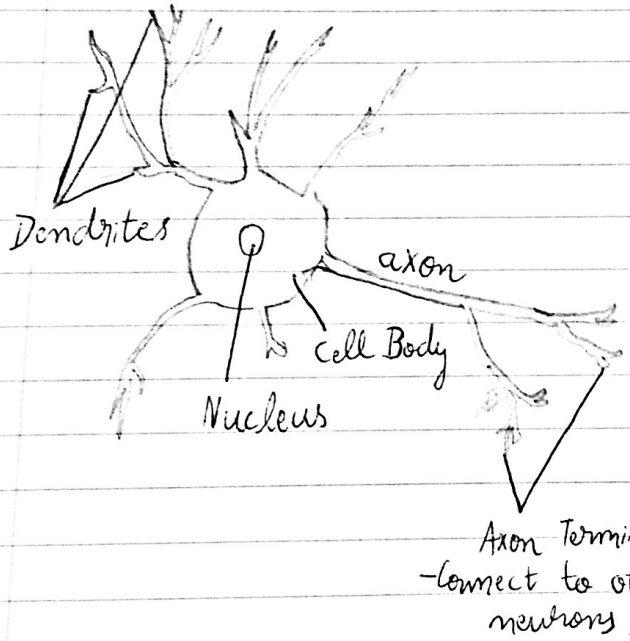
Emmagerd!
Finally!
Magic and superpowers!

Neural Networks !

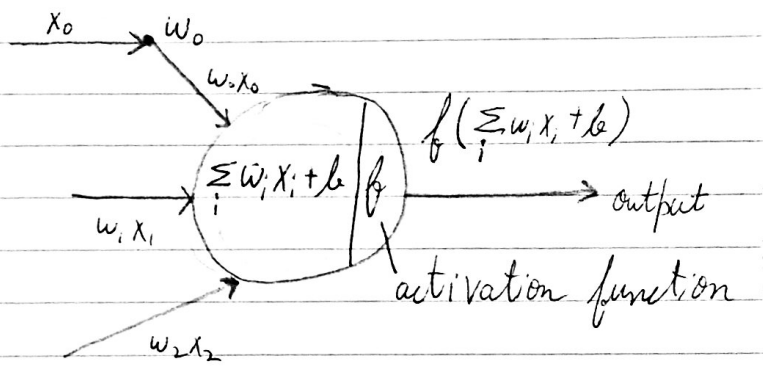
☆ Spoiler: It's not actually a network, just a series of operations. False advertising.

☆ Neuron: A neuron is a computational model of a function that accepts inputs and "fires" an output depending on some measure of strength of the inputs.

Biological Model



Computational Model



☆ Activation Function: Computes the output / "strength" of the sum of inputs (weighted inputs). This is a ~~the~~ core element of Neural Nets. More on this later.

☆ So essentially, Neuron (also called "Perceptron" or "Artificial Neuron") is a model of a function that accepts weighted inputs.

☆ Depending on Activation function, a Neuron can "act" as other computational machinery (?)

For example, choosing it to be sigmoid results in:

$\sigma(\sum_j w_j x_i + b) \rightarrow$ Binary Softmax classifier

$= P(y_i = 1 \mid x_i; w)$

Use hinge loss function (max-margin) $\rightarrow$ Binary SVM

☆ Activation Functions in Neural Nets:

1. Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$ $\rightarrow$ Squashes values to between 0 & 1. Range: $[0, 1]$

Disadvantages:

i) Saturates & kills gradients: Gradient is 0 at both 0 and 1 tails of sigmoid (because value becomes constant at 0 and 1 for large values of input, so differential is 0.) So gradient signal is 0 and network learns nothing

ii) Sigmoid output not zero-centered: Since sigmoid is always greater than 0, all gradients are either positive or all negative

2. Tanh: Hyperbolic Tangent. Range: $[-1, +1]$

$\tanh(x) = 2\sigma(2x) - 1$

3. RELU (Rectified Linear Unit): $f(x) = \max(0, x)$

→ Simple thresholding, but shown to be 6x better for convergence of SGD.

→ However, they are fragile. May never be activated and can "die". ~~If network can die.~~

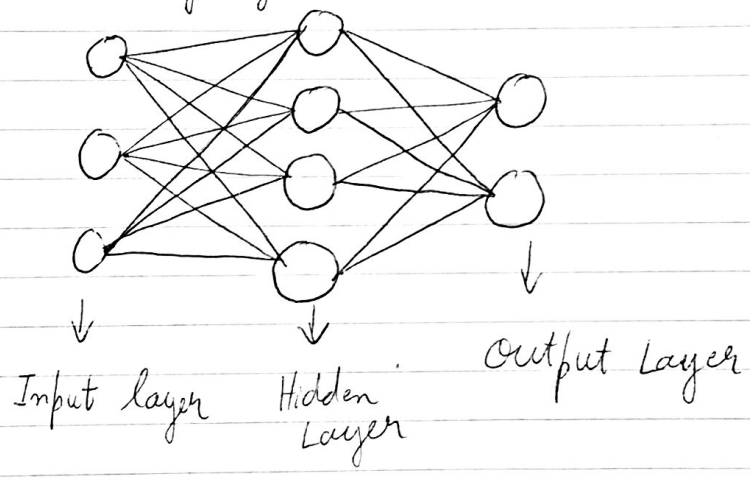
if learning rate is too high ↑

→ As much as 40% of network can be dead!, so it is important to set/tune learning rate.

