1. Why is it generally preferable to use a Logistic Regression classifier rather than a classical Perceptron (i.e., a single layer of linear threshold units trained using the Perceptron training algorithm)? How can you tweak a Perceptron to make it equivalent to a Logistic Regression classifier?

### Answer:- Preference for Logistic Regression over the Classical Perceptron

1. Probability Estimates:

* Logistic Regression: Provides probabilistic predictions, meaning it outputs probabilities that an input belongs to a particular class. This is achieved using the logistic function (sigmoid function) to map the linear combination of features to a probability value between 0 and 1. These probabilities are valuable for understanding the confidence of predictions and are useful in many practical applications (e.g., risk assessment).
* Perceptron: Outputs binary class labels directly (0 or 1) based on a linear decision boundary. It does not provide probabilistic estimates of class membership, which limits its applicability in scenarios where probability information is important.

2. Loss Function and Convergence:

* Logistic Regression: Uses the log loss (cross-entropy loss) which is well-suited for optimization with gradient descent methods. The log loss provides a smooth, continuous gradient that helps the model converge more effectively.
* Perceptron: Uses a hinge loss (or a variation) that can lead to suboptimal convergence properties compared to the log loss. It’s not always smooth or differentiable, which can make it harder to optimize using gradient-based methods.

3. Regularization:

* Logistic Regression: Supports regularization (L1, L2) to prevent overfitting and improve model generalization. Regularization is an integral part of logistic regression and can be easily implemented.
* Perceptron: Classical perceptrons do not inherently include regularization. While it is possible to add regularization, this is not part of the original perceptron algorithm.

4. Learning Stability and Convergence:

* Logistic Regression: Typically converges more reliably because it minimizes a convex loss function. The optimization problem in logistic regression is convex, which guarantees that gradient descent methods will find a global minimum.
* Perceptron: The perceptron algorithm may not always converge, especially in non-linearly separable cases. It converges only if the data is linearly separable and does not guarantee optimal convergence if the data is not linearly separable.

Tweaking a Perceptron to Make It Equivalent to Logistic Regression

To make a perceptron equivalent to logistic regression, you can modify the perceptron to incorporate the following aspects:

1. Use a Sigmoid Activation Function:

* Perceptron Adjustment: Replace the hard threshold activation function with the sigmoid function. This allows the perceptron to produce outputs in the range [0, 1], representing class probabilities.

σ(z)=11+e−z\sigma(z) = \frac{1}{1 + e^{-z}}σ(z)=1+e−z1​

where z=w⋅x+bz = \mathbf{w} \cdot \mathbf{x} + bz=w⋅x+b is the linear combination of inputs and weights plus bias.

2. Optimize with Cross-Entropy Loss:

* Perceptron Adjustment: Train the modified perceptron using the cross-entropy loss function (also known as log loss) instead of the traditional perceptron loss function.

L(y,y^)=−[ylog⁡(y^)+(1−y)log⁡(1−y^)]L(y, \hat{y}) = - [y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]L(y,y^​)=−[ylog(y^​)+(1−y)log(1−y^​)]

where y^=σ(z)\hat{y} = \sigma(z)y^​=σ(z) is the predicted probability, and yyy is the true label.

3. Implement Gradient Descent:

* Perceptron Adjustment: Use gradient descent or a variant to minimize the cross-entropy loss. This involves computing the gradient of the loss with respect to the weights and bias and updating them accordingly.

By incorporating these adjustments, the perceptron becomes equivalent to a logistic regression classifier. The key transformation is the use of the sigmoid function for producing probabilistic outputs and using cross-entropy loss for training, both of which align the perceptron’s behavior with that of logistic regression.

1. Why was the logistic activation function a key ingredient in training the first MLPs?

Answer:- The logistic activation function, also known as the sigmoid function, was a key ingredient in training the first Multi-Layer Perceptrons (MLPs) for several important reasons:

1. Non-Linearity

Purpose:

* The logistic activation function introduces non-linearity into the network. This non-linearity is crucial for allowing MLPs to model complex functions and learn from data that cannot be separated by a linear decision boundary.

