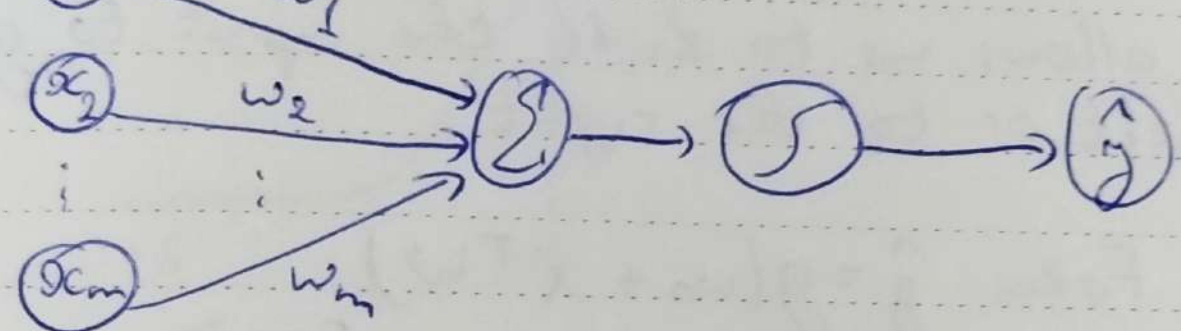
The Perceptron

The structural building block of deep learning



Inputs Weights Sum Non-linearity Output

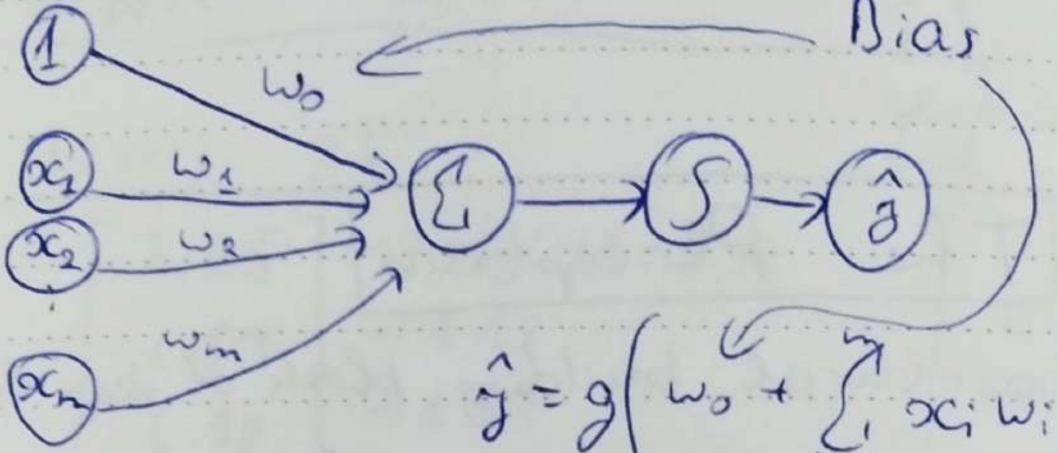
$$\hat{y} = g\left(\sum_{i=1}^{m} x_i w_i\right)$$

output

non-linear activation function

linear combination of inputs

The process of moving from left to right is called "Forward Propagation".



$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i w_i\right)$$

The bias allows us to shift the input to g to the left or to the right.

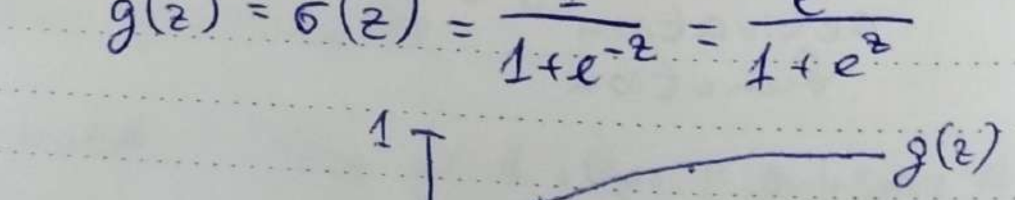
Matrix Form: $\hat{y} = g(w_0 + X^T W)$

where: $X = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ and $W = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$

