Implementaton of Feedforward Neural Network for Number Classification Problem

Philip Ng

***Abstract*— Neural networks are becoming increasingly important in the era of big data. Tasks once deemed impossible due to large data sizes are now feasable with the advent of neural networks. and machine learning. In this project, we demonstrate the implementation of a feedforward neural network to solve a classification problem, which involves identifying the digits 0 through 9 from raw data. The identification was successful, and the results of the classification are shown in the results. Such mechanisms can be extended to solve more complex problems, such as the classification of human faces and various objects.**

I. INTRODUCTION AND THEORY

Artificial neural networks are important in developing machine learning and deep learning. They help with tasks like recognizing images, understanding speech, and processing language. In this paper, we look closely at the basic structure of two important types of artificial neural networks: single-layer perceptrons and feedforward neural networks.

First, one must understand single-layer perceptron. It has a simple structure with just one layer of neurons. Each neuron calculates a sum based on inputs and creates an output. The activation function, like Sigmoid or ReLU, applies a threshold to this sum to produce the output. The formula $y = \sigma(\sum w_i x_i + b)$ shows how the activation function, connection weights, inputs, and bias terms all interact. In this case, we use the tanh2 function, which will be defined later.

Next, we look at feedforward neural networks. These have an input layer, hidden layers, and an output layer. Data flows in one direction, from input to output. The structure and number of neurons in each layer depend on what the problem is (like classification or regression) and the data available.

We also emphasize the role of hyperparameters. These are parameters that stay the same during training, including the learning rate, batch size, and number of epochs. They directly affect different parts of the training process and how well the network performs.

Finally, we outline all the steps involved in training a feedforward neural network. This includes defining the network structure, preparing the training data, choosing hyperparameters, doing forward and backward passes, and evaluating performance. The aim is to help people understand how artificial neural networks work. We hope to shed light on the complex algorithms that are driving changes in machine learning and deep learning applications.

This project focuses on employing a feedforward neural network for classification tasks, specifically for digit recognition in grayscale images (28x28 pixels). The images are first converted into a 784-length vector, which is then input into the neural network. The network, utilizing multiple hidden layers and activation functions, outputs a vector of length 'K' (10 in this context, representing digits 0-9). The output vector is distinct in having a single non-zero element, the index of which indicates the predicted digit.

II. GENERAL IMPLEMENTATION

For this model, we are given four sets of data, the analysis of which requires several helper functions. The mechanisms of each function are detailed in this section.

A. `load_train_and_test_data()`

This function loads and preprocesses training and testing data for a machine learning model. First, the function loads the testing and training image and label data from separate files. Next, it determines the dimensions of the training and testing images using the size function. Then, it reshapes the training images into a 2D matrix with the number of rows equal to the total number of pixels in each image and the number of columns equal to the number of channels. The pixel values are normalized by dividing by 255 to ensure they are in the range of 0 to 1. This process is repeated for the testing images as well.

After that, the training labels are converted to categorical format using the categorical function to represent the different classes. The training labels are then one-hot encoded using the onehotencode function to convert them into a binary matrix representation. Similarly, the testing labels are converted to categorical format and then one-hot encoded.

Finally, the preprocessed training and testing data, along with their corresponding labels, are returned as output variables Xtrain, Ytrain, Xtest, and Ytest, respectively. Below is the pseudocode:

B. `initialize_parameters(layer_dims)`

This function initializes the parameters (weights and biases) for each layer in a neural network. First, the function

Algorithm 1 Load Train and Test Data

function LOADTRAINANDTESTDATA
    Load 'testimages.mat' into testimages
    Load 'testlabels.mat' into testlabels
    Load 'trainimages.mat' into trainimages
    Load 'trainlabels.mat' into trainlabels
    Find size of trainimages and testimages
    Reshape trainimages and testimages, and divide by 255
    Convert trainlabels and testlabels into categories
    One-hot encode the categories in trainlabels and testlabels
    Return reshaped images and one-hot encoded labels as output
end function

determines the total number of layers by getting the length of the layerdims input. It initializes the parameters as a cell array with two elements. Next, the function iterates through each layer from 1 to numlayers - 1. For each layer, it initializes the weights (W) randomly using a standard normal distribution. The size of the weight matrix for the current layer is determined by layerdims(i+1) (number of neurons in the current layer) and layerdims(i) (number of neurons in the previous layer).

