UNIT-II

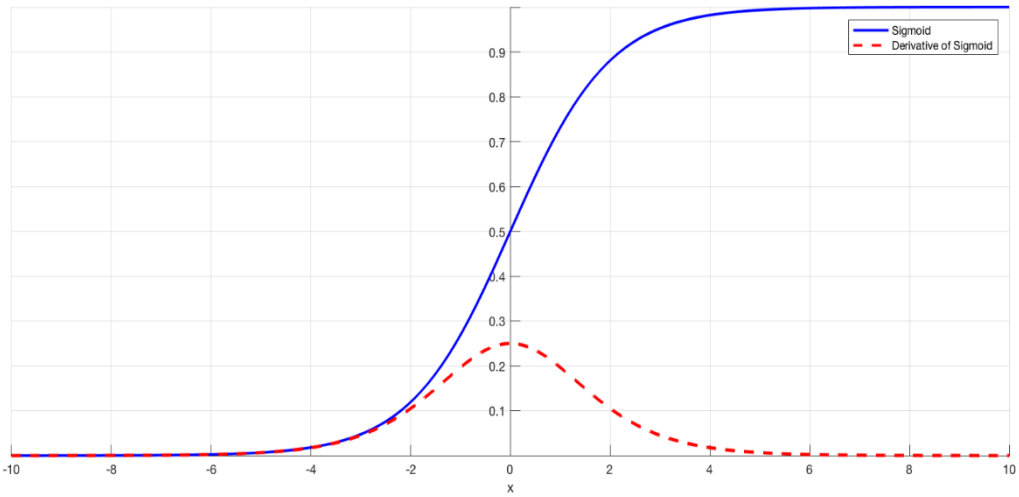
Deep Networks: Vanishing Gradients and Exploding Gradients problem. Hyper Parameters – layer size, Magnitudes, regularization, activation functions, weight initialization strategies, mini-batch size, vectorization. Building blocks of deep networks- Feed Forward multi-layer neural networks, Restricted Boltzmann Machines, Auto encoders. Unsupervised Pretrained Networks- Auto encoders -Sparse Autoencoders, Denoising Autoencoders, Contractive Autoencoders and Variational auto encoders, Deep Belief Networks (DBNs), Generative Adversarial Networks (GANs). Training a multi-layer artificial neural network and deep neural network.

Vanishing Gradients Problem

The Problem: As more layers using certain activation functions are added to neural networks, the gradients of the loss function approaches zero, making the network hard to train.

Reason: Use of Sigmoid activation function.

The activation function sigmoid, squishes a large input space into a small input space between **0 and 1.** Therefore, a large change in the input of the sigmoid function will cause a small change in the output. Hence, the derivative becomes small.



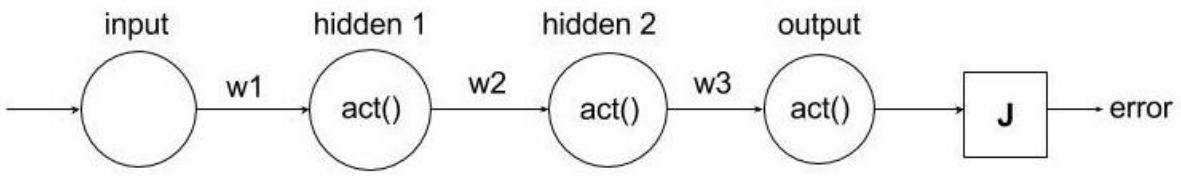
With the sigmoid function, when the inputs of the sigmoid function become larger or smaller the derivative becomes close to zero, the range is from 0 to 0.25.

Gradients of neural networks are found using backpropagation. i.e. Backpropagation finds the derivatives of the network by moving layer by layer from the final layer to the initial one. Using chain rule, the derivatives of each layer are multiplied down the network (from

the final layer to the initial) to compute the derivatives of the initial layers.

When n hidden layers use an activation like the sigmoid function, n small derivatives are multiplied together. Thus, the gradient decreases exponentially as we propagate down to the initial layers.

The structure of a neural network and backprop and their implications on the size of gradients.



Recall this general structure for a simple, univariate neural network. Each neuron or "activity" is derived from the previous: it is the previous activity multiplied by some weight and then fed through an activation function. The input, of course, is the notable exception. The error box J at the end returns the aggregate error of our system. We then perform backpropagation to modify the weights through gradient descent such that the output of J is minimized.

To calculate the derivative to the first weight, we used the chain rule to "backpropagate" like so:

$$\frac{\partial error}{\partial w1} = \frac{\partial error}{\partial output} * \frac{\partial output}{\partial hidden2} * \frac{\partial hidden2}{\partial hidden1} * \frac{\partial hidden1}{\partial w1}$$

If we see individual derivatives

$$\frac{\partial output}{\partial hidden2} * \frac{\partial hidden2}{\partial hidden1}$$

With regards to the first derivative — since the output is the activation of the 2nd hidden unit, and we are using the sigmoid function as our activation function, then the derivative of the output is going to contain the derivative of the sigmoid function. In specific, the resulting expression will be:

$$z_1 = hidden2 * w3$$

$$\frac{\partial output}{\partial hidden2} = \frac{\partial Sigmoid(z_1)}{\partial z_1} w3$$

The same applies for the second:

$$z_2 = hidden1 * w2$$

$$\frac{\partial hidden2}{\partial hidden1} = \frac{\partial Sigmoid(z_2)}{\partial z_2} w2$$

In both cases, the derivative contains the derivative of the sigmoid function. Now, let's put those together:

$$\frac{\partial output}{\partial hidden2} \frac{\partial hidden2}{\partial hidden1} = \frac{\partial Sigmoid(z_1)}{\partial z_1} w3 * \frac{\partial Sigmoid(z_2)}{\partial z_2} w2$$

Recall that the derivative of the sigmoid function outputs values between 0 and 1/4. By multiplying these two derivatives together, we are multiplying two values in the range (0, 1/4]. Any two numbers between 0 and 1 multiplied with each other will simply result in a smaller value. For example, 1/3 × 1/3 is 1/9.

During the weight updating in backpropagation, the new weight and old weight will be closure to same.

A small gradient means that the weights and biases of the initial layers will not be updated effectively with each training session. Since these initial layers are often crucial to recognizing the core elements of the input data, it can lead to overall inaccuracy of the whole network.

with deep neural nets, the vanishing gradient problem becomes a major concern.

The solution to minimize the Vanishing Gradients Problem

1. **Use of other activation functions, such as ReLU or its family, which doesn't cause a small derivative.**
2. **Using appropriate weight initialization techniques.**

Exploding Gradients problem

An error gradient is the direction and magnitude calculated during the training of a neural network that is used to update the network weights in the right direction and by the right amount.

In deep networks or recurrent neural networks, error gradients can accumulate during an update and result in very large gradients. These in turn result in large updates to the network weights, and in turn, an unstable network. At an extreme, the values of weights can become so large as to overflow and result in NaN values.

The explosion occurs through exponential growth by repeatedly multiplying gradients through the network layers that have values larger than 1.0.

- **Exploding gradients can make learning unstable.**
- **the exploding gradients problem refers to the large increase in the norm of the gradient during training. Such events are due to the explosion of the long term components.**

The exploding Gradients can be identified as

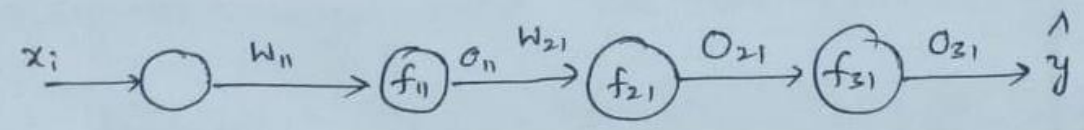
- **The model weights quickly become very large during training.**
- **The model weights go to NaN values during training.**
- **The error gradient values are consistently above 1.0 for each node and layer during training.**

The exploding Gradients Problem can be resolved using

- **Re-Design the Network Model to have few layers**
- **Use Long Short-Term Memory Networks**
- **Use Gradient Clipping to limit the size of gradients during the training of network.**
- **Appropriate weight initialization technique.**
- **Regularization with L1 or L2**

The higher weights in neural network leads to exploding gradients Problem.

Consider the following part of neural network



Using chain rule of back propagation

$$\frac{\partial L}{W_{11}} = \frac{\partial O_{31}}{\partial O_{21}} \cdot \frac{\partial O_{21}}{\partial O_{11}} \cdot \frac{\partial O_{11}}{\partial W_{11}}$$

Let the $\frac{\partial O_{21}}{\partial O_{11}}$ is computed as

$$\frac{\partial O_{21}}{\partial O_{11}} = \frac{\partial \psi(z)}{\partial z} \cdot \frac{\partial z}{\partial O_{11}}$$

Here $z = W_{21} \cdot O_{11} + \text{bias}$

$O_{21} = \psi(z)$.

Activation function

if Activation function is segmoid.

$\frac{\partial \psi(z)}{\partial z}$ is in the range 0 to 0.25.

Assume W_{21} weight as $\Rightarrow$ 500, a large value.

$$\frac{\partial O_{21}}{\partial O_{11}} = 0.2 * \frac{\partial (W_{21} O_{11} + \text{bias})}{\partial O_{11}}$$

$$= 0.2 \times W_{21} = 0.2 \times 500 = \underline{100}$$

lly the products in chain are computed.

$$\text{Eg: } \frac{\partial L}{\partial W_{11}} = 100 \times 100 \times 50 = 500000$$

P.T.O.

Using this bigger value the new weight computed as

$$W_{11new} = W_{11old} - \eta \cdot \frac{\partial L}{\partial W_{11old}} \quad [\eta = 0.9]$$

Assume $W_{11old} = 300$

$$W_{11new} = [300 - 0.9 \times 50000] = -ve \text{ value in high range.}$$

i.e. The new weights have very large negative value.

This is called exploding gradient problem.

Because of exploding gradients, the training never converge or reach global minima.

Hyper Parameters

A hyperparameter as any configuration setting that is free to be chosen by the user that might affect performance.
Hyperparameters are into the following categories:

1. **Layer size**
2. **Magnitude (momentum, learning rate)**
3. **Regularization (dropout, drop connect, L1, L2)**
4. **Activations (and activation function families)**
5. **Weight initialization strategy / technique**
6. **Loss functions**
7. **Mini-batch an Epoch size**
8. **Normalization scheme for input data (vectorization)**

- **Some hyperparameters apply only some of the time.**
- **Changing a specific hyperparameter might affect the best settings for other hyperparameters.**
- **Some hyperparameters are incompatible with one another (e.g., Adagrad + momentum).**

1. Layer Size:

- **Layer size is defined by the number of neurons in a given layer. Input and output layers are relatively easy to figure out because they correspond directly to how our modeling problem handles input and output.**
- **For the input layer, this will match up to the number of features in the input vector.**
- **For the output layer, this will either be a single output neuron or a number of neurons matching the number of classes we are trying to predict.**
- **Deciding on neuron counts for each hidden layer is where hyperparameter tuning becomes a challenge. We can use an**

arbitrary number of neurons to define a layer and there are no rules about how big or small this number.

- The complexity of a problem we can model is directly correlated to how many neurons are in the hidden layers of our networks.
- This might push you to begin with a large number of neurons from the start but these neurons come with a cost.
- Depending on the deep network architecture, the connection schema between layers can vary.
- However, the weights on the connections, are the parameters we must train. As we include more parameters in our model, we increase the amount of effort needed to train the network. Large parameter counts can lead to long training times and models that struggle to find convergence.

2. Magnitude Hyperparameters

Hyperparameters in the magnitude group involve the gradient, step size, and momentum.

Learning rate:

- The learning rate in machine learning is how fast we change the parameter vector as we move through search space.

- If the learning rate becomes too high, we can move toward our goal faster (least amount of error for the function being evaluated), but we might also take a step so large that we shoot right past the best answer to the problem, as well. Another side effect of learning rates that are large is that we run the risk of having unstable training that does not converge over time.

