Learnable Weight Initialization for Deep Neural Networks

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Arkajit Bhattacharya
born in Tripura, India
Student id: 4787315
Email: arkajitb@gmail.com

Thesis Committee:

Chair:	Prof. Dr. Jan Van Gemert, TU Delft (supervisor)
Committee Member:	Dr. David Tax, TU Delft
Committee Member:	Dr. Matthijs Spaan, TU Delft

Deep Learning
Department of Computer Vision
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands

Contents

Chapter 1

Introduction

1.1 Research and Problem

In recent years, deep learning has shown pronounced results in the field of image recognition, computer vision, and speech recognition tasks. These were achieved regardless of the issues faced while training deep neural networks. One of the most common issues is regarding the initialization of weights prior to the training of the network. Proper weight initialization prevents the layer outputs and the gradients from exploding or vanishing during the learning process. There has been a lot of research in recent years about the same. the authors of [2] introduced a proper weight initialization technique for the first time where they show that it is possible to prevent the activation output from exploding/vanishing by initializing the weights from a uniform distribution and then multiplying(scaling) it based on the number of incoming nodes for a single node. This method has proven to be very effective for certain criteria and activation functions which are symmetric around zero and outputs value within the range [-1, 1]. This observation was confirmed in [3], where the authors mention Xavier weight initialization's inefficiency when used with ReLU activation function since ReLU outputs all negative values to zero which leads to the dying ReLU problem. Thus, the authors introduce a new weight initialization technique, popularly known as Kaiming weight initialization which is advisable to be used with ReLU activation function. That said, the issue of finding an independent weight initialization technique still prevailed. As mentioned before, there are different types of activation functions based on how they distribute the layer output to achieve non-linearity. The input to the activation function is the weights multiplied by the data flowing through the network with an optional bias. Since it is a multiplication operation, if the weights are not initialized properly, depending on the activation function, the output might not be scaled accordingly. Certain weights might keep on growing or scale down at an exponential rate during the training process which might lead to exploding/vanishing gradient during the back-propagation of the loss gradient. These findings lead to the introduction of independent weight initialization techniques in recent years. The aim is to find a generalized way to initialize the weights which are not dependent on the activation function to be used in the network.

In [6], the authors conclude that initializing weights with orthonormal values can be beneficial to the performance of the network, thus, establishing the fact that optimal weight initialization can be done regardless of the activation function used. This was followed by another technique, known as Layer Sequential Unit Variance Initialization(LSUV)[8], which learns the optimal weights by training each layer with the dataset provided to have activation output of one with a tolerance value. LSUV could be considered as batch normalization before the training process starts. Considering the fact that it is less computationally expensive when compared to batch-normalization, it could be used as a replacement for the same. The experiments has shown that LSUV can perform similar to full batch normalization. Regardless of it's achievements, it couldn't be considered as a genelarized solution. It achieved SOTA in few scenarios for image classification but failed to even train the model

for sigmoid activation function. The idea of pre-training the model with data is taken from LSUV. The algorithm used for the proposed method tries to control the mean of the layer output and variance of the activation output within a pre-defined range. This method can be used to find the optimal weights for a network regardless of the activation function used. Moreover, it is believed to replace batch-normalization from the entire network.

1.2 Research Scope

1. **Is it possible to initialize weights of a network with optimal values independent of the activation function used?**
   Hypothesis: If each layer in the model is trained to have a pre-defined mean of 0 for the layer output and pre-defined variance of 1 for the activation output, the trained weights will be able to perform equally when compared with other weight initializers which are dependent on the activation function used, thus, providing a generalized approach towards the problem of optimal weight initialization.

2. **Is it possible to remove batchnorm from a deep neural network?**
   Hypothesis: The proposed method could be seen as a batch normalization process for the activation output before the training starts. Thus, it can remove batch-normalization from a network.

Chapter 2

Related Work

Our work is concerned with the initialization of weights in deep neural networks and it's relation to activation functions. To understand further, it is important to explain what neural networks are, how do they work and why is it important to initialize the weights of the neural networks properly.

2.1 Deep neural networks

Deep Neural networks, popularly known as deep learning is a subset in the field of artificial intelligence which aims at teaching the computer complex relationship between objects by feeding them simpler ones. A neural network is a combination of layers, each layer consisting of multiple nodes or neurons. Each node is initialized with weights and biases(optional). Data is processed through each layer and gets multiplied by the weights in each node. The last layer in a neural network is a decision layer that aims at classifying each data processed accurately. The difference between the actual class and the predicted value is the loss which is based on the loss function used(L1 loss, L2 loss etc). The gradient of the loss with respect to each weight is back-propagated and each weight is updated based on the given magnitude of change(learning rate). The figure 2.1 provides a visual representation of a simple neural network with one input layer, two hidden layers and one decision layer.

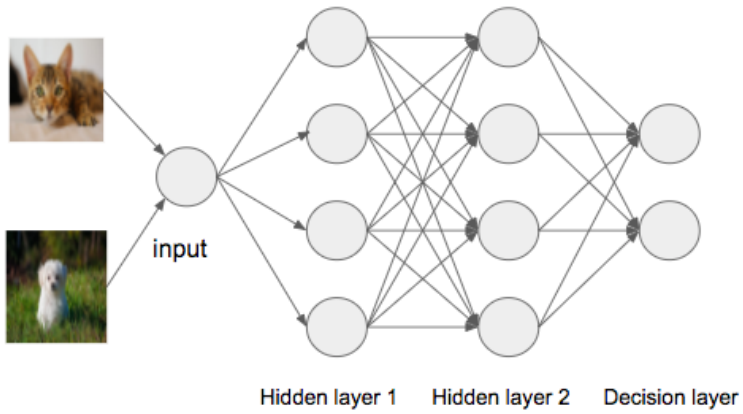

Figure 2.1: A neural network with one input layer, two hidden layers and one classification layer. The data contains images of cats and dogs. The task in hand is to classify the images into the mentioned categories

The diagram above gives a high level understanding of the flow of data through the network.

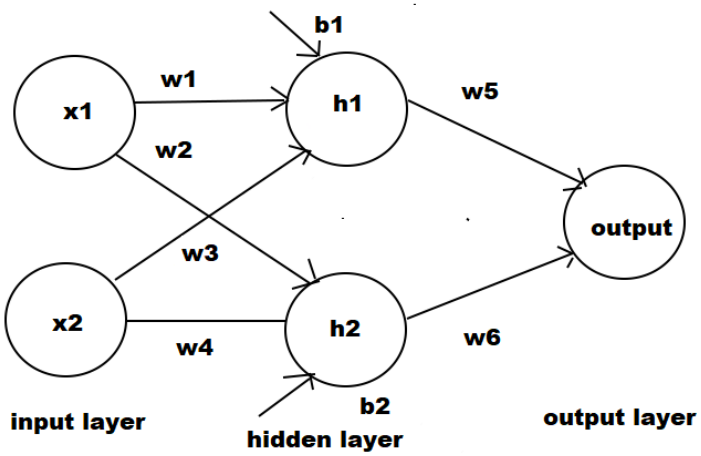

Figure 2.2: A neural network with one input layer, one hidden layer and one output layer. Each connection between the nodes contains a weight multiplied and a bias added to it.

To understand how the gradient flows, a more detailed visual representation with a lesser number of nodes is provided in Fig 2.2, which contains one input layer, which receives the data for forward propagation, one hidden layer, and one output layer which is the decision layer in this scenario.

2.2 Related Concepts

There are a number of concepts that are required to understand how a neural network works. The important functionalities which are required to understand the process of learning are described in the sections below.

2.2.1 Back-propagation

Back-propagation is the method of updating the weights and biases through a backward pass for each forward pass in the network. The aim of back-propagation is to reduce the loss for the network by updating the trainable parameters. The gradient describes the rate of change of a parameter(weight) based on the loss calculated. The gradient calculation is achieved by chain rule. Chain rule can be described by the equation mentioned below:
$f(g(x))' = f'(g(x))g'(x)$

Forward propagation can be seen as a series of equations nested with each other. This property helps to calculate composite functions in a simpler fashion. It finds the impact of each variable separately, making it easier for calculating the gradients required for the back-propagation update.

2.2.2 SGD : Stochastic Gradient Descent

Gradient is the slope of a surface. Thus, Gradient descent describes the process of descending to reach the lowest point in the slope. Stochastic means random. In this context, stochastic gradient descent takes a random point in the slope and tries to reach the lowest point. The slope represents the loss function, thus, SGD targets to reach the point where the loss is lowest.

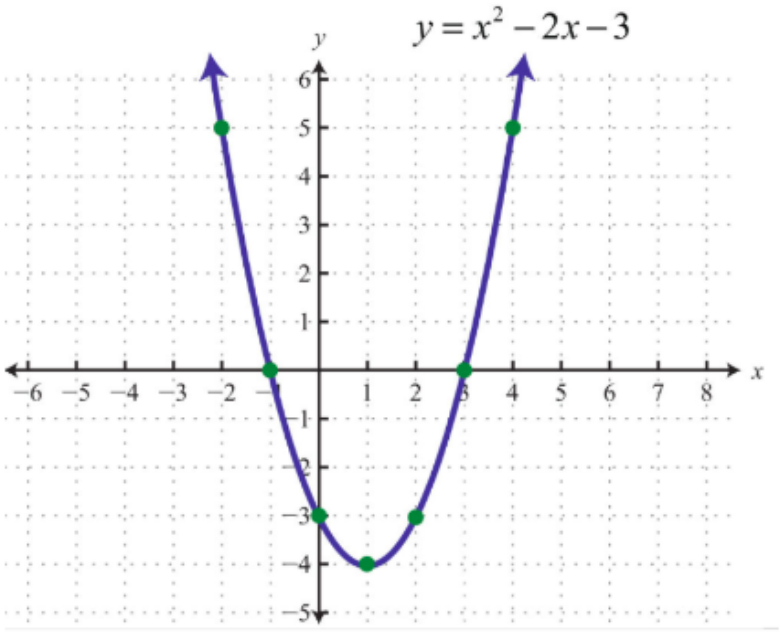

Figure 2.3: A 2 dimentional parabolic function where the lowest point is at x = 1. Loss function targets to find the lowest point of x where y is minimum[11]

2.2.3 Covariance shift

Covariance shift means that the distribution of training and test data is different. For instance, if a model is trained on black and white images and tested on colored images, the distribution of the data changes, and the model won't be able to perform as expected. This phenomenon is important to understand how to batch normalization affects the model in these situations.

2.2.4 BN : Batch Normalization

Batch normalization is the process of scaling the activations to reduce covariance shift. Moreover, it helps each layer to learn features independently of the other layers. It is possible to have a higher learning rate if batch-normalization is introduced since it scales the activations during the training process. Batch normalization normalizes the output of the activation layer by subtracting the batch mean from the output and dividing it by the batch standard deviation.

