Lecture # 4 Content: ANN Modeling

1- Introduction

2- ANN vs Biological Neurons

- 2.1- Nodes vs Neuron
- 2.2- Perceptron and how perceptron works
- 2.3- Activation functions

3- ANN Computation

- 3.1 Forward pass
- 3.2 Backward propagation
  - 3.2.1 Gradient descent algorithm (How it works?)
  - 3.2.2 Optimizers (Types and how they work?)

4 ANN Modeling with Keras.

- 4.1 Data preprocessing
- 4.2 ANN modeling
  - 4.2.1 splitting scaled data into traing/test.
  - 4.2.2 Creating ANN / Fully connected
    - INPUT LAYER
    - HIDDEN LAYER
    - OUTPUT LAYER
  - 4.2.3 Compile the model → model.fit()
    → perform optimisation
  - 4.2.4 Fitting ANN model with trainingdata
- 4.3 Model prediction
  - 4.3.1 Inverse transform of scaled data
  - 4.3.2 predict with test inputdata
- 4.4 Model accuracy analysis
  - 4.4.1 Compute R^2 (y_{pred}, y_{test})
  - 4.4.2 Compute MSE (y_{pred}, y_{test})

5. Summary

Lecture#4 How Artificial Neural Network (ANN) works?

4.0 Introduction

In the previous chapter, we learned how the Linear regression (simple linear, Polynomial, Multivariable) and the Non-Linear regression (Exponential, Logarithmic, Power, ..) work. For these type of regression we need to first display data to estimate the best mapping function

On the other hand, when the dataset, for instances, behaves as the following, we can not map the dataset by the above regression functions

- For datasets such as these and even for linear / non linear trend dataset, ANN based modeling perform the job.

In this chapter, we will learn the basic concept of how ANN modeling works.
The issues to be discussed are:

⇒ ANN - building block / perceptron
⇒ How - Perceptron perform computation
⇒ How ANN Perform computation?
  - Forwardfeed / loss computation
  - Backward propagation / optimization
⇒ Finally, do practical ANN modeling with (a) Synthetic data
  (b) Field data

Lecture How ANN works?

Artificial neural network (ANN)

(2.0) ⇨ • ANN vs Biological neuron network

ANN is a mathematical model of a system that simulates similar to a biological neural network in human brain capable of learning, prediction, and recognition

2.1 ⇨ • **Nodes vs Neurons**

- ANN uses Nodes, which are similar to Neurons (cellular building block of brain)

- Cellular Neurons are interconnected, so does ANN nodes

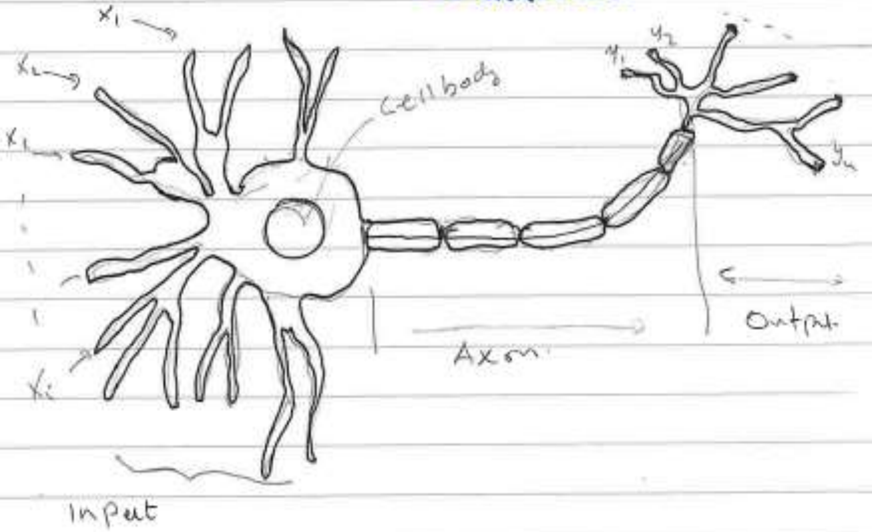
⇨ Description of biological neurons:

Neurons includes (Fig below)

→ Cell

→ Axon

→ Dendrites



In 1943, Warren Mc Culloch and Walter Pitts created the first Mathematical model of a neural network to model/describe how brain works.

→ The model is a simple linear model that results in a positive or negative output given a set of inputs and weights.

The function reads

$$\boxed{f(x, w) = x_1 w_1 + w_2 x_2 + \cdots x_n w_n}$$

where, $f(x, w)$ = output
- x_i = input
- w_i = weight

→ This model of computation was called Neuron since it tried to simulate how the building block of the brain work

→ Their model doesn't learn to find weights

2.2 Perceptron

Being inspired from the biological neuron and its ability to learn, In 1957, Frank Rosenblatt ~~was~~ ~~the~~ introduced the concept of PERCEPTRON. He developed an algorithm that could learn the weight to produce output

- Perceptron is the basic building block of a neural network

→ It is simple model of Neurons

→ The perceptron model is used for for binary classifers.

→ **Perceptron** consists of four parts.

- → Input values
- → Weignts and a bias
- → Weighted Sum
- → Activation function

⇒ How perceptron works?

→ Perceptron works by receiving numerical input values along with weights and bias

input (x_1, x_2, x_3), weight (w_1, w_2, w_3), b_i

→ It then dot product the inputs and with the respective weigths

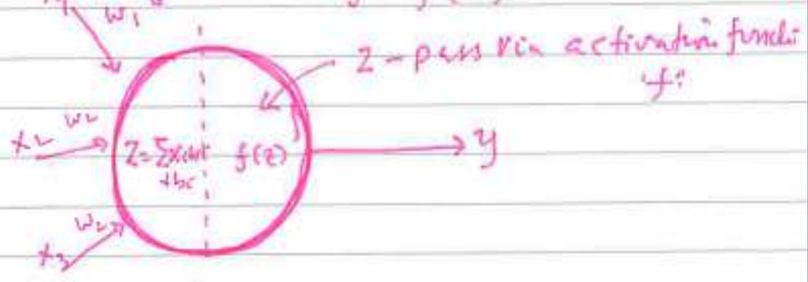
$x_i w_i$

→ The individual weignted inputs are add sumed and then added together along with the bias

$$Z = \sum_{i=1} x_i w_i + b_i$$

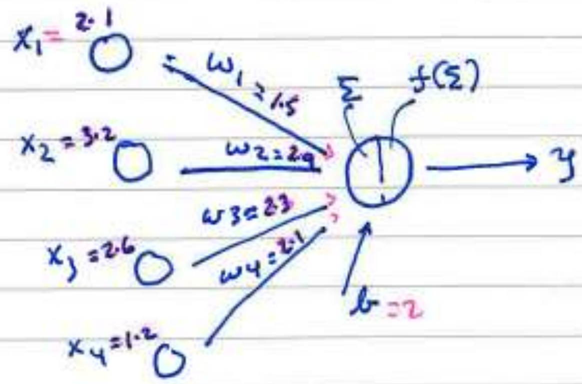
→ The sum result will then pass througn the activation function and return a final out put

$$y = f(z)$$



Example 1 How a single perceptron computation performed in python?

Example 1: By implimenting Perceptron Model



input = [2.1, 3.2, 2.6, 1.2]

Weight = [1.5, 2.9, 2.3, 2.1]

bias = 2

Step 1: Compute sum.

$$Z = \sum x_i w_i + b_i$$

= input[0] * weight[0] + input[1] * weight[1] + input[2] * weight[2] + input[3] * weight[3] + bias

Step 2: Pass the computed sum through activation function

Define activation functions

(1) Sigmoid function

```
def sigmoid (x):
        return 1/(1 + np.exp(-x))

print ('Sigmoid:', sigmoid(z))
```

2.3 ACTIVATION FUNCTIONS

What are ACTIVATIN Functions?

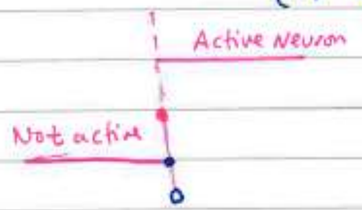
> The activation function / transfer function CONVERT the input Signal to the output signal.

> There are various kind of activation function. These are

(1) Binary step function
(2) Linear function
(3) Non-Linear function

> Description of activation functions

(1) Binary step function



⇒ The function decides whethere a neuron shoucd be ACTIVATED or NOT based on the threshold value

Binary step

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

→ used for binary classifer.

Issue with binary step function:

- It is used for binary target value. If the targets are more than two, it is not useful
- Derivative of step function = 0. This does not allow BACK PROPAGATION

(2) Relu activation function

```
def Relu(x):
    return max(0,x)

Print('Relu:', Relu(z))
```

(3) LeakyRelu activation function

```
def LeakyRelu(x):
    if x>0:
        return x
    else:
        return 0.01*x
Print('LeakyRelu:' LeakyRelu(z))
```

(4) Tanh activation function

```
def Tanh(x)
    return (np.exp(x) - np.exp(-x))/
           (np.exp(x) + np.exp(-x))

Print('Tanh:' Tanh(z))
```

(5) Swish activation function

```
def swish(x)
    beta=1.0
    return x/(1+np.exp(-beta*x))

print('Swish:' Swish(z))
```

Example 2 : By using $\boxed{\text{numpy}}$ to compute perceptron model [perceptron computation)

Import library

```
import numpy as np
```

Input / Weight List + bias

```
input = [2.1, 3.2, 2.6, 1.2]
weight  [1.5, 2.9, 2.3, 2.1]   ← Single neuron
bias = 2
```

dot product $(x_i \cdot w_i)$ + bias

```
Z = np.dot(input, weight) +bias
print(z)
```

Pass the sum via activation function

Here define functions

(1) Sigmoid function

```
def sigmoid(x):

    return 1/(1+ np.exp(-x))

print('Sigmoid:', sigmoid(z))
```

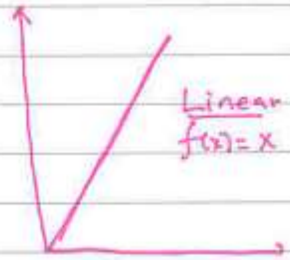
(2) Relu function

```
def Relu(x):

    return max(0,x)

print('Relu:', Relu(z))
```

(3) Leaky Relu function

```
def LeakyRelu(x):
    if x >0
        return x
    else:
        return 0.01x
print('LeakyRelu:', LeakyRelu(z))
```

(2) Linear Activation function



→ The activation function is proportional to the input
$f(x) = x$

→ properties of the activation fun

⇒ The gradient is non-zero, constant value

→ It allow backpropagation. But the updating factor is the same. Therfore, the network improve the Error since the gradient is the same for every iteration

(3) Non-linear activation function

→ The Linear activation does not allow the model to create complex mapping between the networks input/output

→ Therefore, non-linear activation functions Solve the limitations of linear activation functions such as:

- The existance of gradient that allow backpropagation to update weight/bias.

