Shallow vs Deep NNs
DSE 3151 Deep Learning

DSE 3151, B.Tech Data Science & Engineering

August 2023

Rohini R Rao & Abhilash Pai

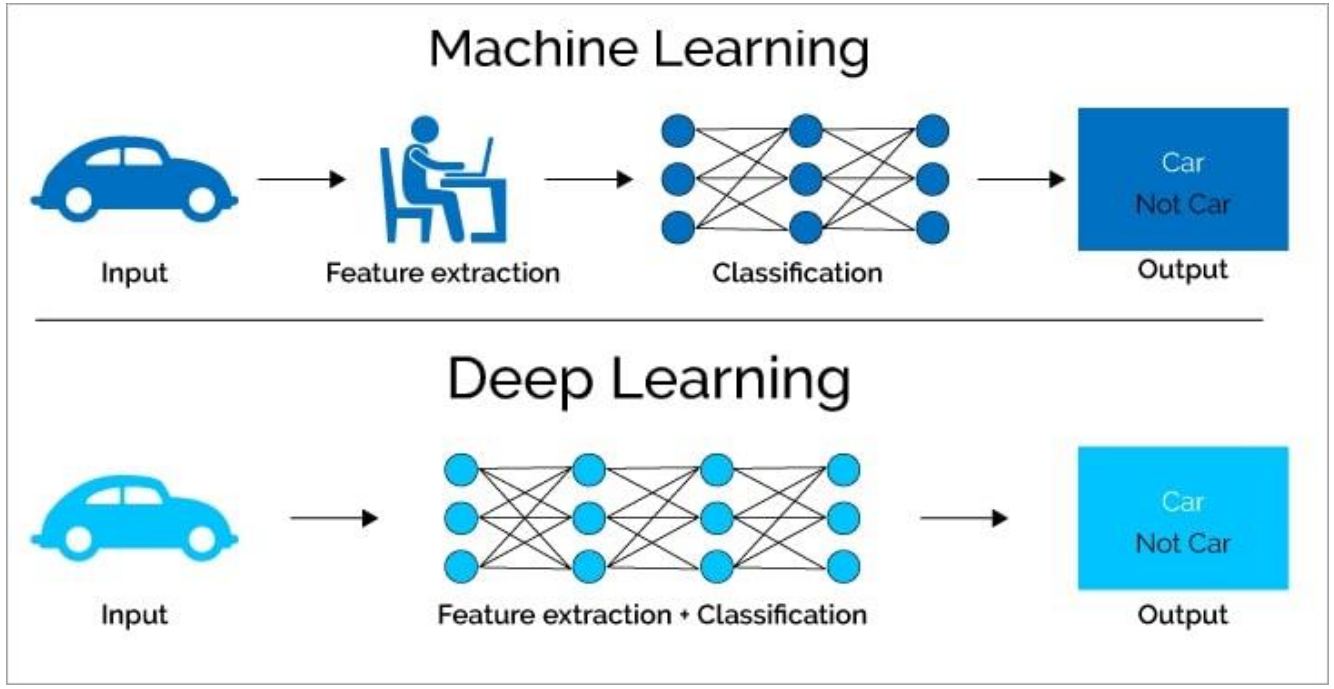
Department of Data Science and Computer Applications

MIT Manipal

Contents

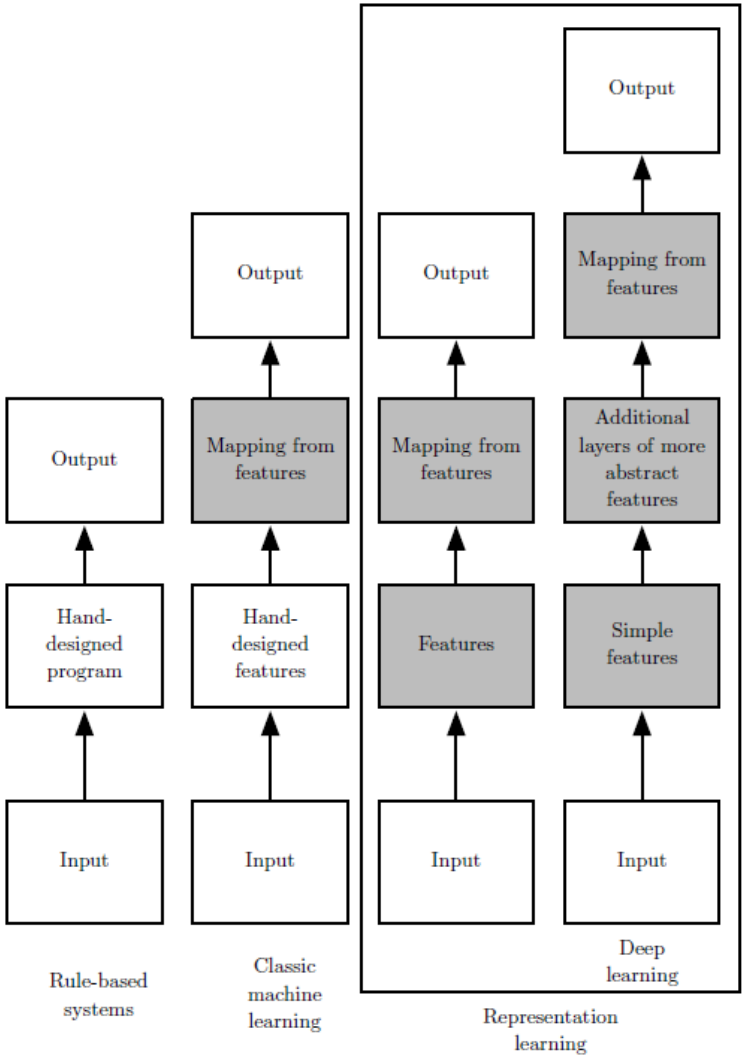
- Shallow Networks
  - Neural Network Representation
  - Back Propagation
  - Vectorization
- Deep Neural Networks
  - Example: Learning XOR
  - Architecture Design
  - Loss Functions
  - Metrics
  - Gradient-Based Learning
  - Optimization
  - Diagnosing Learning Curves
  - Strategies for overfitting
  - Learning rate scheduling

Deep Learning and Machine Learning

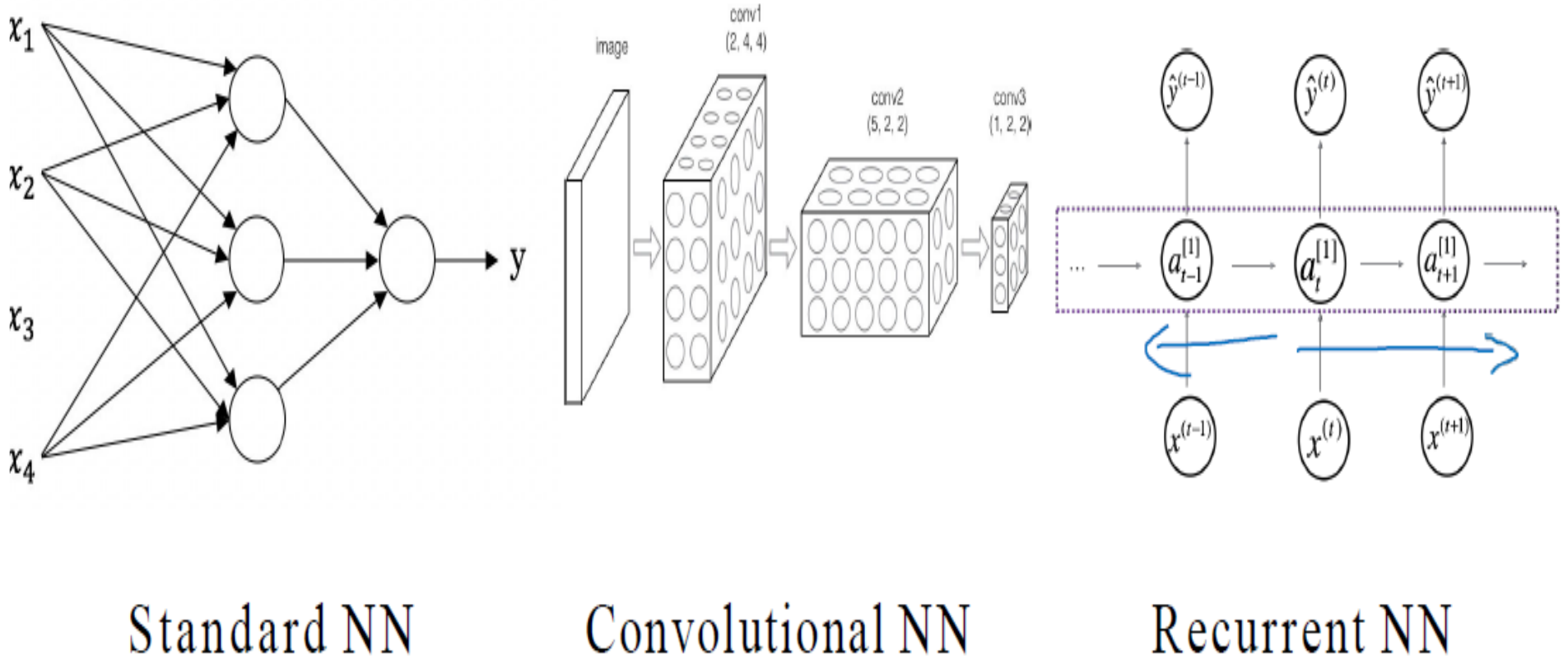


(Source: softwaretestinghelp.com)

Learning Multiple Components



Neural Network Examples



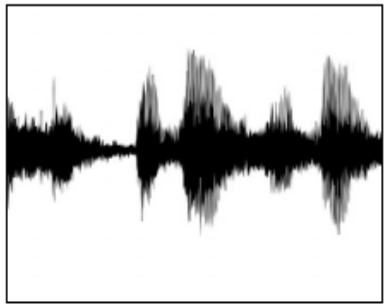
Standard NN Convolutional NN Recurrent NN

Structured Data

Size	#bedrooms	...	Price (1000$s)
2104	3		400
1600	3		330
2400	3		369
...	...		...
3000	4		540

User Age	Ad Id	...	Click
41	93242		1
80	93287		0
18	87312		1
...	...		...
27	71244		1

Unstructured Data



Audio

Image

Four scores and seven years ago…

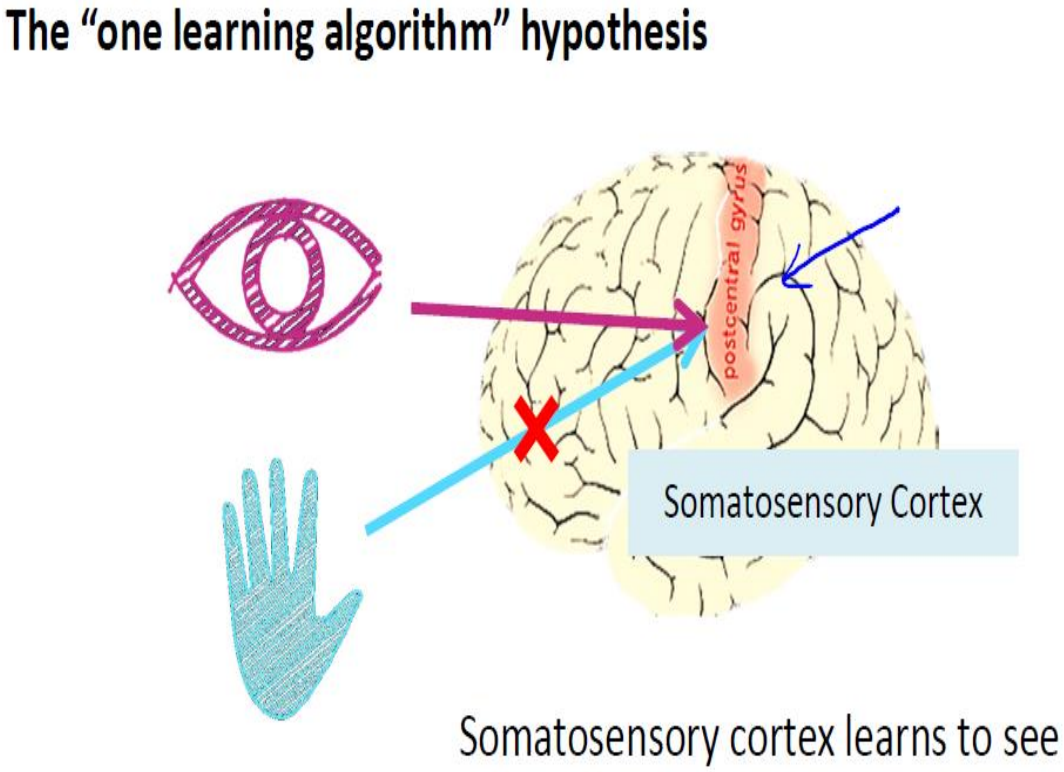
Text

Applications of Deep Learning

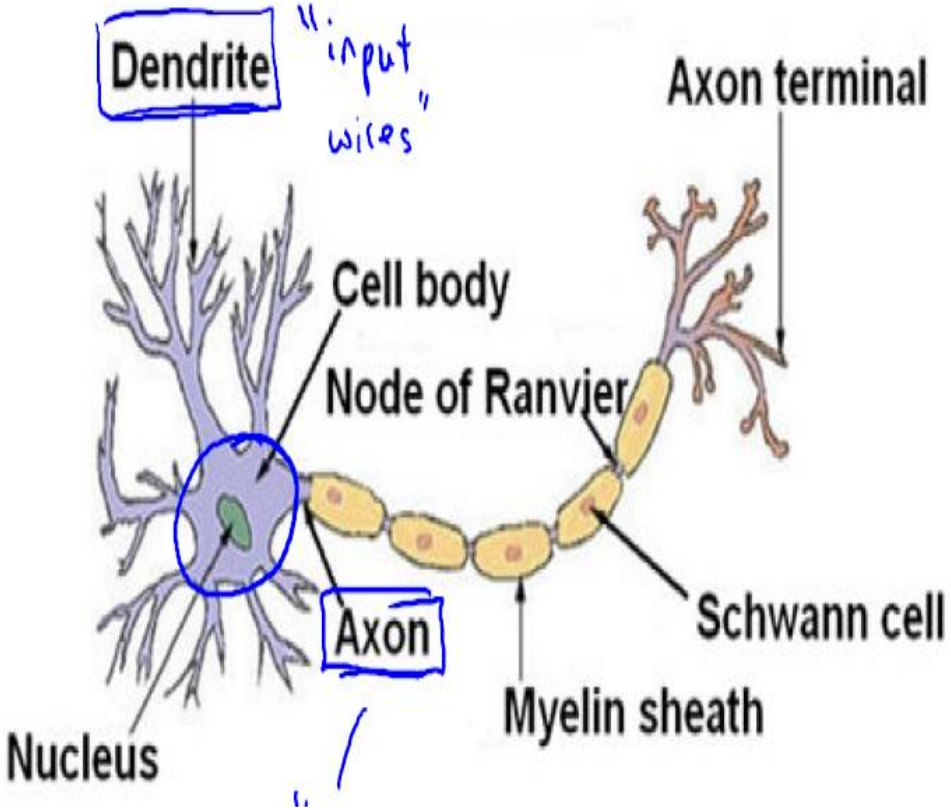
Input(x)	Output (y)	Application
Home features	Price	Real Estate
Ad, user info	Click on ad? (0/1)	Online Advertising
Image	Object (1,...,1000)	Photo tagging
Audio	Text transcript	Speech recognition
English	Chinese	Machine translation
Image, Radar info	Position of other cars	Autonomous driving

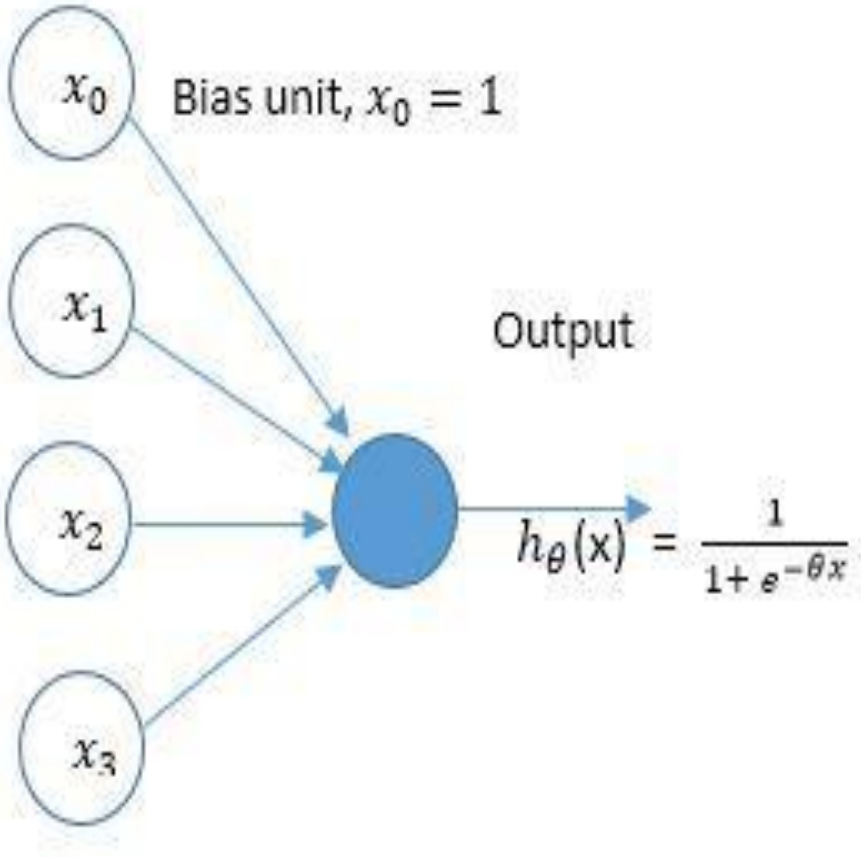
Neural Network

- is a set of connected input/output units in which each connection has
- a weight associated with it.
- During the learning phase, the network learns by adjusting
- the weights so as to be able to predict the correct class label of the input tuples.
- Also referred to as **Connectionist learning**



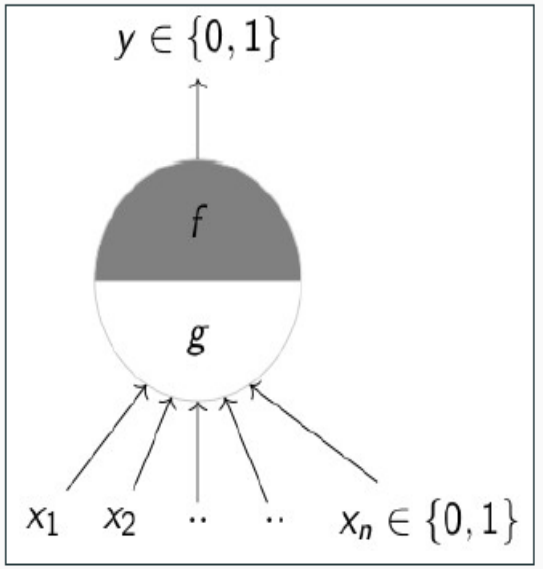
Neuron is a computational , logistic unit

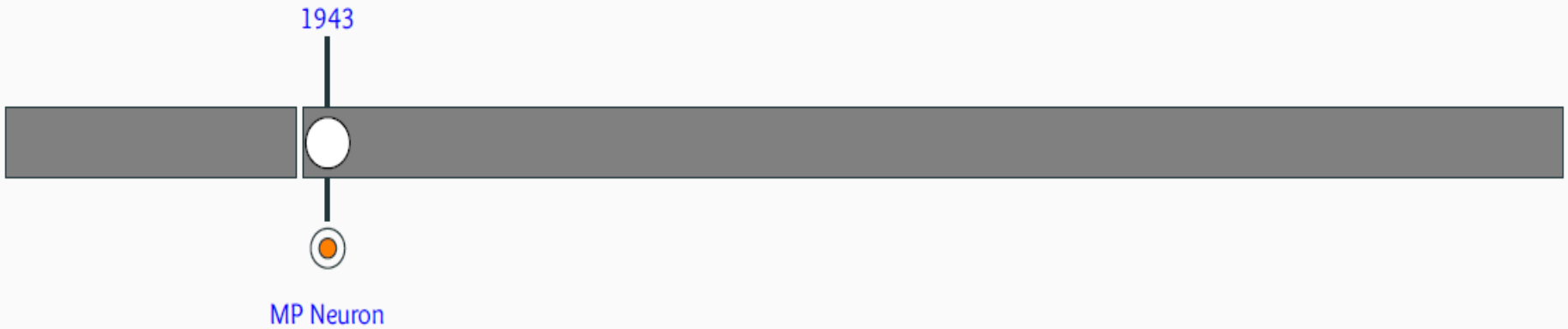




McCulloch Pitts Neuron

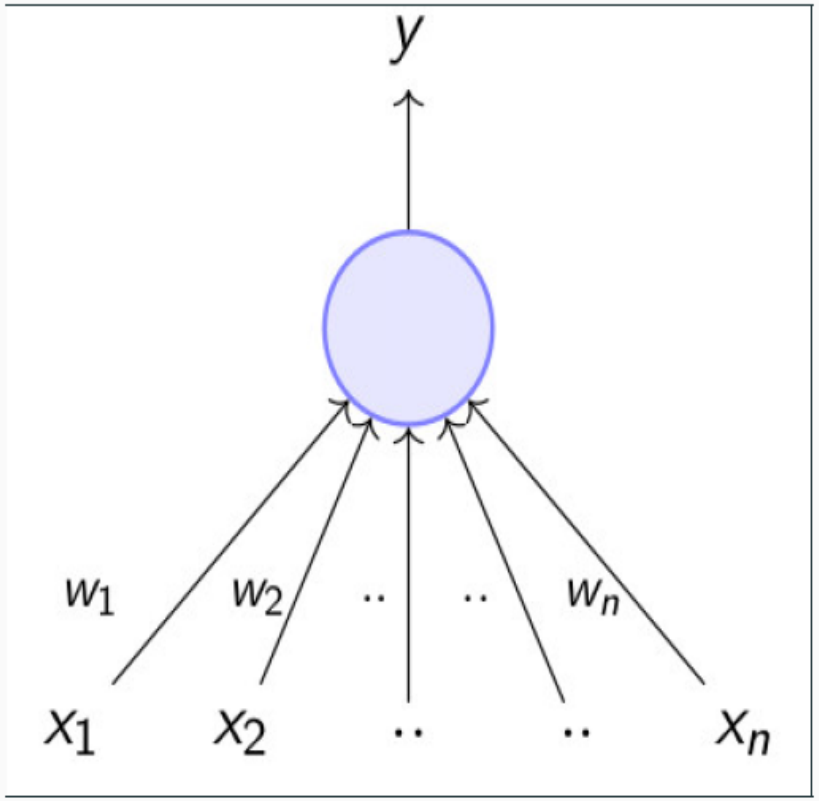
McCulloch (neuroscientist) and Pitts (logician) proposed a highly simplified model of the neuron (1943) [2]

