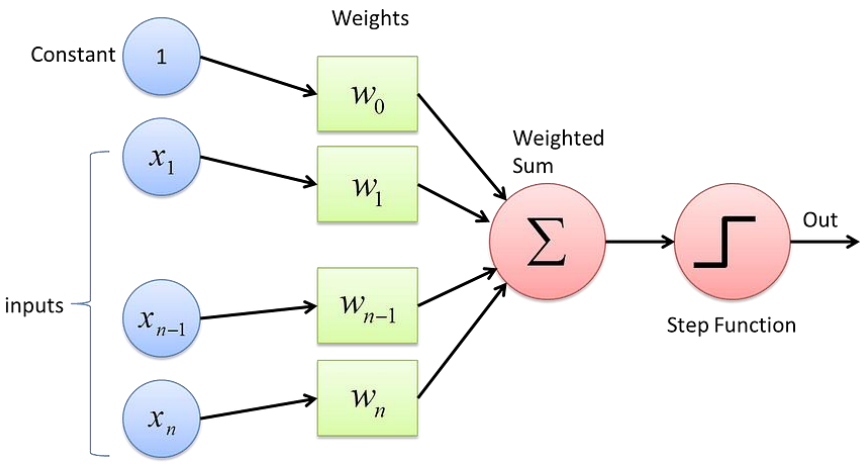
Tutorial 1:
Building a Perceptron

Objectives:

- Building Perceptron from Scratch
- How Weights and Bias get Updated
- Activation Function
- Data Loading
- Training & Testing

Lets build a Perceptron?



Step 1 — Perceptron Class

Follow these simple steps:

- Create a file called “Perceptron.py”
- Define a class called Perceptron inside

```python
import numpy as np

class Perceptron (object):

    def __init__(self, eta=0.01, n_iter=10):
        self.eta = eta
        self.n_iter = n_iter
```

To build a perceptron, we need 3 attributes:

- **η (eta)**: the learning rate is usually a small value between 0.0 and 1.0 which defines how quickly the model learns.
- **n_iter**: the number of iterations.
- **w_**: the weights. A weight defines the importance of the corresponding input value

Step 2 — Weighted sum

```python
def weighted_sum(self, X):
    return np.dot(X, self.w_[1:]) + self.w_[0]
# np.dot(X, self.w_[1:]) computes the dot product of the input features `X` and the model's weights (excluding the bias weight)
# self.w_[0] represents the bias term (also known as the intercept), which is added to the dot product
```

np.dot(X, w_) is simply:

$$\sum_{i=0}^{n} Xi.Wi$$

Step 3 — Step function

```python
def predict(self, X):
    return np.where(self.weighted_sum(X) >= 0.0, 1, -1)
```

The step function predict () returns:

- 1 — if the weighted sum is greater than 0.0
- -1 — Otherwise.

Step 4— Create the training method

An explanation will follow.

