Calculus

We can use sympy for symbolic math and symbolic differentiation

Symbolic differentiation does have some limitations because we can get very complicated functions as output of symbolic computation (expression swell)

We can also use numerical differentiation. This does not take into account the function expression, what's important is that the function can be evaluated in nearby points x and $x + \Delta x$. Where Δx is sufficiently small.

Then $\frac{df}{dx} \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}$ which can be called a numerical approximation of the derivative. There are limitations of course which one of the is that numerical differentiation is not exact (accuracy is normally strong enough for machine learning applications) and the second is similar to the error which appeared for symbolic differentiation, it is inaccurate at the points where there are jumps of the derivative. But the biggest problem with numerical differentiation is slow speed. It requires function evaluation every time. in ML models there are hundreds of parameters and there are hundreds of derivatives to be calculated, performing full function evaluation every time slows down the computation process.

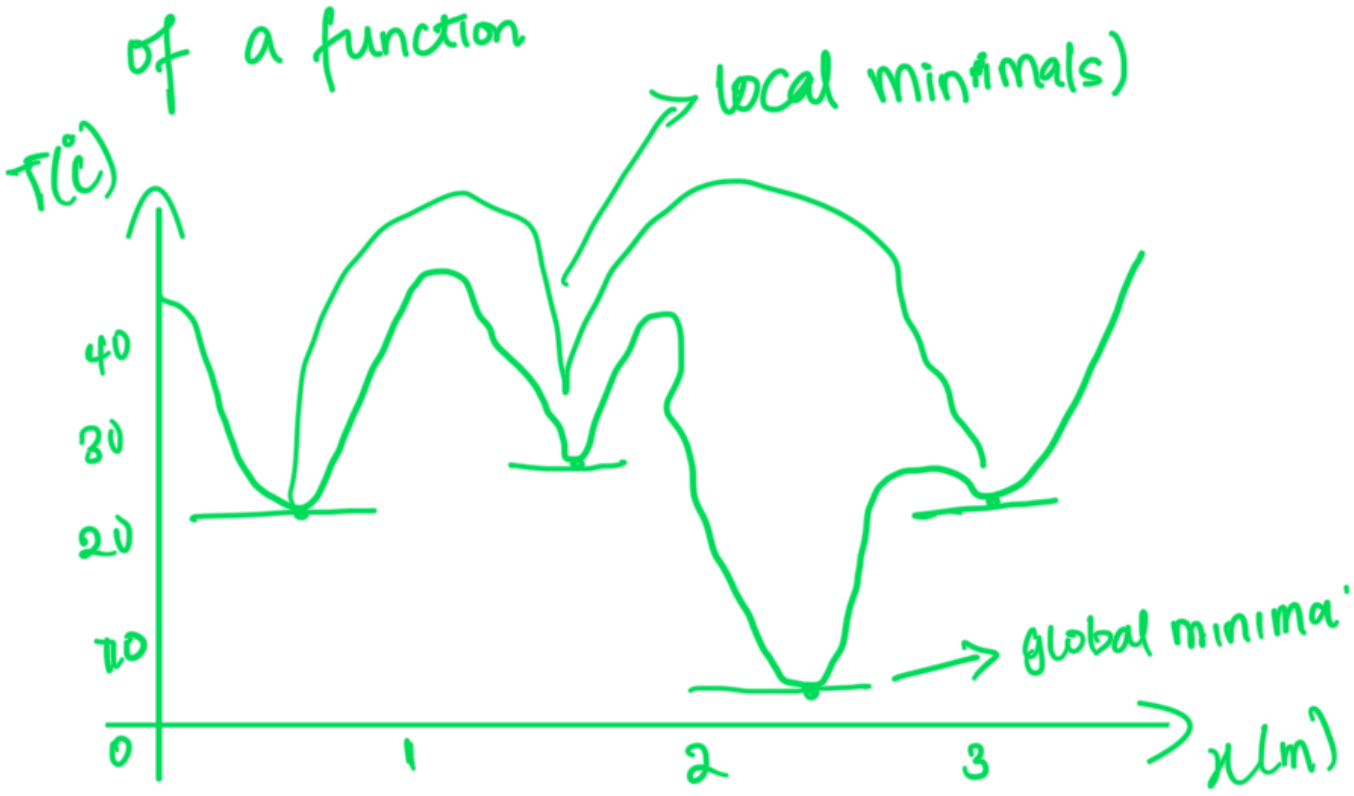
Automatic differentiation (autodiff) breaks down the function into common functions (sin, cos, log, power functions, etc) and constructs the computational graph consisting of the basis functions. Then the chain rule is used to compute the derivative at any node of the graph. It is the most commonly used approach in ML applications and neural networks, as the computational graph for the function and its derivates can be built during construction of the neural network, ready in future computations. The main disadvantage of it is

neural network, saving in future

Implementation difficulty. But libraries like My Grad, Autograd and Jax makes it convienent to use. Most used are AutoGrad and JAX, JAX brings together AutoGrad functionality for optimizatun problems and XLA (Accelerated Linear Algebra) compiler for parallel computing.

Optimization

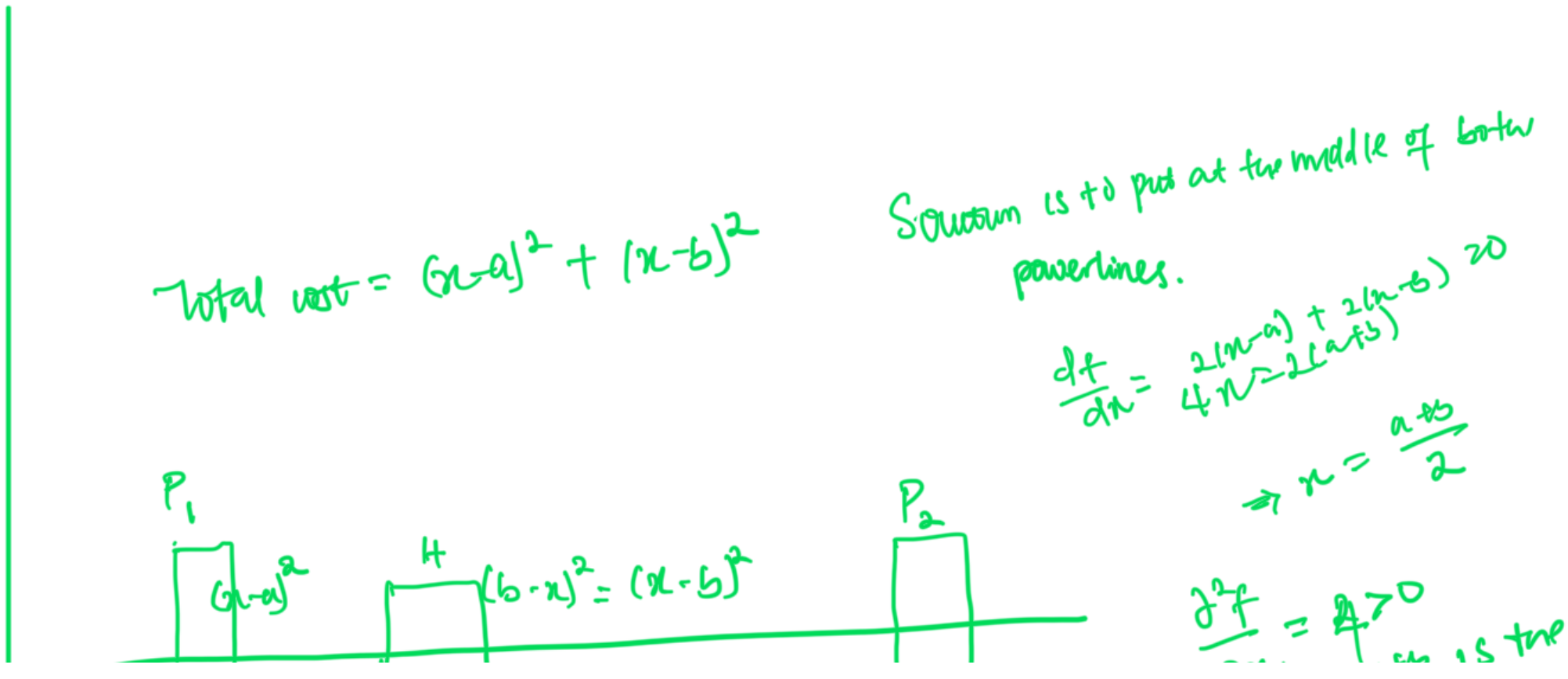
Optimization is when we want to find maximum and minimum value of a function

