Deep Learning (B22EA0601)

Course Objectives

1. Explain the fundamentals of neural network based paradigm to problem solving.
2. Inculcate knowledge of concepts involved in training of convolutional neural networks.
3. Discuss the concepts and issues in recurrent neural networks.
4. Introduce prominent Generative Adversarial Networks and unsupervised learning paradigms.

Course Outcomes

- Explain the fundamental architecture of neural network and the concepts involved.
- Apply the shallow neural network models - Perceptron, Least-Squares Regression, Logistic Regression, Support Vector machines to solve real world binary and multiclass classification problems.
- Develop simple deep neural networks to solve problems in unsupervised learning.
- Create deep neural models like CNN and RNN to solve problems.
- Illustrate the applications of CNN, RNN and GAN for solving real world Problems
- Develop solutions using neural network based algorithms for the
- complex problems, either individually or as a part of the team and report the results with proper analysis and interpretation.

Course Content/Syllabus

Unit 1: Introduction to Deep Learning:

Neural networks, Training Neural Networks, Activation Functions, Multilayer Perceptrons, Implementation of Multilayer Perceptrons, Forward Propagation, Backward Propagation and Computational Graphs, Numerical Stability and Initialization, Generalization in Deep Learning, Dropout, Case study: Simple Neural Networks' Implementation using keras. (Text 1 Chapter 5.1 to 5.6)

Course Content/Syllabus

Unit 2: Convolutional Neural Network(CNN)

CNN- From Fully Connected layers to Convolutions, Convolution for images, Padding and Stride, Pooling, Convolution Architectures - Alexnet, NiN, VGGNet, ResNet, DenseNet, Transfer Learning, Case Study: Image Recognition Using CNN.
(Text 1 Chapter 7.1 to 7.3,7.5,8.1 to 8.4, 8.6 to 8.7)

Unit 3: Recurrent Neural Networks (RNN)

LSTM, GRU, Deep RNN, Sequence to Sequence Learning for Machine Translation, The Transformer Architecture, BERT Model, Data Set for Pretraining BERT, Pretraining BERT, Encoder-Decoder Architecture, Multilayer Perceptrons, Case study: RNN model implementation.
(Text 1 Chapter 9,10.1 to 10.3, 10.6,11.7, 15.8-15.10)

Unit 4: Generative Adversial Networks (GAN)

GAN , Deep Convolutional GAN,
Feature Learning Autoencoders, Regularization in autoencoders, Denoising autoencoders, Sparse autoencoders, Contractive autoencoders and Variational Auto Encoders
(Text 1 Chapter 20)

Text Book:

1. Aston Zhang, Zack C. Lipton, Mu Li, Alex J. Smola, “Dive into Deep Learning”, Amazon Science, 2020
2. Charu Aggarwal, “Neural Networks and Deep Learning”, Springer, 2018.
3. Ian Goodfellow, YoshuaBengio, Aaron Courville, “Deep Learning”, MIT Press

Reference Books:

1. Francis Chollet, “Deep Learning with Python”, Manning, 2018.
2. Jacek M. Zurada, Introduction to Artificial Neural Systems, PWS Publishing Company, 1995.
3. Simon Haykin, Neural Networks: A Comprehensive Foundation, Macmillan College Publishing Company, 1994.
4. Mohamad H. Hassoun, Fundamentals of Artificial Neural Networks, The MIT Press, 1995.
5. Laurene Fausett, Fundamentals of Neural Networks: Architectures, Algorithms, and Applications, Prentice Hall International, Inc., 1994.
6. B. D. Ripley, Pattern Recognition and Neural Networks, Cambridge University Press. 1996.

Motivation: Neural Networks are taking over!

- Neural networks have become one of the major thrust areas recently in various pattern recognition, prediction, and analysis problems
- In many problems they have established the state of the art
  - Often exceeding previous benchmarks by large margins

Breakthroughs with neural networks

wsworld.com/story/84013.html

azon Web Services | Primers | Math ∩ Prog | deeplearning.net/tuto | Deep Learning Tutori | deep learning | PHILIPS - Golden Ears | Language Technology | MyIDCare - Dashbo

TECHNEWSWORLD EMERGING TECH SEARCH

Computing Internet IT Mobile Tech Reviews Security Technology Tech Blog Reader Services

Microsoft AI Beats Humans at Speech Recognition

By Richard Adhikari
Oct 20, 2016 11:40 AM PT

Print Email

Image: Adobe Stock

Most Popular Newsletters News Alerts

How do you feel about Black Friday and Cyber Monday?

- They're great -- I get a lot of bargains!
- The deals are too spread out -- I'd prefer just one day.
- They're a fun way to kick off the holiday season.
- I don't like the commercialization of Thanksgiving Day.
- They're crucial for the retail industry and the economy.
- The deals typically aren't that good.

Vote to See Results

E-Commerce Times

Black Friday Shoppers Hungry for New Experiences, New Tech

Pay TV's Newest Innovation: Giving Users Control

Apple Celebrates Itself in $300 Coffee Table Tome

AWS Enjoys Top Perch in IaaS, PaaS Markets

US Comptroller Gears Up for Blockchain and

Microsoft's Artificial Intelligence and Research Unit earlier this week reported that its speech recognition technology had surpassed the performance of human transcriptionists.

Breakthrough with neural networks

Image segmentation and recognition

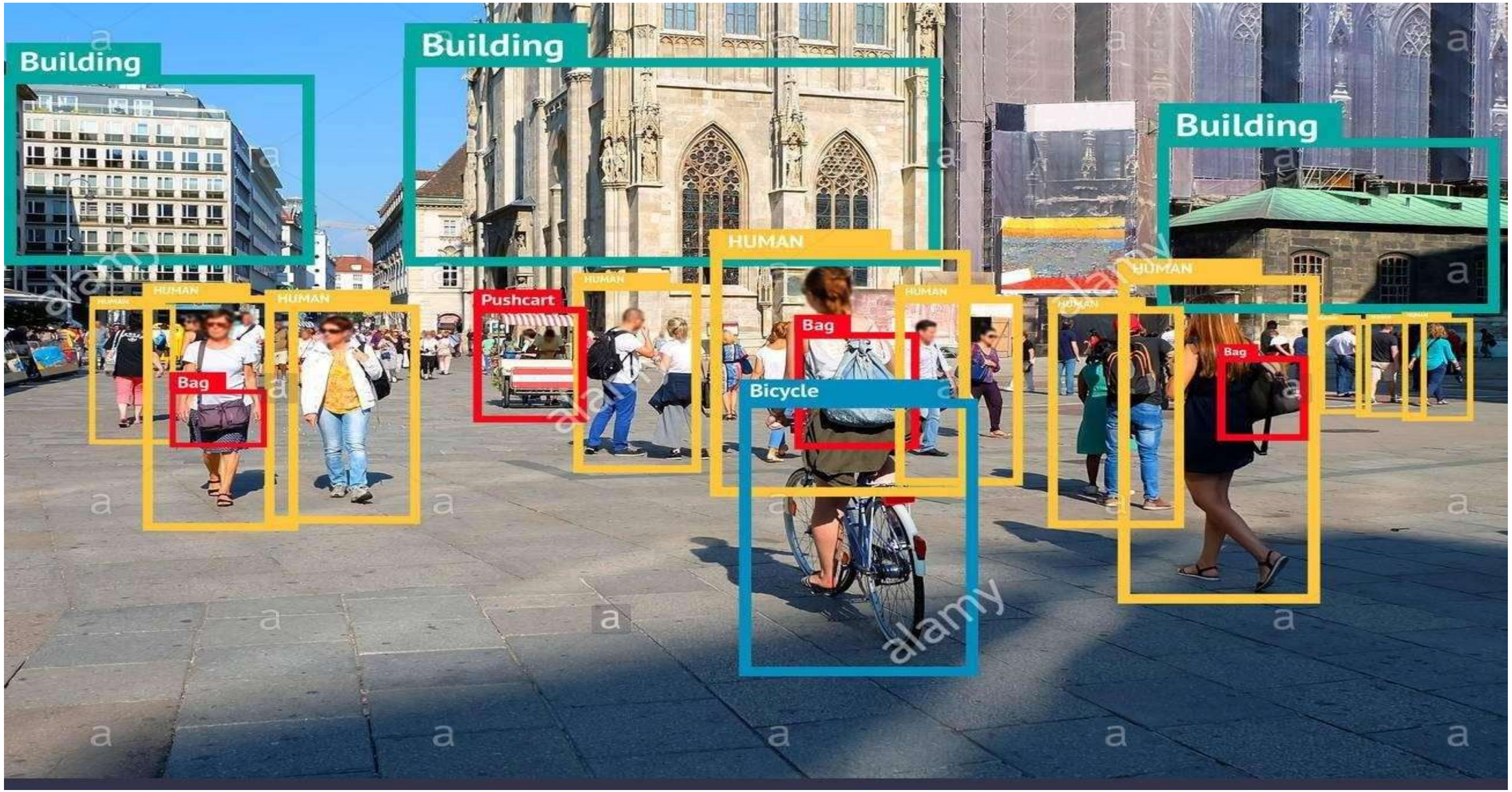
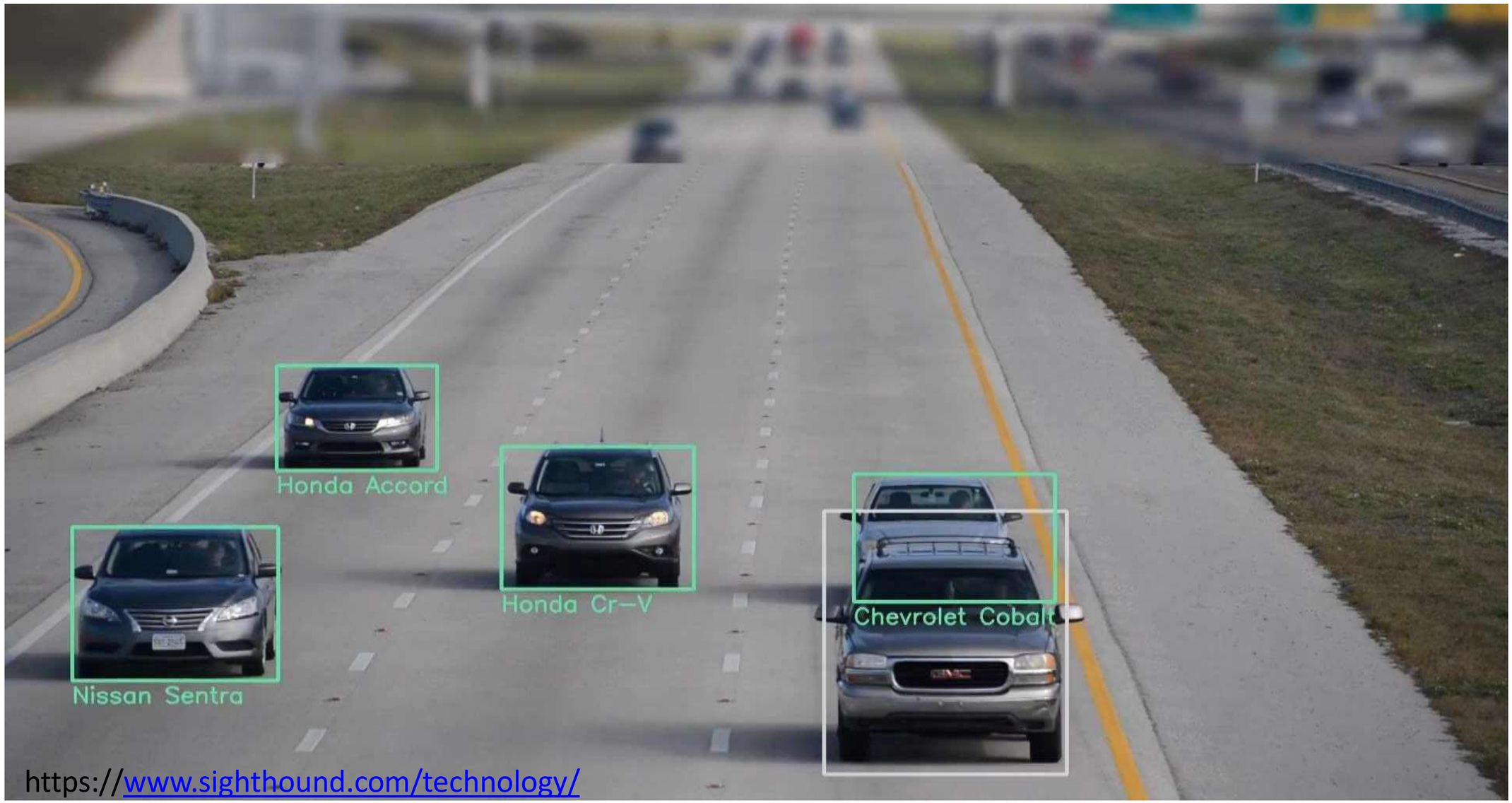


Image recognition



https://www.sighthound.com/technology/

Breakthroughs with neural networks

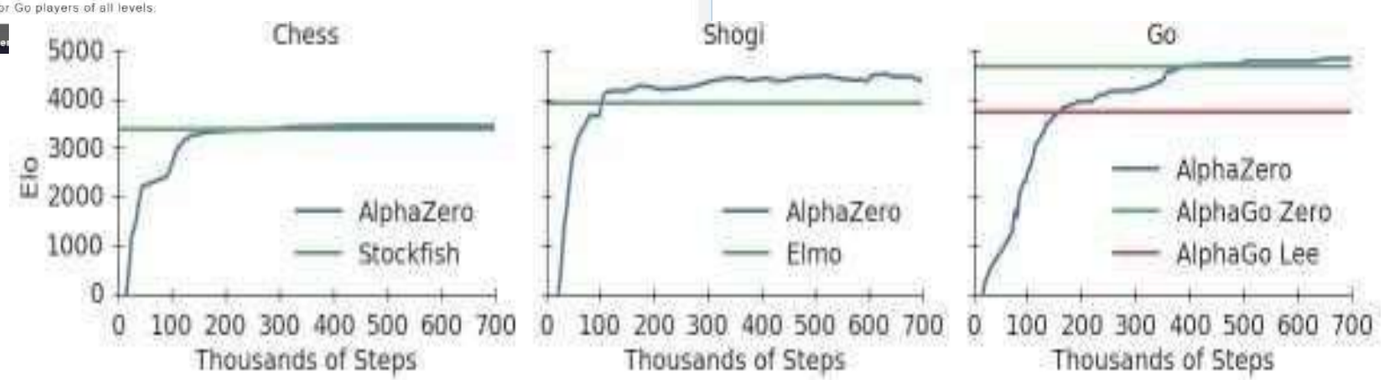
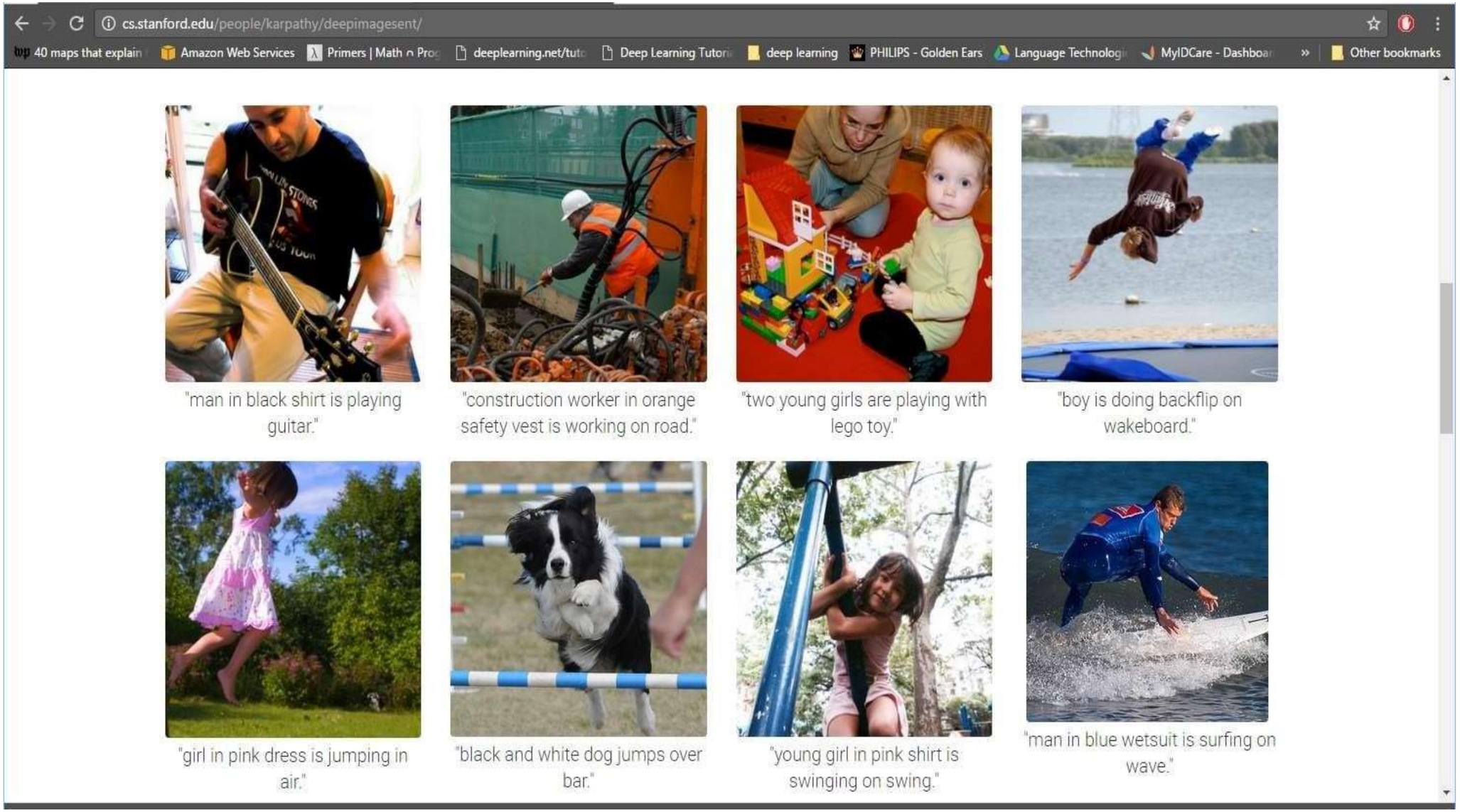


Figure 1: Training *AlphaZero* for 700,000 steps. Elo ratings were computed from evaluation games between different players when given one second per move. **a** Performance of *AlphaZero* in chess, compared to 2016 TCEC world-champion program *Stockfish*. **b** Performance of *AlphaZero* in shogi, compared to 2017 CSA world-champion program *Elmo*. **c** Performance of *AlphaZero* in Go, compared to *AlphaGo Lee* and *AlphaGo Zero* (20 block / 3 day) (29).

Success with neural networks



- Captions generated entirely by a neural network

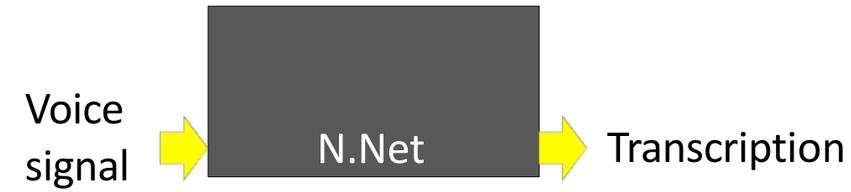
Generative AI Tools: ChatGPT, Gemini, Deep Seek, Whisper AI ……

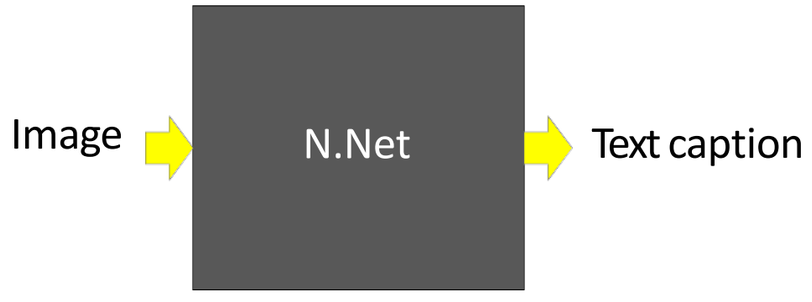
- And a variety of other problems:
  - From art to astronomy to healthcare..
  - and even predicting stock markets!

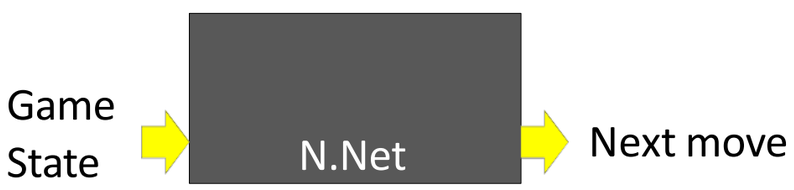
Successes with neural networks

Imagination is the limit…….

So what are neural networks??





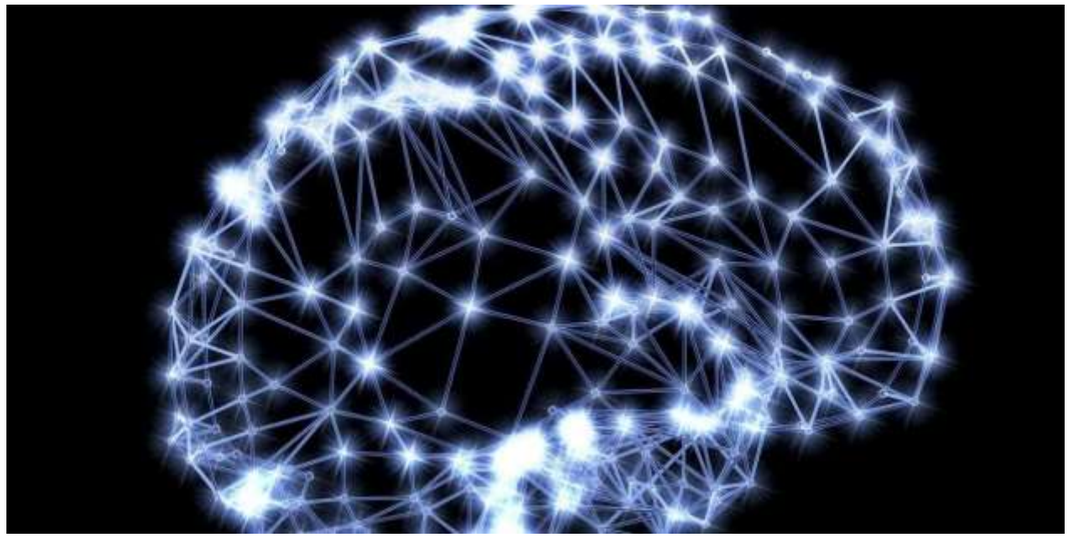


- What are these boxes?

So what are neural networks??

- It begins with this..

Observation: *The Brain*



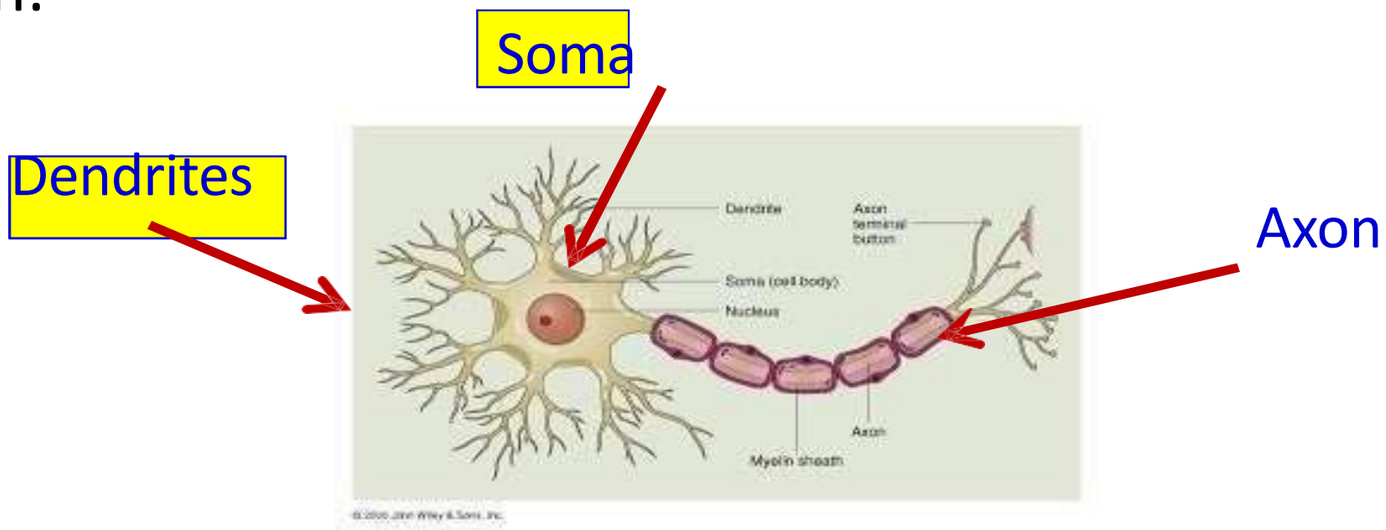
- Mid 1800s: The brain is a mass of interconnected neurons

Brain: Interconnected Neurons

- Many neurons connect *in* to each neuron
- Each neuron connects *out* to many neurons

Modelling the brain

- What are the units?
- A neuron:



- Signals come in through the dendrites into the Soma
- A signal goes out via the axon to other neurons
  - Only one axon per neuron
- Factoid that may only interest me: Neurons do not undergo cell division

