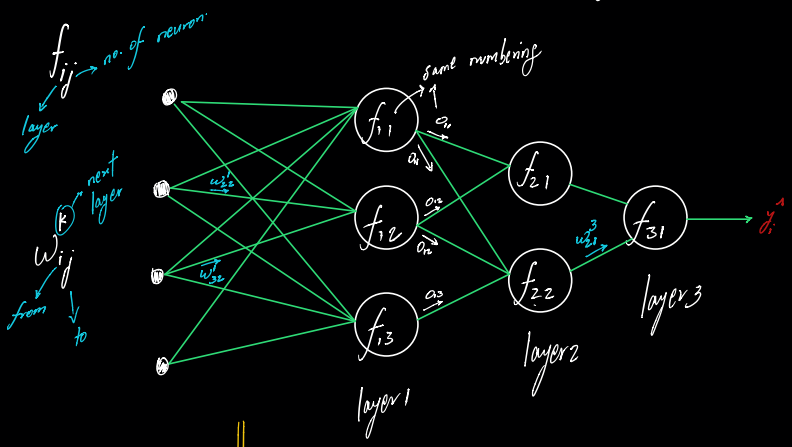
# Fully connected NN

$$D = \{ x_i , y_i \} \; ; \quad x_i \in \mathbb{R}^4$$
$$y_i \in \mathbb{R}$$




→ The math becomes easy once the notation is clear.

$W^1$

$$\begin{bmatrix} w^1_{11} & w^1_{12} & w^1_{13} \\ w^1_{21} & w^1_{22} & w^1_{23} \\ w^1_{31} & w^1_{32} & w^1_{33} \\ w^1_{41} & w^1_{42} & w^1_{43} \end{bmatrix}_{4 \times 3}$$

4 → next layer.

3 → previous layer

#Quick Note:

Linear Regression:

$$\min_{w_i} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + \text{regularization}, \qquad \hat{y}_i = w^T x_i$$

# NN

$$D = \{ x_i, y_i \}_{i=1}^{n}$$

$$x_i \in \mathbb{R}^4$$

$$y_i \in \mathbb{R}$$

① ⇒ loss function:

$$L = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + \text{reg}$$

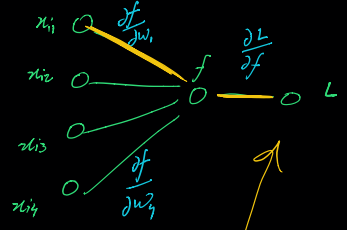
($L$: for all data)

② ⇒ optimization:

$$\min_{w_i} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + \text{regularization}, \qquad \hat{y}_i = w^T x_i$$

$$w^* = \arg\min_{w} \sum_{i=1}^{n} (y_i - f(w^T x_i))^2 + \text{regularization}$$

($w^*$: optimal)




$\nabla_w L$: gradient of $L$ w.r.t $w$

$$\nabla_w L = \left[ \frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \frac{\partial L}{\partial w_3}, \frac{\partial L}{\partial w_4} \right]^T$$

if we know

chain rule:

$$\frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial w_1} = \frac{\partial f(w^T x_i)}{\partial w_i} = x_i$$

follow the connecting path and keep differentiating until input is reached

this, we can then change w accordingly to minimize loss.

$$\frac{\partial \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + \text{reg}}{\partial f}$$

↑ ignore for simplicity.

$$= 2 \sum_{i=1}^{n} (y_i - \hat{y}_i)(-1) \frac{\partial \hat{y}_i}{\partial f}$$

$$= 2 \sum_{i=1}^{n} (y_i - f(w^T x_i)) \cdot (-1) \cdot \frac{\partial f}{\partial f}$$

$\frac{\partial f}{\partial f} \to 1$

$$\therefore \frac{\partial L_i}{\partial w_1} = -2 x_i (y_i - \hat{y}_i)$$

$$\frac{\partial L}{\partial w_1} = -2 \sum_{i=1}^{n} x_i (y_i - \hat{y}_i)$$

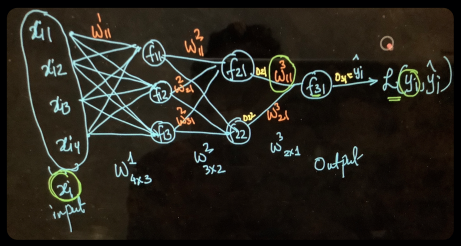
Similarly we can get stuff for $\frac{\partial L}{\partial w_2}, \dots$

