Building neural networks layer by layer

Adanna V. Obibuaku*, Stephen McGough, David Towers
School of Computer Science
Newcastle University University
(a.obibuaku,stephen.mcgough,d.towers2)@newcastle.ac.uk

Abstract

Backpropagation is the traditional learning algorithm used to train neural networks. However, due to the complexity of a model and behaviour of backpropagation certain issues arise such as internal covariate shift, exploding and vanishing gradients. There are strategies (batch normalisation, residual networks etc) used in the deep learning industry to combat issues of backpropagation. However, these strategies focus on modifying the neural network model rather than the traditional learning algorithm. Forward-thinking is an alternative learning algorithm that deviates away from the issues caused by backpropagation. Forward-thinking is a greedy algorithm that constructs a neural network layer by layer, choosing the next layer in such a way that it provides the best parameters (weights and biases) that fit the layer. The behaviour of this algorithm increases the performance in both the model's accuracy and training time, even without the competence of hyperparameter tuning. This paper presents and validates the forward-thinking algorithm by utilising it across industry-standard neural network architectures. Furthermore, the paper illustrates the flexibility of the algorithm by extending it to an approach called: Forward-thinking with multiple source transfer learning. This is where each layer is trained using a unique subset of data from the original dataset. This paper looks into the performance of the forward-thinking algorithm when utilising similar strategies such as batch normalisation and residual networks that perform well on backpropagation. It is found that forward-thinking converges faster to a higher accuracy than backpropagation. In addition, the extended approach increases the training time by at least 50% given that the dataset for each layer is 20% smaller than the original dataset.

Keywords: Shallow neural networks, Deep neural networks, backpropagation, transfer learning, Greedy algorithm

1 Introduction

Deep neural networks (DNN) models provide an approximation for a given problem. Backpropagation is a traditional learning algorithm used to train neural networks [2]. Due to the complexity of such models and the behaviour of backpropagation, various problems can arise. These problems include internal covariate shift [1], vanishing and exploding gradients that result in the training process becoming slower.

In recent years, world-renowned DNN architectures have been developed to combat the issues caused by backpropagation. This includes residual networks and transformers. The current trends focus on reconstructing the DNN architecture to combat the problems of backpropagation. However, limited solutions are focusing on the origin of these issues -backpropagation. This paper explores

*Corresponding author.: Email address: aaddanna@outlook.com

Preprint. Under review.

the possibility of training a neural network using a forward-thinking algorithm. Forward-thinking deviates away from the problems related to backpropagation. Given the behaviour of the forward-thinking algorithm, it is objectively a greedy-based algorithm. The algorithm constructs a neural network layer by layer, choosing the next layer in such a way that it provides the best parameters (weights and biases) that fit the layer. Once a layer has been trained, its parameters are frozen and added to the next sequential layer. Thus reaching the best objective for a particular learning problem.

Backpropagation by design is sequential, in that the algorithm must complete a full forward pass to calculate the loss followed by a backward pass to find the optimal parameters. This behaviour becomes computationally expensive in a DNN with many layers. For this reason, this paper explores the possibility of training deep neural networks without backpropagation. Greedy approaches such as the forward-thinking algorithm developed by [3] are proven to effectively train DNNs. The proposed approach will extend this algorithm to multiple source transfer learning whereby the sample dataset is subdivided into unique sets for each layer. These approaches will be compared and discussed in experiments.

Batch normalisation and residual networks are strategies created with the idea of backpropagation. Batch normalisations are used to solve the internal covariate shift and residual networks are used to solve the vanishing and exploding gradient problem. These issues diminish when forward-thinking is applied. This paper further investigates how forward-thinking performs on DNN architectures, using similar strategies that work for backpropagation. This is then further explored in experiments. In addition, this paper will explain how forward-thinking relates to the key ideas in the cognitive process from which these approaches are derived. Namely the idea that each layer can be visualised as an independent machine learning (ML) problem. This will be discussed in the background.

This paper aims to explore how well greedy algorithms such as forward-thinking, and forward-thinking with multiple source learning perform in comparison to the traditional training method. Furthermore, this paper explores standard strategies applied to improve the performance of backpropagation and performance using forward-thinking. The contributions of this paper can be summarised below:

- To strengthen the validity of forward-thinking, this paper extends the experiments of forward-thinking to train industry-standard DNNs architectures with different datasets.
- Illustrates that the algorithm: Feed-forward algorithm can be extended to a multiple-source transfer learning, whereby each layer is trained on a unique dataset.
- Forward-thinking can be efficiently used to train deep standalone neural network architectures without specialised strategies such as batch normalisation and residual networks.

2 Theoretical Background

2.1 Backpropagation

Backpropagation is explained in more detail in [1]. In general terms, backpropagation is used to update the parameters in a network. The inputs and outputs of neurons influence the inputs and outputs of their ongoing connections. To find the optimal proportional corresponding parameters of the network, the chain rule is utilised. This formula is shown in appendix 6.2 equation 5. The chain rule reaches back into the network to find the gradients of each neuron in the network.

2.2 Vanishing and exploding gradients

Exploding and vanishing gradient is a problem that occurs during backpropagation. The more layers a DNN has, the more it becomes susceptible to vanishing and exploding gradients. This particularly affects neurons that reside in earlier layers of the network. The gradient is a product of derivatives of neurons positioned at later layers of the DNNs. Therefore, the higher a neuron is in a network, the more derivative terms are required to calculate the gradient. The gradient is used to calculate the change needed to update the neuron's weight. A small gradient leads to a small change. Updating the weight by a small value means the weight converges slowly to an optimal value. This creates implications for the remaining networks, thus impairing the ability of the network to learn. This is known as the vanishing gradient problem. Exploding gradients are the opposite case where subsequent large derivative terms lead to exponentially exploding gradients.

2.3 Transfer learning in forward-thinking

Transfer learning is a fundamental technique in forward-thinking that drives the performance of the algorithm. Forward-thinking, proposes that each layer in DNN should act as an independent ML model. Transfer learning improves the performance of a target model $f(x)$. This is achieved by transferring the knowledge contained from a set of different model sources $[f_1(x_1)...f_n(x_n)]$ that have related source task to the target model, $f(x)$ [4]. This is achieved by adding the parameters learned from different source models to the target model $f(x)$. Forward-thinking utilises transfer learning as new layers are sequentially added while transferring the knowledge from prior layers to the new layer. More information about transfer learning can be found in appendix 6.3

2.4 Forward-thinking

The forward-thinking algorithm begins with a defined base DNN containing the initial input and output layers including a defined set of layers to be added to the network. In each iterative process, a new layer is added and then trained. Once a layer has been trained the parameters of the layer are frozen. This process keeps iterating until the DNN reaches an optimal accuracy or when there are no more layers to add. Freezing the previous weights means that in each iteration only a shallow neural network is trained. As the previous layers are frozen, backpropagation is not needed to reach back into the network to train the layers.

To understand forward-thinking, consider the DNN shown in appendix 6.2 in figure 12 and its corresponding mathematical notation shown in equation (1). The neural network is denoted as $f(x)$ and has 2 hidden layers in addition to the input and output layers.

$$f(x) = l_{out}(l_2(l_3(x))) \tag{1}$$

where l_3 denotes the 1^{st} layer, l_2 denotes the 2^{nd} layer and l_{out} denotes the output layer This loss function for $f(x)$ is defined in equation (2).

$$Loss(\hat{Y}, Y) = \sigma(l_{out}(l_2(l_3(X))), Y) \tag{2}$$

where Y is the actual output and $\hat{Y}$ is the model predictions. The loss is a composition of all layer parameters defined in a DNN, $f(x)$. $f(x)$ consists of successive linear transformations. Each layer is an intermediate transformation that transforms the data into a new representation. In backpropagation, the updated weights in an individual layer affect the earlier layers while training. In forward-thinking, each layer behaves like an independent ML model. Therefore, the loss for the 2 intermediate layers is defined in the equation (3).

$$\begin{aligned} Loss_3 &= \sigma(l_{out}(l_3(x)), Y) \\ Loss_2 &= \sigma(l_{out}(l_2(l_3)), Y) \end{aligned} \tag{3}$$

This algorithm for forwarding thinking is illustrated in appendix 6.4. Forward-thinking focuses on optimising the loss for a single layer. Rather than optimising the loss for a whole network defined in equation (2).