The step-by-step equations are provided below :

1. Calculate mean for the batch : $\mu = 1/n(\sum_{i=1}^{m} x_i)$, where m is the number of samples in the mini-batch.
2. Calculate variance of the batch : $\sigma^2 = 1/m\sum_{i=1}^{m}(x_i - \mu)^2$
3. normalize each data point: $x_i = x_i - \mu/\sqrt{\sigma^2 + \varepsilon}$
4. scale the output: $y_i = x_i + \beta$

Advantages

1. Learning rate can be increased during the training which will lead to faster convergence of the loss.
2. In some cases, accuracy improves with the introduction of batch-norm.
3. Dropout can be avoided.

Disadvantages

1. Batch size cannot be 1.
2. During testing, it becomes difficult to calculate mean and variance.
3. Batch-Norm is computationally expensive.

2.2.5 Loss functions

Loss function can be described as the function that needs to be minimized to achieve the goal. There are different types of loss functions based on the problem in hand. The choice of the loss function is dependent on the output layer activation function. The types of problems associated with neural networks and the preferred loss functions for the same are described below:
1. Regression problem : Regression problem can be described as a problem where the aim is to find a real-valued number. The output layer contains only one node with a linear activation. The loss function adequate for this situation is the **Mean Squared Error(MSE)** loss.
2. Binary Classification problem : A problem where the input needs to be classified between two target outputs. The output layer contains one node for each class. The activation function used is softmax, which basically gives a probability of the input belonging to a particular class. The loss function used in this situation is **Cross-Entropy/Logarithmic loss**.
3. Multi-Class Classification: The input needs to be classified amongst multiple classes. Similar to binary class classification, **Cross-entropy/Logarithmic loss** is used.

For this research, learning the mean of the layer output and variance of the activation output, MSE loss is used, since it is a regression problem. The equation for MSE loss is provided below:

- Calculate the square error for each predicted value: $e = (actual - predicted)^2$
- sum all the square errors and divide by the total number of values to get the mean squared error: MSE = $1/n\sum_{i=1}^{n} e_i$

2.2.6 Activation Functions

Activation functions are important for neural networks since it introduces non-linearity to the network. Linear functions will have a constant derivative during gradient descent which is not advisable for a deep neural network. It basically decides whether a neuron will be fired or not, to be more specific, whether the neuron will take part in the learning process. There are various types of activation functions. The most popular once are as follows:

Sigmoid Activation function Sigmoid activation function gives an output within the range of [0,1]. The formulated version of sigmoid is:

$$A = 1/(1+e^{-x})$$

where x is the output of the neuron and A is the output of the activation function. Fig 2.4 gives a graphical representation of the function.

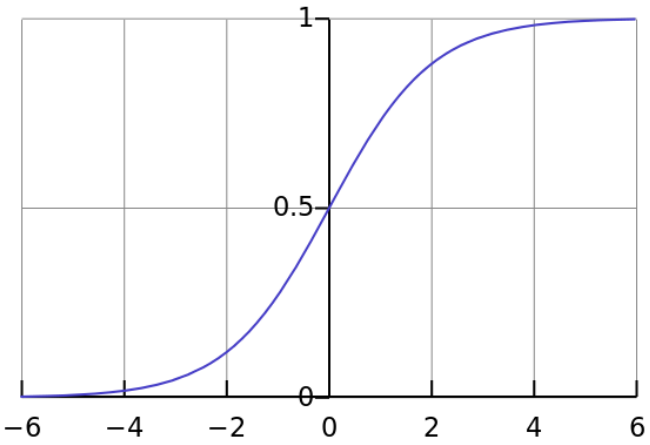

Figure 2.4: Sigmoid activation function with x-axis represents the input and y-axis represents the output of the activation function[12]

One of the important advantages of this function is that the output doesn't blow since it gives an output between 0 and 1. That said, it faces the problem of vanishing gradient.

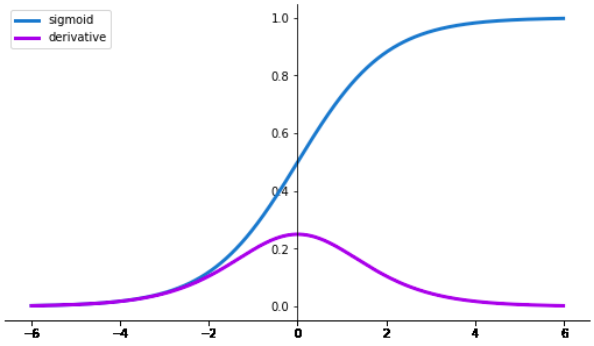

Figure 2.5: sigmoid and it's derivative[10]

As shown in Fig 2.5, it is clearly visible that the derivative for any value for sigmoid is low which eventually might vanish if the network is deep.

Tanh Activation Function Tanh is an upgraded version of the sigmoid activation function which gives an output symmetric around zero and within a range of [-1,1]. The formulated version of tanh is as follows:

$$A = 2/(1+e^{-2x}) - 1$$

Fig 2.6 gives a graphical representation of the activation output and it's derivative for Tanh activation function. It is better than sigmoid since it is symmetric around zero but it also faces the issue of gradient saturation.

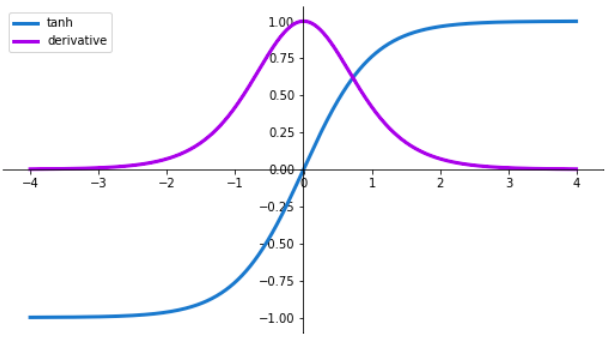

Figure 2.6: Tanh and it's derivative[10]

ReLU Activation Function ReLU stands for Rectified Linear Unit. The equation of ReLU is:

$$A = max(0, x)$$

where A is the activation output and X is the output of the neuron. ReLU solved the issue of gradient saturation it doesn't scale down the neuron outputs if it is positive. Since it scales down all the negative values to zero, it adds sparsity to the network. In addition, it is computationally less expensive when compared with Sigmoid and Tanh.

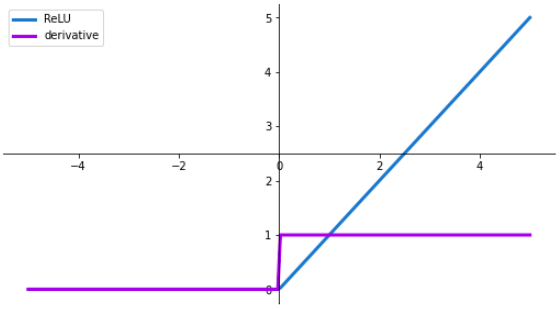

Figure 2.7: ReLU and it's derivative[10]

As shown in Fig 2.7, the positive side doesn't have any bounds. The disadvantage of using ReLU is that it always discards values lower than zero which kills any neuron associated with the same. It leads to the dying ReLU problem because of which the network might lose important information since these neurons won't respond anymore throughout the learning process.

Leaky ReLU: It is a version of ReLU. The equation is as follows:

$$A = max(ax, x)$$

where x is the neuron output, a is a constant and A is the activation output. It is used as a solution for the dying ReLU problem. As shown in Fig 2.8 Instead of killing all the neurons with a negative value, it scales it down to a value proportional to the input since the same value gets multiplied to the input for the outer bound. It can be seen as a better version but it doesn't guarantee better performance then ReLU and it can only be used as an alternative for ReLU.

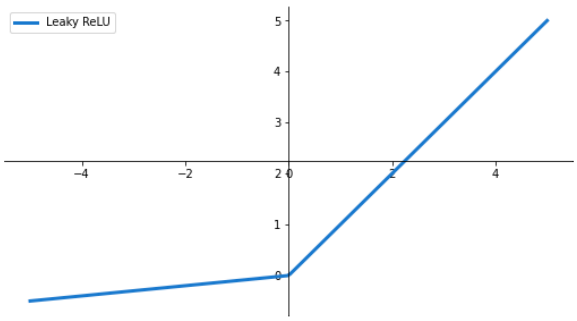

Figure 2.8: Leaky ReLU and it's derivative[10]

One of the disadvantages is the use of a constant 'a' which is dependent on the user,i.e., it is a hyperparameter and the value is different for different scenarios.

Parametric ReLU: It is an updated version of Leaky ReLU where the parameter 'a' is not decided by the user, instead it is a learnable parameter. It has the same equation as of leaky ReLU. Fig 2.9 gives a visual representation of the activation output and it's derivative for parametric ReLU. Similar to Leaky ReLU, this activation function can alos be seen as just an alternative for ReLU since it doesn't guarantee better performance than ReLU.

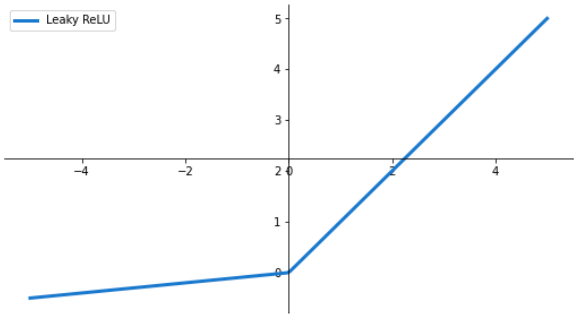

Figure 2.9: Parametric ReLU and it's derivative[10]

Recent years have seen the introduction of new activation functions like **ELU**, **Swish**,**Mish** etc. but it is out of the scope for this research.

2.3 Training a neural network

Training a neural network undergoes the following steps:

- The data is processed through the network one by one or in batches.
- Each neuron has a weight attached to it and an additional bias which is optional.
- Data gets multiplied to it and then it passes through the activation function which introduces non-linearity to the network.
- The process continues till the last layer which is the decision layer with an output equal to the number of classes present in it.

- Loss is calculated based on the output of the decision layer and the loss function. Loss gradient is calculated for each weight.
- Each weight is updated using gradient descent.
- The process continues till the loss converges to it's lowest value.

Once a basic idea of the flow of data is provided, it is important to understand what are activation functions and how does it affects the learning process. As mentioned in [10], the idea behind an activation function is to activate the neurons of the layers. It introduces non-linearity in the network. If activation functions are not used, the neural network will merely be a linear regression model which is not sufficient to learn complex features from the data. The activation function decides which neuron will fire by how much magnitude. It is placed between two layers. The data passes through the activation function once it gets multiplied with the weight of a neuron. Figure 2.10 provides a block diagram of the flow of data between two layers.

