Fall 2023

Homework 3

Due: Wednesday, November 22, 2023, 11:59 pm

Contents

1 Background

1.1 Overview of The Task

The emergence of the era of big data has accelerated the machine learning revolution. The trained model demonstrates extraordinary performance on various tasks by absorbing the "experience" from large data with machine learning algorithms. In this homework, we use machine learning to solve a classification task – classification of handwritten digits.

Classification is a supervised learning task in machine learning that aims to predict a categorical or discrete label for a given input. In classification tasks, the input is typically a set of features or attributes that describe some object or phenomenon (i.e., the input is a grayscale image in this project). The output is a categorical label that assigns the object to a specific class or category (i.e., the output is what the digit (i.e., $0 \sim 9$) shows in the image).

In this project, we learn how to use a feedforward neural network to solve the classification problem. The overview of feedforward neural network is shown in Figure 1. First, we pre-process the grayscale image (28 x 28) to convert it into a vector with a length of 784. Next, we input this vector into the feedforward neural network, and the feedforward neural network will return a $K \times 1$ output vector after performing computations involving multiple

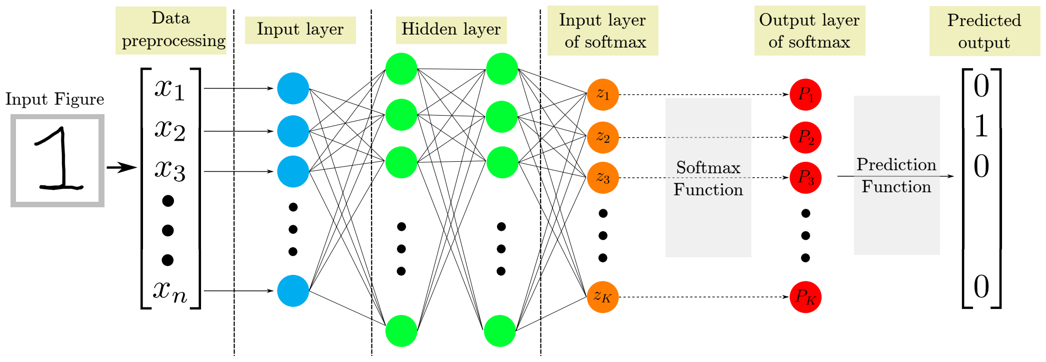

Figure 1: Overview of the handwritten digits classification with feedforward neural network.

hidden layers, the softmax function, and the prediction function. K is the number of total categories and $K = 10$ in this project corresponding to the digits $0 \sim 9$. Note that a single element is 1 in the output vector while all other elements are zeros. The index of the non-zero element corresponds to the predicted digit minus 1. For instance, if the 1st element is 1, the predicted digit is 0; if the second element is 1, the predicted digit is 1, and so on (See Section 1.2.3 for more details).

We will program the feedforward neural network with MATLAB to classify the handwritten digits. The designed feedforward neural network will be trained on the training images and labels stored in `train_images.mat, train_labels.mat`, and we will evaluate the performance of the trained feedforward neural network using the testing images and labels stored in `test_images.mat, test_labels.mat`. The following sections provide detailed instructions on designing and training the feedforward neural network using MATLAB . By the end of this project, we will have gained a preliminary understanding on how to design a neural network for a classification task and how to tune the relevant parameters.

1.2 Data Preparation

Preparing and preprocessing the data are the first step to training a machine learning model. This is an important step, as the quality of training data heavily determines the quality of the trained model. As mentioned in the overview, we will use the MNIST dataset (see Figure 2), which contains images of handwritten digits from $0 \sim 9$. Our goal is to train an image classification model that classifies an image into 10 different classes. Since the number of images for each digit is balanced and is of equal dimensions, our data preprocessing pipeline is reduced to the following tasks: 1) flattening the image, 2) normalizing the grayscale values, and 3) one-hot encoding.

1.2.1 Flattening the image

Flattening the image refers to changing the dimensions of a grayscale image which typically has two dimensions [rows $\times$ columns] into a one-dimensional vector (see Figure 3). The purpose of this is to make the data compatible with the feedforward neural network architecture, since each data point is required to be represented as a vector.

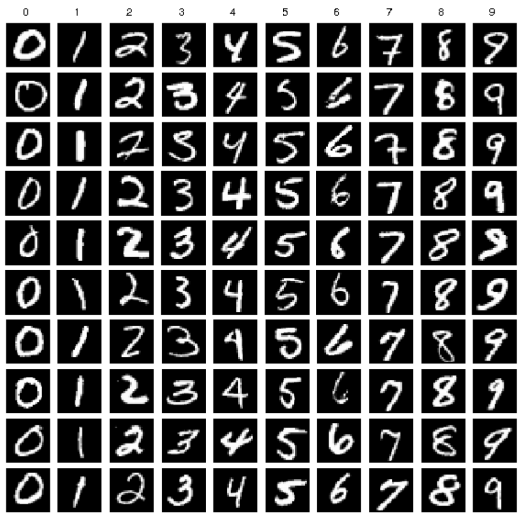

Figure 2: MNIST Dataset

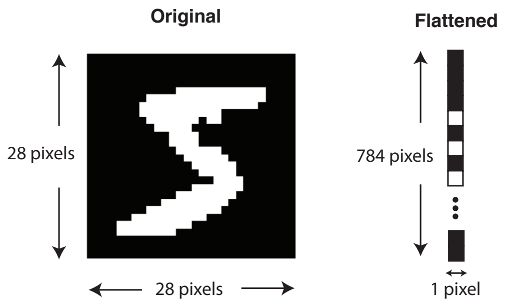

Figure 3: Original Image vs. Flattened Image

1.2.2 Normalization

The intensity of each pixel in a grayscale image ranges from $0 \sim 255$. For our project, we will normalize the pixel intensity by dividing each pixel by 255, such that the intensity of each pixel ranges from $0 \sim 1$.

1.2.3 One-hot encoding

One-hot encoding is used in classification problems to represent categorical labels numerically. For example, in a classification problem where an image is classified as a dog, cat, or horse, the labels are represented as $[1, 0, 0]$, $[0, 1, 0]$

and $[0, 0, 1]$ respectively, instead of the categorical classes in a literal sense. In one-hot encoding, each category corresponds to a unique binary digit, with only one element of 1, and the rest of 0. This ensures that there are no ordinal relationships in the output categories. For example, if the labeled dog, cat, and horse are represented as 1, 2, and 3, the model is more likely to learn an undesirable relationship between the labels, associated with their magnitude. So in your case, you have to represent digit 0 as $[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]$, digit 1 as $[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]$ and so on.

