Artificial Neural Networks

Dr. Ahmad Altarawneh

Department of Data Science

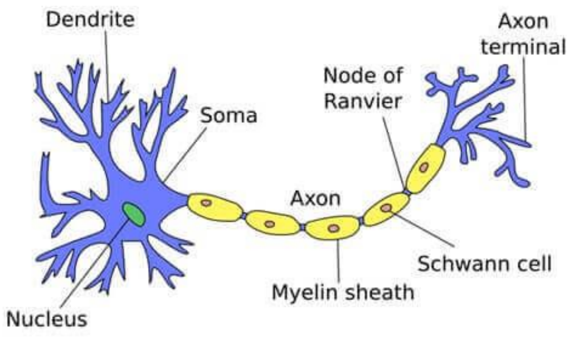
Faculty of information technology, Mutah University

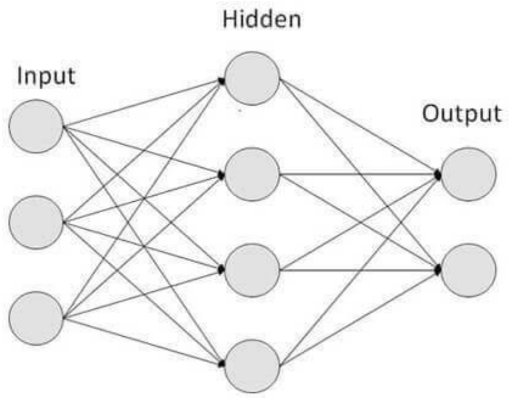
Outline

- What are neural networks
- Perceptron: recalling
- Gradient descent
- Multilayer perceptron (MLP)
  - Mathematical representation
  - Forward pass
  - Calculating the error: loss function
- Backpropagation with gradient descent
- MLP implementation in Python (from scratch example)
  - Linear and nonlinear decision boundary
  - MLP for regression problems

What are Neural Networks

- A Neural Network is a computational model inspired by the structure and functioning of the human brain.
- It consists of interconnected layers of nodes (called neurons), where each node processes input data, applies a mathematical function, and passes the output to the next layer.
- Nowadays, neural networks are used to identify patterns, make decisions, and solve complex problems in fields such as image recognition, natural language processing, and artificial intelligence.





Perceptron

- A perceptron is the simplest NN, as it consists of a single neuron that processes (linear combinations) the input data, passes it through an activation function (step function), and produces an output
  - The perceptron classifier provides a linear decision boundary
- let a perceptron have n input features, $\boldsymbol{x_1, x_2, \dots, x_n}$ and corresponding weights $\boldsymbol{w_1, w_2, \dots, w_n}$

The perceptron calculates the Weighted Sum (Linear Combination):

$$\boldsymbol{z = x_1 w_1, + x_2 w_1, + \cdots + x_n w_1 + b}$$

- The output of the linear combinations then enters to activation function, typically called a step function:

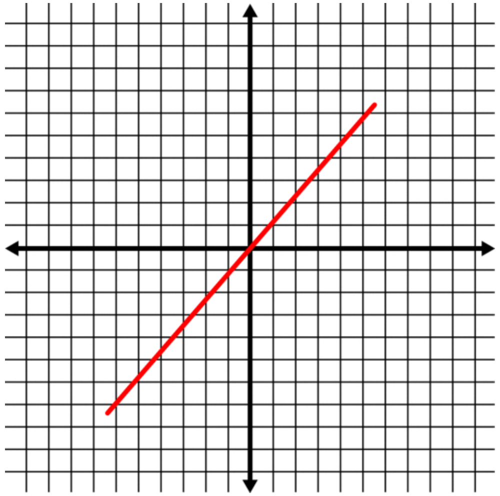
$$\hat{y} = f(z) = \begin{cases} 1, & if\ z \geq 0 \\ 0, & if\ z < 0 \end{cases}$$

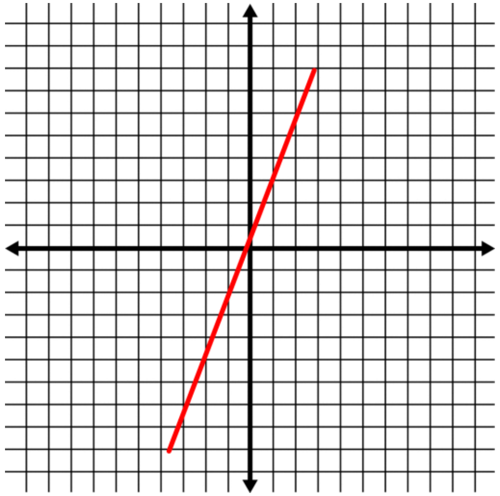
Therefore

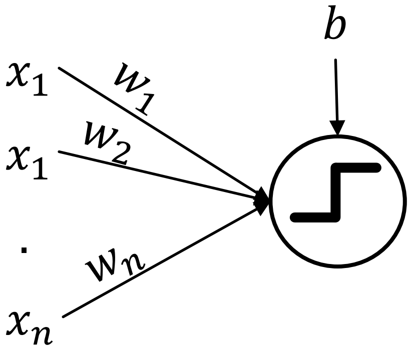
$$\hat{y} = f\left(\sum_{i=1}^{n} w_i x_i + b\right) = f(w.x + b)$$

Perceptron Cont.

- w is called the weights vector, which defines the orientation of the decision boundary, and b is called bias, and controls its shift

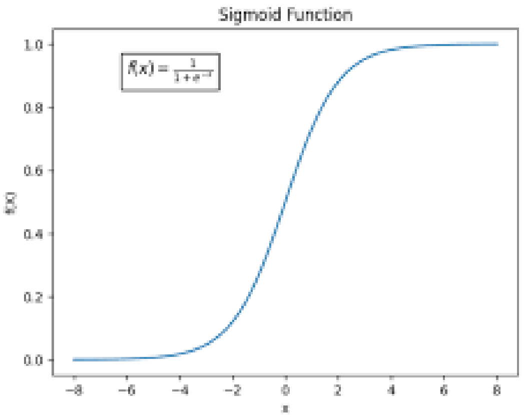






- sometimes instead of a step function, a sigmoid function is used for better learning

$$\hat{y} = \frac{1}{1+e^{-(z)}} = \frac{1}{1+e^{-(w.x+b)}}$$



Perceptron Cont.

- After activating the weighted sum using the sigmoid function we evaluate the error
  - How much $\hat{y}$ is far from a target value (actual class) y
- This can be calculated using a function called the loss function
  - There are many loss functions
- The one we use here is called mean squared error (MSE), which is given by

$$error = (\hat{y} - y)^2$$

- The aim is to make this error as low as possible
  - low error rate means better learning
  - in classification tasks, 0 error means the line (perceptron) is classifying the data points without errors
- But how to reduce the error?

Gradient decent

- We need to change the weights and bias (increase or decrease) in a way that makes the perceptron classify the data correctly
- But how do we know which weight to change, in which direction (increase or decrease), and the quantity of change?
  - We use a method called gradient descent
- gradient descent is a method based on differential equations (to find the slope of a function at a given point
- There are different types of this based on how you use them
  - Batch gradient decent: calculate the gradients on the whole data, then update
  - Mini-batch gradient decent: divide the dataset into small batches, each batch contains number of examples (16, 32, 64 , etc.)
  - Stochastic gradient decent: update after every sample (our focus in the next slides)

Gradient decent

- We need to change the weights and bias (increase or decrease) in a way that makes the perceptron classify the data correctly
- But how do we know which weight to change, in which direction (increase or decrease), and the quantity of change?
  - We use a method called gradient descent
- gradient descent is a method based on differential equations (to find the slope of a function at a given point
- It involves calculating the **partial derivative** of the loss function w.r.t the components of the w vector and b

$$\frac{\partial \text{ loss}}{\partial \text{ w}} \qquad \frac{\partial \text{ loss}}{\partial \text{ b}}$$

  - it gives how to change the w vector and b value so that the loss value is minimized (which we want to be 0

Gradient decent Cont.

- Assume that we have a single datapoint $x = 1$, $b = 0$ and the w initialized randomly to be $w = 3$, and the true label y for this point is 1
  - we have 1 input, so we have 1 w value

$$\hat{y} = w.x + b = 3 * 1 + 0 = 3$$

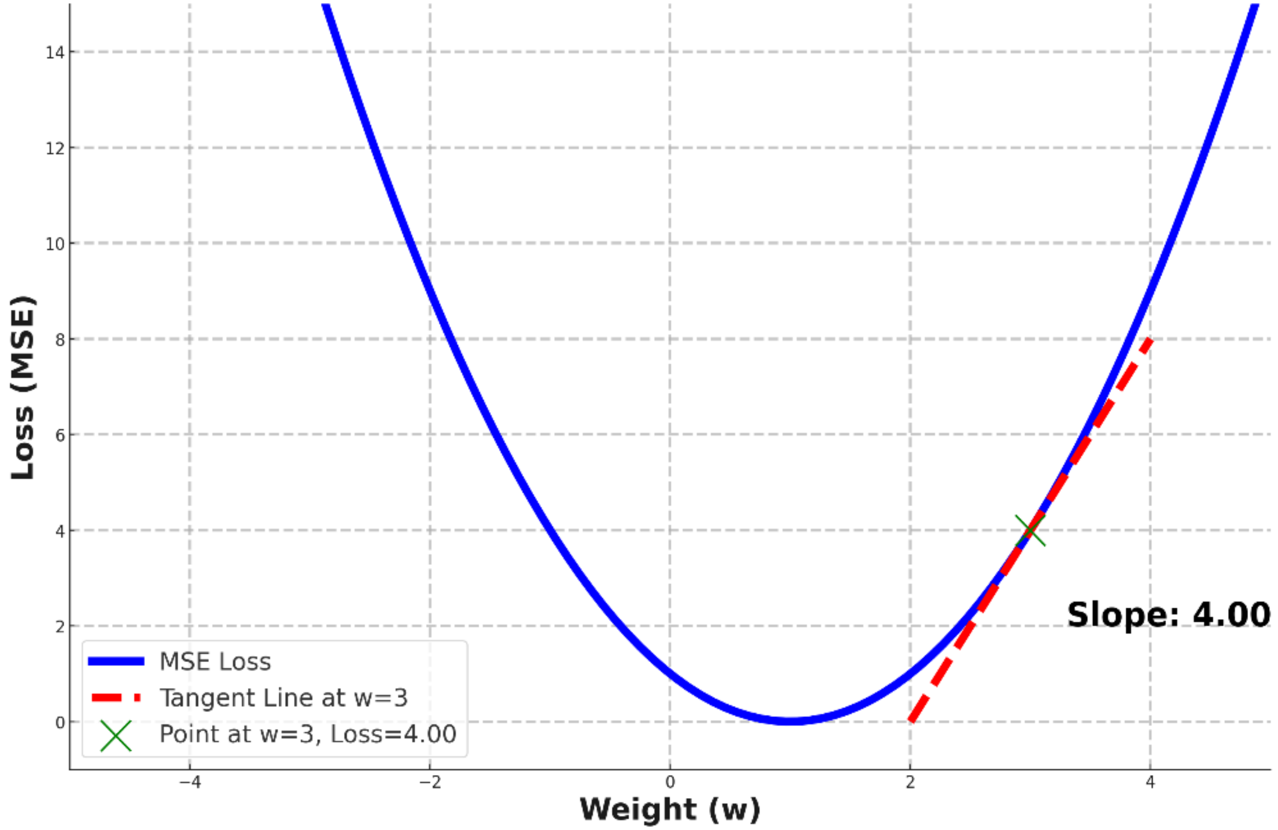
  - using this w and b the predicted value is 4

$$loss = (\hat{y} - \mathrm{y})^2 = (3 - 1)^2 = 4$$

  - to see how to change w we calculate the derivative of loss w.r.t w
- However, this is a compound function, we do not have direct access to w
  - function inside another function

$$((w.x + b) - \mathrm{y})^2$$

  - We need Chain rule!



