Machine Learning Journal

Assessment Task 2 – Mitchell Kijurina – 14274334

Option 1: study a fundamental machine learning model

Table of Contents

1. Introduction

1.1 Purpose of project

The primary purpose of this project is to facilitate a deep, hands-on understanding of the fundamental concepts that drive the functioning of Multi-Layer Perceptrons (MLPs), a type of artificial neural network. By implementing an MLP from scratch, this study aims to bridge the gap between abstract theoretical principles and their implementation in code. The project serves as an exercise to explore key aspects of neural network learning, such the MLP architecture, forward and backwards propagation, activation functions and loss function design, within a practical context. This project aims to provide insights into how MLPs learn to make predictions or decisions based on data, thereby acting as a learning guide to both the implementation and theory of neural networks.

1.2 Objectives

The main objectives of this study are to:

- To explore the theoretical fundamentals of MLPs
- To implement an MLP model from scratch to demonstrate a clear understanding of its technical aspects.

1.3 Scope and Limitations

The study will focus solely on the architecture, theory, and implementation of MLPs, particularly within the context of the learning theory framework. The primary dataset for practical implementation will be the MNIST dataset for digit recognition. However, the study will not delve into variations of neural networks like Convolutional Neural Networks (CNNs) or Recurrent Neural Networks (RNNs). Further the project will not evaluate the impact of adjusting the hyper-parameters for optimisation of the network, as the focus of the study is solely on the implementation of the algorithm.

2. Background and Related Concepts

This section will cover a high-level overview of coming topics that are explored more in-depth when looking at the implementation of the program.

2.1 What is a Neural Network?

Neural networks, also known as artificial neural networks (ANNs) or simulated neural networks (SNNs), are crucial to machine learning and deep learning algorithms. Their structure is inspired by the human brain, mimicking the way biological neurons signal to each other.

The neuron is the fundamental unit of the neural network. Each neuron, or a node, has the following components.

Inputs

- Weights
- Bias
- Activation Function

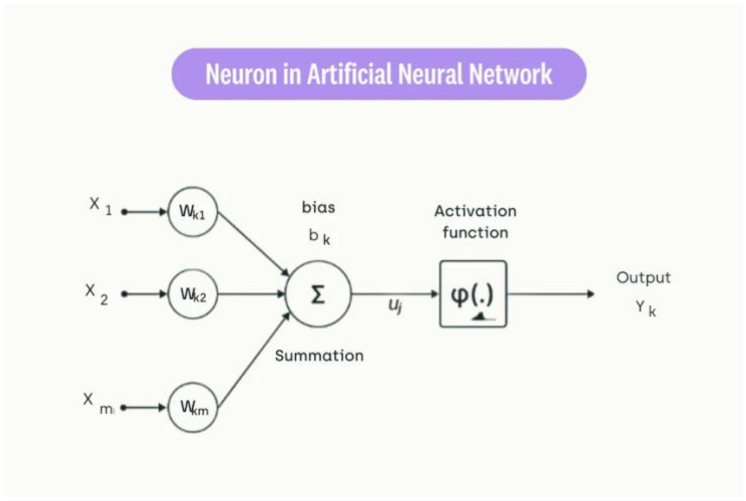


Figure 1 - Diagram of a neuron [2]

2.2 Introduction to Multi-Layer Perceptrons (MLPs)

Multi-Layer Perceptrons (MLPs) are a type of feedforward neural network consisting of at least three layers: an input layer, one or more hidden layers, and an output layer. They can recognize more complex patterns with the inclusion of more layers and nodes per layer.