1.3 Feedforward Neural Network

1.3.1 Single Layer Perceptron

The single-layer perceptron is a type of artificial neural network that consists of a single layer of neurons, each of which computes a weighted sum of its inputs and produces an output that is transmitted to the next layer or the output layer. In a single-layer perceptron, the output of each neuron is computed using a linear activation function (like Sigmoid, ReLU, etc.), which means that the neuron computes a weighted sum of its inputs and applies a threshold function to the result.

Mathematically, the output of a single layer perceptron can be represented as follows:

$$y = \sigma\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{1}$$

where, y is the output of the neuron, σ is the activation function, $w_1, w_2, ..., w_n$ are the weights of the connections between the neuron and its inputs, $x_1, x_2, ..., x_n$ are the inputs to the neuron, and b is the bias term.

The weights $w_1, w_2, ..., w_n$ and the bias term b are used to control the strength of the connections between the neuron and its inputs. The weights are learned and adjusted during training to optimize the network's performance. The inputs $x_1, x_2, ..., x_n$ are typically normalized to ensure that they are in the same range as the weights and bias, which makes it easier to train the network. The bias term b is a constant value that is added to the weighted sum of the inputs and can be thought of as a measure of how easily the neuron can be activated.

1.3.2 Introduction to Feedforward Neural Network

A feedforward neural network is a type of artificial neural network that consists of an input layer, one or more hidden layers, and an output layer. It is called "feedforward" because the data flows through the network in one direction, from the input layer to the output layer. Feedforward neural networks are widely used in machine learning and deep learning applications, including image classification, speech recognition, and natural language processing.

In a feedforward neural network, the input layer receives the input data, which is then processed by the hidden layers, and the output layer produces the final output. The hidden layers perform complex transformations on the input data, allowing the network to learn complex patterns and relationships. The number of neurons in the input layer is based on the input size of the feature vectors. The number of hidden layers and the number of neurons in each layer are hyperparameters that need to be optimized based on the specific problem and the available data. The number of neurons in the output layer is based on the number of classes for classification (in a classification problem), or number of output variables required in regression problem. In your case the number of neurons in the input layer is 784, whereas the number of neurons in the output layer is 10.

1.3.3 Hyperparameters

Hyperparameters are parameters that are not learned during training but are set before training begins. They include the learning rate, batch size, and number of epochs.

Batch Size. Batch size refers to the number of training examples used to compute the gradients and perform a single update step. The larger the batch size, the faster the training process but also the higher the memory requirements. A larger batch size generally leads to better convergence but at the cost of longer training time and increased memory usage. A small batch size, on the other hand, can lead to faster convergence and better generalization, but at the cost of slower training times.

Epoch. An epoch is one complete pass through the entire training dataset. In other words, it is the number of times the entire training dataset is fed to the neural network during the training process. During each epoch, the neural network updates its weights and biases based on the errors in the predictions. The number of epochs needed for training depends on various factors such as the complexity of the problem, the size of the dataset, and the learning rate.

Learning Rate. The learning rate is a hyperparameter that determines the step size at which the weights and biases are updated during the training process. A high learning rate will result in larger updates to the weights and biases, which can lead to faster convergence but can also cause the algorithm to overshoot the optimal weights and biases. On the other hand, a low learning rate will result in smaller updates to the weights and biases, which can lead to slower convergence but may result in a more precise and stable solution.

1.3.4 Training a Feedforward Neural Network

The training of a feedforward neural network involves several steps explained one-by-one below.

1. **Define the Network Architecture:** The first step in training a feedforward neural network is to define the network architecture, which includes the number of input and output nodes, the number of hidden layers, and the number of neurons in each layer. The choice of network architecture depends on the specific problem and the available data.

2. **Prepare the Training Dataset:** The training dataset is typically split into training and validation sets. The training set is used to train the neural network, while the validation set is used to evaluate the performance of the network during training. The dataset is usually normalized or standardized to improve the training process and avoid biases.

3. **Choose Hyperparameters:** Hyperparameters are discussed in Section 1.3.3. Usually, you have to decide your hyperparameters based

on your problem and dataset. The learning rate determines how much the weights and biases are updated during each iteration of the training process. The batch size determines how many samples are used to compute the error and update the weights and biases. The number of epochs determines how many times the training dataset is processed during training. In this problem, the values of hyperparameters are given in Sec. 2.6.

4. **Forward Pass:** During the forward pass, the input data is fed to the neural network, and the output is calculated using the current weights and biases.

   (a) Input Layer: The input data is fed into the neural network's input layer. The input layer consists of neurons that are connected to the input features. Each neuron in the input layer receives one input feature.

   (b) Hidden Layers: The input data is then propagated through one or more hidden layers. Each hidden layer consists of neurons connected to the previous layer's neurons. The connections between neurons in different layers are represented by weights, which are initialized randomly before training begins. Each neuron in a hidden layer receives input from the neurons in the previous layer and produces an output using a non-linear activation function.

   (c) Output Layer: The output of the final hidden layer is propagated through the output layer. The output layer consists of neurons that are connected to the neurons in the final hidden layer. The number of neurons in the output layer depends on the number of classes or the number of regression targets. For example, if the task is binary classification, there will be one output neuron. If the task is multi-class classification, there will be multiple output neurons, one for each class.

   (d) Activation Function: An activation function is applied to the output of each neuron in the hidden and output layers. The activation function introduces non-linearity into the network, which allows the network to learn more complex relationships between the input and output.

The output is then compared to the actual output, and the error (or loss) is calculated using a loss function such as mean squared error, cross-entropy loss, or logistic regression loss. In this project, we only use cross-entropy loss (See Equation 3).

5. **Backward Pass:** The backward pass involves propagating the error backwards through the network and updating the weights and biases using the backpropagation algorithm. The backpropagation algorithm calculates the gradient of the error with respect to the weights and biases in each layer. The weights and biases are then updated using the gradient descent algorithm to minimize the error. This is explained in more detail in Section 1.6.