A Perceptron

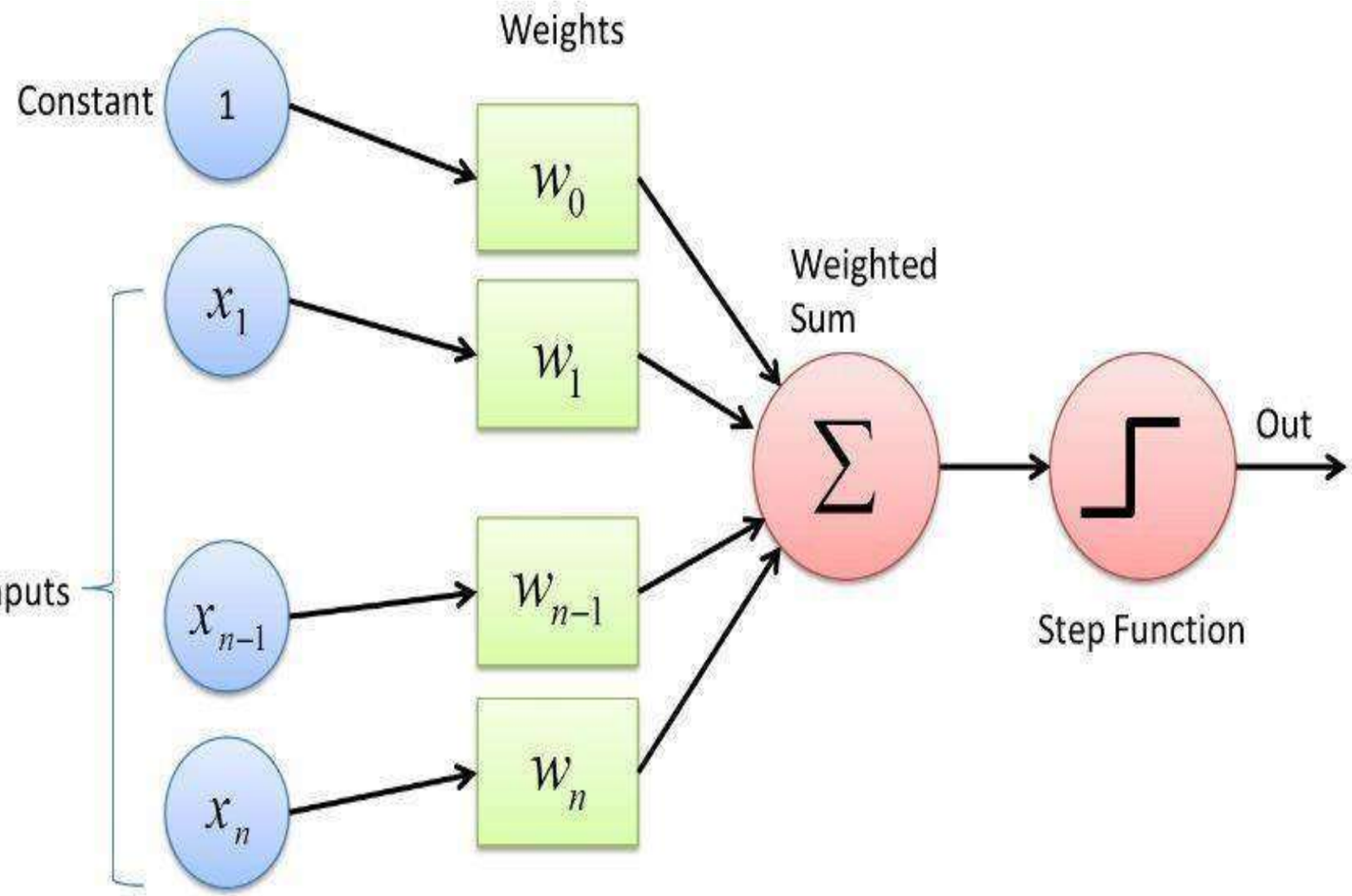
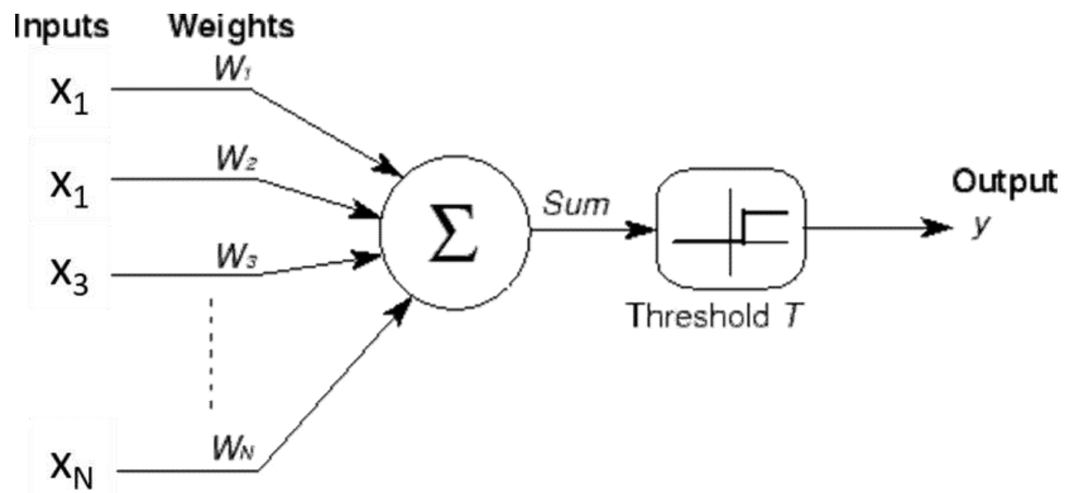


Fig : Perceptron

Simplified mathematical model of Perceptron

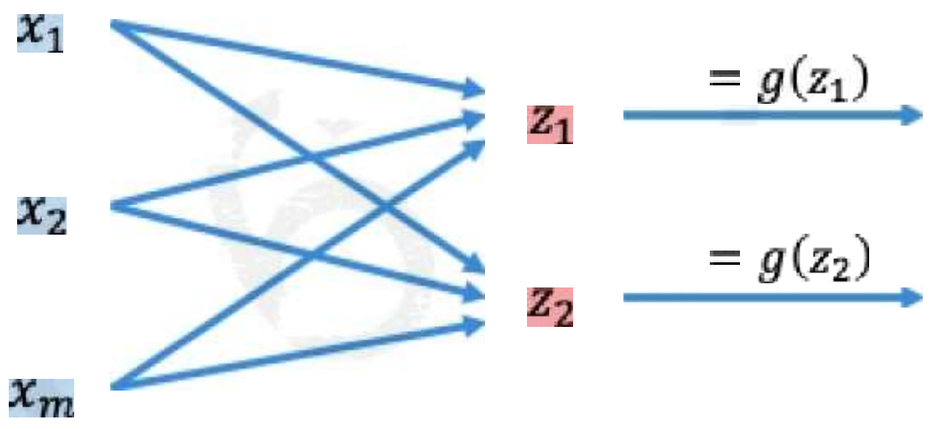


- Number of inputs combine linearly
  - Threshold logic: Fire if combined input exceeds or equal to threshold

$$Y = \begin{cases} 1 & if \sum_i w_i x_i - T >= 0 \\ 0 & else \end{cases}$$

Multi Output Perceptron

all inputs are densely connected to all outputs, these layers aæ called **D**ense layers



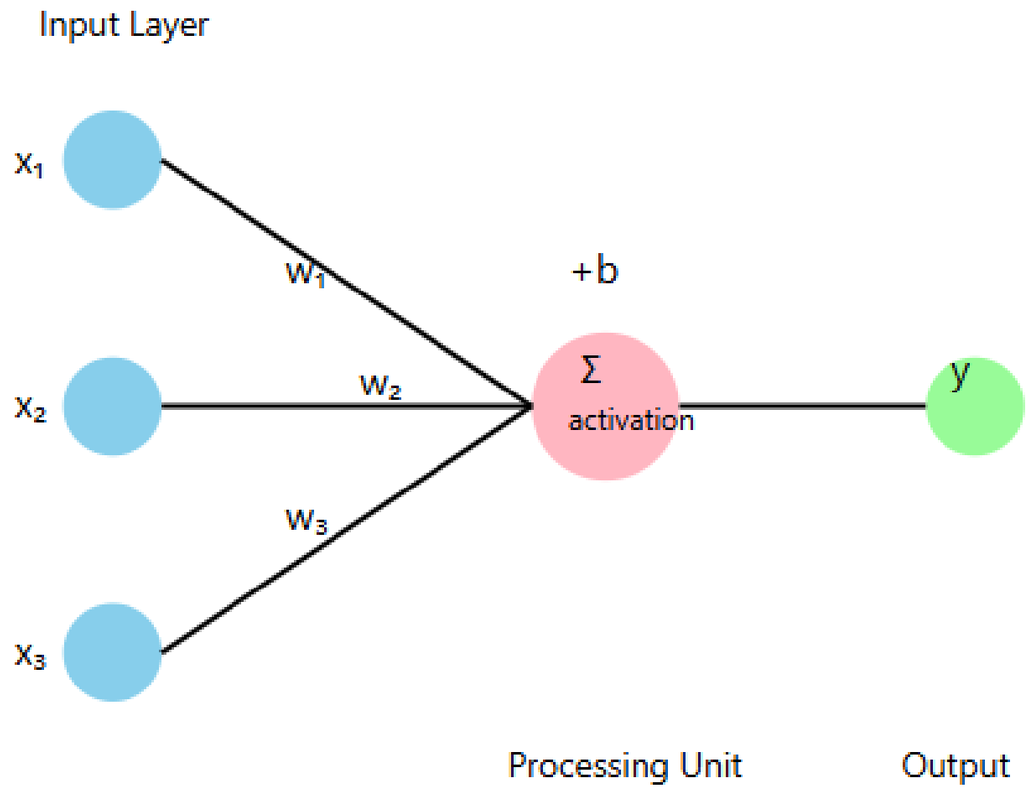
$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j \, w_{j,i}$$

Courtesy: @MiTDeep1ezarning

Applying Neural Networks

A perceptron is the simplest form of a feedforward neural network. It consists of:

- Input features (x_1, x_2, ..., x_n)
- Weights (w_1, w_2, ..., w_n)
- Bias (b)



Output

$y = \text{activation}(w_1x_1 + w_2x_2 + w3\ x3 + b)$

$= f(\sum wixi + b)$

Assume b=x0 and w0 =1

The above equation becomes

$f(\sum wixi)$

Where I ranges between 0 to 3

Perceptron Training Process Step by Step

1. **Initialization**:
    1. Set all weights to 0 or small random values
    2. Set bias to 0
    3. Define learning rate (η)
2. **For each training example**:
    1. Calculate predicted output
    2. Compare with actual output
    3. Calculate error
    4. Update weights and bias using weight update rule
    5. Repeat the process until loss is negligible

Weight Update Rule

w_new = w_old + η * (target - predicted) * input

bias_new = bias_old + η * (target - predicted)

Example Problem

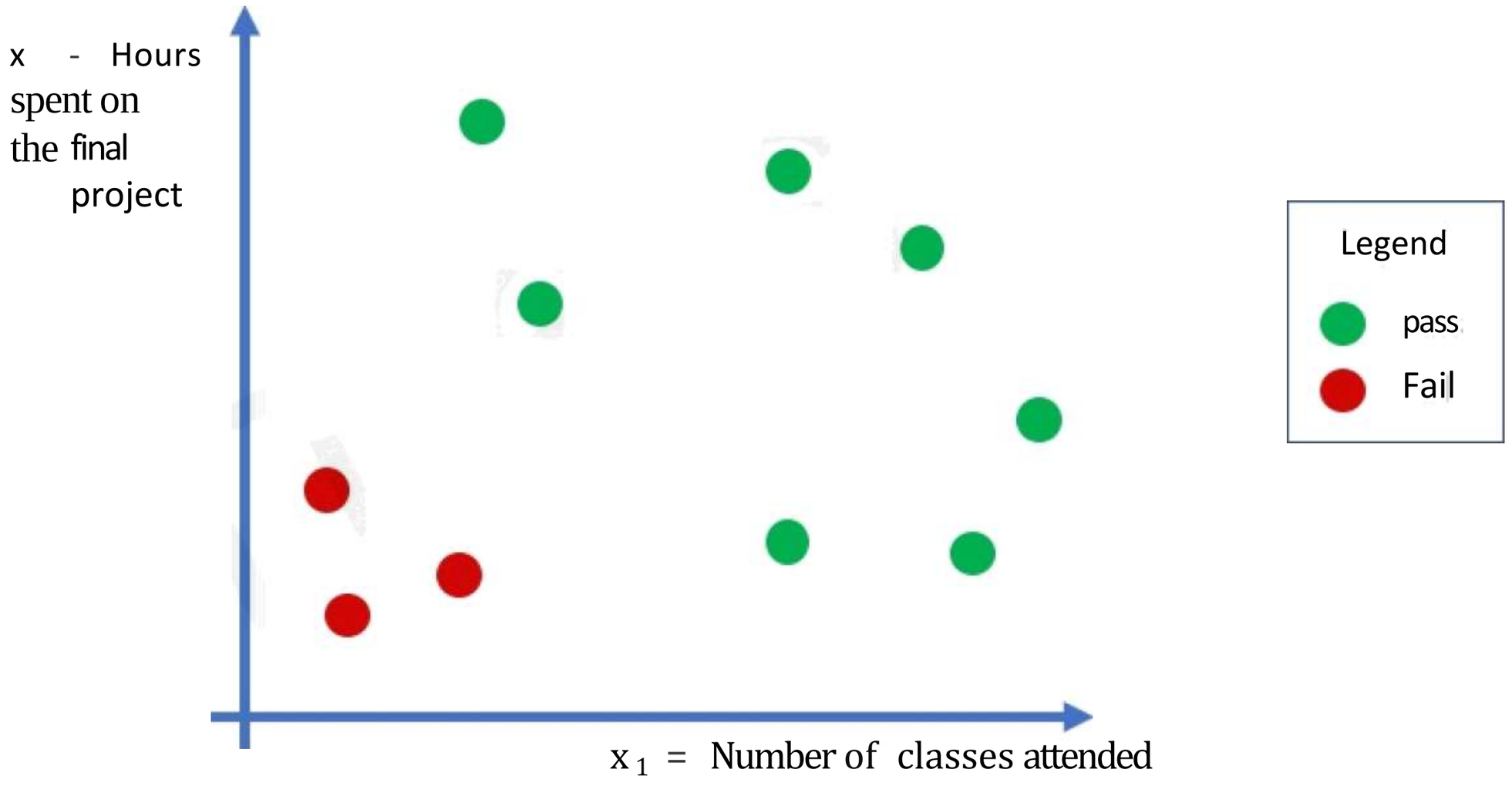
Will I pass this class?

Let's start with a simple two feature model

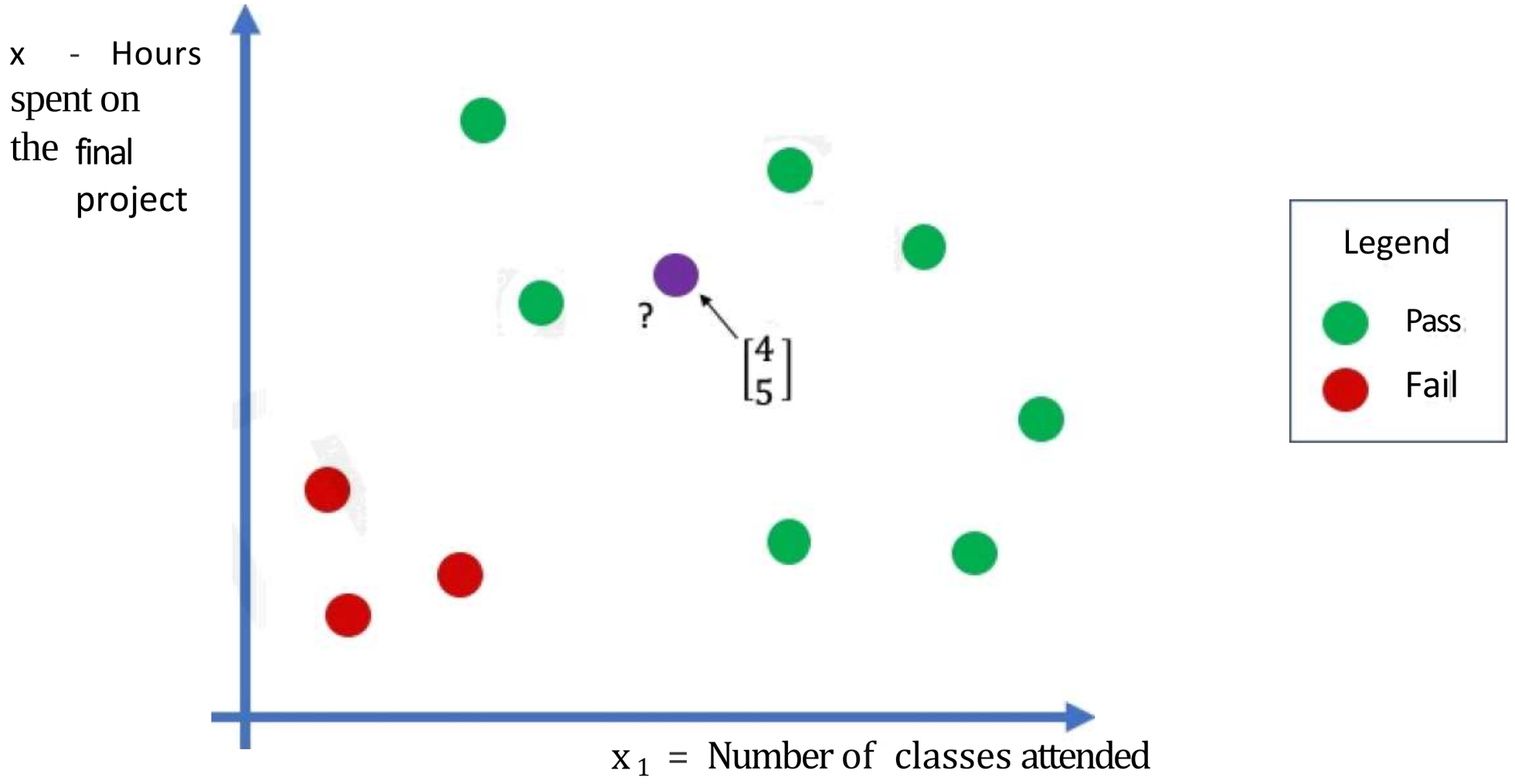
x_1 = Number of lectures you attend

x_2 = Hours spent on the final project

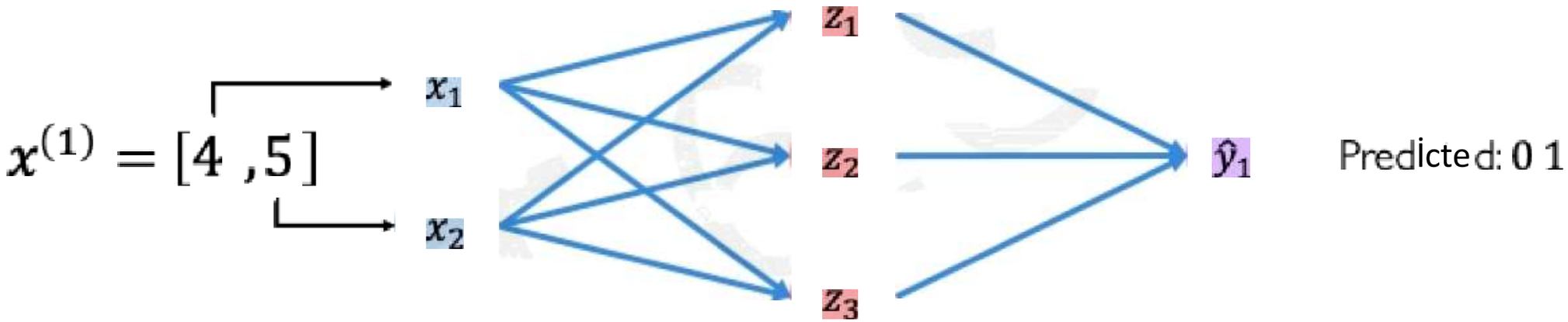
Example Problem Will I pass this class?



Example Problem Will I pass this class?

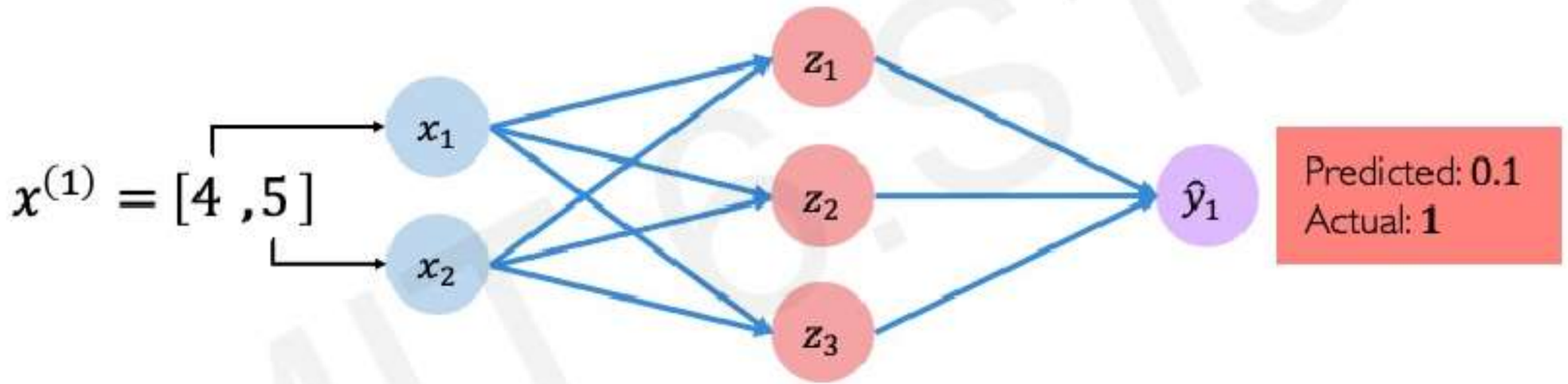


Example Problem Will I pass this class?



Example Problem: Will I pass this class?

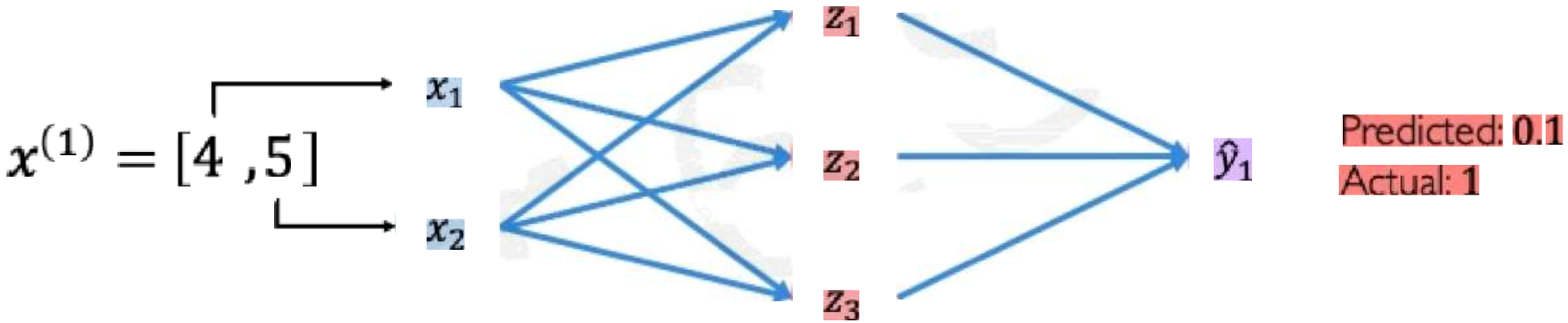
Forward Propagation



Quantifying Loss

Loss Functions

The loss of our network measures the cost incurred from Incorrect predictions



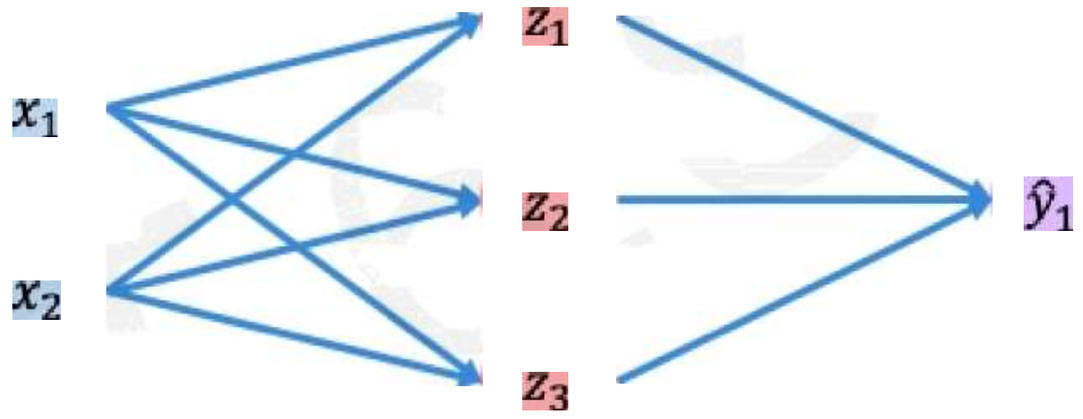
$$\mathcal{L}\left(f\left(x^{(i)}, \boldsymbol{W}\right), y^{(i)}\right)$$

Predicted Actual

Empirical Loss

*The **empirical** loss measures the total loss over our entire dataset*

$$\boldsymbol{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



$f(x)$ y

$$\begin{bmatrix} 0 & 1 \\ 0 & 8 \\ 0 & 6 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

Also known as Objective function, cost function,

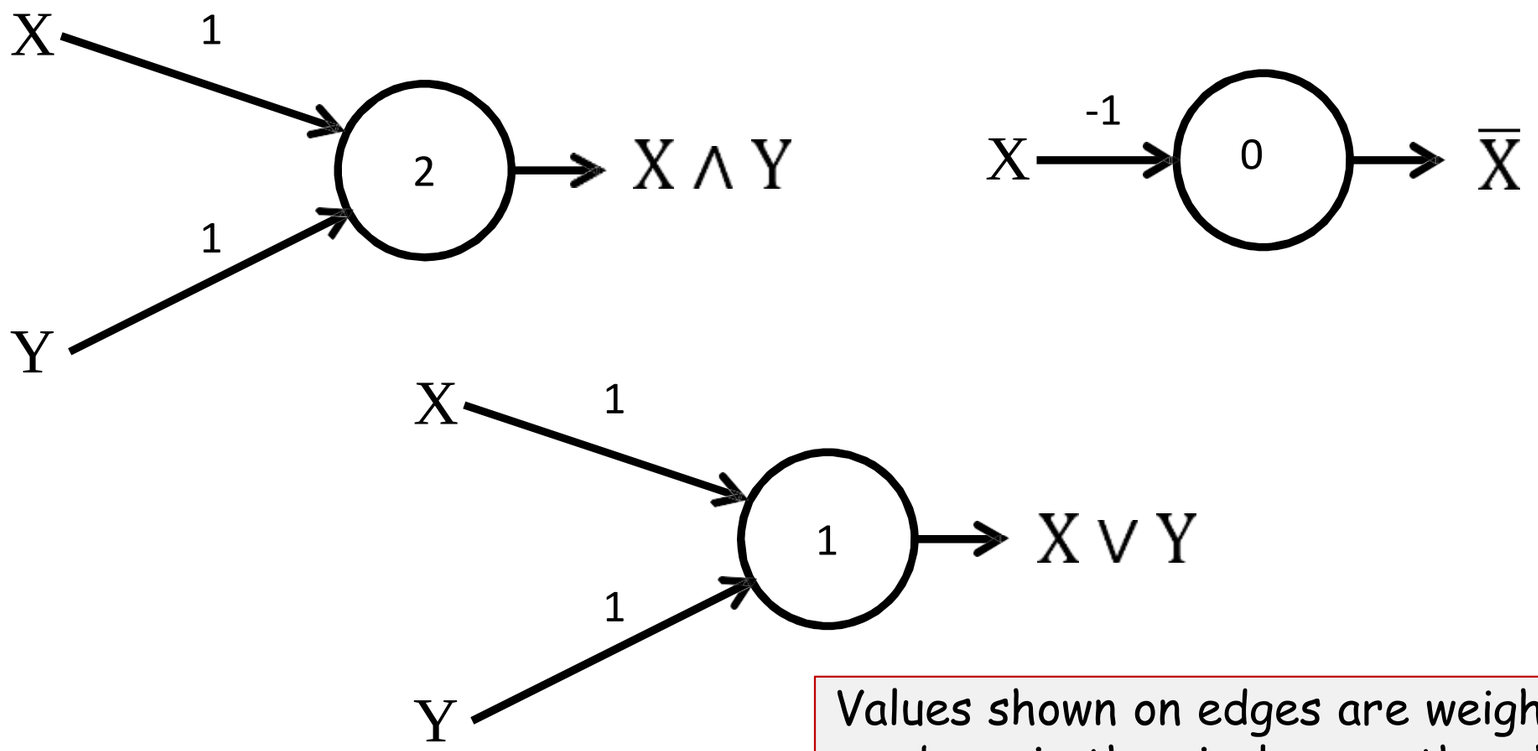
$$\boldsymbol{J(W)} = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}, \boldsymbol{W}\right), y^{(i)}\right)$$

Predicted Actual

Training Process

- **Forward Propagation**:
  - Inputs are passed through the network layer by layer to generate predictions. Each neuron's output is computed using its weights, biases, and activation function.
- **Loss Calculation**:
  - The loss function quantifies how well the model's predictions match the actual labels.
- **Backpropagation**:
  - A method for updating weights by calculating gradients of the loss function with respect to each weight. It uses the chain rule to propagate errors backward through the network.
- **Weight Updates**:
  - Weights are adjusted using optimization algorithms like Stochastic Gradient Descent (SGD) or Adam based on the computed gradients.