the derivative of $f(g(x))$ is $f'(g(x)) \cdot g'(x)$

$$\frac{\partial \text{ loss}}{\partial \text{ w}} = \frac{\partial \text{ loss}}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial \text{ w}}$$

Gradient decent
Weights update

- Thus, the partial derivative of $((w.x+b)-\mathrm{y})^2$ w.r.t w is:

$$\frac{\partial \text{ loss}}{\partial \text{ w}} = \underbrace{\boxed{2*(\hat{y}-\mathrm{y})}}_{\frac{\partial \text{ loss}}{\partial \hat{y}}} * \underbrace{\boxed{x}}_{\frac{\partial \hat{y}}{\partial \text{ w}}} \qquad 2*(3-1)*1=4$$

- Therefore, the slope $\frac{\partial \text{ loss}}{\partial \text{ w}}$ (technically called gradient) is 4

- now we change the w

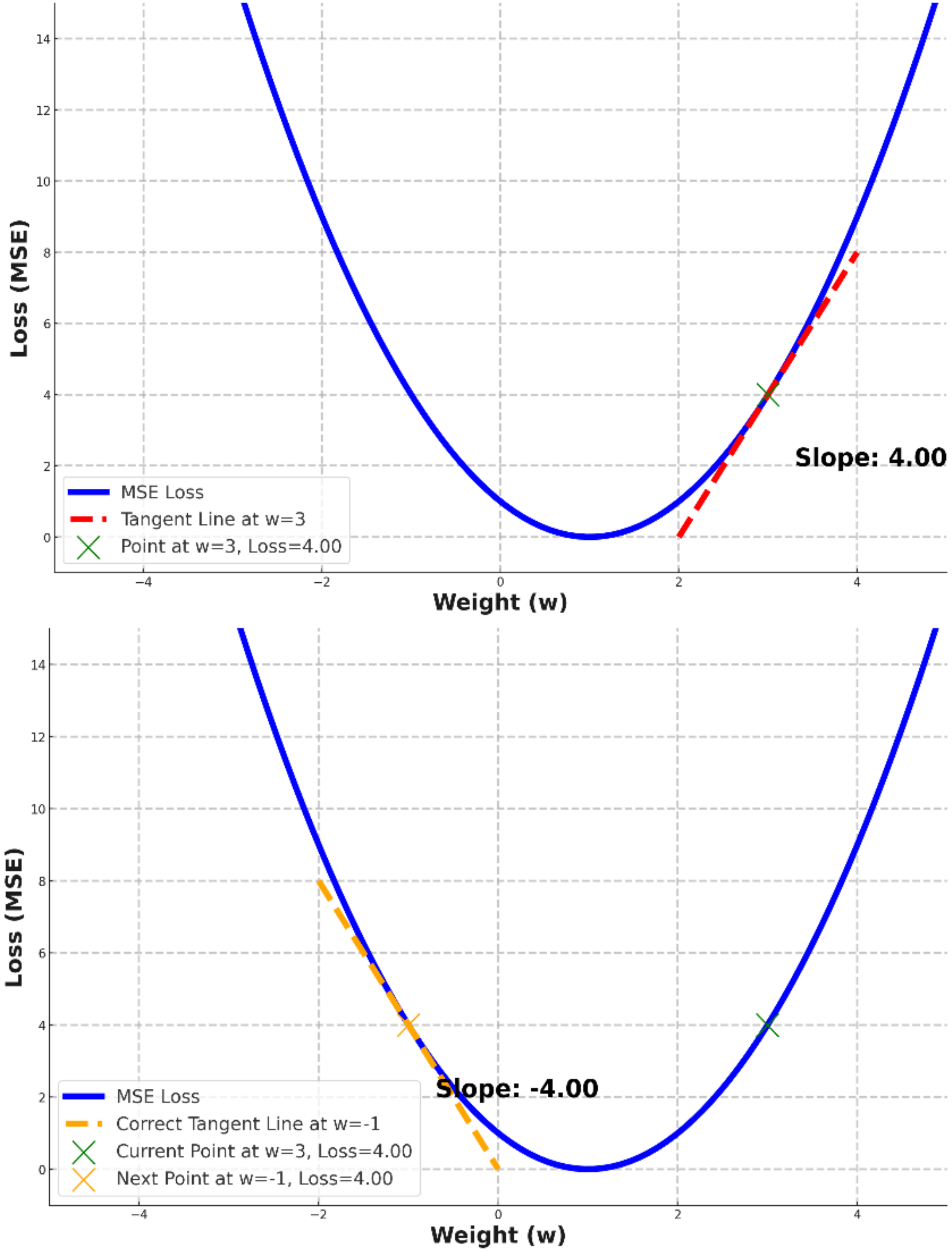
$$w_{new} = w_{old} - \frac{\partial \text{ loss}}{\partial \text{ w}} = 3-4=-1$$

- Repeat until the function reaches its minimum.

- if we use sigmoid we add the derivative of it to the formula to become

$$2*(\hat{y}-\mathrm{y})*\boxed{sig(z)*1-sig(z)}*x$$

where $\quad z = w.x+b$

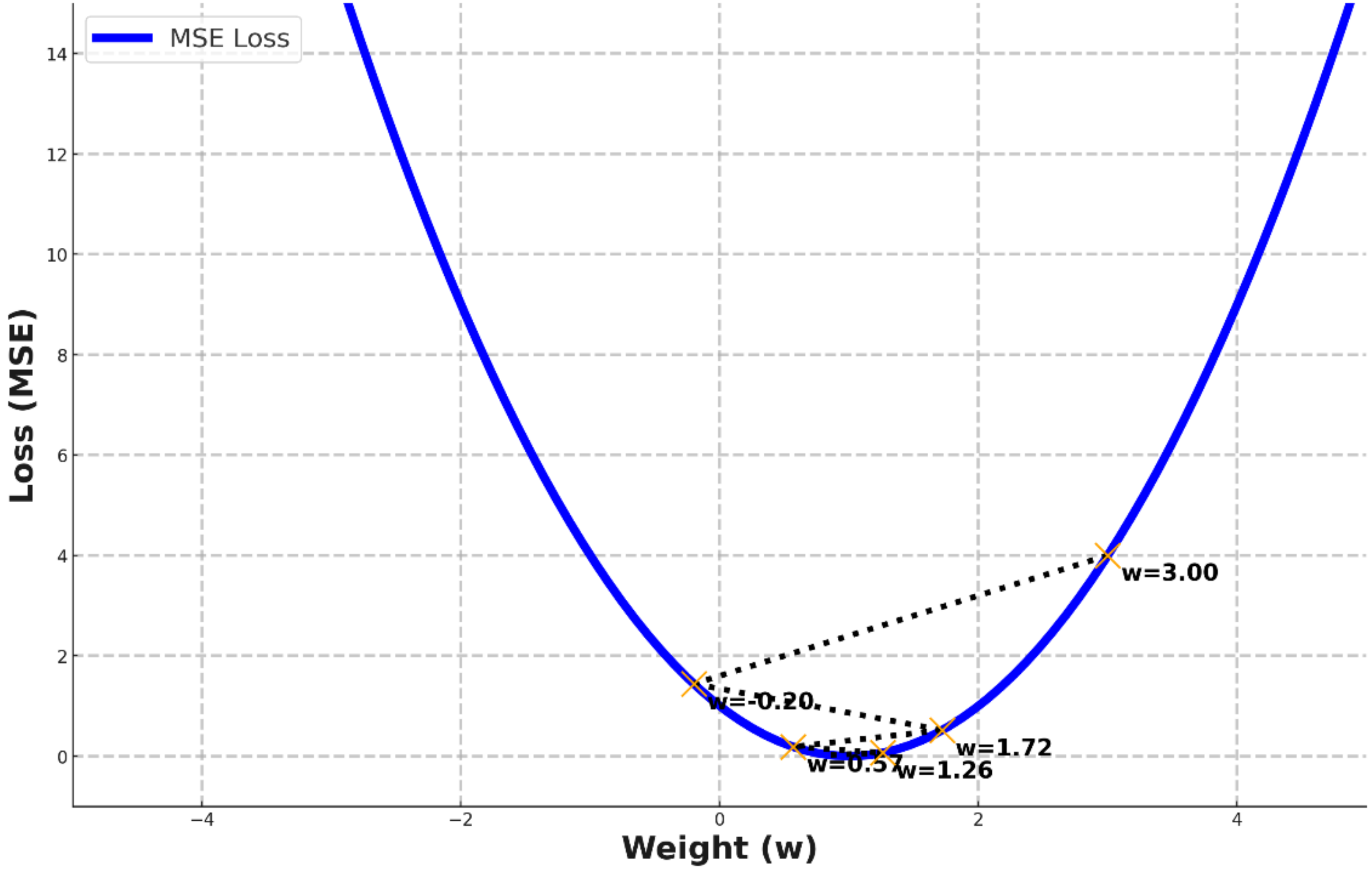


Perceptron
Update weights with learning rate α

- Note that, repeating in the current formula will keep the w fluctuating between 3 and -1 forever, and never reaches the minimum
  - Therefore, we add a small value called α to control the decent of w
  - α is a value between 0-1, e.g., 0.01

$$w_{new} = w_{old} - \alpha \frac{\partial \text{ loss}}{\partial \text{ w}}$$

  - The following figure shows several updates using $\alpha = 0.8$
- if x has more than one input, and therefore w, we do the same w.r.t each component of w



