**Ghazi Ben Henia**

# Report of the AI project
# -Neural Networks-

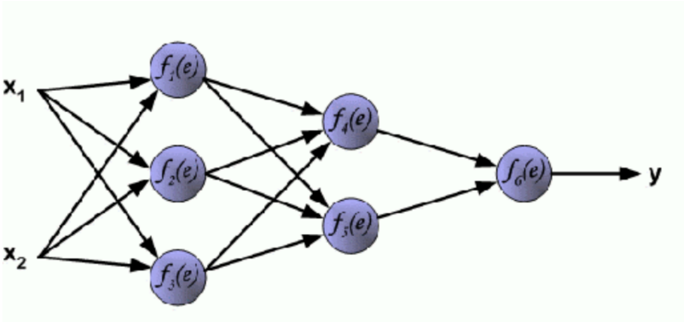
- ## I. Introduction :

In this project, our aim is to gain a foundational understanding of Neural Networks (NN) using Python by programming one from scratch. The task is to design a neural network architecture featuring two inputs, two outputs, and a hidden layer. Neural networks, inspired by the human brain, are potent computational models. Through this exercise, we'll explore core NN concepts, including feedforward and backpropagation algorithms, activation functions, and network structure. By constructing a NN from scratch, we'll deepen our grasp of its inner workings, setting the stage for further exploration in artificial intelligence and machine learning.

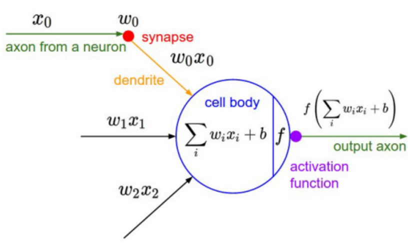
These are the steps we followed:
1. Begin by implementing a Python function that conducts a forward pass of the network, starting with an initial guess of the weights.
2. Within the same function, incorporate the backpropagation process of the network. Validate the functionality by testing it with a dataset containing two labeled samples.
3. Use the same dataset to execute the network (both forward and backward passes) repeatedly to train it across multiple iterations.
4. Train and evaluate the network using a larger dataset provided in the files NNTraining_data.csv and NNTest_data.csv.

## 1) Neural Networks:

Neural networks are computational models inspired by the structure and functioning of the human brain. They consist of interconnected nodes called neurons, organized into layers. Each neuron receives input signals, processes them using weighted connections, and produces an output signal. Neurons within a neural network work collectively to learn complex patterns and relationships in data through a process called training, where the network adjusts its weights based on input-output pairs. Neural networks have found widespread applications in various domains, including image recognition, natural language processing, and predictive analytics, due to their ability to learn and generalize from data.




https://giphy.com/gifs/neural-networks-4LiMmbAcvgTQs



# 1) Preparing the data:

We load the datasets 'NNTraining_data.csv' and 'NNTest_data.csv' into Pandas DataFrames. Initially, the datasets are displayed to understand its structures.

```
[1] import numpy as np
    import matplotlib.pyplot as plt
    import pandas as pd
```

```
[2] from google.colab import drive
    drive.mount('/content/drive')

    Mounted at /content/drive
```

```
[3] %cd /content/drive/MyDrive/dataflower

    /content/drive/MyDrive/dataflower
```

```
NNTraining_data = pd.read_csv('/content/drive/MyDrive/NN_data/NNTraining_data.csv')
NNTest_data = pd.read_csv('/content/drive/MyDrive/NN_data/NNTest_data.csv')
```

```
[5] NNTraining_data.head()
```

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 3.682795 | 4.540686 | 0.119606 | 0.880394 |
| 1 | -4.935752 | -3.951901 | 0.852986 | 0.147014 |
| 2 | 4.938425 | -1.880418 | 0.880796 | 0.119204 |
| 3 | 3.824091 | -3.374826 | 0.880797 | 0.119203 |
| 4 | 4.849526 | 4.542527 | 0.123314 | 0.876686 |

```
[6] NNTest_data.head()
```

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1.287235 | 3.094796 | 0.119462 | 0.880538 |
| 1 | -1.900905 | -2.726798 | 0.878145 | 0.121855 |
| 2 | 1.850491 | -3.554629 | 0.880797 | 0.119203 |
| 3 | 1.475565 | -3.215275 | 0.880796 | 0.119204 |
| 4 | 0.317844 | 2.789317 | 0.119299 | 0.880701 |

## ● II. Programming the network :

### 1)Forward pass:

The forward pass is the process of propagating input data through the network's layers to generate an output prediction.