6. **Repeat:** Steps 4 and 5 are repeated for each batch of data until the entire training dataset is processed epoch number of times. The weights and biases are updated after each batch, and the total error (or loss) is calculated at the end of each epoch.

7. **Evaluate Performance:** After the training is complete, the performance of the network is evaluated using the validation (or test) dataset. The performance is measured using metrics such as accuracy, precision, recall, and F1 score. In this project, you have to evaluate only the "accuracy" of the model on the validation (or test) dataset.

8. **Make Predictions:** Once the network is trained and the performance is satisfactory, the model can be used to make predictions on new data. The input data is fed to the network, and the output is calculated using the trained weights and biases.

In summary, the forward pass is a crucial step in the training process for a feedforward neural network. It involves propagating the input data through the network, applying non-linear activation functions, and calculating the predicted output. The predicted output is then compared to the actual output to calculate the error, which is used to update the weights and biases during the backward pass.

1.4 Softmax Function

To obtain the predicted category in the classification problem, the machine learning algorithm needs to compute the probability distribution, which can be interpreted as the likelihood of each class being the correct classification for a given input.

To convert a vector into a probability distribution, the softmax function is used as the last layer of a neural network for classification tasks. The input of the softmax layer is a vector $(K \times 1)$ of raw scores or logits, representing the confidence of the network's prediction for each class. The softmax function is then applied to these logits, producing a normalized probability distribution vector $(K \times 1)$ over the class. Here, K is the number of all possible classes.

The softmax function is defined as follows:

$$P_i \equiv \mathrm{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)}, \tag{2}$$

where z_i is the i-th element of the input vector z of length K.

In essence, the softmax function exponentiates each input vector element and then normalizes the resulting values by dividing each exponentiated value by the sum of all exponentiated values. The resulting output is a probability distribution over the classes, where each element represents the probability of the input belonging to the corresponding class. The sum of the probabilities for all classes add up to 1. The functionality of softmax is illustrated in Figure 1.

In summary, the softmax function is a commonly used mathematical tool for classification tasks in machine learning and deep learning projects, allowing the neural network to output a probability distribution over the classes for a given input. Since we are solving a classification problem with a feedforward neural network, we should add softmax as the output layer of the neural network so that we can get the predicted category with the highest possibility.

1.5 Cross-Entropy Loss

A loss function, also known as a cost function or objective function, is a function that measures the difference between the predicted output of a machine learning model and the actual output. Training a machine learning model aims to minimize the loss function. In other words, we find the optimal

weights w and bias b of a feedforward neural network by minimizing the loss function.
In this project, we use the cross-entropy loss to evaluate the difference between the machine-learning predicted output vector and the exact labels of each image. As shown in Figure 1, the softmax function output is a $K \times 1$ probability distribution vector $\hat{\mathbf{y}}$ (where K is the number of all possible classes), in which the maximum element's index is the predicted number shown in the input image minus 1. Correspondingly, the exact label of the input image is stored in another $K \times 1$ one-hot encoding vector $\mathbf{y}$. Our model's prediction is perfectly accurate when the difference between $\mathbf{y}$ and $\hat{\mathbf{y}}$ is zero. We use the cross-entropy loss function in this project to evaluate the difference:

$$L(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{K} y_i \log(\hat{y}_i) \tag{3}$$

The cross-entropy loss measures how well the predicted probability distribution vector matches the actual probabilities. Therefore, a lower cross-entropy loss can stand for a better classification prediction. We try to minimize this loss function with the backpropagation algorithm stated in Section 1.6 during training process.

1.6 Backpropagation Algorithm

Backpropagation is a fundamental algorithm used in the field of machine learning, specifically in the training of artificial neural networks. It plays a crucial role in adjusting the weights and biases of a neural network in order to minimize the error or loss between the predicted output and the actual output. To understand the algorithm, we will look into both the forward pass and backward pass phases. We now work step-by-step through the mechanics of a feedforward neural network with two hidden layers to explain the details.

1.6.1 Forward Pass

Forward pass refers to the calculation and storage of intermediate variables (including outputs) for a neural network in order from the input layer to the output layer. As discussed in Section 1.3.4, during the forward pass, input data is fed through the network, and each neuron computes its output based on the weighted sum of its inputs and biases with an activation function.

The computed output is then passed to the subsequent layer until the final output is obtained. We will assume that tanh function, which is defined in Eq. 5, is used as the activation function.

For the sake of simplicity, let's assume that the input example is $\mathbf{X} \in \mathbb{R}^{P \times N}$ (where P is the dimension of input size, and N is the dimension of input data points) and our hidden layers include both weights and biases term. The forward pass proceeds as follows.

1. Input layer:

$$\mathbf{a}^{(1)} = \mathbf{X}$$

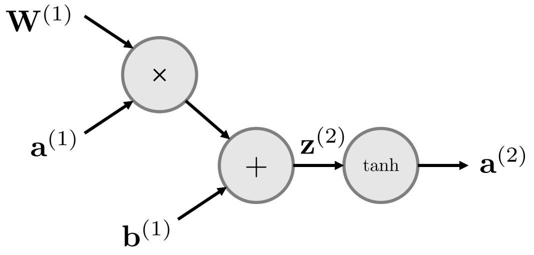

Figure 4: Input layer to hidden layer 1.

2. Hidden layer 1:

$$\mathbf{z}^{(2)} = \mathbf{W}^{(1)}\mathbf{a}^{(1)} + \mathbf{b}^{(1)}$$
$$\mathbf{a}^{(2)} = \tanh(\mathbf{z}^{(2)})$$

3. Hidden layer 2:

$$\mathbf{z}^{(3)} = \mathbf{W}^{(2)}\mathbf{a}^{(2)} + \mathbf{b}^{(2)}$$
$$\mathbf{a}^{(3)} = \tanh(\mathbf{z}^{(3)})$$

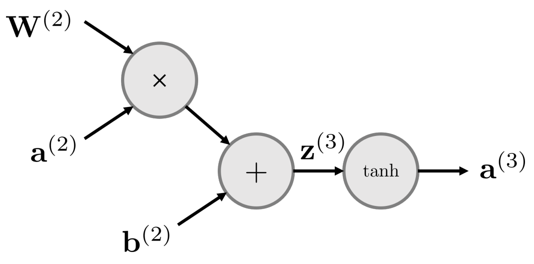

Figure 5: Hidden layer 1 to hidden layer 2.