$\therefore \nabla_w L \to$ #Gradient how much should w change to move the value of L

$$\therefore w_{new} = w_{old} - \eta [\nabla_w L]_{w_{old}}$$

# Training

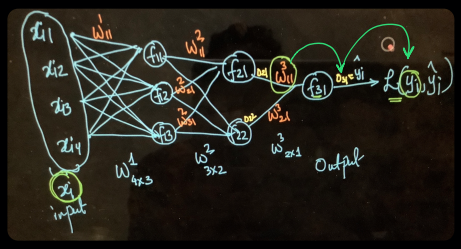
$$\text{regularizer} = \sum_{i,j,k} \left(w_{ij}^{k}\right)^2$$ ($L_2$ norm)




# Gradient Descent

① Initialize $w_{ij}^{k}$ [lots of techniques out there]

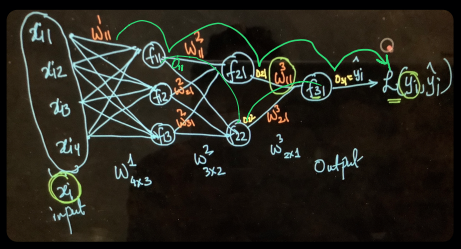
② $$w_{ij\,new}^{k} = \left(w_{ij}^{k}\right)_{old} - \eta \frac{\partial L}{\partial w_{ij}^{k}}$$




$$\frac{\partial L}{\partial w_{11}^{3}} = \frac{\partial L}{\partial o_{31}} \cdot \frac{\partial o_{31}}{\partial w_{11}^{3}}$$

Similarly

$$\frac{\partial L}{\partial w_{21}^{2}} = \frac{\partial L}{\partial o_{31}} \cdot \frac{\partial o_{31}}{\partial o_{21}} \cdot \frac{\partial o_{21}}{\partial w_{11}^{2}}$$




→ multiple paths. so adding them up.

$$\frac{\partial L}{\partial w^1_{11}} = \frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial w^1_{11}}$$

$$= \frac{\partial L}{\partial O_{31}} \cdot \left[ \frac{\partial O_{31}}{\partial O_{21}} \cdot \frac{\partial O_{21}}{\partial w^1_{11}} + \frac{\partial O_{31}}{\partial O_{22}} \cdot \frac{\partial O_{22}}{\partial w^1_{11}} \right]$$

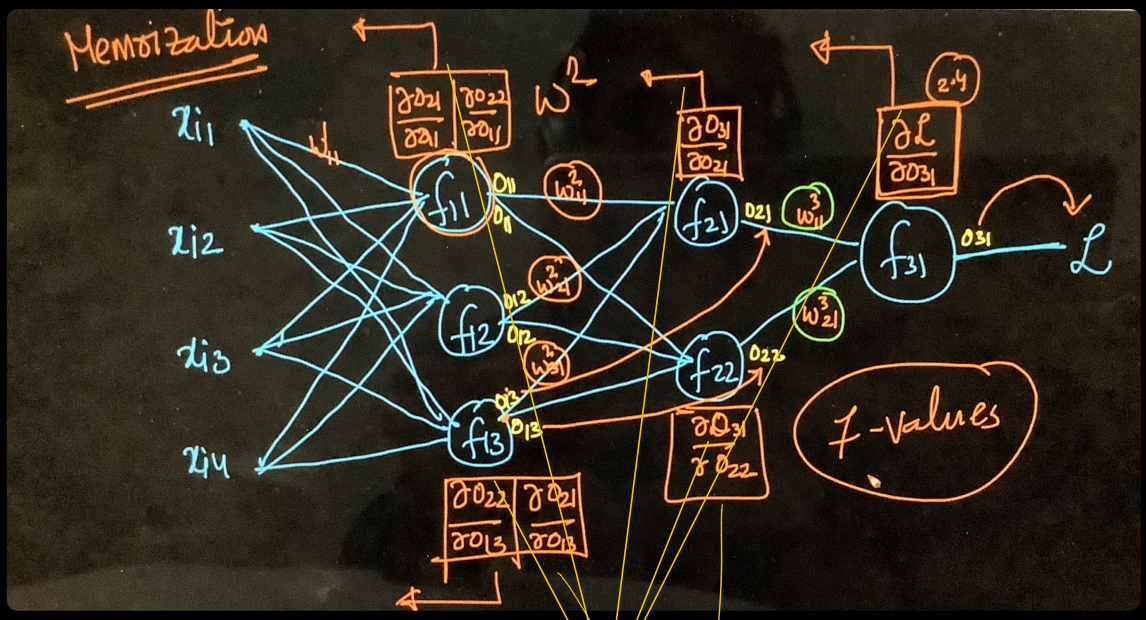
$$\frac{\partial L}{\partial w^1_{11}} = \frac{\partial L}{\partial O_{31}} \cdot \left[ \frac{\partial O_{31}}{\partial O_{21}} \cdot \frac{\partial O_{21}}{\partial O_{11}} \cdot \frac{\partial O_{11}}{\partial w^1_{11}} + \frac{\partial O_{31}}{\partial O_{22}} \cdot \frac{\partial O_{22}}{\partial O_{11}} \cdot \frac{\partial O_{11}}{\partial w_{11}} \right]$$

