Linear Regression Vs Logistic Regression.

* when dependent variable is contionus eg: 2·1, 3·6
  ↓
* Stright line is used to seperate the data.
* Dependent and indipendent variables is required to be linearly related
* indipendent variable could be correlated
  ↓

$Y = b_0 + x_1 b_1 + b_2 x_2 \cdots$

```
y_model = tf.multiply(W, X) + b
cost = tf.square(Y - y_model)
```

when dependent variable is binary.
eg: 0, 1, 2.
↓

some linear function of indipendent variables is used to predict

* no need of linear relation bw Dependent & indipendent Variables
* indipendent should not be correlated.
  ↓

$Y = b_0 + x_1 b_1 + x_2 b_2 \cdots$

$P = \frac{1}{1+e^{-Y}}$; $\ln\left(\frac{P}{1-P}\right)$

```
y_model = tf.Sigmoid(tf.matmul(X,W)+b)
cost = tf.reduce_mean(-tf.reduce_sum
       (Y*tf.log(Y_model)))
```

Soft max.

Same is the case with soft max, where as in logistic regression label will be 0 (or) 1 only 2 chances on the other hand in softmax we can have multiple based on index place of the output

eg $Y = [\underset{A}{[1,0,0]}, \underset{B}{[0,1,0]}, \underset{C}{[0,0,1]}]$ → 3 labels

```
y_model = tf.nn.softmax(tf.matmul(W, X)+b)
cost = tf.reduce_mean(-tf.reduce_sum(Y* tf.log(Y_model))
```

Obs: of these '3' models soft max is best fit for classification of data which is linearly seperable so that we can classify in numbers

Note: Data should be linearly seperable otherwise weights will be NAN.

K nearest neighbour (KNN)

It is based on similarity measure eg: distance functions

Knn has been used in statistical estimation and pattern recognition.

distance functions:

Euclidean $\sqrt{\sum_{i=1}^{k}(x_i - y_i)^2}$ Manhattan $\sum_{i=1}^{K} |x_i - y_i|$

How to choose K value:

Generally K is more means more precise but no guarente else go for crossvalidation method generall K = (3-10) better K = odd to avoid tie

Drawback

distance measure from direct dataset might be a problem if the data is mixture of numerical and categorical values.

eg data is having annual income in dollars and age in years then income have higher influence on distance calculated

Sol: Solution is to standardize the data set.

Standardization.

$$X_i = \frac{X - \text{Min}}{\text{Max} - \text{min}}$$

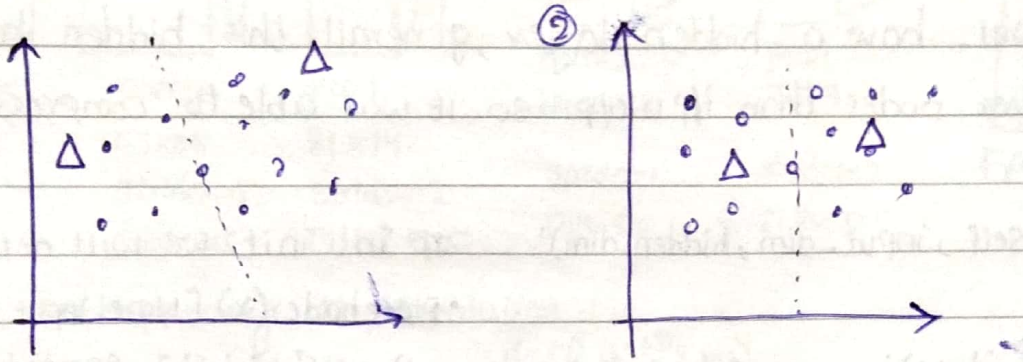
Note: In KNN there will be no weights because because for every test case we are calculating distance blw all train data and placed based on that distance

```
distance = tf.reduce_sum(tf.abs(tf.add(xtr, tf.negative(xte))), reduction_indices=1)
pred = tf.arg_min(distance, 0)
for i in range(len(xte)):
    nn_index = sess.run(pred, feed_dict={xtr: Xtr, xte: xte})
    print('Test', i, 'prediction:', np.argmax(Ytr[nn_index]),
          'True class:', np.argmax(Yte[i]))
```

Kmeans clustering

Clustering comes under unsupervised learning where data is mapped as clusters (groups), where this clusterin (grouping) is done in a procedure.

① ②



- In the first stage random positions of clusters are taken and every point is compared in terms of distance to the clusters based on least distance it is "mapped" to it
- In second stage clusters are "recentered" in sucha way that it will be in optimal distance from all points mapped to it

This process will repeat till nolonger clusters are recentered.

```
kmeans = KMeans(inputs=X, num_clusters=k, distance_metric='cosine', use_mini_batch=True)
training_graph = kmeans.training_graph()
(all_scores, cluster_idx, scores, cluster_centers_intialized, cluster_centers_var,
 init_op, train_op) = training_graph[0]
 cluster_idx = cluster_idx[0]
 avg_distance = tf.reduce_mean(scores)
Start session run init, init_op to initialize variables and random clusters
    for i in range(1, num_steps+1):
        _, d, idx = sess.run([train_op, avg_distance, cluster_idx], feed_dict={X: x_batch})
```

Training completed then • Assigning label to ~~there~~ closest each centroid.
• Sampel to their closest centroid given by idx • Assign most frequent label to centroid.
• Now test and predict accuracy if test data is available.

Auto encoder:

Auto encoder is like a .mp3 file which is of less in size but still given similiar sound o/p, while this o/p of encoded is not exactly same as the one which is before encoding.

Auto encoders will have a hidden layer, generall this hidden layer will be having fewer nodes than i/p or o/p so it was able to compress

```
class Autoencoder:
    def __init__(self, input_dim, hidden_dim):      # In init we will define
        pass                                          • placeholder(x) [None, input_dim]
    def train(self, data):                            • tf_name_scope('encode') → W[i,h], b[h], encoded → tanh
        pass                                          • tf_name_scope('decode') → W, b, decoded → tanh.
    def test(self, data):                             • self.loss → square error   • train_op → RMS optimizer
        pass
# In train the autoencoder:                  for i in range(epoch):        → len(data)
• start session & run variables →              for j in range(num_samples):
• Last save the session for reuse.          _, _ = sess.run([self.loss, self.train_op], feed_dict={self.x: [data[j]]})
# In testing we restore train session.
• and we will run encode & decode feeding test data.
```

