Blocked Direct Feedback Alignment: Exploring the Benefits of Direct Feedback Alignment

Mateo Espinosa Zarlenga*
Department of Computer Science
Cornell University
Ithaca, NY 14850
me326 at cornell.edu

Eyvind Niklasson*
Department of Computer Science
Cornell University
Ithaca, NY 14850
een7 at cornell.edu

Abstract

Backpropagation is undoubtedly the preferred method for training deep feed-forward neural networks. While this method has proven its effectiveness on applications ranging over a myriad of different fields, it has some well-known drawbacks. Moreover, this algorithm is arguably far from being biologically plausible, which makes it very unattractive as a crucial step of any attempt for an accurate model of our brain. Alternatives like feedback alignment and direct feedback alignment has then been proposed recently as possible methods that are more biologically plausible than backpropagation while also correcting some of the know drawbacks of this algorithm. For this project, we explore the uses of this last method, direct feedback alignment (DFA), by looking at variants of the same that could lead to improvements in both training convergence times and testing-time accuracies. We present two main variants: Feedback Propagation (FP) and Blocked Direct Feedback Alignment (BDFA). These variants of DFA attempt to find some sort of equilibrium between DFA and backpropagation that takes advantage of the benefits in both methods. In our experiments we manage to empirically show that BDFA outperforms both DFA and backpropagation in terms of convergence time and testing performance when used to train very deep neural networks with fully connected layers on MNIST and notMNIST.

1 Introduction

For our project, we decided to focus on the exploration of training algorithms for neural networks rather than the design of architectures used for applications. While we do believe that designing network architectures can be extremely interesting, and definitely important for crucial challenges, we wanted to get some better insight of the algorithms that really make all of these networks possible. Rather than taking these algorithms as black boxes, we wanted to explore the different approaches to learning that are out there in the literature and focus on the analysis and extension of a subgroup of these algorithms. Hence we think of the main objectives of our project as :

1. Get a closer and more extensive knowledge of Deep Learning in practice by reading about and implementing by hand several different training algorithms.
2. Get some exposure to designing and running different experiments for testing different algorithms and methods we intend to compare.
3. Explore the effectiveness of DFA on simple datasets in order to get more intuition behind these methods.

*All authors contributed equally.

4. Design, evaluate, and analyze an extension of a specific training algorithm that could hopefully outperform the original method.

In this paper we will first give some motivation behind both the importance and applicability of a deeper study of this field. We will then follow by describing and developing some of the ideas we based our work on for this project. Following the background, we introduce an extensive experimentation section where we describe the experiments we performed as well as the set up used and include a brief discussion of the results obtained. We finally end our paper with a short discussion on possible work that could be worth exploring in the future. One thing to keep in mind at all times is that all of the code developed for this project can be found in our repository where one could recreate the experiments performed in this paper.

1.1 Motivation

In this section we will introduce and discuss the need of re-exploring training methods in the hope of developing an algorithm that outperforms backpropagation in most scenarios, at least in practice. While this algorithm seems to work well in certain scenarios, the literature keeps limiting the complexity of the architectures used in order for these to be able to be trained specifically with backpropagatiopn. This clearly imposes a cap in our ability to explore possibly better architectures. Given that deep neural networks have been shown to perform outstandingly well in tasks from image recognition ([3]) to low-level language understanding ([13]), the idea of developing architectures independently of the training method used is certainty very promising. This is the reason why better and faster training methods could open the door for tackling more complicated learning challenges and maybe start the development of a model that could capture the human brain in a more accurate way. This is the main reason why we believe investigating training methods is something worth our time as much as developing new architectures to handle specific tasks.

Before getting into discussing the previous literature on alternatives to backpropagation, we will first make an thorough recap of fully connected neural networks and backpropagation to establish some useful notation that will become very useful later when we discuss DFA and BDFA.

2 Background

2.1 Fully Connected Neural Networks

Fully connected neural networks are a feed-forward neural network that attempt to closely learn some mapping between some datapoints and labels using a set of known mappings (see [3] for a very detailed description of these models). This is refereed to as a fully supervised setting. Formally speaking, a fully connected neural network attempts to learn a mapping $\Phi : \mathcal{X} \to \mathbb{L}$ between a set of datapoints $\mathcal{X} = \{x_1, x_2, ...\}$ (possibly infinite) and a set of labels $\mathcal{L} = \{l_1, l_2, ...\}$ (also possibly infinite in regression case) where we would like $\Phi(\cdot)$ to map x_i to l_i as close as possible, as defined by some similarity metric between the labels. For our specific project, we focus on the case when we are trying to classify some datapoints into a set of C classes $\{c_1, c_2, ..., c_C\}$ so every label is a one-hot-encoding of one of these classes. This simply means that if datapoint x_i is classified as class c_j then l_i is a vector of dimension C where every entry is zero except for entry j which is set to 1.

In order for a neural network to learn such a mapping, the model is provided a set of n correct mappings $\{(x_1, l_1), (x_2, l_2), ..., (x_n, l_n)\}$, called the training set, that serve as a basis for the model to understand how would such a mapping $\Phi(\cdot)$ look like.

