DEEP LEARNING NOTES

Module–I: Machine Learning Basics

Capacity, Over fitting and Under fitting, Bias and Variance, Hyper parameters and Validation Sets, Estimators, , Gradient-Based Learning, Stochastic Gradient Descent, Building a Machine Learning Algorithm, Challenges and Motivation of Deep Learning, Deep Feed forward Networks Learning using XOR, Hidden Units, Architecture Design, Maximum Likelihood Estimation, Back-Propagation Differentiation Algorithm.

CAPACITY :

In machine learning, capacity refers to a model's ability to fit a wide variety of functions or patterns in the data. It reflects how complex a model is and how much information it can capture.

Types of Capacity :

1. Low Capacity:
    - A model with low capacity cannot learn complex patterns in the data.
    - It may fail to fit the training data properly, leading to under fitting.
    - Example: Using a straight line (linear model) to capture a highly curved relationship.

2. High Capacity:
    - A high-capacity model can fit complex patterns, even very critical details.
    - However, it may memorize the training data instead of generalizing to new data, leading to overfitting.
    - Example: A very deep neural network fitting noise in the training data.

3. Right Capacity:
    - A model with the "right" capacity can learn meaningful patterns in the data and generalize well to unseen data.
    - This balance is often achieved through proper model selection and regularization techniques.

Factors Influencing Capacity :

1. Model Complexity:
    - More complex models (e.g., deeper neural networks, higher-degree polynomials) have higher capacity.

2. Number of Parameters:
    - Models with more parameters (e.g., weights in a neural network) have higher capacity.

3. Training Data Size:
    - A high-capacity model may require more data to avoid over fitting.

Trade-Off: Bias vs. Variance :

- Bias: Error due to over simplification (low capacity).
- Variance: Error due to sensitivity to small data variations (high capacity).

An ideal model minimizes both bias and variance, achieving a good balance between under fitting and over fitting.

Overfitting and Underfitting :

Overfitting (Too much learning, bad for new data)

- Overfitting happens when the model learns too much from the training data, including noise and unnecessary details.
- It performs very well on the training data but poorly on new (test) data.
- The model becomes too complex and memorizes instead of understanding patterns.

Example :

Imagine a student memorizing every question and answer from a textbook. If the exam has the exact same questions, they will do well. But if the exam has new questions, they will struggle because they don't understand the concepts.

Signs of Overfitting :

- Training accuracy is very high, but test accuracy is low.
- The model makes very specific predictions that don't generalize well.

How to Fix Overfitting?

- Use more training data (if possible).
- Reduce model complexity (simpler model).
- Use regularization (L1/L2 techniques to prevent extreme weights).
- Use dropout (in neural networks, randomly remove some connections).

Underfitting (Too little learning, bad for all data)

- Underfitting happens when the model doesn't learn enough from the training data.
- It performs poorly on both training and test data.
- The model is too simple and fails to capture important patterns.

Example :

Imagine a student who only reads the textbook summary. They don't memorize or understand much, so they perform poorly on both practice and real exams.

Signs of Underfitting

- Both training accuracy and test accuracy are low.
- The model fails to find patterns in the data.

How to Fix Underfitting?

- Use a more complex model.
- Train for more epochs (in deep learning).
- Use more features or better data.

Comparison Table :

Feature	Overfitting	Underfitting
Model Complexity	Too complex	Too simple
Training Accuracy	High	Low
Test Accuracy	Low	Low
Cause	Memorizing noise	Ignoring patterns
Solution	Simplify the model	Make the model more complex

Best Model (Balanced Fit)

- A good model finds the right balance between overfitting and underfitting.
- It learns the patterns in training data but can also perform well on new data.

Bias and Variance :

Bias and variance are two important concepts in machine learning. They help us understand how well a model is learning from data.

1. Bias (Too Simple, Can't Learn Well)

- Bias means the model does not learn enough from the training data.
- It makes strong assumptions and ignores important patterns in the data.
- A high-bias model is too simple and cannot capture the complexity of the data.

Example :

Imagine a student who only studies basic math and tries to solve advanced problems. They will make many mistakes because they did not learn enough.

Signs of High Bias

- Training accuracy is low (bad performance on training data).
- Test accuracy is also low (bad performance on new data).

How to Fix High Bias?

- Use a more complex model.
- Train the model for more time.
- Use better features (more useful data).

2. Variance (Too Complex, Over thinks Data)

- Variance means the model learns too much from the training data, including noise and unnecessary details.
- It becomes too sensitive to small changes in the data.
- A high-variance model is too complex and cannot generalize well to new data.