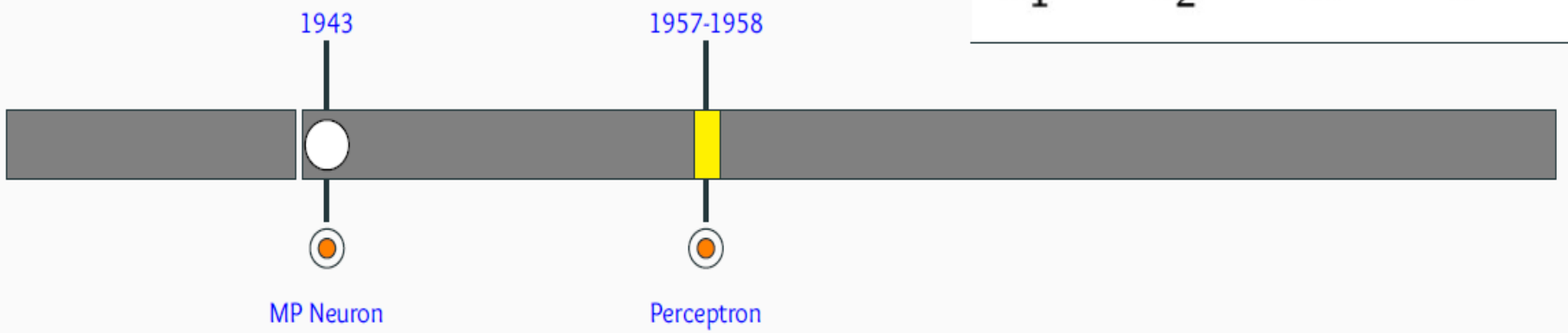




Perceptron

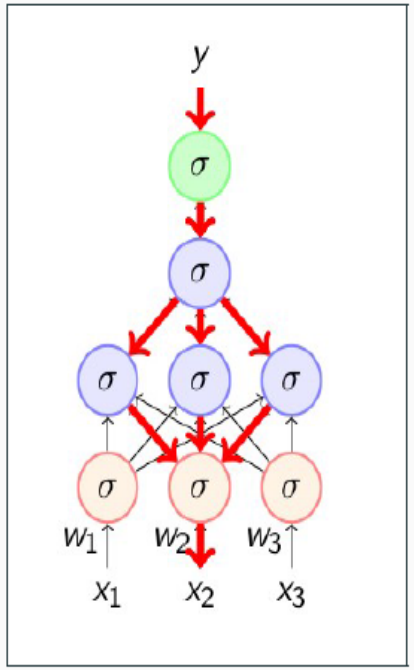
"the perceptron may eventually be able to learn, make decisions, and translate languages" -Frank Rosenblatt

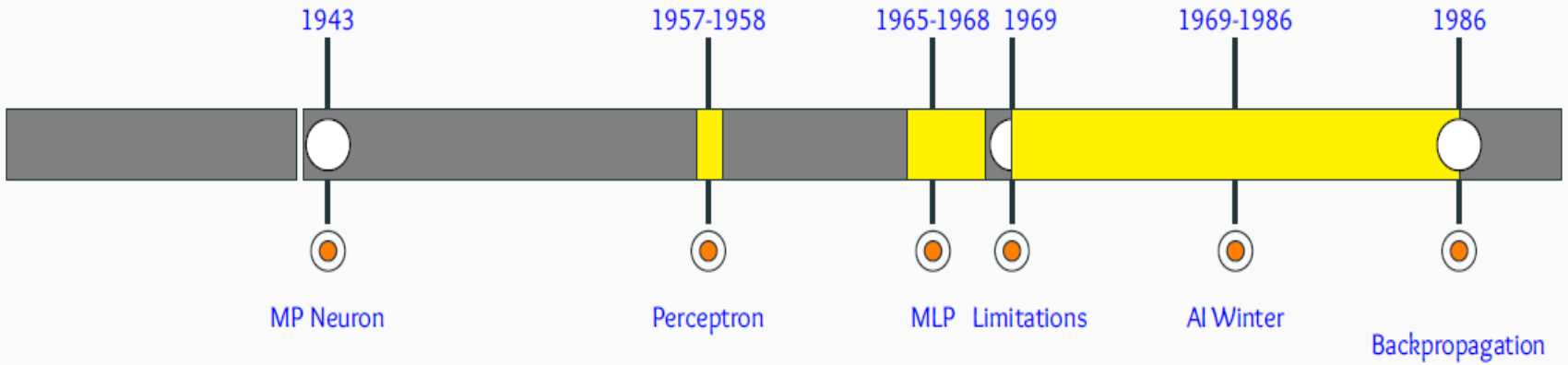




Backpropagation

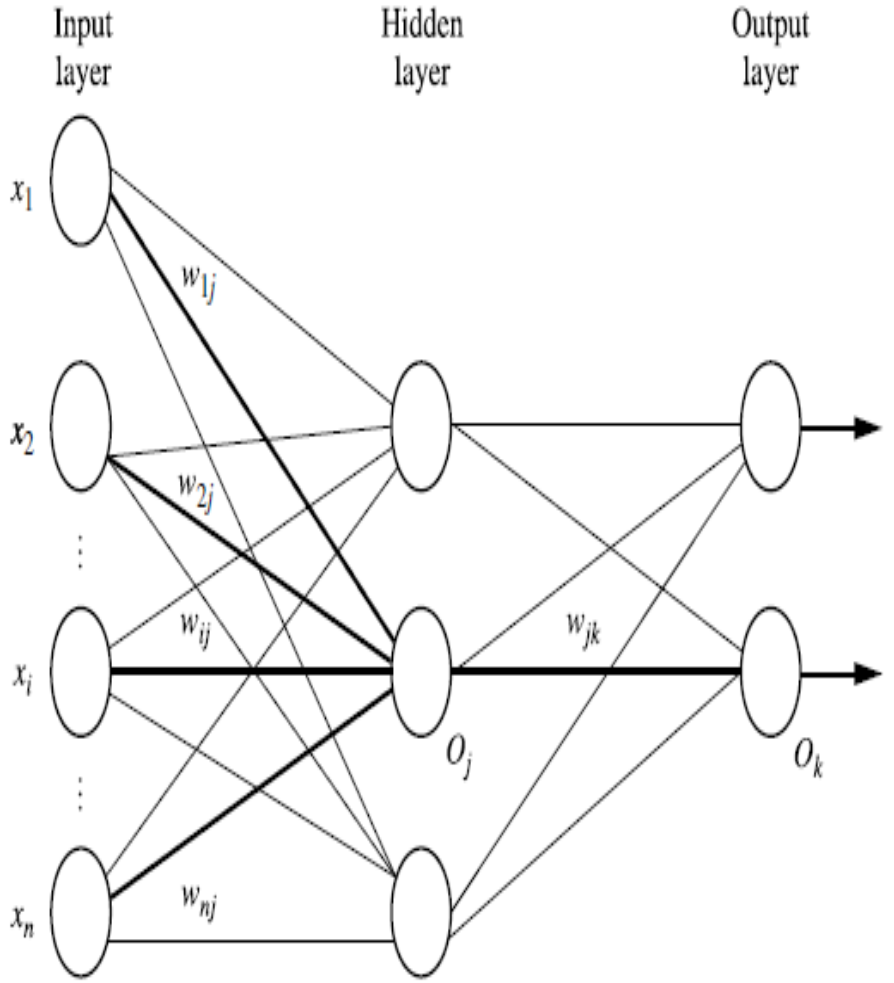
- Discovered and rediscovered several times throughout 1960's and 1970's
- Werbos(1982) [5] first used it in the context of artificial neural networks
- Eventually popularized by the work of Rumelhart et. al. in 1986 [6]





Multilayer Feed-Forward Neural Network

- consists of an *input layer*, one or more *hidden layers*, and an *output layer*
- units in
  - input layer are **input units**
  - hidden and output layer are **neurodes**
- feed-forward network since none of the weights cycles back



Back Propagation Algorithm

(1) Initialize all weights and biases in *network*;
(2) while terminating condition is not satisfied {
(3) for each training tuple X in D {
(4) // Propagate the inputs forward:
(5) for each input layer unit j {
(6) $O_j = I_j$; // output of an input unit is its actual input value
(7) for each hidden or output layer unit j {
(8) $I_j = \sum_i w_{ij} O_i + \theta_j$; //compute the net input of unit j with respect to the previous layer, i
(9) $O_j = \frac{1}{1+e^{-I_j}}$; } // compute the output of each unit j
(10) // Backpropagate the errors:
(11) for each unit j in the output layer
(12) $Err_j = O_j(1 - O_j)(T_j - O_j)$; // compute the error
(13) for each unit j in the hidden layers, from the last to the first hidden layer
(14) $Err_j = O_j(1 - O_j) \sum_k Err_k w_{jk}$; // compute the error with respect to the next higher layer, k
(15) for each weight w_{ij} in *network* {
(16) $\Delta w_{ij} = (l) Err_j O_i$; // weight increment
(17) $w_{ij} = w_{ij} + \Delta w_{ij}$; } // weight update
(18) for each bias θ_j in *network* {
(19) $\Delta \theta_j = (l) Err_j$; // bias increment
(20) $\theta_j = \theta_j + \Delta \theta_j$; } // bias update
(21) } }

Jiawei Han and Micheline Kamber, "Data Mining Concepts And Techniques", 3rd Edition, Morgan Kauffmann

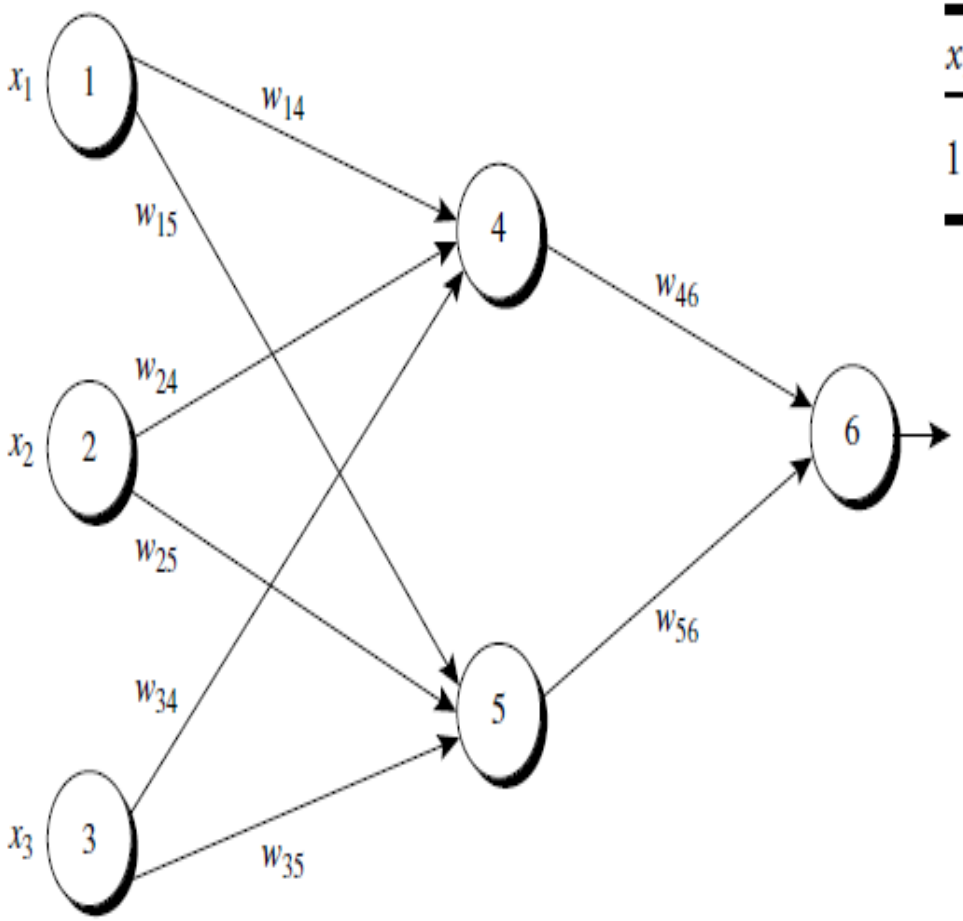
Back Propagation Algorithm

- learns using a gradient descent method to search for a set of weights that fits the training data so as to minimize the mean squared error
- *L* is the **learning rate**, a constant typically having a value between 0.0 and 1.0.
  - helps avoid getting stuck at a local minimum in decision space and encourages finding the global minimum.
- If l is
  - too small, then learning will occur at a very slow pace.
  - too large, then oscillation between inadequate solutions may occur
- rule of thumb is to set the learning rate to 1=/*t*
  - where *t* is the number of iterations through the training set so far
- one iteration through the training set is an **epoch**.
- **Case Updating**
  - updating the weights and biases after the presentation of each tuple.
  - Alternatively,
- **Epoch updating**
  - the weight and bias increments could be accumulated in variables
  - so that the weights and biases are updated after all the tuples in the training set have been presented.

Back Propagation Algorithm

- **Terminating condition:**
  - Training stops when
    - All changes in w_{ij} in the previous epoch are so small as to be below some specified threshold
    - Or The percentage of tuples misclassified in the previous epoch is below some threshold,
    - Or A prespecified number of epochs has expired.
- **The computational efficiency** depends on the
  - time spent training the network.
  - Given $|D|$ tuples and w weights, each epoch requires $O(|D|*w)$ time.
  - In the worst-case scenario, the number of epochs can be exponential in n, the number of inputs.
  - In practice, the time required for the networks to converge is highly variable.

Example



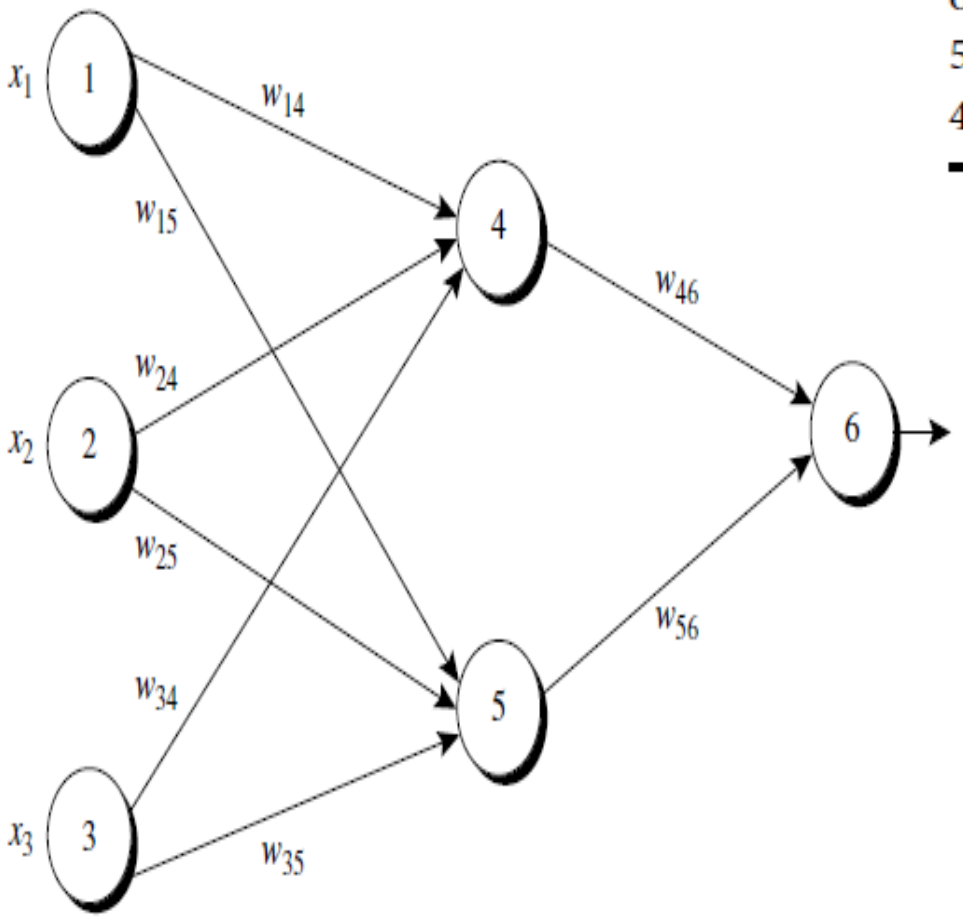
Initial Input, Weight, and Bias Values

x_1	x_2	x_3	w_{14}	w_{15}	w_{24}	w_{25}	w_{34}	w_{35}	w_{46}	w_{56}	θ_4	θ_5	θ_6
1	0	1	0.2	−0.3	0.4	0.1	−0.5	0.2	−0.3	−0.2	−0.4	0.2	0.1

Net Input and Output Calculations

Unit, j	*Net Input, I_j*	*Output, O_j*
4	$0.2+0-0.5-0.4=-0.7$	$1/(1+e^{0.7})=0.332$
5	$-0.3+0+0.2+0.2=0.1$	$1/(1+e^{-0.1})=0.525$
6	$(-0.3)(0.332)-(0.2)(0.525)+0.1=-0.105$	$1/(1+e^{0.105})=0.474$

Example



Calculation of the Error at Each Node

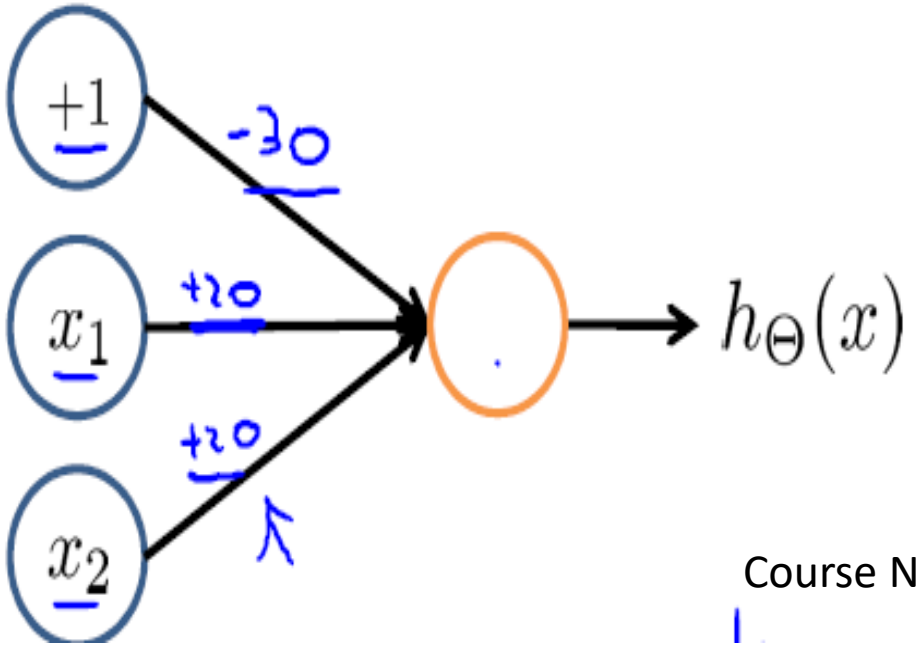
Unit, j	*Err_j*
6	$(0.474)(1-0.474)(1-0.474) = 0.1311$
5	$(0.525)(1-0.525)(0.1311)(-0.2) = -0.0065$
4	$(0.332)(1-0.332)(0.1311)(-0.3) = -0.0087$

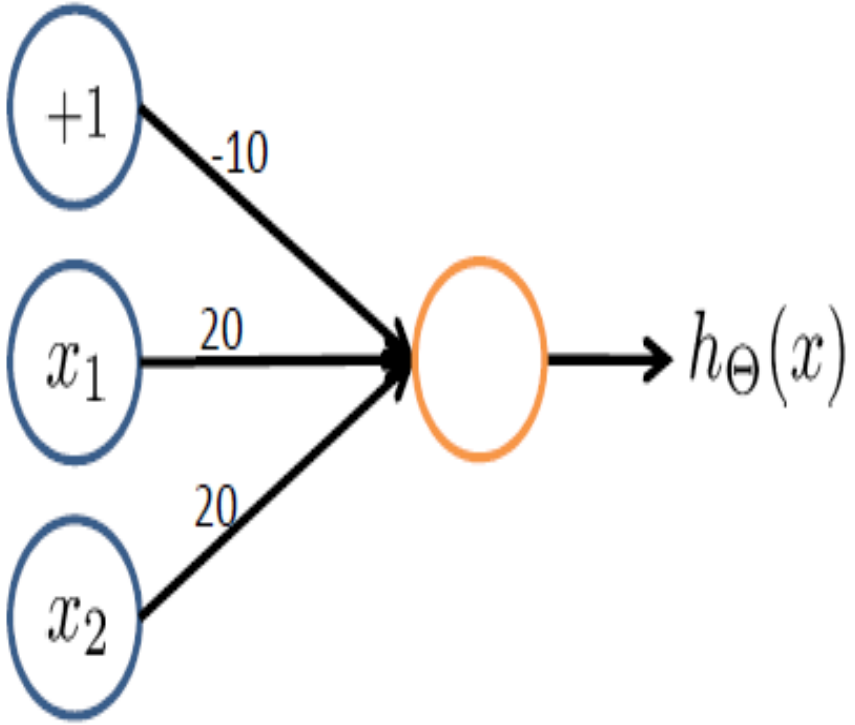
Calculations for Weight and Bias Updating

Weight or Bias	*New Value*
w_{46}	$-0.3 + (0.9)(0.1311)(0.332) = -0.261$
w_{56}	$-0.2 + (0.9)(0.1311)(0.525) = -0.138$
w_{14}	$0.2 + (0.9)(-0.0087)(1) = 0.192$
w_{15}	$-0.3 + (0.9)(-0.0065)(1) = -0.306$
w_{24}	$0.4 + (0.9)(-0.0087)(0) = 0.4$
w_{25}	$0.1 + (0.9)(-0.0065)(0) = 0.1$
w_{34}	$-0.5 + (0.9)(-0.0087)(1) = -0.508$
w_{35}	$0.2 + (0.9)(-0.0065)(1) = 0.194$
θ_6	$0.1 + (0.9)(0.1311) = 0.218$
θ_5	$0.2 + (0.9)(-0.0065) = 0.194$
θ_4	$-0.4 + (0.9)(-0.0087) = -0.408$