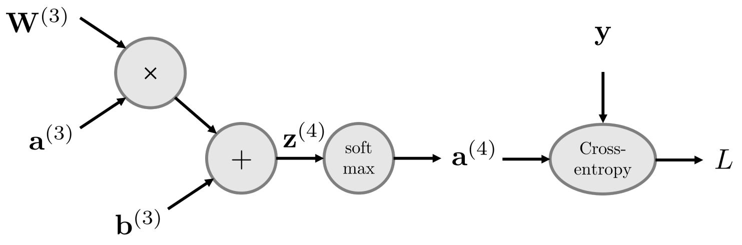

Figure 6: Hidden layer 2 to output layer.

4. Output layer:

$$\mathbf{z}^{(4)} = \mathbf{W}^{(3)}\mathbf{a}^{(3)} + \mathbf{b}^{(3)}$$
$$\mathbf{a}^{(4)} = \mathrm{softmax}(\mathbf{z}^{(4)})$$

Note: $\mathbf{a}^{(4)}$ is the predicted probability distribution vector $\hat{\mathbf{y}}$.

1.6.2 Backward Pass

Once the forward pass is complete, the backward pass begins. In this phase, the error or loss between the predicted output and the actual output is calculated. In this problem, assuming the cross-entropy loss function as Equation 3 and the exact label vector $\mathbf{y}$, we can then calculate the loss term as:

$$L(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{K} y_i \log(\hat{y}_i) \tag{4}$$

The algorithm then propagates this error backward through the layers, computing the gradient of the loss function with respect to each parameter in the network, 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 backward pass proceeds as follows.

1. Output layer:

$$\delta^{(4)} = \frac{\partial L}{\partial \mathbf{z}^{(4)}} = \mathbf{a}^{(4)} - \mathbf{y}$$

2. Hidden layer 2:

$$\begin{aligned}
\frac{\partial L}{\partial \mathbf{W}^{(3)}} &= \delta^{(4)} \frac{\partial \mathbf{z}^{(4)}}{\partial \mathbf{W}^{(3)}} = \delta^{(4)} \mathbf{a}^{(3)} \\
\frac{\partial L}{\partial \mathbf{b}^{(3)}} &= \delta^{(4)} \frac{\partial \mathbf{z}^{(4)}}{\partial \mathbf{b}^{(3)}} = \delta^{(4)} \\
\frac{\partial L}{\partial \mathbf{a}^{(3)}} &= \mathbf{W}^{(3)} \delta^{(4)} \\
\delta^{(3)} &= \frac{\partial L}{\partial \mathbf{z}^{(3)}} = \mathbf{W}^{(3)} \delta^{(4)} (1 - \tanh^2(\mathbf{z}^{(3)}))
\end{aligned}$$

3. Hidden layer 1:

$$\begin{aligned}
\frac{\partial L}{\partial \mathbf{W}^{(2)}} &= \delta^{(3)} \frac{\partial \mathbf{z}^{(3)}}{\partial \mathbf{W}^{(2)}} = \delta^{(3)} \mathbf{a}^{(3)} \\
\frac{\partial L}{\partial \mathbf{b}^{(2)}} &= \delta^{(3)} \frac{\partial \mathbf{z}^{(3)}}{\partial \mathbf{b}^{(2)}} = \delta^{(3)} \\
\frac{\partial L}{\partial \mathbf{a}^{(2)}} &= \mathbf{W}^{(3)} \delta^{(4)} \\
\delta^{(2)} &= \frac{\partial L}{\partial \mathbf{z}^{(2)}} = \mathbf{W}^{(2)} \delta^{(3)} (1 - \tanh^2(\mathbf{z}^{(2)}))
\end{aligned}$$

4. Input layer:

$$\begin{aligned}
\frac{\partial L}{\partial \mathbf{W}^{(1)}} &= \delta^{(2)} \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{W}^{(1)}} = \delta^{(2)} \mathbf{a}^{(1)} \\
\frac{\partial L}{\partial \mathbf{b}^{(1)}} &= \delta^{(2)} \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{b}^{(1)}} = \delta^{(2)}
\end{aligned}$$

The gradients are used to update the parameters through an optimization algorithm, such as gradient descent, which iteratively adjusts the weights and biases in the direction of minimizing the loss.

By iteratively performing forward and backward passes, backpropagation allows neural networks to learn from training data and adjust their parameters to improve their predictions. This process continues until the network reaches a state where the loss is minimized, indicating that it has learned the underlying patterns in the training data.

1.7 Gradient Descent Optimizer

Gradient descent is an optimization algorithm commonly used in conjunction with backpropagation to update the parameters of a neural network during the training process. It is designed to find the optimal values of the network's weights and biases that minimize the loss function.

In the context of backpropagation, the gradient descent algorithm works by computing the gradients of the loss function with respect to the network's parameters. These gradients represent the direction and magnitude of the steepest ascent or descent in the loss function's surface.

Usually, the gradient descent optimizer includes a learning rate ϵ, an initial parameter setting θ (weights and biases in a neural network), and a mini-batch size of m examples. The below processes are repeated until a stopping criterion is met:

1. Samples m examples from the training set, $\{\mathbf{X}_1, \mathbf{X}_2, ..., \mathbf{X}_m\}$ and their corresponding outputs $\{\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_m\}$.
2. Compute the gradient $\mathbf{g}$ according to backward pass (Section 1.6).
3. Update parameters:
$$\theta \leftarrow \theta - \epsilon \mathbf{g}$$

The update of each parameter is performed iteratively for a fixed number of epochs or until a convergence criterion is met. In each iteration, the parameters are adjusted by subtracting the learning rate multiplied by the corresponding gradient.

By using gradient descent in conjunction with backpropagation, neural networks are able to adjust their parameters in the direction that minimizes the loss function, allowing them to learn from training data and improve their predictions. Gradient descent is a key component of the training process and is crucial for optimizing the performance of neural networks in various machine learning tasks.

