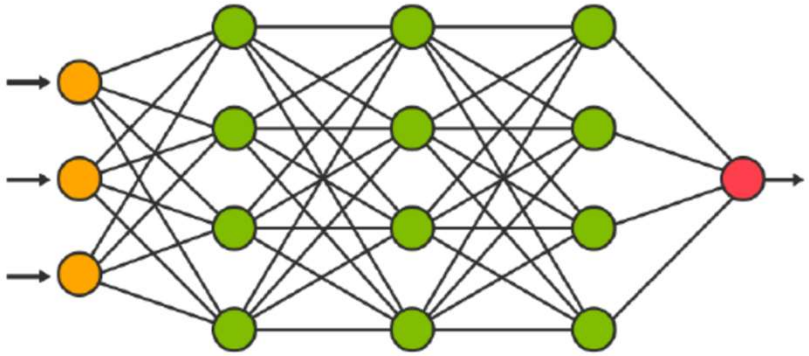
Neural Networks

Fatemeh Mansoori

University of Isfahan

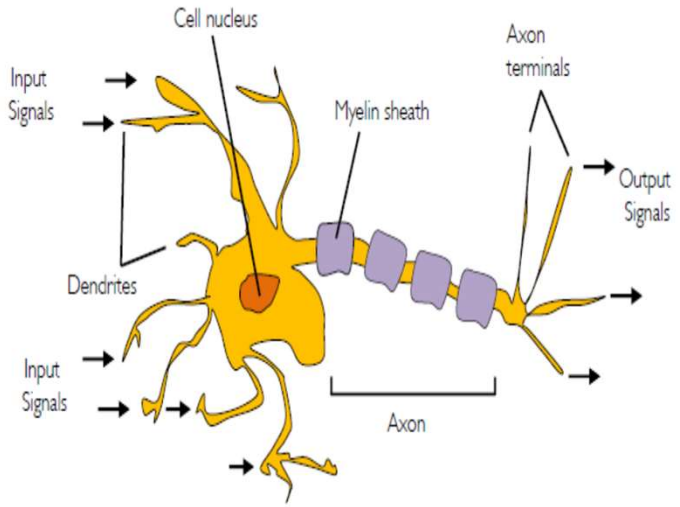
Introduction to Neural Network



This slides are created based on the slide of the machine learning course at sharif university,

Biological Analogy

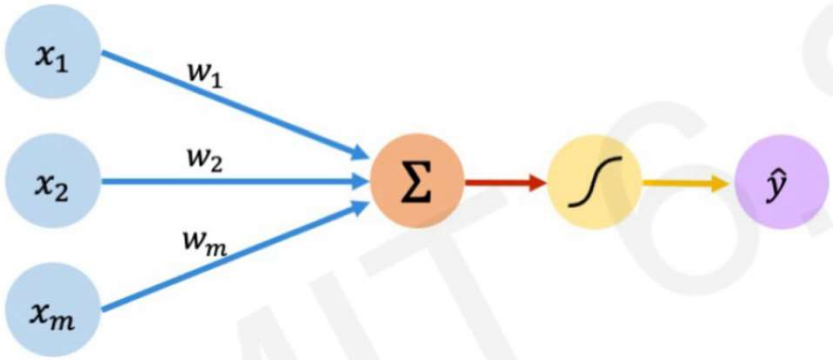
- A brain is a set of densely connected neurons
- Depending on the input signals, the neuron performs computations and decides to fire or not.



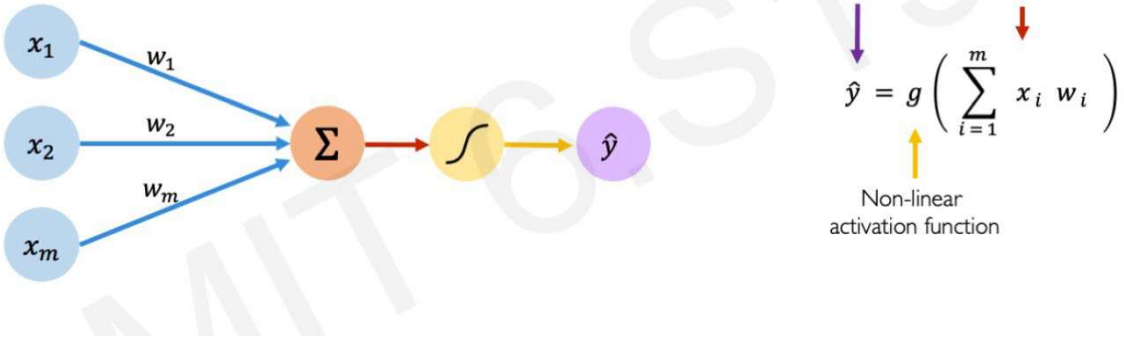
Mathematical formulation of a biological neuron,
could solve AND, OR, NOT problems

A simple mathematical model of neuron

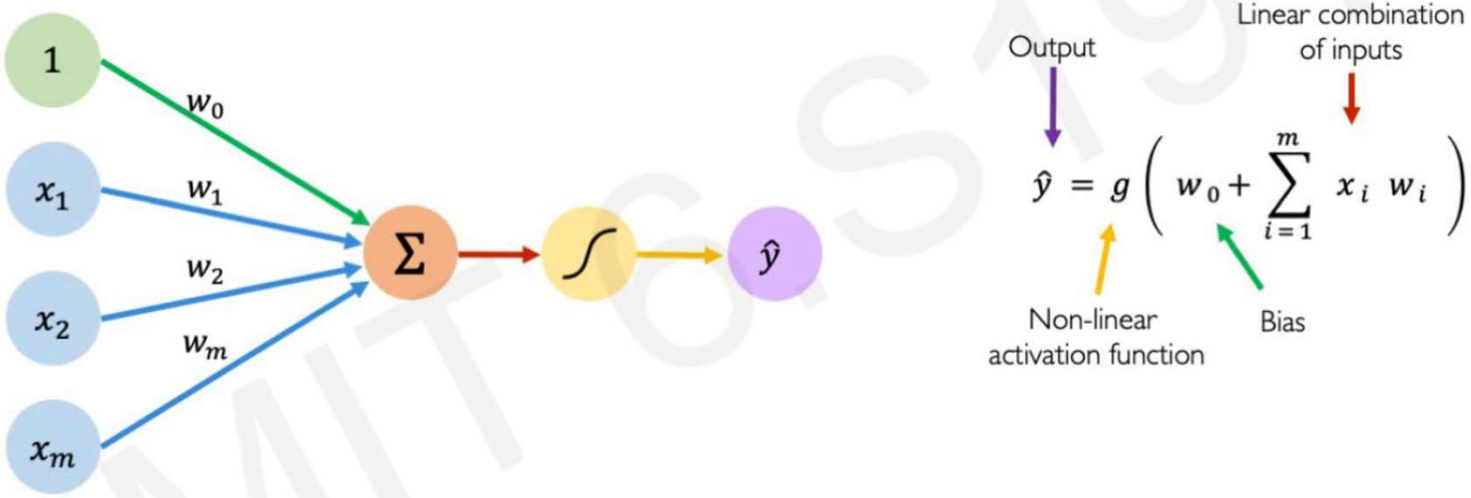
- McCulloch & Pitt's neuron model (1943)
- A linear classifier __ it fires when a linear combinations of its input exceeds some threshold.



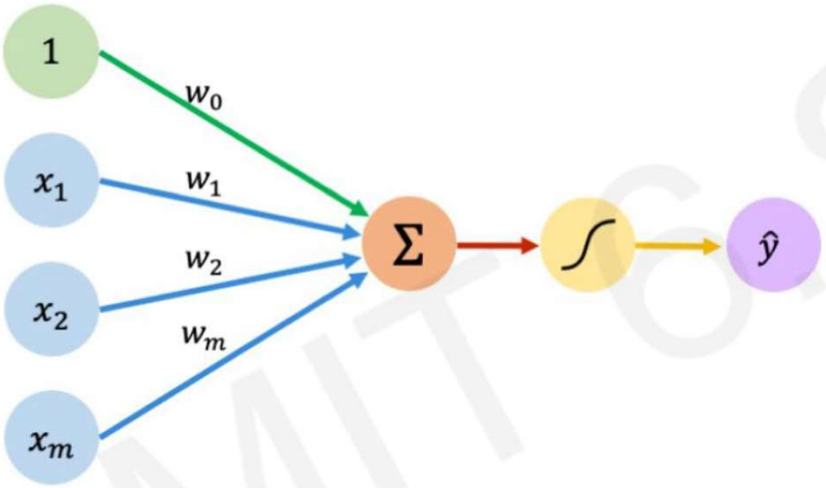
A simple mathematical model of a neuron



A simple mathematical model of a neuron



A simple mathematical model of a neuron



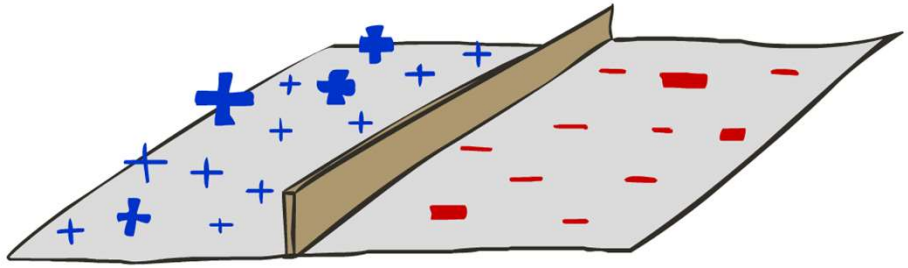
$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i\, w_i\right)$$

$$\hat{y} = g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W}\right)$$

where: $\boldsymbol{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ and $\boldsymbol{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$

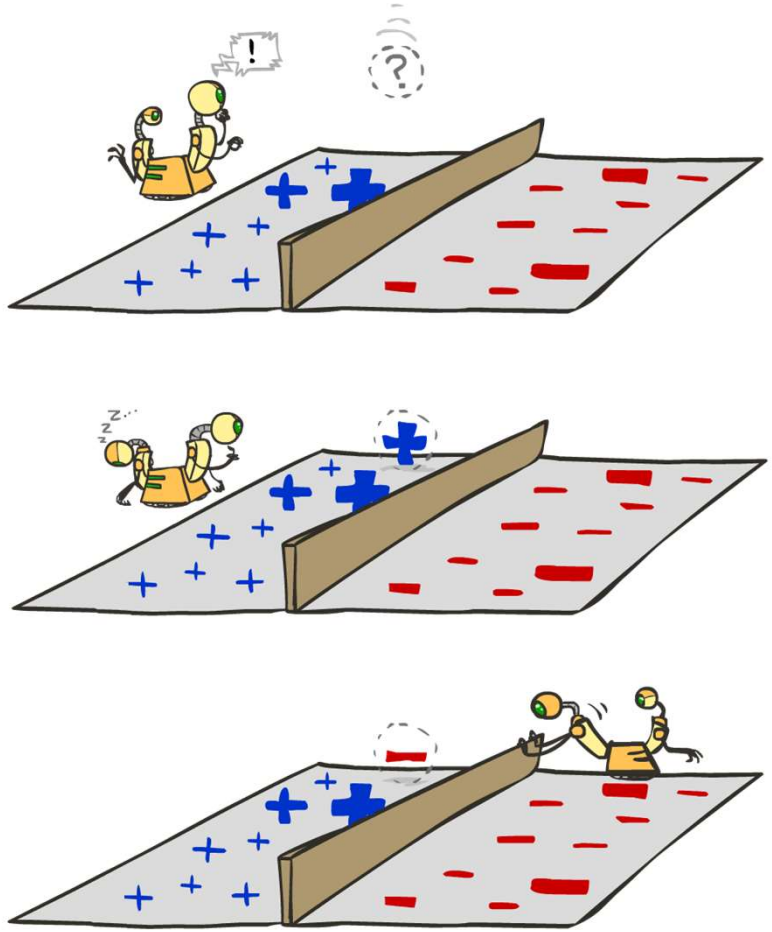
Binary Decision Rule

- In the space of feature vectors
  - Examples are points
  - Any weight vector is a hyperplane
  - One side corresponds to Y=+1
  - Other corresponds to Y=-1



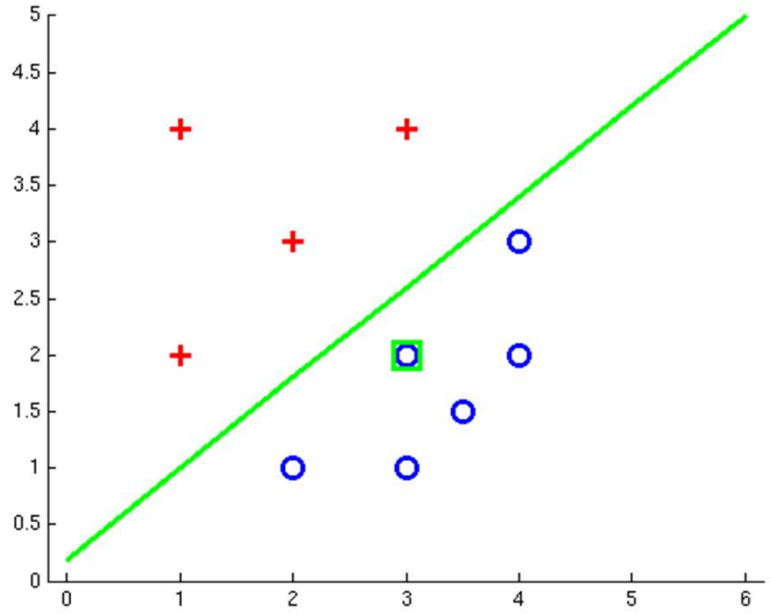
Learning

- Start with weights = 0
- For each training instance:
  - Classify with current weights
  - If correct (i.e., y=y*), no change!
  - If wrong: adjust the weight vector

