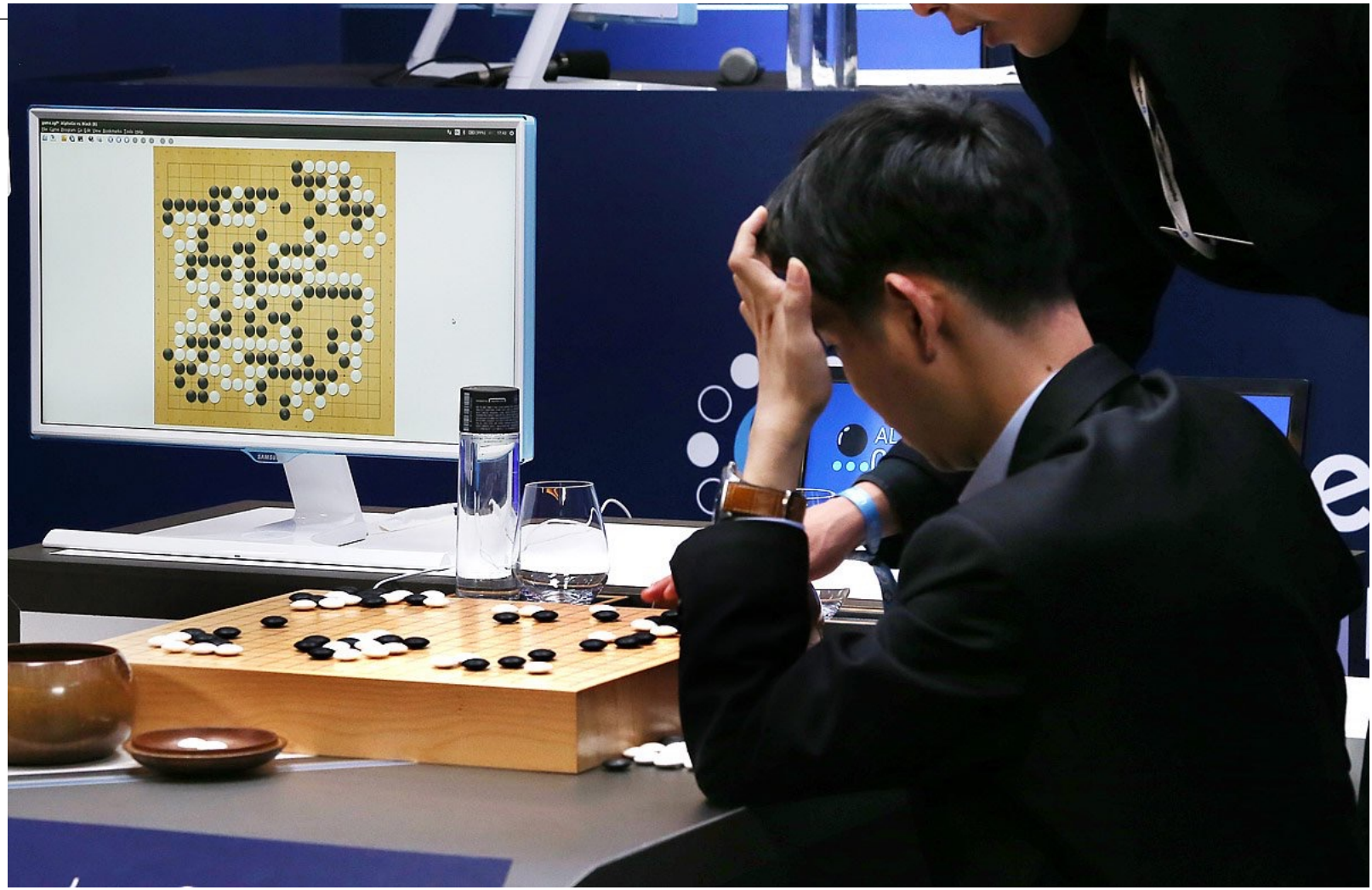
Introduction to

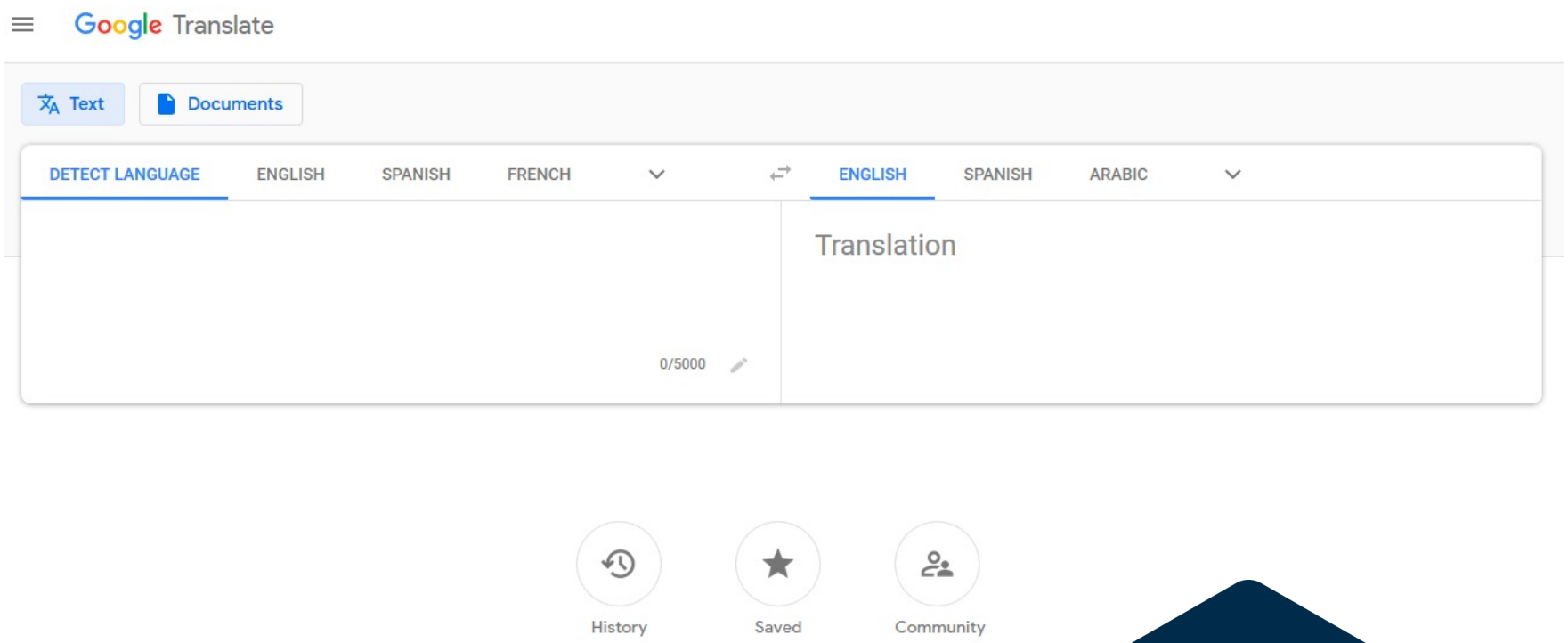
Deep Neural Networks

Grand Achievements of DL

Grand Achievements of DL

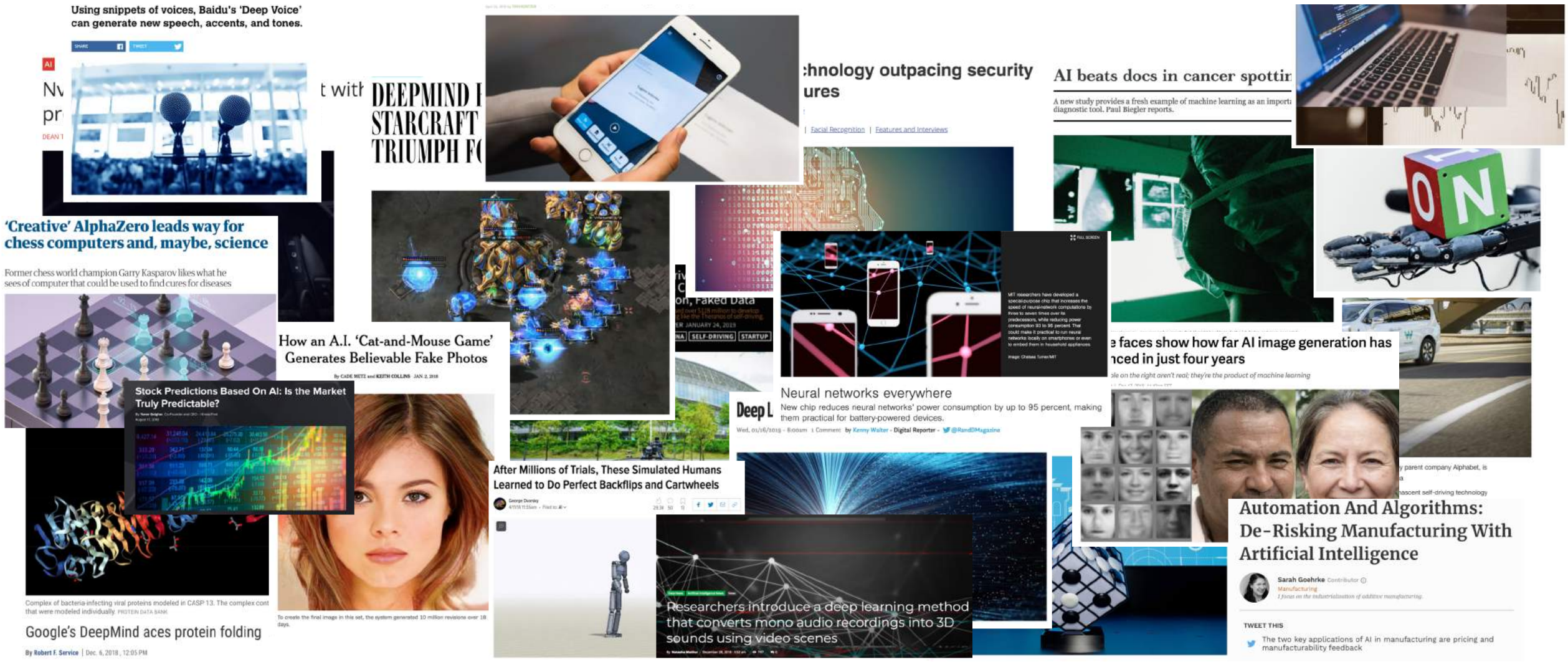


Grand Achievements of DL



DeepL

The Rise of Deep Learning



Why now ?

Not an old idea, but the time has come now ….

1952 Stochastic Gradient Descent

1958 Perceptron
- Learnable Weights

1986 Backpropagation
- Multi-Layer Perceptron

1995 Deep Convolutional NN
- Digit Recognition

1. Big Data
- Larger Datasets
- Easier Collection & Storage

IMAGENET

WIKIPEDIA
The Free Encyclopedia

2. Hardware
- Graphics Processing Units (GPUs)
- Massively Parallelizable

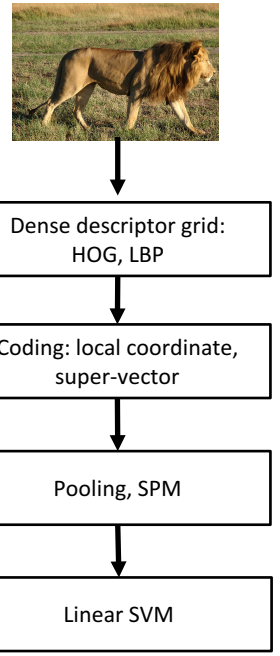
3. Software
- Improved Techniques
- New Models
- Toolboxes

TensorFlow

Classical Machine Learning Pipeline

- Input Data → Engineer Features → Build Model
  - Needs domain knowledge
  - Time consuming
  - Good Features make good models
- Papers in ML, Vision, NLP were focused
  - A new feature was a new finding (SIFT, Syntax parsers, Named entity recognizers)
- What if one could automatically learn features from the input data ?

