Lecture 1 : Artificial Neural Networks 🌱

> **Class**: ANN
>
> **Date**: 07.58 AM, 📅 Today
>
> **Author**: Lasal Hettiarachchi

Key learnings:

- ANN
- Forward Pass
- Backward Pass

Real World Example

You go on shopping for a new Laptop. What are the factors you base your decision on? Assume all of these are binary variables

- The Price (Xp)
- Is it better than the Current Laptop. (Xb)
- Is the merchant reliable. (Xr)

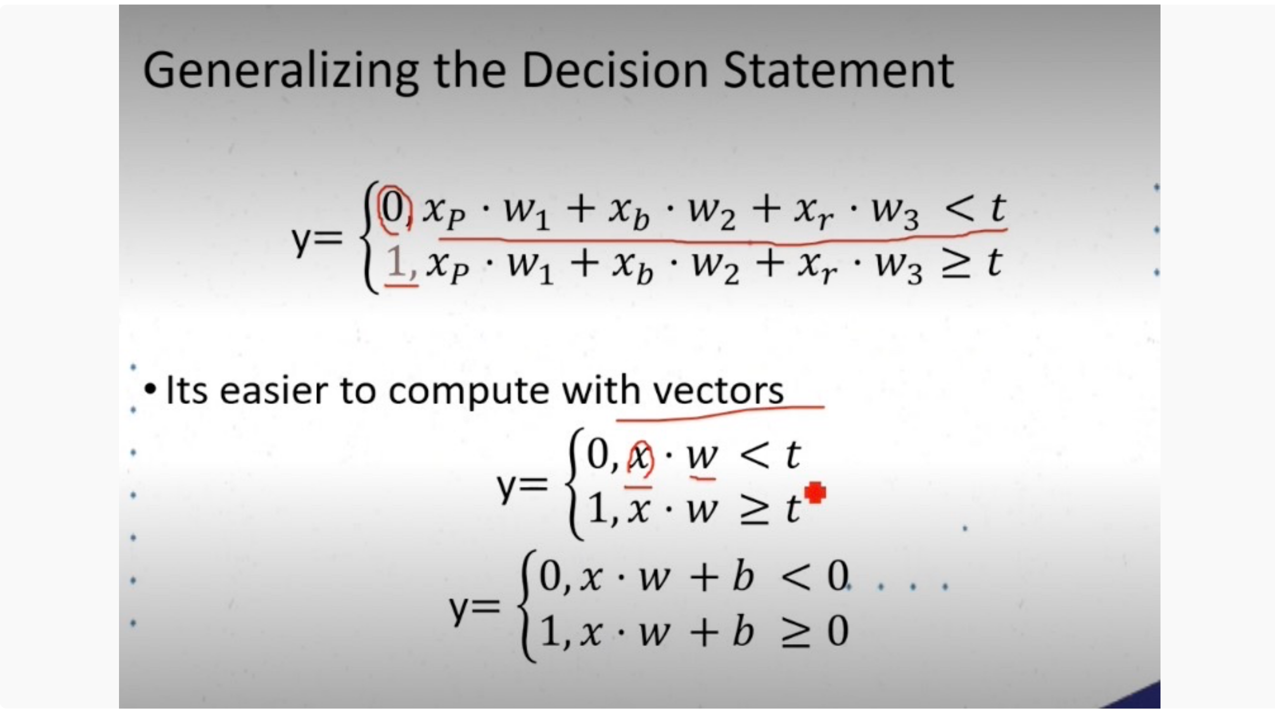
Decision

W1 Xp + W2 Xb + W3 Xr = y

- if y > 5 → yes
- if y < = 5 → no

The weightes can be adjusted to make the decision

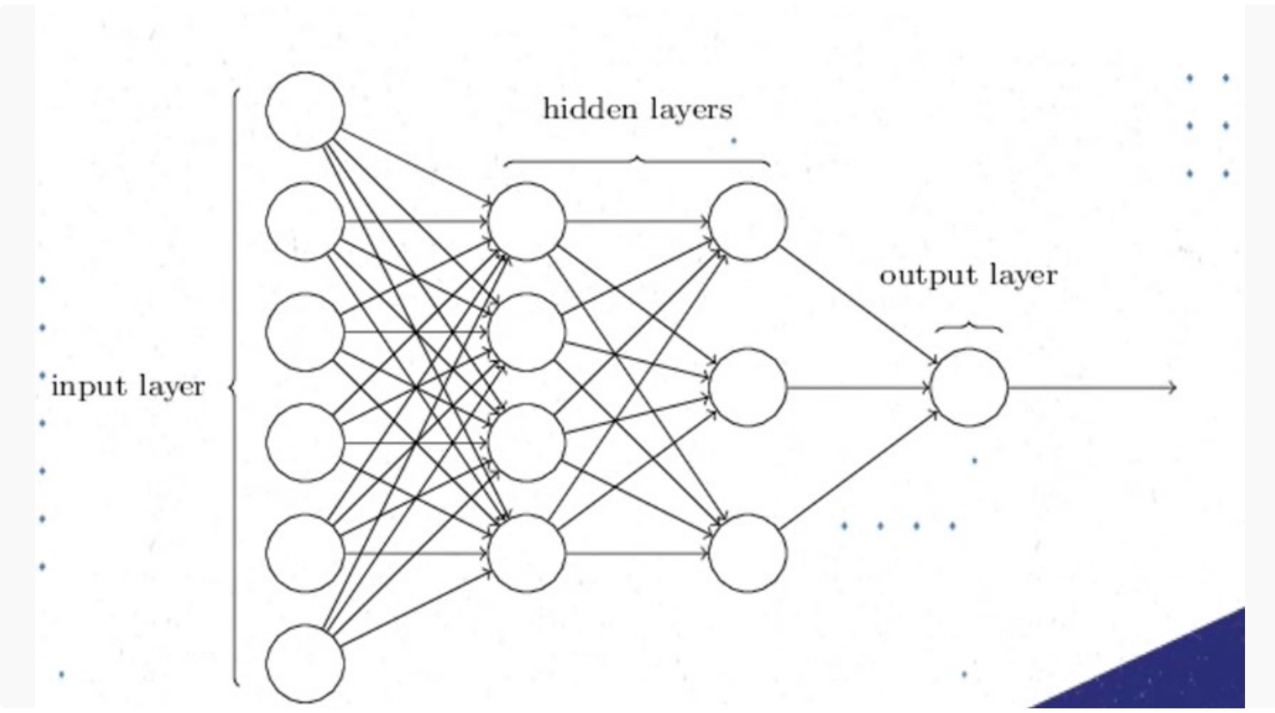
created with craft



- We take the t and make it as a bias (bias is nothing but the negative of the threshhold value)
- From this intuition we create an artificial neuron which takes in multiple features (xp,xb,xr / X) and bias b to output the vector y

What is an ANN?

- Collection of neuron is called an ANN
- When we stack neurons together , we get a layer



Activation Function

created with craft

- Whats the purpose of activation functions?
  - Output of a perceptron is always binary, this is not good always
  - Linear outputs are useful but difficult to learn complex data
  - Non linear complex values are thus required
  - **To add non linearity**
- It is possible to transform any value between 0-1 using a linear activation function
