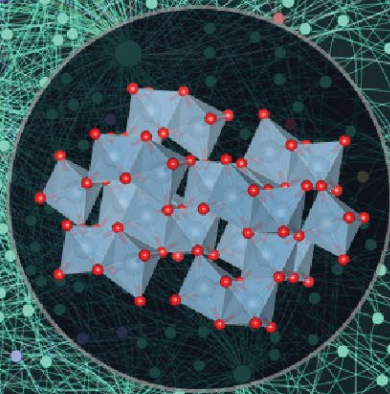
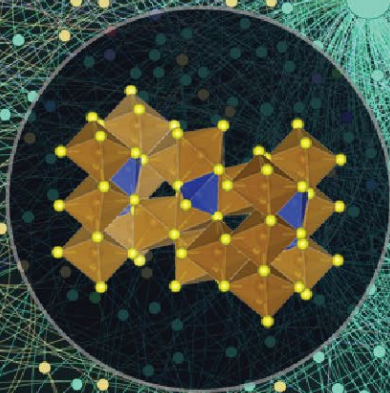
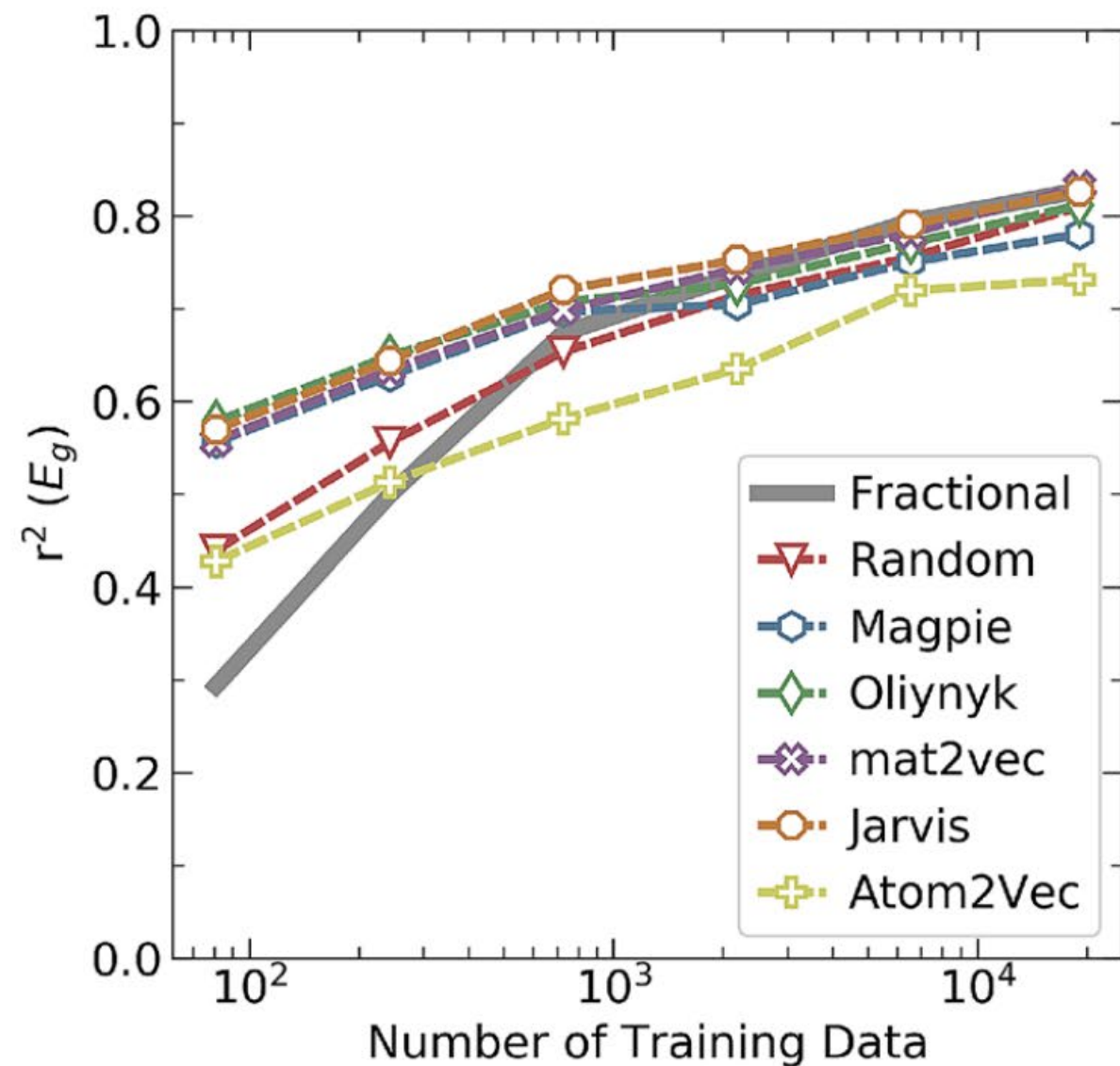
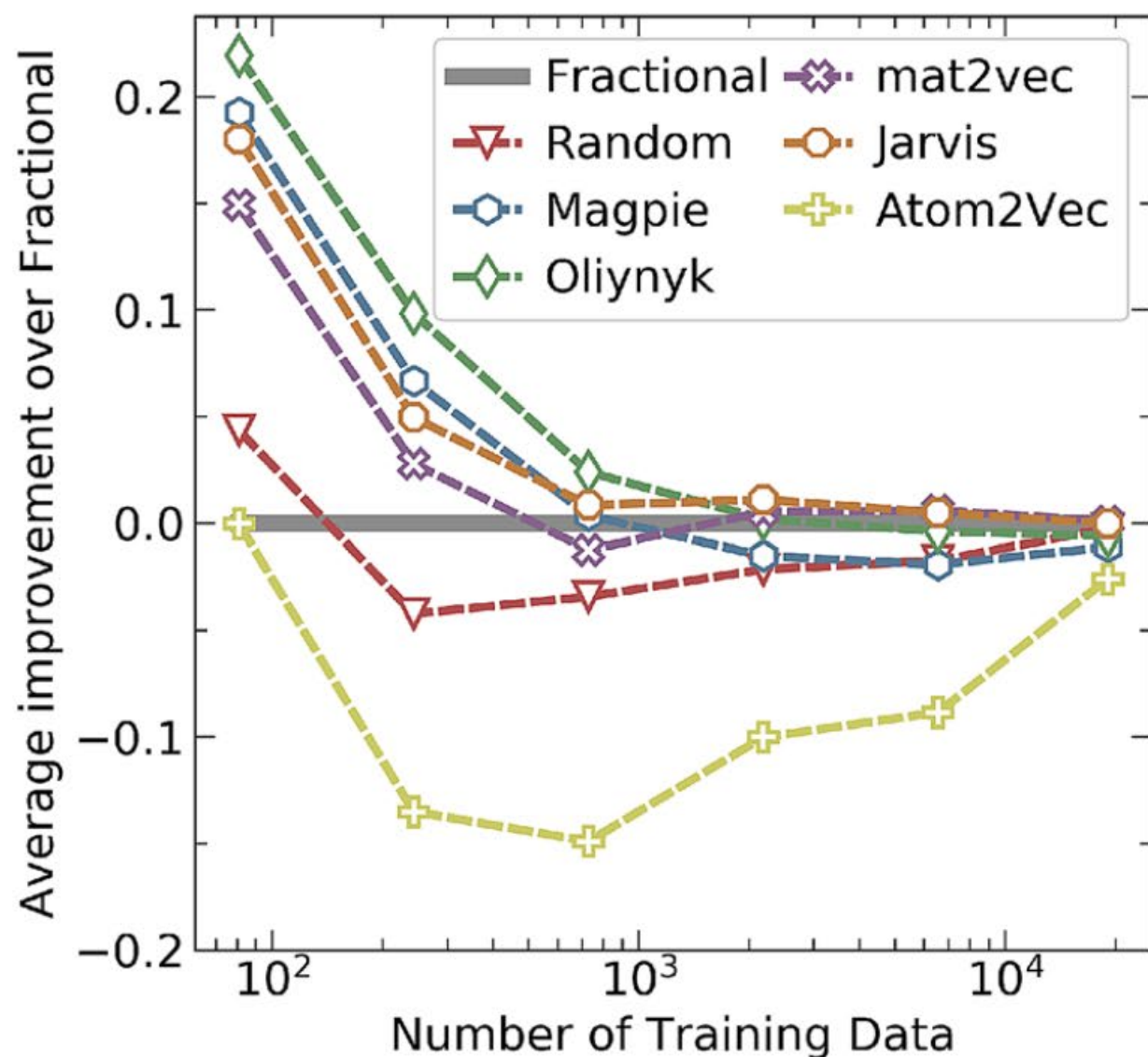