Why:

* Without non-linearity, even multi-layer networks would behave like a single-layer model, as multiple layers of linear transformations are still linear. The sigmoid function enables the network to learn and represent complex, non-linear relationships between inputs and outputs.

2. Smooth Gradient

Purpose:

* The sigmoid function provides a smooth gradient, which is essential for gradient-based optimization methods used during training.

Why:

* The smoothness of the sigmoid function means that its derivative is well-defined and continuous, allowing for efficient and stable gradient descent. This gradient information is used to update the weights and biases during training, making it possible for the network to converge to a solution.

3. Probabilistic Interpretation

Purpose:

* The sigmoid function outputs values between 0 and 1, which can be interpreted as probabilities.

Why:

* This probabilistic interpretation is useful for binary classification problems. It allows the network to produce outputs that represent the probability of a given input belonging to a certain class, which can be directly used in loss functions like cross-entropy.

4. Gradient Descent and Backpropagation

Purpose:

* The sigmoid activation function’s derivative is straightforward to compute and is used in the backpropagation algorithm for updating weights.

Why:

* The derivative of the sigmoid function is given by σ′(z)=σ(z)(1−σ(z))\sigma'(z) = \sigma(z) (1 - \sigma(z))σ′(z)=σ(z)(1−σ(z)), where σ(z)\sigma(z)σ(z) is the sigmoid function. This property makes it easier to apply the chain rule for computing gradients and updating weights efficiently.

5. Historical Context

Purpose:

* The sigmoid function was a natural choice in the early days of neural network research due to its mathematical properties and historical precedence.

Why:

* In the early 1980s, when MLPs were first popularized, sigmoid functions were well-understood and widely used in various applications. Their effectiveness in training early neural networks made them a common choice for activation functions.

Summary

* Non-Linearity: Allows MLPs to model complex functions and decision boundaries.
* Smooth Gradient: Ensures stable and efficient gradient-based optimization.
* Probabilistic Interpretation: Provides useful probability outputs for classification tasks.
* Gradient Descent and Backpropagation: Facilitates the computation of gradients for weight updates.
* Historical Precedence: Was a natural and effective choice in the early days of neural network research.

Although other activation functions like ReLU have since become more popular due to their advantages in training deep networks, the sigmoid function was instrumental in the early success of MLPs and remains important for specific use cases, especially in binary classification problems.

1. Name three popular activation functions. Can you draw them?

Answer:- Here are three popular activation functions commonly used in neural networks, along with their descriptions and graphical representations:

1. Sigmoid Function (Logistic Activation Function)

Formula: σ(z)=11+e−z\sigma(z) = \frac{1}{1 + e^{-z}}σ(z)=1+e−z1​

Description:

* The sigmoid function maps any input zzz to a value between 0 and 1.
* It is often used for binary classification tasks because its output can be interpreted as a probability.

Graph:

2. Hyperbolic Tangent Function (tanh)

Formula: tanh⁡(z)=ez−e−zez+e−z\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}tanh(z)=ez+e−zez−e−z​

Description:

* The tanh function maps any input zzz to a value between -1 and 1.
* It is zero-centered, meaning the output is balanced around 0, which can make learning easier compared to the sigmoid function.

Graph:

3. Rectified Linear Unit (ReLU)

Formula: ReLU(z)=max⁡(0,z)\text{ReLU}(z) = \max(0, z)ReLU(z)=max(0,z)

Description:

* The ReLU function maps any input zzz to the maximum of 0 and zzz.
* It is widely used in deep learning due to its simplicity and effectiveness in helping the network learn faster and perform better.

Graph:

These activation functions each have their own characteristics and are chosen based on the specific requirements of the neural network architecture and the problem being solved.

1. Suppose you have an MLP composed of one input layer with 10 passthrough neurons, followed by one hidden layer with 50 artificial neurons, and finally one output layer with 3 artificial neurons. All artificial neurons use the ReLU activation function.
   * What is the shape of the input matrix X?

Answer:- Given the MLP architecture described:

* Input Layer: 10 passthrough neurons (features)
* Hidden Layer: 50 artificial neurons
* Output Layer: 3 artificial neurons

The shape of the input matrix XXX depends on the number of examples (or samples) you have in your dataset. However, the shape of the input matrix is determined by the number of features (neurons) in the input layer.