The main steps of forward-thinking are summarised as follows [3]:

1. *Initialising the network* The network is initialised with an input and output layer. The output layer, loss function and activation function are selected appropriately with the task to solve. The layer of the networks can be declared at compile time or run time. In compile time, the model will define a sequence of layers that will be added during training. At runtime, during training, an input of the next required layer is needed. Weight initiation can be optionally applied to each layer.
2. *Adding a layer* Once a layer is trained and frozen a new layer is inserted between the previously trained layer and the output layer. The new layer acts as a shallow network taking in trained parameters from the previous layer as inputs. Thereby the knowledge from the previously trained layer is transferred into the new layer. It's assumed that a new layer will

have a different number of output neurons/features than the previous layer. To accommodate this, the weights from the output layer are disregarded to allow the new layer to be added. This causes a temporary dip in accuracy.

3. *Iterating* In forward-thinking each layer is added layer by layer. The algorithm can stop either when there are no more layers to add or when inserting a new layer does not improve the performance.

In some cases, it is not always necessary to use multiple layers to generalise a function. For a neural network with a continuous target value, one hidden layer is sufficient. This stems from the universal theorem which states that a continuous function can be approximated with one layer with any accuracy [12], [13]. Such networks are often referred to as universal approximations. The challenge with these networks is finding a good approximation of a target function.

2.5 Shallow networks in forward-thinking

In Forward-thinking each independent layer acts as a shallow network, whereby at each iteration there is a defined:

Input layer: The prior layer $l_{\text{n-1}}$ that was trained and frozen.

Hidden layer: The new layer l_{n} added is going to be trained.

Output layer: In each iteration the prior output layer l_{out}' is removed and the output layer l_{out} is defined and initialised again.

Shallow networks are neural networks with 1 or 2 hidden layers. Shallow networks are less susceptible to issues such as invariant covariant shift, vanishing and exploding gradients. In shallow networks, the internal covariate shift [21] is not a problem since the fluctuations are within a narrow range and do not pose the problem of a moving target. Issues such as vanishing and exploding gradients are less likely to occur in shallow networks because there are fewer derivative terms to calculate the gradient. In conclusion, forward-thinking reduces the issues presented in backpropagation because prior layers are frozen. As a result, each iteration has fewer layers to propagate through.

2.6 Forward-thinking with batch normalisation

Batch normalisation was created using an idea similar to forward-thinking as every layer in a network can be seen as an independent machine learning problem. In the ML industry, inputs are often normalised before training. Therefore, in a DNN, the outputs of layers should be normalised before the next layer takes those outputs as inputs. In addition, the batch normalisation procedure includes learning parameters (λ and β) used to shift and scale the normalised inputs. These learning parameters transform the inputs into a distribution the DNN learns best from. The λ and β parameters are trained alongside the weights of a DNN. This process is further explained in appendix 6.6. In a DNN architecture that uses batch normalisation, a given layer consists of $[l_n, b_n]$. A batch normalisation layer b_n is applied to normalise the convolutional layer $b_n(l_n)$. The parameters for a given batch normalisation are not frozen. This is because the next sequential layers from l_{n+1} will need to take in the normalised input of l_n. Therefore, the batch normalisation parameters will need to be trained to fit a particular input distribution that l_{n+1} learns best from.

2.7 Proposed approach: Forward-thinking with multi-source transfer

Forward-thinking can be extended to use multi-source transfer learning. In this approach the training dataset X is divided into partitions such that $P_1 \cup P_2 \cup \dots P_n = X$. Each layer is trained on a partitioned data set. Ideally, each training data set should contain the marginal and conditional distribution i.e same number of classes.

2.8 Backpropagation vs Forward-thinking

Forward-thinking reduces the issues that appear in backpropagation. This is due to the nature of the forward-thinking algorithm. In forward-thinking, once a layer is trained the layer is frozen. The parameters remain the same, therefore the chain rule is not needed to update the weights in the

upper layer. In backpropagation, the weights in each layer change, consequentially affecting the distributions of inputs in subsequent layers. This is known as the internal covariate shift. In forward-thinking, since the parameters are frozen, the distribution of inputs does not shift. Furthermore, Forward-thinking is neither less nor more susceptible to overfitting than backpropagation. Overfitting is further explained in appendix 6.7 including the L_2 regularisation method to reduce overfitting.

2.9 Related work

2.9.1 Extreme neural networks

Extreme neural networks (ENN) are an example of a neural network architecture that does not utilise backpropagation to train and uses the idea of the universal theorem. ENN has a single-hidden fully-connected layer that is randomly initialised with weights and bias and then frozen. The weights of the output layers are analytically determined [20]. The proposed algorithm uses the Moore-Penrose generalised inverse and minimum least-squares solutions that determine the weights of the output layers. A singular target y_i from a sample train labels Y is calculated as follows:

$$\hat{y}_i = \beta_{\mathrm{i}}(\sum_i w_i x_i + b) \tag{4}$$

The β_i is the weight parameter for a given neuron in the output layer. The intermediate layer can be seen as a transformation of the input y_i'$= t(x_i) = y_i$. In that the input data is not used directly to find the weights of the output layer, rather the intermediate values created by the hidden layers are used to find the corresponding target y_i, for a given input x_i

In forward-thinking, a shallow neural network model is trained at every iteration. In ENN the randomised weights are mapped to the output layer, whereas in forward-thinking the trained weights are mapped to the next sequential layer. Ultimately the accuracy of a previous layer is carried over to the next layer.

2.9.2 Orthogonal matching pursuit OMP

There have been extended architectures of ENNs that modify the method of how the output of the weight layers is learned. In particular, sources [14],[15] use a greedy algorithm called OMP algorithm that is used for selecting the weights β for the output layer. The OMP was originally used for signal recovery. This method is used in fields such as compressed sensing and sparse regression, where data needs to be recovered [16]. OMP algorithms can be generalised to any system that is overdetermined or underdetermined. The problem to be solved in the ENN is $y = \beta x$ (output weights) which can be viewed as an overdetermined system. To start the algorithm, the residual r is initially set to $y.dot(1)$. The iterative process begins by selecting a feature within the input space of x such that it maximises the correlation between the feature in the dictionary and the residual. A higher correlation between the feature and the residual means that a particular feature captures the patterns in the data. In each iteration, the residual is expected to be smaller than the previous residual.

In forward-thinking, rather than selecting an individual weight at each iteration, a set of weights for a given layer is solved at iteration and then frozen.

2.9.3 Cascade correlation

Another example of a greedy algorithm is the cascade correlation algorithm [18] proposed to address the issue of backpropagation. This is similar to OMP, in that weight units are selected, and frozen in each iteration. Whereas in forward-thinking a layer is trained in each iteration. The Cascade correlation was originally constructed for the fully-connected network. In 2018, cascade correlation was extended to work for convolutional networks [17].

2.9.4 Quantifying transferability in Multiple-source transfer

The paper, A mathematical framework for Quantifying transferability in Multiple-source transfer learning uses a similar approach to forward thinking with multiple source transfer whereby the training set is divided into N disjoint subsets. Each training set corresponds to N models. Each model has a different task meaning each model is classifying different labels. For the N models,

model 0 will be used as the target model/task while the $1..N$ models will be used as source tasks for transferring knowledge. The target model can be trained using the proposed algorithm which linearly combines the source models 5trained on an individual task. The test accuracy will be tested using the target model [19]. Additionally, they propose a new mathematical framework to measure the transferability of multi-source problems.

3 Methodology

3.1 Network Architectures

To validate forward-thinking, the experiments will compare the performance of backpropagation and forward-thinking on DNN architectures listed below:

Fully-connected neural network: In addition to the input and output layers this architecture consists of 3 hidden layers with 150, 100 and 50 layers. This architecture is applied to the MNIST dataset.

VGG11: This is an industry-standard DNN model proposed in the paper [7]. In Visual Geometry Group (VGG), as the number of layers increases, the number of parameters/filters within the layer decreases. This architecture is applied to SVHN and CIFAR10 datasets

Residual network (Resnet18): This is an industry-standard DNN model first proposed in the paper [8] used to combat vanishing and exploding gradient problems. The residual network uses a concept called skip connections. This work differently from traditional neural network layers more information can be found in appendix 6.5

The implementation of the residual network is different from other networks. Therefore, the algorithm forward-thinking is applied differently to residual networks. This is because in each iteration it does not train a shallow network, as each layer (except the first layer) is a composition of 2 residual blocks. Therefore in each iteration, a DNN is trained as opposed to a shallow network. Appendix 6.5 figure 14 shows an example of the forward-thinking algorithm applied to a residual network.