- **If we make our learning rate too small, it might take a lot longer than we'd like for our training process to complete. A low learning rate can make our learning algorithm inefficient.**

- **Learning rates are tricky because they end up being specific to the dataset and even to other hyperparameters. This creates a lot of overhead for finding the right setting for hyperparameters.**

- **AdaGrad, RMSProp, AdaDelta and ADAM provides the right learning rates and gradient during the training.**

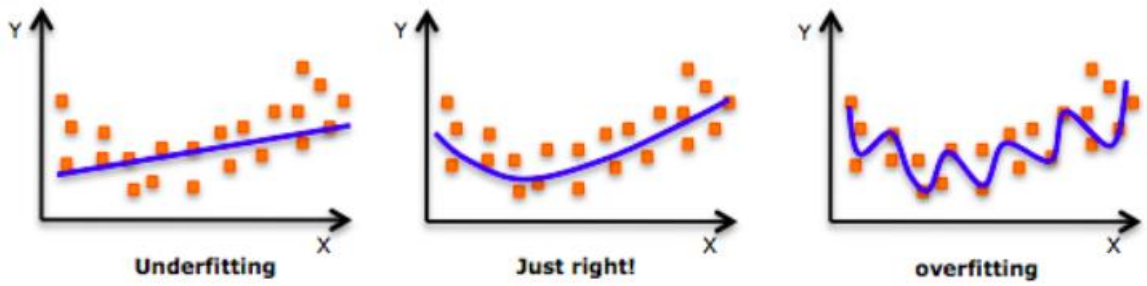
Momentum:

- **The training can be speed up by increasing momentum. Momentum is a factor between 0.0 and 1.0 that is applied to the change rate of the weights over time. Typically, the value for momentum between 0.9 and 0.99.**

3. Regularization

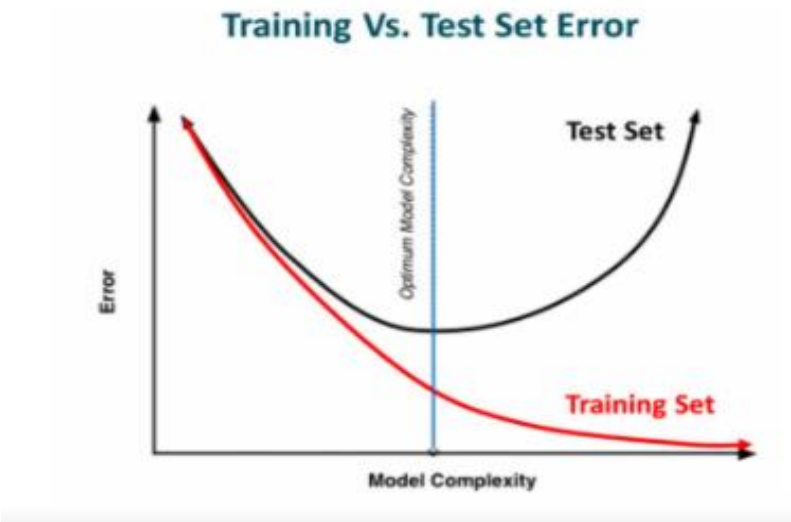
- **Regularization is a measure taken against overfitting.**

Overfitting occurs when a model describes the training set but cannot generalize well over new inputs. i.e A situation where the model performed exceptionally well on train data but was not able to predict test data. Overfitted models have no predictive capacity for data that they haven't seen. Avoiding overfitting can improve model's performance.



In Overfitting the model tries to learn too well the details and the noise from the training data, which ultimately results in poor performance on the unseen

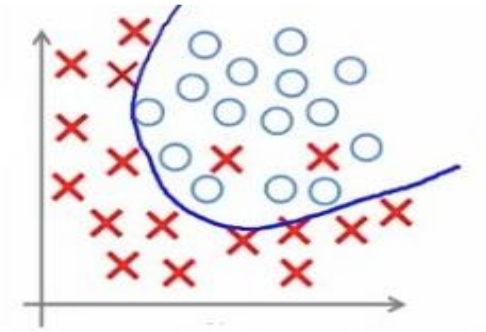
In Overfitting the complexity of the model increases such that the training error reduces but the testing error doesn't. i.e.



"The best way to build a neural network model: Cause it to overfit, and then regularize it to death."

How Regularization helps to reduce overfitting?

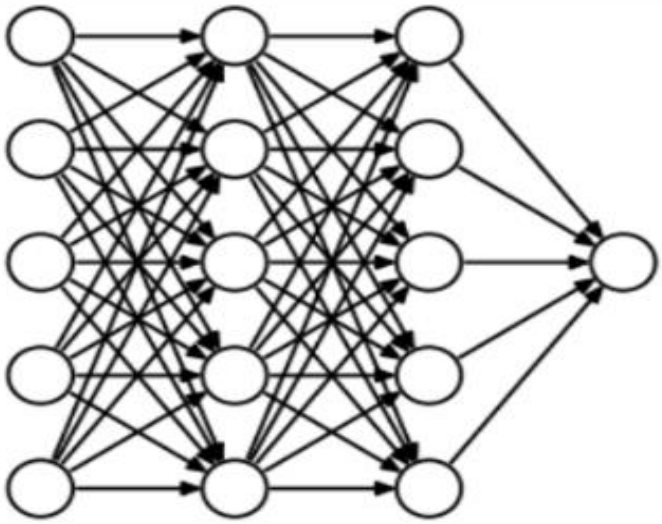
- **Regularization is a technique which makes slight modifications to the learning algorithm such that the model generalizes better. This in turn improves the model's performance on the unseen data as well.**
- **Regularization for hyperparameters helps modify the gradient so that it doesn't step in directions that lead it to overfit.**
- **In deep learning, regularization penalizes the weight matrices of the nodes. We need to optimize the value of regularization coefficient in order to obtain a well-fitted model as below.**



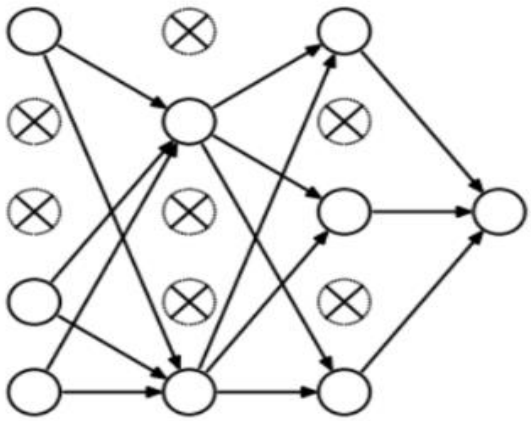
Appropriate-fitting

The different Regularization techniques are

1. **Dropout:**
   - **This is the one of the technique, produces very good results and is consequently the most frequently used regularization technique in the field of deep learning.**
   - **Dropout is a mechanism used to improve the training of neural networks by omitting a hidden unit.**
   - **It also speeds training. Dropout is driven by randomly dropping a neuron so that it will not contribute to the forward pass and backpropagation.**
   - **In the below example neural network**



Using dropout, at every iteration, it randomly selects some nodes and removes them along with all of their incoming and outgoing connections as



So each iteration has a different set of nodes and this results in a different set of outputs.

- Dropout is usually preferred when we have a large neural network structure in order to introduce more randomness.

  **The probability of choosing how many nodes should be dropped is the hyperparameter of the dropout function. **

2. **Drop Connect:**
   DropConnect does the same thing as Dropout, but instead of choosing a hidden unit, it mutes the connection between two neurons.

3. **L1 and L2 Regularizations:**
   - The penalty methods L1 and L2 are a way of preventing the neural network parameter space from getting too big in one direction. They make large weights smaller.
   - L1 and L2 are the most common types of regularization. These update the cost function by adding another term known as the regularization term.

   Cost function = Loss or Error + Regularization term

Due to the addition of this regularization term, the values of weight matrices decrease because it assumes that a neural network with smaller weight matrices leads to simpler models. Therefore, it will also reduce overfitting to quite an extent.
The regularization term differs in L1 and L2.

In L1:

$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum \|w\|$$

- **lambda is the regularization parameter. It is the hyperparameter whose value is optimized for better results.**
- **L1 regularization multiplies the absolute value of weights. This function drives many weights to zero while allowing a few to grow large, making it easier to interpret the weights.**
- **The weights may be reduced to zero here. Hence, it is very useful when we are trying to compress our model.**

In L2:

$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum \|w\|^2$$

- **L2 regularization is also known as weight decay as it forces the weights to decay towards zero (but not exactly zero).**
- **L2 improves generalization, smooths the output of the model as input changes, and helps the network ignore weights it does not use.**
- **As lambda increases, the total W decreases, thus increase the linearity in the model.**

4. Data Augmentation:

The simplest way to reduce overfitting is to increase the size of the training data. In Machine learning the labeled data cannot be increased as it is too costly. But in Deep Learning the training data can be increased.

Eg: When we train with images. The methods to increase the size of the training data is – rotating the image, flipping, scaling, shifting, etc.

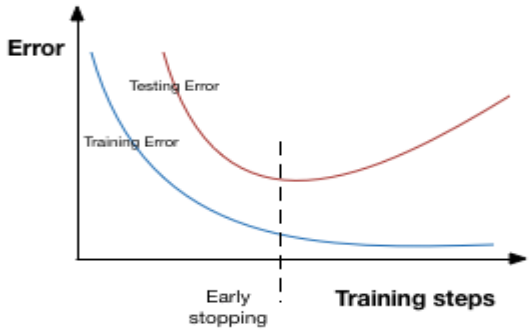
In the below image, some transformation has been done on the handwritten digit's dataset.

- **The technique of increasing training data is known as data augmentation.**
- **This usually provides in improving the accuracy of the model.**
- **It can be considered as a mandatory trick in order to improve our predictions.**

5. Early stopping:

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

we will stop training at the dotted line since after that our model will start overfitting on the training data as shown below.



4. Activation Functions

The activation functions are used in feed-forward neural networks to drive feature extraction.
Activation Functions allows the network to learn patterns in the data within a constrained space.

In the input layer, we want to pass on the raw input vector features so that in practice we don't express an activation function for the input layer.

The Hidden layers are concerned with extracting progressively higher-order features from the raw data. Depending on the architecture we're working with, we tend to use certain subsets of layer activation functions.
Hidden layer activation functions Commonly used are:

1. **Sigmoid**
2. **Tanh**
3. **Rectified linear unit (ReLU) (and its variants)**

A more continuous distribution of input data is generally best modeled with a ReLU activation function.
We use the tanh activation function (if the network isn't very deep) in the event that the ReLU did notachieve good results. issues with the network).
the sigmoid activation function falls out of favor for hidden layers.

In the Output Layer

1. **Output layer for regression: If we want to output a single real-valued number from our model, use a linear activation function.**

2. **Output layer for binary classification: In this case, we'd use a sigmoid output layer with a single neuron to give us a real value in the range of 0.0 to 1.0 (excluding those values) for the single class. This real-valued output is typically interpreted as a probability distribution.**

3. **Output layer for multiclass classification: If we have a multiclass modeling problem and only care about the best score across these classes, use a softmax output layer with an arg-max() function to get the highest score of all the classes. The softmax output layer gives us a probability distribution over all the classes.**

   If we want to get multiple classifications per output (e.g., person + car), we do not want softmax as an output layer. Instead, use the sigmoid output layer with n number of neurons, giving us a probability distribution (0.0 to 1.0) for every class independently.

5. Weight initialization Techniques

Weight initialization is an important consideration in the design of a neural network model. The nodes in neural networks are composed of parameters referred to as weights used to calculate a weighted sum of the inputs.

The most common problem with Deep Neural Networks is Vanishing and Exploding gradient descent. To solve these issues, one solution could be to initialize the parameters carefully.

One of the parameter is weight initialization to minimize the problem of Vanishing and Exploding gradient descent.

Its main objective is to prevent layer activation outputs from exploding or vanishing gradients during the forward propagation. If either of the problems occurs, loss gradients will either be too large or too small, and the network will take more time to converge if it is even able to do so at all.

If we initialized the weights correctly, then our objective i.e, optimization of loss function will be achieved in the least time otherwise converging to a minimum using gradient descent will be impossible.

The basic rules for assigning weights are

1. **Weights should not be too small or too big.**
2. **Weights should not be same. If so all the neurons learn same.**
3. **Weights should have good variance, so that each learn in a different way.**
4. **The weight initialization technique work based on dataset, activation functions and its features. Hence no specific method is chosen for best performance.**

The Different Weight Initialization Techniques are

1. **Zero Initialization**

   If we initialized all the weights with 0, then the derivative with respect to loss function is the same for every weight. Thus all weights have the same value in subsequent iterations. This makes hidden layers symmetric and this process continues for all the n iterations. Thus initialized weights with zero make your network no better than a linear model.