neural networks



Neural networks have emerged as one of the most powerful machine learning algorithms



Neural networks are meant to mimic how neurons in the brain pass information

Major differences:

Size (86 billions vs 10-1000)

Topology (neurons not really connected)

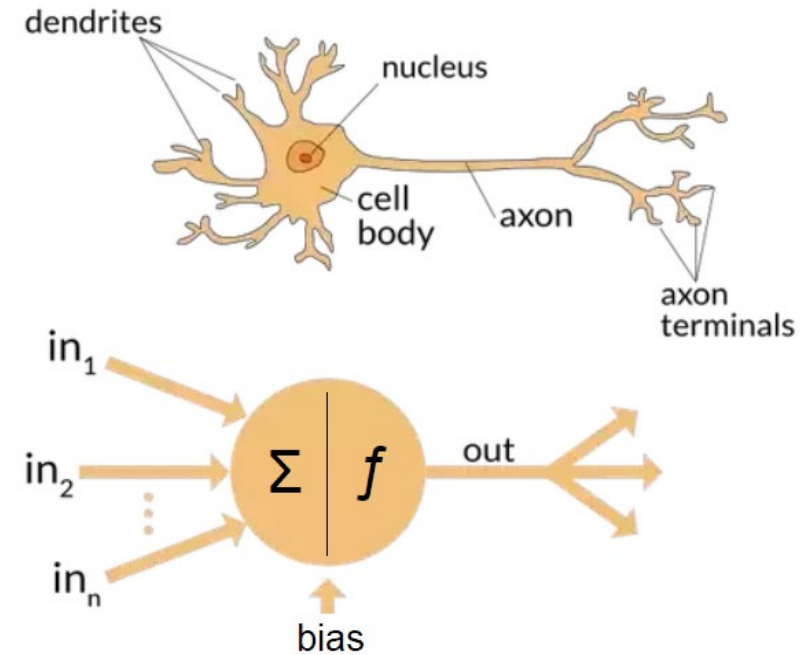
Speed (much faster in computer, and no rest periods)

Fault tolerance (artificial has no redundant info wired into system)

