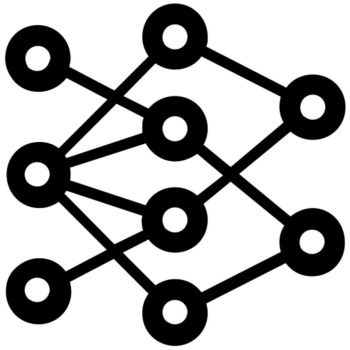
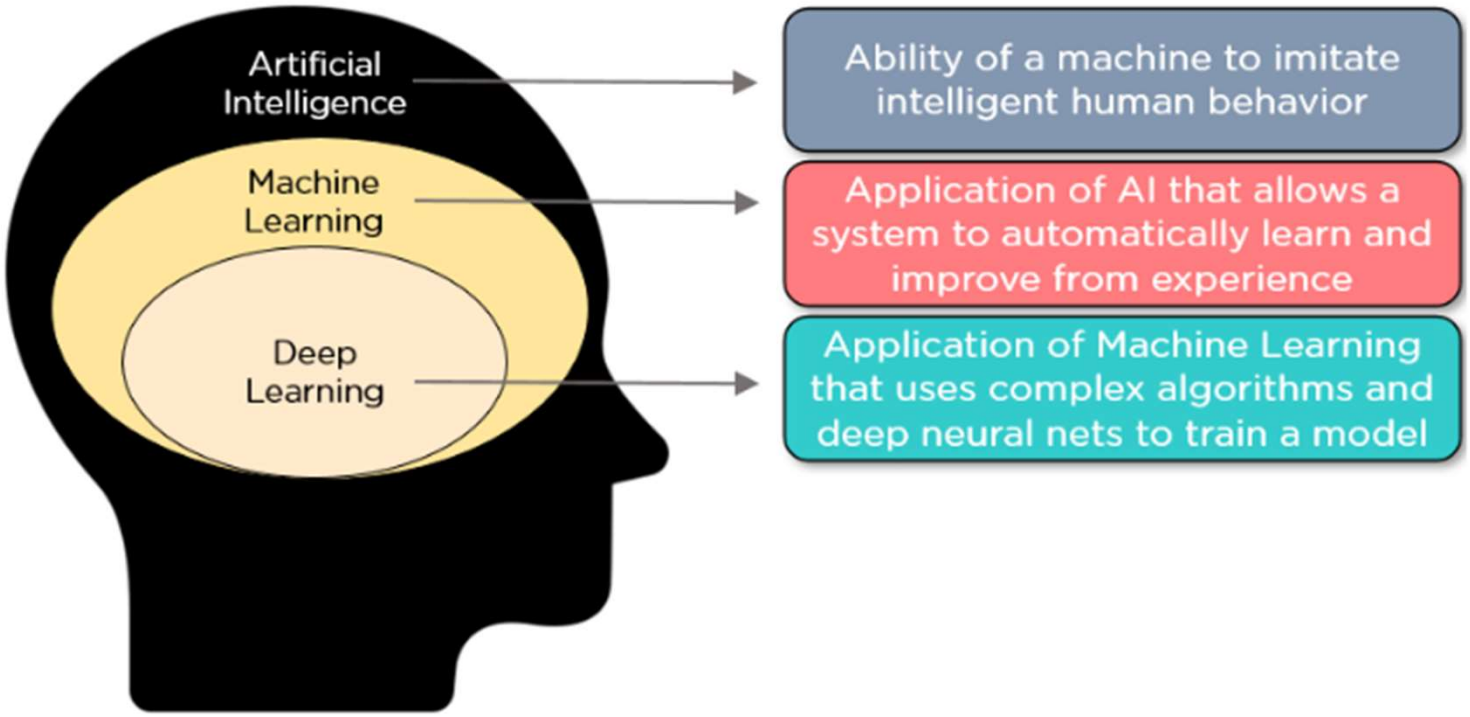


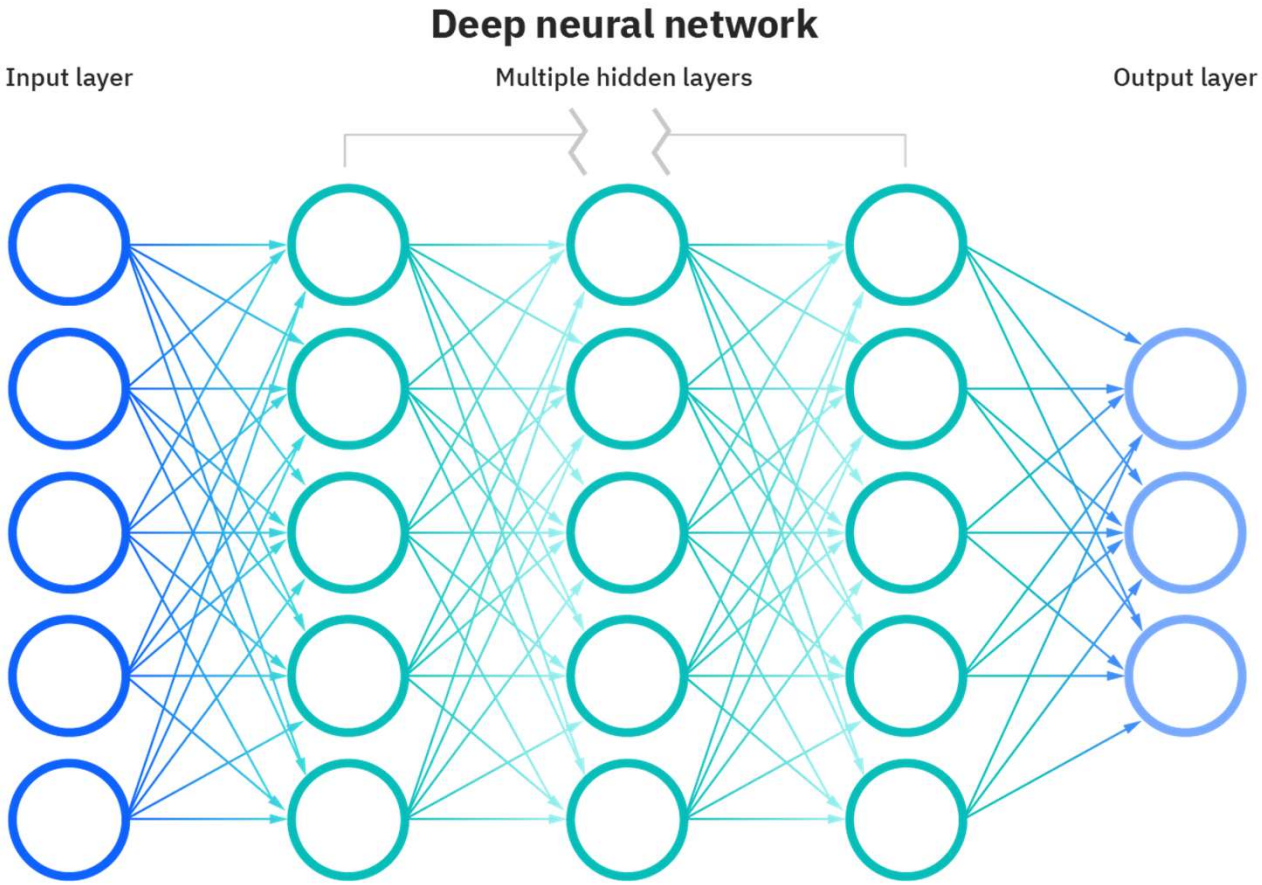
Deep Learning

Prof. R. Raja Subramanian

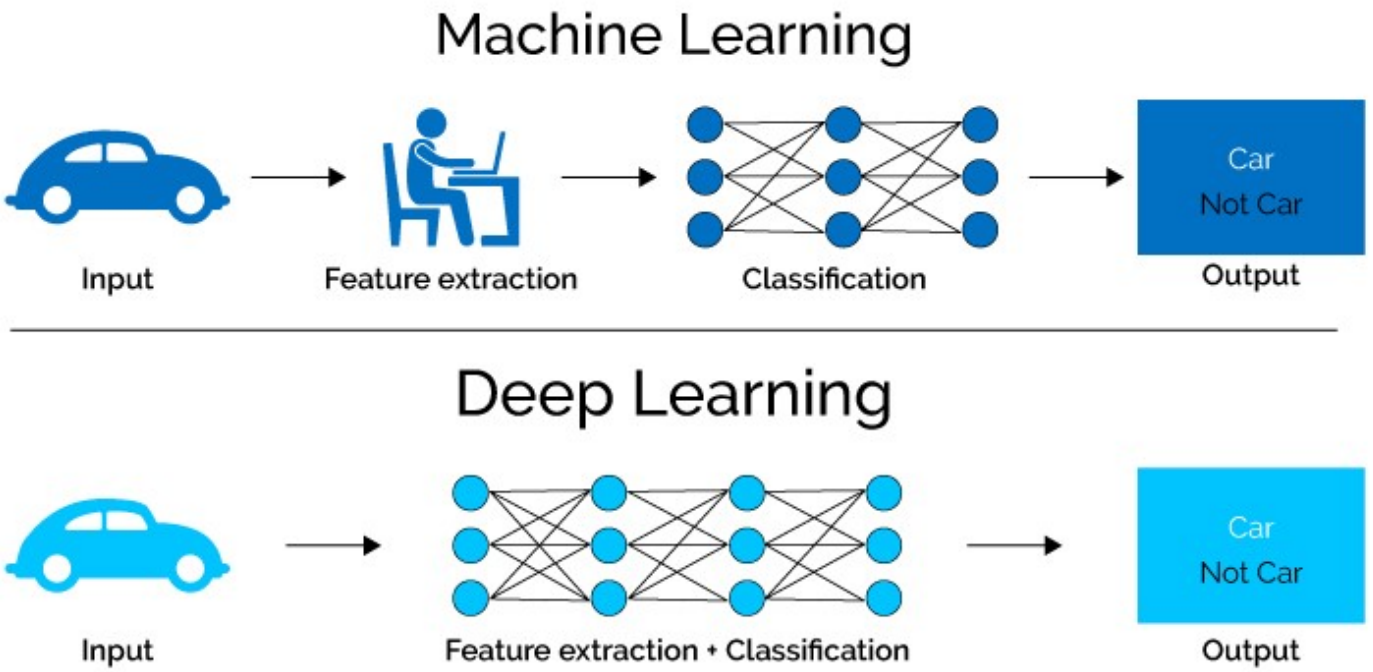
Deep Learning



Deep Learning



Deep Learning



Growing uses for deep learning

Medical

- Skin cancer identification
- FDA approval for death prediction
- Radiology
- Predict disease from patient records

Agriculture

- Identify Plant Pests
- Create more efficient seeds
- Monitor crops in real time
- Identify soil defects & nutrients deficiencies

Pharma

- Design drugs
- Bioinformatics
- Predict the chemical reactions between candidate compounds and target molecules
- Identify one or more genes responsible for a disease

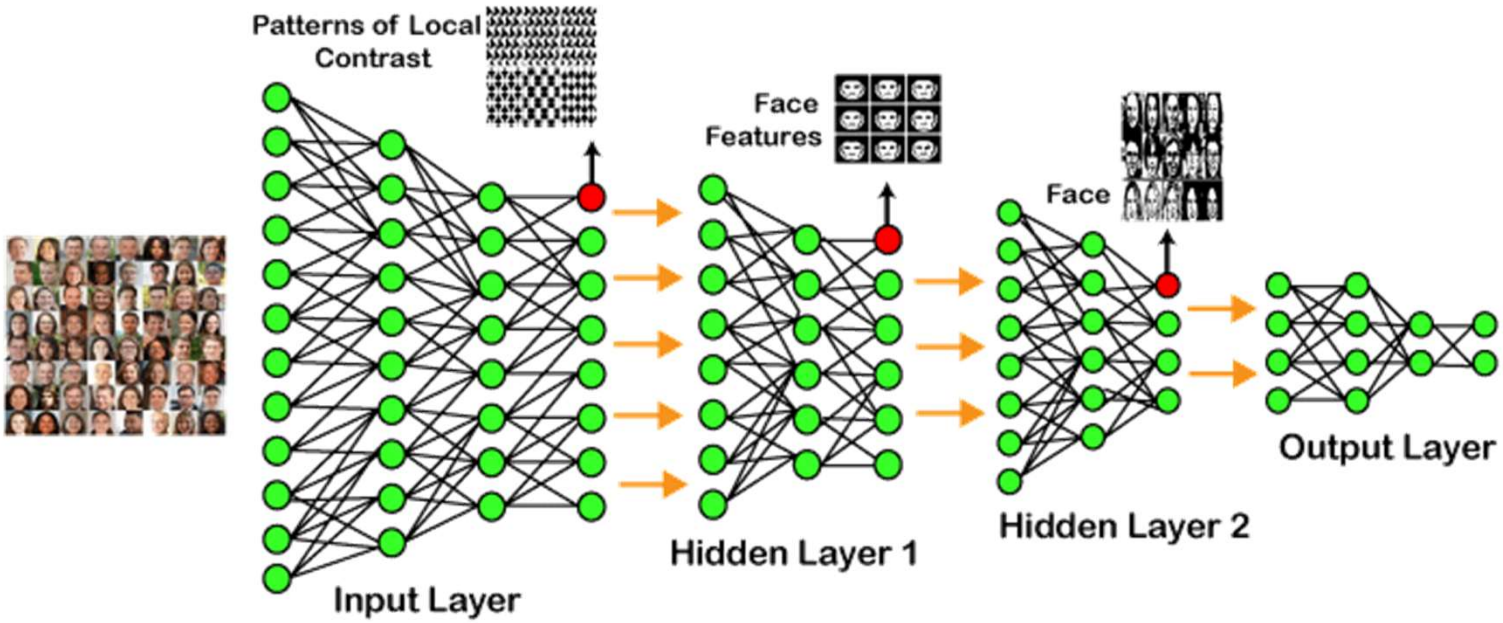
Autonomous Vehicles

- Map raw pixels from camera directly to steering commands
- Drive in unstructured conditions
- Car and lane detection
- Motion control & planning
- Optimization of AV traffic