2 Problem Statement

2.1 Data Preparation

Write a function in MATLAB that loads the training data (`train_images.mat`, `train_labels.mat`) and testing data (`test_images.mat`, `test_labels.mat`). **The data files should be saved in the same directory as the function**. The image data, which is originally $[28 \times 28 \times N]$ should be reshaped to $[784 \times N]$ with normalized values from 0 to 1. The label data, which is originally $[1 \times N]$ should be one-hot encoded to $[10 \times N]$. Refer to the following template.

```matlab
function [X_train, Y_train, X_test, Y_test] = ...
    load_train_and_test_data()
% Reads training and testing data. The images are ...
    flattened out and normalized, labels are converted to ...
    one-hot encoded vector.
%     Inputs:
%         (None)
%     Outputs:
%     X_train: a  784 x N matrix representing the training ...
    images, where each column corresponds to a single ...
    image. N is the number of examples(images), which is ...
    60000. The pixel intensities are normalized, holding ...
    values from 0 to 1.
%     Y_train: a K x N matrix representing the labels of ...
    the training images, where each column corresponds to ...
    a one-hot encoding of a single image. K is the number ...
    of classes, which is 10. N is the number of examples, ...
    60000.
%      X_test: a 784 x N_t matrix representing the testing ...
    images, where each column corresponds to a single ...
    image. The pixel intensities are normalized, holding ...
    values from 0 to 1. N_t is the number of test ...
    examples, which is 100000.
%      Y_test: a K x N_t matrix representing the labels of ...
    the testing images, where each column corresponds to a ...
    one-hot encoding of a single image.
```

2.2 Model Construction

2.2.1 Weights Initialization

Write a function in MATLAB that initializes the weights and biases of the feedforward neural network. The weights are initialized from random values drawn from a standard normal distribution, which can be implemented using `randn(M, N)`, where `M` is the number of units of the next layer, and `N` is the number of units of the previous layer. Biases are initialized as zeros, using `zeros(M, 1)`. The output `parameters` should be a structure array. Refer to the following template.

```
function parameters = initialize_parameters(layer_dims)

% Initializes the weights and biases of the feedforward ...
    neural network
%       Inputs:
%   layer_dims: array of layer dimensions, including input ...
    and output layers
%       Output:
%   parameters: a struct containing W1, b1, W2, b2, etc.
% (Hint: parameters{1}.W should be a ( layer_dims(2) x 784 ...
    ) matrix, which is the initialized weights connecting ...
    the input layer and the first hidden layer. ...
    parameters{1}.b should be a ( layer_dims(2) x 1 ) ...
    array, which is the initialized biases connecting the ...
    input layer and the first hidden layer. )
```

2.2.2 Activation Function

Create functions in MATLAB that can apply the activation functions on each neuron's output to introduce non-linearity into the network, allowing it to model complex relationships between inputs and outputs. In this project, we should design the activation function **tanh2**. To ensure that your implementation is compatible, follow the interface given below.

- **tanh2**: The tanh (hyperbolic tangent) activation function is a mathematical function commonly used in artificial neural networks. It maps any input to a value between -1 and 1, and is defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{5}$$

The MATLAB function should be:

```
function Z = tanh2(X)
% Tanh applies the tanh activation function on the ...
    input to get a nonlinear output.
%    Inputs:
%           X: A M x N matrix representing the output ...
    of the neurons, which serves as the input of the ...
    activation function. M is the number of neurons, ...
    and N is the number of examples
%   Outputs:
%           Z: a M x N matrix representing the output ...
    after the tanh activation function
```

2.2.3 Softmax Function

The softmax function (as discussed in Section 1.4) is a mathematical function commonly used in a neural network's output layer for multiclass classification problems. 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. The softmax function is defined as:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}, \tag{6}$$

where x_i is the input to the function for class i, and the sum is taken over all classes $j \in K$.

```
function Z = softmax(X)
% softmax applies the softmax function on the input to get ...
    the corresponding probability distribution along all ...
    possible classes.
%    Inputs:
%           X: A K x N matrix. K is the number of all ...
    classes, and N is the number of examples.
%   Outputs:
%           Z: A K x N matrix representing the probability ...
    of the distribution. The sum of each row should equal ...
    to 1.
```

2.3 Training the Model

After you initialize the hyperparameters as asked in Section 2.2.1, you also need to initialize other hyperparameters i.e. epochs, batch size & learning rate within your MATLAB script.

2.3.1 Forward Propagation

In your MATLAB script start a for-loop that loops through all epochs and all batches within each epoch. Within each iteration, first, write a function called `forward_propagation` that takes in one batch of your data input `X` as the function input along with the weights and biases stored in the struct variable `parameters`. Then you pass the data input through your neural network architecture (see Figure 1 and Section 1.3) to get the predicted output along with all intermediate outputs after every layer. So the struct variable `activations` stores all intermediate values after each layer and finally stores the predicted output vector. Remember there is an activation function (tanh) after each layer except the last hidden layer after which there is a softmax layer. Please find the pseudocode for the required function in Algorithm 1.

```
function activations = forward_propagation(X, parameters)
% Computes the output of a feedforward neural network ...
    given input data and learned parameters
%       Inputs:
%            X: A P x N matrix. P is the input size (number ...
    of neurons for the current layer), and N is the number ...
    of examples.
%  parameters: learned parameters, a struct containing W1, ...
    b1, W2, b2, etc.
%       Output:
% activations: array of activations at each layer, ...
    including input and output layers
```

2.3.2 Loss Function

After you receive the predicted values from the forward pass you need to evaluate the loss. In this project, we are using cross-entropy loss (see 4). Write a MATLAB function that takes in the predicted values from the forward pass and the ground truth labels and evaluates the loss for each batch. Makes sure to evaluate the loss (or cost) for each batch, then take the average of all

Algorithm 1 Forward Propagation Function

```
function FORWARD_PROPAGATION($X$, parameters)
    $L \leftarrow$ length(parameters)
    $A \leftarrow X$
    activations $\leftarrow$ cell$(1, L+1)$
    activations[1] $\leftarrow X$
    for $l = 1$ to $L$ do
        $Z \leftarrow$ parameters$[l].W \cdot A +$ parameters$[l].b$
        if $l == L$ then
            $A \leftarrow$ softmax$(Z)$
        else
            $A \leftarrow$ tanh2$(Z)$
        end if
        activations$[l+1] \leftarrow A$
    end for
    return activations
end function
```

batches to store the loss (or cost) for each epoch in your training file so that you can plot the loss over epochs later.

```
function cost = compute_cost(AL, Y)
% The given function calculates the cross-entropy loss, ...
    also known as the log loss, given the predicted values ...
    AL and the true values Y
%   Inputs:
%       AL: final predicted values, a K x N matrix. K is ...
    the number of all classes, and N is the number of ...
    examples.
%        Y: ground truth labels, a K x N matrix. K is the ...
    number of all classes, and N is the number of examples.
%   Output:
%     cost: the entropy loss
```

