Chapter 11: Self-Supervised Neural Networks

In the previous chapter, we have described CNN and RNN as two popular neural networks. All algorithms we have described in the previous chapter are supervised learning algorithms. It means we train a model and use the trained model on the test dataset. Then, the accuracy of the result will be evaluated, and the optimizer will try to improve weights in the next epoch.

This chapter focuses on *self-supervised deep learning* approaches, which are known as *unsupervised* or *generative neural networks* as well. Most supervised models we have learned are doing either classification or regression, but self-supervised algorithms are doing *reconstruction*. Many interesting achievements with AI that have happened recently are based on algorithms that we introduce in this chapter. For example, coloring a black and white photo, restoring old photos, converting a photo into a movie, drawing an image based on the given text, etc. Take a look at Figure 11-1, we feed an old photo to a neural network that uses a generative model, and it colorizes the old photo. A text-to-image generation is shown in Figure 11-2, with DALL E v2 [Ramesh '22]. This model is trained to construct images based on the given text, that are described in the caption of the figure.

Figure: 11-2: Text-to-image construction example. (Right) DALL E v2, text prompt: "A happy potato on the beach drinking Pinacolada". (Left) Stable Diffusion XL, text prompt: "A frog talking behind the microphone for a crowd ".

Figure: 11-1: An example of black and white picture which is colorized by generative neural network. This image black and white to color image translation is implemented via https://deepai.org.

We start this chapter by describing Self Organizing Maps (SOM) or Kohonen Maps, an old neural network that its main objective is to change the representation of the give data. It can be used for different tasks, including dimensionality reduction while maintaining the characteristics of the original dataset, classification, clustering, and even solving traveling salesman problem[1]. Next, we describe Deep Belief Network and Restricted Boltzmann Machine as initial models of generative AIs. Afterward, we describe Auto Encoders (AE) and then Generative Adversarial Networks (GAN), which are generative models and very popular, until the introduction of some contrastive learning algorithms, such as diffusion models. Lastly, we explain Contrastive Learning concepts and their algorithms.

These models lead a new direction known as generative AI (e.g., generating poems, illustration, texts, music, etc.).

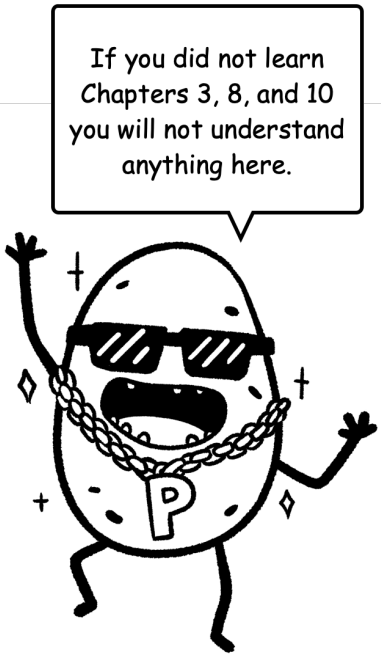
You might think that at the end of this chapter, you are done with deep learning, to be sure you do not feel too happy by reading this chapter, we dedicate another fat chapter about state-of-the-art architectures and models for deep neural networks. Nevertheless, by finishing this chapter and learning all its algorithms, you will be another person. Do not take it as advertisement, but the algorithms of this chapter are extremely exciting to learn.

---

[1] Traveling Salesman Problem (TSP) is a known problem in computer science problem, which tries to answer the following question: “Given the list of cities and distances between each city, what is the shortest path that the salesman can visit each city once and returns to the origin city”

Representation Learning Concepts

Before we start our explanations, we should be familiar with a few basic terms, which we have tried not to explain in earlier chapters. If you start this book from the beginning, at this point in time, you have a good understanding of distributions and probabilities. You are ready to learn new topics. If you skipped those chapters, we strongly recommend you to read Chapter 3, Chapter 8, and Chapter 10 before starting to read this chapter. Otherwise, you will have trouble understanding this chapter.



Self-supervised neural networks have a common characteristic, i.e., instead of training the model on the data in its original dimensions (high dimensional space), these models bring the data into low dimensional space (**latent space**) and train their model in the latent space. This means that each data point in the latent space is a representation of one or more data points in a high dimensional space. Latent space is located in hidden layers. We can call latent space features as *features that best describe the characteristics of data*. Features that are not important to describe the characteristics of the data, either do not exist in the latent space or their impact is reduced there. For example, to construct an artificial human face, the placement of eyes and nose are important and exist in the latent space, but a dot on a cheek is unimportant and will be removed when the data gets into latent space.

Variables inside the latent space are referred to as **latent variables**, and other variables that we can observe, such as input or output, are referred to as **observed variables**.

Generative vs. Discriminative Model: Machine learning algorithms that build models can be classified as generative or discriminative models. Assuming our input data is *x,* and the output that we intend to predict is *y*. A **generative** model uses *joint probability* to make a prediction or an inference, $p(x, y)$. One the other hand, a **discriminative** model uses *conditional probability* $p(x \mid y)$. We can say a generative algorithm cares about *how data has been generated*. These models try to learn the *"distribution" of the training data*, e.g., GMM (Chapter 4). *Once a generative model learns the distribution of the train set, then it can decide about the labels for the test dataset based on the distribution.* In contrast, a discriminative model *discriminate between different data points, disregarding how they have been generated*. A discriminative model tries to *find a decision boundary to separate classes*, e.g., Naive Bayes, logistic regression, or SVM (Chapter 9).

Keep in mind an incomplete but easy explanation about generative models: *algorithms that deal with the data distribution are generative models*. By associating the distribution of data to the generative model, we could remember this definition.

Since Generative models take into account dependencies between data points, this makes them very powerful algorithms. David Foster [Foster ’19] has a good sentence to remember the advantage of generative algorithms: *by sampling from a generative model, we can generate new data*. For example, to generate images, we need generative models because each pixel in an image is highly correlated with all other pixels in that image.

Images are complex and their features are highly correlated and latent, every pixel has a relationship with every other pixel in the image. We, as a human, can not infer their relationships, but neural networks can. Discriminative models do not take into account feature dependencies, and they are incapable generating synthetic data.

Discriminative models usually operate with labels, and thus they have supervised learning algorithms, but generative models do not use labels, and thus they are unsupervised algorithms. However, to keep them separate from traditional unsupervised learning algorithms, scientists call them self-supervised models

Deterministic vs. Stochastic Model: A **deterministic model** is a model that *its’ outputs are determined by the model parameters and initial condition of the dataset*. A **stochastic model** is *a model that has randomness in it.* Therefore, every execution of a stochastic model might have different results. An example of stochastic models is genetic algorithms (check Chapter 6) or manifold dimensionality reductions, tSNE, and UMAP (check Chapter 7). Any approach that use heuristic methods to find result, belong to the category of stochastic models.

Generative models are all stochastic and not deterministic because they sample from the data distribution and do not deal with the exact values of original data. This impose some limitations of them to use in end-user applications, but there are some remedies which we do not explain them here.

Self Organizing Maps (SOM)

One of the oldest practical artificial neural networks is Self Organizing Maps (SOM), which were invented back in 1982 [Kohonen ’82], in Finland, where the Linux operating system and Nokia (the undestroyable smartphones of early 2000) came from.

SOM changes the representation of the data, it can reduce dimensionality of data, and its results can be used for clustering, classification, etc.. We feed a multi-dimensional dataset into a SOM, it provides us with a low dimensional representation of the input dataset. Similar to other

dimensionality reduction methods that we have described in Chapter 6, the SOM algorithm projects data into lower dimensional space, while maintaining the distances between data points.

SOM is a using very special type of ANN. It does not have an activation function, it does not have hidden layers, and from the input layer, it gets directly to the output layer. Besides, its weight assignment process is not based on Backpropagation it uses a different approach for weight assignment. In other words, the concept of weights while discussing SOM differs from other neural networks.

The input layer of SOM is a vector equal to the number of features. For example, if we have five dimensional dataset, e.g., a table with five columns, the input vector includes five neurons.

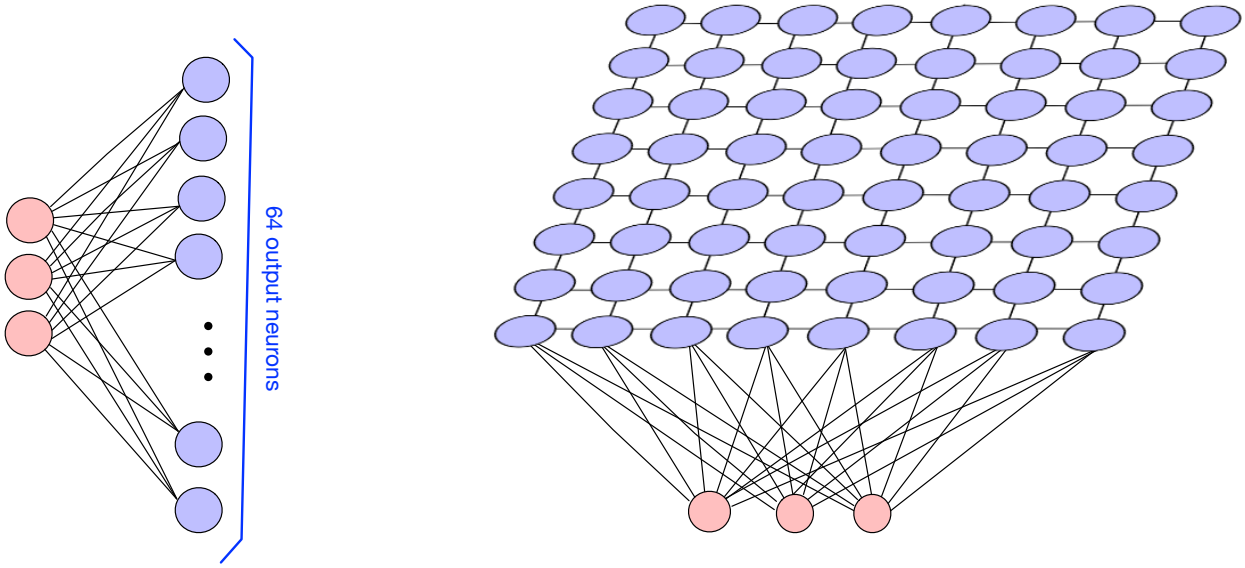


Figure 11-1: A sample SOM, which its red neurons are input and blue ones are output. Until now, we show neural netowkrs like the left side, but SOM is usually shown as the right side of this figure, which is a rotated version of the left side (matrix of neurons).

Assuming the dataset has n number of data points, it is recommended to have $5 \times \sqrt{n}$ neurons in the output [Tian '14]. For example, if our dataset includes 164 records, it is recommended to have $5 \times \sqrt{164} = 64$ output neurons. Take a look at Figure 11-1, there we have a dataset that has three features (three input neurons), our dataset includes 164 records, and the number of output neurons is $5 \times \sqrt{164} = 64$.

The output layer of SOM is also called the "topographical map", which presents the data in a lower dimension. Usually, the output of SOM is presented on a colored network of circles or hexagons in two dimensions. Colors present labels of the data or group of data and each hexagon or circle presents one output neuron (see Figure 11-2). The patterns of color correspond to the distribution of data. Usually, by looking at the topographical map we could identify clusters of data, or we could compare two or more topographical maps to identify some correlations. For

example, by collecting many motherly advices, we realize that there is a correlation between Life

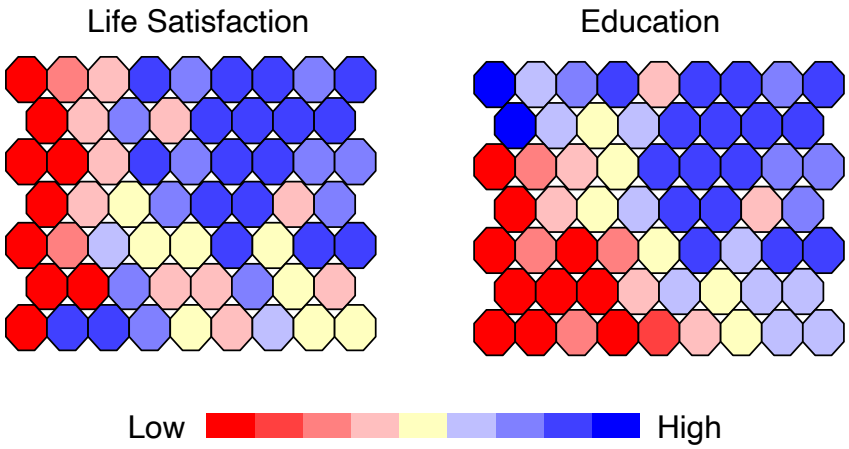


Figure 11-2: Two hexagonal topographical map, which are output of SOM algorithm on a multi feature dataset.

Satisfaction and Education.

Higher dimensional SOM is also possible, but it is not common. The blue layers in Figure 11-1 are the lower dimensional representation of the input dataset. The output layer of the SOM is called 'lattice', which is a specific mathematical structure. In simple words, lattice is a collection of data points that are arranged in a regular pattern, kind of like a grid, and have a special relationship to each other that helps us understand their properties.

SOM algorithm is implemented in four steps (initialization, competition, cooperation and adaption) as follows.

(i) Initialization: All weights of the neural network are initialized with a random number.

(ii) Competition: for every single input tuple (e.g., one record of a table), SOM computes a distance between the given input vector and weights of each node by using the following euclidean distance, which is called discriminant function.

$$distance = \sqrt{\sum_{i=1}^{D} (x_i - w_{j,i})^2}$$

Here, x_i is the input neuron in high dimensional space, i is the index of input neuron, D is the dimensions of the input dataset or number of features, j is the index of output neuron, and $w_{j,i}$ is the weight between output neuron j and input neuron i. The node that has the smallest value is called the winner (o_2 in Figure 11-3) neuron.

To better understand this process, let's take a look at Figure 11-3. There we fed one record from a table with three columns into the network. Then for every single output node, the distance between this node and the input record is calculated. o_2 is the winner node, which has the smallest value. The winner node is also called *Best Matching Unit (BMU)*.

Because of this step in some literature, SOM is called competitive learning. In summary, at every round (i.e., a new row of data is processed), there is competition among neurons, and only one neuron gets activated. This neuron is referred to as "the winner takes all" neuron.

(iii) Cooperation: now a circle with σ radius is drawn around the BMU, and all nodes that fit inside this radius are assumed to be the neighbor of the BMU. The radius size is dynamic. It starts with a large size and can cover all of the output layer's lattice, and then it gets smaller in each round (after a new record is processed), as shown in Figure 11-4. Each time a new record is fed into a network, the radius size will be changed (reduced) by the exponential rate.

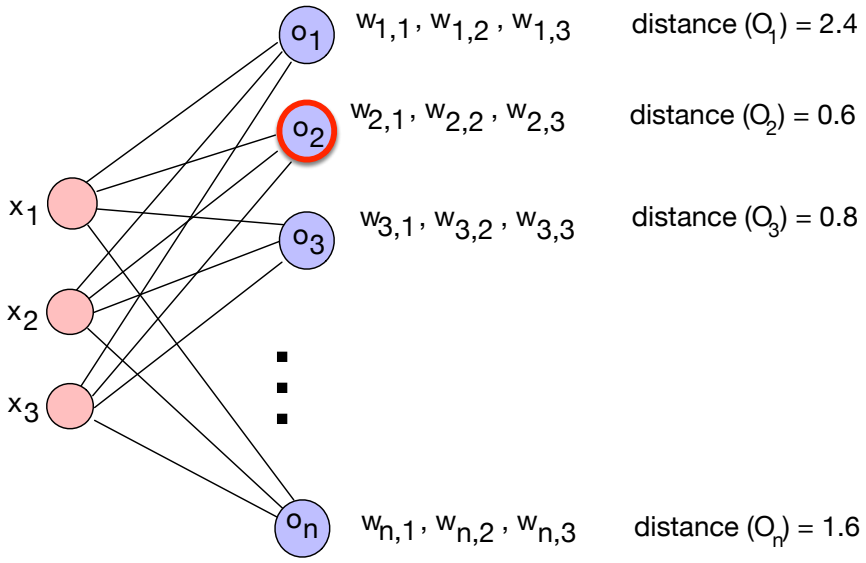


Figure 11-3: For each output node in the SOM a distance between the given input vector and weights of each node is calculated. Then the node with the shortest distance is selected as the winner, i.e. here is o_2 and we mark arounds it with red.

(iv) Adaptation: After the BMU is specified, each node (output neuron) inside the radius adjusts its weights to be similar to the BMU. The neighbor nodes, who are closer to BMU get higher weight updates, and neighbor nodes apart from the BMU get lower weight updates. This process is visualized in Figure 11-4 by using the different intensities of the same color.

When a new record is processed, the BMU changes (from the red dot to the green dot in Figure 11-4) and again weights of the new BMU neighbors will be updated to be similar to the BMU.

Steps (ii) to (iv) will be repeated iteratively until all input records got processed.

As we have explained, one common approach to present SOM output is using hexagonal topological maps (Hex map), see Figure 11-2. Hex map tries to keep similar objects close to each other and dissimilar objects distant from each other. By looking at Hex maps, we can identify correlations between features of the dataset and use the map to identify clusters of data. Nevertheless, SOM is not a clustering algorithm or classification algorithm itself. Instead, its changes the representation of the data and this result can be considered as a clustering. In

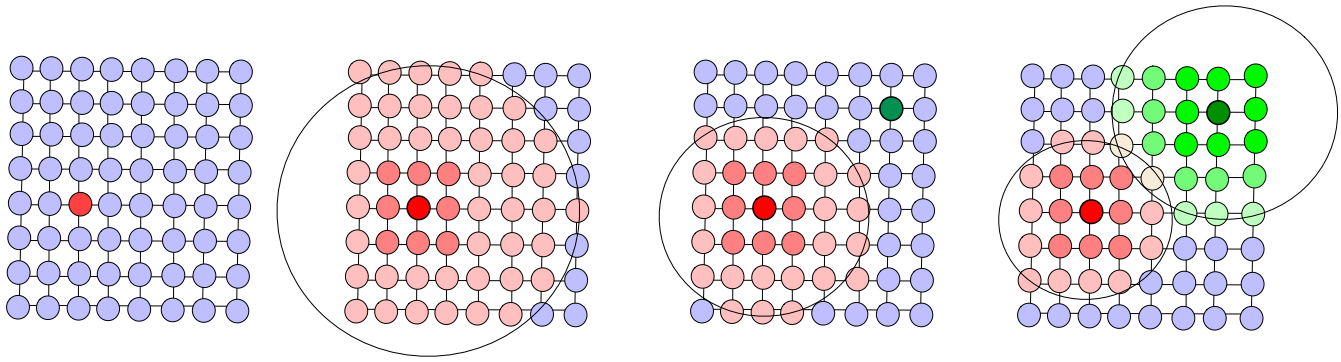


Figure 11-4: Output layer of SOM. From left to right, the first input record is processed, a BMU is identified with dark red, and with a large radius neighbor nodes get its colors. Then, for the second input record, the green dot is identified. The red circle gets smaller (with exponential decay rate) and this process continues for other records.

particular, by assigning each input data point to the cluster represented by the winning neuron in the SOM.

As another example for comparing Hex maps, take a look at Figure 11-2, which shows the result of applying SOM on a dataset that includes the life satisfaction score and education level of several individuals. The result is shown in two hexagonal topological maps. By looking at these two images, we might see that higher education has some correlation with life satisfaction. Higher education is usually correlated with life satisfaction, but as it is presented from these two maps, it is not necessarily always true. We can see at the bottom of the Life Satisfaction Map that there are some few points with low education but high life satisfaction.

Usually, after transferring the data into the lower dimension, it is easier to perform clustering, classification, etc. It is better to experiment with different representation such as SOM and others that we explain like Autoencoder, then decide.

Boltzmann Machines

All previous models we have explained, e.g., ANN, CNN, RNN, and SOM get the input data, process them, and provide output. We can say they include train (fit) and then test (predict).

From now on, in this chapter, the neural networks we explain will perform the prediction on the same data, similar to the unsupervised learning algorithms we learned in Chapter 4 and Chapter 5 and the dataset is not split into train and test sets. These algorithms learn the latent representation of the input data. Latent representation has characteristics of the original data, while some of its noise might get removed.

Other neural networks have a direction, and they start from the input layer and end in the output layer. Boltzmann Machine network, however, does not have a direction. It means that once a neuron acts as an input neuron, and another time it acts as an output neuron. Therefore, unlike previous networks, it does not have output layers. Besides, all neurons in the Boltzmann machine are connected to each other, even input neurons are connected to each other.

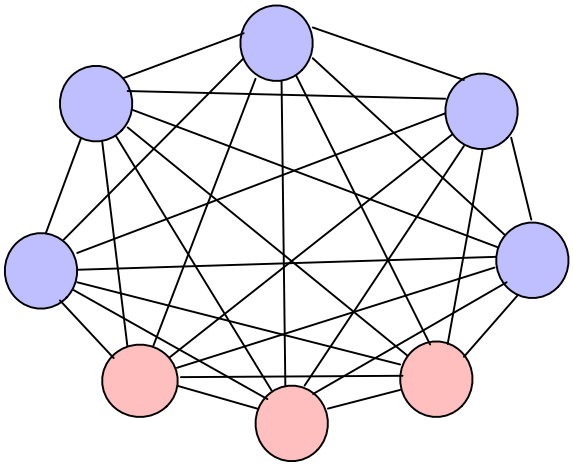


Figure 11-5: a Boltzmann machine, hidden nodes are presented in blue and visible nodes (input neurons) in red color.

Boltzmann machines, similar to SOM are shallow networks and they have only two layers, a **visible (observed) layer,** and a **hidden layer**. The visible neurons are the information that we can measure and hidden layer neurons are information that we can not measure. Figure 11-5 presents a simple Boltzmann Machine.

The most prominent difference of Boltzmann Machine with other neural networks is in addition to the input value that we give to the algorithm, *the algorithm itself generates inputs for its next iterations as well*. It might sound weird, but it is how it works. In particular, the output of one epoch will be considered as the input of the next epoch. Since the Boltzmann machine generates its state, it is called a *generative model*.

The idea of the Boltzmann machine came from physics and the Boltzmann distribution (check Chapter 3). Boltzmann machines are **Energy Based Models (EBM).** EBMs are popular when we need to build generative models with high dimensional datasets that have sophisticated distribution. The ultimate goal of the Boltzmann machines is to iterate the network until it finds the thermal equilibrium (a characteristic of Boltzmann distribution).

We, as prominent data scientists, need to remember that while working with EBMs we use **the energy function** instead of the cost function. The objective of the energy function is to *capture the relative likelihood of different configurations of the inputs*. In other words, the Energy function assigns energy (scalar value) to each possible configuration of the inputs. The result is a

probability distribution over the inputs. The energy function assigns a scalar value, called the energy, to each possible configuration of the inputs. The probability of a particular configuration is then computed using the energy function and the Boltzmann distribution. *When a system is in thermal equilibrium, its energy is minimized, and the energy function reaches its lowest value.*

An *energy function describes the problem*, but a cost/objective/loss function is something we give to the algorithm as input to *calculate the differences between the predicted value to the correct output*. LeCun et al. [LeCun '06] stated that "*A loss score is minimized at the training (learning) phase, and energy is minimized at the testing (prediction) phase".*

Restricted Boltzmann Machine (RBM)

Restricted Boltzmann Machine (RBM) is a unidirectional graph or a neural network with two layers [Ackley '85], similar to other Boltzmann machines, a visible layer, and a hidden layer. It is one of the early version of self-supervised learning used for dimensionality reduction and feature learning.

Boltzmann machine neurons are all connected to each other, but RBM has *no connection between the neurons of the same layer,* and there is *no connection between visible neurons* and *no connection between hidden neurons*, but there are connections from each visible neuron to all hidden neurons and vice versa. Because of these limitations, the word restricted is used to reflect these characteristics in RBM. Compare Figure 11-5, which is a Boltzmann machine to Figure 11-6, which is an RBM. These small differences between RBM with the Boltzmann machine make the energy function of the RBM much simpler than the Boltzmann machine's energy function.

RBM had different use cases, it could be used for dimensionality reduction, classification, regressions, topic modeling, image denoising, etc. However, recently it has been substituted by more recent architecture, which we explain later. Nevertheless, to learn them we should be familiar with RBM as well.

Usually we use a Binary RBM (Bernoulli RBM), it receives input in its visible layer as a binary vector, $v \in \{0,1\}$ and its hidden layer is also a binary vector, $h \in \{0,1\}$. Therefore, we can use binary encoding, such as one-hot encoding to prepare its input data. R

BM configures its weight and biases based on the given inputs, during several iterations between the visible layer and hidden layer. *The hidden layer's neurons are representing features* that are used to reconstruct the data. Note that the RBM algorithm can not label hidden neurons itself, but we might

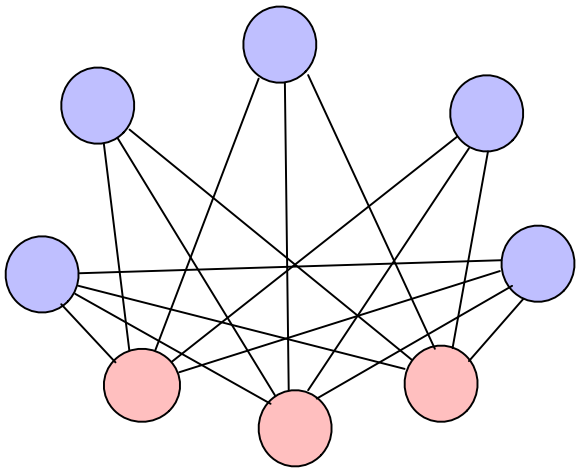
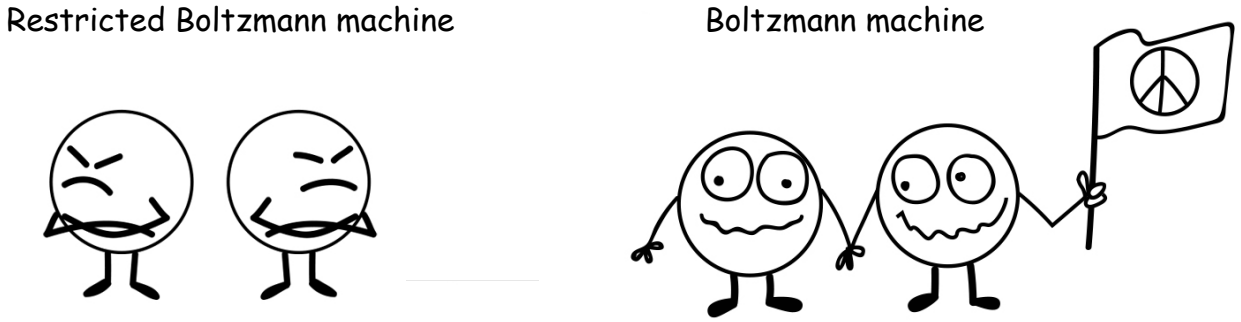


Figure 11-6: a simple Restricted Boltzmann Machine, hidden neurons are presented in blue and visible neurons (observation) in red color. You can see it is very similar to Figure 11-5, the only differences is that nodes are disconnected.

understand them and realize which neuron is presenting which features based on the input, and thus activates that particular neuron.

