Multilayer Perceptron (MLPs) and Backpropagation

Submitted by: Sushma Sree Laskar

Student Id:23032632

Git: https://github.com/Sushma897sree/ML-Tutorial_2303263

Introduction:

Neural Networks and Perceptrons

Introduction to Artificial Neural Networks (ANNs)

Artificial Neural Networks (ANNs) are computational models inspired by the biological neural networks in the human brain.

Each artificial neuron:

1. Receives inputs
2. Computes a weighted sum of these inputs
3. Applies an activation function to produce an output,

Simplest form of an ANN is the perceptron, capable of solving linearly separable problems like AND. However, perceptrons struggle with non-linear problems, such as XOR, highlighting the need for more advanced architectures like Multilayer Perceptrons (MLPs).

Example: XOR Problem

The XOR (exclusive OR) problem is a classic example that demonstrates the limitations of perceptrons. In XOR:

Inputs: X= {(0,0), (0,1), (1,0), (1,1)}

Outputs: y= {0,1,1,0}

A perceptron cannot solve XOR because data points cannot be separated by a single straight line (non-linear separability).

This limitation necessitates multilayer architectures, where hidden layers transform input data into a space where a linear decision boundary can be drawn.

```
# XOR dataset
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 1, 1, 0])  # XOR labels
```

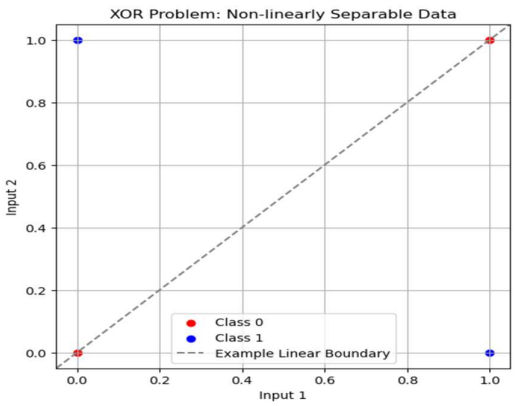

Fig 1.1 XOR Problem

This visualisation shows class separation:

Class 0 points(red) are at (0,0) and (1,1)

Class 1 points(blue) are at (0,1) and (1,0)

Linear Model Failure: Any single straight line (e.g., the dashed grey line in the plot) fails to separate the classes correctly. This is because XOR is not linearly separable. This highlights the limitation of a perceptron, which can only draw straight lines to separate data.

This limitation motivates the use of Multilayer Perceptrons, which introduce hidden layers and non-linear activation functions to solve such problems.

What is an MLP?

A Multilayer Perceptron (MLP) is a type of Artificial Neural Network (ANN) used in supervised learning tasks such as classification and regression. Unlike a simple perceptron (which has no hidden layers), MLPs consist of one or more hidden layers sandwiched between the input and output layers. These

hidden layers enable MLPs to learn and model non-linear relationships, making them highly versatile for complex datasets.

Key Characteristics of MLPs

1.Feedforward Network: MLPs are feedforward neural networks, means the flow of information moves in one direction i.e. from the input layer to the output layer, without looping back.

2.Supervised Learning: MLPs require labelled data during training to learn the mapping between inputs and outputs.

3.Universal Function Approximator: With sufficient hidden layers and neurons, an MLP can approximate any continuous function.

MLP Architecture: It is a type of feedforward artificial neural network (ANN) called as "Multilayer" since it contains several layers of neurons, stacked together. The key components of an MLP are **input layer, hidden layers, output layer.**

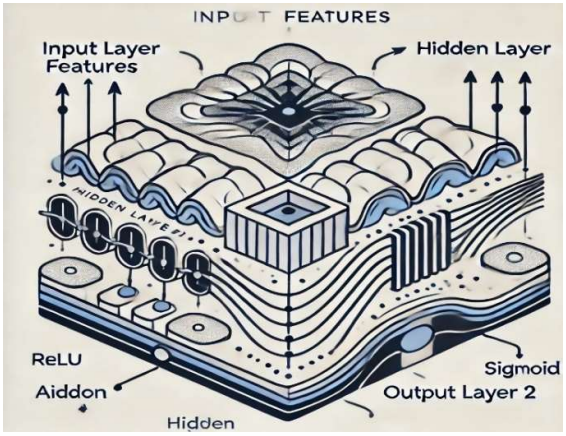

Fig 2.1 Architecture of a Multilayer Perceptron (MLP)

Input Layer

Role: This layer receives the raw data. It doesn't process the data instead it passes it to the next layer. Structure: Each neuron in the input layer corresponds to a feature in the dataset. The raw data is passed to the first hidden layer for further processing.