NEC-UIUC



[Lin CVPR 2011]

Example of Classical ML pipeline

Domain knowledge → Define features → Detect features to classify

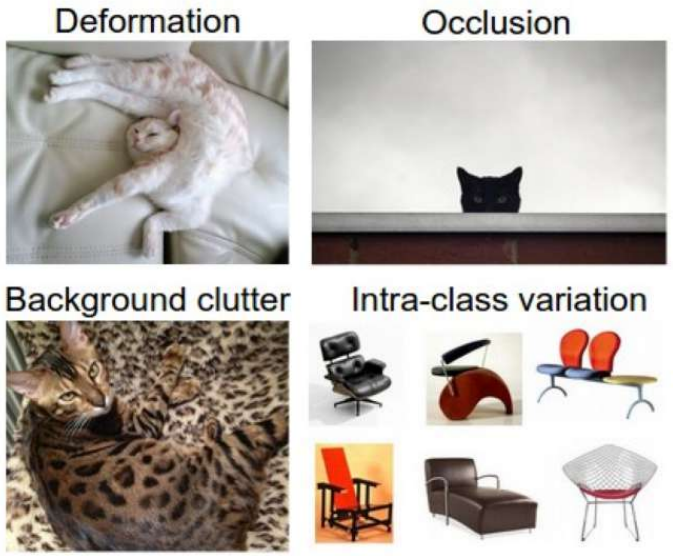
NEC-UIUC

Dense descriptor grid: HOG, LBP

Coding: local coordinate, super-vector

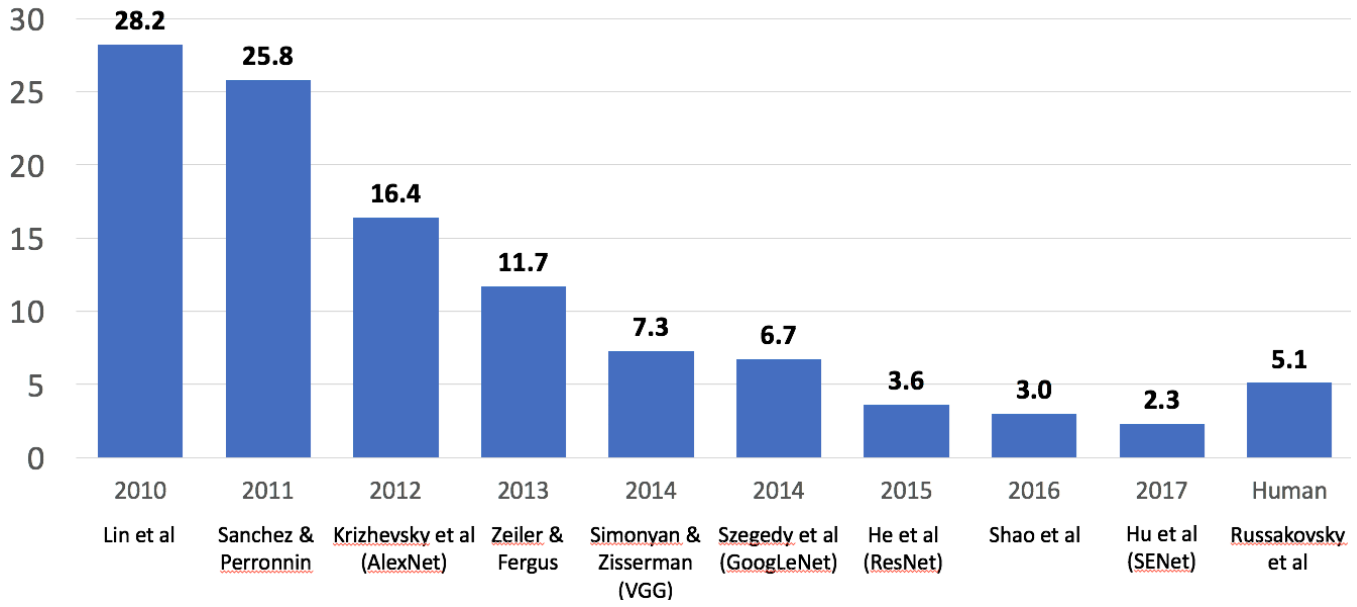
Pooling, SPM

Linear SVM

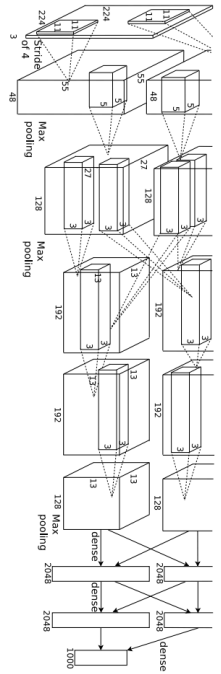


Deep Learning as Representation Learning

- New pipeline advertised mostly by DL:
  - input data -> learn representation -> Build Models
  - learn representation, and classification are part of ML now



SuperVision



[Krizhevsky NIPS 2012]

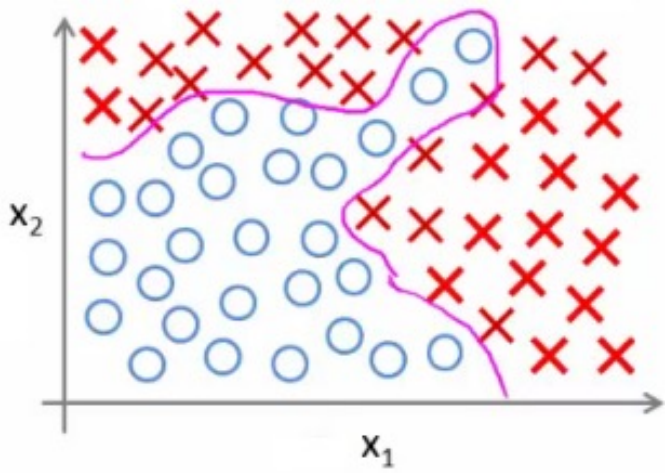
DL as Representation Learning

- A lot of time spent on creating new architectures
  - still better performance than engineering features
  - there is still hope that with progress we will need less and less engineering of the architectures, and this will be automated too

- Great progress when feature engineering is hard or unclear
  - Multimodal representation -- how to model language and visual signals at once?

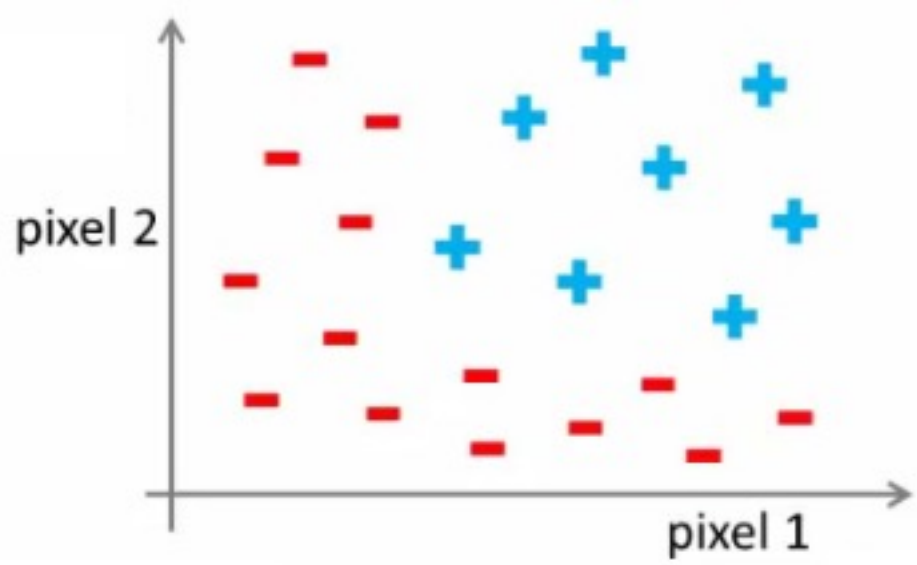
Why Neural Networks? Intuition

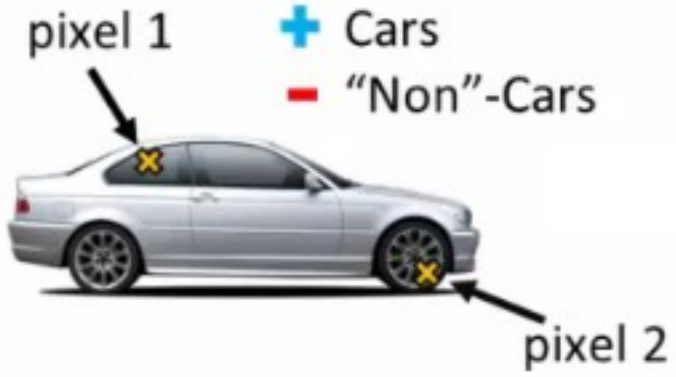
- Given supervised classification task with non linear decision boundary
  - With 2 features what can we do?
  - With 100 features?
    - E.g, housing price prediction
    - You can compute a lot of higher order features
    - (x_1^2 , x_1x_2, x_1x_3 ..., x_1x_{100})
    - 5000 features
    - Number of features grows $O(n^2)$
    - This would be computationally expensive to work with as a feature set (how to solve this problem?)



Example: Problems with large features

- Computer vision sees a matrix of pixel intensity values
- To build a classifier to detect car, look at pixel values at some locations in the image

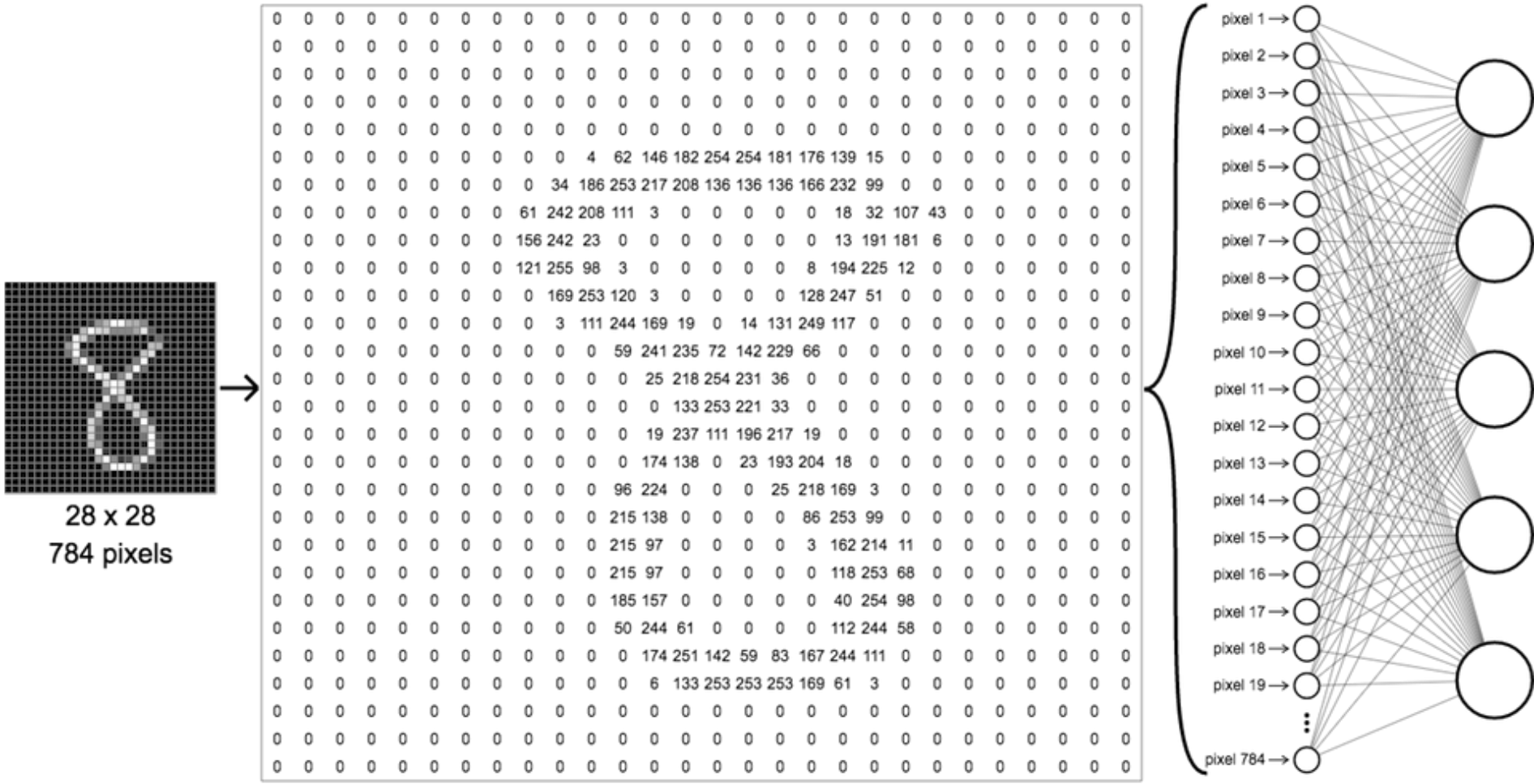




Can we use logistic regression?