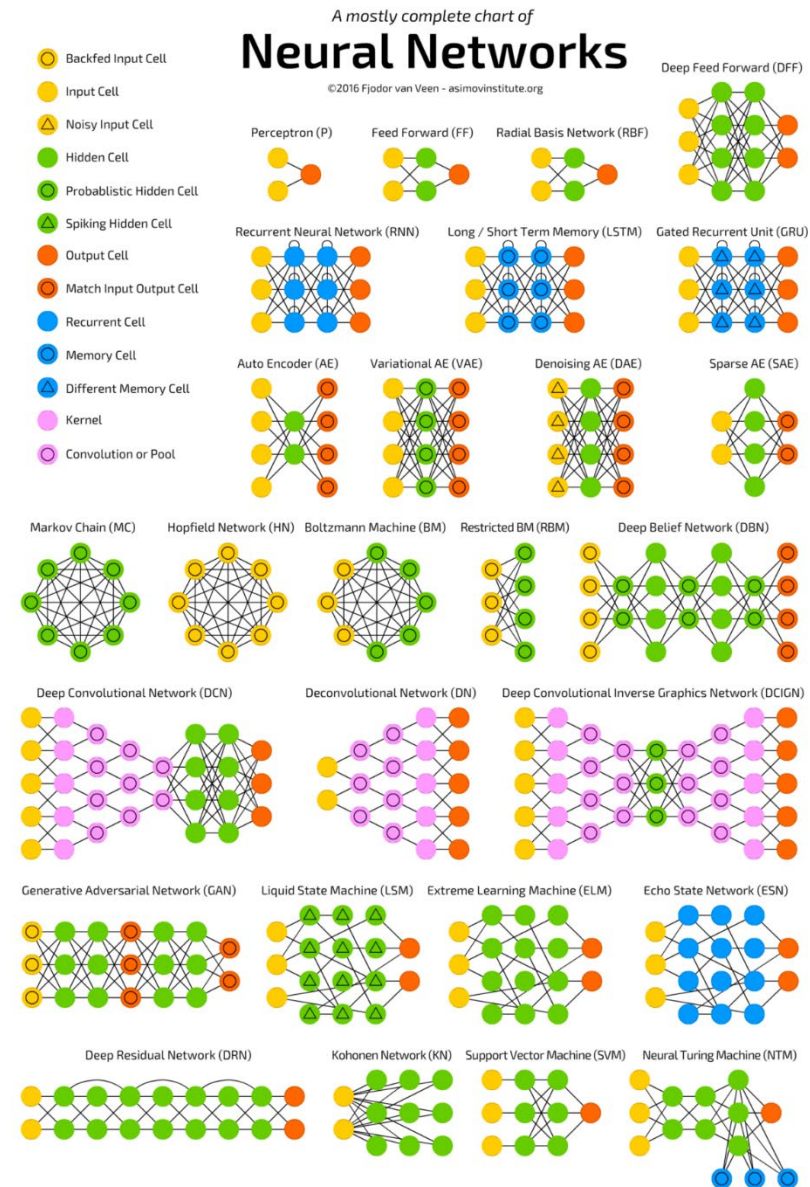
Power consumption (brain runs at 20W, Nvidia GeForce Titan X GPU is 250W)

Signal (0 or 1 for brain, no intermediate)

Learning (all at once or reinforced over time, predefined model or one that evolves)



Neural networks come in different flavors



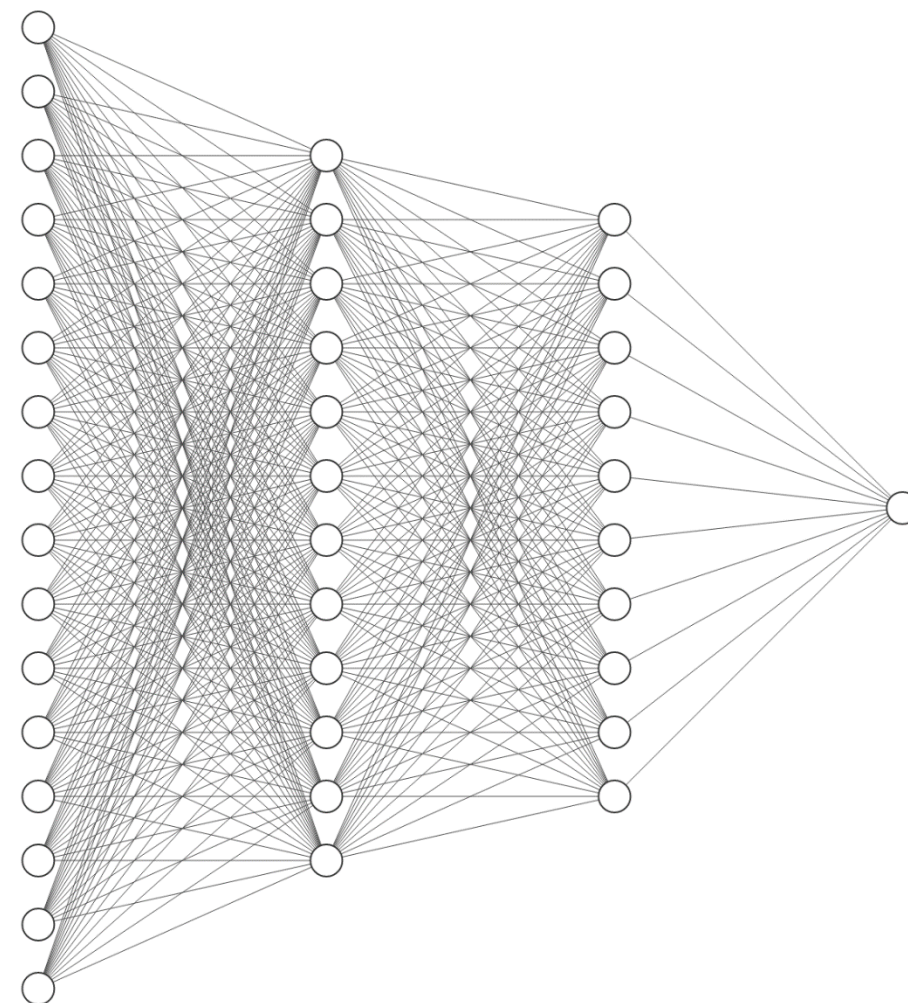
Neural networks are made up of layers of interconnected nodes

Nodes store numbers

- “activated” with a value between 0 and 1
- Functions that pass information

End result of network is a prediction
(regression or classification)

Do intermediate layers build up in explainable
way to the outcome you obtain?