- But if the output is linear, there isnt gonna be much change from layer to layer

$$a^{[1]} = z^{[1]} = W^{[1]} x + b^{[1]}$$

$$a^{[2]} = z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$$

$$a^{[2]} = W^{[2]} \left(\underbrace{W^{[1]} x + b^{[1]}}_{a^{[1]}} \right) + b^{[2]}$$

$$= \underbrace{\left(W^{[2]} W^{[1]} \right)}_{W'} x + \underbrace{\left(W^{[2]} b^{[1]} + b^{[2]} \right)}_{b'}$$

$$= W' x + b'$$

- This will just output a linear function of the input (**All layers of the neural network collapse into one —** with linear activation functions, no matter how many layers in the neural network, the last layer will be a linear function of the first layer)
- We cannot do back propergation aswell
- This helps the network to understand more complex datasets
- Helps normalize the outputs of the specific NN layer

Examples:

  - Sigmoid (This is used in output layers mostly to get values between 0 and 1)
  - tanh (This is better than sigmoid since the mean is 0, this centers the data to have 0 mean)

created with craft

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

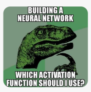
Issue with sigmoid and tanh: When x is very large of very small. The derivative is very small. This can slow down gradient decent

- Relu
- Leaky Relu

The advantage is that for most of the values other than 0, the derivative is not 0 thus gradient decent works well.

