

12-2

Deep Learning:

ANN:

Artificial Neural Network(DNN)

All the kind of ML problems we can achieve using DNN

CNN: Convolution Neural Network

All the kind of image problems

Object detection

Video annotation

RNN: Recurrent Neural Network

Related to NLP Problems

We need to understand what's happens inside a Neuron to understand neural networks

$$x_1 \rightarrow neuron \rightarrow y = b_1 x_1$$

$$x_1 \rightarrow neuron(summation + bias) \rightarrow y = b_0 + b_1 x_1$$

$$y = b_0 + b_1 x_1 + b_2 x_2$$

If you observe in the above neurons we, are just passing the inputs

It is kind of linear combination of inputs

Our model will not identify the patterns

Imagine our brain so many neurons will available

Each neuron connect with other neurons is such a way, even a complex problem also our brain try to get the solution

There are two types of maths:

1. Linear Maths

Passing the outputs

2. Non-Linear Maths

It is able to understand the patterns

$$P(y) = \frac{1}{1 + e^{-(b_0 + b_1 x_1 + b_2 x_2 + \dots + b_n x_n)}}$$

$$P(y) = \sigma(\text{Linear regression equation})$$

$$\sigma(x) = \text{sigmoid function} = \frac{1}{1 + e^{-x}}$$

$$b_0 = w_0$$

$$b_1 = w_1$$

$$y = w_0 + w_1 x_1 + w_2 x_2 + \dots + w_n x_n$$

$w_1, w_2 \dots w_n$ are weights

Step 1: We have input let assume we have inputs: x_1

Step 2: We have one weight for x_1 : w_1

Step 3: we have bias

Neuron = summation + sigmoid

Sigmoid is a non-linear function is useful to logistic regression problems, binary classification problems

There are many non-linear functions and linear functions available

After computation will pass the output to the Non-linear function, this non-linear function called Activation Function

We have some activation function available for the specific use case problem

. sigmoid

. softmax

. Tanh

. ReLU

All the deep learning is all about what happens inside the neuron

Inside the neuron there are two operations

Summation and activation functions

In neural networks we have mainly Three types of layers

1. Input layers
2. Hidden layer
3. Output layer

Each input will pass with each other neurons and will have weights

w_{11}, w_{21}, w_{31}

Neuron number one has linear combination of inputs and weights

First neuron output = $w_0 + x_1 * w_{11} + x_2 * w_{21} + x_3 * w_{31}$

13-2

$W_{mn} = \text{input (m) collaborating with neuron(n)}$

$O_{i1} = \text{first neuron output from input layer}$

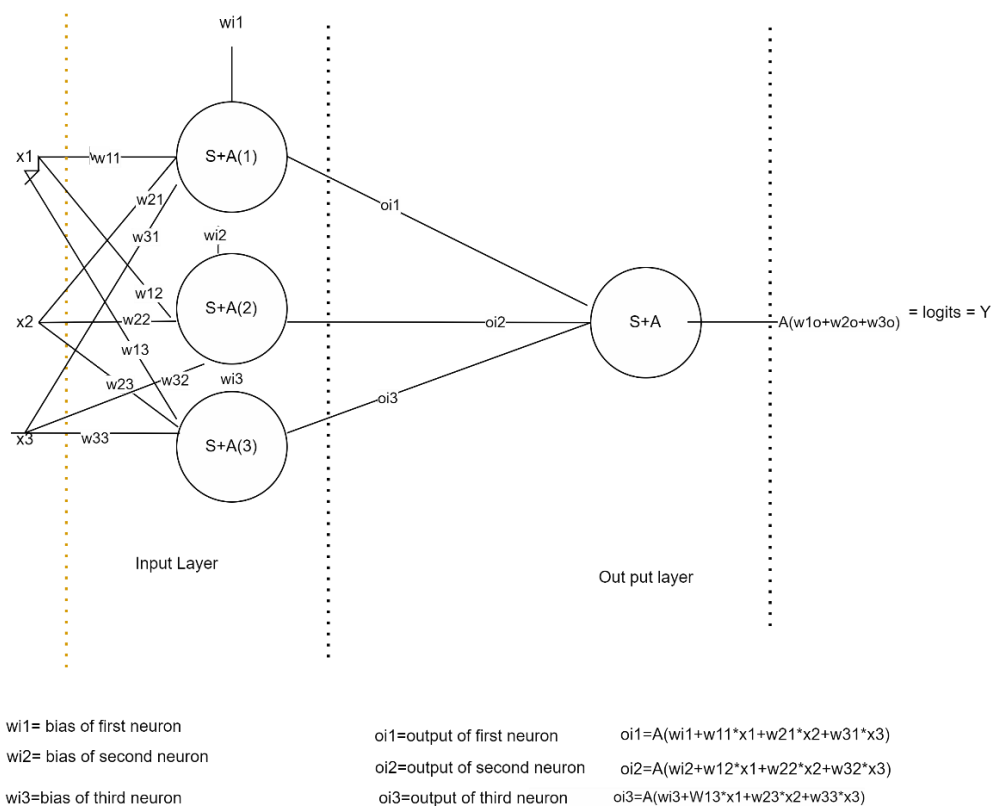
$= \text{Activation}(W_{i1} + X_1 * W_{11} + x_2 * W_{21} + X_3 * W_{31})$