Example :

Imagine a student who memorizes an entire textbook, including typos and mistakes. They might do well on practice questions but fail an actual exam because they over think the answers.

Signs of High Variance

- Training accuracy is very high (model memorizes training data).
- Test accuracy is low (model fails on new data).

How to Fix High Variance?

- Use a simpler model.
- Use more training data.
- Apply regularization (L1/L2 techniques to reduce complexity).

Bias-Variance Tradeoff (Finding the Right Balance)

A good machine learning model should have a balance between bias and variance.

- If bias is too high → the model does not learn enough (underfitting).
- If variance is too high → the model learns too much (overfitting).
- We need to find a sweet spot where the model learns just enough to generalize well.

Comparison Table

Feature	High Bias	High Variance
Model Complexity	Too simple	Too complex
Training Accuracy	Low	High
Test Accuracy	Low	Low
Cause	Ignoring patterns	Memorizing training data
Solution	More complex model	Simpler model, more data

High Bias → Model is too simple, makes a lot of errors.
High Variance → Model is too complex, memorizes training data but fails on new data.
Balanced Model → Learns the right amount, performs well on both training and test data.

Hyper parameters and Validation Sets :

When training a machine learning model, we need to tune it properly to get the best performance. Two important concepts for this are hyper parameters and validation sets.

1. Hyper parameters (Model Settings We Choose Before Training)

- Hyper parameters are the settings that we choose before training a machine learning model.
- They control how the model learns from data.
- Unlike normal parameters (like weights in a neural network), hyper parameters are not learned by the model. We have to set them manually.

Examples of Hyper parameters

- Learning Rate – Controls how fast the model updates itself.
- Number of Trees (for Random Forest) – Decides how many trees the model will use.
- Number of Hidden Layers (for Neural Networks) – Determines the depth of the model.
- Batch Size – Decides how many samples the model processes at a time.
- Regularization (L1/L2, Dropout) – Prevents overfitting by limiting complexity.

Why Are Hyper parameters Important?

- If we choose the wrong values, the model won't perform well.
- We need to test different values to find the best combination.

2. Validation Set (Testing Model Performance Before Final Test)

- A validation set is a small part of the dataset used to tune the model.
- It helps check how well the model is performing before testing it on completely new data.
- This prevents us from choosing a model that performs well only on training data but fails on real-world data.

How Data is Split?

Dataset (Full data)
Training Set – Used to train the model.

Validation Set – Used to tune the model (adjust hyper parameters).
Test Set – Used to check final accuracy (never seen during training).

Why Use a Validation Set?

- Prevents over fitting – Helps ensure the model will work well on new data.
- Helps tune hyper parameters – We can test different values and see what works best.
- Improves final model performance – Ensures we pick the right model before testing.

Hyper parameters and Validation Set Work Together!

- Choose a model and set initial hyper parameters.
- Train the model on the training set.
- Test the model on the validation set.
- Adjust hyper parameters if the validation performance is not good.
- Repeat steps 2-4 until the best hyper parameters are found.
- Final test on the test set.

What is an Estimator?

In machine learning, an estimator is an object (usually a class or function) that implements algorithms for learning and prediction. It is used to fit a model to the data and make predictions.

Key Functions of an Estimator

1. Fit: Learn patterns from the training data.
2. Predict: Make predictions on new, unseen data.
3. Evaluate: Measure the model's performance.
4. Transform: For some estimators (like scalers), modify data (e.g., scaling or encoding).

Examples

1. In scikit-learn, LinearRegression() is an estimator for linear regression.
2. In TensorFlow/Keras, models (like Sequential) are estimators.
3. In PyTorch, an estimator is a custom class that wraps the training and evaluation steps.

Types of Estimators

1. Parametric Estimators

- Assumes the data follows a known distribution with a fixed number of parameters.
- Examples:
    - Linear regression
    - Logistic regression

2. **Non-Parametric Estimators**

- Does not assume any specific data distribution.
- Can adapt to more complex data patterns but may require more data to perform well.
- Examples:
    - k-Nearest Neighbors (k-NN)
    - Decision Trees

4. **Supervised Learning Estimators**

- Require labeled data (input-output pairs).
- Examples:
    - Classification: Support Vector Machines (SVM), Random Forest
    - Regression: Ridge Regression, Gradient Boosting

5. **Unsupervised Learning Estimators**

- Work with unlabeled data.
- Examples:
    - Clustering: k-Means, DBSCAN
    - Dimensionality Reduction: PCA, t-SNE