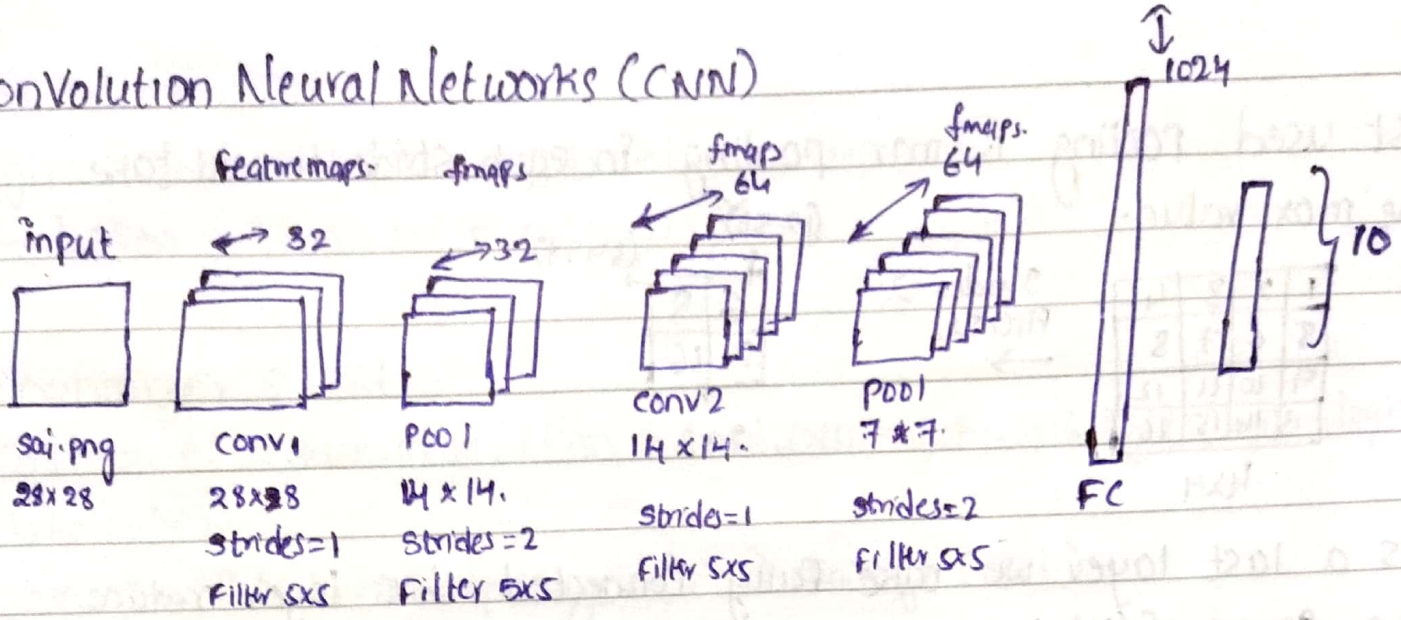
In this method we can compress data and re produce it also

eg i/p → [8, 4, 6, 2] → 4 values
compressed → [0.78238308] → 1 value.
reconstructed [6.87, 2.748, 6.25, 2.23]

It is better to make it batchwise training

Autoencoders are unsupervised Neural Networks that use ML to do compression for us. The aim of autoencoders is to learn a compressed, distributed representation for given data for purpose of dimensionality reduction.

Convolution Neural Networks (CNN)



What is happening at convolution:

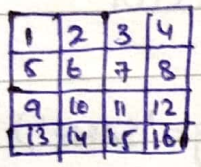
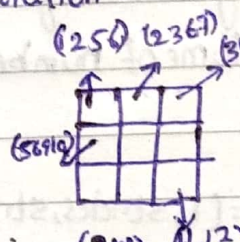


image of (4x4)

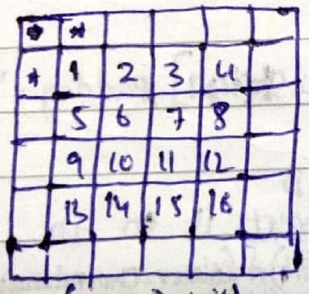
using filter (2x2) strides=1 without padding.



→ Filter size is specified in weights and strides are specified in conv2d and max pool

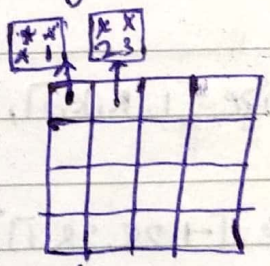
so in with out padding size will reduce.

If Padding='SAME' then it will altomatically adjust the size in different way



(4x4) with added duto to same padding.

using filter (2x2) strides=1 Padding - Same



If Padding is "same"

strides is → 1 → size will not change

strides is → 2 → size will divid by 2.

After convolution of image it will be called as feature maps.

size of filters and no of filters are specified in weights [5, 5, 1, 32]

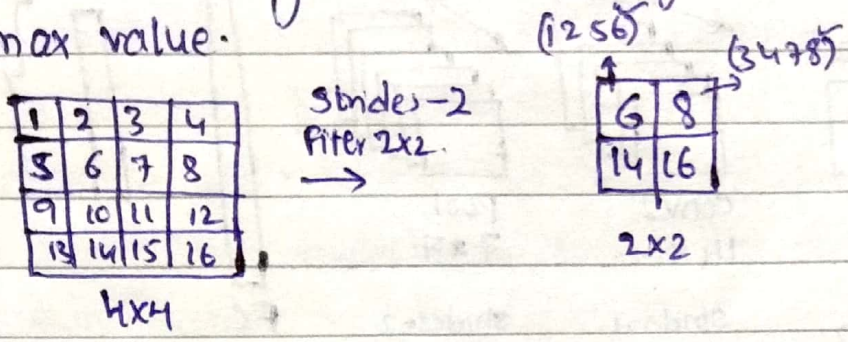
5,5 → Filter ← (5x5)

1 → Dimage

32 → 32 filters creates 32 fmaps

After convolution we usally do pooling to reduce dimensionality. this enables to reduce the dimensionality, no of parameters & overfitting & time taken for process

most used pooling is max pooling in each stride it will take the max value.



As a last layer we take fully connected this layer reduce the fmaps of (64) into one big 1-D array called flattening
Flattening is arranging 3D volume of numbers into 1D vector

- conv2d:

```
x = tf.nn.conv2d(x, W, strides=[1, strides, strides, 1], padding='SAME')
x = tf.nn.bias_add(x, b)
return tf.nn.relu(x)
```

- maxpool2d:

```
tf.nn.max_pool(x, ksize=[1, k, k, 1], strides=[1, k, k, 1], padding='SAME')
```

- conv_net:

```
x = tf.reshape(x, shape=[-1, 28, 28, 1])  # input x is 1D now we convert it to 4D
conv1 = conv2d(x, weights['wc1'], biases['bc1'])   (-1)*(width)*(height)*(colour_channel_no.)
conv1 = maxpool(conv1, k=2)                        ↳ '-1' is standard to calculate image shape so that total size remains same.
conv2 = conv2d(conv1, weights['wc2'], biases['bc2'])   '1' is standard feature from numpy.
conv2 = conv2d(conv2, k=2)
fc1 = tf.reshape(conv2, [-1, weights['wd1'].get_shape.as_list[0]])   same as (7*7*64)
fc1 = tf.add(tf.matmul(fc1, weights['wd1']), biases['bd1'])
fc1 = tf.nn.relu(fc1) # Apply dropout if need fc1 = tf.nn.dropout(fc1, dropout)
fc1 = tf.add(tf.matmul(fc1, weights['out']), biases['out'])
return out
```

```
#model creation
logits = conv_net(X, weights, biases, keep_prob)
prediction = tf.nn.softmax(logits).

#optimizer & cost
cost = tf.reduce_mean(tf.nn.softmat_cross_entropy_with_logits(logits=logits,
labels=Y))
train_op= tf.train.AdamOptimizer(0.01).minimize(cost)
```

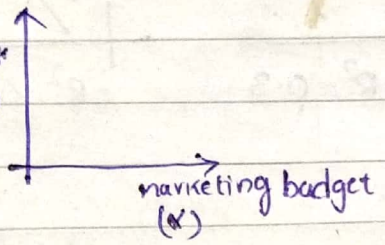
Types of ML

① Regression → continuous Variable
② Classification → categorical Variable
③ clustering → No labels only on properties / feature

Supervised < Regression / classification unsupervised → clustering.