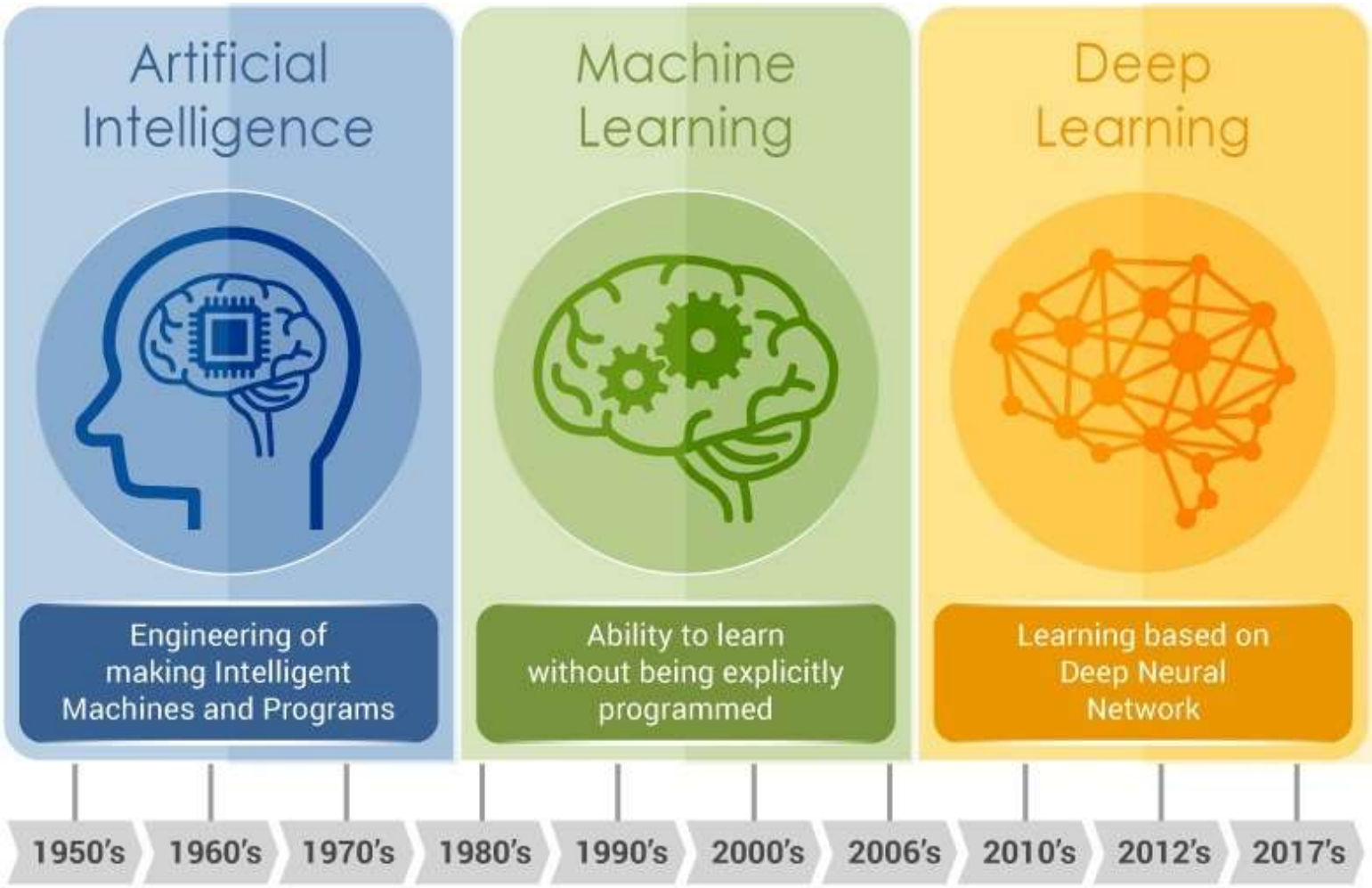
Data Centers

- Data center security
- Reduce electricity usage
- Server optimization

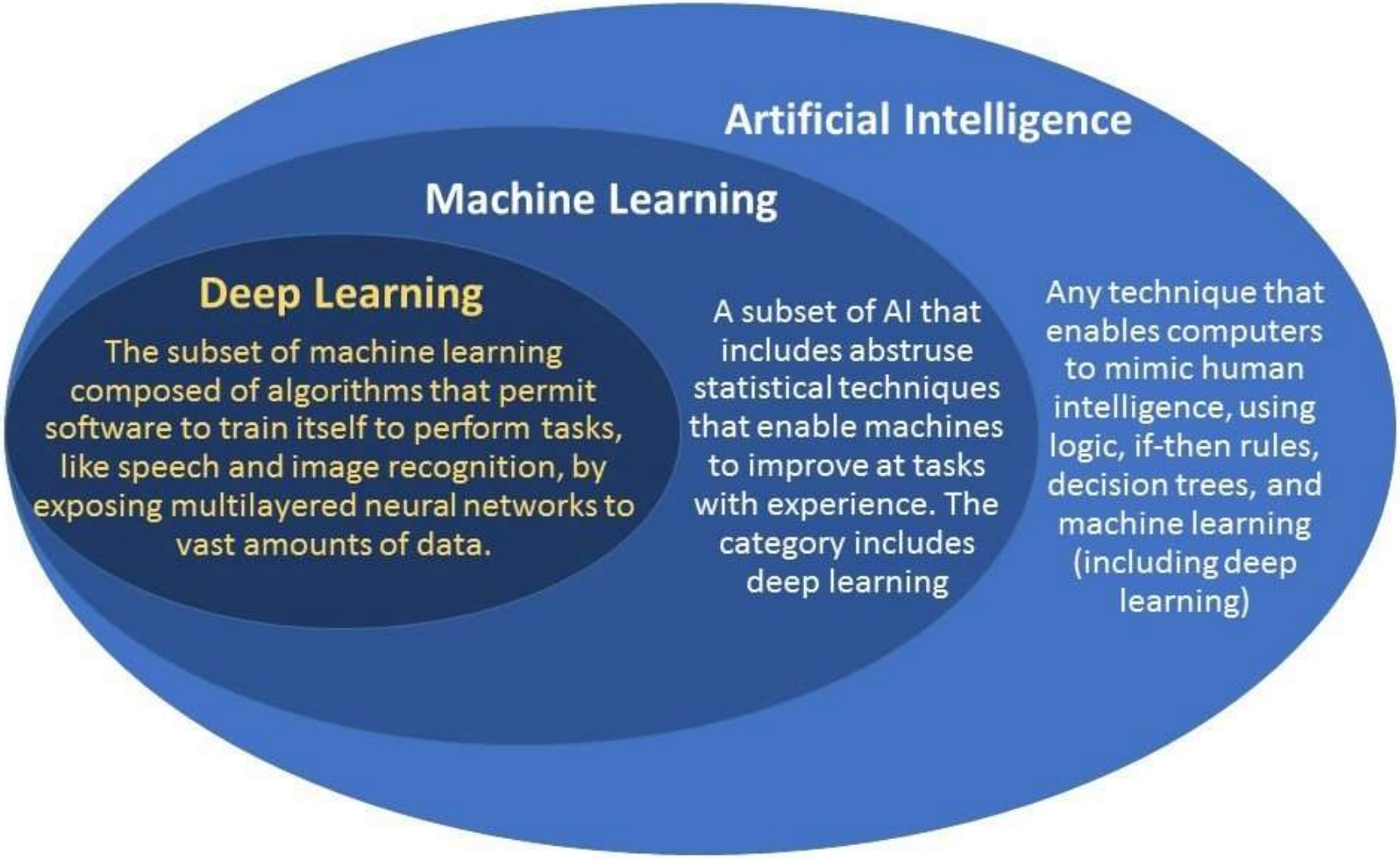
Deep Learning



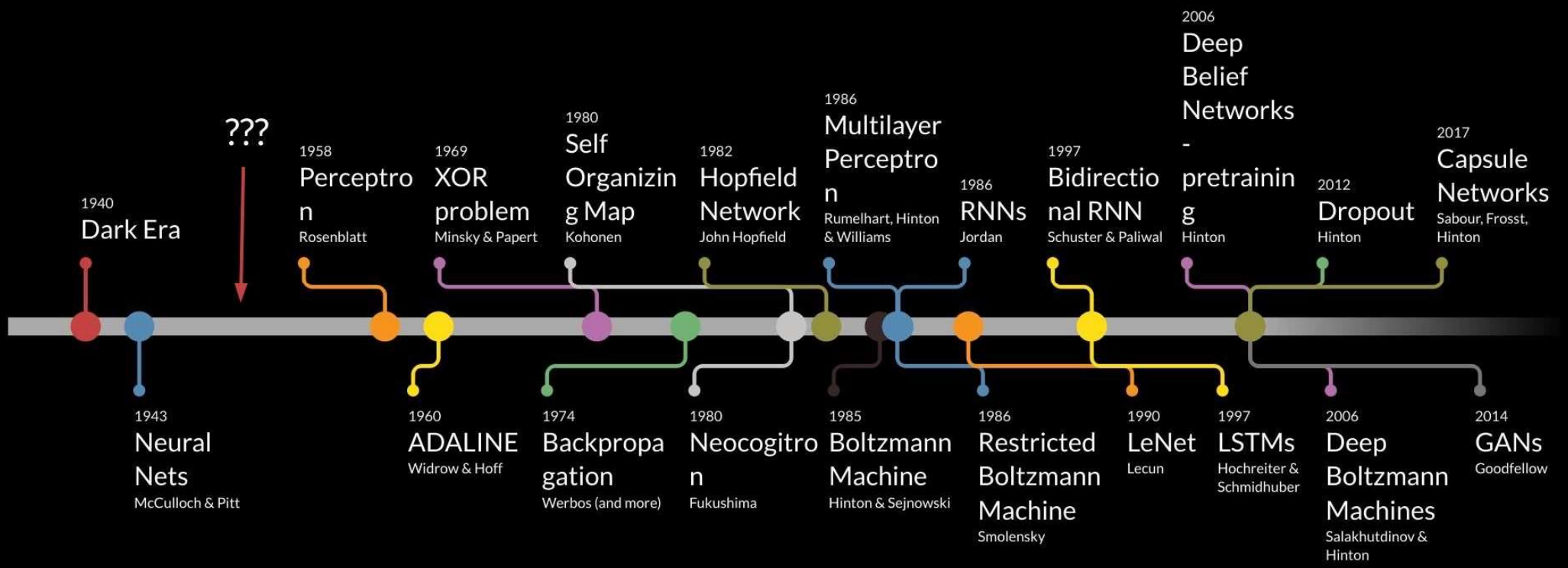
Deep Learning



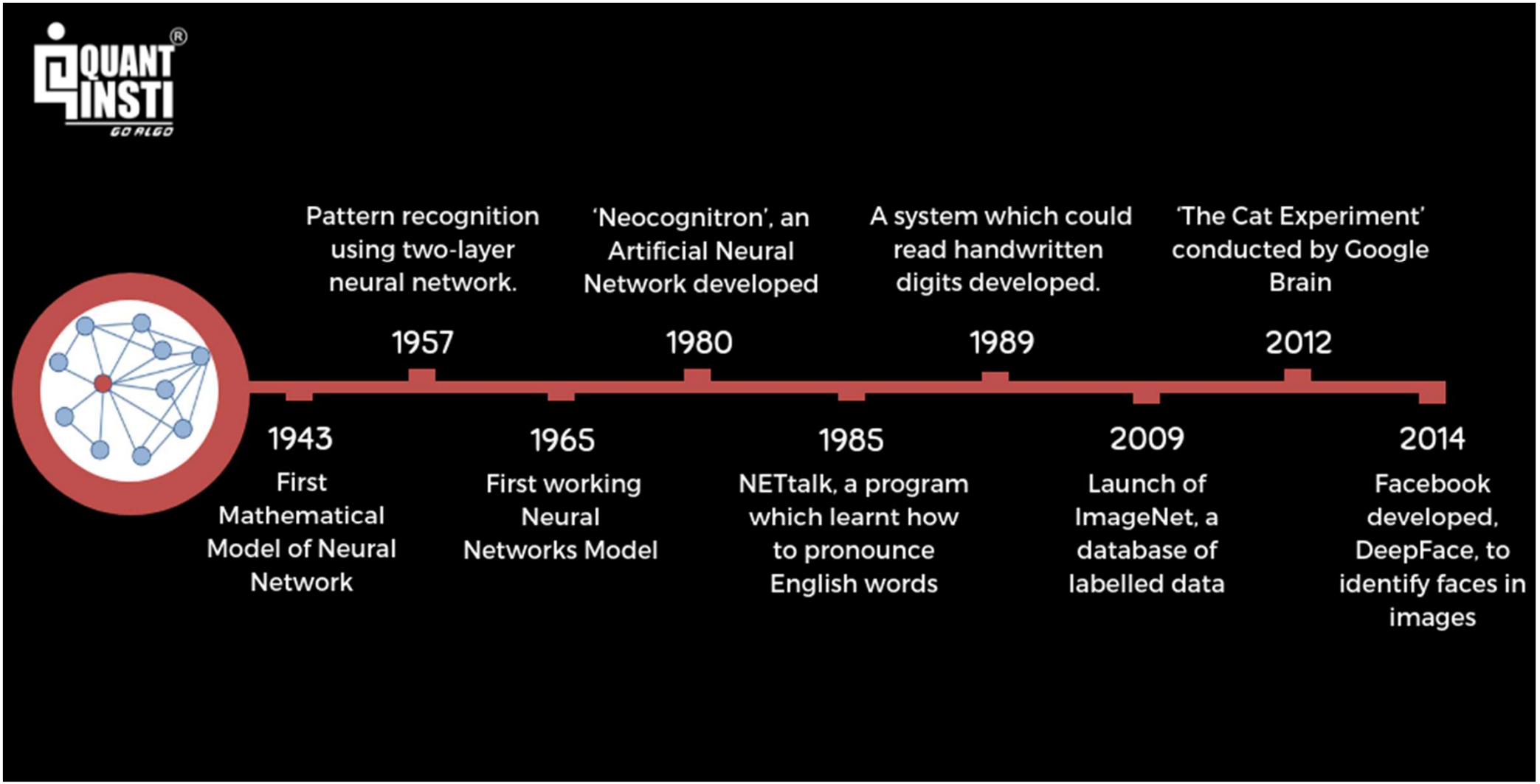
AI Vs ML Vs DL

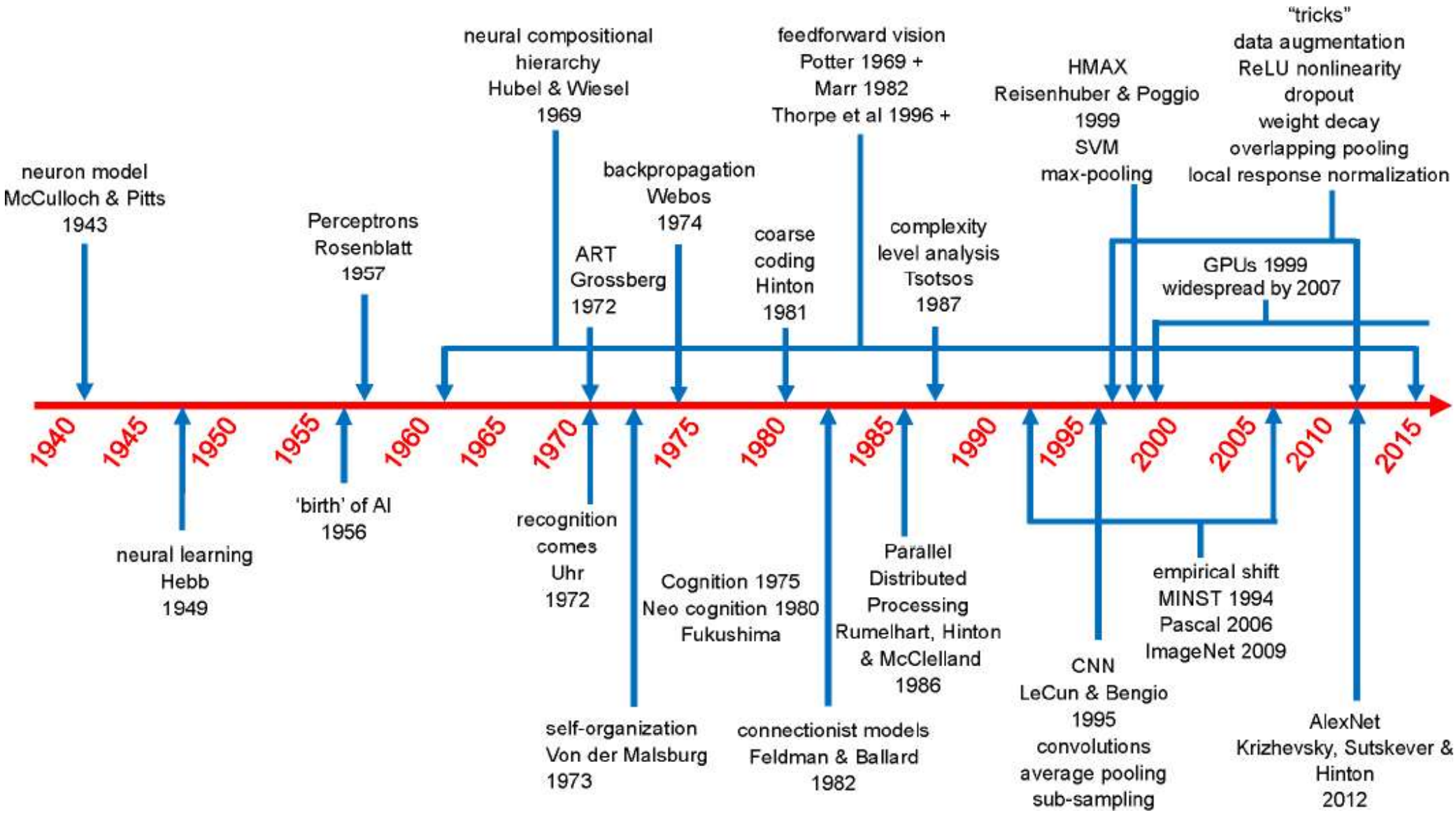

Artificial Intelligence
Any technique that enables computers to mimic human intelligence, using logic, if-then rules, decision trees, and machine learning (including deep learning)
Machine Learning
A subset of AI that includes abstruse statistical techniques that enable machines to improve at tasks with experience. The category includes deep learning
Deep Learning
The subset of machine learning composed of algorithms that permit software to train itself to perform tasks, like speech and image recognition, by exposing multilayered neural networks to vast amounts of data.

Deep Learning Timeline

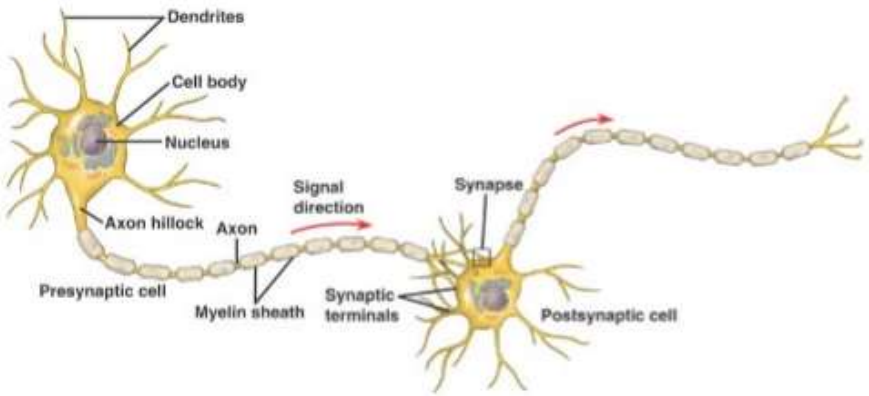


Made by Favio Vázquez


QUANT INSTI
GO ALGO
1943
First Mathematical Model of Neural Network
1957
Pattern recognition using two-layer neural network.
1965
First working Neural Networks Model
1980
'Neocognitron', an Artificial Neural Network developed
1985
NETtalk, a program which learnt how to pronounce English words
1989
A system which could read handwritten digits developed.
2009
Launch of ImageNet, a database of labelled data
2012
'The Cat Experiment' conducted by Google Brain
2014
Facebook developed, DeepFace, to identify faces in images

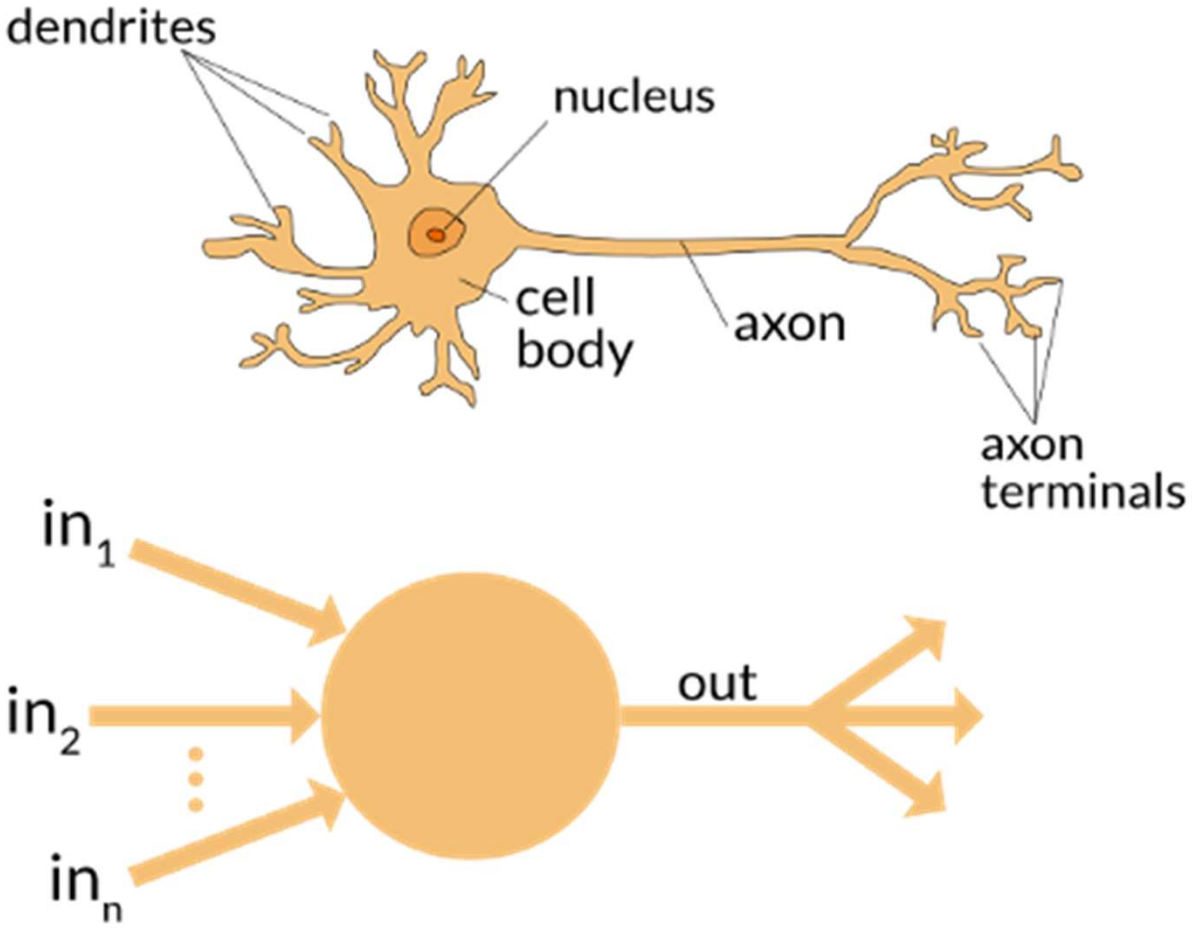

neuron model
McCulloch & Pitts
1943
neural learning
Hebb
1949
Perceptrons
Rosenblatt
1957
'birth' of AI
1956
neural compositional
hierarchy
Hubel & Wiesel
1969
ART
Grossberg
1972
recognition
comes
Uhr
1972
backpropagation
Webos
1974
self-organization
Von der Malsburg
1973
coarse
coding
Hinton
1981
Cognition 1975
Neo cognition 1980
Fukushima
connectionist models
Feldman & Ballard
1982
feedforward vision
Potter 1969 +
Marr 1982
Thorpe et al 1996 +
complexity
level analysis
Tsotsos
1987
Parallel
Distributed
Processing
Rumelhart, Hinton
& McClelland
1986
HMAX
Reisenhuber & Poggio
1999
SVM
max-pooling
CNN
LeCun & Bengio
1995
convolutions
average pooling
sub-sampling
"tricks"
data augmentation
ReLU nonlinearity
dropout
weight decay
overlapping pooling
local response normalization
GPUs 1999
widespread by 2007
empirical shift
MINST 1994
Pascal 2006
ImageNet 2009
AlexNet
Krizhevsky, Sutskever &
Hinton
2012
1940
1945
1950
1955
1960
1965
1970
1975
1980
1985
1990
1995
2000
2005
2010
2015

Inspiration from Human brain

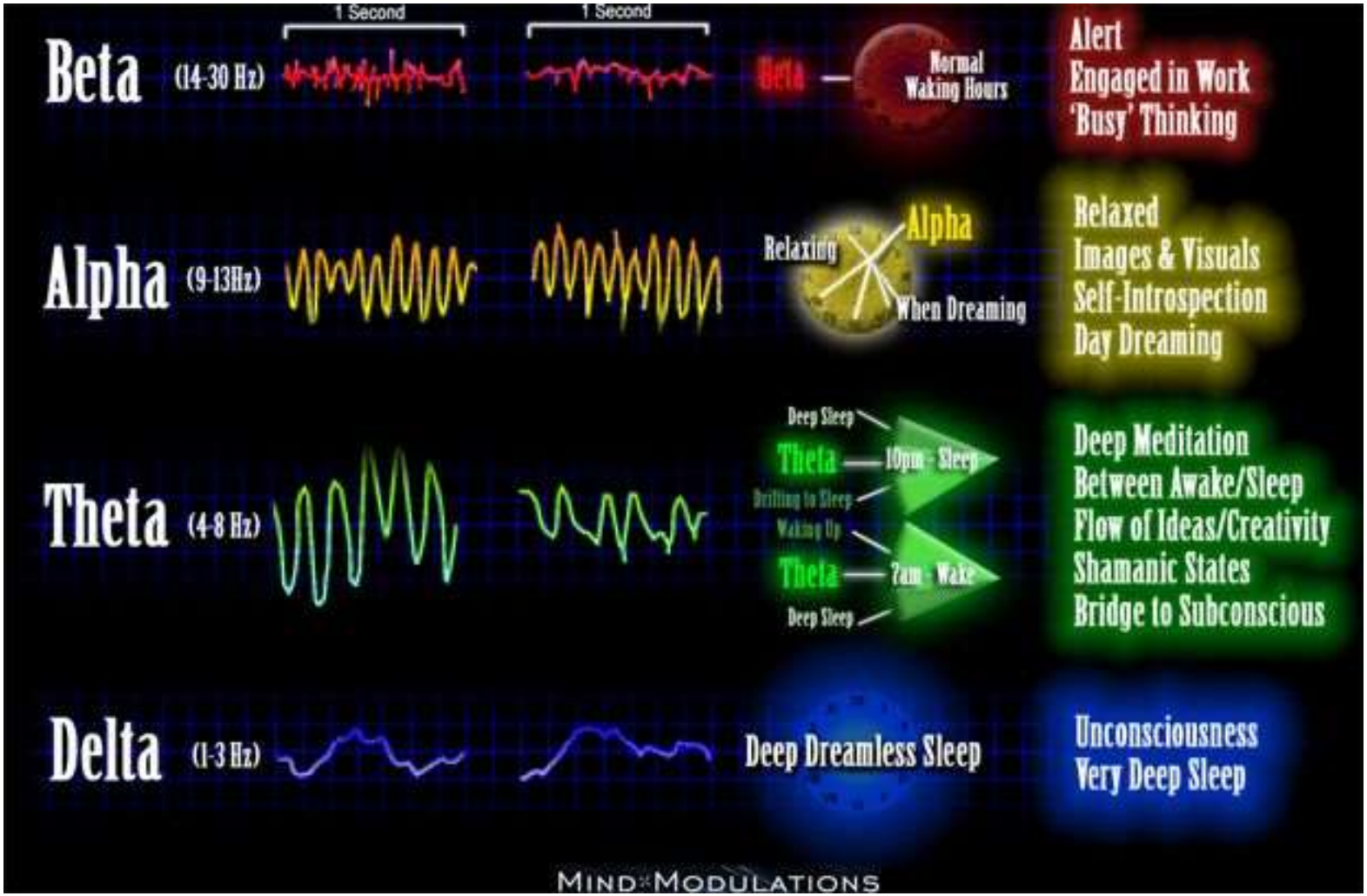


- Over 100 billion neurons
- Around 1000 connections
- Electrical signal transmitted through axon, can be inhibitory or excitatory
- Activation of a neuron depends on inputs

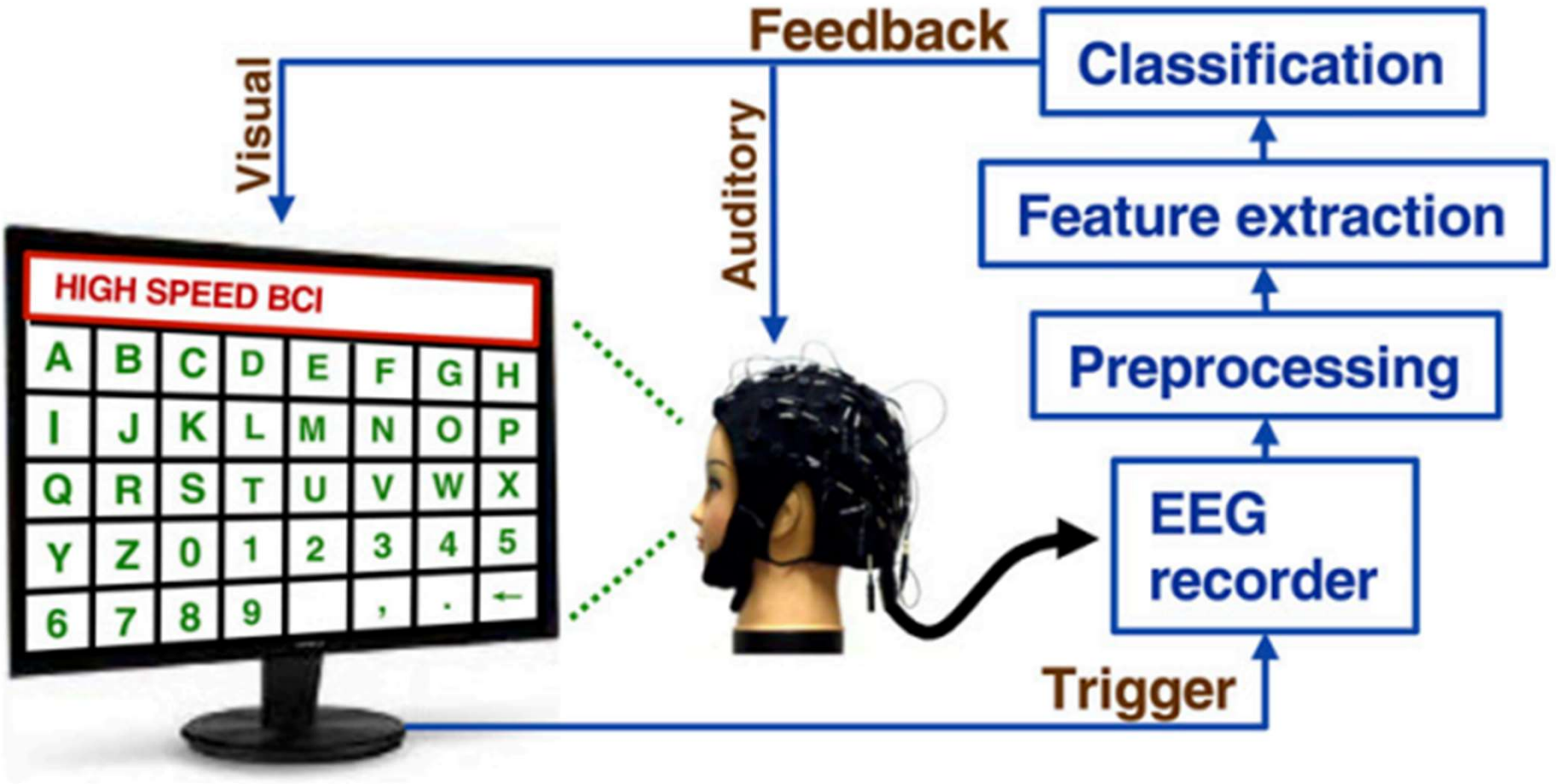
Human Brain => Perceptron Deep Learning

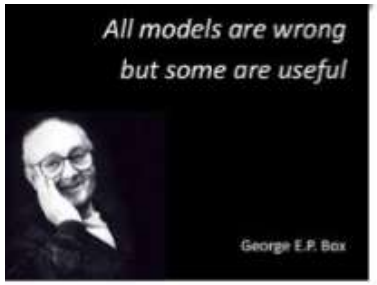


Human Brain Frequencies

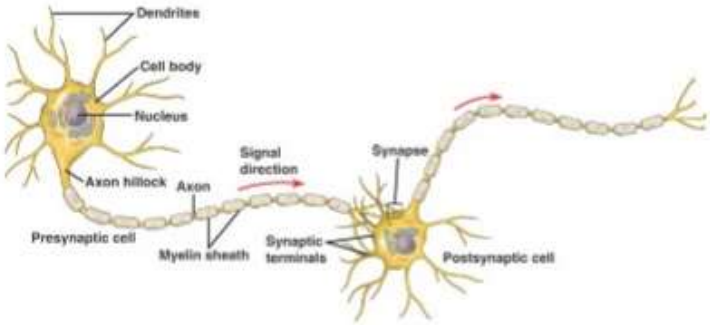


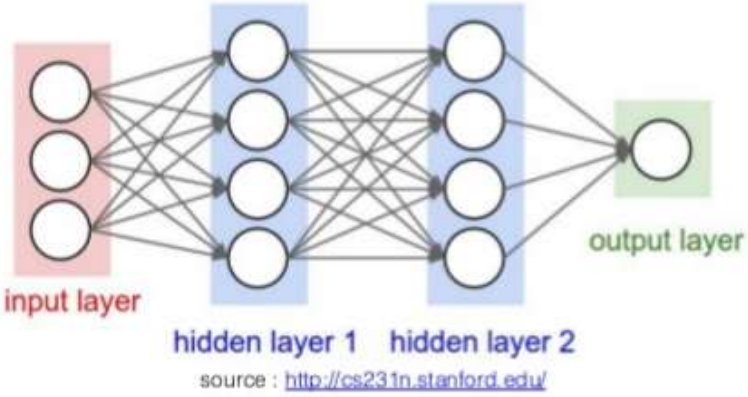
Human Brain – Machine Mapping





Artificial Neuron

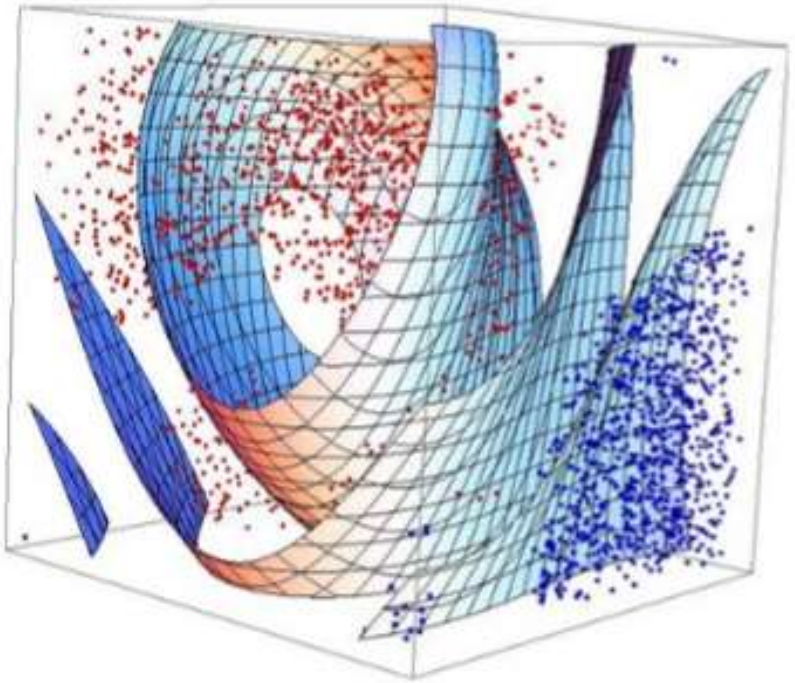




source : http://cs231n.stanford.edu/

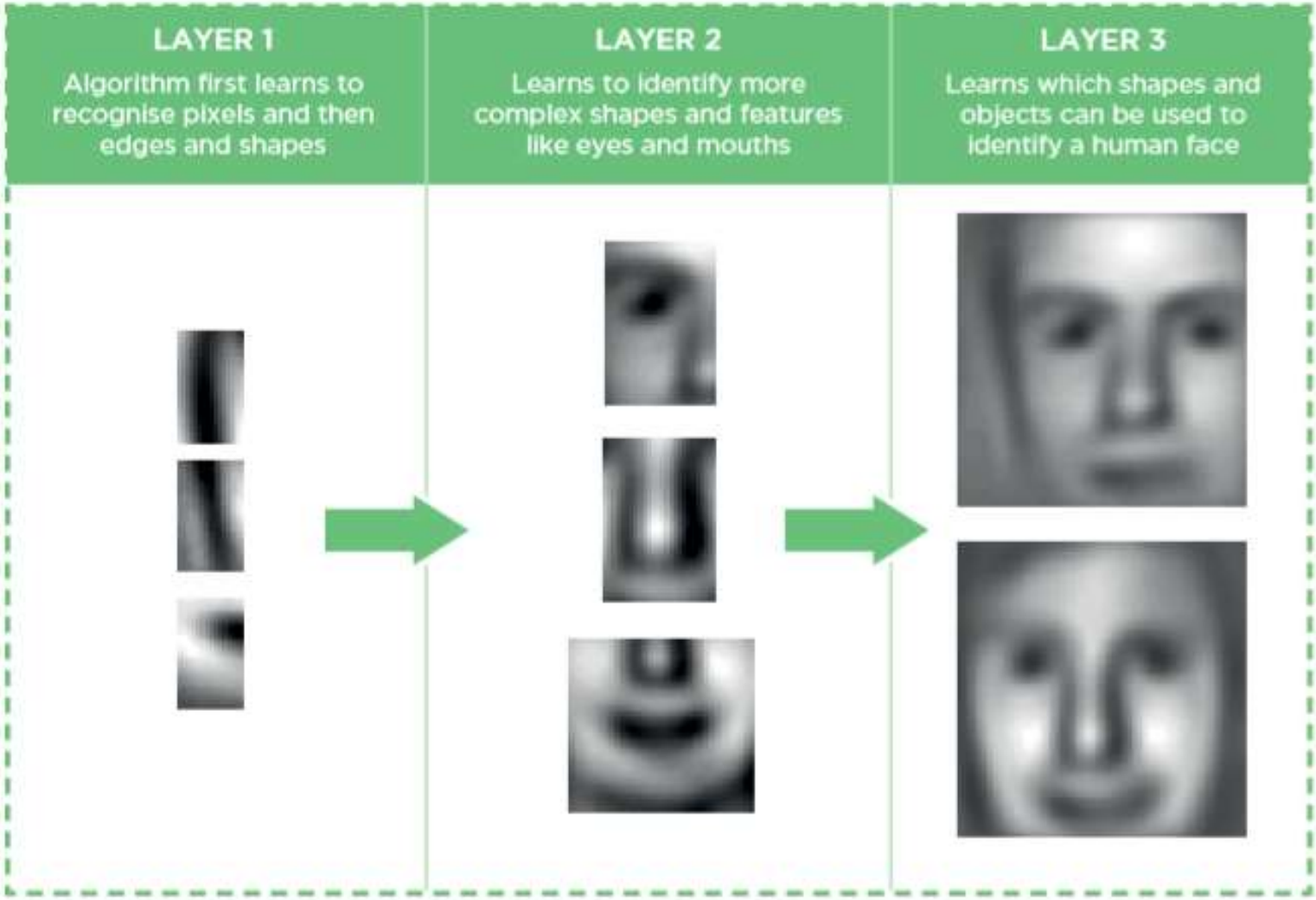
- Circles ——————————- Neuron
- Arrow ——————————— Synapse
- Weight ——————————- Electrical signal

