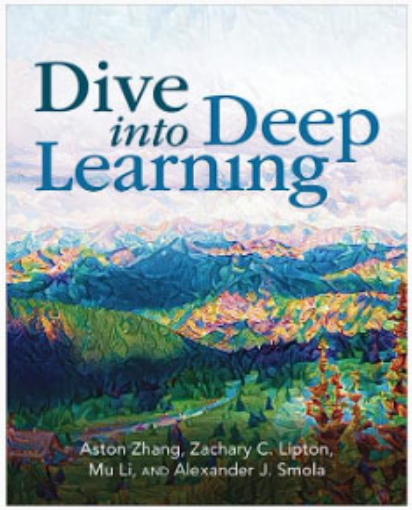
Neural Networks

Fatemeh Mansoori

University of Isfahan



Dive into Deep Learning

Interactive deep learning book with code, math, and discussions

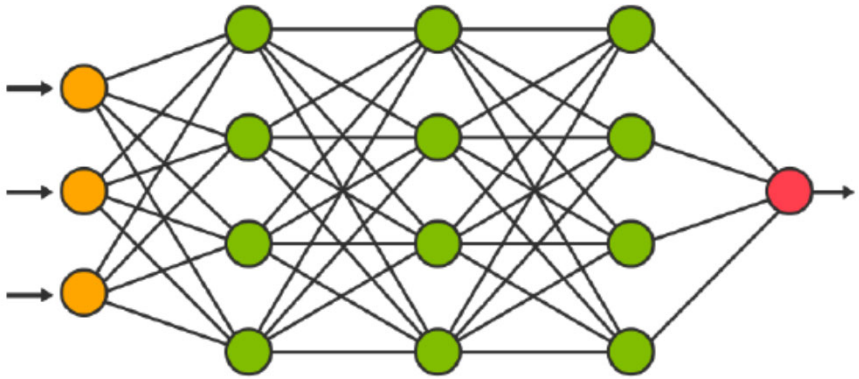
Implemented with **PyTorch**, **NumPy/MXNet**, **JAX**, and **TensorFlow**

Adopted at 500 universities from 70 countries

☆ Star 21,711

https://d2l.ai/index.html

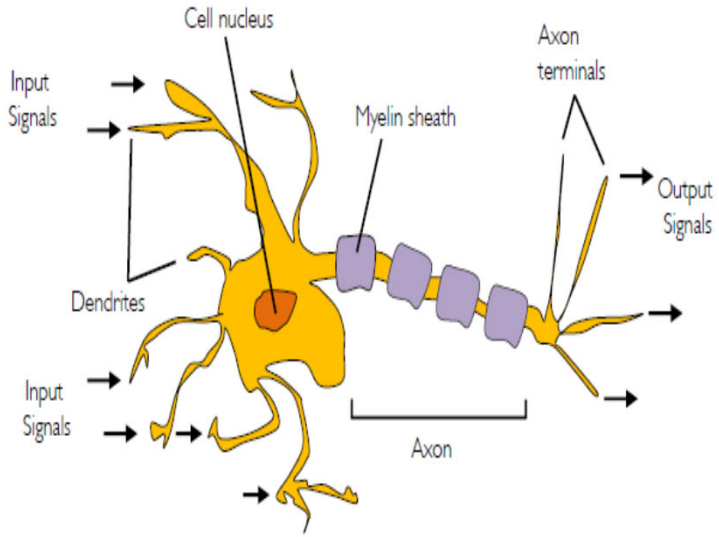
Introduction to Neural Network



This slides are created based on the slide of the machine learning course at sharif university, Deap learning course Andeow Ng, Deap learning course by Sebastian rashka

Biological Analogy

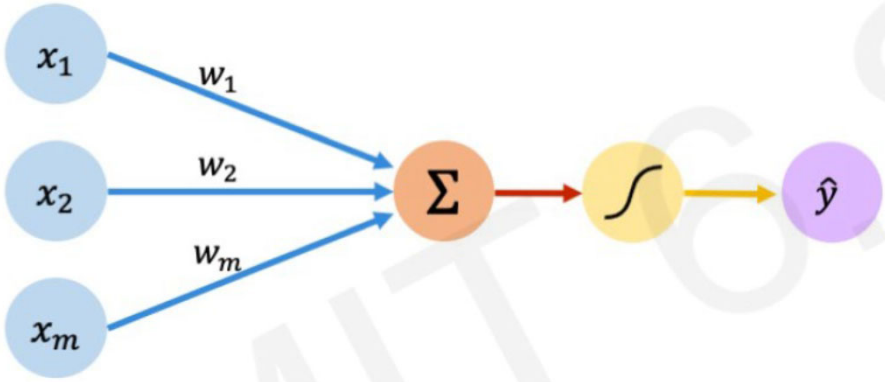
- A brain is a set of densely connected neurons
- Depending on the input signals, the neuron performs computations and decides to fire or not.



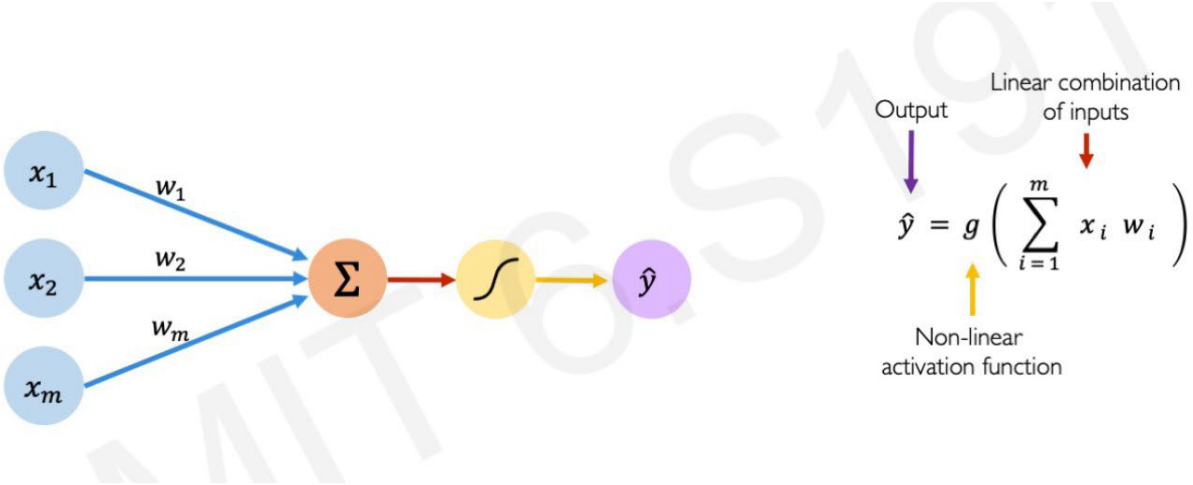
Mathematical formulation of a biological neuron,
could solve AND, OR, NOT problems

A simple mathematical model of neuron

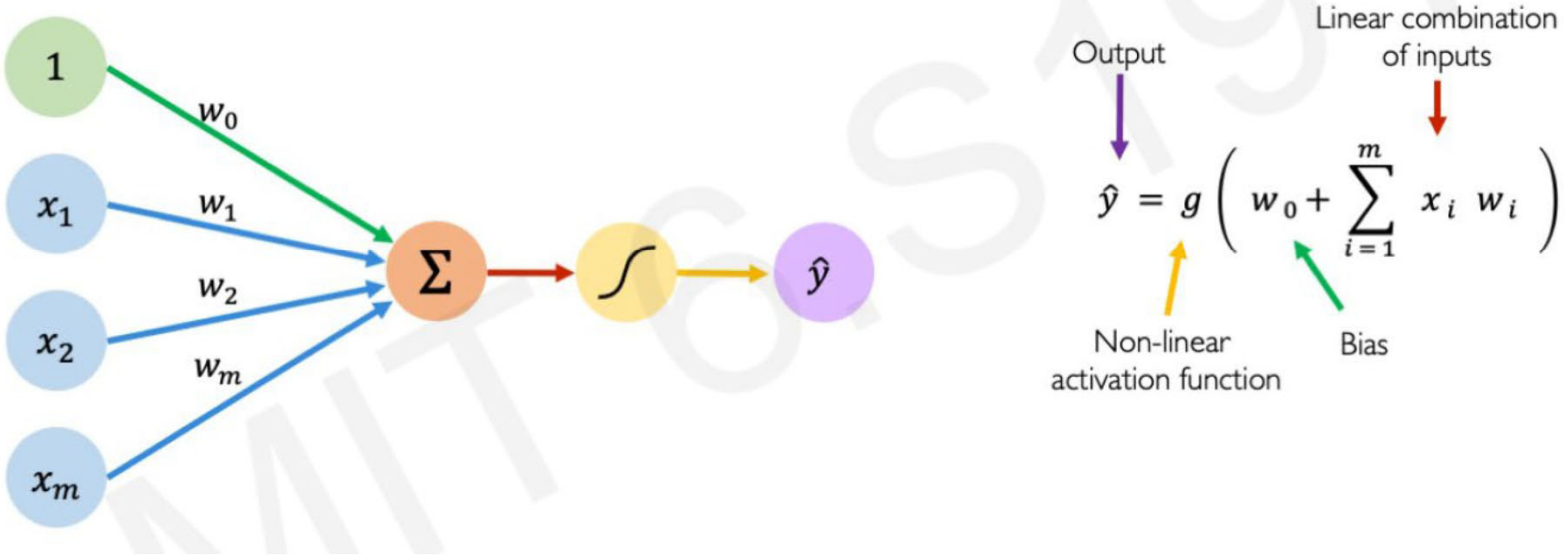
- McCulloch & Pitt's neuron model (1943)
- A linear classifier __ it fires when a linear combinations of its input exceeds some threshold.



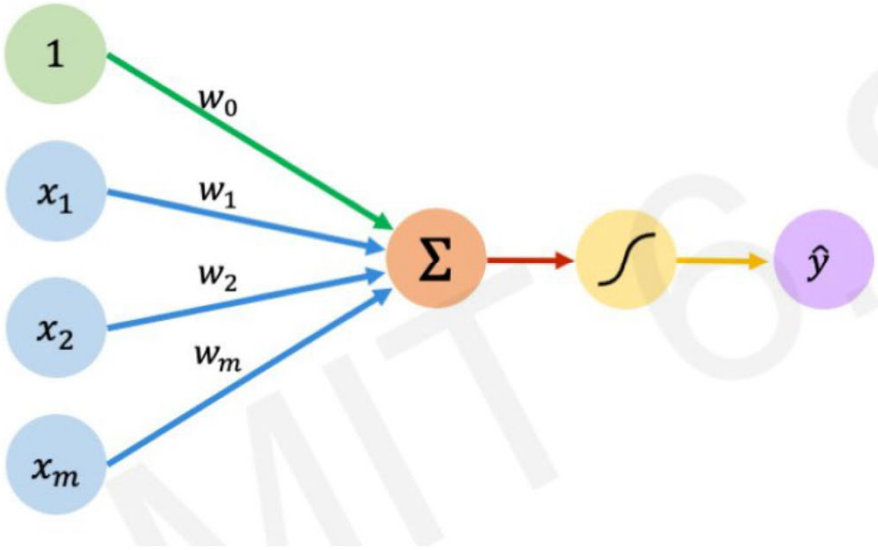
A simple mathematical model of a neuron



A simple mathematical model of a neuron



A simple mathematical model of a neuron



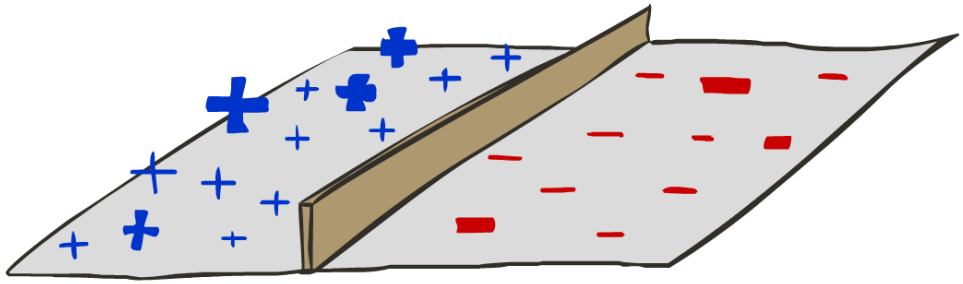
$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i\, w_i\right)$$

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

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

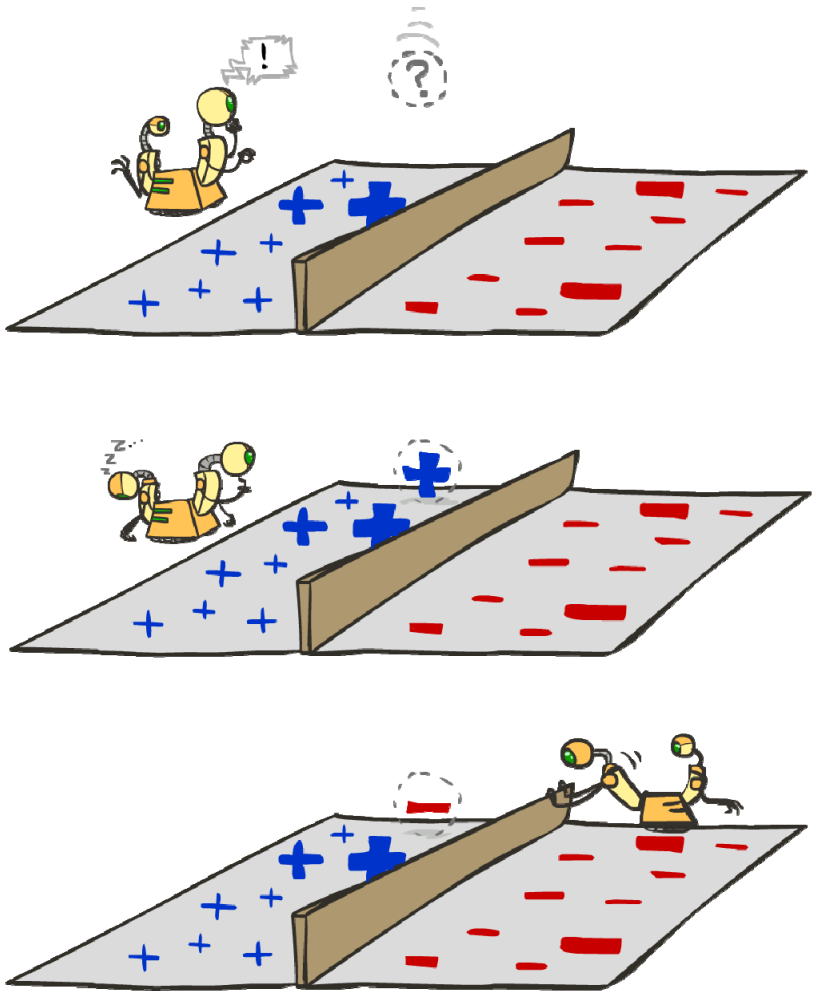
Binary Decision Rule

- 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



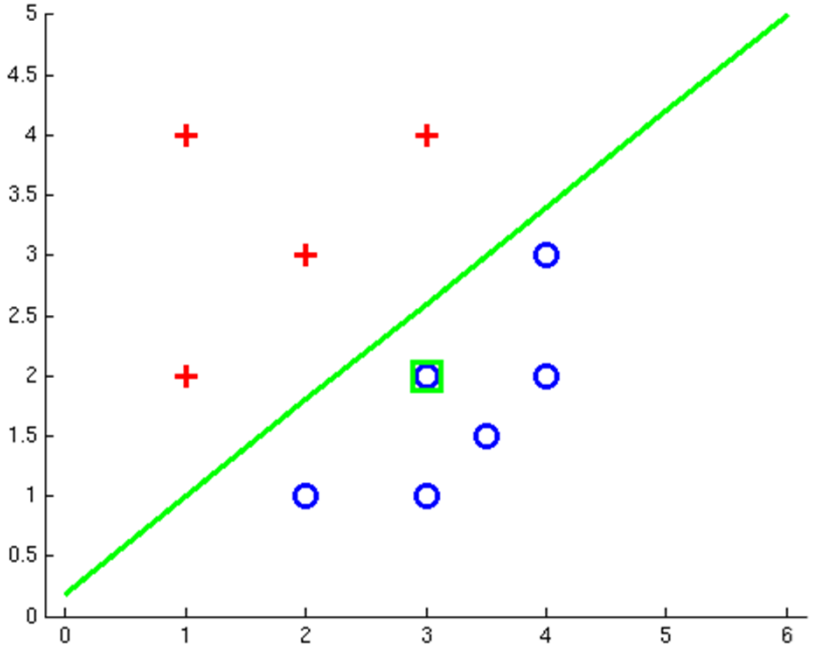
Learning

- 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

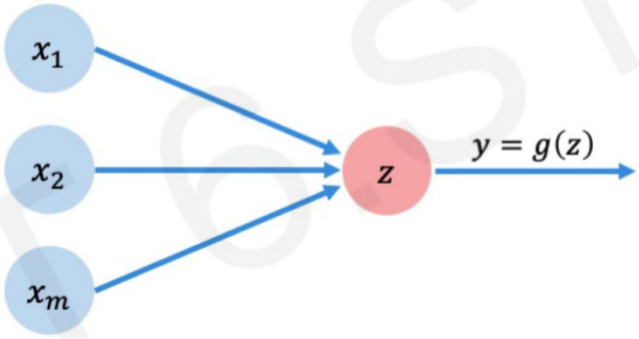


Examples: Perceptron

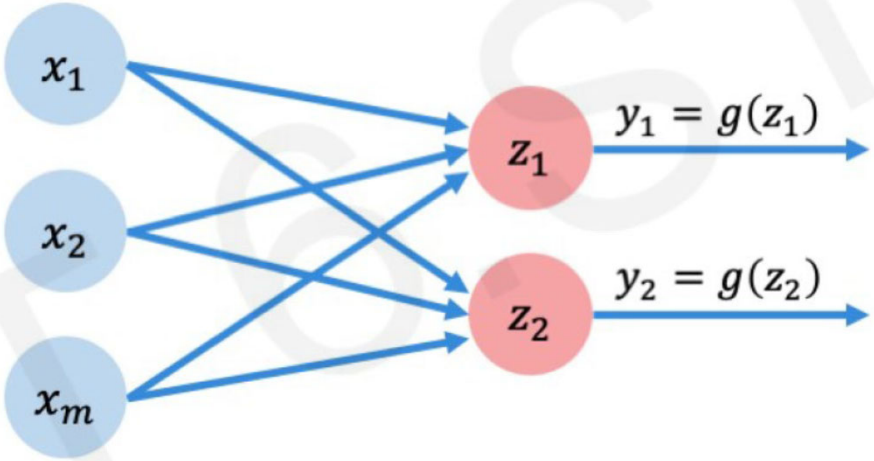
- Separable Case



Multi Output Perceptron



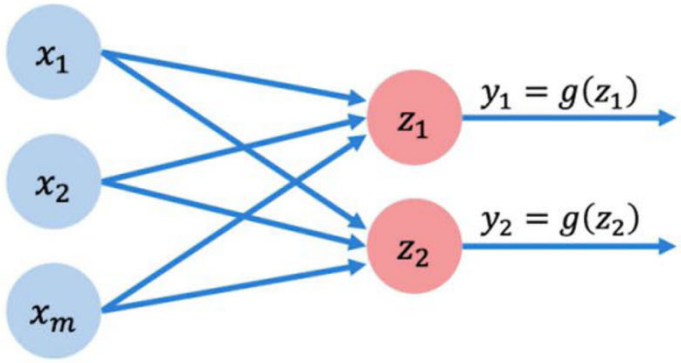
$$z = w_0 + \sum_{j=1}^{m} x_j w_j$$



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

Multi Output perceptron

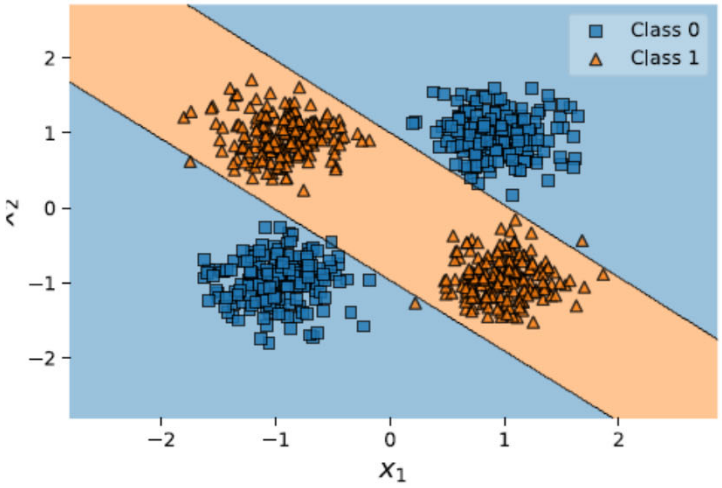
Because all inputs are densely connected to all outputs, these layers are called **Dense** layers

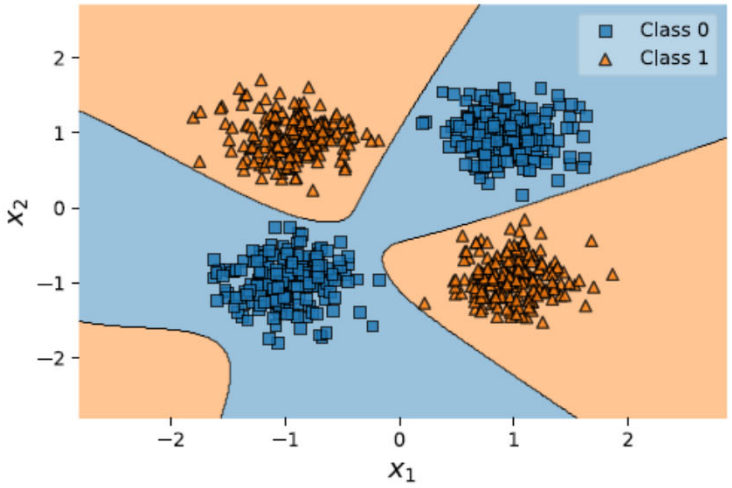


Limitation of perceptron

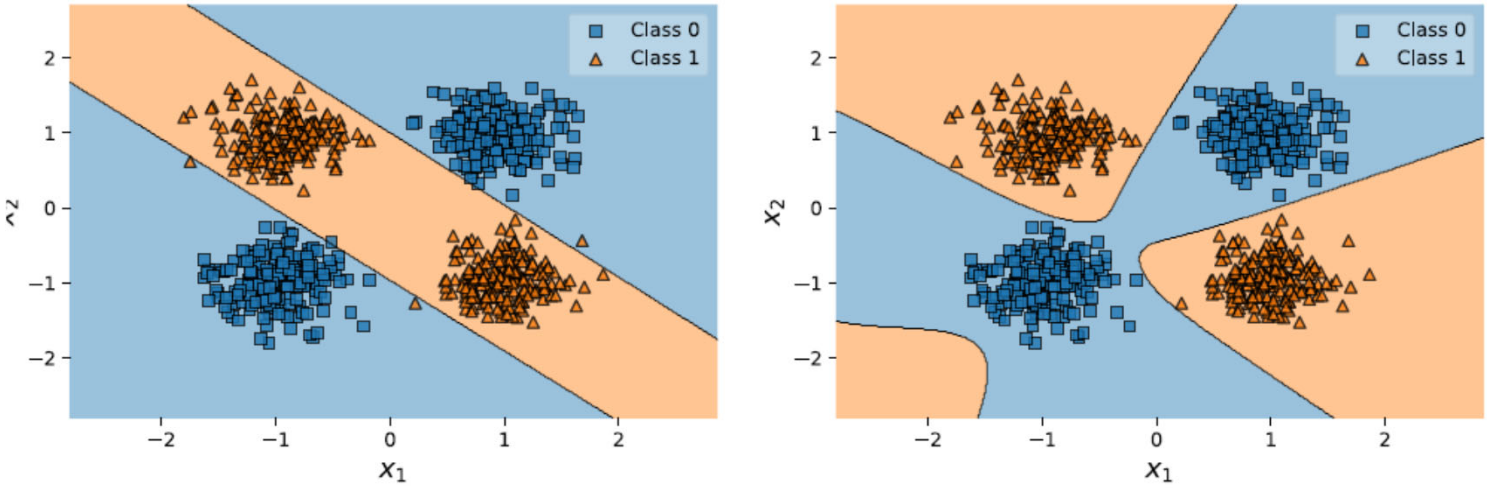
- XOR can not be represented by perceptron
- We need a deeper network
- No one knew how to train deeper networks

Why can’t a perceptron represent XOR?