Perceptron

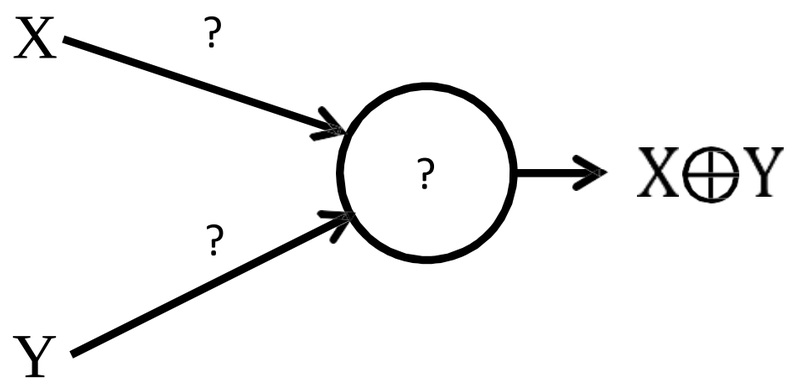


Values shown on edges are weights, numbers in the circles are thresholds

- Easily shown to mimic any Boolean gate
- But…

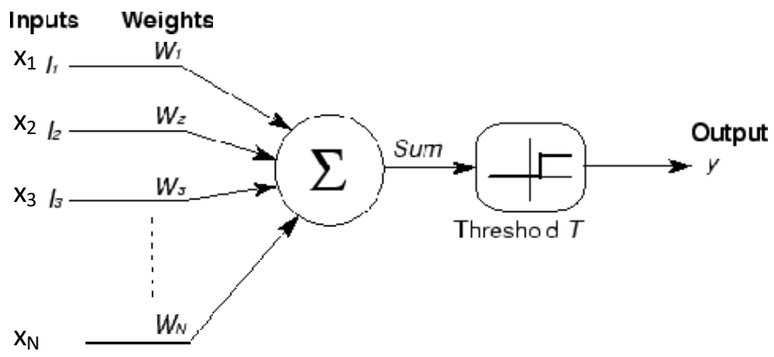
Perceptron

No solution for XOR!
Not universal!

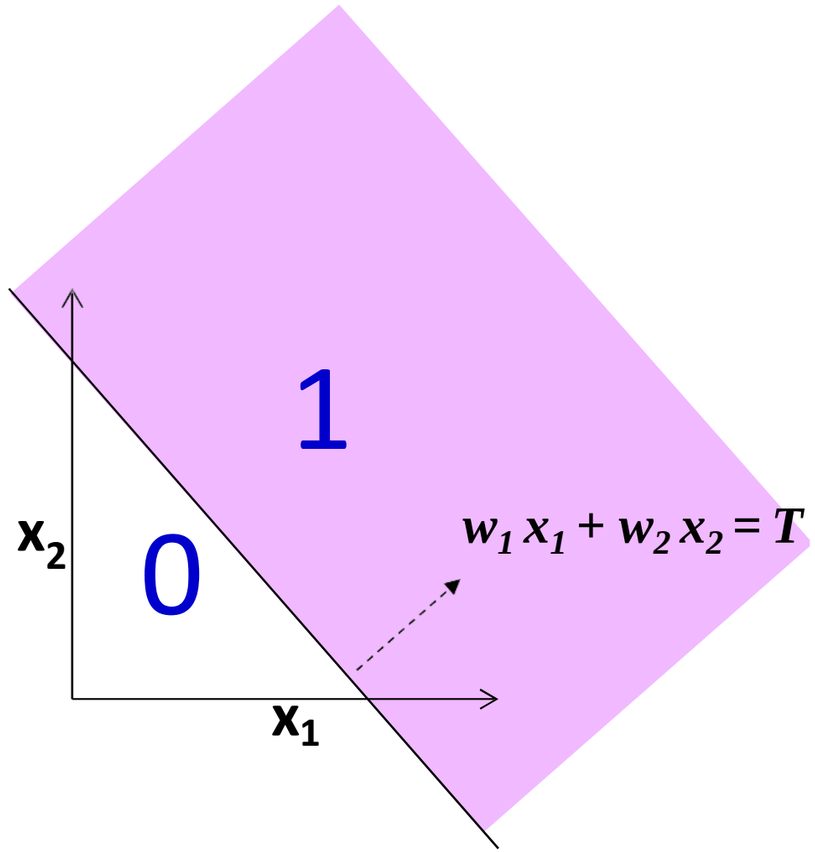


- Minsky and Papert, 1968

A Perceptron on Reals

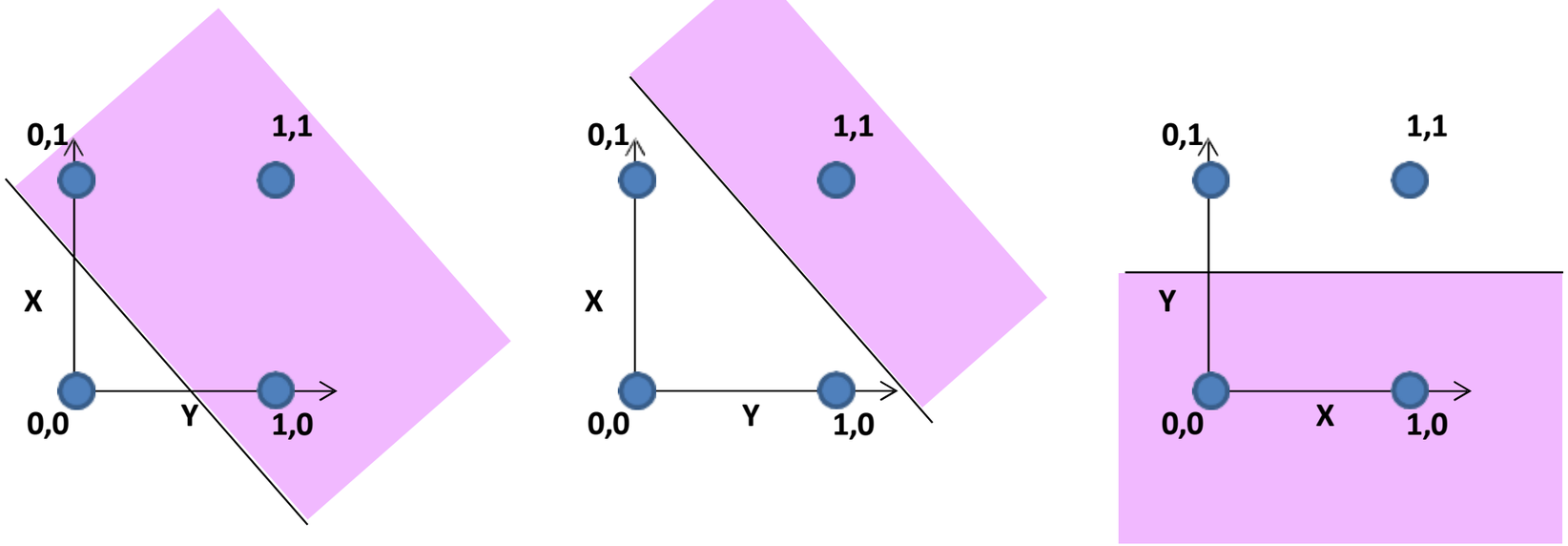


$$y = \begin{cases} 1 \; if \; \sum_{i} w_i x_i \geq T \\ 0 \; else \end{cases}$$



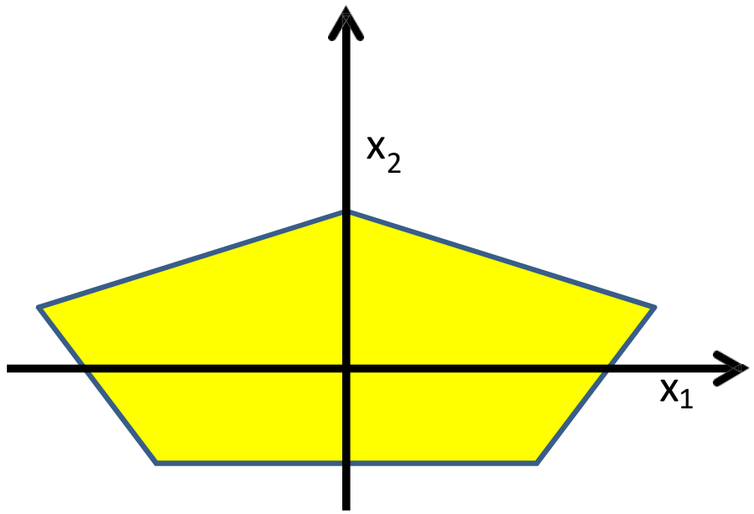
- A perceptron operates on *real*-valued vectors
  - **This is a *linear classifier***

Boolean functions with a real perceptron



- Boolean perceptrons are also linear classifiers
  - Purple regions have output 1 in the figures
  - What are these functions
  - Why can we not compose an XOR?

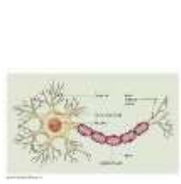
Composing complicated "decision" boundaries

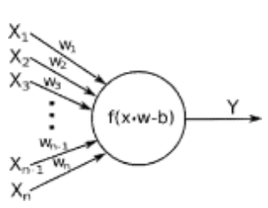


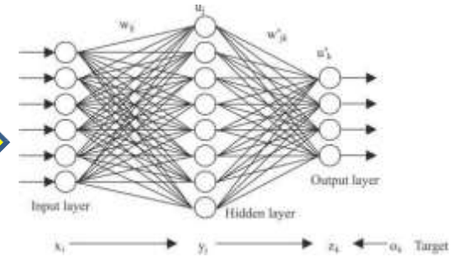
Can now be composed into "networks" to compute arbitrary classification "boundaries"

- Build a network of units with a single output that fires if the input is in the coloured area

A single neuron is not enough



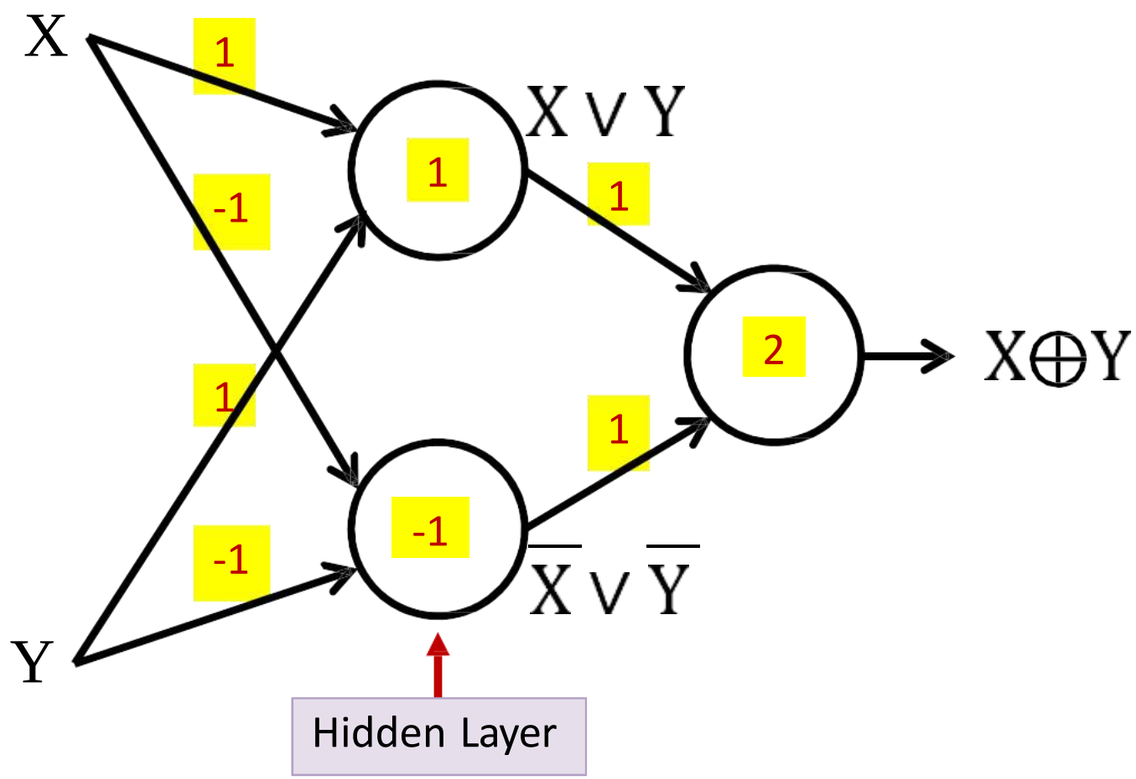




- Individual elements are weak computational elements
  - Marvin Minsky and Seymour Papert, 1969, *Perceptrons: An Introduction to Computational Geometry*

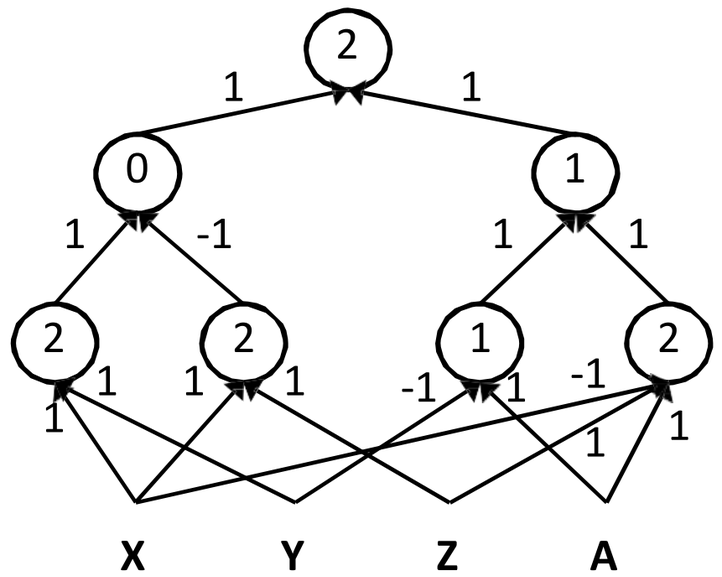
- *Networked* elements are required

Multi-layer Perceptron!



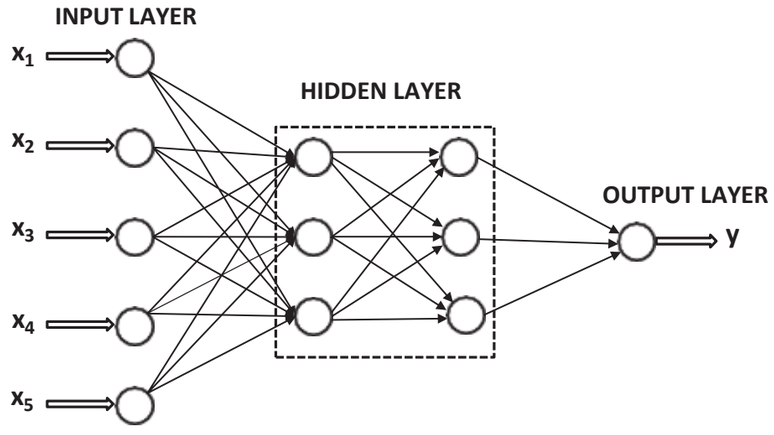
- XOR
  - The first layer is a “hidden” layer
  - Also originally suggested by Minsky and Papert 1968

A more generic model



- A “multi-layer” perceptron
- Can compose arbitrarily complicated Boolean functions!
  - In cognitive terms: Can compute arbitrary Boolean functions over sensory input

Multilayer Neural Networks



- The layers between the input and output are referred to as *hidden* because they perform intermediate computations.
- Each hidden node uses a combination of a linear transformation and an activation function $\Phi(\cdot)$ (like the output node of the perceptron).
- The use of *nonlinear* activation functions in the hidden layer is crucial in increasing learning capacity.

Activation Functions

1. An **activation function** is a **function** that is added to an **artificial neural network** in order to help the network learn **complex patterns in the data**.
2. An activation function is a function used in artificial neural networks which outputs a small value for small inputs, and a larger value if its inputs exceed a threshold.
3. If the inputs are large enough, the activation function "fires", otherwise it does nothing. In other words, an activation function is like a gate that checks that an incoming value is greater than a critical number.
4. Activation functions are useful because they add non-linearities into neural networks, allowing the neural networks to learn powerful operations.
5. If the activation functions were to be removed from a feedforward neural network, the entire network could be re-factored to a simple linear operation or matrix transformation on its input, and it would no longer be capable of performing complex tasks such as image recognition.

Neural Network and Deep Learning

Activation Functions

Typically the type of problems we consider under a machine learning environment falls either as classification or regression problem.

So we need to know

1. **What should be the activation function for a regression problem?**
2. **What should be the activation function for classification problem?**

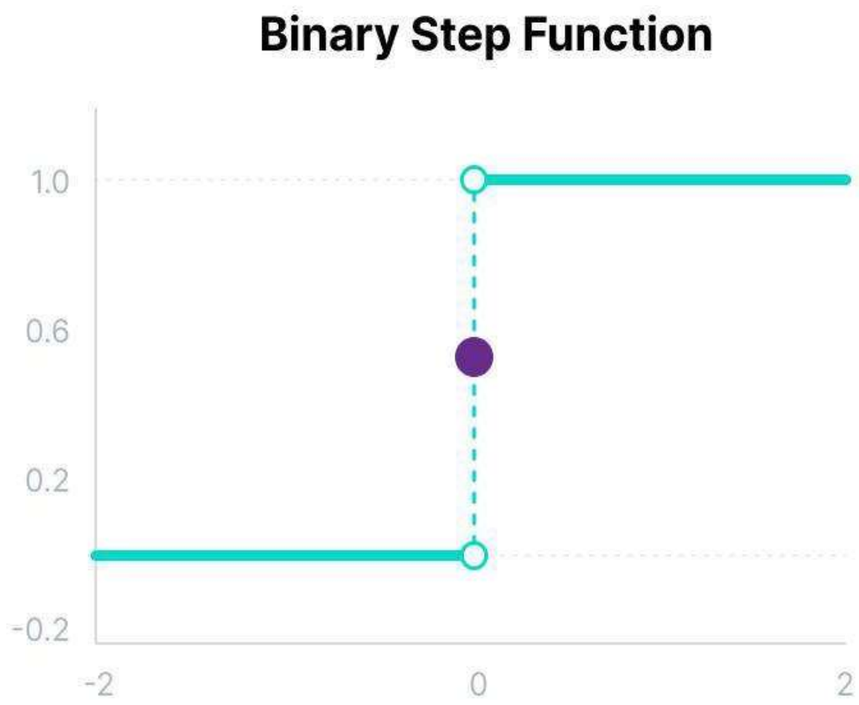
What is a Step Function?

Binary step function depends on a threshold value that decides whether a neuron should be activated or not.
The input fed to the activation function is compared to a certain threshold; if the input is greater than it, then the neuron is activated, else it is deactivated, meaning that its output is not passed on to the next hidden layer.

Input:
weighted sum of the inputs and biases of the neurons in a layer

Output:
A value of 0 or 1

Disadvantage:
- It cannot provide multi-value outputs—for example, it cannot be used for multi-class classification problems.
- The gradient of the step function is zero, which causes a hindrance in the backpropagation process.



Binary step

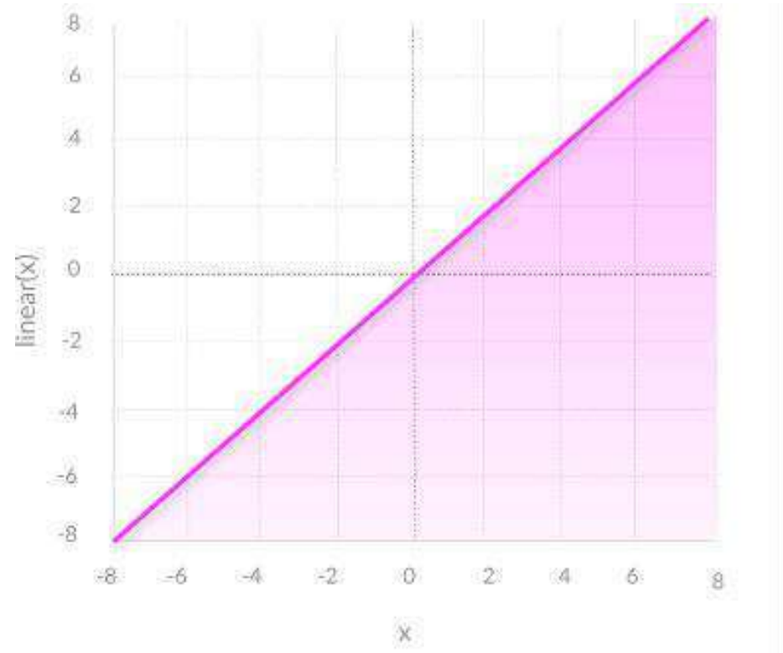
$$f(x) = \begin{cases} 0 & for\ x < 0 \\ 1 & for\ x \geqslant 0 \end{cases}$$

Neural Network and Deep Learning

What is a Linear Activation Function?

The linear activation function, also known as "no activation," or "identity function" (multiplied x1.0), is where the activation is proportional to the input.
The function doesn't do anything to the weighted sum of the input, it simply spits out the value it was given.

Input:
weighted sum of the inputs and biases of the neurons in a layer
Output:
it allows multiple outputs, not just yes and no.
Disadvantage:
•It’s not possible to use backpropagation as the derivative of the function is a constant and has no relation to the input x.
•All layers of the neural network will collapse into one if a linear activation function is used. No matter the number of layers in the neural network, the last layer will still be a linear function of the first layer. So, essentially, a linear activation function turns the neural network into just one layer.



Linear

$$f(x) = x$$

Non Linear Activation Function

The linear activation function shown above is simply a linear regression model. Because of its limited power, this does not allow the model to create complex mappings between the network's inputs and outputs.

Non-linear activation functions solve the following limitations of linear activation functions:

•They allow backpropagation because now the derivative function would be related to the input, and it's possible to go back and understand which weights in the input neurons can provide a better prediction.

•They allow the stacking of multiple layers of neurons as the output would now be a non-linear combination of input passed through multiple layers. Any output can be represented as a functional computation in a neural network.

1.Sigmoid Function or Logistic Function

Input: weighted sum of the inputs and biases of the neurons in a layer
Output: A value between 0 and 1
Variation: Continuously differentiable over different values of *z* and has a fixed output range.
Advantage: **Binary classification problems** capture the probability ranging from 0 to 1**.**
Disadvantages:

- value of *f(x)* increases at a very slow rate
- susceptible to the vanishing gradient problem
- becomes hard to optimize

  Vanishing gradient—for very high or very low values of z, there is almost no change to the prediction, causing a vanishing gradient problem. This can result in the network refusing to learn further, or being too slow to reach an accurate prediction.