(a) Sigmoid activation function

→ Sigmoid is an S-shaped non linear function

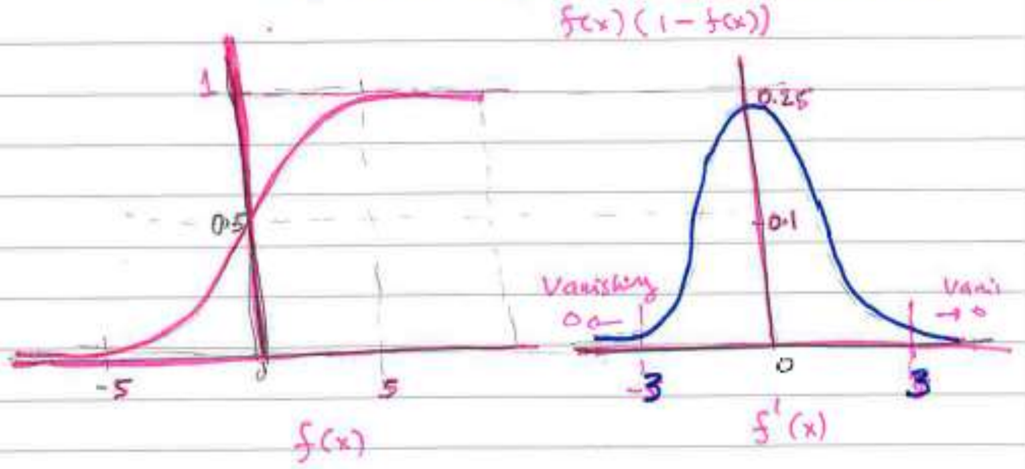
> Sigmoid function, f(x)

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

> Derivative function $\frac{df}{dx}$

$$f'(x) = \text{Sigmoid}(x)\,(1 - \text{Sigmoid}(x))$$

$$f(x)(1-f(x))$$



→ For any input real value for the function, the output value will be in the range of 0 to 1

→ The decision for the out come 0/1 is based on the threshold value 0.5

→ The gradient values are significant for the range -3 to 3

⇒ the values outside [-3 3], the function will have small gradient

⇒ Network stops LEARNING as the gradient approach zero

→ Sigmoid will return to zero/closer when the value < -5 and closer to 1 when the value > 5.

(b) Tanh Function (Hyperbolic Tangent)

→ The hyperbolic tangent is S-shaped like Sigmoid activation func

→ The difference is that the output is range of -1 to 1

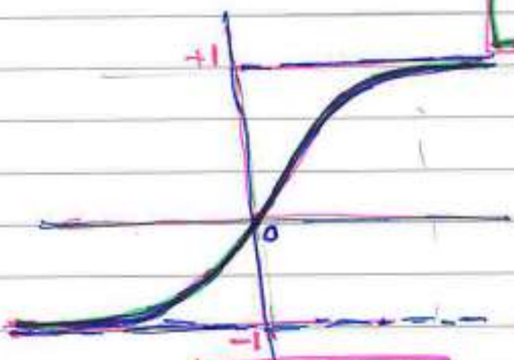
$$\boxed{f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}}$$

→ at center, the output of the tanh function is zero

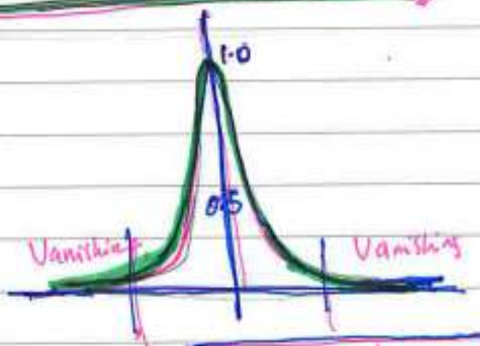
→ It is used as hidden layer in Neural network

Gradient function: $\frac{df}{dx}$

$$\boxed{f'(x) = 1 - \tanh(2x)}$$



$\boxed{f(x) = \tanh(x)}$



$\boxed{f'(x) = 1 - \tanh(2x)}$

→

→ The gradient is steeper as compared to the Sigmoid function

→ like sigmoid, tanh also has the issue of Vanishing gradient

→ Tanh is preferred to Sigmoid
gradient

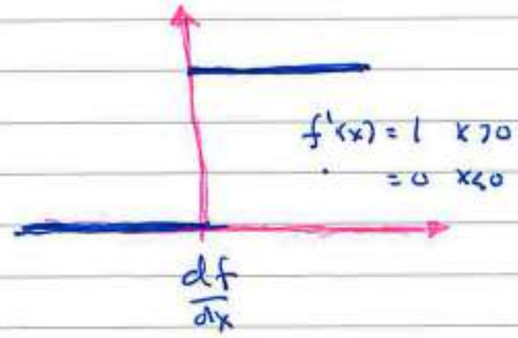
(c) ReLu Function (Rectified Linear Unit)

ReLu has a linear function

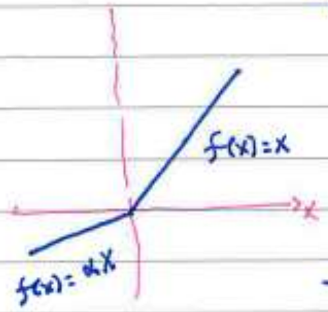
$$f(x) = \max(0, x)$$

Gradient of ReLu function

– has zero at negative side. During backpropagation, it can create dead neurons which never get activated.



(d) LeakyReLu function (LeakyReLu)



→ LeakyRelu is a modified version of Relu. The difference is that it has a small slope for the negative ~~side~~ values instead of a flat (0) slope

→ The slope is determined before training. It means that slope is not learnt during training.

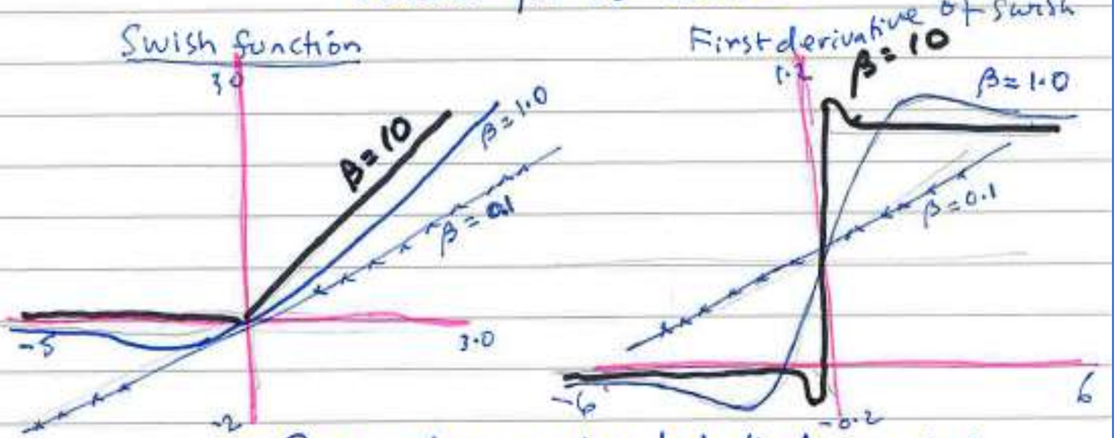
→

(e) Swish Function

Swish is a modified Sigmoid function

$$f(x) = x \cdot \text{Sigmoid}(\beta x)$$

Where β = is constant



- Researches indicated that using Swish function improves performance Compared to Relu and Sigmoid

- Reason:

  The reason for the improvement is that the Swish function helps reduce the vanishing gradient problem during Backpropagation.

```
def swish(x):
    beta = 1.0
    return x/(1+np.exp(-beta*x))
output = 2
print(swish(output))
```

(f) Softmax function

- The Softmax function is the Combination of MULTIPLE SIGMOIDS

  - Sigmoid is used for BINARY Classification

  - Softmax is used for multiclass Classification

Softmax function

$$f(z_i) = \frac{\exp(z_i)}{\sum_j^n \exp(z_j)}$$

input

n Elements

⇒ Sigmoid function returns values between 0 and 1

⇒ Softmax function gives the probabilities of the data points belonging to the particular class.

Softmax is used for multiclass problem.

