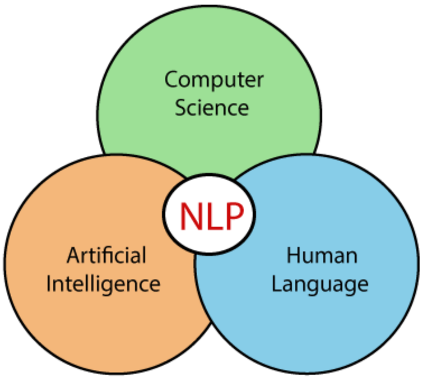
UNIT-5

Topics

- **Interactive Applications of Deep Learning**
  - **Natural Language Processing**
  - **Generative Adversarial Networks**
  - **Deep Reinforcement Learning**
- **Deep Learning Research**
  - **Auto Encoders**
  - **Deep Generative Models**
    - **Boltzmann Machine**
    - **Restricted Boltzmann Machine**
    - **Deep Belief Networks**
    - **Deep Boltzmann Machine**

Natural Language Processing

- Natural language processing (NLP) is a branch of artificial intelligence (AI) that enables machines to understand human language.
- The main intention of NLP is to build systems that are able to make sense of text and then automatically execute tasks like spell-check, text translation, topic classification, etc.
- NLP stands for **Natural Language Processing**, which is a part of **Computer Science, Human language,** and **Artificial Intelligence**.
- It is the technology that is used by machines to understand, analyze, manipulate, and interpret human's languages.
- It helps developers to organize knowledge for performing tasks such as **translation, automatic summarization, Named Entity Recognition (NER), speech recognition, relationship extraction,** and **topic segmentation**.



Components

- Natural Language Generation (NLG)
- Natural Language Understanding (NLU)

Natural Language Generation (NLG)

- NLG is a method of creating meaningful phrases and sentences (natural language) from data. It comprises three stages: text planning, sentence planning, and text realization.

  - Text planning: Retrieving applicable content.
  - Sentence planning: Forming meaningful phrases and setting the sentence tone.
  - Text realization: Mapping sentence plans to sentence structures.

- Chatbots, machine translation tools, analytics platforms, voice assistants, sentiment analysis platforms, and AI-powered transcription tools are some applications of NLG.

Natural Language Understanding (NLU)

- NLU enables machines to understand and interpret human language by extracting metadata from content. It performs the following tasks:
  - Helps analyze different aspects of language.
  - Helps map the input in natural language into valid representations.
- NLU is more difficult than NLG tasks owing to referential, lexical, and syntactic ambiguity.
  - Lexical ambiguity: This means that one word holds several meanings. For example, "The man is looking for the match." The sentence is ambiguous as ‘match’ could mean different things such as a partner or a competition.
  - Syntactic ambiguity: This refers to a sequence of words with more than one meaning. For example, "The fish is ready to eat.” The ambiguity here is whether the fish is ready to eat its food or whether the fish is ready for someone else to eat. This ambiguity can be resolved with the help of the part-of-speech tagging technique.
  - Referential ambiguity: This involves a word or a phrase that could refer to two or more properties. For example, Tom met Jerry and John. They went to the movies. Here, the pronoun ‘they’ causes ambiguity as it isn’t clear who it refers to.

Natural Language Processing

- **Natural Language Processing (NLP)** is the ability of a machine to read, write, understand and derive meaning from a human language.
- **Steps in NLP**
  - Tokenization
  - Stemming
  - Lemmatization
  - Part-of-speech (POS) tagging
  - Named entity recognition
  - Chunking

1. **Tokenization**: We break down the text into **tokens**. Check the example below to see how this is done.
   - **Text:** The cat sat on the bed. **Tokens:** The, cat, sat, on, the, bed
2. **Stemming**: We remove the prefixes and suffixes to obtain the root word. Check the example below to see how it’s done.
   - **List of words**: Affection, Affects, Affecting, Affected, Affecting
     Root word: Affect

3. **Lemmatization**: We group together different inflected forms of a word into a base word called **lemma**. Check the example below how it's done.

   – **List of words**: going, gone, went
     Lemma: go

4. **POS tagging**: We identify the parts of speech for different tokens. Check the example below to see how it's done.

   – **Sentence:** The dog killed the bat.
     Parts of speech: Definite article, noun, verb, definite article, noun.

5. **Named entity recognition**: We classify named entities mentioned in the text into categories such as "People," "Locations," "Organizations," and so on. Check the example below to see how it's done.

   – **Text**: Google CEO Sundar Pichai resides in New York.
     Named entity recognition:
     Google — Organization
     Sundar Pichai — Person
     New York — Location

6. **Chunking**: We pick up individual pieces of information and group them into bigger pieces.

How Does Natural Language Processing (NLP) Work?

- NLP models work by finding relationships between the constituent parts of language — for example, the letters, words, and sentences found in a text dataset. NLP architectures use various methods for **data preprocessing, feature extraction, and modeling**. Some of these processes are:

- **Data preprocessing:** Before a model processes text for a specific task, the text often needs to be preprocessed to improve model performance or to turn words and characters into a format the model can understand. Data-centric AI is a growing movement that prioritizes data preprocessing. Various techniques may be used in this data preprocessing:

  - **Stemming and lemmatization**
  - **Sentence segmentation**
  - **Stop word removal**
  - **Tokenization**

- **Stemming and lemmatization**: Stemming is an informal process of converting words to their base forms using heuristic rules.

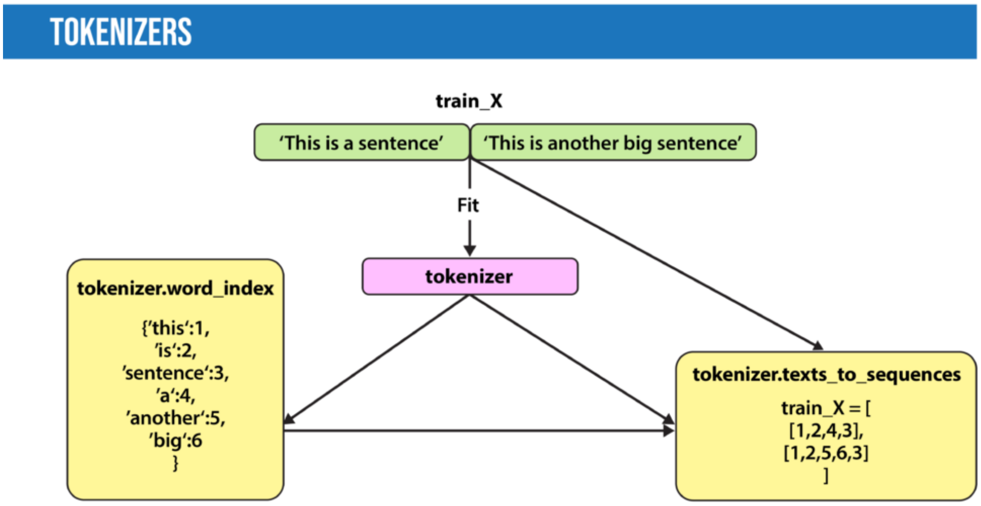
  - For example, "university," "universities," and "university's" might all be mapped to the base *universe*. (One limitation in this approach is that "universe" may also be mapped to *universe*, even though universe and university don't have a close semantic relationship.)

  - Lemmatization is a more formal way to find roots by analyzing a word's morphology using vocabulary from a dictionary.

  - Stemming and lemmatization are provided by libraries like spaCy and NLTK.

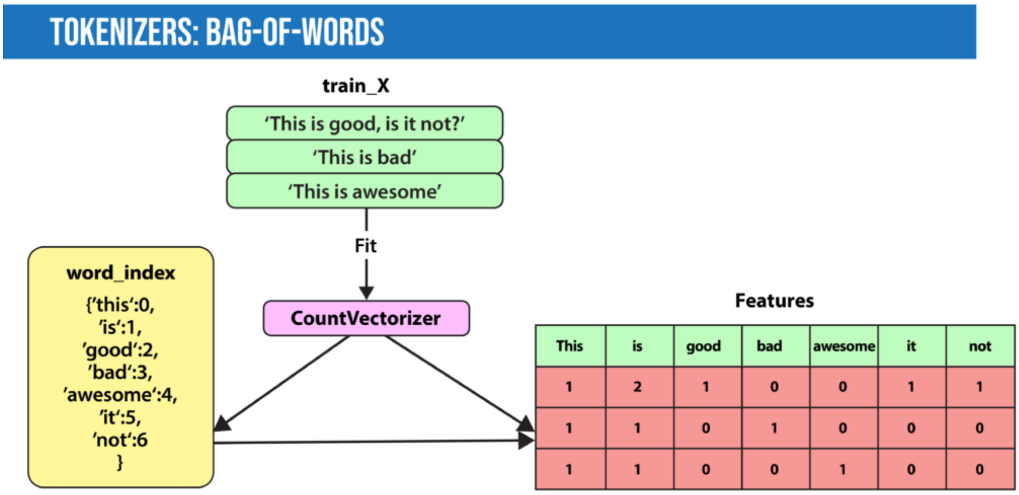
- **Sentence segmentation** breaks a large piece of text into linguistically meaningful sentence units.

- This is obvious in languages like English, where the end of a sentence is marked by a period, but it is still not trivial.
- A period can be used to mark an abbreviation as well as to terminate a sentence, and in this case, the period should be part of the abbreviation token itself.
- The process becomes even more complex in languages, such as ancient Chinese, that don't have a delimiter that marks the end of a sentence.
- **Stop word removal** aims to remove the most commonly occurring words that don't add much information to the text.
    - For example, "the," "a," "an," and so on.
- **Tokenization** splits text into individual words and word fragments.
- The result generally consists of a word index and tokenized text in which words may be represented as numerical tokens for use in various deep learning methods.
- A method that instructs language models to ignore unimportant tokens can improve efficiency.



Given a corpus of documents, a tokenizer maps every word to an index. Then it can translate any document into a sequence of numbers.

- **Feature extraction:** Most conventional machine-learning techniques work on the features – generally numbers that describe a document in relation to the corpus that contains it – created by either Bag-of-Words, TF-IDF, or generic feature engineering such as document length, word polarity, and metadata (for instance, if the text has associated tags or scores).
- More recent techniques include Word2Vec, GLoVE, and learning the features during the training process of a neural network.
- **Bag-of-Words:** Bag-of-Words counts the number of times each word or n-gram (combination of n words) appears in a document.
    - For example, below, the Bag-of-Words model creates a numerical representation of the dataset based on how many of each word in the word_index occur in the document.



TOKENIZERS: BAG-OF-WORDS

train_X

'This is good, is it not?'

'This is bad'

'This is awesome'

Fit

CountVectorizer

word_index

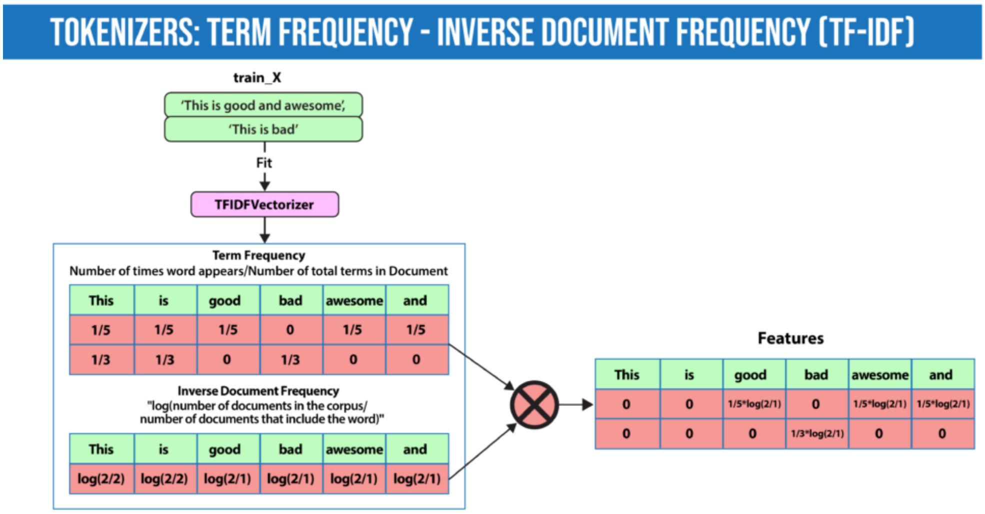
{'this':0,
'is':1,
'good':2,
'bad':3,
'awesome':4,
'it':5,
'not':6
}

Features

This	is	good	bad	awesome	it	not
1	2	1	0	0	1	1
1	1	0	1	0	0	0
1	1	0	0	1	0	0

Bag-of-Words (through the CountVectorizer method) encodes the total number of times a document uses each word in the associated corpus.

- **TF-IDF:** In Bag-of-Words, we count the occurrence of each word or n-gram in a document. In contrast, with TF-IDF, we weight each word by its importance. To evaluate a word's significance, we consider two things:
  - **Term Frequency:** How important is the word in the document?
    - *TF (word in a document) = Number of occurrences of that word in document / Number of words in document*
  - **Inverse Document Frequency:** How important is the term in the whole corpus?
    - *IDF (word in a corpus) = log (number of documents in the corpus / number of documents that include the word)*
- A word is important if it occurs many times in a document. But that creates a problem. Words like "a" and "the" appear often. And as such, their TF score will always be high. We resolve this issue by using Inverse Document Frequency, which is high if the word is rare and low if the word is common across the corpus. The TF-IDF score of a term is the product of TF and IDF.



TOKENIZERS: TERM FREQUENCY - INVERSE DOCUMENT FREQUENCY (TF-IDF)

train_X

'This is good and awesome',

'This is bad'

Fit

TFIDFVectorizer

Term Frequency

Number of times word appears/Number of total terms in Document

This	is	good	bad	awesome	and
1/5	1/5	1/5	0	1/5	1/5
1/3	1/3	0	1/3	0	0

Inverse Document Frequency

"log(number of documents in the corpus/
number of documents that include the word)"

This	is	good	bad	awesome	and
log(2/2)	log(2/2)	log(2/1)	log(2/1)	log(2/1)	log(2/1)

Features

This	is	good	bad	awesome	and
0	0	1/5*log(2/1)	0	1/5*log(2/1)	1/5*log(2/1)
0	0	0	1/3*log(2/1)	0	0

TF-IDF creates features for each document based on how often each word shows up in a document versus the entire corpus.

- **Word2Vec**, introduced in 2013, uses a vanilla neural network to learn high-dimensional word embeddings from raw text. It comes in two variations: Skip-Gram, in which we try to predict surrounding words given a target word, and Continuous Bag-of-Words (CBOW), which tries to predict the target word from surrounding words. After discarding the final layer after training, these models take a word as input and output a word embedding that can be used as an input to many NLP tasks. Embeddings from Word2Vec capture context. If particular words appear in similar contexts, their embeddings will be similar.

- **GLoVE** is similar to Word2Vec as it also learns word embeddings, but it does so by using matrix factorization techniques rather than neural learning. The GLoVE model builds a matrix based on the global word-to-word co-occurrence counts.
- **Modeling:** After data is preprocessed, it is fed into an NLP architecture that models the data to accomplish a variety of tasks.
    - Numerical features extracted by the techniques described above can be fed into various models depending on the task at hand. For example, for classification, the output from the TF-IDF vectorizer could be provided to logistic regression, naive Bayes, decision trees, or gradient boosted trees. Or, for named entity recognition, we can use hidden Markov models along with n-grams.
    - Deep neural networks typically work without using extracted features, although we can still use TF-IDF or Bag-of-Words features as an input.
    - **Language Models**: In very basic terms, the objective of a language model is to predict the next word when given a stream of input words. Probabilistic models that use Markov assumption are one example:
        - $P(W_n)=P(W_n|W_{n-1})$
- Deep learning is also used to create such language models. Deep-learning models take as input a word embedding and, at each time state, return the probability distribution of the next word as the probability for every word in the dictionary.
- Pre-trained language models learn the structure of a particular language by processing a large corpus, such as Wikipedia.
- They can then be fine-tuned for a particular task. For instance, BERT has been fine-tuned for tasks ranging from fact-checking to writing headlines.

Top Natural Language Processing (NLP) Techniques

- Most of the NLP tasks discussed above can be modeled by a dozen or so general techniques.
- It's helpful to think of these techniques in two categories: Traditional machine learning methods and deep learning methods.
    - **Traditional Machine learning NLP techniques**
        - Logistic Regression
        - Naïve Bayes
        - Decision Trees
        - Latent Dirichlet Allocation (LDA)
        - Hidden Markov models
    - **Deep learning NLP Techniques**
        - CNN
        - RNN
        - Autoencoders

Six Important Natural Language Processing (NLP) Models

- **Eliza** was developed in the mid-1960s to try to solve the Turing Test; that is, to fool people into thinking they're conversing with another human being rather than a machine.
- **Tay** was a chatbot that Microsoft launched in 2016. It was supposed to tweet like a teen and learn from conversations with real users on Twitter.
- **BERT** and his Muppet friends: Many deep learning models for NLP are named after Muppet characters, including ELMo, BERT, BigBIRD, ERNIE, Kermit, Grover, RoBERTa, and Rosita. Most of these models are good at providing contextual embeddings and enhanced knowledge representation.
- **Generative Pre-Trained Transformer 3 (GPT-3)** is a 175 billion parameter model that can write original prose with human-equivalent fluency in response to an input prompt. The model is based on the transformer architecture. The previous version, GPT-2, is open source. Microsoft acquired an exclusive license to access GPT-3's underlying model from its developer OpenAI, but other users can interact with it via an application programming interface (API). Several groups including EleutherAI and Meta have released open-source interpretations of GPT-3.
- **Language Model for Dialogue Applications (LaMDA)** is a conversational chatbot developed by Google. LaMDA is a transformer-based model trained on dialogue rather than the usual web text. The system aims to provide sensible and specific responses to conversations. Google developer Blake Lemoine came to believe that LaMDA is sentient. Lemoine had detailed conversations with AI about his rights and personhood. During one of these conversations, the AI changed Lemoine's mind about Isaac Asimov's third law of robotics. Lemoine claimed that LaMDA was sentient, but the idea was disputed by many observers and commentators. Subsequently, Google placed Lemoine on administrative leave for distributing proprietary information and ultimately fired him.
- **Mixture of Experts (MoE):** While most deep learning models use the same set of parameters to process every input, MoE models aim to provide different parameters for different inputs based on efficient routing algorithms to achieve higher performance. Switch Transformer is an example of the MoE approach that aims to reduce communication and computational costs.

NLP Libraries

- **Scikit-learn:** It provides a wide range of algorithms for building machine learning models in Python.
- **Natural language Toolkit (NLTK):** NLTK is a complete toolkit for all NLP techniques.
- **Pattern:** It is a web mining module for NLP and machine learning.
- **TextBlob:** It provides an easy interface to learn basic NLP tasks like sentiment analysis, noun phrase extraction, or pos-tagging.
- **Quepy:** Quepy is used to transform natural language questions into queries in a database query language.
- **SpaCy:** SpaCy is an open-source NLP library which is used for Data Extraction, Data Analysis, Sentiment Analysis, and Text Summarization.

- **Gensim:** Gensim works with large datasets and processes data streams.

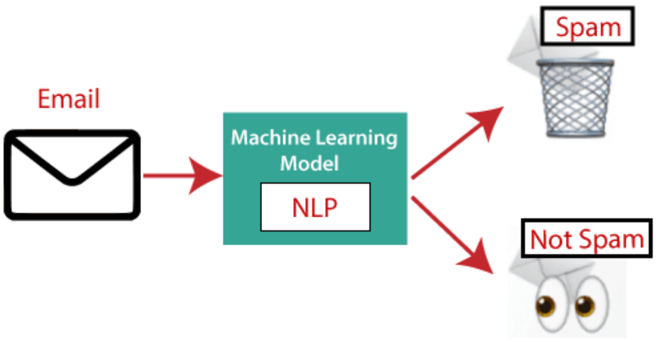
Applications of NLP

Question Answering

- Question Answering focuses on building systems that automatically answer the questions asked by humans in a natural language.

Spam detection

- Spam detection is used to detect unwanted e-mails getting to a user's inbox.

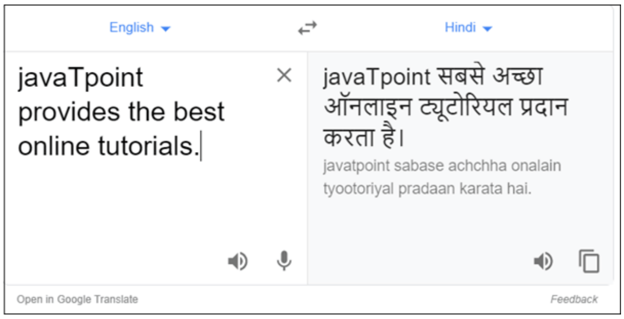


Sentiment Analysis

- Sentiment Analysis is also known as **opinion mining**. It is used on the web to analyse the attitude, behavior, and emotional state of the sender.
- This application is implemented through a combination of NLP (Natural Language Processing) and statistics by assigning the values to the text (positive, negative, or natural), identify the mood of the context (happy, sad, angry, etc.)

