Neural Network Architectures for FashionMNIST Classification

Kutay Demiralay
Aeronautics and Astronautics Department Department, University of Washington, Seattle, WA

In this project, our objective is to train different neural network architectures (FCNs/DNNs/CNNs) for image classification using the FashionMNIST dataset. We'll be tasked with determining the architecture of our model, fine-tuning hyperparameters for optimal performance, and comparing the performance of different models against each other.

Introduction and Overview

The original MNIST training dataset comprises 60,000 samples, each representing a distinct handwritten digit changing from 0 to 9 randomly. Each digit is characterized by 784 features, denoting individual pixels in a 28x28 resolution image. This dataset is utilized for training our algorithm. Additionally, the test dataset consists of 10,000 sample images, each with the same 784 features, and is employed for validating the performance of the algorithm. Different classification methods are applied to analyze and classify this dataset using python programming language.

As in MNIST, in FashionMNIST each sample is a grayscale 28 × 28 image, 784 dimensions in total, and the training set consists of 60K images and the testing set consists of 10K images. There are 10 different classes, each representing a different type of clothing or accessory: T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, and Ankle boot. FashionMNIST was created by Zalando Research to provide a more difficult and realistic dataset for benchmarking machine learning algorithms, as the original MNIST dataset had become too easy for modern models

Theoretical Background

A Fully Connected Deep Neural Network (FCN/DNN) model is a layered architecture that includes multiple interconnected neurons. This model features three primary layers: the input layer, hidden layers, and output layer. In our image classification task, the input layer takes in pixel values representing fashion items. The hidden layers work to uncover features and patterns embedded in these pixel values, while the output layer delivers the final predictions on the fashion item's class. The interplay between these layers empowers the model to grasp intricate relationships within the data, refining its performance through training over time. Central to how the model operates are the weights linked with each connection between neurons as can be seen on figure 1. These weights essentially dictate the strength of these connections, serving as key factors in molding the model's capacity to learn from the data. As the model undergoes the training process, it fine-tunes these weights, honing its grasp of intricate relationships within the data. This ongoing adjustment significantly contributes to the model's improvement over time, refining its performance without relying on explicit programming.

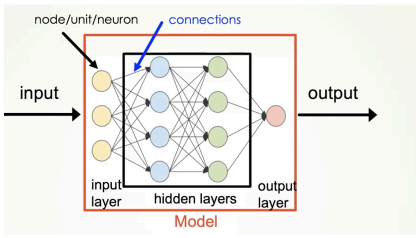

Figure 1: Fully Connected Deep Neural Network Model [2]

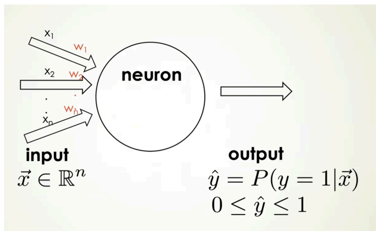

Figure 2: Neuron-Computational Unit [2]

Similar to how weights influence the strength of connections, the activation energy, akin to a decision threshold, plays a role in determining whether a neuron 'activates' or not. The equation (1) provides the expression for the output of a neuron,

$$y=f(\sum_{i=1}^{n} x_i\, \omega_i + b) \quad (1)$$

The output equation provides a probability value where f is the activation function, x is the input, w represents weights assigned to each input, and b is the bias term. This encapsulates the role of each fundamental neural unit in a neural network, which is modeled in fig. 2. An activation function is like a special rule applied to the information that enters a node in a neural network. It brings in a bit of nonlinearity which is important for the network to understand intricate patterns and connections in the information. In our case, we used the function ReLU (Eq. (2)) in code as activation func.

$$f=\max(x,0) \quad (2)$$

Cross-entropy loss, or log loss, is a commonly used measure in classification problems. It assesses how well a classification model is performing when its output is a probability value between 0 and 1. The loss increases when the predicted probability deviates from the actual class label. Loss is described in Equation 3. Here, y is the true label (0 or 1), and y hat is the predicted probability of the instance belonging to class 1.

$$L(y,\hat{y})=-(y \log \hat{y} + (1+y) \log (1-\hat{y})) \quad (3)$$

In the end our cost is the sum of all the losses, normalized by the m, number of points in the set. We need to minimize the cost, described by Equation 4, as much as possible for best classification

$$J=\frac{1}{m}\sum_{i=1}^{m} L(y,\hat{y}) \quad (4)$$

The optimization process involves adjusting the weights and biases to minimize the loss. The gradient descent equation in Equations 5 and 6, shows how we can iteratively optimize the weight vector and bias term. The alpha value, our learning rate, determines the size of each step in the direction of the gradient.

