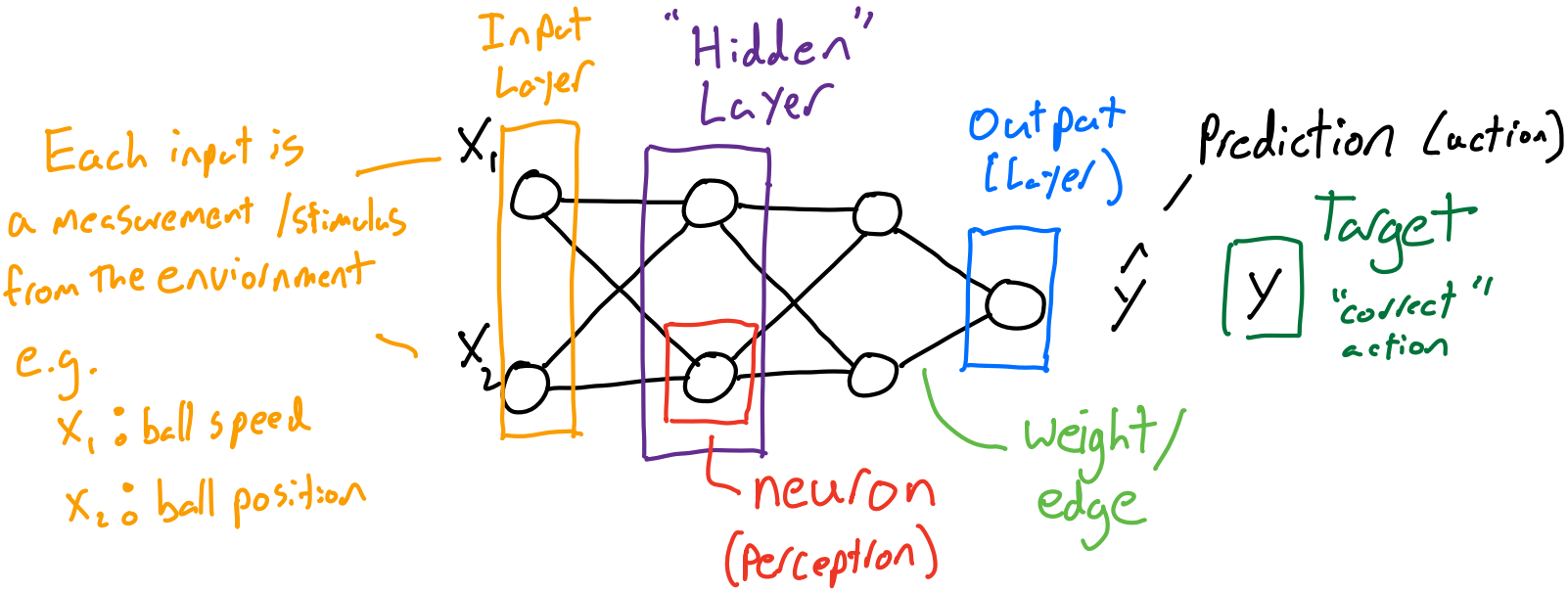
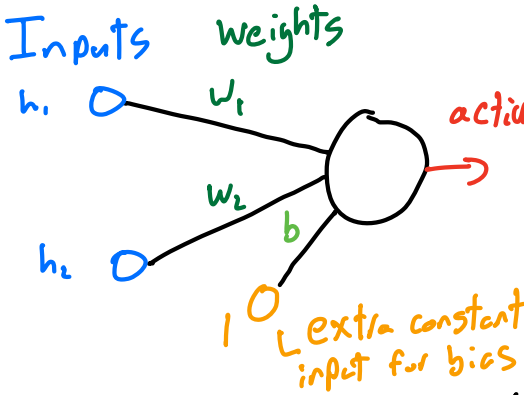
Anatomy of a neural network



A single neuron (perceptron)

