Experiment Number: 01

Experiment Name: Write a program to evaluate AND function with bipolar inputs and targets and also show the convergence curves and the decision boundary lines.

Objectives: The primary objective is to implement a Perceptron-based neural network for learning the AND logic function with bipolar inputs and targets, and to visualize:

- The convergence of error during training.
- The decision boundary separating the classes.

Theory: The Perceptron is the simplest type of artificial neural network and is used for binary classification problems. It is composed of a single layer of weights and an activation function (typically the sign or step function).

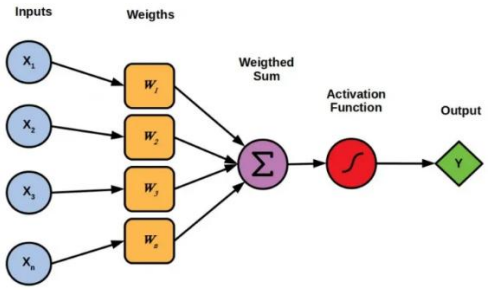


Figure 01: Simple overview of a perceptron network

Bipolar Logic: In bipolar representation:

- Logical True (1) is represented as $+1$
- Logical False (0) is represented as -1

Table 01: AND Function (Bipolar Truth Table)

Input X1	Input X2	Target (T)
-1	-1	-1
-1	+1	-1
+1	-1	-1
+1	+1	+1

Perceptron Learning Rule: The weights are updated iteratively using the following formula:

$$w_{new} = w_{old} + \eta(t - y)x$$

Where:

- w_{new} = new weight vector
- η = learning rate
- t = target output
- y = predicted output
- x = input vector

Convergence Curve: The convergence curve in the context of machine learning, particularly in training a perceptron, represents how the error (or loss) of the model changes over time (i.e., over the training epochs). It visually shows the process of learning and how quickly the model is converging to the correct solution.

Decision Boundary: A decision boundary is a concept in machine learning that represents the region of a problem space where the output label of a classifier changes. It is a line, curve, or surface (in higher dimensions) that separates different classes in a classification problem. The decision boundary for the perceptron is a line in the input space that separates classes where the output is +1 or -1.

Code with explanation:

1. Import Libraries

```
import numpy as np
import matplotlib.pyplot as plt
```

Explation:

- numpy is used for matrix and array operations.
- matplotlib.pyplot is used for plotting graphs (convergence curve and decision boundary).

2. Define Input and Target Data

```
X = np.array([[-1, -1],
              [-1,  1],
              [ 1, -1],
              [ 1,  1]])
T = np.array([-1, -1, -1, 1])  # Targets
```

Explanation:

- X: The 4 possible input combinations for a 2-input AND gate using bipolar format.
- T: Corresponding target outputs (truth table for bipolar AND).

3. Add Bias Term

```
X_bias = np.hstack((np.ones((X.shape[0], 1)), X))
```

Explanation:

- Adds a column of 1s to the input matrix for the bias.
- X_bias becomes a 4×3 matrix: [1, x1, x2],…….
- Helps include the threshold (bias) in the weight update rule.

4. Set Training Parameters

```
epochs = 20
lr = 0.1  # Learning rate
weights = np.zeros(X_bias.shape[1])  # [bias, w1, w2]
```

Explanation:

- epochs: Maximum number of times the algorithm will iterate over the dataset.
- lr: Learning rate that controls the update size.
- weights: Initialize all weights (including bias) to 0.

5. Define Activation Function

```
def bipolar_step(x):
    return 1 if x >= 0 else -1
```

Explanation:

- A bipolar step function: outputs $+1$ if input ≥ 0, else -1.
- This is used as the neuron's activation function.

6. Train the Perceptron

```
errors = []
for epoch in range(epochs):
    total_error = 0
    for i in range(len(X_bias)):
        net_input = np.dot(weights, X_bias[i])
        output = bipolar_step(net_input)
        error = T[i] - output
        weights += lr * error * X_bias[i]
        total_error += abs(error)
    errors.append(total_error)
    if total_error == 0:
        break
```

Explanation:

- Loop over each epoch:
    - net_input = dot(weights, X_bias[i]): compute weighted sum (including bias).
    - output = bipolar_step(net_input): apply activation function.
    - error = target − output: calculate prediction error.
    - weights += lr * error * X_bias[i]: update weights using Perceptron learning rule.
    - Accumulate total_error for the epoch.
- Store total error to plot convergence.
- Break if total error becomes zero (i.e., convergence reached).

7. Print Final Weights

```
print("Final Weights:", weights)
```

Explanation:

- Shows the final weight vector [bias,w1,w2][bias, w1, w2][bias,w1,w2].

8. Plot Convergence Curve

```
plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.plot(errors, marker='o')
plt.title("Convergence Curve")
plt.xlabel("Epochs")
plt.ylabel("Total Error")
plt.grid(True)
```

Explanation:

- Plots the total error per epoch to show how quickly the Perceptron learns.

9. Plot Decision Boundary

```
plt.subplot(1, 2, 2)
for i in range(len(X)):
    if T[i] == 1:
```

```
        plt.plot(X[i][0], X[i][1], 'bo')  # Blue for class +1
    else:
        plt.plot(X[i][0], X[i][1], 'rx')  # Red for class -1
```

Explanation:

Plots input points:

- Blue circles for target = +1
- Red crosses for target = −1

10. Draw Decision Boundary

```
x_vals = np.linspace(-2, 2, 100)
y_vals = -(weights[0] + weights[1] * x_vals) / weights[2]
plt.plot(x_vals, y_vals, 'k--', label='Decision Boundary')
```

Explanation:

- The decision boundary is a straight line
- Plotted as a black dashed line to show the separation between classes.

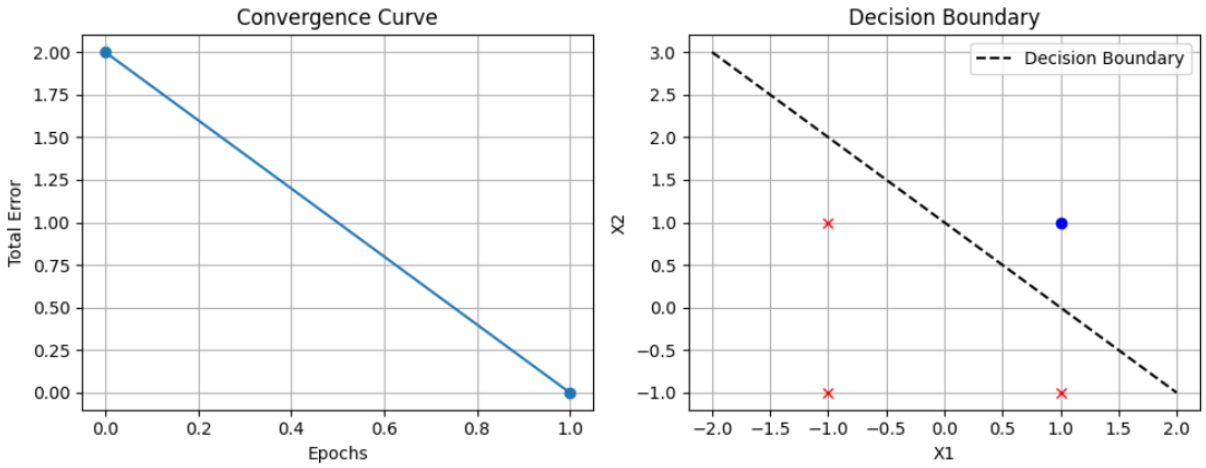
11. Adds labels, grid, and displays both plots (convergence + decision boundary).

```
plt.title("Decision Boundary")
plt.xlabel("X1")
plt.ylabel("X2")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

Explanation:

Adds labels, grid, and displays both plots (convergence + decision boundary).

Results: Final Weights [-0.2 0.2 0.2]



Experiment Number: 02

Experiment Name: Write a program to evaluate X-OR function and also show the convergence and the decision boundary.

Objectives: The primary objective is to design and implement a neural network capable of learning the XOR logic function, which is a non-linearly separable problem.

Theory: Artificial Neural Networks (ANNs) are computational models inspired by the human brain, capable of learning complex input-output relationships from data. One of the fundamental challenges in early neural network research was the inability of simple models, like the single-layer perceptron, to solve non-linearly separable problems—most notably the XOR (exclusive OR) function. The XOR problem, despite its simplicity, played a critical role in advancing neural network research by highlighting the necessity of hidden layers and non-linear activation functions.
This experiment aims to explore how a Multilayer Perceptron (MLP), trained using the backpropagation algorithm, can learn to accurately classify XOR inputs. By implementing a network with a hidden layer and training it using stochastic gradient descent (SGD), we demonstrate the ability of ANNs to approximate non-linear decision boundaries and solve logical functions that are otherwise not solvable using linear models.

Table 01: Truth table of X-OR

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Feedforward Neural Network (Multilayer Perceptron - MLP): A multilayer perceptron consists of:

- Input layer
- Hidden layer with nonlinear activation functions (e.g., sigmoid)
- Output layer

The hidden layer enables the network to learn complex patterns and nonlinear decision boundaries.

Activation Function: The sigmoid function is used in hidden and output layers. It maps input values to a range between 0 and 1, allowing the network to learn probabilities.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Backpropagation Algorithm: Backpropagation is a supervised learning algorithm used for training neural networks. Steps involved:

- **Forward Pass:** Compute the output of the network.
- **Error Calculation:** Measure the difference between predicted and target values.

- **Backward Pass:** Propagate the error backward through the network using the **chain rule** to compute gradients.
- **Weight Update:** Adjust weights using **Stochastic Gradient Descent (SGD)**-

$$w = w + \eta.\delta.x$$

Where, η is the learning rate, δ is the error signal, and x is the input.

Convergence Curve: This curve Plots the **training error** (number of misclassifications or mean squared error) vs. **epochs** and shows how quickly and effectively the network is learning.

Decision Boundary: A contour plot is used to **visualize the regions** in the input space classified as 0 or 1 by the trained model. This helps demonstrate the network's ability to model non-linear separations, such as the XOR problem.

Code with explanation:

1. Importing Libraries

```
import numpy as np
import matplotlib.pyplot as plt
```

Explanation:

- numpy is used for numerical operations like matrix multiplication and array manipulation.
- matplotlib.pyplot is used for plotting graphs (convergence curve, decision boundary).

2. Activation Functions

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)
```

Explanation:

- sigmoid(x) is the **sigmoid activation function**, squashes input to a value between 0 and 1.
- sigmoid_derivative(x) is the derivative used for **backpropagation** to calculate gradients

3. XOR Input and Target Data

```
inputs = np.array([[0, 0],
                   [0, 1],
                   [1, 0],
                   [1, 1]])

targets = np.array([0, 1, 1, 0])
```

Explanation:

- These are the input-output pairs for the **XOR logical function**.
- The goal is for the network to learn this mapping.

4. Network Architecture Parameters

```
input_layer_size = 2
hidden_layer_size = 2
output_layer_size = 1
learning_rate = 0.1
max_epochs = 10000
```

Explanation:

- Network has 2 inputs → 2 hidden neurons → 1 output.
- Learning rate controls how big each weight update is.
- max_epochs limits training iterations.