3.2 Implementation

The DNN architecture is trained end-to-end using backpropagation and forward-thinking. The performance metrics (loss, time elapsed (milliseconds), test accuracy, train accuracy) are logged and later used in experiments. The underlying architecture of the DNN is modified to allow compatibility with the forward-thinking algorithm. This is done by defining an independent layer as a list of operations, c_n. Where c_n is composed of the fully-connected/convolutional layer itself and/or the layers corresponding operations. For example, a convolutional layer with an applied max pooling [9] operations $max_n(l_n(x))$ will be defined as $[l_n, max_n]$. The intermediate hidden layers [c_1 ... c_n] will be trained sequentially as discussed in section 2.4.

Each DNN architecture discussed in this paper uses the relu activation function [10] and the Kailming weight initialisation [6] strategy is applied. Weight initialisation is a process which sets the initial starting values of the DNN weights before the learning algorithm begins. This is further explained in appendix 6.9. The extended approach of forward-thinking: forward-thinking with multiple source transfer is validated on the VGG11 architecture.

Further experiments are conducted to explore the performance of forward-thinking when training a DNN architecture with batch normalisation operations. To support this experiment, each convolutional DNN has another variant architecture where batch normalisation is used. Therefore, each convolutional layer l_n is followed by a batch layer b_n. In contrast to l_n, the learning parameters are not frozen in b_n during training. The reason being that the normalised distribution of l_n directly affects the next sequential layer l_{n+1} as explained in section 2.6.

3.3 Experiments

3.3.1 Datasets used in experiments

For exploring and comparing the performance of the learning algorithms backpropagation and forward-thinking the MNIST, SVHN and CIFAR10 datasets are used. These are explained further explained in appendix 6.13

Further experiments are done on forward-thinking with multiple source transfer. This algorithm is tested on the VGG11 DNN architecture. The behaviour of this algorithm requires that each layer has a unique dataset that is a subset of the main training data. To fulfil this requirement the CIFAR10 dataset and SVHN are divided into partitions. These datasets were divided as follows:

CIFAR10: The CIFCAR10 is divided into 5 partitions. The original CIFAR10 dataset consists of 50,000 training images. This dataset was split into 5 data partitions consisting of 10,000 images. The data partitions consisted of an equal distribution of target images, in that there are 1000 training images for each class. These data partitions for the VGG11 are divided across the layers as follows: The first 4 data partitions $[d_1 \dots d_4]$ are used to train the first 4 layers hidden layers $[l_1 \dots l_4]$, while the 5^{th} data partition d_5 is used to train the remaining 7 layers.

SVHN: For the SVHN dataset the additional 270000 images from the extra dataset are used for training. This dataset is divided into 9 data partitions each containing $30,000$ images. The data partitions $[d_1 \dots d_8]$ were used to train the first 9 layers $[l_1 \dots l_8]$ of VGG11 while the remaining layers $[l_{10}, l_{11}]$ were trained using d_9.

3.3.2 Experimental settings

To allow a fair comparison between backpropagation and forward-thinking, the same neural network architecture is used. However, the hyperparameters are tuned for a particular learning algorithm. Specifically, the number of epochs used. Forward-thinking provides more flexibility than backpropagation as improvements can be made by taking advantage of the fact that each layer acts as an independent ML problem. Therefore, learning hyperparameters such as learning rate, the number of epochs, batch size, and even the loss function can be updated for a new layer to provide the locally optimal choice. In experiments, the number of epochs, learning rate and batch size are set to 2, 0.01 and 64 respectively. For simplicity, these hyperparameters were kept the same for each iteration. For backpropagation, the learning rate is set to 0.01 and uses a 60 number of epochs for training. Backpropagation and forward-thinking both use stochastic gradient descent (SGD) to train the DNNs. [11] provides a further explanation of an SGD. Furthermore, both algorithms use the L_2 regularisation with λ set to $5 * 10^4$. Appendix 6.8 provides a detailed explanation of the L_2 regularisation technique.

Forward-thinking can either stop iterating when there are no more layers to add or when the accuracy is not improving. To allow a better comparison of backpropagation, the current implementation of forward-thinking stops when there are no more layers to add. Moreover, in experiments, the performance (test accuracy, train accuracy, loss etc) of each learning algorithm is measured against the time elapsed (milliseconds) rather than using the number of epochs. This reason is that the behaviour of each algorithm is different. Therefore, the number of epochs in forward-thinking does not directly translate to the number of epochs used in backpropagation.

Residual networks and batch normalisation are strategies used in the industry that works best for backpropagation. Further experiments, apply these same strategies to forward-thinking to validate whether it offers the same improvements.

3.3.3 Accelerating forward-thinking

Simply training a DNN using a forward-thinking algorithm on architectures does not illustrate how effective the algorithm is. To further illustrate and validate the full advantage of forward-thinking, additional experiments are performed with the architectures of the neural networks modified as follows:

Remove the batch normalisation layers: Batch normalisation has been proven to increase the accuracy of backpropagation [21]. This paper conducts experiments that find out whether

batch normalisation offers the same increase in performance when training DNNs with the forward-thinking algorithm. In addition, this paper compares the original approach of forward-thinking when training a DNNs with batch normalisation layers explained in section 2.6 with other approaches such as:

- Forward-thinking with no batch normalisation layers.
- The batch normalisation learning parameters (λ and β) are frozen along with the weights of a layer.
- The batch normalisation learning parameters are omitted. This means that the scale and shift step does not occur. As each layer acts as an ML learning problem, the normalisation procedure should be applied as such, where scale and shift step is not applied.

In experiments, it is found that forward-thinking can achieve good accuracy without the need for batch normalisation layers.

Remove weight initialization: In backpropagation, for the algorithm to converge, the weights must be initialised. Experiments are made to find the performance of forward-thinking with and without weight initialisation. In experiments, it is found that the algorithm reaches the same performance and has the potential to perform better without weight initialization.

4 Results

4.1 Forward-thinking on deep convolutional DNN results

4.1.1 Results of forward-thinking

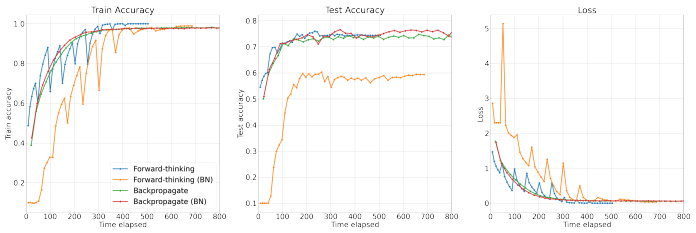


Figure 1: Compares the performance of backpropagation and forward-thinking by using the training, test accuracy and the loss. The learning algorithms were applied to the Resnet18 DNN architecture, to solve the CIFAR10 image recognition task.

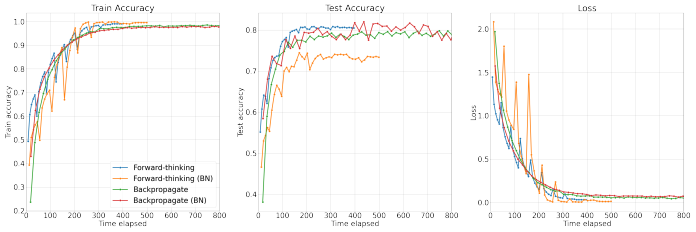


Figure 2: Compares the performance of backpropagation and forward-thinking by using the training, test accuracy and the loss. The learning algorithms were applied to the VGG11 DNN architecture, to solve the CIFAR10 image recognition task.