To begin, I established a function to compute the sigmoid output σ(z), serving as the network's activation function. Subsequently, I crafted a forward pass function, which accepts input vectors x, target vectors t, and weight vectors w, yielding the loss L (squared error between predicted output y and target t). Upon initializing the network with prescribed weights and applying the forward pass function to inputs x1 and x2, we ascertain losses of 0.245 and 0.330, respectively.

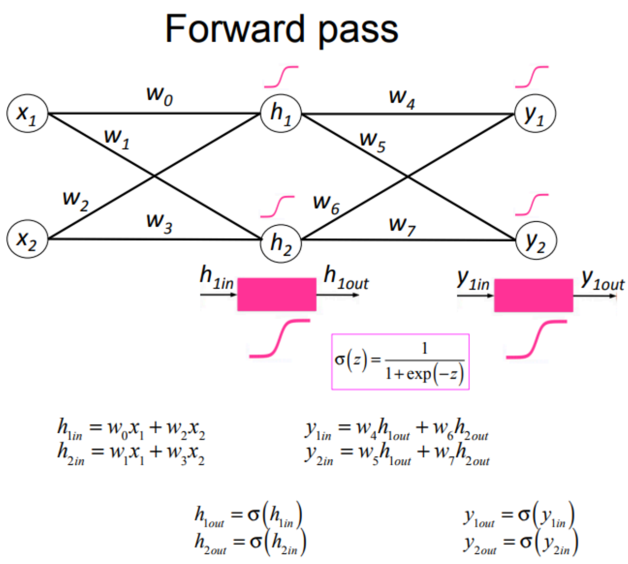
2.1 Forward pass function

```
[18] # Sample 1
     x1 = np.array([[2], [1]])
     t1 = np.array([[1], [0]])
     # Sample 2
     x2 = np.array([[-1], [3]])
     t2 = np.array([[0], [1]])
     # Weights
     W = np.array([[2], [-3], [-3], [4], [1], [-1], [0.25], [2]])
```

```
[19] # Sigmoid function
     def sigmoid(x):
         return 1 / (1 + np.exp(-x))
```

```
[9]  def forward_pass(x, t, w):

         h1 = w[0][0]*x[0][0]+w[2][0]*x[1][0]
         h2 = w[1][0]*x[0][0]+w[3][0]*x[1][0]


         h = np.array([[h1], [h2]])
         h = sigmoid(h)


         y1 = w[4][0]*h[0][0]+w[6][0]*h[1][0]
         y2 = w[5][0]*h[0][0]+w[7][0]*h[1][0]


         y = np.array([[y1], [y2]])
         y = sigmoid(y)


         # Squared error loss
         loss = np.sum((y - t) ** 2)
         return loss
```

```
[41] # Perform forward pass
     loss1 = forward_pass(x1, t1, W)
     loss2 = forward_pass(x2, t2, W)
     # Print results
     print("Loss for Sample 1: L =", round(loss1,3))
     print("Loss for Sample 2: L =", round(loss2,3))

     Loss for Sample 1: L = 0.245
     Loss for Sample 2: L = 0.33
```



## 2)Backpropagation:

Using the same function, I incorporated the backpropagation process into the network. In addition to returning the loss L, the function now also provides its gradient ∇wL. I also created a sigmoid derivative function. Upon evaluating the function with the same samples, we obtained gradients of [-0.12, 0.07, -0.06, 0.03, -0.1, 0.13, -0.02, 0.02] for x1 and [-0.0, -0.0, 0.0, 0.0, 0.0, -0.0, 0.277, -0.025] for x2. Furthermore, I adjusted the function to accommodate multiple samples simultaneously because it is better to pass the whole batch to the network than to use a loop, which will prove beneficial during the network's training phase.