Activation Functions

- Example: Sigmoid function

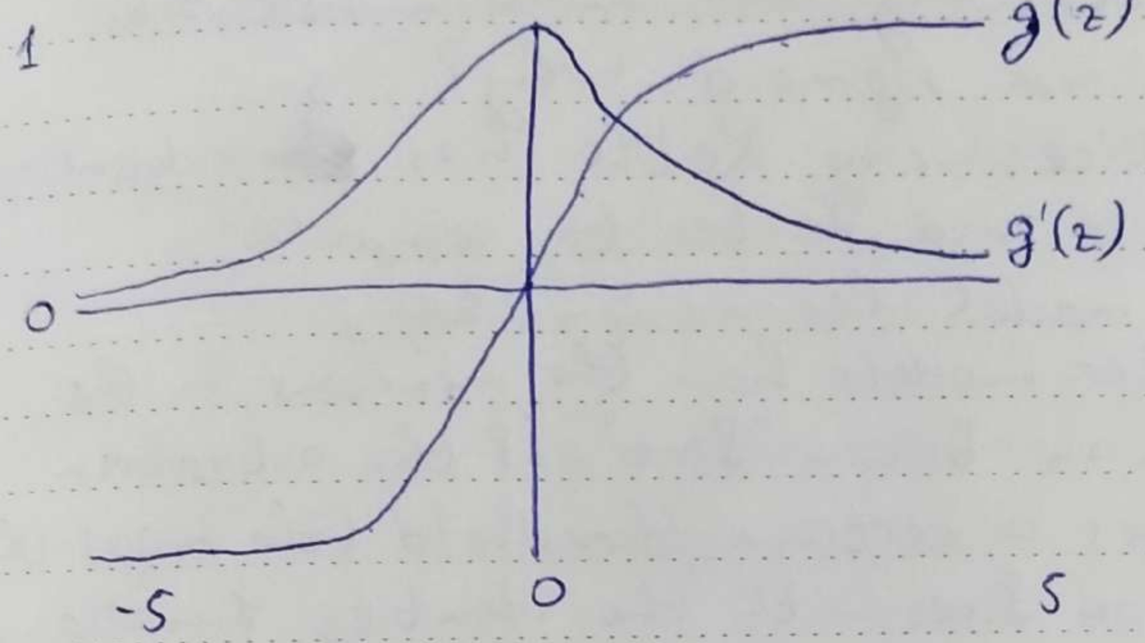
$$g(z) = \sigma(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{1+e^z}$$



