# DATA130051.01 Computer Vision Project 1

吴嘉骜 21307130203

2024 年 5 月 1 日

**Abstract**

This report presents the implementation of an image classifier for the `Fashion-MNIST` dataset using a 3-layer Fully Connected Neural Network (FCNN). The classifier is implemented in Python from scratch, and the source code is provided as below. We apply the ReLU activation function to both hidden layers and the softmax function to the output layer. The optimizer is the stochastic gradient descent (SGD) algorithm with Momentum. Some usual techniques such as learning rate decay, mini-batch processing, and $L2$ regularization are also applied. The final accuracy of the classifier on the test set is 0.85 and on the training set is 0.87. In this report, we present the code structure, the training details, and the experiment results of the classifier, with an analysis of the impact of hyperparameters. We also find some patterns of the learned weights through visualization, and finally limitations and future enhancements.

**Project description**

The objective of this project is to implement an image classifier based on a 3-layer neural network for the `Fashion-MNIST` dataset. The classifier should be implemented from scratch in Python, without using any deep learning libraries such as TensorFlow or PyTorch.

**Experiment environment**

Windows 11 VsCode Python 3.8.18

NumPy 1.23.5, Matplotlib 3.7.2

**Model links**

Source code: Github link

Model parameters: Baidu Netdisk link

## 1 Introduction

The task of image classification has seen significant advancements with the advent of neural networks. In this project, we will construct a three-layer neural network from scratch to classify images. The network is trained on the Fashion-MNIST dataset, and the performance is evaluated based on the accuracy of the classification.

### 1.1 Dataset

The Fashion-MNIST dataset, hosted at Fashion-MNIST GitHub, comprises 60,000 training and 10,000 testing images. Each image is a 28x28 grayscale depiction of various fashion articles, equally classified into 10 categories, making it an excellent benchmark for evaluating machine learning algorithms. In this project, we randomly split the original training dataset into a new training set (45,000 images) and a validation set (15,000 images). The test set remains unchanged.

### 1.2 Neural Network

A neural network is a computational model inspired by the structure of the brain and is composed of interconnected nodes or neurons, which process data in layers. In this project, we are instructed to implement a three-layer neural network. Considering the simplicity of the structure, we choose a fully connected neural network (FCNN) with two hidden layers. A sketch of the network architecture is shown in figure 1. Note that the number of neurons in each layer is just schematic and is not the actual settings in our implementation.

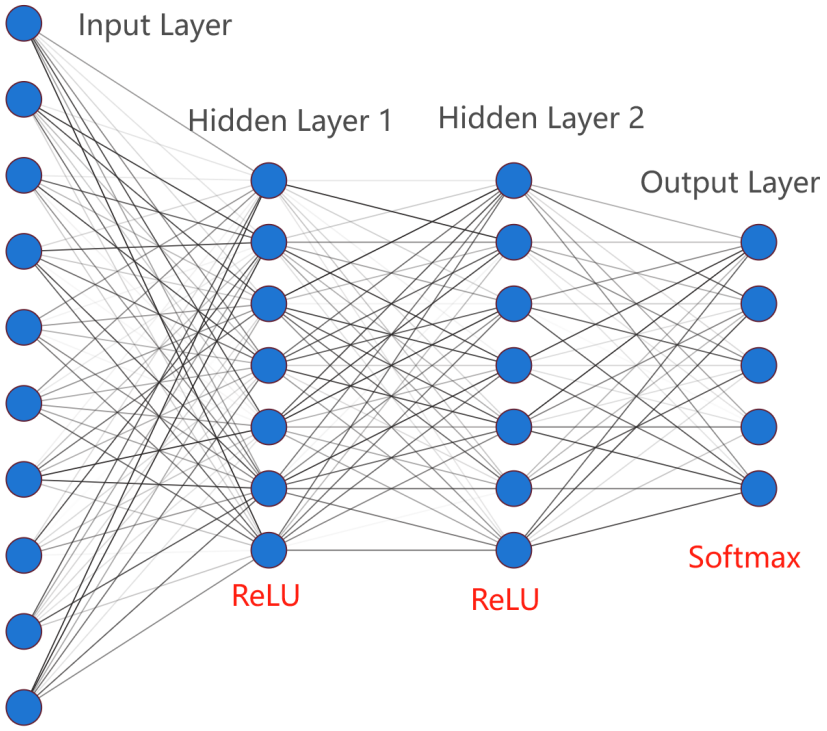



Figure 1: Three-layer neural network architecture

### 1.3 Outline of the Report

This report is structured as follows:

- Section 2 discusses the code structure and implementation details of the neural network, including data preprocessing, training methodology, and parameter tuning.