• Regression:

① Simple linear regression.
② multiple linear regression.



X → independent Var (Predictor)
Y → Dependent Var (target var)

Optimal line → best fit line

Residual → error → difference b/w actual and prediction.

$e_i = y_i - y_{pred}$ → $e_1^2 + e_2^2 + e_3^2 + \cdots$ = RSS (Residual sum of squares)

$RSS = e_1^2 + e_2^2 + e_3^2 + \cdots e_n^2$

$e_i = y_i - y_{pred}$.

$y_{pred} = \beta_0 + \beta_1 x_i$ ∴ $RSS = \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i)^2$

• Our linear model ⇒ $y = mx + c$.

To determine how good is this model by 2 < R^2 → R-square) (coefficient of determination)
RSE → Residual Standard error

Problem with RSS:

RSS value changes if our data is changed its units / dimen

X	Y
1 kg	1000
2K	1500 Rs

→

X	Y
1000 gms	1·2 $
2000 gms	2·3 $

g & Rs → (grams & $)

change in units bring change in RSS value to overcome that we go for TSS (Total Sum of square) method

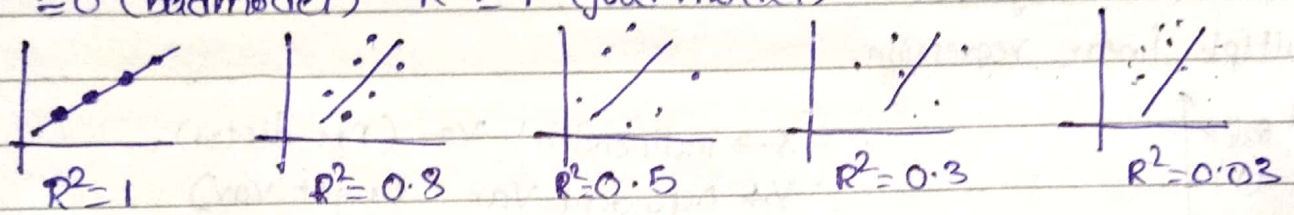
TSS → Sum of the error for your data points from the mean of your response: $TSS = \sum_{i=1}^{n} (y_{pred} - \bar{y}_i)^2$ (worst model)

R^2 (coefficent of determination) → $R^2 = 1 - \frac{RSS}{TSS}$

RSS is already least value, now we want TSS to be least so R^2 vary from 0 to 1.

if RSS ≅ TSS that means RSS equal to worst model ∴ $R^2 = 0$ (Model bad)

$R^2 \cong 0$ (bad model) $R^2 \cong 1$ (good model)



case study:

① Took a dataset for a T.V selling company

② seperated x & y, splited and using Sklearn Train & Predicted.

③ ploted residual error graph for index vs (y-ypred) → Residual error

④ calculated from sklearn.metrics import mean_square_error, r2_score.

mse = mean_square_error(y_test, y_Pred)

r_squared = r2_score(y_test, y_Pred)

- mse (mean square error) means (ie 7.957) → our model was unable to match 7.957% value only thats nearly 8% value it can't match (good)
- r_square (R2_Score) → ie. 0.59 ⇒ 59% (acutal 60% of R^2 is good model)

Residual square error (RSE)

$RSE = \sqrt{\frac{RSS}{df}}$ → degree of freedom

$df = n - 2$

n ⇒ no of data points

RSE → has same advantages & disadvantages like RSS so it good to go with R^2 to judge model

Most used activation functions:

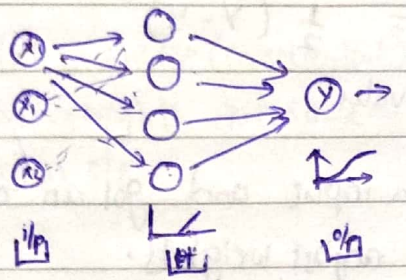
① Threshold function. $\phi(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$

② Sigmoid function. $\phi(x) = \frac{1}{1+e^{-x}}$ (mostly in o/p layer)

③ Rectifier function $\phi(x) = \max(x, 0)$

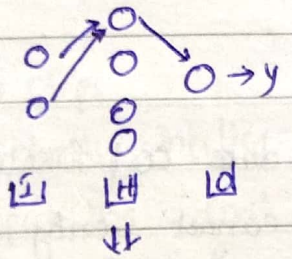
④ Hyperbolic Tangent (tanh) $\phi(w) = \frac{1-e^{-2x}}{1+e^{-2x}}$

exercise ①



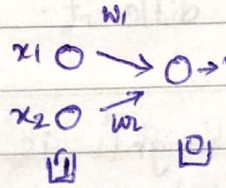
Most cases we use Rectifier in the "hidden" layer and sigmoid at "o/p"

Why do we need a hidden layer?



have plenty of combinations (Y)

So each hidden layer can be specified to particular case.

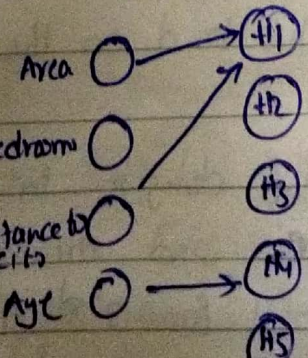


$\Rightarrow y = w_1 x_1 + w_2 x_2$.

have only one combination.

* If there is "No hidden" layer then obviously o/p will be a linear combination of input [Area, Bedrooms, distance to city, Age] → [price]
x_1 x_2 x_3 x_4 Y

* where as if it has hidden layes it can have other specific combinations. (i.e)



[In "H1" case it only considers "Area" & "distance" as major parameters which is general case]

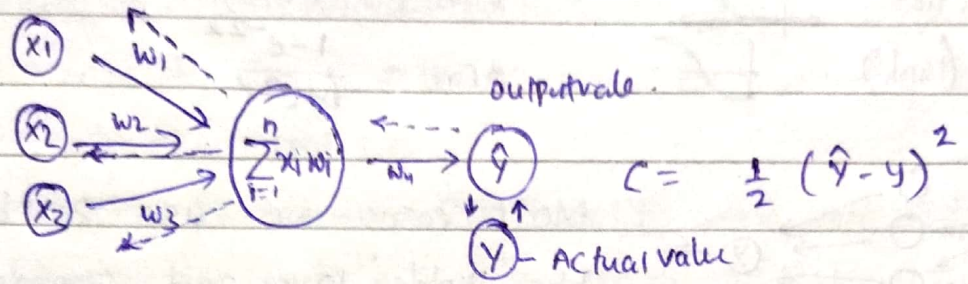
(Y)

[In "H4" case it only consider Age as a major parameter in this case for "vintage" building it happens]

∴ H1 & H4 the parameters are different though both are for house pricing ∴ hidden layers are useful

How do Neural networks learn?

While teaching neural network we don't teach it with hard coded rules instead we let it to figure out the rules.