Machine translation

Machine translation is used to translate text or speech from one natural language to another natural language.



javaTpoint provides the best online tutorials.

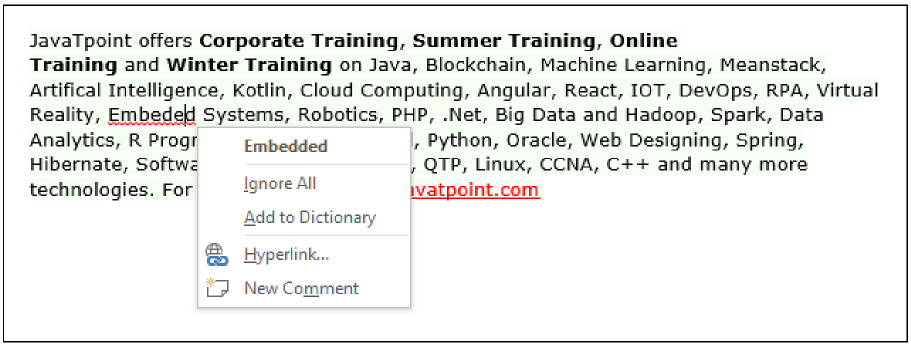
javaTpoint सबसे अच्छा ऑनलाइन ट्यूटोरियल प्रदान करता है।

javatpoint sabase achchha onalain tyootoriyal pradaan karata hai.

Example: Google Translator

Spelling correction

- Microsoft Corporation provides word processor software like MS-word, PowerPoint for the spelling correction.



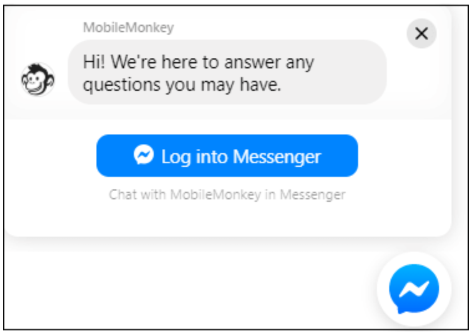
JavaTpoint offers **Corporate Training**, **Summer Training**, **Online**
Training and **Winter Training** on Java, Blockchain, Machine Learning, Meanstack,
Artifical Intelligence, Kotlin, Cloud Computing, Angular, React, IOT, DevOps, RPA, Virtual
Reality, Embeded Systems, Robotics, PHP, .Net, Big Data and Hadoop, Spark, Data
Analytics, R Progr , Python, Oracle, Web Designing, Spring,
Hibernate, Softwa , QTP, Linux, CCNA, C++ and many more
technologies. For vatpoint.com

Speech Recognition

- Speech recognition is used for converting spoken words into text. It is used in applications, such as mobile, home automation, video recovery, dictating to Microsoft Word, voice biometrics, voice user interface, and so on.

Chatbot

- Implementing the Chatbot is one of the important applications of NLP.
- It is used by many companies to provide the customer's chat services.



Hi! We're here to answer any questions you may have.

Information extraction

- Information extraction is one of the most important applications of NLP. It is used for extracting structured information from unstructured or semi-structured machine-readable documents.

Natural Language Understanding (NLU)

- It converts a large set of text into more formal representations such as first-order logic structures that are easier for the computer programs to manipulate notations of the natural language processing.

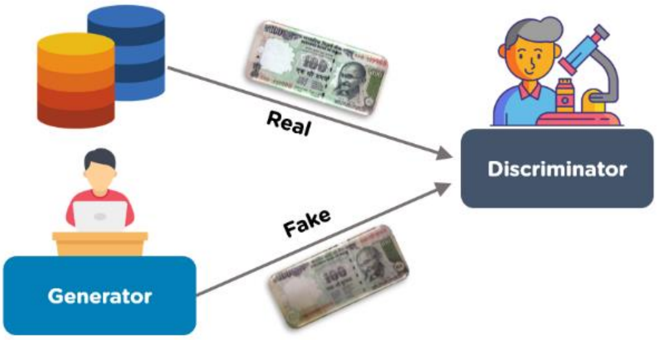
Generative Adversarial Networks (GANs)

- Generative Adversarial Networks, or GANs for short, are an approach to generative modeling using deep learning methods, such as convolutional neural networks.
- Generative modeling is an unsupervised learning task in machine learning that involves automatically discovering and learning the regularities or patterns in input data in such

a way that the model can be used to generate or output new examples that plausibly could have been drawn from the original dataset.

- GANs are a clever way of training a generative model by framing the problem as a supervised learning problem with two sub-models: the generator model that we train to generate new examples, and the discriminator model that tries to classify examples as either real (from the domain) or fake (generated).
- The two models are trained together in a zero-sum game, adversarial, until the discriminator model is fooled about half the time, meaning the generator model is generating plausible examples.
- GANs are an exciting and rapidly changing field, delivering on the promise of generative models in their ability to generate realistic examples across a range of problem domains, most notably in image-to-image translation tasks such as translating photos of summer to winter or day to night, and in generating photorealistic photos of objects, scenes, and people that even humans cannot tell are fake.

Shown below is an example of a GAN. There is a database that has real 100 rupee notes. The generator neural network generates fake 100 rupee notes. The discriminator network will help identify the real and fake notes.



The GAN model architecture involves two sub-models: a *generator model* for generating new examples and a *discriminator model* for classifying whether generated examples are real, from the domain, or fake, generated by the generator model.

- **Generator**. Model that is used to generate new plausible examples from the problem domain.
- **Discriminator**. Model that is used to classify examples as real (*from the domain*) or fake (*generated*).

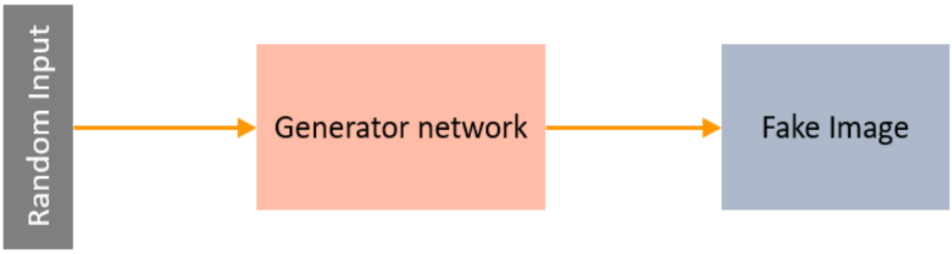
Generative adversarial networks are based on a game theoretic scenario in which the generator network must compete against an adversary. The generator network directly produces samples. Its adversary, the discriminator network, attempts to distinguish between samples drawn from the training data and samples drawn from the generator.

The GAN architecture was first described in the 2014 paper by Ian Goodfellow, et al. titled "Generative Adversarial Networks."

A standardized approach called Deep Convolutional Generative Adversarial Networks, or DCGAN, that led to more stable models was later formalized by Alec Radford, et al. in the 2015 paper titled “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.

What is a Generator?

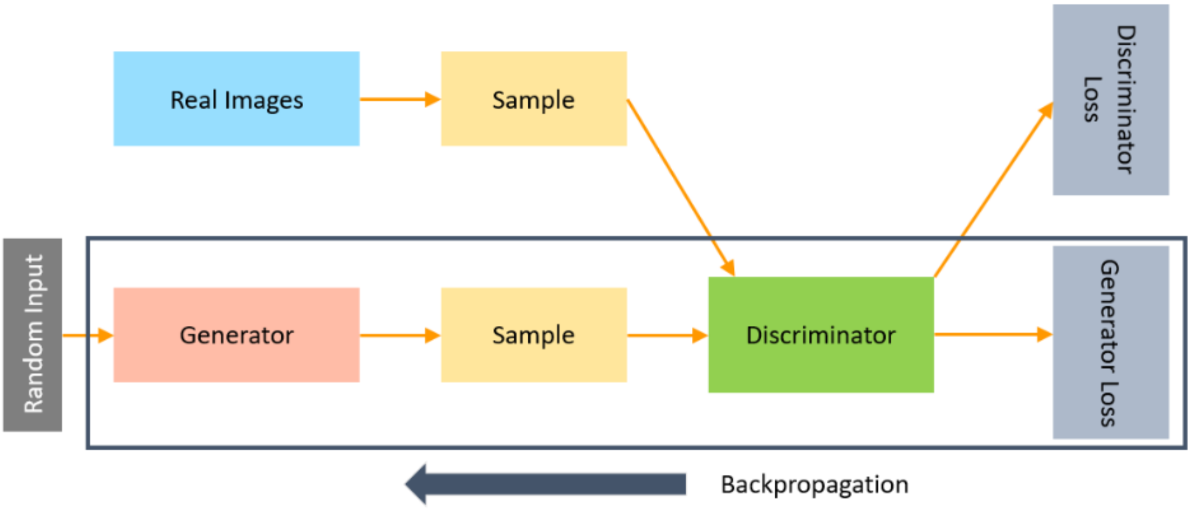
A Generator in GANs is a neural network that creates fake data to be trained on the discriminator. It learns to generate plausible data. The generated examples/instances become negative training examples for the discriminator. It takes a fixed-length random vector carrying noise as input and generates a sample.



The main aim of the Generator is to make the discriminator classify its output as real. The part of the GAN that trains the Generator includes:

- noisy input vector
- generator network, which transforms the random input into a data instance
- discriminator network, which classifies the generated data
- generator loss, which penalizes the Generator for failing to dolt the discriminator

The backpropagation method is used to adjust each weight in the right direction by calculating the weight's impact on the output. It is also used to obtain gradients and these gradients can help change the generator weights.

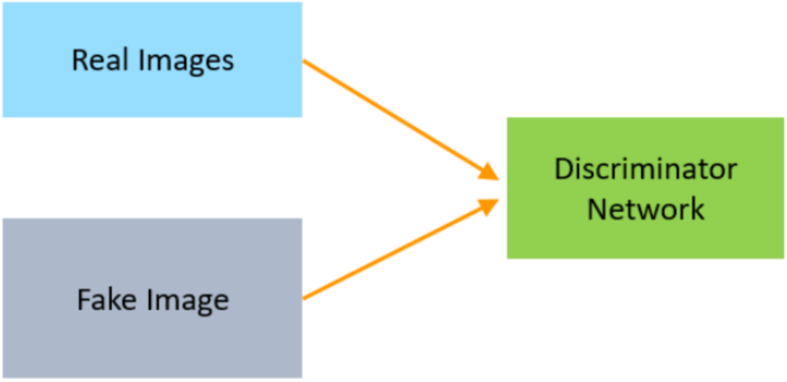


Let’s see the next topic in this article on what GANs are, i.e., a Discriminator.

What is a Discriminator?

The Discriminator is a neural network that identifies real data from the fake data created by the Generator. The discriminator's training data comes from different two sources:

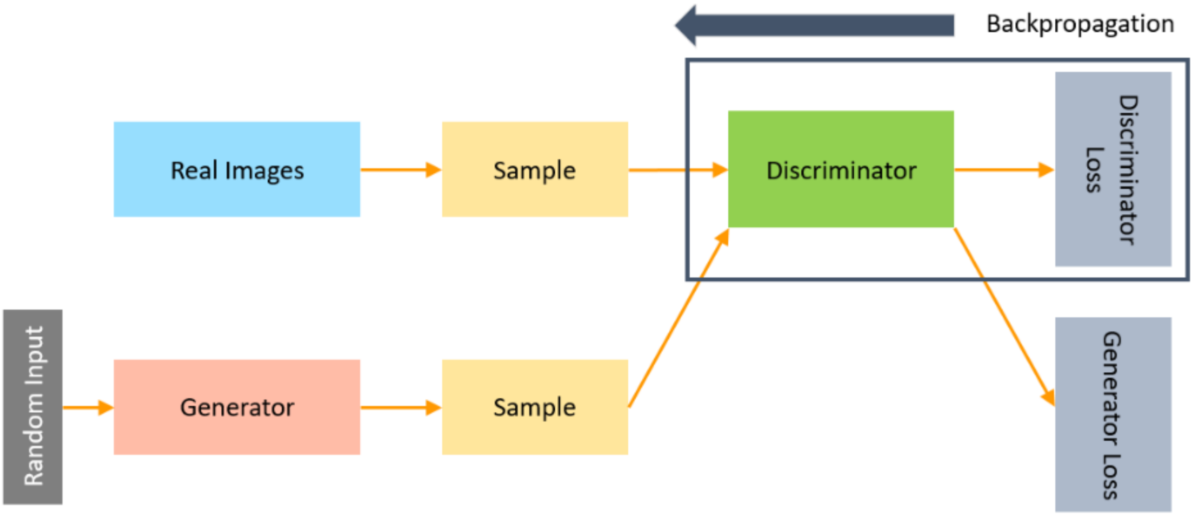
- The real data instances, such as real pictures of birds, humans, currency notes, etc., are used by the Discriminator as positive samples during training.
- The fake data instances created by the Generator are used as negative examples during the training process.



While training the discriminator, it connects to two loss functions. During discriminator training, the discriminator ignores the generator loss and just uses the discriminator loss.

In the process of training the discriminator, the discriminator classifies both real data and fake data from the generator. The discriminator loss penalizes the discriminator for misclassifying a real data instance as fake or a fake data instance as real.

The discriminator updates its weights through backpropagation from the discriminator loss through the discriminator network.

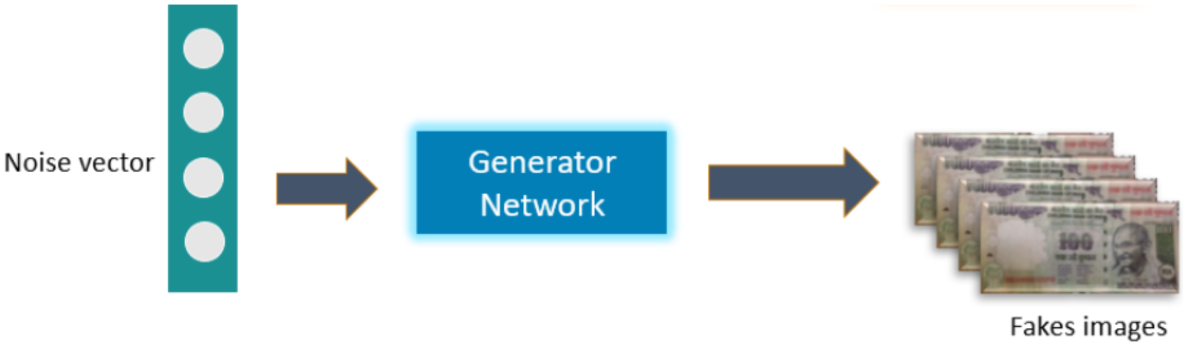


Now, let’s learn how GANs work in this article on ‘What are GANs’.

How Do GANs Work?

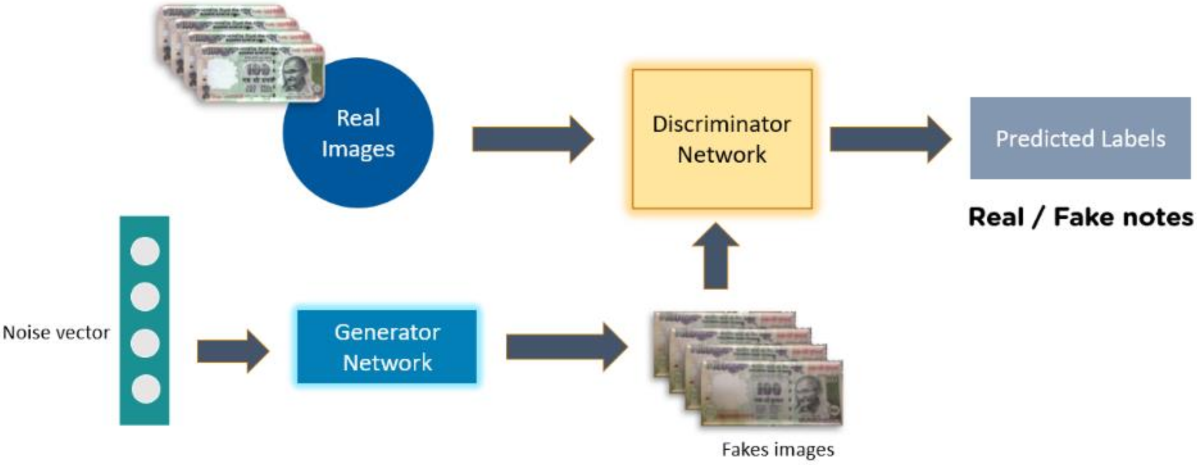
GANs consists of two neural networks. There is a Generator G(x) and a Discriminator D(x). Both of them play an adversarial game. The generator's aim is to fool the discriminator by producing data that are similar to those in the training set. The discriminator will try not to be fooled by identifying fake data from real data. Both of them work simultaneously to learn and train complex data like audio, video, or image files.

The Generator network takes a sample and generates a fake sample of data. The Generator is trained to increase the Discriminator network's probability of making mistakes.

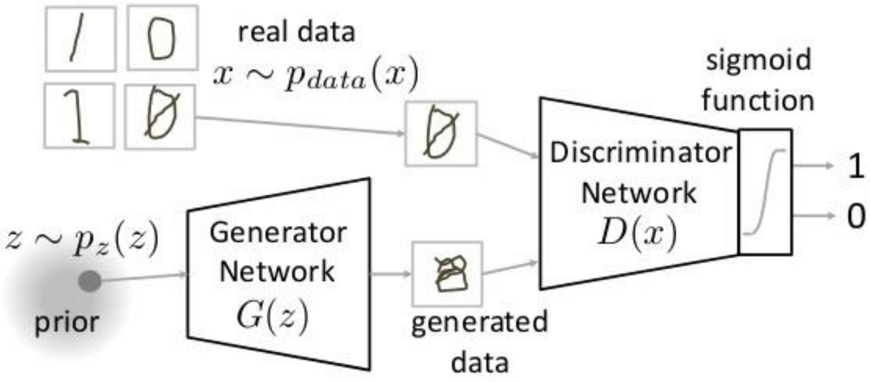


Below is an example of a GAN trying to identify if the 100 rupee notes are real or fake. So, first, a noise vector or the input vector is fed to the Generator network. The generator creates fake 100 rupee notes. The real images of 100 rupee notes stored in a database are passed to the discriminator along with the fake notes. The Discriminator then identifies the notes as classifying them as real or fake.

We train the model, calculate the loss function at the end of the discriminator network, and backpropagate the loss into both discriminator and generator models.



There are two main components of a GAN – Generator Neural Network and Discriminator Neural Network.



The Generator Network takes an random input and tries to generate a sample of data. In the above image, we can see that generator G(z) takes a input z from p(z), where z is a sample from probability distribution p(z). It then generates a data which is then fed into a discriminator network D(x). The task of Discriminator Network is to take input either from the real data or from the generator and try to predict whether the input is real or generated. It takes an input x from $p_{data}(x)$ where $p_{data}(x)$ is our real data distribution. D(x) then solves a binary classification problem using sigmoid function giving output in the range 0 to 1.

Let us define the notations we will be using to formalize our GAN,

$P_{data}(x)$ -> the distribution of real data
X -> sample from pdata(x)
P(z) -> distribution of generator
Z -> sample from p(z)
G(z) -> Generator Network
D(x) -> Discriminator Network

Now the training of GAN is done (as we saw above) as a fight between generator and discriminator. This can be represented mathematically as

$$\min_G \max_D V(D, G)$$

$$V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z))]$$

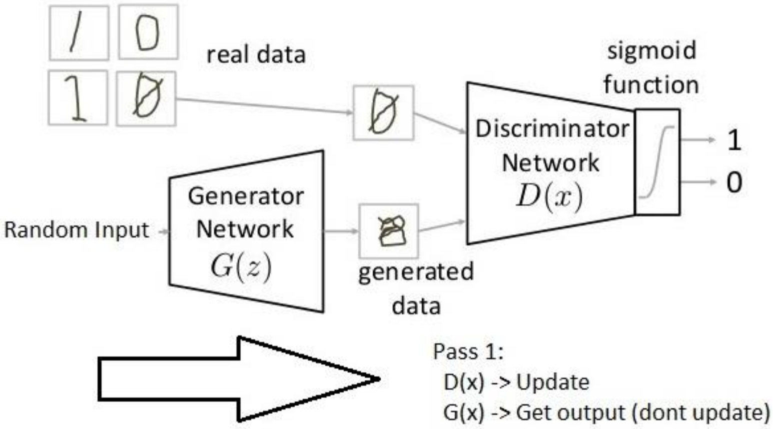
In our function V (D, G) the first term is entropy that the data from real distribution (pdata(x)) passes through the discriminator (aka best case scenario). The discriminator tries to maximize this to 1. The second term is entropy that the data from random input (p(z)) passes through the generator, which then generates a fake sample which is then passed through the discriminator to identify the fakeness (aka worst case scenario). In this term, discriminator tries to maximize it to 0 (i.e. the log probability that the data from generated is fake is equal to 0). **So overall, the discriminator is trying to maximize our function V**.

