Name –Sandra Maria John

Student ID - 23082951

GitHub link:

MULTI-LAYER PERCEPTRON (MLP)

1.INTRODUCTION

A **Multi-Layer Perceptron (MLP)** is a type of artificial neural network (ANN). It is commonly used in field of machine learning to perform tasks such as classification and regression. It consists of several neurons and each neuron are linked to every neuron in upcoming layers. It is a form of feedforward neural network. Through weight modifications backpropagation process, the network adjusts in weights which enable it to handle a different of computational challenges.

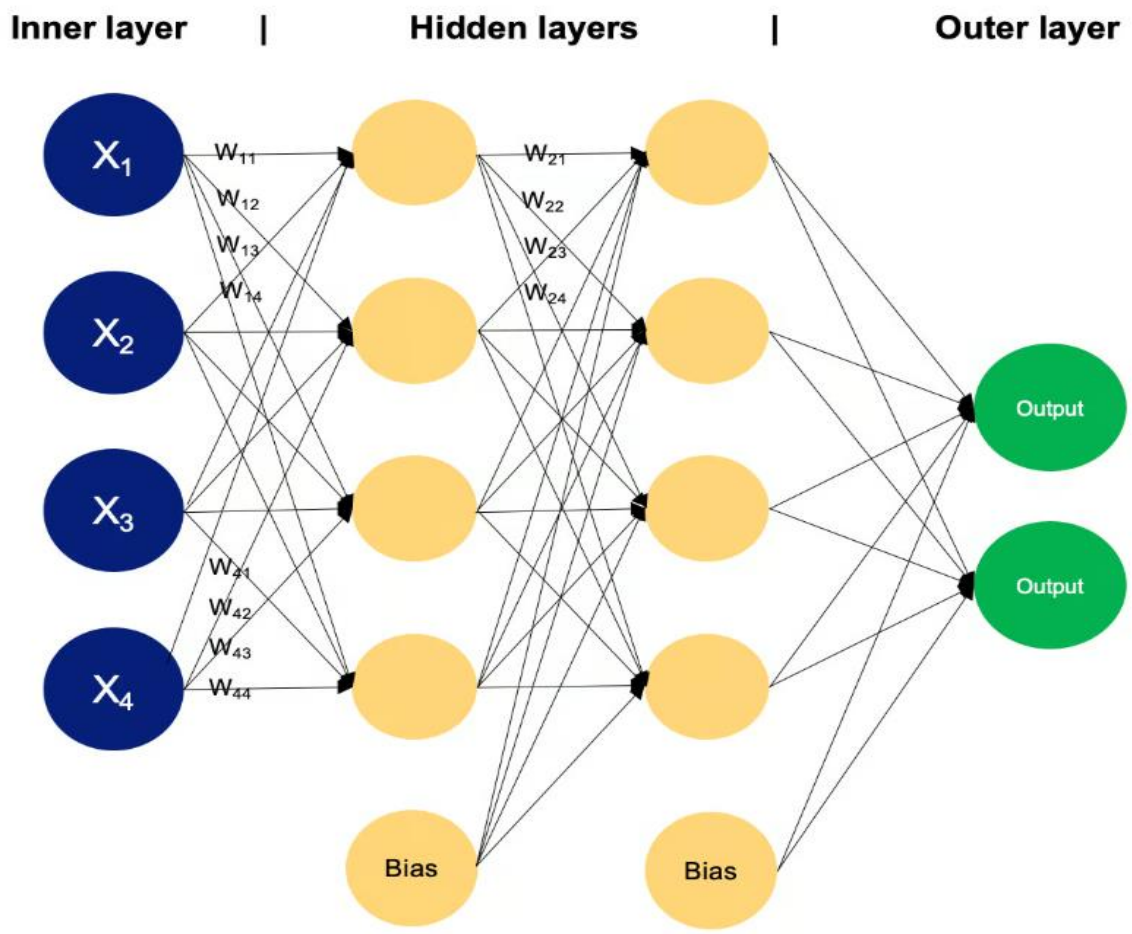
The MLP architecture is especially recognized for its potential to model complex, non-linear relationships in data, which makes it a fundamental element in contemporary deep learning. This tutorial will explore the essential keys of MLPs, examining their architecture, working principles, types of activation functions, and demonstration of how to implement an MLP using widely used machine learning libraries like **TensorFlow** and **Scikit-learn**.

What is an MLP?

A supervised learning technique that consists of several layers of neurons is called **MLP**.