Simple AND and Simple OR operations

- $x_1, x_2 \in \{0, 1\}$
- $y = x_1 \text{ AND } x_2$

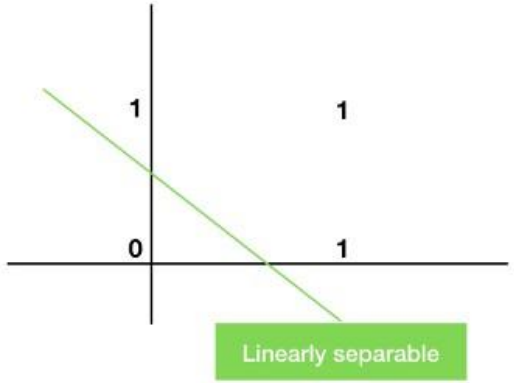




Course Notes – Deep Learning , Andrew NG

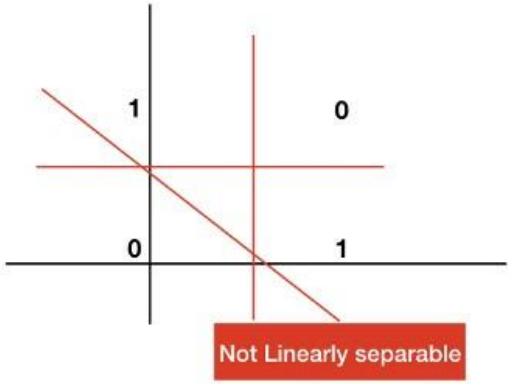
XOR Problem - Perceptron Learning

Inclusive-OR



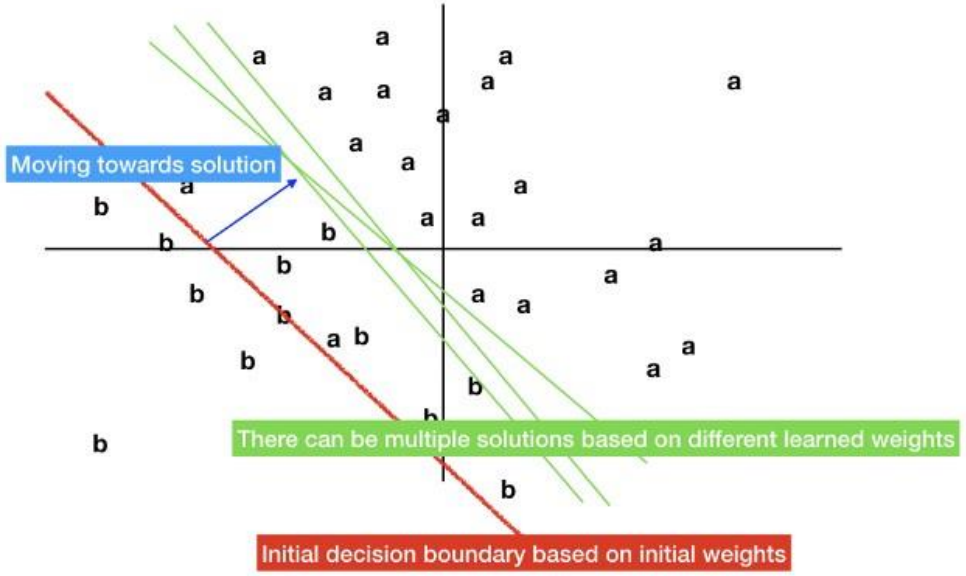
a	b	c
0	0	0
0	1	1
1	0	1
1	1	1

Exclusive-OR



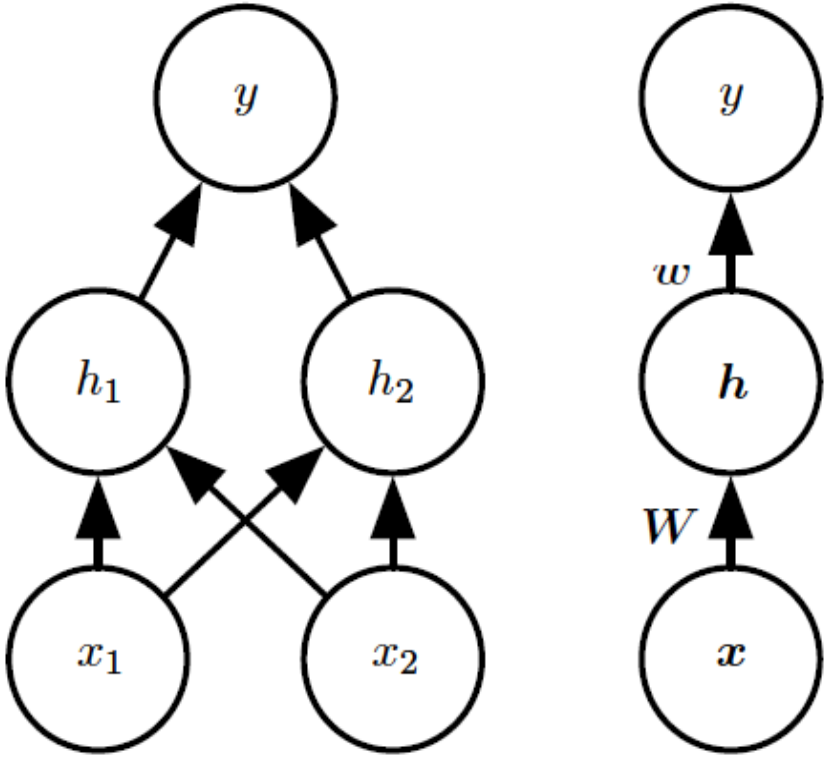
a	b	c
0	0	0
0	1	1
1	0	1
1	1	0

Perceptron Learning



XOR Problem

Network Diagram



Solving XOR

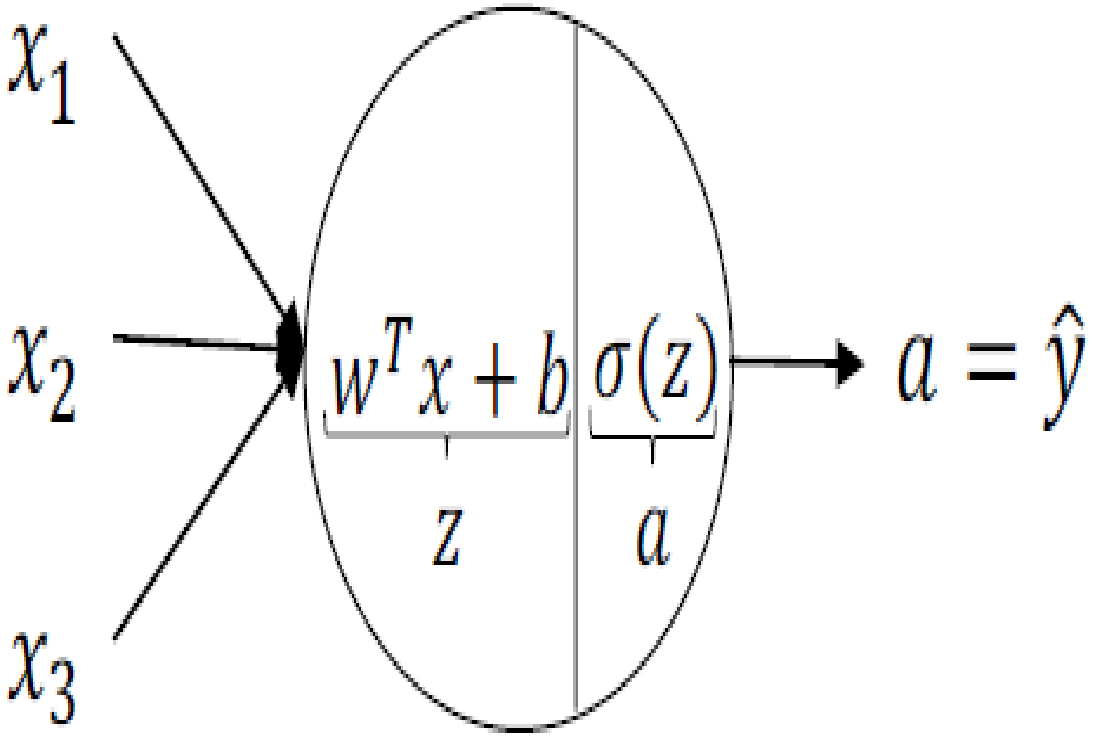
$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^\top \max\{0, \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}\} + b.$$

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad \boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$$

$$\boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \quad b = 0$$

$$\boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix},$$

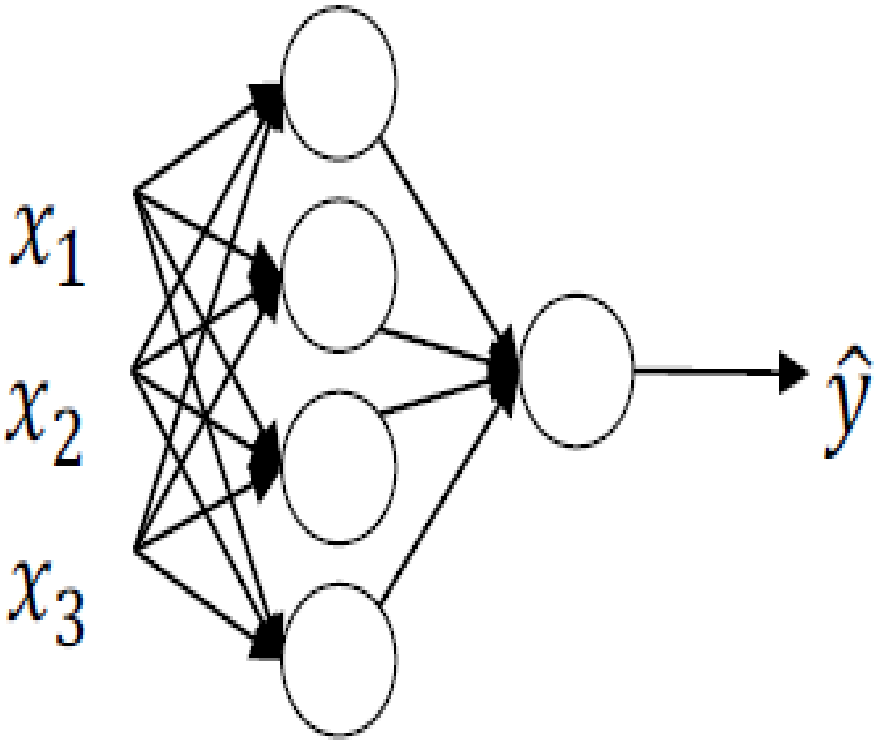
Neural Network Representation



$$z = w^T x + b$$

$$a = \sigma(z)$$

Neural Network Representation



$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}, a_1^{[1]} = \sigma\left(z_1^{[1]}\right)$$

$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}, a_2^{[1]} = \sigma\left(z_2^{[1]}\right)$$

$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]}, a_3^{[1]} = \sigma\left(z_3^{[1]}\right)$$

$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]}, a_4^{[1]} = \sigma\left(z_4^{[1]}\right)$$

$$Z^{[1]} = X^{[1]T} X + b^{[1]}$$

$$A^{[1]} = \sigma\left(Z^{[1]}\right)$$

Course Notes – Deep Learning , Andrew NG

The Shallow Neural Network
Vectorizing across multiple examples

```
for i = 1 to m:
```

$$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]}$$

$$a^{[1](i)} = \sigma(z^{[1](i)})$$

$$z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2]}$$

$$a^{[2](i)} = \sigma(z^{[2](i)})$$

$$Z^{[1]} = W^{[1]T} X + b^{[1]}$$

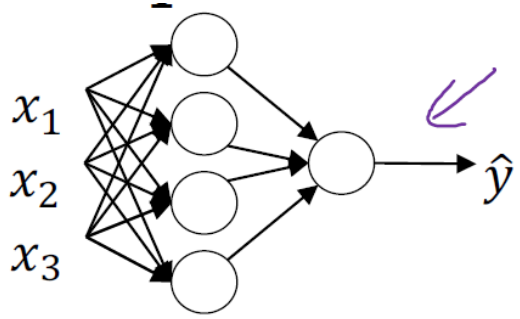
$$A^{[1]} = \sigma\left(Z^{[1]}\right)$$

$$Z^{[2]} = W^{[2]T} A^{[1]} + b^{[2]}$$

$$\hat{y} = A^{[2]} = \sigma\left(Z^{[2]}\right)$$

Course Notes – Deep Learning , Andrew NG

The Shallow Neural Network
Vectorizing across multiple examples



$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

$$A^{[1]} = \begin{bmatrix} | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \cdots & a^{[1](m)} \\ | & | & & | \end{bmatrix}$$

```
for i = 1 to m
```

$$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$$
$$a^{[1](i)} = \sigma(z^{[1](i)})$$
$$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$$
$$a^{[2](i)} = \sigma(z^{[2](i)})$$

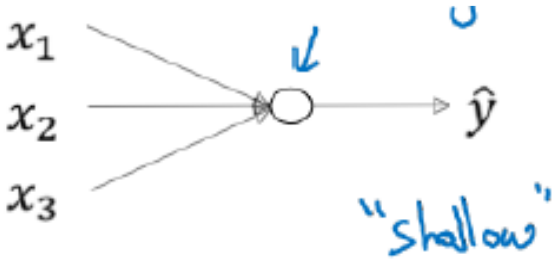
$A^{[0]}$ $\quad x = a^{[0]} \quad x^{(i)} = a^{[0](i)}$

$$Z^{[1]} = W^{[1]}X + b^{[1]} \quad \leftarrow \quad w^{[1]}A^{[0]} + b^{[1]}$$
$$A^{[1]} = \sigma(Z^{[1]})$$
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = \sigma(Z^{[2]})$$

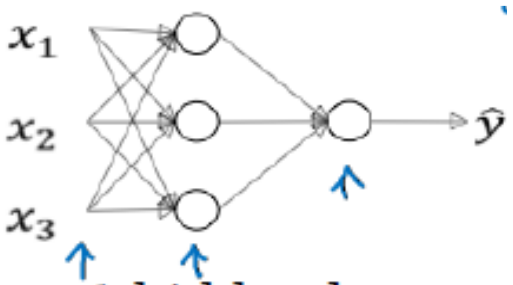
Andrew Ng

Course Notes – Deep Learning , Andrew NG

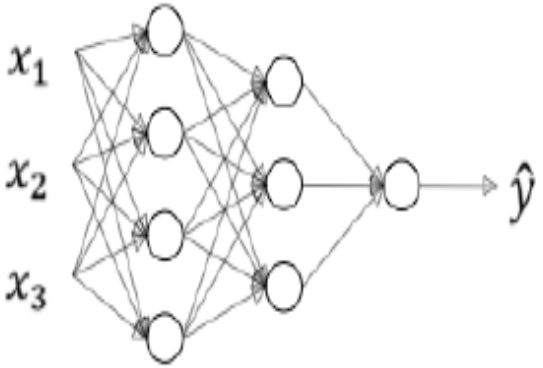
Deep vs Shallow Network



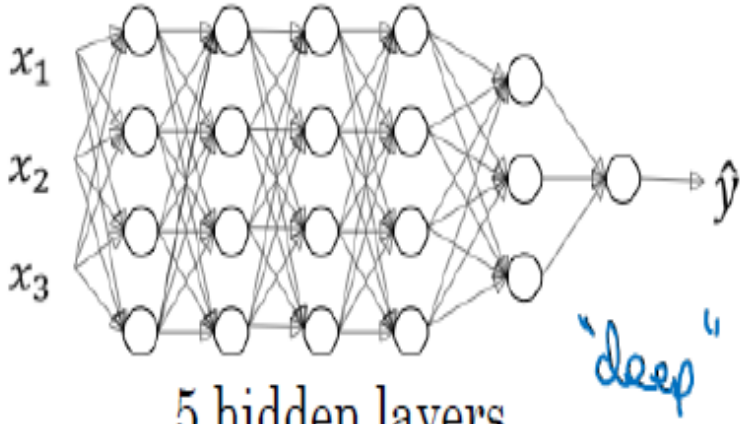
logistic regression



1 hidden layer

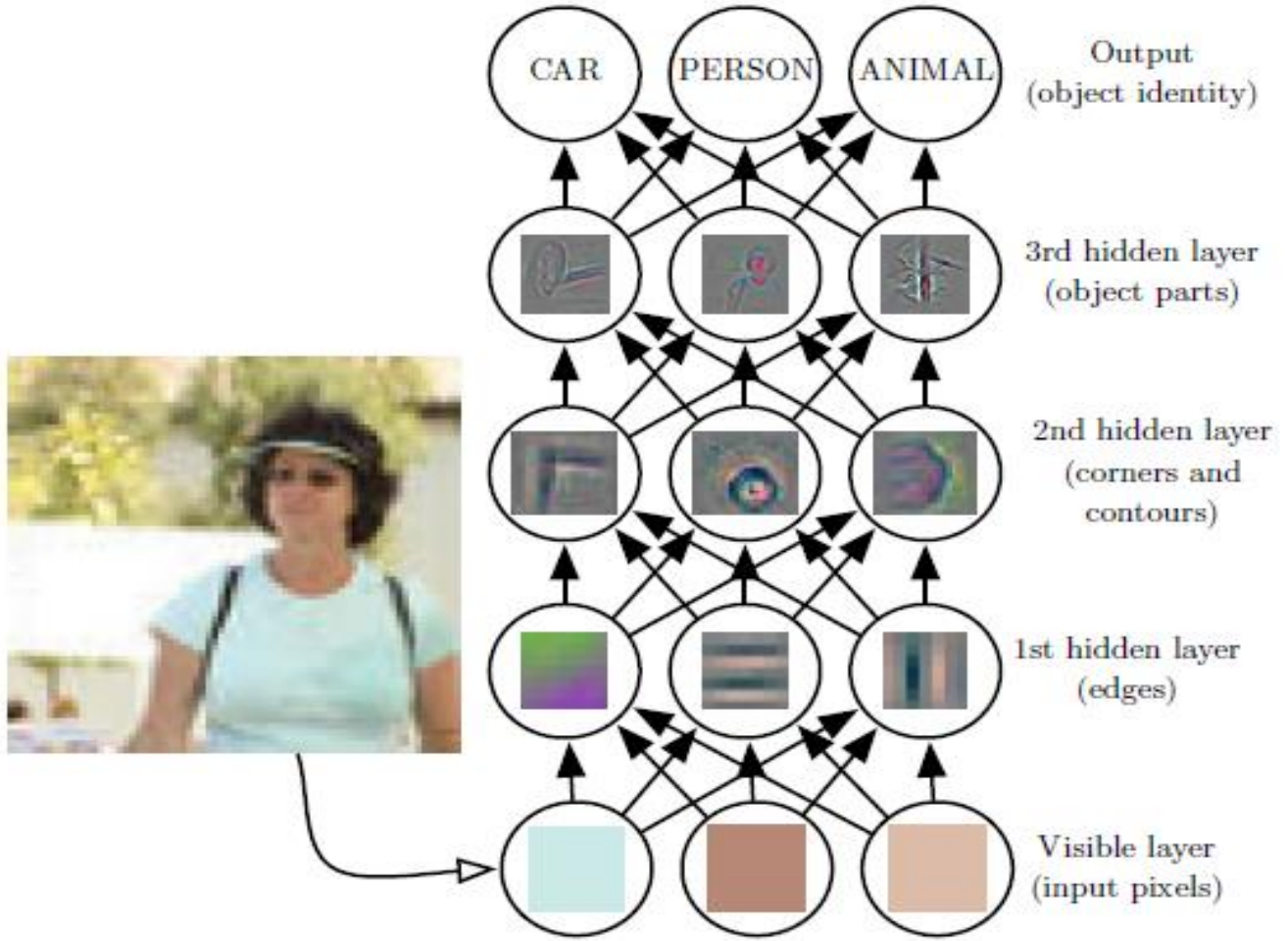


2 hidden layers



5 hidden layers

Depth: Repeated Composition



Neural Network learning as optimization

- Cannot calculate the perfect weights for a neural network since there are too many unknowns
- Instead, the problem of learning is as a search or optimization problem
- An algorithm is used to navigate the space of possible sets of weights the model may use in order to make good or good enough predictions

- **Objective of optimizer** is to minimize the loss function or the error term.
- **Loss function**
  - gives the difference between observed value from the predicted value.
  - must be
    - **Continuous**
    - **Differentiable at each point (**allows the use of gradient-based optimization)
  - To minimize the loss generated from any model compute
    - The magnitude that is by how much amount to decrease or increase, and
    - direction in which to move

Machine Learning Setup

Data: $\{x_i, y_i\}_{i=1}^{n}$

Model: Our approximation of the relation between $\mathbf{x}$ and y. For example,

$$\hat{y} = \frac{1}{1 + e^{-(\mathbf{w}^T\mathbf{x})}}$$

$$or \quad \hat{y} = \mathbf{w}^T\mathbf{x}$$

$$or \quad \hat{y} = \mathbf{x}^T\mathbf{W}\mathbf{x}$$

or just about any function

Parameters: In all the above cases, w is a parameter which needs to be learned from the data

Learning algorithm: An algorithm for learning the parameters (w) of the model (for example, perceptron learning algorithm, gradient descent, etc.)

Objective/Loss/Error function: To guide the learning algorithm

Activation Functions

- To make the network robust use of **Activation or Transfer functions**.
- Activations functions introduce non-linear properties in the neural networks.
- A good Activation function has the following properties:
- **Monotonic Function:**
  - should be either entirely non-increasing or non-decreasing.
  - If not monotonic then increasing the neuron's weight might cause it to have less influence on reducing the error of the cost function.
- **Differential:**
  - mathematically means the change in y with respect to change in x.
  - should be differential because we want to calculate the change in error with respect to given weights at the time of gradient descent.
- **Quickly Converging:**
  - Should reach its desired value fast.