- Takes in Inputs from the previous layer
- Computes an activation using its weights (and bias)



$$\text{activation} = \sigma\left(\sum_{\text{inputs}} w_i h_i + b\right)$$

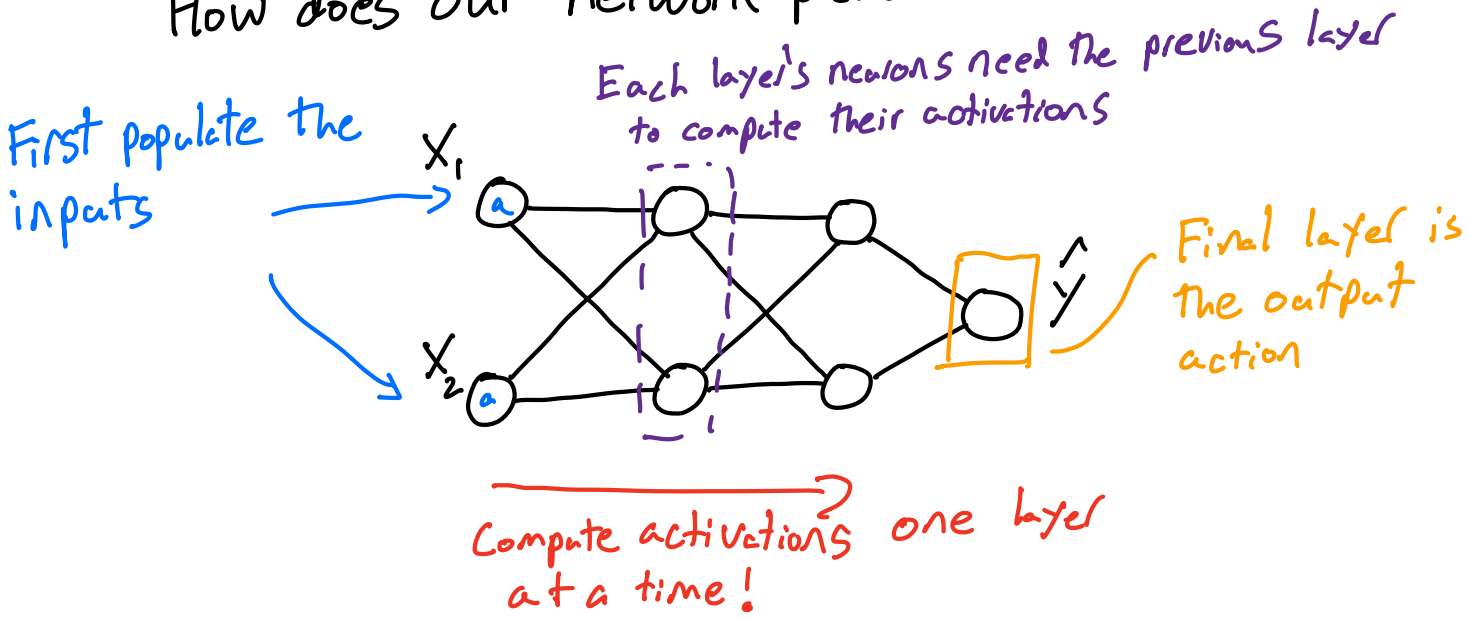
"activation function":

sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$, derivative $\frac{d}{dx}\sigma(x) = \sigma(x)(1-\sigma(x))$

Computing a prediction

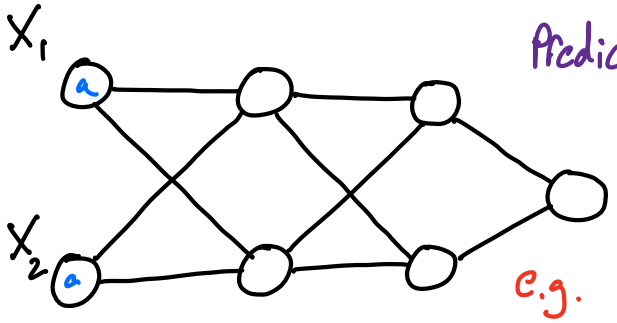
We get measurements x_1, x_2 from our environment

How does our network predict an action?



