NNDL ASSIGNMENT ANSWERS MODULE -1

2. Explain the different types of Hyperparameters in a neural network and explain how fine-tuning is done.

A) Hyperparameters are user-defined parameters that control how a neural network is trained. They are not learned from the data but are set before training begins.

The first category is **architecture hyperparameters**, which define the structure of the network. These include the number of hidden layers, the number of neurons in each layer. These hyperparameters decide the model's capacity to learn complex patterns.

The second category is **optimization hyperparameters**, which control the learning process. The most important one is the learning rate, which decides how big a step is taken while updating weights. Other optimization hyperparameters include the choice of optimizer.

The third category is **regularization hyperparameters**, which help avoid overfitting. Examples include dropout rate. L1 or L2 weight regularization (penalizing large weights), and early stopping criteria (stopping training when validation loss stops improving).

Hyperparameter Fine-Tuning

Fine-tuning is the process of systematically searching for the best combination of hyperparameter values to achieve maximum performance of the neural network. The first step in fine-tuning is to select which hyperparameters to optimize, such as learning rate, number of hidden neurons, or dropout rate.

Then we choose a search method — the most common ones are grid search (trying all possible combinations), random search (sampling random combinations), and Bayesian optimization (using a probabilistic model to guide the search efficiently).

3. Explain saving and restoring a model with a code snippet.

A) When we train a neural network, we often want to **save its weights/architecture** so that we can:

- Reuse the trained model later (without retraining).
- Resume training from a checkpoint if training was interrupted.

In most deep learning frameworks (like TensorFlow/Keras, PyTorch), we can **save** and **restore** models easily.

---

1. Saving a Model

We can save:

- **Only the weights**
- **Entire model (architecture + weights + optimizer state)**

---

2. Restoring (Loading) a Model

Once saved, we can load the model and directly use it for **inference or further training**.

Code:- import tensorflow as tf

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
model = Sequential([
    Dense(16, activation='relu', input_shape=(10,)),
    Dense(8, activation='relu'),
    Dense(3, activation='softmax')
])


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

```
model.save("my_model.h5")

print(" ✅ Model saved successfully!")

restored_model = tf.keras.models.load_model("my_model.h5")

print(" ✅ Model restored successfully!")
```

4. Explain Callbacks a model with a code snippet.

A) Callbacks are special functions that are executed during the training process of a neural network at specific points, such as at the end of each batch or epoch. They are used to monitor training, adjust parameters, save models, or stop training early. Callbacks make training more efficient and help in achieving better performance.

Common examples of callbacks include ModelCheckpoint, which saves the model automatically when it achieves the best validation accuracy, EarlyStopping, which stops training if validation loss stops improving, and LearningRateScheduler.

Code:-

```
from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras.callbacks import EarlyStopping


model = Sequential([Dense(8, activation='relu', input_shape=(10,)),

Dense(3, activation='softmax')])


model.compile(optimizer='adam', loss='categorical_crossentropy')stop = EarlyStopping(monitor='val_loss', patience=2)

model.fit(X_train,y_train,epochs=10,validation_data=(X_val,y_val)callbacks=[stop])
```

5. Explain Subclassing API for dynamic models with figure and block diagram.

A) The **Subclassing API** in TensorFlow/Keras allows the creation of **custom, dynamic neural networks** by subclassing the Model class. Unlike Sequential or Functional APIs, it does not require a fixed architecture.

This flexibility allows you to design networks where the **forward pass can change dynamically**, such as conditional branching, variable-length input processing, or looping structures. This is particularly useful for research or experimental architectures like RNNs, attention models, or reinforcement learning networks.

Another advantage of subclassing is that each layer can be defined as a **class attribute**, and the forward pass is implemented in the **call() method**. This method is automatically invoked during training or inference, giving full control over how data flows through the network.

Block Diagram:

Input --> Dense(16, ReLU) --> Dense(8, ReLU) --> Output(3, Softmax)

In this block diagram, each dense layer represents a trainable layer. The input flows sequentially through the hidden layers to the output, but the subclassing API allows adding conditional paths or loops if needed.

6. Explain Functional API with respect to

a. Wide & Deep architecture.

b. Wide & Deep with multiple inputs.

c. Wide & Deep with multiple inputs and outputs. (with respective figures and code snippets.)

A) The **Functional API** in Keras provides a more flexible way of building models compared to Sequential API. Instead of stacking layers linearly, the Functional API treats **layers as functions**, allowing tensors to flow through multiple paths. This is ideal for **non-linear, branched architectures, multi-input, and multi-output networks**.

a. Wide & Deep Architecture

The Wide & Deep model combines a **wide (linear) component** and a **deep (non-linear) component**. The wide component captures **memorization**, such as frequent patterns in data, while the deep component captures **generalization**, helping the network make predictions on unseen combinations.