If we set biases to 0 will not create any problems as non-zero weights take care of breaking the symmetry and even if bias is 0, the values in every neuron will still be different.

2. **Random Initialization: – This technique tries to address the problems of zero initialization since it prevents neurons from learning the same features of their inputs since our goal is to make each neuron learn different functions of its input and this technique gives much better accuracy than zero initialization.**
   In general, it is used to break the symmetry. It is better to assign random values except 0 to weights.

   If the weights are too small then Vanishing Gradient Problem, thus converge slowly.
   If the weights are too large, exploding gradient problem, and take more number of iterations or epochs.

   To overcome these problems heuristics are used for weight initialization.

   The commonly used heuristics or weight initialization techniques are

   1. **Xavier / Glorot Distribution**
   The standard approach for initialization of the weights of neural network layers and nodes that use the Sigmoid or TanH activation function is called "glorot" or "xavier" initialization.
   Xavier Glorot, a research scientist.
   There are two versions of this weight initialization method, which we will refer to as "uniform xavier" and "normalized xavier."

The xavier initialization method is calculated as a random number with a uniform probability distribution (U) between the range -(1/sqrt(n)) and 1/sqrt(n), where n is the number of inputs to the node or also called fan-in.

$$\text{weight} = U\,[-(1/\text{sqrt}(n)),\ 1/\text{sqrt}(n)]$$

or

$$\text{weight} = U\,[-(2/\text{sqrt}(n)),\ 2/\text{sqrt}(n)]$$

In the below example assumes 10 inputs to a node, then calculates the lower and upper bounds of the range and calculates 1,000 initial weight values that could be used for the nodes in a layer or a network that uses the sigmoid or tanh activation function.

```
from math import sqrt
from numpy import mean
from numpy.random import rand
# number of nodes in the previous layer
n = 10
# calculate the range for the weights
lower, upper = -(1.0 / sqrt(n)), (1.0 / sqrt(n))
# generate random numbers
numbers = rand(1000)
# scale to the desired range
scaled = lower + numbers * (upper - lower)
# summarize
print(lower, upper)
print(scaled.min(), scaled.max())
print(scaled.mean(), scaled.std())
```

The bounds of the weight values are about -0.316 and 0.316.
-0.31622776601683794 0.31622776601683794
-0.3157663248679193 0.3160839282916222
0.006806069733149146 0.17777128902976705

2. Normalized Xavier Weight Initialization

   The normalized xavier initialization method is calculated as a random number with a uniform probability distribution (U) between the range -(sqrt(6)/sqrt(n + m)) and sqrt(6)/sqrt(n + m).

   Where n is the number of inputs to the node or fan-in (e.g. number of nodes in the previous layer) and m is the number of outputs from the layer or fan-out (e.g. number of nodes in the current layer).

   weight = U [-(sqrt(6)/sqrt(n + m)), sqrt(6)/sqrt(n + m)]

   The normalized xavier initialization method is calculated as a random number with a Gaussian probability distribution (G) with a mean of 0.0 and a standard deviation of sqrt(2/sqrt(n+m)), where n is the number of inputs to the node.

   weight = G (0.0, sqrt(2/sqrt(n+m))

   Used for the Sigmoid or TanH activation functions.

3. He init Distribution

The approach was developed specifically for nodes and layers that use ReLU activation, popular in the hidden layers of most multilayer neural network models.

The He Uniform distribution

weight = U [-(sqrt(6)/sqrt(n)), sqrt(6)/sqrt(n)]

The He Normal Distribution

The he initialization method is calculated as a random number with a Gaussian probability distribution (G) with a mean of 0.0 and a standard deviation of sqrt(2/n), where n is the number of inputs to the node.

weight = G (0.0, sqrt(2/n))

6. Loss functions

7. Settings for epochs and mini batch size.

- **With mini-batching, we send more than one input vector (a group or batch of vectors) to be trained in the learning system. This allows us to use hardware and resources more efficiently at the computer-architecture level.**
  The batch size is a number of samples processed before the model is updated. The size of a batch must be in between 1 to number of samples in the training dataset. The normal range is from 50 to 256 data points.

- **The number of epochs is the number of complete passes through the training dataset.**

  The number of epochs can be set to an integer value (1 to any number). We can run the algorithm for as long as you like and even stop it using other criteria besides a fixed number of epochs, such as a change (or lack of change) in model error over time.

They are both integer values and they are both hyperparameters for the learning algorithm, we must specify the batch size and number of epochs for a learning algorithm. We must try different values and see what works best for the problem.

8. Normalization scheme for input data (vectorization)

A method used to convert an image into vector, used in CNN.

Training a multi-layer artificial neural network

1. Classification

```
#Part-1: Data Preprocessing
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

#Uploading the Dataset file
from google.colab import files
uploaded = files.upload()

#loading the dataset
import pandas as pd
import io

dataset =
pd.read_csv(io.BytesIO(uploaded['Churn_Modelling.csv']))
print(dataset)

X = dataset.iloc[:, 3:13]
y = dataset.iloc[:, 13]

#crating dummy variables
geography=pd.get_dummies(X["Geography"],drop_first=True)
gender=pd.get_dummies(X['Gender'],drop_first=True)

#Concatenate the Dataframes
X=pd.concat([X,geography,gender],axis=1)

#Drop unnecesary columns
X=X.drop(['Geography','Gender'],axis=1)
```

```
#displaying column headings
for col in X.columns:
   print(col)
print(X)

#Splitting Dataset into Training Set and Test Set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
0.2, random_state = 0)

#feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
print(X_train)

#Part-2 ANN Model Construction
#import keras libraries and packages
import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LeakyReLU,PReLU,ELU
from keras.layers import Dropout

#Initializing ANN
classifier = Sequential() #empty neural network

#Input Layer and First Hidden Layer
classifier.add(Dense(units=6,kernel_initializer= 'he_uniform', activation=
'relu', input_dim = 11))
```

```
#Hidden Layer-2
classifier.add(Dense(units = 6, kernel_initializer= 'he_uniform', activation ='relu'))

#Activation Function -> output layer
classifier.add(Dense(units= 1, kernel_initializer = 'glorot_uniform', activation = 'sigmoid'))

#Compiling the Artificial Neural Network
classifier.compile(optimizer = 'Adamax', loss = 'binary_crossentropy', metrics = ['accuracy'])

#fitting ANN to the training Set
model_history=classifier.fit(X_train, y_train,validation_split=0.33, batch_size= 10, epochs = 100)

#List all data in History
print(model_history.history.keys())

#summarize history for accuracy
plt.plot(model_history.history['accuracy'])
plt.plot(model_history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

plt.plot(model_history.history['loss'])
plt.plot(model_history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
```

```
plt.legend(['train', 'test'], loc='upper left')
plt.show()

#Part-3 : Making the predictions and evaluating the model
y_pred = classifier.predict(X_test)
y_pred = (y_pred > 0.5)

#making confuction matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
print(cm)

#calculate the Accuracy
from sklearn.metrics import accuracy_score
score=accuracy_score(y_pred,y_test)
print(score)
```

2. Regression:

```
#To predict the burned area of forest fires, in the northeast region of Portugal, by using meteorological and other data
#Part-1: Pre Processing
import numpy as np
import pandas as pd

#Upload dataset into googlecolab
from google.colab import files
upload=files.upload()

#load the dataset into environment
forest=pd.read_csv('forestfires.csv')
forest.head()
```

```
print(forest)

#pop the month column
forest.pop('month')
forest.head()

#pop the day column
forest.pop('day')
forest.head()

#Count missing values for each column of the dataframe df
forest.isna().sum()

#LabelEncoder can be used to normalize labels. It is used to
transform non-numerical labels to numerical labels.
from sklearn import preprocessing
label_encoder=preprocessing.LabelEncoder()
forest['size_category']=label_encoder.fit_transform(forest['
size_category'])

forest.head()

#Identify the dependant and independant variabels
x=forest.iloc[:,0:28]
y=forest.iloc[:,28]

#Splitting Dataset into Training Set and Test Set
from sklearn.model_selection import train_test_split
```

```
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0
.2,random_state=0)

#feature Scaling
from sklearn.preprocessing import StandardScaler
sc=StandardScaler()
x_train=sc.fit_transform(x_train)
x_test=sc.fit_transform(x_test)

#Part-2: Model Construction
#Neural Network - ANN
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout

#initializing ANN
model=Sequential()

#adding input and 1st hidden layer
model.add(Dense(units=10,activation='relu',kernel_initializ
er='he_uniform',input_dim=28))

#adding 2nd hidden layer
model.add(Dense(units=8,activation='relu',kernel_initializer
='he_uniform'))

#adding output layer
model.add(Dense(units=1,kernel_initializer='glorot_uniform
',activation='sigmoid'))
```

```
#compile the model
model.compile(optimizer='adam',loss='binary_crossentropy
',metrics=['accuracy'])

#fit the model
x1=model.fit(x_train,y_train,batch_size=10,epochs=150,vali
dation_split=0.33)

#list the performance metrics
print(x1.history.keys())

#Part-3: Making the predictions and evaluating the model
y_pred=model.predict(x_test)

print(y_pred)

y_pred = (y_pred > 0.5)
y_pred

#evaluating the model
scores = model.evaluate(x_test, y_test)
print("%s: %.2f%%" % (model.metrics_names[1],
scores[1]*100))
```

Building blocks of deep networks

Building deep networks is beyond use of basic feed-forward multilayer neural networks. Deep networks combine smaller networks as building blocks into larger networks or they use a specialized set of layers.

The specific building blocks for deep neural networks are

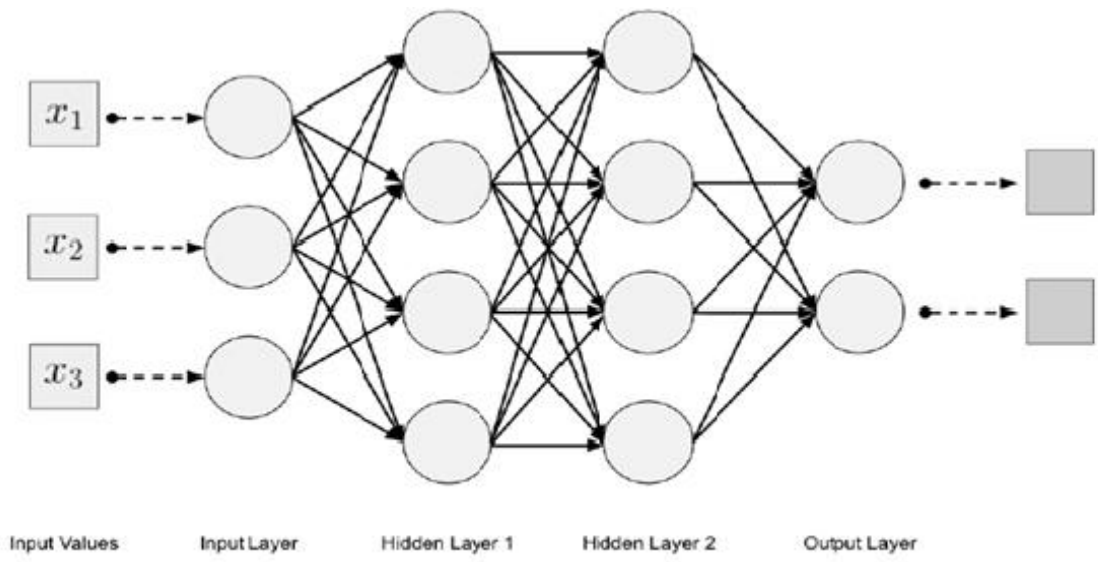
1. **Feed-forward multilayer neural networks**
2. **RBMs**
3. **Autoencoders**

1. Feed-forward multilayer neural networks

Inspired by networks of biological neurons, feed-forward networks are the simplest artificial neural networks.