Non-Linearity

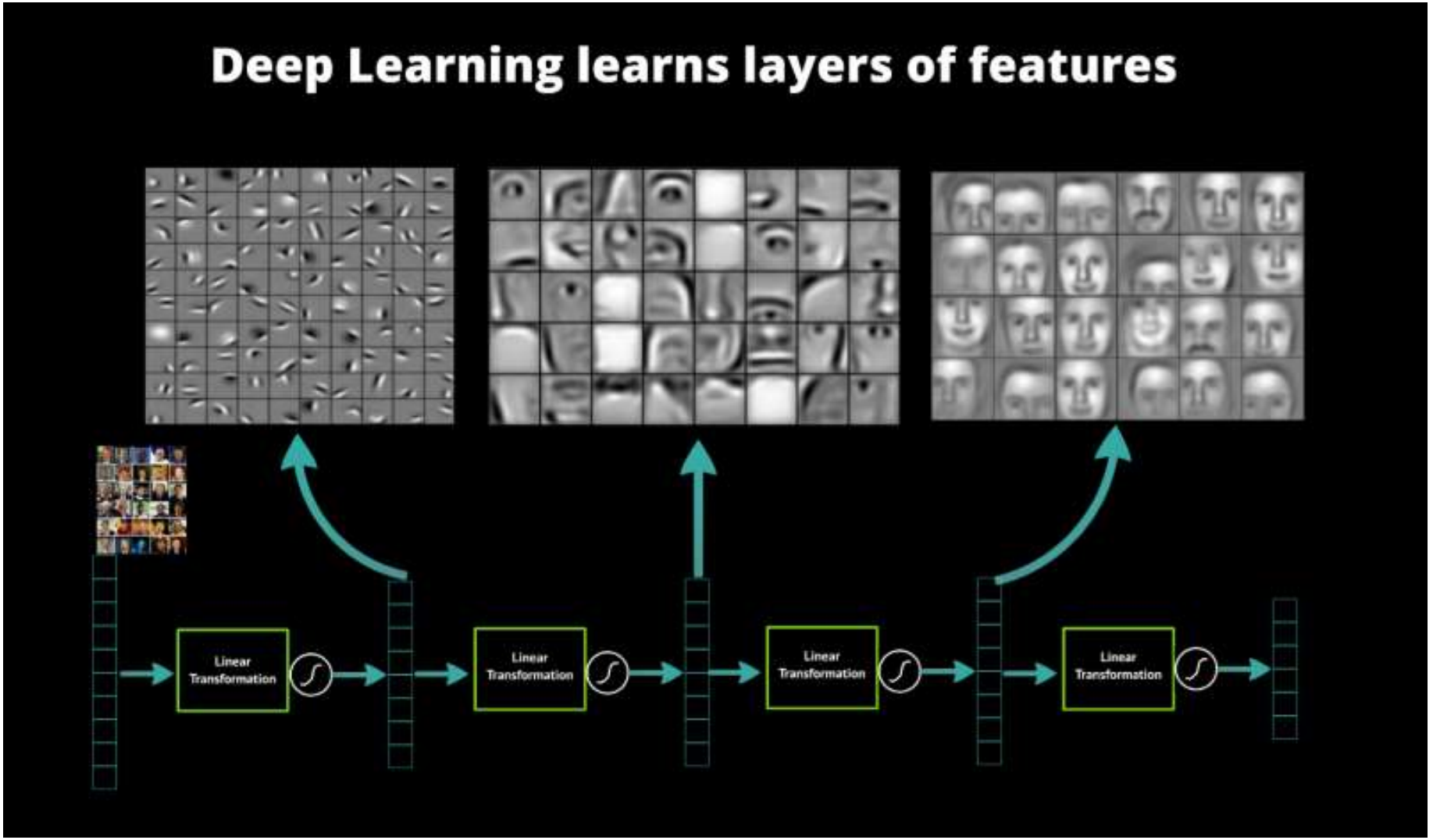


- Neural network learns complicated structured data with non-linear activation functions

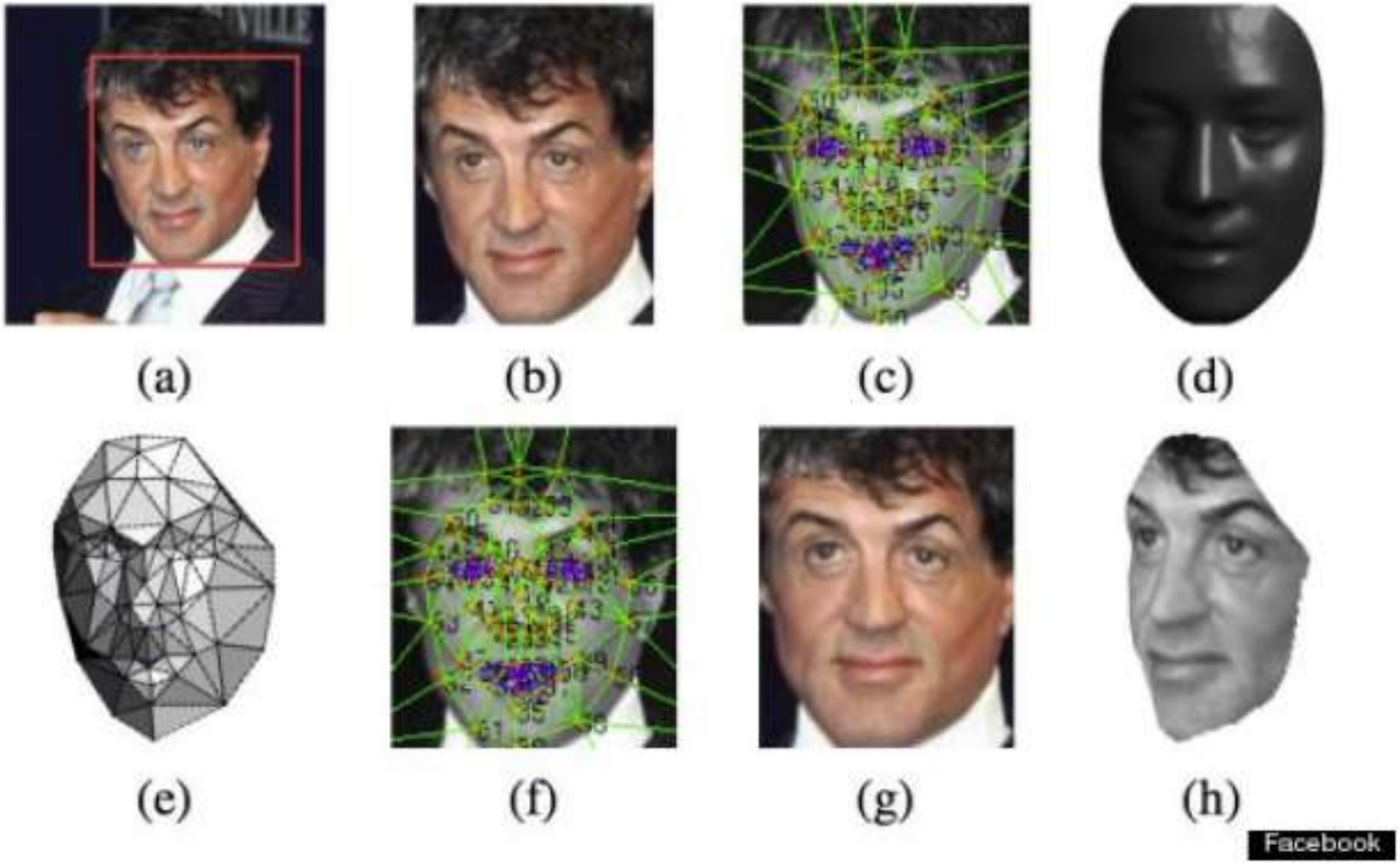
Compositionality of Data



Compositionality of Data

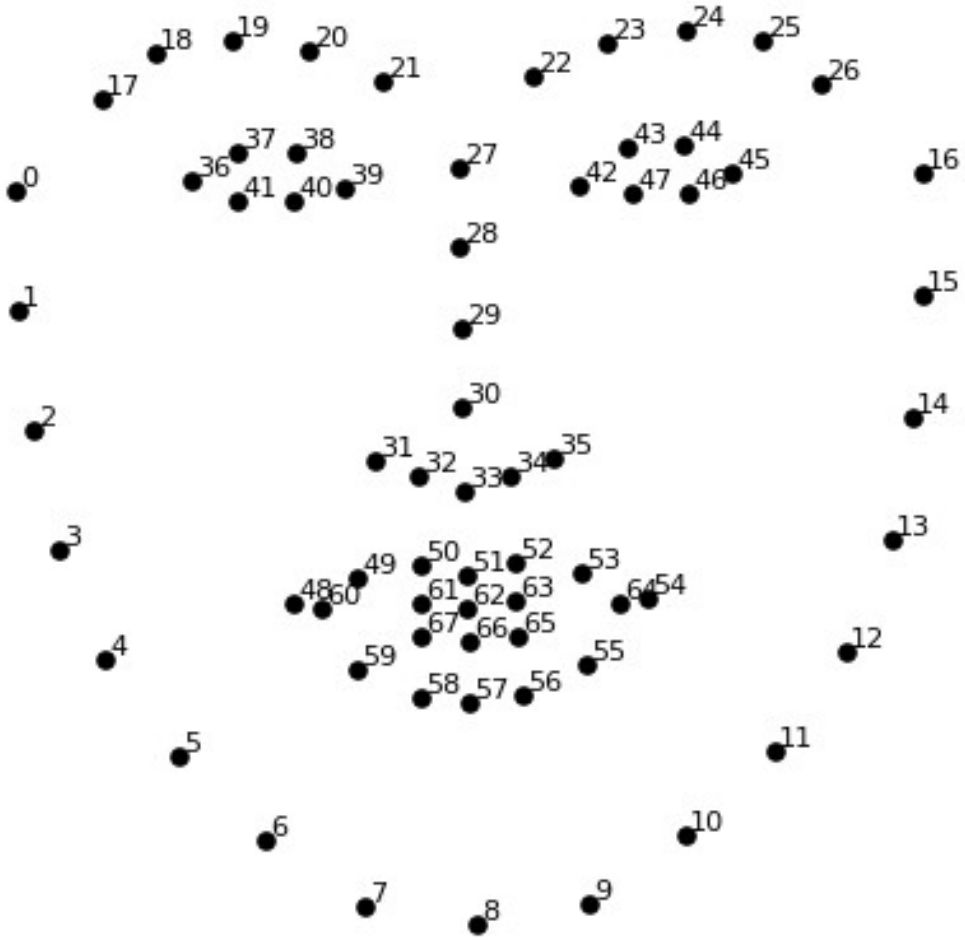


Applications of Neural Networks in Social Media



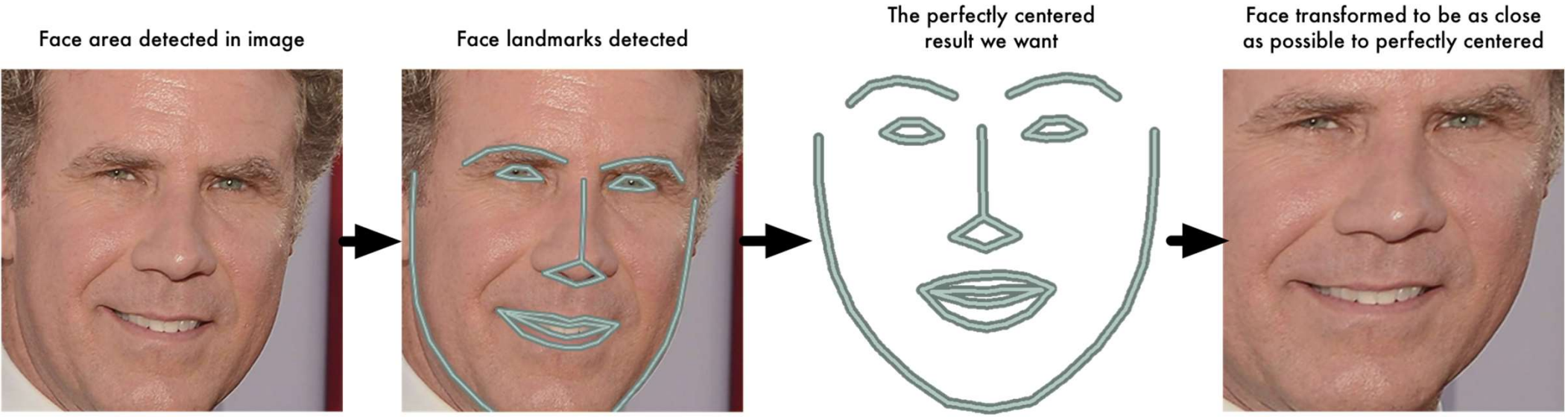
Applications of Neural Networks in Social Media

✓**68 Face Landmark Estimation – Used by Facebook**

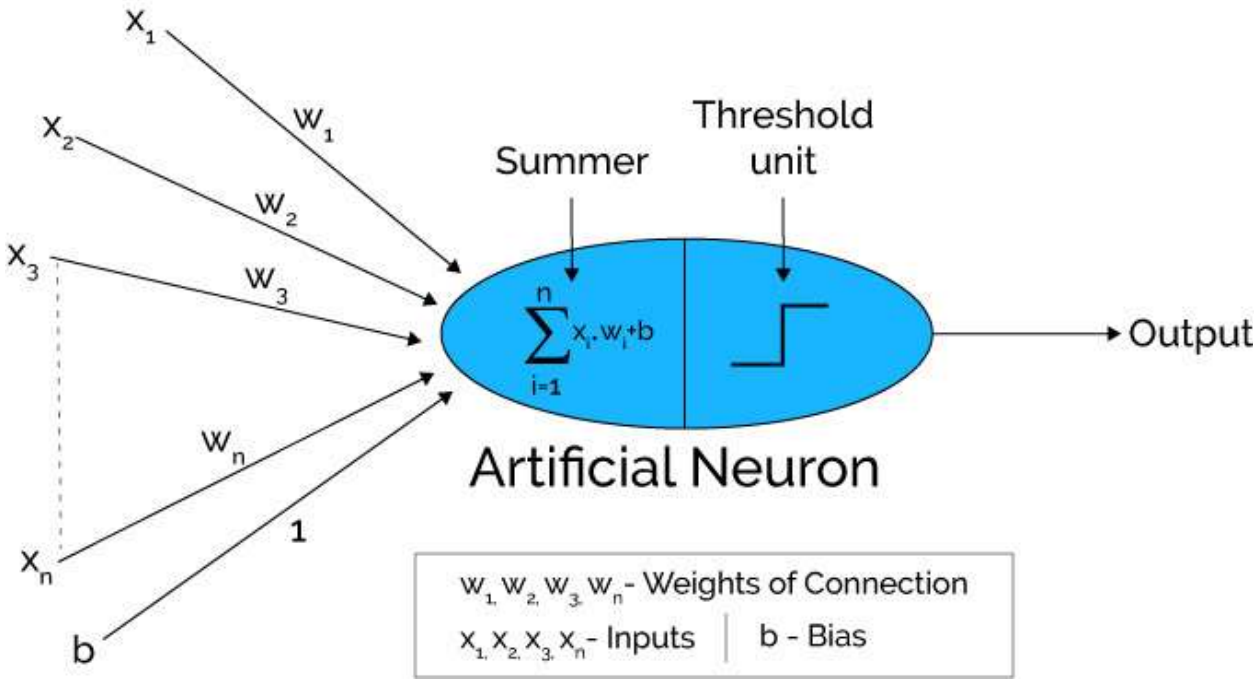


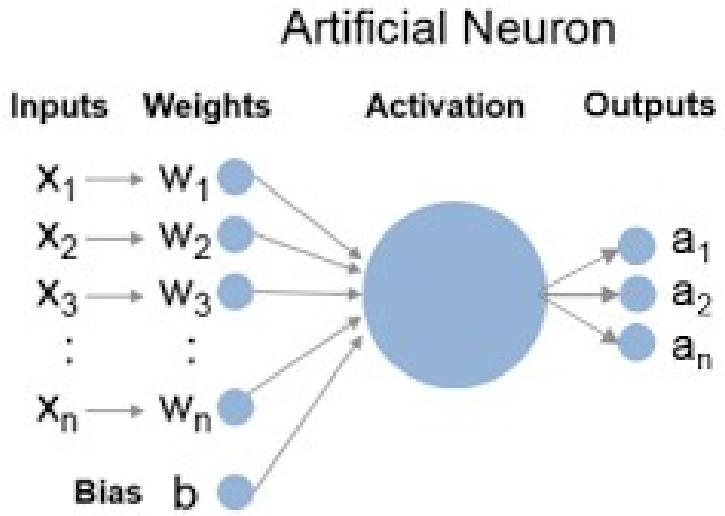
Applications of Neural Networks in Social Media

✓**68 Face Landmark Estimation – Used by Facebook**



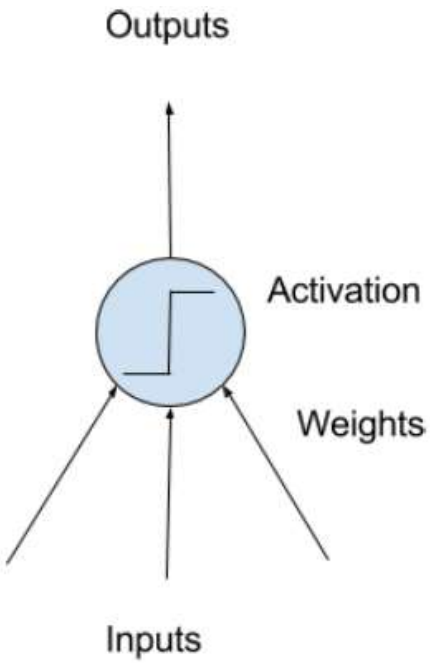
Artificial Neuron





Artificial Neurons

- The building block for neural networks are artificial neurons.
- These are simple computational units that have weighted input signals and produce an output signal using an activation function

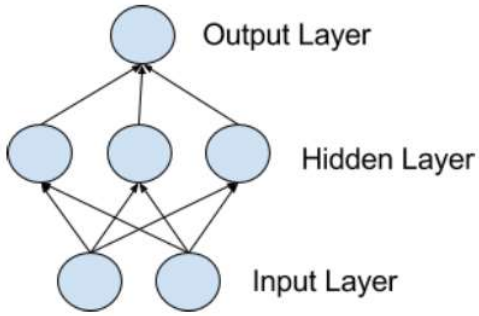


- **Neuron Weights**
- Each neuron has a bias which always has the value 1.0
- A neuron may have two inputs in which case it requires three weights.
- One for each input and one for the bias.
- Weights are often initialized to small random values, values in the range 0 to 0.3,
- Better to keep weights in the network small and regularization techniques can be used.

- **Activation**
- The weighted inputs are summed and passed through an activation function, called a transfer function.
- An activation function is a simple mapping of summed weighted input to the output of the neuron.
- It is called an activation function because it governs the threshold at which the neuron is activated and the strength of the output signal.
- A Simple step AF is where if the summed input was above a threshold, for example 0.5, then the neuron would output a value of 1.0, otherwise it would output a 0.0.

Artificial Neurons

- Networks of Neurons
- Neurons are arranged into networks of neurons.
- A row of neurons is called a layer and one network can have multiple layers.
- The architecture of the neurons in the network is often called the network topology.



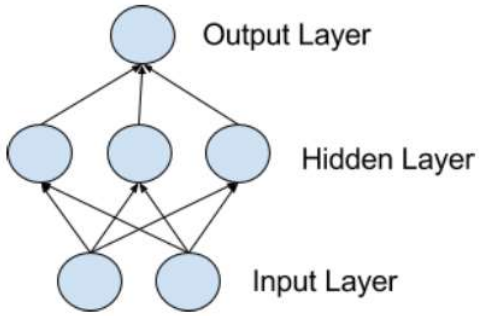
- Input or Visible Layers
- The bottom layer that takes input from your dataset is called the visible layer, because it is the exposed part of the network.
- Often a neural network is drawn with a visible layer with one neuron per input value or column in your dataset.
- These are not neurons as described above, but simply pass the input value though to the next layer.

- Hidden Layers
- Layers after the input layer are called hidden layers because they are not directly exposed to the input.
- The simplest network structure is to have a single neuron in the hidden layer that directly outputs the value.
- Deep learning can refer to having many hidden layers in your neural network.

Artificial Neurons

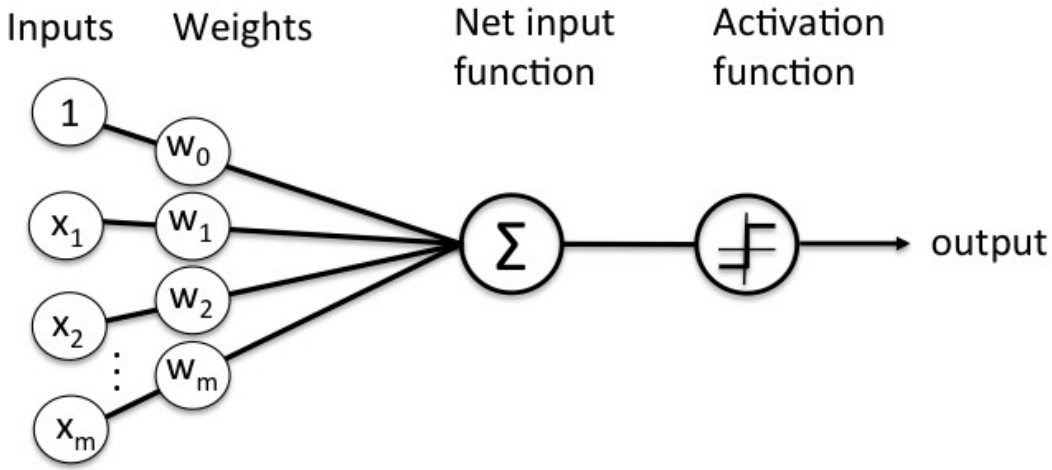
- Output Layer

- The final hidden layer is called the output layer and it is responsible for outputting a value or vector of values that correspond to the format required for the problem.
- The choice of activation function in the output layer is strongly constrained by the type of problem that you are modeling.



- For example:
- A regression problem may have a single output neuron and the neuron may have no activation function.
- A binary classification problem may have a single output neuron and use a sigmoid activation function to output a value between 0 and 1 to represent the probability of predicting a value for the primary class.
- This can be turned into a crisp class value by using a threshold of 0.5 and snap values less than the threshold to 0 otherwise to 1.
- A multiclass classification problem may have multiple neurons in the output layer, one for each class (e.g. 3 neurons for the 3 classes in the famous iris flowers classification problem).
- In this case a softmax activation function may be used to output a probability of the network predicting each of the class values.
- Selecting the output with the highest probability can be used to produce a crisp class classification value.

Single Layer Perceptron



Schematic of Rosenblatt's perceptron.

- The simplest of the neural network models, SLP, was designed by researchers McCulloch and Pitts.
- The SLP architecture is such that a single neuron is connected by many synapses, each of which contains a weight
- The weights affect the output of the neuron.
- The aggregate values of the weights multiplied by the input are then summed within the neuron and then fed into an activation function

Step 0. Initialization: $b = 0, w_i = 0, i = 1$ to n

Step 1. For each of the training sample (x,y), do the following:

Step 2. $w_i := w_i + x_i * y, i = 1$ to n /* update weight */
$b := b + x_i * y$ /* update bias */

Single Layer Perceptron

Let the vector of inputs $x = [x_1, x_2, \ldots, x_n]^t$ and the vector of weights $w = [w_1, w_2, \ldots, w_n]$.

The output of the function is given by

$$y = f\left(x, w^T\right)$$

where the activation function, when using a logistic function, is the following:

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

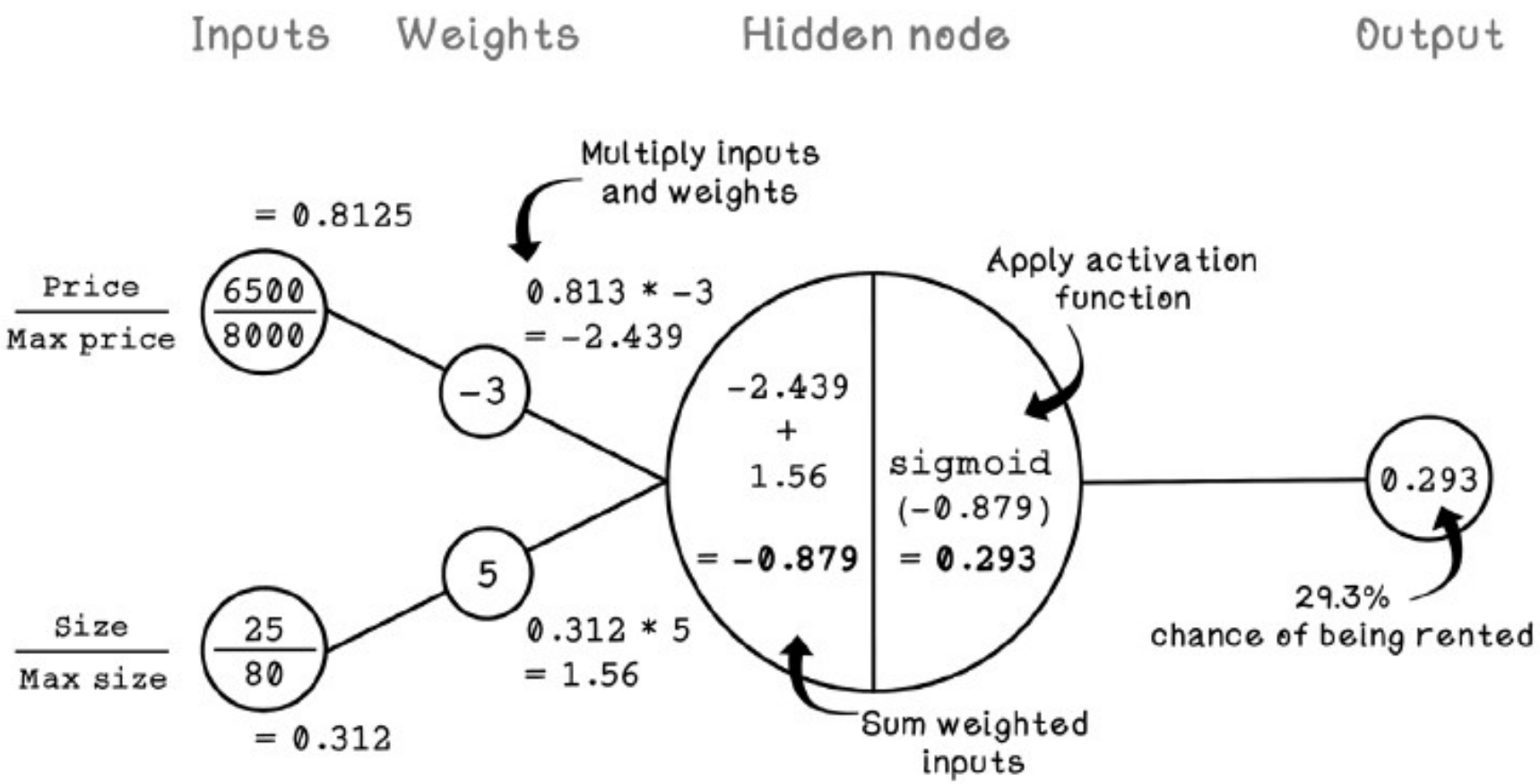
$$\hat{y} = f\left(x, w^T\right) = \sigma\left(\sum_i^n x_i w_i\right)$$

where

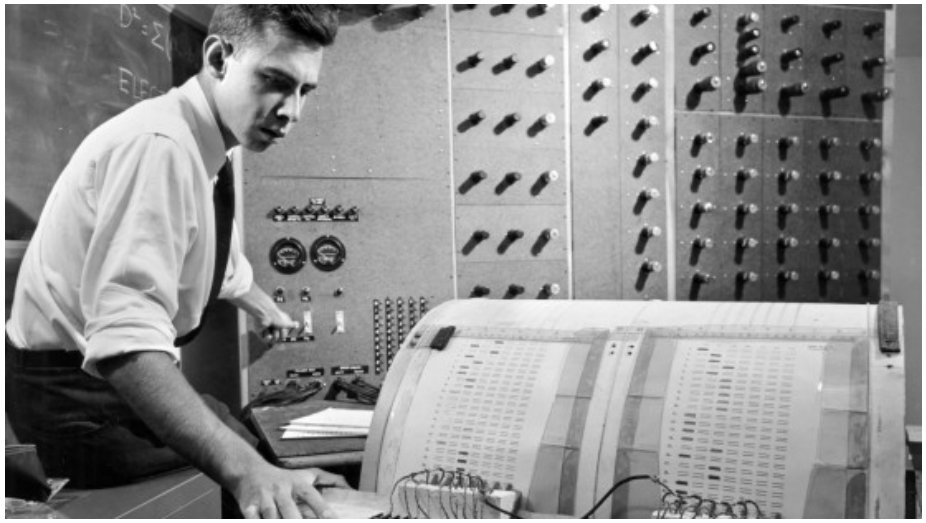
$$\sigma = \frac{1}{1+e^{-x}},$$

$$\hat{y} = \begin{cases} 1 & \text{if } y \geq \pi^*, \\ 0 & \text{elsewhere} \end{cases}$$

Single Layer Perceptron



Training the Perceptron



- Proposed by Rosenblatt in 1962
- Perceptron Algorithm is an iterative algorithm to learn the weight vector
- Basic idea is to Update the weights in proportion to the error contributed by the inputs.