$O_{i2} = \text{second neuron output from input layer}$

$= \text{Activation}(W_{i2} + x_1 * W_{12} + X_2 * W_{22} + X_3 * W_{32})$

$O_{i3} = \text{Third neuron output from input layer}$

$= \text{Activation}(W_{i3} + X_1 * W_{13} + x_2 * W_{23} + X_3 * W_{33})$



w_{11} = weight of first input with first neuron w_{21} = weight of second input with first neuron w_{31} = weight of Third input with first neuron

w_{12} = weight of first input with second neuron w_{22} = weight of second input with second neuron w_{32} = weight of third input with second neuron

w_{13} = weight of first input with third neuron w_{23} = weight of second input with third neuron w_{33} = weight of Third input with third neuron

$$a_x + b_y = p$$

$$c_x + d_y = q$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} P \\ Q \end{bmatrix}$$

$$\begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \end{bmatrix} X \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} w_{i1} \\ w_{i2} \end{bmatrix} = \begin{bmatrix} O_{i1} \\ O_{i2} \end{bmatrix}$$

Will have an activation function on this

- How good your weight is (knowledge) so, weights are the most important thing

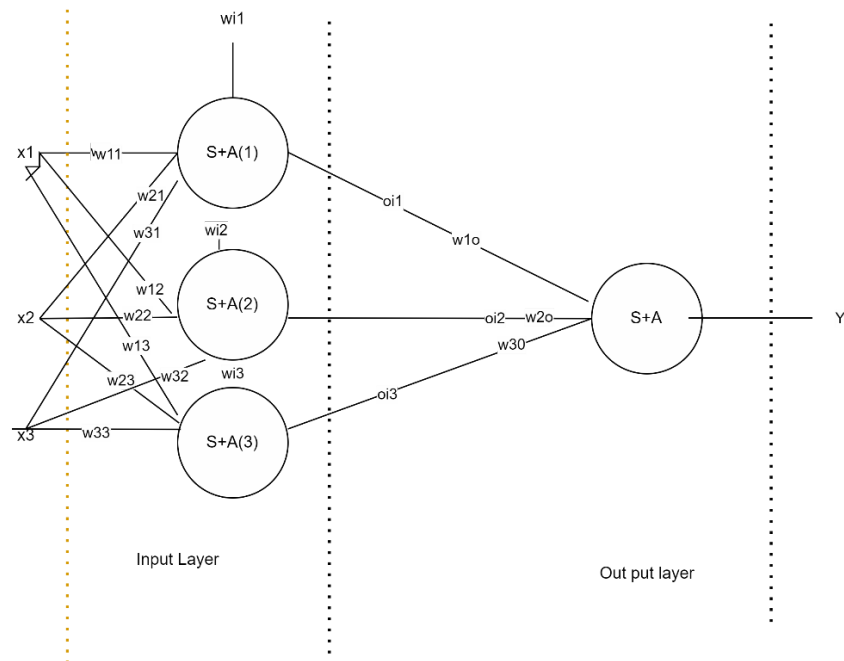
No of parameters:

Output layer parameters are = 3

Input layer parameters are $9+3 = 12$ (9 are each feature (3) to every other neuron + 3 bias for each neuron)

Total parameters or weights = $3+12 = 15$

(without hidden layer)



w_{i1} = bias of first neuron
 w_{i2} = bias of second neuron
 w_{i3} = bias of third neuron

oi_1 = output of first neuron
 oi_2 = output of second neuron
 oi_3 = output of third neuron

$oi_1 = A(w_{i1} + w_{11}x_1 + w_{21}x_2 + w_{31}x_3)$
 $oi_2 = A(w_{i2} + w_{12}x_1 + w_{22}x_2 + w_{32}x_3)$
 $oi_3 = A(w_{i3} + w_{13}x_1 + w_{23}x_2 + w_{33}x_3)$

w_{11} = weight of first input with first neuron w_{21} = weight of second input with first neuron w_{31} = weight of Third input with first neuron
 w_{12} = weight of first input with second neuron w_{22} = weight of second input with second neuron w_{32} = weight of third input with second neuron
 w_{13} = weight of first input with third neuron w_{23} = weight of second input with third neuron w_{33} = weight of Third input with third neuron