- **Input Layer**: This is the first layer and entrance point of neutral network which accepts the unprocessed features of the data. It passes this feature to following layer without doing any calculations. Here each neurons indicates a feature. So, number of neurons is the number of features
- **Hidden Layer:** This layer is made of neurons, transforms the data by applying learned weights, biases, and activation functions by performing tasks. To learn more intricate patterns, a deeper MLP with more hidden layers are necessary.
- **Output Layer**: It uses an activation function such as **SoftMax or sigmoid** to generate network's final prediction.

The input layer, hidden layers, and output layer constitutes the architecture of an MLP.



Why is it called multi-layer?

MLPs have **at least one hidden layer**, hence it got the name "multi-layer". SPLs are less effective for complex patterns as it can handle only linear separable problems, whereas MLPs can deal with complex patterns as it has multiple layers to process with its non-linear activation function

Mathematical Representation

For each layer in the MLP, the following operations take place:

$$Z=W\cdot X+bZ = W \cdot X + b \; A=f(Z)A = f(Z)$$

X = input vector (either the original input or the output from the previous layer).

W = weight matrix.

b = bias term.

$f(Z)$ = activation function applied to the result.

2.ACTIVATION FUNCTION IN MLP

Activation functions in MLP introduce a non-linearity into the network by allowing the network to capture and model the complex patterns. Hence activation functions play a crucial role in MLP.

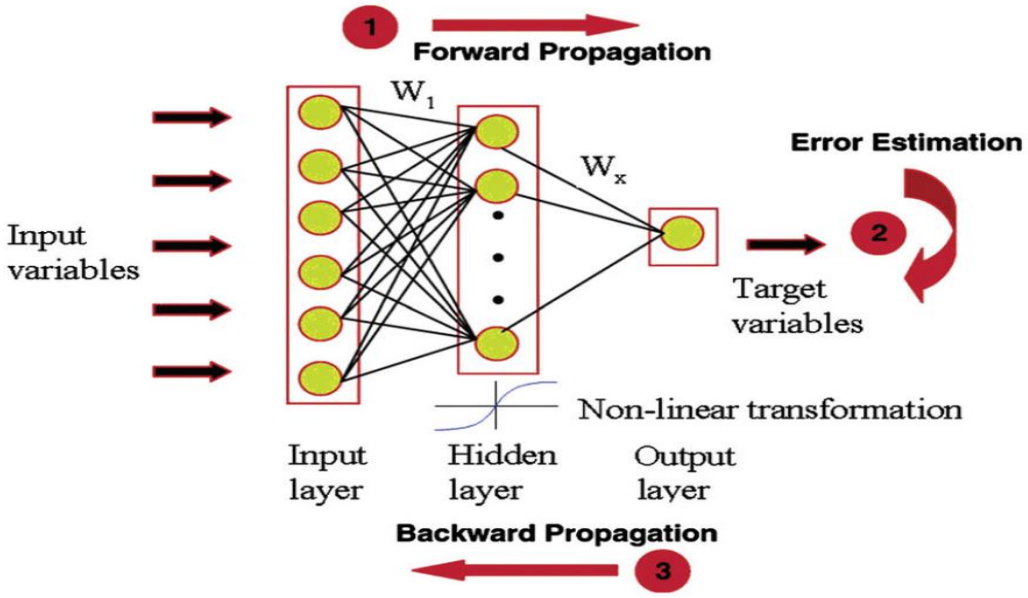
Common Activation Functions:

- **ReLU (Rectified Linear Unit)**:
  - Equation: $f(x)=\max(0,x)$ $f(x) = \max(0, x)$
  - ReLU avoids the disappearing gradient issue and enable the network to study complicated patterns. It commonly used for hidden layers

- **Sigmoid**:
  - Equation: $f(x)=11+e-xf(x) = \frac{1}{1 + e^{-x}}$
  - These are used during binary classification as it gives outputs values between 0 and 1. When input is extreme, they encounter vanishing gradient issues

- **Tanh (Hyperbolic Tangent)**:
  - Equation: $f(x)=ex-e-xex+e-xf(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
  - Tanh resembles with sigmoid but is often favored due to its better gradient behavior. It gives outputs values between -1 and 1.
- **SoftMax**:
  - It normalizes the output values to sum to1, allowing them to interpret as class probabilities. It is used for multi-class classification problems.

3. FORWARD AND BACKWARD PROPAGATION



Forward Propagation

In forward propagation, to produce final output the input data moves through the network layer by layer. To compute predictions, the network performs weighted summations, adds biases, and applies activation functions

Loss Calculation: It measures the difference the network's predictions from actual target values. The objective of training is to reduces loss. It is calculated using the functions **MSE** and **cross-entropy**

Backward Propagation

Backpropagation trains MLP by adjusting weights and calculating gradient using methods like **Gradient Descent** for optimization.