Perceptron
Bias update

- Note that we did not change b due to the simplicity of our example
- However, b needs to be changed in the same way, using gradient decent

$$\frac{\partial\ \text{loss}}{\partial\ b} = \frac{\partial\ \text{loss}}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial\ b}$$

$$\frac{\partial\ \text{loss}}{\partial\ b} = 2 * (\hat{y} - \text{y}) * \boxed{1}$$

derivative of $w.x + b$ w.r.t b

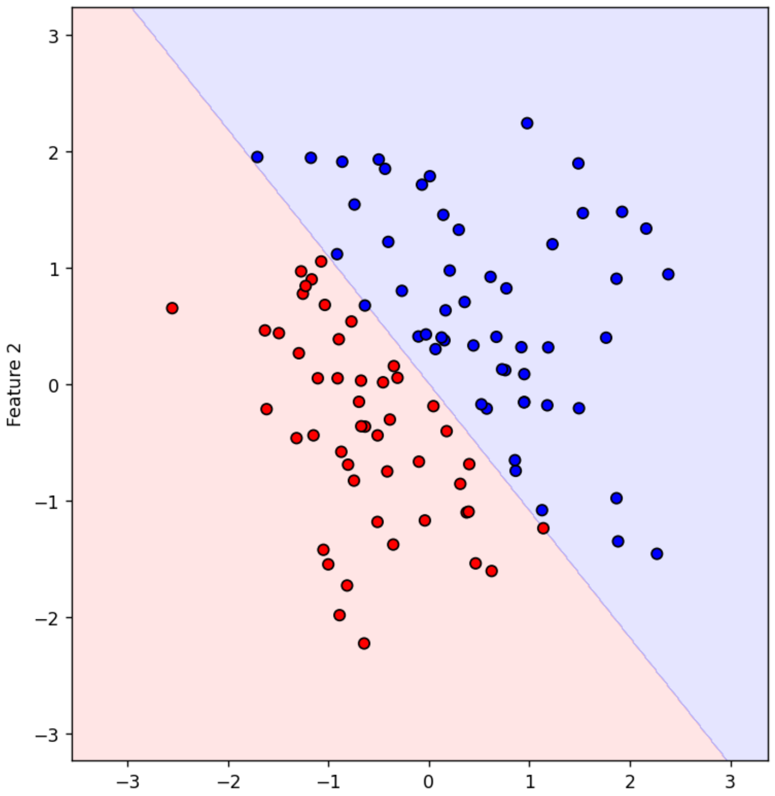
- and with sigmoid

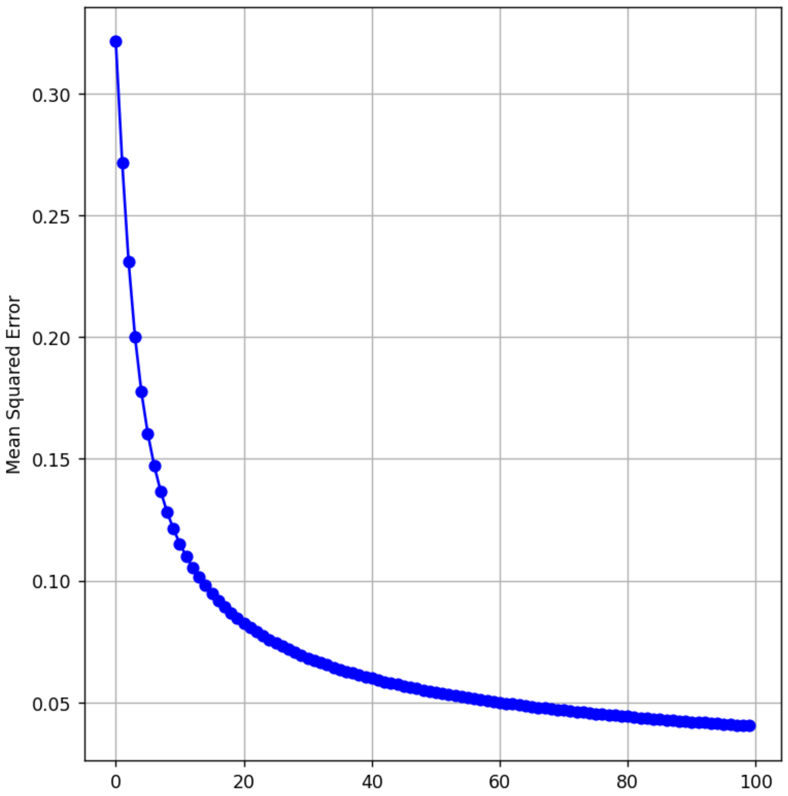
$$\frac{\partial\ \text{loss}}{\partial\ b} = 2 * (\hat{y} - \text{y}) * sig(z) * 1 - sig(z) * 1$$

Perceptron
Python code

```python
class Perceptron:
    def __init__(self, ninputs, epochs, alpha=0.01):
        self.n_epochs = epochs
        self.ninputs = ninputs
        self.weights = np.random.randn(ninputs)
        self.bias = 0
        self.alpha = alpha
        self.loss_history = []

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def mse(self, preds, y):
        return (preds - y) ** 2

    def dmse(self, preds, y):
        return 2 * (preds - y)

    def fit(self, X, y):
        for epoch in range(self.n_epochs):
            error = 0
            for sample, label in zip(X, y):
                z = np.dot(sample, self.weights) + self.bias
                preds = self.sigmoid(z)

                error += self.mse(preds, label)

                gradientsW = self.dmse(preds, label) * preds * (1 - preds) * sample
                gradientsB = self.dmse(preds, label) * preds * (1 - preds)

                self.weights -= self.alpha * gradientsW
                self.bias -= self.alpha * gradientsB

            avg_error = error / len(X)
            self.loss_history.append(avg_error)
            print(f'Epoch {epoch + 1}/{self.n_epochs}, Error: {avg_error}')

    def predict(self, x):
        y_hat = np.dot(x, self.weights) + self.bias
        return (self.sigmoid(y_hat) >= 0.5).astype(int)
```

Perceptron
Visualization

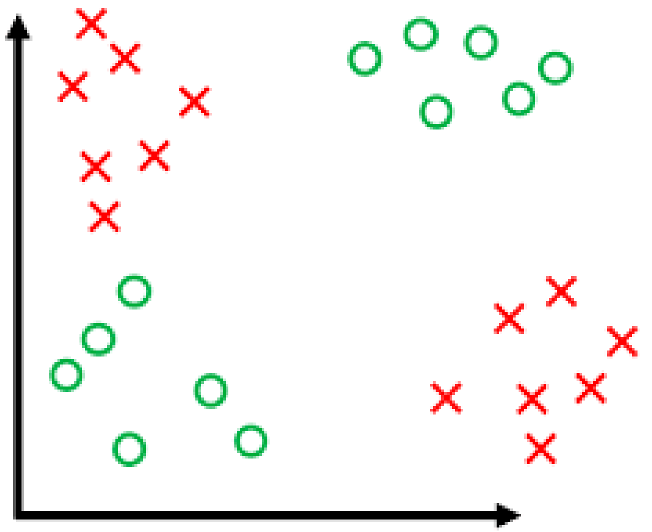




Loss over epochs

Perceptron
Final note

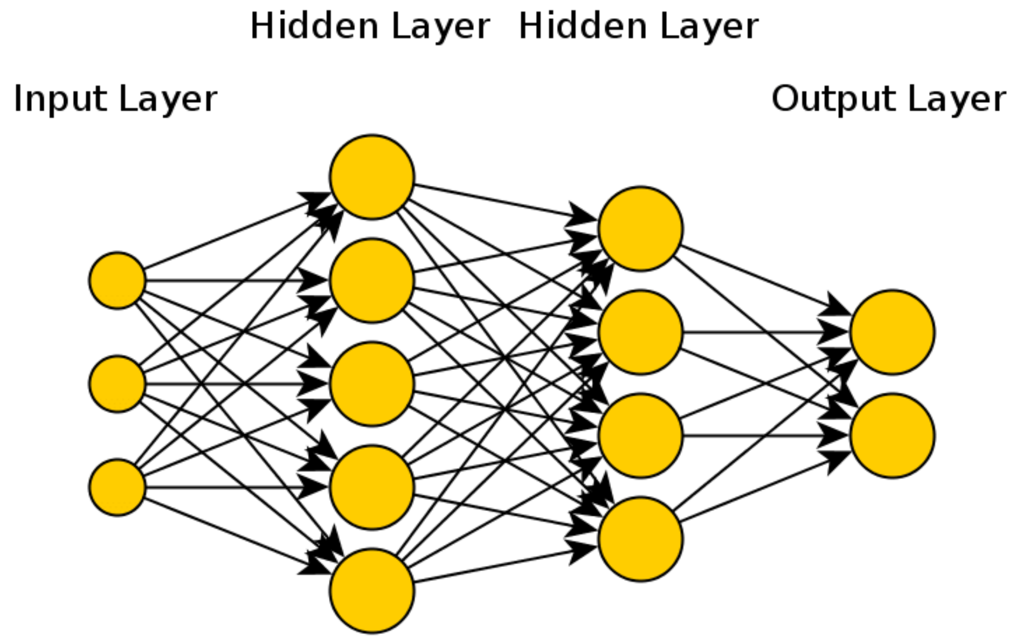
- Perceptron is weak in classifying nonlinearly separable data, as it is a linear model
  - Think of the XOR problem
- Impossible to solve using single perceptron
  - Can you find a line that can separate the following data points?



- We need something more complex
  - Multilayer Perceptron (MLP)

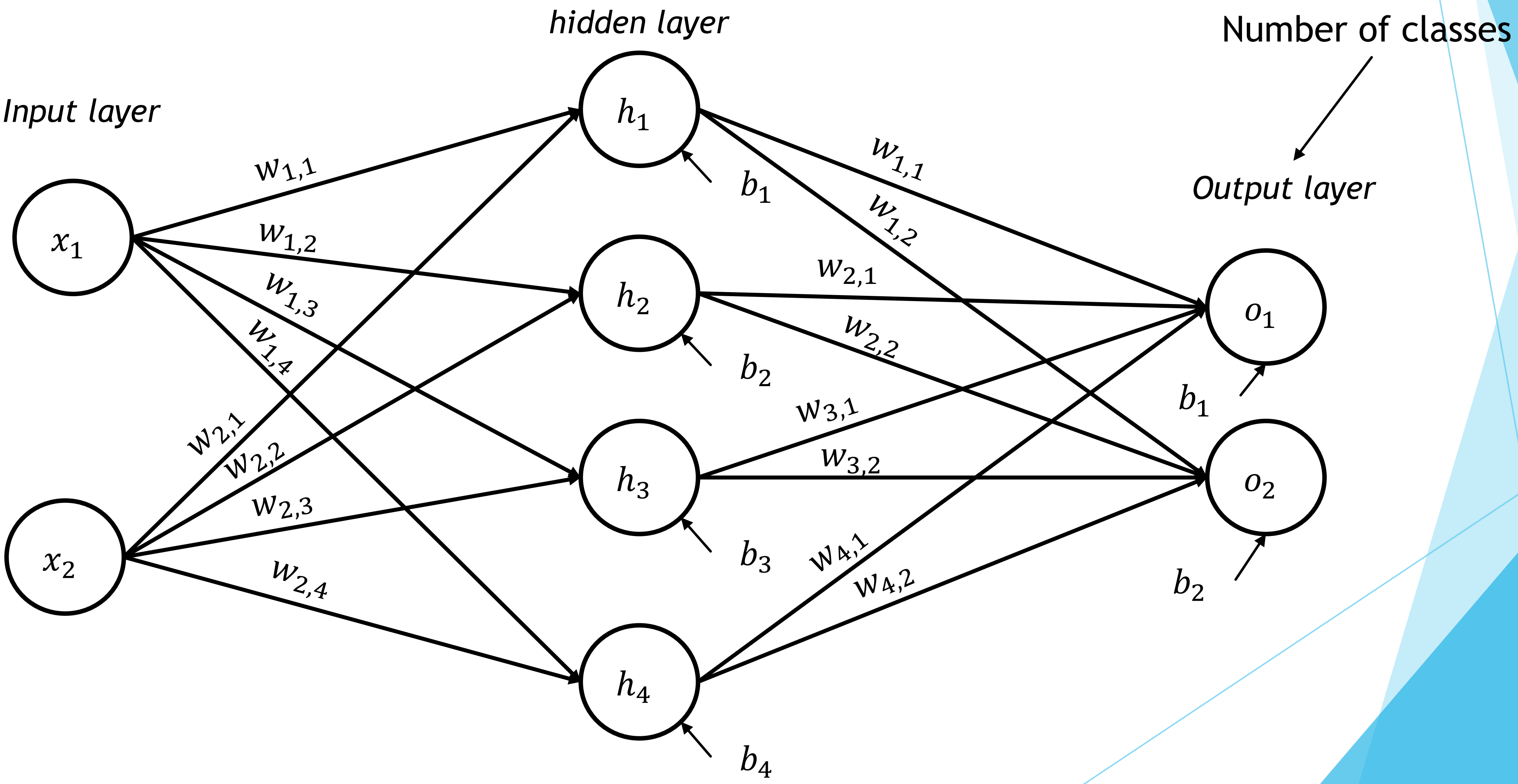
Multilayer Perceptron

- An MLP is a type of artificial neural network that consists of multiple layers of Perceptron (neurons), including an input layer, one or more hidden layers, and an output layer
  - Input Layer: The first layer that receives the input features.
  - Hidden Layers: One or more intermediate layers where the actual processing is done through weighted connections. Each neuron in a hidden layer applies a non-linear activation function to the weighted sum of inputs.
  - Output Layer: The final layer that produces the output of the network, representing the result of the prediction or classification.