5. Weight and Bias Initialization

```
np.random.seed(42)
weights_input_hidden = np.random.randn(input_layer_size, hidden_layer_size)
bias_hidden = np.random.randn(hidden_layer_size)

weights_hidden_output = np.random.randn(hidden_layer_size, output_layer_size)
bias_output = np.random.randn(output_layer_size)
```

Explanation:

- Randomly initialize weights and biases for hidden and output layers.
- np.random.seed(42) ensures reproducible results.

6. Training the Neural Network

```
convergence_curve = []

for epoch in range(max_epochs):
   misclassified = 0
   for i in range(len(inputs)):
      # Forward pass
      hidden_layer_input = np.dot(inputs[i], weights_input_hidden) + bias_hidden
      hidden_layer_output = sigmoid(hidden_layer_input)

      output_layer_input = np.dot(hidden_layer_output, weights_hidden_output) +
bias_output
      predicted_output = sigmoid(output_layer_input)

      # Backpropagation
      error = targets[i] - predicted_output
      if targets[i] != np.round(predicted_output):
         misclassified += 1

      output_delta = error * sigmoid_derivative(predicted_output)
      hidden_delta = output_delta.dot(weights_hidden_output.T) *
sigmoid_derivative(hidden_layer_output)

      # Update weights and biases
      weights_hidden_output += hidden_layer_output[:, np.newaxis] * output_delta *
learning_rate
      bias_output += output_delta * learning_rate

      weights_input_hidden += inputs[i][:, np.newaxis] * hidden_delta * learning_rate
```

```
        bias_hidden += hidden_delta * learning_rate

    err = 1 - (len(inputs) - misclassified) / len(inputs)
    convergence_curve.append(err)

    if misclassified == 0:
        print("Converged in {} epochs.".format(epoch + 1))
        break
```

Explanation:

- For each epoch (iteration over all data):
  - Perform **forward pass** to get predictions.
  - Use **backpropagation** to calculate error gradients.
  - **Update weights** using the delta rule.
- Tracks misclassifications and error for plotting.
- Stops early if perfect classification is achieved.

7. Plotting the Convergence Curve

```
plt.figure(figsize=(8, 4))
plt.plot(range(1, len(convergence_curve) + 1), convergence_curve)
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.title('Convergence Curve')
plt.grid()
plt.show()
```

Explanation:

- Plots how error decreases over training epochs, helping to visualize learning progress.

8. Prediction Function for Plotting

```
x1 = np.linspace(-0.5, 1.5, 200)
x2 = np.linspace(-0.5, 1.5, 200)
X1, X2 = np.meshgrid(x1, x2)

def predict(x1, x2):
    hidden_input = np.dot(np.c_[x1.ravel(), x2.ravel()], weights_input_hidden) + bias_hidden
    hidden_output = sigmoid(hidden_input)
    output_input = np.dot(hidden_output, weights_hidden_output) + bias_output
    output = sigmoid(output_input)
    return output.reshape(x1.shape)

Z = predict(X1, X2)
```

Explanation:

- Creates a grid of input points to evaluate the trained model.
- Predicts output for each point using the current network.

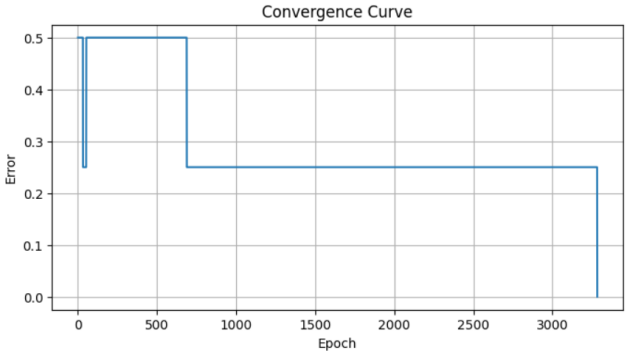
9. Plotting the Decision Boundary

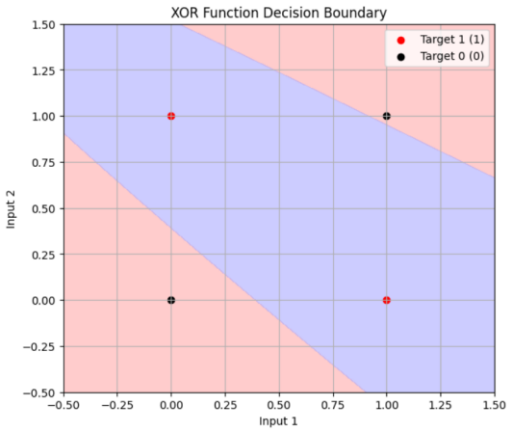
```
plt.figure(figsize=(8, 6))
plt.contourf(X1, X2, Z, levels=[0, 0.5, 1], colors=['#FFAAAA', '#AAAAFF'], alpha=0.6)

plt.scatter(inputs[targets == 1][:, 0], inputs[targets == 1][:, 1], label='Target 1 (1)', color='red')
plt.scatter(inputs[targets == 0][:, 0], inputs[targets == 0][:, 1], label='Target 0 (0)', color='black')

plt.xlabel('Input 1')
plt.ylabel('Input 2')
plt.title('XOR Function Decision Boundary')
plt.legend()
plt.grid()
plt.show()
```

Explanation:

- Uses contourf to fill regions where the output is above or below 0.5.
- Shows the learned decision boundary for the XOR function.
- Data points are overlaid for visual comparison.

Results:





Experiment Number: 03

Experiment Name: Implement the SGD Method using Delta learning rule for following input-target sets.

- X_{input}= [0 0 1; 0 1 1;1 0 1; 1 1 1]
- D_{target} = [0; 0; 1; 1]

Objectives: The primary objective of this experiment is to implement a single-layer perceptron model and apply the Stochastic Gradient Descent (SGD) method for weight updates.

Theory: To implement the Stochastic Gradient Descent (SGD) method using the Delta learning rule, we need to build a simple perceptron model that updates its weights iteratively based on the error between the predicted output and the target output.

Perceptron Model: A perceptron is the simplest type of artificial neural network, used for binary classification. It consists of a weighted sum of inputs, followed by an activation function (typically a step function or sigmoid for more flexibility). Mathematically,

$$y = f(w^T x + b)$$

Where:

- x = Input vector
- w = Weight vector
- b = Bias
- f = Activation function (e.g., sigmoid or threshold)

Delta Learning Rule: The delta rule is a gradient descent-based learning algorithm that adjusts weights to minimize the mean squared error between the actual output and target output.

$$w_{new} = w_{old} + \eta(d - y)x$$

Where:

- η = Learning rate
- d = Desired (target) output
- y = Predicted output

This update is done for each sample individually in SGD.

Stochastic Gradient Descent (SGD): SGD updates weights incrementally after processing each training example (as opposed to batch gradient descent, which updates weights after a full pass through the dataset). This makes it more responsive and often faster to converge.

Code with explanation:

1. Importing Required Libraries

```
import numpy as np
import matplotlib.pyplot as plt
```

Explanation:

- numpy is used for numerical computations (matrix operations, arrays, etc.).
- matplotlib.pyplot is used to plot the **convergence curve** and the **decision boundary**.

2. Input and Target Data

```
X = np.array([[0, 0, 1],
              [0, 1, 1],
              [1, 0, 1],
              [1, 1, 1]])

D = np.array([[0],
              [0],
              [1],
              [1]])
```

Explanation:

- X is the **input matrix**. Each row is a sample with 3 features: x1, x2, and the **bias** term set to 1.
- D is the **desired output** or **target vector**, representing the logical OR function.

3. Weight Initialization

```
# W = np.random.randn(3, 1)
W = np.array([[1.9189432],
              [0.614925],
              [-0.956587]])
```

Explanation:

- W is a column vector of weights: one for each input and one for the bias.
- We can randomly initialize weights (np.random.randn) or set them manually (as done here for reproducibility).

4. Learning Parameters

```
eta = 0.1
epochs = 20
```

Explanation:

- eta is the **learning rate** that controls how much weights change during each update.
- epochs is the number of times the model goes through the entire dataset.

5. Training with SGD + Delta Rule

```
mse_list = []

for epoch in range(epochs):
    epoch_error = 0
    for i in range(len(X)):
        x = X[i].reshape(3, 1)
```

```
        d = D[i]
        y = np.dot(W.T, x)
        e = d - y
        W += eta * e * x
        epoch_error += e**2
    mse = epoch_error / len(X)
    mse_list.append(mse.item())
```

Explanation:

- mse_list tracks mean squared error per epoch.
- For each epoch:
  - Loop through each input sample x and target d.
  - Compute the model output: $y = W^t x$.
  - Calculate error: e = d - y.
  - **Delta rule update**: $W \leftarrow W + \eta \cdot e \cdot x$.
  - Accumulate squared error for MSE calculation.
- At the end of each epoch, compute MSE and store it.

6. Plotting the Convergence Curve

```
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(range(1, epochs + 1), mse_list, marker='o')
plt.title('Convergence of MSE')
plt.xlabel('Epoch')
plt.ylabel('Mean Squared Error')
plt.grid(True)
```

Explanation:

- Creates a figure with 2 subplots.
- The first subplot visualizes the **MSE over epochs**, showing how the model is learning and reducing error over time.

7. Decision Boundary Visualization

```
plt.subplot(1, 2, 2)
X_no_bias = X[:, :2]
class_0 = X_no_bias[D[:, 0] == 0]
class_1 = X_no_bias[D[:, 0] == 1]

plt.scatter(class_0[:, 0], class_0[:, 1], color='red', label='Class 0')
plt.scatter(class_1[:, 0], class_1[:, 1], color='blue', label='Class 1')
```

Explanation:

- Second subplot: separates the input features (excluding bias) into class 0 and class 1 for plotting.
- Plots the inputs in 2D space with different colors for each class.

8. Drawing the Decision Boundary

```
w1, w2, b = W[0, 0], W[1, 0], W[2, 0]
x_vals = np.array([0, 1])
```

```
y_vals = -(w1 * x_vals + b) / w2
plt.plot(x_vals, y_vals, 'k--', label='Decision Boundary')
```

Explanation:

- From the equation, $w_1x_1 + w_2x_2 + b = 0$, solve for x_2 to plot the decision boundary:

$$x_2 = -\frac{w_1x_1 + b}{w_2}$$

- Plots the linear decision boundary as a dashed black line.

