Mechanics of Flexible Structures and Soft Robots Homework 3

706180341 Evelyn Nahyeon Kim
Github: https://github.com/lmcr136a/MAE263F

I. Neural Network Framework

A. Datasets: MNIST

In this project, the MNIST dataset is used, which is widely used in computer vision. The training set consists of 50,000 samples, and the test set has 1,000 samples. The images are 28x28 pixels, and to apply them to the multi-layer perceptron model, flattening is applied to get a vector of length 784. After flattening, the vector is normalized to have values between 0 and 1. As for the labels, single-digit labels were encoded into a one-hot encoded vector for use in the model.

B. Model Construction

```
function parameters = initialize_parameters(layer_dims)
    L = length(layer_dims);
    parameters = cell(1, L-1);
    for l = 1:(L-1)
        parameters{l}.W  = randn(layer_dims(l+1), layer_dims(l));
        parameters{l}.b = zeros(layer_dims(l+1), 1);
    end
end
```

Listing 1: Initialization

The model basically multiplies weight and bias to the input to get the output of the layer. Multiply the weight to the input, add bias, apply the activate function, and move the output to the next layer. In the case of the last layer, the softmax function is applied instead of the activate function, and the loss is calculated by comparing the output with the ground truth. The model weights are randomly initialized to follow the standard normal distribution, and the model bias is initialized to zero, as shown in Listing 1. The tanh2 used as the activation function in this model follows the equation:

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

In the final layer, after calculating the weights, biases, and inputs, instead of using the tanh activation, a softmax activation is applied. It maps the inputs to a probability distribution over the possible classes, ensuring that the output values are between 0 & 1, and sum up to 1:

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \tag{2}$$

where i is an index of K classes.

C. Overall Implementation Structure of Learning Process

The overall learning structure is represented in Algorithm 1. Training the model by dividing the dataset into batches allows for faster epoch iterations and results in improved performance. It helps prevent learning towards individual data points, allowing the model to understand the data more comprehensively during training.

Algorithm 1 illustrates the overall process of the model learning with dataset in batches. We can obtain a trained model as the output with this procedure.

```
Input: Model Parameters, $X_{train}$, $Y_{train}$, bs: batch size
Output: Updated Model Parameters
1:  for $i = 1$ to Epochs do
2:      $indices$ = Random order indices of the Inputs
3:      $X_{train} \leftarrow X_{train}(:, indices)$
4:      $Y_{train} \leftarrow Y_{train}(:, indices)$
5:      for $i = 1$ to BatchNumber do
6:          $X_{batch} \leftarrow X_{train}(:, (j-1)*bs+1 : j*bs)$
7:          $Y_{batch} \leftarrow Y_{train}(:, (j-1)*bs+1 : j*bs)$
8:          $Pred = ForwardPropagation(X_{batch})$
9:          Compute Loss with $Pred, Y_{batch}$
10:         Update Parameters (Backpropagation)
```

By randomly selecting indices to create batches, we can achieve a higher validation performance than repeatedly training with the same batch groups. *ForwardPropagation* is the process where the input goes through the model to become the model's output, prediction. *Backpropagation* is the process of updating the model in the reverse direction by calculating the loss with the output and the ground truth. The model is updated from the layers closer to the output layer. This is explained more specifically in the next section.

```
function loss = calculate_loss(Prediction, Y)
    loss = -sum(Y .* log(Prediction));
end
```

Listing 2: Loss Computation

For calculating loss in line 9 in Algorithm 1, the most widely used method for calculating loss, cross-entropy, is used. It calculates the loss between the model's predictions, *Prediction*, and the ground truth Y.

```
function parameters = update_parameters(parameters, gradients, learning_rate)
    L = length(parameters);
    for l = 1:L
        % Update Weights
        parameters{l}.W = parameters{l}.W - learning_rate * gradients{l}.dW;
        % Update Bias
        parameters{l}.b = parameters{l}.b - learning_rate * gradients{l}.db;
    end
end
```

Listing 3: Updating model's parameters

The process of updating the model's parameters, as shown in line 10 in Algorithm 1, is depicted in the listing 3. It involves updating the parameters by subtracting the product of the learning rate and gradients from their current values.

D. Training Process: Forward & Backward Propagation

There are several steps to proceed training of the model. The training process involves multiplying the inputs by the weights, adding the bias, applying activation, and passing it to the next layer. If it's the last layer, softmax is applied, and the loss is calculated by comparing it with the ground truth (GT). Subsequently, backpropagation is carried out to update weights and biases, completing one learning iteration. In this section, a more detailed explanation of these steps is provided, and the implementations of the following equations are shown in Listing 4.

```
function activations = forward_propagation(X, parameters)
    L = length(parameters);
    A = X;
    activations = cell(1, L+1);
    activations{1} = X;
    for l = 1:L
        % Equation 3
        Z = parameters{l}.W*A + parameters{l}.b;
        if l == L
            % Equation 6
            A = softmax(Z);
        else
            % Equation 4
            A = tanh2(Z);
        end
        activations{l+1} = A;
    end
end
```