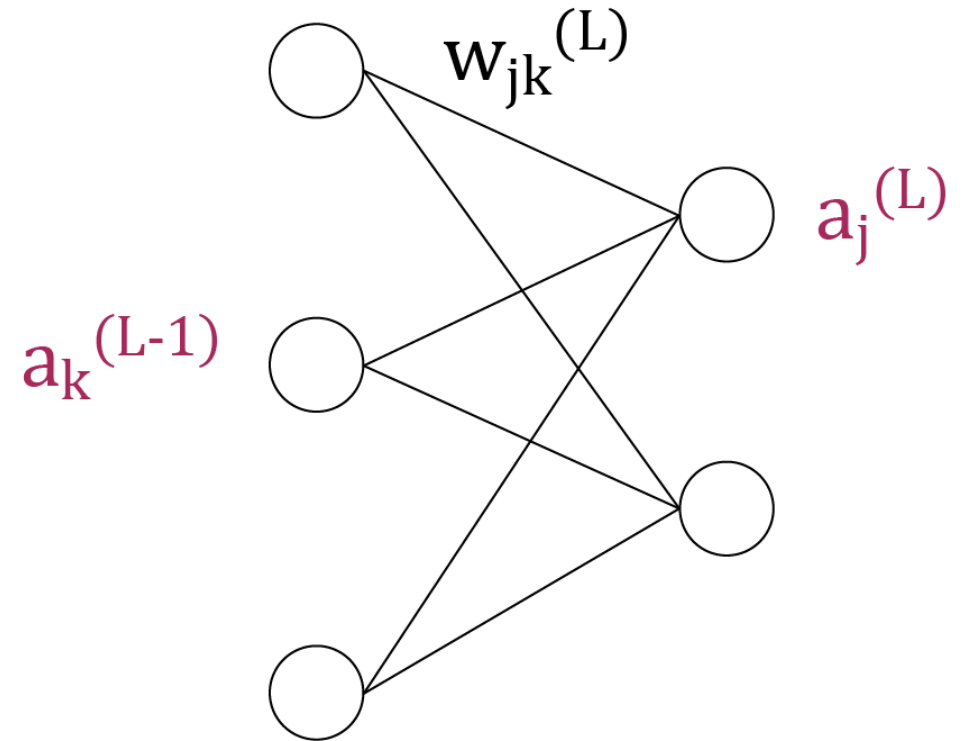
Neural networks are made up of layers of interconnected nodes

- Each node is fully connected to previous layer
- Information is passed and summed to new node
- Node value is “squished” to between 0 and 1 via an activation function

$a_0^{(1)}$ means 0-th node of first layer

$$a_0^{(1)} = \sigma \left(w_{0,0}a_0^{(0)} + w_{0,1}a_1^{(0)} + \dots + w_{0,n}a_n^{(0)} + b_0 \right)$$

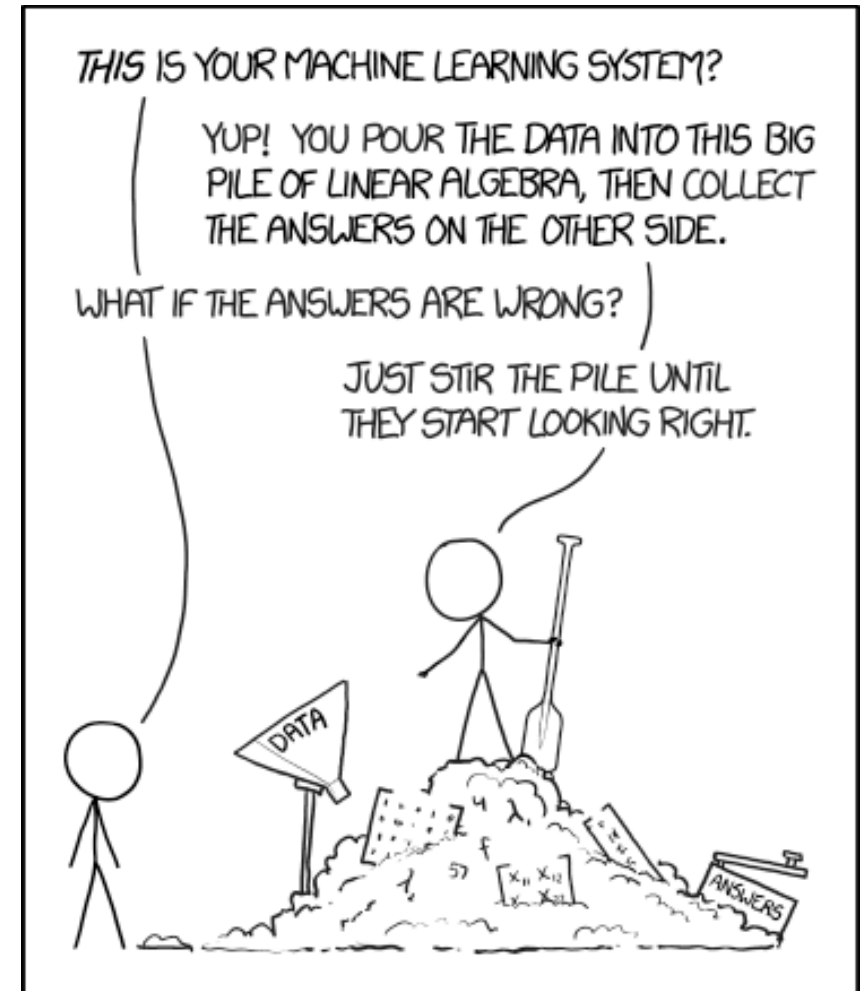
Tunable parameters are weights and biases!



Machine learning is just linear algebra

$$\sigma \left(\begin{bmatrix} w_{0,0} & w_{0,1} & \dots & w_{0,n} \\ w_{1,0} & w_{1,1} & \dots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \dots & w_{k,n} \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix} \right)$$

$$a^{(1)} = \sigma(Wa^{(0)} + b)$$



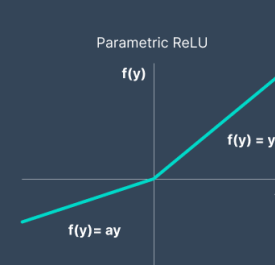
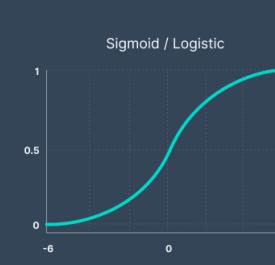
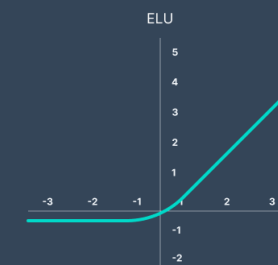
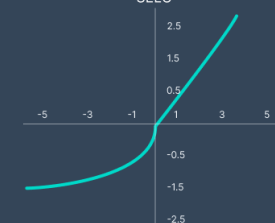
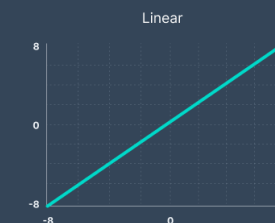
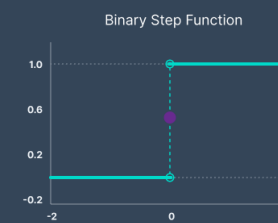
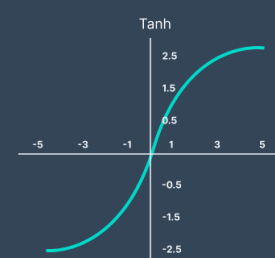
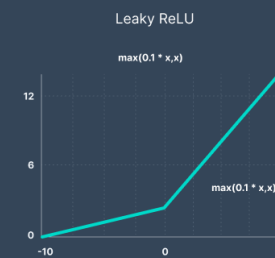
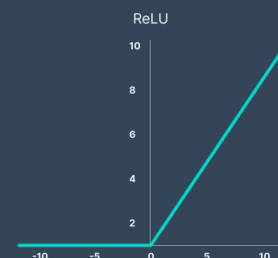
Many different activation functions are available

