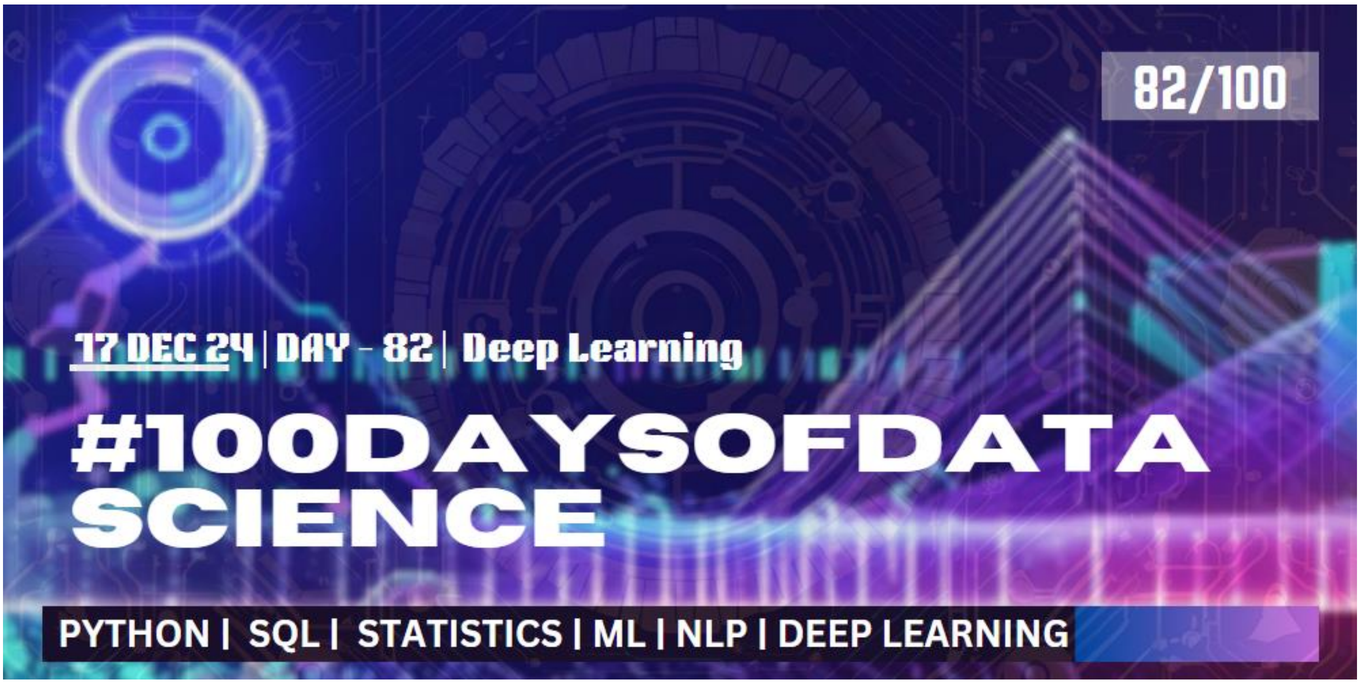


Perceptron

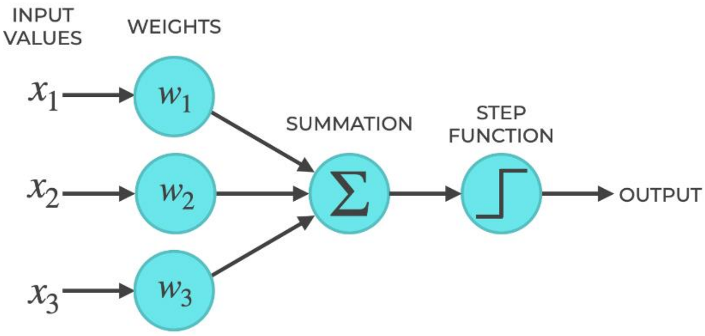
The **Perceptron** is one of the simplest types of artificial neural networks, introduced by Frank Rosenblatt in 1958. It's a foundational concept in machine learning, designed for binary classification tasks. The Perceptron models the way biological neurons work, making it an essential building block for modern deep learning systems.

