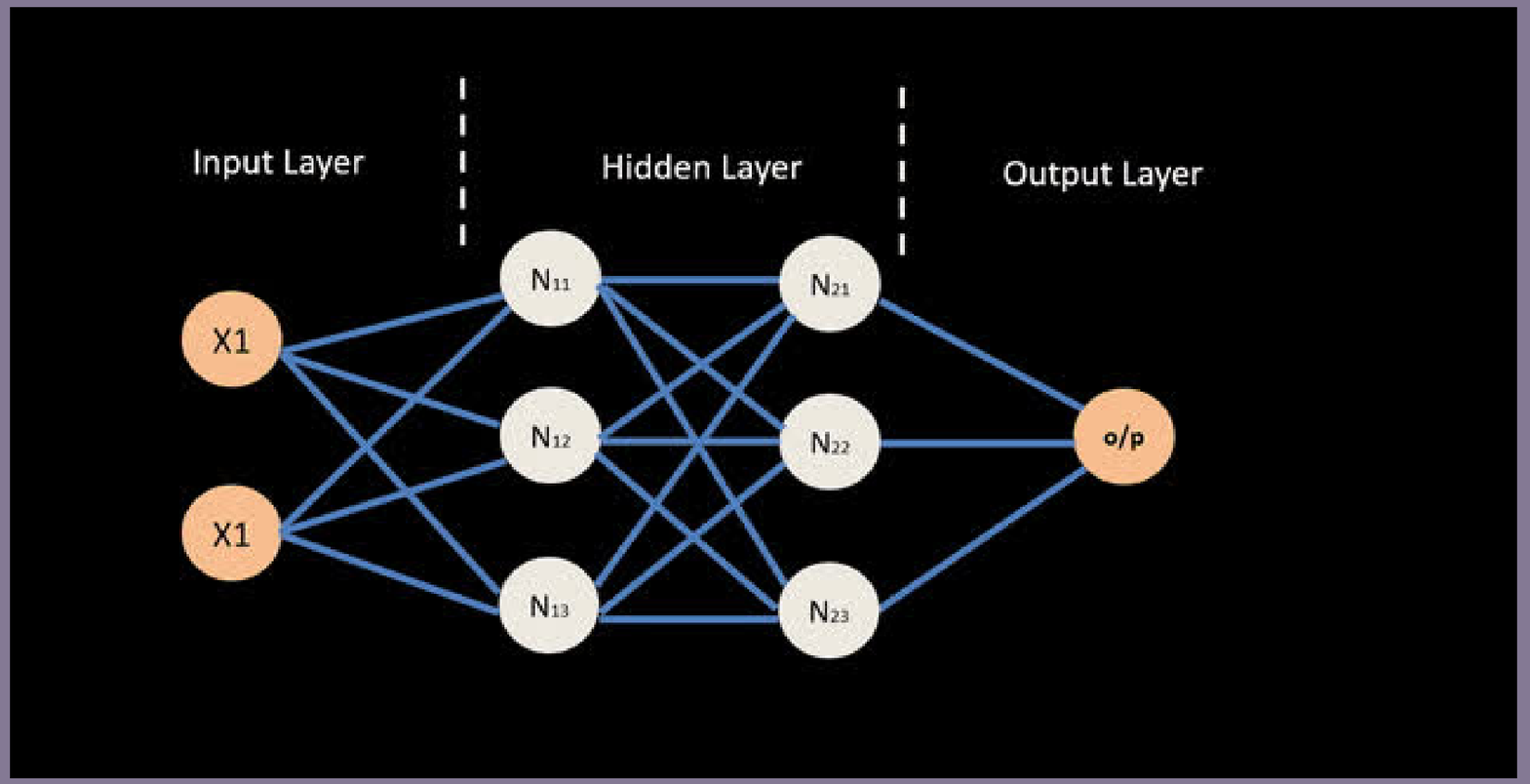

Introduction to
Deep Learning II
CFI SUMMER SCHOOL '21

Agenda

- Recap
- Vanishing and exploding gradients
- Gradient Clipping
- Nonsaturating Activation Functions
- Cost Functions
- Optimizers
- Techniques to improve training

Recap


Input Layer
Hidden Layer
Output Layer
X1
X1
N11
N12
N13
N21
N22
N23
o/p

A question arises....

Why not add as many layers as physically possible?
We can model more complexity and even getter better results!

Vanishing & Exploding Gradient Problem

- Backpropagation involve going from output layer to input layer, propagating error gradients along the way.

- Gradients often get smaller and smaller as the algorithm progresses down to the lower layers, leaving the gradients to almost vanish. This is called Vanishing gradients problem.

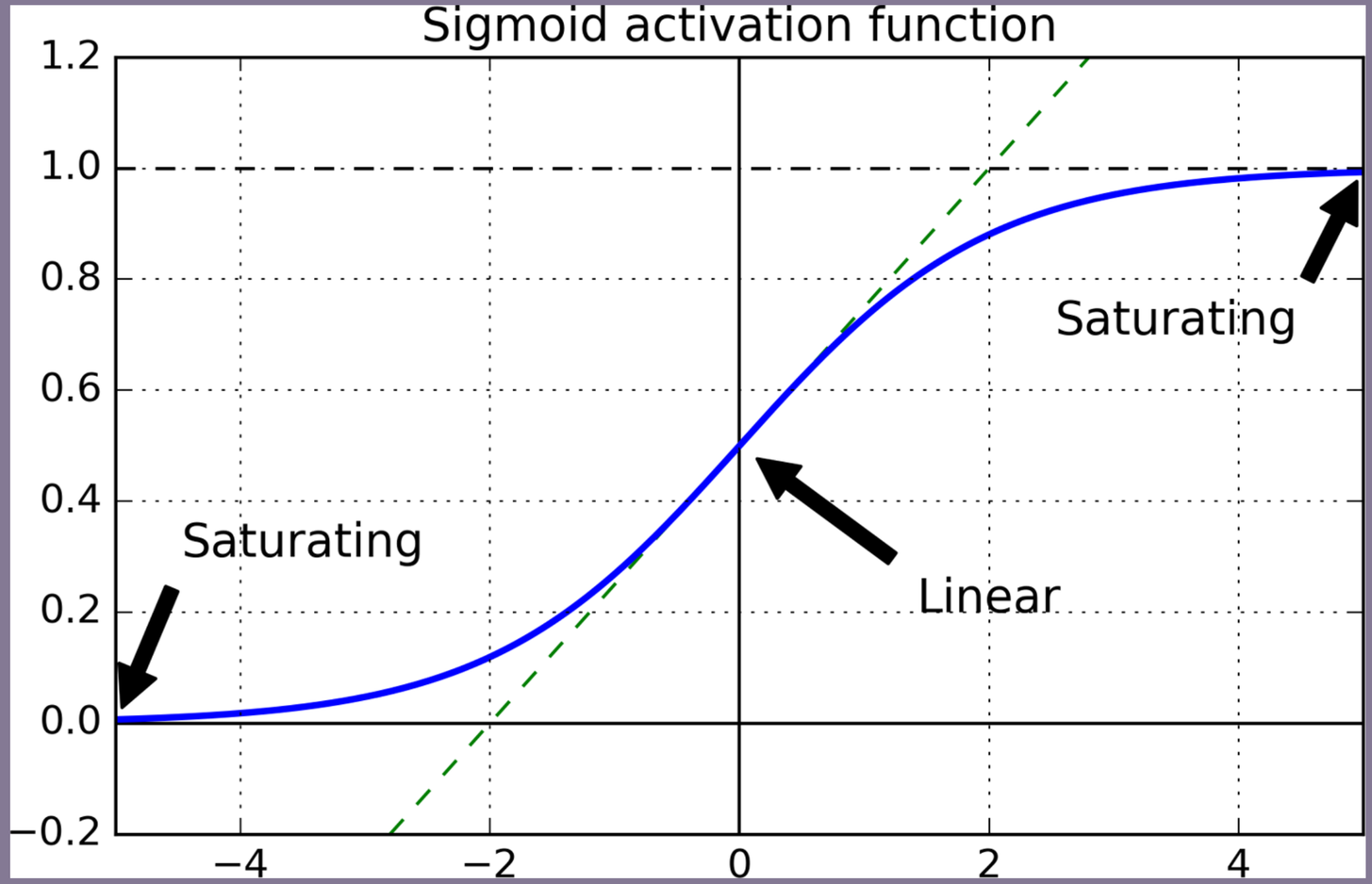
- In some cases, gradients can grow bigger and bigger until layers get insanely large weight updates and the algorithm diverges. This is called Exploding gradients problem.

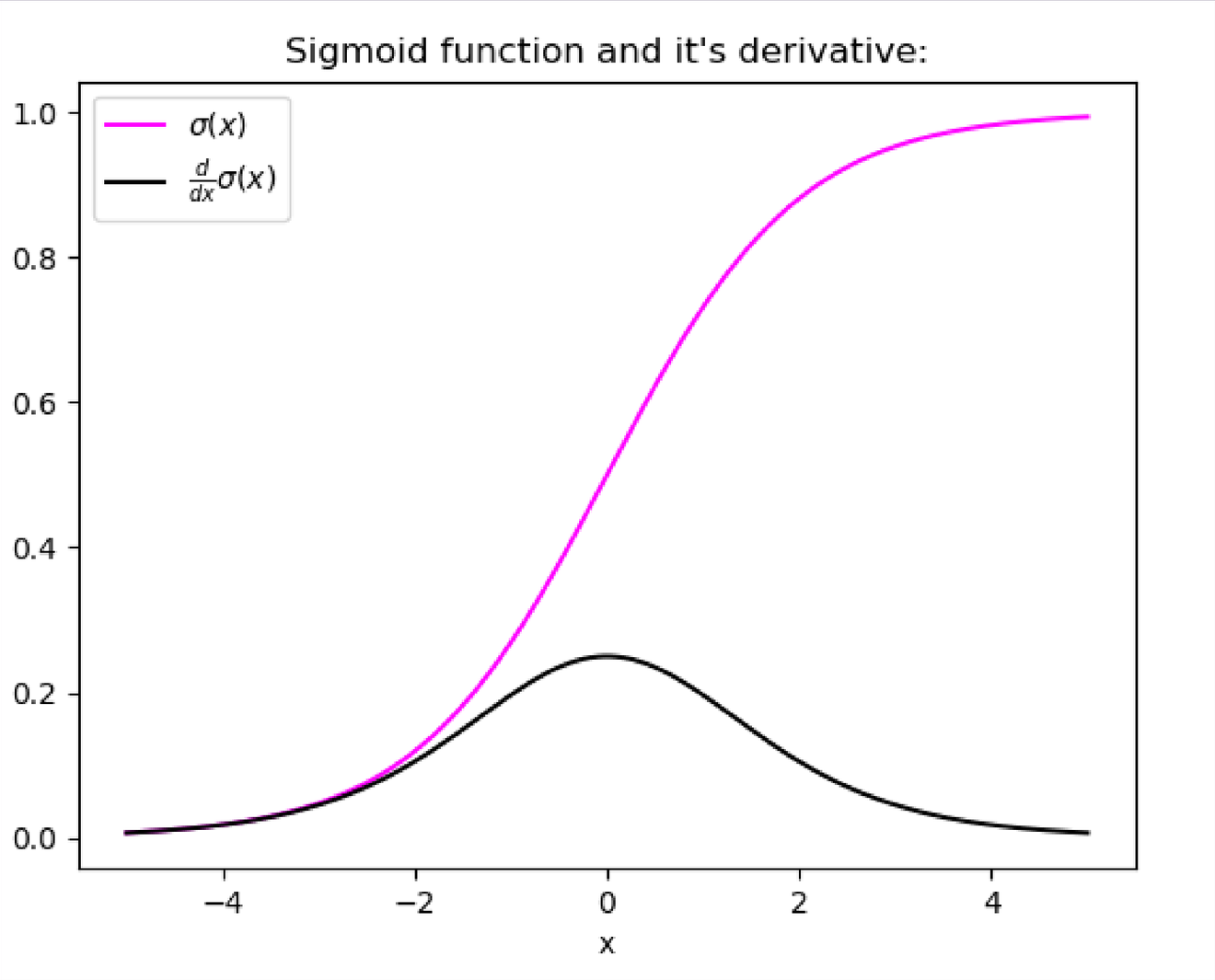
Cause of vanishing and exploding gradients