Back propagation Update

Goal: Given an example (x, y), we want our network's prediction to match y



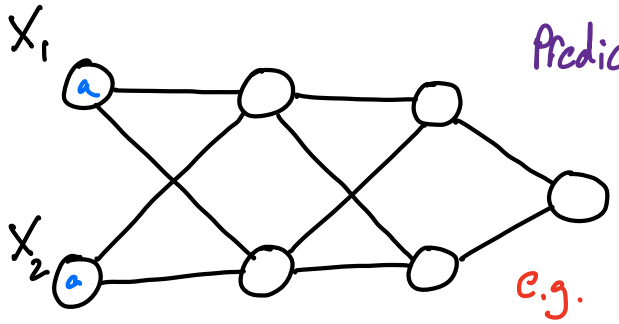
Note: With our "soft" sigmoid activation $\hat{y} \in [0, 1]$

We can round to get a concrete action:

$\begin{cases} \hat{y} \geq 0.5 \rightarrow \text{do action 1} \\ \hat{y} < 0.5 \rightarrow \text{do action 0} \end{cases}$

Accuracy: $\frac{1}{N} \sum [\text{round}(\hat{y}_i) = y_i]$

for x_i, y_i in dataset of size N



Error: $(y - \hat{y})$

- difference between $\hat{y}$ and y, tells us how to change $\hat{y}$

Positive error $(\underset{y}{1} - \underset{\hat{y}}{0.2}) = 0.8$, need to increase $\hat{y}$

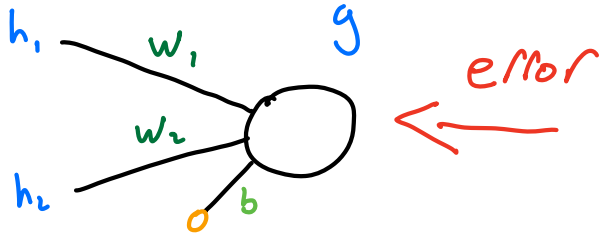
Negative error $(\underset{y}{0} - \underset{\hat{y}}{0.6}) = -0.6$, need to decrase $\hat{y}$

Updating weights for a neuron

Goal: Update weights based on error
(move it towards 0)

Neuron gets error based on its activation and the target

Let g = activation "for neuron g"



Conceptually

Weight update = γ [Direction activation should change] x [Effect of weight on activation]

γ: Scaling factor "learning rate"

Direction activation should change: Error

Effect of weight on activation: derivative of activation w/ respect to weight

For output layer:

error $= (y - \hat{y})$

$$W_i \leftarrow W_i + \gamma \text{ error } \frac{dg}{dw_i}$$

$$\frac{dg}{dw_i} = \frac{d}{dw_i}\left[\sigma\left(\sum_{i=1}^{inputs} w_i h_i + b\right)\right]$$

Let $a = \sum_{i=1}^{inputs} w_i h_i + b$

$$\frac{dg}{dw_i} = \sigma(a)(1-\sigma(a))\frac{da}{dw_i}$$

Using
$$\frac{d}{da}\sigma(a) = \sigma(a)(1-\sigma(a))$$
$$\frac{d}{dx}f(g(x)) = \frac{df}{dg}\cdot\frac{dg}{dx}$$

$$\frac{da}{dw_i} = \frac{d}{dw_i}\left(\sum_{i=1}^{inputs} w_i h_i + b\right) = h_i$$

Therefore

activation!

$$W_i \leftarrow W_i + \gamma \underbrace{\text{error}\,\sigma(a)(1-\sigma(a))}_{\text{"delta" in lab}} h_i$$