Sigmoid was a popular early choice, but is a slow learner

ReLU (Rectified linear unit) has become much more common

$$\text{ReLU}(a) = \max(0, a)$$

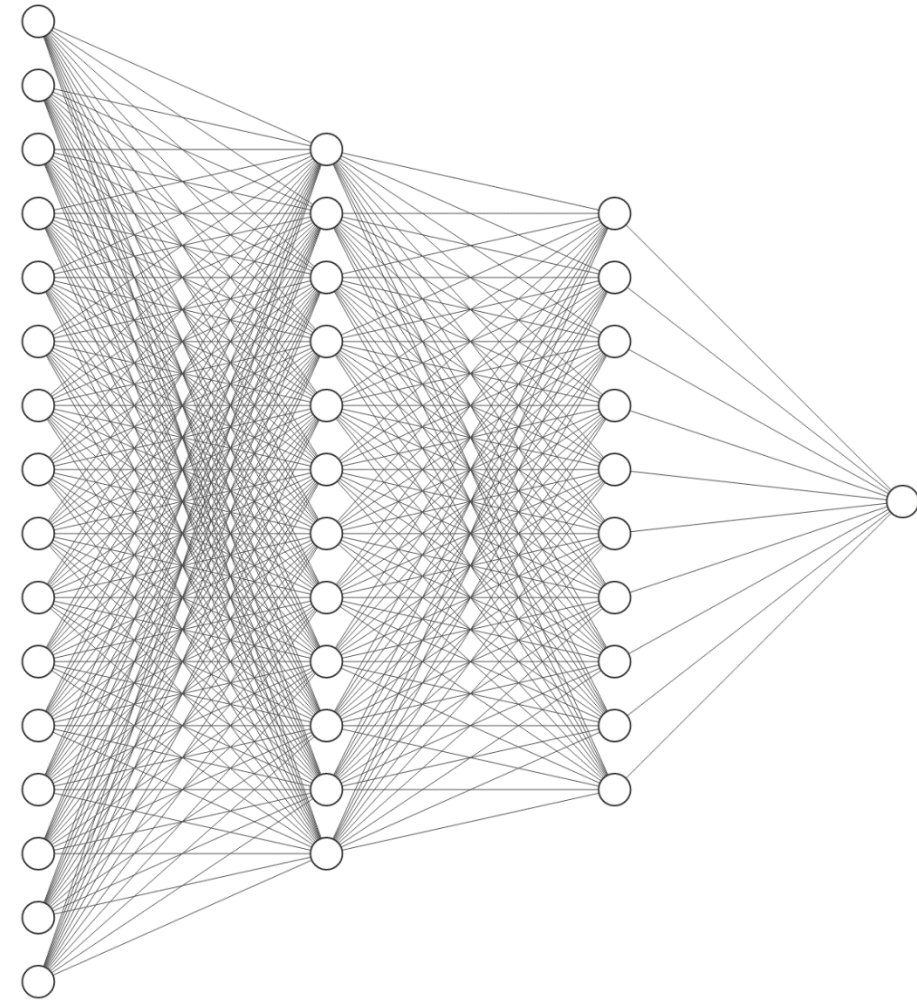
Inactive before some threshold, $f(a)=a$ after threshold



How does a neural network learn then?

- Randomly initializing all weights and biases
- Introduce a cost function
 - Add up the square of the differences of each node in final layer
 - “the cost of a single training run”
 - Calculate the average cost over all of the data available to train from
- Adjust the free parameters in order to minimize our cost function

How???



A gradient allows us to find the most efficient minimization of our cost function

- Calculate the slope of our function to decide where to move next
- Beware of local vs global minimum
- Make step size proportional to the slope!

Learning steps:

Calculate gradient

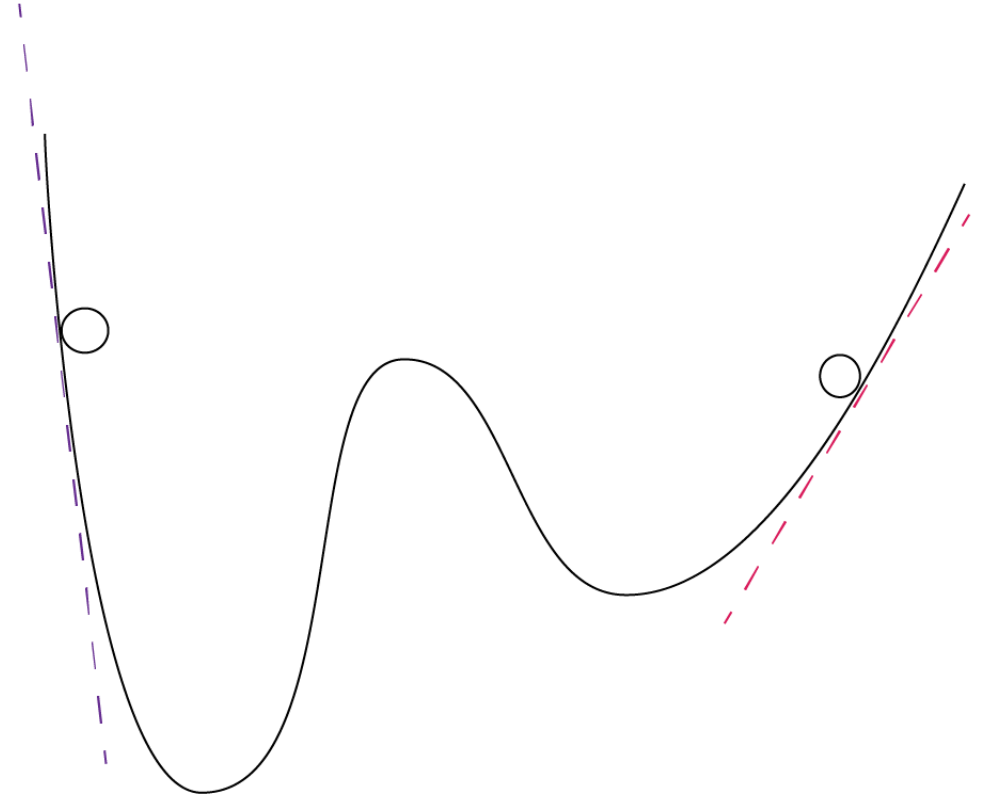
Take step in negative gradient direction