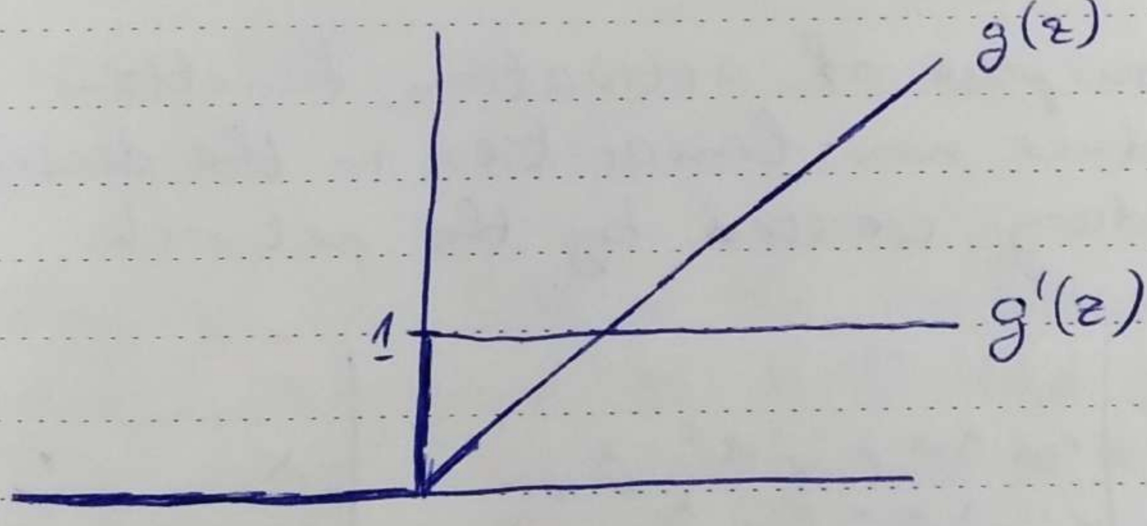
$$g'(z) = g(z)(1 - g(z))$$

- Hyperbolic Tangent (used in RNNs)

$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \quad \Bigg| \quad g'(z) = 1 - g(z)^2$$



- Rectified Linear Unit (ReLU)



$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1 & , z > 0 \\ 0 & , \text{otherwise} \end{cases}$$

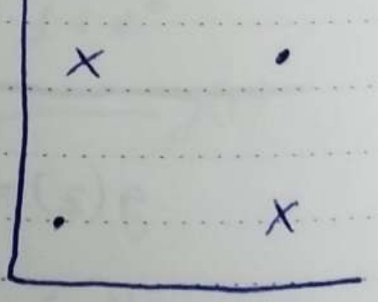
Note: All activation functions are non-linear.

Sigmoid vs. ReLU

The sigmoid is popular largely because it directly outputs values between 0 and 1. This makes it suitable for modelling probabilities.
However, relatively few modern networks actually use sigmoid. Why?
In practice, using ReLU has 3 advantages:
- easier and faster to compute;
- the model learns quicker;
- better models how the neurons in the human brain "fire". If the activation passes a certain threshold (the bias) the neuron fires with the identity function. Otherwise, it stays inactive.

Why use Activation Functions?

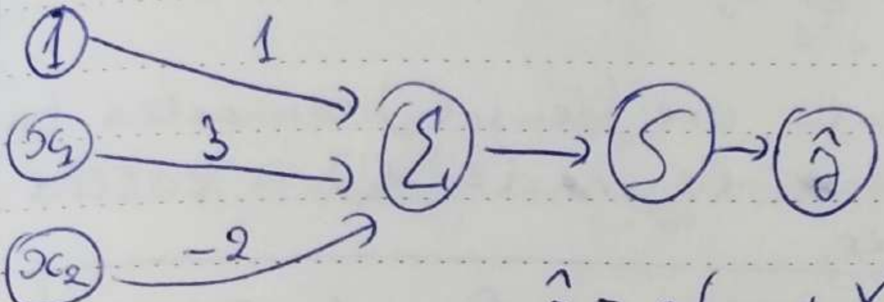
The purpose of activation functions is to introduce non-linearities in the decision boundary, created by the network.



If only linear activation f-ns are used, no matter the number of neurons or the number of layers, only one line will be produced.
line + line = line

Non-linearities allow the approximation of arbitrarily complex functions.

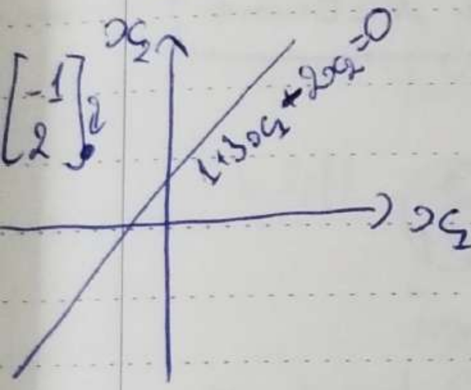
Perceptron: Example



Let $w_0 = 1$,

$W = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

$$\hat{y} = g(w_0 + X^T W)$$
$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right)$$
$$= g(\underbrace{1 + 3x_1 - 2x_2}_{\text{This is a line in 2D!}})$$