Cont.

- The following present an example architecture of MLP
  - Each node in the hidden and output layers has associated weights and bias



MLP, Cont.

- The training process of MLP network is similar to that of perceptron, using gradient decent
  - it consists of forward pass and then correct the weights based on the error in a step called backpropagation

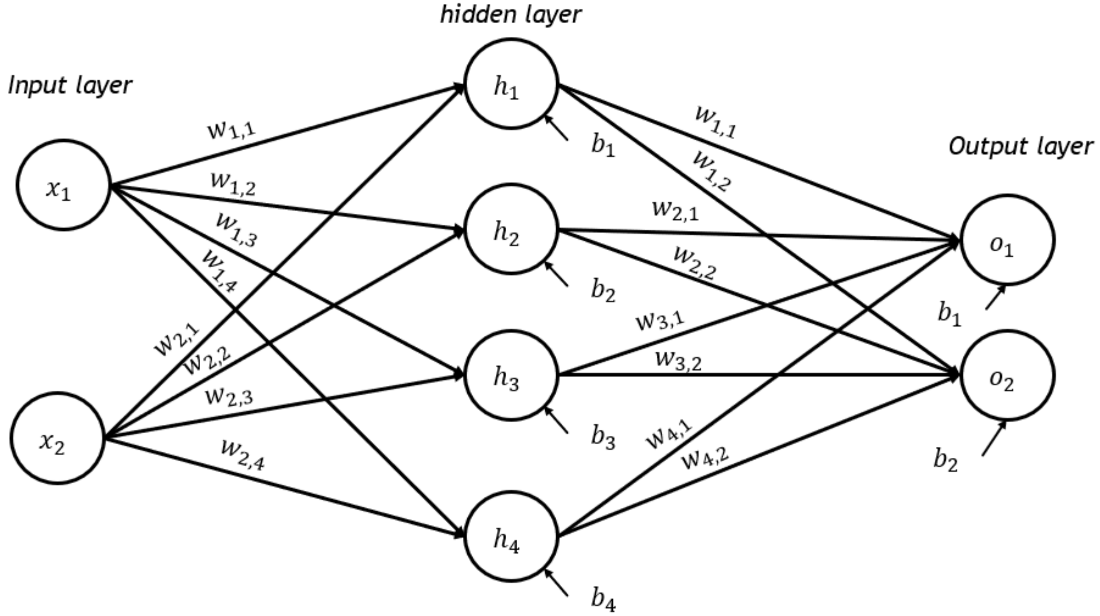
- In the **forward pass** we calculate the linear combination of each single node in the network sequential order using the same formula used in the perceptron

$$\hat{y} = w.x + b$$

- Then we calculate the error for each output node

- In the **backward pass** we correct the parameters w and b in whole architecture in a reversed sequential order

Cont.



The forward pass of the previous architecture looks as follows

$$h_1 = x_1 * w_{1,1} + x_2 * w_{2,1} + b_1$$

$$h_2 = x_1 * w_{1,2} + x_2 * w_{2,2} + b_2$$

$$h_3 = x_1 * w_{1,3} + x_2 * w_{2,3} + b_3$$

$$h_4 = x_1 * w_{1,4} + x_2 * w_{2,4} + b_4$$

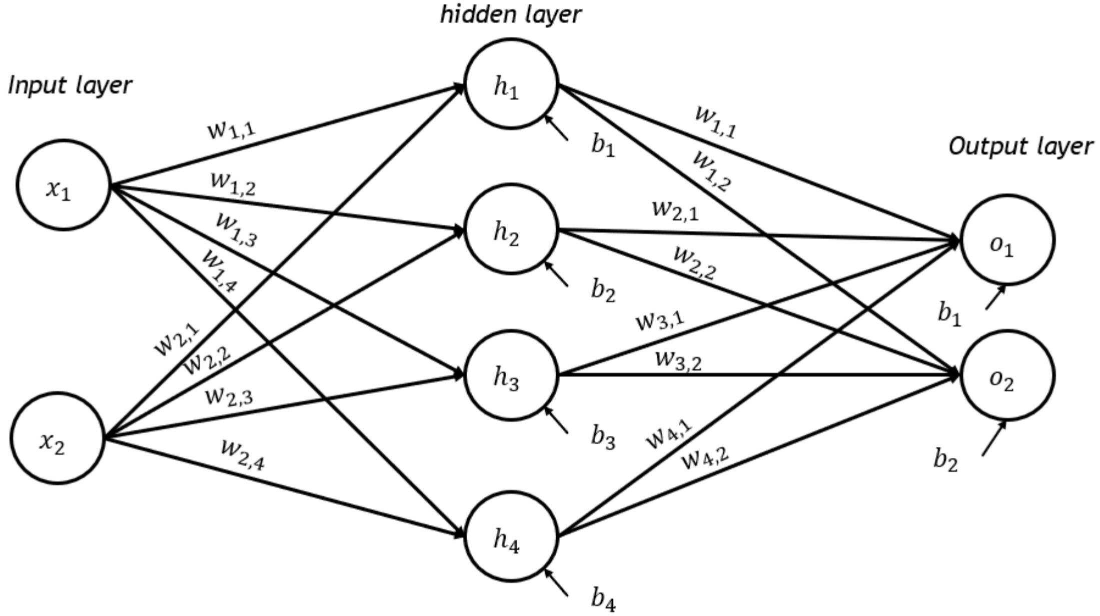
$$o_1 = h_1 * w_{1,1} + h_2 * w_{2,1} + h_3 * w_{3,1} + h_4 * w_{4,1} + b_1$$

$$o_2 = h_1 * w_{1,2} + h_2 * w_{2,2} + h_3 * w_{3,2} + h_4 * w_{4,2} + b_2$$

But MLP network typically include much greater number of nodes, which makes the calculations, in the above manner, complex and inefficient

SOLUTION: Use matrix operations

Cont. MLP Matrices



- w in each layer can be represented as weight matrix of size $m \times n$, where m is the input values, and n is the output values (number of nodes in the next layer)
  - w in the first layer is 2×4 ; 2 is the number of inputs to this layer (x_1 and x_2) and 4 is the number of outputs (h_1, h_2, h_3 and h_4)

$$w1 = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} \end{bmatrix}$$

  - Also, the inputs x can be represented as a matrix $s \times m$, where s is the number of samples
    - if we pass one sample at a time (Stochastic gradient decent), then $s = 1$
    - m is the number of values in x

$$x = \begin{bmatrix} x_1 & x_2 \end{bmatrix}$$

  - Also b in the hidden layer h is a vector matrix of size 1×4

$$b = [\, b_1, b_2, b_3 \; and \; b_4]$$

Cont.

- Now instead of calculating the h values individually we perform matrix multiplication to directly calculate the terms in the hidden layer h_1, h_2, h_3 and h_4

$$h = x.w1 + b1$$

  - remember the inner dimension should match in the matrix multiplications

$$h = \begin{bmatrix} x_1 & x_2 \end{bmatrix} . \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} \end{bmatrix} + \begin{bmatrix} b_1 & b_2 & b_3 & b_4 \end{bmatrix}$$

  - matrix multiplication is performed by weight sum row from the first matrix and the column from the second matrix
  - The output has the outer dimension of the multiplied matrices $s \times n$, in our case 1×4

Cont.

- This how it works

$$h_{temp} = \boxed{[x_1 \quad x_2]} \cdot \begin{bmatrix} \boxed{\begin{matrix} w_{1,1} \\ w_{2,1} \end{matrix}} & \boxed{\begin{matrix} w_{1,2} \\ w_{2,2} \end{matrix}} & \boxed{\begin{matrix} w_{1,3} \\ w_{2,3} \end{matrix}} & \boxed{\begin{matrix} w_{1,4} \\ w_{2,4} \end{matrix}} \end{bmatrix}$$

$$h_{temp} = [x_1 * w_{1,1} + x_2 * w_{2,1}, x_1 * w_{1,2} + x_2 * w_{2,2}, x_1 * w_{1,3} + x_2 * w_{2,3}, x_1 * w_{1,4} + x_2 * w_{2,4}]$$

- Then we add, elementwise summation, the bias terms to get the final outputs of in h

$$h = h_{temp} + b1$$

$$h = [x_1 * w_{1,1} + x_2 * w_{2,1} + b, \; x_1 * w_{1,2} + x_2 * w_{2,2} + b, \; x_1 * w_{1,3} + x_2 * w_{2,3} + b, \; x_1 * w_{1,4} + x_2 * w_{2,4} + b]$$

$$h = [h_1, h_2, h_3, h_4]$$

- Now this h becomes the input for the next layer and we repeat until we get the outputs
- Same results we get before using math operations, but what makes it efficient is that this process is equivalent to dot product and vector summation in numpy, Python

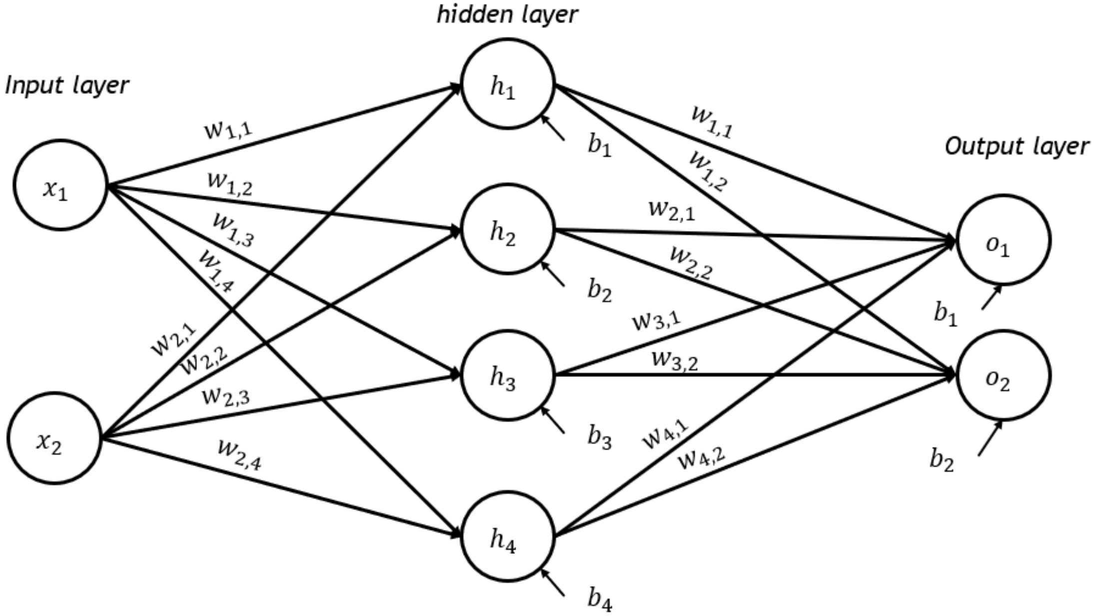
```
h = np.dot(x,w)+b
```

Example

- Now let us take example to see how to calculate the forward pass using our architecture
  - Assume that our dataset has 1 example
  - y (class) should be one-hot encoded, having 1 at index 0 means that the actual class is 0

$$x = [3 \quad 5] \qquad y = [1 \quad 0]$$

  - w is randomly initialized, and b is a vector of ones



$$x = [3 \quad 5] \qquad w1 = \begin{bmatrix} 0.5 & 1 & 7 & 3 \\ 1 & 2 & 1 & 0.5 \end{bmatrix} \qquad b1 = [1 \quad 1 \quad 1 \quad 1]$$