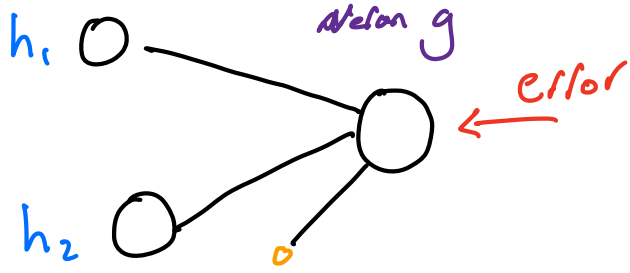
Note: For bias,

$$b \leftarrow b + \gamma\,\sigma(a)(1-\sigma(a))\,(1)$$

The Crucial Recursive Step!

h_1 and h_2 may also be neurons, They need an error to update their weights!

"How should we change h_1, h_2?"



Conceptually:

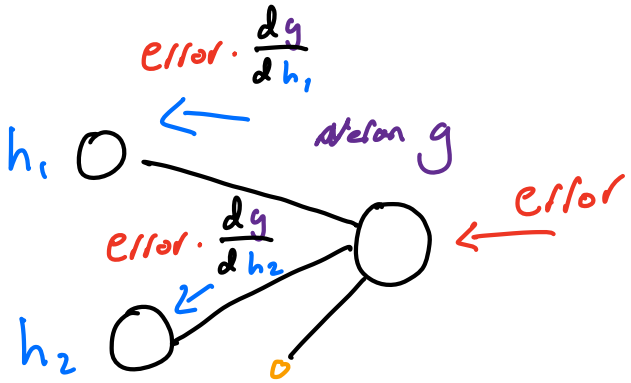
Direction we should change h_i = [direction we should change activation(g)] × [effect of h_i on activation(g)]

↑ error for h_i

↑ error for g

↑ $\frac{dg}{dh_i}$

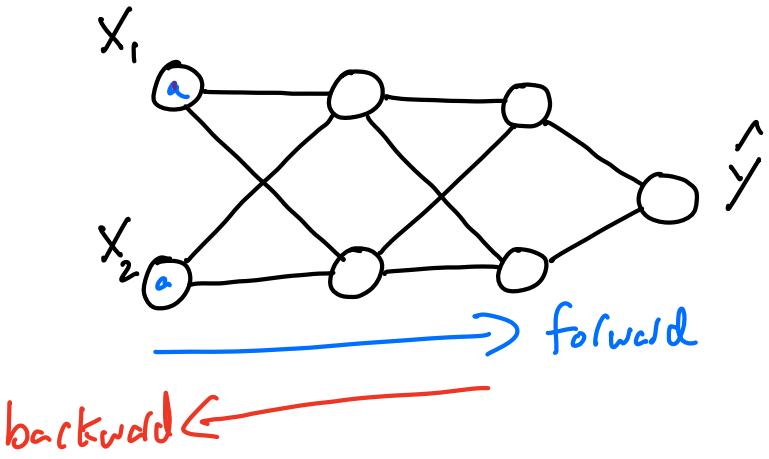
Pass updated errors to input neurons!



$$\frac{dg}{dh_i} = \frac{d}{dh_i}\sigma\left(\underbrace{\sum_{i=1}^{\text{inputs}} w_i h_i + b}_{a}\right) = \sigma(a)(1-\sigma(a))\,w_i$$