With multilayer feed-forward neural networks, we have artificial neurons arranged into groups called layers. They are composed of

- **A single input layer**
- **One or many hidden layers, fully connected**
- **A single output layer**



- **The neurons in each layer (represented by the circles) are all fully connected to all neurons in all adjacent layers.**
- **The neurons in each hidden layer use the same type of activation function**
- **For the input layer, the input is the raw vector input. The input to neurons of the other layers is the output (activation) of the previous layer's neurons. As data moves through the network in a feed-forward fashion, it is influenced by the connection weights and the activation function type.**

Input layer:

- **This layer is how we get input data (vectors) fed into our network. The number of neurons in an input layer is typically the same number as the input feature to the network.**
- **Input layers are followed by one or more hidden layers**
- **Input layers in classical feed-forward neural networks are fully connected to the next hidden layer, yet in other network architectures, the input layer might not be fully connected.**

Hidden layer:

There are one or more hidden layers in a feed-forward neural network.

The weight values on the connections between the layers are how neural networks encode the learned information extracted from the raw training data.

Hidden layers are the key to allowing neural networks to model nonlinear functions.

Output layer:

- **We get the answer or prediction from our model from the output layer.**

- **the output layer gives us an output based on the input from the input layer.**
- **Depending on the setup of the neural network, the final output may be a real-valued output (regression) or a set of probabilities (classification).**
- **The output layer typically uses either a softmax or sigmoid activation function for classification.**

Connections between layers:

- **In a fully connected feed-forward network, the connections between layers are the outgoing connections from all neurons in the previous layer to all of the neurons in the next layer.**
- **We change these weights progressively as our algorithm finds the best solution it can with the backpropagation learning algorithm.**
- **We can understand the weights mathematically by thinking of them as the parameter vector.**

2. RBMs (Restricted Boltzmann Machines)

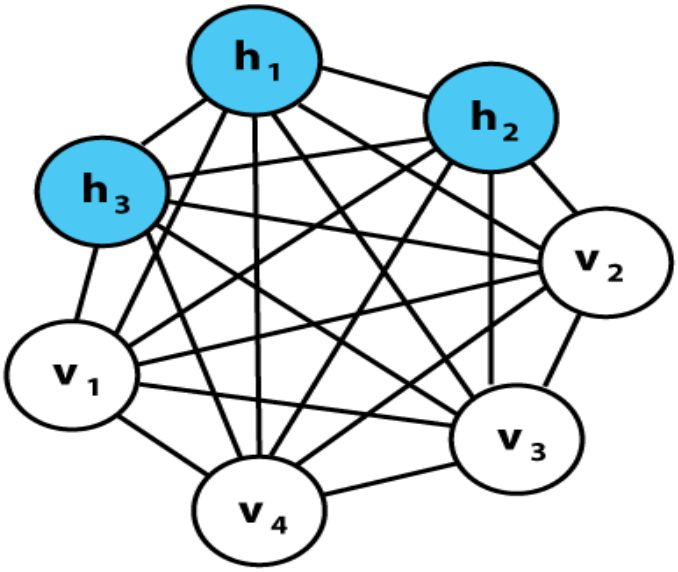
What are Boltzmann Machines?

It is a network of neurons in which all the neurons are connected to each other. In this machine, there are two layers named visible layer or input layer and hidden layer.

The visible layer is denoted as v and the hidden layer is denoted as the h.

In Boltzmann machine, there is no output layer.

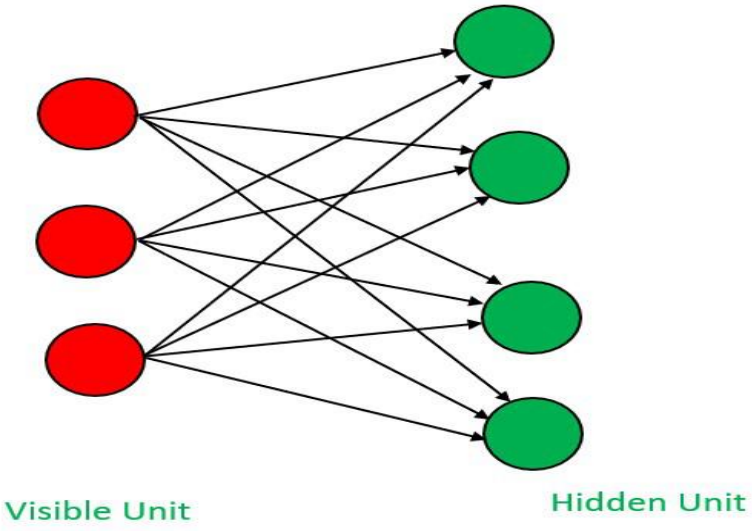
Boltzmann machines are random and generative neural networks capable of learning internal representations and are able to represent and (given enough time) solve tough combinatoric problems.



What are Restricted Boltzmann Machines (RBM)?

The "restricted" part of the name "Restricted Boltzmann Machines" means that connections between nodes of the same layer are prohibited (e.g., there are no visible-visible or hidden-hidden connections along which signal passes). we are not allowed to connect the same type layer to each other. Although the hidden layer and visible layer can be connected to each other.

Geoff Hinton, the deep learning pioneer who popularized RBM



- **RBMs are characterized by an extra layer-wise step for training.**
- **They are often used for the pretraining phase in other larger deep networks.**
- **A network of symmetrically connected, neuron-like units that make stochastic decisions about whether to be on or off.**
- **RBMs model probability and are great at feature extraction. They are feedforward networks in which data is fed through them in one direction with two biases rather than one bias as in traditional backpropagation feed-forward networks.**
- **RBMs are used in deep learning for the following:**
    1. **Feature extraction**
    2. **Dimensionality reduction**
    3. **Classification**
    4. **Regression**
    5. **Collaborative filtering**
    6. **Topic modeling**

Network layout of RBMs

There are five main parts of a basic RBM:

- **Visible units**
- **Hidden units**
- **Weights**
- **Visible bias units**
- **Hidden bias units**

- **With RBMs, every visible unit is connected to every hidden unit, yet no units from the same layer are connected.**
- **Each layer of an RBM can be imagined as a row of nodes. The nodes of the visible and hidden layers are connected by connections with associated weights.**

Visible and hidden layers:

- In an RBM, every single node of the input (visible) layer is connected by weights to every single node of the hidden layer, but no two nodes of the same layer are connected.
- The second layer is known as the "hidden" layer. Hidden units are feature detectors, learning features from the input data.
- Nodes in each layer are biologically inspired as with the feed-forward multilayer neural network. Units (nodes) in the visible layer are "observable" in that they take training vectors as input.
- Each layer has a bias unit with its state always set to on.
- Each node performs computation based on the input to the node and outputs a result based on a stochastic decision whether or not to transmit data through an activation.
- The activation computation is based on weights on the connections and the input values.
- The initial weights are randomly generated.

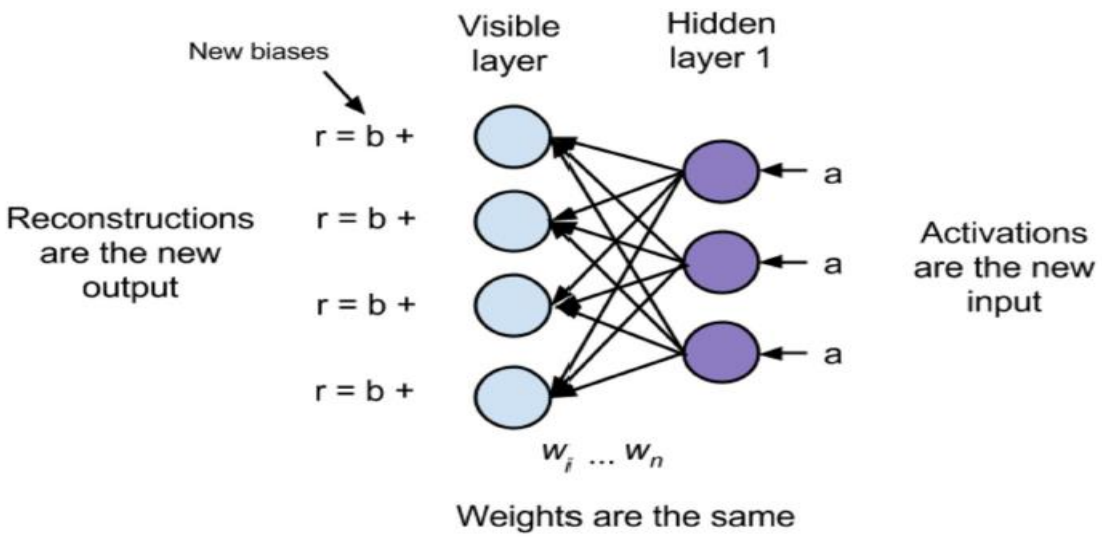
Connections and weights:

- All connections are visible-hidden; none are visible-visible or hidden-hidden.
- The edges represent connections along which signals are passed.
- Nodes, act like human neurons. They are decision units. They make decisions about whether to be on or off through acts of computation. "On" means that they pass a signal further through the net; "off" means that they don't.
- "on" means the data passing through the node is valuable; it contains information that will help the network make a decision. Being "off" means the network thinks that particular input is irrelevant noise.

- **A network comes to know which features/signals correlate with which labels (which code contains which messages) by being trained. With training, networks learn to make accurate classifications of the input they receive.**

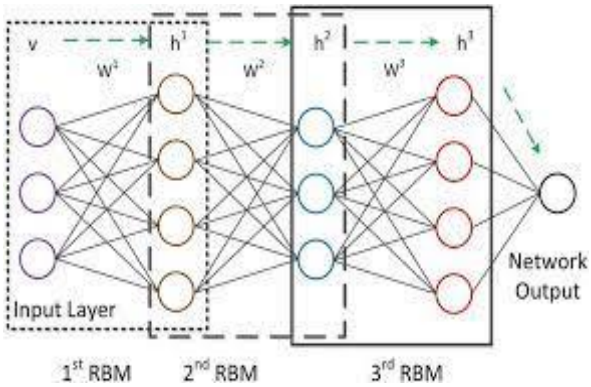
Biases:

- **There is a set of bias weights ("parameters") connecting the bias unit for each layer to every unit in the layer.**
- **Bias nodes help the network better triage and model cases in which an input node is always on or always off.**



Training:

- **The technique known as pretraining using RBMs means teaching it to reconstruct the original data from a limited sample of that data.**
- **RBMs learn to reconstruct the input dataset.**



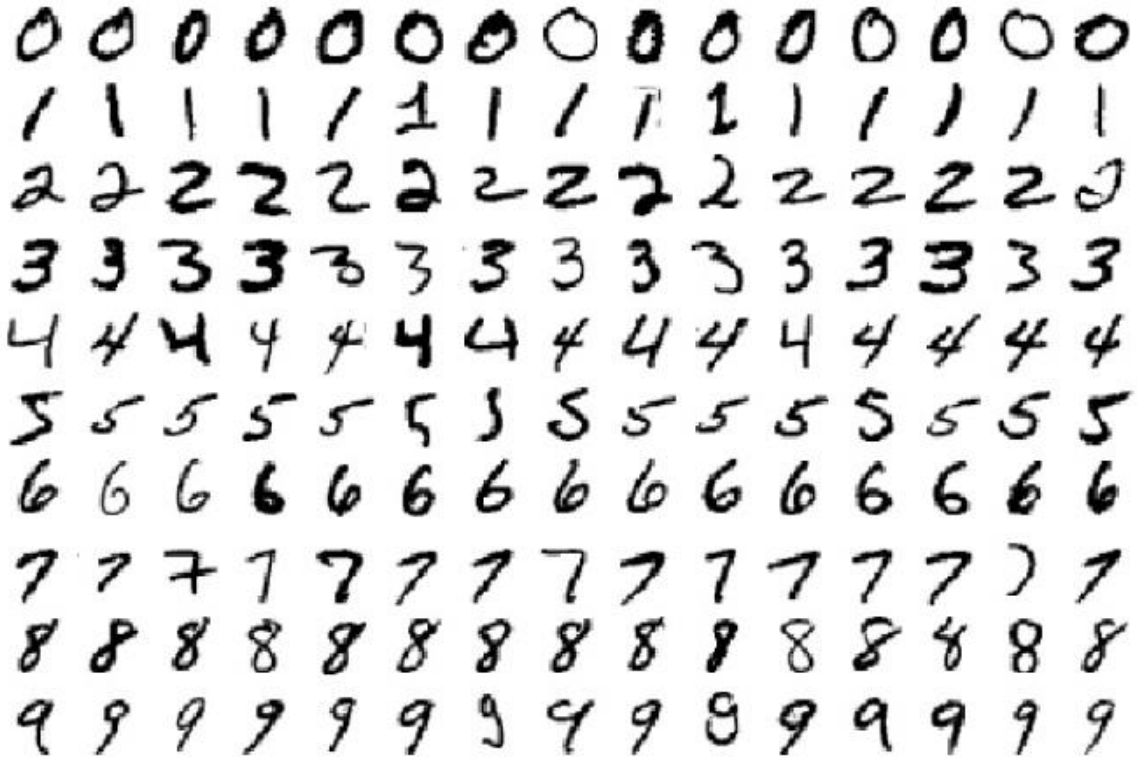
Reconstruction:

Deep neural networks with unsupervised pretraining (RBMs) perform feature engineering from unlabeled data through reconstruction.