Few suspects found out against

- Sigmoid activation function (popular at that time)
- Weight Initialisation technique

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

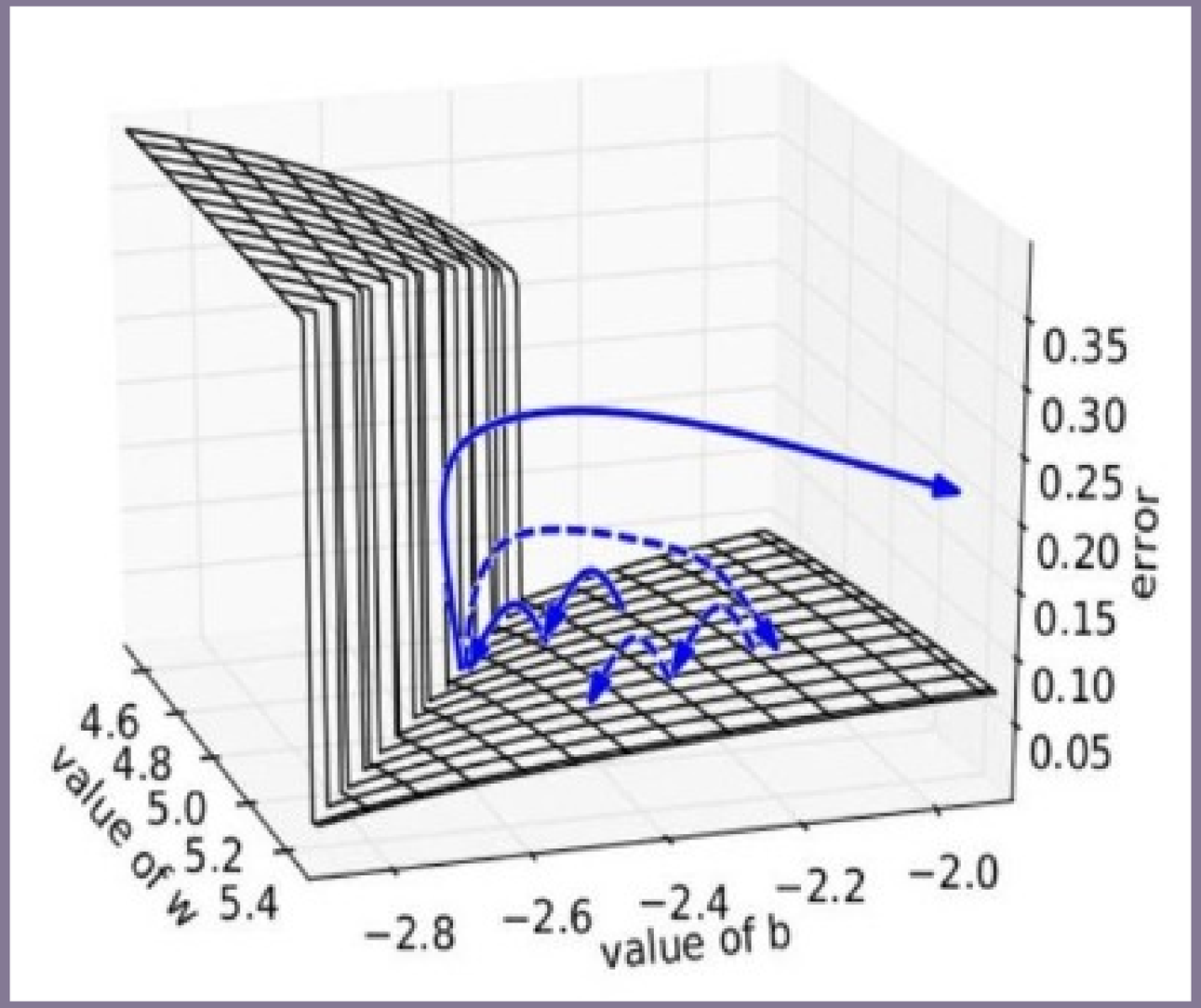




$$\sigma'(x) = \frac{1}{(1+e^{-x})}\left(1 - \frac{1}{(1+e^{-x})}\right)$$

$$\sigma'(x) = \sigma(x).(1 - \sigma(x))$$

Observing the graph of the Sigmoid function, we can see that for larger inputs (negative or positive), it saturates at 0 or 1 with a derivative very close to zero. Thus, when the backpropagation algorithm chips in, it virtually has no gradients to propagate backward in the network, and whatever little residual gradients exist keeps on diluting as the algorithm progresses down through the top layers. So, this leaves nothing for the lower layers.



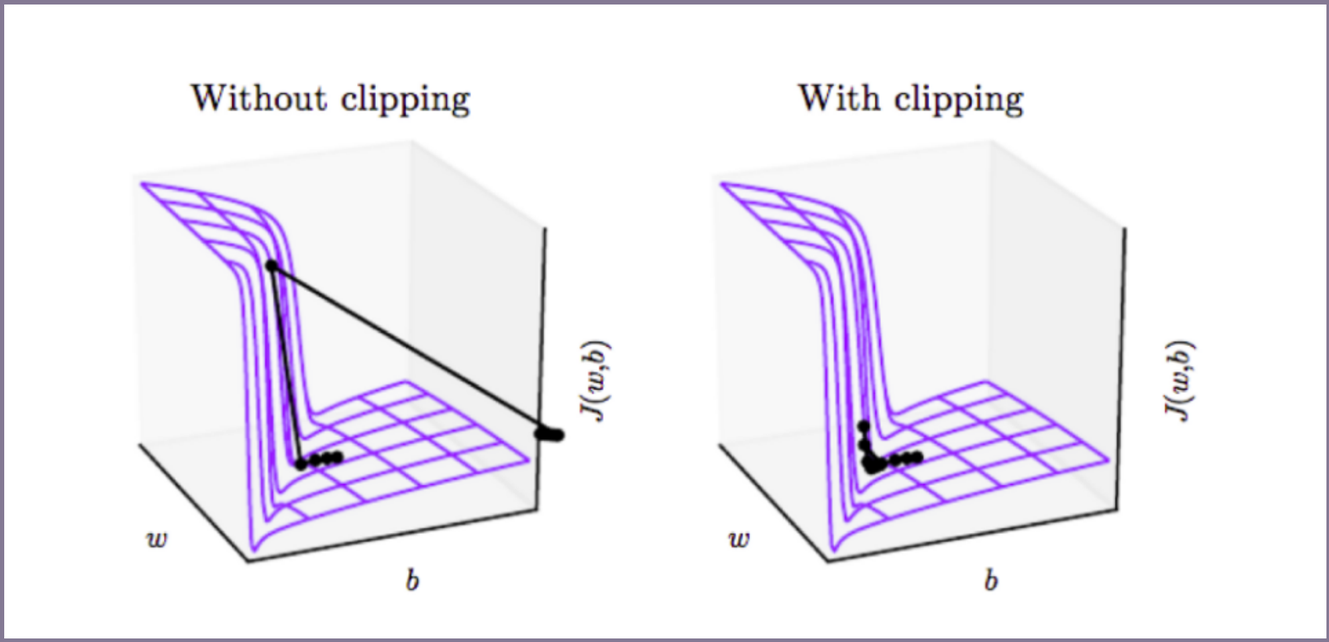
Similarly, in some cases suppose the initial weights assigned to the network generate some large loss. Now the gradients can accumulate during an update and result in very large gradients which eventually results in large updates to the network weights and leads to an unstable network. The parameters can sometimes become so large that they overflow and result in NaN values.

- From what we have seen so far, the deeper we go the higher the chance of the model failing.

- What can we do to increase depth and complexity without increasing the chance of the aforementioned problems?

Gradient Clipping

Another popular technique to mitigate the exploding gradients problem is to clip the gradients during backpropagation so that they never exceed some threshold. This is called Gradient Clipping.



This technique is most often used in recurrent neural networks

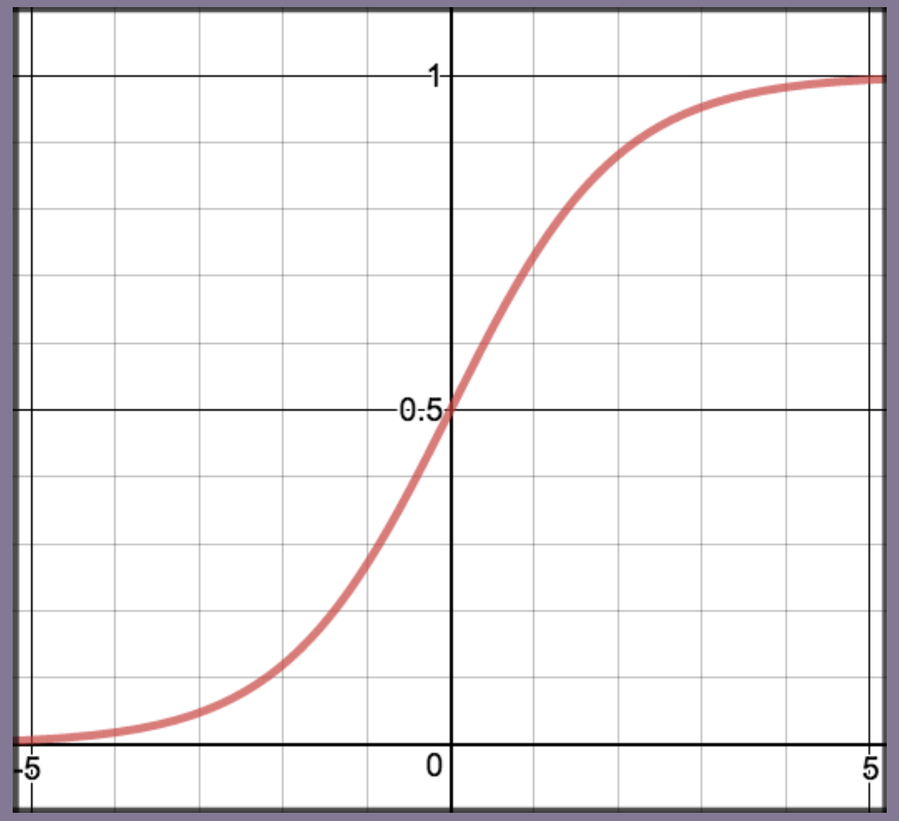
- A major culprit was sigmoid.
- Hence if we find a suitable activation function we could solve the problem of gradient vanishing.

Sigmoid

Other than the vanishing gradient problem, sigmoid had other issues as well.