Properties of a good activation function

- Func should be continous and differentiable everywhere
- Derivative should not saturate(tend toward 0) over its expected input range , since this may stall learning
- Derivatives should not explode. (This may cause neumerical instabillity and pivot on a single layer)



What, Why and Which?? Activation Functions
While studying neural network, we often come across the term—“Activation functions”. What are these activation functions?
https://medium.com/@snaily16/what-why-and-which-activation-functions-b2bf748c0441

Function	Advantage	Disadvantage
Sigmoid	The function is **differentiable**.That means, we can find the slope of the sigmoid curve at any two points.	**Vanishing gradient —** for very high or very low values of X, there is almost no change to the prediction, causing a vanishing gradient problem.
	Output values bound between 0 and 1, normalizing the output of each neuron.	Due to vanishing gradient problem, sigmoids have **slow convergence.**
		Outputs not zero centered.
		Computationally expensive.

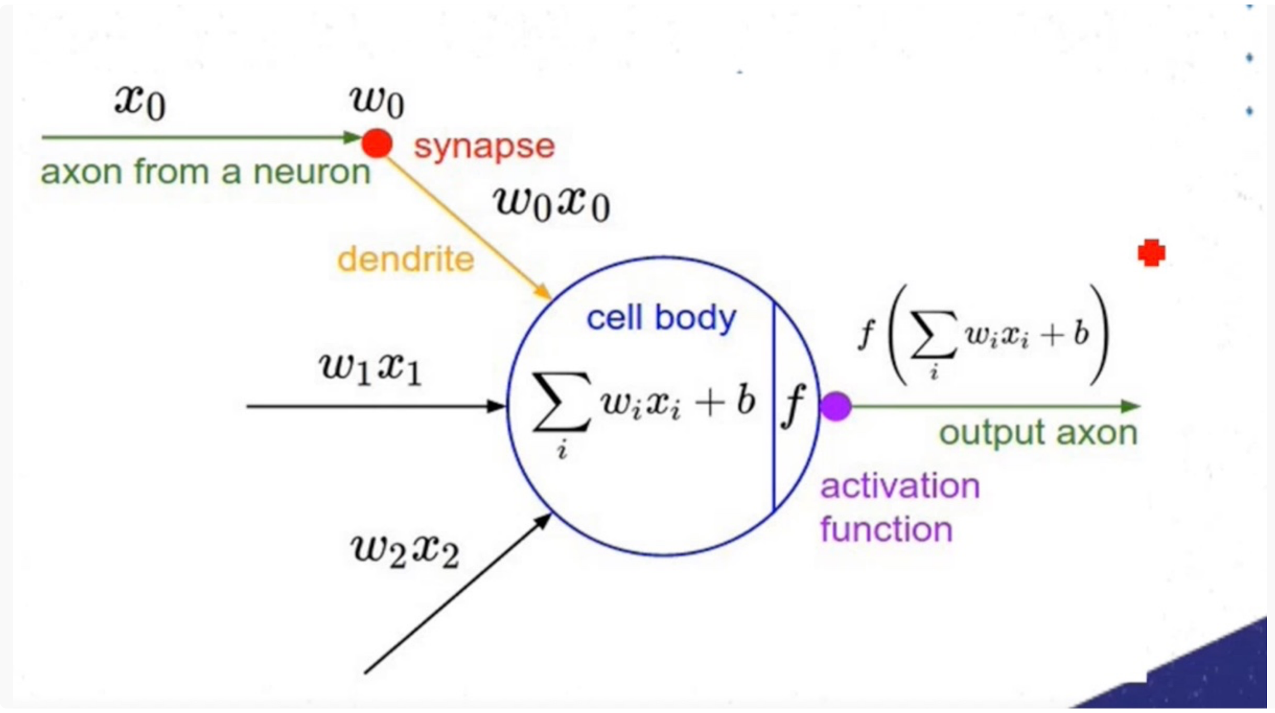
Function	Advantage	Disadvantage
Tanh	**Zero centered** — making it easier to model inputs that have strongly negative, neutral, and strongly positive values.	It also suffers **vanishing gradient problem** and hence **slow convergence.**
	The function and its **derivative** both are **monotonic.**	
	Works better than sigmoid function	