Let's do a brief review of what we have explained. RBM is an EBM model, and EBM models are generative models which learn the underlying distribution of the data (training set). Once a model learns the underlying distribution of the training set, it can produce new data points that match the learned distribution (underlying distribution of the training set).



In other words, neural networks that we have learned until now are identifying a mapping between input and output neurons, but RBM defines the *probabilistic distribution* instead of mapping between input-output neurons.

Assuming i presents the index of a visible neuron (v), and j presents the index of a hidden neuron (h), the energy function of RBM will be written as follows:

$$E(v,h) = -\sum_{i}^{D} a_i v_i - \sum_{j}^{M} b_j h_j - \sum_{i}\sum_{j} v_i h_j W_{ij}$$

In the above equations, D is the total number of visible neurons and M is the total number of hidden neurons, v is a vector of visible neurons (units), h is a vector of hidden neurons (units), a is a bias of visible neurons, b is the bias of hidden neurons and W_{ij} is a matrix of weights between v_i and h_j. An RBM network uses this energy function to find patterns in the data by *reconstructing the input*.

This equation shows the energy as the sum of three terms, *visible neurons and their biases*, *hidden neurons and their biases*, and a *combination of hidden, visible neurons and their weights* (edges that connect visible and hidden neurons together). We can see from the described equation that RBM has two biases, b is a hidden layer bias that enables the RBM to produce activations on the forward pass, a is the visible layer bias that enables the RBM to reconstruct the input in the backward pass. Weights (W_{ij}) in RBM are presented in a matrix that each of its rows presents a visible neuron and each of its columns presents a hidden neuron. Figure 11-7 visualizes both a and b biases and weights on a simple RBM with three visible neurons and three hidden neurons. Take a look at Figure 11-7, it visualizes a forward pass and backward pass in RBM.

The intractable problem of Z in RBM: The process of training RBM involves configuring biases (a_i, b_j), weights (w_{ij}), and visible/hidden states (v_i, h_i), toward *minimizing the energy for*

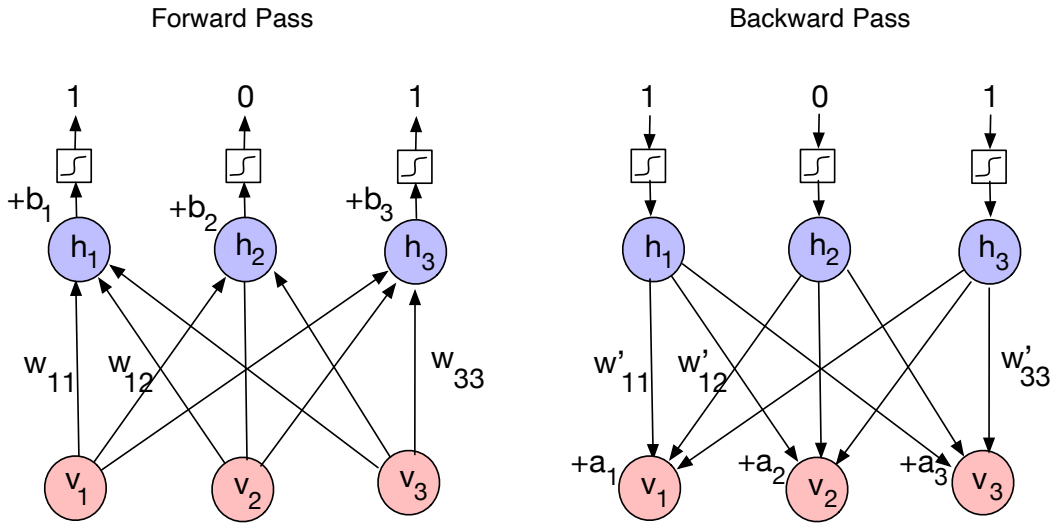


Figure 11-7: Visualizing the forward pass versus backward pass in RBM. The input and output of RBM is binary and its activation function is Sigmoid.

getting into thermal equilibrium, and constructing the $p(v, h)$ distribution. In simple words, the model makes several forward and backward passes between the visible layer and the hidden layer and configures weight and biases (see Figure 11-7).

In the training phase, the RBM needs to model the joint probability distribution of observed variables along with hidden variables, i.e., $p(v, h)$. The following equation is used to model this joint probability distribution.

$$p(v,h) = \frac{1}{Z} e^{-E(v,h)}$$

Energy function

Visible neuron

Hidden neuron

Partition function

This equation shows that the probability of a state between v and h is inversely related to the energy function. In this equation, Z is called the **partition function** or **normalization factor** and is written as follows: $Z = \sum_{v} \sum_{h} e^{-E(v,h)}$.

As we can see from the above equation, Z is the sum of *all values* for *all possible combinations* of visible and hidden (all possible states). v and h are vectors of 0s and 1s (a.k.a Bernoulli vectors). Assuming D is the number of visible neurons and M is the number of hidden neurons, the number of all possibilities will be $2^M \times 2^D = 2^{D+M}$.

Now, do you smell the smoke? Yes, the smell of smoke comes from the computer that is going to compute Z. It is clear that Z is intractable and it is not possible to compute it with the brute force method (check Chapter 5 for the definition of brute force).

Let's review again what is the problem. The objective of the RBM algorithm is to find all $p(v, h)$ and compute all possible combinations in Z, which is computationally intractable. Therefore, we should look for an approximation method to mitigate this problem.

To identify $p(v, h)$ we can calculate instead $p(v \mid h)$ and $p(h \mid v)$, because of **Gibbs sampling** (Check Chapter 16 for details of Gibbs Sampling). Therefore, using Gibbs sampling we can approximate the $p(v, h)$, via conditional probability and thus get rid of Z.

Neuron Activation Probabilities: By using Gibbs sampling, assuming v presents visible layers and h hidden layer, the probabilities of visible layer and hidden layer in RBM, can be decomposed as the following joint probabilities:

$$P(h \mid v) = \prod_i P(h_i \mid v)$$

$$P(v \mid h) = \prod_j P(v_j \mid h)$$

The activation function for each neuron in RBM is a Sigmoid function, and if a neuron is in state 1 it is activated, otherwise, it is 0. Therefore, given the visible vector v the probability for a single hidden neuron h_j activated is written as follows:

$$p(h_j = 1 \mid v) = Sigmoid(b_j + W_{ij} \,.\, v_i) = \frac{1}{1 + exp(-b_j - \sum_{i=1}^{D} v_i W_{ij})}$$

Respectively, given a hidden vector h the probability for a single visible neuron v_i activated is written as follows:

$$p(v_i = 1 \mid h) = Sigmoid(a_i + W_{ij} \,.\, h_j) = \frac{1}{1 + exp(-a_i - \sum_{j=1}^{M} h_j W_{ij})}$$

RBM Training: $p(v)$ presents the likelihood of distribution of visible neurons and it is calculated as follows:

$$p(v) = \frac{1}{Z} \sum_h^{-E(v,h)}$$

However, since it includes Z, we can not calculate it, and thus we use the energy of visible neurons $E(v)$ to approximate it. $E(v) = -\,log\ p(v)$, we can see $E(v)$ is a negative log-likelihood of $p(v)$, why do we say $-log()$? If you recall, while describing maximum likelihood estimation (MLE) in Chapter 3, we said that instead of maximum likelihood, we use a negative log-likelihood to have a minimization and then inverse it (for the sake of computational efficiency). Therefore we can state RBM includes minimizing the negative log-likelihood for $p(v)$.

The derivative of likelihood with respect to weight equals the expectation of the given data minus the expectation provided by the model.

$$\frac{\partial log\ p(v)}{\partial w_{ij}} = \mathbb{E}_{data} - \mathbb{E}_{model}$$

or

$$w_{ij}(t+1) = w_{ij}(t) + \eta \frac{\partial log\ p(v)}{\partial w_{ij}}$$

Here η is the learning rate. $\mathbb{E}$ denoted the expectation. $\mathbb{E}_{data}$ (positive gradient) means that we update the hidden neurons at the beginning, where visible neurons are the input we give into the network, and hidden neurons are constructed by the initial values of visible neurons.

After a certain number of iterations, visible and hidden vectors are updated, and the model is constructed, i.e., $\mathbb{E}_{model}$ (negative gradient).

The differences between the two expectations will be used to identify the optimal values for weights. This means that after $\mathbb{E}_{data}$ and $\mathbb{E}_{model}$ have been computed, the weight matrix and biases could be computed.

Now a question arises: how did we identify $\mathbb{E}_{data}$ and $\mathbb{E}_{model}$? To identify $\mathbb{E}_{data}$ RBM use the following algorithm:

$S = 0$ `# S is a matrix that has the same shape as the weight matrix W.`

for each v_t *in* (1 *to* D) {`# D is the number of visible neurons`

`# generates a sample of hidden variables given the visible vector v_t`

sample $h \leftarrow p(h = 1 \mid v_t) = \sigma(b + W^T v_t)$

sample $S \leftarrow S + v_t h^T$

}

$$\mathbb{E}_{data} \leftarrow \frac{1}{D} S$$

After we have calculated the $\mathbb{E}_{data}$ with direct sampling, we can calculate $\mathbb{E}_{model}$ with plain Gibbs sampling, but it is too slow. This is due to the fact that one iteration of Gibbs sampling consists of updating all of the hidden neurons in parallel, followed by updating all of the visible neurons in parallel. To improve its training time, the performance RBM uses an algorithm called "Contrasting Divergence (CD)" [Hinton '02]. CD estimates the energy function's gradient using a set of model parameters and the training data as input. CD is making a series of approximations to the true posterior distribution over the hidden units in the network using Gibbs sampling.

Following is a simplified description of the CD algorithm:

$Q = 0$ `# a matrix that has the same shape as weight matrix W and it is used to approximate` p

for(1 *to* k) { `# k is a small number, and it is used to perform k times Gibbs sampling, not until it converges. Of course, larger k results in better accuracy, but at the expense of slower convergence.`

sample $h \leftarrow p(h = 1 \mid v) = \sigma(b + W^T v)$

sample $v \leftarrow p(v = 1 \mid h) = \sigma(a + Wh)$

$Q = Q + vh^T$ `# Re-update the hidden neurons given the reconstructed visible neurons using the same equation`

}

$$\mathbb{E}_{model} \leftarrow \frac{1}{k} Q$$

In this algorithm, instead of iterating *n* times, if we use a condition that continues the loop until it converges, then it is plain Gibbs sampling. However, CD applies a small, subtle change that reduces the number of iterations.

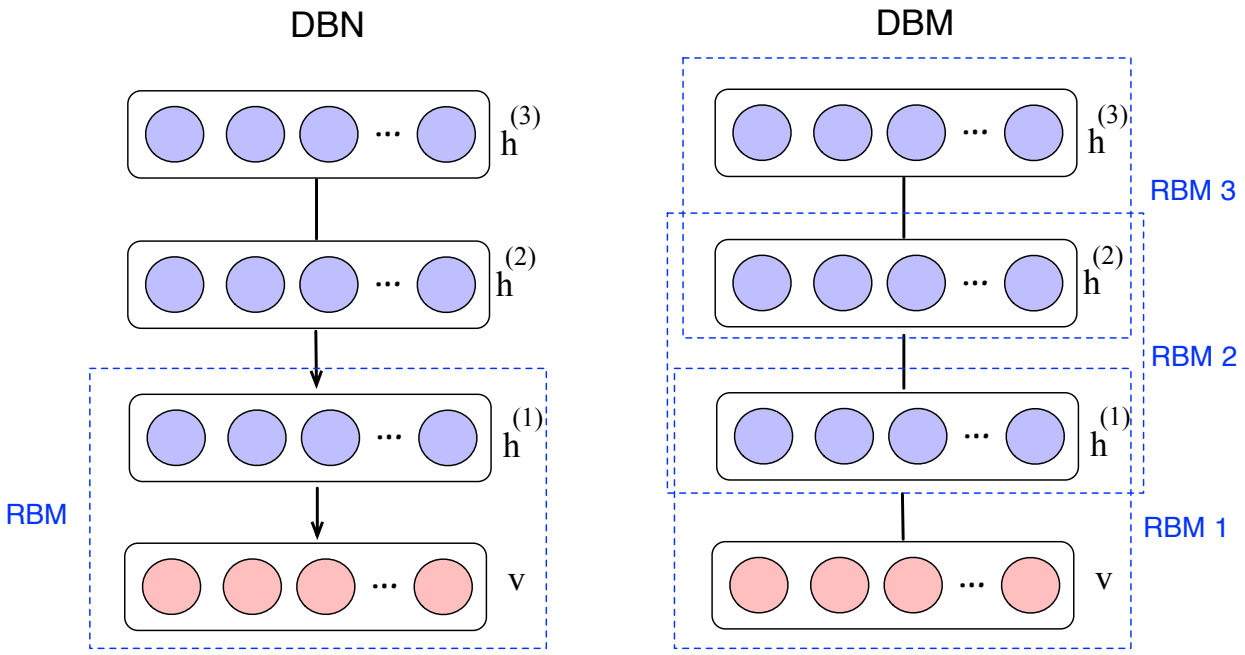


Figure 11-8: A sample DBN and DBM pre-training stage. The output of each RBM (hidden layer) is the visible layer of the next RBM. Arrows in this figure present the direction of the generative model (Gibbs sampling). For example updated values in $h^{(1)}$ layer is used to reconstruct v layer, but in DBN, v layer can not be used to reconstruct $h^{(1)}$.

In summary, the CD algorithm continues to change visible and hidden neurons back and forth in fewer iterations than Gibbs sampling.

Note that the loop does not stop after it converges, instead it performs k number of iterations and then it stops. After both $\mathbb{E}_{data}$ and $\mathbb{E}_{model}$ are specified by the algorithm, assuming ϵ is the learning rate, the algorithm uses gradient ascending (not descent) to compute weight and biases as follows:

$$\Delta W_{i,j} = \epsilon(\mathbb{E}_{data}[v_i, h_j] - \mathbb{E}_{model}[v_i, h_j])$$

$$\Delta a_i = \epsilon(\mathbb{E}_{data}[v_i] - \mathbb{E}_{model}[v_i])$$

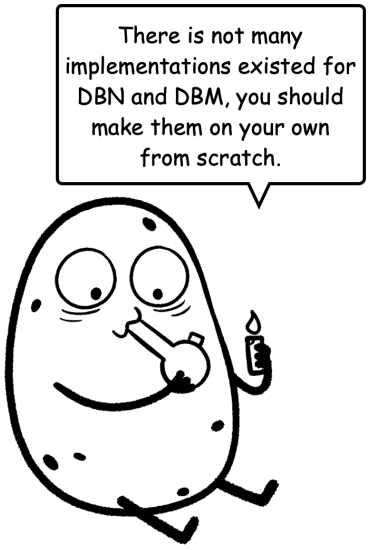
$$\Delta b_j = \epsilon(\mathbb{E}_{data}[h_j] - \mathbb{E}_{model}[h_j])$$

That is enough to know about the RBM algorithm because, as we have explained, it is a historical algorithm. Through learning history, we cultivate an understanding of how later, more practical algorithms have been designed.

Deep Belief Network and Deep Boltzmann Machine

Deep belief network (DBN) and Deep Boltzmann Machine (DBM) are two models based on RBM. DBM is composed of stacking RBM layers on top of each other, and DBN is composed of

stacking hidden layers on one RBM (see Figure 11-8). They both can be used for generative tasks such as image generation and dimensionality reduction.



What is the advantage of stacking multiple RBMs? The first RBM includes latent features that are better than random inputs. The hidden layer of the second RBM includes a combination of hidden features that are better than random inputs, and each layer includes more accurate hidden features.

DBM is trained using the CD algorithm. The training phase in DBN includes two stages, *unsupervised pre-training* and *supervised fine-tuning*. The **pre-training** initializes weights on the network, but every hidden layer is trained based on its given input (only its given input, not the previous layers' data), it is a greedy layer-wise approach (short sighted to one previous layer and not all previous layers), and thus this training can stick in local optima. Because of this limitation, it is called pre-training.

After pre-training, the algorithm performs fine-tuning, this step is used to correct weights in supervised manner with Backpropagation. At the fine-tuning stage, the output layer is added (does not present in Figure 11-8, because this figure focuses on pre-training only) and supervised learning is used to train the network, with the forward and backward propagations. Therefore, fine-tuning assigned the final weight and biases in a supervised manner, but still, due to their pre-training stage, these algorithms are called unsupervised deep learning.

Keep in mind that both pre-training and fine-tuning stages use a greedy approach to train each RBM and configure weights.

In their architectures, a hidden layer of one RBM is the visible layer for the next RBM, as you can see from Figure 11-8. However, at the pre-training stage of DBN its hidden layers are undirected, but the first two layers (visible layer and hidden layers close to it) are unidirectional.

In particular, (v) and second layer ($h^{(1)}$) is unidirectional, and in some literature, it is called *Sigmoid Belief Net.* Other layers are connected together, but they are un-directed. Because they are not part of the generative model and they are used for inferences only. It might seem a bit odd, but it is how this network is presented.

At the pre-training stage, the undirected connection between $h^{(1)}$ and $h^{(2)}$ means that Gibbs sampling of RBM once sample from $h^{(1)}$ to model $h^{(2)}$ and then once from $h^{(2)}$to model $h^{(1)}$, but for a directed (undirected) connection from $h^{(1)}$ to v sampling goes from $h^{(1)}$ to v and not from v to $h^{(1)}$. In other words, it uses a model to generate visible layer data.

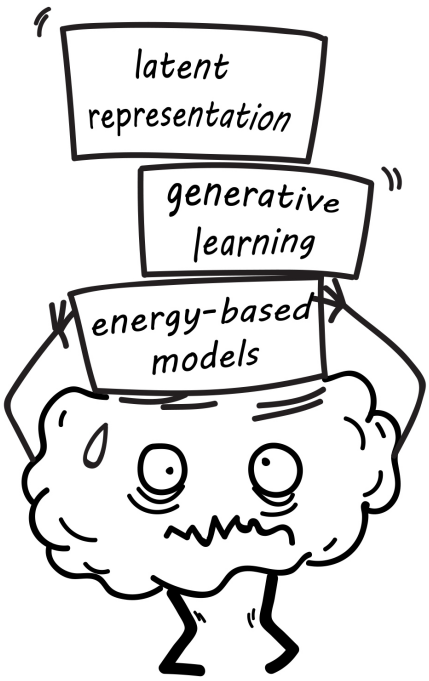
You might ask how the algorithm can skip v to $h^{(1)}$, and how can it start from the top layer $h^{(3)}$? The algorithm does not skip weight assignment (training) from v to $h^{(1)}$. It performs weight

training once and goes up by using the same weights on all layers, until it reaches the top layer, i.e. $h^{(3)}$. Because of that, it is called pre-training. Then, in the next step, the Gibbs sampling and weight adjustment (fine-tunning) start from the top layer until they reach the directed layers.

On the other hand, DBM is just stacking layers of RBMs on top of each other. Figure 11-8, shows all layers in DBM are undirected, but not in DBN.

Why do we need to stack lots of RBM and train them (DBN or DBM)? Why not use a deep feed forward algorithm? The motivation is that since they use RBM uses a contrastive divergence and wake-sleep algorithm [Hinton '95] (we do not explain this algorithm), there is no use of backpropagation in the pre-training stage while using a discriminative model. By not using backpropagation, we get rid of the vanishing or exploding gradient problem that existed at this stage. However, the fine-tuning uses backpropagation, but it starts with well initialized weights in hidden layers.

Similar to MLP these architectures have input, hidden layers, and output layers. The output of each RBM is an input of the next RBM, as it is shown in Figure 11-8. However, unlike other deep learning algorithms, each layer in DBN or RBM learns the entire input. For example, while using CNN the first layer gets the original input, and the next layers work with the convolved input data, not the original input data, but both DBN and DBM they learn the input which is received from the previous layer.



To summarize their differences: a DBN has directed connections between all its layers except for the top two layers, which have undirected connections. However, in DBM all connections are undirected.

DBN uses pretraining and fine-tuning phases for its training, and it is easy to sample from visible and hidden units. DBM uses contrastive divergence for its training, but it requires a complex approach to sample from visible and hidden units.

To summarize their similarities: both use greedy layer-wise training (pre-training) features of RBM, and both use RBM to identify a latent feature of data.

Some literature claims that DBM is better than DBN for learning complex models [Wang ’17], but we recommend experimenting with both architectures on your own and deciding which one to use.

NOTE:

* A basic model of RBM is inspired by Hopfield Network [Hopfield ‘82]. Hopfield network neurons contain binary neurons, which present binary attributes of the dataset. They create a deterministic model (not stochastic unlike RBM) of the relationship among different features by changing weights on edges. If you like to learn more about Hopfield networks, you can check Aggrawal’s book [Aggarwal '18] which has a detailed explanation about it, but it requires a strong mathematical background to understand it.

* By learning the weights the RBM implicitly memorizes the training dataset, therefore we could say the RBM is becoming the representation of the input data which we fed to it. RBM minimizes the energy or maximizes the probability. This feature of RBM enables both DBN and DBM to understand internal representations of the input data that is complex. Also, it can assist to improve the model by adding very few labeled data (semi-supervised learning), but still RBM is called self-supervised learning.

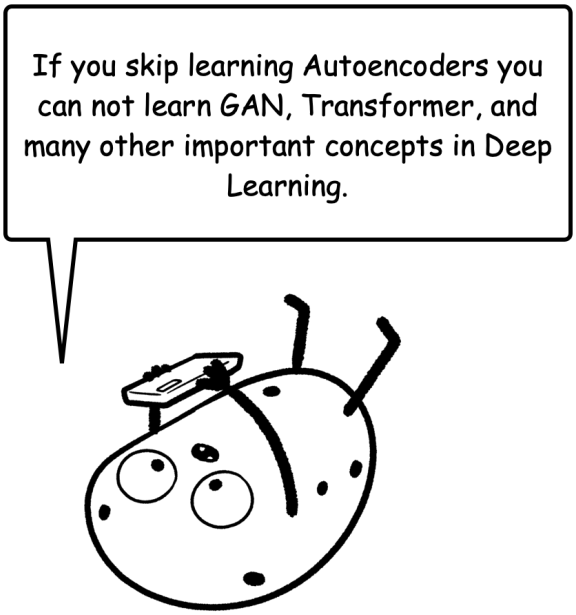
* Contrastive divergence is a popular algorithm for training EBM algorithms. It operates based on the assumption that the *derivative of the log-likelihood function can be defined as differences between two expectations (*$\mathbb{E}_{data}$ *and* $\mathbb{E}_{model}$*).*

* DBN, DBM, and RBM are not widely in use nowadays, because Autoencoders and Generative Adversarial Networks outperform them. Nevertheless, learning them is useful to understand the intuition behind the new algorithms. There are algorithms that are not in use for many years and somebody extracts an idea from them and creates something very useful, like Geoff Hinton whose team created DBN, RBM, tSNE and also applies backpropagation on DNN. Therefore, it is worth learning as much as we can, even algorithms that at the current time m,are not popular.

* Bengio [Bengio ’09] has a good generalization on unsupervised deep learning models, such as DBN and Autoencoders. He states: *“unsupervised training signal at each layer may help to guide the parameters of that layer towards better regions in parameter space”*.

Autoencoders

Previous neural network models we have explained in this chapter, are not state-of-the-art algorithms, but they are important to learn, because they could inspire us while designing our own algorithms. Besides, in some cases due to resource limitations, we can not go for a complex neural network algorithm, and using light algorithms such as SOM could resolve our need. Algorithms are not like technologies that get outdated, they are science and we should learn them as much as we can. Software developers learn technologies and they need to filter what technology to learn and what to filter. However, learning the algorithm is a different policy, and we believe we should learn as much as algorithms we can.



At the time of writing this chapter, in early 2021, Autoencoders were one of the state-of-the-art components of generative AI. They have applications denoising data (e.g., watermark removal from watermarked images), better weight assignment in neural networks, image segmentation, etc.

Autoencoders are neural networks that reconstruct (not copy) their inputs in their output layer and try to make a reconstructed output similar to the original input.

An Autoencoder is composed of two components, an encoder and a decoder. The encoder creates a compressed representation of input data, i.e., transferring the input data into **latent space**. In other words, *latent space is the representation of input data in hidden layers.* Transferring data into a latent space is associated with compression, and this compression removes non-descriptive features (e.g. redundant and noise) while keeping the important features. The decoder uses the latent space to reconstruct the data similar to the original data, but without its unimportant features.

Note that autoencoders' outputs are not going to be identical to their inputs, they provide an approximate copy of the input. While training, the output entails important features of the input data and removes useless features of the input data.

We also use something similar to autoencoders in our daily communications. Let's say you are healthy and ran 12,342 steps in the morning. Then you visit a friend in the evening and tell her you had a productive day because you ran more than 10,000 steps (you compress the data and instead of 12,342 you say more than 10,000). Then your friend visits another friend of hers and tells him, she has a friend (you) who is an athlete, and also that person is reading an amazing data science book. She used to reconstruct the data you provided to her, i.e. you said you "ran more than 10,000 steps" and she compress/decompress it as you are an "athlete".

Take a look at Figure 11-9, we add noise to an image, from the MNIST dataset, and then feed it into an Autoencoder. The output removes the noise because it reconstructs the data by keeping the useful properties of the data and getting rid of non-useful features. The Autoencoder has seen many '4' (plenty of clean images without noise) in the training set, and thus it learns to reconstruct '4', by removing un-important features and only keeping the important ones. If there is no '4' in the dataset, then the denoising was not successful.