Key Features of the Perceptron

1. **Linear Decision Boundary**
   The Perceptron separates data into two classes using a straight line (or hyperplane in higher dimensions).
2. **Adaptive Weights**
   The model learns by updating its weights based on classification errors during training.
3. **Binary Output**
   It produces a binary outcome (e.g., 0 or 1) based on the input and learned parameters.
4. **Iterative Learning**
   Uses a simple learning rule to minimize classification errors over multiple epochs.

Applications of the Perceptron

- **Spam Detection**: Classify emails as spam or not.
- **Binary Sentiment Analysis**: Determine positive or negative sentiment in text.
- **Stock Price Prediction**: Predict whether a stock will go up or down.



Workflow of the Perceptron Algorithm

1. **Initialize Parameters**
   Start with random weights and a bias term.

```
[239]: # Create an instance of the Perceptron with 2 features
ppn = Perceptron(num_features=2)

# Print the initialized weights
print("Initial weights:", ppn.weights)

# Print the initialized bias
print("Initial bias:", ppn.bias)

Initial weights: [0.0, 0.0]
Initial bias: 0
```

2. **Forward Pass**
   Compute the weighted sum of inputs and apply an activation function.

$$y = \text{step}\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

```
[240]: # Initialize the Perceptron model with zero features (likely a placeholder)
ppn = Perceptron(num_features=0)

# Define the input vector
x = [1.23, 2.13]

# Perform a forward pass with the input vector
# Debugging: Print the output of the forward pass
output = ppn.forward(x)
print("Forward Pass Output:", output)

Forward Pass Output: 0
```

3. **Weight Update Rule**
   Adjust weights based on errors using the formula:

$$w_i = w_i + \eta \cdot (y - \hat{y}) \cdot x_i$$

where η is the learning rate.

```
[241]: # Initialize the Perceptron model with two features
ppn = Perceptron(num_features=2)

# Define the input vector and the true label
x = [1.1, 2.1]
y_true = 1

# Update the perceptron model with the input vector and true label
# Debugging: Print the weights and bias after the update
ppn.update(x, y_true=y_true)
print("After Update:")
print("Weights:", ppn.weights)
print("Bias:", ppn.bias)

After Update:
Weights: [1.1, 2.1]
Bias: 1
```

4. **Repeat**
   Iterate through the data for several epochs until convergence.

Limitations of the Perceptron

1. **Linear Separability**
   The Perceptron only works when data is linearly separable. It fails for non-linearly separable datasets like the XOR problem.
2. **Binary Classification**
   It cannot handle multi-class problems without modifications.
3. **Convergence Time**
   May require many iterations to converge, especially for complex datasets.

Example: Perceptron Training in Python
Training the Perceptron

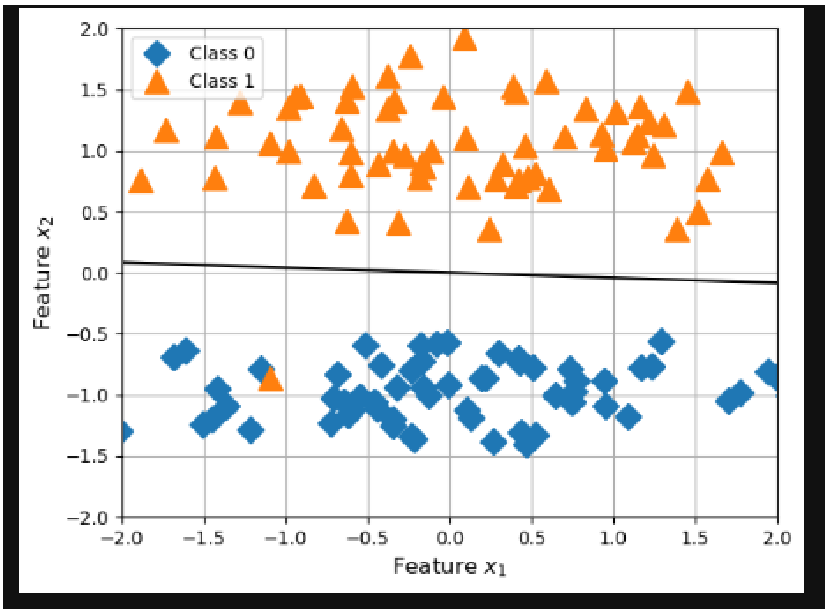
```
[243]: def train(model, X_train, y_train, epochs):
           # Training loop iterating through the specified number of epochs
           for epoch in range(epochs):
               error_count = 0  # Counter to track the number of misclassifications

               for x, y in zip(X_train, y_train):  # Iterate through each training sample
                   error = model.update(x, y)  # Update the model with the current sample
                   error_count += abs(error)  # Accumulate the absolute error for this epoch

               # Print the number of errors after each epoch
               print(f"Epoch {epoch + 1} errors {error_count}")


       # Create a perceptron object with 2 features
       ppn = Perceptron(num_features=2)

       # Train the perceptron using the training data for 10 epochs
       train(ppn, features, targets, epochs=10)

       Epoch 1 errors 6
       Epoch 2 errors 2
       Epoch 3 errors 4
       Epoch 4 errors 1
       Epoch 5 errors 2
       Epoch 6 errors 2
       Epoch 7 errors 3
       Epoch 8 errors 3
       Epoch 9 errors 2
       Epoch 10 errors 3
```