Specifically, a fully connected neural network with n layers has some weight matrices $W_1, W_2, ..., W_n$ and some bias-terms $b_1, b_2, ..., b_n$. We think of each layer i of the network as a vector of inputs that somehow gets mapped to the following layer using a combination of the weight matrix W_i and bias b_i of this layer. Given an input datapoint x, the output $\hat{y}$ of the function $\Phi(x)$ modeled by this network is computed as follows: we first define $h_0 \stackrel{\text{def}}{=} x$. Now, for each layer in $\{1, 2, ..., n-1\}$ we compute the activation of layer i, refereed to as a_i, given by the following expression

$$a_i \stackrel{\text{def}}{=} W_i h_{i-1} + b_i$$

This variable is refeered to as the activation of layer i. We then "pass" this activation through a non-linearity (i.e. any non linear function) $f(\cdot)$ to get the final output of layer i, and hence the input of the next layer $(i+1)$, defined by the variable h_i given by

$$h_i \stackrel{\text{def}}{=} f(a_i)$$

Common non-linearities used are the sigmoid function, tanh, and today more commonly the ReLU activation function ([5]). Now we make things a little different for the last layer (layer n) where we define a_y to be the activation of this layer as

$$a_y = W_n h_{n-1} + b_n$$

However we require this activation vector to have a dimension of C. With this activation vector, we compute the output h_y of this layer using use a "softmax function" that turns all the values in a_y into probabilities that indicate how likely the given input is from being one class over another. Formally speaking, we calculate this probability vector h_y (where the ith input of h_y is the probability that the given input is classified as class c_i) as follows:

$$\left[h_y\right]_i \stackrel{\text{def}}{=} \frac{\exp\left(\left[a_y\right]_i\right)}{\sum_j \exp\left(\sum_j \left[a_y\right]_j\right)}$$

Where we use $\left[x\right]_i$ to indicate the ith entry in vector x. Given this probability distribution, the network will output the class whose corresponding entry in h_y is the highest. In other words, if the output of the network is $\hat{y}$ (i.e. the value of $\Phi(x)$), then this value is given by

$$\hat{y} \stackrel{\text{def}}{=} c_{\arg\max_i \left[h_y\right]_i}$$

Intuitively, we can think of every layer in the network as some sort of specialized filter that extracts crucial information from the data passed from the previous layer that could be of importance for the correct classification of the given datapoint. Because of the nature of how every layer in this model gets fed by the output of the previous layer in the process of computing the output for a given datapoint x, these models are referred to as feed-forward networks. Furthermore, the computation of the activations of the output layer given an input is usually refereed to as the forward pass.

The last element we need to completely define a fully connected neural network is some sort of metric that can tell us how good this the function we generate with this model is compared to what we know from the data. For this we use something referred to as a loss function $\mathcal{L}(\cdot)$ that takes the output label of a given example x and the correct label of this datapoint and returns a real number whose magnitude tells us how bad our solution is. Ideally low values are very good (i.e. the model is correctly classifying a lot of points) while high values correspond to bad models. When using softmax, a commonly used loss function if what is referred to as "cross-entropy". For more on this topic, please refer to [3].

2.2 Backpropagation

Now that we have defined the model used for fully connected neural nets, we have to worry about how we could possibly learn the parameters of our model (the weights and the biases) to get an good approximation of the real mapping between the datapoints and the labels. Here is where Backpropagation comes in play.

Backpropagation (BP) is a learning algorithm firstly introduced in the early 70's, yet not quite popular until the late 80's ([7]), where the each layer in a feed-forward neural network receives some "blame" (i.e. how much they affected the resulting loss) through a signal that is continuously passed from the last layer of the network to the first layer of the network. Generally, this algorithm is used to train networks in a fully supervised scenario as defined above. The algorithm, as complicated as it may look later, is simply a continuous application of the chain rule to compute the gradient of the loss with respect to the different weights and then use this value to do a gradient descent update on the parameters.

Informally all layers are assigned some sort of "blame" (i.e. gradient of the loss with respect to their weights) so that the layer can upgrade its respective weight matrix by "shifting" these

weights an amount proportional to the blame assigned to this neuron in the opposite direction of highest possible future blame (i.e. a gradient decent step). Intuitively, all weights are being updated in order to locally minimize the error they just saw given the training example just used.

Formally speaking, if $h_0 \stackrel{\text{def}}{=} \mathbf{x}$ is a training example with true label y (in a one-hot encoding) we first compute a forward pass for this given input to get an output $\hat{y}$. Once this is done, we save all the activations of all layers and compute the loss $\mathcal{L}$ given the true label y and the returned label $\hat{y}$. The crucial step in backpropagation comes from the computation of something called "feedback vectors" (which we will also call "deltas" sometimes) that can be directly used to compute the gradients of the loss with respect to each weight matrix. These vectors are computed starting from the output layer moving toward the input layer by propagating the feedback vector of a layer to the previous layer. This is the reason why the name of this algorithm is backpropagation and this update step is usually called the backwards pass.