→ This involves computing a lot of gradients.

There's a surity that one gradient will be computed many times. This is why we use memoization → store the grades in a hash table and use those instead of calculating → only works for the current iteration.

↓ Dynamic Programming

them again & again
for entire $\nabla L$




store them like this.
It helps a TON with
speed. Uses only
slightly more memory

Chain rule + memorization.

↓

BOOM!

↓

Backpropagation!

① Get gradient, update weight matrix

② Update matrix closest to output first and then the one at the back.

③ Do this for every $D = \{x_i, y_i\}_{i=1}^{N}$

④ Stop when $(w_{ij}^k)_{new} \approx (w_{ij}^k)_{old}$

↳ convergence.

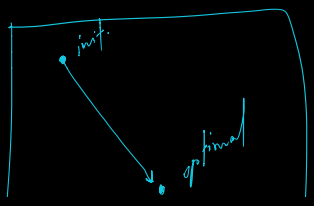
* Backprop only works if the activation functions are differentiable. Otherwise the whole idea fails.

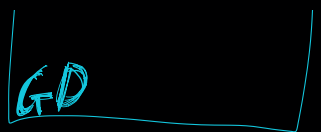
→ Computing the derivatives is the most time consuming step. So the activation functions need to be easily differentiable.

→ Instead of doing backprop for every point, $\Sigma$ Loss over a batch of points and then do backprop.

→ kinda the same thing but a TON faster.

Here, the term "stochastic" comes from the fact that the gradient based on a single training sample is a "stochastic approximation" of the "true" cost gradient. Due to its stochastic nature, the path towards the global cost minimum is not "direct" as in Gradient Descent, but may go "zig-zag" if we are visuallizing the cost surface in a 2D space. However, it has been shown that Stochastic Gradient Descent almost surely converges to the global cost minimum if the cost function is convex (or pseudo-convex)[1].