Gradient Descent Rule

- The direction u that we intend to move in should be at 180° w.r.t. the gradient
- In other words, move in a direction opposite to the gradient

Parameter Update Equations

$$w_{t+1} = w_t - \eta \nabla w_t$$

$$b_{t+1} = b_t - \eta \nabla b_t$$

$$where, \nabla w_t = \frac{\partial \mathscr{L}(w,b)}{\partial w}_{at\ w=w_t, b=b_t}, \nabla b = \frac{\partial \mathscr{L}(w,b)}{\partial b}_{at\ w=}$$

Algorithm: gradient_descent()

$t \leftarrow 0$;

$max\_iterations \leftarrow 1000$;

while $t < max\_iterations$ **do**

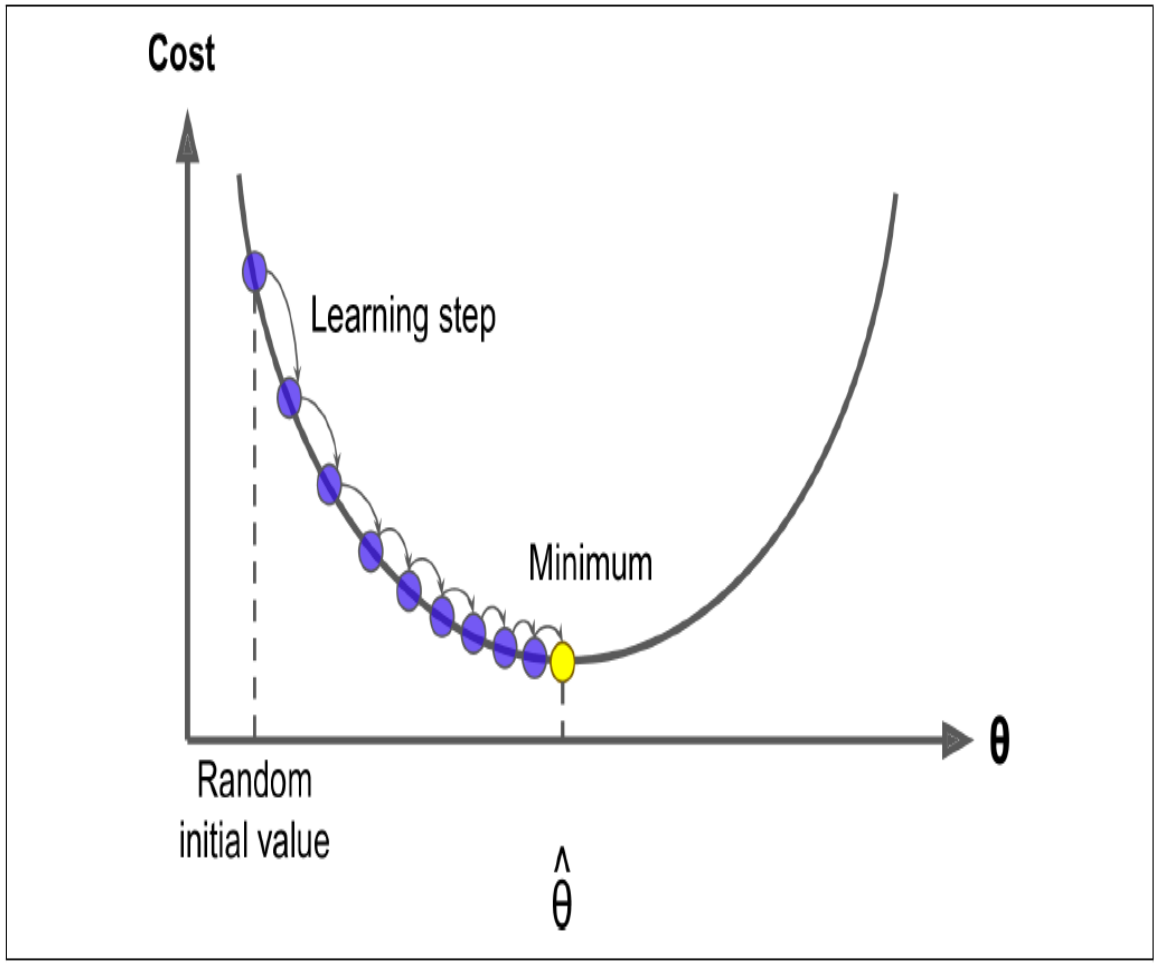
  $w_{t+1} \leftarrow w_t - \eta \nabla w_t$;

  $b_{t+1} \leftarrow b_t - \eta \nabla b_t$;

  $t \leftarrow t + 1$;

end

Gradient Descent



- Generic Optimization Algorithm
- Starts with random values
- Improves gradually , in an attempt to decrease loss function
- Until algorithm converges to minimum
- Learning Rate-
  - Hyperparameter
  - Indicates size of steps

Gradient Descent & Learning Rate

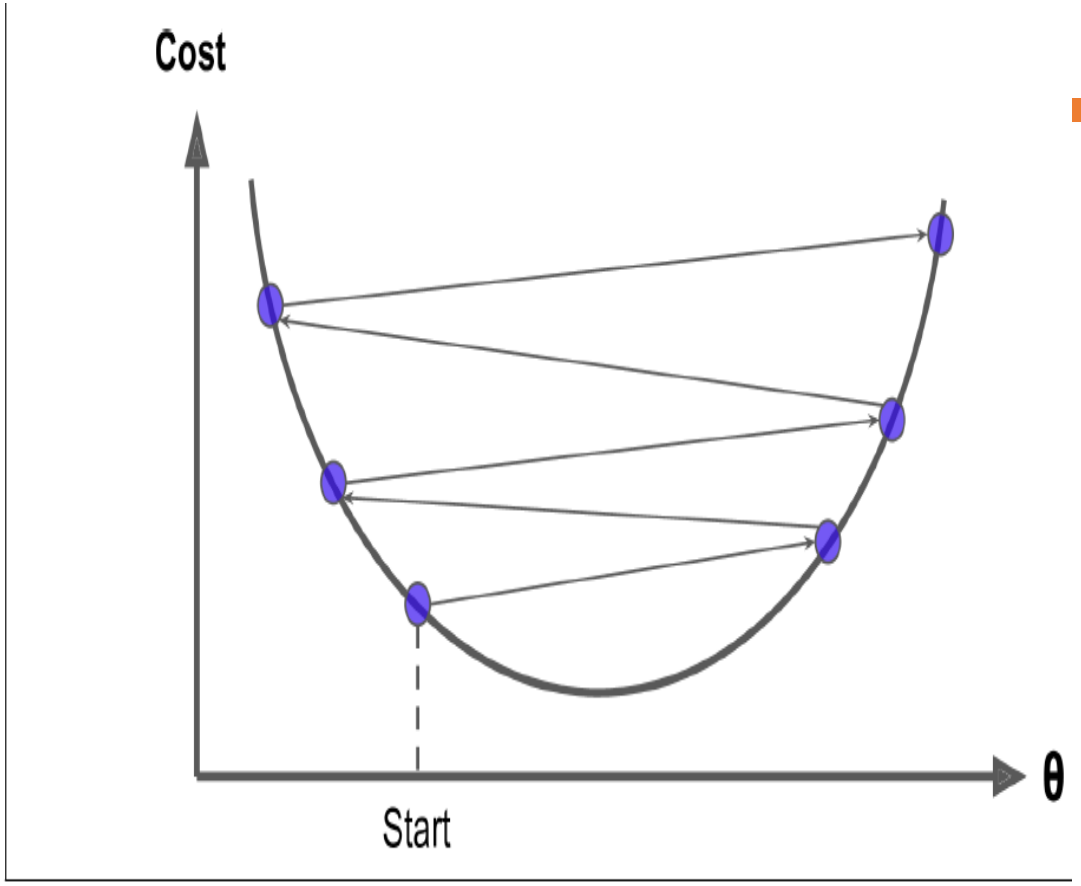


Figure 4-5. Learning rate too large

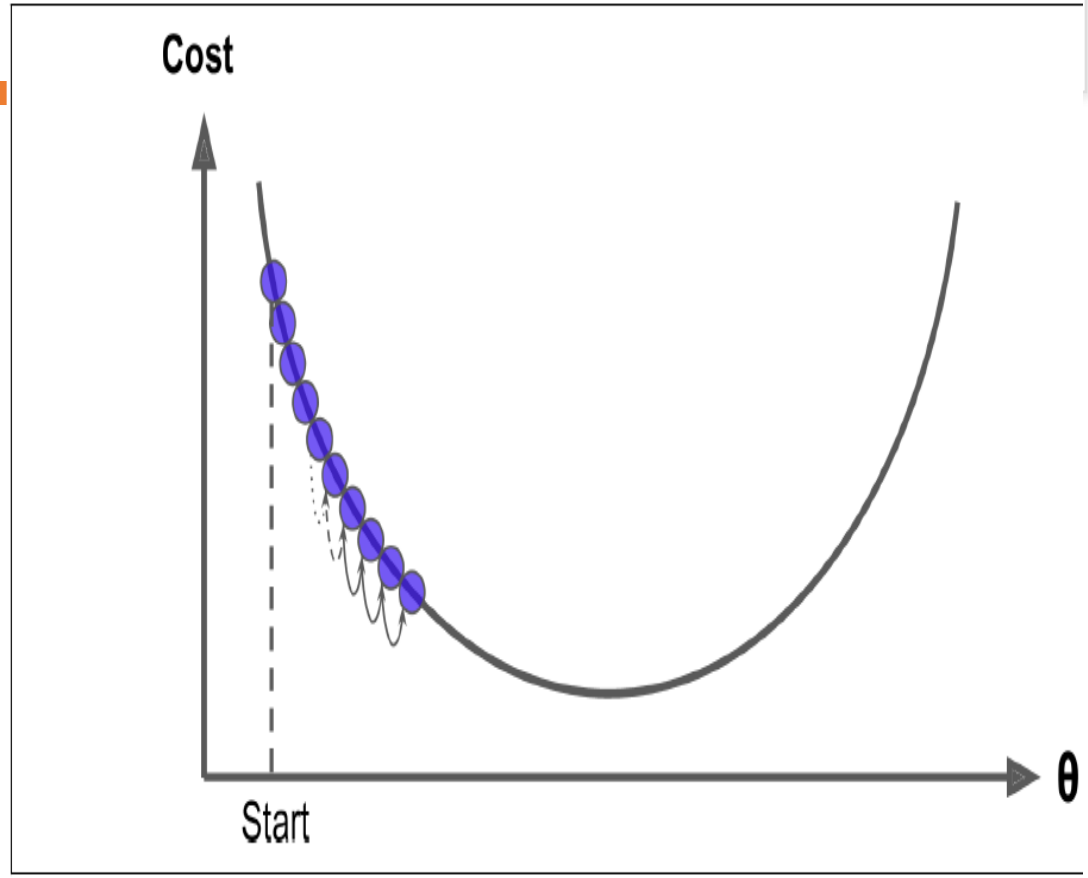
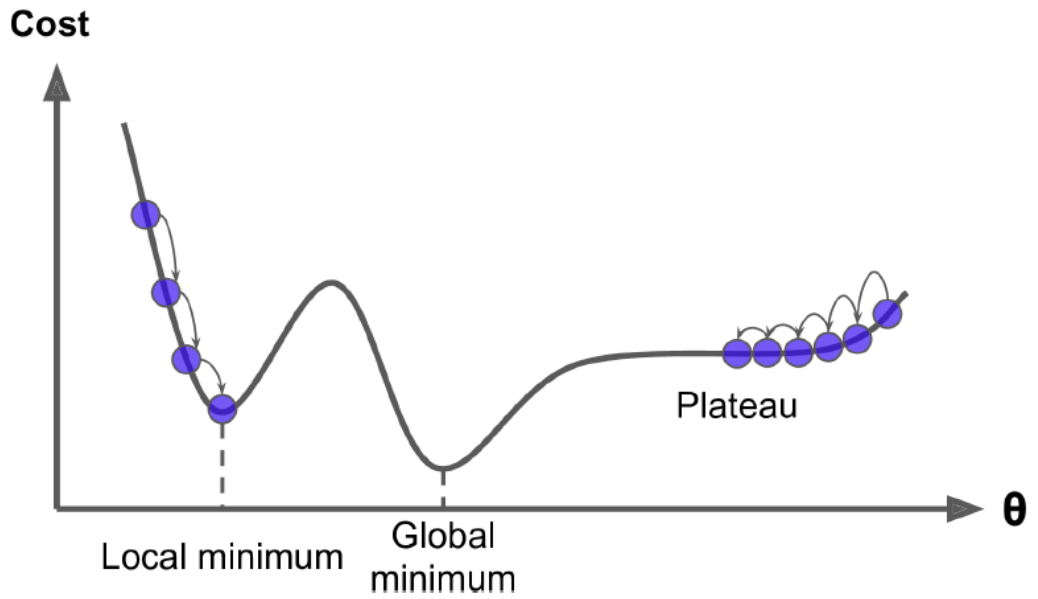


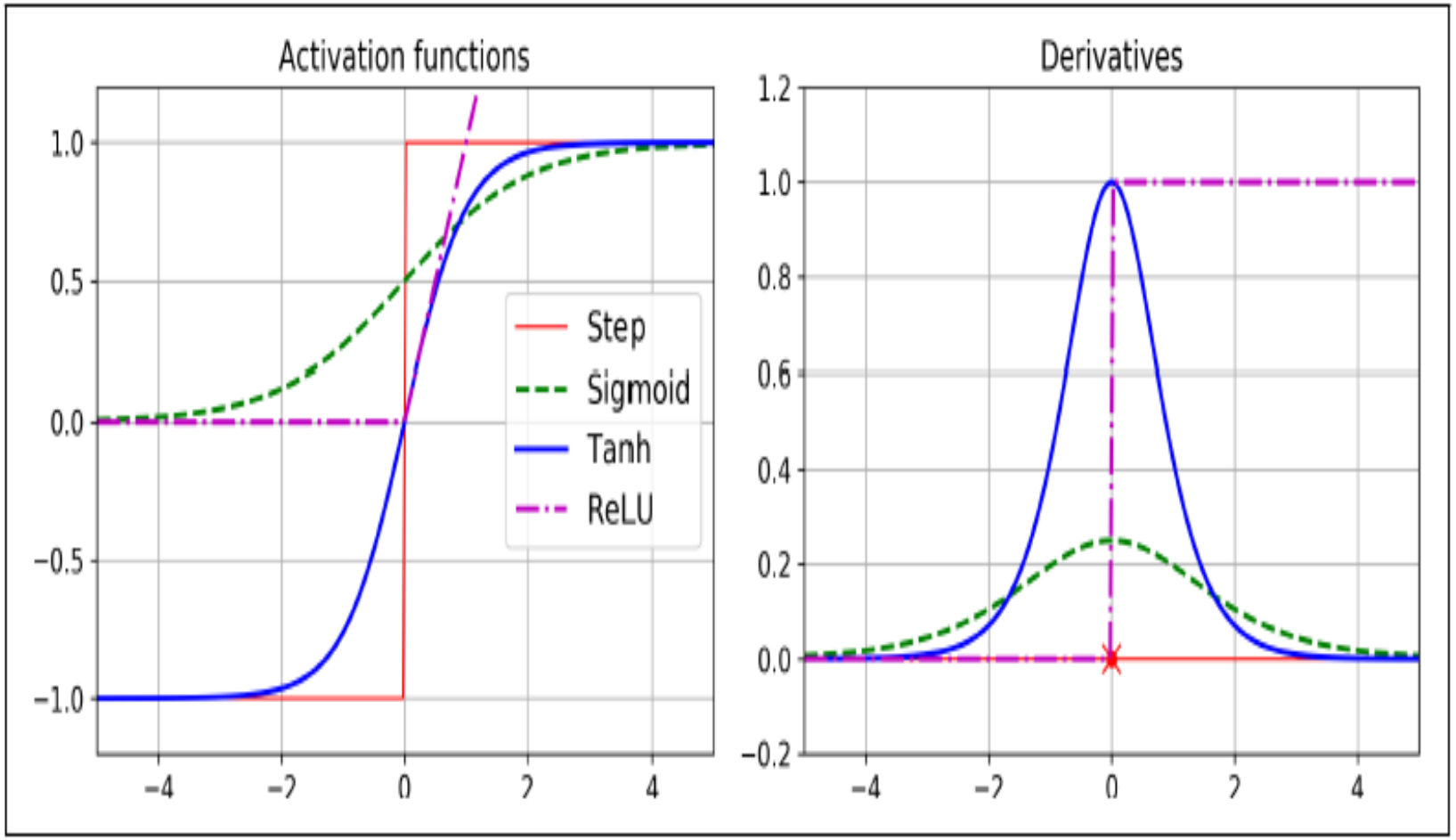
Figure 4-4. Learning rate too small

Gradient Descent Pitfalls

- If it starts on left can reach local minimum
- If it stars from right can hit plateau
- So pick Cost Functions which are convex functions
  - Has no local minimum
  - Continuous function with slope that does not change abruptly
- Then Gradient Descent will approach close to global minimum

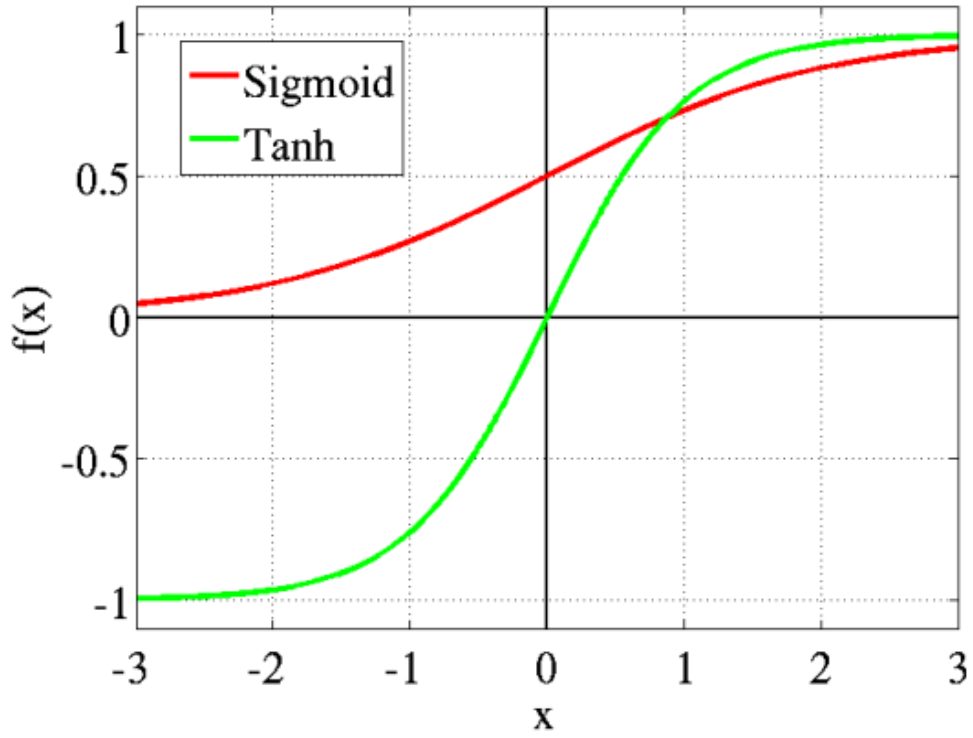


Activation Functions and their derivatives



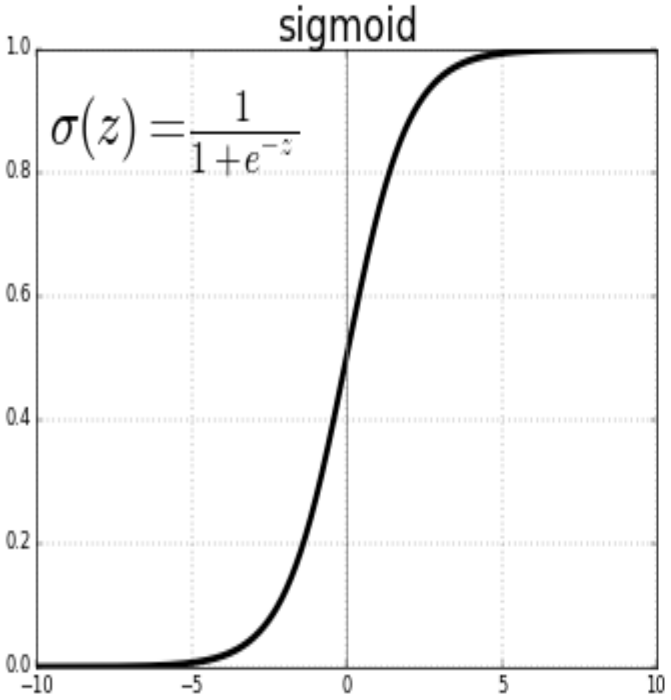
Aurelien Geron, "Hands-On Machine Learning with Scikit-Learn , Keras & Tensorflow, OReilly Publications

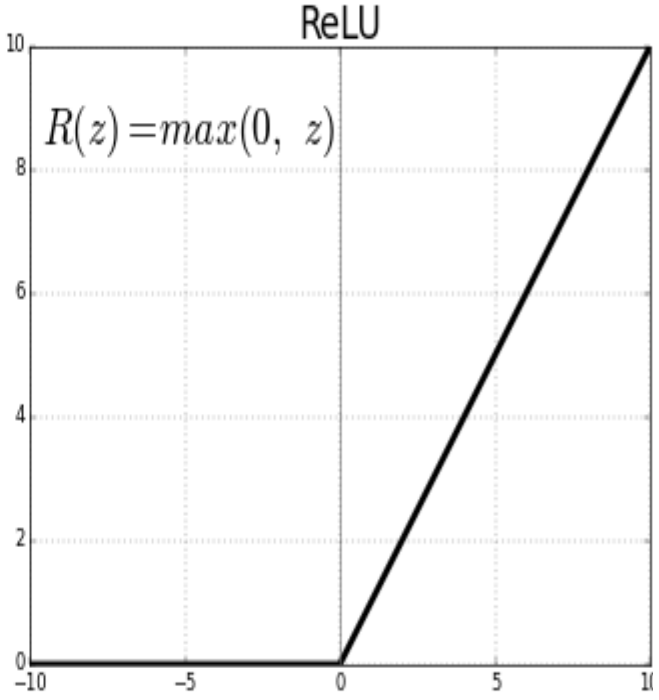
Activation Functions – Sigmoid vs Tanh

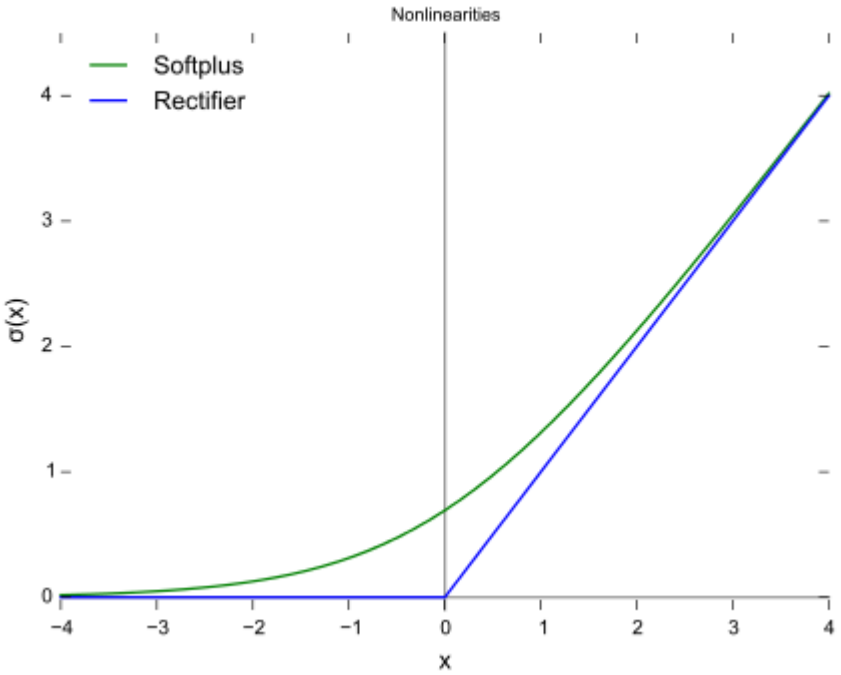


- **Sigmoid**
  - $s(x) = 1/(1 + e^{-x})$ where e ≈ 2.71 is the base of the natural logarithm
  - to **predict the probability**
  - between the range of **0 and 1,** sigmoid is the right choice.
  - is **differentiable**-, we can find the slope of the sigmoid curve at any two points.
  - The function is **monotonic** but function's derivative is not.
  - can cause a neural network to get stuck at the training time.
- **Tanh**
  - Range is from (-1 to 1)
  - Centering data – mean = 0
  - is also sigmoidal (s - shaped)
  - Advantage is that the -ve inputs will be mapped strongly negative
- Disadvantage – If x is very small or very large slope or gradient becomes 0 which slows down gradient descent