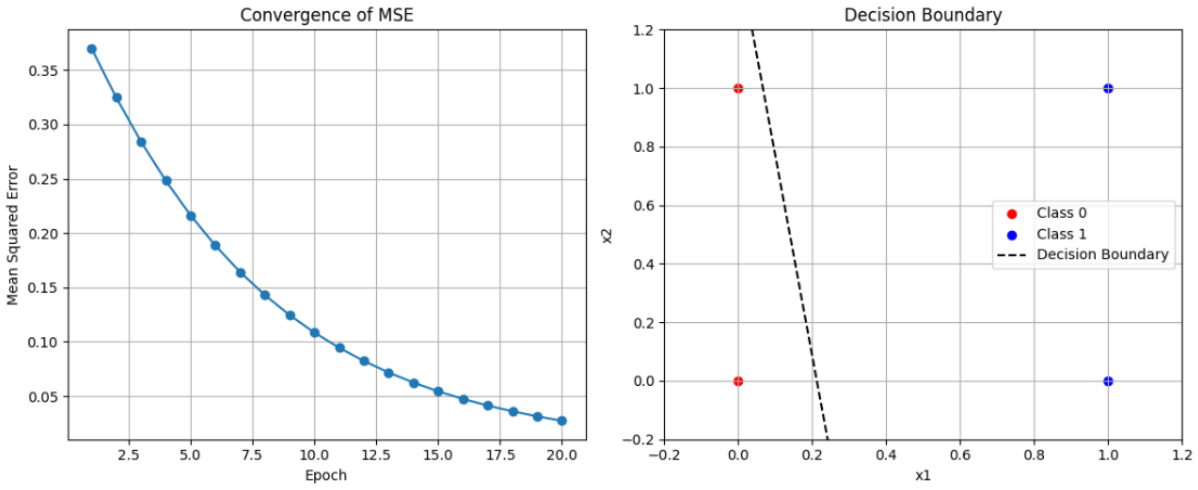
9. Final Plot Settings

```
plt.xlim(-0.2, 1.2)
plt.ylim(-0.2, 1.2)
plt.title('Decision Boundary')
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```

Explanation:

- Sets axis limits and labels.
- Adds a title, legend, and grid for clarity.
- plt.tight_layout() adjusts spacing between subplots.
- plt.show() displays the final visualization.

Results:



Experiment Number: 04

Experiment Name: Write a program to evaluate a simple feedforward neural network for classifying handwritten digits using the MNIST dataset.

Objectives: The objective of this experiment is to implement and evaluate a simple feedforward neural network using the MNIST dataset of handwritten digits. The goal is to train a neural network to accurately classify digits from 0 to 9 based on pixel values of input images.

Theory: Feedforward Neural Network (FNN) is a type of artificial neural network in which information flows in a single direction—from the input layer through hidden layers to the output layer—without loops or feedback. It is mainly used for pattern recognition tasks like image and speech classification. For example in a credit scoring system banks use an FNN which analyze users' financial profiles—such as income, credit history and spending habits—to determine their creditworthiness. Each piece of information flows through the network's layers where various calculations are made to produce a final score.

Explanation of FNN architecture: Feedforward Neural Networks have a structured layered design where data flows sequentially through each layer.

1. **Input Layer**: The input layer consists of neurons that receive the input data. Each neuron in the input layer represents a feature of the input data.

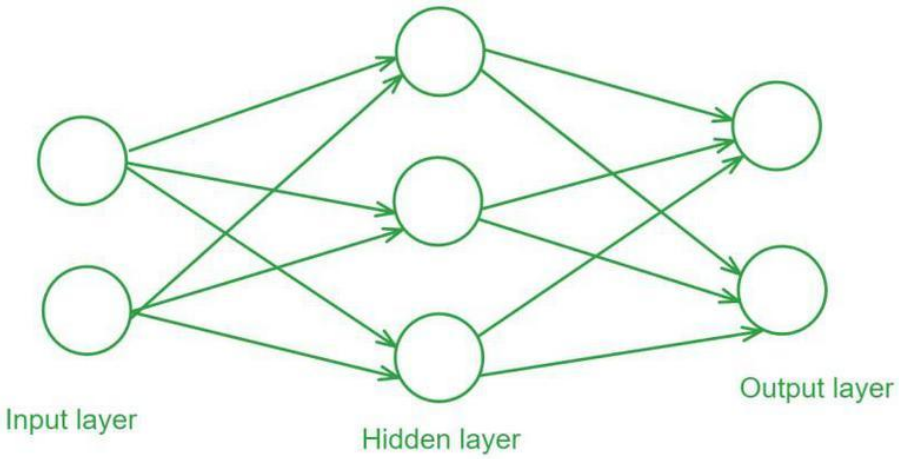


Figure 01: FNN Architecture

2. **Hidden Layers**: One or more hidden layers are placed between the input and output layers. These layers are responsible for learning the complex patterns in the data. Each neuron in a hidden layer applies a weighted sum of inputs followed by a non-linear activation function.

3. **Output Layer**: The output layer provides the final output of the network. The number of neurons in this layer corresponds to the number of classes in a classification problem or the number of outputs in a regression problem.

Each connection between neurons in these layers has an associated weight that is adjusted during the training process to minimize the error in predictions.

Training a Feedforward Neural Network: Training a Feedforward Neural Network involves adjusting the weights of the neurons to minimize the error between the predicted output and the actual output. This process is typically performed using backpropagation and gradient descent.

1. **Forward Propagation**: During forward propagation the input data passes through the network and the output is calculated.
2. **Loss Calculation**: The loss (or error) is calculated using a loss function such as Mean Squared Error (MSE) for regression tasks or Cross-Entropy Loss for classification tasks.
3. **Backpropagation**: In backpropagation the error is propagated back through the network to update the weights. The gradient of the loss function with respect to each weight is calculated and the weights are adjusted using gradient descent.

In this neural network model, there is used two key terms and they are:

- **Activation Functions:** Activation functions introduce non-linearity into the network enabling it to learn and model complex data patterns. Common activation functions include:
  - Sigmoid function: $\sigma(x) = \frac{1}{1+ e^{-x}}$
  - Tanh function: $tanx = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$
  - ReLU function: $ReLU(x) = \max(0, x)$
- **Gradient descent:** Gradient Descent is an optimization algorithm used to minimize the loss function by iteratively updating the weights in the direction of the negative gradient. Common variants of gradient descent include:
  - Batch Gradient Descent: Updates weights after computing the gradient over the entire dataset.
  - Stochastic Gradient Descent (SGD): Updates weights for each training example individually.
  - Mini-batch Gradient Descent: It Updates weights after computing the gradient over a small batch of training examples.

Code with explanation: To understand FNN more efficiently, we can consider the following example, which walks through a real-world hand written digit classification task using MNIST dataset.

- First of all, the first block imports the necessary Python libraries. numpy is used for numerical operations, especially when working with arrays. matplotlib.pyplot is a plotting library used to visualize images and data. From sklearn.datasets, the fetch_openml function is used to load the MNIST dataset, which contains images of handwritten digits. The train_test_split function from sklearn.model_selection

helps in dividing the dataset into training and testing sets. Although StandardScaler is imported to scale features, it is not used in this code. MLPClassifier from sklearn.neural_network is used to create a multi-layer perceptron (neural network). Lastly, accuracy_score from sklearn.metrics is used to evaluate how well the model performs on the test data.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score
```

- In the second block, the MNIST dataset is loaded using fetch_openml. This dataset contains 70,000 grayscale images of handwritten digits (0 through 9), each of size 28×28 pixels. These images are flattened into 784-dimensional vectors, stored in X. The corresponding labels (the digits) are stored in y. Since the original labels are in string format, they are converted to integers using .astype(int) for compatibility with the classifier.

```
mnist = fetch_openml('mnist_784', version=1)
X, y = mnist.data, mnist.target.astype(int)
```

- The third block deals with preprocessing. Since the pixel values range from 0 to 255, they are normalized by dividing each value by 255.0. This scales the input features to a range of 0 to 1, which helps the neural network train more effectively. The dataset is then split into training and testing sets using train_test_split, with 80% of the data used for training and 20% for testing. The random_state=42 ensures that the split is reproducible.

```
X = X / 255.0
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

- In the fourth block, an MLP (Multi-Layer Perceptron) classifier is created. This neural network has one hidden layer with 128 neurons and uses the ReLU (Rectified Linear Unit) activation function, which helps the model learn non-linear patterns. The adam optimizer is chosen for weight updates during training, which is effective for large datasets. The training is limited to a maximum of 10 iterations (max_iter=10) to keep the process fast, though more iterations would likely improve performance. The model is trained using the fit() method on the training data.

```
model=MLPClassifier(hidden_layer_sizes=(128,),activation='relu', solver='adam', max_iter=10, random_state=42)
model.fit(X_train, y_train)
```

- The fifth block evaluates the trained model. It uses the predict() method to generate predictions on the test set. These predictions are compared to the actual

labels using accuracy_score, which computes the ratio of correct predictions to the total number of samples. The test accuracy is printed, giving a quick insight into how well the model is performing.

```
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f'\nTest accuracy: {accuracy}')
```

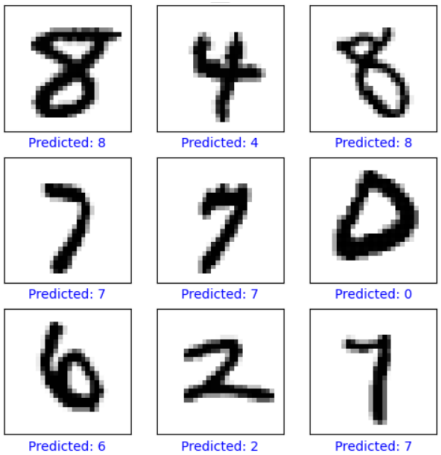
- The sixth block defines a function named plot_image that is responsible for visualizing individual test images. It removes the axis ticks and grid lines for a cleaner look. The image is reshaped from a flat 784-length vector back into its original 28×28 shape and displayed in grayscale using a binary colormap. The function also labels the image with the predicted digit, which is shown in blue text.

```
def plot_image(i, true_label, img):
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])
  plt.imshow(img.reshape(28, 28), cmap=plt.cm.binary)
  plt.xlabel(f"Predicted: {y_pred[i]}", color='blue')
```

- The final block displays a grid of 9 sample predictions. It defines the number of rows and columns (3x3) to be shown, creating a figure with appropriate size using plt.figure(). A loop iterates through the first 9 test images, and for each one, a subplot is created using plt.subplot(). The plot_image() function is called for each subplot to display the image and its predicted label. Finally, plt.show() renders the full set of images.

