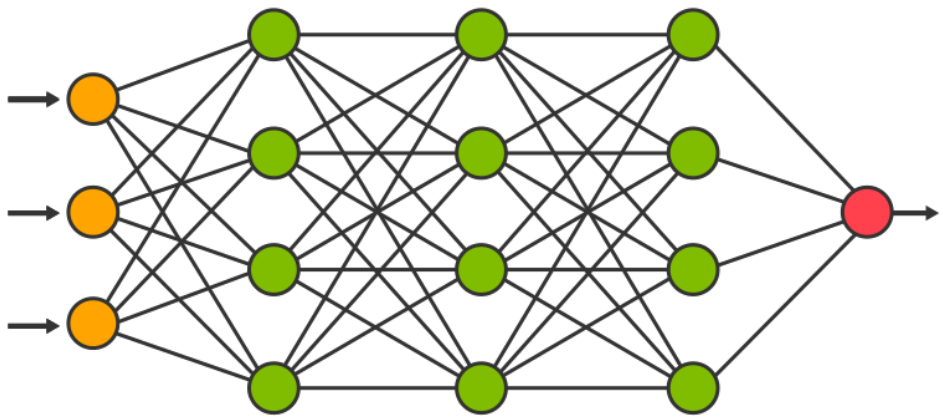
Introduction to Neural Networks

ML Instruction Team, Fall 2022

CE Department
Sharif University of Technology



Biological Analogy

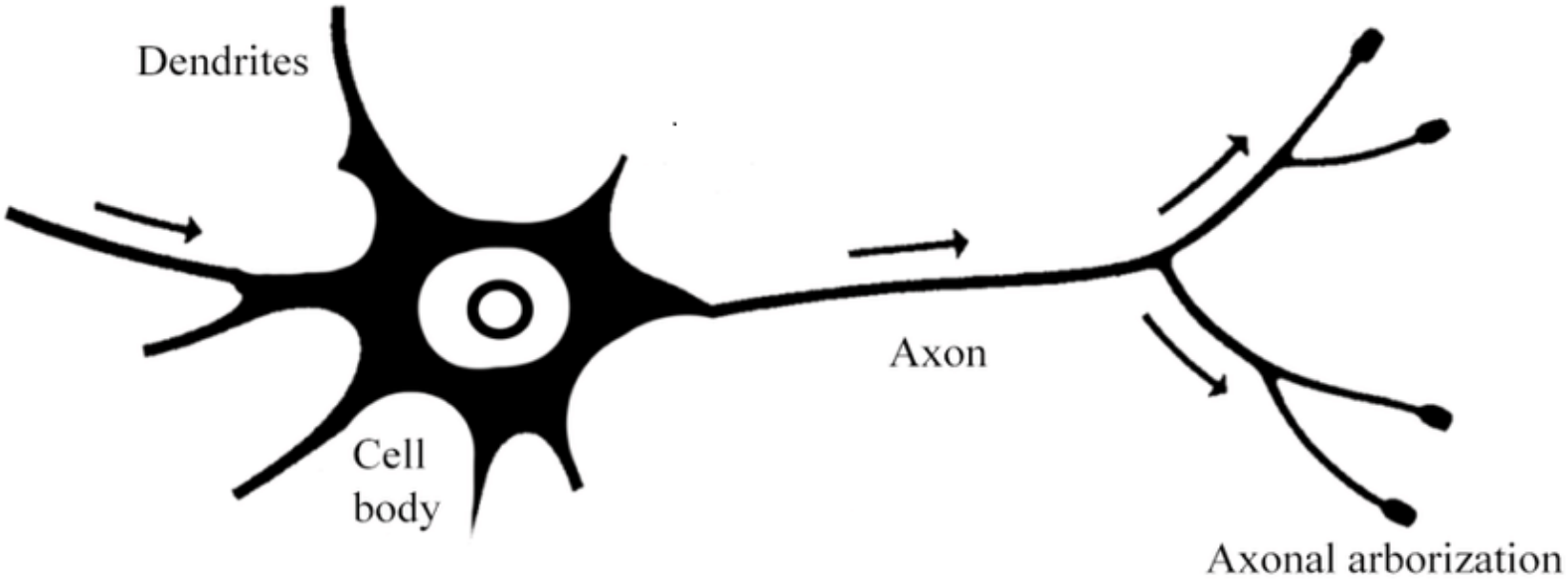


Figure: Anatomy of a biological neuron [1].

Activation Functions

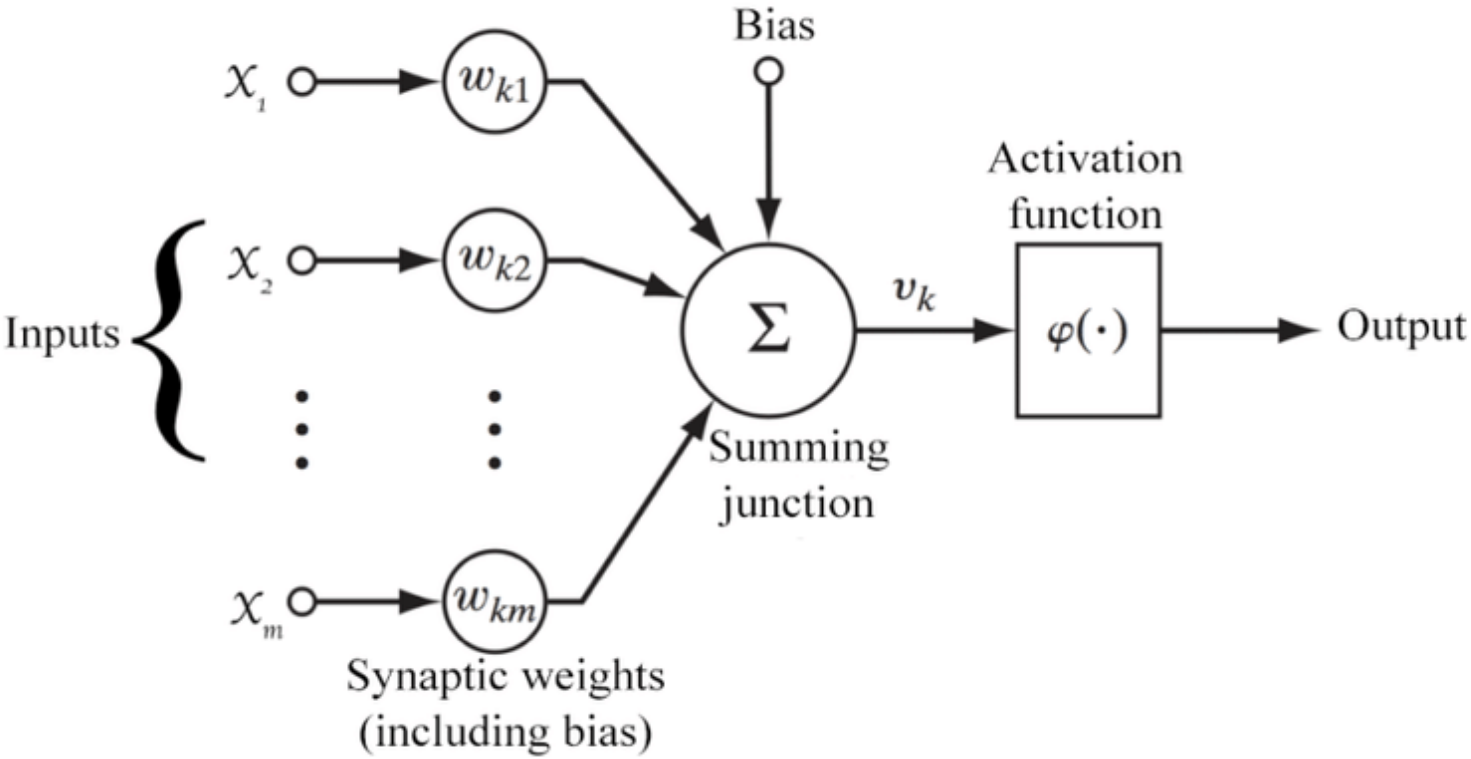


Figure: Neural network neuron [1].

Activation Functions

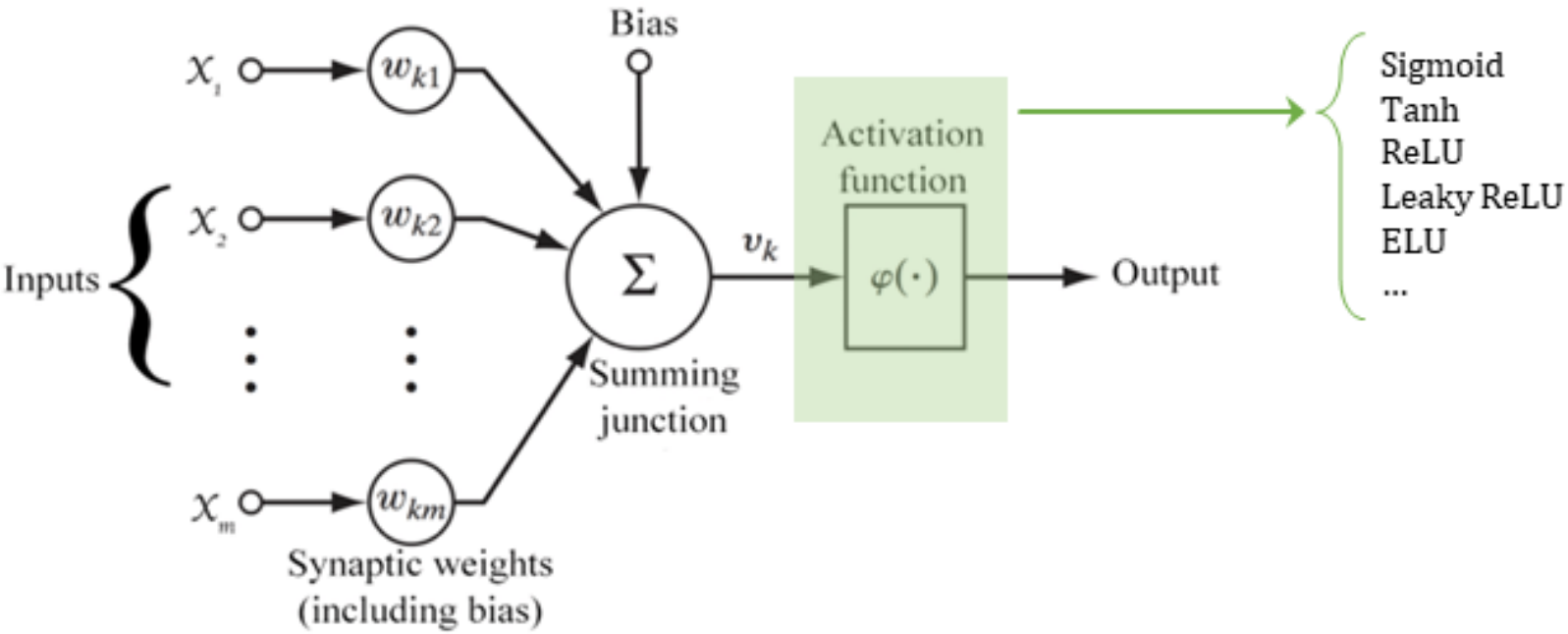
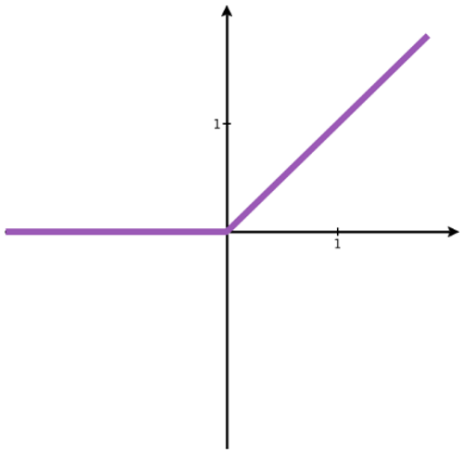
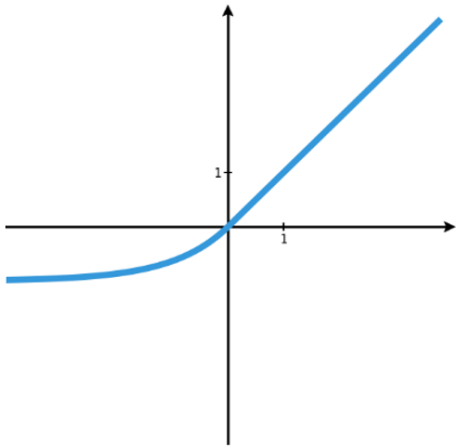
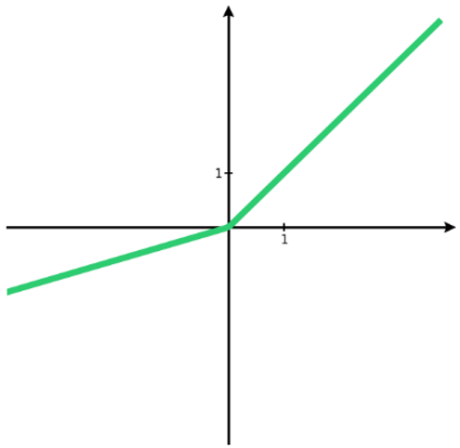
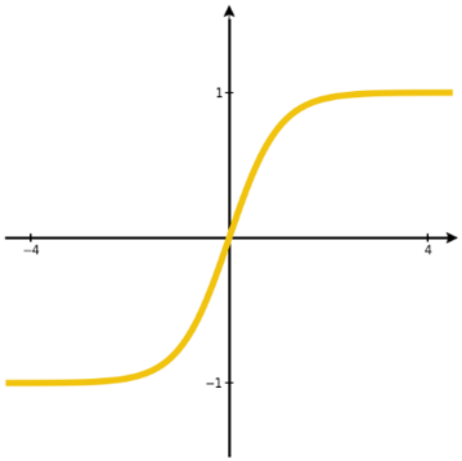
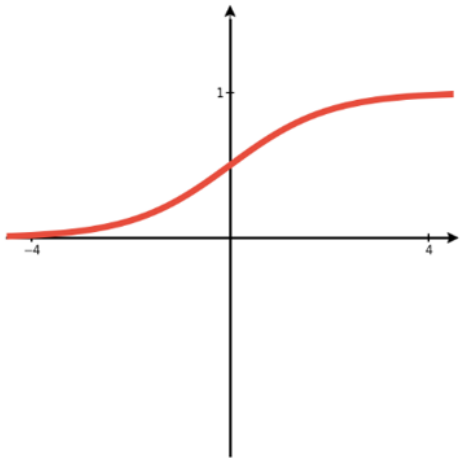
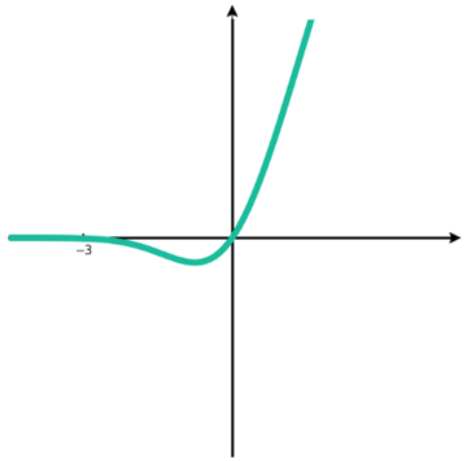


Figure: Activation function

Activation Functions

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

Activation Functions

Tanh	Sigmoid	GELU
$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$\sigma(x) = \frac{1}{1 + e^{-x}}$	$f(x) = \frac{1}{2}x\left(1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$
		

Softmax

$$f(x) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}} \quad i = 1, \cdots, J$$

Gradient Descent

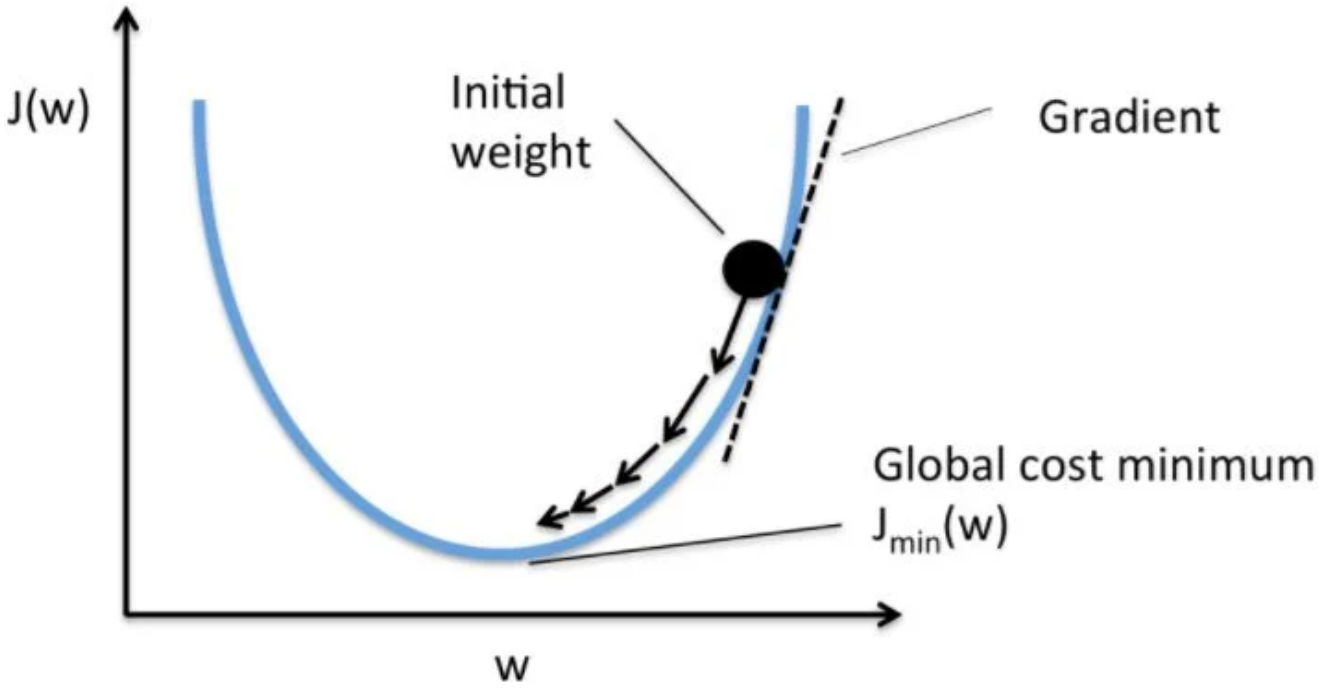


Figure: Gradient descent [2].

Gradient Descent

- Let's define our problem:
  - We have dataset $\mathcal{D} = \{x^i, y^{(i)}\}_{i=1}^n$.
  - f is a single layer perceptron.
  - Define $\hat{y}^{(i)} = f(x^{(i)})$.
- We want to minimize following cost function:

$$\mathcal{J}(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2$$

- We are going to use gradient descent algorithm. $\boldsymbol{w}$ will be updated as follows:

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \eta\nabla_{\boldsymbol{w}}\mathcal{J}$$

Gradient Descent

- Let's find $\nabla_{\boldsymbol{w}} \mathcal{J}$:

$$\begin{aligned}
\frac{\partial J}{\partial w_j} &= \frac{\partial}{\partial w_j} \frac{1}{2} \sum_i (y^{(i)} - \hat{y}^{(i)})^2 \\
&= \frac{1}{2} \sum_i \frac{\partial}{\partial w_j} (y^{(i)} - \hat{y}^{(i)})^2 \\
&= \frac{1}{2} \sum_i 2(y^{(i)} - \hat{y}^{(i)}) \frac{\partial}{\partial w_j} (y^{(i)} - \hat{y}^{(i)}) \\
&= \sum_i (y^{(i)} - \hat{y}^{(i)}) \frac{\partial}{\partial w_j} \left(y^{(i)} - \sum_j w_j x_j^{(i)} \right) \\
&= \sum_i (y^{(i)} - \hat{y}^{(i)})(-x_j^{(i)})
\end{aligned}$$

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = -\eta \sum_i (y^{(i)} - \hat{y}^{(i)})(-x_j^{(i)}) = \eta \sum_i (y^{(i)} - \hat{y}^{(i)}) x_j^{(i)}$$

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}$$

Limitations of SLP

- What are the limitations of SLP?
- Can we learn all functions with SLP?

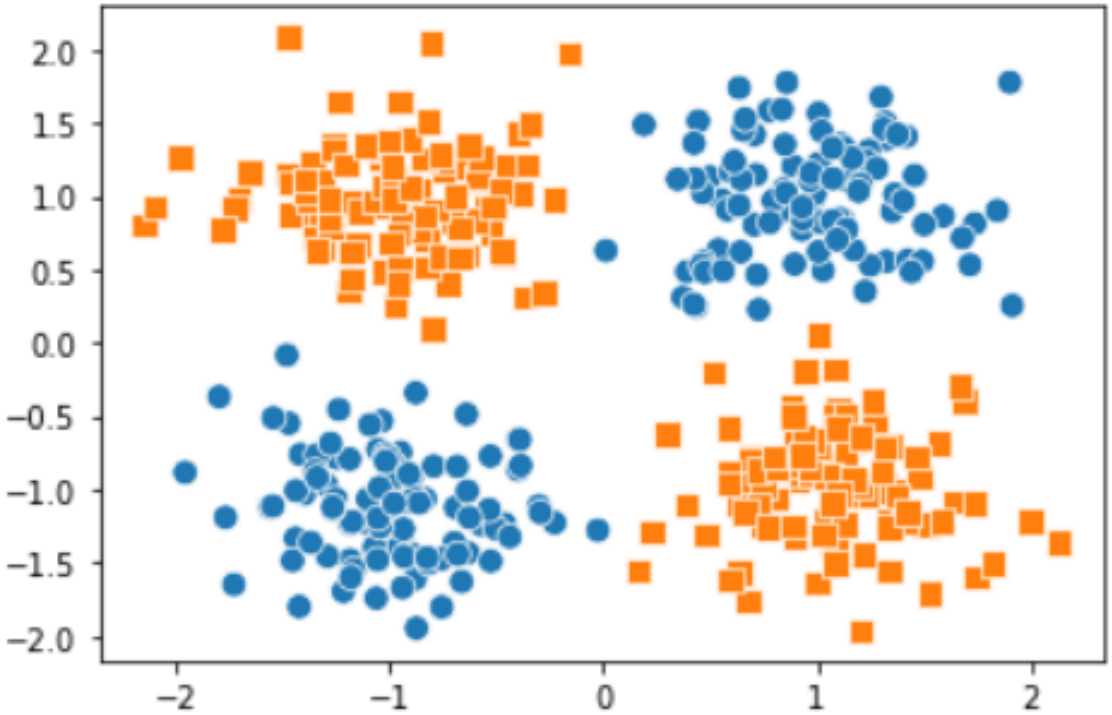


Figure: The XOR function is not linear separable.

Limitations of SLP

- As we saw in the XOR case, nonlinear separable functions can not be learned by SPLs.