Function	Advantage	Disadvantage
Relu	**Computationally efficient** — allows the network to converge very quickly	**The Dying ReLU problem** — when inputs approach zero, or are negative, the gradient of the function becomes zero, the network cannot perform back-propagation and cannot learn.
	Non-linear — although it looks like a linear function, ReLU has a derivative function and allows for back-propagation	

Function	Advantage	Disadvantage
Leaky Relu	**Prevents dying ReLU problem** — this variation of ReLU has a small positive slope in the negative area, so it does enable back-propagation, even for negative input values	**Results not consistent** — leaky ReLU does not provide consistent predictions for negative input values.
		During the front propagation if the learning rate is set very high it will overshoot killing the neuron.

Perceptron architecture

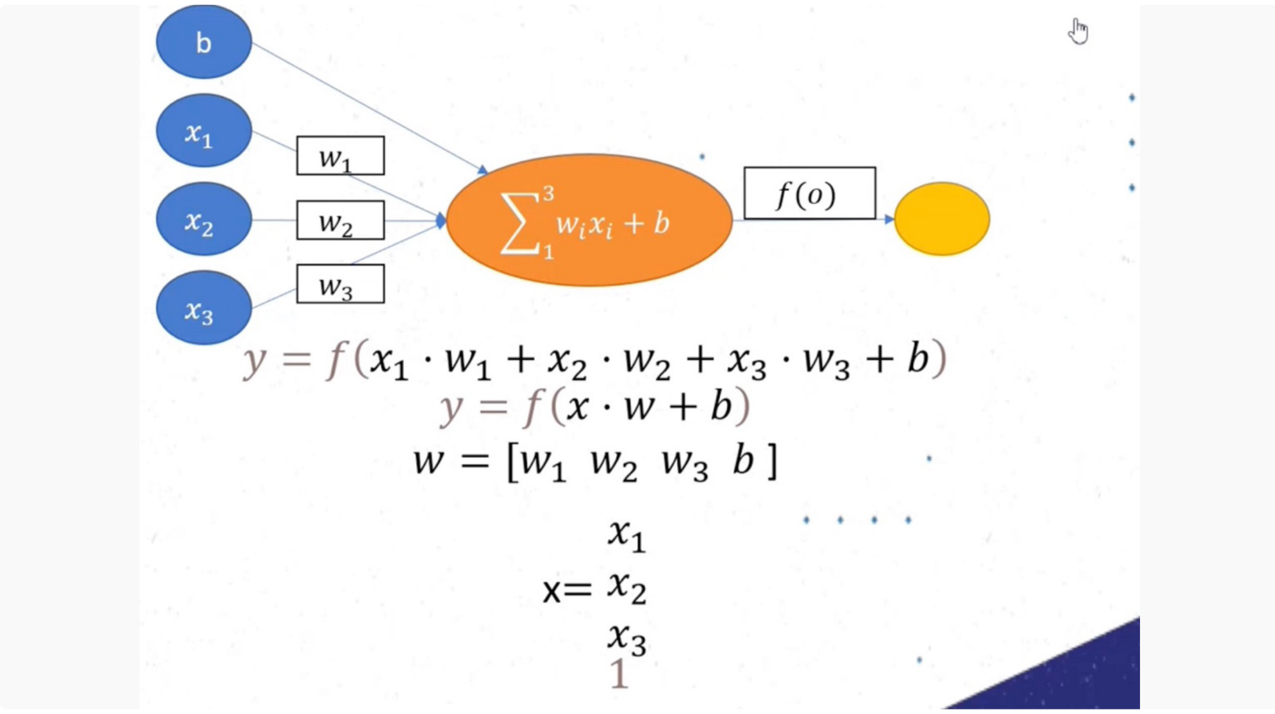
created with craft



Feed forward NN

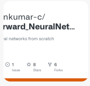
- Most ANNs are Single flow, uni directional
- No single loops or backtracking
    - CNN
    - MLP

Forward pass



Implementing the feed forward NN

Before we start training the data on the sigmoid neuron, We will build our model inside a class called *SigmoidNeuron*.