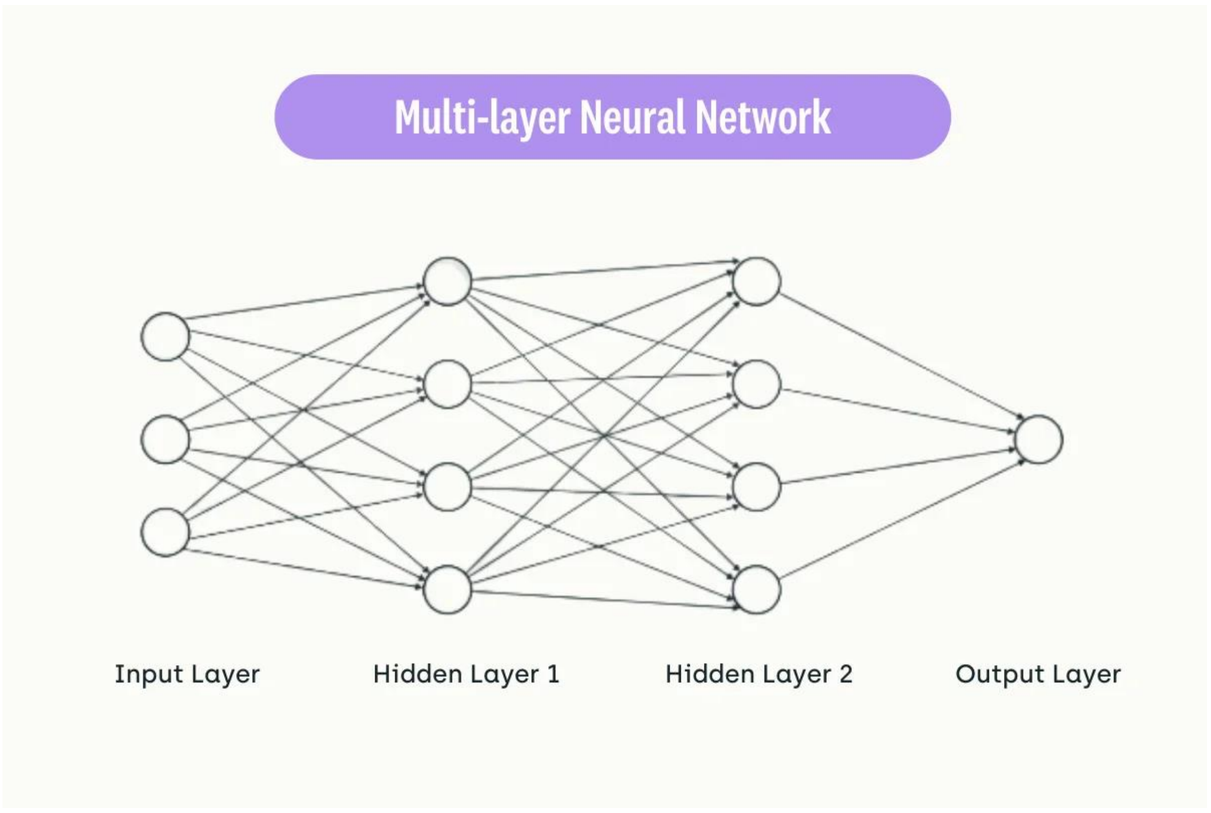


Figure 2 - Diagram of a MLP [2]

2.3 Overview of Activation Functions

Activation functions introduce non-linear properties to the network, enabling it to learn from the error and to handle complex problems. Common activation functions include Rectified Linear Unit (ReLU), Sigmoid, and Hyperbolic Tangent (tanh). These functions are crucial for neural network's learning process as they help in adjusting the weights during the training phase, aiding in the propagation of useful information through the network.

2.4 Understanding Loss Functions

Loss functions are pivotal as they measure the disparity between the predicted values and actual values, guiding the optimisation of the network's weights. The choice of a loss function, like Mean Squared Error (MSE) for regression tasks and Cross-Entropy Loss for classification tasks, can significantly affect the network's performance and convergence speed.

2.5 Basics of Training Algorithms

Training algorithms aim to minimise the loss function by iteratively adjusting the weights of the network. Backpropagation, combined with optimisation algorithms like Gradient Descent, is commonly used for this purpose. During training, the algorithm iteratively adjusts the weights to find the loss function's minimum, effectively "training" the network to improve its predictions over time.

3. Dataset

The Modified National Institute of Standards and Technology (MNIST) dataset is one of the most well-known and widely used datasets in the machine learning community, particularly for benchmarking classification algorithms. It consists of 60,000 grayscale images for training and an additional 10,000 images for testing. Each image in the dataset is 28x28 pixels and represents a handwritten digit from 0 to 9.

Characteristics of MNIST:

- **Grayscale Images**: The images are grayscale, simplifying the complexity as compared to coloured images. Each pixel has a single intensity value between 0 (black) and 255 (white).
- **Uniform Size**: All images are resized to a uniform 28x28 pixel size, thereby eliminating the need for additional pre-processing steps related to image resizing.
- **Digit Labels**: Each image comes with a corresponding label indicating the actual digit it represents, making it a supervised learning problem.
- **Balanced Classes**: The dataset is relatively balanced, meaning that there are almost an equal number of images for each of the 10-digit classes.

Relevance to the Project:

The MNIST dataset provides an ideal ground for implementing and understanding the Multi-Layer Perceptron model. Its simplicity and well-defined structure allow for a focus on the learning algorithm rather than data pre-processing. Moreover, the nature of the task—digit recognition—serves as a practical yet challenging problem, aligning well with the objectives of dissecting the intricacies of neural network learning both in theory and in practice.

Source of dataset:
The dataset was souced using the platform Kaggle; which is a machine learning and data science community driven site: https://www.kaggle.com/datasets/hojjatk/mnist-dataset

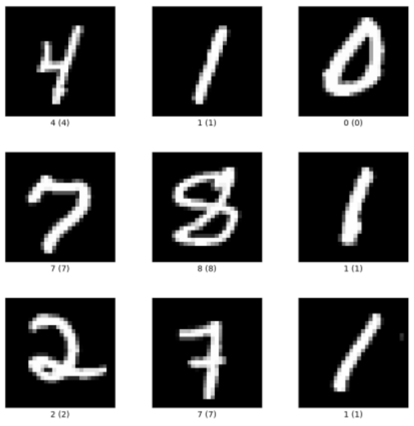


Figure 3- MNist Dataset [3]

4. Implementation and theoretical understanding

4.1 Forward Propagation

Forward propagation is an essential phase in the training cycle of a neural network. During this phase, input data passes sequentially through each layer of the network to produce an output. The two main operations performed in each layer are linear transformation and activation.

