Unit – 2

Part – I – Building Blocks of Deep Learning

We introduce networks that are considered building blocks of larger deep networks:

• RBMs

• Autoencoders

Both RBMs and autoencoders are characterized by an extra layer-wise step for train- ing. They are often used for the pretraining phase in other larger deep networks.

RBMs model probability and are great at feature extraction. They are feed-forward 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.

Autoencoders are a variant of feed-forward neural networks that have an extra bias for calculating the error of reconstructing the original input. After training, autoencoders are then used as a normal feed-forward neural network for activations.

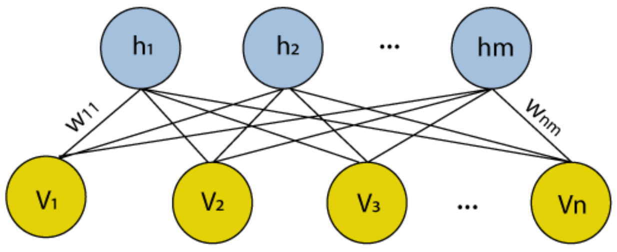
Deep networks can use either RBMs or autoencoders as building blocks for larger networks.

RBMs

Restricted Boltzmann Machine is an undirected graphical model that plays a major role in Deep Learning Framework in recent times.

It is an algorithm which is useful for dimensionality reduction, classification, regression, collaborative filtering, feature learning, and topic modeling.

The "restricted" part of the name "Restricted Boltzmann Machines" means that con- nections between nodes of the same layer are prohibited (e.g., there are no visible-visible or hidden-hidden connections along which signal passes).



Network layout

There are five main parts of a basic RBM:

• Visible units

• Hidden units

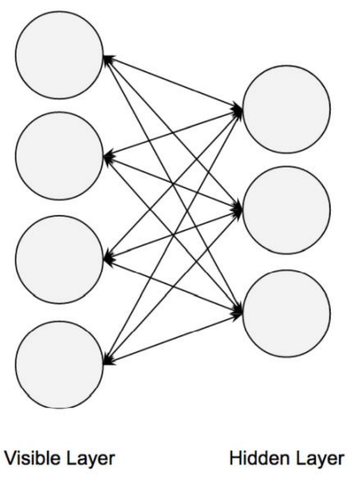
• Weights

• Visible bias units

• Hidden bias units

Layers

Restricted Boltzmann Machines are shallow, two-layer neural nets that constitute the building blocks of ***deep-belief networks***. The first layer of the RBM is called the **visible**, or input layer, and the second is the **hidden** layer. Each circle represents a neuron-like unit called a node. The nodes are connected to each other across layers, but no two nodes of the same layer are linked.



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. Hid- den units are feature detectors, learning features from the input data.

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.

Connections and weights.

All connections are visible-hidden; none are visible-visible or hidden-hidden. The edges represent connections along which signals are passed. Loosely speaking, those circles, or nodes, act like human neurons. 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.

Usually, being "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 net- work thinks that particular input is irrelevant noise.

Biases.

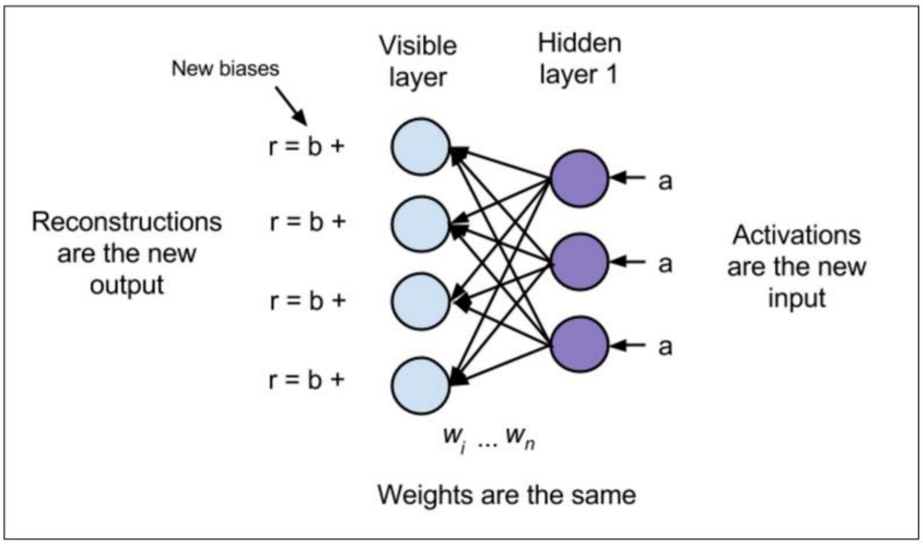
There is a set of bias weights ("parameters") connecting the bias unit for each layer to every unit in the layer. Bias nodes help the network better triage and model cases in which an input node is always on or always off.