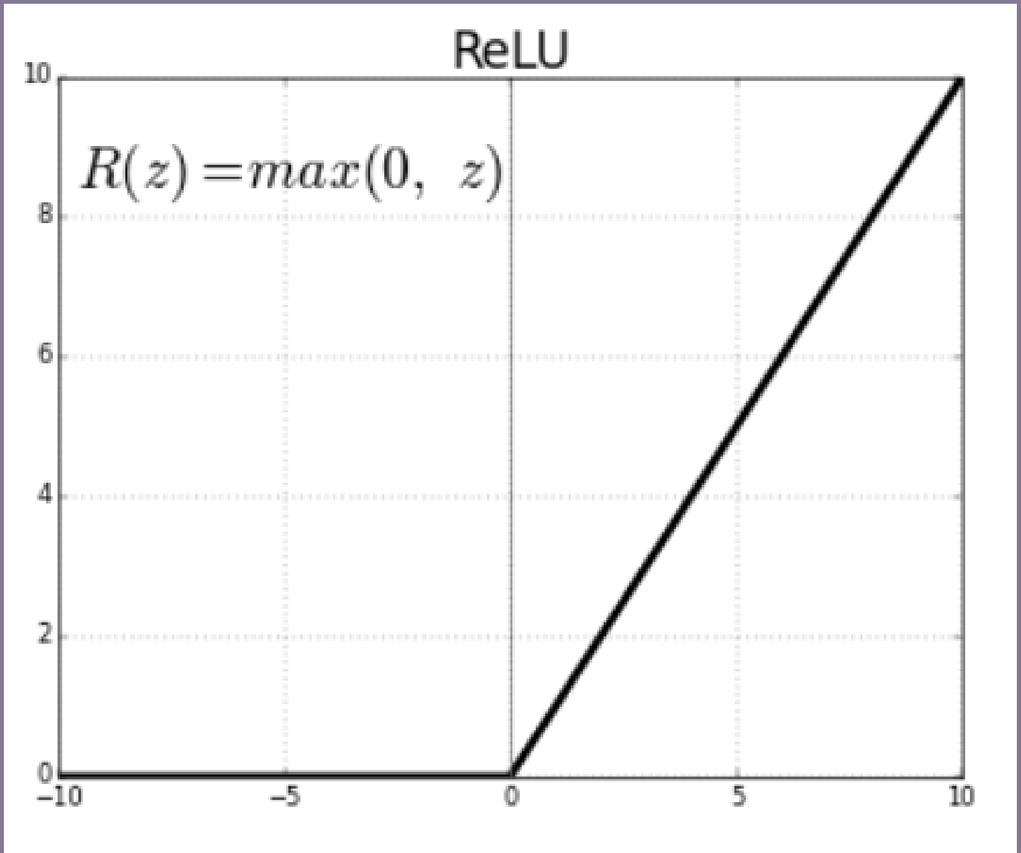
- Note how quickly values saturate in the sigmoid graph. This leads to weaker expressibility.

- It's computationally slower compared to its alternatives when it comes to backpropagation.



Nonsaturating Activation Functions

Rectified Linear Unit (ReLU)



- Despite its name and appearance, it's not linear and provides the same benefits as Sigmoid but with better performance.
- It avoids and rectifies vanishing gradient problem.
- ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations.

Nonsaturating Activation Functions

Disadvantages of ReLU

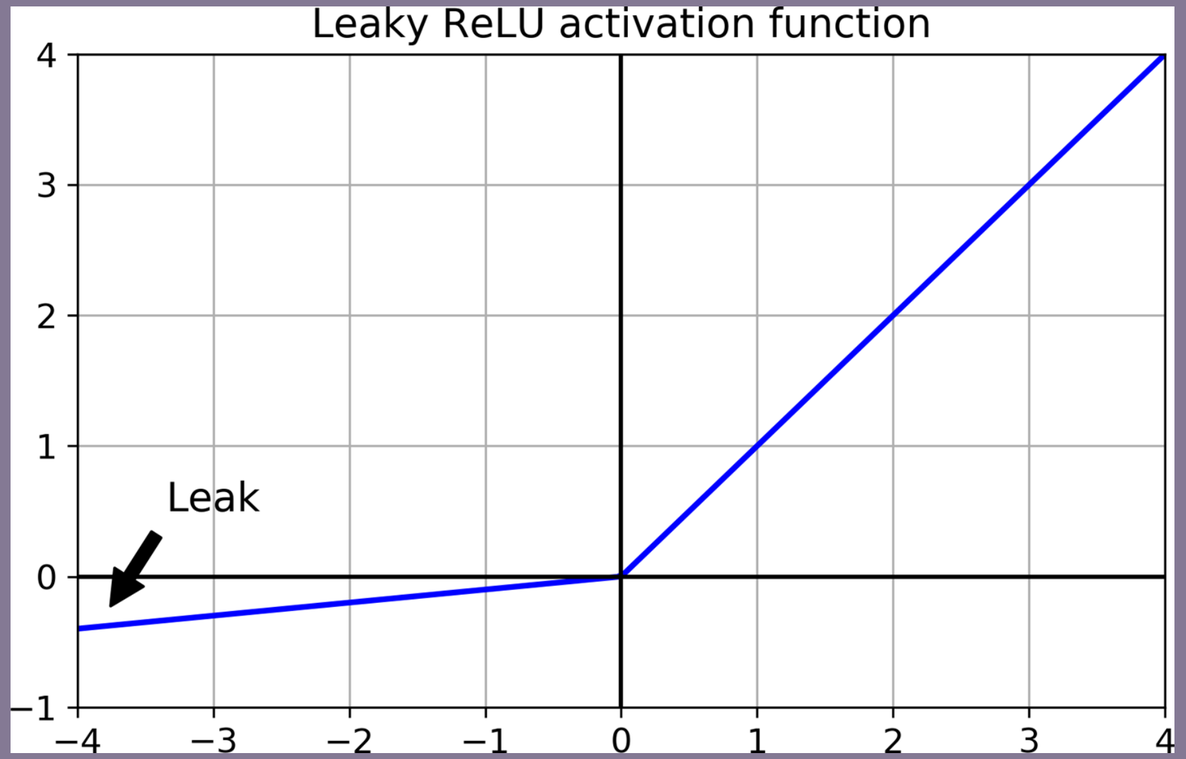
- One of its limitations is that it should only be used within Hidden layers of a Neural Network Model.

- Some gradients can be fragile during training and can die. It can cause a weight update which will make it never activate on any data point again. Simply saying that ReLu could result in Dead Neurons.

Nonsaturating Activation Functions

Leaky-ReLU

- LeakyReLU α (z) = max(αz, z)
- The hyperparameter α defines how much the function “leaks”: it is the slope of the function for z < 0 and is typically set to 0.01.
- This small slope ensures that leaky ReLUs never die;



Nonsaturating Activation Functions

Randomised Leaky Relu

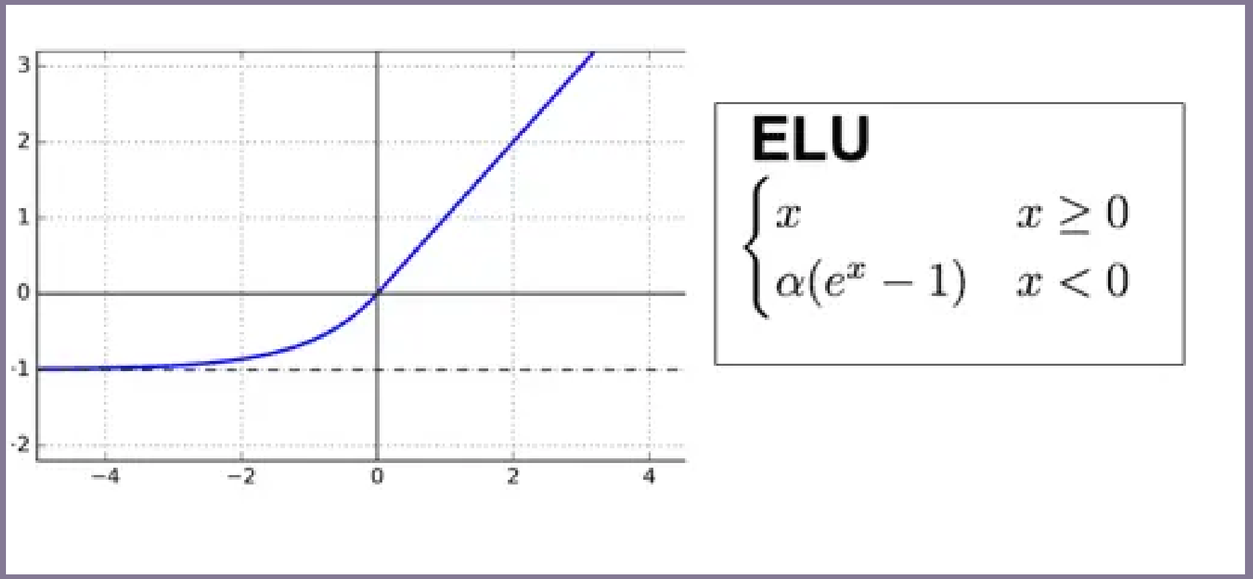
α is picked randomly in a given range during training and is fixed to an average value during testing.

Parametric Leaky Relu

α is authorized to be learned during training (instead of being a hyperparameter, it becomes a parameter that can be modified by backpropagation like any other parameter)

Nonsaturating Activation Functions

Exponential linear unit



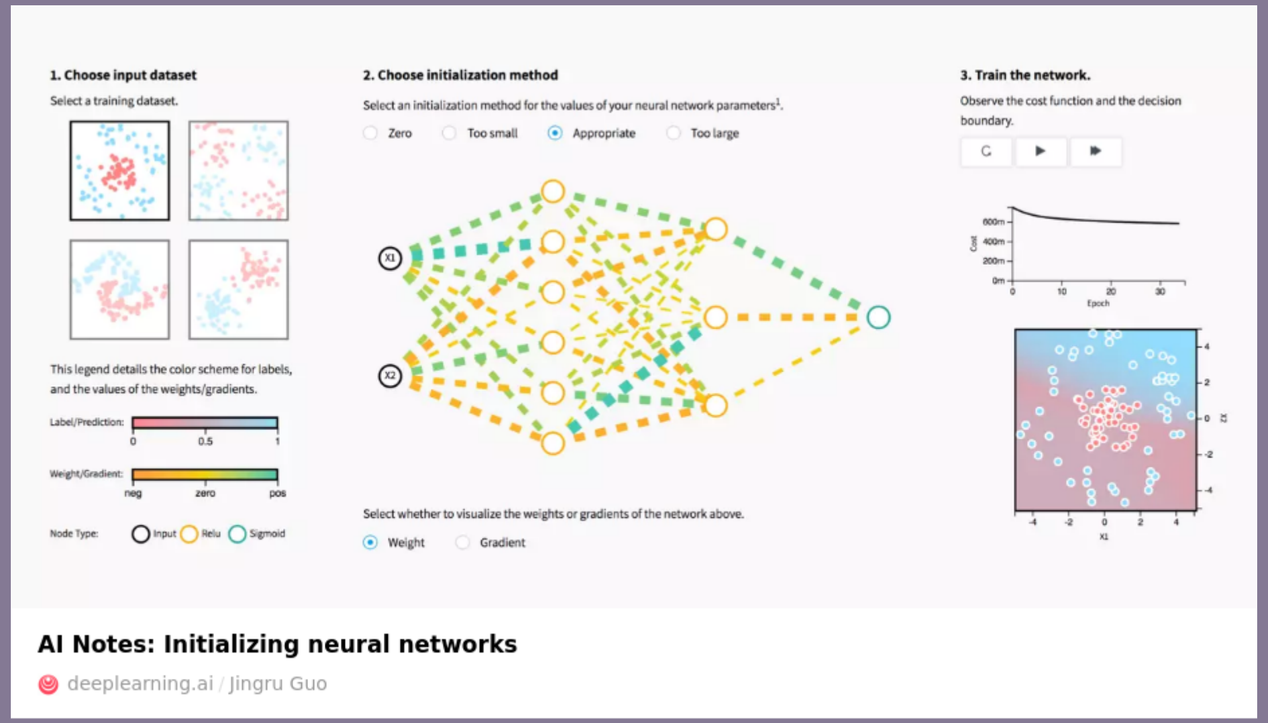
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

- For z < 0, it takes on negative values which allow the unit to have an average output closer to 0 thus alleviating the vanishing gradient problem
- For z < 0, the gradients are non zero. This avoids the dead neurons problem.
- For α = 1, the function is smooth everywhere, this speeds up the gradient descent since it does not bounce right and left around z=0.
- A scaled version of this function (SELU: Scaled ELU) is also used very often in Deep Learning.