Repeat

Gradient just gets multiplied by the parameters to nudge these all in the right direction

$$-\nabla C[\vec{W}]$$

if we treat the weights as a vector, it just changes the direction!



Backpropagation is the process of applying gradients to update our free parameters

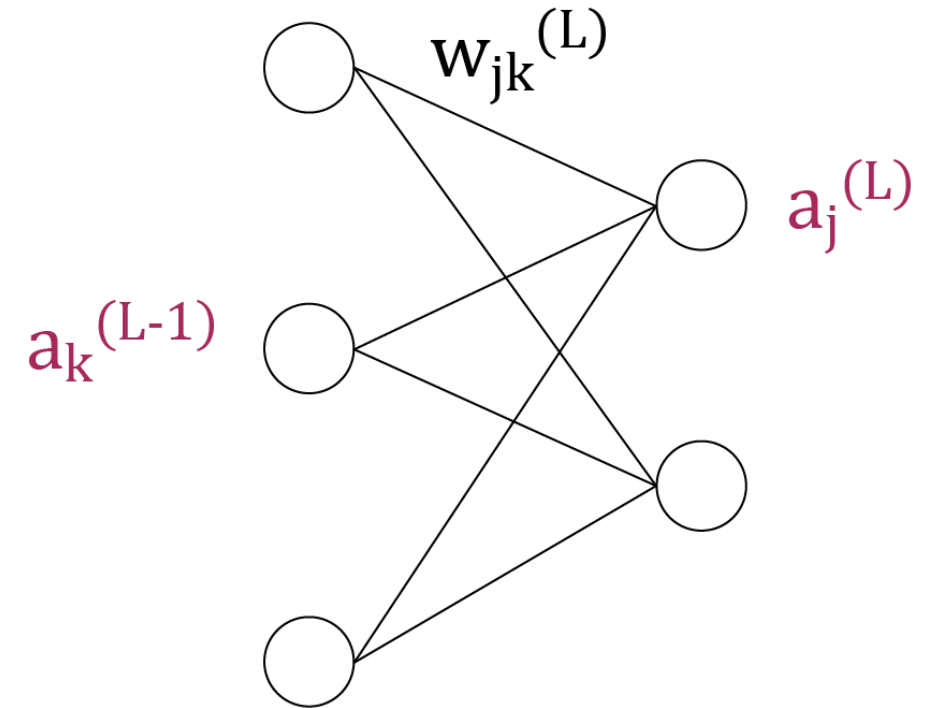
The magnitude of each element in your gradient tells you how sensitive that weight is for your cost function

Remember: $a^{(L)} = \sigma(Wa^{(L-1)} + b)$

To increase a final node we can...

1. increase bias
2. increase weights
3. ~~change~~ influence activations from previous layer

The corrections of each node in our final output get averaged

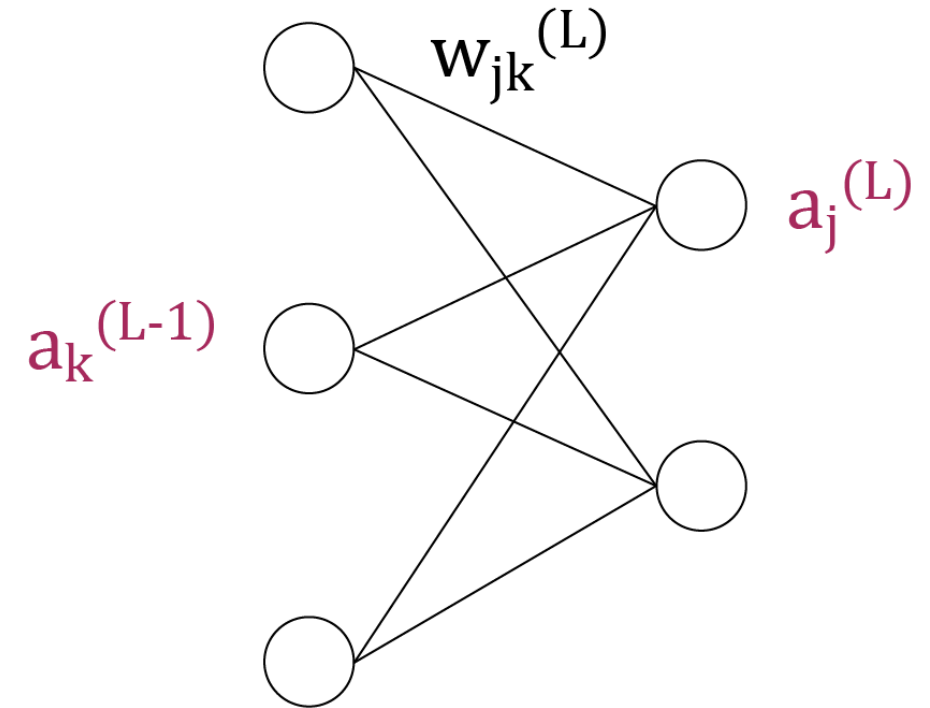


Backpropagation moves from final desired output backwards

For each step you are getting a list of nudges you want to happen to previous layer, so collect them for all weights and biases in whole network

Generate this list of nudges for each and every sample in training data and then average across all samples

This average is the negative gradient of the cost function!!! (or at least something proportional to it)