Training.

The technique known as pretraining using RBMs means teaching it to reconstruct the original data from a limited sample of that data. That is, given a chin, a trained net- work could approximate (or "reconstruct") a face. RBMs learn to reconstruct the input dataset.

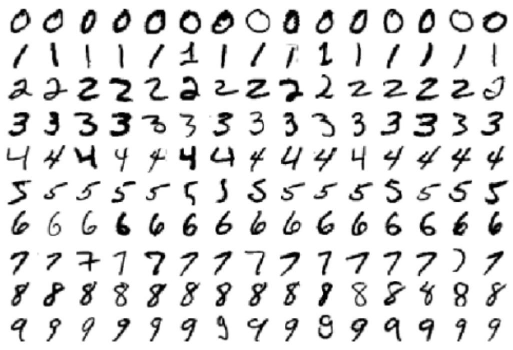
Reconstruction

Deep neural networks with unsupervised pretraining (RBMs, autoencoders) 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.



Example:

We can visually explain reconstruction in RBMs by looking at the MNIST dataset. The MNIST dataset is a collection of images representing the handwritten numerals 0 through 9.



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 to see how well it can reconstruct the digits.

If the training data has a normal distribution, most of them cluster around a central mean, or average, and become scarcer the further you stray from that average. It looks like a bell curve. If we know the mean and the variance, or sigma, of normal data, we can reconstruct that curve. But suppose that we don't know the mean and variance. Those are parameters we then need to guess. Picking them randomly and contrasting the curve they produce with the original data can operate similarly to a loss function. We measure the difference between two probability distributions much like we measure erroneous classifications, adjust our parameters, and try again.

Other uses of RBMs

Here are some other places we see RBMs used: • Dimensionality reduction • Classification • Regression • Collaborative filtering • Topic modelling

Training of Restricted Boltzmann Machine

The training of the Restricted Boltzmann Machine differs from the training of regular **neural networks** via stochastic gradient descent.

The Two main Training steps are:

- **Gibbs Sampling**

The first part of the training is called *Gibbs Sampling*. Given an input vector **v** we use **p(h|v)**for prediction of the hidden values **h.** Knowing the hidden values we use **p(v|h)** :

$$\boxed{p(v_i = 1 \mid \mathrm{h}) = \frac{1}{1+e(-(\mathrm{a}_i + w_i h_j))} = \tilde{\mathrm{O}}\ (\mathrm{a}_i + \sum \mathrm{h}_j \mathrm{w}_{ij})}$$

for prediction of new input values **v**. This process is repeated *k* times. After *k* iterations, we obtain another input vector **v_k** which was recreated from original input values **v_0**.

$$\boxed{p(h_i = 1 \mid \mathrm{v}) = \frac{1}{1+e(-(\mathrm{b}_j + W_j v_i))} = \sigma(\mathrm{b}_j + \sum_i v_i w_{ij})}$$

- **Contrastive Divergence step**

The update of the weight matrix happens during the *Contrastive Divergence* step. Vectors **v_0** and **v_k** are used to calculate the activation probabilities for hidden values **h_0** and **h_k :**

$$\boxed{p(v_i = 1 \mid \mathrm{h}) = \frac{1}{1+e(-(\mathrm{a}_i + w_i h_j))} = \tilde{\mathrm{O}}\ (\mathrm{a}_i + \sum \mathrm{h}_j \mathrm{w}_{ij})}$$

The difference between the outer products of those probabilities with input vectors **v_0** and **v_k** results in the updated matrix :

$$\boxed{\Delta W = \mathrm{v_O}) \otimes P(\mathrm{h_O} \mid \mathrm{v_O}) - \mathrm{v_k} \otimes P(\mathrm{h_k} \mid \mathrm{v_k}) - \mathrm{v_k})}$$

Using the update matrix the new weights can be calculated with gradient **ascent,** given by:

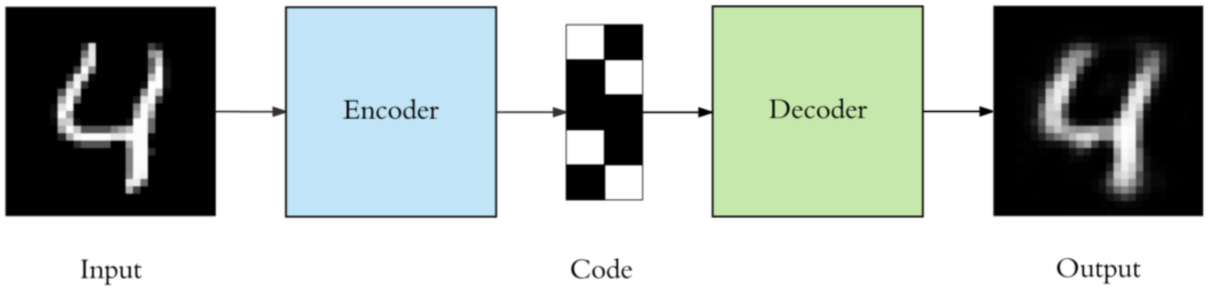
$$\boxed{W_{new} = W_{old} + \triangle W}$$