1. **Linear Transformation**: For any given layer, the input data or activations from the previous layer undergo a linear transformation. Mathematically, this is represented as:

$$Z = A_{\text{prev}}W + b$$

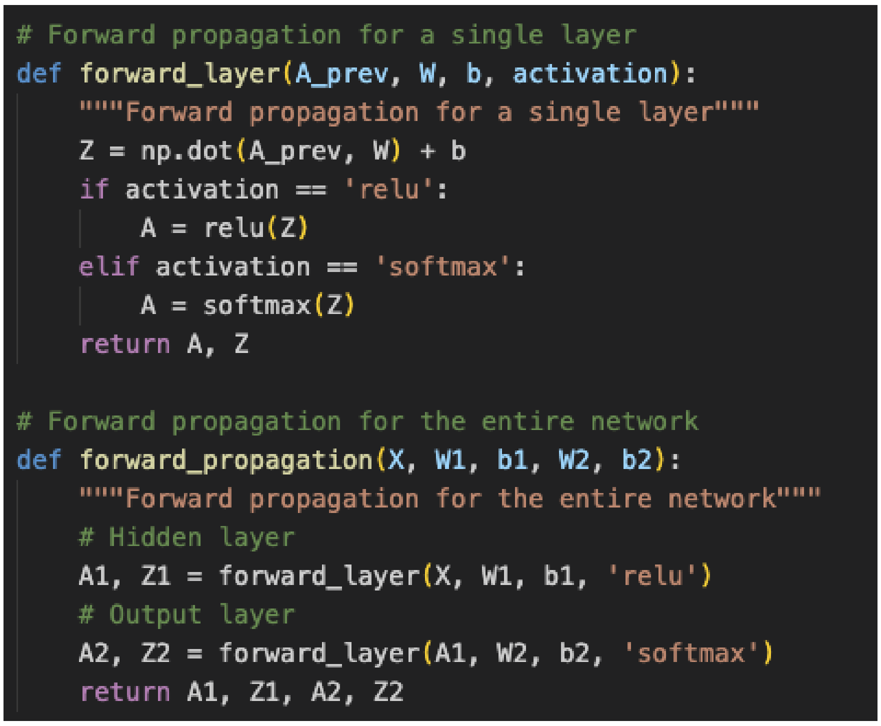
Where Z is the pre-activated output, A_prev are the activations from the previous layer, W is the weight matrix, and b is the bias vector for the current layer.

Activation: After the linear transformation, a non-linear activation function is applied to the pre-activated output.

The forward propagation process is encapsulated into two Python functions:

forward_layer: This function handles the forward propagation for an individual layer. It performs both the linear transformation and the activation.

forward_propagation: This function calls forward_layer for each layer in the network to perform forward propagation through the entire architecture.



```python
# Forward propagation for a single layer
def forward_layer(A_prev, W, b, activation):
    """Forward propagation for a single layer"""
    Z = np.dot(A_prev, W) + b
    if activation == 'relu':
        A = relu(Z)
    elif activation == 'softmax':
        A = softmax(Z)
    return A, Z

# Forward propagation for the entire network
def forward_propagation(X, W1, b1, W2, b2):
    """Forward propagation for the entire network"""
    # Hidden layer
    A1, Z1 = forward_layer(X, W1, b1, 'relu')
    # Output layer
    A2, Z2 = forward_layer(A1, W2, b2, 'softmax')
    return A1, Z1, A2, Z2
```

4.2 Backward Propagation

Backward propagation is the process through which gradients are calculated for each parameter in the network. This is crucial for the optimisation step, where these gradients are used to update the model parameters. Backward propagation starts from the output layer and proceeds in the opposite direction towards the input layer.

Gradient of the Loss: Initially, we calculate the derivative of the loss function L with respect to the output A, denoted as $\frac{\partial L}{\partial A}$

Backward Linear Transformation: For each layer, we compute the gradients with respect to the pre-activated output Z, weights W, and biases b. The formulae are as follows:

$$\frac{\partial L}{\partial Z} = \frac{\partial L}{\partial A} \odot f'(Z)$$

$$\frac{\partial L}{\partial W} = A_{\text{prev}}^{T} \frac{\partial L}{\partial Z}$$

$$\frac{\partial L}{\partial b} = \sum_{i=1}^{m} \frac{\partial L}{\partial Z^{(i)}}$$

The backward propagation is implemented through two main functions:

backward_layer: This function performs backward propagation for a single layer, calculating the gradients above.

backward_propagation: This function orchestrates the backward propagation through the entire network by sequentially calling backward_layer for each layer.