Randomly initialize weight vector $\vec{w}_0$

Repeat until *error is less than a threshold* γ or max_iterations M:

For each training example $(\vec{x}_i, t_i)$:

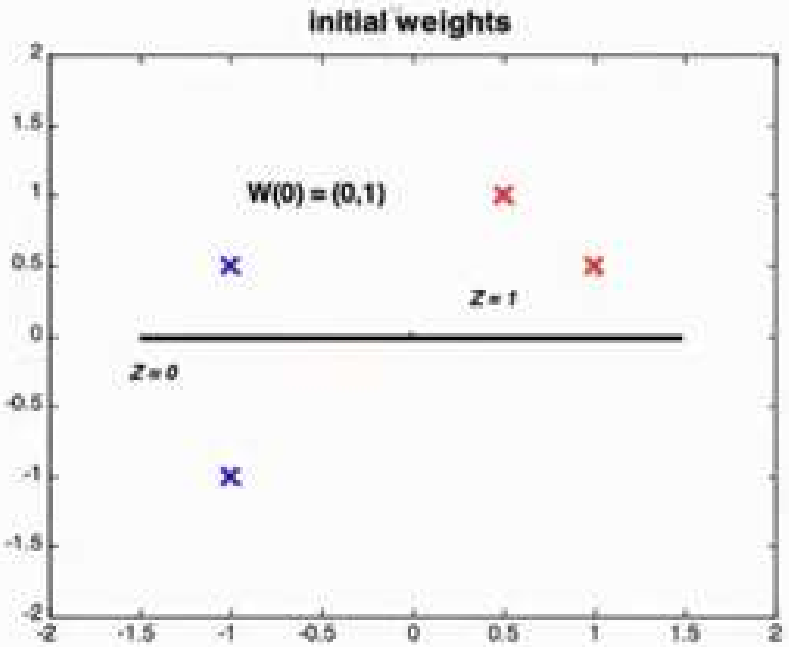
Predict output y_i using current network weights $\vec{w}_n$
Update weight vector as follows:

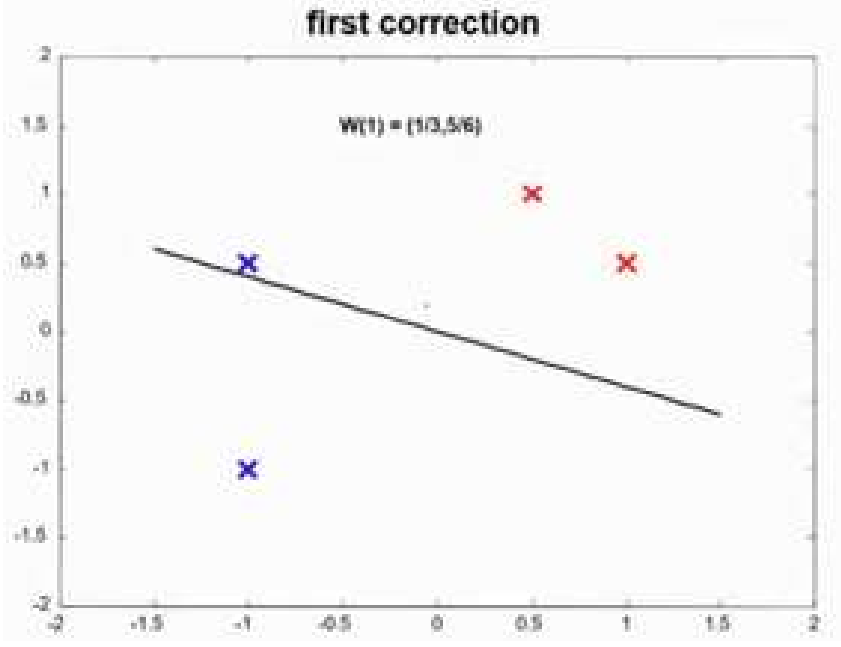
$$\vec{w}_{n+1} = \vec{w}_n + \eta * (t_i - y_i) * \vec{x}_i$$

Learning Rate

Error

Training the Perceptron - Example



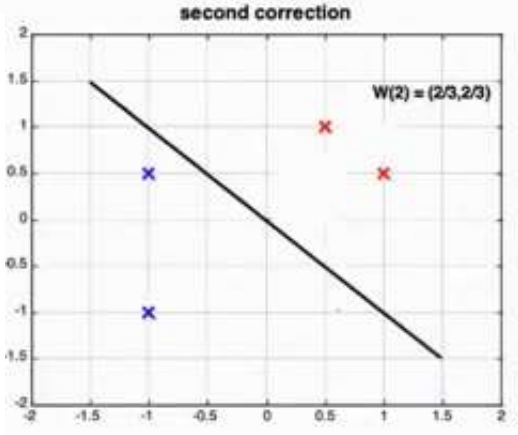


$\eta = 1/3$, $W(0) = (0,1)$

$$\overrightarrow{w_{n+1}} = \overrightarrow{w_n} + \eta \cdot (t_i - y_i) \cdot \vec{x}_i$$

W(1)_x = 0 + 1/3 *(-1) * (-1) = 1/3
W(1)_y = 1 + 1/3 * (-1) * (1/2) = 5/6

W(1) = (1/3,5/6)



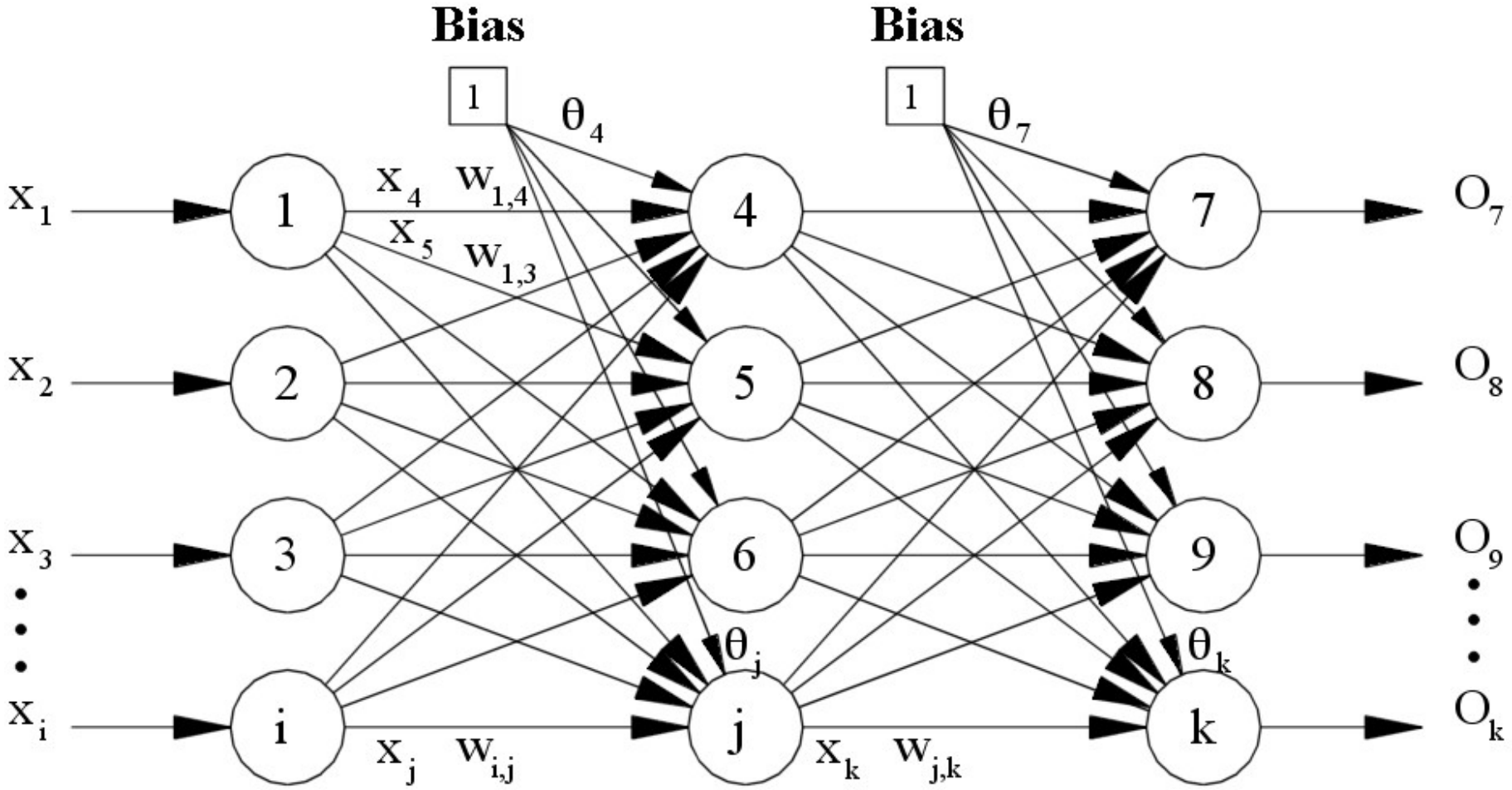
$\eta = 1/3$, $W(1) = (1/3,5/6)$

$$\overrightarrow{w_{n+1}} = \overrightarrow{w_n} + \eta \cdot (t_i - y_i) \cdot \vec{x}_i$$

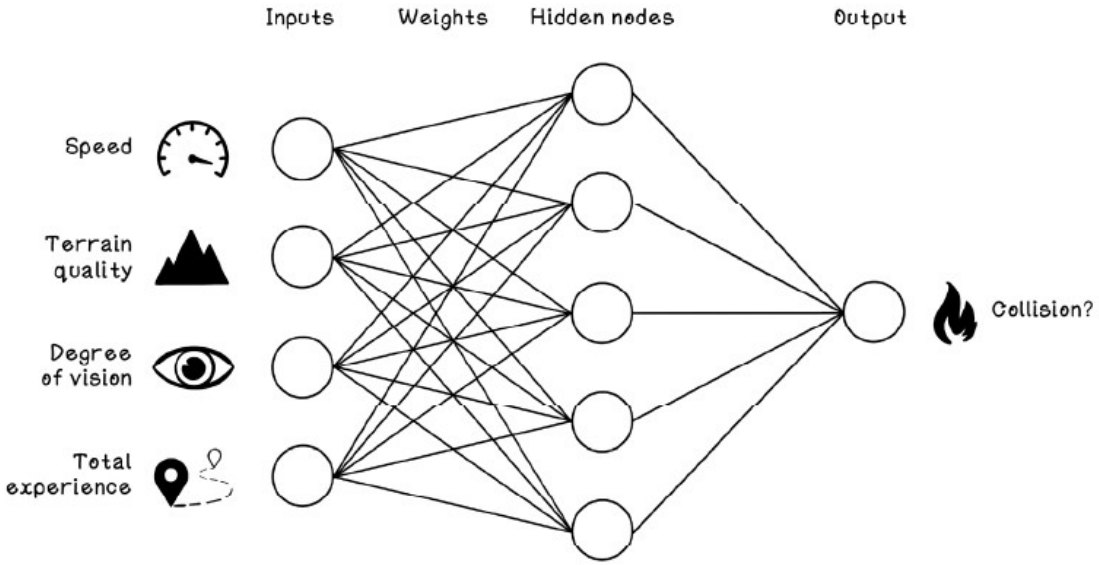
W(2)_x = 1/3 + 1/3 *(-1) * (-1) = 2/3
W(2)_y = 5/6 + 1/3 * (-1) * (1/2) = 4/6 = 2/3

W(2) = (2/3,2/3)

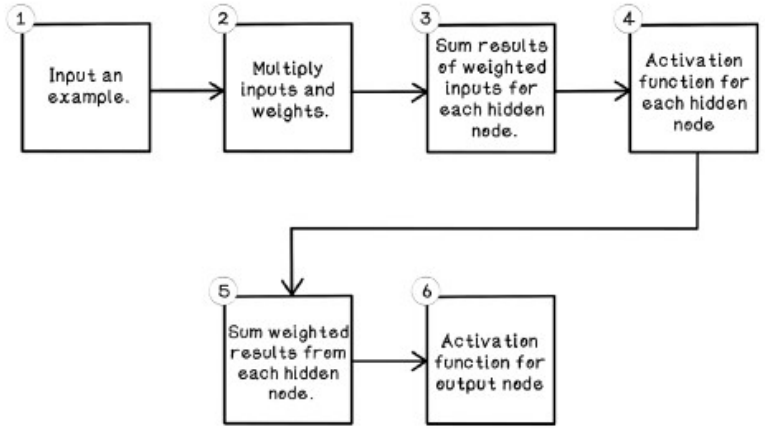
Multi Layered Perceptron – Dense Network

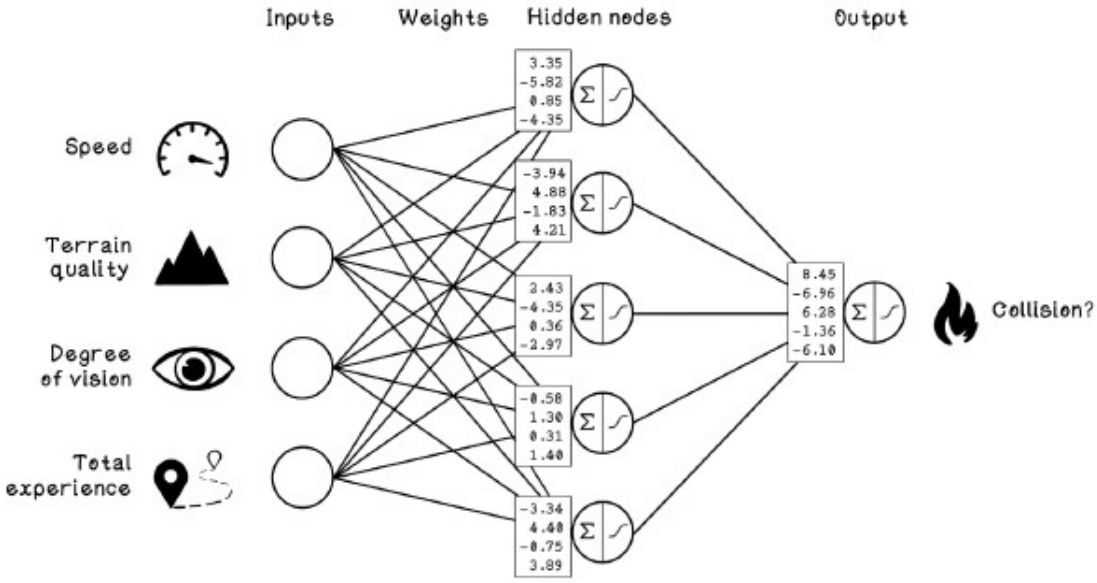


Multi Layered Perceptron



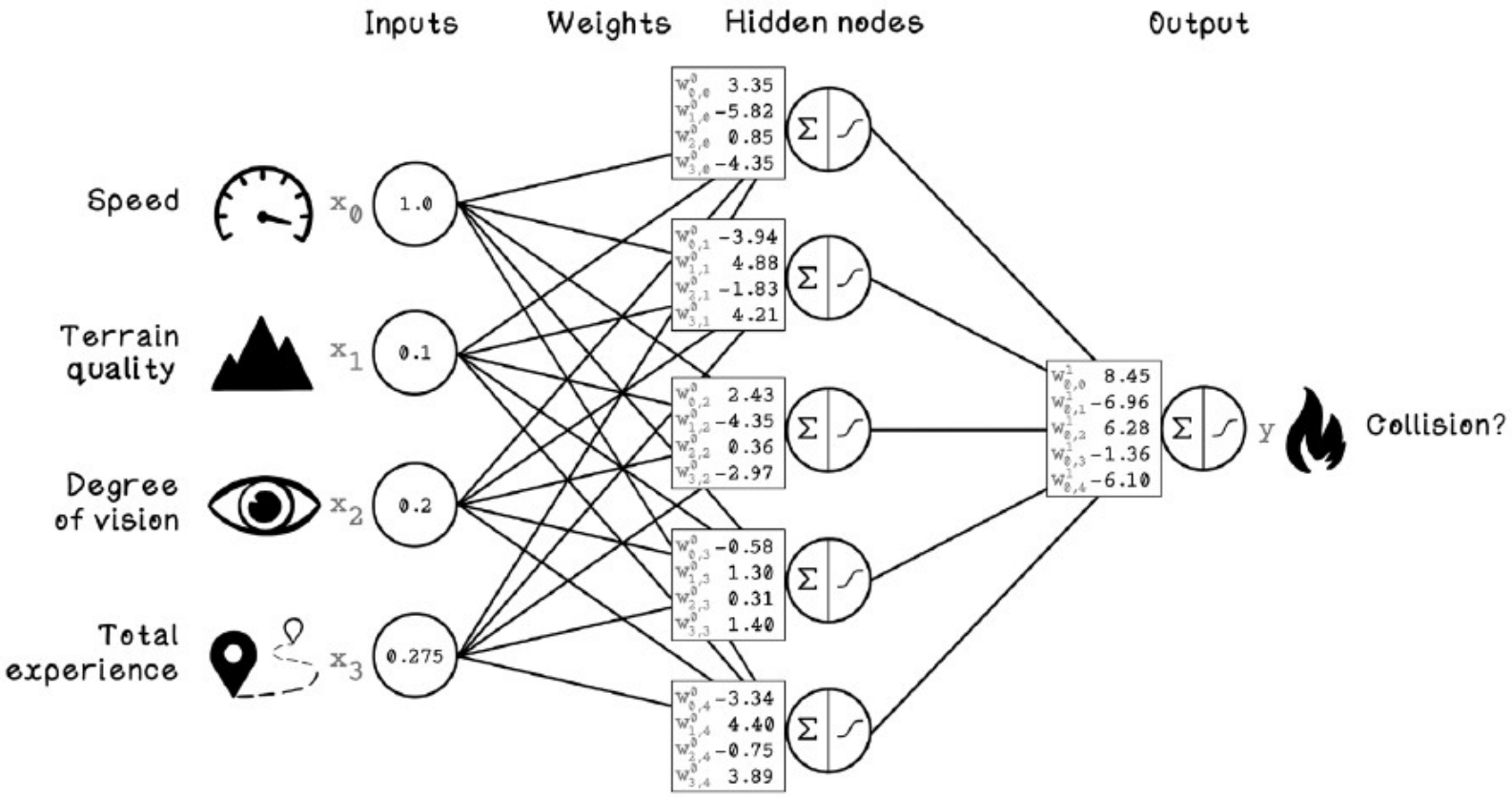
	Speed	Terrain quality	Degree of vision	Total experience	Collision occurred?
1	65 km/h	5/10	180°	80,000 km	No
2	120 km/h	1/10	72°	110,000 km	Yes
3	8 km/h	6/10	288°	50,000 km	No
4	50 km/h	2/10	324°	1,600 km	Yes
5	25 km/h	9/10	36°	160,000 km	No
6	80 km/h	3/10	120°	6,000 km	Yes
7	40 km/h	3/10	360°	400,000 km	No





CALCULATE THE PREDICTION FOR THE EXAMPLE BY USING FORWARD PROPAGATION WITH THE FOLLOWING ANN

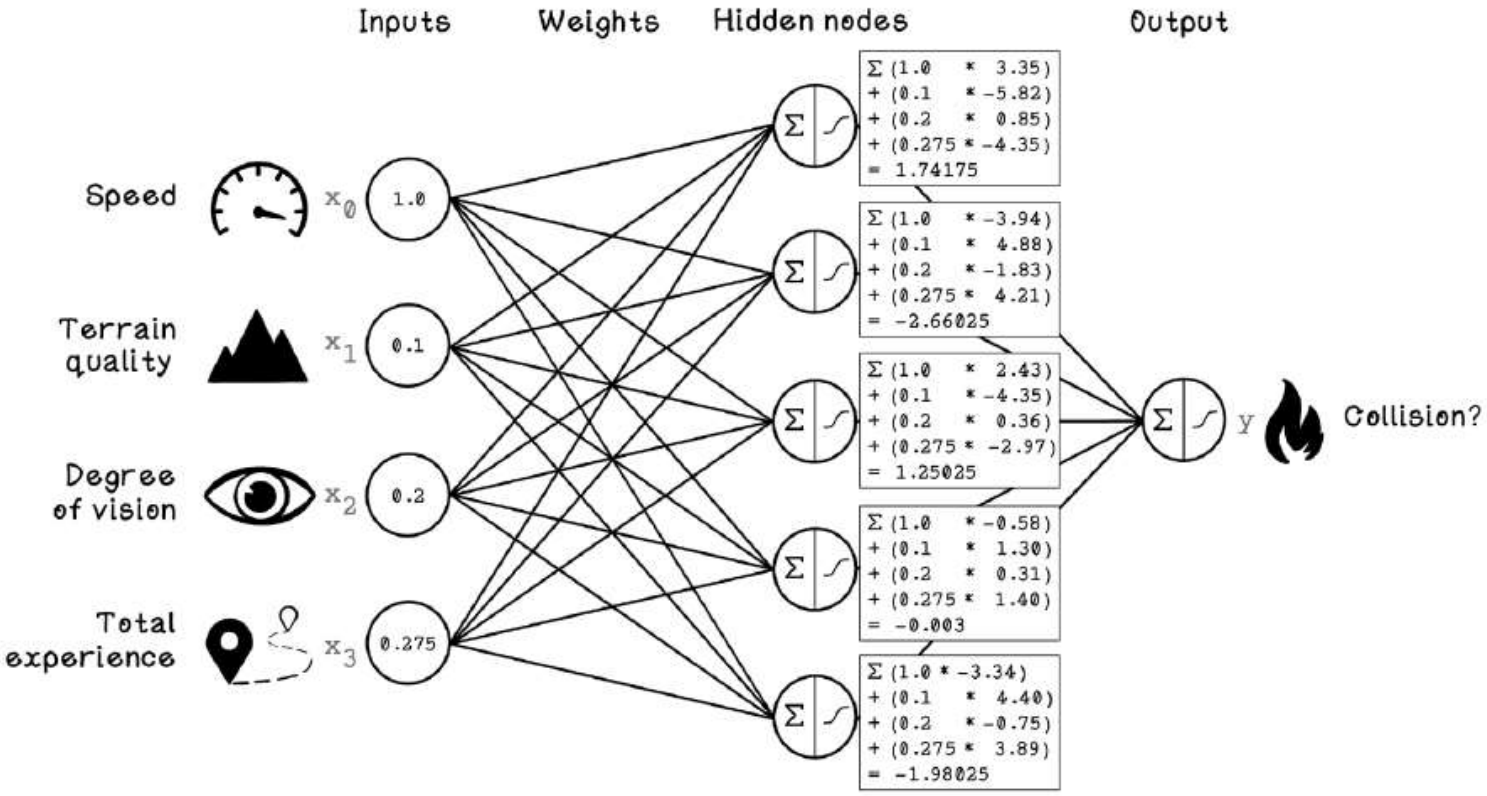
	Speed	Terrain quality	Degree of vision	Total experience	Collision occurred?
2	1.000	0.1	0.2	0.275	1



CALCULATE THE PREDICTION FOR THE EXAMPLE BY USING FORWARD PROPAGATION WITH THE FOLLOWING ANN

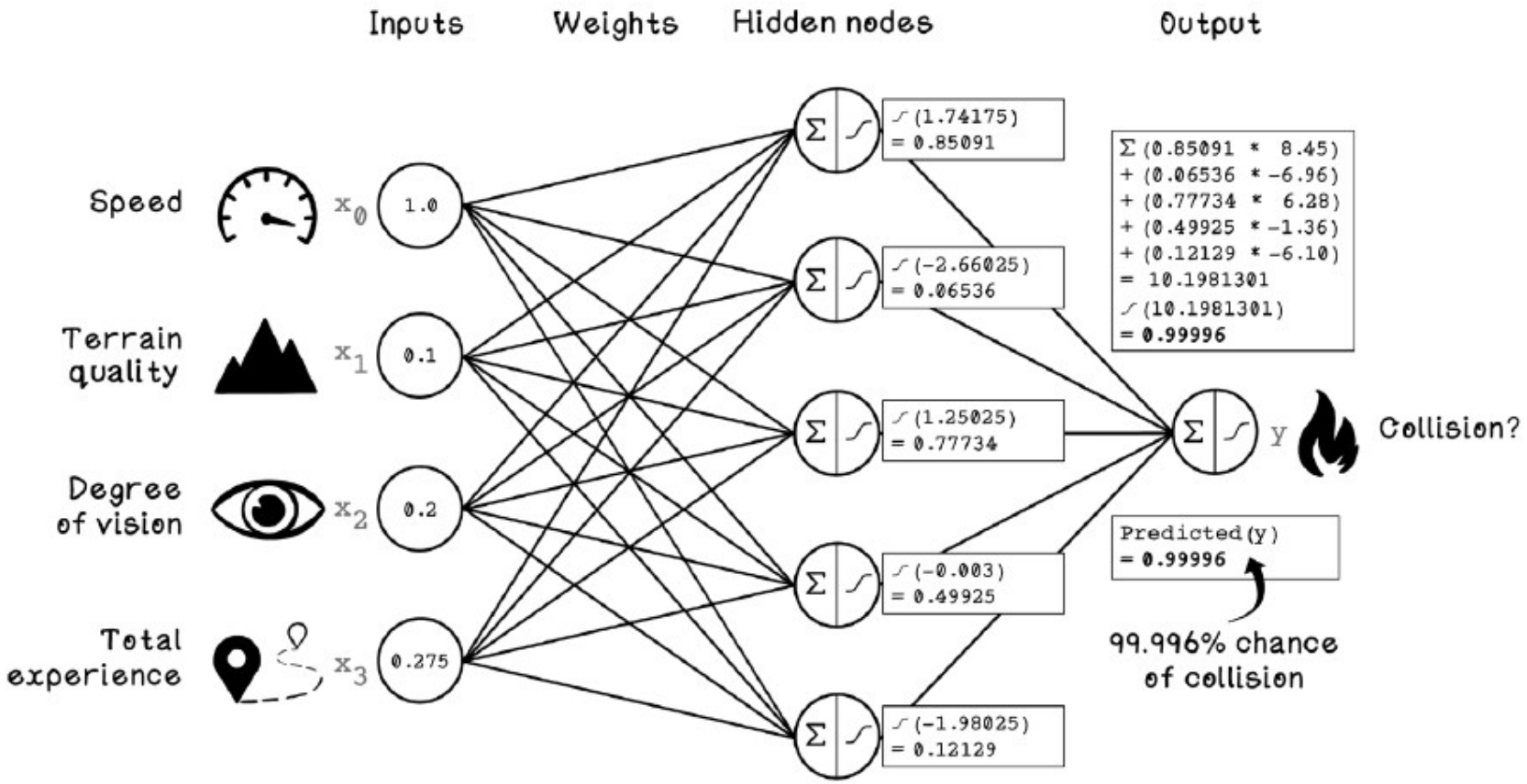
	Speed	Terrain quality	Degree of vision	Total experience	Collision occurred?
2	1.000	0.1	0.2	0.275	1

SOLUTION: CALCULATE THE PREDICTION FOR THE EXAMPLE BY USING FORWARD PROPAGATION WITH THE FOLLOWING ANN



CALCULATE THE PREDICTION FOR THE EXAMPLE BY USING FORWARD PROPAGATION WITH THE FOLLOWING ANN

	Speed	Terrain quality	Degree of vision	Total experience	Collision occurred?
2	1.000	0.1	0.2	0.275	1



Activation Functions

Commonly Used Activation Functions

Range

1. Step function : $f(z) = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases}$ — ① — $\{0, 1\}$

2. Signum function: $f(z) = \begin{cases} -1 & z < 0 \\ 0 & z = 0 \\ 1 & z > 0 \end{cases}$ — ② — $\{-1, 1\}$

3. Linear function: $f(z) = x$ — ③ — $(-\infty, \infty)$

4. ReLU function : $f(z) = \begin{cases} 0 & z < 0 \\ z & z \geq 0 \end{cases}$ — ④ — $(0, \infty)$

5. Sigmoid function: $f(z) = \frac{e^x}{1+e^x}$ — ⑤ — $(0, 1)$

6. Hyperbolic tan : $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ — ⑥ — $(-1, 1)$

by Dr. Pankaj Kumar Porwal (BTech - IIT Mumbai, PhD - Cornell University) : Principal, Techno India NJR Institute of Technology, Udaipur

Activation Functions

1) Linear Function :-

- Equation : Linear function has the equation similar to as of a straight line i.e. y = ax
- No matter how many layers we have, if all are linear in nature, the final activation function of last layer is nothing but just a linear function of the input of first layer.

