Introduction to Neural Networks and Deep Learning

(Practical Sec-01)

---

1. Introduction to Neural Networks

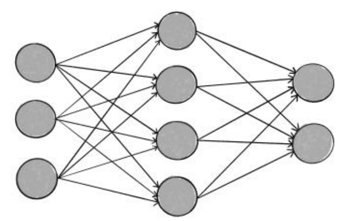
Basic concepts of neural networks.

What is a Neural Network?

- Inspired by the human brain.
- Composed of layers (input, hidden, and output).
- Learn patterns from data using weights and activation functions.

Structure of a Neural Network

- **Input Layer:** Receives input data.
- **Hidden Layers (Dense):** Extracts patterns & features.
- **Output Layer (Dense):** Produces final predictions.



Where are Neural Networks Used?

- Image recognition (e.g., face detection, medical imaging).
- Natural language processing (e.g., chatbots, language translation).
- Self-driving cars, fraud detection, etc.

Basic Structure of a Neural Network

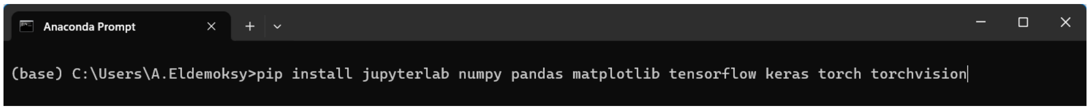
- **Neurons:** Mathematical functions that process inputs.
- **Weights & Biases:** Control the importance of inputs.
- **Activation Functions:** Transform input values (ReLU, Sigmoid, etc.).
- **Forward Propagation:** Computes output.
- **Backpropagation:** Adjusts weights to improve learning.

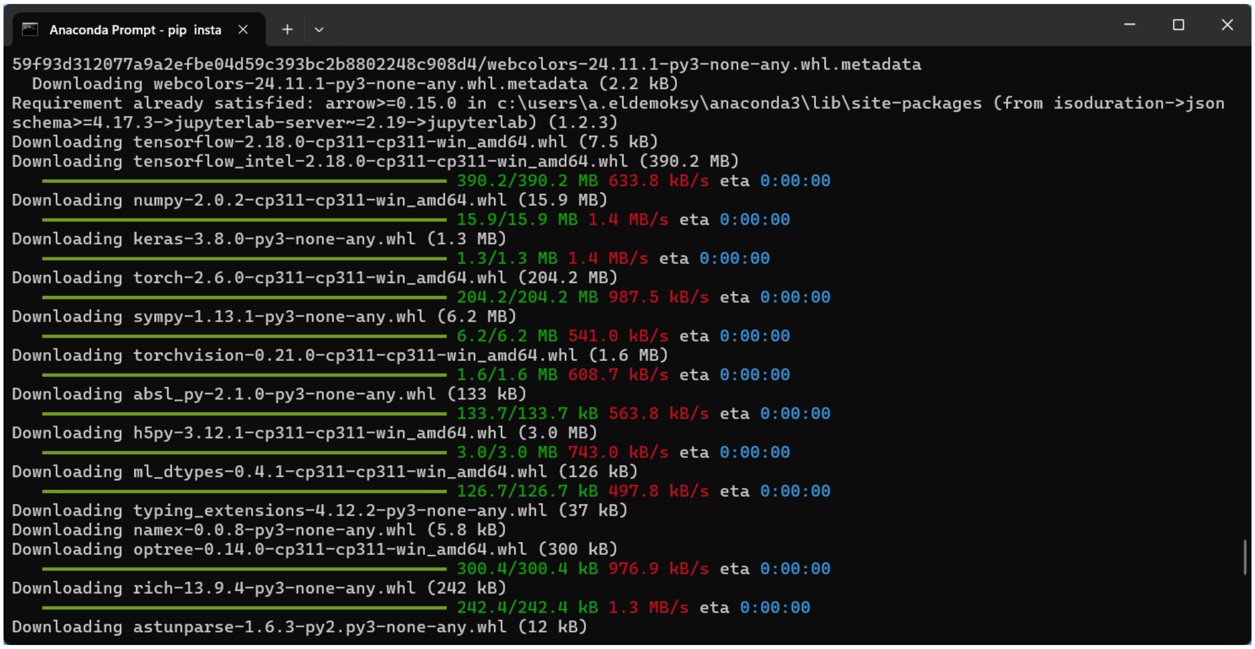
---

2. Setting Up the Practical Environment

Install Required Libraries

pip install jupyterlab numpy pandas matplotlib tensorflow keras torch torchvision