In these functions, dA, Z, A_prev, and W are the gradients of the loss with respect to the activations, the pre-activated output, the activations from the previous layer, and the weights, respectively. The activation argument specifies the type of activation function ('relu' or 'softmax') to be used.

4.3 Loss Function (Cross-Entropy)

In machine learning, the loss function measures the discrepancy between the predicted values and the actual ground-truth labels. One of the most commonly used loss functions for classification problems, like the one at hand, is the Categorical Cross-Entropy Loss. Mathematically, for a single data point, the categorical cross-entropy loss is defined as:

$$\text{Loss} = -\sum_{i=1}^{C} y_i \log(p_i)$$

Where C is the number of classes, y_i is the actual label (1 if the class is the true class, 0 otherwise), and p_i is the predicted probability of the data point belonging to class i.

For N data points, the loss function can be calculated as the average loss across all points:

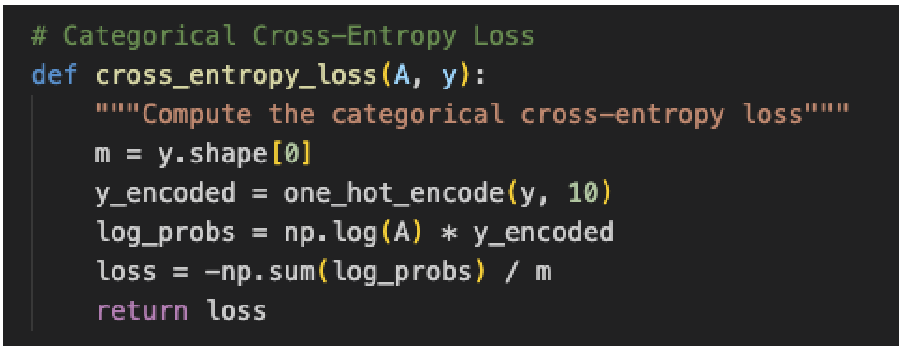
$$\text{Average Loss} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{C} y_{ni} \log(p_{ni})$$

In the program the categorical cross-entropy loss is computed using the Cross_entropy_loss function. The function takes in two arguments:

A: NumPy array containing the predicted probabilities for each class. This comes from the output layer of the neural network.
y: A NumPy array containing the actual labels.

- The function starts by calculating m, the number of samples in the mini batch.
- Next, it one-hot encodes the actual labels using the one_hot_encode function. Each row in y_encoded corresponds to a label in yy, represented as a one-hot encoded vector.
- The function then computes the logarithm of the predicted probabilities (log(A)) and multiplies elementwise with the one-hot encoded actual labels (y_encoded). The result is stored in log_probs.
- Finally, the function sums up all the values in log_probs and divides by m to get the average loss across the mini batch.

This loss value is then used in the training loop for backpropagation to update the model's parameters, thereby minimising the loss over time and improving the model's performance.



```python
# Categorical Cross-Entropy Loss
def cross_entropy_loss(A, y):
    """Compute the categorical cross-entropy loss"""
    m = y.shape[0]
    y_encoded = one_hot_encode(y, 10)
    log_probs = np.log(A) * y_encoded
    loss = -np.sum(log_probs) / m
    return loss
```

4.6 Weights and Biases

Theoretical Foundations

1. **Role in the Network**: Weights and biases are the learnable parameters in a neural network. They are the components that the model adjusts during training. Weights control the strength of the connection between neurons, while biases allow neurons to have some flexibility in activation.
2. **Initialisation**: Proper initialisation of weights and biases is crucial for effective training. Poor initialisation can lead to issues like vanishing or exploding gradients. Various strategies exist for initialisation, such as setting them to small random values.
3. **Dimensions**: The dimensions of the weight matrices and bias vectors depend on the architecture of the neural network. Specifically, the dimensions are influenced by the number of neurons in the connected layers.

Implementation in the Project

1. **Initialisation**: In the code, weights (W1 and W2) are initialised to small random values close to zero, multiplied by 0.01, ensuring they are in a range that helps avoid the aforementioned issues. Biases (b1 and b2) are initialised to zero arrays.
2. **Training Loop**: Within the training loop, the weights and biases are updated during each iteration of Mini-batch Gradient Descent. They are adjusted based on the computed gradients and the learning rate.
3. **Dimensionality**: In this specific project, the weights and biases have dimensions that are suited to the problem at hand. For example, W1 has dimensions (784, 128) indicating it connects an input layer of 784 neurons to a hidden layer of 128 neurons.
4. **Role in Forward and Backward Propagation**: During forward propagation, weights and biases are used to compute the activations of neurons. During backward propagation, gradients with respect to these weights and biases are calculated, which are then used to update these parameters.