2.3.3 Backpropagation

Write a function called `backward_propagation` that takes in one batch of your data input `X` and output `Y` as the function input along with the struct variables **parameters** and **activations**. Then you calculate the gradients according to the chain rule in Section 1.6. Please find the pseudocode for the required function in Algorithm 2

```
function gradients = backward_propagation(X, Y, ...
    parameters, activations)
% Computes the gradients of a feedforward neural network ...
    given input data, true labels, and learned parameters
%       Inputs:
%            X: A P x N matrix. P is the number of input ...
    features, and N is the number of examples.
%            Y: A K x N matrix. K is the number of all ...
    classes, and N is the number of examples.
%  parameters: learned parameters, a struct containing W1, ...
    b1, W2, b2, W3, b3.
% activations: array of activations at each layer, ...
    including input and output layers, computed using ...
    forward propagation
%      Outputs:
%   gradients: gradients of the cost with respect to each ...
    parameter, a struct containing dW1, db1, dW2, db2, ...
    dW3, db3.
```

Algorithm 2 Backward Propagation Function

```
function BACKWARD_PROPAGATION($X, Y$, parameters, activations)
    $L \leftarrow$ length(parameters)
    $m \leftarrow$ size($X, 2$)
    gradients $\leftarrow$ cell($1, L$)
    $dZ \leftarrow$ activations[end] $- Y$
    gradients[$L$].$dW \leftarrow dZ \cdot$ activations[$L$]$'/m$
    gradients[$L$].$db \leftarrow$ sum($dZ, 2$)$/m$
    for $l \leftarrow (L - 1)$ downto 1 do
        $dA \leftarrow$ parameters[$l + 1$].$W' \cdot dZ$
        $dZ \leftarrow dA \cdot (1 -$ tanh2(activations[$l + 1$])$^2$)
        if $l == 1$ then
            $A_{\text{prev}} \leftarrow X$
        else
            $A_{\text{prev}} \leftarrow$ activations[$l$]
        end if
        gradients[$l$].$dW \leftarrow dZ \cdot A'_{\text{prev}}/m$
        gradients[$l$].$db \leftarrow$ sum($dZ, 2$)$/m$
    end for
    return gradients
end function
```

2.3.4 Gradient descent

Write a function called `update_parameters` that takes the old weights and biases stored in the struct variable `parameters` and backward propagation `gradients` as the function input along with a user-specified `learning_rate` to update weights and biases.

```matlab
function parameters = update_parameters(parameters, ...
    gradients, learning_rate)
% Updates the parameters of a feedforward neural network ...
    using gradient descent
%        inputs:
%   parameters: learned parameters, a struct containing ...
    W1, b1, W2, b2, etc.
%    gradients: gradients of the cost with respect to ...
    each parameter, a struct containing dW1, db1, dW2, ...
    db2, etc.
% learning_rate: learning rate for gradient descent
%        output:
%   parameters: updated learned parameters, a struct ...
    containing W1, b1, W2, b2, etc.
```

2.4 Testing the model

2.4.1 Inference

Once you have trained your model, you can test the performance of your model on the test data. You are given the `predict_single_image()` function, which visualizes a random image from the test data and plots the probabilities of each class using a bar graph. The input to the function is `parameters`, a structure containing the trained weights and biases. Once your model is trained, `predict_single_image(parameters)` should output a result like Figure 7.

Create a function called `predict` that returns the predicted classes given an input image and parameters. Refer to the following template.

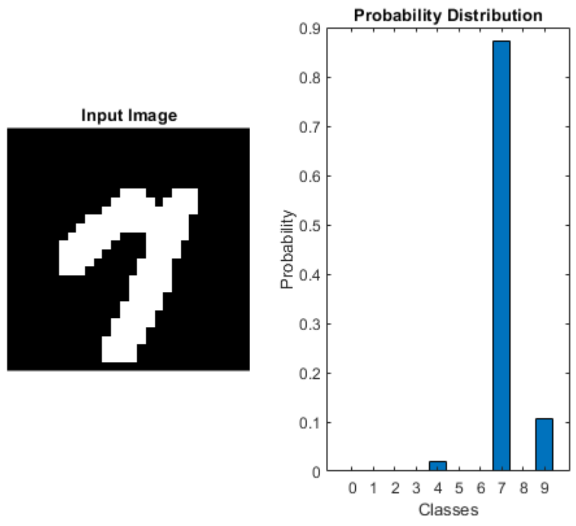

Figure 7: Inference results on a single test image.

```
function Y_pred = predict(X, parameters)

% Returns the predicted classes of the inference images ...
    using parameters containing the weights and biases of ...
    the neural network.
%   Inputs:
%        X: Inference images with sizes 784 x N. N is the ...
    number of images.
%   Output:
%   Y_pred: a K x N array containing the predicted labels ...
    of each input image, where K is the number of classes, 10.
%
% (Hint: The predicted class is the class which has the ...
    highest probability.  )
```

2.4.2 Calculating Accuracy

Create a function called `accuracy`, which returns the accuracy (%) given the predicted labels and actual labels. It is simply the ratio of number of samples predicted correctly by the total number of samples tested.

```matlab
function acc = accuracy(Y_pred, Y)

% Returns the accuracy, which is the percentage of correct ...
    predictions.
%    Inputs:
%   Y_pred: a K x N array containing the predicted labels ...
    of each input image. K is the number of classes, 10.
%        Y: a K x N array containing the actual labels of ...
    each input image.
%   Output:
%      acc: The percentage of accurately predicted samples.
```

2.5 Plot

Create a function called `visualize_history` that generates a plot with two subplots to show the relationship between training loss L_{train} vs. epochs E, and testing accuracy A_{test} vs. epochs E. Here, L_{train} is the cross-entropy loss of the current neural network's loss computed with `compute_cost` function with training data. A_{test} is the accuracy of the current model's prediction compared to the exact labels for the testing data. Use the function `subplot` to place the two plots on the same figure. More information about the usage of `subplot` can be obtained from https://www.mathworks.com/help/matlab/ref/subplot.html. Your function should programmatically save the generated figure as `model_lr_numLayer_epochs.png` file. Here, `lr` is the value of the learning rate, `numLayer` is the number of the hidden layers, and `epochs` is the number of epochs. Note that the learning rate values, epochs, and the number of layers should be expressed in a compact form. For example, if the epochs is 150, the learning rate is 0.01, and the number of hidden layers is 2, then the name should be `model_0.01_2_150.png`.
Embed the generated figures into your report. Be sure to label all your axes and give meaningful titles to your subplots. The information about the learning rate, the number of hidden layers, and the activation function should also be programmatically included in the title string. Hint: `sprintf` can be used to format a string.