Eg. For the three classes, three neurons in the output layer are required

Example. For the output ~~layer~~ neurons as [1.2, 0.9, 0.75] applying softmax weget

List

List

$$\begin{bmatrix} 1.2 \\ 0.9 \\ 0.75 \end{bmatrix} \rightarrow \frac{\exp(z_i)}{\sum_j^n \exp(z_j)} \rightarrow \begin{bmatrix} 0.42 \\ 0.32 \\ 0.25 \end{bmatrix}$$

$n=3$ $j=1 \dots n$

$$f(1.2) = \frac{1.2}{1.2+0.9+0.75} = 0.42$$

$$f(0.9) = \frac{0.9}{1.2+0.9+0.75} = 0.32$$

$$f(0.75) = \frac{0.75}{1.2+0.9+0.75} = 0.26$$

```
def softmax(x):
    return np.exp(x)/np.sum(np.exp(x))
output = [1.2, 0.9, 0.75]
print(softmax(output))
```

The sum of all values = 1

Softmax is often used as the
≫ Last activation function of neural network
to normalize the output of neuralnetwork to a probability distribution over predicted output

Choosing the Right Activation Functions

For hidden layer

- As a rule of thumb, begin with Relu, then try with other provided that Relu doesn't give optimum result.

→ Swish function is belived to reduce the vanishing gradient problem during back propagation

→ Tanh and Sigmoid has vanishing gradient issue.

→ For output layer →

- Regression → use linear activation func

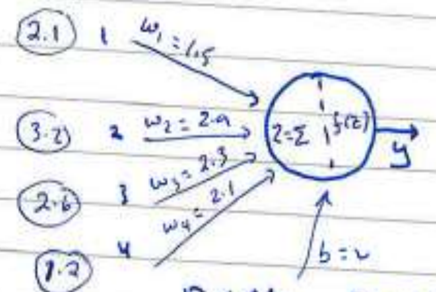
→ Binary classification → use Sigmoid/activation func

→ Multiclass classification - use - softmax

Lab# How Perceptron Computation perform
(use 6 activation functions)

Ex1a Single perceptron computation
→ Here, by implimenting the perceptron model.

Input

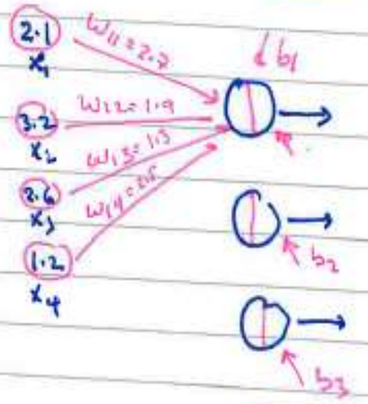


Input = [2.1, 3.2, 2.6, 1.2]
weight = [1.5, 2.9, 2.3, 2.1]
bias = 2

Ex1b Single perceptron computation
Here we use numpy to perform the Computation
Input = [2.1, 3.2, 2.6, 1.2]
weight = [1.5 2.9 2.3 2.1]
bias = 2

Compare the results of Ex 1a and 1b. which one is easy for the computation?

Ex.2a Multiperceptron. Computation in hidden layer
multilayer perceptron
→ use: implimenting the perceptron model.



Inputs = [2.1, 3.2, 2.6, 1.2]
weight 1 = [2.7, 1.9, 1.3, 2.5]
weight 2 = [2.5, 3.9, 2.9, 2.5]
weight 3 = [2.5, 3.9, 4.3, 5.1]
bias 1 = 1, bias 2 = 2 bias 3 = 3

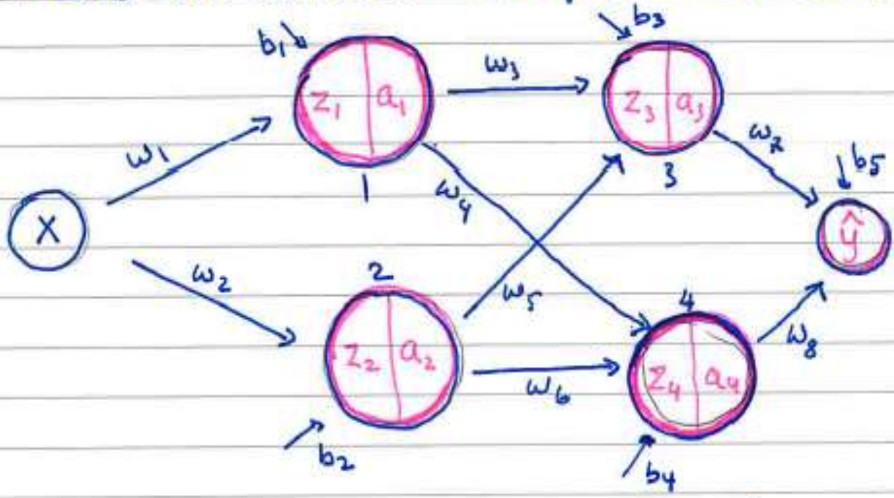
Ex.2b: Using the inputs / weights, / biase, of Ex1a, applying numpy, perform perceptron Computation

Finally → Compare the results of Ex 2a and 2b. which one is easy to perform the computation?

3.0 ANN - COMPUTATION

ANN → How Forward-and Backpropagation Performed to TRAIN the dataset?

→ Let us consider a simple neural network



→ a_1, a_2, a_3, a_4 are activation function

→ for hidden layer, as a rule of thumb, we use Relu activation function

→ Z_1, Z_2, Z_3 and Z_4 are linear sum of $\sum w_i x_i + b$

→ The input data (x) pass through node 1 and node 2

→ The output from nodes 1 and 2 will feed nodes 3 and 4 respectively.

→ Finally, nodes 3 and 4 feed the output node.

→ The weights for each node labeled as w_1 - to w_8 and bias $b_1 - b_5$

3.1 FORWARD Propagation / Pass.

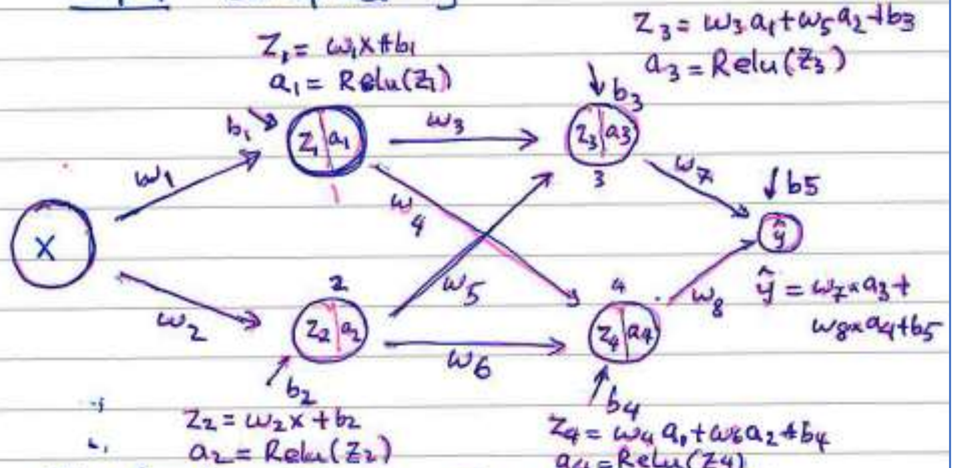
→ The two steps during forward pass are (a) Computation of the output $\hat{y}$

(b) Then, Compute the loss (MSE)

How to compute the output $\hat{y}$ and loss?

For the better illustration, the following figure shows the details of forward pass Calculation to get $\hat{y}$ and loss?

Step 1 compute $\hat{y}$



The functions Z_1 Z_2, Z_3 and Z_4 are obtained through matrix multiplication

$$\begin{bmatrix} Z_1 \\ Z_2 \end{bmatrix} = \begin{bmatrix} w_1 & b_1 \\ w_2 & b_2 \end{bmatrix} \begin{bmatrix} x \\ 1 \end{bmatrix} = \begin{bmatrix} w_1x+b_1 \\ w_2x+b_2 \end{bmatrix}$$

$$\begin{bmatrix} Z_3 \\ Z_4 \end{bmatrix} = \begin{bmatrix} w_3 & w_5 & b_3 \\ w_4 & w_6 & b_4 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} Z_3 \\ Z_4 \end{bmatrix} = \begin{bmatrix} w_3a_1 + w_5a_2 + b_3 \\ w_4a_1 + w_6a_2 + b_4 \end{bmatrix}$$

$$\hat{y} = \begin{bmatrix} w_7 & w_8 & b_5 \end{bmatrix} \begin{bmatrix} a_3 \\ a_4 \\ 1 \end{bmatrix}$$

$$\boxed{\hat{y} = w_7a_3 + w_8a_4 + b_5}$$

Step 2 Loss.

3.2 Backpropagation

→ The unkown parameters of the neural network are their weights and biases

⇒ The right values of weights and biases need to be determine, so that they allow the best fit for our datasets.

⇒ Best fit model means that the loss/error between the model and the dataset is minimized.

How to find the optimized weight/bias?

