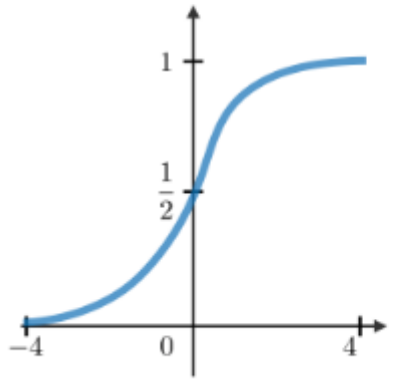
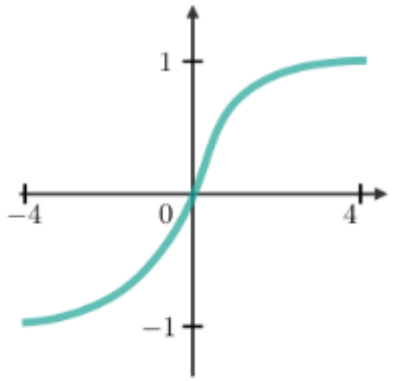
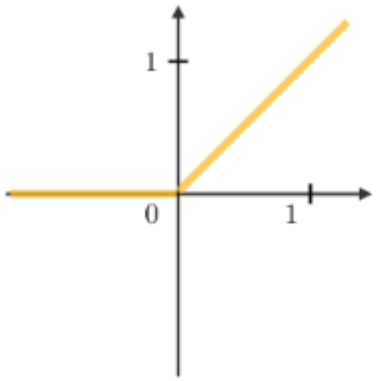
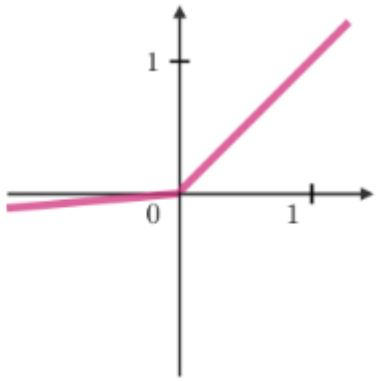
Deep Learning - 2

RRC Summer School 2025

Activation Functions

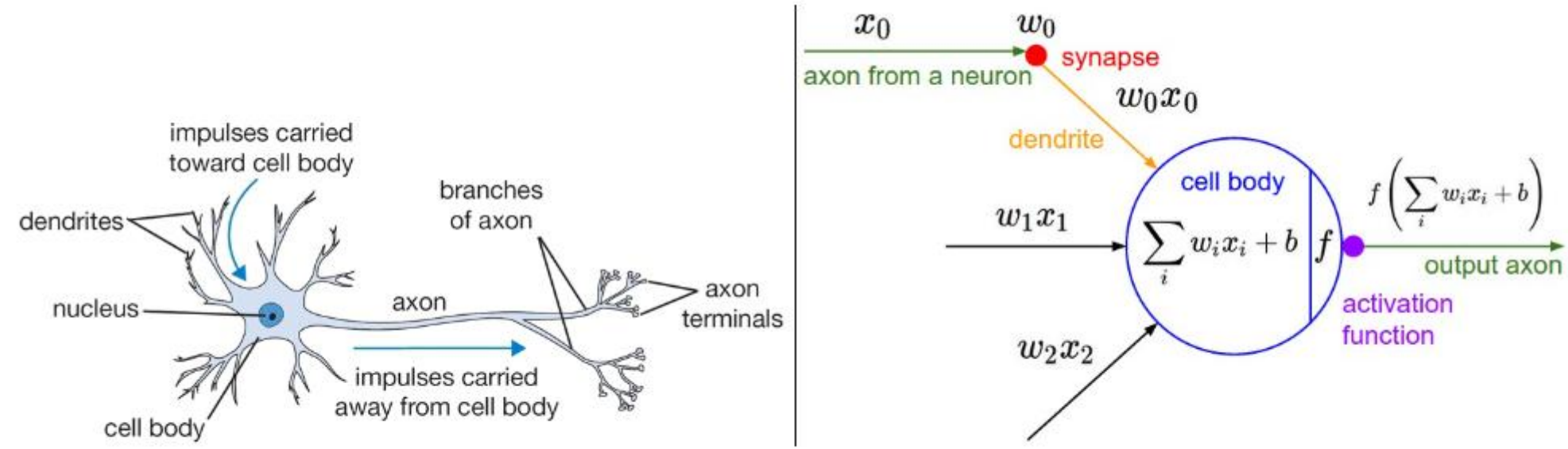
Sigmoid	Tanh	ReLU	Leaky ReLU
$g(z) = \frac{1}{1+e^{-z}}$	$g(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	$g(z) = \max(0, z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$
			

What is an Activation Function?

- An activation function, decides whether a neuron “fires”, or is “activated” or not.
- This means that it will decide whether the neuron’s input to the network is important or not in the process of prediction using simpler mathematical operations.

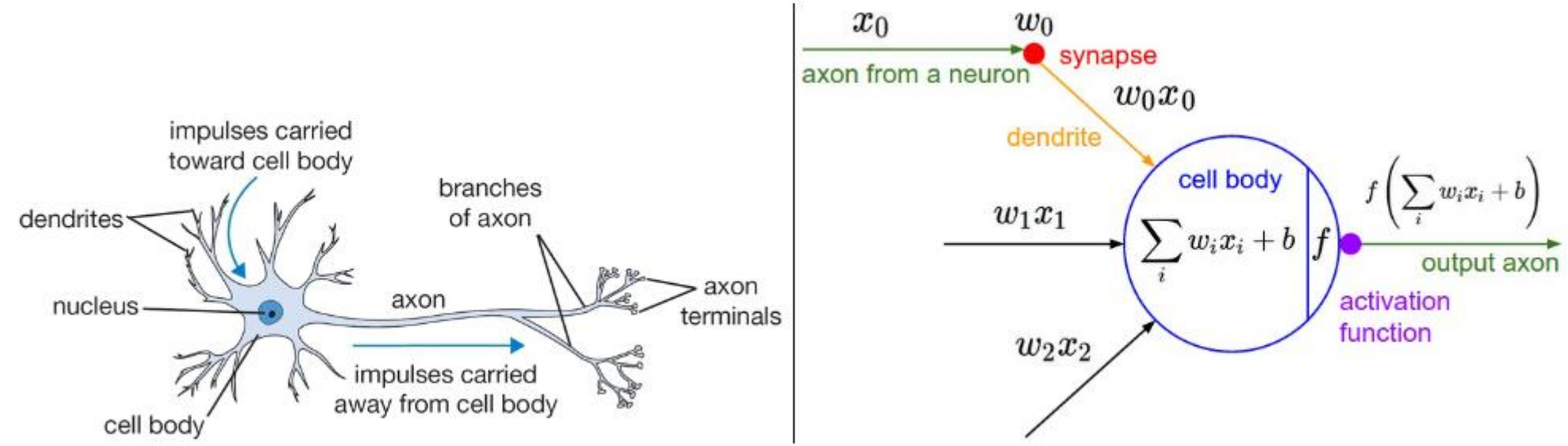
- In practice, it is used to add non-linearities into the neural network.

Biological Motivation of Activation Functions



A cartoon drawing of a biological neuron (left) and its mathematical model (right).

Biological Motivation of Activation Functions



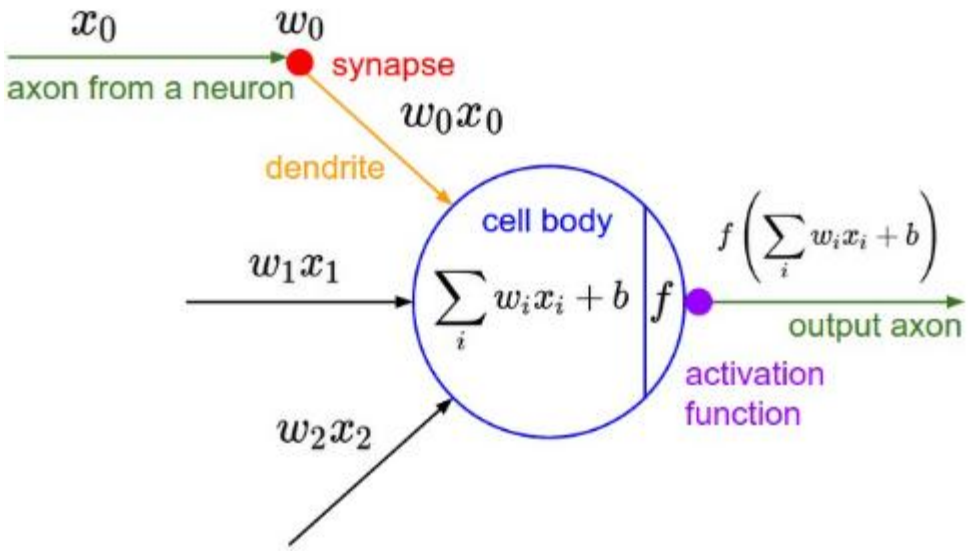
A cartoon drawing of a biological neuron (left) and its mathematical model (right).

The basic computational unit of the brain is a **neuron**. Approximately 86 billion neurons can be found in the human nervous system and they are connected with approximately 10^14 - 10^15 **synapses**. The diagram below shows a cartoon drawing of a biological neuron (left) and a common mathematical model (right). Each neuron receives input signals from its **dendrites** and produces output signals along its (single) **axon**. The axon eventually branches out and connects via synapses to dendrites of other neurons.

Biological Motivation of Activation Functions

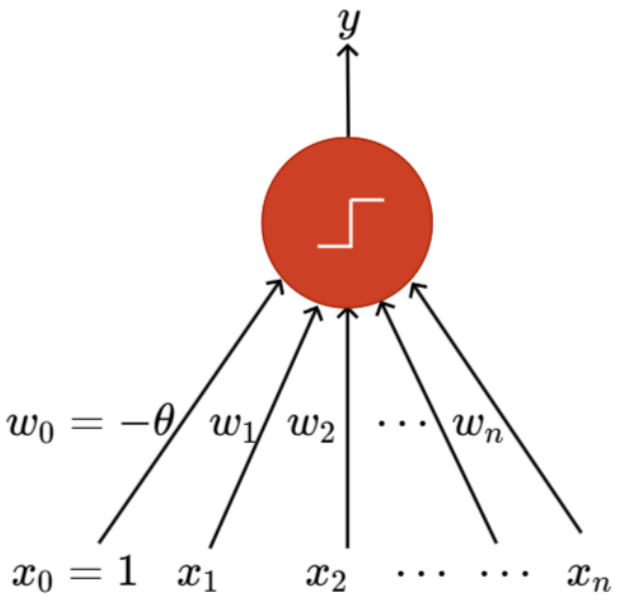
- In the computational model of a neuron, the signals that travel along the axons (e.g. x0) interact multiplicatively (e.g. w0x0) with the dendrites of the other neuron based on the synaptic strength at that synapse (e.g. w0).

- The idea is that the synaptic strengths (the weights w) are learnable and control the strength of influence (and its direction: excitory (positive weight) or inhibitory (negative weight)) of one neuron on another.



Recap

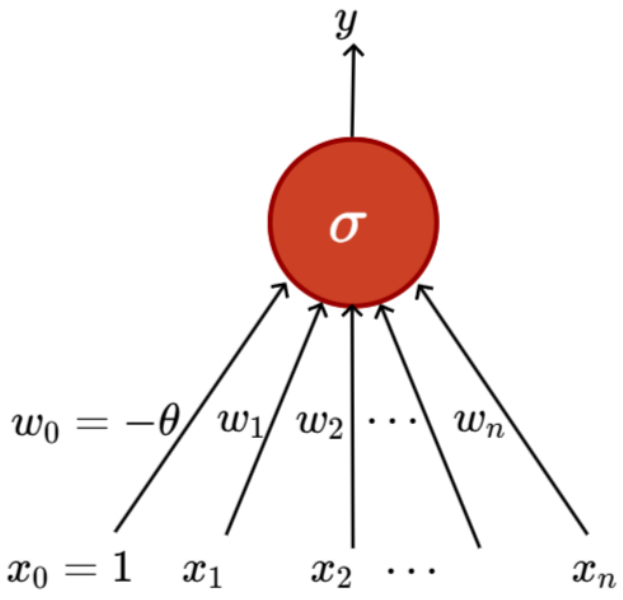
Perceptron



$$y = 1 \text{ if } \sum_{i=0}^{n} w_i x_i \geq 0$$

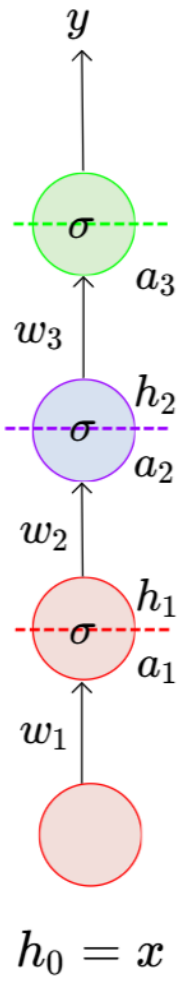
$$= 0 \text{ if } \sum_{i=0}^{n} w_i x_i < 0$$

Sigmoid (Logistic) Neuron



$$y = \frac{1}{1+exp\left(-(w_0+\sum_{i=1}^{n} w_i x_i)\right)}$$

Why are activation Functions Needed



Consider this deep neural network

Imagine if we replace the sigmoid in each layer by a simple linear transformation

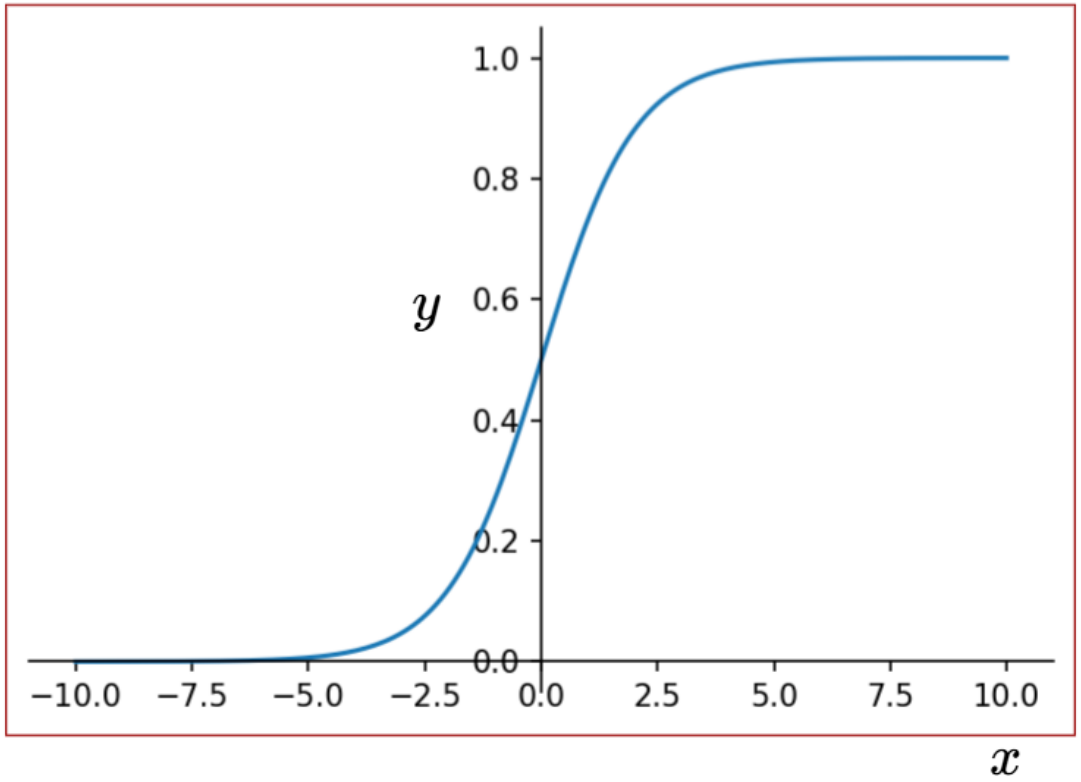
$$y = (w_4 * (w_3 * (w_2 * (w_1 * x))))$$

Then we will just learn y as a linear transformation of x

In other words we will be constrained to learning linear decision boundaries

We cannot learn arbitrary decision boundaries

Types of Activation Functions: Sigmoid



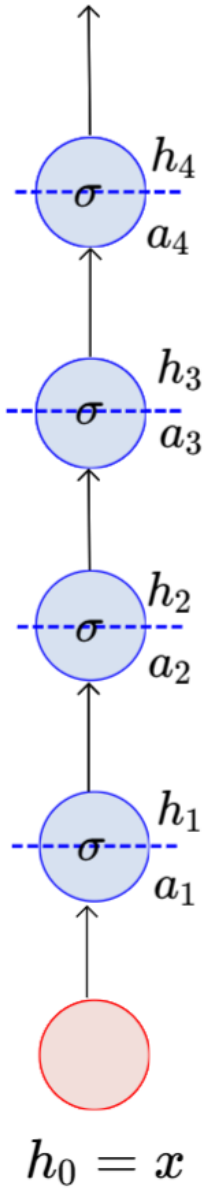
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

As is obvious, the sigmoid function compresses all its inputs to the range [0,1]

Since we are always interested in gradients, let us find the gradient of this function

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$$

Why using Sigmoid is not a good idea



$$a_3 = w_3 h_2$$

$$h_3 = \sigma(a_3)$$

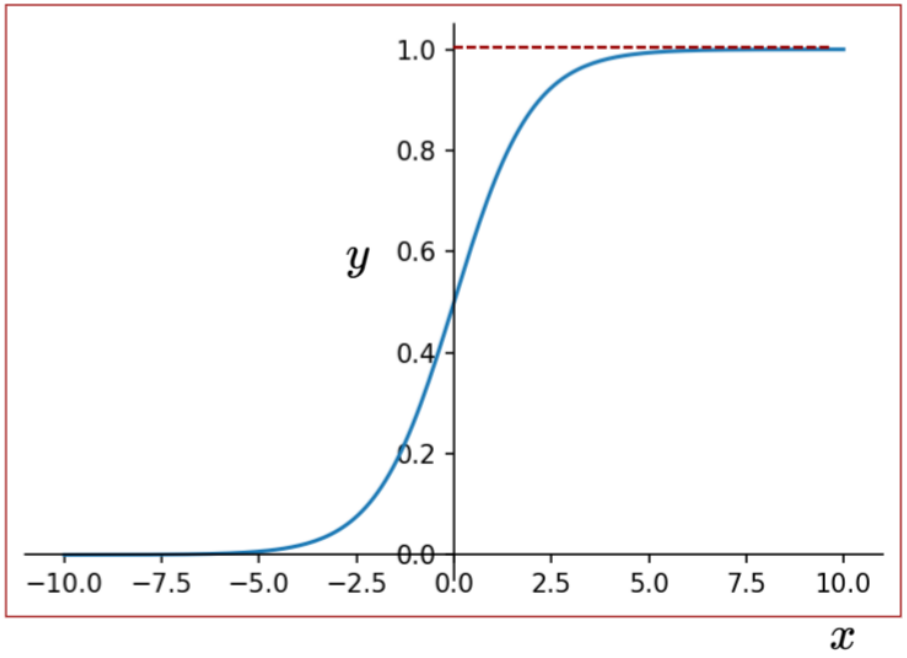
While calculating ∇w_2 at some point in the chain rule we will encounter

$$\frac{\partial h3}{\partial a_3} = \frac{\partial \sigma(a_3)}{\partial a_3} = \sigma(a_3)(1 - \sigma(a_3))$$

What is the consequence of this ?

