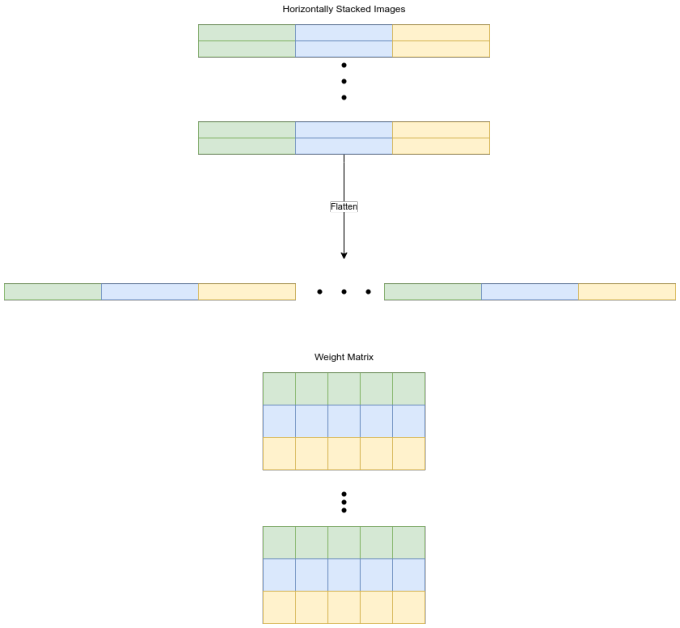

Figure B.1: Process of Hidden Representation

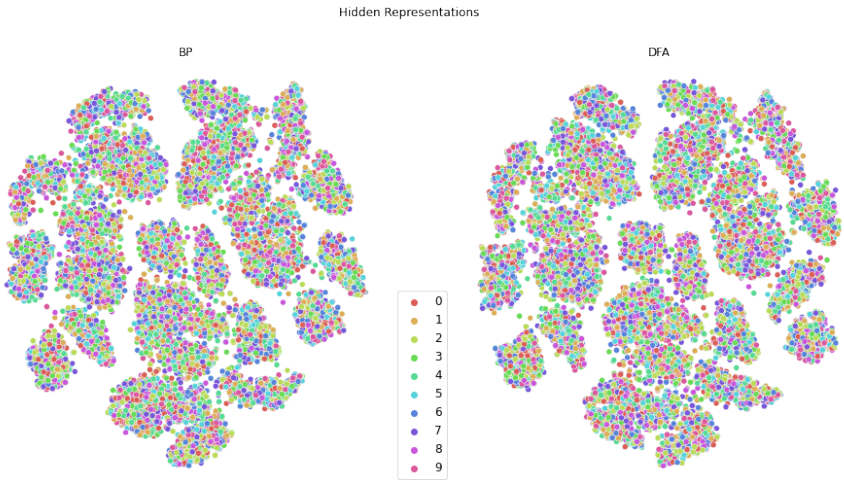

Figure B.2: Hidden Representation of Digits in BP and DFA

results in two dimensions and say that the networks don't capture the information about the digits individually. They learn by memorizing the samples, not the digits.

C
Reproducibility

TODO: Write down the library versions, refer to the Github repository for the experiments, give information about the hardware. Explain the stochastic behavior of data.

References

[1] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning Representations by Back-propagating Errors". In: *Nature* 323.6088 (1986), pp. 533–536. DOI: 10.1038/323533a0. URL: http://www.nature.com/articles/323533a0.

[2] Arild Nøkland. *Direct Feedback Alignment Provides Learning in Deep Neural Networks*. 2016. arXiv: 1609.01596 [stat.ML].

[3] Amit Daniely and Eran Malach. "Learning Parities with Neural Networks". In: *CoRR* abs/2002.07400 (2020). arXiv: 2002.07400. URL: https://arxiv.org/abs/2002.07400.

[4] Yoshua Bengio et al. *Towards Biologically Plausible Deep Learning*. 2016. arXiv: 1502.04156 [cs.LG].

[5] Dong-Hyun Lee et al. *Difference Target Propagation*. 2015. arXiv: 1412.7525 [cs.LG].

[6] Wan-Duo Kurt Ma, J. P. Lewis, and W. Bastiaan Kleijn. *The HSIC Bottleneck: Deep Learning without Back-Propagation*. 2019. arXiv: 1908.01580 [cs.LG].

[7] Timothy P. Lillicrap et al. *Random feedback weights support learning in deep neural networks*. 2014. arXiv: 1411.0247 [q-bio.NC].

[8] Laurens van der Maaten and Geoffrey Hinton. "Visualizing Data using t-SNE". In: *Journal of Machine Learning Research* 9 (2008), pp. 2579–2605. URL: http://www.jmlr.org/papers/v9/vandermaaten08a.html.

[9] Julien Launay et al. *Direct Feedback Alignment Scales to Modern Deep Learning Tasks and Architectures*. 2020. arXiv: 2006.12878 [stat.ML].

[10] Maria Refinetti et al. *Align, then memorise: the dynamics of learning with feedback alignment*. 2021. arXiv: 2011.12428 [stat.ML].

[11] Julien Launay, Iacopo Poli, and Florent Krzakala. *Principled Training of Neural Networks with Direct Feedback Alignment*. 2019. arXiv: 1906.04554 [stat.ML].

[12] Lenaic Chizat, Edouard Oyallon, and Francis Bach. *On Lazy Training in Differentiable Programming*. 2020. arXiv: 1812.07956 [math.OC].

[13] Jaehoon Lee et al. *Deep Neural Networks as Gaussian Processes*. 2018. arXiv: 1711.00165 [stat.ML].

[14] Pedro Domingos. *Every Model Learned by Gradient Descent Is Approximately a Kernel Machine*. 2020. arXiv: 2012.00152 [cs.LG].

[15] Arthur Jacot, Franck Gabriel, and Clément Hongler. "Neural Tangent Kernel: Convergence and Generalization in Neural Networks". In: *CoRR* abs/1806.07572 (2018). arXiv: 1806.07572. URL: http://arxiv.org/abs/1806.07572.

[16] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *CoRR* abs/1609.04747 (2016). arXiv: 1609.04747. URL: http://arxiv.org/abs/1609.04747.

[17] Matthew D. Zeiler. *ADADELTA: An Adaptive Learning Rate Method*. 2012. arXiv: 1212.5701 [cs.LG].

[18] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[19] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.