Image Classification Based On Three-Layer Neural Network

Xiang Zheng, 21307110169

School of Data Science, Fudan University

Abstract—This project focuses on developing and training a neural network to accurately classify grayscale fashion items in the Fashion-MNIST dataset. The neural network architecture consists of three linear layers, with a thorough exploration of activation functions and loss functions to optimize performance. The report covers dataset acquisition, architectural decisions, training procedures, and evaluation metrics, underscoring the importance of meticulous design and optimization in neural network development for image classification tasks. The trained model achieves a training accuracy of 94.8% and a testing accuracy of 89.6%.

keywords—*Fashion-MNIST, image classification, neural network*

1. Introduction

In this project, our goal is to train a fully connected neural network to classify images from the Fashion-MNIST dataset. Curated by Zalando, this dataset contains grayscale images of various fashion items, along with corresponding category labels. With 60,000 training examples and 10,000 test examples, Fashion-MNIST serves as a standard benchmark for fashion image classification.

Our neural network consists of three linear layers, trained to learn meaningful representations and make accurate predictions. We explore various components like activation functions and training strategies to optimize performance.

The remainder of this report is structured as follows: Section 2 provides the dataset acquisition and preprocessing method. Section 3 delves into the specifics of the neural network architecture, detailing the design choices for its components. Section 4 outlines the training process, including the selection of hyperparameters and evaluation metrics. In Section 5, we evaluate the trained model on the test dataset and present visualizations to assess its effectiveness. Finally, Section 6 discusses potential avenues for future improvement.

Note: You can find the project files in the GitHub repo and the model parameters in the Google Drive. For instructions on how to train and test the neural network, please refer to the readme file. You can also directly preview the project files in this website.

2. Dataset

Having already covered the dataset's characteristics earlier, we'll now delve straight into the downloading and preprocessing steps. Utilizing the Python library `urllib`, we'll fetch the data from its official repo and organize it into a dictionary containing images and labels. Each numpy array representing a grayscale image will be reshaped into the format $(1, 28, 28)$ to maintain consistency with their original shapes.

3. Model

A standard linear layer neural network comprises three main components: the input layer, hidden layers, and output layer. The transition between the input (hidden) layer and hidden layers can be described as a `LinearActivation` layer, while the transition between the hidden layer and output layer involves a `Linear` layer along with a `loss` function. Other parameters like `reg` and `weight_scale` also influence the performance of the model.

Before delving into the discussion of our model, it's essential to establish the key notations utilized throughout this report. Let N represent the batch size, indicating the number of samples processed in each training or inference iteration. D denotes the flattened dimension of the image, symbolizing the size of the image vector after flattening. $H_1, H_2, \cdots, H_n$ refer to the dimensions of the hidden layers within the neural network architecture. Finally, C signifies the output dimension, representing the number of classes in the classification task.

3.1. Linear Layer

Consider the first linear layer as an illustration. The forward pass can be concisely expressed as:

$$\mathbf{Y} = \mathbf{X}\mathbf{W} + \mathbf{b} \tag{3.1.1}$$

Here, $\mathbf{Y} \in \mathbb{R}^{N\times H_1}$, $\mathbf{X} \in \mathbb{R}^{N\times D}$, $\mathbf{W} \in \mathbb{R}^{D\times H_1}$, and $\mathbf{b} \in \mathbb{R}^{H_1}$. It's worth noting that `numpy` broadcasts $\mathbf{b}$ to fit into the computation. Hence, we represent $\mathbf{b}$ as a vector to maintain alignment with my code implementation. Also, please note that we need to 'cache' the current values of $\mathbf{X}$ and $\mathbf{W}$ for the backward pass.

Regarding the backward pass, the input of the backward function is `dout` and `cache`, where `dout` represents the gradient at $\mathbf{Y}$ with shape (N, H_1). Taking derivatives with respect to $\mathbf{X}$ and $\mathbf{W}$, we obtain:

$$\mathrm{d}\mathbf{X} = \mathrm{d}\mathbf{Y} \cdot \mathbf{W}^\top \tag{3.1.2}$$
$$\mathrm{d}\mathbf{W} = \mathbf{X}^\top \cdot \mathrm{d}\mathbf{Y} \tag{3.1.3}$$

For bias $\mathbf{b}$, the derivative involves summing `dout` over the first dimension.

3.2. Activation Functions

When it comes to activation functions, they all perform an element-wise operation on the given matrix. There are several commonly used types of activation functions, including `ReLU`, `Tanh`, and `Sigmoid`. Let's introduce each of them individually.