On the other hand, **the task of generator is exactly opposite, i.e. it tries to minimize the function V** so that the differentiation between real and fake data is bare minimum. This, in other words is a cat and mouse game between generator and discriminator!

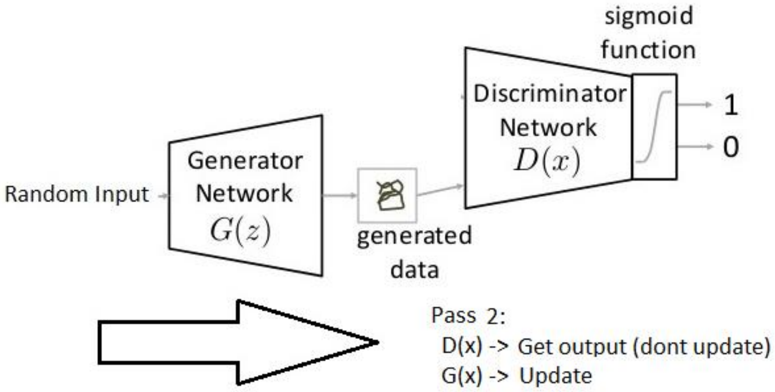
Parts of training GAN

So broadly a training phase has two main subparts and they are done sequentially

- **Pass 1**: Train discriminator and freeze generator (freezing means setting training as false. The network does only forward pass and no backpropagation is applied)



- **Pass 2:** Train generator and freeze discriminator



Steps to train a GAN

Step 1: Define the problem. Do you want to generate fake images or fake text. Here you should completely define the problem and collect data for it.

Step 2: Define architecture of GAN. Define how your GAN should look like. Should both your generator and discriminator be multi layer perceptrons, or convolutional neural networks? This step will depend on what problem you are trying to solve.

Step 3: Train Discriminator on real data for n epochs. Get the data you want to generate fake on and train the discriminator to correctly predict them as real. Here value n can be any natural number between 1 and infinity.

Step 4: Generate fake inputs for generator and train discriminator on fake data. Get generated data and let the discriminator correctly predict them as fake.

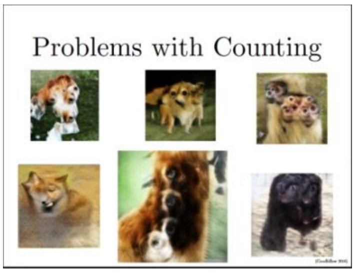
Step 5: Train generator with the output of discriminator. Now when the discriminator is trained, you can get its predictions and use it as an objective for training the generator. Train the generator to fool the discriminator.

Step 6: Repeat step 3 to step 5 for a few epochs.

Step 7: Check if the fake data manually if it seems legit. If it seems appropriate, stop training, else go to step 3. This is a bit of a manual task, as hand evaluating the data is the best way to check the fakeness. When this step is over, you can evaluate whether the GAN is performing well enough.

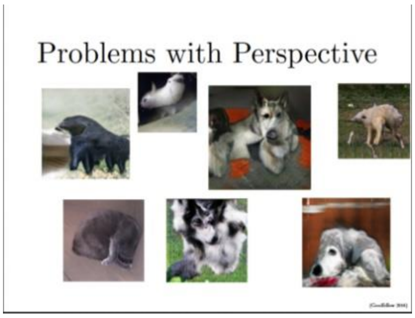
Challenges with GANs

- **Problem with Counting**: GANs fail to differentiate how many of a particular object should occur at a location. As we can see below, it gives more number of eyes in the head than naturally present.



Problems with Counting

- **Problems with Perspective:** GANs fail to adapt to 3D objects. It doesn't understand perspective, i.e.difference between frontview and backview. As we can see below, it gives flat (2D) representation of 3D objects.



Problems with Perspective

- **Problems with Global Structures:** Same as the problem with perspective, GANs do not understand a holistic structure. For example, in the bottom left image, it gives a generated image of a quadruple cow, i.e. a cow standing on its hind legs and simultaneously on all four legs. That is definitely not possible in real life!

Problems with Global Structure

(Goodfellow 2016)

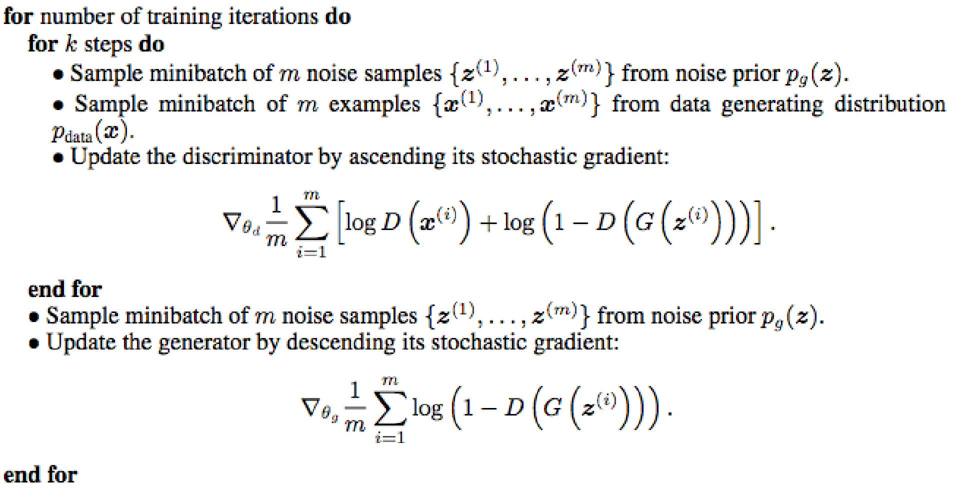
Implementing a Toy GAN

Lets see a toy implementation of GAN to strengthen our theory. We will try to generate digits by training a GAN on Identify the Digits dataset. A bit about the dataset; the dataset contains 28×28 images which are black and white. All the images are in ".png" format. For our task, we will only work on the training set.

You also need to setup the libraries , namely

- numpy
- pandas
- tensorflow
- keras
- keras_adversarial

Before starting with the code, let us understand the internal working thorugh pseudocode. A pseudocode of GAN training can be thought out as follows



```
for number of training iterations do
  for $k$ steps do
    • Sample minibatch of $m$ noise samples $\{\boldsymbol{z}^{(1)}, \dots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of $m$ examples $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:
        $\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[ \log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$
  end for
  • Sample minibatch of $m$ noise samples $\{\boldsymbol{z}^{(1)}, \dots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  • Update the generator by descending its stochastic gradient:
        $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$
end for
```

Note: This is the first implementation of GAN that was published in the paper. Numerous improvements/updates in the pseudocode can be seen in the recent papers such as adding batch normalization in the generator and discrimination network, training generator k times etc.

Now lets start with the code!

Let us first import all the modules

```
# import modules
%pylab inline


import os
import numpy as np
import pandas as pd
from scipy.misc import imread


import keras
from keras.models import Sequential
from keras.layers import Dense, Flatten, Reshape, InputLayer
from keras.regularizers import L1L2
```

To have a deterministic randomness, we set a seed value

```
# to stop potential randomness
seed = 128
rng = np.random.RandomState(seed)
```

We set the path of our data and working directory

```
# set path
root_dir = os.path.abspath('.')
data_dir = os.path.join(root_dir, 'Data')
```

Let us load our data

```
# load data
train = pd.read_csv(os.path.join(data_dir, 'Train', 'train.csv'))
test = pd.read_csv(os.path.join(data_dir, 'test.csv'))


temp = []
for img_name in train.filename:
    image_path = os.path.join(data_dir, 'Train', 'Images', 'train', img_name)
    img = imread(image_path, flatten=True)
    img = img.astype('float32')
    temp.append(img)


train_x = np.stack(temp)
train_x = train_x / 255.
```

To visualize what our data looks like, let us plot one of the image

```
# print image
img_name = rng.choice(train.filename)
filepath = os.path.join(data_dir, 'Train', 'Images', 'train', img_name)

img = imread(filepath, flatten=True)

pylab.imshow(img, cmap='gray')
pylab.axis('off')
pylab.show()
```

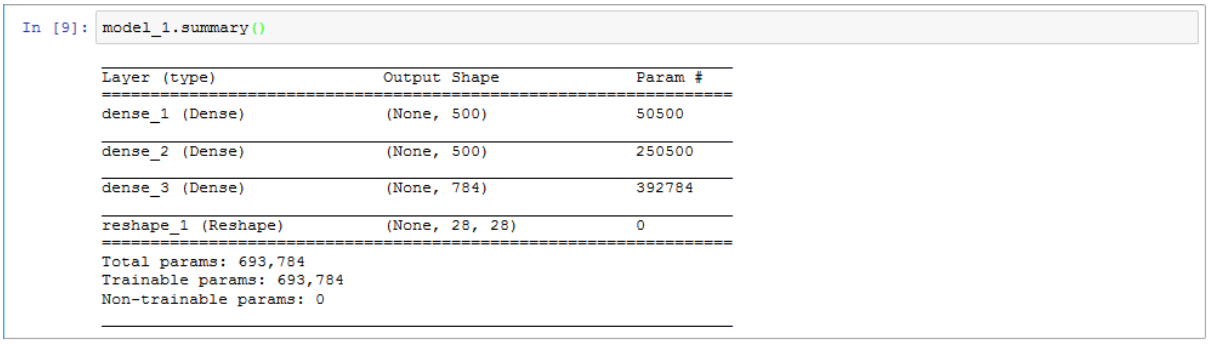
Define variables which we will be using later

```
# define variables
# define vars g_input_shape = 100 d_input_shape = (28, 28) hidden_1_num_units = 500
hidden_2_num_units = 500 g_output_num_units = 784 d_output_num_units = 1 epochs = 25
batch_size = 128
```

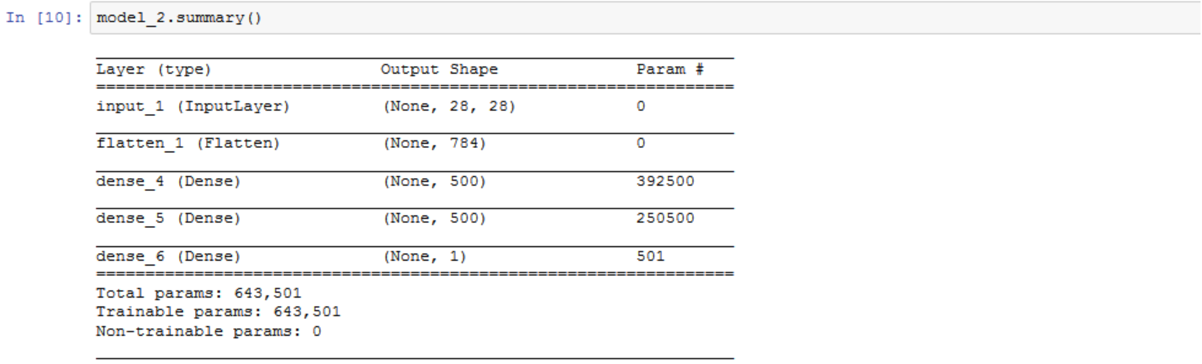
Now define our generator and discriminator networks

```
# generator
model_1 = Sequential([
    Dense(units=hidden_1_num_units, input_dim=g_input_shape, activation='relu',
kernel_regularizer=L1L2(1e-5, 1e-5)),
    Dense(units=hidden_2_num_units, activation='relu', kernel_regularizer=L1L2(1e-5, 1e-
5)),
    Dense(units=g_output_num_units, activation='sigmoid', kernel_regularizer=L1L2(1e-5,
1e-5)),
    Reshape(d_input_shape),
])

# discriminator
model_2 = Sequential([
    InputLayer(input_shape=d_input_shape),
    Flatten(),
    Dense(units=hidden_1_num_units, activation='relu', kernel_regularizer=L1L2(1e-5, 1e-5)),
    Dense(units=hidden_2_num_units, activation='relu', kernel_regularizer=L1L2(1e-5, 1e-5)),
    Dense(units=d_output_num_units, activation='sigmoid', kernel_regularizer=L1L2(1e-5, 1e-5)),
])
```

Here is the architecture of our networks



```
In [9]: model_1.summary()

_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_1 (Dense)              (None, 500)               50500     
_________________________________________________________________
dense_2 (Dense)              (None, 500)               250500    
_________________________________________________________________
dense_3 (Dense)              (None, 784)               392784    
_________________________________________________________________
reshape_1 (Reshape)          (None, 28, 28)            0         
=================================================================
Total params: 693,784
Trainable params: 693,784
Non-trainable params: 0
_________________________________________________________________
```



```
In [10]: model_2.summary()

_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_1 (InputLayer)         (None, 28, 28)            0         
_________________________________________________________________
flatten_1 (Flatten)          (None, 784)               0         
_________________________________________________________________
dense_4 (Dense)              (None, 500)               392500    
_________________________________________________________________
dense_5 (Dense)              (None, 500)               250500    
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 501       
=================================================================
Total params: 643,501
Trainable params: 643,501
Non-trainable params: 0
_________________________________________________________________
```

We will then define our GAN, for that we will first import a few important modules

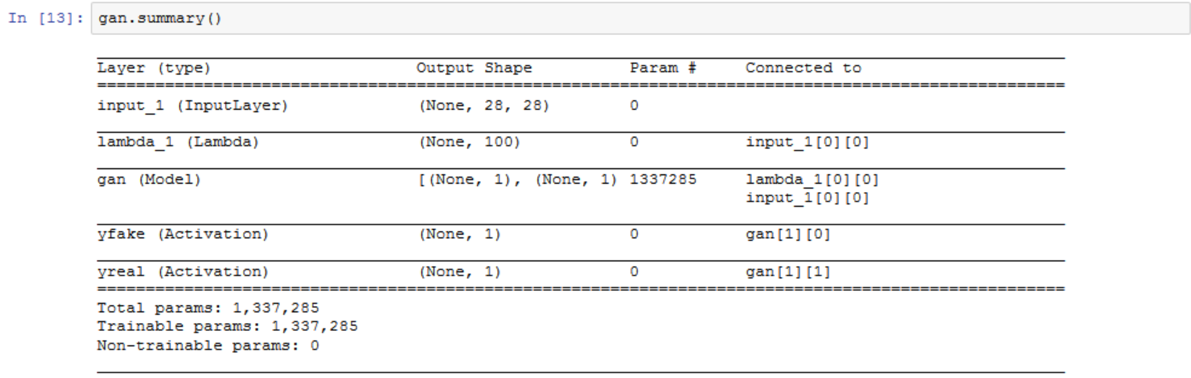
```
from keras_adversarial import AdversarialModel, simple_gan, gan_targets
from keras_adversarial import AdversarialOptimizerSimultaneous, normal_latent_sampling
```

Let us compile our GAN and start the training

```
gan = simple_gan(model_1, model_2, normal_latent_sampling((100,)))
model = AdversarialModel(base_model=gan,player_params=[model_1.trainable_weights, model_2.trainable_weights])
model.adversarial_compile(adversarial_optimizer=AdversarialOptimizerSimultaneous(), player_optimizers=['adam', 'adam'], loss='binary_crossentropy')
```

```
history = model.fit(x=train_x, y=gan_targets(train_x.shape[0]), epochs=10,
batch_size=batch_size)
```

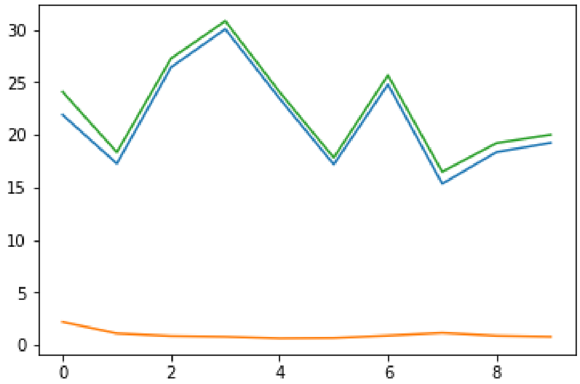
Here’s how our GAN would look like,



```
In [13]: gan.summary()

Layer (type)                    Output Shape          Param #     Connected to
==================================================================================================
input_1 (InputLayer)            (None, 28, 28)        0
__________________________________________________________________________________________________
lambda_1 (Lambda)               (None, 100)           0           input_1[0][0]
__________________________________________________________________________________________________
gan (Model)                     [(None, 1), (None, 1) 1337285     lambda_1[0][0]
                                                                  input_1[0][0]
__________________________________________________________________________________________________
yfake (Activation)              (None, 1)             0           gan[1][0]
__________________________________________________________________________________________________
yreal (Activation)              (None, 1)             0           gan[1][1]
==================================================================================================
Total params: 1,337,285
Trainable params: 1,337,285
Non-trainable params: 0
__________________________________________________________________________________________________
```

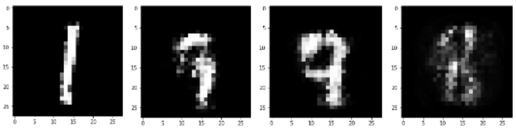
We get a graph like after training for 10 epochs.

```
plt.plot(history.history['player_0_loss'])
plt.plot(history.history['player_1_loss'])
plt.plot(history.history['loss'])
```



After training for 100 epochs, I got the following generated images

```
zsamples = np.random.normal(size=(10, 100))
pred = model_1.predict(zsamples)
for i in range(pred.shape[0]):
    plt.imshow(pred[i, :], cmap='gray')
    plt.show()
```



Applications of Generative Adversarial Networks (GANs)

1. Generate new data from available data – It means generating new samples from an available sample that is not similar to a real one.
2. Generate realistic pictures of people that have never existed.
3. Gans is not limited to Images, It can generate text, articles, songs, poems, etc.
4. Generate Music by using some clone Voice – If you provide some voice then GANs can generate a similar clone feature of it. In this research paper, researchers from NIT in Tokyo proposed a system that is able to generate melodies from lyrics with help of learned relationships between notes and subjects.
5. Text to Image Generation (Object GAN and Object Driven GAN)
6. Creation of anime characters in Game Development and animation production.
7. Image to Image Translation – We can translate one Image to another without changing the background of the source image. For example, Gans can replace a dog with a cat.
8. Low resolution to High resolution – If you pass a low-resolution Image or video, GAN can produce a high-resolution Image version of the same.
9. Prediction of Next Frame in the video – By training a neural network on small frames of video, GANs are capable to generate or predict a small next frame of video. For example, you can have a look at below GIF
10. Interactive Image Generation – It means that GANs are capable to generate images and video footage in an art form if they are trained on the right real dataset.
11. Speech – Researchers from the College of London recently published a system called GAN-TTS that learns to generate raw audio through training on 567 corpora of speech data.

Deep Reinforcement Learning

Why Deep Reinforcement Learning?

To understand Deep Reinforcement Learning better, imagine that you want your computer to play chess with you. The first question to ask is this:

Would it be possible if the machine was trained in a supervised fashion?

In theory, yes. But—

There are two drawbacks that you need to consider.

- Firstly, to move forward with supervised learning you need a relevant dataset.
- Secondly, if we are training the machine to replicate human behavior in the game of chess, the machine would never be better than the human, because it’s simply replicating the same behavior.

So, by definition, we cannot use supervised learning to train the machine.

But is there a way to have an agent play a game entirely by itself?

Yes, that’s where Reinforcement Learning comes into play.

Deep Reinforcement Learning

- Reinforcement Learning is a type of machine learning algorithm that learns to solve a multi-level problem by trial and error. The machine is trained on real-life scenarios to make a sequence of decisions. It receives either rewards or penalties for the actions it performs. Its goal is to maximize the total reward.
- By Deep Reinforcement Learning we mean multiple layers of Artificial Neural Networks that are present in the architecture to replicate the working of a human brain**.**

Reinforcement Learning Definitions

- **Agent -** Agent (A) takes actions that affect the environment. Citing an example, the machine learning to play chess is the *agent***.**
- **Action -** It is the set of all possible operations/moves the agent can make. The agent makes a decision on which action to take from a set of discrete actions (a).

- **Environment -** All actions that the reinforcement learning agent makes directly affect the environment. Here, the board of chess is the environment. The environment takes the agent's present state and action as information and returns the reward to the agent with a new state.
- For example, the move made by the bot will either have a negative/positive effect on the whole game and the arrangement of the board. This will decide the next action and state of the board**.**
- **State -** A state (S) is a particular situation in which the agent finds itself.
- **Reward (R) -** The environment gives feedback by which we determine the validity of the agent's actions in each state. It is crucial in the scenario of Reinforcement Learning where we want the machine to learn all by itself and the only critic that would help it in learning is the feedback/reward it receives.
- For example, in a chess game scenario it happens when the bot takes the place of an opponent's piece and later captures it.
- **Discount factor -** Over time, the discount factor modifies the importance of incentives. Given the uncertainty of the future it's better to add variance to the value estimates. Discount factor helps in reducing the degree to which future rewards affect our value function estimates.
- **Policy (π) -** It decides what action to take in a certain state to maximize the reward.
- **Value (V)**—It measures the optimality of a specific state. It is the expected discounted rewards that the agent collects following the specific policy.
- **Q-value or action-value -** Q Value is a measure of the overall expected reward if the agent (A) is in state (s) and takes action (a), and then plays until the end of the episode according to some policy (π).

