Image Classification Based On Three-Layer Neural Network

Xiang Zheng, 21307110169

School of Data Science, Fudan University

Abstract—Welcome to tau (τ) LATEX class for making academic articles and lab reports. In this example template, we will guide you through the process of using and customizing this class to your needs. For more information of this class check out the appendix section. There, you will find snippets codes that define key aspects of the template, allowing you to explore and modify them.

keywords—*LATEX class, lab report, academic article, tau class*

1. Introduction

In this project, we aim to develop and train a fully connected neural network to classify images from the Fashion-MNIST dataset. The Fashion-MNIST dataset, curated by Zalando, comprises a collection of grayscale images representing various fashion items, along with corresponding labels indicating their respective categories. With 60,000 training examples and 10,000 test examples, this dataset serves as a standard benchmark for evaluating machine learning algorithms in the context of fashion image classification.

The neural network architecture we will employ consists of three linear layers, which will be trained to learn meaningful representations from the input images and make accurate predictions about their corresponding classes. Throughout this project, we will explore different components of the neural network, including activation functions, loss functions, and training strategies, to optimize its performance on the Fashion-MNIST dataset.

The remainder of this report is structured as follows: Section 2 provides the downloading the pre-process method of the Fashion-MNIST dataset. 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 hyper-parameters 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 offers concluding remarks and discusses potential avenues for future research and improvement.

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 website 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:

$$\mathbf{Y} = \mathbf{XW} + \mathbf{b} \tag{3.1.1}$$

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, I 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:

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

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

You can observe that the shapes of these matrices match perfectly. The same technique can be applied to other following linear layers. Feel free to refer to my code in `linear_layer.py` to see the details of implementation.

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:

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

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:

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

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

$$\mathbf{Y}' = \tanh \mathbf{Y} \qquad (3.2.5)$$
$$d\mathbf{Y} = d\mathbf{Y}' \cdot (1 - \tanh^2 \mathbf{Y}) \qquad (3.2.6)$$

3.2.3. Sigmoid

Let's denote the sigmoid function as:

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

Taking the derivative with respect to x, we obtain:

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

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

$$\mathbf{Y}' = \sigma(\mathbf{Y}) \qquad (3.2.9)$$
$$d\mathbf{Y} = d\mathbf{Y}' \cdot \sigma(\mathbf{Y})(1 - \sigma(\mathbf{Y})) \qquad (3.2.10)$$

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, I'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

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

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:

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

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:

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

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

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

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:

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

and it's implemented in Python code as

```python
# 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 hyper-parameters such as `hidden_dims`, `reg`, and `weight_scale`.

For `hidden_dims`, it determines the size of the hidden layers. Here, I've chosen the empirically recommended values $[128, 64]$.

`reg` is a regularization parameter that helps prevent overfitting by penalizing large weights in the model. In the next section, I'll discuss how grid search is used to find the best value for `reg`.

For 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.

4.1. Update Rule

For update rules, I 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

$$\mathbf{w} = \mathbf{w} - \text{lr} \cdot d\mathbf{w} \qquad (4.1.1)$$

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

$$\mathbf{v} = \mathtt{momentum} \cdot \mathbf{v} - \mathtt{lr} \cdot \mathtt{d}\mathbf{w} \qquad (4.1.2)$$
$$\mathbf{w} = \mathbf{w} + \mathbf{v} \qquad (4.1.3)$$

where $\mathbf{v}$ denotes the velocity. The empirical default values for $\mathtt{lr}$ and $\mathtt{momentum}$ 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:
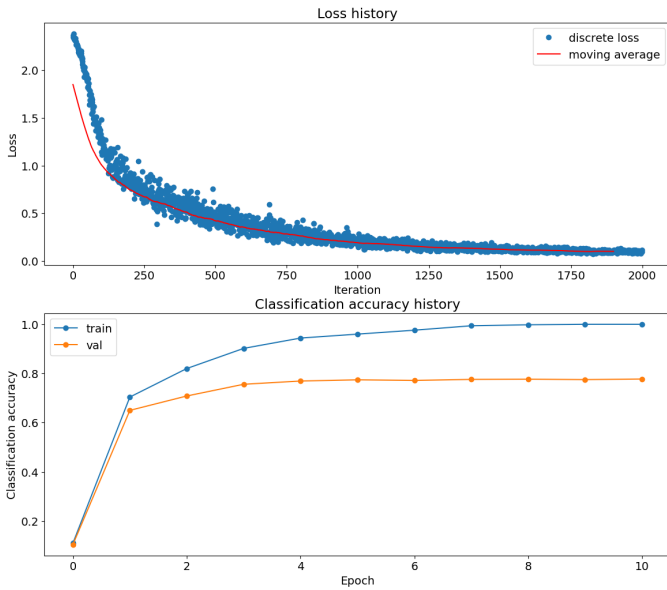

Figure 1. Training on the Small Dataset

After 9 epochs of training (approximately 1800 iterations), the model achieved perfect accuracy on the training dataset, while the validation (testing) accuracy plateaued around 0.75. 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 using 9 different combinations of activation functions while keeping other parameters constant, which can be seen as a greedy search. Subsequently, we will implement a grid search method to find the best hyper-parameters `reg`, `learning_rate`, and `lr_decay`. Finally, we will experiment with 4 different update rules mentioned in section 4.1.2 and compare their performance.

4.3.1. Activation Function

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.
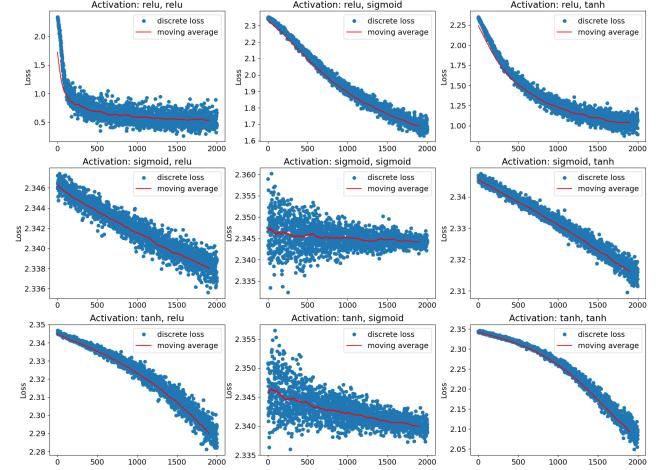

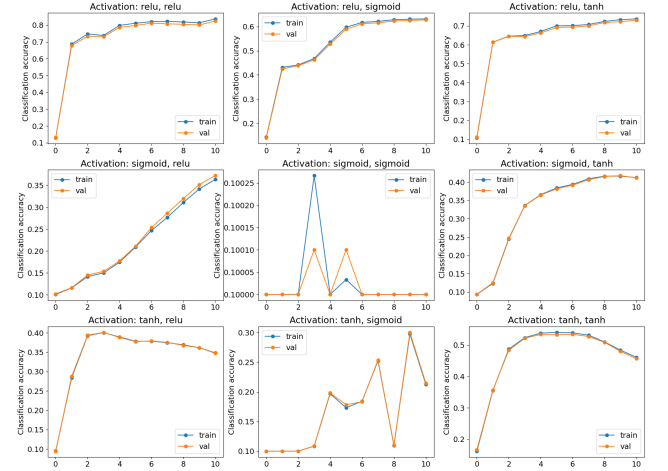**Figure 2.** Loss on different combinations of activation functions

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. Grid Search Hyper-parameters

4.3.3. Update Rules

4.4. Train on Full Dataset

5. Load & Test

5.1. Load Pre-trained Model

5.2. Evaludate Model on Testing Dataset

6. Discussion