→ we use the gradient decent algorithm to minimize the error between the predicted ($\hat{y}$) and the target (y)

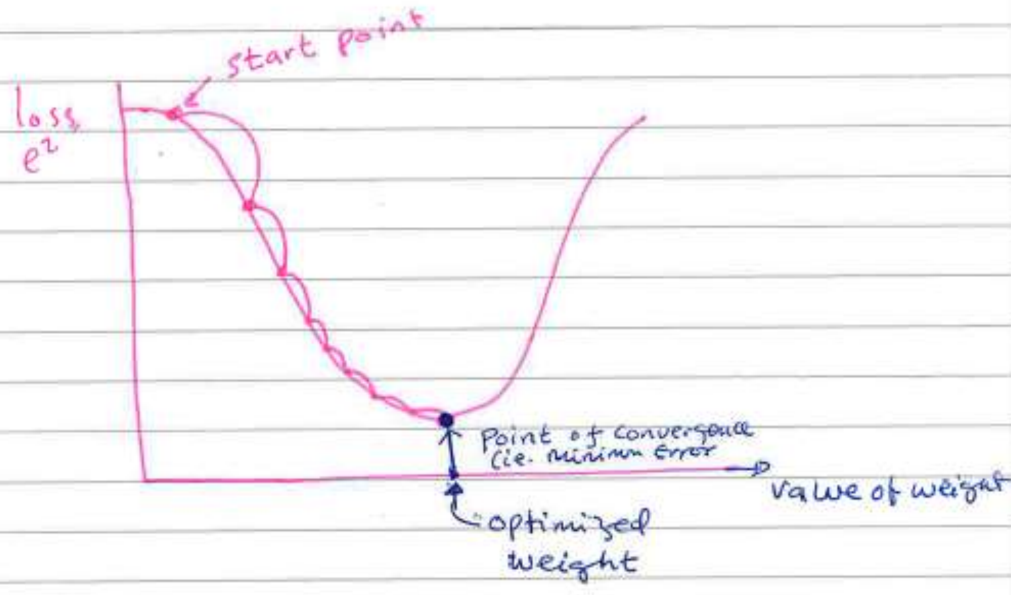
3.2.1 Gradient descent

→ Gradient decent is an optimization algorithm that uses the gradient of the loss function to search for the minimum error

» As the minimization process starts, the neural network uses random weight and biases. It means that we start at a random point on the loss surface.

» To reach the lowest point on the surface, we start taking steps along the direction of the steepest downward slope

→ The name is therefore called GRADIENT DECENT



→ The gradient decent algorithm can achieve finding the point of minimum during each training epoch or iteration process.

→ For n^{th} iteration (epoch), the gradient decent back method will update the weight and biases by the following method

$$\boxed{W_i^{n+1} = W_i^{n} - \beta \nabla_w L \quad i = 1 \cdots m}$$

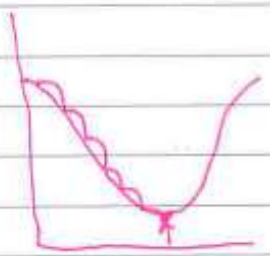
where

W_i^{n+1} = Newly updated weight/bias

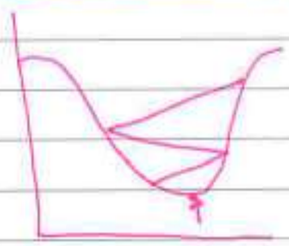
m = total # of weight/bias in the network

β = Learning rate that determine the size of each steps

Effect of Learning rate



Small Size Learning rate



Large Learning rate

→ Small size – gives good precision, but ~~takes time.~~ takes time

→ Large size → risk of overshooting the minimum

The partial derivative of the loss function with respect to each weight and bias is calculated during the backpropagation process as

$$\nabla_w L = \begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \vdots \\ \frac{\partial L}{\partial w_m} \end{bmatrix}$$

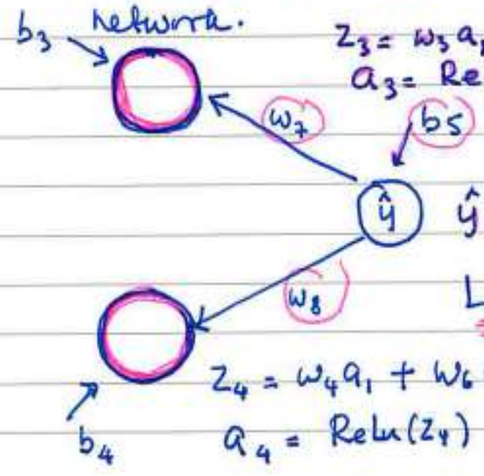
where L: loss $= (\hat{y} - y_i)^2$

> Backward propagation computation process starts at the output node, it systematically progress through the layers until reaching the input layer

> Therefore the name backward propagation is derived from the computation process.

> During the computation process, the chain rule is applied for computing the derivations at each step.

$\boxed{\text{Example\#1}}$ Let us now compute the partial derivation for the considered neural network.



$z_3 = w_3 a_1 + w_5 a_2 + b_3$

$a_3 = \text{Relu}(z_3)$

$\hat{y} = w_7 a_3 + w_8 a_4 + b_5$

$L = (y - \hat{y})^2$

$z_4 = w_4 a_1 + w_6 a_2 + b_4$

$a_4 = \text{Relu}(z_4)$

For the above nodes, we will apply the Partial derivative with respect to w_7, w_8 and b_5

⇒ We first start with the partial derivative of the Loss, L, w.r.t the output $\hat{y}$

$$\boxed{L_y = \frac{\partial L}{\partial \hat{y}} = -2(y-\hat{y})}$$

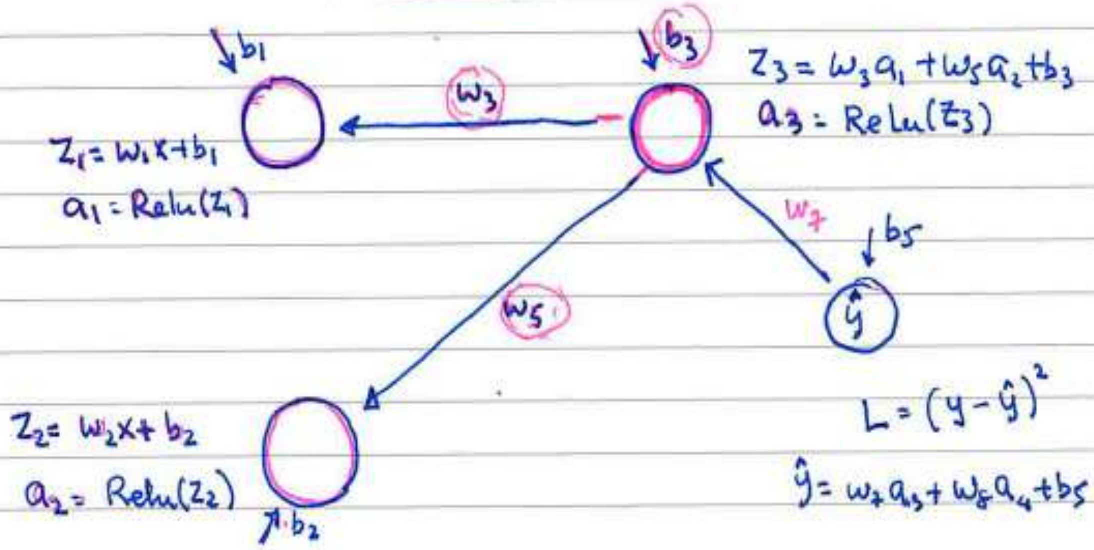
⇒ Applying the partial derivation of the loss wrt the weight, w_7

$$\boxed{\frac{\partial L}{\partial w_7} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_7} = L_y \cdot a_3}$$

⇒ Apply Partial derivation w.r.t w_8, and b_5

$$\boxed{\frac{\partial L}{\partial w_8} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_8} = L_y \cdot a_4}$$

$$\boxed{\frac{\partial L}{\partial b_5} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b_5} = L_y}$$



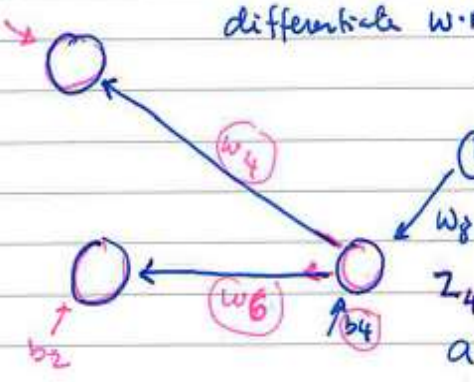
For the above nodes, we will apply Partial derivation wrt w_3, w_5 and b_3

$$\boxed{\begin{aligned}\frac{\partial L}{\partial w_3} &= \frac{\partial L}{\partial a_3}\cdot\frac{\partial a_3}{\partial z_3}\cdot\frac{\partial z_3}{\partial w_3} = \frac{\partial L}{\partial \hat{y}}\cdot\frac{\partial \hat{y}}{\partial a_3}\cdot\frac{\partial a_3}{\partial z_3}\cdot\frac{\partial z_3}{\partial w_3}\\ &= L_y \cdot W_7 \cdot dRL(z_3)\cdot a_1\\ &\text{where } dRL(z_3) = \frac{d\,ReLu(z_3)}{\partial z_3}\end{aligned}}$$

$$\boxed{\begin{aligned}\frac{\partial L}{\partial w_5} &= \frac{\partial L}{\partial a_3}\cdot\frac{\partial a_3}{\partial z_3}\cdot\frac{\partial z_3}{\partial w_5} = \frac{\partial L}{\partial \hat{y}}\cdot\frac{\partial \hat{y}}{\partial a_3}\cdot\frac{\partial a_3}{\partial z_3}\cdot\frac{\partial z_3}{\partial w_5}\\ &= L_y \cdot W_7 \cdot dRL(z_3)\cdot a_2\\ &\text{where } dRL(z_3) = \frac{d\,Relu(z_3)}{dz_3}\end{aligned}}$$

$$\boxed{\begin{aligned}\frac{\partial L}{\partial b_3} &= \frac{\partial L}{\partial a_3}\cdot\frac{\partial a_3}{\partial z_3}\cdot\frac{\partial z_3}{\partial b_3} = \frac{\partial L}{\partial \hat{y}}\cdot\frac{\partial \hat{y}}{\partial a_3}\cdot\frac{\partial a_3}{\partial z_3}\cdot\frac{\partial z_3}{\partial b_3}\\ &= L_y \cdot W_7\, dRL(z_3)\cdot\\ &\text{where } dRL(z_3) = \frac{d\,Relu(z_3)}{dz_3}\end{aligned}}$$

