PA2 Report: Neural Network Implementation for Classification on the Fashion MNIST dataset

Huimin Zeng
Computer Science & Engineering
UC San Diego
La Jolla, CA 92093
zenghuimin@yahoo.com

Zhenrui Yue
Computer Science & Engineering
UC San Diego
La Jolla, CA 92093
yuezrhb@gmail.com

Abstract

This document is the report of the first programming assignment (PA2) in CSE 253 Neural Networks/Pattern Recognition at UC San Diego in Winter 2020 by Huimin Zeng and Zhenrui Yue. The report is based on the requirements of the assignment and contains sections for the programming problems as well as the individual contributions of each team member in this group.

In this report, we first described the provided dataset: Fashion-MNIST, which contains 10 classes of 28x28 resolution images of fashion articles such as bags and sneakers. Then, we implemented a three-layer deep learning neural network with an input layer, a hidden layer and an output layer. We utilized softmax function as the output layer and cross-entropy loss to be the objective function, with the integrated derivation and backpropagation step in each component. After that, mini-batch stochastic gradient descent and momentum was introduced in the training process to reduce the training period, achieved a highest validation accuracy of 84.21% and test accuracy of 85.47%. Finally, we experimented with different regularization configurations, activation functions and modified the neural network structure to explore their influences on the model performance. In the last section, individual contributions to this assignment was briefly described.

1 Dataset and Preprocessing (3a)

Fashion-MNIST is a machine learning dataset of fashion items provided by Zalando Research, the dataset contains 10 classes of items such as coats and dresses, all in the form of 28x28 aligned grayscale images, it has a training set of 60,000 examples and a test set of 10,000 examples. With the same number of labels, data formats and split of training and test set, the Fashion-MNIST dataset is an updated version of MNIST dataset that could be used to train advanced machine learning models with modern computer vision tasks.

In the first task, the data was normalized simply by being divided by 255, since all pixel values in the dataset are in the range between 0 and 255 representing different grades in grayscale. Then, the data labels were also processed and transformed into an 2d array using one-hot encoding that was introduced in the last assignment. The last step before constructing the neural network was to shuffle the data and divide the training data into training and validation data with the ratio of 8:2.

2 Neural Network

We implemented a neural network with three layers for this multiclass classification task. Specifically speaking, the first layer of the network is namely the input layer, with 784 inputs representing all pixel values in one image. The second layer is the hidden layer with 100 perceptron units, fully connected to the first layer, the third layer is the output layer with 10 units, with each of them outputting a value for the probability of each class. The second and the third layer is also fully connected, with the sigmoid function as the default activation function to project outputs of each layer into a probability space, i.e. (0,1).

Before making predictions, the values of the output layer have to go through the softmax function to project their original values into comparable probability values. With y_k^n representing the probability of the nth data point in class k, the softmax function computes probabilities of the data in all possible classes and pick the class with highest:

$$y_k^n = \frac{\exp{(a_k^n)}}{\sum_{k'} \exp{(a_{k'}^n)}}$$

Well is known that it is of little efficiency to compute the Mean Squared Error (MSE) for the classification problem, since there might be relatively less information for the update step. Therefore, cross-entropy loss is regarded as better loss function in the multiclass classification problem. Please see the formal definition of the cross-entropy below:

$$E(w) = -\frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{m} t_i^n \ln(y_i^n)$$

2.1 Gradient Computation (3b)

In this section, we chose some weights in the neural network and computed their slope using the following formula:

$$\frac{\partial E(w)}{\partial w} \approx \frac{E(w+\epsilon) - E(w-\epsilon)}{2\epsilon}$$

We selected some weights w, then increment these weights by ϵ and conduct a forward pass. We compute the losses and with the above equation, the gradients were obtained. The corresponding values are listed below.