Model-based vs Model-free learning algorithms

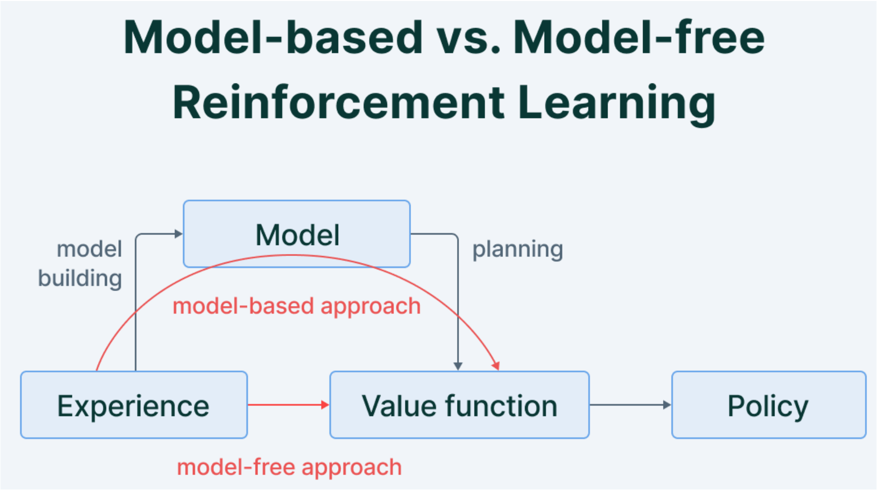
- There are two main types of Reinforcement Learning algorithms:
- 1. Model-based algorithms
- 2. Model-free algorithms

Model-based algorithms

- Model-based algorithm use the transition and reward function to estimate the optimal policy.
  - They are used in scenarios where we have complete knowledge of the environment and how it reacts to different actions.
  - In Model-based Reinforcement Learning the agent has access to the model of the environment i.e., action required to be performed to go from one state to another, probabilities attached, and corresponding rewards attached.
  - They allow the reinforcement learning agent to plan ahead by thinking ahead.
  - For static/fixed environments, Model-based Reinforcement Learning is more suitable.

Model-free algorithms

- Model-free algorithms find the optimal policy with very limited knowledge of the dynamics of the environment. They do no thave any transition/reward function to judge the best policy.
  - They estimate the optimal policy directly from experience i.e., interaction between agent and environment without having any hint of the reward function.
  - Model-free Reinforcement Learning should be applied in scenarios involving incomplete information of the environment.
  - In real-world, we don't have a fixed environment. Self-driving cars have a dynamic environment with changing traffic conditions, route diversions etc. In such scenarios, Model-free algorithms outperform other techniques

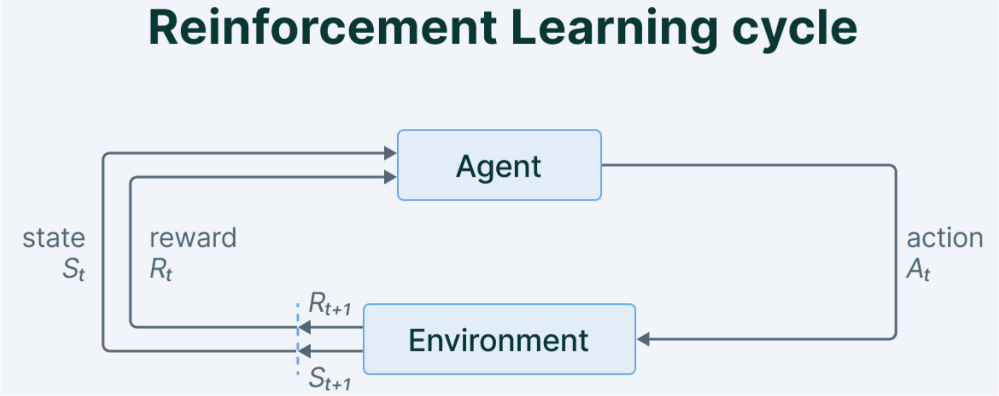


Common mathematical and algorithmic frameworks

- Markov Decision Process (MDP)
- Bellman Equations
- Dynamic Programming
- Value iteration
- Policy iteration
- Q-learning

Markov Decision Process (MDP)

- Markov Decision Process is a Reinforcement Learning algorithm that gives us a way to formalize sequential decision making.
- This formalization is the basis to the problems that are solved by Reinforcement Learning. The components involved in a **Markov Decision Process (MDP)** is a decision maker called an agent that interacts with the environment it is placed in.
- These interactions occur sequentially overtime.
- In each timestamp, the agent will get some representation of the environment state. Given this representation, the agent selects an action to make. The environment is then transitioned into some new state and the agent is given a reward as a consequence of its previous action.
- Let's wrap up everything that we have covered till now.
- The process of selecting an action from a given state, transitioning to a new state and receiving a reward happens sequentially over and over again. This creates something called a *trajectory* that shows the sequence of states, actions and rewards.
- Throughout the process, it is the responsibility of the reinforcement learning agent to maximize the total amount of rewards that it received from taking actions in given states of environments.
- The agent not only wants to maximize the immediate rewards but the cumulative reward it receives in the whole process.



- An important point to note about the Markov Decision Process is that it does not worry about the immediate reward but aims to maximize the total reward of the entire trajectory.
- Sometimes, it might prefer to get a small reward in the next timestamp to get a higher reward eventually over time.

Bellman Equations

- ➔ **State** is a numerical representation of what an agent observes at a particular point in an environment.
- ➔ **Action** is the input the agent is giving to the environment based on a policy.
- ➔ **Reward** is a feedback signal from the environment to the reinforcement learning agent reflecting how the agent has performed in achieving the goal.
- Bellman Equations aim to answer these questions:
  - *The agent is currently in a given state 's'. Assuming that we take best possible actions in all subsequent timestamps, what long-term reward the agent can expect?*

    or
  - *What is the value of the state the agent is currently in?*

- Bellman Equations are a class of Reinforcement Learning algorithms that are used particularly for deterministic environments.
- The value of a given state (s) is determined by taking a maximum of the actions we can take in the state the agent is in. The aim of the agent is to pick the action that is going to maximize the value.
- Therefore, it needs to take the addition of the reward of the optimal action ‘a’ in state ‘s' and add a multiplier ‘γ’ that is the discount factor which diminishes its reward over time. Every time the agent takes an action it gets back to the next state ‘s'.

$$V(s) = max_a(R(s,a)+\gamma V(s'))$$

- Rather than summing over numerous time steps, this equation simplifies the computation of the value function, allowing us to find the best solution to a complex problem by breaking it down into smaller, recursive subproblems.

Dynamic Programming

- In Bellman Optimality Equations if we have large state spaces, it becomes extremely difficult and close to impossible to solve this system of equations explicitly.
- Hence, we shift our approach from recursion to Dynamic Programming.
- Dynamic Programming is a method of solving problems by breaking them into simpler sub-problems. In Dynamic Programming, we are going to create a lookup table to estimate the value of each state.
- There are two classes of Dynamic Programming:
  - 1. Value Iteration
  - 2. Policy Iteration

- **Value iteration**
  - In this method, the optimal policy (optimal action for a given state) is obtained by choosing the action that maximizes optimal state-value function for the given state.
  - The optimal state-value function is obtained using an iterative function and hence its name—Value Iteration.
  - By iteratively improving the estimate of V,the Value Iteration method computes the ideal state value function (s). V (s) is initialized with arbitrary random values by the algorithm. The Q (s, a) and V (s) values are updated until they converge. Value Iteration is guaranteed to get you to the best results.

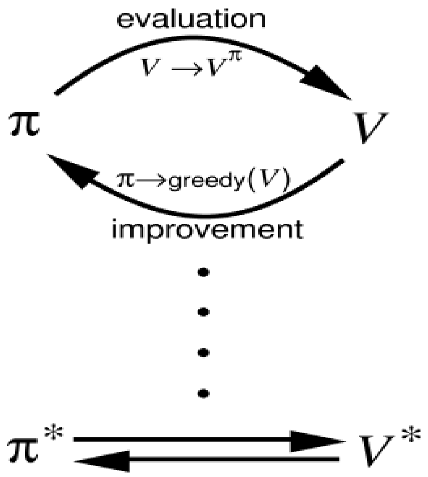
- **Policy iteration**

This algorithm has two phases in its working:

  - 1. **Policy Evaluation**—It computes the values for the states in the environment using the policy provided by the policy improvement phase.
  - 2. **Policy Improvement**—Looking into the state values provided by the policy evaluation part, it improves the policy so that it can get higher state values.

- Firstly, the reinforcement learning agents tarts with a random policy π (i). Policy Evaluation will evaluate the value functions like state values for that particular policy.
- The policy improvement will improve the policy and give us π (1) and so on until we get the optimal policy where the algorithm stops. This algorithm communicates back and forth between the two phases—Policy Improvement gives the policy to the policy evaluation module which computes values.

- Later, looking at the computed policy, policy evaluation improves the policy and iterates this process.

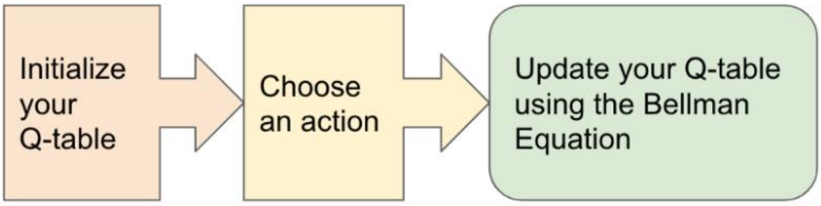


Q-learning

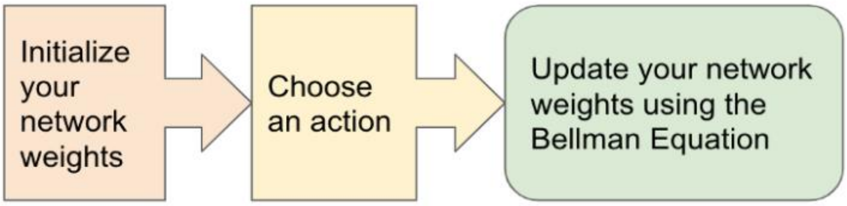
- Q-Learning combines the policy and value functions, and it tells us jointly how useful a given action is in gaining some future reward.
- Quality is assigned to a state-action pair as Q (s,a) based on the future value that it expects given the current state and best possible policy the agent has. Once the agent learns this Q-Function, it looks for the best possible action at a particular state (s) that yields the highest quality.
- Once we have an optimal Q-function (Q*), we can determine the optimal policy by applying a Reinforcement Learning algorithm to find an action that maximizes the value for each state.

$$q_* (s, a) = \max_{\pi} q_\pi (s, a)$$

- In other words, Q* gives the largest expected return achievable by any policy π for each possible state-action pair.



- In the basic Q-Learning approach, we need to maintain a look-up table called q-map for each state-action pair and the corresponding value associated with it.


Initialize your network weights
Choose an action
Update your network weights using the Bellman Equation

Applications of deep Reinforcement Learning

- Industrial manufacturing
- Self-driving cars
- Trading and Finance

- Natural Language Processing
- Healthcare

Industrial manufacturing

- Deep Reinforcement Learning is very commonly applied in Robotics.
- The actions that the robot has to take are inherently sequential. Agents learn to interact with dynamic changing environments and thus find applications in industrial automation and manufacturing.
- Labor expenses, product faults and unexpected downtime are being reduced with significant improvement in transition times and production speed.

Self-driving cars

- Machine Learning technologies power self-driving cars.
- Autonomous vehicle used large amounts of visual data and leveraged image processing capabilities in cohesion with Neural Network architecture.
- The algorithms learn to recognize pedestrians, roads, traffic, detect street signs in the environment and act accordingly. It is trained in complex scenarios and trained to excel in decision making skills in scenarios involving minimal human loss, best route to follow etc.

Trading and Finance

- We have seen how supervised learning and time-series analysis helps in prediction of the stock market. But none helps us in making decisions of what to do in a particular situation.
- An RL agent can select whether to hold, buy, or sell a share. To guarantee that the RL model is working optimally, it is assessed using market benchmark standards.

Natural Language Processing

- Reinforced Learning is expanding in wings and has conquered NLP too. Different NLP tasks like question-answering, summarization, chatbot implementation can be done by a Reinforcement Learning agent.
- Virtual Bots are trained to mimic conversations. Sequences with crucial conversation properties including coherence, informativity, and simplicity of response are rewarded using policy gradient approaches.

Healthcare

- Reinforced Learning in healthcare is an area of continuous research. Bots equipped with biological information are extensively trained to perform surgeries that require precision.
- RL bots help in better diagnosis of diseases and predict the onset of disease if the treatment is delayed and so on.

Autoencoder

- An **autoencoder** is a type of artificial neural network used to learn efficient data coding's in an unsupervised manner.
- The goal of an autoencoder is to:

- learn a representation for a set of data, usually for dimensionality reduction by training the network to ignore signal noise.
- Along with the reduction side, a reconstructing side is also learned, where the autoencoder tries to generate from the reduced encoding a representation as close as possible to its original input. This helps autoencoders to learn important features present in the data.

- Recently, the autoencoder concept has become more widely used for learning generative models of data.

Types of Autoencoders

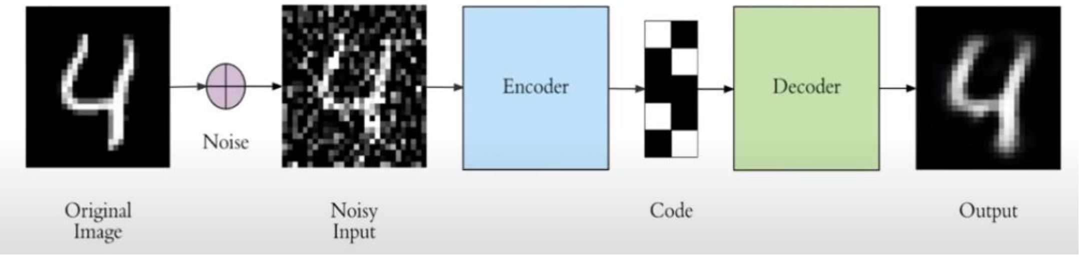
- Denoising autoencoder
- Sparse Autoencoder
- Deep Autoencoder
- Contractive Autoencoder
- Undercomplete Autoencoder
- Convolutional Autoencoder
- Variational Autoencoder

1) Denoising Autoencoder

- Denoising autoencoders create a corrupted copy of the input by introducing some noise. This helps to avoid the autoencoders to copy the input to the output without learning features about the data.

- These autoencoders take a partially corrupted input while training to recover the original undistorted input.

- The model learns a vector field for mapping the input data towards a lower dimensional manifold which describes the natural data to cancel out the added noise.



Advantages:

- It was introduced to achieve good representation. Such a representation is one that can be obtained robustly from a corrupted input and that will be useful for recovering the corresponding clean input.
- Corruption of the input can be done randomly by making some of the input as zero. Remaining nodes copy the input to the noised input.
- Minimizes the loss function between the output node and the corrupted input.
- Setting up a single-thread denoising autoencoder is easy.

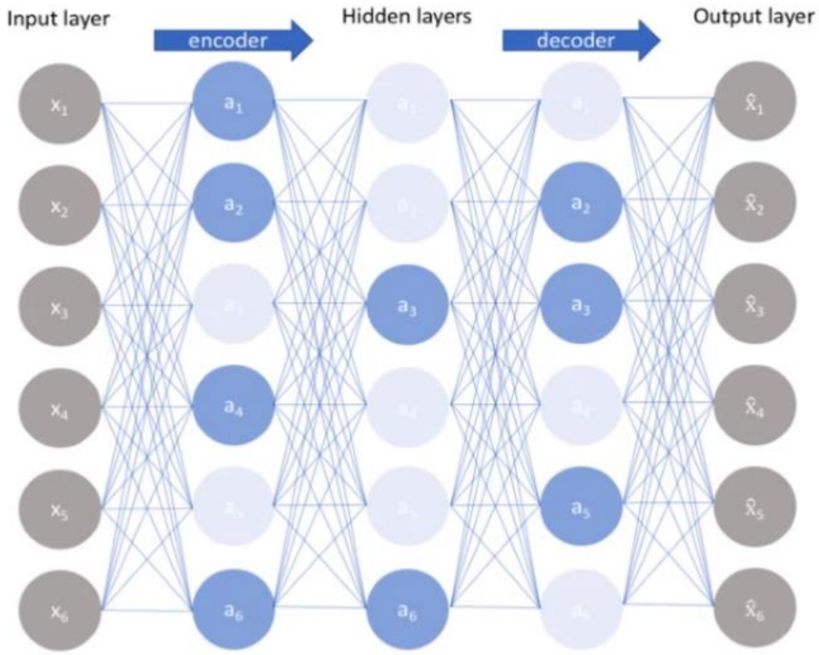
Drawbacks:

- To train an autoencoder to denoise data, it is necessary to perform preliminary stochastic mapping in order to corrupt the data and use as input.

- This model isn't able to develop a mapping which memorizes the training data because our input and target output are no longer the same.

2) Sparse Autoencoder

- Sparse autoencoders have hidden nodes greater than input nodes. They can still discover important features from the data.
- A generic sparse autoencoder is visualized where the obscurity of a node corresponds with the level of activation.
- Sparsity constraint is introduced on the hidden layer. This is to prevent output layer copy input data.
- Sparsity may be obtained by additional terms in the loss function during the training process, either by comparing the probability distribution of the hidden unit activations with some low desired value, or by manually zeroing all but the strongest hidden unit activations.



Advantages:

- Sparse autoencoders have a sparsity penalty, a value close to zero but not exactly zero. Sparsity penalty is applied on the hidden layer in addition to the reconstruction error. This prevents overfitting.
- They take the highest activation values in the hidden layer and zero out the rest of the hidden nodes. This prevents autoencoders to use all of the hidden nodes at a time and forcing only a reduced number of hidden nodes to be used.

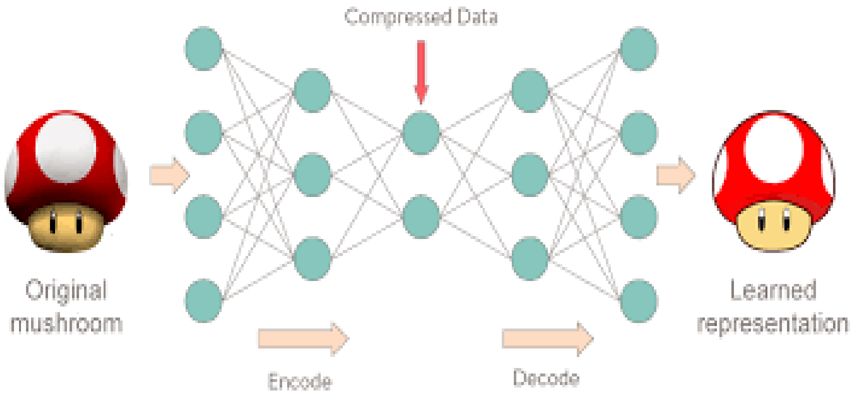
Drawbacks:

- For it to be working, it's essential that the individual nodes of a trained model which activate are data dependent, and that different inputs will result in activations of different nodes through the network.

3) Deep Autoencoder

- Deep Autoencoders consist of two identical deep belief networks,

  - One network for encoding and another for decoding.
- Typically, deep autoencoders have 4 to 5 layers for encoding and the next 4 to 5 layers for decoding.
- We use unsupervised layer by layer pre-training for this model. The layers are Restricted Boltzmann Machines which are the building blocks of deep-belief networks.



- Processing the benchmark dataset MNIST, a deep autoencoder would use binary transformations after each RBM.
- Deep autoencoders are useful in topic modeling, or statistically modeling abstract topics that are distributed across a collection of documents.
- They are also capable of compressing images into 30 number vectors.

Advantages:

- Deep autoencoders can be used for other types of datasets with real-valued data, on which you would use Gaussian rectified transformations for the RBMs instead.
- Final encoding layer is compact and fast.

Disadvantages:

- Chances of overfitting to occur since there's more parameters than input data.
- Training the data maybe a nuance since at the stage of the decoder’s backpropagation, the learning rate should be lowered or made slower depending on whether binary or continuous data is being handled.

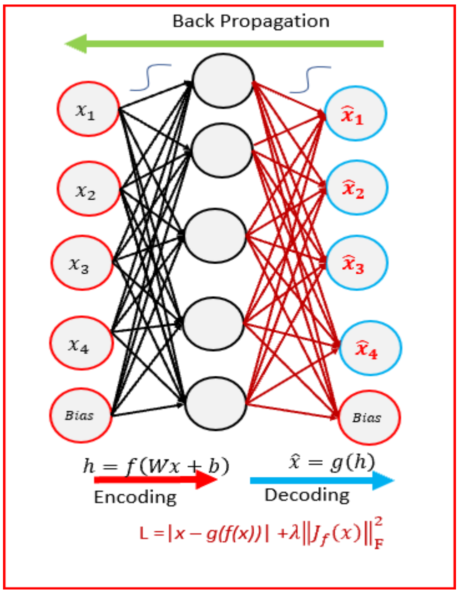
4) Contractive Autoencoder

- The objective of a contractive autoencoder is to have a robust learned representation which is less sensitive to small variation in the data.