Activation Functions – Sigmoid vs ReLU

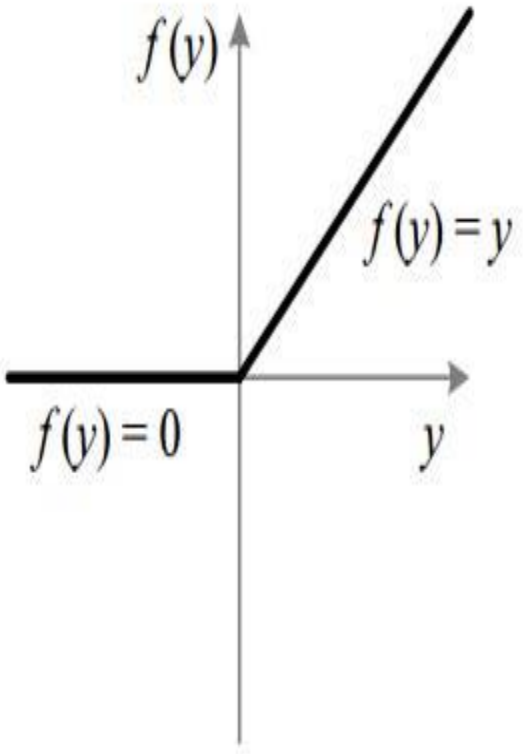


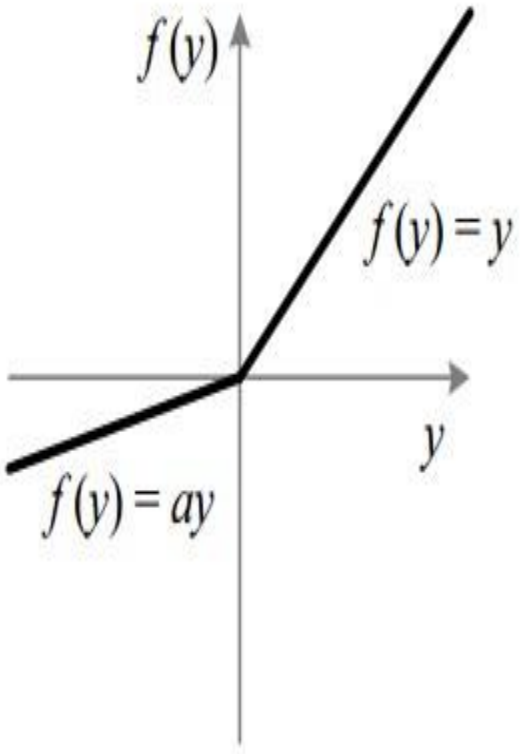




- ReLU is half rectified
- f(z) is 0 when z is < 0 and f(z) is equal to z when z is >= zero.
- Derivative =1 when z is +ve and 0 when z is 0-ve
- The function and its derivative **both are monotonic**
- **Alternate to ReLU is softplus activation function**
  - **Softplus(z) = log(1+exp(z), Close to 0 when z is –ve and close to z when z is +ve**

Activation Functions- ReLU vs Leaky ReLU





- The leak helps to increase the range of the ReLU function.
- Usually, the value of **a** is 0.01.
- When **a is not 0.01** then it is called **Randomized ReLU**.
- **range** of the Leaky ReLU is (-infinity to infinity)

Activation Functions SoftMax

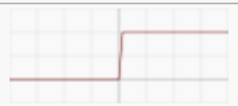
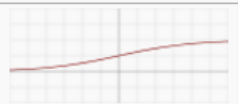
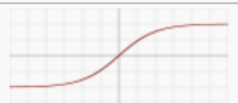
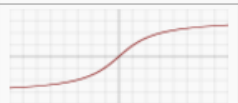
- *Usually last activation function*
- *to normalize the output of a network to a probability distribution over predicted output classes*

$$S(y)_i = \frac{\exp(y_i)}{\sum_{j=1}^{n} \exp(y_j)}$$

where,

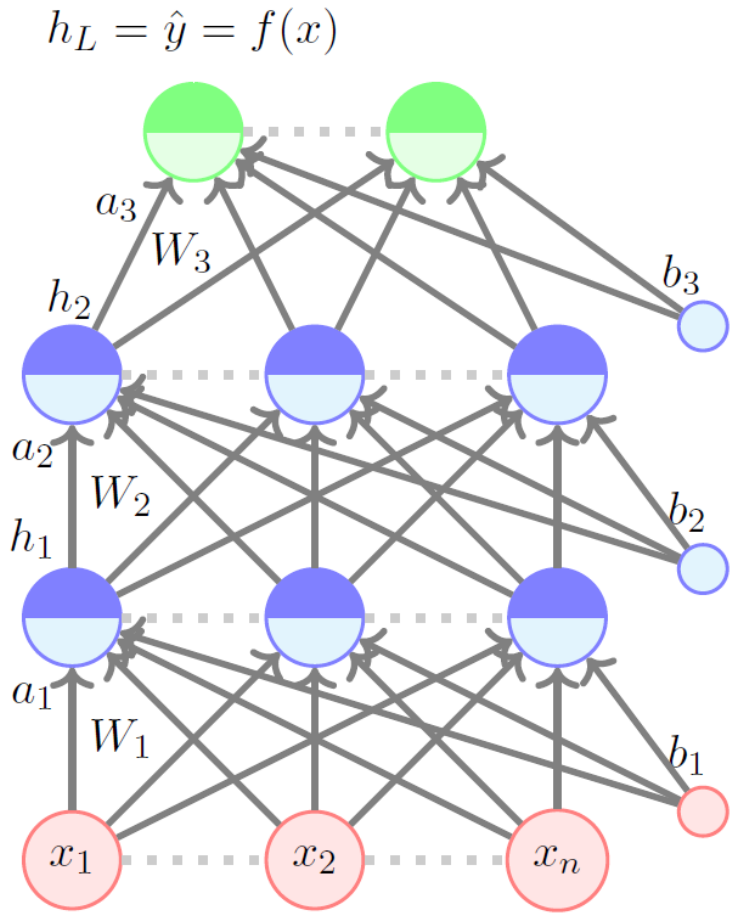
y	is an input vector to a softmax function, S. It consist of n elements for n classes (possible outcomes)
y_i	the i-th element of the input vector. It can take any value between -inf and +inf
$exp(y_i)$	standard exponential function applied on y_i. The result is a small value (close to 0 but never 0) if $y_i < 0$ and a large value if y_i is large. eg • $\exp(55) = 7.69e{+}23$ (A very large value) • $\exp(-55) = 1.30e$-24 (A very small value close to 0) **Note**: $\exp(*)$ is just $\mathbf{e}^*$ where $\mathbf{e} = 2.718$, the Euler's number.
$\sum_{j=1}^{n} \exp(y_j)$	A normalization term. It ensures that the values of output vector $S(y)_i$ sums to 1 for i-th class and each of them and each of them is in the range 0 and 1 which makes up a valid probability distribution.
n	Number of classes (possible outcomes)

Activation Function Cheat Sheet

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

https://patrickhoo.wixsite.com/diveindatascience/single-post/2019/06/13/activation-functions-and-when-to-use-them

Output Functions



- The pre-activation at layer i is given by

$$a_i(x) = b_i + W_i h_{i-1}(x)$$

- The activation at layer i is given by

$$h_i(x) = g(a_i(x))$$

where g is called the activation function (for example, logistic, tanh, linear, *etc.*)

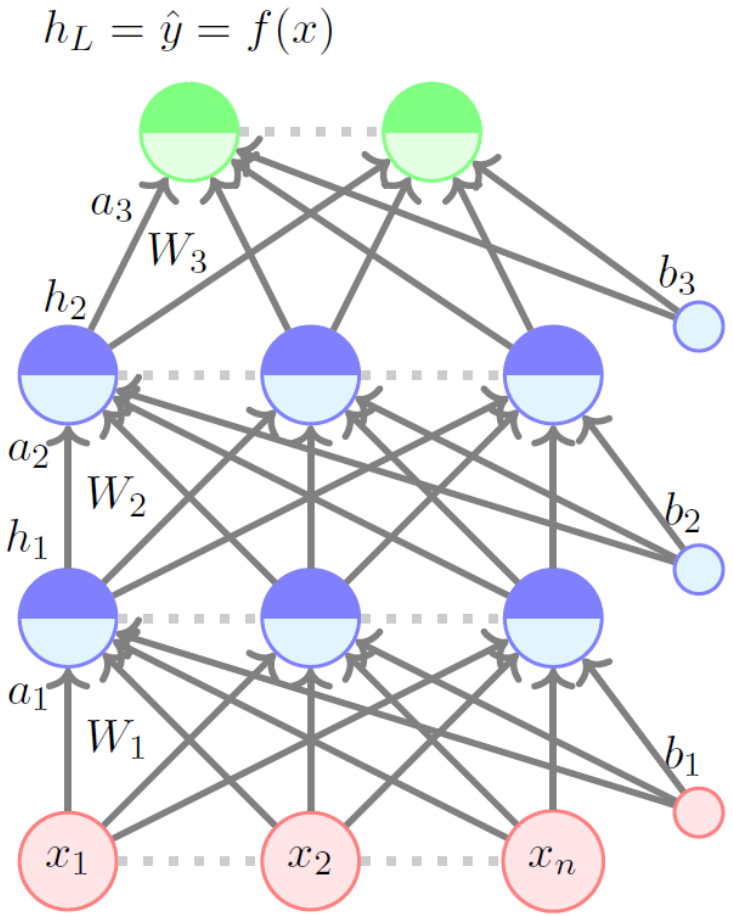
- The activation at the output layer is given by

$$f(x) = h_L(x) = O(a_L(x))$$

where O is the output activation function (for example, softmax, linear, *etc.*)

- To simplify notation we will refer to $a_i(x)$ as a_i and $h_i(x)$ as h_i

Regression problems - Output Function

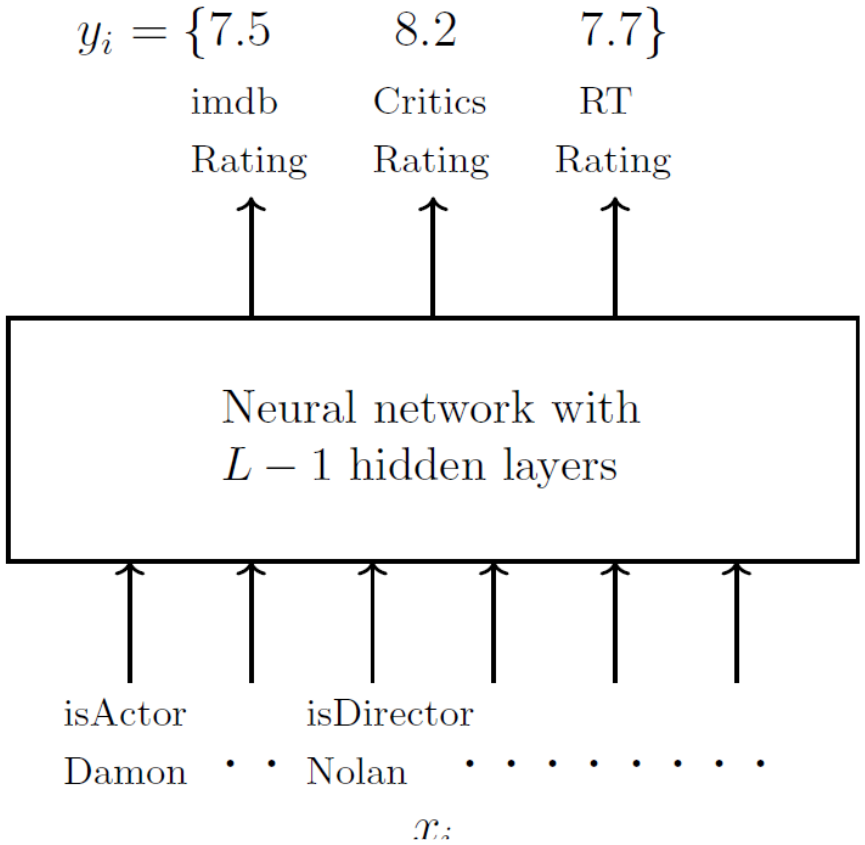


- A related question: What should the output function 'O' be if $y_i \in \mathbb{R}$?
- More specifically, can it be the logistic function?
- No, because it restricts $\hat{y}_i$ to a value between 0 & 1 but we want $\hat{y}_i \in \mathbb{R}$
- So, in such cases it makes sense to have 'O' as linear function

$$\begin{aligned} f(x) = h_L &= O(a_L) \\ &= W_O a_L + b_O \end{aligned}$$

- $\hat{y}_i = f(x_i)$ is no longer bounded between 0 and 1

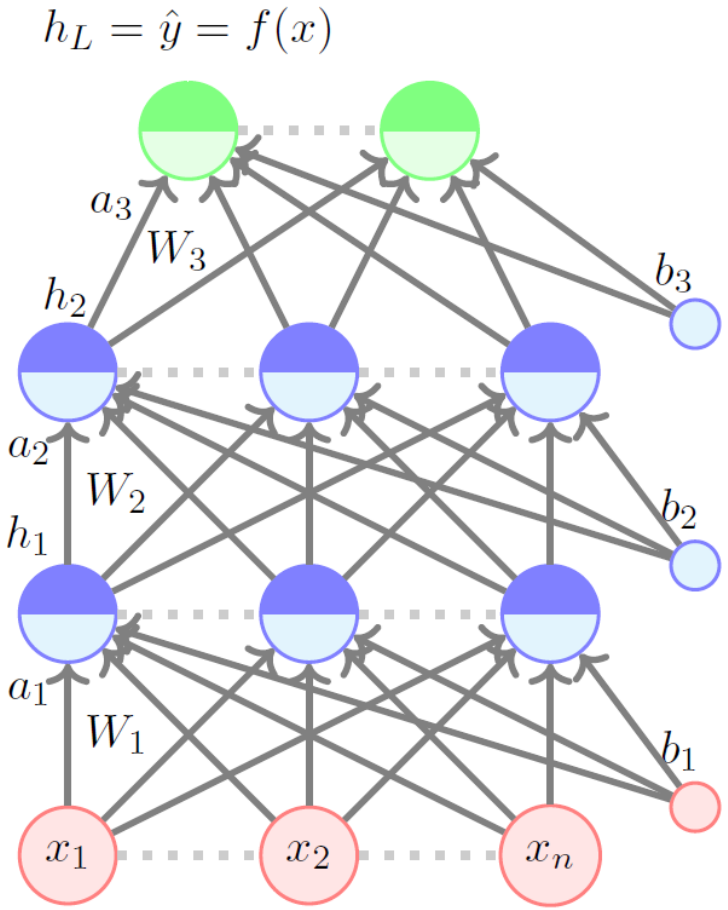
Regression problems - Loss Function



- The choice of loss function depends on the problem at hand
- We will illustrate this with the help of two examples
- Consider our movie example again but this time we are interested in predicting ratings
- Here $y_i \in \mathbb{R}^3$
- The loss function should capture how much $\hat{y}_i$ deviates from y_i
- If $y_i \in \mathbb{R}^n$ then the squared error loss can capture this deviation

$$\mathscr{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{3}(\hat{y}_{ij} - y_{ij})^2$$

Classification problem- Output Function



- Notice that y is a probability distribution
- Therefore we should also ensure that $\hat{y}$ is a probability distribution
- What choice of the output activation 'O' will ensure this ?

$$a_L = W_L h_{L-1} + b_L$$

$$\hat{y}_j = O(a_L)_j = \frac{e^{a_{L,j}}}{\sum_{i=1}^{k} e^{a_{L,i}}}$$

$O(a_L)_j$ is the j^{th} element of $\hat{y}$ and $a_{L,j}$ is the j^{th} element of the vector a_L.

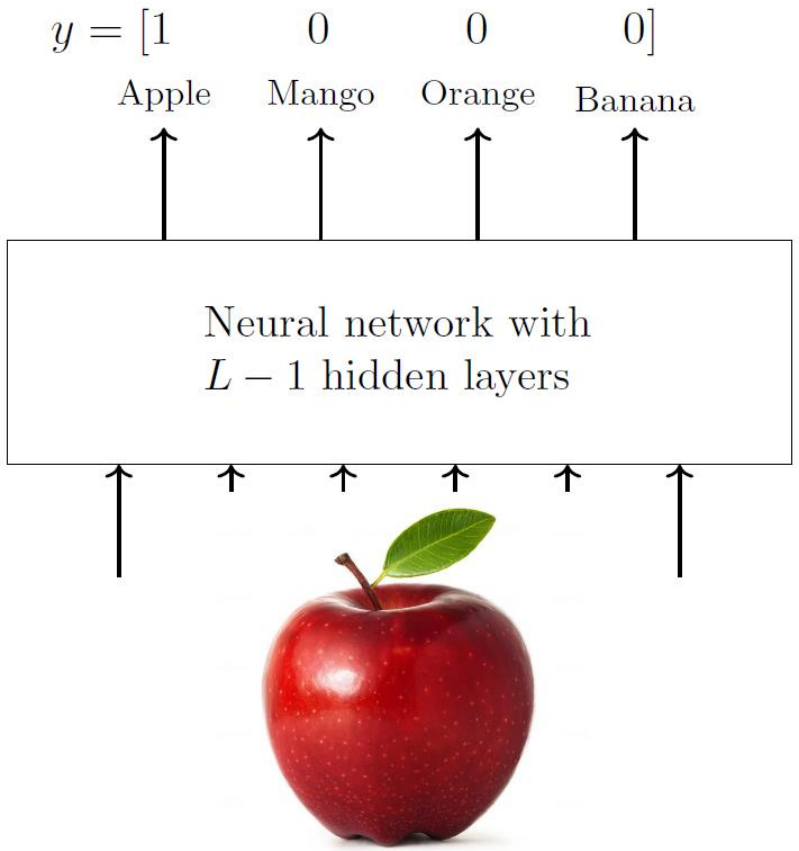
- This function is called the *softmax* function

What does $\hat{y}_\ell$ encode?

It is the probability that x belongs to the ℓ^{th} class (bring it as close to 1).

$\log \hat{y}_\ell$ is called the *log-likelihood* of the data.

Classification Problems- Loss Function



- Now that we have ensured that both y & $\hat{y}$ are probability distributions can you think of a function which captures the difference between them?
- Cross-entropy

$$\mathscr{L}(\theta) = -\sum_{c=1}^{k} y_c \log \hat{y}_c$$

- Notice that

$$\begin{aligned} y_c &= 1 \quad \text{if } c = \ell \text{ (the true class label)} \\ &= 0 \quad \text{otherwise} \\ \therefore \ \mathscr{L}(\theta) &= -\log \hat{y}_\ell \end{aligned}$$

So, for classification problem (where you have to choose 1 of K classes), we use the following objective function

$$\underset{\theta}{\text{minimize}} \quad \mathscr{L}(\theta) = -\log \hat{y}_\ell$$

or

$$\underset{\theta}{\text{maximize}} \quad -\mathscr{L}(\theta) = \log \hat{y}_\ell$$

Typical MLP architecture

Regression

Hyperparameter	Typical Value
# input neurons	One per input feature (e.g., 28 x 28 = 784 for MNIST)
# hidden layers	Depends on the problem. Typically 1 to 5.
# neurons per hidden layer	Depends on the problem. Typically 10 to 100.
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU, see Chapter 11)
Output activation	None or ReLU/Softplus (if positive outputs) or Logistic/Tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

Classification

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross-Entropy	Cross-Entropy	Cross-Entropy

Aurelien Geron, “Hands-On Machine Learning with Scikit-Learn , Keras & Tensorflow, OReilly Publications

Regression Loss Functions

- **Mean Squared Error (MSE)**
  - values with a large error are penalized.
  - is a convex function with a clearly defined global minimum
  - Can be used in **gradient descent optimization** to set the weight values
  - Very sensitive to outliers , will significantly increase the loss.
- **Mean Absolute Error (MAE)**
  - used in cases when the training data has a large number of outliers
    as the average distance approaches 0, gradient descent optimization will not work
- **Huber Loss**
  - Based on absolute difference between the actual and predicted value and threshold value, δ
  - Is quadratic when error is smaller than δ but linear when error is larger than δ

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

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

$$Huber\ Loss = \begin{cases} \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 & |y^{(i)} - \hat{y}^{(i)}| \le \delta \\ \frac{1}{n}\sum_{i=1}^{n}\delta(|y^{(i)} - \hat{y}^{(i)}| - \frac{1}{2}\delta) & |y^{(i)} - \hat{y}^{(i)}| > \delta \end{cases}$$

Classification Loss Functions

Cross entropy measures entropy between two probability distributions

- **Binary Cross-Entropy/Log Loss**
  - Compares the actual value (0 or 1) with the probability that the input aligns with that category
    - *p(i)* = probability that the category is 1
    - 1 — *p(i)* = probability that the category is 0)

$$CE\ Loss = \frac{1}{n}\sum_{i=1}^{N} -\left(y_i \cdot log(p_i) + (1 - y_i) \cdot log(1 - p_i)\right)$$

- **Categorical Cross-Entropy Loss**
  - In cases where the number of classes is greater than two

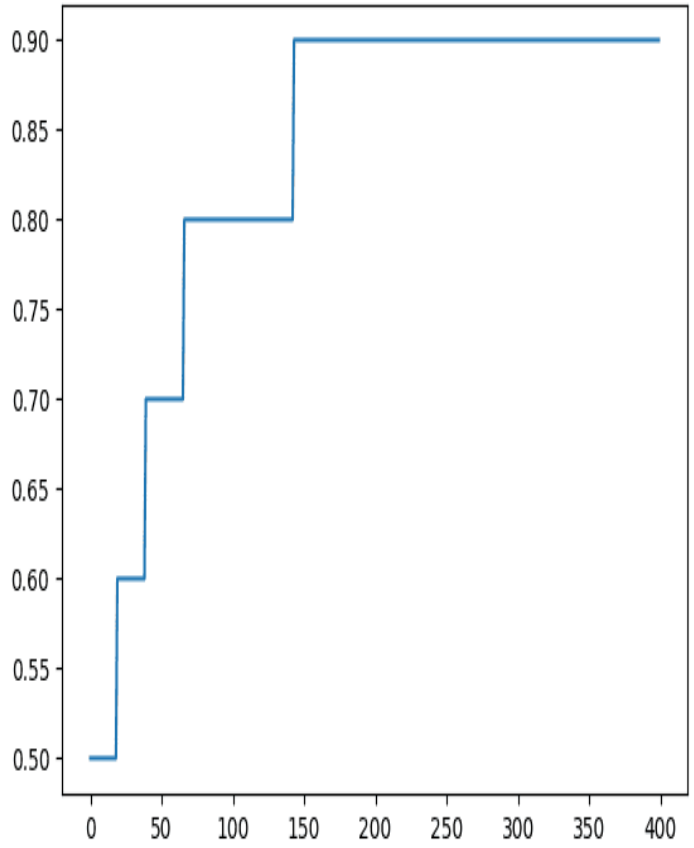
$$CE\ Loss = -\frac{1}{n}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij} \cdot log(p_{ij})$$

Keras Metrics

- model.compile(..., metrics=['mse'])
- Metric values are recorded at the end of each epoch on the training dataset.
- If a validation dataset is also provided, then is also calculated for the validation dataset.
- All metrics are reported in verbose output and in the history object returned from calling the fit() function.

