Karlsruhe Institute of Technology

SCC
Steinbuch Centre for Computing

Feed-Forward-Only Training of Neural Networks

Master Thesis of

Katharina Flügel

at the Department of Informatics
Steinbuch Centre for Computing (SCC)

Reviewer:	Prof. Dr. Achim Streit
Second reviewer:	Prof. Dr. Bernhard Neumair
Advisor:	Dr. Markus Götz
Second advisor:	Daniel Coquelin

June 1st 2021 – December 1st 2021

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, December 1st 2021

. . . . . .

(Katharina Flügel)

Abstract

While artificial neural networks have reached immense advances over the last decade, the underlying approach to training neural networks, that is, solving the credit assignment problem by computing gradients with back-propagation, has remained largely the same. Nonetheless, back-propagation has long been criticized for being biologically implausible as it relies on concepts that are not viable in the brain. With delayed error forward projection (DEFP), I introduce a feed-forward-only training algorithm that solves two core issues for biological plausibility: the weight transport and the update locking problem. It is based on the similarly plausible direct random target projection algorithm but improves the approximated gradients by using delayed error information as a sample-wise scaling factor in place of the targets. By evaluating delayed error forward projection on image classification with fully-connected and convolutional neural networks, I find that it can achieve higher accuracy than direct random target projection, especially for fully-connected networks. Interestingly, scaling the updates with the error yields significantly better results than scaling with the gradient of the loss for all networks and datasets. In total, delayed error forward projection demonstrates the applicability of feed-forward-only training algorithms. This offers exciting new possibilities for both in-the-loop training on neuromorphic devices and pipelined parallelization.

Zusammenfassung

Obwohl künstliche neuronale Netze in den letzten Jahren beeindruckende Fortschritte erzielen konnten, ist der für ihr Training verwendete Ansatz – eine Optimierung mit dem Gradientenverfahren basierend auf dem Backpropagation-Algorithmus – weitestgehend gleich geblieben. Aus biologischer Sicht wird Backpropagation allerdings als unplausibel kritisiert, da die zugrunde liegenden Konzepte nicht in dieser Form im menschlichen Gehirn ablaufen können. Mit Delayed-Error-Forward-Projection (DEFP) stelle ich einen neuen Trainingsalgorithmus vor, der ausschließlich Vorwärtspropagierung verwendet und zwei Kernprobleme der biologischen Plausibilität von Backpropagation löst: das Gewichtstransport- und das Update-Locking-Problem. Delayed-Error-Forward-Projection beruht auf dem ähnlich plausiblen Direct-Random-Target-Projection-Algorithmus, verbessert aber die Gradientenapproximation durch eine Skalierung mit verzögerten Fehlerinformationen anstelle einer rein auf den Zielvariablen basierenden Skalierung. Bei der experimentellen Evaluation anhand von Bildklassifikationsproblemen unter Verwendung von vollständig verbundenen sowie Faltungsnetzwerken zeigt sich, dass Delayed-Error-Forward-Projection eine höhere Genauigkeit als Direct-Random-Target-Projection erreichen kann. Dies gilt insbesondere für vollständig verbundene Netzwerke. Interessanterweise führt die Skalierung der Aktualisierungsschritte mit dem Fehler bei allen getesteten Netzwerken und Datensätzen zu einer deutlichen Verbesserung im Vergleich zu einer Skalierung mit dem Gradienten der Verlustfunktion. Diese Ergebnisse zeigen das große Potential reiner Vorwärtsalgorithmen wie Delayed-Error-Forward-Projection zum Training neuronaler Netze auf. Damit eröffnen sich vielfältige neue Trainingsmöglichkeiten sowohl auf neuromorphen Geräten als auch für die Parallelisierung durch Pipelining.

Acknowledgments

This work was performed on the HoreKa supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research.

Contents

List of Figures

List of Tables

1. Introduction

Within the last decade, there has been a surge of interest in artificial neural networks. Facilitated by the availability of large datasets, increased computing power, and new methodological approaches, there has been an immense increase in the predictive performance and prevalence of neural networks. Used in a wide array of applications, they often outperform previous approaches, thus setting a new state-of-the-art. These applications include computer vision tasks like image recognition [28, 70, 43], object detection and segmentation [24, 49], and image captioning [78] but also tasks in natural language processing (NLP) such as speech recognition [31, 1] and machine translation [37]. With these tasks, artificial neural networks solve many different practical problems, from self-driving cars [3, 10] to biomedical applications like detecting tumors [66, 63].

While there has been immense progress with how neural networks are trained and designed [40, 54, 28, 32], the underlying algorithms used to train them have remained mostly the same. Today, the vast majority of neural networks is still trained using a gradient-descent-based optimization algorithm combined with back-propagation [64] for computing the required gradients [26]. The forward pass determines the network's output, which a loss function compares to the desired target value. In the backward pass, back-propagation computes the gradient of the resulting loss with respect to the model parameters. Based on these gradients, the optimization algorithm adjusts the parameters in the direction of the negative gradient to reduce the loss and improve the network's output. Alternating these forward and backward passes iteratively, the training seeks to minimize the loss over all training samples.

Back-propagation offers many advantages by efficiently computing the exact gradient of the loss with respect to all model parameters. However, it also comes with several disadvantages that hinder both its computational performance and make it biologically implausible. Two of the most significant issues are the weight transport and the update locking problem. In back-propagation, the forward weights are reused in the backward pass to propagate the gradients backward. This is, however, biologically implausible as synapses are unidirectional and separate forward and backward pathways would require a synchronization of the weights between these pathways while the brain is inherently asynchronous and event-based [79, 72]. This implausibility is referred to as the *weight transport problem* [27]. The *update locking problem* [36, 33, 22] relates to the dependencies between the forward and backward passes. Having been processed in the forward pass, a layer needs to wait until all its downstream layers have been processed by both the forward and the backward pass before it can be updated. This can take a considerable amount of time and thus cause earlier layers to become desynchronized with the error.

Several approaches to solving these problems exist. One such approach is *feed-forward-only training*, which attempts to solve the weight transport and update locking problems by effectively removing the backward pass, thus training neural networks with only a

forward pass. With *feedback alignment (FA)*, Lillicrap et al. [48] provide a possible solution to the weight transport problem. They demonstrate that the backward weights can be replaced by fixed random feedback weights, as the forward weights come to align with these feedback weights enabling effective training. With this strategy, FA is competitive to back-propagation on multiple different tasks, including multi-layer networks with non-linear layers, and can even achieve higher accuracy on some. *Direct feedback alignment (DFA)* by Nøkland [56] is an extension to FA. It replaces the feedback pathways, which propagate the feedback backward layer-by-layer, with direct pathways passing the error directly from the output layer to all hidden layers. This solves the non-locality of back-propagation, relaxes constraints on the memory access pattern [15, 22], and reduces the dependency on the initialization of the forward weights [56]. Direct feedback alignment can train deep neural networks on different image classification datasets and comes at only a low cost in accuracy compared to back-propagation. Both FA and DFA solve the weight transport problem but still suffer from update locking. They are thus no feed-forward-only algorithms as updates have to wait until the full forward pass is complete. The latest feedback-alignment-based training algorithm, *direct random target projection (DRTP)* by Frenkel et al. [22], overcomes this issue by introducing a purely feed-forward algorithm. It is based on DFA and solves both the weight transport and the update locking problem by using the targets instead of the error as feedback signals. Since the targets are already known, no complete forward pass is required, allowing for updates of earlier layers independently of the forward pass in later layers.

Even though biological plausibility is not a necessary requirement for training artificial neural networks, there has been much interest in methods that combine biological plausibility with effective training [76, 59, 8, 74, 46, 47]. In addition, there are incentives for exploring alternatives to back-propagation, even from a purely machine-learning motivated standpoint. Using deeper networks or stronger non-linearities, neural networks compute increasingly non-linear up to almost discrete functions [46]. This can lead to vanishing and exploding gradients, as the gradients are almost zero for most inputs and extremely large at the few points of discrete changes in the function's output. As a result, optimizing such networks with back-propagation can be challenging [46]. Beyond that, back-propagation also exhibits technical constraints, such as the buffering of activations and outputs required due to update locking, which negatively impacts the memory overhead, communication, and energy consumption [53]. Thanks to their reduced power and resource consumption, feed-forward-only training algorithms have a great potential to overcome these limitations in edge computing scenarios and neuromorphic engineering [22, 23].

1.1. Contributions

This thesis introduces *delayed error forward projection (DEFP)*, a DRTP-based feed-forward-only training algorithm for neural networks that solves both the weight transport and the update locking problem. DRTP replaces the gradients with an approximation based on the feedback weights and the targets. With DEFP, I aim to increase the resulting model's accuracy by improving the information used to approximate the gradient while retaining

the biological plausibility and purely feed-forward computation. Due to update locking, a feed-forward-only training algorithm cannot access the current loss value. It can, however, store the loss from a previous epoch. DEFP uses this delayed error information as an additional sample-wise scaling factor in place of the targets to more accurately model the loss. Compared to using only the targets, this can give a more accurate approximation of the current error by including the error magnitude. While there has been discussion on the biological plausibility of delayed errors, especially in combination with non-local errors [53], the argument that the neurons need to store the input for the duration of the delay does not apply to DEFP as the feed-forward nature allows updating each layer immediately. Furthermore, the delayed information could be seen as an additional input, passed from one epoch to the next.

Evaluating DEFP on different neural networks and classification datasets, I find that it can improve the top-1-accuracy by up to 2.02% compared to DRTP while being less than one percent below back-propagation on fully-connected neural networks. This thesis presents two variants of DEFP, using either the gradient of the delayed loss or the delayed error to scale the updates. Surprisingly, error-scaling yields significantly better results than loss-based scaling for all networks and datasets. I presume that loss-scaled updates require a more careful selection of the learning rate or even a learning rate scheduler due to the substantially wider range of potential scaling factors compared to the error or the targets values. Additionally, I explore different initialization approaches for the feedback weights based on distributions typically used for initializing the forward weights of neural networks. Computing the gradient with the chain rule requires the weights and activation derivatives of all downstream layers. With the direct feedback pathways, the feedback weights summarize the whole backward pass. Based on these considerations, I test alternative initialization schemes for the feedback weights. I find only a marginal impact of the initialization method, which further depends on the network topology and dataset at hand. The observed behavior provides no clear indication of a general best choice but suggests that selecting the initialization as a hyper-parameter similarly to the learning rate might yield a slight increase in performance. Comparing the technical performance of the different algorithms, I find that the execution time per epoch differs only slightly on typical network architectures. Only in very deep networks with one thousand layers, back-propagation leads to notably faster epochs than the feed-forward-only approach. This is most likely caused by the overhead of separate update steps in the current implementation compared to the highly optimized backward pass used by back-propagation and might be alleviated using an implementation tuned for fast epoch times. Furthermore, I find that the memory overhead for storing the delayed error information is negligible compared to the overall memory consumption.

1.2. Structure

The remainder of this thesis is structured as follows. Chapter 2 serves as an outline of neural network training, summarizing the definitions and approaches used in later parts of this thesis. An overview of relevant related work is provided in Chapter 3, which discusses why back-propagation is biologically implausible and introduces different approaches

to solving these issues. Additionally, the three predecessors of DEFP, namely FA, DFA, and DRTP, are compared in detail. Chapter 4 introduces DEFP, a weight-transport and update-locking agnostic feed-forward-only training approach for neural networks, and the primary contribution of this thesis. It is evaluated on different datasets and network topologies in Chapter 5, followed by a discussion of the results in Chapter 6. Finally, Chapter 7 summarizes my findings and gives an outlook on possible next steps.

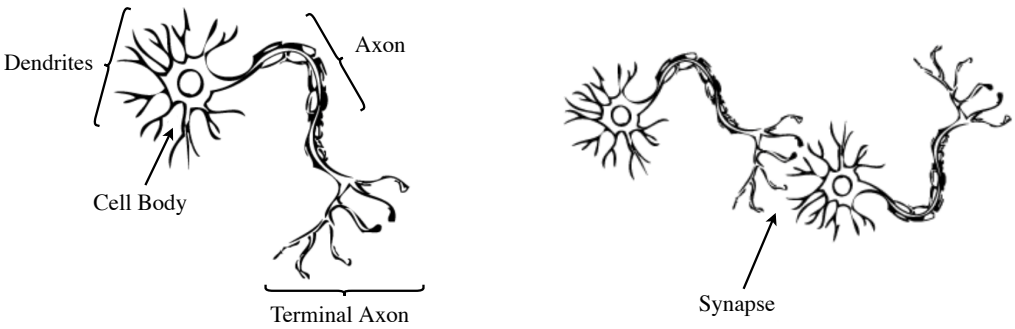
2. Preliminaries

This chapter gives an overview of the preliminary knowledge required to understand the remainder of this thesis, including what a neural network is, how it makes predictions, and what it means to train a neural network. Additionally, Section 2.3 introduces a running example used to compare the different training algorithms in the following chapters.

2.1. Neural Networks

Artificial neural networks are inspired by the human brain. While complex processes within the brain are still not understood in their entirety, the basic functionality can be summarized as follows. The brain consists of a large number of neurons interconnected in a biological neural network. Current estimates suggest that the human brain consists of about 100 billion neurons, each connected to thousands of other neurons via synapses [2, 73]. A neuron receives information—a stimulus—from other neurons via its dendrites. This stimulus is processed by the neuron. If the corresponding output exceeds the action potential threshold, the neuron spikes and emits the action potential via its axon [71, 72]. Figure 2.1 illustrates such a biological neuron.

Like their biological inspiration, artificial neural networks consist of a myriad of interconnected neurons. Each neuron receives inputs from other neurons, processes these inputs, and emits an output based on the result. This output may again be used as input to another neuron. The connections between neurons are weighted to adjust the impact of incoming values on a neuron's output. To compute its output, a neuron takes the weighted



(a) Biological neuron [51] (b) Synapse [51]

Figure 2.1.: Illustration of a biological neuron and a synapse connecting two neurons by Maltarollo et al. [51, Figure 1]. The neuron receives input via its dendrites. These stimuli are then processed within the neuron's cell body, and the output is emitted to another neuron via the axon.

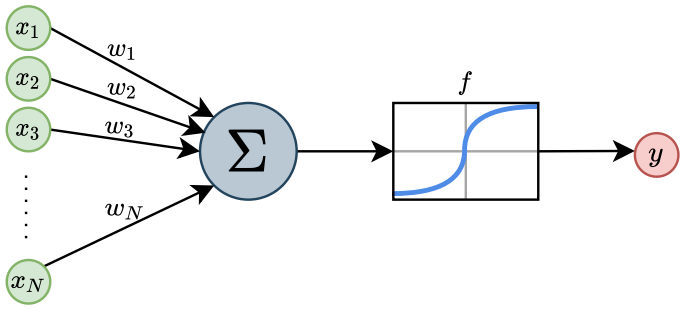


Figure 2.2.: An artificial neuron receives an input x via weighted connections. It computes a weighted sum $\sum_{i=1}^{N} w_i x_i$ and passes it through a non-linear activation function f to generate its output y.

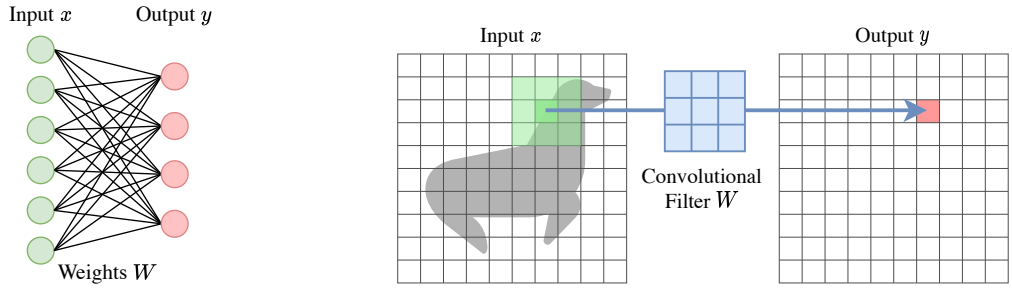
sum of all its inputs and passes it through a non-linear function, the so-called activation function. Figure 2.2 illustrates a simple artificial neuron. Neurons can be organized into *layers* where all neurons of a layer receive their input from the previous layer and pass their output on to the next layer. The input layer receives the input to the network, for example, an input image, while the output layer provides the network's output, such as the detected class. Usually, there are multiple intermediate layers—so-called hidden layers—between the input and the output layer. The *depth* of the network, that is, the number of layers, has an important impact on its performance. The area of deep learning focuses specifically on deep networks with more than one hidden layer. Such networks have been shown to perform exceptionally well on a multitude of tasks as their hierarchical architecture can leverage structures within the data by building a hierarchy of reusable features [25, 44].

This thesis focuses on *feed-forward neural networks* where the connection of neurons is acyclic; that is, the connections of all neurons build a directed acyclic graph (DAG). Information in such a network moves in a single direction from the input layer through the hidden layers to the output layer. In contrast, recurrent neural networks allow connections to neurons of the same or previous layers, making cycles possible. Feed-forward neural networks are not to be confused with the feed-forward-only training discussed in this thesis. In a feed-forward network, the term "feed-forward" refers to the flow of information during the inference, that is, the forward pass through the network. By contrast, feed-forward-only training focuses on eliminating the backward pass typically required to train neural networks with back-propagation.

Different types of layers can be distinguished depending on how the neurons are connected to the previous layer. In this thesis, I focus on two of them, namely fully-connected and convolutional layers. A *fully-connected (FC) layer*, also known as linear layer, connects N input neurons x to M output neurons y through an $M \times N$ weight matrix W. Every input neuron i of the layer is linked to each output neuron j via a connection with the weight w_{ji}. Fully-connected layers are thus defined through their weights W and biases b and compute the linear function

$$y = Wx + b. \tag{2.1}$$

Figure 2.3a illustrates such a fully-connected layer. Since all neurons are mutually connected, independent of their relative position, fully-connected layers have a high number



(a) Fully-connected layer (b) Convolutional layer

Figure 2.3.: Illustration of a fully-connected and convolutional layer. The fully-connected layer (a) connects each input neuron with each output neuron, resulting in a high number of connections and thus parameters W. In contrast, the convolutional layer (b) shifts a filter kernel across the input, reusing the kernel weights W for each position.

of parameters and lack locality. As a result, they cannot differentiate between input coming from neighboring neurons and input from completely unrelated neurons. This increases memory consumption and can inhibit training. The lack of locality is especially detrimental when solving visual tasks, as the interpretation of an input pixel is highly dependent on its surroundings.

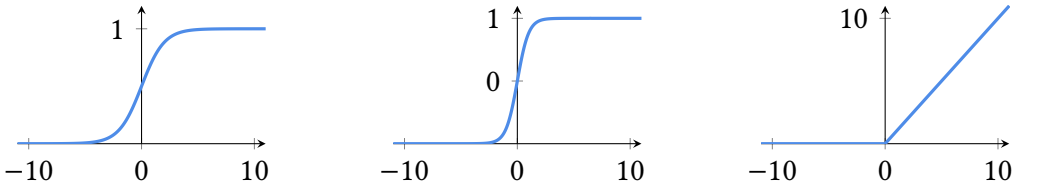
Convolutional layers can solve this problem using a discrete convolution operation. Such a convolution combines the input x to the layer with a so-called kernel K via

$$(x * K)(i) = \sum_{m=-\infty}^{\infty} x(i-m)K(m). \tag{2.2}$$

In neural networks, both the input and the kernel are typically finite, resulting in a finite sum over the available values. The kernel values correspond to the learnable weights W of the layer. The output obtained from applying the kernel to the input is known as feature map [26]. Convolutions for image processing are best explained at the example of applying a two-dimensional convolution to an input image x, as illustrated in Figure 2.3b. The kernel K of size k, typically a small, odd integer, is defined by a $k \times k$ weight matrix W. It is shifted across the input image, computing the weighted sum of the current pixel and its surrounding neighbors in a $k \times k$ patch. Thus, for the pixel at position $[i, j]$, the result is defined as

$$y[i,j] = (x * K)[i,j] = \sum_{m=1}^{k}\sum_{n=1}^{k} x[i-m+a, j-n+a]K(m,n) \quad \text{with} \quad a = \left\lfloor \frac{k}{2} \right\rfloor. \tag{2.3}$$

Convolutional layers offer multiple advantages compared to a fully-connected layer. Since the output value for a pixel is based directly on its neighbors, convolutional layers take account of locality. This is important as it is often less relevant where something is within the image than what surrounds it. Reusing the convolution kernel for different sections of the input, a concept known as weight sharing, not only significantly reduces the number



(a) Sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$ (b) $\tanh(x)$ (c) ReLU $\text{relu}(x) = \max(0, x)$

Figure 2.4.: The sigmoid, hyperbolic tangent, and rectified linear unit activation functions for neural networks.

of parameters but can also improve the model's generalization. The intuition behind this is similar to locality: the same features can occur at different positions in the input; for example, a car may have multiple wheels visible at different pixel locations. A convolutional layer can consist of multiple such kernels, thus producing a number of different activation maps for the same input. Neural networks containing convolutional layers are also referred to as *convolutional neural networks*. Such networks often contain multiple different layer types. For example, typical networks for image classification employ several convolutional layers as feature extractors. Subsequent fully-connected layers generate the network's output by converting the extracted features into the predicted class.