And we continue for the following,
differentiate w.r.t w_6, w_4, and b_4



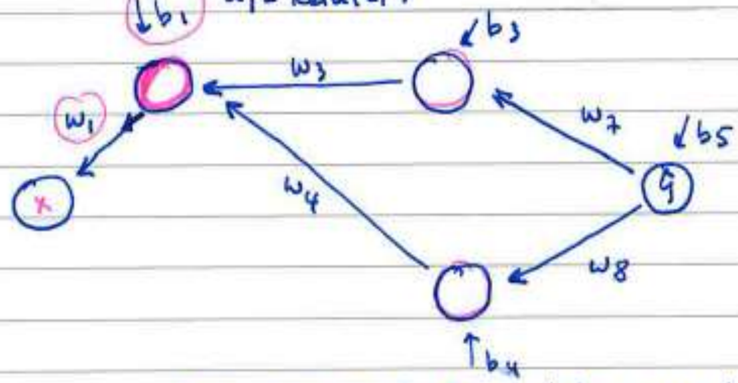
$\hat{y} = w_7 a_3 + w_8 a_4 + b_8$

$z_4 = w_4 a_1 + w_6 a_2 + b_4$

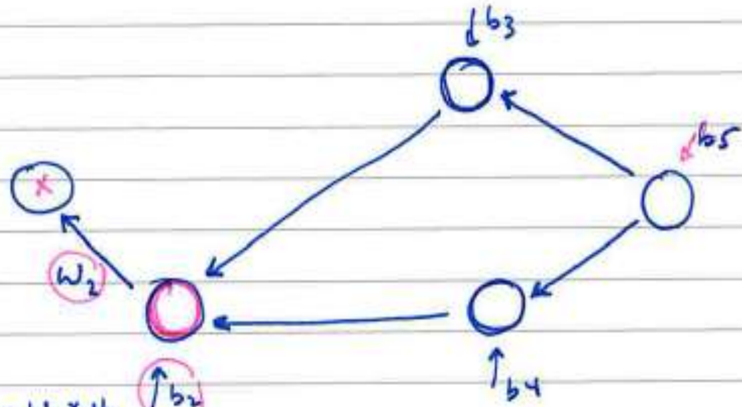
$a_4 = ReLu(z_4)$

$Z_1 = w_1 x + b_1$

$a_1 = \text{Relu}(Z_1)$



Apply partial derivation w.r.t $\underline{w_1, b_1}$



$Z_2 = W_2 x + b_2$

$a_2 = \text{Relu}(Z_2)$ Here we apply partial derivation w.r.t $\underline{W_2, b_2}$

Summary

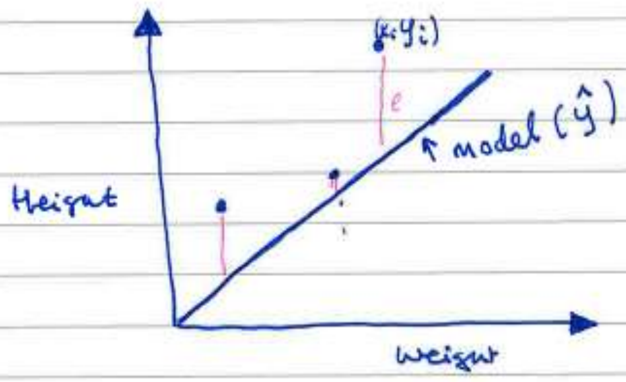
- The above example illustrates how the backward propagation computation process performed throgh the Layers Starting from the output untill reaching the input
- Remember that during the ANN modeling You don't need to impliment the mathematics from the scratch. The Python library will perform the computation.
- Example #2, In the next Page, present numerical example to understand how gradient decent works

Example #2. Gradient Decent → Numerical example

Step-by-step ~~how~~ illustration of how gradient decent algorithm works?

→ Let us consider the height and weight dataset.

→ **Task**: For simplicity, let us just optimize intercept for the known slope = 0.63.



Data #	Height	Weight
1	1.4	0.5
2	1.9	2.3
3	3.2	2.9

The desire is to generate a model / regression line that relates height based on weight as

$$\boxed{\text{Predicted height} = \text{Slope} * \text{Weight} + \text{Intercept}}$$

We assumed Slope to be = 0.63

Task! Find "Intercept".

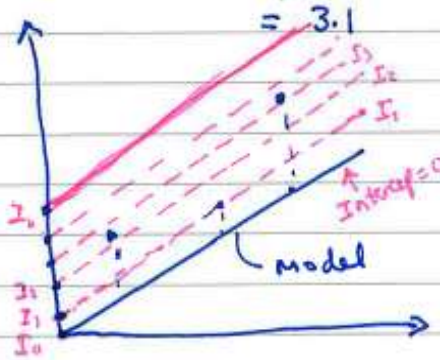
Step1: Calculate residual Sum

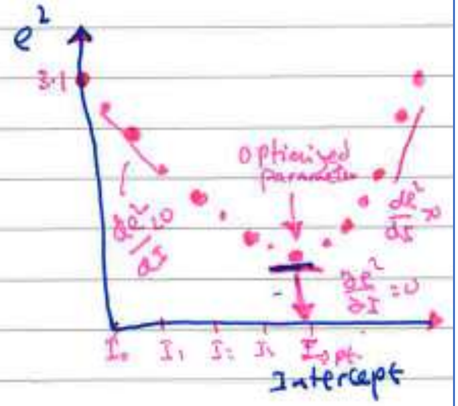
→ For a given Predicted model ($\hat{y}$) and data, we will compute the residual sum.

Example → when the intercept = 0, the residual sum square is

$$\Rightarrow e^2 = \left(1.4 - (0 + 0.64 * 0.5)\right)^2 + \left(1.9 - (0 + 0.64 \times 2.3)\right)^2 + \left(3.2 - (0 + 0.64 \times 2.3)\right)^2$$

$= 3.1$





> Increasing the ~~Ste~~ Intercept, and Calulating residual sum square, then ploting, we can see the reduction of e^2 and then increasing e^2.

>> The optimized Intercept value associated with minimum residual gives the best-fit model

⇒ The optimized Parameter is obtained at turning point ⇒ $\boxed{\frac{\partial e^2}{\partial I} = 0}$

Step 2

The best optimized Parameters are Obtaind by appling Least Square Error method: ie

$$\boxed{\frac{de^2}{d\,\text{Intercep}} = 0}$$

Similary for ~~weight~~ Slope

$$\boxed{\frac{\partial e^2}{\partial\,\text{Slope}} = 0}$$

Since we allready assumed the Slope to be 0.64, we will only apply Least square error for the intercept.

$$\frac{de^2}{d\,\text{Intercept}} = -2\left(1.4-(\text{Intercept}+0.64*0.5)\right) + -2\left(1.9-(\text{Intercept}+0.64*2.3)\right) + -2\left(3.2-(\text{Intercept}+0.64*2.9)\right)$$

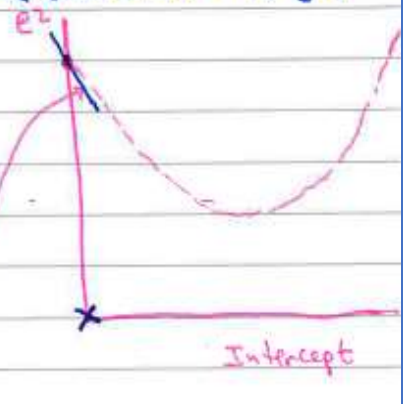
Step 3

→ Let us start by setting the Intercept to a random number. In this Case, Intercept = 0

→ So, we plug Intercept=0 in the derivative to get the Slope = -5.7

$$\frac{de^2}{d\,\text{Intercept}} = -2\left(1.4-(0+0.64*0.5)\right) + -2\left(1.9-(0+0.64*2.3)\right) + -2\left(3.2-(0+0.64*2.9)\right)$$

$$= -5.7$$



Step 4. Determine Step Size

Gradient decent determines the Step size by multiplying the slope by a small number called Learningrate (Eg, 0.1, 0.01)

→ when the intercept is zero, the stepsize

⇒ Stepsize = LearningRate × Slope

= 0.1 × (−5.7)

= −0.57

Step 5 Compute / update new Intercept

$\boxed{\text{New Intercept} = \text{Old intercept} - \text{Stepsize}}$

For the given Step-size, the new intercept

~~New Interc~~

$\boxed{\text{New Intercept} : \text{Old intercept} - \text{Learningrate} \times \text{Slope}}$

New Intercept : $0 - (-0.57) = 0.57$

(a) Now take another Step, we go back to the derivative and plug the new Intercept (0.57)

$$\frac{de^2}{d\,\text{Intercept}} = -2(1.4 - (0.57 + 0.64 \times 0.5)) + -2(1.9 - (0.57 + 0.64 \times 2.3)) + -2(3.2 - (0.57 + 0.64 \times 2.9))$$

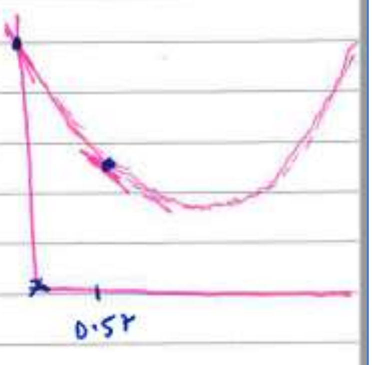
$$= -2.3$$

Step size = $0.1 \times (-2.3)$

$= -0.23$

New Intercept = $0.57 - (-0.23)$

$= \underline{0.8}$

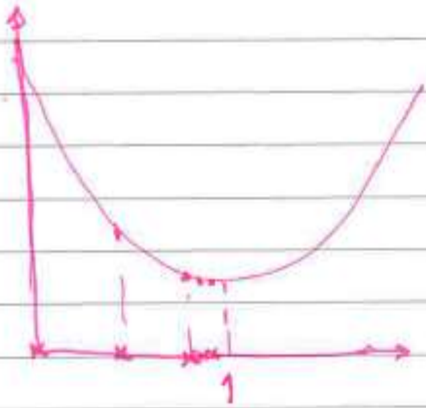


(5) Repeat for Intercept: 0.8

$$\frac{de^2}{d\,\text{Intercept}} = -2\big(1.4 - (0.8 + 0.64 \times 0.5)\big) + -2\big(1.9 - (0.8 + 0.64 \times 2.3)\big) + -2\big(3.2 - (0.8 + 0.64 \times 2.9)\big) = \underline{\underline{-0.9}}$$

Step Size = Learning Rate * Slope
$= 0.1 \times (-0.9)$
$= \underline{-0.09}$

New Intercept = $0.8 - (-0.09) = \underline{\underline{0.89}}$



→ Again, we take another step, and the new intercept = 0.92

→ And then, we take another step, the new intercept = 0.94

→ we take another step, the new intercept = $\underline{\underline{0.95}}$

→ After 6 steps, the gradient decent estimates for the intercept = $\underline{\underline{0.95}}$

The Least Square estimate for the intercept is also = $\underline{\underline{0.95}}$

The Question is ~~that~~ how does gradient know stop taking steps?