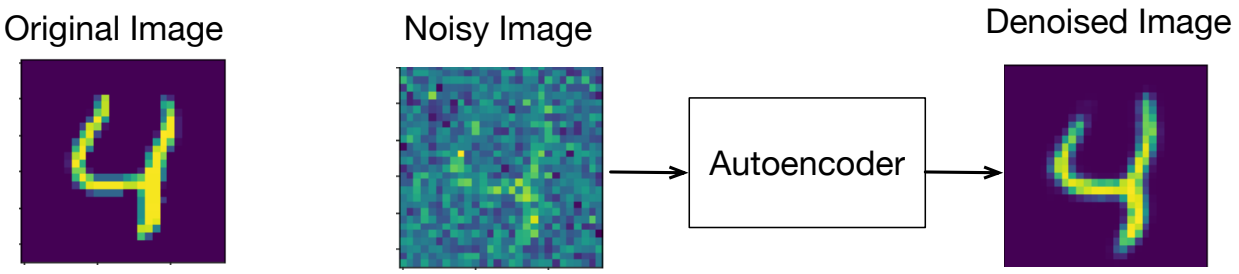


Figure 11-9: Autoencoder is used to Denoise a noisy image.

How do Autoencoders work?

The typical architecture of Autoencoder is presented in Figure 11-10. We have explained that during the encoding stage the encoder removes unimportant features and during the decoding stage the decoder reconstructs the data by using its important feature, but how does this happen? The hidden layers in autoencoders typically have fewer neurons, and thus the features of data (dimensionality) are reduced. This reduction in the number of features has many useful applications and removes unnecessary features.

Autoencoder has at least one hidden layer, that includes the features used to encode the input data and change its representation (reducing its features).

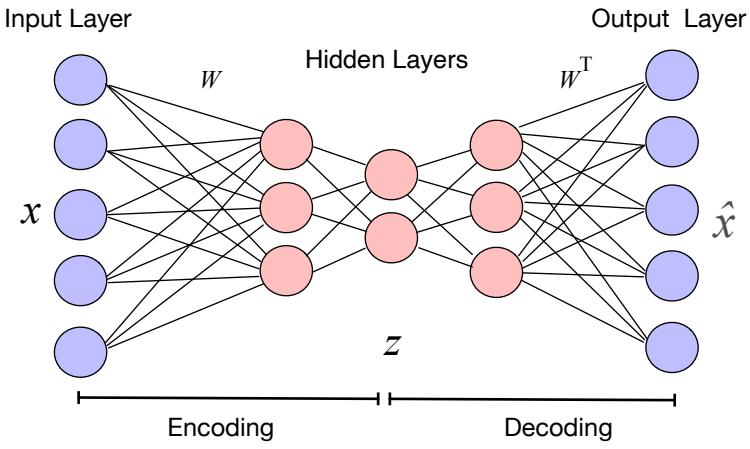


Figure 11-10: A simple one layer autoencoder which has only one layer of hidden layer.

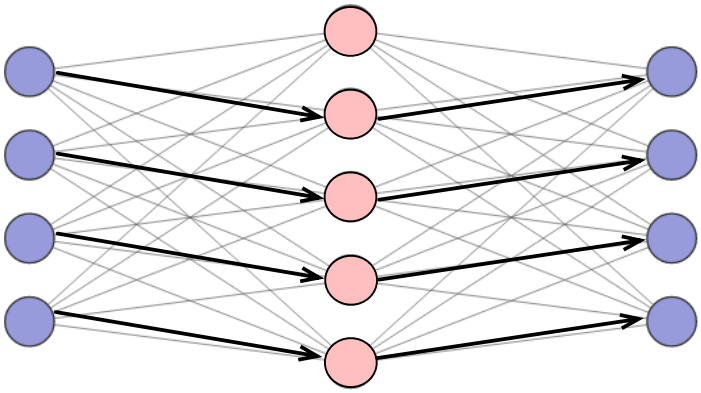


Figure 11-11: An over complete auto encoder that its hidden layer acts as identify function and input nodes are copied exactly as they are into the output node.

Training auto encoder involves three steps, (i) input dataset x is fed into the autoencoder, passing through the encoder to construct the latent space features. (ii) Then, it passes through the decoder and the decoder provides output $\hat{x}$ as a result of x reconstruction. (iii) The cost function calculates the differences between $\hat{x}$ and x. Based on the cost (error) backpropagation algorithm is used to reconfigure weights and biases in the next iteration and backpropagates from the decoder to the beginning of the encoder.

After training Autoencoder either the encoder or decoder could be used separately. For example, only an encoder could be used if our goal is to reduce the dimensionality of the data, or a decoder along with the encoder could be used to generate new data, but with a pattern similar to the given input data. Since the distribution of the generated data (output) is the same as the distribution of given data (input), autoencoders can construct new data, which is similar to the given input data.

We can interpret an autoencoder as a mathematical function, it has an encoder function $f(x)$, it receives input x. Another function is the decoding function g, which receives the encoding result and provides the output $\hat{x} = g(f(x))$. The cost function of autoencoders is to minimize the error of dissimilarity between input and output, $L(x, \hat{x})$ or $L(x, g(f(x))$. A common cost function in autoencoders is "mean square error", and usually, they use non-linear activation functions (e.g. Sigmoid, ReLU, Hyperbolic Tangent). If we use the linear activation function, the dimensionality reduction of Autoencoder will be similar to PCA.

Now that we understand the basics of Autoencoders, we need to be familiar with some specific types of Autoencoders.

Autoencoders whose hidden layers have fewer neurons than their input layer are called **undercomplete autoencoders**, such as the one presented in Figure 11-10. Autoencoders whose hidden layers have more neurons than input layer neurons are called **overcomplete autoencoders**, such as Figure 11-11. An autoencoder is called a **deep autoencoder** if it has more than one layer of hidden layers, such as Figure 11-10. Most of the time, we use deep autoencoder for our problems.

Autoencoders can Cheat

The magic of autoencoder happens when we get an *abstract representation of the input data*, and one way this can be achieved is by having a smaller number of hidden neurons activated. The hidden layers in Figure 11-10 are smaller than the input layer. However, experiments show that increasing the number of hidden neurons can enable the network to construct a more accurate representation of the input data. Therefore, in some autoencoder models which we explain later (Sparse, Denoising, and Contractive autoencoders) hidden neurons in a layer are larger than input neurons. On the other hand, *increasing the neurons of hidden layers to have more neurons than input layers, might cause the network to simply copy/paste the input into output without any changes (i.e. identity function)*, Figure 11-11 visualizes this problem. This error (a.k.a. cheating) should be avoided and in the following, we describe three types of auto-encoders that avoid this problem, while benefiting from having more hidden neurons than input neurons.

Autoencoder Types

Autoencoders are grouped into two categories, regularized autoencoders, and variational autoencoders. Common regularized autoencoders include sparse autoencoders, denoising autoencoders, and contractive autoencoders. They provide a regularization to method prevent the output layer from copying the input layer data (act as an identity function). To simplify the motivation of using "regularizations" in Autoencoders, think about the time we study for a mathematical exam. We need to learn things and we should not memorize them. Because learning enables us to resolve the question, even if we have not encountered it before. However, just memorizing them is a sort of copy/pasting answer and thus if we memorized only limited knowledge and not the concepts, it is infeasible to answer questions we have not encountered before, by using the existing knowledge base.
After we are done with regularized autoencoders we delve into a more advanced autoencoder, "Variational Autoencoders" that can generate data very close to the given input data [Kingma '13].

Sparse Autoencoder

Sparse autoencoders (SAE) [Makhzani '13, Ng '11] are usually overcomplete autoencoders (i.e. their hidden layers are larger than the neurons of their input layer). If hidden nodes are larger than the input, the autoencoder increases the dimension of data (more useful features will be extracted). This is useful for algorithms that are seeking to find descriptive features of the dataset. Nevertheless, adding the extra hidden neurons requires some administration to disallow the autoencoder to act as the identity function (copy/paste of input neurons and cheating).

SAE regularizer disables some neurons at any pass and thus the autoencoder will be limited to work with a smaller number of neurons than input neurons, but in the next pass those disabled neurons will be enabled and a new group of neurons will be disabled (see Figure 11-12). In practice, it is similar to undercomplete autoencoders which use a smaller number of neurons, but in every iteration, it uses a different set of nodes.

The cost function of the SAE is written as: $L(x, \hat{x}) + \Omega(f(x))$, which $\Omega(f(x))$ is the regularizer that receives the encoder output $(f(x))$ as its input. However, unlike regularizers we have explained in Chapter 8, these regularizers do not have weight decay. Instead, they measure the hidden layer activations for each training dataset and add some penalty to the loss function to penalize excessive changes in the activation.

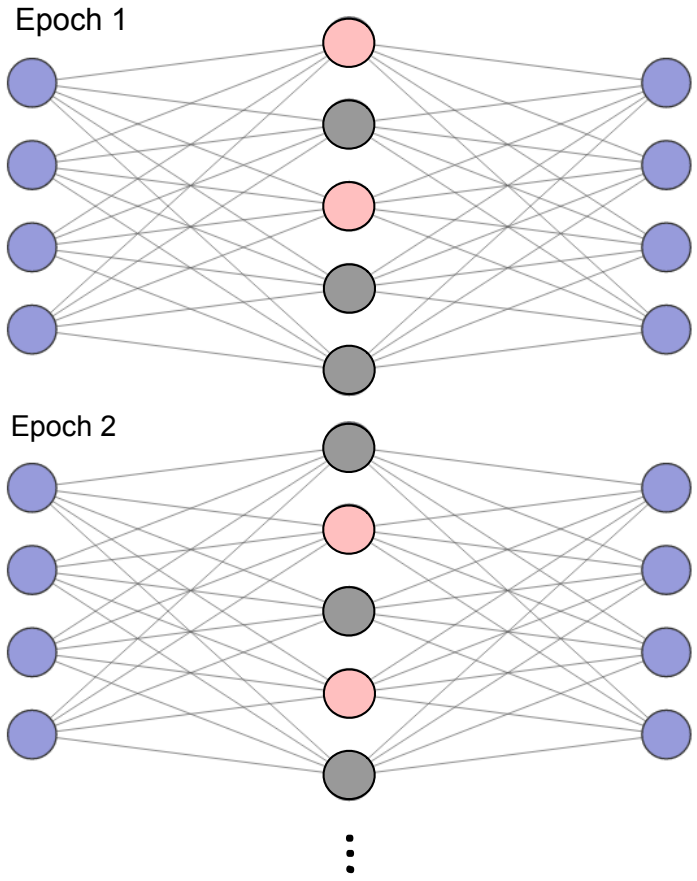
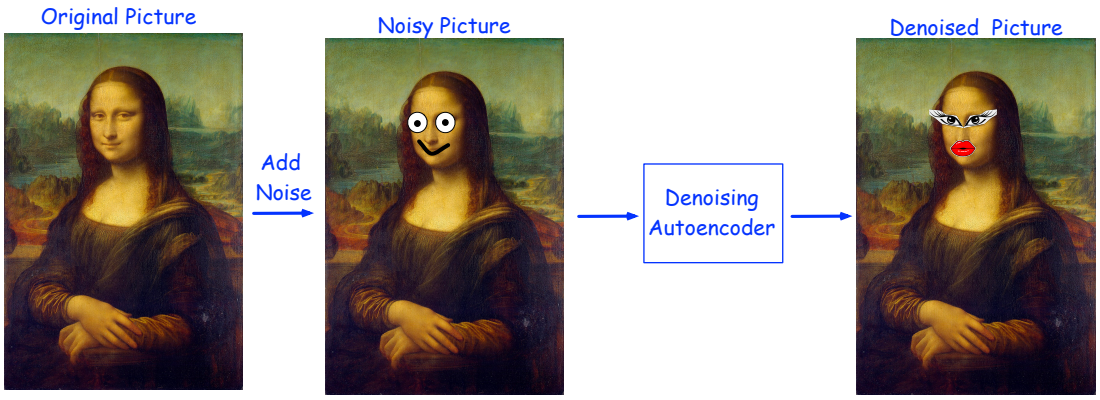


Figure 11-12: An example of sparse auto encoder, which has two neuron of its hidden layer activated at each epoch. The grey neurons are regularized and thus their value is too small to provide any significant impact on the output.

There are two known methods that we use $\Omega(f(x))$, L_1 *regularization* and *KL-Divergence*.
The L_1 *regularization* is using L_1 the norm as a penalization score. Assuming we have n number of hidden neurons, a_i the activation of ith neuron, its cost function will be written as:

$$L(x, \hat{x}) + \lambda \sum_{i=0}^{n} |a_i|$$

We have explained in Chapter 3 and Chapter 6 that KL-Divergence measures the differences between two statistical distributions. The KL-Divergence regularizer defines a **sparsity parameter** ρ that specifies the desired level of sparsity. In other words, it is the *average activation of neurons in a hidden layer* over the sample of m observations (m is the number of data points in the training set). Usually, ρ is set to a small variable such as 0.05.



$\hat{\rho}_j$ specifies the *average activation of hidden neuron j*. Assuming $a_j(x^{(i)})$ is the activation function of the hidden neuron j, and m number of input data gets into neuron j, $\hat{\rho}_j$ is calculated as follows:

$$\hat{\rho}_j = \frac{1}{m}\sum_{i=0}^{m}[a_j(x^{(i)})]$$

The KL-divergence between ρ and $\hat{\rho}_j$ distributions can be used as the penalty function, the penalty will be 0 if $\rho = \hat{\rho}_j$. Therefore, assuming we have n number of hidden neurons in the hidden layer, the cost function of SAE with KL-Divergence penalty is written as follows:

$$L(x,\hat{x}) + \sum_{j=1}^{n} KL(\rho||\hat{\rho}_j)$$

Denoising Autoencoder

The objective of denoising autoencoder (DAE) [Vincent '10] is to have a representation that can handle noise. Figure 11-9 is a denoising Autoencoder. Same as sparse autoencoders, the hidden layers of denoising autoencoders usually have more neurons than the input layer (overcomplete but could also be undercomplete). Therefore, again some neurons get inactive, and a regularizer is used to make those neurons inactive.

The denoising autoencoder first adds some noise to the input data and then feeds the noisy input data, instead of original input data, into the encoder.

There are many types of noise that can be used to make an image noisy, like Gaussian noise, Perlin noise, etc. in Chapter 16 a bit more detailed description of noise signals will be provided.

Figure 11-13 presents the architecture of the denoising autoencoder. Assuming original data is x and the original data with added noise is x', the process of adding noise is a probabilistic process $P(x'|x)$. Nevertheless, the cost function $L(\hat{x}, x)$ does not compare $\hat{x}$ (output data) to x' (input data with noise) instead it compares the $\hat{x}$ (output data) to x (original input data).

As is shown in Figure 11-13, denoising autoencoders change the input neurons first, for example setting some of them randomly to 0. It has a hyper-parameter that specifies what percentages of

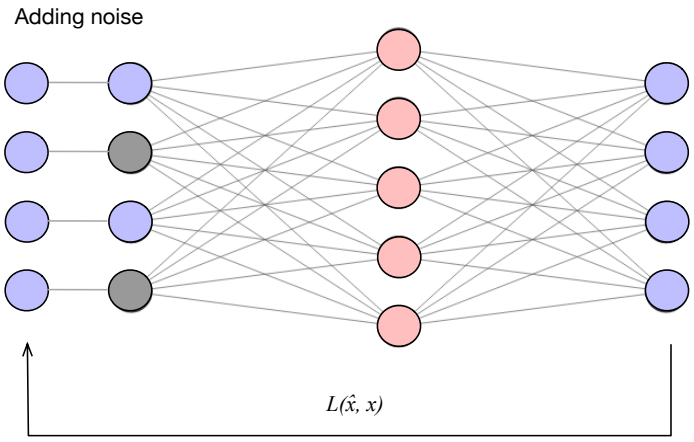


Figure 11-13: Denoising autoencoder. In each iteration the input layer with noise will be added to the network. Afterward, the cost function is compares output layer to original input layer, to tune weights and biases for the next iteration.

neurons in each pass randomly should change (or turn off) by the denoising function. Since in every pass a random number of neurons are changed, this type of autoencoder is called stochastic autoencoder.

Contractive Autoencoder

A contractive autoencoder [Rifai '11] is an overcomplete Autoencoder (usually but not always) that motivates the model to provide similar encoded data for similar input data. In simple words, similar input data should result in similar output data (i.e., the result of encoding).

It operates based on adding a penalty into a cost function. It applies a regularization penalty to the activation function (not on weights but only on activation function results).

The penalty tries to ensure that small changes in the input do not change the output (encoding result) significantly, and this small change maintains a very similar change in the encoded state. In other words, *with respect to the variation in input data, the variation of activation function in neurons of hidden layers should be small*. For example, if two input data have small differences and they are fairly similar, the activation values of the hidden layer neurons for these two input data should be small as well.

Some scientists describe a contractive autoencoder as extracting features that only reflect variations observed in the training set, and it is invariant to the other variations.

The cost function of the contractive autoencoder is written as: $L(x, \hat{x}) + \lambda(||J||_F)^2$, which λ is a hyperparameter given by the user and controls the strength of the regularizer, $||J||_F$ is a

minimization of the *Frobenius norm*[2] *on the Jacobian matrix*[3] *of hidden layer activations with respect to inputs*. You can check Chapter 8 to recall the Jacobian Matrix.

Assuming n presents the number of neurons in the hidden layer(s) and m presents the number of input neurons, $a_i(x)$ is the activation of hidden neuron i for the given x input.

We can write Frobenius norm on the Jacobian matrix of hidden layer activations with respect to inputs as follows:

$$||J||_F = \sqrt{\sum_{j=1}^{m}\sum_{i=1}^{n}\left(\frac{\partial a_i(x)}{\partial(x_j)}\right)^2} = \begin{bmatrix} \frac{\partial a_1(x)}{\partial x_1} & \frac{\partial a_1(x)}{\partial x_2} & \cdots & \frac{\partial a_1(x)}{\partial x_m} \\ \frac{\partial a_2(x)}{\partial x_1} & \frac{\partial a_2(x)}{\partial x_2} & \cdots & \frac{\partial a_2(x)}{\partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial a_n(x)}{\partial x_1} & \frac{\partial a_n(x)}{\partial x_2} & \cdots & \frac{\partial a_n(x)}{\partial x_m} \end{bmatrix}$$

In this Jacobian matrix, n is the number of activation functions, and m is the number of input neurons. Each row refers to a *gradient of one hidden neuron's output with respect to all input neurons.* For example, the first row gives a gradient of the first hidden neuron output with respect to all inputs, or the first column gives us all hidden neurons' activation gradients with respect to the first input neuron. The column refers to all hidden neurons' activation gradients with respect to one single input neuron.

If the partial derivative is zero, it means changing input (x_j) does not change the activation value of the hidden unit, i.e., $a_i(x)$.

We have understood that a contractive autoencoder makes the feature extraction function (i.e., encoder) resist small changes in the input. How does this happen? By using a penalty which is a Frobenius norm of the described Jacobian matrix.

Denoising autoencoder is simple to implement and does not need to compute the Jacobian of hidden layers. Contractive autoencoders are as simple as denoising autoencoders. However, they are more stable than denoising autoencoders because they introduce penalties to measure the sensitivity of hidden representation.

Stacked Autoencoder

In some cases the input dataset is complex, one single autoencoder can not remove unnecessary information and provide us only useful features. Besides, adding more hidden layers might cause underfitting or overfitting, and it is not easy to optimize weights in non-linear autoencoders with several hidden layers. This issue is handled by introducing a ***greedy layer-wise pretraining*** phase

---

[2] Frobenius norm (Euclidean norm or L_2 norm) of a matrix A with m number of rows and n number of columns is written as follows:

$$||A||_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{i,j}|^2}$$

[3] The Jacobian matrix captures the variation of weights of output vectors with respect to the input vector.

[Hinton '06, Bengio '07]. Autoencoders with a pretraining phase are referred to as stacked autoencoders. A stacked autoencoder has *shortsighted hidden layers*. Shortsighted here means every hidden layer is trained only based on its given input and not the previous layers' data. This is a greedy layer-wise pretraining approach because it is shortsighted to the previous layer only, and not all previous layers.

Why greedy layer-wise training resolves the challenges associated with having too many hidden layers?

When a neural network gets deep (it has many hidden layers) its hidden layers' weights close to the output layer are updated very frequently, but hidden layers' weights close to the input layer might not get updated at all. To resolve this issue, a greedy layer-wise pretraining builds the model by using the current hidden layer and not all previous layers.

For example, we have a small autoencoder that gets input X and the hidden layer of this autoencoder outputs the results as Z_1. The next autoencoder receives Z_1 an input, and its hidden layer produces Z_2. The third hidden layer receives Z_2 as input (not Z_1, only it previous layer) This process repeats until the last layer produces Z_n which is then fed into the last layer.

Figure 11-14 presents a stacked autoencoder. The last layer is usually a Softmax (or logistic regression) and it has a randomized initial weight. Randomized initialized weights here refer to

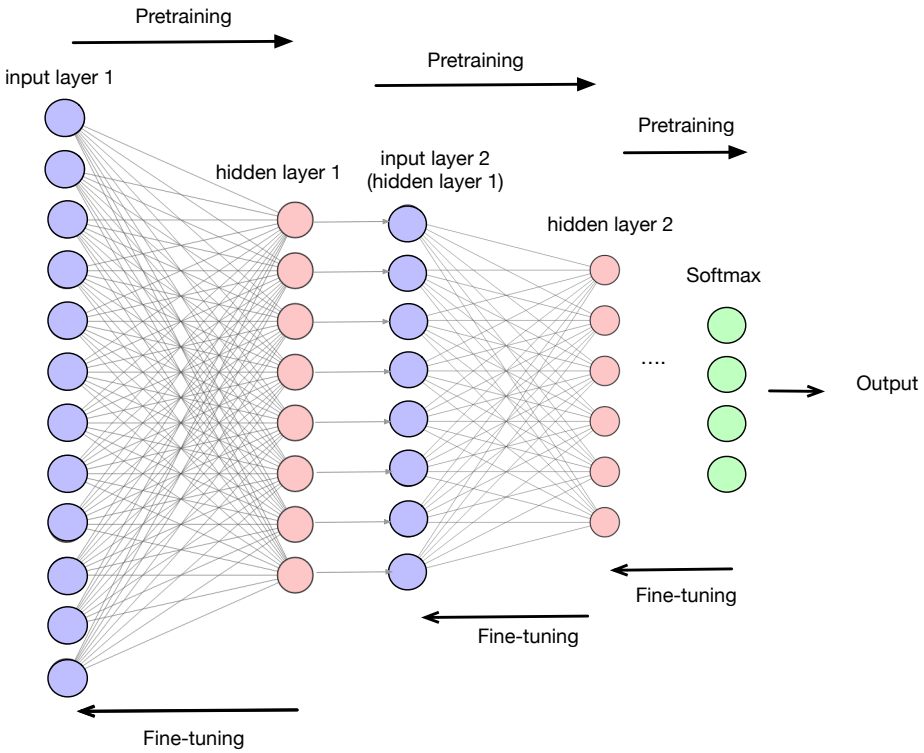


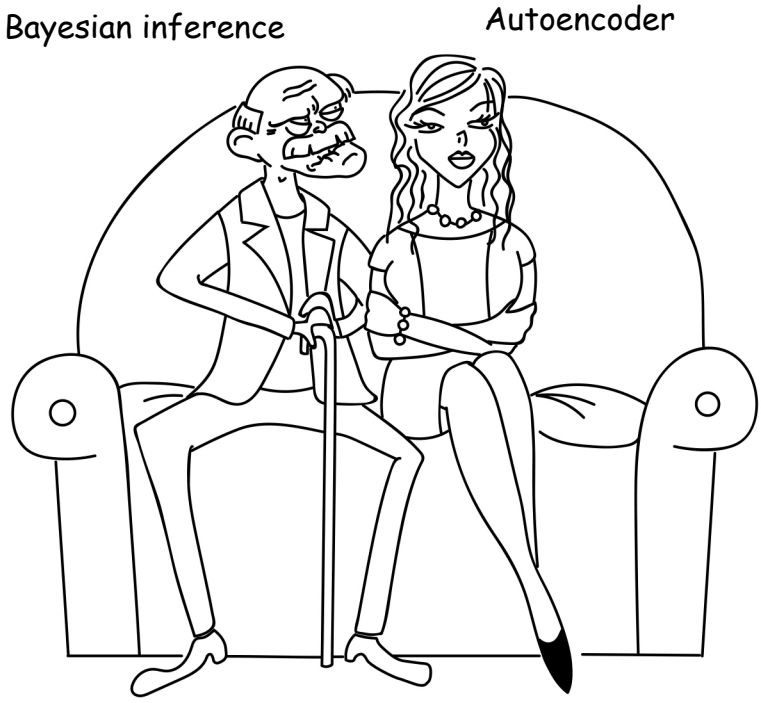
Figure 11-14: Stacked autoencoder with two autoencoders (they have only encoder and there is no decoder in stacked autoencoder), followed by a Logistic regression or a Softmax layer for classification.

the weights that must be corrected, and thus, we need to perform Backpropagation. By using this

approach, weights values are not very far away from optimal, and its backpropagation has a less complex path than other neural networks' backpropagation. This type of light backpropagation in stacked autoencoders is called *fine-tuning*. In other words, once all hidden layers are trained, the network starts fine-tuning. Note that fine-tuning is a supervised process because it uses labeled data to compute errors and perform backpropagation.

Variational Auto-Encoder (VAE)

All autoencoders we have described until now are regularized autoencoder. However, there is another variation of autoencoder, variational autoencoders (VAE) [Kingma '13], which is very popular. VAE is a combination of variational Bayesian inference and autoencoder. Bayesian inferences are one of the oldest machine learning approaches, in this book we only explained Naive Bayes in Chapter 9. Variational autoencoders are generative models that could be considered as an extension of the expectation maximization (EM) algorithm (check Chapter 3).



VAE is a latent variable model and latent variable models focus on understanding the structure of a dataset that has some hidden and unobserved variables. For example, while a document includes "foreign policy", "sanctions", and "allies in the region", we can guess its topic is talking about "international politics". This topic is not written in the document and because of that, it is called a latent variable. While dealing with a high dimensional and noisy dataset, it is common that the classification algorithm focuses on extracting latent features of the dataset and not all existing features.

The Magic of Generative Algorithms in Data Reconstruction

The VAE fits into the category of generative algorithms, and generative algorithms are very successful to reconstruct artificial data similar to the original data, due to their focus on the distribution of data. It means, that instead of working with the original dataset, they get the distributions of the original dataset and make a new dataset from the distribution of the original dataset. This leads to lots of success in generative AI applications.

