


Free Project Techhnical Report - M2 CompuPhys

- Technical Report - Implementation of Direct Feedback Alignment Using PyTorch

Léo BECHET
University of Franche-Comté

Under the supervision of Pr.Daniel BRUNNER,
and Dr.Anas SKALLI from the FEMTO-ST Institute

Contents

Abstract

This project aims to evaluate the effectiveness of Direct Feedback Alignment (DFA) as an alternative to backpropagation in training neural networks, particularly focusing on its applicability to batched data. Averaging on batches aims to lower the sampling cost. We introduce various averaging techniques for batch processing to better align with experimentally plausible models. The study contrasts DFA with traditional backpropagation by implementing both methods on simple and complex tasks. Specifically, we use the XOR problem as a proof of concept to demonstrate basic functionality and then transition to the MNIST dataset for a more rigorous and complex validation. These tests will provide insights into the performance, stability, and efficiency of DFA compared to backpropagation across different scales of complexity in neural network training.

Keywords: Direct Feedback Alignment, Backpropagation, Machine Learning, Neural Network Training.

1 McCulloch and Pitts Neuron

The **McCulloch and Pitts (MCP) neuron**[1] is one of the earliest models in artificial neural networks. It was introduced by Warren S. McCulloch and Walter Pitts in 1943 to formalize the functioning of neurons in biological brains.

1.1 Mathematical Model

The MCP neuron operates on a set of input signals, each weighted according to its importance. The neuron's output is determined based on whether the weighted sum of these inputs exceeds a certain threshold.

Given n input signals $x_1, x_2, \ldots, x_n$, each with corresponding weights $w_1, w_2, \ldots, w_n$, and a bias term b, the MCP neuron's output is given by:

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} w_i x_i + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

Here, y represents the binary output of the neuron (1 for "firing" or "activated," and 0 for "not firing").

1.2 Training Process

The training process involves adjusting the weights and bias to minimize errors. This is done using a set of input-output pairs where each pair consists of an input vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ and its corresponding desired output t.

The goal is to find the optimal weights $\mathbf{w} = [w_1, w_2, \ldots, w_n]$ and bias b such that the neuron's output matches the desired output for all training examples.

1.2.1 Learning Rule

A simple learning rule used in training MCP neurons is the perceptron learning rule. The steps are as follows:

1. **Initialization**: Start with random initial weights $\mathbf{w}_0$ and bias b_0.
2. **Iteration**:
   - For each input-output pair $(\mathbf{x}, t)$:

(a) Compute the weighted sum: $z = \sum_{i=1}^{n} w_i x_i + b$.

(b) Determine the output of the neuron:

$$y = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$

(c) If the output does not match the desired output (i.e., $y \neq t$):

i. Adjust the weights and bias:

$$w_i \leftarrow w_i + \eta(y - t)x_i$$

$$b \leftarrow b + \eta(y - t)$$

where η is a small positive constant known as the learning rate.

3. **Termination**: Repeat the iteration until all input-output pairs are correctly classified or a maximum number of iterations is reached.

1.2.2 Convergence

The perceptron learning rule guarantees that if the training data is linearly separable (i.e., there exists a hyperplane that can separate the two classes), the weights and bias will converge to values that correctly classify all input-output pairs.

1.3 Example

Consider a simple example with two inputs x_1 and x_2, and desired outputs for training examples as follows:

1. **Input**: $\mathbf{x}_1 = [1, 0]$, Desired Output: $t_1 = 1$
2. **Input**: $\mathbf{x}_2 = [0, 1]$, Desired Output: $t_2 = 0$

Starting with random weights and bias, after several iterations of the perceptron learning rule, the neuron's parameters will converge to values that correctly classify these inputs.

Initial weights: $w_1 = 0.5, w_2 = -0.5, b = 0$
After training, let's assume: $w_1 = 0.6, w_2 = -0.4, b = 0.3$

These values would ensure the neuron correctly outputs $t_1 = 1$ for $\mathbf{x}_1$ and $t_2 = 0$ for $\mathbf{x}_2$.

This completes the presentation of the McCulloch and Pitts neuron, including its mathematical model and training process.

2 Multi-Layer Perceptron (MLP)

2.1 Introduction

The Multi-Layer Perceptron (MLP)[2] is a fundamental type of artificial neural network. It consists of multiple layers of nodes connected in a feedforward manner, with each node applying a weighted sum and an activation function to its inputs.

2.2 Neuron Model

A single neuron in the MLP is an extension of the McCulloch and Pitts neuron and can be mathematically modeled as follows:

$$z = \sum_{j=1}^{n} w_j x_j + b \tag{1}$$

where:

z is the pre-activation (or net input).

w_j are the weights associated with the inputs.

x_j are the inputs to the neuron.

b is the bias term.