```
num_rows, num_cols = 3, 3
num_images = num_rows * num_cols
plt.figure(figsize=(2*num_cols, 2*num_rows))
for i in range(num_images):
   plt.subplot(num_rows, num_cols, i+1)
   plot_image(i, y_test.iloc[i], X_test.iloc[i])
plt.show()
```

Results: Test accuracy: 97.17%. The model shows the prediction in following:



Experiment Number: 05

Experiment Name: Write a program to evaluate a Convolutional Neural Network (CNN) for image classification.

Objectives: The objective of this experiment is to design, train, and evaluate a Convolutional Neural Network (CNN) model for image classification tasks. The goal is to enable the model to accurately learn and recognize patterns and features from image data, thereby classifying images into their respective categories with high accuracy.

Theory: Image classification using CNN involves the extraction of features from the image to observe some patterns in the dataset. In this work, we use CIFAR-10 dataset. The CIFAR-10 dataset is a widely used benchmark in Machine Learning and Computer Vision, consisting of 60,000 color images with a resolution of 32x32 pixels. These images are categorized into 10 distinct classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck. The dataset is split into 50,000 training images and 10,000 test images, making it ideal for training and evaluating Convolutional Neural Networks (CNNs). In this work, we train a CNN model on the CIFAR-10 dataset to perform accurate image classification across these ten categories.

A Convolutional Neural Network (CNN) is a type of deep learning algorithm specifically designed for processing and analyzing visual data, such as images. It mimics the way the human brain processes visual information by learning to recognize patterns, edges, textures, and objects in images. CNNs are widely used in tasks like image classification, object detection, and image segmentation. A CNN (Convolutional Neural Network) has two main parts:

1. **Feature Extraction** – This part uses filters to scan the image and find important features like edges, colors, or shapes.
2. **Classification** – After the features are found, this part (a fully connected layer) uses them to decide what the image is showing, like identifying if it's a cat or a dog.

Explanation of CNN architecture: CNNs consist of several layers that transform input images into meaningful representations for classification. The architecture includes:

1. **Input Layer**: Accepts raw image data, such as the CIFAR-10 dataset images (32x32 pixels with RGB channels). Each image is represented as a 3D matrix (height, width, depth).
2. **Convolutional Layer**: Applies filters (kernels) to the input image to detect features like edges, textures, or shapes. The convolution operation involves sliding a kernel across the image and computing the dot product between the kernel and the image patch. This layer produces feature maps that highlight detected features.

$$S(i,j) = \sum_m \sum_n X(i+m, j+n).K(m,n)$$

Where,
X = input image
K = Kernel(filter)
S (i, j) = Output feature map at position (i, j)
m, n = indices of the kernel window
This layer produces feature maps that highlight detected features.

3. **Activation Function (ReLU):** The **ReLU (Rectified Linear Unit)** introduces non-linearity by setting all negative values to zero. It helps the network learn complex patterns without affecting the positive values.

$$f(x) = \max(0, x)$$

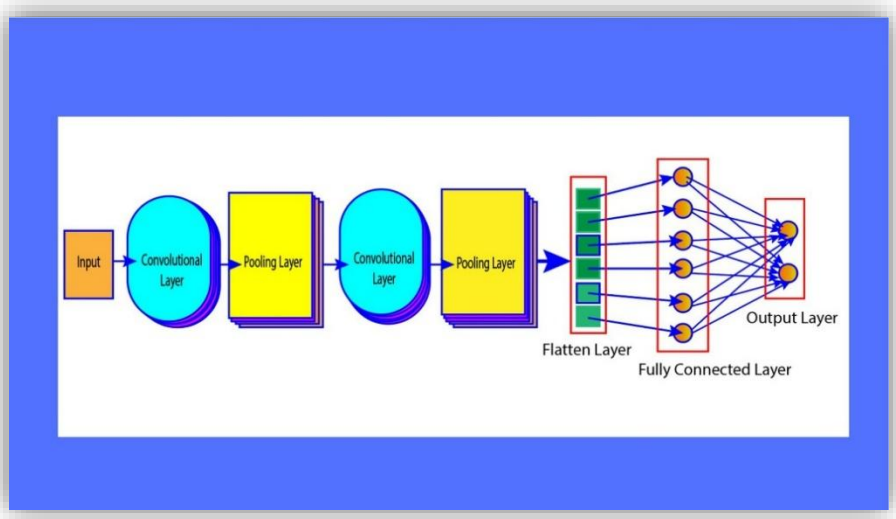


Figure 01: CNN Architecture

4. **Pooling Layer**: The main goal of this layer is to reduce the convoluted size of the feature map and to reduce Computational costs. This layer preserve important information and the common pooling methods include max pooling:

$$P(i, j) = \max_{m,n} S(i + m, j + n)$$

Where,
P (i, j) = Output after pooling
S = input feature map
m, n = window size

5. **Flatten Layer**: This layer converts multi-dimensional feature maps into a one-dimensional vector for input into fully connected layers.
6. **Fully Connected Layers:** This layer performs high-level reasoning using learned features. Here, each neuron is connected to every neuron in the previous layer. Outputs probabilities for each class using a softmax function:

$$Softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_i}}$$

7. **Output Layer:** This layer provides classification results, such as identifying one of 10 classes in CIFAR-10 (e.g., airplane, car, bird).

Code with explanation:

Block 01: Importing Libraries: These lines import necessary Python libraries. tensorflow and its submodules like datasets, layers, and models are used for deep learning tasks, especially model building and training. matplotlib.pyplot is used for data visualization (such as displaying images or training performance graphs). numpy is used for numerical computations and data manipulation.

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
import numpy as np
```

Block 02: Loading the CIFAR-10 Dataset: This line loads the CIFAR-10 dataset, which is a collection of 60,000 32x32 color images in 10 different classes. The data is split into 50,000 training and 10,000 test images, and stored in X_train, y_train, X_test, and y_test.

```
(X_train, y_train), (X_test, y_test) = datasets.cifar10.load_data()
```

Block 03: Reshaping Label Arrays: The labels are initially in a 2D shape like (50000, 1). These lines reshape the labels into 1D arrays (like (50000,)) to make them compatible with Keras model training functions.

```
y_train = y_train.reshape(-1,)
y_test = y_test.reshape(-1,)
```

Block 04: Class Names for Visualization: This list stores human-readable class names corresponding to label indices (0 to 9). It's used to display the predicted and actual class names instead of raw numbers.

```
classes = ["airplane", "automobile", "bird", "cat", "deer", "dog", "frog", "horse", "ship", "truck"]
```

Block 05: Function to Plot a Sample Image: This function displays one sample image from the dataset at a specific index. It shows the image and labels it using the corresponding class name from the classes list.

```
def plot_sample(X, y, index):
    plt.figure(figsize=(2.5,2.5))
    plt.imshow(X[index])
    plt.xlabel(classes[y[index]])
    plt.axis("off")
    plt.show()
```

Block 06: Displaying Sample Images: These lines call the plot_sample function to visually display two training images with their class labels. This helps confirm that the dataset has been loaded correctly.

```
plot_sample(X_train, y_train, 0)
plot_sample(X_train, y_train, 5)
```

Block 07: Normalizing the Data: These lines scale the pixel values of images from the range [0, 255] to [0, 1]. Normalization improves the learning process by ensuring that input values are small and consistent.

```
X_train = X_train / 255.0
X_test = X_test / 255.0
```

Block 08: Splitting Training and Validation Sets: This line uses train_test_split from sklearn to divide the original training set into a new training set and a validation set (80% train, 20% validation). This allows the model to be evaluated during training without using the test set.

```
from sklearn.model_selection import train_test_split
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2, random_state=42)
```

Block 09: Defining the CNN Model: This block creates a convolutional neural network using the Sequential model. Here's what each layer does:

- Conv2D (32, (3,3), ...): Applies 32 filters of size 3x3 to extract features from the input image.
- MaxPooling2D ((2,2)): Reduces the feature map size to half, improving computational efficiency and reducing overfitting.
- Another two Conv2D and MaxPooling2D layers deepen the network and help it learn more complex features.
- Flatten(): Converts the 2D feature maps into a 1D vector.
- Dense(64, activation='relu'): A fully connected layer with 64 neurons.
- Dense(10, activation='softmax'): Final output layer with 10 neurons (one per class), using softmax for multiclass probability output.

```
cnn = models.Sequential([
    layers.Conv2D(32, (3,3), activation='relu', input_shape=(32,32,3)),
    layers.MaxPooling2D((2,2)),
    layers.Conv2D(64, (3,3), activation='relu'),
    layers.MaxPooling2D((2,2)),
    layers.Conv2D(64, (3,3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

Block 10: Compiling the Model: This line configures the model for training:

- optimizer='adam': An efficient gradient descent algorithm
- loss='sparse_categorical_crossentropy': A loss function for classification with integer labels.
- metrics=['accuracy']: Tells Keras to evaluate model performance using accuracy.

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

Block 11: Training the Model with Validation: This trains the CNN model for 10 epochs using the training data. At each epoch, it also evaluates performance on the validation set. The training history (loss and accuracy for each epoch) is stored in the history object.

```
print("Training CNN model...")
history = cnn.fit(X_train, y_train, epochs=10, validation_data=(X_val, y_val))
```

Block 12: Evaluating the Model on Test Set: After training, the model is evaluated on the test dataset to get final performance metrics. The evaluate function returns the loss and accuracy, and the result is printed as a percentage.

```
print("\nEvaluating on test data...")
test_loss, test_accuracy = cnn.evaluate(X_test, y_test)
print(f"Test Accuracy: {test_accuracy*100:.2f}%")
```

Block 13: Plotting Training & Validation History: This function uses matplotlib to create two plots side by side:

- One for training and validation accuracy over epochs.
- One for training and validation loss. These graphs help analyze if the model is overfitting or underfitting.

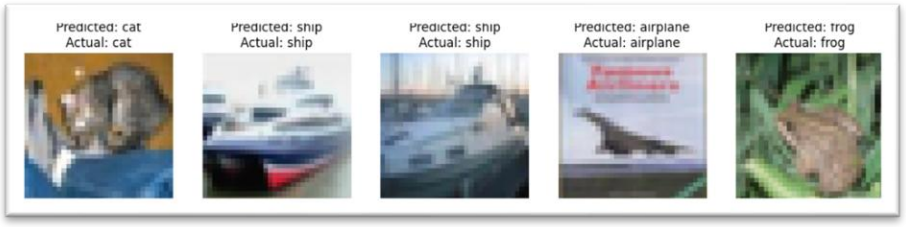
```
def plot_history(history):
    plt.figure(figsize=(12,5))
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Train Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.title('Accuracy over Epochs')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title('Loss over Epochs')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    plt.tight_layout()
```

```
    plt.show()

plot_history(history)
```

Block 14: Making Predictions and Showing Results: This function predicts labels for a few test images and displays the image alongside the predicted class and actual class. np.argmax() is used to find the most probable class.