3.2.1. ReLU

For the forward pass, ReLU is expressed as:

$$\mathbf{Y}' = \max(\mathbf{Y}, \mathbf{0}) \tag{3.2.1}$$

For the backward pass, it's:

$$\mathrm{d}\mathbf{Y} = \max(\mathrm{d}\mathbf{Y}', \mathbf{0}) \tag{3.2.2}$$

3.2.2. Tanh

The mathematical expression of the hyperbolic tangent function, Tanh, is:

$$\tanh x = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{3.2.3}$$

Taking the derivative with respect to x, we get:

$$\frac{\partial \tanh x}{\partial x} = \frac{4}{(e^{2x}+1)^2} = 1 - \tanh^2 x \tag{3.2.4}$$

The forward and backward pass follows directly from the above equations:

$$\mathbf{Y}' = \tanh \mathbf{Y} \tag{3.2.5}$$

$$d\mathbf{Y} = d\mathbf{Y}' \cdot (1 - \tanh^2 \mathbf{Y}) \tag{3.2.6}$$

3.2.3. Sigmoid

Let's denote the sigmoid function as:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{3.2.7}$$

Taking the derivative with respect to x, we obtain:

$$\frac{\partial \sigma(x)}{\partial x} = \frac{e^{-x}}{(1+e^{-x})^2} = \sigma(x)(1-\sigma(x)) \tag{3.2.8}$$

Then, the forward and backward pass of the sigmoid function are:

$$\mathbf{Y}' = \sigma(\mathbf{Y}) \tag{3.2.9}$$

$$d\mathbf{Y} = d\mathbf{Y}' \cdot \sigma(\mathbf{Y})(1-\sigma(\mathbf{Y})) \tag{3.2.10}$$

3.3. Loss Function

The loss function plays a crucial role in training a neural network as it quantifies the discrepancy between the predicted output and the actual target. Here, we'll implement the `cross_entrophy`. Since this is a multi-class classification task, the CE is slightly different from the binary case.

For traditional binary cross entropy, the loss function is

$$\text{loss}(\mathbf{y}, \hat{\mathbf{y}}) = -\frac{1}{N}\sum_{i=1}^{n} y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i) \tag{3.3.1}$$

where y_i can only take the value of 0 or 1. However, this is not appropriate in multi-class classification.

To compute the loss function in the forward pass of the neural network, we'll first apply the softmax function to the raw output scores to obtain the predicted probabilities for each class. The softmax function is defined as:

$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{C} e^{x_j}} \tag{3.3.2}$$

where $\mathbf{x}$ is the vector of raw output scores, and C is the number of classes. This function ensures that the predicted probabilities are normalized and sum up to 1 for each sample. Besides, the 'shifted x' method is used here to avoid overflow.

Once the softmax probabilities are computed, the cross-entropy loss can be calculated using the true labels and the predicted probabilities. The cross-entropy loss is defined as:

$$\text{CE} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{s=1}^{C} t_{i,s}\log(f(s)_i) \tag{3.3.3}$$

where N is the number of samples, C is the number of classes, $t_{i,s}$ is the true label for the i-th sample and s-th class (which is 1 if the sample belongs to class s and 0 otherwise), and $f(s)_i$ is the predicted probability for the i-th sample belonging to the s-th class. Hence the loss can be further simplified as

$$\text{CE} = -\frac{1}{N}\sum_{i=1}^{N} t_i \log(f(s)_i) \tag{3.3.4}$$

In the backward pass, we'll need to compute the gradient of the loss function with respect to the raw output scores. This involves subtracting the true labels from the softmax probabilities, resulting in the gradient matrix. The gradient can be calculated as:

$$\frac{\partial \text{CE}}{\partial \mathbf{x}} = \frac{1}{N}(\text{softmax}(\mathbf{x}) - \mathbf{t}) \tag{3.3.5}$$

and it's implemented in Python code as

```
# probs denotes the softmax result
probs = np.exp(log_probs)
loss = -np.sum(log_probs[np.arange(N), y])
loss /= N
dx = np.copy(probs)
dx[np.arange(x), y] -= 1
dx /= N
```

3.4. Other Parameters

In addition to the model architecture, such as linear and activation layers, neural networks often involve hyperparameters such as `hidden_dims`, `reg`, and `weight_scale`.

For `hidden_dims` and `reg`, they determine the size of the hidden layers and serve as a regularization parameter that helps prevent overfitting by penalizing large weights in the model respectively. In the next section, we'll discuss how grid search is used to find the optimal values of them.

For `weight_scale`, the initial scale of the weight matrix, a commonly recommended value is 0.01. This value provides a good starting point for weight initialization, ensuring that the network begins training with reasonable weights.