Computing Accuracy

```
[244]: def compute_accuracy(model, features, targets):
           correct = 0.0  # Counter to track correct predictions

           # Iterate through each feature-target pair
           for x, y in zip(features, targets):
               prediction = model.forward(x)  # Get the model's prediction for the sample
               correct += int(prediction == y)  # Increment counter if prediction matches target

           # Calculate and return accuracy as the ratio of correct predictions
           return correct / len(targets)

       # Compute the training accuracy of the perceptron
       train_acc = compute_accuracy(ppn, features, targets)

       # Print the computed accuracy
       print("Model Accuracy:", train_acc)

       Model Accuracy: 0.9923076923076923
```

Visualizing the Decision Boundary

```python
[245]: def plot_boundary(model):
    # Extract weights and bias from the model
    w1, w2 = model.weights[0], model.weights[1]
    b = model.bias

    # Compute the x2 values for the minimum and maximum x1 values
    x1_min = -20  # Define the minimum x1 value
    x2_min = (-(w1 * x1_min) - b) / w2  # Compute corresponding x2 value

    x1_max = 20  # Define the maximum x1 value
    x2_max = (-(w1 * x1_max) - b) / w2  # Compute corresponding x2 value

    # Return the boundary line coordinates
    return x1_min, x1_max, x2_min, x2_max

# Get decision boundary coordinates
x1_min, x1_max, x2_min, x2_max = plot_boundary(ppn)

# Plot data points for Class 0
plt.plot(
    features[targets == 0, 0],  # Feature x1 values for Class 0
    features[targets == 0, 1],  # Feature x2 values for Class 0
    marker="D",  # Diamond marker for Class 0
    markersize=10,
    linestyle="",
    label="Class 0",  # Label for the legend
)

# Plot data points for Class 1
plt.plot(
    features[targets == 1, 0],  # Feature x1 values for Class 1
    features[targets == 1, 1],  # Feature x2 values for Class 1
    marker="^",  # Triangle marker for Class 1
    markersize=13,
    linestyle="",
    label="Class 1",  # Label for the legend
)

# Plot the decision boundary line
plt.plot([x1_min, x1_max], [x2_min, x2_max], color="k")  # Black line for boundary

# Add legend to the plot
plt.legend(loc=2)

# Set the x and y limits for the plot
plt.xlim([-2, 2])
plt.ylim([-2, 2])

# Label the axes
plt.xlabel("Feature $x_1$", fontsize=12)
plt.ylabel("Feature $x_2$", fontsize=12)

# Add a grid for better visualization
plt.grid()

# Display the plot
plt.show()
```



The **Perceptron** algorithm provides an intuitive and practical introduction to machine learning. Although limited in scope, it lays the groundwork for understanding more advanced neural networks and classification models. Implementing the Perceptron helps grasp essential concepts like weight updates, activation functions, and decision boundaries.