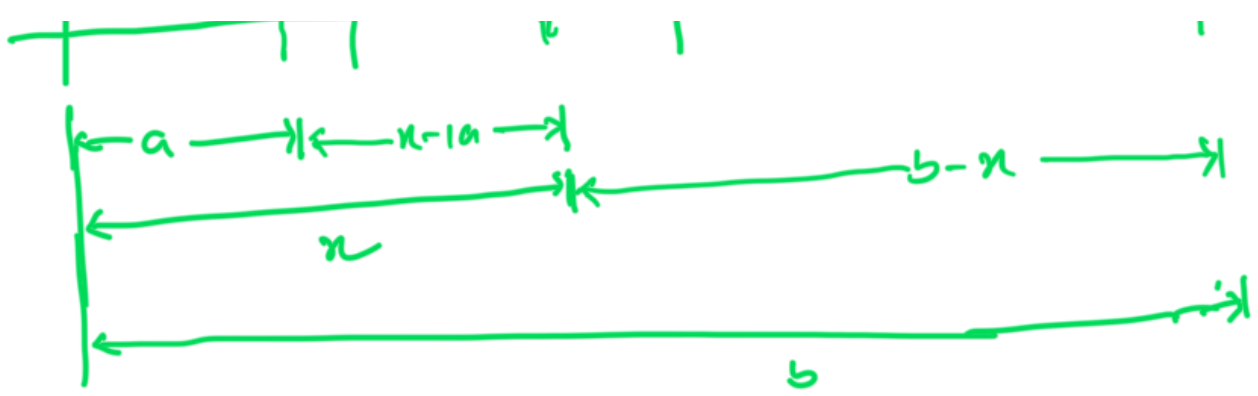


→ Squared loss (Powerline problem)

One powerline case ⇒ the distance is zero, just put it where powerline is located.

Two power lines





hence $x = \frac{a+b}{2}$ minimum

Three powerlines

third at distance c.

Total cost $= (x-a)^2 + (x-b) + (x-c)$

$$\frac{\partial f}{\partial x} = 2(x-a) + 2(x-b) + 2(x-c) = 0 \Rightarrow x = \frac{a+b+c}{3}$$

$$\frac{\partial^2 f}{\partial x^2} = 4 > 0 \text{ (minimum point)} \quad , \quad x = \frac{a+b+c}{3}$$

$$f = (x-a_1)^2 + (x-a_2)^2 + \cdots (x-a_n)^2 \quad , \quad x_{min} = \frac{a_1+a_2+\cdots+a_n}{n}$$

$\longrightarrow$ Log loss

$$g(p) = p^7(1-p)^3 \quad , \quad \frac{dg}{dp} = -3p^7(1-p)^2 + 7p^6(1-p)^3 = 0$$

$$\Rightarrow p^6(1-p)^2(-10p+7) = 0$$

$p = 0 \; , \; p = 1$ and $p = 7/10$

$$\frac{d^2g}{dp^2} = 6p^7(1-p) - 21p^6(1-p)^2 - 21p^6(1-p)^2 + 42p^5(1-p)^3$$

$$= 6p^7(1-p) - 42p^6(1-p)^2 + 42p^5(1-p)^3$$

$$= 6p^5(1-p)(p^2 - 42p + 42p^2 + 42 - 84p + 42p^2)$$

$$= 6p^5(1-p)(85p^2 - 124p + 42)$$

at $p \approx 0$, $\frac{d^2g}{dp^2} \approx 0$

at $p \approx 1$, $\frac{d^2g}{dp^2} \approx 0$

$\frac{d^2g}{dp^2} < 0 \Rightarrow p = 7/10$ is the max point.

at $p = \frac{7}{10}$, $\frac{d^2}{dp^2}$

Lot of work .... easier way???

yeah, take $\log(g(p))$ because if $g(p)$ is maximal, so is its log

$$\log(g(p)) = \log\left(p^7(1-p)^3\right) = \log(p^7) + \log((1-p)^3) = 7\log p + 3\log(1-p) = G(p)$$

$$\frac{dG}{dp} = \frac{7}{p} - \frac{3}{1-p} = \frac{7-10p}{p(1-p)} = 0 \Rightarrow 7-10p = 0 \Rightarrow p = 7/10.$$

This is a pretty popular trick in machine learning

- $G(p)$ is called the log loss, we will see it a lot in classification problems and the reason that we take negative of $G(p)$ instead of $G(p)$ is that logarithm of p is actually a negative number when p is between 0 and 1. So we want $-G(p)$ to be a positive number and instead of maximizing it like we did earlier, we aim to minimize it.

→ Relationship with ML.

We basically did machine learning with the coin flips earlier. The model is a coin that lands in heads with probability p and tails with probability $1-p$. The dataset was the 10 coin flips which are seven heads followed by 3 tails. So what we did was finding the model that most likely fits the dataset. We find the model by minimizing the log loss and obtain that the optimal p is 7/10.

Why logarithm?

→ Derivative of a product is hard, derivative of sum is easy.

→ Product of lots of tiny things is tiny. Like 1000 products of very small numbers

Gradients and Gradients Descents

→ Tangent Plane

$f(x,y) = x^2 + y^2$

$\frac{\partial f}{\partial x} = 2x, \frac{\partial f}{\partial y} = 2y$ → Tangent plane contains both tangent lines.

Partial derivatives

→ Gradient describes the slope of the lines that form the tangent.

$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$, $f(x,y) = x^2 + y^2$, $\nabla f = \begin{bmatrix} 2x \\ 2y \end{bmatrix}$

→ Minima and Maxima

$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} = 0 \Rightarrow \nabla f = 0$

→ Optimization using gradients – Analytical method.