Figures 1 and 2 show the result of forward-thinking and backpropagation across the architectures of Resnet18 and VGG11. In addition, it compares the performance of the learning algorithms applied to the same DNN architectures when they utilise batch normalisation layers. These DNN architectures are used to solve the CIFAR image recognition task. Appendix 6.10 shows the learning algorithm applied to the SVHN image recognition task. From these figures, it is evident that forward-thinking does not train well on DNN architectures that use batch normalisation layers. From the figures and tables 1 and 2, there is a significant decrease in maximum test accuracy when forward-thinking is applied to a DNN architecture with batch normalisation layers. However, this is the opposite of

Table 1: Maximum test accuracy achieved training SVHN

Maximum test accuracy %				
Learning algorithm	VGG11	VGG11 (BN)	Resnets18	Resnet18 (BN)
Forward-thinking	94.0	73.6	92.0	87.8
Backpropagation	93.6	93.9	92.0	93.0

Table 2: Maximum test accuracy achieved training CIFAR10

Maximum test accuracy %				
Learning algorithm	VGG11	VGG11 (BN)	Resnets18	Resnet18 (BN)
Forward-thinking	80.8	74.1	76.0	59.4
Backpropagation	80.5	82.0	75.1	78.1

backpropagation where there is an average increase of 2% in maximum test accuracy. Forward-thinking reaches a maximum higher accuracy than backpropagation. Furthermore, forward-thinking converges faster to a higher test accuracy. For example, on figure 2 forward-thinking training the VGG11 architecture (without batch normalisation layers) converges to a higher test accuracy of 80.8% compared to 77.3% to backpropagation around the time elapsed < 500 milliseconds. It is worth mentioning, that in forward-thinking a high accuracy is achieved even though not all the layers for the network have been added.

The performance of forward-thinking as with backpropagation is strongly influenced by the dataset and DNN architecture. Forward-thinking performs better on the SVHN datasets when utilising the Resnet18 architecture with batch normalisation layers achieving an accuracy of 87.8%. In contrast to VGG11 with batch normalisation which achieved an accuracy of 73.6%. This contrasts with DNN architectures with batch normalisation layers trained using the CIAF10 dataset, where a higher accuracy of 74.1% is achieved. This is significantly lower in Resnet18 where 59.4% is achieved. Overfitting occurs for both backpropagation and forward-thinking shown within the figures.

Overall, the sequential training of each layer allows forward-thinking to converge faster to a higher accuracy than backpropagation. However, there is a significant performance gap when forward-thinking trains DNNs with batch normalisation. This decrease in accuracy influenced the experiment conducted below which explores the different approaches that forward-thinking can use to train DNNs with batch normalisation.

4.1.2 Results of forward-thinking of batch normalisation

This experiment takes a closer look at how forward-thinking performs with batch normalisation layers. As explained in 3.3.3, this section will take a look at the different methods used in forward-thinking to train DNNs with batch normalisation layers.

On figures (3-4) and (5-6), shows the performance of the forward-thinking algorithm training on across the architectures Resnets11 and VGG11 utilising different batch normalisation techniques to solve the CIFAR10 and SVHN image recognition task respectively. All of the graphs show that forward-thinking performs best on a DNN architecture that does not use a batch normalisation layer. The figures (3 - 6) illustrate that forward-thinking applied to the BN approach performs the worst. The exception is the Resnet18 architecture used to solve SVHN image recognition tasks where the algorithm stalls, and does not converge. In figures (5-6) where forward-thinking is applied on the VGG11 architecture, the second highest test accuracy utilises batch normalisation layers where the learning parameters are frozen in each iteration in addition to the layer in the weights. However, on the Resnet18 architecture shown in figures (3-4), the second highest batch normalisation is utilised with a layer without learning parameters. This is evident that the method of batch normalisation used in forward-thinking depends on the DNN architecture.

In summary, forward-thinking does not train well on DNNs with batch normalisation layers. Batch normalisation is used to prevent the problem of internal covariate shift [21]. In forward-thinking, each iteration trains a shallow network. Therefore, the problem of internal covariate shift becomes less

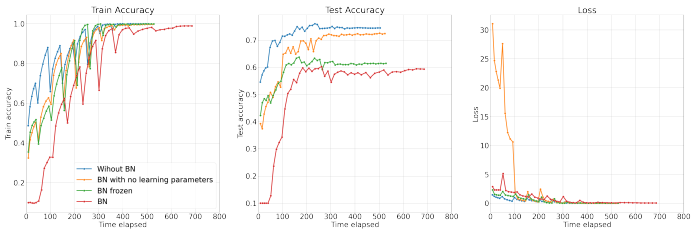


Figure 3: A comparison of how forward-thinking performs with different implementations of the algorithm training a Resnet18 architecture with batch normalisation layers. This architecture is used to solve the CIFAR10 image recognition problem. The different implementations are as follows: BN - The standard implementation of forward-thinking with batch normalisation explained in section 2.6. Without BN - Batch normalisation layers are omitted from the architecture. BN frozen - when learning parameters (λ and β) batch normalisation is frozen in each iteration of forward-thinking. BN no learning parameters - when the learning parameters are not used, therefore the transformation technique in the standard batch normalisation procedure is omitted.

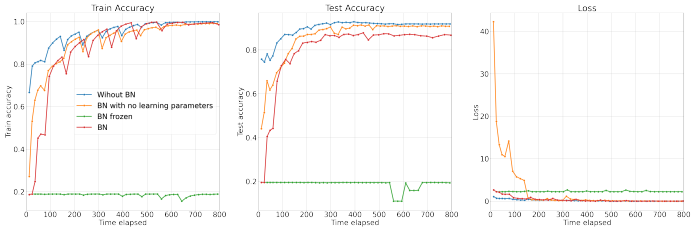


Figure 4: A comparison of how forward-thinking performs with different implementations of the algorithm training a Resnet18 architecture with batch normalisation layers. This architecture is used to solve the SVHN image recognition problem.

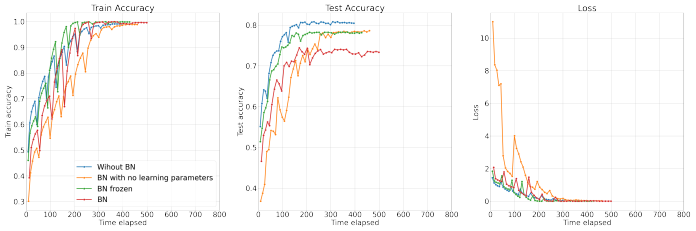


Figure 5: A comparison of how forward-thinking performs with different implementations of the algorithm training a VGG11 architecture with batch normalisation layers. This architecture is used to solve the CIFAR10 image recognition problem.

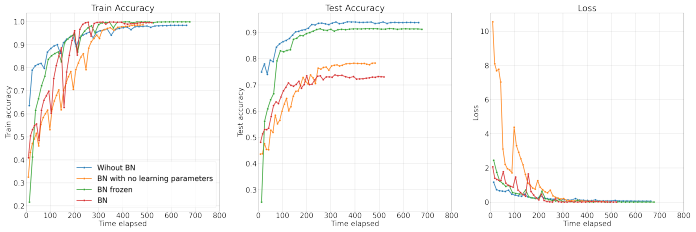


Figure 6: A comparison of how forward-thinking performs with different implementations of the algorithm training a VGG11 architecture with batch normalisation layers. This architecture is used to solve the SVHN image recognition problem.

of a problem as explained in 2.5. Moreover, batch normalisation does not appear to work well with transfer learning in forward-thinking. The decrease in test accuracy is not caused by optimization issues in batch normalisation layers but rather caused by overfitting. However, batch normalisation

offers some regularisation effect, reducing generalisation error [21], [22]. Therefore, this suggests the method in which batch normalisation is used forward-thinking has the potential to be improved.

4.1.3 Results of forward-thinking with weight initialisation

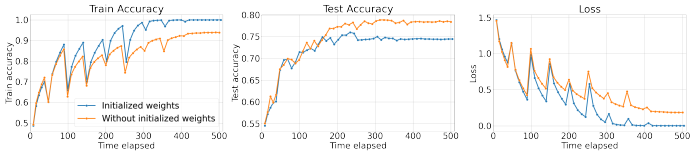


Figure 7: A comparison on forward-thinking with and without weight initialisation on the Resnet18 architecture solving the CIFAR10 image recognition problem.

Figure 7, shows the performance of forward-thinking on a Resnet18 architecture that utilises weight initialisation compared to a Resnet18 architecture that does not use weight initialisation. Figure 7 demonstrates that forward-thinking performance is higher without the use of weight initialisation. This holds across all DNN architectures found in appendix 6.11. Forward-thinking tends to overfit for DNN architecture that uses weight initialisation. This indicates that more regularisation techniques need to be applied when using forward-thinking to train DNN architectures that use weight initialisation. This could potentially increase the test accuracy.

Although, weight initialisation tend to overfit. This experiments illustrates that forward-thinking tends can reach a good accuracy without the use of weight initialisation. Whereas in backpropagation, it tends to stall and does not converge [33].