- Range : -inf to +inf

- Uses : Linear activation function is used at just one place i.e. output layer.

- Issues : If we will differentiate linear function to bring non-linearity, result will no more depend on input “x” and function will become constant, it won’t introduce any ground-breaking behavior to our algorithm
- For example : Calculation of price of a house is a regression problem.
- House price may have any big/small value, so we can apply linear activation at output layer.
- Even in this case neural net must have any non-linear function at hidden layers.

Activation Functions

2) Sigmoid Function :-

It is a function which is plotted as ‘S’ shaped graph.

- Equation : A = 1/(1 + e-x)

- Nature : Non-linear. Notice that X values lies between -2 to 2, Y values are very steep.

- This means, small changes in x would also bring about large changes in the value of Y.

- Value Range : 0 to 1

- Uses : Usually used in output layer of a binary classification, where result is either 0 or 1, as value for sigmoid function lies between 0 and 1 only so, result can be predicted easily to be 1 if value is greater than 0.5 and 0 otherwise.

Activation Functions

3) Tanh Function :-

The activation that works almost always better than sigmoid function is Tanh function also knows as Tangent Hyperbolic function.

- It's actually mathematically shifted version of the sigmoid function.
- Both are similar and can be derived from each other.

- Equation :- f(x) = tanh(x) = 2/(1 + e-2x) - 1
- OR tanh(x) = 2 * sigmoid(2x) - 1
- Value Range :- -1 to +1
- Nature :- non-linear

- Uses :- Usually used in hidden layers of a neural network as it's values lies between -1 to 1 hence the mean for the hidden layer comes out be 0 or very close to it, hence helps in centering the data by bringing mean close to 0.

- This makes learning for the next layer much easier

Activation Functions

4) RELU :-

Stands for Rectified linear unit. It is the most widely used activation function.
Chiefly implemented in hidden layers of Neural network.

- Equation :- A(x) = max(0,x). It gives an output x if x is positive and 0 otherwise.
- Value Range :- [0, inf)
- Nature :- non-linear, which means we can easily backpropagate the errors and have multiple layers of neurons being activated by the ReLU function.

- Uses :- ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations.

- At a time only a few neurons are activated making the network sparse making it efficient and easy for computation.
- In simple words, RELU learns much faster than sigmoid and Tanh function.

Activation Functions

5)Softmax Function :-

The softmax function is also a type of sigmoid function but is handy when we are trying to handle classification problems.

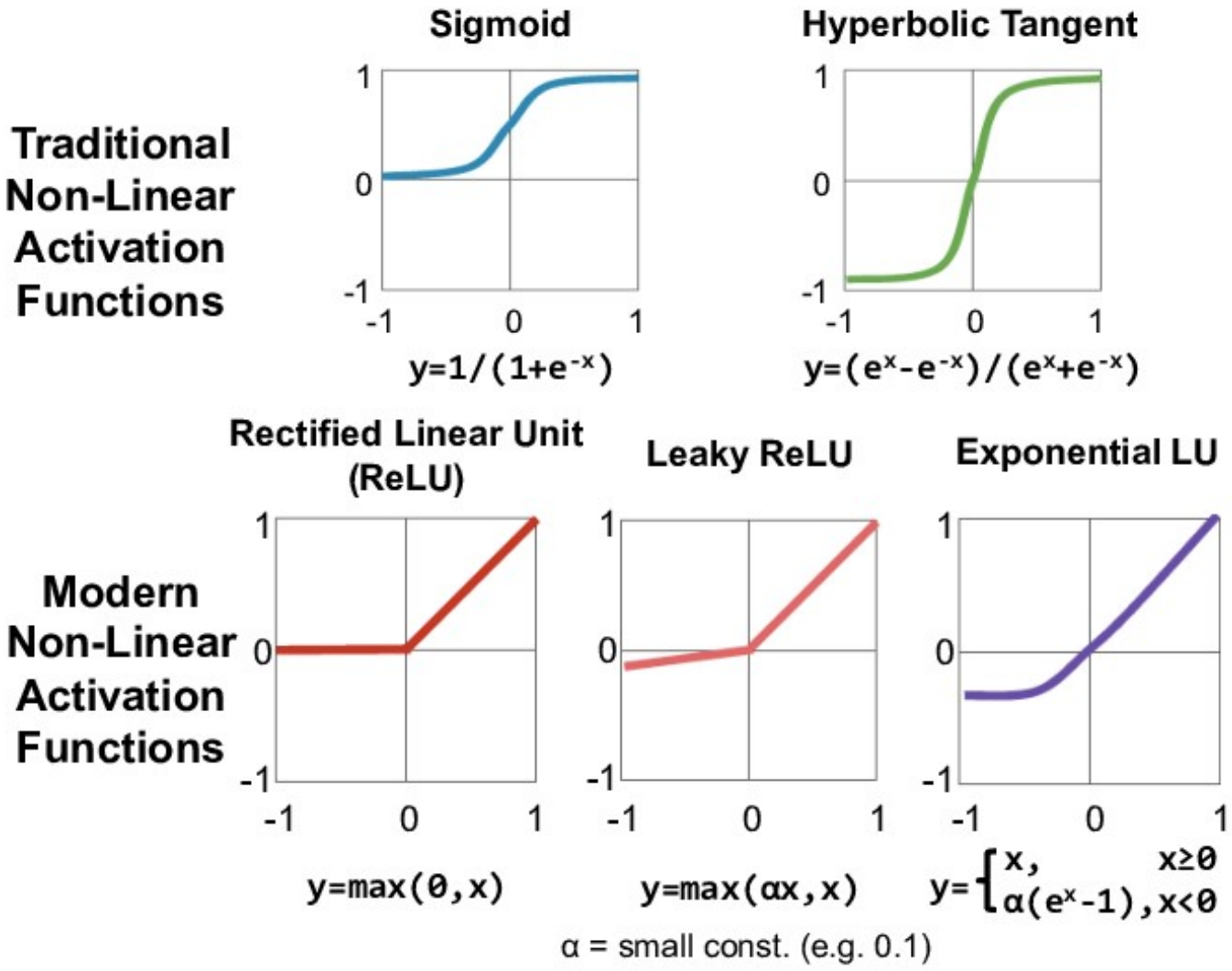
- Nature :- non-linear

- Uses :- Usually used when trying to handle multiple classes.

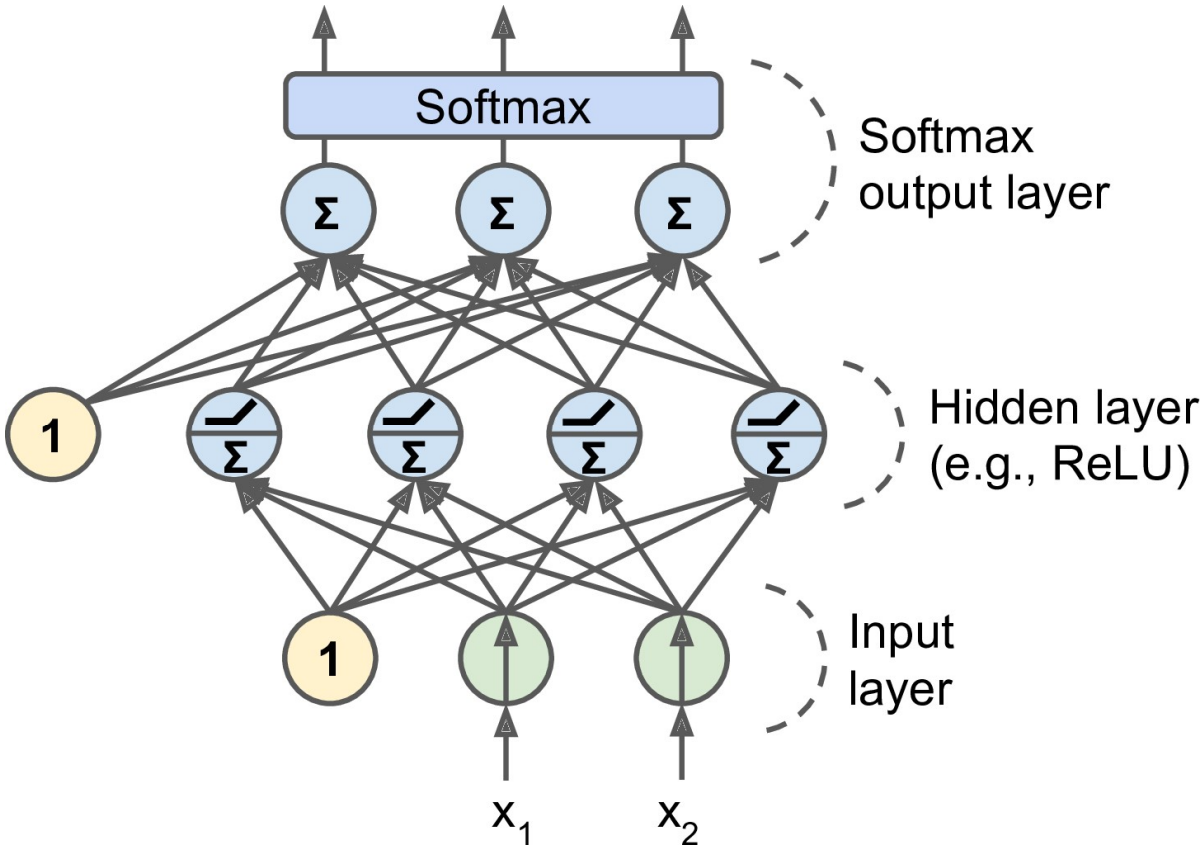
- The softmax function would squeeze the outputs for each class between 0 and 1 and would also divide by the sum of the outputs.

- Output:- The softmax function is ideally used in the output layer of the classifier where we are actually trying to attain the probabilities to define the class of each input.

Activation Functions

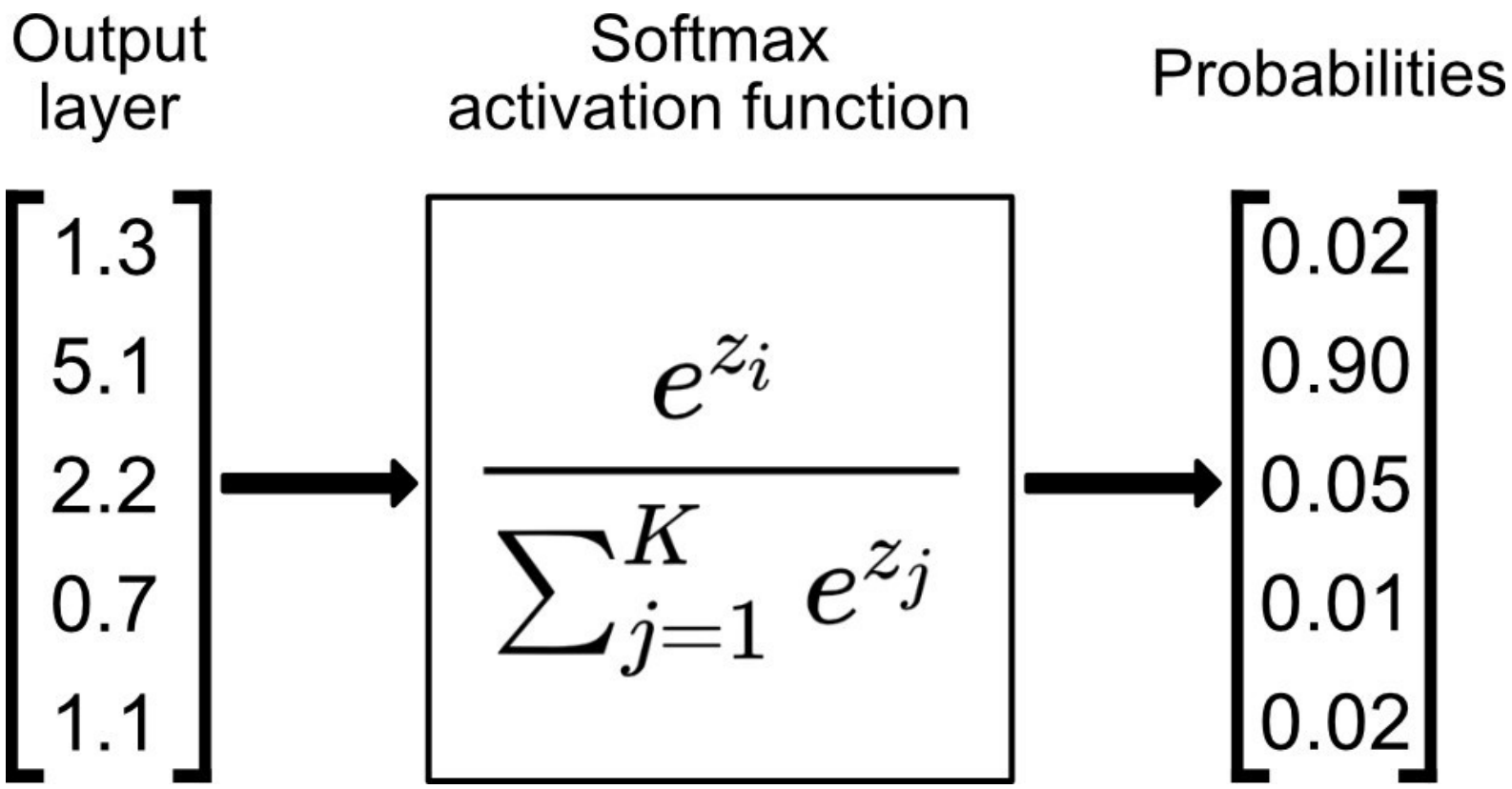


Softmax Activation Functions



- The softmax function is a function that turns a vector of K real values into a vector of K real values that sum to 1.
- The input values can be positive, negative, zero, or greater than one, but the softmax transforms them into values between 0 and 1, so that they can be interpreted as probabilities.
- If one of the inputs is small or negative, the softmax turns it into a small probability, and
- if an input is large, then it turns it into a large probability, but it will always remain between 0 and 1.

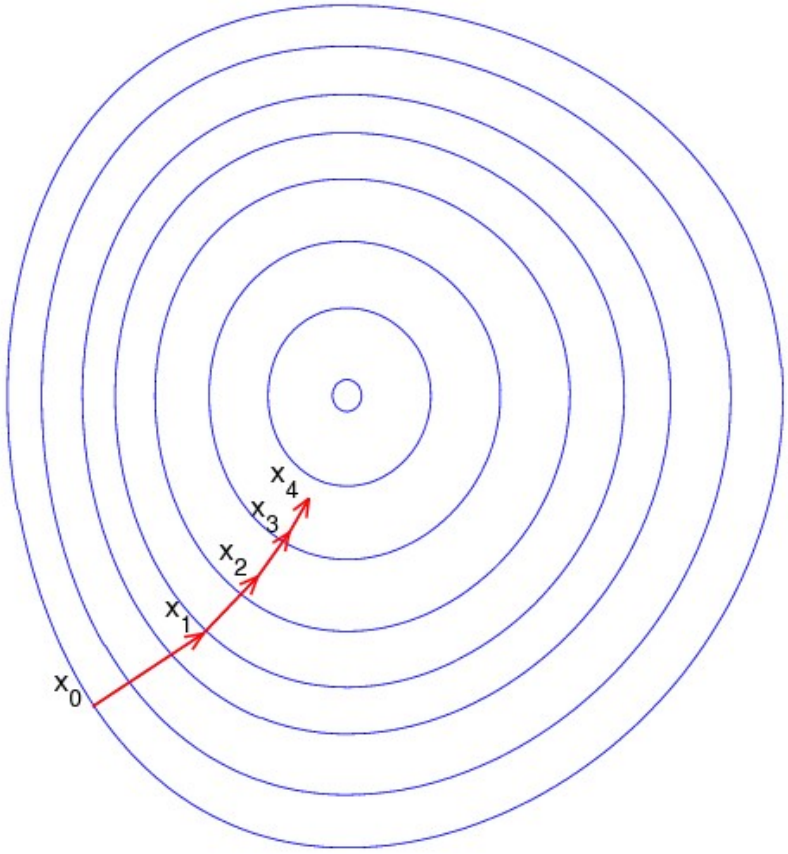
Softmax Activation Functions



Gradient Descent

WHAT IS GRADIENT DESCENT?

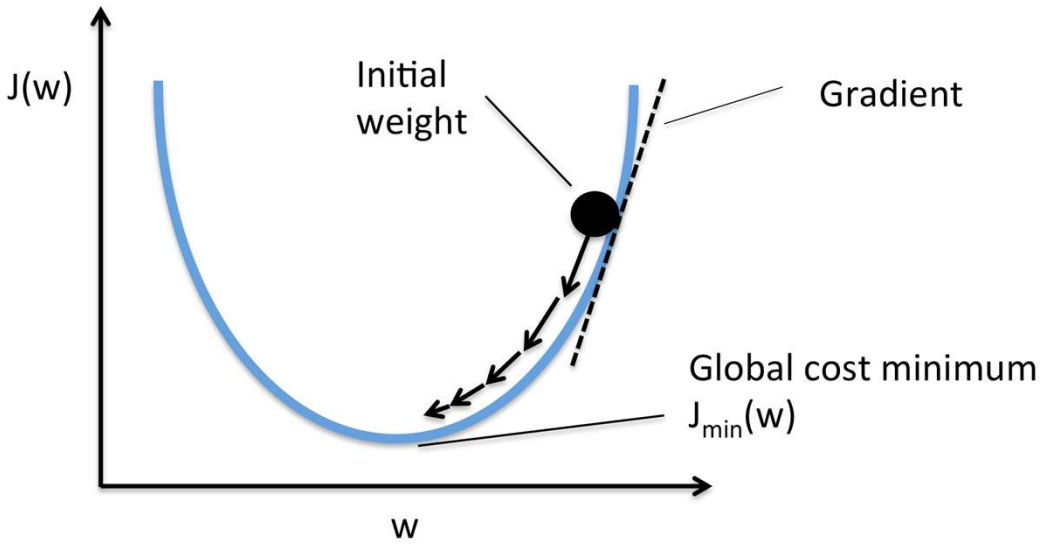
- Imagine a blindfolded man who wants to climb to the top of a hill with the fewest steps along the way as possible.
- He might start climbing the hill by taking really big steps in the steepest direction, which he can do as long as he is not close to the top.
- As he comes closer to the top, however, his steps will get smaller and smaller to avoid overshooting it.
- This process can be described mathematically using the gradient.
- Imagine the image illustrates our hill from a top-down view and the red arrows are the steps of our climber.
- Think of a gradient in this context as a vector that contains the direction of the steepest step the blindfolded man can take and also how long that step should be.
- Note that the gradient ranging from X0 to X1 is much longer than the one reaching from X3 to X4.
- This is because the steepness/slope of the hill, which determines the length of the vector, is less.



Gradient Descent

HOW GRADIENT DESCENT WORKS?

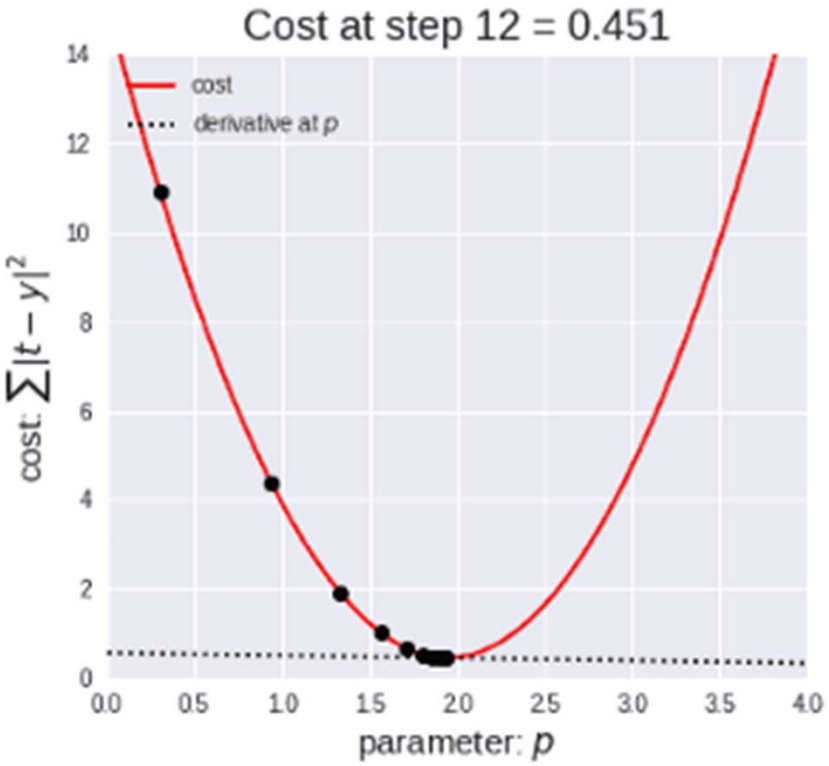
- Think of gradient descent as hiking down to the bottom of a valley.
- The minimization algorithm minimizes a given function.
- The equation below describes what gradient descent does:
- b is the next position of our climber, while **a** represents his current position.
- The minus sign refers to the minimization part of gradient descent.
- The gamma in the middle is a waiting factor and the gradient term (Δf(a)) is simply the

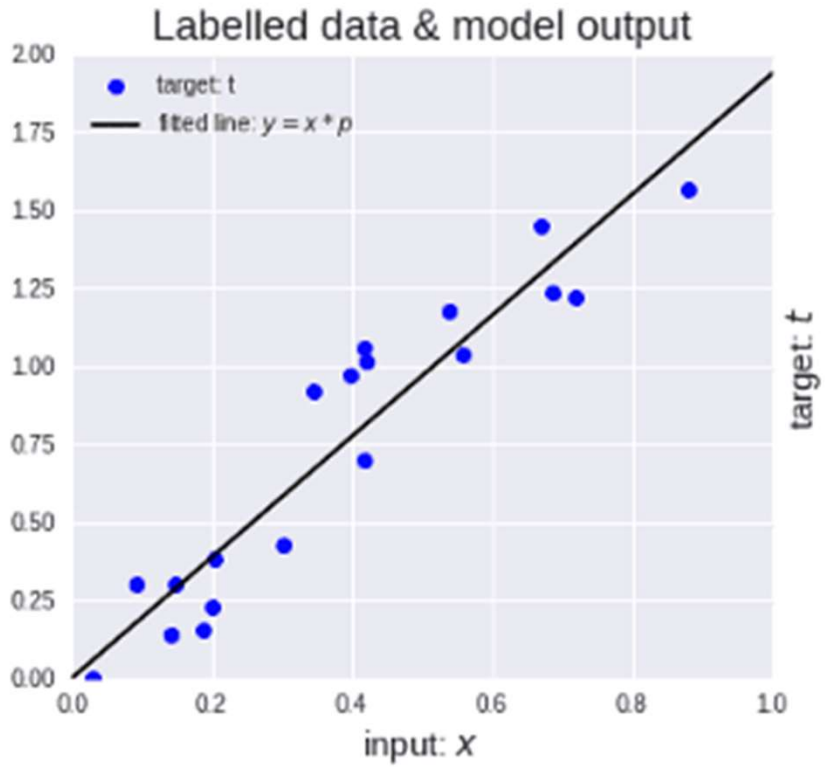


$$\mathbf{b} = \mathbf{a} - \gamma \nabla \mathbf{f}(\mathbf{a})$$

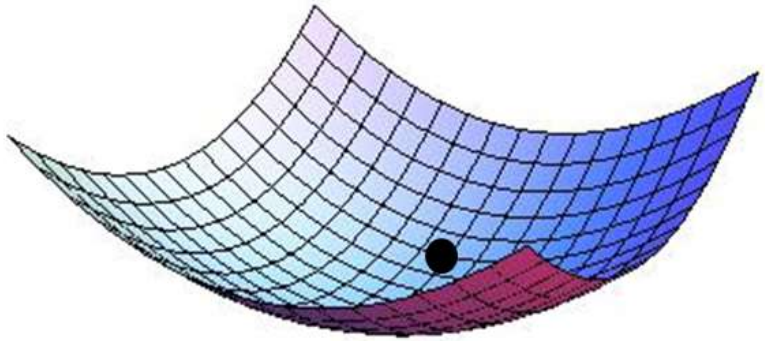
Gradient Descent

HOW GRADIENT DESCENT WORKS?