Other than playing around with the activation functions we can also try various methods of initializing the parameters, i.e, the weights.

Why initialisation matters



https://www.deeplearning.ai/ai-notes/initialization/#I

How to find a good Initialisation paramters

- The mean should be 0
- The variance should be the same across all layers

Popular Initialization methods

- Xavier / Glorot initialization
- He initialization

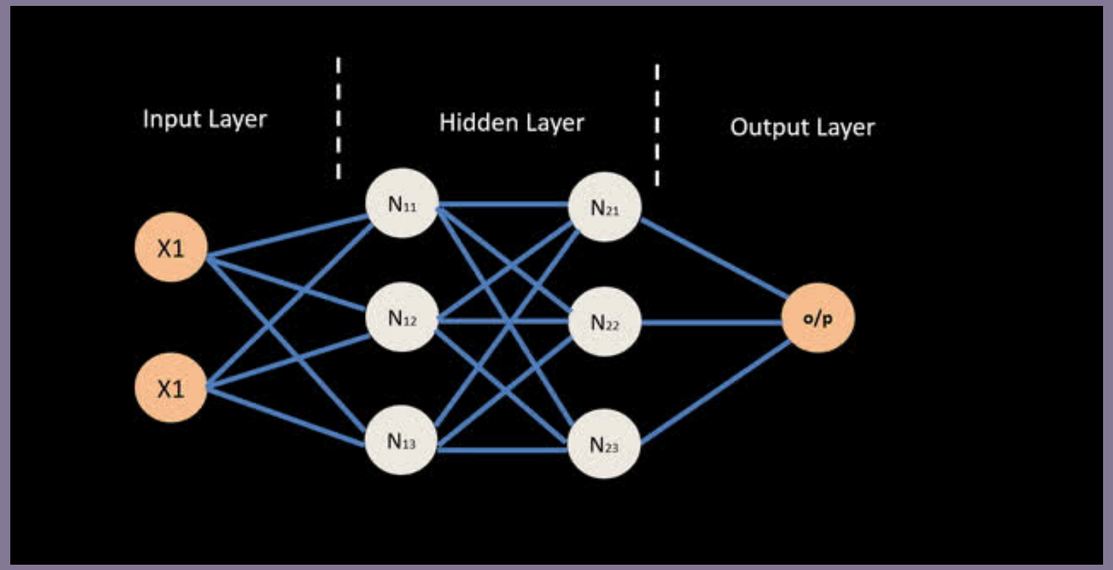
These methods only differ in the variance used per layer, but the idea is essentially the same in both.

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, Tanh, Logistic, Softmax	$1 / fan_{avg}$
He	ReLU & variants	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

The gist is that if we control the forward propagation. The backpropagation will follow suit.

We will see another method of achieving the same later.

Cost functions

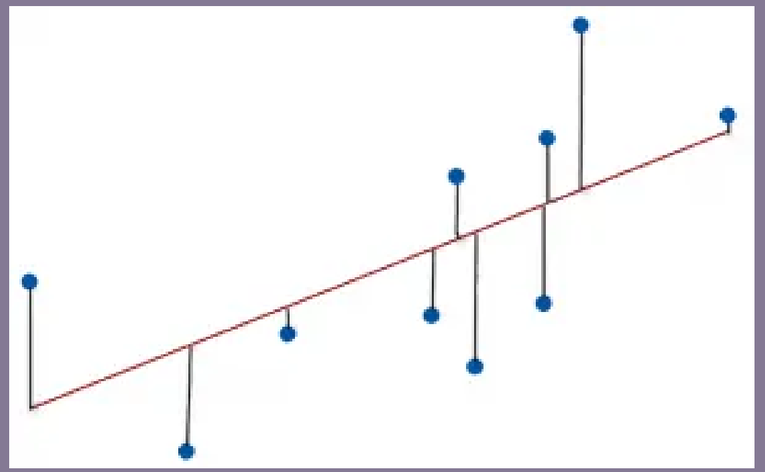


Commonly Used cost functions:
Mean Absolute Error (MAE)
Mean Squared Error (MSE)
Cross-Entropy

Cost functions for Regression Problems

- In regression, the model predicts an output value for each training data during the training phase. The cost functions for regression are calculated on distance-based error. Let us first understand this concept first.
- Let us say that for a given set of input data, the actual output was y and our regression model predicts y’ then the error in prediction is calculated simply as

  Error = y-y’
- This also known as distance-based error and it forms the basis of cost functions that are used in regression models. Below animation gives a more clear geometrical interpretation of distance-based error.
- Various distance-based error functions include:
  - Mean Error (ME)
  - Mean Absolute Error (MAE)
  - Mean Square Error (MSE)



Mean Error (ME)

- In this cost function, the error for each training data is calculated and then the mean value of all these errors is derived. Calculating mean of the errors is the simplest and most intuitive way possible. But there is a catch here.
- The errors can be both negative or positive and during summation, they will tend to cancel each other out. In fact, it is theoretically possible that the errors are such that positive and negatives cancel each other to give zero error mean error for the model.
- Mean Error is not a recommended cost function for regression. But it does lay the foundation for our next cost functions.

Y (Actual)	Y' (Predicted)	Error = Y-Y'
10.2	9.4	0.8
3.5	1.7	1.8
7.1	6.9	0.2
14.5	15.4	-0.9
17.2	18.4	-1.2
9.5	11.3	-1.8
2.7	2.5	0.2
11.5	11.1	0.4
5.9	6.7	-0.8
15.3	15.2	0.1
Sum		-1.2
Mean Error		(-1.2/10) = -0.12

Mean Absolute Error (MAE)

- This addresses the shortcoming of ME. Here an absolute difference between the actual and predicted value is calculated to avoid any possibility of negative error.
- MAE loss is also called L1 loss.

$$MAE = \frac{\sum_{i=1}^{n} | y_i - \hat{y}_i |}{n}$$

Y (Actual)	Y' (Predicted)	Error = |Y-Y'|
10.2	9.4	0.8
3.5	1.7	1.8
7.1	6.9	0.2
14.5	15.4	0.9
17.2	18.4	1.2
9.5	11.3	1.8
2.7	2.5	0.2
11.5	11.1	0.4
5.9	6.7	0.8
15.3	15.2	0.1
Sum		8.2
Mean Absolute Error		(8.2/10) = 0.82

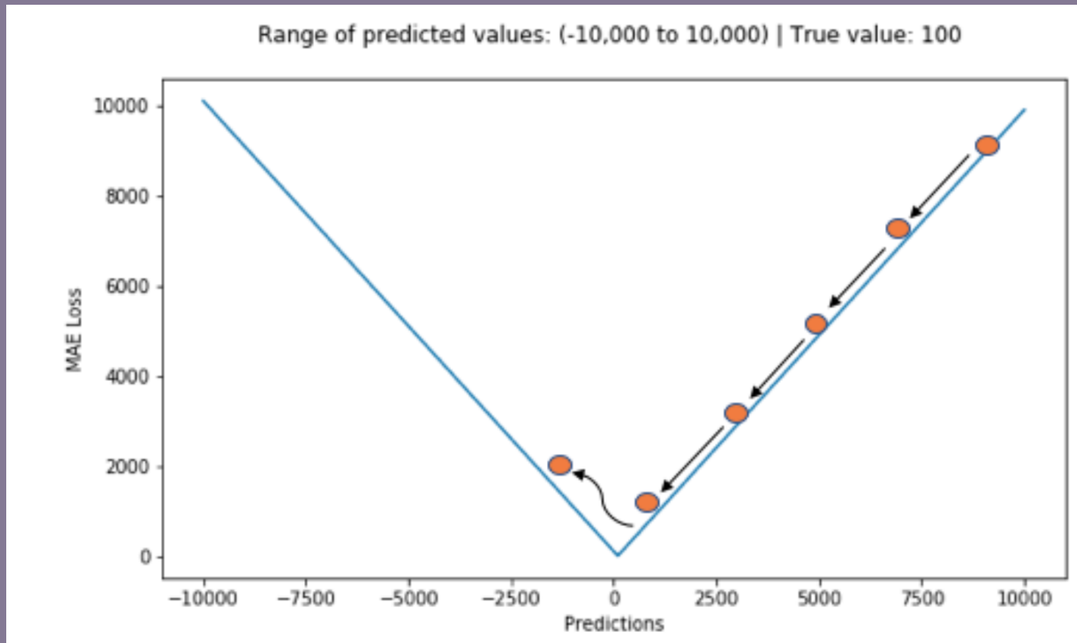
Mean Squared Error (MSE)

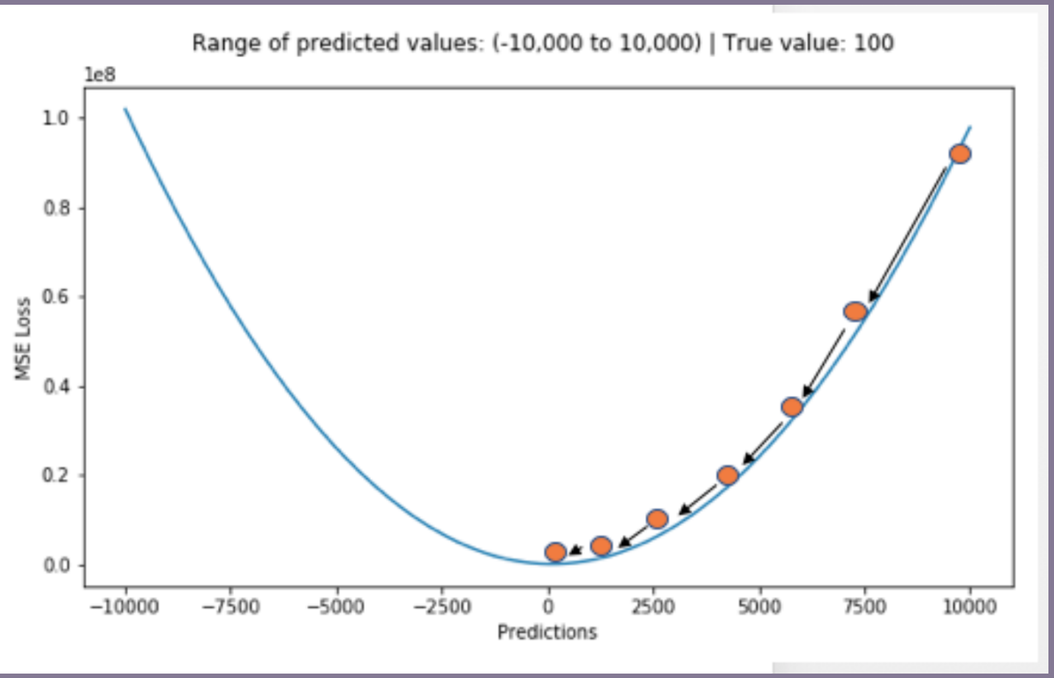
- Here the square of the difference between the actual and predicted value is calculated to avoid any possibility of negative error.
- MSE Loss is also called L2 loss.