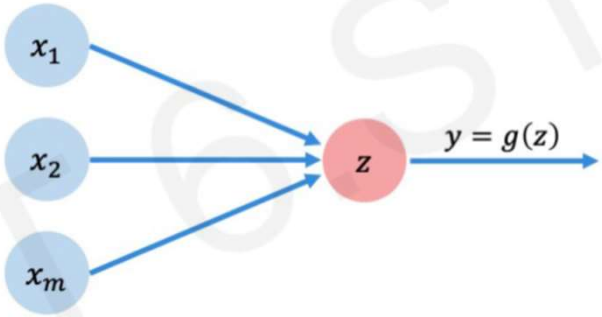


Examples: Perceptron

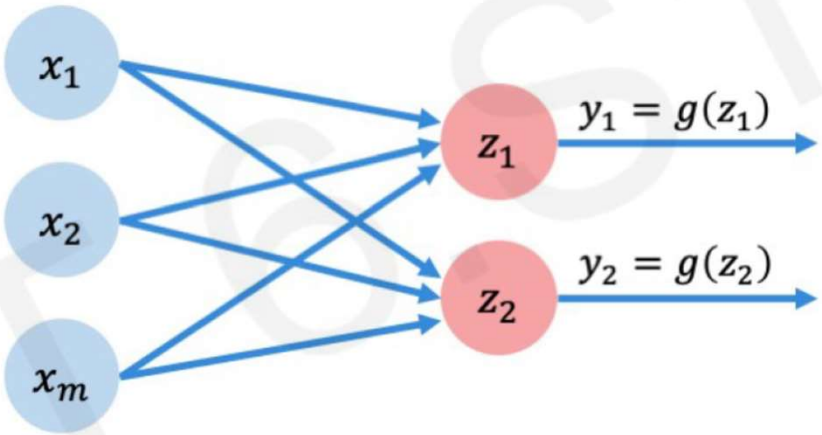
- Separable Case



Multi Output Perceptron



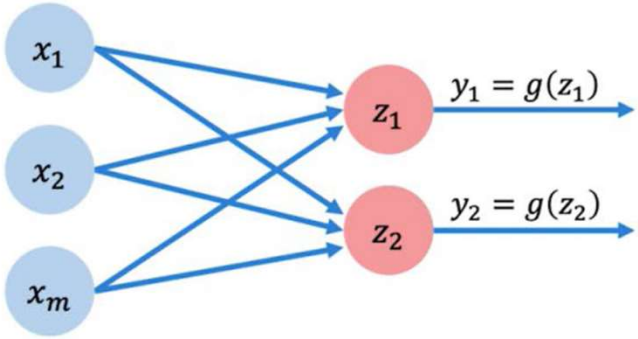
$$z = w_0 + \sum_{j=1}^{m} x_j \, w_j$$



$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j \, w_{j,i}$$

Multi Output perceptron

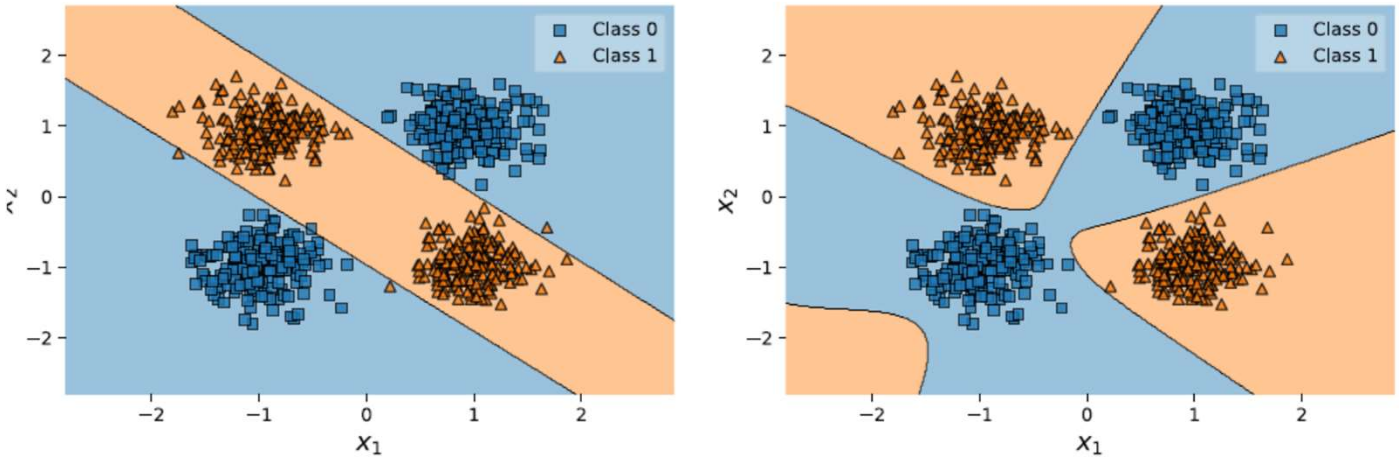
Because all inputs are densely connected to all outputs, these layers are called **Dense** layers



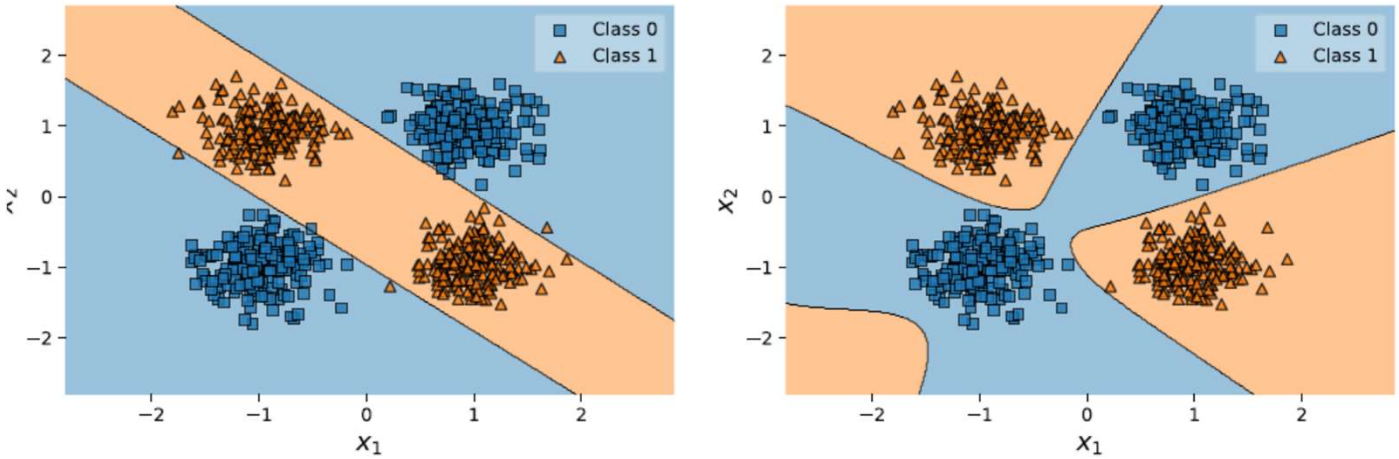
Limitation of perceptron

- XOR can not be represented by perceptron
- We need a deeper network
- No one knew how to train deeper networks

Why can’t a perceptron represent XOR?

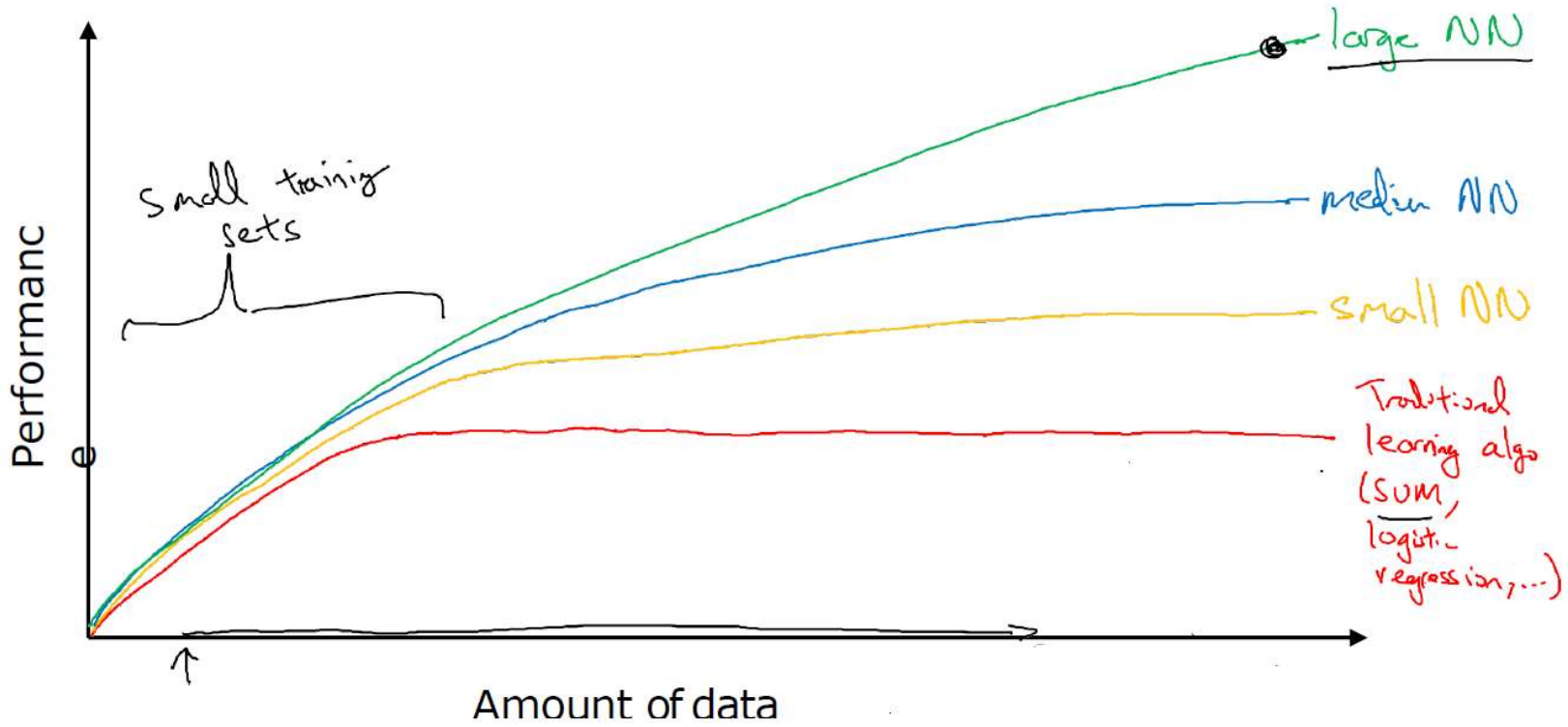


Multilayer Neural Networks Can Solve XOR Problems

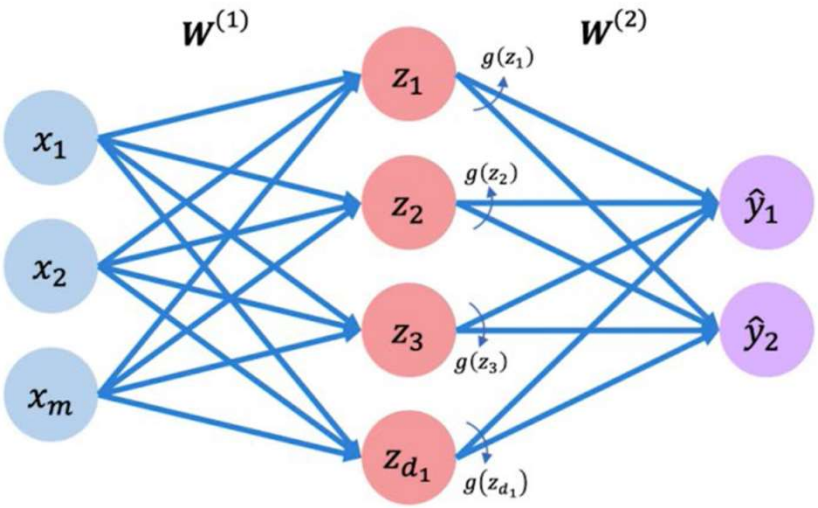


- Decision boundaries of two different multilayer perceptions on simulated data solving the XOR problem

Scale drives deep learning progress



Neural Network with 1 Hidden Layer



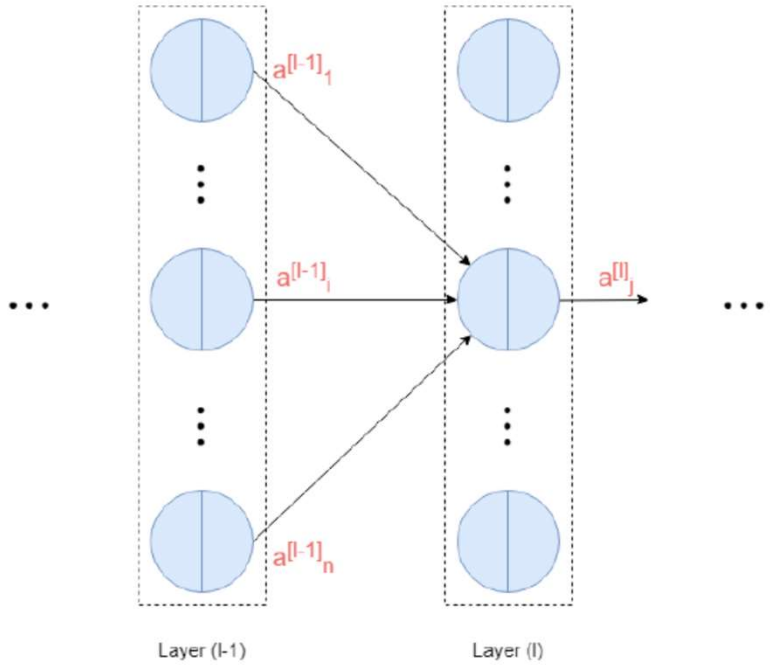
$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)}$$

$$\hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} g(z_j) \, w_{j,i}^{(2)}\right)$$

Standard notations for MLP

$a_i^{[l]}$: i-th neuron output in layer l

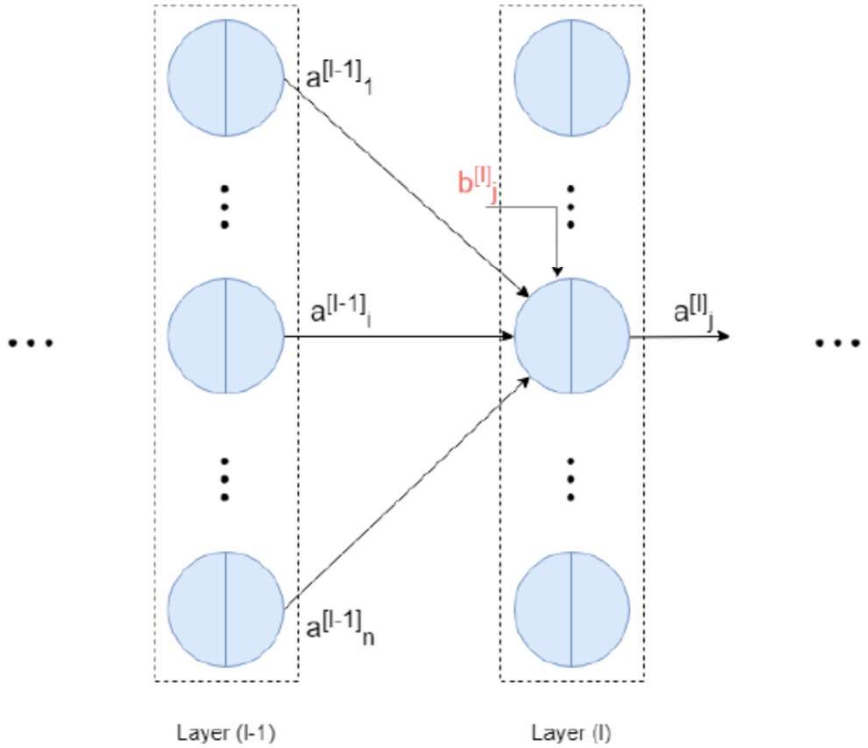
$\boldsymbol{a}^{[l]}$: layer l output in vector form