Formally we proceed as follows: we first let e be the output gradient defined as

$$e \stackrel{\text{def}}{=} \delta_{a_y} = \frac{\partial \mathcal{L}}{\partial a_y} \tag{1}$$

This function can be computed directly from the nature of the loss function. For example, when the loss function used is cross entropy, this value corresponds to $\hat{y} - y$ where y is the true label of the given input. With this, we propagate this delta all the way to the first layer by computing the gradient at each layer $i \in \{1, 2, ..., n-1\}$ as follows:

$$\delta_{a_i} \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial a_i} \left(W_{i+1}^T \delta_{i+1}\right) \odot \left(f'(a_i)\right) \tag{2}$$

Where $\odot$ represents entry-wise multiplication between vectors, $f'(\cdot)$ is the derivative of the activation function, and, for simplicity of notation, we let $\delta_{a_{n+1}}$ be equal to δ_{a_y}. With these gradients we can compute the gradient of the loss with respect to the weights through a simple chain rule as follows:

$$\frac{\partial \mathcal{L}}{\partial W_i} = \left(\frac{\partial \mathcal{L}}{\partial a_i}\right)\left(\frac{\partial a_i}{\partial W_i}\right) = \delta_{a_i} h_{i-1}^T$$

Similarly we can compute the gradient of the loss with respect to the bias of layer i as:

$$\frac{\partial \mathcal{L}}{\partial b_i} = \left(\frac{\partial \mathcal{L}}{\partial a_i}\right)\left(\frac{\partial a_i}{\partial b_i}\right) = \delta_{a_i}$$

A graphical representation of both the forward and backward pass can be seen in figure 1.

With this in mind we can then update the weights and biases of any given layer in the direction opposite to the computed gradient for those weights using some sort of multiplying factor λ (refereed to as the learning rate) that modulates how big of an update we would like to take. Note then how we are propagating these "deltas" from one layer to the previous one so that every layer can, at the end of this process, compute their respective gradients and update their weights accordingly. Once we have updated all the parameters accordingly, we repeat this process for the next batch input in our training set (or batch of inputs if we are using mini-batches) to further improve the performance of our network.

2.3 Drawbacks of Backpropagation

While backpropagation has been shown to work amazingly well for even complex architectures with all sorts of complex architectures ranging from convolutional networks ([10] to LSTM networks ([4]), there are a few important drawbacks of this algorithm that are worth knowing and mentioning.

The first drawback we will discuss is something that is usually referred to as the "vanishing gradient" problem in the literature ([8]). Note that because backpropagation is a simply a sequential use of the chain rule in order to compute the gradients of all layers, it can be the case that the gradients are not big enough and hence this constant multiplication of small terms can lead to a gradient that is

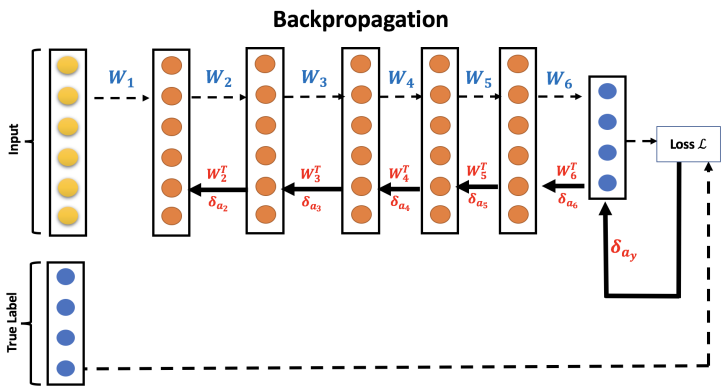

Figure 1: Diagram of how backpropagation works in a network with 5 hidden layers. Layers are the vertical columns where the i-th vertical column represents the $(i-1)$-th layer. The first column is the input and the last column is the output (the n-th layer). The dashed line represents the forward pass and the solid line represents the backwards pass. On top of every line (in red color for the backpass and blue for the forward pass) we indicate what parameters are required for the pass to be executed. Note how on the backwards pass we always need to propagate the deltas and the transpose of the weight matrix to the previous layer. This can be understood by looking at equation (2)

.

pretty much meaningless for the first layers of the network. This means that the later layers will be updated greatly while the first few layers will be barely updated due to the fact that when the gradient reaches those layer, it is already too small. This effect gets only worse as the depth of the network increases. This is a very important issue as we have seen empirically that the more layers we can add to a network the more expressive our model becomes ([3]). Intuitively, you can think of this problem as a game of "Chinese whispers": you have some information that is being transported from the last layer to the first layers one layer at a time. This information can and will get corrupted by some errors that are being introduced as it passes each layer. This means that the more layers this information "travels" the more corrupted it becomes and thus the less reliable it can be for the first few layers to "trust" this information enough to make a big update to their weights.