The output of the neuron, often denoted as a, is then passed through an activation function f:

$$a = f(z) \tag{2}$$

Common activation functions include:

Sigmoid: $f(z) = \frac{1}{1+e^{-z}}$

Tanh: $f(z) = \tanh(z)$

ReLU (Rectified Linear Unit): $f(z) = \max(0, z)$

2.3 Multi-Layer Perceptron Architecture

An MLP consists of multiple layers:

1. **Input Layer**: Receives the raw input data.
2. **Hidden Layers**: Intermediate layers that transform the input data using a series of neurons with activation functions.
3. **Output Layer**: Produces the final output, often corresponding to the class labels or continuous values.

The architecture can be represented as follows:

$$\text{Input} \xrightarrow{\text{Weights}} \text{Hidden Layers} \xrightarrow{\text{Activation Functions}} \text{Output Layer} \tag{3}$$

2.4 Training the MLP

Training an MLP involves adjusting the weights and biases to minimize a loss function. This is typically done using an optimization algorithm like gradient descent.

The goal is to find the set of parameters $\theta = [w, b]$ that minimizes the loss function:

$$L(\theta; x^{(i)}, y^{(i)}) = \text{Loss Function}(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}) \tag{4}$$

where:

- $\mathbf{x}^{(i)}$ is the input vector for example i.

- $\mathbf{y}^{(i)}$ is the target output vector for example i.

Common loss functions include:

Mean Squared Error (MSE):

$$L(\theta; x^{(i)}, y^{(i)}) = \frac{1}{2}\|a_L - t\|^2$$

where a_L is the predicted output and t is the true target.

This loss function is suitable for regression problems.

Binary Cross-Entropy (BCE):

$$L(\theta; x^{(i)}, y^{(i)}) = -[y^{(i)} \log(a_L) + (1 - y^{(i)}) \log(1 - a_L)]$$

This loss function is suitable for binary classification problems.

Root Mean Squared Error (RMSE):

$$L(\theta; x^{(i)}, y^{(i)}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(a_L - t)^2}$$

This loss function is also a variant of the MSE and is used in regression tasks.

The loss function measures the discrepancy between the network's predictions and the actual targets. It must be wisely chosen to fit the problem the MLP has been tasked to solve.

The training process can be summarized as follows:

1. **Forward Propagation**: Compute the outputs through each layer using the current weights and biases.
2. **Compute Loss**: Calculate the loss function for a given set of inputs and targets.
3. **Backward Propagation**: Use backpropagation to compute the gradient of the loss with respect to each weight and bias in the network.
4. **Update Weights**: Adjust the weights using an optimization algorithm such as stochastic gradient descent (SGD):

$$w \leftarrow w - \eta \frac{\partial L}{\partial w} \tag{5}$$

where η is the learning rate.

5. **Repeat Steps 1-4**: Iterate over the training data and update the weights until convergence or a stopping criterion is met.

2.5 Example Training Process

Here's an example of the training process for a simple MLP with one hidden layer:

1. Initialize random weights w and biases b.
2. For each input $x^{(i)}$:

(a) Compute the pre-activation: $z = \sum_{j=1}^{n} w_j x_j + b$

(b) Apply the activation function to get the output: $a = f(z)$

(c) Calculate the loss and its gradients.

3. Update the weights using gradient descent:

$$w \leftarrow w - \eta \frac{\partial L}{\partial w} \tag{6}$$

4. Repeat steps 2-3 until convergence or a stopping criterion is met.

3 Direct Feedback Alignment (DFA)

3.1 Introduction

Direct Feedback Alignment[3] (DFA) is an algorithm introduced as an alternative to back-propagation (BP) for training artificial neural networks (ANNs). Unlike BP, DFA does not require symmetric weights for forward and backward propagation. Instead, it uses fixed random feedback connections to propagate error signals directly from the output layer to each hidden layer. This approach eliminates the need for layer-by-layer backpropagation of errors, simplifying the computational process and making the method more biologically plausible.

3.2 Principle Behind Direct Feedback Alignment

The principle of DFA is based on the feedback alignment concept, where fixed random feedback weights are used to transmit the error signal to hidden layers. This allows each layer to be trained more independently of the others while avoiding the constraints of symmetric feedback weights. The network adapts to utilize the random feedback effectively, enabling it to achieve error-driven learning comparable to BP. This method also reduces the dependency on layer-wise error propagation and derivative computations, offering a simpler and potentially more robust training mechanism for deep networks as well as a high potential for parallelizability.

3.2.1 Mathematical Model

Disclaimer: The following is directly taken out of the paper *Direct Feedback Alignment Provides Learning in Deep Neural Networks* from Arild Nokland, 2016, and informations relevant to DFA are extracted.