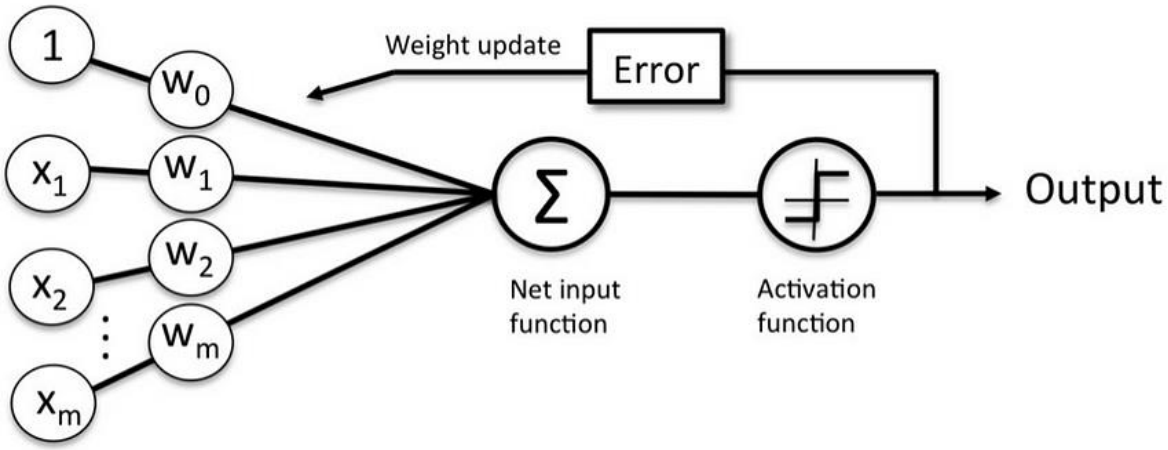
```python
def fit(self, X, y):
    # initializing the weights to 0
    self.w_ = np.zeros(1 + X.shape[1])  # Weights initialized to zeros, with one extra element for the bias
    self.errors_ = []  # List to track the number of errors in each iteration

    print("Weights:", self.w_)  # Print the initial weights for reference

    # training the model for 'n_iter' number of iterations
    for _ in range(self.n_iter):  # Loop for the specified number of iterations
        error = 0  # Initialize error counter for the current iteration

        # Loop through each input and corresponding target value
        for xi, y in zip(X, y):
            # 1. Calculate the predicted value (ŷ) using the current weights
            y_pred = self.predict(xi)  # Predict the output for the current input 'xi'

            # 2. Calculate the weight update
            # update = η * (y - ŷ) where 'eta' is the learning rate
            update = self.eta * (y - y_pred)  # Compute the update value based on prediction error

            # 3. Update the weights for each feature
            # Wi = Wi + Δ(Wi) where Δ(Wi) = η * (y - ŷ) * Xi
            self.w_[1:] = self.w_[1:] + update * xi  # Update the weights (excluding bias)
            print("Updated Weights:", self.w_[1:])  # Print updated weights for debugging

            # Update the bias (X0 = 1)
            self.w_[0] = self.w_[0] + update  # Update the bias term separately

            # Increment the error count if the model's prediction was incorrect
            error += int(update != 0.0)  # If update != 0, it means the prediction was wrong

        # Append the total number of errors in this iteration to the errors list
        self.errors_.append(error)  # Store the error count after each iteration
```

We initialize the weights to 0, then start our training.

For each input:

1. we calculate ŷ, the predicted output.
2. then we update the weights from W1 to Wn (remember W*o* is the bias).

We repeat this process n_iter times (n_iter is the number of iterations).



Weights update in the perceptron

How are the weights updated?

Every weight W*i* is updated by Δ(W*i*).

W*i* = W*i* + *Δ(*W*i)*

Where:

*Δ(*W*i)* = *η (y — ŷ)* * X*i*

Don't panic, it's easier than it seems:

- W*i* is i-th weight.
- Δ(W*i*) is the update. It determines if the weight increases or decreases.
- η is the learning rate.
- y is the actual value.
- ŷ is the predicted value.
- X*i* is the i-th input.

Example

Get your pen, we are going to do some math.

η	y	ŷ	X*i*
0.01	1	1	0.5
0.01	-1	-1	1
0.01	1	-1	4
0.01	-1	1	2

Random data to calculate weights update

*Δ(*W*i)* = *η (y — ŷ)* * X*i*

When y is equal to ŷ, there is no update:

- Δ(W*1*) = 0.01 * (1 − 1) ∗ 0.5 = 0
- Δ(W2) = 0.01 * ((−1) − (−1)) ∗ 1 = 0

When y is different than ŷ, it means that there's an error and the weights must be updated by Δ(*W*i):

- Δ(W*3*) = 0.01 * (1 − (−1)) ∗ 4 = 0.08

- $\Delta(W4) = 0.01 * ((-1) - 1) * 2 = -0.04$

Whew!

We made it through to the end, let's test our Perceptron.

Step 5 : Loading Data:

The data is loaded from https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data it represents the Iris dataset. It is a famous dataset used for classification tasks. It contains 150 samples of iris flowers, each described by four features (sepal length, sepal width, petal length, and petal width) and a target label that indicates the species of the flower (either *Iris-setosa*, *Iris-versicolor*, or *Iris-virginica*).

```
import pandas as pd  # Importing the pandas library for data manipulation and analysis
from sklearn.utils import shuffle  # Importing the shuffle function from sklearn to shuffle the data
import numpy as np  # Importing the numpy library for numerical operations (used indirectly here)

# Reading the Iris dataset from the given URL using pandas' read_csv function
# The 'header=None' parameter is used because the dataset doesn't have a header row, so we treat the first row as data
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data', header=None)

# Shuffling the rows of the dataset to randomize the order of the data
df = shuffle(df)

# Displaying the first 5 rows of the shuffled dataset using the head() function
df.head()
```