Outputs not zero centered.
Computationally expensive

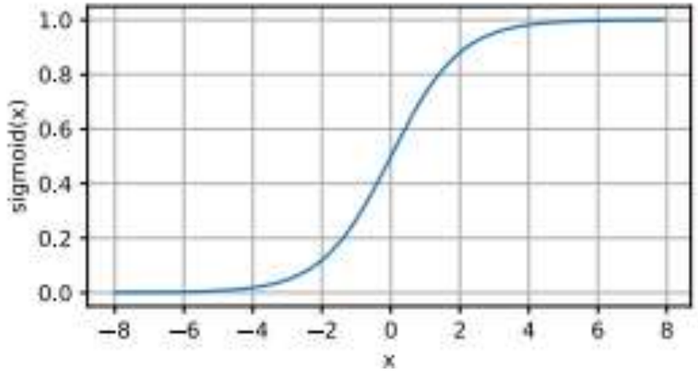
Sigmoid / Logistic

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

Neural Network and Deep Learning

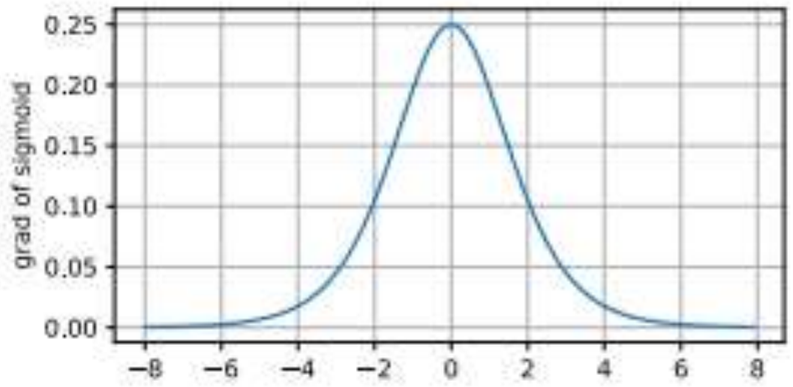
1.Sigmoid Function or Logistic Function

Here's why sigmoid/logistic activation function is one of the most widely used functions:
It is commonly used for models where we have to predict the probability as an output. Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice because of its range.
•The function is differentiable and provides a smooth gradient, i.e., preventing jumps in output values. This is represented by an S- shape of the sigmoid activation function.



the derivative of the sigmoid function is

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

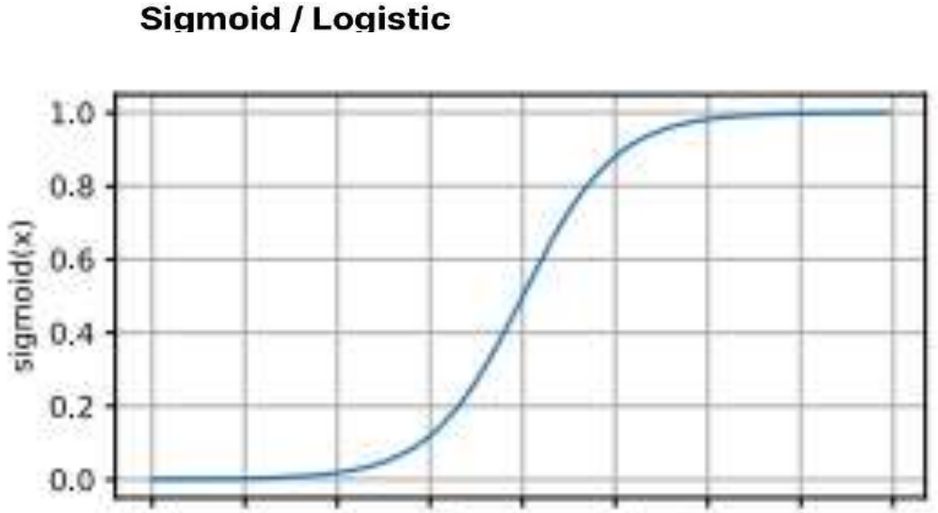


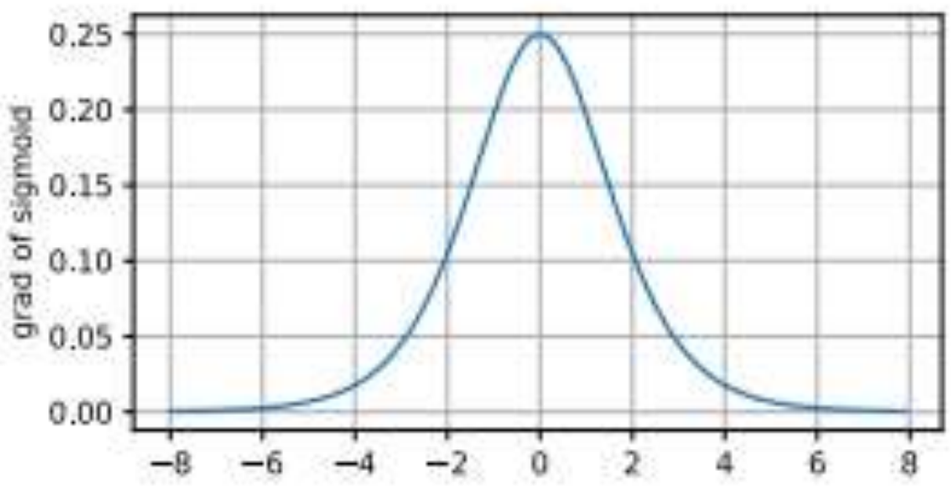
1.Sigmoid Function or Logistic Function

The gradient values are only significant for range -3 to 3, and the graph gets much flatter in other regions.

It implies that for values greater than 3 or less than -3, the function will have very small gradients. As the gradient value approaches zero the network ceases to learn and suffers from the *Vanishing gradient* problem.

The output of the logistic function is not symmetric around zero. So the output of all the neurons will be of the same sign. This makes the training of the neural network more difficult and unstable.





Python code for Sigmoid activation

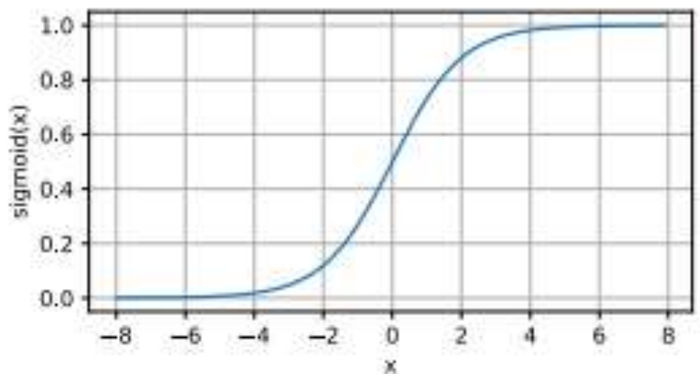
```
import numpy as np
def sigmoid(x):
    """
    Compute the Sigmoid activation function.
    """
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    """
    Compute the derivative of the Sigmoid function.
    """
    sig = sigmoid(x)
    return sig * (1 - sig)

# Example usage
x = np.array([-2, -1, 0, 1, 2])
print("Sigmoid Output:", sigmoid(x))
print("Sigmoid Derivative:", sigmoid_derivative(x))
```

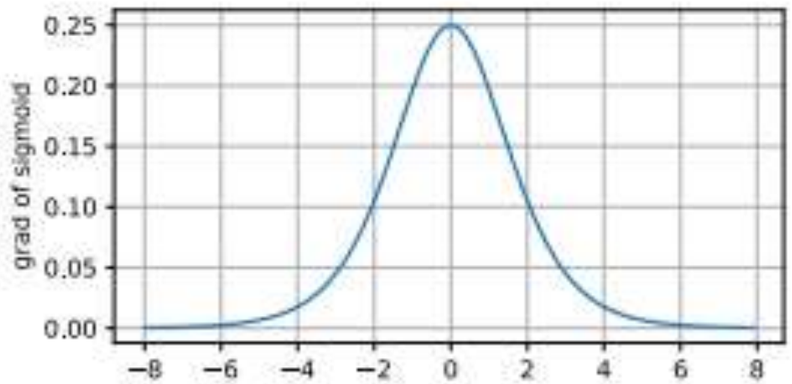
Vanishing Gradient

For the nodes with sigmoid activation functions, we know that the partial derivative of the sigmoid function reaches a maximum value of 0.25.



When there are more layers in the network, the value of the product of derivative decreases until at some point the partial derivative of the loss function approaches a value close to zero, and the partial derivative vanishes. We call this the vanishing gradient problem.

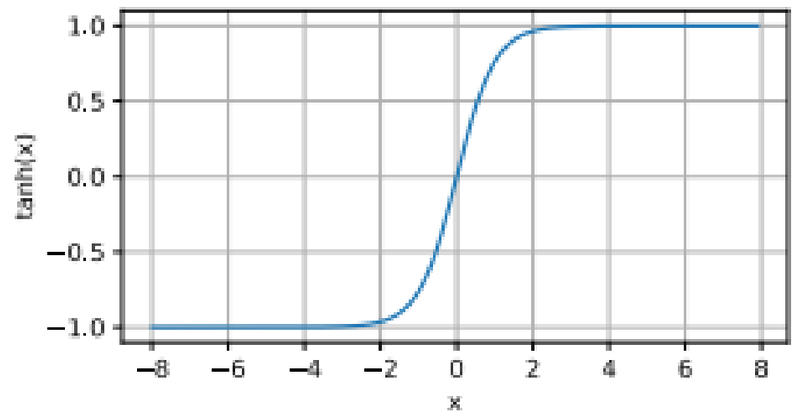
With shallow networks, sigmoid function can be used as the small value of gradient does not become an issue.
When it comes to deep networks, the vanishing gradient could have a significant impact on performance. The weights of the network remain unchanged as the derivative vanishes.
During back propagation, a neural network learns by updating its weights and biases to reduce the loss function. In a network with vanishing gradient, the weights cannot be updated, so the network cannot learn.
The performance of the network will decrease as a result.



Tanh Function

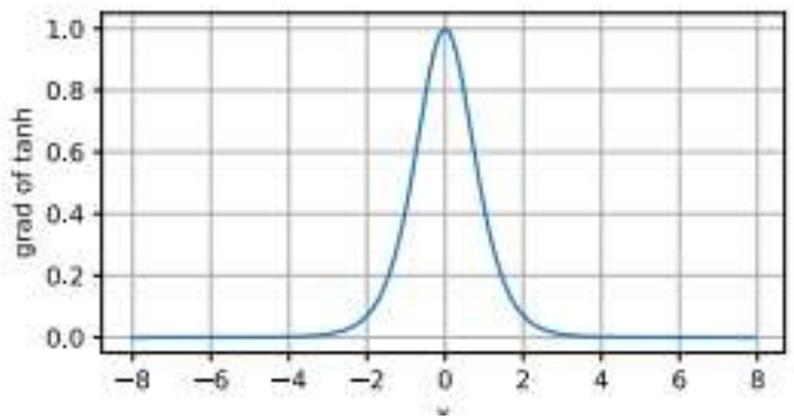
Like the sigmoid function, the tanh (hyperbolic tangent) function also squashes its inputs, transforming them into elements on the interval between −1 and 1

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}.$$



The derivative of the tanh function is:

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x).$$



What is a Tanh Function?

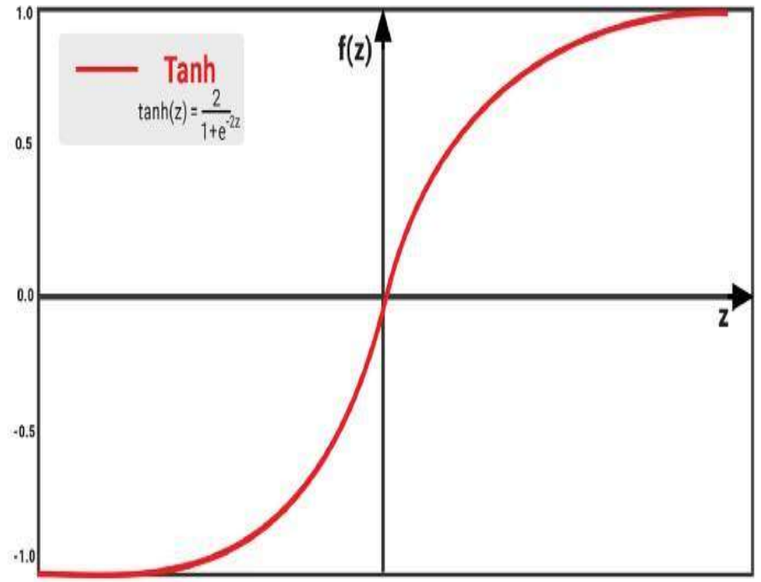
Input:

weighted sum of the inputs and biases of the neurons in a layer

Output: Bounded between -1 to 1.

Variation:

Gradient or the derivative of the Tanh function is steeper as compared to the sigmoid function . It is the shifted version of sigmoid with mean value of zero



Advantage:

Zero centered—making it easier to model inputs that have strongly negative, neutral, and strongly positive values.

Otherwise like the Sigmoid function.

Disadvantages:

- susceptible to the vanishing gradient problem
- Model slows down exponentially beyond +2 and -2.

TanH Function

```
import numpy as np

def tanh(x):
    #Compute the Tanh activation function.

    return np.tanh(x)

def tanh_derivative(x):
    #Compute the derivative of the Tanh function.

    return 1 - np.tanh(x) ** 2

# Example usage
x = np.array([-2, -1, 0, 1, 2])
print("Tanh Output:", tanh(x))
print("Tanh Derivative:", tanh_derivative(x))
```

ReLU (Rectified Linear Unit)

Formula: f(x)=max(0,x)

Retains positive values, discards negatives

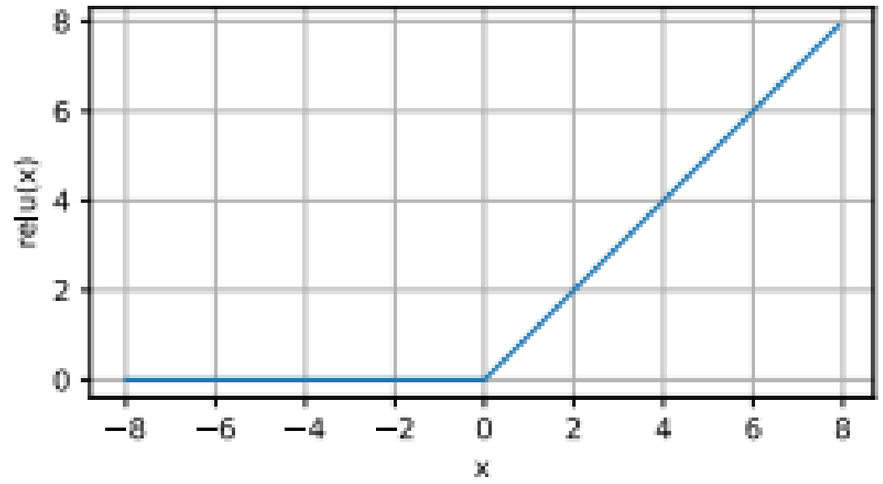
Advantages:

- Avoids vanishing gradient issues.
- Efficient and simple to compute.

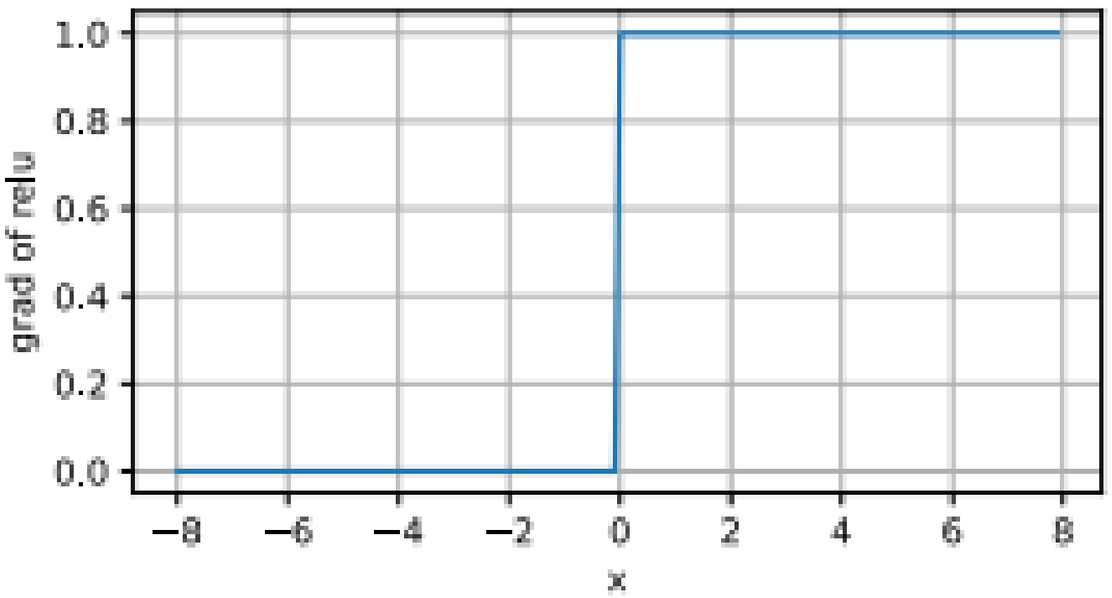
Drawbacks:

- "Dead neurons“ or Dying ReLU (weights don’t updat

Variants: Leaky ReLU, Parametric ReLU (pReLU).



ReLU



When the input is negative, the derivative of the ReLU function is 0, and when the input is positive, the derivative of the ReLU function is 1

Python code for ReLU

```
def relu(x):
    """
    Compute the ReLU activation function.
    """
    return np.maximum(0, x)

def relu_derivative(x):
    """
    Compute the derivative of the ReLU function.
    """
    return np.where(x > 0, 1, 0)
# Example usage
x = np.array([-2, -1, 0, 1, 2])
print("ReLU Output:", relu(x))
print("ReLU Derivative:", relu_derivative(x))
```

What is a Leaky ReLU Function?

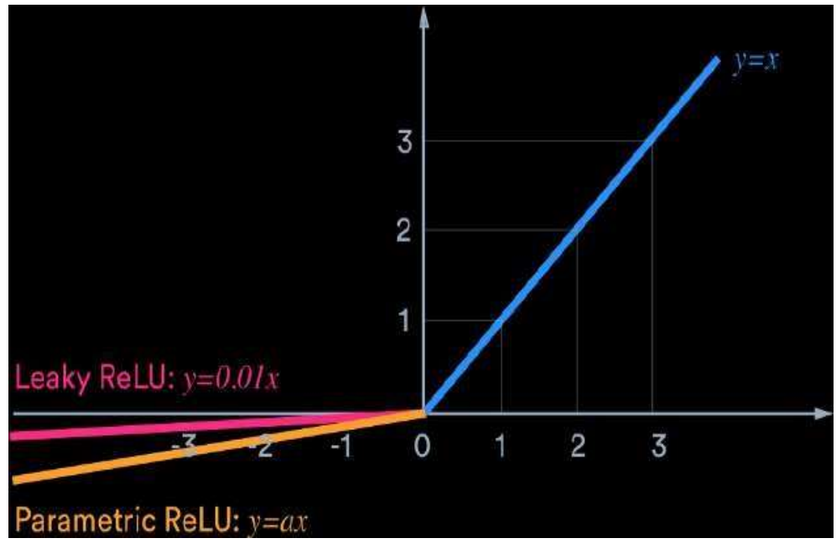
Input:

weighted sum of the inputs and biases of the neurons in a layer

Output:

The function simply outputs the value of 0 if it receives any negative input, but for any positive value z, it returns that value back like a linear function

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



Variation:

where slope is changed left of x=0 as shown in figure and thus causing a **leak** and extending the range of ReLU.

Introduce a small slope to keep the update alive

Advantage:

Prevents dying ReLU problem—this variation of ReLU has a small positive slope in the negative area, so it does enable backpropagation, even for negative input values

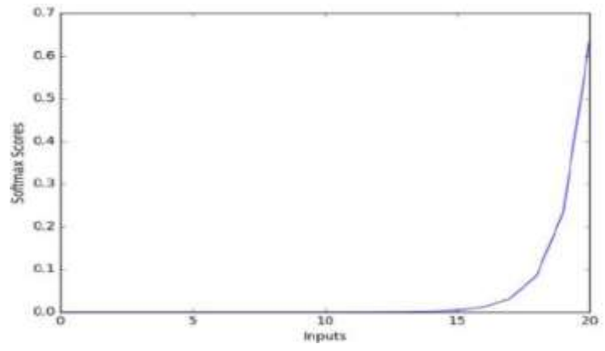
What is a Softmax Function?

Softmax is a very interesting activation function because it not only ***maps our output to a [0,1] range*** but also maps each output in such a way that the ***total sum is 1***. The output of Softmax is therefore a ***probability distribution***.

Mathematically Softmax is the following function where ***z*** is vector of inputs to output layer and ***j*** indexes the output units from ***1,2, 3 …. k*** :

Softmax Function

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$



In conclusion, Softmax is used for ***multi-classification in logistic regression model*** whereas Sigmoid is used for ***binary classification in logistic regression model***, the sum of probabilities is One for Softmax.

Using Activation Functions

- The nature of the activation in output layers is often controlled by the nature of output
  - Identity activation for real-valued outputs, and sigmoid/softmax for binary/categorical outputs.
  - Softmax almost exclusively for output layer and is paired with a particular type of *cross-entropy* loss.
- Hidden layer activations are almost always nonlinear and often use the same activation function over the entire network.
  - Tanh often (but not always) preferable to sigmoid.
  - ReLU has largely replaced tanh and sigmoid in many applications.

Loss Optimization

We want to find the network weights that ***achieve the lowest loss***

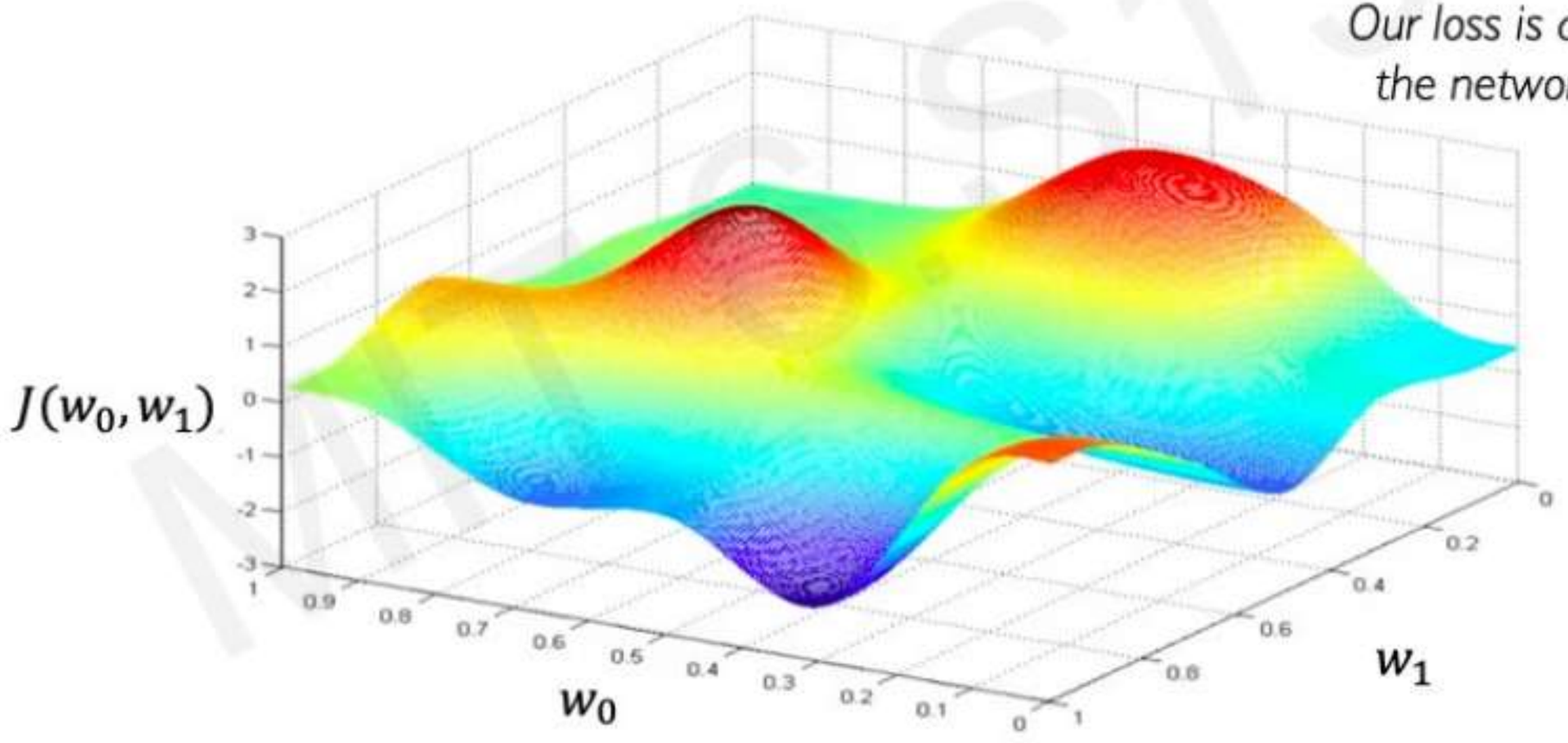
$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} J(\boldsymbol{W})$$

Loss Optimization

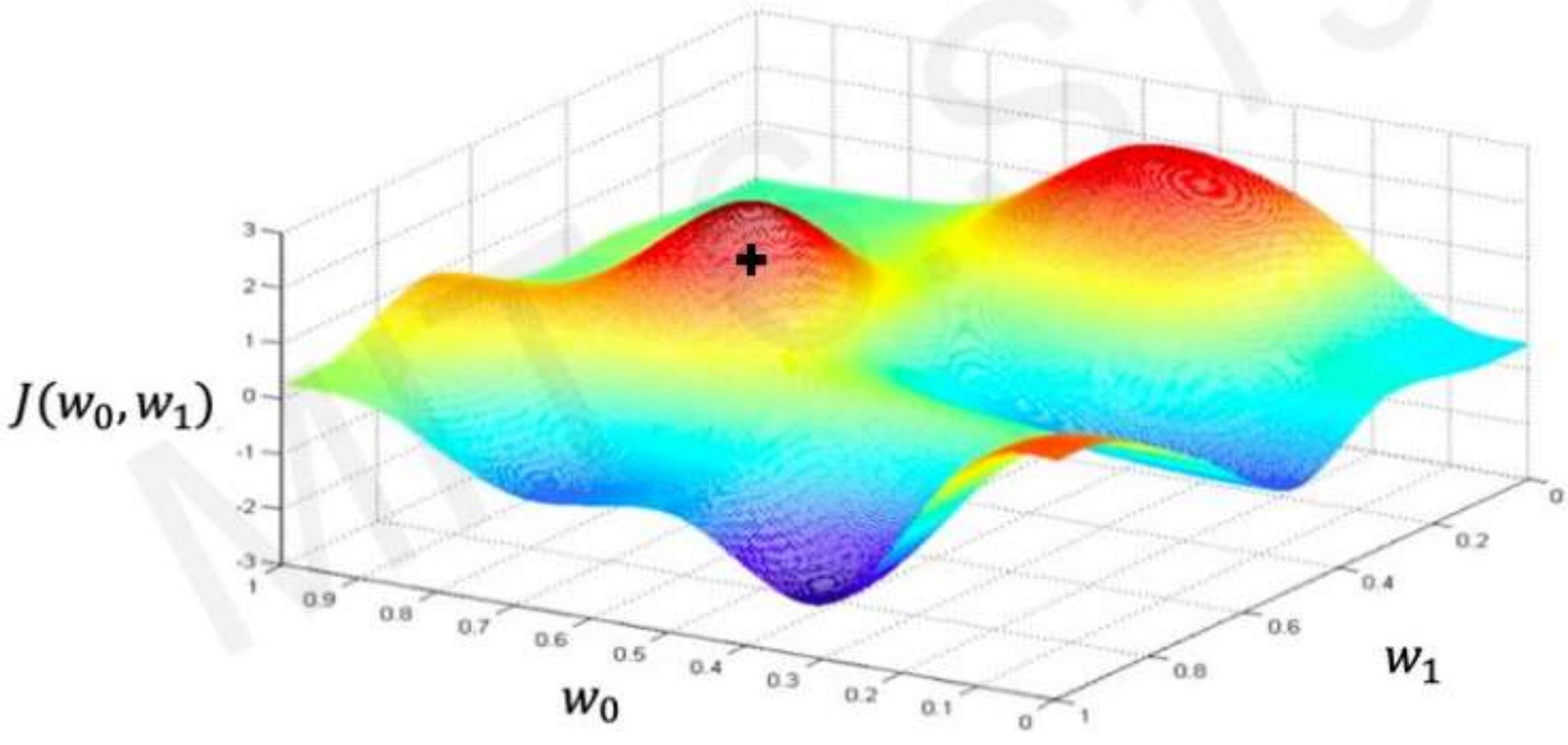
$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} J(\boldsymbol{W})$$

Remember:
Our loss is a function of the network weights!