- **Gradient Calculation**: The derivative of the loss function with respect to each weight is computed using the chain rule.
- **Weight Update**: To reduce the loss function and improve the model's predictions, the weights are adjusted using optimization algorithms like SGD or Adam

4. IMPLEMENTING MLP IN PYTHON

Detailed Code Description of MLP Implementation

The given code illustrates how to develop a Multi-Layer Perceptron (MLP) using **TensorFlow/Keras** and **Scikit-learn**. For classification tasks it enables the creation, training, and evaluation of neural network models.

TensorFlow/Keras Implementation

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Defining MLP model
model = Sequential([
    Dense(64, activation='relu', input_shape=(10,)),
    Dense(32, activation='relu'),
    Dense(1, activation='sigmoid')
])

# Compiling model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Summary of the model
model.summary()
```

1. Importing Libraries: The code starts by importing various libraries such as **TensorFlow** (an open-source machine learning framework) and **Keras**.The core element of MLP, a fully connected layer is represented by **Dense** layer. For stacking layers linearly **Sequential** is imported.

2. Defining the MLP Model: A Sequential model is created Using the `Sequential` class a Sequential model is created which contains 3 layers.

- **Input Layer**: This layer is made up of **64 neurons** containing 10 features in each of them. During backpropagation, the activation function **ReLU** is used to reduce disappearing gradients issue, allowing a better gradient flow.
- **Hidden Layer**: The layer enables learning of non-linear patterns and data transformation with **ReLu** activation. This layer has **32 neurons.**
- **Output Layer**: This layer performs binary classification tasks with sigmoid activation. It gives output value lying between 0 and 1, indicating the probability of class membership. It only has a **single neuron**.

3. Compiling the Model: The model is compiled after the architecture has been defined. During the compilation phase:

Adam is the optimizer that is employed. Because of its effectiveness and minimal memory needs, Adam, an adaptive learning rate optimization method, has taken over as the standard option for many machine learning applications.

Loss Function: The binary cross-entropy loss function is applied because this is a binary classification problem. For issues where the output is either 0 or 1, this function is perfect.

Metrics: Accuracy, a popular metric for classification issues, is used to assess the model's performance.

4. Model Summary: Lastly, a summary of the model architecture is printed by calling model.summary(), which includes information on the number of layers, neurons, and parameters in the model. This stage aids the user in comprehending the model's intricacy, structure, and necessary computing power.

5. Training the Model: The model.fit() function, which accepts the training data (X_train and y_train) and trains the model for a predetermined number of epochs, is usually used after the model has been compiled, though this isn't stated explicitly in the code that is provided. The number of data processed prior to updating the model weights is specified by the batch_size argument.

Model: "sequential_2"

Layer (type)	Output Shape	Param #
dense_6 (Dense)	(None, 64)	704
dense_7 (Dense)	(None, 32)	2,080
dense_8 (Dense)	(None, 1)	33

Total params: 2,817 (11.00 KB)
Trainable params: 2,817 (11.00 KB)
Non-trainable params: 0 (0.00 B)

Scikit-learn Implementation

```python
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# Generating  dataset
X, y = make_classification(n_samples=1000, n_features=10, random_state=42)

# Spliting into training and testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Define MLP model
mlp = MLPClassifier(hidden_layer_sizes=(64, 32), activation='relu', solver='adam', max_iter=500)