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

Y (Actual)	Y' (Predicted)	Error = $(Y-Y')^2$
10.2	9.4	0.64
3.5	1.7	3.24
7.1	6.9	0.04
14.5	15.4	0.81
17.2	18.4	1.44
9.5	11.3	3.24
2.7	2.5	0.04
11.5	11.1	0.16
5.9	6.7	0.64
15.3	15.2	0.01
Sum		10.26
Mean Square Error		(10.26/10) = 1.02

MAE vs MSE


Range of predicted values: (-10,000 to 10,000) | True value: 100
MAE Loss
10000
8000
6000
4000
2000
0
−10000
−7500
−5000
−2500
0
2500
5000
7500
10000
Predictions


Range of predicted values: (-10,000 to 10,000) | True value: 100
1e8
MSE Loss
1.0
0.8
0.6
0.4
0.2
0.0
−10000
−7500
−5000
−2500
0
2500
5000
7500
10000
Predictions

Cost functions for Classification Problems

Cost functions used in classification problems are different than what we saw in the regression problem above. There is a reason why we don’t use regression cost functions for classification problem and we will see it later. But before that let us see the classification cost functions.

There are definitely some variety of loss functions for classification problems also but we will cover one of the most used one "**Cross-entropy**"

Cross-entropy

Cross-entropy is a measure of the difference between two probability distributions for a given random variable or set of events.

Let us understand this with a small example:
Consider that we have a classification problem of 3 classes as

Orange, Apple and Tomato.

The machine learning model will actually give a probability distribution of these 3 classes as output for a given input data. The class having the highest probability is considered as a winner class for prediction.

Orange = [1,0,0]
Apple = [0,1,0]
Tomato = [0,0,1]

Cross-entropy

During the training phase, for example, if the training data is Orange, the predicted probability distribution should tend towards the actual probability distribution of Orange. If predicted probability distribution is not closer to the actual one, the model has to adjust its weight.
This is where cross entropy becomes a tool to calculate how much far is the predicted probability distribution from the actual one. This intuition of cross entropy is shown in the animation in next slide.

Cross Entropy - Intuition

Categorical Cross Entropy Cost Function

This cost function is used in classification problems where there are multiple classes and input data belongs to only one class, i.e, multiclass classification.
Consider an example to understand this. Let's assume we have a classification task containing *M* classes.

Our model gives the probability distribution as below for M classes for a particular input data D.

$$P(D) = [y1' , y2' , y3' \dots yM']$$

And the actual or target probability distribution of the data D is

$$A(D) = [y1 , y2 , y3 \dots yM]$$

Then cross entropy for that particular data D is calculated as

$$CrossEntropy(A,P) = – (y1*\log(y1') + y2*\log(y2') + y3*\log(y3') + \dots + yM*\log(yM'))$$

Categorical Cross Entropy Cost Function

The above formula just measures the cross entropy for a single observation or input data.

The error in classification for the complete model is given by categorical cross entropy which is nothing but the mean of cross entropy for all N training data.
Categorical_Cross_Entropy = (Sum of Cross Entropy for N data)/N

Binary Cross Entropy Cost Function

Binary cross entropy is a special case of categorical cross entropy when there is only one output which just assumes a binary value of 0 or 1 to denote negative and positive class respectively

Let us assume that actual output is denoted by a single variable y, then cross entropy for a particular data D is can be simplified as follows –

cross_entropy(D) = – y*log(y') when y = 1

cross_entropy(D) = – (1-y)*log(1-y') when y = 0

The error in binary classification for the complete model is given by binary cross entropy which is nothing but the mean of cross entropy for all N training data.

Binary_Cross_Entropy = (Sum of Cross_Entropy for N data)/N

Why Cross Entropy and Not MAE/MSE in Classification?

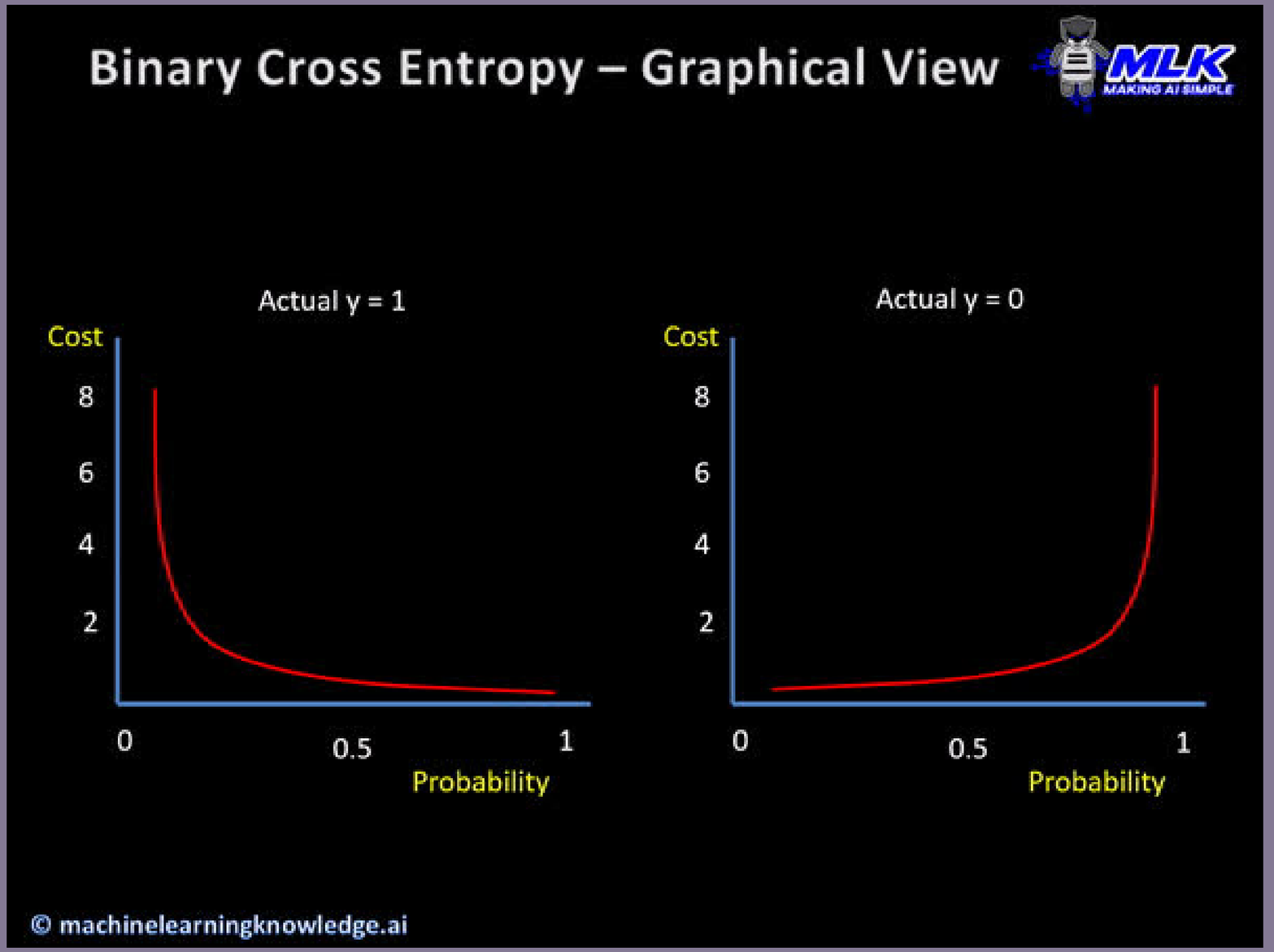
Sometimes machine learning model, especially during the training phase not only makes a wrong classification but makes it with so confidence that they deserve much more penalization.

Such models deserve severe penalization during the training phase. For simplicity purpose let us see how binary Cross Entropy, MAE and MSE penalize in such a situation.

In the below example, the two scenarios of y=1, y'=0.2 and y=0, y'=0.8 are an example of confidently wrong classification. As we can see Binary Cross Entropy is doing a more severe penalty than MAE or MSE for this situation.

Scenario	Actual y	Predicted y'	MAE	MSE	Binary Cross Entropy
Prediction is confidently closer to actual class 1	1	0.9	$\|1 - 0.9\| = 0.1$	$(1 - 0.9)^2 = .01$	-1*LOG(0.9) = 0.1
Prediction is confidently closer to wrong class 0	1	0.2	$\|1 - 0.2\| = 0.8$	$(1 - 0.2)^2 = .64$	-1*LOG(0.2) = 1.64
Prediction is confidently closer to actual class 0	0	0.1	$\|0 - 0.1\| = 0.1$	$(0 - 0.1)^2 = .01$	-(1-0)*LOG(1-0.1) =0.1
Prediction is confidently closer to wrong class 1	0	0.8	$\|0 - 0.8\| = 0.8$	$(0 - 0.9)^2 = .64$	-(1-0)*LOG(1-0.8) = 1.64

Binary Cross Entropy – Graphical View



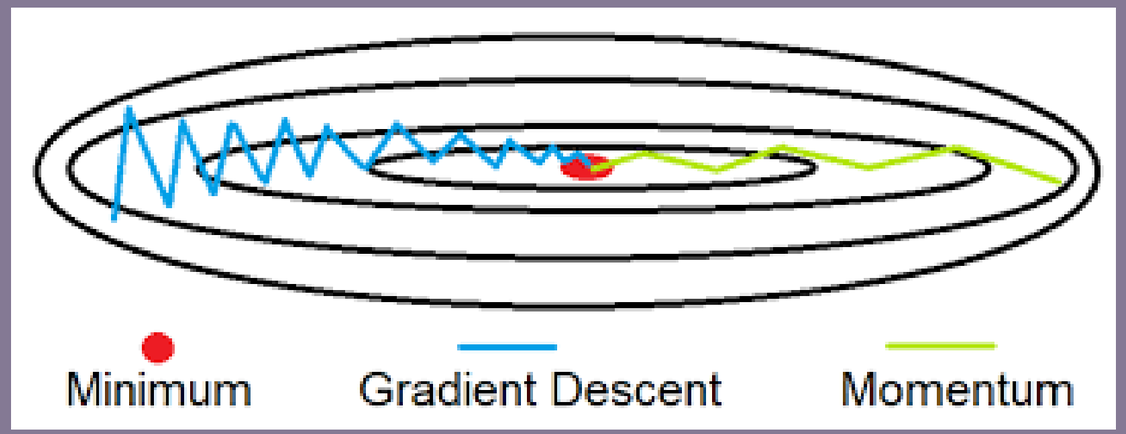
Optimizers

Optimizers, as the name says, guide the model to reach their most optimum state, i.e, when the loss is least.

A huge speed boost comes from using a faster optimizer than the regular Gradient Descent optimizer. In this section, we will present the most popular algorithms: momentum optimization, Nesterov Accelerated Gradient, AdaGrad, RMSProp, and finally Adam and Nadam optimization.