Linear regression is when you have a bunch of points and you're looking for the line of best fit (closest line to all of them)

→ Optimization using Gradient Descent-

1 Variable

$f(x) = e^x - \log x$, $\frac{df}{dx} = e^x - \frac{1}{x} = 0 \Rightarrow e^x = 1/x$, hard to solve analytically

move in both directions. Like pick a random value say 0.1, move

Method 1: [illegible]

left and right by a radndom number, say 0.1, 0.2 and 0.3

We keep moving but both of our moves are on the same side

Method 2: Be clever

more based on the sign of the slope at that point, of you want a new point

new point = old point - slope.

$$x_{n+1} = x_n - f'(x_n)$$

To avoid an arbitrary large jump, we add a small number called the learning rate

$$x_{n+1} = x_n - \alpha f'(x_n) \rightarrow \text{Gradient Descent.}$$

Algorithm

Function: $f(x)$ Goal: find minimum of $f(x)$

Step 1: Define a learning rate α

Choose a starting point x_0

Step 2: Update $x_k = x_{k-1} - \alpha f'(x_{k-1})$

Step 3: Repeat step 2 until you are close enough to the true minimum.

It's a research problem to find good learning rate....

We can also be stuck in a local minima when using gradient descent based on our starting point, a way to mitigate this is to have many starting point.

2 variables

α variables

$$(x_n, y_n) = (x_{n-1}, y_{n-1}) - \alpha \nabla f(x_{n-1}, y_{n-1})$$

$$\begin{bmatrix} x_n \\ y_n \end{bmatrix} = \begin{bmatrix} x_{n-1} \\ y_{n-1} \end{bmatrix} - \alpha \nabla f(x_{n-1}, y_{n-1})$$

⟶ Least squares (power line problem)

We can still use gradient descent for this as well.

$y = mx + b$

$$\mathcal{L}(m, b) = \frac{1}{2n}\left[(mx_1 + b - y_1)^2 + \cdots + (mx_n + b - y_n)^2\right]$$

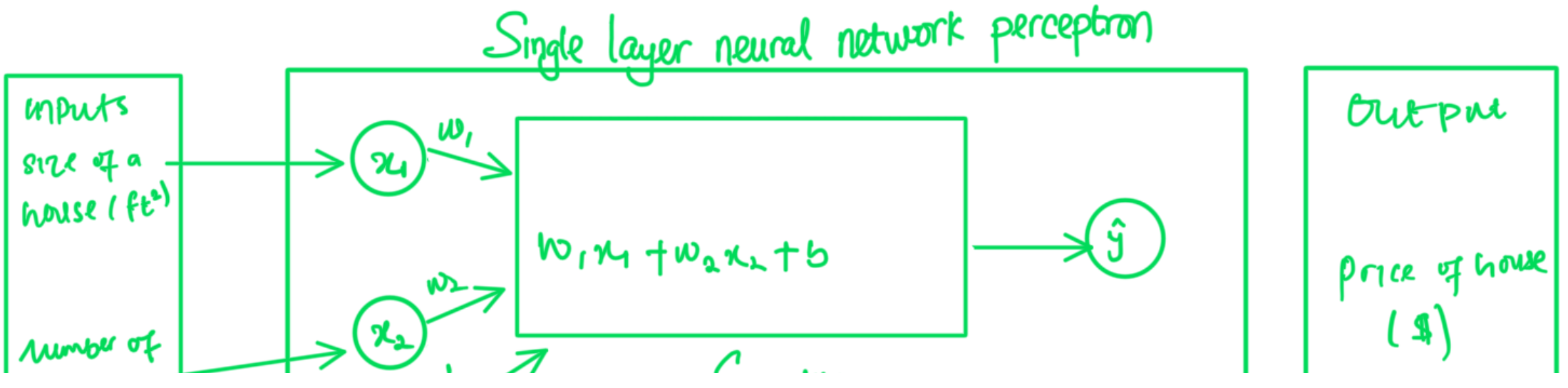
$$= \frac{1}{2n}\sum_{i=1}^{n}(mx_i + b - y_i)^2$$

$$\begin{bmatrix} m_0 \\ b_0 \end{bmatrix} \to \begin{bmatrix} m_1 \\ b_1 \end{bmatrix} = \begin{bmatrix} m_0 \\ b_0 \end{bmatrix} - \alpha \nabla \mathcal{L}_1(m_0, b_0)$$

$$\begin{bmatrix} m_k \\ b_k \end{bmatrix} \to \begin{bmatrix} m_k \\ b_k \end{bmatrix} = \begin{bmatrix} m_{k-1} \\ b_{k-1} \end{bmatrix} - \alpha \nabla \mathcal{L}_k(m_{k-1}, b_{k-1})$$

Optimization in Neural Networks and Newton's method

⟶ Regression with a perception



rooms

summation function

Main goal: Find the weights and bias that will optimize the predictions.
i.e reduce the errors in the predictions and how do we do that? We introduce a loss function.

⟶ Loss function

error = $y - \hat{y}$ and we square the error $(y-\hat{y})^2$

$\hat{y} = w_1x_1 + w_2x_2 + b$, $L(y,\hat{y}) = \frac{1}{2}(y-\hat{y})^2$

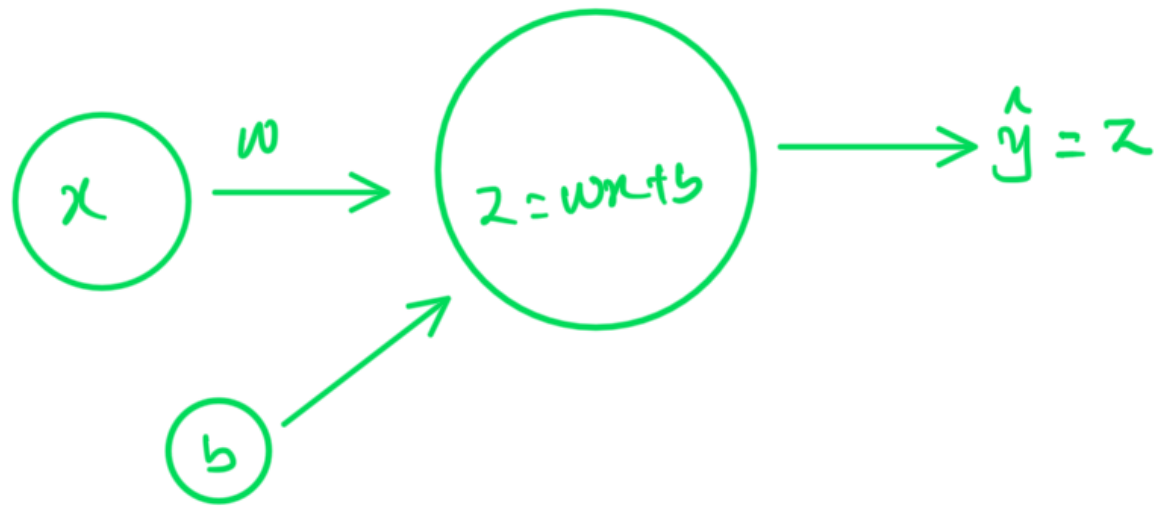
So we find w_1, w_2 and b that gives $\hat{y}$ with the least error

We use gradient descent for finding w_1, w_2 and b.