6. **Transformers**

- A special type of estimator that modifies data.
- Examples:
    - StandardScaler (scales features)
    - OneHotEncoder (converts categorical data to numeric)

Example: scikit-learn Estimator Workflow

Training a Model

```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

# Example dataset
X, y = load_data()

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize the estimator
```

```
model = LinearRegression()

# Fit the model
model.fit(X_train, y_train)
```
2. Making Predictions
python
CopyEdit
```
# Predict on test data
y_pred = model.predict(X_test)
```
3. Evaluating the Model
```
from sklearn.metrics import mean_squared_error

# Calculate performance
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse}")
```

Differences Between Estimators, Transformers, and Models

Type	Purpose	Examples
Estimator	General term for objects that fit data.	Linear Regression, Random Forest
Transformer	Specifically modifies data.	Standard Scaler, PCA
Model	A trained estimator used for predictions.	Logistic Regression, SVM

What is Gradient-Based Learning?

Gradient-based learning is an optimization method where a model updates its parameters (weights and biases) based on the gradient (derivative) of a loss function. The goal is to find the best parameters that minimize the loss.
It is primarily used in neural networks, deep learning, and optimization algorithms like gradient descent.

Key Concepts in Gradient-Based Learning

1. Loss Function (Objective Function)
   - The function that measures how well the model's predictions match the actual values.
   - The goal is to minimize this function.
   - Examples:
     - Mean Squared Error (MSE) for regression.
     - Cross-Entropy Loss for classification.

2. Gradient (Derivative)
   - The slope of the loss function with respect to the model parameters.

- Helps determine the direction to update weights.

3. Learning Rate (Step Size)
   - Controls how much the model updates its parameters in each step.
   - Too high → May overshoot the minimum.
   - Too low → Converges too slowly or gets stuck in local minima.

4. Optimization Algorithm

- An algorithm that adjusts model parameters using gradients.
- Example: Gradient Descent (most common).

What is Gradient Descent?

Gradient Descent (GD) is an iterative optimization algorithm that updates parameters using the gradient of the loss function.
Formula (For parameter θ\ thetaθ):

$$\theta = \theta - \alpha \cdot \partial L \ \partial \theta$$

Where:
- θ\theta θ = Model parameter (weight/bias).
- α\alpha α = Learning rate.
- ∂L ∂θ= Gradient of the loss function L.

Types of Gradient Descent

1. Batch Gradient Descent (BGD)
   - Computes the gradient using the entire dataset.
   - Pros: More stable updates.
   - Cons: Slow for large datasets.

2. Stochastic Gradient Descent (SGD)
   - Updates model parameters for each individual data point.
   - Pros: Faster and works well for large datasets.
   - Cons: More noisy updates, may not converge smoothly.

3. Mini-Batch Gradient Descent
   - Uses a small batch of data points to compute the gradient.
   - Pros: Balances speed and stability.
   - Cons: Choosing the right batch size is crucial.

4. Adaptive Learning Rate Methods
   - Adjusts the learning rate dynamically.
   - Popular methods:
     - Adagrad: Adapts learning rate for each parameter.
     - RMSprop: Uses moving average of squared gradients.

- Adam (Adaptive Moment Estimation): Combines momentum and RMSprop.

Applications of Gradient-Based Learning

1. Neural Networks (Deep Learning)
   - Backpropagation uses gradients to update weights in deep neural networks.
2. Regression Models
   - Gradient descent is used to minimize MSE in linear and logistic regression.
3. Computer Vision
   - Convolutional Neural Networks (CNNs) use gradient-based learning to recognize images.
4. Natural Language Processing (NLP)
   - Transformers and recurrent neural networks (RNNs) rely on gradient optimization for text processing.
5. Reinforcement Learning
   - Policy gradient methods use gradients to improve decision-making models.

EXAMPLE :

```
import numpy as np
# Data (X: input, y: output)
X = np.array([1, 2, 3, 4, 5])
y = np.array([2, 4, 6, 8, 10]) # y = 2x
# Initialize w and b
w, b = 0, 0
# Hyperparameters
learning_rate = 0.1
epochs = 10 # Number of iterations
# Gradient Descent
for i in range(epochs):
y_pred = w * X + b # Prediction
error = y_pred - y # Error
# Compute gradients
dw = (2 / len(X)) * np.sum(X * error)
db = (2 / len(X)) * np.sum(error)
# Update parameters
w -= learning_rate * dw
b -= learning_rate * db
# Print progress
loss = np.mean(error**2) # Mean Squared Error
print(f"Epoch {i+1}: w = {w:.2f}, b = {b:.2f}, Loss = {loss:.2f}")
# Final optimized parameters
print(f"\nOptimized: w = {w:.2f}, b = {b:.2f}")
```