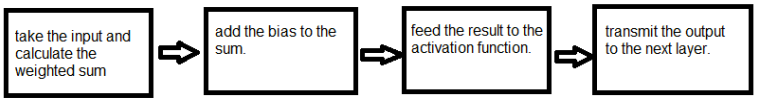

Figure 2.10: A block representation of the flow of data through the network.

2.4 Vanishing/Exploding Gradients

Vanishing gradient problem: It occurs when the weights scale down to zero. If the weights at the initial stage are small, the gradients go smaller with each backpropagation leading to the problem of vanishing gradient.

Exploding gradient problem: It occurs when the weights become NaN, i.e., infinity. If the weights are large, the gradient gets higher with every iteration and the neurons might explode. Thus, weight initialization techniques were introduced according to the activation functions used. Earlier years have seen random initialization of weights from a uniform distribution as a better option but it was never the optimal solution to the problem. The following sections will describe the weight initialization techniques used currently in the field of deep learning in order of their introduction to the field. **Signs of exploding/vanishing gradients**: The network gradients are exploding/vanishing if one or more of the signs mentioned below are seen in a network:

- The weights associated with the layers scale up faster than expected.
- The model weights become NaN during the training.
- The error is always above 1 during the whole training phase.

Fixing the problem There can be multiple approaches to this issue. The best approaches are:

- **Re-designing the network** : Vanishing/exploding gradient problem mostly occurs due to a large number of layers in a network. Thus, re-designing the network to have fewer layers could be a possible solution.
- **Reducing batch-size**: Reducing the batch-size could be a possible solution.

- **Use gradient-clipping** : Limiting the gradient size could tackle the issue of vanishing/exploding gradients.
- **Weight regularization** : It is possible to check the weights of the network for each iteration and punish larger weights accordingly, which could be done based on the loss function used for the network.

2.5 Weight initialization

The methods mentioned in 2.4 are useful if the network is already facing Vanishing/exploding gradient problem. The ideal situation would be not to experience the same in the first place, since, it is computationally expensive and time-consuming to train a deep network for a large dataset. The concept of weight initialization cam into the picture for the same reason. The neural network learns through gradient descent method where activation function plays an important role in introducing non-linearity to the network. Thus, randomly initializing the weights should work since the network will be able to learn gradually by multiple iterations. In reality, the performance of the network depends on the initialization of the learnable parameters. In general, learnable weights are initialized with random values as it is supposed to reach it's optimal value through the learning process. It cannot be initialized with the same number as the activation output will have the same value for all the neurons, thus, the model won't learn anything. Similarly, if the weights are randomly initialized, it might face the issue of **vanishing and exploding gradients** during gradient descent.

2.5.1 Xavier Weight Initialization

The goal of Xavier weight initialization[2] is to keep the variance for each layer during the training process. The weights are randomly initialized from a random uniform distribution with mean zero and multiplied by $1/N_{Avg}$ where N_{Avg} is the average of input and output neurons. This initialization may result in dependence on the backpropagated gradient variance on the layer, and it might decrease throughout the training process. Thus, to add a normalization factor, the final weight initialization formula is:

$$W = U[\sqrt{6}/\sqrt{N_{in}+N_{out}+1}, -\sqrt{6}/\sqrt{N_{in}+N_{out}+1}]$$

where N_{in} is the number of incoming nodes and N_{out} is the number of outgoing nodes. This method is suitable for activation functions which are symmetric around zero and gives an output within the range of $[-1,1]$.

2.5.2 Kaiming weight initialization

Kaiming weight initialization[3] was introduced since Xavier[2] cannot be used with ReLU activation function. Xavier tries to initialize the weights in such a way that the mean of the activation output should be near to zero. While using ReLU activation function, this method won't work since it scales down all values lower than or equal to zero to zero which results in might kill most of the neurons and the learning process will get slower. For the network with ReLU as the activation function to learn, it is very important that the mean of the layer output is zero and it should gradually increase with each iteration and the variance of the activation function is approximately one. This will prevent the gradients from exploding and vanishing throughout the learning process.

Below are the steps involved in the Kaiming initialization process:

- Create a tensor with the same size as of the input and initialize the values from a random uniform distribution.
- Scale all the values with $\sqrt{2}/N$ where N is the number of incoming nodes.

Kaiming weight initializer has shown great promise when used with ReLU activation function but it is specific to only ReLU and is not recommended to be used with other activation functions.

2.5.3 Layer Sequential Unit Variance weight initialization

Layer Sequential Unit Variance Initialization(LSUV)[8] was introduced to create a weight initialization technique that doesn't depend on the activation function used for the network. For instance, Kaiming weight initializer was introduced only for ReLU activation function since it is not advisable to use Xavier weight initializer with Sigmoid or Tanh activation functions. The authors mention another important point which is often neglected, i.e., the previous weight initialization techniques donât consider all types of layers, like max-pooling or normalization layers which also affects the activation variance. this method follows a data-driven approach to learn the weights rather than pre-defining it which was not done before. It follows the algorithm stated below:

- Initialize the weights from a gaussian distribution with a variance of 1.
- Decompose the weight matrix with QR matrix decomposition method or Singular Value decomposition(SVD) method.
- Train the convolution and inner product layers to have an output variance of one with a small batch of samples from the dataset.

Algorithm 1 Layer-sequential unit-variance orthogonal initialization. L – convolution or full-connected layer, W_L - its weights, B_L - its output blob., Tol_{var} - variance tolerance, T_i – current trial, T_{max} – max number of trials.

Pre-initialize network with orthonormal matrices as in Saxe et al. (2014)
for each layer L **do**
  while $|Var(B_L) - 1.0| \geq Tol_{var}$ and $(T_i < T_{max})$ **do**
    do Forward pass with a mini-batch
    calculate $Var(B_L)$
    $W_L = W_L / \sqrt{Var(B_L)}$
  end while
end for

Figure 2.11: LSUV algorithm presented in the paper[8]

Fig 2.11 shows the algorithm provided in the original paper[8].

LSUV was able to achieve State-of-the-art(SOTA) results for many combinations of activation functions, models, and datasets. To be more specific, it achieved SOTA when for FitNets with CIFAR-10 and MNIST datasets.

LSUV could be seen as batch-normalization before the start of the training process. Thus, it is computationally more effective than batch-normalization since it is done only once using a mini-batch before the training. This built the foundation for the proposed method, i.e., it is possible to teach the model to scale the weights accordingly by using samples from the same dataset that will be used for training the model.

2.6 Weight Initialization vs Batch Normalization

Batch normalization and weight initialization both plays an important part in the training of a neural network. The addition of batch normalization is mainly useful for faster training and to avoid the exploding/vanishing gradient problem. Proper weight initialization can also ensure that the gradients donât vanish or explode during the training process. Moreover, it is possible to train the model with a batch-size of 1 if batch normalization is not used in the network. Batch normalization is computationally expensive when compared to weight initialization since the later only needs to be initialized once for the whole learning cycle. The only issue associated to weight initialization is the fact that it is difficult to find the optimal weights for all the layers which can make sure that the network's performance is at par with that of batch-normalization. Another factor that contributes to this issue is that the most famous weight initialization techniques used are dependent on the activation function used. This research aims at finding a weight initialization technique independent of the activation function used and could remove batch normalization from a network.

Chapter 3

Datasets and Preparation

3.1 Datasets

3.1.1 FashionMNIST

It is a dataset of grayscale images[13] which consists of images related to the world of fashion. It was introduced in zalando's article[13] as an upgraded version of the MNIST dataset which is a collection of handwritten grayscale images. It consists of 60,000 training images and 10,000 images for test. Each image is 28x28 in size with a total number of 10 classes.

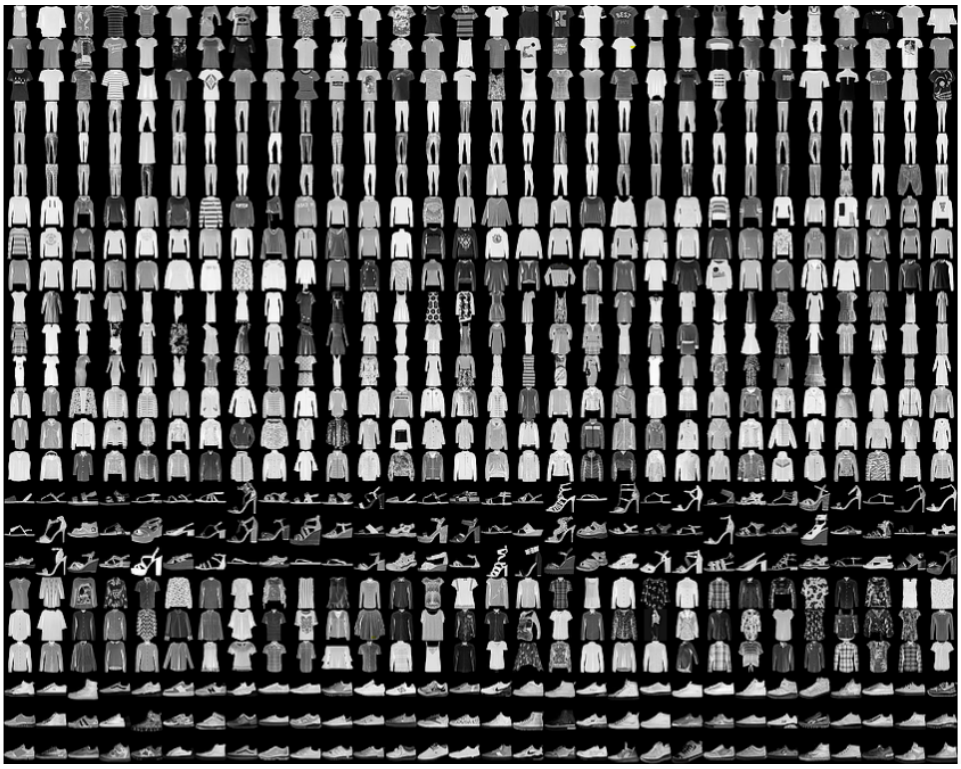

Figure 3.1: FashionMNIST dataset[13]

Fig 3.1 gives a visual representation It was created since MNIST is not adequate for the Computer Vision tasks and lacks complexity. For this research it was used as a sanity check for the proposed method in combination with LeNet architecture which is a network with only two convolution layers, thus, making it easier to understand the flow of data.