4. Train

Now that we've covered the details of the Fully Connected Neural Network (FCNN) model in the previous section, let's delve into the training procedures. We'll begin by introducing some update rules, followed by a sanity check of the `Solver` on a small dataset to ensure its functionality. Subsequently, we'll conduct training for 2000 iterations for each configuration to identify the best configuration. Finally, the model will undergo training based on the optimal parameters for a significantly larger number of iterations to achieve the best results.

By the way, unless otherwise specified, the parameters used in this section are set to their default values, as detailed in Table 1.

Parameter Name	Default Value
hidden_dims	[128, 64]
activation	[relu, relu]
reg	$1e^{-3}$
update_rule	sgd
learning_rate	$5e^{-3}$
lr_decay	0.9

Table 1. Default value of parameters (partial listing)

4.1. Update Rule

For update rules, we have provide four commonly used methods: stochastic gradient descent (SGD), SGD with momentum,

Adam, and RMSprop. The mathematical explanation and some intuitions will be given below, while the experiment results of comparison of these four update rules will be defer to section 4.3.

4.1.1. SGD

The vanilla stochastic gradient descent is

$$\mathbf{w} = \mathbf{w} - \texttt{lr} \cdot \mathrm{d}\mathbf{w} \tag{4.1.1}$$

where the default value of `lr`(learning rate) is $1e^{-3}$.

4.1.2. SGD Momentum

For stochastic gradient descent with momentum, the update rule is

$$\mathbf{v} = \texttt{momentum} \cdot \mathbf{v} - \texttt{lr} \cdot \mathrm{d}\mathbf{w} \tag{4.1.2}$$

$$\mathbf{w} = \mathbf{w} + \mathbf{v} \tag{4.1.3}$$

where $\mathbf{v}$ denotes the velocity. The empirical default values for `lr` and `momentum` are $1e^{-3}$ and 0.9, respectively.

The advantages of SGD with momentum includes: Momentum is faster than stochastic gradient descent the training will be faster than SGD. Local minima can be an escape and reach global minima due to the momentum involved. They'll be validated in the following experiments.

4.1.3. Adam & RMSprop

Adam and RMSprop can be viewed as applying first-order and second-order normalization on the weight vector $\mathbf{w}$ respectively. These update rules are more complex compared to the ones discussed above. For a detailed explanation, please refer to the code implementation in `optimization.py`.

4.2. Sanity Check

For the sanity check, we trained the model using the first 500 samples of the training dataset for 2000 iterations. We set the hidden dimensions as $[128, 64]$ and used the `ReLU` activation function. The training results are depicted in Figure 1:

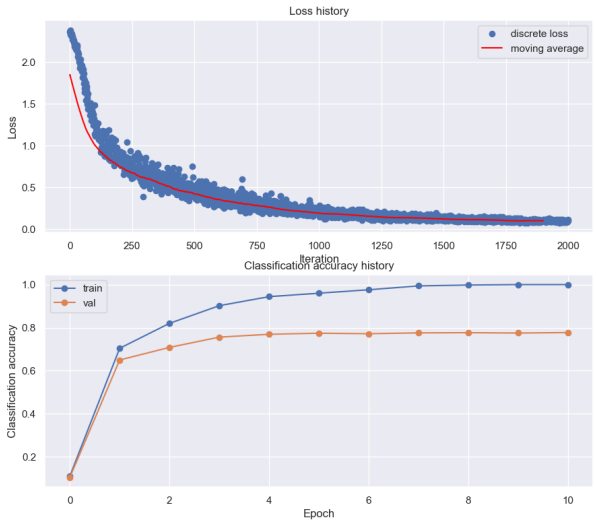

Figure 1. Training on the Small Dataset

After 9 epochs of training (approximately 1800 iterations), the model achieved perfect accuracy on the training dataset, while the validation (testing) accuracy plateaued around 77.5%. This indicates that our model is capable of overfitting the small subset of training data well, ensuring its availability for use.

Now, let's proceed to fine-tune the model using the entire training dataset to find the best parameters.

4.3. Search Best Configuration

The process of searching the best configuration is divided into three subsections. First, the model will undergo training for 2000 iterations using 9 different combinations of activation functions while keeping other parameters constant to find the best combination, which can be seen as a greedy search. Subsequently, we will implement a grid search method to find the best hyperparameters `reg`, `learning_rate`, and `hidden_dims`. Finally, we will experiment with 4 different update rules mentioned in section 4.1.2 and compare their performance.