Example: For an image classification task (e.g., Fashion-MNIST dataset), the input layer would have one neuron per pixel in the image, meaning for a 28x28 image, there would be 784 neurons.

Hidden Layers

Role: These layers where model learns from the data. They perform most of the computation in the network and their job is to transform the input data into a format that model can use to make the predictions. The more hidden layers you have, the more complex patterns the model can learn.

Structure: Each neuron in the hidden layers receives weighted inputs from the previous layer applies a bias and passes the result through an activation function.

Mathematical Representation: For a neuron in the $l-th$ hidden layer:

$$z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)}$$

$$a^{(l)} = f(z^{(l)})$$

where: $z^{(l)}$ is the weighted sum of inputs, $W^{(l)}$ is the weight matrix, $a^{(l-1)}$is the activation of the previous layer, $b^{(l)}$ is the bias for layer l , $f(z^{(l)}$ is the activation function.

Why Hidden Layers Matter:

They allows network to capture non-linear relationships between inputs and outputs.

Each hidden layer processes information in a way that creates a more abstract representation of the input.

Output Layer

Role: Here in this layer network's predictions are made. This layer determines the outcome based on the learned weights.

Structure: For classification problems, the number of neurons in the output layer is equal to the number of classes.

For binary classification, you may use a single neuron with a Sigmoid activation function. In multi-class classification, you'll use one neuron per class, and typically for each class a

SoftMax activation function is used to output probabilities.

For **regression** problems, there is usually just **one neuron** that predicts a continuous value.

Mathematical Representation: For a neuron in the output layer:

$$z^{(L)} = W^{(L)}a^{(L-1)} + b^{(L)}$$

$$\hat{y} = f(z^{(L)})$$

$z^{(L)}$ is the weighted sum at the output layer, $W^{(L)}$ is the weight matrix, $a^{(L-1)}$is the activation of the previous layer, $\hat{y}$ is the final output.

For binary classification, the output might use the Sigmoid activation produces a value between 0 and 1 which represents the probability of a certain class. For multi-class classification, SoftMax is typically used to ensure the output sums to 1, representing the class probabilities.

Activation Functions in MLP

Importance of Activation Functions

Activation functions are essential because they adds **non-linearity** to the network, allows to model complex relationships. Without them, network would only be able to learn linear mappings, which is insufficient for most real-world tasks.

Sigmoid: Often used for binary classification, the sigmoid function squashes the output between 0 and 1, which makes useful for probability outputs.

- **Formula**: $f(z) = \frac{1}{1+e^{-z}}$

Range: (0, 1)

Problem: The sigmoid suffers from the vanishing gradient problem, making it harder to train deep networks.

Visual Representation:

Input: -5, -3, 0, 2, 5 → Output: 0.0067, 0.047, 0.5, 0.88, 0.993

Sigmoid squashes the values to lie between 0 and 1, making it suitable for probabilities.

Tanh: Purpose: Like sigmoid but with output values between -1 and 1, maybe suitable for tasks where the data has a centered distribution. Effect: Like sigmoid, but more powerful as it ensures the output is centered around 0.

Formula: $f(z) = \frac{1-e^{-z}}{1+e^{-z}}$ Used for data that is centered around 0.

Range: (-1, 1)

ReLU (Rectified Linear Unit): One of the most used activation functions due to its simplicity and efficiency.

Formula: f(z)=max (0, z),

Range: (0, ∞).

Purpose: Used in hidden layers of MLPs, ReLU is efficient and helps in mitigating the vanishing gradient problem. It outputs 0 for negative values and the input value for positive values, allows it computationally efficient.

Effect: ReLU adds non-linearity, allowing MLPs capturing the complex patterns in the data.