The `visualize_history` function should have the following interface:

```
function visualize_history(epochs, train, testAccuracy, ...
    learning_rate, numLayer)
% visualize_history: a function that plots and saves the ...
    training loss and testing accuracy versus time under ...
    specific hyperparameters.
%        Inputs:
%        epochs: a integer shows the number of training epochs
%     trainLoss: an epochs x 1 vector, where ith element ...
    in the vector corresponds to the training loss after ...
    ith epochs training
%  testAccuracy: an epochs x 1 vector, where ith element ...
    in the vector corresponds to the testing accuracy ...
    after ith epochs training
% learning_rate: the value of learning rate
%      numLayer: the value of the number of hidden layers
%       Outputs:
%       This function has no outputs
```

2.6 Main Script

The main script titled `project_UID.m` is available at the end of the document. When you submit the codes, please replace `UID` with your numerical UID. In this main script contains how to construct a feedforward neural network model with the previously defined functions and how to train the models with different combinations of hyperparameters and evaluate their performance. Based on the comments in the main script to build the required functions. Finally, your script should visualize the plots to show the training loss vs. epochs and testing accuracy vs. epochs by executing your `visualize_history` functions. The main script should also print the training accuracy and testing accuracy in the command window.
The general structure of your script should follow the steps:

1. Load data from directory and preprocess.
2. Define network architecture and hyperparameters
3. Train the model.
4. Evaluate the model on the test set.

In this project, the combination of default hyperparameters are: number of hidden layers is 2; learning rate is 0.1; and training epochs is 150. The submitted codes should be executable with those hyperparameters defined. In addition, students are supposed to discuss the influence of those hyperparameters on the training progress. The following combinations of hyperparameters should be executed:

1. Run the codes with the training epochs as 50, 150, and 300 independently. Keep the learning rate as 0.01 and the number of layers as 2.

2. Run the codes with the learning rate as 0.1, 0.01, and 0.001. Keep the epochs as 150 and the number of layers as 2.

3. Run the codes with the number of layer as 2, 3, and 5. Keep the epochs as 150 and the learning rate as 0.01.

Within all the plots generated with the `visualize_history`, try to discuss the influence of those hyperparameters' influence on the training progress.

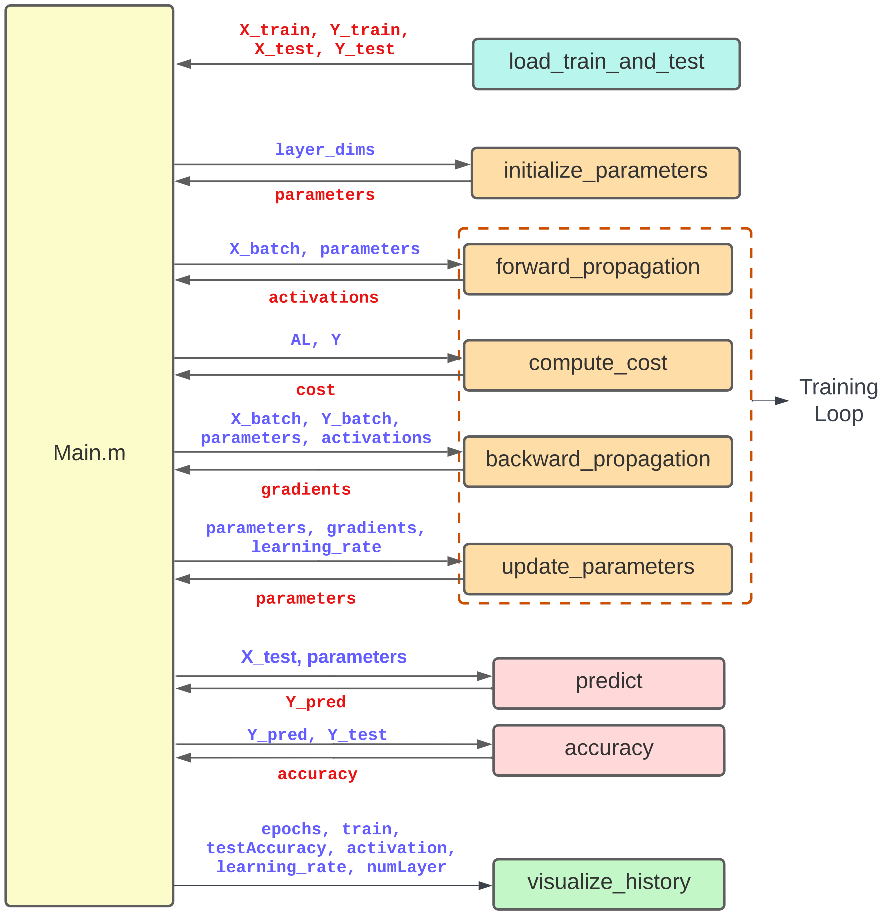

Figure 8: Overview of your code architecture

3 pcode Functions

To help you get started in this project, we have included the following MATLAB pcode files for all of the deliverables. These files are executable versions of the functions that you are required to implement but obfuscated so that you cannot read them. You may use them to test all aspects of your project working together. To use a pcode function, simply treat it as a regular MATLAB script file by placing it in your working directory. Note that when you have implemented your `.m` files, remove the `.p` files from your working directory so MATLAB knows to call your function instead.

- `load_train_and_test_data.p`
- `intialize_parameters.p`
- `forward_propagation.p`
- `backward_propagation.p`
- `update_parameters.p`
- `compute_cost.p`
- `predict.p`
- `accuracy.p`
- `tanh2.p`
- `softmax.p`
- `visualize_history.p`

Instruction of pcode usage
In the pcode folder, you can find a MATLAB script file named as `example_code.m` to practice with the basic usage of some functions mentioned above.