4.3.1. Combinations Of Activation Functions

As discussed in Section 3.2, we offer three commonly used activation functions: `ReLU`, `Tanh`, and `Sigmoid`. In a three-layer neural network, there are two layers for activation functions, leading to 9 different combinations. The results of the experiments are presented in Figure 2 and Figure 3.

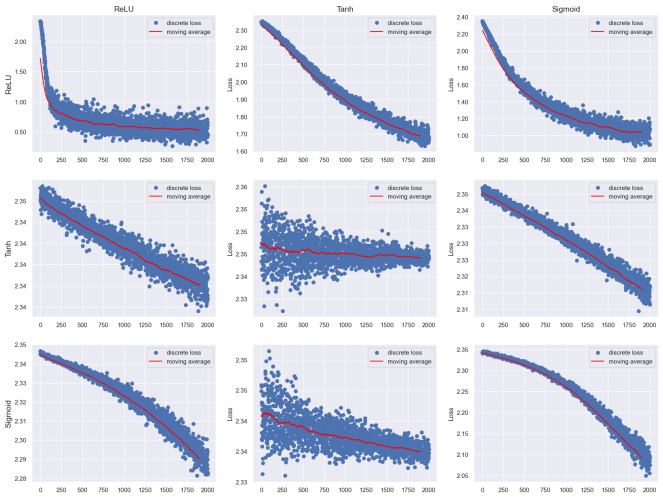

Figure 2. Loss on different combinations of activation functions

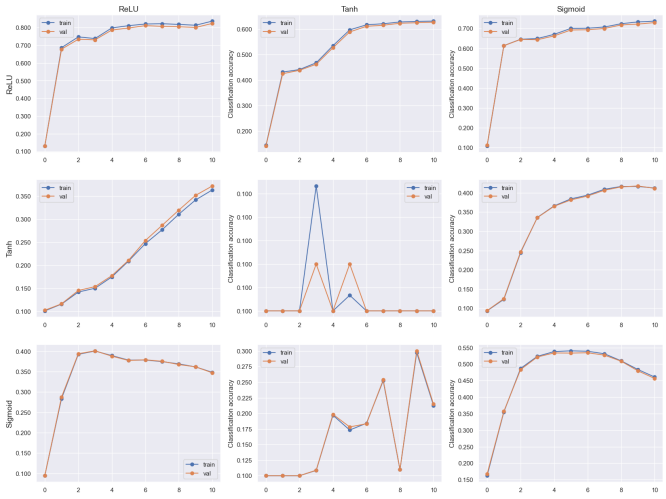

Figure 3. Accuracy on different combinations of activation functions

Based on the results, we observe that the top three performing combinations are ['ReLU', 'ReLU'], followed by ['ReLU',

'Sigmoid'], and then ['ReLU', 'Tanh']. Notably, for combinations other than these top three, their loss remains consistently above 2 even after 2000 iterations. The superiority of `ReLU` over other activation functions is evident, likely attributed to mitigating the gradient vanishing problem present in both `Tanh` and `Sigmoid` functions.

Furthermore, it's noteworthy that when `ReLU` is exclusively applied in the second layer, performance degradation occurs. This observation suggests neural network saturation after the first activation layer if it's not implemented with `ReLU`. Consequently, we will proceed with the combination ['ReLU', 'ReLU'] in the subsequent sections.

4.3.2. Tune Hyperparameters

We will now employ a grid search method to fine-tune the parameters `hidden_dims`, `learning_rate`, and `reg`. Given a training dataset size of 60,000 with 10 labels, and to ensure generality and computational feasibility, we'll explore combinations of hidden dimensions such as 128, 64, and 48. For the remaining parameters, we'll adopt empirically optimal values. The results of the grid search are presented in the Table 3 in Appendix A.

Upon examining the table, we observe that `hidden_dims` and `reg` have slightly less impact on the loss compared to `learning_rate`. The optimal learning rate may fall around $1e^{-2}$ since their validation accuracy are the highest on average.

To further fine-tune these parameters, we'll utilize a random search. For simplicity of visualization, we'll set the hidden dimensions as [128, 64], randomly sample regularization values from $[1e^{-2.5}, 1e^{-2}]$, and learning rates from $[1e^{-3}, 1e^{-1.5}]$. The optimal parameters found in the former grid search are also added into the searching range of configuration. The precise search results are listed in the Table 4 in Appendix B and visualized in the Figure 4.