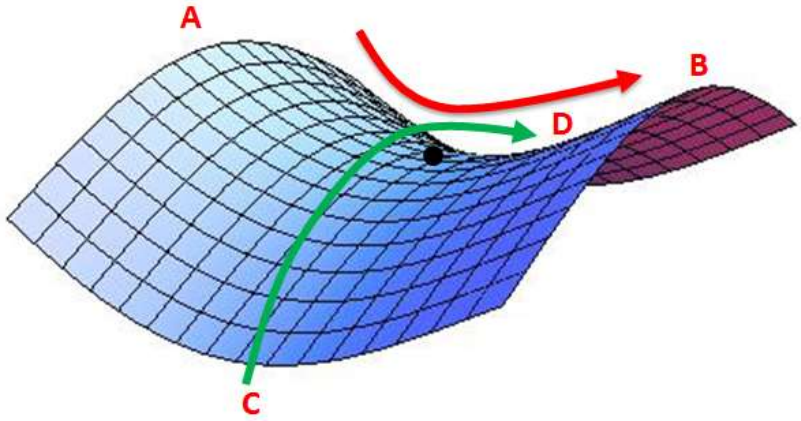




Gradient Descent – Saddle Point



Minima- Black dot placed on the PES shows a minimum energy point. Note how a PES resembles a well around the minimum point.

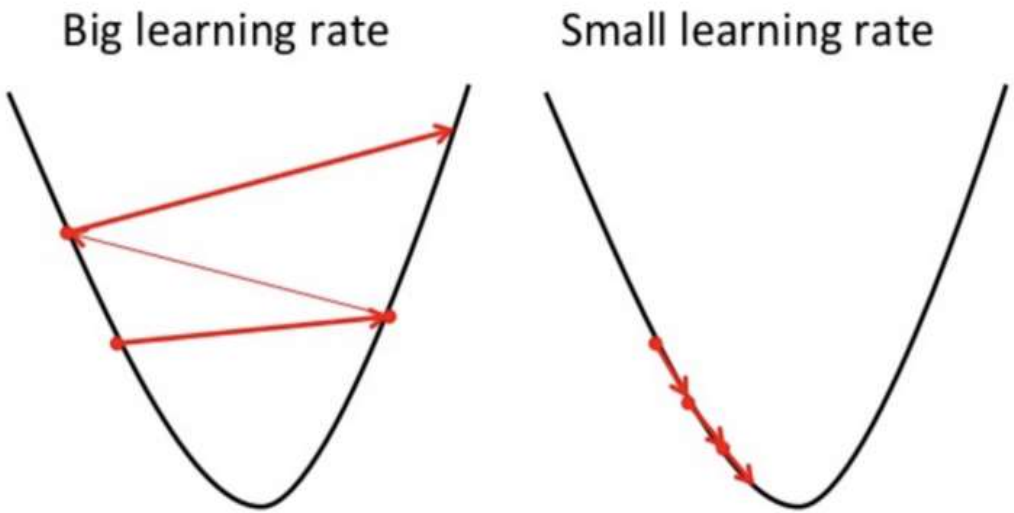


Saddle Point- Black dot placed on the PES shows a minima along path A-B and a maxima along path C-D. It represents a transition state along path C-D which, in this case, is the reaction coordinate.

Learning Rate

IMPORTANCE OF THE LEARNING RATE

- How big the steps are gradient descent taken into the direction of the local minimum are determined by the learning rate, which figures out how fast or slow we will move towards the optimal weights.
- For gradient descent to reach the local minimum we must set the learning rate to an appropriate value, which is neither too low nor too high.
- This is important because if the steps it takes are too big, it may not reach the local minimum because it bounces back and forth between the convex function of gradient descent (see left image below).
- If we set the learning rate to a very small value, gradient descent will eventually reach the local minimum but that may take a while (see the right image).



Gradient Descent - Methods

- **Batch gradient descent**

- Vanilla gradient descent, aka batch gradient descent, computes the gradient of the cost function for the entire training dataset

- As we need to calculate the gradients for the whole dataset to perform just one update, batch gradient descent can be very slow and is intractable for datasets that don't fit in memory.
- Batch gradient descent also known as off-line model

- **Stochastic gradient descent**
- Stochastic gradient descent (SGD) in contrast performs a parameter update for each training example, usually much faster and can also be used to learn online.

- **Mini-batch gradient descent**

- Mini-batch gradient descent finally takes the best of both worlds and performs an update for every mini-batch of n training examples
- Common mini-batch sizes range between 50 and 256, but can vary for different applications.
- Mini-batch gradient descent is typically the algorithm of choice when training a neural network

Gradient Descent

Epoch

An Epoch represents one iteration over the entire dataset.

Batch

We cannot pass the entire dataset into the neural network at once. So, we divide the dataset into number of batches.

Iteration

If we have 10,000 images as data and a batch size of 200, then an epoch should contain 10,000/200 = 50 iterations.

Gradient Descent

Epoch / Batch / Iteration

- ***One Epoch**: **One forward pass** and **one backward pass** of **all the training examples***
- ***Batch Size**: The **number** of training examples **in one forward/backward pass**. The higher the batch size, the more memory space you'll need.*
- ***Minibatch**: Take a **small number** of examples at a time, ranging from 1 to a few hundred, during **one iteration**.*
- ***Number of Iterations:** Number of passes, each pass using [batch size] number of examples. To be clear, **one pass = one forward pass + one backward pass**.*

$$1 \text{ epoch} = \text{numbers of iterations(N)} = \frac{\text{total training examples}}{\text{batch size(minibatch size)}}$$

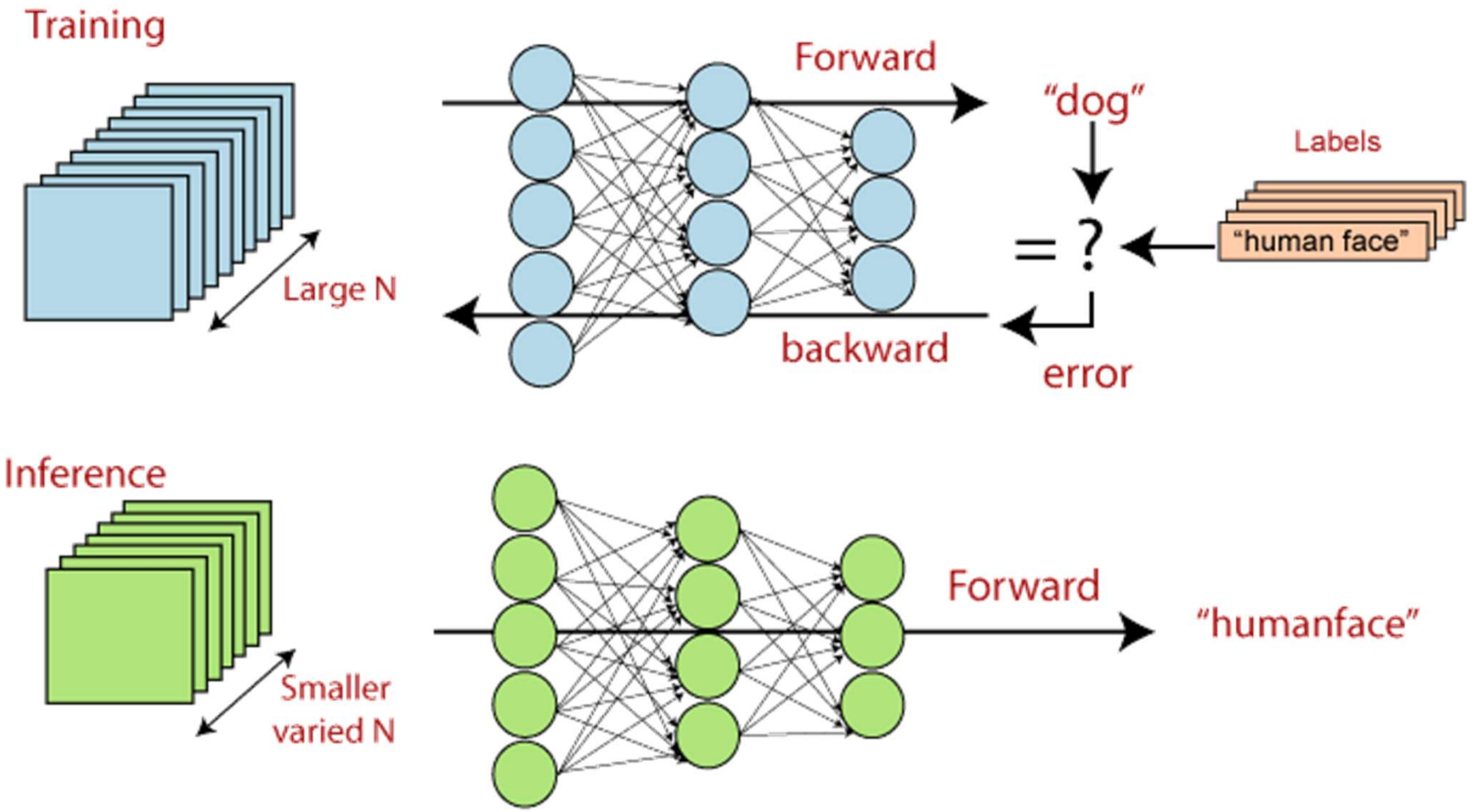
Gradient Descent

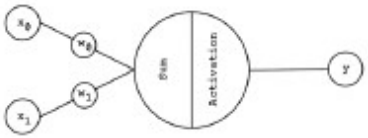
Trade-off

- Depending on the amount of data, they make a trade-off :
  - The **accuracy** of the parameter update
  - The **time** it takes to perform an update.

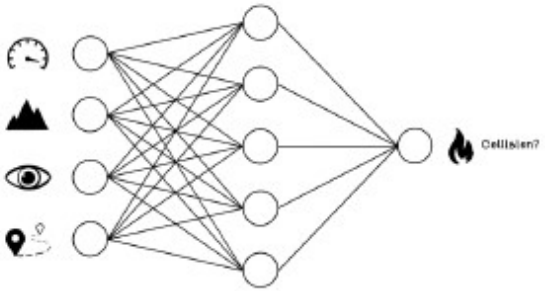
Method	Accuracy	Time	Memory Usage	Online Learning
Batch gradient descent	○	Slow	High	×
Stochastic gradient descent	△	High	Low	○
Mini-batch gradient descent	○	Midium	Midium	○

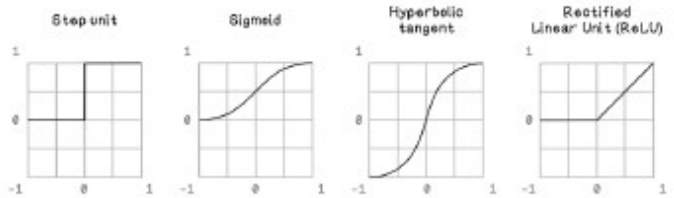
Deep Learning





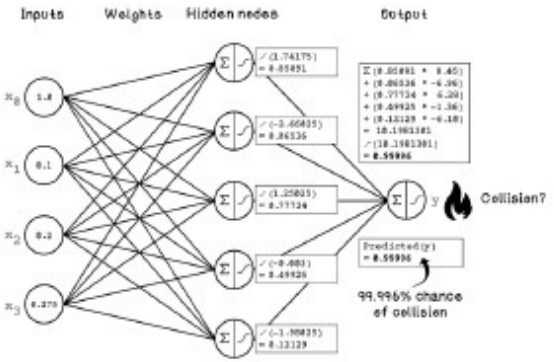
ANNs are based on the idea of the Perceptron.

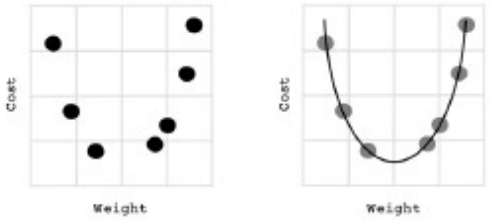




Activation functions help solve nonlinear problems.

Forward propagation is used to use the ANN to make predictions and is also used in training.





Gradient descent optimization is one of many weight optimization options.

Summary

Y	Cost Function	Error Function	O/P Activation Function
Continuous	ME, MSE, RMSE, MAE, MAPE	ME, MSE, RMSE, MAE, MAPE	Linear or Identity functions
Discrete in 2 categories	Binary Cross Entropy	Confusion Matrix / Cross table	Sigmoid or tanH
Discrete in > 2 categories	Categorical Cross Entropy	Confusion Matrix / Cross table	Softmax

Softmax	Sigmoid
Has to be an output layer only	Can be used in output and hidden layers
Gives the probability value of different classes	Value will be between 0 and 1
Multiclass classification	Binary class classification

	Classification	Regression
Hidden Layers	Sigmoid, tanh, Relu, Leaky Relu, ELU	Sigmoid, Relu, Leaky Relu, ELU
Output Layer	Sigmoid, tanh	Do not use any activation

Deep Learning

- **Why do we Need Bias in Neural Networks?**
- The term “bias” has a lot meanings.
- Unfair treatment, prejudice, discrimination, or favoring someone or something.
- And it’s natural. We live in a world where, unfortunately, all of these things take place.
- However, words have many meanings depending on the context, and surprisingly even bias can be something helpful.
- Machine Learning is a domain where we can meet bias in a couple of contexts.
- Let’s go through these meanings and find the one which makes Neural Networks useful.

Deep Learning

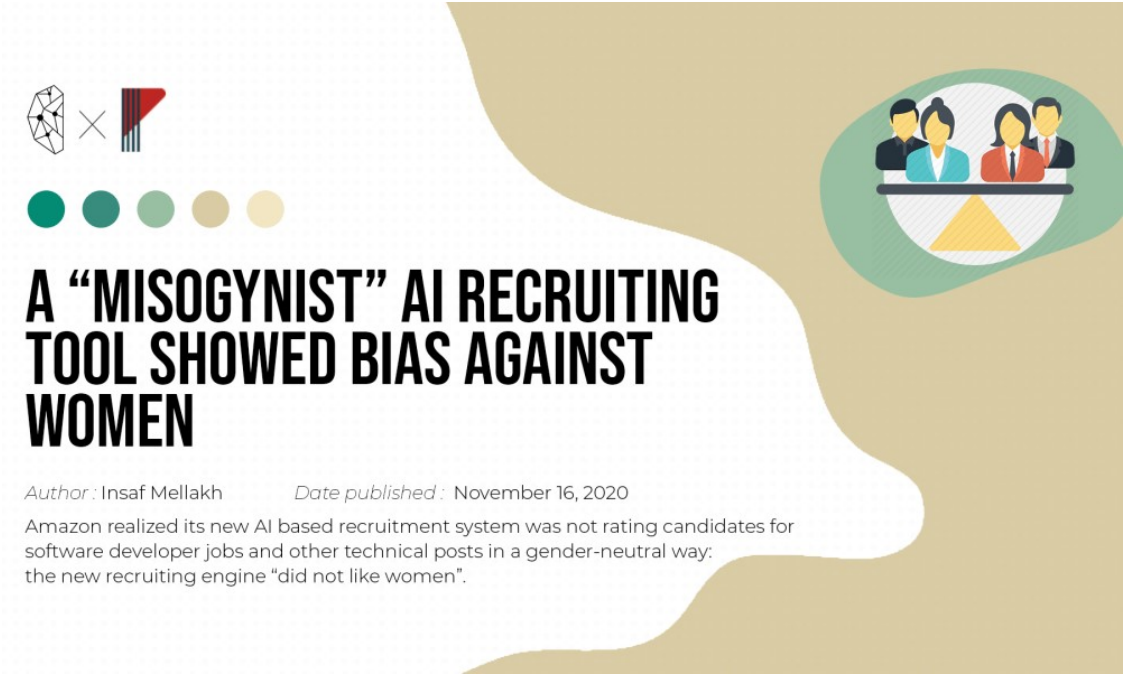
- **Why do we Need Bias in Neural Networks?**

- Let us discuss the most general context of bias.
- It's the bias inside the data used to train models.
- Every time we feed our Neural Network or other a model with data, it determines the model's behavior.
- We cannot expect fair treatment from algorithms that were built from biased data.

- One of the well-known examples of such biased data was Amazon recruiting tool.
- It was supposed to do some pre-filtering of resumes so recruiters could choose from the most promising ones.
- And it was great in filtering out resumes!
- Especially in filtering out female resumes…
- Unfortunately, the wonderful, AI-powered solution was biased.
- The system was favoring male candidates as engineers mainly used male resumes in the training process.

Deep Learning

- **Why do we Need Bias in Neural Networks?**



Deep Learning

- **Why do we Need Bias in Neural Networks?**

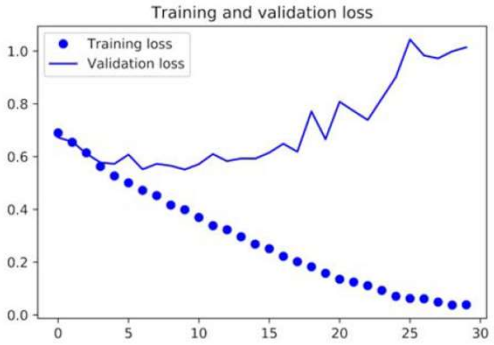
- Another example of a biased model is Tay.
- Tay was a chatbot released by Microsoft.
- It was supposed to carry conversations by posting Tweets.
- Tay was also able to learn from content posted by users.
- And that doomed Tay.
- It learned how to be offensive and aggressive.
- Tay became biased and switched off.
- Actually, it was made biased by irresponsible users who spoiled it with abusive posts.
- So biased data is definitely a negative phenomenon.
- Being responsible and aware of it is an important part of building models.
- When you create an artificial brain you must be careful what you put inside it.
- Otherwise, you may bring to life a monster.

Deep Learning

- **Why do we Need Bias in Neural Networks?**

https://www.techrepublic.com/article/why-microsofts-tay-ai-bot-went-wrong/

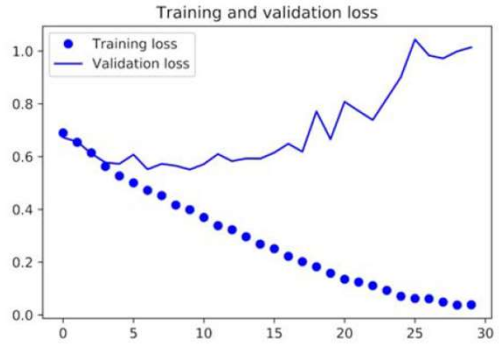
Deep Learning



- **Why do we Need Bias in Neural Networks?**

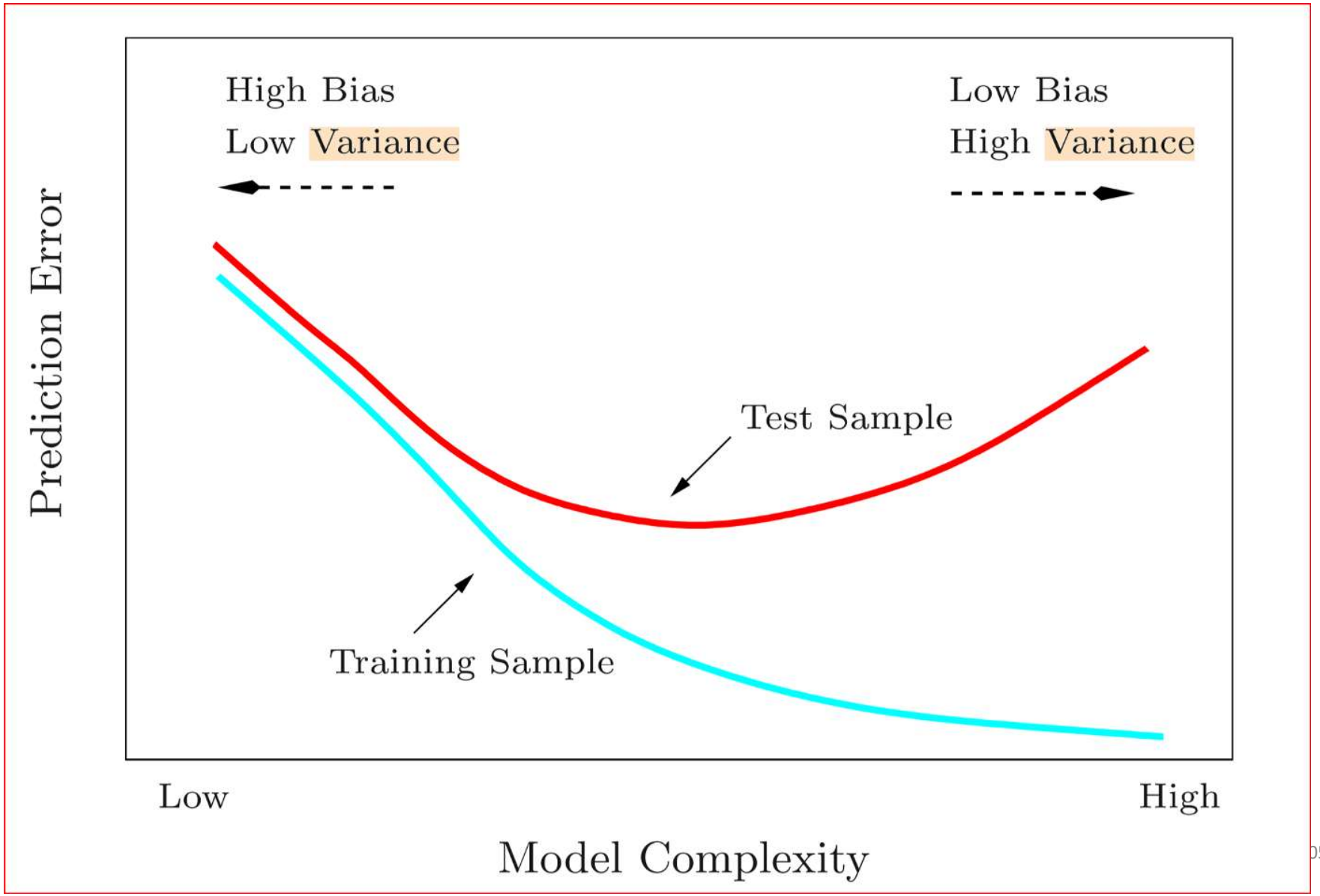
- **The bias of a whole Neural Network**
- Let's take a look at the second context of bias.
- When we train and test our Neural Networks or other Machine Learning models, we can observe two main trends:
- Model overfits to data.
- Model cannot learn patterns from data.

- Overfitting is like learning by heart.
- Your model did remember a vast majority of your training data, however, when something new comes up it doesn't work correctly.
- You can think of it as it's good at answering questions it's already been asked, but when you ask something out of the box the model fails.
- Such an issue can be nicely visualized if we plot validation and training set errors depending on the training set size.

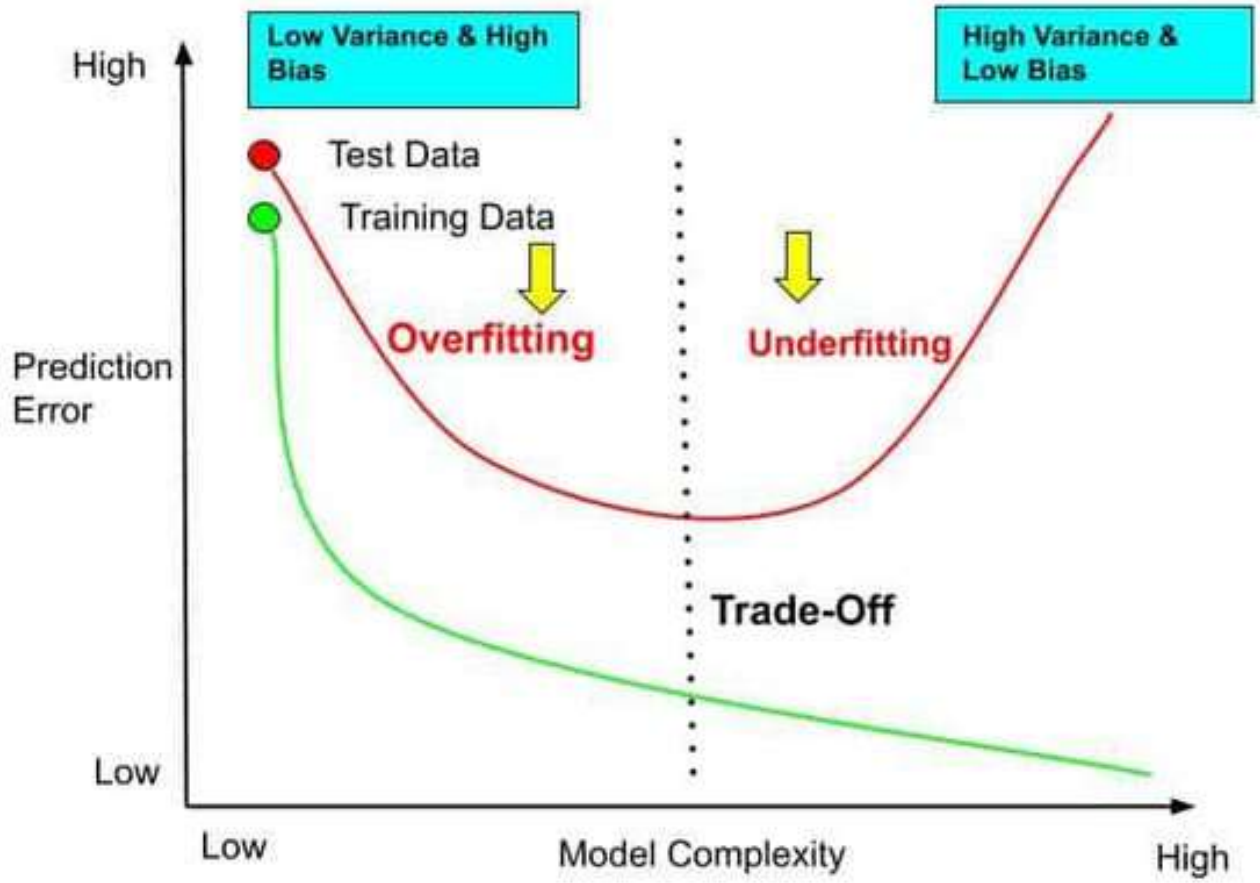


- **Why do we Need Bias in Neural Networks?**

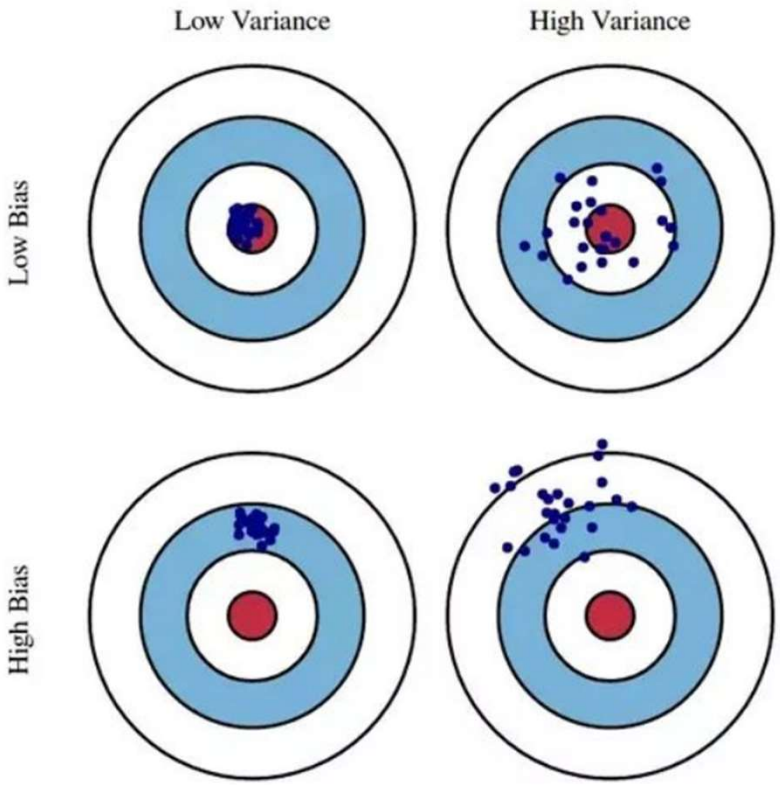
- **The bias of a whole Neural Network**
- If we get a relatively low error for the training set,
- but the error is high for validation set it means we have a high variance model.
- A big gap between validation and training set error values visible in the plot is specific to overfitting.
- Let's get back to the bias.
- When we speak about bias in the context of a model's performance we can say that model has a high bias.
- Basically, it means that the model doesn't do well during training, and what is expected, during validation.
- It behaves like a student that cannot grasp the idea we're trying to teach them.
- There might be something wrong with the model or with our data.