$\Rightarrow w_1 \to w_1 - \alpha \frac{\partial L}{\partial w_1}$, $\frac{\partial L}{\partial w_1} = -x_1(y-\hat{y})$, $w_1 \to w_1 + \alpha x_1(y-\hat{y})$

$w_2 \to w_2 - \alpha \frac{\partial L}{\partial w_2}$, $\frac{\partial L}{\partial w_2} = -x_2(y-\hat{y})$, $w_2 \to w_2 + \alpha x_2(y-\hat{y})$

$b \to b - \alpha \frac{\partial L}{\partial b}$, $\frac{\partial L}{\partial b} = -\hat{y} = -(y-\hat{y}) \to b \to b + \alpha(y-\hat{y})$



for each training examples $x^{(i)}$, the prediction $\hat{y}^{(i)}$ can be calculated as:

$z^{(i)} = wx^{(i)} + b$, $\hat{y}^{(i)} = z^{(i)}$ where $i = 1, 2, \ldots, m$

We can organize all training examples as a vector X of size $(1 \times m)$ and perform scalar multiplication of $X(1 \times m)$ by a scalar w, adding b, which will be broadcasted to a vector of size $(1 \times m)$:

(for forward propagation)

$Z = wX + b$, $Y = Z$ (forward propagation)

for each training examples, we can measure the difference between the original values $y^{(i)}$ and predicted values $\hat{y}^{(i)}$ with the loss function.

$L(w,b) = \frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2$, we can take average of the loss function values for each of the training examples:

$$L(w,b) = \frac{1}{2m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})^2 \longrightarrow \text{cost function}$$

So the aim is to minimize the cost function during training which will minimize the difference between original values $y^{(i)}$ and predicted values $\hat{y}^{(i)}$,

When the weights were just initialized with some random values and no training was done yet, we can't expect good results. We need to calculate the adjustments for the weight and bias, minimizing the cost function. This process is called backward propagation.

We use gradient descent:

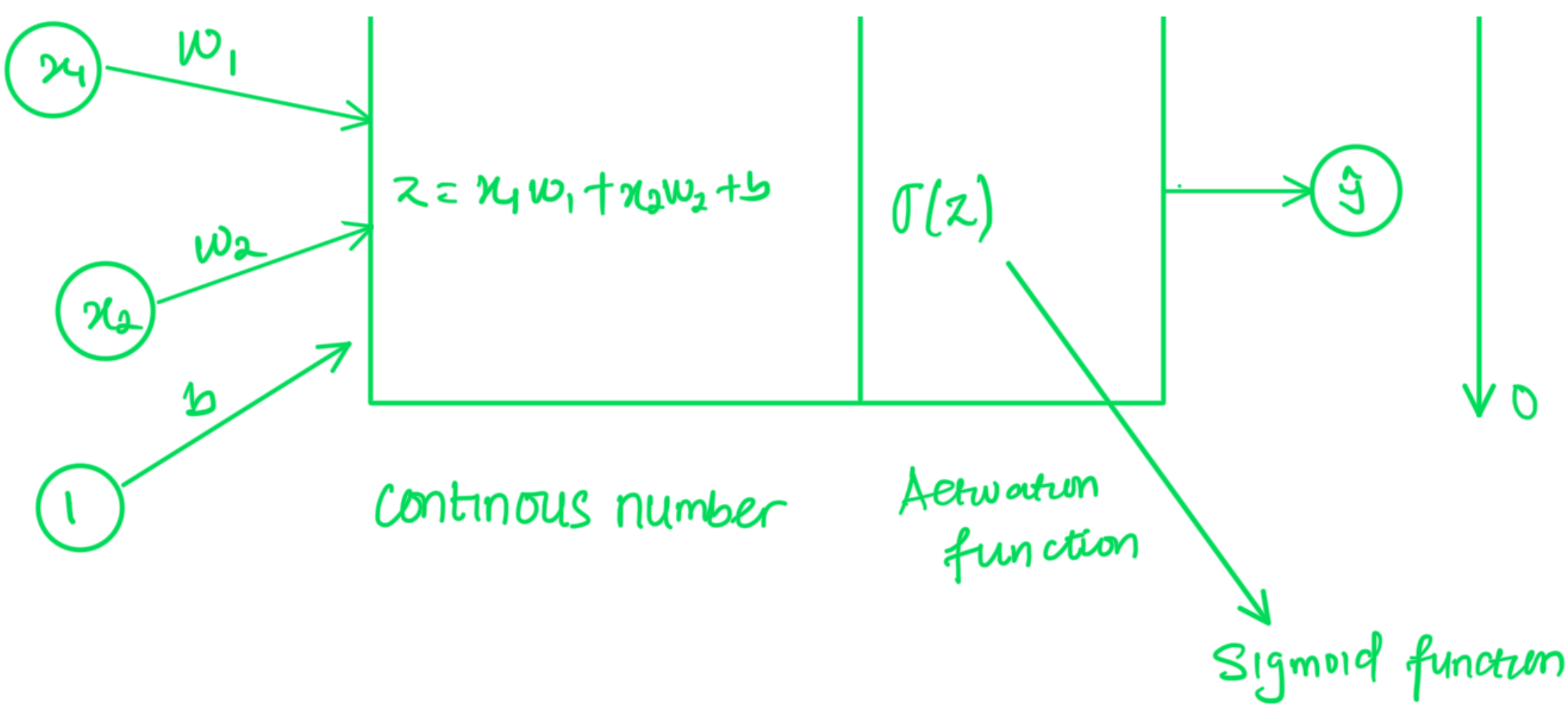
$$\frac{\partial L}{\partial w} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})\,x^{(i)}$$

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

$$w = w - \alpha\frac{\partial L}{\partial w}$$

$$b = b - \alpha\frac{\partial L}{\partial b}$$

⟶ Classification with perception.