Listing 4: Forward Propagation

Let's assume that the input example is $\mathbf{x} \in \mathbb{R}^d$ and that our layer does not include a bias term. Here, the intermediate variable is:

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \tag{3}$$

where $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ is the weight parameter of the layer 1. After running the intermediate variable $\mathbf{z}^{(1)} \in \mathbb{R}^h$ through the activation function $tanh(x)$ we obtain our hidden activation vector of length h:

$$\mathbf{h}^{(1)} = tanh(\mathbf{z}^{(1)}) \tag{4}$$

The equation of $tanh$ function is described in Equation 1 The layer output $\mathbf{h}^{(1)}$ is also an intermediate variable. The same process can be applied to the further layers. If the model only has two layers, the weight of the second layer is $\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$, we can obtain an output layer variable with a vector of length q:

$$\mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)} \tag{5}$$

$$\mathbf{o} = softmax(\mathbf{z}^{(2)}) \tag{6}$$

The equation of $softmax$ function is described in Equation 2 Assuming that the example label is y, we can then calculate the loss term for a single data example,

$$\mathbf{L} = loss(\mathbf{o}, y) \tag{7}$$

Backpropagation refers to the method of calculating the gradient of neural network parameters. In short, the method traverses the network in reverse order, from the output to the input layer, according to the chain rule from calculus. The algorithm stores any intermediate variables (partial derivatives) required while calculating the gradient with respect to some parameters. The objective of backpropagation is to calculate the gradients $\frac{\partial \mathbf{L}}{\partial \mathbf{W}^{(1)}}$ and$\frac{\partial \mathbf{L}}{\partial \mathbf{W}^{(2)}}$. To accomplish this, we apply the chain rule and calculate, in turn, the gradient of each intermediate variable and parameter. The order of calculations is reversed relative to those performed in forward propagation since we need to start with the outcome of the computational graph and work our way toward the parameters. The first step is to calculate the gradients of the gradient $\frac{\partial \mathbf{L}}{\partial \mathbf{W}^{(2)}}$ of the model parameters closest to the output layer. Using the chain rule yields:

$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}^{(2)}} = prod(\frac{\partial \mathbf{L}}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}}) = \frac{\partial \mathbf{L}}{\partial \mathbf{o}}\mathbf{h}^{(1)T} \tag{8}$$

To obtain the gradient with respect to $\mathbf{W}^{(1)}$ we need to continue backpropagation along the output layer to the layer. The gradient with respect to the layer output $\frac{\partial \mathbf{L}}{\partial \mathbf{h}^{(1)}}$ is given by:

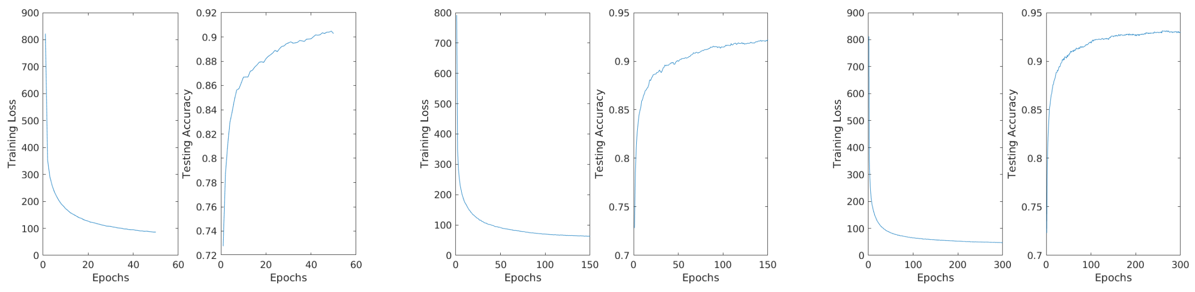
$$\frac{\partial \mathbf{L}}{\partial \mathbf{h}^{(1)}} = prod(\frac{\partial \mathbf{L}}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}^{(1)}}) = \mathbf{W}^{(2)T}\frac{\partial \mathbf{L}}{\partial \mathbf{o}} \tag{9}$$

Since the activation function $tanh$ applies elementwise, calculating the gradient $\frac{\partial \mathbf{L}}{\partial \mathbf{z}^{(1)}} \in \mathbb{R}^h$ of the intermediate variable $\mathbf{z}^{(1)}$ requires that we use the elementwise multiplication operator, which we denote by $\circ$.

$$\frac{\partial \mathbf{L}}{\partial \mathbf{z}^{(1)}} = prod(\frac{\partial \mathbf{L}}{\partial \mathbf{h}^{(1)}}, \frac{\partial \mathbf{h}^{(1)}}{\partial \mathbf{z}^{(1)}}) = \frac{\partial \mathbf{L}}{\partial \mathbf{h}^{(1)}} \circ tanh'(\mathbf{z}^{(1)}) \tag{10}$$