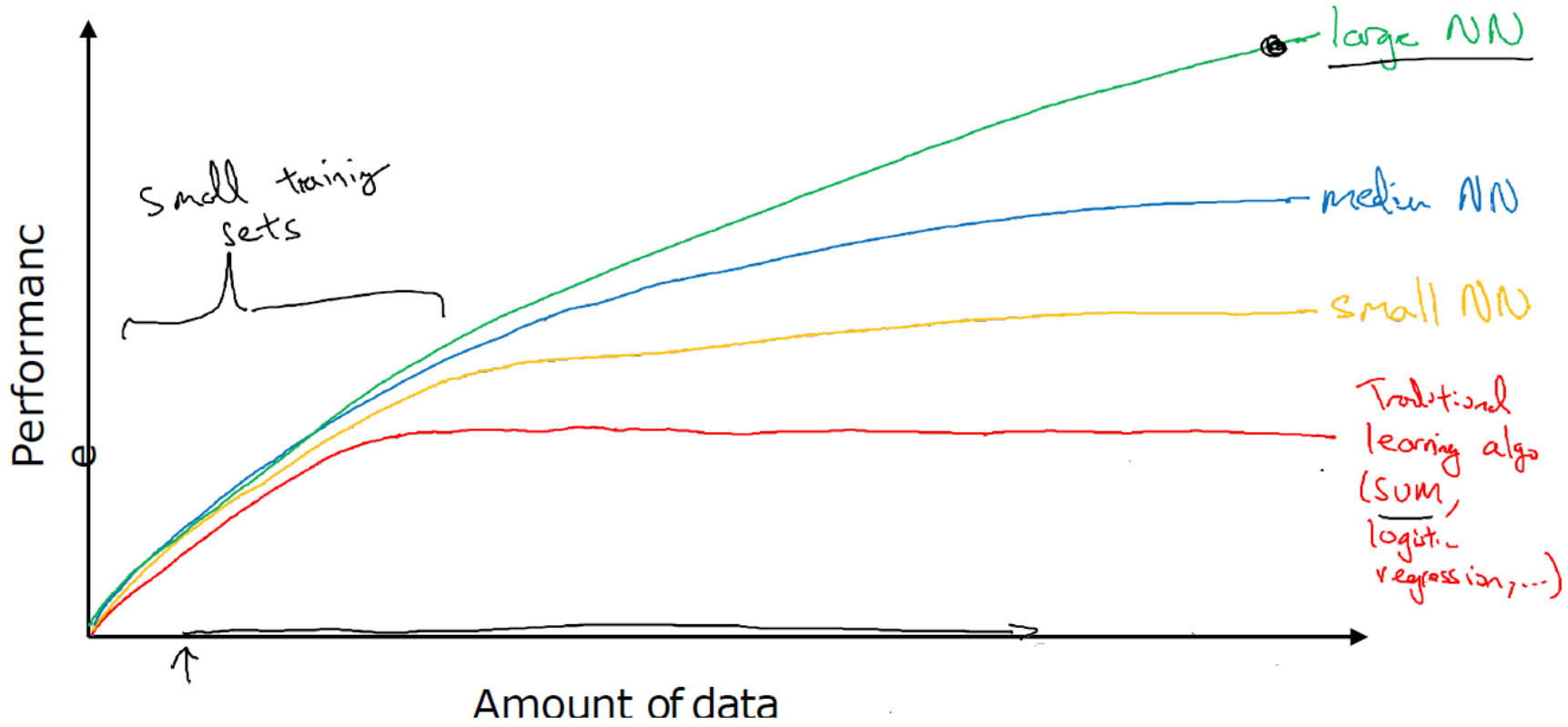


Multilayer Neural Networks Can Solve XOR Problems

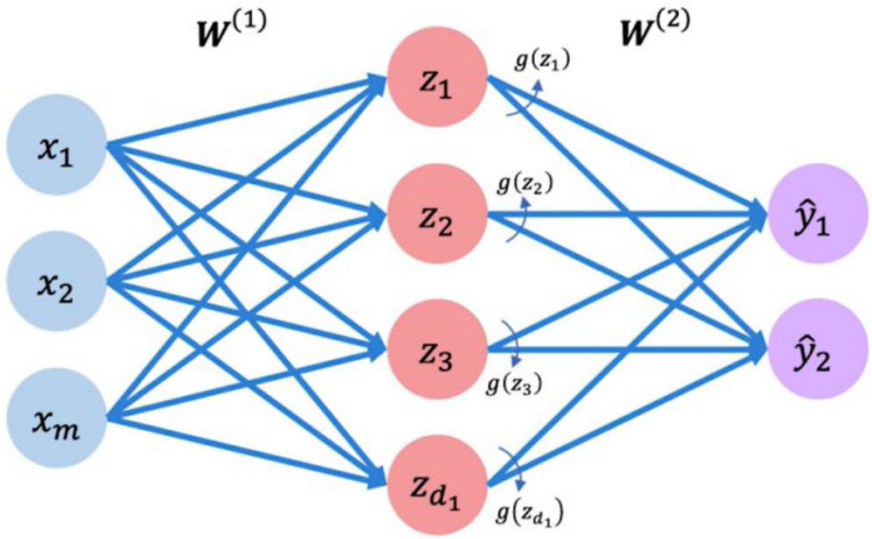


- Decision boundaries of two different multilayer perceptions on simulated data solving the XOR problem

Scale drives deep learning progress



Neural Network with 1 Hidden Layer

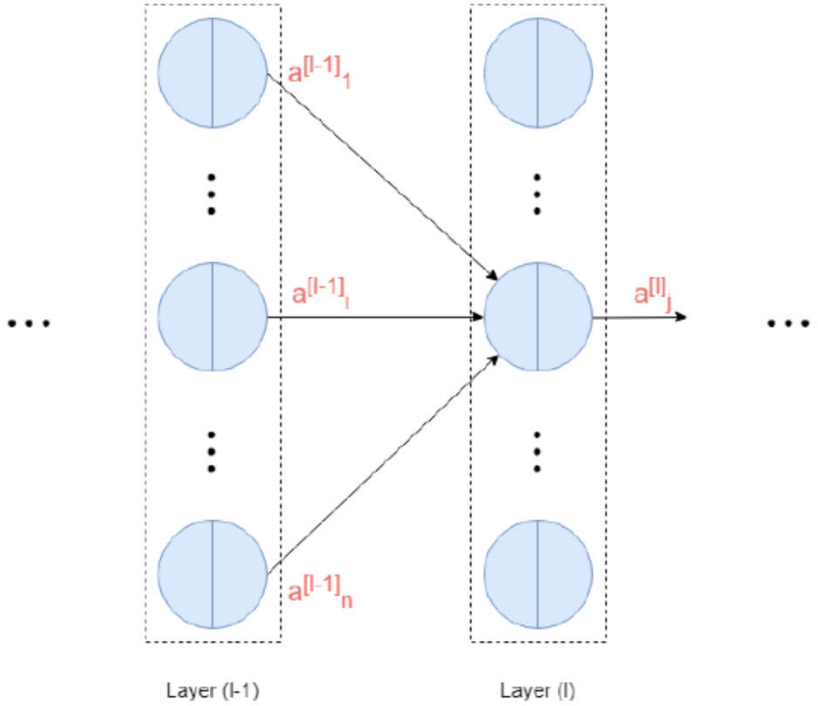


$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} g(z_j) \, w_{j,i}^{(2)}\right)$$

Standard notations for MLP

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

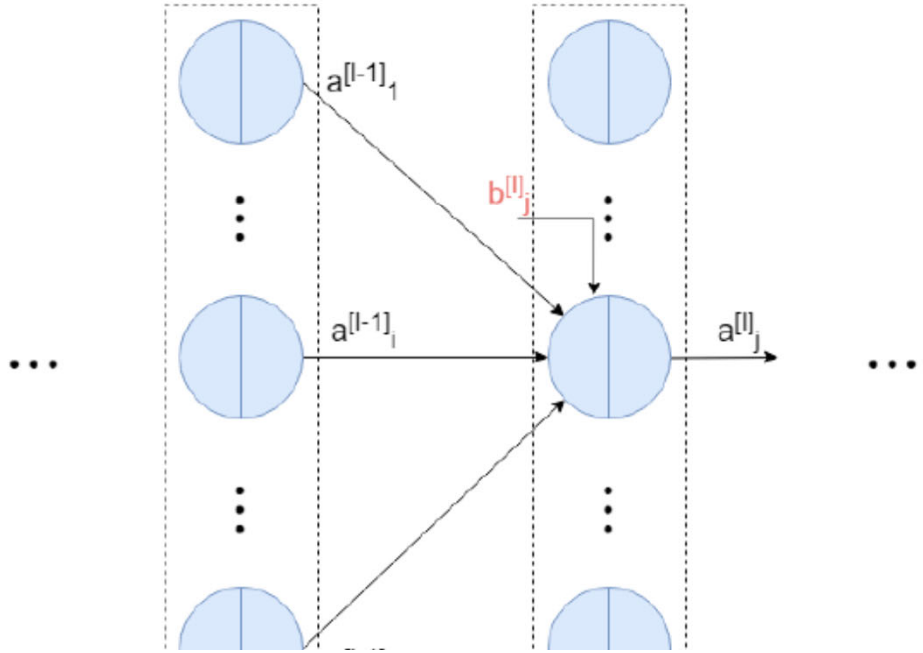
$\boldsymbol{a}^{[l]}$: layer l output in vector form



MLP notation

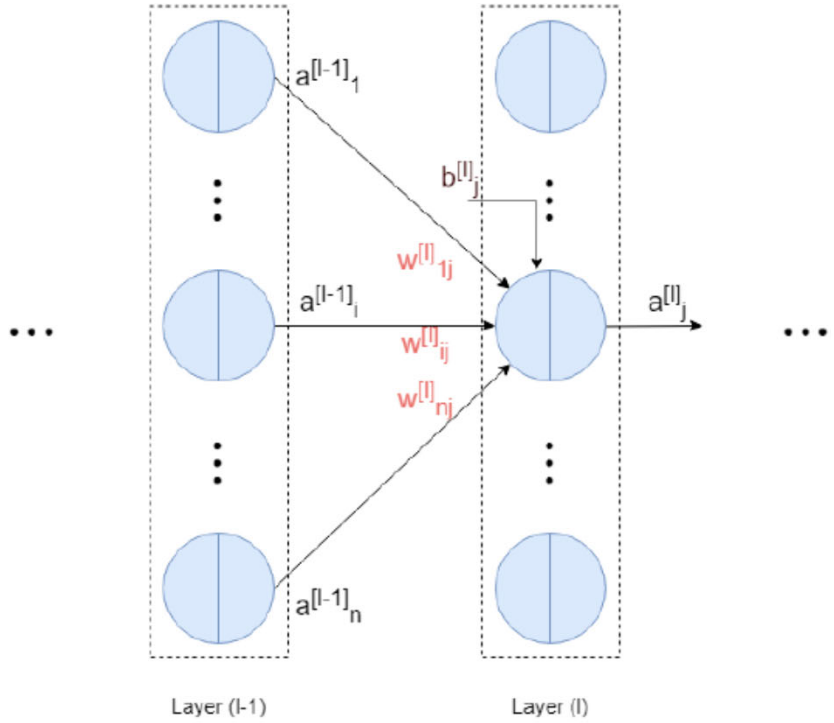
$b_i^{[l]}$: i-th neuron bias in layer l

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



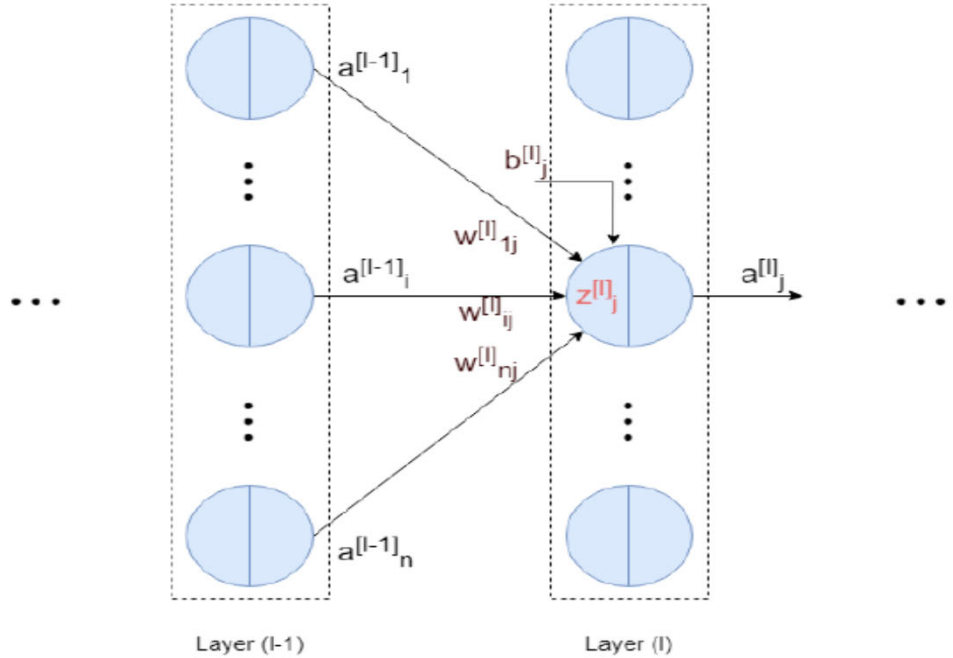
MLP notation

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



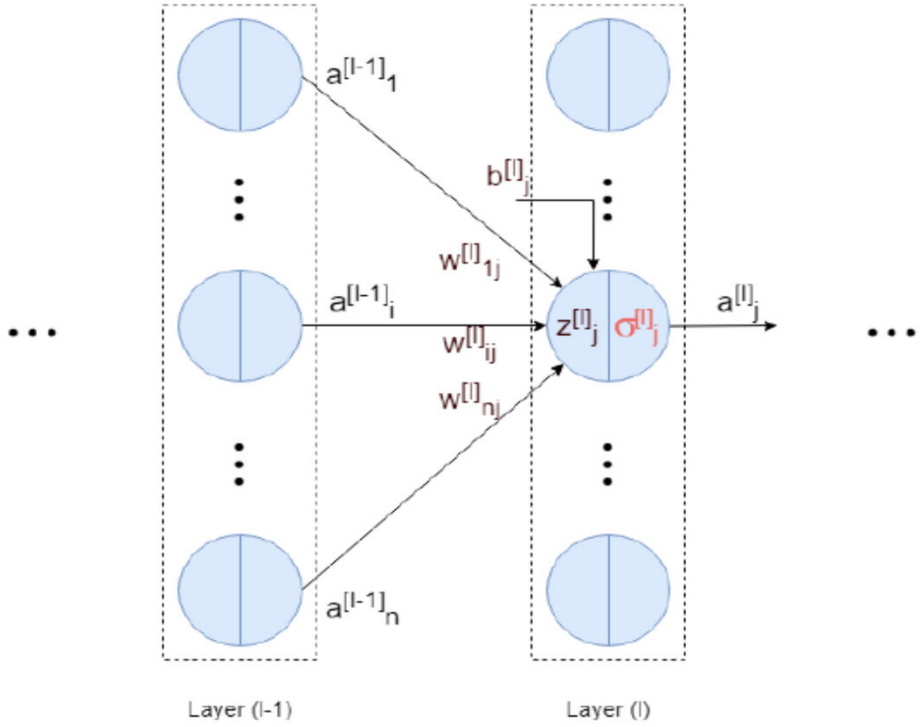
MLP notation

- $z_j^{[l]}$: 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

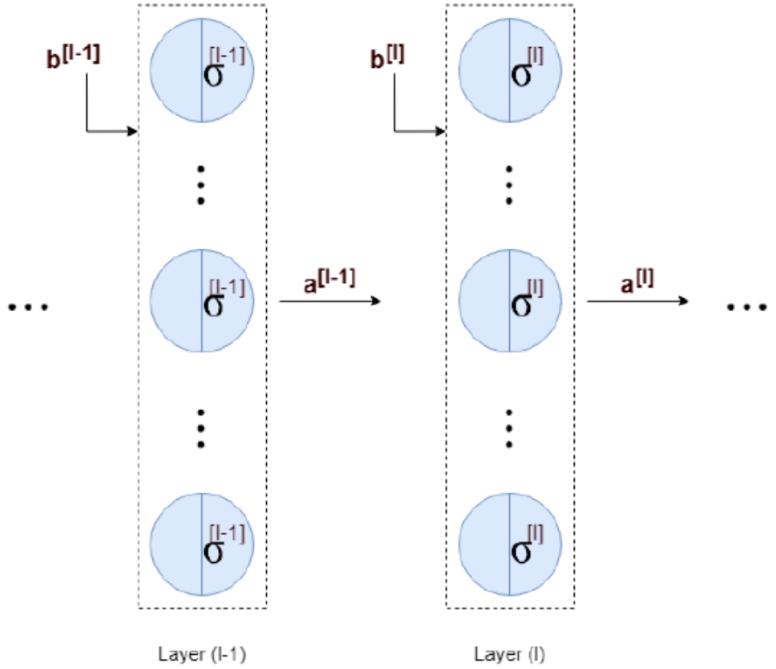
$\sigma_j^{[l]}$: j-th neuron activation function in layer l



MLP notation

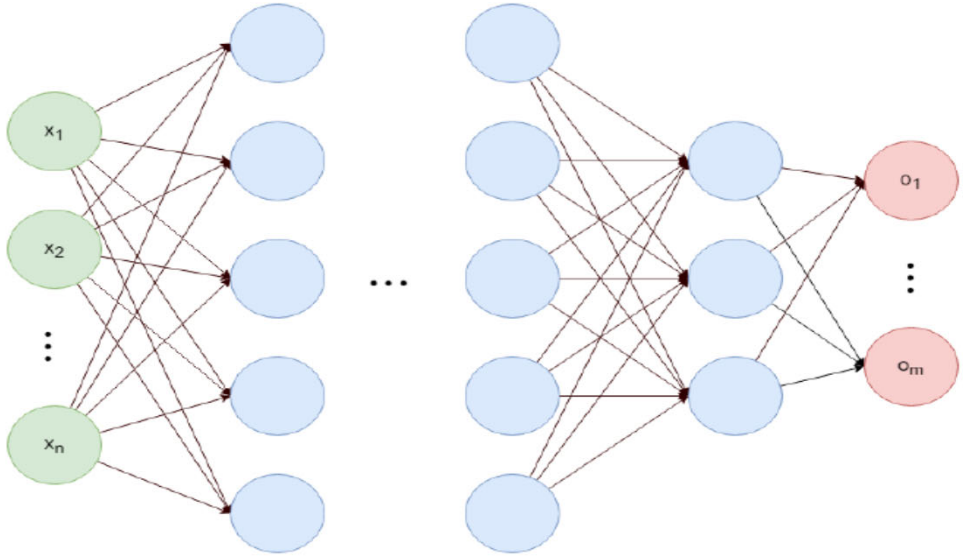
- If all neurons in one layer have the same activation function then :