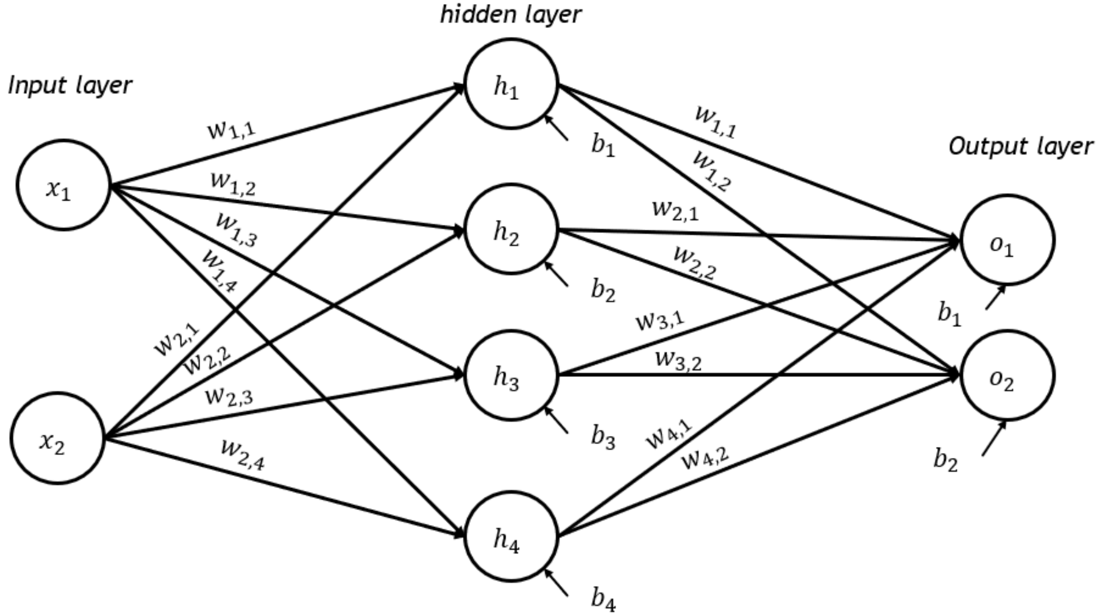
$$h = [3 * 0.5 + 5 * 1 + 1, \quad 3 * 1 + 5 * 2 + 1, \quad 3 * 7 + 5 * 1 + 1, \quad 3 * 3 + 5 * 0.5 + 1]$$

$$h = [7.5, 14, 27, 12.5]$$

```
x = np.array([3, 5])
w = np.array([[0.5, 1, 7, 3],[1, 2, 1, 0.5]])
b = np.array([1, 1, 1, 1])
h = np.dot(x,w)+b
```

array([7.5, 14. , 27. , 12.5])

Example, Cont.



- Now h becomes the input to the new layer which has $w\ 4 \times 2$

$$h = [7.5, 14, 27, 12.5] \qquad w2 = \begin{bmatrix} 1 & 2 \\ 0.5 & 0.5 \\ 3 & 1 \\ 2 & 1 \end{bmatrix} \qquad b2 = [1 \quad 1 \quad 1 \quad 1]$$

$$\text{o} = [7.5 * 1 + 14 * 0.5 + 27 * 3 + 12.5 * 2 + 1, \qquad 7.5 * 2 + 14 * 0.5 + 27 * 1 + 12.5 * 1 + 1]$$

$$\text{o} = [121.5\,, 62.5]$$

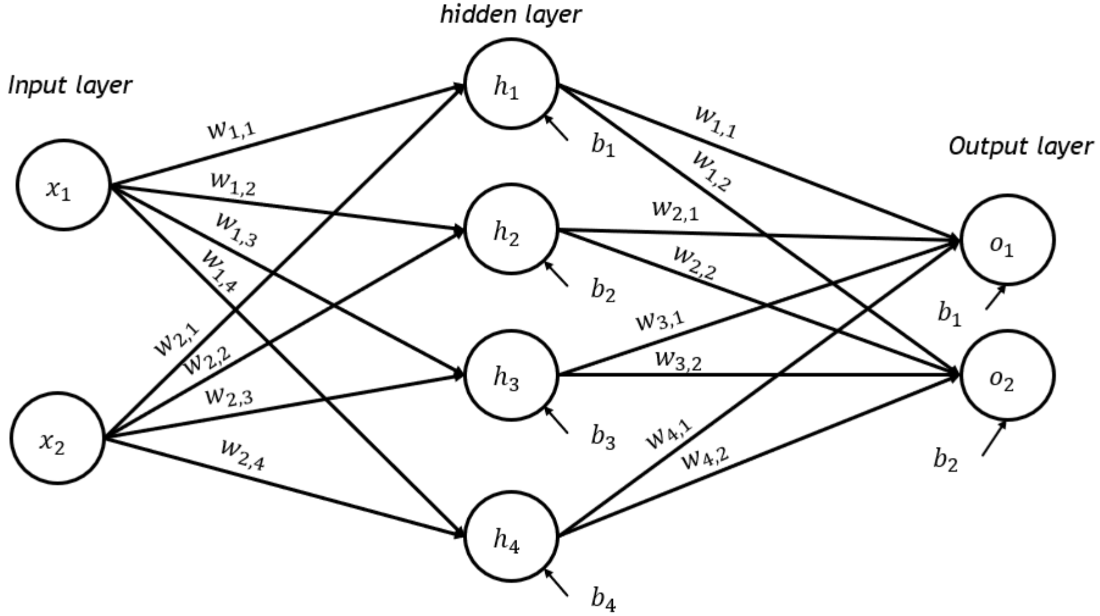
- The error is calculated between the predicted and the actual using MSE loss
  - its equation now is a bit different as we have multi label

$$error = \frac{1}{n}\sum_{i=1}^{n}(o_i - y_i)^2$$

$$error = \frac{1}{2}\sum_{i=1}^{n}(\text{o}_i - y_i)^2 = \frac{1}{2} * (121.5 - 1)^2 + (62.5 - 0)^2 = \frac{1}{2} * 14{,}520 + 3{,}906 = 9{,}213$$

Backward pass

- Now we have the error, and we want to adjust the weights and the bias terms in both layers to reduce this error
  - Using backpropagation algorithm (gradient decent)
- remember from chain rule we have to calculate the derivatives w.r.t $w2, \mathrm{b}2, w1$and $b1$
- we start with calculating the gradient of loss w.r.t each element in o, $\frac{\partial \text{ loss}}{\partial o_i}$
  - note here o is $\hat{y}$ in the previous example (predictions)



$$\frac{\partial \text{ loss}}{\partial o_i} = \frac{2}{2}(o_i - y_i)$$

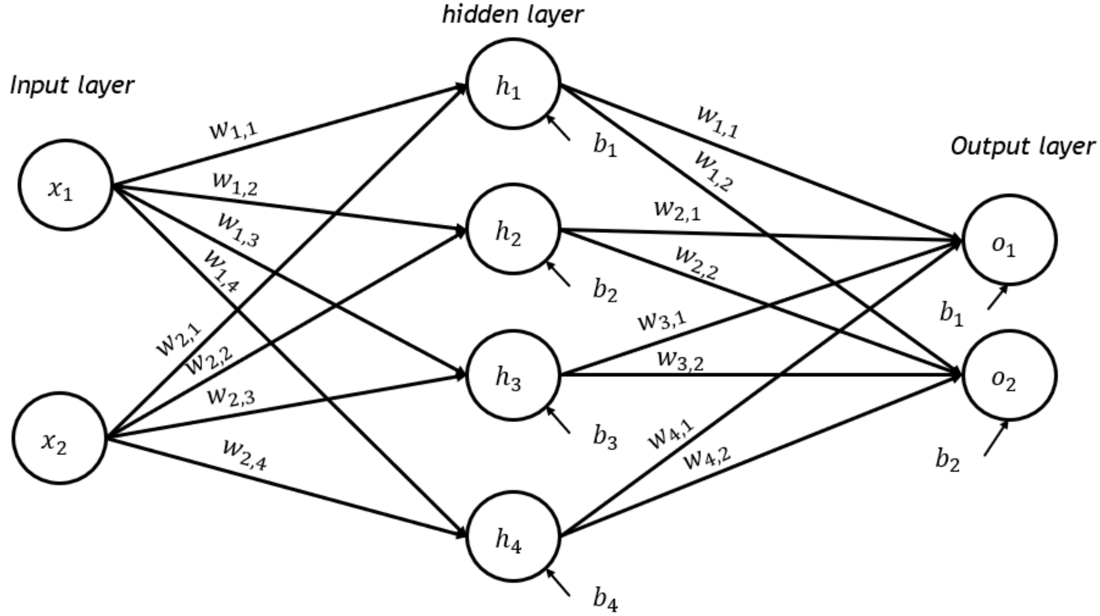
$$\frac{\partial \text{ loss}}{\partial o_1} = 1 * (121.5 - 1) = 120.5$$

$$\frac{\partial \text{ loss}}{\partial o_2} = 1 * (62.5 - 0) = 62.5$$

$$\frac{\partial \text{ loss}}{\partial o} = [120.5 , 62.5]$$

Backward pass, Cont.