In pretraining, the weights learned through unsupervised pretrain learning are used for weight initialization in networks such as Deep Belief Networks.

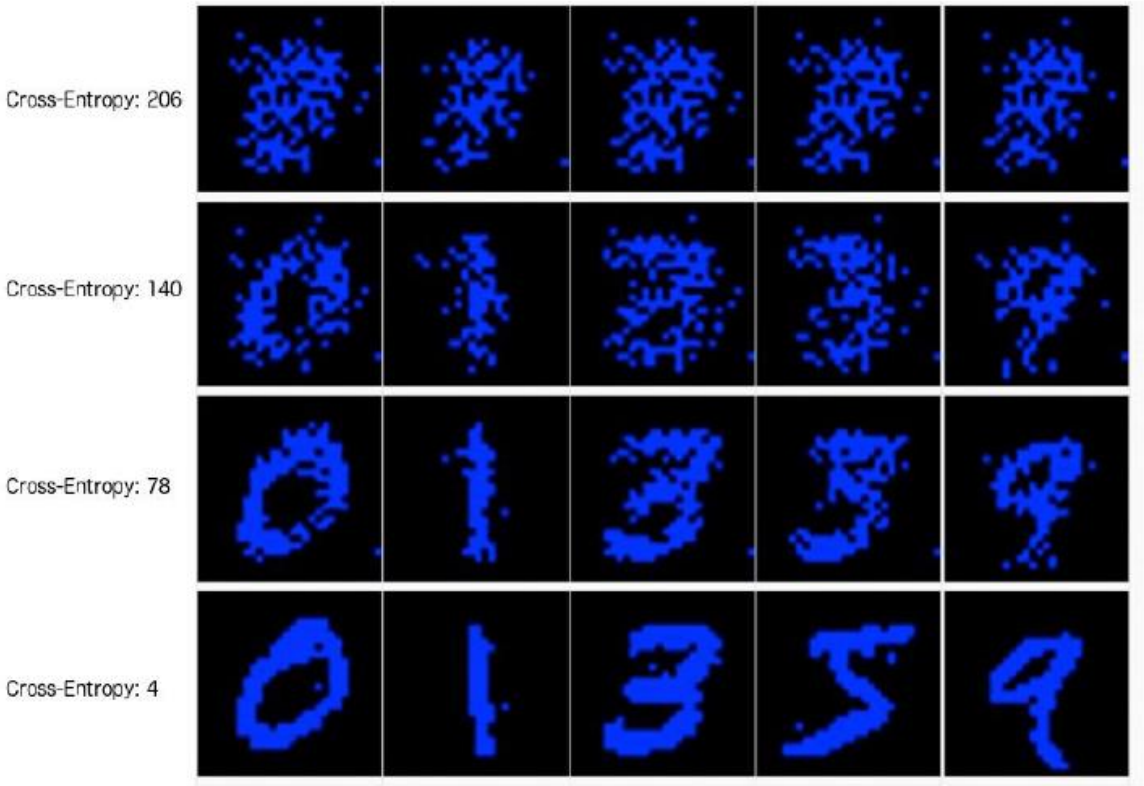
The reconstruction in RBMs is shown with the MNIST dataset. MNIST stands for the mixed National Institute of Standards and Technology dataset that contains the images. The MNIST dataset is a collection of images representing the handwritten numerals 0 through 9.

A sample of some of the handwritten digits in MNIST.



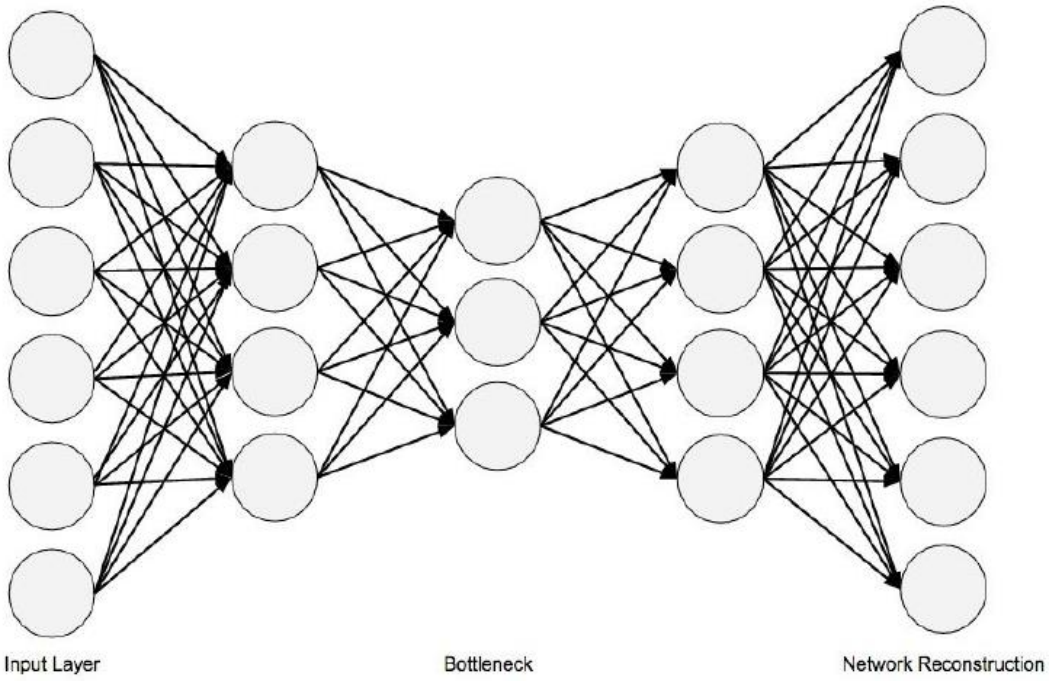
The training dataset in MNIST has 60,000 records and the test dataset has 10,000 records.

If we use a RBM to learn the MNIST dataset, we can sample from the trained network reconstruction of the digits.


Cross-Entropy: 206
Cross-Entropy: 140
Cross-Entropy: 78
Cross-Entropy: 4

3. Auto encoders

- **We use autoencoders to learn compressed representations of datasets. Typically used to reduce a dataset's dimensionality.**
- **The output of the autoencoder network is a reconstruction of the input data in the most efficient form.**
- **Autoencoders share a strong resemblance with multilayer perceptron neural networks in that they have an input layer, hidden layers of neurons, and then an output layer.**
- **The key difference between a multilayer perceptron network and an autoencoder diagram is the output layer in an autoencoder has the same number of units as the input layer does.**

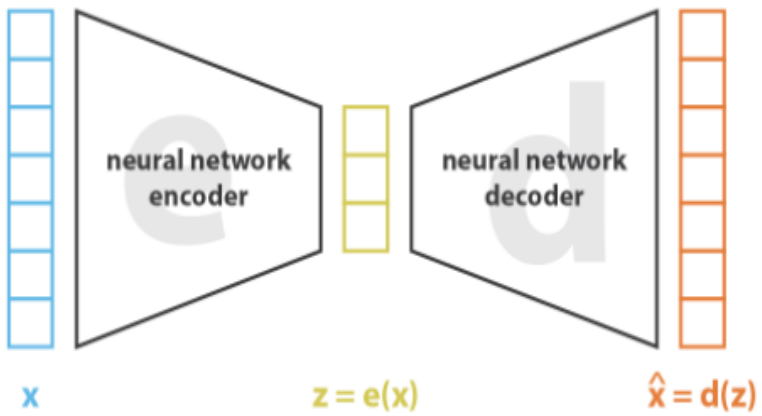


Training autoencoders:

Autoencoders rely on backpropagation to update their weights. The main difference between RBMs and the more general class of autoencoders is in how they calculate the gradients.

- **The general idea of autoencoders is simple and consists in setting an encoder and a decoder as neural networks and to learn the best encoding-decoding scheme using an iterative optimisation process.**
- **At each iteration we feed the autoencoder architecture (the encoder followed by the decoder) with some data, we compare the encoded-decoded output with the initial data and backpropagate the error through the architecture to update the weights of the networks.**

- **The overall autoencoder architecture (encoder+decoder) creates a bottleneck for data that ensures only the main structured part of the information can go through and be reconstructed.**
- **In general framework, the family E of considered encoders is defined by the encoder network architecture, the family D of considered decoders is defined by the decoder network architecture and the search of encoder and decoder that minimize the reconstruction error is done by gradient descent over the parameters of these networks.**



$$\text{loss} = \| x - \hat{x} \|^2 = \| x - d(z) \|^2 = \| x - d(e(x)) \|^2$$

Autoencoders differ from multilayer perceptrons as

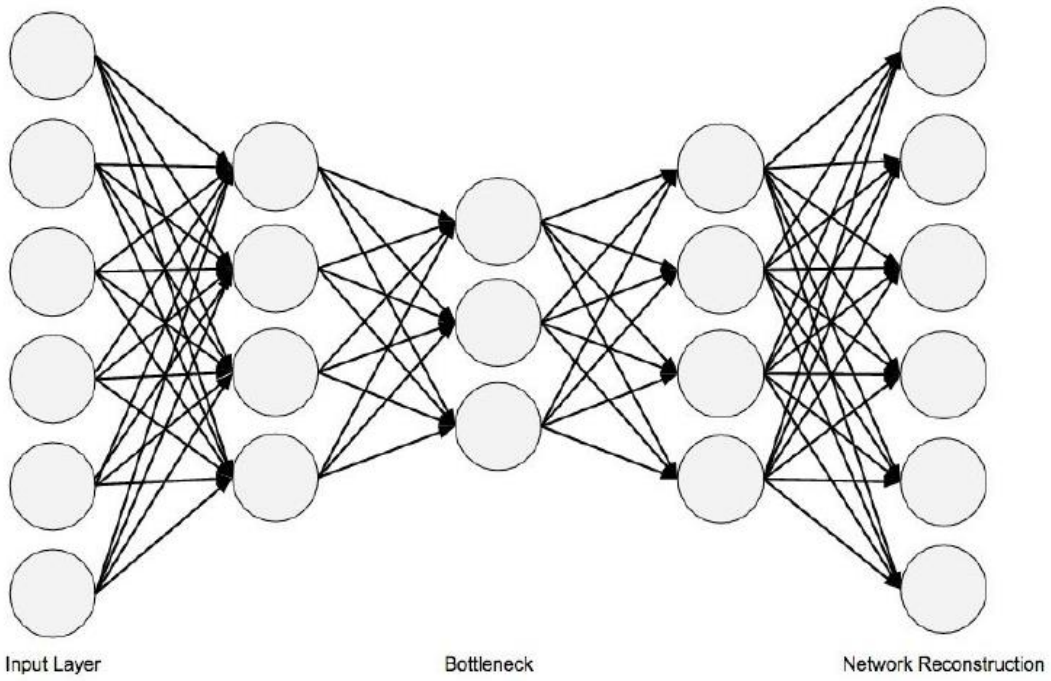
- **They use unlabeled data in unsupervised learning. i.e. The autoencoder learns directly from unlabeled data.**
- **They build a compressed representation of the input data. i.e The goal of a multilayer perceptron network is to generate predictions over a class (e.g., fraud versus not fraud). An autoencoder is trained to reproduce its own input data.**
- **The output layer has the same number of units as the input layer does.**

Common variants of autoencoders:

Two important variants of autoencoders to note are compression autoencoders and denoising autoencoders.