Autoencoders

An autoencoder neural network is an Unsupervised Machine learning algorithm that applies backpropagation, setting the target values to be equal to the inputs. Autoencoders are used to reduce the size of our inputs into a smaller representation. If anyone needs the original data, they can reconstruct it from the compressed data.

Autoencoders are a specific type of feedforward neural networks trained to copy its input to output. A bottleneck is imposed in the network to represent a compressed knowledge of the original input. The input is compressed into a lower-dimensional code and then the output is reconstructed from this representation. The code is also called as latent-space representation which is a compact "summary" or "compression" of the input.

An autoencoder consists of 3 components: encoder, code and decoder. The encoder compresses the input and produces the code, the decoder then reconstructs the input only using this code.



We have a similar machine learning algorithm i.e. PCA which does the same task.

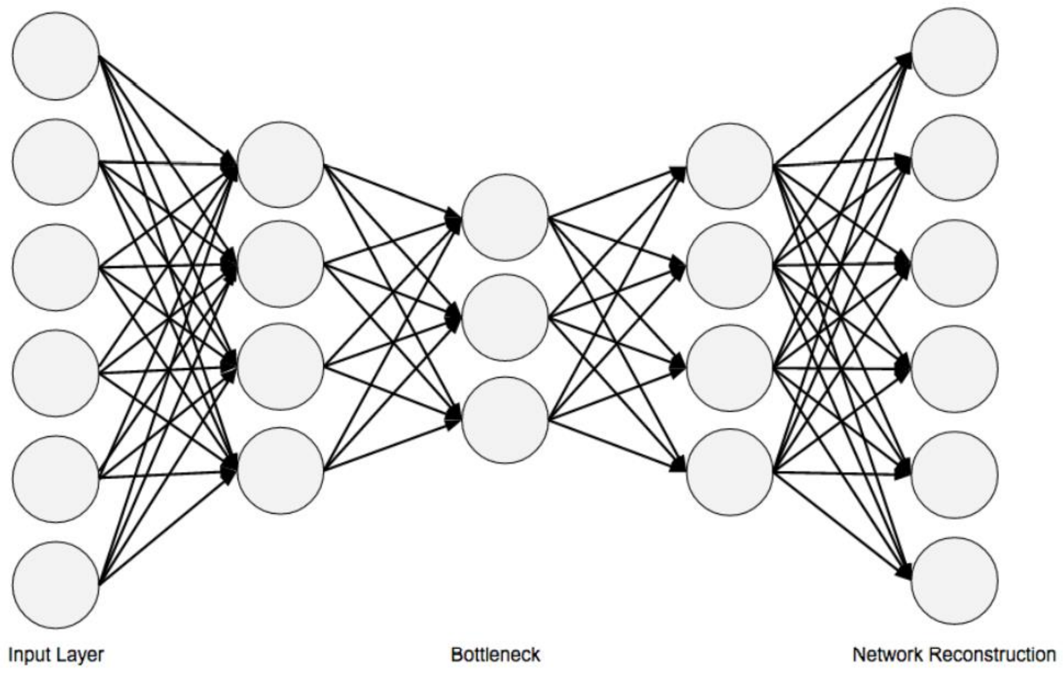
Autoencoders are preferred over PCA because:

- An autoencoder can learn **non-linear transformations** with a **non-linear activation function** and multiple layers.
- It doesn’t have to learn dense layers. It can use **convolutional layers** to learn which is better for video, image and series data.
- It is more efficient to learn several layers with an autoencoder rather than learn one huge transformation with PCA.
- An autoencoder provides a representation of each layer as the output.
- It can make use of **pre-trained layers** from another model to apply transfer learning to enhance the encoder/decoder.

Architecture

- **Encoder:** This part of the network compresses the input into a **latent space representation**. The encoder layer **encodes** the input image as a compressed representation in a reduced dimension. The compressed image is the distorted version of the original image.
- **Code:** This part of the network represents the compressed input which is fed to the decoder.

- **Decoder:** This layer **decodes** the encoded image back to the original dimension. The decoded image is a lossy reconstruction of the original image and it is reconstructed from the latent space representation.



The layer between the encoder and decoder, ie. the code is also known as **Bottleneck**. This is a well-designed approach to decide which aspects of observed data are relevant information and what aspects can be discarded. It does this by balancing two criteria:

- Compactness of representation, measured as the compressibility.
- It retains some behaviourally relevant variables from the input.