4.2 Forward-thinking on fully-connected DNN results

4.2.1 Results of forward-thinking

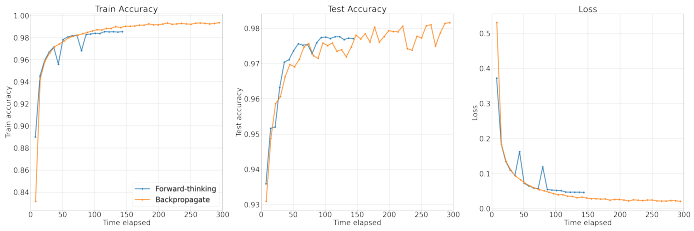


Figure 8: A comparison on forward-thinking with and without weight initializations on the Resnet18 architecture solving the CIFAR10 image recognition problem.

Figure 8 shows the performance of forward-thinking and backpropagation applied on the Full connected DNN architecture to train the MNIST dataset. Forward-thinking reaches a maximum test accuracy of 97.7% at < 150 milliseconds. Backpropagation reaches a maximum test accuracy of 98.2% at < 300 milliseconds. Both learning algorithms perform within a similar range.

4.3 Forward-thinking with multiple source training results

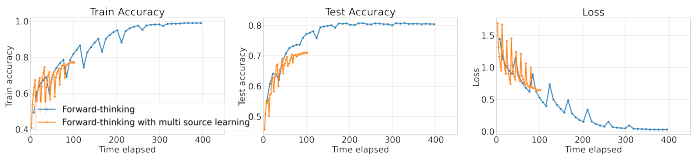


Figure 9: A comparison or forward-thinking vs backpropagation with multiple source transfer on a VGG11 architecture. This is trained using the CIFAR10 dataset.

Figures 9 and 10 show that forward-thinking reaches a higher accuracy than forward-thinking with multiple source training. However, forward-thinking with multiple source transfer trains the DNN

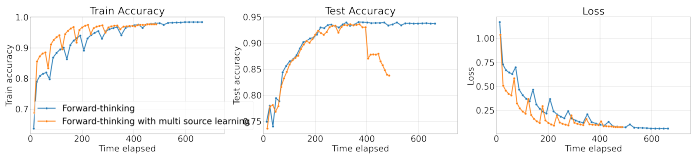


Figure 10: A comparison of forward-thinking vs backpropagation with multiple source transfer on a VGG11 architecture. This is trained using the SVHN dataset.

architecture significantly faster. In figure 9 the forward-thinking algorithm (trained using 50,000) took around 400 milliseconds to finish. The extended approach (trained using 10,000 images at each layer) around 110 milliseconds to finish. In figure 10 the forward-thinking algorithm (trained using 73257 images) took around 650 milliseconds to finish. The extended training using 30,000 images at each layer took around 450 milliseconds to finish. The significant decrease in training time suggests the size of the dataset is directly proportional to the time it takes for the procedure to finish.

The extended approach of forward-thinking does tend to overfit. This is particularly true in the figure 10 where forward-thinking with multiple source transfer decreases in test accuracy at around 400 milliseconds while the training accuracy continues to increase. This illustrates that more regularisation techniques should be enforced when using the extended approach of forward-thinking.

5 Conclusion

A comprehensive set of experiments have been done to prove that the forward-thinking algorithm can be used as an alternative learning algorithm. This algorithm dramatically accelerates the training of DNNs. During training, each layer in a DNN behaves as an independent ML problem, Specifically a shallow neural network. This reduces the likelihood of covariant shift, exploding and vanishing neural networks. Further extensive hyperparameters tuning could be done, to increase the performance of both learning algorithms. However, forward-thinking provides a reasonable increase in performance without much competence in configuring hyperparameters. Further experiments are conducted with a different approach of forward-thinking training DNNs with batch normalisation layers. It is found the method in which batch normalisation is used with forward-thinking; whether the learning parameters are frozen or not frozen, is strongly influenced by the architecture and data used for training. Forward-thinking can train to a good accuracy without the need for batch normalisations and weight initialisation. Furthermore, the algorithm provides more flexibility as hyperparameters can be updated for a specific layer in an iteration. This flexibility is utilised, as this paper extends the idea of forward-thinking to multiple source transfer whereby each layer gets a different subset of data from the original dataset.

This paper explored the range of possibilities that forward-thinking can potentially enable. In the future, this work can be extended to:

Training a DNN architecture that utilises vision transformers The vision transformer model splits the images into a series of positional embedding patches, that are processed by the transformer encoder [32]. Experiments could be conducted to explore the performance of Forward-thinking utilised in those algorithms.

Further investigation of regularisation methods that perform best on forward-thinking to reduce overfitting Forward-thinking tends to overfit with batch normalisation and weight initialisation. More experiments need to be conducted to find out the regularisation methods that perform well on forward-thinking.

Re-implementing the Resnet18 architecture in a way that utilises forward-thinking fully The current implementation of Resnet18 does not utilise the full advantage of forward-thinking. Residual networks are composed of a series of Residual blocks [8]. In each residual block, there are 2 sets of convolutional layers. Therefore, when training the Resnet18 using a forward-thinking algorithm a DNN is trained in each iteration as opposed to a shallow neural network. A DNN is still susceptible to invariant covariant shift and exploding and vanishing gradient. Nevertheless, the residual blocks reduce the exploding and vanishing gradient problem [8]. Further experiments could explore the performance of Resnet18 when each iteration the individual Residual blocks are trained instead.

References

[1] Schmidhuber, J. Deep Learning in Neural Networks: An Overview. *CoRR*. **abs/1404.7828** (2014), http://arxiv.org/abs/1404.7828

[2] Mehlig, B. Artificial Neural Networks. *CoRR*. **abs/1901.05639** (2019), http://arxiv.org/abs/1901.05639

[3] Hettinger, C., Christensen, T., Ehlert, B., Humpherys, J., Jarvis, T. & Wade, S. Forward Thinking: Building and Training Neural Networks One Layer at a Time. (arXiv,2017), https://arxiv.org/abs/1706.02480

[4] Zhuang, F., Qi, Z., Duan, K., Xi, D., Zhu, Y., Zhu, H., Xiong, H. & He, Q. A Comprehensive Survey on Transfer Learning. (arXiv,2019), https://arxiv.org/abs/1911.02685

[5] Wang, Z., Dai, Z., Póczos, B. & Carbonell, J. Characterizing and Avoiding Negative Transfer. (arXiv,2018), https://arxiv.org/abs/1811.09751

[6] He, K., Zhang, X., Ren, S. & Sun, J. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. (arXiv,2015), https://arxiv.org/abs/1502.01852

[7] Simonyan, K. & Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. (arXiv,2014), https://arxiv.org/abs/1409.1556

[8] He, K., Zhang, X., Ren, S. & Sun, J. Deep Residual Learning for Image Recognition. (arXiv,2015), https://arxiv.org/abs/1512.03385

[9] Wu, H. & Gu, X. Max-Pooling Dropout for Regularization of Convolutional Neural Networks. (arXiv,2015), https://arxiv.org/abs/1512.01400

[10] Agarap, A. Deep Learning using Rectified Linear Units (ReLU). (arXiv,2018), https://arxiv.org/abs/1803.08375

[11] Ruder, S. An overview of gradient descent optimization algorithms. (arXiv,2016), https://arxiv.org/abs/1609.04747

[12] Funahashi, K. On the approximate realization of continuous mappings by neural networks. *Neural Networks*. **2**, 183-192 (1989), https://www.sciencedirect.com/science/article/pii/0893608089900038

[13] Cybenko, G. Approximation by superpositions of a sigmoidal function. *Mathematics Of Control, Signals, And Systems*. **2**, 303-314 (1989,12), https://doi.org/10.1007/bf02551274

[14] Alcin, O., Sengur, A., Qian, J. & Ince, M. OMP-ELM: Orthogonal Matching Pursuit-Based Extreme Learning Machine for Regression. *Journal Of Intelligent Systems*. **24**, 135-143 (2015,3), https://doi.org/10.1515/jisys-2014-0095

[15] Dereventsov, A., Petrosyan, A. & Webster, C. Greedy Shallow Networks: An Approach for Constructing and Training Neural Networks. (arXiv,2019), https://arxiv.org/abs/1905.10409

[16] Aich, A. & Palanisamy, P. On application of OMP and CoSaMP algorithms for DOA estimation problem. *2017 International Conference On Communication And Signal Processing (ICCSP)*. (2017,4), https://doi.org/10.1109/iccsp.2017.8286749