Consider a 1 hidden layer MLP, we can define a forward pass the following way:

$$a_1 = W_1 * x + b_1, \quad h_1 = f(a_1) \tag{7}$$

$$a_2 = W_2 * h_1 + b_2, \quad h_2 = f(a_2) \tag{8}$$

$$a_y = W_3 * h_2 + b_3, \quad \hat{y} = f_y(a_y) \tag{9}$$

We can again define the error between the network output and the the target as follows:

$$e = \hat{y} - y \tag{10}$$

DFA defines the hidden layers update directions using the following formula:

$$\delta_{a_2} = (B_2 e) f'(a_2), \delta_{a_1} = (B_1 e) f'(a_1) \tag{11}$$

Where B_1 and B_2 are random matrices. Ignoring the learning rate, the weight updates for all methods are calculated as

$$\delta W_1 = -\delta a_1 x^T, \delta W_2 = -\delta a_2 h_1^T, \delta W_3 = -e h_2^T \tag{12}$$

Bias update method is left to the user. In this work, the method used is the following:
For each hidden layer i:

$$\delta b_i = -\sum_{j=1}^{n_j} \delta a_i \tag{13}$$

For the output layer:

$$\delta b_y = -\sum_{j=1}^{n_j} e \tag{14}$$

3.2.2 Advantages and Limitations

Simplicity: DFA simplifies the backpropagation, making it more computationally efficient.

Parallelizability: DFA can be efficiently parallelized since all layers can be updated at once, contrary to backpropagation which depends on the result of the previous layer.

Gradient Stability: By ignoring feedback weights, gradients are less likely to vanish or explode.

Limited Backtracking: Only forward propagation of errors is used, which can be a limitation in complex ANN architectures.

4 DFA Implementation

4.1 Forewords

Implementation

The implementation is done in Python, leveraging PyTorch to provide a widely used, flexible and efficient framework for defining and training neural networks. The goal is to create a library that includes a DFA model class and various training loops. Please refer to the Technical Report for the code-specific explanations

Training Loops

Different training loops are implemented:

- Train the network using standard batching method (i.e. compute the output for each items in a batch, before correcting the error of each element).
- Train the network by averaging on the whole batch.
- Train by averaging on the whole batch for each label.
- Same as 3 but with a learning rate scheduler.

Basic Examples

Three basic examples are included:

- Proof of concept training using XOR.
- Training on MNIST with different available training loops.
- Comparison between backpropagation and DFA (convergence and angles between gradients).

4.2 Proof of concept on XOR

XOR Truth Table

The XOR problem is defined in Table 1:

Input	Output
(0, 0)	0
(0, 1)	1
(1, 0)	1
(1, 1)	0

Table 1: Truth table of the XOR logic door.

Inducing Non-Linearity

The XOR problem is a classic example that requires a network with at least one hidden layer to model due to its non-linear nature. This characteristic makes it ideal for demonstrating the capabilities of DFA.

Training Configuration and Results

The training was performed using:

- Batch size: 1.
- Number of epochs: 1000.
- Learning rate: 0.1.
- Architecture: 2 input neurons, 4 hidden neurons, 1 output neuron.

Output Table

Table 2 shows the outputs for different inputs:

Input	Expected Output	Actual Output
(0, 0)	0	0.0057
(0, 1)	1	0.9882
(1, 0)	1	0.9894
(1, 1)	0	0.0103

Table 2: Outputs following training using DFA.

The goal was to approximate output values close to 0 and 1, rather than using a simple threshold function that would directly give the exact outputs. This exercise demonstrates DFA's ability to optimize weights effectively.

Loss and Training Error Plots

Fig.1 and Fig.2 show the loss and training error plots, respectively. These plots demonstrate that DFA minimizes the loss and reduces training error, confirming that our preliminary implementation works as intended. The minimized loss indicates successful optimization. DFA is thus a valid training method for Neural Networks.