<u>Properties of Autoencoders:</u>

- **Data-specific**: Autoencoders are only able to compress data similar to what they have been trained on.
- **Lossy:** The decompressed outputs will be degraded compared to the original inputs.
- **Learned automatically from examples:** It is easy to train specialized instances of the algorithm that will perform well on a specific type of input.

<u>Hyperparameters of Autoencoders:</u>

There are **4** hyperparameters that we need to set before training an autoencoder:

- **Code size**: It represents the number of nodes in the middle layer. Smaller size results in more compression.
- **Number of layers**: The autoencoder can consist of as many layers as we want.

- **Number of nodes per layer**: The number of nodes per layer decreases with each subsequent layer of the encoder, and increases back in the decoder. The decoder is symmetric to the encoder in terms of the layer structure.
- **Loss function:** We either use mean squared error or binary cross-entropy. If the input values are in the range [0, 1] then we typically use cross-entropy, otherwise, we use the mean squared error.

Applications of Autoencoders

Image Coloring

Autoencoders are used for converting any black and white picture into a colored image. Depending on what is in the picture, it is possible to tell what the color should be.

Feature variation

It extracts only the required features of an image and generates the output by removing any noise or unnecessary interruption.

Dimensionality Reduction

The reconstructed image is the same as our input but with reduced dimensions. It helps in providing the similar image with a reduced pixel value.

Denoising Image

The input seen by the autoencoder is not the raw input but a stochastically corrupted version. A denoising autoencoder is thus trained to reconstruct the original input from the noisy version.

Watermark Removal

It is also used for removing watermarks from images or to remove any object while filming a video or a movie.

Common variants of autoencoders

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

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

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.

Comparison with Multilayer perceptron

Autoencoders differ from multilayer perceptron in a couple of ways:

• They use unlabeled data in unsupervised learning.

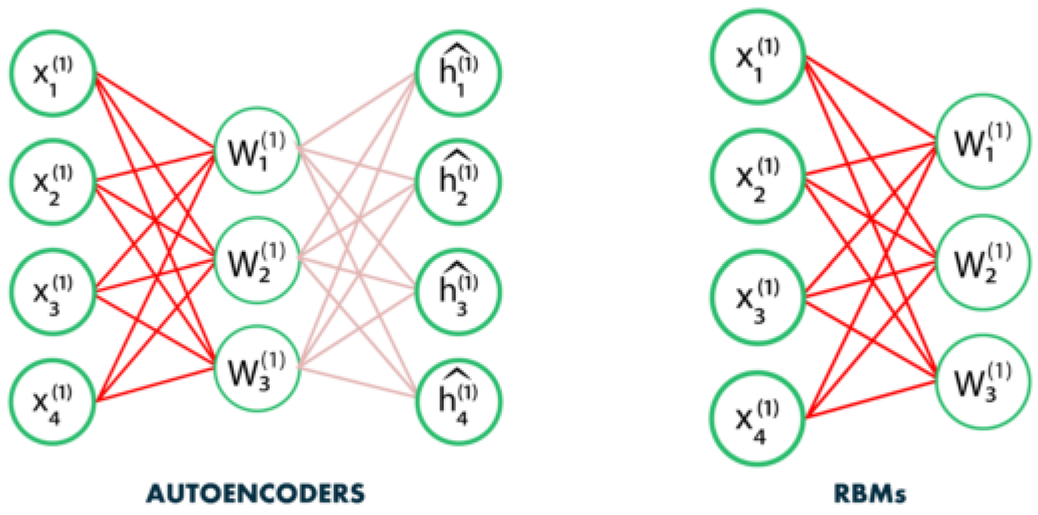
• They build a compressed representation of the input data

Unsupervised learning of unlabeled data. The autoencoder learns directly from unlabeled data. This is connected to the second major difference between multilayer perceptrons and autoencoders.

Learning to reproduce the input data. 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.

Autoencoders vs RBMs

Autoencoder is a simple 3-layer neural network where output units are directly connected back to input units. Typically, the number of hidden units is much less than the number of visible ones. The task of training is to minimize an error or reconstruction, i.e. find the most efficient compact representation for input data.



RBM shares a similar idea, but it uses stochastic units with particular distribution instead of deterministic distribution. The task of training is to find out how these two sets of variables are actually connected to each other.

One aspect that distinguishes RBM from other autoencoders is that it has **two biases**.