To answer this question let us first understand the concept of saturated neuron ?

Why using Sigmoid is not a good idea



A sigmoid neuron is said to have saturated when $\sigma(x) = 0$ or $\sigma(x) = 1$

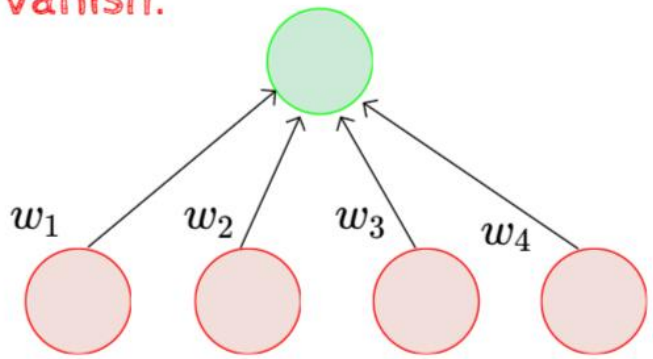
What would the gradient be at saturation?

Well it would be 0 (you can see it from the plot or from the formula that we derived)

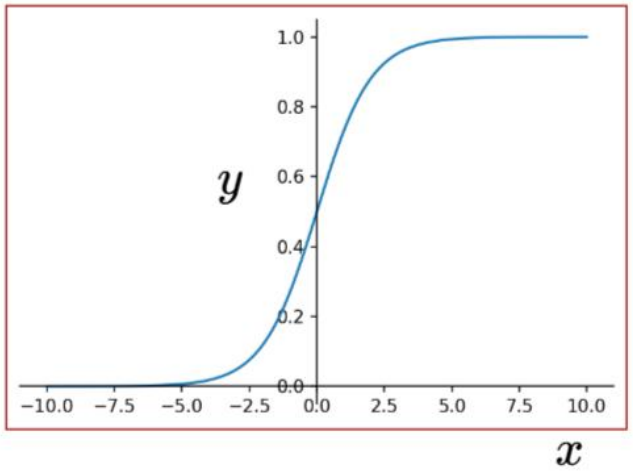
Saturated neurons thus cause the gradient to vanish.

Why using Sigmoid is not a good idea

Saturated neurons thus cause the gradient to vanish.



$$\sigma\left(\sum_{i=1}^{n} w_i x_i\right)$$



But why would the neurons saturate ?

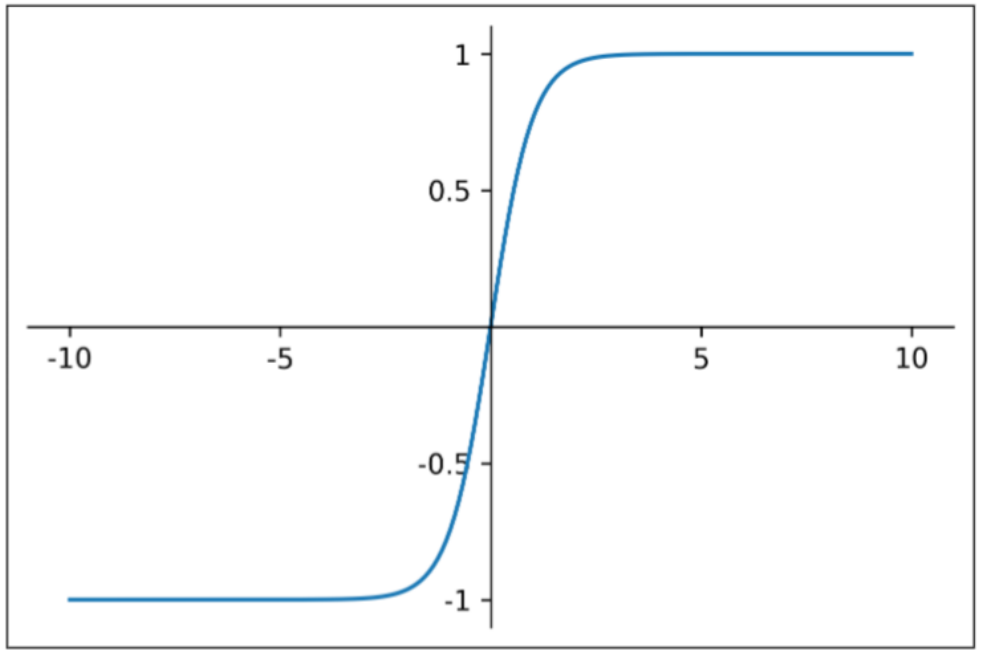
Consider what would happen if we use sigmoid neurons and initialize the weights to very high values ?

The neurons will saturate very quickly

The gradients will vanish and the training will stall (more on this later)

Tanh

$tanh(x)$



Compresses all its inputs to the range [-1,1]

Zero centered

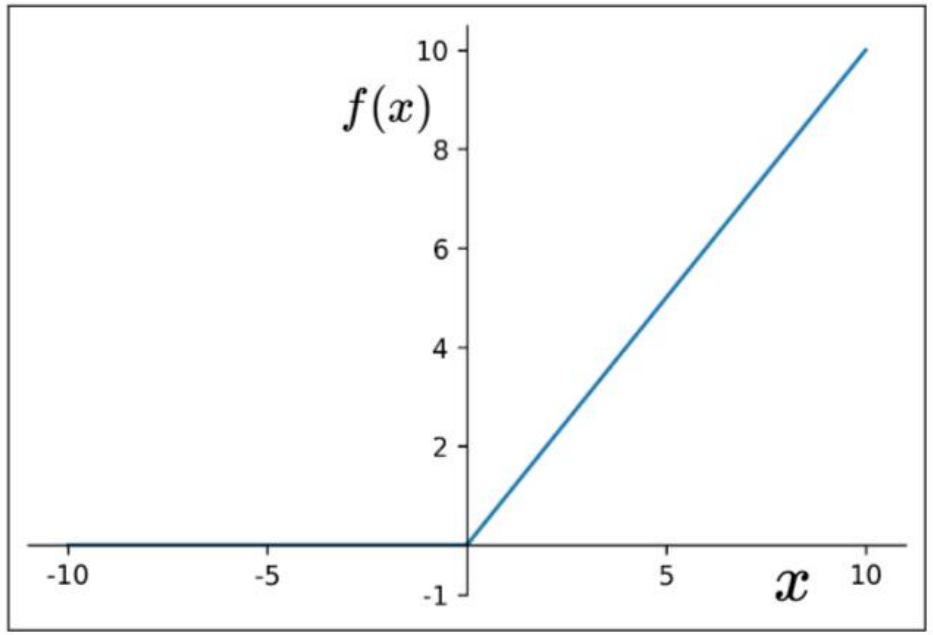
What is the derivative of this function?

$$\frac{\partial tanh(x)}{\partial x} = (1 - tanh^2(x))$$

The gradient still vanishes at saturation

Also computationally expensive

ReLU (Rectified Linear Unit)



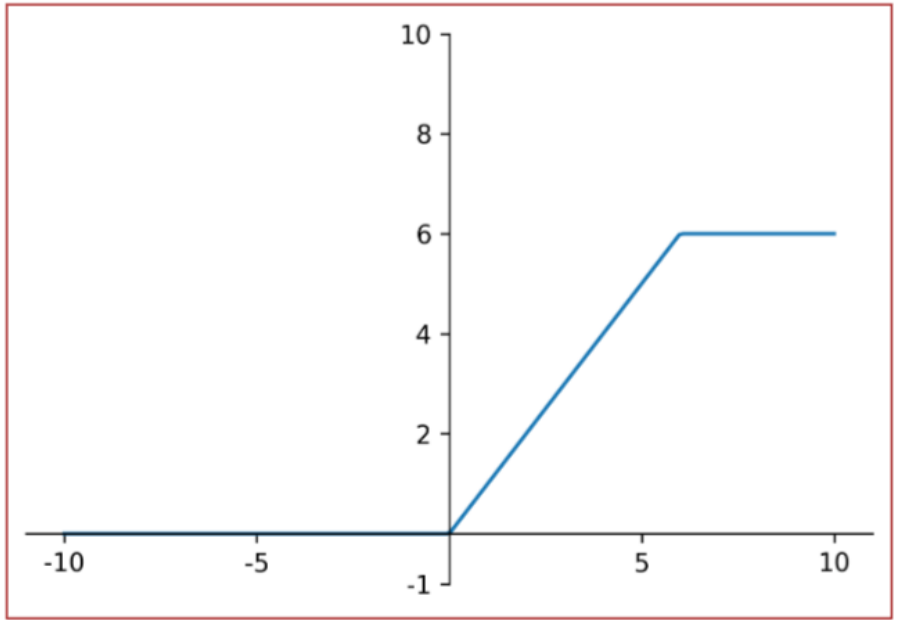
$$f(x) = max(0, x)$$

Is this a non-linear function?

Indeed it is!

Small Activity on Representation Power and Activation Functions

ReLU (Rectified Linear Unit)



$$f(x) = max(0, x) - max(0, x - 6)$$

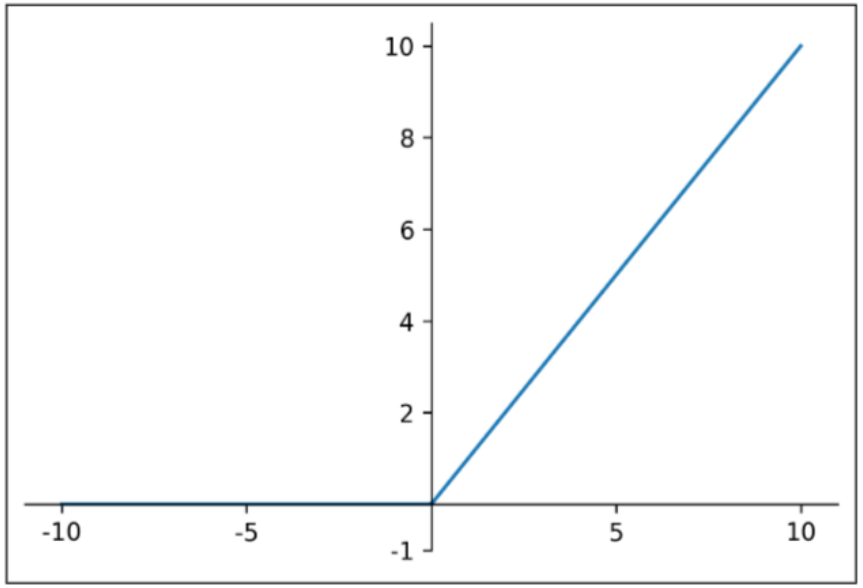
Is this a non-linear function?

Indeed it is!

In fact we can combine two ReLU units to recover a piecewise linear approximation of the scaled sigmoid function

It is also called ReLU6 (6 denotes the maximum value of the ReLU)

ReLU (Rectified Linear Unit)



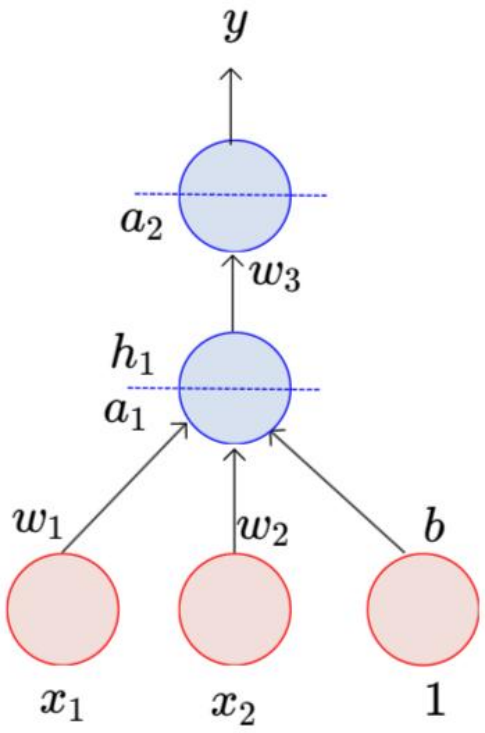
$$f(x) = max(0, x)$$

Advantages of ReLU

- Does not saturate in the positive region
- Computationally efficient
- In practice converges much faster than $sigmoid/tanh$[1]

[1]ImageNet Classification with Deep Convolutional Neural Networks- Alex Krizhevsky Ilya Sutskever, Geoffrey E. Hinton, 2012

ReLU: Dead Neurons



In practice there is a caveat

Let's see what is the derivative of ReLU(x)

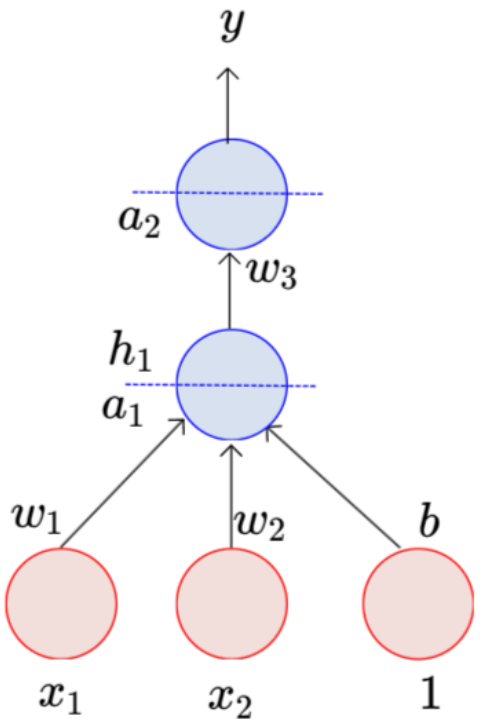
$$\frac{\partial ReLU(x)}{\partial x} = 0 \quad if \quad x < 0$$

$$= 1 \quad if \quad x > 0$$

Now consider the given network

What would happen if at some point a large gradient causes the bias b to be updated to a large negative value?

ReLU: Dead Neurons



$$w_1x_1 + w_2x_2 + b < 0 \quad [if \quad b \quad << 0]$$

The neuron would output 0 [dead neuron]

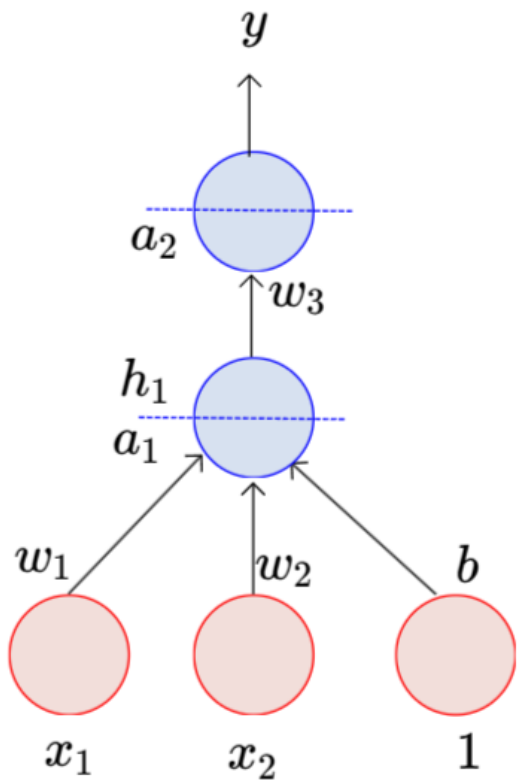
Not only would the output be 0 but during backpropagation even the gradient $\frac{\partial h_1}{\partial a_1}$ would be zero

The weights w_1, w_2 and b will not get updated [∵ there will be a zero term in the chain rule]

$$\nabla w_1 = \frac{\partial \mathscr{L}(\theta)}{\partial y} \frac{\partial y}{\partial a_2} \frac{\partial a_2}{\partial h_1} \frac{\partial h_1}{\partial a_1} \frac{\partial a_1}{\partial w_1}$$

The neuron will now stay dead forever!!