How does such a thing happen? Assume we write an autoencoder to classify images of potato, carrot and onion. The one hot encoder can be used and each of them can be presented with a bit vector, potato: [0,0,1], carrot: [0,1,0] and onion [1,0,0]. If the algorithm trained on white onion, and not red onion it might not recognize the red onion, their shapes are the same (something can be used for inferences), but their colors are different (the color is something that has not been seen by the algorithm before). We can fix this problem by adding one bits and have one hot encoding as follows: potato: [0,0,0,1], carrot: [0,0,1,0], white onion [0,1,0,0] and red onion [1,0,0,0]. The more images we need to classify the larger will get the one hot encoding bit vector. Besides, if the algorithm encounters a red potato it has the same problem, despite its shape having similarities with the yellow potato. However, relying on the distribution of data, instead of exact data, can provide some flexibility in terms of learning new data. Because it can handle some not previously encountered (but similar) data, e.g. red potato. In this case, let's assume an onion has the distribution of [0.3,0.5,1,1.2, 1.4], a potato has a distribution of [2,1, 1.5,1.2, 0.9], and when an object came with the distribution of [0.5,0.7,1.1,1.4, 1.7] the algorithm can label it as onion because its shape of the distribution is more similar to the onion [0.3,0.5,1,1.2, 1.4] and methods such as KL-Divergence (check Chapter 3) can be used to perform this comparison.

How does VAE work?

The VAE extract input features, but it does not assign a single value to each of the features (e.g. the face has eyeglass: yes, the face has silicon in its lip: no), instead it presents them in a range of values (e.g. the face has eyeglass: 0.7 probability, the face has silicon in its lip: 0.2 probability) and each of extracted features is called a **latent attribute**. These latent attributes construct a **latent space**, in which features that are close to each other in the input space, stay close to each other in the latent space as well.

The output of the encoder is latent space. In particular, latent space is a representation of the input data in a multi-dimensional Gaussian distribution (each feature corresponds to one dimension) in hidden layers. Foster [Foster '19] provides a short and good description for VAE as follows: *"while using regularized Autoencoders a set of data points will map directly to one single data point in latent space, but VAE maps each data point into a multivariate Gaussian distribution around a point in the latent space."* Figure 11-15 visualizes the architecture of VAE.

Formally speaking, the output of the encoder in VAE is not a vector z (e.g. a vector of one hot encoding) instead it is $q(z|x)$ or posterior which includes distribution parameters (e.g. mean and standard deviation if we deal with Gaussian distributions) that present the PDF of z (Check Chapter 3 if you can't recall PDF). In other words, VAE turns the output of the encoder into

mean and *variance* (i.e. parameters used to describe Gaussian distribution). Previously described autoencoders map input into a fixed-size vector, but VAE maps the input into a distribution $p(z)$, which is parameterized by the distribution parameters. The decoder samples from the $p(z)$, and construct $p(x \mid z)$ or prior. Afterward, $p(x \mid z)$ will be fed into an activation function e.g. logit or Softmax function, to reconstruct the $\hat{x}$ which is similar to x (but not exactly equal). For example, if the decoder output is a binary variable (Bernoulli distribution), the Bernoulli distribution (or any other distribution) will be fed into a logistic function to convert it into categorical labels. Figure 11-15 presents the VAE structure in abstraction.

Another thing we need to remember is that variance values are always positive and to increase the variance variability the VAE is using the logarithm of the variance, i.e. $log(\sigma)$, instead of the variance. Because logarithms are numerically stable and can cover any number from $-\infty$ to $+\infty$.

For example, if the last layer of the encoder has two neurons, it will represent a two dimensional Gaussian distribution[4]. We know a Gaussian distribution is presented with mean and variance, and since we have two dimensions, we need two neurons for mean and four for standard deviation (check Chapter 3 to recall multivariate Gaussian distribution parameters).

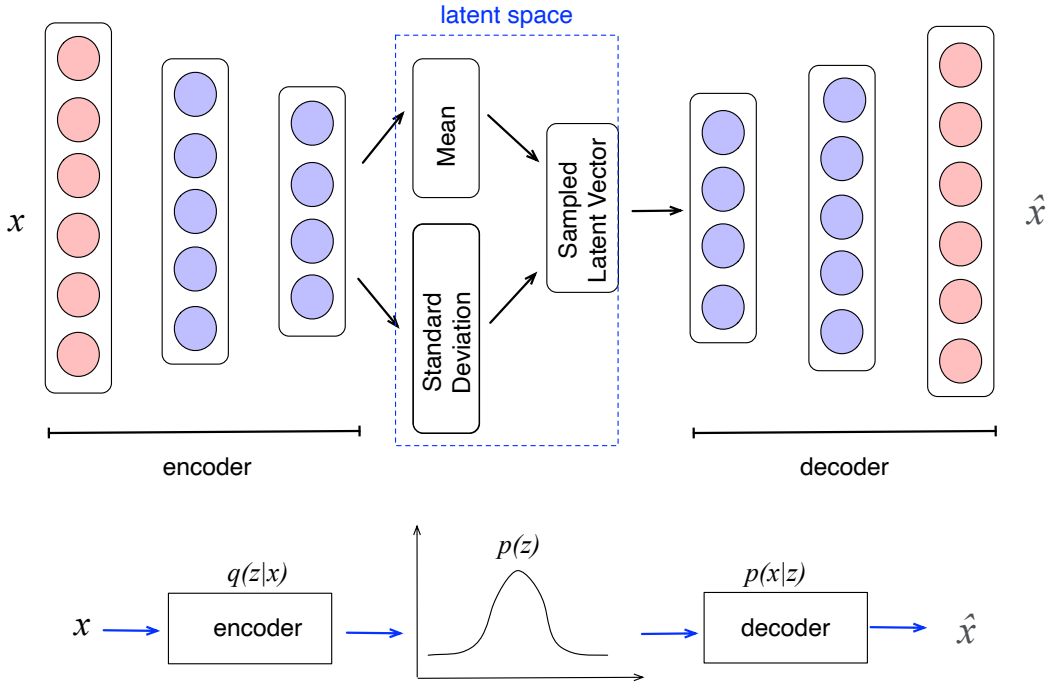


Figure 11-15: Schematic presentation of a Variational Autoencoder (VAE).

[4] VAE uses Gaussian distribution but there are few works that uses another distributions as well. Also authors of VAE paper explained that it is possible to use other distributions as well.

Assuming ϵ is a point sampled from a standard normal distribution, the encoding process converts the original input data into a latent space data (z) by using its mean vector (μ_x) and variance vector (σ_x) using the following equation: $z = \mu_x + \exp(log(\sigma_x)/2) \times \epsilon$

To summarize our discussion we can say VAE algorithm operates as follows:
1- It feeds the input data x into the encoder, and constructs $q(z|x)$, i.e. $x \rightarrow q(z|x)$.
2- It samples z from $q(z|x)$ to construct $p(z)$ in latent space, i.e. $q(z|x) \rightarrow p(z)$.
3- The decoder samples from $p(z)$ to construct $p(x|z)$, i.e. $p(z) \rightarrow p(x|z)$.
4- Then the cost function is used to perform the backward pass from step 1 to step 4. The algorithm updates weights and biases, and at the end of the iteration, step 5 will be executed.
5- It uses $p(x|z)$ to construct $\hat{x}$ (reconstruct x while some information from x is removed).

VAE Cost Function

The cost function of VAE seems a bit different than other cost functions. However, its nature is the same as other cost functions, its first part calculates the *reconstruction penalty (generative loss)* and the second part calculates the *latent loss (regularizer)*. VAE cost function is called *Evidence Lower Bound (ELBO)* and it is written as an *expected log-likelihood*, $\mathbb{E}[log]$, (because it is a probabilistic method and using expected log-likelihood as a cost function is common among probabilistic methods) minus KL-Divergence of the latent posterior probability distribution, i.e., $q(z|x)$ and prior probability distribution, i.e. $p(z)$.

$$ELBO = \mathbb{E}[log\ p(x|z)] - D_{KL}(q(z|x)||p(z))$$

The expected log-likelihood ($\mathbb{E}[log(P(x|z))]$) is a negative cross entropy, which is the data reconstruction loss. The expected log-likelihood is similar to other neural network cost functions. The second part, KL-divergence measures how close the distribution of the latent variables matches the decoder distribution (output). Here KL-divergence acts as a regularizer that penalizes the encoding process and ensures the sampled data point does not stay far away from the center of the distribution.

Similar to other neural network algorithms, it is needless to say that gradient descent and backpropagation are used to minimize the cost function.

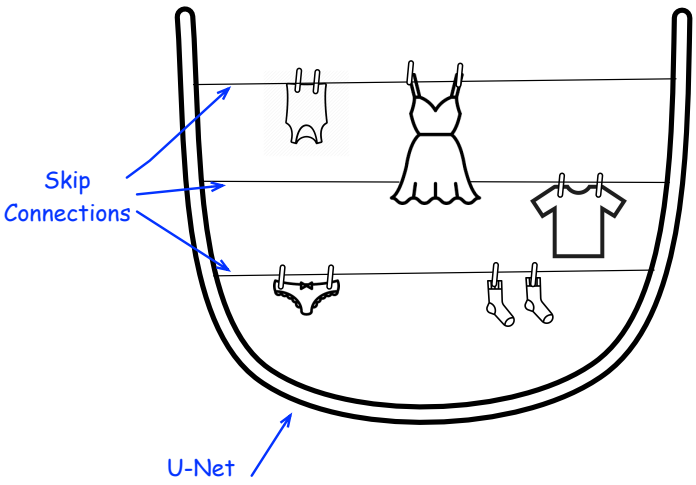
U-Net

One of the very popular segmentation algorithms is U-Net[5] which has an autoencoder architecture. We have briefly described two segmentation algorithms back in Chapter 6, MSER and Watershed algorithms, and more about segmentation algorithms will be explored in Chapter 12. Nevertheless, it is a big sin not to explain this legendary algorithm while discussing Autoencoders.

---

[5] Before you proceed with reading U-Net be sure that you recall terms we have used in describing CNN in Chapter 10, including max pooling, stride, down sampling and up sampling.

U-Net [Ronneberger '15] is a **semantic segmentation** algorithm designed for medical image segmentation, such as separating nucleus and cytoplasm in the microscopic image of a cell, extracting the lung tissue from fat and skin tissue in CT images, etc. At the time of writing this Chapter, it is state-of-the-art semantic segmentation for medical images, and algorithms that provide a new segmentation usually compare their approach with U-Net.



Authors of U-Net did not refer to it as Autoencoder, but it has a very similar architecture as autoencoders, and thus we explain it in this section. The original U-Net architecture is shown in Figure 11-16. It has a U-shape architecture and it is composed of **contractor/encoder** layers (on the left side) and **extractor/decoder** layers

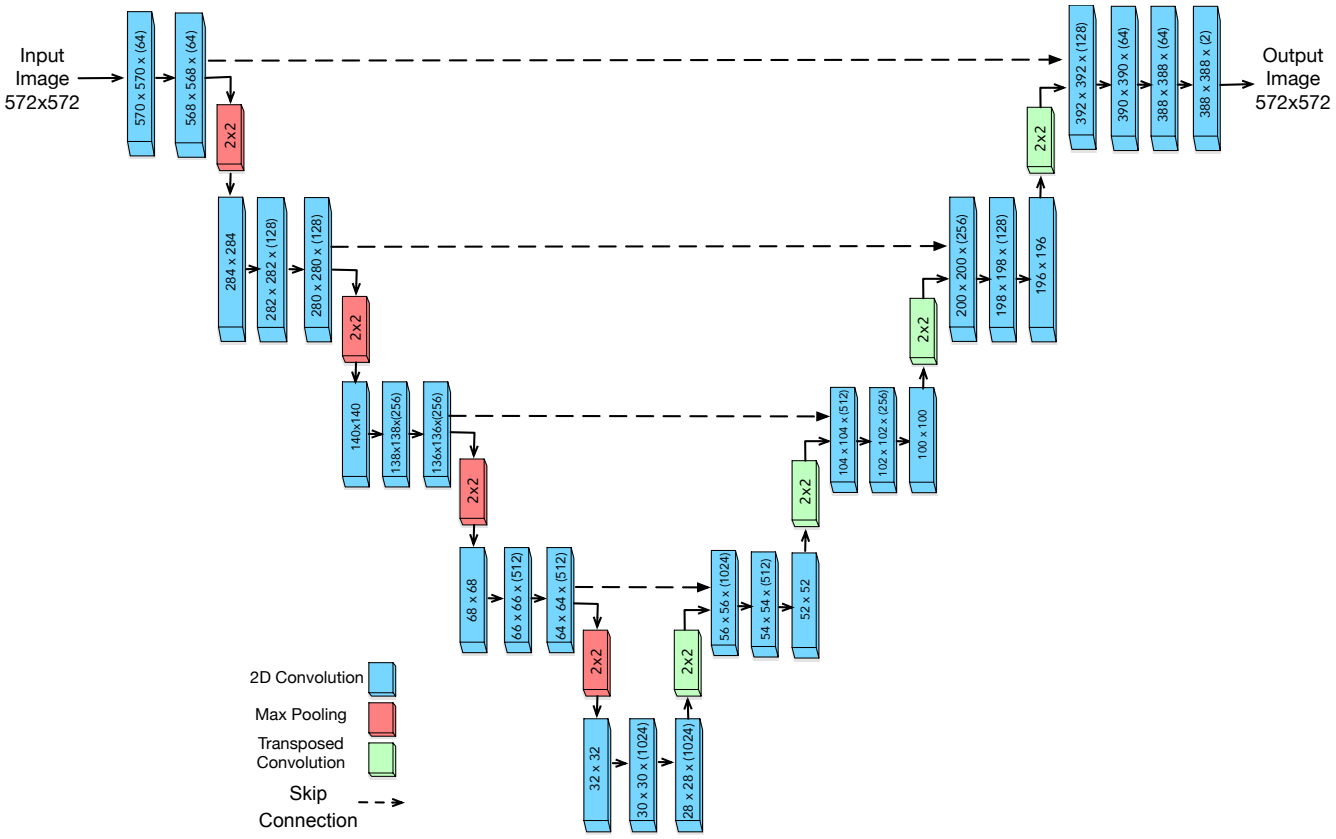


Figure 11-16: U-Net architecture. The left side presents contraction/encoder and the right side presents extraction/decoder part of the network. The dotted arrows (copy and crop) present skip connections. The number of each channel is written on the top of its layer.

(on the right side). Note that not all implementations of U-Net should follow the standard U-Net numbers (or other neural network numbers) usually, we can play with its parameters based on our application needs.

U-Net is moving a sliding window on the image and classifies every single pixel of the image, thus it is very expensive and slow, but due to its superior accuracy, it is widely used for image segmentation, and at the time of writing this text, it is the most popular one.

As you can see in Figure 11-16, its contraction path (left side of U) *down samples* the input image into a smaller size, and the extractor path (right side of U) *up samples* the contracted data that it has received from the last layer of contraction path. In each down sampling stage of the contraction path, the number of feature channels doubled and 2×2 max pooling with a stride of size 2 will be used. In summary, the contraction path is a set of convolutional and max pooling layers.

The first level of the contraction path applies 3×3 padding, which reduces the input size from 572×572 to 568×568. Then, the output of the convolution is down sampled for the next level. In the next level, again, the convolution will be applied and the result will be 280×280 with 128 channels (the number of channels is doubled). This process continues until four levels of down sampling.

Next, the extractor path starts the process of up sampling, which we can read its numbers from the right side of Figure 11-16. The extraction path is a set of convolutional layers and transposed convolutions (instead of max pooling) to upsample the data. At each level of the extraction path, the number of feature channels will be halved.

Both contraction path and extraction path use ReLU as an activation function. The contraction path learns *what* is in the image, but since it makes the image smaller it *loses the spatial information* of the image (it reduces *where* information of the image). The extraction path specifies *where* the information is located in the image, which means it handles the *localization of segments* in the image.

There is something interesting in U-Net architecture, which makes it different than other Autoencoders. Between the contraction path and extraction path, there are **skip connections**, marked with dotted lines in Figure 11-16 and Figure 11-17. Skip connections are responsible for cropping the image (by using padding) in the contraction path to match the extraction path image at the same level. In short, skip connections *concatenates the output of (feature maps) of contraction layers with the output of transposed convolution layers*. Using skip connection, skip layers of non-linear processing, and enables the network to mitigate the risk of vanishing gradients. Also, there is another advantage of using skip connections, it mitigates the **degradation problem**. The degradation problem refers to a phenomenon in a deep neural network in which increasing the depth of a network cause decrease in the accuracy of train and test.

Using skip connection is a good hack to avoid both issues and try to keep in mind while designing your neural network architecture. You can also employ the use of skip connection for

your own neural network, especially while working with image data and we need encoding and decoding.

We are done with the explanation of U-Net, but if you think that a U-Net architecture does not need to have a U-shape and it could be designed with a simple autoencoder shape, we do not disagree with you. However, the authors of the paper prefer to design it something like Figure 11-16. A more abstract shape of U-Net is designed by the author of Pix2Pix [Isola '17], is shown in Figure 11-17 and it is easy to memorize.

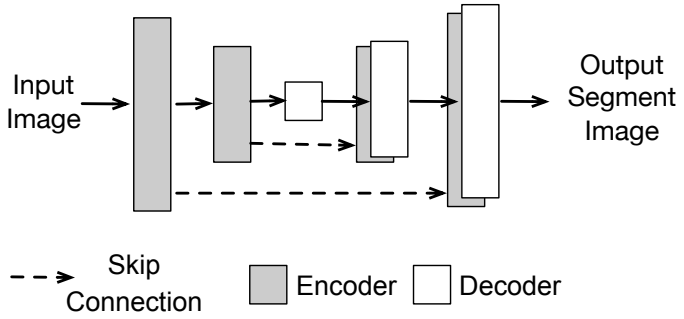


Figure 11-17: A simplified shape of UNet architecture.

NOTE:

* Autoencoders are unsupervised learning, but they are self-supervised learning, we can also improve the accuracy of the algorithm by introducing a small amount of labeled data.

* It is not mandatory but most of the time Autoencoder architectures are symmetric. Since they use a non-linear activation function such as a hyperbolic tangent function, the symmetry between input and output data will be never absolute.

* Aggarwal [Aggarwal '18] recommended transferring non-sparse data into sparse data while describing sparse Autoencoder because it provides a more flexible representation of data. The author of this book is a developer who struggles to push machine learning algorithms into battery-powered devices, a.k.a on-device machine learning, and always suffers from a lack of resources. Therefore, we do not agree with his recommendation. Nevertheless, it makes sense to share this opinion with the reader. If you are hired by a rich corporation that does not have limitations on its resources, you can use his recommendation for designing your model.

* Despite autoencoder compressing features, if we use linear activation functions, they have no additional superiority over traditional image compression algorithms. Therefore, they are not usually used for image compression and they do not act well as well.

* Both RBM and Autoencoder can **share weights**. Instead of using another set of weights at the output layer, we just use the transpose of weights from the input layer. Therefore, if we have W input layer weights, we can have W^T output layer weights. In other words, the weights of the decoder are the transposed weights of the encoder. This approach can operate as a regularizer, which could prevent making a linear transformation (like PCA), while we need a non-linear transformation. Therefore, it prevents the autoencoder to acts as an identity function. Besides, it enforces the model to have fewer parameters. Because of reducing the number of parameters and the likelihood of overfitting will be reduced.

* An Autoencoder that is composed of several hidden layers is called a **deep autoencoder**. Deep Autoencoders are useful for information retrieval tasks, especially for image search and matching. Due to the universal approximation theorem, having some additional hidden layers can guarantee that basically any function could be represented with the neural network, and this brings promising functionalities to the deep autoencoders.

* U-Net does not use any fully connected layer, it uses only the result of each convolutional layer, you can check the original paper for a justification of their approach and more details about the U-Net architecture [Ronneberger ’15].

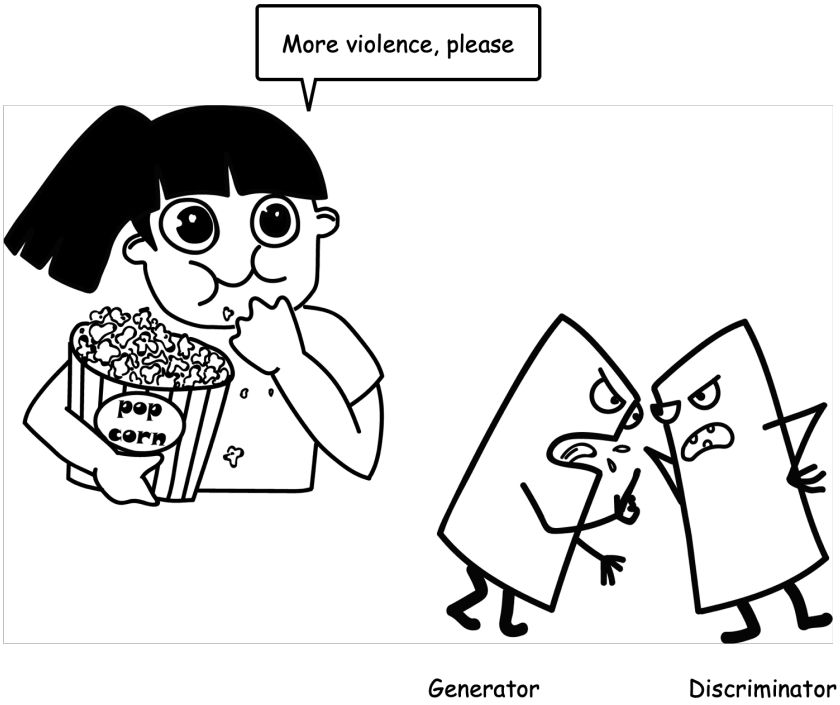
Generative Adversarial Network

One of the attractive approaches used for self-supervised learning is Generative Adversarial Network (GAN). In 2010, Olli Niemitalo wrote a concept paper [Niemitalo '11], without any implementation and introduced the concept of an adversarial network, which is known as Conditional GAN nowadays. Later in 2013, a similar model is proposed for animal behavior modeling by Li et al. [Li '13]. Afterward, Goodfellow et al. in 2014 [Goodfellow '14] introduced GAN and popularized it among the data science community.

Figure 11-18: Fake image generated by NVIDA open source tool, StyleGAN. Image source: https://en.wikipedia.org/wiki/Generative_adversarial_network

GAN employs generative models to construct a fake (to be polite, we can say synthetic) data object similar to the actual data object. This is scary (for its deep fake applications) but an attractive approach. They have many applications, including generating artistic images and music, image modification (e.g., converting low-resolution images to high-resolution images), photo-realistic image construction, face image manipulation (e.g., aging the face or making it younger), image translation (e.g., converting day taken picture into night taken picture), etc.

GAN is a neural network architecture that is composed of two neural networks. One network is called **Generator,** and the other one is called **Discriminator**. The Generator acts as a "counterfeiter" which tries to generate fake data, by using the original observed data. The Generator network is usually composed of a series of dense layers followed by transposed convolutional layers. The Discriminator acts as a "detective" and tries to *identify the fake data from the original data*. This process iteratively



continues as a competition between Generator and Discriminator. In each iteration, the fake data gets more realistic because Generator learns to improve its model. Still, the Discriminator also learns to recognize the fake data from the previous iteration, and it improves its recognition capability as well. Usually, the Discriminator is a convolutional neural network to classify the generated image.

The Generator starts from random noise and improves itself in each iteration. We could say the Generator has a similar role to the encoder in VAE, i.e., *converting the input into a latent space*. The result of a GAN is fake data that is similar to the original data, like audio or images that even we humans can not recognize the differences from the original data as well. For example, Figure 11-18 presents a human face image by using StyleGAN [Karras '19]. The lady in this figure does not exist in reality, but you can see her face is very realistic that we can not recognize it is fake.

Both Generator and Discriminator networks compete with each other, and in every round, they get stronger. The **Generator** is improving its fake data construction, and the Discriminator encounters more fake data and improves its discrimination capability. These two networks are playing a min-max game based on a particular objective function, which is cross entropy. Figure 11-19 a. presents the architecture of GAN. As it has been described, we have two neural networks, Generator and Discriminator.

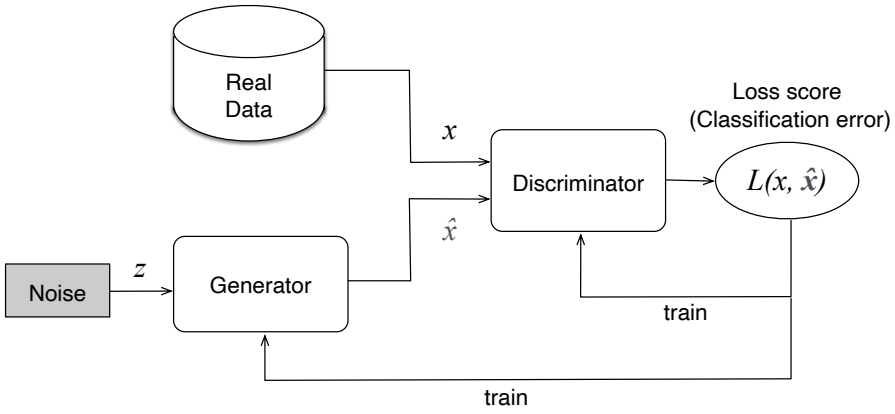


Figure 11-19: GAN architecture.

"Real Data" (x) is the dataset we have, and we intend to use it to construct a fake version of it ($\hat{x}$), which is similar to the "Real Data". The Generator performs the task of fake data construction. It starts by getting a vector of random "noise" (z) and then refines it in each iteration with the Backpropagation algorithm. Therefore, after several iterations, the random noise will be converted to data that has a distribution similar to the "Real Data".

The "Discriminator" gets "Real Data" (x) and "Generator" output ($\hat{x}$) as input to determine if the data is real or not, with a probability score. In other words, the Discriminator (D) is a binary classifier that specifies whether the output is fake or real. Considering 1 as a label for correct classification and 0 as incorrect classification, the D output (which is a probability) can be interpreted as follows:
True Positive: $D(x) \approx 1$

False Negative: $D(x) \approx 0$
False Positive: $D(\hat{x}) \approx 1$
True Negative: $D(\hat{x}) \approx 0$

The result of the Discriminator and Generator is classification error and backpropagation is used to improve both Generator and Discriminator weights and biases.

In particular, the Backpropagation is used by Discriminator to improve the *accuracy of identifying real versus fake data*. The Backpropagation is used by Generator to increase the *probability of the Discriminator misclassifying fake data ($\hat{x}$) as real data (x)*. After the training phase, the Discriminator will be discarded and the user will use the Generator model to construct new fake data.