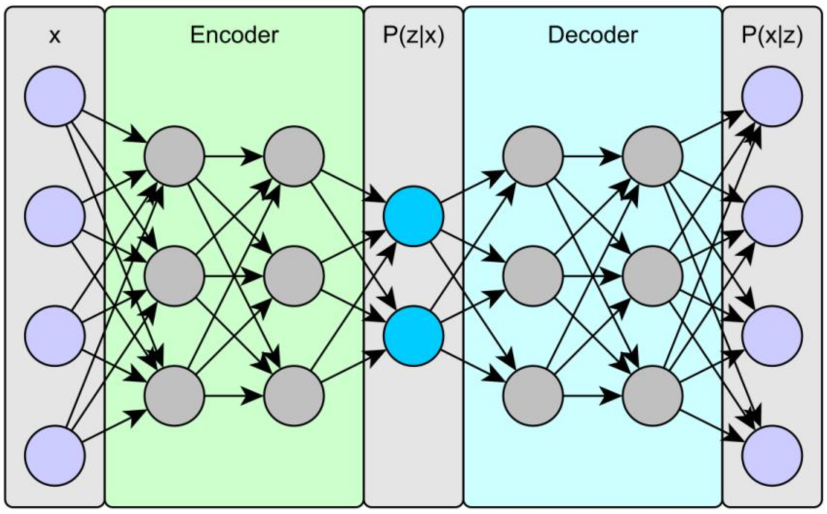
- The hidden bias helps the RBM produce the activations on the **forward pass**, while
- The visible layer's biases help the RBM learn the reconstructions on the **backward pass**.

Variational Autoencoders

A more recent type of autoencoder model is the variational autoencoder (VAE) introduced by Kingma and Welling. The VAE is similar to compression and denoising autoencoders in that they are all trained in an unsupervised manner to reconstruct inputs.

However, the mechanisms that the VAEs use to perform training are quite different. In a compression/denoising autoencoder, activations are mapped to activations throughout the layers, as in a standard neural network; comparatively, a VAE uses a probabilistic approach for the forward pass.

Architecture



Autoencoders are a type of neural network that learns the data encodings from the dataset in an unsupervised way. It basically contains two parts: the first one is an encoder which is similar to the convolution neural network except for the last layer. The aim of the encoder to learn efficient data encoding from the dataset and pass it into a bottleneck architecture. The other part of the autoencoder is a decoder that uses latent space in the bottleneck layer to regenerate the images similar to the dataset. These results backpropagate from the neural network in the form of the loss function.

Variational autoencoder uses KL-divergence as its loss function, the goal of this is to minimize the difference between a supposed distribution and original distribution of dataset.

Suppose we have a distribution z and we want to generate the observation x from it. In other words, we want to calculate

We can do it by following way:

$$p(z|x)$$

But, the calculation of p(x) can be quite difficult

$$p(x) = \int p(x|z)\, p(z)\, dz$$

This usually makes it an intractable distribution. Hence, we need to approximate p(z|x) to q(z|x) to make it a tractable distribution. To better approximate p(z|x) to q(z|x), we will minimize the KL-divergence loss which calculates how similar two distributions are:

$$\min KL\left(q\left(z|x\right)||p\left(z|x\right)\right)$$

By simplifying, the above minimization problem is equivalent to the following maximization problem:

$$E_{q(z|x)}\log p\left(x|z\right)-KL\left(q\left(z|x\right)||p\left(z\right)\right)$$

The first term represents the reconstruction likelihood and the other term ensures that our learned distribution q is similar to the true prior distribution p.

Thus, our total loss consists of two terms, one is reconstruction error and other is KL-divergence loss:

$$Loss=L\left(x,\hat{x}\right)+\sum_{j}KL\left(q_j\left(z|x\right)||p\left(z\right)\right)$$

Differences between Autoencoders and VAE

Autoencoders	Variational Autoencoders
Used to generate a compressed transformation of input in a latent space	Enforces conditions on the latent variable to be the unit norm
The latent variable is not regularized	The latent variable in the compressed form is mean and variance
Picking a random latent variable will generate garbage output	A random value of latent variable generates meaningful output at the decoder
The latent variable has a discontinuity	The latent variable is smooth and continuous
Latent variable has deterministic values	The input of the decoder is stochastic and is sampled from a gaussian with mean and variance of the output of the encoder.
The latent space lacks the generative capability	The latent space has generative capabilities.

Part – II - Major Architectures of Deep Networks

Unsupervised Pretrained Networks

Unsupervised pre-training initializes a discriminative neural net from one which was trained using an unsupervised criterion, such as a deep belief network or a deep autoencoder. This method can sometimes help with both the optimization and the overfitting issues.