	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6	Class 7	Class 8	Class 9	Class 10
Output Layer Weight Gradient	1.37E-08	-1.94E-09	3.92E-08	-1.01E-07	-2.82E-09	-8.18E-12	-1.03E-07	3.41E-13	7.61E-08	-1.44E-10
Hidden Layer Weight Gradient	-1.20E-11	1.39E-15	-1.41E-07	8.96E-16	1.07E-14	-7.44E-08	1.42E-15	5.78E-14	-3.26E-15	4.88E-09
Input Layer Weight Gradient	0	0	0	0	0	0	0	0	0	0
Output Layer Bias Gradient	8.50E-08	2.37E-09	1.52E-07	1.40E-07	3.36E-08	1.79E-08	1.04E-07	1.17E-07	1.56E-07	1.49E-07
Hidden Layer Bias Gradient	8.55E-08	4.55E-08	1.41E-07	3.28E-08	1.28E-07	9.45E-08	1.97E-09	4.58E-08	1.51E-08	2.06E-08
Input Layer Bias Gradient	4.21E-08	1.69E-14	7.88E-11	1.99E-14	2.69E-13	5.52E-08	1.09E-13	5.18E-09	2.50E-10	2.36E-08

Table 1: Error values on Happiness vs Anger (Resized) in 100 Epochs

2.2 Training the Neural Network (3c)

In this part, we start to use the training set to train the neural network. The forward step is defined within each class depending on if the class is activation or a real layer. This means that the neural network has 5 components altogether, with 3 layers: the input layer, layer of hidden units and the output layer as well as 2 activation components each respectively assigned after the first two layers. The layer component would take the inputs and multiply the inputs with the weights and plus the bias terms, whereas the activation component would take this value and pass it to the activation function, and compute the inputs for the next layer, see formulas below:

$$Layer\ Component: a = w^T x + b$$

$$Activation\ Component: z = h(a)$$

For the forward step, we start from the input layer and use the formula above to compute corresponding values, then process the output values with softmax function and get the cross-entropy loss. For backpropagation (backward) step, we start from the output layer and go back layer by layer, for activation components, this will be the gradient of the activation function multiplied with the corresponding values. For layers however, we will first have to compute the gradients of the cross-entropy loss with respect to weights to begin with, here we use the results from the individual part of (2c) like below:

$$\frac{\partial E}{\partial \boldsymbol{W_k}} = -\frac{1}{y_C}\frac{\partial y_C}{\partial \boldsymbol{a_k}}\boldsymbol{z_j}$$

To speed up the training process, we also implemented momentum for the mini-batch stochastic gradient descent process, mini-batch is a commonly used gradient descent process, which randomly split the training dataset into multiple mini-batches, and utilize each mini-batch to compute gradients and update weights, this ensures a much faster and less random convergence process. Momentum is a method that helps speed up the convergence by increasing the step size when the gradient direction is consistent and decreasing it in the other case, please see the formula below:

$$\Delta W_t = V_t = \alpha V_{t-1} - \varepsilon \frac{\partial E}{\partial w}(t)$$

Here α is the coefficient of momentum, ε the learning rate, with V representing the current velocity and ΔW_t the weight change. The minus term would decrease once the gradient of the loss function is stabilized, meaning the gradient direction is consistent. The above term together would be the weight change of every update, with the momentum coefficient equals 0.9, the training epochs to an accuracy of around 80% reduced from ca. 20 epochs to around 10 epochs. After tuning the training parameter, we were able to achieve a highest training accuracy of 82.13% and validation accuracy 83.55% in around 20 epochs. If we further increase the epoch number to 50, the test accuracy would eventually hit 85%. Please see the chart below for accuracy and loss values with increasing epoch number.

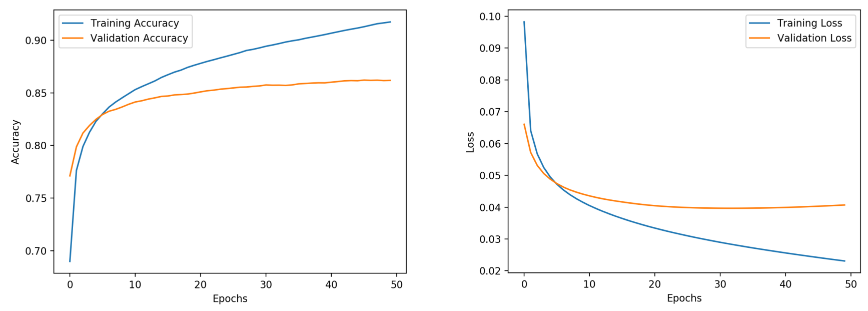

Figure 2: Accuracy and Loss Values in different Epochs

This chart above is the result of our model performance in 50 epochs of training, as we see here, the training accuracy has been steadily increasing from circa 70% of the first epoch to over 90% in the 50th epoch, with the stagnated values of validation accuracy and validation loss, which is a clear sign of overfitting. The optimal model parameter would be achieved around 30-40 epochs, with a highest test accuracy of 85.24%. Since the training accuracy keeps increasing after the optimal model complexity, it is necessary to implement an early-stop mechanism during the training process, so that the neural network could reach its optimal configuration for the test set.