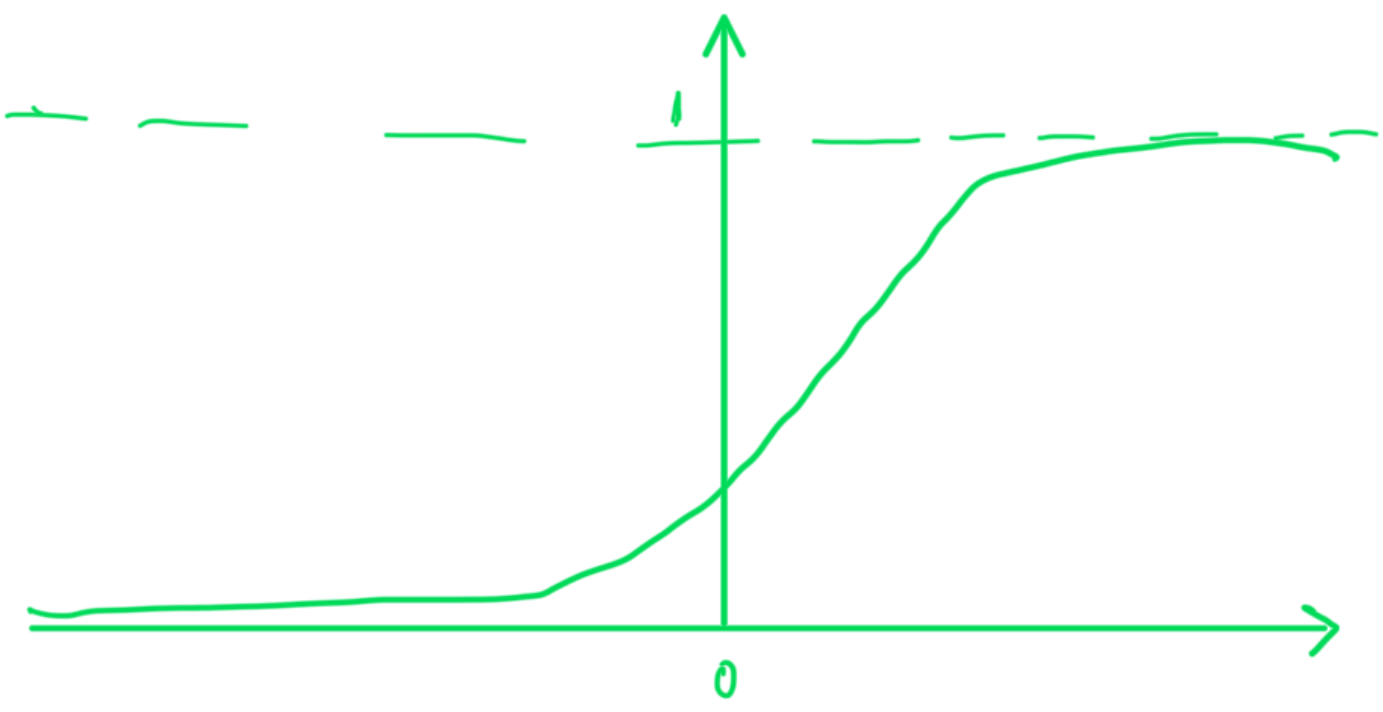


We need the activation function

to convert the continous number

into a discrete value like yes/no

$$\sigma(z) = \frac{1}{1+e^{-z}}$$



$$\sigma'(z) = \frac{e^{-z}}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}} - \frac{1}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right)$$

$$\sigma'(z) = \sigma(z)(1-\sigma(z))$$

the error in this case $L(y, \hat{y})$

Our prediction function here is $\hat{y} = \sigma(w_1 x_1 + w_2 x_2 + b)$

Loss function here will be $L(y, \hat{y}) = -y \log(\hat{y}) - (1-y)\log(1-\hat{y})$

Main goal: find w_1, w_2, b that gives $\hat{y}$ with the least error

We need to use gradient descent:

$\sigma'(z) = \sigma(z)(1-\sigma(z))$

$W \to W - \alpha \frac{\partial}{\partial w}$ [illegible] $\frac{\partial}{\partial b}$ → [illegible]

$\frac{\partial L}{\partial \hat{y}} = \frac{-(y-\hat{y})}{\hat{y}(1-\hat{y})}$, $\frac{\partial \hat{y}}{\partial w_1} = \hat{y}(1-\hat{y}) \cdot x_1$, $\frac{\partial \hat{y}}{\partial w_2} = \hat{y}(1-\hat{y})x_2$, $\frac{\partial \hat{y}}{\partial b} = \hat{y}(1-\hat{y})$

$\frac{\partial L}{\partial w_1} = -(y-\hat{y})x_1$, $\frac{\partial L}{\partial w_2} = -(y-\hat{y})x_2$, $\frac{\partial L}{\partial b} = -(y-\hat{y})$

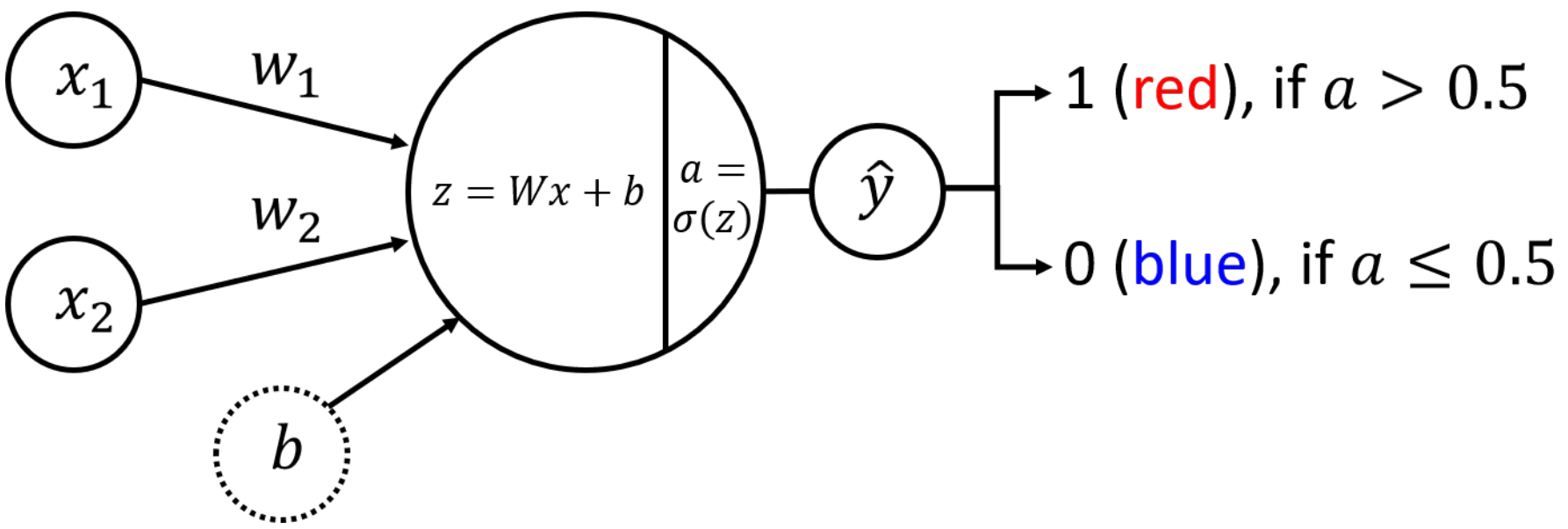
$w_1 \to w_1 - \alpha(-x_1(y-\hat{y}))$

$w_2 \to w_2 - \alpha(-x_2(y-\hat{y}))$

$b \to b - \alpha(-(y-\hat{y}))$

Classification is the problem of identifying which of a set of categories an observation belongs to.

Always turn on light mode to see image.



$$z^{(i)} = w_1 x_1^{(i)} + w_2 x_2^{(i)} + b = W x^{(i)} + b$$

But we cannot take a real number $z^{(i)}$ into the output as we need to perform classification, we need an extra step in forward propagation which is application of an activation function.