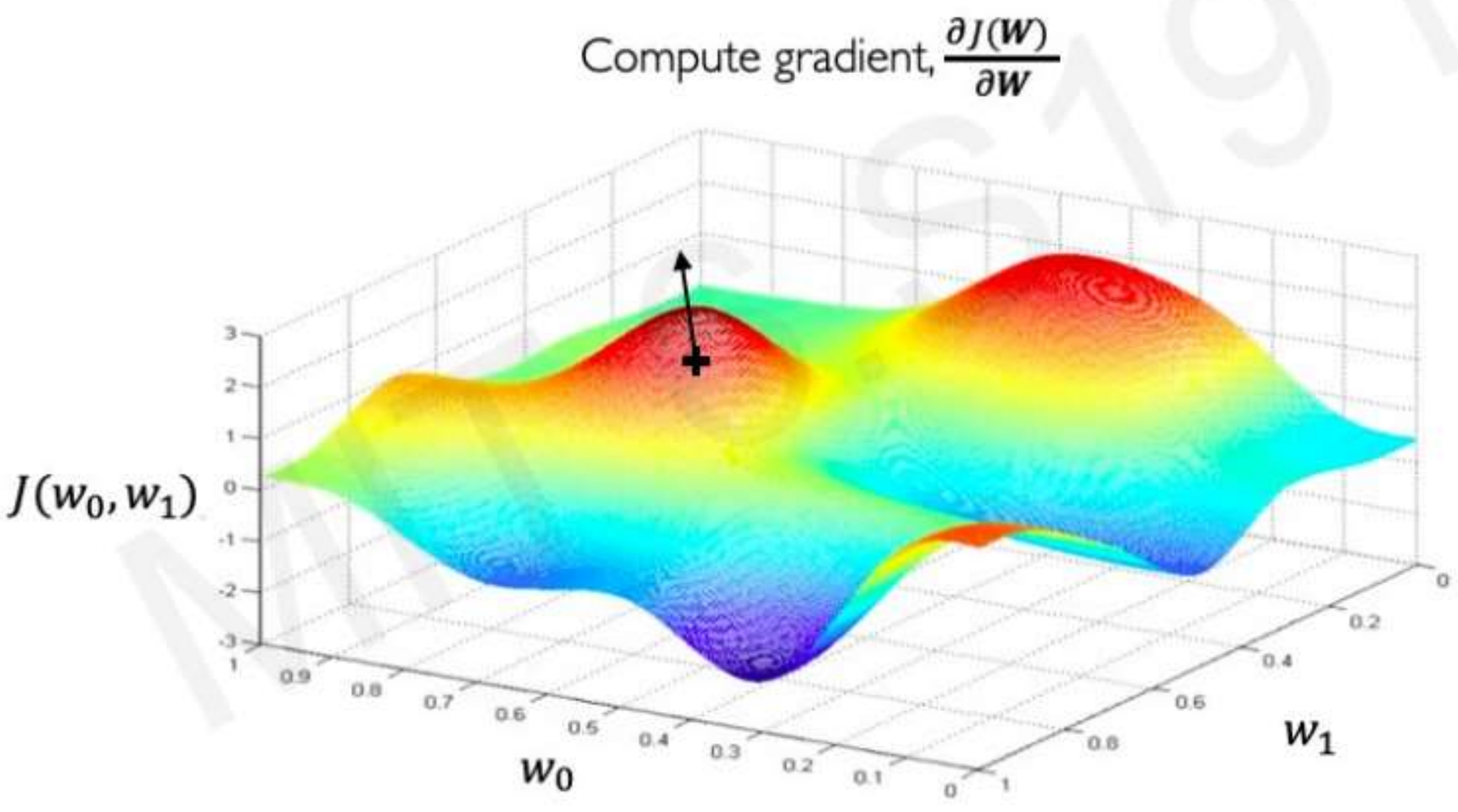


Loss Optimization

Randomly pick an initial (w_0, w_1)

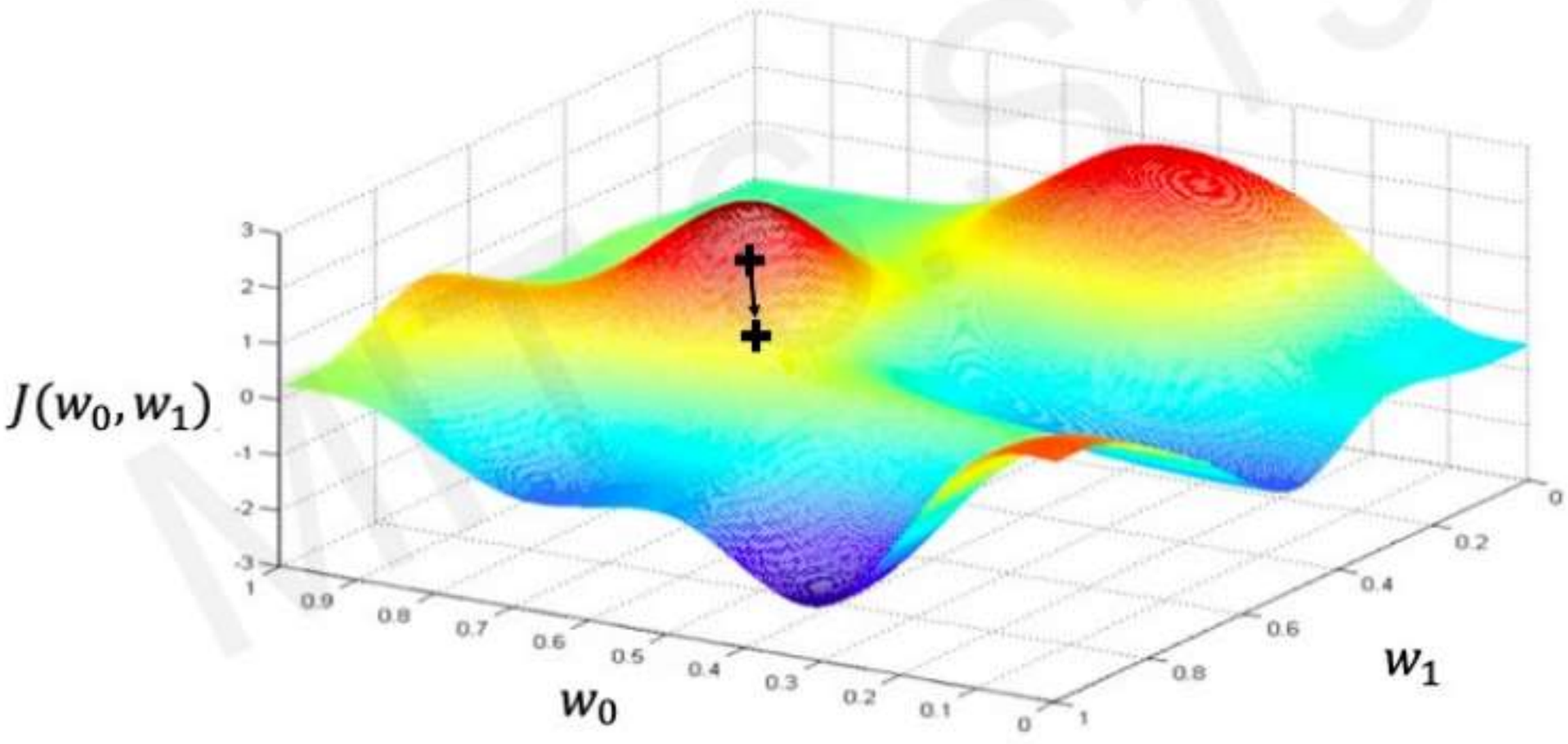


Loss Optimization



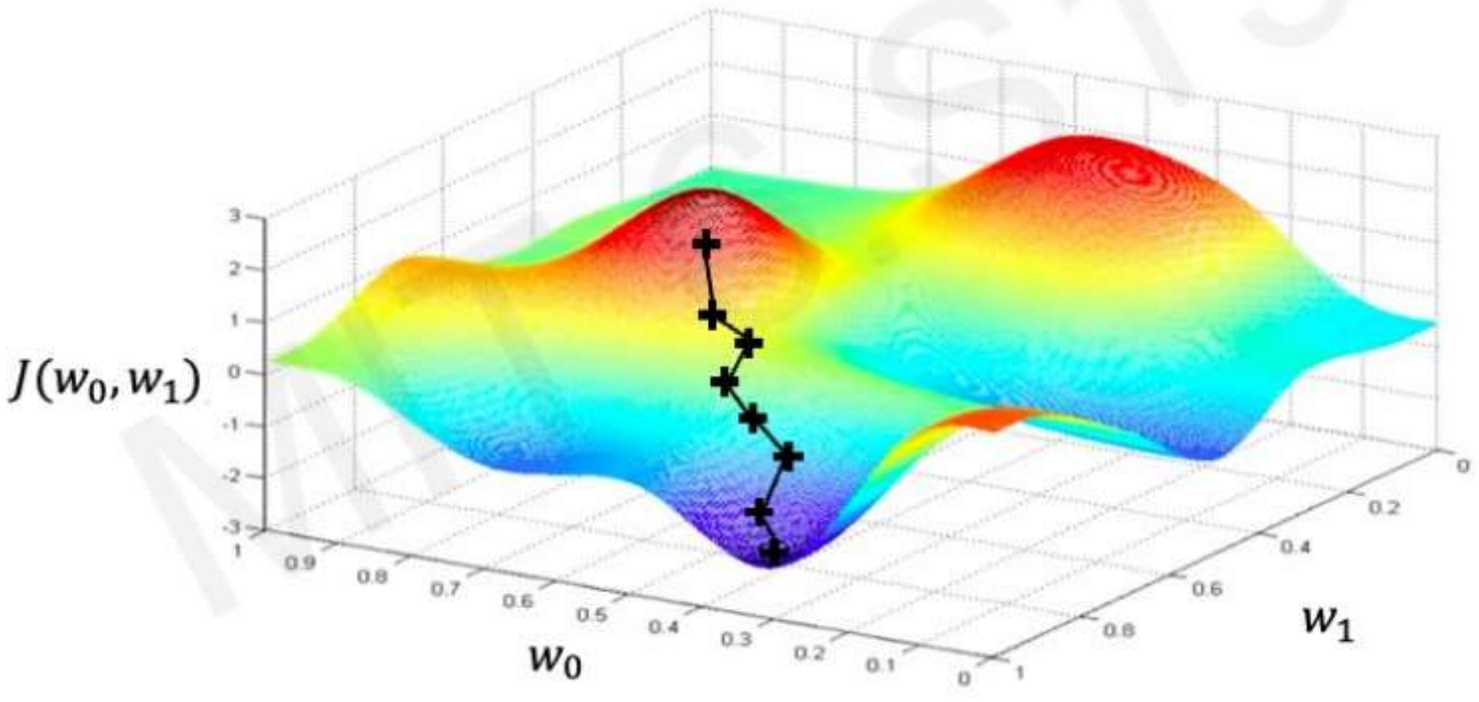
Loss Optimization

Take small step in opposite direction of gradient



Gradient Descent

Repeat until convergence



Computational Graph

Plotting computational graphs helps us visualize the dependencies of operators and variables within calculations

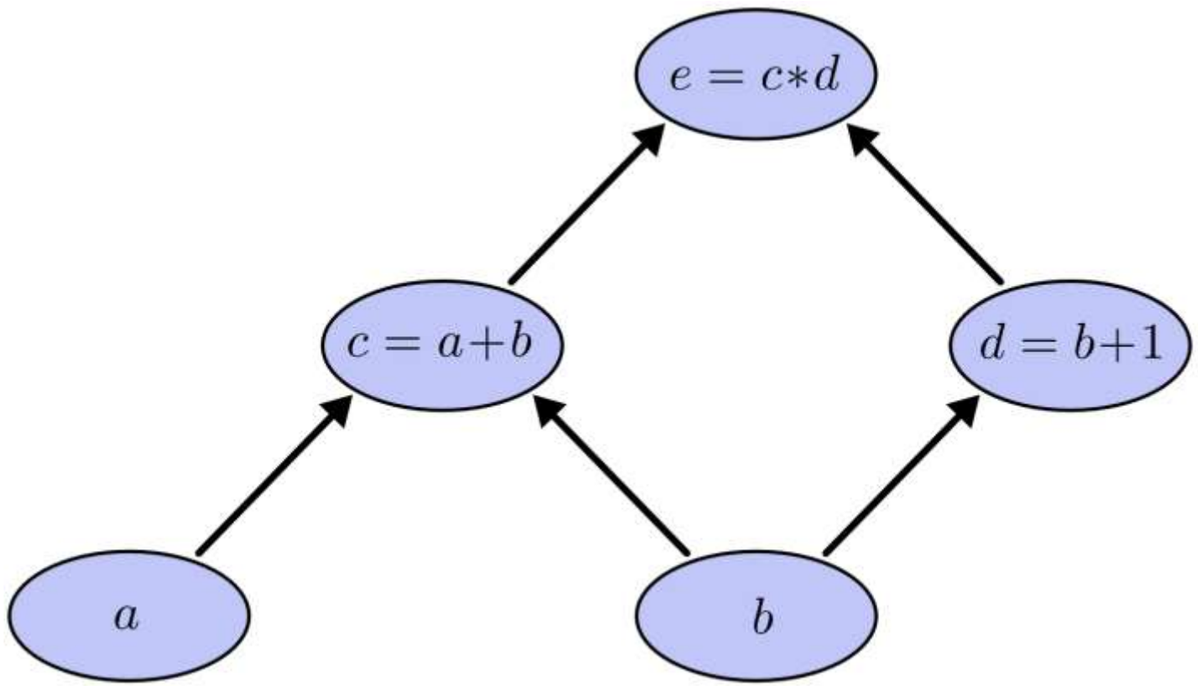
Computational graphs are a nice way to think about mathematical expressions.
For example, consider the expression $e=(a+b)*(b+1)$.
There are three operations: two additions and one multiplication.
let's introduce two intermediary variables, c and d so that every function's output has a variable

$c=a+b$
$d=b+1$
$e=c*d$

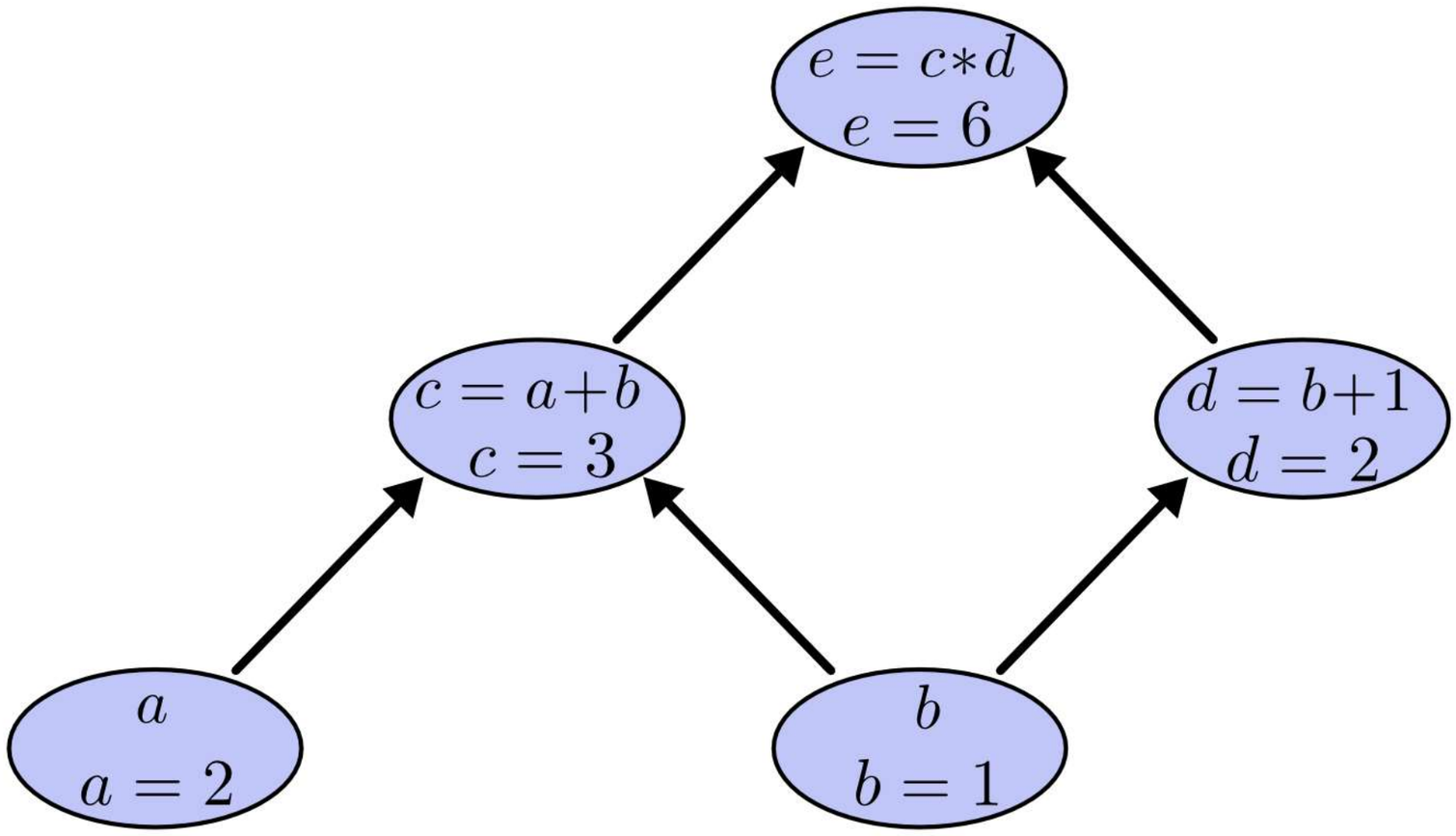
To create a computational graph, we make each of these operations, along with the input variables, into nodes. When one node's value is the input to another node, an arrow goes from one to another.

Courtesy: https://colah.github.io/posts/2015-08-Backprop/

Computational graph for e=(a+b)∗(b+1)



Computational graph helps in understanding both forward and backward propagation


$e = c*d$
$e = 6$
$c = a+b$
$c = 3$
$d = b+1$
$d = 2$
a
$a = 2$
b
$b = 1$

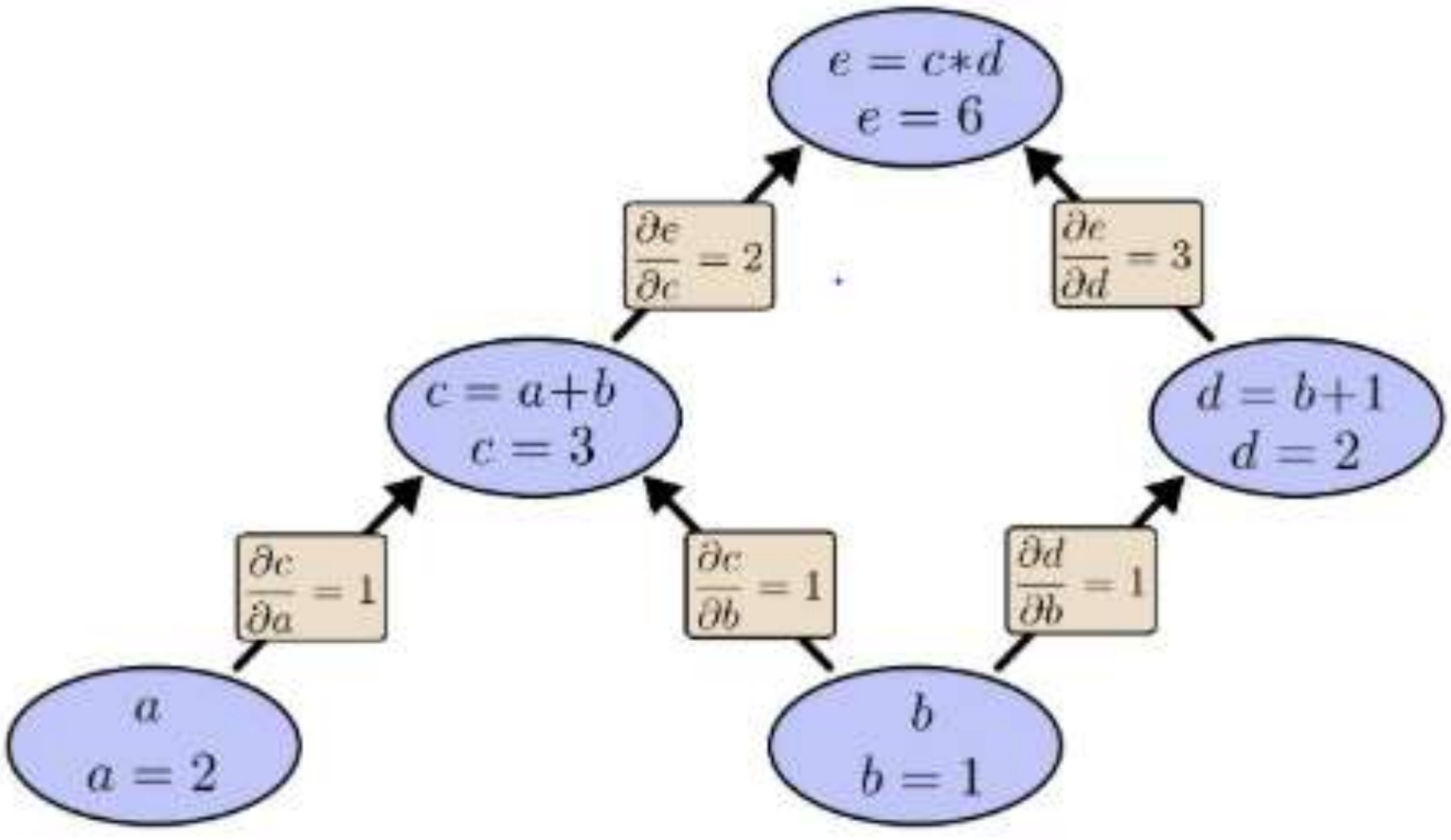
Derivatives on Computational Graphs

To evaluate the partial derivatives in this graph,
we need the sum rule and the product rule:

$$\frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$$

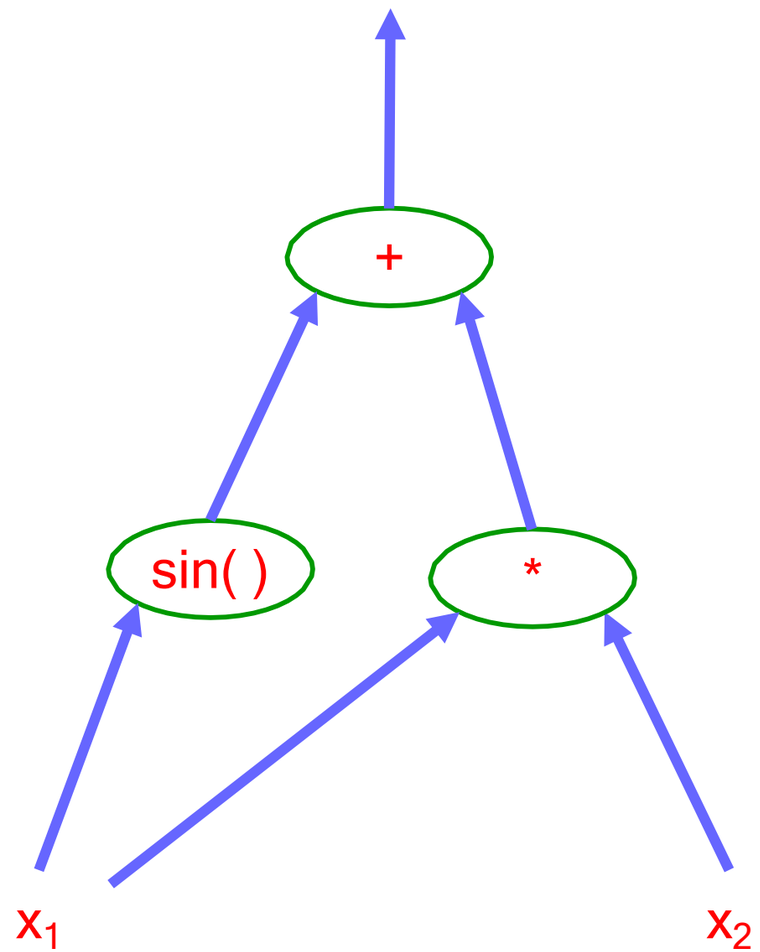
$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$

Derivatives on Computational Graphs



Computational Graph: Example

$$f(x_1, x_2) = x_1 x_2 + \sin(x_1)$$



Computing Gradients Backpropagation

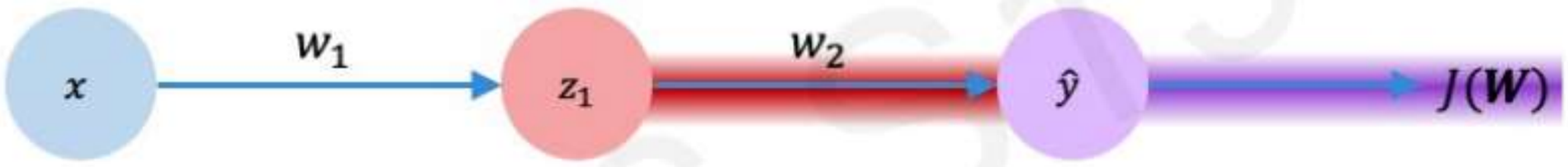
How does a small change in one weight (w2) affect the final loss J(w)

Computing Gradients: Backpropagation

$$\frac{\partial J(\boldsymbol{W})}{\partial w_2} =$$

Let's use the chain rule!

Computing Gradients: Backpropagation



$$\frac{\partial J(\boldsymbol{W})}{\partial w_2} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2}$$

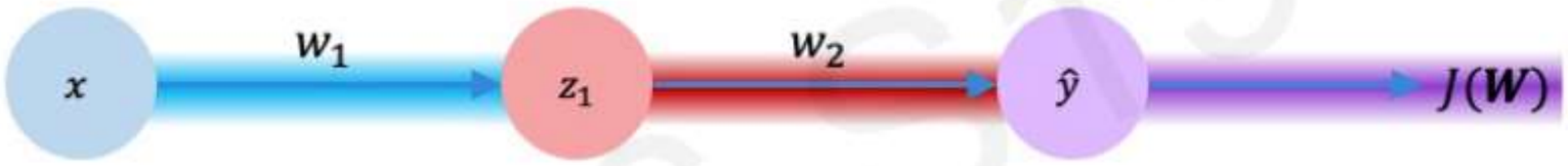
Computing Gradients: Backpropagation

$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

Apply chain rule!