After combining the inputs from the previous layer with a weighted sum, the result is passed through an *activation function*. Without non-linear activation functions, a network of multiple linear layers is just a composition of linear functions, which again is a linear function. Non-linear activations are thus required to utilize multiple layers, to begin with. Different activation functions exist, such as the sigmoid function, the hyperbolic tangent, or the more recently used rectified linear unit [54], illustrated in Figure 2.4. Multiple factors can influence the choice of the activation function. In general, we seek efficient computation and efficient propagation of the results through the network. One particular issue is the *vanishing gradient problem*: If the gradient of the activation function is almost zero for some inputs, it can hinder the gradient propagation to previous layers.

Neural networks can be used to solve a multitude of different tasks. Typically, we differentiate between supervised and unsupervised tasks. For *supervised* tasks, the expected output, the so-called ground truth y^*, is known in advance. This requires labeled training data where each input is labeled with the corresponding ground truth. During the training, these labels can be used to compute the current output error and adjust the model accordingly. An example of a supervised task is *classification*, where we aim to categorize data into one (or more) of multiple previously known discrete classes. Another example is regression, which uses continuous instead of discrete outputs. In contrast, for *unsupervised* tasks, the correct labels are not known to the training algorithm. Training the model thus has to rely on other measures. An example of an unsupervised task is clustering, where the data is grouped into previously unknown clusters, usually aiming to maximize the separation of data points in different clusters while minimizing it within each cluster.

In this thesis, I restrict myself to the classification problem, focusing primarily on image classification for increased comparability to prior works. In *image classification*, the

network receives an input image and determines the correct class depicted in the image. An example would be distinguishing between images of cats and dogs. An image classification *dataset* thus consists of a set of images, each labeled with the correct class. Typically, we separate the dataset into at least a training set and a test set. While the training set is used during the training, the test set remains unseen by the training algorithm and is used to evaluate the trained model. Well-known examples of such datasets are the MNIST [45] and CIFAR-10 [42] datasets, described in more detail in Section 5.1.1, and the large-scale ImageNet [18] dataset.

2.2. Training Neural Networks

Training a neural network is the process of adjusting the model's parameters θ, such as the weights and biases of the layers, to improve its predictive accuracy. It can be formulated as an optimization problem, where we try to minimize the error of the model, typically measured with a loss function, by adjusting its parameters. In a neural network with thousands of parameters and multiple layers, it is non-trivial to identify how much each parameter contributed to the error. This is known as the *credit assignment problem* [52]. By solving the credit assignment problem, we can determine which parameters need to be changed to improve the network's performance. Typically, neural networks are trained with iterative algorithms, alternatingly computing the model's output, evaluating the current error, and updating the parameters accordingly.

The *forward pass* determines the output for a given input sample x by propagating it forward through the network. The input is passed layer-by-layer from the input layer through all hidden layers to the output layer. This process is also known as forward propagation, in accordance with the back-propagation performed by the backward pass. Consider the forward pass for a fully-connected layer i with weights W_i, bias b_i, and an activation function f_i. The layer receives the output h_{i-1} of the previous layer $i-1$ and computes its own output h_i based on h_{i-1}. This output is then passed on to the next layer $i+1$. In the fully-connected layer i, the output is obtained by first passing the input through the linear function defined by the weights W_i and biases b_i, computing the intermediate result z_i as

$$z_i = W_i h_{i-1} + b_i. \tag{2.4}$$

This intermediate result is then passed through the activation function f_i to compute the output h_i of the layer as

$$h_i = f_i(z_i). \tag{2.5}$$

The output $y = h_k$ of the last layer k—the output layer—corresponds to the output of the whole network. Figure 2.5 illustrates this forward pass on a four-layer neural network.

By comparing the output y of the forward pass to the actual label y^*, we can measure how far the model deviates from the desired output. Usually, this error is measured through a *loss function* $J(\theta) = J(y, y^*)$, where θ comprises all trainable model parameters. The training objective is to adjust the model parameters θ such that the total loss over all input samples in the training set is minimized. Multiple different loss functions exist, and the choice of loss typically depends on both the task at hand and the output format of the

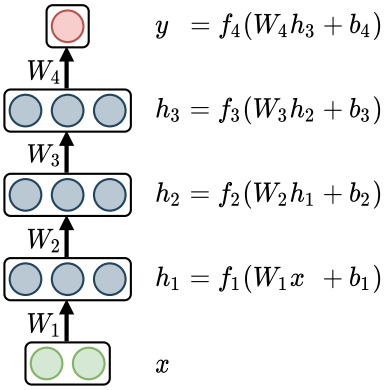


Figure 2.5.: The forward pass in a fully-connected neural network with four layers. The network receives an input $x = h_0$. Each layer i computes its output h_i. The result of the last layer corresponds to the output of the network $y = h_4$.

model. For regression, one might use the mean squared error (MSE), while for classification, categorical cross-entropy loss (CE) or its variant, binary cross-entropy (BCE), is more common [75].

Most training approaches for neural networks utilize gradient-based optimization methods such as gradient descent [13]. To minimize the loss with gradient descent, we need to compute the gradient of the loss with respect to the model parameters. When training neural networks, we typically use the back-propagation algorithm [64] to compute these gradients efficiently without unnecessary re-computations. Back-propagation is based on the chain rule for computing the derivative of function compositions and is explained in more detail in Section 2.2.1. Computing the gradient with back-propagation is done in the *backward pass* after the forward pass has computed the loss.

Having computed the gradients, we can adjust the model parameters in an *optimization step* based on the chosen optimization algorithm. Gradient descent is based on the idea of stepping in the direction of the negative gradient, as this is the direction of the steepest descent. The optimization step adjusts the model parameters θ as

$$\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}, \tag{2.6}$$

where η is the learning rate.

The *learning rate* η is a scaling factor that determines the step size of the update and is typically a small positive scalar. Selecting the right learning rate can have a crucial impact on the training [67, 7, 68, 41]—setting the learning rate too small might lead to slow convergence, whereas choosing it too large can lead to slow convergence, oscillation, or even divergence. The learning rate can be either constant or adjusted dynamically by a *learning rate scheduler* throughout the training. Such a learning rate scheduler can update the learning rate based on the current training state, for example, reducing the learning rate at fixed epochs or updating it dynamically based on the current loss.

In addition to the basic update rule stated above, modern optimization algorithms also utilize additional concepts such as momentum and adaptive learning rates. Section 2.2.2

gives an overview of the different optimization algorithms used in this thesis. Usually, neural networks are trained in multiple *epochs*, where one epoch equals one iteration through all samples of the training set. The training samples can be processed in *batches* of a certain batch size. All samples in a batch are processed simultaneously, and the resulting gradients are accumulated over the samples in the batch.

2.2.1. Back-Propagation and the Chain Rule

The back-propagation algorithm is used to compute the gradients in a multi-layer neural network efficiently. It is based on the *chain rule* for computing the derivative of function compositions, which states that for two univariate functions, f and g, the derivative of their composition $h = f \circ g$, with $y = g(x)$ and $z = h(x) = f(y)$, corresponds to

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{2.7}$$

This can be generalized to vectors and even tensors of arbitrary dimensionality. The derivative of the value z with respect to a tensor X is then defined as

$$\nabla_X z = \sum_j \left(\nabla_X Y_j\right) \frac{\partial z}{\partial Y_j} \tag{2.8}$$

with $Y = g(X)$ and $z = f(Y)$ [26].

In a neural network, we need to compute the gradient of the loss value $J(\theta)$, obtained from the forward pass, with respect to all model parameters θ. The gradient of the loss with respect to a value x is often also denoted as

$$\delta x = \frac{\partial J(\theta)}{\partial x}. \tag{2.9}$$

Applying the chain rule to the fully-connected layer i from the example above yields the following gradients for the intermediate result z_i, the input from the previous layer h_{i-1}, the weights W_i, and the bias b_i:

$$\delta z_i = \frac{\partial J(\theta)}{\partial z_i} = \frac{\partial J(\theta)}{\partial h_i}\frac{\partial h_i}{\partial z_i} = \delta h_i \odot f_i'(z_i) \tag{2.10}$$

$$\delta h_{i-1} = \frac{\partial J(\theta)}{\partial h_{i-1}} = \frac{\partial J(\theta)}{\partial z_i}\frac{\partial z_i}{\partial h_{i-1}} = W_i^T \delta z_i \tag{2.11}$$

$$\delta W_i = \frac{\partial J(\theta)}{\partial W_i} = \frac{\partial J(\theta)}{\partial z_i}\frac{\partial z_i}{\partial W_i} = \delta z_i h_{i-1}^T \tag{2.12}$$

$$\delta b_i = \frac{\partial J(\theta)}{\partial b_i} = \frac{\partial J(\theta)}{\partial z_i}\frac{\partial z_i}{\partial b_i} = \delta z_i, \tag{2.13}$$

where $\odot$ is the element-wise product, also known as Hadamard product. Figure 2.6 illustrates the information flow through a fully-connected layer i for both the forward and the backward pass.

As is already apparent from the example above, there can be subexpressions that are required multiple times to compute all gradients in the network. In this example,

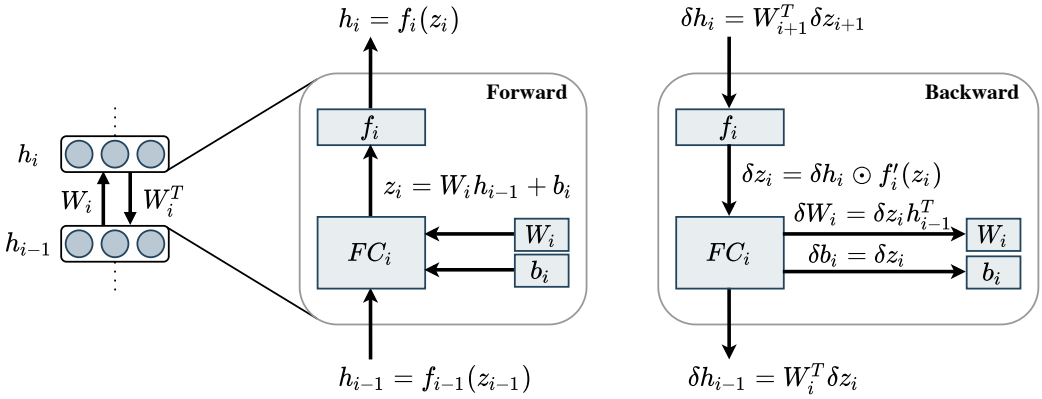


Figure 2.6.: A detailed look into the forward and backward pass within a fully-connected layer i with weights W_i, bias b_i, and activation function f_i.
Forward Pass: The layer receives the output h_{i-1} of the previous layer $i-1$. Passing it through the linear function FC_i creates the intermediate output z_i, which is passed through the activation function f_i to compute the output h_i of this layer i. The output h_i is then passed forward to the next layer $i+1$.
Backward Pass: The layer receives the gradient δh_i from the next layer $i+1$. The gradients δz_i and therefrom δW_i, δb_i, and δh_{i-1} are computed by applying the chain rule. The gradient δh_{i-1} is passed backward to the previous layer $i-1$, which computes its gradients accordingly.

the gradient δz_i is required to compute the gradients for the input from the previous layer, the weights, and the biases. There can be an exponential amount of such repeated subexpressions, making an efficient implementation necessary for large networks [26]. The *back-propagation algorithm (BP)* [64] is such an implementation of the chain rule. It focuses on computing the gradients in an efficient order and reusing repeated subexpressions in later computations instead of recomputing them. Back-propagation computes the gradients in precisely the reverse order of the forward pass. This means it starts at the loss function and proceeds layer-by-layer backward through the network up to the first layer. With this approach, all gradients need to be computed only once. Consider, for example, the gradient δz_i of the loss with respect to z_i as given by Equation (2.10). Computing it requires the gradient δh_i of the loss with respect to h_i and the gradient of h_i with respect to z_i. Since δh_i has already been computed in the previous step, it can be reused, and only $\partial h_i / \partial z_i$, the gradient of h_i with respect to z_i, needs to be computed in this step.

The *backward pass* computes the gradients for all model parameters θ in this manner with back-propagation. When training a neural network, forward and backward passes are performed alternatingly, first computing the loss with the forward pass, then the gradients in the backward pass. Algorithm 1 describes training a neural network with back-propagation in more detail. Processing a batch containing the samples with indices b, the forward pass (Lines 5 to 7) determines the output $y = h_k$ for the input $x = h_0$ with a forward propagation through the layers L_i of the model. After computing the loss,

Algorithm 1: Training a model M with layers $L_1, \ldots, L_k$ on a dataset D with the optimizer Opt for $t_{\max}$ epochs using back-propagation.

Input: model $M = (L_1, \ldots, L_k)$, optimizer Opt, dataset D, epochs $t_{\max}$

```
1  for t ← 1 to t_max do
2      foreach b ∈ D.batches do
3          h_0 ← D.x[b]
4          y* ← D.y*[b]
5          for i ← 1 to k do
6              h_i ← L_i.forward(h_{i−1})
7          end
8          loss ← J(h_k, y*)
9          δh_k ← ∂loss/∂h_k
10         for i ← k to 1 do
11             δh_{i−1} ← L_i.backward(δh_i)
12         end
13         Opt.update_step(M)
14     end
15 end
```

the backward pass (Lines 10 to 12) processes the layers in reverse order, computing the gradients of the loss with respect to all model parameters.

2.2.2. Optimization Algorithms

After computing the gradients with back-propagation, the parameters are updated according to the chosen optimization algorithm. Section 2.2 already introduced gradient descent, a first-order optimization algorithm that steps in the direction of the steepest descent, that is, in the direction of the negative gradient. There are several approaches to further accelerate the optimization compared to the general gradient descend step. This section introduces four popular optimization algorithms: Stochastic gradient descent, Nesterov's Accelerated Gradient, RMSProp, and Adam.

In the context of neural networks, we can differentiate between batch, mini-batch, and stochastic gradient descent. In batch gradient descent, all training examples are processed simultaneously. This corresponds to a batch size equal to the number of training samples. *Stochastic gradient descent (SGD)* selects training samples at random without replacement. In its original form, SGD uses only a single training sample at a time, corresponding to a batch size of one. Mini-batch gradient descent lies between these two extremes. It uses batches containing more than one but not all the training samples. Today, mini-batch and stochastic gradient descent are often used interchangeably. Choosing the batch size depends on multiple factors. While larger batches offer a more accurate gradient estimate, smaller batches can positively impact the generalization through a regularization effect [77, 39, 26]. Additionally, there are technical constraints: while larger batch sizes usually speed up the training time per epoch, the maximum batch size is limited by the available memory.

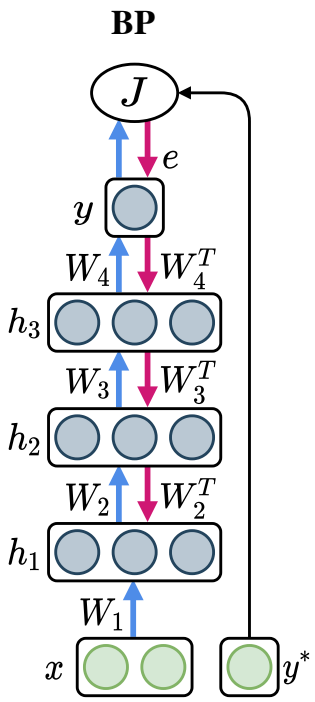


Figure 2.7.: Example of back-propagation on a four-layer network. In the forward pass (blue), the input x is passed through the network layer-by-layer until the output layer generates the output y. The output is compared to the target y^* using the loss function J. To compute the gradient of the loss with respect to each weight, the backward pass (red) processes the layers in reverse order compared to the forward pass.

Momentum [62] is a method to speed up the training by considering the current momentum of the gradient in the update step. The moving average of the gradient with exponential decay is accumulated as velocity and used to update the parameters instead of the gradient itself. If the gradients remain largely the same for multiple steps, momentum speeds up the training by effectively increasing the step size. If the direction of the gradient varies wildly, momentum avoids a zigzag course by averaging over the past gradients, which also automatically reduces the step size. A variant of momentum is *Nesterov's Accelerated Gradient (NAG)* [55, 69] which evaluates the gradient after applying the current velocity instead of before.

Another approach is using *adaptive learning rates*. This introduces a separate learning rate for each parameter instead of a single combined value for all parameters and adapts these learning rates automatically based on the current state of the training. *RMSProp* [30] again accumulates the gradient in an exponentially weighted moving average. This gradient is then used to decrease the learning rate of the parameters based on the magnitude of the corresponding part of the gradient. For parameters with a large gradient, the learning rate is decreased more than for parameters with a relatively small gradient. This speeds up

the training in the shallower directions of the parameters space compared to the steeper directions. *Adam* [40] is another optimization algorithm based on adaptive learning rates.

2.3. Running Example

Throughout this thesis, a fully-connected neural network with four layers is used as a running example to compare the different training algorithms. Figure 2.7 illustrates training this neural network with back-propagation. For simplicity, the biases are omitted. This is without loss of generality, as the bias can be encoded in the weights by applying the bias-trick and changing the input h_{i-1} to $h_{i-1}^{+} = [h_{i-1}, 1]$ and the weights to $W_i^{+} = [W_i, b_i]$. In the forward pass (blue), information flows forward through the layers from the input x to the output y based on the forward weights W_i of each layer. The loss is computed by comparing the actual output y to the expected output or target y^*. In the backward pass (red), the gradient information flows backward layer-by-layer based on the transpose of the forward weights W_i^T. The derivative of the i-th layer's output h_i thus corresponds to

$$\delta h_i = W_{i+1}^T \delta z_{i+1} \tag{2.14}$$

as shown in Section 2.2.1. The computations within each fully-connected layer are summarized in Figure 2.6.

3. Related Work

This chapter gives an overview of prior work related to feed-forward-only training. First, Section 3.1 discusses why training neural networks with back-propagation is biologically implausible. Then, Section 3.2 introduces different approaches to biologically inspired training. Finally, Section 3.3 discusses feedback alignment and derived algorithms in detail as the foundation for the approach introduced in the subsequent chapter.

3.1. On the Biological Plausibility of Back-Propagation

While the back-propagation algorithm is very successful in training artificial neural networks to a high degree of accuracy, it is improbable that the same mechanism is used for learning within the human brain. Back-propagation relies on multiple factors that are, with our current understanding, thought to be impossible to implement in the brain, making back-propagation biologically implausible. An overview of these factors is given, for example, by Bengio et al. [8] and Hunsberger [33]. In addition to the biological plausibility, some of these factors also affect the computational performance of back-propagation, such as memory usage or execution time. This section focuses primarily on the weight transport and the update locking problem. Further biologically implausible factors are discussed briefly at the end of this section.

3.1.1. Weight Transport Problem

Recalling the definition of the forward and backward path through a fully-connected layer from Chapter 2,

$$h_i = f(W_i h_{i-1} + b_i) \qquad \text{forward} \tag{3.1}$$

$$\delta h_{i-1} = W_i^T \left(\delta h_i \odot f_i'(z_i)\right) \qquad \text{backward,} \tag{3.2}$$

we notice that the weights W_i are used in both the forward and the backward steps. This symmetry of forward and backward weights is biologically implausible because synapses are unidirectional [33], meaning a synapse is never used in both the forward and the backward path. Using separate pathways—one for the forward pass and one for the backward pass—is not plausible either, as this would require a synchronization or transportation of the weights between the two paths [27, 33, 22]. This issue is known as the *weight transport problem* [27], also called the weight symmetry problem [47], and is illustrated in Figure 3.1.

Besides being biologically implausible, the weight transport problem can also negatively impact the hardware efficiency of back-propagation. Following the chain rule, the gradient

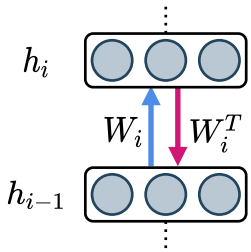


Figure 3.1.: The weight transport problem: Back-propagation uses the same weights W_i in both the forward (blue) and the backward pass (red). This is not biologically plausible as synapses are unidirectional.

δh_{i-1} depends not only on the forward weights of layer i but also on the forward weights of all downstream layers via the gradient δh_i. This non-locality of the weights constrains the memory access patterns, as computing the update for one layer requires reading the weights of all downstream layers. Especially on non-von-Neumann architectures with in-memory computing, this can severely impact the computational efficiency [15, 22].

3.1.2. Update Locking Problem

To compute the gradients in the backward pass, the training algorithm first needs to finish the complete forward pass. Consequently, every layer has to wait until the forward pass has processed all succeeding layers to compute the output and the backward pass has returned through all downstream layers. This problem is known as the *update locking* [36] or timing problem [33] and is illustrated in Figure 3.2. It is biologically implausible, as a full forward and backward pass requires a significant amount of time. This applies to both artificial neural networks as well as the neural connections in the brain. The activity of the layer might have already changed when the backward pass reaches it again. This can be a critical problem, especially for earlier layers with a considerable distance to the output layer, and thus a high delay between forward and backward pass. As a result, the activation becomes out of sync with the error signal [33].

When training with back-propagation, there is a clear separation between the forward and backward passes, and no forward pass starts before the previous backward pass has been completed. Thus, update locking might not lead to out-of-sync problems, yet it still impacts the training of artificial neural networks. For one, this clear separation between forward and backward passes might not be desirable as it restricts computation and prevents pipelining. On top of that, it can lead to memory and communication overheads, which can also impact the energy consumption, as explained in detail by Mostafa et al. [53]. Since the backward pass requires both the input h_{i-1} to the layer and the derivative of the activation function $f_i'(z_i)$, the corresponding data needs to be buffered from the forward pass. This buffering would not be necessary without update locking as the updates could be performed immediately after finishing the forward pass on the current layer [53, 22].