Training GAN

Generator and Discriminator are trained alternatively, and their training process is different. Discriminator training can be summarized in the following four steps:
1- Sample (random mini-batch) x from the"Real Data".
2- Take a random noise z, add it to x and construct the output of Generator (G), i.e. $G(z) = \hat{x}$.
3- Then, the Discriminator classifies x and $\hat{x}$, and calculates their error (loss score), i.e. $L(x, \hat{x})$.
4- The Backpropagation algorithm improves the weights and biases of the Discriminator to reduce the loss score of the discriminator in the next iteration.
Generator training can be summarized in the following three steps:
1- Use a random noise z and feed it into the Generator network to construct fake data, i.e $G(z) = \hat{x}$.
2- The $\hat{x}$ will be classified by the Discriminator, i.e. $D(\hat{x})$, and it computes the loss score (classification error).
3- The loss score will be Backpropagated to the Generator network to update weights and biases, toward reducing the loss score of the generator.
The Generator weight and biases remain fixed while the algorithm is training the Discriminator, and also Discriminator's weight and biases remain fixed while the algorithm is training the Generator.

Now a question might arise: "When does the GAN iterative training stop?" The GAN stops when both Discriminator and Generator reach a stage where neither Generator nor Discriminator can not improve their accuracy, which means neither of these two networks can improve their loss score. This is referred to as Nash equilibrium or Zero-Sum game, in which no parties in the game can change or improve their state without changing the other party's parameters.

GAN stops when either (i) the Generator produces fake examples that are not distinguishable from real data, or (ii) the Discriminator guess starts to get random for the Generated fake data. In layman's terms, GAN stops when Discriminator will be defeated and cannot recognize fake from real data. Nevertheless, in real-world applications, it is very hard to get into Nash equilibrium and we should stop the algorithm after we get some satisfactory result.

The training of other neural networks is an optimization problem, but in GAN, every network can train itself and can not affect another network. Therefore, it is better to say its training is like a game, rather than optimization for problem solving.

GAN Cost Function

The GAN training is two player game. One player is the Generator (one player) that tries to fool the Discriminator by generating fake data that looks like real data. The other player is Discriminator (another player) that tries to distinguish whether a data is real or fake. The cost function of GAN is a Minimax game and it is written as a cross-entropy between real ($\mathbb{E}_x[log\ D(x)]$) and fake data ($\mathbb{E}_z[log(1 - D(G(z)))]$) distributions.

$$J(D, G) = \underset{G}{min}\ \underset{D}{max}\ \mathbb{E}_x[log\ D(x)] + \mathbb{E}_z[log(1 - D(G(z)))]$$

In this cost function:
$\mathbb{E}_x$ presents expectation of x, which is drawn from real data.
$D(x)$ is Discriminator output for real input x.
$\mathbb{E}_z$ is the expectation of fake data samples which comes from the Generator.
$G(z)$ or $\hat{x}$ is the Generator output for a given noise z.
$D(G(z))$ is the Discriminator output for generated fake data, i.e. $G(z)$.

In the cost function, the Generator objective is to minimize the loss score. It means we want $D(G(z))$ to be close to 1. On the other hand, we want $D(x)$ to get close to 0. From the Discriminator perspective, we want $D(x)$ to get close to 1 and $D(G(z))$ close to 0.

This cost function could be also re-written as a Binary Cross Entropy (BCE):

$$-\frac{1}{n}\sum_{i=1}^{n}(y_i\ log(p_i) + (1 - y_i)log(1 - p_i))$$

In summary, each of these two neural networks uses the same function, but their objectives are in contrast to each other.

GAN Challenges

Training GAN is not trivial and there are some major challenges associated with GAN training. Here we list some common challenges of training GAN [Foster '19].

Loss Oscillation: Sometimes the loss score for Generator and Discriminator could oscillate and not converge. There are some GAN models that mitigate this oscillation, but still, it is a common issue in training GAN. Oscillation in loss scores of Generator and Discriminator states that something in the GAN model is not correct and thus we should revise our GAN model.

Slow Convergence: Another issue that is common in GAN is the slow convergence, which does not make GAN training practical in real-world software applications because it is very resource intensive and slow to make the model converge. Since GAN training is slow it requires lots of GPU to train it, and this disables small or middle enterprises from being able to train a GAN from scratch, usually, big corporation trains a GAN and others are using the trained model in their applications.

Mode Collapse: Real-world data are usually multimodal (check chapter Chapter 3). For example, assume we are building a GAN to create a fake human conversational agent (chatbot)

and make it so good that humans mistake this chatbot with a real human. To implement this chatbot, a GAN is used to intimidate real humans (the chatbot's conversational text is fake data and the human conversational text is the real data.)

A human has a combination of emotions, including anger, joy, and so forth. Once a GAN samples from a distribution of Anger (called mode in the context of GAN) and it is able to fool the Discriminator, because the Discriminator recognizes being angry is a human emotion and is not a chatbot. Then the Generator continuously keeps the agent angry (sample from anger distribution), and the Discriminator fails to recognize that this is not a real human. The result is a chatbot that is always angry, and users can recognize it as fake. If you can not connect well with the described example, let's use another one. Assume we create a GAN to construct handwritten digits similar to MNIST, but the GAN fails to produce all numbers e.g. 3 is never produced, despite the model is converged. Usually, a low standard deviation among samples is a sign of mode collapse.

In summary, sometimes a Generator finds a few data points that can fool Discriminator and stick with those data points. Nevertheless, there are too few data points and they are not a representation of the real data.

Uninformative Loss: Theoretically a small loss score of a Generator should translate to a high quality of fake data. Nevertheless, the Generator is compared with Discriminator and sometimes a decrease or an increase in Generator loss is not correlated with the quality of the fake data they produce. This lack of correlation between loss score and the quality of generated data is known as uninformative loss and it is another known challenge in GAN.

There are lots of improvements in the GAN architecture to mitigate the described challenges. Later in this chapter, we explain a few of the popular GAN architectures.

Evaluating GAN Result

GANs are a subset of generative models and most generative models are derived from the Maximum Likelihood Estimation (MLE) algorithm (Check Chapter 4). In particular, we could consider GAN as a non-approximate maximum likelihood maximization problem. However, in practice, we can not use MLE, because it overgeneralizes and delivers data (fake data) that are very different from real data. This is due to the fact that we need to have an accurate estimate of the distributions and their likelihood. To have such an accurate estimation we require having an extremely large training set, and this is not realistic and infeasible. Instead, there are two common metrics to evaluate the result of GAN, **inception score (IS)** and **Fréchet inception distance (FID)**.

Inception score (IS): The IS [Salimans '16] has been introduced in 2016 and uses a pre-trained neural network model, i.e. Inception-v3 [Szegedy '16] for image classification. We will explain more about Inception-v3 in Chapter 12. IS tries to capture two properties of data, *quality (fidelity),* and *variety (diversity)*.

Quality: First, a pre-trained Inception-v3 model is used to label each generated image (in probabilities). Assuming y is the label for generated image x, the result gives a *conditional label distribution, i.e.,* $p(y|x)$ for *every generated image*.

Generated samples that are close to real ones, will have low entropy ($H(y|G(z))$ is low). In simple words, *the first step maps a generated image to its class probability*.

Variety: In the second step, IS calculation algorithm identifies the *variety* of generated samples, $G(z)$, by using a marginal probability distribution, which is the probability distribution of all generated images, i.e., $p(y)$, e.g. 10% potatoes, 30% cats, etc. To have a high variety of generated images, the integral of the marginal probability distribution ($\int p(y|x = G(z))dz$) should have high entropy, i.e., $H(y)$. In simple words, the Generator should synthesize as many different classes as possible.

Next, the KL-Divergence (Check Chapter 3) between these two probability distributions will be calculated, $KL(p(y|x) \,||\, p(y))$. In the last step, it exponentiates[6] the KL-Divergence score to make it easier for humans to read, $IS = exp[KL(p(y|x) \,||\, p(y))]$.

By adding a logarithm we can get rid of *exp* and re-write the KL-Divergence as the following equation.

$$log(IS) = H(y) - H(y|G(z))$$

We can see from this equation the IS score is high (higher IS is better), if we have high variety, i.e., $H(y)$, and low entropy for generated samples, i.e., $H(y|G(z))$.

If you are not happy with too much mathematics, let's switch to the elementary school explanation. The IS score calculates the differences between the entropy of the *variety of data*, $H(y)$, and the difference between generated samples and their labels should be low, $H(y|G(z))$, which presents the *quality*. A higher score of IS means a better result.

Fréchet Inception Distance (FID): IS does not take into account statistical characteristics (i.e., *mean* and *variance*) of both real and generated data. Therefore, if a GAN produces a very small number of samples, let's say it generates only three images from a 900,000 input dataset, but the quality is very high the IS score could get high, despite having very few instances of generated data. It is the limitation of IS score, and FID score [Heusel '17] can study the mean and variance of both real data and generated samples, using the inception network. This capability makes the FID score more attractive than using IS score.

Before getting into the details of FID, we should learn *Fréchet distance.* It is a similarity measurement method that measures the similarity between two curves in a discrete manner, taking into account the orders of data points in each curve. Back in Chapter 4 we have explained DTW, which we use for time series similarity measurement, Fréchet distance also belongs to the same category of similarity measurements.

---

6 Exponentiating a number x means converts it to e^x

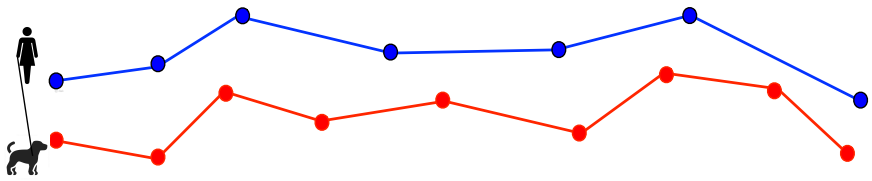


Figure 11-20: A toy example of two curves that have the same path but their trajectory is different.

To understand Fréchet distance, assume a person is traveling along a curve and walking with her dog from a source to a destination. The dog and the owner have different path preferences, but they both move from the same starting point to the same target, also getting back is not allowed, but their trajectory could be slightly different. Fréchet distance identifies the length of the shortest leash that is sufficient for traveling both curves, without getting back and only moving forward, while the owner and dog could have different speeds, see Figure 11-20.

In simple words, this distance specifies *how long should be the dog leash, so both owner and dog can travel from start to finish points on their separate path.* This distance is used for curve and surface matching. We can also use it to calculate similarities between distributions.

Now that we understand Fréchet distance, let's get back FID score, unlike IS which uses the output (labels) of Inception-v3 network, FID uses the *activations from the last pooling layer* (before the output layer) of the inception-v3 network. This layer has 2,048 activations, therefore, each data (image in this case) is modeled as a vector of 2,048 features. We can say that FID uses Inception-v3 network for *feature extraction*.

Assuming the real data has multidimensional Gaussian distribution of (μ and Σ), which μ presents the mean and Σ presents covariance matrix, and the generated data has the multidimensional Gaussian distribution of (μ_g and Σ_g), *Tr* is the trace operator [7], the FID distance between two data will be calculated by using the following equation:

$$FID = |\mu - \mu_g|^2 + Tr(\Sigma + \Sigma_g - 2(\Sigma\Sigma_g)^{1/2})$$

A lower FID score means better quality.

GAN Architectures

Min-Max GAN or vanilla GAN is very limited and has a slow convergence. At the time of writing this section in late 2021, probably one of the most attractive areas in computer science for new PhD students is GAN networks and their synthetic data generation capability. This leads to the introduction of too many GAN models, which are referred to as GAN zoo. Therefore, we limit this discussion to a few well-known GAN architectures.

---

[7] Tr(X) denotes trace of a square matrix X. It is the sum of elements on the main diagonal (from the upper left to the lower right) of matrix X.

Note that most of the GAN architectures we describe here are operating on image data, but we try to call it data, because these architectures might be applicable to other data sources as well, such as music or text.

Conditional GAN (CGAN)

The traditional GAN, Min-Max GAN, has no control over the output, or types of synthetic data being generated. This process can be easily improved by adding a label to both real data and synthetic data [Mirza '14]. For example, assume a GAN, which is generating pictures of handwritten numbers from the MNIST dataset. A label (y), which specifies the digit could be assigned to real data, and this label helps the Generator to produce more realistic synthetic data. This label will be incorporated in the cost function and therefore, CGAN cost function will be written as follows:

$$\underset{G}{min}\ \underset{D}{max}\ \mathbb{E}_x[log\ D(x|y)] + \mathbb{E}_z[log(1 - D(G(z|y)))]$$

In this function, y presents the label. We can see, in the red section of the above equation, that the CGAN incorporates conditional probability into the cost function. The only difference is GAN uses probability and CGAN uses conditional probability.

The Min-Max GAN cost is written as follows:

$$J(D,G) = \underset{G}{min}\ \underset{D}{max}\ \mathbb{E}_x[log\ D(x)] + \mathbb{E}_z[log(1 - D(G(z)))],$$

The CGAN cost function can be written based on vanilla GAN as follows:

$$J_c(D,G) = \underset{G}{min}\ \underset{D}{max}\ \mathbb{E}_x[log\ D(x,\hat{x})] + \mathbb{E}_z[log(1 - D(x,G(x,z)))]$$

To summarize, we can say GAN Generator learns a mapping from noise z to output image $\hat{x}$, i.e., $G : z \rightarrow \hat{x}$. Generator of CGAN learns a mapping from noise z and input x to output image $\hat{x}$, i.e., $G : z, x \rightarrow \hat{x}$.

Deep Convolutional GAN (DCGAN)

Usually, GAN models are unstable to train and could cause Generator to produce non-essential outputs, DCGAN mitigates this issue by using convolutional layers in Discriminator and transposed convolutional layers (check Chapter 10) in Generator.

Back in Chapter 10, we described that convolutional neural networks are very popular for image classification. DCGAN benefits from CNN as well. In particular, its Discriminator includes convolutional layers, which compress the data, and its Generator includes a transposed convolutional layer that decompresses the input image. In other words, the idea of DCGAN is to add transposed convolutional layers between input vector z and the output image generated by Generator and use a convolutional layer in the discriminator to classify the images.

As you can see in Figure 11-19, Generator starts with a vector of noise (in the original paper is set to a vector of size 100). It has four transposed convolutional layers, and each of them

increases the height and width, but decreases the channel (in the original paper the result was $64 \times 64 \times 3$), as it is shown in Figure 11-21. The Discriminator includes four convolutional layers, and each convolutional layer decreases the height and width but increases the depth (the last layer of original paper delivers the output with size $4 \times 4 \times 512$). The activation function of the last layer is Sigmoid, which specifies if the output is 'real' or 'fake'.

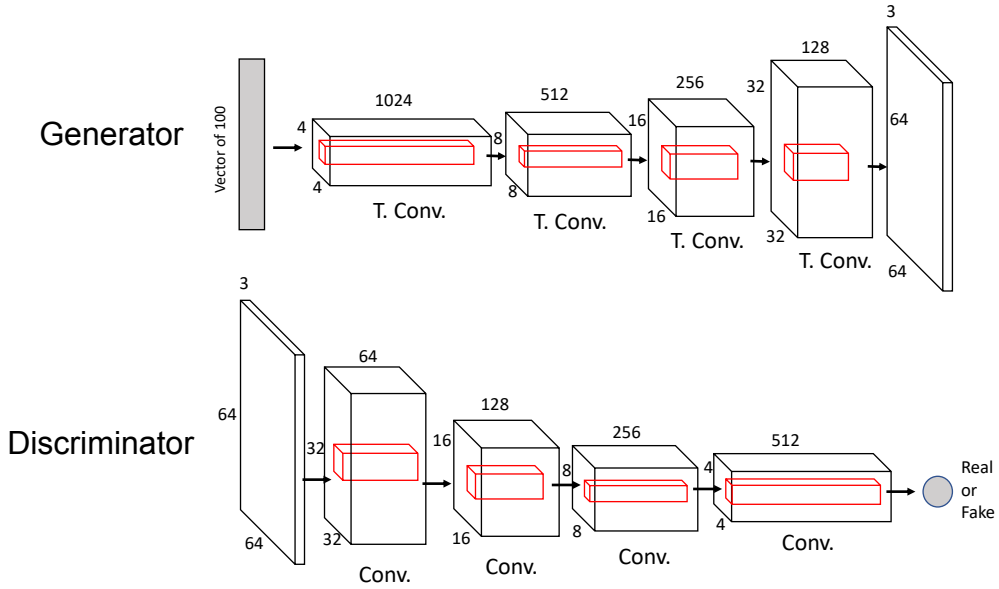


Figure 11-21: DCGAN discriminator and generator architecture. T Conv. stayed for transposed convolutional layer and Conv. Stayed for convolutional layer.

Both Generator and Discriminator neural networks have specific characteristics and some differences from commonly used CNN networks. For example, we have explained in Chapter 10, that usually after convolutional layers we have pooling layers, and after that, we have fully connected layers. DCGan has the following characteristic differences from Min-Max GAN:

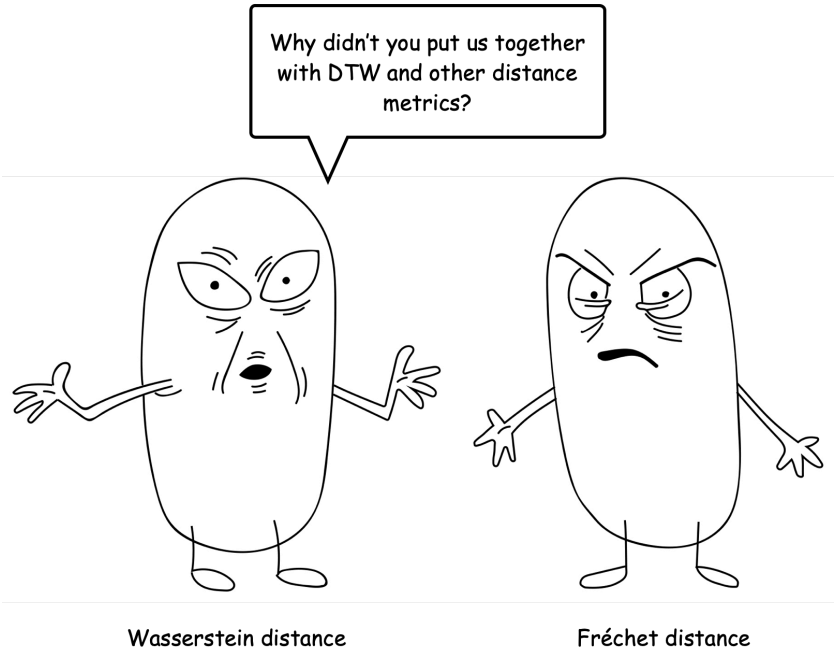
- It replaced any pooling layers with stridded convolutions in Discriminator, and fractional-stridded convolutions[8] in Generator.
- Both Discriminator and Generator use Batch normalization.
- They use ReLU activation in Generator for all layers, except for the output, which uses tanh (hyperbolic tangent). They use LeakyReLU activation in the Discriminator for all layers.
- The authors recommend initializing all weights by using a Gaussian distribution, with a mean of 0 and a standard deviation of 0.02.

Wasserstein GAN (WGAN)

WGAN introduces a different loss function for both Discriminator and Generator and it has a stable optimization process, which mitigates the "mode collapse" and "vanishing gradient"

[8] We skip explaining this convolution, you can check visualizations of Dumoulin and Visin [Dumoulin '16] for learning it.

problems. While using WGAN, we refer to Discriminator as **Critic** and not Discriminator. The reason is that the Discriminator output is 0 or 1, which discriminates between real and fake, but in WGAN it is a number because there is no "*log"* and the output is not a probability. Therefore, Sigmoid activation is not applied at the output of Critic (Discriminator). Besides, to understand WGAN, we should briefly describe two concepts first, Wasserstein distance and continuity condition.



Wasserstein Distance: Wasserstein (or Kantorovich–Rubinstein or Earth Moving) distance [Rüschendorf '85] distance is a method to compute distances between two distributions. For example, imagine we intend to compute the distance between Nigeria and Indonesia, not between their capitals, but the distances between two countries, which is comparing the entire ground size of one country to the entire ground size of another country. We could use different methods for this need, for example, we can calculate the central point in each country and calculate the distances between central points. Another approach is to calculate distances from all data points in each country to other ones and then average them. Wasserstein or Earth Moving (EM) distance is a popular method for this distance calculation.

Figure 11-22 presents two distributions P and Q. For sake of simplicity, we present the area under each distribution as a set of numbered rectangles. The EM distance between these two distributions will be calculated as *the minimum cost of transporting the mass converting distribution P to distribution Q*. Informally, we can say this distance metric is the minimum energy required to transfer one pile of soil in its shape to another pile, i.e. *amount of earth moved* $\times$ *the moving distance*. Figure 11-22 visualizes this figure in three steps for sake of readability and it does not cover all details. Step 1, shovels cube 1, 2, and 3 from the P distributions, moves them d size, and adds them to the Q distribution. The cost of this step is $3 \times d$. Step 2, moves P distributions cube 4 and cube 5 and adds them to the proper side in the Q distribution. The cost

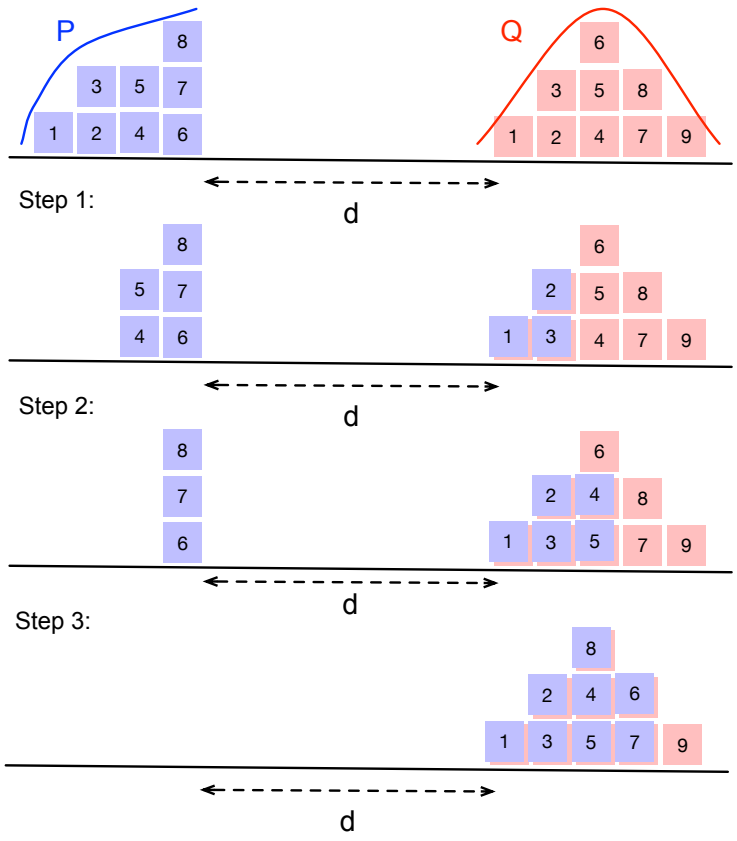


Figure 11-22: Two sample distributions, *P* and *Q* that stays in a *d* distance from each other. These steps try to simulate how Earth Moving distance calculates the minimum distance between two distributions.

of this step is $2 \times d$. Step 3, moves the cube 6,7, and 8 from P distribution and adds them into the Q distribution, but the cost is $2 \times d + (1 \times d - 1)$, the $1 \times d - 1$ is because 8 blue cube shifts one cube less than d. The final cost of this simplified EM distance is:

$$3 \times d + 2 \times d + 2 \times d + (d - 1).$$

Formally speaking, the EM distance between two distributions is the difference between their cumulative distribution (Check Chapter 3 for CDF), which is the area under the curve. $\Pi(P, Q)$ is a *transport plan* of P and Q, and it denotes the sum (joint) of both P and Q cumulative distribution, γ specifies the amount of mass (area under the curve) that is transported from x of P distribution to y of Q distribution. The EM distance between P and Q is written as follows[9]:

$$W(P, Q) = inf_{\gamma \in \Pi(P,Q)} \int ||x - y|| d(x, y)$$

The authors of the WGAN paper describe the Wasserstein distance as follows:

$$W(C, G) = inf_{\gamma \in \Pi(C,G)} \mathbb{E}_{(x,y)\sim\gamma} ||x - y||$$

[9] In this equation, the term "*inf*" refers to infimum, which presents the minimum in this context and $d(x, y)$ presents the distance between two points. The opposite of infimum is supremum, which is presented as *"sup"* and in this context means maximum.

$\mathbb{E}$ presents the excepted value, $(x, y) \sim \gamma$ means x and y sampled from γ (x for C and y for G).

Continuity Condition: The Critic network of WGAN must have the continuity condition. Continuity is very easy to understand, as you can see in Figure 11-23 in layman's terms a function is continuous if we can walk on it without lifting a pen from the paper. If we must jump and lift the writing pen from the paper to follow the path of function, this function is not continuous.

Mathematically speaking, Function $f(x)$ is called continuous at a point 'a' if three conditions are met.

(i) $f(a)$ must exist

(ii) $\lim_{x \to a} f(x)$ must exist.

(iii) $\lim_{x \to a} f(x) = f(a)$

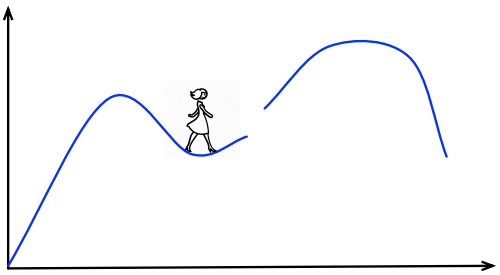


Figure 11-23: A simple example of non-continuous function

There is a specific form of continuity called **Lipschitz continuity**. A function is called L-Lipschitz continuous if there exists a constant $L \geq 0$ that for all x and y, we have $|f(x) - f(y)| \leq L|x - y|$. L here is called **Lipschitz constant**.