While several solutions have been proposed to somehow go around this problem to train very deep networks (see [[9]], [[15]], [[6]] for some great examples of architectures), there architecture seem to be just ways to circumvent an inherent issue to a training method that is widely use. While this does work in practice, in fact it works pretty well, it is still cumbersome that we tend to find solutions around this problem instead of tackling the problem directly by finding a training algorithm that does not have this issue. This, as we will see later, will be one of the inspirations for Direct Feedback Alignment (DFA).

Another issue that is currently seen with backpropagation is its inherent intractability. Unfortunately backpropagation cannot be fully parallelized in practice. While it is true that it can be parallelized at a batch level as mentioned in [3], it still has a significant running time until we seem to reach some sort of convergence. Even though GPUs and TPUs may seem to make this better in practice, it would be ideal if we could find some sort of algorithm that can be fully parallelized allowing us to train very deep networks in multiple machines at the same time. This is something that we will surely discuss later.

A final problem we want to address with backpropagation is its biological implausibility. While it is well known that feed-forward neural networks are far from being an accurate model of the brain, it could still be viewed as a primitive attempt to reach a model that in fact resembles the way we learn in a more accurate matter. One of the big issues with trying achieve this is that bakcpropagation has a lot of features that are hard to justify in any biologically plausible model ([[12]]). In this paper we will focus on tackling two of these implausibilities: (1) backpropagation seems to assume that

neurons from one layer to another are capable of coordinate the computation of the transpose of the weights of the following matrix during training. This means that not only neurons are fully aware of the axon's weights used by the neurons in the following layer but neurons in one layer can also choreographically find a symmetric matrix to the weights of the following layer. As discussed in [[12]], this is believed to be far from possible.

The second big implausibility is the fact that (2) backpropagation assumes that the feedback path, the information path used to update the weights, is exactly the same as the forward pass. This implies that connection between neurons are reciprocal (i.e. information can flow both ways) which, even though some reciprocal connections have been observed in neurons, is highly unlikely to be the case ([12]). This means that an accurate model of the brain should not always assume reciprocal connections between layers as backpropagation does.

3 Previous Work and New Ideaas

In this section we will go over some previous work that has been made as an attempt to mainly solve the biological implausibilities that were mentioned in the previous sections, and present some novel ideas based on Direct Feedback Alignment.

3.1 Feedback Alignment

Feedback Alignment (FA) was first suggested as a modification of the BP algorithm that does not require symmetric weights in order to train a fully connected neural network. In order to achieve this, this model assumes that every layer i has both a feed-forward matrix W_i as in the normal case, but now we have a **fixed** "backwards" matrix B_i that is used during training instead of the transpose of the weight matrix of the following layer. This matrix is a random matrix (usually a sampled from a uniform distribution in $[0, 1]$) and is never updated at any time during training or testing. This matrix is uniquely used during training as a substitute of the transpose of the following layer's weight matrix.

Formally, FA proceeds very similarly to BP but now we compute the feedback vectors (deltas) as follows: for the last layer the delta is still the same as in BP:

$$e \stackrel{\text{def}}{=} \delta_{a_y} = \frac{\partial \mathcal{L}}{\partial a_y} \tag{3}$$

Now, for every hidden layer $i \in \{1, 2, ..., n-1\}$, the feedback vector δ_{a_i} is computed as follows:

$$\delta_{a_i} \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial a_i} \big(B_i \delta_{i+1}\big) \odot \big(f'(a_i)\big) \tag{4}$$

A graphical representation of this method can be seen in figure 2.

Lillicrap et. al. managed to show that this method in fact works amazingly well when training a fully connected neural network ([[12]]) . In their experiments, it seems like FA converges faster than BP while achieving very similar testing errors. Furthermore, by looking at the weights learned by FA during training, Lillicrap et. al. showed that as the number of training steps increase the weights W_i will start to align to the transpose of the weights B_{i+1}. This means that after some training steps, these algorithm will "learn to learn" as backpropagation does. This surprising result shows that there is clearly some unnecessary things in BP that we can and should get rid of in order to explore potentially better learning algorithms.

3.2 Direct Feedback Alignment

DFA was proposed by Arild Nokland in his paper [14], and presents a variant of Feedback Alignment where each layer in the network only depends on the error in the last layer - namely the update step

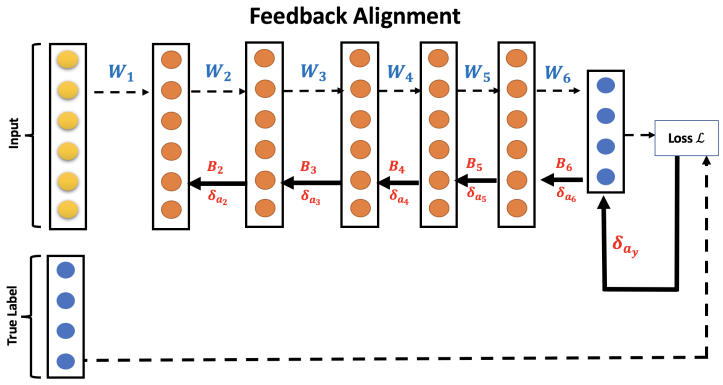

Figure 2: Diagram of how Feedback Alignment (FA) works in a network with 5 hidden layers. The notation used in this figure is the same one as in figure 1. Note how, differently from figure 1, we are not required to pass the transpose of the weight matrix during the backwards pass.