ReLU: Dead Neurons

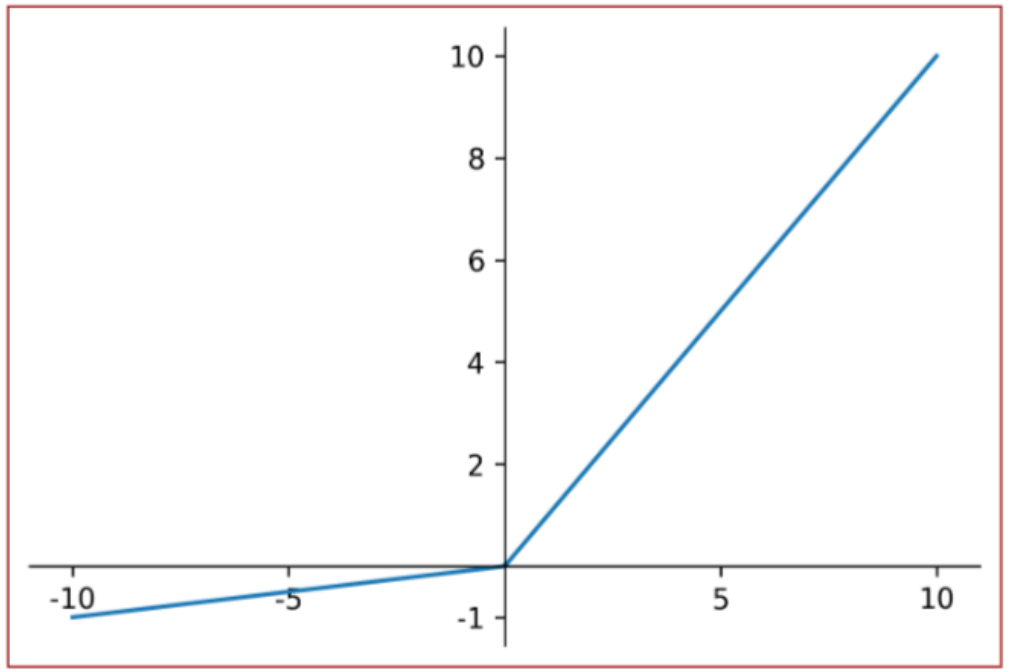


In practice a large fraction of ReLU units can die if the learning rate is set too high

It is advised to initialize the bias to a positive value (0.01)

Use other variants of ReLU (as we will soon see)

Leaky ReLU



$$f(x) = max(0.1x, x)$$

No saturation

Will not die (0.1x ensures that at least a small gradient will flow through)

Computationally efficient

Close to zero centered ouputs

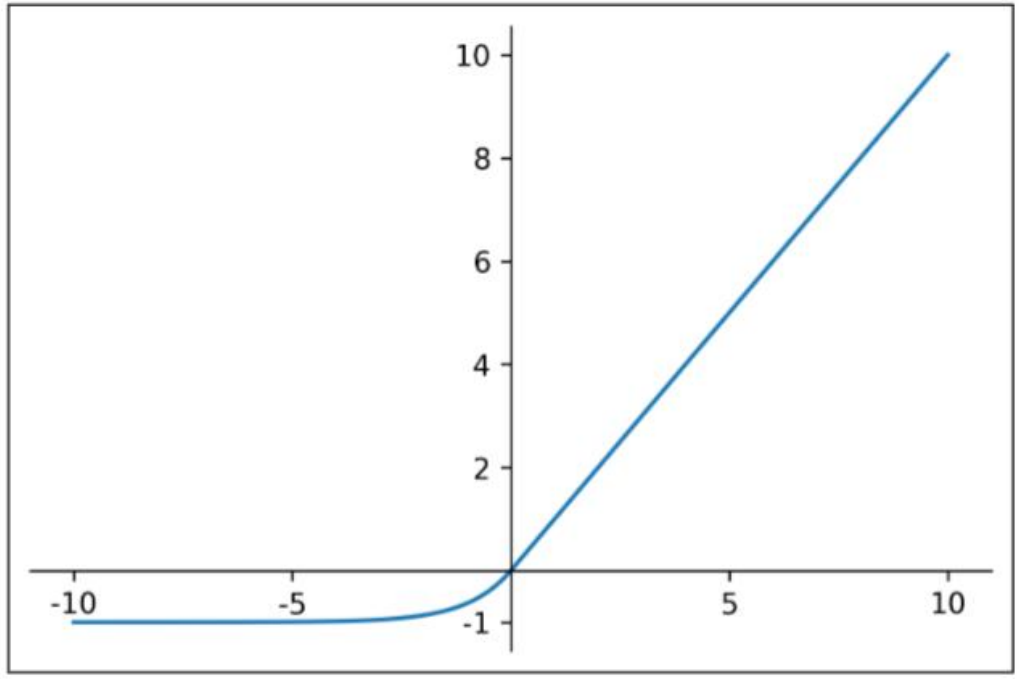
Parametric ReLU

$$f(x) = max(\alpha x, x)$$

α *is a parameter of the model*

α *will get updated during backpropagation*

Leaky ReLU



All benefits of ReLU

$ae^x - 1$ ensures that at least a small gradient will flow through

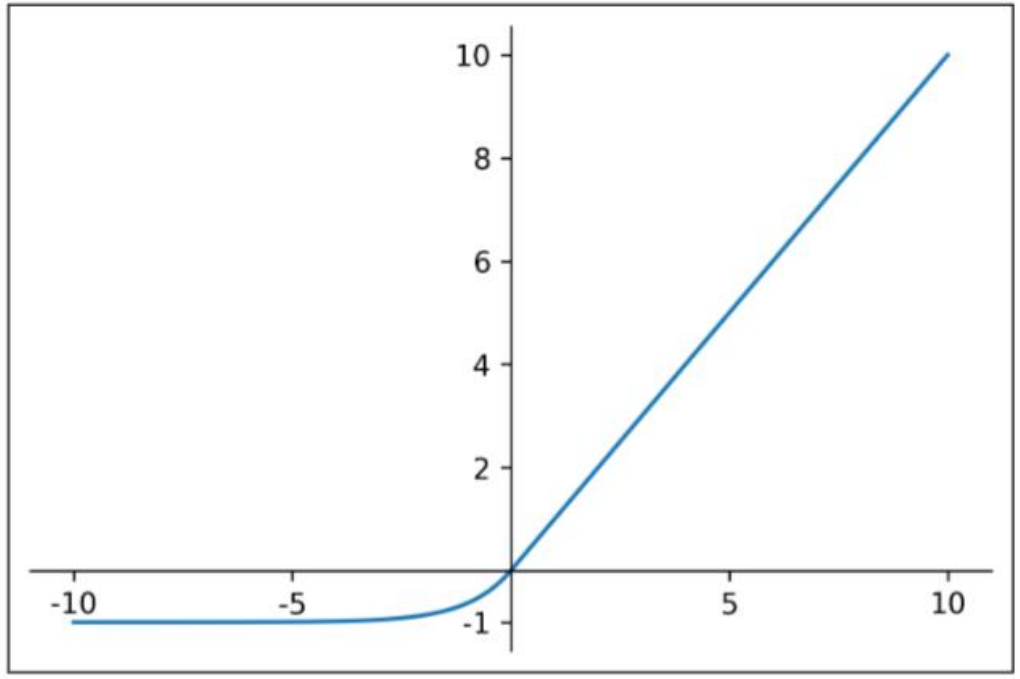
Close to zero centered outputs

Expensive (requires computation of exp(x))

$$f(x) = x \quad if \quad x > 0$$

$$= ae^x - 1 \quad if \quad x \leq 0$$

Leaky ReLU



All benefits of ReLU

$ae^x - 1$ ensures that at least a small gradient will flow through

Close to zero centered outputs

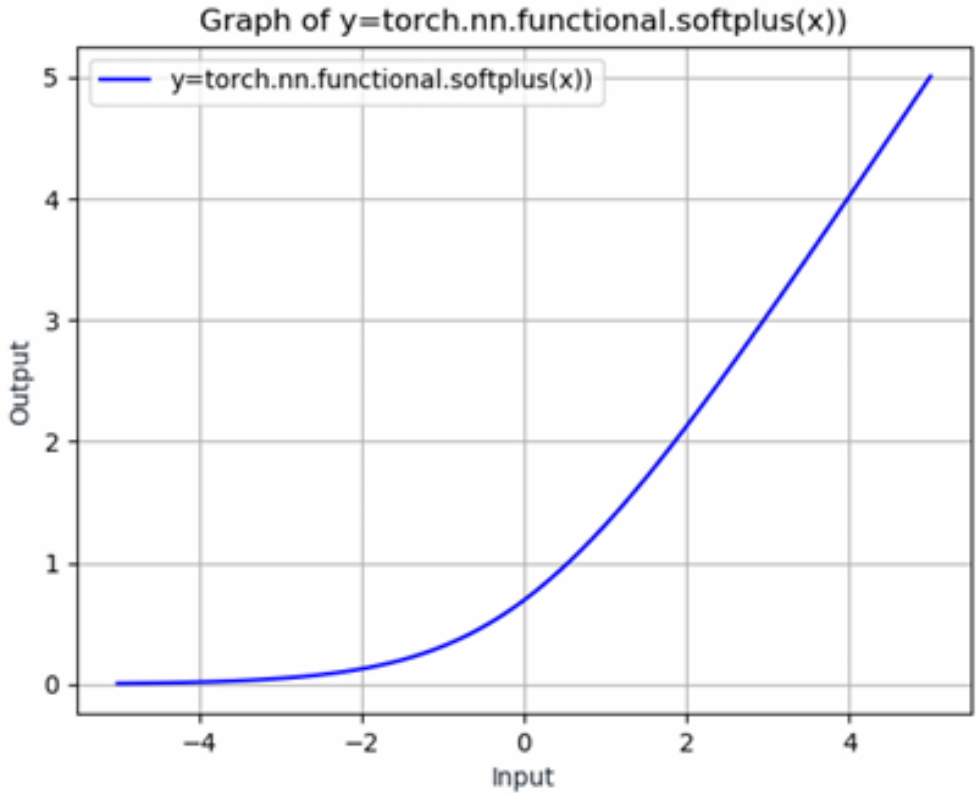
Expensive (requires computation of exp(x))

$$f(x) = x \quad if \quad x > 0$$

$$= ae^x - 1 \quad if \quad x \leq 0$$

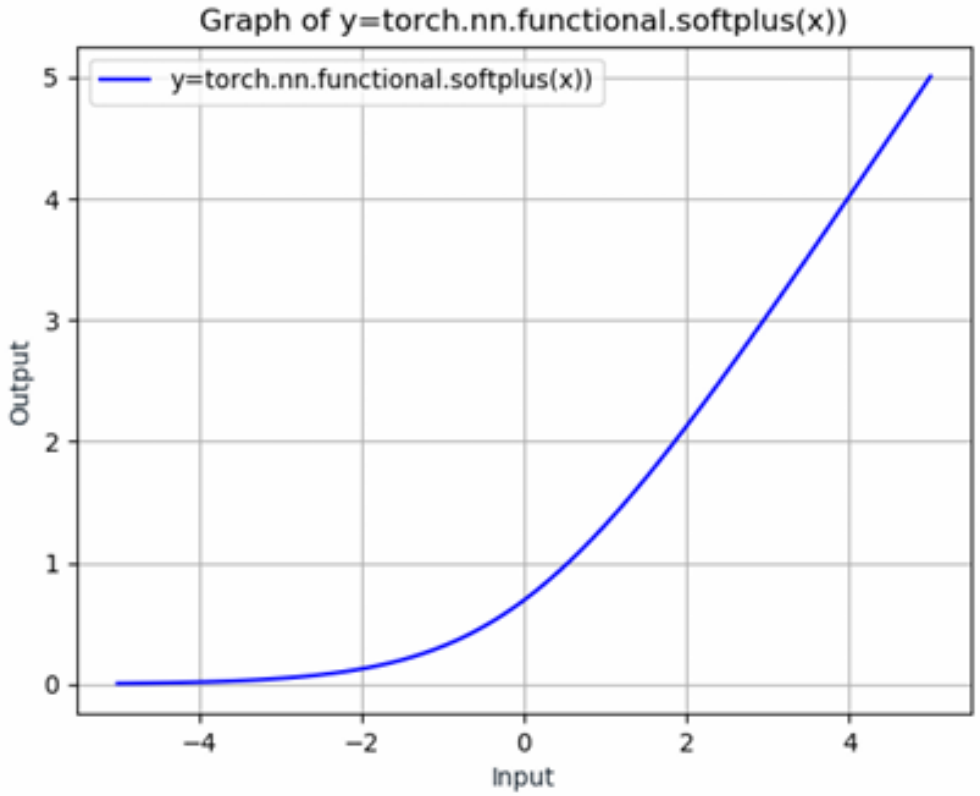
SoftPlus

- $Output = \frac{1}{\beta} \cdot \log\left(1 + e^{(\beta \cdot x)}\right)$
- Smooth approximation of ReLU
- Output always positive
- Numerical stability:
  use linear if
  $(\beta \cdot x) > threshold$



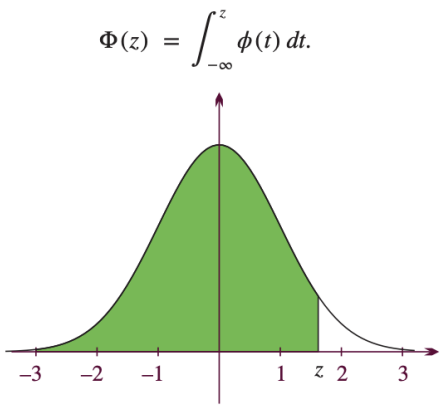
SoftPlus

- $Output = \frac{1}{\beta} \cdot \log\left(1 + e^{(\beta \cdot x)}\right)$
- Smooth approximation of ReLU
- Output always positive
- Numerical stability:
  use linear if
  $(\beta \cdot x) > threshold$

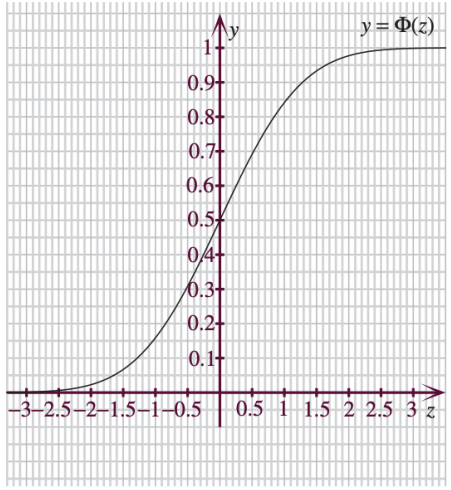


GeLU

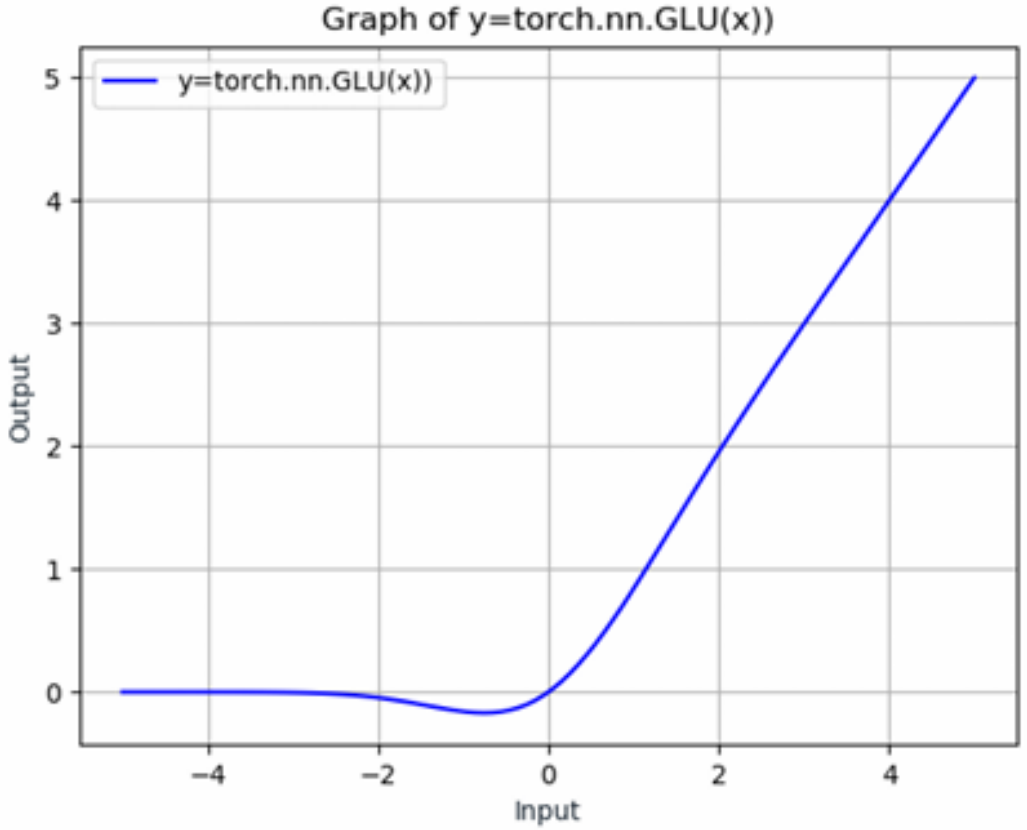
- $Output = x \cdot \Phi(x)$
- $\Phi(x)$ is the Cumulative Distribution Function for Gaussian Distribution.



This is the graph of the standard normal probability density function $\phi(z)$.



This is the graph of the standard normal cumulative distribution function $\Phi(z)$.



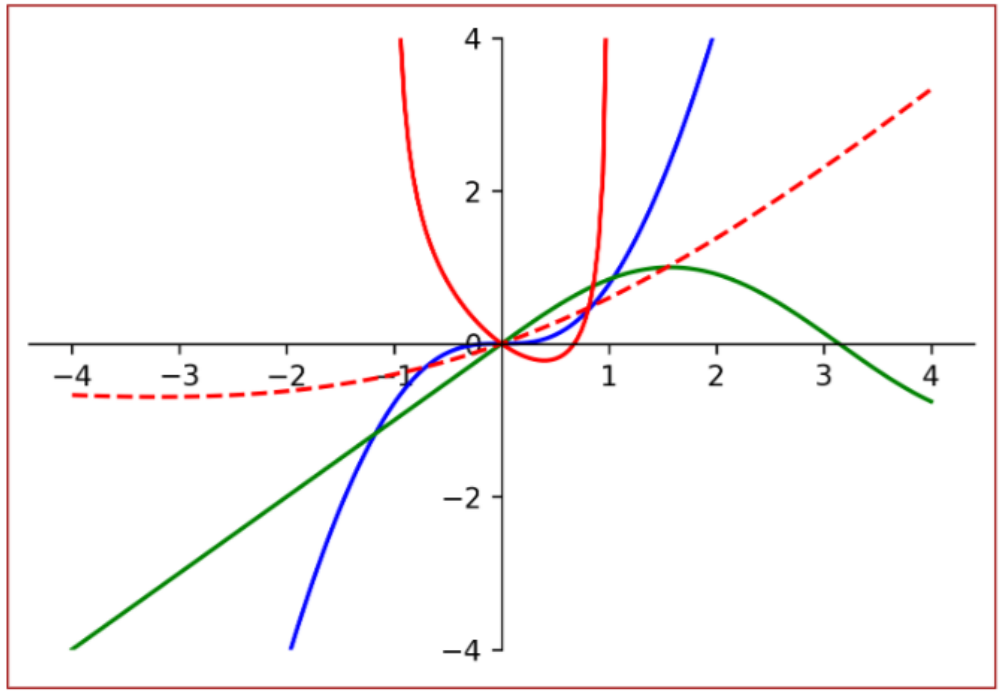
Automatic Search for Activation functions

Figure on the right shows top four activation functions found by the search method.