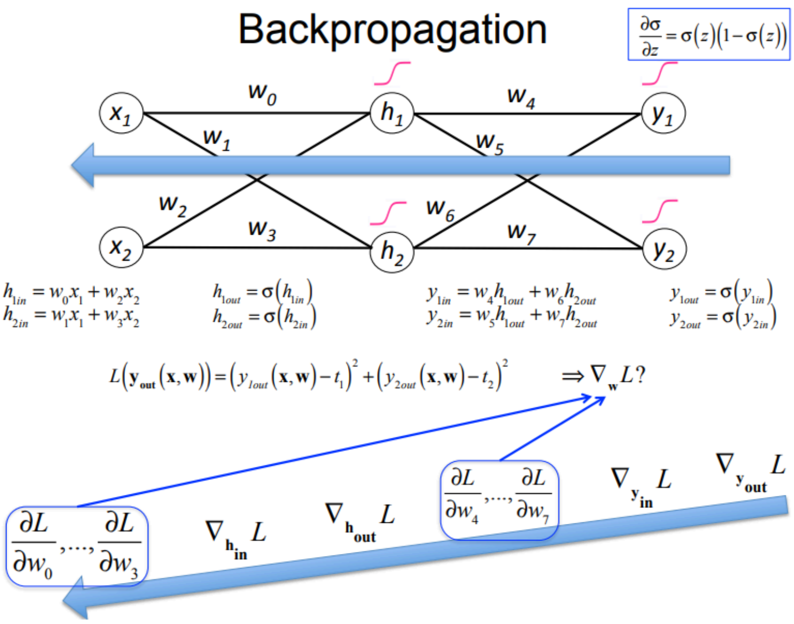
2.2 Implementing the backpropagation of the network

```
[10] def sigmoid_derivative(x):
         return x * (1 - x)
```

```
[11] def forward_backward_pass(x, t, w):
         # Forward pass
         h1 = w[0][0]*x[0][0] + w[2][0]*x[1][0]
         h2 = w[1][0]*x[0][0] + w[3][0]*x[1][0]

         h = np.array([[h1], [h2]])
         h = sigmoid(h)

         y1 = w[4][0]*h[0][0] + w[6][0]*h[1][0]
         y2 = w[5][0]*h[0][0] + w[7][0]*h[1][0]

         y = np.array([[y1], [y2]])
         y = sigmoid(y)

         # Squared error loss
         loss = np.sum((y - t) ** 2)

         # Backward pass
         delta_y = 2 * (y - t) * sigmoid_derivative(y)

         grad_w = np.zeros_like(w)

         grad_w[4][0] = delta_y[0][0] * h[0][0]
         grad_w[5][0] = delta_y[1][0] * h[0][0]
         grad_w[6][0] = delta_y[0][0] * h[1][0]
         grad_w[7][0] = delta_y[1][0] * h[1][0]

         delta_h = np.zeros_like(h)
         delta_h[0][0] = delta_y[0][0] * w[4][0] + delta_y[1][0] * w[5][0]
         delta_h[1][0] = delta_y[0][0] * w[6][0] + delta_y[1][0] * w[7][0]
         delta_h *= sigmoid_derivative(h)

         grad_w[0][0] = delta_h[0][0] * x[0][0]
         grad_w[1][0] = delta_h[1][0] * x[0][0]
         grad_w[2][0] = delta_h[0][0] * x[1][0]
         grad_w[3][0] = delta_h[1][0] * x[1][0]

         return loss, grad_w
```

```
[40] W = np.array([[2], [-3], [-3], [4], [1], [-1], [0.25], [2]])

     # Perform forward and backward pass
     final_loss1, updated_weights1 = forward_backward_pass(x1, t1, W)
     final_loss2, updated_weights2 = forward_backward_pass(x2, t2, W)
     # Results
     print("Final Loss:", round(final_loss1,3))
     print("Updated Weights:\n",  [[round(val, 2) for val in row] for row in updated_weights1.T][0])


     print("Final Loss:", round(final_loss2,3))
     print("Updated Weights:\n", [[round(val, 3) for val in row] for row in updated_weights2.T][0])
```

```
Final Loss: 0.245
Updated Weights:
 [-0.12, 0.07, -0.06, 0.03, -0.1, 0.13, -0.02, 0.02]
Final Loss: 0.33
Updated Weights:
 [-0.0, -0.0, 0.0, 0.0, 0.0, -0.0, 0.277, -0.025]
```