ReLU is most widely used in these modern times.

☆ Neural Network Architectures

Layer-wise organization:



- Input Layer: This is the vector of inputs.
- Output Layer: The output vector of the network.
- Hidden Layers: This is the layer that contains Neurons and does the computation. When we say n-layer neural net, n is the no. of Hidden layers. These are called "Hidden" because they detect the "hidden" features not provided / visible in the training data.

★ Feed-Forward Computation:

A Neural Network computation is Repeated matrix multiplications interwoven with activation function. Example:

Inputs are 3×1 vectors (3 no.s). Outputs are k-vectors containing scores for each of the k-classes.

Input Layer: $in = \begin{bmatrix} 3 \\ -1 \\ 2 \end{bmatrix}$ (3×1)

Hidden Layer 1: $W1 = \begin{bmatrix} -1 & 2 & 9 \\ 0 & 1 & 6 \\ 11 & 21 & 4 \\ 6 & 69 & 83 \end{bmatrix}$ (4×3)

$h1 = f(W1 . in)$

$= \begin{bmatrix} * \\ * \\ * \\ * \end{bmatrix}$ (4×1)

Hidden Layer 2: $W2 = \begin{bmatrix} * & * & * & * \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$ (4×4)

$h_2 = f(W2 . h1)$

$= \begin{bmatrix} * \\ * \\ * \\ * \end{bmatrix}$ (4×1)

Output Layer: ~~out~~ $W_3 = \begin{bmatrix} * & \cdots & \cdots \\ * & \cdots & \cdots \\ * & \cdots & \cdots \\ * & \cdots & \cdots \end{bmatrix}$ $((\text{no. of outputs}) \times 4)$

W_3 is $(y \times 4)$ where y is desired outputs, we can use later.

$out = W_3 . h_2 = \begin{bmatrix} * \\ * \\ \vdots \\ \vdots \end{bmatrix}$ $(y \times 1)$

- $W_1, W_2, W_3, b_1, b_2, b_3$ are all learnable parameters.
- Input neurons = ~~Width~~ Rows in input vector.
Input vector could hold all (batch of training data, where each column is an input example.)

Representational Power:

- Neural Networks with fully connected layers can be seen as representing a family of functions parameterized by the weights.

★ Neural Nets can model/represent these functions because a Neural Net with at least one hidden layer is a Universal Approximator capable of approximating any real valued function, given we choose an appropriate non-linearity as the activation function. (See paper by G. Cybenko, 1989, for details.)

This is why Neural Nets can learn a huge class of functions and ~~approximate~~ map new, hidden features. (Keep in mind much of ML deals with approximating/mining the function or the process that generated the data. We learn this function and use it for predicting new information.)

★ About Layers, Neurons and Capacity:

- No. of layers = Depth No. of Neurons = Width
- ~~Although~~ more layers does not mean better representative power. One hidden layer - necessary, but more is not better (in general) except in case of Convolutional Nets.
- Layers + Neurons = Capacity. High Capacity = Overfitting, learn noise in the data. Low Capacity = Underfitting.

- However, to minimize Overfitting, use Regularization, do not reduce capacity. It is easier to train larger networks. Larger networks contain many more local minima and is easier for loss function to converge to.

★ Regularization:

1. L2 Regularization: $Loss_{penalized} = Loss + \frac{1}{2}\lambda w^2$, for every weight in the network. Like before, the squared-magnitude penalizes large weights. The $\frac{1}{2}$ is there because when we take it's differential in gradient, it ends up as λw.
   (λ - regularization strength - hyperparameter)

2. Dropout: While training, keep a neuron alive with probability p, or else set it's output to 0.

```
H1 = f(W1.X + b)
if (p = randomNumber()) < 0.5:
        H1 = 0
H2 = ...
... ...
```

Note: In L2 reg, we add the regularization gradient to the gradient w.r.t to the weights as well.

☆ How To Train Your Neural Network:

(Coming to ~~Movies~~ a Cinema near you, Winter 2017.)

1. ~~Eat~~ Initialize weights, hidden layers using these weights, hyper parameters.
2. Gradient Descent start. Loop:
3. Evaluate class scores by performing feed-forward computation, as shown previously.
4. Compute class probabilities using softmax over scores (optional, but better)
5. Compute Loss, using loss function + Regularization.
6. Compute gradient on scores. For example:

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \quad \text{[Cross-Entropy Loss]}$$

$$\frac{dL_i}{df_k} = \frac{e^{f_k}}{\sum_j e^{f_j}} - \mathbb{1}(y_i = k) \quad \text{[-1 for all ~~scores~~ probs of correct class]}$$

$\frac{e^{f_k}}{\sum_j e^{f_j}} \rightarrow p_k$

Python:

```
probs = [p_{y_1}, p_{y_2}, ... p_{y_k}]
d_scores = probs
d_scores[range(0, num_examples), y] -= 1
```

7. Backpropagate gradient to weights, biases (if any)
8. Add regularization to gradient $\boxed{dW_i = dW_i + \lambda W_i}$
9. Update parameters $\boxed{\begin{aligned} W_i &= W_i + dW_i \\ b_i &= b_i + db_i \end{aligned}}$