for each layer is computed using the layer's own activation function derivative and the "delta" from the last layer. The intuition behind this seemingly crazy idea is that this model does not assume that feedback connections are not the same as forward connection (i.e. connections are not reciprocal). This is extremely counterintuitive as this means that the update of the layer's weight matrix not only does not depend on the matrix itself, but also does not depend on the "effect" of the layer on the next layer, but rather only on the effect of the last layer on the output. However, as shown in experiments in [14] as well as our own experiments, DFA quickly and efficiently converges to minimize loss in a simple fully connected network.

Formally, DFA computes the feedback vectors (deltas) as follows: for the last layer the delta is still the same as in BP

$$e \stackrel{\text{def}}{=} \delta_{a_y} = \frac{\partial \mathcal{L}}{\partial a_y} \tag{5}$$

Now, for every hidden layer $i \in \{1, 2, ..., n-1\}$, the feedback vector δ_{a_i} is computed as follows:

$$\delta_{a_i} \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial a_i}\left(B_i e\right) \odot \left(f'(a_i)\right) \tag{6}$$

As we can see in here, the update rule only depends on the delta of the last layer and the feedback matrix as used in FA. The amazing thing about this idea is that it can be easily parallelized: once computed e, you can update all the labels in parallel in one step! This makes this training method significantly faster than BP and FA. This is one of the reasons why we wanted to explore this method. Ideally, we would want to reach to training algorithms that are as parallelizable as possible.

A graphical representation of this method can be seen in figure 3.

4 Feedback Propagation

Given the success of DFA, a clear next step is to experiment with combinations of DFA and backpropagation. The inspiration for such an experiment is that it may be beneficial to introduce DFA only after the "vanishing gradient" starts becoming a problem, or possibly minimize the use of backpropagation in order to increase performance, but still try and optimize accuracy using a few layers of backpropagation. The Nokland paper presents some cases where DFA converges well but not to the same testing accuracy as backpropagation (we did not replicate the same issue).

Hence for we introduce Feedback Propagation (FP) as an alternative algorithm where the last k layers of the network (we will call this number k the BP depth parameter) are trained using BP and then remaining $(n-k)$ layers are trained using DFA with the last delta computed in the backpropagation

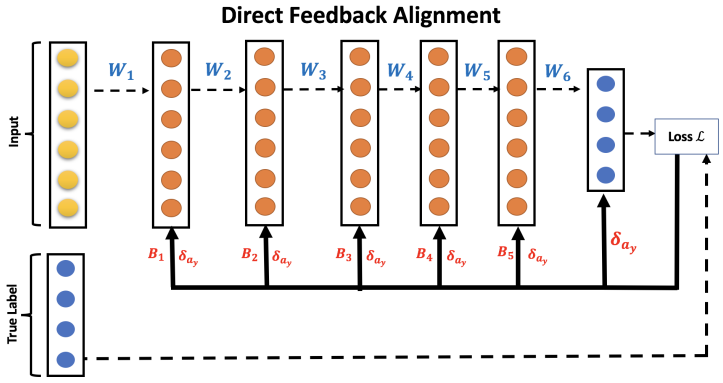

Figure 3: Diagram of how Direct Feedback Alignment (DFA) works in a network with 5 hidden layers. The notation used in this figure is the same one as in figure 1. Note how the error gradient is now being propagated only from the last layer to all layers and all layers use a random matrix as in FA.

stage of this algorithm. This would will make the overall algorithm less parellizable as the update of the last k layers cannot be parallelized. Neveretheless, the rest can in fact be parallelized and thus this algorithm is faster than BP.

A description of this algorithm can be seen in figure 4.

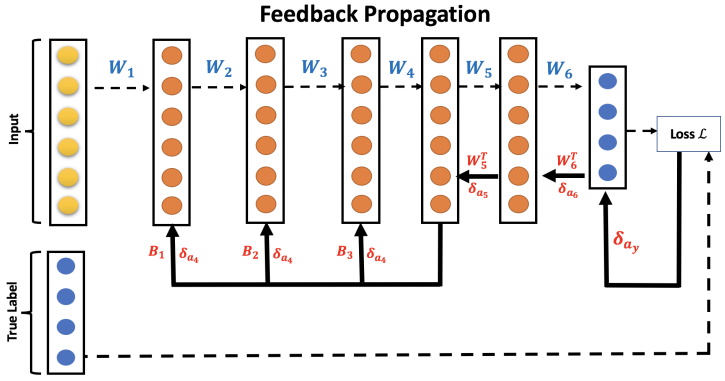

Figure 4: Diagram of how Feedback Propagation (FB) works in a network with 5 hidden layers. The notation used in this figure is the same one as in figure 1. In this example we use a BP depth of 3 layers (the last three layers are updated using normal BP).