4 Deliverables

Submit the url to your GitHub repository in BruinLearn. Upload your .pdf report named as `HW3.pdf`, and all your code files in a folder named `HW3`.

4.1 Code (70%)

You should upload the following files zipped. If you used helper functions or scripts, you should also include those in your submission. Every file must include a few comment lines (your name, UID, filename, description) at the beginning. The description in this top comment section should be a few sentences on the inputs, outputs, and methods. The entire code should be properly.

- `load_train_and_test_data.m` [10 pts]
- `intialize_parameters.m` [5 pts]
- `forward_propagation.m` [10 pts]
- `backward_propagation.m` [10 pts]
- `update_parameters.m` [5 pts]
- `compute_cost.m` [5 pts]
- `predict.m` [5 pts]
- `accuracy.m` [5 pts]
- `tanh2.m` [5 pts]
- `softmax.m` [5 pts]
- `visualize_history.m` [5 pts]
- `project_UID.m` [available at the end of the document]
- `predict_single_image.m` [available at the end of the document]

The following deductions apply to all the scripts in your submission:

- Insufficient commenting (-25% per script)

- Failure to follow required function interfaces (-25% per script)
- Very inefficient implementation (-20% per script)
- Logic errors (-15% per instance)
- Unused variables (-5% per instance)
- Plots missing labels, titles, and legends (-5% per instance)
- Typos or lack of proof-reading (-5% per instance)
- Missing a few comment lines with your name, UID, filename, and description at the beginning of each file (-10% per script)

4.2 Report (30%)

Your report should be named `HW3.pdf` and document all the essential steps of your code, similar to the reports for the homework. Your report should be formatted as a journal article and divided into multiple sections and subsections (and, if necessary, a reference section). There is no page limit. It should read like a story with coherently organized sections, figures, and plots. Your report should also include any code that requires a detailed explanation. Be mindful of the following common pitfalls.

- Your report should have multiple plots, images, and schematics. Make sure to number these graphic elements (e.g., Figure 1, Figure 2, ...). Please include a caption for each of them. These elements must be referenced and described in the text of the report.
- Your report must be typed. Do not use images of handwritten equations or pictures of plots taken with a camera.
- Do not take screenshots of plots from MATLAB. Save the plot using `saveas()` as an image file in pdf/png/jpg/other format and then insert it into the report.
- Do not take screenshots of code snippets. Type it out in the report. Every code snippet (<5 lines) must be described in the report.
- Try to use pseudocodes, flowcharts, or schematics instead of MATLAB code in the report.

- Do not take screenshots of this document. If you are copying any part of this document to use in your report, please cite this document in your report. Any recognized citation style is acceptable.

You are also encouraged to take this opportunity to learn LaTeX, given the abundance of scientific notations. You are also highly encouraged to use `git` as version control for your files, and also `GitHub`[1] as a place to store your code and serve as a portfolio of your projects.

5 Available Codes

Here, we offer the contents for the `project_UID.m`, which is the main file and the function `predict_single_image.m`. In the main script, you can find how to organize all the functions together to complete the project. In the `predict_single_image.m`, you can find how to use this function to visualize the performance of your trained model.

[1]GitHub available at https://github.com/

Table of Contents

Main File

```
clear all; close all; clc;
```

Load Training and Testing Data

```
[X_train, Y_train, X_test, Y_test] = load_train_and_test_data();
```

Define network architecture and hyperparameters

```
input_size = size(X_train, 1); % input size of the FNN
output_size = size(Y_train, 1);% output size (number of classes) of the FNN
neurons = 64;  % neurons each layer
numLayer = 2;
lr = 0.01; % learning rate
epochs = 150; % epochs

layer_dims = zeros(1, numLayer + 2);
layer_dims(1) = input_size;
layer_dims(end) = output_size;
for i = 1:numLayer
    layer_dims(i+1) = neurons;
end
```

Train the model

```
parameters = initialize_parameters(layer_dims);

fprintf('Initialize cost somewhere so that we do not have warnings.\n');

% Train the model using mini-batch gradient descent
m = size(X_train, 2);
batch_size = 64;
num_batches = floor(m / batch_size);
trainLoss = zeros(epochs, 1);
testAccuracy = zeros(epochs, 1);
for i = 1:epochs
```

```
    % Shuffle the training data
    indices = randperm(m);
    X_train = X_train(:, indices);
    Y_train = Y_train(:, indices);

    % Initialize cost matrix
    cost = zeros(num_batches, batch_size);

    % Train on mini-batches
    for j = 1:num_batches
        X_batch = X_train(:, (j-1)*batch_size+1:j*batch_size);
        Y_batch = Y_train(:, (j-1)*batch_size+1:j*batch_size);
        forward_pass = forward_propagation(X_batch, parameters);
        cost(j,:) = compute_cost(forward_pass{end}, Y_batch);
        gradients = backward_propagation(X_batch, Y_batch, parameters,
 forward_pass);
        parameters = update_parameters(parameters, gradients, lr);
    end

    Y_pred = predict(X_test, parameters);
    acc = accuracy(Y_pred, Y_test);
    % Print the cost each epoch
    fprintf('Loss after epoch %d: Training: %f\n', i, norm(cost));
    trainLoss(i) = norm(cost);
    testAccuracy(i) = acc;
end
```

Evaluate the model on the test set

```
fprintf('Test accuracy: %f\n', testAccuracy(end));
```

Visualize the training progress

```
visualizeHistory(epochs, trainLoss, testAccuracy, lr, numLayer);
```

Published with MATLAB® R2022b

```matlab
function predict_single_image(parameters)
% Visualizes the input image and plots a bar graph of probabilities for
% each class
% Inputs:
%         parameters: a struct containing the weights and biases (W1, b1,
%         W2, b2, etc.)
% Output:
%         (None)

    % Load data
    load('test_images.mat');
    index = randi(length(pixel));
    X = reshape( pixel(:,:,index), [size(pixel, 1)*size(pixel, 2), 1]) / 255;

    forward = forward_propagation(X, parameters);
    probabilities = forward{end};

    figure();
    subplot(1, 2, 1)
    imshow(pixel(:,:,index))
    title('Input Image')
    subplot(1, 2, 2)
    bar(0:9, probabilities)
    xlabel('Classes');
    ylabel('Probability');
    title('Probability Distribution')

end
```

Published with MATLAB® R2021b