HW4: IMAGE CLASSIFICATION WITH DEEP NEURAL NETWORKS

TIANBO ZHANG

University of Washington, Department of Applied Mathematics, Seattle

Abstract. This report investigates the application of Fully Connected Deep Neural Network for image classification in the FashionMNIST dataset. Through the implementation and iterative refinement of FCNs, including adjustments in network architecture, hyper-parameters, and regularization techniques, we aim to achieve high classification accuracy.

1. Introduction and Overview

Introduction: The FashionMNIST dataset is one of the most famous Classification challenges in machine learning. Unlike the simpler MNIST dataset, FashionMNIST provides a more challenging benchmark for machine learning algorithms, necessitating the development of powerful and nonlinear models. In this report we will delves into the utilization of Fully Connected Deep Neural Networks (FCNs) to this dataset. Our approach involves designing and optimizing FCNs with a focus on exploring the impact of various architectural decisions, hyperparameters, and regularization methods on the model's performance.

Overview: FashionMNIST dataset consists samples of 28×28 graysacale image. We have 60000 images for training and 10000 images for testing. And the images are clothing that belong to a class from one of 10 classes. We will implement the following steps to delve deeper into our research:

(1) Design the FCN model with adjustable hidden layers and nerusons in each layer with relu activation.
(2) Find a configuration of the FCN that trains with reasonable time, training loss curve, and testing accuracy.
(3) Perform hyper-parameter tuning.
(4) Perform hyper-parameter tuning of FCN to achieve testing accuracy of $> 90\%$ for FashionMNIST and $> 98\%$ for MNIST.

2. Theoretical Background

2.1. **Activation Function.** Activation functions are non-linear functions performed by neurons on each layer. Commonly used activation function is:

(1) ReLU - Rectified Linear Unit: $y \geq 0$
(2) Tanh: $-1 < y < 1$
(3) Sigmoid: $0 < y < 1$

2.2. **Cross Entropy Loss.** Cross entropy loss measures the performance of a classification model whose output is a probability value between 0 and 1. The cross entropy loss increases as the predicted probability diverges from the actual label.

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

2.3. **Optimizer.** An optimizer is an algorithm used to change the attributes of a neural network model, such as weights and learning rate, to reduce losses. There are several types of optimizer:

(1) Gradient Descent: A variation of gradient decent that updates all weights in the direction of the negative gradient of the loss function with small batches.
(2) RMSprop: Adjusts the learning rate adaptively for each parameter, making it smaller for updates associated with large gradients to prevent exploding and larger for updates associated with small gradients to prevent vanishing.
(3) Adam: Store an exponentially decaying average of past squared gradients, and also keeps an exponentially decaying average of past gradients.

2.4. **Over-fitting and Under-fitting.** The investigation of over-fitting and under-fitting is important for model training. If over-fit our model learns the training data too well. It captures noise and fluctuations in the training data where it negatively impacts the model's performance on new data. If under-fit our model is not utilizing the parameters enough to learn the underlying structure of the data. It can't capture the complexity of the data and result in poor performance for the model. We can look at the train/validation loss curve to find a solution.

2.5. **Back Propagation.** Back propagation is a gradient estimation method used to train neural network models. The gradient estimate is used by different optimization algorithm to compute the parameters. We have:

$$\nabla_{\hat{w}} L(\hat{y}, y) = \nabla_{\overrightarrow{w}} z \longleftarrow \frac{\partial \hat{y}}{\partial z}(z) \longleftarrow \frac{\partial L}{\partial \hat{y}}(\hat{y}, y)$$

2.6. **Dropout.** The concept of dropout is similar to L2 regularization. It effectively spreading the weights. Note it can be dependent on weight dimension.

$$\overrightarrow{a}_l^d = \overrightarrow{a}_l \times \overrightarrow{d}_l / (1 - p_d)$$

2.7. **Batch Normalization.** Batch normalization normalize the outputs of each layer. We have:

$$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$$

Then after normalization we have:

$$\mu = \frac{1}{m} \sum_{i=1}^{m} z^{(i)},\ \sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (z^{(i)} - \mu)^2,\ z_{norm}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

2.8. **Weight Initialization.** Weight initialization involve the concept of placing small random numbers into the weight. It can prevent vanishing/exploding gradients and also breaking symmetry. There are several different types of initialization. We are focusing on:Random Normal, Xavier Normal, Kaiming Uniform.

3. Algorithm Implementation and Development

3.1. **Model Setup.** There are several different parameters we need for our model. We will include the following parameters:

(1) Input/output dimension.