WGAN cost function: let's get back to the WGAN description and the use of Wasserstein distance. Now, a question arises and one might ask: "why not use more popular distribution similarity metrics such as KL-Divergence or JS-Divergence (check Chapter 3), and instead use Wasserstein distance?" Authors of WSGAN explained that *if two distributions do not have overlap (disjoint)*, such as the example proposed in Figure 11-22, the KL-Divergence result is infinity, and the JS-Divergence result is not converging, and thus both are unreliable. However, EM distance performs better than two KL-Divergence or JS-Divergence, its result will be θ, which is a number that is between 0 and 1. The mathematical explanation is fairly simple, you can check the original paper for more detail. However, you do not need to learn it in more detail, because at the time of writing this section, we are in Boston, it is spring search and good weather is too short here, we want to go jogging now.

Wasserstein is favored over KL-Divergence and JS-Divergence when distributions are disjoint, but computing *infimum of* $\gamma \in \Pi(P, Q)$ *is intractable in EM.* Therefore, WGAN authors propose a transformation using Kantorovich-Rubinstein duality[10] to the original EM distance, as follows:

$$W(C, G) = \frac{1}{K} \sup_{||f||_L \leq K} \mathbb{E}_{x \sim C}[f(x)] - \mathbb{E}_{x \sim G}[f(x)]$$

[10] If you like to understand the details of this transformation this blogpost is helpful: https://vincentherrmann.github.io/blog/wasserstein

In this equation, K is the Lipschitz constant, and the maximum of Wasserstein distance satisfies the Lipshitz continuity ($||f||_L \leq K$). The term $\sup_{||f||_L \leq K}$ read as all functions with Lipschitz constant smaller than K. $\mathbb{E}_{x \sim C}[f(x)]$ is the expected value of the Critic function f (it is a 1-Lipschitz function), with real data x, and $\mathbb{E}_{x \sim G}[f(x)]$[11] is the expected value of the Generator function f (it is a 1-Lipschitz function), with synthetic data, x.

Assuming the function f is a Lipschitz continuous, parameterized by w, the Critic network is used to learn w to find a proper f_w.

The described WGAN cost function can be configured to measure the distance between C and G, as follows:

$$W(C, G) = \max_{w \in W} \mathbb{E}_{x \sim C}[f_w(x)] - \mathbb{E}_{z \sim G}[f_w(g_\theta(z))]$$

The Critic wants to maximize this $W(C, G)$ and Generator wants to minimize it (make those expected values close to each other). Explaining more about the cost function will explode both your brain and ours, but please be patient, a small part needs to be explained as well.

Weight clipping: While updating the weights in every iteration of the Critic training phase, the algorithm should regulate the Lipschitz continuity of weight function. Therefore, they introduce a parameter $c = 0.01$, which clamps the weights into $[-c, c]$ range. This simple trick preserves the Lipschitz continuity of weight function during the training phase. In other words, after every gradient update on the Critic function, weights are enforced to be in the range of $[-c, c]$, and this guarantees its Lipschitz continuity.

The rest of the algorithm is the same as other GAN, it uses RMSProp as an optimizer, $\alpha = 0.0005$ and so forth, you can read from its paper and understand it. Hopefully, our explanation here makes it easy enough to understand its main characteristics.

WGAN with Gradient Penalty (WGAN-GP)

After WGAN has been introduced and provided superior accuracy among other GAN models, later an improved version of WGAN in 2017 has been introduced, WGAN with Gradient Penalty (WGAN-GP) [Gulrajani '17]. Although WGAN is much more stable than other GAN versions at its time, it failed to capture higher level data distributions, and thus it generates poor samples, due to its weight clipping. Instead of performing weigh clipping, it penalizes the norm of the gradient of the Critic with respect to its input.

WGAN-GP has three differences major differences from WGAN. (i) It does not use gradient clipping. (ii) It applies gradient penalty in the cost function to enforce Lipschitz continuity. (iii) WGAN-GP Critic does not have batch normalization, because batch normalization makes the gradient loss penalty less efficient.

---

[11] We will encouter this weired writing more in the rest of this chapter, try to keep in mind that $\mathbb{E}_{x \sim P_{data}(x)}[\dots]$ means expected value of the content inside braces

The gradient penalty is only applied on the Critic and not the Generator. Therefore, assuming the WGAN cost function is $\mathbb{E}_{x\sim C}[C(x)] - \mathbb{E}_{z\sim G}[C(G(z))]$, Critic for WGAN-GP is written as $W_{GP}(C,G) = \mathbb{E}_{x\sim C}[C(x)] - \mathbb{E}_{z\sim G}[C(G(z))] + \lambda \mathbb{E}_{x\sim P_x}[(||\nabla C(G(z))||_2 - 1)^2]$.

In this equation, λ is the gradient penalty coefficient and $\lambda = 10$, P_x is sampling *between* pairs of points sampled from the real data distribution and the generated data distribution. We could say P_x described as interpolated data instead of real/synthetic data.

There are other slight differences, such as the use of Adam instead of RMSProp, which are also not important to understand; you can check the detail from its paper as well.

Pix2Pix

Previous models of GAN we have explained are focused on generating images. One of the fascinating applications of GANs are the transformation of an image into another image (synthetic image). Pix2Pix [Isola '17] is a GAN algorithm that uses a CGAN to translate an input image into a different image. For example, convert a satellite image into an outline map.

Pix2Pix gets a label, similar to CGAN, in its Generator along with L_1 regularizer. It is similar to CGAN because it conditions an input image and generates an output image based on the conditioned input image. This makes it attractive for image translation. L_1 regularizer is used to penalize the output of the Generator if it is not close to the original input.

One popular application of Pix2Pix is to transfer sketches into photo realistic images. For example, you can see in Figure 11-24 that a dataset of Pokemon figures is used to train the algorithm. We draw a masterpiece of art and Pix2Pix uses the Pokemon trained dataset to make a realistic figure out of it. If you do not read this chapter properly and skip our mathematical explanations the monster of Figure 11-24 will hunt you in your dream and kisses your chick.

Figure 11-24: Using Pix2Pix trained on Pokemon image data to transfer an sketch we draw into photo realistic images. To construct this image, we use following link https://zaidalyafeai.github.io/pix2pix.

The Generator of Pix2Pix is based on a U-Net architecture, which operates similarly to Autoencoder. The encoder downsamples the input image to a low-resolution feature map, and the decoder upsamples it to the output image. The skip connections help preserve the spatial information from the input image to the output image. Using this architecture, which has skip layers, assists the output to have the same structure as the input. In summary, Pix2Pix uses a U-Net architecture for its cGAN generator

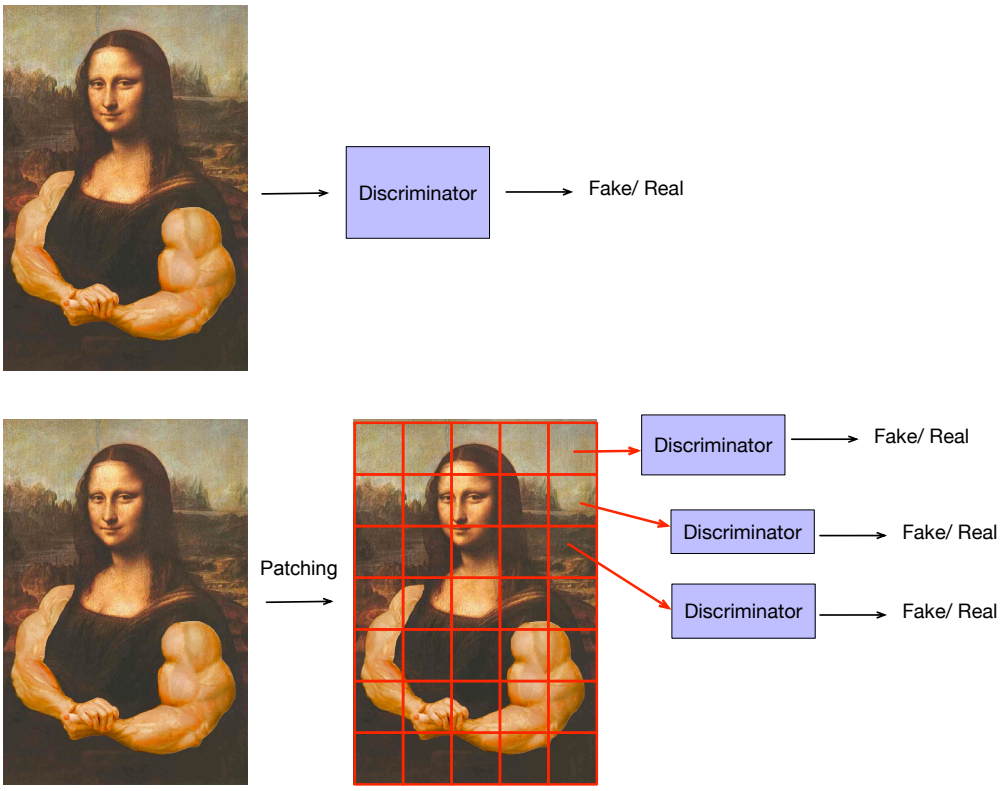


Figure 11-25: A vanilla GAN discriminator on top and PatchGAN discriminator in the bottom. The PatchGAN is feeding each patch into a discriminator and at the end averaging will be performed to decide if the image is fake or real. Image source: https://www.deviantart.com/califjenni3/art/Mona-Lisa-the-Bodybuilder-62547732

The Discriminator of Pix2Pix is a **PatchGan.** In short, a PatchGAN is patching an image into smaller patches (check Chapter 5 disjoint sliding windows), and instead of using a Discriminator for the entire image, a Discriminator is used for every single patch.

Figure 11-25, visualizes the differences between vanilla GAN and PatchGAN. The PatchGAN Discriminator tries to classify each patch (70×70) in an image as real or fake. It is very similar to CNN, and PatchGAN leverages the power of CNN, but instead of providing a scalar value at the end, it provides a binary result (fake, real) for each patch.

The PatchGAN applies convolution across the image, averaging all responses to provide the output, which presents if the image is real or fake.

Both generator and discriminator of Pix2Pix use Batch normalization after each convolutional layer and use ReLU activation function. You can read more about the details of their model in its original paper, but the information we provide here is enough to understand its important concepts.

CycleGAN

CycleGAN [Zhu '17] is another image translation GAN. It is an image-to-image translation algorithm applied to an unpaired set of images. Pix2Pix can not perform well for unpaired image translation. By unpaired here, we mean that CycleGAN assumes we have two datasets with different images, e.g., one for zebras and one for horses. It is obviously infeasible to have a paired dataset of two images (e.g., an image of a horse and a zebra, both with equal size, background, etc.). Nevertheless, with reasonable accuracy, CycleGAN is still able to transfer an image from one domain into another synthetic image from another domain.

We can see from Figure 11-26 that it can transform a zebra image into a horse image and transfer a landscape photo from winter into a photo in summer.

Figure 11-26: Three examples of image to image translation by CycleGAN.
Image credit [Zhu '17]

CycleGAN assumes the data translation is cycle consistent. It means if a translator G translates an image from domain X to an image in domain Y ($G : X \rightarrow Y$), another translator F can translate back the translated data into the original domain data, i.e., $F : Y \rightarrow X$. To clarify cycle consistency, the authors described that if an English sentence is translated to French, the translation of the French sentence back to English should show us the same original sentence in English.

CycleGAN Architecture: CycleGAN has two Discriminators, i.e., D_X and D_Y, two Generators, i.e., G and F, which are described as follows:

- G transforms images of type X to images of type Y.
- F transforms images of type Y to images of type X.
- D_X distinguishes differences between images X and translated images $\hat{X} = F(Y)$.
- D_Y distinguishes differences between images Y and translated images Y, which is $\hat{Y} = G(X)$.

To handle cycle consistency authors of CycleGAN introduce another cost function, which is called *cycle-consistency loss*. The cycle-consistency loss regularizes the model. In total CycleGAN has two cost functions, *cycle-consistency loss,* and *adversarial loss.*

Take a look at Figure 11-25 to understand the CycleGAN. On the top of this figure, we have two GAN networks and our objective is to translate data from domain X to data from domain Y (e.g. image of a cat into a lion). Also to hold the cycle consistency we need to have the image from

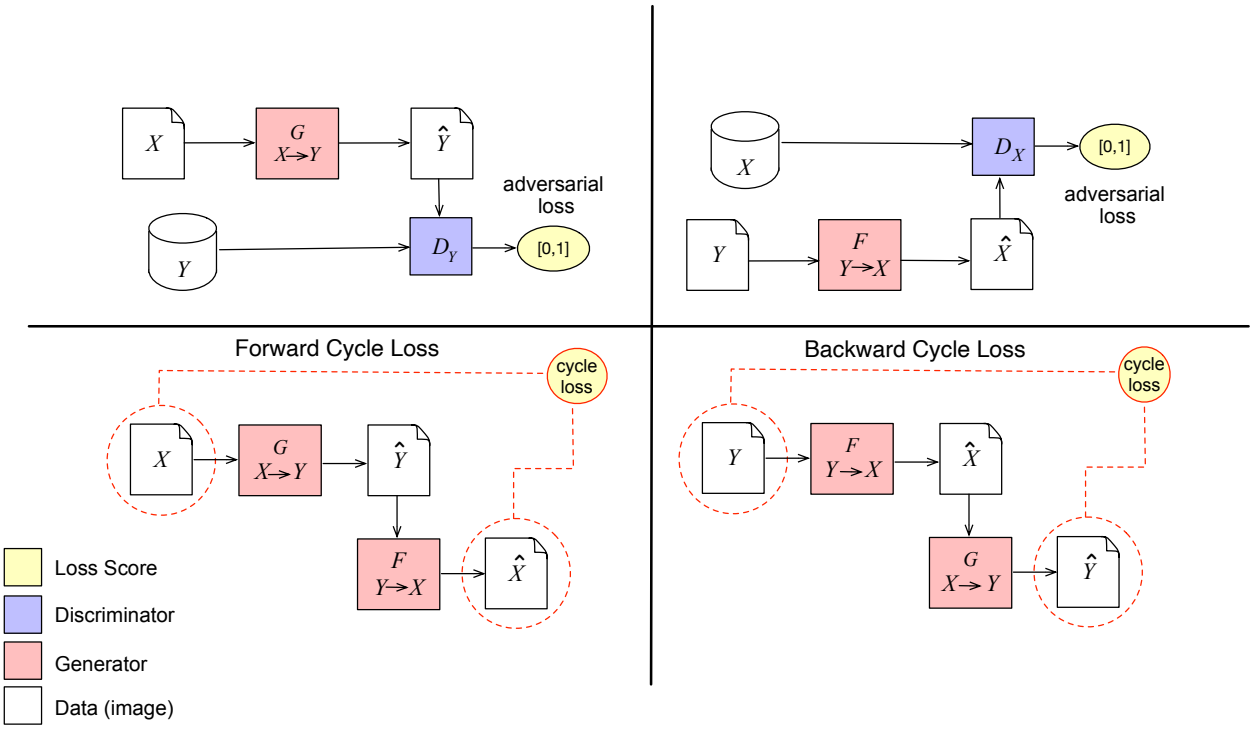


Figure 11-27: CycleGAN architecture is composed of two discriminators and two generators (We separate this figure into four subfigures for the sake of readability).
On the top, there are two GAN networks, with two generators, *F* and *G*. The one on the top left translates X to Y and the one on the top right translates Y to X. At the bottom, cycle consistency is presented with different figures. There is cycle consistency, I.e., forward and backward cycle loss. Therefore, the CycleGAN has four loss scores, two are discrimination losses of its GAN network and two are cycle losses.

domain Y translated back to domain X as well. (e.g. lion image back to a cat image). Therefore, two GAN networks are required and they are shown on the top of Figure 11-27. The one on the top left gets an image from domain X and uses Generator G to translate it to an image in domain Y, i.e. $\hat{Y}$. Then, the Discriminator D_Y uses images from domain Y to compare the $\hat{Y}$ images from domain Y and calculates the adversarial loss. The one on the top right gets an image from domain Y and uses Generator F to translate it to an image in domain X, i.e. $\hat{X}$. Then, the Discriminator D_X uses images from domain X to compare the $\hat{X}$ images from domain X and calculate the adversarial loss. As we have stated before there are two cycle loss functions as well. As is shown from the bottom parts of Figure 11-27. They compare the translated image with the original images before translation and calculate the cycle loss. For example, cat A is translated into a lion

B and the lion B is translated back into a cat A ($\hat{A}$). The cycle loss is measuring the differences between A and $\hat{A}$. Authors call it "Forward Cycle Loss", i.e. $x \rightarrow G(x) \rightarrow F(G(x)) = \hat{x} \sim x$. For each generated image in domain Y getting back to the original image is called Backward Cycle Loss, i.e. $y \rightarrow F(y) \rightarrow G(F(y)) = \hat{y} \sim y$.

CycleGAN Cost functions: The adversarial loss is similar to other GAN networks and for one GAN it is formalized as follows:

$$\min_{G} \max_{D_Y} J(G, D_Y, X, Y) = \min_{G} \max_{D_Y} \mathbb{E}_{y \sim P_{data(y)}}[log D_Y(y)] + \mathbb{E}_{x \sim P_{data(x)}}[log(1 - D_Y(G(x)))]$$

In this equation, G gets an image x and transforms it into the image y. D_Y tries to identify if the transformed data $G(x)$, which is shown with $\hat{y}$, and determines if $\hat{y}$ is a real data or not (by comparing with y).

For the second GAN it is the same equation, just changing the input, output and functions, i.e. $\min_{F} \max_{D_X} J(F, D_X, Y, X)$.

The cycle consistency cost will be written as sum of forward cycle cost and backward cycle loss.

$$J_{cyc}(G, F) = \mathbb{E}_{x \sim P_{data(x)}}[\|F(G(x)) - x\|_1] + \mathbb{E}_{y \sim P_{data(y)}}[\|G(F(y)) - y\|_1]$$

The first part of this cost function calculates the forward cycle loss, and the second part calculates the backward cycle loss and $\|...\|_1$ is a sign of L_1 norm. In particular, the cycle cost is either L_1 norm or summed absolute difference in pixel values between images.

By merging these two types of cost functions the overall CycleGAN cost function $J(G, F, D_X, D_Y)$ is written as follows:

$$J(G, F, D_X, D_Y) = J(G, D_Y, X, Y) + J(F, D_X, Y, X) + \lambda \, . \, J_{cyc}(G, F)$$

The parameter λ specifies the importance of the cycle consistency cost function. Authors of the paper set $\lambda = 10$ for their experiments.

Generator and Discriminator Networks: It makes sense to briefly go through the Discriminator and Generator architecture, you also can check several open source implementations, which could small differences from each other, for more details.

Each Generator has three sections, (i) an encoder, (ii) a transformer, and (iii) a decoder. The input image (with size $256 \times 256 \times 3$) is fed into the encoder and the encoder shrinks the input image but increases its depth. The encoder is composed of three convolution layers and its result (output size: $64 \times 64 \times 256$) will be fed into the transformer. The transformer is composed of a set of six Resnet block layers (no worries we will explain them in Chapter 12). The decoder get the image from the last layer of the transformer and reconstruct the image to its initial size by using three transpose convolutional layers. It means the decoder gets input of size $64 \times 64 \times 256$ and provides output of size $256 \times 256 \times 3$.

The CycleGAN discriminator is using the PatchGAN architecture [Isola '17], same as Pix2Pix. You can find more details about the details of architecture such as activation functions, stride size, etc. by checking its original paper or open source implementations of CycleGAN.

StyleGAN Models

At the time of rewriting this chapter, the StyleGAN3 (a.k.a Alias-Free GAN) [Karras '21] is among the most updated and advanced synthetic human face generators.

In this section, we explain StyleGAN [Karras '19], but we should neglect some details that you can further check in their papers or implementation. If you are feeling tired of reading about GAN architectures, take a look at Figure 11-28 and see how realistic are StyleGAN synthetic faces created. That person does not exist in reality [12] and StyleGAN generates it.

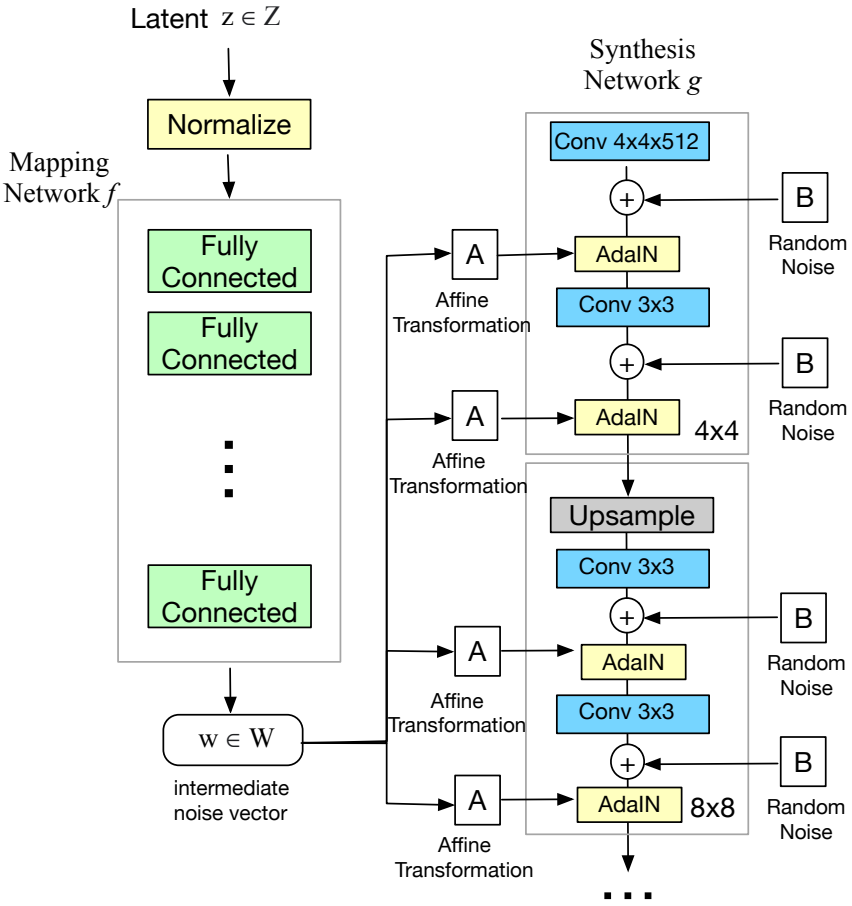


Figure 11-28: StyleGAN architecture.

The novelty of StyleGAN is in its Generator, its Discriminator is the same as Min-Max GAN. The StyleGAN Generator can control the fine-grained details (style) of the generated images. For example, in generated faces, the algorithm can separate the pose, identity, face, hair color, etc. StyleGAN also prevents mode collapse (e.g. generate only one particular face).

Besides, images could be mixed to construct new images. For example, taking parents' picture and guessing the face of their child in the future or control accessories on the face, such as adding or removing glass from the image.

StyleGAN generator has three architectural characteristics, which makes it different than vanilla GAN.

(i) Vanilla GAN Generator starts from a noise vector z and improves it until the Generator provides fake data that is hard for Discriminator to distinguish from real data. The StyleGAN Generator does not start with plain noise. It has a mapping network architecture (The sequence of Fully Connected layers which we can see on the left side of Figure 11-28), that gets a set of plain

[12] At the time of writing this book there is a web page that show StyleGAN2 sythentic images as well you can check it in the following address: https://thispersondoesnotexist.com

noise vectors $z \in Z$ (not just one noise vector more than one noise vector) and maps it to another set of vectors, $w \in W$. This noise vector (w) is called an **intermediate noise vector**. This network is known as mapping vector and apply a non-linear transformation on noise z. The purpose of this mapping network will be described shortly.

(ii) StyleGAN introduces an Affine Transformation (Four 'A' in Figure 11-28) and Adaptive Instance Normalization (AdaIN) [Huang '17], which incorporate **style (statistical characteristics of the image)** into the generated image as well (blue boxes in Figure 11-28presents AdaIN).

(iii) After the w is transferred into the Generator network, the algorithm adds another random noise (Gaussian noise) into the Generator as well. This means that StyleGAN uses two noises, unlike other GANs, which use one noise.

Noise Sources and AdaIN: In terms of noise, if we summarize Figure 11-28, we get something like Figure 11-29. In particular, There are two different types of noises added to the Generator, one is the intermediate noise vector, and the other one is the random noise.

The Generator got noise vector z with the size of 512 and passed it through eight fully connected (FC) layers (Multi Layer Perceptron), as you can see from the left side of Figure 11-28.

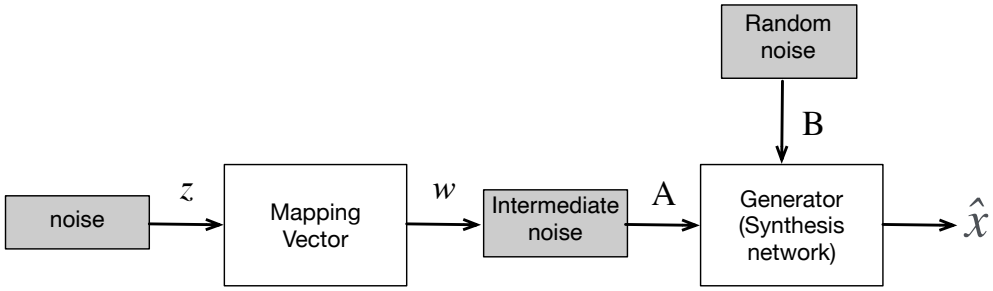


Figure 11-29: StyleGAN noises flow to the Generator

Why do the authors add eight additional Fully Connected layers? This process of mapping a noise vector into an intermediate noise vector provides a *disentangled representation of features*. Feature entanglement here means that the noise vector is not mapped to features we expect from the output (e.g. eyes, lips, nose). A noise vector has a Gaussian distribution, but features we need from output have a certain distribution around the actual object, e.g. the eyes distribution is not Gaussian and z noises do not entail those distributions. Therefore, these FC layers convert z noise vector distributions into distributions that are a bit closer to those output features distributions. In other words, authors state that w noises are less entangled and closer to the distributions of real image components (e.g. eyes, nose, lips). The result is w which is still 512 in its size.

AdaIN injects the intermediate noise into the layers of the Generator. AdaIN normalization is a type of Instance Normalization (check Chapter 10) that adds a type of style to the data. Style in this context refers to *statistical characteristics of the image components based on the incoming*

intermediate noise (A). We can say: *"Styles are extracted from w and then added to various points (As) into the Generator"*.