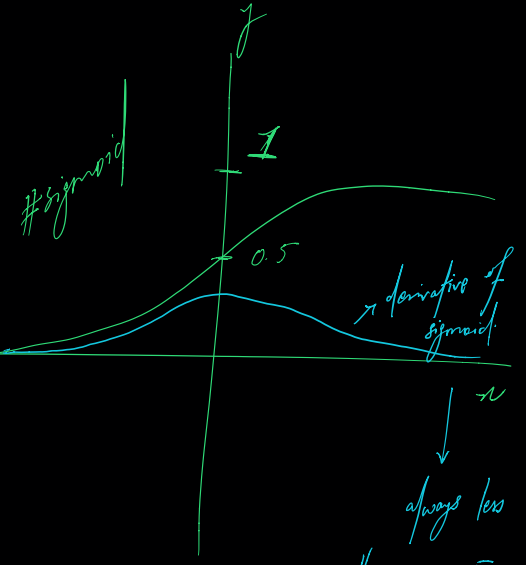



# Activation functions

① Differentiable

② Easy to differentiate

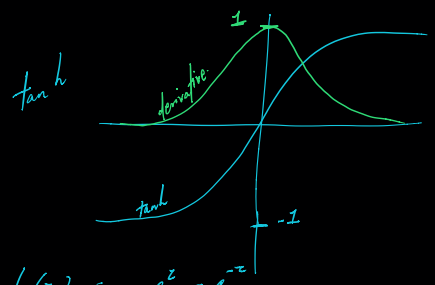



$$\sigma(z) = \frac{1}{1+e^{-z}}$$

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

vanishing gradient !!

Derivative is expressed in terms of the f" itself. So it can be used for forward as well as back prop.




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

→ fucking expensive (computationally) to calculate all these $e^z$ & $e^{-z}$

$$\frac{d}{dz}\tanh(z) = 1 - \tanh^2(z)$$

① Range is -1 to 1 compared to 0 to 1 for sigmoid

② Derivative still less than 1 ⇒ Vanishing Gradient

# Vanishing Gradient:

Derivatives < 1

∴ Chain rule makes them << 1

∴ $W_{new} \approx W_{old}$

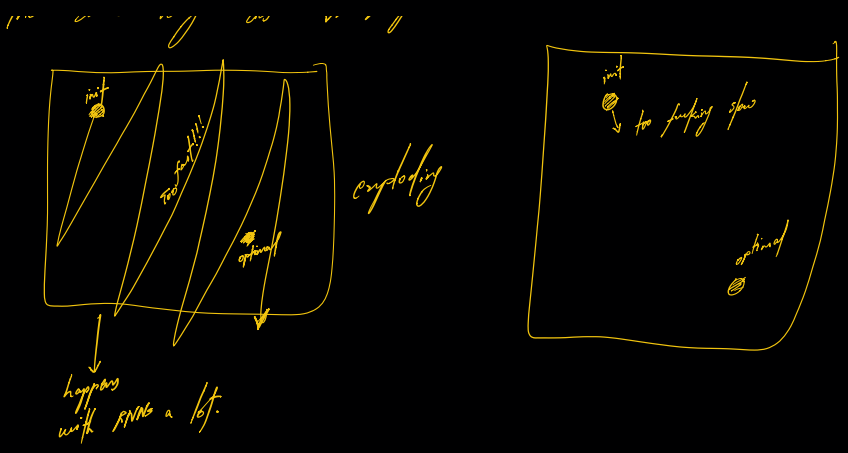
kinda remains the same → no realistic gradient with more hidden layers → because multiple values < 1 are multiplied.

∴ These activations are bad

↓

gradient cannot propagate down the network.

There's also something called exploding gradient, which also fucks up our Wnew the same way as vanishing




→ more parameters
allow to fit more complex stuffs

we need to have the right amount
of parameters to prevent over/under fit

→ with NNs, it's always overfitting.

↓

we can use dropouts,
Batch Normalization,
L1/L2 regularization, etc. Or all of them
together.

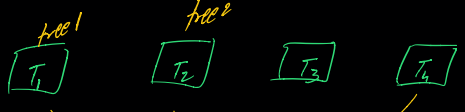
L1 regularization creates sparsity in weights

Most deep learning is empirical.
↳ experiments first → the engineer's way
↳ theory will be built later on.

↓

too much time, too slow.

=> Random Forest.




each of them has a random subset of features, rows & columns

results = majority vote.

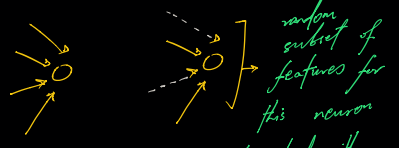
-> Each tree may overfit but a collection is used to regularize the DT

# Dropout is like applying the same philosophy to NNs. $0 \leq p \leq 1$

$p = 0.2$ → chosen randomly for each iteration.

That means 20% of the neurons are inactive for each layer.

-> This is like trying out a random subset of features.




-> There is a mathematical proof of why this thing works. => not worth the time investment

But we care about the results for now.

-> Dropout is only for training time., not during runtime.