Keras Metrics

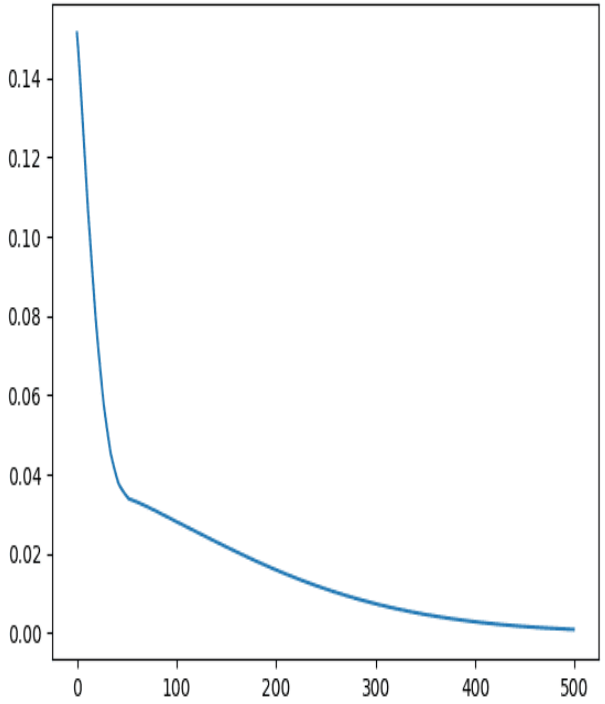
- **Accuracy metrics**
  - Accuracy
    - Calculates how often predictions equal labels.
  - Binary Accuracy
    - Calculates how often predictions match binary labels.
  - Categorical Accuracy
    - Calculates how often predictions match one-hot labels
  - Sparse Categorical Accuracy
    - Calculates how often predictions match integer labels.
  - TopK Categorical Accuracy
    - calculates the percentage of records for which the targets are in the top K predictions
      - rank the predictions in the descending order of probability values.
      - If the rank of the yPred is less than or equal to K, it is considered accurate.
  - Sparse TopK Categorical Accuracy class
    - Computes how often integer targets are in the top K predictions.



Aurelien Geron, "Hands-On Machine Learning with Scikit-Learn , Keras & Tensorflow, OReilly Publications

Keras Metrics

- **Regression metrics**
  - Mean Squared Error
    - Computes the mean squared error between y_true and y_pred
  - Root Mean Squared Error
    - Computes root mean SE metric between y_true and y_pred
  - Mean Absolute Error
    - Computes the mean absolute error between the labels and predictions
  - Mean Absolute Percentage Error
    - **MAPE** = (1/n) * Σ(|actual – prediction| / |actual|) * 100
    - Average difference between the predicted and the actual in %
  - Mean Squared Logarithmic Error
    - measure of the ratio between the true and predicted values.



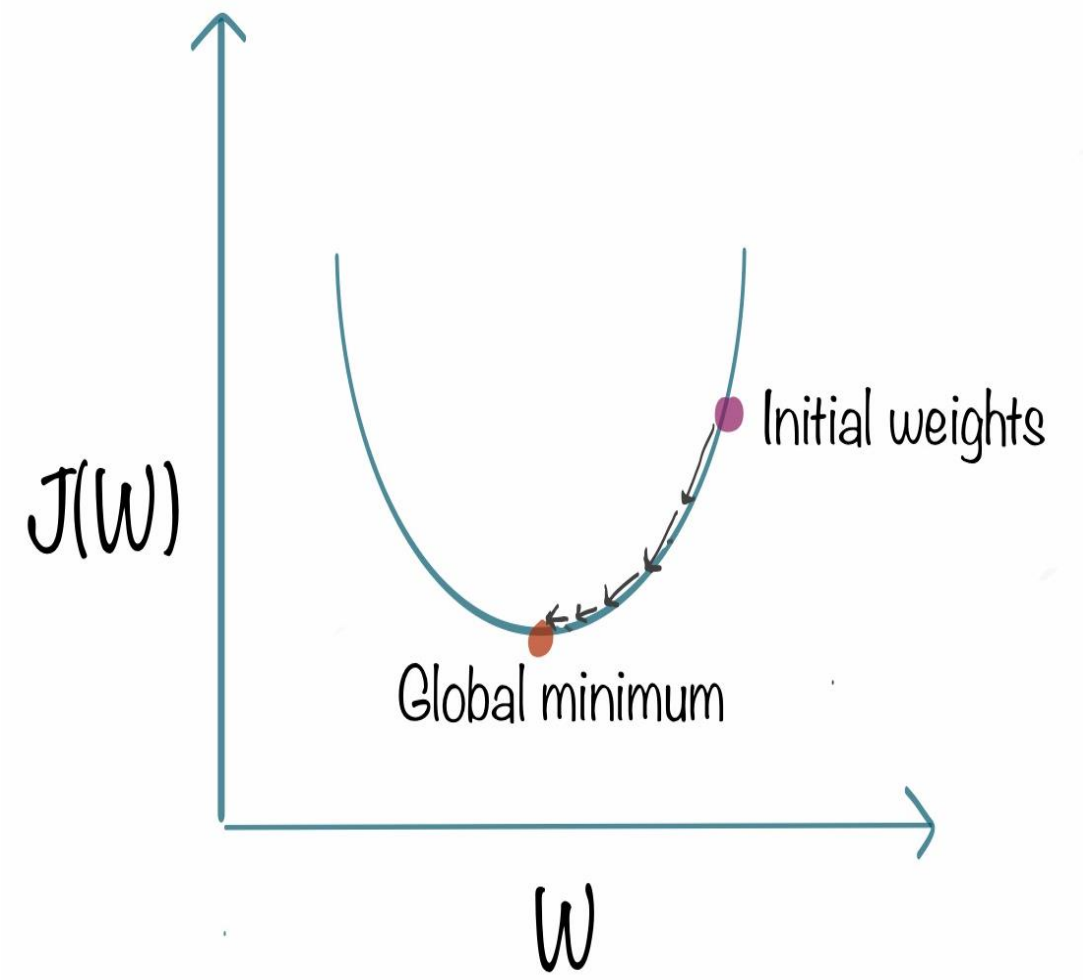
Keras Metrics

- AUC
- Precision
- Recall
- TruePositives
- TrueNegatives
- FalsePositives
- FalseNegatives
- PrecisionAtRecall
  - Computes best precision where recall is >= specified value
- SensitivityAtSpecificity
  - Computes best sensitivity where specificity is >= specified value
- SpecificityAtSensitivity
  - Computes best specificity where sensitivity is >= specified value

- **Probabilistic metrics**
  - Binary Crossentropy
  - Categorical Crossentropy
  - Sparse Categorical Crossentropy
  - KLDivergence
    - a measure of how two probability distributions are different from each other
  - Poisson
    - if dataset comes from a Poisson distribution

Gradient based learning

- **Gradient Based Learning**
  - seeks to change the weights so that the next evaluation reduces the error
  - is navigating down the gradient (or slope) of error
  - The process repeats until the global minimum is reached.
  - works well for convex functions
  - It is expensive to calculate the gradients if the size of the data is huge.



The Vanishing/Exploding Gradient Problems

- Algorithm computes the gradient of the cost function with regard to each parameter in the network
- **Problems include**
- **Vanishing Gradients**
  - Gradients become very small as algorithm progresses down to lower layers
  - So connection weights remain unchanged and training never converges to a solution
- **Exploding Gradients**
  - Gradients become so large until layers get huge weight updates and algorithm diverges
- Deep Neural Networks suffer from unstable gradients, different layers may learn at different speeds.
- **Reasons for unstable gradients** Glorot & Bengio (2010)
  - Because of combination of Sigmoid activation and weight initialization (normal distribution with mean 0 and std dev 1)
  - Variance of outputs of each layer is much greater than the variance of its inputs
  - Variance goes on increasing after each layer
  - until the activation function saturates(0 or 1, with derivative close to 0) at the top layers

Solutions include : 1. Weight Initialization

- ***The variance of the outputs of each layer has to be equal to the variance of its inputs***
- ***Xavier's Initialization***

$$W^{[l]} \sim \mathcal{N}\left(\mu = 0, \sigma^2 = \frac{1}{n^{[l-1]}}\right)$$

$$b^{[l]} = 0$$

- Weight Matrix ***W*** of a particular layer ***l***
- picked randomly from a *normal distribution* with
  - ***mean μ= 0***
  - ***variance sigma² = multiplicative inverse of the number of neurons in layer l–1***.
- The bias ***b*** of all layers is initialized with 0

Solutions Include : Weight Initialization

1. **Xavier's or Glorot Initialization**

Variance of outputs of each layer to be equal to the variance of its inputs
Gradients to have equal variance before and after flowing through a layer in the reverse direction

Fan-in – Number of inputs
Fan-out- Number of neurons

$$fan_{avg} = (fan_{in} + fan_{out})/2.$$

Table 11-1. Initialization parameters for each type of activation function

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

Solutions include

2. Using Non-saturating Activation functions

1. ReLU does not saturate for +ve values
2. Suffer from problem of dying ReLU- neurons output only 0
   - Weighted sum of its inputs are negative for all instances in training set
3. Use Leaky ReLU instead – (only go into coma don't die, may wake up)
   1. $\alpha = 0.01$, sometimes 0.2
4. Flavors include

   Randomized leaky ReLU(RReLU)

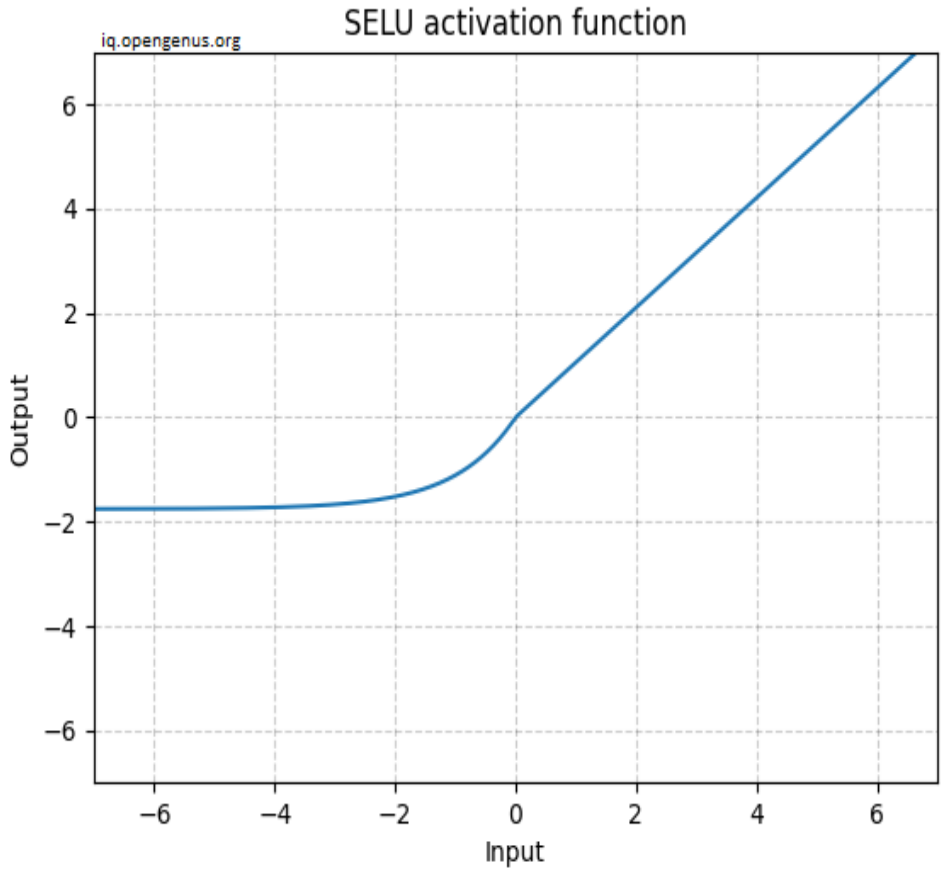
   α is picked randomly in a given range during training and is fixed to an average value during testing

   Parametric Leaky Relu (PReLU)

   α is authorised to be learned during training as a parameter

SELU

- **Scaled Exponential Linear Units**
  - Induce self-normalization
  - the output of each layer will tend to preserve mean 0 and standard deviation 1 during training
  - $f(x) = \lambda x$ if $x >= 0$
  - $f(x) = \lambda\alpha(\exp(x)-1)$ if $x < 0$
  - $\alpha = 1.6733$, $\lambda = 1.0507$
- conditions for self-normalization to happen:
  - The input features must be standardized (mean 0 and standard deviation 1).
  - Every hidden layer's weights must also be initialized using normal initialization.
  - The network's architecture must be sequential.



Solutions Include

3**. Batch Normalization -Ioffe and Szegedy** (2015)

- designed to solve the vanishing/exploding gradients problems, is also a good regularizer
- BN layer performs the standardizing and normalizing operations on the input of a layer coming from a previous layer.
- Normalization
  - brings the numerical data to a common scale without distorting its shape.
  - (mean = 0, std dev= 1)
- BN adds extra operations in the model , before activation
  - Operation zero centres and normalizes each input
  - Then scale and shift the result using two new parameter vectors per layer
  - Each BN Layer learn 4 parameter vectors
    - Output Scale Vector
    - Output offset vector
    - Input mean vector
    - Input standard deviation
- To zero-centre and normalize the inputs , mean and standard deviation of input needs to be computed
- Current mini-batch is used to evaluate mean and standard deviation

Solutions Include

3. Batch Normalization -Ioffe and Szegedy (2015)

1. $$\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$$

2. $$\sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} \left(\mathbf{x}^{(i)} - \mu_B\right)^2$$

3. $$\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

4. $$\mathbf{z}^{(i)} = \gamma \otimes \hat{\mathbf{x}}^{(i)} + \beta$$

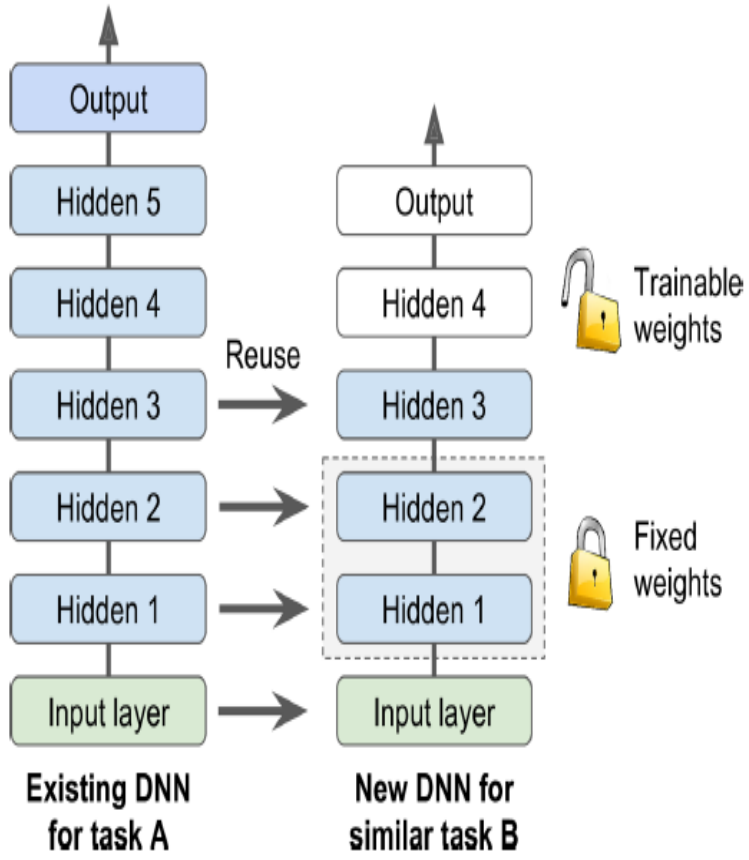
μ_B is the vector of input means, evaluated over the whole mini-batch B (it contains one mean per input).

- σ_B is the vector of input standard deviations, also evaluated over the whole mini-batch (it contains one standard deviation per input).
- m_B is the number of instances in the mini-batch.
- $\hat{\mathbf{x}}^{(i)}$ is the vector of zero-centered and normalized inputs for instance i.
- γ is the output scale parameter vector for the layer (it contains one scale parameter per input).
- $\otimes$ represents element-wise multiplication (each input is multiplied by its corresponding output scale parameter).
- β is the output shift (offset) parameter vector for the layer (it contains one offset parameter per input). Each input is offset by its corresponding shift parameter.
- ϵ is a tiny number to avoid division by zero (typically 10^{-5}). This is called a *smoothing term*.
- $\mathbf{z}^{(i)}$ is the output of the BN operation: it is a rescaled and shifted version of the inputs.

Aurelien Geron, "Hands-On Machine Learning with Scikit-Learn , Keras & Tensorflow, OReilly Publications

Solutions include

4. Gradient Clipping
   - Clip gradient during back propagation so that they never exceed some threshold
   - All the partial derivatives of the loss will be clipped between -0.1 to 0.1
   - Threshold can also be a hyperparameter to tune
5. Reusing Pretrained Layers
   - Transfer Learning



6. Faster Optimizers

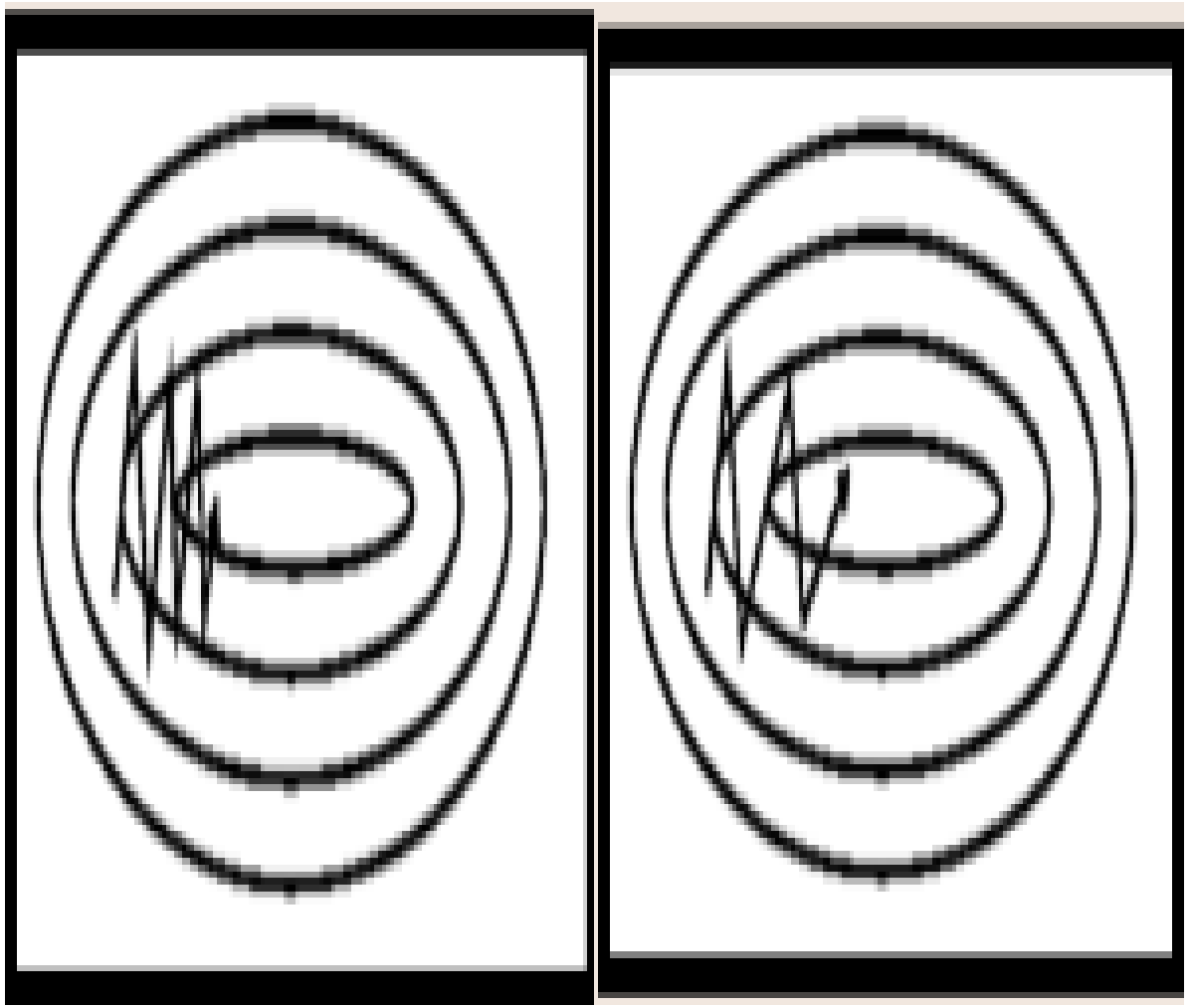
- Optimizer
  - is a function or an algorithm that modifies the attributes of the neural network, such as weights and learning rate.
- Terminology
  - Weights/ Bias – The learnable parameters in a model that controls the signal between two neurons.
  - Epoch – The number of times the algorithm runs on the whole training dataset.
  - Sample – A single row of a dataset.
  - Batch –denotes the number of samples to be taken for updating the model parameters.
  - Learning rate –defines a scale of how much model weights should be updated.
  - Cost Function/Loss Function -is used to calculate the cost that is the difference between the predicted value and the actual value.