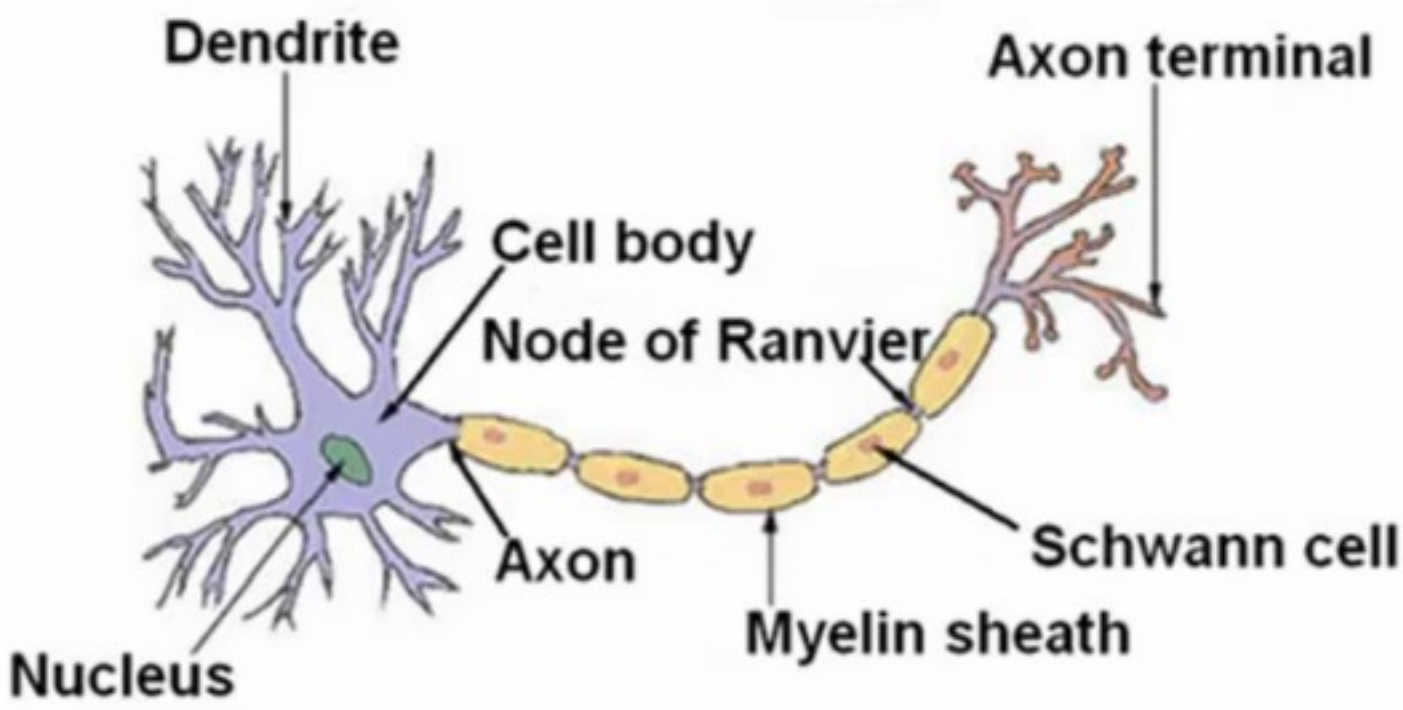
- Logistic regression with polynomial hypothesis is way too expensive. Why?
- Feature space:
  - If we use 50 x 50 pixels => 2500 pixels n = 2500, n^2=?
  - If 100 x 100 RB then => $n^2/2$ = 50 000 000 features
  - Way too many input features to optimize

Linearizing the input



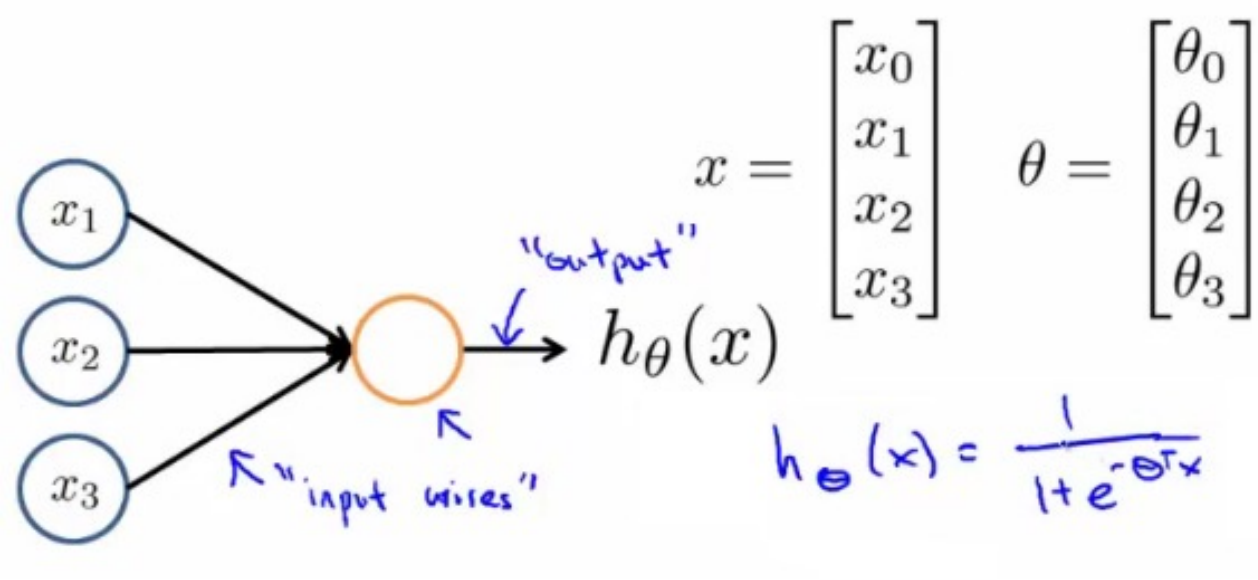
Neural Networks and Brain

- There are 100 billion individual neurons in the human brain.



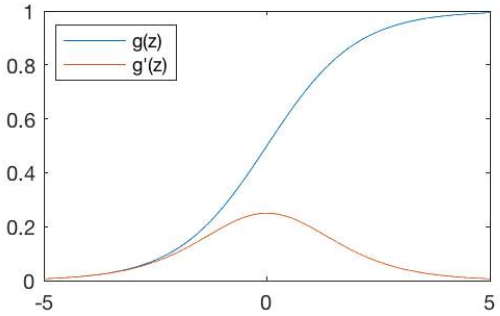
Artificial Neuron

- Feed input via input nodes
- Logistic unit does computation
- Sends output



Activation Functions

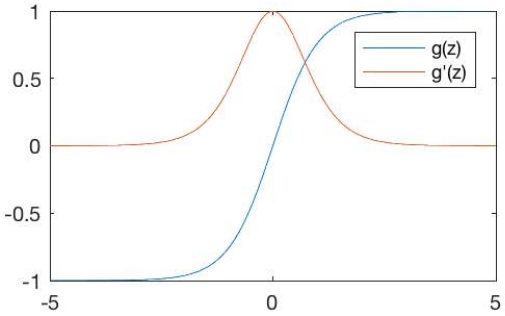
Sigmoid Function



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

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

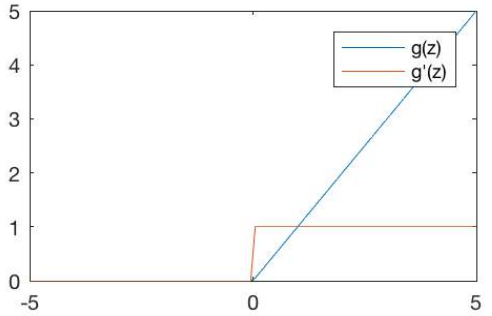
Hyperbolic Tangent



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

$$g'(z) = 1 - g(z)^2$$

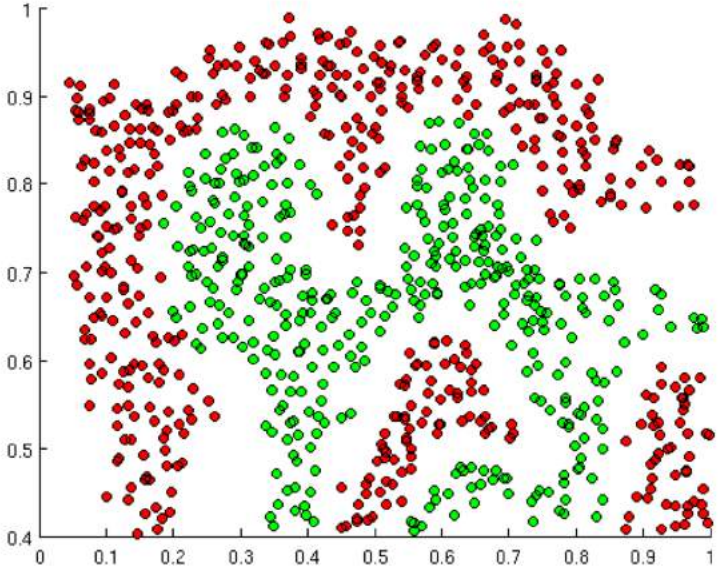
Rectified Linear Unit (ReLU)



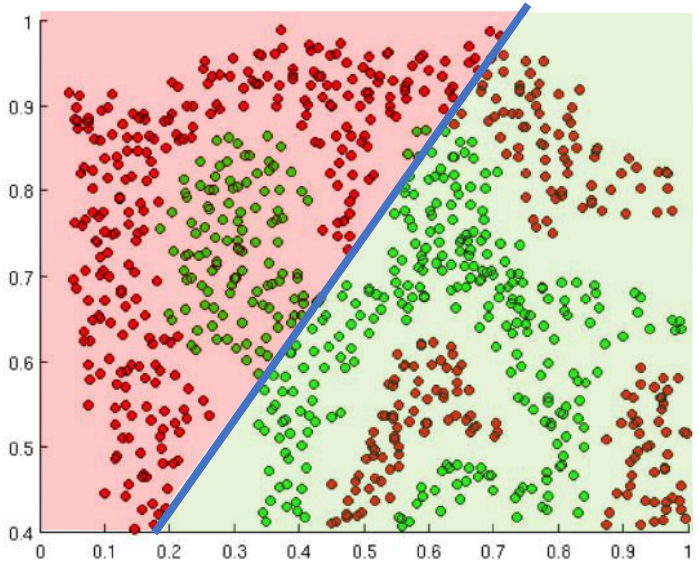
$$g(z) = \max(0, z)$$

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

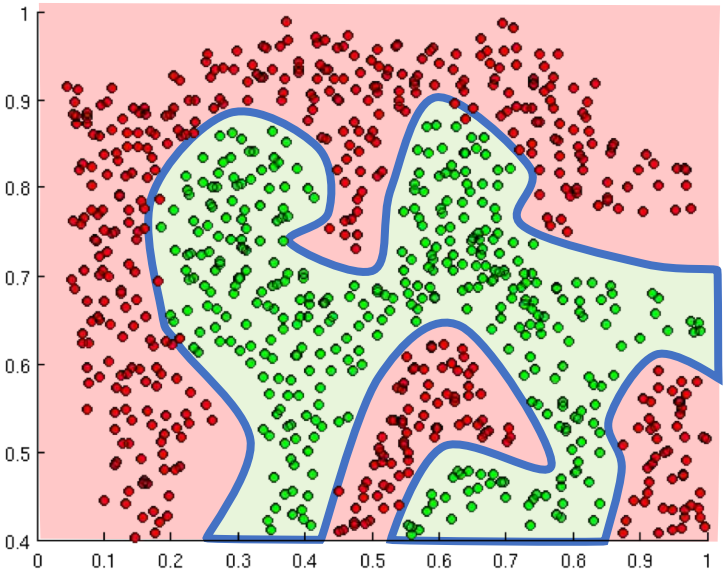
Why Activations?