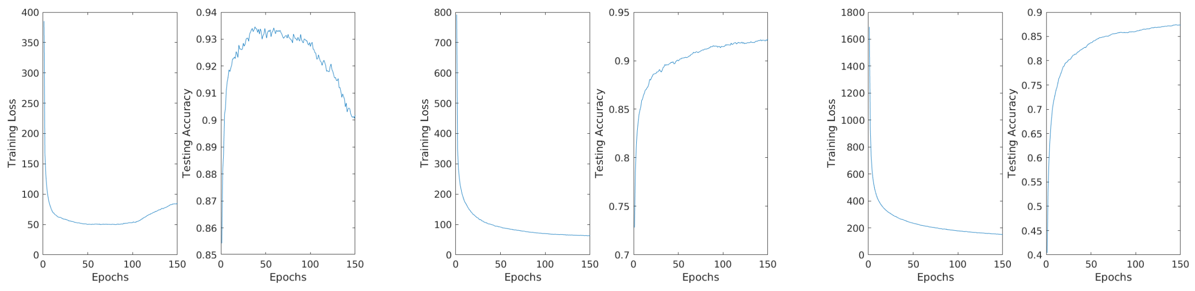
$$\frac{\partial \mathbf{L}}{\partial \mathbf{W}^{(1)}} = prod(\frac{\partial \mathbf{L}}{\partial \mathbf{z}^{(1)}}, \frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{W}^{(1)}}) = \frac{\partial \mathbf{L}}{\partial \mathbf{z}^{(1)}}\mathbf{x}^T \tag{11}$$

Finally, we can obtain the gradient $\frac{\partial \mathbf{L}}{\partial \mathbf{W}^{(1)}}$ of the model parameters closest to the input layer.



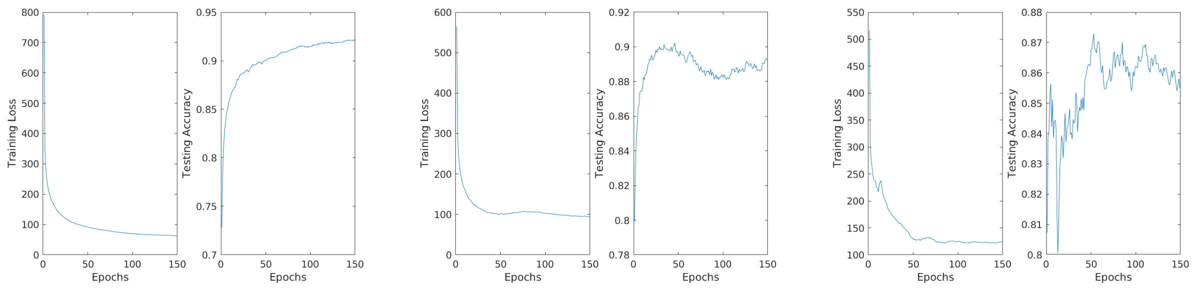
(a) Epoch = 50 (b) Epoch = 150 (c) Epoch = 300

Fig. 1: Training loss and test accuracy of experiments conducted with different epochs.



(a) Learning rate = 0.1 (b) Learning rate = 0.01 (c) Learning rate = 0.001

Fig. 2: Training loss and test accuracy of experiments conducted with different learning rates.



(a) 2 Layers (b) 3 Layers (c) 5 Layers

Fig. 3: Training loss and test accuracy of experiments conducted with different number of layers.

II. Training Results and Ablation Study

The training result of the basic configuration of $epoch = 150$, $learningrate = 0.01$, and 2 *layers* is as shown in Figure 1b As the training progress, the loss consistently decreases and reaches a nearly stable point. The test accuracy shows a steady increase until the end of epochs. The final accuracy is around 92.5%. To investigate which parameters impact the learning performance, an ablation study will be conducted in the subsequent section.

a) Epoches: The results of experiments with varying epoch numbers while keeping other conditions the same are shown in Figure 1. When the epoch is set to 300, the test accuracy slightly increases compared to the case with 150 epochs, but the improvement is not proportional to the doubled epochs.

b) Learning rates: When only the learning rate changes under identical conditions, the results are as depicted in Figure 2. With a learning rate of 0.1, the training diverges due to the too-high learning rate, resulting in a decline in test accuracy. In the case of a learning rate of 0.001, the training progresses excessively slowly; even at epoch 150, the learning is ongoing but at a sluggish pace. Furthermore, a too-low learning rate increases the risk of falling into the local minimum, leading to lower final performance.

c) The number of layers: When the number of layers changes while keeping other conditions the same, the results are represented in Figure 3. While it might be intuitive to expect better performance with an increased number of layers, this is not always the case. More layers sometimes can allow for handling more complex tasks, but other conditions, such as epoch or learning rate, should be tuned for the task. Figure

3b and Figure 3c illustrate that improper hyper-parameters can hinder effective learning, and a higher number of layers doesn’t result in better performance.