3.1.2 Imagenette

Imagenette dataset[5] is a subset of the original Imagenet dataset with only 10 classes out of the original version, with a purpose of creating a dataset which can be used for new ideas and algorithms since Imagenet takes longer because of it's size. It comes in three different sizes : Full Size, 320 px and 160 px. For this research, the full size is used and reduced according to the requirement. The data manipulation part has been explained in the next section.

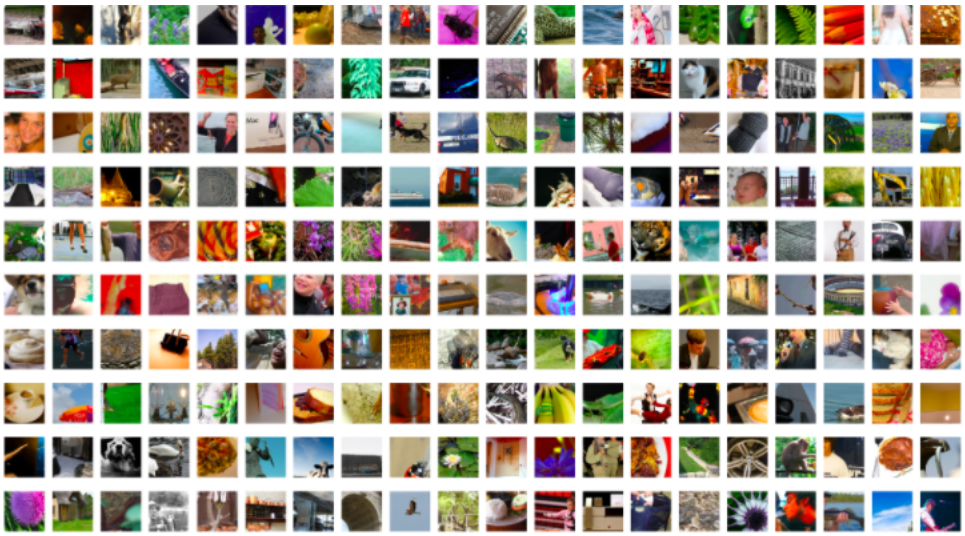

Figure 3.2: Imagenette dataset[9]

Fig 3.2 gives a visual representation of the imagenette dataset.

3.2 Preparation

3.2.1 Fashion-MNIST

For preparing the dataset, below mentioned steps were followed :

- Apply the suggested mean and variance transformations for an image with a single channel.
- Shuffle the images in the training dataset.
- Remove labels from the images.
- Create a batch of 32 for training the model to learn the optimal weights. It is recommended to use the same batch size while training the model with the training dataset.
- Randomize the batches and create a mini-batch which will be used to learn the weights.

3.2.2 Imagenette

For preparing the dataset, below mentioned steps were followed :

- Resize images to 224x224 pixels.
- Apply the suggested mean and variance transformations for an image with three channels.
- Shuffle the images in the training dataset.
- Create a batch of 32 for training the model to learn the optimal weights. It is recommended to use the same batch size while training the model with the training dataset.
- Randomize the batches and create a mini-batch which will be used to learn the weights.

The size of the mini-batch can be changed accordingly. It is suggested to take thirty percent of the images from the dataset to keep it more generalized since the same dataset will be used to train the model as well.

Chapter 4

Network Architectures

The neural networks used for this research and the justification for selection of the same is provided in this chapter.

4.1 LeNet

LeNet[7] is a simple architecture to work with. It contains a set of two convolution layers, activation layers and maxpool layers, followed by three linear layers. The aim of this network was to implement text character recognition. To understand how theoretically correct the proposed method is, the selection of a simple network for sanity check is the logical approach.

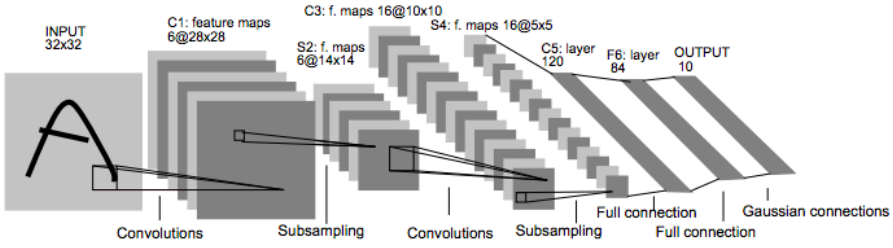

Figure 4.1: LeNet architecture[7]

4.2 ResNet

Introduction of Residual Networks(ResNets)[4] solved the problem of overfitting which occurred if the number of convolution layers increased beyond a certain point.

Fig 4.2 shows the difference between a 20-layer network and a 56-layer network. Contrary to the idea that more layers should give better results, the 20-layers network performed better when compared to the 56-layer network. This could have occurred based on the selection of hyperparameters, for instance, weight initialization of the network, or the selection of the optimization function, etc. However, the vanishing/exploding gradient could have played a role in this as well, which is a common issue seen in deep neural networks. Introduction of ResNets solved this issue, to be more specific, the introduction of skip connections in a network was the trick.

Fig 4.3 gives a visual representation of the process of identity mapping, i.e., to add the output of the previous layer, or input to the current layer, to the output of the same. The equation for identity mapping is:

$$y = F(x_o) + x_i$$

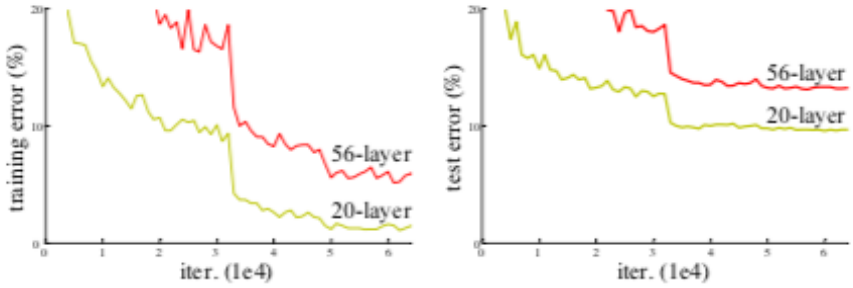

Figure 4.2: Left graph shows the training error and right graph shows the test error on CIFAR-10 dataset with 20-layer vs 56-layers plain network[4]

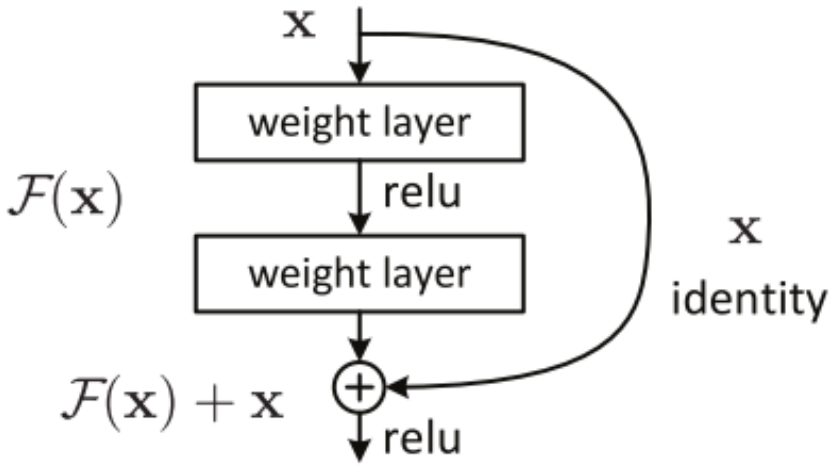

Figure 4.3: Building block of residual learning. The input to the block is added at the end to the output of the block. This process is called identity mapping[4]

where x_o is output of the current block(layer) and x_i is the identity(input to the block). In most of the scenarios, the identity won't have the same dimensions as the output of the block because of the convolution layers in between which changes the dimensions. Thus, the dimensions of the identity are changed accordingly. This process is called identity downsampling.

ResNets made it possible to add 100 layers without any issues of vanishing/exploding gradients. There are many versions of ResNets based on the number of layers added to it. All the versions have a set of convolution layer, max pool layer and activation function(ReLU) in common before the data passes through the layers.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 4.4: Architecture for ResNets. The column names depicts the number of layers present in the network. The first set of convolution layer, max-pool layer and ReLU activation function is common for all the architectures

Fig 4.4 gives a tabular representation of all the architectures of ResNets.

For this research, ResNet18 is selected, which contains 18-layers in total. One of the contributions of this research is to remove batch normalization from ResNets, which is computationally expensive, as mentioned in [8]. The experiments done on ResNets are targeted to achieve the mentioned purpose.

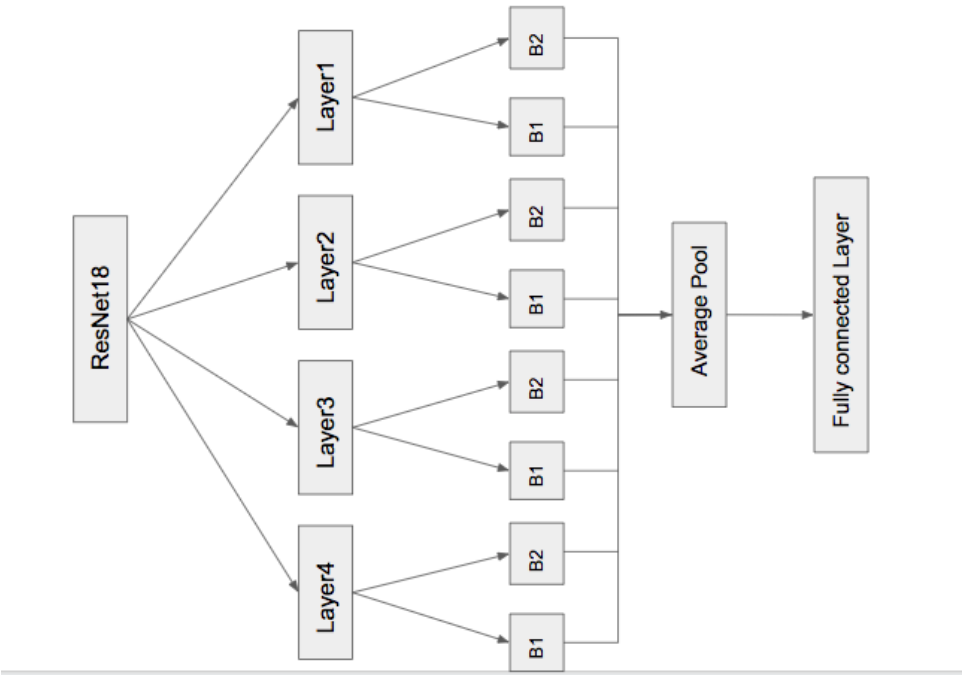

Figure 4.5: ResNet18 is divided into 4 layers, each layer containing two blocks, each block is a set of two convolution layers and activation function with identity downsampling. The blocks are followed by an average pooling layer, followed by a fully connected layer with output nodes based on the number of classifications

For this research, ResNet18 is selected to keep it simple in terms of the number of layers present in the architecture. Fig 4.5 gives a visual representation of the architecture for ResNet18. The first ResNet18 layer contains the set of common layers for all the architectures.