$$\vec{w}_{k+1}=\vec{w}_{k}-\alpha\Delta_{\vec{w}} J(\vec{w}_{k};b) \quad (5) \qquad b_{k+1}=b_{k}-\alpha\frac{\delta}{\delta_b} J(\vec{w}_{k};b_{k}) \quad (6)$$

In the diagram below in figure 3, the flow of our model is illustrated. During forward propagation, computations are made to compute the loss for a single training example and to predict the output. In contrast, backward propagation calculates partial derivatives to determine the gradient, aiding in the optimization process. Smaller subsets that we compute the gradients of, are known as batches, where the training set is divided into more manageable portions. When we compute the computation for an entire batch, we will call it an iteration. An epoch occurs when the entire dataset undergoes both forward and backward propagation, which consists of computing the gradients, summing them up, updating the weights, and iterating for a number of epoch times.

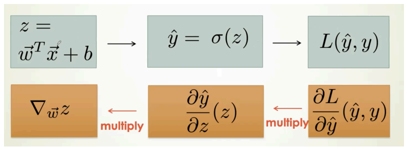

Figure 3: Flow of Model [2]

Scholastic gradient descent utilizes the gradient descent method mentioned above. However, instead of summing the gradients over the entire training set for one update, it updates the weights for each sample individually. In other words, it performs multiple updates (m updates) for the entire training set. Other Gradient Descent variations include:
Momentum: Accelerates gradient vectors in the right directions for faster convergence
RMS Prop: Extended and weighted version of AdaGrad.
AdaM: Adaptive Learning Rate + Momentum.

In dropout regularization, during training, randomly selected neurons are ignored or "dropped out" on each forward pass. This means that the contribution of those neurons to the network is temporarily removed. The idea is that this helps prevent the network from relying too much on specific neurons and encourages it to learn more robust features, decreasing overfitting characteristics of the model.. The probability of dropout is a hyperparameter, which should be pre determined in code.

We will standardize the outputs of each layer by normalizing the mean to be zero and the variance to be one. This process, known as batch normalization, helps control the propagation through the network. This is done using Equation 7. Where z norm is the normalized output value we acquire through normalization, miu is mean and sigma is the standard deviation.

$$\mu=\frac{1}{m}\sum_{i=1}^{m} z^{(i)} \qquad \sigma^2=\frac{1}{m}\sum_{i=1}^{m} (z^{(i)}-\mu)^2 \qquad z^{(i)}_{\text{norm}}=\frac{z^{(i)}-\mu}{\sqrt{\sigma^2-\epsilon}} \quad (7)$$

Initialization is the setting of initial values for weights and biases. The Xavier Initialization method maintains consistent activation variances across layers by starting with weights from a distribution with a zero mean. He Initialization, suitable for ReLU activation, initializes weights with a variance linked to the number of input units. Random Normal Initialization disrupts symmetry by assigning small random values drawn from a normal distribution to weights. The choice of initialization method impacts the network's ability to converge and its overall performance, addressing challenges such as vanishing or exploding gradients while facilitating effective learning across layers.

Convolutional Neural Networks

Convolutional Neural Networks are a specialized kind of model used for tasks involving images. At their core, CNNs use a technique called convolution, which is essentially a way of scanning through an image to find patterns or features. In this process, filters, which are small and adjustable matrices, slide across the image, multiplying and adding up the values they encounter. This operation results in what we call a feature map, highlighting important patterns in the image. Unlike FCNs, which reshape the input into a vector, CNNs work directly with the input's spatial structure. CNNs explicitly aim to preserve spatial information throughout the network, thanks to the convolutional and pooling layers. CNNs use filters that slide over the input, focusing on local regions and capturing features. This is in contrast to FCNs, which might lose spatial information by flattening the input. The use of convolutional filters is parameter-efficient compared to fully connected layers. The lecture highlights the significant reduction in the number of parameters, making the network more manageable and computationally efficient. The convolution involves sliding the filter over the input signal and computing the element-wise product of the filter and the corresponding section of the input. The products are summed to obtain a single value, which is recorded in the result, known as the feature map.