Visual Representation:

Input: -5, -3, 0, 2, 5 → Output: 0, 0, 0, 2, 5

ReLU activation creates a threshold at 0, ensuring positive values pass through as they are, while negative values are clipped.

SoftMax

Formula: $f(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$ **Purpose**: Used in the output layer for multi-class classification tasks. SoftMax helps in converts logits into probabilities, here the sum of the outputs equals 1.

Effect: Useful for classification problems where each class has a probability, and you need the model to select the most likely class.

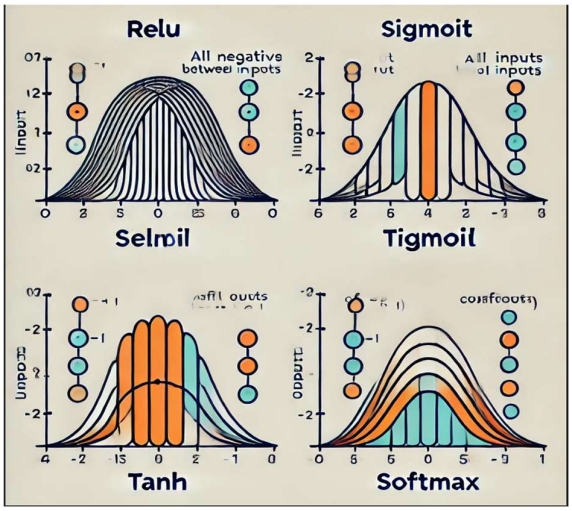

Fig 2.1 Activation Functions

Here is a graph comparing the four common activation functions used in **MLPs**:

ReLU: Output has zero for all negative inputs and the input itself for positive values.

Sigmoid: Squashes has values between 0 and 1, often used for binary classification.

Tanh: Squashes had values between -1 and 1, commonly used for hidden layers.

SoftMax: Converts raw output values into probabilities, summing to 1.

Let's have a look how these are implemented in Multilayer Perceptron (MLP) for the image classification using the Fashion-MNIST dataset.

Implementation

Fashion-MNIST is a dataset contains 60,000 training images and 10,000 test images of fashion items, each 28x28 pixels. It includes 10 categories such as T-shirts, trousers, and sneakers, commonly used for benchmarking image classification models.

Why Use MLP for Fashion-MNIST?

Multilayer Perceptrons (MLPs) are ideal for **Fashion-MNIST** because they could learn **non-linear relationships** in the data. MLPs automatically extract **features from raw pixel data** and are particularly effective in **high-dimensional tasks** like image recognition. By using **hidden layers**, **non-linear activation functions**. MLPs classifies complex data patterns, making them suitable for tasks such as **Fashion-MNIST**.

Backpropagation Concept:

Backpropagation is a key part of training artificial neural networks, including MLPs (Multilayer Perceptrons). It processes adjusting the weights of the neural network by minimizing the loss function through the gradient of the error with respect to every weight. The aim of backpropagation is to minimize the error between the predicted and actual outputs by adjusting the model's parameters (weights) step-by-step.

Steps in Backpropagation:

Forward Propagation:

Forward propagation happens during the **training phase** when the input data (Fashion-MNIST images) is passed through the model to produce predictions. In the provided code:

Each layer performing a weighted sum of its inputs and applies an activation function to the result. This process continues til the output layer is reached.

```
from tensorflow.keras import layers, models

# Building the MLP model
model = models.Sequential([
    layers.Flatten(input_shape=(28, 28)),  # Flatten the images into vectors
    layers.Dense(128, activation='relu'),  # Hidden layer with ReLU activation
    layers.Dropout(0.2),  # Dropout for regularization to prevent overfitting
    layers.Dense(10, activation='softmax')  # Output layer for classification (Softmax)
])

# Display the model architecture
model.summary()
```

Fig 3.1 Building MLP Model

Model Architecture: Input Layer: The images are flattened from 28x28 pixels into a 1D vector (784 features) using the Flatten layer. Hidden Layer: The dense layer with 128 neurons uses ReLU activation function in order to introduce non-linearity. Dropout: A dropout layer with 20% rate helps prevents overfitting. Output Layer: The final output layer will have 10 neurons (one per class in Fashion-MNIST) and uses the SoftMax activation function to produce class probabilities. In forward propagation, the

image data flowing from the input layer to the hidden layer, and then flows to the output layer where final probabilities are calculated.