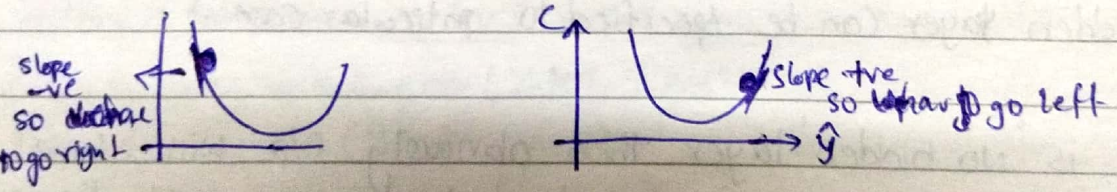


once all the dataset is given as an input and got an o/p now you compare with actual value and adjust weights.

"epoch" → inputting all the input's is called as one epoch. (usually) but for batch training it is different.

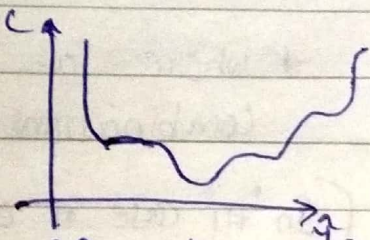
How weights are adjusted?

The primary reason for adjusting weights is to reduce cost function. [best optimal value for the cost function]. Simply we cannot verify all the possible values of 'w' to know optimal values. So we opted "Gradient descent method"



Stochastic Gradient discent ?

This topic comes when cost fun is not proper curvy there is every change it might land on the local maxima or minima.



in stochastic GD method instead of doing it in batches (for all the inputs) at a time we take only inputs once and calculate ŷ and find cost fun and do adjust weights and repeat it for 2nd inputs.

Batch GD
↓
Whole batch at once
↓
(may lead to local minima)
↓
it has to load all inputs at once
so its slow.
↓
it is a deterministic (consistent)
same starting weights give same
accuracy in o/p

Stochastic GD
↓
only one i/p data time.
↓
(only leads to global minima)
↓
it is faster & lighter.
(cause it don't load all the inputs)
↓
it is stochastic (inconsistent)
(it has random choose in rows & weights so not possible to get same results every time).

Note: There is also mini batch gradient descent method.

Trainng steps

① Randomly intilise the weights to small numbers close to 0 but not '0'.
② Input the first observation of your dataset in the input layer. each feature in one input node.
③ Forward Propagation from Left to right neurons are activated in a way that the impact of each neuron's activation is limited by weights propagate the activations until getting the predicted results
④ compare predicted result to actual results and generate error.
⑤ Back propagation Right to left. Update the weight according to how much they are responsible for the error the learning rate decides how much we update weights.
⑥ Repeat steps 1 to 5 and update the weights after each observation (Reinforcement Learning) (or) Repeat 1 to 5 steps and update weight only after batch of observation (Batch learning).
⑦ When all the dataset get passed through it it is called an Epoch we do more & more epochs.

Pd.read_csv('....') → To read a csv file it gives a DF as o/p

le = LabelEncoder() → (Categorical to → 1,2,3)

oh = OneHotEncoder() → (1,2,3 → (000)(001))

sc = StandardScaler() → Standardize/normalize

} sklearn.preprocessing

in data frame we use ⇒ data.iloc[:, 1:3] → to access rows & cols

data frame cannot be processes directly so we convert to array form DF using data.values → this convert it to array format

sklearn.model_selection → train_test_split (X, Y, test_size = 0.2, random_state = 1)

even for onehot encoder we should use oh.fit_transform(X).toarray()

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

classifier = Sequential()

classifier.add(Dense( )) input_dim = , output_dim = $\left(\frac{\text{input} + \text{outputs}}{2}\right)$, init = "uniform", activation = 'relu'

from second layer we dont need to specify the input_dim.

for final layer we use sigmoid for 1 o/p Softmax for multiple o/p.

classifier.Compile (optimizer = 'adam', loss = 'binary_crossentropy' / 'categorical_crossentropy', metrics = ['accuracy'])

```
import keras
from keras.models import Sequential
from keras.layers import Dense
from sklearn.metrics import accuracy, confusion_matrix
classifier = Sequential()
classifier.add(Dense(units = 6, input_dim = 11, activation = 'relu', kernel_initializer = 'uniform'))
  "        "       "      "       x          "              "            )
  "        "       "      " = 1   x          "      = 'sigmoid',   "   )
classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['Accuracy'])
Y_pred = classifier.predict(X_test)
print accuracy_score(Y_test, Y_pred)
```

Scanned with CamScanner

Linear Regression.

* Blw Linearly Seperable variables only
* To predict continus variable
* Straight line.
* forecasting sales etc
* cost = least square error
* $Y = b_0 + b_1 x + e$

Logistic Regression.

* Blw linearly seperable & non-seperable.
* To predict categorical Variable
* S-curve
* classification etc
* cost (binary_cross entropy)
* $\log\left(\frac{Y}{1-Y}\right) = C + B_1X_1 + B_2X_2$.

Bias-Variance tradeoff: (it is an error due to bias and variance).
generall we try to acheive "low bias" and "low variance" for a model.
(to get consistent predictions on different datasets) (To avoid underfit and over fit)

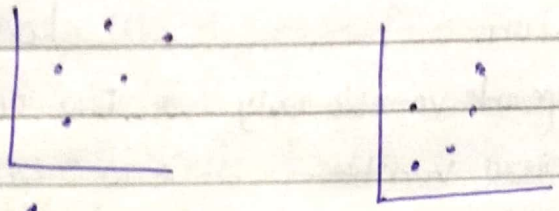
machile learning errors → Bias error → expected error in the predection of model.
→ Variance error →
(expectation of error in prediction ofa model)

Bias errors: are due to our wrong assumption in a model. for example we apply linear regression to a dataset assuming that both variables have a linear relationship blw them, but they minght not have linear relation ship necessarly, this leads to high bias (ie. model is underfitting).

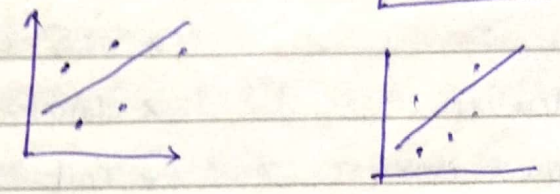
variance error: measure of variability in the results given by model when the dataset is changed.

innermost circle is our target
① model with low-Bias, low-variance have predrictions near to target.
② low-bias high variance says the model predrictions vary rapidly. ✗ (model is not consist
③ Low variance high bias says model is consistent but far from target
④ high bias high variance ✗

example take 2 data sets

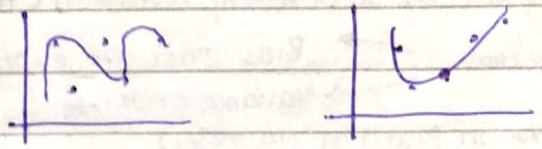


Apply LR model on both



in this case we can say the model is having bias error because for both the data sets its prediction is almost similar but actual o/p's of both are different therefore model is consitent but having bias error (3) ~~high~~ bias ~~low~~ variance

Apply Polynomial LR model



in this cas the model is having variance error because for both datasets the Prediction is very different ∴ we can say that this model is inconsistent (high-variance)

How to reduce Bias-variance tradeoff?