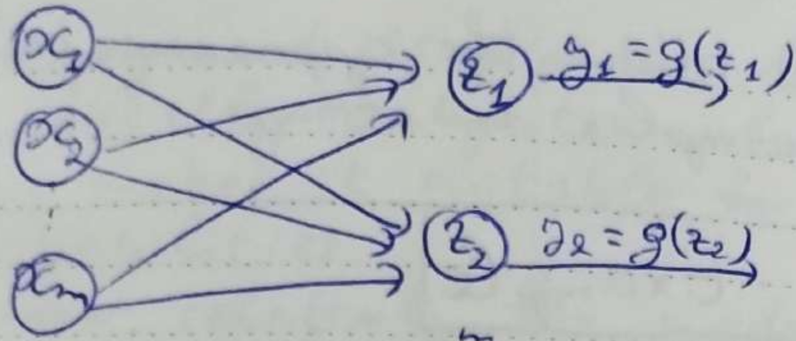


$$\hat{y} = g(1 + (3 \times (-1)) - 2 \times 2) =$$
$$= g(-6) \approx 0{,}002$$

- Anything to the left of the line (decision boundary) corresponds to $z < 0$ and $y < 0.5$.
- Anything on the line corresponds to $z = 0$, $y = 0.5$.
- Anything to the right of the line corresponds to $z > 0$, $y > 0.5$.

Notice how here there're only 2 weights. In practice, there're millions or billions of weights.

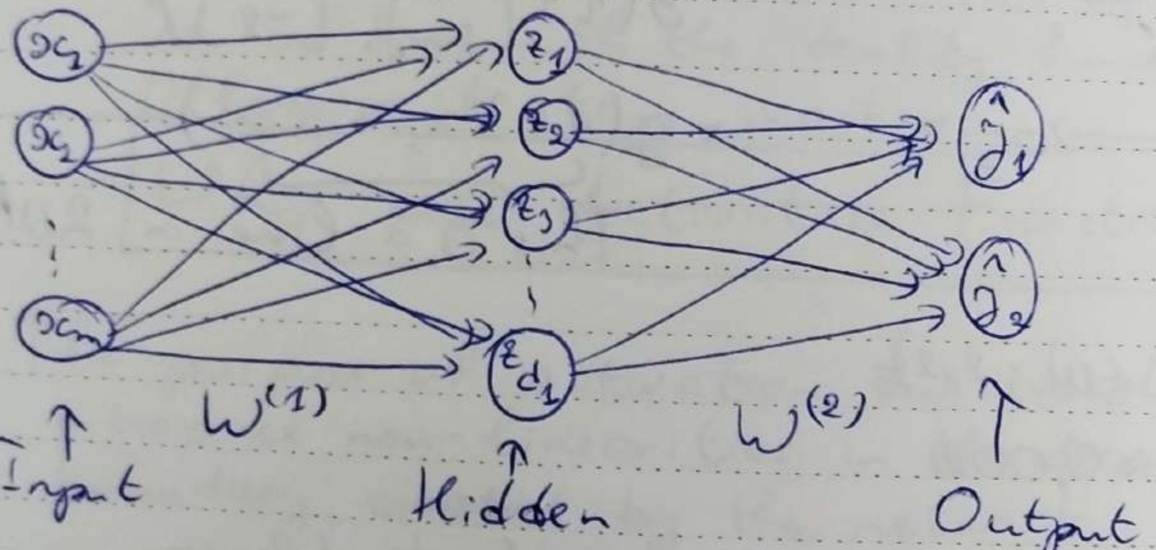
Multi Output Perceptron



$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j \, w_{j,i} \qquad , i = \{1, 2\}$$

Because all inputs are (densely) connected to all outputs, the layer (z_1, z_2) is called a Dense layer.