If you want to separate green and red points



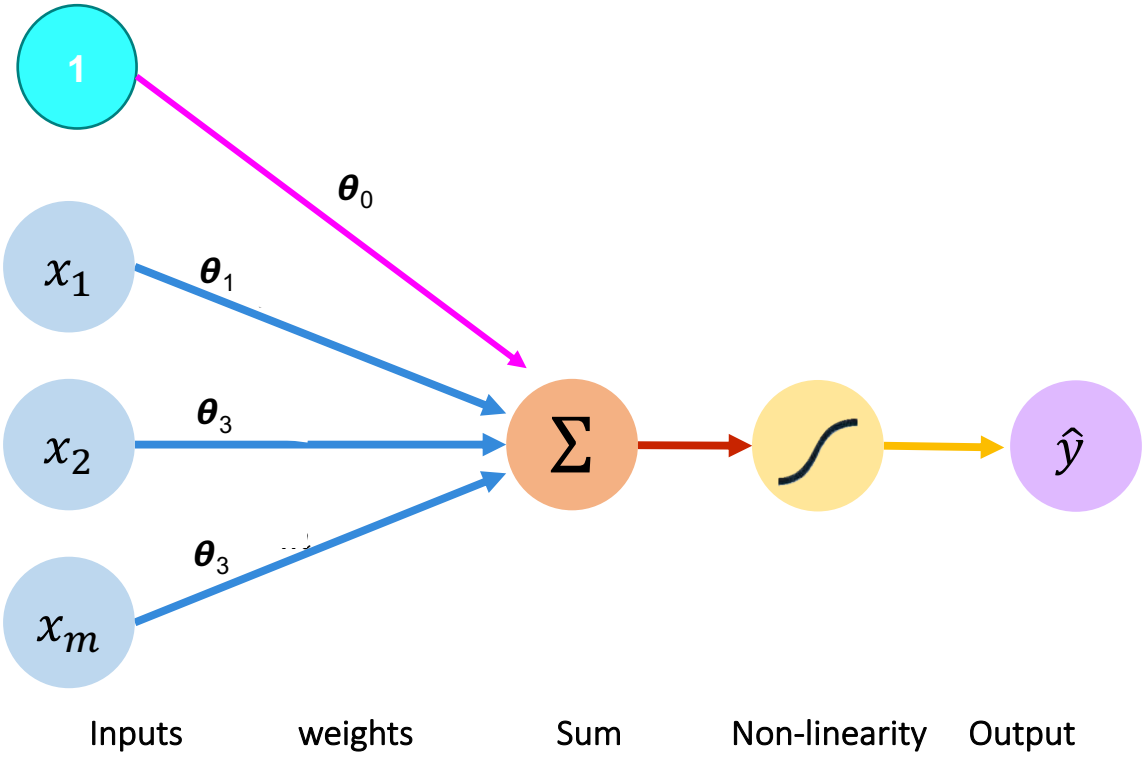
Linear activations produce linear decisions irrespective of the network size



Non linear activations approximate complex decision boundaries

Activations are key to introducing non-linearity into the representation space

Perceptron



Output — Linear combination of inputs

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

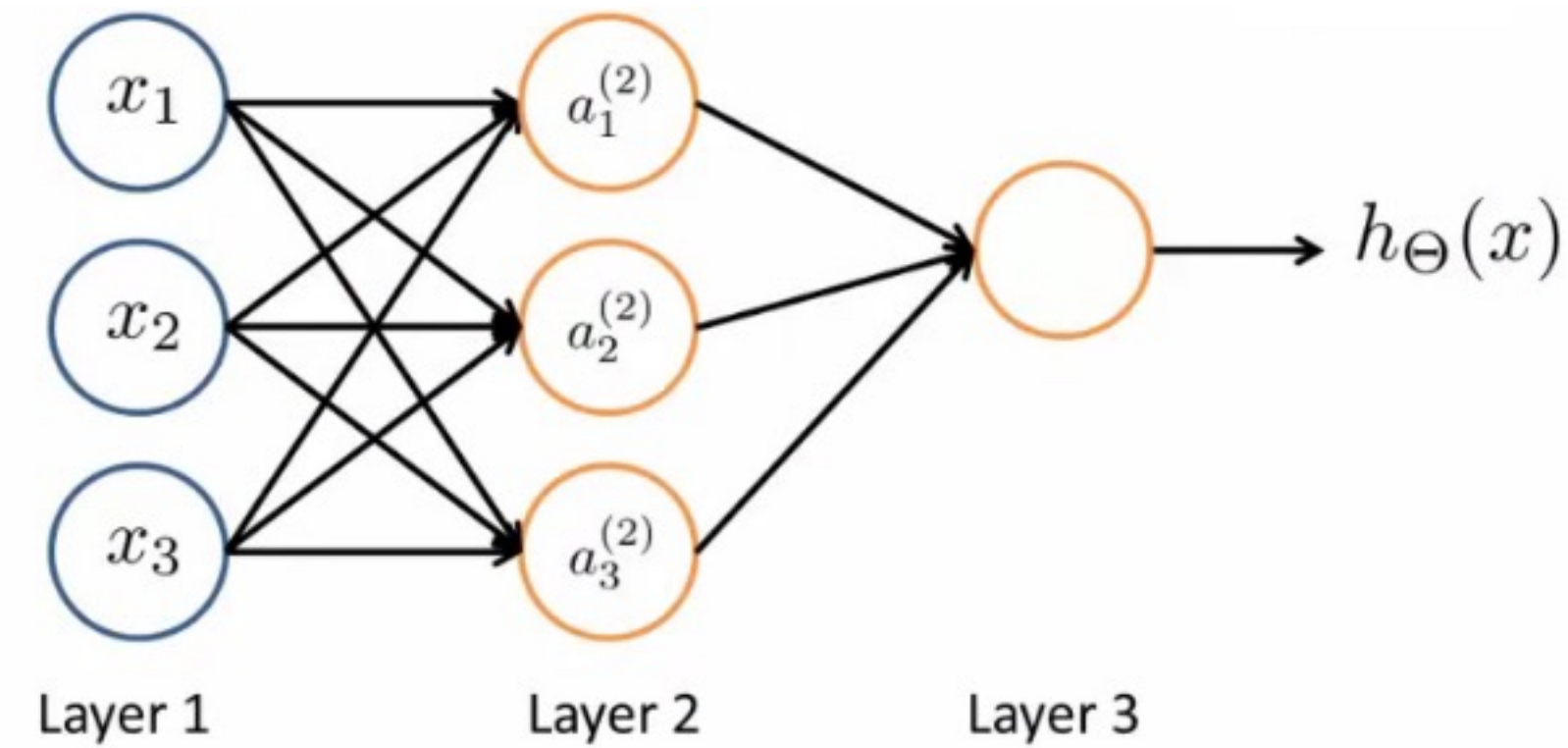
Non-linear activation function

Output — Linear combination of inputs

$$\hat{y} = g\left(\theta_0 + \sum_{i=1}^{m} x_i\ \theta_i\right)$$

Non-linear activation function — Bias

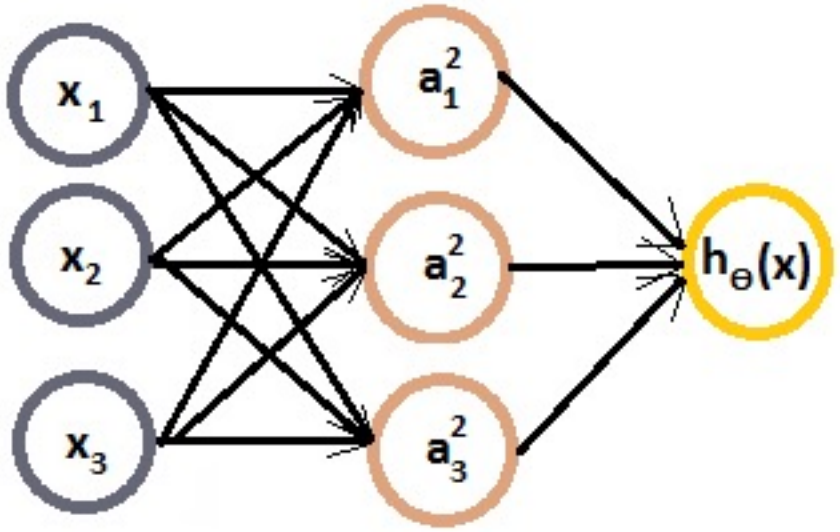
Perceptron



Neural Networks - Notations

- **$a_i^{(j)}$ - activation of unit *i* in layer *j***
  - So, a_1^2 - is the **activation** of the 1st unit in the second layer
  - By activation, we mean the value which is computed and output by that node
- **$\Theta^{(j)}$ - matrix of parameters**
  - Parameters for controlling **activation** from one layer to the next
  - If s_j units are in layer *j* and s_{j+1} units in layer j + 1
  - Then Θ^j will be of dimensions $[s_{j+1} \times s_j + 1]$
- $g(\Theta^T x)$ is the activation function

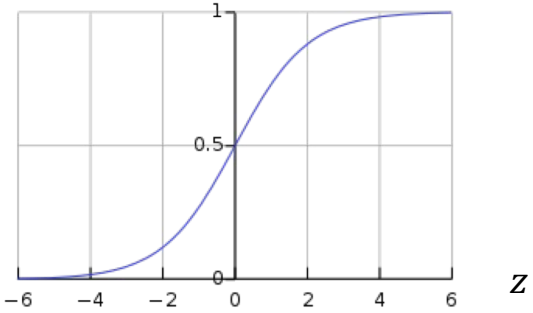
Neural Network Computations



$$\hat{y} = g\left(\theta_0 + \sum_{i=1}^{m} x_i \theta_i\right)$$

- Example: sigmoid function

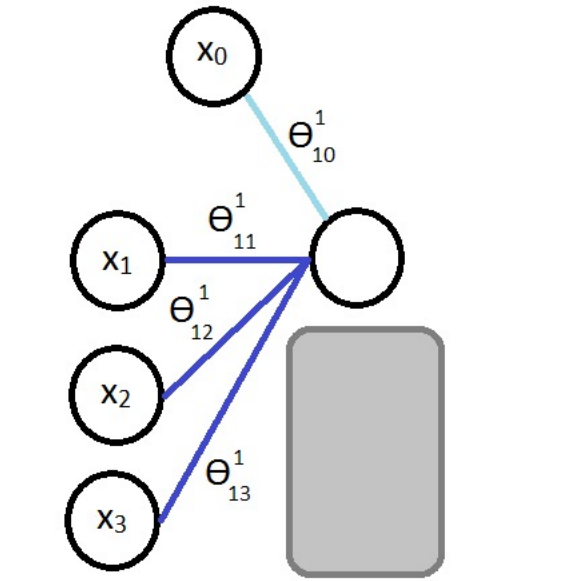
$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$



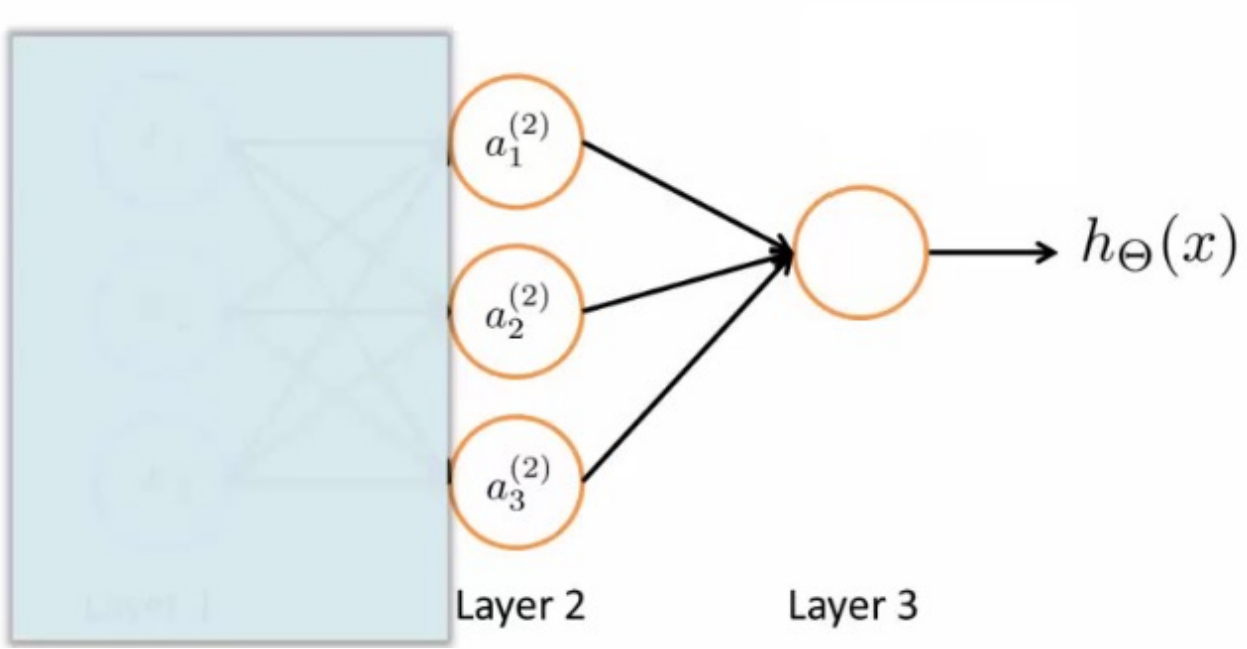
$$a_1^{(2)} = g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3)$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3)$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3)$$
$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$$