[17] Marquez, E., Hare, J. & Niranjan, M. Deep Cascade Learning. *IEEE Transactions On Neural Networks And Learning Systems*. **29**, 5475-5485 (2018,11), https://doi.org/10.1109/tnnls.2018.2805098

[18] Fahlman, S. & Lebiere, C. The Cascade-Correlation Learning Architecture. *Advances In Neural Information Processing Systems*. **2** (1989), https://proceedings.neurips.cc/paper/1989/file/69adc1e107f7f7d035d7baf04342e1ca-Paper.pdf

[19] Tong, X., Xu, X., Huang, S. & Zheng, L. A Mathematical Framework for Quantifying Transferability in Multi-source Transfer Learning. *Advances In Neural Information Processing Systems*. **34** pp. 26103-26116 (2021), https://proceedings.neurips.cc/paper/2021/file/db9ad56c71619aeed9723314d1456037-Paper.pdf

[20] Huang, G., Zhu, Q. & Siew, C. Extreme learning machine: a new learning scheme of feedforward neural networks. *2004 IEEE International Joint Conference On Neural Networks (IEEE Cat. No.04CH37541)*. **2** pp. 985-990 vol.2 (2004)

[21] Ioffe, S. & Szegedy, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. (arXiv,2015), https://arxiv.org/abs/1502.03167

[22] Luo, P., Wang, X., Shao, W. & Peng, Z. Towards Understanding Regularization in Batch Normalization. (arXiv,2018), https://arxiv.org/abs/1809.00846

[23] Lian, X. & Liu, J. Revisit Batch Normalization: New Understanding and Refinement via Composition Optimization. *Proceedings Of The Twenty-Second International Conference On Artificial Intelligence And Statistics*. **89** pp. 3254-3263 (2019,4,16), https://proceedings.mlr.press/v89/lian19a.html

[24] Bengio, Y. Deep Learning. (MIT Press,2016,11)

[25] Kumar, S. On weight initialization in deep neural networks. *CoRR*. **abs/1704.08863** (2017), http://arxiv.org/abs/1704.08863

[26] Sirignano, J. & Spiliopoulos, K. Scaling Limit of Neural Networks with the Xavier Initialization and Convergence to a Global Minimum. (arXiv,2019), https://arxiv.org/abs/1907.04108

[27] Glorot, X. & Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. *Proceedings Of The Thirteenth International Conference On Artificial Intelligence And Statistics*. **9** pp. 249-256 (2010,5,13), https://proceedings.mlr.press/v9/glorot10a.html

[28] Deng, L. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*. **29**, 141-142 (2012)

[29] Krizhevsky, A. Learning multiple layers of features from tiny images. (2009)

[30] Peking SVHN Dataset. *Roboflow Universe* . (2022,5),https://universe.roboflow.com/peking-uni/svhn-x4hnw, visited on 2022-06-14

[31] Krizhevsky, A., Nair, V. & Hinton, G. CIFAR-10 (Canadian Institute for Advanced Research). (0), http://www.cs.toronto.edu/ kriz/cifar.html

[32] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J. & Houlsby, N. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. (arXiv,2020), https://arxiv.org/abs/2010.11929

[33] Goodfellow, I., Bengio, Y. & Courville, A. Deep Learning. (MIT Press,2016), http://www.deeplearningbook.org

6 Appendix

6.1 Deep neural networks

DNN aims at learning patterns from data. For Instance, a DNN with an image recognition task will have higher levels of layers capturing high-level features (abstract shapes) and low-level levels capturing low-level features (edges). A neuron is a fundamental component of a neural network. In figure 11 the neuron takes three inputs x_1, x_2, x_3. Each input is multiplied by its corresponding weights. These weights measure the importance of input towards the output neuron. Moreover, the greater the weight the more influence it has on the output. A summation is performed along with an

added bias $\sum_i, w_i x_i + b = y$. A bias is a real number that adjusts the output. The resulting number is then passed through a non-linear activation function $g(y)$. Neurons are connected together to build neural network which processes data. DNN has massively impacted vision, speech and many other areas. The DNN illustrated in figure 12 is an example of a fully-connected neural network. This idea could be scaled up to convolutional neural networks, where each neuron has a kernel (set of weights). As usual, the inputs are connected to a hidden layer of neurons. The receptive field is a defined region in the input space that connects to a particular neuron.

Neural networks are trained to recognise patterns in a training set, X by accurately adjusting the weights W and biases b between each neuron in a network, therefore it can learn to generalise out-of-sample data. Backpropagation is the traditional method to train DNN.

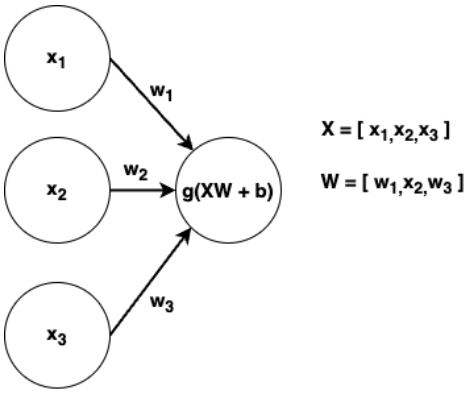


Figure 11: A example of a neuron

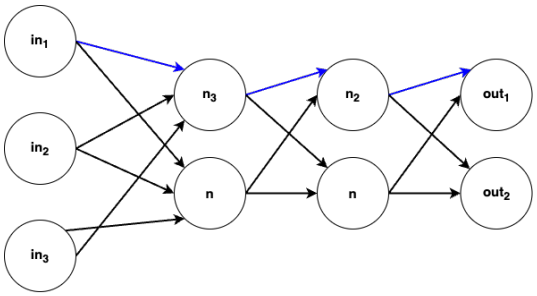


Figure 12: A simple example of a DNN

6.2 Backpropagation

The fundamental aim of backpropagation is to minimise the loss function $Loss(\hat{y}, y)$ for a neural network. To understand backpropagation, consider a single feed-through path in a network highlighted blue in figure 12. Given a training set, consisting of pairs of inputs and outputs (X, Y). The neural network $f(X)$ will provide the predictions $\hat{Y}$. The gradient for the output out_1 neuron is $\frac{\varrho(\sigma(\hat{Y},Y))}{\varrho(W_1,b_1)}$. The gradient is calculated to update (W, B) in a way that minimises the loss. Additionally, the neurons n_2, n_3 in the path are to be updated. Each neuron's parameters affect other ongoing neurons in the same path. Therefore a neuron's change in weights depends on the change in weights in subsequent neurons. As a result, the gradient of a neuron is a product of subsequent derivatives of neurons that reside later in the path. To calculate the gradient the chain rule is applied backwards to find the gradient of each neuron from the output neuron to the first neuron. For example, the gradient for the intermediate neuron n_2 is denoted as $\frac{\varrho(\sigma(\hat{n}_2,n_2))}{\varrho(W_2,b_2}$ where $\hat{n}_2$ is the output $g(W_2\text{'}X_2\text{'}+b_2\text{'})$ and n_2 is the updated output. This gradient is a product of the derivative of the previous neuron out_1. The same procedure is applied to calculate the gradient of the second intermediate neuron n_3. The

gradient $\frac{\varrho(\sigma(\hat{n}_3, n_3))}{\varrho(W_3, b_3)}$ is is a product of the previous derivatives of out_1, n_2. The general equation of the gradient is shown in equation (5).