Mini Batch Gradient Descent Deep Learning Optimizer

- **Batch gradient descent**:
  - gradient is average of gradients computed from ALL the samples in dataset
- Mini Batch GD:
  - subset of the dataset is used for calculating the loss function, therefore fewer iterations are needed.
  - batch size of 32 is considered to be appropriate for almost every case.
  - Yann Lecun (2018) – “Friends don’t let friends use mini batches larger than 32”
- is faster , more efficient and robust than the earlier variants of gradient descent.
- the cost function is noisier than the batch GD but smoother than SDG.
- Provides a good balance between speed and accuracy.
- It needs a hyperparameter that is “mini-batch-size”, which needs to be tuned to achieve the required accuracy.

Stochastic GD & SGD with Momentum DL

- stochastic means randomness on which the algorithm is based upon.
- Instead of taking the whole dataset for each iteration, randomly select the batches of data
- The path taken is full of noise as compared to the gradient descent algorithm.
- Uses a higher number of iterations to reach the local minima, thereby the overall computation time increases.
- The computation cost is still less than that of the gradient descent optimizer.
- If the data is enormous and computational time is an essential factor, SGD should be preferred over batch gradient descent algorithm.
- **Stochastic Gradient Descent with Momentum Deep Learning Optimizer**
  - momentum helps in faster convergence of the loss function.



SGD with Momentum Optimizer

- GD takes small , regular steps down the slope so algorithm takes more time to reach the bottom
- adding a fraction of the previous update to the current update will make the process a bit faster.
- Hyperparameter β - Momentum
  - To simulate a friction mechanism and prevent momentum from becoming too large
- Also rolls past local minima
- learning rate should be decreased with a high momentum term.

$$1. \quad \mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\theta}J(\theta)$$

$$2. \quad \theta \leftarrow \theta + \mathbf{m}$$

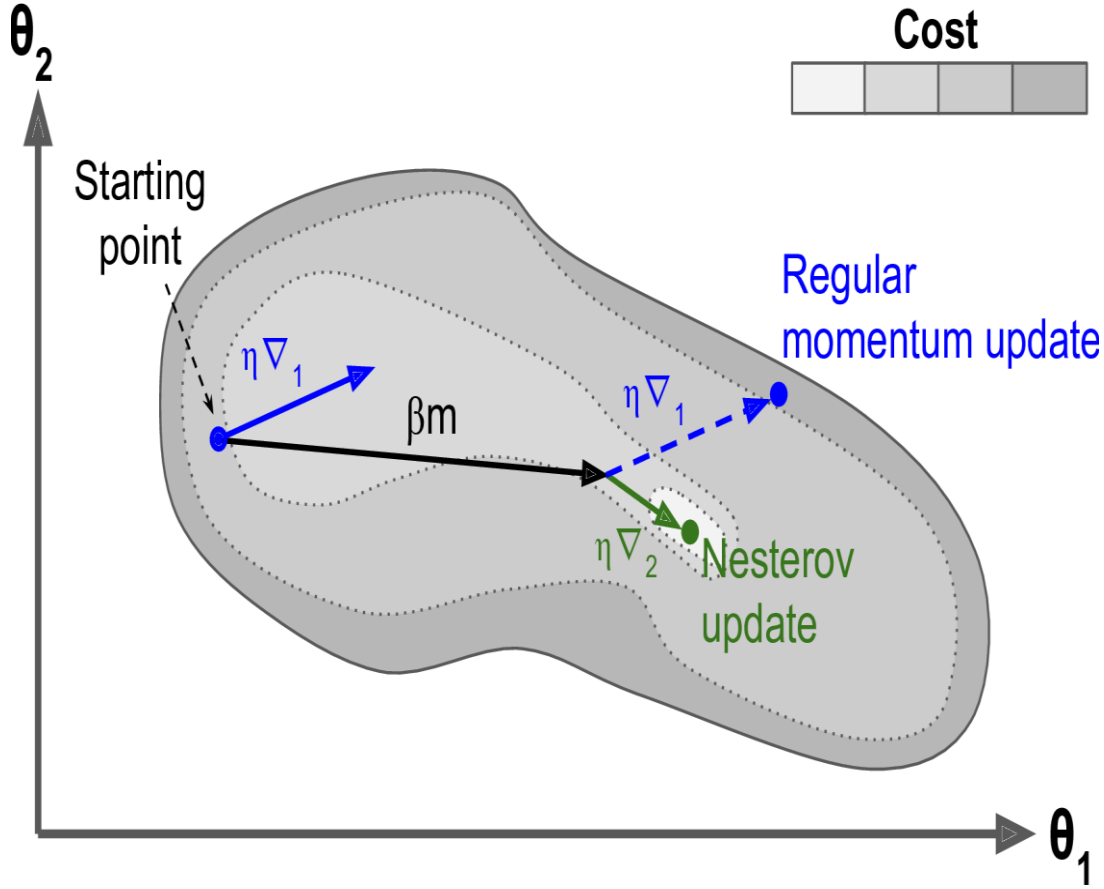
- β - Momentum
- set between 0 (high friction) and 1 (low friction)
- Typically 0.9

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)
```

Aurelien Geron, "Hands-On Machine Learning with Scikit-Learn , Keras & Tensorflow, OReilly Publications

SGD with Nesterov Momentum Optimization

- Yurii Nesterov in 1983
- to measure the gradient of the cost function not at the local position but slightly ahead in the direction of the momentum
- the momentum vector will be pointing in the right direction (i.e., toward the optimum)
- it will be slightly more accurate to use the gradient measured a bit farther in that direction rather than using the gradient at the original position



1. $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\theta}J(\theta + \beta\mathbf{m})$
2. $\theta \leftarrow \theta + \mathbf{m}$

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
```

Aurelien Geron, "Hands-On Machine Learning with Scikit-Learn , Keras & Tensorflow, OReilly Publications

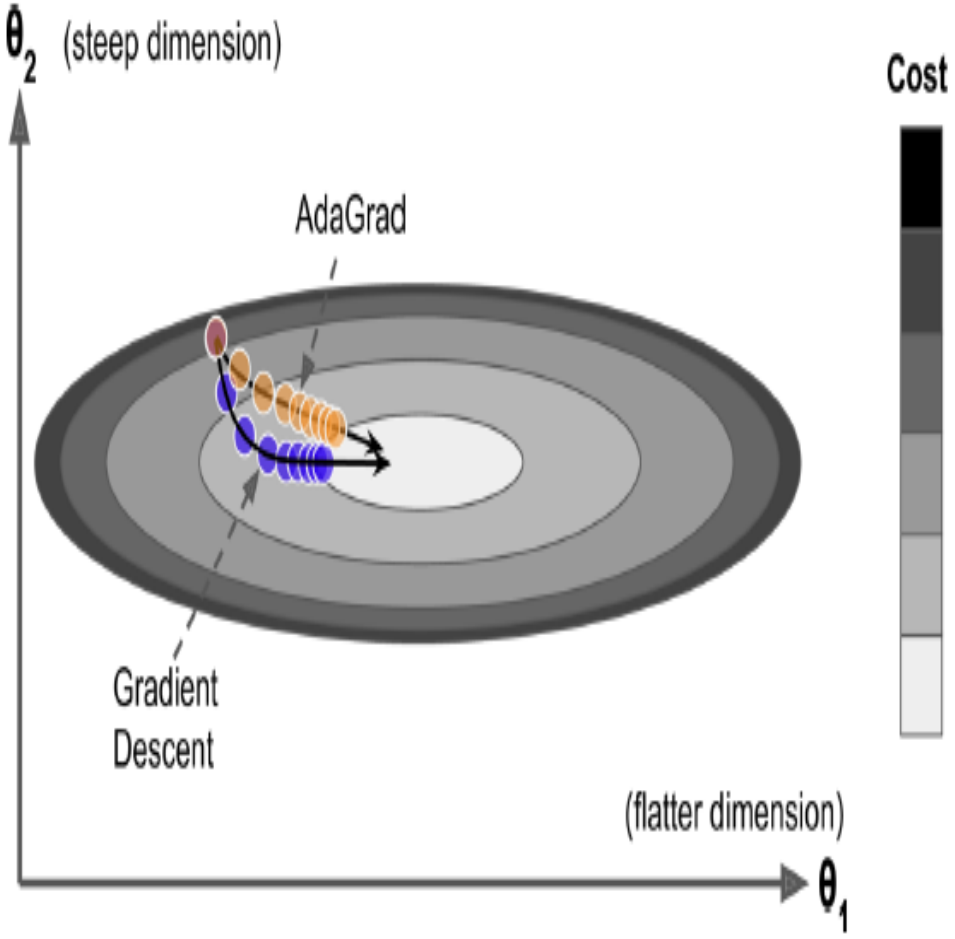
Adagrad (Adaptive Gradient Descent) Deep Learning Optimizer

- Adaptive Learning Rate
- Scaling down the gradient vector along the steepest dimension
- If the cost function is steep along the ith dimension, then s will get larger and larger at each iteration
- No need to modify the learning rate manually
- more reliable than gradient descent algorithms, and it reaches convergence at a higher speed.

$$\mathbf{s} \leftarrow \mathbf{s} + \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$$

$$\theta \leftarrow \theta - \eta \nabla_\theta J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$$

- Disadvantage
  - it decreases the learning rate aggressively and monotonically.
  - Due to small learning rates, the model eventually becomes unable to acquire more knowledge, and hence the accuracy of the model is compromised.



RMS Prop(Root Mean Square) Deep Learning Optimizer

- The problem with the gradients some are small while others may be huge
- Defining a single learning rate might not be the best idea.
- accumulating only the gradients from the most recent iterations (as opposed to all the gradients since the beginning of training).

$$\mathbf{s} \leftarrow \beta \mathbf{s} + (1 - \beta) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$$

$$\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$$

- β= 0.9

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

- Works better than Adagrad , was popular until ADAM

Adam Deep Learning Optimizer

- is derived from adaptive moment estimation.
- inherit the features of both Adagrad and RMS prop algorithms.
  - like Momentum optimization keeps track of an exponentially decaying average of past gradients,
  - like RMSProp it keeps track of an exponentially decaying average of past squared gradients
  - β1 and β2 represent the decay rate of the average of the gradients.
- Advantages
  - Is straightforward to implement
  - faster running time
  - low memory requirements, and requires less tuning
- Disadvantages
  - Focusses on faster computation time, whereas SGD focus on data points.
  - Therefore SGD generalize the data in a better manner at the cost of low computation speed.

$$\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

$$\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

$$\widehat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^{\,t}}$$

$$\widehat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^{\,t}}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta\, \widehat{\mathbf{m}} \oslash \sqrt{\widehat{\mathbf{s}} + \epsilon}$$