Vanishing Gradients: Occurs when gradients become too small to update the weights effectively during backpropagation.

Exploding Gradients: Occurs when gradients become too large, causing unstable and erratic updates to the weights.

Mitigation in Code:

- **Small Initial Weights**: The code uses small random initial values for weights, thus reducing the risk of encountering these issues.
- ReLU **Activation**: The use of ReLU for the hidden layers helps mitigate the vanishing gradient problem.
- Learning **Rate**: A moderate learning rate of 0.1 is used to control the step size during updates, helping to avoid both vanishing and exploding gradients.
- **Mini-batch Gradient Descent**: The code employs mini-batch gradient descent, providing a balance between computational efficiency and gradient estimation accuracy.

5. Implementation Details

5.1 Input, Hidden and output Layers.

In a neural network, layers are structured units of nodes or neurons that transform the input data. Each layer performs specific operations dictated by activation functions and modifiable parameters (weights and biases). The network architecture in the code comprises:

Input Layer: This is the entry point of the network where each neuron corresponds to one feature of the dataset. In the code, the input layer implicitly has 784 neurons, matching the number of features in the MNIST dataset.

Hidden Layer: Hidden layers reside between the input and output layers, performing transformations on the input data. In this network, there is one hidden layer with 128 neurons, characterized by ReLU (Rectified Linear Unit) activation. The variables W1 and b1 hold the weights and biases for this layer, respectively.

Output Layer: This is the final layer of the network, and it typically transforms the values from the last hidden layer into output values that make sense for the given problem. In this case, the output layer has 10 neurons, each representing a class of digits (0-9). The Softmax function is applied to convert these values into probabilities. The weights and biases for this layer are stored in W2 and b2, respectively.

Each layer in the neural network serves to gradually transform the raw input into a form that makes it easier to produce the desired output. The transformations are governed by the layer's activation functions and its learnable parameters.

5.2 Activation Functions

Activation functions introduce non-linearity into a neural network, enabling it to learn complex mappings from the input data. They act as the "gating mechanisms" that either allow or restrict information to flow through the network. Different activation functions offer various properties that can affect the performance of a neural network.

Types of Activation Functions

1. **ReLU (Rectified Linear Unit)**

The ReLU function replaces all negative values in the input with zero. ReLU introduces non-linearity while solving some issues that other activation functions like sigmoid or tanh might face, such as the vanishing gradient problem. It's computationally efficient because it only requires a simple thresholding at zero.

2. **SoftMax**

The SoftMax function is often used in the output layer of a classifier to represent probability distributions of target classes. Given a vector of raw scores (or logits), SoftMax squashes the values between 0 and 1 and ensures they sum up to 1. Softmax is advantageous when you want to classify an input into one of multiple classes. It returns the probability distribution of the classes, which is convenient for not just determining the most likely class but also understanding the model's confidence in its prediction.

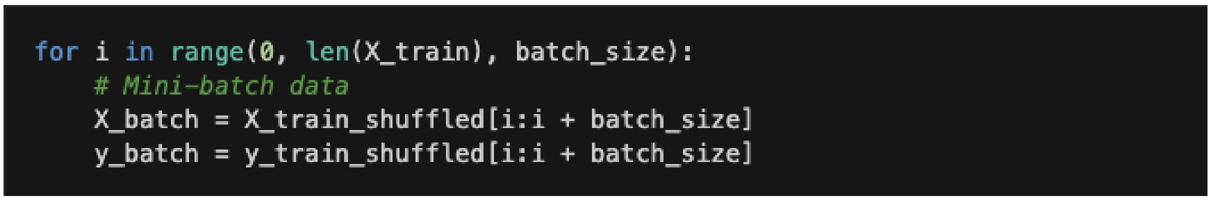
In the forward propagation process, ReLU is used in the hidden layer while Softmax is employed in the output layer.

5.3 Optimisation function

The optimisation algorithm used in the code is a basic form of Gradient Descent, specifically Mini-batch Gradient Descent. This is seen from the loop structure within the train_neural_network() function, where the model parameters (weights and biases) are updated incrementally for each mini-batch of data.

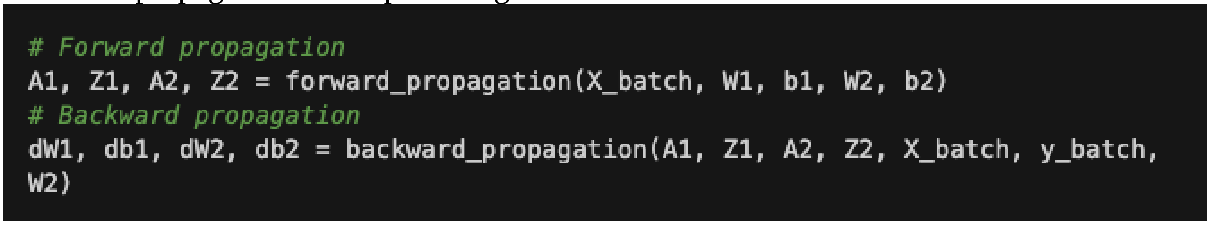
Mini-batch Gradient Descent strikes a balance between these two approaches. It divides the dataset into smaller batches and updates the model parameters for each batch.