Output :

```
Epoch 1: w = 2.20, b = 0.60, Loss = 44.00
Epoch 2: w = 1.52, b = 0.28, Loss = 9.82
Epoch 3: w = 1.86, b = 0.38, Loss = 2.19
Epoch 4: w = 1.96, b = 0.33, Loss = 0.49
Epoch 5: w = 1.99, b = 0.26, Loss = 0.11
Epoch 6: w = 2.00, b = 0.20, Loss = 0.02
Epoch 7: w = 2.00, b = 0.15, Loss = 0.01
Epoch 8: w = 2.00, b = 0.11, Loss = 0.00
Epoch 9: w = 2.00, b = 0.08, Loss = 0.00
Epoch 10: w = 2.00, b = 0.06, Loss = 0.00
Optimized: w = 2.00, b = 0.06
```

Stophastic Gradient Decent

```
import numpy as np
# Data (X: input, y: output)
X = np.array([1, 2, 3, 4, 5])
y = np.array([2, 4, 6, 8, 10]) # y = 2x
# Initialize w and b
w, b = 0, 0
# Hyperparameters
learning_rate = 0.1
epochs = 10 # Number of passes through the dataset
# Stochastic Gradient Descent
for epoch in range(epochs):
for i in range(len(X)): # Update w, b for each data point
xi, yi = X[i], y[i] # Select one data point
y_pred = w * xi + b # Prediction
error = y_pred - yi # Error
# Compute gradients
dw = 2 * xi * error
db = 2 * error
# Update parameters
w -= learning_rate * dw
b -= learning_rate * db
# Print progress
total_loss = np.mean((w * X + b - y) ** 2)
print(f"Epoch {epoch+1}: w = {w:.2f}, b = {b:.2f}, Loss = {total_loss:.2f}")
# Final optimized parameters
print(f"\nOptimized: w = {w:.2f}, b = {b:.2f}")
```

Output :

```
Epoch 1: w = 2.09, b = 0.38, Loss = 2.04
Epoch 2: w = 2.02, b = 0.14, Loss = 0.20
```

Epoch 3: w = 2.00, b = 0.05, Loss = 0.02
Epoch 4: w = 2.00, b = 0.02, Loss = 0.00
Epoch 5: w = 2.00, b = 0.01, Loss = 0.00
Epoch 6: w = 2.00, b = 0.00, Loss = 0.00
Epoch 7: w = 2.00, b = 0.00, Loss = 0.00
Epoch 8: w = 2.00, b = 0.00, Loss = 0.00
Epoch 9: w = 2.00, b = 0.00, Loss = 0.00
Epoch 10: w = 2.00, b = 0.00, Loss = 0.00
Optimized: w = 2.00, b = 0.00
Explanation
1. Start with random values: w=0, b=0
2. Loop through each data point (one by one)
3. Compute gradients dw, db for that data point
4. Update w and b immediately after processing each point
5. Repeat for 10 epochs until w≈2., b≈0.
6. Loss decreases over time, showing that the model is learning

Difference Between Gradient Descent and Stochastic Gradient Descent (SGD)	**Gradient Descent (GD)**	**Stochastic Gradient Descent (SGD)**
Feature		
Updates per Epoch	Updates parameters once per epoch (using all data points)	Updates parameters after each data point
Speed of Convergence	Slower, but smoother	Faster, but fluctuates
Computational Cost	High (computes gradients for all data points)	Low (computes gradients for only one data point)
Memory Usage	High (stores entire dataset in memory)	Low (only needs one data point at a time)
Stability	Smooth, stable convergence	Noisy, fluctuates more
Chance of Getting Stuck in Local Minima	More likely (follows the same path every time)	Less likely (random updates help escape local minima)
Suitability for Large Datasets	Not suitable (slow for big data)	Very suitable (updates quickly)
Implementation Complexity	Easier to implement	Slightly more complex (needs randomized data selection)
Loss Curve Behavior	Smooth and gradually decreasing	Noisy but decreasing over time
Example Use Case	Small datasets, batch learning	Large datasets, online learning