In this group, we cover these specific architectures:

• Deep Belief Networks (DBNs)

• Generative Adversarial Networks (GANs)

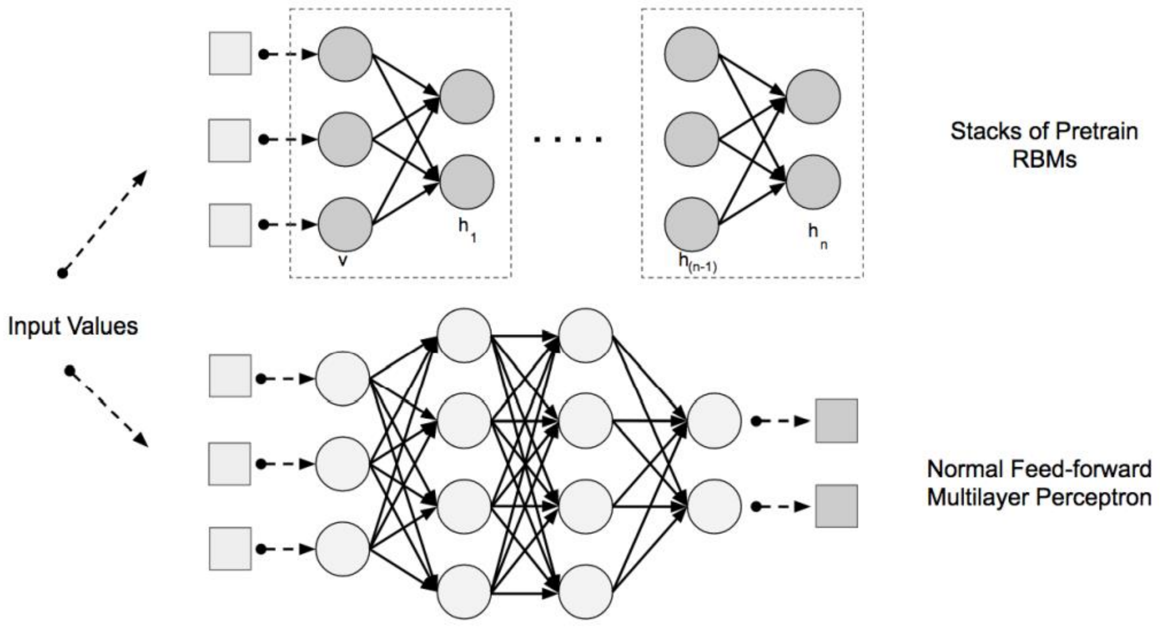
Deep Belief Networks

Deep belief networks (DBNs) are a type of deep learning algorithm that addresses the problems associated with classic neural networks. They do this by using layers of stochastic latent variables, which make up the network. These binary latent variables, or feature detectors and hidden units, are binary variables, and they are known as stochastic because they can take on any value within a specific range with some probability.

The top two layers in DBNs have no direction, but the layers above them have directed links to lower layers. DBNs differ from traditional neural networks because they can be generative and discriminative models. For example, you can only train a conventional neural network to classify images.

Architecture

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



Feature Extraction with RBM Layers

We use RBMs 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 the record, it generates something pretty close to the original input vector.

Learning higher-order features automatically.

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.

Initializing the feed-forward network.

We then use these layers of features as the initial weights in a traditional backpropagation driven feed-forward neural network. These initialization values help the training algorithm guide the parameters of the traditional neural network toward better regions of parameter search space. This phase is known as the fine-tune phase of DBNs.

Fine-tuning a DBN with a feed-forward multilayer neural network

In the fine-tune phase of a DBN we use normal backpropagation with a lower learning rate to do “gentle” backpropagation. We consider the pretraining phase to be a general search of the parameter space in an unsupervised fashion based on raw data.

Gentle backpropagation.

The pretrain phase with RBM learns higher-order features from the data, which we use as good initial starting values for our feed-forward net- work. We want to take these weights and tune them a bit more to find good values for our final neural network model.

The output layer.

The normal goal of a deep network is to learn a set of features. The first layer of a deep network learns how to reconstruct the original dataset. The subsequent layers learn how to reconstruct the probability distributions of the activations of the previous layer. The output layer of a neural network is tied to the overall objective.

Applications

We employ deep belief networks in place of deep feedforward networks or even convolutional neural networks in more sophisticated setups.

Applications of DBN are as follows:

- Recognition of images.
- Sequences of video.
- Data on mocap.
- Speech recognition.

What is the difference between deep belief and deep neural networks?

Deep belief networks differ from deep neural networks in that they make connections between layers that are undirected (not pre-determined), thus varying in topology by definition.

What type of algorithms are DBNs?

Greedy learning algorithms are used to train deep belief networks. In the greedy approach, the algorithm adds units in top-down layers and learns generative weights that minimize the error on training examples. Gibbs sampling is used to understand the top two hidden layers.

What is a deep belief network used for?

Deep Belief Networks (DBNs) have been used to address the problems associated with classic neural networks, such as slow learning, becoming stuck in local minima owing to poor parameter selection, and requiring many training datasets.