Shape of the Input Matrix XXX

* Features: The input layer has 10 neurons, which corresponds to 10 features.
* Examples: Suppose you have mmm examples in your dataset.

The input matrix XXX is typically organized such that each row represents one example, and each column represents one feature. Therefore:

* Number of Rows: Represents the number of examples (or samples), which is mmm.
* Number of Columns: Represents the number of features, which is 10.

So, the shape of the input matrix XXX is:

(m,10)(m, 10)(m,10)

where mmm is the number of examples in your dataset, and 10 corresponds to the number of features (input neurons).

* + What about the shape of the hidden layer’s weight vector Wh, and the shape of its bias vector bh?

Answer:- To determine the shapes of the weight and bias vectors for the hidden layer in your MLP, let's analyze the structure of the network:

Network Structure

1. Input Layer: 10 neurons
2. Hidden Layer: 50 neurons
3. Output Layer: 3 neurons

Weight and Bias Vectors for the Hidden Layer

Hidden Layer Weights WhW\_hWh​

* Dimensions: The weight matrix WhW\_hWh​ connects the input layer to the hidden layer.
* Shape Calculation: Each neuron in the hidden layer receives inputs from all neurons in the input layer. Therefore, the weight matrix WhW\_hWh​ has dimensions: Wh (shape)=(number of neurons in input layer,number of neurons in hidden layer)=(10,50)W\_h \text{ (shape)} = (\text{number of neurons in input layer}, \text{number of neurons in hidden layer}) = (10, 50)Wh​ (shape)=(number of neurons in input layer,number of neurons in hidden layer)=(10,50)

Hidden Layer Biases bhb\_hbh​

* Dimensions: Each neuron in the hidden layer has a bias term. Therefore, the bias vector bhb\_hbh​ has one bias term for each neuron in the hidden layer.
* Shape Calculation: Since there are 50 neurons in the hidden layer, the bias vector bhb\_hbh​ has dimensions: bh (shape)=(number of neurons in hidden layer)=(50,)b\_h \text{ (shape)} = (\text{number of neurons in hidden layer}) = (50,)bh​ (shape)=(number of neurons in hidden layer)=(50,)

Summary

* Weight Vector WhW\_hWh​: Shape = (10, 50)
* Bias Vector bhb\_hbh​: Shape = (50,)
  + What is the shape of the output layer’s weight vector Wo, and its bias vector bo?

Answer:- For the given Multi-Layer Perceptron (MLP) architecture, the shapes of the weight vector WoW\_oWo​ and bias vector bob\_obo​ for the output layer can be determined as follows:

Architecture:

1. Input Layer: 10 neurons
2. Hidden Layer: 50 neurons
3. Output Layer: 3 neurons

Activation Function: ReLU

Weight Vector WoW\_oWo​

* Purpose: Connects the hidden layer to the output layer.
* Shape Calculation:
  + Each neuron in the output layer is connected to all neurons in the hidden layer.
  + Therefore, WoW\_oWo​ must have a shape that represents these connections: each row corresponds to the weights connecting a neuron in the output layer to all neurons in the hidden layer.

The shape of WoW\_oWo​ is:

Shape of Wo=(number of neurons in output layer,number of neurons in hidden layer)\text{Shape of } W\_o = (\text{number of neurons in output layer}, \text{number of neurons in hidden layer})Shape of Wo​=(number of neurons in output layer,number of neurons in hidden layer)

* + Number of neurons in the output layer: 3
  + Number of neurons in the hidden layer: 50

So:

Shape of Wo=(3,50)\text{Shape of } W\_o = (3, 50)Shape of Wo​=(3,50)

Bias Vector bob\_obo​

* Purpose: Adds a bias term to each neuron in the output layer.
* Shape Calculation:
  + The bias vector bob\_obo​ has one bias term per neuron in the output layer.