(2) A list of numbers indicating the number of neurons for each layer and the length of the list indicts the amount of hidden layers we have.
(3) Drop out rate (Set to 0 if don't want dropout).
(4) Boolean type holder to determine if we want to perform batch normalization.

Then we have the following constructor for our model:

Algorithm 1 ACAIGFCN Model

```
set up the first layer with the input dim and first element in the
neuron list
for i from 0 to the neuron list length minus 1 do
   if batch_norm is set to be true then
      add the batch normalization layer with ith neuron dimension
   end if
   add a drop out layer with drop out rate given
   add a linear layer with ith and i+1th neuron dimension
end for
add the last drop out and batch normalization layer
add the Last layer with last neuron and output dimension
set up the drop out function with dropout rate
```

Then we also have to implement a forward function for our model. The forward function will will pass the input through all the layers to generate the output. We will have the following algorithm:

Algorithm 2 Forwad function for ACAIGFCN Model

```
set output equal to input feature
for every layer except the last one do
   apply activation function to the layer
end for
return raw last layer output
```

3.2. **Weight Initialization.** We will have three different methods for initialization. Each method has different effect on the model. We have the following algorithm:

Algorithm 3 Weight Initialization

```
set output equal to input feature
if it is a linear layer then
   apply the initialization with the required method
end if
```

3.3. **Model Train and Validation.** During the training and validation process, we will loop it for **epochs** of time. For each epoch we will train the model with all the train batches and then validate the model with all the validation batches. We will record the train/valiation loss and validation accuracy for each epoch. We have the following algorithm:

Algorithm 4 Hyper-parameter Tuning

```
set up the holder for train/validation loss and validation
accuracy, also define the loss function
for every epoch do
   Apply activation function to the layer
   Apply dropout to the layer
   for every train batch do
      reshape the features and use it to train
      use loss function to calculate loss and record
      perform optimization and back propagation
   end for
   for every validation batch do
      reshape the features and use it to validate
      use loss function to calculate loss and record
      calculate the accuracy and record
   end for
   record the mean of batches of loss and accuracy
end for
Return the trained model
```

4. Computational Results

4.1. **Baseline Model Setup.** After implementing the basic structures of our neural network using Algorithm 1 and Algorithm 2 (with no drop out, no initialization, no batch normalization) we found our baseline model. We used gradient descent optimizer with a learning rate of 0.005 and our model has two hidden layers each with 200 neurons. The epochs we used is 30. Then we have the following result:

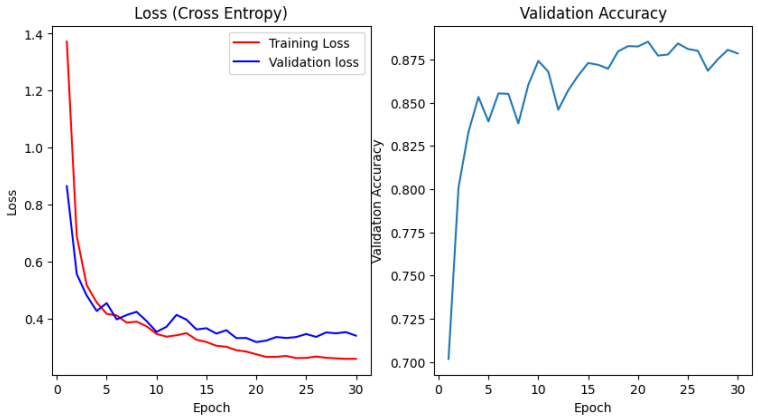

FIGURE 1. Baseline model results

The baseline model we get has an robust test accuracy above 85%. The highest validation accuracy we get is around 88%. And from Figure 1 we can see that the Loss curve seems reasonable. And after a few runs we found that the time is around 6 minutes. However, I noticed that the validation and train loss curve suggesting

there's a possibility of over-fitting for the data set. We will address this problem later in the report.

4.2. **Optimizer Determination.** After finding the baseline model for our dataset we start optimizing our model with the discussion around the differences between different optimizer for our model.