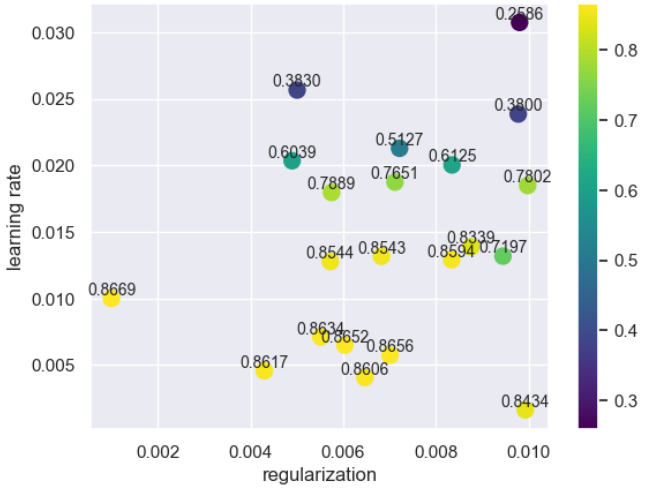

Figure 4. Scatter plot of random search results

Combining the results from both the table and figure, it's evident that the validation accuracy increases as the learning rate increases from $1e^{-3}$ to approximately $1e^{-2}$, but decreases rapidly beyond this point. Hence, it's inferred that the optimal learning rate should be chosen around $1e^{-2}$.

To further substantiate the selection of the learning rate, we will fix the `reg` parameter at its optimal value and plot the loss history of neural networks with different manually selected learning rates. This comparison is depicted in Figure 5. It is observable that a lower learning rate (0.0001) may result in a slower decay rate of the loss. Conversely, a high learning rate (0.03) could lead to escaping the saddle point of the loss function and increasing the loss as training progresses. Additionally, there are cases where a high learning rate may cause the loss to decrease too rapidly initially but plateau after several hundred iterations, maintaining a high loss in the end. Unfortunately, this phenomenon did not occur in our training process, which could be attributed to the dataset specifications and the choice of evaluation metrics beforehand.

In the end, the optimal parameters found after the two-phase search are `hidden_dims` = [128, 64], `reg` = 0.001, and `learning_rate` = 0.01.

4.3.3. Applying Different Update Rules

After obtaining the best parameters from the aforementioned procedures, we proceed to apply four different update rules, each with 2000 iterations. The loss history for each update rule is illustrated in Figure 6.

Clearly, the update rules, excluding SGD, demonstrate substantial enhancements, resulting in a hastened decay of the loss function.

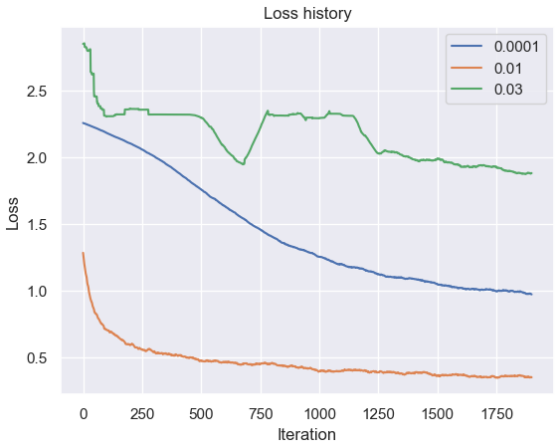

Figure 5. Loss history for different learning rates

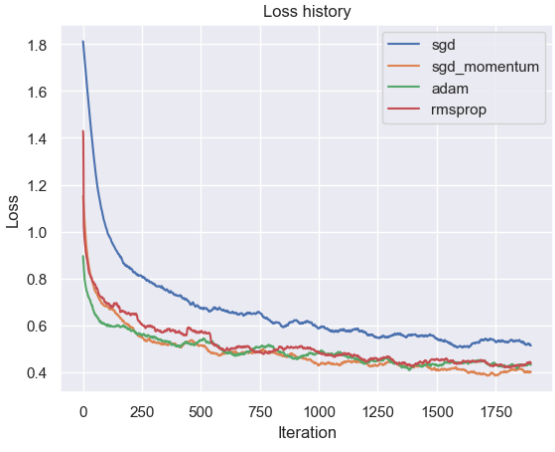

Figure 6. Loss history of different update rules

4.4. Training on Full Dataset

Now that we have obtained the optimal hyperparameters, let's train our model on the full training dataset using these parameters. We'll run the training for 30,000 iterations, with other parameters set to their default values as specified in the comments for the `Solver` and `FullConnectNet` classes. Besides, the loss history and accuracy history during the training process are visualized in the accompanying Figure 7. The training accuracy of the model reaches 94.8% upon completion of the training process.