- Robustness of the representation for the data is done by applying a penalty term to the loss function.

- Contractive autoencoder is another regularization technique just like sparse and denoising autoencoders. However, this regularizer corresponds to the Frobenius norm of the Jacobian matrix of the encoder activations with respect to the input.

- Frobenius norm of the Jacobian matrix for the hidden layer is calculated with respect to input and it is basically the sum of square of all elements.



Advantages:

- Contractive autoencoder is a better choice than denoising autoencoder to learn useful feature extraction.
- This model learns an encoding in which similar inputs have similar encodings. Hence, we're forcing the model to learn how to contract a neighborhood of inputs into a smaller neighborhood of outputs.

5) Undercomplete Autoencoder

- The objective of undercomplete autoencoder is to capture the most important features present in the data.
- Undercomplete autoencoders have a smaller dimension for hidden layer compared to the input layer. This helps to obtain important features from the data.
- It minimizes the loss function by penalizing the g(f(x)) for being different from the input x.

Advantages:

- Undercomplete autoencoders do not need any regularization as they maximize the probability of data rather than copying the input to the output.

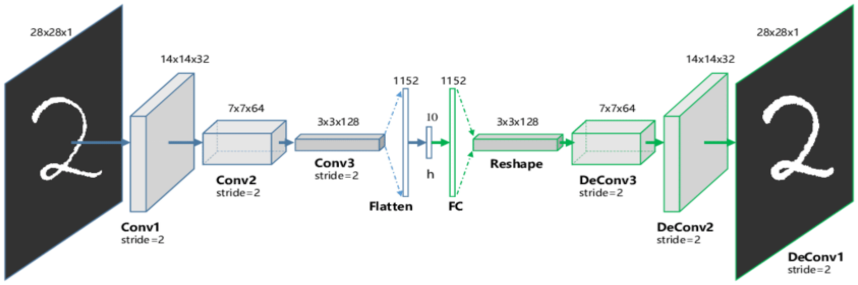
Drawbacks:

- Using an overparameterized model due to lack of sufficient training data can create overfitting.

6) Convolutional Autoencoder

- Autoencoders in their traditional formulation does not take into account the fact that a signal can be seen as a sum of other signals.
- Convolutional Autoencoders use the convolution operator to exploit this observation.
- They learn to encode the input in a set of simple signals and then try to reconstruct the input from them, modify the geometry or the reflectance of the image.

- They are the state-of-art tools for unsupervised learning of convolutional filters.
- Once these filters have been learned, they can be applied to any input in order to extract features. These features, then, can be used to do any task that requires a compact representation of the input, like classification.



Advantages:

- Due to their convolutional nature, they scale well to realistic-sized high dimensional images.
- Can remove noise from picture or reconstruct missing parts.

Drawbacks:

- The reconstruction of the input image is often blurry and of lower quality due to compression during which information is lost.

7) Variational Autoencoder

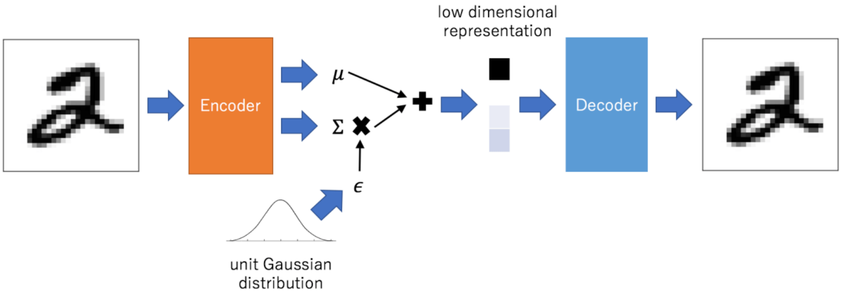
- Variational autoencoder models make strong assumptions concerning the distribution of latent variables. They use a variational approach for latent representation learning, which results in an additional loss component and a specific estimator for the training algorithm called the Stochastic Gradient Variational Bayes estimator.
- It assumes that the data is generated by a directed graphical model and that the encoder is learning an approximation to the posterior distribution where Φ and θ denote the parameters of the encoder (recognition model) and decoder (generative model) respectively. The probability distribution of the latent vector of a variational autoencoder typically matches that of the training data much closer than a standard autoencoder.

Advantages:

- It gives significant control over how we want to model our latent distribution unlike the other models.
- After training you can just sample from the distribution followed by decoding and generating new data.

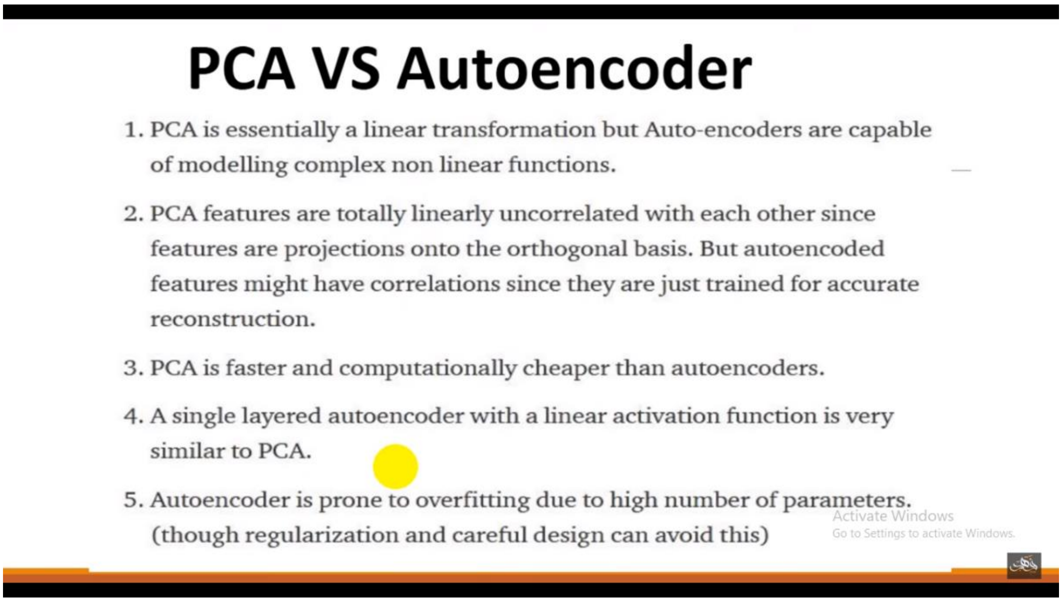
Drawbacks:

- When training the model, there is a need to calculate the relationship of each parameter in the network with respect to the final output loss using a technique known as backpropagation. Hence, the sampling process requires some extra attention.



Applications

1. Dimensionality Reduction
2. Image Compression
3. Image Denoising
4. Feature Extraction
5. Image Generation
6. Sequence to Sequence Prediction
7. Recommendation System



PCA VS Autoencoder

1. PCA is essentially a linear transformation but Auto-encoders are capable of modelling complex non linear functions.
2. PCA features are totally linearly uncorrelated with each other since features are projections onto the orthogonal basis. But autoencoded features might have correlations since they are just trained for accurate reconstruction.
3. PCA is faster and computationally cheaper than autoencoders.
4. A single layered autoencoder with a linear activation function is very similar to PCA.
5. Autoencoder is prone to overfitting due to high number of parameters. (though regularization and careful design can avoid this)

Summary

- Autoencoders work by compressing the input into a latent space representation and then reconstructing the output from this representation. This kind of network is composed of two parts:
- Encoder: This is the part of the network that compresses the input into a latent-space representation. It can be represented by an encoding function h=f(x).
- Decoder: This part aims to reconstruct the input from the latent space representation. It can be represented by a decoding function r=g(h).
- If the only purpose of autoencoders was to copy the input to the output, they would be useless. We hope that by training the autoencoder to copy the input to the output, the latent representation will take on useful properties. This can be achieved by creating constraints on the copying task.
- If the autoencoder is given too much capacity, it can learn to perform the copying task without extracting any useful information about the distribution of the data. This can

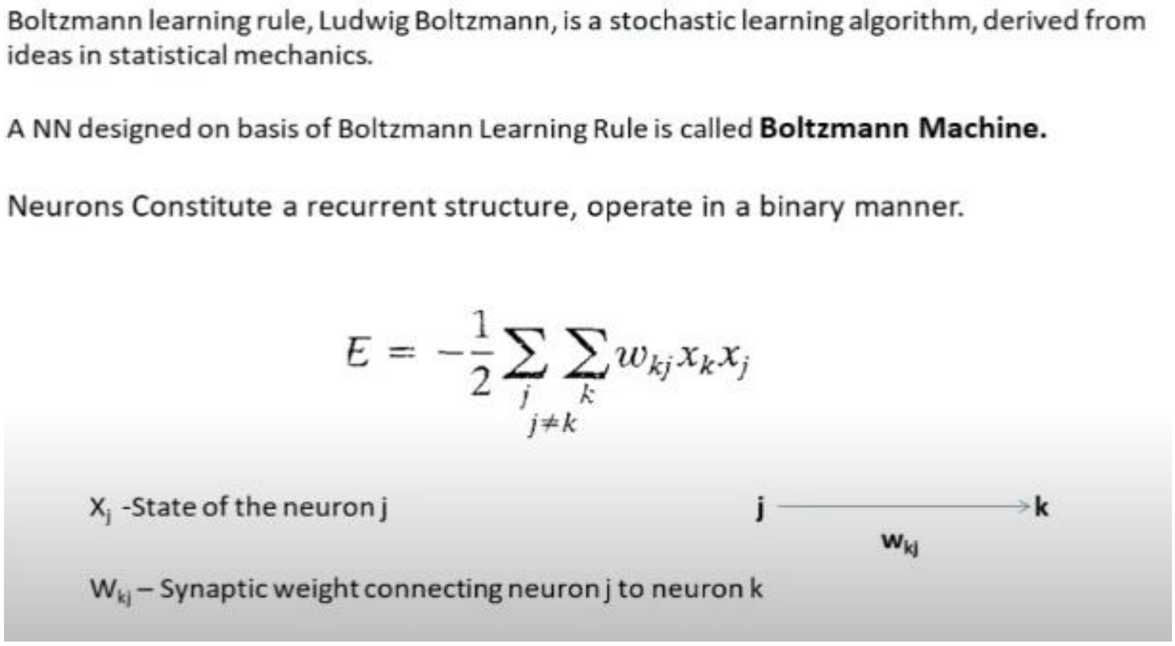
also occur if the dimension of the latent representation is the same as the input, and in the overcomplete case,

- where the dimension of the latent representation is greater than the input. In these cases, even a linear encoder and linear decoder can learn to copy the input to the output without learning anything useful about the data distribution.
- Ideally, one could train any architecture of autoencoder successfully, choosing the code dimension and the capacity of the encoder and decoder based on the complexity of distribution to be modeled.
- Autoencoders are learned automatically from data examples. It means that it is easy to train specialized instances of the algorithm that will perform well on a specific type of input and that it does not require any new engineering, only the appropriate training data.
- However, autoencoders will do a poor job for image compression. As the autoencoder is trained on a given set of data, it will achieve reasonable compression results on data similar to the training set used but will be poor general-purpose image compressors.
- Autoencoders are trained to preserve as much information as possible when an input is run through the encoder and then the decoder, but are also trained to make the new representation have various nice properties.

Deep Generative Models

- Boltzmann Machine
- Restricted Boltzmann Machine
- Deep Belief Networks
- Deep Boltzmann Machines

Boltzmann Learning



Boltzmann learning rule, Ludwig Boltzmann, is a stochastic learning algorithm, derived from ideas in statistical mechanics.

A NN designed on basis of Boltzmann Learning Rule is called **Boltzmann Machine.**

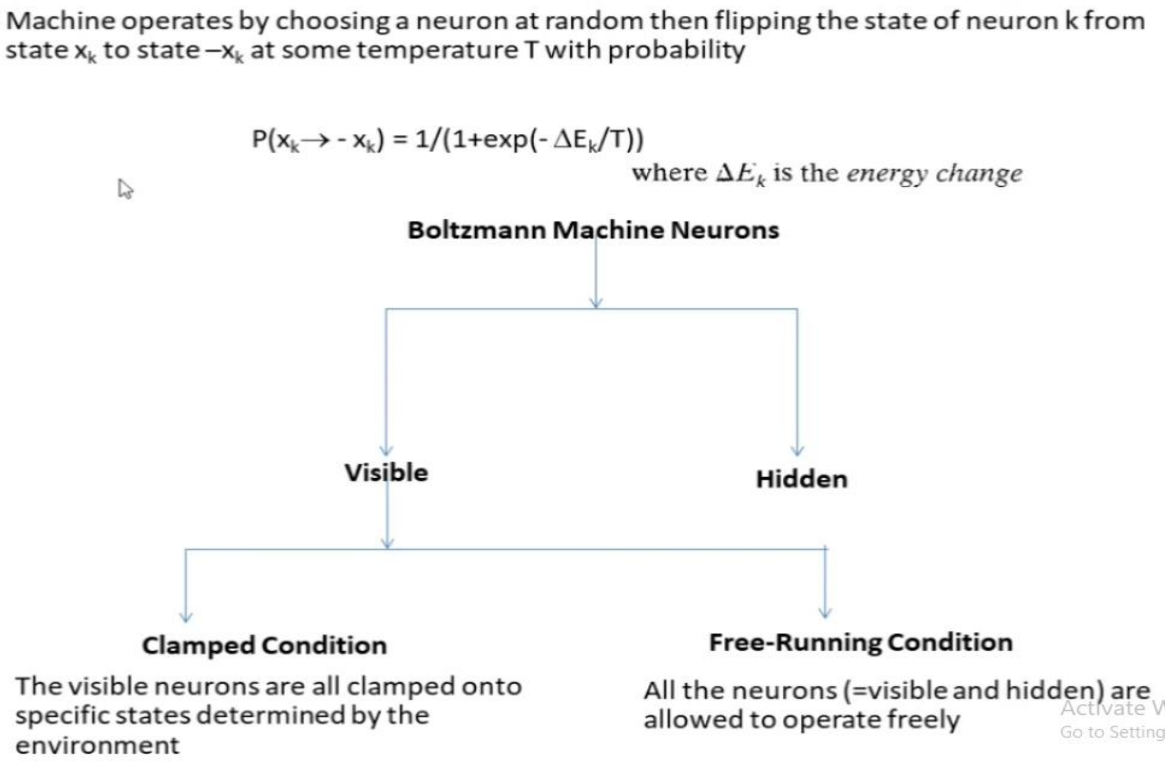
Neurons Constitute a recurrent structure, operate in a binary manner.

$$E = -\frac{1}{2}\sum_{\substack{j \\ j\neq k}} \sum_{k} w_{kj} x_k x_j$$

X_j -State of the neuron j

$$\mathbf{j} \xrightarrow[\mathbf{w_{kj}}]{} \mathbf{k}$$

W_{kj} – Synaptic weight connecting neuron j to neuron k



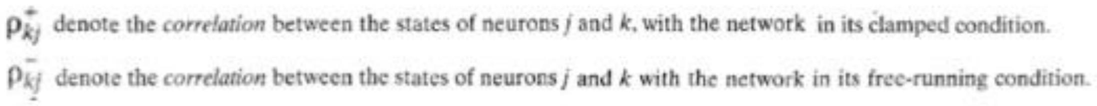
Machine operates by choosing a neuron at random then flipping the state of neuron k from state x_k to state $-x_k$ at some temperature T with probability

$$P(x_k \rightarrow - x_k) = 1/(1+\exp(- \Delta E_k/T))$$

where ΔE_k is the *energy change*

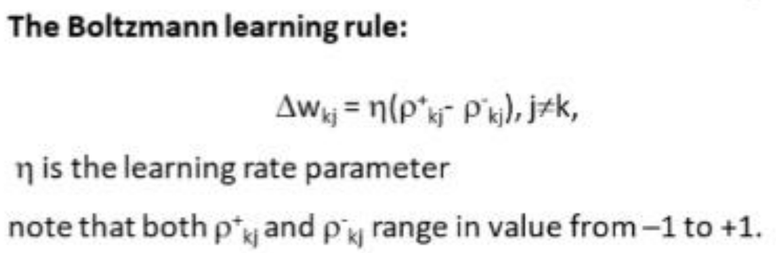
The visible neurons are all clamped onto specific states determined by the environment

All the neurons (=visible and hidden) are allowed to operate freely



ρ^{+}_{kj} denote the *correlation* between the states of neurons *j* and *k*, with the network in its clamped condition.

ρ^{-}_{kj} denote the *correlation* between the states of neurons *j* and *k* with the network in its free-running condition.

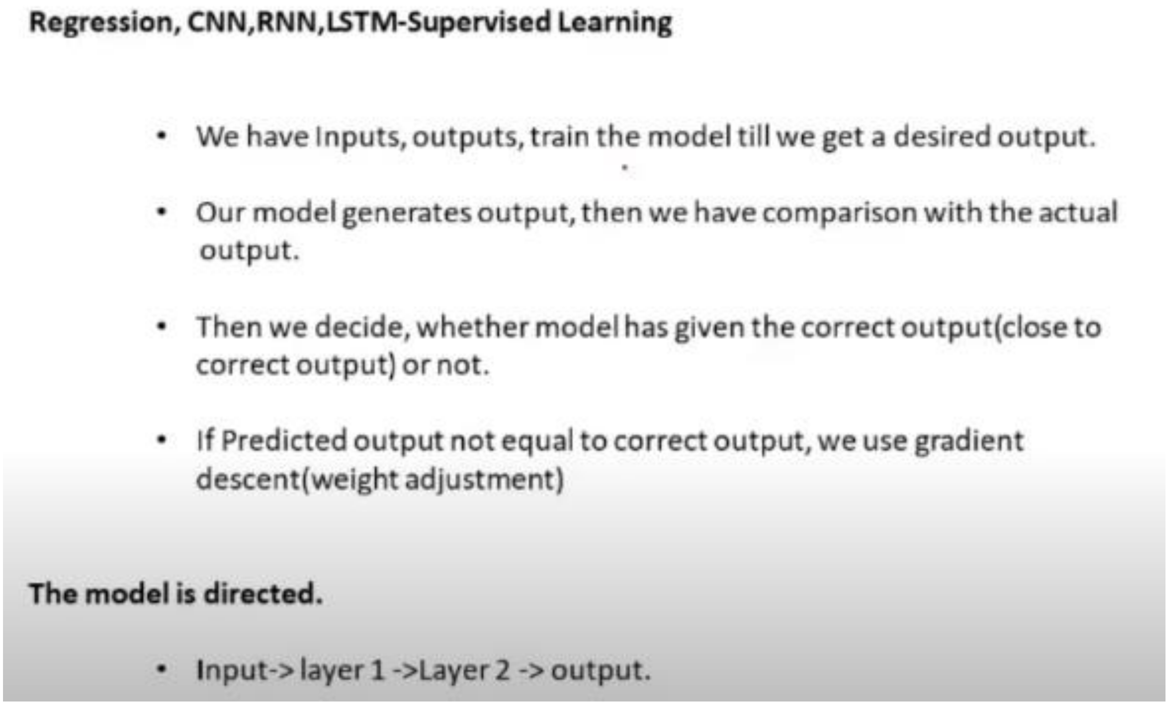


The Boltzmann learning rule:

$$\Delta w_{kj} = \eta(\rho^{+}_{kj} - \rho^{-}_{kj}), j \neq k,$$

η is the learning rate parameter

note that both ρ^{+}_{kj} and ρ^{-}_{kj} range in value from –1 to +1.

