# EEE 443/543 - Project #5

## Introduction

In this project the aim was to design and train a multi-layer neural network for digit classification using the back-propagation algorithm similar to Project #4. MNIST database is used for this project as in the Project #3.

## Methodology

First, the input and output layer dimensions are determined. The MNIST database, includes 60k train images and 10k test images with labels. The size of each image is 28×28 pixels, so the input layer has 784 nodes. There are 10 digits to classify {0,1,…, 9} therefore there should be 10 nodes in the output layer. For each input image, the network produces a probability distribution over the digits. The index with the maximum probability is then chosen as the predicted digit.

In this project at least 95% accuracy rate is requested. This accuracy rate is so high that it is difficult and maybe impossible to achieve with a single layer network. Therefore, a hidden layer with adjustable number of neurons are implemented to the network architecture. After trials with different sizes, the number of neurons are determined as 100 because 100 neurons in a hidden layer produced the best results without overly increasing the computational complexity. Since the required threshold is achieved with 1 hidden layer with 100 neurons another layer is not added to architecture.

The activation function of the hidden layer neurons determined as ***tanh***. This is because it produces both negative and positive outputs in range [-1, 1] which enables the network to learn more complex features. Also, the gradient of ***tanh*** is relatively smooth and produces non-zero gradients for wide range of input sand it has a steep slope around 0, which helps propagate gradients effectively during backpropagation and avoids issues like vanishing gradients when compared to the ***sigmoid*** function.

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

The activation function of the output layer neurons determined as ***softmax(x)*** because it converts the raw outputs into a probability distribution over the classes, making it ideal for classification tasks. Also, ensures that the sum of the output values is 1, which is necessary when dealing with multiple classes and allows for easy interpretation of the model's predicted class by selecting the one with the highest probability.

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

The loss function of the back-propagation algorithm is chosen as **cross-entropy loss**, which is commonly used for classification problems, particularly when the outputs are probabilities. For multi-class classification, this loss function calculates the cross-entropy between the true label distribution and the predicted label distribution. The goal of training is to minimize this

loss function, which implies the network is trying to make the predicted probabilities as close as possible to the true probabilities.

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_i^c \log(\hat{y}_i^c)$$

- N is the number of samples,
- C is the number of classes,
- $y_i^c$ is the one-hot encoded true label for sample i in class c,
- $\hat{y}_i^c$ is the predicted probability for sample i in class c.

The number of epochs is limited to 30 because no significant improvement was observed after that point. While fewer epochs could have been used, 30 epochs were chosen to show the stability of the network after some point on the graph.

The initial batch size was set to 64, but it was too large to produce accurate results. I gradually decreased it to 32 and then to 16. A batch size of 16 provided the best accuracy while still being sufficiently fast.

The learning rate of 0.01 worked best on the first try. However, other values are also tested: a larger learning rate of 0.1 caused the model to diverge, while a smaller learning rate of 0.05 was too small, causing slow convergence. Both alternatives resulted in less accurate outcomes.

Momentum was added to the training process to help accelerate convergence and reduce oscillations during gradient descent. It works by adding a fraction of the previous weight update to the current update, which helps the model escape local minima and improves the stability of the learning process. Since we are dealing with a deep neural network, a momentum = 0.9 is added to speed up training and ensure more stable convergence. The mathematical formula for gradient descent with momentum is as follows:

$$v_t = \beta v_{t-1} - \eta \nabla J(w)$$
$$w = w - v_t$$

- $v_t$ is the momentum at time step t,
- $v_{t-1}$ is the previous momentum,
- $\beta$ is the momentum term,
- $\nabla J(w)$ is the current gradient,
- $\eta$ is the learning rate.