Loss Function:

The error is calculated by comparing the predicted output (from the forward pass) and the actual target value (from the training data) It is sparse_categorical_crossentropy is utilised for multi-class classification, where the model is predicting the probabilities of 10 different classes (Fashion-MNIST categories).

This loss function calculating the difference between the predicted probability distribution and the actual label.

Weight Update (Gradient Descent):

The optimizer (such as Adam, SGD, etc.) uses the calculated gradients to update the weights in the direction which minimizes the error. This is done iteratively across all layers of the network.

Optimizer (Adam):

The Adam optimizer is specified here, which combines the benefits of Momentum and RMSprop. Adam helps to optimize the learning rate dynamically, making it suitable for complex and large datasets.

During training, the Adam optimizer adjusts the weights to minimize the loss by using the gradients computed via backpropagation.

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Fig 3.3 Adam's Optimizer usage

Gradient Calculation and Weight Updates

Backpropagation is automatically handled by Keras during the fit () method, where it calculates the gradients for each weight and updates them accordingly. The gradient descent algorithm (or Adam, in this case) updates the weights iteratively after each batch of data.

Training Process:

The model performs **forward propagation** and back propagation when calling model.fit():

The model computed the weighted sum of inputs for each layer, applied the activation function, and has outputs a prediction for each image. Back propagation process works for each batch of training data, adjusting the weights by calculating the loss, computing gradients, and then applying them using the optimizer (Adam).

This process is repeated for each epoch, continuously improving the model by reducing the loss.

```
history = model.fit(X_train, y_train, epochs=5, validation_data=(X_test, y_test))

Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - accuracy: 0.7576 - loss: 0.6866 - val_accuracy: 0.8443 - val_loss: 0.4311
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.8526 - loss: 0.4108 - val_accuracy: 0.8481 - val_loss: 0.4149
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.8632 - loss: 0.3718 - val_accuracy: 0.8634 - val_loss: 0.3830
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.8741 - loss: 0.3479 - val_accuracy: 0.8711 - val_loss: 0.3550
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.8764 - loss: 0.3349 - val_accuracy: 0.8700 - val_loss: 0.3528
```

Fig 3.2 Training the model with advanced techniques.

Let us discuss some more advance techniques in MLPs

Regularization Techniques:

Dropout: In this technique we randomly "dropping out" neurons during training to prevent overfitting. It helps network generalize better by preventing it from becoming too reliant on specific neurons or features.

```
from tensorflow.keras import layers, models

# Building the MLP model
model = models.Sequential([
    layers.Flatten(input_shape=(28, 28)),  # Flatten the images into vectors
    layers.Dense(128, activation='relu'),  # Hidden layer with ReLU activation
    layers.Dropout(0.2),  # Dropout for regularization to prevent overfitting
    layers.Dense(10, activation='softmax')  # Output layer for classification (Softmax)
])

# Display the model architecture
model.summary()
```

```
C:\Users\sushm\anaconda3\Lib\site-packages\keras\src\layers\reshaping\flatten.py:37: UserWa
layer. When using Sequential models, prefer using an `Input(shape)` object as the first la
  super().__init__(**kwargs)
Model: "sequential"
```

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 128)	100,480
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 10)	1,290

```
Total params: 101,770 (397.54 KB)
Trainable params: 101,770 (397.54 KB)
Non-trainable params: 0 (0.00 B)
```

Fig 3.4 Dropout Technique to prevent overfitting

We have a dropout layer in the code: layers. Dropout (0.2). It prevents overfitting by randomly setting 20% of the neurons to zero in between training. It helps the model generalize to better by reducing reliance on the specific neurons.