(1) **Learning Rate and Epochs:** After some experimentation, we noticed that the our baseline learning rate is too high for Adam and RMSprop optimizer. Especially for Adam optimizer, the validation accuracy dramatically dropped after the first epoch. An explanation for this is that the learning rate is too big for the optimizer where the direction of the gradient descent to go completely wrong. We also raised the amount of epochs to 30.
(2) **Layer and Neuron Dimensions:** We noticed that the baseline dimensions for our network is not the optimal parameter for the Adam and RMSprop optimizer. After some experimentation, we decided to use [100, 100] as our model dimension to solve the over-fitting problem observed.
(3) **Optimizer Selection:** After our analysis we found that with a learning rate of 0.0001 and epochs of 60 all three methods have a reasonable performance. Among all three optimizers, we found that the RMSprop has the highest validation accuracy and lowest loss.
(4) **Selection Explanation:**
    (a) Adaptive Learning Rates: RMSprop adjusts the learning rate for each weight individually, which means that the frequently updated weights will reduce their learning rate and also a boost. This allow the algorithm for more balanced convergence.
    (b) Mini-batch Learning: RMSprop is especially good for mini-batch learning since it compensates the noisy gradients from smaller sample size.
    (c) Vanishing Learning Rate: RMSprop uses moving average which prevents vanishing learning rate.
    (d) Sensitivity to Oscillations: RMSprop adjusts its learning to oscillations where it solves the problem of saddle points and local minima.

4.3. **Over-fitting and Under-fitting.** Proceeding out model analysis with the RMSprop optimizer, we noticed that the train and validation loss shows the possibility of over-fit. Hence we will apply drop out function to our layers. Then we have the following result:

Drop-out Rate	Train Loss	Validation Loss	Validation Acc	Test Acc
0	0.1999	0.3183	86.47%	85.7051%
0.15	0.3616	0.3238	87%	86.5840%
0.25	0.4183	0.3306	87.09%	86.2422%
0.33	0.4797	0.3438	88.7143%	87.7539%

Notice that with dropout regularization, out performance is improved by 1 to 2 percent. From figures in the Appendix we can see that where the validation loss curve cross over the train loss curve is no longer shown. Hence, dropout is indeed a helpful technique to solve our over-fit problem.

4.4. **Weight Initialization.** Weight initialization is another technique that is helpful for over-fitting problems. To see what kinds of effect for different weight initialization, we'll compare the results for three initialization methods with no drop out for RMSprop.

Weight	Train Loss	Validation Loss	Validation Acc	Test Acc
xavier normal	0.2910	0.3503	88.13%	85.6347%
random normal	2.1977	2.1695	15.73%	9.9218%
Kaiming Uniform	0.2999	0.3510	87.86%	86.0644%

As we can see from the table, random normal initialization performed really poorly for our dataset. Xavier normal initialization seems to perform a bit better than the Kaiming normal initialization.

4.5. **Batch Normalization.** Lastly, we investigate the effect of batch normalization on our dataset. We found that with batch normalization, the test accuracy is raised by 1 percent.

5. Summary and Conclusions

This report aimed to build a fully connected neural network for image classification on the FashionMNIST dataset. It explores various architectural decisions, hyper-parameters, and regularization methods to achieve high classification accuracy.

These outcomes not only enhance our understanding of image classification but also propose innovation approaches to prevent over-fitting and under-fitting problems. Techniques like dropout and weight initialization methods are analyzed for their impact on model performance, highlighting their importance in addressing over-fitting and improving model accuracy.

Conclusively, our investigation underscores the significance of RMSprop's superior performance in optimizing Fully Connected Deep Neural Networks for the FashionMNIST dataset. These outcomes not only enhance our understanding of effective neural network training techniques but also propose further exploration of convolutional layers, advanced regularization methods, and data augmentation for improving model robustness. Moving forward, it's imperative to delve into these areas, to bolster the efficiency and accuracy of neural networks in image classification tasks.

6. Acknowledgements

The author is thankful to Pro.Zhang from Beijing University of Post and Telecommunications for useful discussions about hyper-parameter tuning techniques and FCN structures.

7. References

References

[1] J.N. Kutz (2013) *Methods for Integrating Dynamics of Complex Systems and Big Data*, Oxford.

[2] Diederik P. Kingma, Jimmy Lei Ba (2015) *ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION*

8. Appendix (Extra Credit)

Optimizer Graphs:

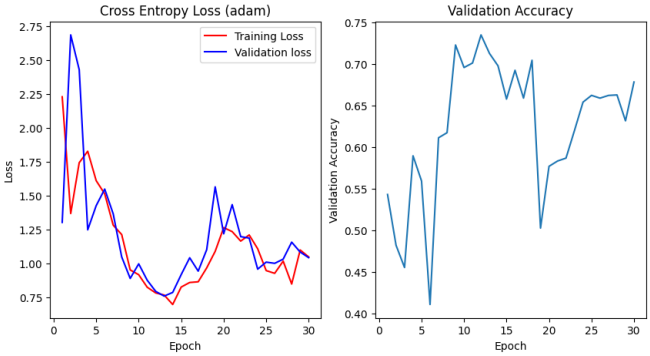

FIGURE 2. Adam Optimizer

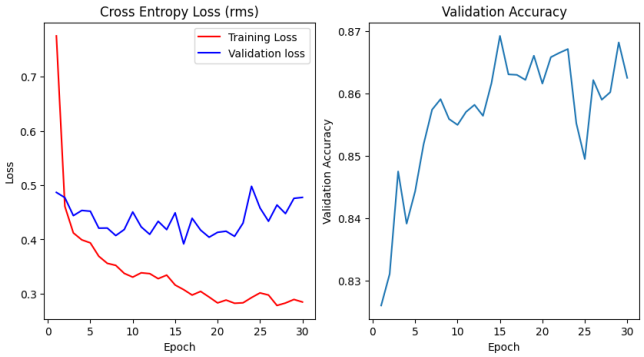

FIGURE 3. RMSprop Optimizer

Dropout Comparison Graphs:

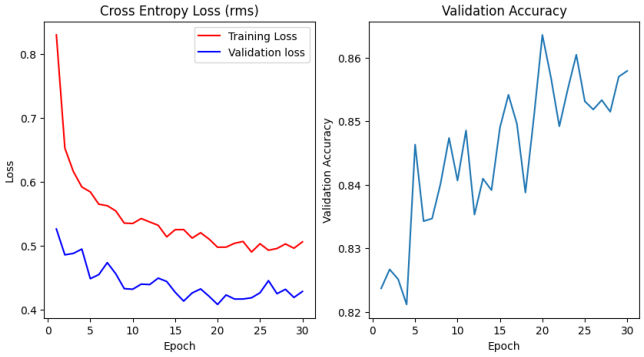

FIGURE 4. RMSprop Optimizer with 0.33 dropout

Weight Initialization:

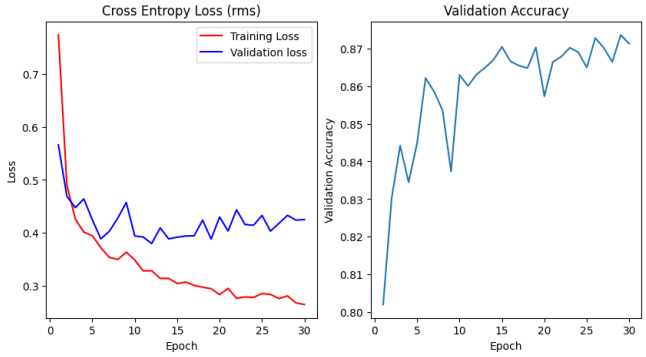

FIGURE 5. RMSprop Optimizer with xavier normal

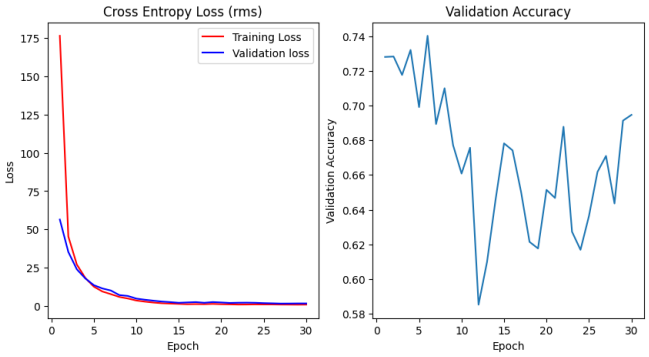

FIGURE 6. RMSprop Optimizer with random normal

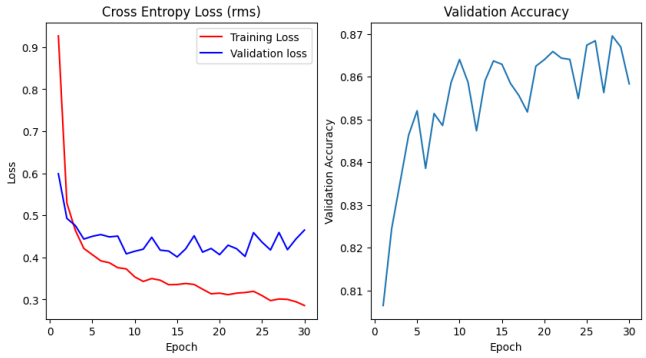

FIGURE 7. RMSprop Optimizer with kaiming uniform