Parameter Θ Details

- Θ_{ji}^{ℓ}
  - j (first of two subscript numbers)= ranges from 1 to the number of units in layer $\ell+1$
  - i (second of two subscript numbers) = ranges from 0 to the number of units in layer ℓ
  - ℓ is the layer you're moving FROM
- Additional terms
  - $z_1^2 = \Theta_{10}^1 x_0 + \Theta_{11}^1 x_1 + \Theta_{12}^1 x_2 + \Theta_{13}^1$
  - $a_1^2 = g(z_1^2)$

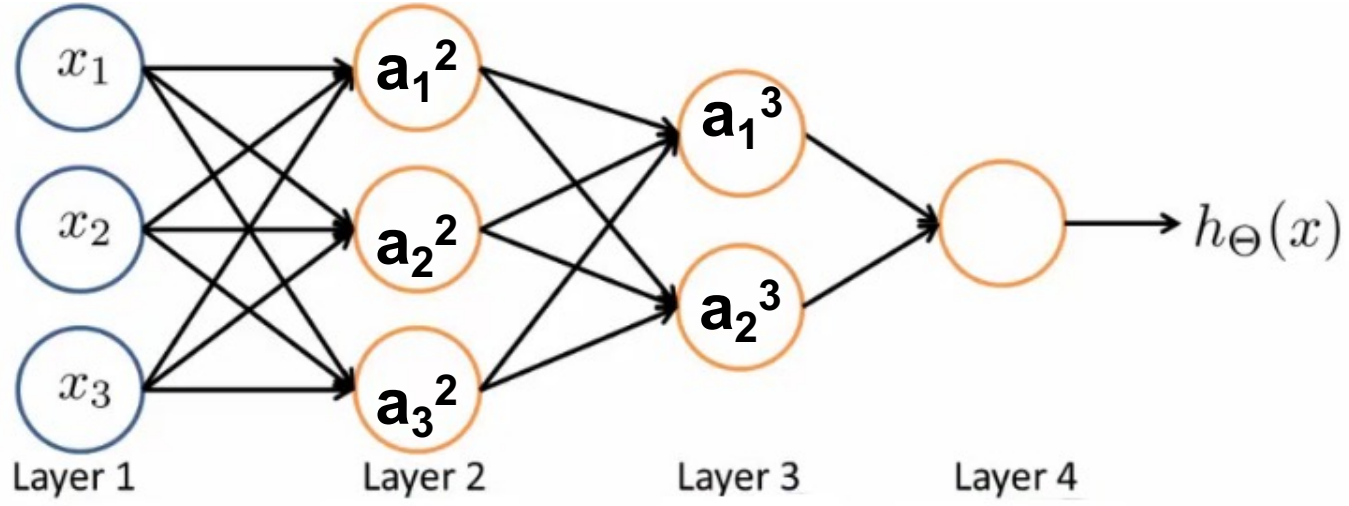


How do neural networks solve large feature problem?



- Output layer (Layer 3) is a logistic regression node
- $h_\Theta(x) = g(\Theta_{10}^2 a_0^2 + \Theta_{11}^2 a_1^2 + \Theta_{12}^2 a_2^2 + \Theta_{13}^2 a_3^2)$

How do neural networks solve large feature problem?



Multiple layers and multiple nodes per layer produce interesting non-linearity

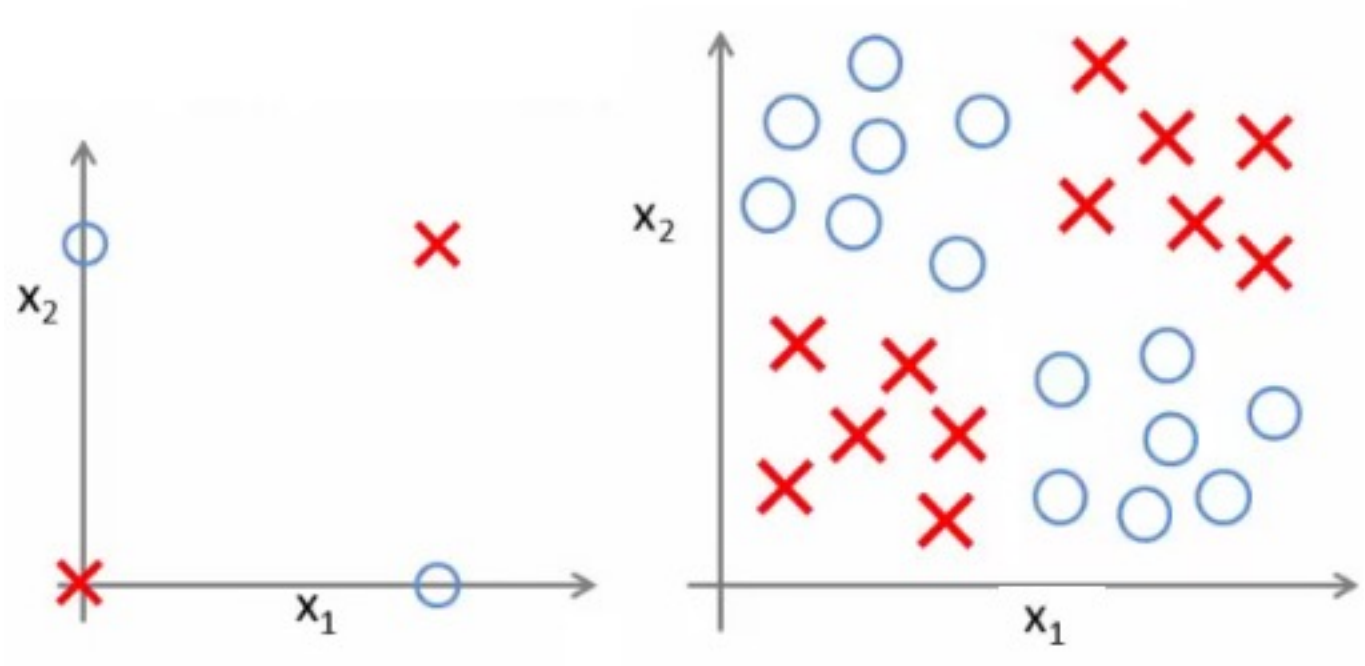
- $a_1^2 = g(\Theta_{10}^1 x_0 + \Theta_{11}^1 x_1 + \Theta_{12}^1 x_2 + \Theta_{13}^1 x_3)$
- $a_1^3 = g(\Theta_{10}^2 a_0^2 + \Theta_{11}^2 a_1^2 + \Theta_{12}^2 a_2^2 + \Theta_{13}^2 a_3^2)$

How does it look?

$$\frac{1}{1+e^{\left(\theta_{11}^{2}\frac{1}{1+e^{-\left(\theta_{10}^{1}x_0+\theta_{11}^{1}x_0+\theta_{12}^{1}x_0+\theta_{13}^{1}x_0\right)}}+\theta_{12}^{2}\frac{1}{1+e^{-\left(\theta_{20}^{1}x_0+\theta_{21}^{1}x_0+\theta_{22}^{1}x_0+\theta_{23}^{1}x_0\right)}}+\theta_{13}^{2}\frac{1}{1+e^{-\left(\theta_{30}^{1}x_0+\theta_{31}^{1}x_0+\theta_{32}^{1}x_0+\theta_{33}^{1}x_0\right)}}\right)}}$$

Neural Network Simple Example

- Non-linear classification: XOR/XNOR
  - x_1, x_2 are binary

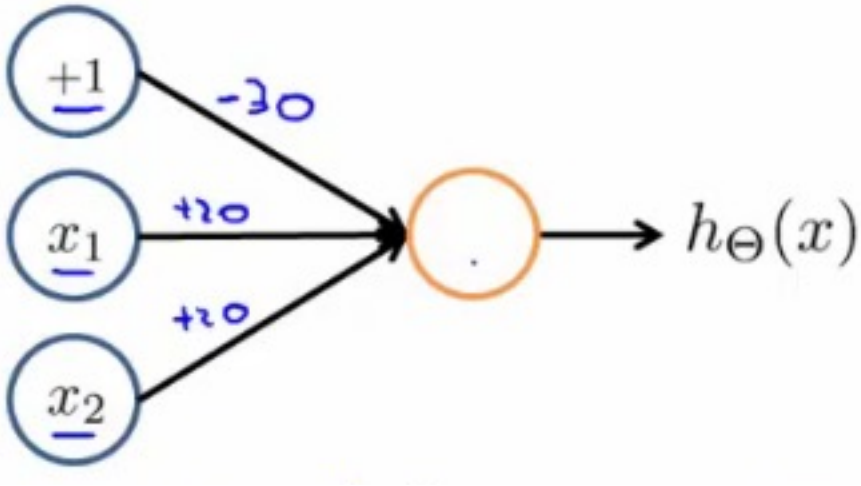


XOR/XNOR

INPUT		OUTPUT
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Input		Output
A	**B**	**A XNOR B**
0	0	1
0	1	0
1	0	0
1	1	1

AND Function

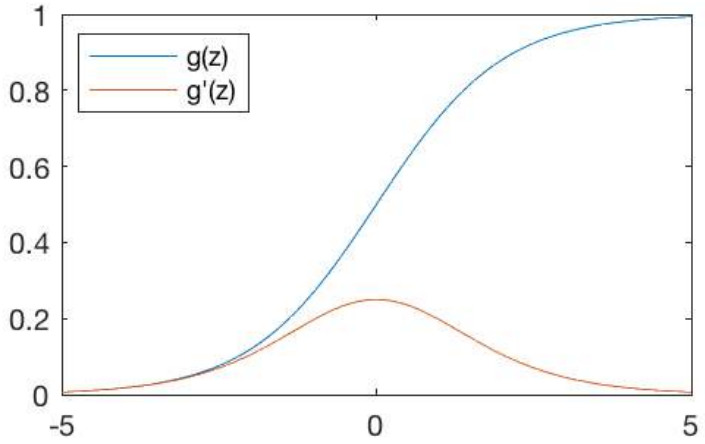


$$h_\theta(x) = g(-30 + 20x_1 + 20x_2)$$

AND Function Example

$$h_\theta(x) = g(-30 + 20x_1 + 20x_2)$$

Sigmoid Function



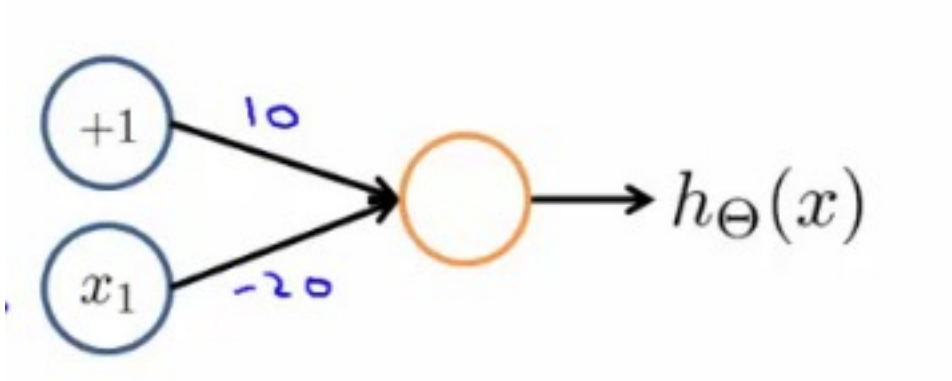
$$g(z) = \frac{1}{1 + e^{-z}}$$

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

x_1	x_2	$h_\theta(x)$	y
0	0	g(-30)	9.3576229688392989 5384E-14
1	0	g(-10)	4.5397868702434394 50478E-5
0	1	g(-10)	4.5397868702434394 50478E-5
1	1	g(10)	0.9999546021312975 656055