The shape of bob\_obo​ is:

Shape of bo=(number of neurons in output layer)\text{Shape of } b\_o = (\text{number of neurons in output layer})Shape of bo​=(number of neurons in output layer)

* + Number of neurons in the output layer: 3

So:

Shape of bo=(3,)\text{Shape of } b\_o = (3,)Shape of bo​=(3,)

Summary:

* Weight Vector WoW\_oWo​: Shape is (3,50)(3, 50)(3,50)
* Bias Vector bob\_obo​: Shape is (3,)(3,)(3,)
  + What is the shape of the network’s output matrix Y?

Answer:- Given the structure of the MLP:

1. Input Layer: 10 neurons
2. Hidden Layer: 50 neurons
3. Output Layer: 3 neurons

The shape of the network’s output matrix YYY depends on the number of samples in your input batch.

Understanding Output Matrix Shape

* Neurons in Output Layer: The output layer has 3 neurons, which means the network will produce 3 output values for each input sample.
* Batch Size: If you process a batch of NNN input samples, each input sample will produce a vector of 3 output values.

Shape of Output Matrix YYY

* For a Single Sample: The output vector has a shape of (3,), where 3 is the number of output neurons.
* For a Batch of NNN Samples: The output matrix YYY will have a shape of (N, 3).

Summary:

* The shape of the network’s output matrix YYY is (N, 3), where NNN is the number of samples in the batch. Each row of YYY represents the output vector for one sample, and there are 3 values (one for each neuron in the output layer) per sample.
  + Write the equation that computes the network’s output matrix Y as a function of X, Wh, bh, Wo and bo.

Answer:- To compute the output matrix YYY of the Multi-Layer Perceptron (MLP) with one hidden layer using ReLU activation functions, follow these steps:

1. Input Layer:
   * Input matrix XXX with shape (N,10)(N, 10)(N,10), where NNN is the number of input samples.
2. Hidden Layer:
   * Weight matrix WhW\_hWh​ with shape (10,50)(10, 50)(10,50).
   * Bias vector bhb\_hbh​ with shape (1,50)(1, 50)(1,50).
   * ReLU activation function.
3. Output Layer:
   * Weight matrix WoW\_oWo​ with shape (50,3)(50, 3)(50,3).
   * Bias vector bob\_obo​ with shape (1,3)(1, 3)(1,3).

Here's how to compute the output matrix YYY as a function of XXX, WhW\_hWh​, bhb\_hbh​, WoW\_oWo​, and bob\_obo​:

1. Compute the Hidden Layer Output:

Zh=X⋅Wh+bhZ\_h = X \cdot W\_h + b\_hZh​=X⋅Wh​+bh​ Ah=ReLU(Zh)A\_h = \text{ReLU}(Z\_h)Ah​=ReLU(Zh​)

Where:

* + ZhZ\_hZh​ is the linear combination of inputs and weights plus biases for the hidden layer.
  + AhA\_hAh​ is the activated output of the hidden layer after applying the ReLU function.

1. Compute the Output Layer Output:

Zo=Ah⋅Wo+boZ\_o = A\_h \cdot W\_o + b\_oZo​=Ah​⋅Wo​+bo​ Y=ZoY = Z\_oY=Zo​

Where:

* + ZoZ\_oZo​ is the linear combination of hidden layer outputs and weights plus biases for the output layer.
  + YYY is the final output matrix of the network.

Summary of the Equations:

1. Hidden Layer Computation:

Zh=X⋅Wh+bhZ\_h = X \cdot W\_h + b\_hZh​=X⋅Wh​+bh​ Ah=ReLU(Zh)A\_h = \text{ReLU}(Z\_h)Ah​=ReLU(Zh​)

1. Output Layer Computation:

Zo=Ah⋅Wo+boZ\_o = A\_h \cdot W\_o + b\_oZo​=Ah​⋅Wo​+bo​ Y=ZoY = Z\_oY=Zo​

Explanation:

* XXX is the input matrix with dimensions (N,10)(N, 10)(N,10).
* WhW\_hWh​ is the weight matrix for the hidden layer with dimensions (10,50)(10, 50)(10,50).
* bhb\_hbh​ is the bias vector for the hidden layer with dimensions (1,50)(1, 50)(1,50).
* WoW\_oWo​ is the weight matrix for the output layer with dimensions (50,3)(50, 3)(50,3).
* bob\_obo​ is the bias vector for the output layer with dimensions (1,3)(1, 3)(1,3).