Chapter 5

Methodology

In this section, the methodology implemented is explained for each network. It is required to discuss the implementation in detail separately for LeNet and ResNet18 since the architecture of LeNet is different from ResNet. Moreover, LeNet is used to show that the proposed weight initialization technique can be used with any activation function, whereas ResNet18 is used to remove batchnorm from a network using the proposed method. The hyperparameters used for both the architectures are the same. Similarly, the approach for training individual layers are identical. Thus, both hyperparameters and the approach for training the individual layers are described together for LeNet and ResNet18.

5.1 Hyperparameters

In this section, the hyperparameters selected for learning the weights are described. Each selection is justified and compared with all the possibilities.

5.1.1 Loss and Optimizer

Loss: The loss function selected for the proposed method is mean squared error loss. Learning the optimal weights involves keeping the mean to zero for the layer output and variance to one for the activation outputs which means that it is a regression problem.

Optimizer: Optimizers are functions which are used to update the weights or learning rate of the neural network with the aim of reducing the loss for each iteration. The optimizer used for this research is Stochastic Gradient Descent(SGD).

5.1.2 Learning rate

Learning rate is related to the task in hand and cannot be generalized. For instance, in LeNet, a learning rate of 0.01 is used for all the layers.

5.1.3 Step Learning

Step Learning is the process of decaying the learning rate based on the number of epochs provided. For this research, a learning decay of 0.01 is applied after every fifth epoch. The interval is decided based on the layer trained.

5.1.4 Constant Parameter(Alpha)

A constant parameter(alpha) is introduced to reduce the difference in magnitude, if any, between the mean loss and the variance loss. For instance, if the mean loss has a magnitude in multiples of 1000 and the variance loss is lower than one, the mean loss gets multiplied

by alpha which scales down the magnitude of it which could be compared to the variance loss. The equation is provided for better understanding:

$$loss = (alpha * mean_loss) + variance_loss$$

Similarly, if the variance has a higher magnitude, it is scaled down using alpha:

$$loss = mean_loss + (alpha * variance_loss)$$

5.2 Approach

5.2.1 Target mean and variance for each layer

Standardization: The output of standardization basically means that it will have a mean of zero and a variance of one. It results in all the values centered around zero with a standard deviation of one. It is an important feature when gradient descent comes into the picture. If the scales of the data features differ a lot, few weights will get updated at a faster rate when compared to the other neuron weights which is not advisable for the model to learn. Moreover, symmetric activation functions, like tanh assume that the weights are distributed around 0. Thus, keeping the mean of the layer output is important for the network to learn. To avoid the exploding/vanishing gradient problem, it is important to keep the variance within a bound for each activation output. This approach is similar to the one mentioned in both Kaiming[3] and [2] with only one difference, i.e., Xavier weight initialization tries to keep the mean of the activation output to zero which doesn't work for ReLU[1] activation function which leads to the introduction of Kaiming[3]. The aim of kaiming is to keep the variance to one for the activation outputs with a small increase in the mean of the activation outputs through each iteration.

The proposed method can be described in two steps:

1. Train the model to keep the activation output variance to one.

2. Train the model to keep the mean to zero for each layer output.

The first step is self-explanatory. The layer is trained to keep the variance of the activation output to one using samples from the same dataset which will be used for training. This makes sure that the activation output never vanishes or explodes during forward propagation of data and similarly, this ensures that the gradient also stays in check when the loss is propagated backward and the weights are updated accordingly. That said, if the layers are trained to have an activation output of one, the layer weights will get updated with each iteration. This means that which each iteration, the mean of the layer outputs will also change which is not the appropriate scenario since the updated weights are considered to be the optimal weights and it is advisable to initiate the weights of a layer with a mean of zero to prevent the layer outputs from vanishing or exploding throughout the training process.

5.2.2 Initializing weights for each layer

The weights of each layer are initialized from a normal distribution within an interval [0,0.1]. This decision could be justified by the fact that the aim of the proposed method is to learn the optimal weights, but it is important to initialize the weights in such a way that it is easier for the network to reach its target in an optimal way. Thus, the weights are initialized randomly from the given interval such that all the values are near to zero and the difference is small between the values. A smaller interval could have been taken but this selection provided the optimal results.

5.2.3 Training each learnable layer and activation layer independently

As described in 5.3.1, all the layers with learnable weights and their corresponding activation layers are converted to models to be trained for the optimal weights.

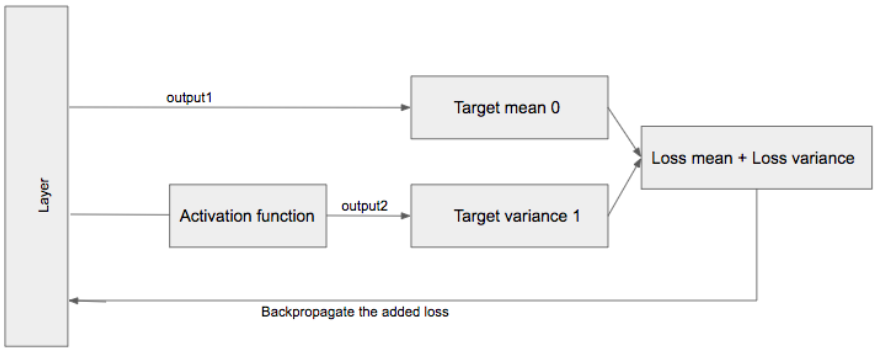

Figure 5.1: Data is processed through each layer.Mean loss is calculated for the layer output and variance loss is calculated for the activation output. The losses are added and back-propagated through the network. The weights are updated at each iteration.

5.2.4 Update weights and train the model

Once all the layers are trained separately and updated with the optimal weights, the weights in the original architecture are updated with the newly learned weights. The model is trained with the newly learned weights and the results are recorded.

5.2.5 Greedy Approach

Greedy approach can be described as the method of dividing the problem into multiple parts in sequential order with a target of achieving the optimal result when the last task has been executed, without considering the whole picture.

The proposed approach follows greedy algorithm. Each layer is converted to a model and the layers are trained sequentially, i.e., not more than one layer gets prioritized at a time. For instance, the second convolution layer in LeNet will only be trained if the first layer mean and variance loss converges.

The steps are described below in detail where two consecutive layers are considered for better understanding of the reader:

- Use the mini-batch created for learning the weights to train the first layer.
- Define the hyperparameters needed to train the layer.
- Make sure the loss converges for both mean and loss independently.
- Freeze the weights of the trained layer.
- Start training the second layer where the input to it is the output of the first layer.

The steps mentioned above are applied to all the layers in order. Since the proposed method makes sure that the previous layer has already been updated with the optimal weights.

As shown in Fig 5.2, greedy algorithm is used to learn the optimal weights at each instance. First, weights for Convolution 1 are learned, and the weights are frozen for that particular layer. Then, the learning process starts again for the second convolution layer,

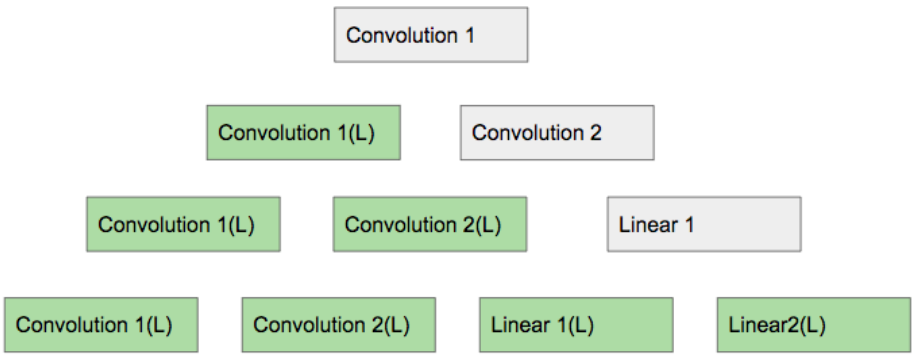

Figure 5.2: Each level describes the weight learning process of one layer at a time. Green layer depicts that the layer has already been trained. Last level shows that all the layers are trained to be initialized with the optimal weight.To describe the greedy approach, LeNet5 architecture is selected

where the data is first passed through the first convolution layer and that output is used as an input for the second convolution layer. This process is continued till the second last layer, i.e., excluding the decision layer. For visualizing the greedy approach, LeNet5 architecture is selected since ResNet18 architecture is complicated and difficult for the reader to understand from the visualization.

5.3 LeNet:Removing dependency on Activation functions

5.3.1 Each layer as a model

The model used for the algorithm is LeNet which consists of a set of two convolution layers, activation layers and maxpool layers and three linear layers with the last one as the decision layer.

Fig 4.1 gives a visual representation of the architecture for LeNet.

Each layer is considered as a model and the optimal weights are learned one-by-one in order.

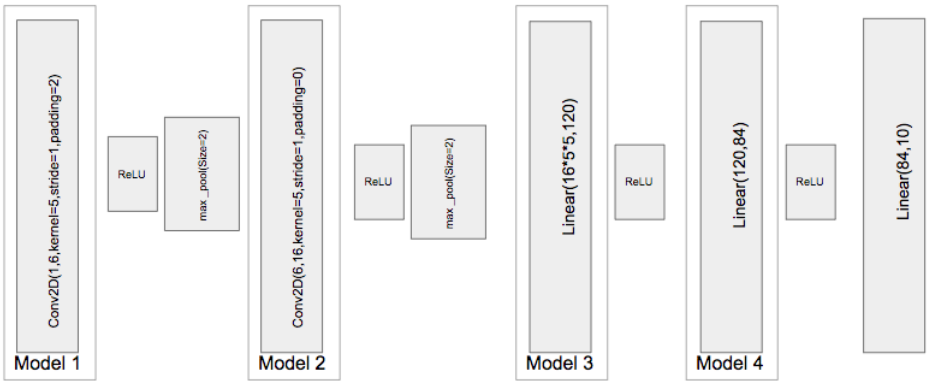

Figure 5.3: Each layer is considered as an independent model excluding the final layer. In LeNet, four new models are created, i.e., two convolution layers and two linear layers. Maxpool layers are not considered and added as separate entities but not involved in the learning process

Fig 5.3 gives a visual representation of the first step of learning, i.e., separating all the layers. The hyperparameters used are described in 5.1 section. Different combinations of activation functions and weight initialization techniques are used and the results are recorded for comparison with the proposed method.