Momentum Optimizers

$$m \leftarrow \beta m - \eta(\nabla_\theta J(\theta))$$
$$\theta \leftarrow \theta + m$$

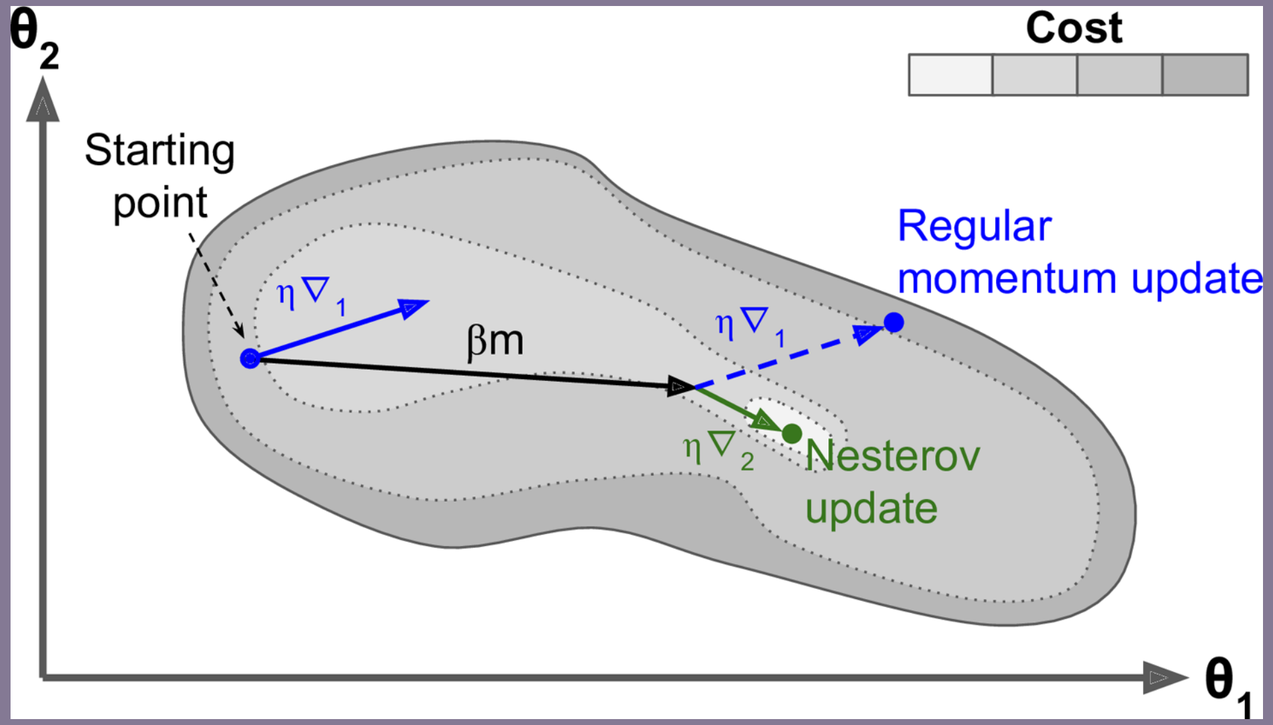


Recall that Gradient Descent updates the weights θ by directly subtracting the gradient of the cost function $J(\theta)$ with regard to the weights ($\nabla_\theta J(\theta)$) multiplied by the learning rate η. The equation is: $\theta \leftarrow \theta - \eta \nabla_\theta J(\theta)$. It does not care about what the earlier gradients were. If the local gradient is tiny, it goes very slowly.

Momentum optimization cares a great deal about what previous gradients were: at each iteration, it subtracts the local gradient from the momentum vector m (multiplied by the learning rate η), and it updates the weights by adding this momentum vector

Nesterov Accelerated Gradient

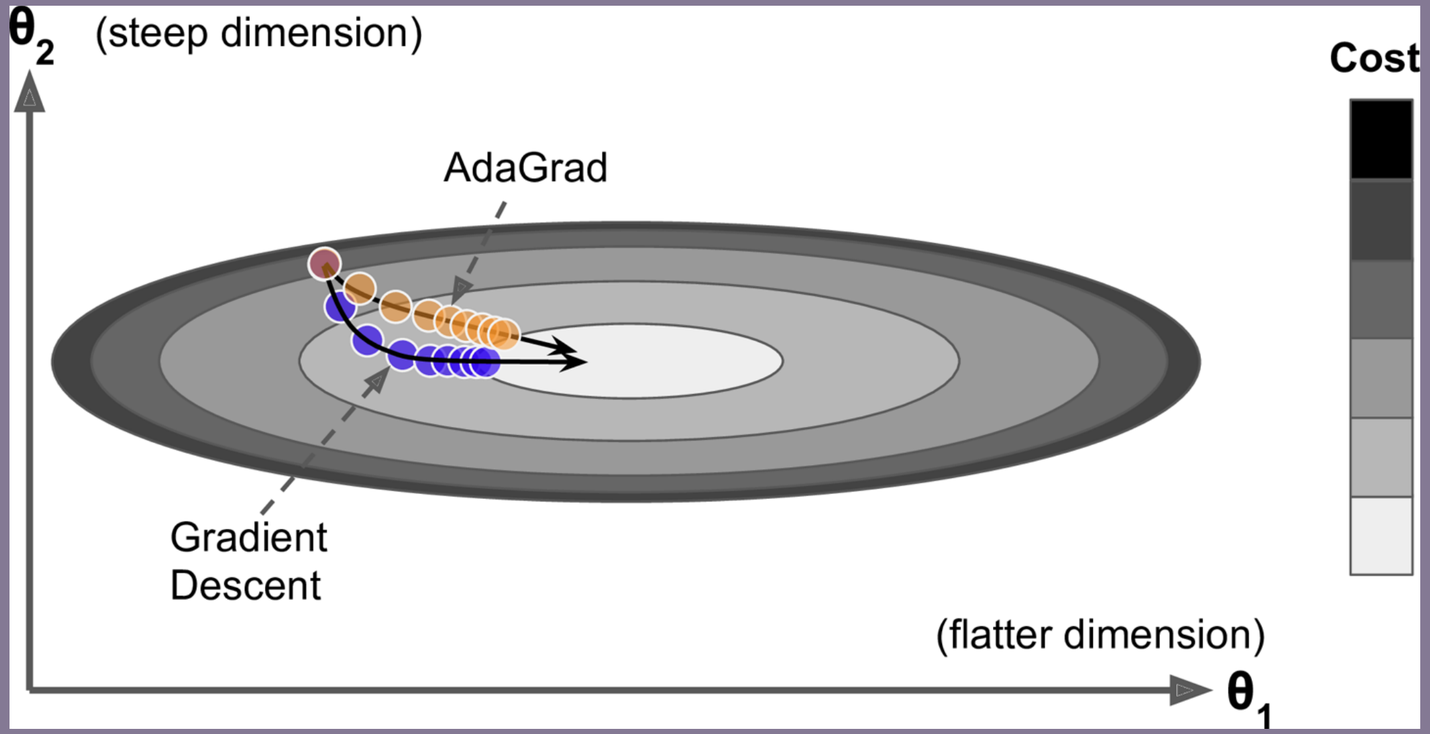
$$m \leftarrow \beta m - \eta \nabla_\theta (J(\theta) + \beta m))$$
$$\theta \leftarrow \theta + m$$



The Nesterov Accelerated Gradient (NAG) method, also known as Nesterov momentum optimization, measures the gradient of the cost function not at the local position θ but slightly ahead in the direction of the momentum, at θ + βm

AdaGrad

$$s \leftarrow s + \nabla_\theta(J(\theta)) \otimes \nabla_\theta(J(\theta))$$
$$\theta \leftarrow \theta - \eta \nabla_\theta(J(\theta)) \oslash \text{sqrt}(s + \varepsilon)$$



In short, this algorithm decays the learning rate, but it does so faster for steep dimensions than for dimensions with gentler slopes. This is called an adaptive learning rate.
It helps point the resulting updates more directly toward the global optimum.
Additionally, it requires much less tuning of learning rate.

AdaGrad

Disadvantages of AdaGrad:

The learning rate gets scaled down so much
that the algorithm ends up stopping entirely before reaching
the global optimum.

To solve this problem we use RMSProp

RMSProp

$$s \leftarrow \beta s + (1 - \beta) \nabla_\theta(J(\theta)) \otimes \nabla_\theta(J(\theta))$$
$$\theta \leftarrow \theta - \eta \nabla_\theta(J(\theta)) \oslash \text{sqrt}(s + \varepsilon)$$

The RMSProp algorithm fixes the problem of early stopping of AdaGrad by accumulating
only the gradients from the most recent iterations (as opposed to all the gradients since the beginning of training)

The decay rate β is typically set to 0.9. Yes, it is once again a new hyperparameter, but this default value often works well, so you may not need to tune it at all.

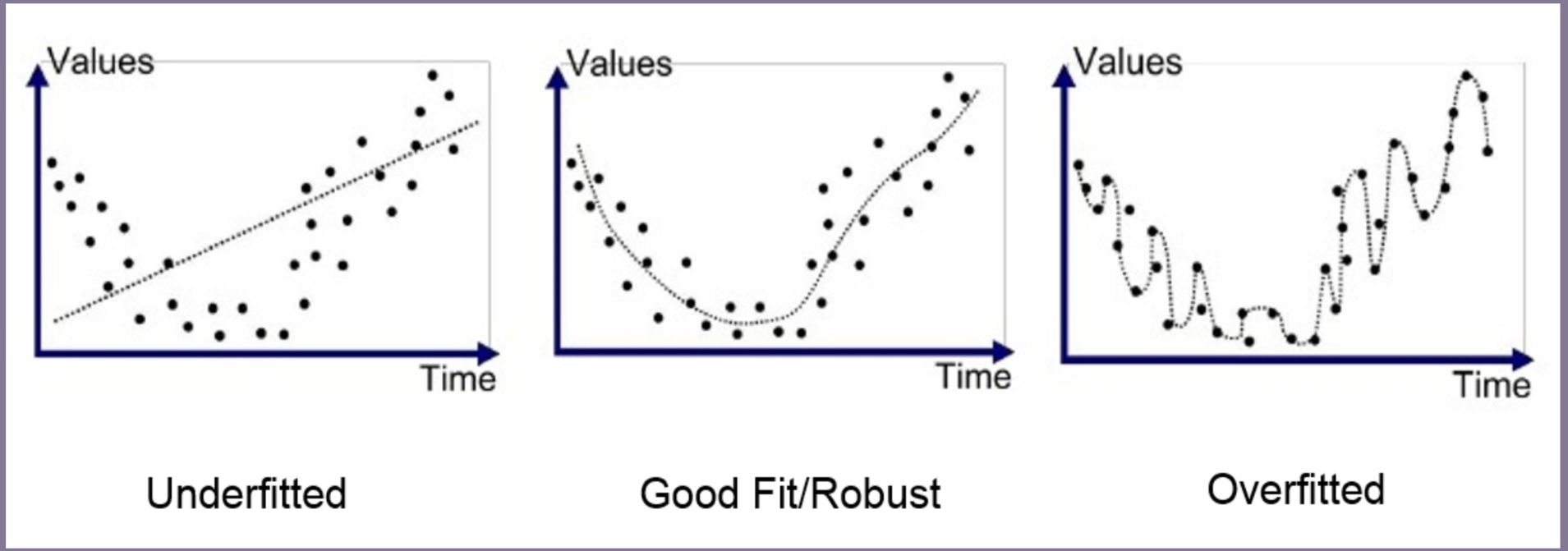
Adam and Nadam Optimization

$$m \leftarrow \beta_1 m - (1 - \beta_1)\nabla_\theta J(\theta)$$
$$s \leftarrow \beta_2 s + (1 - \beta_2)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$$
$$\hat{m} \leftarrow \frac{m}{1 - \beta_1}$$
$$\hat{s} \leftarrow \frac{s}{1 - \beta_2}$$
$$\theta \leftarrow \theta + \eta \hat{m} \oslash \sqrt{\hat{s} + \varepsilon}$$

Adam, which stands for adaptive moment estimation, combines the ideas of momentum optimization and RMSProp: just like momentum optimization, it keeps track of an exponentially decaying average of past gradients; and just like RMSProp, it keeps track of an exponentially decaying average of past squared gradients

A few tricks and techniques to improve training

Generalization



Regularizers

There are several ways of controlling the capacity of Neural Networks to prevent overfitting:

- L2 Regularisation
- L2 Regularisation
- Drop-out

A regression model that uses L1 regularization technique is called Lasso Regression and model which uses L2 is called Ridge Regression.