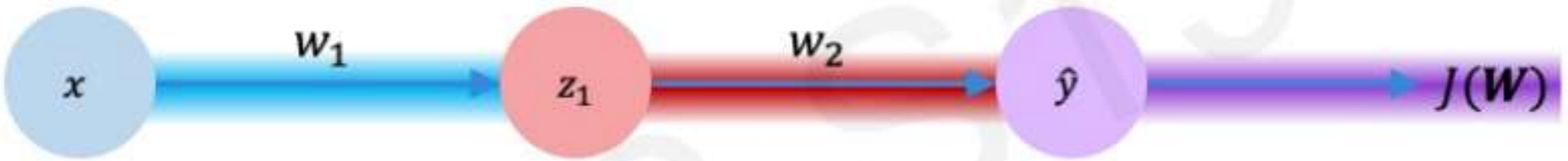
Apply chain rule!

Computing Gradients: Backpropagation



$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

Computing Gradients: Backpropagation



$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

Repeat this for ***every weight in the network*** *using gradients from later layers*

Training a Neural Network

Lets fix the pieces we have learnt

- Perceptrons
- Activation functions
- Loss functions, Optimization
- Computational graphs

- Construct the skeleton of neural network – Decide number of hidden layers, number of neurons
  - No of input neurons are based on input size
  - No of output neurons based on output size

  - Decide on activation and loss function based on either classification/regression problem
  - Start training - but how?

Training a Neural Network in simple words

1. Initialize Weights randomly
2. Forward propagation (summation + activation) from input layer to the output layer
3. Calculate loss
4. Based on loss, update weights (back propagation) from output layer to the input layer
5. Repeat step 2 to step 4 until loss is minimum

Training Neural Nets through Gradient Descent

Total training Loss:

$$Loss = \frac{1}{T}\sum_{t} Div(\boldsymbol{Y}_t, \boldsymbol{d}_t)$$

Assuming the bias is also represented as a weight

- Initialize all weights and biases $\{w_{ij}^{(k)}\}$
  - Using the extended notation: the bias is also a weight
- Do:
  - For every layer k for all i, j, update:
    - $w_{i,j}^{(k)} = w_{i,j}^{(k)} - \eta \frac{d\,Loss}{dw_{i,j}^{(k)}}$
- Until $Loss$ has converged

Optimizers

TOPICS IN DEEP LEARNING

Optimizers

- Optimizers are algorithm that is used to tweak the parameters such that we reach the minimum of the loss function.

- How should one change the weights and biases is defined by the optimizers we use.

- Optimizers have been undergoing constant evolution, adding techniques that makes them perform faster and more efficiently

TOPICS IN DEEP LEARNING

Optimizers

Different types of Optimizers used frequently:

1. Gradient Descent
2. Stochastic Gradient Descent
3. Mini-Batch Gradient Descent
4. SGD with Momentum
5. Adagrad
6. RMS-Prop
7. Adam (**Adaptive Moment Estimation** -RMS prop and momentum together)

TOPICS IN DEEP LEARNING

Optimizers - Gradient Descent

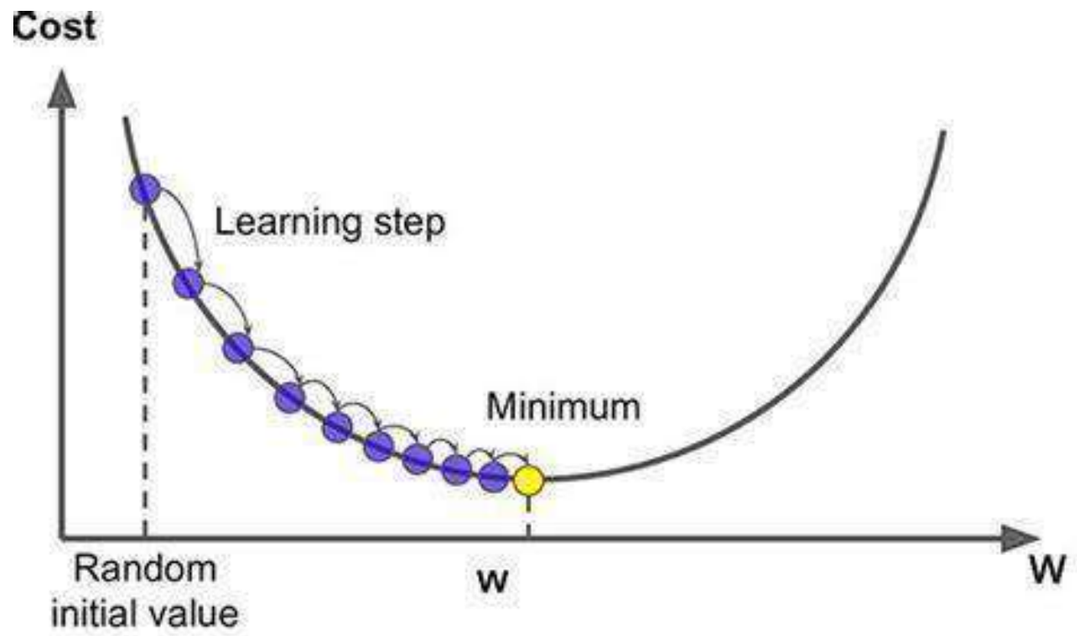
Gradient Descent is something we have learnt before.

- We find the gradient of the loss function with respect to the weights and biases
- And move in small steps towards the opposite direction of the gradient (Gradient points to ascent, we want direction of descent to reach the minimum)
- Gradient Descent iteratively reduces a loss function by moving in the direction opposite to that of steepest ascent.

Gradient Descent Algorithm:

$$\Theta_{n+1} = \Theta_n - \alpha \frac{\partial}{\partial \Theta_n} J(\Theta_n)$$

$\Theta \rightarrow$ Parameter Vector
$J \rightarrow$ Cost Function
$\alpha \rightarrow$ Slope Parameter



TOPICS IN DEEP LEARNING

Optimizers - Gradient Descent

Advantages

- Easy to implement
- Good results most of the times

Disadvantages

- Can get stuck in local minimas
- Because this method calculates the gradient for the entire data set in one update, the calculation is very slow
- It requires large memory and it is computationally expensive

TOPICS IN DEEP LEARNING

Optimizers - Stochastic Gradient Descent

Vanilla Gradient descent, find the gradient for the entire dataset, and the updates the parameters, this method is slow and hence the convergence is slow. Further this may not be suitable for huge datasets.

Stochastic GD is a variation of Vanilla Gradient descent were, the gradient is updated every data item, instead of after going through entire dataset.

- As the model parameters are frequently updated parameters have high variance and fluctuations in loss functions at different intensities
- SGD uses a higher number of iterations to reach the local minima. Due to an increase in the number of iterations, the overall computation time increases. But even after increasing the number of iterations, the computation cost is still less than that of the vanilla gradient descent optimizer.
- If the data is enormous and computational time is an essential factor, stochastic gradient descent should be preferred over batch gradient descent algorithm

TOPICS IN DEEP LEARNING

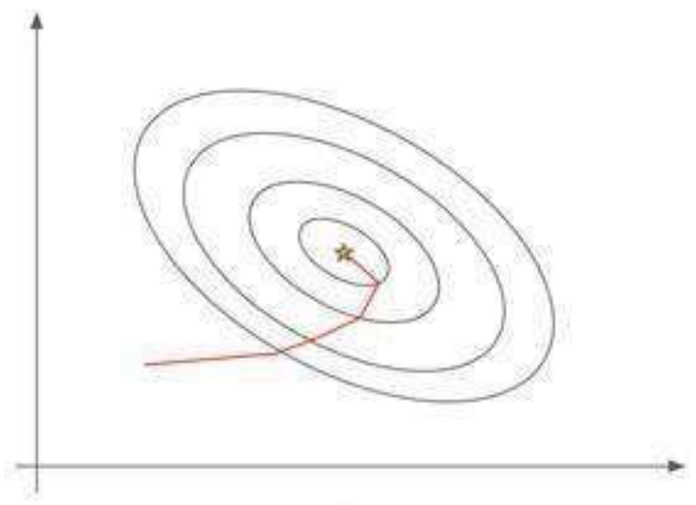
Optimizers - Stochastic Gradient Descent

Advantages

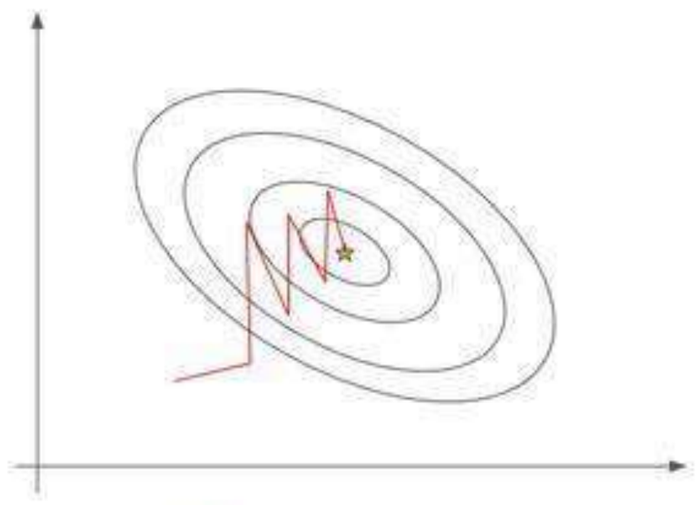
- Frequent Updation of parameters
- Less computationally expensive than Vanilla Gradient Descent
- Allows the use of large data sets as it has to update only one example at a time
- Uses less memory

Disadvantages

- The frequent can also result in noisy gradients which may cause the error to increase instead of decreasing it.
- High Variance.
- Frequent computationally expensive
- Still prone to local minima



Gradient Descent



Stochastic Gradient Descent

TOPICS IN DEEP LEARNING

Optimizers - Mini Batch Gradient Descent

Mini Batch Gradient Descent combines the ideas of Vanilla GD and Stochastic GD.
We shuffle the data, and choose a mini batch of *K* items, we then calculate gradients for this mini batch, and update the gradients.

This avoids the problem of computing gradients for the entire dataset and also the problem of frequent updation which causes noisy gradients. By finding a middle ground it is faster than both the variants.

- As the algorithm uses batching, all the training data need not be loaded in the memory, thus making the process more efficient to implement

- Moreover, the cost function in mini-batch gradient descent is noisier than the batch gradient descent algorithm but smoother than that of the stochastic gradient descent algorithm.

- Because of this, mini-batch gradient descent is ideal and provides a good balance between speed and accuracy.

TOPICS IN DEEP LEARNING

Optimizers - Mini Batch Gradient Descent

Batch vs. mini-batch gradient descent

$$X = [x^{[1]}\ x^{[2]}\ x^{[3]}\ \dots\ x^{[1000]}\,|\,x^{[1001]}\ \dots\ x^{[2000]}\,|\,\dots\,|\ \dots\ x^{[m]}]$$

(n_x, m) $\quad X^{\{1\}}(n_x, 1000)\quad X^{\{2\}}(n_x, 1000)\ \dots\quad X^{\{5000\}}(n_x, 1000)$

M=5,000,000 5000 mini batches of 1000 each

$$Y = [y^{[1]}\ y^{[2]}\ y^{[3]}\ \dots\ y^{[1000]}\,|\,y^{[1001]}\ \dots\ y^{[2000]}\,|\,\dots\,|\ \dots\ y^{[m]}]$$

$(1, m)$ $\quad Y^{\{1\}}(1, 1000)\quad Y^{\{2\}}(1, 1000)\ \dots\quad Y^{\{5000\}}(1, 1000)$

TOPICS IN DEEP LEARNING

Optimizers - Mini Batch Gradient Descent

- Size of the mini batch needs to be chosen carefully.
- If mini batch size =m, then it is batch gradient descent.
- If mini batch size=1, then all individual samples are mini batch them selves. Then training becomes very slow.
- The size of the mini batch need not be too small or too big. To take advantage of the CPU/GPU Memory mini batch size may be in the context of the available RAM.

- Generally the mini batch size is taken as power of 2
- If the training set is less than 2000, then there is no need of using mini-batch.

TOPICS IN DEEP LEARNING

Summary

- (mini batch size = m) ==> Batch gradient descent
- (mini batch size = 1) ==> Stochastic gradient descent (SGD)
- (mini batch size = between 1 and m) ==> Mini-batch gradient descent
- Mini-batch size is a hyperparameter

TOPICS IN DEEP LEARNING

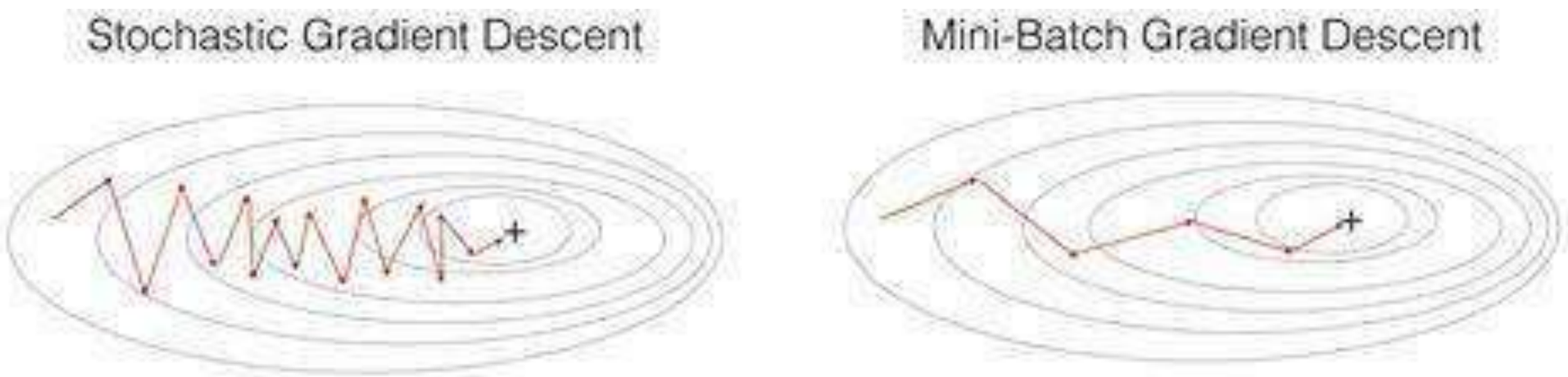
Optimizers - Mini Batch Gradient Descent

Advantages:

- Faster and more efficient than previous variants of GD
- More efficient gradient calculation
- Requires less amount of memory (Loads only one batch per updation)

Disadvantages:

- Still prone to local minima
- It introduces another hyperparameter *K* (the batch size)



TOPICS IN DEEP LEARNING

Challenges with all variants of GD

- Choosing an optimum value of the learning rate.
- If the learning rate is too small than gradient descent may take ages to converge.
- Having a constant learning rate for all the parameters.
- There may be some parameters which we may not want to change at the same rate.
- May get trapped at local minima.

Generalization in Multilayer Perceptrons (MLPs)

Generalization refers to the ability of a model to perform well on unseen data (test or validation datasets) after being trained on a specific dataset.
For MLPs, this capability is critical to ensure that the model doesn't just memorize the training data but learns patterns that apply broadly.

Need for Batch Normalization, Bias & Variance Tradeoff, Regularization, Optimizers

TOPICS IN DEEP LEARNING

Batch Normalization

Why normalization?

1. If the features are on different scale 1,1000 and 0,1 weights will end up taking different values.
2. More steps may be needed to reach optimal value and the learning can be slow.
3. Shape of the normalized bowl will be spherical and symmetrical making easier to faster to optimize.

$$\mu = \frac{1}{m}\Sigma_{i=1}^{m} x^{(i)}$$
$$x = x - \mu$$

Subtract mean

$$\sigma^2 = \frac{1}{m}\Sigma_{i=1}^{m} x^{(i)} ** 2$$
$$x /= \sigma^2$$

Normalize variance

TOPICS IN DEEP LEARNING

Batch Normalization- Summary

Idea behind Batch Normalization

- It is well known that the inputs to a neural network should be normalized before passing them to the network.

    - This usually involves transforming the inputs to have a mean of zero and unit variances.
    - The inputs should also be decorrelated. All of this is done in order to aid learning.

- So if normalizing the inputs to the input layer improves learning, then normalizing the inputs to the hidden layers must also improve learning

Normalizing the hidden layer inputs is basically called Batch Normalization. It helps in faster convergence and learning and is very frequently used technique to train models faster.

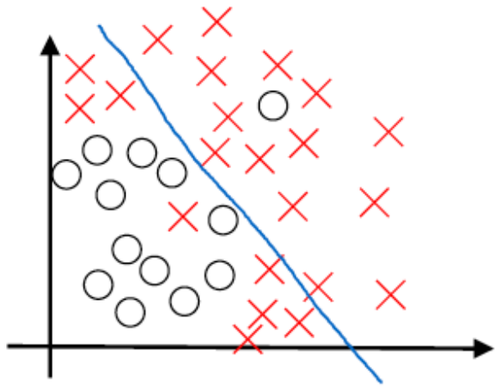
TOPICS IN DEEP LEARNING

Batch Normalization

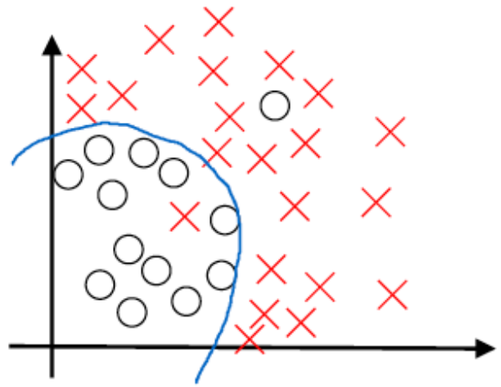
Advantages of Batch Normalization

- Networks train faster
- Allows for higher learning rates
- Makes weights easier to initialise
- Provides some regularization

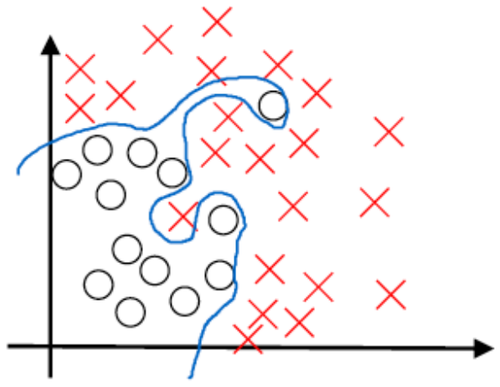
Bias and Variance: Introduction



high bias



"just right"



high variance

- Bias / Variance techniques are Easy to learn, but difficult to master.
- So here the explanation of Bias / Variance:
  - If your model is underfitting (logistic regression of non linear data) it has a "high bias"
  - If your model is overfitting then it has a "high variance"
  - Your model will be alright if you balance the Bias / Variance

https://github.com/ashishpatel26/Andrew-NG-Notes/blob/master/andrewng-p-2-improving-deep-learning-network.md#bias--variance

TOPICS IN DEEP LEARNING

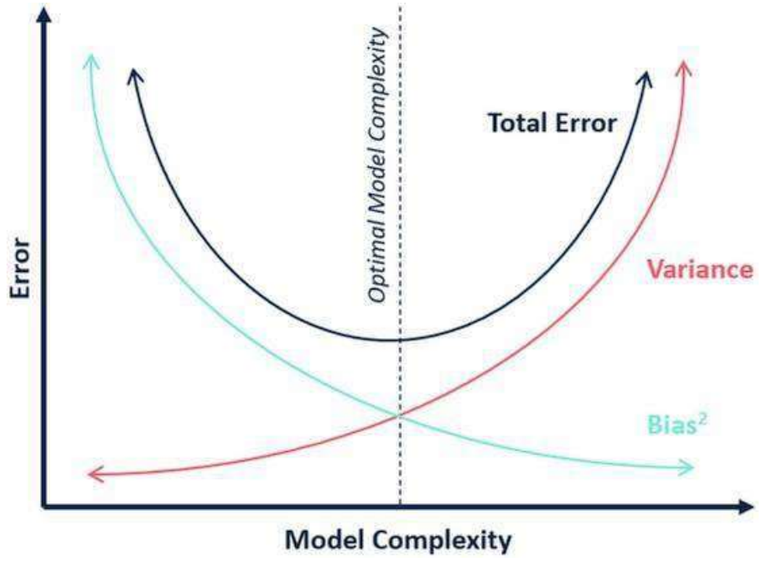
To reduce High Bias

1	Increase the model size(Try to make your NN bigger (size of hidden units, number of layers)	It allows to fit the training set better. If you find this increases variance, then use regularization, which will usually eliminate variance
2	Modify input features based on insight from error analysis	Create additional features that help the algorithm eliminate a particular category of errors. These new features could help with both bias and variance
3	Reduce or eliminate regularization(L1,L2,drop out)	Reduces avoidable bias, but increases variance
4	Try to run it longer	Might reduce the bias
5	Modify model architecture	This can affect both bias and variance

https://github.com/ashishpatel26/Andrew-NG-Notes/blob/master/andrewng-p-2-improving-deep-learning-network.md#bias--variance

Bias and Variance Trade Off

- There is inverse relationship between bias and variance in machine learning.
- Increasing the bias will decrease the variance.
- Increasing the variance will decrease the bias.
- If our model is too simple and has very few parameters then it may have high bias and low variance. On the other hand if our model has large number of parameters then it's going to have high variance and low bias.
- We need to find the right/good balance without overfitting and underfitting the data.
- This tradeoff in complexity is why there is a tradeoff between bias and variance. An algorithm can't be more complex and less complex at the same time.



TOPICS IN DEEP LEARNING

To reduce High Variance

1	Add more training data	Simplest and most reliable way to address variance, so long as you have access to significantly more data and enough computational power to process the data
2	Add regularization(L2, L1 regularization, dropout, data augumentation)	This reduces variance, but increases bias
3	Add early stopping(stop gradient descent early based on dev set error)	reduces variance, but increases bias
4	Modify model architecture	This can affect both bias and variance

Reference: https://towardsdatascience.com/regularization-in-machine-learning-76441ddcf99a

Regularization

How do you prevent overfitting?

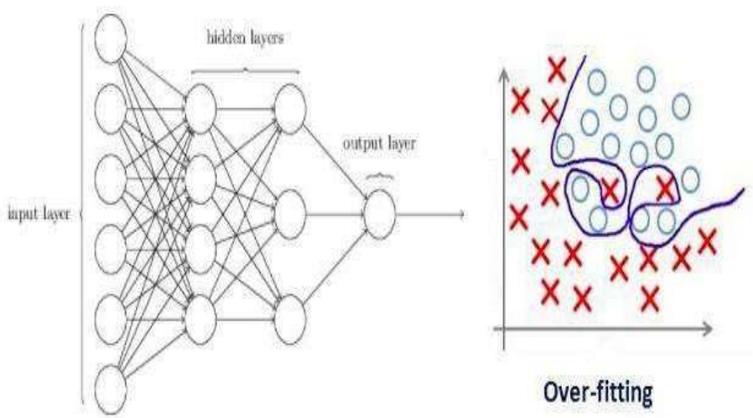
Regularization

Reference: https://towardsdatascience.com/regularization-in-machine-learning-76441ddcf99a

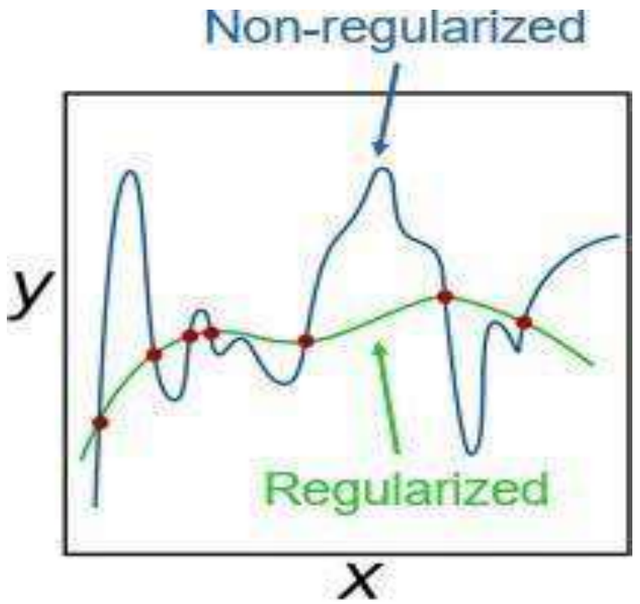
TOPICS IN DEEP LEARNING

Regularization

- Regularization is a technique which makes slight modifications to the learning algorithm such that the model generalizes better. This in turn improves the model's performance on the unseen data as well.



- Regularization is a process of introducing additional information in order to prevent overfitting.



Reference: https://towardsdatascience.com/regularization-in-machine-learning-76441ddcf99a

Regularization

Types of Regularization

- Dropout
- Data Augmentation
- Early stopping

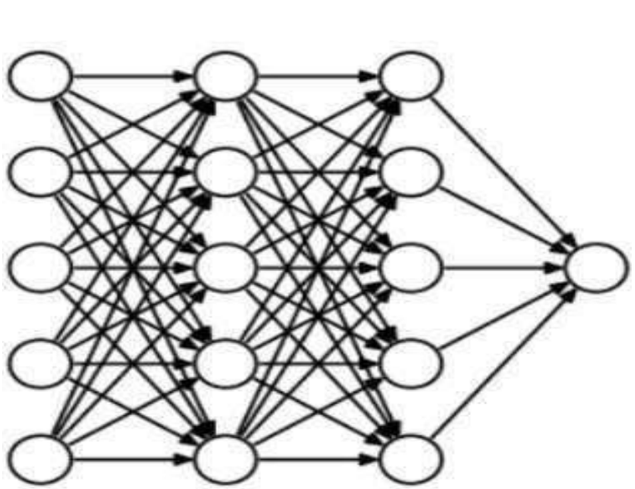
Type of Regularization

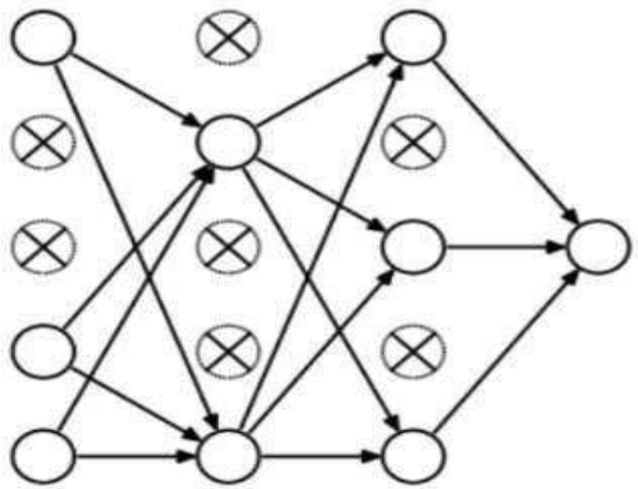
Dropout

Dropout

This is the one of the most interesting types of regularization techniques. It also produces very good results and is consequently the most frequently used regularization technique in the field of deep learning.