$$F[m,n] = I[m,n] \times w[m,n] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} I[i,j] w[m-i,n-j] \quad (27)$$

CNNs can provide a more computationally efficient way to process images, especially when dealing with large networks. This efficiency is due to the shared weights in the convolutional layers. Equation for convolution operation is given at Eq.(27). Where F represents the feature map, I represents Input, and W represents filter. In this case, it is a double summation over i and j from −∞ to +∞ , which accounts for all possible values of the discrete indices. The expression inside the double summation is I [i , j] × w [m − i , n − j] I[i,j]×w[m−i,n−j], which is the product of the input signal I I at coordinates (i , j) (i, j) and the filter w w at the relative coordinates (m − i , n − j) (m−i,n−j). This product is computed for all possible values of i i and j j within the specified range. The result of each product is summed over both i i and j j, and the overall sum represents the value of the convolution at the output coordinate.In simpler terms, this formula describes how each point in the output signal is obtained by summing the products of corresponding points in the input signal and the flipped and shifted filter. Using multiple filters (e.g. a thousand filters) allows the network to capture various features in the input image, enhancing its ability to learn diverse pattern Filters are typically square, with dimensions denoted as fxf , where f is an odd number (e.g., 3x3, 5x5). The choice of filter size depends on the problem and input dimensions. Padding involves adding extra elements (often zeros) around the boundaries of the input.Padding helps preserve the spatial dimensions of the input data. Equation for width/height of feature maps of filtering is shown in Eq.(28)

$$n^{[l]} = (n^{[l-1]} + 2p - f)/s + 1 \quad (28)$$

Where n (l) is the dimension of output of filtering, n (l-1) s is the dimension of input of filtering, p is padding, padding helps preserve the spatial dimensions of the input data. and f is the dimension of the filter and s is the stride, which is the number of positions the filter moves in each step.

A full CNN consists of several layers of different kinds, including pooling, convolutional, and fully connected layers within the network In a general convolutional neural network (CNN), the total number of weights is determined by the architecture and parameters of the network. The weights are associated with learnable parameters, including the convolutional kernels, fully connected layers, and biases. The formula to calculate the number of weights in a layer depends on the layer type.For a convolutional layer, the number of weights is given by number of filters×number of input channels×filter height×filter width. For a fully connected layer, the number of weights is given by:number of neurons in the current layer×number of neurons in the previous layer. The total number of weights in the CNN is the sum of the weights in all layers.

Algorithm Implementation and Development

All coding was done in the python programming language. Numpy, Troch, tqdm, Torchvision, matplotlib libraries were used. Template provided on canvas was filled.

First Step: Downloading HW4_Helper.jpyn

This file worked as a template for the whole model. In this file, the FashionMNIST dataset for training and testing was downloaded using torchvision. It has useful transformations (e.g. normalization) and loads data as Pytorch tensors. The training data is split into Training and Validation datasets and loaded into DataLoader as batches. Sample code to visualize the first sample in first 16 batches and to plot 64 images from the dataset.

Second Step: Neural Network Model for Image Classification Using PyTorch

```
class ACAIGFCN(nn.Module):
  def __init__(self, input_dim, output_dim, hidden_dim1, hidden_dim2):
    super(ACAIGFCN, self).__init__()
    self.fc1 = nn.Linear(input_dim, hidden_dim1)
    self.relu1 = nn.ReLU()
    self.fc2 = nn.Linear(hidden_dim1, hidden_dim2)
    self.relu2 = nn.ReLU()
    self.fc3 = nn.Linear(hidden_dim2, output_dim)
```

```
def forward(self, x):
    x = x.view(x.size(0), -1)
    x = self.fc1(x)
    x = self.relu1(x)
    x = self.fc2(x)
    x = self.relu2(x)
    x = self.fc3(x)
    return x
```

The model is designed as a fully connected neural network with two hidden layers and ReLU activation.The initialization method initializes the neural network architecture. It sets up three fully connected layers (fc1, fc3, and fc2) and a ReLU activation function. Input dimension is 784 and output dimensiın is 10. The forward def method defines how input data is propagated through the network during the forward pass. It flattens the input, applies the fully connected layer with ReLU activation for two consecutive times, and then applies the output layer.

Third Step: Initializing neural network model with input, output, and hidden layer dimensions, defining learning rate, epoch number, loss function and optimizer type.

Hyper Parameters can be tuned in this part.