↑ Same form as $\frac{dg}{dw_i}$, $w_i \Leftrightarrow h_i$

Update algorithm

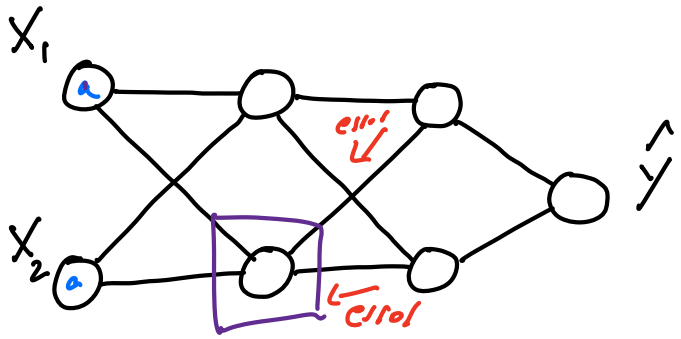


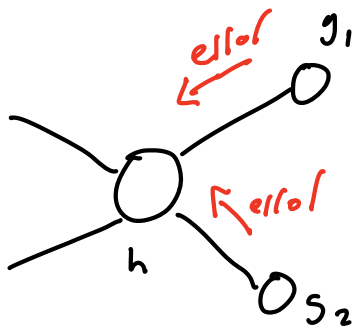
2 passes

Forward pass:
Compute activations 1 layer at a time from inputs to output

Backward pass:
Compute errors 1 layer at a time from output to input
+ update weights as we go

Notes:





A neuron might get multiple error updates if its activation is used more than once

Just add them! $\left[\frac{d}{dx} f(g(x), h(x)) = \frac{df}{ds}\frac{ds}{dx} + \frac{df}{dt}\cdot\frac{dh}{dx}\right]$

$$\text{error}_h = \text{error}_{g_1}\frac{dg_1}{dh} + \text{error}_{g_2}\frac{dg_2}{dh}$$

This Update is for a Single example!

To "train" a neural network we need to learn from lots of data.

Reminder: 2 options

"online learning" 1) send agent out into the world, have it update as it collects each new input/output example

"offline learning" 2) Collect a Dataset of examples $[(x_1, y_1), (x_2, y_2) \ldots (x_n, y_n)]$ ahead of time and update our network on each in turn (usually multiple times)

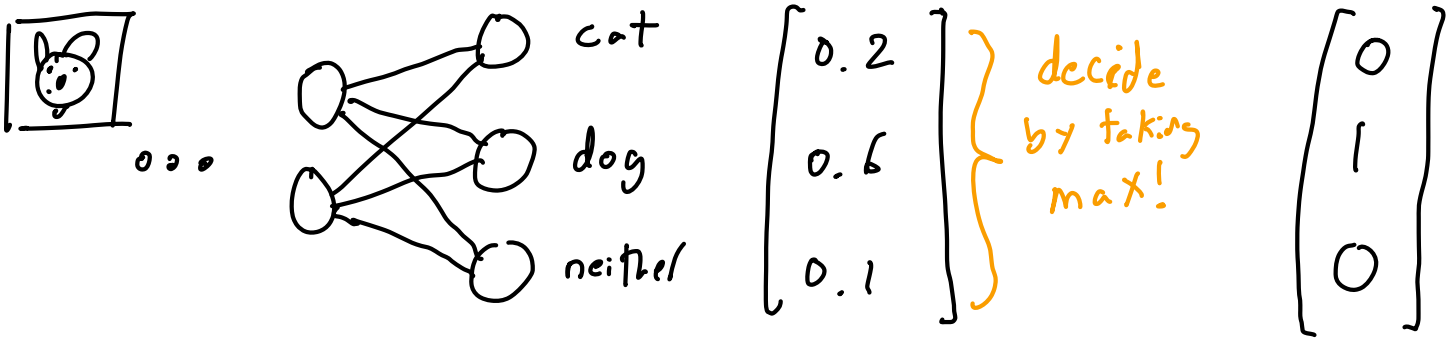
Handling > 2 possible outcomes

Example: Cat: 0 dog: 1 Neither: 2



Output neuron either fires or doesn't

Solution: Multiple outputs



Target (y) becomes a vector, 1 at index of correct response, 0 otherwise.

