Introduction

Goal: train a single-layer perceptron such that it can draw a decision boundary that separates (1,1) from (1,0), (0,0), and (0,1). Check AND_gate_single_perceptron_CE for full code.

A perceptron functions as a single neuron that takes inputs $\rightarrow$ computes a weighted sum $\rightarrow$ applies an activation (like sigmoid) $\rightarrow$ outputs a prediction

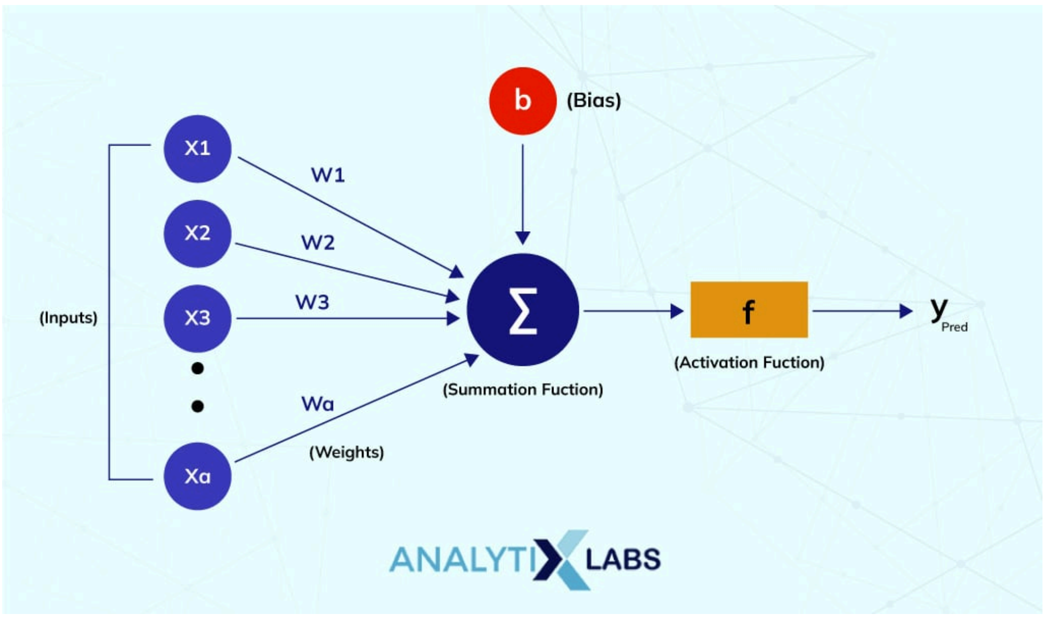


Figure 1: Source: https://www.analytixlabs.co.in/blog/what-is-perceptron/

The inputs and expected output are as follows:

x1	x2	AND
0	0	0
0	1	0
1	0	0
1	1	1

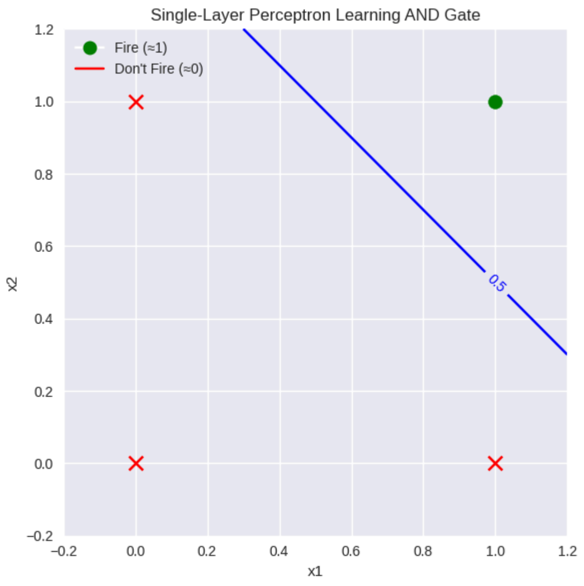


Figure 2: Decision boundary between the four coordinates

Refresher: Gradient Descent

In gradient descent we calculate the gradient of the loss function at the particular weights and bias, and take steps downwards to find the minimum of the loss function.
For the weights, we want to find out how much changing each w_i affects the overall loss L, i.e. find $\frac{\partial L}{\partial w_i}$. If $\frac{\partial L}{\partial w_i}$ is positive, we know we can continue taking steps downwards, as the gradient is positive. Conversely, if $\frac{\partial L}{\partial w_i}$ is negative, we need to take steps upwards.
We need to do the same thing for the bias, as $\nabla L = \left(\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \frac{\partial L}{\partial b}\right)$.

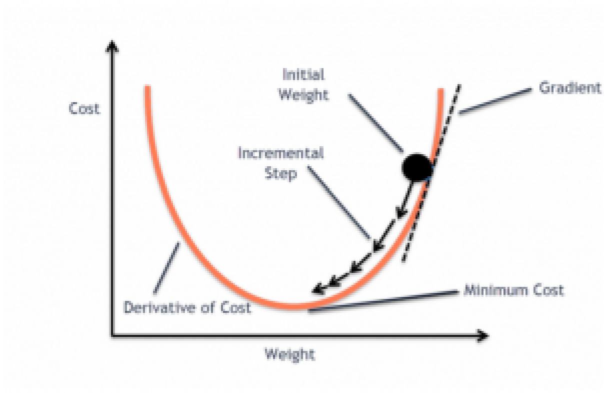


Figure 3: Source: https://vitalflux.com/gradient-descent-explained-simply-with-examples/ (content not sourced from this link, just the diagram)

Refresher : Cross Entropy Loss

The loss function used is the binary cross-entropy loss, since the AND gate is a classification task with two classes (True or False).

$$L(y, \hat{y}) \ = \ -(y\log(\hat{y}) \ + \ (1-y)\log(1-\hat{y}))$$

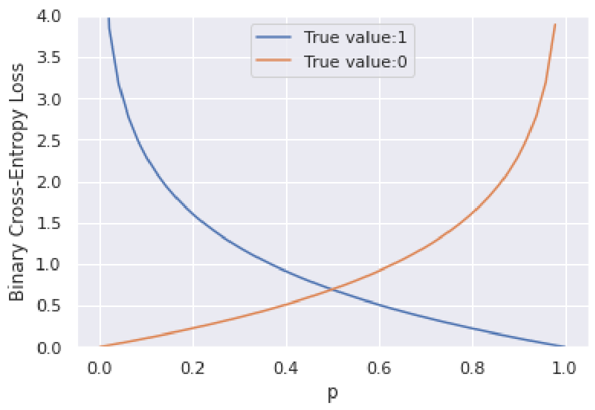
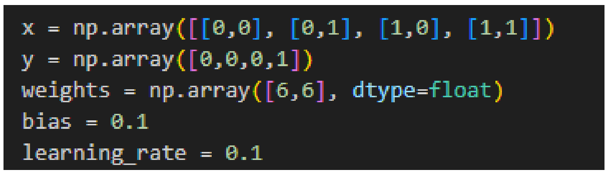


Figure 4: Source https://www.pinecone.io/learn/cross-entropy-loss/. Material in site not referenced, just the graph

When y is 1, then $L \ = \ -\log(\hat{y})$. When y is 0, then $L \ = \ -\log(1-\hat{y})$. Meaning, if y is 1 but $\hat{y}$ is 0 this will be heavily penalised (have a look at the curve for True value:1). Same heavy penalisation if y is 0 but $\hat{y}$ is 1.