Fourth Step: Model Training

```
for epoch in tqdm.trange(epochs):
   for train_features, train_labels in train_batches:
      model.train()       # Set model into training mode
      train_features = train_features.reshape(-1, 28*28)    # Reshape images into a vector
      optimizer.zero_grad()     # Reset gradients, Calculate training loss on model
      outputs = model(train_features)
      loss = loss_func(outputs, train_labels)
      loss.backward()
      optimizer.step()
```

Loss function was determined as cross entropy loss. This training process repeats for each batch in each epoch, gradually improving the model's ability to make accurate predictions on the training data. The overall goal is to minimize the loss function by adjusting the model's weights. The outer loop iterates over the specified number of epochs, where each epoch represents a complete pass through the entire training dataset, while The inner loop iterates over batches of training data.

Fifth Step: Model Validation

The accuracy is a measure of how many predictions match the actual labels in the validation set.

```
   val_acc = 0.0
   for val_features, val_labels in val_batches:
        with torch.no_grad():    # Telling PyTorch we aren't passing inputs to network for training purpose
         model.eval()
         val_features = val_features.reshape(-1, 28*28)    # Reshape validation images into a vector
         val_outputs = model(val_features)    # Compute validation outputs (predictions)
         val_acc += (torch.argmax(val_outputs, dim=1) == val_labels).float().mean().item() # Compute accuracy
```

Sixth Step: Testing the Model

```
   for test_features, test_labels in test_batches:
        with torch.no_grad():     # Telling PyTorch we aren't passing inputs to the network for training purpose
         model.eval()
         test_features = test_features.reshape(-1, 28*28)    # Reshape test images into a vector
         test_outputs = model(test_features)           # Compute test outputs (predictions)
        test_acc += (torch.argmax(test_outputs, dim=1) == test_labels).float().mean().item()      # Computing accuracy
```

Seventh Step: Plotting Loss and Accuracy Curves vs. Epochs
Plotting Testing, Training, Validation Loss and Accuracy Curves vs. Epochs for all different models to compare them.

Bonus Coding Implementation Application, AlexNet (Intro To Machine Learning):

```
class SimpleCNN(nn.Module):    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5, stride=1, padding=2)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5, stride=1, padding=2)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc3 = nn.Linear(64 * 7 * 7, 256)
        self.fc4 = nn.Linear(256, 128)
        self.softmax = nn.Linear(128, 10)
```

```
def forward(self, x):
    x = self.conv1(x)
    x = torch.relu(x)
    x = self.pool1(x)
    x = self.conv2(x)
    x = torch.relu(x)
    x = self.pool2(x)
    x = x.view(-1, 64 * 7 * 7)
    x = self.fc3(x)
    x = torch.relu(x)
    x = self.fc4(x)
    x = torch.relu(x)
    x = self.softmax(x)
    return x
```

I also designed a CNN to get an idea about the performance difference between FCN and CNN. I used the famous classical network AlexNet's architecture shown in figure 4 below. Most of the process was the same as FCN, like training, testing, validation, loading the dataset, initializing the model, defining hyperparameters. However I used the above code for defining the CNN model. FashionMNIST Dataset is used.

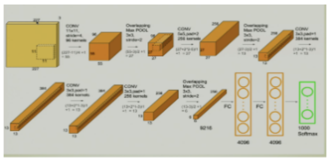

Figure 4: AlexNet Network Structure [2]

Computational Results

To achieve optimal performance, hyperparameters such as learning rate, the number of hidden layers, and neuron counts were determined through trial and error. The chosen hyperparameters are a learning rate of 0.02 , three hidden layers with the first layer having 300 neurons and the second layer having 200 neurons. The model was trained for 50 epochs with a training batch size of 512 and a test batch size of 256. These hyperparameters were carefully selected to ensure a testing accuracy of at least 85%, reasonable computation time, small sensitivity with no huge spikes in the loss curve plot, sensitivities, and convergence. The neural network architecture includes three hidden layers with ReLU activation functions; cross entropy loss was used.