## Pseudocode

```
# Load MNIST data
Function load_mnist_images(filename):
    Open the file in binary read mode
    Skip the first 16 bytes (header)
    Read the remaining file data into a buffer
    Convert the buffer into a numpy array of uint8 values
    Reshape the array to have 784 columns (representing 28x28 images)
    Normalize the images by dividing by 255 to scale pixel values between 0 and 1
    Return the processed images
```

```
Function load_mnist_labels(filename):
    Open the file in binary read mode
    Skip the first 8 bytes (header)
    Read the remaining file data into a buffer
    Convert the buffer into a numpy array of uint8 values representing the labels
    Return the processed labels

# Softmax function
Function softmax(x):
    Subtract the maximum value in each row of x for numerical stability
    Compute the exponential of each element in x
    Normalize each row by dividing by the sum of its elements
    Return the softmax output

# Cross-entropy loss function
Function cross_entropy_loss(y_true, y_pred):
    Compute the negative log of the predicted probabilities
    Multiply by the true labels
    Return the average loss over all samples

# Classification error function
Function classification_errors(y_true, y_pred):
    Compare the predicted labels with the true labels
    Return the count of incorrect classifications

# Neural Network Class
Class NeuralNetwork:
    Initialize the neural network with:
        Input size, hidden size, output size, learning rate (eta), number of epochs, batch size, and momentum
        Randomly initialize weights and biases for input-hidden and hidden-output layers
        Initialize velocity for momentum-based gradient descent

    Function forward(x):
        Compute the hidden layer input by multiplying the input by weights and adding biases
        Apply the tanh activation function to compute hidden layer output
        Compute the output layer input by multiplying hidden output by weights and adding biases
        Apply softmax activation function to compute output layer probabilities
        Return the output probabilities

    Function backward(x, y_true):
        Compute the error at the output layer by subtracting the predicted output from true labels
        Compute the error at the hidden layer using the output error and the derivative of the tanh function
        Compute the gradients of weights and biases for both layers
        Update the velocities using momentum and learning rate
        Update the weights and biases using the gradients and velocities

    Function train(train_images, train_labels, test_images, test_labels, epochs, batch_size):
        For each epoch:
            Initialize error counters and loss accumulators for training
            For each batch of training data:
                Forward propagate the input through the network
                Compute the loss using cross-entropy
                Compute classification errors
                Perform backward propagation to update weights and biases
            After processing all batches:
                Log the errors and losses for the current epoch
                Evaluate the network on the test data and log results
```

```
Function evaluate(images, labels):
    For each batch of images:
        Forward propagate the input through the network
        Compute the loss and classification errors
    Return the total loss and classification errors

Function test(test_images, test_labels):
    For each test image:
        Forward propagate the input through the network
        Compare the predicted label with the true label
    Compute and print the accuracy of the network

# Main Code
Load training and testing images and labels
Initialize the neural network
Train the network using training data
Plot the classification errors and loss curves for training and testing
Evaluate the network's performance on the test set and print the accuracy
```

## Results

The neural network is trained and tested with chosen parameters and the results are recorded.

```
Epoch 0/30, Train Loss: 1265.3387029686262, Train Errors: 5826, Test Loss: 61.627767381268306, Test Errors: 623
Epoch 5/30, Train Loss: 192.56243321742662, Train Errors: 834, Test Loss: 25.580403005898802, Test Errors: 250
Epoch 10/30, Train Loss: 77.20846485751045, Train Errors: 234, Test Loss: 23.471700751718473, Test Errors: 233
Epoch 15/30, Train Loss: 34.98579528367504, Train Errors: 48, Test Loss: 23.70696059152812, Test Errors: 213
Epoch 20/30, Train Loss: 18.783668072429858, Train Errors: 8, Test Loss: 23.99668088813391, Test Errors: 204
Epoch 25/30, Train Loss: 12.109101837584754, Train Errors: 1, Test Loss: 24.23583385045069, Test Errors: 207
Test Accuracy: 97.95%
```

*Fig.1 Test Accuracy of the Final Algorithm*

*Fig.1* depicts that in the final design a network accuracy of 97.95% has achieved.