- Dimensionality Reduction
- Regularzation in Linear models / ANN
- Using mixture model and Ensemble learning
- optimal value of K in KNN.

To check Bias-variance tradeoff we use "K-fold cross validation" (Split data set into more parts and selects random sets from them to test and train).

* Transfer learning is a technique used to reduce the development of model from scratch to used the model & it's weights by adding our desired layers to it.

Dropout is the solution for overfitting.

classifier.add(Dropout()) ⇒ p = 0.1 (10%) → add after every layer in which you want to fix some neurons

Parameter tuning to improve best model:

Parameters such as 'no of hyperparameters', 'batch size', 'Learning rate'

classifier = KerasClassifier(build_fun = build_classifier) ← in keras.wrappers.scikit_learn
↓ from the definition of classifier (same as above)

```
parameters = {'batch_size' : [25, 32],
              'epochs' : [100, 500],
              'optimizer' : ['adam', 'rmsprop']}
```

```
grid_search = GridSearch.fit(X_train, y_train)
grid_search.best_params_
best_accuracy.best_score_
```

Cross validation to check the model with different test sets:

```
classifier = KerasClassifier(build_fn = build_classifier, batch_size = 10, epochs = 100)
```
← keras.wrappers.scikit_learn

```
accuracy = cross_val_score(estimator = classifier, X = X_train, y = y_train, cv = 10, n_jobs = -1)
accuracy.mean(), accuracy.std().
```

Standardisation

$$x_{stand} = \frac{x - \text{mean}(x)}{\text{standard deviation}(x)}$$

Normalisation

$$x_{norm} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

```
import numpy as np
np.random.seed(123)
from keras.models import Sequental
from keras.layers import Dense, Flatten, Accuracy, Dropout, Convolution2D, MaxPooling2D
from keras.utils import np_utils
from keras.datasets import mnist
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
#preprocessing input data
X_train = X_train.reshape(X_train.shape[0], 1, 28, 28)
X_test = X_test.reshape(X_test.shape[0], 1, 28, 28)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
Y_train = np_utils.to_categorical(Y_train, 10)
Y_test = np_utils.to_categorical(Y_test, 10)
# model architecture.
model = Sequential()
model.add(Convolution2D(32, 3, 3, activation='relu', input_shape=(1, 28, 28)))
model.add(Convolution2D(32, 3, 3, activation='relu'))
model.add(MaxPooling2D(Pool_size = (2,2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model(Dense(10, activation='softmax'))
model.compile(optimizer='adams', loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(X_train, Y_train, batch_size=32, nb_epoch=10)
model.predict(X_test) (■) model.eval(X_test, Y_test, verbose=0)
```

binary_cross entropy <= 0.1
categorical cross entropy <= 1/2 <= 3
mse → for regression problem "continuous"

```
from keras.models import Sequential
from keras.layers import Dense, Flatten, Dropout, Conv2D, MaxPool2D
model = Sequential()
model.add(Conv2D(64, input_shape=(120,120,3), kernel_size=(3,3), strides=(1,
                 activation='relu'))
model.add(Conv2D(32, input_shape, kernel_size=(3,3), strides=(1,1), activation='relu'))
model.add(MaxPool2D(strides=2, pool_size=(2,2))
model.add(Flatten())
model.add(Dense(units=16, activation='relu', kernel_initializer='uniform'))
model.add(  ,  =8  ,  ,  ))
model.add(  ,  =2  ,  'softmax'  ,  )
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(X_train, Y_train, batch_size=10, epochs=10)
model.batch_test(X_test, Y_test)
```

(convolution) → (maxpooling) → (Flattening) → (full connection)

Max pooling: (Spatial invariance) means features what it is looking for need not to be in the same position or size (ratio) they can vary a bit. There are maxpooling, mean pooling, sumpooling, average pooling - -

ReLU layer: (rectilinear operation) Relu increases non linearity because basically images are non linear (at edges [borders] during convolution it might turn into linear so that relu is need. with dg

Convolution: (Yann Lecun)

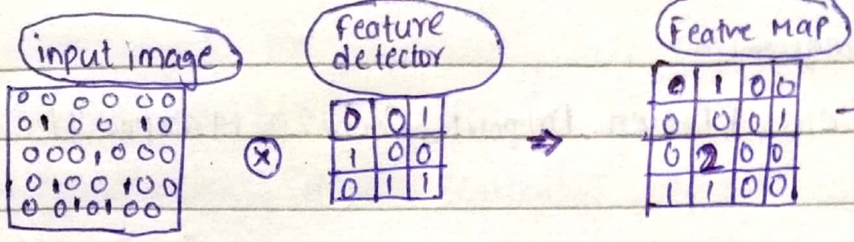
Flattening: is rearranging the pooled feature maps in single array that goes into N.N

input layer:- Flatten pooled feat maps are fed to input layer.

Fully connected: (hidden layers) (must be fully connected) to input layer

Output layer: each for each class → connected to fully connected layer.

machine will be trying to learn these (feature dectors)/(Filters)/(Kernel)



/(activation map)

→ Based on (Kernal_size)/(filter size) and stride's feature map size vary.

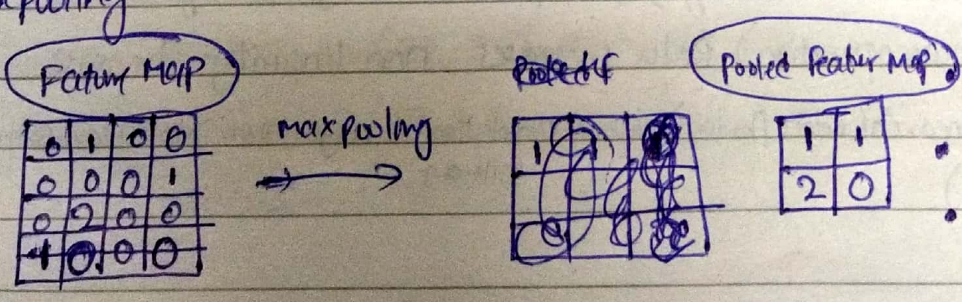
We took an image and we took one random feature detector/(Filter) and we convoluted both then we got convolved features called "feature map"

* In this process based on stride size & kernel size we can reduce feature map size

* what exactly is a feature detector?

- feature detector is a pattern/shape which we are looking for in a image,
- feature map will have highest values if the specific pattern of the image and feature detector match.
- along the training we are making it to learn these feature dectors in terms of weights i.e (64, 32 → in conv2D (64), → This can look for 64 different patterns (feature dectors) in a image.
  kernel.size=(3,3) → ∴ in this layer we are have 64 (3x3) Filters
  ∴ (64x3x3) weights ← learnd

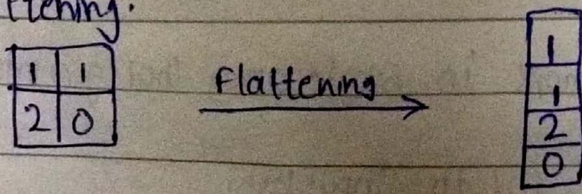
max pooling



size got reduced

- improving spatial invariance
- reducing no of features to look for.

Flattening.