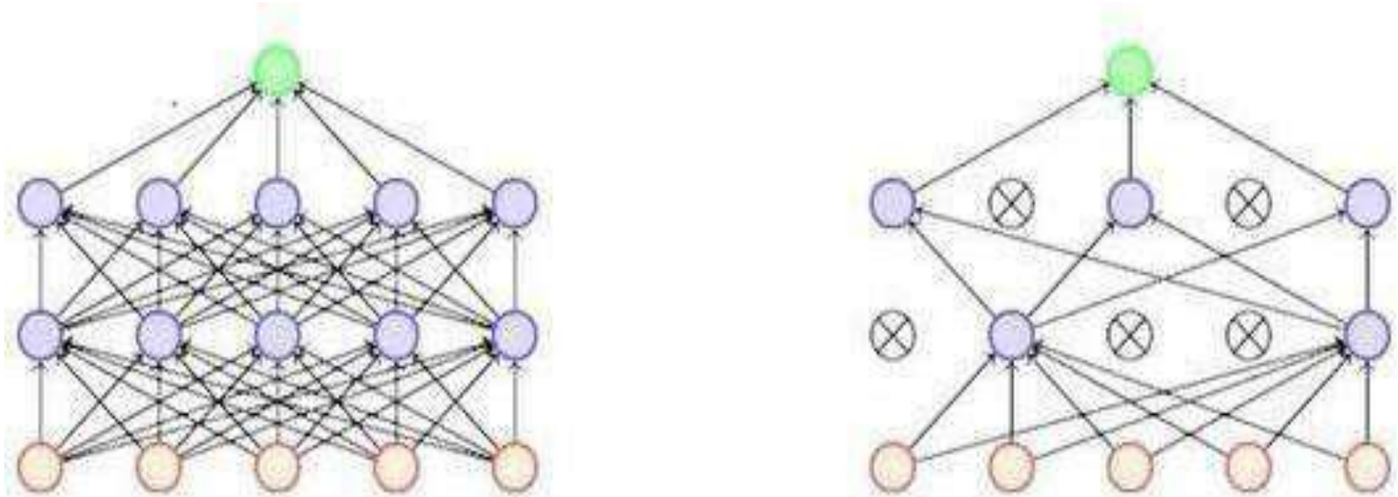
To understand dropout, let's say our neural network structure is akin to the one shown below





So what does dropout do? At every iteration, it randomly selects some nodes and removes them along with all of their incoming and outgoing connections as shown in right figure

What is Dropout?



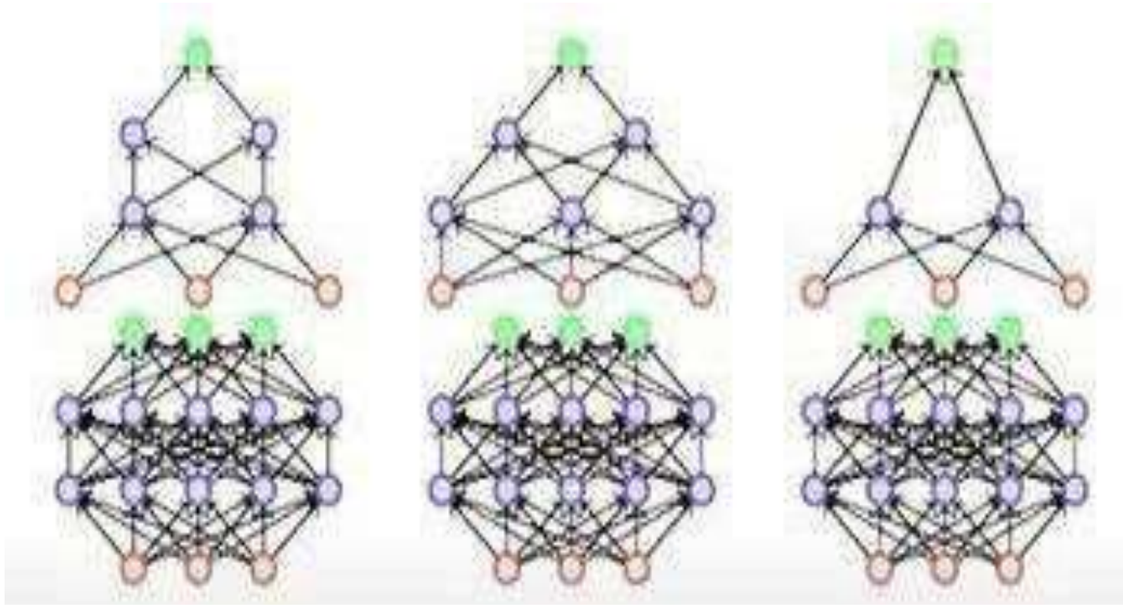
- Dropout means leaving some units
- Temporarily remove a node and both its incoming and outgoing connections.
- This thins the neural network

Note: **Given a total of “n” nodes what are the total number of thinned networks that can be formed?**

2^n

Why dropout regularization?

- Train several neural networks having different architectures
- Train multiple instances of the same network using different training samples.



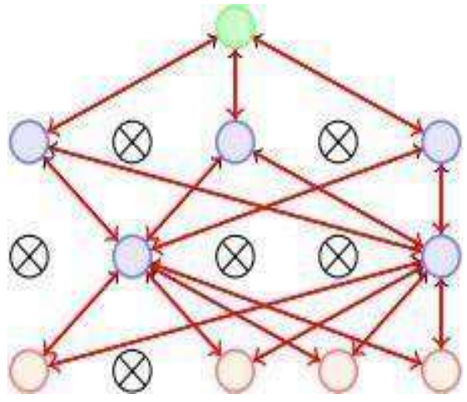
“ Expensive options”

Both at training as well as at testing levels

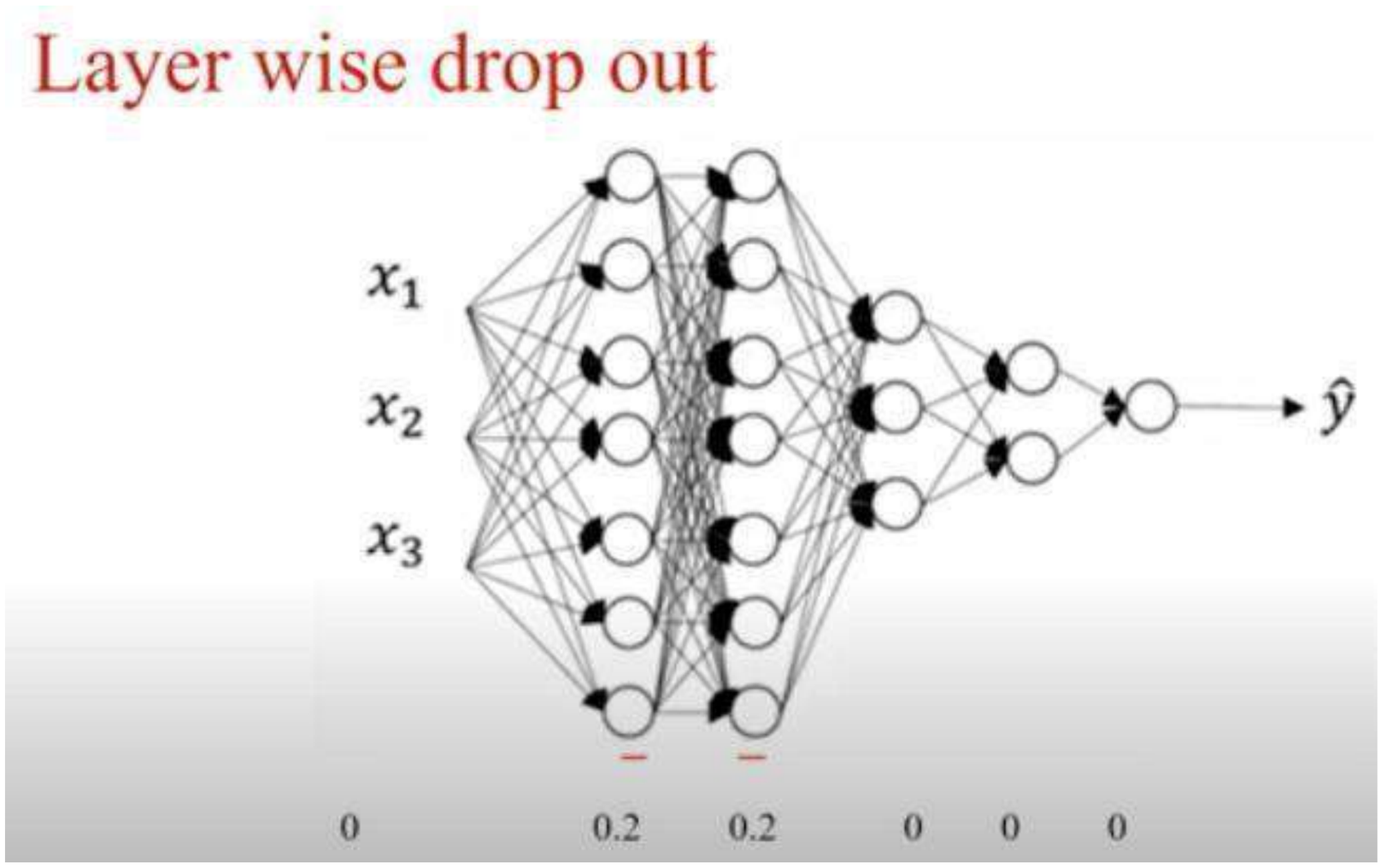
Hence solution is dropout

Dropout Training

- Dropout essentially applies a masking noise to the hidden units
- Prevents hidden units from coadapting
- Essentially a hidden unit cannot rely too much on other units as they may get dropped out any time
- Each hidden unit has to learn to be more robust to these random dropouts

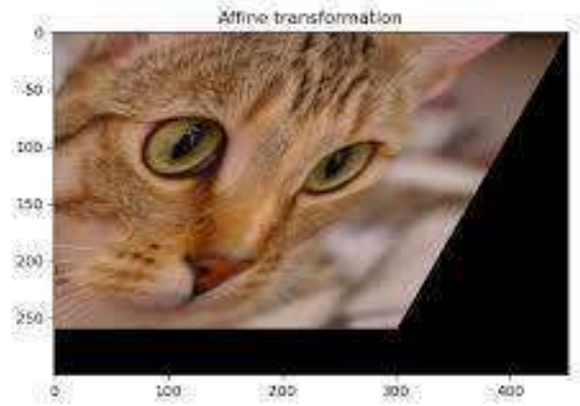


Dropout Training



Data Augmentation

1. More training data is one more solution for over fitting
2. As getting additional data may be expensive and may not be possible
3. Flipping of all images can be one of the ways to increase the data set size.
4. Randomly zooming in and zooming out can be another way.
5. Distorting some of the images based on your application may be an another way to increase the data set size.



Type of Regularization

Data Augmentation

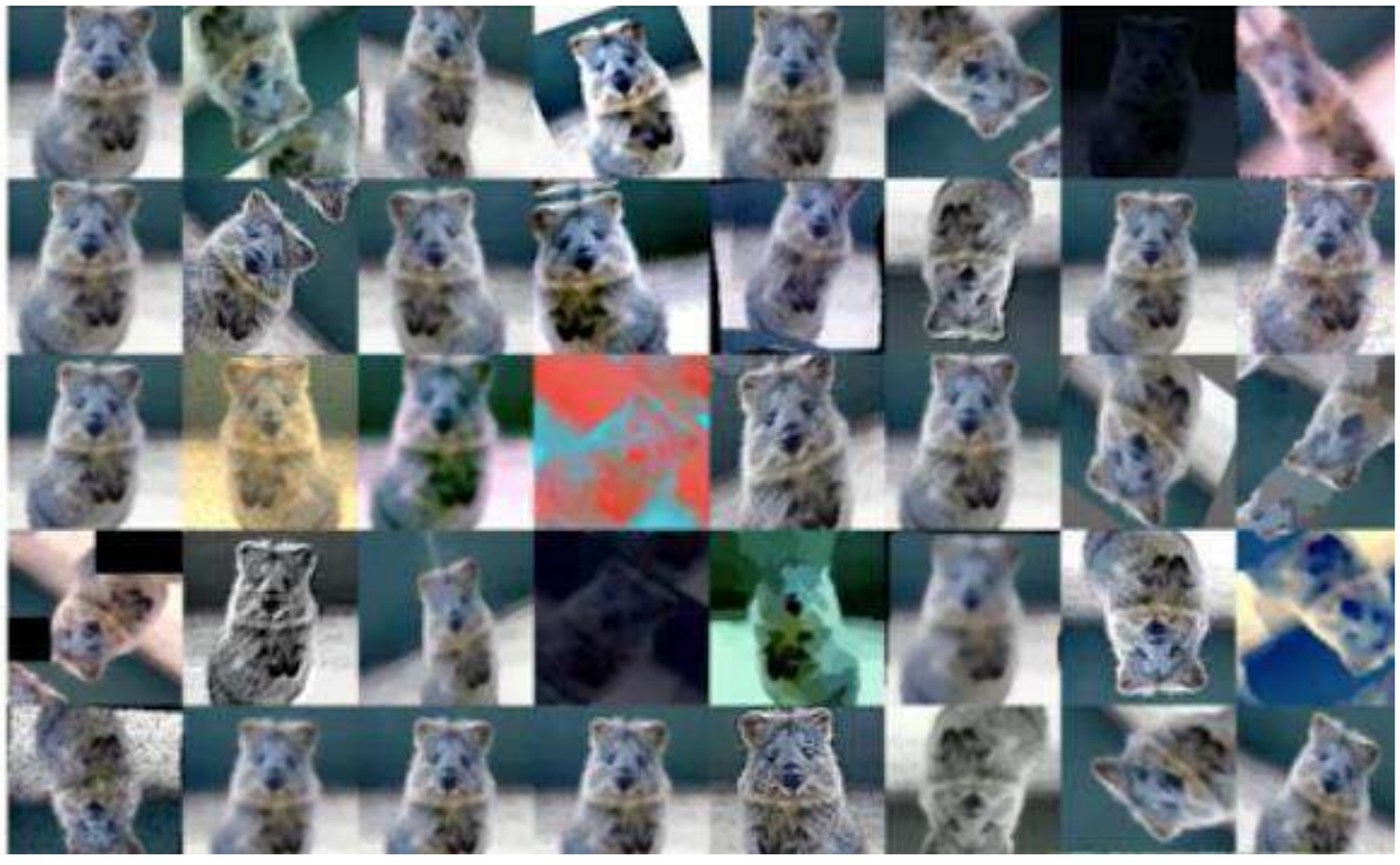
Affine: Translate

Affine: Rotate

Affine: Shear

Type of Regularization

Data Augmentation



Noise Robustness

TOPICS IN DEEP LEARNING

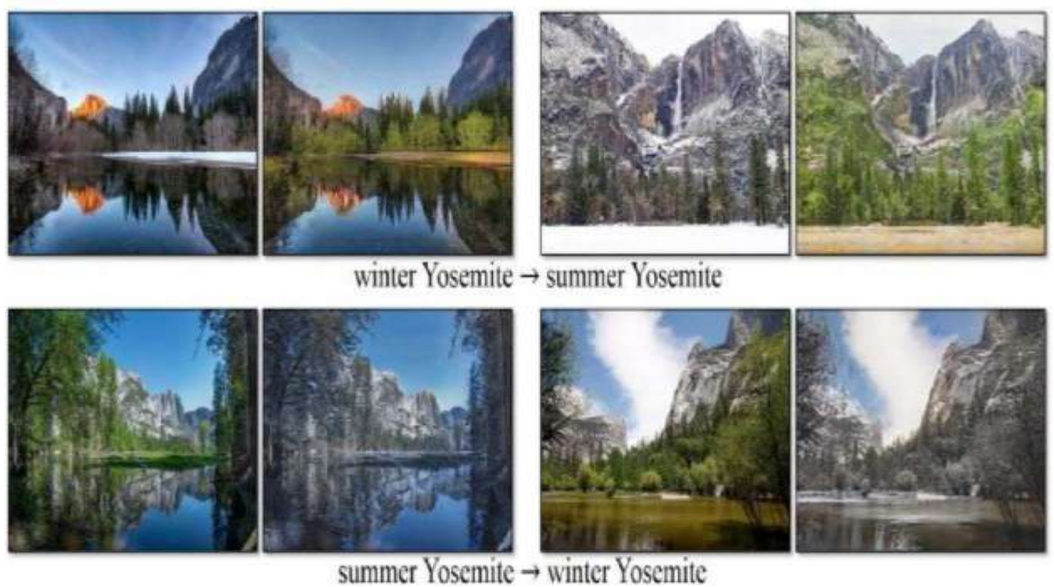
Data Augmentation

Popular Augmentation Techniques:

1. **Flip**
2. **Rotation**
3. **Scale**
4. **Crop**
5. **Translation**
6. **Gaussian Noise**
7. **Conditional GANs (Advanced Augmentation Technique)**

Conditional GANs can transform an image from one domain to an image to another domain.

An example of conditional GANs used to transform photographs of summer sceneries to winter sceneries.



Reference: https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/

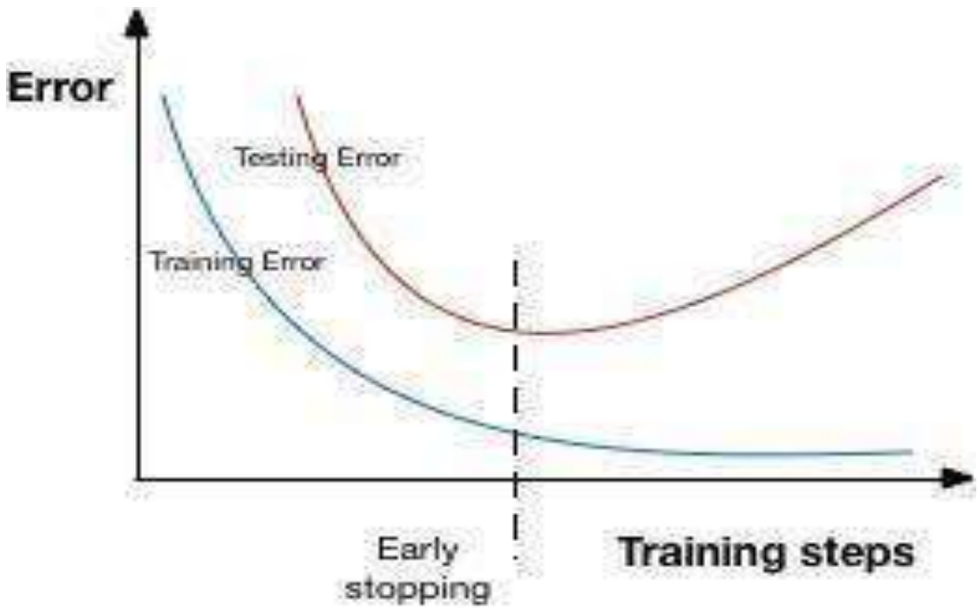
Type of Regularization

Early Stopping

TOPICS IN DEEP LEARNING

Early Stopping

Early stopping is a kind of cross-validation strategy where we keep one part of the training set as the validation set. When we see that the performance on the validation set (or test set) is getting worse, we immediately stop the training on the model. This is known as early stopping.



The above image, we will stop training at the dotted line since after that our model will start overfitting on the training data.

How long to train a neural network?

- Too little training will mean that the model will underfit the train and the test sets.

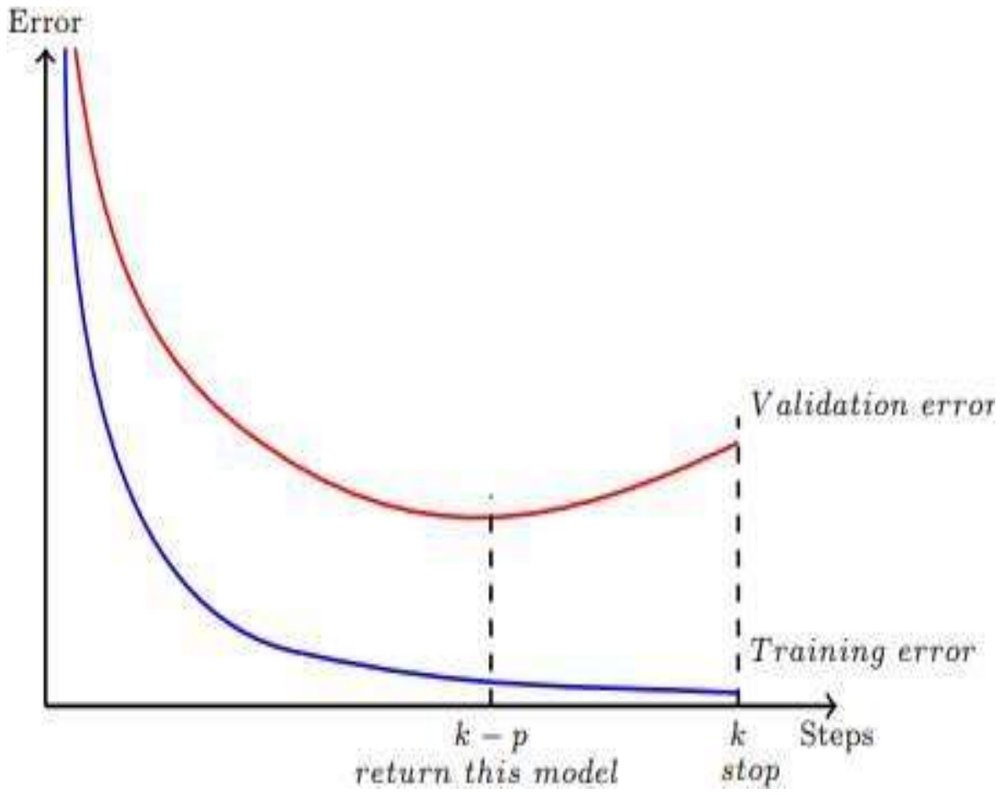
- Too much training will mean that the model will overfit the training dataset and have poor performance on the test set.

- A compromise is to train on the training dataset but to stop training at the point when performance on a validation dataset starts to degrade.

- This simple, effective, and widely used approach to training neural networks is called early stopping.

Early Stopping

- Track the validation error
- Have a patience parameter p
- If you are at step k and there was no improvement in validation error in the previous p steps then stop training and return the model stored at step k −p
- Basically, stop the training early before it drives the training error to 0 and blows up the validation error



Its advantage is that you don't need to search a hyperparameter like in other regularization approaches (like lambda in L2 regularization).

Summary- Why so much care?

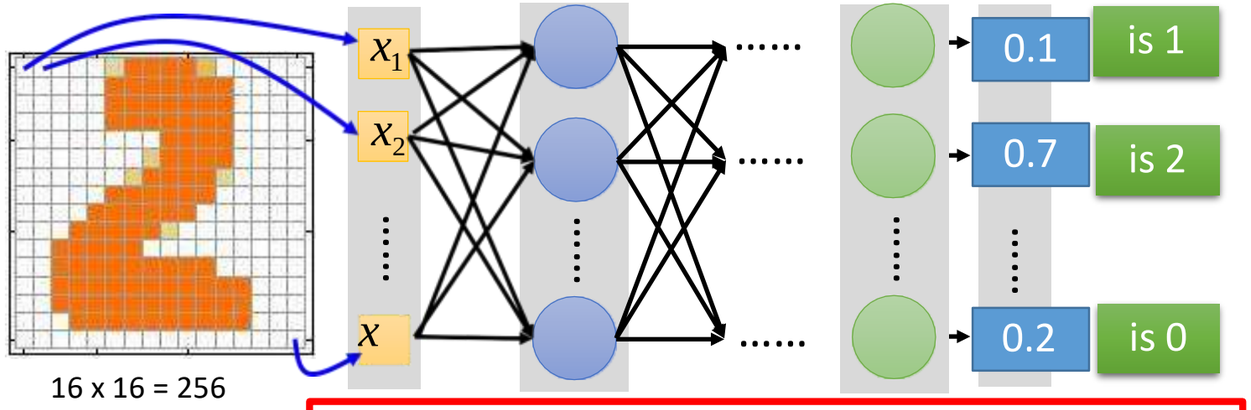
- Deep Neural networks are highly complex models.
- Many parameters, many non-linearities.
- It is easy for them to overfit and drive training error to 0. Hence we need some form of regularization.

TOPICS IN DEEP LEARNING

Fine-Tuning Hyper parameters

Fine-Tuning Hyper parameters

$$\theta = \{W^1, b^1, W^2, b^2, \cdots W^L, b^L\}$$



Set the network parameters θ such that ……

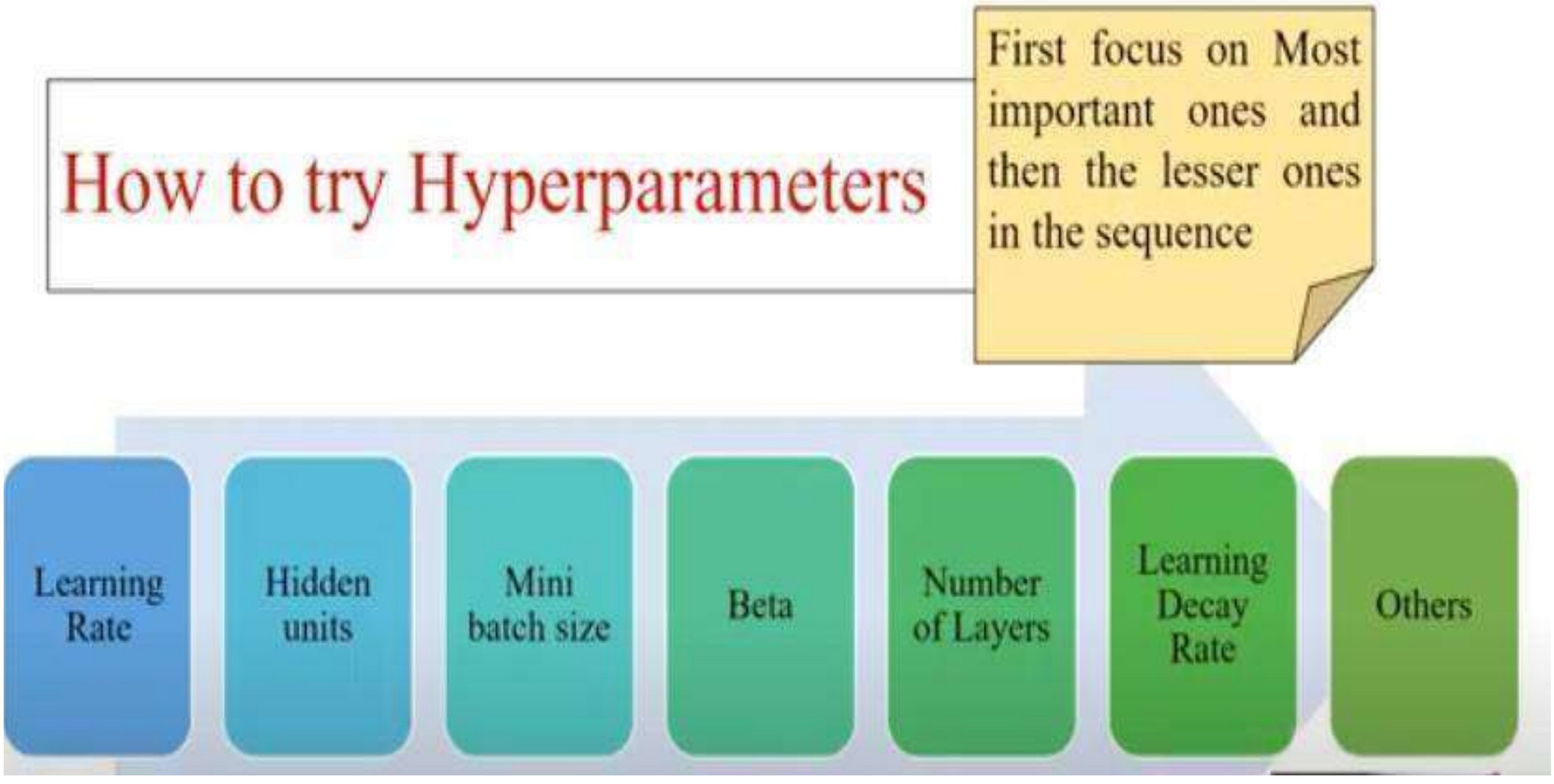
Inp … um value

Inp … um value

How to let the neural network achieve this

How to try Hyperparameters?

