## Feed-Forward Neural Network, FFNN

**Feed-Forward Neural Network (FFNN)**
A Feed-Forward Neural Network (FFNN) is the most basic type of Artificial Neural Network (ANN). Information flows only in one direction: **forward, from the input to the output, with no cycles or loops.**

**1. FFNN Components (Layers)**
FFNNs are made of three main types of layers:

- **Input Layer:**
- Receives the raw data from the outside world.
- Each neuron (node) represents one feature of your input data.
- **Example:** Pixel values for an image, sensor readings, or word embeddings for text.
- These neurons don't do any calculations; they just pass the data to the next layer.
-
- **Hidden Layer(s):**
- Located between the input and output layers.
- An FFNN can have one or many hidden layers. More layers make it a "Deep" Neural Network.
- Each neuron in a hidden layer:
- Receives values from the previous layer, each multiplied by a **weight**.
- Sums these weighted values (and adds a **bias**).
- Applies an **activation function** to this sum.
- Passes the result to the next layer.
-
- This is where the network learns and extracts complex patterns and features from the data.
-
- **Output Layer:**
- Produces the final result of the neural network.
- The number of neurons here depends on the problem:
- **Regression (predicting a continuous value):** Usually one neuron (e.g., predicting house prices).
- **Binary Classification (two categories):** Often one neuron outputting a probability (e.g., spam or not spam).
- **Multi-class Classification (many categories):** One neuron per category, outputting probabilities for each (e.g., classifying types of animals in an image).
-
- Uses a specific activation function suitable for the task (e.g., Sigmoid, Softmax, Linear).
-

## FFNN Advantages Vs Disadvantages

**Advantages:**

- **Easy to Implement:** Simple, foundational structure, making it easy to understand and build.
- **Versatile:** Can be applied to various problem types, including classification and regression.
- **Parallel Processing Friendly:** Calculations within each layer's neurons are independent, suitable for parallel computation.

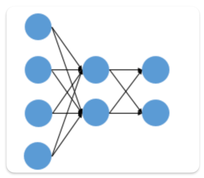
**Disadvantages:**

- **Poor at Sequential Data:** Struggles with data where order matters (e.g., time series, natural language), as it can't remember past information. (RNNs, LSTMs, Transformers were developed for this.)
- **Lacks Spatial Feature Recognition:** Not ideal for data with important spatial relationships (e.g., images), as it doesn't inherently understand adjacency. (CNNs were developed for this.)
- **High Parameter Count:** The number of weights can grow exponentially with more input features or hidden neurons, leading to high computational cost and risk of overfitting.

**6. Applications**

- **Simple Classification and Regression:** Predicting stock prices, spam detection.
- **Pattern Recognition:** Early stages of digit recognition, speech recognition.
- **Tabular Data Analysis:** Learning patterns and making predictions from structured table data.
- **Building Block for Complex Networks:** Often used as fundamental layers within more advanced architectures like CNNs and RNNs.

## FFNN Process



**Input Layer = Data**

**Hidden Layer**

The data received from the input layer is t**ransformed within each neuron by multiplying it with a 'weight', adding a 'bias', and then passing it through an 'activation function**'. These transformed values are then passed to the next hidden layer, and this process repeats, allowing complex features of the data to be extracted and combined. It's like ingredients undergoing multiple stages to become complex culinary components.

**1 Input Data:** First, your raw data becomes numbers (vectors).
**2 First Layer:** We multiply these numbers by W1 (weights) and add b1 (bias) to get Z1. Then, we put Z1 through an activation function (like ReLU) to get A1. This A1 is the output of the first hidden layer and becomes the input for the next layer.
**3 Next Layers (and Output Layer):** This process repeats. A1 is now multiplied by W2 and b2 is added to get Z2. This Z2 then goes through its own activation function (maybe Softmax if it's the final output).
**4 Many Layers:** If you have more layers, you just keep repeating this "multiply by weights, add bias, activate" step for each one, until you reach the final output.

**Hidden Layer**

The data received from the input layer is t**ransformed within each neuron by multiplying it with a 'weight', adding a 'bias', and then passing it through an 'activation function**'. These transformed values are then passed to the next hidden layer, and this process repeats, allowing complex features of the data to be extracted and combined. It's like ingredients undergoing multiple stages to become complex culinary components.

**Output Layer**

- Produces the final result of the neural network.
- The number of neurons here depends on the problem:
- **Regression (predicting a continuous value):** Usually one neuron (e.g., predicting house prices).
- **Binary Classification (two categories):** Often one neuron outputting a probability (e.g., spam or not spam).
- **Multi-class Classification (many categories):** One neuron per category, outputting probabilities for each (e.g., classifying types of animals in an image).
-
- Uses a specific activation function suitable for the task (e.g., Sigmoid, Softmax, Linear).
-