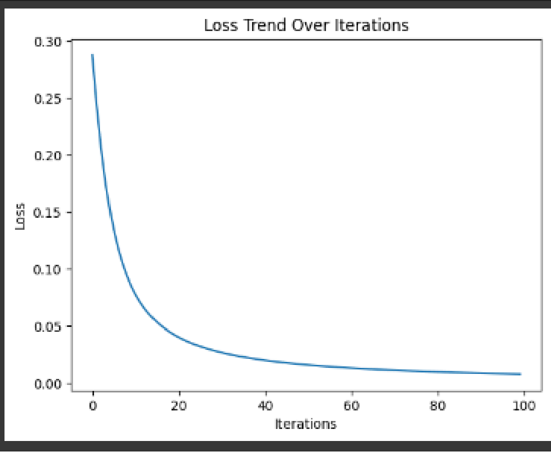


- ## III. Training and testing the network :

### 1)Training and testing on three samples:

Using the identical dataset, we execute the network (both forward and backward passes) to train it until the loss function has decreased sufficiently. I manually adjust two hyperparameters—the step size ρ (Learning rate) and the number of iterations N—until a satisfactory trend is observed. Following each iteration, a list is maintained to store the loss L, facilitating the plotting of its trend over the course of the iterations. Then I Tested it with a new labeled sample (x3, t3).

```python
# Initializing weights
W = np.array([[2], [-3], [-3], [4], [1], [-1], [0.25], [2]])


# Hyperparameters
learning_rate = 1.5
num_iterations = 100

losses = []

for iteration in range(num_iterations):
    total_loss = 0
    total_grad_w = np.zeros_like(W)

    # Iterate over individual samples
    for x_sample, t_sample in zip([x1, x2], [t1, t2]):
        # Forward and backward pass for the sample
        loss, grad_w = forward_backward_pass(x_sample, t_sample, W)

        total_loss += loss
        total_grad_w += grad_w

    # Updating the weights using the accumulated gradients
    W -= learning_rate * (total_grad_w / 2)

    # Calculating average loss for the iteration
    avg_loss = total_loss / 2
    losses.append(avg_loss)

# Plotting the loss trend
plt.plot(losses)
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss Trend Over Iterations')
plt.show()
```




```python
# New Sample 3
x3 = np.array([[1], [4]])
t3 = np.array([[1], [0]])

loss3, grad_w3 = forward_backward_pass(x3, t3, W)

print("Loss for Sample 3: L =", loss3)
```

```
Loss for Sample 3: L = 1.7596489314534582
```

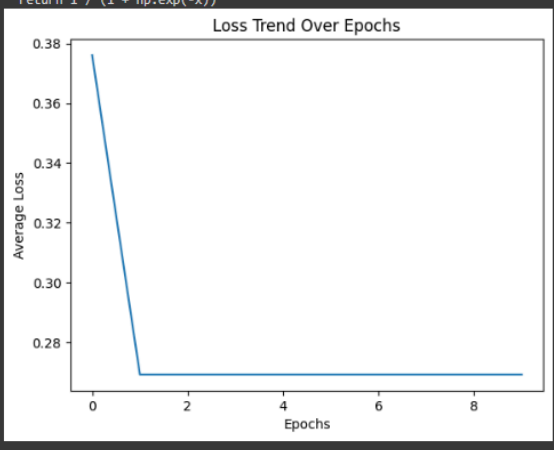
## 2)Training and testing on a large dataset:

In this phase, I'll conduct training of the network using the supplied dataset "NNTraining_data.csv", commencing from the initial weights. Following each epoch, which represents each instance where all batches have undergone forward propagation, the loss L will be stored to facilitate plotting its trend post-training. This iteration necessitates manual tuning of three hyperparameters: the step size ρ (Learning rate), the batch size m, and the number of epochs N.