1. Compression autoencoders

The network input must pass through a bottleneck region of the network before being expanded back into the output representation.



2. Denoising autoencoders

The denoising autoencoder is the scenario in which the autoencoder is given a corrupted version (e.g., some features are removed randomly) of the input and the network is forced to learn the uncorrupted output.

Autoencoders are commonly used in systems in which we know what the normal data will look like, yet it's difficult to describe what is anomalous. Autoencoders are good at powering anomaly detection systems.

Applications of Autoencoders:

- **Dimensionality Reduction.**
- **Image Compression.**
- **Image Denoising.**
- **Feature Extraction.**
- **Image generation.**
- **Sequence to sequence prediction. (text data)**
- **Recommendation system.**

Unsupervised Pretrained Networks

1. Auto encoders

Autoencoder is an unsupervised artificial neural network that learns how to efficiently compress and encode data then learns how to reconstruct the data back from the reduced encoded representation to a representation that is as close to the original input as possible.
Autoencoder, by design, reduces data dimensions by learning how to ignore the noise in the data.

Autoencoders consists of 4 main parts:

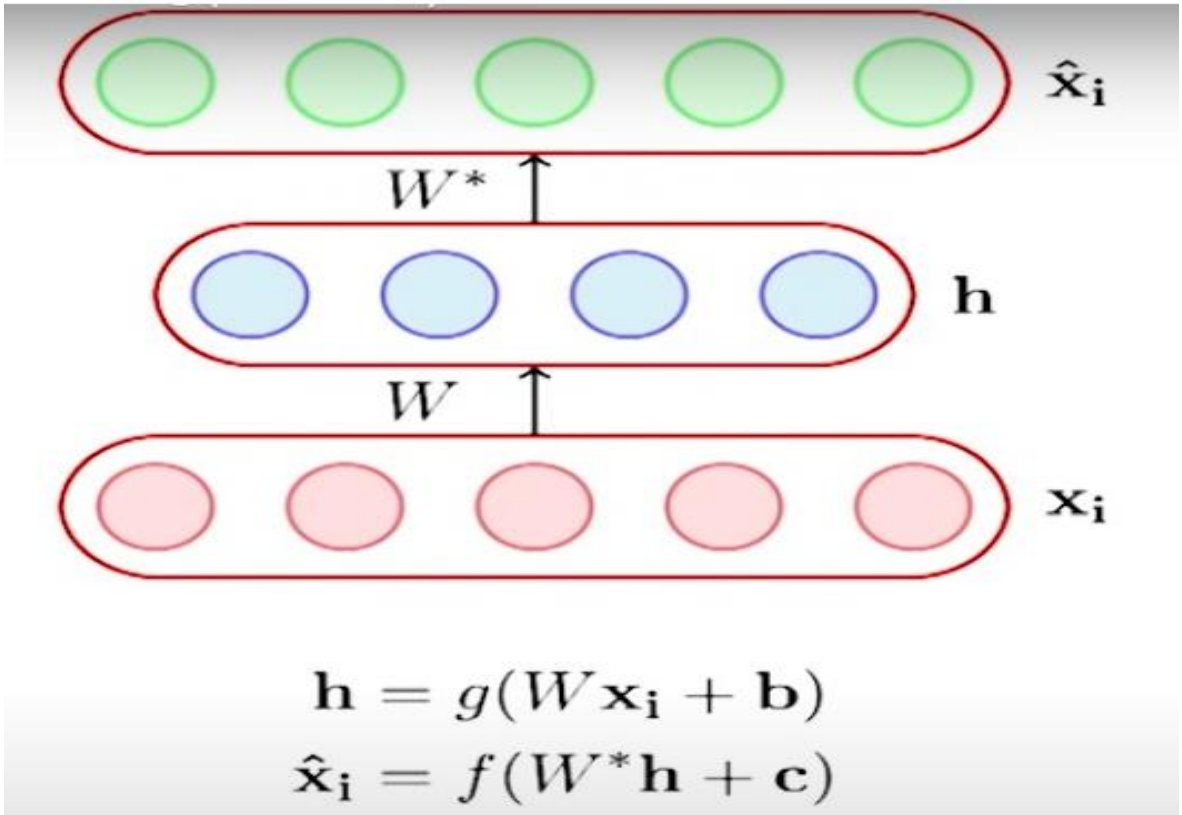
1- Encoder: In which the model learns how to reduce the input dimensions and compress the input data into an encoded representation.

2- Bottleneck: which is the layer that contains the compressed representation of the input data. This is the lowest possible dimensions of the input data.

3- Decoder: In which the model learns how to reconstruct the data from the encoded representation to be as close to the original input as possible.

4- Reconstruction Loss: This is the method that measures measure how well the decoder is performing and how close the output is to the original input.

An auto encoder is a special type of feed forward neural network which does the following.



Encodes its input Xi into a hidden representation 'h', Uses an Encoder function

i.e. $h = g(WX_i + b)$

Decodes the input again from this hidden representation and reconstructs the input again from it.

i.e. $\hat{\mathbf{x}}_\mathbf{i} = f(W^*\mathbf{h} + \mathbf{c})$

The compression is done to choose most important characteristics of input features.

The model is trained to minimize a certain loss function which will ensure that $\hat{\mathbf{x}}_\mathbf{i}$ **is close to** $\mathbf{x}_\mathbf{i}$ **.**

Let us consider the following two cases.

1. dim(h) < dim(xi)

   Here the dimensions of hidden layer (bottleneck) is less than the input dimensions.

   In such a case , We can reconstruct the $\hat{x}_i$ perfectly , able to identify the useful insights.

   Here h is a loss-free encoding of x_i , it captures all the important characteristics of x_i

2. dim(h) >= dim(xi)

   Here the dimensions of hidden layer are greater than or equal to the input dimensions.

   In such a case, the auto encoder could learn a trivial encoding by simply copying the x_i into h and then copying h into $\hat{x}_i$.

   Such an identity encoding is not useful in practice as it does not really tell us anything about the important characteristics of the data.

   An auto encoder where dim(h) >= dim(xi) is called an "Over Complete Autoencoder".

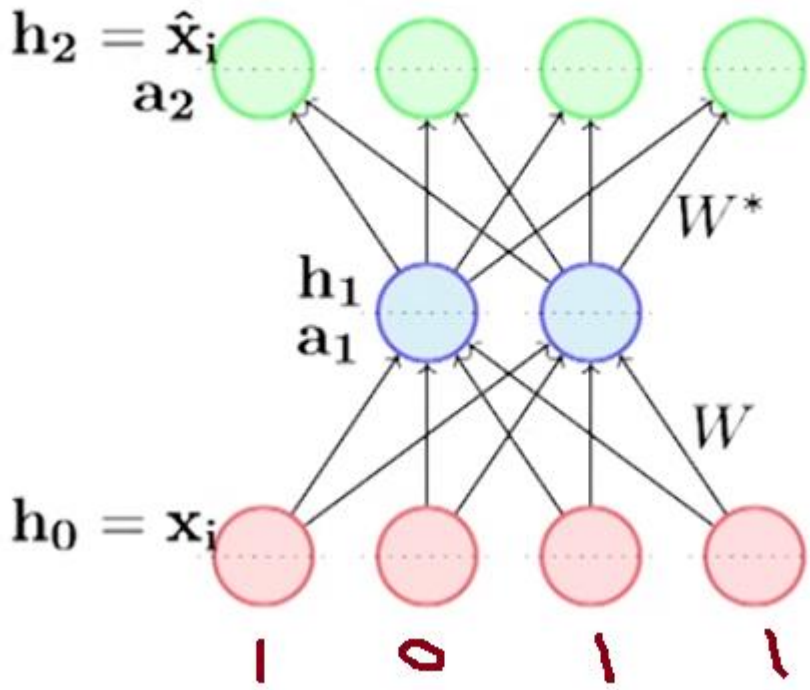
The hyper parameters to train the autoencoder are

1. Code Size (Bottleneck) -> Number of nodes in middle layer.
2. Number of Layers -> two or more
3. Loss Function -> Mean Squared Error / Binary Cross Entropy
4. Number of Nodes per Layer -> For Encoder the number of layers decrease in subsequent layers, increases in decoder.

The training then involves using back propagation in order to minimize the network's reconstruction loss.

Suppose all our inputs are binary and the output is binary

$$\mathscr{L}(\theta) = -\sum_{j=1}^{n} (x_{ij} \log \hat{x}_{ij} + (1 - x_{ij}) \log(1 - \hat{x}_{ij}))$$



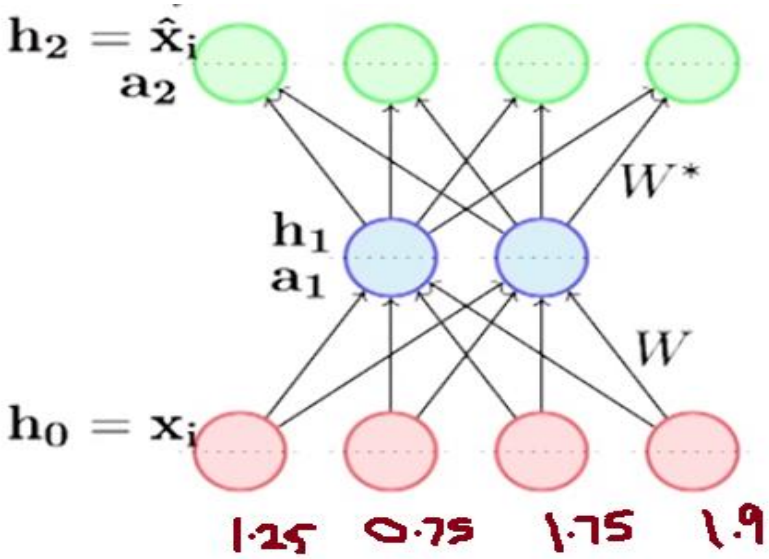
The activation function for encoder [g()] during encoder can be choosen as sigmoid/tanh.

The activation function for decoder [f()] during decoder can be sigmoid or logistic.

The cost function is Cross Entrophy

$$\mathscr{L}(\theta) = -\sum_{j=1}^{n} (x_{ij} \log \hat{x}_{ij} + (1 - x_{ij}) \log(1 - \hat{x}_{ij}))$$

Suppose all our inputs the output is real numbers



The activation function for encoder [g()] during encoder can be choosen as sigmoid/tanh.

The activation function for decoder [f()] during decoder can be sigmoid or logistic.

The cost function is "Mean Squared Error".

$$\mathscr{L}(\theta) = (\hat{\mathbf{x}}_\mathbf{i} - \mathbf{x}_\mathbf{i})^T(\hat{\mathbf{x}}_\mathbf{i} - \mathbf{x}_\mathbf{i})$$

The training then involves using back propagation in order to minimize the network's reconstruction loss.