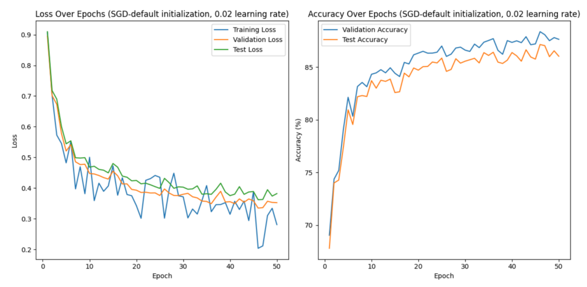

Figure 4: Loss and Accuracy Curves of Model With SGD Optimizer and Default Initialization, 0.02 Learning Rate
Final Train Loss: 0.2811, Final Validation Loss: 0.3526, Final Test Loss: 0.3822, Final Validation Accuracy: 87.63%, Final Test Accuracy: 86.03% , Computation Time: 13.65 minutes

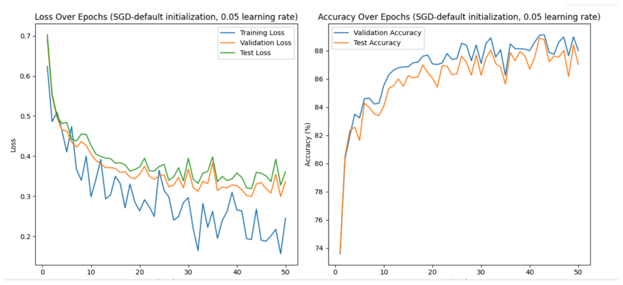

Figure 5: Loss and Accuracy Curves of Model With SGD Optimizer, Default Initialization, 0.05 Learning Rate
Final Train Loss: 0.2456, Final Validation Loss: 0.3369, Final Test Loss: 0.3614, Final Validation Accuracy: 88.01%, Final Test Accuracy: 87.03%, Computation Time: 14.312 minutes

With the learning rate increased from 0.02 to 0.05, we observed an increase in accuracies and a decrease in losses overall. However, the higher learning rate introduced more sensitivity, slightly more computation time and more underfitting, noticeable in both accuracy and loss curves plotted in the figure 5 above. The plots exhibited increased zig-zag patterns, indicating less stability and higher standard deviation with the elevated learning rate, and looking at the loss curves, a noticeable gap emerged between the training and validation curves,after 30 epochs, indicating a heightened possibility of underfitting. As a result, I prioritized stability and avoiding underfitting over a marginal 1% increase in testing accuracy. Therefore, I decided to maintain the base model's learning rate at 0.02. The test accuracy, which was already above 85%, remained satisfactory. This is how I determined learning rates for various models in this Homework, and this is how I determined if the model is underfitting, overfitting, through looking at loss and accuracy curves.

Next, I performed hyperparameter tuning from the baseline, considering different optimizers, including RMSProp and Adam each with varying learning rates. Table for comparison of their performance values is below in table 1.

Optimizer	Best Performing Learning Rate	Final Training Loss	F. Validation Loss	F. Test Loss	F. Validation Accuracy	F. Test Accuracy	Comp. Time (m)
SGD	0.02	0.2811	0.3526	0.3822	87.63%	86.03%	13.65
RMS Prop	0.0001	0.2353	0.3111	0.3505	88.81%	87.94%	14.66
Adam	0.00008	0.2557	0.3006	0.3253	88.50%	88.41%	14.75

Table 1: Performance Parameters of FCN models using different optimizers and different learning rates

After fine-tuning the hyperparameters of my SGD base model, I proceeded to explore the performance of Adam and RMSProp optimizers on the same base model. I focused on optimizing only the learning rates to achieve the best possible performance for my fully connected neural (FCN) models across different optimizers. What I found was that when using the learning rate employed for the SGD optimizer, the FCN models using Adam and RMSProp optimizers exhibited significant underfitting characteristics and heightened sensitivity. Consequently, I had to substantially decrease the learning rates to address these issues. We got the highest final testing accuracy and lowest final testing loss using Adam optimizer. Adam optimizer has very slightly less Testing accuracy then Validation accuracy, indicating very small underfitting. Both RMSprop and Adam optimizers exhibited better accuracy performance than the SGD optimizer, even with significantly lower learning rates. However, it is noteworthy that they displayed more pronounced underfitting characteristics compared to the SGD optimizer, as it can be seen in their respective loss curves in fig (6). This is evident from their loss versus epoch graphs presented below, where the training loss plot converges to a lower loss value than the validation and test plots. And also RMSProp and Adam optimizers used slightly more computation time, with Adam optimizer having the highest computation time.