t represents iteration

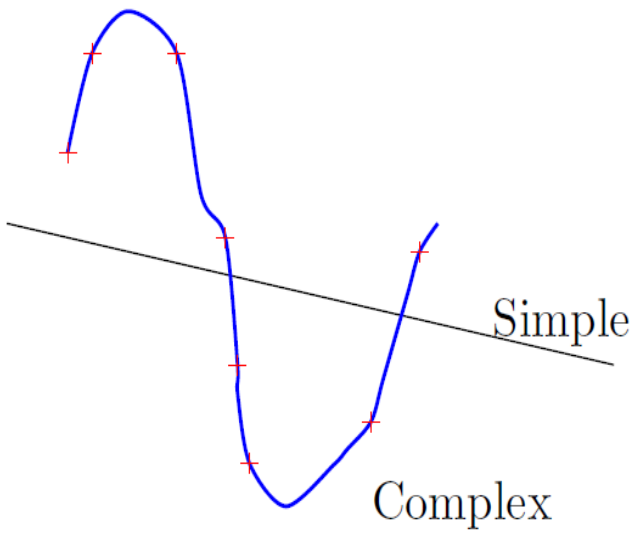
β1= 0.9

β 2=0.999

Smoothing term ϵ is=10^{-7}

```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

Curve Fitting – True Function is Sinusoidal



The points were drawn from a sinusoidal function (the true $f(x)$)

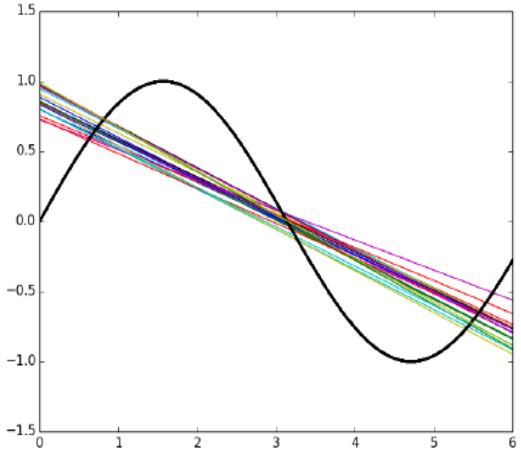
- Let us consider the problem of fitting a curve through a given set of points
- We consider two models :

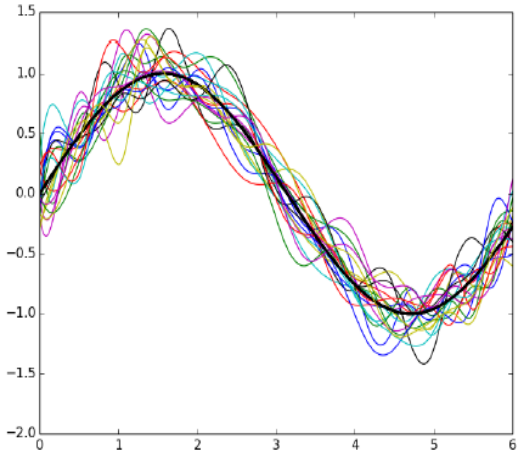
$$\underset{(degree{:}1)}{Simple} \quad y = \hat{f}(x) = w_1 x + w_0$$

$$\underset{(degree{:}25)}{Complex} \quad y = \hat{f}(x) = \sum_{i=1}^{25} w_i x^i + w_0$$

- Note that in both cases we are making an assumption about how y is related to x. We have no idea about the true relation $f(x)$

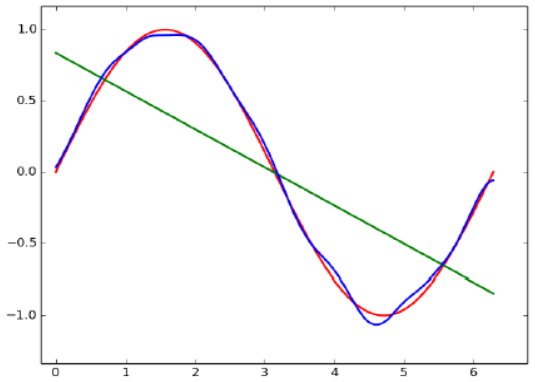
Curve Fitting – True Function is Sinusoidal





- Simple models trained on different samples of the data do not differ much from each other
- However they are very far from the true sinusoidal curve (under fitting)
- On the other hand, complex models trained on different samples of the data are very different from each other (high variance)

Bias



Green Line: Average value of $\hat{f}(x)$ for the simple model
Blue Curve: Average value of $\hat{f}(x)$ for the complex model
Red Curve: True model ($f(x)$)

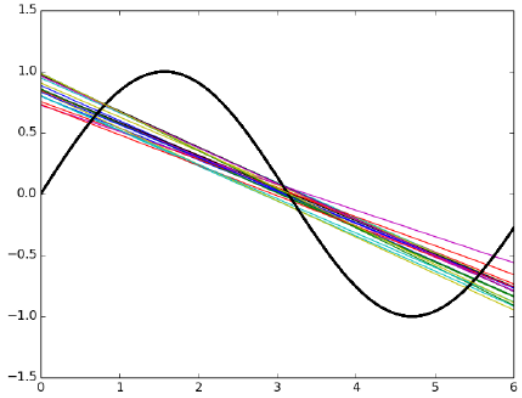
- Let $f(x)$ be the true model (sinusoidal in this case) and $\hat{f}(x)$ be our estimate of the model (simple or complex, in this case) then,

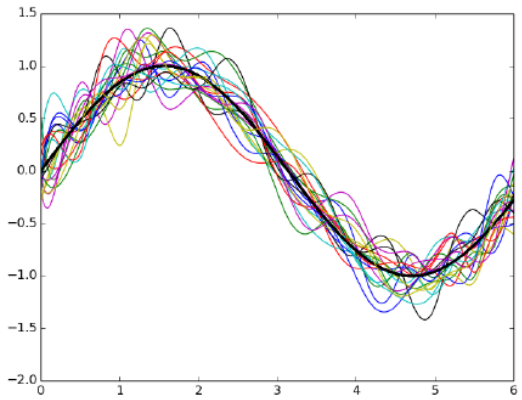
$$\text{Bias } (\hat{f}(x)) = E[\hat{f}(x)] - f(x)$$

- $E[\hat{f}(x)]$ is the average (or expected) value of the model
- We can see that for the simple model the average value (green line) is very far from the true value $f(x)$ (sinusoidal function)
- Mathematically, this means that the simple model has a high bias
- On the other hand, the complex model has a low bias

8

Variance





- We now define,

$$\text{Variance } (\hat{f}(x)) = E[(\hat{f}(x) - E[\hat{f}(x)])^2]$$

(Standard definition from statistics)

- Roughly speaking it tells us how much the different $\hat{f}(x)$'s (trained on different samples of the data) differ from each other

- It is clear that the simple model has a low variance whereas the complex model has a high variance

Mean Square Error

- We can show that

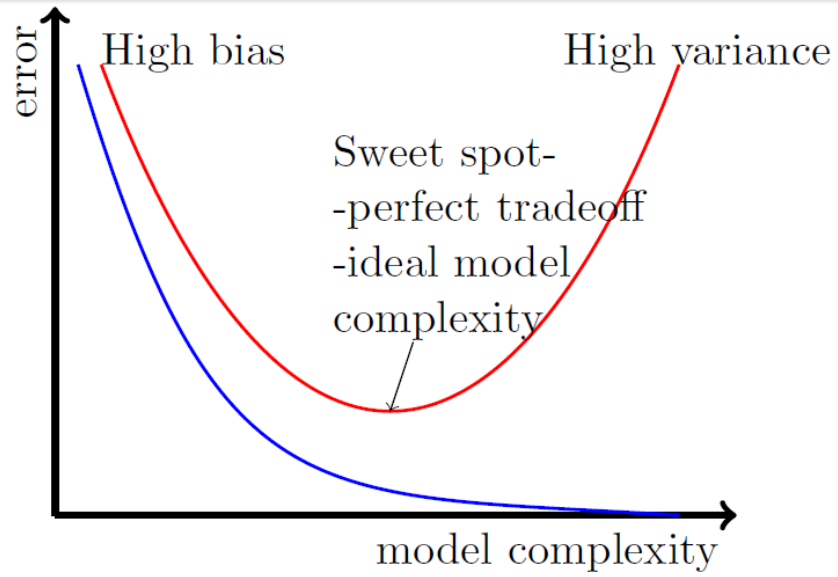
$$E[(y - \hat{f}(x))^2] = Bias^2 + Variance + \sigma^2 \text{ (irreducible error)}$$

- Consider a new point (x, y) which was not seen during training
- If we use the model $\hat{f}(x)$ to predict the value of y then the mean square error is given by

$$E[(y - \hat{f}(x))^2]$$

(average square error in predicting y for many such unseen points)

Train vs Test Error



- The parameters of $\hat{f}(x)$ (all w_i's) are trained using a training set $\{(x_i, y_i)\}_{i=1}^{n}$
- However, at test time we are interested in evaluating the model on a validation (unseen) set which was not used for training
- This gives rise to the following two entities of interest:
  $train_{err}$ (say, mean square error)
  $test_{err}$ (say, mean square error)
- Typically these errors exhibit the trend shown in the adjacent figure

Train vs Test Error

- Let there be n training points and m test (validation) points

$$train_{err} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{f}(x_i))^2$$

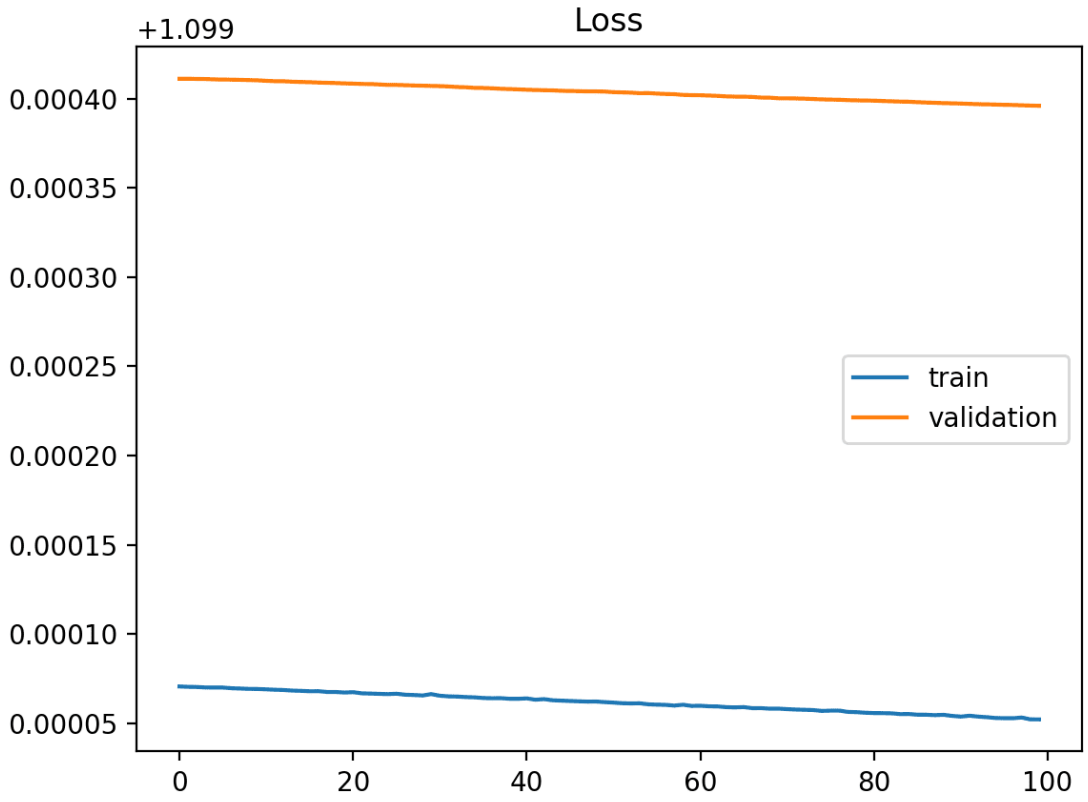
$$test_{err} = \frac{1}{m}\sum_{i=n+1}^{n+m}(y_i - \hat{f}(x_i))^2$$

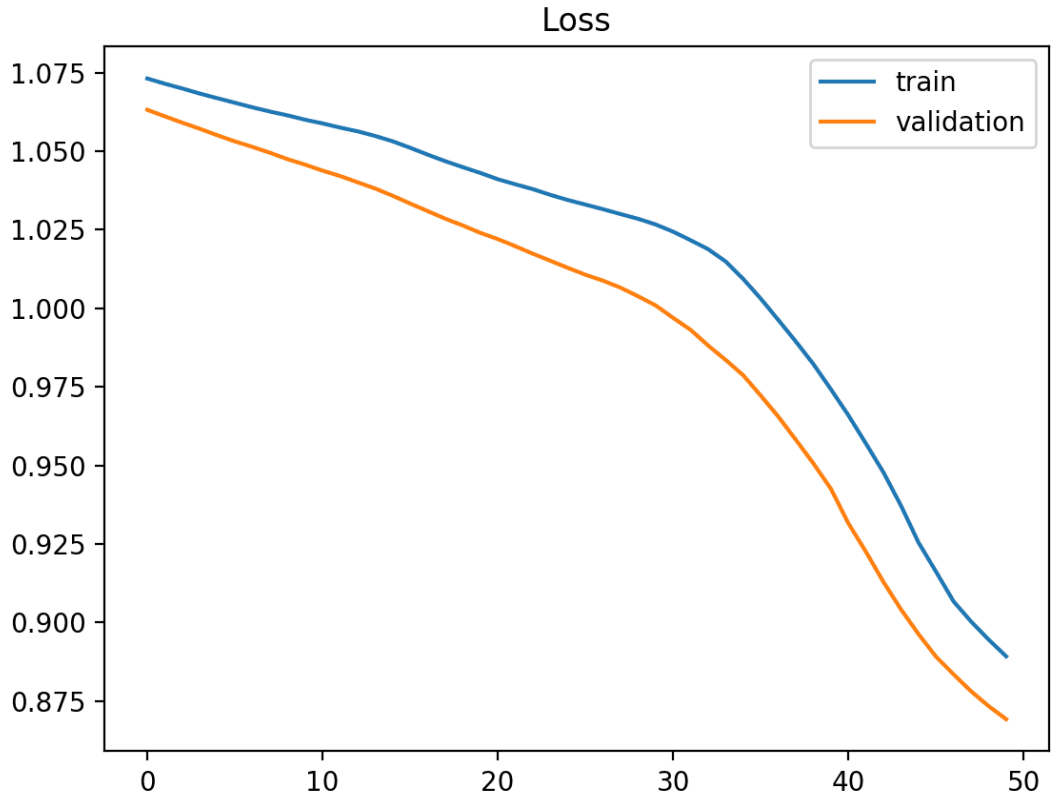
- As the model complexity increases $train_{err}$ becomes overly optimistic and gives us a wrong picture of how close $\hat{f}$ is to f
- The validation error gives the real picture of how close $\hat{f}$ is to f

Learning Curves

- Line plot of learning (y-axis) over experience (x-axis)
- The metric used to evaluate learning could be
- **Optimization Learning Curves**:
  - calculated on the metric by which the parameters of the model are being optimized, e.g. loss.
  - Minimizing, such as loss or error
- **Performance Learning Curves**:
  - calculated on the metric by which the model will be evaluated and selected, e.g. accuracy.
  - Maximizing metric , such as classification accuracy
- **Train Learning Curve**:
  - calculated from the training dataset that gives an idea of how well the model is learning.
- **Validation Learning Curve**:
  - calculated from a hold-out validation dataset that gives an idea of how well the model is generalizing.

Underfit Learning Curves



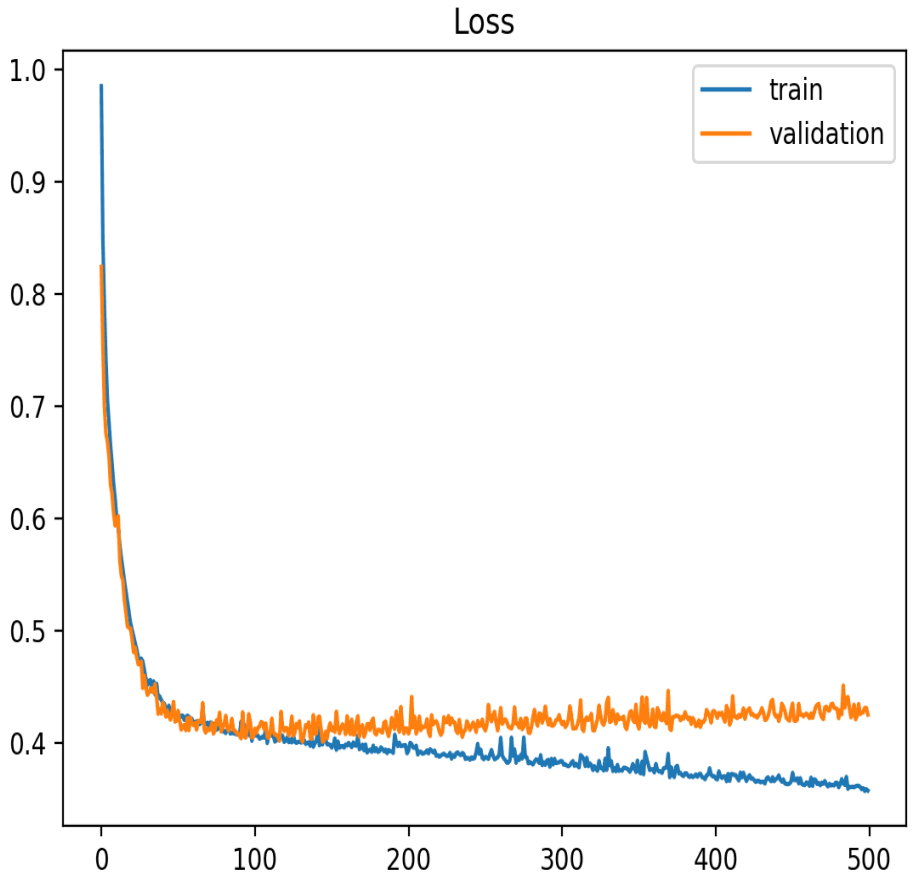


A plot of learning curves shows underfitting if:

- The training loss remains flat regardless of training.
- The training loss continues to decrease until the end of training.

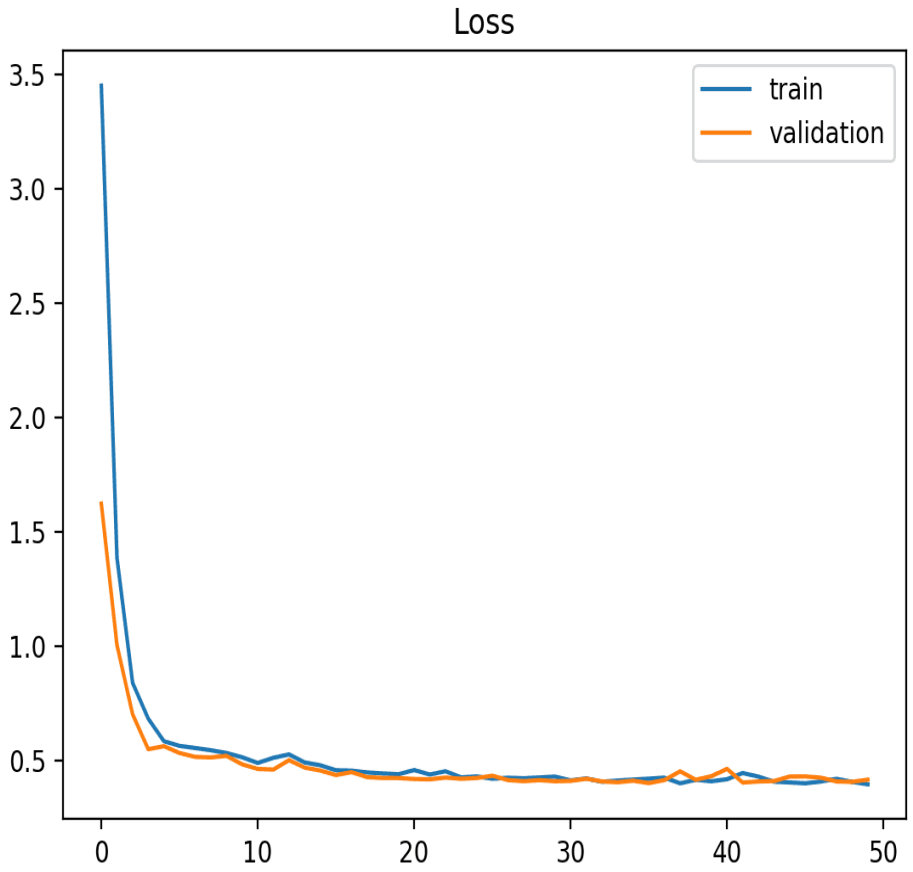
Overfitting Curves

- Overfitting
  - Model specialized on training data, it is not able to generalize to new data
  - Results in increase in generalization error.
  - generalization error can be measured by the performance of the model on the validation dataset.
- A plot of learning curves shows overfitting if:
  - The plot of training loss continues to decrease with experience.
  - The plot of validation loss decreases to a point and begins increasing again.
  - The inflection point in validation loss may be the point at which training could be halted as experience after that point shows the dynamics of overfitting.



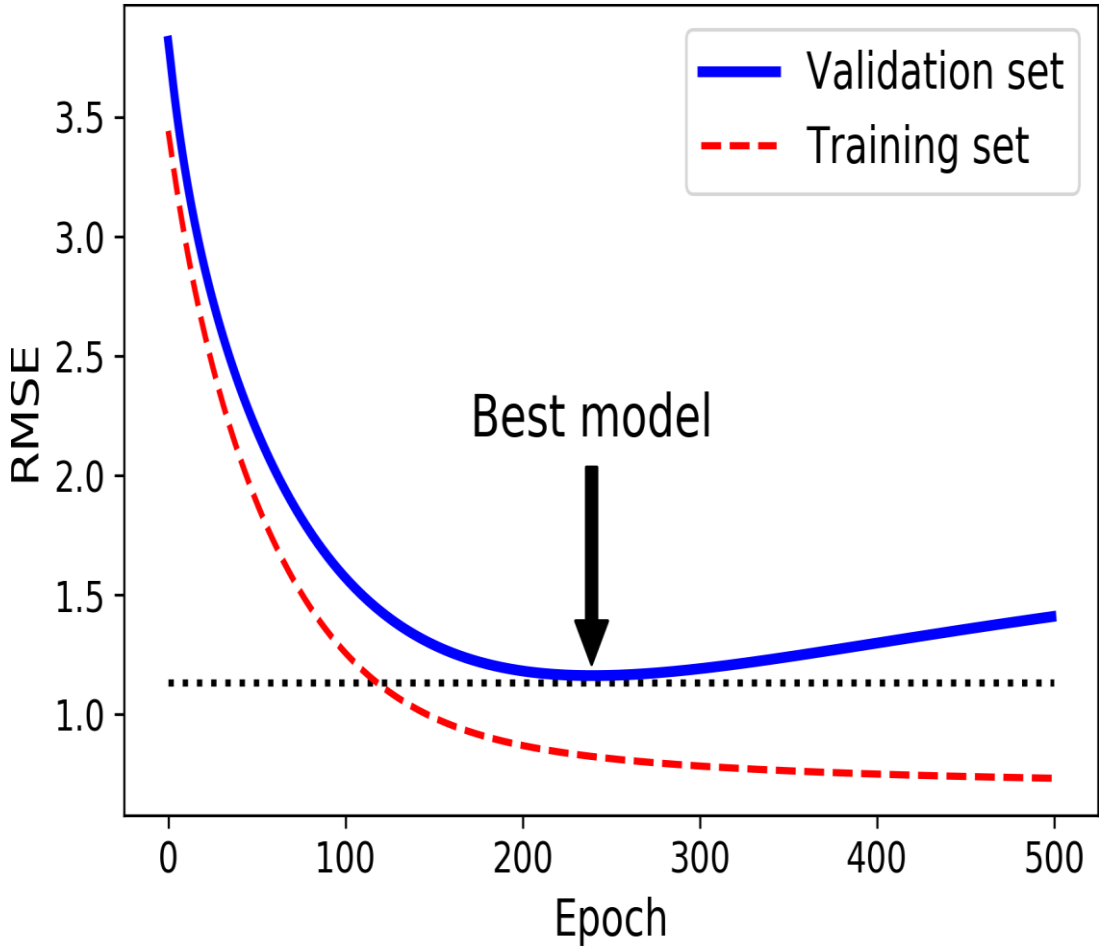
Good Fit Learning Curves

- A good fit is the goal of the learning algorithm and exists between an overfit and underfit model.
- A plot of learning curves shows a good fit if:
  - Plot of training loss decreases to a point of stability
  - Plot of validation loss decreases to a point of stability and has a small gap with the training loss.
- Loss of the model will almost always be lower on the training than the validation dataset.
- We should expect some gap between the train and validation loss learning curves.
- This gap is referred to as the "generalization gap."



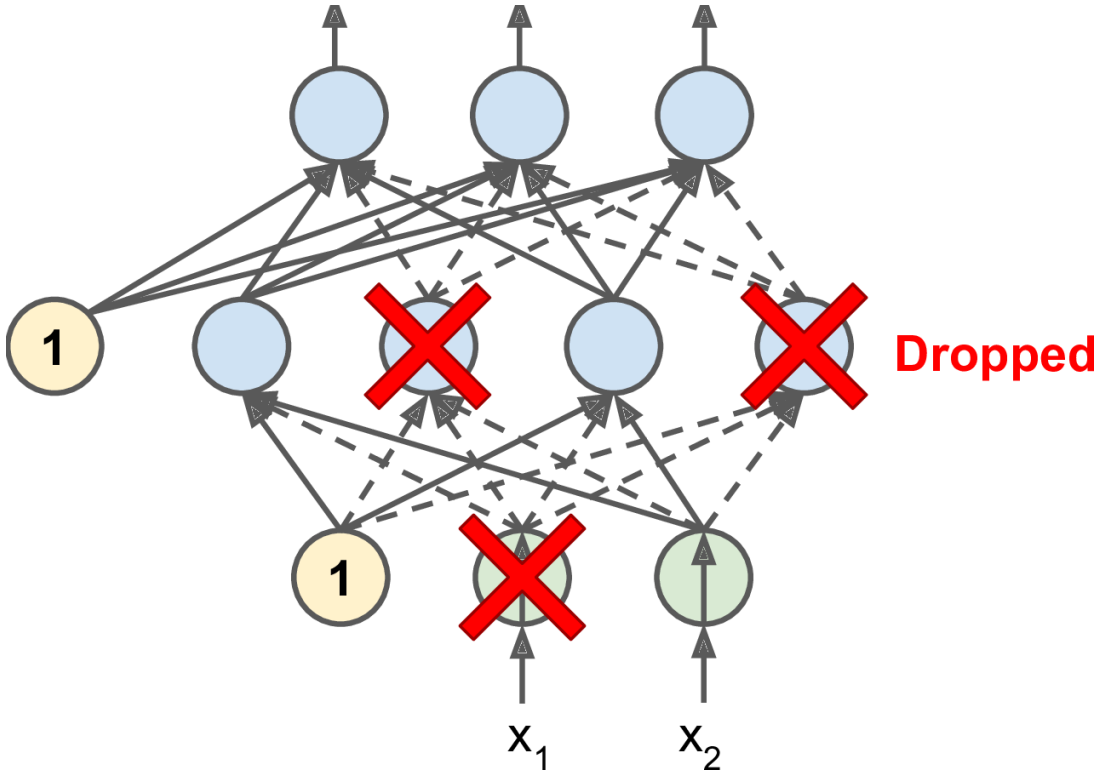
Avoiding Overfitting

- With so many parameters, DNN can fit complex datasets.
- But also prone to overfitting the training set.
- Early Stopping
  - stop training as soon as the validation error reaches a minimum
  - With Stochastic and Mini-batch Gradient Descent, the curves are not so smooth, and it may be hard to know whether you have reached the minimum or not.
  - Stop only after the validation error has been above the minimum for some time



Avoiding overfitting

- Dropout
  - Proposed by Geoffrey Hinton in 2012
  - at every training step, every neuron has a probability p of being temporarily "dropped out,"
  - it will be entirely ignored during this training step, but it may be active during the next step
  - p is called the dropout rate, usually 50%.
  - After training, neurons don't get dropped
  - If p = 50%
    - during testing a neuron will be connected to twice as many input neurons as it was (on average) during training.
    - multiply each input connection weight by the keep probability (1 – p) after training
    - Alternatively, divide each neuron's output by the keep probability during training



Overfitting using Regularization

- L1, l2 regularization
  - Regularization can be used to constrain the NN weights
  - Lasso – (l1) least absolute shrinkage and selection operator
    - adds the “absolute value of magnitude” of the coefficient as a penalty term to the loss function.
  - Ridge regression (l2)
    - adds the “squared magnitude” of the coefficient as the penalty term to the loss function.
  - Use l1(), l2(), l1_l2() function
    - which returns a regularizer that will compute the regularization loss at each step during training
  - Regularization loss is added to final loss
- Max-Norm Normalization
  - It **constrains** the weights w of the incoming connections

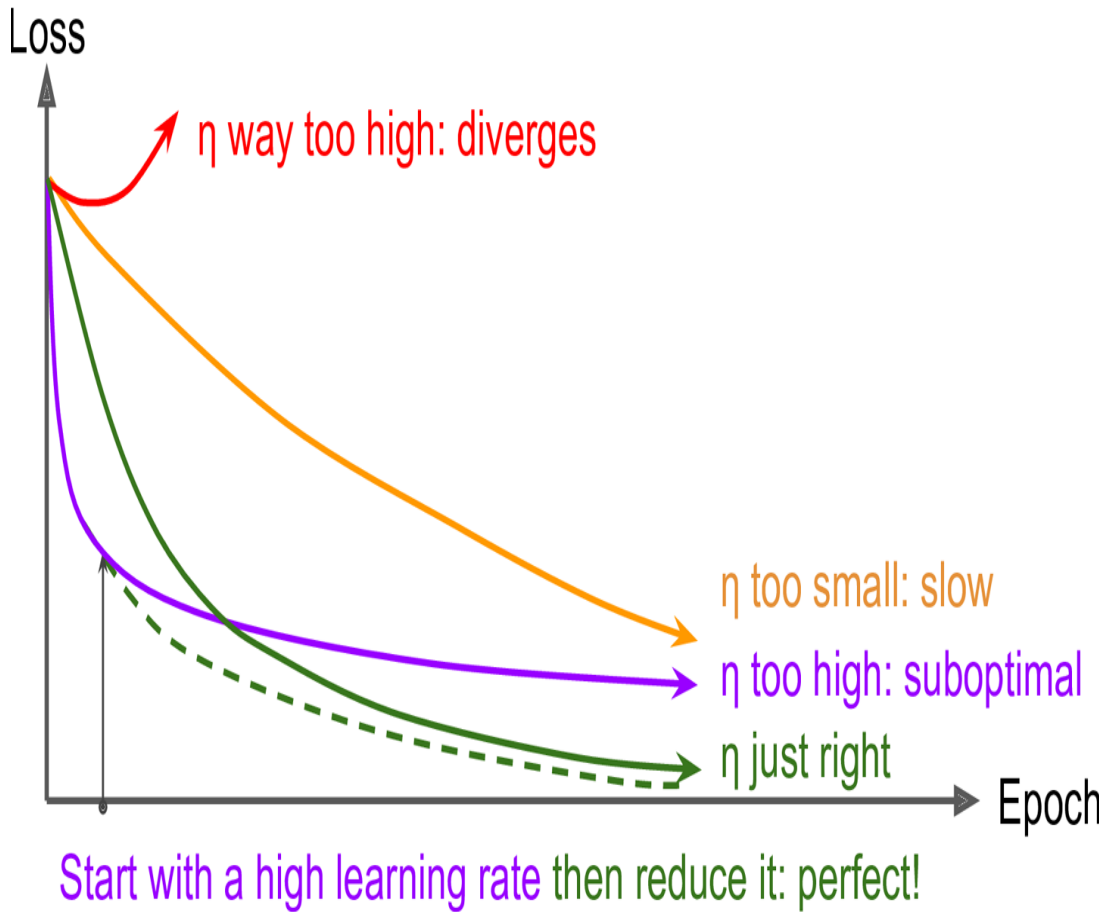
$$(\mathbf{w} \leftarrow \mathbf{w}\frac{r}{\| \mathbf{w} \|_2}).$$

  - r is max-norm hyper parameter **and** $\| \cdot \|2$ is the ℓ2 norm
  - Reducing r increases the amount of regularization and helps reduce overfitting
  - Can also help alleviate the unstable gradients problem if we are not using Batch normalization

Constant learning rate not ideal

- Better to start with a high learning rate
- then reduce it once it stops making fast progress
- can reach a good solution faster

- Learning Schedule strategies can be applied
  - Power Scheduling
  - Exponential Scheduling
  - Piecewise Constant Scheduling
  - Performance Scheduling



Learning Rate Scheduling

- *Power scheduling*
  - Learning rate set to a function of the iteration number 't'
    - t: $\eta(t) = \eta_0 / (1 + t/k)^c$
    - The initial learning rate $\eta 0$, the power c (typically set to 1)
    - The learning rate drops at each step, and after s steps it is down to $\eta 0 / 2$ and so on
  - schedule first drops quickly, then more and more slowly
  - *optimizer = keras.optimizers.SGD(lr=0.01, decay=1e-4)*
  - *The decay is the number of steps it takes to divide the learning rate by one more unit,*
  - *Keras assumes that c is equal to 1.*
- *Exponential scheduling*
  - Set the learning rate to: $\eta(t) = \eta_0 \, 0.1^{t/s}$
  - learning rate will gradually drop by a factor of 10 every s steps.

Learning Rate Scheduling

- *Piecewise constant scheduling*
  - Constant learning rate for a number of epochs
    - e.g., $\eta_0 = 0.1$ for 5 epochs
  - then a smaller learning rate for another number of epochs
    - e.g., $\eta_1 = 0.001$ for 50 epochs and so on
- *Performance scheduling*
  - Measure the validation error every *N* steps (just like for early stopping)
  - reduce the learning rate by a factor of λ when the error stops dropping

References

- Ian Goodfellow, Yoshua Bengio and Aaron Courville, “Deep Learning”, MIT Press 2016
- Swayam NPTEL Notes- Deep Learning, Mitesh Khapra
- Course Notes – Neural Networks and Deep Learning, Andrew NG
- Aurelien Geron, “Hands-On Machine Learning with Scikit-Learn , Keras & Tensorflow, OReilly Publications
- Jiawei Han and Micheline Kamber, “Data Mining Concepts And Techniques”, 3rd Edition, Morgan Kauffmann