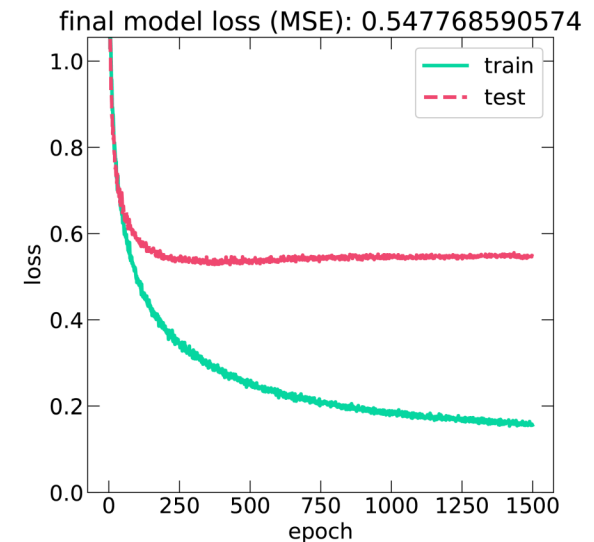
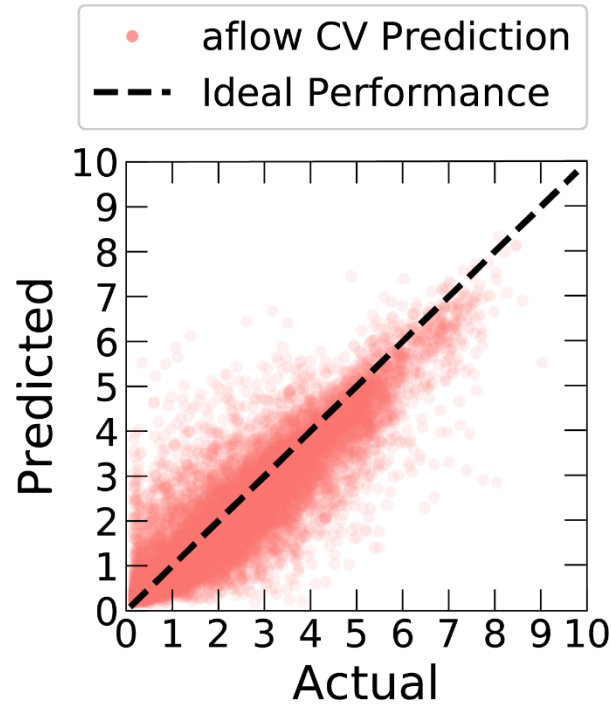
Backpropagation would be way too slow if we did every step on every sample

To speed training up, we perform calculations on mini-batches

- “stochastic gradient descent”
- Randomly shuffle data and then divide into mini-batches
- Compute a step (gradient calculation) according to mini batch
- Pretty good approximation, but not technically the steepest descent, but way faster

As you train you actually go through your whole dataset multiple times

- Epoch is when entire dataset is passed through one time



Let's look at the math of backpropagation in a really simple network



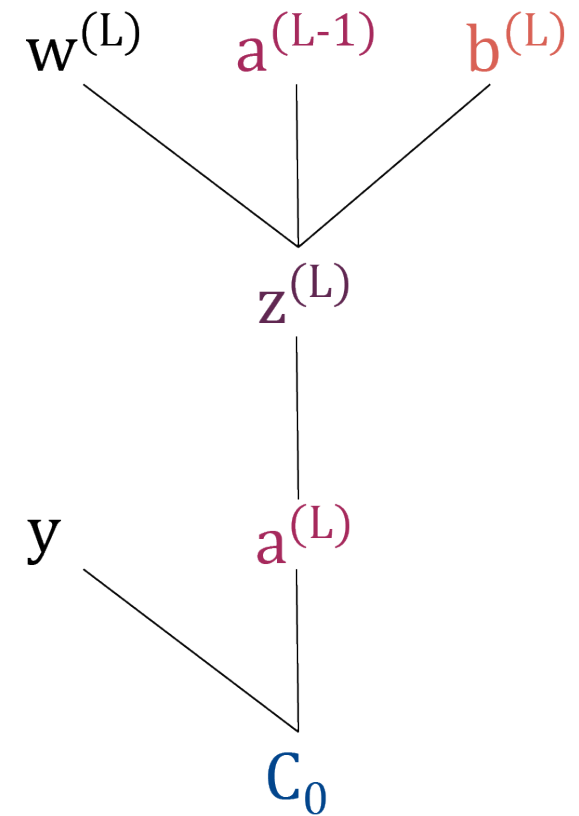
What free parameters do we have?

$$C(w_1, b_1, w_2, b_2, w_3, b_4)$$

How sensitive is our cost function to these variables?

$$C_0(\dots) = (a^{(L)} - y)^2$$

$$a^{(L)} = \sigma(w^{(L)} a^{(L-1)} + b^{(L)}) = \sigma(z^{(L)})$$



Derivatives let us calculate how much to change our parameters to affect cost function

These happen sequentially

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_0}{\partial a^{(L)}}$$

Derivatives

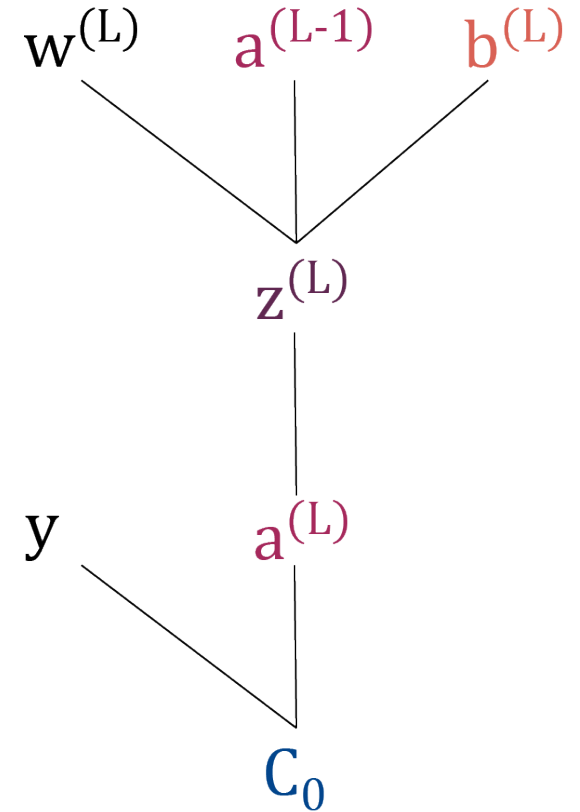
$$\frac{\partial C_0}{\partial a^{(L)}} = 2(a^{(L)} - y)$$