Early Stopping: Monitors the model's performance on validation set and stops training when the performance has begun to degrade, thus prevents overfitting.

```
from tensorflow.keras.callbacks import EarlyStopping

early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)

history = model.fit(X_train, y_train, epochs=50, validation_data=(X_test, y_test), callbacks=[early_stopping])
```

```
Epoch 1/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.8934 - loss: 0.2874 - val_accuracy: 0.8724 - val_loss: 0.3542
Epoch 2/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.8941 - loss: 0.2823 - val_accuracy: 0.8769 - val_loss: 0.3488
Epoch 3/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.8968 - loss: 0.2759 - val_accuracy: 0.8818 - val_loss: 0.3307
Epoch 4/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.8989 - loss: 0.2705 - val_accuracy: 0.8787 - val_loss: 0.3513
Epoch 5/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.8993 - loss: 0.2690 - val_accuracy: 0.8782 - val_loss: 0.3426
Epoch 6/50
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.9004 - loss: 0.2623 - val_accuracy: 0.8871 - val_loss: 0.3310
```

Fig 3.5 Early stopping Technique

This has stopped training when the validation loss did not improve for 3 consecutive epochs since patience=3 was given.

L2 Regularization: It adds a penalty term to the loss function to decrease the magnitude of the weights, helping in preventing overfitting by discouraging large weights.

This can be added to the Dense layers to penalize large weights:

kernel_regularizer=regularizes. l2(0.005) adds L2 regularization with a weight decay of 0.005

```
from tensorflow.keras import layers, models
from tensorflow.keras import regularizers

# Building the MLP model
model = models.Sequential([
    layers.Flatten(input_shape=(28, 28,1)),  # Flatten the images into vectors
    layers.Dense(256, activation='relu'),
    layers.Dense(128, activation='relu',kernel_regularizer=regularizers.l2(0.005)),  # Hidden layer with ReLU activation # L2 regularization
    layers.BatchNormalization(),  # Add batch normalization
    layers.Dropout(0.3),  # Dropout for regularization to prevent overfitting
    layers.Dense(10, activation='softmax')  # Output layer for classification (Softmax)
])

# Display the model architecture
model.summary()
```

Fig 3.6 L2 Regularization

Learning Rate Schedules:

Gradually decreasing the learning rate during training which can help the model converge more efficiently and avoids overshooting the optimal solution. This decreases the learning rate by 4% every 100,000 steps

```
import tensorflow as tf

from tensorflow.keras.optimizers.schedules import ExponentialDecay


model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001), loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

Fig 3.7 Learning Rate Schedules

Hyperparameter Tuning:

Techniques like grid search and random search has been used to decode the optimal configuration of hyperparameters like the number of layers, number of neurons per layer, learning rate, and also batch size.

```
import keras_tuner as kt

def build_model(hp):
    model = models.Sequential([
        layers.Flatten(input_shape=(28, 28)),
        layers.Dense(hp.Int('units', min_value=32, max_value=256, step=32), activation='relu'),
        layers.Dropout(0.2),
        layers.Dense(10, activation='softmax')
    ])
    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    return model

tuner = kt.Hyperband(build_model,
                     objective='val_accuracy',
                     max_epochs=10,
                     factor=3,
                     directory='my_dir',
                     project_name='fashion_mnist_tuning')

tuner.search(X_train, y_train, epochs=10, validation_data=(X_test, y_test))

# Get the best model
best_model = tuner.get_best_models(num_models=1)[0]
```

```
Reloading Tuner from my_dir\fashion_mnist_tuning\tuner0.json
```

Fig 3.8 Hyperparameter Tuning

Performance Evaluation in MLPs

1.Accuracy: The Starting Point

Definition: It is the ratio of correctly predicted instances to the total instances.

Why It's Not Enough: In imbalanced datasets, accuracy alone can be misleading. For example, if one class dominates, a high accuracy can still mean poor performance on minority classes.

```
# Evaluate the model on the test set
test_loss, test_acc = model.evaluate(X_test, y_test)

# Print the test accuracy and loss
print(f"Test accuracy: {test_acc:.4f}")
print(f"Test loss: {test_loss:.4f}")
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8700 - loss: 0.3985
Test accuracy: 0.8663
Test loss: 0.4041
```

Fig 4.1 Test accuracy and loss

We can observe accuracy, and loss has been improved by using advanced techniques we did before.

Significance: Accuracy is a basic measure to assess if the model is making more correct than incorrect predictions, but it doesn't explain *how* the model is performing for each class.

2. Precision, Recall, and F1-Score

Precision: It Measures how many in the predicted positives are true positives.

Recall: It Measures how many in the actual positives are correctly identified.

F1-Score: here harmonic mean of precision and the recall; balances both the metrics.