Assuming x_i is a feature map that is normalized separately, as it has been described in Chapter 10, Instance Normalization (IN) is written as follows:

$$IN(x_i) = \gamma \frac{x_i - \mu(x_i)}{\sigma(x_i)} + \beta$$

The intermediate noise *w* does not directly go into AdaIN, it goes into two fully connected layers and produces y_s (scale) and y_b (bias). In other words, this step converts *w* into a style *y*. AdaIN got style *y* as input, and substitutes γ and β of instance normalization with $y_{s,i}$ and $y_{b,i}$ (*i* subscript stands for the instance) which are scale and bias factors *coming from the intermediate noise* and it is written as follows:

$$AdaIN(x_i, y) = y_{s,i} \frac{x_i - \mu(x_i)}{\sigma(x_i)} + y_{b,i}$$

These two parameters (y_s, y_b) are parameters that construct the new style on the image and construct fake image. In Figure 11-28, y_s, y_b are presented as A. Therefore, we can say that AdaIN is responsible to transfer style information onto generated image from the intermediate noise vector *w*.

The second source of noise is a simple Gaussian distribution noise (stochastic noise). These noises are used to add more diversity and variation to the output.

Progressive Growing: StyleGAN is using progressive growing, in which the Generator gradually increases the fake data quality. The concept of progressive growing concept is proposed by Progressive GAN (ProGAN) architecture [Karras '17]. Progressive growing means at the beginning layers the model has coarser component of a face, such as nose, eyes, etc. and as it moves forward it adds more details such as hair color and eyebrow style.

In progressive growing the resolution grows slowly. In particular, the Generator starts by constructing a tiny image with 4×4 pixel size, and the Discriminator tries to analyze this image as well. Then, the resolution of the image increases by simple upsampling (after 4×4 will be 8×8) and in parallel applies a convolution on it, until it ends up in 1024×1024 pixel images. Other GAN architectures usually use transposed convolution for upsampling, but here a combination of nearest neighbor upsampling and convolution is used. On the right bottom side of each grey box in Figure 11-28 you see 4×4 and then 8×8, this represents the progressive growing.

This slow pace of evolvement in the network, causes the network to learn simple features (eyes, nose in the face) first and then progress toward more complex features (hairstyle, wrinkles on the face, etc.). This approach hinders the mode collapse problem as well. If you are interested to learn about more details of progressive growing you can check the ProGAN paper [Karras '17]. Very briefly, this process will be implemented with a combination of upsampling and convolutions.

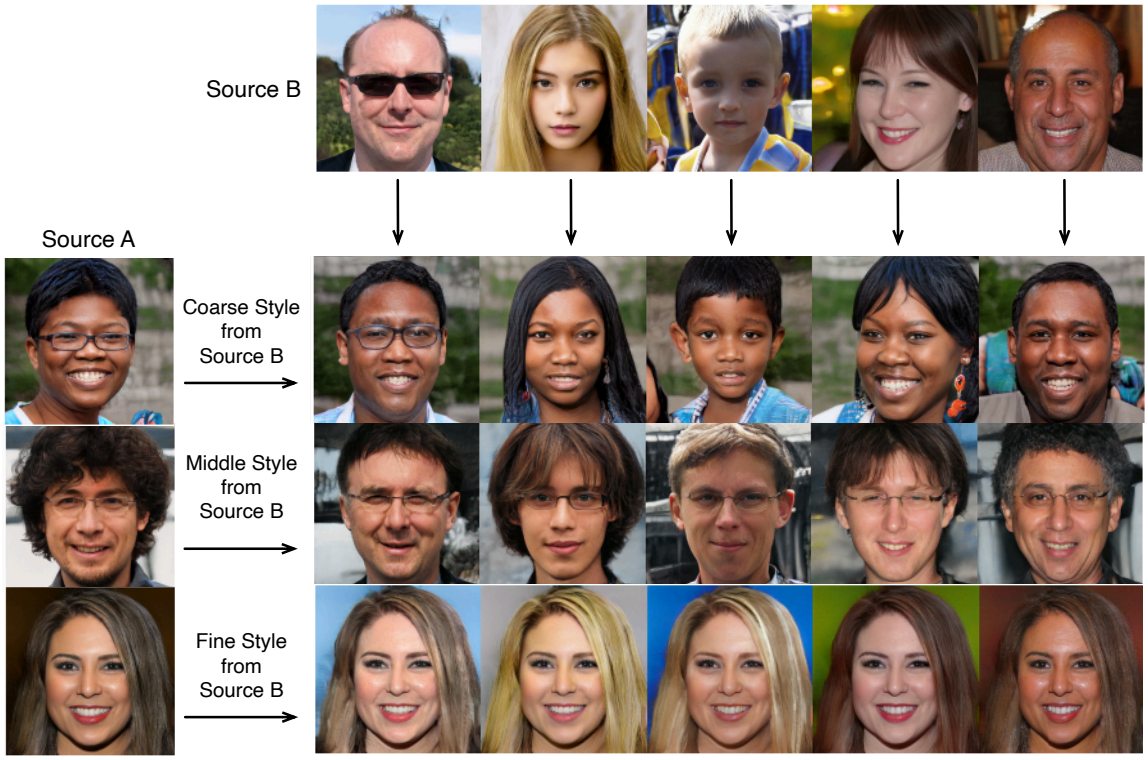


Figure 11-30: Mixing styles in StyleGAN: the impact of adding the intermediating noise vector into the Generator.

Style Mixing: One of the interesting characteristics that Style GAN is mixing images and generates again photo realistic images based on the mixture of images. This is an amazing feature and some recent mobile selfie apps adopt it as well. Figure 11-30 presents an example of this feature.

The process of style mixing occurs by mixing several intermediate noises (A noises in Figure 11-28) along with stochastic noises (B noises in Figure 11-28). As you can see from Figure 11-28, different A noises will be added to the Generator (Synthesis network g Figure 11-28) Some A noises control coarse style features (the one added at the first layers of the Generator), some middle style features and some fine style features (the one which is added into bottom layers of Generator).

Figure 11-30 presents two sources of images that are mixed together and construct new images, by mixing images from source A and source B together. As you can see, images in the first row got coarse styles from source B, images in the second row got middle styles from source A, and images in the third row got fine styles from source A. You can see in the bottom row, the women's face in source A face got the fine styles from source B, and the changes in her face are insignificant. Only her hair and skin colors have changed. This bottom line is in contrast to the first line that the amount of changes in the face is very significant.

We can control the degree of mixing styles by intermediate noise vector and this is the reason at the beginning of our explanation we mentioned StyleGAN can control the fine-grained style of the generated image. It was not possible with the previously described GAN architectures.

Figure 11-30 describes the impact of mixing styles (the impact of A noises). In some cases we do not intend to mix different styles, instead, we want to see different faces with one single image that is generated. This will be handled by stochastic noise (B noises), which is added before AdaIN. In other words, adding additional noise to the model adds additional variation to a single image. This variation can change very small details of the image, such as a wisp of hair. Figure 11-31 presents these features of StyleGAN as well, and you can see the impact of stochastic noise.

Figure 11-31: The impact of stochastic noise on the generated image. (Right) no stochastic noise. (Left) stochastic noises added in fine layers. Image credit: [Karras'19]

Finally, we are done with StyleGAN, but … wait … oh no … StyleGAN2, StyleGAN3 (Alias-Free GAN), remained.

StyleGAN Successors: At the time of writing this book, NVIDIA is the largest GPU producer and a team of researchers from NVIDIA is behind StyleGAN. Unlike academics who are continuously looking for grants, they are very rich to afford their GPU settings and improve StyleGAN. For example, Alias-Free GAN used 100, V100 NVIDIA GPU. If we can afford 100 NVIDIA GPUs we definitely do not write this book and enjoy a lavish life on a private island with tropical smoothies.

After the success of StyleGAN, the same group announce an even more accurate version of StyleGAN in 2020 and which is known as StyleGAN2 [Karras '20], next to that StyleGAN3 is introduced as Alias-Free GAN [Karras '21]. Here we briefly explain their differences with StyleGAN, but just to make us familiar with their differences and for more information, we should check their papers.

One known limitation of StyleGAN is water droplet effect on the generated image (see Figure 11-32). Another problem with StyleGAN is that some features are highly localized and their position in Generated images was pinned on a fixed position, such as the location of teeth or eyes staying in the same region in different generated images (take a look at Figure 11-30 the region of teeth in all images are the same).

Figure 11-32: Water droplet sign on generated images, because of AdaINST.

StyleGAN2 applied several changes to StyleGAN including substituting the use of AdaIN with "weight demodulation", changing the progressive growth, and adding two new regularizations, including "lazy regularization" and "path length regularization". These changes mitigate the limitations of StyleGAN.

After the authors introduced StyleGAN2, the same team from NVIDIA introduced StyelGAN3 (Alias-Free GAN) [Karras '21]. Although StyleGAN2 mitigates the issue of fixed positions of features in the image but does not resolve it completely. StyleGAN3 improves this limitation, its FID score is equal to StyleGAN2, but its internal image representation is better than StyleGAN2.

In general, coarse features (e.g. eyes, nose) mainly control the presence of finer features (e.g. hairstyle, wrinkles on the skin) in GAN architectures and they are fixed in their positions (referred to as "texture sticking" by authors). Authors identified that a network has a mean, and motivation to amplify this mean, which causes the generated images to include texture motifs. They find that the upsampling filters can not suppress aliasing, and aliasing causes the generation of texture motifs. By removing aliasing the issue of texture sticking has been resolved.

What remained to be explored? We are done with explaining some popular GAN architectures. You might be tired of learning many GANs, but by doing a search in arXiv.org, there are only more than 1,800 GAN models remaining that you can learn on your own[13].

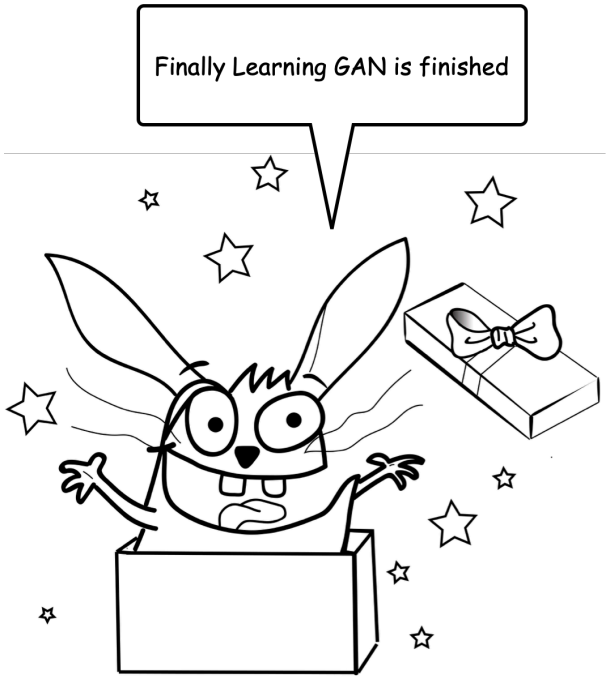
All jokes aside, if you are young and a student do not spend your valuable time on making another GAN architecture; there are lots of interesting things that remain to be discovered and learned in machine learning.

NOTE:

* There are claims that all generative models are derived from maximum likelihood algorithm.

* Training GAN is hard and there are numerous work on this topic. At the time of writing this chapter every year published in related conferences. If you like to join the army of GAN makers be sure to design a model that can handle all challenges we have listed for GAN (loss oscillation, uninformative loss, mode collapse and slow convergence).

* While working with other autoencoder and other described neural network algorithms we have single cost function that the optimizer need to improve. However, GAN has two cost functions that need to compete with each other.



[13] https://arxiv.org/search/?query=GAN&searchtype=title

* To have a stable GAN it is recommended to not mix fake and real data into a batch (stochastic gradient descent batch). Instead use separate batches and each batch only contain either fake or real data.

* Another approach to have an stable GAN is label smoothing, which means instead of 0 and 1 labels for fake and real data, the algorithm can use probability instead, and enable to algorithm to provides values slightly larger than one or slightly smaller than zero, i.e. label smoothing [Brownlee '19]. This has a regularization effect on the data as well.

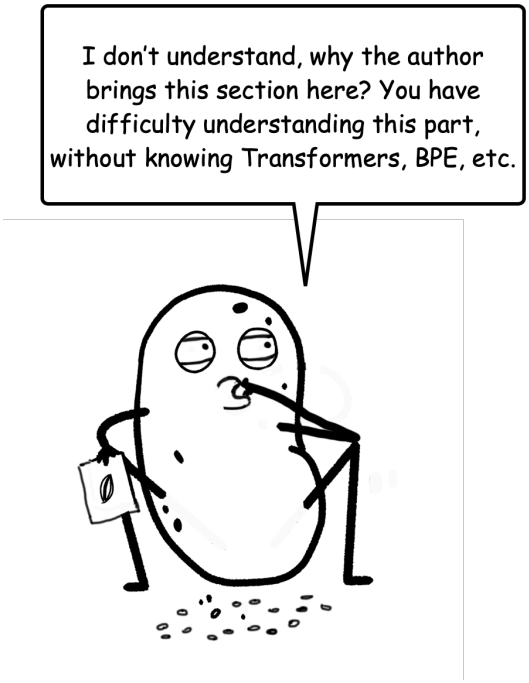
* One potential harm of image to image or video translations is fake video or fake audio generation, which is known as **deep fake** and it might be hard in the future to distinguish if a real person saying something in a video is real or it is deep fake. There are interesting works where you can see a face of an old painting is converted into a moving face and change his/her mimic, like Mona Lisa is smiling or changing her facial expressions [Zakharov '19]. Some scientists make a speculation that in the future it is possible to create an immortal version of a human in the digital world [Bell '10]. Another extreme example is that criminals can simulate a fake digital avatar which we can distinguish if it is fake or real, e.g. swapping a face of a popular person with a porn actor in a porn movie.

* Only rich corporations such as NVIDIA, OpenAI, Google and Facebook afford to build these large models. Even non-super rich universities fall behind the trend of making fancy GAN or other deep learning models, due to their dependency on GPUs.

Contrastive Representation Learning

There are different methods for preparing an object for comparison. None deep learning approaches use a similarity measurement (Chapter 4) and compare similarities between two objects. These approaches are used in the traditional image-matching algorithm, but they are not flexible. For example, a face of a person who rotates his head in two different images will be identified as two different images. A better approach is to convert data objects into a lower dimension, such as applying PCA (Chapter 7) and then comparing them, but still, they are inflexible.

Feature engineering in the context of deep learning is called *Representation Learning*. Representation learning is the process of mapping from raw input data to a more useful representation or feature space. To implement representation learning, a composition of multiple non-linear transformations is applied to the original data, and they transfer the input data into new data whose features are useful for the machine learning algorithm (it is called a downstream task as well) [Bengio '13]. A good representation of the data entails important features of the original data. For example, the word embedding methods we have learned in Chapter 6, are representation learning methods because they convert text data into vectors while maintaining the semantics of the text.



Using representation and using self-supervised learning to distinguish between similar and dissimilar data points leads to the introduction of **Contrastive Representation Learning (CLR)**, a.k.a **Contrastive Self-supervised Learning**. In short, CLR algorithms focus on learning by comparing [Le-Khac '20].

From a technical perspective, the objective of CLR is to build an embedding space (e.g. in a lower-dimensional space) where similar data points stay close together, and dissimilar data points stay far away from each other.

For example, we show an image of one rabbit to the algorithm and then we ask the algorithm to search a database of other animals and find other rabbits. The algorithm should recognize the correct object (image of rabbits) and contrast that object to other objects (images of other animals) in the dataset.

A question might arise; what is the difference between CLR and other supervised learning? Traditionally supervised learning learns from each single data point one at a time, but CLR algorithms learn by comparing different data points. In other words, CLR trains algorithms to classify similar and dissimilar objects.

We describe one type of neural network in this category, i.e., Siamese Neural Network, and leave the rest, such as SimCLR models [Chen '20 A, Chen '20 B] and MoCo models [He '20, Chen '20 C] to you to read them. Before we describe Siamese Network we need to be familiar with two popular loss functions, i.e. contrastive loss and triplet loss.

Contrastive Loss: We measure the contrastive loss [Chopra '05] between two objects X_1 and X_2 that each object describes a data point in embedding space, e.g., a vector. The generalized contrastive loss (L) between these two data objects is written as follows:

$$L(W,(Y,X_1,X_2)^i) = (1-Y)\frac{1}{2}(E_w)^2 + Y\frac{1}{2}\{max(0,\, m - E_w)\}^2$$

i specifies the index of the current data point. $Y = 1$ if two data points X_1 and X_2 are dissimilar, and $Y = 0$ if they are similar. m stayed for the margin of distance, it is a hyperparameter, and it defines the threshold (lower band) distance between data points of different classes.

E_W is the distance value (e.g., euclidean) between data points of two groups and assuming the G_W as a mapping function, it is computed as follows: $E_W(X_1, X_2) = ||\, G_W(X_1) - G_w(X_2)\,||_2$.

We can see that the right part of the equation penalizes the model for dissimilar data points having a distance $E_w < m$ (close to each other). It also penalizes similar data objects for being far from each other (stay outside m margin of distance). If $E_w \geq m$, then $m - E_w$ will be negative, and because of the *max*, the right part of the loss function will be zero.

Contrastive loss is usually used when we do not have the proper amount of labels, which is common because acquiring labels for raw data is expensive.

Triplet Loss: This loss function was first time proposed in the FaceNet [Schroff '15] model, which is a successful face recognition algorithm that can recognize the same person in different poses and different camera angles.

It compares the baseline or anchor sample x, against the positive sample, x^+ and a negative sample, x^-. By positive sample, we mean x and x^+ belongs to the same class (they have the same label), and by negative sample, we mean x^- and x belongs to different classes (their labels are different). The objective of this loss function is to encourage the network to minimize the distance between x and x^+, and, at the same time, maximize the distance between x and x^-.

It is formulated as follows.

$$L(x, x^+, x^-) = max(0, E(f(x) - f(x^+)) - E(f(x) - f(x^-)) + \epsilon)$$

In this equation $f(\,)$ is the embedding function that transforms the data into a different representation, ϵ is the margin between positive and negative data. This margin distance ensures ensure a distance exists between negative pairs and positive pairs. $E(\,)$ presents the euclidean distance (or L_2 norm), but other distance functions could be used as well. Some uses $||\ldots||_2$ to present this L_2 norm but to reduce your risk of getting a mathematical panic attack chance we used $E(\,)$. Anyway, you might find it more readable as follows;

$$L(x, x^+, x^-) = max(0, ||f(x) - f(x^+)||_2 - ||f(x) - f(x^-)||_2 + \epsilon)$$

Triplet loss is useful when it is important to set the negative sample properly, e.g., face recognition among many existing faces.

Siamese Network

The class of neural networks that they learn to *differentiate between two inputs* are called Siamese[14] Network. Unlike other neural networks, they do not learn to classify the data. Instead, they learn to measure the similarity between two inputs. Why not use traditional similarity measurements we have learned in this book, like correlation coefficient analysis (Chapter 3), or similarity metrics we have learned in Chapter 4 ? The answer is that described similarity metrics can not compare complex data structures together. For example, a traditional facial recognition approach can compare images from the same camera location toward the face (e.g., front view). It can not recognize the same person when her face image is taken from another direction (e.g., side view). These limitations led to the introduction of neural networks used specifically for comparison.

A siamese neural network (twin neural network) is composed of two or more identical neural networks. Here identical means the same number of layers, same neurons, same parameters, and even the same weights. After training, they can be used to recognize whether or not two input data are similar.

The objective of Siamese neural network is to have high similarity scores for similar inputs (e.g., images of the same person, signatures of the same person, the same food) and low similarity scores for different inputs.

A neural network that is common to be used inside a Siamese Network is a CNN. It means it has some layers of convolution followed by a few fully connected layers but no Softmax, because it is not a classification problem. Some suggest calling the output of each network the encoding of the input data.

Training Siamese Network: First, the network should be fed with positive (1) and negative (0) pairs. For example, *{(cat_img_1, cat_img_2, 1), (cat_img_3, cat_img_2, 1), (cat_img_1, dog_img_2, 0), (cat_img_2, squirrel_img_2, 0), …}.*

Figure 11-33 presents the architecture of training Siamese network, assuming we have an input set of pairs with labels, and we fed each pair into one network. For example, we feed x_1 to one network and x_2 to the other network. The result of each network will be a feature vector, let's say f(x_1) and f(x_2). Then, the difference between these two vectors will be calculated as shown in Figure 11-33, and they fed into a fully connected (FC) layer(s). The result of a FC layer(s) is given to a Sigmoid to identify if x_1 and x_2 are similar or not.

Now, the output of a Sigmoid function will be compared with the original label (similar:1, dissimilar:0) and the Cost function (Triplet loss, Contrastive loss, or Cross entropy) will be used to measure the loss score. Then the optimizer (e.g. SGD) back-propagates the loss score to change model weights and biases on the FC layer(s), and on both twin networks, toward reducing the loss score.

---

[14] Siamese twin is an extremely rare medical phenomenon in which two babies (twins) are conjoined and must be separated by surgery.

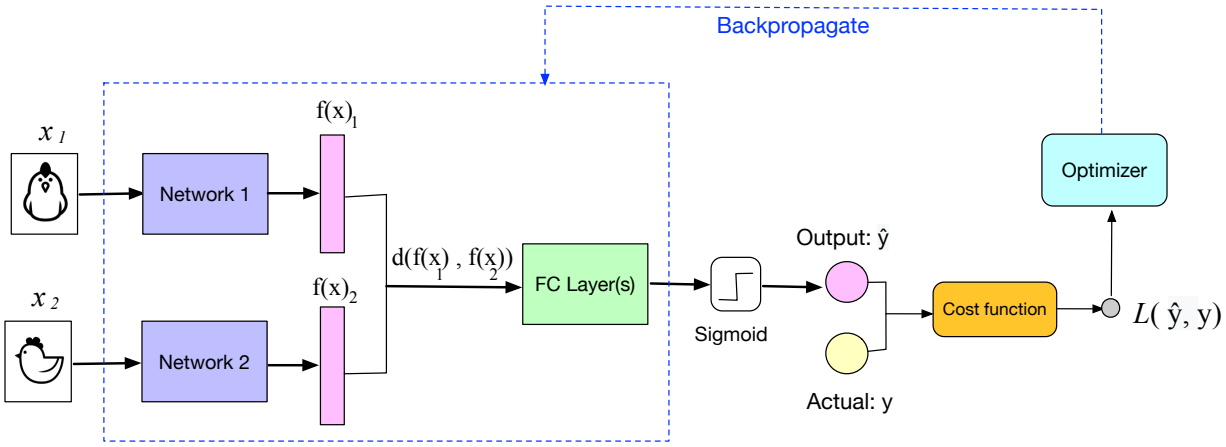


Figure 11-33: Training phase for Siamese network, in this example we show only one positive pair of example, which is a chicken. In practice, the network will be trained for many positive and negative pairs.

This process will be done for all pairs of images, both positive ones, and negative ones. The resulting model will be saved for the inference phase.

How to prepare data for training: we select a random input (anchor sample), and then we select other samples that have the same labels to build the positive samples. Next, we use the same anchor sample, and this time we select samples with different labels to construct the negative samples. All images in the training set are once considered anchor images, and this process will be repeated for them.

Testing phase of Siamese Network: The test dataset will benefit from the existing model constructed in the previous stage. For example, assume we trained the model on human faces, and as an input of the model, we give a pair of face images to the model. Our goal is to find out if they are the same person or if they are different persons. These two image passes to the network (the same network that is used for training), and the model provides us with a similarity score. If the similarity score is close to zero, it means they are not presenting the same person, but if the similarity score is high (i.e., close to one).

Text-to-Image Models

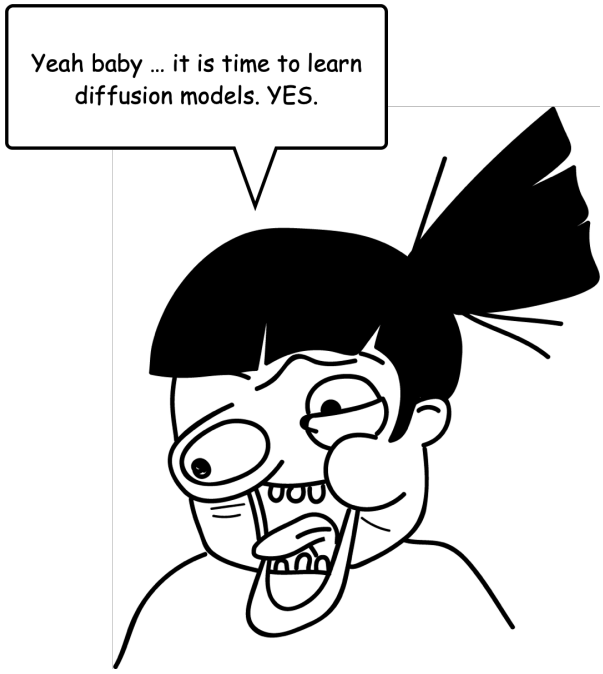
In the summer of 2022 when we revise this chapter, OpenAI is a pioneer in models that get the text and construct an image from the text, aka, text2image models. Since the author of this book has a mental disorder to explain to all models in the world to readers of the book, we list some of the popular text-to-image models here as well. If you have difficulties understanding the architecture of some of them, please come back here after reading Chapter 12. There you will learn transformer and thus be able to understand these models.

Before we explain models we need to get familiar with the concept of zero-shot learning.

Zero-Shot Learning: this style of learning enables the model to label data that it has not encountered in its training phase. However, during the testing, based on the given instructions and other labels (auxiliary information) it can make a label for the data that has not previously been labeled in the training set [Larochelle ’08, Chang ’08]. In other words, in Zero-shot learning, the model is trained to recognize and classify objects it has never seen before. This is achieved by providing the model with descriptions or attributes of the objects rather than actual examples.

How is such a thing possible? For example, a model gets a text. There are lots of descriptions of horses, and these texts are labeled as a horse in the training phase. Therefore, the model can label a text segment about a horse easily in its testing phase. However, there is no label for text about zebra provided, but when it encounters a text about zebra since it is similar to a horse and it has a stripes (auxiliary information), by looking at the text, it recognizes the horse and forms other parts of text it recognizes that the horse which has stripe is called zebra, then it assigns label zebra to that text (include information about zebra).

To remember this explanation in one sentence, *zero-shot learning enables the model to assign labels to unseen data in the training set.*



Now we learned what zero-shot learning is, but don’t get very happy because it requires a large-scale amount of input data, and at the time of writing this, only big corporations can provide their algorithms with such a huge dataset.

Autoregressive Models: The term autoregressive (AR) is often used in time series and refers to a model whose future values are regressed on previous values. In simple words, the autoregressive model predicts a variable based on its own past values, or a model is AR if it predicts future behavior based on leveraging past behaviors.