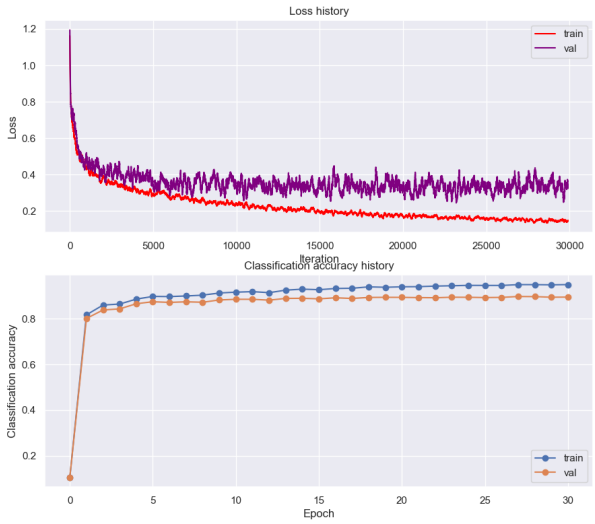

Figure 7. Training on the Full Dataset

Once the training is completed, we save the trained model as a `.npz` file in the path `./model/fcnn.npz`.

5. Load & Test

Having successfully trained the three-layer neural network and saved the weight parameters, we now proceed to load the trained model and evaluate its performance on the testing dataset. The accuracy table is presented in Table 2 and the confusion matrix is visualized in Figure 8.

	Accuracy(Recall)	Precision	F1-score
T-shirt/top	0.85	0.843254	0.846614
Trouser	0.974	0.986829	0.980372
Pullover	0.823	0.809243	0.816063
Dress	0.898	0.887352	0.892644
Coat	0.819	0.81982	0.81941
Sandal	0.976	0.973081	0.974538
Shirt	0.726	0.751553	0.738555
Sneaker	0.956	0.955045	0.955522
Bag	0.976	0.975025	0.975512
Ankle boot	0.966	0.960239	0.963111
Total	0.8964	0.896144	0.896234

Table 2. Accuracy table

The model achieves an accuracy of 89.64% on the testing dataset. It performs exceptionally well on the labels "Sandal" and "Bag," with accuracies exceeding 97.5%. However, it struggles with the "Shirt" category, achieving an accuracy below 73%.

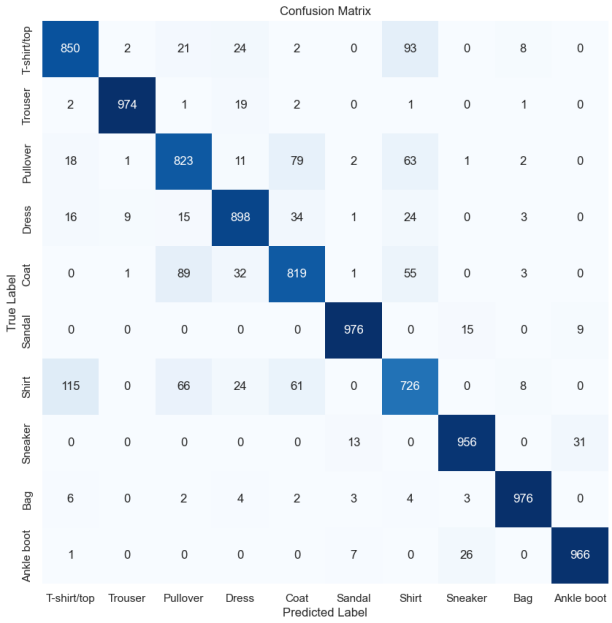

Figure 8. Confusion matrix

Examining the confusion matrix reveals notable misclassifications between "T-shirt/top" and "Shirt," as well as between "Pullover" and "Coat." These errors likely stem from the similarities in their shapes. Conversely, "Sandal" and "Bag" exhibit unique shapes, contributing to their high accuracy. This demonstrates that our neural network has indeed gleaned insightful knowledge about the dataset.

Next, we select 10 samples from each of the 10 labels and visualize them after passing through the first and second layers. This will provide us with some intuition about the trained neural network. The visualizations are depicted in Figures 9, 10, and 11.