```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred_labels))
```

	precision	recall	f1-score	support
0	0.82	0.83	0.82	1000
1	0.98	0.96	0.97	1000
2	0.75	0.80	0.78	1000
3	0.86	0.88	0.87	1000
4	0.77	0.80	0.78	1000
5	0.94	0.93	0.94	1000
6	0.72	0.60	0.66	1000
7	0.92	0.93	0.92	1000
8	0.94	0.98	0.96	1000
9	0.94	0.94	0.94	1000
accuracy			0.87	10000
macro avg	0.87	0.87	0.87	10000
weighted avg	0.87	0.87	0.87	10000

Fig 4.2 Classification report - Evaluation

Significance:

Here we can observe high precision means fewer false positives (important in tasks like spam detection). And also, in some cases it is observed high recall ensures fewer false negatives (critical in medical diagnoses).F1-score is balanced precision and recall, particularly useful for imbalanced datasets.

Confusion Matrix

Definition: It describes the performance of the model by comparing actual with the predicted classes.

```
from sklearn.metrics import confusion_matrix
import seaborn as sns

conf_matrix = confusion_matrix(y_test, y_pred_labels)
plt.figure(figsize=(5, 4))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=range(10), yticklabels=range(10))
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
```

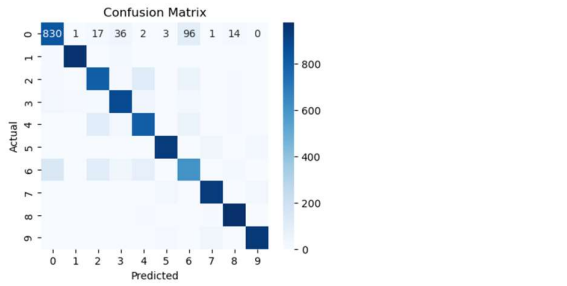

Fig 4.3 confusion Matrix for Visualising the predictions

Significance:

We can Visualize the number of correct and incorrect predictions in each class. Highlights which classes are most often confused, guiding improvements in the model.

4. Training and Validation Curves

Why: Identify overfitting or underfitting by comparing training and validation accuracy and loss.

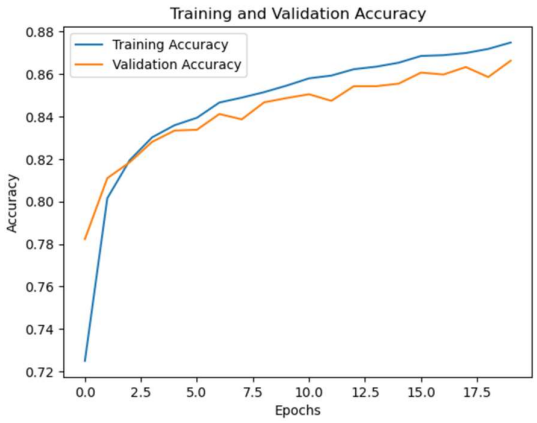

Fig 4.4 Training and Validation Accuracy plots

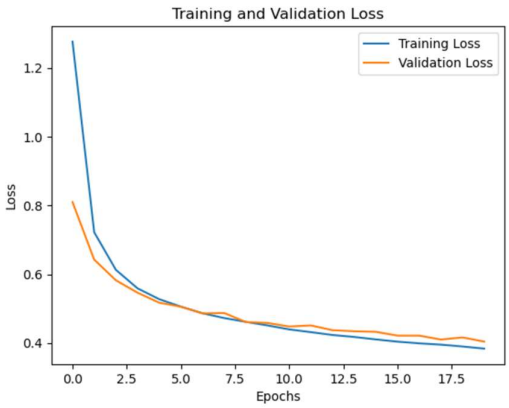

Fig 4.5 Training and Validation Loss plots

Significance:

Here training accuracy and validation accuracy continues to improve indicating best fit. (If validation accuracy stagnates or drops, then model is overfitting.) similarly in opposite with loss function

A balanced improvement in both curves indicates a well-generalized model. By observing above figure overall model is trained properly and had a good performance.

Predictions and visualisation:

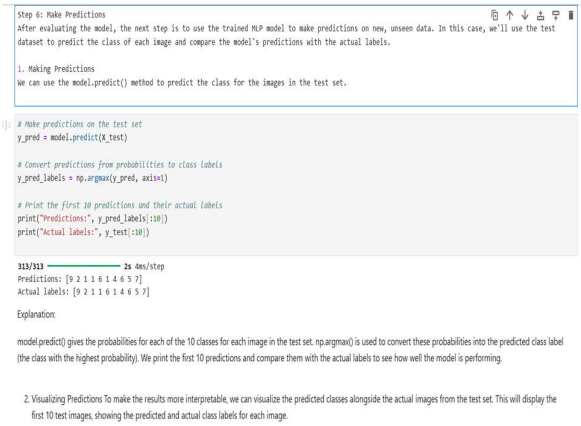

Fig 6.1 Predictions and visualisation

How ML challenges are solved using MLPs

Machine Learning (ML) has revolutionized industries engaging systems to learn from data and make data-driven predictions. Despite its transformative potential, ML models often face several significant challenges that limit their effectiveness in complex scenarios:

Non-linearity:
Many real-world problems involves non-linear relationships between input features and outputs, where traditional models like linear regression fail to capture. For instance, predicting house prices requires modelling intricate, non-linear interactions among factors like location, size, and market trends. **Multilayer Perceptrons (MLPs)** overcome this limitation by introducing the hidden layers with **non-linear activation functions** (e.g., ReLU). These layers transform the input space, enabling MLPs to solve non-linear problems like image classification, having the relationships between pixels are highly complicated.