Furthermore, when training on a graphical processing unit (GPU), update locking can cause GPU under-utilization. In modern GPU computing, the CPU can offload computation to the GPU by asynchronously enqueuing kernel execution requests in a workload queue,

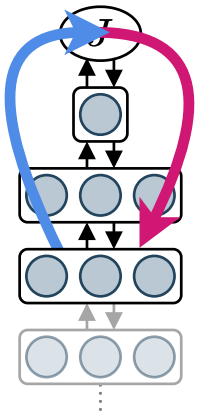


Figure 3.2.: The update locking problem: To update a layer, the training algorithm first needs to finish the complete forward pass (blue), compute the loss, and return to the layer in the backward pass (red). This is biologically implausible as it would cause a delay between the forward step and the update of a single layer, potentially leading to desynchronization.

the so-called CUDA stream [58]. The GPU processes requests from these streams and multiple requests can be performed in parallel if they do not depend on each other and do not fully utilize the GPU on their own. The more requests are queued, the more scheduling options exist to optimize the GPU utilization. Since the requests are enqueued asynchronously, the CPU can continue computation unless it requires the result of a kernel execution. Without update locking, the backward pass of a layer could start immediately after its forward pass has been completed and would not need to wait for later layers. The computations required for the update could thus be queued in parallel to the forward pass for the next layer, improving the overall GPU utilization, especially on smaller networks where a single layer does not utilize the GPU to its capacity.

3.1.3. Further Implausibilities

In addition to the weight transport and update locking problems discussed above, there are further issues impacting the biological plausibility of back-propagation. This section briefly discusses some of these problems but is not meant as an exhaustive list. The derivative transport problem [33] is due to back-propagation requiring the derivative of the activation $f_i'(z_i)$, used to modulate the error signal with

$$\delta z_i = \delta h_i \odot f_i'(z_i), \tag{3.3}$$

which is then passed on backward through the network. However, it is unclear how the brain could compute this derivative or use it to modulate the error signals [8, 33]. The linear feedback problem [33] relates to the fact that back-propagation has a purely linear feedback path, while biological neurons are non-linear [8, 33]. It is known that biological neurons are spiking; that is, they encode information into the frequency and timing of binary pulses [17]. Artificial neural networks, on the other hand, traditionally

use continuous values, typically between −1 and 1. This difference is known as the spiking problem [8, 33]. The area of spiking neural networks tries to address this problem and is discussed briefly in Section 3.2. Furthermore, back-propagation violates Dale's Law [20], stating that a neuron outputs either excitatory or inhibitory signals but never both [35]. In the context of artificial neural networks, this corresponds to the outgoing weights of a neuron [72]. Since the outgoing weights can be both positive and negative and can even change their sign during training, back-propagation conflicts with Dale's Law [4, 33].

A frequent problem in supervised learning is obtaining a sufficient amount of labeled data. Back-propagation often requires millions of labeled examples to generalize to unseen inputs. In contrast, the human brain can learn from only a handful of labeled examples. This issue was coined the target problem by Hunsberger [33]. The brain might solve this problem by combining a large amount of unlabeled data with a few labeled examples, yet it is still unclear how exactly this semi-supervised learning would work in the brain [8, 33].

3.2. Biologically Inspired Neural Networks

As discussed in the previous section, back-propagation is biologically implausible. While this is not a necessary requirement for training artificial neural networks, there has been much interest in methods that combine biological plausibility with effective training [76, 59, 8, 74, 46, 47]. This section gives an overview of different approaches to more biologically inspired artificial neural networks, tackling some of the issues described in Section 3.1.

Spiking neural networks are inspired by the fact that biological neurons like those in the human brain use discrete spikes instead of continuous outputs as used by standard artificial neural networks. They encode information into the spikes via their timing and frequency [72]. Besides being more biologically plausible, spiking neural networks are also more efficient by using fewer operations and less energy. However, their non-differentiable nature makes training with traditional back-propagation difficult [74, 71]. Instead, multiple alternative training approaches exist, including unsupervised learning, for example, with spike timing-dependent plasticity (STDP) [12], and supervised methods like SpikeProp [9] and ReSuMe [38]. An overview of different training algorithms for spiking neural networks is given by Tavanaei et al. [74].

To solve the weight transport problem, *target propagation* by Lee et al. [46] replaces the loss gradients with layer-wise target values computed by auto-encoders. Using an additional linear correction of these auto-encoders called difference target propagation, they reach results comparable to back-propagation on deep networks. On stochastic neural networks, they even achieve state-of-the-art performance [46]. Ororbia and Mali [59] take a similar approach, but instead of generating the layer-wise targets with auto-encoders, they employ the pre-activation h_i of the current layer and the post-activation z_{i+1} of the next layer. Further solutions to the weight transport problem include the FA [48] and DFA [56] algorithms described in the following section.

Multiple algorithms solving update locking have been proposed. Mostafa et al. [53] use an approach based on *local errors* generated for each layer with the help of fixed random classifiers. These auxiliary classifiers receive the output of a layer and generate a local error.

With this strategy, they outperform FA and approach near back-propagation performance. Improving upon this, Nøkland and Eidnes [57] demonstrate that combining local classifiers with a so-called local similarity matching loss can close the gap to back-propagation. With *decoupled greedy learning (DGL)*, Belilovsky et al. [5] follow a similar approach based on greedy objectives, which even scales to large-scale datasets such as ImageNet [18]. They focus specifically on the parallelization of the training, including an extension to asynchronous settings.

Another approach to solving update locking uses *synthetic gradients*, introduced by Jaderberg et al. [36] and Czarnecki et al. [16]. Synthetic gradients model a subgraph of the network and predict its future output based on only local information. By replacing back-propagation with these synthetic gradients, they decouple the layers, resulting in so-called *decoupled neural interfaces (DNIs)* and solving the update locking problem. Using the same approach, they can also predict inputs and thus solve the *forward locking problem* [36] by employing a synthetic input signal.

Approaching biologically inspired computing from the hardware perspective, *neuromorphic devices* are a novel hardware architecture inspired by the human brain emulating the interactions of biological neurons and synapses with CMOS integrated circuits [72]. With Moore's law slowing down due to power constraints, we can no longer expect similar increases in the efficiency of devices as in the past. On top of that, the von-Neumann-bottleneck caused by physically separating memory and data processing increases the cost of memory movements and is further intensified by the increasing gap in the performance of memory and processors, known as the memory wall [72, 34]. These developments are some of the main contributing factors in the recent interest in neuromorphic devices. With massive parallelization, these devices could significantly accelerate the training of neural networks but also come with several challenges, such as complex architectures with high energy consumption and required chip area [72]. Frenkel et al. [23] demonstrate how biologically plausible algorithms like DRTP can be deployed on neuromorphic devices for low-cost adaptive edge computing.

3.3. Feedback Alignment and Related Approaches

This section introduces the feedback alignment algorithm and two derived approaches in detail. First, Section 3.3.1 discusses feedback alignment by Lillicrap et al. [48], demonstrating that symmetric feedback weights are not necessary to train neural networks and thus solving the weight transport problem. Subsequently, Section 3.3.2 introduces direct feedback alignment by Nøkland [56], which replaces the feedback paths by propagating the error directly to the hidden layers. Section 3.3.3 then discusses direct random target projection by Frenkel et al. [22], which further improves upon direct feedback alignment by resolving the update locking problem. Finally, Section 3.3.4 summarizes the differences between the three approaches. As this thesis is based primarily on direct random target projection, these three approaches can be seen as direct predecessors to the training algorithm introduced in Chapter 4.

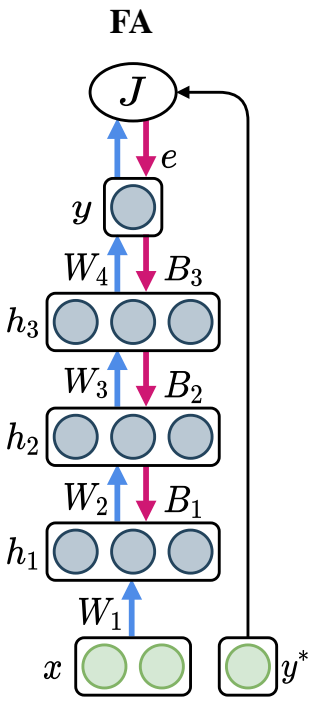


Figure 3.3.: Training the neural network introduced in Section 2.3 with FA. In contrast to back-propagation, FA uses fixed random feedback weights B_i on the backward path (red). Since these feedback weights are independent of the forward weights W_{i+1}, FA solves the weight transport problem. The forward path (blue) remains unchanged.

3.3.1. Feedback Alignment

Feedback alignment (FA) by Lillicrap et al. [48] solves the weight transport problem by replacing the transposed forward weights W_{i+1}^T in the backward pass with fixed random feedback weights B_i. Lillicrap et al. show that symmetric backward weights are not necessary and that these fixed random feedback weights are sufficient for effective training. Figure 3.3 illustrates FA on the running example introduced in Section 2.3. Feedback alignment is based on the idea that we can use any feedback weights B_i as long as the training signal δh_i—also denoted the modulator or modulatory signal—is within 90° of the actual training signal prescribed by back-propagation. In other words, for the training signals defined by FA

$$\delta^{\text{FA}} h_i = B_i \delta z_{i+1} \tag{3.4}$$

and back-propagation

$$\delta^{\text{BP}} h_i = W_{i+1}^T \delta z_{i+1} \tag{3.5}$$

holds

$$\left(\delta^{\text{BP}} h_i\right)^T \delta^{\text{FA}} h_i = \delta z_{i+1}^T W_{i+1} B_i \delta z_{i+1} > 0. \tag{3.6}$$

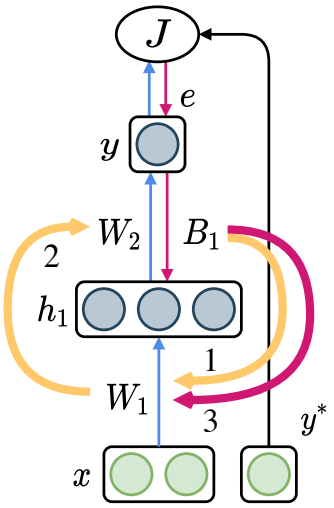


Figure 3.4.: A two-layer network demonstrating how the forward weights W_2^T come into alignment with the feedback weights B_1, based on [48, Figure 5b]. In phase one, information flows from B_1 to W_1. In phase two, this information is used to adjust W_2. In phase three, the forward and backward weights are aligned, allowing effective training.

The intuition behind this condition is that if the direction of the training signal is within 90° of the correct direction, the training still moves in roughly the same direction as back-propagation would.

While this condition is sufficient to train neural networks, better alignment is necessary for faster training. To achieve this and increase the alignment, one can either adjust the feedback weights B_i or the forward weights W_{i+1}. With FA, Lillicrap et al. [48] opt for adjusting the forward weights while keeping the feedback weights fixed. When training a neural network with FA, the forward weights are modified to achieve a soft alignment with the feedback weights. This means that the modulator signals used by FA become more similar to those used by back-propagation, and the angle between them decreases. Since the angle never reaches zero, the authors conclude that exact symmetry between the forward and feedback weights is not necessary.

To illustrate how the forward and feedback weights can come to an alignment without direct communication, Lillicrap et al. [48] demonstrate how information can flow from the feedback weights to the forward weights via previous layers. While there is no direct communication between the forward weights W_{i+1} and the feedback weights B_i, the feedback weights B_i can influence the forward weights W_{i+1} indirectly over the course of multiple iterations via the weights of the previous layer W_i. Information first moves from B_i into W_i via the update rule

$$\Delta W_i \propto \delta^{\mathrm{FA}} z_i h_{i-1}^T = \Big(\delta^{\mathrm{FA}} h_i \odot f_i'(z_i)\Big) h_{i-1}^T = \Big(B_i \delta^{\mathrm{FA}} z_{i+1} \odot f_i'(z_i)\Big) h_{i-1}^T. \tag{3.7}$$

This information can then move from W_i into W_{i+1} via

$$h_i = f_i(W_i h_{i-1} + b_i) \tag{3.8}$$

and the update rule

$$\Delta W_{i+1} \propto \delta z_{i+1} h_i^T. \tag{3.9}$$

Lillicrap et al. [48] were able to visualize this process in a two-layer network by artificially breaking the training into three phases and alternately freezing the forward weights W_1 and W_2. By monitoring the error throughout the three phases, they illustrate how the forward weights come to an alignment with the fixed feedback weights. This process is also illustrated in Figure 3.4. In phase one, only W_1 is trained while W_2 remains frozen. The information from B_1 is accumulated in W_1. The overall error remains unchanged as W_2^T and B_1 are not yet aligned. In phase two, W_1 is frozen while W_2 is adjusted. The information from B_1, previously collected in W_1, can now flow into W_2, bringing it in alignment with B_1. The overall error improves as the two weight matrices begin to align. Finally, in phase three, W_1 is trained again while W_2 is frozen. Since W_2^T and B_1 are now aligned, W_1 can learn effectively from the feedback signals via B_1, further reducing the error. This separation into training phases is only for demonstration purposes. The three phases happen simultaneously when training a neural network with FA, and it is not necessary to freeze any layers.

3.3.2. Direct Feedback Alignment

Feedback alignment solves the weight transport problem, yet the training signals remain non-local, meaning the gradient estimates are propagated backward layer-by-layer through the hidden layers. As discussed in Section 3.1, this is biologically implausible and comes with additional constraints on the memory access pattern. *Direct feedback alignment (DFA)* by Nøkland [56] builds upon FA to solve this problem by propagating the error directly from the output layer to each hidden layer. Like FA, DFA uses fixed random feedback weights B_i instead of the transpose of the forward weights W_{i+1}^T to compute the updates. However, instead of propagating the feedback information successively through all layers, the error e is passed directly to the hidden layers. This results in the following training signals

$$\delta^{\mathrm{DFA}} h_i = B_i e \tag{3.10}$$

and is illustrated in Figure 3.5. Compared to FA, the feedback pathways through the downstream layers are no longer necessary.

Nøkland [56] identifies some necessary constraints and helpful conditions for efficient training with DFA. The feedback weights B_i should be selected randomly, as this yields a very high probability of having full rank. If B_i has full rank, the training signals $\delta^{\mathrm{DFA}} h_i$ are non-zero as long as the error e is non-zero. As shown by Nøkland [56], this is necessary to achieve an alignment angle within 90°. Fixed feedback weights offer multiple advantages over variable feedback weights. First, with B_i being constant, its rank also remains constant, preserving the fact that $B_i e$ is non-zero. Furthermore, it keeps the direction of $\delta^{\mathrm{DFA}} h_i$ more consistent, which additionally helps with the alignment. Finally, when solving a task with target labels in $\{0, 1\}$, for example, classification with one-hot encoding and cross-entropy loss, the sign of the error $\mathrm{sign}(e_j)$ is constant for each sample j. Thus, $B_i \mathrm{sign}(e_j)$ is constant for each sample j, and the magnitude of the update direction $\delta^{\mathrm{DFA}} h_i$

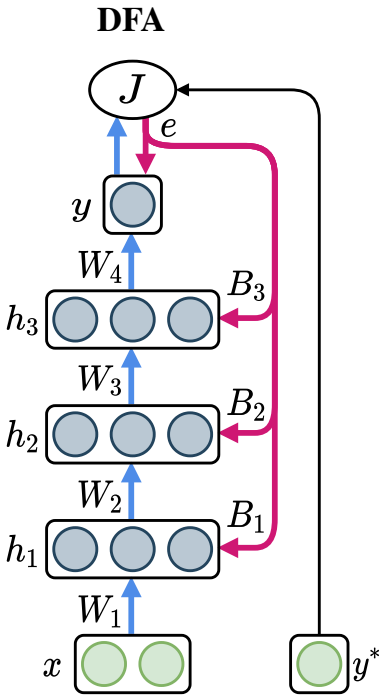


Figure 3.5.: Training the neural network introduced in Section 2.3 with DFA. In contrast to FA, illustrated in Figure 3.3, DFA uses direct feedback pathways (red) to pass the loss directly to the hidden layers instead of propagating it backward layer-by-layer. As for FA, the forward pass (blue) remains unchanged.

varies only with the magnitude of the error. In the case of classification, $B_i \text{sign}(e_j)$ is even constant for all samples of a class.

Direct feedback alignment does not depend on a sophisticated initialization of the forward weights. In fact, zero-initialized weights offer a good starting point as they result in exactly orthogonal modulator signals

$$e^T W_{i+1} B_i e = e^T \mathbf{0} B_i e = 0, \tag{3.11}$$

which helps to align the weights quickly. Zero-initialization is, however, not necessary for effective training. Nøkland [56] demonstrates experimentally that DFA can even recover from particularly bad initializations. Zero-initialized weights would, however, not work in conjunction with an activation function like the rectified linear unit, where the gradient at zero is zero, resulting in update steps of zero and thus no training progress.

Even without zero-initialization, the rectified linear unit is not necessarily a good fit for DFA. In training algorithms without direct feedback pathways, such as back-propagation or FA, the hidden layers are implicitly bounded by the layers above them. If the training signal δh_{i+1}, coming from the next layer, is zero, the signal for the current layer

$$\delta h_i = W_{i+1}^T \delta z_{i+1} = W_{i+1}^T \left(\delta h_{i+1} \odot f'_{i+1}(z_{i+1}) \right) \tag{3.12}$$

becomes zero, too. This also holds for FA with $W_{i+1}^T = B_i$. However, due to the direct feedback, this does not apply to DFA. As already established previously, the training signals

$\delta^{\text{DFA}} h_i$ are non-zero as long as the error e is non-zero. To limit the growth of the hidden layers, Nøkland [56] suggests using a squashing activation function such as the hyperbolic tangent or sigmoid. Being a zero-centered function, the hyperbolic tangent works best, especially in combination with zero-initialization.

Finally, networks can only be trained effectively by DFA if the whole network is trained collectively. This is related to the intuition on FA, explained briefly in Section 3.3.1. With W_2 remaining frozen, the training remains in phase two. The feedback weights B_1 will never align with W_2 and W_1 cannot be trained effectively. This is in contrast to back-propagation, where arbitrary layers may be frozen, and the remaining layers can still be trained.

Nøkland [56] evaluates DFA on the MNIST [45] and CIFAR [42] datasets. The author demonstrates that DFA achieves near back-propagation performance even without weight initialization. Even very deep networks can be trained with zero-initialization while back-propagation and FA struggle without a sophisticated weight initialization scheme. However, DFA has difficulties training convolutional networks where it results in noticeably higher error rates on the test set. This might be related to the bottleneck training of DRTP discussed in Section 3.3.3.

3.3.3. Direct Random Target Projection

Direct random target projection (DRTP) by Frenkel et al. [22] aims to solve not only the weight transport problem but also the update locking problem. It is based on DFA but uses the targets y^* instead of the error e. The key advantages of DRTP are increased biological plausibility and decreased hardware requirements. While the final test error rate tends to be slightly higher than for previous approaches, DRTP performs significantly better than shallow learning, demonstrating its ability to train multi-layer networks. This shows that the weight transport and the update locking problem are solvable at the expense of a slightly decreased accuracy. Solving both of these problems makes DRTP biologically more plausible than back-propagation. Not suffering from update locking, DRTP is a feed-forward-only training algorithm, as all information required to update a layer is available immediately after the forward pass through that layer. In contrast, both FA and DFA need to finish the complete forward pass to compute the error and then communicate it to the hidden layers.

Additionally, DRTP also decreases the computational and memory demands. Solving the weight transport problem relaxes constraints on the memory access patterns, which can improve performance, especially on non-von-Neumann architectures [15, 22] but also on standard hardware for sufficiently large networks [53]. Solving the update locking problem decreases the memory overhead, as it is no longer necessary to buffer the inputs and activations for all layers computed in the forward pass until the backward pass performs the weight updates. Furthermore, the approximate gradients can be computed very efficiently. In contrast to approaches like error locality and synthetic gradients, no side networks are required. Compared to DFA, the multiplication of the error with the weights can be replaced by simply selecting the corresponding entry in the feedback weights based on the target.

Frenkel et al. [22] first introduce *sDFA*, a variant of DFA using the sign of the error instead of the actual error. The modulatory signal for sDFA thus corresponds to

$$\delta^{\text{sDFA}} h_i = B_i \text{sign}(e). \tag{3.13}$$

The sDFA approach can be seen as an intermediate step between DFA and DRTP. The authors demonstrate experimentally that sDFA can train multi-layer networks on both regression and classification tasks and that the modulatory signals remain within 90° of back-propagation.

Comparing sDFA to other training algorithms, they make the following observations. In general, sDFA performs worse than the other tested training algorithms, yet still significantly better than shallow training of only the output layer. A potential problem of sDFA is that the error's magnitude is lost when using only its sign. Usually, the magnitude of the error would decrease over the course of the training, automatically decreasing the size of the update steps. The authors observe this effect especially on the regression dataset, where sDFA stagnates earlier than the other approaches. Frenkel et al. [22] thus suggest testing a learning rate scheduler. An advantage of the direct feedback pathways is that they protect both DFA and sDFA against vanishing gradients, while both back-propagation and FA are affected by these. This aligns with the observations on activation functions by Nøkland [56] discussed in Section 3.3.2. While there are effective approaches against vanishing gradients, such as activation functions like the rectified linear unit or batch normalization, Frenkel et al. [22] argue that algorithms based on direct feedback pathways need no additional mitigation techniques, thus reducing the hardware requirements.