# Train the model
mlp.fit(X_train, y_train)

# Calculate accuracy
accuracy = mlp.score(X_test, y_test)* 100
print(f"Test Accuracy: {accuracy:.2f}")
```

```
Test Accuracy: 85.00
```

1. Importing Libraries: The Scikit-learn method focusses on the **MLPClassifier** class and employs a distinct collection of libraries. The **train_test_split** method divides the data into training and testing sets, while the make_classification function creates a synthetic dataset. These features are derived from Scikit-learn, a well-known Python machine learning package that provides easy-to-use tools for constructing models and analyzing data.

2. Creating the Dataset: A synthetic classification dataset is created using the make_classification function. 1000 samples with 10 characteristics are produced in this instance. The model is trained using these features to divide the samples into two groups. The function is perfect for testing machine learning algorithms since it offers a rapid method of creating data without requiring real-world samples.

3. Splitting the Data: The dataset is divided into training and testing sets via the train_test_split function, with 20% going to testing and 80% going to training. To verify the model's generalization performance, it is customary to evaluate it on data that has not yet been seen.

4. Defining the MLP Model: Scikit-learn's MLP model is defined using the MLPClassifier. The following are the model's parameters:

- The hidden layer architecture is specified by the **hidden_layer_sizes** argument. In this instance, 64 and 32 neurons, respectively, are used to construct two hidden layers.

- **Activation**: ReLU is the activation function in this case. The logistic sigmoid, Tanh, and ReLU are among the activation functions that Scikit-learn's MLPClassifier supports. ReLU's ability to expedite training by resolving the vanishing gradient issue makes it a popular option.

- **Solver**: Just like in the Keras example, the Adam solver is chosen here. Many neural network models favor Adam due to its effectiveness and quickness of adaptive learning.

- **max_iter**: The optimization algorithm's maximum iteration count is specified by this option. To give the optimization process enough time to converge, the model will in this instance run for up to 500 iterations before ceasing.

5. Model Training: The model is trained using the training data (**X_train** and **y_train**) by the **fit ()** technique. The model uses gradient descent and backpropagation to get the ideal parameters (weights and biases).

6. Model Evaluation: Following model training, the model is assessed using the score () method on the test data (**X_test** and **y_test**). The model's accuracy on the test set is returned by this procedure. The score is then converted to a percentage by multiplying it by 100. This offers a straightforward assessment of the model's ability to generalize to new data.

Summary of Both Implementations

TensorFlow/Keras is typically chosen for deep learning jobs involving complicated networks because it provides more model design flexibility. Neural network construction and compilation are made easier by the Keras API, which makes it usable by both novice and expert users.

In contrast, Scikit-learn offers a high-level, intuitive interface for creating and assessing MLPs. It works well for routine machine learning tasks, while TensorFlow provides more flexibility for creating intricate or unique models.

The definition, compilation, training, and evaluation of an MLP model are demonstrated in both implementations. The intricacy of the task, the computational resources at hand, and the user's level of experience with the framework all influence the choice of framework.

5. MLP'S ADVANTAGES AND DISADVANTAGES

Advantages:

1. **Non-linearity**: By learning complex patterns, MLPs produce linear models that are more effective.
2. **Versatility**: MLPs can be used for image recognition, regression, and classification.
3. **Scalability**: With adequate data, MLPs can handle complex jobs and generalize well.

Disadvantages:

1. **Dropout** can stop overfitting, which is a problem with MLPs with many neurons.
2. **Training Time**: When dealing with big datasets, MLPs can be expensive.
3. **Interpretability**: Since MLPs are black-box models, it might be difficult to comprehend their predictions.

6. APPLICATIONS OF MLP

- **Healthcare**: Using patient data, MLPs can identify and forecast illnesses.
- **Image identification**: MLPs can perform tasks relating to images, such as digit identification and object classification.
- **Speech Recognition:** Used in early models to identify patterns in audio.
- **Finance**: financial market modelling, fraud detection, and stock price estimation.

7.STEPS TO IMPROVE MLP EFFICIENCY

- **Regularization**: Use L2 or dropout regularization to avoid overfitting.
- Standardize layer outputs by batch normalization to reduce feature variation and speed up convergence.
- **Hyperparameter tuning** involves modifying the learning rate, hidden layers, and neurons.
- **Advanced Architectures**: For difficult problems, use CNNs or RNNs.

8.CONCLUSION

MLPs are crucial models used in neural networks that are intended to manage complex tasks such as non-linear patterns in data. By examining the design, algorithms, and best practices of MLP, one can apply this potent tool for several purposes, such as regression and classification. Although MPLs are quite flexible, regularization, hyperparameter changes, and effective training methods are necessary to achieve optimal performance.

9.REFERENCE

- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.
- Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.
- Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). "Learning Representations by Back-Propagating Errors." *Nature*, 323(6088), 533–536.
- LeCun, Y., Bengio, Y., & Hinton, G. (2015). "Deep Learning." *Nature*, 521(7553), 436–444.
- Nielsen, M. (2015). *Neural Networks and Deep Learning*. Determination Press.

- Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* (2nd ed.). O'Reilly Media.
- TensorFlow Developers. (n.d.). "Neural Network Guide." Retrieved from https://www.tensorflow.org/
- Pedregosa, F., Varoquaux, G., Gramfort, A., et al. (2011). "Scikit-learn: Machine Learning in Python." *Journal of Machine Learning Research*, 12, 2825–2830.
- Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning* (2nd ed.). Springer.
- https://www.datacamp.com/tutorial/multilayer-perceptrons-in-machine-learning
- Artificial neural networks - Scientific Figure on ResearchGate https://www.researchgate.net/figure/Schematic-of-a-MLP-which-shows-the-forward-and-backward-passes-of-the-error-back_fig15_257318930