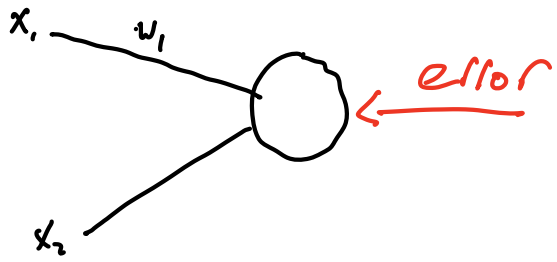
- Can still compute error at each output!

$$error_i = (y_i - \hat{y}_i)$$

Accuracy unchanged!

"Vanishing gradients"

A problem:



$$w_1 \leftarrow w_1 + \gamma \cdot \text{error} \cdot \sigma(a)(1-\sigma(a)) x_1$$

$$a = \sum_{i=1}^{\text{inputs}} x_i w_i + b$$

if $\sigma(a) = 0.999$,

$\sigma(a)(1-\sigma(a)) = (0.999)(0.001) \approx 0$ ← Basically no update!

Same thing if $\sigma(a) = 0.001$!

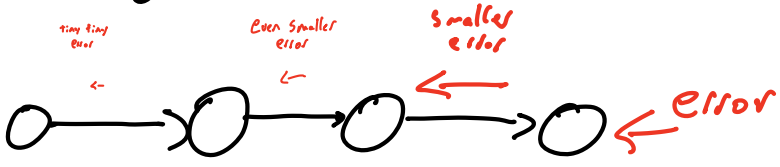
If w_1 is large we can get pushed to these extremes for any x_1!

derivative is slope of function at x



only a small region where derivative is large enough to allow updates

With multiple layers, issue can compound!



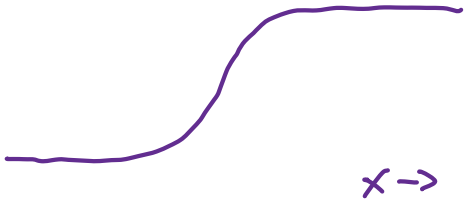
Rectified Linear activations

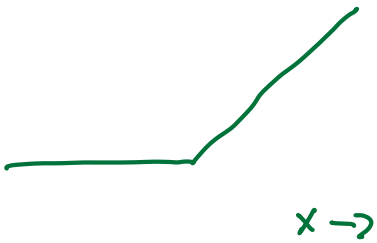
Solution: for hidden neurons, don't cap activation at 1. $[0,1] \rightarrow [0,\infty)$

Replace Sigmoid with ReLU:

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

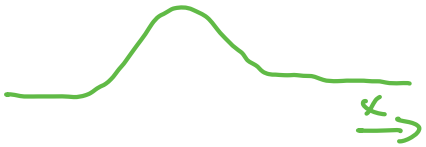
$$ReLU(x) = \max(x, 0)$$





$$\frac{d}{dx}\sigma(x) = \sigma(x)(1-\sigma(x))$$

$$\frac{d}{dx}ReLU(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$





Error only disappears on 1side

More like original perceptron update!

$$w_1 \leftarrow w_1 + \gamma \, error(a>0) \, x_1$$

Important for large networks!

Loss functions

We can also view error as a derivative:

$$-2(y-\hat{y}) = \frac{d}{d\hat{y}}(y-\hat{y})^2$$

We call $(y-\hat{y})^2$ a loss function, intuitively it gives us a value to minimize.

$\frac{dL}{d\hat{y}}$ tells us the effect of $\hat{y}$ on this loss, by moving along $-\frac{1}{2}\frac{dL}{d\hat{y}}$, we move toward smaller values.

Cross-entropy loss

We can use this idea to change our loss. Another common loss is "cross entropy"

$$-y \log \hat{y}$$

Applying neural networks in practice

Step 1: Identify our problem

What are our inputs? $[x_1, x_2 \ldots]$

- Should be a set of measurements about the state of the environment we want to react to.

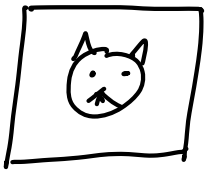
What is our Output? $[y]$

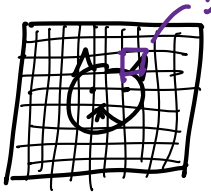
- Should be a decision we want to make based on our inputs

Example: Cat or dog?