Note:

- It is important to scale the data before applying any of the regularisation as it is sensitive to the scale of the input features.
- The regularisation term should only be added to the cost function during training.

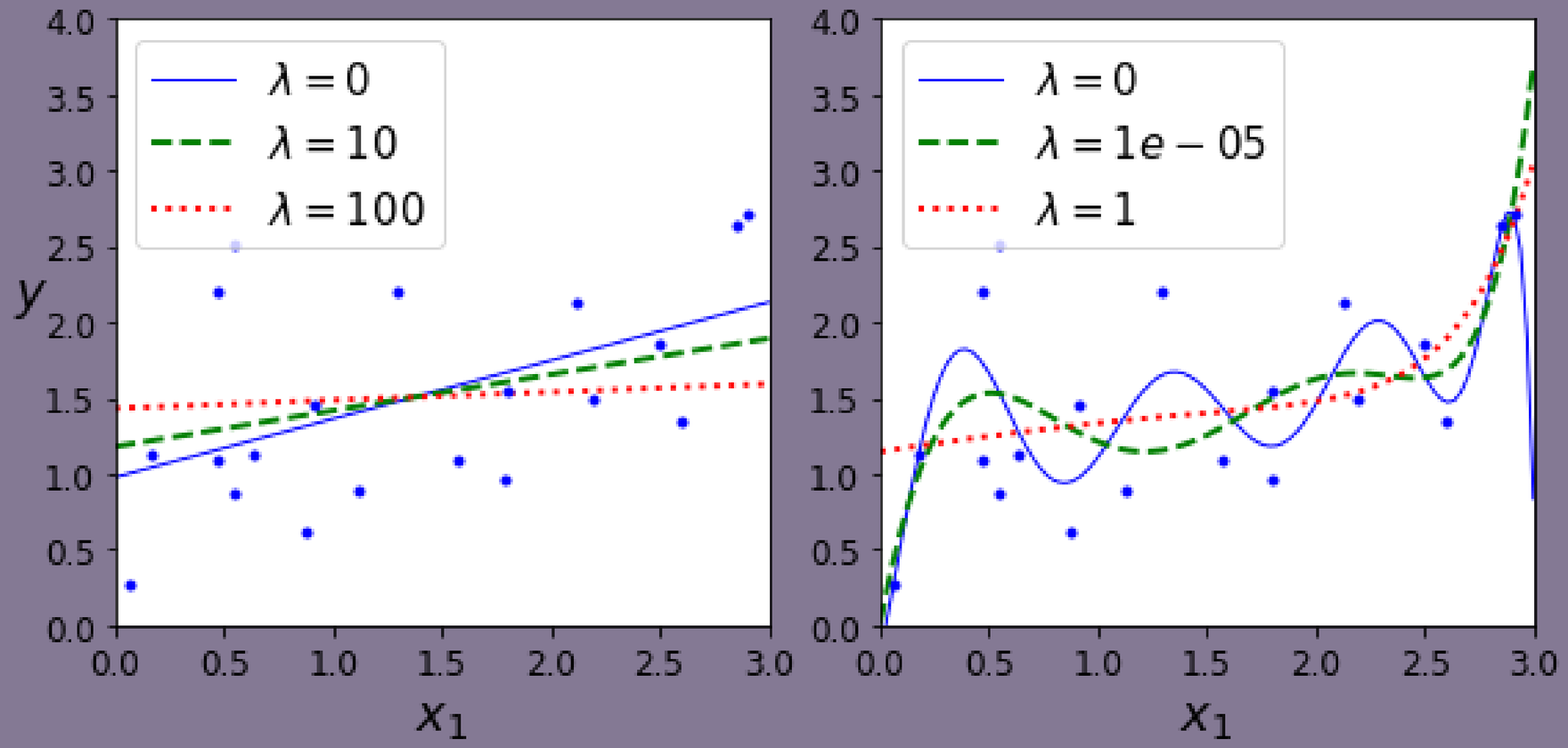
L2 Regularisation

$$L(x, y) \equiv \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2 + \lambda \sum_{i=1}^{n} \theta_i^2$$

It is implemented by adding a regularization term equal to *lambda*∑ (θ_i)^2* to the cost function.
This forces the learning algorithm to not only fit the data but also keep the model weights as small as possible.
That is, for every weight θ in the network, we add the term *lambda*(θ_i)^2* to the objective, where *lambda* is the regularisation strenght.

L2 Regularisation

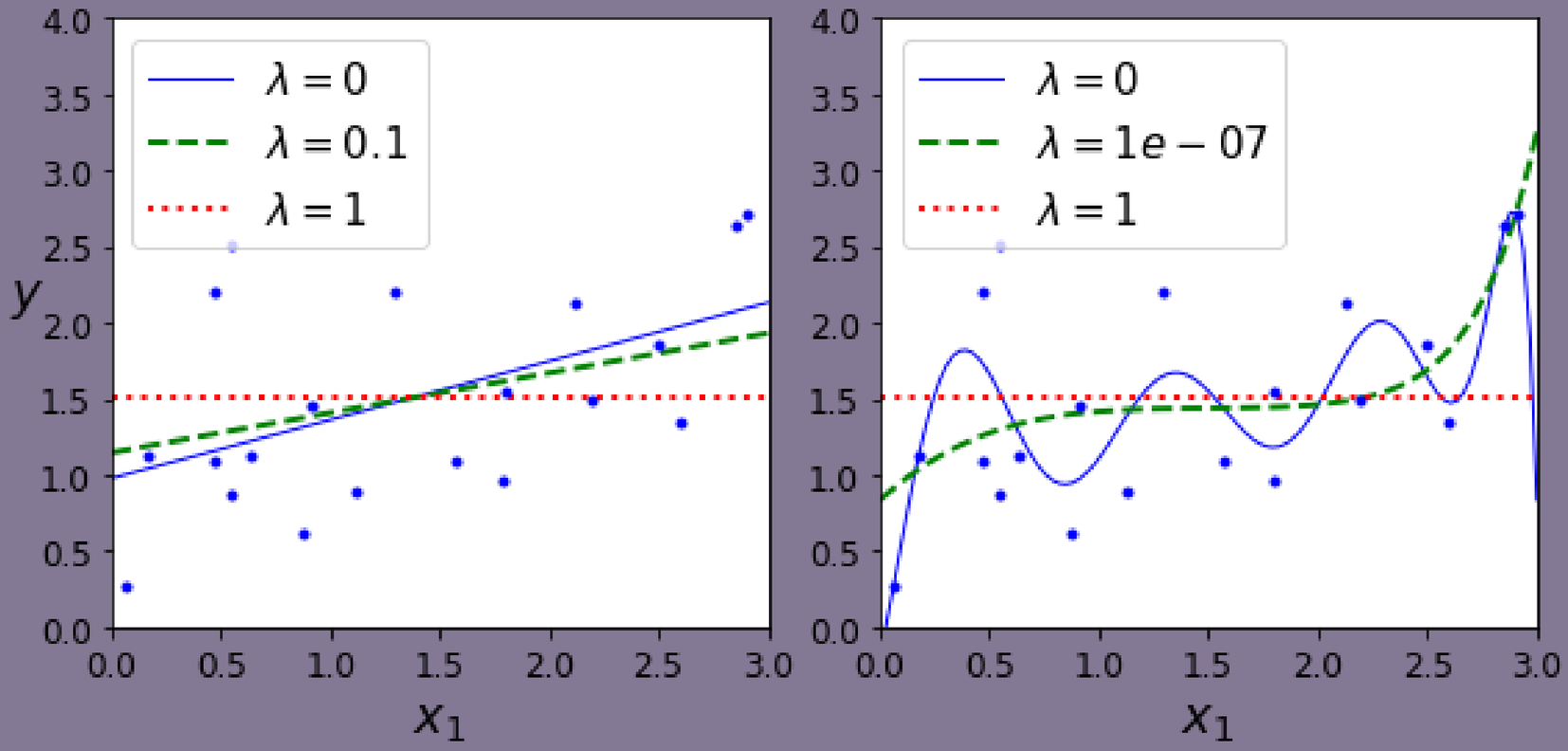
The hyperparameter *lambda* controls how much you want to regularize the model. If *lambda* = 0, then Ridge Regression is just Linear Regression. If α is very large, then all weights end up very close to zero and the result is a flat line going through the data's mean.



L1 Regularisation

$$L(x,y) \equiv \sum_{i=1}^{n}(y_i - h_\theta(x_i))^2 + \lambda \sum_{i=1}^{n}|\theta_i|$$

Just like L2 Regularisation, it adds a regularisation term to the cost function, but it uses the $\ell 1$ norm of the weight vector instead of half the square of the $\ell 2$ norm



L1 Regularisation

An important characteristic of L1 Regularisation is that it tends to eliminate the weights of the least important features (i.e., set them to zero).
In other words, Lasso Regression automatically performs feature selection and outputs a sparse model (i.e., with few nonzero feature weights).