Output

	0	1	2	3	4
34	4.9	3.1	1.5	0.1	Iris-setosa
43	5.0	3.5	1.6	0.6	Iris-setosa
91	6.1	3.0	4.6	1.4	Iris-versicolor
88	5.6	3.0	4.1	1.3	Iris-versicolor
71	6.1	2.8	4.0	1.3	Iris-versicolor

Separating the data (X) from the labels (y)

```
# Extracting the first four columns (features) as a NumPy array
# These columns correspond to sepal length, sepal width, petal length, and petal width
X = df.iloc[:, 0:4].values

# Extracting the fifth column (target) as a NumPy array
# This column corresponds to the species of the flower (Iris-setosa, Iris-versicolor, Iris-virginica)
y = df.iloc[:, 4].values

# Print the first 5 rows of the feature matrix (X)
# Each row represents a flower's measurements (sepal length, sepal width, petal length, petal width)
print(X[0:5])

# Print the first 5 rows of the target array (y)
# Each value represents the species of the corresponding flower
print(y[0:5])
```

Splitting the data into train and test set

```
from sklearn.model_selection import train_test_split
# 75% for train and 25% for test
train_data, test_data, train_labels, test_labels = train_test_split( X, y, test_size=0.25)

# Encoding the labels: 1 for setosa, -1 otherwise
train_labels = np.where(train_labels == 'Iris-setosa', 1, -1)
test_labels = np.where(test_labels == 'Iris-setosa', 1, -1)

print('Train data:', train_data[0:2])
print('Train labels:', train_labels[0:2])

print('Test data:', test_data[0:2])
print('Test labels:', test_labels[0:2])
```

Output: **(The output might Appear different on your screens)**

```
Train data: [[6.3 2.9 5.6 1.8]
 [5.5 4.2 1.4 0.2]]
Train labels: [-1  1]
Test data: [[5.1 3.8 1.6 0.2]
 [6.7 3.  5.  1.7]]
Test labels: [ 1 -1]
```

Step 6: Training the perceptron:

```
from sklearn.linear_model import Perceptron
# Importing the Perceptron model from scikit-learn's linear_model module

perceptron = Perceptron(eta0=0.1, max_iter=10)
# Initializing the Perceptron model:
# - eta0=0.1: Learning rate, controlling how much the model adjusts the weights during training
# - max_iter=10: The maximum number of iterations (epochs) to run during the training process

perceptron.fit(train_data, train_labels)
# Training the Perceptron model using the training data (train_data) and labels (train_labels)
# The model will adjust the weights over a maximum of 10 iterations or until it converges
```

Step 7 : Making predictions on test data

```
# Using the trained perceptron model to predict labels for the test data
test_preds = perceptron.predict(test_data)
# The predict() method generates predictions based on the learned weights from the training process.
# It returns an array of predicted labels (1 for 'Iris-setosa', -1 for other species) for the test_data.
# test_preds stores the predicted labels.

# Printing the predicted labels for the test dataset
print(test_preds)
# This will print the array of predicted labels for each sample in the test dataset.
```

Step 8: Measuring Performances

```
from sklearn.metrics import accuracy_score
# Importing the accuracy_score function from sklearn's metrics module.
# This function is used to calculate the accuracy of the model's predictions.

y_preds = perceptron.predict(test_data)
# Using the trained perceptron model to make predictions on the test data.
# The predict() method returns an array of predicted labels for the test dataset (test_data).
# The predicted labels are stored in y_preds.

# Calculate the accuracy of the model by comparing predicted labels (y_preds) with actual test labels (test_labels)
accuracy = accuracy_score(y_preds, test_labels)
# The accuracy_score function compares the predicted labels (y_preds) with the actual labels (test_labels).
# It returns the proportion of correct predictions, which is the accuracy of the model.

# Print the accuracy, rounded to two decimal places, and multiplied by 100 to show it as a percentage.
print('Accuracy:', round(accuracy, 2) * 100, "%")
# The accuracy is rounded to 2 decimal places, then multiplied by 100 to convert it to percentage format for readability.
```

Tasks:

1. **Take manual input** for the four features (sepal length, sepal width, petal length, and petal width).
    - **Use the model to predict** whether it's Iris-setosa or not based on the manually provided values.
    - **Display the prediction** result.

2. Replace the Step Function with Sigmoid.
3. Replace the labels of 1,-1 with 1 & 0.