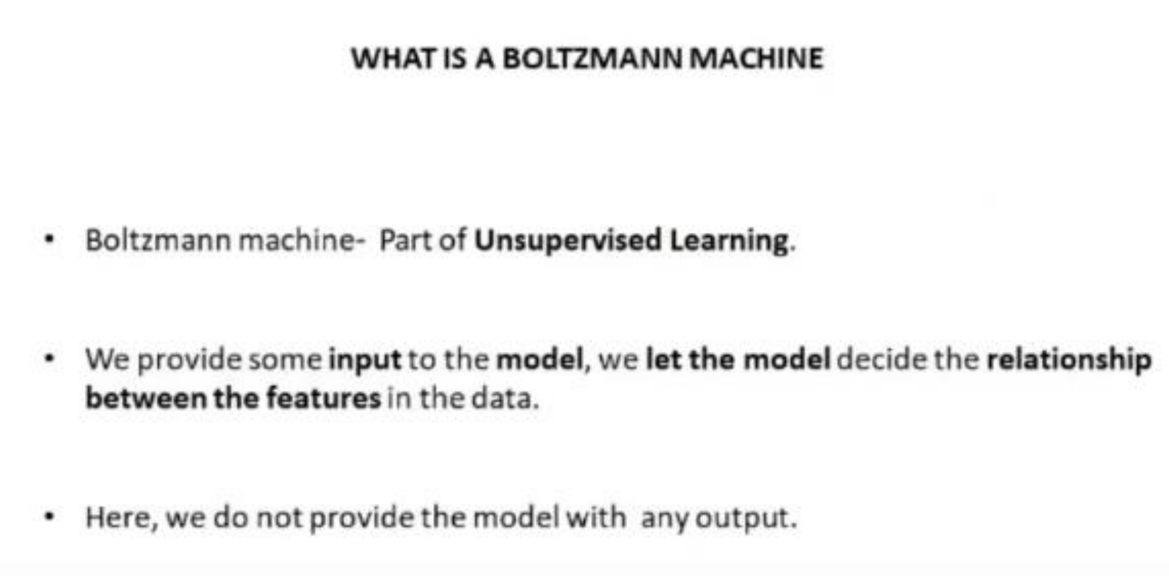


Regression, CNN,RNN,LSTM-Supervised Learning

- We have Inputs, outputs, train the model till we get a desired output.
- Our model generates output, then we have comparison with the actual output.
- Then we decide, whether model has given the correct output(close to correct output) or not.
- If Predicted output not equal to correct output, we use gradient descent(weight adjustment)

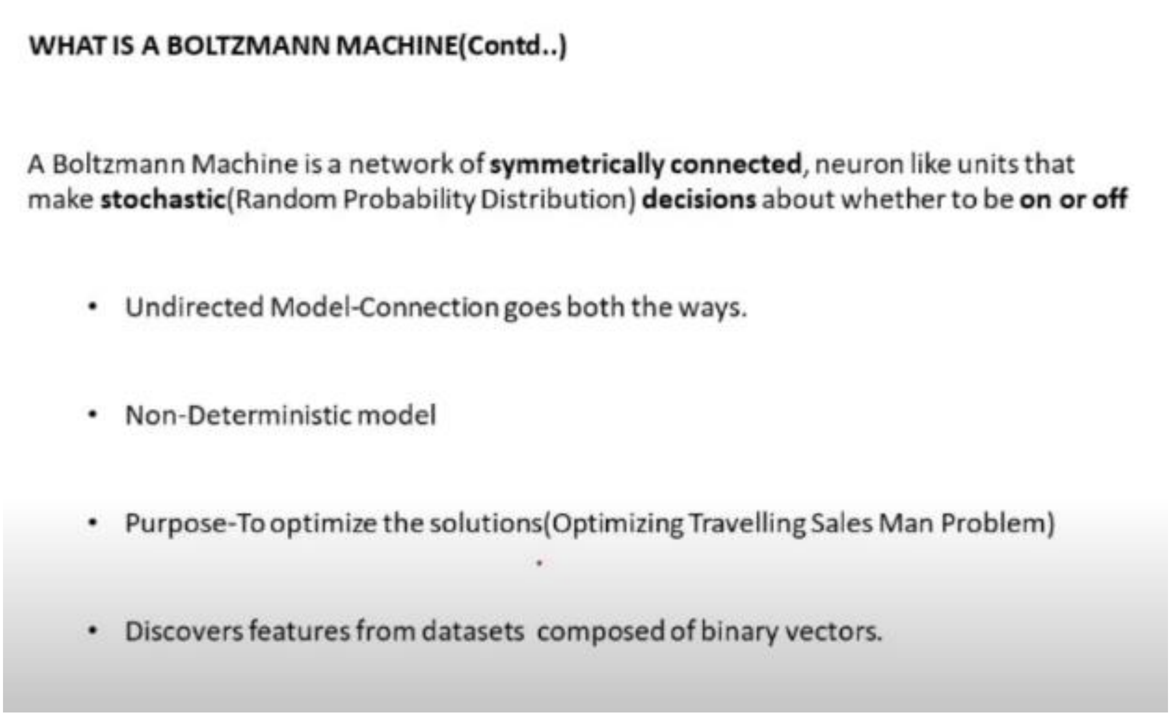
The model is directed.

- Input-> layer 1 ->Layer 2 -> output.



WHAT IS A BOLTZMANN MACHINE

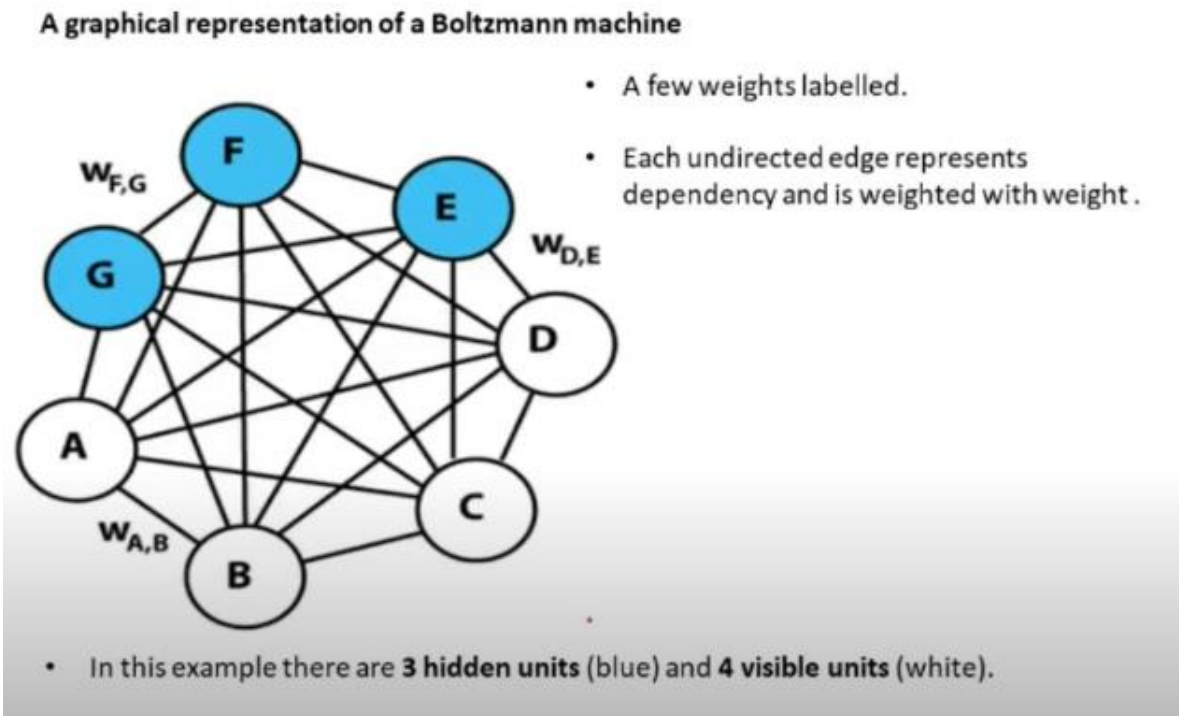
- Boltzmann machine- Part of **Unsupervised Learning**.
- We provide some **input** to the **model**, we **let the model** decide the **relationship between the features** in the data.
- Here, we do not provide the model with any output.



WHAT IS A BOLTZMANN MACHINE(Contd..)

A Boltzmann Machine is a network of **symmetrically connected**, neuron like units that make **stochastic**(Random Probability Distribution) **decisions** about whether to be **on or off**

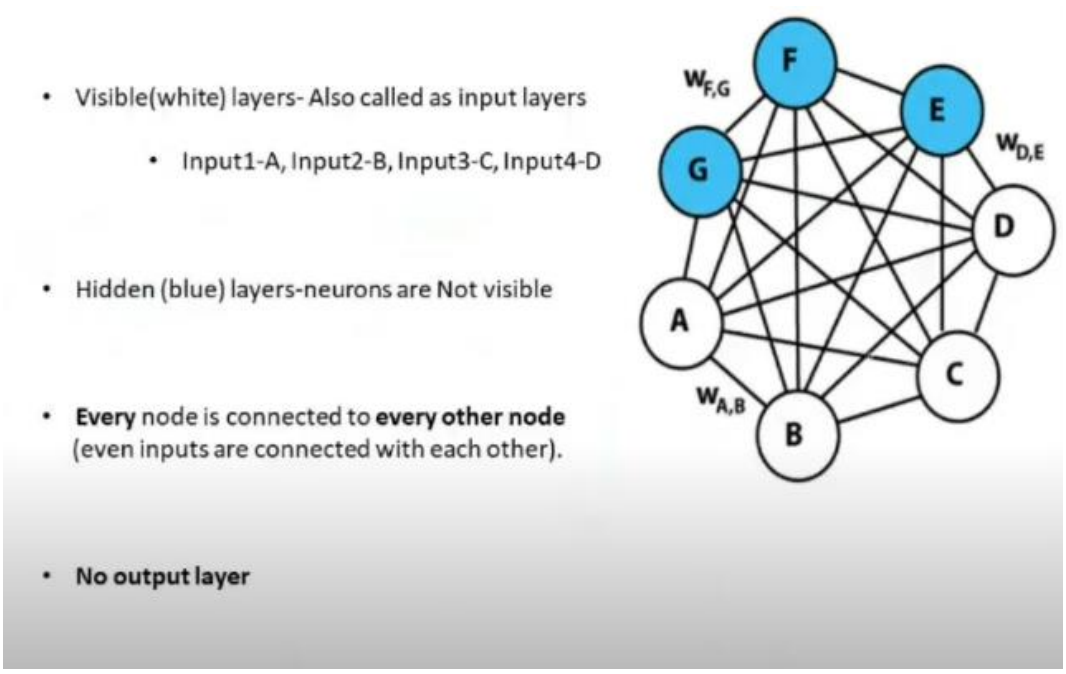
- Undirected Model-Connection goes both the ways.
- Non-Deterministic model
- Purpose-To optimize the solutions(Optimizing Travelling Sales Man Problem)
- Discovers features from datasets composed of binary vectors.



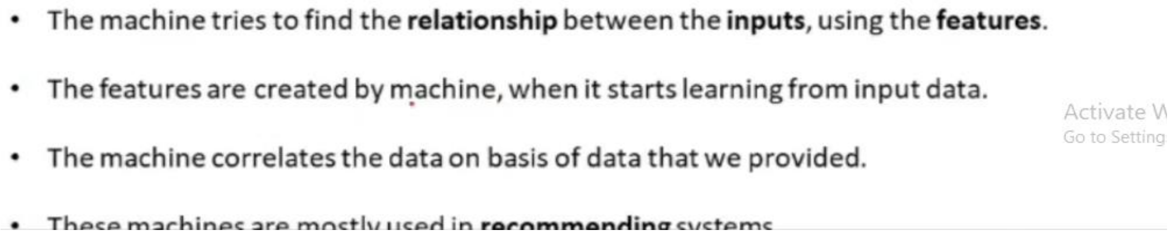
A graphical representation of a Boltzmann machine

- A few weights labelled.
- Each undirected edge represents dependency and is weighted with weight .

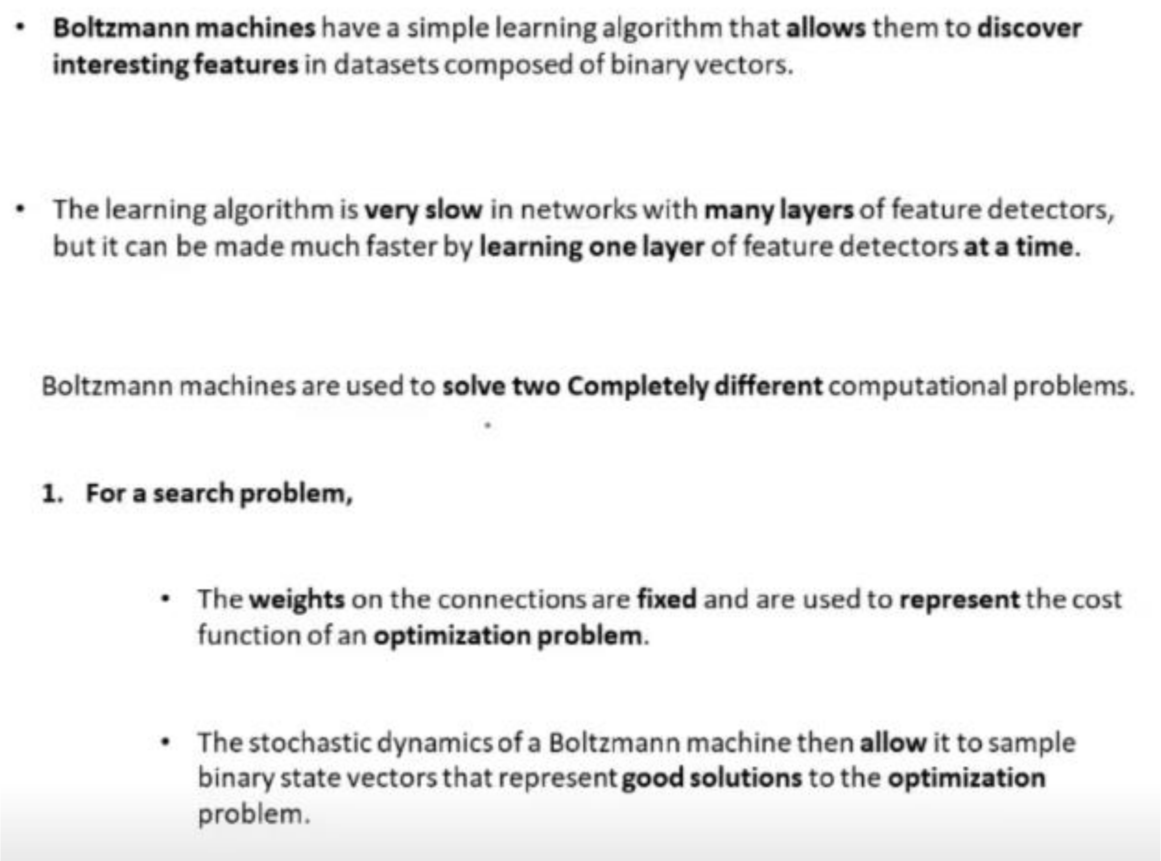
- In this example there are **3 hidden units** (blue) and **4 visible units** (white).



- Visible(white) layers- Also called as input layers
  - Input1-A, Input2-B, Input3-C, Input4-D
- Hidden (blue) layers-neurons are Not visible
- **Every** node is connected to **every other node** (even inputs are connected with each other).
- **No output layer**



- The machine tries to find the **relationship** between the **inputs**, using the **features**.
- The features are created by machine, when it starts learning from input data.
- The machine correlates the data on basis of data that we provided.
- These machines are mostly used in **recommending** systems



- **Boltzmann machines** have a simple learning algorithm that **allows** them to **discover interesting features** in datasets composed of binary vectors.

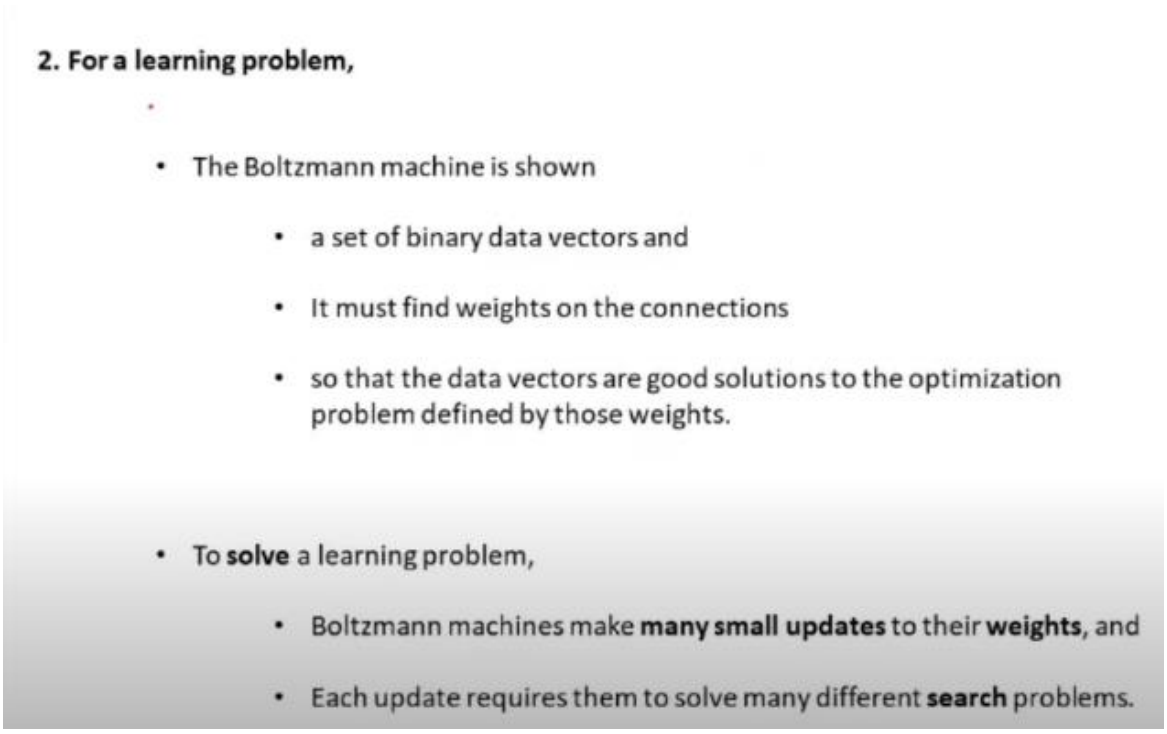
- The learning algorithm is **very slow** in networks with **many layers** of feature detectors, but it can be made much faster by **learning one layer** of feature detectors **at a time.**

Boltzmann machines are used to **solve two Completely different** computational problems.

1. **For a search problem,**

   - The **weights** on the connections are **fixed** and are used to **represent** the cost function of an **optimization problem**.

   - The stochastic dynamics of a Boltzmann machine then **allow** it to sample binary state vectors that represent **good solutions** to the **optimization** problem.



2. For a learning problem,

- The Boltzmann machine is shown

  - a set of binary data vectors and

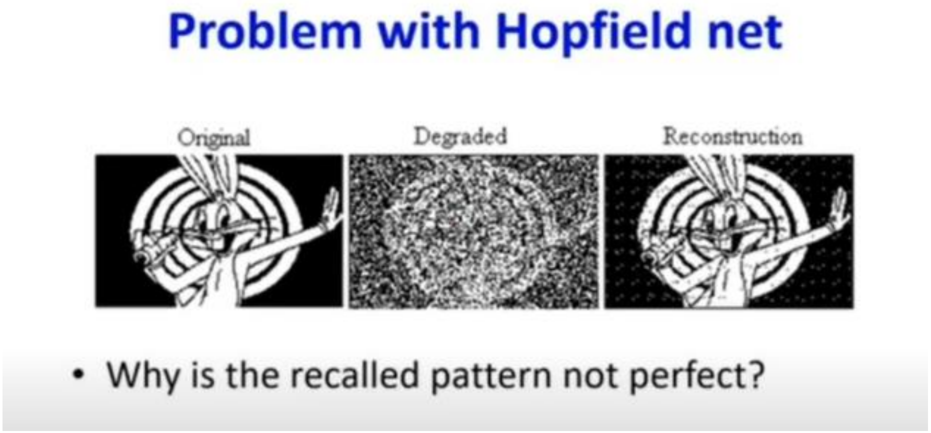
  - It must find weights on the connections

  - so that the data vectors are good solutions to the optimization problem defined by those weights.

- To **solve** a learning problem,

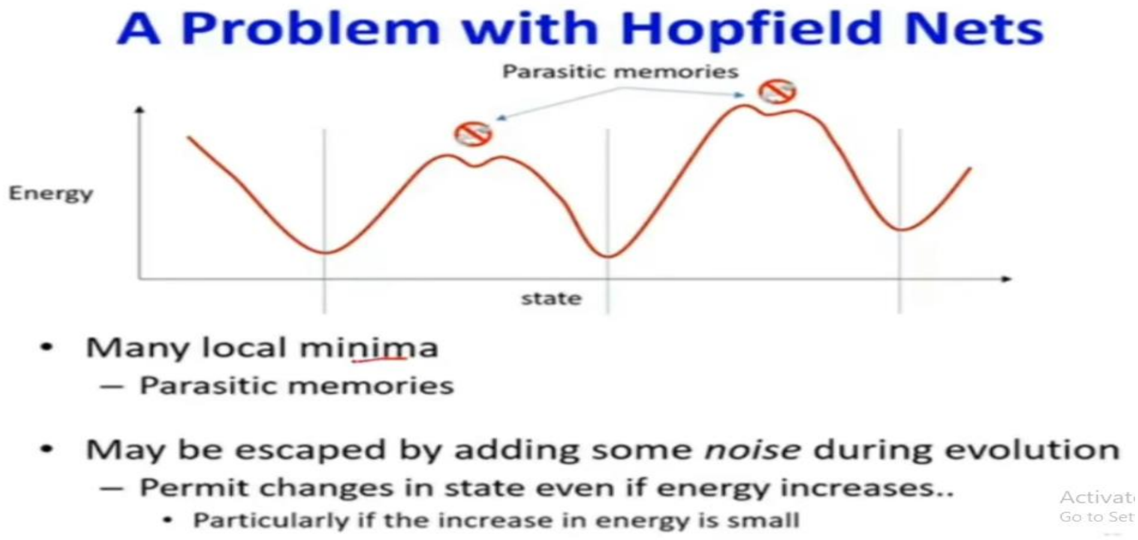
  - Boltzmann machines make **many small updates** to their **weights**, and

  - Each update requires them to solve many different **search** problems.