5.4 ResNet18: Removing batch normalization

The two main properties which differentiate ResNet from other architectures is the introduction of skip connections and identity downsampling. Each layer in each block is trained separately. In ResNet18, there are three blocks with identity downsampling, i.e., a convolution layer is introduced to downsample the input to the block for the purpose of adding it to the block output. All the convolution layers are trained, including the downsample convolution layer.

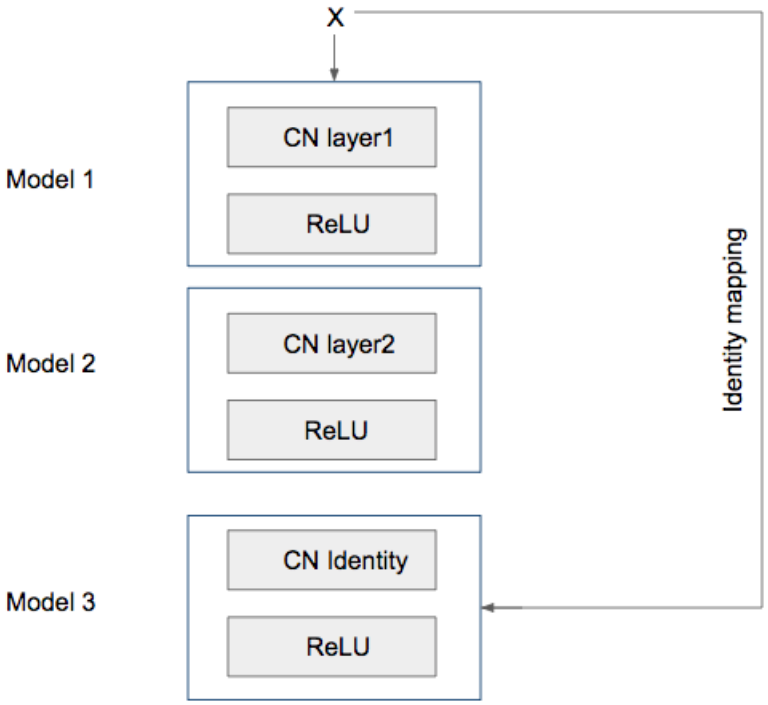

Figure 5.4: Each layer and it's corresponding ReLU activation layer is considered as a trainable model. The identity is recorded and transferred to the convolution layer used for downsampling and adding identity to the output.

Fig 5.4 shows the training process of a block with identity downsampling. The same process is applied to all the 18 layers present in the architecture. The training process for the mean and variance is same as described for LeNet5.2.3.

5.5 Ablation Study

To check whether the proposed method initializes the weights with optimal values, it is important to check whether targeting only mean and variance would not give the same performance. The ablation study is divided into two parts, based on the architecture and the purpose of the research. The first set of ablation experiments is conducted to find out the practicality of the proposed method, i.e., training the layers to have a mean 0 for the layer output and variance 1 is able to learn the optimal weights for a given network and dataset. The second set of ablation experiments are conducted to understand the impact of the proposed method compared to the kaiming weight initializer after removing batch normalization from the network. The below-mentioned ablation experiments are followed to understand the impact of the alternatives for LeNet :

1. **LAE1:**Learn weights for mean 0 and variance 1 for the activation output, initialize the weights in the network and train the model. Test the model accuracy on the test dataset.
2. **LAE2:** Learn weights for only mean 0 for the activation output, initialize the weights in the network and train the model. Test the model accuracy on the test dataset.
3. **LAE3:**Learn the weights for only variance 1 for the activation output, initialize the weights in the network and train the model. Test the model accuracy on the test dataset.
4. **LAE4:**Learn the weights for mean 0 and variance 1 for the layer output, initialize the weights in the network and train the model. Test the model accuracy on the test dataset.

The ablation experiments for LeNet are discussed in detail below:

LAE1: Learn weights for mean 0 and variance 1 for the activation layer Mini-batch created with random samples are used to feed forward through the network. Learning the mean to be zero and variance to be one for the activation function is a linear regression problem. Thus, mean squared error loss is used to calculate both the losses. Both mean loss and variance loss are added and back-propagated through the network and the weights are updated. The process continues till the loss converges.

Figure 5.5: Updating weights for one iteration for one layer to have a mean zero and variance one for the activation output

Fig 5.5 gives a visual representation of the learning process for the given ablation experiment, where both mean and variance loss are calculated for the activation output.

LAE2: Learn weights for only mean 0 for the activation output: In this ablation experiment, the network weights are learned only to have a mean of 0 for the activation output. Similar to the previous experiment, a mini-batch of data samples and mean squared error loss are used to learn the weights for the network. Fig 5.6 gives a visual representation of

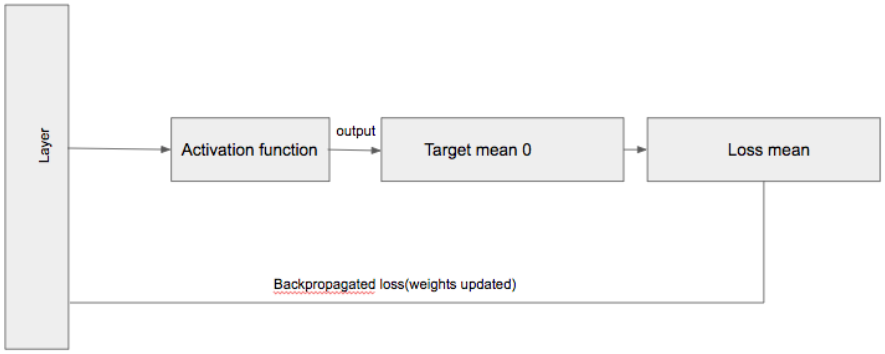

Figure 5.6: Updating weights for one iteration for one layer to have a mean zero for the activation output

the learning process for the given ablation experiment.

LAE3: Learn weights for only variance 1 for the activation function A mini-batch of data and mean squared error loss is used to learn the weights for a variance of 1 for the activation output.

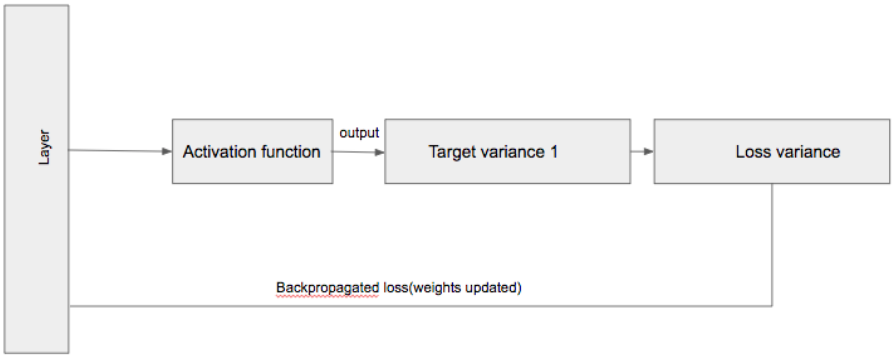

Figure 5.7: Updating weights for one iteration for one layer to have a mean zero for the activation output

Fig 5.7 gives a visual representation of the learning process for the given ablation experiment. Training the variance of the activation output to be 1 will change the mean of the layer output, thus, deviating it from 0, which is theoretically not recommended for initializing the weights.

LAE4: Learn weights for mean 0 and variance 1 for the activation output A mini-batch of data and mean squared error loss is used to learn the weights for a mean of 0 and variance of 1 for the layer output. Fig 5.8 gives a visual representation of the learning pro-

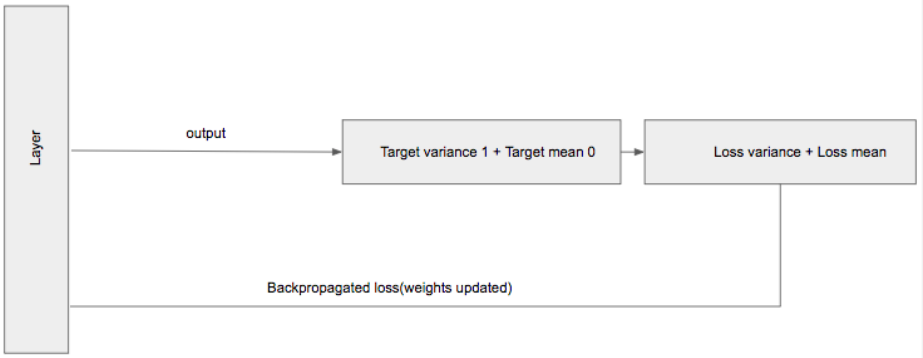

Figure 5.8: Updating weights for one iteration for one layer to have a mean zero and a variance of 1 for the activation output

cess for the given ablation experiment.

The below mentioned ablation experiments are followed to understand the impact of the alternatives for ResNet18:

1. **RAE1:**Remove BN and add kaiming weight initializer to all the layers. Train the model and check accuracy on test data.
2. **RAE2:** Remove BN and randomly initialize the weights for each layer. Train the model and check accuracy on test data.

RAE1: Remove BN and add kaiming weight initializer In this ablation experiment, BN is removed from each layer and all the layers are initialized with kaiming weight initialization technique. Visual representation of the same is not provided since there is no change in the learnable parameters or the architecture of ResNet18.

RAE2: Remove BN and add random weight initializer In this ablation experiment, BN is removed from each layer and all the layers are initialized with random weight initialization technique. That said, the weights must be initialized in an acceptable way. Thus, the traditional way of initializing the weights from a normal distribution with mean 0 and variance 1 is used.Visual representation of the same is not provided since there is no change in the learnable parameters or the architecture of ResNet18.

5.6 ResNet:Removing Batch Normalization

ResNets use ReLU activation function. Thus, the advisable weight initializer to be used is Kaiming weight initializer. ResNets solved the issue of the decrease in accuracy with an increase in the number of layers with the introduction of identity downsampling. That said, ResNets experience vanishing/exploding gradient without batch normalization. Batch normalization is computationally expensive and can only be used when the data is processed through the network through batches. Moreover, calculating the mean and variance for the batches during the testing process is difficult.

The proposed method is intended to replace the kaiming weight initializer as well as remove batch-normalization from all the layers. The aim is to get equal or better accuracy when compared to the model with both kaiming and batch-normalization.

ResNet18 architecture is used for this research. Each set of convolution layer and it's activation output is trained separately to have a mean of 0 for the layer output and a variance of 1 for the activation output. As shown in Fig 4.5, the architecture is divided into four layers. Layer2, Layer3 and Layer4 has identity downsampling. Thus, identity is stored separately which gets added to the last convolution layer of the block since all the convolution layers are trained independently.