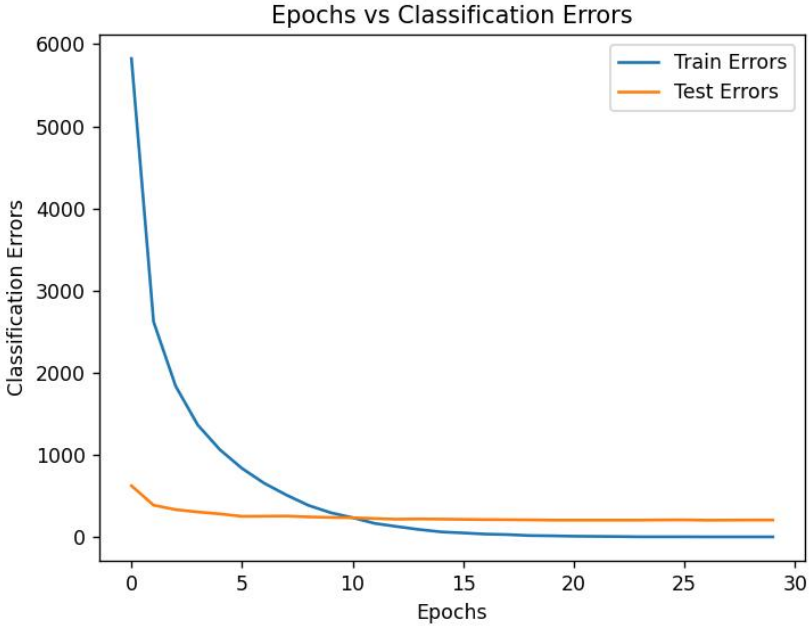


*Fig.2 Classification Errors after each epoch for both train and test sets*

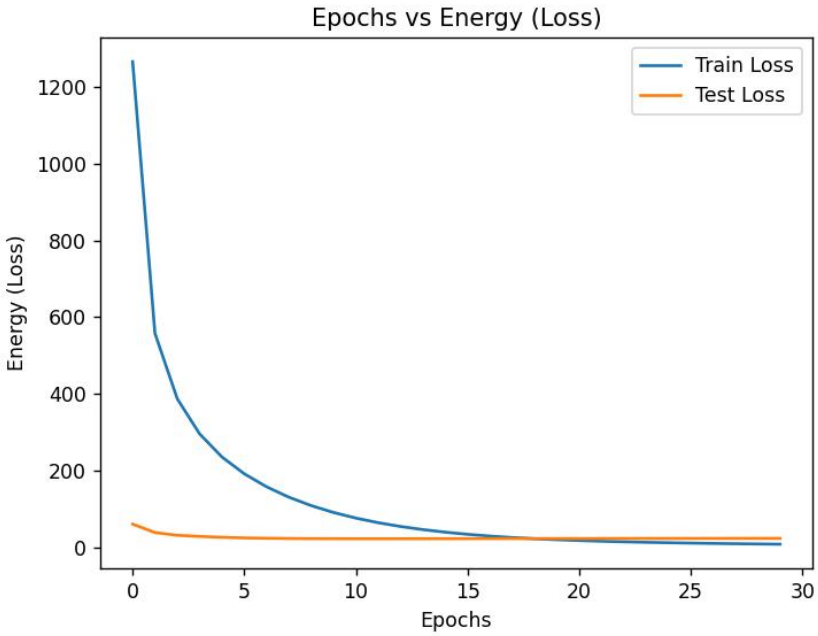


*Fig.3 Energy (Loss) after each epoch for both train and test sets*

From *Fig. 2&3* it can be seen that, after 15 epochs the loss and classification errors are almost converged to zero.

**Conclusion**

In this project, we successfully designed and implemented a multi-layer neural network for digit classification using the MNIST dataset. By employing the back-propagation algorithm, we achieved a final test accuracy of 97.95%, which exceeded the target of 95% accuracy. The architecture consisted of an input layer with 784 neurons, a single hidden layer with 100 neurons, and an output layer with 10 neurons representing the digits from 0 to 9.

Through various experiments, we fine-tuned the network parameters, including the batch size, learning rate, and momentum, to optimize performance. The use of the tanh activation function in the hidden layer and softmax in the output layer, coupled with the cross-entropy loss function, enabled the network to effectively learn and classify digits. The inclusion of momentum helped accelerate convergence and reduce oscillations, ensuring stable training throughout the process.

Additionally, the network's performance was validated through a series of tests, and the results demonstrated that the network could accurately classify digits with minimal error. The results from the epoch-wise analysis of classification errors and loss function confirmed the stability of the model after a sufficient number of training iterations. Overall, the project showcased the effectiveness of back-propagation and neural networks in solving complex classification tasks.