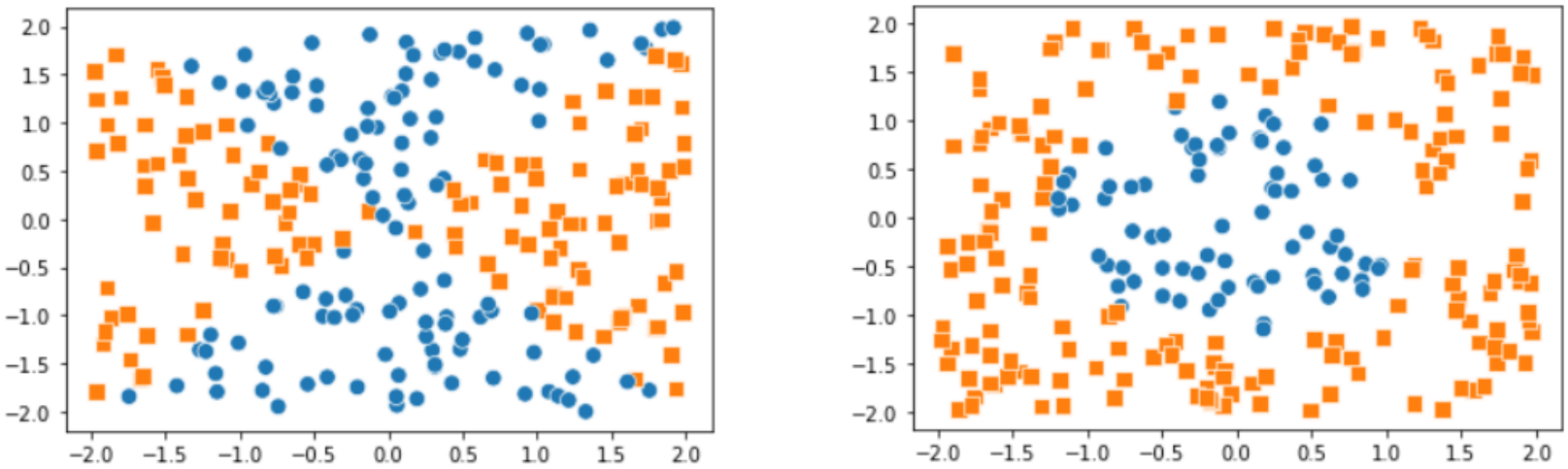


Figure: Examples of nonlinear separable functions.

- How to solve this?

Limitations of SLP

- What if we knew some feature space which our data is linear separable in?!

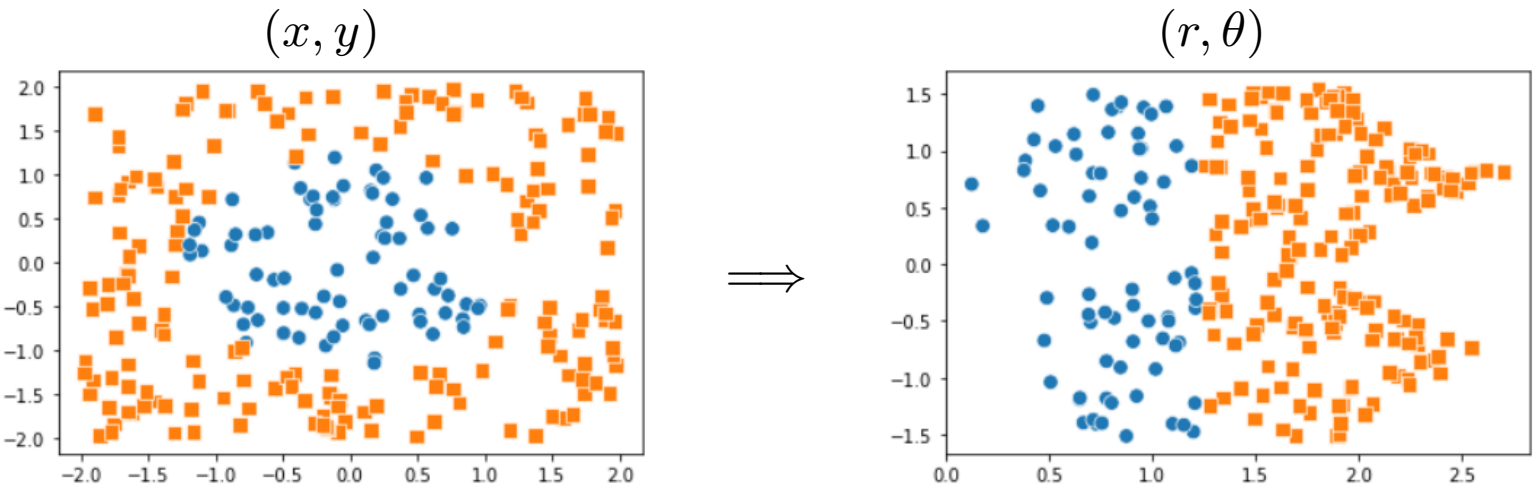


Figure: Data is linear separable after transformation.

Multi-Layer Perceptron

- So if we know some $f_1, \cdots, f_4$ we can use SLP to solve problem.

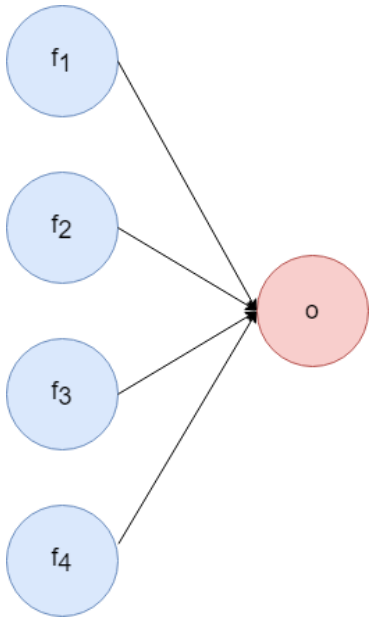


Figure: Using feature space f to solve problem.

- How to learn this f_is? Use SLP!

Multi-Layer Perceptron

- We can use inputs (x_i) to learn features (f_i)

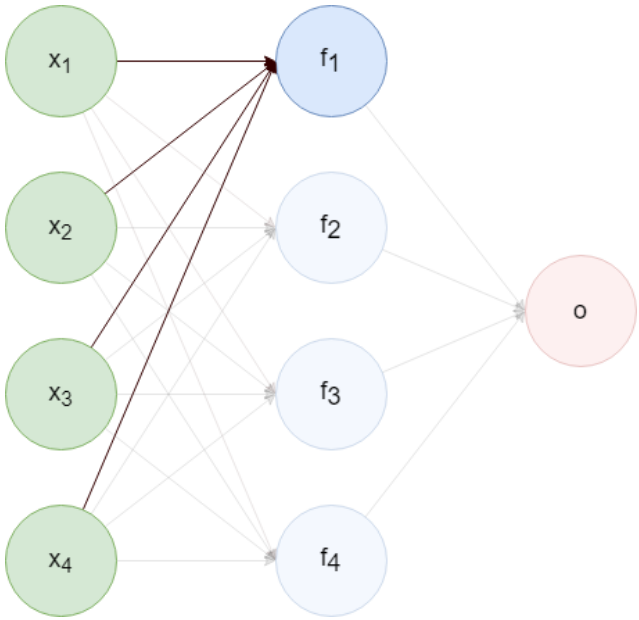


Figure: Using inputs to learn features.

- What if f_is are not sufficient? We can add more layers!

Multi-Layer Perceptron

- Adding more layers we will have a bigger network.

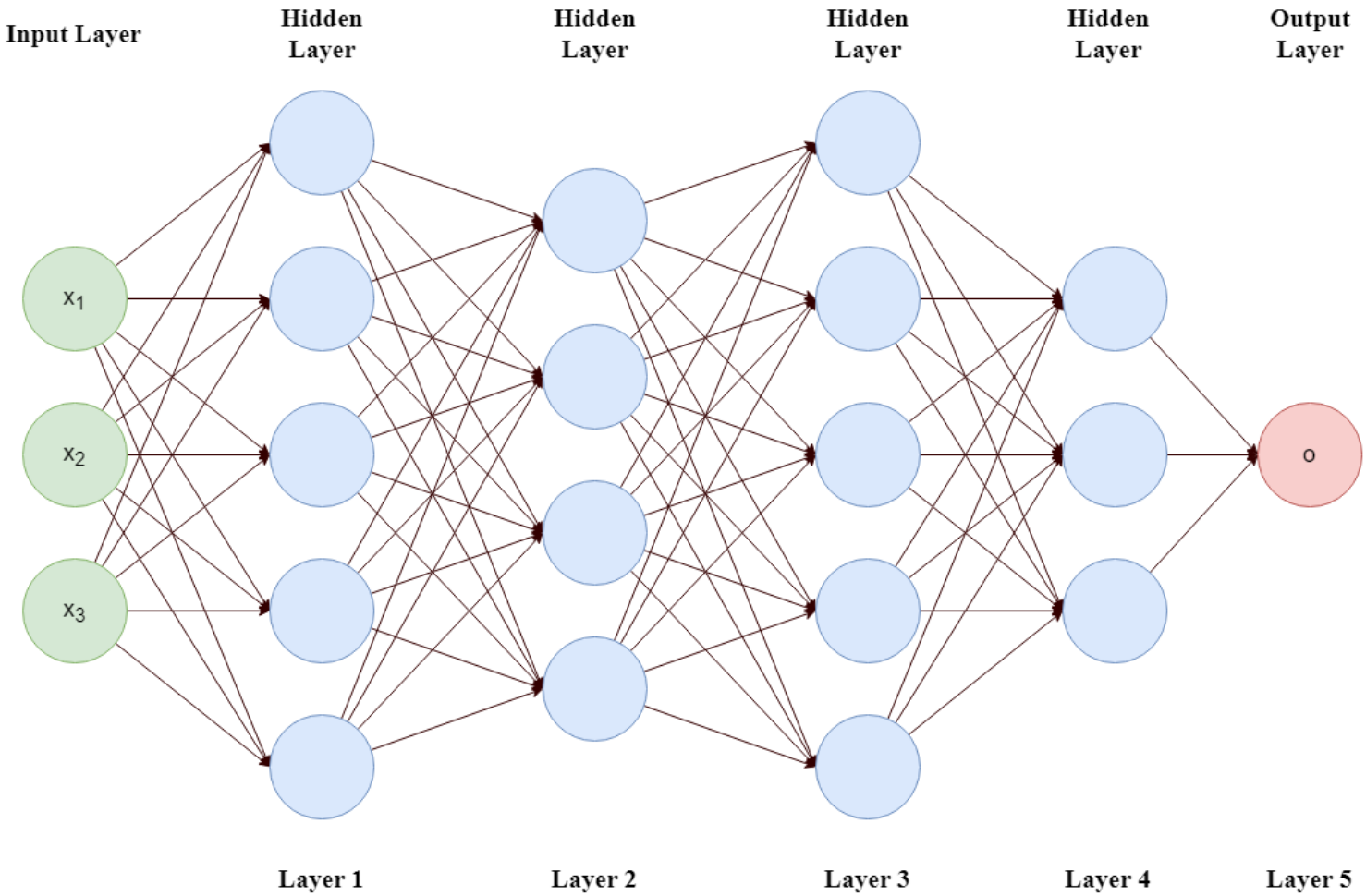


Figure: A 5 layer MLP.

Architecture of MLPs

- Important questions:
  - How many hidden layer should we have?
  - In each hidden layer, how many neuron should we have?

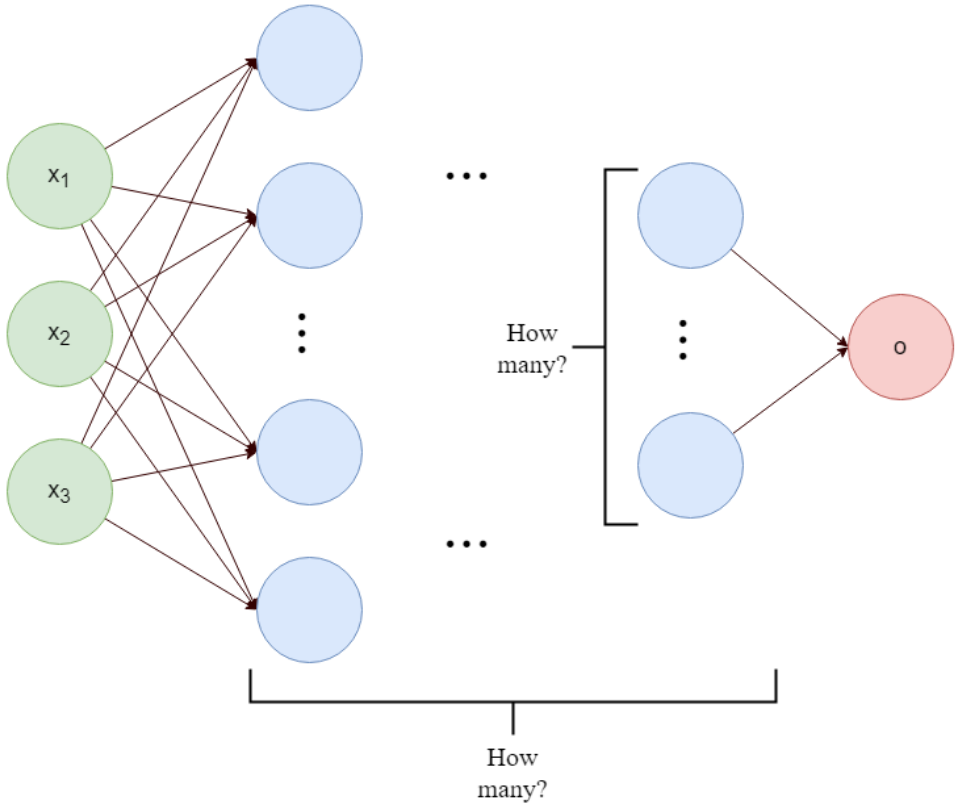


Figure: How many layers and neurons should we have?

Architecture of MLPs

- In practice:
  - You have limited resources
  - There is no universal rule to choose this hyperparameters
  - Need to experiment for different number of layers and neurons in each layer

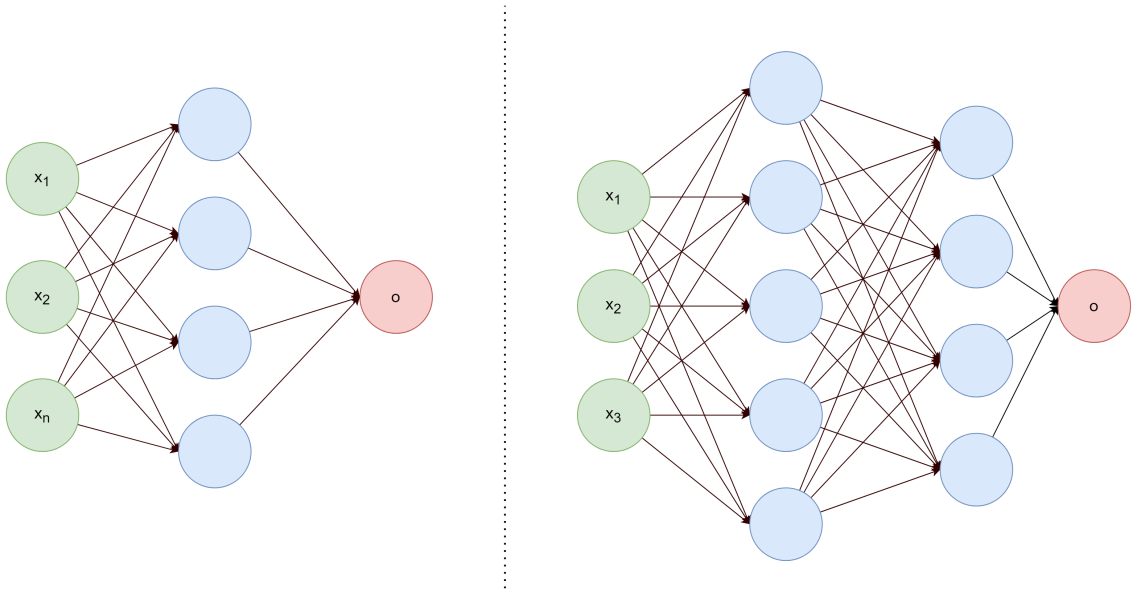


Figure: Experiment for different architecture and choose the best model.

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

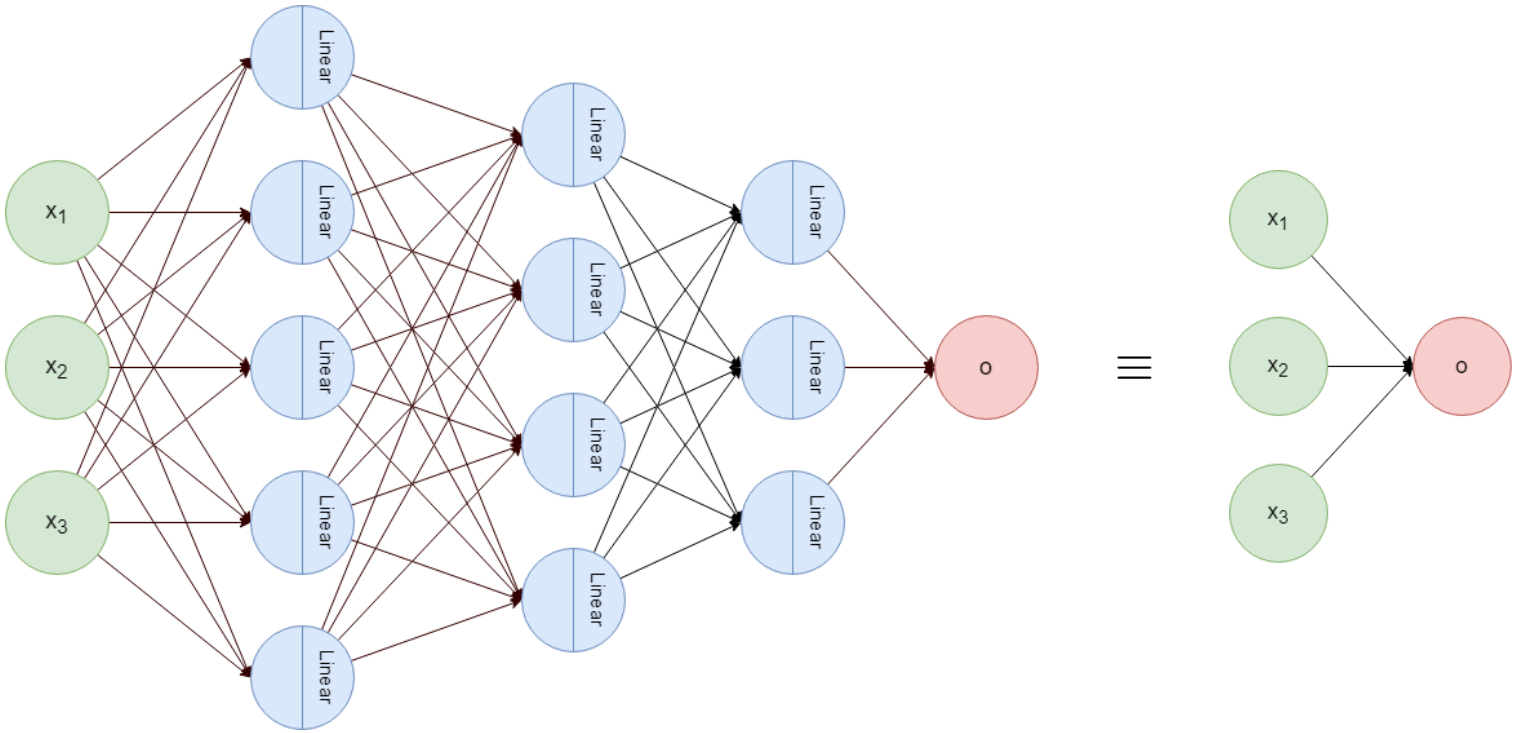


Figure: MLP with linear activation functions is equivalent to simple SLP.

XOR problem

- Now let's solve XOR problem with MLPs.
- We have two binary inputs, build an MLP to calculate their **XOR**.

- First let's build logical **AND** and **OR** functions.

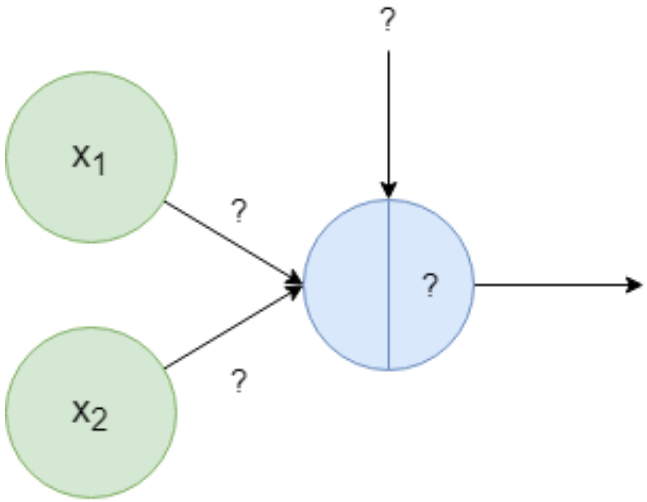
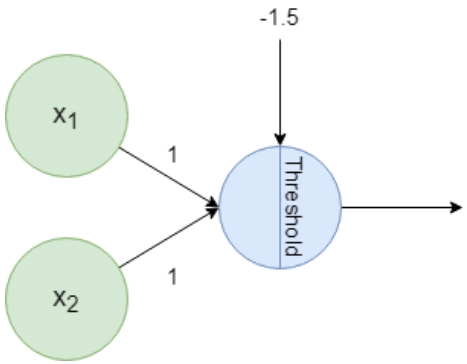
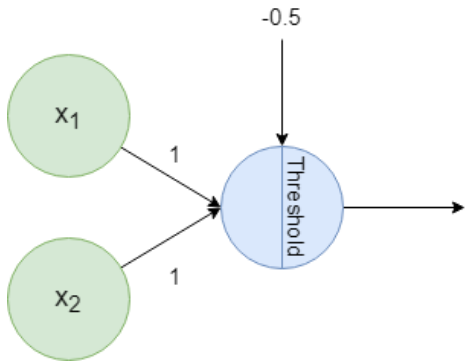


Figure: We need to find weights, biases and activation function.