The biases (b) for the current layer are initialized as zeros. The size of the bias vector for the current layer is determined by layerdims(i+1). Finally, the initialized parameters are stored in the cell array and returned as the output of the function. Below is the pseudocode for the function:

Algorithm 2 Initialize Parameters

function INITIALIZEPARAMETERS(layerdims)
    Set numlayers as the total number of elements in layerdims Initialize parameters as a 2-element list
    for each index i from 1 to numlayers - 1 **do**
        Set parameters[i].W as a random matrix with dimensions (layerdims[i+1], layerdims[i])
        Set parameters[i].b as a zero matrix with dimensions (layerdims[i+1], 1)
    end for
end function

C. `tanh2(X)`

In this project, we utilize the tanh (hyperbolic tangent) activation function. The tanh function has a unique property where it can convert any input into a value that falls within the range of -1 and 1. The formula used is shown below:

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

First, the function takes the output of the neurons, represented by a matrix X, as an input, and generates a nonlinear output. This matrix, X, is an M x N matrix where M is the number of neurons, and N is the number of examples. Applying the tanh function on this matrix results in another matrix Z of the same dimensions, M x N. This matrix Z represents the output after the tanh activation function has been applied. Below is the pseudocode:

Algorithm 3 Get tanh2

function T(a)nh2(X)
    Set Z as an empty matrix of the same dimensions as X
    for each element x in X **do**
        Compute z = 2 / (1 + exp(-2*x)) - 1
        Append z to Z
    end for
    Return Z
end function

D. `softmax(Z)`

The softmax function calculates the softmax activation for each element in the input matrix X. It ensures that the output values are in the range of 0 to 1 and sum up to 1, making them suitable for probability distribution interpretation. The softmax equation is as follows:

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

The function takes an input matrix X and calculates the softmax of each element in X using the exp and sum functions. The resulting matrix Z contains the softmax values corresponding to each element in X.

The pseudocodes of the functions mentioned in the previous section are detailed here.

E. 5`forward_propagation(X, parameters)`

The forward propagation function takes two inputs: an input data matrix X and a set of parameters that include weights and biases for each layer of a neural network. It performs the forward propagation process, which calculates the activations of each layer in the network.

First, it determines the number of layers in the network by getting the length of the parameters array. Then, it creates an empty cell array called activations with a size equal to the number of layers. The input data matrix X is set as the first element of the activations array.

Next, the function iterates through each layer of the network. For each layer, it retrieves the corresponding weights and biases from the parameters. It then performs a linear transformation by multiplying the weights with the activations from the previous layer and adding the bias term. This calculation is stored in a variable called Z.

Depending on whether the current layer is a hidden layer or the output layer, the function applies the appropriate activation function. If it's a hidden layer, it applies the tanh2 activation function to the Z values. If it's the output layer, it applies the softmax activation function. The resulting activation values are stored in the activations array at the corresponding layer index.

F. `backward_propagation(X,Y,parameters, activations)`

The backwardpropagation function takes four inputs: an input data matrix X, the corresponding target values Y, the parameters containing weights and biases for each layer of a neural network, and the activations computed during the forward propagation step. It performs the backward propagation process, which calculates the gradients of the parameters with respect to the loss function.

First, the function determines the number of layers in the network by getting the length of the parameters array. It also retrieves the number of training examples m from the size of the input data matrix. An empty cell array called gradients is created to store the gradients of the parameters.

The function calculates the gradient of the last layer by subtracting the target values Y from the activations of the last layer (activations(L+1)). This gradient is stored in a variable called dZ. Then, the gradients of the weights and biases for the last layer are computed and stored in the gradients cell array.

Next, the function iterates backwards through the layers, starting from the second-to-last layer (L-1) down to the first layer (1). For each layer, it computes the gradient of the activations of the previous layer (Aprev) by multiplying the weight matrix of the current layer (parameters(i+1).W) with dZ.