According to the search method, the best activation takes the following form

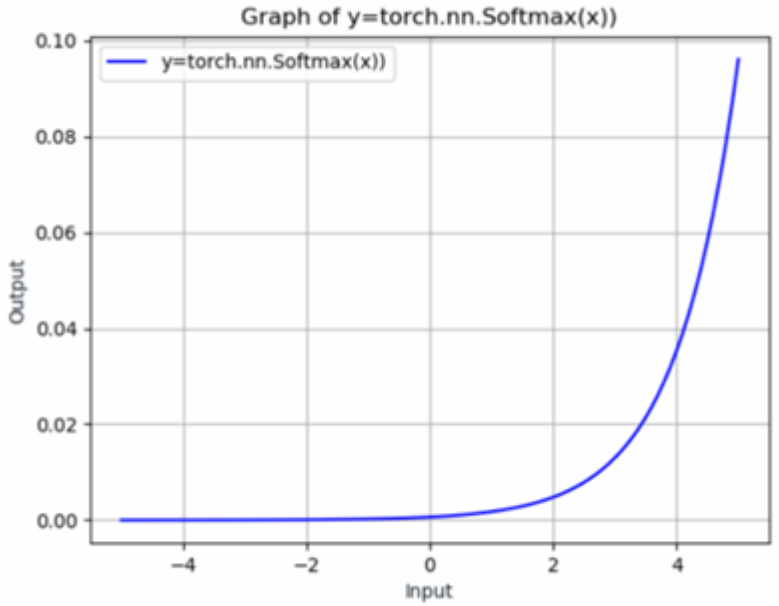
$$f(x) = x\sigma(\beta x)$$

It is named as SWISH. Here, β can be a constant like in GELU $\beta = 1.702$ or a learnable parameter

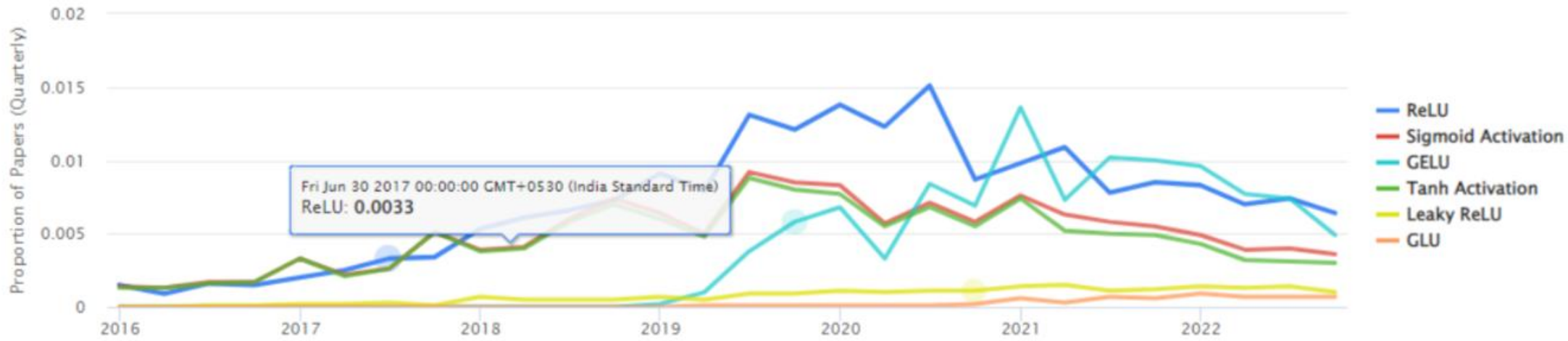


SoftMax

- $Output = \frac{e^{x_i}}{\sum_j e^{x_j}}$
- Applies the Softmax function to an n-dimensional input Tensor rescaling them so that the elements of the n-dimensional output Tensor lie in the range [0,1] and sum up to 1.



Usage Over Time



Things to Remember

Sigmoids are bad

ReLU is more or less the standard unit for Convolutional Neural Networks

Can explore Leaky ReLU/Maxout/ELU

tanh sigmoids are still used in LSTMs/RNNs (we will see more on this later)

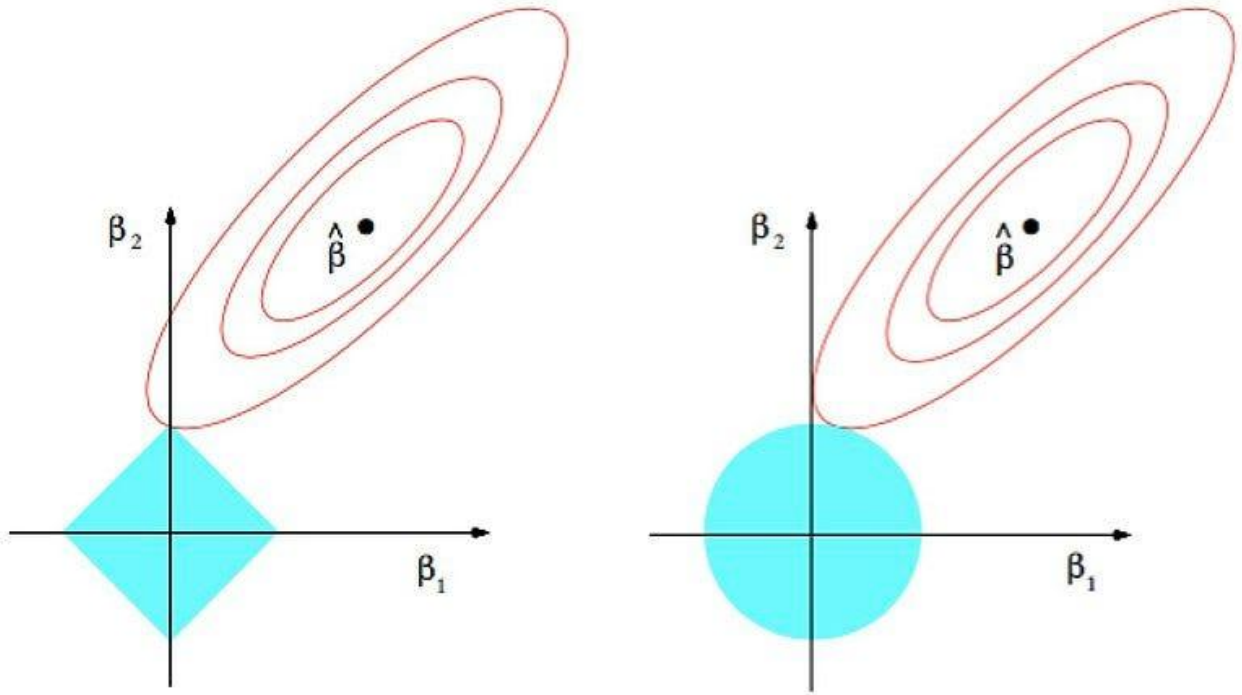
GELU is most commonly used in Transformer based architectures like BERT, GPT

Periodic activations, SIREN

- Difficult convergence properties for general problems
- Has been used for implicit representations (find a continuous function that represents sparse input data, e.g. an image)
- https://vsitzmann.github.io/siren/

$$\Phi(\mathbf{x}) = \mathbf{W}_n \left(\phi_{n-1} \circ \phi_{n-2} \circ \ldots \circ \phi_0\right)(\mathbf{x}) + \mathbf{b}_n, \quad \mathbf{x}_i \mapsto \phi_i(\mathbf{x}_i) = \sin\left(\mathbf{W}_i \mathbf{x}_i + \mathbf{b}_i\right).$$

Regularization

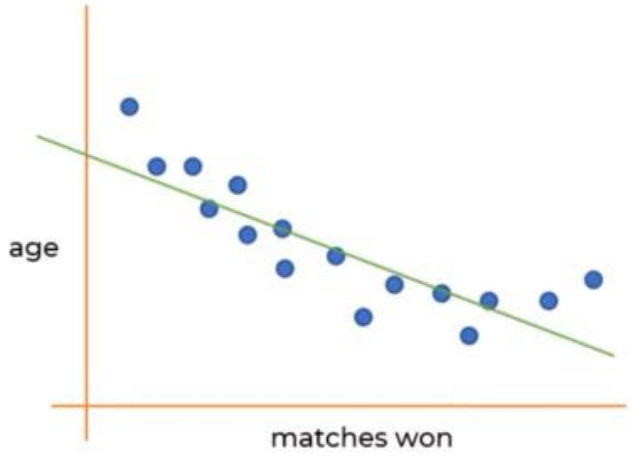


What is regularization?

- In general: any method to prevent overfitting or help the optimization
- Specifically: additional terms in the training optimization objective to prevent overfitting or help the optimization

What is Overfitting

underfit

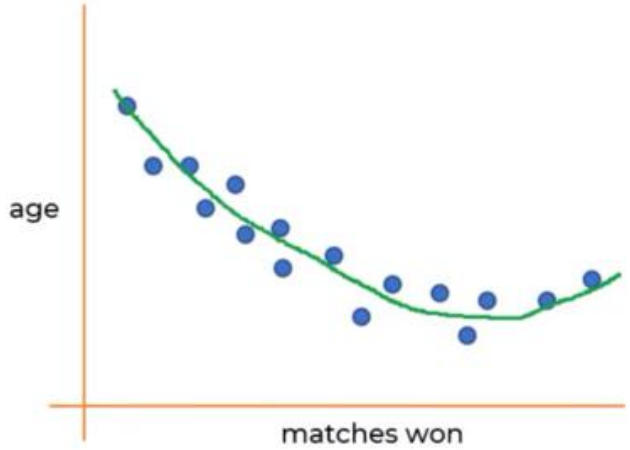


$$\text{match won} = \theta_0 + \theta_1 * age$$

overfit

$$\text{match won} = \theta_0 + \theta_1 * age \quad +\theta_2 * age^2 \quad +\theta_3 * age^3 \quad +\theta_4 * age^4$$

balanced fit



$$\text{match won} = \theta_0 + \theta_1 * age \quad +\theta_2 * age^2$$

What is Overfitting

Overfitting example: regression using polynomials

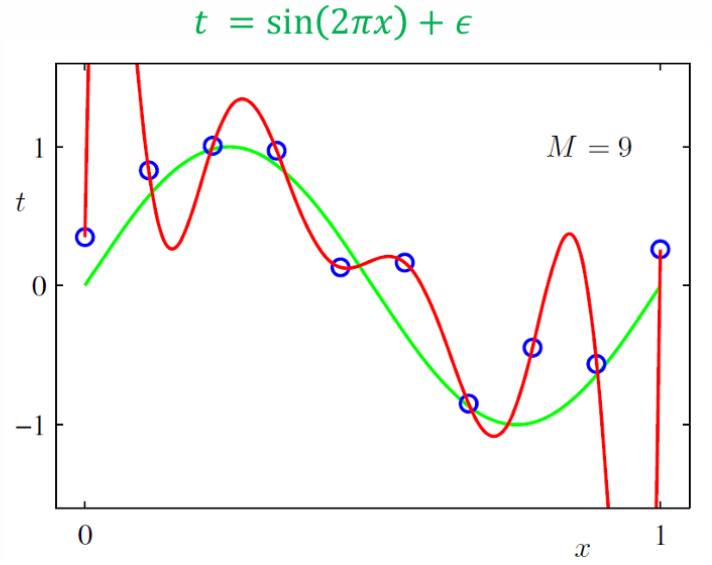


Figure from *Machine Learning and Pattern Recognition*, Bishop

Overfitting example: regression using polynomials

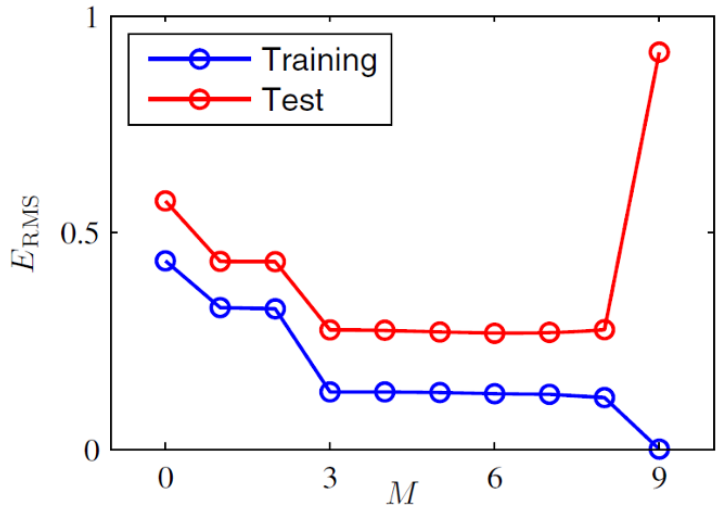


Figure from *Machine Learning and Pattern Recognition*, Bishop

Overfitting

- Key: empirical loss and expected loss are different

- Smaller the data set, larger the difference between the two
- Larger the hypothesis class, easier to find a hypothesis that fits the difference between the two
  - Thus has small training error but large test error (overfitting)

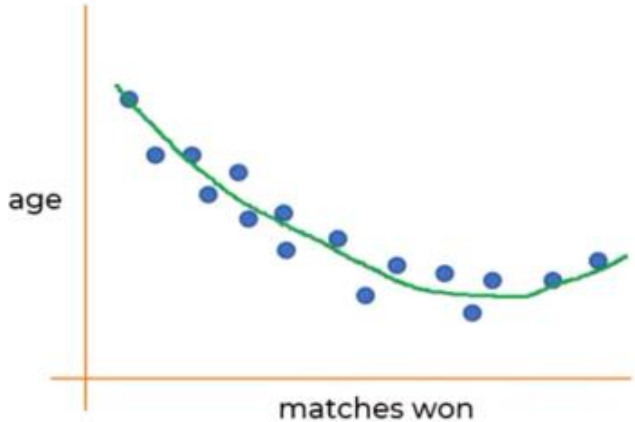
- Larger data set helps
- Throwing away useless hypotheses also helps (regularization)

Different Ways of Regularization

How can we go from the overfit example to the balanced fit?

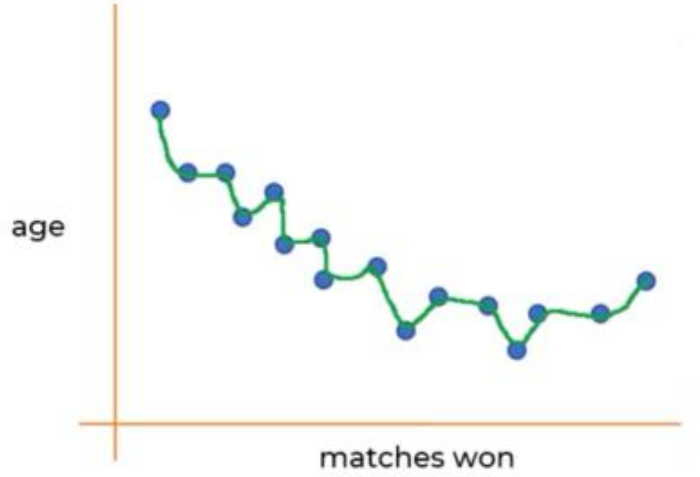
$$\text{match won} = \theta_0 + \theta_1 * age \quad + \theta_2 * age^2 + \theta_3 * age^3 + \theta_4 * age^4$$

Try to make θ_3 and θ_4 almost close to zero



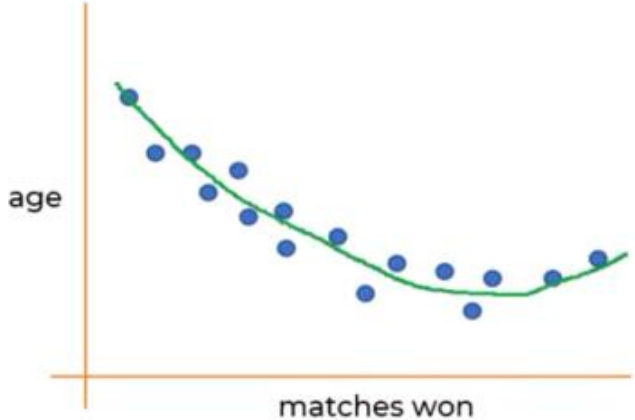
$$\text{match won} = \theta_0 + \theta_1 * age \quad + \theta_2 * age^2$$

Regularization as a hard constraint



$$\text{match won} = \theta_0 + \theta_1 * age \quad + \theta_2 * age^2 + \theta_3 * age^3 + \theta_4 * age^4$$

Try to make θ_3 and θ_4 almost close to zero



$$\text{match won} = \theta_0 + \theta_1 * age \quad + \theta_2 * age^2$$

Regularization as soft constraint

Mean Squared Error

$$mse = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - h_\theta(x_i)\right)^2$$

$$h_\theta(x_i) = \theta_0 + \theta_1 x_1 + \theta_2 x_2^2 + \theta_3 x_3^3$$

Regularization as soft constraint

L2 Regularization

$$mse = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - h_\theta(x_i)\right)^2 + \lambda \sum_{i=1}^{n} \theta_i^2$$

$$h_\theta(x_i) = \theta_0 + \theta_1 x_1 + \theta_2 x_2^2 + \theta_3 x_3^3$$

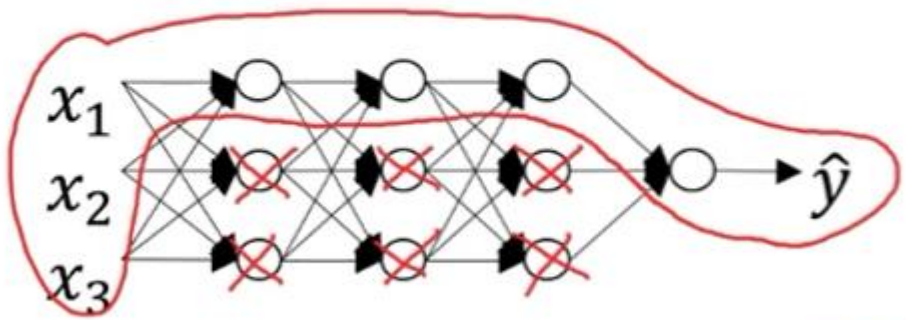
Regularization as soft constraint

L1 Regularization

$$mse = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - h_\theta(x_i)\right)^2 + \lambda \sum_{i=1}^{n}|\theta_i|$$