Building a machine learning algorithm

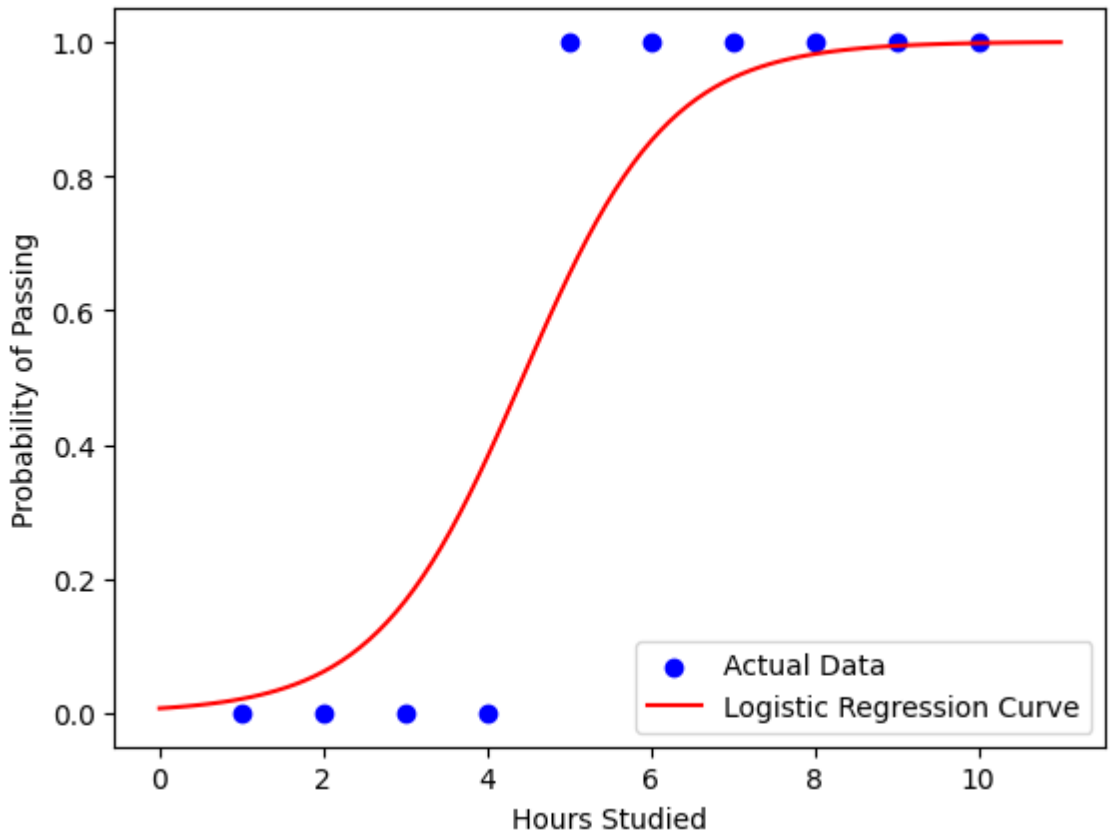
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.metrics import accuracy_score, confusion_matrix
# Sample dataset: Hours Studied vs Pass/Fail
data = {'Hours Studied': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
'Pass': [0, 0, 0, 0, 1, 1, 1, 1, 1, 1]} # 0 = Fail, 1 = Pass
df = pd.DataFrame(data)
# Split into features (X) and target labels (Y)
X = df[['HoursStudied']]
Y = df['Pass']
# Train-Test Split (80% Training, 20% Testing)
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=42)
# Create and train the model
model = LogisticRegression()
model.fit(X_train, Y_train)
# Predict on test data
Y_pred = model.predict(X_test)
# Calculate Accuracy
accuracy = accuracy_score(Y_test, Y_pred)
print("Model Accuracy:", accuracy)
# Confusion Matrix
conf_matrix = confusion_matrix(Y_test, Y_pred)
print("Confusion Matrix:\n", conf_matrix)
# Plot the data points
plt.scatter(X, Y, color='blue', label="Actual Data")
# Plot the logistic regression curve
X_range = np.linspace(0, 11, 100).reshape(-1, 1)
Y_prob = model.predict_proba(X_range)[:, 1] # Probability of passing
```

```
plt.plot(X_range, Y_prob, color='red', label="Logistic Regression Curve")
plt.xlabel("Hours Studied")
plt.ylabel("Probability of Passing")
plt.legend()
plt.show()
```

OUTPUT :