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



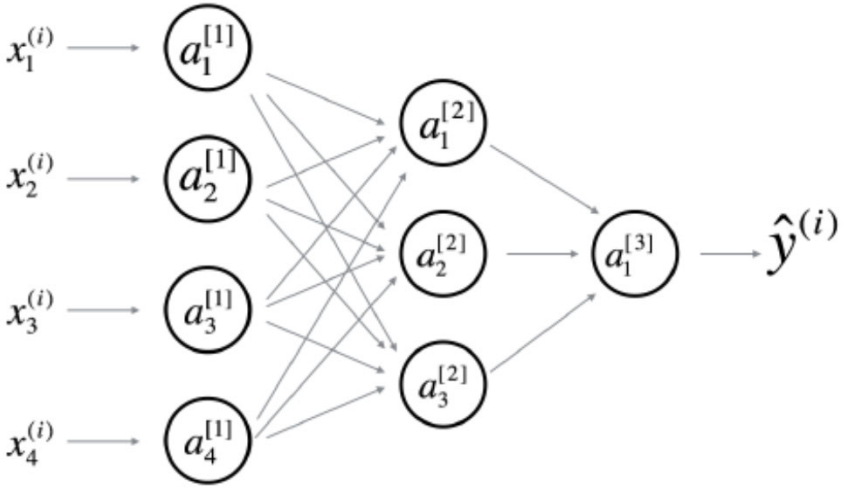
MLP notation

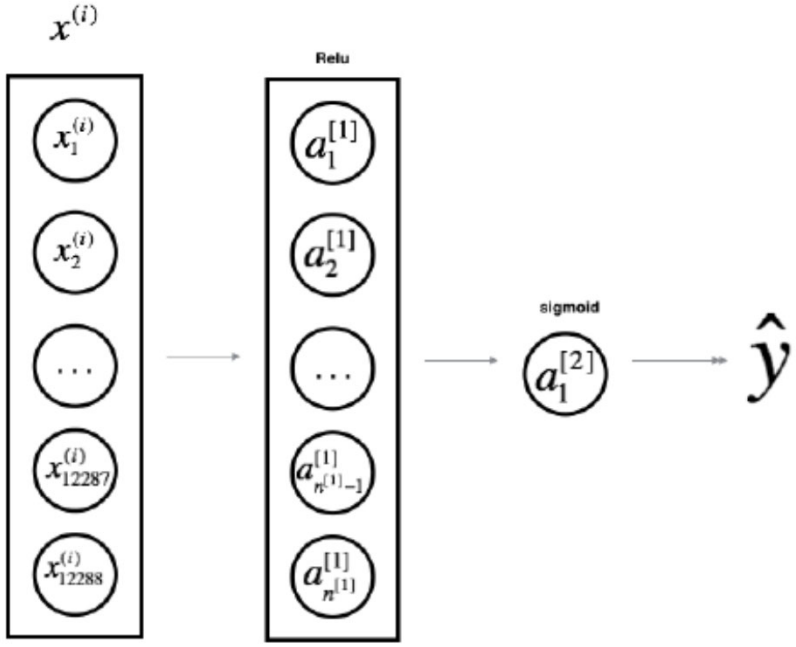
- For a network with L layer and x as its input we have :



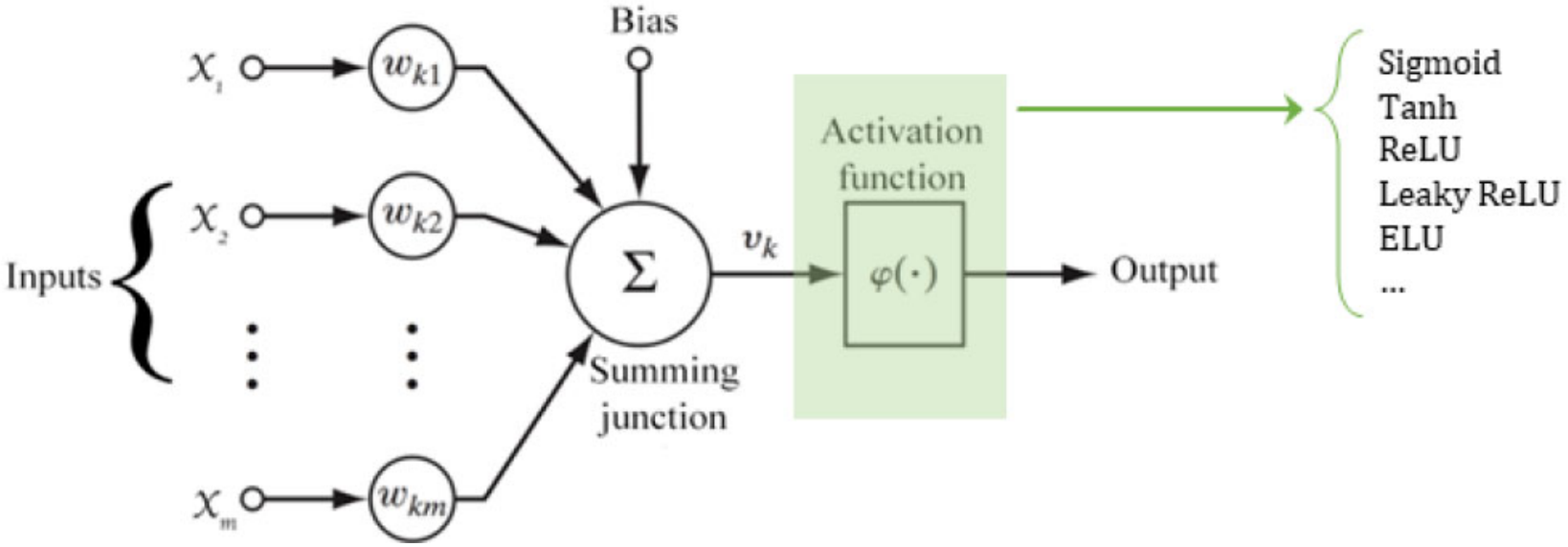
$$\boldsymbol{o} = \boldsymbol{a}^{[L]} = \sigma^{[L]}\left(\boldsymbol{b}^{[L]} + (W^{[L]})^T \sigma^{[L-1]}\left(\cdots \sigma^{[1]}\left(\boldsymbol{b}^{[1]} + (W^{[1]})^T \boldsymbol{x}\right)\cdots\right)\right)$$

Deep Learning representations



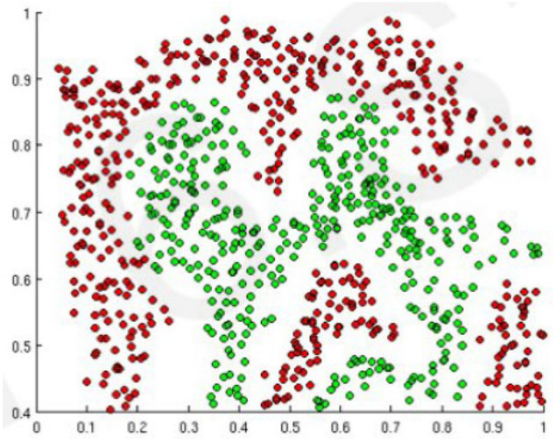


Activation function



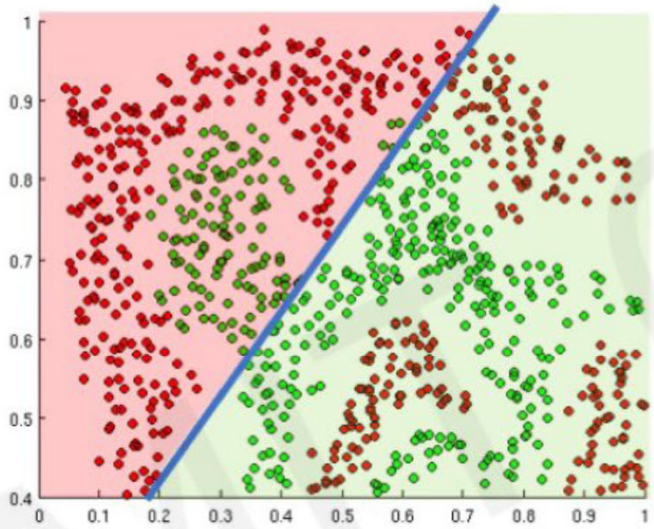
Importance of Activation Function

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

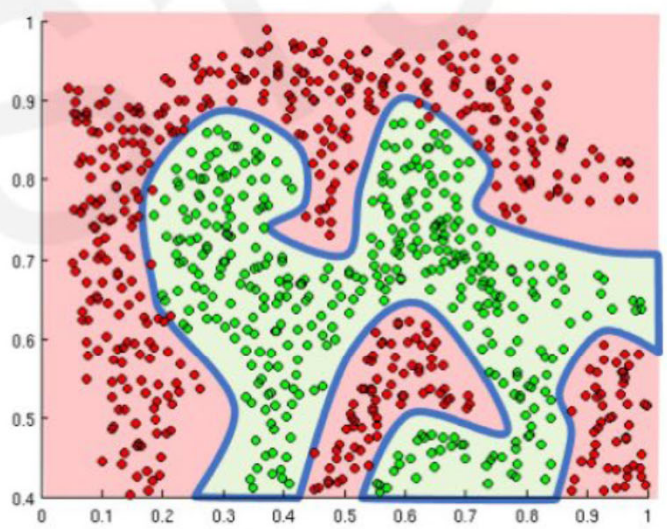


- What if we wanted to build a neural network to distinguish green vs red points?

Importance of Activation Function



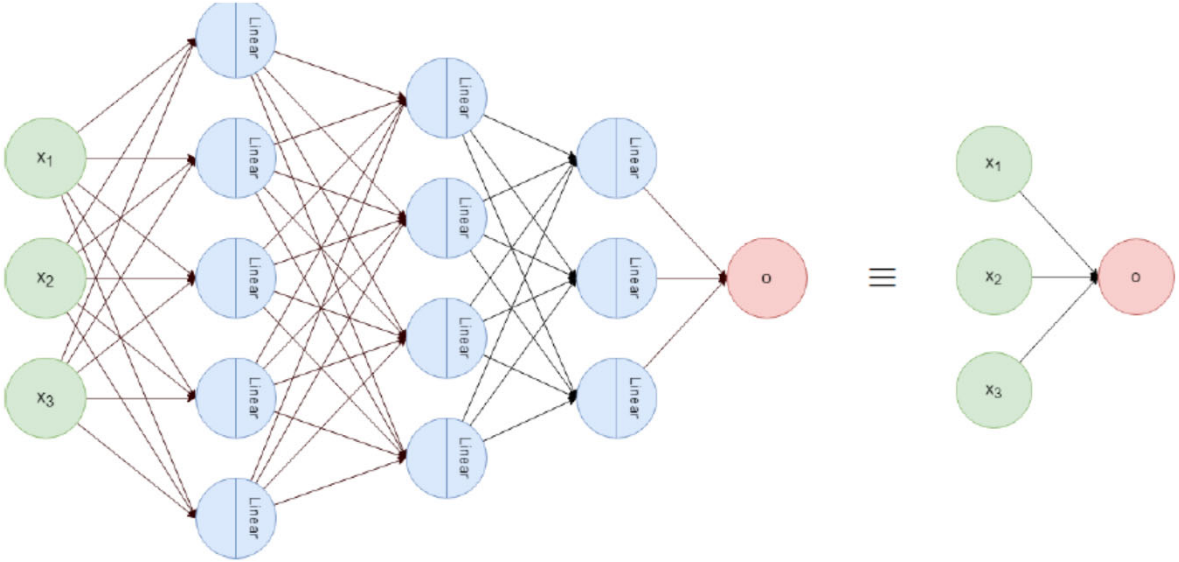
Linear activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

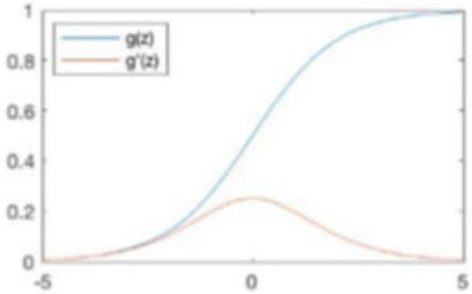
Activation Function of Hidden Layers

- 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

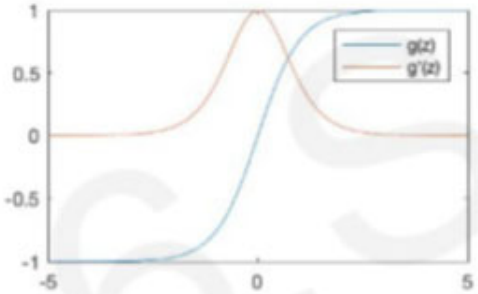
Sigmoid Function



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

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

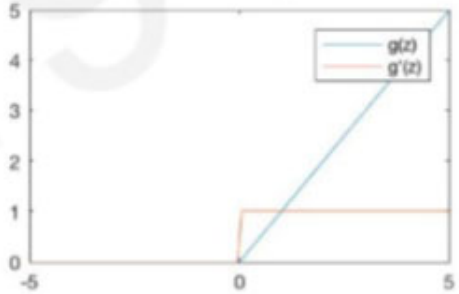
Hyperbolic Tangent



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

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

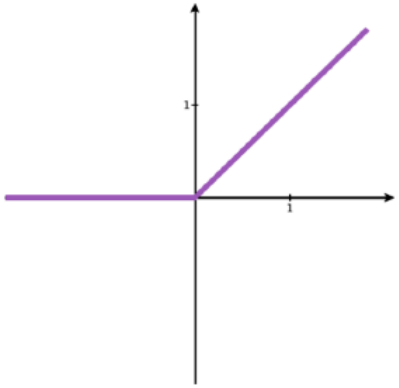
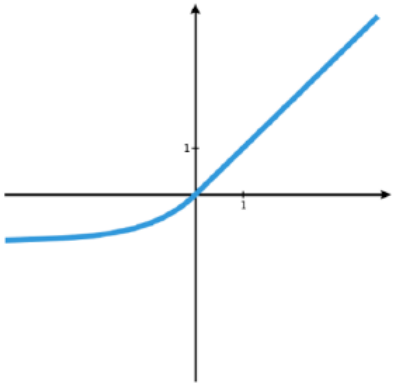
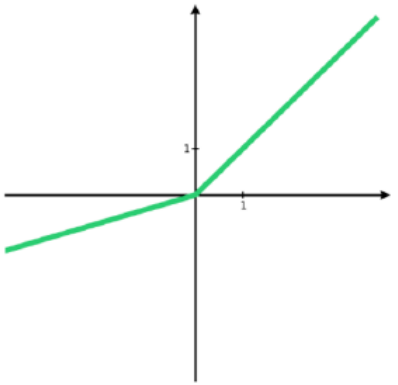
Rectified Linear Unit (ReLU)



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

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

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**.

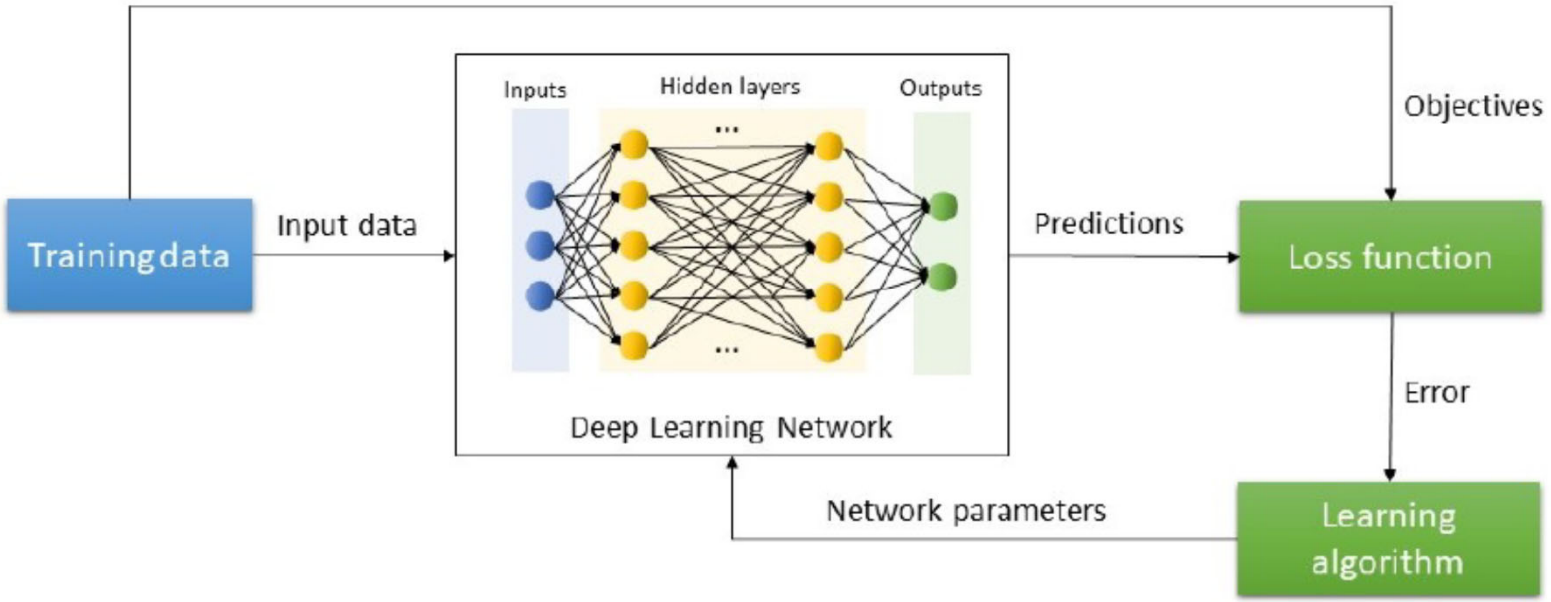
- **z** is the vector of raw outputs from the neural network

- 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.

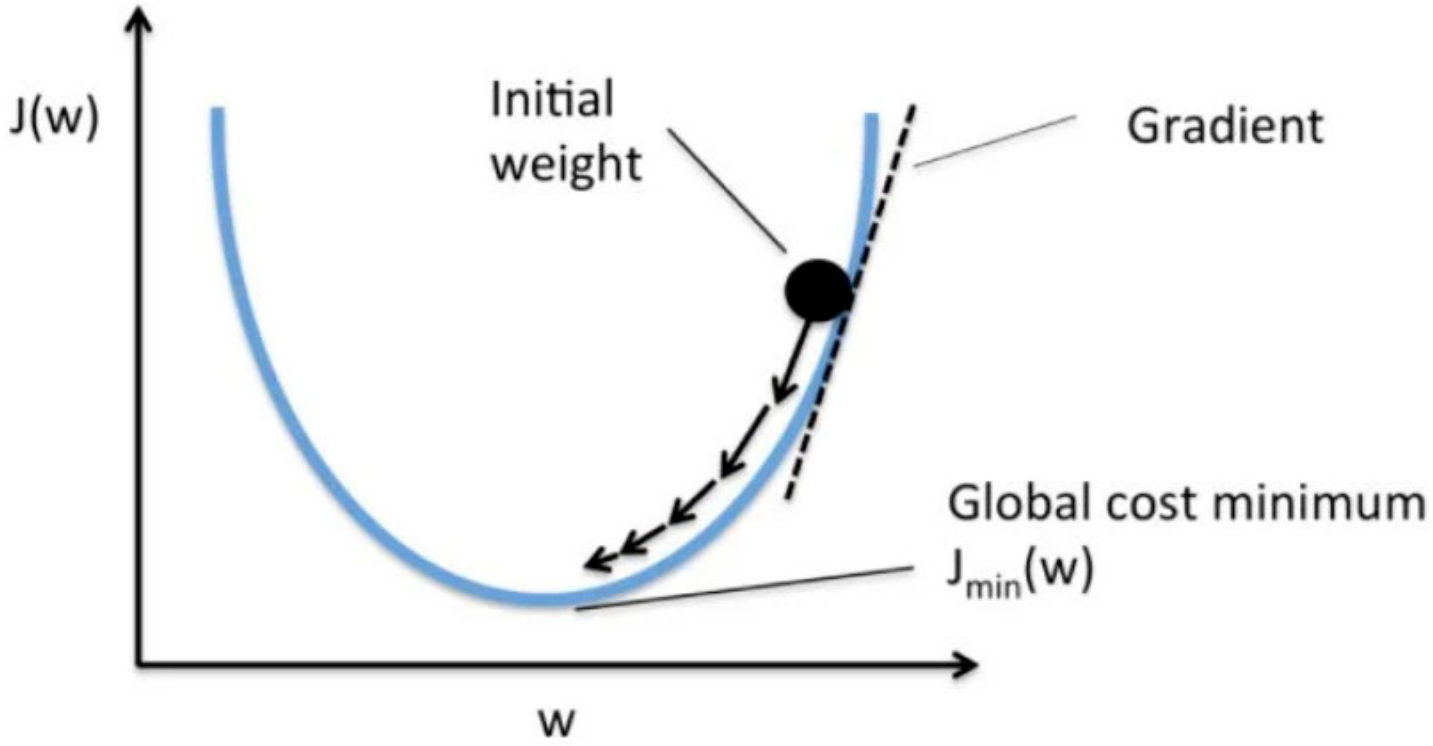
regardless of whether the input x is positive, negative, or zero, e^x is always a positive number.

Why Won't Normalization by the Sum Suffice ?

Training Process

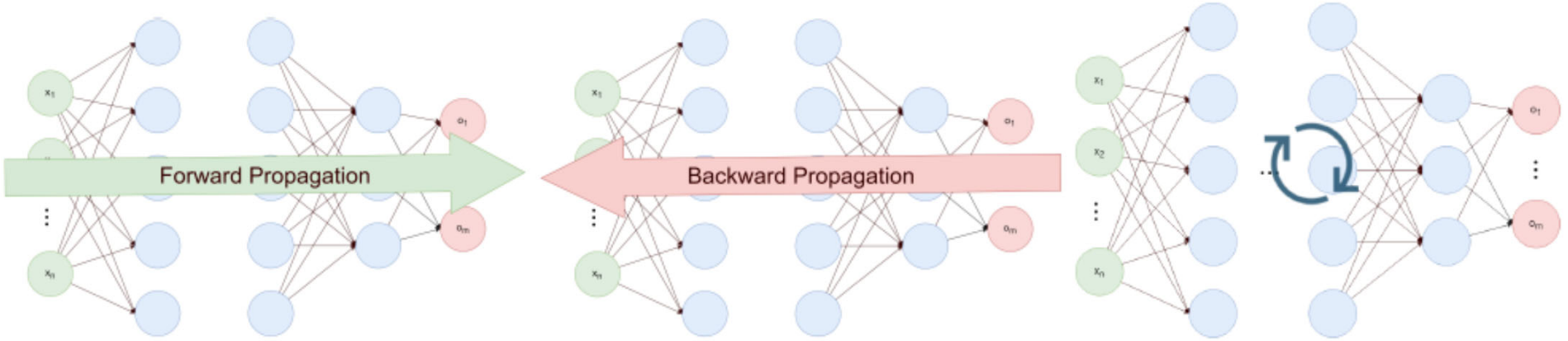


Gradient Descent



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.