Single Hidden Layer Neural Network

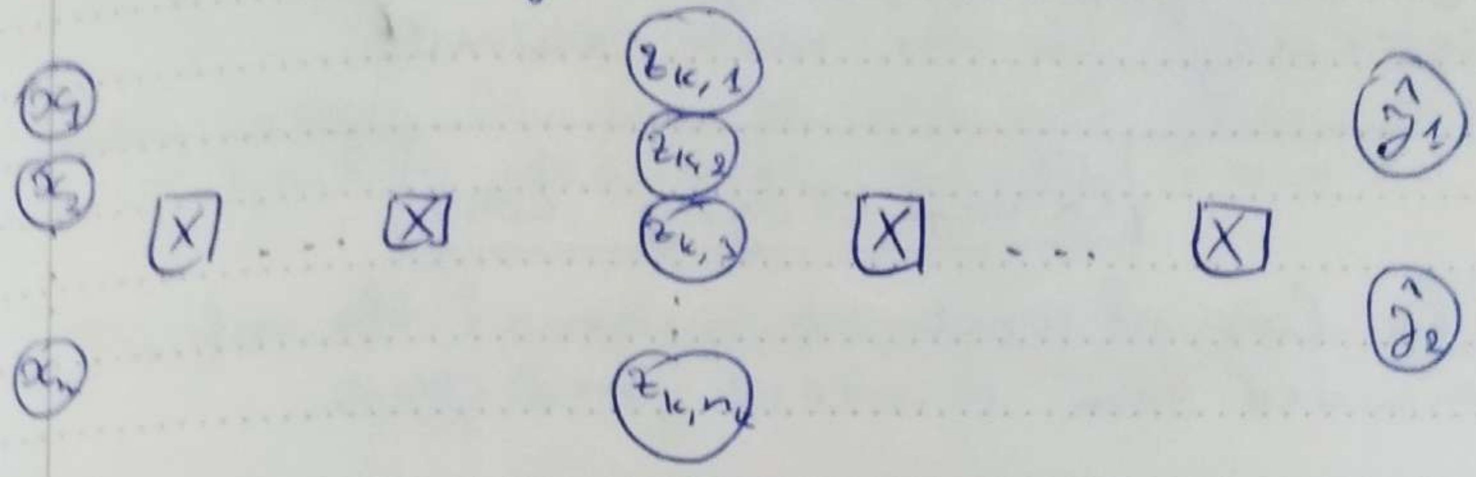


$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)}$$

$$\hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} g(z_j) \, w_{j,i}^{(2)}\right)$$

Note: Weights and biases get initiallized with random value. In practice, these values are drawn from the normal distribution : $\mathcal{N}(0, \sigma^2)$.