Having shown the error sign to be sufficient for training with sDFA, they demonstrate that for classification, the error sign can be deduced directly from the one-hot encoded class labels as long as the output is strictly bounded between 0 and 1. This condition can be met by, for example, using a sigmoid or softmax activation function in the output layer. Under these constraints, the targets correspond to rescaling and shifting the error sign. Frenkel et al. [22] argue that the rescaling can be included in the fixed random feedback weights and that the shift operation is actually beneficial to the training. First, the targets improve the computational efficiency. While the error sign can take the value ± 1, the targets y^* are always 1 for the correct class and 0 for all other classes. Thus, the approximate gradient

$$\delta^{\text{DRTP}} h_i = B_i y^* \tag{3.14}$$

can be computed with a label-dependent selection from the feedback weights instead of a multiplication. Second, using the targets instead of the error sign improves the training results, as experiments show that DRTP systematically outperforms sDFA on both the MNIST and the CIFAR-10 dataset. According to Frenkel et al. [22], this is caused by the $C - 1$ incorrect classes outweighing the single correct class when the magnitude of the error is the same for all classes and only the sign changes. In contrast, with the targets y^*, incorrect classes are assigned 0, thus reducing their impact compared to the correct class. Figure 3.6 summarizes the DRTP training approach.

Note that since both the targets y^* and the feedback weights B_i are fixed and do not change over the course of the training, the estimated gradient $\delta^{\text{DRTP}} h_i$ is constant throughout the training and for each sample of a class. Changes to the estimated gradients for

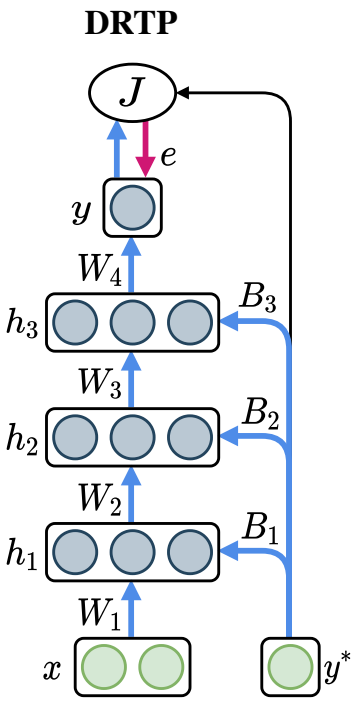


Figure 3.6.: Training the neural network introduced in Section 2.3 with DRTP. Compared to DFA, the backward pass is no longer required since DRTP uses the targets y^* instead of the error e to train the hidden layers. Thus, the hidden layers can be trained during the forward pass (blue). This resolves update locking and makes DRTP a purely feed-forward algorithm.

the parameters δW_i and δb_i thus depend only on the output of the previous layer h_{i-1} and the derivative of the activation function $f_i'(z_i)$. This is best illustrated in Figure 3.7: With DRTP, δh_i is constant for each class; thus, δz_i changes only with $f_i'(z_i)$.

Frenkel et al. [22] extend the mathematical proof by Lillicrap et al. [48] to show that the modulatory signals of DRTP on multi-layer networks with linear hidden layers and a non-linear output layer are always within 90° of those prescribed by back-propagation. This guarantees that DRTP reaches a soft alignment between the forward and feedback weights. The proof is limited to a single training example, and the activation function for the output layer is limited to either sigmoid or softmax with a binary or categorical cross-entropy loss.

In their experiments, Frenkel et al. [22] compare DRTP to back-propagation, FA, and DFA. They train both fully-connected and convolutional networks on the MNIST and CIFAR-10 datasets and observe the following results: While DRTP cannot reach the same accuracy as the other training algorithms, it still significantly outperforms shallow learning, demonstrating the ability to train hidden layers. A likely reason for the decreased accuracy compared to DFA is that with the restriction to the error sign (through the targets), the class-dependent magnitude of the error is lost. To mitigate this effect, they suggest either tracking the magnitude of the error over the last samples to modulate the learning rate or

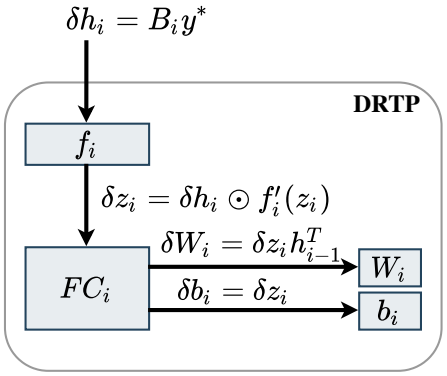


Figure 3.7.: The backward pass within a single fully-connected layer i trained with DRTP. In contrast to back-propagation, illustrated in Figure 2.6, the approximate gradient $\delta^{\text{DRTP}} h_i = B_i y^*$ is constant for each class. Changes to δz_i thus depend only on $f_i'(z_i)$.

using a learning rate scheduler. However, tracking the error of previous samples would breach the feed-forward-only nature. While the update step would not have to wait for the forward pass of the current sample, it would still have to wait for the forward pass of the last samples. Especially in very deep networks, this could lead to a high delay compared to an actual feed-forward update.

As already observed by Nøkland [56], all feedback-alignment-based algorithms have difficulties training convolutional layers. Frenkel et al. [22] investigate this further by comparing random kernels—untrained convolutional layers frozen with their random initialization—to trained kernels. They observe that training the convolutional layer generally does not improve the results for feedback-alignment-based algorithms and might even negatively impact them. In contrast, training all layers, including the convolutional layers, brings a significant advantage for back-propagation as expected. As a potential reason, Frenkel et al. [22] identify the bottleneck effect, stating that convolutional layers lack the parameter redundancy required for feedback-alignment-based algorithms. Another interesting observation is that in contrast to the other tested algorithms, DRTP does not profit from dropout and might even be impaired by it.

3.3.4. Comparison

This section summarizes the differences between back-propagation and the three more biologically inspired algorithms introduced in the previous sections, feedback alignment, direct feedback alignment, and direct random target projection. Figure 3.9 gives an overview of how the forward and backward passes change for the different algorithms. With back-propagation, the backward pass, depicted in red, computes the gradients with the chain rule, propagating them backward from layer $i + 1$ to i based on the transposed forward weights W_{i+1}^T. FA releases this dependency on the forward weights by exchanging them with the fixed random feedback weights B_i, thus solving the weight transport problem.

As with back-propagation, the training signals δh_i are propagated backward layer by layer. DFA replaces this propagation with direct feedback paths communicating the error straight to the hidden layers. By substituting the error with the targets, DRTP exchanges these direct backward paths with direct forward paths to the hidden layers. This allows updating the hidden layers immediately after the forward step. The general forward pass, depicted in blue, used to determine the network's output remains the same for all algorithms. However, DRTP additionally allows updating the hidden layers during the forward pass.

All three feedback-alignment-based approaches only replace the gradients in between two layers with approximations. This means that for a layer i, solely the gradient δh_i coming from the downstream layer $i + 1$ is replaced, while all other gradients within the layer are computed as with back-propagation. The other gradients—such as for the weights and biases—are computed using the chain rule as with back-propagation but based on an approximated gradient δh_i. The algorithms, therefore, differ exclusively in their computation of the gradient δh_i passed between layers:

$$\delta^{\mathrm{BP}} h_i = W_{i+1}^T \delta^{\mathrm{BP}} z_{i+1} \tag{3.15}$$

$$\delta^{\mathrm{FA}} h_i = B_i \delta^{\mathrm{FA}} z_{i+1} \tag{3.16}$$

$$\delta^{\mathrm{DFA}} h_i = B_i e \tag{3.17}$$

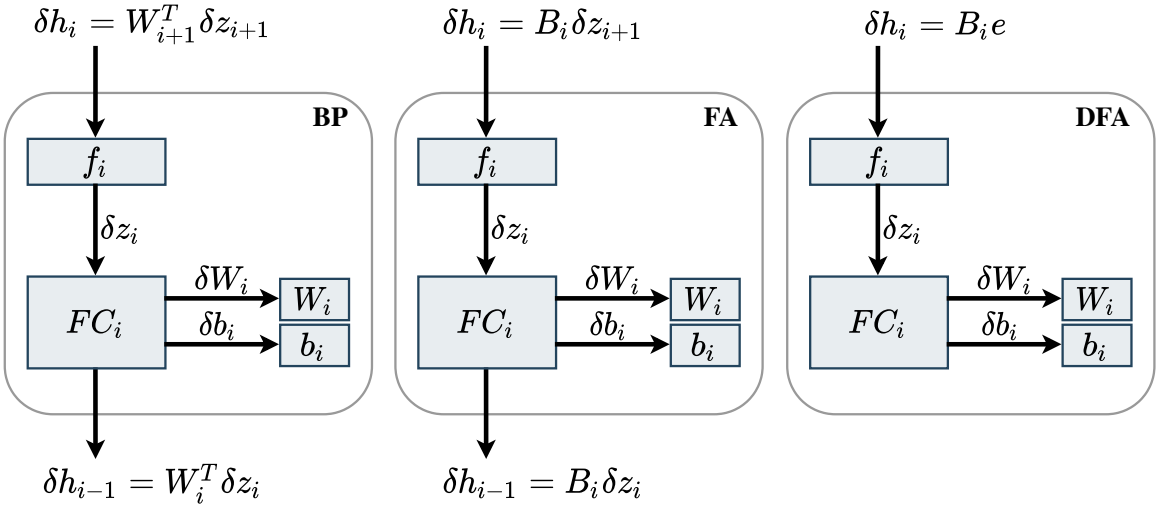
$$\delta^{\mathrm{DRTP}} h_i = B_i y^*. \tag{3.18}$$

Figure 3.8 gives a detailed look into the backward pass for a single fully-connected layer, highlighting these differences and similarities between the training algorithms.

FA solves the weight transport problem by using the feedback weights B_i instead of the forward weights W_{i+1}^T. However, with the dependency on δz_{i+1}, feedback is still propagated backward throughout the whole network, and update locking applies. DFA introduces direct feedback paths by replacing the gradient δz_{i+1} with the error e. This removes the dependency on downstream layers but does not solve update locking. Finally, DRTP releases update locking by using the targets y^* instead of the error. These gradual improvements are summarized in Table 3.1. The next chapter introduces delayed error forward projection, which is based directly on DRTP, thus extending this chain of biologically plausible training algorithms.

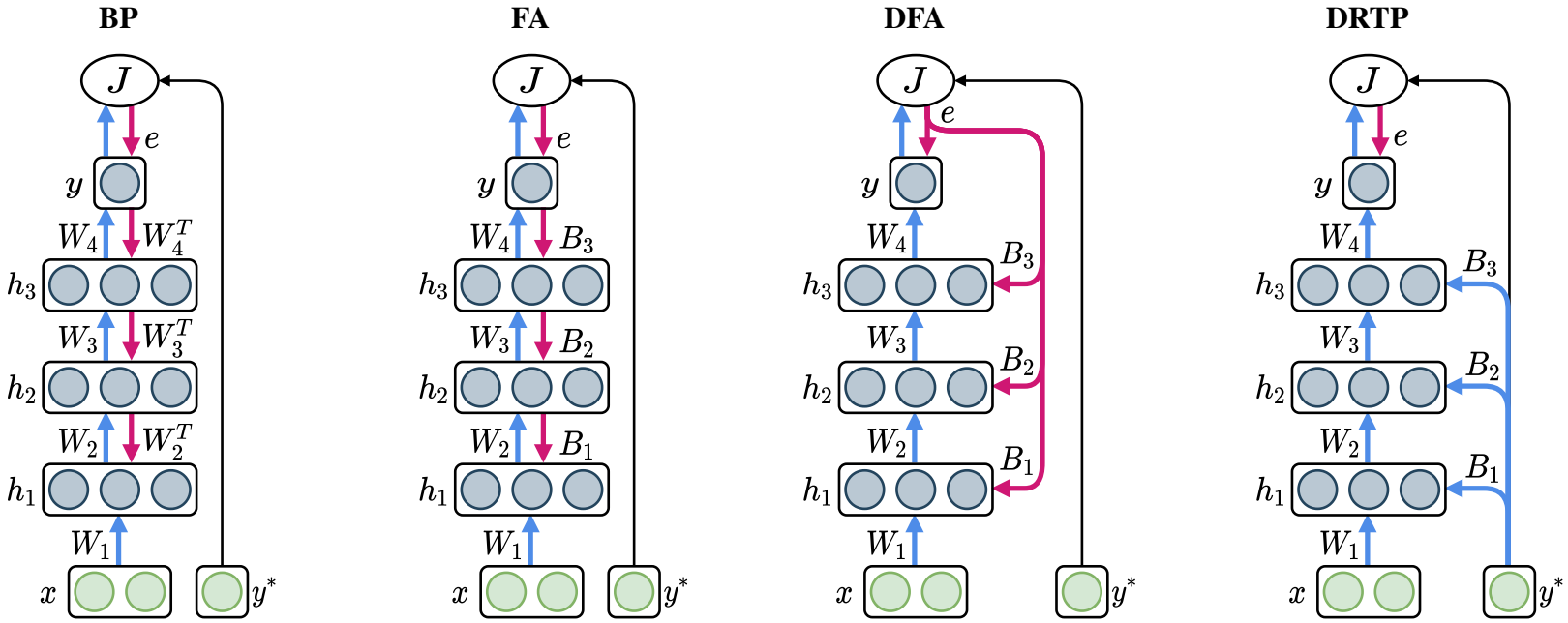
Table 3.1.: Problems solved by back-propagation, feedback alignment, direct feedback alignment, and direct random target projection.

	BP	**FA**	**DFA**	**DRTP**
No Weight Transport	×	✓	✓	✓
Direct Feedback Paths	×	×	✓	✓
No Update Locking	×	×	×	✓



(a) Back-propagation (b) Feedback alignment (c) Direct feedback alignment

Figure 3.8.: A detailed look into the backward pass within a single fully-connected layer i with weights W_i, bias b_i, and activation function f_i, trained with different algorithms. With back-propagation (a), the layer receives the gradient δh_i from the downstream layer $i + 1$, computes the gradients based on the chain rule, and passes the gradient δh_{i-1} on to the upstream layer $i - 1$. With FA (b), the computation of the gradients δh_i changes, that is, fixed random feedback weights B_i are used instead of the transposed forward weights W_{i+1}^T. As with back-propagation, the gradients δh_i are passed layer-by-layer backward through the network. With the direct feedback paths introduced by DFA (c), gradients are no longer passed backward through the hidden layers of the networks. Instead, each layer i receives the gradient δh_i directly via the feedback weights B_i. No gradients need to be passed to the upstream layer $i - 1$, as it also receives the corresponding gradient δh_{i-1} via a direct feedback path. Since DRTP is also based on direct feedback pathways, its backward pass within a layer matches that of DFA. In summary, the gradient computations within a layer—δz_i, δW_i, and δb_i—remain the same for all approaches. The algorithms differ only in how the gradient δh_i is passed to the corresponding layer.



(a) Back-propagation (b) Feedback alignment (c) Direct feedback alignment (d) Direct random target projection

Figure 3.9.: An overview of the different algorithms discussed in this section illustrated on the running example for a four-layer fully-connected network. Figure adapted from [22, Figure 1]. Back-propagation (a) is used as a baseline. The forward pass (blue) computes the output y using the forward weights W_i. The backward pass (red) computes the gradients with the chain rule based on the transposed forward weights W_i^T and updates them accordingly. Feedback alignment (b) solves the weight transport problem by replacing the transposed forward weights W_{i+1}^T in the backward path with fixed random feedback weights B_i. The forward pass remains the same. Direct feedback alignment (c) replaces the feedback paths with direct connections to the hidden layers. Finally, DRTP (d) solves the update locking problem by replacing the error e with the targets y^* for all hidden layers, thus releasing the dependency on the output of the forward pass. Each layer can be updated during the forward pass, making DRTP a feed-forward-only training algorithm.

4. Feed-Forward-Only Training of Neural Networks

This chapter introduces *delayed error forward projection (DEFP)*, my approach to increase the accuracy of direct random target projection by improving how the error information and feedback weights are modeled. First, Section 4.1 analyzes the differences between back-propagation and feed-forward-only algorithms such as DRTP. I explore which information is inaccessible to feed-forward-only approaches and which additional information could be available that is not yet used by DRTP. After that, Section 4.2 introduces DEFP, an extension of DRTP that scales the update steps with delayed error information from the previous epoch. Like DRTP, DEFP is a feed-forward-only training algorithm for neural networks, solving both the weight transport and the update locking problem, which make back-propagation biologically implausible. By using this additional delayed error information, DEFP can increase the resulting accuracy of the training by modeling the loss more accurately. Finally, Sections 4.3 and 4.4 discuss different approaches to modeling the feedback weights and activation derivatives.

4.1. Approximating Back-Propagation with Feed-Forward-Only Training

To compare the information used by back-propagation and DRTP, I examine how the gradient for the output h_i of layer i is computed. Back-propagation computes the gradient as prescribed by the chain rule via

$$\delta h_i = W_{i+1}^T \delta z_{i+1} = W_{i+1}^T \left(\delta h_{i+1} \odot f'_{i+1}(z_{i+1})\right) = \frac{\partial J(\theta)}{\partial y} W_{i+1}^T \left(\frac{\partial y}{\partial h_{i+1}} \odot f'_{i+1}(z_{i+1})\right). \tag{4.1}$$

This requires the gradient of the loss $\partial J(\theta)/\partial y$ and for all downstream layers $j \in \{i+1, \ldots, k\}$ both the transpose of the forward weights W_j^T and the derivative of the activation f_j at the input z_j. This information is not available to a feed-forward-only algorithm. To solve the weight transport problem, the algorithm cannot rely on symmetric feedback weights and thus cannot utilize the forward weights W_j or their transpose. To overcome update locking, the update step furthermore cannot wait until the downstream layers have computed their output. Thus, neither the intermediate values z_j nor the final output y nor the loss $J(\theta)$ are available. An entire forward pass would be necessary to obtain the loss. Even for the intermediate values z_j, the forward pass needs to reach the downstream layer j, making the derivative $f'_j(z_j)$ inaccessible.

Direct random target projection approximates the gradient with

$$\delta^{\mathrm{DRTP}} h_i = B_i y^*. \tag{4.2}$$

As discussed in Section 3.3.3, the targets y^* correspond under certain conditions to rescaling and shifting the error sign. Compared to the actual error, this lacks information on the error magnitude, which is essential in scaling the update steps appropriately. The remaining information, that is, the forward weights and the derivatives of the activation functions for all downstream layers, are encoded by the feedback weight matrix B_i.

Based on these observations, I identify three possible approaches to improve upon DRTP by enhancing the information used to approximate the gradients.

Error information The gradient of the loss $\partial J(\theta)/\partial y$ is inaccessible to a feed-forward-only algorithm since computing the loss $J(\theta)$ requires a complete forward pass, which would conflict with update locking. DRTP thus approximates the gradient of the loss with the targets y^*. However, finding a better approximation of the current error could improve the overall update direction and thus the resulting accuracy.

Feedback weights The feedback weights B_i are a crucial component of the training as they determine how the error information is communicated to the hidden layers and the forward weights need to come to an alignment with them. Different initialization methods thus have the potential to impact the training results considerably.

Activation derivatives The derivatives of the activation functions $f'_{i+1}(z_{i+1})$ are modeled only implicitly in the feedback weights B_i. A more explicit modeling could improve the training accuracy.

All three approaches aim to achieve models with higher accuracy while retaining the solutions to the weight transport and update locking problems. The subsequent sections discuss these approaches in more detail. First, I focus on refining the error information, especially considering the progress made during the training. Based on this idea, Section 4.2 introduces DEFP, a feed-forward-only training algorithm that improves upon DRTP by using the error information computed during the previous epoch. Section 4.3 explores different methods to model the feedback weights. Finally, Section 4.4 discusses ideas on how the statistical behavior of activation functions could be utilized for a more explicit approximation of their derivatives.

Besides improving the update direction, adjusting the step size through an improved learning rate is also a promising strategy, as already suggested by Frenkel et al. [22]. Currently, all feedback-alignment-based algorithms tested here employ a fixed learning rate. However, it has been shown that even with standard back-propagation, learning rate schedulers can provide a significant boost to performance [41]. In combination with the fixed targets as used by DRTP, decreasing the learning rate dynamically with a learning rate scheduler might yield an even greater improvement.

4.2. Scaling the Update Steps with Delayed Error Information

Based on the considerations in Section 4.1, this section discusses how to approximate the loss used to scale the update steps with additional error information. The loss measures

how close the model output is to the desired target values. In this thesis, I use the term error information to refer collectively to such measures, including both the loss and the error. As already established, a feed-forward-only algorithm has no access to exact, up-to-date error information since that would require waiting until the forward pass is finished. To compute the loss on a batch of training samples, one first needs to compute the model output by performing a complete forward pass through all layers of the model and then compare this output to the expected targets. Relying on this loss to update the parameters is thus in conflict with solving the update locking problem.