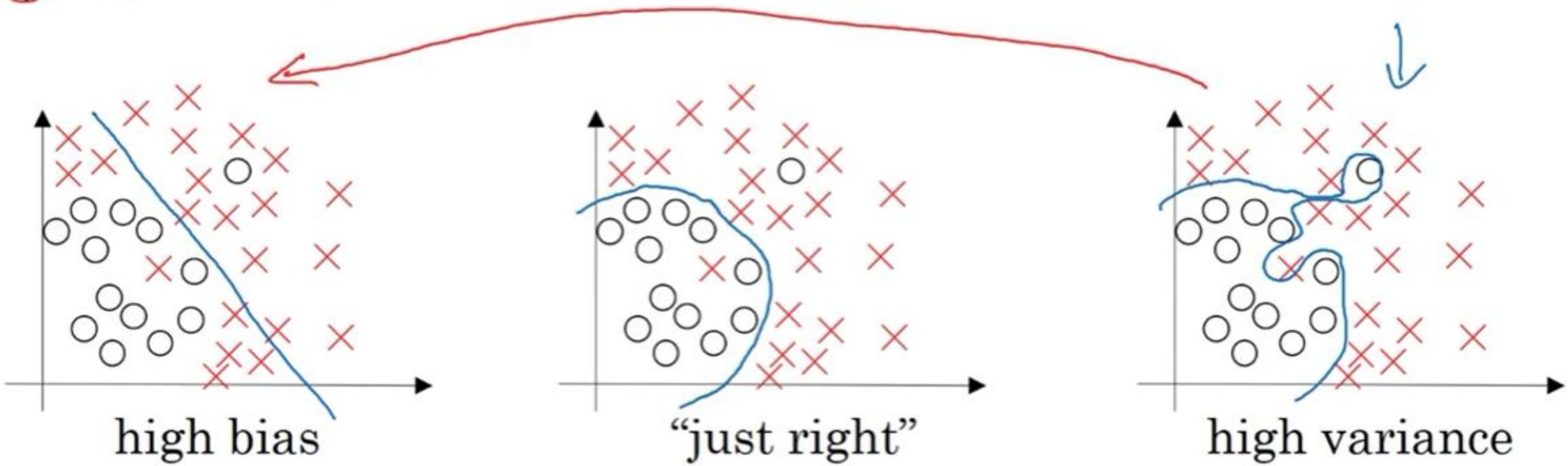
$$h_\theta(x_i) = \theta_0 + \theta_1 x_1 + \theta_2 x_2^2 + \theta_3 x_3^3$$

How does regularization prevent overfitting?



$$J(w^{[l]}, b^{[l]}) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} \|w^{[l]}\|_F^2$$

$$w^{[l]} \approx 0$$



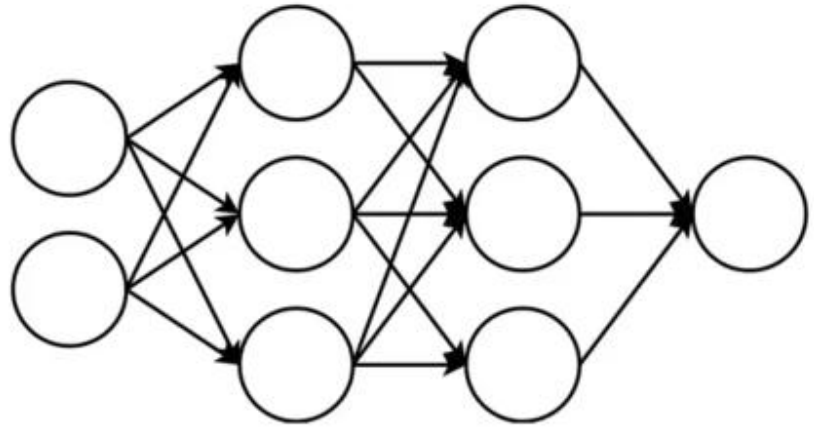
Loss with L2 Regularization

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N} Cost(y_i, f_\theta(x_i)) + \lambda\|\theta\|_2^2$$

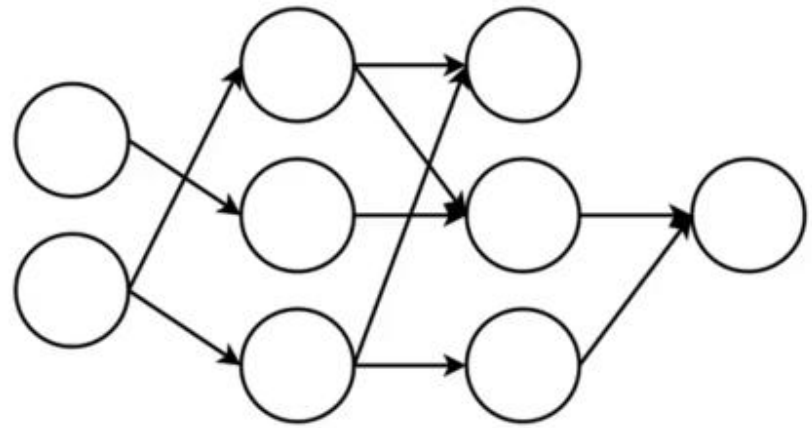
As this becomes lower
while optimizing the loss…

The more likely this
scenario becomes…

w/o regularization



w/ regularization



L1 vs L2 Regularization

L1 Regularization

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N} Cost(y_i, f_\theta(x_i)) + \lambda\|\theta\|_1$$

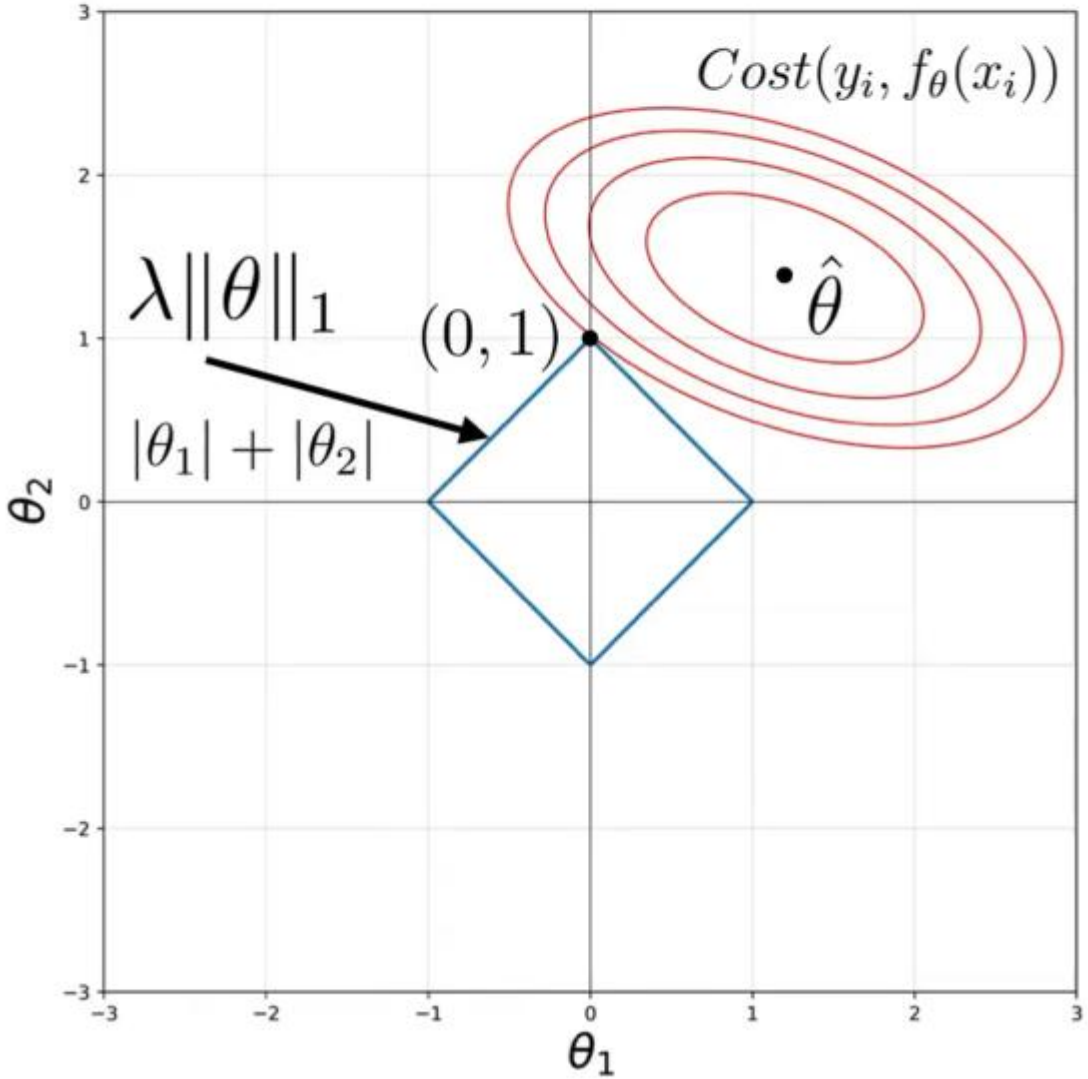
Tends to produce sparse weights, meaning that many of the weights become **exactly zero**

L2 Regularization

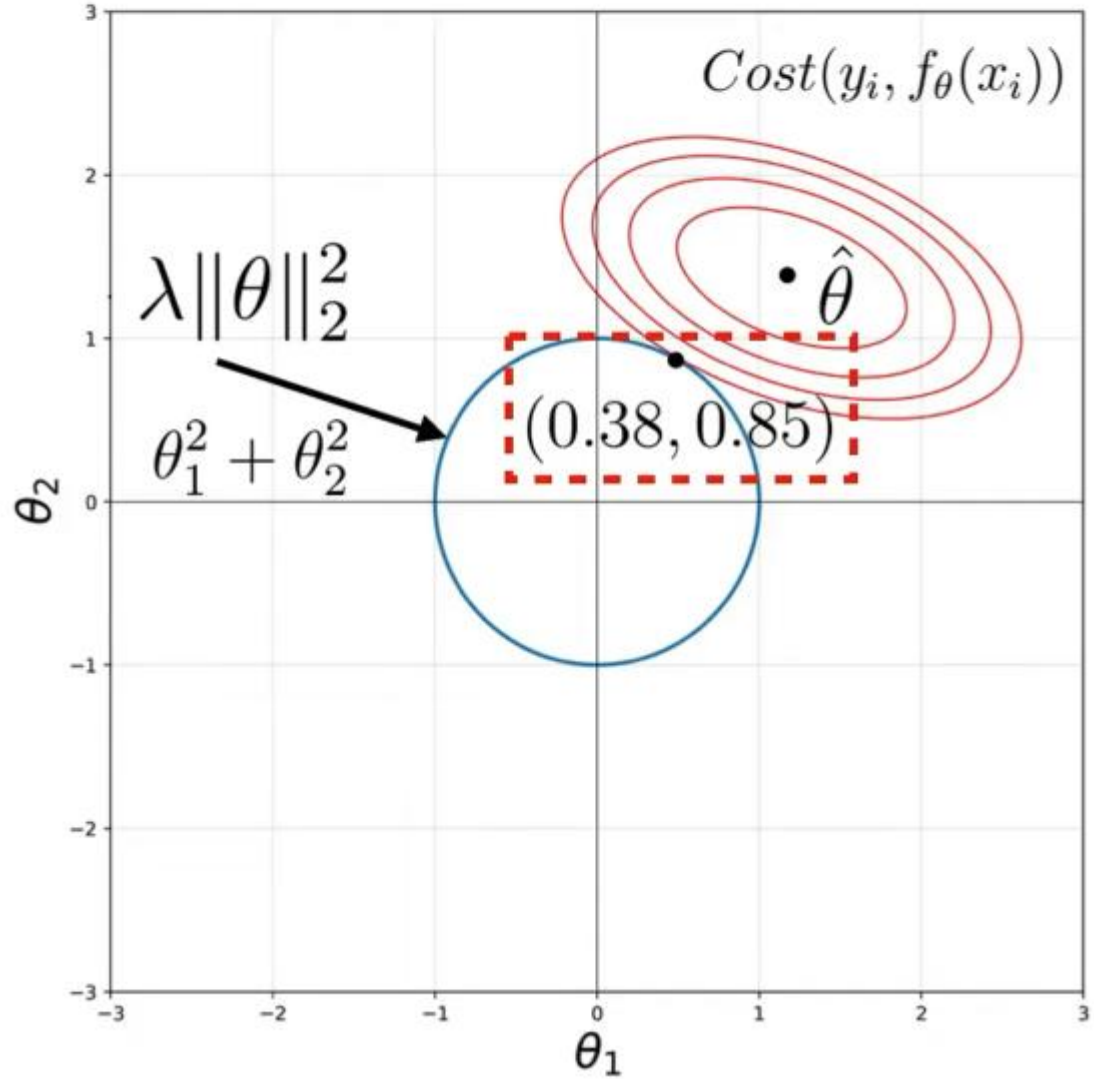
$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N} Cost(y_i, f_\theta(x_i)) + \lambda\|\theta\|_2^2$$

Prefers to keep all weights small but **not exactly zero**

L1 Regularization

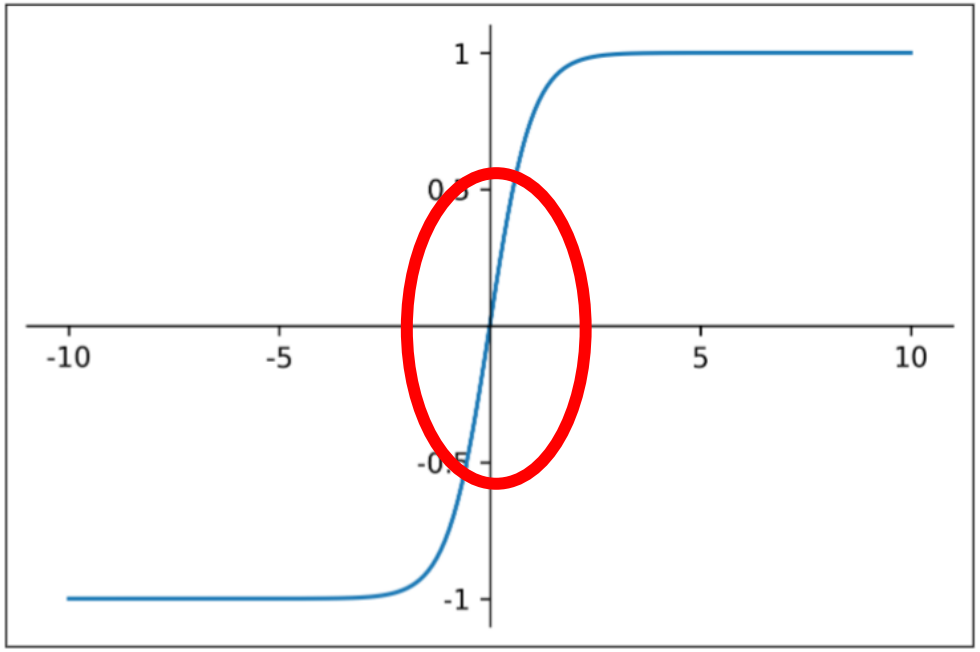


L2 Regularization



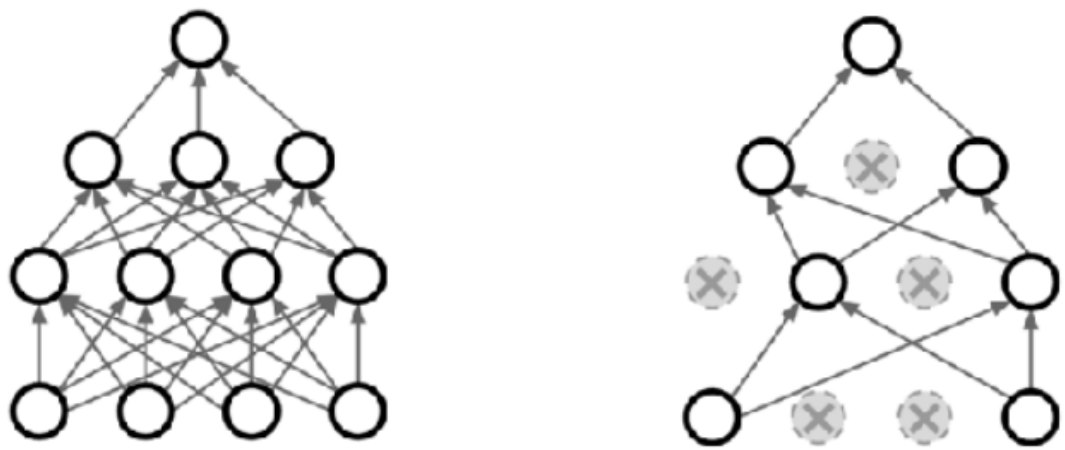
Possible intuition (when using Tanh)

$$tanh(x)$$



Regularisation: Dropout

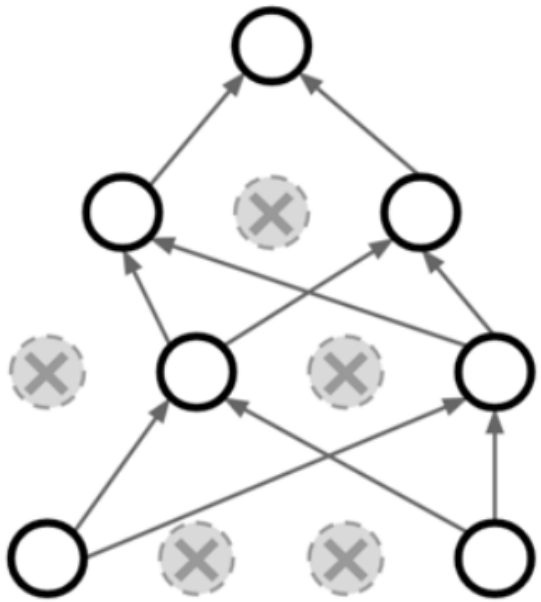
- We add something to our model to prevent it from fitting our training model too well, so it could perform better on unseen data!
- In dropout, we randomly set some neurons to zero for each forward pass.
- Dropout is like training a large ensemble of models!