Deep Neural Network



$$z_{ki} = w_{0,i}^{(k)} + \sum_{j=1}^{n_{k-1}} g(z_{k-1,j})\, w_{ji}^{(k)}$$

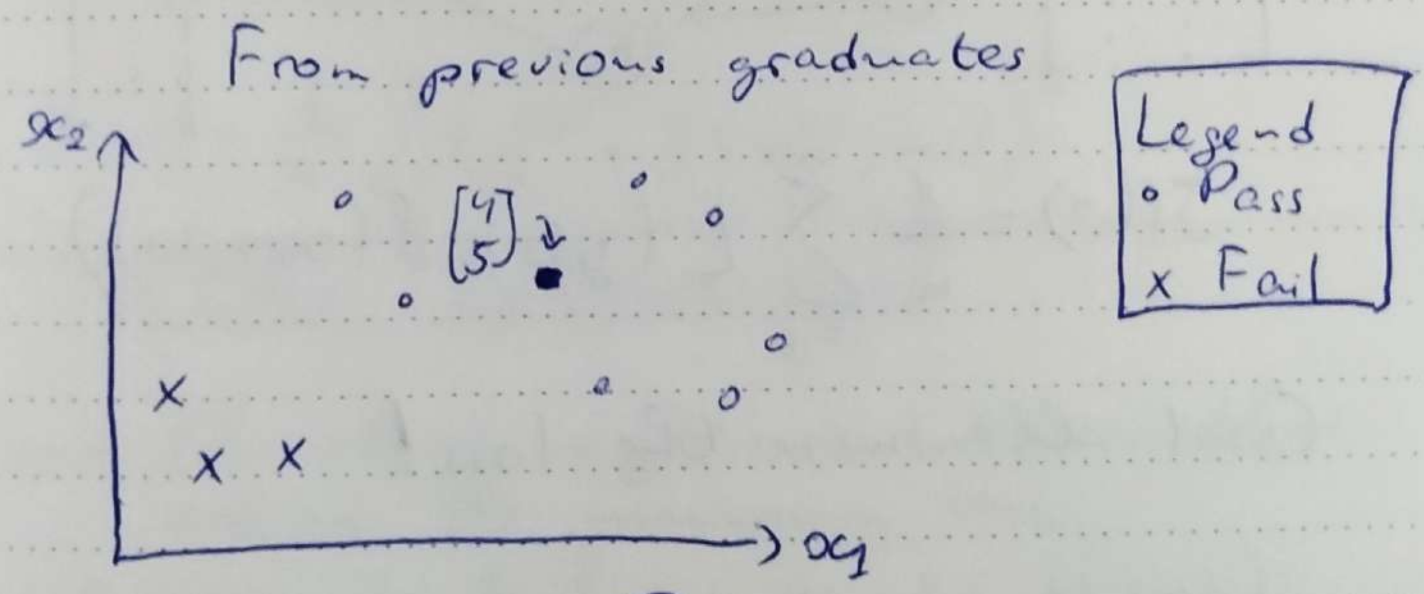
Example Problem

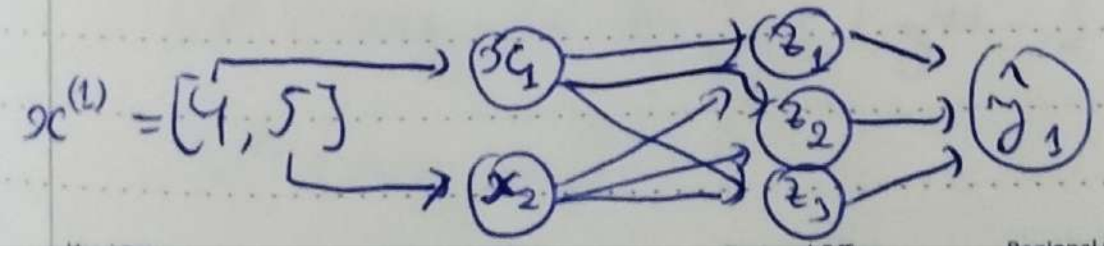
Will we get our Bachelor's in 4 years?

Input features:

x_1 = number of lectures attended
x_2 = hours spent studying

From previous graduates





Predicted : 0.1
Actual : 1

Why did the network get this answer incorrectly? It was never trained!

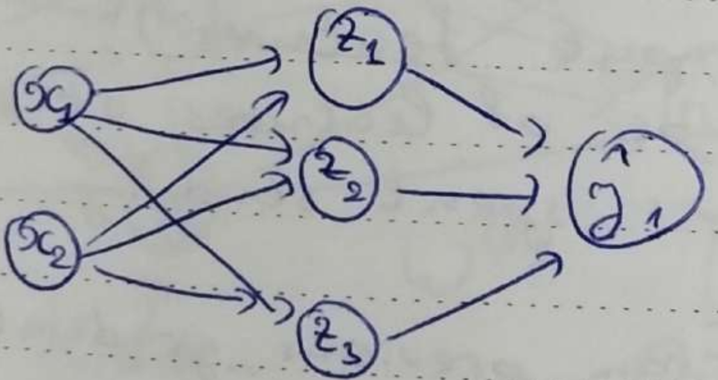
Quantifying Loss

The loss of a network measures the cost incurred from incorrect predictions.