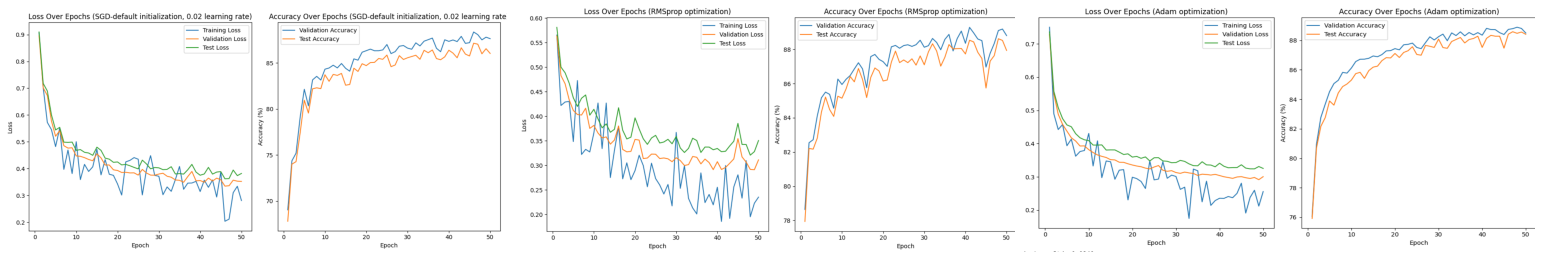

Figure 6: Loss and Accuracy Curves of Models Using SGD, RMS Prop, Adam Optimizer Respectively

Next I added Dropout regularization to my model and analyzed whether this improves performance. I used the same hyperparameters, learning rate, as I used in my base model, to directly compare the effect of dropout regularization. Table 2 shows the comparison of the performance values between my base model and dropout regularization done over the base model, with dropout probability equal to 0.1.

Model	Final Training Loss	F. Validation Loss	F. Test Loss	F. Validation Accuracy	F. Test Accuracy
Base	0.2811	0.3526	0.3822	87.63%	86.03%
Dropout Regularized Base Model	0.2392	0.3238	0.3514	88.23%	87.42%

Table 2: Performance Parameters of FCN of Base Model and Model using Dropout Regularization

The performance of our model is slightly affected with the same hyperparameters. At high dropout probability rates it even affected the model negatively, because of excessive regularization. And below 0.1 dropout probability, we won't see any meaningful changes in models performance, architecture. And with 0.1 dropout probability, we will see only slight improvement in accuracy and loss values. This is because our base model was already non-overfitting. The dropout regularization is used for preventing overfitting, which was already non-existent in our base model.

Next I Considered different Initializations , such as Random Normal, Xavier Normal, Kaiming (He) Uniform, using the exact same hyperparameters. Analyzing how these initializations affect the training process and the accuracy, the table 4 is summary, comparison of performance values obtained from models using different initialization.

Initialization	Training Loss	Validation Loss	Test Loss	Validation Accuracy	Test Accuracy	Comp. Time (m)
Default Initialization	0.2811	0.3526	0.3822	87.63%	86.03%	13.65
Random Normal	0.3416	0.3723	0.3864	86.46%	86.11%	14.74
Xavier Normal	0.3053	0.3225	0.3572	88.55%,	87.20%	14.82
Kaiming (He) Uniform	0.2280	0.3207	0.3450	88.25%,	87.33%	14.77

Table 3: Performance Parameters of FCN of Models using different Intilizations

The best testing accuracy was getten using Kaiming Uniform initialization. However Xavier Normal initialization has higher Validation Accuracy. Computation times are increased with respect to default initialization, but Xavier initialization and Kaiming initialization surely increased the performance of our model. Random Normal did also increase the performance however its effect was not as much as Xavier and Kaiming. Kaiming and Xavier introduced underfitting into our model, as it can be seen in their loss curves in fig (7).