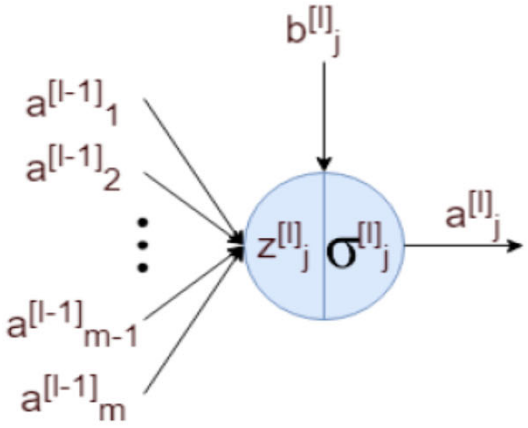


Figure: $a_j^{[l]} = \sum_{i=1}^{m} W_{ij}^{[l]} a_i^{[l-1]} + b_j^{[l]}$

So we can calculate these outputs layer by layer.

Forward propagation

After forward pass we will know:

- Loss value
- Network output
- Middle values

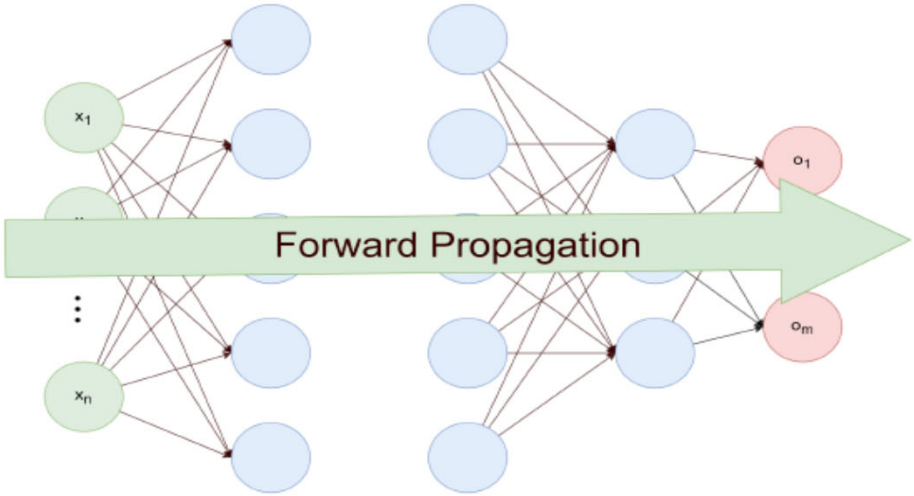


Figure: Forward pass

Backward Propagation

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

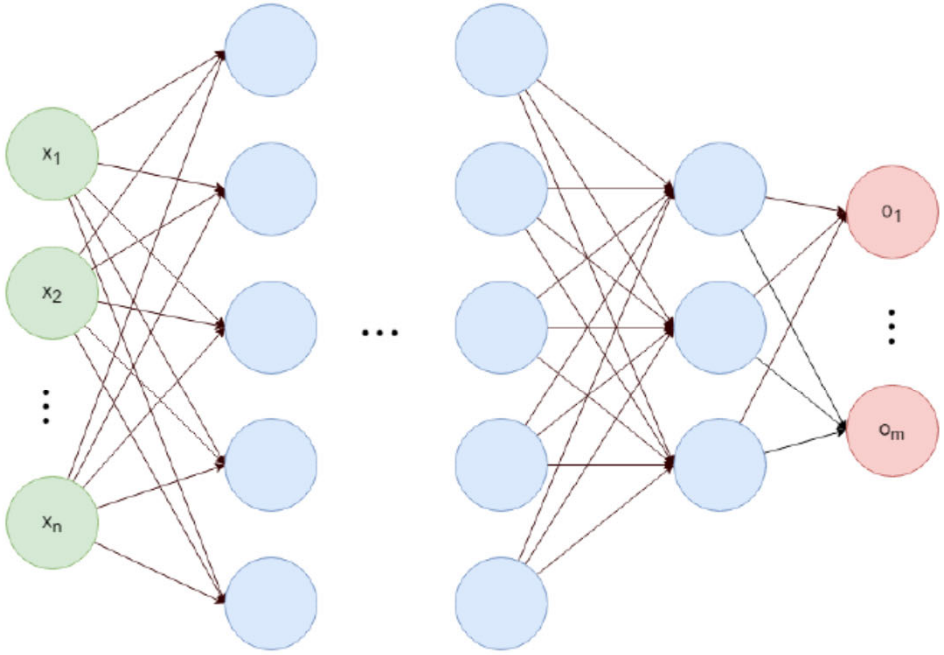
First idea:

- 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



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

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.

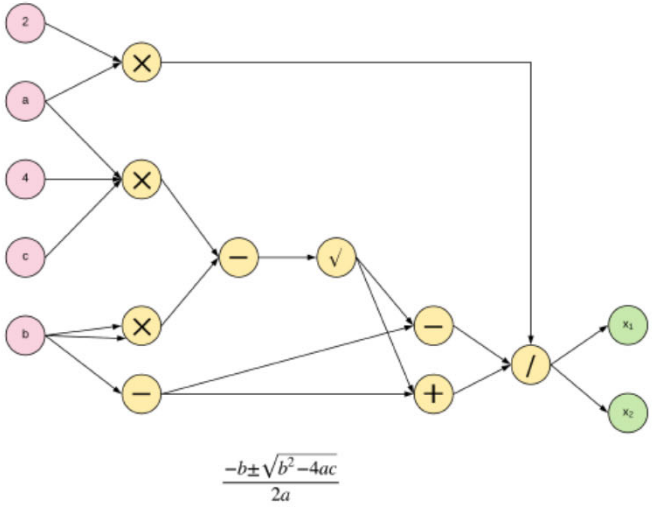
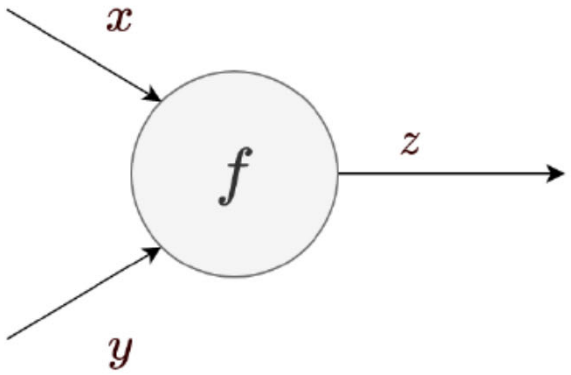


Figure: An example of computational graph, Source.

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?

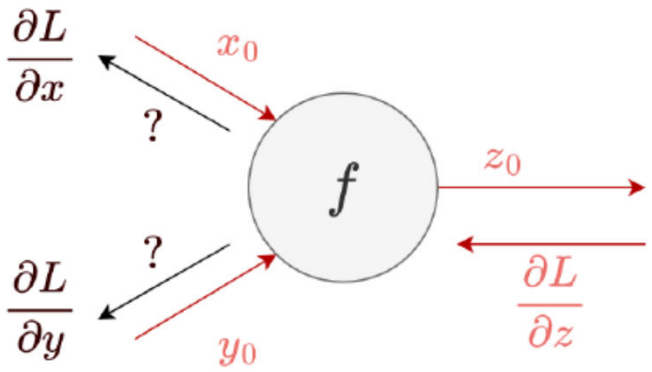
Backward propagation

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, $\left(\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}\right)$.

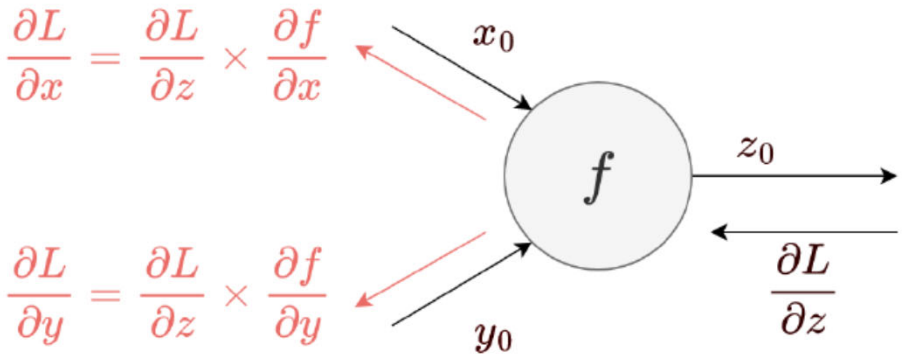


Backward Propagation

- We can use chain rule to do so.

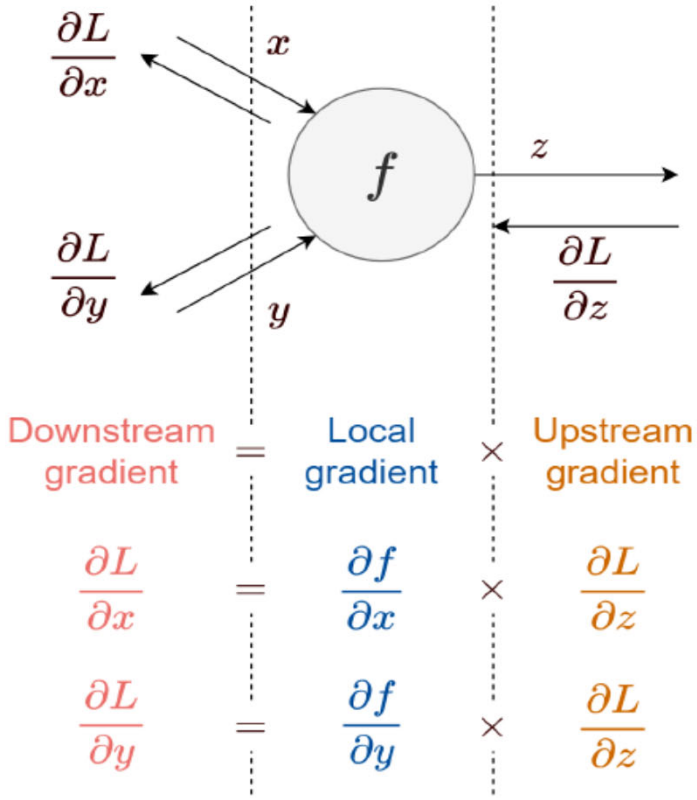
Chain rule:

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



Backward Propagation

Backpropagation for single module:

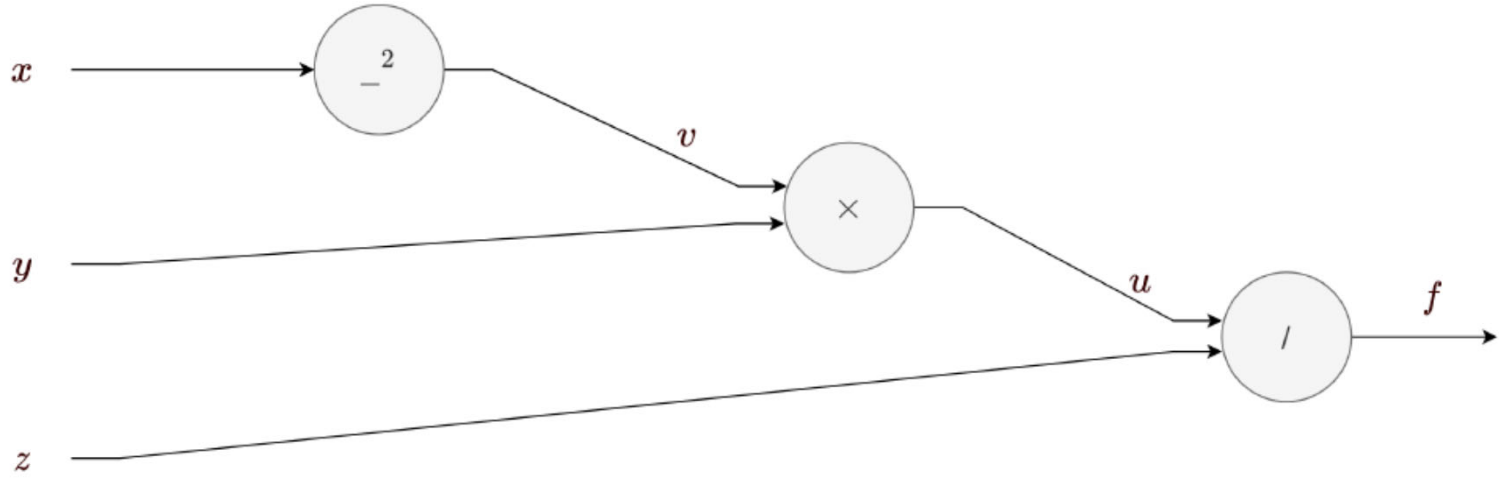


Backward Propagation : Example

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:

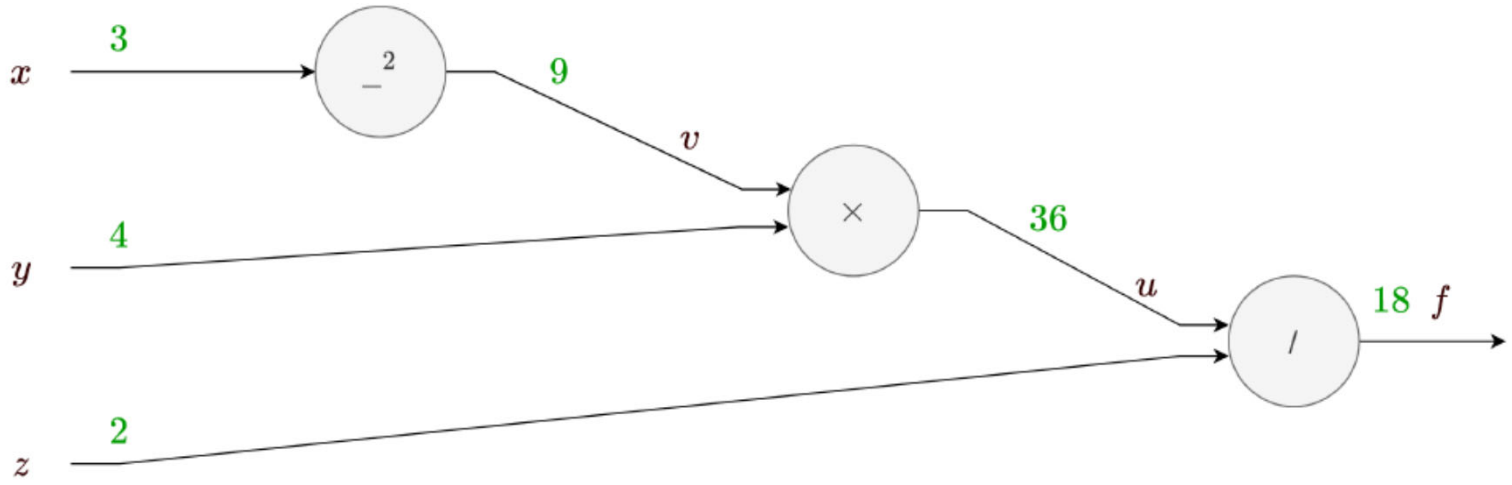
$$\begin{cases} v = x^2 \\ u = vy \\ f = \frac{u}{z} \end{cases} \qquad \begin{cases} \frac{\partial f}{\partial z} = -\frac{u}{z^2} \\ \frac{\partial f}{\partial y} = \frac{\partial u}{\partial y} \times \frac{\partial f}{\partial u} = v \times \frac{1}{z} = \frac{v}{z} \\ \frac{\partial f}{\partial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

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

Backward Propagation : Example

Now let's use backpropagation.

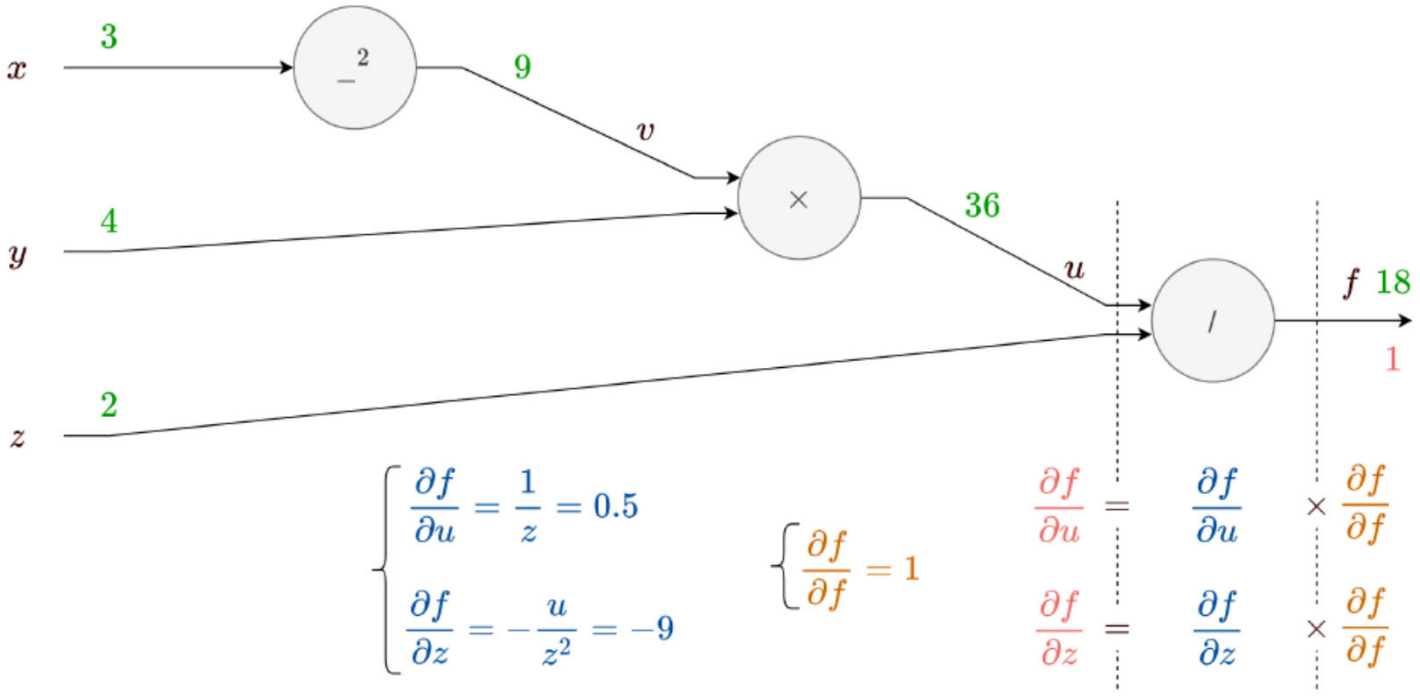
First we do forward propagation.



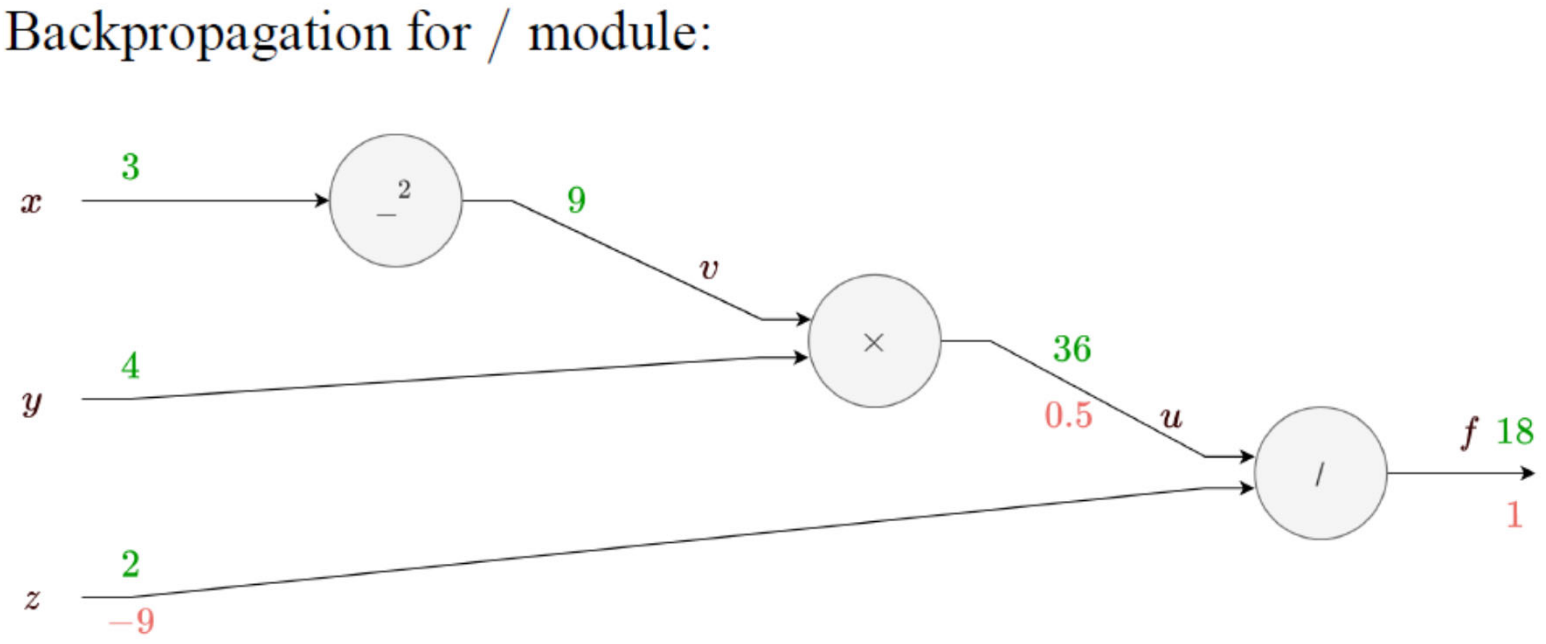
Second we will do backpropagation for each module.