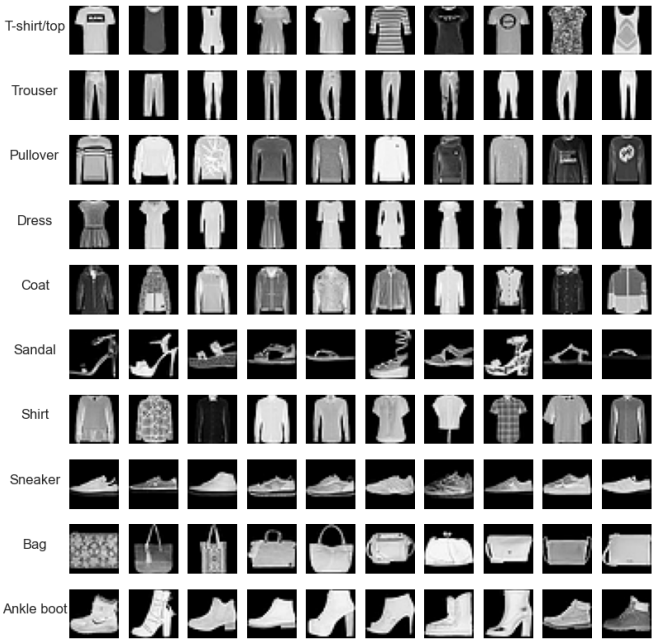

Figure 9. Original images

Upon inspection, it's evident that the images after the first layer for each class lack significant similarity. This may be at-