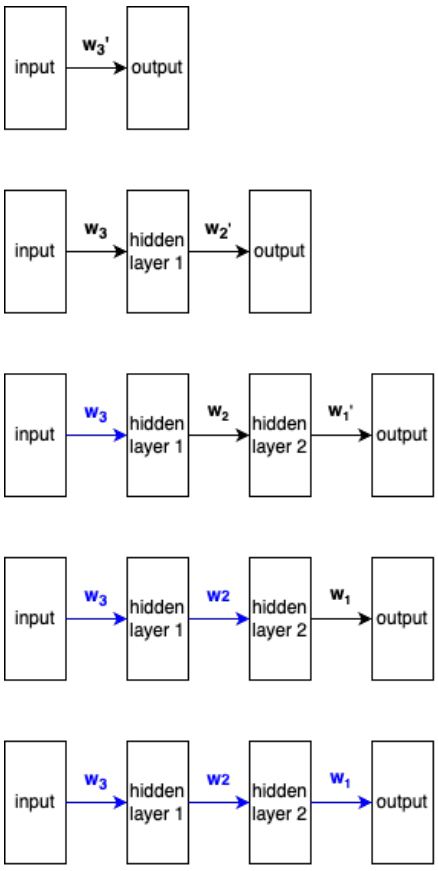
$$\frac{\varrho(\sigma(\hat{n}_n, n_n))}{\varrho(W_n, b_n)} = \sum_{i=0}^{n} \frac{\varrho(\sigma(\hat{W}_n, b_n))}{\varrho(n_n)} * \frac{\varrho(n_n)}{\varrho(W_n, b_n)}, \tag{5}$$

6.3 Transfer learning

There are two types of transfer learning. Homogeneous and Heterogeneous. Homogenous transfer learning learns from the same feature space, i.e. the training sets for both tasks overlap. Heterogeneous is when the feature spaces are different i.e the training sets for both tasks are disjoint [4]. Forward-thinking utilises the homogeneous approach.

Transfer learning seeks to improve the accuracy of a model by utilising the data from related source models. However, there are circumstances where transfer learning reduces the accuracy of a model. This is known as negative transfer learning. This often occurs due to several factors, such as the relevance between the source and target [5].

6.4 Forward-thinking algorithm



Initialisation: Set the network to have input and output layers (ℓ_{in} and ℓ_{out}) and set:

- quene_of_incoming_layers = [ℓ_3, ℓ_2]

Iteration: remove the next element from the queue and train

while queue_of_incoming_layers.isNotEmpty():

1. ℓ_n = queue_of_incoming_layers.next()
2. insert_layer_into_network(ℓ_n)
3. Train(ℓ_n)
4. Freeze_weights(ℓ_n)

Figure 13: A visualisation of the forward-thinking algorithm. The blue lines indicate that weights for a given layer have been trained and frozen. Weights denoted with $w_n\prime$ indicate they are temporary weights that are going to be disregarded and updated to w_n on the next iteration when a layer is added.

6.5 Forward-thinking algorithm on Residual networks

Residual networks are used to combat vanishing and exploding gradient problems. DNNs with many layers become more susceptible to a decrease in training and test accuracy during training. This decrease in accuracy is not caused by overfitting but by the number of layers. The more layers added the more difficult the DNN becomes to optimise.

The idea of Residual networks was developed as follows: There exists a deep model $d(x)$ that approximates the function of a shallow model $s(x)$ to at least the same accuracy, by copying the layers from $s(x)$ to $d(x)$, while the remaining layers of $d(x)$ learn the identity function.

The remaining layers can learn the identity function. Since the identity function is difficult to learn, skip connections are built into the architecture. The input, x (the identity function) of a hidden layer $s(x)$ goes into the input of the subsequent layer $d(x)$ in addition to the output of $s(x)$. Therefore, the subsequent layer takes in $d(s(x) + x)$. The hypothesis is that the deeper the layers go, each layer learns the identify function therefore the accuracy of a given layer will be at least the same accuracy of the subsequent layer. This is different from traditional layers in that a layer does not receive original input given to a prior layer. This architecture is applied to SVHN and CIFAR10 datasets. Figure 14, illustrates a visualisation of the behaviour of the forward-thinking algorithm applied to a residual network.

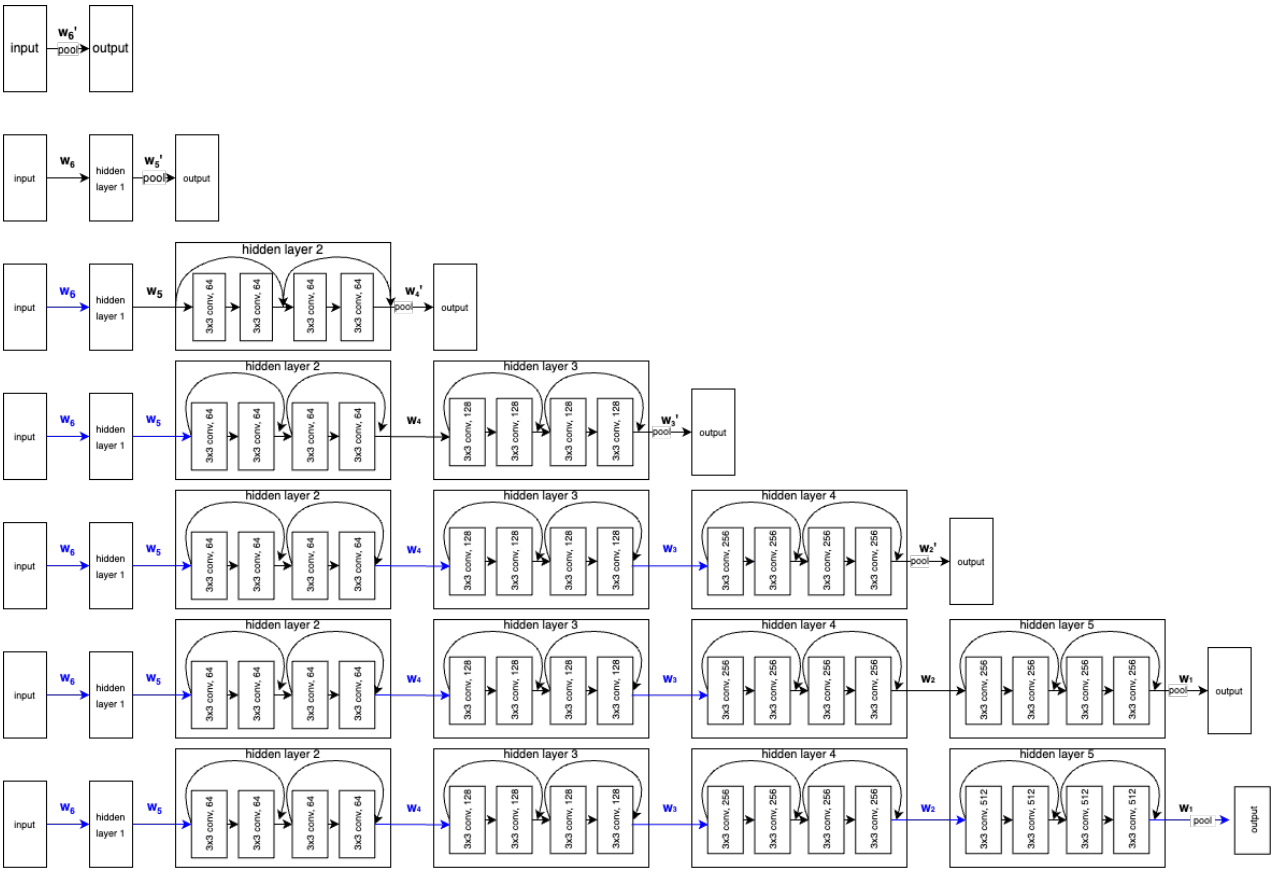


Figure 14: A visualisation of the forward-thinking algorithm when training a Residual network (Resnet18). This is different from forward-thinking training of a simple neural network, in that a shallow network is not trained within each iteration, instead, it is training a DNN composed of residual blocks.

6.6 Batch normalisation

Batch normalisation was developed to solve the "internal covariate shift" problem that occurs when performing backpropagation. In backpropagation, the weights are changed after each mini-batch. Consequently, this changes the distribution of inputs to subsequent layers. These shifts in input distributions mean it is difficult to reach an approximation of a function, as the targets for each layer are constantly changing. This is known as "internal covariate shift" [21]. Batch normalisation is a technique proposed to normalise data for each mini-batch This allows the distribution of inputs to remain the same.

The idea of batch normalisation was created by looking at each layer in a neural network as an independent machine learning problem with inputs and outputs. In machine learning problems, the inputs are often normalised. Normalisation is a pre-processing technique that transforms the sample data so that the mean is 0 and the variance is 1. This helps speed up training and leads to a faster convergence [21] [23]. With this in perspective, this means that each layer in a neural network should normalise the data.

$$\begin{aligned} \mu &= \frac{1}{m}\sum_{i=1}^{m}\mathbf{x}^{(i)} && \text{1. mean} \\ \sigma^2 &= \frac{1}{m}\sum_{i=1}^{m}(\mathbf{x}^{(i)\smile}\mu)^2 && \text{2. variance} \\ \mathbf{x}_{new}^{(i)} &= \frac{\mathbf{x}^{(i)\smile}\mu}{\sqrt{\sigma^2+\epsilon}} && \text{3. normalise} \\ \mathbf{z}^{(i)} &= \gamma\mathbf{x}_{new}^{(i)} + \beta && \text{4. scale and shift} \end{aligned} \tag{6}$$

6.6.1 How batch normalisation works

The most notable factor about batch normalisation that differentiates itself from other proposed methods is the normalisation operation applied in a mini-batch during training. Batch normalisation is built around backpropagation. In that, the data is constantly shifted during training for every mini-batch. Therefore normalisation is applied in a mini-batch for a particular layer.