```
def show_predictions(model, X, y, num=5):
    predictions = model.predict(X[:num])
    pred_classes = np.argmax(predictions, axis=1)

    plt.figure(figsize=(15,3))
    for i in range(num):
        plt.subplot(1, num, i+1)
        plt.imshow(X[i])
        plt.title(f"Predicted: {classes[pred_classes[i]]}\nActual: {classes[y[i]]}")
        plt.axis('off')
    plt.show()

show_predictions(cnn, X_test, y_test, num=5)
```

Results and Insights: After 10 epochs of training,
Test accuracy is 68.86% and the model show the predicted outputs as following:



Experiment Number: 06

Experiment Name: Write a program to evaluate a Recurrent Neural Network (RNN) for text classification.

Objectives: The primary objective of this experiment is to understand the architecture and working principles of Recurrent Neural Networks (RNNs) in handling sequential data.

Theory: Recurrent Neural Networks (RNNs) are a class of neural networks specifically designed to handle sequential data such as time series, natural language, or audio. Unlike traditional feed forward neural networks, RNNs have loops in their architecture, allowing information to persist and be reused across different time steps. This makes them especially powerful for tasks where the current input is dependent on previous inputs.
Why RNN: Recurrent Neural Networks (RNNs) were introduced to overcome the limitations of traditional Artificial Neural - (ANNs) when dealing with sequence data. RNNs address the 3 main issues shown in figure 1.

- Traditional ANNs expect fixed-size input and output, which is a problem in tasks like text generation or speech recognition. Where RNNs can process sequences of arbitrary length, making them suitable for dynamic input/output sizes.
- In ANNs, modeling temporal dependencies over long sequences often requires deep, complex architectures. But RNNs reuse the same weights across time steps, reducing model size and computation.
- ANNs treat each input independently, which leads to a lack of context and large numbers of parameters. On the other hand, RNNs share parameters across all time steps, allowing the network to generalize better and learn temporal patterns efficiently.

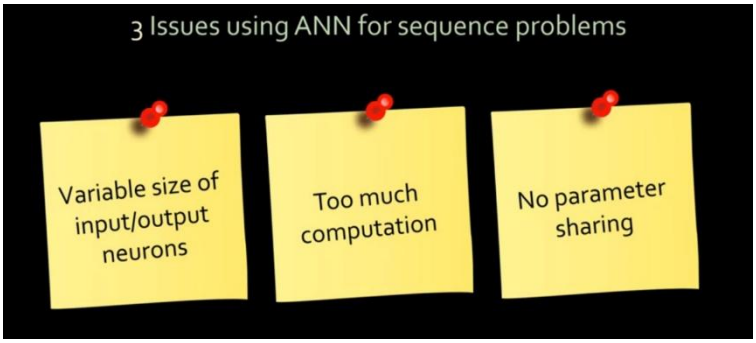


Figure 01: Three issues using ANN for sequence problems.

For example, when training an RNN for a task like name/entity identification (also called Named Entity Recognition, or NER), each word in the input sequence is passed through the RNN one step at a time.
In this example (shown in figure 2), the sentence is: "Ironman punched on hulk's face"
Here, the task is to identify names in the sequence (like "Ironman" and "hulk").
The expected output is: 1 0 0 1 0 (1 if the word is a name, 0 otherwise)
Here's how training works:

- Each word is input sequentially into the RNN.

- At each time step, the model produces a prediction y (whether the current word is a name or not).
- This prediction is compared with the actual label (0 or 1), and a loss is calculated.
- The total loss is the sum of all individual losses across the sequence.
- The model uses this total loss to update its weights using backpropagation through time (BPTT).

This structure allows the model to learn from context, so it can distinguish whether a word is a name based on surrounding words — for example, recognizing that "hulk's" refers to a person because of the possessive form and prior context.

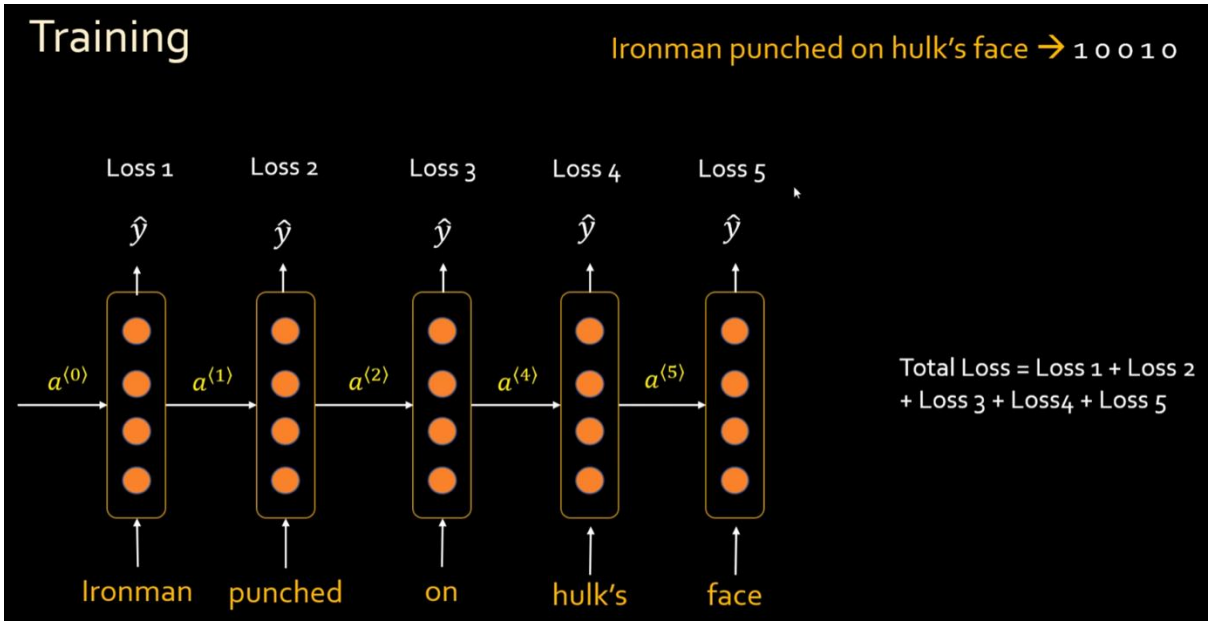


Figure 02: Named Entity Recognition using RNN

The working principle of an RNN revolves around the concept of a hidden state, which captures the memory of the network. At each time step t, the RNN takes the input x_t and the previous hidden state h_{t-1} to compute the new hidden state h_t. This hidden state acts as a summary of all previous inputs. The recurrence relation is typically defined by the equation:

$$h_t = tanh(W_{xh} * x_t + W_{hh} * h_{t-1} + b_h)$$

here, W_{xh} and W_{hh} are weight matrices for the input and hidden state, and b_h is the bias term. This equation updates the memory of the network. It combines the current input and past memory (via h_{t-1}), processes them through a weighted sum, and then squashes the result with the tanh function to get the new memory.

The output y_t at each step is often computed as:

$$y_t = W_{hy} * h_t + b_y$$

Here, W_{hy} is weight matrix from hidden to output layer

To improve learning and avoid vanishing gradients, the error is backpropagated through time (BPTT) using:

$$\frac{\partial L}{\partial W} = \sum \frac{\partial L}{\partial h_t} * \frac{\partial h_t}{\partial W}$$

Here, L is the loss function, $\frac{\partial L}{\partial h_t}$ is gradient of loss w.r.t. hidden state and $\frac{\partial h_t}{\partial W}$ is gradient of hidden state w.r.t. weight. To train an RNN, we need to compute gradients of the loss with respect to all weights. This equation shows that we accumulate gradients over all time steps to properly update the weights.

$$\frac{\partial h_t}{\partial W} = \frac{\partial \tanh(.)}{\partial W} = \left(1 - {h_t}^2\right) * input$$

This is the derivative of the tanh activation function, used during backpropagation to compute how the hidden state changes with respect to the weights. The term $\left(1 - {h_t}^2\right)$ comes from the derivative of tanh, and it is multiplied by the input to complete the gradient.

However, RNNs suffer from problems like vanishing and exploding gradients when dealing with long sequences. To address this, advanced variants like Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) were introduced. These use gating mechanisms to control the flow of information and preserve long-term dependencies more effectively.

RNNs find widespread applications in natural language processing (NLP) tasks such as sentiment analysis, machine translation, and text generation. They're also used in speech recognition, music composition, video captioning, and time-series forecasting.

Despite their strength in modeling sequences, vanilla RNNs are being gradually replaced by more efficient and scalable architectures like Transformers. Still, RNNs remain foundational in understanding the evolution of deep learning for sequential data.

Code with explanation: To understand RNN more efficiently, we can consider the following example, which walks through a real-world binary text classification task using movie reviews. We'll use TensorFlow and the IMDB dataset for this purpose, and try different RNN variants to understand how they behave.

First of all, install and import all the necessary packages. TensorFlow provides high-level APIs that simplify the model-building and training process. Along with TensorFlow, we'll use NumPy for general-purpose numerical operations.

```
import tensorflow as tf
from tensorflow import keras
import numpy as np
```

Then, load the popular IMDB dataset which is preprocessed and readily available via Keras. This dataset contains 25,000 movie reviews for training and 25,000 for testing, labeled by sentiment (positive/negative). We limit the vocabulary size and handle special tokens for better analysis.

```
vocab_size = 10000
max_length = 200
(x_train, y_train), (x_test, y_test) = keras.datasets.imdb.load_data(num_words=vocab_size)
start_char = 1
oov_char = 2
index_from = 3
word_index = keras.datasets.imdb.get_word_index()
inverted_word_index = dict(
   (i + index_from, word) for (word, i) in word_index.items()
)
inverted_word_index[start_char] = "[START]"
inverted_word_index[oov_char] = "[OOV]"
decoded_sequence = " ".join(inverted_word_index[i] for i in x_train[0])
print(decoded_sequence)
```

We can decode the first review to see what it looks like in human-readable form. This helps verify what kind of language and sentence structure is present in the data.

Output: [START] this film was just brilliant casting location scenery story direction everyone's really suited the part they played and you could just imagine being there robert [OOV] is an amazing actor and now the same being director [OOV] father came from the same scottish island as myself so i loved the fact there was a real connection with this film the witty remarks throughout the film were great it was just brilliant so much that i bought the film as soon as it was released for [OOV] and would recommend it to everyone to watch and the fly fishing was amazing really cried at the end it was so sad and you know what they say if you cry at a film it must have been good and this definitely was also [OOV] to the two little boy's that played the [OOV] of norman and paul they were just brilliant children are often left out of the [OOV] list i think because the stars that play them all grown up are such a big profile for the whole film but these children are amazing and should be praised for what they have done don't you think the whole story was so lovely because it was true and was someone's life after all that was shared with us all

Next, we preprocess the dataset by padding all sequences to the same length. This ensures uniformity in input shape, which is required when feeding data into RNN models.

```
x_train = keras.preprocessing.sequence.pad_sequences(x_train, maxlen=max_length)
x_test = keras.preprocessing.sequence.pad_sequences(x_test, maxlen=max_length)
```

LSTM Model: LSTM networks are a type of recurrent neural network (RNN) designed to learn from sequences of data. Unlike traditional RNNs, LSTMs can remember information over long periods of time, making them useful for tasks like language modeling, sentiment analysis, and time series forecasting.
They achieve this by using memory cells and gates:

- Forget Gate: Decides what information from the past should be discarded.
- Input Gate: Decides what new information should be added to the memory.
- Cell State Update: Combines the forget and input gates to update the memory.
- Output Gate: Determines what the next hidden state (output) should be.

These components allow LSTM to retain and forget information dynamically as it reads through a sequence.

```
# LSTM model
model = keras.Sequential([
   keras.layers.Embedding(vocab_size, 32, input_length=max_length),
   keras.layers.LSTM(64, return_sequences=True),
   keras.layers.LSTM(32),
   keras.layers.Dense(1, activation='sigmoid')
])
```

GRU Model: GRU is a simpler and faster alternative to LSTM. It combines some of the internal gates found in LSTMs, which reduces the number of operations and makes training more efficient.

GRU uses:

- Update Gate: Controls how much of the past information is carried forward.
- Reset Gate: Decides how much past information to forget when computing new content.

Despite being simpler, GRUs often perform just as well as LSTMs on many tasks.