1. Initialising Variables

$$x = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix},\ y = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix},\ b = 0.1,\ w_1 = 6,\ w_2 = 6,\ \ \eta = 0.1$$



```python
x = np.array([[0,0], [0,1], [1,0], [1,1]])
y = np.array([0,0,0,1])
weights = np.array([6,6], dtype=float)
bias = 0.1
learning_rate = 0.1
```

2. Calculating $z = w_1x_1 + w_2x_2 + b$

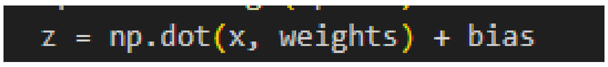
Multiplying x_1 and x_2 by the weights w_1 and w_2

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 6 \\ 6 \end{bmatrix} = \begin{bmatrix} 0 \\ 6 \\ 6 \\ 12 \end{bmatrix} \qquad (1)$$

adding the bias

$$\begin{bmatrix} 0 \\ 6 \\ 6 \\ 12 \end{bmatrix} + \begin{bmatrix} 0.1 \\ 0.1 \\ 0.1 \\ 0.1 \end{bmatrix} = \begin{bmatrix} 0.1 \\ 6.1 \\ 6.1 \\ 12.1 \end{bmatrix}$$

we get $z = w_1x_1 + w_2x_2 + b$

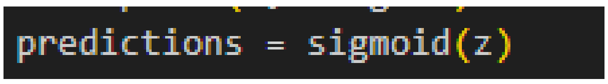


```python
z = np.dot(x, weights) + bias
```

3. Calculating $z = w_1x_1 + w_2x_2 + b$

Passing through a sigmoid

$\hat{y} = \sigma(z)$ where $\sigma(x) = \dfrac{1}{1 + e^{-x}}$



```python
predictions = sigmoid(z)
```

$$\sigma(z) = \begin{bmatrix} 0.52498 \\ 0.99776 \\ 0.99776 \\ 0.9999945 \end{bmatrix}$$

$$\text{predictions} = \begin{bmatrix} 0.52498 \\ 0.99776 \\ 0.99776 \\ 0.9999945 \end{bmatrix}$$

4. Computing $\frac{\partial L}{\partial w_i}$

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_i}$$

$$\frac{\partial L}{\partial w_i} = \frac{\hat{y} - y}{\hat{y}(1-y)} \cdot \hat{y}(1-\hat{y}) \cdot x_i$$

Overall, $\frac{\partial L}{\partial w_i} = (\hat{y} - y) \cdot x_i$

4.1 Computing $\frac{\partial L}{\partial \hat{y}}$

$\frac{\partial L}{\partial \hat{y}}$ measures how the loss changes based on the prediction $\hat{y}$. We take the derivative of the Cross Entropy Loss with respect to $\hat{y}$.

$$\frac{\partial L}{\partial \hat{y}} = \frac{\hat{y} - y}{\hat{y}(1-y)}$$

4.2 Computing $\frac{\partial \hat{y}}{\partial z}$

$\frac{\partial \hat{y}}{\partial z}$ measures how the prediction (predicted probability) changes based on the linear combination of weights and inputs. This is the derivative of the sigmoid function $\sigma'(z)$

$$\frac{\partial \hat{y}}{\partial z} = \sigma'(z) = \sigma(z)(1-\sigma(z)) = \hat{y}(1-\hat{y})$$

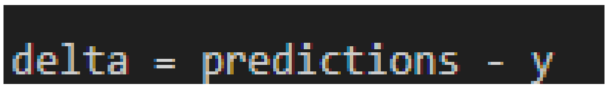
4.3 Computing $\frac{\partial z}{\partial w_i}$

For $\frac{\partial z}{\partial w_i}$, recall $z = w_1x_1 + w_2x_2 + b$

Hence $\frac{\partial z}{\partial w_i} = x_i$ by simple rules of differentiation (e.g. imagine differentiating based on w_1)

4.4 Computing $(\hat{y}-y)$

In code, separating $(\hat{y}-y)$ to be a delta δ allows us to more smoothly compute the gradient descent using numpy.