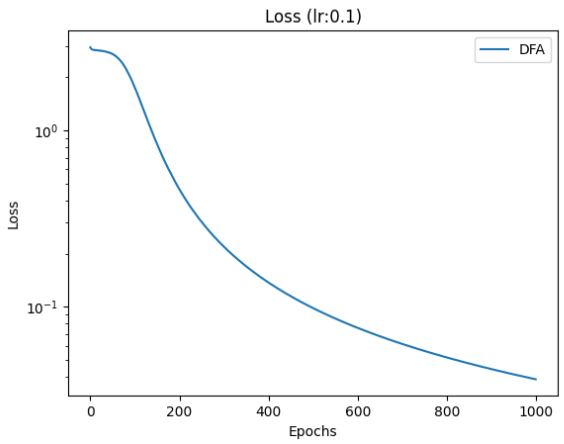

Figure 1: Loss Plot for XOR Proof of Concept

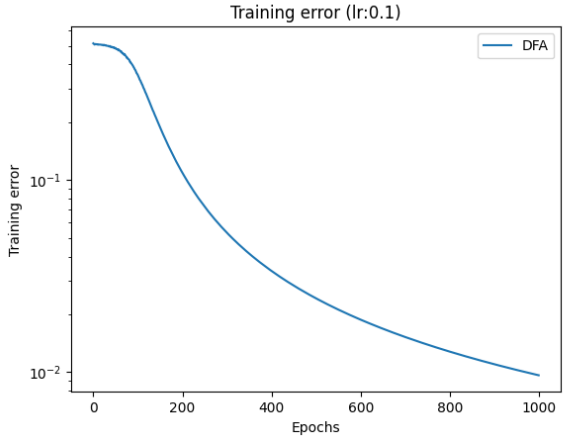

Figure 2: Training Error Plot for XOR Proof of Concept

4.3 DFA vs Backpropagation

Direct Feedback Alignment (DFA) is now being used for a more challenging problem with MNIST, since the proof of concept has already been validated. We compare DFA with traditional backpropagation using a training setup with 500 samples per batch, a learning rate of 1×10^{-1}, and 200 epochs. The architecture consists of an input layer of size 784 (from the 28x28 images), an 80-node hidden layer, and an output layer with 10 nodes for the 10 digits.

Fig.3 depicts the training error (TE) over epochs for both methods, highlighting their similar performance in terms of reducing the error. This plot demonstrates that both Direct Feedback Alignment (DFA) and backpropagation effectively minimize the training error during the learning process.

Fig.4 illustrate that comparing losses directly can be misleading due to the differing scales of DFA and backpropagation. Here, BP uses RMSELoss while DFA is defined with an error computed using $e = \hat{y} - y$. As such, for the DFA we accumulate the error instead of computing the RMSELoss. However, their training errors align quite closely. Specifically, both methods exhibit similar performances in terms of reducing the error over epochs.

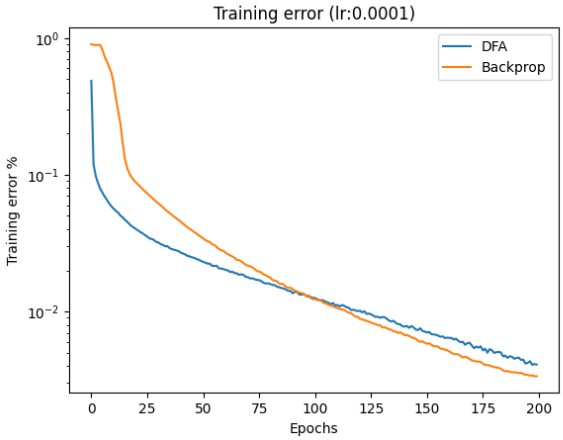

Figure 3: Training error (TE) over epochs for both methods

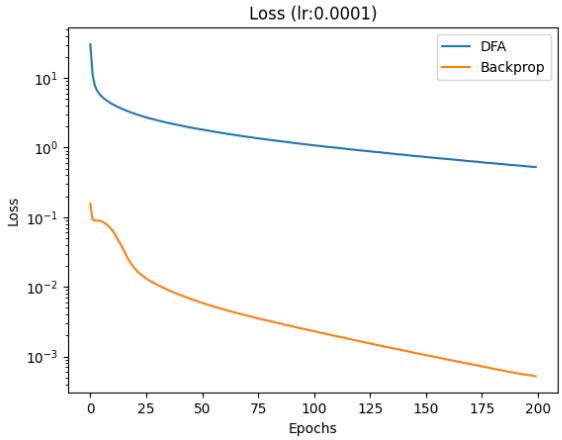

Figure 4: Comparison of losses between DFA and backpropagation.