- Now we have to find the derivative w.r.t $w2$ and $b2$



$$\frac{\partial\,\text{loss}}{\partial w2} = \begin{bmatrix} \frac{\partial\,\text{loss}}{\partial w_{1,1}} & \frac{\partial\,\text{loss}}{\partial w_{1,2}} \\ \frac{\partial\,\text{loss}}{\partial w_{2,1}} & \frac{\partial\,\text{loss}}{\partial w_{2,2}} \\ \frac{\partial\,\text{loss}}{\partial w_{3,1}} & \frac{\partial\,\text{loss}}{\partial w_{3,2}} \\ \frac{\partial\,\text{loss}}{\partial w_{4,1}} & \frac{\partial\,\text{loss}}{\partial w_{4,2}} \end{bmatrix}$$

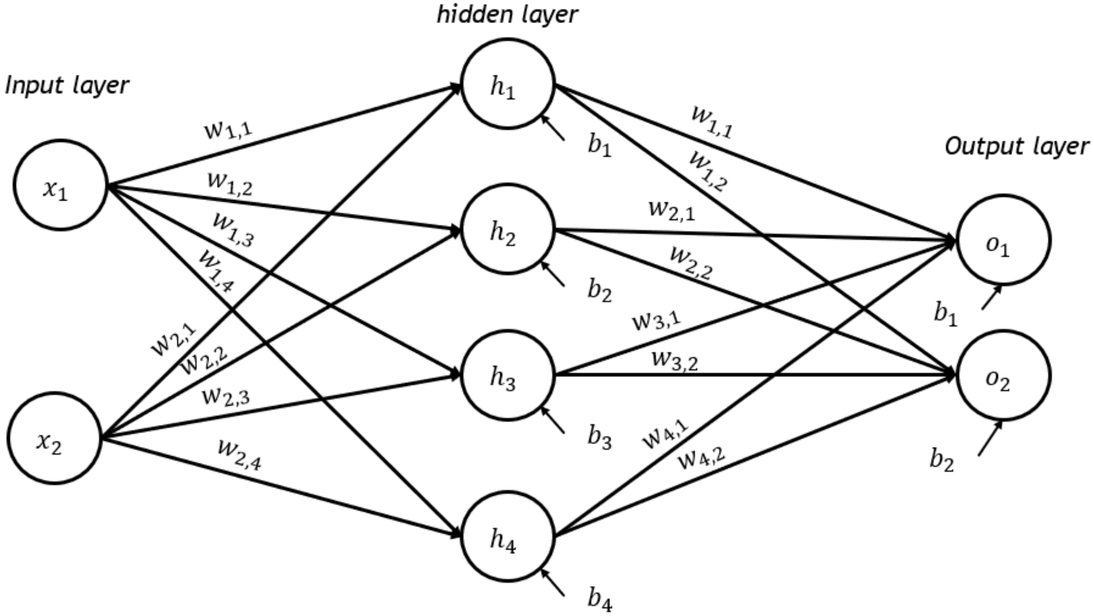
$$\frac{\partial o_1}{\partial w_{1,1}} = h1$$

$$\frac{\partial o_1}{\partial w_{1,2}} = h2$$

$$\frac{\partial\,\text{loss}}{\partial w_{1,1}} = \frac{\partial\,\text{loss}}{\partial o_1} * \frac{\partial o_1}{\partial w_{1,1}} = 120.5 * 7.5 = 903.75$$

$$\frac{\partial\,\text{loss}}{\partial w_{1,2}} = \frac{\partial\,\text{loss}}{\partial o_2} * \frac{\partial o_2}{\partial w_{1,2}} = 62.5 \ * 7.5 = 468.75$$

Cont.



- This calculations can be done simply using matrix operations

$$h^T . o$$

$$\frac{\partial \text{ loss}}{\partial w2} = \begin{bmatrix} 7.5 \\ 14 \\ 27 \\ 12.5 \end{bmatrix} . [121.5\,, 62.5] \qquad \frac{\partial \text{ loss}}{\partial w2} = \begin{bmatrix} 903.75 & 468.75 \\ 1687 & 875 \\ 3253.5 & 1687.5 \\ 1506.25 & 781.25 \end{bmatrix}$$

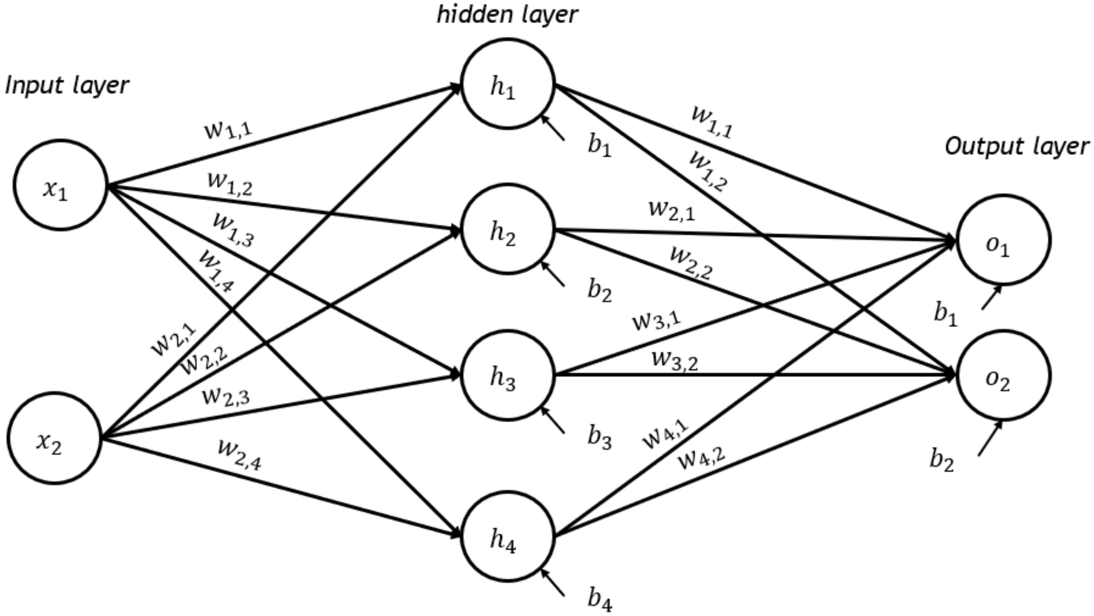
$$\frac{\partial \text{ loss}}{\partial b2} = \frac{\partial \text{ loss}}{\partial o} * \frac{\partial \text{ o}}{\partial b2} *= [121.5\,, 62.5] * [1, 1] = [121.5\,, 62.5]$$

- Although these numbers looks large, we update the weights in $w2$ with a proportion of them
    - do not forget the learning rate α

$$w2_{new} = w2_{old} - \alpha * \frac{\partial \text{ loss}}{\partial w2}$$

$$b2_{new} = b2_{old} - \alpha * \frac{\partial \text{ loss}}{\partial b2}$$

Cont.



- In this way we successfully updated the weights $\mathrm{w}2$ and bias $b2$ in the second layer
  - But what about the weights and bias in the first layer, $\mathrm{w}1$ and $b1$?

- for this we need to calculate the gradient with respect to the input $\frac{\partial\,\mathrm{loss}}{\partial h}$, so that we pass it to the previous layer and repeated the calculations
  - it is simply given by

$$\frac{\partial\,\mathrm{loss}}{\partial h} = [121.5\,, 62.5].\,w2_{old}{}^{T}$$

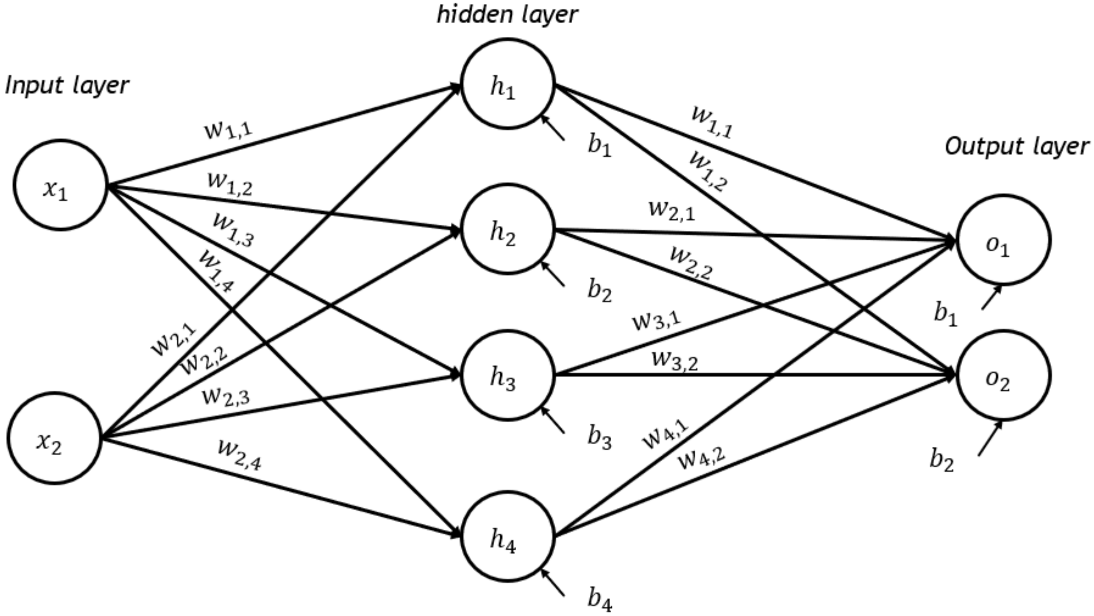
- The above dot product gives a matrix 1×4 which is the gradient w.r.t h
  - Use it to calculate for the $\mathrm{w}1$ and $b1$

Cont.

- Note that in the previous explanation we did not include any activation functions for **simplicity**
- The MLP will remain linear no matter how layers or nodes you add
- to solve this, we use a nonlinear activation function to activate the output of each node in the hidden or output layers
  - Sigmoid for example
- the math will work in the same way with additional step, which is calculating and propagating the derivative of sigmoid to the process

Code

- Let us see how can we implement this architecture and all its math using Python
- We implement a class called layer, so that each object (layer) has its own weights and bias stored in it and perform the operations independently




```python
import numpy as np

class Dense:
    def __init__(self, inF, outF):
        self.weights = np.random.randn(inF, outF)
        self.biases = np.ones((1, outF))

    def forward(self, inputs):
        self.inputs = inputs
        return np.dot(inputs, self.weights) + self.biases

    def backward(self, dvalues):
        self.dweights = np.dot(self.inputs.T, dvalues)
        self.dbiases = np.sum(dvalues, axis=0, keepdims=True)
        self.dinputs = np.dot(dvalues, self.weights.T)

layer1 = Dense(2,4)
layer2 = Dense(4,2)
```

Let us check if we use the data in our example, we will get the same results

Code, Cont.

```
layer1.weights = np.array([[0.5, 1, 7, 3],[1, 2, 1, 0.5]])
layer2.weights = np.array([[1, 2],[0.5, 0.5],[3, 1],[2, 1]])

x = np.array([[3,5]])
y = np.array([[1,0]])

h = layer1.forward(x)
print(f"the output of the first layer is {h}")
o = layer2.forward(np.array(h))
print(f"the output of the second (output) layer is {o}")

def mse_loss(y_true, y_pred):
    return np.mean((y_pred - y_true) ** 2)

# Derivative of MSE Loss
def mse_loss_derivative(y_true, y_pred):
    return 2 * (y_pred - y_true) / y_true.size

loss = mse_loss(y, o)
print(f"Mean Squared Error Loss: {loss}")

dL_do = mse_loss_derivative(y, o)
layer2.backward(dL_do)
layer1.backward(layer2.dinputs)

print(f"Gradients of layer2 weights: \n {layer2.dweights}")
print(f"Gradients of layer2 biases: \n {layer2.dbiases}")

print(f"Gradients of layer1 weights: \n {layer1.dweights}")
print(f"Gradients of layer2 biases: \n {layer2.dbiases}")
```

```
the output of the first layer is [[ 7.5 14.   27.   12.5]]
the output of the second (output) layer is [[121.5   62.5]]
Mean Squared Error Loss: 9213.25
Gradients of layer2 weights:
 [[ 903.75   468.75]
 [1687.     875.  ]
 [3253.5   1687.5 ]
 [1506.25   781.25]]
Gradients of layer2 biases:
 [[120.5   62.5]]
Gradients of layer1 weights:
 [[ 736.5   274.5 1272.    910.5]
 [1227.5   457.5 2120.   1517.5]]
Gradients of layer2 biases:
 [[120.5   62.5]]
```

Code, Cont.