5.7 Converging mean and variance loss

5.7.1 Training LeNet layers

Converging loss for mean and variance Each set of LeNet layer and activation layer is trained separately till the loss converges to it's lowest value. The graphs below shows the learning process for each layer and it's activation output. The dataset used is Fashion-MNIST for the graphs. Activation function used is ReLU. Each layer is trained for 200 epochs. The number of data samples used for training the layers are 25.

1. First Convolution layer Mean loss

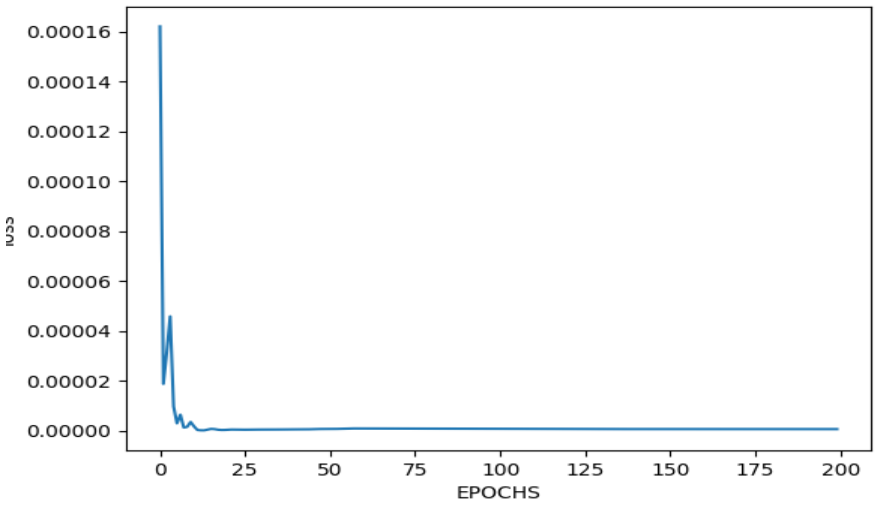

Figure 5.9: First convolution layer loss per epoch for mean 0.x-axis : loss, y-axis: epoch number

Fig 5.9 shows the learning process for the first convolution layer output to have a mean of 0. Step Learning rate with a learning rate decay of 0.01 per 5 epochs is used for this layer. Fig 5.9 shows the drop and convergence of mean loss for the first convolution layer in the network.

2. First Activation layer Variance loss

Fig 5.10 shows the learning process for the first activation layer output to have a variance of 1. Step Learning rate with a learning rate decay of 0.01 per 5 epochs is used for this layer.

Fig **??** and Fig **??** shows the learning process for the mean and variance separately to give a better understanding of the learning process. In reality, both mean for the layer output and variance for the activation output were trained together. The mean loss and variance loss were added and the weights were updated accordingly during the backpropagation of the loss.

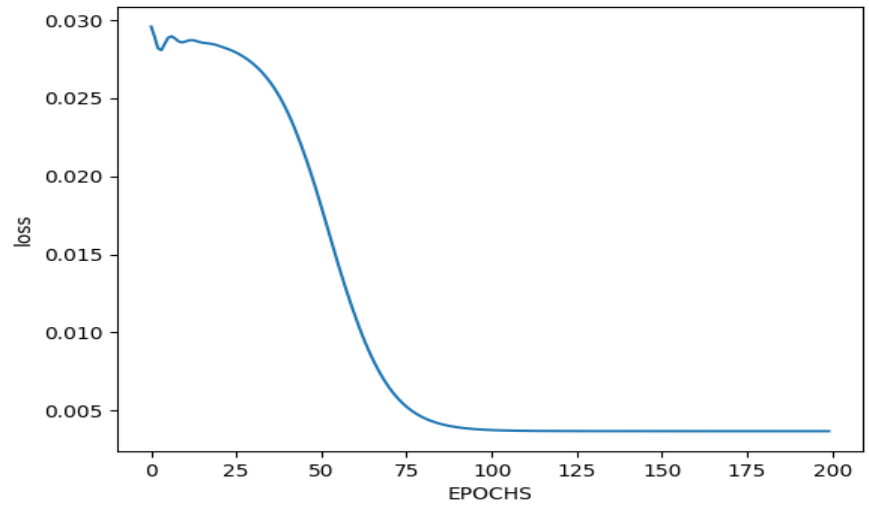

Figure 5.10: First activation layer loss per epoch for variance 1.x-axis : loss, y-axis: epoch number

Similarly, the remaining convolution layer, followed by the activation layer are trained together but the graphs are plotted separately.

The learning graph for the remaining layers are provided as follows:

3. Second Convolution layer mean loss

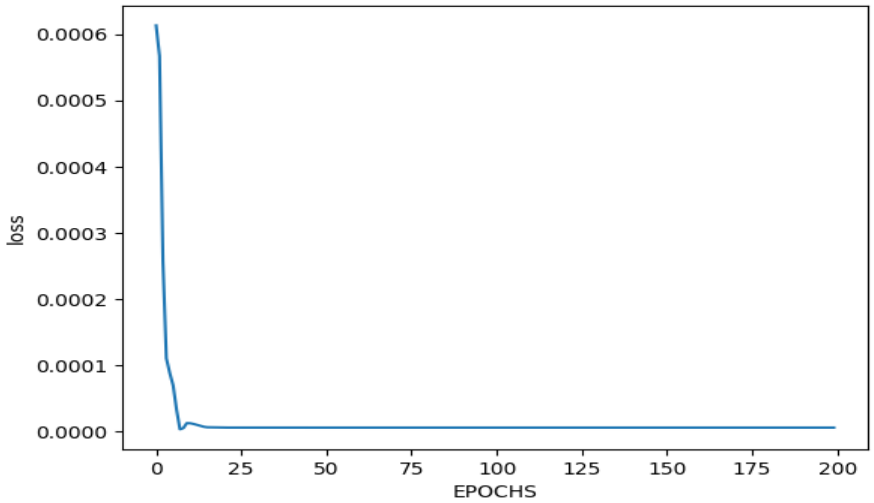

Figure 5.11: Second Convolution layer loss per epoch for mean 0.x-axis : loss, y-axis: epoch number

3. Second Activation layer variance loss

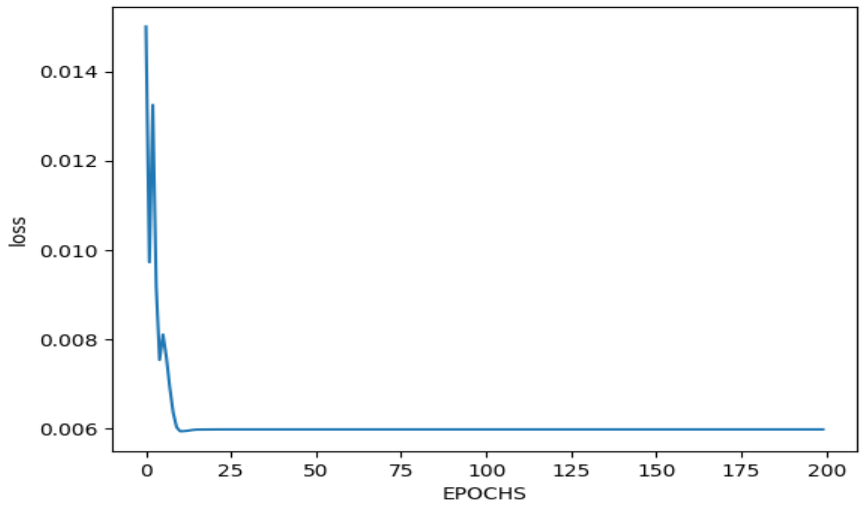

Figure 5.12: Second Activation layer loss per epoch for variance 1.x-axis : loss, y-axis: epoch number
Second Convolution layer and second activation layer are trained together.

3. First Linear layer mean loss

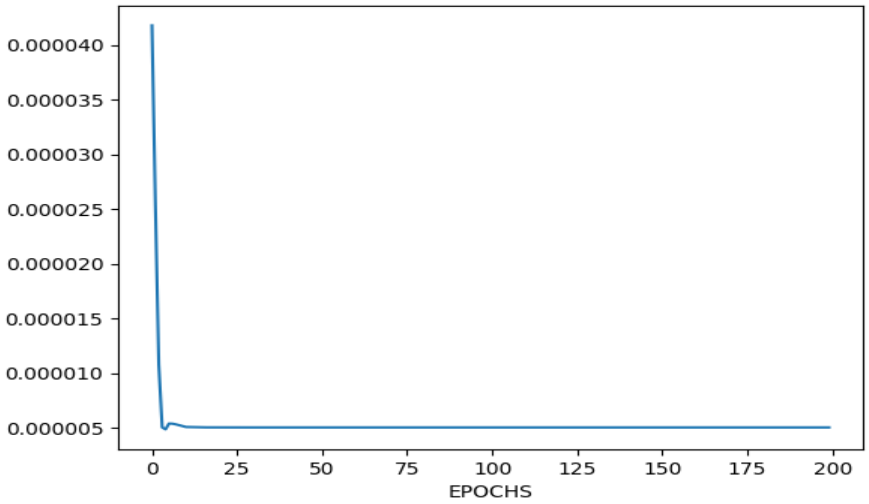

Figure 5.13: First linear layer loss per epoch for mean 0.x-axis : loss, y-axis: epoch number

3. Third Activation layer variance loss

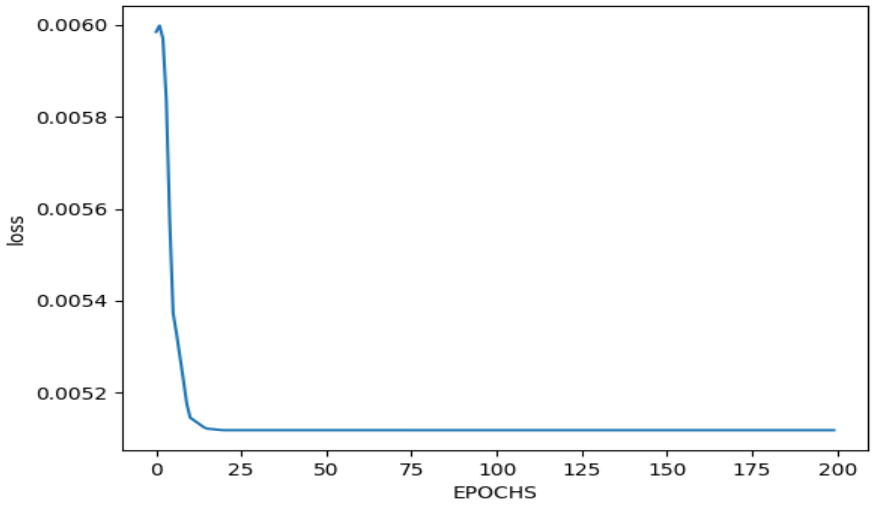

Figure 5.14: First linear layer loss per epoch for mean 0.x-axis : loss, y-axis: epoch number

Similar to the previous layers, First linear layer and third activation layer are trained together.