3 Experiments with the Neural Network (see code)

After implementing and tuning the neural network for image recognition, we tried to add more useful features to the network, so that the deep learning model could converge faster or reach a higher accuracy value without overfitting. We first introduced an L2 penalty term into the loss function in order to constraint the weights, then used different activation functions and observed the changes of the model's performance. Last but not least, we also experimented modified and different network structures to search for an ideal neural network model for this task.

3.1 Experiments with Regularization (3d)

Regularization is a machine learning technique that prevents the model parameter from increasing too fast and avoids overfitting, specifically, regularization constraints the model coefficients from increasing using an additional L2 norm in the loss function, so that the new loss functions could be formulated as:

$$E(w) = -\frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{m} t_i^n \ln(y_i^n) + \frac{\lambda}{2}\sum \|w_j\|^2$$

The left term is the same as the original loss function, namely the cross-entropy loss, while the right-hand side is the summation of all squared weights, multiplied with the λ term that controls the penalty. We see that as the model weights getting greater, so will the entire loss term which eventually penalizes the weight values. Regularization is widely used method in neural networks that helps the model from overfitting, as the penalty term discourages the network from learning of noises and forming a complex model. However, the penalty term would also cause slower convergence due to this L2 term, therefore, it could take longer for the same model with regularization to reach the same accuracy. In this sense, we tried out various coefficients for the penalty term and recorded the accuracy and loss values of each epoch, please refer to the following chart. Note that the training epochs were increased to 60 to ensure convergence.

In the following charts, we tried out four different penalty terms, from 0.0001, 0.001 to 0.01 and 0.1. The results are visually represented below, here the left side represents lower penalty terms and the right side higher.

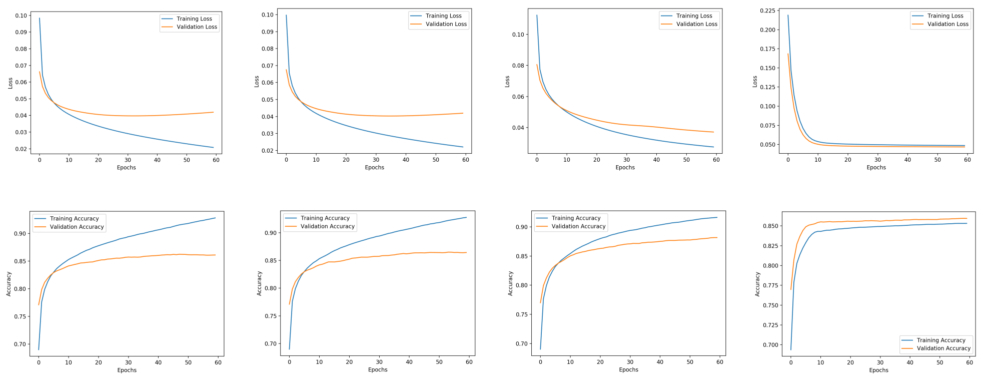

Figure 3: Loss and Accuracy Values of different Penalty Term in 60 Epochs

As we see from the chart, an optimal λ value would be around 0.01, as the training of neural networks converges in reasonable number of epochs, the loss curve keeps decreasing and the accuracy value eventually surpasses the other penalty values. As the penalty keeps increasing, the loss function converges very fast and stabilizes around 0.05, but its accuracy value was also constraint by the oversized penalty term to only around 85% by the end of the 60th epoch. Overall, a penalty term does help restricting the model from getting too complex and overfits data, in our case, the best λ is about 0.01 and helps the neural network further decrease the loss and reaches a test accuracy of 87.14%.