Robotics Research Center

Regularization: Dropout

How can this possibly be a good idea?

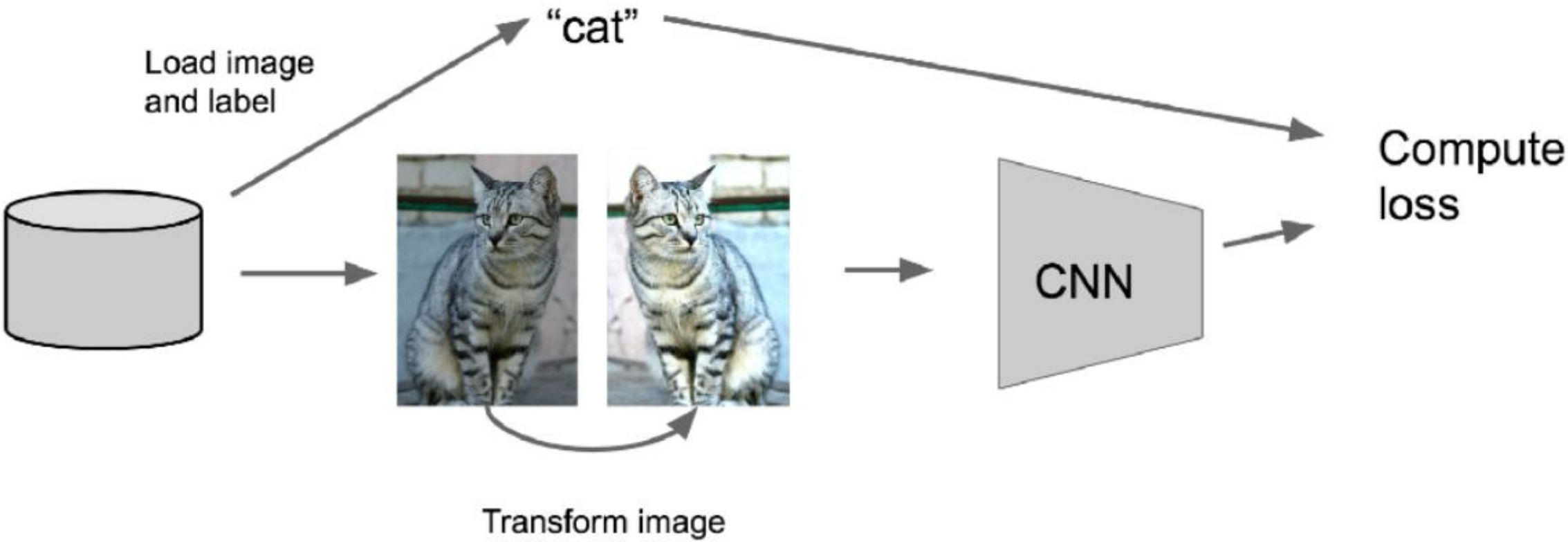


Forces the network to have a redundant representation;
Prevents co-adaptation of features

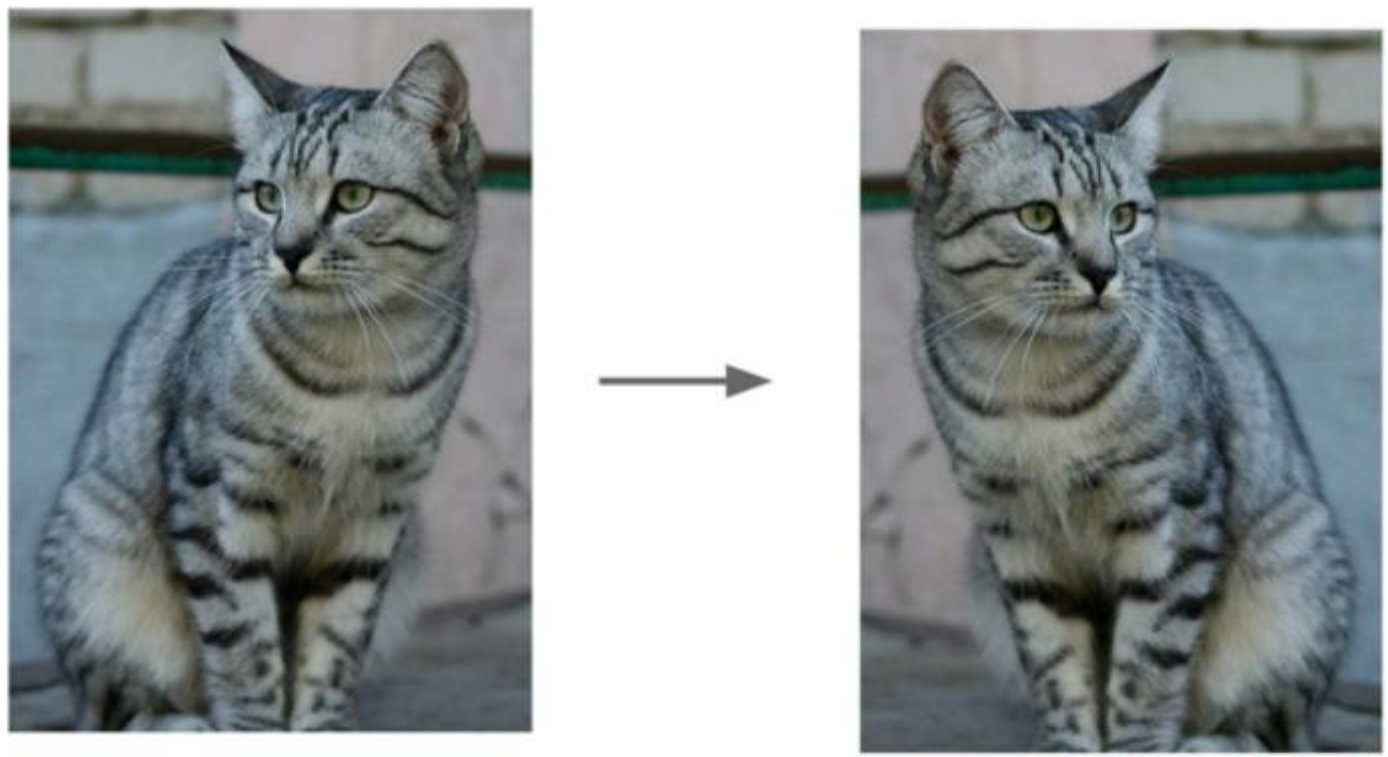
We want to prevent over fitting and co adaptation of all the features. So even if the cat doesn't have a tail or isn't furry, the model should be able to detect the cat-ness. This helps increase performance for unseen data.

Robotics Research Center

Regularisation: Data Augmentation



Horizontal Flip



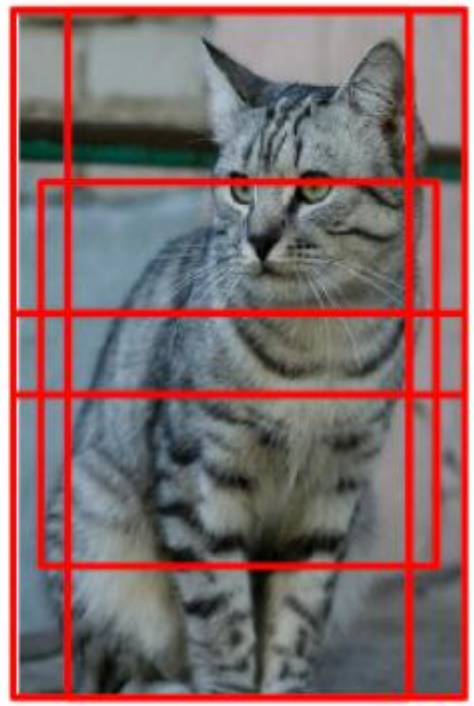
Data Augmentation

Random crops and scales

Training: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch



Testing: average a fixed set of crops

ResNet:

1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224 x 224 crops: 4 corners + center, + flips

Color Jitter

Simple: Randomize
contrast and brightness

Random mix/combinations of :

- translation
- rotation
- stretching
- shearing,
- lens distortions, … (go crazy)

Regularization: DropConnect

Training: Drop connections between neurons (set weights to 0)

Testing: Use all the connections

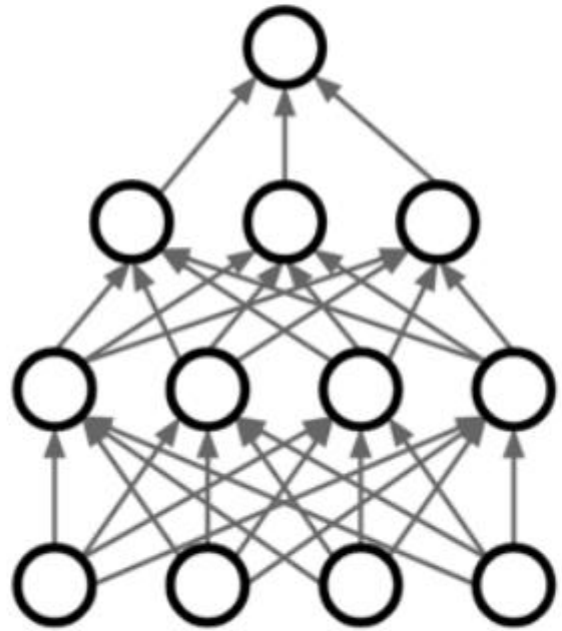
Examples:

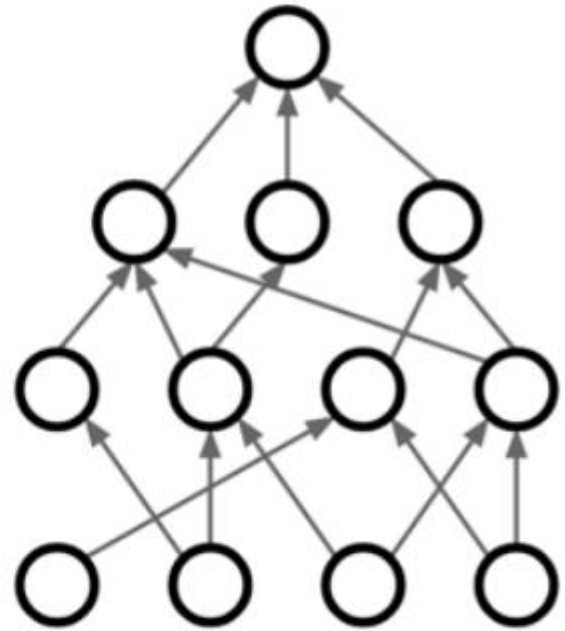
Dropout

Batch Normalization

Data Augmentation

DropConnect





Wan et al, "Regularization of Neural Networks using DropConnect", ICML 2013

Regularization: Mixup

Training: Train on random blends of images

Testing: Use original images

Examples:

Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling
Stochastic Depth
Cutout / Random Crop
Mixup

Randomly blend the pixels of pairs of training images, e.g. 40% cat, 60% dog

Target label:
cat: 0.4
dog: 0.6

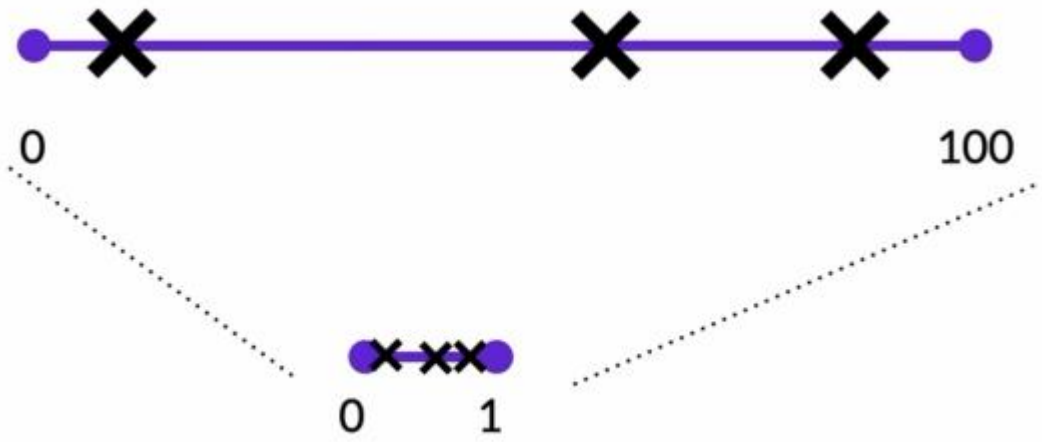
Zhang et al, "*mixup*: Beyond Empirical Risk Minimization", ICLR 2018

Batch Normalization

What is normalization?

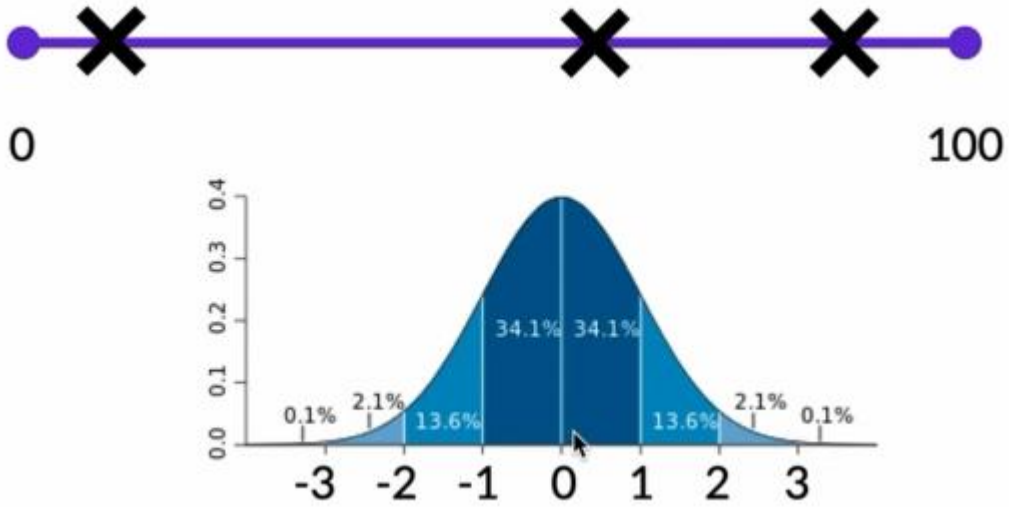
Normalization

Collapse inputs to be between 0 and 1.

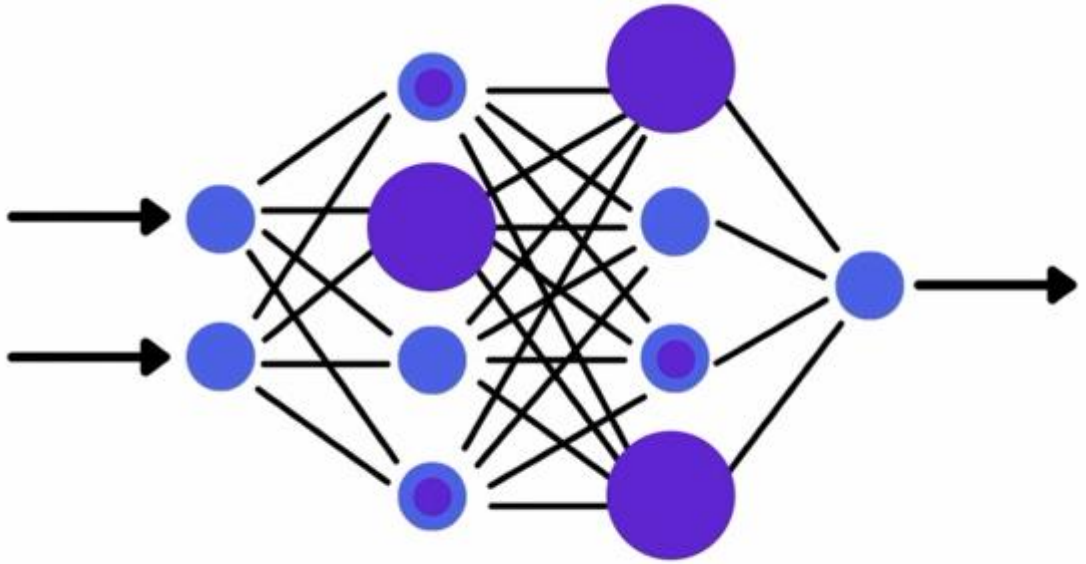


Standardization

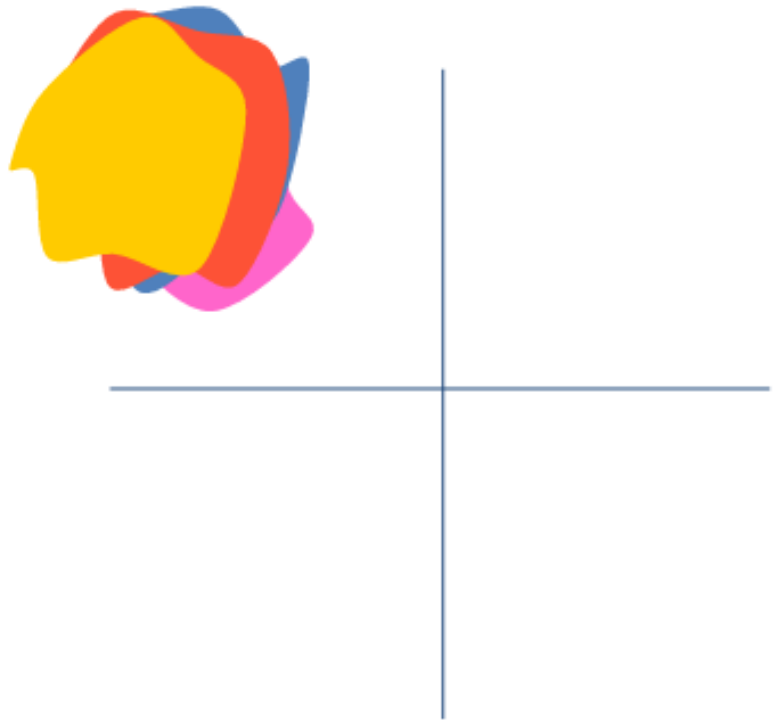
Make mean 0 and variance 1.