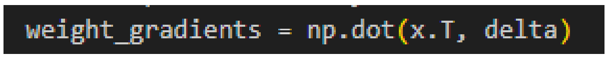
```
delta = predictions - y
```

$$\delta = (\hat{y}-y) = \begin{bmatrix} 0.52498 \\ 0.99776 \\ 0.99776 \\ 0.9999945 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.52498 \\ 0.99776 \\ 0.99776 \\ -0.0000055 \end{bmatrix}$$

$$\texttt{delta} = \begin{bmatrix} 0.52498 \\ 0.99776 \\ 0.99776 \\ -0.0000055 \end{bmatrix}$$

4.5 Computing $\begin{bmatrix} \frac{\partial \mathcal{L}}{\partial w_1} \\ \frac{\partial L}{\partial w_2} \end{bmatrix}$

Let's break down the next step in the code:



```
weight_gradients = np.dot(x.T, delta)
```

We are effectively doing:

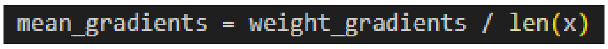
$$\begin{bmatrix} \sum x_1 \cdot (\hat{y} - y) \\ \sum x_2 \cdot (\hat{y} - y) \end{bmatrix}$$

$$x^T = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

$$x^T \cdot \delta = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0.52498 \\ 0.99776 \\ 0.99776 \\ -0.0000055 \end{bmatrix} = \begin{bmatrix} 0.9977545 \\ 0.9977545 \end{bmatrix}$$

$$\texttt{weight\_gradients} = \begin{bmatrix} 0.9977545 \\ 0.9977545 \end{bmatrix}$$

And we need to divide these sums by 4 because each x_i has 4 elements.



```
mean_gradients = weight_gradients / len(x)
```

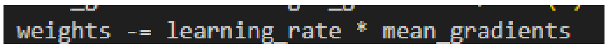
$$\begin{bmatrix} \frac{\sum x_1 \cdot (\hat{y} - y)}{4} \\ \frac{\sum x_2 \cdot (\hat{y} - y)}{4} \end{bmatrix} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial w_1} \\ \frac{\partial L}{\partial w_2} \end{bmatrix}$$

$$= \begin{bmatrix} 0.2494386 \\ 0.2494386 \end{bmatrix}$$

$$\texttt{mean\_gradients} = \begin{bmatrix} 0.2494386 \\ 0.2494386 \end{bmatrix}$$

5. Updating weights based on gradient

Update the weights based on the mean gradients (that are scaled by the learning rate):



```
weights -= learning_rate * mean_gradients
```

$$w_{new} = w - \eta \cdot \texttt{mean\_gradients}$$

$$= \begin{bmatrix} 6 \\ 6 \end{bmatrix} - 0.1 \cdot \begin{bmatrix} 0.2494386 \\ 0.2494386 \end{bmatrix} = \begin{bmatrix} 5.975056 \\ 5.975056 \end{bmatrix}$$

6. Updating bias based on gradient

We need to do the same thing for the bias, i.e. find out how much adjusting the bias affects the loss. Adjusting bias = moving line up and down.

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial b}$$

$$= \frac{\hat{y} - y}{\hat{y}(1-y)} \cdot \hat{y}(1-\hat{y}) \cdot 1$$

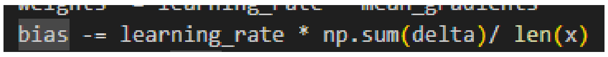
$$= \hat{y} - y$$

Since taking derivative of $w_1x_1 + w_2x_2 + b$ w.r.t b is clearly 1

$$\frac{\partial L}{\partial b} = \frac{1}{4}\sum_{i=1}^{4}(\hat{y}_i - y_i)$$

$$= 0.6301236$$

Updating the bias



```
bias -= learning_rate * np.sum(delta)/ len(x)
```

$$b_{new} = b - \eta \cdot \frac{\partial \mathcal{L}}{\partial b}$$

$$= 0.1 - (0.1 \times 0.6301236) = 0.0369876$$

7. Final answer after 1 epoch

$$\boxed{\begin{aligned} w_1 &= 5.9751 \\ w_2 &= 5.9751 \\ b &= 0.0370 \end{aligned}}$$

8. Repeat!

We repeat this for a number of epochs until the CE loss is at an acceptably small number. i.e. we keep adjusting the weights and bias until we reach the weights and bias that correspond to an appropriate decision boundary.