In the train_neural_networks() function



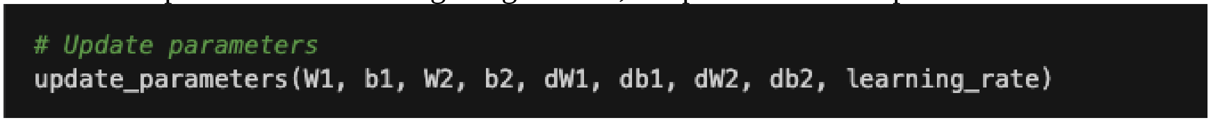
```
for i in range(0, len(X_train), batch_size):
    # Mini-batch data
    X_batch = X_train_shuffled[i:i + batch_size]
    y_batch = y_train_shuffled[i:i + batch_size]
```

Forward and Backward Propagation: For each batch, the network undergoes a forward and a backward propagation to compute the gradients.



```
# Forward propagation
A1, Z1, A2, Z2 = forward_propagation(X_batch, W1, b1, W2, b2)
# Backward propagation
dW1, db1, dW2, db2 = backward_propagation(A1, Z1, A2, Z2, X_batch, y_batch,
W2)
```

Parameter Update: After obtaining the gradients, the parameters are updated.



```
# Update parameters
update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, learning_rate)
```

The learning_rate controls the step size during the optimisation process. It's set prior to the training loop and is used in the update_parameters() function to adjust the weights and biases.

5.4 Training Loop

The training loop is a critical component in the neural network training pipeline. It orchestrates the forward and backward propagation steps, loss computation, and parameter updates, iterating over the entire dataset multiple times (epochs) to minimise the loss function.

Key Components

1. **Epochs**: An epoch is one complete forward and backward pass of all the training examples.
2. **Batch Size**: The number of training examples utilised in one iteration.
3. **Learning Rate**: A hyperparameter that controls how much to update the model weights.

Overview of the Training Loop in the Provided Code

The training loop in the given code is implemented in the `train_neural_network` function. It performs the following steps:

1. **Initialisation**: Sets up initial weights and biases.
2. **Epoch Loop**: Iterates through the dataset for a fixed number of epochs.
3. **Batch Loop**: Splits the training data into mini-batches and performs forward and backward propagation on each.
4. **Forward Propagation**: The network makes predictions based on current weights and biases.
5. **Loss Calculation**: Computes the Categorical Cross-Entropy loss.
6. **Backward Propagation**: Computes gradients of the loss with respect to the model parameters.
7. **Parameter Update**: Updates weights and biases based on the computed gradients and the learning rate.
8. **Evaluation**: Computes the accuracy of the model on the test set after each epoch.

Further explanation

Initialisation

The weights (W1, W2) and biases (b1, b2) for each layer are initialised. This is crucial for breaking the symmetry during training, allowing each neuron to learn different features.

Epoch Loop

The model iterates through the entire dataset multiple times (defined by epochs). Each epoch consists of several mini-batches.

Batch Loop

In each epoch, the training data is divided into mini-batches. This enables the model to update its weights more frequently, aiding in faster convergence.

Forward propagation
The mini-batch data is passed through the network layers, applying the ReLU and Softmax activation functions.

Loss Calculation

The Categorical Cross-Entropy loss is computed using the network's output and the true labels.

Backward Propagation

Gradients of the loss function with respect to each parameter are computed. This is used for updating the weights and biases.

Parameter Update

The weights and biases are updated in the direction that minimises the loss. The learning rate controls the size of these updates.

Evaluation

After each epoch, the model is evaluated on the test set to monitor its performance.

Training Loop Variables

1. **X_train, y_train**: These are the training data and labels, respectively. They are used to train the neural network.
2. **X_test, y_test**: These are the test data and labels, used for evaluating the model.
3. **epochs**: Number of complete passes through the entire training dataset.

4. **batch_size**: The size of each mini-batch for training.
5. **learning_rate**: The rate at which the model updates its parameters during training.
6. **train_losses**: List to store the computed loss for each epoch.
7. **test_accuracies**: List to store the test accuracy for each epoch.
8. **shuffle_indices**: Randomly shuffled indices used for randomising the training data each epoch.
9. **X_train_shuffled, y_train_shuffled**: The training data and labels, shuffled for each epoch.

Loss Computation Variables

1. **A2**: The predicted probabilities for each class, obtained from the output layer.
2. **y_batch**: The actual labels corresponding to the current mini-batch.
3. **loss**: Computed using the cross_entropy_loss function, which calculates the categorical cross-entropy loss between A2 and y_batch.