Why do we need normalization?

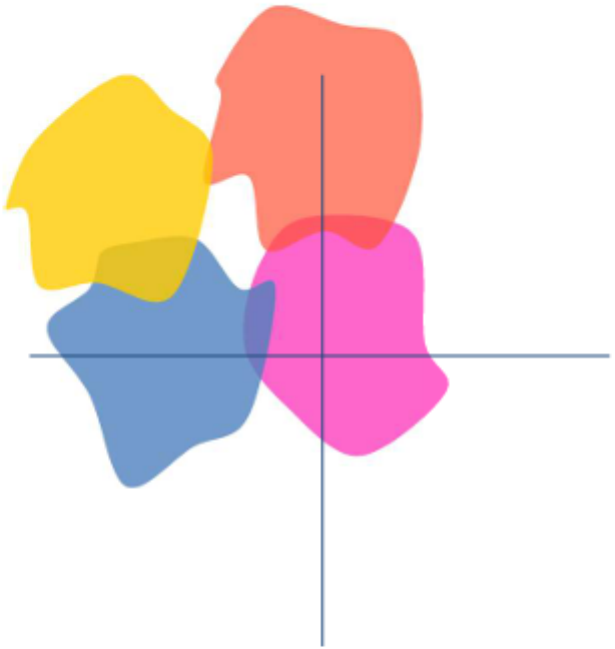


The problem of covariate shifts



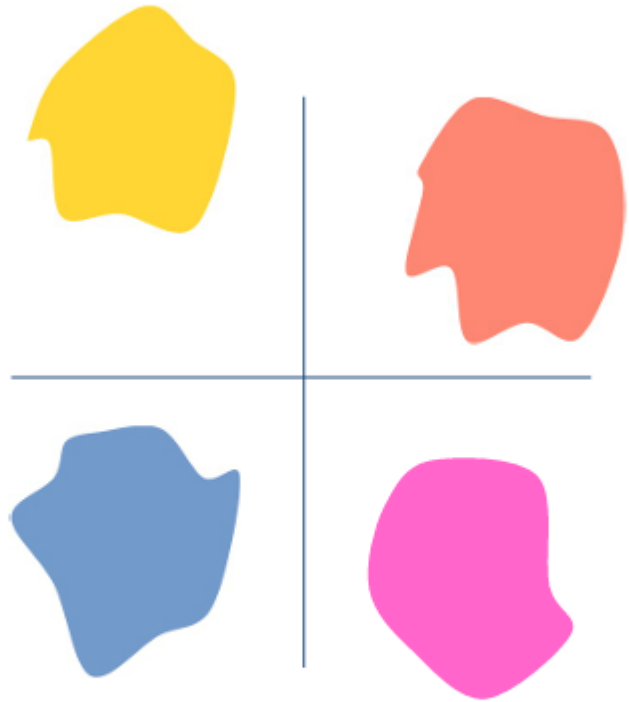
- Training assumes the training data are all similarly distributed
  - Minibatches have similar distribution

The problem of covariate shifts



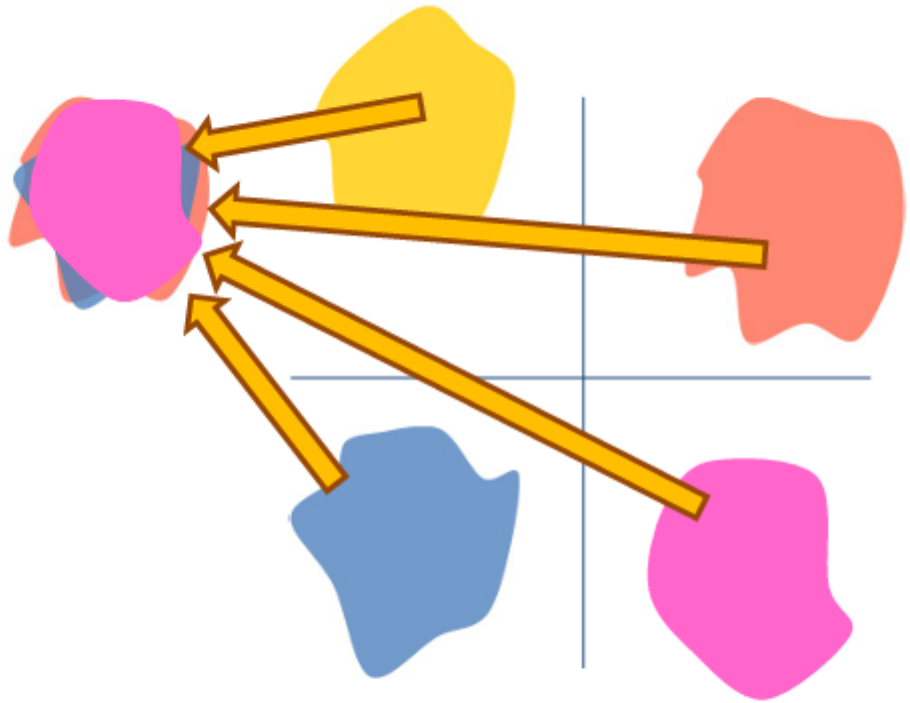
- Training assumes the training data are all similarly distributed
  - Minibatches have similar distribution
- In practice, each minibatch may have a different distribution
  - A “covariate shift”
  - Which may occur in *each* layer of the network

The problem of covariate shifts

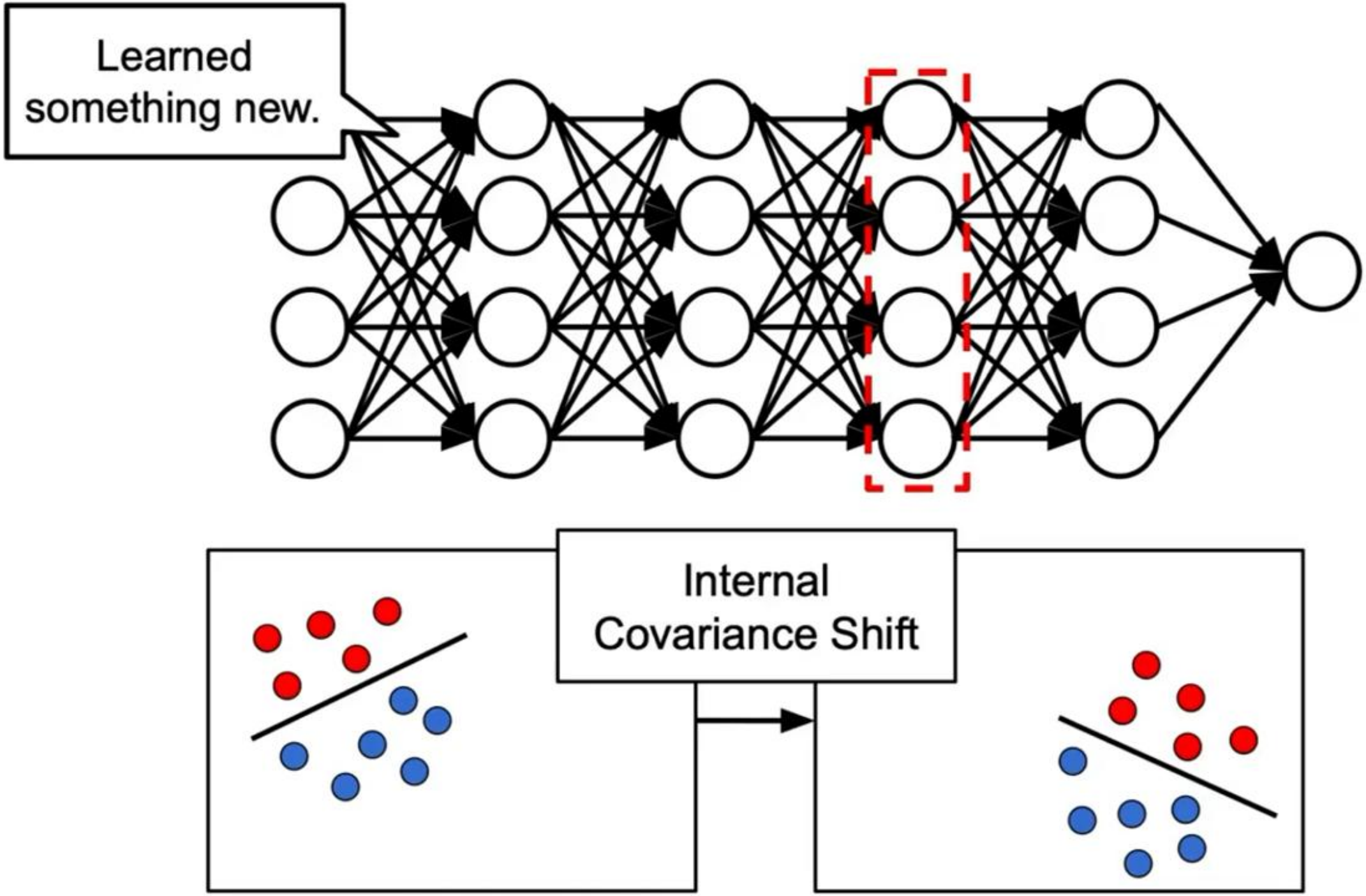


- Training assumes the training data are all similarly distributed
  - Minibatches have similar distribution
- In practice, each minibatch may have a different distribution
  - A “covariate shift”
- Covariate shifts can be large!
  - All covariate shifts can affect training badly

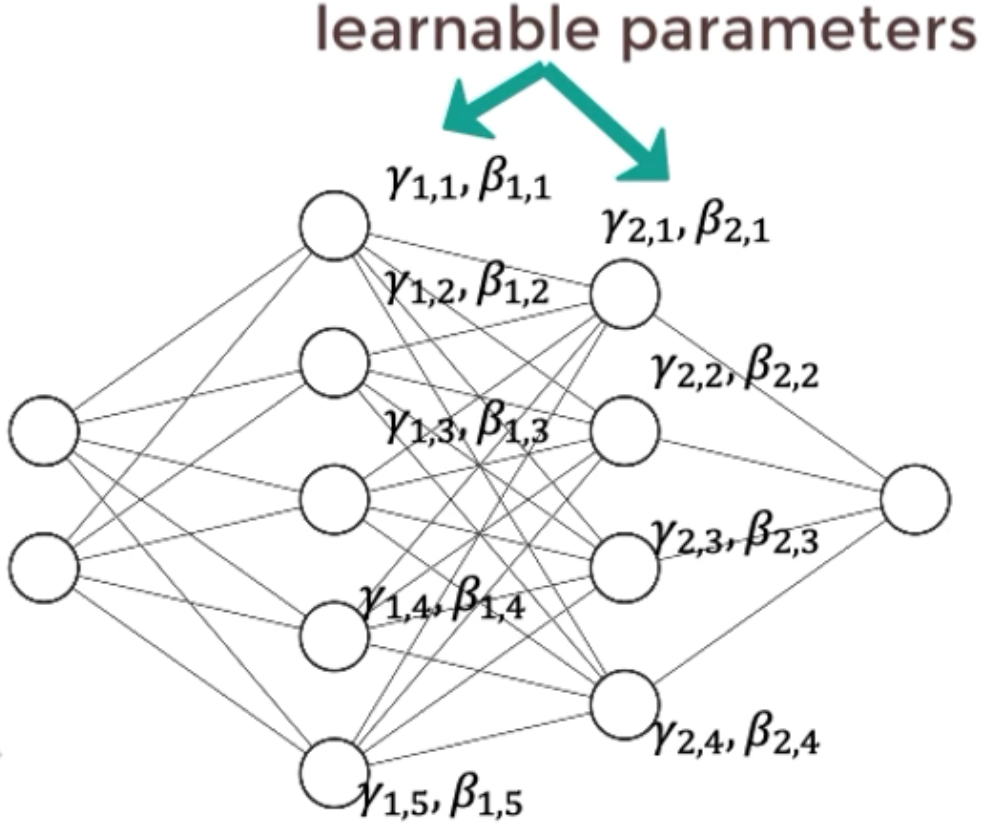
Solution: Move all minibatches to a "standard" location



- "Move" all batches to a "standard" location of the space
  - But where?
  - To determine, we will follow a two-step process


Learned
something new.
Internal
Covariance Shift

Batch Norm Details



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

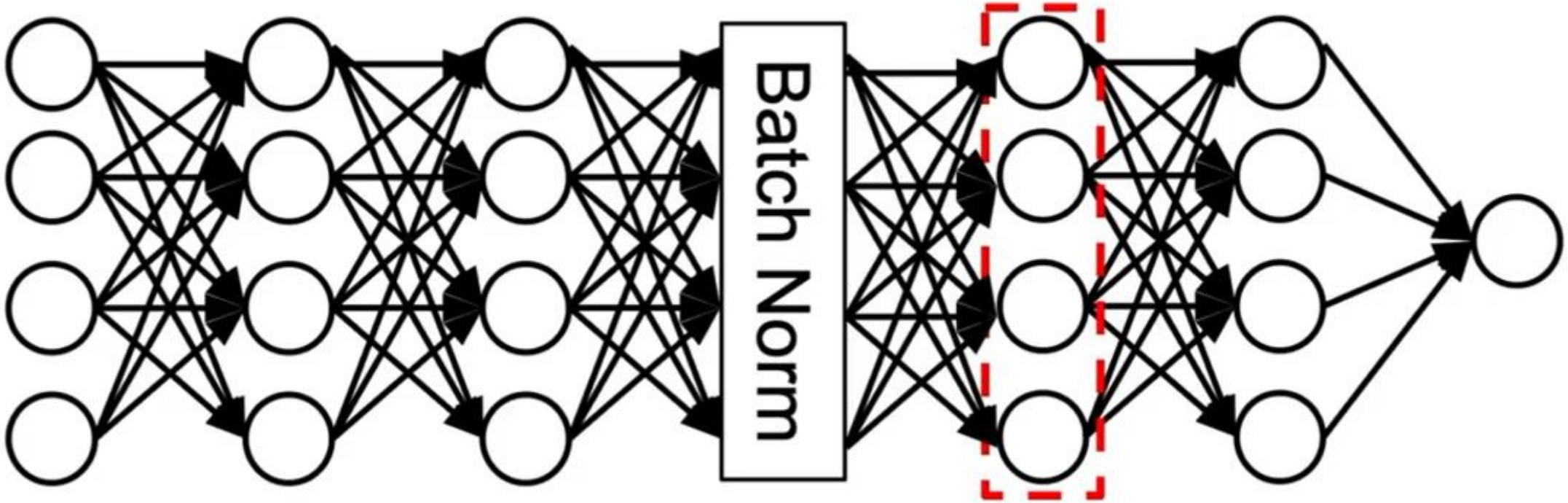
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

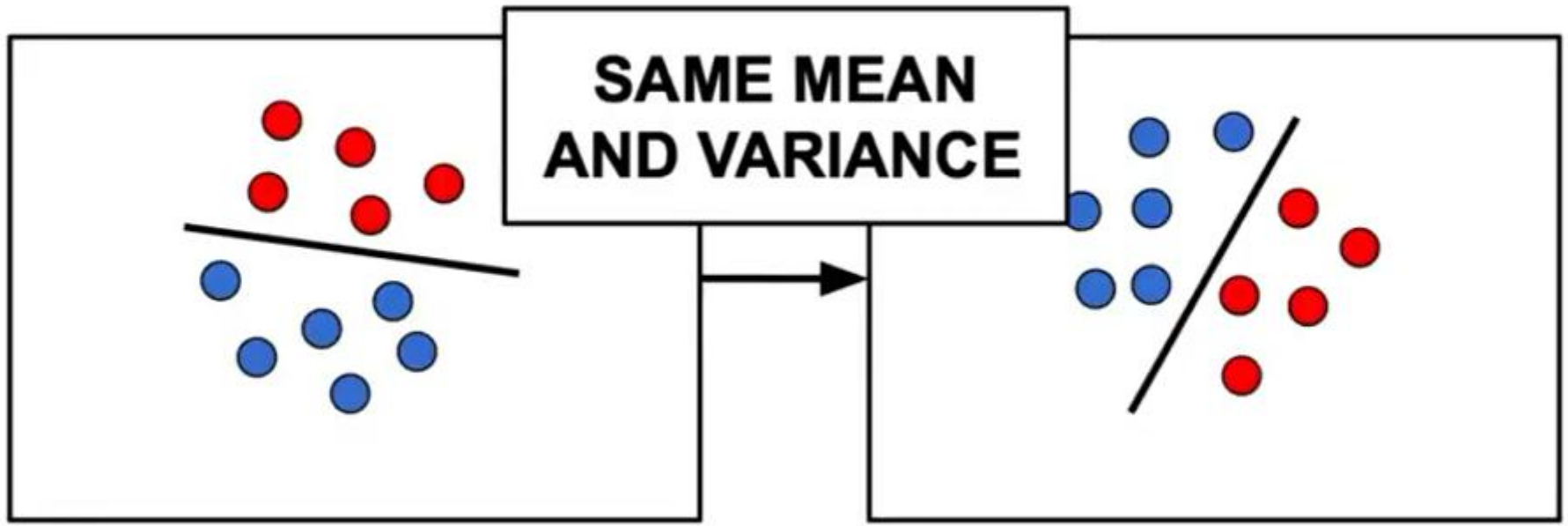
$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