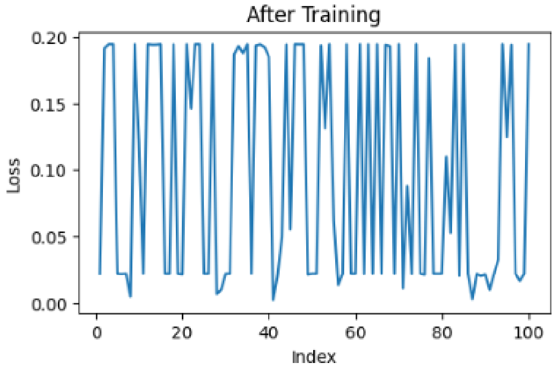
```python
# Initializing weights
W = np.array([[2], [-3], [-3], [4], [1], [-1], [0.25], [2]])

# Hyperparameters
learning_rate = 0.01
batch_size = 90
num_epochs = 10

# Storing losses for plotting at the end
epoch_losses = []

# Converting x1, x2, t1, t2 to numpy arrays
X1, X2, T1, T2 = np.array(X1), np.array(X2), np.array(T1), np.array(T2)

# Iterate over epochs
for epoch in range(num_epochs):
    epoch_loss = 0

    # Iterating over batches
    for i in range(0, len(X1), batch_size):
        # Extract batches
        x1_batch, x2_batch = X1[i:i+batch_size], X2[i:i+batch_size]
        t1_batch, t2_batch = T1[i:i+batch_size], T2[i:i+batch_size]

        # Reshaping batches to make them 2D
        x1_batch, x2_batch = x1_batch.reshape(-1, 1), x2_batch.reshape(-1, 1)
        t1_batch, t2_batch = t1_batch.reshape(-1, 1), t2_batch.reshape(-1, 1)

        x_batch = np.concatenate((x1_batch, x2_batch), axis=1)
        t_batch = np.concatenate((t1_batch, t2_batch), axis=1)

        # Forward and backward pass for the batch
        batch_loss, batch_grad_w = Modified_forward_backward_pass(x_batch, t_batch, W)

        # Accumulating loss for the epoch
        epoch_loss += batch_loss

        # Accumulating gradients for the epoch
        total_grad_w += batch_grad_w

    # Updating the weights using the accumulated gradients
    W -= learning_rate * total_grad_w / len(range(0, len(x1), batch_size))

    # Calculating average loss for the epoch
    avg_epoch_loss = epoch_loss / len(range(0, len(x1), batch_size))
    epoch_losses.append(avg_epoch_loss)
```

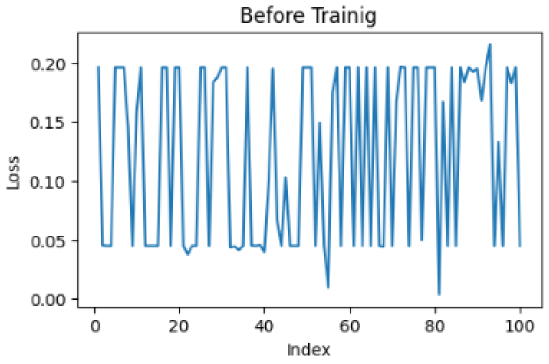
```python
# Plotting the loss trend
plt.plot(epoch_losses)
plt.xlabel('Epochs')
plt.ylabel('Average Loss')
plt.title('Loss Trend Over Epochs')
plt.show()
```

```
<ipython-input-8-113d8bc4b92c>:3: RuntimeWarning: overflow encountered in exp
  return 1 / (1 + np.exp(-x))
```



To evaluate the trained network's ability to generalize to new data, we will use the dataset "NNTest_data.csv".We will compare the loss before training, assessed using the forward_pass() function, with the loss after training, determined using the Modified_forward_backward_pass() function, by plotting them for comparison.




We can notice that the loss didn't change much after training. It is possible that the hyperparameters were not tuned optimally, leading to suboptimal learning rates or batch sizes that hindered significant improvement in the loss.

- ## IV. Conclusion :

In conclusion, this lab has provided a comprehensive exploration of building and training neural networks from scratch using Python. By implementing forward and backward propagation algorithms, we gained insights into the fundamental principles underlying neural network training. Through manual tuning of hyperparameters such as learning rate, batch size, and number of epochs, we explored their impact on training dynamics and model performance. Additionally, the assessment of model generalization using separate test data highlighted the importance of parameter optimization in achieving effective neural network training.