Feature Engineering: Traditional ML approaches rely heavily on manual feature engineering to extract meaningful information from raw data, a time-intensive, also error-prone process. MLPs eliminate this dependency by **automatically learning important features** from data during training. For example, in image recognition, MLPs can detect edges, textures, and patterns directly from pixel data, progressing from low-level features to high-level abstractions.

Overfitting:
Overfitting occurs when models memorize training data instead of generalizing patterns, leads to poor performance on unseen data. MLPs address overfitting using **regularization techniques** such as:

Dropout: Randomly deactivating neurons during training to reduce reliance on specific pathways.

Early Stopping: Halting training when the validation performance plateaus.

L2 Regularization: Penalizing large weights to simplify the model.

Interpretability:
While MLPs are powerful, they are often criticized for being "black box" models. This lack of transparency could pose challenges in critical domains like healthcare, finance, where

understanding model decisions is essential. Recent advancements, such as **Layer-wise Relevance Propagation (LRP)** and **SHAP (Shapley Additive Explanations)**, enhance interpretability of MLPs, making it possible to explain predictions and gain insights into the underlying decision-making processes.

Drawbacks

Overfitting: MLPs can easily be overfit on small datasets or when regularization is not applied. This happens because the model memorizes training data instead of learning generalized patterns.

Suggestion: Ensure regularization techniques like Dropout, L2 regularization, and Early Stopping are properly implemented to overcome overfitting.

Black-box Nature

Issue: MLPs are often seen as black-box models, making it hard to interpret their decision-making process.

Suggestion: To improve interpretability, introduce methods like SHAP or Layer-wise Relevance Propagation (LRP) to visualize how the model makes predictions.

Vanishing Gradient Problem

Computational Cost

Difficulty in Scaling to Large Datasets

For more complex image data, consider using **Convolutional Neural Networks (CNNs)**. While MLPs can handle high-dimensional data, CNNs specialize in extracting hierarchical features from images, leading to better Performance

References:

- Aggarwal, C. (2018). *Neural Networks and Deep Learning: A Textbook*. Springer.
- DataCamp. (2020). *Mastering Backpropagation: A Comprehensive Guide for Neural Networks*. Retrieved from https://www.datacamp.com.
- Scikit-learn Documentation. (2021). *Varying Regularization in Multi-layer Perceptron*. Retrieved from https://scikit-learn.org.
- Chollet, F. (2017). *Deep Learning with Python*. Manning Publications.
- Towards Data Science. (2020). *Understanding the Basics of Multi-layer Perceptrons (MLPs)*. Retrieved from https://towardsdatascience.com.