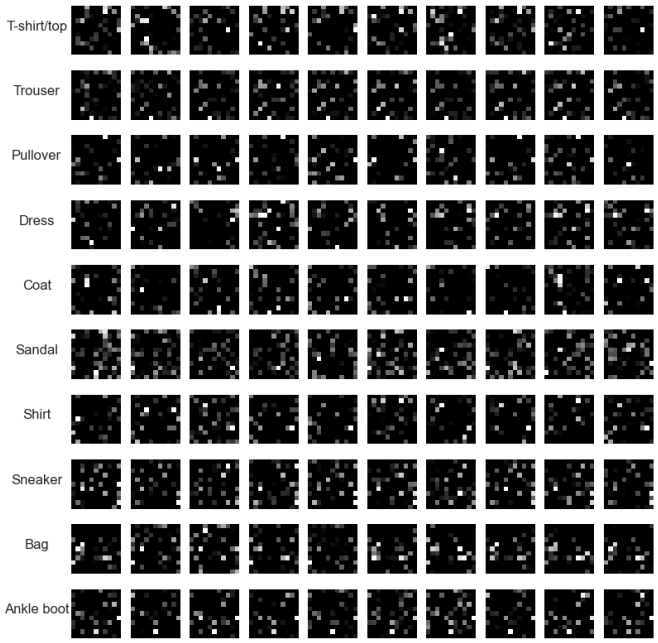

Figure 10. Images after the first layer

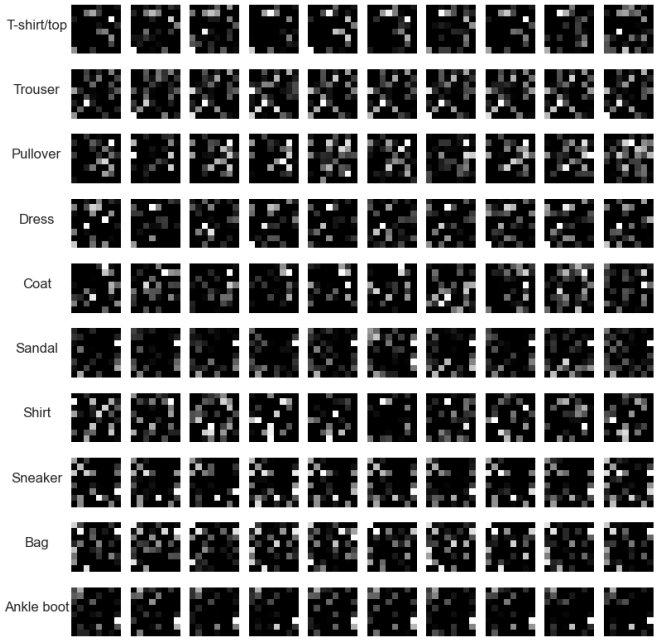

Figure 11. Images after the second layer

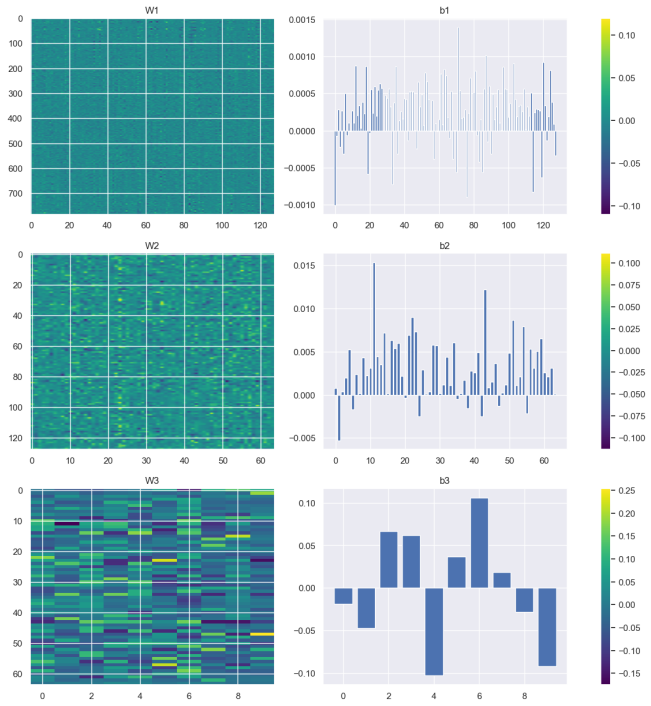

Figure 12. Weights and biases of the network

tributed to the neural network's insufficient learning of dataset features within a single layer. However, in the images after the second layer, distinct patterns for each class become apparent. Notably, the Sandal and Bag classes exhibit the most similarity among the selected 10 images, while the patterns for the Shirt class are less consistent, consistent with the observations in the accuracy table.

Lastly, the weights and biases of the trained neural network are depicted in Figure 12. We observe that the absolute values of $\mathbf{W}_1$ and $\mathbf{W}_2$ are slightly lower than those of $\mathbf{W}_3$, potentially due to the combined effect of the l_2 penalty imposed on the weight matrices and gradient flow. Additionally, a discernible pattern emerges in each row or column of $\mathbf{W}_1$, characterized by alternating positive and negative values. This behavior suggests that the weight matrix may be capturing information related to edge detection in the image data. However, for $\mathbf{W}_2$ and $\mathbf{W}_3$, the patterns appear random and lack a clear explanation.

6. Discussion

While our neural network has achieved relative good accuracy in classifying images from the Fashion-MNIST dataset, there are several areas where further refinement and exploration could potentially lead to even better performance and model robustness.

6.1. Weight Initialization

In this project, we briefly touched upon weight initialization in the neural network. In practice, various initialization methods like Xavier initialization and Kaiming initialization can significantly impact network performance. Exploring these methods could lead to improved results.

6.2. Hyperparameter Tuning

Although we provided hyperparameters such as `lr_decay`, `batch_size`, and `epochs`, we opted for their default values. However, fine-tuning these hyperparameters could potentially enhance the neural network's performance. Further investigation into optimal hyperparameter settings may yield better results.

7. Appendix

A. Grid Search Results

hidden_dims	reg	learning_rate	Val acc
[128, 64]	0.001	0.0001	0.6629
[128, 64]	0.001	0.001	0.8312
[128, 64]	0.001	0.01	0.8665
[128, 64]	0.01	0.0001	0.6599
[128, 64]	0.01	0.001	0.8279
[128, 64]	0.01	0.01	0.8598
[128, 64]	0.1	0.0001	0.6587
[128, 64]	0.1	0.001	0.8222
[128, 64]	0.1	0.01	0.8429
[128, 48]	0.001	0.0001	0.6456
[128, 48]	0.001	0.001	0.8298
[128, 48]	0.001	0.01	0.8646
[128, 48]	0.01	0.0001	0.6463
[128, 48]	0.01	0.001	0.8286
[128, 48]	0.01	0.01	0.8653
[128, 48]	0.1	0.0001	0.6508
[128, 48]	0.1	0.001	0.8240
[128, 48]	0.1	0.01	0.8416
[64, 48]	0.001	0.0001	0.6183
[64, 48]	0.001	0.001	0.8172
[64, 48]	0.001	0.01	0.8633
[64, 48]	0.01	0.0001	0.6176
[64, 48]	0.01	0.001	0.8186
[64, 48]	0.01	0.01	0.8593
[64, 48]	0.1	0.0001	0.6100
[64, 48]	0.1	0.001	0.8100
[64, 48]	0.1	0.01	0.8410

Table 3. Grid search results

B. Random Search Results

reg	learning_rate	Val acc
0.0083	0.02	0.6124
0.005	0.0257	0.3829
0.0088	0.0139	0.8339
0.0057	0.018	0.7889
0.0072	0.0213	0.5127
0.0083	0.0129	0.8594
0.0065	0.0041	0.8606
0.007	0.0057	0.8655
0.0057	0.0128	0.8544
0.001	0.0185	0.7801
0.0043	0.0046	0.8617
0.0049	0.0203	0.6039
0.0094	0.0132	0.7196
0.006	0.0065	0.8651
0.0071	0.0187	0.7650
0.0055	0.0071	0.8634
0.0098	0.0307	0.2586
0.0068	0.0132	0.8542
0.0098	0.0239	0.3799
0.0099	0.0016	0.8464
0.001	0.01	0.8668

Table 4. Random search results