By solving the credit assignment problem, back-propagation determines how much of the loss can be attributed to the individual parameters and how they should be changed consequentially. With DRTP, Frenkel et al. [22] demonstrate that an exact solution to the credit assignment problem is not necessary and that neural networks can be trained even without the loss by using the targets instead. The objective of this thesis is to improve upon DRTP without reintroducing update locking. Note that I do not expect an improvement over DFA with this approach because the latter has access to the exact loss. In contrast to DFA, the loss can only be approximated with the delayed error information to remain unaffected by update locking.

I propose *delayed error forward projection (DEFP)*, which approximates the loss by storing the sample-wise error information e_{t-1} from the previous epoch $t-1$ and applying it in place of the actual current loss. The feedback is thus always delayed by one epoch. Algorithm 2 describes DEFP more formally, while Figure 4.1 illustrates it on the previously introduced running example. In contrast to back-propagation, DEFP requires no backward pass, and computations are performed during the forward pass in Lines 6 to 16. The incoming gradient δh_i for a layer L_i is approximated as

$$\delta^{\text{DEFP}} h_i = B_i e_{t-1} \tag{4.3}$$

and thus independent of other layers. As for the other feedback-alignment-based algorithms, the remaining gradient computations within the layer remain the same. This is represented by the *backward* call in Line 14. The last layer L_k receives the actual gradient of the loss $\delta h_k = \partial loss / \partial h_k$. After processing the last layer, the error information e_t is updated for the next epoch $t+1$. The advantage of this approach is that the delayed loss is a much more accurate representation of the actual loss than the targets because it includes the magnitude of the error. This magnitude helps with differentiating samples that are already solved quite well from problematic samples with high error. Moreover, it reintroduces the gradual reduction in error as the network gets better throughout the training, which is lacking for DRTP, as already noted by Frenkel et al. [22].

To implement DEFP, I store the error information for each training sample at the end of the forward pass. As the loss is already used to train the last layer, this requires no additional computation. Storing one additional real value per sample also has no significant memory overhead compared to the neural network and the training data themselves unless the samples are very small. Especially in the field of computer vision, where a single input image consists of at least a few hundred but often many more real-valued pixels, this additional memory consumption is negligible. The error information is initialized with the targets y^*. Thus, the first epoch of DEFP, until the delayed information has been updated

Algorithm 2: Training a model M with layers L_i on a dataset D with the optimizer Opt for $t_{\max}$ epochs using DEFP to approximate the gradients.

Input: model $M = (L_1, \ldots, L_k)$, optimizer Opt, dataset D, epochs $t_{\max}$

$e_0 \leftarrow D.y^*$
for $t \leftarrow 1$ **to** $t_{\max}$ **do**
  foreach $b \in D.batches$ **do**
    $h_0 \leftarrow D.x[b]$
    $y^* \leftarrow D.y^*[b]$
    for $i \leftarrow 1$ **to** k **do**
      $h_i \leftarrow L_i.forward(h_{i-1})$
      if $i \neq k$ **then**
        $\delta h_i \leftarrow B_i e_{t-1}[b]$
      else
        $loss \leftarrow J(h_k, y^*)$
        $\delta h_k \leftarrow \partial loss / \partial h_k$
      end
      $L_i.backward(\delta h_i)$
      $Opt.update\_step(L_i)$
    end
    $e_t[b] \leftarrow error\_information(h_k, y^*)$
  end
end

with the actual error, is equivalent to DRTP. Alternatively, one could determine the initial error with an additional epoch $t = 0$, computing only the error without updating any model parameters.

With this approach, each epoch remains free of update locking as all weights in every layer can be updated immediately without waiting until the forward pass computes the model output. However, it introduces a dependency between epochs, as processing a sample requires the delayed error information for that sample computed during the previous epoch. Since samples are usually shuffled between epochs, the sample processed last by one epoch could be processed first by the next epoch. Nonetheless, this is generally not a problem when training neural networks as there typically is a clear separation between epochs, and consecutive epochs are usually not executed concurrently. Often, there is even additional auxiliary computation in between epochs, such as intermediate evaluation or check-pointing. One could even relax this requirement at the cost of a potentially higher delay of the error information by using whatever loss is available for the current sample, be it from the previous or any earlier epoch. In summary, DEFP promises an improved approximation of the loss while remaining free of update locking, as demonstrated in Chapter 5.

There are multiple different possibilities of which error information to store.

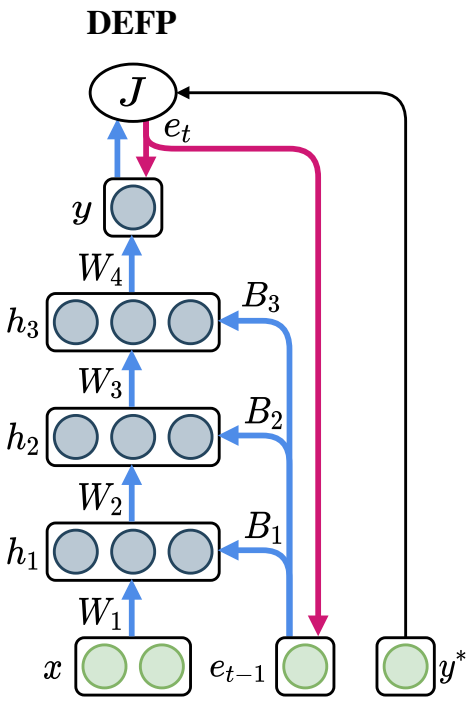


Figure 4.1.: Training the neural network introduced in Section 2.3 with DEFP. As with DRTP, the hidden layers are trained immediately during the forward pass (blue). However, instead of the targets y^*, DEFP uses the delayed error information e_{t-1} from the last epoch. At the end of the forward pass, this error information is updated for the next epoch. Using the delayed error e_{t-1} instead of the current error e_t, DEFP still solves the update locking problem.

Loss Gradient The sample-wise delayed loss gradient is the most natural information available to model the actual loss and its gradient. This corresponds to the information back-propagation would use, just delayed by one epoch.

Error Instead of the loss gradient, one can also use the error $e = y^* - y$, which essentially corresponds to the gradient of the MSE loss. Surprisingly, using the error seems to yield better results than using the actual gradient of the loss, which would more accurately conform to back-propagation. Section 6.2 discusses this observation and potential reasons in more detail.

Aggregated Error Information Instead of sample-wise information, one could also aggregate the error over all samples in the last epoch. This would reduce the already small memory overhead to just a single value for the whole training set, which could be beneficial in applications with severe hardware constraints. While losing the ability to differentiate between samples with low and high error, this still adjusts the update step size as the error decreases throughout the training, which is an advantage over DRTP.

Besides the loss gradient and the error, further scaling factors are possible, for example, the loss itself.

4.3. Modeling the Feedback Weights

As discussed in Section 4.1, the feedback weights B_i of a direct feedback pathway model the backward weights and the derivative of the activations for all downstream layers. That is, the feedback weights B_i approximate the full feedback path

$$W_{i+1}^T\Big(f'_{i+1}(z_{i+1}) \odot \Big(W_{i+2}^T\Big(f'_{i+2}(z_{i+2}) \odot \Big(W_{i+3}^T\Big(f'_{i+3}(z_{i+3}) \odot \cdots \odot \Big(W_k^T\big(f'_k(z_k)\big)\Big)\Big)\Big)\Big)\Big)\Big). \quad (4.4)$$

Both DFA and DRTP initialize all feedback weights with a He uniform distribution [29]. This section examines different approaches to model the feedback weights B_i.

First, I explore different initialization schemes for the feedback weights. There has been much research on initializing the forward weights W_i in neural networks trained with backpropagation [25, 65, 29]. A good initialization of the forward weights is essential to ensure good propagation of information in both the forward and the backward pass by avoiding vanishing and exploding values and leads to faster and better convergence, especially for very deep networks [25, 26]. The *Xavier initialization* by Glorot and Bengio [25] draws the weights from the uniform distribution

$$\mathcal{U}[-a, a] \text{ with } a = \sqrt{\frac{6}{\textit{fan-in} + \textit{fan-out}}} \quad (4.5)$$

but is based on a model with only linear activations. The *He* or *Kaiming initialization* by He et al. [29] was developed specifically for networks with rectified linear units as activation functions. It draws weights from a normal distribution with

$$\mathcal{N}\big(\mu, \sigma^2\big) \text{ with } \mu = 0 \text{ and } \sigma = \frac{\textit{gain}}{\sqrt{\textit{fan-mode}}}. \quad (4.6)$$

It is not entirely clear whether normal or uniform distributions are a better choice for initializing the forward weights, but the differences are not too significant [26]. Transferring these observations for forward weights to the feedback weights, I compare the different training algorithms initialized with both Xavier and He distributions in both the uniform and normal form and, additionally, with a normal distribution with mean $\mu = 0$ and standard deviation $\sigma = 1$.

Instead of using the same initialization method for each layer, the initialization could also be varied based on the layer's depth. This is motivated by the observation that the chain of multiplications in equation 4.4 extends with increasing distance from the output layer. Potential parameters to adjust are the standard deviation for normal distributions or the interval of a uniform distribution, both affecting the width of the distribution. However, research on initializing the forward weights actually aims to keep a relatively consistent distribution of gradients independent of the layer depth [25, 29].

Another approach is making the feedback weights variable instead of fixed. One could, for example, periodically update them with the current forward weights. This would reintroduce the weight transport problem and thus is biologically implausible. However, some of the technical advantages would still apply as the weight transport occurs significantly less often. One could also re-roll the feedback weights by drawing new random weights. The new weights could be drawn from either the same distribution, for example, when the training is stuck in a plateau, or from a different distribution adjusted based on the current training state. However, I find variable feedback weights to perform significantly worse compared to fixed weights. This is most likely related to the observations made by Nøkland [56] for DFA, where fixed weights help with the alignment between forward and feedback weights, as already discussed in Section 3.3. I, therefore, did not pursue variable weights any further.

4.4. Modeling the Activation Derivatives

The third strategy to improve the accuracy of the approximate gradients is a more explicit modeling of the activation functions and their derivatives $\delta f_i'(z_i)$. This could be based on the behavior of the chosen activation function for certain inputs. For example, with the rectified linear unit, the derivative is either one for positive inputs or zero for negative inputs. Since the derivative of the activation function is incorporated into the overall gradient via an element-wise multiplication, it either has no effect on the gradient or sets it to zero. Moreover, once the input z_i is negative and the gradient becomes zero, further modifications to any upstream weights are impossible. This is known as the dying ReLU problem [50]. To increase the similarity to back-propagation, a feed-forward-only algorithm could emulate this behavior by deactivating connections throughout the training and setting the corresponding feedback weights to zero—either temporarily or permanently. Similar considerations apply to any activation function with a zero gradient section and also relate to the vanishing gradient problem. This is only one example of incorporating the derivatives of the activation functions more explicitly into the feed-forward-only update steps. Other approaches could consider further characteristics of the activation, improve how the dying ReLU behavior is modeled, or focus on entirely different activation functions.

5. Evaluation

This chapter evaluates the performance of delayed error forward projection by training different neural networks on multiple classification datasets. With DEFP, I aim to increase the accuracy of direct random target projection while retaining its biological plausibility and purely feed-forward computation. Comparing the test error of networks trained by the different algorithms, I find that DEFP can achieve higher accuracy than DRTP, especially on fully-connected neural networks. Interestingly, DEFP with error-scaling yields significantly better results than with loss-scaling for all networks and datasets. Potential causes for this are examined further in Chapter 6. I also compare different distributions to initialize the feedback weights. However, the overall impact of the exact initialization method is marginal and depends on the network topology and dataset at hand. Finally, I examine the technical performance of the different algorithms by comparing the execution time per epoch and the total memory usage.

5.1. Methodology

This section summarizes the methodology used to obtain the results presented in the following sections. In general, I follow the methodology of Frenkel et al. [22] for better comparability to previous work.

5.1.1. Datasets

I evaluate the training algorithms on three different classification tasks, namely a synthetically generated dataset and the well-known image classification datasets MNIST [45] and CIFAR-10 [42]. While larger and more complex datasets, such as CIFAR-100 [42] or ImageNet [18], exist, I reuse the same datasets used in prior works for increased comparability. One can also argue that the biologically motivated training algorithms examined here are more suited to edge computing where MNIST and CIFAR-10 represent the required complexity more accurately [22].

5.1.1.1. Synthetic Classification

I reuse the synthetic classification dataset generated by Frenkel et al. [22, Section 4.2] using sklearn's [61] `make_classification` function. It has 25,000 training samples and 5000 test samples with 256 features, 128 of which are informative. It contains ten classes with five clusters per class and a class separation factor of 4.5.

(a) MNIST [45] (b) CIFAR-10 [42]

Figure 5.1.: Examples of images and classes for the MNIST and CIFAR-10 datasets.

5.1.1.2. MNIST

MNIST [45] is an image classification dataset of handwritten digits. It contains 60,000 training images and 10,000 test images, initially collected by the American National Institute of Standards and Technology (NIST). Each image contains a single digit from the classes zero to nine. The samples are provided as 28 px × 28 px, gray-scale images and have been normalized and centered. MNIST has long been solved with an error rate of less than one percent [45]; the most recent results achieve a test error of 0.13% [11]. Figure 5.1a gives some examples of images and classes contained in MNIST.

5.1.1.3. CIFAR-10

CIFAR-10 [42] is another image classification dataset of 32 px × 32 px color images. The 50,000 training images and 10,000 test images contain ten mutually exclusive classes of animals and vehicles. Examples for each of the ten classes are depicted in Figure 5.1b. Current approaches achieve a test error of 0.5% [19].

5.1.2. Models and Training Parameters

For the sake of comparability, I reuse the neural network topologies and training configurations used by Frenkel et al. [22]. Table 5.1 gives an overview of the configuration for each dataset. Generally, I train between 100 and 500 epochs with batch sizes between 50 and 100. Unless otherwise specified, I use the Adam optimizer [40] and binary cross-entropy loss. All models use the hyperbolic tangent as activation function for both fully-connected and convolutional hidden layers and sigmoid activation for the output layer. This keeps my results comparable to prior work and follows the observations by Nøkland [56] discussed in Section 3.3.2 that squashing activation functions limit the growth of hidden layers trained with direct feedback connections.

I apply the different training algorithms to both fully-connected and convolutional neural networks with up to five layers. These are the same networks as used by Frenkel

Table 5.1.: Training configuration and hyper-parameters used for the training unless otherwise specified. BCE refers to the binary cross-entropy loss.

	Synth	MNIST	CIFAR-10
Batch size	50	60	100
Epochs	500	100	200
Optimizer	Adam	Adam	Adam
Hidden layer activation	tanh	tanh	tanh
Output layer activation	Sigmoid	Sigmoid	Sigmoid
Loss	BCE	BCE	BCE

et al. [22]. All networks have a fully-connected output layer with ten neurons since all datasets used in these experiments have ten classes. The fully-connected networks consist of only fully-connected layers with 256 to 1000 neurons in the hidden layers. To differentiate between them, they are named based on the number of hidden layers and the maximum number of neurons per layer. Two different convolutional networks are used, one for each MNIST and CIFAR-10. The convolutional layers have between 32 and 256 activation maps, kernel sizes between $k = 3$ and $k = 5$, and a stride of one. The edges are padded with zeros to keep the image size consistent: for a kernel size k, the padding is set to $\lfloor k/2 \rfloor$. The network topologies are summarized in Table 5.2.

Table 5.2.: Neural network topologies: FC 10 is a fully-connected layer with ten neurons. CONV 32, 5×5 is a convolutional layer with a kernel size of $k = 5$ and 32 activation maps. All convolutional layers use a stride of one and zero padding to retain the image size.

Topology	Layer 1	Layer 2	Layer 3	Layer 4	Layer 5
FC1-500	FC 500	FC 10	–	–	–
FC1-1000	FC 1000	FC 10	–	–	–
FC2-500	FC 500	FC 500	FC 10	–	–
FC2-1000	FC 1000	FC 1000	FC 10	–	–
FC3-500	FC 256	FC 500	FC 500	FC 10	–
CONV-MNIST	CONV 32, 5×5	FC 1000	FC 10	–	–
CONV-CIFAR	CONV 64, 3×3	CONV 256, 3×3	FC 1000	FC 1000	FC 10

Depending on the dataset and network topology, a fixed learning rate is selected based on those determined by Frenkel et al. [22]. Different learning rates are chosen for back-propagation and feed-forward-only approaches such as DRTP and DEFP. Table 5.3 states the selected learning rates. All fully-connected networks of a certain depth use the same learning rate, independent of their maximum width.

Table 5.3.: Fixed learning rate depending on the dataset, the network topology, and the training algorithm as determined by [22].

Dataset	Topology	Back-Propagation	Feed-Forward-Only
Synth	**FC3**	5×10^{-4}	5×10^{-4}
MNIST	**FC1**	1.5×10^{-4}	1.5×10^{-4}
MNIST	**FC2**	5×10^{-4}	1.5×10^{-4}
MNIST	**CONV**	5×10^{-4}	1.5×10^{-4}
CIFAR-10	**FC1**	1.5×10^{-5}	1.5×10^{-4}
CIFAR-10	**FC2**	5×10^{-6}	5×10^{-5}
CIFAR-10	**CONV**	1.5×10^{-4}	5×10^{-5}

5.1.3. Execution Environment

All experiments are conducted on a single accelerated compute node of the HoreKa supercomputing system. The CPU is an Intel Xeon Platinum 8368 with two sockets, 38 cores per socket, and two threads per core. It has 64 KB L1 and 1 MB L2 cache per core and 57 MB shared L3 cache per CPU. The node has 512 GB of main memory and a 960 GB NVMe SSD. Four NVIDIA A100-40 GPUs with 40 GB memory each, driver version 470.57.02, and CUDA version 11.4 are available, of which I only ever use one. The operating system is Red Hat Enterprise Linux 8.2 with kernel version 4.18.0-193.60.2.el8_2.x86_64. All experiments are implemented in Python 3.8.0 compiled with GCC 8.3.1 20191121 (Red Hat 8.3.1-5) using the PyTorch framework [60] with version 1.11.0.dev20210929+cu111. The implementation is available open-source at `https://github.com/fluegelk/DEFP` and is based on the DRTP implementation by Frenkel et al. [22].

5.1.4. Quality and Performance Metrics

I use the accuracy and error rate of the resulting models to compare the different training methods. The top-1-accuracy is the percentage of samples where the predicted class, that is, the class with the highest output, is correct. Correspondingly, the error rate is the number of incorrectly classified samples. These metrics can be calculated with

$$accuracy = \frac{Correct\ Classifications}{Total\ Samples} \tag{5.1}$$

$$error = \frac{Incorrect\ Classifications}{Total\ Samples} = 1 - accuracy. \tag{5.2}$$

To infer how well a model generalizes to new inputs, I evaluate the models on the test set unless otherwise specified.

In addition, I analyze the technical performance during the training, such as memory usage and execution time. To measure the execution time, I use Python's `time perf_counter`, a monotonic, non-adjustable clock with a resolution of 10^{-9}. For memory usage, I measure the peak GPU memory allocated during the training as reported by the PyTorch memory allocator and reset the peak before each training run.

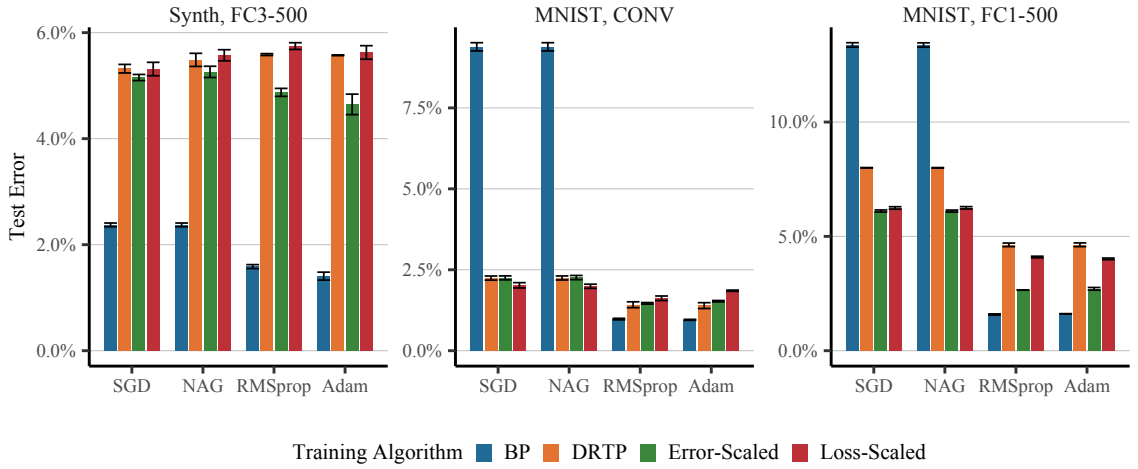


Figure 5.2.: The best test error for different optimization and training algorithms. Mean and standard error aggregated over three independent runs.

5.2. Delayed Error Forward Projection

First, I evaluate delayed error forward projection, the scaled feed-forward-only training algorithm introduced in Section 4.2. I compare DEFP to the standard back-propagation algorithm and DRTP by Frenkel et al. [22]. For DEFP, I compare two versions, either error-scaled using the error $y^* - y$ or loss-scaled using the loss gradient as delayed error information from the previous epoch. For this, I examine different optimization algorithms and their impact on the training in Section 5.2.1, followed by a detailed comparison of the training algorithms using the different error information in Section 5.2.2.