$$L\Big(\underset{\text{actual}}{\underline{y(i)}}\ ,\ \underset{\text{predicted}}{\underline{f(x(i); W)}}\Big)$$

Empirical Loss
(AKA Objective f-n, Cost f-n, Error f-n)

$$X = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



$$\begin{array}{ccc} f(x) & & y \\ \begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} & \begin{matrix} \times \\ \times \\ \checkmark \\ \end{matrix} & \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix} \end{array}$$

$$J(W) = \frac{1}{n} \sum_{i=1}^{n} L\big(y(i), f(x(i); W)\big)$$

Goal: Minimize the loss!

Binary Cross Entropy Loss

Cross entropy loss can be used with models that output a probability between 0 and 1

$$J(W) = -\frac{1}{n}\sum_{i=1}^{n} y^{(i)} \log(f(x^{(i)};W)) + (1-y^{(i)}) \log(1-f(x^{(i)};W))$$

Recap. For multiclass:

$$J(W) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{k} y_{ij} \log(f(x^{(i)};W))$$

n - samples
k - classes
y_{ij} - 1 if $x^{(i)}$ has label j

Mean Squared Error Loss

MSE is used in regression models that output continuous real numbers

$$J(W) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - f(x^{(i)};W)\right)^2$$

Loss Optimization

Find the set of weights and biases that achieve the minimum loss.

$$W^* = \underset{W}{\operatorname{argmin}} \frac{1}{n}\sum_{i=1}^{n} L(y^{(i)}, f(x^{(i)};W))$$

$$W^* = \underset{W}{\operatorname{argmin}}\, J(W)$$

Gradient Descent

1. Randomly pick an initial (w_0, w_1).
2. Compute gradient : $\frac{\partial J(W)}{\partial W}$
3. Take a step in opposite direction of gradient.
4. Repeat until convergence (change of loss $< \varepsilon$)
5. Return W (weights + biases).

Backpropagation

The algorithm for determining by how much a single training example would like to nudge the weights and biases.

$$x \xrightarrow{w_1} z_1 \xrightarrow{w_2} \hat{y} \longrightarrow J(W)$$

How does a small change in w_2 affect $J(W)$?

$$\frac{\partial J(W)}{\partial w_2} = \frac{\partial J(W)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_2}$$

For w_1:

$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_1} \cdot \frac{\partial z_1}{\partial w_1}$$

Weight update: $W \leftarrow W - \eta \cdot \frac{\partial J(W)}{\partial W}$

This is the "learning rate" parameter.
It determines how big of a step we make.

How to set the learning rate?

Idea 1: Try lots of different values and see what works "just right".

Idea 2: Design an adaptive learning rate that "adapts" to the landscape.

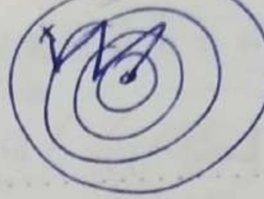
Gradient Descent Algorithms

(also called "optimizers")

- SGD
- Adam
- Adadelta
- Adagrad
- RMSProp

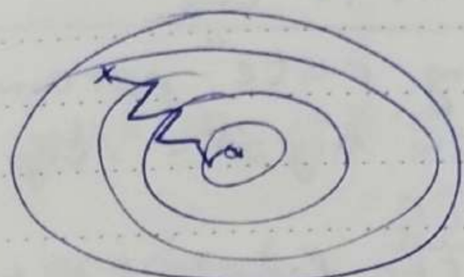
Batch gradient descent vs.
Mini-batch gradient descent vs.
Stochastic gradient descent

SGD | 1. Init randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. pick a single data point i
4. compute gradient