We use sigmoid:

$$a = \sigma(z) = \frac{1}{1+e^{-z}}$$

Then a threshold value of 0.3 can be used for predictions.

$$z^{(i)} = W x^{(i)} + b$$

$$a^{(i)} = \sigma(z^{(i)})$$

Model can be written as:

$$Z = WX + b$$

$$A = \sigma(Z)$$

$$J(W,b) = \frac{1}{m} \sum_{i=1}^{m} L(W,b)$$

$$= \frac{1}{m} \sum_{i=1}^{m} \left(-y^{(i)} \log(a^{(i)}) - (1-y^{(i)}) \log(1-a^{(i)}) \right)$$

Where $y^{(i)} \in \{0,1\}$

We will want to minimize the cost function during training.

$$\frac{\partial J}{\partial w_1} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial L}{\partial a^{(i)}} \frac{\partial a^{(i)}}{\partial z^{(i)}} \frac{\partial z^{(i)}}{\partial w_1}$$

$$= \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)}) x_1^{(i)}$$

$$\frac{\partial L}{\partial w_2} = \frac{1}{m}\sum_{i=1}^{m}\left(a^{(i)} - y^{(i)}\right)x_2^{(i)}$$

$$\frac{\partial L}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}\left(a^{(i)} - y^{(i)}\right)$$

Writing in matrix form:

$$\frac{\partial L}{\partial w} = \left[\frac{\partial L}{\partial w_1} \quad \frac{\partial L}{\partial w_2}\right] = \frac{1}{m}\left(A - Y\right)X^T$$

$$\frac{\partial L}{\partial b} = \frac{1}{m}\left(A - Y\right)1 \rightarrow \text{(m x 1) vector of ones.}$$

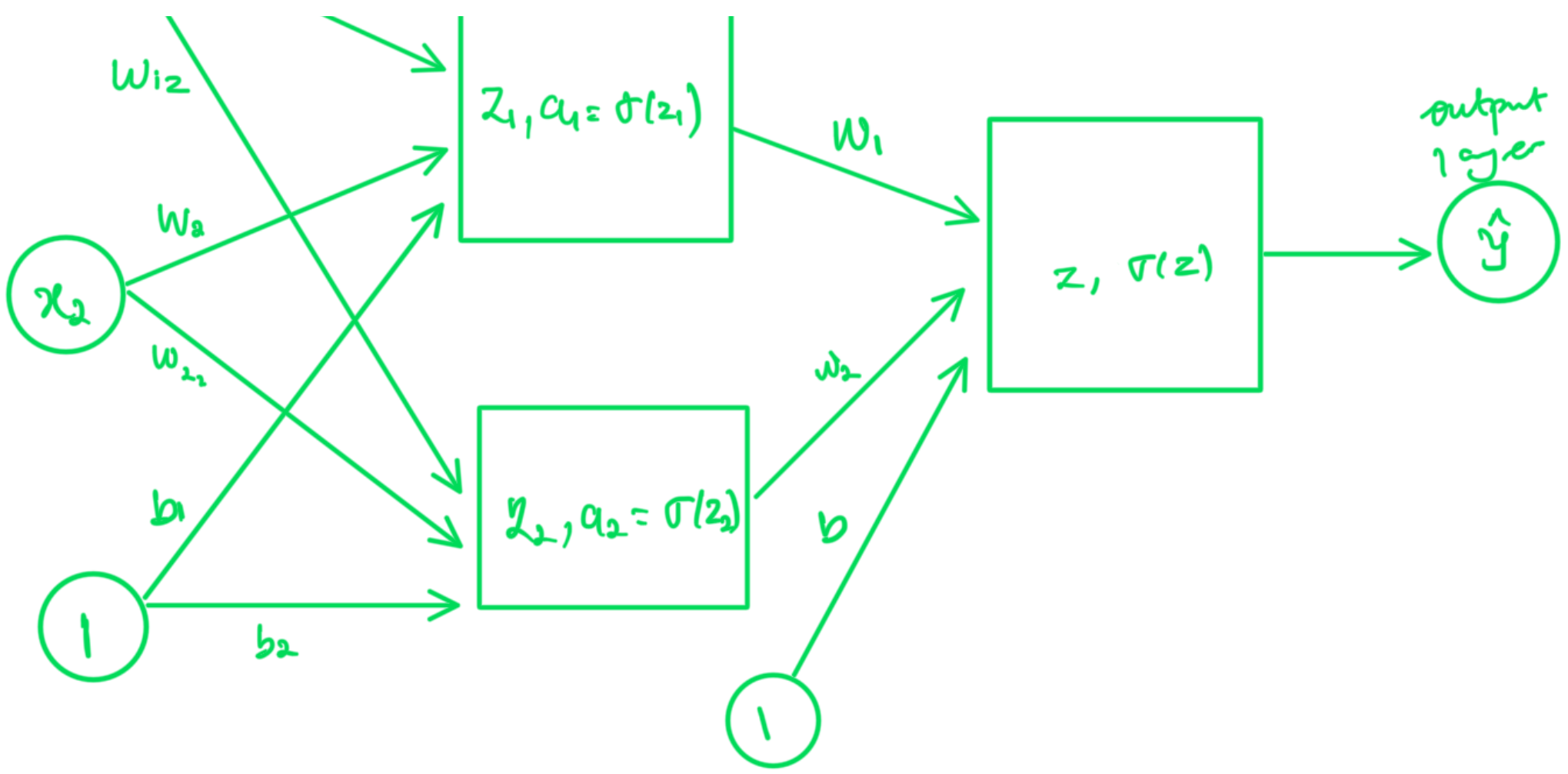
The gradient descent:

$$W = W - \alpha\frac{\partial L}{\partial W}, \quad b = b - \alpha\frac{\partial L}{\partial b}$$

$\alpha \rightarrow$ learning rate.

finally we get, $\hat{y} = \begin{cases} 1 & \text{if } a > 0.5 \\ 0 & \text{otherwise} \end{cases}$

Neural network is a bunch of perceptions organized in layers where the intermation of current perception is passed to the next laye and we train neural networks using gradients.



$a_1 = \sigma(z_1)$, $z_1 = x_1 w_{11} + x_2 w_{21} + b_1$

$a_2 = \sigma(z_2)$, $z_2 = x_1 w_{12} + x_2 w_{22} + b_2$

$\hat{y} = \sigma(z)$, $z = a_1 w_1 + a_2 w_2 + b$

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

$$\frac{\partial L}{\partial w_{11}} = \frac{\partial z_1}{\partial w_{11}} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z}{\partial a_1} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial L}{\partial \hat{y}}$$

$$= [x_1] \cdot [a_1(1-a_1)] [w_1] \cdot [\hat{y}(1-\hat{y})] \left[\frac{-(y-\hat{y})}{\hat{y}(1-\hat{y})}\right]$$

$$= -x_1 w_1 a_1 (1-a_1)(y-\hat{y})$$

$$w_{11} \rightarrow w_{11} + \alpha x_1 w_1 a_1 (1-a_1)(y-\hat{y})$$

$$w_{12} \rightarrow w_{12} + \alpha x_2 w_1 a_1 (1-a_1)(y-\hat{y})$$

$$b_1 \rightarrow b_1 + \alpha w_1 a_1 (1-a_1)(y-\hat{y})$$

$$w_{21} \rightarrow w_{21} + \alpha x_1 w_2 a_2 (1-a_2)(y-\hat{y})$$

$$w_{22} \rightarrow w_{22} + \alpha x_2 w_2 a_2 (1-a_2)(y-\hat{y})$$

$$b_2 \rightarrow b_2 + \alpha w_2 a_2 (1-a_2)(y-\hat{y})$$

$w_1 \rightarrow w_1 + \alpha a_1 (y - \hat{y})$

$w_2 \rightarrow w_2 + \alpha a_2 (y - \hat{y})$

$b \rightarrow b + \alpha (y - \hat{y})$

Newton's method

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

do iteratively to solve for $f(x) \approx 0$

So how does this help us minimize a function $g(x)$?

let $f(x) = g'(x)$, then solve for $f(x) = 0$, since we can get the minimum of $g(x)$ at $g'(x) = f(x) = 0$ $\Rightarrow$ there is your answer ski

Hence for $g(x)$, we have:

$$x_{n+1} = x_n - \frac{g'(x_n)}{g''(x_n)} \Rightarrow \text{Really really fast man!}$$

$\left.\frac{d^2 f}{dx^2}\right|_{x=a} > 0 \rightarrow$ minimum value at $x = a$

$\left.\frac{d^2 f}{dx^2}\right|_{x=a} < 0 \rightarrow$ maximum value at $x = a$

Hessian

$f(x,y) = 2x^2 + 3y^2 - xy$ $\rightarrow$ $\begin{bmatrix} 4 & -1 \\ -1 & 6 \end{bmatrix} \Leftrightarrow \begin{bmatrix} f_{xx} & f_{xy} \\ f_{yx} & f_{yy} \end{bmatrix}$ ↑ Hessian matrix

$f_{xx} = 4, f_{xy} = f_{yx} = -1, f_{yy} = 6$

$\begin{bmatrix} f_{xx} & f_{xy} \end{bmatrix}$

$H(x,y) = \begin{bmatrix} f_{xx} & f_{xy} \\ f_{yx} & f_{yy} \end{bmatrix}$

⟶ Hessian and Concavity

How do we determine if we have a minimum or maximum value from the Hessian?