By following these steps, you compute the output matrix Y of the MLP for the given input X.

1. How many neurons do you need in the output layer if you want to classify email into spam or ham? What activation function should you use in the output layer? If instead you want to tackle MNIST, how many neurons do you need in the output layer, using what activation function?

### Answer:- Classifying Emails into Spam or Ham

* Number of Neurons:
  + You need 1 neuron in the output layer because this is a binary classification problem (spam or ham).
* Activation Function:
  + Use the sigmoid activation function. The sigmoid function will output a probability between 0 and 1, which can be interpreted as the likelihood of the email being spam.

Tackling the MNIST Dataset

* Number of Neurons:
  + You need 10 neurons in the output layer because MNIST is a multi-class classification problem with 10 classes (digits 0 through 9).
* Activation Function:
  + Use the softmax activation function. The softmax function will output a probability distribution over the 10 classes, with each neuron representing the probability of the input image belonging to one of the digits (0-9).

Summary:

* Spam/Ham Classification:
  + Output Neurons: 1
  + Activation Function: Sigmoid
* MNIST Classification:
  + Output Neurons: 10
  + Activation Function: Softmax

1. What is backpropagation and how does it work? What is the difference between backpropagation and reverse-mode autodiff?

### Answer:- What is Backpropagation?

**Backpropagation** (short for "backward propagation of errors") is a key algorithm used for training neural networks, particularly Multi-Layer Perceptrons (MLPs). It efficiently computes the gradients of the loss function with respect to the weights of the network, allowing these weights to be updated in a way that minimizes the loss function during training.

#### How Backpropagation Works:

1. **Forward Pass:**
   * Inputs are passed through the network, and activations are computed layer by layer until the output is obtained.
   * The loss (error) is then calculated based on the difference between the predicted output and the actual target values.
2. **Backward Pass (Error Propagation):**
   * The goal is to minimize the loss by adjusting the network's weights.
   * The algorithm computes the gradient of the loss with respect to each weight by applying the chain rule of calculus.
   * It works from the output layer back to the input layer, hence the name "backpropagation."

**Steps in the Backward Pass:**

* + **Compute Output Error:** Calculate the gradient of the loss function with respect to the output layer’s activation.
  + **Propagate Error Backwards:** For each layer, starting from the output and moving backward:
    - Compute the gradient of the loss with respect to the weights and biases.
    - Calculate the gradient of the loss with respect to the input of that layer (which becomes the error term for the previous layer).
  + **Update Weights:** Use the computed gradients to update the weights and biases (typically using gradient descent).

#### Summary of Backpropagation:

* **Forward Pass:** Compute activations and loss.
* **Backward Pass:** Compute gradients of the loss w.r.t. weights.
* **Update Weights:** Adjust weights using gradient descent.

### Difference Between Backpropagation and Reverse-Mode Autodiff:

**1. Reverse-Mode Autodiff (Automatic Differentiation):**

* **General Concept:** Reverse-mode autodiff is a technique for automatically computing the derivatives of a function. It's a more general concept that is not limited to neural networks.
* **How It Works:** It computes derivatives by evaluating the function in two phases:
  + **Forward Pass:** Compute the function's value and store intermediate variables.
  + **Backward Pass:** Propagate the derivatives of the final output with respect to each intermediate variable.
* **Efficiency:** Reverse-mode autodiff is efficient for functions with many inputs and a single output (like in neural networks) because it computes gradients with respect to all inputs in a single backward pass.

**2. Backpropagation:**

* **Special Case:** Backpropagation can be seen as a specific application of reverse-mode autodiff tailored to neural networks.
* **Scope:** It’s specifically designed to compute gradients for neural network training.
* **Mechanism:** It applies reverse-mode autodiff to the network’s computational graph, using the chain rule to compute the gradient of the loss function with respect to each parameter in the network.