Backward Propagation

Backpropagation for / module:

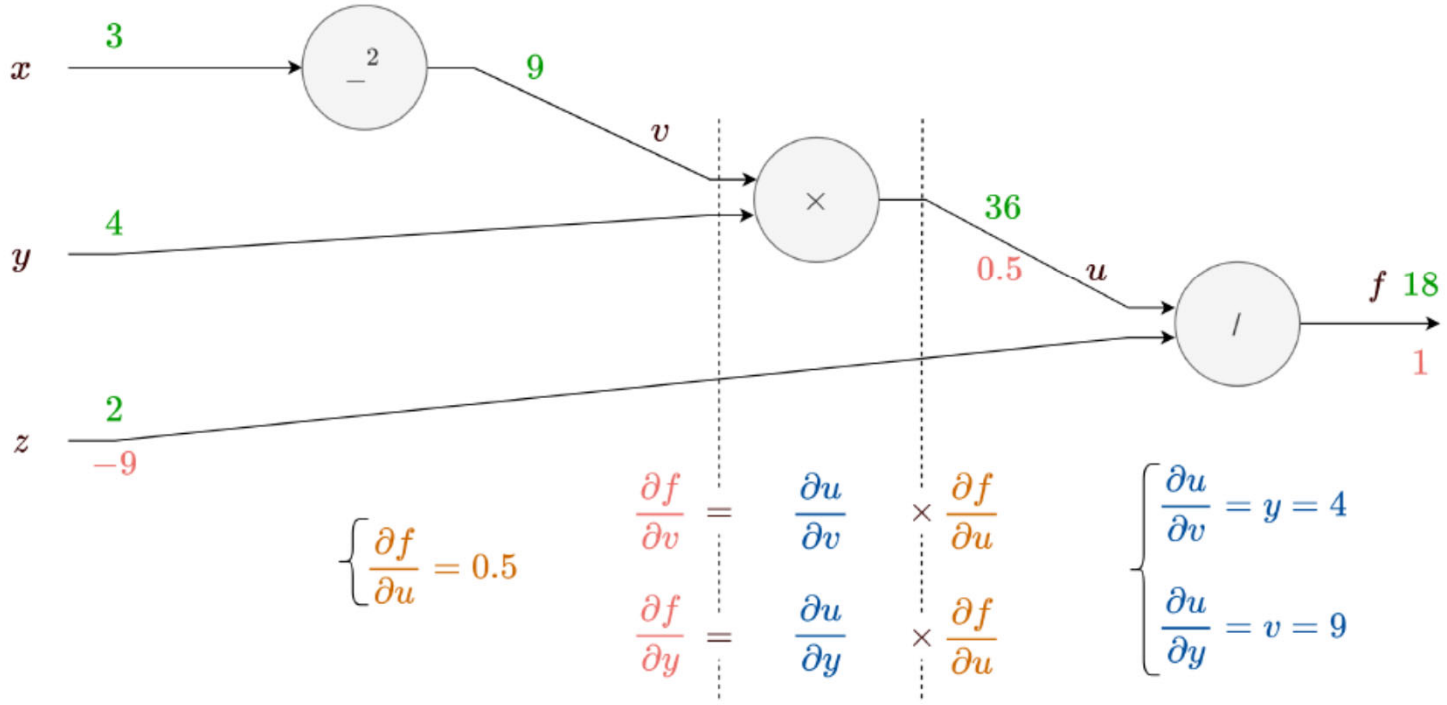


Backward Propagation : example



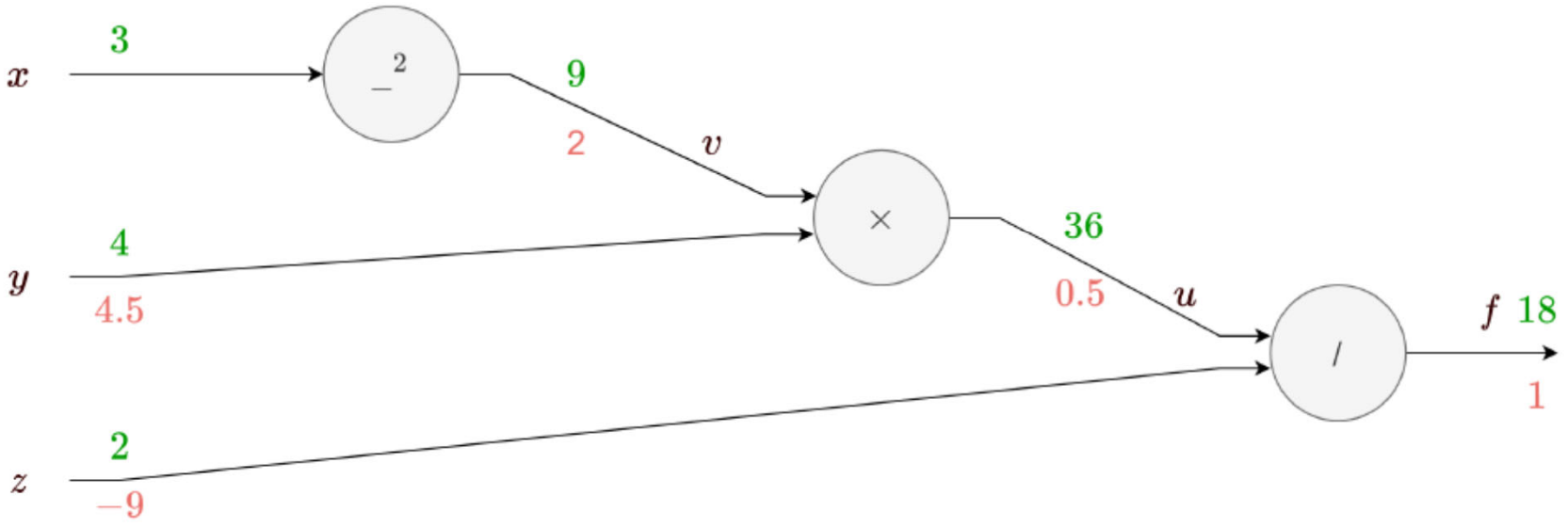
Backward propagation : example

Backpropagation for $\times$ module:



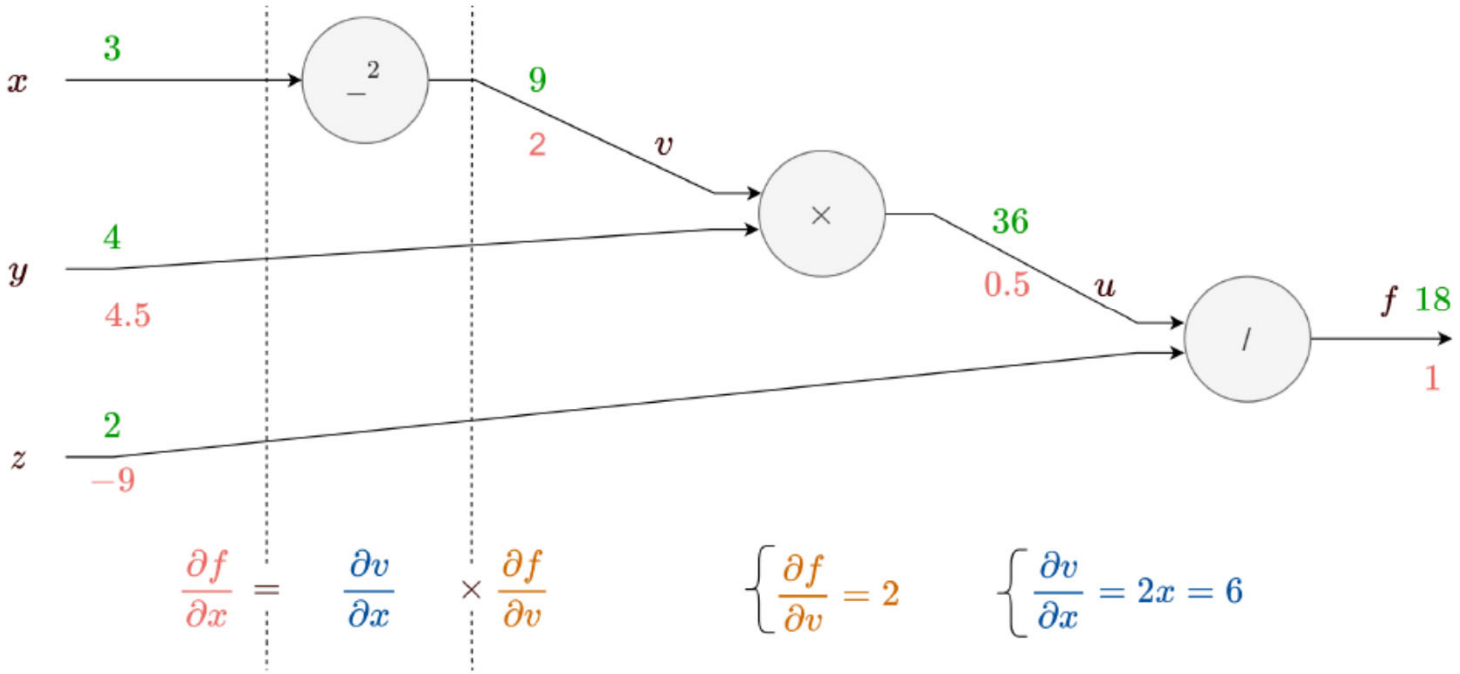
Backward propagation : example

Backpropagation for $\times$ module:



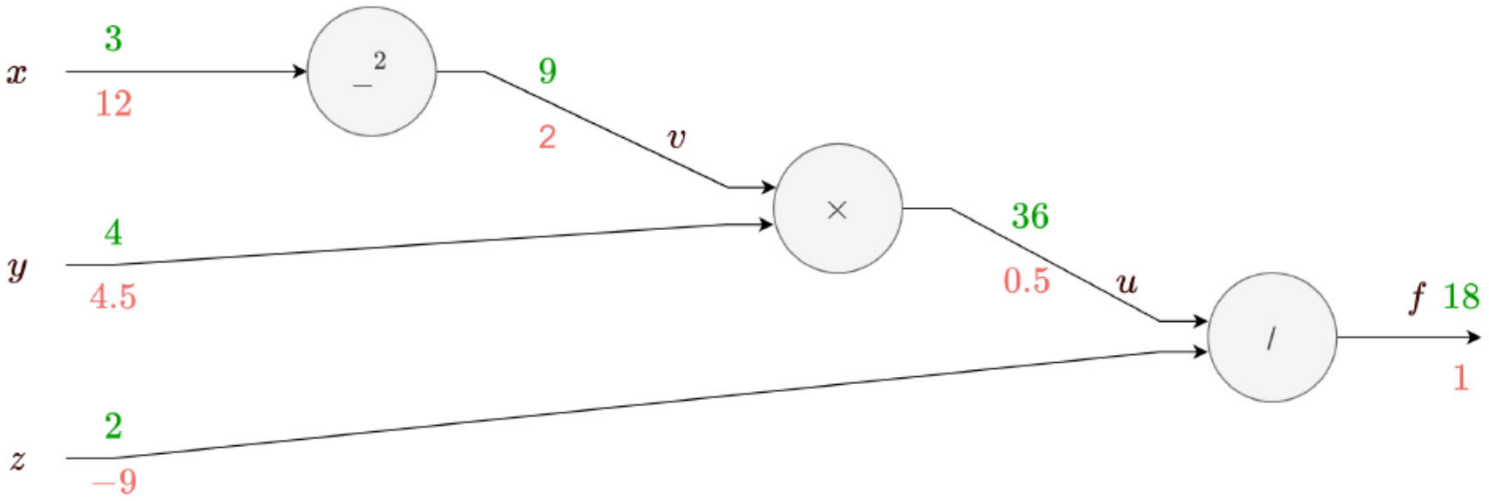
Backward propagation : example

Backpropagation for $\_^2$ module:



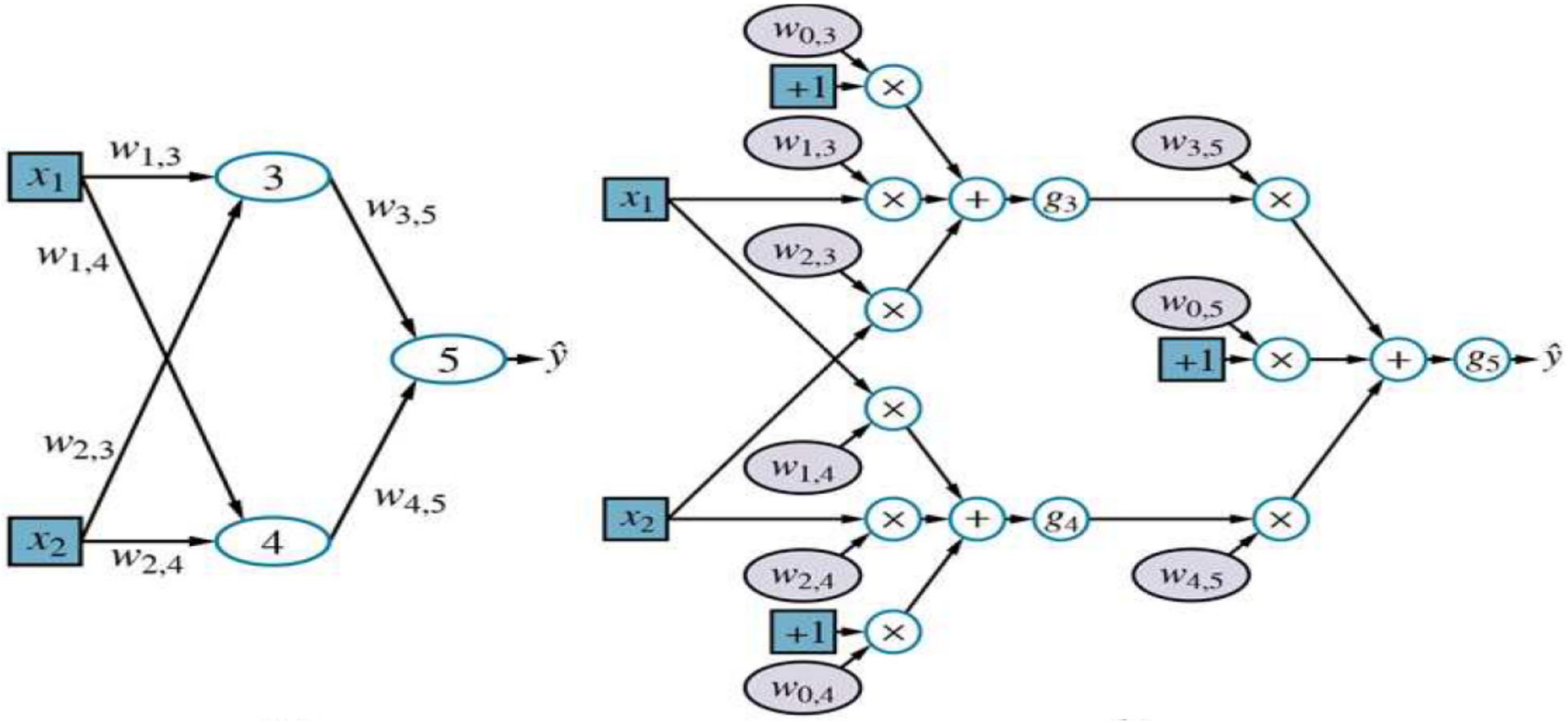
Backward propagation : example

Backpropagation for $\_^2$ module:

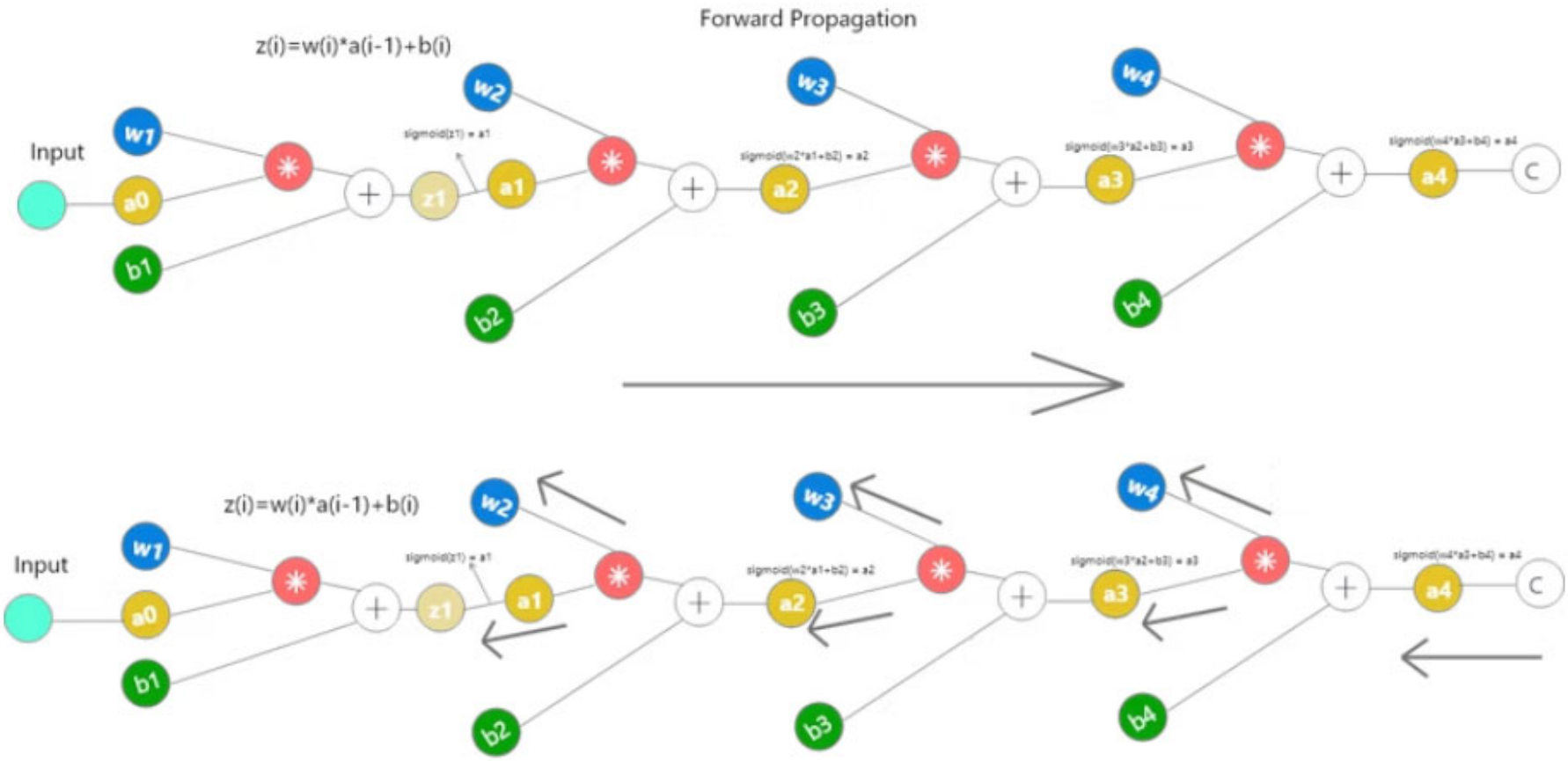


Results are the same as analytical results.

Computation Graph



Computation Graph for NN



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.

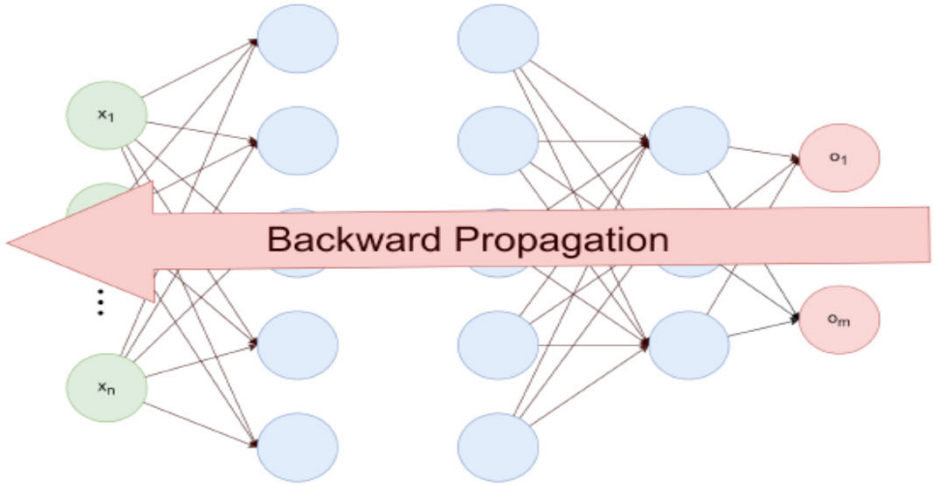


Figure: Backward pass

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

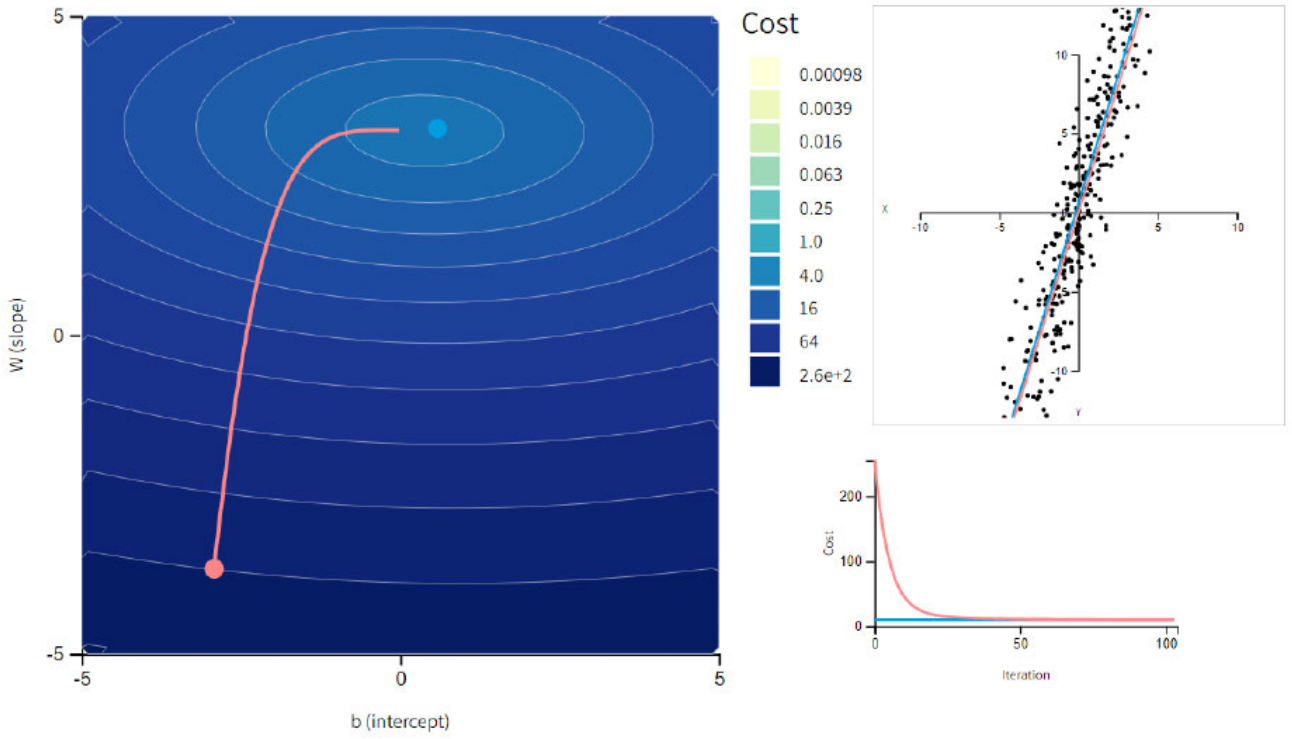
- In this type we use entire training set to calculate gradient.