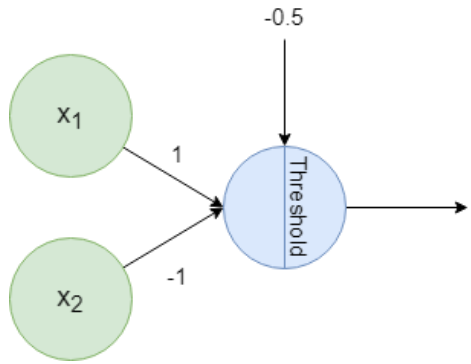
XOR problem



(a) $x_1 \wedge x_2$

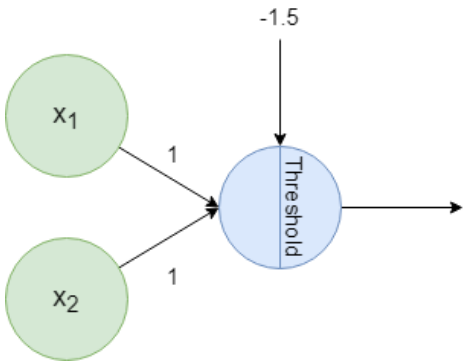


(b) $x_1 \vee x_2$

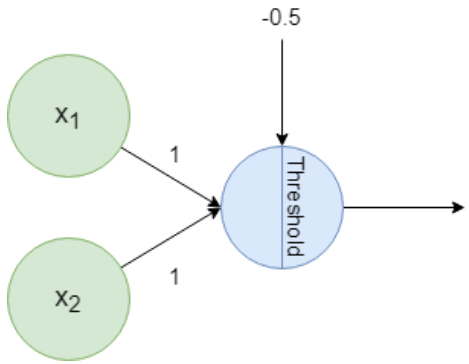


(c) $x_1 \wedge \overline{x_2}$

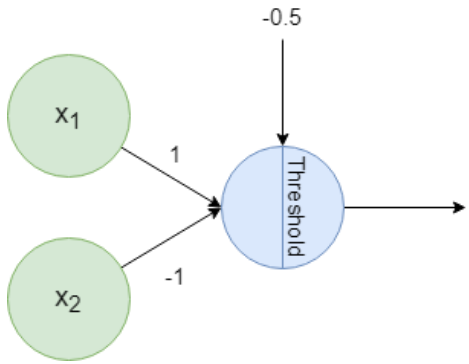
XOR problem



(a) $x_1 \wedge x_2$



(b) $x_1 \vee x_2$



(c) $x_1 \wedge \overline{x_2}$

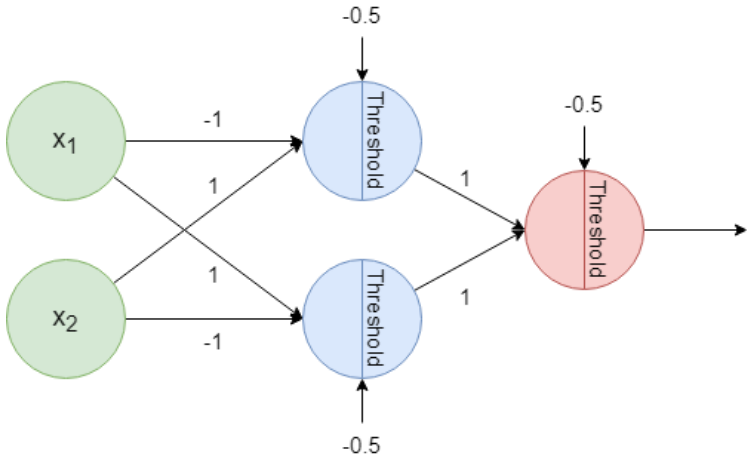
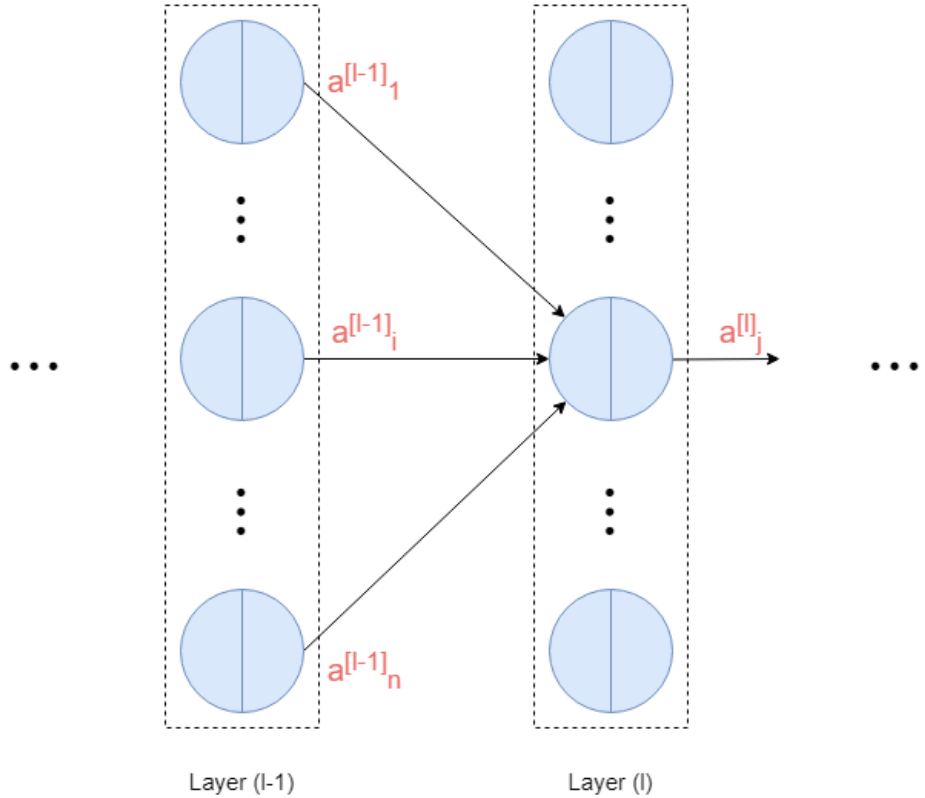


Figure: MLP for XOR function.

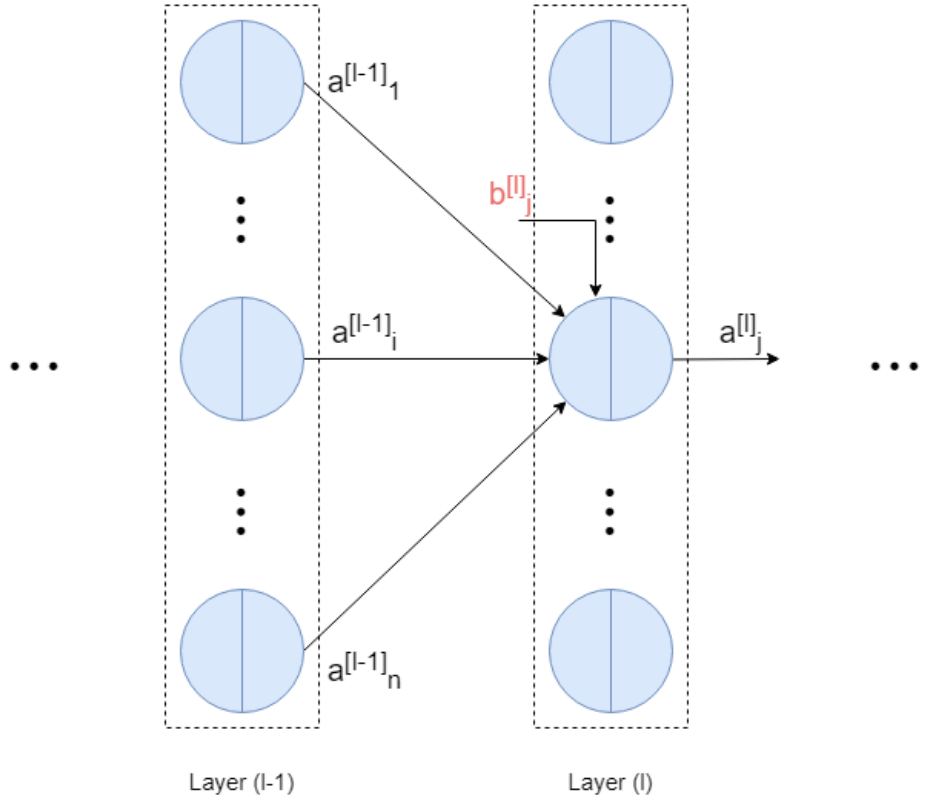
MLP notation

- $a_i^{[l]}$: i-th neuron outpu in layer l
- $\boldsymbol{a}^{[l]}$: layer l output in vector form



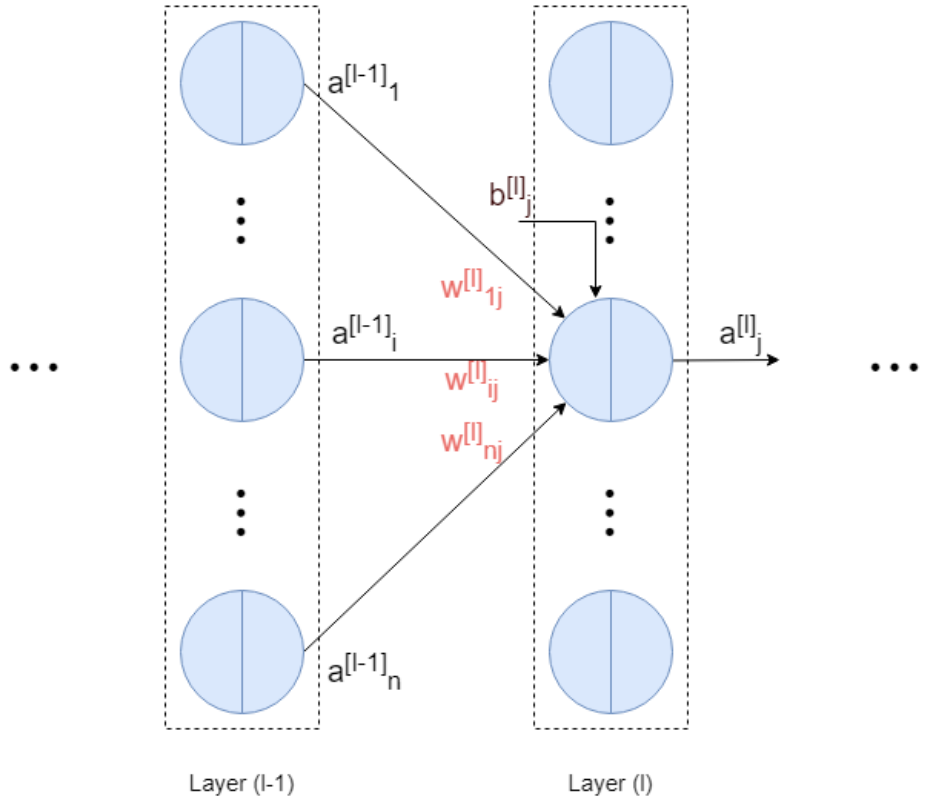
MLP notation

- $b_i^{[l]}$: i-th neuron biase in layer l
- $\boldsymbol{b}^{[l]}$: layer l biases in vector form



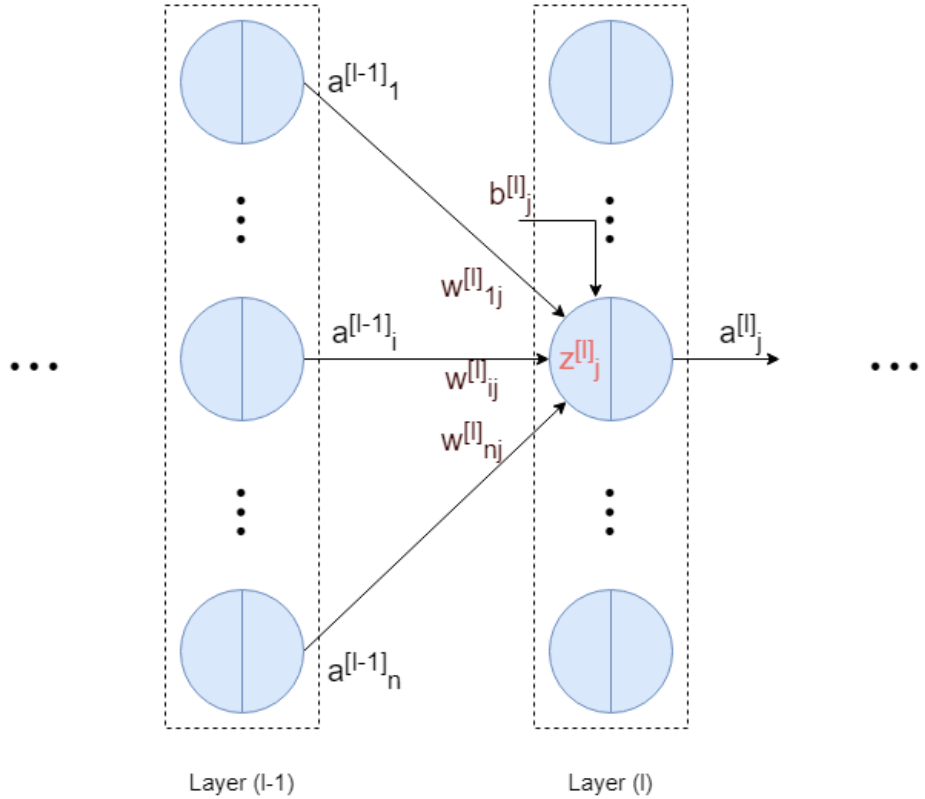
MLP notation

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



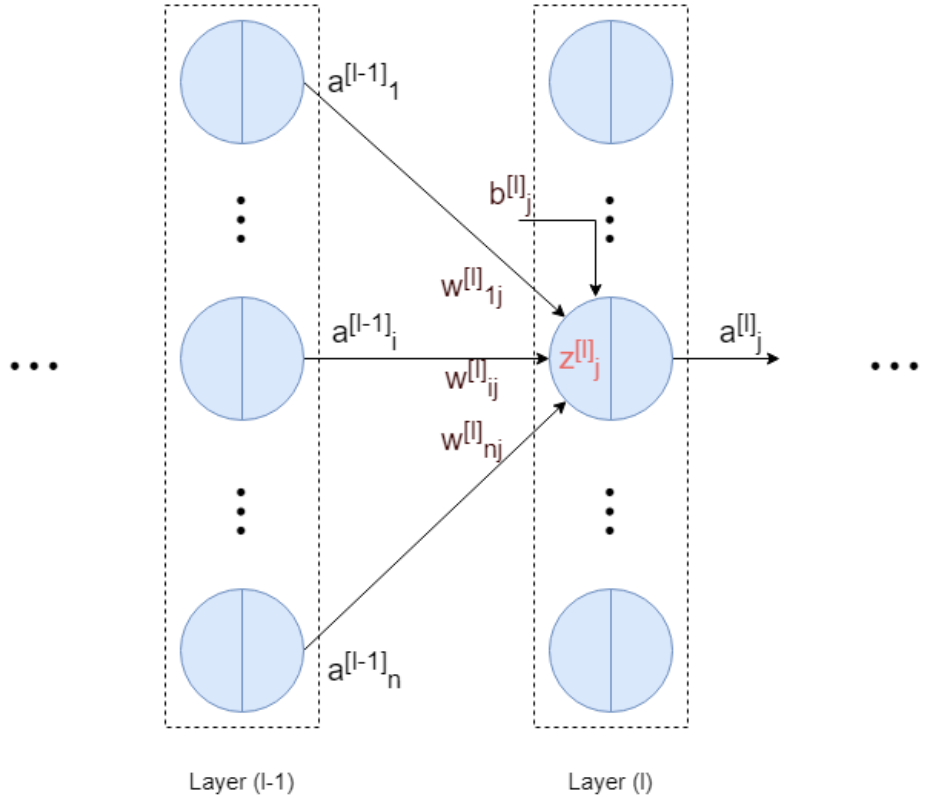
MLP notation

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



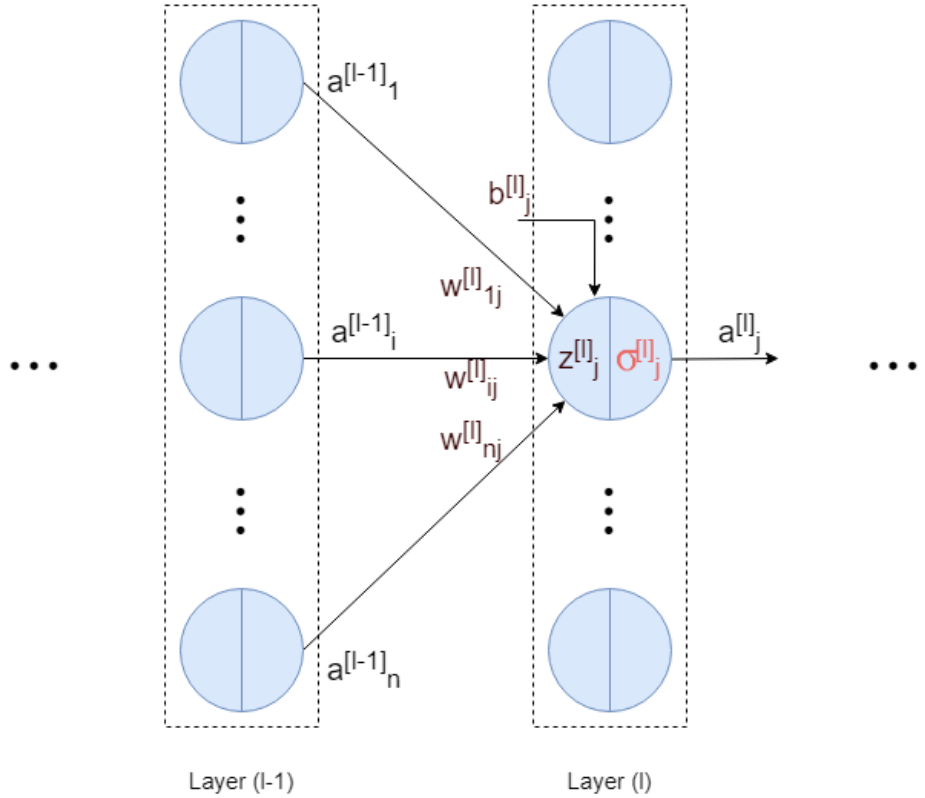
MLP notation

- $\boldsymbol{z}^{[l]}$: input of layer l in vector form
- $\boldsymbol{z}^{[l]} = \boldsymbol{b}^{[l]} + (W^{[l]})^T \boldsymbol{a}^{[l-1]}$



MLP notation

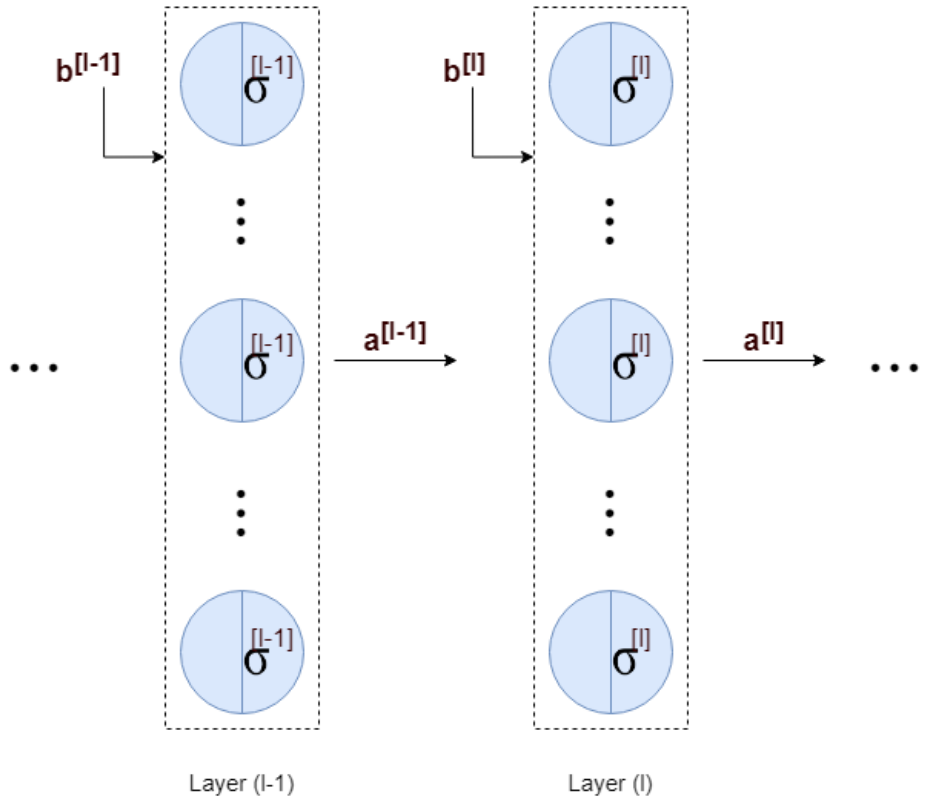
- $\sigma_j^{[l]}$: j-th neuron activation function in layer l



MLP notation

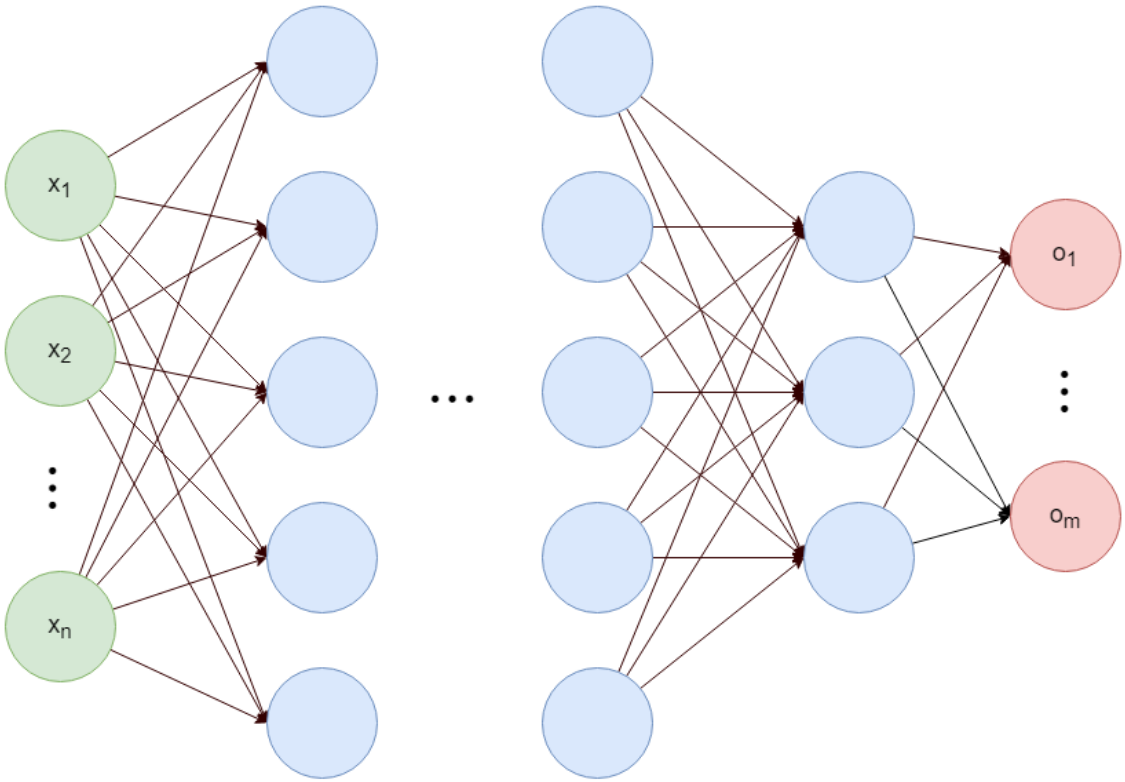
- If we assume all neurons in one layer have the same activation function then:

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



MLP notation

- So for a network with L layer, and $\boldsymbol{x}$ as its input we will have:

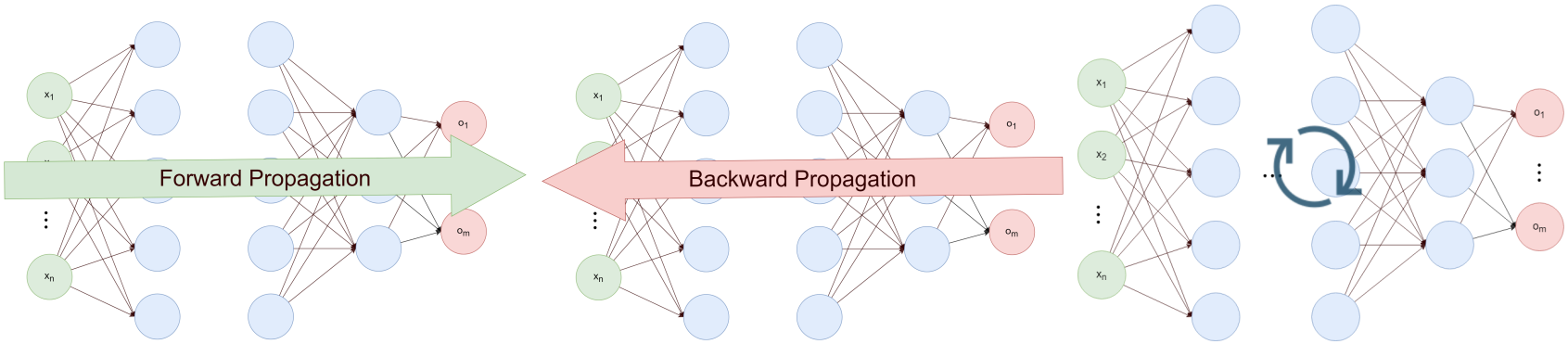


$$\boldsymbol{o} = \boldsymbol{a}^{[L]} = \sigma^{[L]}\left(\boldsymbol{b}^{[L]} + (W^{[L]})^T \sigma^{[L-1]}\left(\cdots \sigma^{[1]}\left(\boldsymbol{b}^{[1]} + (W^{[1]})^T \boldsymbol{x}\right)\cdots\right)\right)$$

Learning MLPs

- Till here we have used networks with predefined weights and biases.
- How to learn these parameters?

- 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.