MLP notation

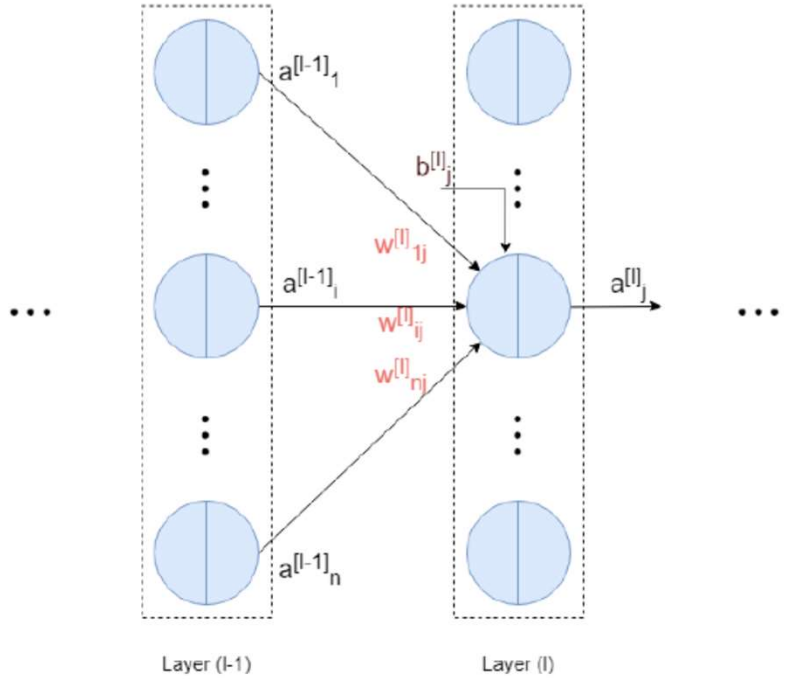
$b_i^{[l]}$: i-th neuron bias in layer l

$\boldsymbol{b}^{[l]}$: layer l biases in vector form



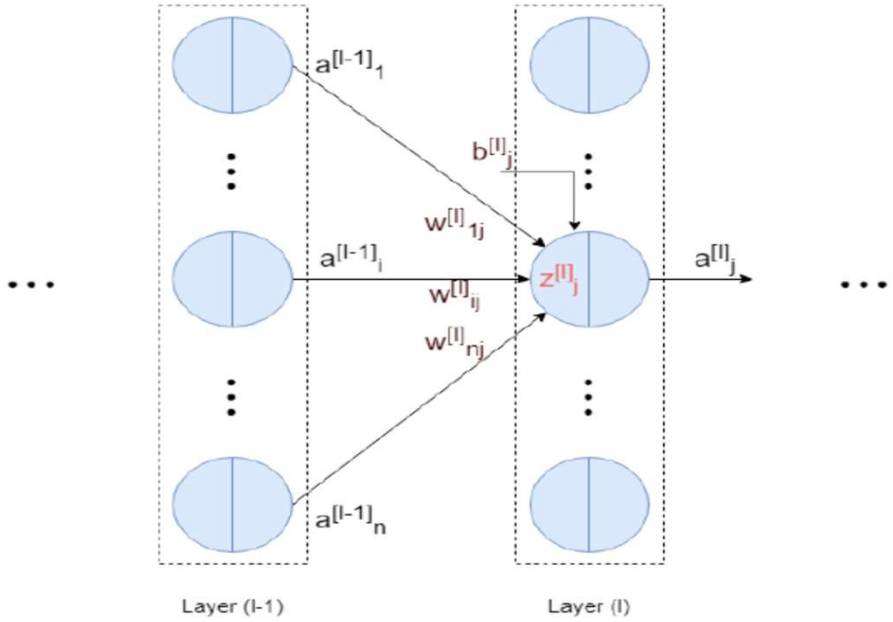
MLP notation

$W_{ij}^{[l]}$: weight of the edge between i-th neuron in layer $l-1$ and j-th neuron in layer l



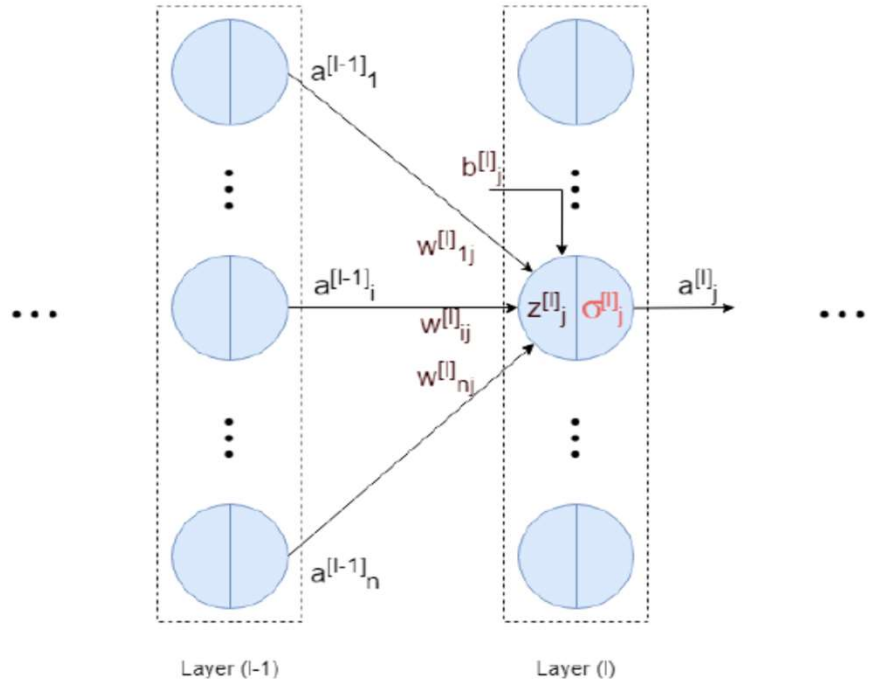
MLP notation

- $z_j^{[l]}$: j-th neuron input in layer l
- $z_j^{[l]} = b_j^{[l]} + \sum_{i=1}^{n} W_{ij}^{[l]} a_i^{[l-1]}$



MLP notation

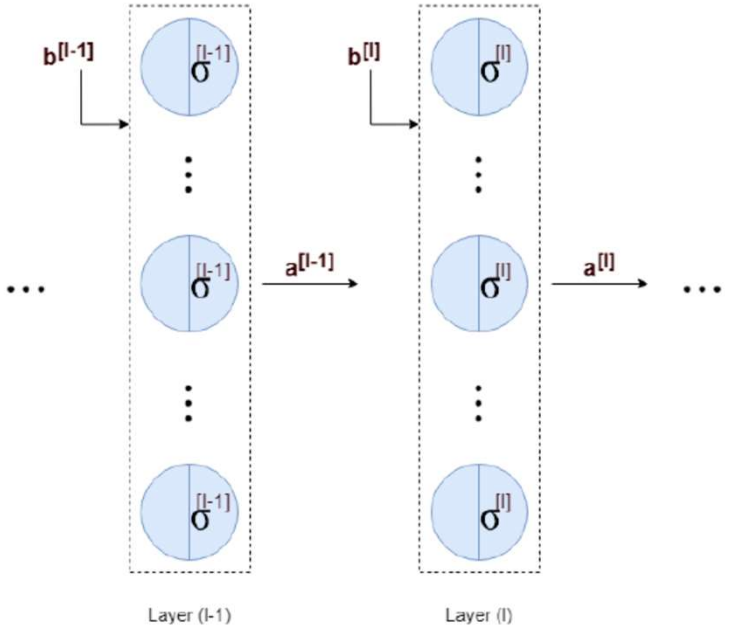
$\sigma_j^{[l]}$: j-th neuron activation function in layer l



MLP notation

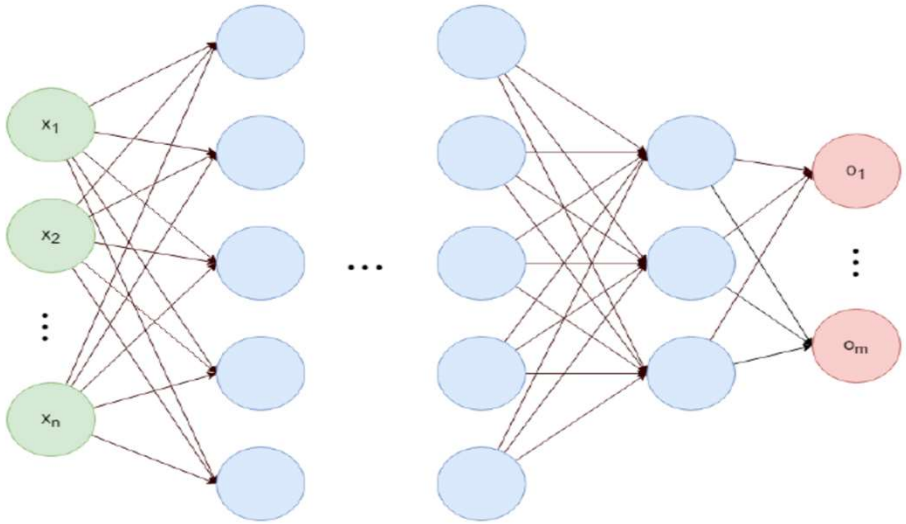
- If all neurons in one layer have the same activation function then :

$$\boldsymbol{a}^{[l]} = \sigma^{[l]}\left(\boldsymbol{b}^{[l]} + (W^{[l]})^T \boldsymbol{a}^{[l-1]}\right)$$



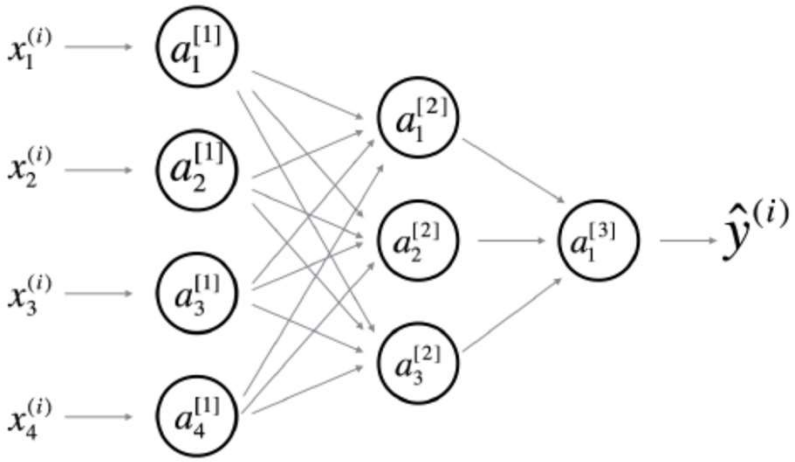
MLP notation

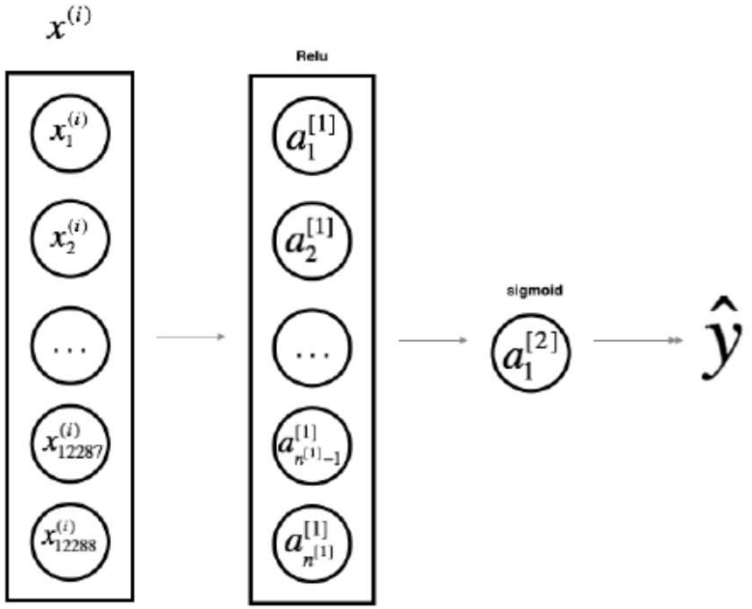
- For a network with L layer and x as its input we have :



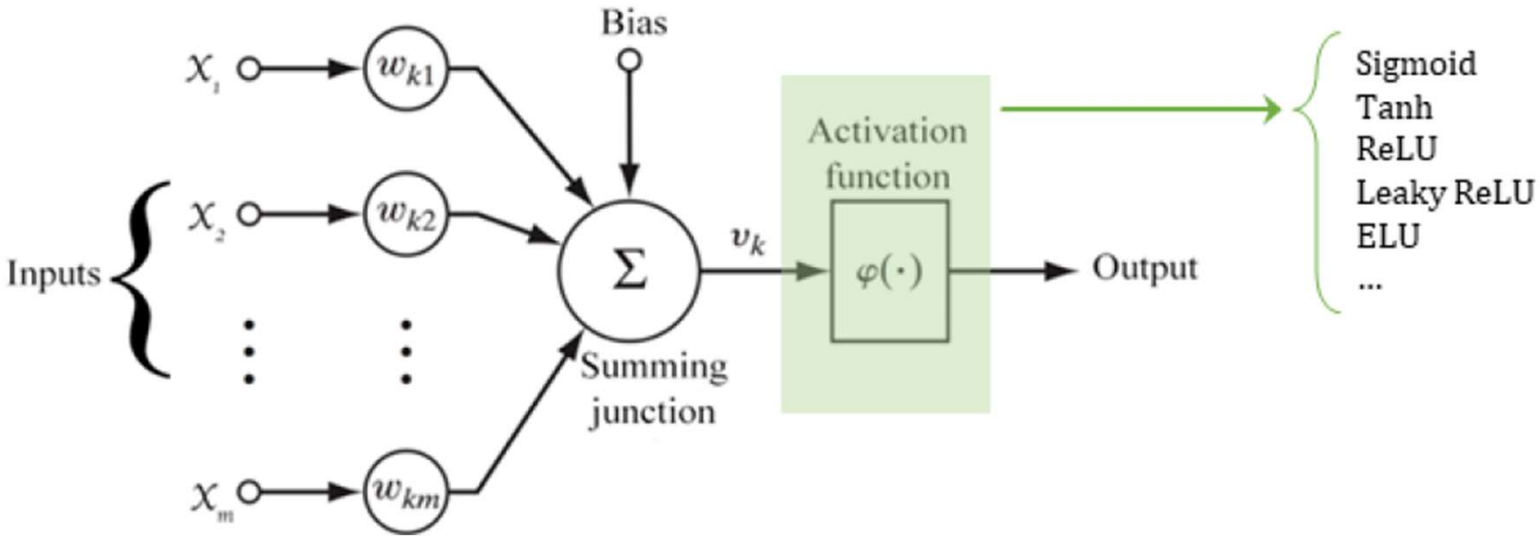
$$\boldsymbol{o} = \boldsymbol{a}^{[L]} = \sigma^{[L]}\left(\boldsymbol{b}^{[L]} + (W^{[L]})^T \sigma^{[L-1]}\left(\cdots \sigma^{[1]}\left(\boldsymbol{b}^{[1]} + (W^{[1]})^T \boldsymbol{x}\right)\cdots\right)\right)$$