The function then calculates the new dZ by element-wise multiplication of dA with the derivative of the tanh2 activation function applied to the activations of the current layer (activations(i+1)). If it's not the first layer, the activations of the previous layer (Aprev) are set to the activations at index i in the activations cell array. Otherwise, it is set to the input data matrix X. The gradients of the weights and biases for the current layer are then computed and stored in the gradients cell array. After iterating through all the layers, the function returns the gradients cell array, containing the gradients of the weights and biases for each layer.

G. `compute_cost(AL,Y)`

The compute cost function takes two inputs: the predicted activations AL from the output layer of a neural network, and the corresponding target values Y. It calculates the cost, which represents the discrepancy between the predicted activations and the target values.

First, the function determines the number of training examples m by getting the size of AL along the second dimension. The cost is then computed by taking the negative average of the element-wise multiplication between Y and the logarithm of AL, summed over all elements. This measures the dissimilarity between the predicted activations and the target values. The computed cost is returned as the output of the function.

H. `update_parameters(parameters, gradients, learning_rate)`

The update parameters function takes three inputs: the current set of parameters, the gradients computed during backpropagation, and the learning rate. It updates the parameters of each layer in the neural network based on the gradients and learning rate. First, the function determines the number of layers in the network by getting the length of the parameters array. Then, it iterates through each layer from 1 to L-1. For each layer, it updates the weights and biases by subtracting the learning rate multiplied by the corresponding gradients. Finally, the updated parameters are returned as the output of the function.

I. `predict(X, parameters)`

The predict function takes two inputs: the input data X and the parameters of a trained neural network. First, the function performs forward propagation on the input data using the forwardpropagation function. This computes the activations of each layer in the network. Next, the final activations, which represent the predicted outputs, are extracted from the activations cell array. The end index is used to access the activations of the last layer. The activations are then converted from a cell array to a numeric array using cell2mat, and then cast to a double type. The predicted values, Ypred, are returned as the output of the function.

J. `accuracy(Y_pred, Y)`

The accuracy function calculates the accuracy of a prediction by comparing the predicted values Ypred with the true labels Y. The function first initializes a variable 'correct' to keep track of the number of correct predictions. The total number of samples is determined by getting the length of the true labels array Y. Next, the function iterates through each sample using a for loop. For each sample, it checks if the predicted value at index i matches the true label at the same index. If they match, the count of correct predictions is incremented by 1.

After iterating through all the samples, the accuracy is calculated by dividing the number of correct predictions by the total number of samples and multiplying by 100 to obtain a percentage. The calculated accuracy acc is returned

as the output of the function.

K. `visualize_history(epochs, trainLoss, testAccuracy, learning_rate, numLayer)`

The visualize history function generates a plot with two subplots to display the relationship between training loss and epochs, as well as testing accuracy and epochs. In the first subplot, the training loss is plotted against the epochs using a blue line. The x-axis is labeled as ”Epochs”, and the y-axis is labeled as ”Training Loss”. The subplot title includes the learning rate and the number of hidden layers.

In the second subplot, the testing accuracy is plotted against the epochs using a red line. The x-axis is labeled as ”Epochs”, and the y-axis is labeled as ”Testing Accuracy”. The subplot title also includes the learning rate and the number of hidden layers. The figure, containing the two subplots, is then saved as a PNG file with a filename generated using the provided learning rate, number of hidden layers, and number of epochs.

III. Main Code

The main code puts all the functions together. The code begins by clearing the workspace, closing any open figures, and clearing the command window. Next, it loads the training and testing data using the loadTrainAndTestData function, which returns the training and testing datasets (Xtrain, Ytrain, Xtest, Ytest).

The network architecture and hyperparameters are defined. The input size is determined as the number of features in the input data (Xtrain), and the output size is determined as the number of classes in the target labels (Ytrain). The number of neurons per hidden layer is set to neurons, and the total number of hidden layers is set to numLayer. The learning rate (lr) and number of epochs (epochs) are set.

The layerdims array is initialized with the dimensions of each layer in the network. The first element is set to inputsize, and the last element is set to outputsize. The number of neurons (neurons) is used to set the dimensions of the hidden layers.

The model is trained using mini-batch gradient descent. The parameters are initialized using the initializeparameters function. The cost is initialized as a matrix to store the cost of each mini-batch. The training data is shuffled, and then the model is trained on mini-batches. The forward propagation, cost computation, backward propagation, and parameter update steps are performed for each mini-batch. The training loss (cost) and testing accuracy (acc) are computed after each epoch.