### Key Differences:

* **General vs. Specific:** Reverse-mode autodiff is a general method for computing derivatives, while backpropagation is specifically designed for training neural networks.
* **Application:** Backpropagation uses reverse-mode autodiff within the context of neural networks.
* **Focus:** Reverse-mode autodiff focuses on efficiently computing gradients for functions with many inputs and one output, while backpropagation focuses on the practical application of this method in updating neural network weights.

In essence, backpropagation is a specialized implementation of reverse-mode autodiff for the purpose of training neural networks by optimizing their parameters.

1. Can you list all the hyperparameters you can tweak in an MLP? If the MLP overfits the training data, how could you tweak these hyperparameters to try to solve the problem?

### Answer:- Hyperparameters in an MLP (Multi-Layer Perceptron)

1. Learning Rate:
   * Controls the size of the steps the optimizer takes during gradient descent.
2. Number of Layers:
   * The total number of hidden layers in the MLP.
3. Number of Neurons per Layer:
   * The number of neurons in each hidden layer.
4. Activation Functions:
   * The type of activation function used in each layer (e.g., ReLU, sigmoid, tanh).
5. Weight Initialization:
   * The method used to initialize the weights (e.g., Xavier, He initialization).
6. Batch Size:
   * The number of training samples used in one iteration before the weights are updated.
7. Number of Epochs:
   * The number of times the entire training dataset is passed through the network.
8. Optimizer:
   * The optimization algorithm used for training (e.g., SGD, Adam, RMSprop).
9. Momentum:
   * A hyperparameter used in optimizers like SGD with momentum, controlling the influence of past gradients on the current update.
10. Learning Rate Schedule:
    * The method for adjusting the learning rate during training (e.g., step decay, exponential decay).
11. Regularization Techniques:
    * L1/L2 Regularization: Adds a penalty to the loss function based on the magnitude of weights.
    * Dropout Rate: The fraction of neurons to drop during training to prevent overfitting.
12. Early Stopping:
    * A technique that stops training when the validation loss stops improving.
13. Gradient Clipping:
    * Limits the magnitude of gradients to prevent exploding gradients.
14. Batch Normalization:
    * A technique to normalize the input to each layer, speeding up training and improving stability.
15. Learning Rate Warmup:
    * Gradually increasing the learning rate during the initial phase of training.

Addressing Overfitting in an MLP

If an MLP overfits the training data, meaning it performs well on the training set but poorly on the validation set, you can tweak the following hyperparameters to try to solve the problem:

1. Increase Regularization:
   * L2 Regularization (Weight Decay): Increase the L2 regularization term to penalize large weights, encouraging the network to use smaller weights that may generalize better.
   * L1 Regularization: This can also be used to encourage sparsity in the model by driving some weights to zero.
2. Increase Dropout Rate:
   * Increase the dropout rate to reduce overfitting by randomly dropping more neurons during training, which forces the network to be less reliant on specific neurons.
3. Reduce Network Complexity:
   * Number of Layers: Reduce the number of hidden layers to make the model simpler.
   * Number of Neurons per Layer: Reduce the number of neurons in each hidden layer.
4. Early Stopping:
   * Implement early stopping to halt training when the validation loss stops improving, preventing the model from overfitting to the training data.
5. Use Batch Normalization:
   * Apply batch normalization to stabilize and speed up training, which can help the model generalize better.
6. Data Augmentation:
   * If applicable, use data augmentation techniques to artificially increase the size of the training dataset, improving the model’s ability to generalize.
7. Reduce Learning Rate:
   * A smaller learning rate might prevent the model from converging too quickly to a sharp local minimum that doesn’t generalize well.
8. Increase Batch Size:
   * Increasing the batch size can lead to more stable gradient estimates, potentially reducing overfitting.

Summary

* Increase Regularization and Dropout to penalize complexity.
* Simplify the Model by reducing layers or neurons.
* Use Early Stopping to prevent overfitting during training.
* Apply Batch Normalization to stabilize training.
* Adjust Learning Rate and Batch Size for better generalization.
* Use Data Augmentation to improve the model's robustness to unseen data.

By carefully tweaking these hyperparameters, you can often mitigate the overfitting problem and improve the generalization performance of your MLP.