Feedforward_NeuralNetworrks/FeedForwardNeuralNetwork.ipynb at master · Niranjankumar-
Build our neural networks from scratch. Contribute to Niranjankumar-c/Feedforward_NeuralNetworrks development by creati...
https://github.com/Niranjankumar-c/Feedforward_NeuralNetworrks/blob/master/FeedForwardNetworks/FeedForwardNeural...

```
class SigmoidNeuron:
  #intialization
  def __init__(self):
    self.w = None
    self.b = None
  #forward pass
  def perceptron(self, x):
    return np.dot(x, self.w.T) + self.b

  def sigmoid(self, x):
    return 1.0/(1.0 + np.exp(-x))
  #updating the gradients using mean squared error loss
  def grad_w_mse(self, x, y):
    y_pred = self.sigmoid(self.perceptron(x))
    return (y_pred - y) * y_pred * (1 - y_pred) * x

  def grad_b_mse(self, x, y):
    y_pred = self.sigmoid(self.perceptron(x))
    return (y_pred - y) * y_pred * (1 - y_pred)

  #updating the gradients using cross entropy loss
  def grad_w_ce(self, x, y):
    y_pred = self.sigmoid(self.perceptron(x))
    if y == 0:
      return y_pred * x
    elif y == 1:
      return -1 * (1 - y_pred) * x
    else:
      raise ValueError("y should be 0 or 1")

  def grad_b_ce(self, x, y):
    y_pred = self.sigmoid(self.perceptron(x))
    if y == 0:
      return y_pred
    elif y == 1:
      return -1 * (1 - y_pred)
    else:
      raise ValueError("y should be 0 or 1")
```

created with craft

```
#model fit method
  def fit(self, X, Y, epochs=1, learning_rate=1, initialise=True, loss_fn="mse",
display_loss=False):

    # initialise w, b
    if initialise:
      self.w = np.random.randn(1, X.shape[1])
      self.b = 0

    if display_loss:
      loss = {}

    for i in tqdm_notebook(range(epochs), total=epochs, unit="epoch"):
      dw = 0
      db = 0
      for x, y in zip(X, Y):
        if loss_fn == "mse":
          dw += self.grad_w_mse(x, y)
          db += self.grad_b_mse(x, y)
        elif loss_fn == "ce":
          dw += self.grad_w_ce(x, y)
          db += self.grad_b_ce(x, y)

      m = X.shape[1]
      self.w -= learning_rate * dw/m
      self.b -= learning_rate * db/m

      if display_loss:
        Y_pred = self.sigmoid(self.perceptron(X))
        if loss_fn == "mse":
          loss[i] = mean_squared_error(Y, Y_pred)
        elif loss_fn == "ce":
          loss[i] = log_loss(Y, Y_pred)

    if display_loss:
      plt.plot(loss.values())
      plt.xlabel('Epochs')
      if loss_fn == "mse":
        plt.ylabel('Mean Squared Error')
      elif loss_fn == "ce":
        plt.ylabel('Log Loss')
      plt.show()

  def predict(self, X):
    Y_pred = []
    for x in X:
      y_pred = self.sigmoid(self.perceptron(x))
      Y_pred.append(y_pred)
    return np.array(Y_pred)
```

- The weight values are initialized randomly and the bias with 0

```
if initialise:
      self.w = np.random.randn(1, X.shape[1])
      self.b = 0
```

- Mean squared error with respect to w and updating gradients using cross entropy loss

```
 #updating the gradients using mean squared error loss
  def grad_w_mse(self, x, y):
    y_pred = self.sigmoid(self.perceptron(x))
    return (y_pred - y) * y_pred * (1 - y_pred) * x

#updating the gradients using cross entropy loss
  def grad_w_ce(self, x, y):
    y_pred = self.sigmoid(self.perceptron(x))
    if y == 0:
      return y_pred * x
    elif y == 1:
      return -1 * (1 - y_pred) * x
    else:
      raise ValueError("y should be 0 or 1")
```