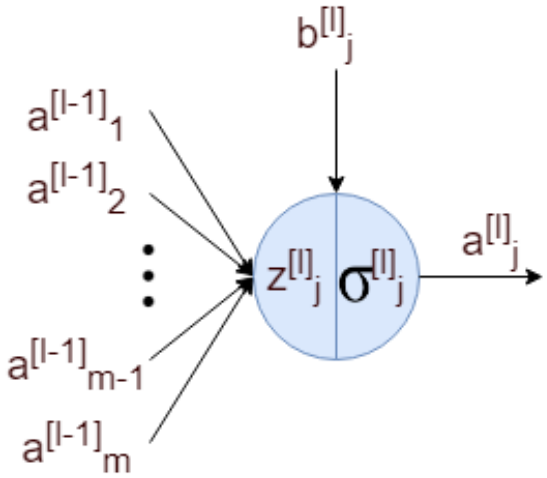


Figure: $a_j^{[l]} = \sum_{i=1}^{m} W_{ij}^{[l]} a_i^{[l-1]} + b_j^{[l]}$

- So we can calculate these outputs layer by layer.

Forward Propagation

- After forward pass we will know:
  - Loss value
  - Network output
  - Middle values

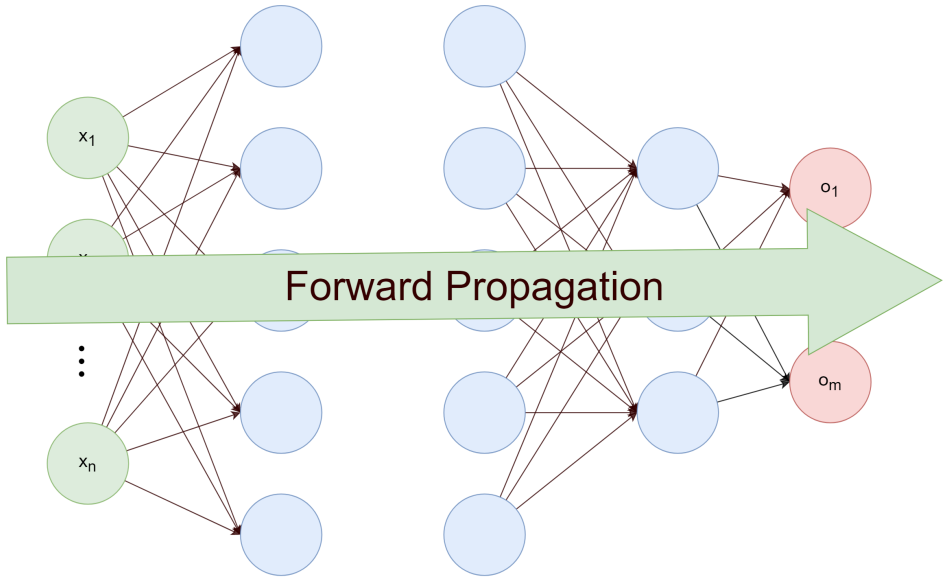


Figure: Forward pass

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

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.

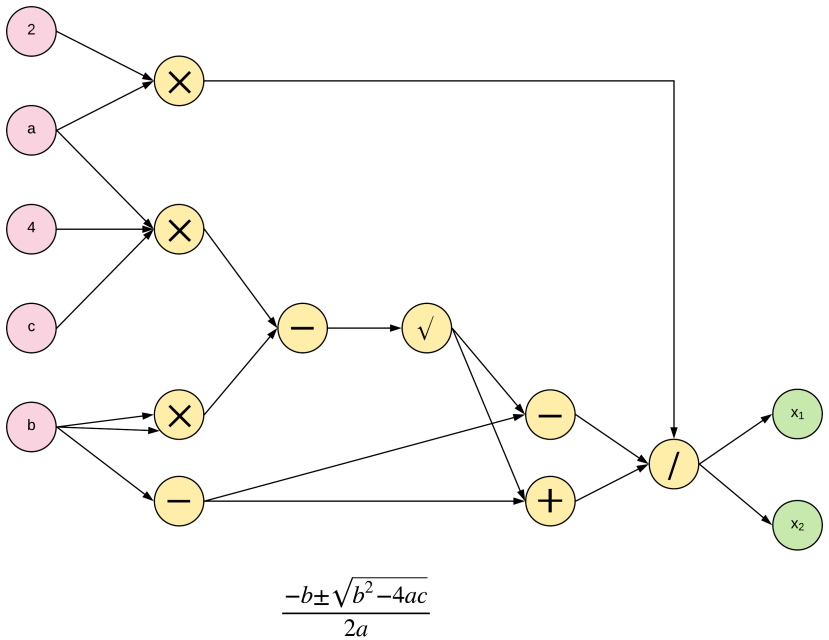
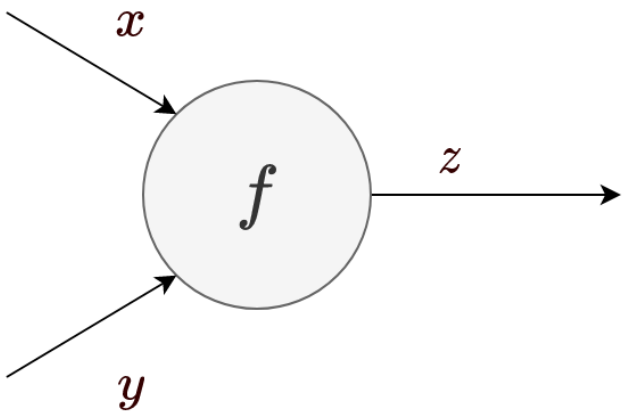


Figure: An example of computational graph, Source.

Backward Propagation

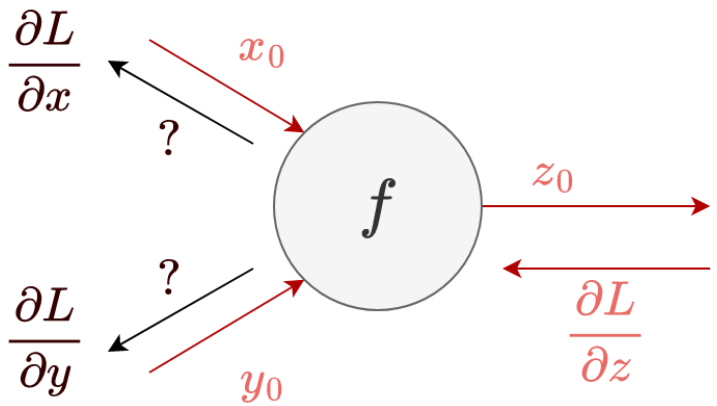
- In this approach if we know how to calculate gradient for single node or module, then we can find gradient with respect to each variables.
- Let's say we have a module as follow:



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

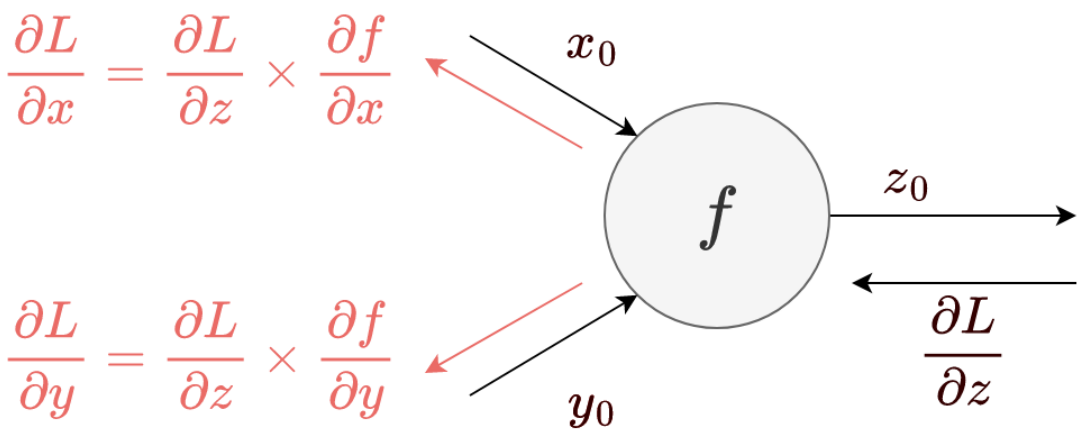


Backward Propagation

- We can use chain rule to do so.

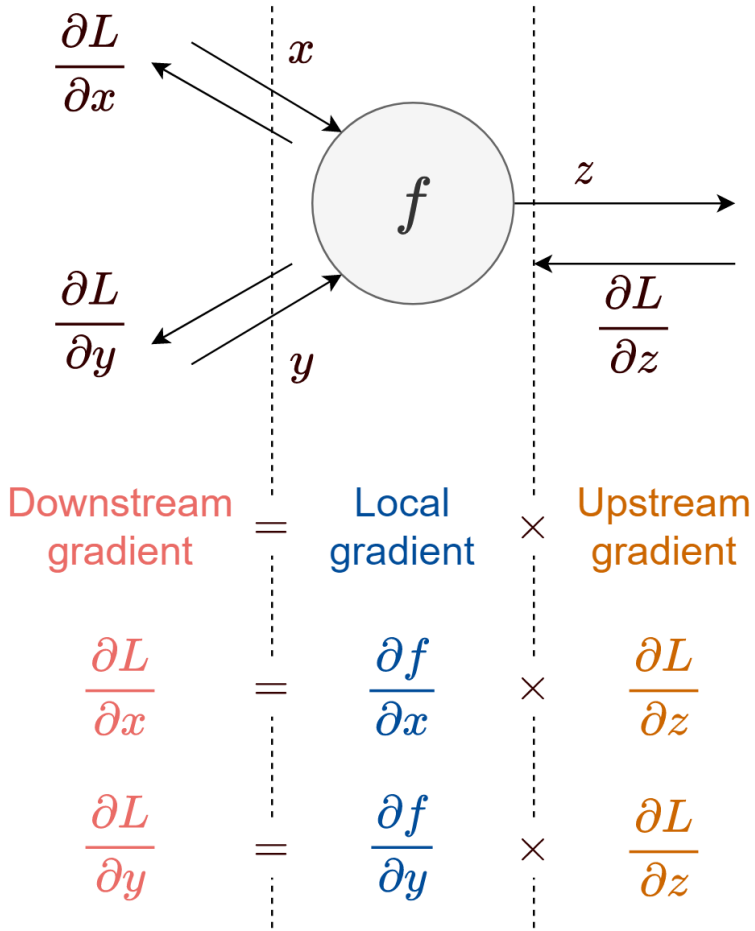
Chain rule:

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



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 $z = 2$.

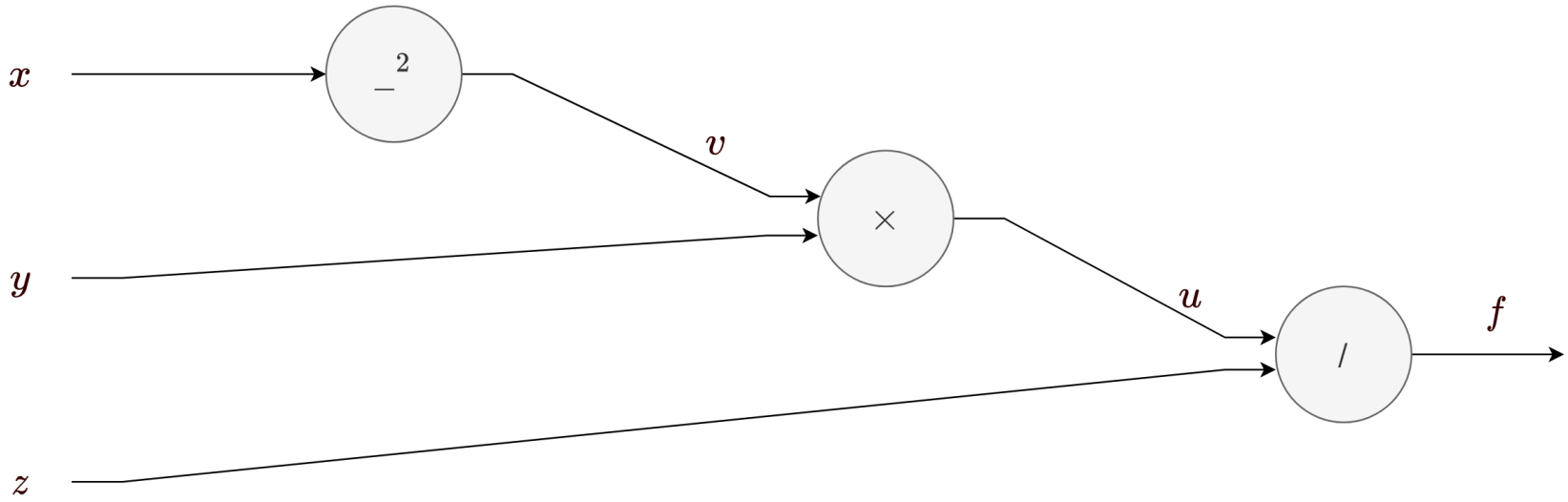


Figure: Computational graph of f.

Backward Propagation: Example

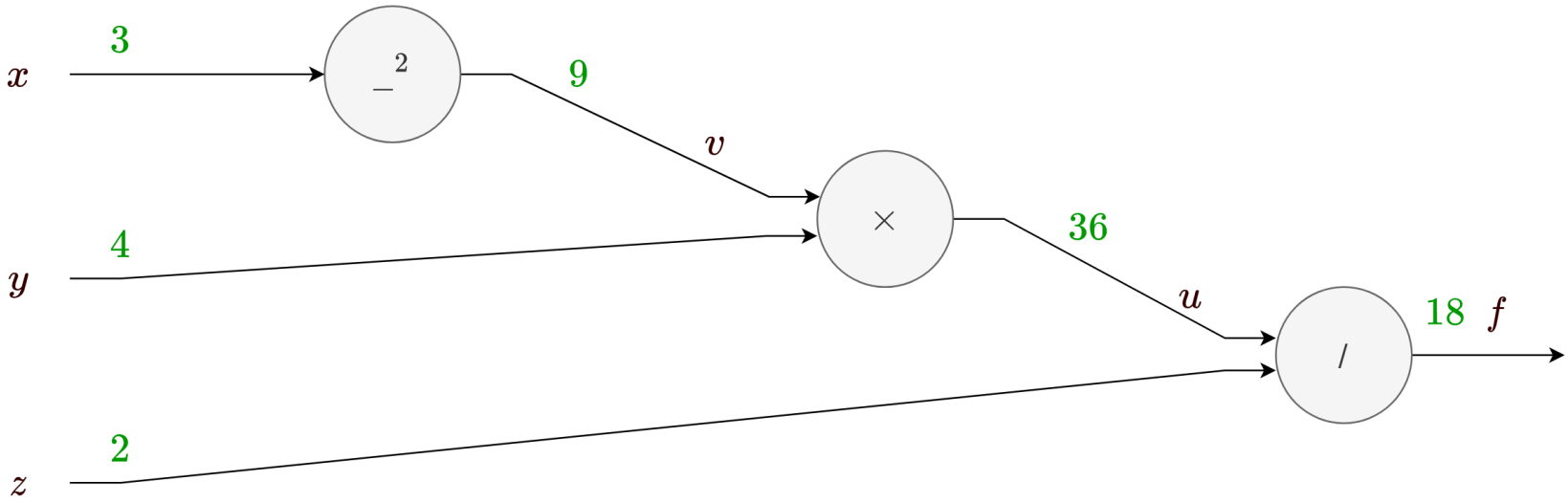
- First let's find gradient analytically.
- We have:

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

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

Backward Propagation: Example

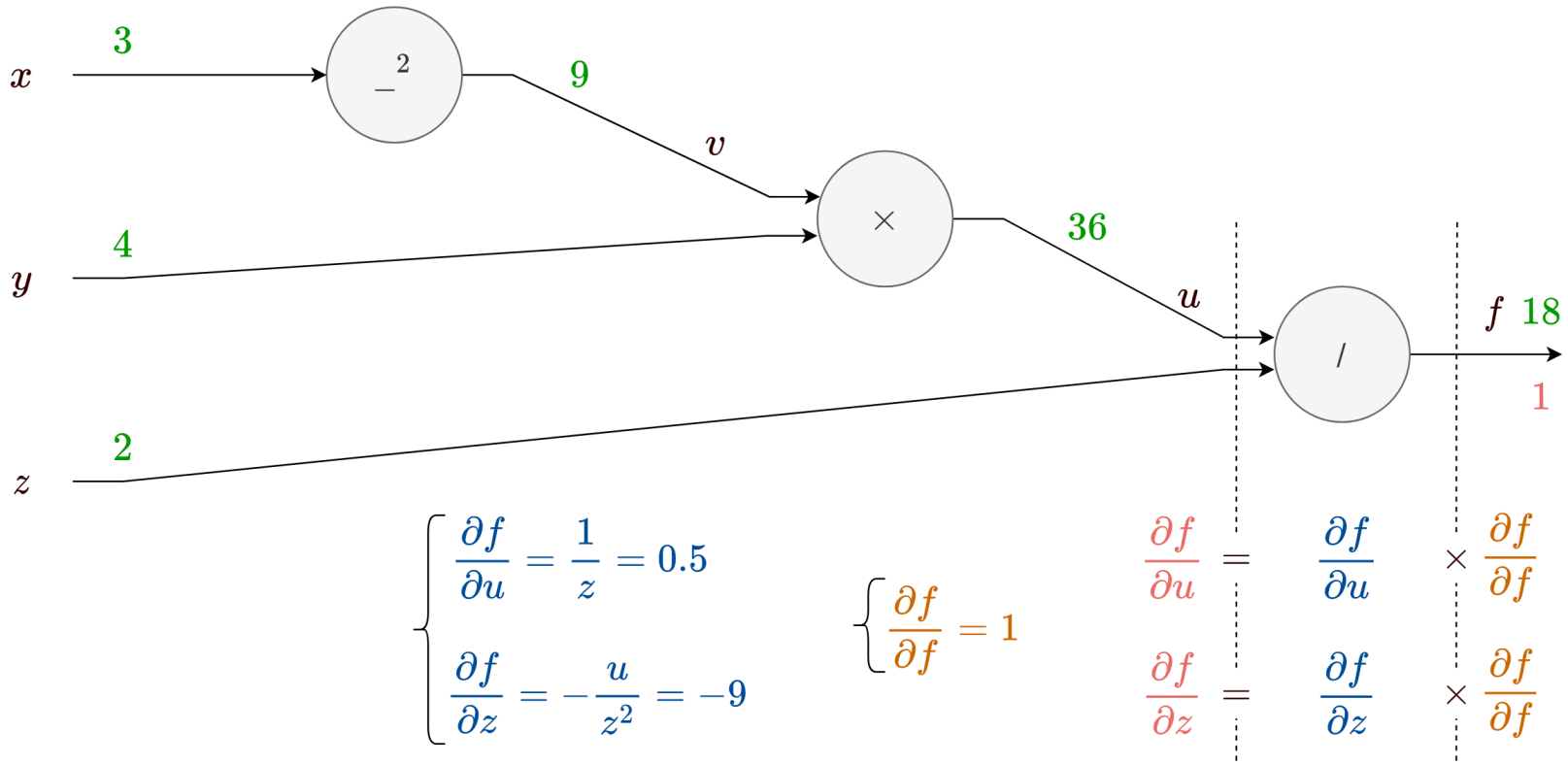
- Now let's use backpropagation.
- First we do forward propagation.



- Second we will do backpropagation for each module.

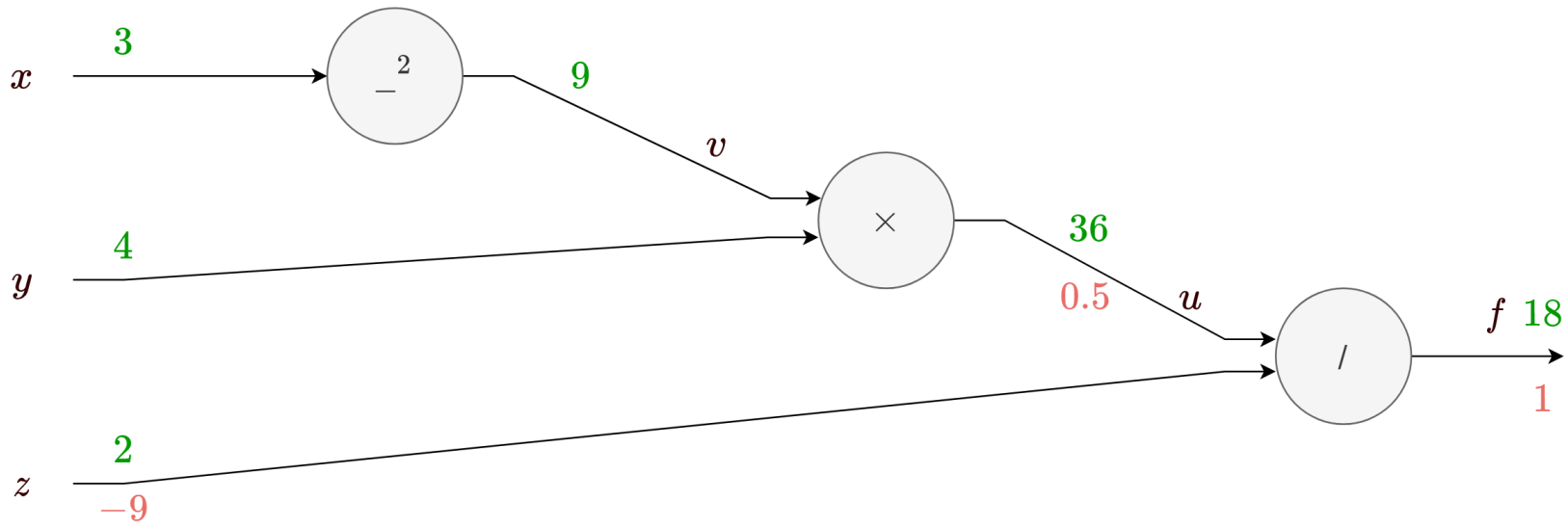
Backward Propagation: Example

- Backpropagation for / module:



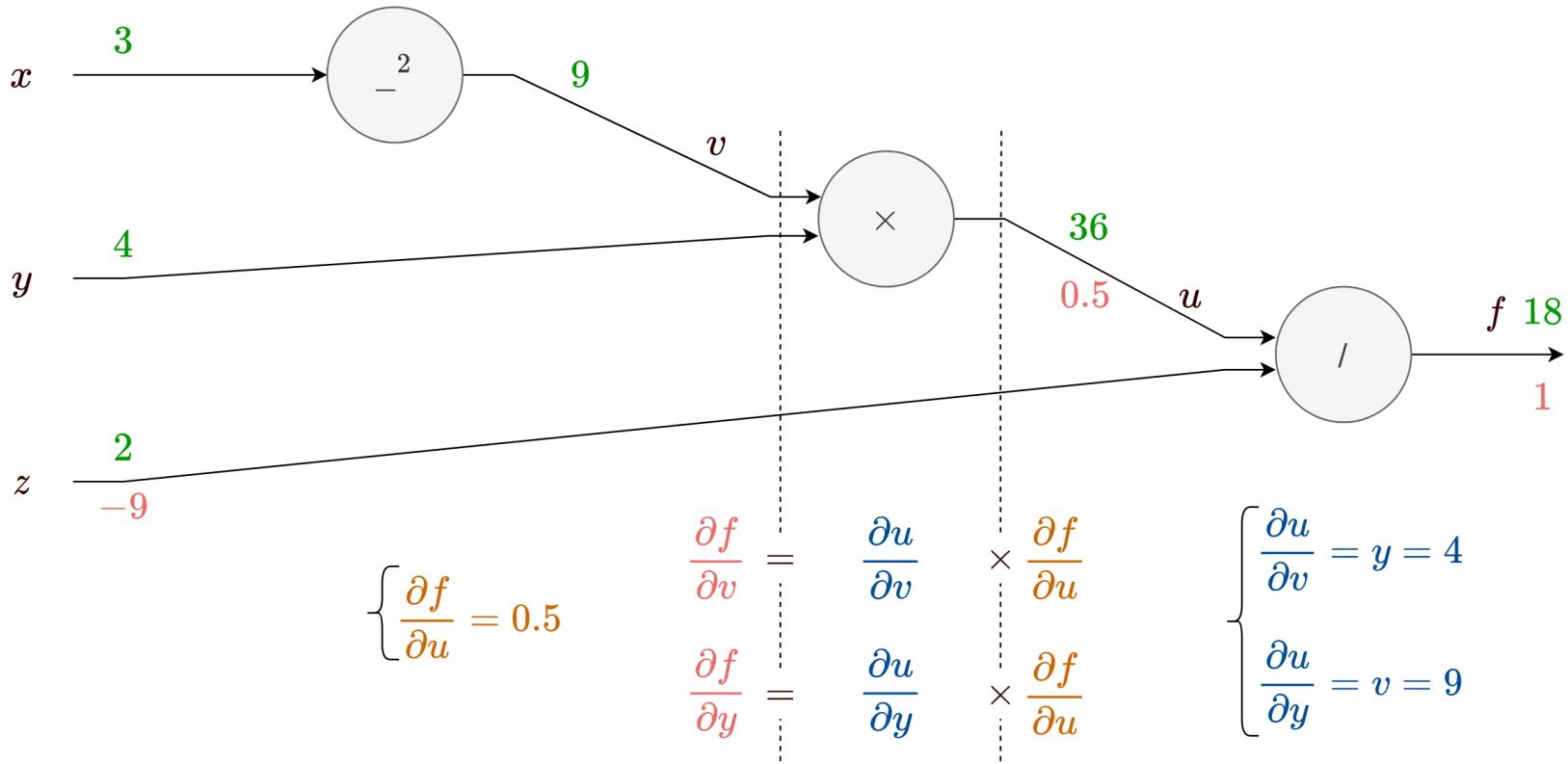
Backward Propagation: Example

- Backpropagation for / module:



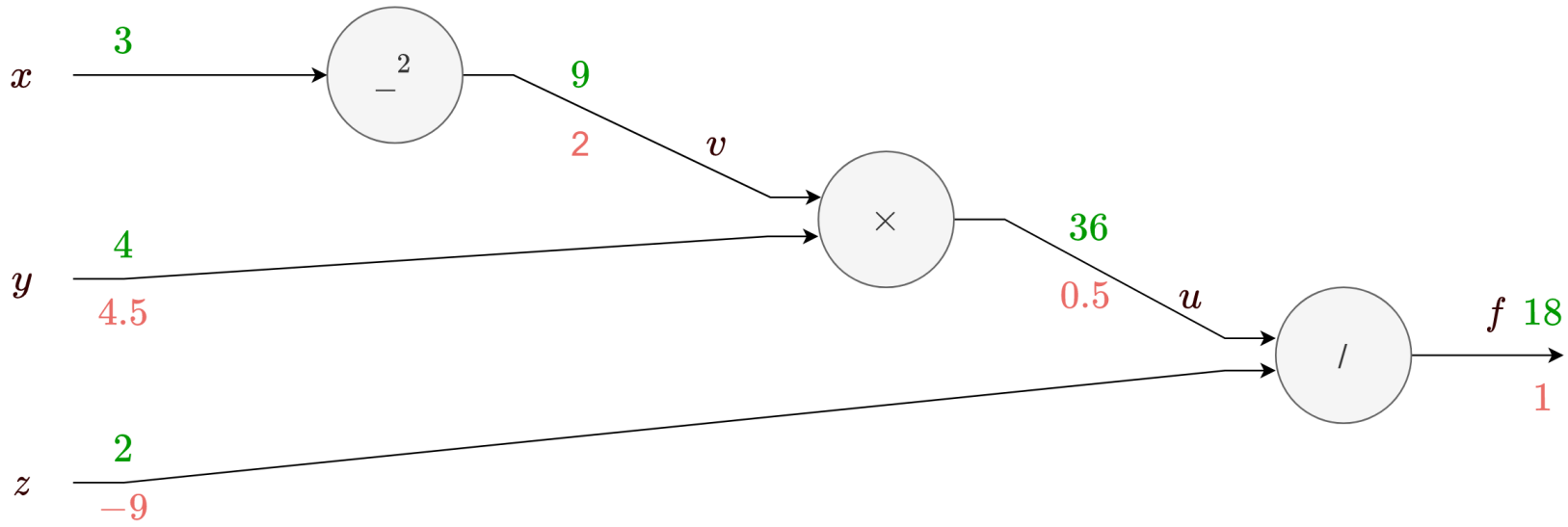
Backward Propagation: Example

- Backpropagation for $\times$ module:



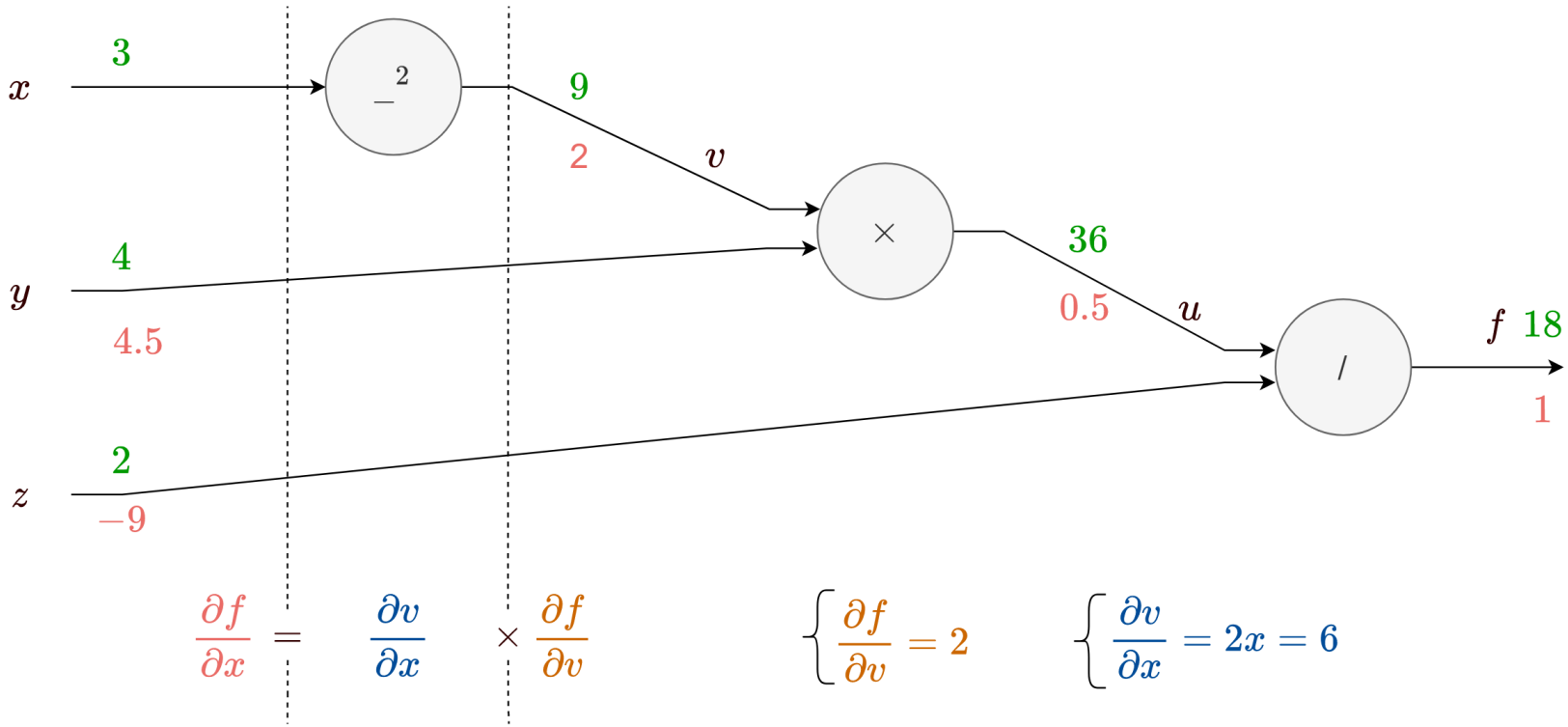
Backward Propagation: Example

- Backpropagation for $\times$ module:



Backward Propagation: Example

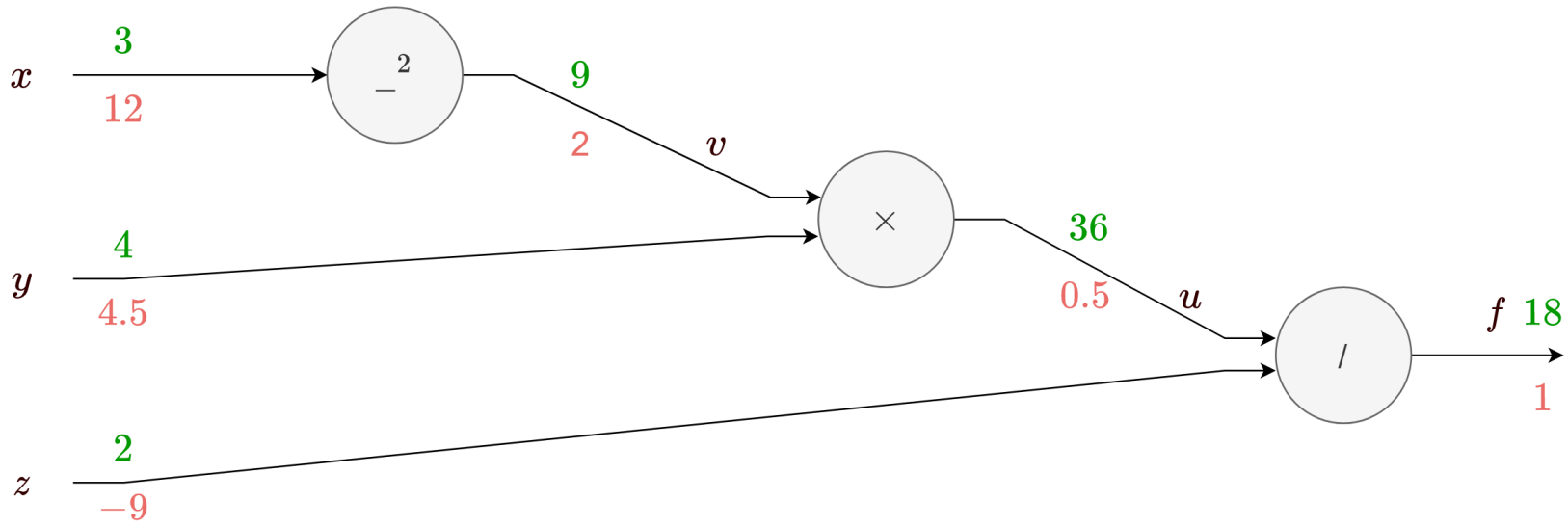
- Backpropagation for $\_^2$ module:



$$\frac{\partial f}{\partial x} = \frac{\partial v}{\partial x} \times \frac{\partial f}{\partial v} \qquad \begin{cases} \frac{\partial f}{\partial v} = 2 \\ \frac{\partial v}{\partial x} = 2x = 6 \end{cases}$$

Backward Propagation: Example

- Backpropagation for $\_^2$ module:



- Results are the same as analytical results.

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.

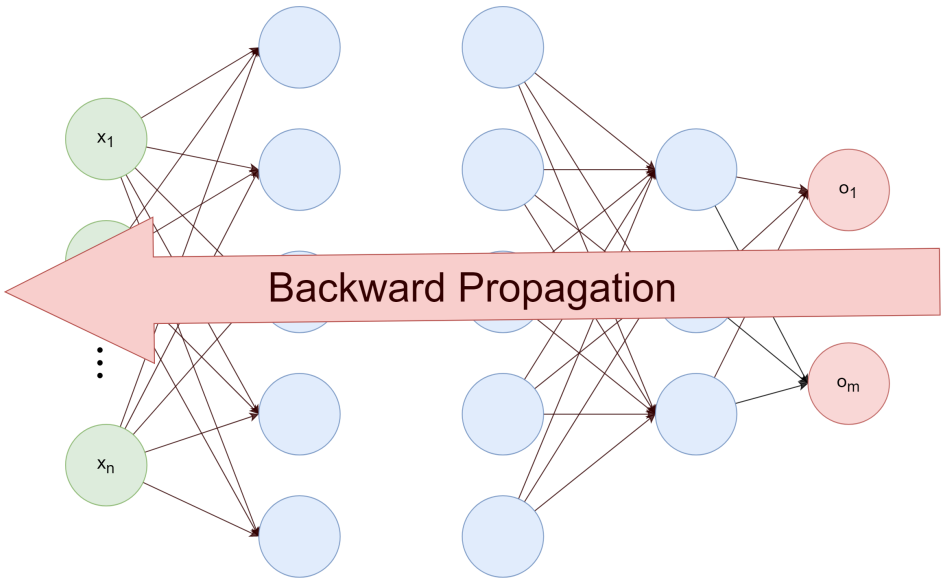


Figure: Backward pass

Various GD types

- 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

Recap:

Training cost function ($\mathcal{J}$) over a dataset usually is the average of loss function ($\mathcal{L}$) on entire training set, so for a dataset $\mathcal{D} = \{d_i\}_{i=1}^{n}$ we have:

$$\mathcal{J}(\mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(d_i;\boldsymbol{\theta})$$

For example:

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

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

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

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

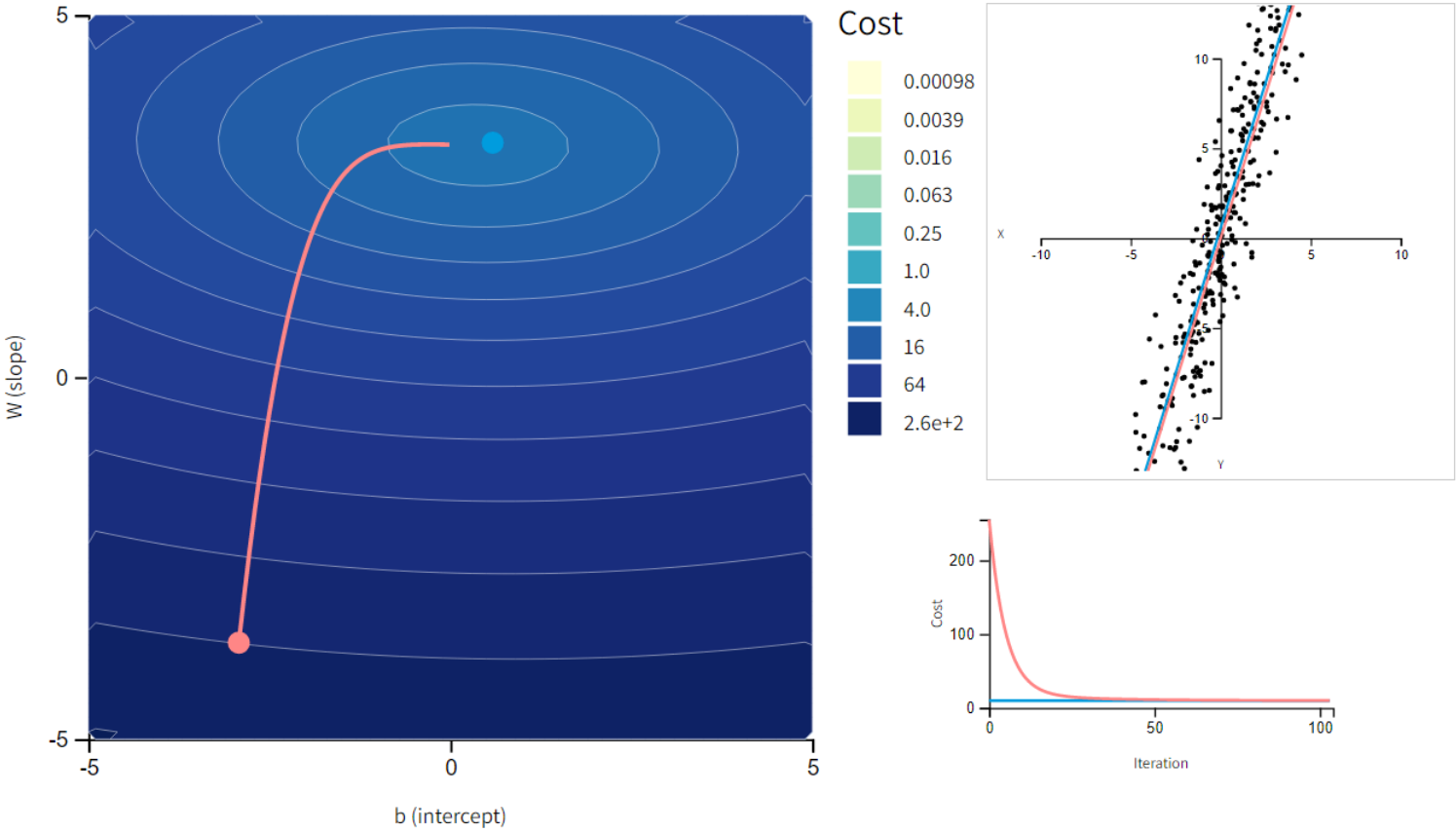


Figure: Optimization of parameters using BGD. Movement is very smooth [3].

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

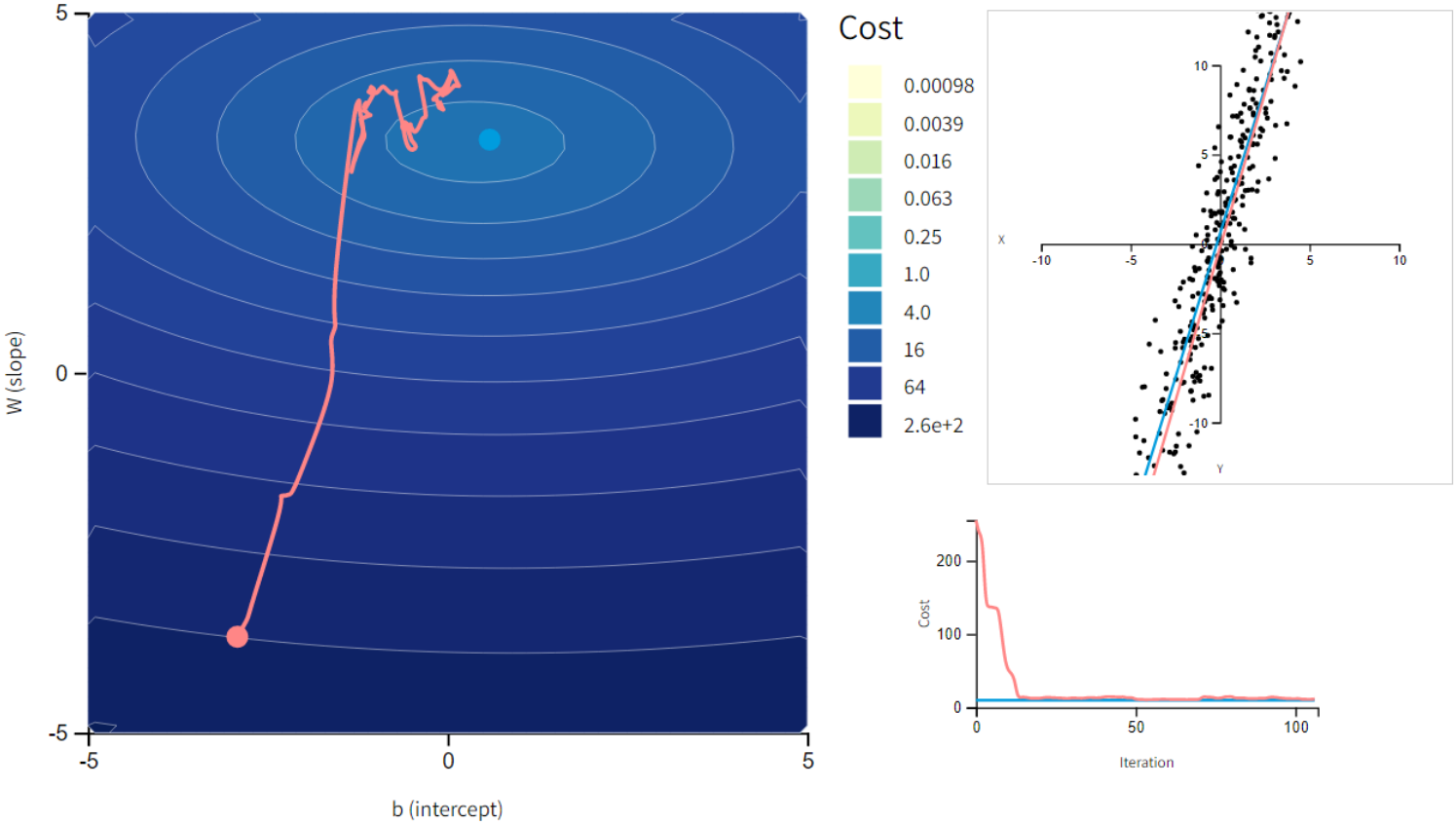


Figure: Optimization of parameters using SGD. As we expect, the movement is not that smooth [3].

Various GD types: Mini-Batch Gradient Descent

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

Mini-Batch Gradient:

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

- 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: Mini-Batch Gradient Descent

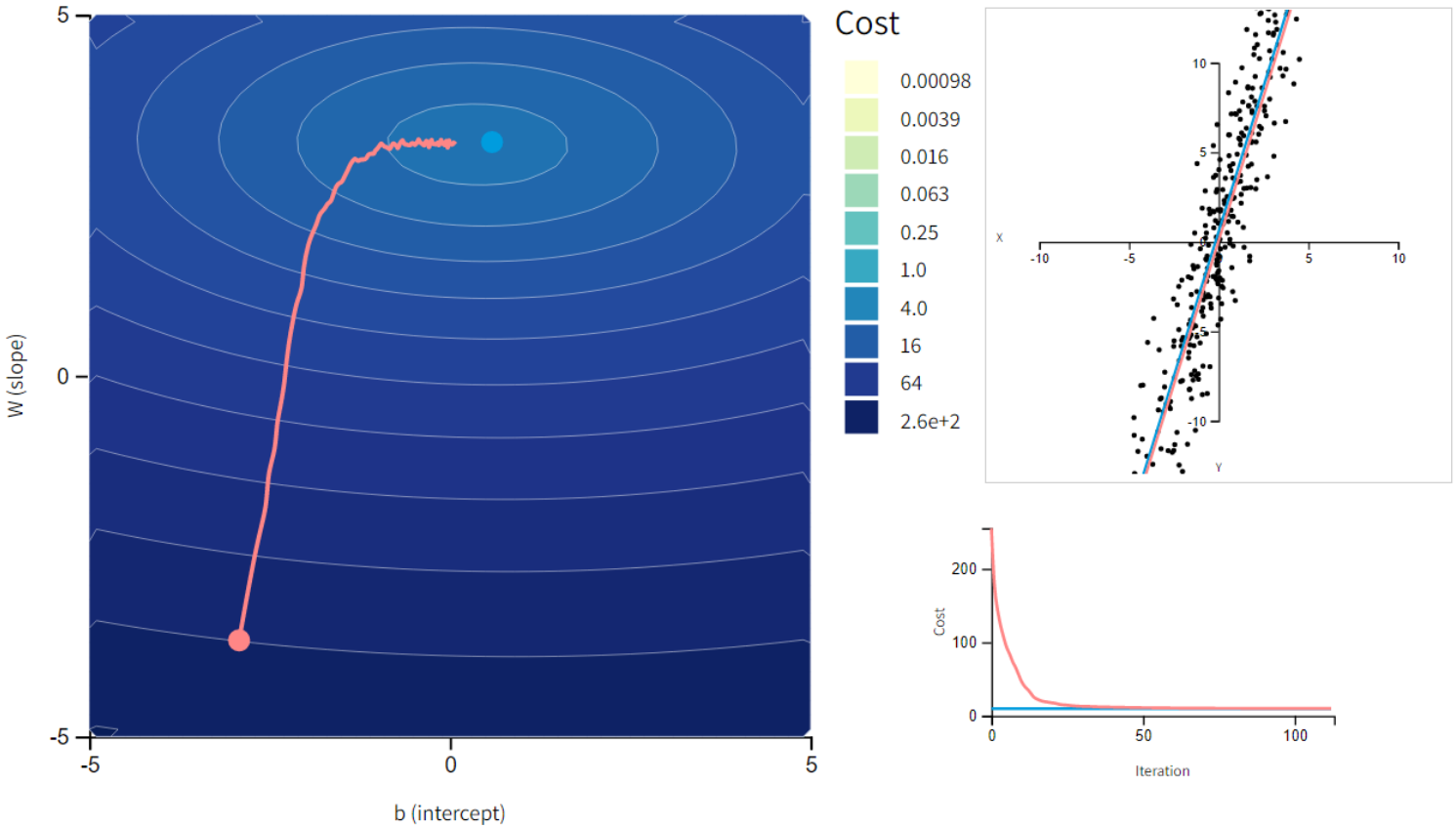


Figure: Optimization of parameters using MBGD. The movement is much smother than SGD and behave like BGD [3].