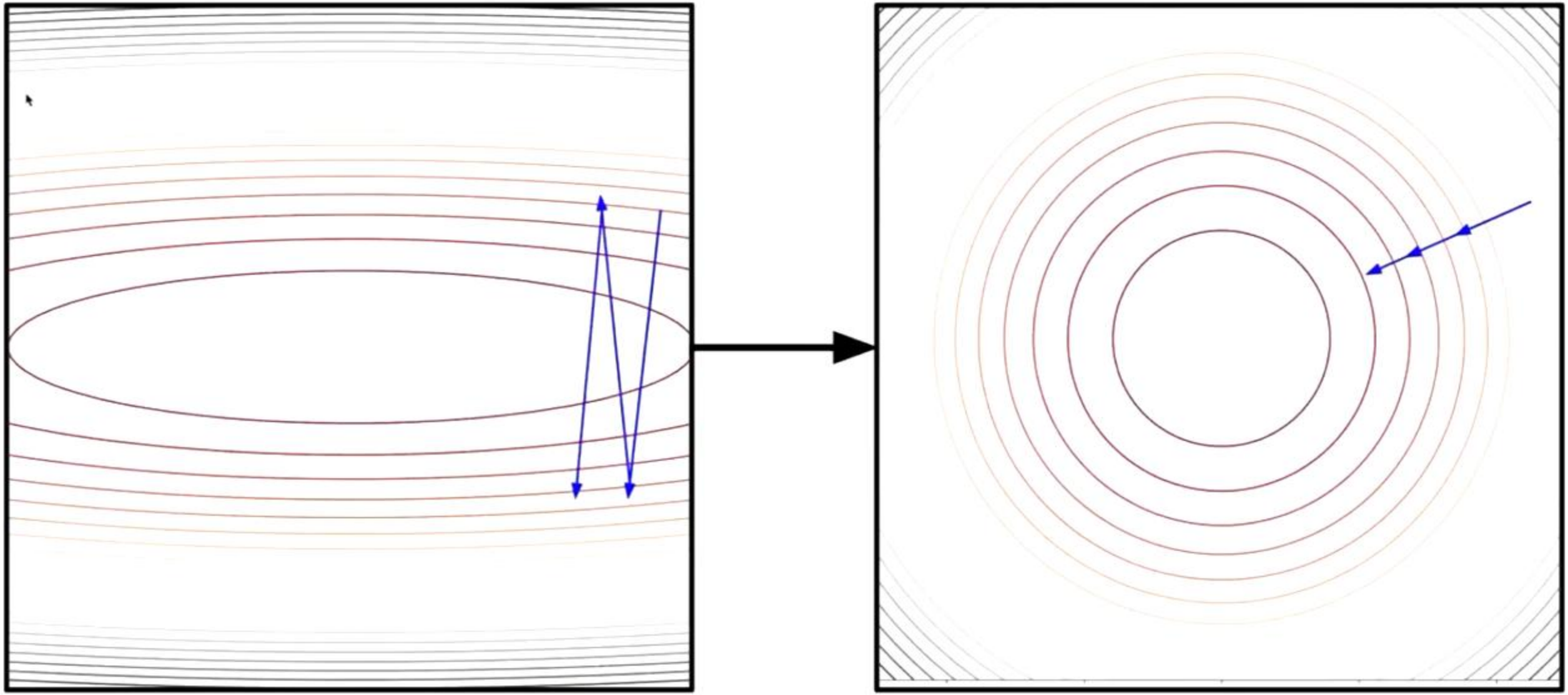
Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

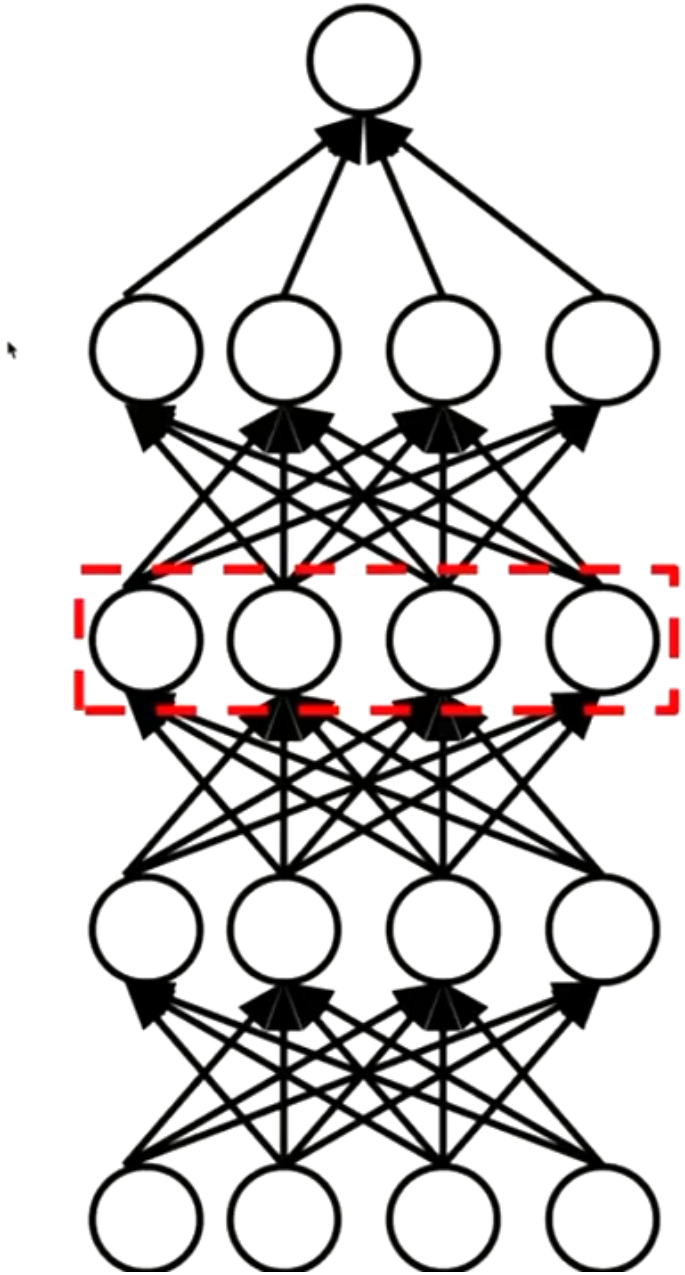

Batch Norm

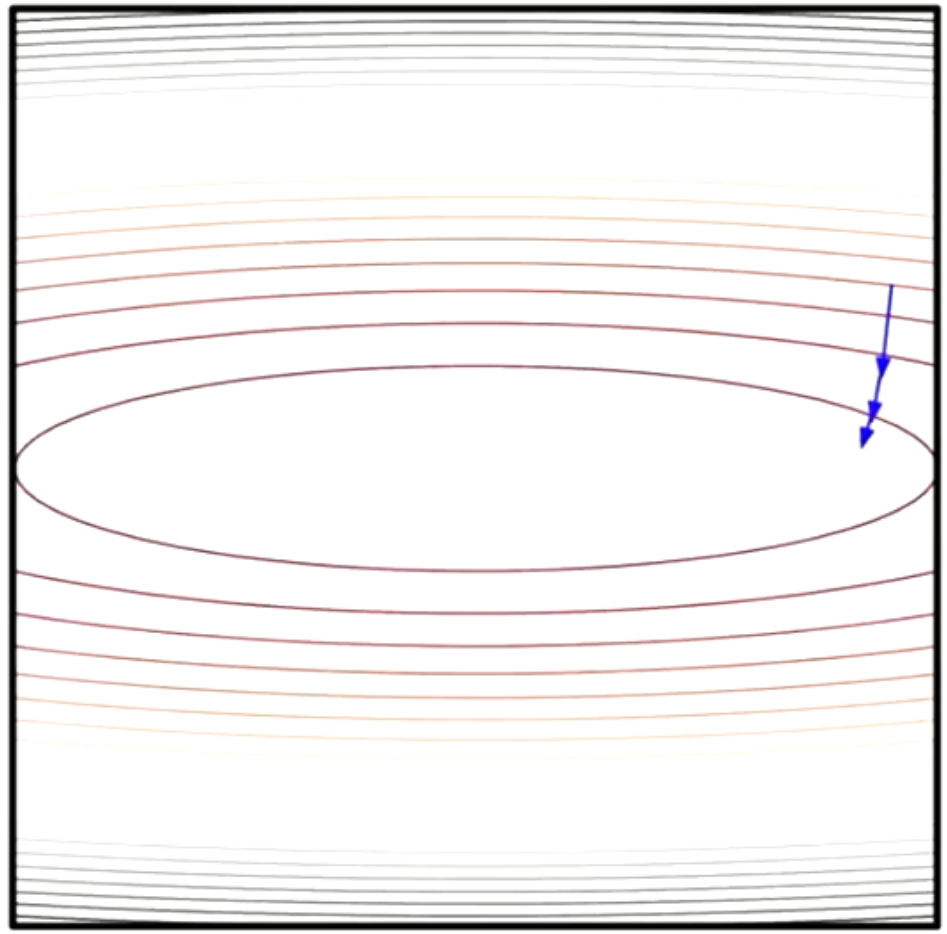

SAME MEAN
AND VARIANCE

Unnormalized

Normalized



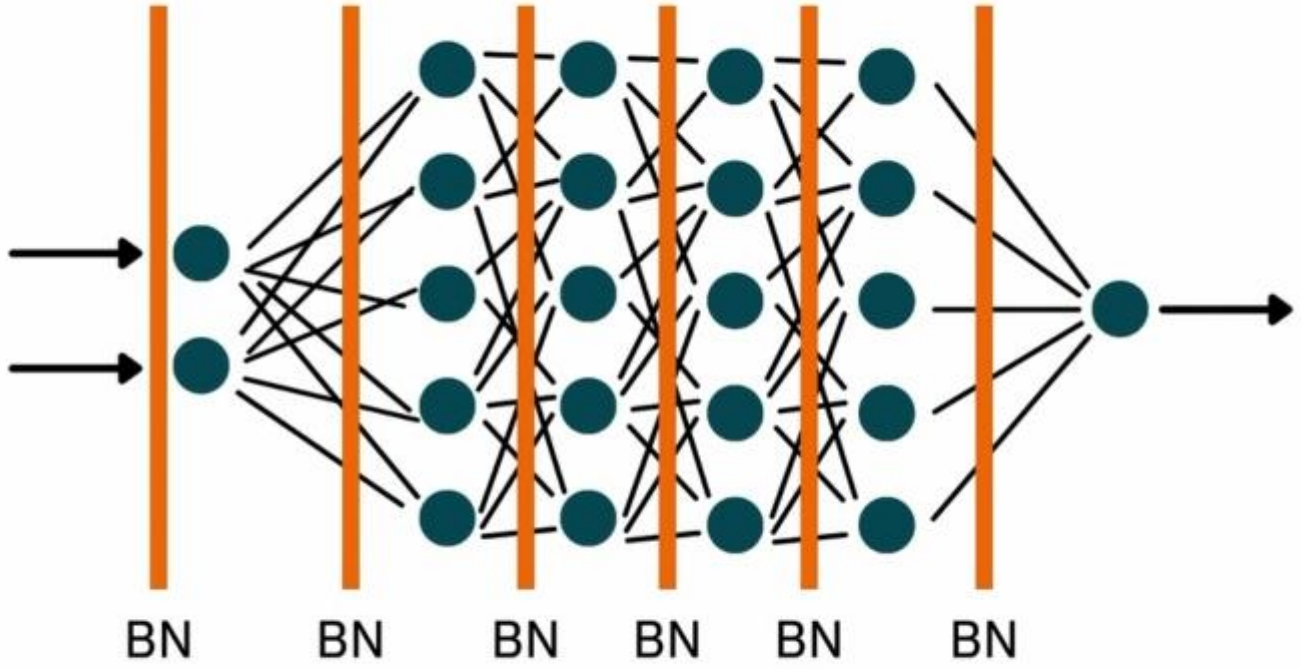




We have to reduce our learning rate to ensure that our model converges.

Batch Normalization

- Center around zero and normalize the inputs
- Achieves same accuracy faster
- Can lead to better performance
- No need to have a standardization layer
- Reduces the need for other regularization
- Epochs take longer due to the amount of computations but convergence will be faster

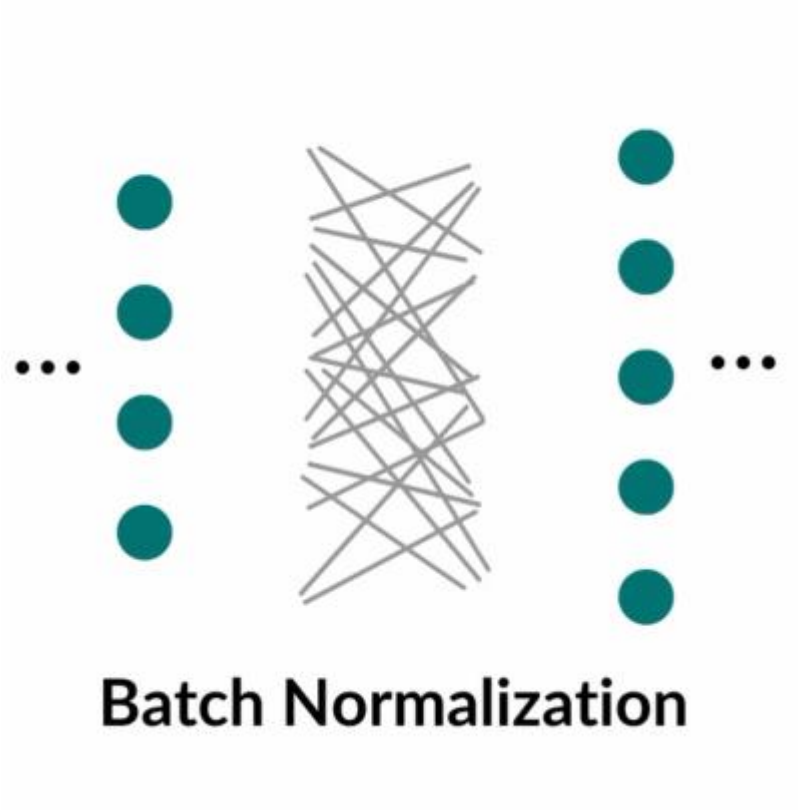


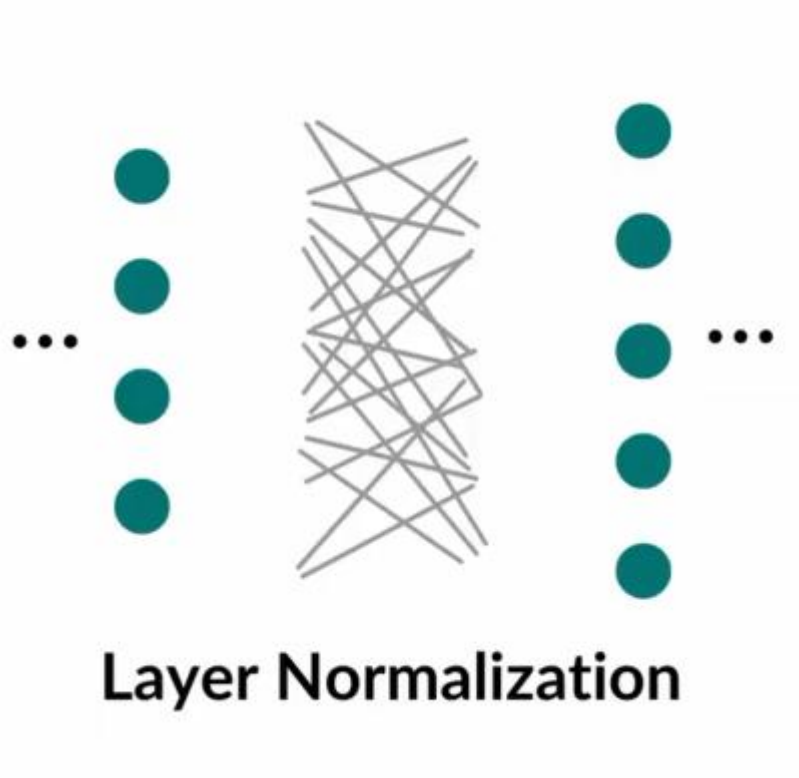
Layer Normalization

Batch Norm has some issues, mostly due the fact that it normalizes across batches

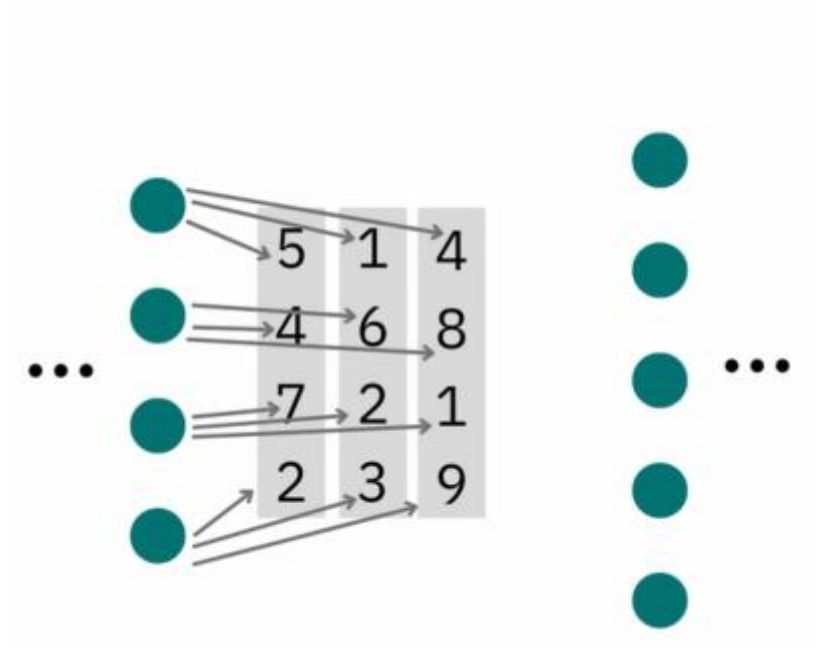
- Difficult to use when small batch sizes and variable sequences are used.
- Hard to parallelize across gpus, if you mini batch is spread across gpus

Batch Norm vs Layer Norm





Batch Norm vs Layer Norm



Batch Norm vs Layer Norm