NOT function Example

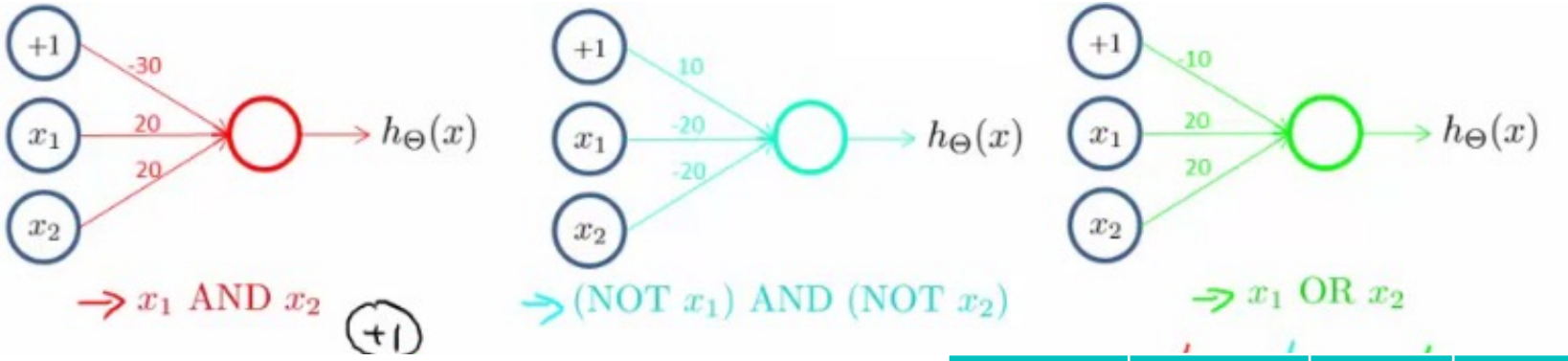
$$h_\theta(x) = g(10 - 20x_1)$$

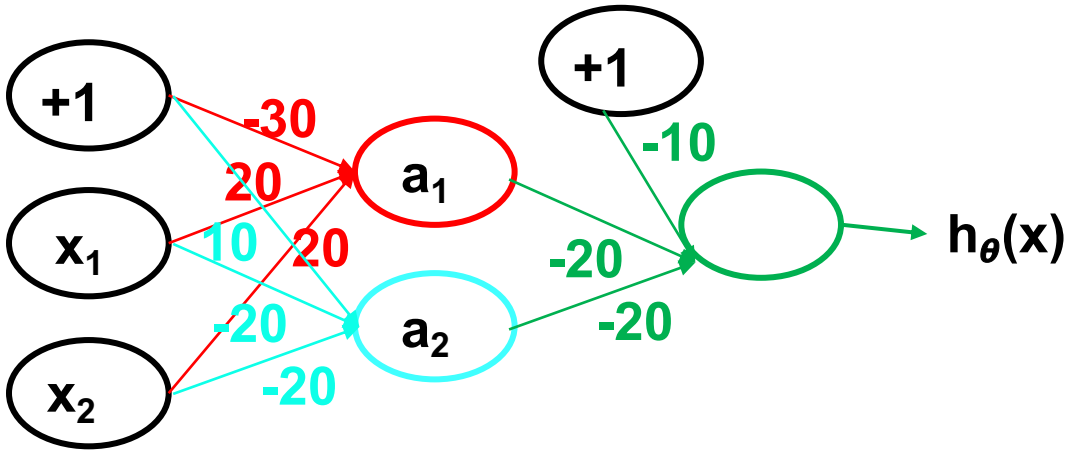


x_1	$h_\theta(x)$	y
0	g(10)	0.9999546021312975656055
1	g(-10)	4.539786870243439450478E-5

XNOR = NOT XOR

- (X1 AND X2) OR ((NOT X1) AND (NOT X2))



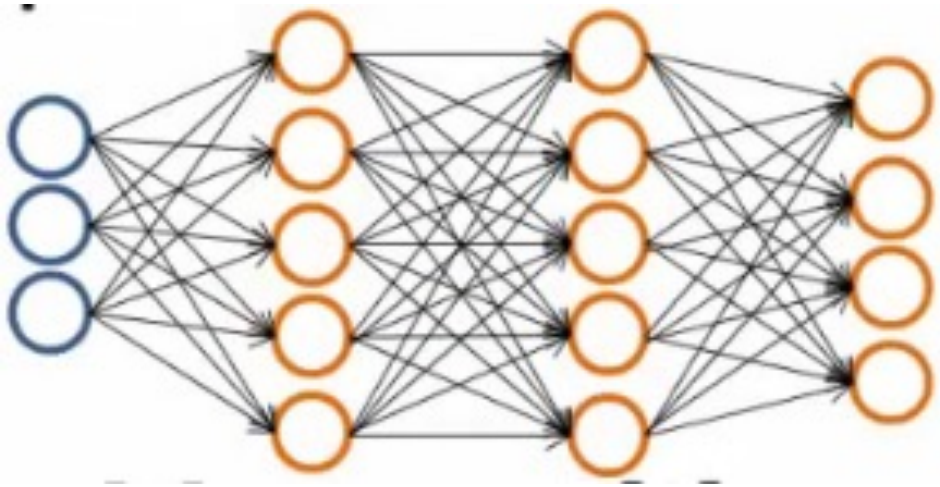


x_1	x_2	a_1	a_2	$h_\theta(x)$
0	0	0	1	1
1	0	0	0	0
0	1	0	0	0
1	1	1	0	1

Multiclass classification

- With handwritten digital recognition problem - 10 possible categories (0-9)
  - How do you do that?
  - Done using an extension of one vs. all classification
- Recognizing pedestrian, car, motorbike or truck
  - Build a neural network with four output units
  - Output a vector of four numbers
    - 1 is 0/1 pedestrian, 2 is 0/1 car ,3 is 0/1 motorcycle, 4 is 0/1 truck
  - 1-hot encoding When image is a pedestrian get [1,0,0,0]

Multiclass Neural Network



$$h_\Theta(x) \in \mathbb{R}^4$$

- Just like one vs. all described earlier
  - Here we have four logistic regression classifiers

$$y^{(i)} \text{ one of } \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

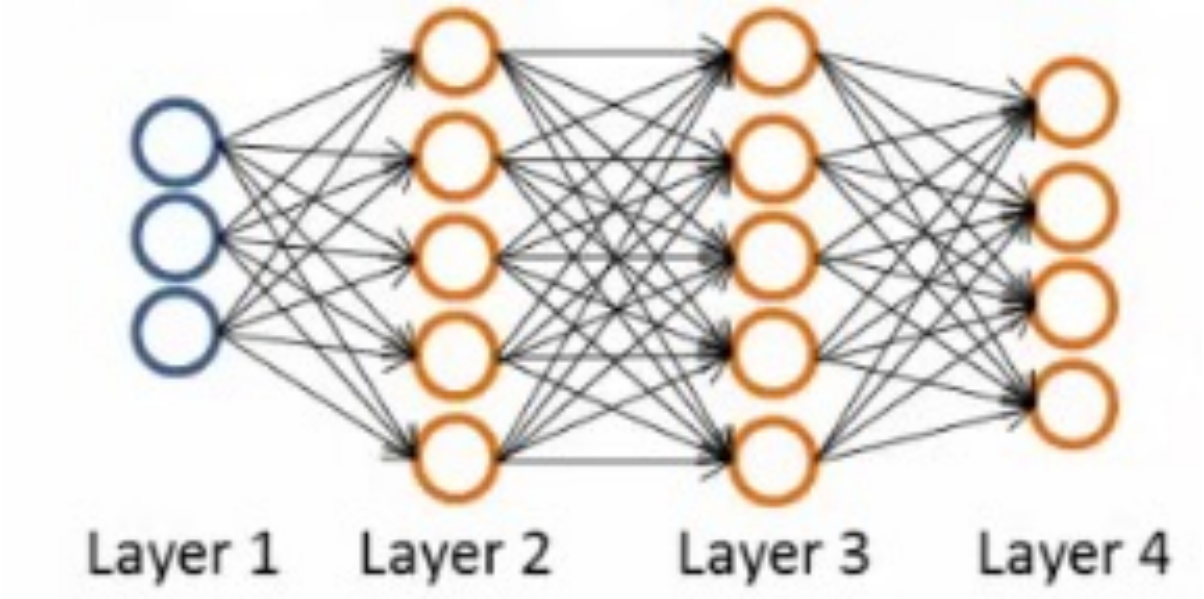
- k distinct classifications
- Typically k is greater than or equal to three
- If only two just go for binary
- $s_L = k$
- So y is a k-dimensional vector of real numbers

$y \in \mathbb{R}^K$ E.g. $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

pedestrian car motorcycle truck

Neural network cost function

- Training set is $\{(x^1, y^1), (x^2, y^2), (x^3, y^3) \ldots (x^m, y^m)$
- L = number of layers in the network
- s_l = number of units in layer l



Cost function for neural networks

- The (regularized) logistic regression cost function is as follows;

$$J(\theta) = -\frac{1}{m}\sum_{t=1}^{m}\sum_{k=1}^{K}\left[y_k^{(t)}\ \log(h_\theta(x^{(t)}))_k + (1-y_k^{(t)})\ \log(1-h_\theta(x^{(t)})_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(\theta_{j,i}^{(l)})^2$$

- neural network cost function is a generalization
  - instead of one output we generate *k* outputs (for K classes)
-

$$h_\Theta(x) \in \mathbb{R}^K \quad (h_\Theta(x))_i = i^{th}\ \text{output}$$

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)}\log(h_\Theta(x^{(i)}))_k + (1-y_k^{(i)})\log(1-(h_\Theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

Cost Function Details

- **First half**

$$-\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k)\right]$$

  - For each training data example (i.e. 1 to m - the first summation)
    - Sum for each position in the output vector
    - This is an average sum of logistic regression
- **Second half**
  - This is a triple nested regularization term

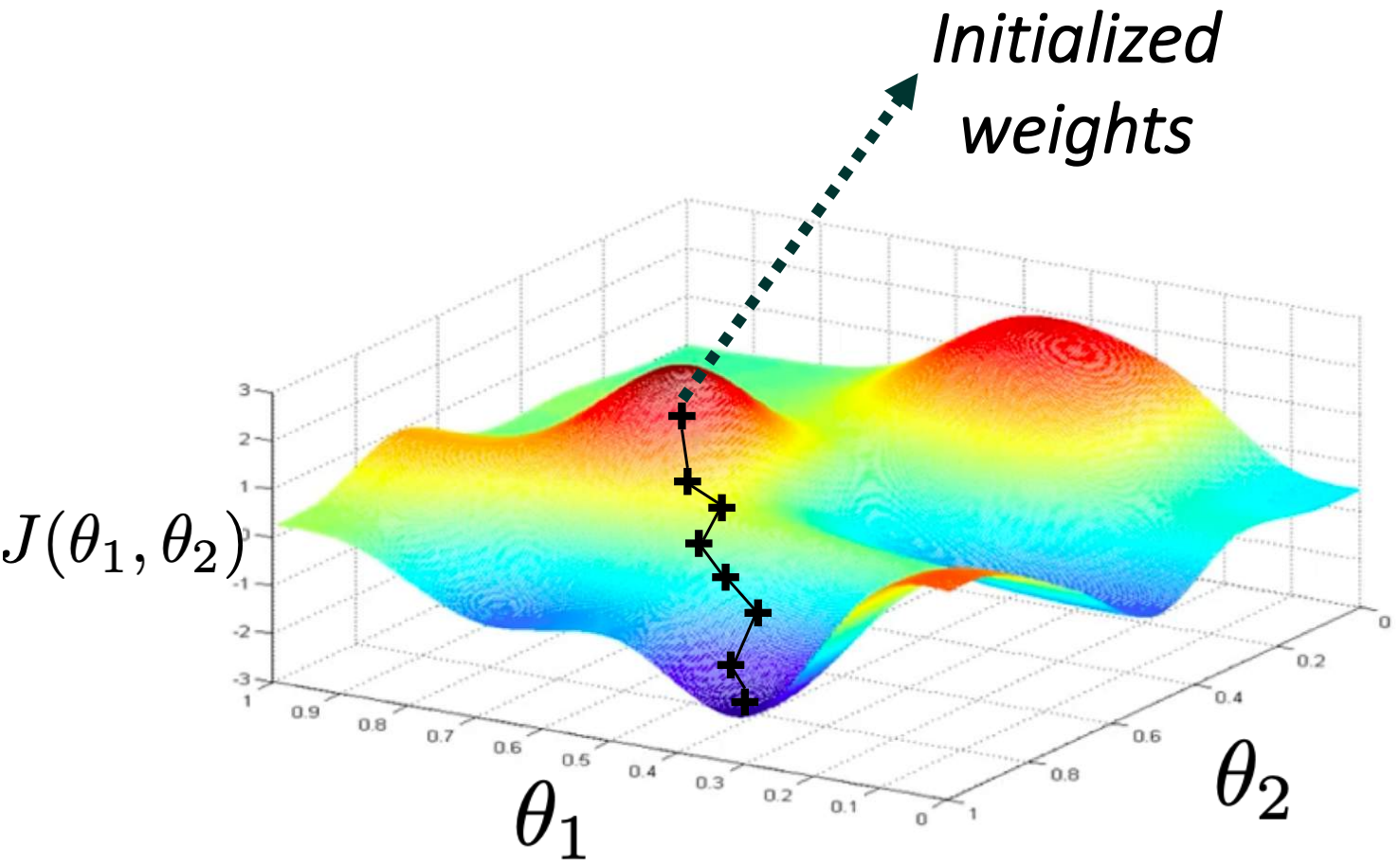
$$\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

How to minimize the cost function?