5.2.1. Optimization Algorithms

I compare four different optimization algorithms: the standard stochastic gradient descend (SGD), Nesterov's Accelerated Gradient (NAG) [55, 69], RMSprop [30], and Adam [40]. Section 2.2.2 gives an overview of these optimization algorithms with a focus on how they differ and their typical impact on training.

Figure 5.2 depicts the best test error achieved on a representative subset of the tested network topologies and datasets. The results are aggregated over three runs with different random seeds, using the same seeds for all algorithms; the error bars indicate the standard error from the mean. For back-propagation, momentum seems to have no significant impact on the test error. There is no substantial difference between either SGD without momentum and NAG or RMSprop and Adam. However, adaptive learning rates have an essential effect on the results. This becomes especially clear on the image classification task MNIST, where RMSprop and Adam outperform SGD and NAG by about a factor of ten on the achieved test error.

Compared to back-propagation, the feed-forward-only approaches are generally less affected by the selected optimization algorithm. For example, on MNIST with the convolutional network, back-propagation improves from an error of almost 10% to about

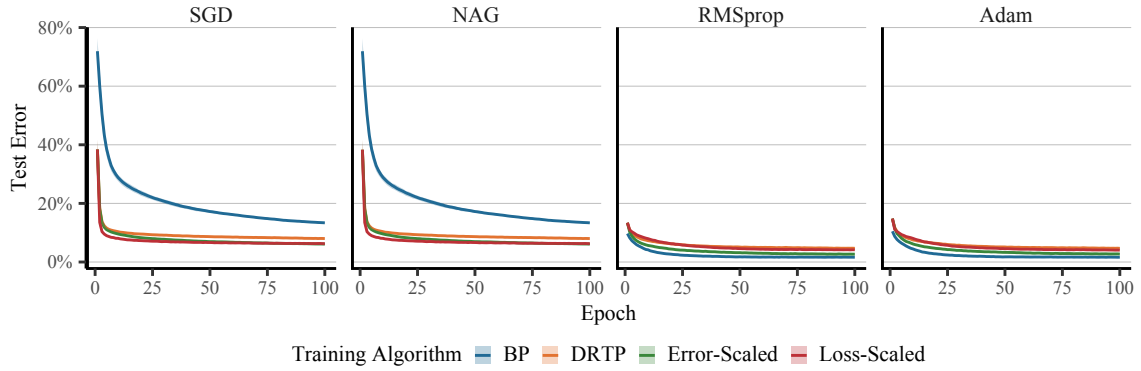


Figure 5.3.: Test error over the course of the training for different optimizers and training algorithms when training a neural network with topology FC1-500 on the MNIST dataset. Mean aggregated over three independent runs.

1% when introducing adaptive learning rates. In comparison, the feed-forward-only approaches remain between 1.5% and 2.5% error independent of the chosen optimizer. As for back-propagation, momentum has no significant impact, while there are noticeable differences with adaptive learning rates. However, adaptive learning rates do not always come with an improvement. Both DRTP and DEFP with loss-scaling lose accuracy compared to optimization without adaptive learning rates on the synthetic dataset. Of all feed-forward-only approaches, DEFP with error-scaling behaves most similar to back-propagation: neither momentum nor adaptive learning rates impair the results, and there is a significant improvement from using adaptive learning rates.

Figure 5.3 illustrates the development of the test error over the course of the training for a neural network with the FC1-500 topology on the MNIST dataset. Comparing the results achieved with adaptive learning rates, using RMSprop and Adam, to those without, adaptive learning rates appear to lead to a larger decrease in the test error and thus to faster convergence. With adaptive learning rates, back-propagation converges at a similar pace as the feed-forward-only approaches. After epoch 50 the network converges, no longer showing any significant changes. In contrast, without adaptive learning rates, back-propagation has not converged even after 100 epochs. The difference is not as notable for the feed-forward-only approaches, but adaptive learning rates still lead to overall better accuracy.

In their experiments, Frenkel et al. [22] use Adam when training on MNIST and CIFAR but NAG for the synthetic classification. However, based on these results, I use Adam for all datasets as it is clearly the best for back-propagation, and there is no valid reason to make an exception for the synthetic classification. Unless otherwise specified, all following results thus use the Adam optimizer.

5.2.2. Error Information

Figure 5.4 compares the different training algorithms on a more extensive set of networks and depicts the mean and standard error of the best test error over three independent runs.

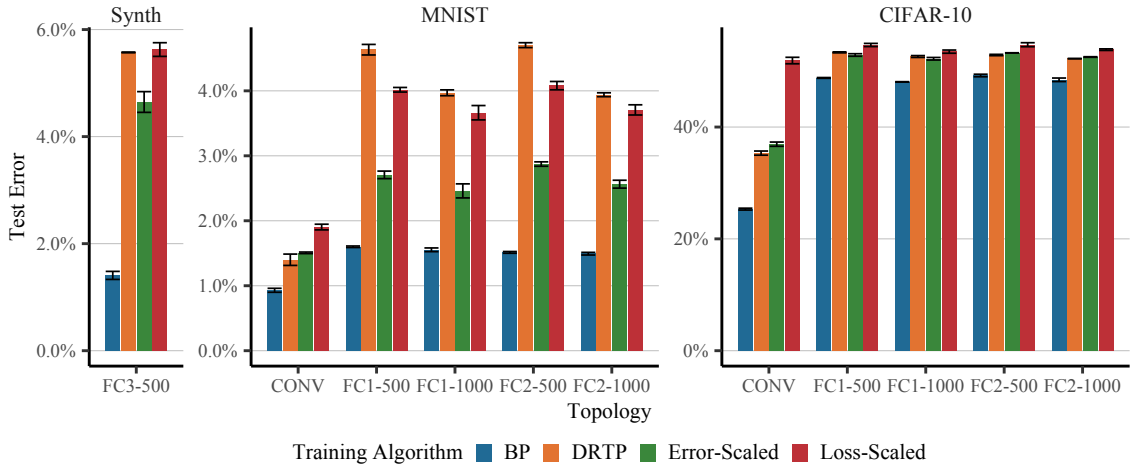


Figure 5.4.: Comparison of the best test error for different training algorithms. Aggregated over three independent runs.

I make multiple interesting observations. First, back-propagation remains better than all feed-forward-only solutions on all tested networks. It appears that ensuring biological plausibility comes at the cost of reduced accuracy.

Section 4.2 introduced two new approaches to handle the error information based on information from the previous epoch, namely DEFP with loss- and with error-scaling. While the update steps are scaled based on the loss gradient in loss-scaled DEFP, the error-scaled version uses the error instead. In both cases, the delayed value from the previous epoch is used. Even though loss-scaling is more similar to back-propagation than error-scaling from a theoretical point of view, the latter yields significantly better results over all networks and datasets. The only exception where loss-scaling improves over DRTP is for fully-connected networks on the MNIST dataset. In contrast, error-scaling offers a notable improvement over DRTP for both MNIST and the synthetic classification on all fully-connected networks. Only on the convolutional networks, DRTP appears to be better than error-scaling.

For CIFAR-10, none of the algorithms or networks can solve the task to a satisfying level. For the fully-connected networks, the test error remains at approximately 50%. While better than plain guessing—with ten classes and balanced class frequencies, one would expect a top-1-error of 90% when randomly selecting a class—correctly classifying only every second image on average is still unacceptable. Although achieving slightly better results, even back-propagation cannot reasonably train these networks, implying that the feed-forward-only training approaches are not the problem here. As expected, the convolutional network yields better results. However, even with back-propagation, a test error of 25% is far from the current state-of-the-art, which is in the order of 0.5% as achieved by Dosovitskiy et al. [19].

Figure 5.5 illustrates the test error throughout the training for a subset of selected networks representing the different behaviors observed. The fully-connected networks not shown here generally exhibit a behavior highly similar to the FC1-500 topology. A comparison of all networks can be found in supplementary Figure A.1. The data

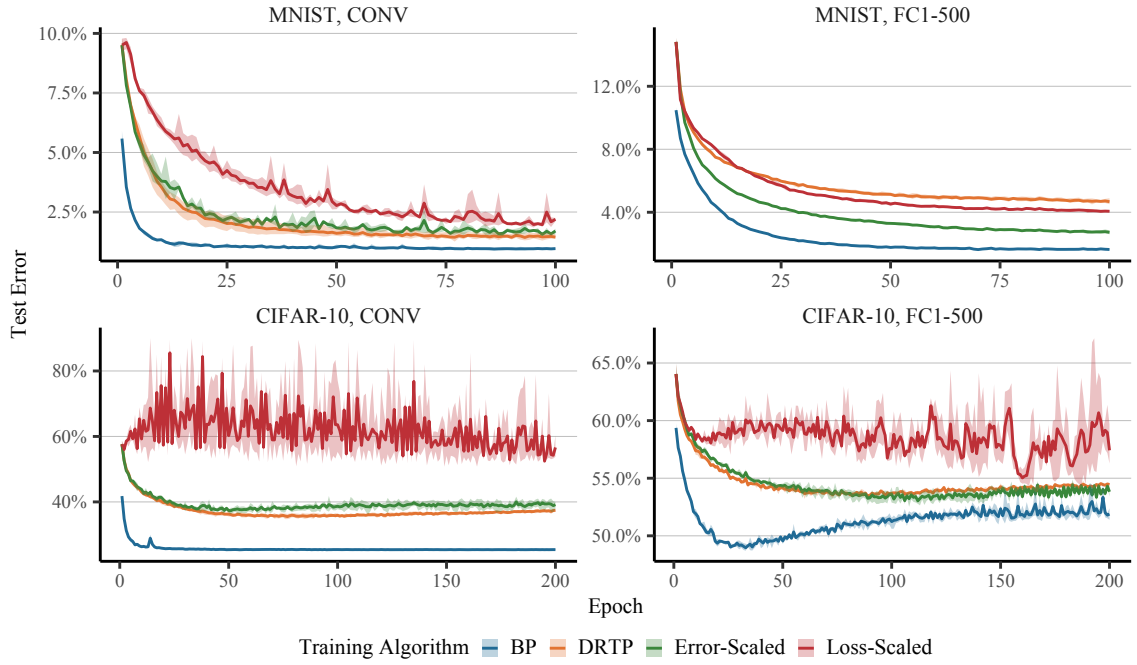


Figure 5.5.: Test error over the course of the training for different training algorithms. Aggregated over three independent runs.

is aggregated over three independent runs. Displayed here are the mean and the 95% confidence interval. With numerous spikes throughout the curve, the test error for loss-scaled DEFP is significantly more volatile than for the other algorithms. This behavior is particularly pronounced for CIFAR-10 and the convolutional networks. For error-scaled DEFP, the test error is also less smooth than for DRTP and back-propagation but to a much lesser extent than for loss-scaling. For CIFAR-10, I observe back-propagation overfitting on the fully-connected networks while the feed-forward-only approaches never improve over back-propagation, even after it overfits.

5.3. Feedback Weight Initialization

This section explores the impact of the feedback weight initialization on the models' final results. As discussed in Section 4.3, I initialize the feedback weights used to approximate the gradient once before starting the training. Weights for different layers are initialized independently, meaning I do not reuse feedback weights as proposed by Nøkland [56].

I test two types of distributions, namely normal and uniform distributions. All normal distributions are centered around 0, that is $\mathcal{N}(\mu, \sigma^2)$ with $\mu = 0$, but differ in their standard deviation σ. The normal distribution without any further designation refers to the distribution with a standard deviation of $\sigma = 1$. The Xavier normal distribution is based on the Xavier uniform distribution [25] introduced in Section 4.3. It has a standard deviation of

$$\sigma = \sqrt{\frac{2}{\textit{fan-in} + \textit{fan-out}}}. \tag{5.3}$$

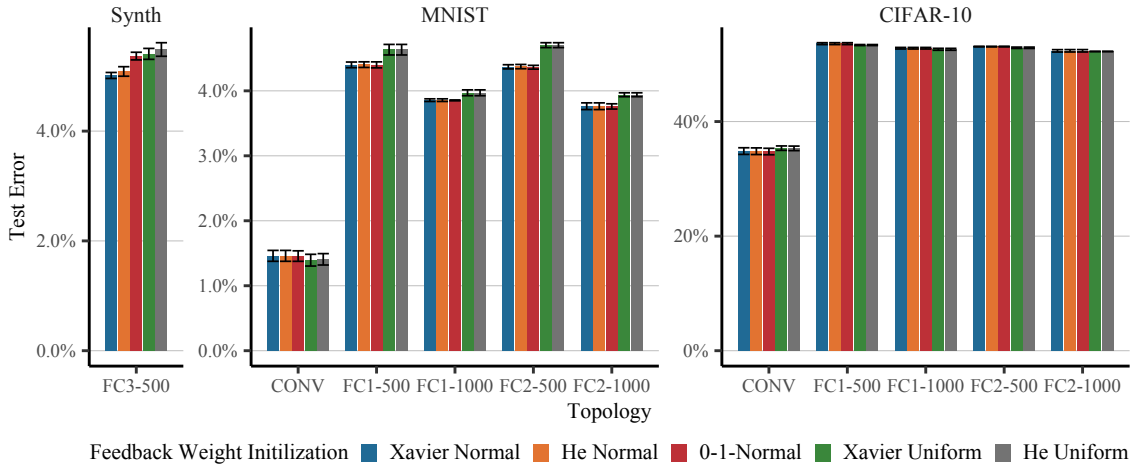


Figure 5.6.: Comparison of the best test error for initializations of the feedback weights. All weights are initialized independently with the specified distribution. Aggregated over three independent runs.

The He normal distribution [29] has a standard deviation of

$$\sigma = \frac{gain}{\sqrt{fan\text{-}mode}}. \tag{5.4}$$

In my experiments, I use a gain of $\sqrt{2}$ and the fan-in mode. The uniform distributions are also centered around 0 but vary in their interval length. The Xavier uniform distribution [25] uses an interval of

$$\pm \sqrt{\frac{6}{fan\text{-}in + fan\text{-}out}}, \tag{5.5}$$

while the He uniform distribution [29] uses the interval

$$\pm \frac{gain \cdot \sqrt{3}}{\sqrt{fan\text{-}mode}}. \tag{5.6}$$

Figure 5.6 compares the test error for different distributions to initialize the feedback weights. Generally, the errors differ only marginally, with a maximum difference of 1% between any two initialization schemes. For fully-connected networks, normal distributions seem to perform slightly better than uniform distributions. This choice is less clear for convolutional networks, which occasionally benefit from using uniform distributions, for example, on the MNIST dataset.

5.4. Training Performance

This section explores the technical performance of different feed-forward-only training methods compared to back-propagation in terms of per-epoch execution time and required GPU memory.

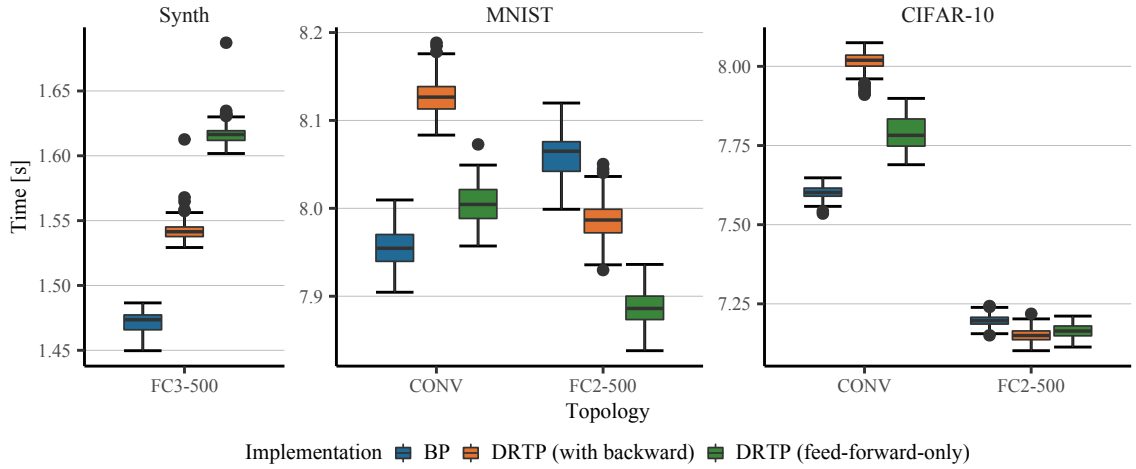


Figure 5.7.: Execution time per epoch for fully-connected and convolutional neural networks.

5.4.1. Execution Time

To evaluate the training algorithms regarding their technical performance, I first examine the impact of the training algorithm on the average execution time per epoch for different datasets and network topologies. I compare two implementations of the feed-forward-only approach. Frenkel's original implementation of DRTP [22] replaces the gradients during the backward pass with approximate gradients based on the feedback weights. As separate forward and backward passes are still required, it is actually not completely free of update locking. My novel implementation, in contrast, is purely feed-forward, allowing each layer to be trained directly after finishing its forward pass. Note that neither of the implementations has been optimized specifically for technical performance. Figure 5.7 shows the execution time for the topologies introduced in Section 5.1.2. With the differences between the algorithms and implementations being relatively small, there is no clear best pick.

To investigate these differences more thoroughly, I compare the execution times on three additional fully-connected networks. They are chosen as rather extreme topologies, i.e., very small, very wide, or very deep, to examine how this impacts the training time. The first network consists of a single output layer with ten neurons. The wide network contains ten hidden layers with 10,000 neurons each, while the deep network uses 1000 hidden layers, each consisting of 1000 neurons. Thus, the number of parameters in the hidden layers of the wide and deep networks is approximately equal. Figure 5.8 illustrates the execution time on these additional networks, which paints a much clearer picture of how the training approach and network topology relate to the execution time. With average epoch times of slightly more than a second, there are only marginal differences between the implementations on the single-layer network. The differences between the approaches are slightly more pronounced on the ten-layer network. An epoch takes approximately 25 s, with the different methods being less than two seconds apart. Interestingly, back-propagation and gradient replacement exhibit almost equal execution times of approximately 25.0 s,

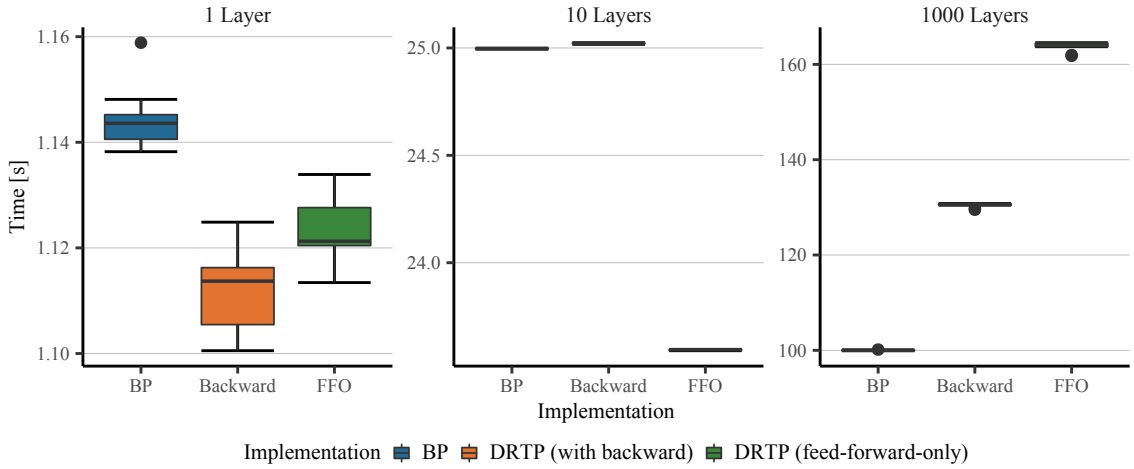


Figure 5.8.: Execution time per epoch for fully-connected neural networks with extreme topologies.

while the feed-forward implementation is slightly faster at 23.6 s. On the deep network with one thousand layers, I observe a distinct ordering of the implementations. With the average execution time ranging between 100 s and 165 s per epoch, there appear to be significant differences between the training methods. Traditional back-propagation is the fastest option, followed by replacing the gradients in the backward pass, which takes approximately 30 s longer per epoch. The purely feed-forward implementation has an even higher overhead of approximately 65 s compared to back-propagation.

5.4.2. Memory Consumption

Table 5.4 gives the peak consumption of GPU memory for the different training methods on both fully-connected and convolutional neural networks. In general, there is no significant difference in memory consumption between back-propagation and the feed-forward-only algorithms. While I observe a slight increase in memory for DRTP compared to back-propagation, this amounts to less than one percent of the total memory consumption. The memory used to store the additional feedback weights B_i seems to be mostly compensated by the remaining gradient computations. For DEFP, there is an additional overhead of 1 to

Table 5.4.: Peak GPU memory consumption of the different training algorithms for fully-connected and convolutional networks.

Dataset	Topology	BP	DRTP	DEFP
Synth	FC3-500	11.94 MB	11.99 MB	12.95 MB
MNIST	FC2-500	18.70 MB	18.74 MB	21.03 MB
CIFAR-10	FC2-500	45.40 MB	45.44 MB	46.83 MB
MNIST	CONV	291.27 MB	292.27 MB	294.56 MB
CIFAR-10	CONV	842.04 MB	847.12 MB	849.03 MB

2.3 MB compared to DRTP to store the delayed error information. Thus, using feed-forward-only algorithms appears to be neither advantageous nor significantly disadvantageous in terms of memory usage compared to back-propagation, at least when using high-level frameworks such as PyTorch without explicitly optimizing for memory consumption.