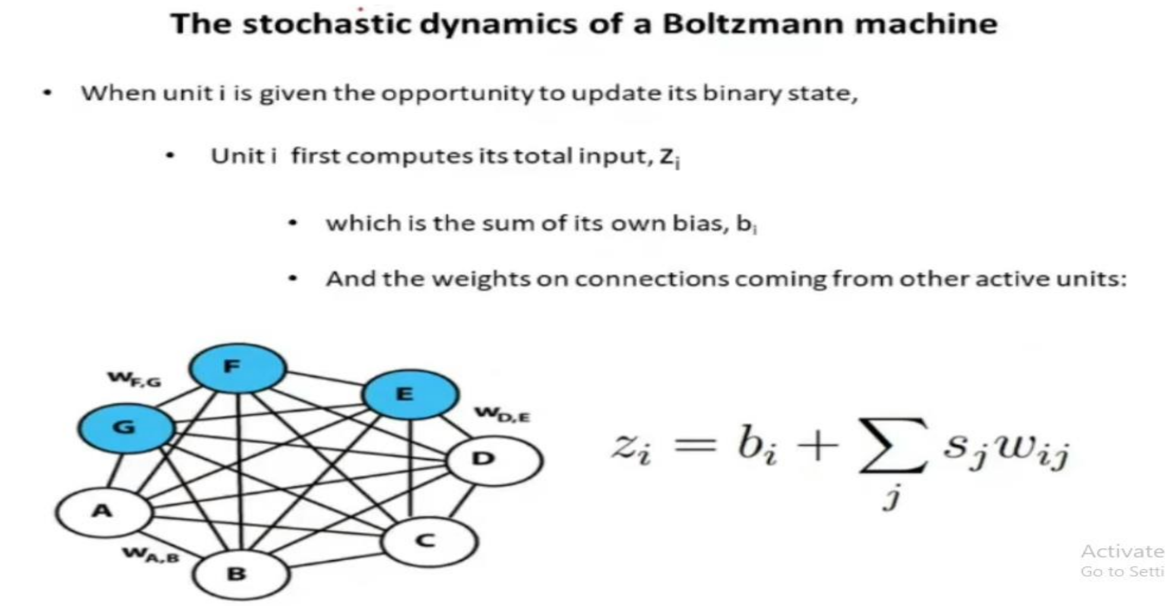
Problem with Hopfield net

- Why is the recalled pattern not perfect?



A Problem with Hopfield Nets

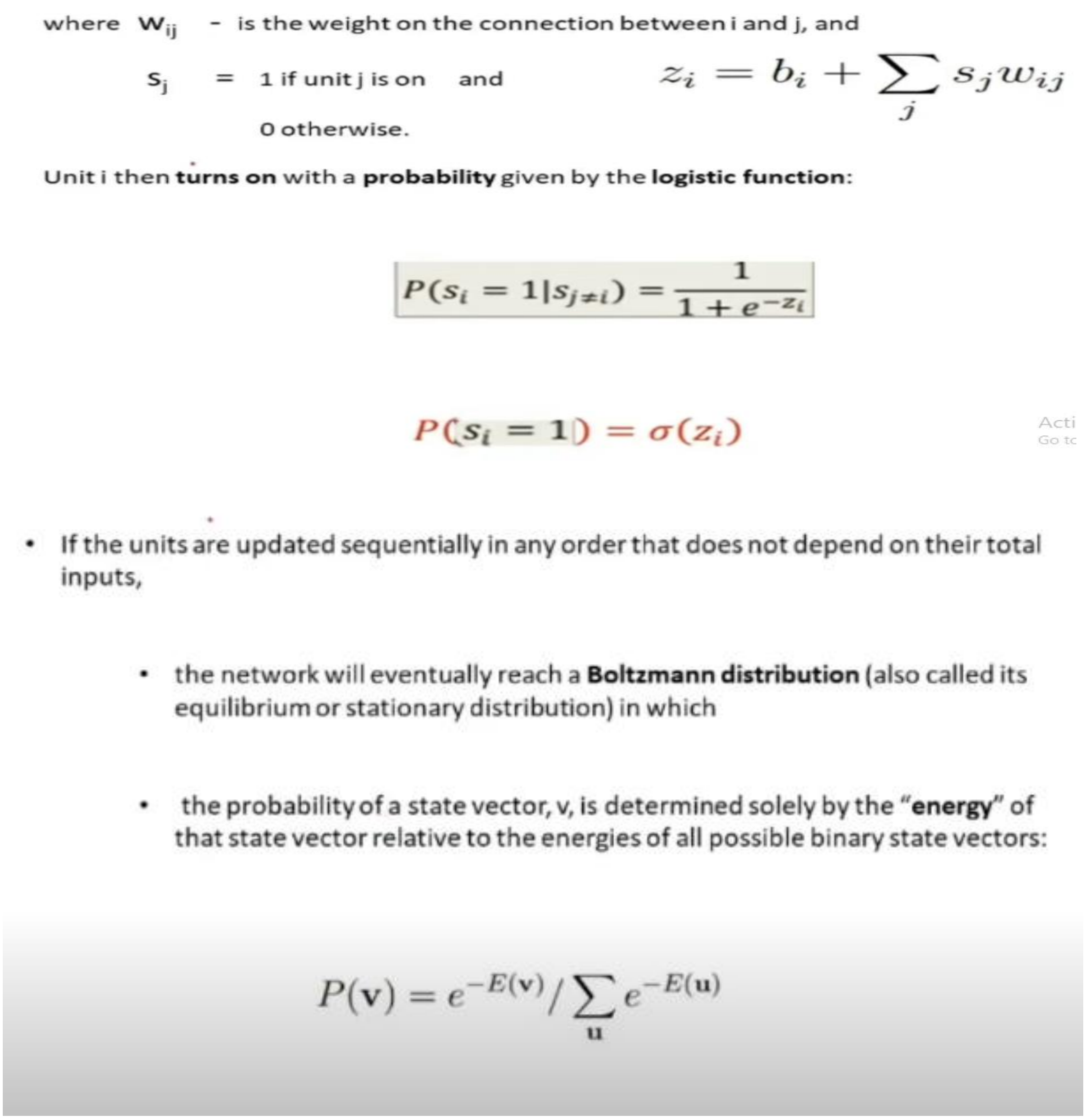
- Many local minima
  - Parasitic memories
- May be escaped by adding some *noise* during evolution
  - Permit changes in state even if energy increases..
    - Particularly if the increase in energy is small



The stochastic dynamics of a Boltzmann machine

- When unit i is given the opportunity to update its binary state,
  - Unit i first computes its total input, Z_i
    - which is the sum of its own bias, b_i
    - And the weights on connections coming from other active units:

$$z_i = b_i + \sum_j s_j w_{ij}$$



where w_{ij} - is the weight on the connection between i and j, and

s_j = 1 if unit j is on and

0 otherwise.

$$z_i = b_i + \sum_j s_j w_{ij}$$

Unit i then **turns on** with a **probability** given by the **logistic function**:

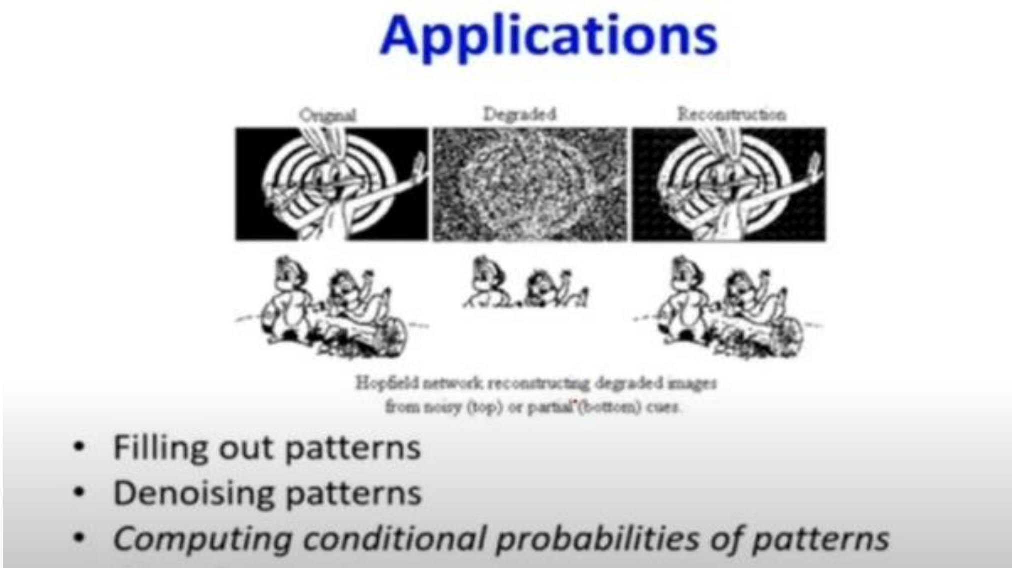
$$P(s_i = 1 | s_{j \neq i}) = \frac{1}{1 + e^{-z_i}}$$

$$P(s_i = 1) = \sigma(z_i)$$

- If the units are updated sequentially in any order that does not depend on their total inputs,
    - the network will eventually reach a **Boltzmann distribution** (also called its equilibrium or stationary distribution) in which
    - the probability of a state vector, v, is determined solely by the "**energy**" of that state vector relative to the energies of all possible binary state vectors:

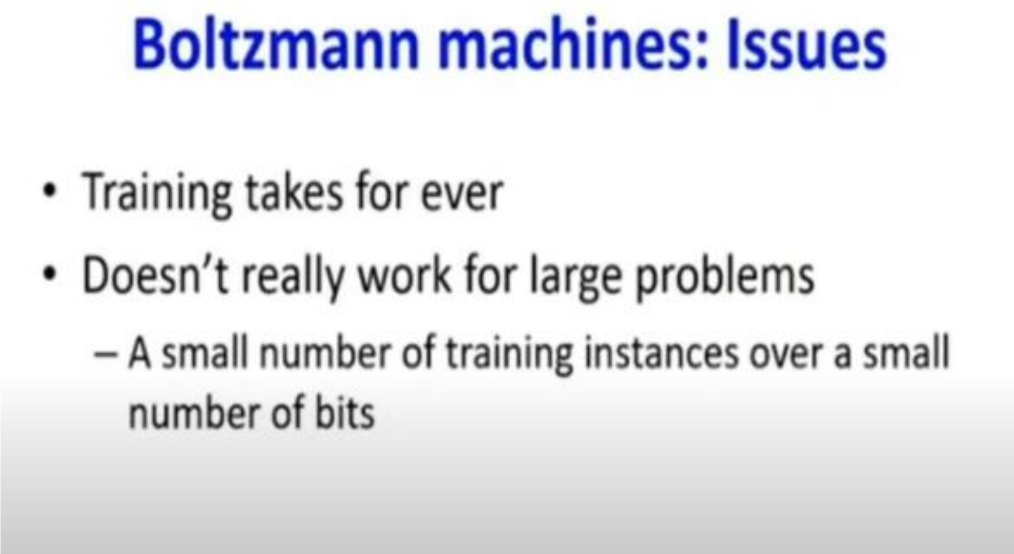
$$P(\mathbf{v}) = e^{-E(\mathbf{v})} / \sum_{\mathbf{u}} e^{-E(\mathbf{u})}$$

Applications



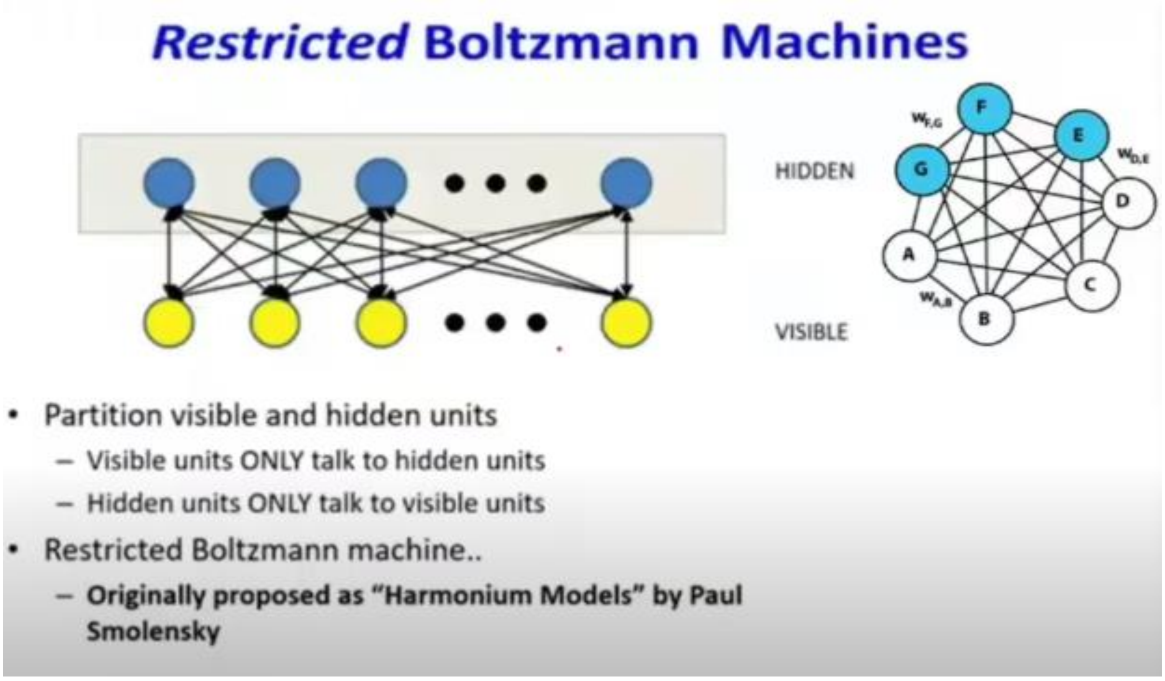
Hopfield network reconstructing degraded images from noisy (top) or partial (bottom) cues.

- Filling out patterns
- Denoising patterns
- *Computing conditional probabilities of patterns*



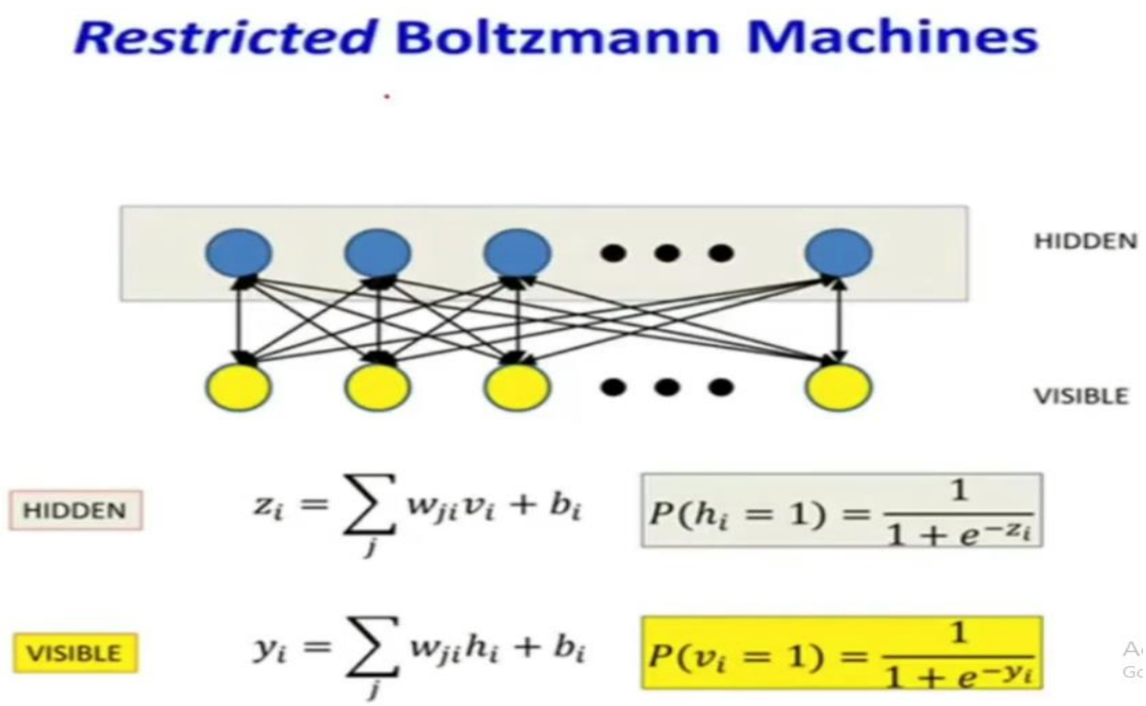
Boltzmann machines: Issues

- Training takes for ever
- Doesn’t really work for large problems
  - A small number of training instances over a small number of bits



Restricted Boltzmann Machines

- Partition visible and hidden units
  - Visible units ONLY talk to hidden units
  - Hidden units ONLY talk to visible units
- Restricted Boltzmann machine..
  - **Originally proposed as “Harmonium Models” by Paul Smolensky**



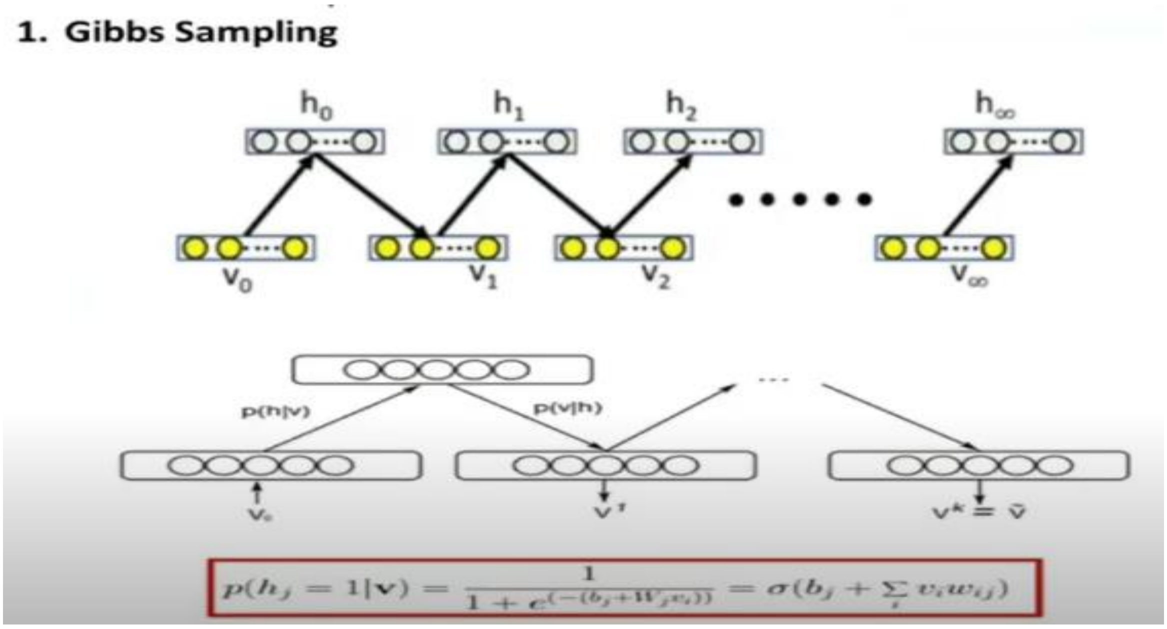
Restricted Boltzmann Machines

HIDDEN $$z_i = \sum_j w_{ji} v_i + b_i \qquad P(h_i = 1) = \frac{1}{1 + e^{-z_i}}$$

VISIBLE $$y_i = \sum_j w_{ji} h_i + b_i \qquad P(v_i = 1) = \frac{1}{1 + e^{-y_i}}$$