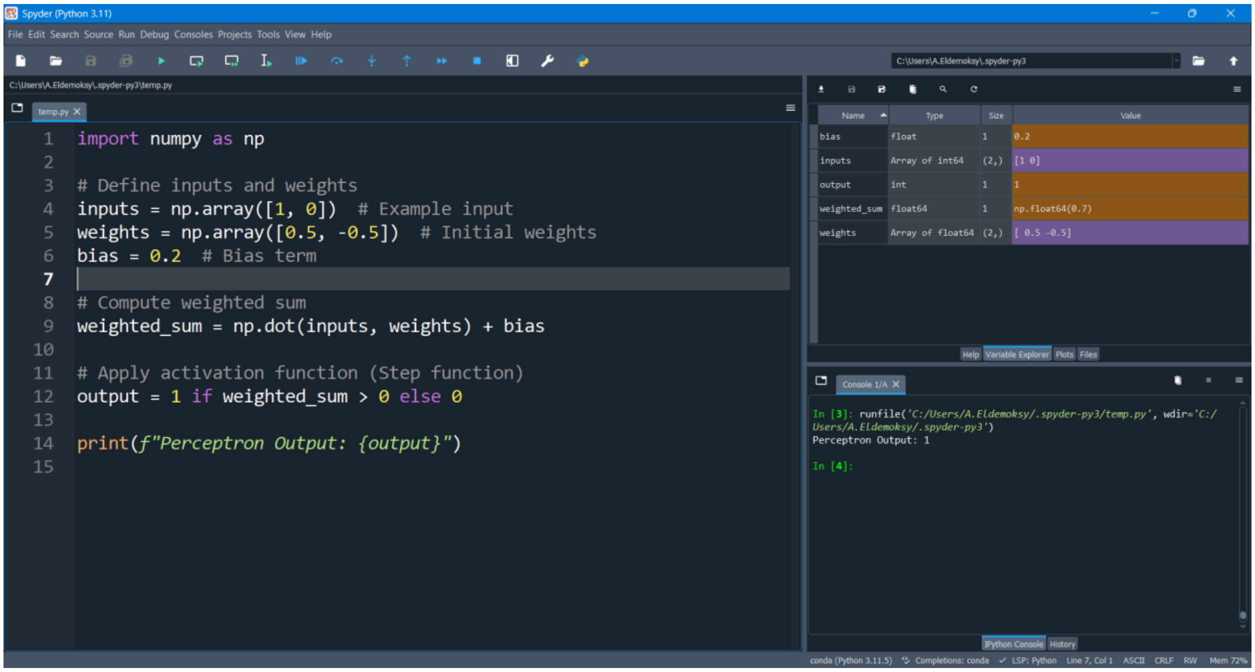


---

3. Implementing a Simple Perceptron

Example1: Implement a Perceptron using NumPy

```
import numpy as np

# Define inputs and weights

inputs = np.array([1, 0])  # Example input

weights = np.array([0.5, -0.5])  # Initial weights

bias = 0.2  # Bias term

# Compute weighted sum

weighted_sum = np.dot(inputs, weights) + bias

# Apply activation function (Step function)

output = 1 if weighted_sum > 0 else 0

print(f"Perceptron Output: {output}")
```



Key Takeaways:

- Adjusting weights and bias impacts learning.
- Activation functions determine neuron output.

What is Bias in Neural Networks?

Bias is an additional parameter in a **neural network** that helps adjust the output along with the weighted sum of inputs. It ensures that the model can **shift** the activation function, allowing it to learn patterns more effectively.

Why Do We Need Bias?

- Without bias, a neural network might be too rigid and unable to **fit complex data**.
- It allows the model to **generalize better** by shifting activation functions.
- Helps prevent **underfitting**, especially when data doesn't pass through the origin (0,0).

Python code for activation functions

```
import numpy as np

def binary_step(x):
  return np.where(x >= 0, 1, 0)


def linear(x):
  return x
print('binary_step',binary_step(0.5))


def sigmoid(x):
  return 1 / (1 + np.exp(-x))
print('Sigmoid',sigmoid(0))
```

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

```
def relu(x):
  return np.maximum(0, x)
print('relu',relu([-1, 0, 2]))


def tanh(x):
  return np.tanh(x)
print('tanh',tanh(0.2))
```

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

```
def softmax(x):
  exp_x = np.exp(x - np.max(x))  # Stability improvement
  return exp_x / exp_x.sum(axis=0)
print('softmax',softmax([2,1,0]))
```

$$f(x_i) = \frac{e^{x_i}}{\sum e^{x_j}}$$

Function	Output Range	Used For
Binary Step	{0,1}	Simple classification
Linear	(-∞, ∞)	Regression
Sigmoid	(0,1)	Binary classification
ReLU	[0,∞)	Deep learning (hidden layers)
Tanh	(-1,1)	RNNs (saturates less than sigmoid)
Softmax	(0,1) (sums to 1)	Multi-class classification