- Update the weights and biases in these layers and recheck, using the same point, if the loss is decreasing

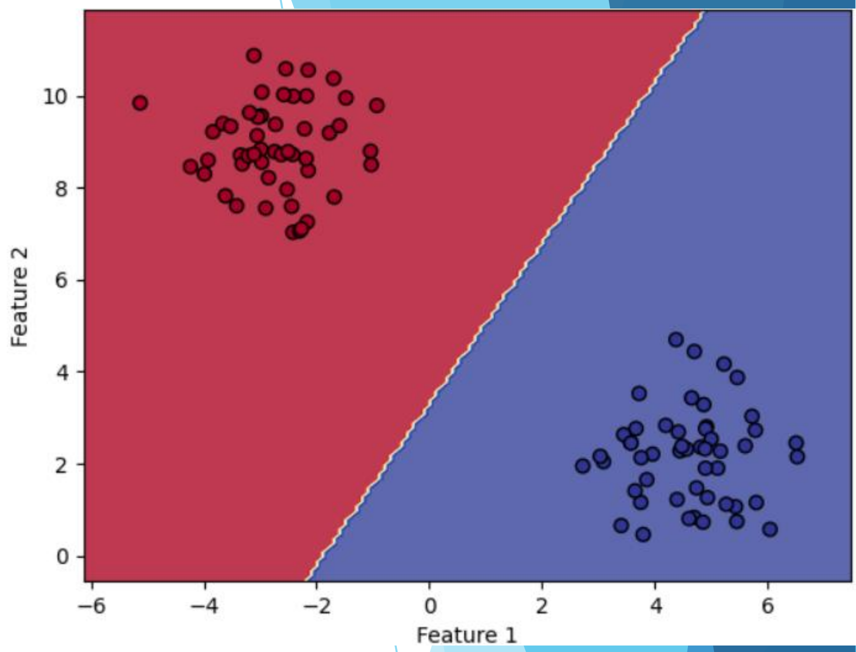
```
alpha = 0.001

layer1.weights -= alpha * layer1.dweights
layer2.weights -= alpha * layer2.dweights

layer1.biases -= alpha * layer1.dbiases
layer2.biases -= alpha * layer2.dbiases

h = layer1.forward(x)
o = layer2.forward(h)
loss = mse_loss(y, o)
print(f"Mean Squared Error Loss: {loss}")
```

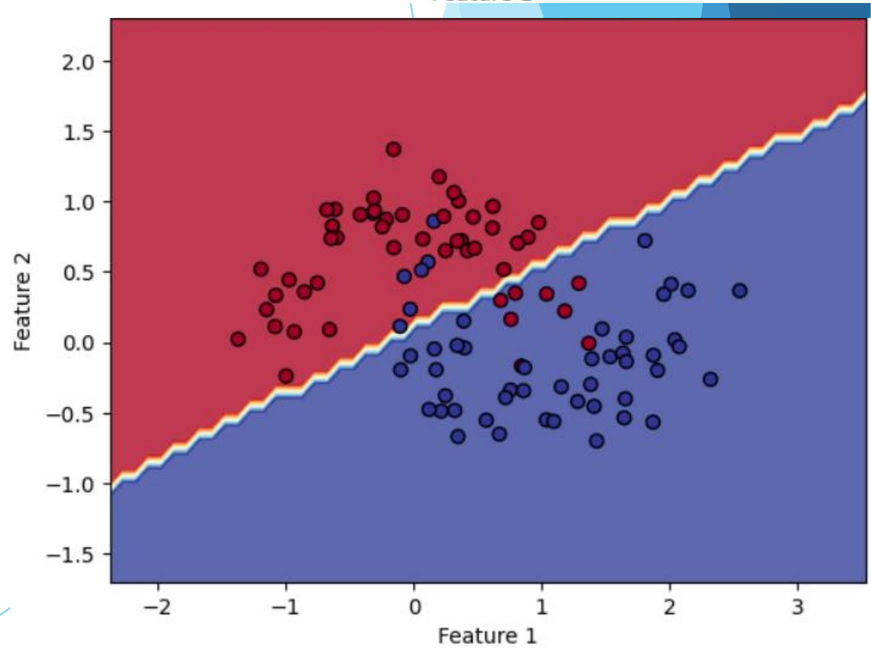
```
Loss: 196.56
```

The loss decreases.
There is learning!

Example using dataset

```python
X, y = make_blobs(n_samples=100, centers=2, n_features=2, random_state=42)

one_hot_encoder = OneHotEncoder(sparse=False)
y_onehot = one_hot_encoder.fit_transform(y.reshape(-1, 1))  # Shape: (100, 2)

layer1 = Dense(2, 4)
layer2 = Dense(4, 2)

alpha = 0.01
epochs = 10
# Training loop
for epoch in range(epochs):
    loss_epoch = 0
    for i in range(len(X)):
        x_sample = np.expand_dims(X[i], axis=0)  #  (shape: (1, 2))
        y_sample = np.expand_dims(y_onehot[i], axis=0)  # (shape: (1, 2))

        h = layer1.forward(x_sample)
        o = layer2.forward(h)  # Raw output
        # loss
        loss = mse_loss(y_sample, o)
        loss_epoch += loss
        # Calculate gradients
        dL_do = mse_loss_derivative(y_sample, o)
        layer2.backward(dL_do)
        layer1.backward(layer2.dinputs)
        # Update w and b
        layer1.weights -= alpha * layer1.dweights
        layer1.biases -= alpha * layer1.dbiases
        layer2.weights -= alpha * layer2.dweights
        layer2.biases -= alpha * layer2.dbiases

    # Print average loss for the epoch
    if epoch % 1 == 0:
        print(f"Epoch {epoch}, Loss: {loss_epoch / len(X)}")
```





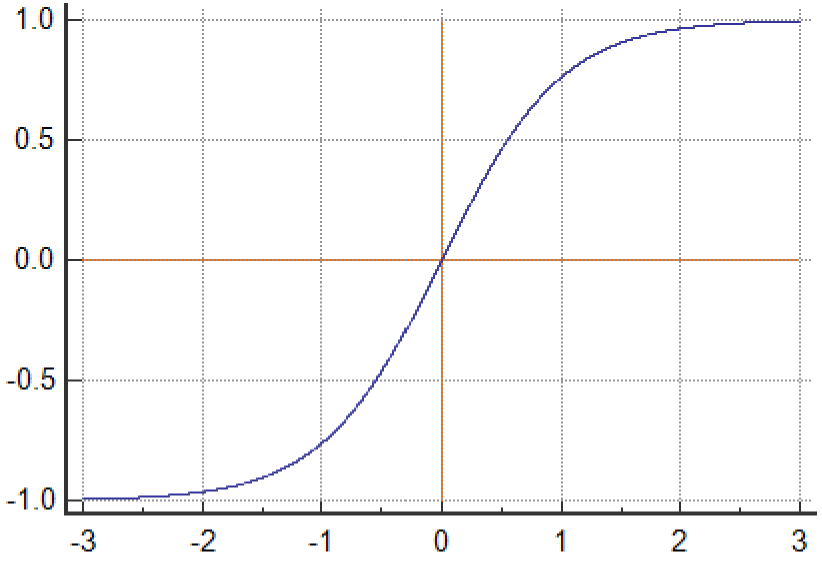
Activation functions

- For the previous example we note that the form of MLP we used before cannot separate nonlinearly separable
- to make it able to solve this kind of data we have to add non linear activation functions to the MLP after each layer, as we discussed earlier
- There are many activation functions
  1. Sigmoid (discussed before)
  2. Tanh
  3. ReLU
  4. Leaky ReLU
  5. Softmax

Tanh: the Hyperbolic function

- Tanh is similar to the Sigmoid except that it squishes the values between -1 and 1 instead of 0 and 1
- it is given by the following expression

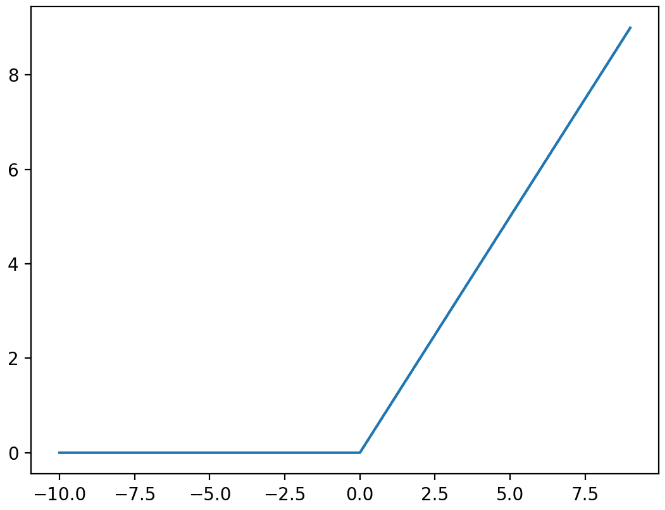
$$\tanh = \frac{e^x - e^{-x}}{e^x + e^{-x}} \qquad \tanh' = 1 - tanh^2(x)$$



ReLU: Rectified Linear Unit

- ReLU is the most common activation function in deep learning as it helps preventing what is called **gradient vanishing** problem
  - Gradients become so small and eventually become zeros

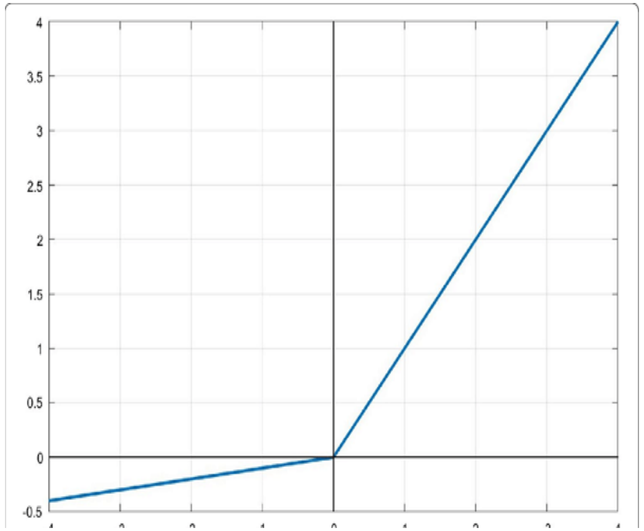
$$ReLU\ (x) = \begin{cases} 0, & if\ x < 0 \\ x, & if\ x\ \geq 1 \end{cases} \qquad ReLU'\ (x) = \begin{cases} 0, & if\ x < 0 \\ 1, & if\ x\ \geq 1 \end{cases}$$



leaky ReLU

- the standard ReLU suffers from a problem called dead neuron dying ReLU
  - this happens when the input to ReLU is negative
- So instead of zeroing negative elements we use a small scaler for negative values

$$ReLU\ (x) = \begin{cases} \alpha x, & if\ x < 0 \\ x, & if\ x\ \geq 1 \end{cases} \qquad ReLU'\ (x) = \begin{cases} \alpha, & if\ x < 0 \\ 1, & if\ x\ \geq 1 \end{cases}$$