w_{1o} = weight of first neuron with output layer
 w_{2o} = weight of second neuron with output layer
 w_{3o} = weight of third neuron with output layer

$$y = \text{sigmoid}(oi_1 * w_{1o} + oi_2 * w_{2o} + oi_3 * w_{3o}) = 0.7(\text{logit}) = \text{Yes}$$

$$y = \text{Activation}(oi_1 * w_{1o} + oi_2 * w_{2o} + oi_3 * w_{3o})$$

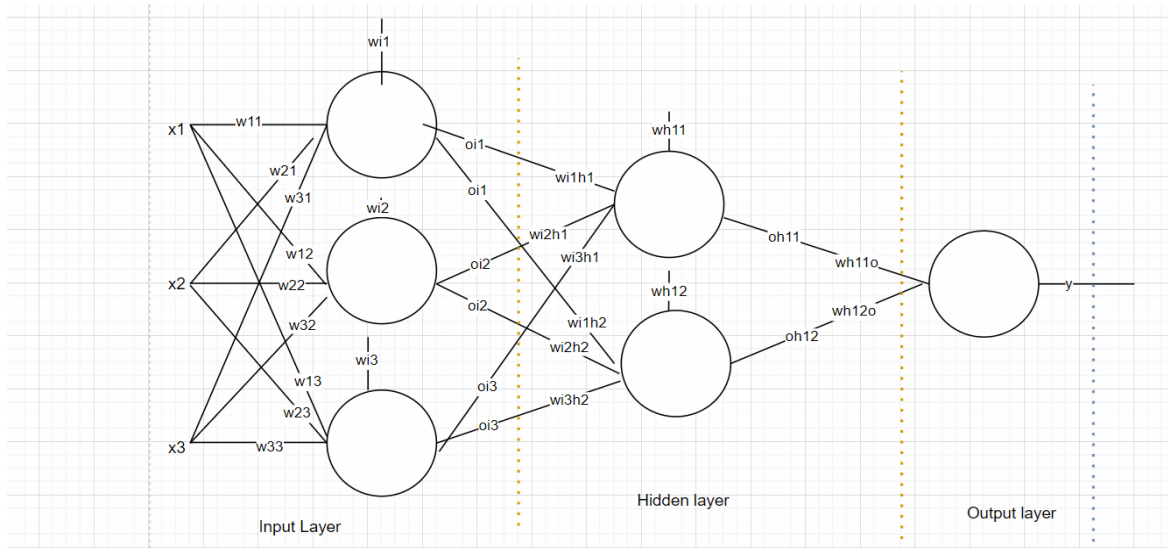
Input layer →

Neurons will depend on number of inputs

Hidden layer -1 → How many neurons

Output layer →

Bi classification one neuron



$w_{i_1 h_1}$ = input layer neuron – 1 with hidden layer – 1

$w_{i_1 h_2}$ = input layer neuron – 1 with hidden layer – 2

w_{h11} = bias hidden layer – 1(for)first neuron

w_{h12} = bias hidden layer – 2(for)secondneuron

Weights = $6+2=8$

Hidden layer has two neurons = 2 outputs

Output layers

O_{h11} = output of hidden layer 1 of first neuron

O_{h12} = output of hidden layer 1 of second neuron

w_{h11o} = weight hidden layer – 1 neuron – 1 to output

w_{h12o} = weight hidden layer – 1 neuron – 2 to output

Output layer =2

Total weights = input layer + Hidden layer + output layer

Input layer = $9+3=12$

Hidden layer = $6+2=8$

Output layer =2

$12+8+2=22$ weights need to be train

14-2:

We have different types of networks is available

Standard Network

Input layer: No need of provide weights, No need of provide bias

Number of neurons = Number of input features

Hidden Layer → More Hidden Layer, model will be complex

Between input layer and hidden layer, we have weights, and hidden layer neurons has bias

Output layer:

We have weights between hidden layer and output layer

How many neurons we need to choose in output layers

Bi classification: one neuron

Multi classification: number of neurons = number of labels

Hidden Layer is used to understand more patterns

One neuron layer information pass through another neuron layer, it is a continuous process to understand more patterns

Hidden layers also refer to the model complexity like in Decision tree depth, like in linear regression adding more features

Hidden layers you need to compare with bias-variance trade off

More hidden layers → overfit

Less hidden layers → underfit

How Many hidden layers we need to choose

How many total parameters to be trained?