Deep Learning representations



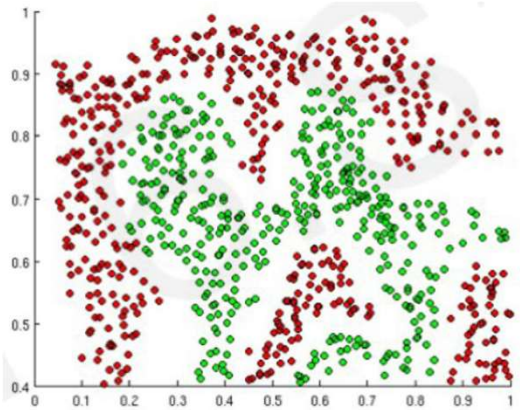


Activation function



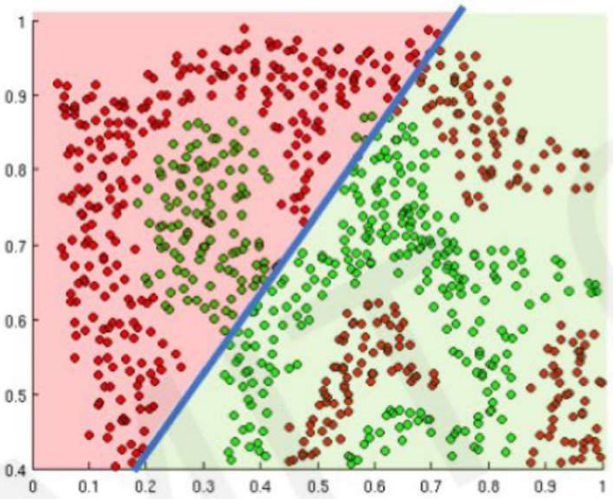
Importance of Activation Function

- The propose of activation function is to introduce non-linearities into network

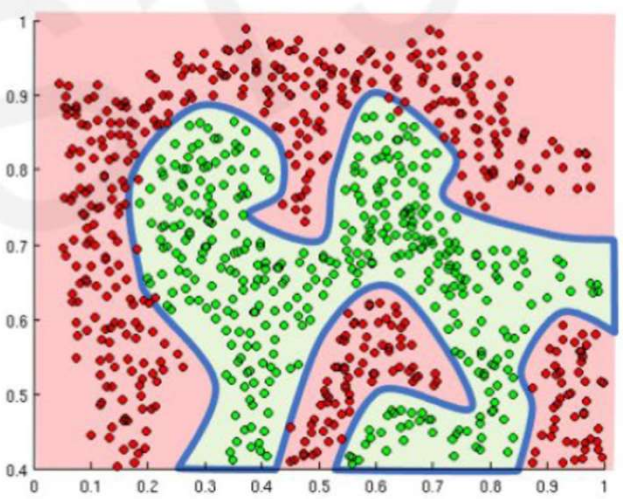


- What if we wanted to build a neural network to distinguish green vs red points?

Importance of Activation Function



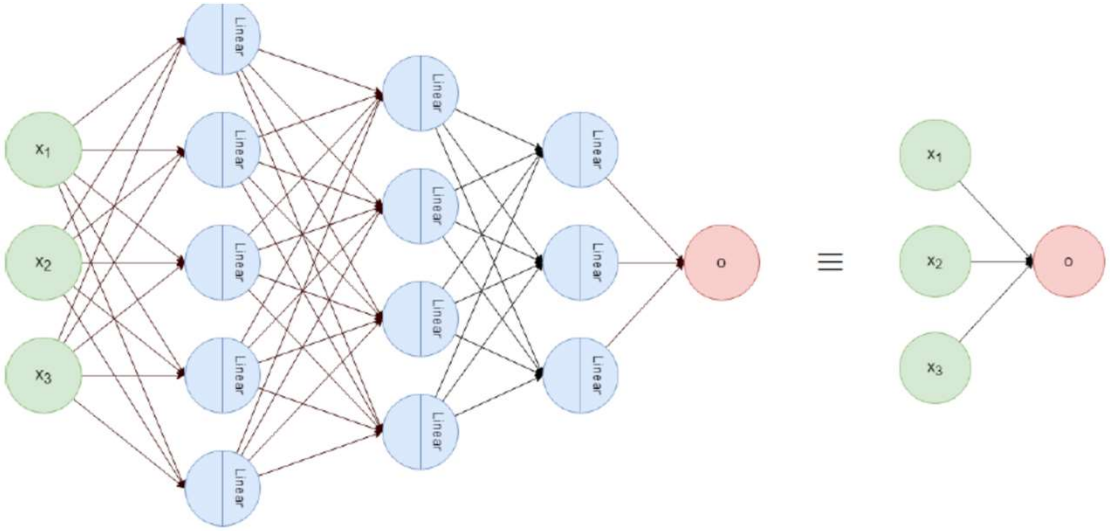
Linear activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

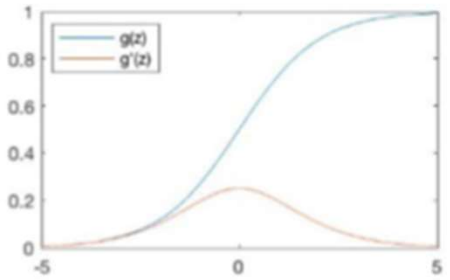
Activation Function of Hidden Layers

- One can use any activation function for each hidden units
- Usually people use the same activation function for all neurons in one Layer
- The important point is to use nonlinear activation functions



Common Activation functions

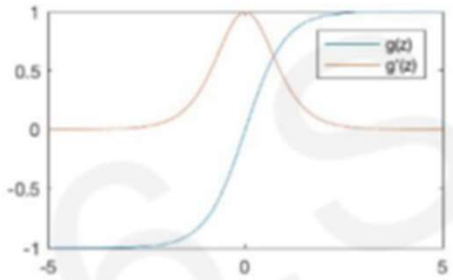
Sigmoid Function



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

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

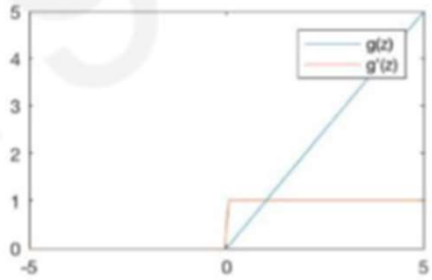
Hyperbolic Tangent



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

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

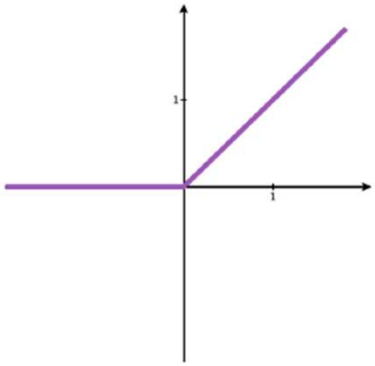
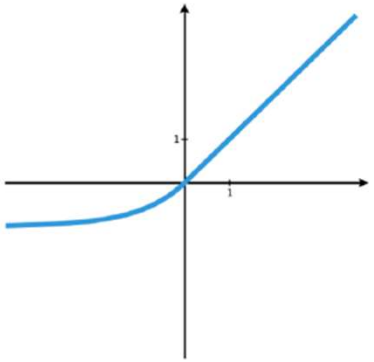
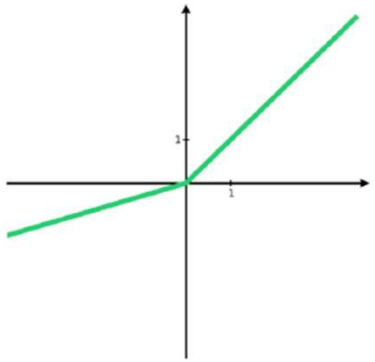
Rectified Linear Unit (ReLU)



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

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

Variation of ReLU

ReLU	ELU	Leaky ReLU
$f(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}$	$f(x) = \begin{cases} x & x > 0 \\ \alpha(e^x - 1) & x \leq 0 \end{cases}$	$f(x) = \begin{cases} x & x \geq 0 \\ 0.01x & x < 0 \end{cases}$
		

Softmax activation function

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \ldots, K \text{ and } \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K. \qquad \sigma : \mathbb{R}^K \to (0,1)^K,$$

The softmax activation function takes in a vector of **raw outputs** of the neural network and returns a vector of **probability scores**.

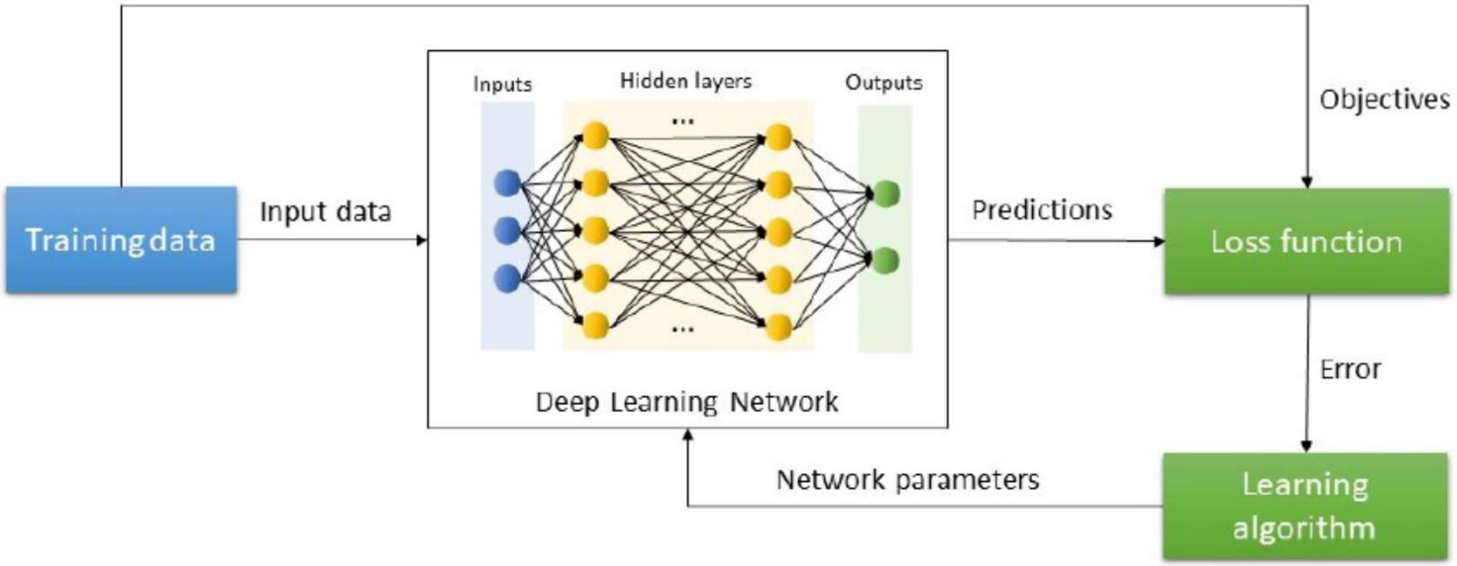
- **z** is the vector of raw outputs from the neural network

- The i-th entry in the softmax output vector softmax(**z**) can be thought of as the predicted probability of the test input belonging to class i.

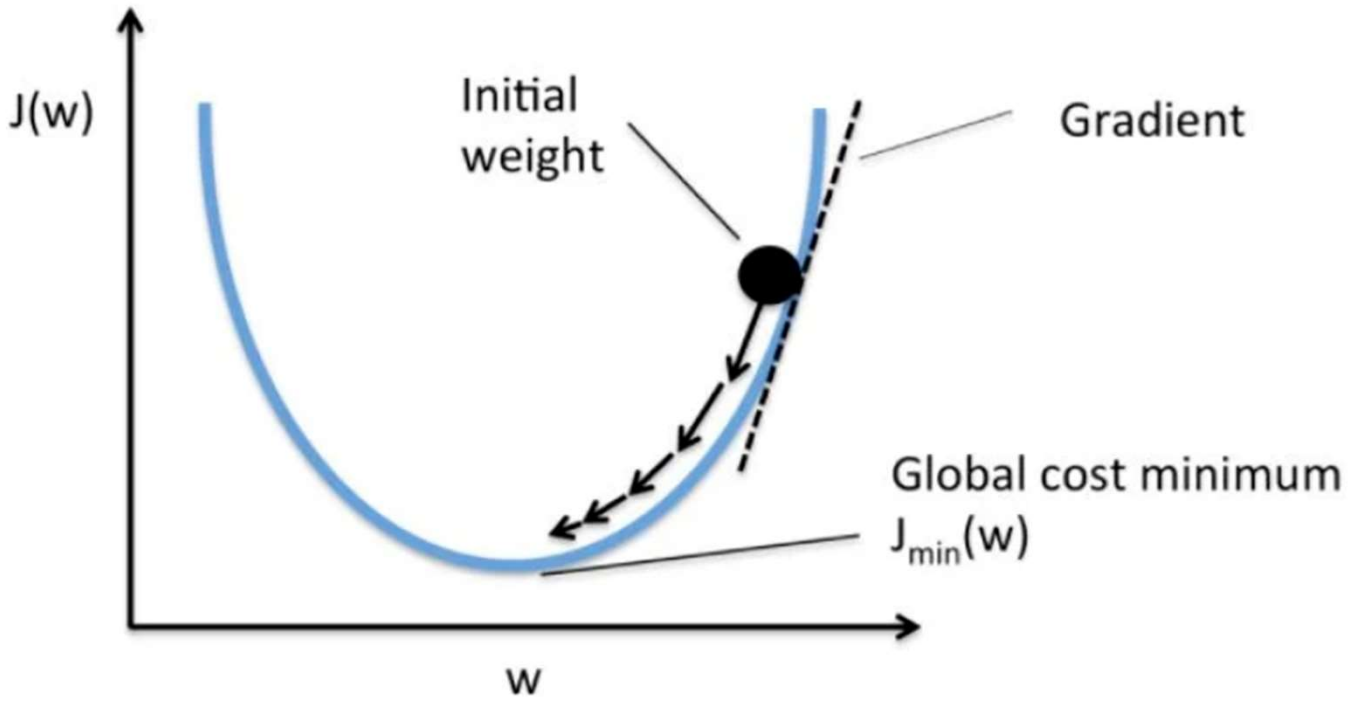
regardless of whether the input x is positive, negative, or zero, e^x is always a positive number.

Why Won't Normalization by the Sum Suffice ?