- Section 3 presents the results and evaluates the network's performance.
- Section 4 concludes the report with a discussion and potential future work.

# 2 Implementation

During the implementation of the neural network, we followed the standard procedure presented in our course materials and the textbook *Deep Learning* [1].

## 2.1 Code structure

The code is organized into several modules, each responsible for a specific aspect of the neural network implementation. The main modules are as follows:

- `utils.py`: Includes various utility functions needed across the project. `DataHandler` is responsible for loading and preprocessing the dataset, `save_model` and `load_model` handles the saving and loading of the trained model, and `Visualizer` provides methods for plotting training progress and results.
- `layers.py`: Contains the implementation of one single layer in the network. It incldes classes for the input layer (`InputLayer`) and general hidden layers (`FullyConnectedLayer`). Each hidden layer has a forward and backward method to compute the output and gradients, respectively. For clearity, we implement the activation functions `ActivationReLU` and `ActivationSoftmax` as separate classes in this module.
- `model.py`: Defines the `Model` class that encapsulates the entire neural network. This class integrates multiple layers and provides an interface for building the network architecture, setting optimizer and loss function, and integrating with the training routines like forward pass and backward propagation. Saving and loading parameters are also supported.
- `train.py`: Includes classes and functions essential for training the network. `SGDOptimizer` manages the stochastic gradient descent optimization process, `CELoss` calculates the cross-entropy loss, and `Trainer` finishes the training sessions by handling data batches, loss computation, and parameter updates.
- `evaluate.py`: Hosts the `Evaluator` class used to evaluate the trained model on a testing set.
- `metrics.py`: Defines the `Accuracy` metric class to compute the accuracy of the model's predictions against the ground truth, providing a straightforward way to gauge performance during and after training.
- `para_tune.ipynb`: A Jupyter notebook that contains the hyperparameter tuning process using grid search. Also provides an example to load a saved model.
- `main.py`: The main script that orchestrates the entire training process with the best hyperparameters found during the tuning phase. Prints detailed training and validation metrics.

## 2.2 Data Preprocessing

Thankfully, the Fashion-MNIST dataset is well-structured and balanced, eliminating the need for extensive preprocessing steps. Each class in the dataset contains an equal number of images, ensuring class balance without the need for specific treatments.

Since the images in the dataset are grayscale, with pixel values ranging from 0 to 255, normalization is performed to scale the pixel values to a range between -1 and 1. This normalization is achieved by subtracting 127.5 from each pixel value and then dividing by 127.5. This transformation centers the pixel values around zero, which can improve the convergence of the training process.

Furthermore, before initiating the training phase, the dataset is split into training and validation sets using a 3:1 ratio. Additionally, to enhance the randomness and generalization of the training process, the training dataset is shuffled after the initial split. This shuffling step prevents any inherent order or patterns in the dataset from influencing the training process.

## 2.3 Training Methodology

**Model Part**: The architecture of our model is designed to be flexible, allowing for easy modifications of its structure. This flexibility includes the ability to add or remove hidden layers, customize the size of these layers, select activation functions, choose the type of optimizer, and specify regularization methods. For this project, given time constraints, we followed standard practices and opted to construct a three-layer FCNN with two hidden layers of equal size. Both layers used the ReLU activation function, while the output layer employs the Softmax activation function for multi-class classification. The optimizer selected for this network is based on Stochastic Gradient Descent (SGD) with an added L2 regularization term, although experiments later showed that L2 regularization had a minimal impact on this particular problem.