Forward Propagation Variables

1. **A1, Z1**: Activations (A1) and pre-activations (Z1) for the hidden layer.
2. **A2, Z2**: Activations (A2) and pre-activations (Z2) for the output layer.
3. **X_batch**: A mini-batch of the input data.

Backward Propagation Variables

1. **dW1, db1**: Gradients of the loss function with respect to the weights (dW1) and biases (db1) of the hidden layer.
2. **dW2, db2**: Gradients of the loss function with respect to the weights (dW2) and biases (db2) of the output layer.
3. **dA1, dA2**: Derivatives of the loss function with respect to the activated outputs (A1, A2) for the hidden and output layers.
4. **dZ1, dZ2**: Derivatives of the loss function with respect to the linear outputs (Z1, Z2) for the hidden and output layers.

Neural network vs MLP (https://chat.openai.com/share/a5a062ed-a4be-407d-aeb7-4b10e642b175)

1. **Neural Network (NN)**:
   - A Neural Network is a computational model inspired by the human brain's interconnected neuron structure.
   - It consists of layers of nodes or "neurons," where each node in a layer is connected to every node in the previous and subsequent layers.
   - The connections between nodes have associated weights, which are adjusted during training to minimize the error in predictions.

    - Neural Networks can have a wide variety of structures including differing numbers of layers, nodes per layer, and types of connections (e.g., convolutional, recurrent, etc.).
2. **Multilayer Perceptron (MLP)**:
    - A Multilayer Perceptron is a specific type of Neural Network.
    - It consists of at least three layers of nodes: an input layer, one or more "hidden" layers, and an output layer.
    - Every node in each layer is connected to every node in the adjacent layers, similar to the generic description of a Neural Network. This type of connectivity is often referred to as "fully connected" or "dense."
    - Unlike some other types of Neural Networks, MLPs do not have recurrent connections (connections that loop back) or convolutional connections (localized and shared weights).

Here's a comparative analysis to give a clearer picture:

- **Architecture**:
    - While both MLPs and general NNs comprise layers of nodes, NNs encompass a broader variety of architectures. For example, Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) are under the umbrella of NNs but are not considered MLPs due to their distinct connectivity patterns.
- **Complexity**:
    - MLPs are simpler in architecture compared to some other types of NNs which might have more complex connection patterns or additional mechanisms (e.g., memory cells in LSTM networks).
- **Use Cases**:
    - MLPs are often used for simpler, tabular data or when the data doesn't have a spatial or temporal component. On the other hand, other types of NNs like CNNs and RNNs are used for more complex data types like images, videos, and sequences.
- **Training**:
    - Both MLPs and NNs learn by adjusting their weights to minimize a loss function. However, the learning algorithms and the efficiency/effectiveness of learning can be quite different across different types of NNs.