Binary Classification on a Linearly Separable Dataset

Here, we are going to process building, training, and evaluating a **Perceptron model** for binary classification using a synthetic, linearly separable dataset. It covers data preprocessing, model training, and performance evaluation. We will follow these steps:

1. Import Libraries
2. Generate Dataset using **make_blobs()**
3. Train-Test Split with **train_test_split()**
4. Scale Features using **StandardScaler()**
5. Initialize Perceptron with appropriate input size
6. Train the Model with **fit()** over 100 epochs
7. Predict on test data and evaluate accuracy by comparing predictions with actual labels
8. Visualize Results using a scatter plot

```
# Import the necessary library

import numpy as np

from sklearn.datasets import make_blobs

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

from sklearn.linear_model import Perceptron
```

```
# Generate a linearly separable dataset with two classes
X, y = make_blobs(n_samples=1000,
   n_features=2,
   centers=2,
   cluster_std=3,
   random_state=23)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X,
   y,
   test_size=0.2,
   random_state=23,
   shuffle=True
   )

# Scale the input features to have zero mean and unit variance
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Set the random seed legacy
np.random.seed(23)

# Initialize the Perceptron with the appropriate number of inputs
perceptron = Perceptron(max_iter=100, random_state=23)

# Train the Perceptron on the training data
```

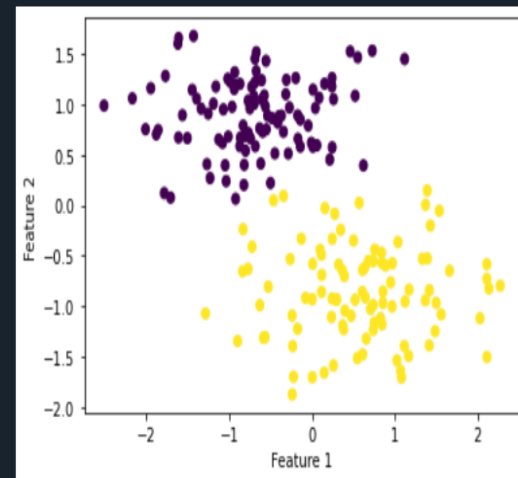
```
perceptron.fit(X_train, y_train)


# Prediction
pred = perceptron.predict(X_test)


# Test the accuracy of the trained Perceptron on the testing data
accuracy = np.mean(pred == y_test)
print("Accuracy:", accuracy)


# Plot the dataset
plt.scatter(X_test[:, 0], X_test[:, 1], c=pred)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```

\MET\2025\Term-2\NN&DL\Neural-Networks-and-Deep-Learning\Section02\Binary-Classification.py

Binary-Classification.py* | Implement-aPerceptron-using-NumPy.py | activation-function.py

```python
# Import the necessary library
import numpy as np
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import Perceptron
# Generate a linearly separable dataset with two classes
X, y = make_blobs(n_samples=1000,
    n_features=2,
    centers=2,
    cluster_std=3,
    random_state=23)
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X,
    y,
    test_size=0.2,
    random_state=23,
    shuffle=True
    )
# Scale the input features to have zero mean and unit variance
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
# Set the random seed legacy
np.random.seed(23)
# Initialize the Perceptron with the appropriate number of inputs
perceptron = Perceptron(max_iter=100, random_state=23)
# Train the Perceptron on the training data
perceptron.fit(X_train, y_train)
# Prediction
pred = perceptron.predict(X_test)
# Test the accuracy of the trained Perceptron on the testing data
accuracy = np.mean(pred == y_test)
print("Accuracy:", accuracy)
# Plot the dataset
plt.scatter(X_test[:, 0], X_test[:, 1], c=pred)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```

134 %



Help | Variable Explorer | Plots | Files

Console 1/A

```
   ...: print('softmax',softmax([2,1,0]))
softmax [0.66524096 0.24472847 0.09003057]

In [22]: runfile('D:/MET/2025/Term-2/NN&DL/Neural-Networks-and-Deep-Learning/Section02/Binary-Classification.py', wdir='D:/MET/2025/Term-2/NN&DL/Neural-Networks-and-Deep-Learning/Section02')
Accuracy: 0.98
```

Important

Figures are displayed in the Plots pane by default. To make them also appear inline in the console, you need to uncheck "Mute inline plotting" under the options menu of Plots.

```
In [23]:
```

IPython Console | History