**Training Part**: During training, we employed several techniques to optimize performance and ensure robust learning:

- **Mini-batch training:** The model is trained using mini-batches, with each batch containing 128 samples. We found this batch size to be optimal for balancing computational efficiency and model performance. We trained the model for 10 epochs, with each epoch iterating over the entire training dataset.
- **Momentum in SGD:** We enhanced the SGD optimizer by incorporating momentum, which accelerates the gradient vectors in the right directions, thus leading to faster converging. The momentum is calculated using the formula:

  $$v_t = \mu \cdot v_{t-1} + \eta \cdot \nabla_\theta J(\theta),$$

  where $v_t$ is the current velocity, $\mu$ is the momentum coefficient, $\eta$ is the learning rate, and $\nabla_\theta J(\theta)$ is the gradient of the loss function with respect to the parameters.
- **Learning rate decay:** The learning rate is adjusted over time using the decay formula:

  $$\text{cur\_lr} = \text{lr} \times \left( \frac{1}{1 + \text{decay} \times \text{iterations}} \right),$$

where lr is the initial learning rate, decay is the decay rate, and iterations counts the number of updates made so far. This approach helps in reducing the learning rate gradually, ensuring that the training does not oscillate or diverge during later stages.

- **Performance monitoring on validation set:** The model's performance is continuously monitored using a validation set, namely, recording the loss and accuracy for each epoch. Additionally, the best model weights are saved automatically based on the accuracy metrics from the validation set.

By employing these methods, we aim to effectively train our neural network, optimizing its performance while also maintain its generalization capabilities.

# 3 Results and analysis

After completing the development of the training code, the initial step was to identify the best hyperparameters for the model. We conducted extensive experiments by adjusting key hyperparameters and recorded the model's performance under various configurations. Once the optimal hyperparameters were determined, we proceeded with the full training and testing of the model.

## 3.1 Hyperparameter Tuning

**Parameter search Part**: The primary metrics monitored during this phase were the accuracy and loss values on both the training and validation datasets. We experimented with several hyperparameters, including the learning rate, size of the hidden layers, and regularization strength. Each parameter was varied within a predetermined range to observe its impact on the model's performance.
The hyperparameters tuning result can be found in Table 1.

## 3.2 Experiment performance

**Testing Part**: Through iterative testing and validation, the optimal settings for the hyperparameters were established, as shown in Table 2.
We trained the model using the optimal hyperparameters and evaluated its performance on the test set. The plot of training and validation accuracy and loss metrics are shown in Figure 2. Note that the validation accuracy is calculated per epoch for time costing, so the curve is actually dot connected by lines. We observe that on average the training and validation metrics are close, indicating that the model generalizes well.

# 4 Discussion

## 4.1 Weight Visualization

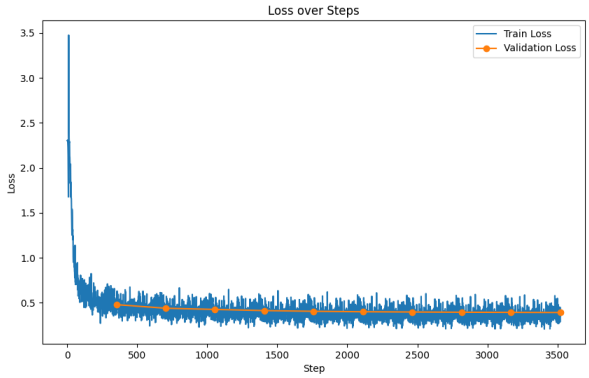
To gain insights into the model's learning process, we visualized the weights of the first hidden layer, as shown in Figure 3. The choice is due to the fact that the first hidden layer is directly connected to the