3.2 Experiments with Activations (3e)

So far, we have been using sigmoid function as our activation function in the neural network, sigmoid function is a very common activation function in the field of deep learning. However, there are other options which might work better in our case, such as the tanh and the ReLU (Rectified Linear Unit) activation function, three activation functions and their plots are posted below:

$$ReLU: f(x) = \max(0, x)$$

$$Sigmoid: \sigma(x) = \frac{1}{1 + e^{-x}}$$

$$Tanh: tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$

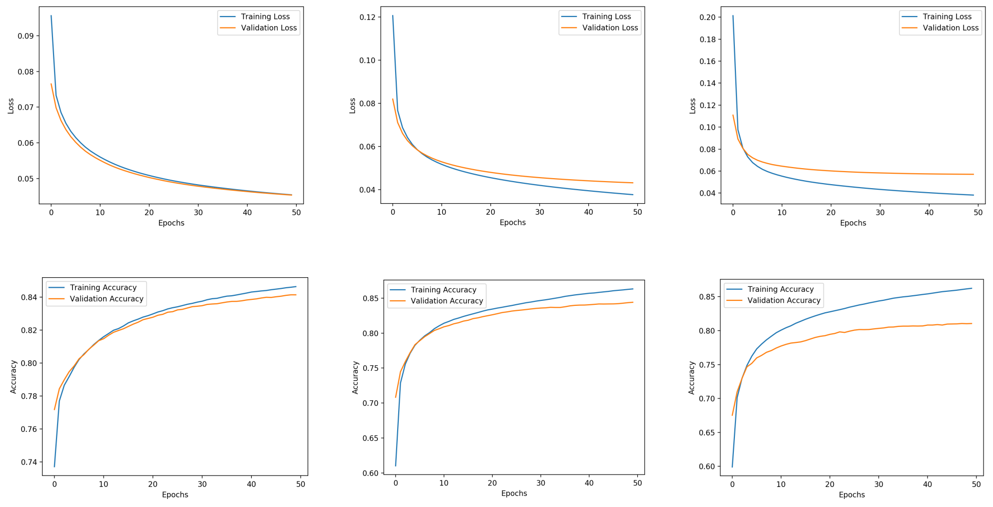

Figure 4: Performances of Activation Functions ReLU, Sigmoid, Tanh

We implemented all above activation functions and made used of all three activation functions, with all other model parameter staying unchanged. In this case, we used a uniform configuration of the neural network model, and only changed the activation function each time to measure the performance changes. The performances in 50 epochs of all three alternatives are listed above, with the first row representing the loss values and the second row the accuracy values. As we see in the plots, the sigmoid activations function has the best convergence speed and a quick drop of loss value, whiel the other two activation functions have either a slower convergence process or an unsatisfactory performance on the validation set in 50 epochs. Thus, the sigmoid function could be better utilized for the task of image classification.

3.3 Experiments with Network Topology (6c)

We have experimented with different regularization configuration and activation functions, now we explore the possibilities of the neural network itself and test the model performance with a modified structure. Originally, we have a three-layer neural network with one input layer, one hidden layer of 100 perceptron units and one output layer. In this section, we will first reduce or double the hidden units (200 in total) and test its performance. Then, another layer of hidden units would be added to the original model, however with same number of hidden units (50 hidden units each layer). The performances of the modified model would be tested and compared to the first model. After experimenting the neural network, we plotted the performances of each modified network below.

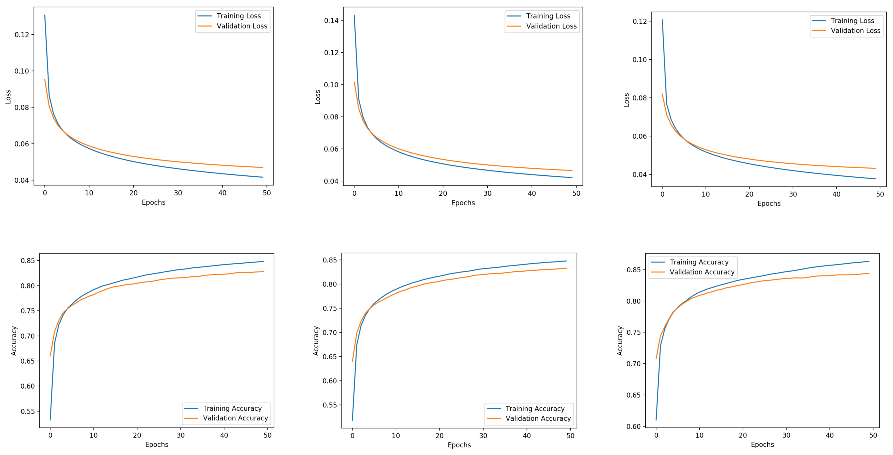

Figure 5: Performances of Modified Neural Networks, Left: Original Neural Network, Middle: Network with half Hidden Units, Right: Network with doubled (200) Hidden Units