Input neurons = 28

Hidden neuron = 100

Output neurons = 10

$28 \times 100 + 100 = 2900$

$100 \times 10 + 10 = 1010$

Total = 3910

Image of size 28×28

Input neurons = $28 \times 28 = 784$ pixels need to make a flatten vector

Hidden layer = 128

Output layer=10

$28*28 *128 +128 = 100480$

$128*10+10 = 1290$

Total =101770

ARCHITECTURE OF NEURAL NETWORKS

Import questions

1. How to choose weights
2. How to choose activation function
3. How to choose hidden layers
 - a. Model will be over fit
 - b. How to avoid over fit
4. How to choose hidden layers Neurons
 - a. More parameters to train
 - b. More time is required
 - c. How can I optimise that

How to choose the Weights:

Goal is to minimize the error to find suitable weights

Initially will provide weights randomly

Forward propagation

Assigning all the random weights to all the neurons and check compare the ypred with ytrue to minimize the error

Initialize the random weights, inputs values pass through weights, and neuron sum of all the inputs with weights and apply the activations and pass through another layer

Back propagation:

To minimize the error, go back to previous layers and modify the weights and then do the forward propagation to check the cost function loop till the error is minimum

Check the error to reduce the error we go back and update the weights

Every layer weight is responsible for to get error either low or max

Forward Propagation + Backward Propagation = Epoch

10 Epoch: 10 times we are iterating

Forward : $\text{Activation}(\text{inputs} * \text{weights}) = \text{output}$

Loss or cost, you want to minimise that

Backward : update the weights

How to update the weights:

In linear regression : OLS method

In Neural network: Gradient descent

Gradient means = slope

$$y_2 - y_1 = \Delta y = \partial y$$

$$x_2 - x_1 = \Delta x = \partial x$$

$$m = \text{slope} \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x} = \frac{\partial y}{\partial x} = \tan \theta$$

I want to find minimum value $= y = x^2$

Case -1:

At $x=4$ I assume we have minimum point for $y = x^2$

If you want to get minimum, there is no slope at minimum values

$$\frac{\partial y}{\partial x} = 2 * x$$

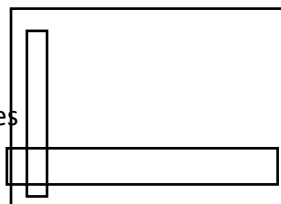
Now substitute $x=4$ at $\frac{\partial y}{\partial x} = 2 * 4 = 8$

Case -2 :

If slope is positive

$$x_{new} = x_{old} - \left(\frac{\partial y}{\partial x} \right) \text{ at } x = x_{old}$$

There is no slope at minimum values



Case -3 :

If slope is positive

$$x_{new} = x_{old} + \left(\frac{\partial y}{\partial x} \right) \text{ at } x = x_{old}$$

$$x_{new} = x_{old} - \alpha(\text{learning rate } (0 - 1)) * \left(\frac{\partial y}{\partial x}\right)$$

Gradient Descent

$$W_{new} = W_{old} - \alpha * \left(\frac{\partial J}{\partial w}\right)_{w=W_{old}} \quad \mathbf{w = weights}$$

$$b_{new} = b_{old} - \alpha * \left(\frac{\partial J}{\partial b}\right)_{b=b_{old}} \quad \mathbf{b = bias}$$

Iterative above equation till you get differentiation cost function =0

$W_{new} = w_{old}$ at some point weights are fixed, which means cost function =0

15-2

Neural Network:

Inputs will pass through the randomly initialise weights, it provide the output

It provide the output: **Forward propagation**

Then will calculate the error or cost function(j)

We need to minimise the error,

Will go back through the network by updating the weights: **back propagation**

The technique we use to update the weights is **gradient descent**

Gradient Descents

Batch Gradient:

- All the train observations we are passing at a time to NN
- Assume that 50k train observations at a time
- Average error you will calculate
- Again, it will come back to update the weights
- For forward propagation 1 sample → 1 min
- For 50k observations: 50k min required for only forward propagation
- For backward propagation also 50k min
- For one epoch 50k min + 50K min = 100k min
- When we calculate the average error : avg(50k)

Mini batch Gradient:

- Divide data into batches
- 50k divides into 50 batches

- Each batch 1000 observations
- Forward + backward is become fast
- When we calculate the average error: avg(1k)

Stochastic Gradient

One by one observation

When we calculate average error: avg (1k)

Exponential moving average(uses for the stock market)

Step - 1: will randomly initialise the weight and bias

Step - 2: $y = b_o + w * x$

Step - 3: when we substitute b_o and w and pass the input will get predictions

Step - 4: we are calculating the error

Step - 5: Total cost function = avg of all $error^2$

Step - 6: will differentiate the cost function wrt weight

Step - 7: We choose the learning rate

Step - 8 : we are updating the weights $W_{new} = W_{old} - \alpha * \left(\frac{\partial J}{\partial w} \right)_{w=W_{old}}$

Step - 9 : The same process happened for bias also

Step 10 : The entire flow will repeated many times (epochs)

16-2:

How to choose the weights: Gradient descent back propagations

2. Activation Functions:

In neural network neurons = summation + activation

If we have only summation: linear combination of inputs = Regression problem

It will not able to identify complex solutions

Complex solutions require non-linearity

To provide the non-linearity in the NN we use activation functions

Non-linear functions are called activation functions

1. Sigmoid
2. Softmax
3. Tanh
4. ReLU (Rectified Linear Unit)

5. Leaky ReLU

What is the Equation

What is the range

On which layer will use

$$\text{Sigmoid} = \sigma(x) = \frac{1}{1+e^{-x}}$$

Range = 0 to 1

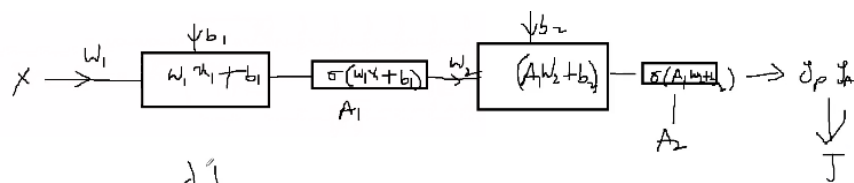
Sigmoid activation function use at output layer only and only for Binary classification

If you have output labels 0 and 1 then use sigmoid function

Why sigmoid function will not use at hidden layers

Vanishing Gradients or Explode Gradient

$$w_{new} = w_{old} - \alpha * \left(\frac{\partial J}{\partial w} \right)_{w=w_{old}}$$



Chain Rule:

$$\frac{\partial J}{\partial w_2} = \frac{\partial J}{\partial A_2} \times \frac{\partial A_2}{\partial w_2}$$

$$\frac{\partial J}{\partial w_1} = \frac{\partial J}{\partial A_2} \times \frac{\partial A_2}{\partial w_2} \times \frac{\partial w_2}{\partial A_1} \times \frac{\partial A_1}{\partial w_1}$$

We are differentiating the activation functions also

$$\sigma(w_1 * x + b_1) = \frac{1}{1 + e^{-(w_1 * x + b_1)}}$$

$$\frac{\partial}{\partial} \left(\frac{1}{1 + e^{-x}} \right) = \frac{e^{-x}}{(1 + e^{-x})^2}$$

$$\text{sigmoid} = \frac{1}{1 + e^{-x}} = z$$

$$\text{sigmoid}' = 1(-Z) * z$$

Sigmoid is for only bi-classification

Differentiation of constants are constants

The weights become smaller are zero at some point and weights updating is not possible to that layer so we can't use the sigmoid function in the hidden layers

If you use sigmoid function in hidden layers at some point for low w value the gradient becomes zero and for high value w

Due to these gradients become zero the weights updation not happened

Sigmoid, Tanh, Softmax is same as sigmoid function what is the equation, what is the range, on which layer it is used, graph

17-2 [practice check the ipynb]

NLTK

Import strings

Import re

Findall() # all the given values

Remove the punctuations of the text

```
text_1 = "H@i kow$hik wh@t'$ up"
```

```
re_punct = re.compile('%s'%re.escape(string.punctuation))
```

```
[re_punct.sub("",w) for w in text_1.split()]
```

Stop words

Stem words

Pos_tag

```
Nltk.help.up()
```

Wordnet

Wordcloud

19-2:

Softmax:

Softmax is for multi classification

$$P(c1) = \frac{e^{c1}}{e^{c1} + e^{c2} + e^{c3}}$$

$$P(c2) = \frac{e^{c2}}{e^{c1} + e^{c2} + e^{c3}}$$

$$P(c3) = \frac{e^{c3}}{e^{c1} + e^{c2} + e^{c3}}$$

$$softmax = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$$

Max (c1, c2, c3) is your final answer

Probabilities

Softmax used for multiclassification at output layer

Vanishing gradient problem

Tanh:

ReLU : Rectified Linear unit:

- It avoids the vanishing Gradient Problems
- Activations is used to provide non linearity in the neurons
- It is not linear curve it is a non-linear curve only because all the negative values considered as 0
- It is used on hidden layers
- Suppose if all the inputs are -ve values then total network will fail

Max(0,x)

If input is >0

X= input

Else

X=0

Leaky ReLU:

X is input suppose this input is -ve value: 0

X is input suppose this input is -ve value: a*x

If +ve value then x

a alpha learning rate

Equation

Range

Graph

Use case

Vanish Gradients

Inputs -ve values

For the Linear regression we are applying the activation function

20-2:

Regularization Techniques :

How to find the weights

Activation functions

Over fit if use more hidden layers

Over fit: low train error and High-test error

Low bias and High variance

Model complexity increase → analogy → Depth of tree

Model complexity increase → analogy → Adding more features

Model complexity increase → analogy → adding more hidden layers also more neurons

Regularization Techniques:

Ridge and Lasso Regression Techniques

→ Generally in regression aim is to reduce cost function

$$\begin{aligned} &\rightarrow y = b + w x \\ &J = \text{cost function} + \lambda * (\text{slope})^2 \end{aligned}$$

Because of the extra term J not equal to zero even though model overfitted

We will feel like error is not equal to zero, this might not be a good model will try to get another equation

Step - 1: We want to find the weights : OLS and GD

Step - 2: we will form equation: $y = 1.5 + 1.4 * x$

Step - 3: Will pass train data , y predictions

Step - 4: J = cost functions

Step - 5: $J = 0$ (very happy)

Step - 6: Pass the test data, Test error huge ==== BAAAM

Step - 7: We realized model is overfitted

Ridge regression:

Step - 1: We want to find the weights : OLS and GD

Step - 2: we will form equation: $y = 1.5 + 1.4 * x$

Step - 3: Will pass train data , y predictions

Step - 4: $J = \text{cost function} + \lambda * (\text{slope})^2$

Step - 5: $J = 0 + \lambda * (\text{slope})^2$

Step – 6: you will feel we have still some j

Step – 7: Will try to create another equation

$$y = b + wx$$

$$J = \text{cost function} + \lambda * (w)^2$$

$$y = b + w_1x_1 + w_2x_2$$

$$J = \text{cost function} + \lambda * (w_1 + w_2)^2$$

$$\lambda = 0 \text{ to } \text{inf}$$

Best use of this is, suppose model complexity is high, you don't want to reduce your features, and you want avoid over fit(ridge regression)

In ridge regression slope values will never =0

Lasso Regression:

$$J = \text{cost function} + \lambda * |\text{slope}|$$

Lasso regression makes weights = 0 after so many iterations

Huge maths research paper is there

Both are used to avoid over fitting

Comes under regularization methods

If you want all the features go for Ridge $L2 = J = \text{cost function} + \lambda * (w)^2$

If you want less feature then go for Lasso $L2 = J = \text{cost function} + \lambda * |w|$

Lasso is not only regularization method it is also feature selection method

Choosing λ is not easy (penalize)

Elastic Net(alpha, lambda):

L1 : Lasso(alpha) L2 : Ridge (lambda) → Elastic Net

2 Drop out:

For every iteration drop the neurons randomly

Cross validation: Train and test

For every iteration will drop neurons randomly as well as we include neurons in training

When you drop the neurons the model become less complex while developing NN,

Drop out =0.3 assume

Suppose in hidden layer we have 10 neurons

Every time randomly 3 neurons will drop 7, neurons are present

3 Data Augmentation

4 Early stopping:

Forward vs Backward

Tree pruning

Optimizers:

Gd with

GD with momentum

Adaptive GD

RMS Prop

ADAM

21-2 practice

Tensor flow and keras

Tensor flow is language is always based on tensors

List == array == tensors (Vector representation)

Keras backend of tensor flow

Keras.datasets

Inbuilt datasets

returns two tuples

```
(X_train,y_train) ,(X_test, y_test) = minist.load_data()
```

```
X_train.shape, y_train.shape
```

60k images of size 28x28 = 57 pixels

```
X_train[0]
```

28 lists are there

in each list 28 values are there

```
X_train[0] # first image
```

in image we have 784 pixels are there

28 X 28

RGB picture : color picture

Gray : not a color picture

RGB values ranges between 0 to 255

Rainbow : 7

```

# 2^7 = 256
# 0 = black
# 255 = white
# any value from these is a combination of vibgyor
y_train
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(X_train[i],cmap='grey')
    plt.xlabel(class_names[y_train[i]])
plt.show( )

```

- . We read the data
- . we understand the values
- . we understand the shape of the data
- we plot the data
- . we understand the labels of the data
- we normalize the data
- . here train and test data both are available
- here X(input data), y (target data) already provided

Build the model with TF 2.0

- we will import the necessary layers to build the model. The Neural Network is constructed

****Sequential Process****

****Flatten layer --vectorizing the image values****

****Dense layer (hidden layer)****

****input layer:**** Initialize data for the neural network

****Hidden layer:**** Intermediate layer between input and output layer and place where all

****Output layer:**** produce the result for given inputs

****Flatten()**** is used as the input layer to convert the data into a

1-dimensional array for input it to the next layer.

Our image 2D image will be converted to a single 1D column.

`input_shape = (28,28)` because the size of our input image is 28x28.

`Dense()`** layer is the regular deeply connected neural network layer.

It is most common and frequently used layer.

We have a dense layer with 128 neurons with activation function `relu`.

The rectified linear activation function or ReLU for short is a piecewise linear function that will output the input directly

if it is positive, otherwise, it will output zero.

. The output layer is a dense layer with 10 neurons because we have 10 classes.