6. Discussion

With delayed error forward projection, I introduced a feed-forward-only training algorithm for neural networks. Like DRTP, DEFP solves both the weight transport and the update locking problem. Using the delayed error from the previous epoch as an additional sample-wise scaling factor, DEFP can improve upon the accuracy of DRTP by up to 2.02%, a relative improvement of more than 40%. This chapter discusses the results presented in Chapter 5 and sketches possible next steps for future work.

6.1. Comparison to Existing Approaches

Delayed error forward projection is a purely feed-forward, biologically plausible training algorithm for feed-forward neural networks. It comes with only a slight decrease in accuracy compared to the biologically implausible back-propagation. As discussed in Section 3.1, back-propagation requires symmetry between forward and backward weights—the so-called weight transport problem—and updates have to wait until both the forward and the backward pass have fully processed all downstream layers, known as the update locking problem. With DEFP, I aim to solve these issues.

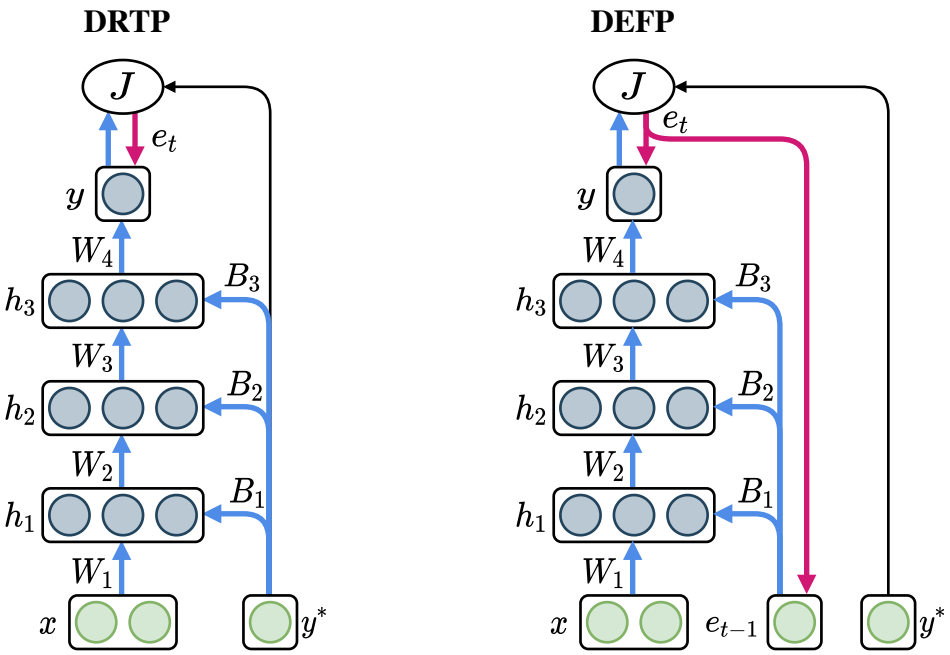
Compared to the previous feedback-alignment-based training algorithms introduced in Section 3.3, DEFP retains the biological plausibility of DRTP while improving the achieved accuracy. Figure 6.1 gives a conceptual comparison of DEFP and DRTP. Instead of the targets y^*, DEFP propagates the delayed error e_{t-1} to the hidden layers. In each epoch t, the current error e_t is computed by comparing the network's output y to the targets y^*. In DRTP, this error is used only to train the output layer, while all hidden layers are trained based on the targets alone. In contrast, DEFP records this error for each sample and stores it as delayed error to be used in the next epoch. The approximate gradient δh_i passed to each hidden layer i is thus defined as

$$\delta^{\text{DRTP}} h_i = B_i y^* \tag{6.1}$$

$$\delta^{\text{DEFP}} h_i = B_i e_{t-1}. \tag{6.2}$$

With this approach, DEFP can use the actual error of each sample—albeit delayed by one epoch—while remaining free of update locking. While this comes with a slight memory overhead from storing the delayed error, it is generally negligible compared to the overall memory usage as observed in Section 5.4.2. With this additional delayed error information, DEFP can notably increase its accuracy over DRTP.

In the context of brain-inspired neuromorphic devices, training neural networks with back-propagation has to happen out-of-the-loop due to back-propagation's biological implausibility. After the network has been trained on separate hardware, it needs to be



(a) Direct random target projection

(b) Delayed error forward projection

Figure 6.1.: A comparison of DRTP (a) and DEFP (b). DEFP stores the error information e_{t-1} from the previous epoch $t-1$, which can then be used in the current epoch t instead of the targets y^* at virtually no cost.

converted to and deployed on the neuromorphic device to handle the inference. Feed-forward-only training algorithms like DEFP and DRTP close this training loop. By solving the weight transport and the update locking problem, they enable training the neural network directly on the neuromorphic device without the need for external hardware.

6.2. Comparison of Loss- and Error-Scaling

I introduced two versions of delayed error forward projection, using either the gradient of the loss or the error itself, that is, the difference between the expected and the actual output, as additional error information. Both variants use the delayed error information from the previous epoch, allowing DEFP to avoid update locking. As illustrated in Section 5.2, this can provide a significant boost to the model's accuracy. However, contrary to my expectations, scaling the updates with the error consistently outperforms loss-based scaling.

When training neural networks, loss functions are used to penalize incorrect predictions so the model parameters can be adjusted to improve the overall accuracy. How severe a specific misprediction is, and thus which loss function should be selected, depends on the task at hand. When solving regression problems, the output values are continuous, whereas a classification model predicts discrete classes. This distinction impacts how incorrect outputs are perceived. For example, when predicting a continuous value from 1

to 100, an output of 5 instead of 6 is almost correct. In contrast, selecting the fifth instead of the sixth class is entirely wrong.

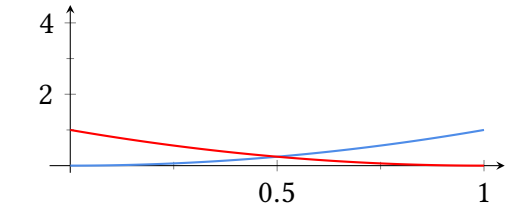
By choosing the correct loss function, a network can be penalized accordingly. For regression, the mean squared error (MSE) loss

$$MSE(y, y^*) = \frac{1}{N}\sum_{i=1}^{N}\left(y_i^* - y_i\right)^2 \tag{6.3}$$

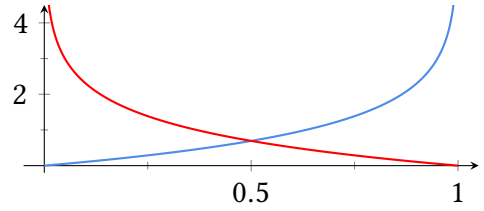
is a common choice. It increases quadratically with the difference between the expected value y^* and the actual output y. For classification tasks, we often use a cross-entropy loss based on information theory. In the binary form, it is defined as

$$BCE(y, y^*) = -\frac{1}{N}\sum_{i=1}^{N} y_i^* \log(y_i) + \left(1 - y_i^*\right)\log(1 - y_i). \tag{6.4}$$

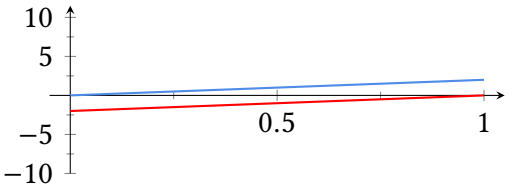
Figure 6.2 illustrates the behavior of cross-entropy and mean squared error loss for one sample of a binary classification problem. For an incorrect classification, the cross-entropy loss is significantly higher than the mean squared error and approaches infinity when predicting the opposite class. This behavior is also represented in the gradients: the gradient of mean squared error scales linearly with the error, whereas the gradient of cross-entropy increases significantly the closer the error gets to one, again approaching $\pm\infty$.



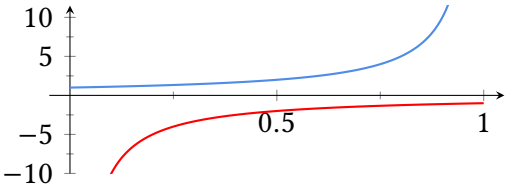
(a) MSE Loss



(b) BCE Loss



(c) MSE Gradient



(d) BCE Gradient

Figure 6.2.: Mean squared error and binary cross-entropy loss with corresponding gradients for a binary classification sample with target 0 (blue) or 1 (red).

For DEFP, error-scaling is equivalent to loss-scaling with a mean squared error loss, as already noted in Section 4.2. Since all experiments in this work use a binary cross-entropy loss, the difference between error- and loss-scaling corresponds to the difference between the gradients depicted in Figures 6.2c and 6.2d. At this point, I can only presume why loss-scaling performs worse than error-scaling. A probable cause is that with loss-scaling, the

effective step size varies too much due to the non-linearity of the gradient, not accounted for by the learning rate. This could also explain why the error curve of loss-scaled training in Figure 5.5 contains so many spikes compared to the other approaches. A potential solution to this is using better learning rates or even a learning rate scheduler. Currently, I use the same fixed learning rate for all feed-forward-only algorithms. However, the magnitude of the error information, which can be considered a scaling factor to the update steps, differs widely: DRTP corresponds to using the targets as error information which are either 0 or 1. Similarly, the error ranges between -1 and 1. In contrast, the gradient of the cross-entropy loss ranges from ± 1 to $\pm\infty$. It is thus likely that loss-scaling requires a substantially different learning rate compared to the other two approaches. In addition to the learning rate, a future study could investigate adjusting the effective step size via other loss functions with different gradients.

6.3. Feedback Weight Initialization

Section 5.3 compares different distributions to initialize the feedback weights. I test zero-centered normal and uniform distributions with different widths as prescribed by the Xavier and He distributions. While some distributions offer a slight improvement over others, I find the overall impact to be marginal. There is no clear indication of which initialization methods are more suitable, as their relative performance varies by dataset and network topology. For example, with the fully-connected networks trained on MNIST, normal distributions consistently outperform uniform distributions. In contrast, the convolutional network seems to work best with the Xavier uniform distributions.

This behavior is somewhat unsurprising since even for traditional neural networks trained with back-propagation, it is still unclear if the forward weights should be initialized with a normal and uniform initialization [26]. A potential approach is choosing the distribution of the feedback weights as a hyper-parameter explicitly tailored to the network and dataset, similarly to the learning rate.

6.4. Hardware Performance

As demonstrated in Section 5.4, even straightforward implementations of feed-forward-only training algorithms such as DEFP can compete with back-propagation in terms of memory usage and execution time on conventional machines. Choosing to train with a feed-forward-only approach thus brings no significant disadvantages for the technical performance. Further optimizations might be possible to increase the efficiency even in less restricted settings. The current implementation is based on PyTorch's autograd framework [60], which is intended to be used with back-propagation. While it can easily be adjusted to train in a feed-forward-only fashion, further care might be necessary for an optimized implementation. Under stricter hardware constraints, Frenkel et al. [23] demonstrate a highly efficient, low-cost implementation on an event-driven processor enabled by DRTP.

6.5. Layer-Wise Parallel Training with Pipelining

Feed-forward-only training algorithms offer intriguing possibilities for layer-wise parallel training of neural networks. Currently, there are two main strategies to parallelize the training of a neural network: data parallelism and model parallelism. An extensive overview of parallelization in neural networks, including data, model, and pipeline parallelization, is given by Ben-Nun and Hoefler [6].

In data parallelism [6, 14], the training is parallelized over the samples in a batch. A batch is partitioned and distributed to the different processing units, which process these parts in parallel. Each processing unit has its own copy of the neural network and computes a complete forward and backward pass on its assigned samples, communicating with the other units when necessary to keep the copies consistent. To achieve an efficient parallelization, data parallelism requires a sufficiently large batch size to counterbalance the communication and synchronization overhead. However, choosing an overly large batch size can negatively impact the quality of the trained networks due to the regularization effect of smaller batches [77, 26, 6].

The second fundamental parallelization technique is model parallelism [21, 6]. In model parallelism, the network itself is partitioned and distributed across different processing units. In contrast to data parallelism, each processing unit holds only a part of the model but processes all data samples. There is communication between units holding adjacent model parts, similar to how the corresponding neurons interact in the forward and backward passes. Model parallelism comes with the additional advantage of enabling the training of larger models that would not fit on a single processing unit. However, its parallelization capabilities are limited by the dependencies between the layers since two consecutive layers depend on their reciprocal output in both the forward and the backward pass. Thus, model parallelism mainly profits from partitioning the model along the data path by splitting each layer across the processing units instead of separating the network layer-wise. A further challenge to model parallelism is reversing the communication for the backward pass in arbitrary neural networks [6].

A third parallelization strategy is pipelining [6], which can be interpreted as both a form of data parallelism and model parallelism. Overlapping the computations for multiple samples, pipelining employs data parallelism while partitioning the layers of a network across the pipeline stages corresponds to model parallelism [6]. Feed-forward-only training algorithms relax the inter-dependency of layers by solving the update locking problem and replacing the backward pass with individual update steps. While a layer still depends on the output of the previous layer during the forward pass, the update step is independent of other layers and can thus be performed immediately after the forward step. This enables an improved layer-wise parallelization of the training based on pipelining. To illustrate how this layer-wise pipelining works, recall the running example of a four-layer neural network and consider its training on four batches of training samples. For simplification, this illustration assumes that the forward and backward steps in all layers require the same amount of time, which is abstractly referred to as a time step. While this is an abstraction from reality, the computational costs of forward and backward steps are mostly the same [26]. However, in most networks, the compute intensity varies by layer, which can result in load balancing issues.

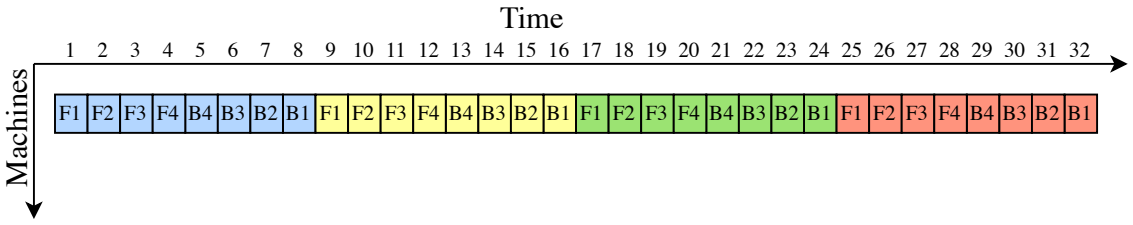


Figure 6.3.: Training a four-layer neural network on four different batches (colored blue, yellow, green, and red) with back-propagation. The forward pass processes the layers in order (F1–F4), followed by the backward pass in reverse order (B4–B1). Because of the inter-dependency of adjacent layers in both the forward and the backward pass, a batch cannot be parallelized. Furthermore, the next batch may only enter a layer once it has been updated by the previous batch. Thus, processing the four batches requires 32 time slots.

Figure 6.3 outlines training the four-layer network on the four batches with back-propagation. The forward pass for each batch consists of four forward steps, F1 to F4, for the four layers. Conversely, the backward pass processes the layers in reverse order B4 to B1. As each layer requires the previous layer's output in the forward pass, the forward steps successively depend on each other, prescribing a fixed step order. The backward steps similarly depend on the downstream layer's gradient, restricting their order as well. With the gradients depending on the loss, the forward pass needs to be finished before any backward step can start. Additionally, there is a dependency between the batches as a batch may only enter a layer after the previous batch has updated it. These restrictions force back-propagation to perform the required steps sequentially: first finish the forward pass, followed by the backward pass, and only then start the next batch. With this approach, processing the four batches requires 32 time slots.

In contrast, feed-forward-only algorithms solving the update locking problem allow training the same network in only 11 time slots using pipelining. This pipelined parallelization is illustrated in Figure 6.4. While the inter-layer dependency in the forward pass remains, a layer can now be updated immediately after its forward step. As it is no longer required to wait for the forward pass to complete or the backward pass to return to this layer, it can be updated in parallel to the remaining forward pass. Furthermore, layers no longer need to be updated in reverse order. This allows us to start processing the next batch long before the current batch has been finished. After only two steps, the first layer has processed the current batch, updated its weights accordingly, and can continue with the next batch. In contrast, the second batch can only be started after eight time steps when training with back-propagation. This delay is even higher for deeper networks.

In total, feed-forward-only training allows overlapping multiple batches as well as the forward pass and update steps of a single batch. In the example, this reduces the total time to process the four batches from 32 to only 11 time units using four processing units, corresponding to a relative speedup of

$$S_p = \frac{T_1}{T_p} = \frac{32}{11} \approx 2.9 \tag{6.5}$$

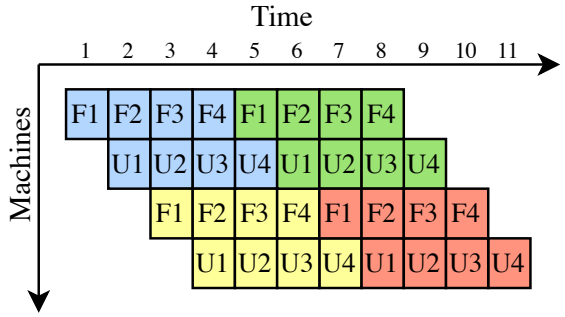


Figure 6.4.: One possibility of training the same four-layer neural network on the four batches (colored blue, yellow, green, and red) with a feed-forward-only algorithm free of update locking. In the forward pass (F1–F4), layers still depend on the output of the previous layer. However, without update locking, the update steps (U1–U4) can occur anytime after the corresponding forward step in that layer has been completed, independent of the progress of either forward or update steps in other layers. This allows updating the layers in parallel to the remaining forward pass. Furthermore, since the updates no longer have to be performed in reverse order, the next batch (yellow) can be started only two time steps after starting the first batch (blue). This adds an additional layer of parallelization, allowing an overlap of both forward and update passes and separate batches. In total, pipelining reduces the required time slots to 11.

and an efficiency of

$$E_p = \frac{S_p}{p} = \frac{2.9}{4} \approx 0.73. \tag{6.6}$$

In most real-world examples, an epoch contains significantly more than four batches, increasing the scalability even further. Generally speaking, a pipeline with k stages processing n elements requires

$$k + n - 1 \tag{6.7}$$

time steps to process all elements while a sequential approach takes

$$k \cdot n \tag{6.8}$$

time steps. With the presented strategy to layer-wise pipelined training, the number of stages k corresponds to the number of layers, and the number of elements n corresponds to twice the number of batches b since a batch requires two steps per layer—forward and update. Starting a new batch every second time slot, this approach can utilize up to k processing units in parallel, yielding a theoretical speedup and efficiency of

$$S_k = \frac{T_1}{T_k} = \frac{2kb}{k + 2b - 1} \tag{6.9}$$

$$E_k = \frac{S_k}{k} = \frac{2b}{k + 2b - 1}. \tag{6.10}$$

The allocation of steps to processing units and time slots described here is only one of multiple possibilities intended to outline how solving update locking enables layer-wise parallelization through pipelining. In an actual implementation, another allocation might be preferable. This layer-wise parallelization brings several advantages. In contrast to data parallelism, it does not depend on sufficiently large batches. While it can be seen as a form of model parallelism, relaxing the dependencies between layers opens a whole new dimension to parallelization. Moreover, this pipelining approach should work in conjunction with both data and model parallelism and can thus complement existing parallelization approaches for even higher efficiency.

7. Conclusion

Artificial neural networks have reached immense advances in performance over the last decade, leading to a wide range of practical applications. However, the underlying approach to training neural networks, that is, solving the credit assignment problem by computing gradients with back-propagation, has remained largely the same. Nonetheless, back-propagation has long been criticized for being biologically implausible as it relies on concepts that are not viable in the brain. In this thesis, I focus primarily on two core issues of back-propagation: the weight transport and the update locking problem. In addition to being biologically implausible, these issues lead to technical constraints impacting the memory overhead, communication, and energy consumption of back-propagation, thus limiting its viability under strict hardware and resource constraints. Even from an optimization point of view, back-propagation can become impeded by the increasingly deeper networks with stronger non-linearities, as demonstrated by the vanishing and exploding gradient problem.

With delayed error forward projection, I introduced a feed-forward-only training algorithm for multi-layer neural networks. DEFP solves the weight transport problem of back-propagation by implementing direct random feedback connections and avoids update locking by using the error information from the previous epoch. This makes it more biologically plausible than back-propagation and increases the resulting accuracy compared to the similarly plausible DRTP algorithm.

I evaluated DEFP on different neural networks, consisting of both fully-connected and convolutional layers, and three different classification datasets. Comparing the test error, I found that DEFP can achieve higher accuracy than DRTP, especially on fully-connected neural networks. Surprisingly, DEFP with error-scaling yields significantly better results than with loss-scaling for all networks and datasets. A probable cause for this is that loss-scaling requires a more careful selection of the learning rate, as the range of potential values for the loss gradient is substantially larger than for the error or targets. An interesting future step entails a more thorough investigation of this effect, including determining a better learning rate or even employing a learning rate scheduler. I also compared different feedback weight initialization schemes, where I found their overall impact to be only marginal. With a further dependency on the network topology and dataset at hand, there was no clear indication of a general best choice. Comparing the technical performance of feed-forward-only training to back-propagation, I found that the execution time per epoch differs only slightly on most tested network architectures. However, in very deep networks with one thousand layers, back-propagation clearly outperformed the feed-forward-only approach. This is most likely caused by an implementation not optimized for pure execution time. In terms of memory consumption, I found the overhead to store the delayed error information to be negligible compared to the overall memory consumption.