Block Diagram:

Input_wide ----\

--> Concatenate --> Dense --> Output

Input_deep --> Dense --> Dense --/

This shows how wide and deep components are merged via concatenation before producing the final output.

b. Wide & Deep with Multiple Inputs

Some tasks require **heterogeneous data** (e.g., user features and item features). Each input can have its own preprocessing layers before concatenation. This allows the network to **learn separate representations** for each input type, improving performance.

Block Diagram:

Input1 --> Dense \

--> Concatenate --> Dense --> Output

Input2 --> Dense /

c. Wide & Deep with Multiple Inputs and Outputs

In **multi-task learning**, a single network can generate multiple outputs from shared hidden layers. This allows the model to **learn correlated tasks together**, improving overall performance.

Block Diagram:

Input1 --> Dense \

--> Concatenate --> Dense --> Output1

Input2 --> Dense / --> Output2

This illustrates how shared layers can be used for multiple outputs, with each output having potentially different loss functions.

---

7. Explain MLP with respect to a Classification and Regression task along with necessary block diagram and code snippet.

A) A **Multi-Layer Perceptron (MLP)** is a feedforward neural network that can be applied to both **classification** and **regression** tasks. Classification involves predicting discrete labels, while regression involves predicting continuous values. The architecture of MLP remains similar, but the **output layer and loss function** change depending on the task.

For **classification**, the output layer typically uses a **softmax** activation (for multi-class) or **sigmoid** activation (for binary classification). The network learns patterns from input features to predict class probabilities. For **regression**, the output layer is linear, and the network predicts continuous values.

Block Diagram:

Input --> Dense(16, ReLU) --> Dense(8, ReLU) --> Output

This shows that the hidden layers extract **non-linear features** from the input, which are then used for final predictions. The MLP can be adapted to various problem types simply by adjusting the output layer.

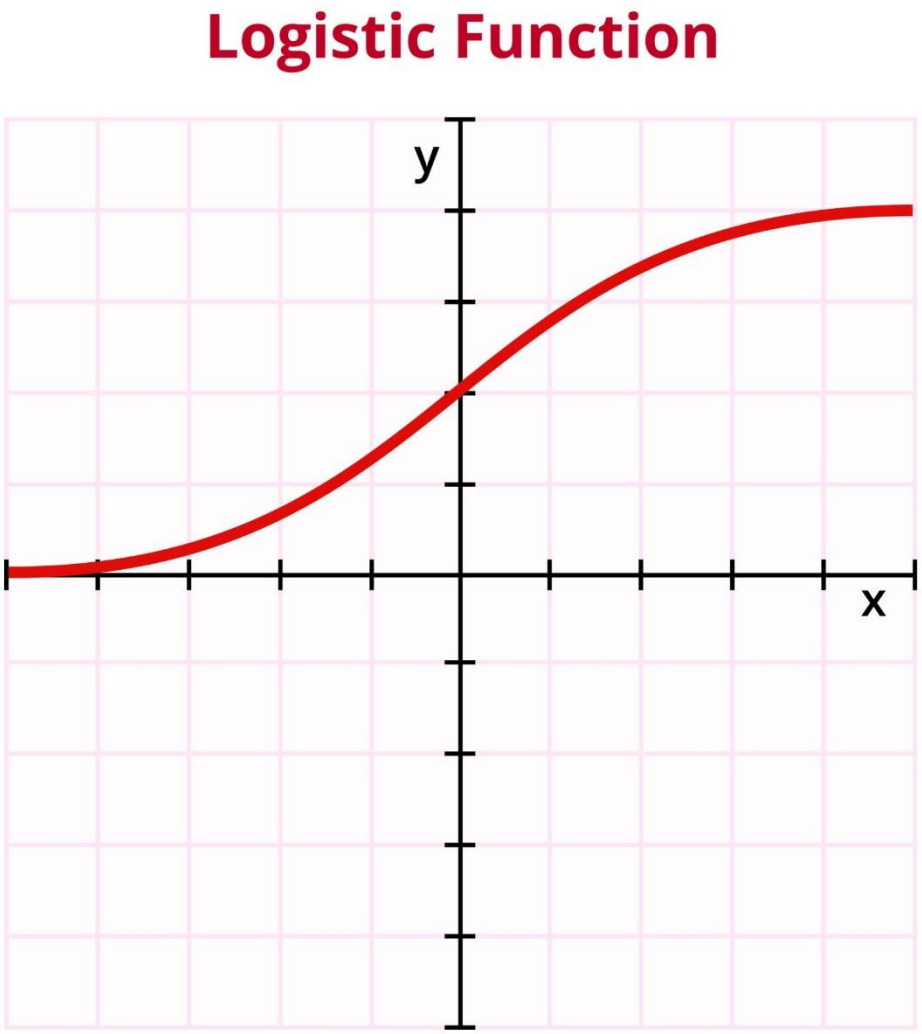
8. Explain with plots 3 most popularly used activation functions. And justify the need of activation functions.

A) They introduce non-linearity, allowing the network to learn complex patterns and relationships in data that a simple linear model couldn't capture.