During Backpropagation the derivative is computes as

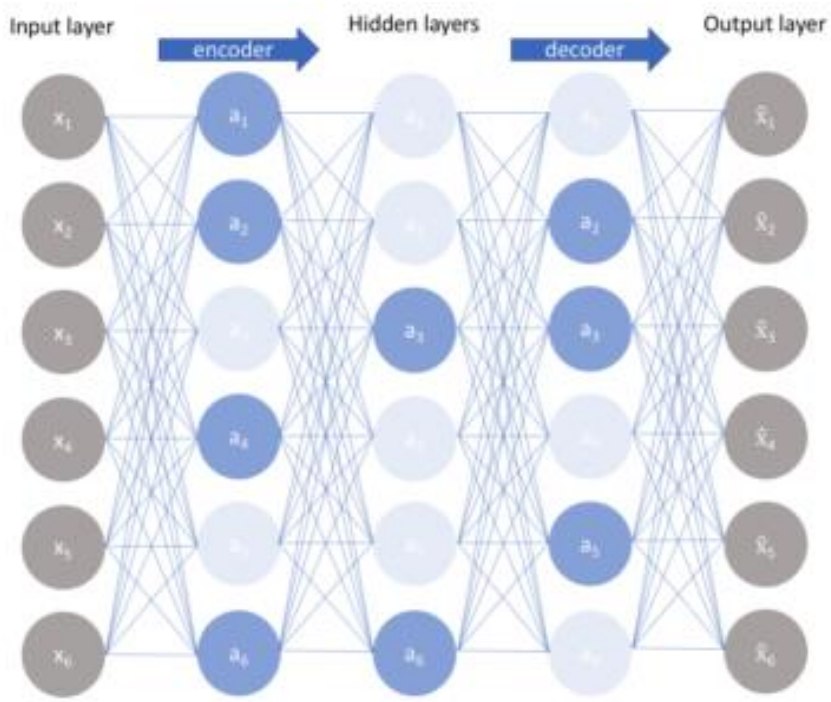
- $$\frac{\partial \mathscr{L}(\theta)}{\partial W^*} = \frac{\partial \mathscr{L}(\theta)}{\partial \mathbf{h_2}} \frac{\partial \mathbf{h_2}}{\partial \mathbf{a_2}} \boxed{\frac{\partial \mathbf{a_2}}{\partial W^*}}$$
- $$\frac{\partial \mathscr{L}(\theta)}{\partial W} = \frac{\partial \mathscr{L}(\theta)}{\partial \mathbf{h_2}} \frac{\partial \mathbf{h_2}}{\partial \mathbf{a_2}} \boxed{\frac{\partial \mathbf{a_2}}{\partial \mathbf{h_1}} \frac{\partial \mathbf{h_1}}{\partial \mathbf{a_1}} \frac{\partial \mathbf{a_1}}{\partial W}}$$

The various types of autoencoders are

1. Sparse Autoencoders

Sparse autoencoders offer us an alternative method for introducing an information bottleneck without requiring a reduction in the number of nodes at our hidden layers. Instead, we'll construct our loss function such that we penalize activations within a layer.

This type of auto-encoder typically contains more hidden units than the input but only a few are allowed to be active at once. This property is called the sparsity of the network. The sparsity of the network can be controlled by either manually zeroing the required hidden units, tuning the activation functions or by adding a loss term to the cost function.



A hidden neuron with sigmoid activation function will have the values between 0 and 1. We will think of a neuron as

being "active" (or as "firing") if its output value is close to 1, or as being "inactive" if its output value is close to 0.

Even when the number of hidden units is large (perhaps even greater than the number of input pixels), we can still discover interesting structure, by imposing other constraints on the network.

A sparse auto encoder tries to ensure the neuron is inactive most of the times. i.e. close to zero for lot of inputs. The average activation of a neuron is close to Zero.

If we impose a sparsity constraint on the hidden units, then the autoencoder will still discover interesting structure in the data, even if the number of hidden units is large.

Recall that a_j denotes the activation of hidden unit j in the autoencoder. Thus, we will write $a_j(x)$ to denote the activation of this hidden unit when the network is given a specific input x.

$$\hat{\rho}_j = \frac{1}{m} \sum_{i=1}^{m} \left[a_j \quad (x^{(i)}) \right]$$

The average activation of hidden unit j (averaged over the training set).

We would like to (approximately) enforce the constraint

$$\rho_j\hat{} = \rho,$$

The sparse autoencoder uses a sparsity parameter 'ρ' typically a small value close to zero (say ρ = 0.05). In other words, we would like the average activation of each hidden neuron j to be close to 0.05 (say). To satisfy this constraint, the hidden unit's activations must mostly be near 0.

The idea is to activate only few neurons so that meaningful insights will be extracted. Restricting the neuron is a form of regularization

To achieve this, to the loss function the following regularization parameter is added.

$$\sum_{j=1}^{s_2} \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j}.$$

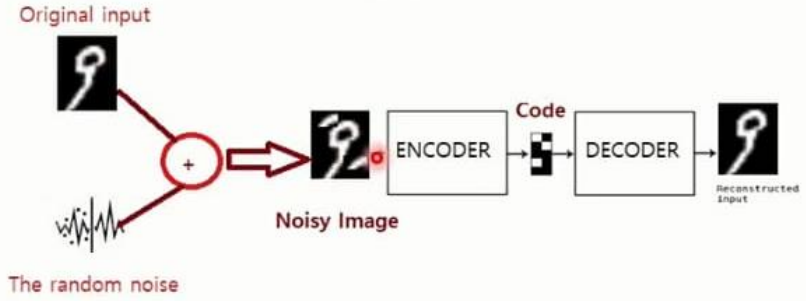
Here, s2 is the number of neurons in the hidden layer, and the index j is summing over the hidden units in our network.

2. Denoising Autoencoders

Autoencoders are Neural Networks which are commonly used for feature selection and extraction. However, when there are more nodes in the hidden layer than there are inputs, the Network is risking to learn the so-called "Identity Function", also called "Null Function", meaning that the output equals the input, marking the Autoencoder useless.

This type of auto-encoder works on a partially corrupted input and trains to recover the original undistorted image. As mentioned above, this method is an effective way to constrain the network from simply copying the input.

- This is a peculiar one !! But, you will appreciate the purpose once you know how it works!
- Denoising Autoencoders produce a Corrupted (noised) copy of the input through the introduction of some noise. i.e. noise added to corrupt the input!
- Why to do this? **Autoencoders with more hidden layers than inputs run the risk of learning the identity function – where the output simply equals the input – thereby becoming useless. i.e. the no learning the features happen.**
- **So, this random noise inclusion avoids that. Here, we force the autoencoder to learn the original data after the noise being removed. Here, the autoencoder identifies the noise, removes the noise, learns the important features from the input data.**



Denoising Autoencoders are an important and crucial tool for feature selection and extraction

3. Contractive Autoencoders

The idea behind contractive autoencoders is to make the autoencoders robust of small changes in the training dataset.

To deal with the above challenge that is posed in basic autoencoders, the authors proposed to add another penalty term to the loss function of autoencoders.

The Contractive auto-encoder (CAE) is obtained with the regularization term, yielding objective function

$$\mathcal{J}_{\mathrm{CAE}}(\theta) = \sum_{x \in D_n} \left(L(x, g(f(x))) + \lambda \| J_f(x) \|_F^2 \right)$$

Here the regularization term $\lambda \| J_f(x) \|_F^2)$ **is evaluated as**

$$\|J_f(x)\|_F^2 = \sum_{ij} \left(\frac{\partial h_j(x)}{\partial x_i} \right)^2$$

To encourage robustness of the representation f(x) obtained for a training input x we propose to penalize its sensitivity to that input, measured as the Frobenius norm of the Jacobian Jf (x) of the non-linear mapping.

Formally, if input $x \in \mathbb{R}^{d_x}$ is mapped by encoding function f to hidden representation $h \in \mathbb{R}^{d_h}$, this sensitivity penalization term is the sum of squares of all partial derivatives of the extracted features with respect to input dimensions:

$$\|J_f(x)\|_F^2 = \sum_{ij} \left(\frac{\partial h_j(x)}{\partial x_i} \right)^2 .$$

It encourages the mapping to the feature space to be contractive in the neighborhood of the training data.

4. Variational Autoencoders

2. Deep Belief Networks (DBNs)

DBNs are composed of layers of Restricted Boltzmann Machines (RBMs) for the pretrain phase and then a feed-forward network for the fine-tune phase.

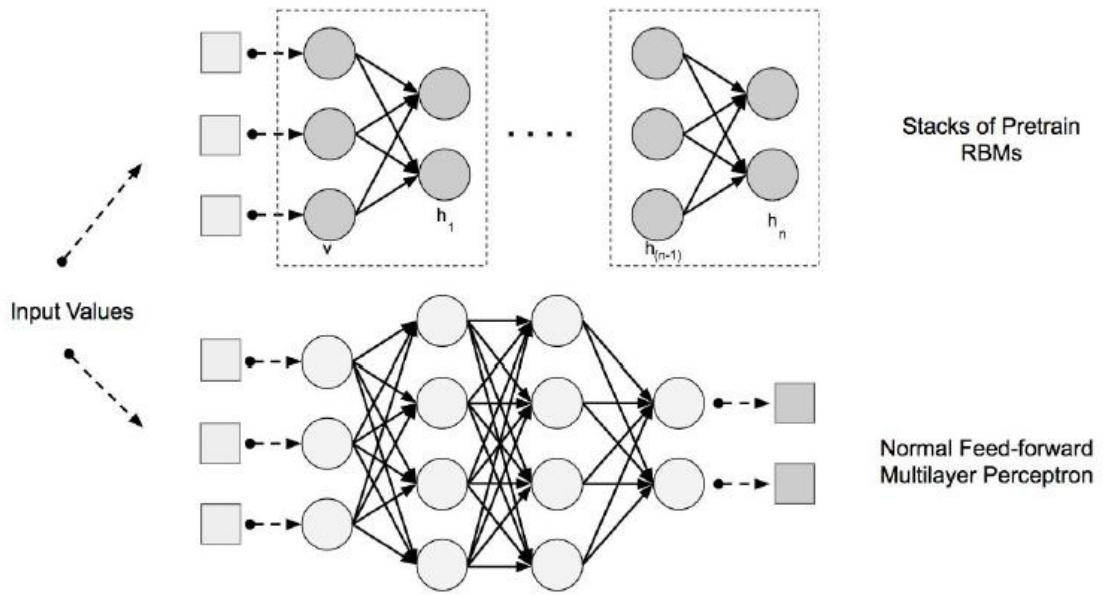
The Deep Belief Network (DBN) is a kind of Deep Neural Network, which is composed of stacked layers of Restricted Boltzmann Machines (RBMs). It is a generative model and was proposed by Geoffrey Hinton. DBN can be used to solve unsupervised learning tasks to reduce the dimensionality of features, and can also be used to solve supervised learning tasks to build classification models or regression models.

To train a DBN, there are two steps

1. **layer-by-layer training: Layer-by-layer training refers to unsupervised training of each RBM**
2. **fine-tuning: refers to the use of error back-propagation algorithms to fine-tune the parameters of DBN after the unsupervised training is finished.**

DBNs take advantage of RBMs to better model training data.

The network architecture of a DBN



In DBNs

Feature Extraction with RBM Layers:

RBMs used to extract higher-level features from the raw input vectors. To do that, we want to set the hidden unit states and weights such that when we show the RBM an input record and ask the RBM to reconstruct the record, it generates something pretty close to the original input vector.

The fundamental purpose of RBMs in the context of deep learning and DBNs is to learn these higher-level features of a dataset in an unsupervised training fashion.

We can train better neural networks by letting RBMs learn progressively higher-level features using the learned features from

a lower level RBM pretrain layer as the input to a higher-level RBM pretrain layer.

Learning these features in an unsupervised fashion is considered the pretrain phase of DBNs. Each hidden layer of the RBM in the pretrain phase learns progressively more complex features from the distribution of the data.

These higher-order features are progressively combined in nonlinear ways to do elegant automated feature engineering.

3. Generative Adversarial Networks (GANs).

Generative Adversarial Networks (GANs) are a powerful class of neural networks that are used for unsupervised learning.

GANs are basically made up of a system of two competing neural network models which compete with each other and are able to analyze, capture and copy the variations within a dataset.

Generative Adversarial Networks (GANs) can be broken down into three parts:

Generative: To learn a generative model, which describes how data is generated in terms of a probabilistic model.

Adversarial: The training of a model is done in an adversarial setting.

Networks: Use deep neural networks as the artificial intelligence (AI) algorithms for training purpose.

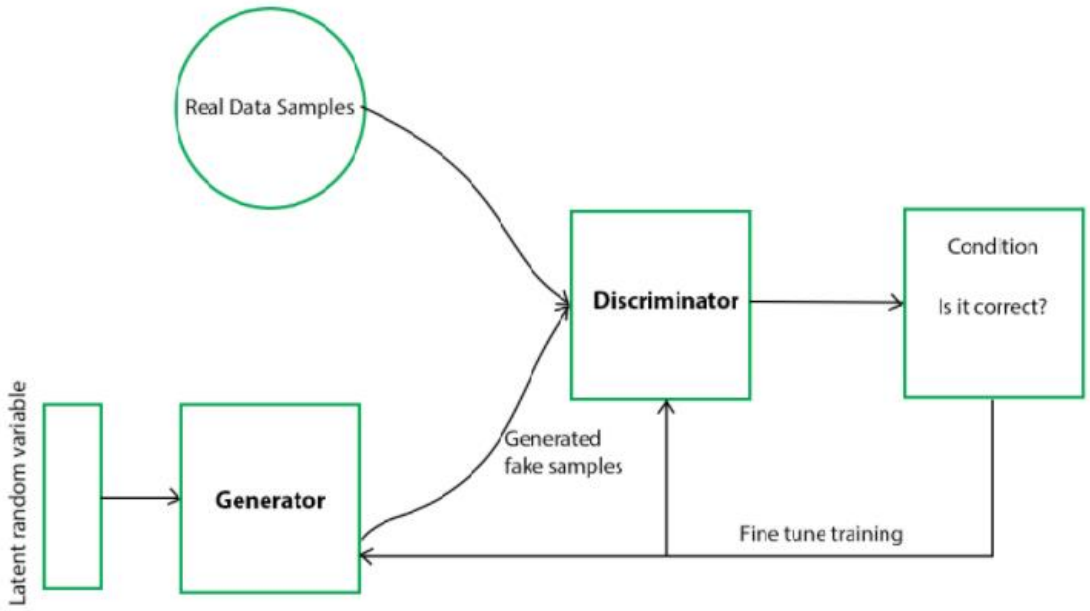
In GANs, there is a generator and a discriminator.