⇒ when the step sizes become closer to zero, the gradient decent process stops.

⇒ Step size close to ZERO occurs when the slope $\left(\frac{de^2}{\partial Int}\right)$ close to ZERO

⇒ The minimum step size is 0.001 or smaller is the common practice.

⇒ If the slope is 0.009 and If we plug the learning rate = 0.1, we get the step size = 0.0009, which is smaller than 0.001, so gradient decent would stop.

⇒ Gradient decent also includes a limit on the number of steps it will take before giving up. In practice the maximum number of steps = 1000 or greater

⇒ On the other hand, if we define the maximum number of steps, the gradient decent will stop even if the step size is large.

Gradient decent algorithm

We understand how gradient decent can estimate the intercept.

To estimate both slope and intercept.

Step 1: Sum square Error, $L = e^2$

Step 2: take derivation of the loss function w.r.t slope and intercept (Eq *, Eq **)

Step 3 plug random initial value for slope and intercept in the gradient

Step 4: Calculate Step Size = ~~Learning rate x gradient~~
= Learning rate * gradient

Step 5: Calculate new slope and intercept

New slope = old slope − Learning rate * $\frac{\partial e^2}{\partial slope}$

New Intercept = old intercept − Learning rate $\frac{\partial e^2}{\partial intercept}$

go to step 3:

→ Add / Plug the new slope and new intercept in the gradient.

→ and step 4. Calculate new step size
step 5 − Calculate new slope / new intercept.

− Repeat steps 3 − 5 until the step size is small (0.001 or Reaches to the maximum number of steps

The gradient:

Eq.*

$$\frac{de^2}{d\,slope} = -2 * 0.5\left(1.5 - (\text{Intercept} + \text{Slope} * 0.5)\right) + -2 * 2.9\left(3.2 - (\text{Intercept} + \text{Slope} * 2.9)\right) + -2 * 2.3\left(1.9 - (\text{intercept} + \text{Slope} * 2.3)\right)$$

Eq.**

$$\frac{de^2}{d\,\text{Intercept}} = -2\left(1.4 - (\text{intercept} + \text{slope} * 0.5)\right) + -2\left(3.2 - (\text{Intercept} + \text{Slope} * 2.9)\right) + -2\left(1.9 - (\text{Intercept} + \text{Slope} * 2.3)\right)$$

3.2.2 Types of Optimizers and How Optimizers Work?

→ During the backpropagation process, optimizers are used to update weigts and biases.

→ With the updated weight/biases, the forward pass computes to get a reduced Error.

→ During the training process of Neural network our aim is to try and minimize the loss function by updating weigh/bias as:

$$\boxed{\begin{aligned} w_{i+1} &= w_i - \eta \cdot \frac{\partial Loss}{\partial w_i} \\ b_{i+1} &= b_i - \eta \frac{\partial Loss}{\partial b_i} \end{aligned}}$$

where η: learning rate

$\frac{\partial loss}{\partial w_i}, \frac{\partial loss}{\partial b_i}$ = gradient of error w.r.t weight/bias

w_i, b_i = are old weight/bias

$w_{i+1}, b_{i+1} \Rightarrow$ Newly updated weight and bias

→ The detail of the mathematics how gradient decent works, along with numerical example are shown in the previous section

When updating parameter with gradient descent, the learning rate is always constant

$$\boxed{\theta_{i+1} = \theta_i - \eta \cdot g_t,} \quad g_t = \frac{\partial \ell}{\partial \theta_i}$$

η = LR = constant for the whole training process.

⇒ — Researchers came to an idea that an optimiser can change the LEARNING RATE as per previous gradient. By doing so the optimization process CONVERGE FASTER

(1) Adagrad

→ For this Adagrad was born. → Adaptive gradient

→ The Learning rate becomes

$$\boxed{\eta' = \frac{\eta}{\sqrt{\alpha_t + \varepsilon}}}$$

Initialize $\alpha = 0$

$$\alpha_t = \alpha_{t-1} + g_t^2, \rightarrow g_t^2 = \sum \left(\frac{\partial \ell}{\partial \theta}\right)^2$$

→ In every iteration, we add the square of the gradient in α_t so that it will have the history of the past gradients.

→ Now the learning rate will vary for every iterations

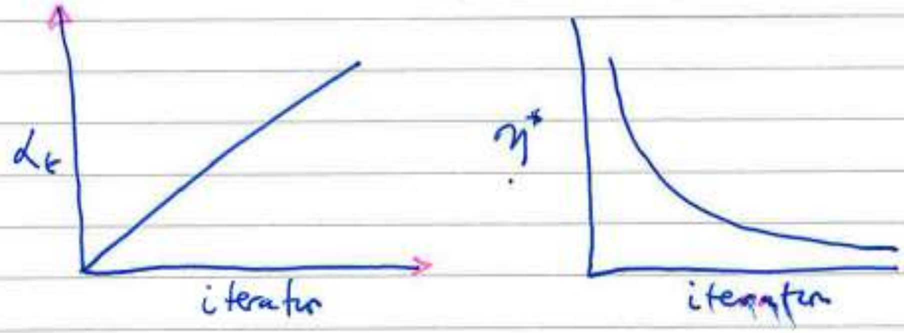
$$\boxed{\theta_{i+1} = \theta_i - \frac{\eta}{\sqrt{\sum \left(\frac{\partial \ell}{\partial \theta}\right)^2 + \varepsilon}} \cdot g_t}$$

$\varepsilon = 10^{-8}$ in case $\alpha_t = 0$

Weakness of Adagrad.

For every iterations, we add $\left(\frac{\partial L}{\partial \theta_i}\right)^2$, thus α_t will increase. As a result the $\eta^* = \frac{\eta}{\sqrt{\alpha_t+\epsilon}}$ will be smaller for every iteration

Thus the continual decay of η^* (LR) is the weakness of adagrad. Since for small value of η^*, the training process will stop



→(2) Adadelta

To overcome the issue of Adagrad, Adadela was born.

→ Here we don't need to chose learning rate. Adadelta automatically computes the learning set.

Initialize $\alpha=0$, $\Delta X=0$

$$\alpha_t = \alpha_{t-1} + g_t^2 \quad \text{— Adagrad}$$

Adadelta — $\boxed{\alpha_t = \rho\,\alpha_{t-1} + (1-\rho)\cdot g_t^2}$

$$\Delta\theta_t = -\frac{\sqrt{\Delta x_t + \varepsilon}}{\sqrt{\alpha_t + \varepsilon}} \cdot g_t$$

$$\Delta x_t = \rho\, \Delta x_{t-1} + (1-\rho)\, \Delta\theta_t^2$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

Δx – is the accumulated rate used to calculate learning rate

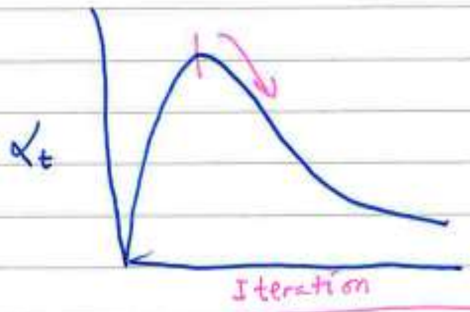
ρ – is used to avoid an infinity increase of α_t value, as show for Adagrad.

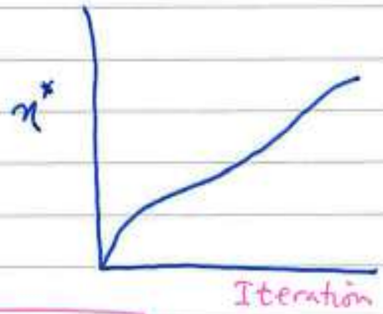
- In case of Adadelta, α_t value increase upto 50 iteration then decrease.

- Therefore, there is no problem of η^* decay. This is made possible by ρ – hyperparameter.

→ ρ is also known as decay constant

→ ρ value is typically $= 0.9$





$$\boxed{\theta_{t+1} = \theta_t + \Delta\theta}$$

$\Delta\theta$ is calculated by Adadelta

(3) Adam

Adam stands for adaptive moment estimation

Adam updates / adjust the learning rate adaptively for each parameters in the model based on the history of gradients calculated for each parameters

Adam optimizer

Initialize $M_0 = 0 \Rightarrow$ First moment vector

$V_0 = 0 \Rightarrow 2^{nd}$ " "

$$\boxed{\begin{aligned} m_t &= \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t \quad g_t = \frac{\partial \ell}{\partial \theta_i} \\ v_t &= \beta_2 V_{t-1} + (1-\beta_2) \cdot g_t^2 \\ \theta_t &= \theta_{t-1} - \frac{\eta \cdot \hat{m}_t}{\sqrt{\hat{v}_t}} \end{aligned}}$$

where $\hat{m} = \dfrac{M_t}{1-\beta_1^t}$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

default 0.9, 0.999.