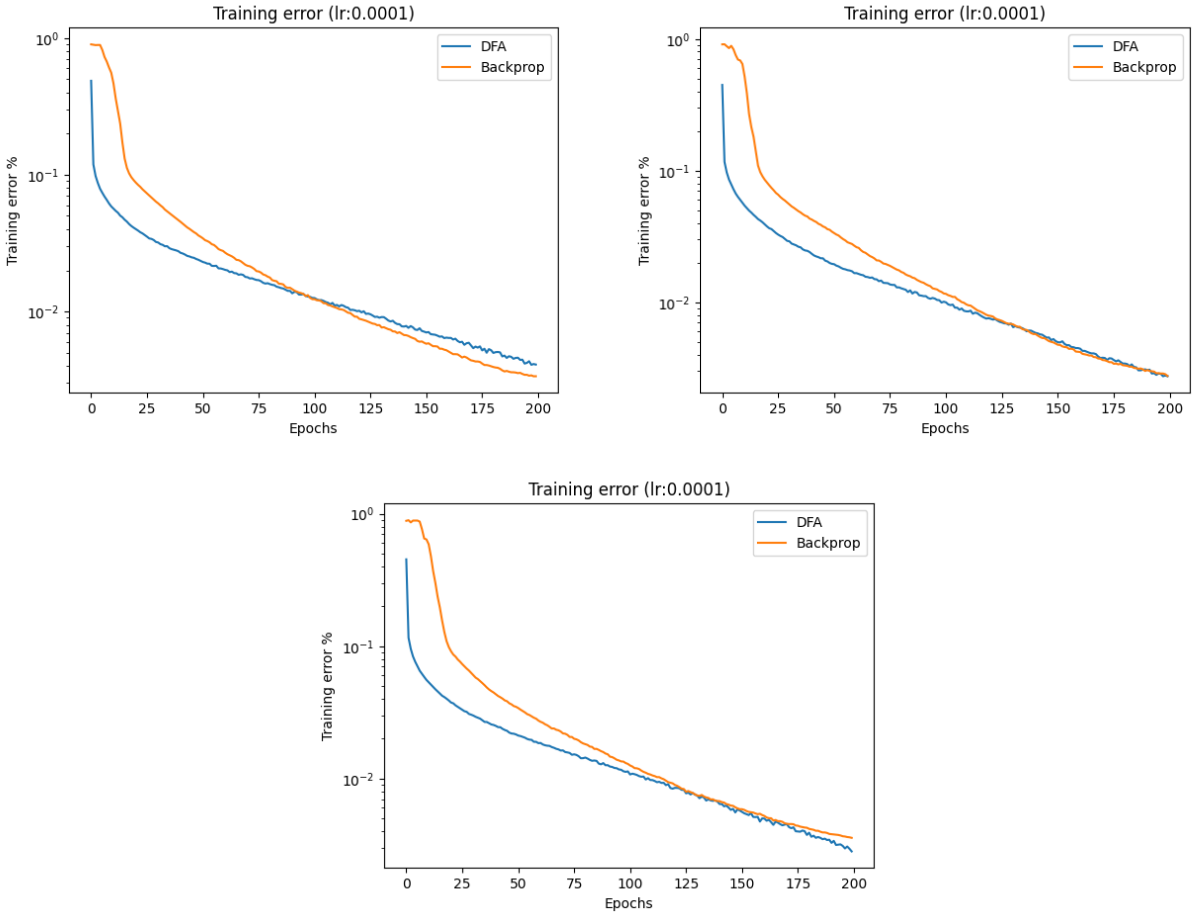

Figure 5: Comparison of training error (TE) and performance between Direct Feedback Alignment (DFA) and backpropagation.

We also found that initial matrix B values are crucial for the performance of both methods. We usually initialized these matrices using a normal distribution with mean 0 and standard deviation (std) set to $\sigma = 1$, as provided by PyTorch's 'randn()' function. However after fiddling with the matrices initialization, we've found out that different initialization leads to different convergence, explaining the different behaviors in Fig.6.

After a quick analysis and tests, we settled at $\sigma = 1.9$. Interestingly, this initialization led to better performance compared to backpropagation. Playing with only the mean parameter resulted in consistently worse performance. Thus, we hypothesize that optimizing the initial matrix standards deviations can significantly improve convergence rates.

Please note that this behavior was discovered close to the end of the allocated time for this project. As such, this behavior was not extensively researched. We can however infer some early hypothesis from our current results

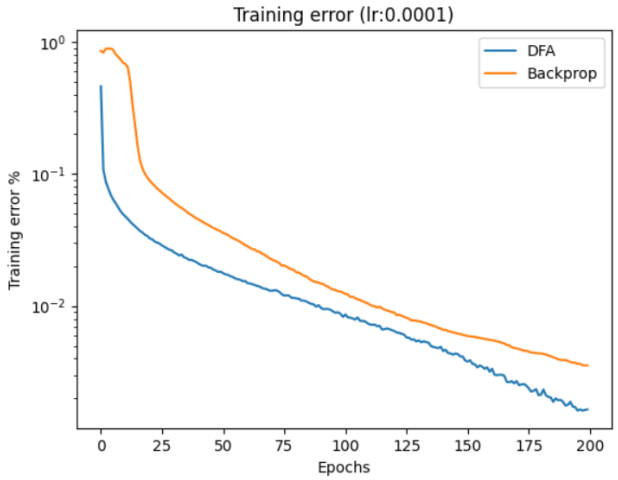

Figure 6: Comparison of training error between optimized DFA and unoptimized backpropagation, with B matrices initialized using a normal distribution of $\sigma = 1.9$

As we can see on Fig.6, the DFA outperforms backpropagation. This is however impossible. It is mathematically certain that DFA cannot outperform Backpropagation[3]. The current behavior is an artefifact of the fact that while we optimized hyper parameters for the DFA, the backpropagation is unoptimized.