Training Process

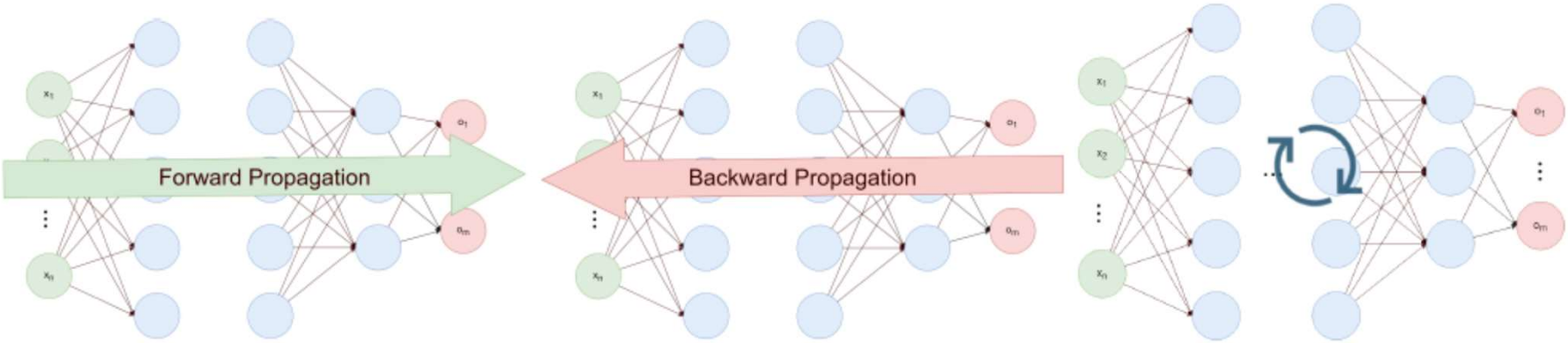


Gradient Descent



Training process

- The idea is to use gradient descent



(a) Forward pass (b) Backward pass (c) Update parameters

Learning MLPs

Let's define the learning problem more formal:

- $\{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$: dataset
- f: network
- W: all weights and biases of the network ($W^{[l]}$ and $\boldsymbol{b}^{[l]}$ for different l)
- L: loss function

We want to find W^* which minimizes following cost function:

$$\mathcal{J}(W) = \sum_{i=1}^{n} L\left(f(x^{(i)}; W), y^{(i)}\right)$$

We are going to use gradient descent, so we need to find $\nabla_W \mathcal{J}$.

Forward propagation

First of all we need to find loss value.

It only requires to know the inputs of each neuron.

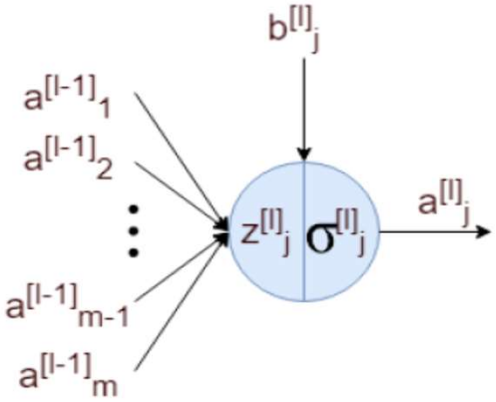


Figure: $a_j^{[l]} = \sum_{i=1}^{m} W_{ij}^{[l]} a_i^{[l-1]} + b_j^{[l]}$

So we can calculate these outputs layer by layer.

Forward propagation

After forward pass we will know:

- Loss value
- Network output
- Middle values

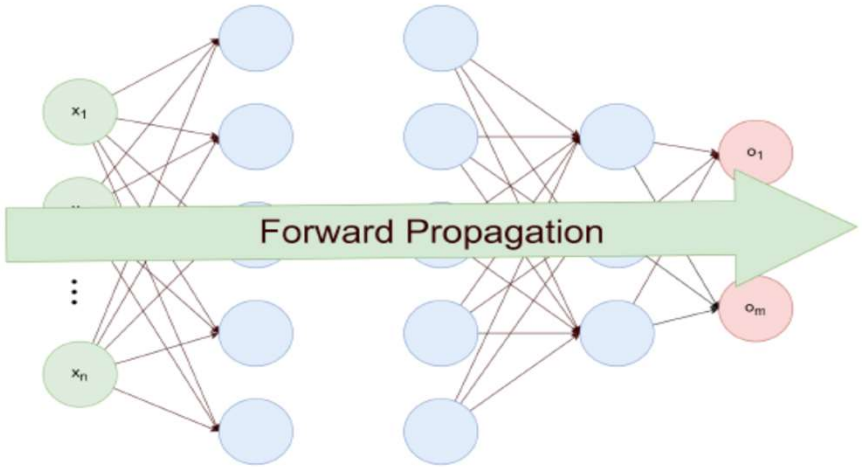


Figure: Forward pass

Backward Propagation

Now we need to calculate $\nabla_W \mathcal{J}$.

First idea:

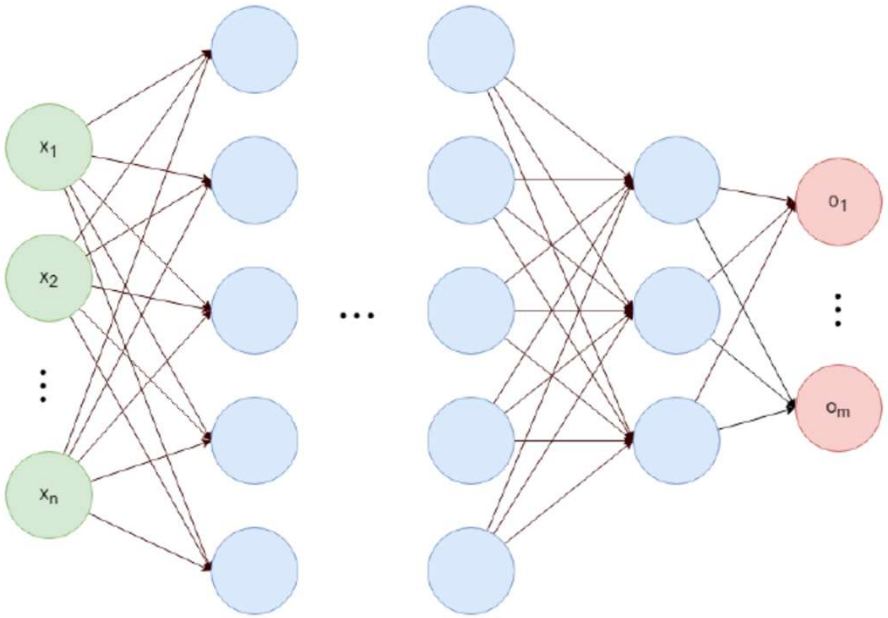
- Use analytical approach.
- Write down derivatives on paper.
- Find the close form of $\nabla_W \mathcal{J}$ (if it is possible to do so).
- Implement this gradient as a function to work with.

- Pros:
  - Fast
  - Exact

- Cons:
  - Need to rewrite calculation for different architectures

The problem of first idea

How to compute the derivation of the blow function



$$\boldsymbol{o} = \boldsymbol{a}^{[L]} = \sigma^{[L]}\left(\boldsymbol{b}^{[L]} + (W^{[L]})^T\sigma^{[L-1]}\left(\cdots\sigma^{[1]}\left(\boldsymbol{b}^{[1]} + (W^{[1]})^T\boldsymbol{x}\right)\cdots\right)\right)$$

Backward propagation

Second idea:

- Using modular approach.
- Computing the cost function consists of doing many operations.
- We can build a computation graph for this calculation.
- Each node will represent a single operation.

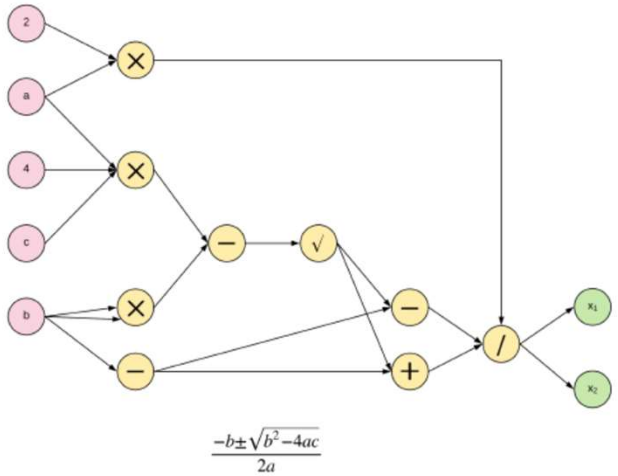
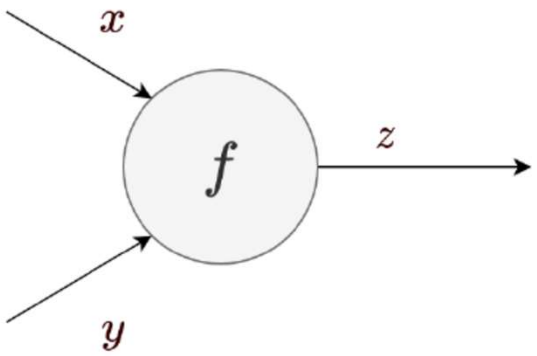


Figure: An example of computational graph, Source.

Backward Propagation

In this approach if we know how to calculate gradient for single node or module, then we can find gradient with respect to each variables.

Let's say we have a module as follow:



It gets x and y as its input and returns $z = f(x, y)$ as its output.

How to calculate derivative of loss with respect to module inputs?

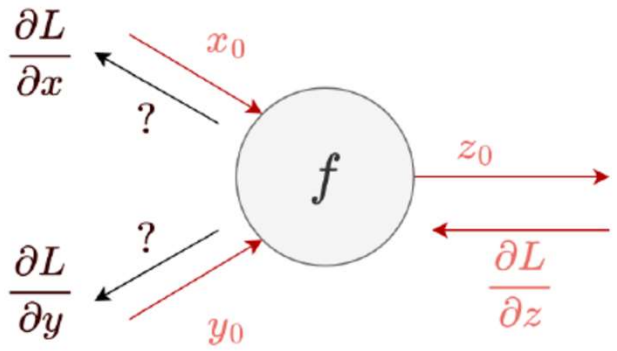
Backward propagation

We know:

- Module output for x_0 and y_0, let's call it z_0.
- Gradient of loss with respect to module output at z_0, $\left(\frac{\partial L}{\partial z}\right)$.

We want:

- Gradient of loss with respect to module inputs at x_0 and y_0, $\left(\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}\right)$.

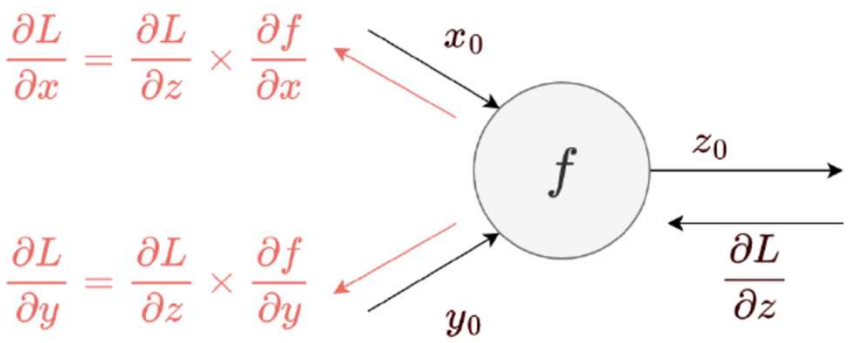


Backward Propagation

- We can use chain rule to do so.

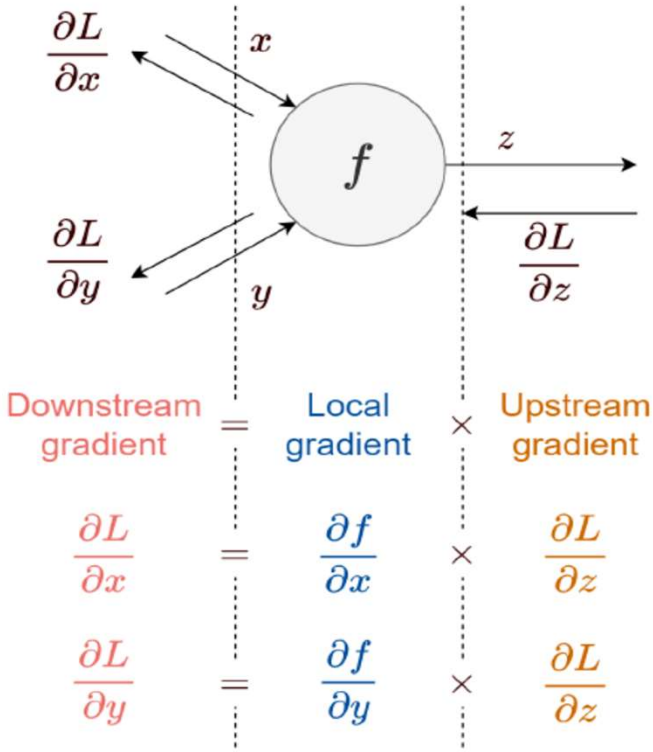
Chain rule:

$$\left.\begin{array}{l} z = f(x, y) \\ L = L(z) \end{array}\right\} \quad \Longrightarrow \quad \frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \times \frac{\partial z}{\partial x}$$



Backward Propagation

Backpropagation for single module:

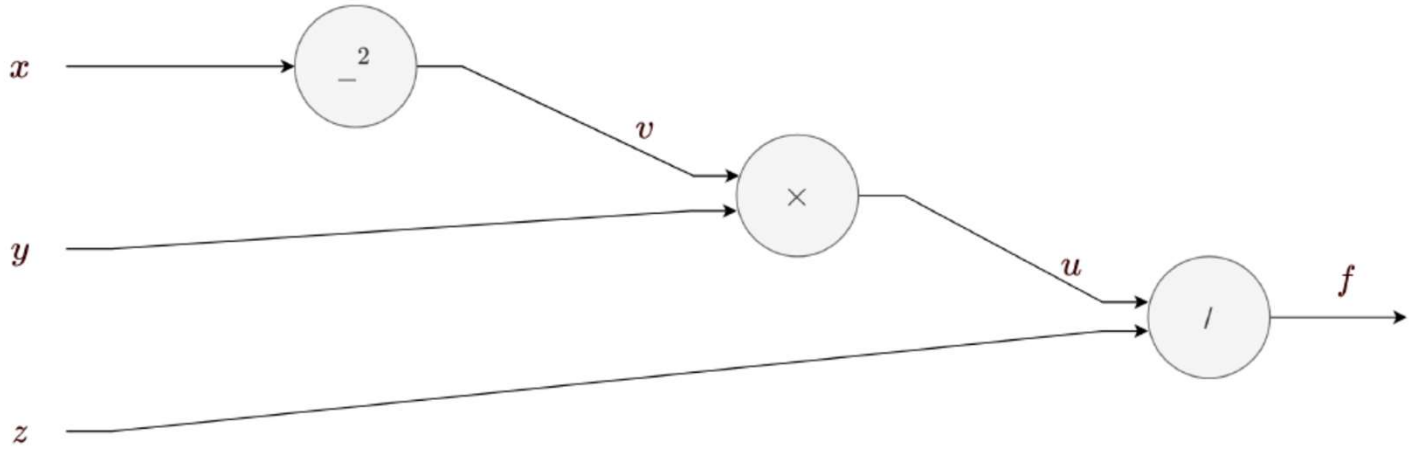


Backward Propagation : Example

Let's solve a simple example using backpropagation.

We have $f(x, y, z) = \frac{x^2 y}{z}$.

Find $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$ and $\frac{\partial f}{\partial z}$ at $x = 3$, $y = 4$ and $z = 2$.



Backward Propagation: Example

First let's find gradient analytically.