Scanned with CamScanner

Softmax & Cross entropy:

Softmax takes the values and normalize them so that their sum is equal to '1' (Softmax is a logistic function $f_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}$

Cross entropy $\rightarrow L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \Rightarrow H(p,q) = -\sum_x p(x) \log q(x)$

linear → mse → 3.14

Softmax → categorical-crossentropy → [0 0 1 0]

relu/sigmoid → binary-crossentropy → 0

image agumentation keras.preprocessing.image import ImageDataGenerator.
generates images in different scales from the given

Supervised	Artifical Neural Networks	used for regression & classification
	Convolution " "	used for computer vision
	Recurring " "	used for Time series Analysis
Unsupervised	Self organizing maps	used for feature Detection
	Deep Boltzmann Machines	used for recomendation systems
	Auto Encoders	used for recomendation systems

cerebrum , cerebellum , Brainstem

↓

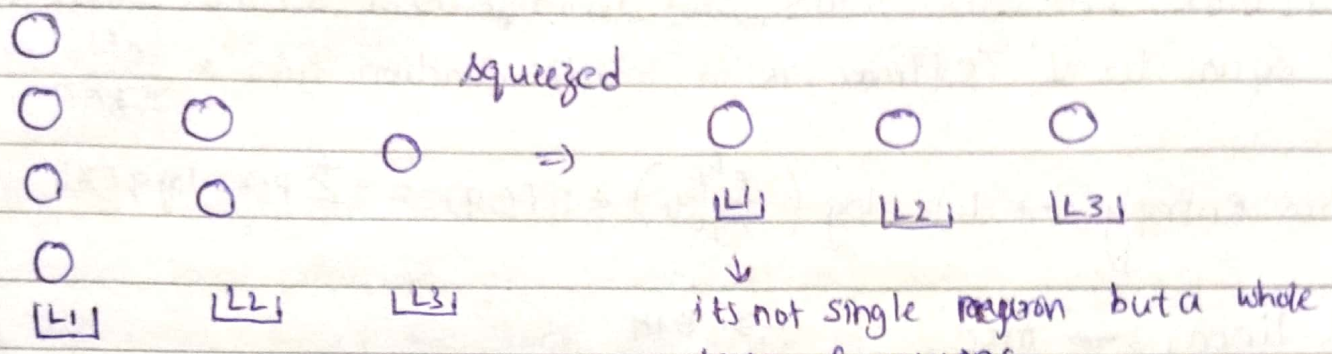
Frontal lobe ← (Temperory memory) RNN (Personality, behaviour, working memory)

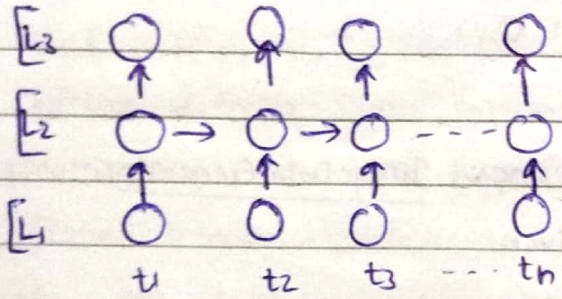
parietal lobe ← (sensation & perception to create a 3D world in mind)

Temporal lobe ← weights of ANN (responsible for long term memory)

occipital lobe ← CNN (Vision)

Recurring neural networ (RNN):





⇒ every neuron represents a layer of neurons, and t1, t2, t3 are different time's.

In RNN while calculating weights (Adjusting weights) we will face a problem called vanishing / exploding gradient

Solutions:

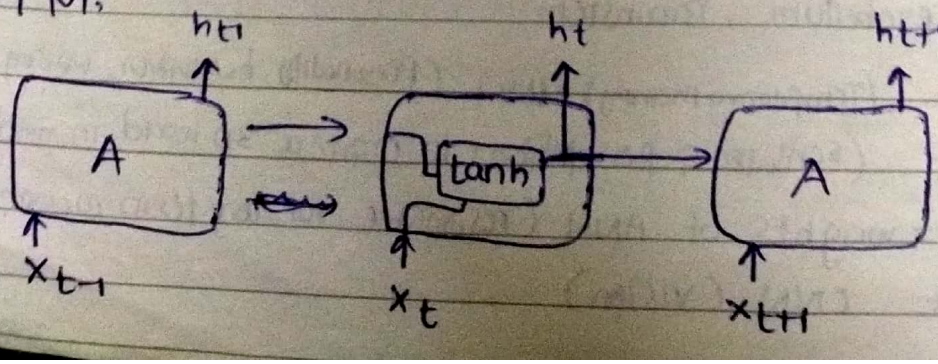
- Truncated Back propagation.
- Penalties
- gradient clipping

} gradient exploding ($w_{rec} > 1$)

- weight initilization
- Echo State Networks
- long short term memory networks

} vanishing gradient ($w_{rec} < 1$)

LSTM:



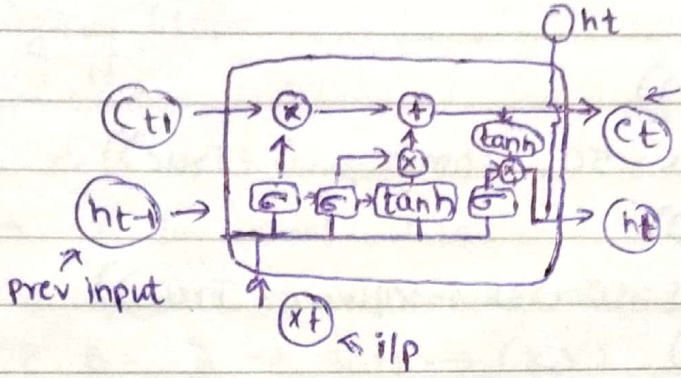
•bag

6905 Punc hefi

•Pcap file [1]

192.168.115.140

LSTM RNN ⇒



← memory pipeline.

every line in this architecure is a vector →

concatenate ⟶ copy ⟶ ○ Pointwise operation. (like valve)

implementation of RNN:

```
from sklearn.preprocessing import MinMaxScaler
sc = MinMaxScaler(feature_range = (0,1))
training_set_scaled = sc.fit_transform(training_set)
```

} Normalized.

#creating a data structure with 60 timesteps and 1 output.
(it will consider previous 60 days stockprice and predict the next).

```
X_train = []
y_train = []
for i in range(60, len(training_set)):
    X_train.append(training_set_scaled[i-60:i, 0])
    y_train.append(training_set_scaled[i, 0])
X_train, y_train = np.array(X_train), np.array(y_train)
X_train = np.reshape(X_train, (X_train.shape[0], 60, 1))
```

← shape expected by RNN

(X_train.shape[0] ↑ no of i/p rows; 60 ↑ i/p cols)

Building the RNN

```
from keras.models import Sequential
from keras.layers import Dense, LSTM, Dropout
regressor = Sequential()
```

← we call it regression as we are predicting.

```
regressor.add(LSTM(units = 50, return_sequences = True, input_shape = (60,1)))
```