The left side of the charts are the original network, with an input layer of 784 units, a hidden layer of 100 units and an output layer of 10 units. The network in the middle represents the same network structure, only with 50 units in the hidden layer, while the charts on the right-hand side have doubled perceptrons (200 in total) in the hidden layer. Compared to the original model, the network with doubled hidden units performs generally better and has a higher test set accuracy of 83.63% compared to 82.80% of the original model after 50 epochs training, but it could also overfit the data to some extent, as the training loss further decreases and enlarges the gap between the training and validation loss after 30 epochs, therefore, it could be of help to apply regularization technique to this model and avoids overfitting. On the other side, the network with less hidden units (Middle) performs worse than the other two models, this could be traced back to its limited perceptron units, especially where the inputs of 784 dimensions were reduced to only 50 dimensions, which causes certain information loss, certain features of the original image could also be lost in this process. Besides, the total weights (edges) in the network are also significantly reduced compared to a network but with more perceptron units, thus this model has less flexibility and tends to underfit the data.

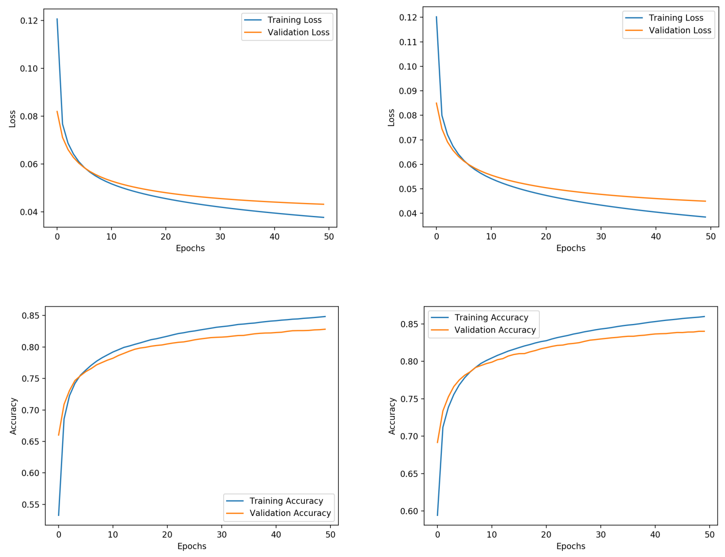

Figure 6: Performances of Modified Neural Networks, Left: Original Neural Network,

Right: With 2 hidden Layers

Next, we tried out different layer numbers and their influences on the model's performance. To validate the comparison between different models, we let the total number of weights and biases to be the same, so that the total model flexibility is comparable. On the left side of the charts, we have the original model whereas the right side is the model with the doubled hidden layer. Compared to the original neural network, the model with two hidden layers has very similar (or slightly better) performances, the network with two hidden layers has a slightly better test set accuracy of 82.96% against 82.81% of the original model. The behavior on convergence and prediction is also very similar, the reason for this similarity could be the comparable number of parameter and consequently similar model complexity.

4 Individual Contributions

This part included the individual contributions of the two team members Huimin Zeng and Zhenrui Yue in the programming assignment.

4.1 Huimin Zeng

I implemented the entire neural network framework with the layer and activation components as well as the forward and backward steps, I also implemented the training function with momentum and mini-batch stochastic gradient descent. Then I tested the neural network and eliminated some problems and bugs of this programming assignment, I also wrote some part of this report.

4.2 Zhenrui Yue

I individually implemented some parts of the neural network including the activation functions, softmax function and cross-entropy loss, moreover, I also wrote the forward and backward functions and was responsible for debugging and running tests to debug and optimize the neural network and its performances. I implemented some tests to generate tables and charts and wrote the most parts of this report.