- Mean squared error with respect to b and updating gradients using cross entropy loss

```
def grad_b_mse(self, x, y):
    y_pred = self.sigmoid(self.perceptron(x))
    return (y_pred - y) * y_pred * (1 - y_pred)

def grad_b_ce(self, x, y):
    y_pred = self.sigmoid(self.perceptron(x))
    if y == 0:
      return y_pred
    elif y == 1:
      return -1 * (1 - y_pred)
    else:
      raise ValueError("y should be 0 or 1")
```

Loss function (Cost)

- Parameter that measures how well the NN is doing
- Goal is to find w , b that minimizes the loss
- No of different loss functions are available
    - MSE
    - RSME
    - Hinge Loss
- This is the different between the expected label in the training set and the predicted value

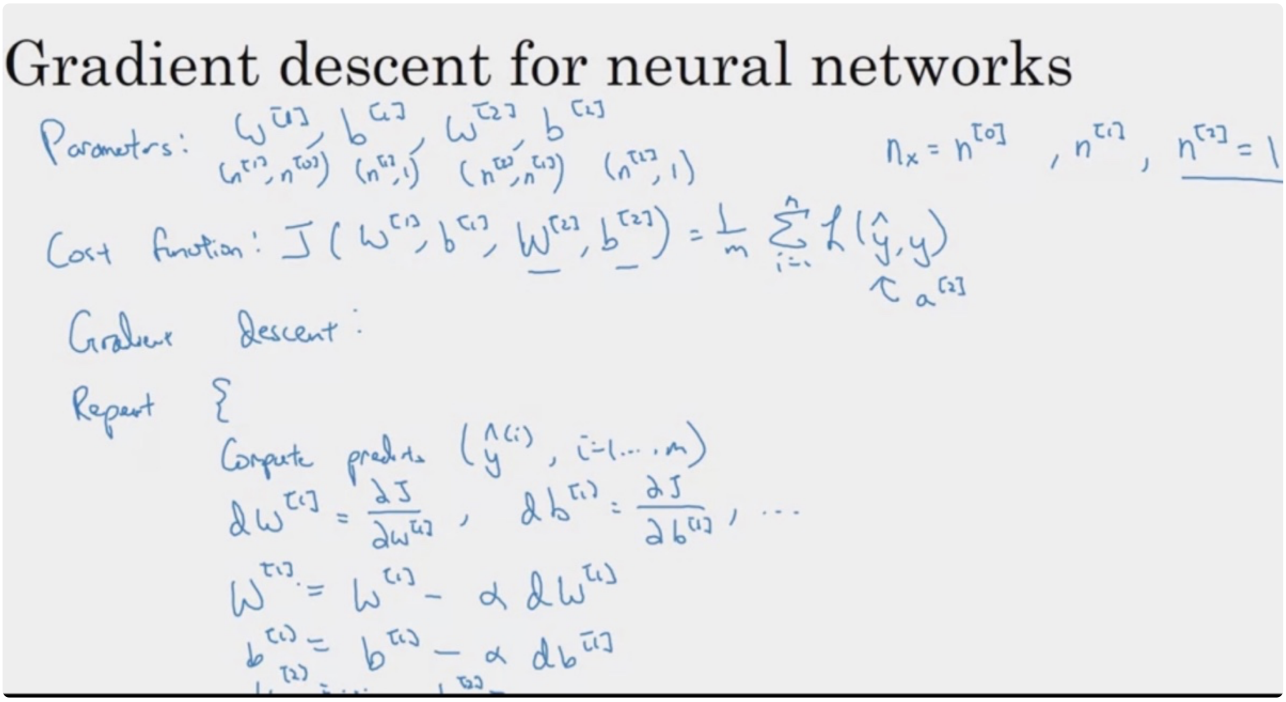
Loss with respect to w can be calculated by when the function is sigmoid,