(units ↑ no of units of LSTM cells; return_sequences ↑ Return sequences LSTM layers T/F; input_shape ↑ input shape)

```
regressor.add(Dropout(0.2))
regressor.add(LSTM(units=50, returnSequence=True))
regressor.add(Dropout(0.2))
regressor.add(LSTM(units=50, return_sequence=True))
regressor.add(Dropout(0.2))
regressor.add(LSTM(units=50, return_sequence=False))
regressor.add(Dropout(0.2))
regressor.add(Dense(units=1)
regressor.compile(optimizer='adam' loss='mse')
regressor.fit(X_train, Y_train, batch_size=32, nods_epochs=100)
# Testing to predict 2017 Jan 1 data. (we have 20 days
  of testing data from Jan 1 to Jan 20) to predict a day we need
  previous 60 days data so that, along with test data 2017 Jan(1-20)
  we also take data in 2016 Dec&sep. To do this we will add
  all the original data from train&test and then we will Normalize from that

dataset_total = pd.concat((dataset_test['open'], dataset_test['open']), axis=0)
inputs = dataset_total[len(dataset_total) - len(dataset_test) - 60:].values.
inputs = inputs.reshape[-1, 1]
inputs = sc.transform(inputs) → must not fit (if we fit prev values might change)
X_test = []
for i in range(60, len(inputs)):
    X_test.append(inputs[i-60:i, 0])
X_test = np.array(X_test)
X_test = np.reshape(X_test, (X_test.shape[0], 60, 1))
pred=regressor.predict(X_test)
Pred = sc.inverse_transform(pred) → To convert into actual values
# Now plot the actual and predicted values to see the predictions accuracy
```

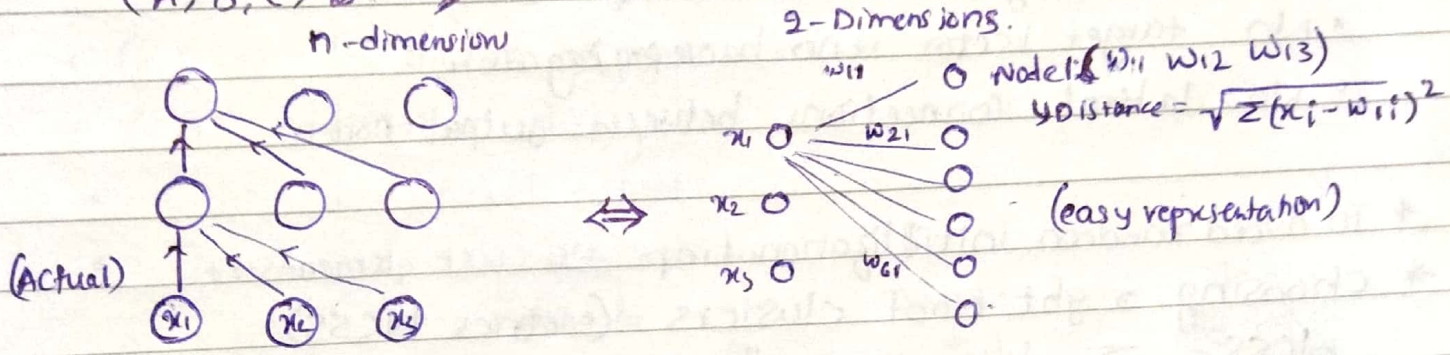
Self organizing Maps:

This comes under unsupervised learning.

we provide data with 'n' no of columns it constrain that entire data into '2' dimensional data.

SOM's are usefull to reduce the dimensionality of dataset.

$(A, B, C, D \cdots) \rightarrow$ SOM $\rightarrow (X, Y)$



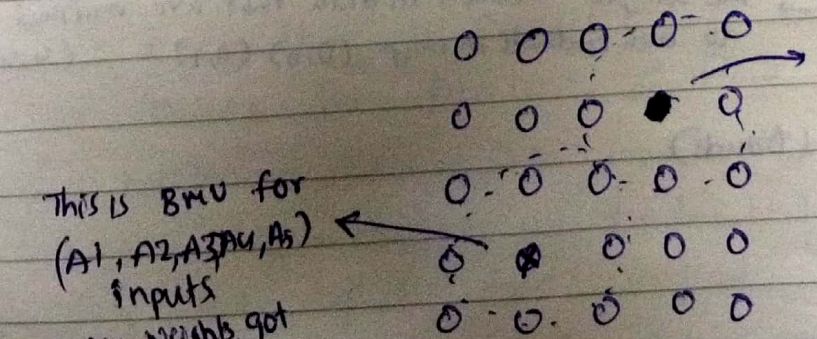
SOM are bit different for ANN.

In ANN weights are multiplied by inputs and activation function decides weather to fire the node or not, where as in SOM there is no activation fun weights are characteristics of node itself.

- Here weights acts as ~~inputs to~~ cordinates in input space.

(First random weights are given to entire network)

- we calculate euclidian distance = $\sqrt{\Sigma (x_i - w_{(node)i})^2}$, and we will choose the least distance node as Best matching unit (BMU)
- Now weights of (BMU) node will change → (in a way to decrease euclidan dist)
- The a whole range of node's around (BMU) node's weights got updated.



and with in certain range all the node weights got updated.

- Based on more iteration the region around BMU getting updated will shrink.
  - SOM retain topology of the input set
  - SOM reveal correlations that are not easily identified.
  - SOM classify data without supervision.
  - NO target vector → no back propagation.
  - NO lateral connections between output nodes.

* To avoid random initialization trap we use kmeans++
* choosing right no of clusters ⇒ (Metrics WCSS)

$$WCSS = \sum_{P_i \text{ in cluster } C_i} \text{distance}(P_i, C_i)^2$$

The method we use is called elbow method.

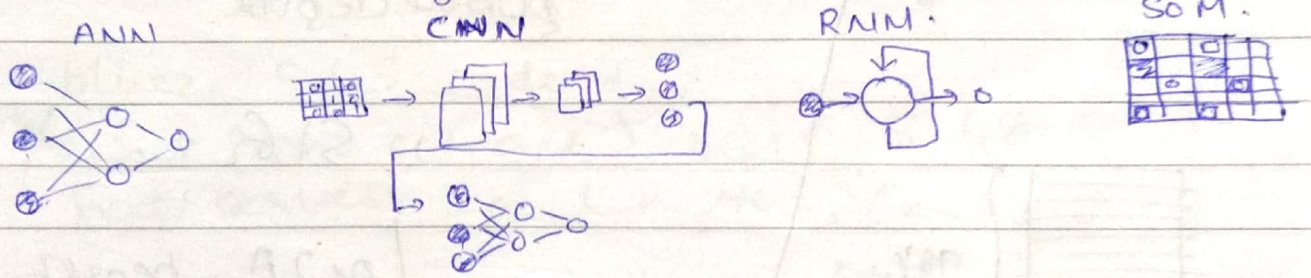
```
from sklearn.preprocessing import MinMaxScaler; from minisom import MiniSom
som = MiniSom(x=5, y=5, input_len = data.shape[1])
som.random_weights_init(data)
som.train_random(data, num_iteration = 1000)
from pylab import bone, pcolor, colorbar, plot, show
bone()  → white chart
pcolor(som.distance_map().T) → Colour map of (5x5) based on distance of each unit
colorbar()  → color bar

mappings = som.win_map(data) → This will give members in data that are mapped to each cordinate in map (0,0) (0,1) ... (4,4)
frauds = mappings[(2,2)]
frauds = ms.inverse_transform(frauds)
```

Boltzmann Machines.