Table 1: Results of Hyperparameter Tuning for Different Configurations

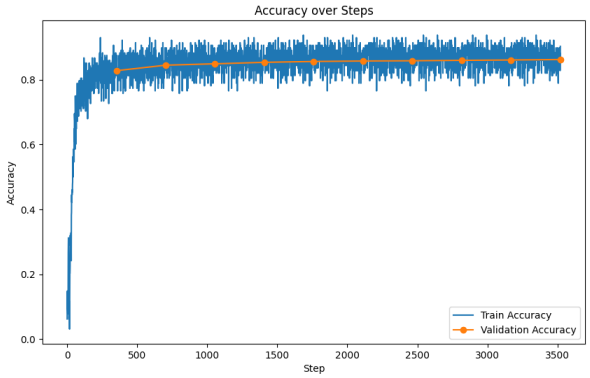
| **Config** | **Training Acc** | **Training Loss** | **Validation Acc** | **Validation Loss** |
|---|---|---|---|---|
| Neuron Layer = 32 | 0.8433 | 0.3620 | 0.8359 | 0.4574 |
| Neuron Layer = 64 | 0.8638 | 0.2639 | 0.8531 | 0.3947 |
| Neuron Layer = 128 | 0.8613 | 0.2914 | 0.8543 | 0.3983 |
| **Neuron Layer = 256** | 0.8711 | 0.2524 | 0.8633 | 0.3763 |
| Batch Size = 32 | 0.8392 | 0.1392 | 0.8298 | 0.4757 |
| Batch Size = 64 | 0.8594 | 0.1481 | 0.8476 | 0.4185 |
| **Batch Size = 128** | 0.8728 | 0.3470 | 0.8605 | 0.3945 |
| Batch Size = 256 | 0.8604 | 0.4013 | 0.8511 | 0.4206 |
| **Learning Rate = 1** | 0.8669 | 0.3720 | 0.8550 | 0.4037 |
| Learning Rate = 0.1 | 0.7382 | 0.7104 | 0.7297 | 0.7365 |
| Learning Rate = 0.01 | 0.3973 | 2.2986 | 0.3947 | 2.2982 |
| Decay = 1 | 0.7659 | 0.6317 | 0.7580 | 0.6637 |
| **Decay = 0.1** | 0.8682 | 0.3474 | 0.8561 | 0.4074 |
| Decay = 0.01 | 0.7846 | 0.6432 | 0.7640 | 0.6848 |
| Decay = 0.001 | 0.1105 | 2.3131 | 0.0987 | 2.3042 |
| Momentum = 0.0 | 0.8159 | 0.4779 | 0.8039 | 0.5426 |
| Momentum = 0.7 | 0.8652 | 0.3521 | 0.8569 | 0.4027 |
| **Momentum = 0.8** | 0.8709 | 0.3646 | 0.8609 | 0.3933 |
| Momentum = 0.9 | 0.8449 | 0.4187 | 0.8381 | 0.4620 |
| L2 Reg Weight = 0 | 0.8689 | 0.3388 | 0.8539 | 0.4009 |
| L2 Reg Weight = 0.1 | 0.0982 | 2.3028 | 0.0969 | 2.3028 |
| L2 Reg Weight = 0.01 | 0.8364 | 0.7465 | 0.8305 | 0.4957 |
| **L2 Reg Weight = 0.001** | 0.8703 | 0.4287 | 0.8579 | 0.3966 |
| L2 Reg Weight = 0.0001 | 0.8654 | 0.3941 | 0.8551 | 0.4080 |

Table 2: Optimal Hyperparameters Configuration

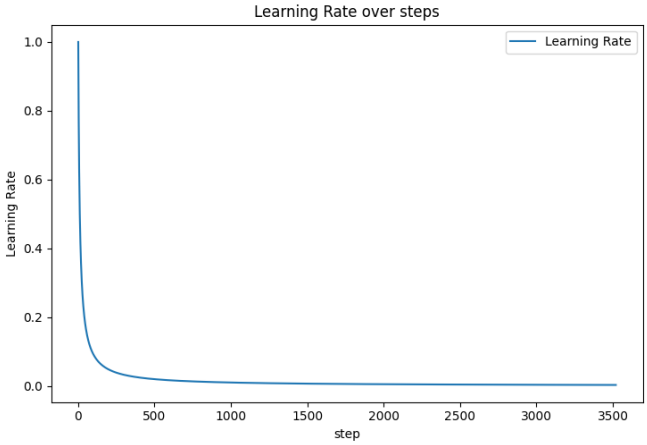
| **Parameter** | **Value** |
|---|---|
| Hidden Layer Size | 128 |
| Initial Learning Rate | 1.0 |
| Decay | 0.1 |
| Momentum | 0.8 |
| L2 Regularization Strength | 0.0 |
| Batch Size | 128 |
| Epochs | 10 |




(a) Loss curve




(b) Accuracy curve




(c) Learning rate curve

Figure 2: Training and validation metrics

input layer and can capture the raw features of the input data. Weights of the middle hidden layers are more abstract and harder to interpret.