5 Blocked Direct Propagation

Another possible approach is to allow *some* of the gradient to pass through the network during training, with the hypothesis that this may further speed up the training process and possibly allow a higher final accuracy. Intuitively, there is really no reason to assume that all connections are not reciprocal as DFA does. In fact, it has been show that biologically speaking reciprocal connections are possible ([12]). Hence allowing the existence of reciprocal connections could in fact improve the performance of the network while sacrificing some parallelization.

We formally implemented this idea using "blocks" of direct feedback aligned layers, but with a gradient being passed through between these blocks. In other words, each subsequent block of DFA trained layers takes its "delta" from the last layer of the proceeding DFA block. We call this method

Blocked Direct Feedback Alignment (BDFA). We do this with the hope that adding blocks will imposes a slight penalty in training, but may improve convergence performance or final testing accuracy.

A description of this algorithm can be seen in figure 5.

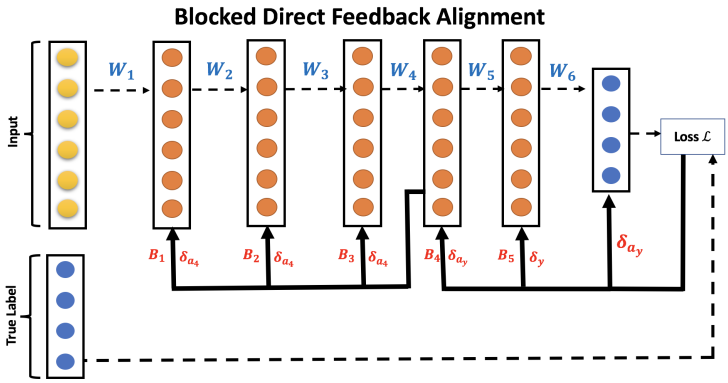

Figure 5: Diagram of how Blocked Direct Feedback Propagation (BDFB) works in a network with 5 hidden layers. The notation used in this figure is the same one as in figure 1. In this example our block size is equal to 3.

6 Experimentation

6.1 Setup

In order to test the various architectures for training proposed above, sample networks were trained on two datasets:

- **MNIST**
  MNIST is an industry standard dataset used for evaluating machine learning performance on a series of hand-drawn digits. Originally created by LeCun ([[11]]), it is often used as a toy dataset for evaluating and setting baselines for new architectures and training methods in Deep Learning.
- **notMNIST**
  notMNIST is a recently created dataset evaluating performance on the recognition of the letters A-J of the alphabet, in varying fonts and stylized in different ways. Arguably it is a harder task than MNIST as the characters themselves vary widely in terms of style (see Figure 6).

Figure 6: Sample images from notMNIST dataset.

The training code itself was implemented in Python via the Tensorflow framework, based on an iPython notebook by SangHyun Lee [2]. We used this code as a base for our code, however our end product end up diverging quite a bit from this original repository. The code is now available on GitHub [1].

We implemented the above experiments in the code, which are available by switching a flag in the python code to switch between normal Backpropagation, Feedback Alignment, Blocked Feedback Alignment, etc. The experiments were run on a variety of depths and hidden-layer sizes, specified below for each experiment.

6.2 Quantitative Results

6.2.1 Verification of DFA Implementation

We started our project by implementing the methods discussed in [14] and [12] to make sure we were capable of recreating the papers' original results before proceeding to trying our own variations. Initial experiments performed confirm the results of Nokland [14] - with Direct Feedback Alignment achieving comparable or better performance in the tests run. Initial tests on MNIST and notMNIST performed with just 15 layers repeat the findings of the Nokland.

An experiment highlighting the nature of DFA as being a solution to the Vanishing Gradient problem is a fully connected network trained on MNIST with *100 layers* and *200 hidden units*. The extremely deep nature of this architecture makes backpropagation all but impossible, but DFA has no issue training this network as can be seen in Figure 8.

6.2.2 Feedback Propagation in 15 Layers

In order to test the concept of mixing backpropagation for the later layers with Direct Feedback Alignment for the earlier layers, we run some experiments using FP while varying the BP depth parameter of this network. The experiments ran consisted of training a fully connected network with 720 hidden units and 15 layers. The last layers were trained using normal backpropagation, while the remaining first layers are trained using DFA with the delta term from the last backpropagation layer. The number of backpropagation layers was varied from five to zero (equivalent to DFA on the whole neural network). The results can be seen in Figure 7, showing that the fastest training procedure to converge was pure DFA, with each additional layer of backpropagation causing the convergence to slow down. The final testing accuracy after 10,000 steps was consistent among all six experiments - *94.6%*.