```
# GRU model
model2 = keras.Sequential([
   keras.layers.Embedding(vocab_size, 64, input_length=max_length),
   keras.layers.GRU(64, return_sequences=True),
   keras.layers.GRU(32),
   keras.layers.Dense(1, activation='sigmoid')
])
```

Bidirectional LSTM Model: A Bidirectional LSTM reads input sequences in both forward and backward directions, capturing context from both past and future words. This is especially powerful in NLP tasks where understanding both the beginning and end of a sentence improves accuracy.

Instead of just processing the input left-to-right, it also does right-to-left and then combines both outputs, giving the model more context.

```
  # Bidirectional LSTM
model3 = keras.Sequential([
   keras.layers.Embedding(vocab_size, 64, input_length=max_length),
   keras.layers.Bidirectional(keras.layers.LSTM(64)),
   keras.layers.Dense(1, activation='sigmoid')
])
```

After model creation, we compile them. Compilation configures the training process by specifying the optimizer, loss function, and evaluation metric.

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

Then we train the model using the `fit` function. This function feeds the training data through the network for a number of epochs, adjusting weights using backpropagation and gradient descent.

```
model.fit(x_train, y_train, epochs=5, validation_data=(x_test, y_test))
```

After training, we evaluate the model's performance on the test dataset. The `evaluate` function returns the loss and accuracy metrics.

```
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f'\nTest accuracy: {test_acc}')
```

The differences between **LSTM**, **GRU**, and **Bidirectional LSTM** models are following:

Features	LSTM	GRU	Bidirectional LSTM
Full Name	Long Short-Term Memory	Gated Recurrent Unit	Bidirectional Long Short-Term Memory
Architecture Complexity	High (3 gates: input, forget, output)	Medium (2 gates: update, reset)	Very High (2 LSTMs running in opposite directions)
Training Speed	Moderate (slower than GRU)	Fast	Slow (due to dual LSTM layers)
Memory Usage	High	Medium	Very High
Context Captured	Past (previous time steps)	Past (like LSTM but less complex)	Past and Future (both directions)
Performance	High accuracy on long sequence data	Comparable to LSTM, slightly less in some cases	Often highest due to richer context
Use Case Suitability	Complex NLP tasks with long dependencies	Faster tasks or when resources are limited	Tasks needing full sequence context (e.g., NLP)
Overfitting Risk	Moderate	Lower	Higher (if not regularized well)
Parameter Count	Higher (more gates)	Lower	Highest (doubled LSTM layers)

Results and Insights

After 5 epochs of training:

- Model 1 (LSTM) achieved 0.8562 accuracy
- Model 2 (GRU) achieved 0.8684 accuracy
- Model 3 (Bidirectional LSTM) achieved 0.8594 accuracy

Although the GRU model performed best, the bidirectional model slightly outperforms the standard LSTM because it can learn dependencies in both directions. This is beneficial in natural language tasks where context from both past and future words influences understanding.

Experiment Number: 07

Experiment Name: Write a program to evaluate a Transformer model for text classification.

Objectives: To evaluate the performance of a Transformer-based model (like BERT) for text classification tasks by leveraging transfer learning, optimizing with labeled data, and assessing accuracy, precision, and recall on unseen text inputs.

Theory: Text classification means sorting text into different groups. For example, with movie reviews, we might want to label them as positive (happy, good) or negative (bad, disappointing).
One of the best models for this is BERT, a powerful AI that understands language really well. BERT is special because it pays attention to the important words in a sentence, even if those words are far apart.

Explanation of BERT Architecture: The architecture of BERT (Bidirectional Encoder Representations from Transformers), a powerful pre-trained language model developed by Google. Let's break down the architecture:

1. Input Text

- This is the raw text input, such as a sentence or a pair of sentences, that BERT will process.

2. Tokenization (WordPiece)

- BERT uses WordPiece Tokenization to break words into subword units.
- Special Tokens:
    - [CLS]: A classification token added at the beginning of every input.
    - [SEP]: Separator token used between sentence pairs or to signal the end of a single sentence.
    - [MASK]: Used during training for the masked language modeling task.

3. Embeddings Layer: This layer combines three types of embeddings to represent each token:

- Token Embeddings: The actual meaning of the word or subword.
- Position Embeddings: Information about the token's position in the sentence.
- Segment Embeddings: Distinguish between sentence A and B when processing sentence pairs.

These embeddings are summed and passed to the encoder.

4. Transformer Encoder (12 or 24 layers): BERT uses the Transformer architecture, but only the encoder part (not the decoder like in GPT). Each encoder block includes:

- Multi-Head Self-Attention: Helps the model focus on different parts of the input sentence simultaneously, learning contextual relationships.
- Add & Normalize: Adds a residual (shortcut) connection around each sub-layer and normalizes the result for stable training.
- Feed-Forward Network: A small neural network applied to each token separately, used for deeper representation.

- Another Add & Normalize layer follows the feed-forward network.

The number of these encoder blocks:

- BERT-Base: 12 layers
- BERT-Large: 24 layers

5. Final Hidden State of [CLS] Token

- After all transformer layers, the final representation of the [CLS] token is used as the summary representation of the input.
- Especially useful for classification tasks.

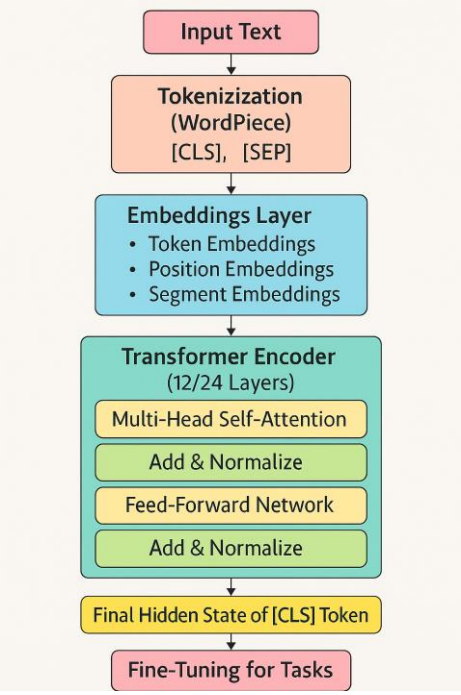


Figure 01: Generalize work flow of a transformer (BERT) model

6. Fine-Tuning for Tasks: After pre-training, BERT is fine-tuned for specific downstream tasks like:

- Sentiment Analysis
- Question Answering
- Named Entity Recognition
- Next Sentence Prediction (during pre-training)

Mathematical Foundation of BERT:

1. Input Representation

Each input token is represented by:

$$E = TokenEmbedding + Position\ Embedding + Segment\ Embedding$$

- TokenEmbedding: WordPiece token embeddings (vocab size = 30,000).
- PositionEmbedding: Learnable positional encoding (maximum length = 512).
- SegmentEmbedding: Embedding indicating Sentence A or B (for Next Sentence Prediction).

Thus, each token is mapped into a 768-dimensional vector (for BERT-base).

2. Transformer Encoder

Each Transformer block consists of:

- Multi-Head Self-Attention
- Feed-Forward Network (FFN)
- Residual Connections and Layer Normalization

2.1 Multi-Head Self-Attention

Each attention head is calculated as:

$$Head_i = Attention\ (QW_i^Q, KW_i^K VW_i^v)$$

where:

$$Attention\ (Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})v$$

- $(Q, K, V)$are query, key, value matrices.
- d_k=64 (for BERT-base).
- $W_i^Q, W_i^K W_i^v \epsilon \mathbb{R}^{768\times64}4$ are learned projection matrices.

Then, concatenate all heads:

$$MultiHead(Q, K, V) = Concat(Head_1, Head_2, \dots\dots\dots, Head_{12})W^O$$

where:

$W^O \in \mathbb{R}^{768\times64}$

2.2 Feed-Forward Network (FFN)

Each token independently passes through a two-layer feed-forward network:

$FFN(x) = Max(0, xW_1 + b_1)W_2 + b_2$

where:

- $W_1 \epsilon\ \mathbb{R}^{768\times3072}$
- $W_2 \in\ \mathbb{R}^{3072\times768}$

(Notice: hidden size expands by 4x.)

2.3 Residual Connection and Layer Normalization

After each sub-layer (attention or FFN):

$$Output = LayerNorm(x + Sublayer(x))$$

- Adds original input to output (**residual**) and applies **layer normalization**.

3. Special Tokens

BERT adds special tokens:

- [CLS] token at the beginning (used for classification).
- [SEP] token between two sentences.
- [MASK] token for masking during pre-training.

4. Pre-Training Objectives

BERT is trained with two main tasks:

4.1 Masked Language Modeling (MLM)

Randomly mask 15% of tokens, and predict them.
Loss:

$$\zeta_{MLM} = -\sum \log P\ (x_{masked}|x_{context})$$

where:

$$P\ (x_{masked}|x_{context}) = softmax(W_{lm}h_l(x))$$

- W_{lm} = weight matrix for the vocabulary.
- $h_l(x)$= final hidden state of the masked token.

4.2 Next Sentence Prediction (NSP)

Predict if sentence B follows sentence A.
Binary classification loss:

$$\zeta_{NSP} = -y\ log\sigma\left(w^T h_{|CLS|}\right) + (1-y)\log\ (1-\sigma\left(w^T h_{|CLS|}\right))]$$

re:

- $h_{|CLS|}$= final hidden state of the [CLS] token.
- w= weight vector for NSP classifier.
- σ = sigmoid activation.

5. Final Output for Downstream Tasks

For classification tasks, use the hidden state of [CLS]:
$y = softmax\left(W^c h_{|CLS|}\right)$where:

- $W^c \in \mathbb{R}^{K\times 768}$ (K = number of classes).

Summary: BERT combines:

- Multi-layer bidirectional self-attention
- Position-wise feed-forward networks
- Masked Language Modeling (MLM)
- Next Sentence Prediction (NSP)

It uses deeply stacked Transformer encoders (12 layers for BERT-base).

BERT (Bidirectional Encoder Representations from Transformers) is a deep learning model that generates **context-aware vector representations** for input text. In sentiment classification, these representations help determine whether a text expresses a **positive** or **negative** sentiment.
The process involves three major stages:

1. **Contextual Embedding Generation**: BERT processes the input through stacked transformer layers using **self-attention**, creating embeddings that capture relationships between all tokens.
2. **Sentence Representation**: The special **[CLS]** token is added at the beginning of the input. After processing, its final-layer embedding serves as a **summary vector** for the entire sequence.

3. **Classification Head**: A **simple neural network layer** maps the [CLS] representation to a **sentiment score**, followed by a **sigmoid function** to produce a probability. This probability indicates the likelihood of the sentiment being positive.
4. **Fine-tuning**: During training, the model adjusts its internal weights to minimize prediction errors using a **binary classification loss function**.

Code with explanation:

Block 1: Importing Libraries

```
import os
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from transformers import BertTokenizer, BertModel
from datasets import load_dataset
from torch.amp import autocast, GradScaler
from tqdm import tqdm
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from datasets import concatenate_datasets
```

Explanation: This block imports all necessary libraries and modules:

- torch, nn, optim, and DataLoader are used for building and training deep learning models.
- transformers provides access to BERT models and tokenizers.
- datasets library loads and processes datasets like IMDB.
- autocast and GradScaler from torch.amp enable mixed precision training.
- tqdm adds progress bars.
- sklearn.metrics helps with evaluation metrics.
- matplotlib and seaborn are used for plotting.
- concatenate_datasets helps in merging subsets of datasets.

Block 2: Setup and Dataset Loading