```
Model Accuracy: 1.0
Confusion Matrix:
[[1 0]
[0 1]]
```



Challenges Motivating Deep Learning

1. Complex Data Patterns
    - Challenge: Traditional machine learning algorithms often struggle to capture complex patterns and relationships in high-dimensional data.
    - Motivation for Deep Learning: Deep learning models, particularly neural networks, can learn hierarchical representations of data, enabling them to identify intricate patterns through multiple layers of abstraction.
2. High-Dimensional Data

- Challenge: Datasets, such as images, audio, and text, are often high-dimensional and unstructured, making it difficult for traditional models to process them effectively.
- Motivation for Deep Learning: Deep learning architectures, like convolutional neural networks (CNNs) for images and recurrent neural networks (RNNs) for sequences, are specifically designed to handle high-dimensional inputs.

3. Feature Engineering Limitations
   - Challenge: Manual feature engineering can be time-consuming and requires domain expertise, limiting scalability and flexibility.
   - Motivation for Deep Learning: Deep learning automates the process of feature extraction, allowing models to learn relevant features directly from raw data, thus reducing the need for extensive manual feature engineering.
4. Non-linearity in Relationships
   - Challenge: Many real-world problems exhibit non-linear relationships that are hard to model with linear algorithms.
   - Motivation for Deep Learning: Neural networks introduce non-linearity through activation functions, enabling them to model complex non-linear relationships effectively.
5. Large Amounts of Data
   - Challenge: Traditional models often perform poorly with large datasets due to overfitting or computational inefficiency.
   - Motivation for Deep Learning: Deep learning models thrive on large datasets, as they can learn more robust and generalized representations, leading to improved performance and accuracy.
6. Computational Power
   - Challenge: Training complex models on large datasets can be computationally expensive and time-consuming.
   - Motivation for Deep Learning: Advances in hardware (GPUs, TPUs) and optimization techniques (e.g., batch normalization, dropout) have made it feasible to train deep learning models on large datasets efficiently.
7. Transfer Learning
   - Challenge: Training models from scratch can be impractical due to data scarcity in certain domains.
   - Motivation for Deep Learning: Transfer learning allows models to leverage pre-trained networks on large datasets, enabling them to be fine-tuned for specific tasks with smaller datasets, improving performance.
8. End-to-End Learning
   - Challenge: Many traditional models require multiple stages of processing, complicating the pipeline.
   - Motivation for Deep Learning: Deep learning facilitates end-to-end learning, where raw input can be directly transformed into output, simplifying model architecture and deployment.
9. Robustness to Noisy Data
   - Challenge: Traditional models may be sensitive to noise and outliers in data.

- Motivation for Deep Learning: Deep learning architectures can inherently learn to ignore irrelevant features and noise, leading to improved robustness and generalization.

10. Real-time Processing
    - Challenge: Applications requiring real-time decision-making (e.g., autonomous driving, fraud detection) are often challenging with traditional algorithms.
    - Motivation for Deep Learning: Deep learning models can process vast amounts of data in real time, making them suitable for applications that demand quick responses.

How Neural Networks Solve the XOR Problem

- The XOR (exclusive OR) is a simple logic gate problem that cannot be solved using a single-layer perceptron (a basic neural network model). We can solve this using neural networks. Neural networks are powerful tools in machine learning.
- •What is the XOR Problem?
- The XOR operation is a binary operation that takes two binary inputs and produces a binary output. The output of the operation is 1 only when the inputs are different.

Input A	Input B	XOR Output
0	0	0
0	1	1
1	0	1
1	1	0

The main problem is that a single-layer perceptron cannot solve this problem because the data is not linearly separable i.e. we cannot draw a straight line to separate the output classes (0s and 1s).

Why Single-Layer Perceptrons Fail?

A single-layer perceptron can solve problems that are linearly separable by learning a linear decision boundary.
Mathematically, the decision boundary is represented by:
$y=\text{step}(w\cdot x+b)$ $y=\text{step}(w\cdot x+b)$
Where:
w is the weight vector.
x is the input vector.
b is the bias term.
Step step is the activation function, often a Heaviside step function that outputs 1 if the input is positive and 0 otherwise.

- For linearly separable data, the perceptron can adjust the weights ww and bias b*b* during training to correctly classify the data. However, because XOR is not linearly separable, no single line (or hyperplane) can separate the outputs 0 and 1.

How Multi-Layer Neural Networks Solve XOR

A multi-layer neural network which is also known as a feed forward neural network or multi-layer perceptron is able to solve the XOR problem. There are multiple layer of neurons such as input layer, hidden layer, and output layer.
The working of each layer:
Input Layer: This layer takes the two inputs (A and B).
Hidden Layer: This layer applies non-linear activation functions to create new, transformed features that help separate the classes.
Output Layer: This layer produces the final XOR result.

Ex :