We have:

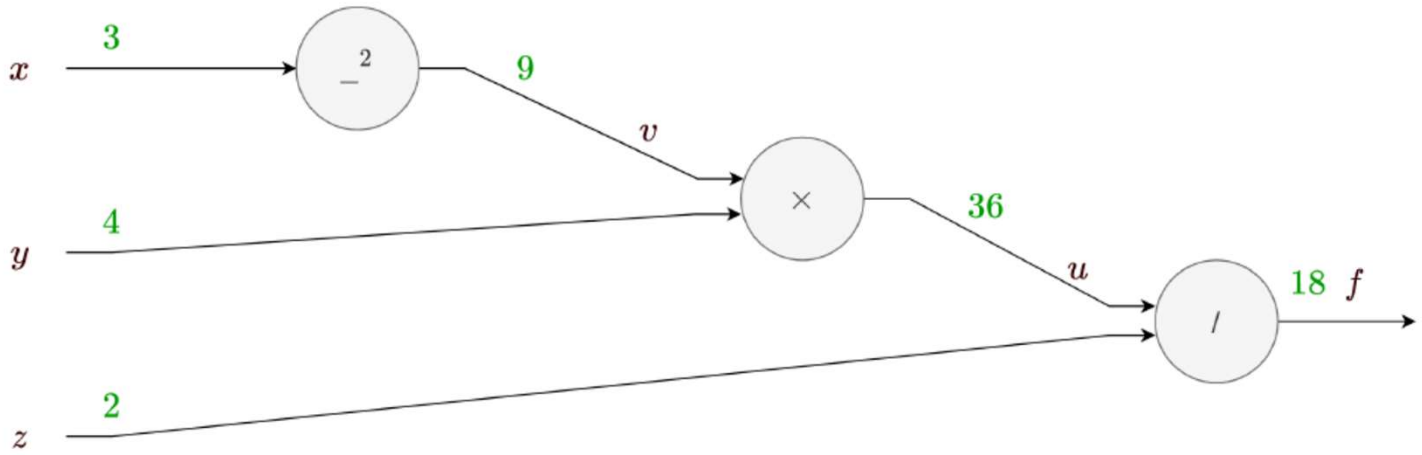
$$\begin{cases} v = x^2 \\ u = vy \\ f = \frac{u}{z} \end{cases} \qquad \begin{cases} \frac{\partial f}{\partial z} = -\frac{u}{z^2} \\ \frac{\partial f}{\partial y} = \frac{\partial u}{\partial y} \times \frac{\partial f}{\partial u} = v \times \frac{1}{z} = \frac{v}{z} \\ \frac{\partial f}{\partial x} = \frac{\partial v}{\partial x} \times \frac{\partial u}{\partial v} \times \frac{\partial f}{\partial u} = 2x \times y \times \frac{1}{z} = \frac{2xy}{z} \end{cases}$$

$$(x = 3, y = 4, z = 2) \implies \begin{cases} v = 9 \\ u = 36 \\ f = 18 \end{cases} \implies \begin{cases} \frac{\partial f}{\partial z} = -9 \\ \frac{\partial f}{\partial y} = 4.5 \\ \frac{\partial f}{\partial x} = 12 \end{cases}$$

Backward Propagation : Example

Now let's use backpropagation.

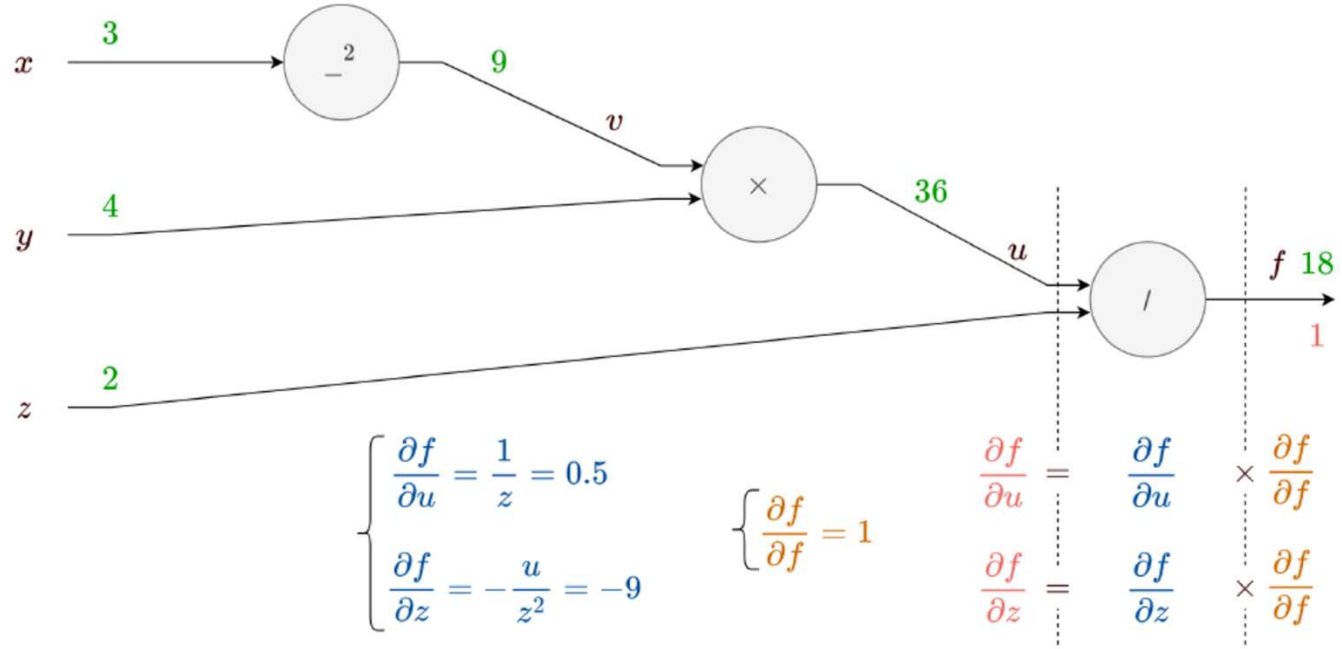
First we do forward propagation.



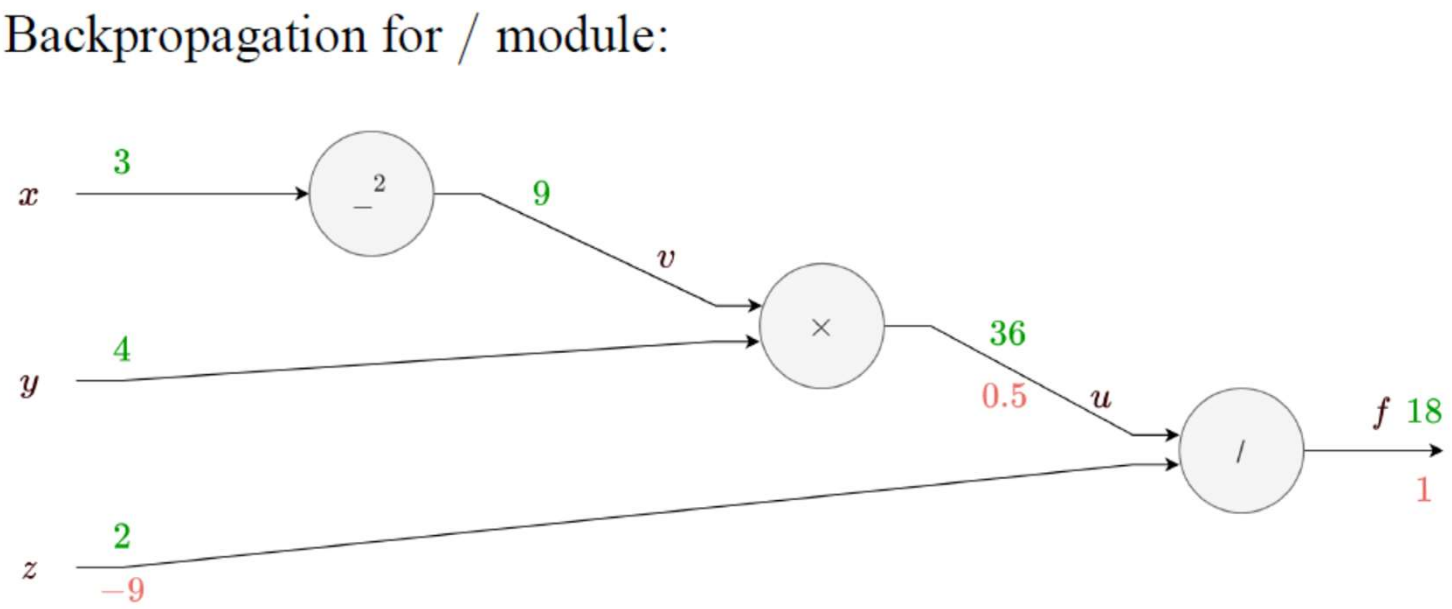
Second we will do backpropagation for each module.

Backward Propagation

Backpropagation for / module:

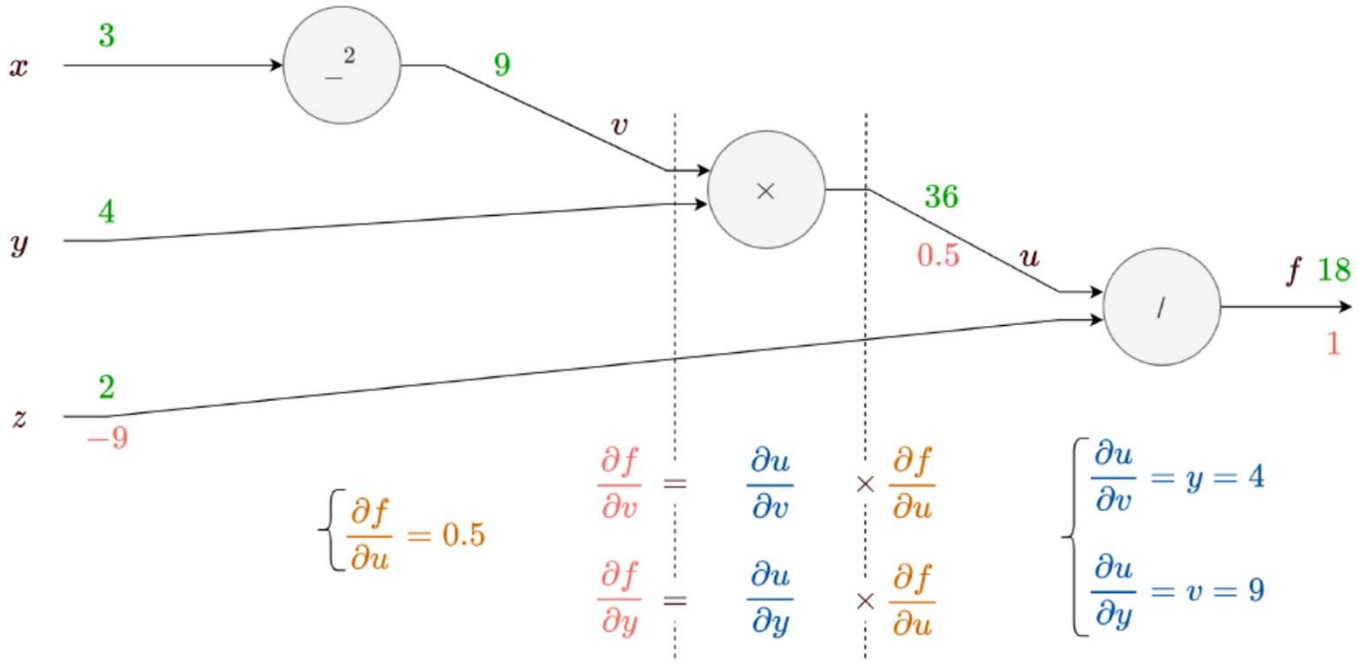


Backward Propagation : example



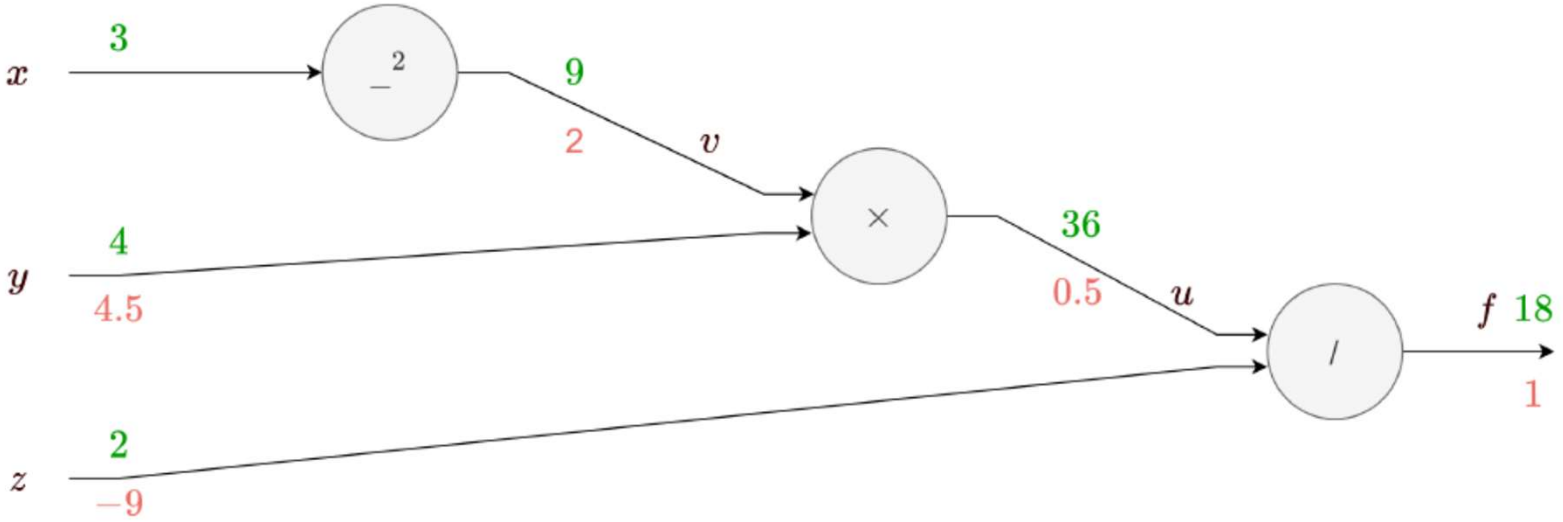
Backward propagation : example

Backpropagation for $\times$ module:



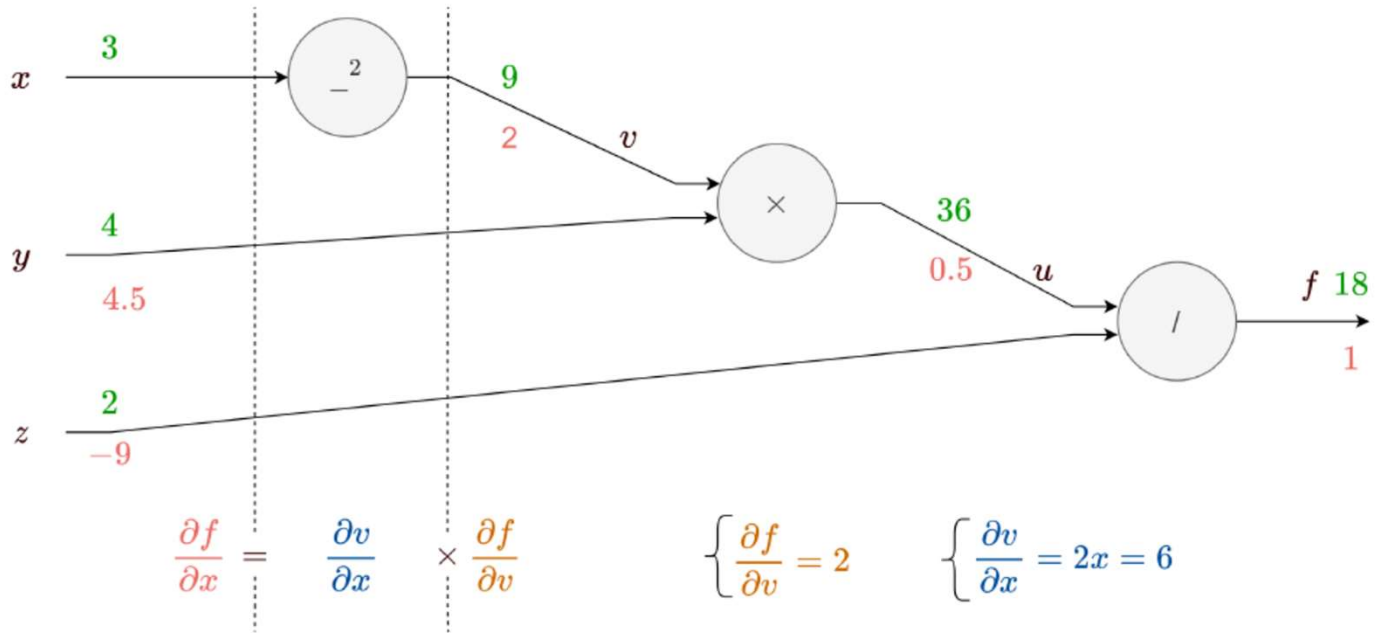
Backward propagation : example

Backpropagation for $\times$ module:



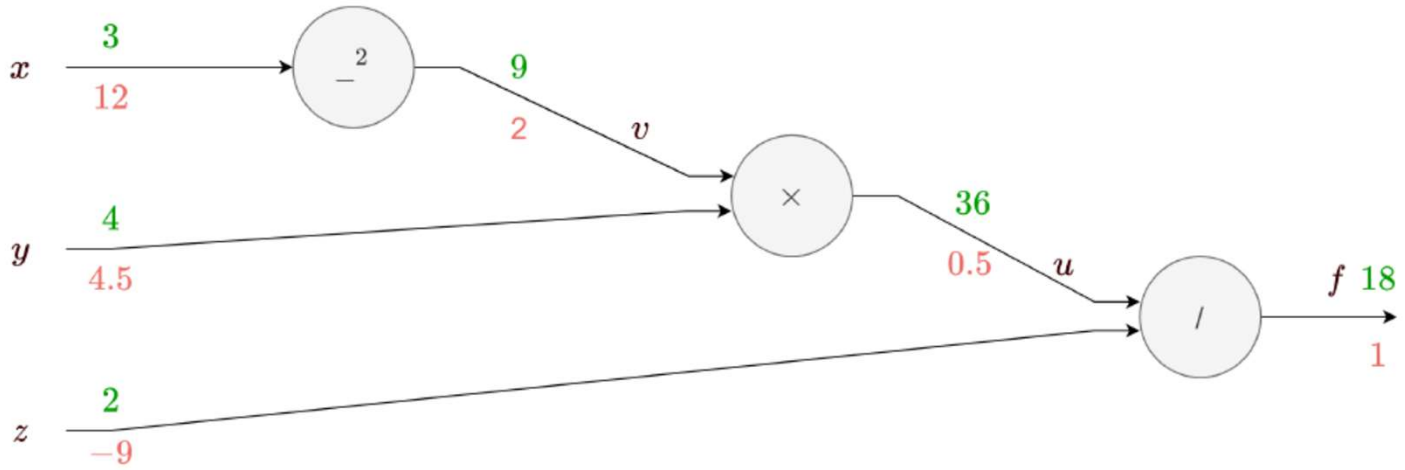
Backward propagation : example

Backpropagation for $\_^2$ module:



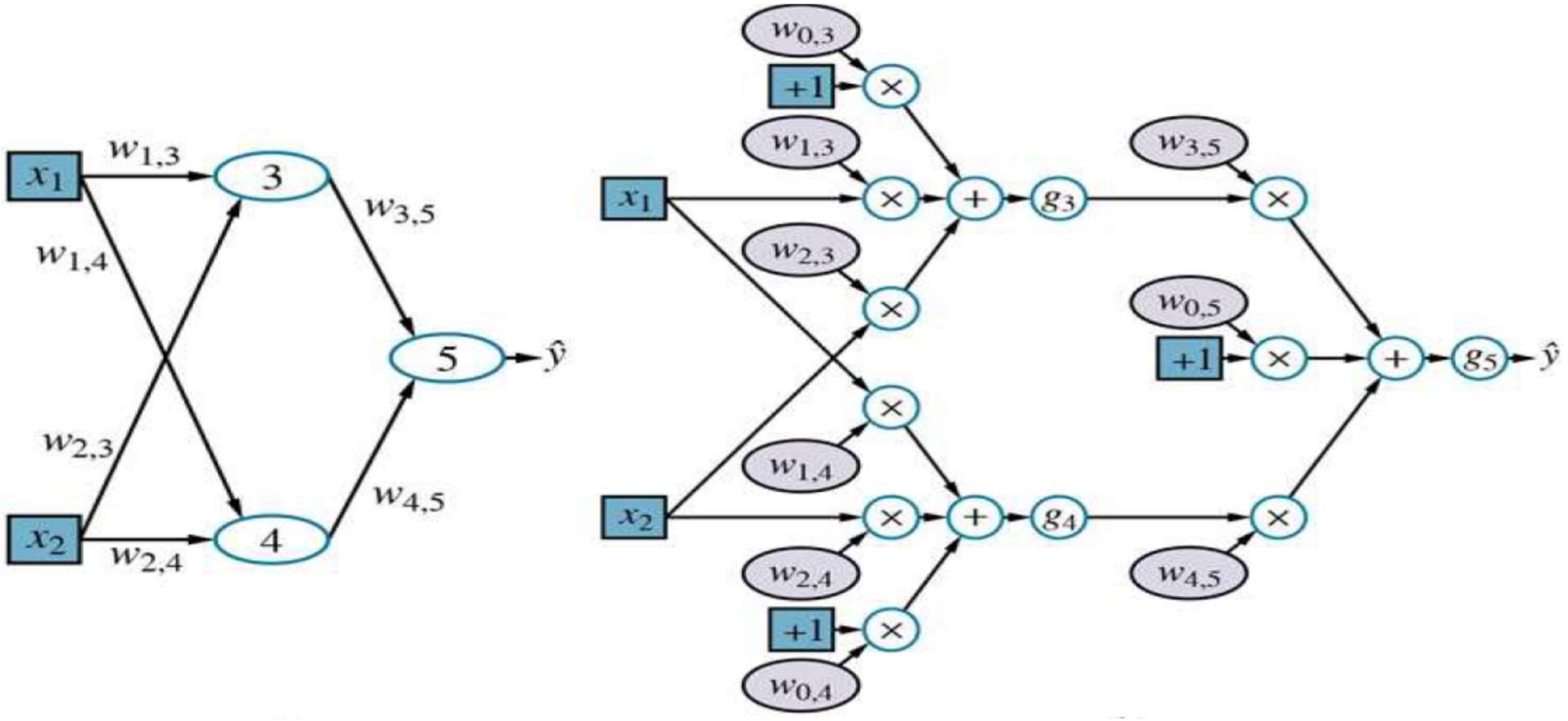
Backward propagation : example

Backpropagation for $\_^2$ module:



Results are the same as analytical results.

Computation Graph



Backward Propagation

So after backward propagation we will have:

- Gradient of loss with respect to each parameter.
- We can apply gradient descent to update parameters.

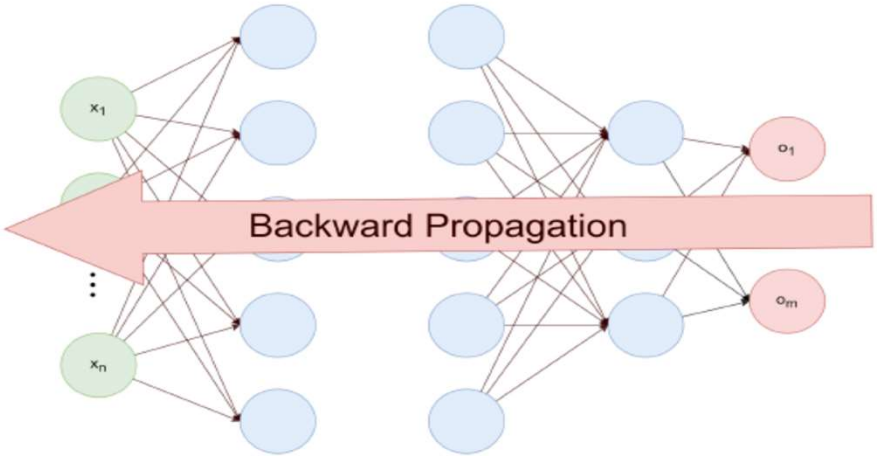


Figure: Backward pass

Various GD type

So far you got familiar with gradient-based optimization.

If $\boldsymbol{g} = \nabla_{\boldsymbol{\theta}} \mathcal{J}$, then we will update parameters with this simple rule:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \boldsymbol{g}$$

But there is one question here, how to compute $\boldsymbol{g}$?

Based on how we calculate $\boldsymbol{g}$ we will have different types of gradient descent:

- Batch Gradient Descent
- Stochastic Gradient Descent
- Mini-Batch Gradient Descent

Various GD types: Batch Gradient Descent

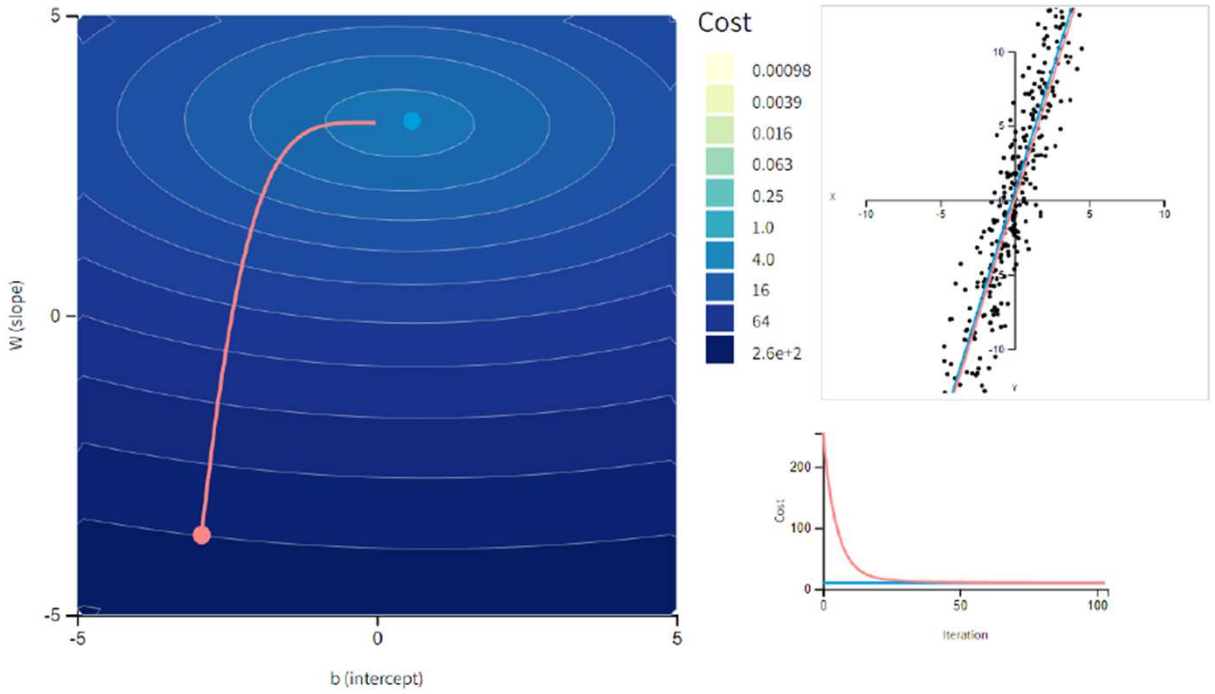
- In this type we use entire training set to calculate gradient.

Batch Gradient:

$$\boldsymbol{g} = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} \mathcal{L}(d_i, \boldsymbol{\theta})$$

- Using this method with very large training set:
  - Your data can be too large to process in your memory.
  - It requires a lot of processing to compute gradient for all samples.
- Using exact gradient may lead us to local minima.
- Moving noisy may help us get out of this local minimas.

Various GD types: Batch Gradient Descent



Various GD types: Stochastic Gradient Descent

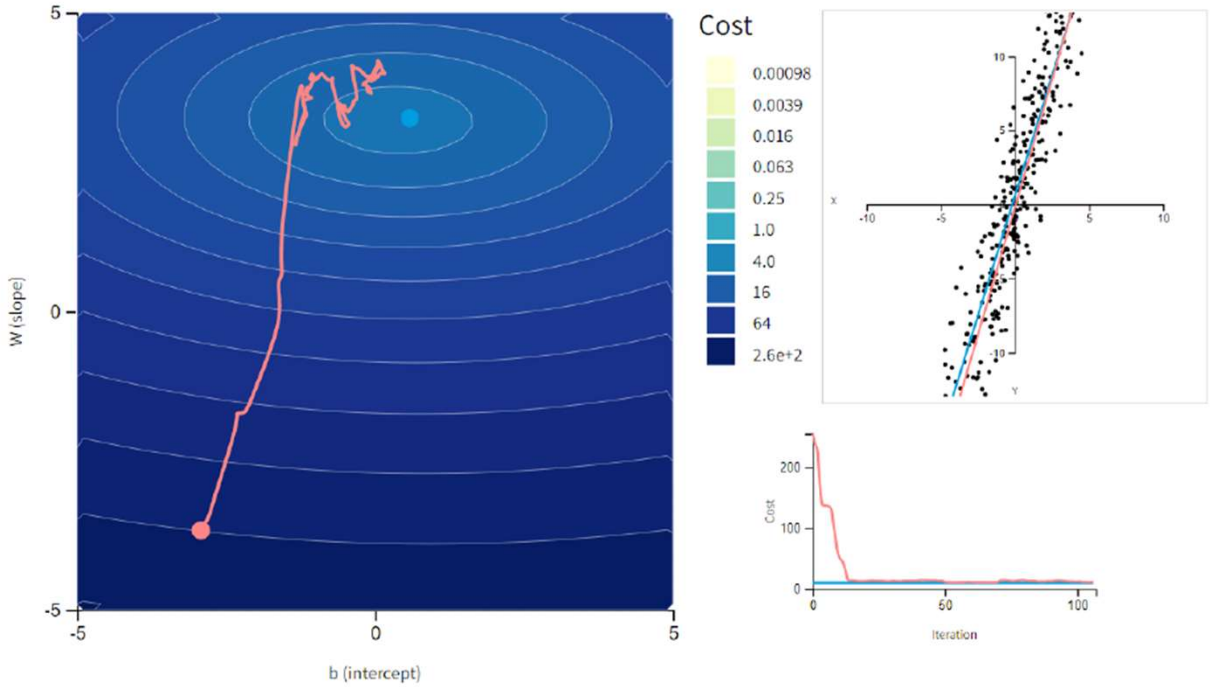
- Instead of calculating exact gradient, we can estimate it using our data.
- This is exactly what SGD does, it estimates gradient using only single data point.