Our agent should be able to decide if what its looking at is a cat or dog

Input: An Image





A set of pixel measurements, x_i is the brightness of pixel i

[Note: could be a lot of inputs e.g. 256x256

x_0
x_1
$\vdots$
x_{n-1}

Output: A Label

0: Cat

1: Dog

Step 2: Collect and process a Dataset

$$D: [(X_1, Y_1), (X_2, Y_2) \ldots (X_N, Y_N)]$$

Example: Find lots of images of cats and dogs

( , cat (0)), ( , dog (1)) ...

- Note: More data is generally better!

Tips for processing our collected data

1) The number of inputs for a network is fixed*, so examples need to be consistent

   Example: Need to resize/crop images so they're all the same size
   Don't mix color and greyscale images

2) We don't want too many inputs if they're not relevant

   Example: Lowering image resolution overall can reduce irrelevant detail

3) Normalize inputs to a standard range (often [0,1] or [-1,1])

$$X_i = \frac{X_i - X_{min}}{X_{max} - X_{min}}$$

   Example: Pixel values are often stored as 0-255

   Convert to [0,1]:

$$X_i = \frac{X_i}{255}$$

Step 3: Split the data into training and test sets

Dataset: $[(x_1, y_1), (x_2, y_2) \ldots (x_i, y_i)(x_{i+1}, y_{i+1}), \ldots (x_N, y_N)]$

Take 70-80% as "trainjng data"

Reserve 20-30% as "test data"

Goal: our test data will let us estimate how our model will perform in the wild.

- We won't use it to update the model so it will be "new"

Important: Always Shuffle your data before you split.

Example: If we collected our cats first, we might end up with a training set of all cats and a test set of all dogs :

Step 4: Train our network using back propagation

- We'll take multiple passes through our dataset ("Epochs"). We can stop early if we predict every action correctly

```
For epoch in NumEpochs:
    Ncorrect = 0  # Num. correct predictions this epoch
    For (x_i, y_i) in dataset:  # Should also shuffle before new epoch
        ŷ = network.predict(x_i)  # run forward pass
        network.backprop(y)  # run backward pass
        if ŷ == y:
            Ncorrect += 1
    if Ncorrect == N:  # Stop if we have 100% accuracy
        break
```

Lingo: Stochastic gradient descent

Track Loss as we go!

Step 6: Evaluate our network

Before we use our network, see how it does on our test data

```
N_correct = 0
for (x_i, y_i) in testDataset:
    ŷ = network.predict(x_i)
    if ŷ == y_i
        N_correct += 1
accuracy = N_correct / N
```

What do we do if accuracy is not good?

- -> Change network structure
- -> Change learning rate (γ)
- -> Get more data

Then re-train and evaluate again

Much more analysis in Machine learning course!

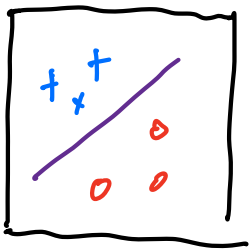
Step 7: Analyze Solution

In lab: Can examine all 10 weights for xor network

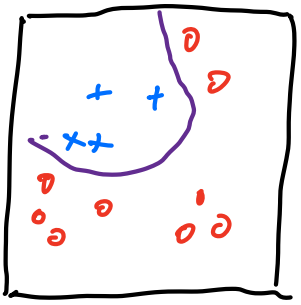
In practice, may be impossible as networks can have 1000's, millions, even billions (trillion's?) of weights.

Neural networks sumarized

Allow us to map continuous measurements to a discrete outcome



Perceptron



neural network

Extends perception to capture non-linear decisions!

- In fact, any boolean function $f(x_1, x_2 \ldots) \rightarrow 0/1$ can be learned by a neural network

Learn with stochastic gradient descent and back propagation