However, for the remainder of our project, we continued using a standard deviation of 1, as the optimal σ was discovered late in the allocated time.

In summary, these observations highlight the importance of careful initialization and suggest that further exploration into alternative distributions might yield even better results.

4.4 Gradient angles between DFA and backpropagation

Here we compute the angles between the gradients of a neural network optimized using DFA and one optimized using traditional backpropagation for each batch in our training process. Specifically, for each batch, we first clone the models to ensure independent optimization paths. One model is updated using DFA while the other uses backpropagation. We then compute the angles between the gradients of these two models for each layer. The angle is accumulated through one full epoch before being appended to a list and reset.

The angles are computed using the following mathematical equation:

$$\theta = \arccos\left(\frac{\mathbf{v}_1 \cdot \mathbf{v}_2}{\|\mathbf{v}_1\|\|\mathbf{v}_2\|}\right)$$

where $\mathbf{v}_1$ and $\mathbf{v}_2$ are the flattened vectors of gradients from the DFA-optimized model and the backpropagation-optimized model, respectively. The dot product $\mathbf{v}_1 \cdot \mathbf{v}_2$ is computed for each layer, followed by normalizing these vectors to find their magnitudes $\|\mathbf{v}_1\|$ and $\|\mathbf{v}_2\|$. This process is repeated for every parameter in the respective layers.

Finally, we plot the results of these angle computations. Fig.7 shows the computed angles for each layer across different training epochs. As can be observed, the angles do not converge but rather oscillate around a stable value. This suggests that DFA and backpropagation do not tend toward the same optimized network structure.

To further illustrate this, Fig.8 presents the training error. A peculiar result is that backpropagation seems to always provide a better update step, being under DFA. However, as we've seen in a previous section, DFA leads. It appears that DFA and Backpropagation minimize the loss function using different directions in the parameter space.

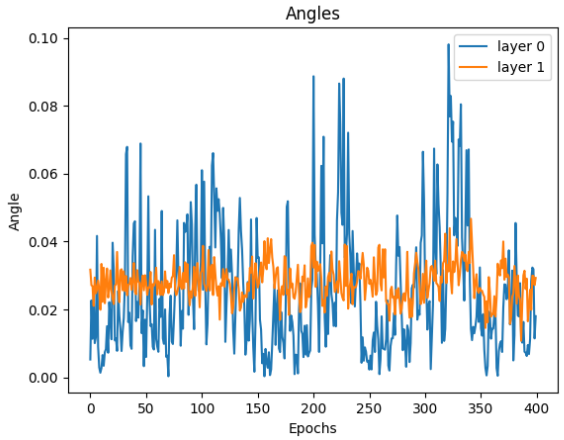

Figure 7: Angles between gradients of DFA and backpropagation-optimized models across different training batches.

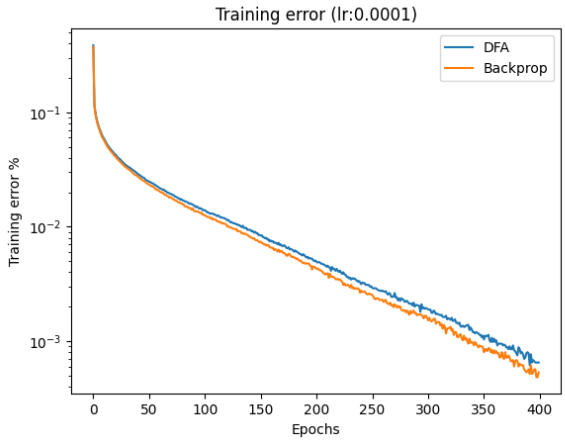

Figure 8: Training error resulting from a backpropagation and DFA optimization step.

4.5 Averaging on a Batch

Here we describe a method where, for each batch, all outputs are averaged into a "master" output, and similarly, all inputs and targets are also averaged. The same process is applied to the intermediate variables a and h. This approach involves computing an error between the master input and the master target, which then updates the network in one step.

The averaging of $\hat{y}$ (the model's output) and batch_y (the actual targets) is performed using the following equation:

$$\text{master\_output} = \frac{1}{N}\sum_{i=1}^{N}\hat{y}_i, \quad \text{master\_target} = \frac{1}{N}\sum_{i=1}^{N} y_i$$

where N is the batch size. Similarly, for inputs and intermediate variables:

$$\text{master\_input} = \frac{1}{N}\sum_{i=1}^{N} x_i, \quad \text{master\_h} = \frac{1}{N}\sum_{i=1}^{N} h_i, \quad \text{master\_a} = \frac{1}{N}\sum_{i=1}^{N} a_i$$

These equations are applied in the context of training, where x represents the input data, and $\hat{y}$ represents the predicted outputs. The error is then computed as:

$$\text{error} = \text{master\_output} - \text{master\_target}$$

This method significantly reduces the number of updates to one per batch, but it faces challenges when dealing with uniform label distributions within batches.