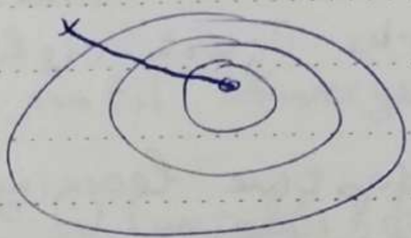


Mini-batch (usually 32 - 256)

Allows for:
- parallel computation
- speed increase



Batch

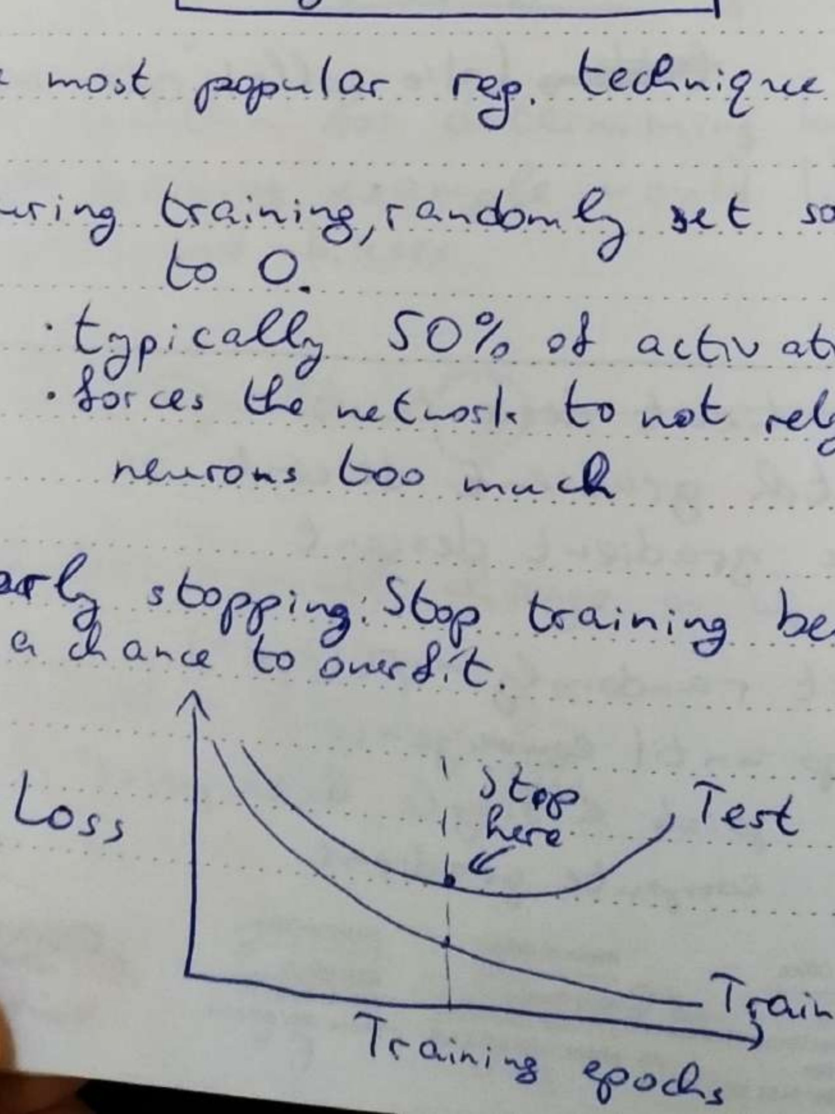


Regularization

1. The most popular reg. technique is Dropout.

   During training, randomly set some activations to 0.
   - typically 50% of activations in a layer
   - forces the network to not rely on those neurons too much

2. Early stopping: Stop training before we have a chance to overfit.



Remember

The Perceptron	NN	Training
· Building blocks	· Stacking Perceptrons	· Adaptive learning
· Nonlinear activation functions	· Optimization through backpropagation	· Batching
		· Regularization