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}} (\Theta_{ji}^{(l)})^2$$

- We want to find parameters Θ which minimize J(Θ)
- To do so we can use Gradient descent or any Advanced optimization algorithms
- Partial derivate terms
  - Each layer had a Θ matrix associated with it
  - We want to calculate the partial derivative Θ with respect to a single parameter $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$

Gradient Descent

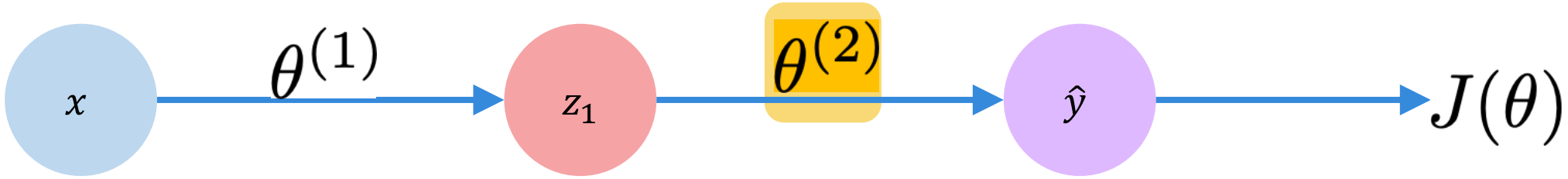


Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\theta)}{\partial \theta}$
4. Update weights, $\theta = \theta - \alpha \frac{\partial J(\theta)}{\partial \theta}$
5. Return weights

Gradient descent steps

How much does a small change in weights affect the empirical loss ?



Algorithm

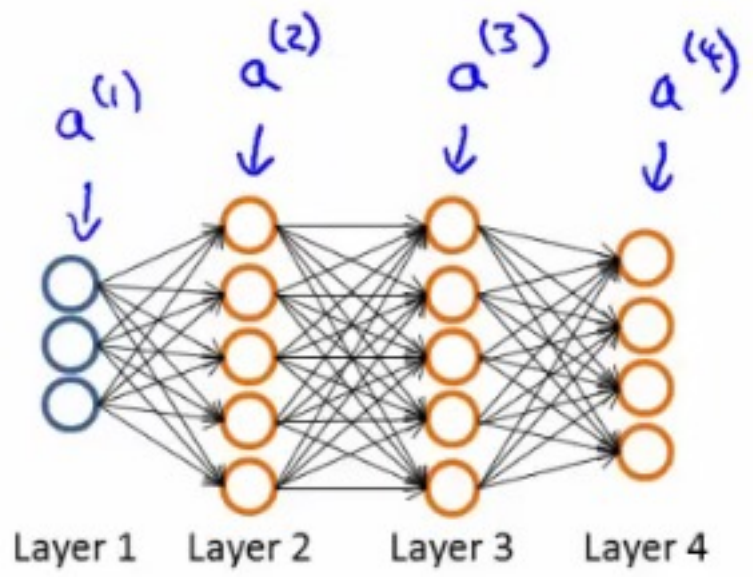
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\theta)}{\partial \theta}$
4. Update weights, $\theta = \theta - \alpha \frac{\partial J(\theta)}{\partial \theta}$
5. Return weights

$$\frac{\partial J(\theta)}{\partial \theta^{(2)}} = \frac{\partial J(\theta)}{\partial y} * \frac{\partial z_1}{\partial \theta^{(2)}}$$

$$\frac{\partial J(\theta)}{\partial \theta^{(1)}} = \frac{\partial J(\theta)}{\partial y} * \frac{\partial y}{\partial z_1} * \frac{\partial z_1}{\partial \theta^{(1)}}$$

Gradient Computation in Practice

- Assuming we have single input data instance x



$$
\begin{aligned}
a^{(1)} &= x \\
z^{(2)} &= \Theta^{(1)} a^{(1)} \\
a^{(2)} &= g(z^{(2)}) \quad (\text{add } a_0^{(2)}) \\
z^{(3)} &= \Theta^{(2)} a^{(2)} \\
a^{(3)} &= g(z^{(3)}) \quad (\text{add } a_0^{(3)}) \\
z^{(4)} &= \Theta^{(3)} a^{(3)} \\
a^{(4)} &= h_\Theta(x) = g(z^{(4)})
\end{aligned}
$$

Backpropagation

- Use it to compute the partial derivatives
- For each node we can calculate (δ_j^l) - this is **the error of node j in layer l**
- We do this from the outermost layer to inner most layer (layer L to layer 1)
- $\delta_j^4 = a_j^4 - y_j$
  - Essentially how close is the hypothesis to the desired output a_j^4 is actually $h_\Theta(x)_j$

Backpropagation

- Compute $\delta^4 = a^4 - y$, then compute

$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$$
$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$$

- Θ^3 is the vector of parameters for the layer 3
- $g'(z^3)$ is the first derivative of the activation function g
- $g'(z^3) = a^3 .* (1 - a^3)$ (deriviative of sigmoid)
- **$\delta^3 = (\Theta^3)^T \delta^4 .*(a^3 .* (1 - a^3))$**
- **$\delta^2 = (\Theta^2)^T \delta^3 .*(a^2 .* (1 - a^2))$**

Backpropagation Steps

- Given Training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$
- We will iteratively update δ values for each layer
- **First**, set the delta values Set $\triangle_{ij}^{(l)} = 0$ (for all l, i, j)
  - Δ is placeholder/accumator for δ values
- **Next**, loop through the training set For $i = 1$ to m
  - calculate δ^L where $\delta^L = a^L - y^i$ (error of output layer)
  - using **back propagation** we move back through the network from layer L-1 down to layer 1
  - Finally, use Δ to accumulate the partial derivative terms $\triangle_{ij}^{(l)} := \triangle_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

Backpropagation Steps

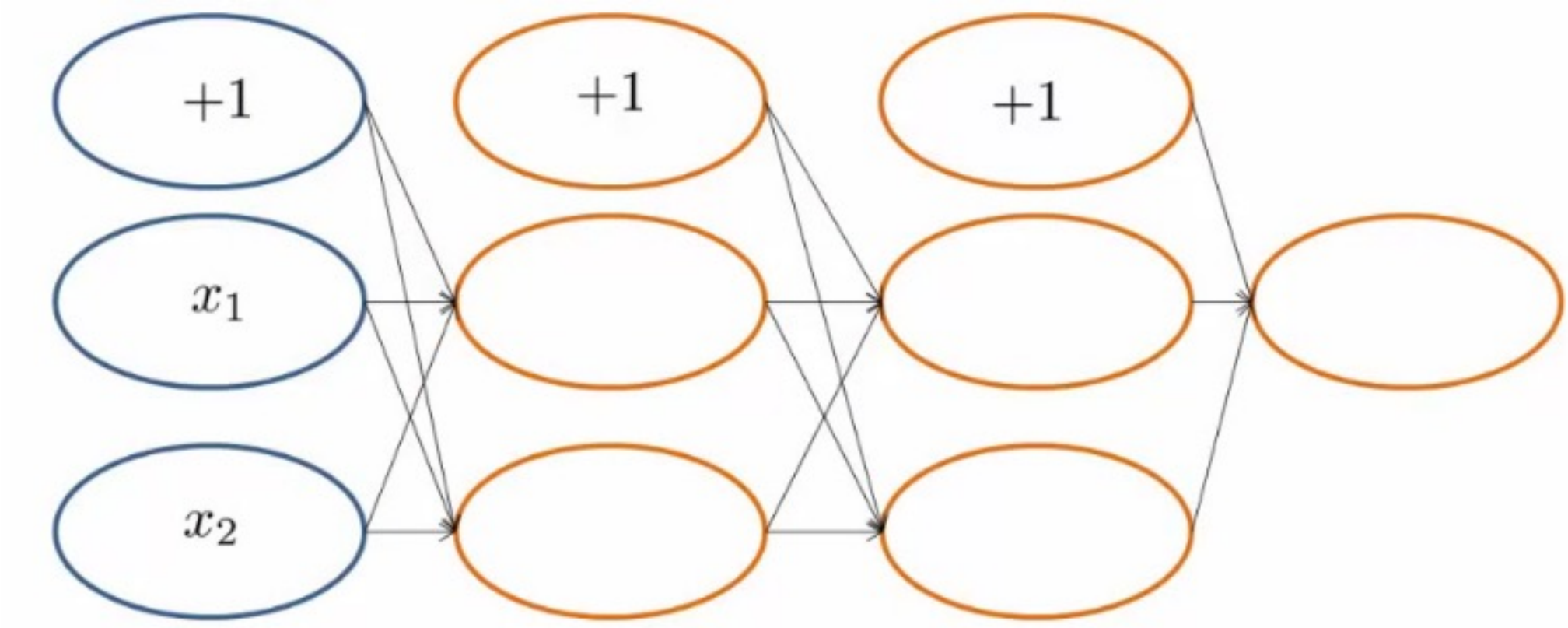
- After executing the body of the loop, exit the for loop and compute average

$$D_{ij}^{(l)} := \frac{1}{m} \triangle_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} \text{ if } j \neq 0$$

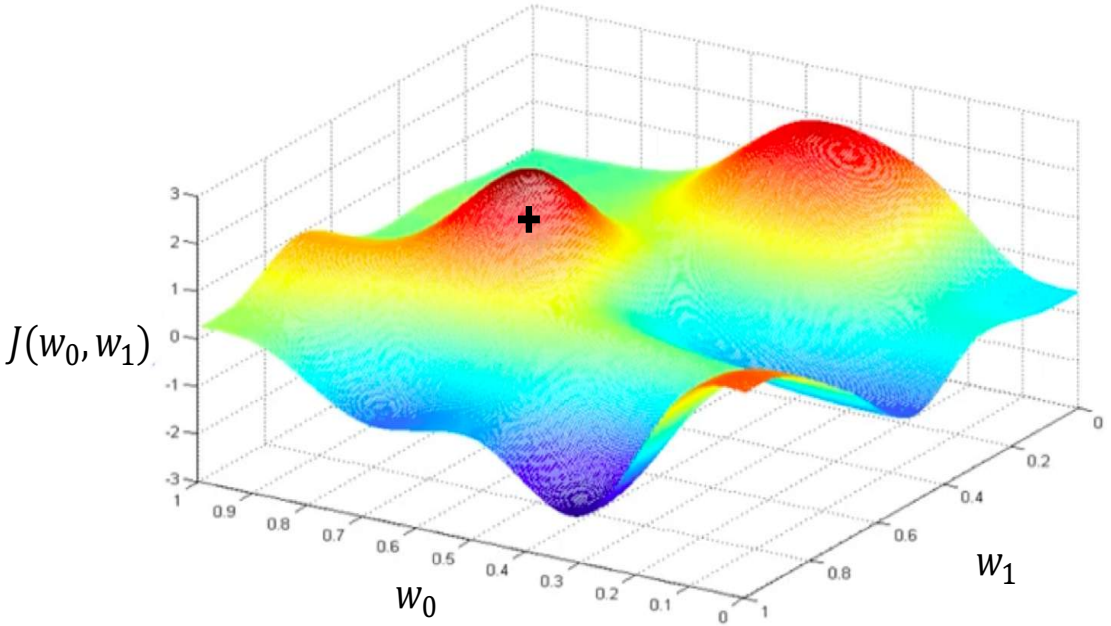
$$D_{ij}^{(l)} := \frac{1}{m} \triangle_{ij}^{(l)} \text{ if } j = 0$$