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.

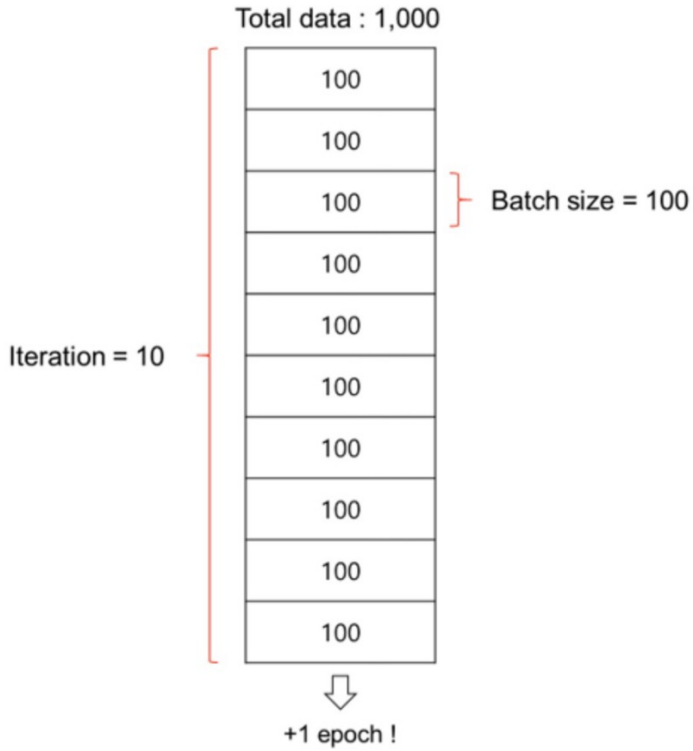


Figure: Epoch vs Iteration, source.

Various GD types

- So we got familiar with different types of GD.
  - ✓ Batch Gradient Descent (BGD)
  - ✓ Stochastic Gradient Descent (SGD)
  - ✓ Mini-Batch Gradient Descent (MBGD)

- The most recommended one is MBGD, because it is computational efficient.
- Choosing the right batch size is important to ensure convergence of the cost function and parameter values, and to the generalization of your model.

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.

Vanishing/Exploding Gradient

- The backpropagation algorithm propagates the error gradient while proceeding from the output layer to the input layer. The issue here is the magnitude of the cost function gradients through the layers...

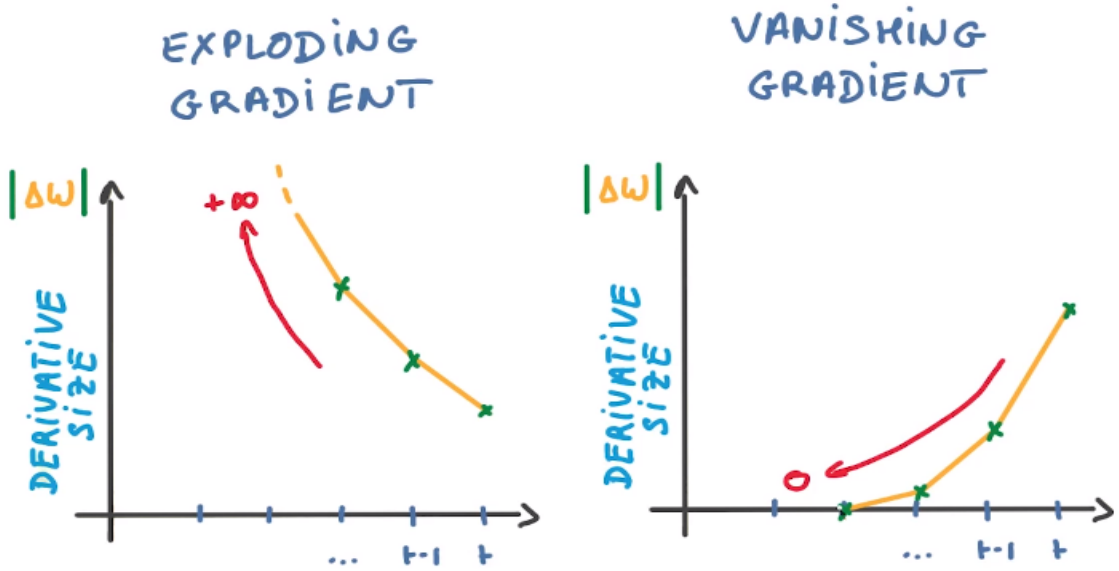


Figure: Vanishing/Exploding Gradient, Source

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.

Weight Initialization

- Is initialization really necessary?
- What are the impacts of initialization?
- A bad initialization may increase convergence time or even make optimization diverge.
- How to initialize?
  - Zero initialization
  - Random initialization

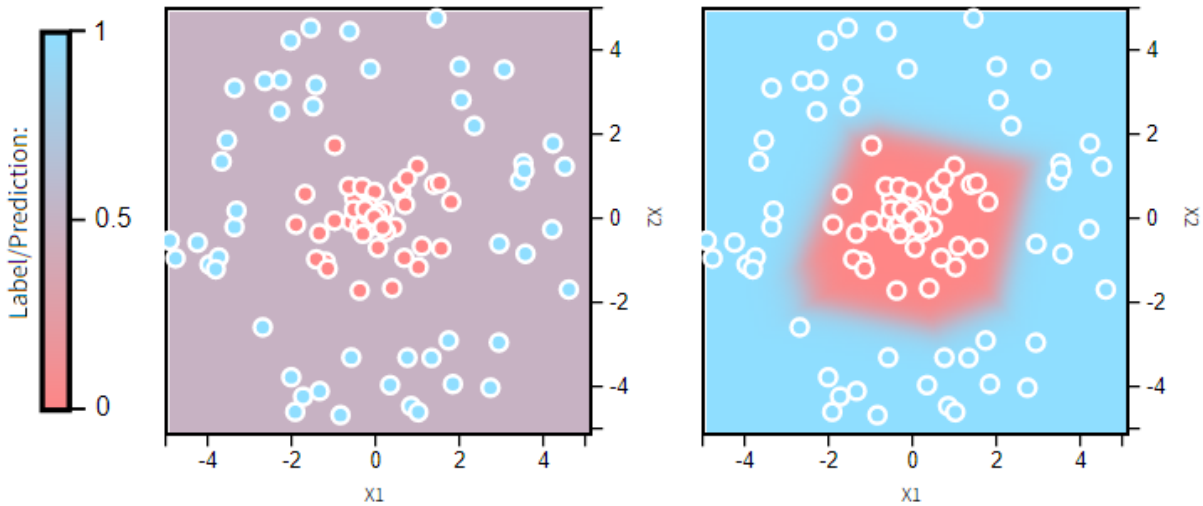


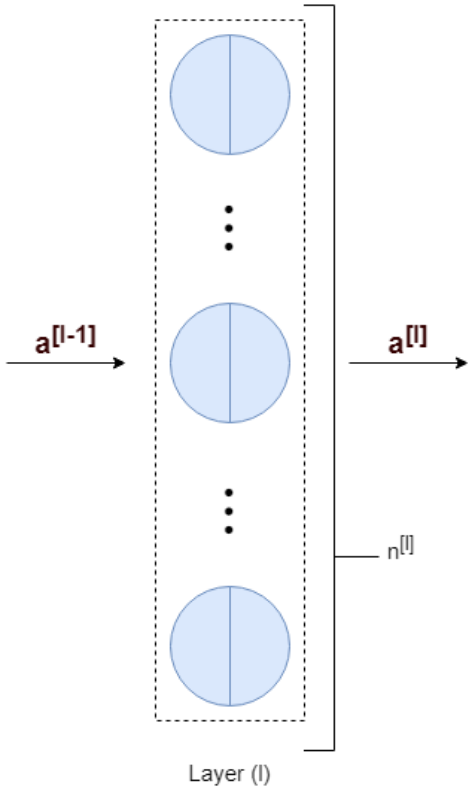
Figure: The output of a three layer network after about 600 epoch. (left) using a bad initialization method and (right) using an appropriate initialization [4].

Weight Initialization

Let's review some notations before we continue:

$$\begin{cases} n^{[l]} := \text{layer } l \text{ neurons number}, \\ W^{[l]} := \text{layer } l \text{ weights}, \\ b^{[l]} := \text{layer } l \text{ biases}, \\ a^{[l]} := \text{layer } l \text{ outputs} \end{cases}$$

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



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

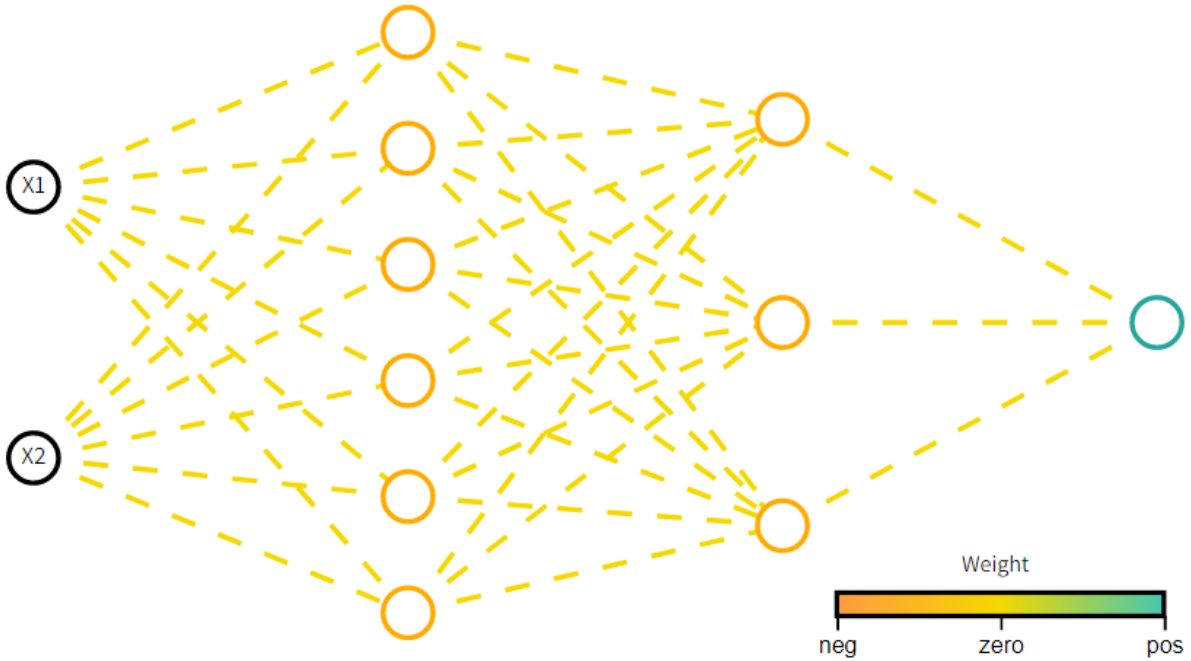


Figure: As we can see network has failed to break symmetry. There has been no improvement in weights after about 600 epochs of training [4].

- 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

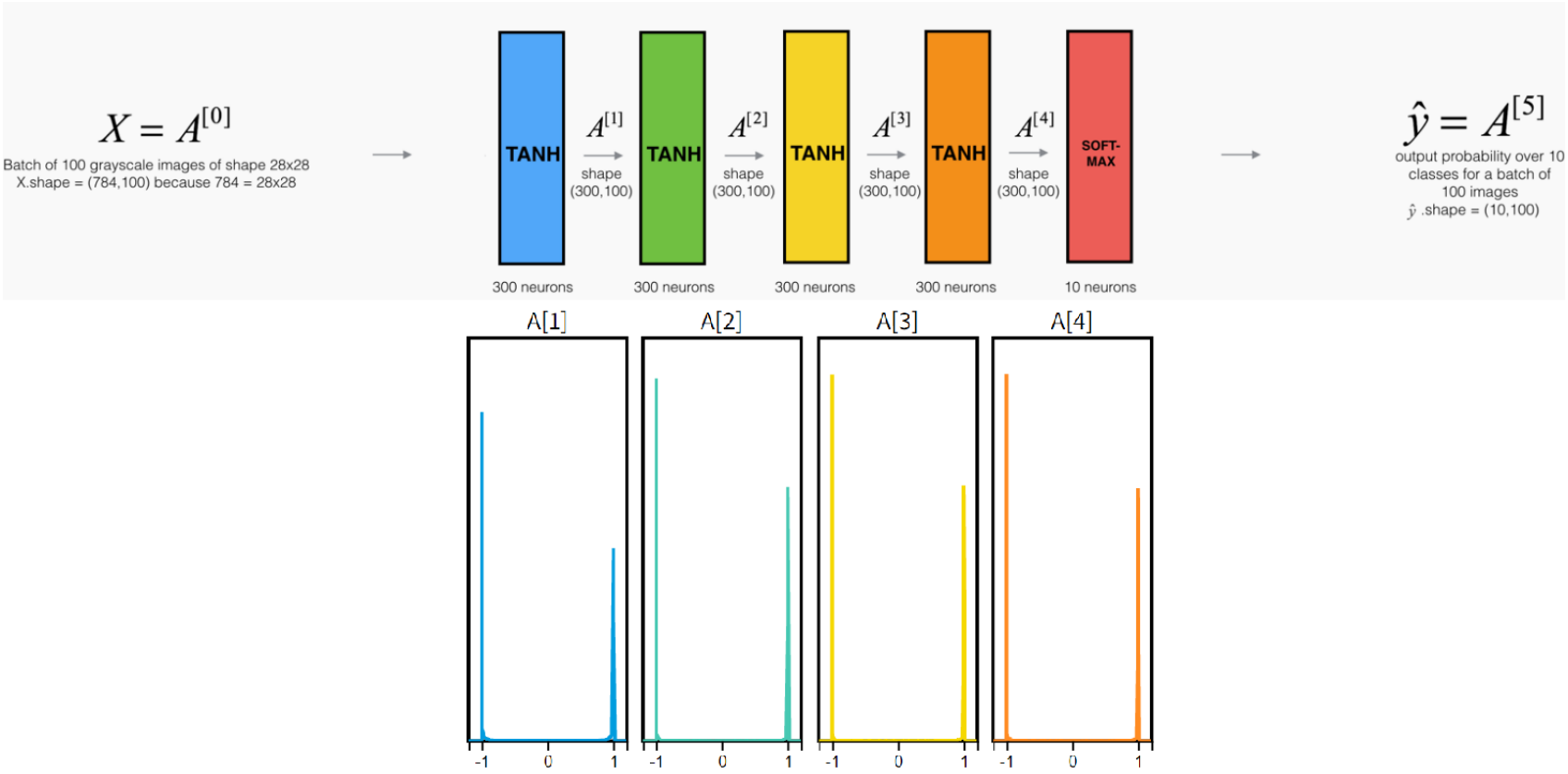


Figure: Uniform initialization problem. On the top, you can see the model architecture, and on the bottom, you can see the density of each layer's output. Model has trained on MNIST dataset for 4 epoch. Weights are initialized randomly from $U(\frac{-1}{\sqrt{n^{[l-1]}}}, \frac{1}{\sqrt{n^{[l-1]}}})$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

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:

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

- For Normal distribution use:

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

(You can read about why this method works at [4].)

Weight Initialization: Xavier Initialization

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

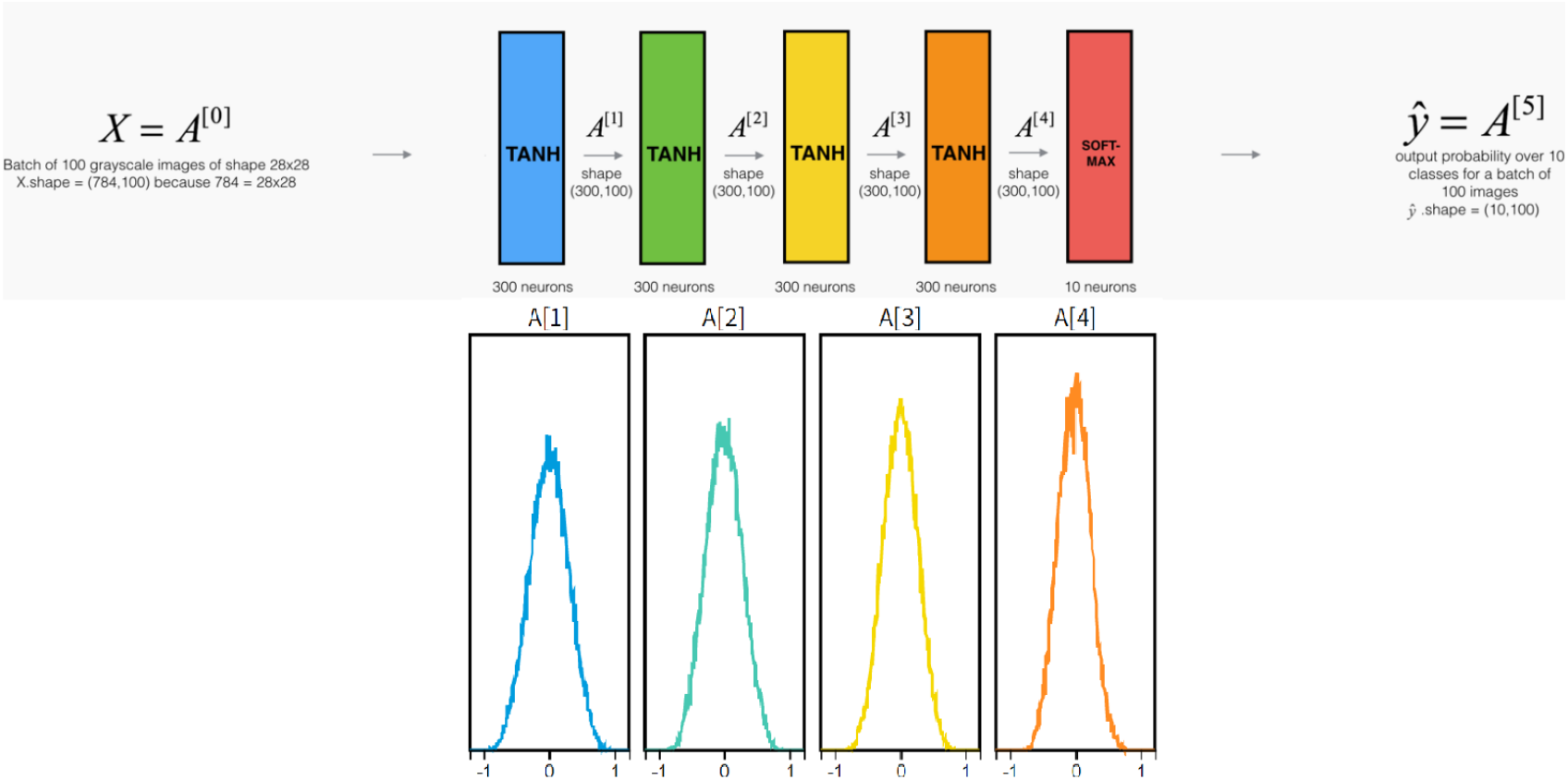


Figure: Vanishing gradient is no longer problem using Xavier initialization. Model has trained on MNIST dataset for 4 epoch. [4].

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.

Saturating Functions

- A non-saturating activation function squeezes the input.

$$(\lim_{x \to -\infty} |f(x)| = +\infty) \vee (\lim_{x \to +\infty} |f(x)| = +\infty)$$

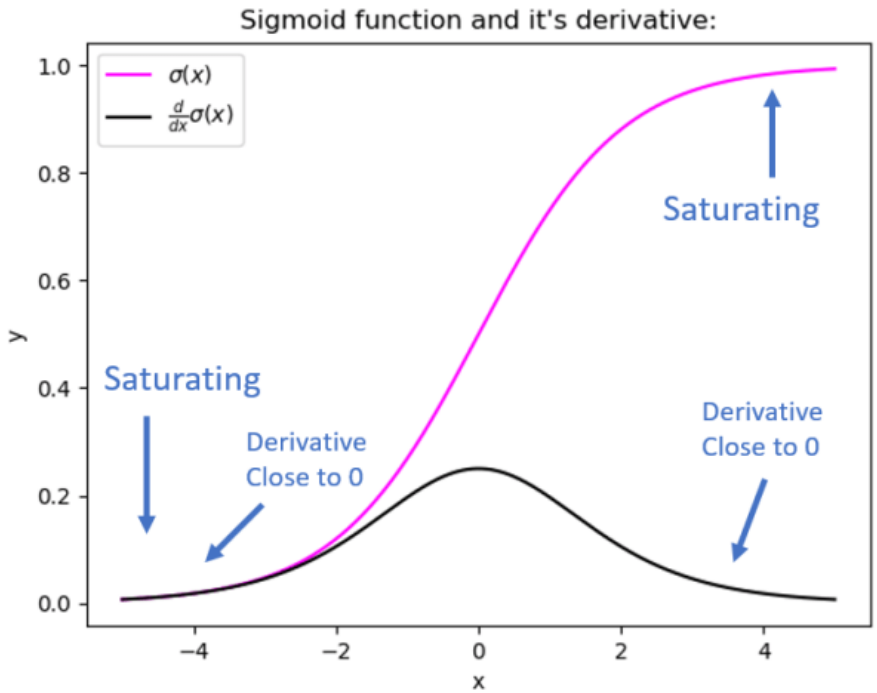


Figure: Saturating Functions, Source

Saturating Functions

- The Tanh (hyperbolic tangent) activation function is saturating as it squashes real numbers to range between $(-1, 1)$

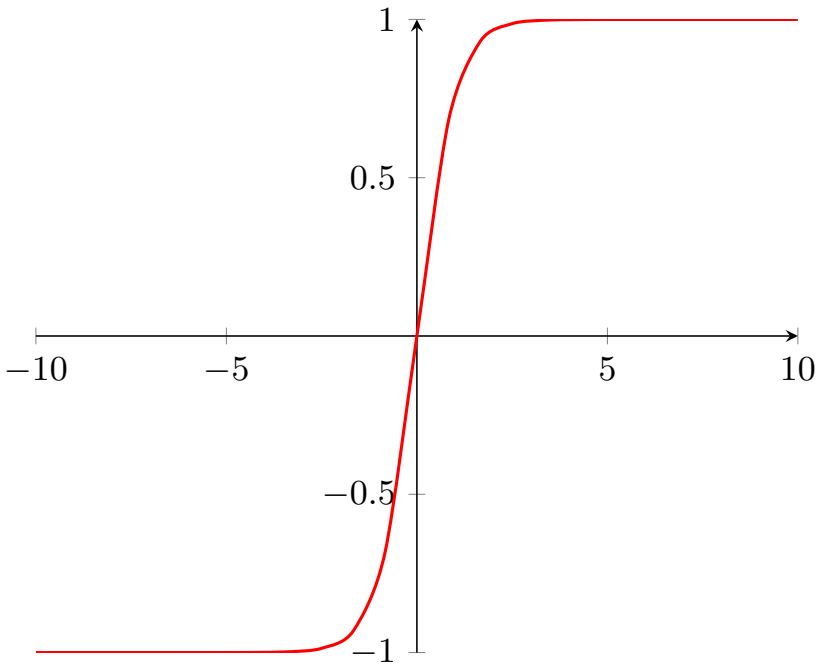


Figure: Hyperbolic Tangent

Saturating Functions

- The Sigmoid activation function $f(x) = \frac{1}{1+\exp^{-x}}$ is also saturating, because it squashes real numbers to range between $(0, 1)$.