The training loss and testing accuracy are stored in the trainLoss and testAccuracy arrays, respectively. After the training process, the final test accuracy is printed. The training progress is visualized using the visualizeHistory function, which plots the training loss and testing accuracy against the epochs. Lastly, the predictsingleimage function is defined, which visualizes a randomly selected image from the test dataset and plots a bar graph of the predicted probabilities for each class.

Overall, the code loads the training and testing data, defines the network architecture and hyperparameters, trains the model using mini-batch gradient descent, evaluates the model on the test set, visualizes the training progress, and provides a function to predict and visualize the probabilities for a single image.

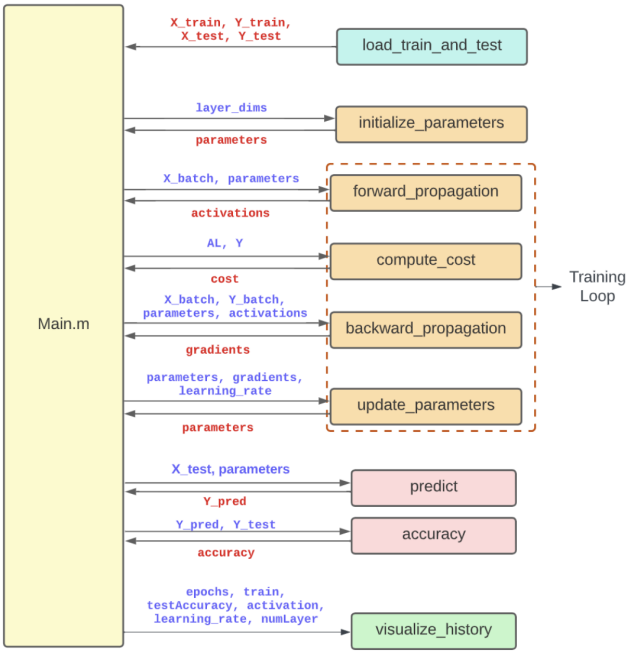


Fig. 1: Flowchart visualizing interaction between each function and the main code.[1]

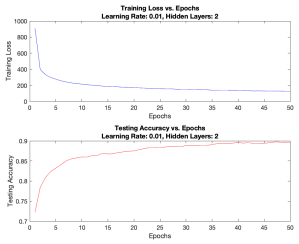
IV. Results and Discussion

In this study, we will analyze the effects of the number of epochs, the learning rate, and the number of hidden layers on the training loss and accuracy of the feedforward neural network. The accuracy is simply the ratio of the number of samples predicted correctly over the total number of samples analyzed. The training loss, also known as cross entropy loss, can be calculated ffrom the following formula:

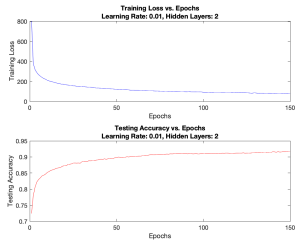
$$L(y, y') = -\sum_{i=1}^{k} y_i log(y'_i)$$

First, we studied the effect of the number of epochs on the loss and accuracy of the neural network. Below are the results:

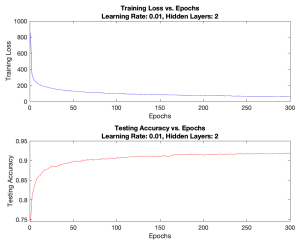
Studying the figure closely, it appears that as the number epochs increases, the cross-entropy loss decreases toward



(a) epochs = 50

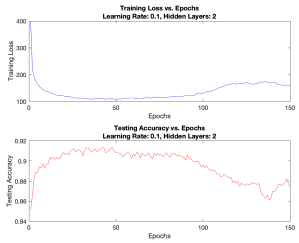


(b) epochs = 150

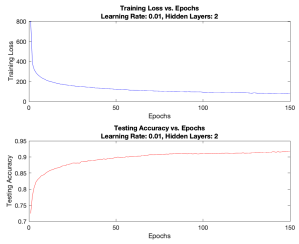


(c) epochs = 300

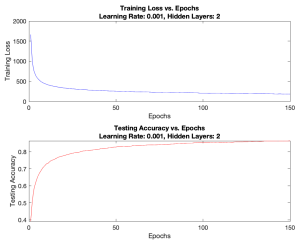
Fig. 2: Training loss and accuracy plots, varying number of epochs