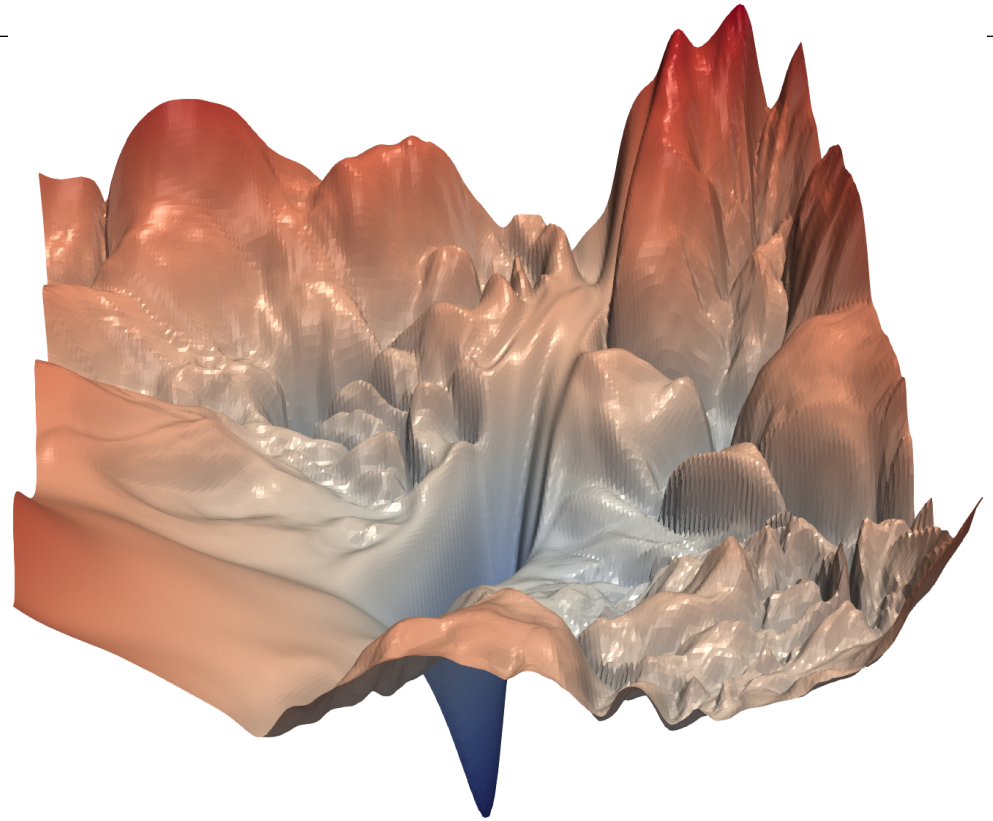
- We can show that each D is equal to the following
- $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$

Back propagation intuition



Loss surface in reality





Local minimas, saddle points

[Visualizing the loss landscape of neural nets, Li et al 2018]

Initialization of $\boldsymbol{\theta}$

- How to initialize $\boldsymbol{\theta}$?
  - Can we initialize them to 0s?
  - What happens if we do?
- Random Initialization
  - Weights are initialized randomly using normal distribution with standard deviation
  - Randomization helps breaking symmetry

Learning rate

- Small learning rates have slow convergence and get stuck in local minimas
- Large learning rates overshoot and might diverge
- Stable learning rates converge smoothly and avoid local minima

$$\theta = \theta - \alpha \frac{\partial J(\theta)}{\partial \theta}$$

How can we set the learning rate?

- Fix learning rates based on
  - The size of the gradient
  - Rate of learning
  - Size of particular weights
  - … <More on this in upcoming lectures>

Momentum

Adagrad

Adadelta

Adam

RMSProp

Mini-batches

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\theta)}{\partial \theta}$
4. Update weights, $\theta = \theta - \alpha \frac{\partial J(\theta)}{\partial \theta}$
5. Return weights

Computationally expensive

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point i
4. Compute gradient, $\frac{\partial J_i(\theta)}{\partial \theta}$
5. Update weights, $\theta = \theta - \alpha \frac{\partial J(\theta)}{\partial \theta}$
6. Return weights

Computationally easier, very noisy

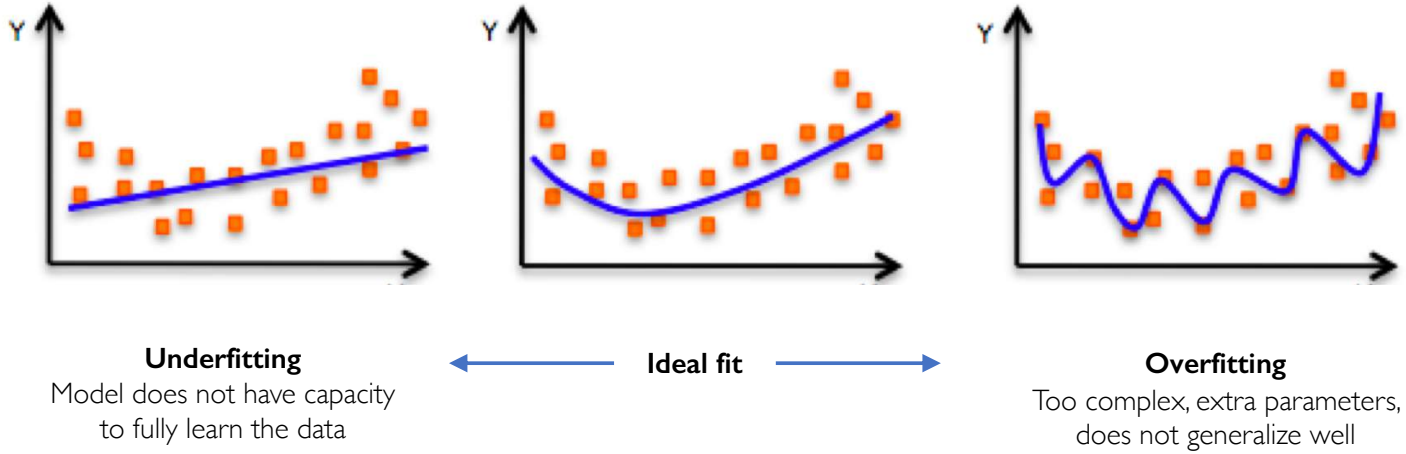
Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J_B(\theta)}{\partial \theta} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial J_k(\theta)}{\partial \theta}$
5. Update weights, $\theta = \theta - \alpha \frac{\partial J(\theta)}{\partial \theta}$
6. Return weights

Fast to compute and much better estimate of the true gradient

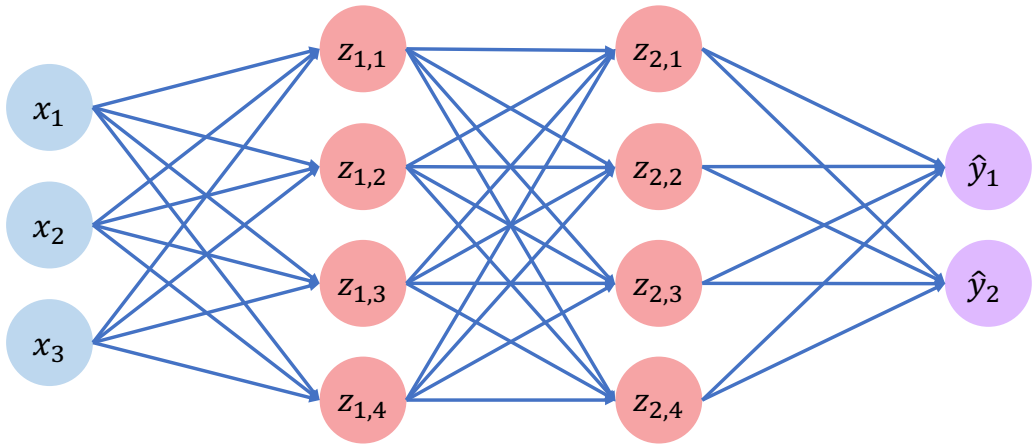
- Mini-batches : Computed the gradients for a batch of points rather than each
- Smoother convergence, allows for larger learning rates
- Can parallelize computation, fully use GPU potential

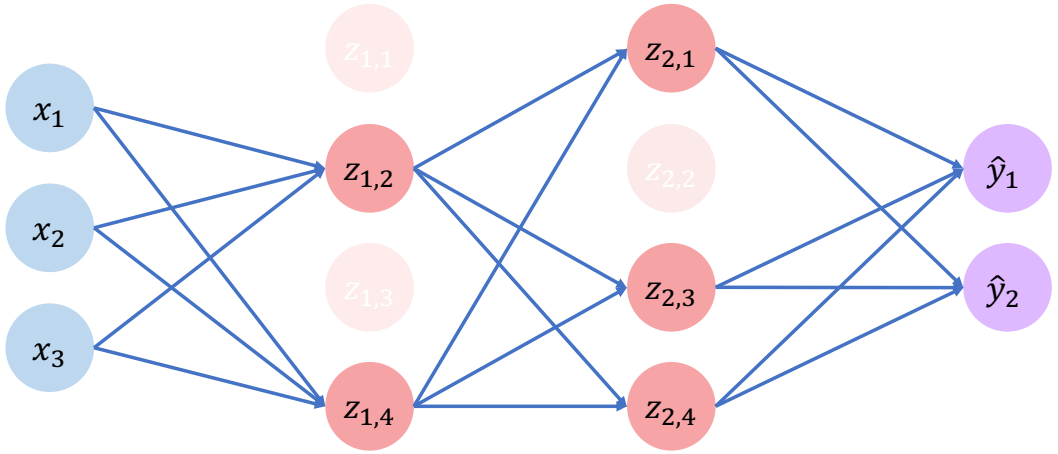
Overfitting and regularization



- Deep Neural Networks have a large capacity (leads to more complex models) and hence tend to overfit
- Regularization: Forces the model to learn simpler model
- How do we regularize deep networks ?

Dropout



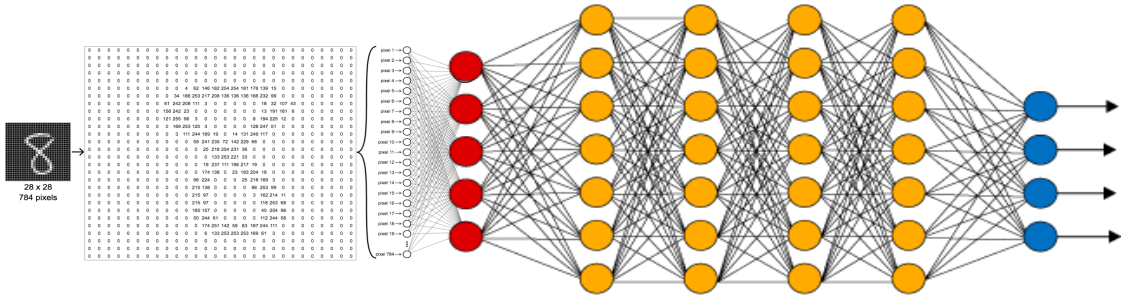


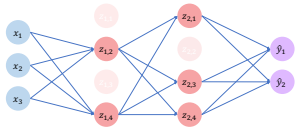
- Drop some of the neurons (setting activation = 0)
- Forces the network to learn using a smaller capacity --> robust to overfitting

Early Stopping (optional)

- Stop before there is a chance to overfit

Summary





- Deep neural networks are compositions of perceptrons
- Activations provide non-linearity
- Back Propagation for parameter learning

- In practice
  - pay attention to learning rates
  - Use mini-batches
  - regularization using drop-outs, early stopping

Deriving Sigmoid Gradient

Deriving the Sigmoid Gradient Function

We let the sigmoid function be $\sigma(x) = \frac{1}{1+e^{-x}}$

Deriving the equation above yields to $(\frac{1}{1+e^{-x}})^2 \frac{d}{ds} \frac{1}{1+e^{-x}}$

Which is equal to $(\frac{1}{1+e^{-x}})^2 e^{-x}(-1)$

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

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

$\sigma(x)(1-\sigma(x))$

Literature

- Chapter 5 from the Tan et. al. Textbook.
- Additional Resources for Backpropagation
- Very thorough conceptual [example] (https://web.archive.org/web/20150317210621/https://www4.rgu.ac.uk/files/chapter3%20-%20bp.pdf)
- Short derivation of the backpropagation algorithm: http://pandamatak.com/people/anand/771/html/node37.html
- Very thorough explanation and proof: http://neuralnetworksanddeeplearning.com/chap2.html