Generative Adversarial Networks

Generative Adversarial Networks (GANs) are a powerful class of neural networks that are used for unsupervised learning. It was developed and introduced by Ian J. Goodfellow in 2014. 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.

To understand the term GAN let’s break it into separate three parts

- Generative – To learn a generative model, which describes how data is generated in terms of a probabilistic model. In simple words, it explains how data is generated visually.
- Adversarial – The training of the model is done in an adversarial setting.
- Networks – use deep neural networks for training purposes.

GAN consists of 2 models that automatically discover and learn the patterns in input data.

The two models are known as Generator and Discriminator.

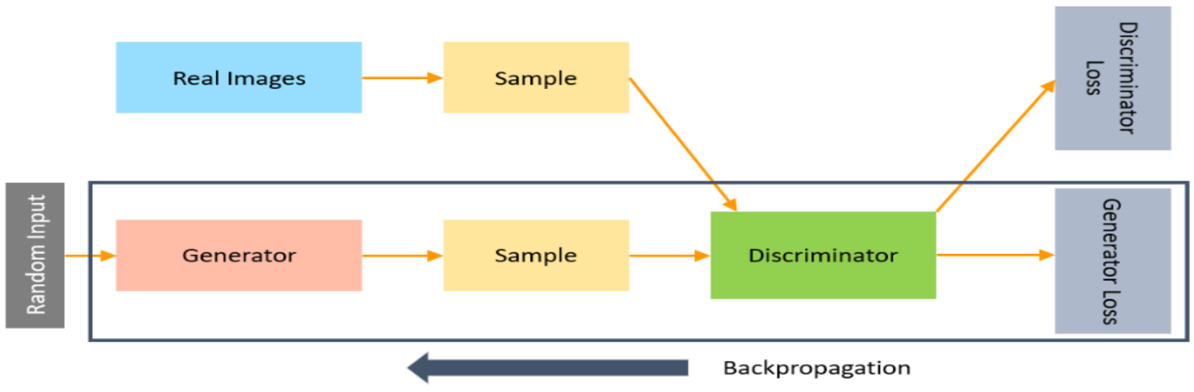
They compete with each other to scrutinize, capture, and replicate the variations within a dataset. GANs can be used to generate new examples that plausibly could have been drawn from the original dataset.

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

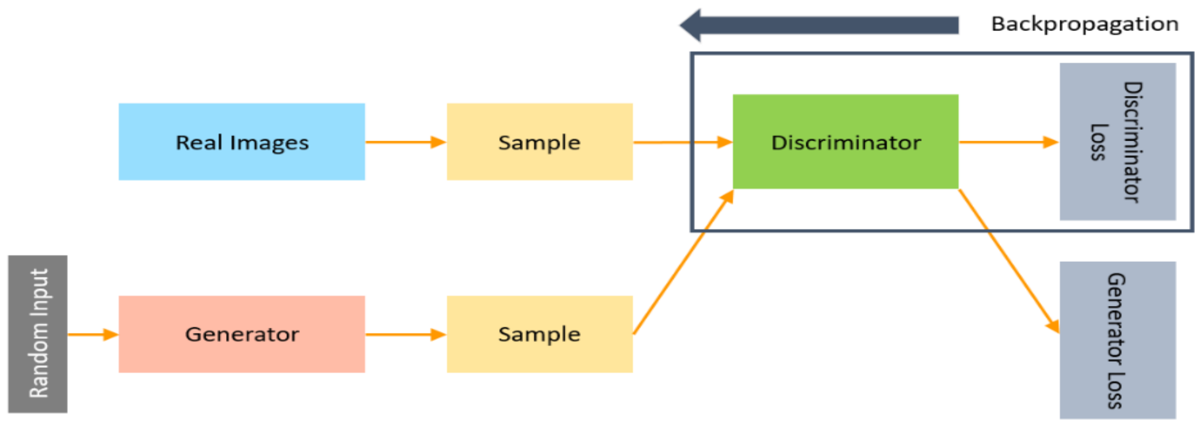


<u>What is a Discriminator?</u>

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:

1. The real data instances, such as real pictures of birds, humans, currency notes, etc., are used by the Discriminator as positive samples during training.