β_1, β_2 are hyper parameter of adam, are initial decay rates used when estimating the first and the second moment gradients

M_t is the first order moment (i.e mean of gradient)

V_t is the second order moment, used to adjust the learning rate at time step, t

η :- global learning rate

$\varepsilon \approx 10^{-8}$

(4) RMSprop

- Root mean square propagation optimizer

→ Use exponentially decay averge of squared gradient and discards history from the extreme past.

$$\theta_{i+1} = \theta_i - \frac{\eta}{\sqrt{r_t + \varepsilon}} \cdot g_t. \qquad g_t = \frac{\partial l}{\partial \theta_i}$$

where, r is exponentially decaying averge

$$r_t = \beta r_i + (1-\beta) \cdot \left(\frac{\partial l}{\partial \theta}\right)^2$$

β = turning / hyperparameter

$\varepsilon = 10^{-8}$.

Types of optimizers

In keras, a number of optimizers are available

- SGD
- RMSprob
- Adam
- AdamW
- Adadelta
- Adagrad
- Adamax
- Adafactor
- Nadam

- During laboratory exercise, you can test the performance of the optimizers.

→ The most commonly used is Adam.

→ In the following we will review just some of these.

4.0 ANN MODELING.

WorkFlow - Summary

4.1 **Step 1:** Data - preprocessing / Feature Eng

- Load data file
- Data - Preprocessing
  - → cleanining
  - - feature selection
  - → Standardisation / scaling

4.2 **Step 2:** ANN Modeling

- Splitting scaled data into training/test
- Creating ANN model / Fully connected
  - → Input layer
  - → Hidden layer
  - → Output layer
- Compile the model → model.fit()
  - → perform optimzation
- Fitting ANN with training data

4.3 **Step 3:** Model prediction

- Perform inverse transform of scaled data to the original
- Predict with test data
- Compare predicted (y_{pred}^{test}) with the true (y_{true})

4.4 **Step 4:** Model Performance accuracy analysis

- Using $y_{pred.}$ and y_{test}, perform the goodness fit of the Model.
  R^2, MSE

→ Following these, the next page present Step by step ANN Modeling

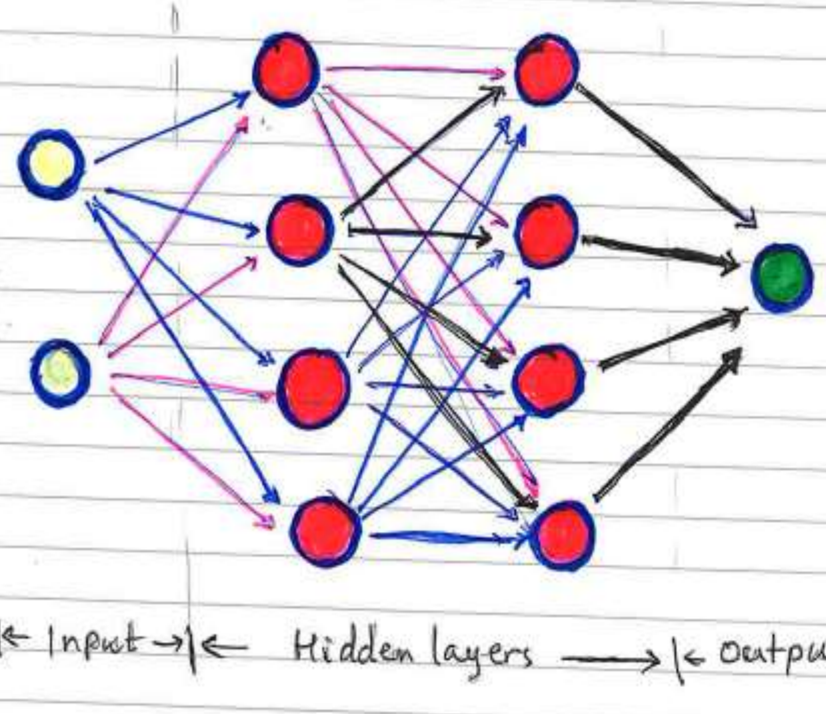
4. ANN Modeling

Laboratory ANN ·modeling

Consider the following Network.
We will learn the ANN modeling algorithm
In the previous Section, we learned the how perceptrons perform the computation to get the output, and based on the loss function, how the gradient decent algorithm performs to update the weights and biases.

In this Section we will learn how the builtin tensorflow / keras library perform the Computation.

We will step-by-step look at how the Computation performed.

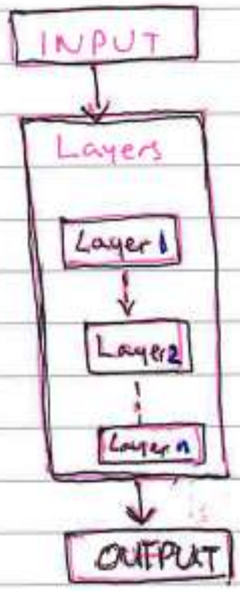


How to build ANN and train data ?

Let us examine / describe the model definition

(a) Sequential ( ) → Initialize the ANN

We start the modeling with a Sequential ( ) function. It specifies that the network is a linear stack of layers. Example : → Describe the network Sequential model



(b) Model.add ( )

- It allows to add Layers

(c) Dense

- It means that neurons between Layers are FULLY CONNECTED

(d) input_dim

- It defines the number of features in the training dataset

(e) activation

- defines activation function

Example Relu, tanh, Sigmoid ...

(f) loss

- It allows us to select the cost/loss function

Ex. MSE

(g) Optimizers

→ It allows to select the learning algorithm

Ex. adam, SGD, RMSopt, adagrad,

(h) metrics

- It allows to select the performance metrics to be saved for further analysis

Ex. R^2, MSE.

(i) model.fit()

→ To initialize the training

(j) Kernel_initializer

→ This allow to initialize weight and bias

Feature scaling is also known as data normalization. is part of data-preprocessing

It is reported that when using gradient-decent-based optimization algoritum, feature-scaling can help speedup convergence and improve model performance. -(i.e. both accuracy and stability of ML-model)

→ For regression problem, it is often desirable to scale /or /transform both
  * → Input features and
  * – target variables

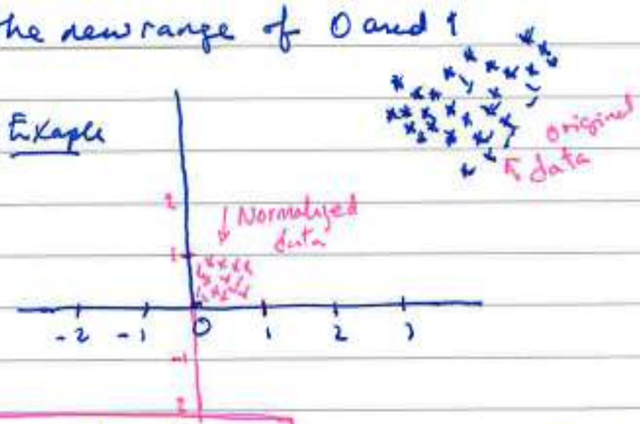
The two most popular techniques for scaling numerical data prior to modelling are

(1) Normalization
(2) Standardization

(a) Normalization:

→ Normalization is rescaling of data from the original range so that all values are within the new range of 0 and 1

Example



$$\boxed{X_{Norm} = \frac{X - X_{min}}{X_{max} - X_{min}}} \Rightarrow X_{Norm} = [0, 1]$$

If we want X_{NOR} to be between [a, b] ER

$$\boxed{X_{Norm} = a + \frac{(X - X_{min})(b-a)}{X_{max} - X_{min}}}$$

Example : Data normalization

Assume, For a data set, the min and Max Observable values are 30 and -10

What is the normalized value of 18.8 ?

$$\boxed{y = \frac{x - x_{min}}{x_{max} - x_{min}}}$$

$$= \frac{18.8 - (-10)}{30 - (-10)}$$

$$= \underline{\underline{0.72}}, \text{ which is in } [0, 1]$$

→ We can normalize dataset using the Skikit-learn object MinMaxScalar

→ The default scale for the MinMaxScalar is to rescale variables into the range [0,1].

Example:

```
X = df.iloc[:, 1:5].values
```

~~from sklearn import preprocessing~~

~~min_max_scalar =~~ preprocessing.MinMaxScalar (

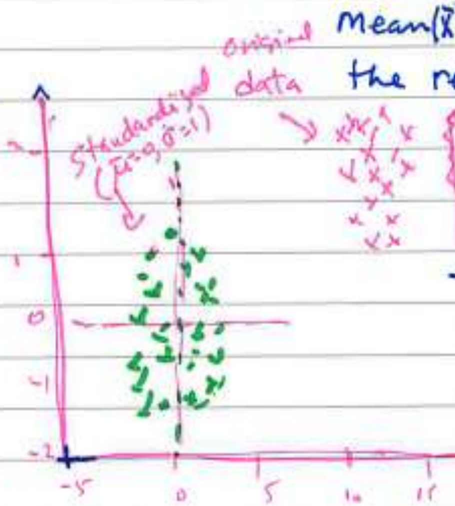
Example

```
import sklearn import preprocessing
min_max_scalar = preprocessing.MinMaxScalar(feature_range=(0,1))

# Scaled feature
  scaled_x = min_max_scalar.fit_transform(x)
 # Display
 print("After min max scaling: \n", scaled_x)
```

(2) Standardization

Feature standardization makes the value of each feature in the data have zero mean and unit variance.