We can look at the eigen values

$f = 2x^2 + 3y^2 - 2xy$, $f_x = 4x - 2y$ and $f_y = 6y - 2x$

$4x - 2y = 0$ and $6y - 2x = 0 \Rightarrow x = 3y$ and $y = 0$ and so $x = 0$.

$H(0,0) = \begin{bmatrix} 4 & -1 \\ -1 & 6 \end{bmatrix}$, we take the eigen values.

$|H(0,0) - \lambda I| = \begin{vmatrix} 4-\lambda & -1 \\ -1 & 6-\lambda \end{vmatrix} = 0 \Rightarrow \lambda_1 = 6.41$ and $\lambda_2 = 3.59$.

Since $\lambda_1 > 0$ and $\lambda_2 > 0$, $(0,0)$ is a minimum

When all eigenvalues are negative, then we have a maximum. for a function with n variables minimum if

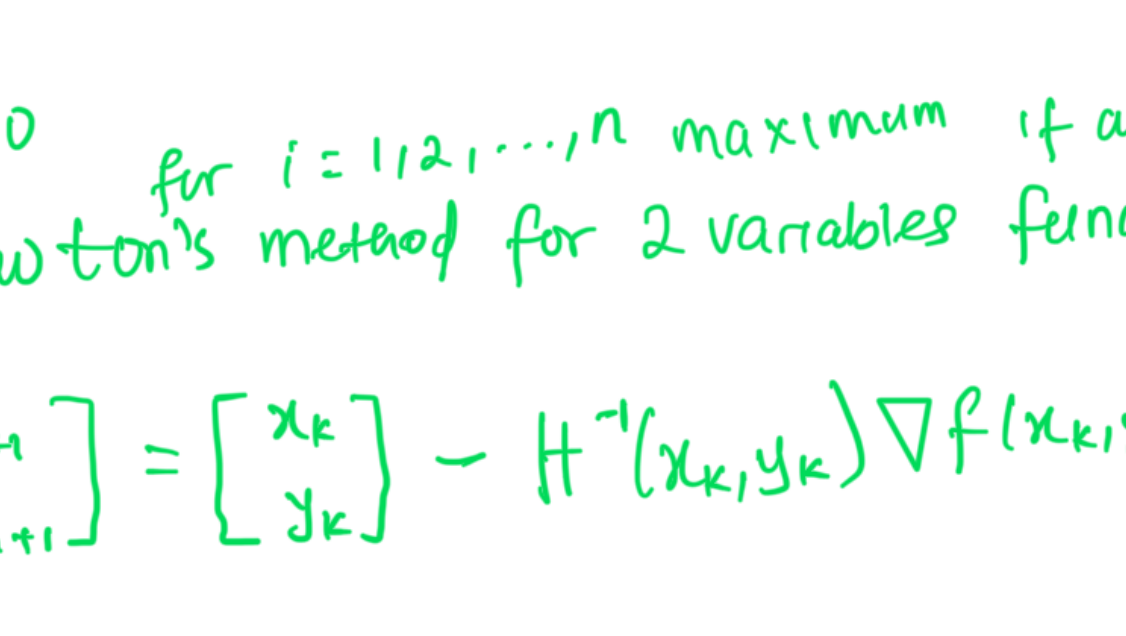
all $\lambda_i > 0$ for $i = 1, 2, \ldots, n$ maximum if all $\lambda_i < 0$

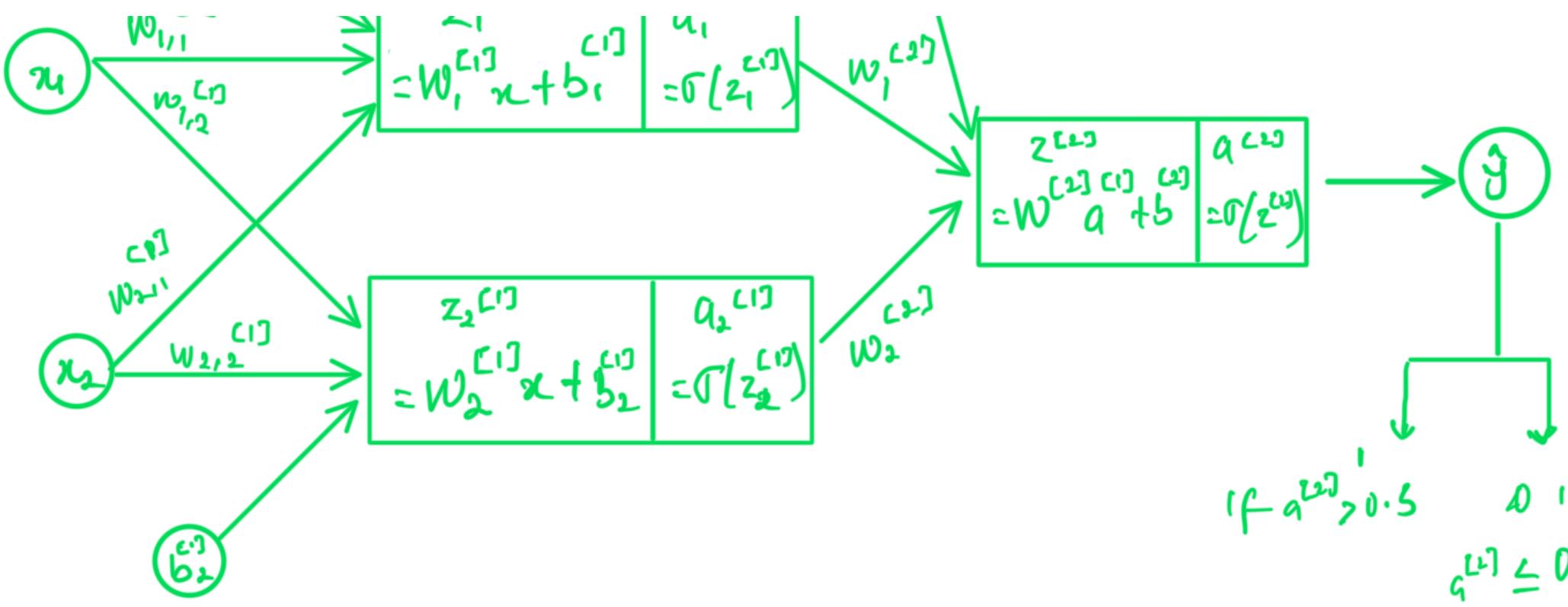
⟶ Newton's method for 2 variables function.