```
os.environ["HF_HUB_DISABLE_SYMLINKS_WARNING"] = "1"
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

dataset = load_dataset("imdb")

print("Training dataset label distribution:", dataset["train"].features["label"].names)
print("Test dataset label distribution:", dataset["test"].features["label"].names)
```

Explanation:

- Disables a warning from Hugging Face.
- Detects whether to use GPU or CPU for training.

- Loads the full IMDB dataset, which contains movie reviews labeled as positive or negative.
- Prints out the label names to show class distribution (e.g., 0: negative, 1: positive).

Block 3: Balancing the Dataset

```
positive_train_data = dataset["train"].filter(lambda example: example["label"] == 1)
negative_train_data = dataset["train"].filter(lambda example: example["label"] == 0)

positive_test_data = dataset["test"].filter(lambda example: example["label"] == 1)
negative_test_data = dataset["test"].filter(lambda example: example["label"] == 0)

balanced_train_data = concatenate_datasets([positive_train_data.select(range(2000)), negative_train_data.select(range(2000))])
balanced_test_data = concatenate_datasets([positive_test_data.select(range(400)), negative_test_data.select(range(400))])
```

Explanation:

- Filters the dataset to separate positive and negative examples for both training and testing.
- Selects an equal number (2000 train, 400 test) from each class to create a balanced dataset.
- This prevents class imbalance, which could bias the model toward the majority class.

Block 4: Tokenization

```
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")

def tokenize_function(example):
  return tokenizer(example["text"], padding="max_length", truncation=True, max_length=256)

print("Tokenizing...")
tokenized_train = balanced_train_data.map(tokenize_function, batched=True)
tokenized_test = balanced_test_data.map(tokenize_function, batched=True)
tokenized_train.set_format(type='torch', columns=['input_ids', 'attention_mask', 'label'])
tokenized_test.set_format(type='torch', columns=['input_ids', 'attention_mask', 'label'])
```

Explanation:

- Loads the pre-trained tokenizer from BERT base (lowercase).
- Defines a tokenization function that processes each text sample:
  - Pads to a fixed length (256),
  - Truncates longer sequences,
  - Converts text into token IDs and attention masks.
- Applies this function to both train and test datasets.
- Sets the format for PyTorch, specifying which columns to use.

Block 5: Creating DataLoaders

```
train_loader = DataLoader(tokenized_train, batch_size=64, shuffle=True)
test_loader = DataLoader(tokenized_test, batch_size=64,  shuffle= False)
```

Explanation:

- Wraps the tokenized datasets into PyTorch DataLoaders.
- train_loader enables mini-batch training with a batch size of 64 and shuffling for better generalization.
- test_loader does not shuffle to keep evaluation consistent.

Block 6: Defining the BERT-based Model

```
class BERTSentimentClassifier(nn.Module):
    def __init__(self):
        super(BERTSentimentClassifier, self).__init__()
        self.bert = BertModel.from_pretrained("bert-base-uncased")
        self.dropout = nn.Dropout(0.3)
        self.classifier = nn.Linear(self.bert.config.hidden_size, 2)

    def forward(self, input_ids, attention_mask):
        outputs = self.bert(input_ids=input_ids, attention_mask=attention_mask)
        pooled_output = outputs.pooler_output
        pooled_output = self.dropout(pooled_output)
        logits = self.classifier(pooled_output)
        return logits
```

Explanation:

- A custom PyTorch class for binary sentiment classification.
- BertModel provides pretrained BERT encoder.
- The [CLS] token's output (from pooler_output) is passed through a dropout layer to prevent overfitting.
- A fully connected (Linear) layer maps the 768-dim BERT output to 2 classes (positive/negative).
- The forward function returns raw class logits (not softmaxed yet).

Block 7: Model, Loss Function, Optimizer, and AMP Setup

```
model = BERTSentimentClassifier().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(model.parameters(), lr=2e-5)

scaler = GradScaler()
```

Explanation:

- Instantiates the model and moves it to the correct device (GPU or CPU).
- Uses Cross Entropy Loss for classification.
- AdamW optimizer is commonly used with transformers.
- GradScaler() enables automatic mixed precision (AMP), which speeds up training while reducing memory usage.

Block 8: Training the Model

```
def train(model, dataloader, optimizer, criterion, scaler):
    model.train()
    running_loss = 0.0

    for batch in tqdm(dataloader, desc="Training"):
        input_ids = batch["input_ids"].to(device)
        attention_mask = batch["attention_mask"].to(device)
        labels = batch["label"].to(device)

        optimizer.zero_grad()

        with autocast():
            outputs = model(input_ids, attention_mask)
            loss = criterion(outputs, labels)

        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()

        running_loss += loss.item()

    return running_loss / len(dataloader)
```

Explanation:

- Puts the model in training mode.
- Iterates over training data in batches.
- Uses mixed precision (autocast) to speed up computations.
- Backpropagates gradients using scaled loss and updates weights using GradScaler.
- Tracks and returns average loss over the epoch

Block 9: Evaluation Function

```
def evaluate(model, dataloader):
    model.eval()
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for batch in tqdm(dataloader, desc="Evaluating"):
            input_ids = batch["input_ids"].to(device)
            attention_mask = batch["attention_mask"].to(device)
            labels = batch["label"].to(device)

            outputs = model(input_ids, attention_mask)
            preds = torch.argmax(outputs, dim=1)

            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())
```

```
    return all_preds, all_labels
```

Explanation:

- Puts the model in evaluation mode (disables dropout etc.).
- Turns off gradient computation to speed up evaluation.
- Uses the model to predict classes for each batch.
- Collects predicted and true labels for computing metrics later.

Block 10: Running the Training Loop

```
num_epochs = 3
for epoch in range(num_epochs):
    print(f"Epoch {epoch+1}/{num_epochs}")
    train_loss = train(model, train_loader, optimizer, criterion, scaler)
    print("Train Loss:", train_loss)
    preds, labels = evaluate(model, test_loader)
    print(classification_report(labels, preds, target_names=["Negative", "Positive"]))
```

Explanation:

- Runs training and evaluation for 3 epochs.
- After each epoch, evaluates the model on the test set.
- Uses classification_report from scikit-learn to print precision, recall, F1-score for each class.

Block 11: Confusion Matrix Visualization

```
cm = confusion_matrix(labels, preds)
plt.figure(figsize=(6, 4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=["Negative", "Positive"],
yticklabels=["Negative", "Positive"])
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Confusion Matrix")
plt.show()
```

Explanation:

- Computes a confusion matrix comparing predicted vs actual classes.
- Uses Seaborn to draw a heatmap of the confusion matrix.
- Helps visualize model performance and misclassifications.

Block 12: Predicting Sentiment on Custom Text Inputs

```
def predict_sentiment(text, model, tokenizer, device):
    model.eval()
    with torch.no_grad():
        # Tokenize input
        encoding = tokenizer(
            text,
            return_tensors='pt',
            truncation=True,
```

```
        padding='max_length',
        max_length=256
    )
    input_ids = encoding['input_ids'].to(device)
    attention_mask = encoding['attention_mask'].to(device)

    # Predict
    output = model(input_ids, attention_mask)
    prob = torch.sigmoid(output)

    # Result
    sentiment = "Positive" if prob.item() > 0.5 else "Negative"
    print(f"\nInput Text: {text}")
    print(f"Predicted Sentiment: {sentiment} (Confidence: {prob.item():.2f})")

user_input = "Every one should watch the movie at least once"
predict_sentiment(user_input, model, tokenizer, device)