Stochastic Gradient:

$$\hat{\boldsymbol{g}} = \nabla_{\boldsymbol{\theta}} \mathcal{L}(d_i, \boldsymbol{\theta})$$

- As we use an approximation of gradient, instead of gently decreasing, the cost function will bounce up and down and decrease only on average.
- This method is really computationally efficient cause we only need to calculate gradient for one point per iteration.

Various GD types: Stochastic Gradient Descent



Various GD types: MiniBatch Gradient Descent

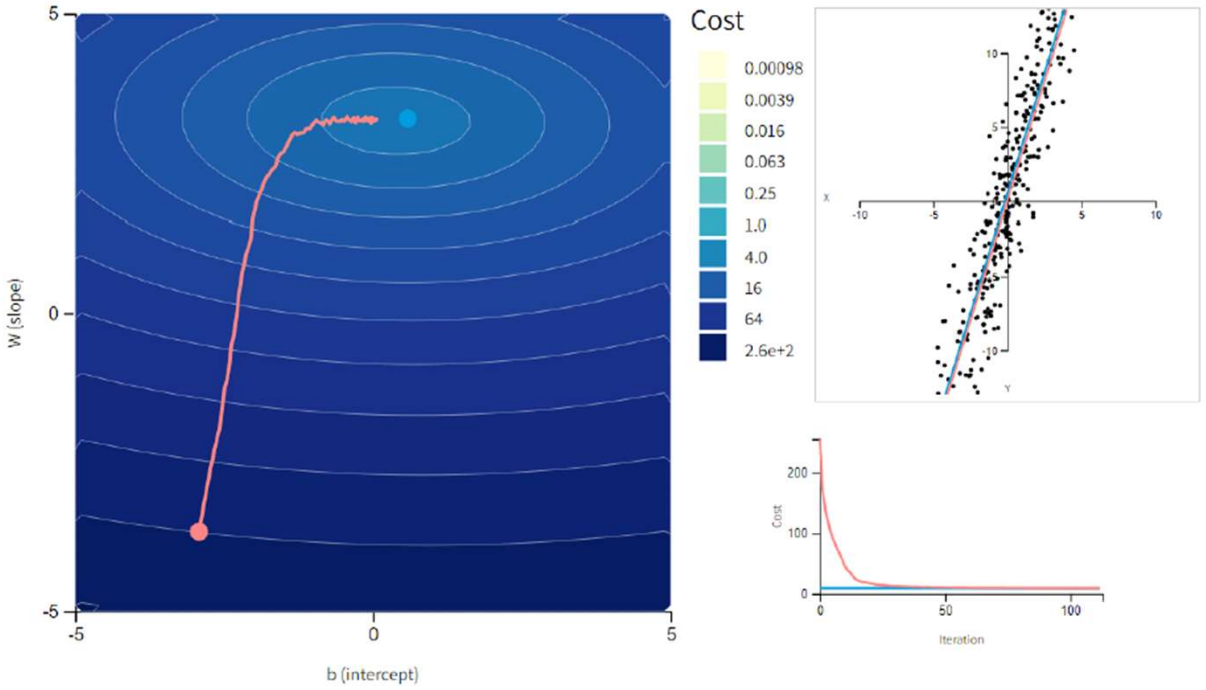
- In this method we still use estimation idea But use a batch of data instead of one point.

Mini-Batch Gradient:

$$\hat{g} = \frac{1}{|\mathcal{B}|} \sum_{d \in \mathcal{B}} \nabla_{\theta} \mathcal{L}(d, \theta), \quad \mathcal{B} \subset \mathcal{D}$$

- This is a better estimation than SGD.
- With this way we can get a performance boost from hardware optimization, especially when using GPUs.
- Batch size ($|\mathcal{B}|$) is a hyperparameter you need to tune.

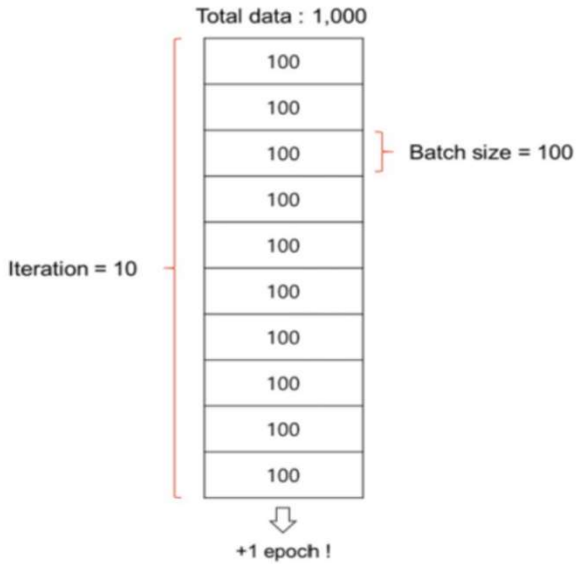
Various GD types: MiniBatch Gradient Descent



Various GD types

Now that we know what a batch is, we can define epoch and iteration:

- One Epoch is when an entire dataset is passed forward and backward through the network only once.
- One Iteration is when a batch is passed forward and backward through the network.



Loss functions

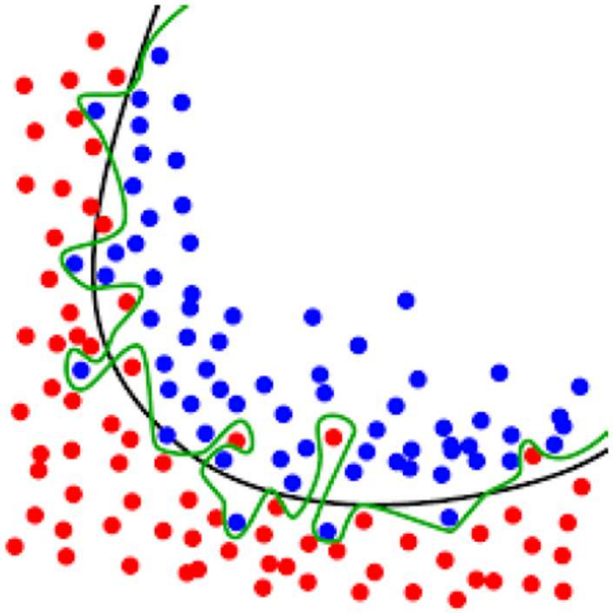
$$H(p,q) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{k} y_j^{(i)} \log(p(y_j^{(i)}))$$

$$\mathrm{MSE}(y,\hat{y}) = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

$$\mathrm{MAE}(y,\hat{y}) = \frac{1}{m}\sum_{i=1}^{m}|(y_i - \hat{y}_i)|$$

Problem: OverFitting in a Neural Network

- Why does overfitting happen in a neural network?
  - There are Too many free parameters.



Solution 1: L1/L2 Regularization

- Sum the regularizer term for every layer weight!

$$L = \frac{1}{N} \sum_{i=1}^{N} L(\phi(x_i), y_i) + \lambda \sum_{i,j,k} R(W_{j,k}^{(i)})$$

- L1/L2 regularizer functions (review)

$$L1 : R(w) = |w|$$

$$L2 : R(w) = w^2$$

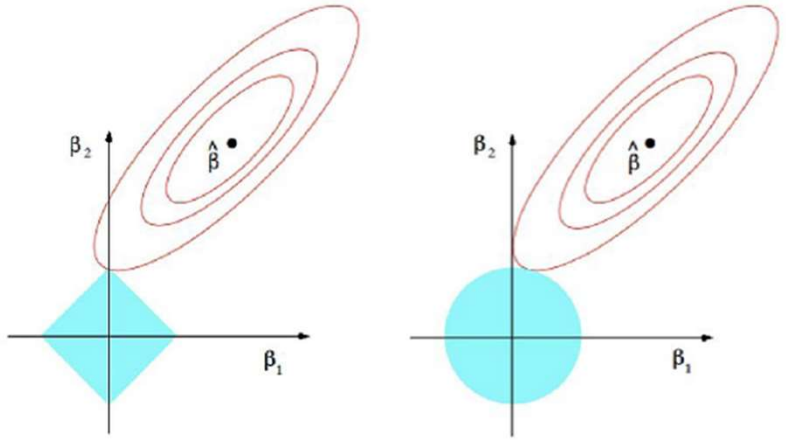


Figure: L1/L2 regularizers' solution diagram, Source

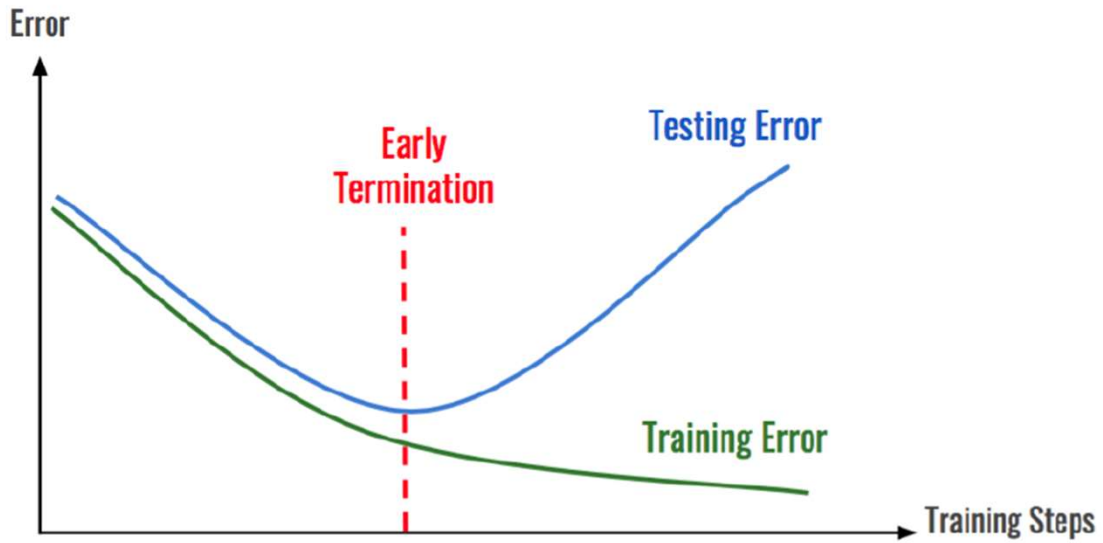
- You can also combine the two different regularizers (Elastic Net).

$$R(w) = \beta w^2 + |w|$$

- What is advantage of the L2 norm versus L1 norm ?

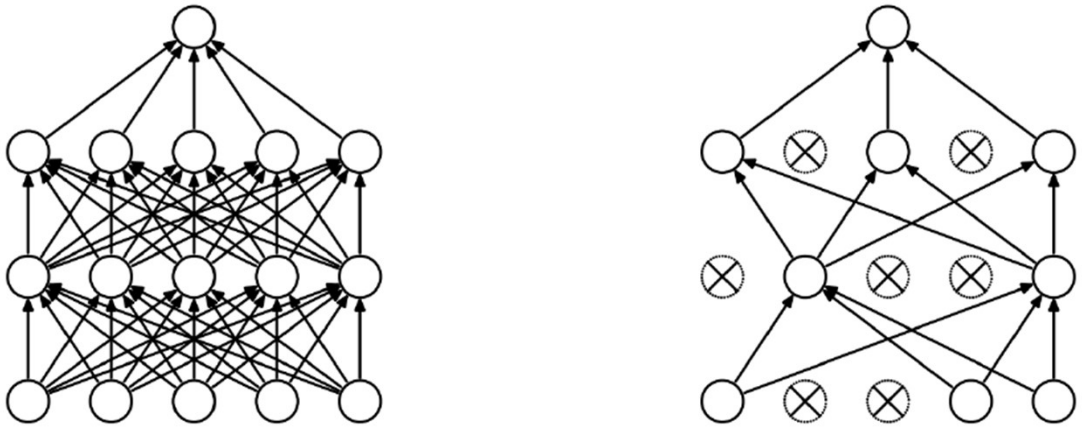
Solution 2: Early Stopping

- Stop the training procedure when the validation error is minimum.



Dropout: Training Time

- In each forward pass, randomly set some neurons to zero.
- The probability of dropping out for each neuron, which is called dropout rate, is a hyperparameter.
  - 0.5 is a common dropout rate.
- The probability of not dropping out is also called the keep probability.

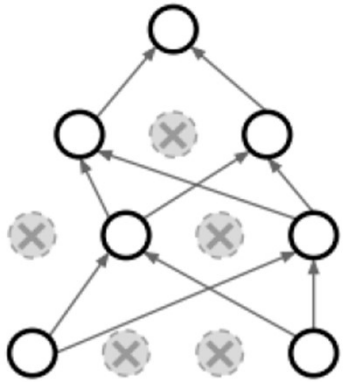


Dropout: Why can this possibly be a good idea?

- Dropout-trained neurons are unable to co-adapt with their surrounding neurons.
- They also can't depend too heavily on a small number of input neurons.
- They become less responsive to even little input changes.
- The result is a stronger network that generalizes better.

Dropout: Why can this possibly be a good idea?

- Dropout trains a large ensemble of models that share parameters.
- Every possible dropout state for neurons of a network, which is called a mask, is one model.



Dropout: Test Time

- Dropout makes our output random at training time.

$$y = f_W(x, \underbrace{z}_{\text{random mask}})$$

- We want to average out the randomness at test time,

$$y = f(x) = E_z[f(x, z)] = \int p(z) f(x, z) dz$$

- But this integral seems complicated.
- Let's approximate the integral for a superficial layer where dropout rate is 0.5.

Dropout: Test Time

$$E_{train}[a] = \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y)$$
$$+\frac{1}{4}(0x + w_2 y) + \frac{1}{4}(0x + 0y)$$
$$= \frac{1}{2}(w_1 x + w_2 y)$$
$$E_{test}[a] = w_1 x + w_2 y$$
$$\Rightarrow E_{train}[a] = \underbrace{0.5}_{\text{keep probability}} \quad E_{test}[a]$$

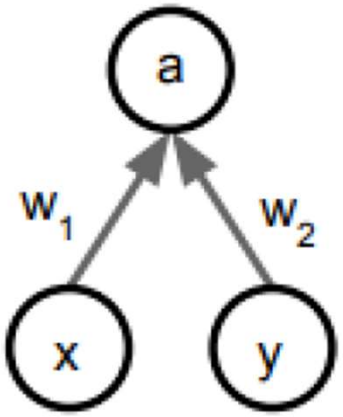


Figure: Simple neural network. [6]

If a unit is retained with probability p during training, the outgoing weights of that unit are multiplied by p at test time