Run the code and change the number of epochs to see this in action.

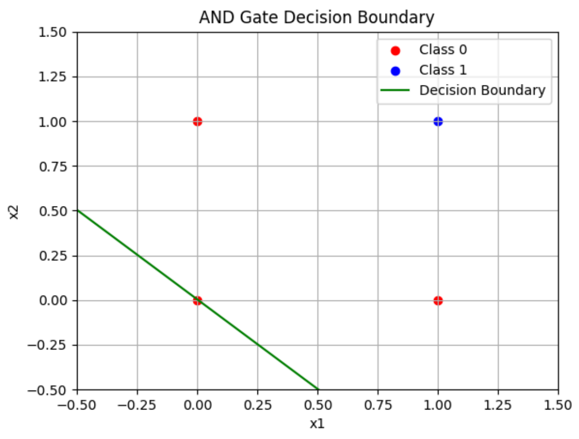


Figure 5: Epochs = 2

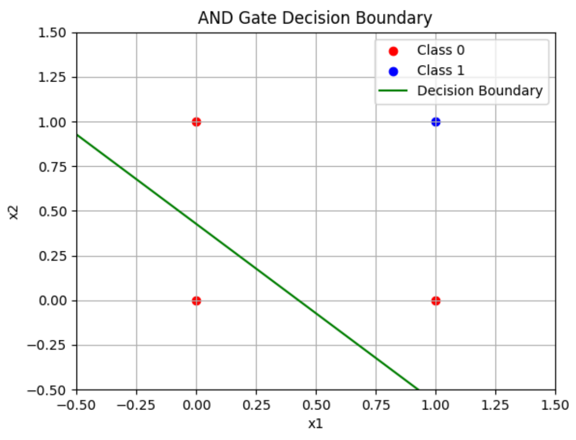


Figure 6: Epcohs = 40

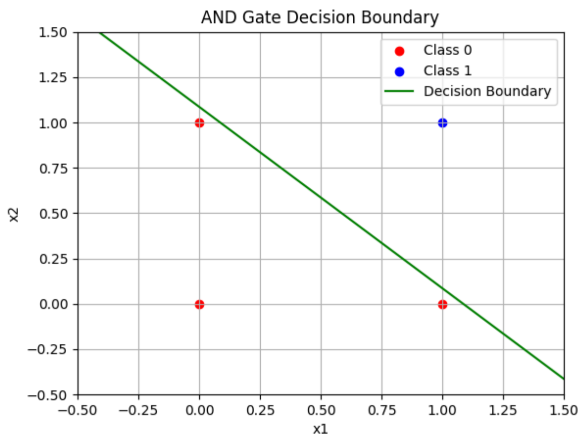


Figure 7: Epochs = 100

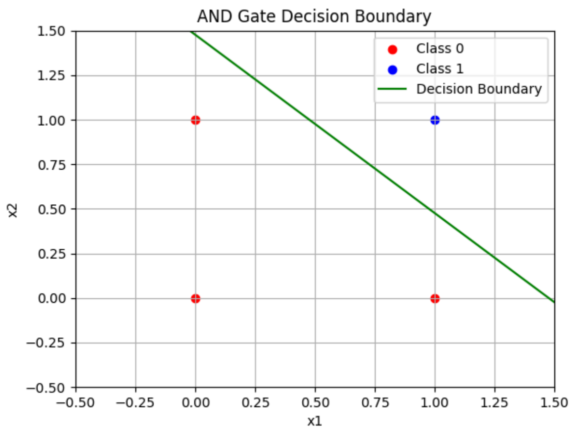


Figure 8: Epochs = 200

Note: Changing y to $\begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}$ changes the implementation to an OR gate.