user_input = "This movie was boring and didn't make sense."
predict_sentiment(user_input, model, tokenizer, device)
```

Explanation:

- This function lets you input your own sentence and get the predicted sentiment from the trained BERT model.
- model.eval() puts the model in inference mode.
- torch.no_grad() disables gradient calculation for faster computation and lower memory use.
- The tokenizer converts the input text into token IDs and attention masks (BERT expects fixed-length inputs, so it pads/truncates to max_length=256).
- The tokenized inputs are passed to the model.
- The model outputs **logits**, and torch.sigmoid() is applied to squash them into the range [0, 1]. This works well here even though it's a 2-class classification (you could also use softmax + argmax).
- If the predicted probability > 0.5, it's considered "Positive"; otherwise, "Negative".
- The function prints both the input and the predicted result with confidence.

Results: Test accuracy is 90.62% and the predictions look like:

- Input Text: Every one should watch the movie at least once
- Predicted Sentiment: Positive (Confidence: 0.91)
- Input Text: This movie was boring and didn't make sense.
- Predicted Sentiment: Negative (Confidence: 0.00)

Experiment Number: 08

Experiment Name: Write a program to evaluate Generative Adversarial Network (GAN) for image generation.

Objectives: The primary objective of this program is to evaluate the performance of a Generative Adversarial Network (GAN) trained for image generation tasks.

Theory: Generative Adversarial Networks (GANs) were introduced by Ian Goodfellow and his colleagues in 2014. GANs are a class of neural networks that autonomously learn patterns in the input data to generate new examples resembling the original dataset. GAN's architecture consists of two neural networks:

1. Generator: creates synthetic data from random noise to produce data so realistic that the discriminator cannot distinguish it from real data.
2. Discriminator: acts as a critic, evaluating whether the data it receives is real or fake.

They use adversarial training to produce artificial data that is identical to actual data.

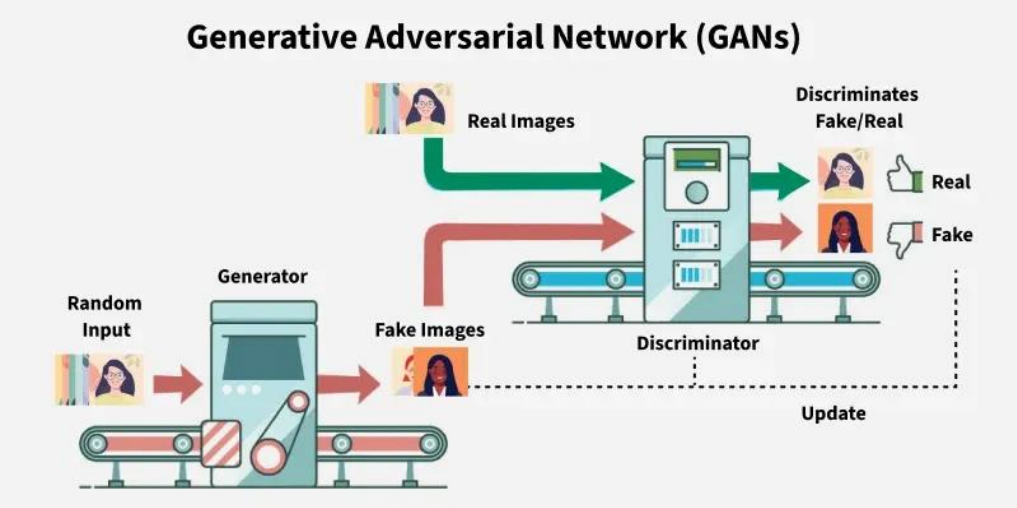


Figure 01: A simple overview of Generative Adversarial Network (GANs)

The two networks engage in a continuous game of cat and mouse: the Generator improves its ability to create realistic data, while the Discriminator becomes better at detecting fakes. Over time, this adversarial process leads to the generation of highly realistic and high-quality data.

Explanation of GAN architecture: Let's explore the generator and discriminator model of GANs in detail:

1. Generator Model: The generator is a deep neural network that takes random noise as input to generate realistic data samples (e.g., images or text). It learns the underlying data distribution by adjusting its parameters through backpropagation.

The generator's objective is to produce samples that the discriminator classifies as real.

The loss function is:

$$J_G = -\frac{1}{m}\sum_{i=1}^{m}\log D(G(Z_i))$$

Where,

- J_G measure how well the generator is fooling the discriminator.
- $\log D(G(Z_i))$ represents log probability of the discriminator being correct for generated samples.
- The generator aims to minimize this loss, encouraging the production of samples that the discriminator classifies as real $\log D(G(Z_i))$, close to 1.

2. Discriminator Model: The discriminator acts as a binary classifier, distinguishing between real and generated data. It learns to improve its classification ability through training, refining its parameters to detect fake samples more accurately. When dealing with image data, the discriminator often employs convolutional layers or other relevant architectures suited to the data type. These layers help extract features and enhance the model's ability to differentiate between real and generated samples. The discriminator reduces the negative log likelihood of correctly classifying both produced and real samples. This loss incentivizes the discriminator to accurately categorize generated samples as fake and real samples with the following equation:

$$J_D = -\frac{1}{m}\sum_{i=1}^{m}\log D(x_i) \; -\frac{1}{m}\sum_{i=1}^{m}\log(1 - D(G(Z_i))$$

Where,

- J_D assesses the discriminator's ability to discern between produced and actual samples.
- The log likelihood that the discriminator will accurately categorize real data is represented by $\log D(x_i)$.
- The log chance that the discriminator would correctly categorize generated samples as fake is represented by $\log(1 - D(G(Z_i)))$.

By minimizing this loss, the discriminator becomes more effective at distinguishing between real and generated samples.

The other term is:

MinMax Loss: GANs follow a minimax optimization where the generator and discriminator are adversaries:

$$min_G, max_D\ (G, D) = [E_{x\sim p_{data}}\left[\log D(x)\right] + E_{z\sim p_{z(z)}}[\log(1 - D(G(Z_i))\right)$$

Where,

- G is generator network and is D is the discriminator network
- Actual data samples obtained from the true data distribution$p_{data}(x)$ are represented by x.
- Random noise sampled from a previous distribution $p_z(z)$ usually a normal or uniform distribution) is represented by z.

- D(x) represents the discriminator's likelihood of correctly identifying actual data as real.
- D(G(z)) is the likelihood that the discriminator will identify generated data coming from the generator as authentic.

The generator aims to minimize the loss, while the discriminator tries to maximize its classification accuracy.

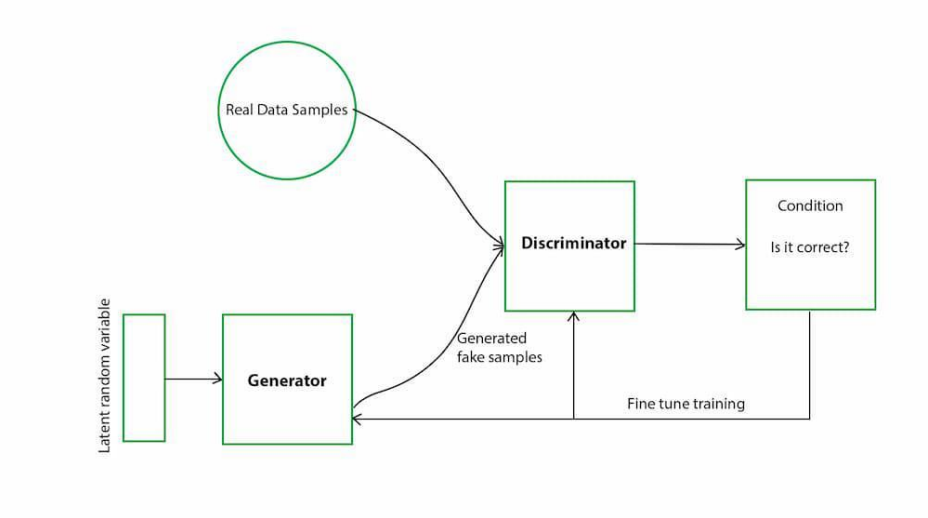


Figure 02: Generator-Discriminator Interaction

How does a GAN work: Let's understand how the generator (G) and discriminator (D) complete to improve each other over time:

- **Generator's First Move:** G takes a random noise vector as input. This noise vector contains random values and acts as the starting point for G's creation process. Using its internal layers and learned patterns, G transforms the noise vector into a new data sample, like a generated image.
- **Discriminator's Turn:** D receives two kinds of inputs:
  - ✓ Real data samples from the training dataset.
  - ✓ The data samples generated by G in the previous step.

  D's job is to analyze each input and determine whether it's real data or something G cooked up. It outputs a probability score between 0 and 1. A score of 1 indicates the data is likely real, and 0 suggests it's fake.
- **Adversarial Learning:**
  - ✓ If the discriminator correctly classifies real data as real and fake data as fake, it strengthens its ability slightly.
  - ✓ If the generator successfully fools the discriminator, it receives a positive update, while the discriminator is penalized.
- **Generator's Improvement:** Every time the discriminator misclassifies fake data as real, the generator learns and improves. Over multiple iterations, the generator produces more convincing synthetic samples.
- **Discriminator's Adaptation:** The discriminator continuously refines its ability to distinguish real from fake data. This ongoing duel between the generator and discriminator enhances the overall model's learning process.

- **Training Progression:**
  - ✓ As training continues, the generator becomes highly proficient at producing realistic data.
  - ✓ Eventually, the discriminator struggles to distinguish real from fake, indicating that the GAN has reached a well-trained state.
  - ✓ At this point, the generator can be used to generate high-quality synthetic data for various applications.

Code with explanation: In this experiment, we use the MNIST dataset, which contains 28x28 grayscale images of handwritten digits (0–9), to train and evaluate the GAN. The goal is for the Generator to learn the underlying distribution of MNIST digits and produce visually realistic handwritten digits that are indistinguishable from real samples.

Block 01: Import Libraries

```
import tensorflow as tf
from tensorflow.keras import layers, models
import numpy as np
import matplotlib.pyplot as plt
```

Explanation:

- TensorFlow & Keras: Used for building and training deep learning models.
- NumPy: Handles numerical operations (e.g., for random noise).
- Matplotlib: Used to visualize generated images.

Block 02: Load and Preprocess MNIST Dataset

```
(train_images, _), (_, _) = tf.keras.datasets.mnist.load_data()
train_images = train_images.reshape(train_images.shape[0], 28, 28, 1).astype('float32')
train_images = (train_images - 127.5) / 127.5  # Normalize to [-1, 1]
```

Explanation:

- Loads the **MNIST** dataset (we don't need the labels).
- Reshapes images to 28x28x1 and normalizes pixel values to the range [-1, 1] which is ideal for tanh activation used in the generator output.

Block 03: Dataset Pipeline

```
BUFFER_SIZE = 60000
BATCH_SIZE = 256
train_dataset = tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

Explanation:

- Creates a TensorFlow dataset from the training images, shuffles it, and batches it.
- Enables efficient training with mini-batches.

Block 04: Define the Generator

```
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
```

```
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    model.add(layers.Reshape((7, 7, 256)))
    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False, activation='tanh'))
    return model
```

Explanation:

- The Generator maps random noise (100-d vector) to a 28x28 image.
- Uses transposed convolutions to upsample the input gradually.
- Applies BatchNorm for stable training and LeakyReLU for non-linearity.
- Output layer uses tanh to match the image normalization.

Block 05: Define the Discriminator

```
def make_discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same', input_shape=[28, 28, 1]))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))
    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))
    model.add(layers.Flatten())
    model.add(layers.Dense(1))
    return model
```

Explanation:

- The Discriminator is a binary classifier that distinguishes real from fake images.
- Uses convolution layers to extract features and Dropout to prevent overfitting.
- Output is a single logit (no activation), processed later with binary cross-entropy loss
-

Block 06: Loss Functions

```
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    return real_loss + fake_loss

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)
```

Explanation:

- **Discriminator loss** penalizes it for incorrect classification of real and fake.
- **Generator loss** tries to fool the discriminator by making fake outputs look real.

Block 07: Optimizers

```
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

Explanation:

- **Adam** optimizer is used for both networks with a learning rate of 0.0001, which works well in practice for GANs.

Block 08: Initialize Models

```
generator = make_generator_model()
discriminator = make_discriminator_model()
```

Explanation:

- Instantiates the generator and discriminator.

Block 09: Training Step Function

```
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, 100])
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)
        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)
        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)
    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss,
discriminator.trainable_variables)
    generator_optimizer.apply_gradients(zip(gradients_of_generator,
generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
discriminator.trainable_variables))
    return gen_loss, disc_loss
```

Explanation: Performs one training step:

- Samples random noise.
- Generates images.
- Computes real/fake logits.
- Calculates loss.
- Computes gradients.
- Updates both generator and discriminator.

Block 10: Training Loop

```
def train(dataset, epochs):
    for epoch in range(epochs):
        for image_batch in dataset:
            gen_loss, disc_loss = train_step(image_batch)
```

```
    print(f'Epoch {epoch + 1}, Generator Loss: {gen_loss:.4f}, Discriminator Loss: {disc_loss:.4f}')
    if (epoch + 1) % 5 == 0:
      generate_and_save_images(generator, epoch + 1, seed)
```

Explanation:

- Loops through epochs and batches.
- Every 5 epochs, it generates and saves sample images for monitoring progress.

Block 11: Image Generation and Visualization

```
def generate_and_save_images(model, epoch, test_input):
  predictions = model(test_input, training=False)
  fig = plt.figure(figsize=(4, 4))
  for i in range(predictions.shape[0]):
    plt.subplot(4, 4, i+1)
    plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')
    plt.axis('off')
  plt.savefig(f'image_at_epoch_{epoch:04d}.png')
  plt.show()
```

Explanation:

- Uses the generator to produce a batch of images from fixed noise.
- Converts pixel values back to [0, 255] for visualization.
- Saves and displays the images as a 4x4 grid.

Block 12: Set Seed and Test the Model

```
seed = tf.random.normal([16, 100])
EPOCHS = 50
train(train_dataset, EPOCHS)
generate_and_save_images(generator, EPOCHS, seed)
```

Explanation:

- Sets a random seed for generating the same images at each checkpoint.
- Trains the GAN for 50 epochs.
- After training, generates a final set of images to visualize performance.

Results: Epoch 50, Generator Loss: 0.8876, Discriminator Loss: 1.2997