1. Sigmoid Function

The **sigmoid function**, also known as the logistic function, squashes its input into a range between 0 and 1. This makes it useful for models where the output needs to be a probability.

Function: $f(x)=1+e-x1$ **Plot:**



Justification: While once widely used, its popularity has waned due to the **vanishing gradient problem**. As the input becomes very large (positive or negative), the derivative of the function approaches zero, which makes it difficult for the network to learn during backpropagation.

---

2. Rectified Linear Unit (ReLU)

The **ReLU** function is one of the most popular activation functions in deep learning today. It's computationally efficient and has largely solved the vanishing gradient problem.

Function: $f(x)=\max(0,x)$ **Plot:**

Justification: The function is very simple: it returns 0 for any negative input and the input value itself for any positive input. This linearity for positive values helps to avoid the vanishing gradient problem, making it faster to train and more effective for deep networks. However, it can suffer from the **"dying ReLU"** problem, where neurons can become permanently inactive if they only receive negative inputs.

---

3. Hyperbolic Tangent (tanh)

The **tanh** function is an alternative to the sigmoid function. It's similar but maps the input to a range between -1 and 1.

Function: f(x)=ex+e−xex−e−x **Plot:**

Justification: The tanh function is zero-centered, which can make training easier and faster than with the sigmoid function. The gradients are stronger than sigmoid, but it still suffers from a similar **vanishing gradient problem** for very large or very small inputs. It's often a better choice than sigmoid for hidden layers.

9. Explain step-by-step process involved in training of Single layered perceptron.

A) Here's the step-by-step process for training a single-layer perceptron:

1. Initialization

The process begins by initializing the perceptron's parameters.

- Weights (w): Each input feature is assigned a weight. These are typically initialized to small random values or zeros.
- Bias (b): The bias is a single value that helps shift the decision boundary. It is also initialized to a small random value or zero.
- Learning Rate (η): A small positive constant (usually between 0.0 and 1.0) is chosen. This controls how much the weights and bias are adjusted during each update. A smaller learning rate means smaller adjustments.

---

2. Iterative Training

The perceptron iterates through the training dataset, one example at a time. This process is repeated for multiple epochs (a full pass through the entire dataset) until the model converges or a maximum number of epochs is reached.

For each training example:

a. Calculate the Weighted Sum: The perceptron computes a weighted sum of the inputs. This is the core "feed-forward" step. $z=(x1*w1)+(x2*w2)+...+(xn*wn)+b$ This can be written in vector form as: $z=x \cdot w+b$

b. Apply the Activation Function: The weighted sum, z, is passed through an activation function, typically a step function for the classic perceptron. This function produces the predicted output, y^. $\hat{y}=step(z)=\{10if\ z\geq 0if\ z<0$

c. Calculate the Error: The predicted output ($\hat{y}$) is compared to the actual, or "target," output (y) from the training data to calculate the error. $e=y-\hat{y}$

- If e=0, the prediction was correct, and no changes are made to the weights or bias.
- If e≠0, an error occurred, and the weights and bias must be updated.

The algorithm repeats these steps for all examples in the training set until all are classified correctly. The perceptron is guaranteed to converge and find a solution if, and only if, the data is linearly separable (meaning a single straight line can perfectly divide the two classes).

10.

a) Explain the limitations of Single-Layer Perceptrons (SLPs) using the OR and XOR problems. Support your explanation with appropriate plots to illustrate the differences.

b) Discuss the solution to overcome these limitations and explain how it addresses the problem.

A) a) Limitations of Single-Layer Perceptrons (SLPs)

The primary limitation of a single-layer perceptron (SLP) is that it can only solve linearly separable problems. This means it can only classify data that can be

perfectly separated by a single straight line (or a hyperplane in higher dimensions).

- The OR Problem: The OR function is a great example of a problem that a single-layer perceptron can solve. As shown in the plot, the data points for the OR function (where the output is 1 if at least one input is 1) can be divided into two classes by a single straight line.
- The XOR Problem: The XOR (exclusive OR) problem, however, is not linearly separable. The XOR function outputs a 1 only when the inputs are different (e.g., 0,1 or 1,0) and a 0 when they are the same (e.g., 0,0 or 1,1). When these points are plotted, the two classes (0 and 1) cannot be separated by a single straight line. A single-layer perceptron, restricted to finding a single linear boundary, will fail to correctly classify all four points. This was a major setback in the early days of neural network research, leading to what's known as the "AI winter."

---

b) Solution: Multi-Layer Perceptron (MLP)

The solution to the limitations of SLPs is to use a multi-layer perceptron (MLP), also known as a feed-forward neural network. This architecture introduces one or more hidden layers between the input and output layers.

The hidden layer's role is to transform the input data into a new, higher-dimensional representation where the problem becomes linearly separable. This transformation is made possible by incorporating non-linear activation functions

For the XOR problem, a hidden layer can learn to create two separate linear boundaries, effectively "bending" the decision surface to correctly classify all the data points. The output layer then takes these transformed features from the hidden layer and performs a final linear classification, which can now easily separate the data.