By improving upon DRTP, DEFP demonstrates that feed-forward-only algorithms can train neural networks effectively, lacking only slightly behind back-propagation in terms of accuracy. Since no backward pass is required, this enables on-device training with highly promising neuromorphic devices, a new generation of AI hardware that emulates the neural structure and operation of the human brain. Thanks to their high efficiency, training neural networks directly on neuromorphic devices instead of out-of-the-loop on separate hardware has the potential to not only economize compute resources and energy tremendously but also to simplify the training process significantly. By releasing update locking and thus the inter-layer dependencies when computing the gradients, feed-forward-only algorithms like DEFP furthermore offer great possibilities for parallelization, an ever-important topic with increasingly large networks and datasets. As outlined in Section 6.5, replacing the layer-by-layer backward propagation with direct forward paths allows for a layer to be updated immediately after its forward step and thus facilitates a pipelined parallelization approach.

Based on my findings, multiple interesting follow-up steps can be taken. As already mentioned, a more thorough investigation of why error-scaling consistently outperforms loss-scaling would be necessary. This includes testing whether this effect can be reduced using better learning rates or a learning rate scheduler. Furthermore, I intend to extend my experiments to more realistic use cases by considering more complicated network topologies, larger datasets, and more challenging problems. Considering the sub-par performance of all training algorithms, including back-propagation, on CIFAR-10, it would be interesting to compare the results on larger networks, where at least back-propagation achieves a satisfying result. Similarly, applying DEFP to networks with residual connections would be of interest. Given that the current implementation offers no notable increase in technical performance over back-propagation, it would be interesting whether a more optimized implementation can achieve a performance improvement even on traditional hardware without the restrictions of edge computing. Finally, having proposed the pipelined parallelization of DEFP in theory, implementing it and demonstrating its practical performance would be an obvious next step.

Bibliography

[1] Dario Amodei et al. “Deep Speech 2: End-to-End Speech Recognition in English and Mandarin”. In: *Proceedings of The 33rd International Conference on Machine Learning*. PMLR, June 2016, pp. 173–182.

[2] Frederico A.C. Azevedo et al. “Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain”. In: *Journal of Comparative Neurology* 513.5 (2009), pp. 532–541. ISSN: 1096-9861. DOI: 10.1002/cne.21974.

[3] Claudine Badue et al. “Self-driving cars: A survey”. In: *Expert Systems with Applications* 165 (Mar. 2021), p. 113816. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2020.113816.

[4] Pierre Baldi, Peter Sadowski, and Zhiqin Lu. “Learning in the machine: Random backpropagation and the deep learning channel”. In: *Artificial Intelligence* 260 (July 2018), pp. 1–35. ISSN: 0004-3702. DOI: 10.1016/j.artint.2018.03.003.

[5] Eugene Belilovsky, Michael Eickenberg, and Edouard Oyallon. “Decoupled Greedy Learning of CNNs”. In: *Proceedings of the 37th International Conference on Machine Learning*. PMLR, Nov. 2020, pp. 736–745.

[6] Tal Ben-Nun and Torsten Hoefler. “Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis”. In: *ACM Computing Surveys* 52.4 (Aug. 2019), 65:1–65:43. ISSN: 0360-0300. DOI: 10.1145/3320060.

[7] Yoshua Bengio. “Practical Recommendations for Gradient-Based Training of Deep Architectures”. In: *Neural Networks: Tricks of the Trade: Second Edition*. Ed. by Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller. Berlin, Heidelberg: Springer, 2012, pp. 437–478. ISBN: 978-3-642-35289-8. DOI: 10.1007/978-3-642-35289-8_26.

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

[9] Sander M. Bohte, Joost N. Kok, and Han La Poutré. “Error-backpropagation in temporally encoded networks of spiking neurons”. In: *Neurocomputing* 48.1 (Oct. 2002), pp. 17–37. ISSN: 0925-2312. DOI: 10.1016/S0925-2312(01)00658-0.

[10] Mariusz Bojarski et al. *End to End Learning for Self-Driving Cars*. 2016. arXiv: 1604.07316.

[11] Adam Byerly, Tatiana Kalganova, and Ian Dear. “No routing needed between capsules”. In: *Neurocomputing* 463 (Nov. 2021), pp. 545–553. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2021.08.064.

[12] Natalia Caporale and Yang Dan. "Spike Timing–Dependent Plasticity: A Hebbian Learning Rule". In: *Annual Review of Neuroscience* 31.1 (2008), pp. 25–46. DOI: 10.1146/annurev.neuro.31.060407.125639.

[13] Augustin Cauchy. "Méthode générale pour la résolution des systemes d'équations simultanées". In: *Comptes Rendus de l'Academie des Science* 25 (1847), pp. 536–538.

[14] Daniel Coquelin et al. *Accelerating Neural Network Training with Distributed Asynchronous and Selective Optimization (DASO)*. 2021. arXiv: 2104.05588.

[15] Brian Crafton et al. "Local Learning in RRAM Neural Networks with Sparse Direct Feedback Alignment". In: *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. July 2019, pp. 1–6. DOI: 10.1109/ISLPED.2019.8824820.

[16] Wojciech Marian Czarnecki et al. "Understanding Synthetic Gradients and Decoupled Neural Interfaces". In: *Proceedings of the 34th International Conference on Machine Learning*. PMLR, July 2017, pp. 904–912.

[17] Peter Dayan and Laurence F. Abbott. *Theoretical neuroscience: computational and mathematical modeling of neural systems*. Cambridge, Massachusetts: MIT Press, 2001. ISBN: 0-262-04199-5.

[18] Jia Deng et al. "ImageNet: A large-scale hierarchical image database". In: *IEEE Conference on Computer Vision and Pattern Recognition*. June 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[19] Alexey Dosovitskiy et al. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *International Conference on Learning Representations*. 2021.

[20] John Eccles. "From Electrical to Chemical Transmission in the Central Nervous System". In: *Notes and Records of the Royal Society of London* 30.2 (1976), pp. 219–230. ISSN: 0035-9149.

[21] B. M. Forrest et al. "Implementing Neural Network Models on Parallel Computers". In: *The Computer Journal* 30.5 (Oct. 1987), pp. 413–419. ISSN: 0010-4620. DOI: 10.1093/comjnl/30.5.413.

[22] Charlotte Frenkel, Martin Lefebvre, and David Bol. "Learning Without Feedback: Fixed Random Learning Signals Allow for Feedforward Training of Deep Neural Networks". In: *Frontiers in Neuroscience* 15 (Feb. 2021). ISSN: 1662-4548. DOI: 10.3389/fnins.2021.629892.

[23] Charlotte Frenkel, Jean-Didier Legat, and David Bol. "A 28-nm Convolutional Neuromorphic Processor Enabling Online Learning with Spike-Based Retinas". In: *IEEE International Symposium on Circuits and Systems (ISCAS)*. Oct. 2020, pp. 1–5. DOI: 10.1109/ISCAS45731.2020.9180440.

[24] Ross Girshick et al. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2014, pp. 580–587.

[25] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, May 2010, pp. 249–256.

[26] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[27] Stephen Grossberg. "Competitive learning: From interactive activation to adaptive resonance". In: *Cognitive Science* 11.1 (Jan. 1987), pp. 23–63. ISSN: 0364-0213. DOI: 10.1016/S0364-0213(87)80025-3.

[28] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016, pp. 770–778.

[29] Kaiming He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. Dec. 2015, pp. 1026–1034.

[30] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. *Neural networks for machine learning lecture 6a overview of mini-batch gradient descent*. 2012.

[31] Geoffrey Hinton et al. "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups". In: *IEEE Signal Processing Magazine* 29.6 (Nov. 2012), pp. 82–97. ISSN: 1053-5888. DOI: 10.1109/MSP.2012.2205597.

[32] Geoffrey E. Hinton et al. *Improving neural networks by preventing co-adaptation of feature detectors*. 2012. arXiv: 1207.0580.

[33] Hunsberger, Eric. "Spiking Deep Neural Networks: Engineered and Biological Approaches to Object Recognition". PhD thesis. 2018.

[34] Daniele Ielmini. "Brain-inspired computing with resistive switching memory (RRAM): Devices, synapses and neural networks". In: *Microelectronic Engineering* 190 (Apr. 2018), pp. 44–53. ISSN: 0167-9317. DOI: 10.1016/j.mee.2018.01.009.

[35] J. R. Ipsen and A. D. H. Peterson. "Consequences of Dale's law on the stability-complexity relationship of random neural networks". In: *Physical Review E* 101.5 (May 2020), p. 052412. ISSN: 2470-0045, 2470-0053. DOI: 10.1103/PhysRevE.101.052412.

[36] Max Jaderberg et al. "Decoupled Neural Interfaces using Synthetic Gradients". In: *Proceedings of the 34th International Conference on Machine Learning*. PMLR, July 2017, pp. 1627–1635.

[37] Melvin Johnson et al. "Google's Multilingual Neural Machine Translation System: Enabling Zero-Shot Translation". In: *Transactions of the Association for Computational Linguistics* 5 (Oct. 2017), pp. 339–351. ISSN: 2307-387X. DOI: 10.1162/tacl_a_00065.

[38] Andrzej Kasiński and Filip Ponulak. "Comparison of supervised learning methods for spike time coding in spiking neural networks". In: *International Journal of Applied Mathematics and Computer Science* 16.1 (2006), pp. 101–113. ISSN: 1641-876X.

[39] Nitish Shirish Keskar et al. "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima". In: *5th International Conference on Learning Representations*. ICLR, 2017.

[40] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations*. Ed. by Yoshua Bengio and Yann LeCun. ICLR, 2015.

[41] Jinia Konar, Prerit Khandelwal, and Rishabh Tripathi. "Comparison of Various Learning Rate Scheduling Techniques on Convolutional Neural Network". In: *IEEE International Students' Conference on Electrical, Electronics and Computer Science (SCEECS)*. Feb. 2020, pp. 1–5. DOI: 10.1109/SCEECS48394.2020.94.

[42] Alex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. Tech. rep. 2009, p. 60.

[43] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012, pp. 1097–1105.

[44] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* 521.7553 (May 2015). ISSN: 1476-4687. DOI: 10.1038/nature14539.

[45] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[46] Dong-Hyun Lee et al. "Difference Target Propagation". In: *Machine Learning and Knowledge Discovery in Databases*. Ed. by Annalisa Appice et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 498–515. ISBN: 978-3-319-23528-8. DOI: 10.1007/978-3-319-23528-8_31.

[47] Qianli Liao, Joel Leibo, and Tomaso Poggio. "How Important Is Weight Symmetry in Backpropagation?" In: *Proceedings of the AAAI Conference on Artificial Intelligence* 30.1 (Feb. 2016). ISSN: 2374-3468.

[48] Timothy P. Lillicrap et al. "Random synaptic feedback weights support error backpropagation for deep learning". In: *Nature Communications* 7.1 (Nov. 2016), p. 13276. ISSN: 2041-1723. DOI: 10.1038/ncomms13276.

[49] Jonathan Long, Evan Shelhamer, and Trevor Darrell. "Fully Convolutional Networks for Semantic Segmentation". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015, pp. 3431–3440.

[50] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. "Rectifier nonlinearities improve neural network acoustic models". In: *ICML Workshop on Deep Learning for Audio, Speech, and Language Processing (WDLASL)*. 2013.

[51] Vinícius Gonçalves Maltarollo, Káthia Maria Honório, and Albérico Borges Ferreira da Silva. *Applications of Artificial Neural Networks in Chemical Problems*. IntechOpen, Jan. 2013. ISBN: 978-953-51-0935-8. DOI: 10.5772/51275.

[52] Marvin Minsky. "Steps toward Artificial Intelligence". In: *Proceedings of the IRE* 49.1 (Jan. 1961), pp. 8–30. ISSN: 0096-8390. DOI: 10.1109/JRPROC.1961.287775.

[53] Hesham Mostafa, Vishwajith Ramesh, and Gert Cauwenberghs. "Deep Supervised Learning Using Local Errors". In: *Frontiers in Neuroscience* 12 (2018), p. 608. ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00608.

[54] Vinod Nair and Geoffrey E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *International Conference on Machine Learning (ICML)*. 2010, pp. 807–814.

[55] Y. E. Nesterov. "A method for solving the convex programming problem with convergence rate O(1/k^2)". In: *Soviet Mathematics. Doklady* 27 (1983), pp. 543–547.

[56] Arild Nøkland. "Direct Feedback Alignment Provides Learning in Deep Neural Networks". In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. NIPS'16. Red Hook, NY, USA: Curran Associates Inc., Dec. 2016, pp. 1045–1053. ISBN: 978-1-5108-3881-9.

[57] Arild Nøkland and Lars Hiller Eidnes. "Training Neural Networks with Local Error Signals". In: *Proceedings of the 36th International Conference on Machine Learning*. PMLR, May 2019, pp. 4839–4850.

[58] Ignacio Sañudo Olmedo et al. "Dissecting the CUDA scheduling hierarchy: a Performance and Predictability Perspective". In: *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Apr. 2020, pp. 213–225. DOI: 10.1109/RTAS48715.2020.000-5.

[59] Alexander G. Ororbia and Ankur Mali. "Biologically Motivated Algorithms for Propagating Local Target Representations". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.01 (July 2019), pp. 4651–4658. ISSN: 2374-3468. DOI: 10.1609/aaai.v33i01.33014651.

[60] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019, pp. 8024–8035.

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

[62] B. T. Polyak. "Some methods of speeding up the convergence of iteration methods". In: *USSR Computational Mathematics and Mathematical Physics* 4.5 (Jan. 1964), pp. 1–17. ISSN: 0041-5553. DOI: 10.1016/0041-5553(64)90137-5.

[63] Ludovic Roux et al. "Mitosis detection in breast cancer histological images An ICPR 2012 contest". In: *Journal of Pathology Informatics* 4 (May 2013), p. 8. ISSN: 2229-5089. DOI: 10.4103/2153-3539.112693.

[64] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0.

[65] Andrew M. Saxe, James L. McClelland, and Surya Ganguli. *Exact solutions to the nonlinear dynamics of learning in deep linear neural networks*. 2014. arXiv: 1312.6120.

[66] Juergen Schmidhuber. "Deep Learning in Neural Networks: An Overview". In: *Neural Networks* 61 (Jan. 2015), pp. 85–117. ISSN: 08936080. DOI: 10.1016/j.neunet.2014.09.003.

[67] Andrew Senior et al. "An empirical study of learning rates in deep neural networks for speech recognition". In: *IEEE International Conference on Acoustics, Speech and Signal Processing*. 2013, pp. 6724–6728. DOI: 10.1109/ICASSP.2013.6638963.

[68] Leslie N. Smith. "Cyclical Learning Rates for Training Neural Networks". In: *IEEE Winter Conference on Applications of Computer Vision (WACV)*. Mar. 2017, pp. 464–472. DOI: 10.1109/WACV.2017.58.

[69] Ilya Sutskever et al. "On the importance of initialization and momentum in deep learning". In: *Proceedings of the 30th International Conference on Machine Learning*. PMLR, May 2013, pp. 1139–1147.

[70] Christian Szegedy et al. "Rethinking the Inception Architecture for Computer Vision". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016, pp. 2818–2826.

[71] Aboozar Taherkhani et al. "A review of learning in biologically plausible spiking neural networks". In: *Neural Networks* 122 (Feb. 2020), pp. 253–272. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2019.09.036.

[72] Jianshi Tang et al. "Bridging Biological and Artificial Neural Networks with Emerging Neuromorphic Devices: Fundamentals, Progress, and Challenges". In: *Advanced Materials* 31.49 (2019), p. 1902761. ISSN: 1521-4095. DOI: 10.1002/adma.201902761.

[73] Yong Tang et al. "Total regional and global number of synapses in the human brain neocortex". In: *Synapse* 41.3 (2001), pp. 258–273. ISSN: 1098-2396. DOI: 10.1002/syn.1083.

[74] Amirhossein Tavanaei et al. "Deep learning in spiking neural networks". In: *Neural Networks* 111 (Mar. 2019), pp. 47–63. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2018.12.002.

[75] Qi Wang et al. "A Comprehensive Survey of Loss Functions in Machine Learning". In: *Annals of Data Science* (Apr. 2020). ISSN: 2198-5812. DOI: 10.1007/s40745-020-00253-5.

[76] James C. R. Whittington and Rafal Bogacz. "Theories of Error Back-Propagation in the Brain". In: *Trends in Cognitive Sciences* 23.3 (Mar. 2019), pp. 235–250. ISSN: 1364-6613. DOI: 10.1016/j.tics.2018.12.005.

[77] D. Randall Wilson and Tony R. Martinez. "The general inefficiency of batch training for gradient descent learning". In: *Neural Networks* 16.10 (Dec. 2003), pp. 1429–1451. ISSN: 0893-6080. DOI: 10.1016/S0893-6080(03)00138-2.

[78] Quanzeng You et al. “Image Captioning With Semantic Attention”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016, pp. 4651–4659.

[79] Semir Zeki. “A massively asynchronous, parallel brain”. In: *Philosophical Transactions of the Royal Society B: Biological Sciences* (May 2015). DOI: 10.1098/rstb.2014.0174.

A. Appendix

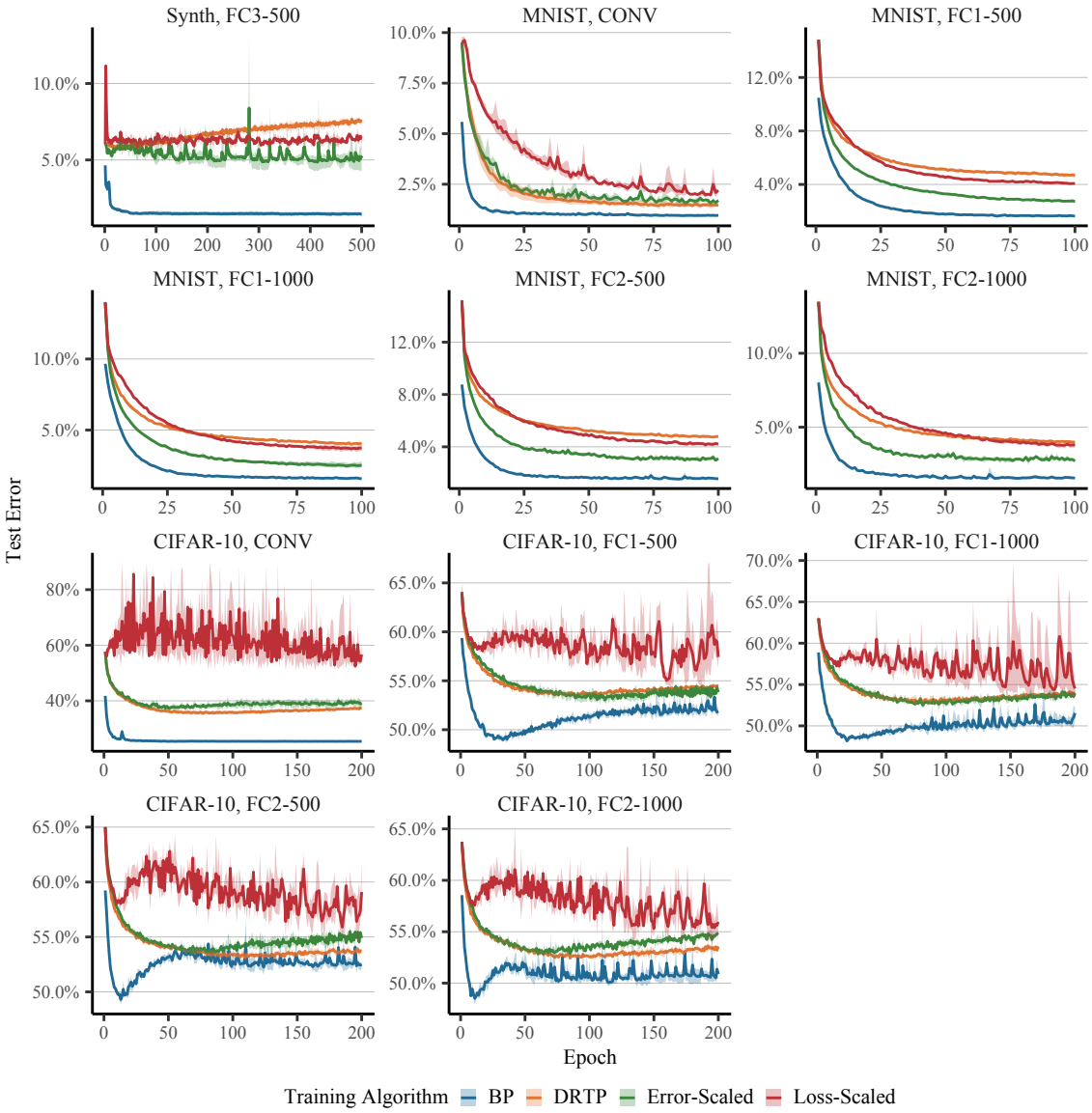


Figure A.1.: Test error over the course of the training for different training algorithms. Aggregated over three independent runs. An extension of Figure 5.5 including all network topologies.