High Bias
Low Variance
Low Bias
High Variance
Prediction Error
Test Sample
Training Sample
Low
High
Model Complexity

Deep Learning



Deep Learning



- The usual analogy is target shooting or archery.
- **High bias is equivalent to aiming in the wrong place.**
- **High variance is equivalent to having an unsteady aim.**

- This can lead to the following scenarios:

1. Low bias, low variance: Aiming at the target and hitting it with good precision.

2. Low bias, high variance: Aiming at the target, but not hitting it consistently.

3. High bias, low variance: Aiming off the target, but being consistent.

4. High bias, high variance: Aiming off the target and being inconsistent.

- **Why do we Need Bias in Neural Networks?**

- **The bias of a whole Neural Network**
- When we take a look at learning curves plots and we see that the error is high for training set as well as for validation set, it may mean your model has high-bias.

- The gap between training and validation set curves will be small, as the model performs poorly in general.
- It lacks the ability to generalize and find patterns in data.
- High bias learning curves with small gap between them.

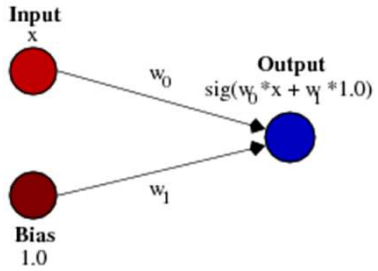
- High bias is also something bad.
- Adding more data probably won't help much.
- However, you can try to add extra features to data set samples.
- This additional information may give the model more clues while searching for patterns.

Deep Learning

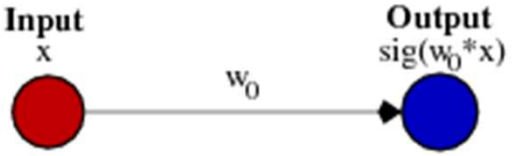
- **Why do we Need Bias in Neural Networks?**
- **The bias of a whole Neural Network**
- You may also need to change a model.
- Sometimes models are too rigid to learn from data.
- Think of non-linearly distributed data points, which look like a parabola.
- If you will try to fit a simple line to this parabola your model will fail due to high bias.
- In such a case, a more flexible model (like a quadratic equation), which has more parameters is needed.

- **Why do we Need Bias in Neural Networks?**

- **Bias as a single neuron**
- Let's analyze the third context, the bias in a particular Neural Network.
- "bias neuron".
- Why we need this special kind of neurons?
- Take a look at the picture:
- 2 circles depicting artificial neurons sending outputs to third circle, an output neuron.
- Simple Neural Network.
- This simple neural network consists of 3 types of neurons.
- Input neuron simply passes feature (x_1) from the data set.
- Bias neuron mimics additional feature, let's call it x_0.
- This additional input is always equal to 1.
- Finally, there is an output neuron, which is a full-fledged artificial neuron that takes inputs, processes them, and generates the output of the whole network.



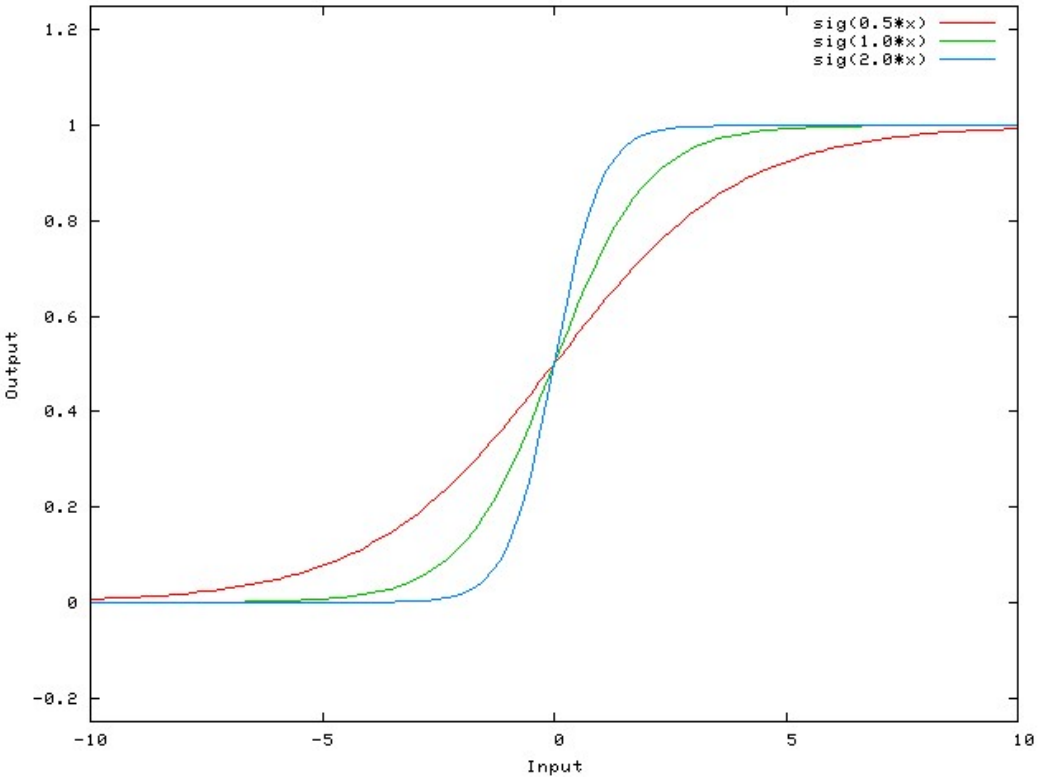
- **Why do we Need Bias in Neural Networks?**
- **Bias as a single neuron**
- In effect, **a bias value allows you to shift the activation function to the left or right**, which may be critical for successful learning.
- It might help to look at a simple example.
- Consider this 1-input, 1-output network that has no bias:



- The output of the network is computed by multiplying the input (x) by the weight (w_0) and passing the result through some kind of activation function (e.g. a sigmoid function.)

- **Why do we Need Bias in Neural Networks?**

- **Bias as a single neuron -** *Here is the function that this network computes, for various values of w_0:*

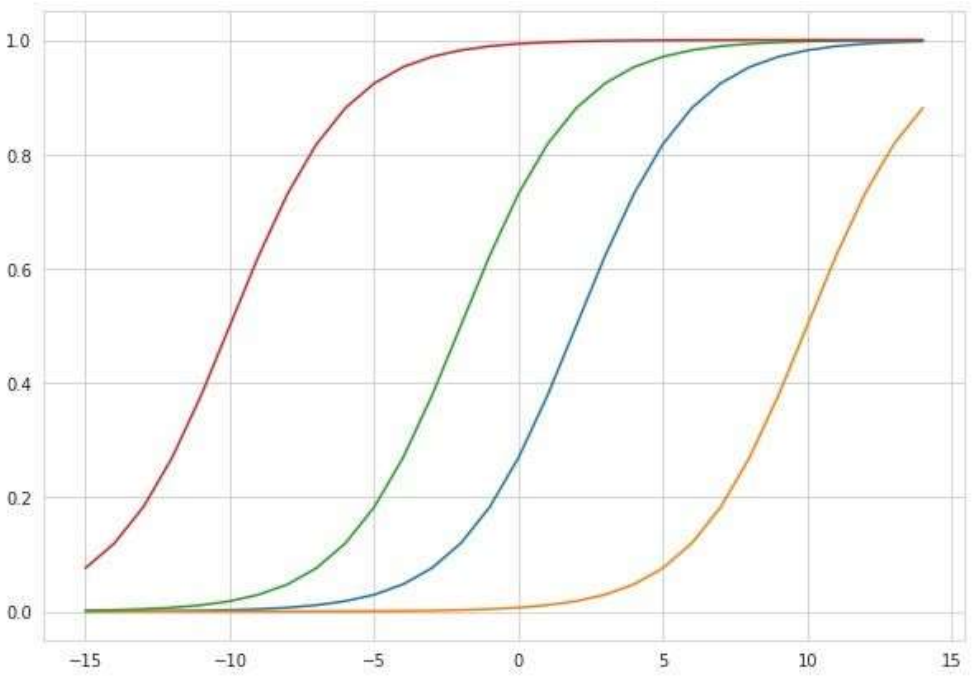


- Changing the weight w_0 essentially changes the "steepness" of the sigmoid.
- If we vary the values of the weight 'w', keeping bias 'b'=0, we will get the following graph
- That's useful, but what if you wanted the network to output 0 when x is 2?
- Just changing the steepness of the sigmoid won't really work -- **you want to be able to shift the entire curve to the right**.

Deep Learning

- **Why do we Need Bias in Neural Networks?**

- **Bias as a single neuron -** *Here is the function that this network computes, for various values of w_0:*

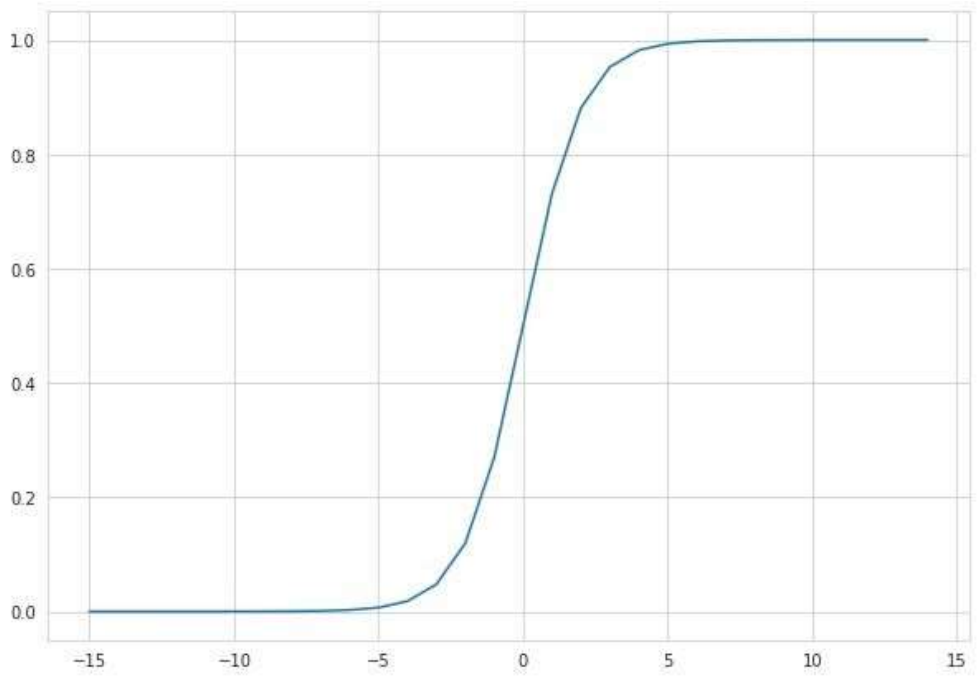


- While changing the values of 'w', there is no way we can shift the origin of the activation function, i.e., the sigmoid function.
- On changing the values of 'w', only the steepness of the curve will change.
- There is only one way to shift the origin and that is to include bias 'b'.
- On keeping the value of weight 'w' fixed and varying the value of bias 'b', we will get this graph.

From the graph, it can be concluded that the bias is required for shifting the origin of the curve to the left or right

Deep Learning

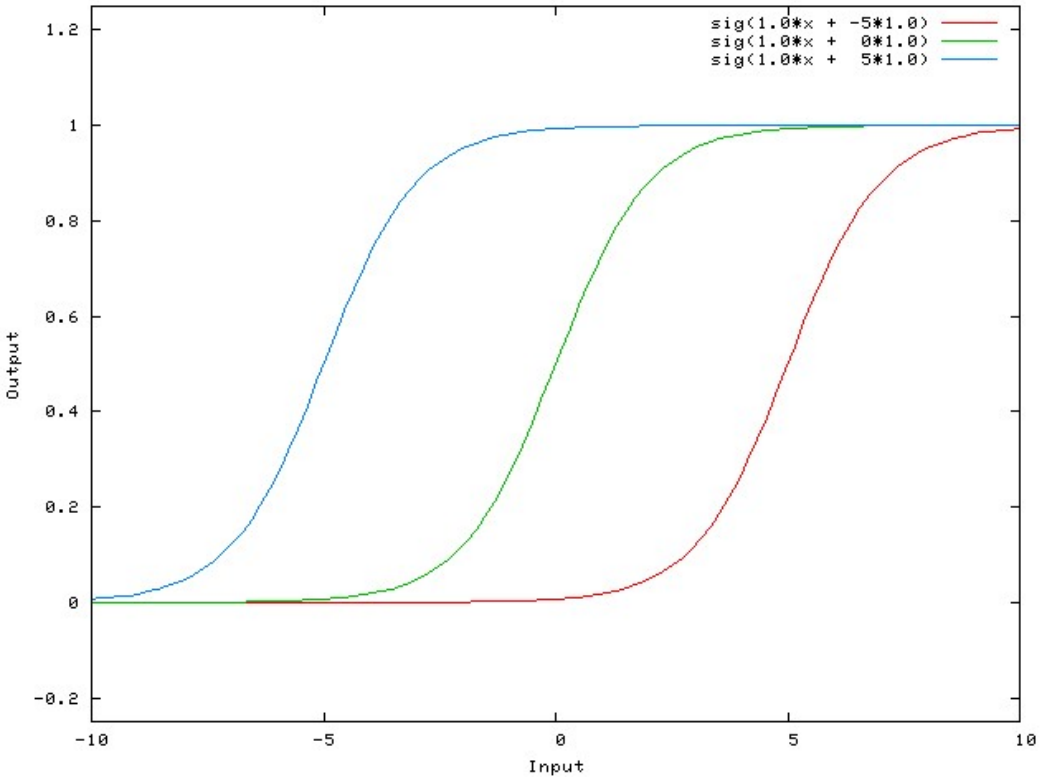
- **Why do we Need Bias in Neural Networks?**
- How to shift the curve to the left or the right?
- If we substitute the value of 'w'= -1 and 'b'=0 in, the graph of the activation function will look as shown below.



- It can be seen that the output is equal to 0 for the values of 'x' less than 0 and equal to 1 for the values of 'x' greater than 0.
- Now, how should we design our equation so that the output of the activation function is equal to 1 for all values of 'x' less than 5?
- We achieve this by introducing the term bias 'b' in our equation with value of 'b'=-5.

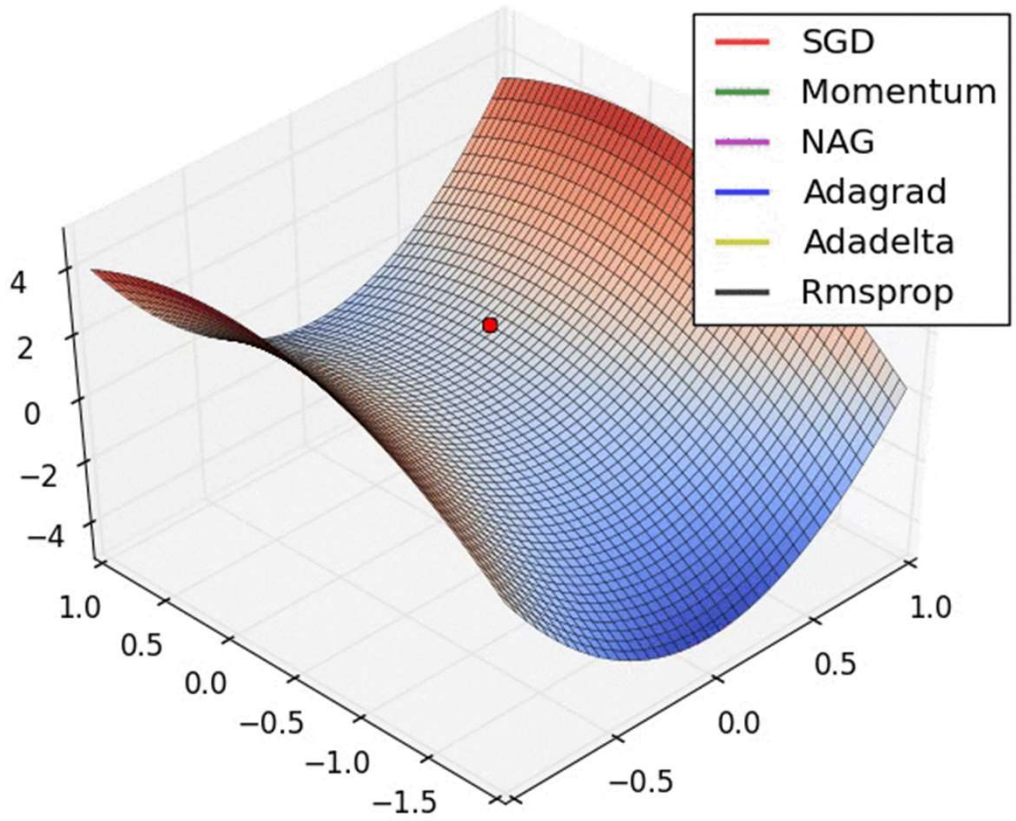
Deep Learning

- **Why do we Need Bias in Neural Networks?**
- How to shift the curve to the left or the right?.



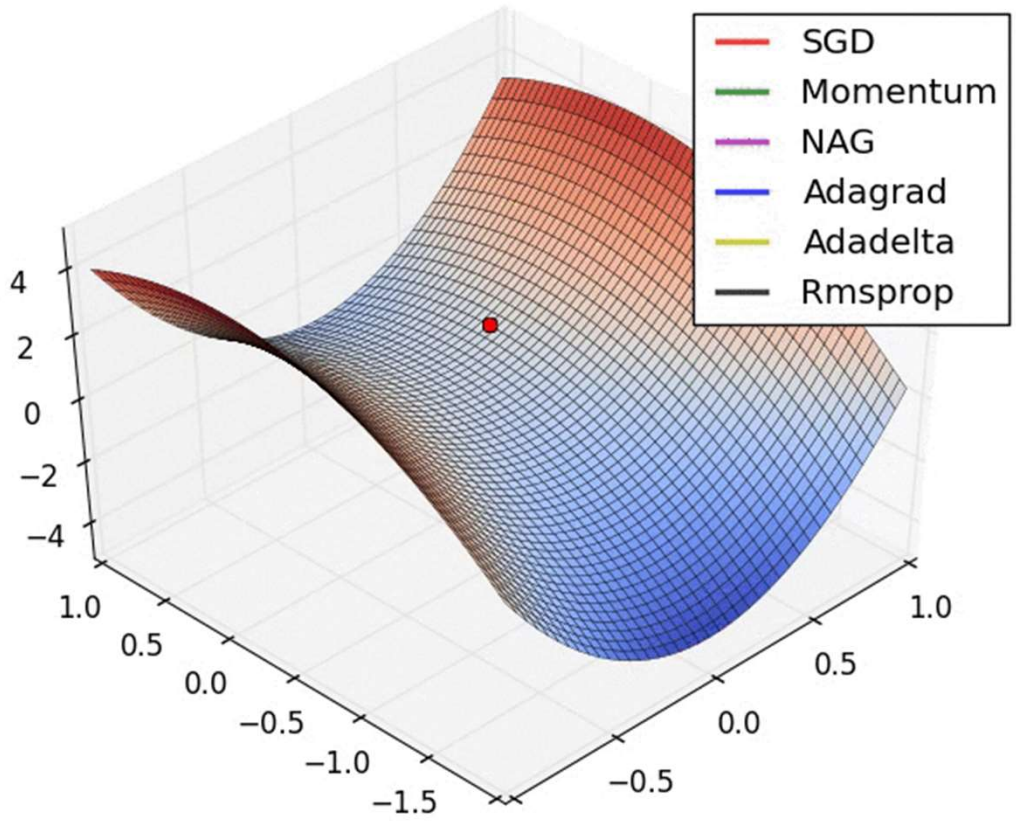
- Now, how should we design our equation so that the output of the activation function is equal to 1 for all values of ‘x’ less than 5?
- We achieve this by introducing the term bias ‘b’ in our equation with value of ‘b’=-5.

Optimization Algorithms in Neural Networks



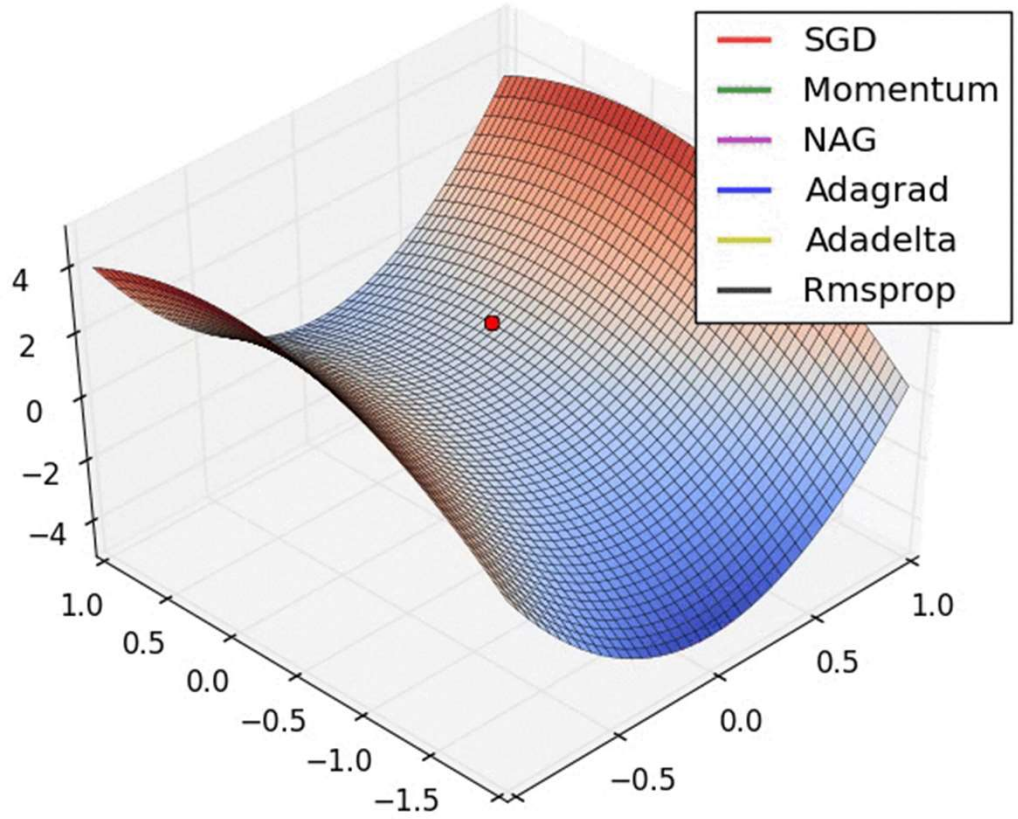
- **How do Optimizers work?**
- For a useful mental model, you can think of a hiker trying to get down a mountain with a blindfold on.
- It's impossible to know which direction to go in, but there's one thing she can know: if she's going down (making progress) or going up (losing progress).
- Eventually, if she keeps taking steps that lead her downwards, she'll reach the base.
- Similarly, it's impossible to know what your model's weights should be right from the start.
- But with some trial and error based on the loss function (whether the hiker is descending), you can end up getting there eventually.

Optimization Algorithms in Neural Networks



- **How do Optimizers work?**
- How you should change your weights or learning rates of your neural network to reduce the losses is defined by the optimizers you use.
- Optimization algorithms are responsible for reducing the losses and to provide the most accurate results possible.
- Various optimizers are researched within the last few couples of years each having its advantages and disadvantages.

Optimization Algorithms in Neural Networks



1. **Gradient Descent**
2. **Stochastic Gradient Descent (SGD)**
3. **Mini Batch Stochastic Gradient Descent (MB-SGD)**
4. **SGD with momentum**
5. **Nesterov Accelerated Gradient (NAG)**
6. **Adaptive Gradient (AdaGrad)**
7. **AdaDelta**
8. **RMSprop**
9. **Adam**

Deep Learning

- **Stochastic gradient descent**

- Given a differentiable function, it's theoretically possible to find its minimum analytically:
- It is known that a function's minimum is a point where the derivative is 0,
- So find all the points where the derivative goes to 0 and check for which of these points the function has the lowest value.

- Using the four-step algorithm:
- modify the parameters little by little based on the current loss value on a random batch of data.
- Because you're dealing with a differentiable function, you can compute its gradient, which gives you an efficient way to implement step 4.

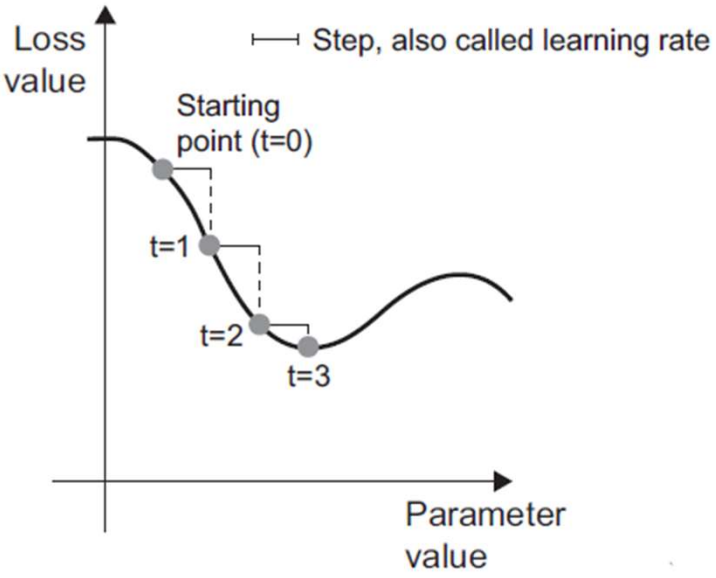
- If you update the weights in the opposite direction from the gradient, the loss will be a little less every time:

Deep Learning

- **Stochastic gradient descent**

- 1. Draw a batch of training samples x and corresponding targets y.
- 2. Run the network on x to obtain predictions y_pred.
- 3. Compute the loss of the network on the batch, a measure of the mismatch between y_pred and y.
- 4. Compute the gradient of the loss with regard to the network's parameters (a backward pass).
- 5. Move the parameters a little in the opposite direction from the gradient—for example W -= step * gradient—thus reducing the loss on the batch a bit.
- What we just described is called mini-batch stochastic gradient descent (minibatch SGD).
- The term stochastic refers to the fact that each batch of data is drawn at random (stochastic is a scientific synonym of random).
- Figure illustrates what happens in 1D, when the network has only one parameter and you have only one training sample.