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

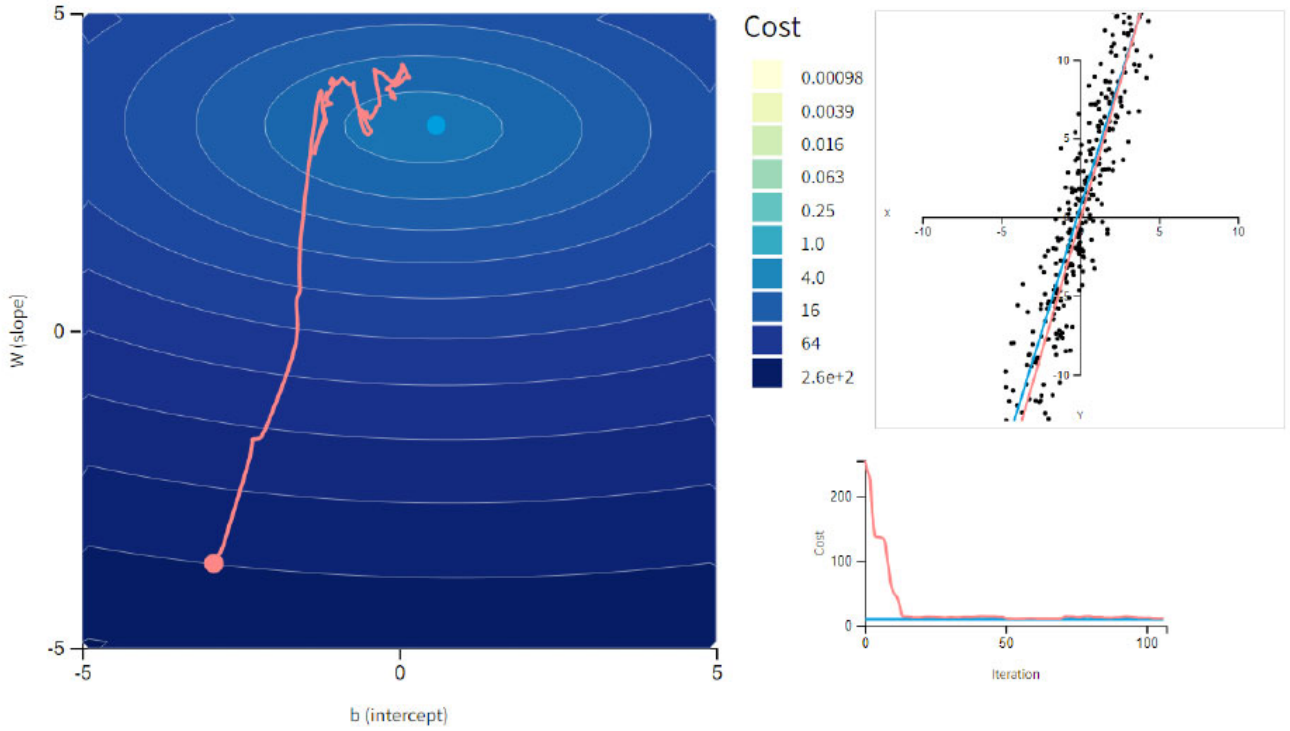
- Instead of calculating exact gradient, we can estimate it using our data.
- This is exactly what SGD does, it estimates gradient using only single data point.

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

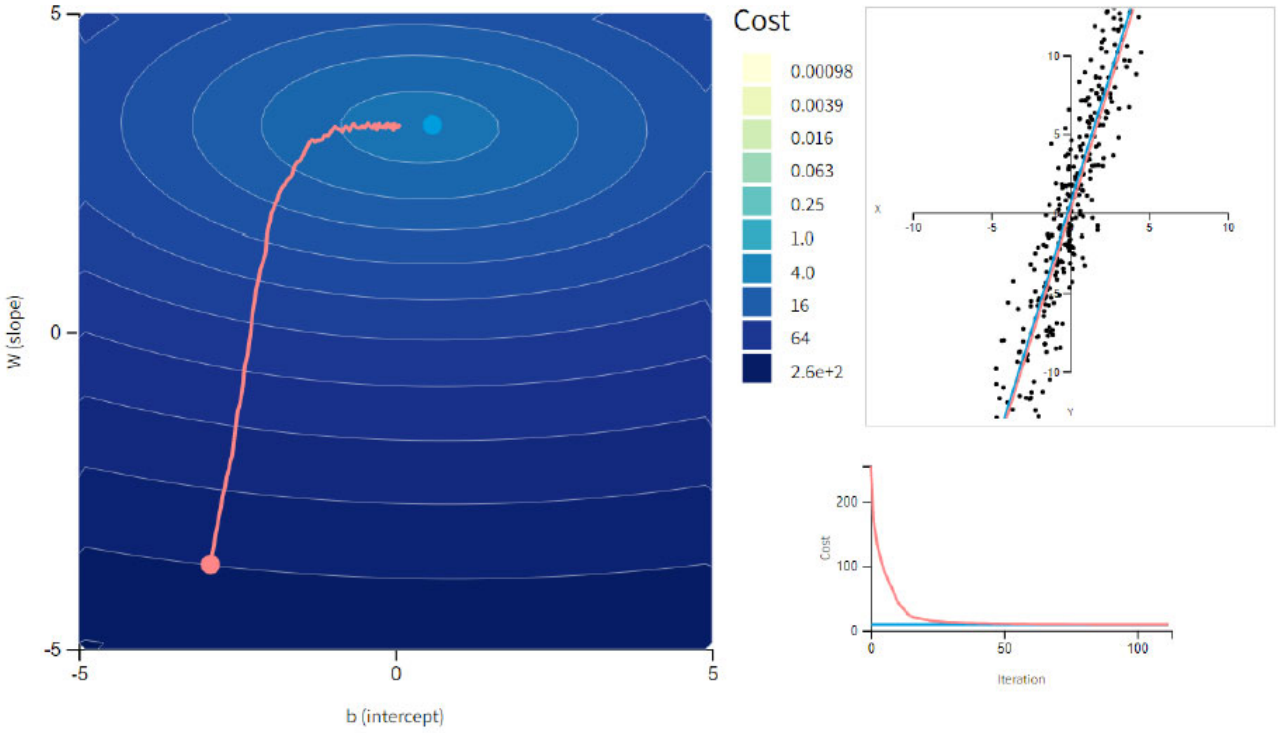
- In this method we still use estimation idea But use a batch of data instead of one point.

Mini-Batch Gradient:

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

- 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.

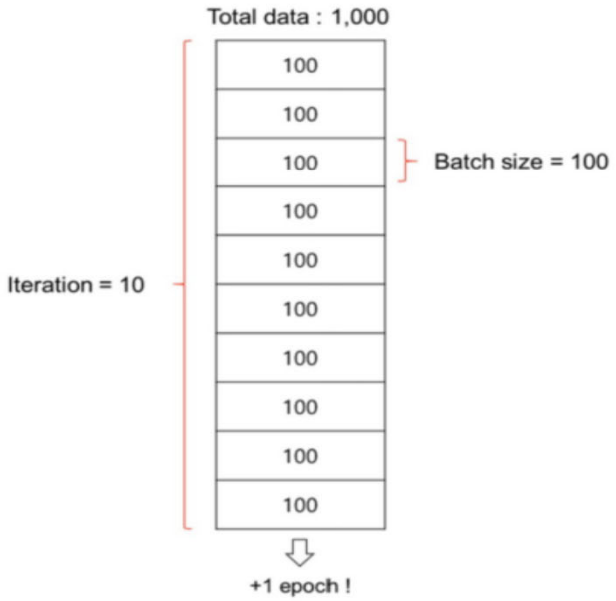
Various GD types: MiniBatch Gradient Descent



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

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

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

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

Improvement to Gradient Descent — Momentum

- One of the tricky aspects of Gradient Descent is dealing with steep slopes
  - Gradient is large there, you could take a large step when you actually want to go slowly and cautiously
  - This could result in bouncing back and forth, thus slowing down the training
- With Gradient Descent you make an update to the weights at each step, based on the gradient and the learning rate
  - Adjust the gradient
  - Adjust the learning rate

Momentum vs SGD

- Momentum is a way to adjust the gradient
- In SGD we look only at the current gradient and ignore all the past gradients along the path we took
  - there is a sudden anomaly in the loss curve, your trajectory may get thrown
  - using Momentum, let the past gradients guide overall direction
- how far in the past do you go?
- does every gradient from the past count equally?
  - this would make sense that things from the recent past should count more than things from the distant past
- Momentum algorithm uses the exponential moving average of the gradient, instead of the current gradient value.
- Which algorithm use momentum to updating the gradient ?
  - Stochastic Gradient Descent with Momentum (SGDM)

```
v = beta * v + (1 - beta) * gradient
parameters = parameters - learning_rate * v
```

Modify Learning Rate

- So for we have a constant learning rate
  - keeping the learning rate constant from one iteration to the next
  - gradient updates are using the same learning rate for all parameters.
- Modify the learning rate based on the gradient
- Use of past gradients (for each parameter separately) to choose the learning rate for that parameter.
- Optimizer algorithms that do this
  - Adagrad (Adaptive Gradient)
  - Adadelta
  - RMS Prop. (Root Mean Square Propagation)

Algorithm for updating learning rate

- for a parameter that has a steep slope
  - the gradients are large and the squares of the gradients are really large and always positive
  - the algorithm calculates the learning rate by dividing the accumulated squared gradients by a larger factor. This allows it to slow down on steep slopes.
- for shallower slopes
  - the accumulation is small and so the algorithm divides the accumulated squares by a smaller factor to compute the learning rate. This boosts the learning rate for gentle slopes.

- Adagrad squares the past gradients and adds them up weighting all of them equally
- RMSProp also squares the past gradients but uses their exponential moving average, thus giving more importance to recent gradients.

$$g_0 = 0$$

$$g_{t+1} \leftarrow g_t + \nabla_\theta \mathcal{L}(\theta)^2$$

$$\theta_j \leftarrow \theta_j - \epsilon \frac{\nabla_\theta \mathcal{L}}{\sqrt{g_{t+1}} + 1e^{-5}}$$

$$w_{t+1} = w_t - \frac{\alpha_t}{(v_t+\varepsilon)^{1/2}} * \left[\frac{\delta L}{\delta w_t}\right]$$

$$v_t = \beta v_{t-1} + (1-\beta) * \left[\frac{\delta L}{\delta w_t}\right]^2$$

Adam optimizer algorithm

- Adam is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iterative based in training data.
- Adam realizes the benefits of both SGDM and RMSProp.
  - **Root Mean Square Propagation** (RMSProp)
  - Stochastic gradient descent with momentum
- algorithm calculates an exponential moving average of the gradient and the squared gradient, and the parameters beta1 and beta2 control the decay rates of these moving averages.

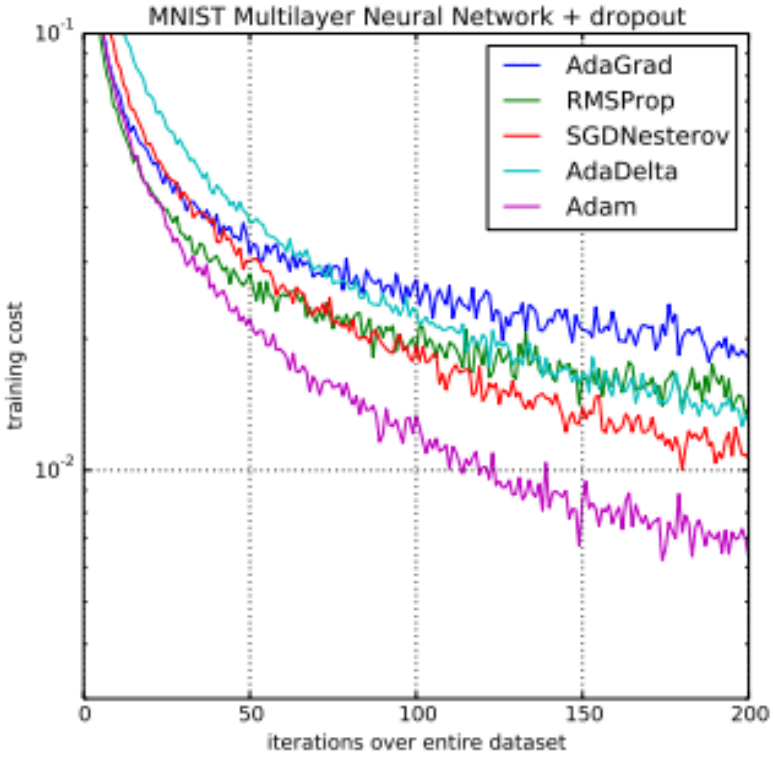
Adam

- Adam is a popular algorithm in the field of deep learning because it achieves good results fast.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \left[\frac{\delta L}{\delta w_t} \right]$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \left[\frac{\delta L}{\delta w_t} \right]^2$$

$$w_{t+1} = w_t - m_t \left(\frac{\alpha}{\sqrt{\hat{v}_t} + \varepsilon} \right)$$

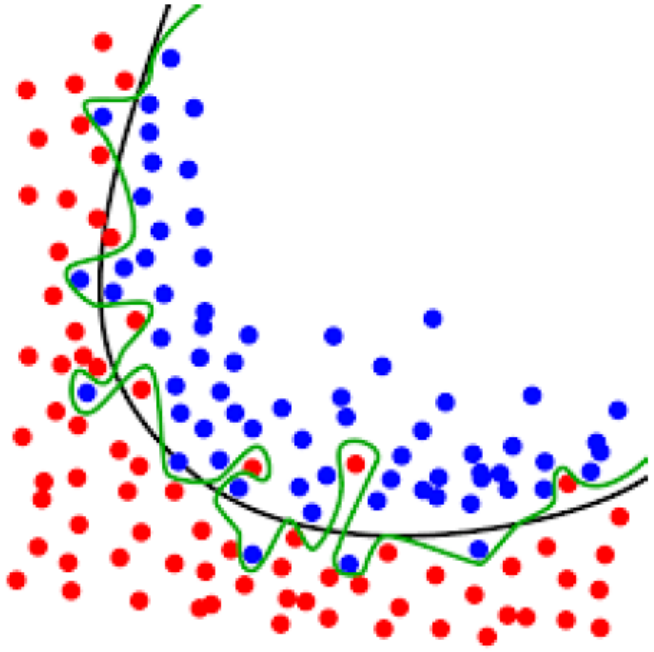


Training MLPs

- So far we have learned about how MLPs work and how to update their parameters in order to perform better.
- But training MLPs is not that easy.
- You will face several different challenges in this procedure.
- In this section we will talk about this challenges and how to solve them.

Problem: OverFitting in a Neural Network

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



Regularization

- It is possible that some features that is actually irrelevant appears by chance to be useful, resulting in overfitting
- Use regularization to avoid overfitting
- Regularization:
  - Minimize the Empirical Loss and the Complexity of the model
  - $Cost = Empirical\ Loss + \lambda\ Complexity$

Regularization

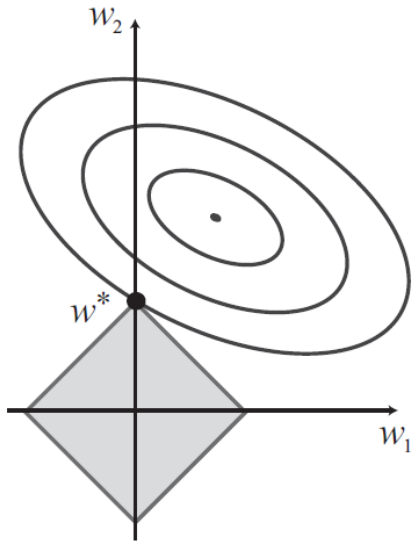
- In NN the complexity can be specified as a function of the weights
  - For e.g. $Complexity = \sum_i |w_i|^q$
- With q=1 we have L1 regularization which minimize the sum of the absolute values
- With q=2 we have L2 regularization which minimize the sum of squares
- Which regularization should pick ?
  - Depend on the specific problem
  - L1 regularization tend to produce sparse model (often set many weights to zero)
- Minimizing $Loss(w) + \lambda\, Complexity$(w) is equivalent to minimizing Loss(w) subject to constraint that $Complexity(w) \leq c$ for some constant c that is related to λ

L1 Regularization

- Figure shows the diamond-shaped box represents the set of points **w** in two-dimensional weight space that have L1 complexity less than c
- The solution is somewhere inside the box
- The concentric ovals represent contours of the loss

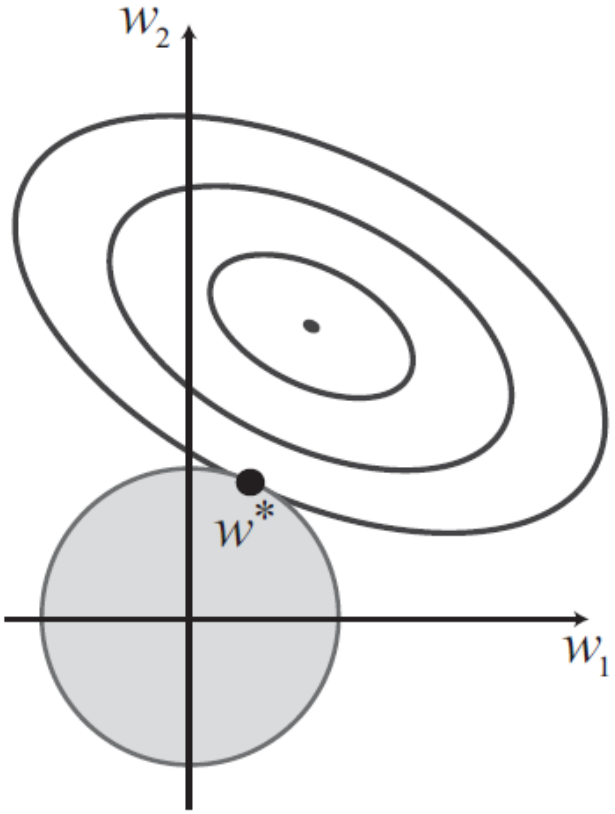
function, with the minimum loss at the center

- We want the point in the box that is closest to the minimum;
- For an arbitrary position of the minimum and its contours, it will be common for the corner of the box to find its way closest to the minimum



L2 Regolarization

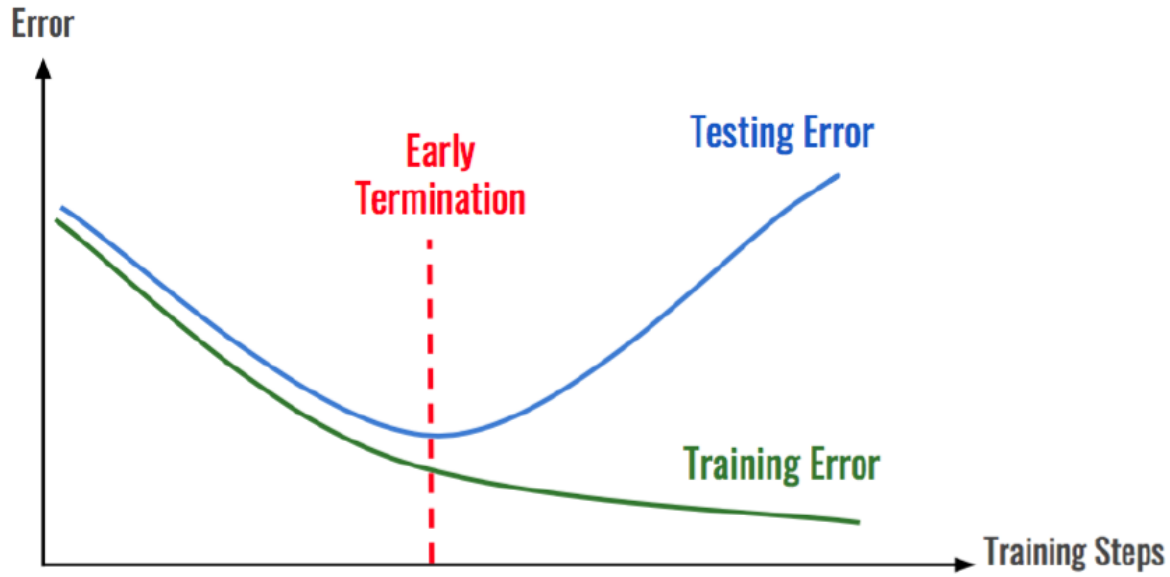
- L2 complexity measure, represent a circle instead of diamond
- in general, there is no reason for the intersection to appear on one of the axes
- L2 regularization does not tend to produce zero weights



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

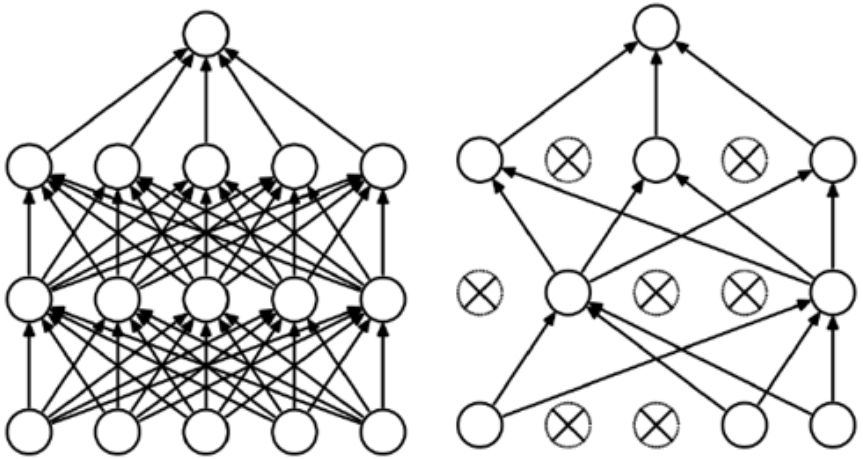
Solution 2: Early Stopping

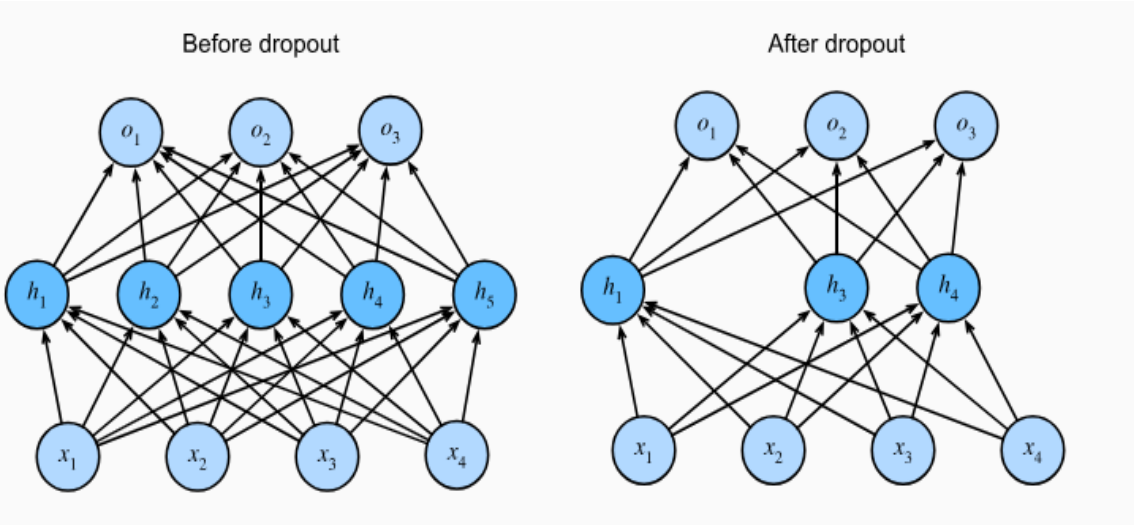
- Stop the training procedure when the validation error is minimum.