We observe that the color mapping of the weights indeed shows some patterns, part of which can be recognized as the shape of the fashion items. This visualization provides a qualitative understanding of the model's learning process and the features it captures.

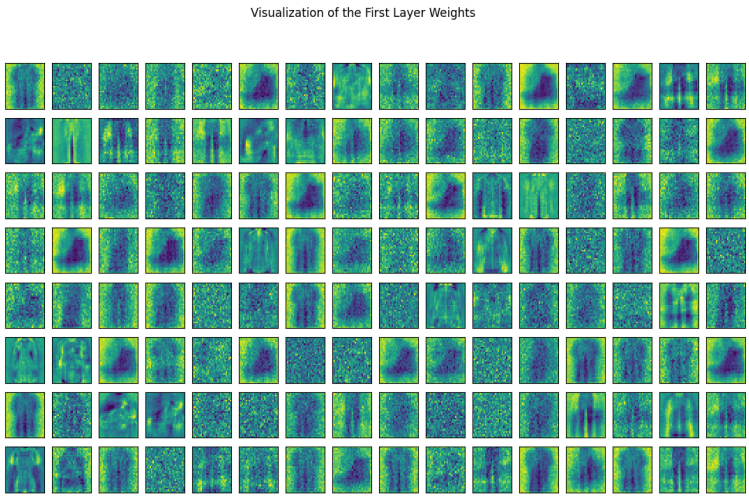



Figure 3: Visualization of the weights of the first hidden layer

We also plot a histogram of biases in the first hidden layer, as shown in Figure 4. This is harder to explain, and we just see some skewness in the distribution of biases. Most of the biases are close to zero, which is expected.

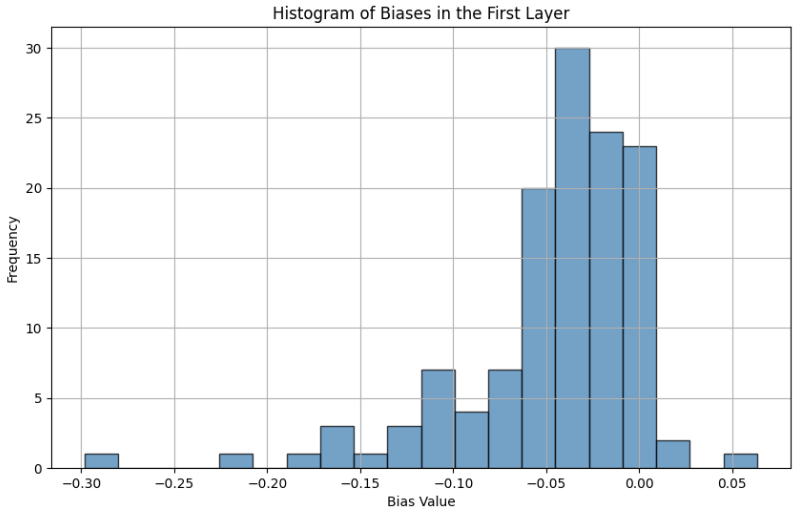



Figure 4: Histogram of biases in the first hidden layer

### 4.2 Limitations and future work

Despite the successes achieved by the current model, there are several areas that could benefit from further exploration and development:

1. **Neural Network Architecture:** The current model utilizes a relatively simple three-layer fully connected neural network. Future work could explore the addition of more hidden layers or the adoption of more complex architectures such as Convolutional Neural Networks (CNNs), which are particularly well-suited for image classification tasks.

2. **Optimizer:** While Stochastic Gradient Descent (SGD) with momentum was used in this project, other optimization algorithms like Adam might offer improvements in convergence rates and overall performance.

3. **Regularization Techniques:** The current model implementation focuses on L2 regularization. Implementing additional regularization methods such as Dropout or Early Stopping could help in combating overfitting more effectively, especially in more complex models.

## References

[1] Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. Cambridge, Mass: The MIT press.