It appears that this approach works well for small batch sizes where labels are not evenly distributed. For instance, consider a scenario where we have 10 cat images and 10 dog images. In such cases, the average target label will be 50/50 between cats and dogs, making it difficult for the network to learn the distinct features of each class. However, if we use batches with only 1 cat image and 9 dog images, the average label would skew towards representing a dog, thereby facilitating better learning.

On increasing the batch size, the distribution of labels tends towards a normal distribution, leading to an averaged output that is less distinctive. For example, averaging 100 images where each class is equally represented (50 cats and 50 dogs) results in an average label that does not favor either category strongly, making it challenging for the network to learn specific features.

This phenomenon can be illustrated with a simple numerical example: suppose we have two categories, A and B, and our batch consists of images from these categories. If the labels are averaged over a small number of samples (e.g., 10), the resulting average might be heavily skewed towards category A if there are more instances of A in the batch. Conversely, with larger batches where the distribution is closer to equal, averaging would result in an almost neutral output.

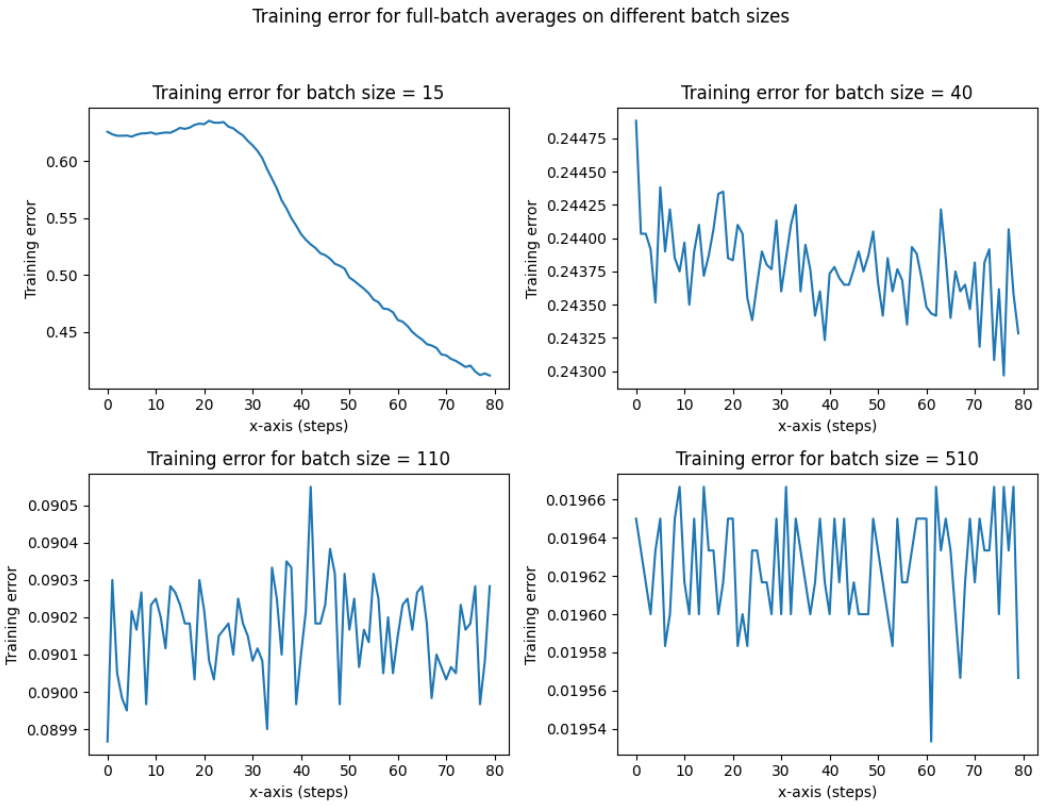

Figure 9: Effect of Batch Size on the training error when using MNIST dataset (10 labels).

In Figure 9, it clearly appears that when the batch is biased towards a specific label due to undersampling, it will still learn features. However as the batch size increases, the average label tends towards a neutral state, reducing the network's ability to learn distinctive features. This is clearly shown by the graphs. While a size of 40 still leads to a decreasing tendency, 110 and 510 are simply too high to show any sign of convergence. However, 15 shows a typical convergence, albeit quite slow.

While this method ensures that the network is trained using a representative average of the batch, it can lead to suboptimal learning when labels are uniformly distributed. Averaging on a batch helps reduce the number of updates but will lead to suboptimal convergence rates and performance. Smaller batch sizes with skewed distributions facilitate better learning of distinctive features, while larger batches tend towards neutral averages, hindering the network's ability to capture these features effectively.