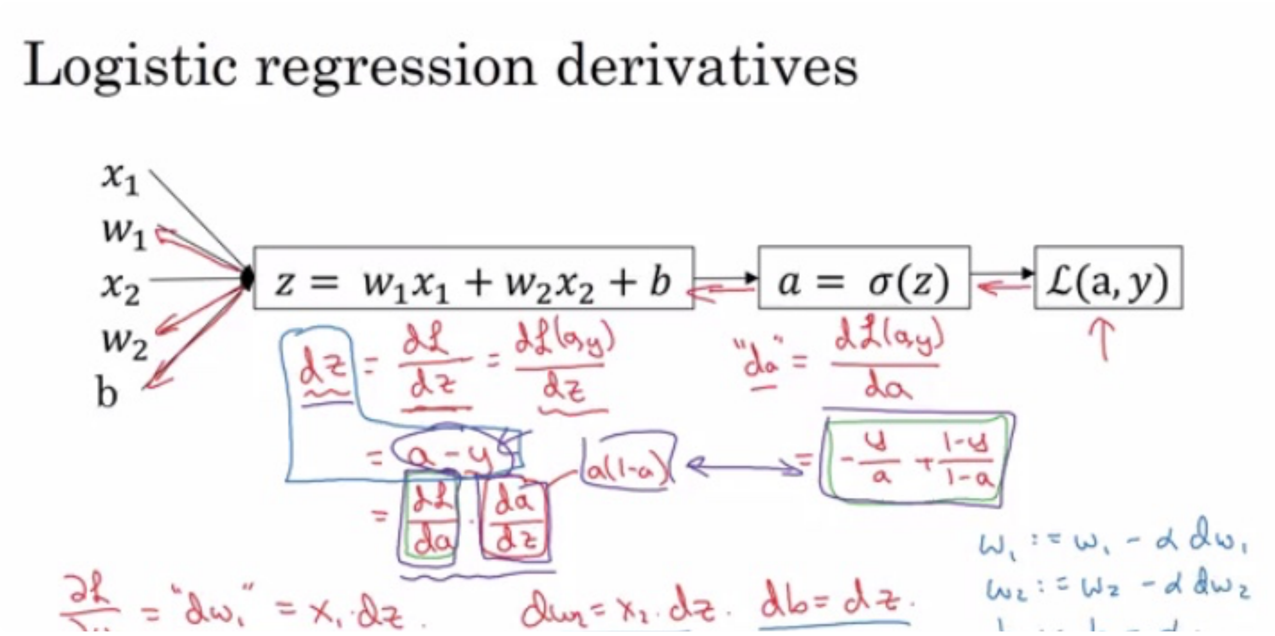
created with craft

$$
\begin{aligned}
\nabla w &= \frac{\partial}{\partial w}[\frac{1}{2} * (f(x)-y)^2] \\
&= \frac{1}{2} * [2 * (f(x)-y) * \frac{\partial}{\partial w}(f(x)-y)] \\
&= (f(x)-y) * \frac{\partial}{\partial w}(f(x)) \\
&= (f(x)-y) * \frac{\partial}{\partial w}\left(\frac{1}{1+e^{-(wx+b)}}\right) \\
&= (f(x)-y) * f(x) * (1-f(x)) * x
\end{aligned}
$$

$$
\begin{aligned}
&\frac{\partial}{\partial w}\left(\frac{1}{1+e^{-(wx+b)}}\right) \\
&= \frac{-1}{(1+e^{-(wx+b)})^2}\frac{\partial}{\partial w}(e^{-(wx+b)})) \\
&= \frac{-1}{(1+e^{-(wx+b)})^2} * (e^{-(wx+b)})\frac{\partial}{\partial w}(-(wx+b))) \\
&= \frac{-1}{(1+e^{-(wx+b)})} * \frac{e^{-(wx+b)}}{(1+e^{-(wx+b)})} * (-x) \\
&= \frac{1}{(1+e^{-(wx+b)})} * \frac{e^{-(wx+b)}}{(1+e^{-(wx+b)})} * (x) \\
&= f(x) * (1-f(x)) * x
\end{aligned}
$$

- W[1] and b[1] are weight and bias vectors of hidden layer 1 while , W[2] and b[2] are weight and bias vectors of hidden layer 2.
- We use gradient decent to update the Vectors





- L is the loss. we need to minimize loss WRT w1,w2,b

Process of an FF NN

1. Randomly initialize weights
2. Implement forward propagation
3. Compute error total Etotal
4. Implement back propagarion dab (Etotal) / dab (Wjk)
5. Use gradient decent or any other optimization technique to update the weights
6. Repeat the process over multiple iterations or error converges

created with craft