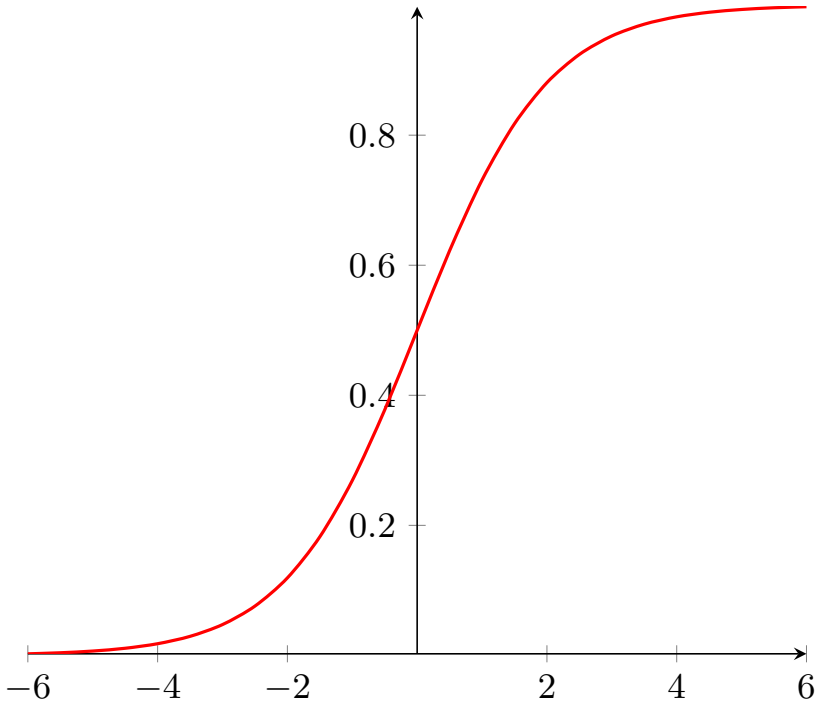


Figure: Sigmoid

Saturating Functions

- In contrast, The Rectified Linear Unit (ReLU) activation function is non-saturating.

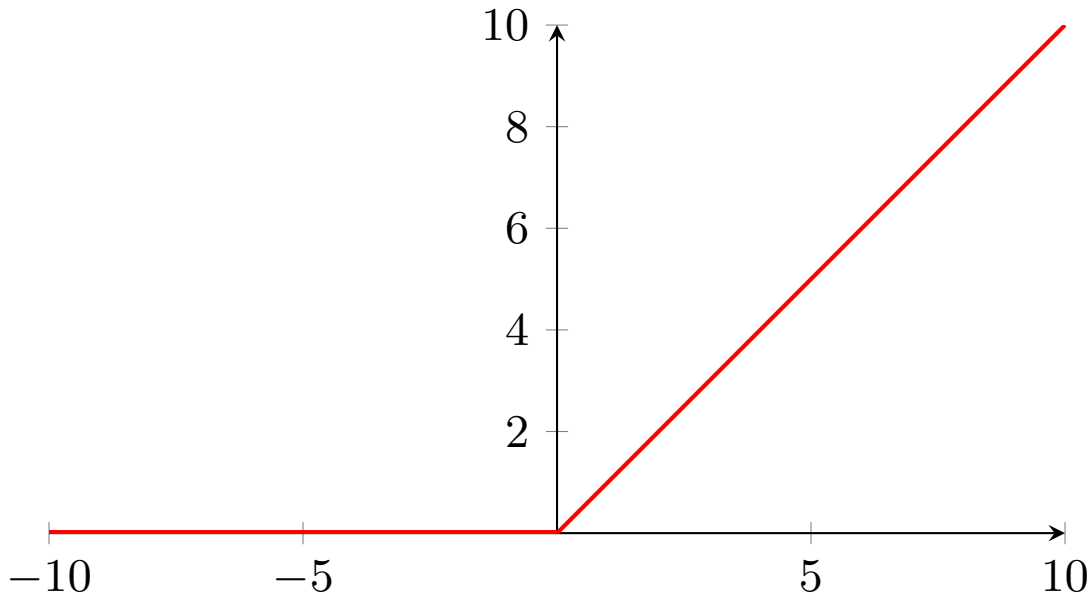


Figure: ReLU

Solution: Batch Norm Layer

- It is used to normalize the data.

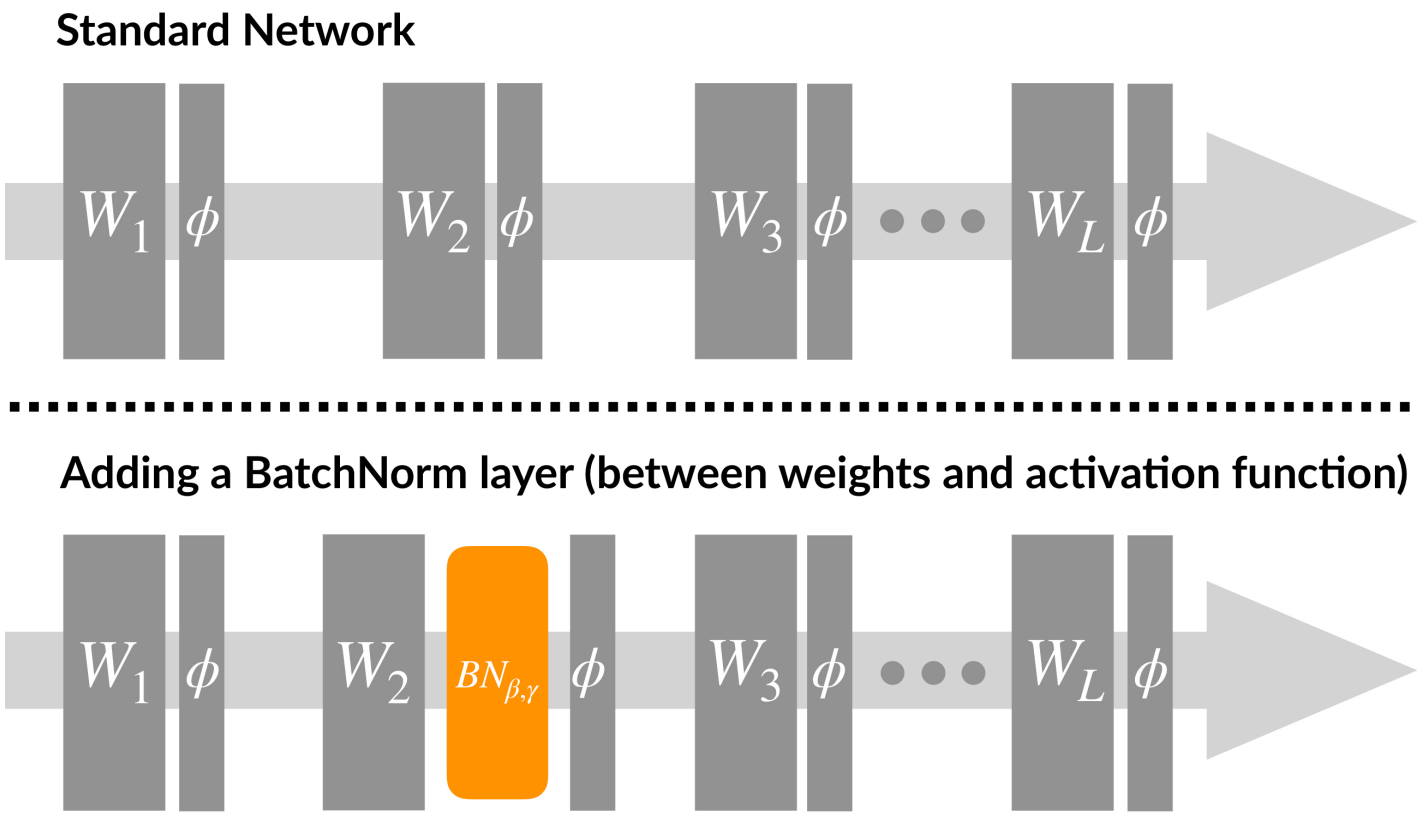


Figure: The suggested place to put a BatchNorm layer, Source

Batch Norm: Training Time

- First, it zero-centers and normalizes the batch.

$$\mu_B := \frac{1}{N_B} \sum x_B^{(i)}$$

$$\sigma_B^2 := \frac{1}{N_B} \sum (x_B^{(i)} - \mu_B)^2$$

$$\hat{x_B}^{(i)} = \frac{x_B^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

- Then, scales and shifts the batch with two learnable parameters γ, β.

$$y_B^{(i)} = \gamma \hat{x_B}^{(i)} + \beta$$

Batch Norm: Test Time

- To zero-center and normalize the input, we need the average and variance of the whole data.
- Those parameters can be acquired during the training.
- Therefore, we need two more trainable parameters.

$$\mu_D := \frac{1}{N} \sum x^{(i)}$$

$$\sigma_D^2 := \frac{1}{N} \sum (x^{(i)} - \mu_D)^2$$

Batch Norm: Test Time

- The majority of Batch Normalization implementations use an exponential moving average of the layer's input means and standard deviations to estimate these final statistics during training.

$$\begin{aligned}\mu_D &= \alpha\mu_D + (1-\alpha)\mu_B \\ &= \mu_D - (1-\alpha)\mu_D + (1-\alpha)\mu_B \\ &= \mu_D - (1-\alpha)(\mu_D - \mu_B)\end{aligned}$$

- α is the momentum hyperparameter.
- Based on the equations, older values are lost earlier when momentum is less.
- As a result, the moving average changes more quickly.

Batch Norm: Performance

- Normalizing the data improves the convergence speed by a considerable amount.

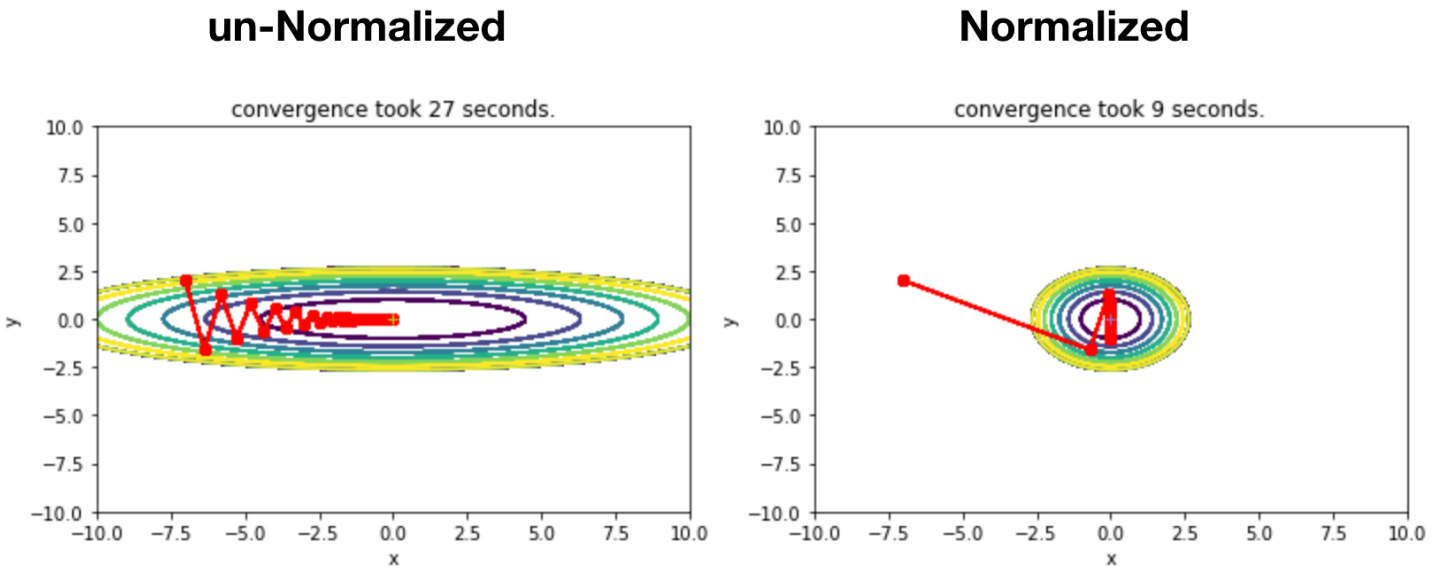


Figure: BatchNorm performance. Convergence speed is increased by 200%, Source

Batch Norm

- Pros
  - Vanishing/Exploding gradient problem is reduced by a considerable amount.
  - You can use even saturating activation functions.
  - The network is much less sensitive to the initial weight.
  - We're able to use larger learning rates, which speeds up the training.
  - It also acts as a regularizer.
    - There is no need for other regularizer techniques.

Batch Norm

- Cons
  - It increases model parameters and prediction latency.
- Solution: during the test time, we can mix the BatchNorm layer with its previous layer to hold the prediction latency.

$$x'^{(i)} = Wx^{(i)} + b$$

$$y^{(i)} = \frac{x'^{(i)} - \mu_D}{\sqrt{\sigma_D^2 + \epsilon}} + \beta$$

$$\Rightarrow$$

$$y^{(i)} = W'x^{(i)} + b'$$

$$W' := \frac{1}{\sqrt{\sigma_D^2 + \epsilon}} W$$

$$b' := \beta + \frac{b - \mu_D}{\sqrt{\sigma_D^2 + \epsilon}}$$

Gradient Clipping

- What will happen in case of a large gradient value?
- The gradient descent will take us far away from our local position.

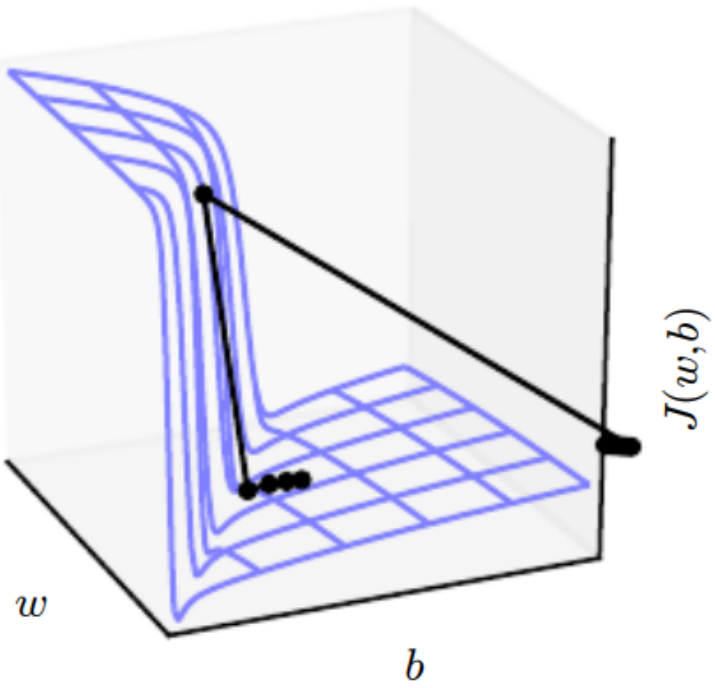


Figure: The problem of large gradient value [5].

Gradient Clipping

- Solve this problem simply by clipping gradient.
- Two approaches to do so:
  - Clipping by value
  - Clipping by norm

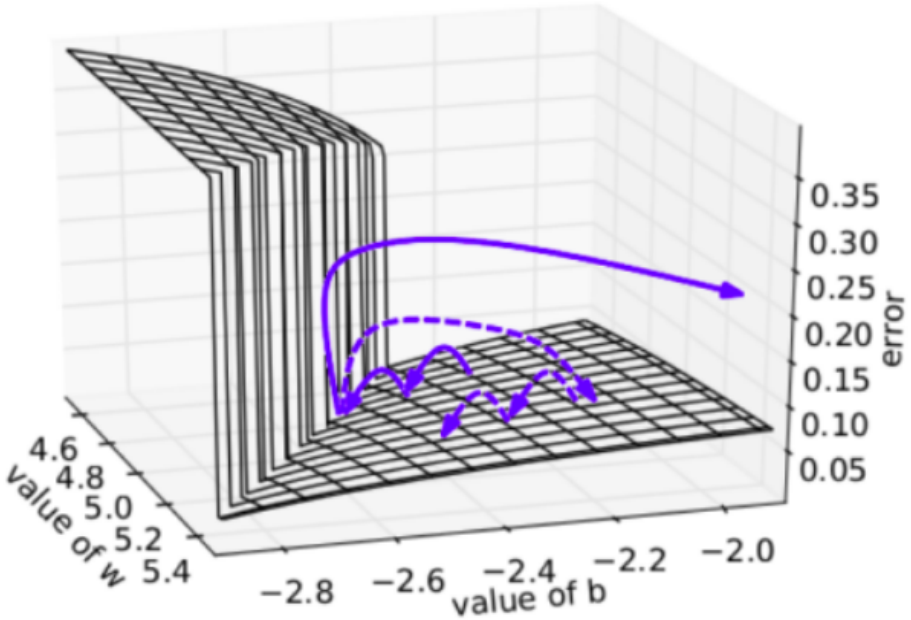
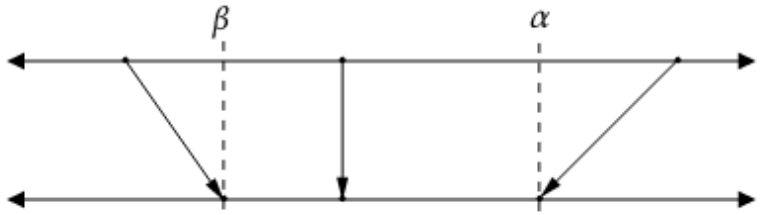


Figure: The effect of gradient clipping. Instead of solid line following dotted line will lead us to minimum, Source

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:

$$\text{if } \boldsymbol{g}_i > \alpha:$$
$$\boldsymbol{g}_i \leftarrow \alpha$$
$$\text{else if } \boldsymbol{g}_i < \beta:$$
$$\boldsymbol{g}_i \leftarrow \beta$$



Gradient Clipping by value

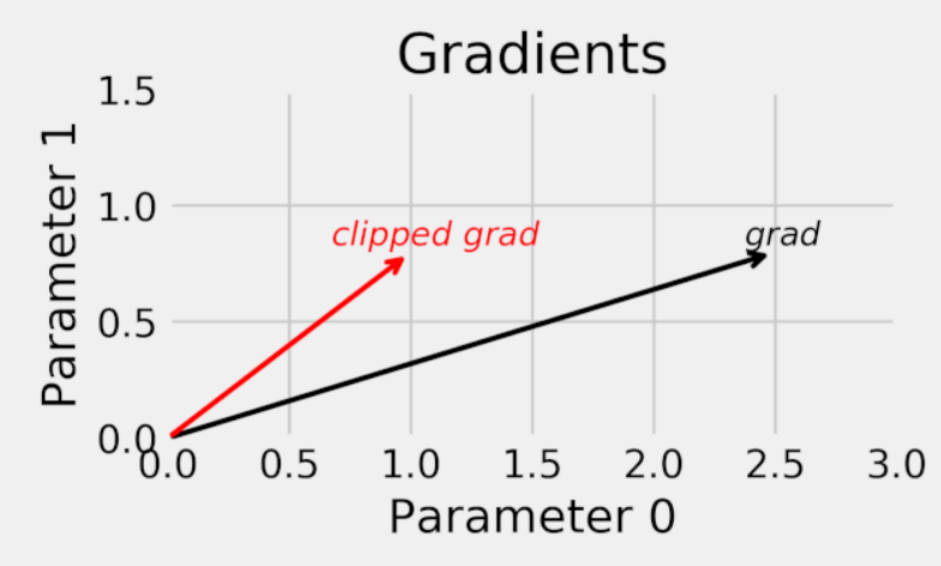


Figure: The effect of clipping by value. Source

- 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:

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

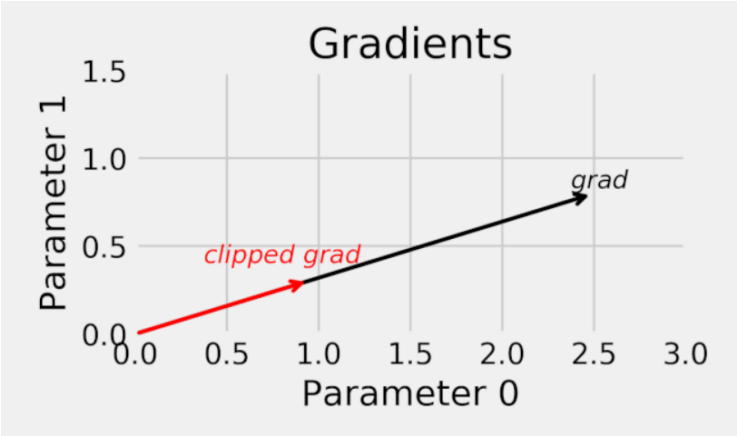


Figure: The effect of clipping by norm. source

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.

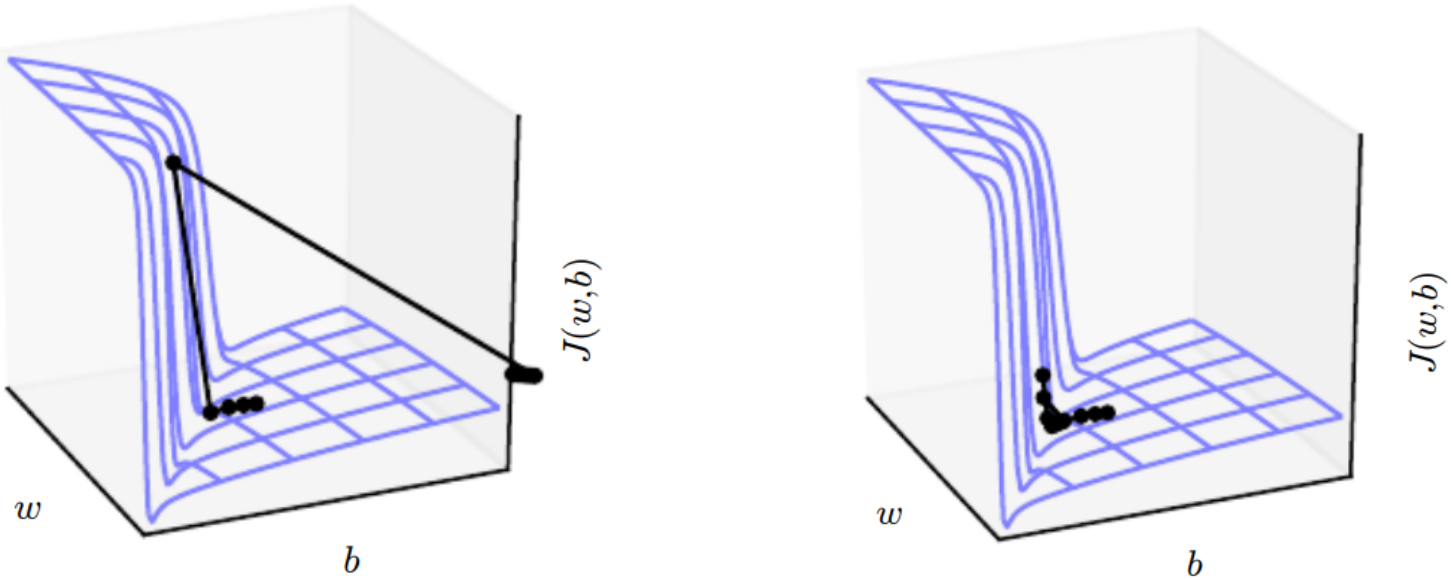


Figure: The "cliffs" landscape (left) without gradient clipping and (right) with gradient clipping [5].