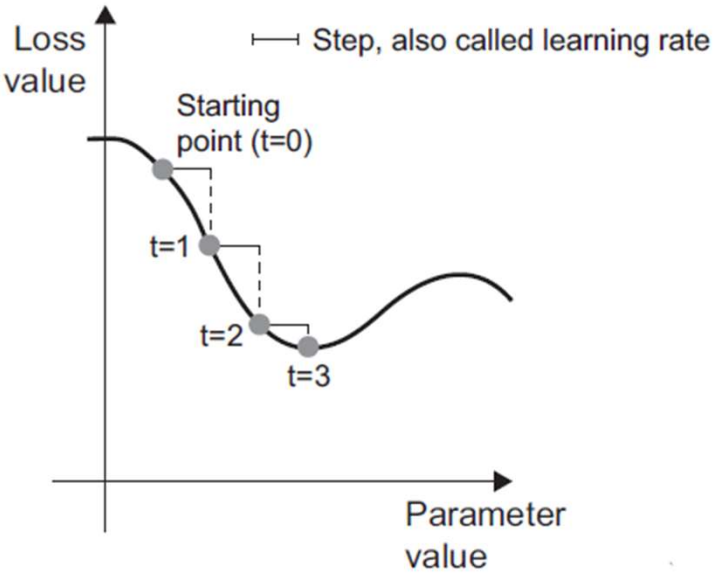
Deep Learning



- **Stochastic gradient descent**

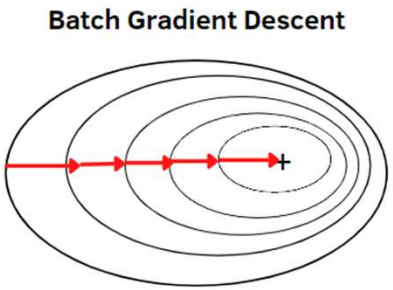
- Intuitively it's important to pick a reasonable value for the step factor.
- If it's too small, the descent down the curve will take many iterations, and it could get stuck in a local minimum.
- If step is too large, your updates may end up taking you to completely random locations on the curve.
- The variant of the mini-batch SGD algorithm would be to draw a single sample and target at each iteration, rather than drawing a batch of data.
- This would be *true* SGD (as opposed to *mini-batch* SGD).
- Alternatively, going to the opposite extreme, you could run every step on *all* data available, which is called *batch SGD*

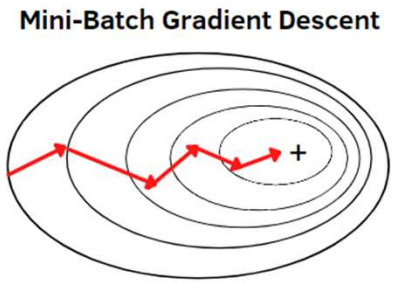
Deep Learning

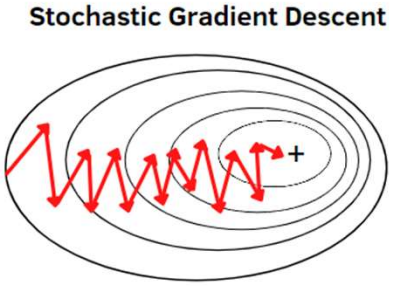


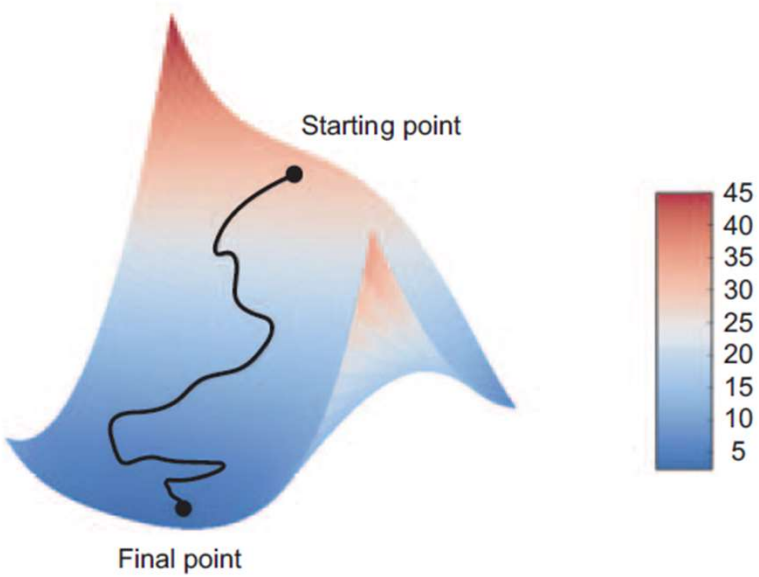
- **Stochastic gradient descent**

- Each update would then be more accurate, but far more expensive.
- The efficient compromise between these two extremes is to use mini-batches of reasonable size





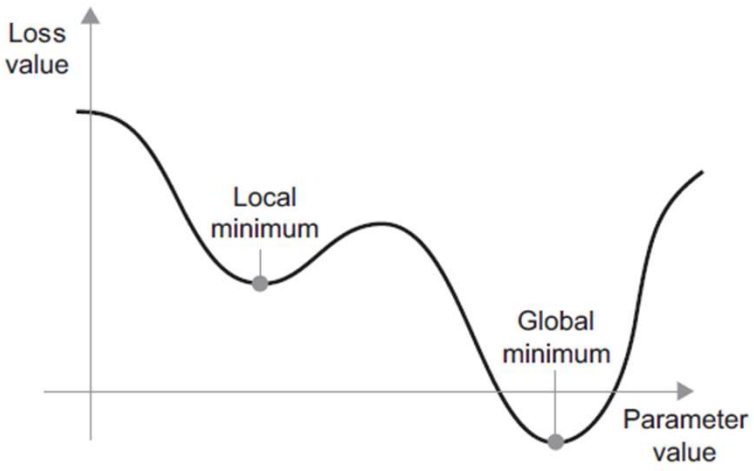




Gradient descent down a 2D loss surface
(two learnable parameters)

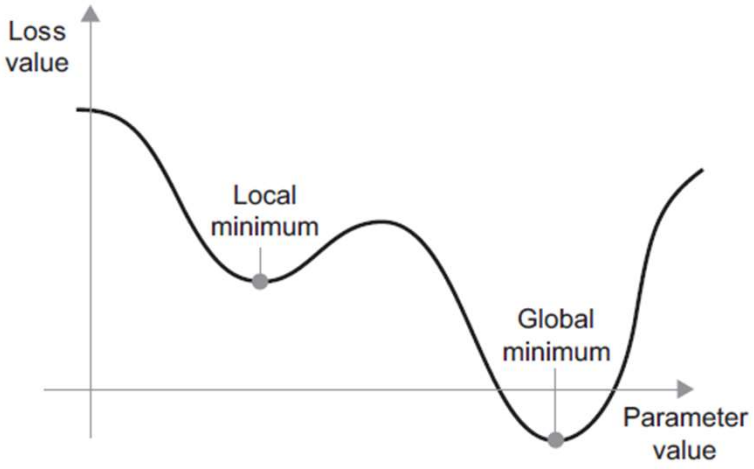
- **Stochastic gradient descent**
- There exist multiple variants of SGD that differ by taking into account previous weight updates when computing the next weight update, rather than just looking at the current value of the gradients.
- There is, for instance, SGD with momentum, as well as Adagrad, RMSProp, and several others.
- Such variants are known as *optimization methods* or *optimizers*.
- In particular, the concept of *momentum*, which is used in many of these variants, deserves your attention.
- Momentum addresses two issues with SGD: convergence speed and local minima

Deep Learning



A local minimum and a global minimum

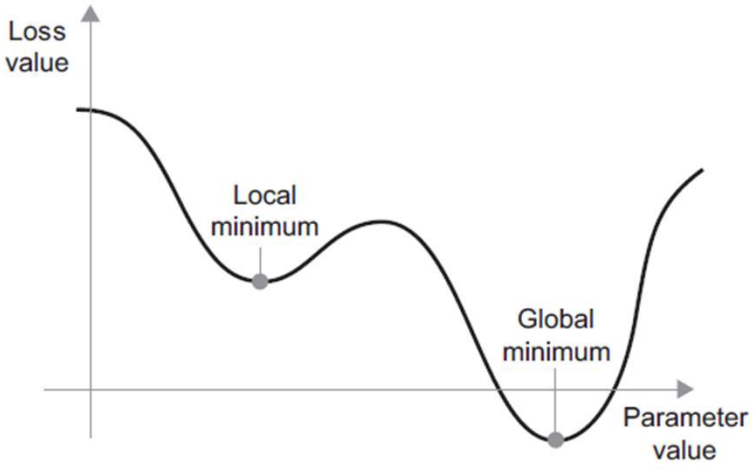
- A local minimum and a global minimum
- As you can see, around a certain parameter value, there is a *local minimum*: around that point, moving left would result in the loss increasing, but so would moving right.
- If the parameter under consideration were being optimized via SGD with a small learning rate, then the optimization process would get stuck at the local minimum instead of making its way to the global minimum.
- You can avoid such issues by using momentum, which draws inspiration from physics.
- A useful mental image here is to think of the optimization process as a small ball rolling down the loss curve.



A local minimum and a global minimum

- A local minimum and a global minimum
- If it has enough momentum, the ball won't get stuck in a ravine and will end up at the global minimum.
- Momentum is implemented by moving the ball at each step based not only on the current slope value (current acceleration) but also on the current velocity (resulting from past acceleration).
- This means updating the parameter w based not only on the current gradient value but also on the previous parameter update.

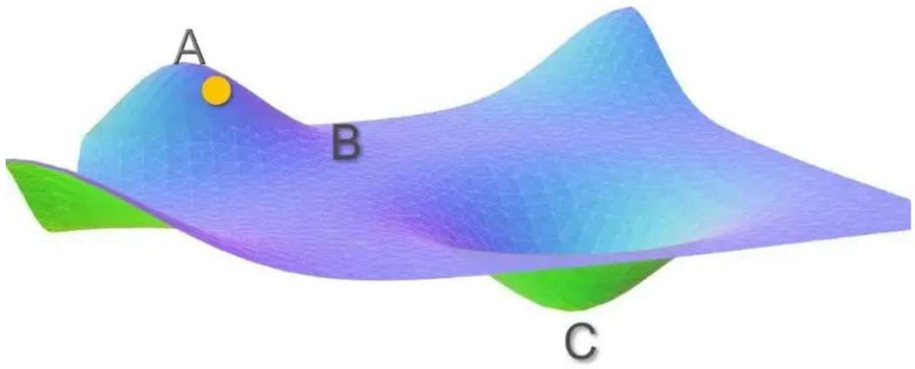
Deep Learning



A local minimum and a global minimum

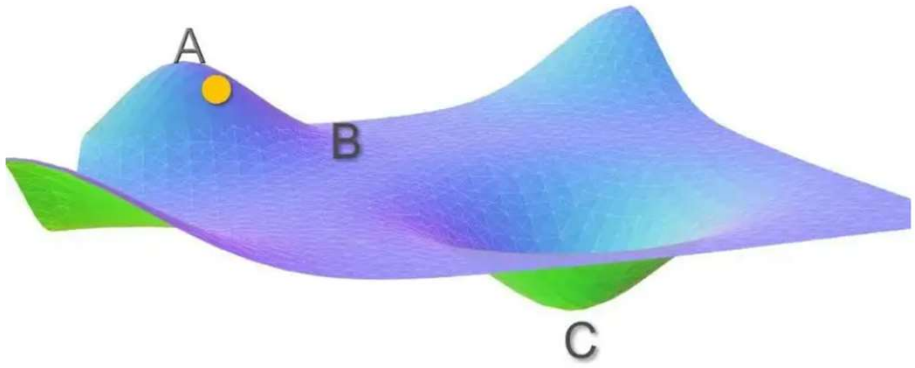
- A local minimum and a global minimum

```
past_velocity = 0.
momentum = 0.1
while loss > 0.01:
w, loss, gradient = get_current_parameters()
velocity = past_velocity * momentum +
learning_rate * gradient
w = w + momentum * velocity - learning_rate *
gradient
past_velocity = velocity
update_parameter(w)
```



Gradient Descent With Momentum

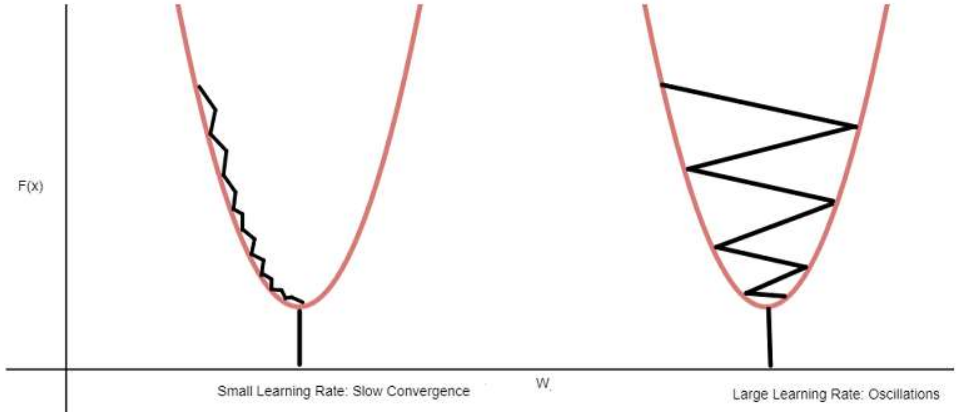
- Let's assume the initial weights of the network under consideration correspond to point A.
- With gradient descent, the Loss function decreases rapidly along the slope AB as the gradient along this slope is high.
- But as soon as it reaches point B the gradient becomes very low.
- The weight updates around B is very small.
- Even after many iterations, the cost moves very slowly before getting stuck at a point where the gradient eventually becomes zero.
- In this case, ideally, cost should have moved to the global minima point C, but because the gradient disappears at point B, we are stuck with a sub-optimal solution.

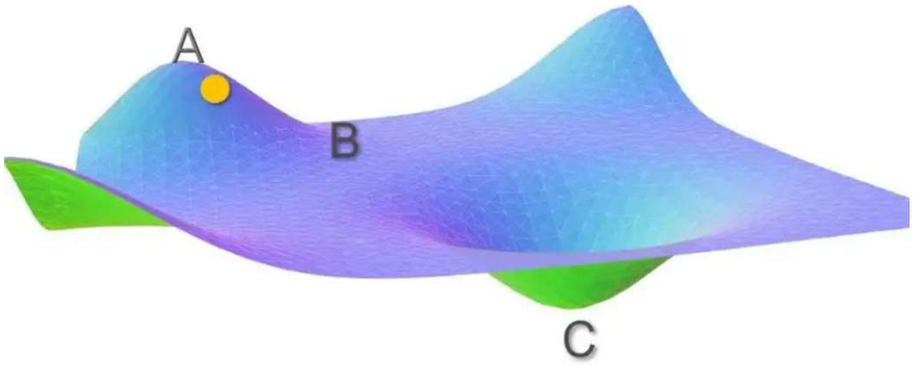


Gradient Descent With Momentum

- **How can momentum fix this?**
- Now, Imagine you have a ball rolling from point A.
- The ball starts rolling down slowly and gathers some **momentum** across the slope AB.
- When the ball reaches point B, it has accumulated enough momentum to push itself across the plateau region B and finally following slope BC to land at the global minima C.

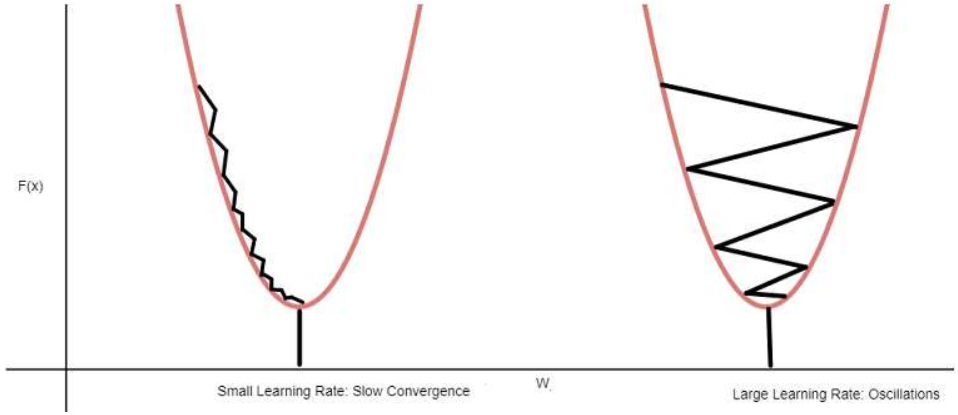
Deep Learning

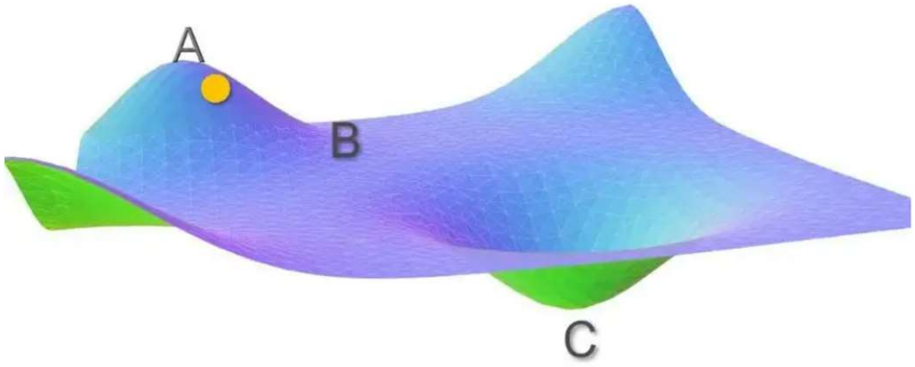




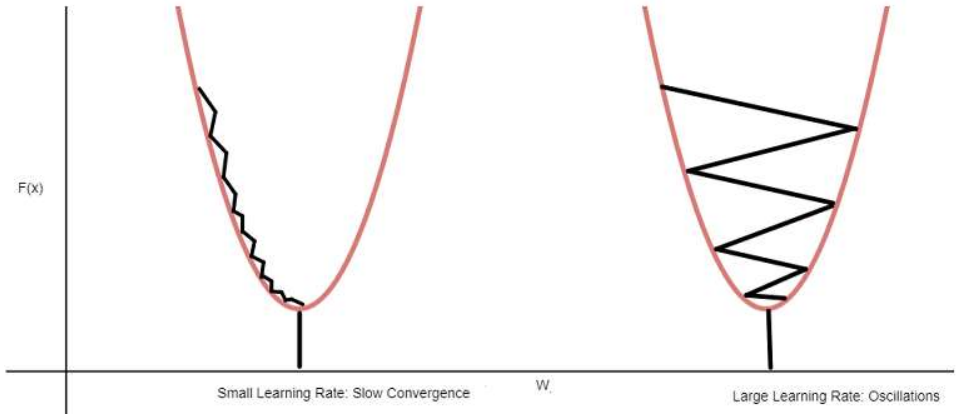
- **How can momentum fix this?**
- In the cost surface shown earlier let's zoom into point C.
- With gradient descent, if the learning rate is too small, the weights will be updated very slowly hence convergence takes a lot of time even when the gradient is high.
- This is shown in the left side image below.
- If the learning rate is too high cost oscillates around the minima as shown in the right side image below.

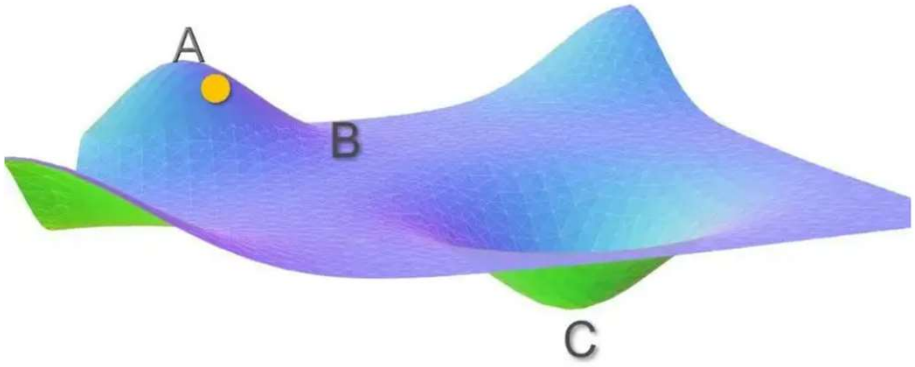
How does Momentum fix this?





- **How can momentum fix this?**
- Let's look at the last summation equation of the **momentum** again.
- **Case 1**: When all the past gradients have the same sign
- The summation term will become large and we will take large steps while updating the weights.
- Along the curve BC, even if the learning rate is low, all the gradients along the curve will have the same direction(sign) thus increasing the **momentum** and accelerating the descent.



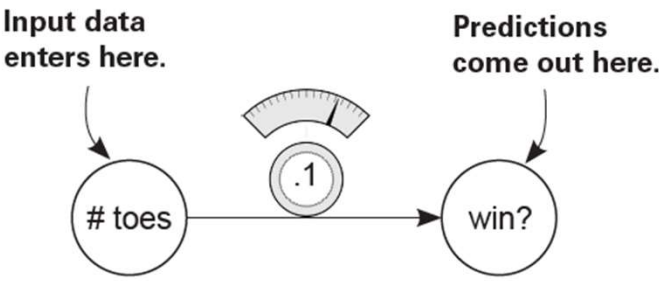


- **How can momentum fix this?**
- Let's look at the last summation equation of the **momentum** again.
- **Case 2**: when some of the gradients have +ve sign whereas others have –ve
- The summation term will become small and weight updates will be small.
- If the learning rate is high, the gradient at each iteration around the valley C will alter its sign between +ve and -ve and after few oscillations, the sum of past gradients will become small.
- Thus, making small updates in the weights from there on and damping the oscillations

Deep Learning

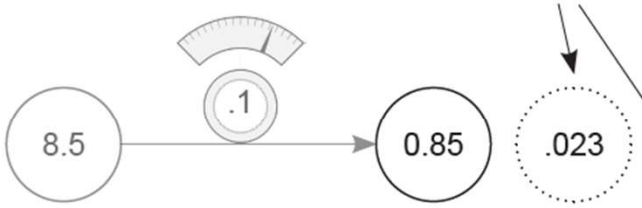
- **One iteration of gradient descent**
- **This performs a weight update on a single training example (input->true) pair.**

❶ An empty network




```
weight = 0.1

alpha = 0.01

def neural_network(input, weight):

    prediction = input * weight

    return prediction
```

❷ PREDICT: Making a prediction and evaluating error



The error is a way to measure how much you missed. There are multiple ways to calculate error, as you'll learn later. This one is *mean squared error*.

```
number_of_toes = [8.5]
win_or_lose_binary = [1] # (won!!!)

input = number_of_toes[0]
goal_pred = win_or_lose_binary[0]

pred = neural_network(input,weight)

error = (pred - goal_pred) ** 2
```

Error

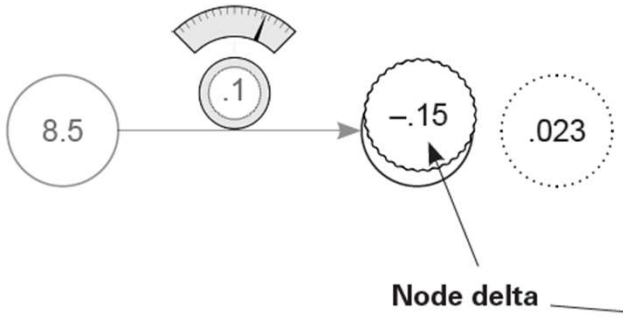
Raw error

Forces the raw error to be positive by multiplying it by itself. Negative error wouldn't make sense.

Deep Learning

- **One iteration of gradient descent**
- **This performs a weight update on a single training example (input->true) pair.**

❸ **COMPARE: Calculating the node delta and putting it on the output node**




```
number_of_toes = [8.5]
win_or_lose_binary = [1] # (won!!!)

input = number_of_toes[0]
goal_pred = win_or_lose_binary[0]

pred = neural_network(input,weight)

error = (pred - goal_pred) ** 2

delta = pred - goal_pred
```

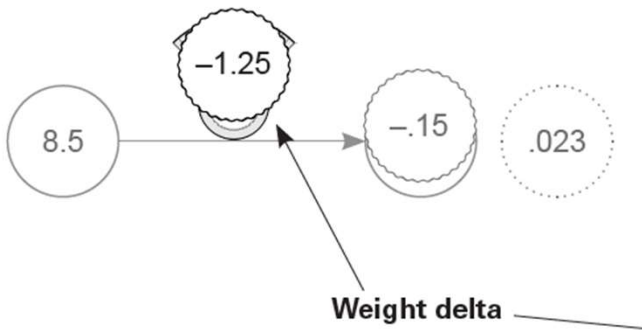
- *delta* is a measurement of how much this node missed.
- The true prediction is 1.0, and the network's prediction was 0.85, so the network was too *low* by 0.15.
- Thus, delta is *negative* 0.15.

Deep Learning

- **One iteration of gradient descent**

- **This performs a weight update on a single training example (input->true) pair.**

❹ **LEARN: Calculating the weight delta and putting it on the weight**

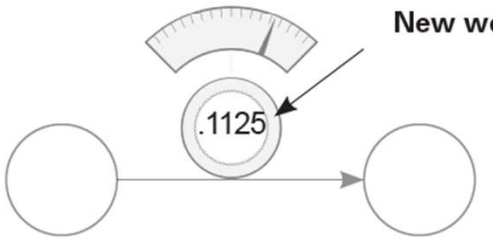



```
number_of_toes = [8.5]
win_or_lose_binary = [1] # (won!!!)

input = number_of_toes[0]
goal_pred = win_or_lose_binary[0]

pred = neural_network(input,weight)

error = (pred - goal_pred) ** 2

delta = pred - goal_pred

weight_delta = input * delta
```

- The primary difference between gradient descent and this implementation is the new variable delta.
- It's the raw amount that the node was too high or too low.
- Instead of computing direction_and_amount directly, you first calculate how much you want the output node to be different.
- Only then do you compute direction_and_amount to change weight (in step 4, now renamed weight_delta):

- **One iteration of gradient descent**
- **This performs a weight update on a single training example (input->true) pair.**

❺ LEARN: Updating the weight




```
number_of_toes = [8.5]
win_or_lose_binary = [1] # (won!!!)

input = number_of_toes[0]
goal_pred = win_or_lose_binary[0]

pred = neural_network(input,weight)

error = (pred - goal_pred) ** 2

delta = pred - goal_pred

weight_delta = input * delta

alpha = 0.01

weight -= weight_delta * alpha
```

Fixed before training →

- You multiply `weight_delta` by a small number `alpha` before using it to update `weight`.
- This lets you control how fast the network learns.
- If it learns too fast, it can update weights too aggressively and overshoot.