1. Train a deep MLP on the MNIST dataset and see if you can get over 98% precision. Try adding all the bells and whistles (i.e., save checkpoints, restore the last checkpoint in case of an interruption, add summaries, plot learning curves using TensorBoard, and so on).

Answer:- Training a deep MLP on the MNIST dataset to achieve over 98% precision is a great exercise. Below is a step-by-step guide on how to set up, train, and monitor the model using TensorFlow and TensorBoard.

### Step 1: Import Required Libraries

import tensorflow as tf

from tensorflow.keras.datasets import mnist

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Dropout, BatchNormalization

from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping, TensorBoard

import matplotlib.pyplot as plt

import os

Step 2: Load and Preprocess the MNIST Dataset

# Load dataset

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

# Preprocess data

x\_train = x\_train.reshape(-1, 28 \* 28).astype('float32') / 255.0

x\_test = x\_test.reshape(-1, 28 \* 28).astype('float32') / 255.0

# Convert labels to one-hot encoding

y\_train = tf.keras.utils.to\_categorical(y\_train, 10)

y\_test = tf.keras.utils.to\_categorical(y\_test, 10)

Step 3: Build the MLP Model

model = Sequential([

Dense(512, activation='relu', input\_shape=(28 \* 28,)),

BatchNormalization(),

Dropout(0.2),

Dense(512, activation='relu'),

BatchNormalization(),

Dropout(0.2),

Dense(512, activation='relu'),

BatchNormalization(),

Dropout(0.2),

Dense(10, activation='softmax')

])

model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

Step 4: Set Up Checkpoints and TensorBoard

checkpoint\_path = "checkpoints/mnist\_model.ckpt"

checkpoint\_dir = os.path.dirname(checkpoint\_path)

# Create checkpoint callback

checkpoint\_callback = ModelCheckpoint(

filepath=checkpoint\_path,

save\_weights\_only=True,

monitor='val\_accuracy',

mode='max',

save\_best\_only=True,

verbose=1

)

# Early stopping callback

early\_stopping\_callback = EarlyStopping(

monitor='val\_loss',

patience=5,

verbose=1,

restore\_best\_weights=True

)

# TensorBoard callback

tensorboard\_callback = TensorBoard(log\_dir="logs")

# Make sure checkpoint directory exists

os.makedirs(checkpoint\_dir, exist\_ok=True)

Step 5: Train the Model

history = model.fit(

x\_train, y\_train,

validation\_data=(x\_test, y\_test),

epochs=50,

batch\_size=128,

callbacks=[checkpoint\_callback, early\_stopping\_callback, tensorboard\_callback],

verbose=2

)

Step 6: Evaluate the Model

# Restore the best weights

model.load\_weights(checkpoint\_path)

# Evaluate the model

test\_loss, test\_accuracy = model.evaluate(x\_test, y\_test, verbose=2)

print(f"Test accuracy: {test\_accuracy \* 100:.2f}%")

### Step 7: Visualize Training with TensorBoard

To visualize the training process, run the following command in your terminal to launch TensorBoard:

tensorboard --logdir=logs

Step 8: Plot Learning Curves

plt.plot(history.history['accuracy'], label='Train accuracy')

plt.plot(history.history['val\_accuracy'], label='Validation accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.title('Training and Validation Accuracy')

plt.legend()

plt.show()

plt.plot(history.history['loss'], label='Train loss')

plt.plot(history.history['val\_loss'], label='Validation loss')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.title('Training and Validation Loss')

plt.legend()

plt.show()

Step 9: Fine-Tuning (if necessary)

If you don't reach the desired accuracy, consider the following:

* Increase the number of neurons or layers in the MLP.
* Adjust the learning rate.
* Try different optimizers like RMSprop or SGD with momentum.
* Increase regularization or dropout if overfitting occurs.

Final Notes

* Achieving 98% accuracy is feasible with the right architecture and training regimen.
* Monitoring with TensorBoard gives you insights into how well your model is training.
* Checkpoints and early stopping help prevent overfitting and ensure you keep the best-performing model.

This setup provides a solid foundation to train a deep MLP on MNIST and achieve high precision.