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.

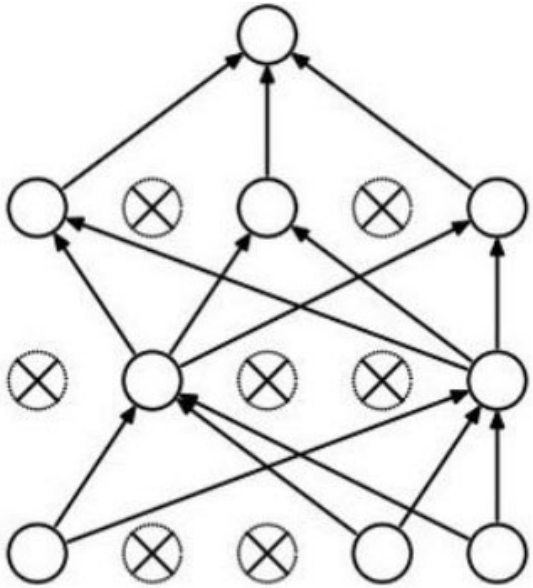




Dropout: Why can this possibly be a good idea?

- 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.

Forces the network to have a redundant representation



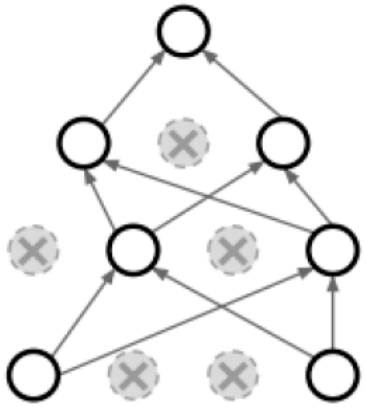
Another interpretation:

Dropout is training a large ensemble of models (that share parameters).

Each binary mask is one model, gets trained on only ~one datapoint.

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

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

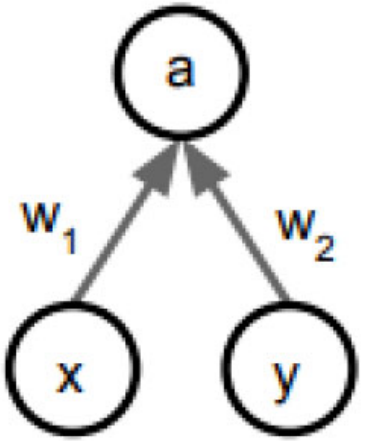


Figure: Simple neural network. [6]

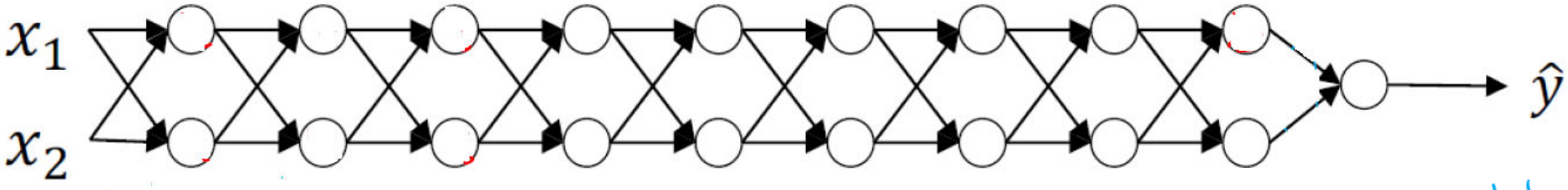
If a unit is retained with probability p during training, the outgoing weights of that unit are multiplied by p at test time

Vanishing/Exploding Gradient

- Vanishing
  - Gradients often get smaller as the algorithm progresses down. As a result, gradient descent updates do not effectively change the weights of the lower layer connections, and training never converges.
  - Make learning slow especially of front layers in the network.
- Exploding
  - Gradients can get bigger and bigger, so there are very large weight updates at many levels, causing the algorithm to diverge.
  - The model is not learning much on the training data therefore resulting in a poor loss.

Exploding Gradient

- Root cause of exploding gradients
  - network architecture and the choice of activation functions
    - when multiple layers have weights greater than 1, the gradients can grow exponentially as they propagate back through the network during training
    - exacerbated when using activation functions with outputs that are not bounded (hyperbolic tangent or the sigmoid function)
  - initialization of the network's weights
    - If the initial weights are too large, even a small gradient can be amplified through the layers,

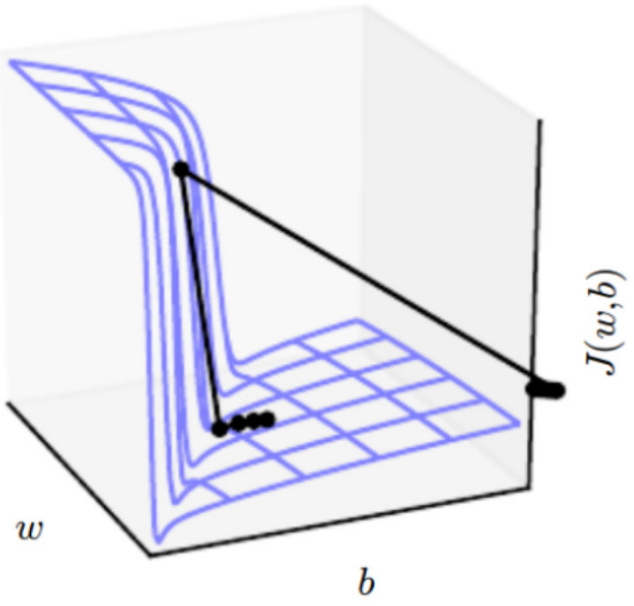


$\hat{y} = w^{[L]} w^{[L-1]} w^{[L-2]} \dots w^{[3]} w^{[2]} w^{[1]} X$

$\hat{y} = w^{[L]} \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1} X$

Consequences of Exploding Gradients

- weight updates during training can become so large that they cause the learning algorithm to overshoot the minima of the loss function
- This can result in model parameters diverging to infinity, causing the learning process to fail



Solutions to exploding gradient

- **Gradient clipping** :
  - This technique involves setting a threshold value, and if the gradient exceeds this threshold, it is scaled down to keep it within a manageable range. This prevents any single update from being too large.
- **Weight Initialization:**
  - Using a proper weight initialization strategy, such as Xavier or He initialization, can help prevent gradients from becoming too large at the start of training.
- **Batch Normalization:**
  - maintain the output of each layer within a certain range, reducing the risk of exploding gradients.

Gradient Clipping

- Two approaches to do so:
  - Clipping by value
  - Clipping by norm

Gradient Clipping by value

- Set a max (α) and min (β) threshold value.
- For each index of gradient $\boldsymbol{g}_i$ if it is lower or greater than your threshold clip it:

$$
\begin{aligned}
&\text{if } \boldsymbol{g}_i > \alpha : \\
&\qquad \boldsymbol{g}_i \leftarrow \alpha \\
&\text{else if } \boldsymbol{g}_i < \beta : \\
&\qquad \boldsymbol{g}_i \leftarrow \beta
\end{aligned}
$$

Gradient clipping by norm

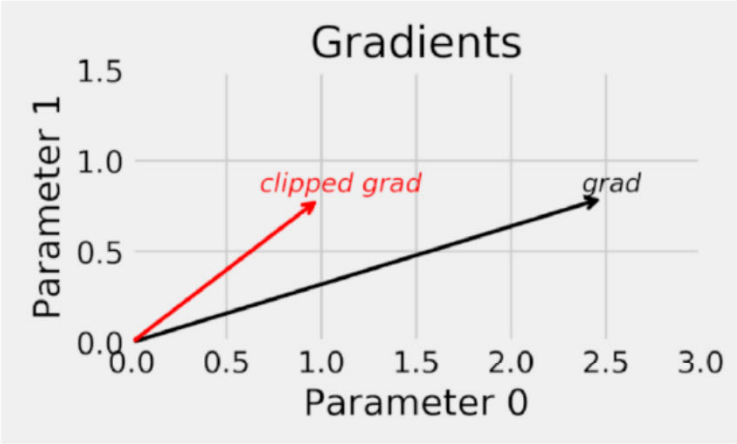


Figure: The effect of clipping by value. Source

- Clipping by value will change gradient direction.
- To preserve direction use clipping by norm.

Gradient clipping by norm

- Clip the norm $\|\boldsymbol{g}\|$ of the gradient $\boldsymbol{g}$ before updating parameters:

$$\text{if } \|\boldsymbol{g}\| > v:$$
$$\boldsymbol{g} \leftarrow \frac{\boldsymbol{g}}{\|\boldsymbol{g}\|} v$$

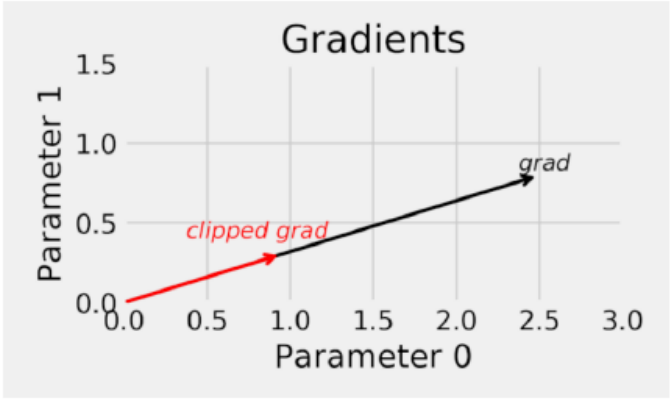


Figure: The effect of clipping by norm. source

solution: **gradient clipping**

Algorithm 1 Pseudo-code for norm clipping

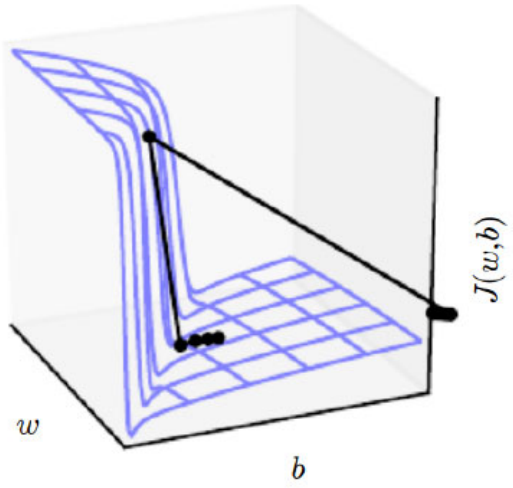
$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
if $\|\hat{\mathbf{g}}\| \geq threshold$ **then**
  $\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|} \hat{\mathbf{g}}$
end if

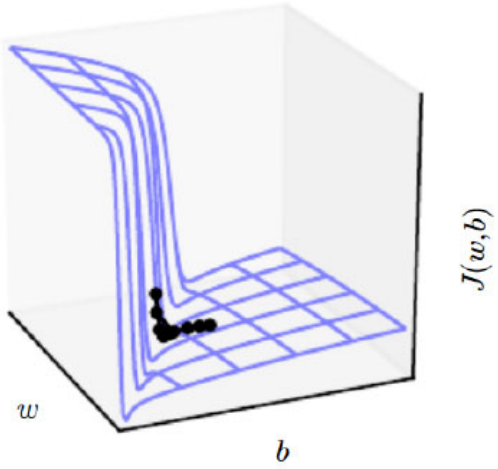
v is the threshold for clipping which is a hyperparameter.

- Gradient clipping saves the direction of gradient and controls its norm.

Gradient Clipping

- Clipping by value will change the direction of the gradient, so it will send us to a bad neighborhood.
- Clipping by norm will preserve the direction and just control the value.
- So it is better to use clipping by norm.





Weight initialization

- A bad initialization may increase convergence time or even make optimization diverge
- How to initialize?
  - Zero initialization
  - Random initialization

Weight Initialization: Zero Initialization

Zero Initialization method:

$$\begin{cases} W^{[l]} = \mathbf{0}, \\ b^{[l]} = \mathbf{0} \end{cases}$$

- Simple but perform very poorly. (why?)
- Zero initialization will lead each neuron to learn the same feature
- This problem is known as network failing to break symmetry
- In fact any constant initialization suffers from this problem.

Weight Initialization: Zero Initialization

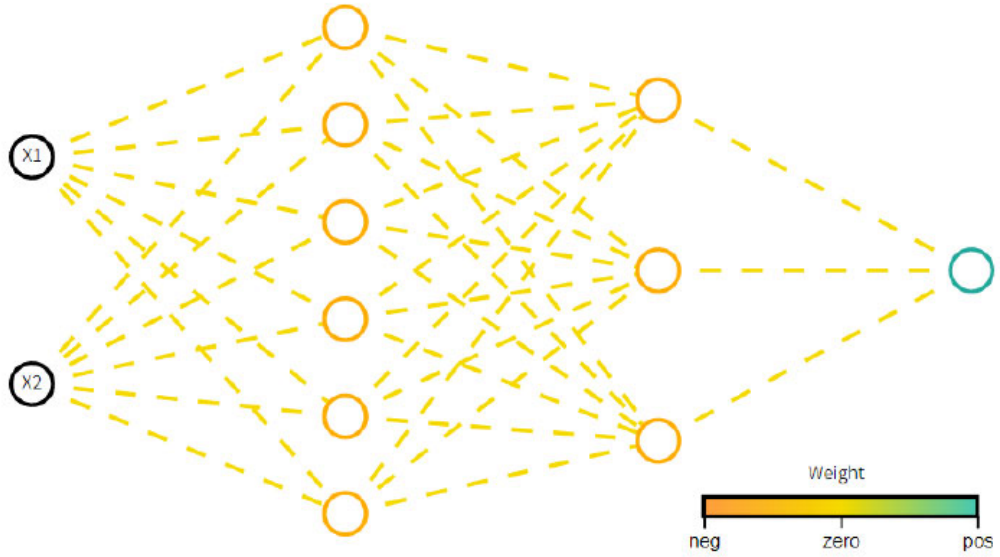


Figure: As we can see network has failed to break symmetry. There has been no improvement in weights after about 600 epochs of training

- We need to break symmetry. How? using randomness.

Weight Initialization: Random Initialization

- To use randomness in our initialization we can use uniform or normal distribution:

General Uniform Initialization:

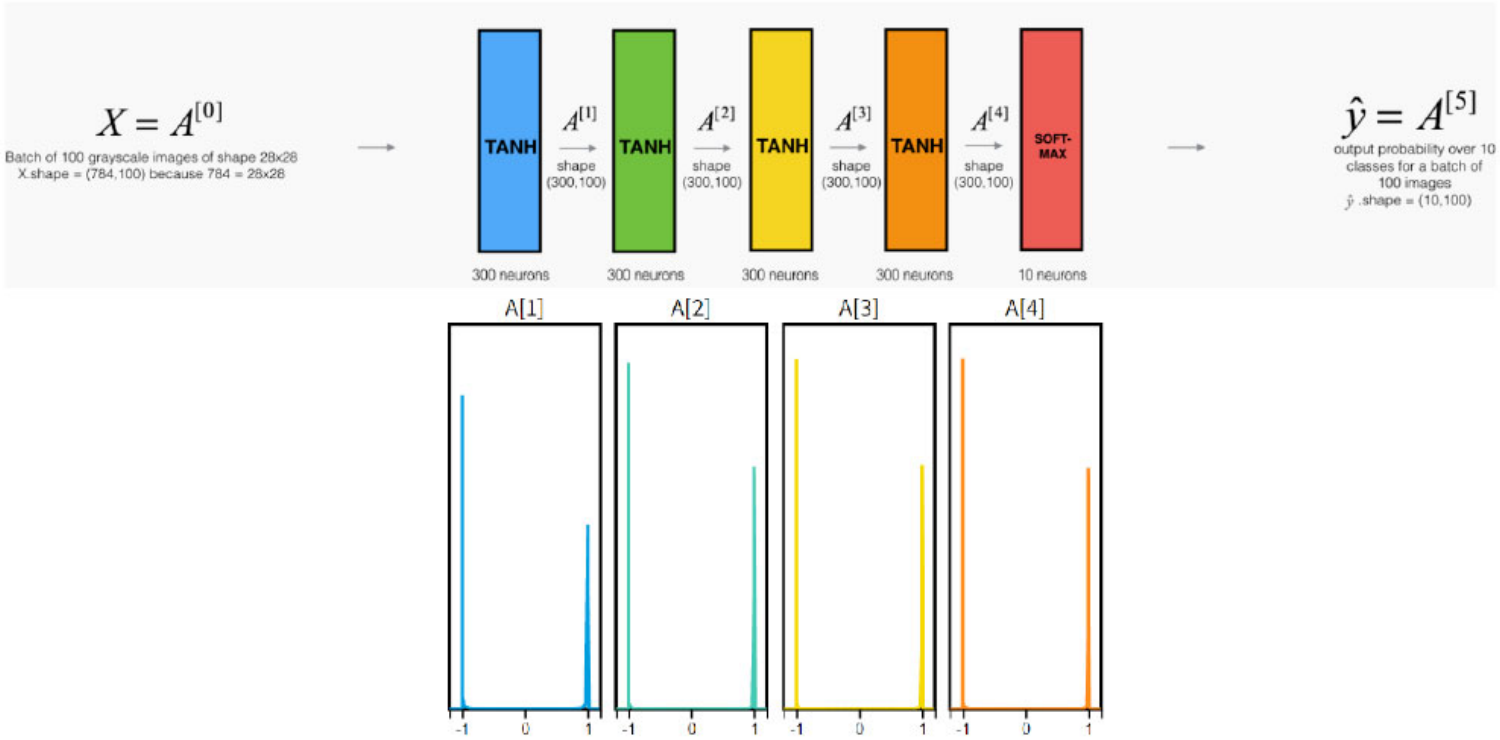
$$\begin{cases} W^{[l]} \sim U(-r, +r), \\ b^{[l]} = 0 \end{cases}$$

General Normal Initialization:

$$\begin{cases} W^{[l]} \sim \mathcal{N}(\mu = 0, \sigma^2), \\ b^{[l]} = 0 \end{cases}$$

- But this is really crucial to choose r or σ properly.

Weight Initialization: Random Initialization



Uniform initialization problem. On the top, you can see the model architecture, and on the bottom, you can see the density of each layer's output. Model has trained on MNIST dataset for 4 epochs. Weights are initialized randomly from $U(\frac{-1}{\sqrt{n^{[l-1]}}}, \frac{1}{\sqrt{n^{[l-1]}}})$

Weight Initialization: Random Initialization

- How to choose r or σ?
- We need to follow these rules:
  - keep the mean of the activations zero.
  - keep the variance of the activations same across every layer.