Softmax

- Softmax is another activation function that is typically used on the output layer of multiclass classification tasks.
- It transforms the raw output logits from the network into a **probability distribution**, where each class is assigned a probability, and the sum of probabilities across all classes equals 1
  - For a problem of n classes, the Softmax is given by

$$Softmax\,(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_i}}$$

- The derivative of Softmax is tricky, will discuss it later on. For this reason usually the softmax is combined with a type of loss called **cross-entropy loss** to simplify its derivative

Python code

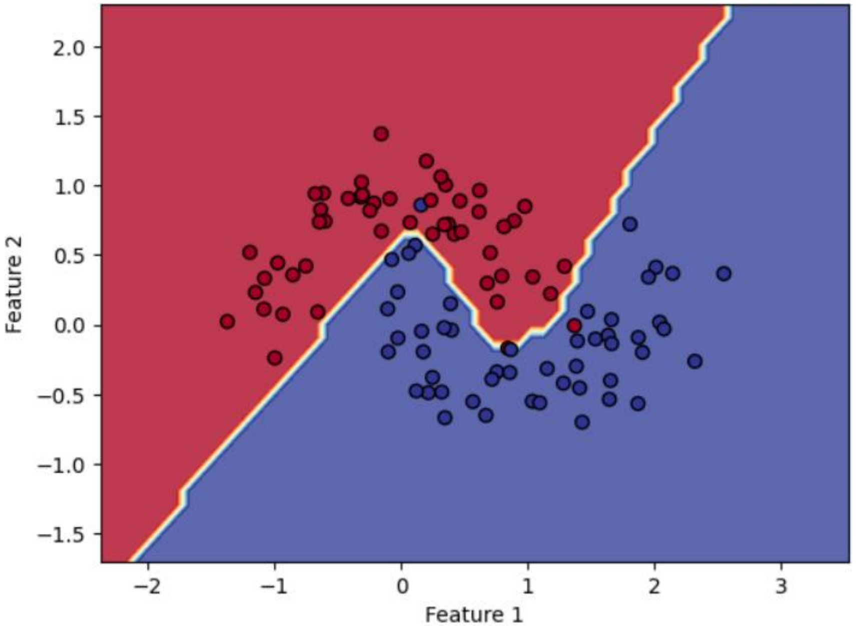
- Now let us add this activation function to our implementation and see how the decision boundary looks

```
class ActivationRelu:
    def forward(self, inputs):
        self.inputs = inputs
        self.output = np.maximum(0, inputs)
        return self.output

    def backward(self, dvalues):
        self.dinputs = dvalues * np.where(self.inputs > 0, 1, 0)


class ActivationSig:
    def forward(self, inputs):
        self.inputs = inputs
        self.output = 1 / (1 + np.exp(-inputs))
        return self.output

    def backward(self, dvalues):
        # Derivative of Sigmoid function
        self.dinputs = dvalues * (self.output * (1 - self.output))
```

- Also, we need to increase the number of layers and nodes to make the MLP able to fit more complex data

Example with ReLU

```python
layer1 = Dense(2, 10)
activation1 = ActivationRelu()

layer2 = Dense(10, 10)
activation2 = ActivationRelu()

layer3 = Dense(10, 2)
activation3 = ActivationRelu()

# Training parameters
alpha = 0.1
epochs = 1000
```

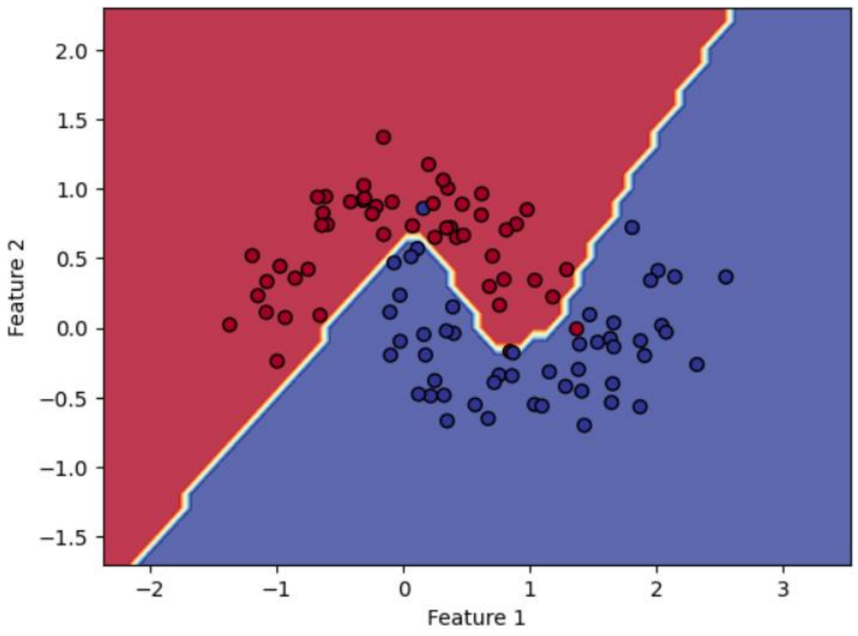



```python
# Training loop
for epoch in range(epochs):
    loss_epoch = 0
    for i in range(len(X)):
        # Get one sample and one-hot label
        x_sample = np.expand_dims(X[i], axis=0)
        y_sample = np.expand_dims(y_onehot[i], axis=0)

        # Forward pass
        h1 = layer1.forward(x_sample)
        h1 = activation1.forward(h1)
        h2 = layer2.forward(h1)
        h2 = activation2.forward(h2)
        o = layer3.forward(h2)
        o = activation3.forward(o)
        # Compute loss
        loss = mse_loss(y_sample, o)
        loss_epoch += loss

        # Backward pass
        dL_do = mse_loss_derivative(y_sample, o)
        activation3.backward(dL_do)
        layer3.backward(activation3.dinputs)
        activation2.backward(layer3.dinputs)
        layer2.backward(activation2.dinputs)
        activation1.backward(layer2.dinputs)
        layer1.backward(activation1.dinputs)
        # Update weights and biases
        layer1.weights -= alpha * layer1.dweights
        layer1.biases -= alpha * layer1.dbiases
        layer2.weights -= alpha * layer2.dweights
        layer2.biases -= alpha * layer2.dbiases
        layer3.weights -= alpha * layer3.dweights
        layer3.biases -= alpha * layer3.dbiases
    # Print average loss for the epoch
    if epoch % 10 == 0:
        print(f"Epoch {epoch}, Loss: {loss_epoch / len(X)}")
```

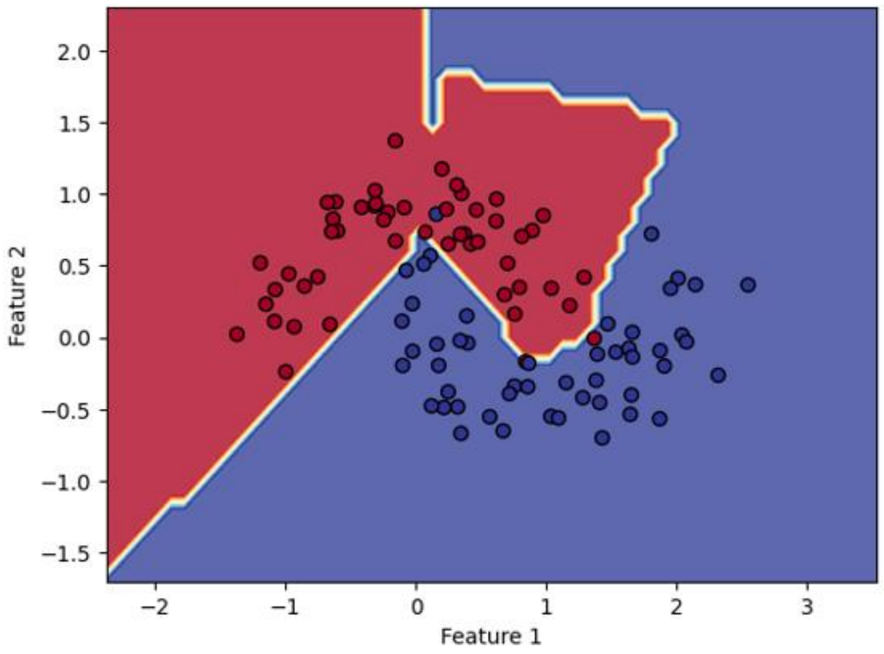
Visualization code

```
def plot_decision_boundary():
    # Define a grid for the contour plot
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                         np.arange(y_min, y_max, 0.1))

    grid_points = np.c_[xx.ravel(), yy.ravel()]
    h1 = layer1.forward(grid_points)
    h1 = activation1.forward(h1)

    h2 = layer2.forward(h1)
    h2 = activation2.forward(h2)

    o = layer3.forward(h2)

    Z = np.argmax(o, axis=1).reshape(xx.shape)

    plt.contourf(xx, yy, Z, levels=25, cmap="RdYlBu", alpha=0.8)
    plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap="RdYlBu", edgecolor="k")
    plt.title("Decision Boundary with 3-Layer Network")
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.show()
```

Using ReLU



Using Softmax



Loss functions

- The loss function we were using so far is the MSE loss, which is usually works for regression tasks
  - Although it can work for classification tasks
- There are better loss functions designed for classification tasks, such as
  - MSE: Regression
  - MAE: Regression
  - binary cross entropy: two classes
  - categorical cross entropy: labels are one-hot-encoded integers
  - sparse categorical cross entropy: labels are integers
  - and many more …

Cross entropy

- for a classification problem with y one-hot-encoded, The formula for **Cross Entropy Loss** in is

$$CELoss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{i,j}\log\left(\hat{y}_{i,j}\right)$$

where:

- N is the number of examples
- C is the number of classes
- $y_{i,j}$ is the true label for sample i and class j, where $y_{i,j} = 1$ if the class is the true class for the sample, and 0 otherwise.
- $\hat{y}_{i,j}$ is the predicted probability (or softmax output) for sample i and class j.

- For binary classification

$$\text{Loss} = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right]$$

Categorical cross entropy with Softmax

- Derivative of cross entropy is tricky, but combining it with Softmax simplifies implementation

$$\frac{\partial \text{Loss}}{\partial z_i} = \hat{y}_i - y_i$$

***i* is the class in the one-hot vector**

```
class SoftmaxCrossEntropy:

    def forward(self, logits, y_true):
        exp_values = np.exp(logits - np.max(logits, axis=1, keepdims=True))
        self.y_pred = exp_values / np.sum(exp_values, axis=1, keepdims=True)

        self.y_pred = np.clip(self.y_pred, 1e-7, 1 - 1e-7)
        # Compute cross-entropy loss
        loss = -np.sum(y_true * np.log(self.y_pred)) / y_true.shape[0]  # Average over batch
        return loss

    def backward(self, y_true):
        # Gradient of the loss w.r.t logits
        grad = self.y_pred - y_true
        return grad
```

-np.max(logits)) is used to avoid very large and very low numbers