> The method calculates the statistical mean and standard deviation of the attribute values, subtract the Mean($\bar{x}$) from each value, and divide the result by the Standard deviation(σ)



$$\boxed{X_{std} = \frac{x - \bar{x}}{\sigma}}$$

→ subtracting the mean from the data is called Centering

→ whereas dividing by the standard deviation is called Scaling

– The method sometimes called 'center scaling

Mean → $\bar{x} = \frac{\sum x_i}{\text{Count}(x)}$

Standard deviation → $\sigma = \sqrt{\frac{\sum (x_i - \bar{x})^2}{\text{Count}(x)}}$

Example:

Assume that $\bar{x} = 10$, $\sigma = 5$.

Using standardization, what is the standard value of 20.7?

$$y_{st.} = \frac{x - \bar{x}}{\sigma} = \frac{20.7 - 10}{5}$$

$$\boxed{y = 2.14}$$

You can Standardize your dataset using Scikit-learn object StandardScalar

Example

x = df.iloc[:,1:5].values

Import preprocessing library

```
from sklearn import preprocessing
Standardisation = preprocessing.StandardScalar()

# Scaled feature
Standardized_x = Standardisation.fit_tranform(x)

# Display
Print('After Standardisation :\n', Standardized_x)
```

Note:

- The splitdata NEEDS to be Standardised befor ML. modeling

⇒

4.0

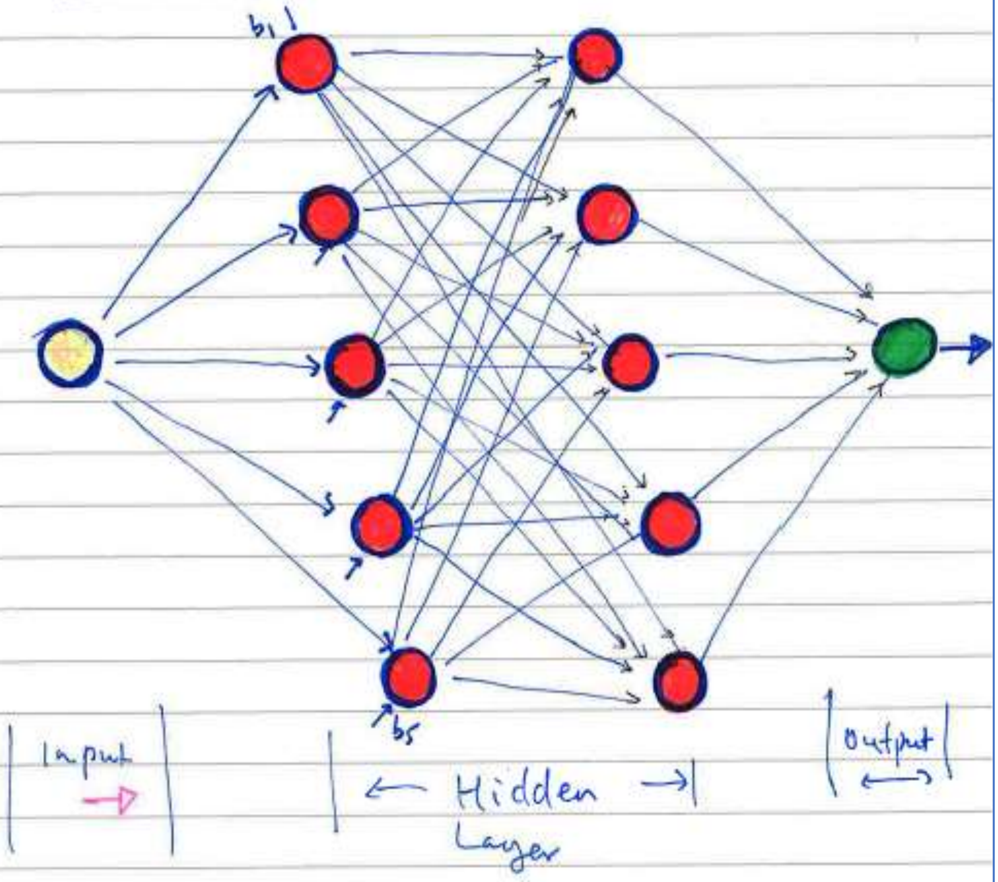
Example — ANN Modeling Application — Regression

Assume dataset — X | y

- a single feature
- a single target

- 2 hidden layers with 5 nodes each
- Activation = Relu.
- Loss = mse
- Optimizer = adam.

Task → Create the ANN Model
Show step-by-step modeling / performance evaluation



4. ANN- Modeling

Step 1: Data Pre processing

Before modeling, data must be cleaned, appropriate features must be selected and finally, it will be scalled for the better model performance

- This section has been done during Lab# . You must run the process and the final cleand data saved in excel will be loaded here.

a) # Load cleaned data

```
Import Pandas as pd
Import numpy as np
logData = pd.read_excel('balistic.xlsx')
logData.head()
```

Out:

	Vo	ang	Time	R
0	-	-	-	-
1	-	-	-	-
2	-	-	-	-
3	-	-	-	-
4	-	-	-	-

b) Separate Target and Predictor variables

```
TargetVariable = ['Time']
or # TargetVariable = ['R']
Predictors = ['Vo', 'ang']

X = logData[Predictors].value
y = logData[TargetVariable].value
```

(c) Standardise and split data for training/test

(c1) # Scaling / standardisation of dataset

```
from sklearn.preprocessing import StandardScalar

PredictorScalar = StandardScalar()
TargetVarScalar = StandardScalar()
```

c2 # Storing the fit object

```
PredictorScalarFit = PredictorScalar.fit()
TargetVarScalarFit = TargetVarScalar.fit()
```

(c3) # Generating the Standardised values of X and y

```
X = PredictorScalarFit.transform(X)
y = TargetVarScalarFit.transform(y)
```

(c4) # Split the scalled data into training / Test

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
                                                    random_state=42)
```

(c5) Quick sanity check with the shape of training and testing dataset

```
print('X_train:', X_tain.shape)
print('y_train:', y_train.shape)
print('X_test:', X_test.shape)
print('y_test:', y_test.shape)
```

```
out: X_train: (210, 2)
     y_train: (210, 1)
     X_test : (90, 2)
     y_test : (90, 1)
```

STEP 2: ANN Modeling

(2a) Import the libraries

```
from keras.models import Sequential
from keras.layers import Dense
```

(2b) Creating ANN

```
model = Sequential()
```

(2b_I) Defining Input layer / First hidden layer

(5 nodes)

.add(

```
model.add(Dense(unit=5, input_dim=1,
                kernel_initializer='normal',
                activation='relu'))
```

(2b_II) Defining the second hidden layer

(5 nodes)

NB. { After the first hidden layer, we don't have to specify input_dim as keras configur it automatically }

```
model.add(Dense(unit=5, kernel_initalizer='normal',
                activation='relu'))
```

(2b_III) Define the output neuron

[output neuron is a single fully connected (node)]

```
model.add(Dense(1, kernel_initializer='normal'))
```

units=1

STEP(2c) Compile the model → • Compile()

→ Model. Compile(loss = 'mean_squared_error', optimizer = 'adam')

[other optimizers: SGD, RMSopt, .

STEP(2d) Fitting the ANN to the training data
• fit()

→ model. fit(X_train, y_train, batch_size= 20, epochs= 1000, Verbose = 0)

here we define the model fit as history thinking that the model performance history parameter will be displayed at the end.

→ history = model. fit(X_train, y_train, batch_size = 20, epochs = 1000, Verbose = 0)

Verbose = 0 = ~~training~~ progress (silent) will not be seen

Verbose = 1 = will be seen. (animation) for each epoch. (progress bar)

Verbose = 2 → one line per epochs.

Step 2e: Model Summary

To display model Summary → • Summary()

→ model. summary()

Output

Layer (type)	Output shape	Parameters (weight + bias)
dense_1 (Dense)	(None, 5)	15
dense_2 (Dense)	(None, 5)	30
dense_3 (Dense)	(None, 1)	6

STEP 3: Model Prediction

\# Fitting ANN to the training data .fit()

```
model.fit(X_train, y_train, batch_size=20, epochs=10(
          verbose=0)
```

\# Generating (y_pred) Predictions on testing dat
.Predict()

```
Predictions = model.Predict(X_test)
```

\# Scaling the predicted data back to the original data set
.inverse_transform()

```
* Predictions = TargetVarScalarFit.inverse_transform(predictions)
```

\# Scaling the y_test data back to the Original dataset

```
* y_test_orig = TargetVarScalarFit.inverse_transform(y_test)
```

\# Scaling the X_test data back to the Original dataset

```
Test_Data = PredictorScalarFit.inverse_transform(X_test)
```

→ Display data → Create Data Frame()

```
TestingData = pd.DataFrame(data=Test_Data, columns=Predictors)
TestingData['True data'] = y_test_orig
TestingData['ANN_Predicted'] = Predictions
TestingData.head()
```

Out put:

	Vo	ang	True data	ANN predicted
0	-	-	-	-
1	-	-	-	-
2	-	-	-	-
3	-	-	-	-
4	-	-	-	-

Display predicted and true data

```
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure

figure(figsize=(8, 6), dpi=380)
plt.plot (y_test_orig)
plt.plot ( Predictions)
plt.ylabel (
plt.xlabel (
plt.legend (
plt.show( )
```

5 Summary

In this chapter both the concept how the ANN computation performed and the ANN modeling in Keras.

In chapter 4, the step-by-step process of ANN is presented.

During Lab class, you will use synthetic data that is computed from the physics model. Then, you will use the data to train/model with ANN. From the result, you will learn how ANN predict/recover the physics data.

After testing the ~~syn~~ ANN with the synthetic data, you will use field data that you have performed datapreprossing during Lab 1.

From these two exercises, you will have good understanding how ANN works.