3. Second Linear layer mean loss

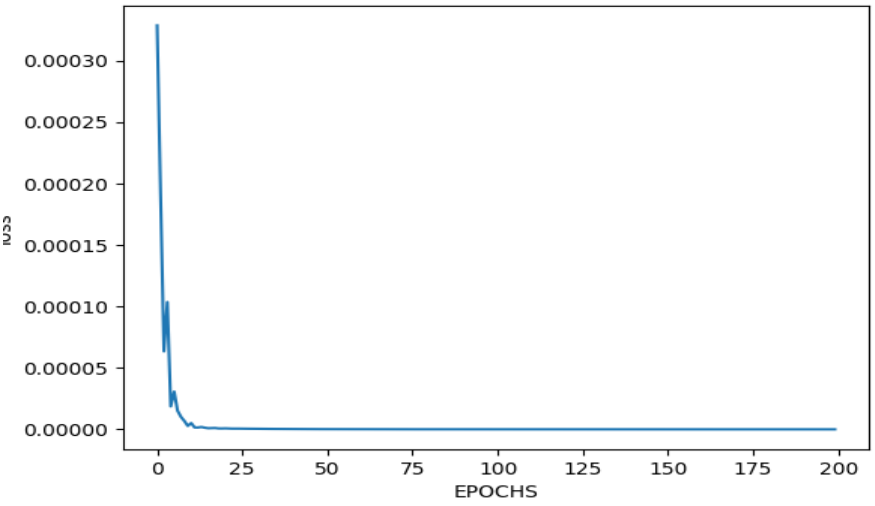

Figure 5.15: Second linear layer loss per epoch for mean 0.x-axis : loss, y-axis: epoch number

3.Fourth Activation layer variance loss

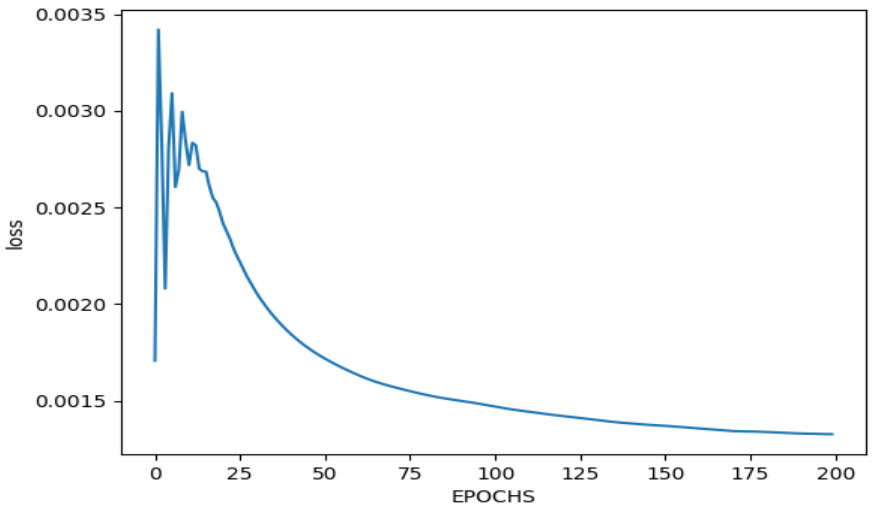

Figure 5.16: Fourth activation layer loss per epoch for variance 1.x-axis : loss, y-axis: epoch number

Each layer is dependent on the previous layer output. As described in the methodology section, once a layer is trained, the layer is frozen so that the weights doesn't change while training the next layer. Thus, the proposed method follows a greedy approach.

The step Learning rate of 0.01 per 5 epochs is used for all the layers.

Chapter 6

Results

6.1 Ablation Study

In this section, results related to the ablation experiments are discussed. For LeNet, the activation function selected for the ablation study is ReLU.

Ablation Experiment	Activation Function	Test Accuracy
LAE1	ReLU	93.36%
LAE2	ReLU	92.24%
LAE3	ReLU	10%
LAE4	ReLU	93.48%

Table 6.1: Test Accuracy for Ablation experiments on LeNet

As shown in table 6.1, the model doesn't train for LAE3. For the other scenarios, the accuracy is high since the dataset selected for the same is FashionMNIST. That said, the proposed method is expected to give a higher accuracy than all the ablation experiments.

Ablation Experiment	Weight initializer	Test Accuracy
RAE1	Kaiming	10.46%
RAE2	Random	10.01%

Table 6.2: Test Accuracy for Ablation experiments on ResNet18

Table 6.2 shows the experiment results for the ablation experiments related to ResNet18. As shown, the model doesn't train without batch-normalization. The activation function used for both the ablation experiments is ReLU. The proposed method is supposed to fix the issue of batch normalization and the model should be able to learn without the same.

6.2 LeNet

6.2.1 FashionMNIST and analysis

FshionMNIST is used for sanity check of the proposed method. To understand the effect and to compare it's performance to the existing methods, all possible combinations of activation functions(Sigmoid,tanh,ReLU) and weight initializers(learned weights, Xavier, Kaiming) are used.

AF	LW	Kaiming	Xavier
Sigmoid	99.56%	99.02%	99.70%
Tanh	95.56%	92.24%	94.68%
ReLU	95.24%	96.36%	95.85%

Table 6.3: Test Accuracy for different combinations of weight initializers and activation functions

Table 6.3 shows that Learned weights(LW) performs slightly better than Kaiming and Xavier for Sigmoid and Tanh activation functions. For ReLU, the performance can be considered same as the difference in accuracy is low. FashionMNIST is a simple dataset to be trained. Thus, the test accuracy is high for all the combinations.

6.2.2 Imagenette and analysis

Imagenette is used to observe the performance of the method for datasets with more complexity, which can show the difference in training if the weight initializer used is not appropriate for the activation function. For instance, table 6.4 shows the drop in accuracy when tanh activation function is used. Theoretically, Xavier weight initializer should be able to perform well with symmetric activation functions like tanh, but for the given scenario, the proposed method performs better than all the combinations used.

AF	LW	Kaiming	Xavier
Sigmoid	99.56%	65.43%	46.80%
Tanh	96.01%	46.02%	47.56%
ReLU	95.24%	92.40%	95.85%

Table 6.4: Test Accuracy for different combinations of weight initializers and activation functions

As shown in table 6.4, the proposed method(LW) performs better than both kaiming and xavier for all the combinations of Activation functions(AF), except ReLU, which can be considered to have the same accuracy since the difference is less than one percent.

6.3 ResNet18

The experiments done with ResNet18 are done with the purpose of verifying the hypothesis presented in **RQ2**, i.e., the proposed method can remove batch normalization from any network, regardless of it's depth.

The weight learning process for a layer in ResNet is similar to that of LeNet. Thus, a detailed description is not provided for the weight learning process. This section contains the experiment results and observations for Imagenette datasets. Different combinations of activation functions and weight initializers are not used since the aim for this architecture is to remove batch normalization from the network.

LW accuracy	Kaiming + BN accuracy
68.82%	72.48%

Table 6.5: Test Accuracy for Learned weigths(LW) and the combination of Kaiming and Batch Normalization(BN)

The table 6.5 shows that the proposed method is able to learn but the accuracy is lower by approximately four percent when compared to the combination of batch normalization(BN) and Kaiming weight initialization. The accuracy of the proposed method can be further improved by changing the hyperparameters and training the model in it's optimal conditions, since training a model depends on other factors and not only on the initialization of weights. That said, LW is still able to learn without introducing batch normalization in the network.

Chapter 7

Conclusion and future work

7.1 Conclusion

The research proposes a new technique to learn the optimal weights through training each layer of a network by following the concept of standardization. The hypothesis regarding the first research question(**RQ1**) is confirmed by the experiments done on LeNet architecture. The proposed method is combined with different activation functions and weight initializers to verify whether it can be used in any architecture regardless of the activation function. It achieves better results than the existing weight initialization techniques in most of the scenarios. For the second hypothesis proposed in the second research question(**RQ2**), i.e., the proposed method can replace batch normalization in a network, experiments were conducted on ResNet18 and accuracy was compared by removing batch normalization and kaiming from the network and replacing it with the proposed method. The ablation study performed on ResNet18 shows that adding kaiming weight initializer or random weight initialization to the network without batch normalization cannot train the network. However, initializing the weights of the network with the method introduced can make the network learn. Moreover, there is a difference of only 5% between batchnorm and the proposed method which can be further improved by tweaking the parameters. The technique has a small time overhead since the weights need to be learned and not defined at the beginning of the training process. That said, the time overhead is small since learning the weights to have a mean of 0 and the activation output to have a variance of 1 is a linear regression problem.

7.2 Future Work

The method introduced can be further used to remove different types of norms, for instance, layer norm or group norm. A more generalized structure for learning the weights can be implemented, where hyperparameter optimization is already introduced for each layer for faster convergence.

Bibliography

[1] Abien Fred Agarap. Deep learning using rectified linear units. URL https://www.researchgate.net/publication/323956667_Deep_Learning_using_Rectified_Linear_Units_ReLU.

[2] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks, 2010.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, December 2015.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. doi: 10.1109/cvpr.2016.90.

[5] Jeremy Howard. Imagenette. URL https://github.com/fastai/imagenette.

[6] Wei Hu, Lechao Xiao, and Jeffrey Pennington. Provable benefit of orthogonal initialization in optimizing deep linear networks, Jan 2020. URL https://arxiv.org/abs/2001.05992.

[7] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278â2324, 1998. doi: 10.1109/5.726791.

[8] Dmytro Mishkin and Jiri Matas. All you need is a good init, Feb 2016. URL https://arxiv.org/abs/1511.06422.

[9] Pytorch. Imagenette pytorch dataset. URL https://mf1024.github.io/2019/06/22/Create-Pytorch-Datasets-and-Dataloaders.

[10] Sweta Shaw. *A Comparative Study of Activation Functions*, 2009 (accessed February 3, 2014). URL https://app.wandb.ai/shweta/Activation%20Functions/reports/A-Comparative-Study-of-Activation-Functions--VmlldzoxMDQwOTQ.

[11] Ashwarya V Srinivasan. Stocahstic gradient descent. URL https://towardsdatascience.com/stochastic-gradient-descent-clearly-explained-53d239905d31.

[12] Avinash Sharma V. Theory of everything. URL https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0.

[13] zalando. Fashion-mnist. URL https://github.com/zalandoresearch/fashion-mnist.