Xavier Initialization:

- For Uniform distribution use:

$$r = \sqrt{\frac{3}{\text{fan}_{\text{avg}}}}$$

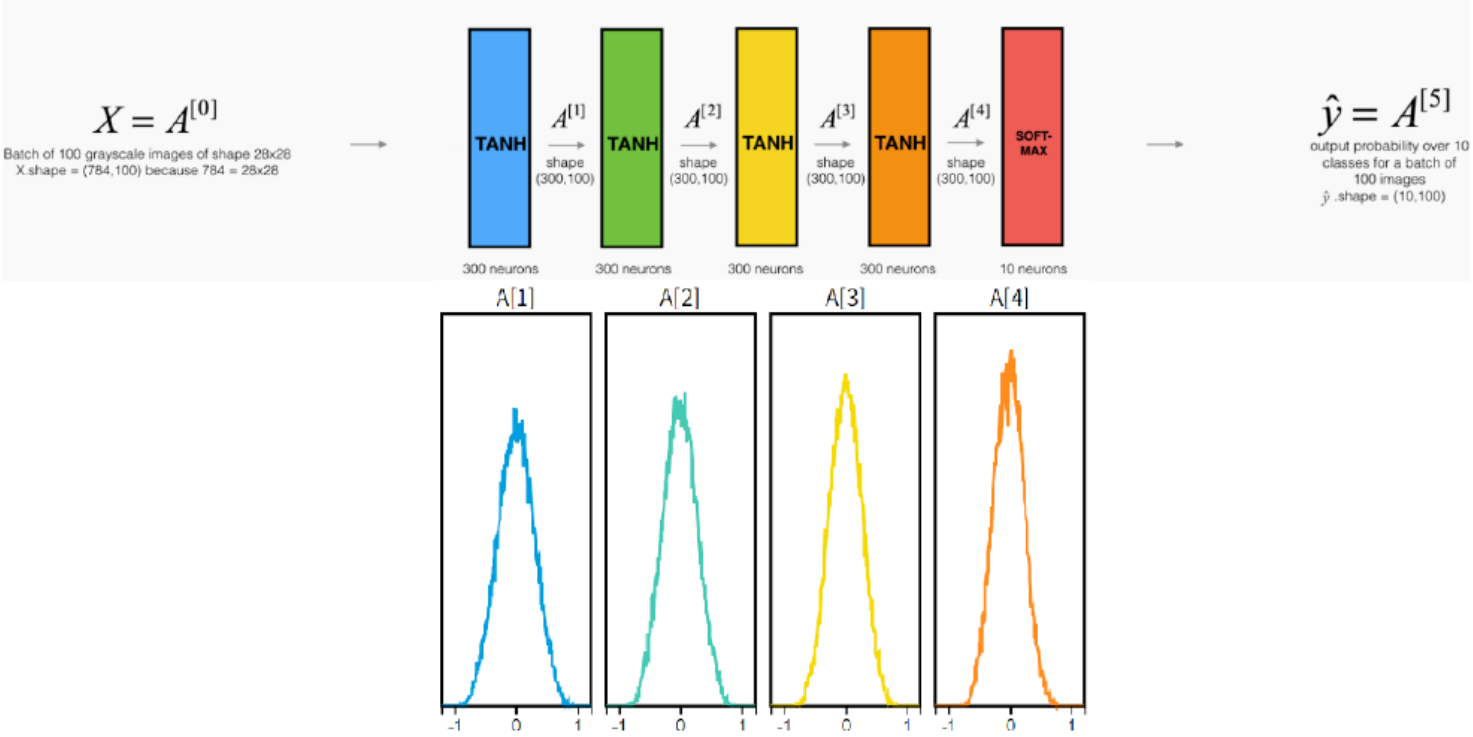
$$\begin{cases} \text{fan}_{\text{in}}^{[l]} = n^{[l-1]} & \text{(layer } l \text{ number of inputs)}, \\ \text{fan}_{\text{out}}^{[l]} = n^{[l]} & \text{(layer } l \text{ number of outputs)}, \\ \text{fan}_{\text{avg}}^{[l]} = \frac{n^{[l-1]}+n^{[l]}}{2} \end{cases}$$

- For Normal distribution use:

$$\sigma^2 = \frac{1}{\text{fan}_{\text{avg}}}$$

Weight Initialization: Xavier Initialization

- Xavier initialization works well on Tanh, Logistic or Sigmoid activation function.



Vanishing gradient is no longer problem using Xavier initialization. Model has trained on MNIST dataset for 4 epochs.

Weight Initialization: He Initialization

- Different method has proposed for different activation functions.

He Initialization:

- For Normal distribution:

$$\sigma^2 = \frac{2}{n^{[l]}}$$

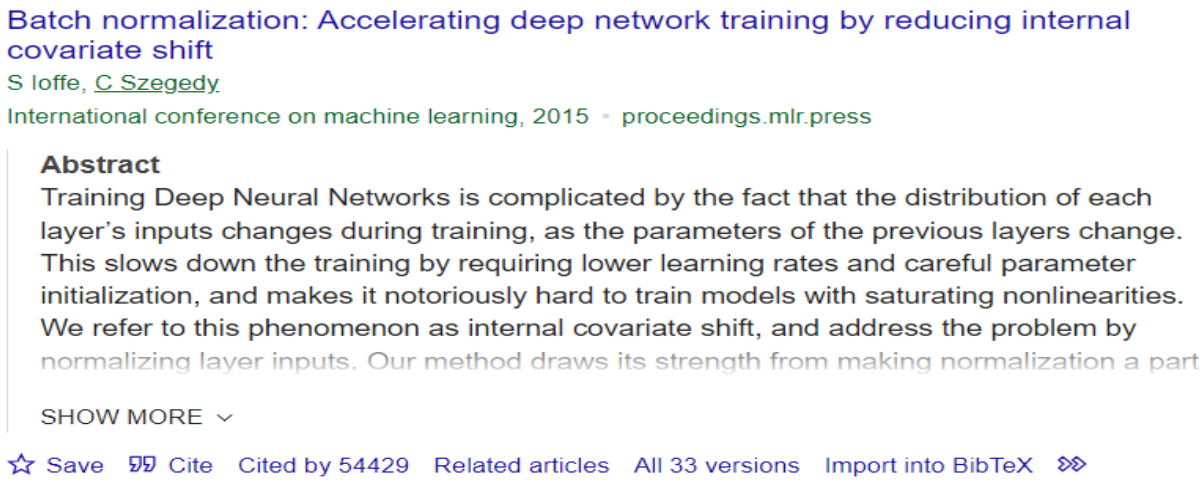
- For Uniform distribution:

$$r = \sqrt{3\sigma^2}$$

- He initialization works well on ReLU and its variants.

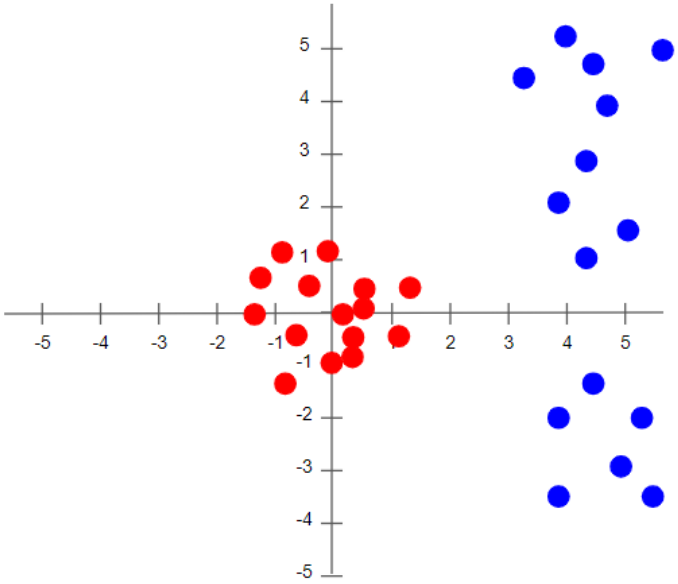
Bach normalization

- batch normalization has made it possible for practitioners to routinely train networks with over 100 layers
- A secondary benefit of batch normalization lies in its inherent regularization.



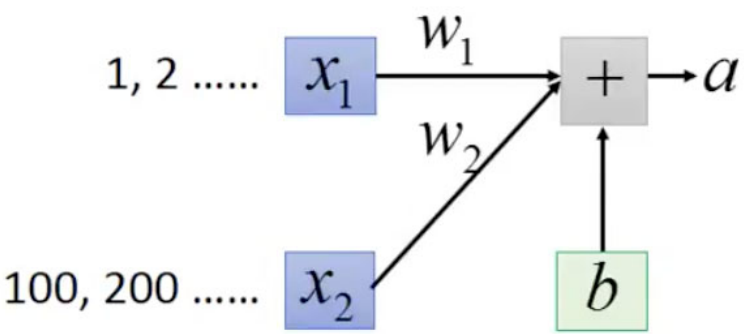
Why normalization is needed

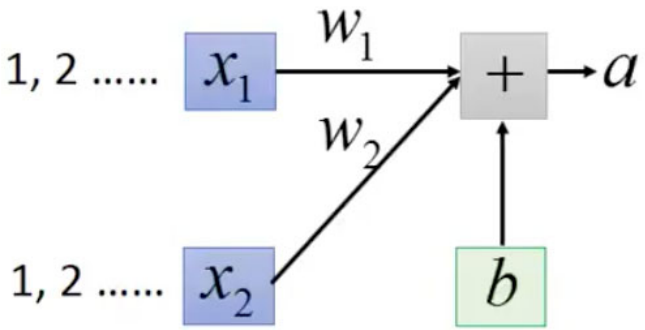
- feature might have a different range of values
  - values for feature x1 might range from 1 through 5
  - values for feature x2 might range from 1000 to 99999.
- In the picture, the effect of normalizing data is shown.
- The original values (in blue) are centered around zero (in red).
- This ensures that all the feature values are now on the same scale.

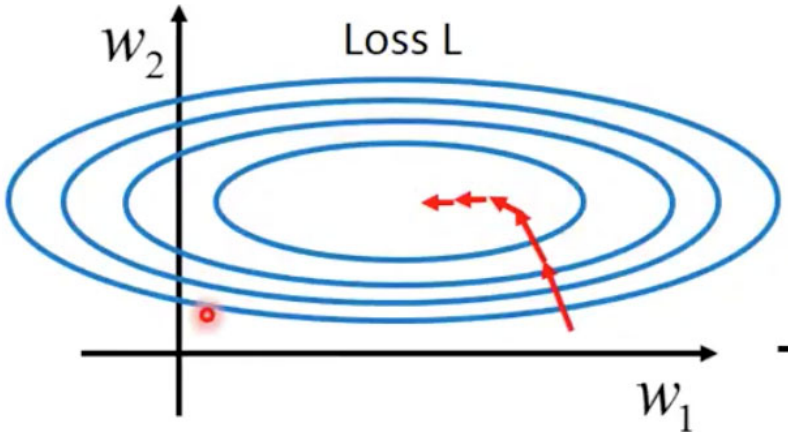


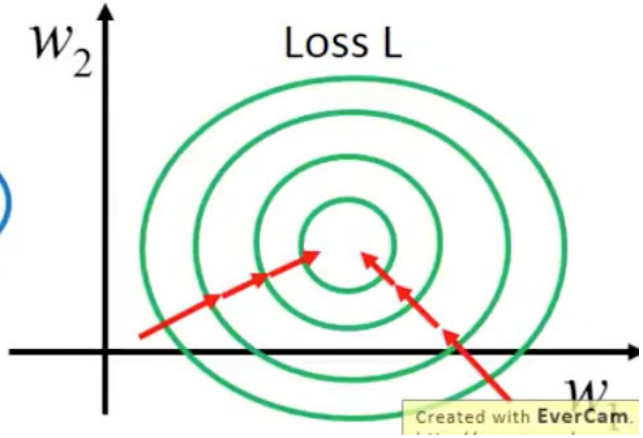
Feature Scaling

Make different features have the same scaling



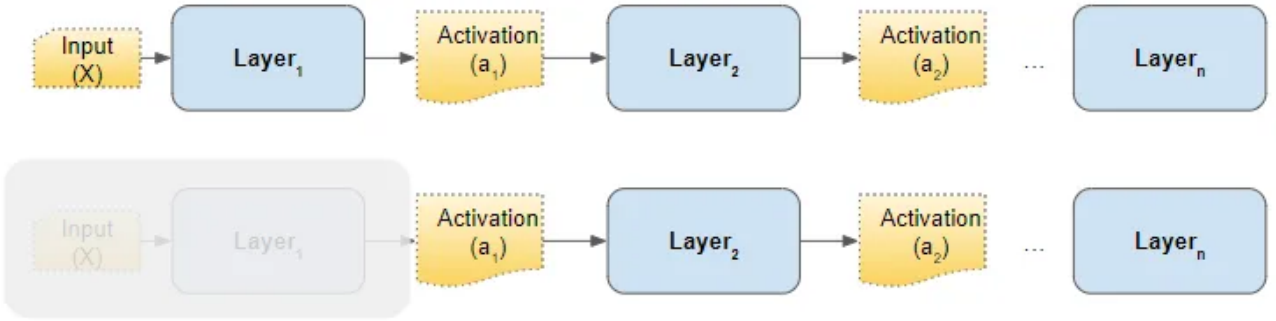






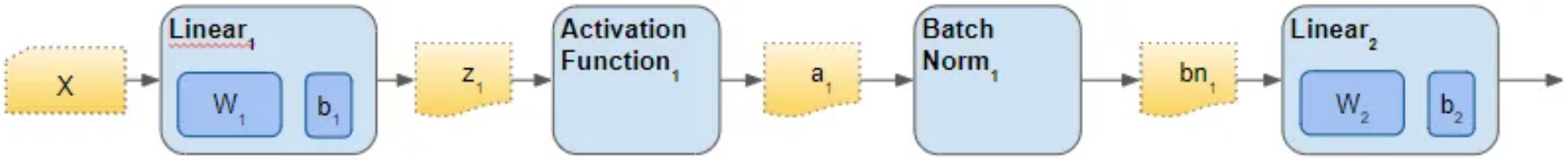
The need for batch norm

- The activations from the previous layer are simply the inputs to current layer.
- The same logic that requires us to normalize the input for the first layer will also apply to each of these hidden layers



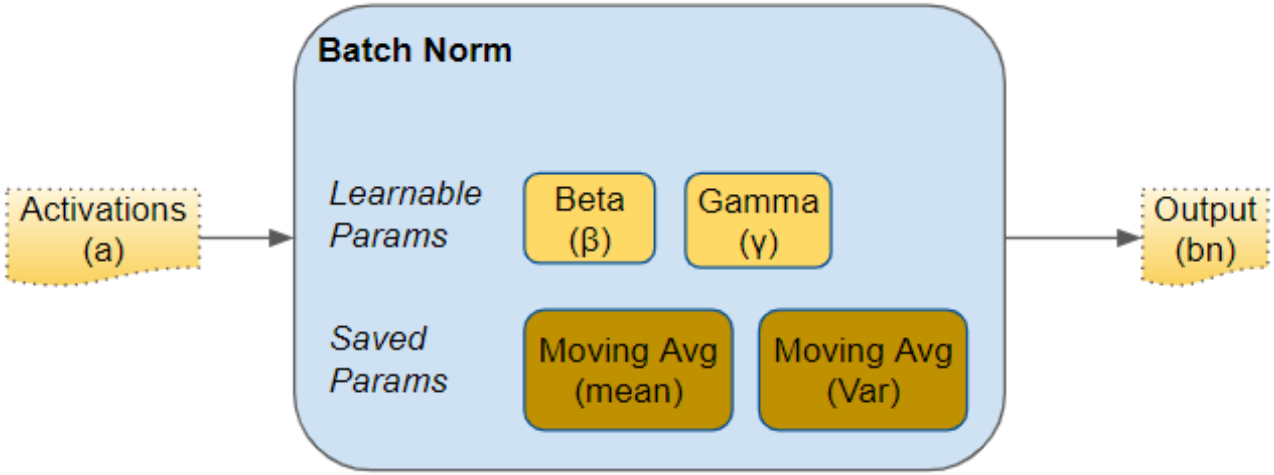
How does batch norm work?

- Batch Norm is just another network layer that gets inserted between a hidden layer and the next hidden layer
- Its job is to take the outputs from the first hidden layer and normalize them before passing them on as the input of the next hidden layer.
- Just like the parameters (eg. weights, bias) of any network layer, a Batch Norm layer also has parameters of its own:
  - Two learnable parameters called beta and gamma.



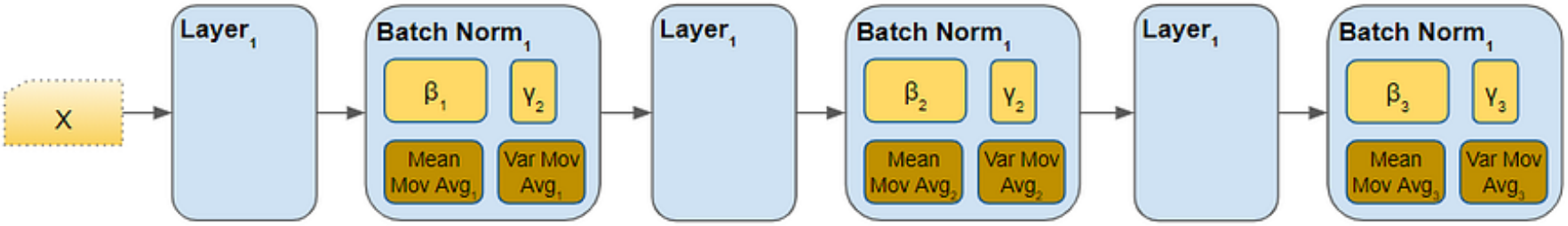
Batch norm layer

- Two learnable parameters called beta and gamma.
- Two non-learnable parameters (Mean Moving Average and Variance Moving Average) are saved as part of the ‘state’ of the Batch Norm layer.
-



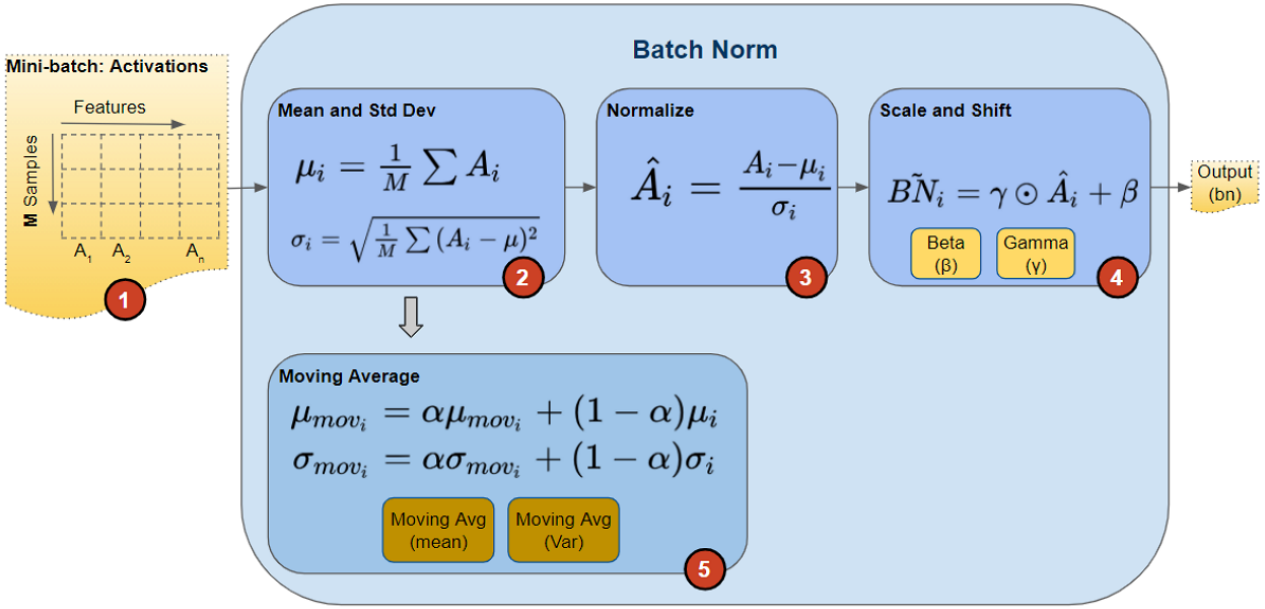
Batch norm layer

- parameters are per Batch Norm layer.
- if we have, say, three hidden layers and three Batch Norm layers in the network, we would have three learnable beta and gamma parameters for the three layers. Similarly for the Moving Average parameters.

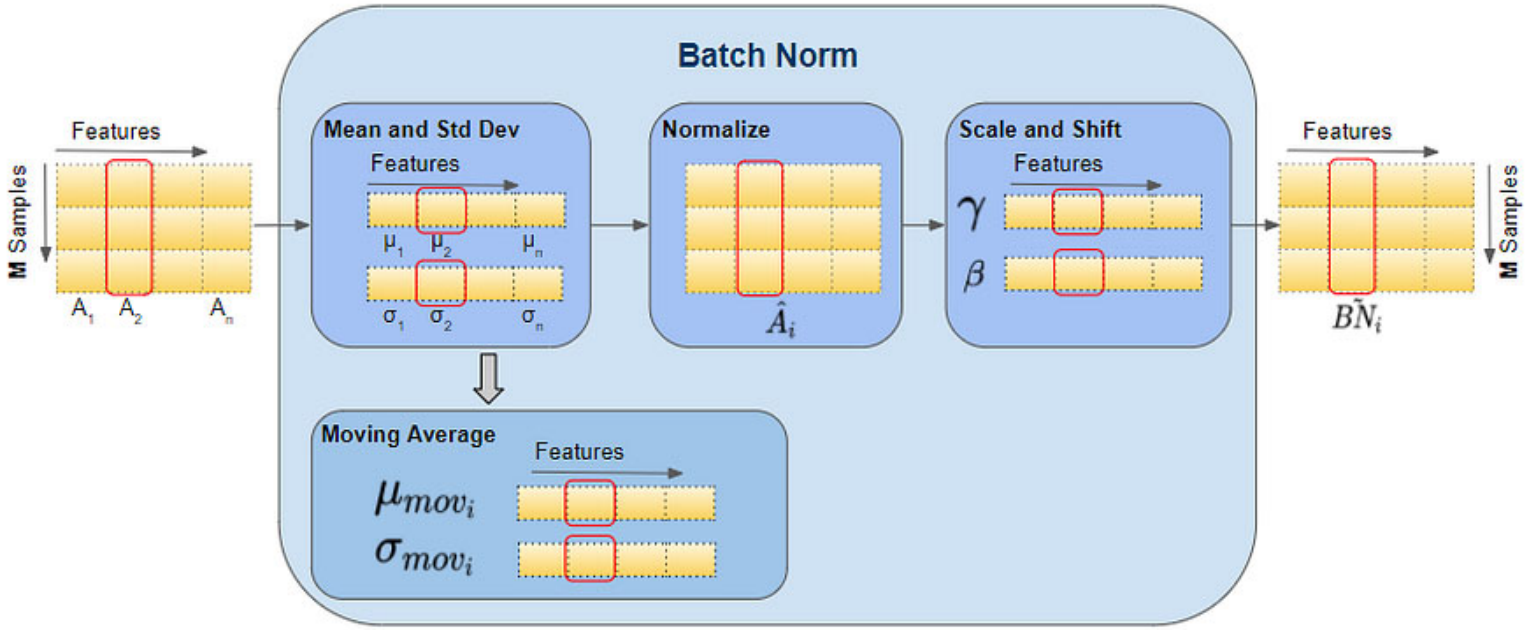


Batch norm during traing

- During training, we feed the network one mini-batch of data at a time.
- The Batch Norm layer processes its data as follows:
-



Batch norm layer vector shape



Batch norm during inference

- during Inference, we have a single sample, not a mini-batch. How do we obtain the mean and variance in that case?

Batch norm

- Pros
  - Vanishing and exploding gradient problem is reduced by a considerable amount
  - The network is much less sensitive to initial weight
  - We are able to use larger learning rates, which speeds up the training
  - It also acts as a regularizer
- Cons
  - It increase model parameter and prediction latency