. The activation function used is softmax.

. Softmax converts a real vector to a vector of categorical probabilities.

. The elements of the output vector are in range (0, 1) and sum to 1.

. Softmax is often used as the activation for the last layer of a classification network because the result could

probability distribution.

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Flatten, Dense
model = Sequential()
# input layer starts flatten 28*28 = 1-d
model.add(Flatten(input_shape=(28,28))) # 28x28 = 784
# one hidden layer
# how many neurons want to use :128
# which activation function want to use: 'relu'
# bcz it avoids vanish gradients problem
model.add(Dense(128,activation='relu')) # 128 : 784*128+128
# output layer
# how many classes are there =10
# how many neurons = 10
# activation function = softmax
model.add(Dense(10,activation='softmax')) # 128 is attached with 10 neurons = 128X10=1280 +10
model.summary()
```

Model compilation

****Loss function****

****Optimizer****

****Metrics****

****Log loss (Binary cross-entropy)****

****Cross Entropy loss****

22-2

****Loss function****

A loss function is used to optimize the parameter values in a neural network model. Loss functions map a set of

values for the network onto a scalar value that indicates how well those parameters accomplish the task the network is intended

****Optimizer****

Optimizers are algorithms or methods used to change the attributes of your neural network such as weights and lea

order to reduce the losses.

****Metrics****

A metric is a function that is used to judge the performance of your model. Metric functions are similar to loss function

the results from evaluating a metric are not used when training the model.

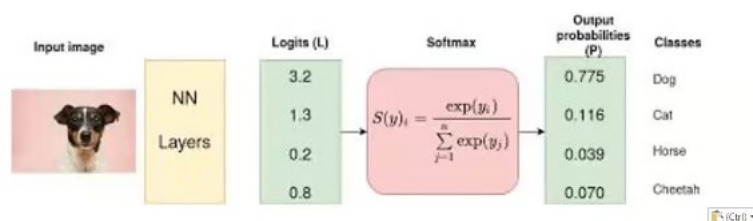
$$\text{entropy} = -p_{\text{yes}} * \log p_{\text{yes}} - p_{\text{no}} * \log p_{\text{no}}$$

$$\text{cross entropy} = -p_{\text{yes}} * \log p_{\text{yes}} - (1 - p_{\text{yes}}) * \log(1 - p_{\text{yes}})$$

Generally, from output layers values pass through activation function this will provide probabilities

Originally Dog Predicated as Rabbit

Ground Truth(Original)	Predicted	loss = Deviation Cross entropy	Output
Dog : 1	DOG: 0.8	0.2	DOG
Type equation here.	Cat: 0.1	0.9	
	Rabbit: 0.1	0.9	



Gradient Descent with Momentum → global and local minima

Adaptive Gradient Descent

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(X_train, y_train, epochs=10, batch_size=64)
```

Given:

- Training data size = 60,000 images
- Batch size = 64
- Epochs = 10

What happens:

- In **each batch**, the model processes **64 images**.
- In **each epoch**, the model goes through **938 batches** (since $\lceil 60000/64 \rceil = 938$).
- So, in **each epoch**, the model processes **all 60,000 images once** — just not all at once, but in chunks of 64.

Over 10 epochs:

- The model processes the entire dataset **10 times**.
- That means a total of **600,000 image passes** (60,000 images × 10 epochs).

23-2:

Convolution Neural Networks(CNN):

Is used to find the object detection problem

Imagine that we have a image 1000x1000 shape, if it is a color image

For input layer $1000 \times 1000 \times 3$ Neurons require = 3m neurons required

One hidden layer: 1000 neurons

So total parameters for the first step itself = $3M \times 1000 + 1000 = 3B$ parameters

3B parameters at starting only(one hidden layer)

So using of traditional neural networks is not a good idea to process the images

Because it requires more computation power, curse of dimensionality

Multiplication of two signal gives one more signal

Convolution main idea:

In order to pass entire pixel data, we can pass useful pixel data which contain more information which means we want to detect edges or important features or important patterns from the original image by multiplying with another image

This another filter image = filter or kernel

Convolution means, multiplying original image with a filter or kernel to get important patterns of the image

Filter is : weights matrix

1	1	1	0	0
1	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

1	0	1
0	1	0
1	0	1

1	1	1	0	0
(1)	(0)	(1)		
1	1	1	1	0
(0)	(1)	(0)		
0	0	1	1	1
(1)	(0)	(1)		
0	0	1	1	0
0	1	1	0	0

$$1*1+1*0+1*1+0*0+1*1+1*0+0*1+0*0+1*1=4$$

1	1	1	0	0
	(1)	(0)	(1)	
1	1	1	1	0
	(0)	(1)	(0)	
0	0	1	1	1

	(1)	(0)	(1)	
0	0	1	1	0
0	1	1	0	0

Input multiply with filter

Multiply all the values

Again, move the filter size and then down and multiply for each iteration

First slide to right = 4,3,4

second slide to right = 2,3,4

Third slide to right = 2,3,4

We will get feature maps with the above calculated values

Here 9 weights is enough

irrespective of image size, in cnn the input weights depends only on filter size

original image size = 5×5 filter size = 3×3 feature map = 3×3

original image size = 6×6 filter size = 3×3 feature map = 4×4

original image size = $N \times N$ filter size = $F \times F$ feature map = $(N-F+1) \times (N-F+1)$