```
import numpy as np
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
# XOR dataset
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
Y = np.array([[0], [1], [1], [0]])
# Create a simple neural network
model = Sequential([
    Dense(4, activation='relu', input_shape=(2,)),  # Hidden layer with 4 neurons
Dense(1, activation='sigmoid')  # Output layer with 1 neuron
])
# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
# Train the model
model.fit(X, Y, epochs=500, verbose=0)  # Train for 500 epochs
# Test the model
predictions = model.predict(X)
print("Predictions:")
print(np.round(predictions))  # Round output to 0 or 1
```

Output :

1/1 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 168ms/step - accuracy: 0.5000 - loss: 0.6931
Accuracy: 50.00%
1/1 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 46ms/step
Predictions:
Input: [0 0] => Predicted Output: [0], Actual Output: [0]
Input: [0 1] => Predicted Output: [0], Actual Output: [1]
Input: [1 0] => Predicted Output: [0], Actual Output: [1]
Input: [1 1] => Predicted Output: [0], Actual Output: [0]

Hidden Units & Architecture Design in Neural Networks :

- Neural Networks (NNs) are composed of layers of neurons.
- Each neuron processes inputs using a weight and an activation function.
- The architecture of an NN defines how neurons are arranged and connected.

Layers in a Neural Network

A typical neural network has three types of layers:

1. Input Layer
    - Takes in the raw data (e.g., images, text, numbers).
    - The number of neurons = the number of features in the dataset.
2. Hidden Layers
    - Located between the input and output layers.
    - They perform computations and extract patterns.
    - Each layer has multiple hidden units (neurons).
    - The number of hidden units determines the model's capacity.
3. Output Layer
    - Produces the final prediction.
    - The number of neurons depends on the task:
        - Regression: 1 neuron (continuous output).
        - Binary classification: 1 neuron with sigmoid activation.
        - Multi-class classification: Neurons = number of classes.

3. Hidden Units (Neurons)

- A hidden unit (or neuron) takes input from the previous layer and applies:

Choosing the Number of Hidden Units

- Too few: Model underfits (fails to learn patterns).
- Too many: Model overfits (memorizes training data but fails on new data).
- Common methods to choose hidden units:
    - Start small and increase until performance stops improving.
    - Use cross-validation to test different architectures.

4. **Neural Network Architecture Design**

The architecture includes:

1. Number of layers
2. Number of neurons per layer
3. Activation functions
4. Connection types (e.g., fully connected, convolutional)

Guidelines for Architecture Design

1. Start with a simple network and increase complexity as needed.
2. Use fewer layers for simple problems (e.g., 1–2 hidden layers).
3. Deep networks (many layers) are better for complex tasks like image and speech recognition.
4. Use dropout & batch normalization to avoid overfitting.
5. Experiment with different architectures and compare performance.

Common Neural Network Architectures

Architecture	Usage
Fully Connected (Feedforward NN)	Simple tasks like regression & basic classification.
Convolutional NN (CNN)	Image recognition, object detection.
Recurrent NN (RNN, LSTM, GRU)	Time-series, speech recognition, NLP.
Transformer Networks	Advanced NLP (e.g., ChatGPT, BERT).

- Hidden units process data between input & output.
- More hidden units = better learning, but too many cause overfitting.
- Start small & tune the architecture based on cross-validation.
- Different architectures work better for different tasks (CNNs for images, RNNs for sequences).

What is Maximum Likelihood Estimation

Maximum Likelihood Estimation (MLE) is a statistical method for estimating parameters of a probability distribution.
It finds the values of parameters that maximize the likelihood of observing the given data.

Likelihood Function

Given a dataset $X=\{x1,x2,...,xn\}$ the likelihood function is:

$L(\theta)=P(X|\theta)$
where:

θ is the parameter we want to estimate.
P(X|θ) is the probability of observing the data X given θ.
For independent observations, the likelihood function is:

$L(\theta)=i=1\Pi n\ P(xi|\theta)$

Log-Likelihood Function

Instead of working with the product L(θ) (which can be very small), we take the natural logarithm:

$\text{Log } L(\theta)=i=1\Sigma n \log P\ (xi|\theta)$

This simplifies calculations and avoids numerical issues.

Steps for MLE

1. Define the likelihood function: Write down L(θ) based on the probability distribution.
2. Take the log-likelihood: Compute logL(θ) to simplify the equation.
3. Differentiate w.r.t. θ\thetaθ: Compute d/dθ logL(θ)
4. Solve for θ\thetaθ: Set the derivative to zero and solve for θ, giving the MLE estimate.

Numerical Example of MLE (Maximum Likelihood Estimation)

Problem: Estimating Probability of Heads in a Coin Toss
A coin is tossed 10 times, and we observe 7 heads and 3 tails. We want to estimate the probability p of getting heads using MLE.

Step 1: Define the Likelihood Function
Since each coin toss follows a Bernoulli distribution, the probability of getting 7 heads and 3 tails (given p) is:
$L(p)=P(X=7|p)=p7\cdot(1-p)3$
where:

$p7$ represents the probability of getting 7 heads.
$(1-p)3$ represents the probability of getting 3 tails.

Step 2: Take the Log-Likelihood

Instead of working with L(p)L(p)L(p), we take the natural logarithm:
$\text{Log } L(p)=7\log p+3\log(1-p)$

Step 3: Differentiate and Solve for p

To find the maximum likelihood estimate, we differentiate logL(p) with respect to p:

$d/dp \log L(p) = 7/p - 3/1-p$
Setting this to zero:
$7/p = 3/1-p$
Solving for p:
$7(1-p) = 3p$
$7-7p = 3p$
$7 = 10p$
$p = 7/10 = 0.7$

Step 4: Interpretation

The MLE estimate for ppp is 0.7, meaning our best guess for the probability of heads is 70%.

Applications of MLE

Coin Toss (Bernoulli/Binomial Distribution) Estimating Mean & Variance (Normal Distribution) Poisson Distribution (Event Counts) Logistic Regression (Classification Models)

Advantages of MLE

Efficient: Gives the best possible estimate for large data. Flexible: Works for different probability distributions. Asymptotically Unbiased: The estimate approaches the true parameter as sample size increases.

Disadvantages of MLE

Sensitive to Small Data: Estimates may be inaccurate for small datasets. Computationally Complex: For complex models, solving MLE can be hard.

Back propagation Method :

- Back propagation is a supervised learning algorithm used for training neural networks.
- It is a method used to train artificial neural networks. Its goal is to reduce the difference between the model's predicted output and the actual output by adjusting the weights and biases in the network.
- Backpropagation is crucial for deep learning.
- Helps neural networks learn from data efficiently.
- Requires proper tuning for optimal performance.

- It minimizes the error by adjusting weights using the gradient descent method.
- Introduced by Rumelhart, Hinton, and Williams in 1986.

Why Backpropagation?

Automates learning in neural networks.
Efficiently updates weights to reduce errors.
Helps deep learning models train complex functions.

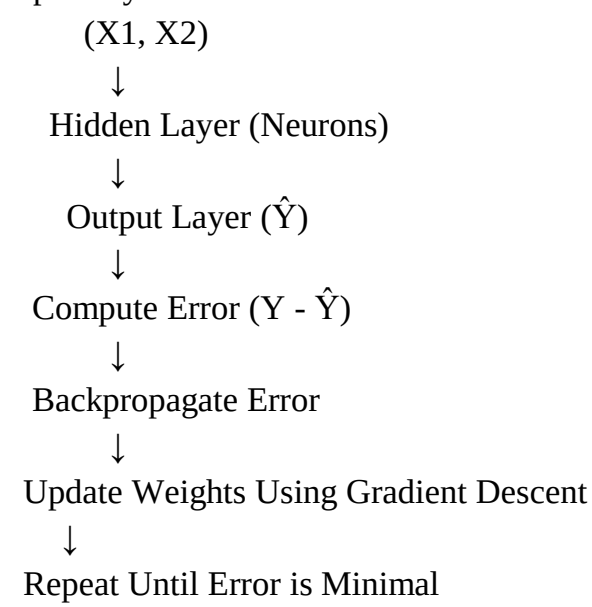
How Back propagation Works

1. Forwrd Propagation: Compute output using current weights.
2. Compute Error: Compare predicted vs. actual output.
3. Backward Propagation: Calculate gradient of error with respect to weights.
4. Weight Update: Adjust weights to minimize the error.

Mathematical Explanation

Error Function: $E = (1/2) \Sigma (y\_actual - y\_predicted)^2$
Gradient Descent Update Rule:
$W = W - \eta * (\partial E / \partial W)$
Where:
- W = weights
- η = learning rate
- $\partial E/\partial W$ = gradient of error

Diagrame



Advantages:

Automates learning in neural networks
Works well for complex problems
Scalable for deep learning

Disadvantages:

Computationally expensive
Requires careful tuning of learning rate

Applications :

Image Recognition (e.g., CNNs)
Natural Language Processing
Self-driving Cars
Medical Diagnosis

- End -