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(L)})$$

$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$$

Then average across all training examples

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{1}{n} \sum_n 2(a^{(L)} - y) \sigma'(z^{(L)}) a^{(L-1)}$$



Derivatives let us calculate how much to change our parameters to affect cost function

$\frac{\partial C_0}{\partial b^{(L)}}$ is nearly identical

These happen sequentially

$$\frac{\partial C_0}{\partial b^{(L)}} = \frac{\partial z^{(L)}}{\partial b^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_0}{\partial a^{(L)}}$$

Derivatives

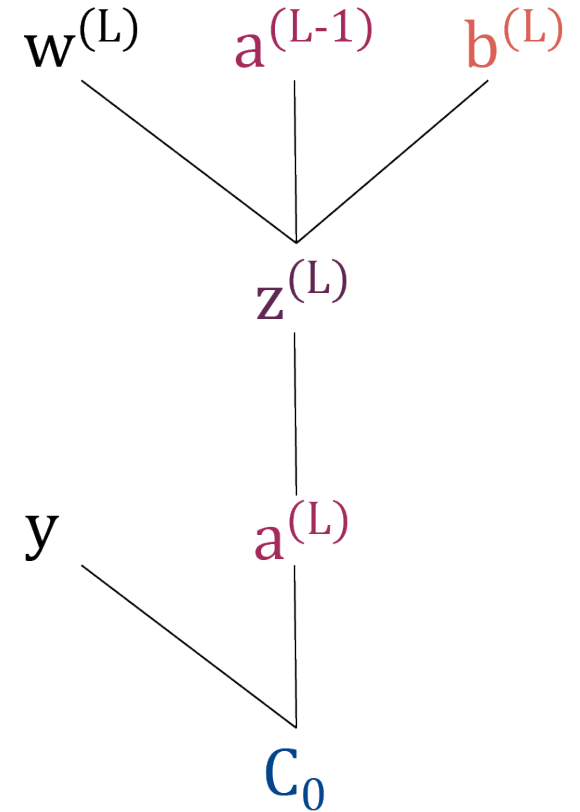
$$\frac{\partial C_0}{\partial a^{(L)}} = 2(a^{(L)} - y)$$

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(L)})$$

$$\frac{\partial z^{(L)}}{\partial b^{(L)}} = 1$$

Then average across all training examples

$$\frac{\partial C_0}{\partial b^{(L)}} = \frac{1}{n} \sum_n 2(a^{(L)} - y) \sigma'(z^{(L)})$$



Remember, gradient depends on all of our parameters, not just one

Gradient is over all tunable parameters

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w^{(1)}} \\ \frac{\partial C}{\partial b^{(1)}} \\ \vdots \\ \frac{\partial C}{\partial w^{(L)}} \\ \frac{\partial C}{\partial b^{(L)}} \end{bmatrix}$$

For more realistic networks (more than 1 node, it's not that much crazier...)

Backpropagation for arbitrary layer size

Cost function

$$C_0 = \sum_{j=0}^{n_L-1} \left(a_j^{(L)} - y_j \right)^2$$

Our z becomes

$$z_j^{(L)} = w_{j0}^{(L)} a_0^{(L-1)} + w_{j1}^{(L)} a_1^{(L-1)} + w_{j2}^{(L)} a_2^{(L-1)} + b_j^{(L)}$$

Activation of node in final layer is then just

$$\sigma \left(z_j^{(L)} \right)$$

Chain rule cost sensitivity expression becomes

$$\frac{\partial C_0}{\partial w_{jk}^{(L)}} = \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial C_0}{\partial a_j^{(L)}}$$

Tuning neural networks can be tricky

Parameters:

Architecture (number of nodes, number of layers)

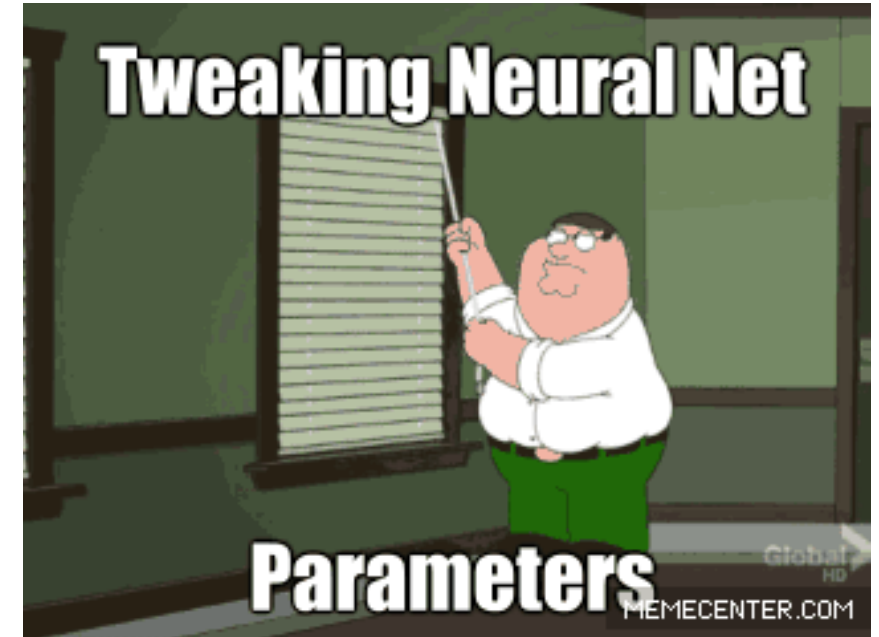
Learning rates and weights (7 optimizers to choose from)

Activation function

Batch size (small is faster, higher variance; large is slow, less variance)

Epochs used for training

Regularization (dropout)



advanced deep learning