The Generator generates fake samples of data (be it an image, audio, etc.) and tries to fool the Discriminator.

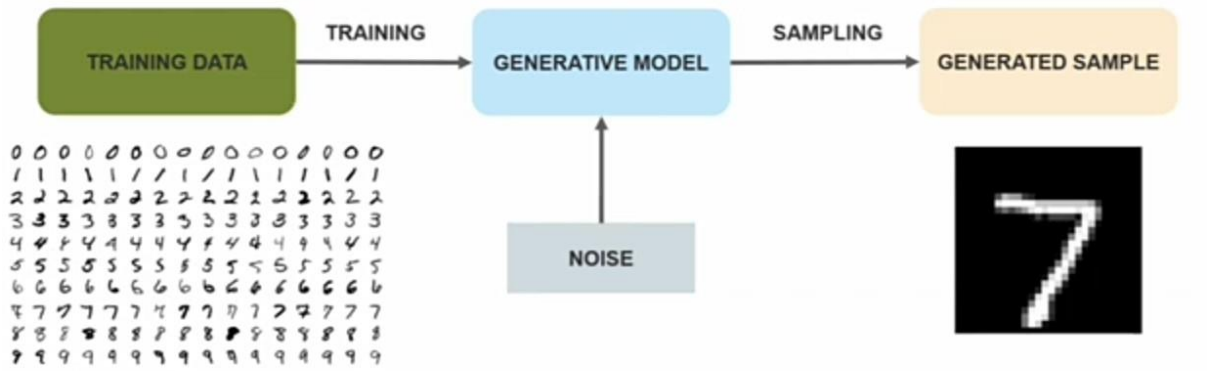
The Discriminator, on the other hand, tries to distinguish between the real and fake samples.

The Generator and the Discriminator are both Neural Networks and they both run in competition with each other in the training phase. The steps are repeated several times and in this, the Generator and Discriminator get better and better in their respective jobs after each repetition.

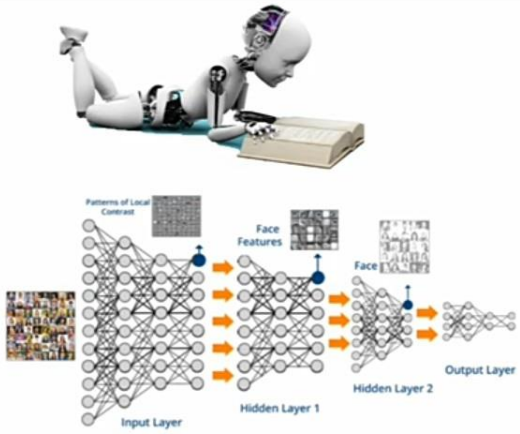
The working can be visualized by the diagram given below:



What Are Generative Model?



What Are GANs?



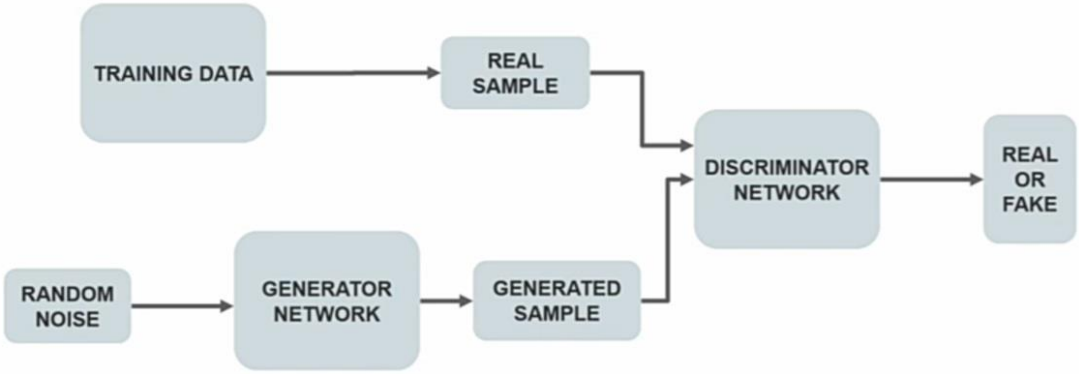
GENERATOR

Generator network that takes a sample and generates a sample of data

DISCRIMINATOR

Discriminator network decides whether the data is generated or taken from the real sample using a binary classification pr oblem with the help of a sigmoid function that gives the output in the range 0 to 1.

How Does A GAN Work?

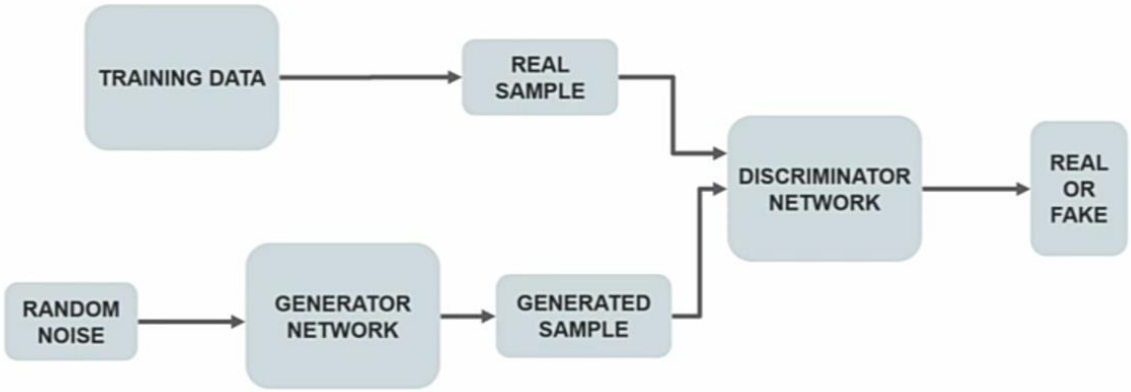


How Does A GAN Work?

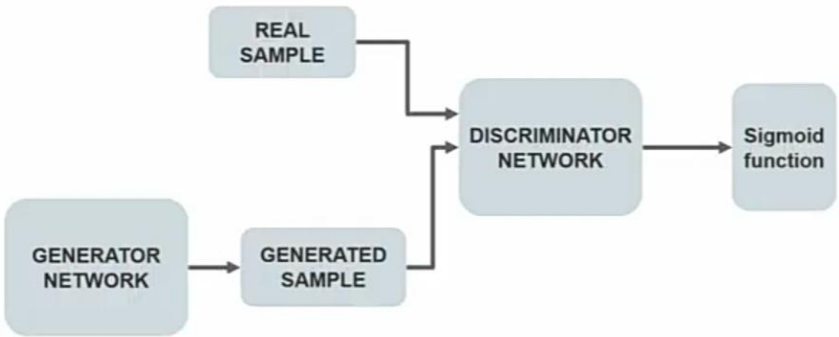
$$V(D,G) = E_{x\sim Pdata(x)} [\log D(x)] + E_{z\sim pz(z)} [\log(1 - D(G(z))]$$

G = Generator	x = sample from real data
D = Discriminator	z = sample from generator
Pdata(x) = Distribution of real data	D(x) = Discriminator Network
Pdata(z) = Distributor of generator	G(z) = Generator Network

How Does A GAN Work?



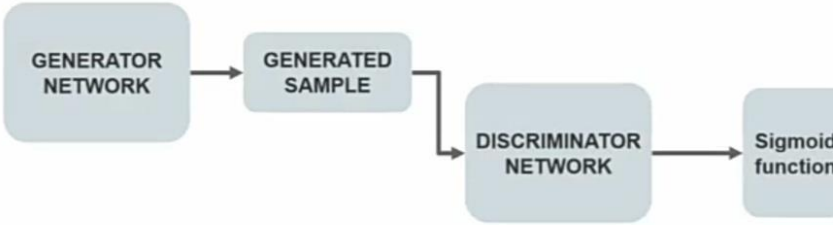
How To Train A GAN?



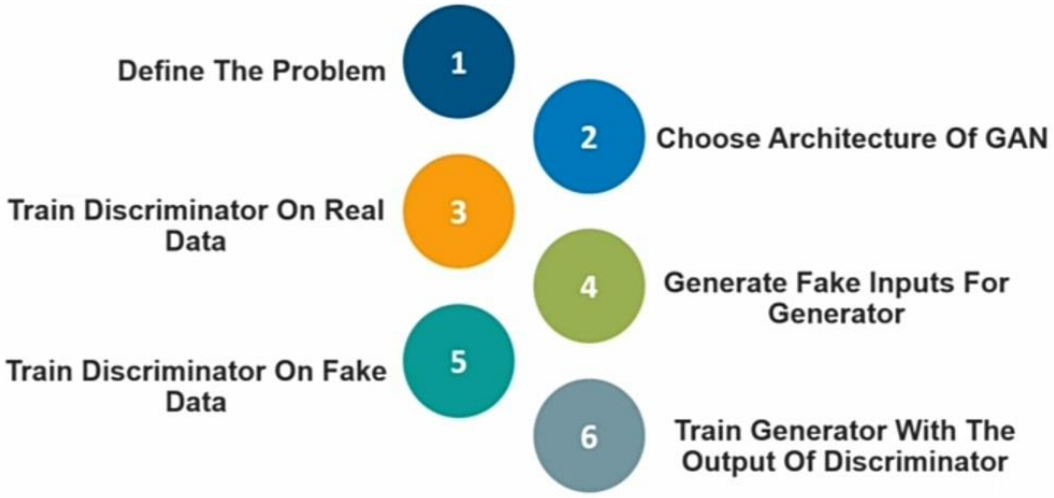
Train the discriminator and freeze the generator, which means the training set for the generator is turned as False and the network will only do the forward pass and no back-propagation will be applied

How To Train A GAN?

Train the generator and freeze the discriminator. In this phase, we get the results from the first phase and can use them to make better from the previous state to try and fool the discriminator better.



How To Train A GAN?



Challenges Faced By GANs

- Problem of stability between generator and discriminator
- Problem to determine positioning of the objects
- Problem in understanding the perspective
- Problem in understanding global objects

GANs Applications

Prediction of Next Frame In A Video

GANs Applications

Text To Image Generation

A Flower With Red Petals And Green Leaves

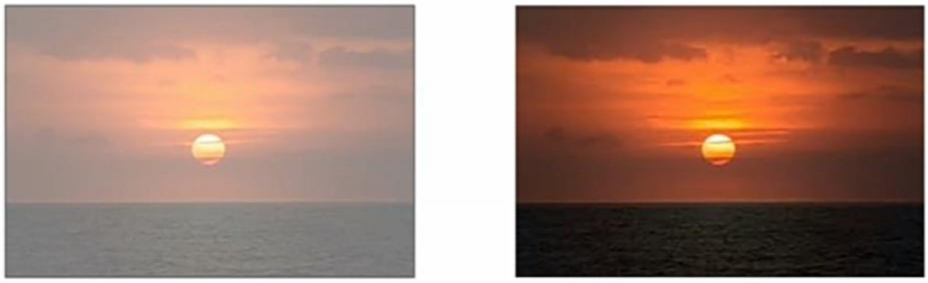
GANs Applications

Image To Image Translation

Real | Generated | Reconstructed

GANs Applications

Enhancing the Resolution Of An Image



GANs Applications

Interactive image generation