In training, the output of a particular layer is normalised within a mini-batch. Therefore the next layer takes in normalised input. Equation 6 shows the steps for batch normalisation. The normalisation formula adds a constant value ε. To prevent division by small values. In addition, two new parameters γ and β are learned during training. γ is a scalar applied to x^i, and β is a shift transformation. This allows the inputs to transforms into the best distribution that the neural network can learn from.

6.7 Overfitting

Forward-thinking is neither less nor more susceptible to overfitting than backpropagation. Overfitting occurs when a neural network fails to capture an out-of-sample dataset containing the same features as the training set. This is because the weights and biases are adjusted well-fitted into fine details (noise in the training set). Therefore the neural network cannot generalise well to other datasets. This often occurs in neural networks when there are enough neurons to capture all the unique patterns for that given dataset.

Although in forward-thinking each step involves only training a shallow network, the layers combine to create a single DNN [21]. The knowledge from a previously trained layer is transferred to subsequent layers. A layer that overfits will propagate the negative knowledge over to other layers. Since all the layers use the same dataset and will further continue to capture the unique properties of the training set. Therefore, during forward-thinking more regularisation techniques should be enforced.

6.8 L_2 regularisation

There are many techniques to reduce overfitting. One common technique is weight decay or the L_2 regularisation. This is a regularisation technique that is applied to the weights of a neural network.

L_2 regularisation works by adding a penalty λ to the cost function. Therefore, the optimisation goal is to minimise the loss function and the additional λ penalty on the sum square of the weights. This is known as structural risk minimisation, $Loss_r$ [24] denoted in equation (7).

$$Loss_r = Loss(\hat{y}, y) + \lambda(w_1^2 \dots w_n^2) \tag{7}$$

6.9 Weight initialisation

Weight initialization is crucial when developing neural networks to ensure the model converges to high accuracy. Weight initialization is a procedure to declare the initial point of weights for a neural network before training [24]. There are many weight initialization strategies such as the Xavier [25], normalised Xavier [26] and Kaiming Initialization [6]. Initialising the weights is critical when training DNN. This determines whether the algorithm converges at all.

The type of weight initialization technique used must take into various factors of the neural network model, particularly the activation function used. For example, the normalised Xavier and Xavier initialization have become standard for DNNs that use the tan or sigmoid activation function while the Kaiming Initialization is suited to DNN architectures that use the ReLU activation function [27].

6.10 forward-thinking vs Backpropagation

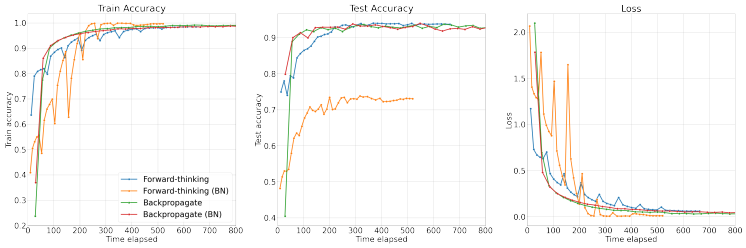


Figure 15: Compares the performance of backpropagation and forward-thinking by using the training, test accuracy and the loss. The learning algorithms were applied to the VGG11 DNN architecture, to solve the SVHN image recognition task.

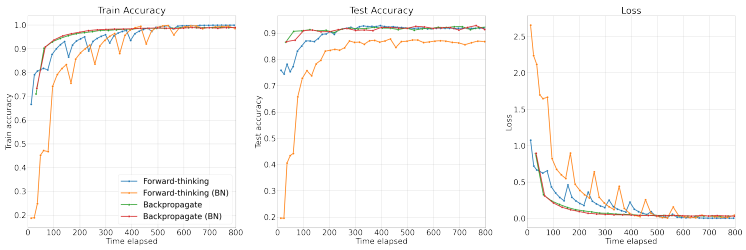


Figure 16: Compares the performance of backpropagation and forward-thinking by using the training, test accuracy and the loss. The learning algorithms were applied to the Resnet18 DNN architecture, to solve the SVHN image recognition task.

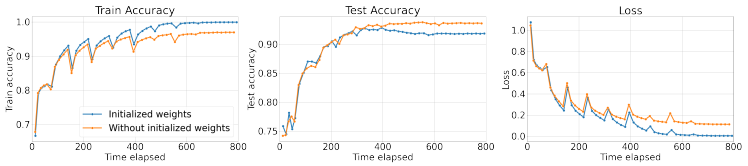


Figure 17: A comparison on forward-thinking with and without weight initialization on the Resnet18 architecture solving the SVHN image recognition problem.

Figure 18: A comparison on forward-thinking with and without weight initialization on the VGG11 architecture solving the Cifar10 image recognition problem.

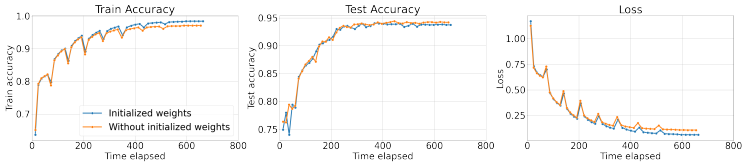


Figure 19: A comparison on forward-thinking with and without weight initialization on the VGG11 architecture solving the SVHN image recognition problem.

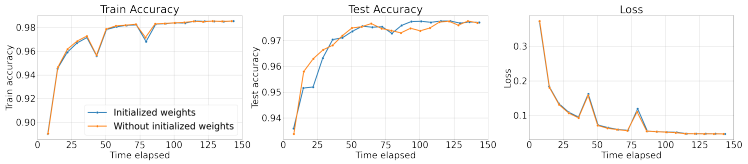


Figure 20: A comparison on forward-thinking with and without weight initialization on the fully-connected DNN architecture solving the MNIST image recognition problem.

6.11 forward-thinking on weight initialization

6.12 Training times for each learning alogrthium

Training time (milliseconds)		
DNN Architecture	Backpropagation	Forward-thinking
VGG11	1069.1	395.3
VGG11 with batch normalisation	1135.1	498.9
Resnets18	1246.4	504.0
Resnet18 with batch normalisation	1349.1	685.6

Table 3: The time in milliseconds it took to train a given DNN architecture using Backpropagation and forward-thinking on the CIFAR10 image recognition task

Training time (milliseconds)		
DNN Architecture	Backpropagation	Forward-thinking
VGG11	1111.1	662.8
VGG11 with batch normalisation	1152.1	520.0
Resnets18	793.0	793.0
Resnet18 with batch normalisation	1325.0	1022.5

Table 4: The time in milliseconds it took to train a given DNN architecture using Backpropagation and forward-thinking on the SVHN image recognition task

Training time (milliseconds)		
DNN Architecture	Backpropagation	Forward-thinking
Feedforward	294.4	143.5

Table 5: The time in milliseconds it took to train a given feed-forward neural network using Backpropagation and forward-thinking on the MNIST image recognition task

6.13 Datasets used in experiments

The datasets used in experiments are described below:

MNIST: The MNIST dataset (Modified National Institute of Standards and Technology database) is a collection of 28*28 grey-scale pixels images of handwritten digits [28]. The training set has 60,000 examples and the test set has 10,000 examples.

SVHN: The SVHN dataset (The Street View House Numbers) [30] is a similar dataset to MNIST. However, it incorporates order of magnitude on the label data and consists of over 600,000 digital images. There are 26,032 images contained in the test set and 73,257 images contained in the training set. The additional 531,131 images can be further used for training.

CIFAR10: The CIFAR10 dataset (Canadian Institute For Advanced Research) has 10 classes/targets including aeroplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. There are 50,000 training images and 10,000 test images. There are 5000 training images and 1000 test images for each class [31].

6.14 Reproducibility

The code used to implement the forward-thinking algorithm can be found on `https://github.com/runnily/forward-thinking`