$$\begin{bmatrix} x_{k+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} x_k \\ y_k \end{bmatrix} - H^{-1}(x_k, y_k) \nabla f(x_k, y_k) \quad \left[\text{not } \nabla f(x_k, y_k) H^{-1}(x_k, y_k) \right]$$

(2,1) (2,2)

Neural network model with two layers





$n_x = 2$
Input layer

$n_h = 2$
Hidden layer

$n_y = 1$
output layer ← Layer size

Training examples $x^{(i)} = \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \end{bmatrix}$ from the input layer of size $n_x = 2$ are first fed into the hidden layer of size $n_h = 2$. They are simultaneously fed into the perceptron with weights $W_1^{[1]} = [w_{1,1}^{[1]} \; w_{2,1}^{[1]}]$, bias $b_1^{[1]}$ and into the second perceptron with weights $W_2^{[1]} = [w_{1,2}^{[1]} \; w_{2,2}^{[1]}]$, bias $b_2^{[1]}$. The integer in square brackets denotes the layer number, because there are two layers with their own parameters and outputs which need to be distinguished.

$$z_1^{[1](i)} = w_{1,1}^{[1]} x_1^{(i)} + w_{2,1}^{[1]} x_2^{(i)} + b_1^{[1]} = W_1^{[1]} x^{(i)} + b_1^{[1]}$$

$$z_2^{[1](i)} = w_{1,2}^{[1]} x_1^{(i)} + w_{2,2}^{[1]} x_2^{(i)} + b_2^{[1]} = W_1^{[1]} x^{(i)} + b_2^{[1]}$$

In matrix form:

$$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]}$$

where $z^{[1](i)} = \begin{bmatrix} z_1^{[1](i)} \\ z_2^{[1](i)} \end{bmatrix}$ is a vector of size $(n_h \times 1) = (2 \times 1)$

$$W^{[1]} = \begin{bmatrix} W_1^{[1]} \\ W_2^{[1]} \end{bmatrix} = \begin{bmatrix} w_{1,1}^{[1]} & w_{2,1}^{[1]} \\ w_{1,2}^{[1]} & w_{2,2}^{[1]} \end{bmatrix}$$ is a matrix of size $(n_h \times n_x) = (2 \times 2)$

$$b^{[1]} = \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \end{bmatrix}$$ is a vector of size $(n_h \times 1) = (2 \times 1)$

Next, the hidden layer activation function needs to be applied for each of the elements of the vectors $z^{[1](i)}$, $\sigma(z) = \frac{1}{1+e^{-z}}$, $\sigma'(z) = \sigma(z)(1-\sigma(z))$

The output of the hidden layer is a vector of size $(n_h \times 1) = (2 \times 1)$

$$a^{[1](i)} = \sigma(z^{[1](i)}) = \begin{bmatrix} \sigma(z_1^{[1](i)}) \\ \sigma(z_2^{[1](i)}) \end{bmatrix}$$

Then the hidden layer output gets fed into the output layer of size $n_y = 1$.

$$z^{[2](i)} = w_1^{[2]} a_1^{[1](i)} + w_2^{[2]} a_2^{[1](i)} + b^{[2]} = W^{[2]} a^{[1](i)} + b^{[2]}$$

$z^{[2](i)}$ and $b^{[2]}$ are scalars for this model, as $(n_y \times 1) = (1 \times 1)$

$W^{[2]} = [w_1^{[2]} \quad w_2^{[2]}]$ is a vector of size $(n_y \times n_h) = (1 \times 2)$

Finally, the same sigmoid function is used as the out put layer activation function:

$$a^{[2](i)} = \sigma(z^{[2](i)})$$

So, we have:

$$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]}$$

$$a^{[1](i)} = \sigma(z^{[1](i)})$$

$$z^{[2](i)} = W^{[2]} x^{(i)} + b^{[2]}$$

$$a^{[2](i)} = \sigma(z^{[2](i)})$$

Note that all of the parameters to be trained in the model are without the $^{(i)}$ index, they are independent of the input data.

Finally, the prediction for some example $x^{(i)}$ can be made taking the $\hat{y}^{(i)} = \begin{cases} 1 & \text{if } a^{[2](i)} > 0.5, \\ \end{cases}$

output $a^{[2]}$ and calculating y as: $y = 1$ if $a^{[2]} > 0.5$ otherwise 0

Let's write in matrix form:

$$Z^{[1]} = W^{[1]} X + b^{[1]}$$

$$A^{[1]} = \sigma(Z^{[1]})$$

$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$

$$A^{[2]} = \sigma(Z^{[2]})$$

m training examples can be organized in a matrix X of shape $(2 \times m)$, putting $x^{(i)}$ into columns, $b^{[1]}$ is the broadcasted matrix of size $(2 \times m)$ and $b^{[2]}$ a vector of size $(n_y \times m) = (1 \times m)$

→ Cost function and training

We use log loss function

$$L(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^{m} \left(-y^{(i)} \log(a^{[2](i)}) - (1-y) \log(1 - a^{[2](i)}) \right)$$

Where $y^{(i)} \in \{0, 1\}$ are the original labels and $a^{[2](i)}$ are the continuous outputs of the forward propagation step (elements of array $A^{[2]}$).

$$\frac{\partial L}{\partial W} = \frac{1}{m}(A - Y) X^T \quad , \quad \frac{\partial L}{\partial b} = \frac{1}{m}(A - Y) 1$$

$$\frac{\partial L}{\partial W^{[2]}} = \frac{1}{m}(A^{[2]} - Y)(A^{[1]})^T$$

$$\frac{\partial L}{\partial b^{[2]}} = \frac{1}{m}(A^{[2]} - Y) 1$$

$$\frac{\partial L}{\partial W^{[1]}} = \frac{1}{m}\left((W^{[2]})^T (A^{[2]} - Y) \cdot (A^{[1]} \cdot (1 - A^{[1]})) \right) X^T$$

$$\frac{\partial L}{\partial b^{[1]}} = \frac{1}{m}\left((W^{[2]})^T (A^{[2]} - Y) \cdot (A^{[1]} \cdot (1 - A^{[1]})) \right) 1$$

$\frac{\partial \mathcal{L}}{\partial b^{[1]}}$ m (\

$\underline{1}$ is an $(m \times 1)$ vector of ones.

Basically $\frac{\partial \mathcal{L}}{\partial W^{[1]}} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial w_{11}^{[1]}} & \frac{\partial \mathcal{L}}{\partial w_{21}^{[1]}} \\ \frac{\partial \mathcal{L}}{\partial w_{12}^{[1]}} & \frac{\partial \mathcal{L}}{\partial w_{22}^{[1]}} \end{bmatrix}$

But we know that

$$\frac{\partial \mathcal{L}}{\partial w_{11}^{[1]}} = \frac{1}{m} \sum_{i=1}^{m} \left(\left(a^{[2](i)} - y^{(i)} \right) w_1^{[2]} \left(a_1^{[1](i)} \left(1 - a^{[1](i)} \right) \right) x^{(i)} \right)$$

. . . . . and we arrive at the formula