**W1, W2:** These are the weight matrices. We initialize them with small random values using np.random.randn. This gives the learning process a starting point.

```
import numpy as np

# ----------------------------------------------------
# 1. Define Activation Functions
# ----------------------------------------------------
def relu(x):
    """ReLU (Rectified Linear Unit) activation function.
    Returns x if x > 0, otherwise returns 0.
    """
    return np.maximum(0, x)

def sigmoid(x):
    """Sigmoid activation function.
    Squashes values between 0 and 1. Often used in output layer for binary classification.
    """
    return 1 / (1 + np.exp(-x))

def softmax(x):
    """Softmax activation function.
    Converts a vector of numbers into a probability distribution.
    Often used in the output layer for multi-class classification.
    """
    # Subtracting the maximum for numerical stability
    exp_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
    return exp_x / np.sum(exp_x, axis=-1, keepdims=True)

# ----------------------------------------------------
# 2. Define a Simple FFNN Class
# ----------------------------------------------------
class SimpleFFNN:
    def __init__(self, input_size, hidden_size, output_size):
        """
        Initializes the Feed-Forward Neural Network.

        Args:
            input_size (int): Number of input features (neurons in the input layer).
            hidden_size (int): Number of neurons in the hidden layer.
            output_size (int): Number of neurons in the output layer.
        """
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # Initialize Weights: Start with small random values (will be optimized during training)
        # np.random.randn samples from a standard normal distribution.
        # W1: Weights from input layer to hidden layer (input_size x hidden_size)
        self.W1 = np.random.randn(self.input_size, self.hidden_size) * 0.01
        # W2: Weights from hidden layer to output layer (hidden_size x output_size)
        self.W2 = np.random.randn(self.hidden_size, self.output_size) * 0.01

        # Initialize Biases: Start with zeros (will be optimized during training)
        # np.zeros returns an array filled with zeros.
        self.b1 = np.zeros((1, self.hidden_size)) # Biases for the hidden layer
        self.b2 = np.zeros((1, self.output_size)) # Biases for the output layer

    def forward(self, X):
        """
        Performs the forward pass computation of the neural network.

        Args:
            X (np.ndarray): Input data, expected shape (batch_size, input_size).

        Returns:
            np.ndarray: The network's predictions (output of the output layer).
        """
        # ----------------------------------------------------
        # 1. Input Layer -> Hidden Layer Calculation
        # ----------------------------------------------------
        # Weighted Sum (Z1)
        # X shape: (batch_size, input_size)
        # self.W1 shape: (input_size, hidden_size)
        # Z1 shape: (batch_size, hidden_size)
        Z1 = np.dot(X, self.W1) + self.b1

        # Apply Activation Function (A1)
        A1 = relu(Z1) # Using ReLU as the activation function for the hidden layer

        # ----------------------------------------------------
        # 2. Hidden Layer -> Output Layer Calculation
        # ----------------------------------------------------
        # Weighted Sum (Z2)
        # A1 shape: (batch_size, hidden_size)
        # self.W2 shape: (hidden_size, output_size)
        # Z2 shape: (batch_size, output_size)
        Z2 = np.dot(A1, self.W2) + self.b2

        # Apply Activation Function (output) - depends on the problem type
        # Assuming a multi-class classification problem, using Softmax
        output = softmax(Z2)

        return output

# ----------------------------------------------------
# 3. Example Usage of the Model
# ----------------------------------------------------
if __name__ == "__main__":
    # Model parameters setup
    input_dim = 4    # Number of input features (e.g., petal length, width)
    hidden_dim = 5   # Number of neurons in the hidden layer
    output_dim = 3   # Number of neurons in the output layer (e.g., 3 classes for classification)

    # Create an instance of the FFNN model
    model = SimpleFFNN(input_dim, hidden_dim, output_dim)

    print("--- Model Initialization Complete ---")
    print(f"Weights (W1) from Input to Hidden layer shape: {model.W1.shape}")
    print(f"Biases (b1) for Hidden layer shape: {model.b1.shape}")
    print(f"Weights (W2) from Hidden to Output layer shape: {model.W2.shape}")
    print(f"Biases (b2) for Output layer shape: {model.b2.shape}\n")

    # Generate sample input data (e.g., batch size 1, 4 input features)
    # Input data is typically shaped as (number_of_samples, number_of_features).
    input_data = np.array([[1.0, 2.5, 0.5, 3.0]]) # Single sample
    # input_data = np.array([[1.0, 2.5, 0.5, 3.0], [2.0, 1.0, 3.5, 0.8]]) # Multiple samples (batch)

    print(f"Input Data:\n{input_data}\n")
    print(f"Input Data Shape: {input_data.shape}\n")

    # Execute the forward pass
    predictions = model.forward(input_data)

    print("--- Forward Pass Results ---")
    print(f"Predictions (Probability Distribution):\n{predictions}")
    print(f"Predictions Shape: {predictions.shape}")

    # Find the index of the class with the highest probability
    predicted_class = np.argmax(predictions, axis=1)
    print(f"Predicted Class Index: {predicted_class[0]}") # For a single sample
```

- **Hidden Layers:** You usually use non-linear functions here. ReLU is very popular, but you can also use others like Leaky ReLU, ELU, GELU, or Swish. These functions help the network learn complex patterns. (Old ones like Sigmoid or Tanh were used, but ReLU-like functions are better now for training.)
- **Output Layer:** The activation function here depends on what kind of answer you want:
- **Regression (predicting a number):** You often use a **linear** function (or no activation function at all) so it can output any number.
- **Binary Classification (yes/no, 0/1):** Use **Sigmoid** to get a probability between 0 and 1.
- **Multi-class Classification (choosing from many categories):** Use **Softmax** to get probabilities for each category, adding up to 1.
-

**In short:**

- You **can use different activation functions for different layers.** For example, ReLU for hidden layers and Softmax for the output layer.
- **All neurons within the same layer usually use the same activation function.**
- **Hidden layers need non-linear functions** to learn complex features.
- **The output layer's function is chosen to match the type of problem** you're solving (like probabilities or continuous values).