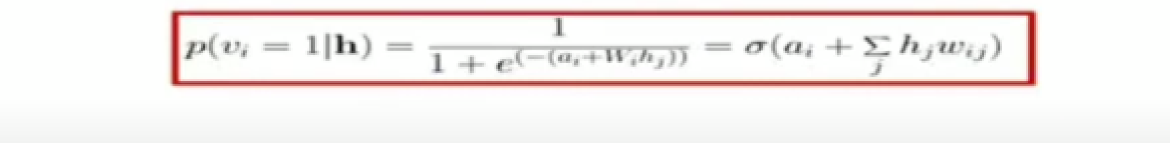
RBM Training

1. Gibbs Sampling
2. Contrastive divergence

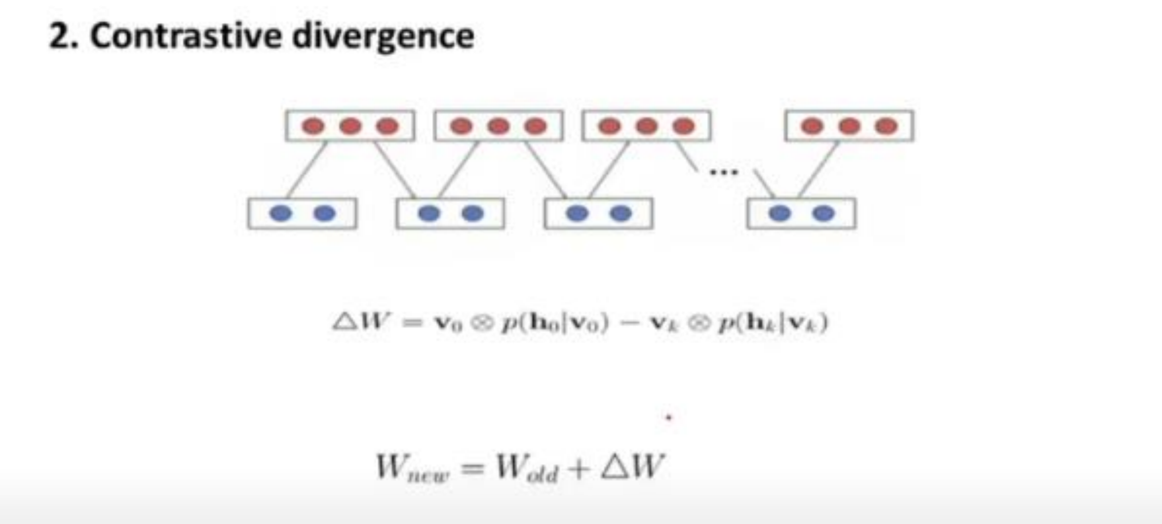


1. Gibbs Sampling

$$p(h_j = 1|\mathbf{v}) = \frac{1}{1+e^{(-(b_j+W_j v_i))}} = \sigma(b_j + \sum_i v_i w_{ij})$$



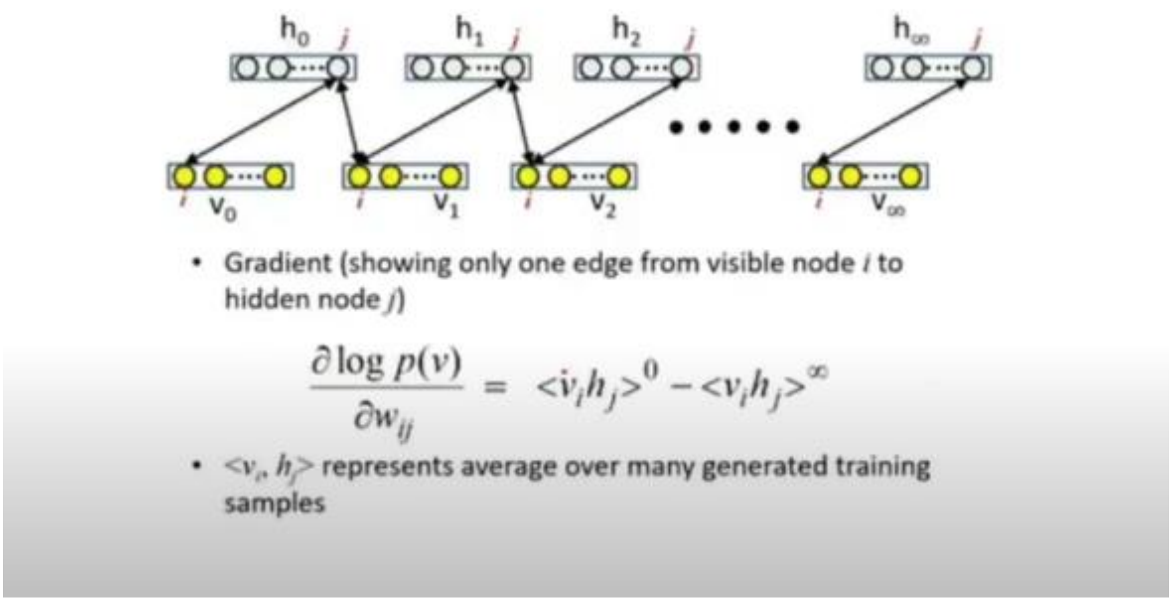
$$p(v_i = 1|\mathbf{h}) = \frac{1}{1+e^{(-(a_i+W_i h_j))}} = \sigma(a_i + \sum_j h_j w_{ij})$$



2. Contrastive divergence

$$\triangle W = \mathbf{v}_0 \otimes p(\mathbf{h}_0|\mathbf{v}_0) - \mathbf{v}_k \otimes p(\mathbf{h}_k|\mathbf{v}_k)$$

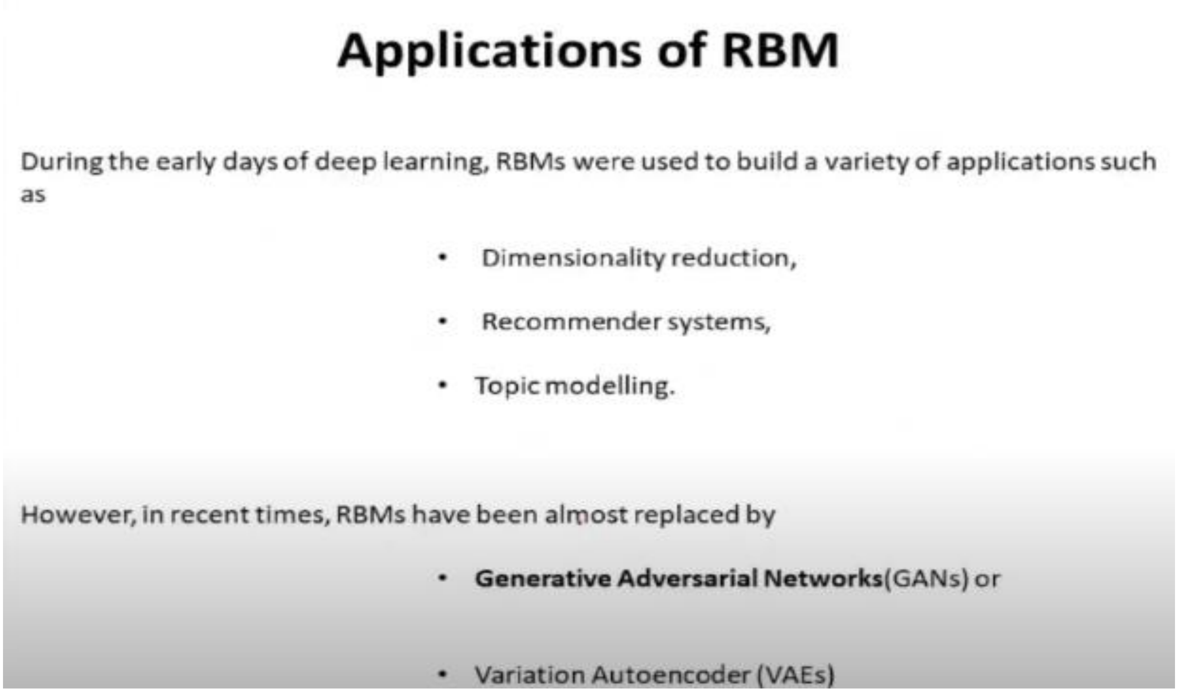
$$W_{new} = W_{old} + \triangle W$$



- Gradient (showing only one edge from visible node *i* to hidden node *j*)

$$\frac{\partial \log p(v)}{\partial w_{ij}} = <v_i h_j>^0 - <v_i h_j>^\infty$$

- $<v_i, h_j>$ represents average over many generated training samples



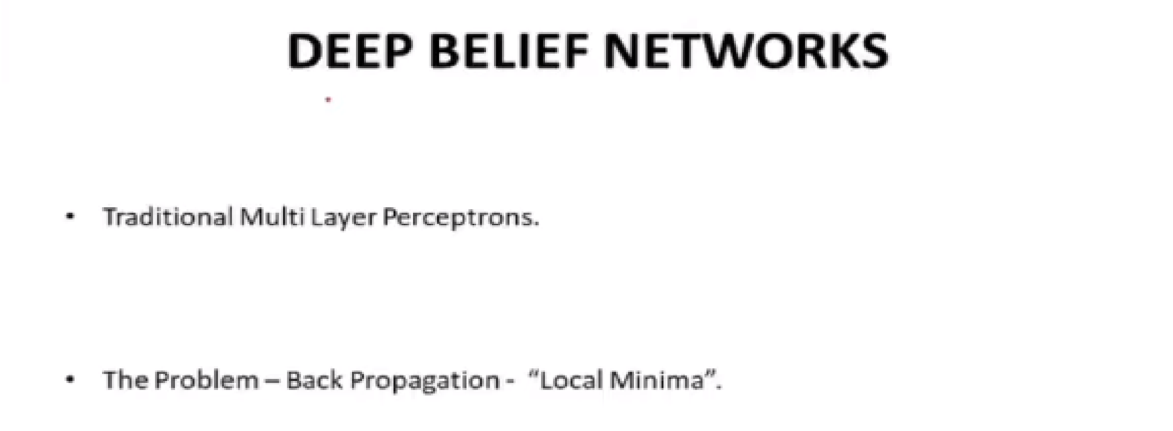
Applications of RBM

During the early days of deep learning, RBMs were used to build a variety of applications such as

- Dimensionality reduction,
- Recommender systems,
- Topic modelling.

However, in recent times, RBMs have been almost replaced by

- **Generative Adversarial Networks**(GANs) or
- Variation Autoencoder (VAEs)



DEEP BELIEF NETWORKS

- Traditional Multi Layer Perceptrons.
- The Problem – Back Propagation - "Local Minima".

Solution: Deep Belief Networks-DBNs

PRETRAINING

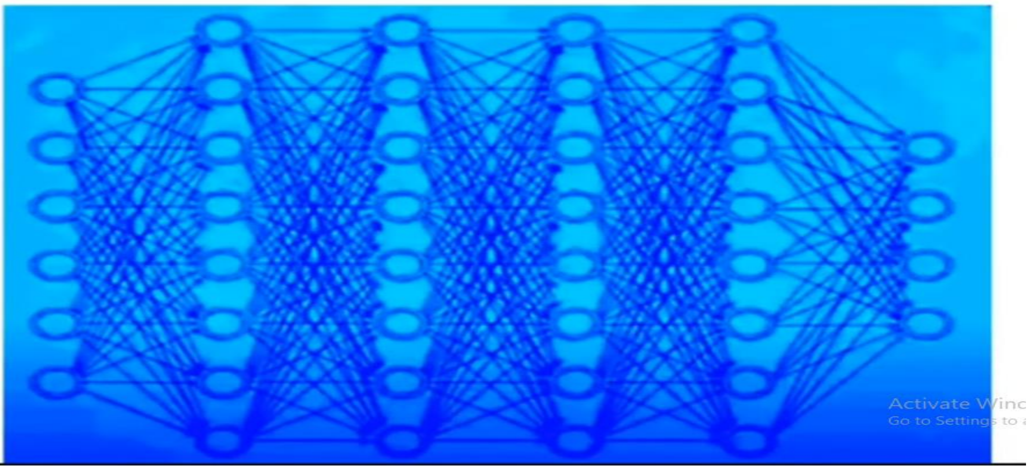
PRETRAINING →BACK PROPAGATION : Leads to reduced **ERROR RATE**

ARCHITECTURE OF DEEP BELIEF NETWORKS

- Network Structure – DBN identical to MLP.

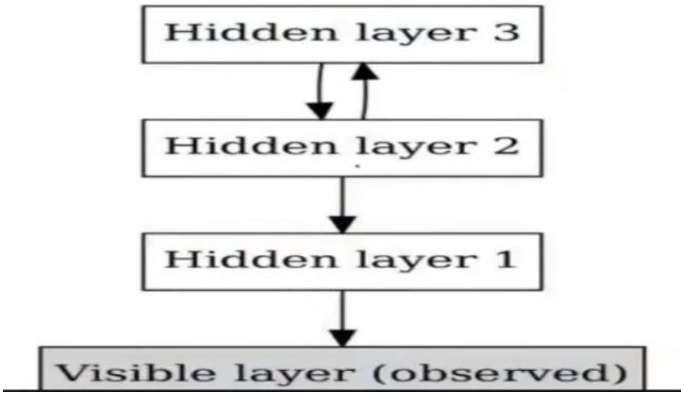
- Training – DBN Differs with MLP.

DBN - Śtack of RBMs



- The connections in the top layers are undirected and associative memory is formed from the connections between them.

- The connections in the lower levels are directed.



The nodes in the hidden layer fulfill two roles—

They act

- as a hidden layer to nodes that precede it and
- as visible layers to nodes that succeed it.

- These nodes identify the correlations in the data.

DBN WORKING

- Greedy learning algorithms are used to pre-train deep belief networks.

- This is a problem-solving approach that involves making the optimal choice at each layer in the sequence, eventually finding a global optimum.

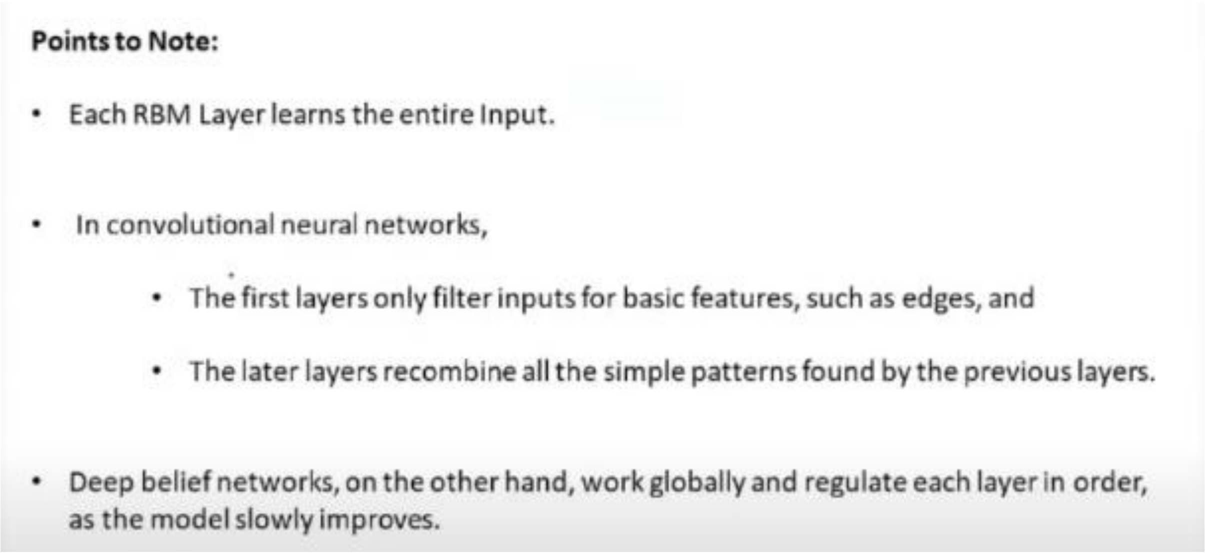
- Greedy learning algorithms start from the bottom layer and move up, fine-tuning the generative weights.

- The learning takes place on a layer-by-layer basis, meaning the layers of the deep belief networks are trained one at a time.

- Therefore, each layer also receives a different version of the data, and each layer uses the output from the previous layer as their input.

Greedy learning algorithms are used to train deep belief networks because they are quick and efficient.

Moreover, they help to optimize the weights at each layer.



Points to Note:

- Each RBM Layer learns the entire Input.
- In convolutional neural networks,
  - The first layers only filter inputs for basic features, such as edges, and
  - The later layers recombine all the simple patterns found by the previous layers.
- Deep belief networks, on the other hand, work globally and regulate each layer in order, as the model slowly improves.

Applications of Deep Belief Networks

- Image Recognition
- Video Recognition
- Motion-Capture Data

Deep Boltzmann Machines

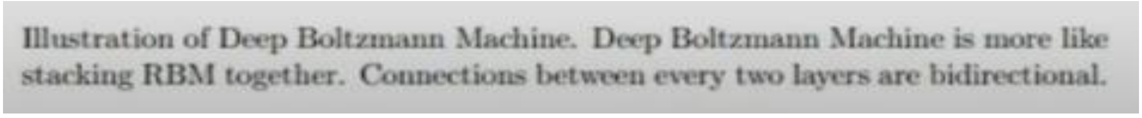
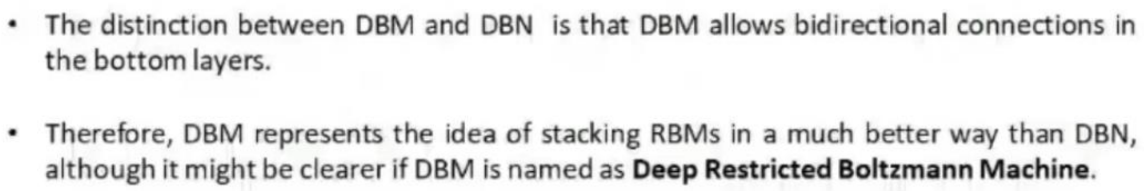
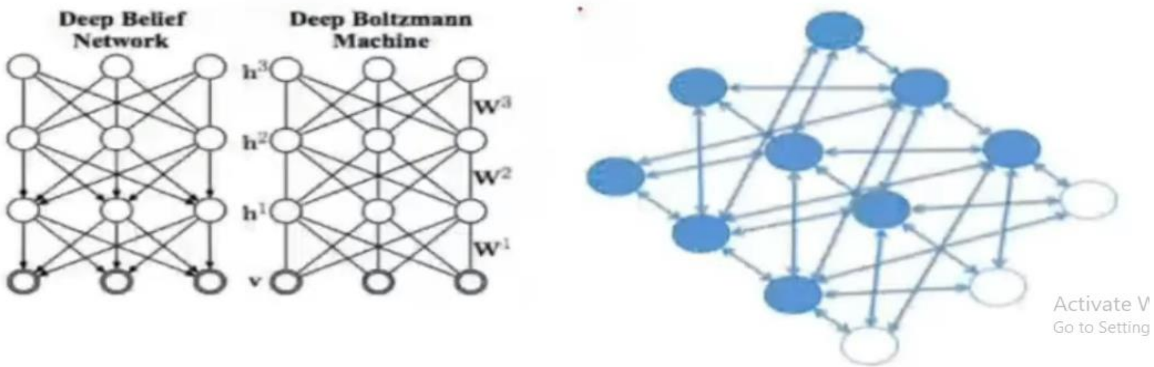


Illustration of Deep Boltzmann Machine. Deep Boltzmann Machine is more like stacking RBM together. Connections between every two layers are bidirectional.



- The distinction between DBM and DBN is that DBM allows bidirectional connections in the bottom layers.
- Therefore, DBM represents the idea of stacking RBMs in a much better way than DBN, although it might be clearer if DBM is named as **Deep Restricted Boltzmann Machine**.



- Due to the nature of DBM, its energy function is defined as an extension of the energy function of an RBM

$$E(v,h) = -\sum_{i}^{i} v_i b_i - \sum_{n=1}^{N}\sum_{k} h_{n,k} b_{n,k} - \sum_{i,k} v_i w_{ik} h_k - \sum_{n=1}^{N-1}\sum_{k,l} h_{n,k} w_{n,k,l} h_{n+1,l}$$

for a DBM with N hidden layers.

Deep Boltzmann Machine (DBM) v.s. Deep Belief Networks (DBN)

Similarities:

- Deep Boltzmann Machine and Deep Belief Networks have many similarities, especially from the first glance.

- Both of them are **deep neural networks** originates from the idea of **Restricted Boltzmann Machine.**

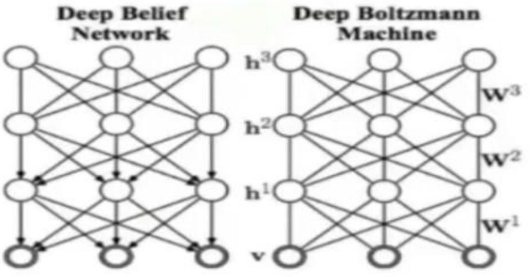
- Both of them also rely on layer wise pre-training for a success of **parameter learning**.

Differences:

- The fundamental differences between these two models is : by how the connections are made between bottom layers (un-directed/bi-directed v.s. directed).

- The bidirectional structure of DBM grants the possibility of DBM to learn a more **complex pattern** of data.

- It also grants the possibility for the approximate inference procedure to incorporate top-down feedback in addition to an initial bottom-up pass, allowing Deep Boltzmann Machines to better propagate uncertainty about ambiguous inputs.


Deep Belief Network
Deep Boltzmann Machine
h³
h²
h¹
v
W³
W²
W¹