Draw Back:

As filter size increases, the result of feature map size decreases

Which means result image has less pixel values, training on less values always not give the best performance.

Convolving filter with edges of image happens one time only, we might loss important patters

The resulting image going to shrink as filter size increases, and then edges important information not captured.

Padding:

To avoid the draw back will add an extra layer is called padding

Will add zeros on extra layer

Striding:

These 3 are hyperparameters

Pooling:

From the feature map we can extract an important feature, this is called pooling

Max pooling

Average pooling

Assume that your feature map size is 8×8

Pool size is 2×2

Original image convolve with filter == feature map

In this process padding stride no of filters

Pooling on feature map → final feature map

This process is called 1conv layer

Final feature map = image $2 \times$ filter

In this process padding stride no of filters

Pooling on feature map → final feature map

2conv layer

Finally, will get one image → flatten + hidden + output layer

26-2: practice [check out the notebook]

Parameters = $(\text{kernel_height} \times \text{kernel_width} \times \text{input_channels} + 1) \times \text{number_of_filters}$

In DL saving model means saving the weights

Save model

Transfer learning

We will not develop the model from scratch

Means we will read the weight file of the google model

After CNN then came LeNet, AlexNet, VGG16, VGG32, Model Net, ResNet-50, RCNN, MASK RCNN, FAST RCNN, YOLO

TRANSFER LEARNING

27-2: no class

28-2 read and writing of image [practice]

Reading videos

Affine method, `getRotationMatrix2D()`

29-2:[practice]

When you are opening video

Two values

1. Boolean value: the frame is available or not

2. Frame

Image = one frame

You don't know how many frames: while loop

Face detection Haar cascade Transfer learning

15 62 33 33 → how much you are travelling

15, 62, 15+33, 62+33 (x,y), (x+h, y+w)

1-3:

[Practice]

OCR: Optical character recog

Tesseract 3rd party

12-3:

PCA (Principal Component Analysis) is a **dimensionality reduction** technique commonly used in data analysis and machine learning. Its main purpose is to **simplify** a dataset by reducing the number of variables (features) while retaining most of the original information (variance).

Core Idea

PCA transforms the original variables into a new set of **uncorrelated** variables called **principal components**. These components are **ordered by the amount of variance** they capture from the data:

- **1st principal component** captures the most variance.
- **2nd principal component** captures the next most, and so on.
- Each new component is a linear combination of the original features

Is a dimension reduction technique dimension means data columns

Suppose a data set has 200 columns are there, if you want to use all the columns but that columns less than 200(best solution is pca)

Consider there are two columns are related to each other

For example age and income

As age increase income also increase

The data vary along age, along income are almost same

Which indicates, if you want choose one column, from two columns

We are not able to select, because the data varies across each columns is significant

Which variable you want to choose age or income

One more problem is age and income are dependent each other

In ML we have property: column should be independent each other

We cant choose one column out of two columns

The columns are dependent each other

Eigen values and Eigen Vector:

You have a **matrix** that transforms vectors — like rotating, stretching, or squishing them.

Most vectors **change direction** when multiplied by a matrix.

But... **some special vectors don't change direction** — they just get **stretched** or **shrunk**.

◆ **Those special vectors are called eigenvectors.**

The amount they're stretched or shrunk is called the **eigenvalue**.

If any matrix is multiplied how much can we stretch its value is given by lambda

$$\text{Matrix} \times \text{Eigenvector} = \text{Eigenvalue} \times \text{Same Eigenvector}$$

$$A\vec{v} = \lambda\vec{v}$$

$$A = \text{matrix}$$

$$\vec{v} = \text{eigenvector}$$

$$\lambda = \text{eigenvalue}$$

Eigen value will response for How much the data should vary

Eigen vector is responsible for which direction the data should vary

Steps need to follow:

1. Calculate covariance matrix of the data:

Covariance gives the relation between variables

2. Eigen values and eigen vectors for covariance matrix
3. Assume that we have p dimensions cov matrix shape: p*p
4. Now we have p*p matrix is there
5. How many eigen value and how many eigen vectors

P = eigen values

P= eigen vectors

If orinally we have 3 dimensions data

Cov matrix 3*3