.



Deep Neural Network: Parameters vs Hyperparameters

Parameters:

$$W^1, b^1, W^2, b^2, \cdots W^L, b^L$$

Hyperparameters:

- **Learning rate** a **in gradient descent**
- **Number of iterations in gradient descent**
- **Number of layers in a Neural Network**
- **Number of neurons per layer in a Neural Network**
- **Activations Functions**
- **Mini-batch size**
- **Regularizations parameters**

- **You have to try values yourself of hyper parameters.**

Fine-Tuning Hyper parameters- Grid Search

Taken from the imperative command "Just try everything!" comes Grid Search – a naive approach of simply trying every possible configuration.

Here's the workflow:

•Define a grid on *n* dimensions, where each of these maps for an hyperparameter. e.g. *n* = (learning_rate, dropout_rate, batch_size)

•For each dimension, define the range of possible values:
e.g. batch_size = [4, 8, 16, 32, 64, 128, 256]

•Search for all the possible configurations and wait for the results to establish the best one:

e.g. *C1* = (0.1, 0.3, 4) -> acc = 92%,
C2 = (0.1, 0.35, 4) -> acc = 92.3%, etc...

Extra Information

Gradient Descent

- Gradient Descent searches the hypothesis space for possible weight vectors to find the most optimal pair of weights for the training example.

- In any training scenario, we will have a loss function to judge how well the weights have fit for the model.

- What gradient descent does essentially is to take a small step in a direction and try to reach the minimal point it can in that direction as soon as possible.

- Choosing this direction needs to be the direction of the steepest descent. That can be found using the derivative of our loss function.

- Lets take the example of loss function to be $E(w) = \frac{1}{2}\Sigma(t_d - o_d)^2$.
    - t is the target output and o is the actual output.

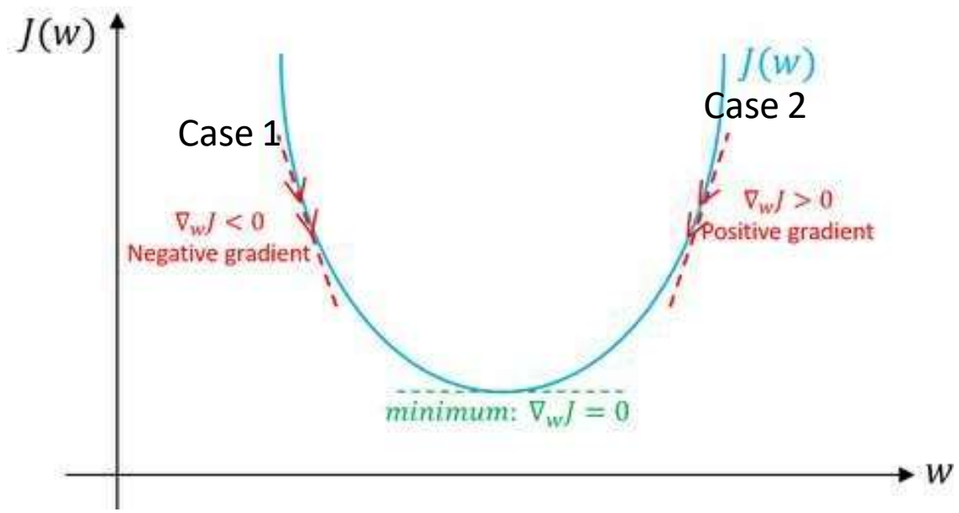
Gradient Descent

In any case, the gradient descent weight change will ensure that we are heading towards the minima.

$$w_i = w_i - \eta \nabla E(w)$$

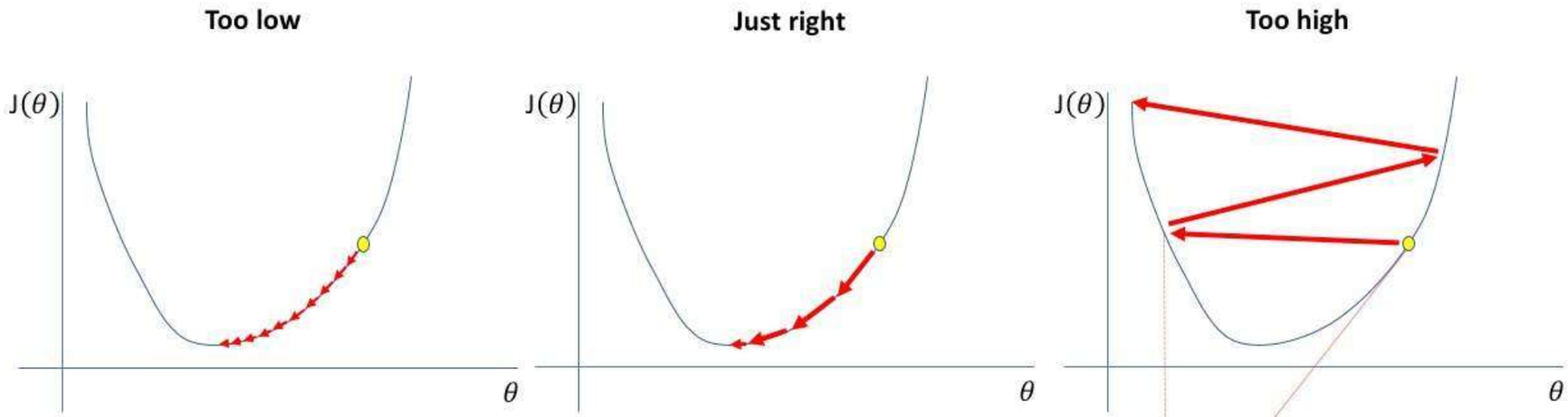
In the case that the gradient is negative, the weight will be updated to increase and thus move to the right as shown in case 1.

In the case that the gradient is positive, the weight will be updated to decrease and thus move to the left as shown in case 2.



Learning rate in gradient descent

η is called the learning rate. This is used to determine how much the weights should change by. This is a very important hyper-parameter. This value must be chosen properly.



A small learning rate requires many updates before reaching the minimum point

The optimal learning rate swiftly reaches the minimum point

Too large of a learning rate causes drastic updates which lead to divergent behaviors

Neural Network and Deep Learning

Activation Functions

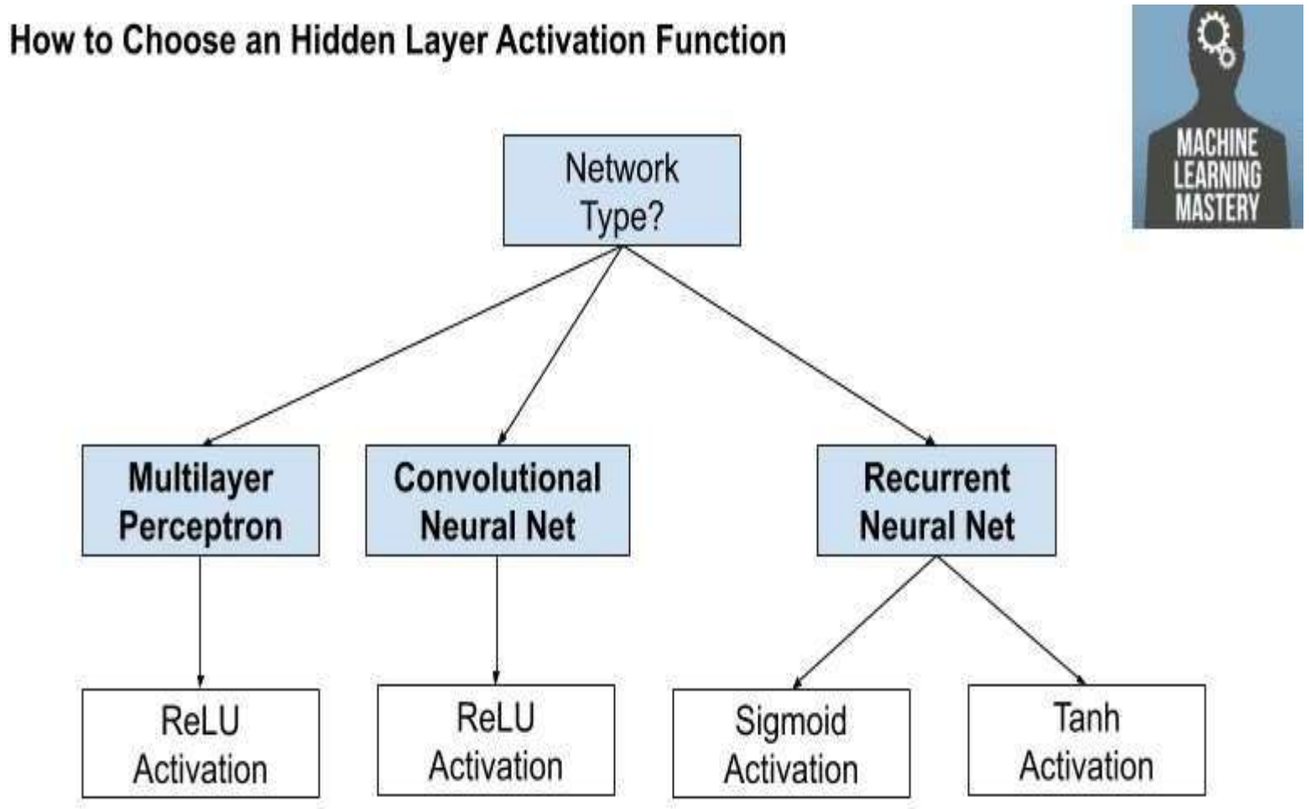
- We have to choose activation functions for the hidden layer and the output layer. Each layer can have its own activation function.
- The choice of activation function depends on the problem that we intend to solve.
- Usually, all hidden layers will have the same activation function and the most commonly used functions are sigmoid and tanh functions.
- However, as we have seen, they have their shortcomings and

we have seen others that are capable of replacing them.

- The output layer will also have its own activation function based on the final output that is needed to solve the problem at hand.

1 Activation Functions

Choosing an activation function for the hidden layer mainly depends on what we expect the hidden layer to do. There are many different types of neural networks and depending on the purpose we will be able to decide which particular activation function to utilize.



What is a loss function?

- *The function we want to minimize or maximize is called the objective function or criterion. When we are minimizing it, we may also call it the cost function, loss function, or error function.*

- It is a function that tells as a **quantifiable value of how much error does our approximated model possess**.
- Typically, this involves the difference between the ***actual value and approximated(predicted) value.***
- Again based on the problem considered, the loss function also varies.

Types of loss function

Regression Loss Functions
- Mean Squared Error Loss
- Mean Squared Logarithmic Error Loss
- Mean Absolute Error Loss

Binary Classification Loss Functions
- Binary Cross-Entropy
- Hinge Loss
- Squared Hinge Loss

Multi-Class Classification Loss Functions
- Multi-Class Cross-Entropy Loss
- Sparse Multiclass Cross-Entropy Loss

Kullback Leibler Divergence Loss

Cross-entropy loss is often simply referred to as "*cross-entropy*," "*logarithmic loss*," "*logistic loss*," or "*log loss*" for short.

Regression: Predicting a numerical value(Preferred Activation F : Linear or RELU)

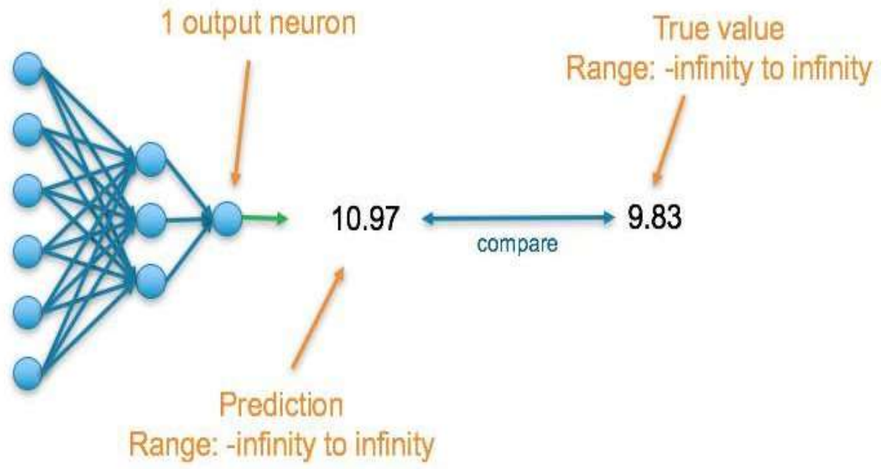
Loss Function

There are three metrics which are generally used for evaluation of Regression problems (like Linear Regression, Decision Tree Regression, Random Forest Regression etc.):

Mean Absolute Error (MAE): This measures the absolute average distance between the real data and the predicted data, but it fails to punish large errors in prediction.

Mean Square Error (MSE): This measures the squared average distance between the real data and the predicted data. Here, larger errors are well noted (better than MAE). But the disadvantage is that it also squares up the units of data as well. So, evaluation with different units is not at all justified.

Root Mean Squared Error (RMSE): This is actually the square root of MSE. Also, this metrics solves the problem of squaring the units.



The final layer of the neural network will have one neuron and the value it returns is a continuous numerical value.

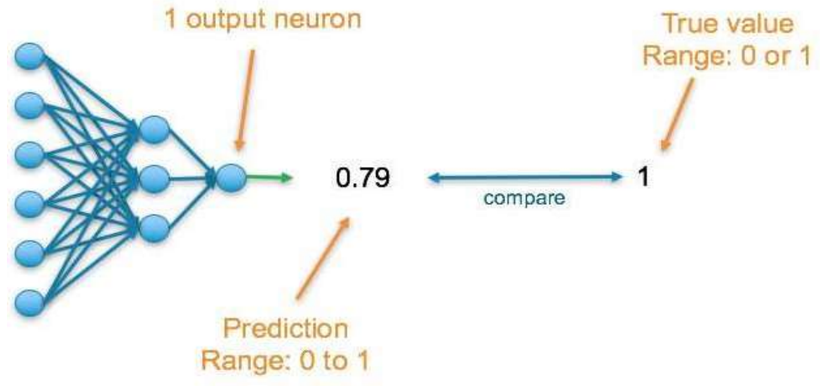
$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Where $\hat{y}$ is the predicted value and y is the true value

Categorical: Predicting a binary outcome (Preferred Activation F : Sigmoid)

Loss Function

Binary Cross Entropy — Cross entropy quantifies the difference between two probability distribution. Our model predicts a model distribution of {p, 1-p} as we have a binary distribution. We use binary cross-entropy to compare this with the true distribution {y, 1-y}



The final layer of the neural network will have one neuron and will return a value between 0 and 1, which can be inferred as a probably.

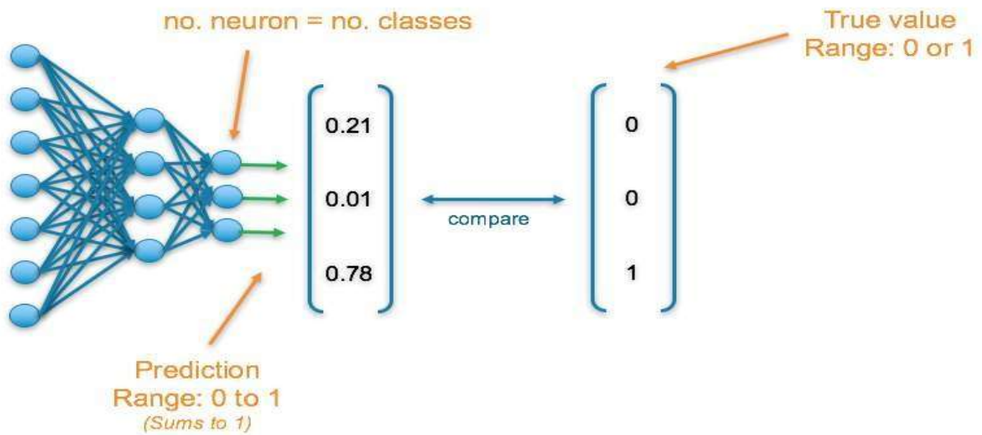
$$\text{Binary cross entropy} = -(y \log(\hat{y}) + (1-y)\log(1-\hat{y}))$$

Where $\hat{y}$ is the predicted value and y is the true value

Categorical: Predicting a single label from multiple classes (Preferred Activation F : Softmax)

Loss Function

Cross Entropy — Cross entropy quantifies the difference between two probability distribution. Our model predicts a model distribution of {p1, p2, p3} (where p1+p2+p3 = 1). We use cross-entropy to compare this with the true distribution {y1, y2, y3}



The final layer of the neural network will have one neuron for each of the classes and they will return a value between 0 and 1, which can be inferred as a probably. The output then results in a probability distribution as it sums to 1.

$$\text{Cross entropy} = -\sum_{i}^{M} y_i \log(\hat{y}_i)$$

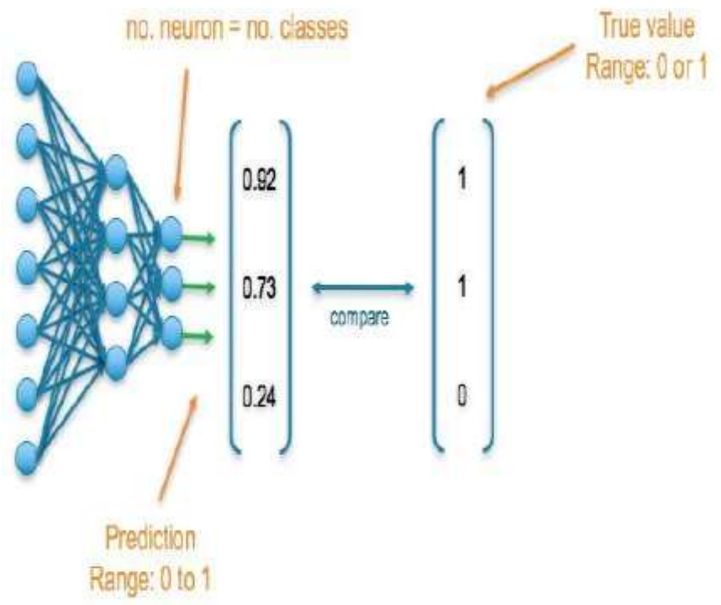
Where $\hat{y}$ is the predicted value, y is the true value and M is the number of classes

Categorical: Predicting multiple labels from multiple classes
(Preferred Activation F : Sigmoid)

Loss Function

E.g. predicting the presence of animals in an image

Binary Cross Entropy — Cross entropy quantifies the difference between two probability distribution.
Our model predicts a model distribution of {p, 1-p} (binary distribution) **for each of the classes.** We use binary cross-entropy to compare these with the true distributions {y, 1-y} for each class and sum up their results



The final layer of the neural network will have one neuron for each of the classes and they will return a value between 0 and 1, which can be inferred as a probably.

$$\text{Binary cross entropy} = -\sum_i^M (y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i))$$

Where $\hat{y}$ is the predicted value and y is the true value

Which Loss Function to Use?

The choice of cost function is tightly coupled with the choice of output unit. Most of the time, we simply use the cross-entropy between the data distribution and the model distribution. The choice of how to represent the output then determines the form of the cross-entropy function.

1.Regression Problem

A problem where you predict a real-value quantity.

Output Layer Configuration: One node with a linear activation unit.

Loss Function: Mean Squared Error (MSE).

2.Binary Classification Problem

A problem where you classify an example as belonging to one of two classes.

The problem is framed as predicting the likelihood of an example belonging to class one, e.g. the class that you assign the integer value 1, whereas the other class is assigned the value 0.

Output Layer Configuration: One node with a sigmoid activation unit.

Loss Function: Cross-Entropy, also referred to as Logarithmic loss.

Which loss function to use?

3. Multi-Class Classification Problem

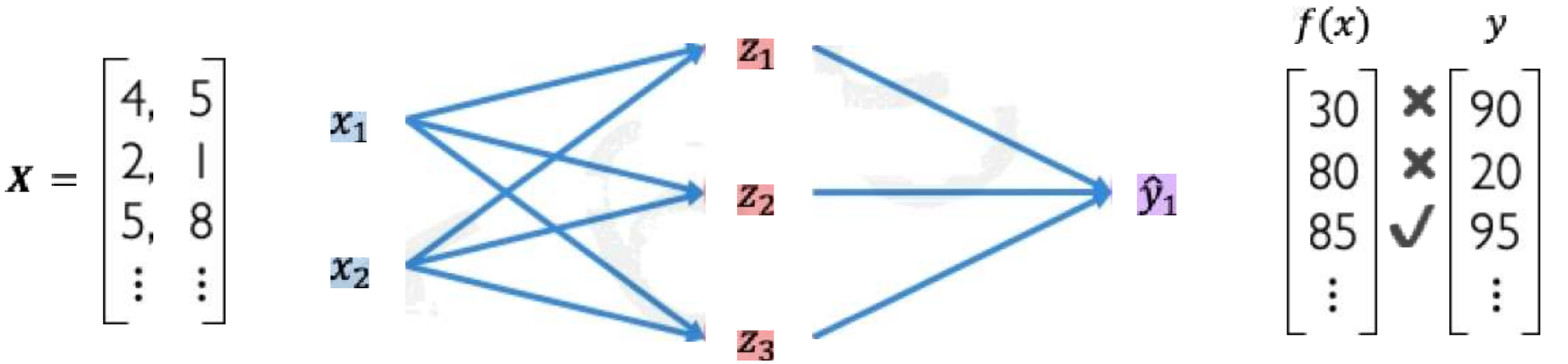
A problem where you classify an example as belonging to one of more than two classes.
The problem is framed as predicting the likelihood of an example belonging to each class.
Output Layer Configuration: One node for each class using the softmax activation function.
Loss Function: Cross-Entropy, also referred to as Logarithmic loss.

Relationship between problems, loss and output activation function

Problem Type	Output Type	Final Activation Function	Loss Function
Regression	Numerical value	Linear	Mean Squared Error (MSE)
Classification	Binary outcome	Sigmoid	Binary Cross Entropy
Classification	Single label, multiple classes	Softmax	Cross Entropy
Classification	Multiple labels, multiple classes	Sigmoid	Binary Cross Entropy

Mean Squared Error Loss

Mean squared error loss can be used with regression models that output continuous real numbers



$$J(W) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - f\left(x^{(i)}\ W\right)\right)^2$$

Actual Predicted

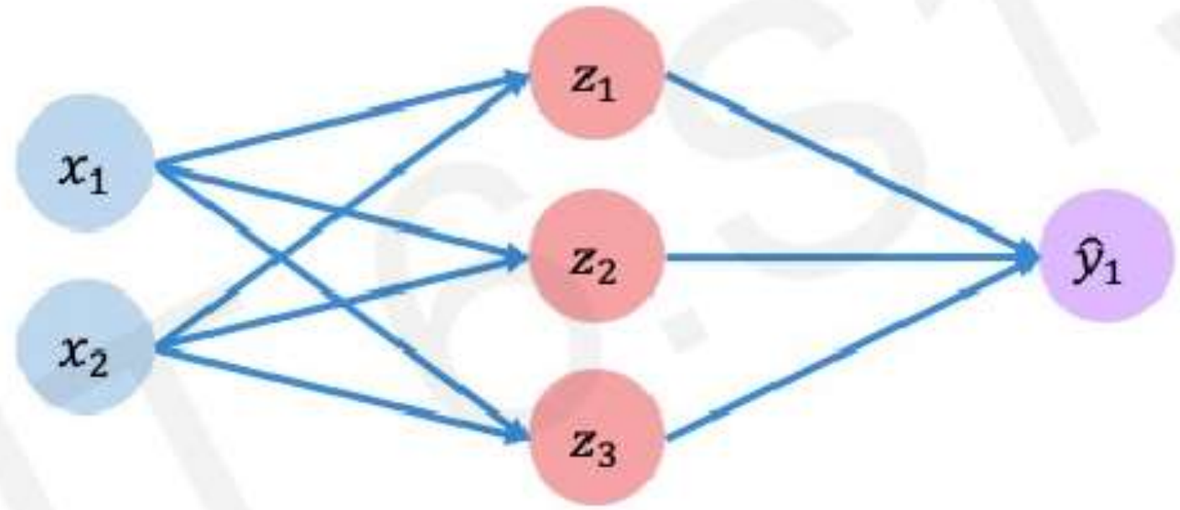
```
loss = tf.reduce_mean( tf.square(tf.subtract(y, predicted)) )
loss = tf.keras.losses.MSE( y, predicted )
```

```
loss = torch.nn.functional.mse_loss(
    predicted, y )
```

Binary Cross Entropy Loss

Cross entropy loss *can be used with models that output a probability between 0 and 1*

$$\boldsymbol{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



$f(x)$ y

$$\begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

$$\boldsymbol{J}(\boldsymbol{W}) = -\frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log \underbrace{\left(f\left(x^{(i)}; \boldsymbol{W}\right)\right)}_{\text{Predicted}} + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log \underbrace{\left(1 - f\left(x^{(i)}; \boldsymbol{W}\right)\right)}_{\text{Predicted}}$$

```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(y, predicted) )
loss = torch.nn.functional.cross_entropy( predicted, y )
```

Loss Functions for Regression

- Mean Squared Error (MSE):

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

where y_i is the actual value, $\hat{y}_i$ is the predicted value, and n is the number of observations.

- Mean Absolute Error (MAE):

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

Loss functions for classification

- Binary Cross-Entropy:

$$L(y, \hat{y}) = -\frac{1}{n} \sum_{i=1}^{n} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

- Categorical Cross-Entropy:

$$L(y, \hat{y}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

where C is the number of classes.