As such, when using this method, it is important to consider the distribution of labels inside a specific training set. Having control on the skewing is critical for performance.

4.6 Averaging per class on a batch

In this section, we describe the process of averaging inputs, targets, and intermediate representations per class within a batch during training. This approach helps reduce the number of updates to the network while retaining meaningful gradient information for each class.

We begin by separating the batch into subsets corresponding to each class. For a batch containing N samples and C classes, we group indices of the samples based on their class label. Denoting $\mathbf{y}_{\text{batch}}$ as the target labels and $\hat{\mathbf{y}}_{\text{batch}}$ as the model predictions, the averaging operations for each class i are expressed as:

$$\hat{\mathbf{y}}_i = \frac{1}{|S_i|} \sum_{j \in S_i} \hat{\mathbf{y}}_j, \quad \mathbf{y}_i = \frac{1}{|S_i|} \sum_{j \in S_i} \mathbf{y}_j,$$

where S_i is the set of indices for samples belonging to class i, and $|S_i|$ is the size of this set. Similar averaging is performed for the intermediate activations $\mathbf{a}$ and $\mathbf{h}$:

$$\mathbf{a}_i = \frac{1}{|S_i|} \sum_{j \in S_i} \mathbf{a}_j, \quad \mathbf{h}_i = \frac{1}{|S_i|} \sum_{j \in S_i} \mathbf{h}_j.$$

This process reduces the computational overhead by averaging over the class-specific subsets instead of updating the network for each individual sample.

4.6.1 Impact of Learning Rate

Figure 10 illustrates the impact of varying the learning rate on convergence behavior for a batch size of 500. The results show that the choice of learning rate is critical. A high learning rate prevents the model from converging due to large update steps that fail to settle in local minima. Conversely, a very small learning rate significantly increases convergence time. We observe a plateau at the beginning of training, which likely corresponds to an initial pre-alignment of the network before converging to a local minimum. Notably, the duration of this plateau and the overall convergence rate are dependent on the learning rate.

Based on these observations, we hypothesize that employing a learning rate schedule could shorten the time spent in the plateau phase, thereby accelerating convergence. Additionally, optimizing the initialization of the feedback matrices $\mathbf{B}$, as discussed in Section 4.3, may further improve convergence performance.

Figure 10: Impact of learning rate on convergence for a batch size of 500.

4.6.2 Impact of Batch Size

Figure 11 shows the effect of batch size on convergence speed per epoch, using a fixed learning rate of 0.001. Larger batch sizes result in fewer updates per epoch, slowing convergence speed in terms of epochs. However, convergence is still achieved even with larger batch sizes. This is particularly promising for experimental setups where extracting separate labels during integration time is feasible.

It is worth noting that while smaller batch sizes converge faster in terms of epochs, the computational time required for each epoch is significantly higher. For practical applications, it is essential to also account for time per epoch when comparing the efficiency of different batch sizes.

Figure 11: Impact of batch size on convergence with a fixed learning rate of 0.001.

5 Conclusion

In this project, we explored the implementation and application of Direct Feedback Alignment (DFA) as an alternative to backpropagation for training neural networks. Our results demonstrate that DFA successfully

minimizes training loss and error in both simple and complex problems, such as the XOR and MNIST datasets, validating its potential as a training method.

Key findings include the sensitivity of DFA's performance to the initialization of weight matrices and the significance of hyperparameter optimization. While DFA performed comparably to backpropagation in terms of error reduction, further investigation revealed that its performance could be enhanced by fine-tuning initialization parameters, such as the standard deviation of matrix B. However, it was also observed that this improvement did not inherently indicate superiority over backpropagation but rather reflected the lack of equivalent optimization for both methods during the experiments.

Additional experiments on gradient angles confirmed that DFA and backpropagation utilize different optimization trajectories, emphasizing the distinct mechanisms underlying these methods. Moreover, the averaging strategies explored in the batch and class-specific contexts provided insights into the trade-offs between computational efficiency and model convergence. Smaller batch sizes with skewed distributions proved beneficial for feature learning, while larger batches hindered convergence due to neutral averaging effects.

While DFA offers a promising alternative to backpropagation, its limitations, such as sensitivity to initialization and hyperparameter selection, warrant further research. Future work could focus on improving DFA's robustness and exploring its applicability to diverse architectures and datasets. Overall, this study highlights the potential of DFA and lays the groundwork for advancing understanding and application of alternative training methods in neural networks, especially in regads to averaged batches in training.

References

[1] W. Mcculloch and W. Pitts, "A logical calculus of ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, vol. 5, 1943.

[2] F. Rosenblatt, "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain," *Psychological Review*, 1958.

[3] A. Nøkland, "Direct Feedback Alignment Provides Learning in Deep Neural Networks," *arXiv e-prints*, p. arXiv:1609.01596, Sept. 2016.