(a) lr = 0.1



(b) lr = 0.01



(c) lr = 0.001

Fig. 3: Training loss and accuracy plots, varying learning rate

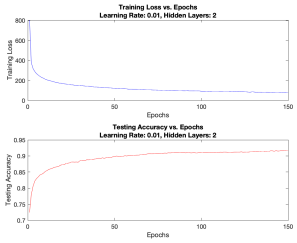
0, and the accuracy increases, approaching 1. This is reasonable, as the number of epochs determines the amount of time that the neural network can ”learn.” The more times the data is passed back and forward through the neural network, the higher the data accuracy will be.

The following analysis depicts the effect of learning rate with neural network accuracy:

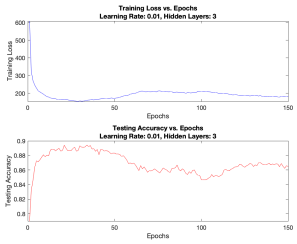
For higher learning rates (i.e. lr = 0.1), the simulation converges faster (20 epochs) and reaches a near-100% accuracy, but overshoots the optimal values after 50 epochs. For the lower learning rates, the model converges slower but reaches near-100% accuracy at much later epochs. For example, when the learning rate is set to 0.001, the simulation's accuracy is 0.9896. Thus decreasing learning rate causes slower convergence, but increasingly accurate values at higher epochs.

Finally, we analyze the effect of the number of hidden layers on the accuracy and loss of the neural network.

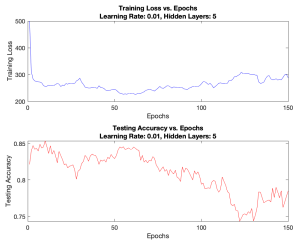
The correlation between accuracy and hidden layers is more complex. We observe the accuracy of the simulation becoming more noisy as additional hidden layers are added. Because there are more layers, more computational intensity is required to produce a meaningful result; more data must be given, and more training time is required. Thus running the simulation with more layers, but the same learning rate and number of epochs, caused excessive noise. Moreover, neural networks with more hidden layers are prone to ”overfitting,” which occurs when the network begins to train data from noise, rather than larger, more meaningful patterns. This can explain the increased noise and decreased accuracy of the simulations involving more hidden layers. Finally, the gradients in simulations with more hidden layers can become very large, making them prone to ”explosion,” causing the neural network to become unstable and even diverge. Overall, larger quantities of data and larger durations of training are required for neural networks with more hidden layers.



(a) hidden layers = 2



(b) hidden layers = 3



(c) hidden layers = 5

Fig. 4: Training loss and accuracy plots, varying number of hidden layers

V. CONCLUSION

In conclusion, this project involved the development and implementation of a neural network model for a classification task. The model was trained and evaluated using a training and testing dataset. The key components of the project included forward propagation, backward propagation, parameter initialization, training, prediction, and evaluation. The project utilized these components to train the model, evaluate its performance, and visualize the training progress. Through the project, important concepts such as cross-entropy loss, accuracy calculation, and parameter updates were applied. The project provides a foundation for further exploration and experimentation in the field of neural networks and machine learning.

Throughout the project, these functions were utilized to train a neural network model, evaluate its performance on a test set, and visualize the training progress. We observed that increasing the epochs incrwased the accuracy of the neural network. Increasing the learning rate caused the simulation to converge faster at the risk of overshooting the optimal accuracy. Increasing the number of hidden layers can potentially provide more meaningful conclusions on larger datasets, but require more training time and computational intensity to achieved the desired result. The project provided valuable insights into the implementation and usage of neural networks for classification tasks.

Overall, this project demonstrates the essential components of a neural network model, including forward and backward propagation, parameter updates, and evaluation. The implemented functions can serve as a foundation for further development and experimentation in the field of neural networks and machine learning.

REFERENCES

[1] K. Jawed, Discretized Structures Notes.Structures Computer Interaction Lab. 2023.