6. Results and evaluations

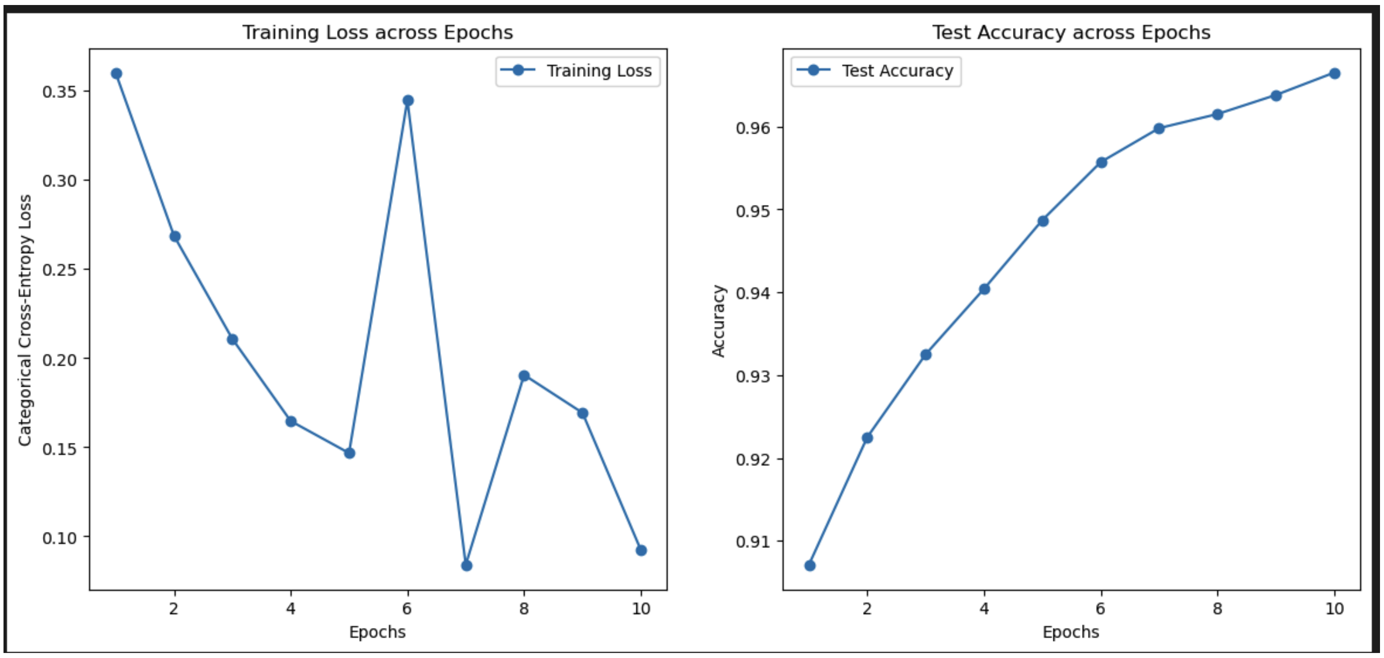
6.1) Results

Epoch-wise Loss and Test Accuracy

The most important metrics are the epoch-wise loss and test accuracy:

Loss: The loss starts at 0.3602 in the first epoch and subsequently decreases. While the exact final value is not provided, the decreasing trend implies that the model is effectively learning to approximate the function that maps the input features to the corresponding labels. A decreasing loss is usually a strong indicator that the Optimisation algorithm—likely some variant of gradient descent in this context—is successfully minimizing the objective function. This minimization is crucial for the model to make accurate predictions.

Test Accuracy: The test accuracy begins at 0.9071 and appears to improve over time. An increasing test accuracy is an indicator that the model is not only fitting to the training data but is also generalizing well to new, unseen data. The implication here is that the model has learned meaningful features from the training data that are representative of the broader dataset, rather than memorizing the training data.



Precision of ~0.967: Precision is the ratio of true positive predictions to the total number of positive predictions made (true positives + false positives). A high precision indicates that the false positive rate is low. In the context of the MNIST dataset, this could mean that when the model predicts a digit, it is highly likely to be correct.

Recall of ~0.967: Recall is the ratio of true positive predictions to the total actual positives (true positives + false negatives). A high recall indicates that the model identifies most of the positive samples correctly. In this case, it suggests that the model is good at identifying the correct digits across the different classes.

F1 Score of ~0.9665: The F1 score is the harmonic mean of precision and recall and takes both false positives and false negatives into account. An F1 score close to 1 is ideal and indicates a well-balanced model in terms of both precision and recall. In this case, the high F1 score corroborates the model's robustness in making accurate predictions.

6.2) Future work

Hyperparameter Optimisation

- The current implementation could benefit from a more extensive hyperparameter search, including learning rate, batch size, and regularization techniques, to further refine the model's performance.

Advanced Optimisation Algorithms

- Exploring different Optimisation algorithms like Adam, RMSprop, or Adagrad could potentially improve the speed and stability of the training process.

Model Architecture

- Experimenting with different architectures, including deeper networks, convolutional layers, or recurrent structures, may provide insights into the model's performance on more complex tasks.

Loss Function Analysis

- A detailed analysis of the loss function, especially the spike observed in the 6th epoch, could be conducted to understand its underlying cause and to develop strategies to mitigate such issues.

Data Augmentation

- Implementing data augmentation techniques could make the model more robust and improve its generalization capabilities.

Transfer Learning

- Future work could explore the applicability of transfer learning, utilising pre-trained models on similar tasks to improve performance and reduce training time.

Evaluation Metrics

- While the current metrics provide a good performance overview, adding more nuanced metrics like AUC-ROC, confusion matrix, or per-class accuracy could offer a more comprehensive evaluation.

Sources

1. https://www.ibm.com/topics/neural-networks#:~:text=Neural%20networks%2C%20also%20known%20as,neurons%20signal%20to%20one%20another
2. https://kili-technology.com/data-labeling/machine-learning/neural-network-architecture-all-you-need-to-know-as-an-mle-2023-edition#:~:text=4,feature%20instead%20of%20different%20ones
3. https://www.tensorflow.org/datasets/catalog/mnist
4. https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi (3b1b series)
5. https://www.youtube.com/watch?v=68BZ5f7P94E (relu statquest)
6. https://www.youtube.com/watch?v=6ArSys5qHAU (cross entropy) statquest

ChatGPT logs

[ChatGPT session for creation](https://chat.openai.com/share/0b8168ea-5d2a-497d-967e-c129e2424fcf)

[ChatGPT session for creation of variable table and description](https://chat.openai.com/share/c0d0bf8c-bbf3-49b1-bdf6-6f6c71055f8c)

https://www.baeldung.com/cs/gradient-stochastic-and-mini-batch

MLP vs NN
https://chat.openai.com/share/a5a062ed-a4be-407d-aeb7-4b10e642b175

Post code assistance chat for understanding 1
https://chat.openai.com/share/32d5a3e0-69d7-4573-84f7-b1564042f3bf