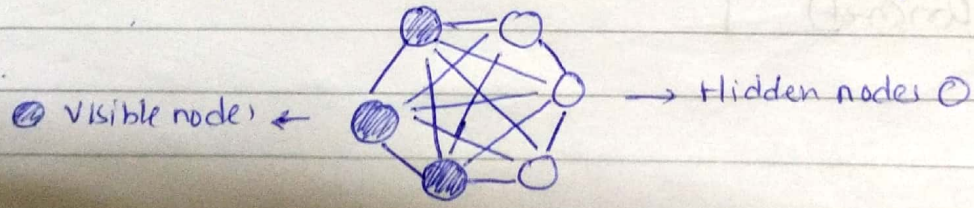
Boltzmann Machines can be seen in two different point of views

1. An energy based model (EBM)
2. A probabilistic Graphic model. (

- (EBM) energy based model
- (RBM) Restricted boltzmann machines
- (CD) contrastive divergence.
- (DBN) deep Belief Networks
- (DBM) deep boltzmann machines



All the above (ANN, CNN, RNN, SOM) have a direction in which they grow where as boltzmann machines dosent have any directions



- Boltzmann machines have only Input/visible & hidden layers but "no output layer"
- every thing is connected to every thing.
- There is no direction of flow

[in boltzmann machine even input [visible nodes are connected to each other because boltzmann machines are fundamentally different from other machines They dont expect data instead they generate data therefore boltzmann machines are generative machines].

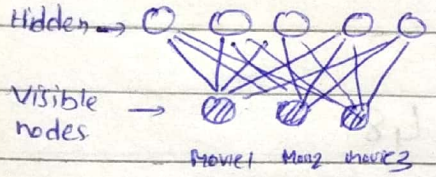
$$P_j = \frac{e^{-\varepsilon_j / kT}}{\sum_{i=1}^{M} e^{-\varepsilon_j / kT}}$$ → Boltzmann equation.

P → Probability is inversly proportional to energy ε.

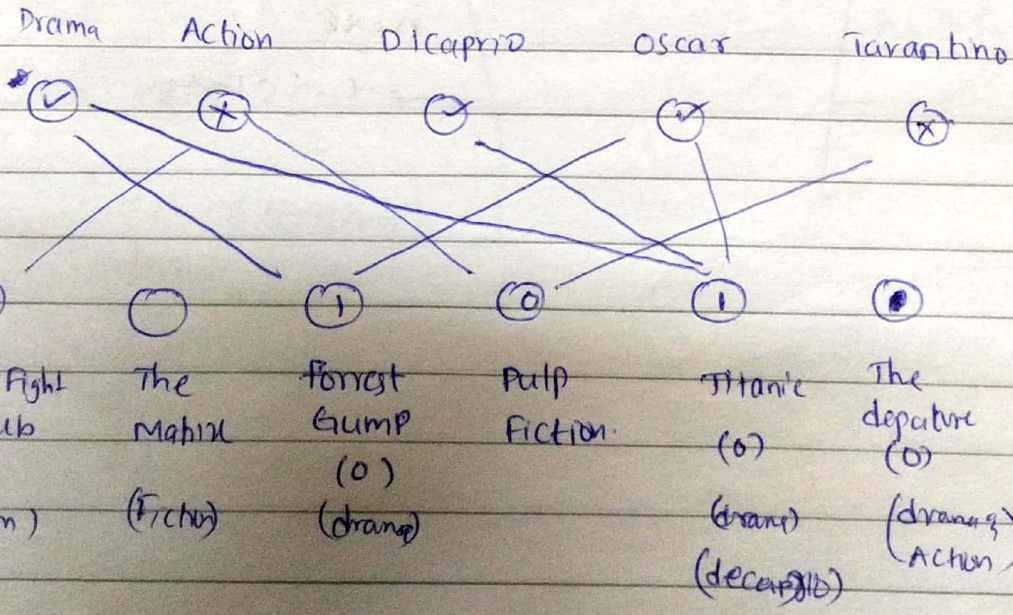
$$E(v,h) = -\sum_i a_i v_i - \sum_j b_j h_j - \sum_i \sum_j v_i w_{ij} h_j$$ → energy state

$$P(v,h) = \frac{1}{Z} e^{-E(v,h)}$$ → probability of each engy state

Due to implementation difficulties in Fully Connected Boltzmann machines they decided to implement (RBM) restricted boltzmann machines



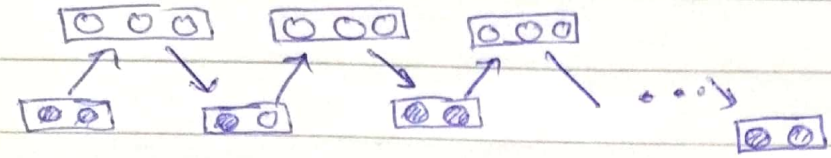
→ Movie Recomended system.



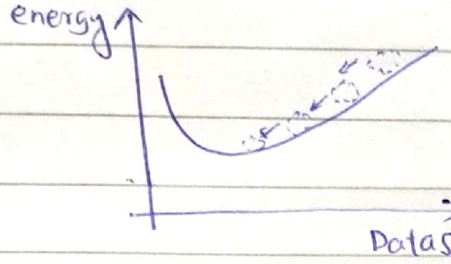
Contrastive Divergence :

Feeding input values to network i.e we will generate input values again (which are slighly vary from actual) again those generated values are feeded into networks and regenerated and again. it is repeated

Contrast Divergence 1



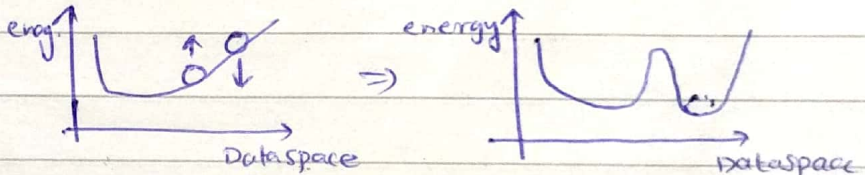
(unit inputs modify no further)



$$\frac{\partial \log P(v^0)}{\partial w_{ij}} = \langle v_i^0 h_j^0 \rangle - \langle v_i^\infty h_j^\infty \rangle$$

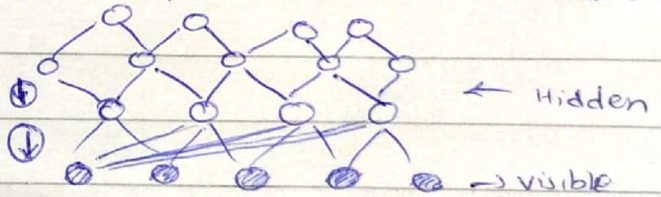
a system always tries to end up in lowest energy state weights are actually deciding the shape of the curve

[Basically this method of generating inputs and feeding back to system take long time so geofry hinton found a shortcut which is contrastive divergence.



Deep Belief Network (DBN)

Stack of (RBM) on one another



implementation of Boltzmann Machines: (RBM)

```
import numpy as np
import pandas as pd
import torch                    # Torch have tensors like Tensorflow
import torch.nn as nn
import torch.nn.parallel
import torch.optim as optim
import torch.utils.data
from torch.autograd import Variable
```