2. The fake data instances created by the Generator are used as negative examples during the training process.

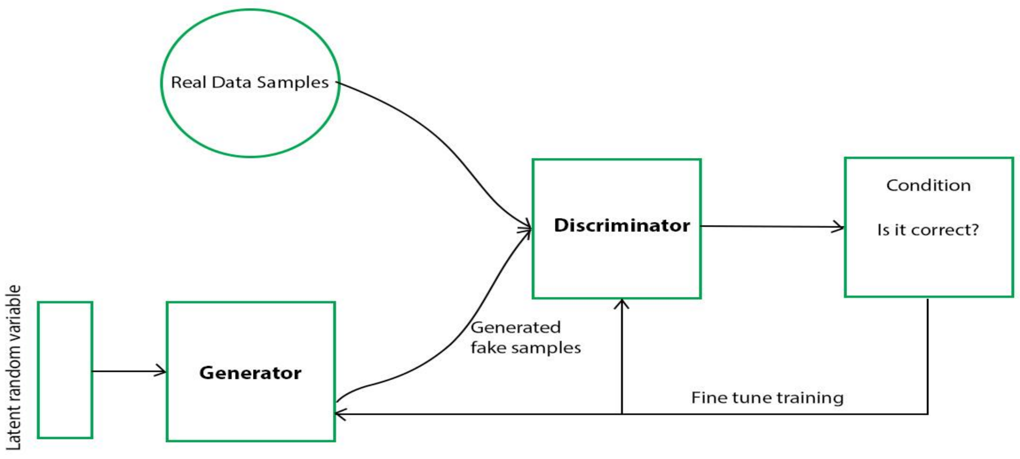


While training the discriminator, it connects to two loss functions. During discriminator training, the discriminator ignores the generator loss and just uses the discriminator loss.

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.

<u>Working of a GAN</u>



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:

Here, the generative model captures the distribution of data and is trained in such a manner that it tries to maximize the probability of the Discriminator in making a mistake. The Discriminator, on the other hand, is based on a model that estimates the probability that the sample that it got is received from the training data and not from the Generator. The GANs are formulated as a minimax game, where the Discriminator is trying to minimize its reward **V(D, G)** and the Generator is trying to minimize the Discriminator's reward or in other words, maximize its loss.

It can be mathematically described by the formula below:

$$\min_{G} \max_{D} V(D, G)$$

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

where,
G = Generator
D = Discriminator
Pdata(x) = distribution of real data
P(z) = distribution of generator
x = sample from Pdata(x)
z = sample from P(z)
D(x) = Discriminator network
G(z) = Generator network

Variations of GAN

Deep Convolutional Generative Adversarial Networks

One variant of GANs is the Deep Convolutional Generative Adversarial Network (DCGAN). DCGAN is one of the most popular also the most successful implementation of GAN. It is composed of ConvNets in place of multi-layer perceptrons. The ConvNets are implemented without max pooling, which is in fact replaced by convolutional stride. Also, the layers are not fully connected.

Conditional GAN (CGAN)

CGAN can be described as a deep learning method in which some conditional parameters are put into place. In CGAN, an additional parameter 'y' is added to the Generator for generating the corresponding data. Labels are also put into the input to the Discriminator in order for the Discriminator to help distinguish the real data from the fake generated data.

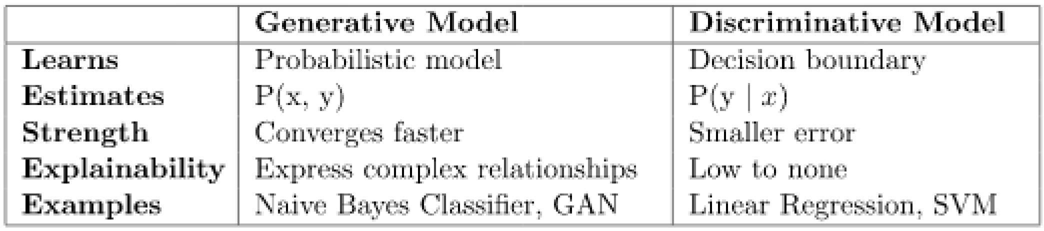
GAN vs Variational Autoencoder

GANs focus on trying to classify training records as being from the model distribution or the real distribution. When the discriminator model makes a prediction in which there is a difference between the two distributions, the generator network adjusts its parameters. Eventually the generator converges on parameters that reproduce the real data distribution, and the discriminator is unable to detect the difference.

With variational autoencoders (VAEs) we're setting up this same problem with probabilistic graphical models to reconstruct the input in an unsupervised fashion. VAEs attempt to maximize a lower bound on the log likelihood of the data such that the generated images look more and more real.

Another interesting difference between GANs and VAEs is how the images are generated. With basic GANs the image is generated with arbitrary code and we don't have a way to generate a picture with specific features. VAEs, in contrast, have a specific encode/decode scheme with which we can compare the generated image to the original image. This gives us the side effect of being able to code for specific types of images to be generated.

Differences between Generative and Discriminative Model



	Generative Model	Discriminative Model
Learns	Probabilistic model	Decision boundary
Estimates	P(x, y)	P(y | x)
Strength	Converges faster	Smaller error
Explainability	Express complex relationships	Low to none
Examples	Naive Bayes Classifier, GAN	Linear Regression, SVM