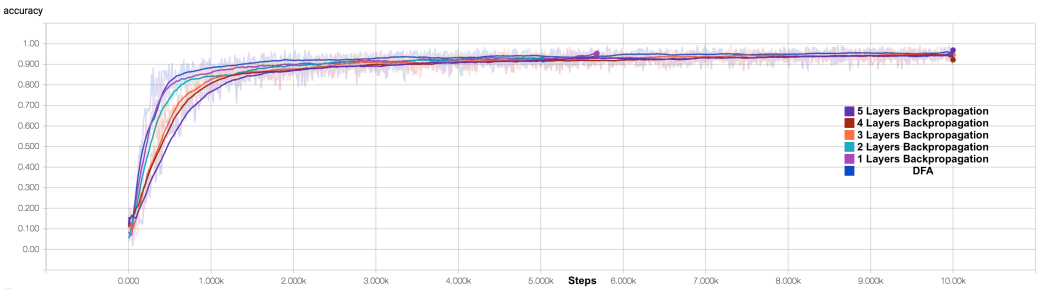

Figure 7: Accuracy results during training of mixed DFA layers followed by various lengths of backpropagation layers.

Surprisingly we were not able to obtain an improvement by increasing the number of layers using BP in our network. This may suggest that in fact the feedback received from the last layer (whose delta is the one that really tells us how we performed) is the best one to update subsequent layers. Intuitively this could make sense if we think of this feedback as the least "corrupted" feedback we can provide to previous layers. Furthermore, doing some backpropagation may damage the quality of the gradient used for the DFA part of the network as a cause of the vanishing gradient problem. Some future experiments to confirm whether this is true could consist on doing the same thing as above but now the DFA part of the network receives the feedback directly from the error in last layer of the network instead of from the last layer where we performed BP.

6.2.3 Blocked DFA

During experimentation we reallized that while normal DFA succeeds in training extremely deep networks, the Blocked DFA could help achieve a higher accuracy across all these layers while still maintaining high speed training and avoiding the vanishing gradient problem. Two experiments were run, each with 100 layers, and one with 200 hidden units and the second with 300 hidden units. The block sizes tested were 100 (pure DFA), 50 and 25. Smaller blocks achieved higher testing accuracy (Table 1,2) and much faster convergence as can be seen in Figure 8 and Figure 9, for 200 and 300 hidden units respectively. However, smaller block sizes had issues with convergence, the reason for which presents an interesting future area to explore.

Block Size	Testing Accuracy
100 (DFA)	89.2%
50	90.8%
25	**93.2**%

Table 1: Testing accuracy for 200 hidden units and 100 layers.

Block Size	Testing Accuracy
100 (DFA)	89.0%
50	90.0%
25	**93.1**%

Table 2: Testing accuracy for 300 hidden units and 100 layers.

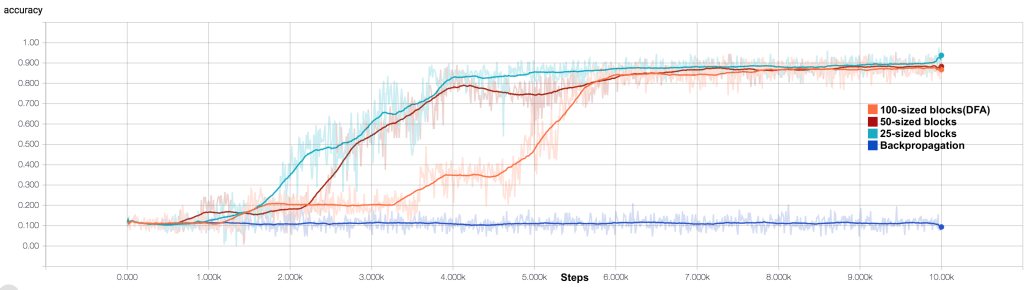

Figure 8: Accuracy results during training of various block sizes for Blocked DFA on a 100-layer fully connected network with 200 hidden units in each layer, also displaying vanishing-gradient issue with backpropagation.

These results may suggest that distributing feedback into blocks can in fact bring significant improvements to the network training. Note that this may somehow corroborate our claim made above that corrupting a gradient with BP may be a bad idea for feedback passing to other layers. The reason why this blocked version out performs DFA, however, is not quite clear for us and we would certainty be interested in running further experiments that could make the reason behind this phenomena more clear for us.

7 Future Work

In this section we will mention some interesting directions we can take in the future for further explore the variants presented in this paper and maybe discover better training algorithms at the end.