L1 vs L2 Regularisation

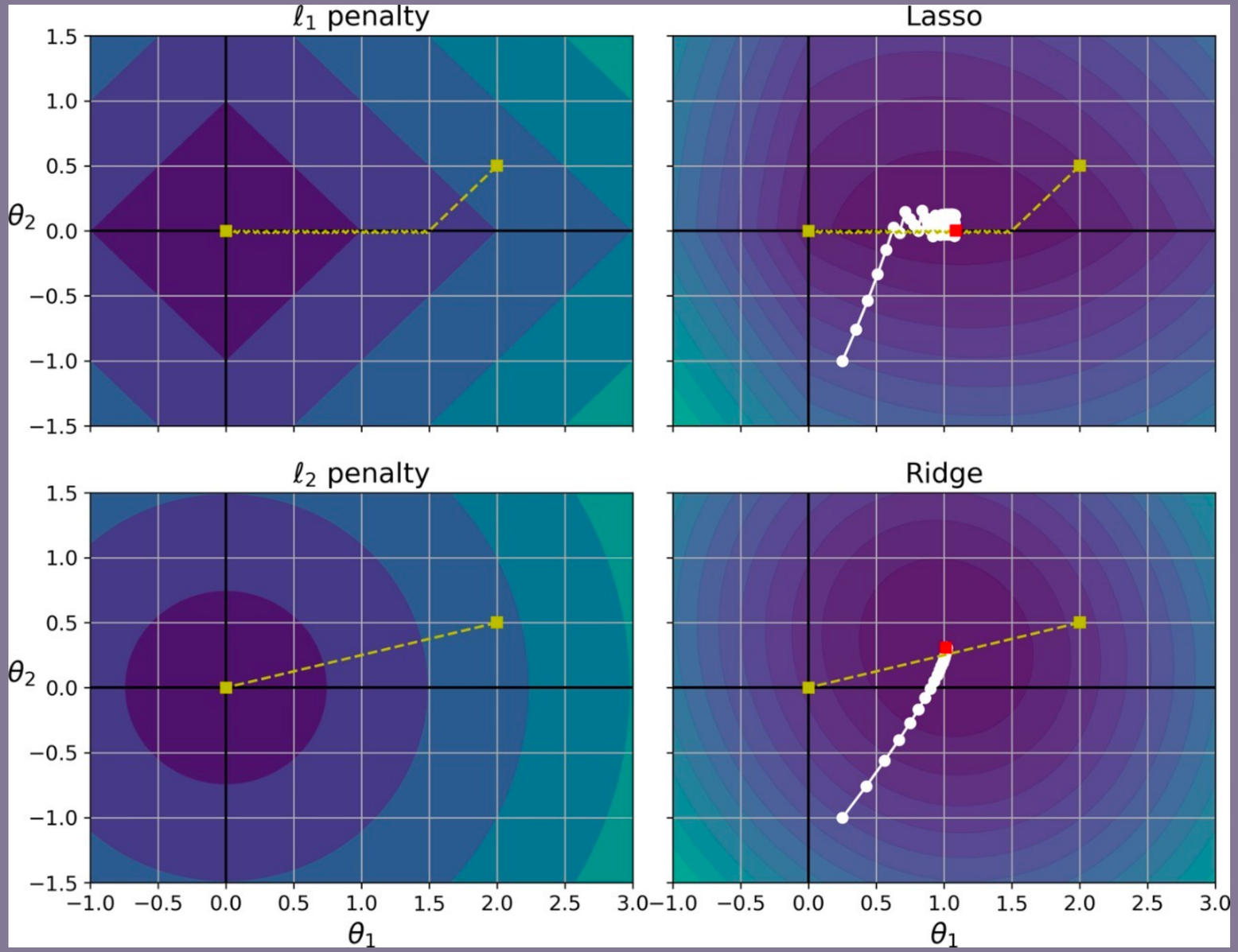
L1:

$$w_{\text{new}} = \begin{cases} (w - \lambda) - H, & w > 0 \\ (w + \lambda) - H, & w < 0 \end{cases}$$

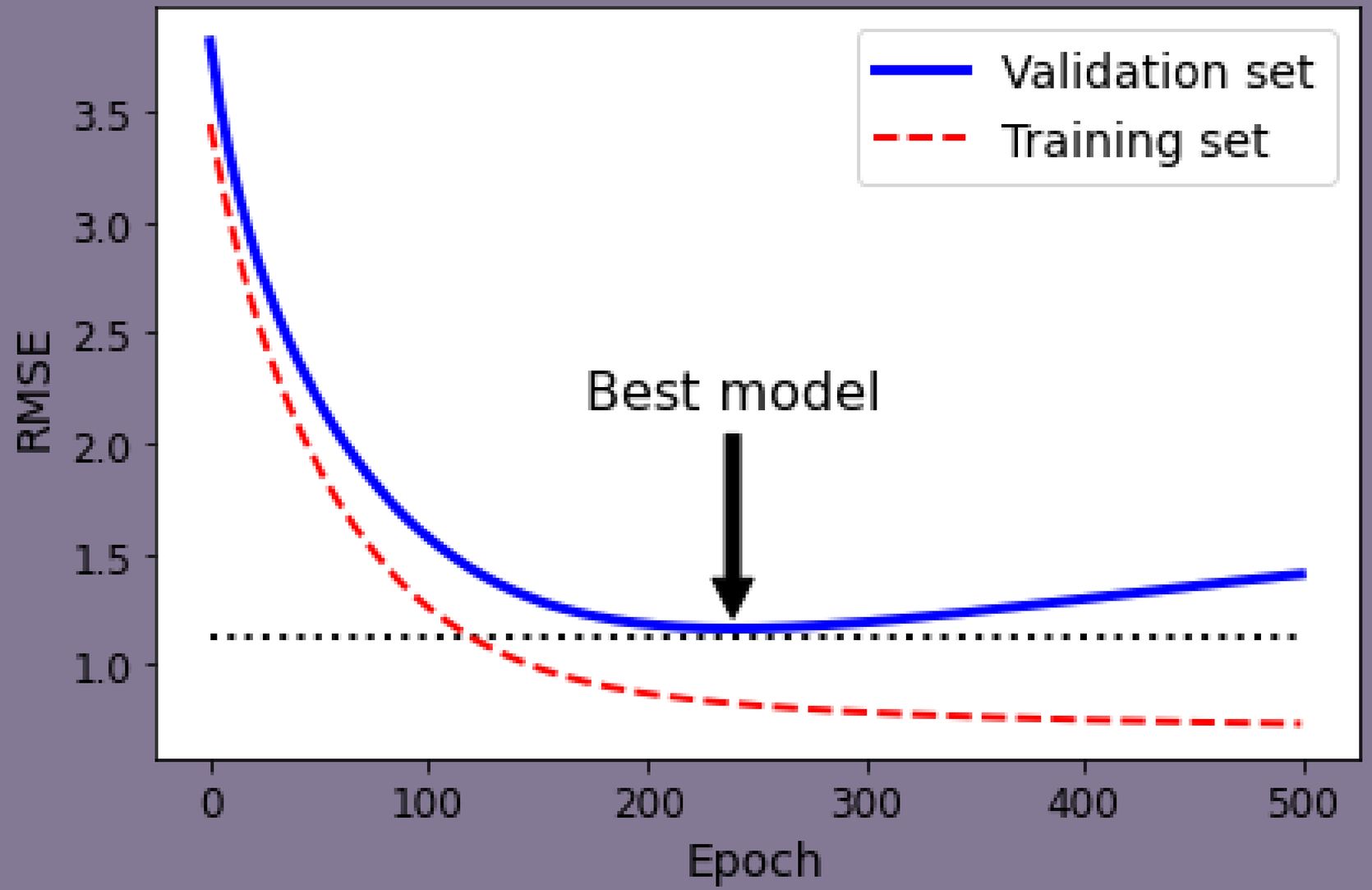
L2:

$$w_{\text{new}} = (w - 2\lambda w) - H$$

L1 vs L2 Regularisation

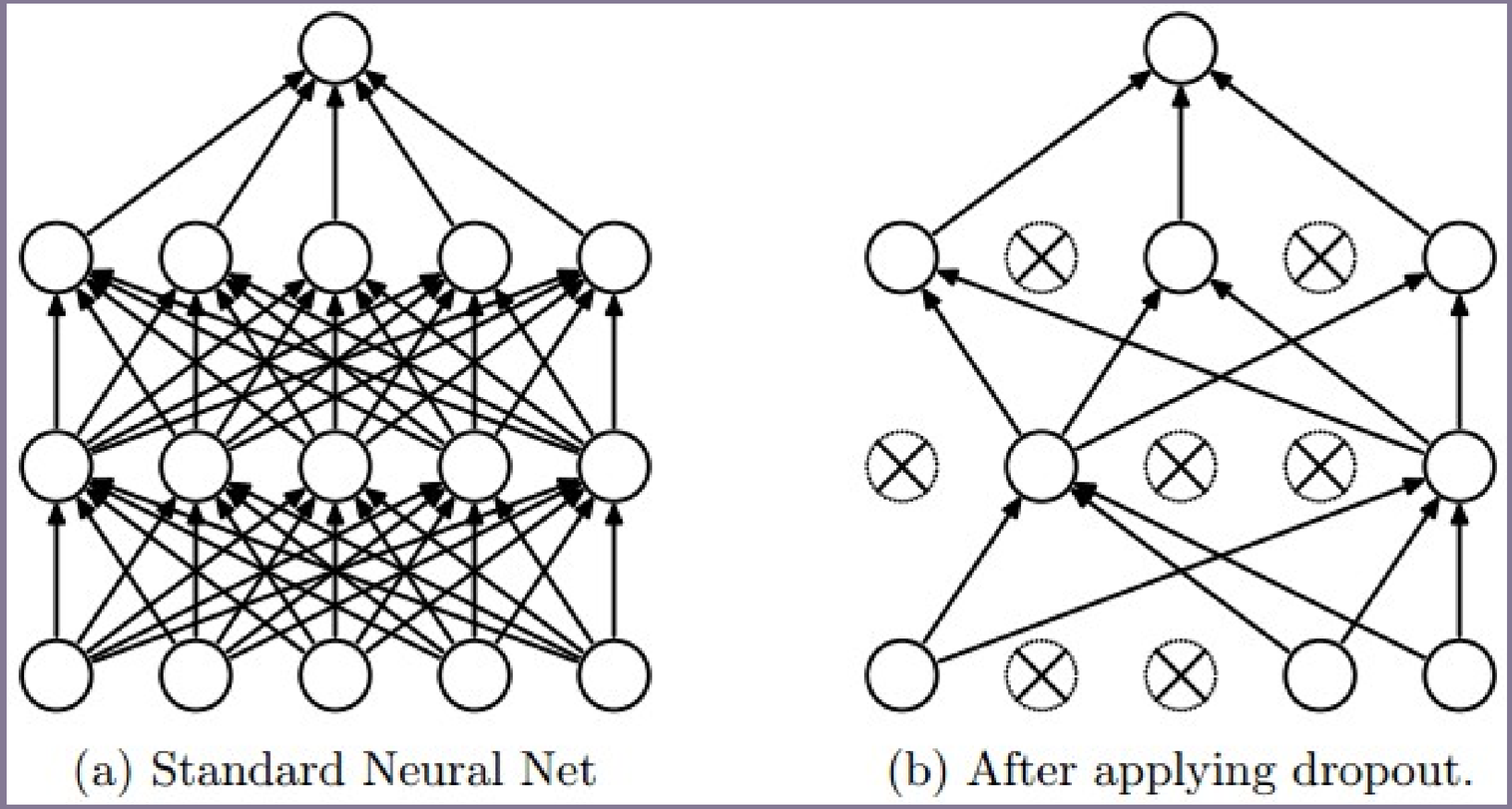


Early Stopping



Dropout

Dropout is an extremely effective, simple and recently introduced regularisation technique that complements the other methods (L1 and L2 norm). While training, dropout is implemented by only keeping a neuron active with some probability *p* (a hyperparameter), or setting it to zero otherwise.



(a) Standard Neural Net

(b) After applying dropout.

Batch Normalisation

Although using He initialization along with ELU (or any variant of ReLU) can significantly reduce the danger of the vanishing/exploding gradients problems at the beginning of training, it doesn’t guarantee that they won’t come back during training. In a 2015 paper, 8 Sergey Ioffe and Christian Szegedy proposed a technique called Batch Normalization (BN) addresses these problems.

This operation simply zero-centers and normalizes each input, then scales and shifts the result using two new parameter vectors per layer: one for scaling, the other for shifting. In other words, the operation lets the model learn the optimal scale and mean of each of the layer’s inputs.

Batch Normalisation

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

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

Input: *x*: *N * D*
Learnable params: *γ*, β: *D*
Intermediates: μ, σ: *D*
x_hat: *N*D*
Output: *y*: *N*D*

All these calculated in train time, so what about test time?

Some Extra Topics