Learning Rate

- Learning rate is a hyper-parameter that controls how much we are adjusting the weights of our network with respect the loss gradient.

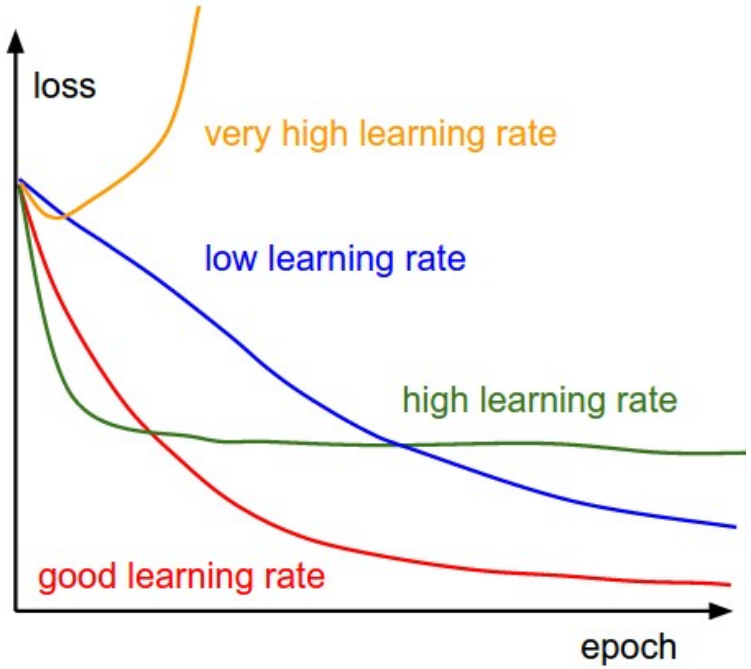


Figure: Learning Rate Effect, Source

Learning Rate

- Low learning rate
  - Not missing Local Minima.
  - But takes too much time to converge!
- High learning rate
  - Fast, But may diverge!

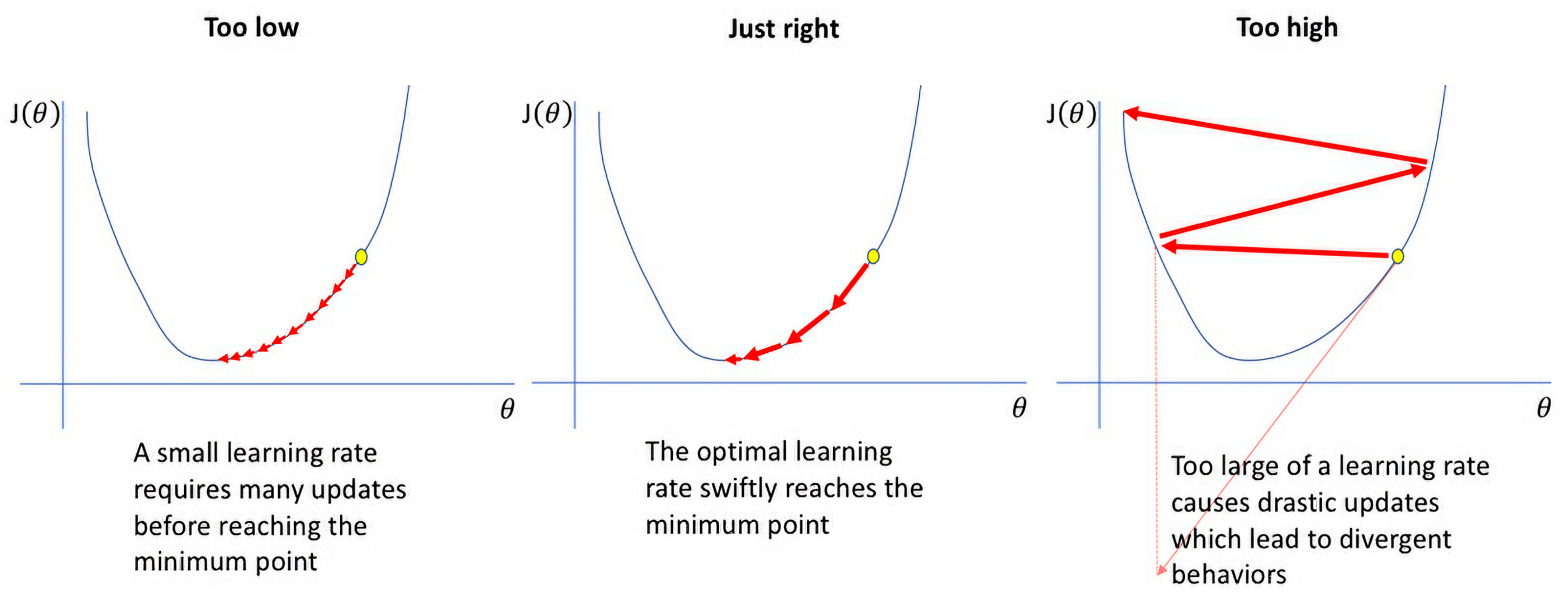


Figure: Learning Rate Effect, Source

Learning Rate

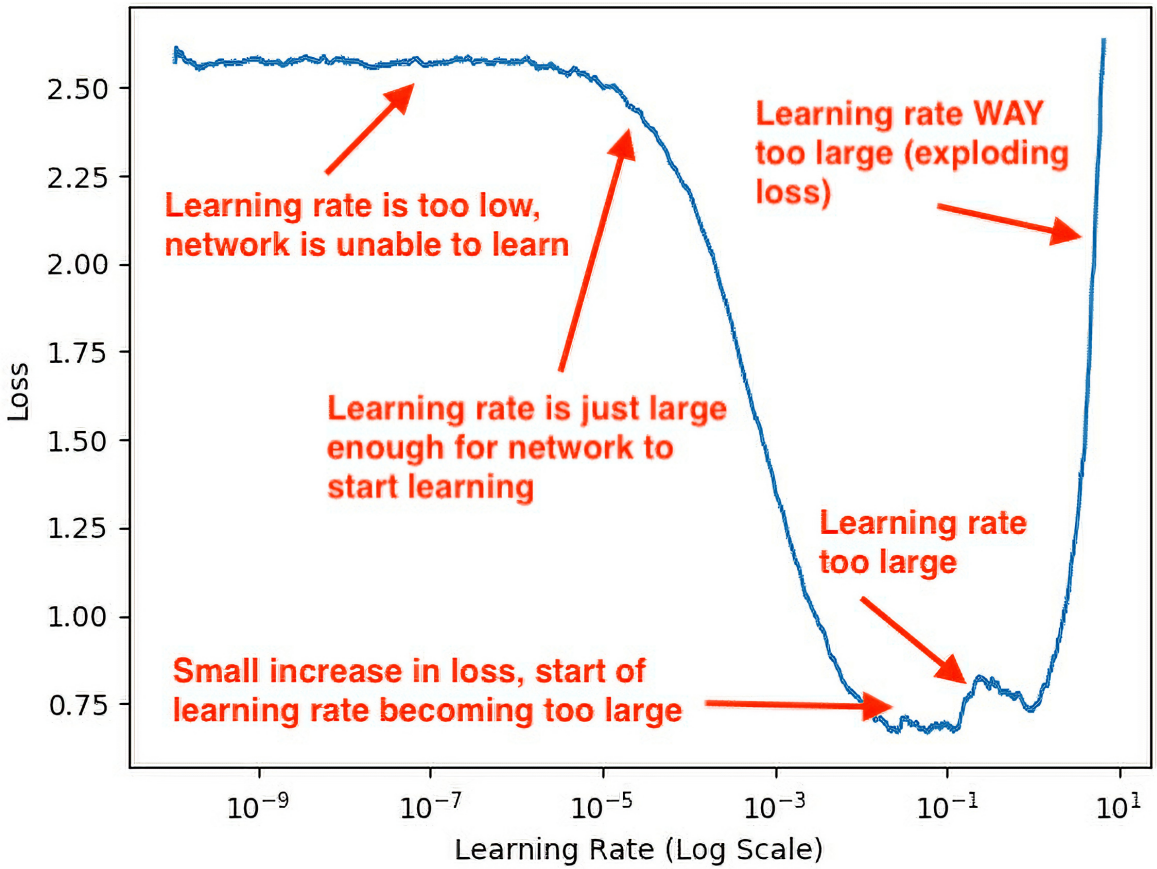


Figure: Learning Rate Effect, Source

Problem: OverFitting in a Neural Network

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

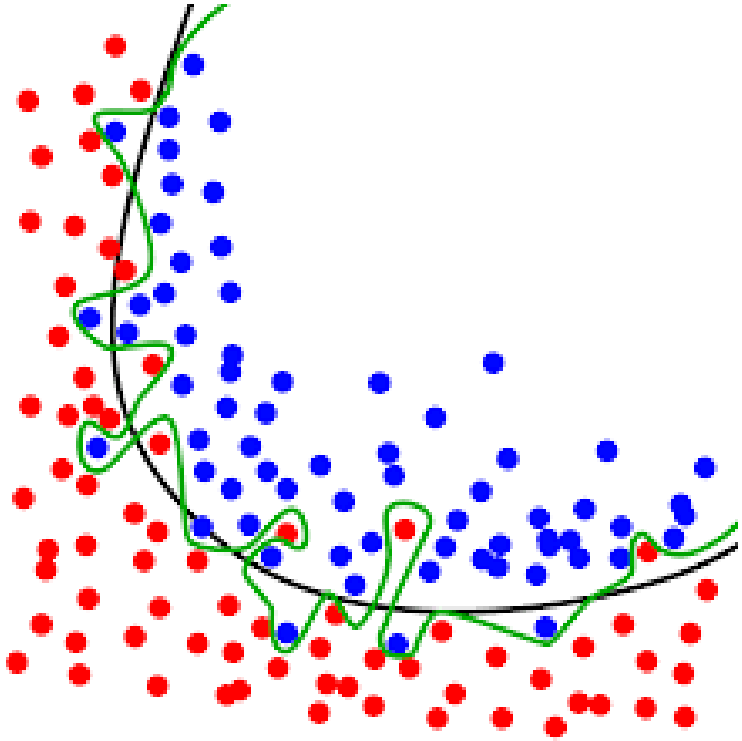


Figure: OverFitting in a neural network, Source

Solution 1: L1/L2 Regularization

- It is like a linear regression regularizer.
- Sum the regularizer term for every layer weight!

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

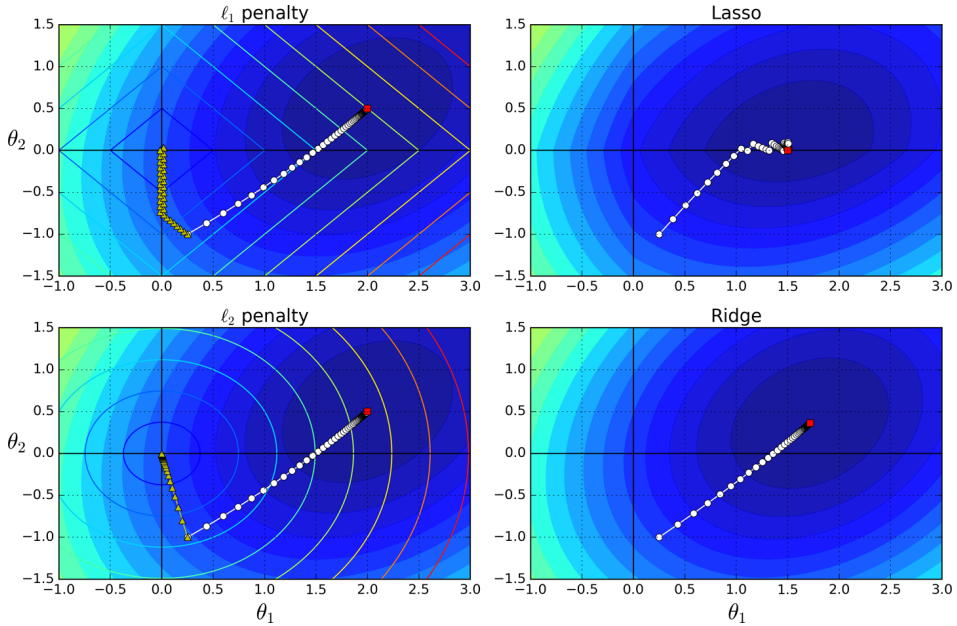


Figure: Convergence diagram for different losses, Source

L1/L2 Regularization

- L1/L2 regularizer functions (review)

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

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

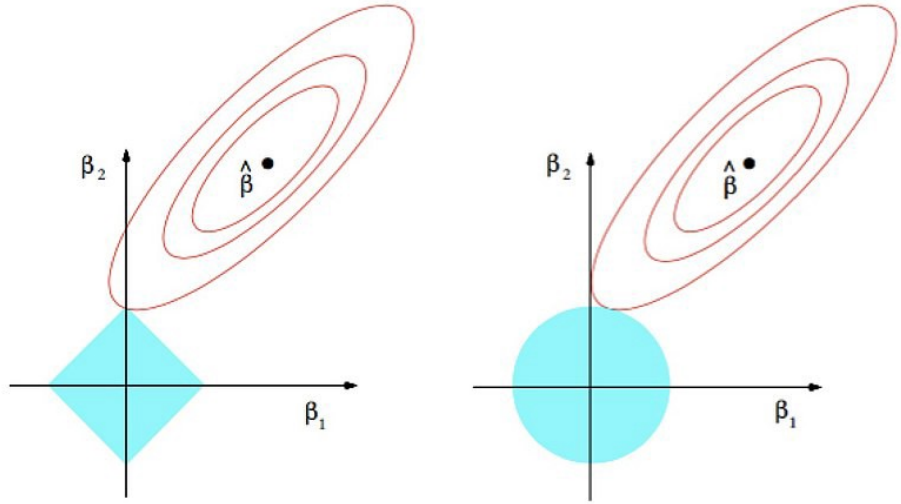


Figure: L1/L2 regularizers' solution diagram, Source

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

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

Solution 2: Early Stopping

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

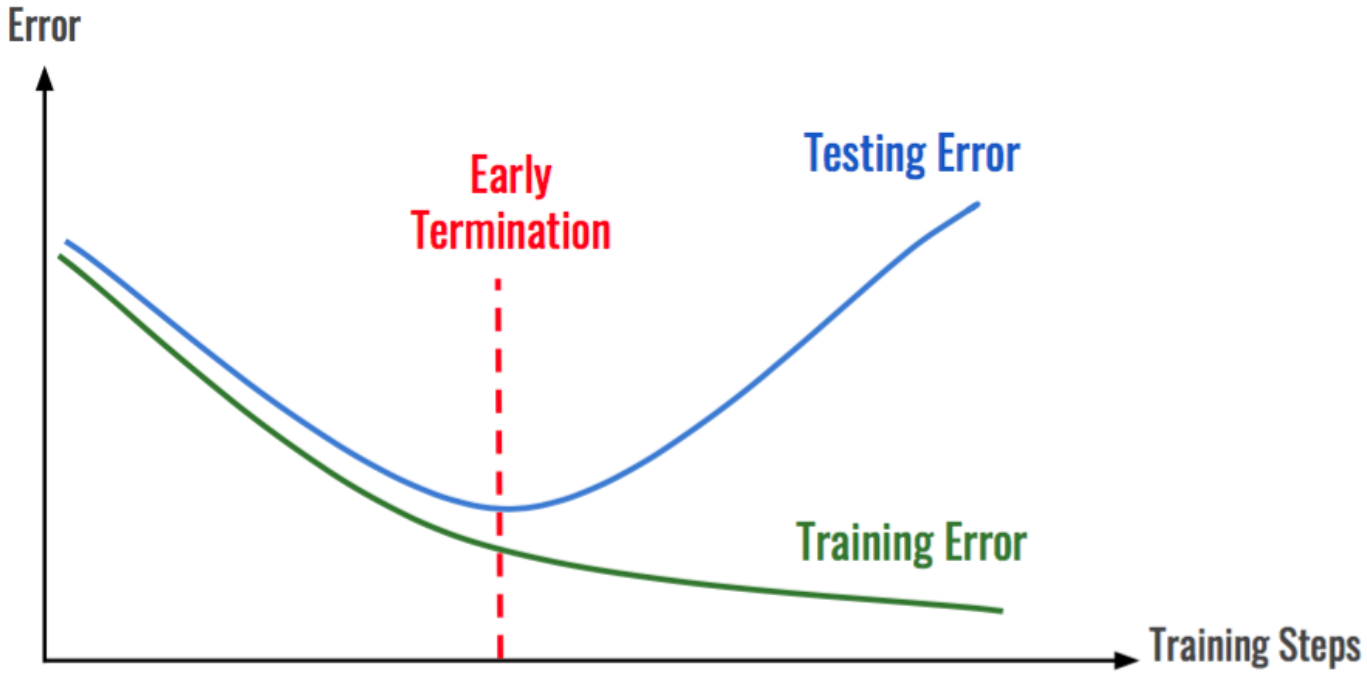


Figure: Early Stopping diagram, Source

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.

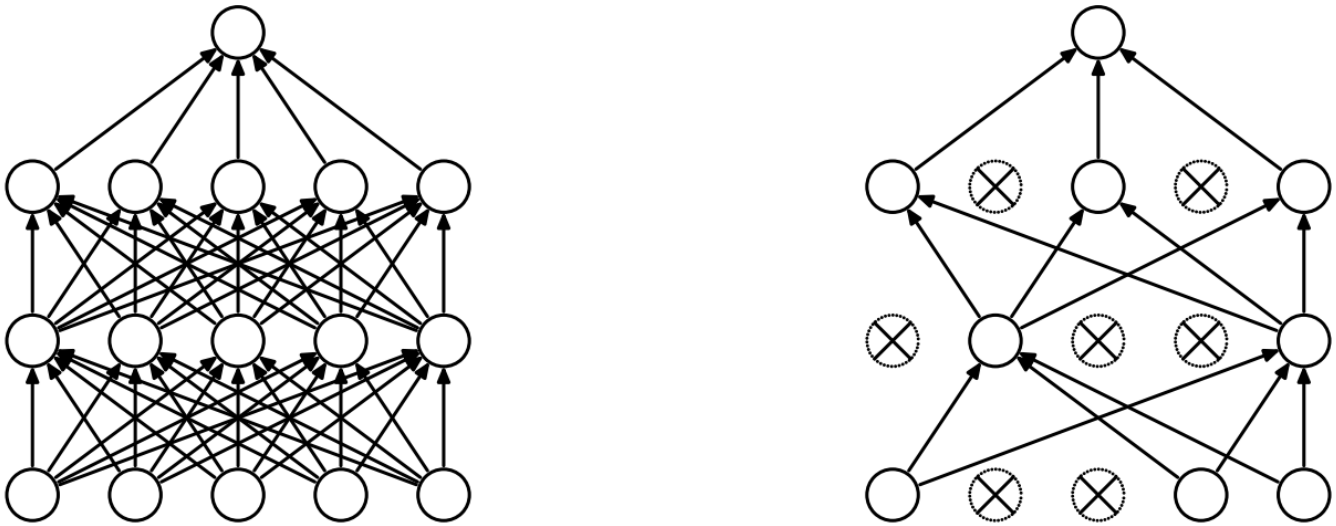


Figure: Behavior of dropout at training time, Source

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.

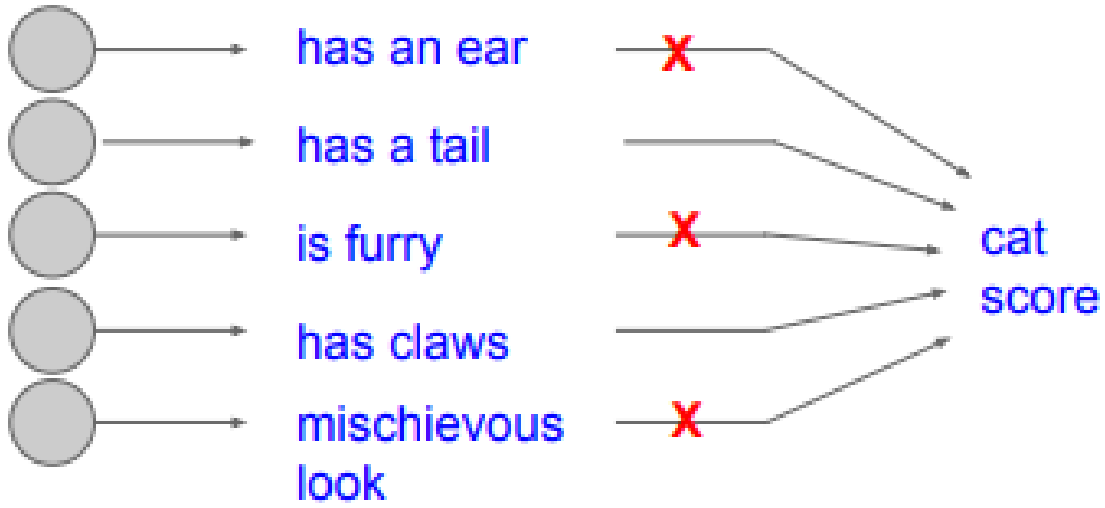


Figure: Discrimination of neurons at training time. [6]

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.
- A fully connected network with 4096 neurons has $2^{4096} \sim 10^{1233}$ possible masks! There are only 10^{82} atoms in the universe!

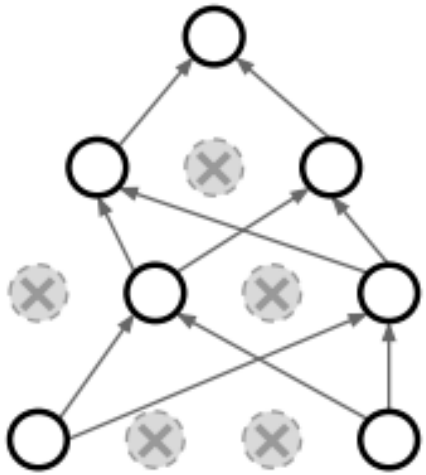


Figure: Behavior of dropout at training time. [6]

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

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

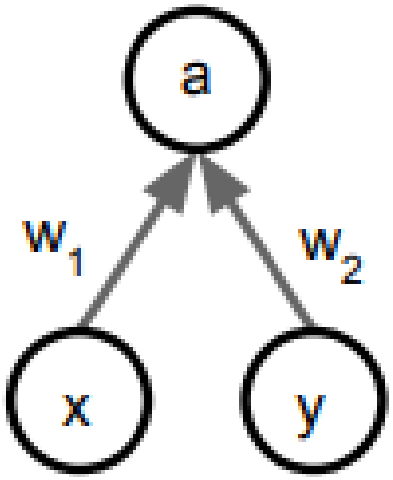


Figure: Simple neural network. [6]

Thank You!

Any Question?

References


M. D. Ergün Akgün, "Biological and neural network neuron," 2018.
https://www.researchgate.net/publication/326417061_Modeling_Course_Achievements_of_Elementary_Education_Teacher_Candidates_with_Artificial_Neural_Networks.


"Gradient descent."
https://subscription.packtpub.com/book/big-data-&-business-intelligence/9781788397872/1/ch01lvl1sec22/gradient-descent.


K. Katanforoosh and D. Kunin, "Parameter optimization in neural networks," 2019.
https://www.deeplearning.ai/ai-notes/optimization/.


K. Katanforoosh and D. Kunin, "Initializing neural networks," 2019.
https://www.deeplearning.ai/ai-notes/initialization/.


I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*.
MIT Press, 2016.
http://www.deeplearningbook.org.


F.-F. L. . J. J. . S. Yeung, "Training neural networks," 2018.
http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture07.pdf.