1. *Explore if knowing the derivative of the non-linearity is necessary: can we learn without knowledge of its derivative?*
   The discovery of the viability of DFA was a major surprise as it lacks a mathematical foundation for the fact that it works. It may be interesting to explore what other unexpected "modifications" could be made with the motivation of making neural networks more

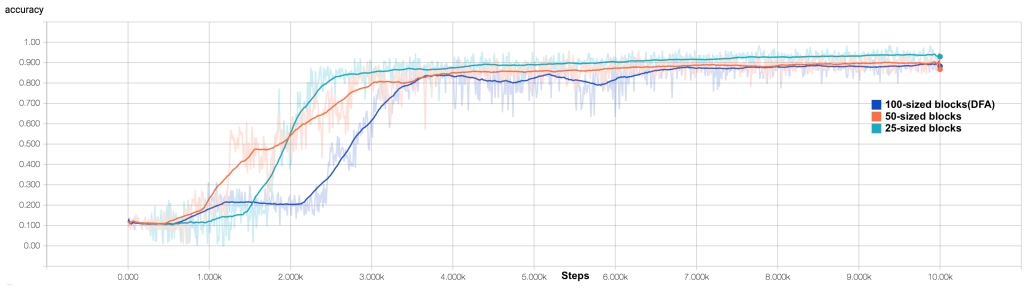

Figure 9: Accuracy results during training of various block sizes for Blocked DFA on a 100-layer fully connected network with 300 hidden units in each layer.

biologically plausible. For instance - could one avoid knowing the derivative of the activation function during the update step? This would certainty be a more biologically plausible model for the brain, as it would further limit any "processing" a neuron would have to do and would not assume neurons know the derivative of their activation function.

2. *Extend experiments to various other architectures and methods currently in use in the state of the art, such as LSTMs and convolutions.*
   Despite the promising results from DFA as well as Blocked DFA, all experiments thus far using DFA have been run on relatively very simple neural networks - fully connected networks with standard activation functions. It is an open question whether DFA or FA works on more complex architectures such as convolutions, where the update steps end up being far more complicated than a standard backpropagation step in a fully connected network. Nøkland claimed to try DFA out on convolutional networks, yet not a lot of experimentation was shown or done.

3. *Develop a mathematical foundation for DFA.*
   DFA and other derivatives of the concept will never be properly understood or accepted without formal justification for their performance. A key step in getting DFA accepted into the machine learning world is to provide justification for why it works as well as information on which conditions are required for it to work. Some early mathematical analysis has been done, but only for extremely simple networks of one or two layers (see [14]).

4. *Explore the use of DFA as a regulariser during training.*
   An interesting experiment to perform would be switching between backpropagation and DFA during training of datasets to explore the impact on the convergence. This could be done non-deterministically across layers and or entire training steps. We already know the "path" taken by DFA to the final state is different from backpropagation - what would allowing the training step to switch between these methods end up causing for the training process?

8 Conclusions

The main conclusions we got by working on this project this semester can be summarized in the following few key points:

- Direct Feedback Alignment does work and converges on fully connected neural networks to amazing results.
- In the limited experiments performed, DFA, FA, FP, and BDFA converges far quicker than back-propagation, both in terms of more quickly decreasing in loss, but also making each update step occur in constant time instead of a function of the depth of the network.

This allows training of huge networks without issue (i.e. avoiding vanishing gradient and performance issues).

- Getting feedback from multiple inner layers can be beneficial for learning better representations at the cost of some training time slowdown. This is true as long as the feedback used in all blocks is not "corrupted" through BP update steps (i.e. multiplications of several gradients from the chain rule).
- Feedback propagation seems to work equally as well as normal DFA. This could be caused by the fact that propagating gradients one or more layers can be enough for the feedback to be useless to update further layers in the correct direction.

Acknowledgments

We would like to thank Professor Bart Selman for his class on Artifical Intelligence which encouraged and allowed us to produce this project. We would also like to thank Professor Carla Gomes for assisting in procuring GPUs for training of our experiments, which proved to be invaluable. Furthermore we are humbled by the work of Arild Nokland, without whose independent and unpaid work we would never have been able to experiment with DFA.

References

[1] Blockeddirect feedback alignment implemenation, 2017.

[2] Feedback alignment ipython notebook, 2017.

[3] Yoshua Bengio, Ian J Goodfellow, and Aaron Courville. Deep learning. *Nature*, 521:436–444, 2015.

[4] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[5] Richard HR Hahnloser, Rahul Sarpeshkar, Misha A Mahowald, Rodney J Douglas, and H Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405(6789):947–951, 2000.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[7] Robert Hecht-Nielsen et al. Theory of the backpropagation neural network. *Neural Networks*, 1(Supplement-1):445–448, 1988.

[8] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[9] Gao Huang, Zhuang Liu, Kilian Q Weinberger, and Laurens van der Maaten. Densely connected convolutional networks. *arXiv preprint arXiv:1608.06993*, 2016.

[10] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[11] Yann LeCun, Corinna Cortes, and Christopher JC Burges. The mnist database of handwritten digits, 1998.

[12] Timothy P Lillicrap, Daniel Cownden, Douglas B Tweed, and Colin J Akerman. Random feedback weights support learning in deep neural networks. *arXiv preprint arXiv:1411.0247*, 2014.

[13] Grégoire Mesnil, Xiaodong He, Li Deng, and Yoshua Bengio. Investigation of recurrent-neural-network architectures and learning methods for spoken language understanding. In *Interspeech*, pages 3771–3775, 2013.

[14] Arild Nøkland. Direct feedback alignment provides learning in deep neural networks. *Advances In Neural Information Processing Systems*, 2016.

[15] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.