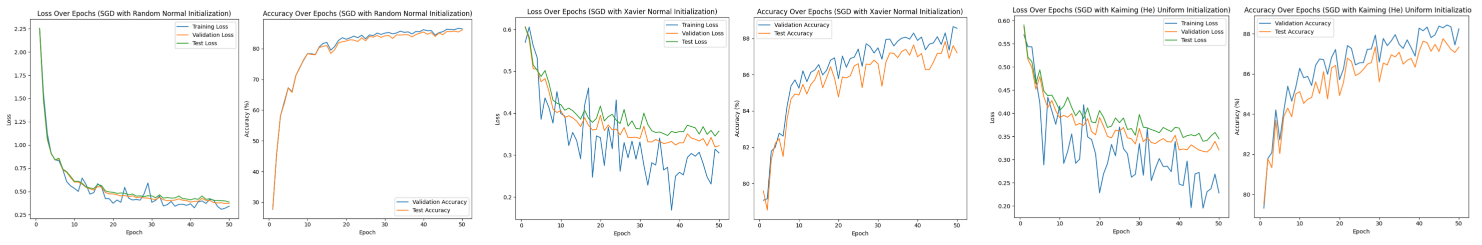

Figure 7: Loss and Accuracy Curves of Models Using Random Normal, Xavier Normal, Kaiming Uniform Intilizations Respectively

Then I Included Batch Normalization and discussed the effect of the normalization on the training process or testing accuracy, I needed to decrease the learning rate from 0.02 to 0.008 because Batch Normalization combined with high learning rate gave really sensitive and overfitting results, meaning the loss and accuracy curves had lots of zig zags, and after a certain epoch, the validation and testing loss started to increase, while training loss kept on converging. 0.008 was the highest learning rate I could use without making plots too sensitive.

Model	Final Training Loss	F. Validation Loss	F. Test Loss	F. Validation Accuracy	F. Test Accuracy	Computation Time (m)
Base	0.2811	0.3526	0.3822	87.63%	86.03%	13.65
Batch Normalized Base Model	0.2632	0.3127,	0.3462	88.84%	87.74%	14.71

Table 4: Performance Parameters of FCN of Models using different Intilizations

Batch normalization increased the performance of my base model, as it can be seen on Table 4l by normalizing inputs, which helps in mitigating issues related to internal covariate shift. Additionally, it acted as a regularizer, reducing sensitivity to weight initialization and contributing to improved generalization. It slightly increased accuracy values while decreasing losses. It introduced slight sensitivity and underfitting, visible on its loss curve.

I wanted to code an example with CNN, AlexNet, using the FashionMnist data set again. I optimized AlexNet with proper hyperparameters. CNN gave final test accuracy of %93.13 , Training Loss around in 10 epochs and 12.5 minutes. Considering we got the highest testing accuracy as %87.33 with Kaiming Initialization in my examples, the results from CNN were both faster and more accurate.

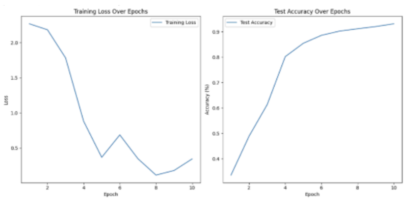

Figure 19: Loss and Accuracy Curves of AlexNet CNN model

Summary and Conclusion

We constructed a base model and hyperparameter-tuned it to achieve optimal performance, focusing on stable results, minimal computation time, high accuracy, low final loss value, and reduced zig-zags in accuracy and loss plots, as well as mitigating underfitting and overfitting characteristics. The Adam optimizer, with an appropriate learning rate, proved to be the most effective for our base model, even though it exhibited some underfitting characteristics. Next in performance was the RMSprop optimizer, followed by the SGD optimizer Dropout regularization did not significantly impact our model. Employing a small dropout probability of 0.1 for optimizing this hyperparameter resulted in a slight improvement in performance. This is attributed to the fact that our base model was inherently non-overfitting. Dropout regularization is typically employed to underfit data and serves as a preventive measure against overfitting.Out of 4 different initialization techniques Default, Random Normal, Xavier, Kaiming, The best testing accuracy was getten using Kaiming Uniform initialization. However Xavier Normal initialization has higher Validation Accuracy. Computation times are increased with respect to default initialization, but Xavier initialization and Kaiming initialization surely increased the performance of our model. Kaiming and Xavier introduced underfitting into our model, as it can be seen in their loss curves.Lastly Batch Normalization was helpful for increasing the performance of our model, however when it combined base learning rate, it gave really overfitting, sensitive results. Thus I had to tune and decrease my learning rate.

Acknowledgements

I would like to thank my classmates for discussions and Prof. Eli, TA's for notes, lectures, Office Hours, code examples

References

[1] Eli Shlizerman (2024). 582 KutzBook. AMATH 582 Wi 24: Computational Methods For Data Analysis.
[2] AMATH582 UW Spring 2023- 2024 Lecture Notes