For example, ARIMA (Chapter 8) is an AR model. We can formalize the prediction of variable x at the t, by using k previous variables: $x_t = f(x_{t-1}, x_{t-2}, \ldots x_{t-k})$

There are differences between RNN and AR models. RNN model connections can go forward and backward, but AR model connections are feedforward only. RNN maintain hidden layers, but AR models do not use hidden layers, and thus they have limited output responses. There are more differences, but we don't need learning them to understand text-to-image generative models.

Diffusion Models: Another category of generative models are referred to as Diffusion models. Diffusion models are inspired by physics and in the context of physics diffusion is the movement of a particle from a high concentration to a lower concentration area, until a thermal equilibrium is achieved. For example, consider a spoon of salt that we put into the water. The salt molecules move in the water (diffuses) until the salt concentration, in the water, at every point is equal.

Diffusion models define a Markov chain (check Chapter 5) and in each Markov state, they gradually add noise (usually Gaussian noise, in Chapter 16 we describe more about noises). The process of adding noise gradually to a data and considering it as a Markov Chain is referred to as a diffusion step. Then, the diffusion model is trying to reconstruct the original data from the noise. The process of Diffusion is presented in Figure 11-34. They learn to generate new synthetic data by reversing a gradually added noise.

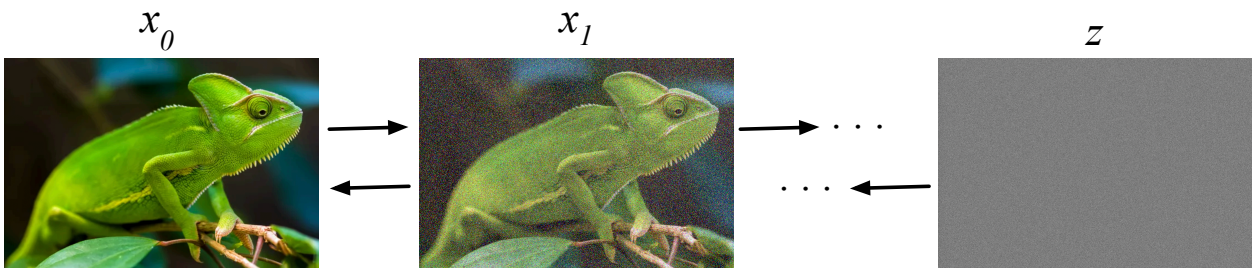


Figure 11-34: A diffusion model that gradually adds a noise at each Markov state, and then from z state, noise gradually reverse to reconstruct a data similar to the original data.

Diffusion models have a major difference from other generative approaches including VAE or GAN models. Their latent variables have the same dimension as the original data and the dimensionally of the data is not reduced during the diffusion process. In other words, diffusion models are trained to de-noise the noise (z) and get back to something similar to the original image (x_0).

CLIP (Contrastive Language–Image Pre-training)

CLIP [Radford '21] is an image-to-text translation approach proposed by OpenAI in 2021. In layman's terms, CLIP can be used to generate a textual caption for a given image[15]. Authors motivate the usefulness of CLIP for two reasons. First, image classification approaches are bound to a predefined set of user-defined labels used in their training set. For example, in the MNIST dataset, we are bounded to 10 labels, in ImageNet, we are bonded to 22,000 labels, etc. Each time a new image is added that has never been seen in the dataset before, it must be manually labeled. We have recently learned that zero-shot learning can resolve this need.

Second, classification labels are limited to one single label. A text caption (prompt text) contains more useful information than a single class label. For example, instead of the label "chihuahua" for a chihuahua dog image, we can have a text caption as "*The boss's chihuahua, whose bark sounds better than the voice of his owner, a.k.a., my boss*".

CLIP implements both zero-shot learning and text prompt labeling. It is trained on a very large number of images paired with their text captions, which are collected from the Internet[16]. Similar to other CLR methods CLIP input is also a set of image-text pairs.

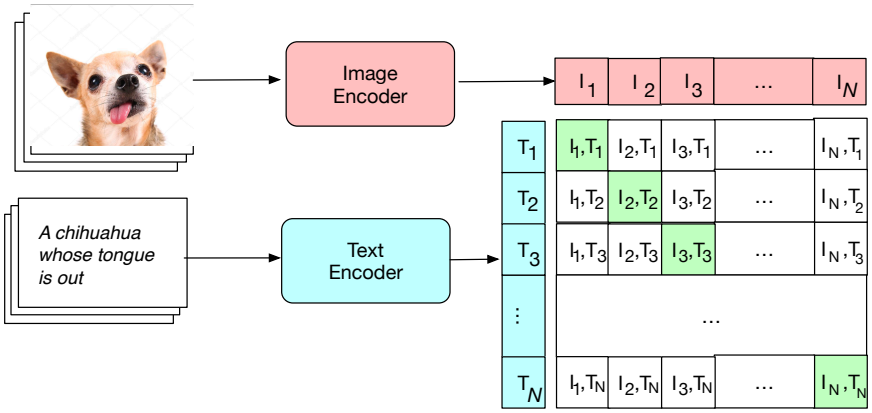


Figure 11-35: CLIP Pre-training phase, each pair of text and image gets into separate encoder and the green diagonal of the result matrix (it is a matrix of cosine similarity) includes positive samples and the white ones are negative samples.

Contrastive pretraining: In the first step, CLIP applies an encoding to both texts and images. Texts are encoded by Transformer (in Chapter 12 we will explain attentions and transformers), trained with BPE (in Chapter 14 we will describe BPE) with 49k vocabulary size. Images are encoded by ViT Transformer (with additional layer norm) or ResNet-50 (again we will explain

---

[15] Unlike other models in this section, CLIP is an image-to-text and not text-to-image, but it uses in many of text-to-image generations and thus we need to learn it.

[16] CLIP has trained (they call it pre-training) on 400 million pairs of image-text pairs. Authors build 500,000 queries and use a web search to find an image and text pairs. Words for queries are selected based on some condition such as occurring at least 100 times on Wikipedia, etc.

them in Chapter 13) whose pooling layers are substituted by the Attention pooling mechanism. You can find more details about the encoding process, and its configuration in their paper.

After both image and text data are encoded, it matches the representation (feature vectors that are the result of encoding) together.

Figure 11-35 shows the pre-training phase of the CLIP. For each image and for each text in the Image-text pairs set, it uses a separate encoding and the result is a vector. The main diagonal of the matrix in this figure (shown in green cells) presents positive samples (the cosine similarity should be maximized) and other cells on the matrix (white cells) present negative samples (the cosine similarity should be minimized). CLIP uses cross entropy as a cost function for pre-training, and implements it as a symmetric loss function, which results in the matrix shape shown in Figure 11-35.

Training: To train images authors experiment with different ResNet and different Transformer models with different resolutions and embedding (feature vector) sizes. Each of those image models is paired with a related Transformer for text training. Based on the result of the experiments Authors identify a large transformer model (ViT-L) for the image and transformer for text, providing the best accuracy.

The optimizer of the training is Adam and it is trained for 32 epochs. It has a hyperparameter, i.e. "temperature parameter", and it is equivalent to 0.07. This hyperparameter is used to clip and prevent scaling the logits by more than 100. The batch sizes used for training have a size of 32,768.

For this setting, the authors trained their model on only 256 NVidia V100 GPU only for 12 days. Yes, it is very affordable and you can train such a small model with a small budget in your basement. Such a pre-training is only possible to be implemented by a large corporation, which has a large number of resources, and even at the time of describing (mid-2022) CLIP, to our knowledge, no university can afford to provide such an infrastructure for pre-training.

Experimenting Zero-Shot learning: CLIP can be used to build both image classifiers and text classifiers. The authors present the zero-shot learning capability of CLIP by using class labels with some manually added prompt text. For example, in Figure 11-36, we feed a chicken image and the CLIP provides the caption for the image. The caption is composed of a text prompt that is manually given to the model (by a model engineer and not the end-user), in this example "A picture of" was manually given to the model. Then, it iterates through class labels (assigned during the training), adding them to the prompt sentence, and afterwards computes the encoding of each sentence. The sentence encoding that has the closest encoding to the image encoding is the result sentence.

Note that zero training is needed on the actual task of building the text caption in CLIP. In other words, the output of the text encoder in Figure 11-36, could be entirely different from the dataset built in Figure 11-35. Simply by finding the closest text (blue part in Figure 11-36) to the image encoder (red part in Figure 11-36) the CLIP will identify the best match based on similarity comparison.

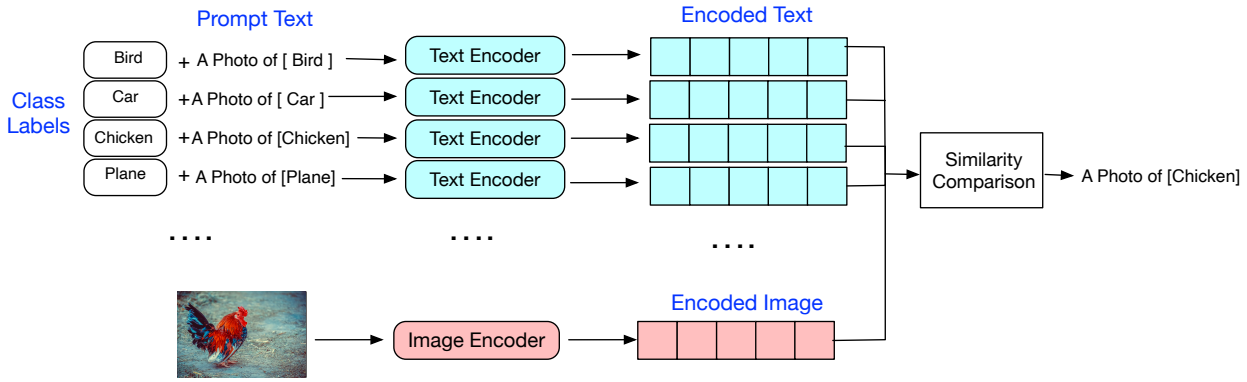


Figure 11-36: Zero shot learning of CLIP for text prompt prediction. Note that text labels has no information about images. The label will be assigned based on comparing the similarity of encoded text to the encoded image.

In summary, we can say CLIP tries to ensure that the similarity between the encoded image and encoded text is high.

VQ-GAN (Vector Quantized GAN)

VQ-GAN is not image-to-text or text-to-image, it can construct high resolution synthetic images. It is used in combination with CLIP and thus enables building images from text [Crowson '22].

Transformer architecture (we will explain it in Chapter 12) is capable of capturing complex relationships between objects in the image. Nevertheless, it is incapable of capturing local information on the image. In contrast, CNN architecture is capable of identifying local relationships properly, and not being able to capture complex higher-level information.

In other words, the transformer is good at capturing spatial information from pixels that have long distances far from each other, such as the shape of the object. On the other hand, CNN is good at capturing spatial information from pixels that are close to each other, such as the texture of the object. VQGAN combines both architectures to produce "high-resolution synthetic images". To explain it we separate our explanation into two stages.

Stage 1: VQ-GAN is inspired a lot by VQ-VAE [Van Den Oord '17]. The main motivation of VQ-VAE is to substitute the continuous latent space of VAE with a discrete latent space. Having discrete latent space, enables the model to use less data for reconstruction and thus make it faster. To implement this discretization, the authors of VQ-VAE use a vector quantization and construct a codebook. Please check Chapter 16 to read about "vector quantization" from the image and how a codebook is constructed. In short, a codebook is a dictionary of discrete vectors all in the same dimensions. Codebook is a dictionary that is used for the reconstruction of the image, for example, it has a vector for a sunny sky, a vector for the garden, a vector or more than one vector for a dog, etc. Then if a query asks to build "a dog in the garden at the sunny sky" these vectors were retrieved from the Codebook and the decoder construct the image.

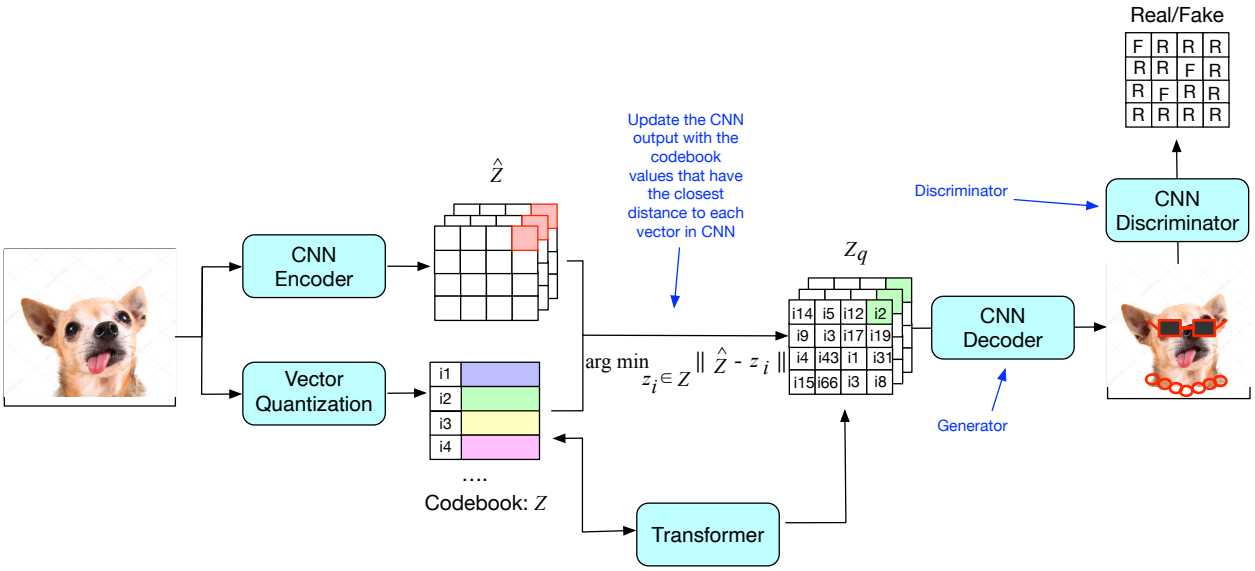


Figure 11-37: VQ-GAN architecture.

Why use the codebook? Because instead of dealing with patches of images, which result in too many different values in latent space, the model can learn from a limited number of values in latent space and thus can index them as well.

VQGAN similar to other GAN models includes a synthetic image generator and discriminator. The generator is a decoder of an Autoencoder as it is shown in Figure 11-37. The encoder includes a CNN whose output ($\hat{Z}$) reduces the size of the input image while maintaining its spatial information and feature information. Besides, it quantizes the image into vectors and stores it in a codebook (Z).

Then, a tensor (Z_q) with the same size as the CNN output will be constructed as follows: each vector of CNN is compared (via Euclidean distance) to a vector inside a codebook and the closest vector in the codebook will be used to substitute the CNN vector value. As a result, instead of the first CNN result ($\hat{Z}$), we have another tensor with the same size as the CNN result, i.e., Z_q, but it includes vectors from the codebook and not CNN. Figure 11-37 visualize the construction of the codebook from the image. As an example, we highlight one channel vector on the $\hat{Z}$ tensor in red. This vector will be compared to the vectors in the codebook and the one that has the shortest distance from the codebook (light green vector) will substitute the vector from $\hat{Z}$ in Z_q.

Since CNN vectors are substituted with the codebook value, from the decoder (which we explain later), it can not back propagate. To handle this issue, the authors use the approach proposed by VQ-VAE, and it is simply, copying the gradient values from the decoder tensor to the same position on the encoder tensor.

Now, the decoder (Generator) operates as other GAN models, but the codebook will get updated during the backpropagation and not the encoder. This means instead of propagating back to the encoder, the algorithm propagates back to the codebook and updates the content of the codebook. This approach enables the codebook to learn features in the image as well.

The authors recommend to start the Discriminator later in the training because the generator has time to learn. Otherwise, the discriminator can easily recognize the fake image and thus the Generator can not learn

Stage 2: Until now we have a GAN that only reconstructs the data. However, the goal is to generate large synthetic images with high resolution. Encoder and Decoder are invariant to image size, but the transformer can assist the decoder to build larger images. To solve it transformer will be used in an autoregressive manner. Besides, we have explained that the codebook is acting as a reference that includes elements of synthetic image construction.

The task of the transformer is to learn which codebook vector members (codeword) are required for the image construction and add them into a sequence of patches for image generation in Zq. For example, as it can be seen from Figure 11-37, the transformer orders codewords as $i_{14}, i_5, i_{12}, i_2, \ldots$ in Z_q which send to the CNN decoder for synthetic image construction.

Why do we feed the codebook to the transformer and not the output of the Encoder? The transformer requires sequential data as input. However, since it uses self-attention it is computational very hard to feed a large sequence to a transformer. Therefore, the codebook is fed into the transformer, and also the authors use a sliding window for the attention model to reduce the size of the image and thus the computation.

To our understanding, the transformer acts as a decoder and upsamples data from the codebook. In other words, it is used to enrich the content of the codebook and thus enable the Generator to build larger than input synthetic images. For training, the transformer uses cross-entropy loss and compares the codebook's original value with the modified codebook (which is a result of training).

We do not explain the cost functions of the VQ-GAN but in short, it uses both the GAN cost function and the cost function introduced in VQ-VAE.

In summary, instead of constructing images from pixels, VQGAN uses the discrete variables extracted from the codebook and the transformer improves the content of the codebook. Therefore, the synthetic images of the Generator are high resolution and can be larger than the original one.

If you would like to baffle about your deep understanding of VQ-GAN say the following in front of others: *Using the concept of the codebook and feeding it into a transformer makes VQ-GAN very attractive and show its high accuracy in generating high-resolution images.*

VQGAN-CLIP

We have explained CLIP and VQGAN to reach this interesting and practical approach. Crowson et al. [Crowson ’22] use a CLIP to guide VQGAN and build a practical text-to-image approach. Respectfully, they share their trained model, unlike models that are provided by large corporations, and they are infeasible to be used by other developers.It is an interaction between VQ-GAN and CLIP networks, which results in using three neural networks, i.e. generator and discriminator of VQ-GAN and CLIP networks.

The objective of VQGAN-CLIP is to generate novel images from the given text. Recall that the VQ-GAN network generates a high-quality image from the text prompt, then the CLIP network assesses the fitness of the text (image caption) to the image, by comparing it to other captions.

In short, VQGAN-CLIP operates as follows; the user inputs a text (prompt) for the desired generated image. In the beginning, VQGAN generates a completely random noisy image, which includes only random pixels. Then, this image will be assessed by the CLIP to see if it matches the user-given prompt. A loss score will be reported to explain how far the generated image is from the text prompt. The loss is backpropagated to VQGAN. Then VQGAN improves its image generation process. This process continues iteratively until a maximum threshold reaches.

We skip the small details to be read in the original paper, but one thing that is worth explaining is the challenge that existed with the CLIP loss. The gradient updated from CLIP is noisy if it is calculated on a single image. To overcome this issue, the authors apply several image augmentation techniques (we will discuss them in Chapter 16) to the generated image to have several images. Their image augmentation technique includes first taking random crops of the candidate image and then applying flipping, noising, etc.

DALL E Models

DALL E version 1 [Ramesh ’21] has been introduced back in early 2021 by OpenAI. They use a transformer architecture and autoregressive model that assume text and image as a single stream of data.

DALL E v1 includes 12 billion parameters and uses a dataset of 250 million text-image pairs collected from the Internet. As you can see it is very small and you can train this model on your phone while lying in bed.

Earlier we have explained that autoregressive models are used for sequential data (token). Authors claimed that they can not consider the pixel of an image as a token of data for the autoregressive model. Because most of the model focus will be on capturing short-range dependencies between pixels and thus long-range relations of pixels will be neglected.

To handle this issue DALL E v1 operates in two stages. The first stage is focused on training a discrete variational autoencoder (dVAE) to compress image data (result in image token). dVAE is similar to VQ-VAE which builds a code book. The difference is that, unlike VQ-VAE which selects one codeword (a codebook vector) from the codebook, dVAE can express some uncertainty, by outputting a distribution over codewords for each latent variable, instead of a

single codeword. Nevertheless, since backpropagation is not able to work with discrete data (discrete data are not differentiable), to transform them into the data continuous authors use a method called Gumbel-softmax [Maddison '16, Jang '16]. The result of this stage is to convert an 256×256 RGB image into a 32×32 grid of tokens. The codebook contains 8192 vectors.

The second stage first concatenates image tokens to BPE-encoded text tokens (check Chapter 14 to learn BPE). Then they trained an autoregressive transformer to model the joint distribution over text and image tokens, by using their 12 billion parameter transformer.

As a result, a large dataset of image and text token pairs is trained. Then they can be used in an autoregressive fashion and predict the next image token in a sequence. In other words, the next tokens in a sequence could be predicted in an autoregressive fashion and a decoder building the entire image from a given text.

DALL E 2 [Ramesh '22] was released in the summer of 2022, it can construct significantly more realistic photos and accurate photos than its earlier version. In contrast to DALL-E v1, which is an autoregressive model, it is a diffusion model.

DALL E 2 is composed of two main components, a "Prior" and a "Decoder".

First, a CLIP model [Radford '21] is trained and each pair of images and text will be encoded to matching image embeddings (z_i) and text embedding (z_t). The general objective of CLIP is to specify how much the given text relates to an image. The top part of Figure 11-38 presents the CLIP training for DALL E 2 architecture. After the CLIP model is trained, it gets frozen.

Next, the "Prior" network uses the frozen CLIP and produces the image embeddings (z_i) that are conditioned on the caption (y), i.e., $P(z_i \mid y)$. In simple words, Prior maps the text embedding z_t to the image embedding by using the CLIP frozen model. Authors claimed that they used both diffusion and autoregressive for the Prior.

Afterward, the diffusion "Decoder" produces images conditioned on image embedding (from Prior) and optionally on the caption, i.e., $P(x \mid z_i, y)$. It means that unlike the diffusion we have explained earlier, this diffusion model does not start from noise. Instead, it starts from the image embedding produced by Prior. The objective of the decoder is to *invert the image embedding that CLIP has learned*. Figure 11-38 presents the DALL E 2 architecture.

Now, you might ask: why does the decoder perform inversion? the goal is to generate a new image from a text prompt and not build the identical image as it is encoded by the Prior. This inversion assists the decoder in creating a new image. To implement the decoder with inversion, they use a modified version of the GLIDE [Nichol '21] (another text-to-image from OpenAI).

The combination of Prior and GLIDE based Decoder is called unCLIP by authors because it reverses the mapping learned by the CLIP image encoder.

From the formalization perspective, by stacking the Prior and Decoder, DALL-E v2 builds a generative model $P(x \mid y)$ of images x given captions y, i.e., $P(x \mid y) = P(x \mid z_i, y)P(z_i \mid y)$.

You might also ask why do authors use Prior and not simply CLIP encoder with a Decoder? To our understanding, with experiments, they realize that having such a network in between

produces better images. Authors claimed Prior creates a gist or summary of the image and the Decoder invert images given their CLIP image embeddings, during this process Decoder learns the details that are useful for image reconstruction but Prior (frozen model of CLIP inside Prior) does not consider them.

DALL E 2 generates high resolution images and to generate high resolution images, they train two diffusion up-sampler models. The first one upsamples images from 64×64 to 256×256 resolution (with Gaussian blur to improve robustness of the upsampling), and the second one applies further upsampling to make a 1024×1024 resolution (with something called BSR degradation [Rombach '22, Zhang '21] to improve robustness of the upsampling). We do not explain the detail of BSR degradation, because it is out of scope here.

To train, the encoder authors use CLIP and DALL-E datasets, which include about 650 million images. To train the encoder, upsamplers, and prior they use about 250 million images of the DALL-E dataset.

To summarize, DALL E 2 operates in three steps. First, a text encoder is trained to convert the user input text (text prompt) into a text embedding (vector instead of a text). In the second step, Prior maps the text encoder result (text embedding) into a corresponding image embedding. The image created by Prior has semantic information from the input text. In the third step, the "Decoder" generates images from the image encoding created by the Prior[17].

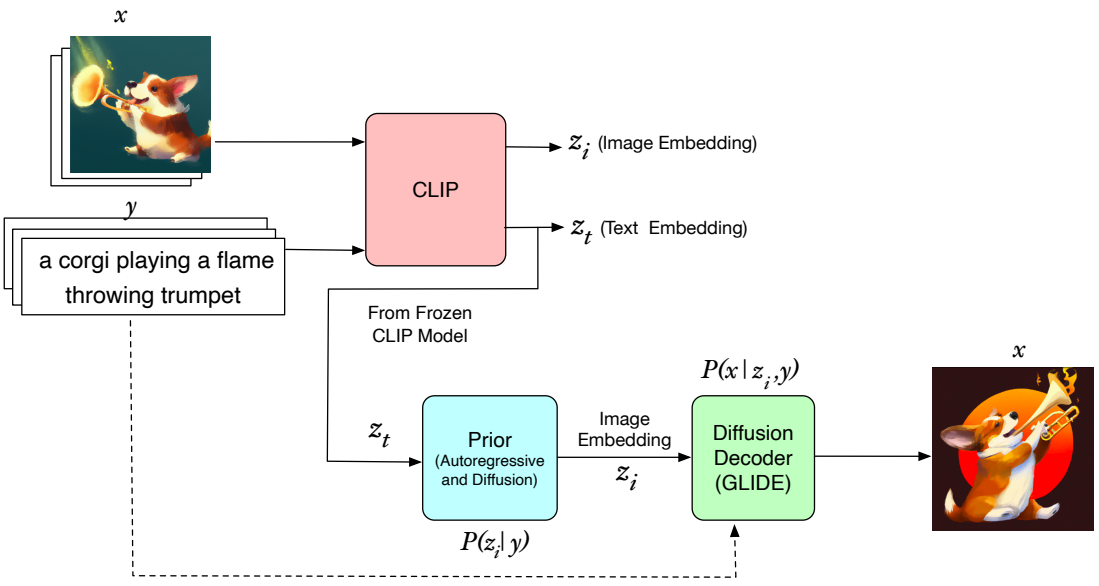


Figure 11-38: DALL E 2 architecture, which is composed of three networks, CLIP, Prior and a decoder, which is a GLIDE model with some changes.

[17] At the time of writing this part, DALL E is released in less than a month and we have trouble finding good resources to explain it. Even its paper is not peer-reviewed and not easy to understand for us. You can also check the Blogpost of one author to better understand DALL E 2, http://adityaramesh.com/posts/dalle2/dalle2.html

IMAGEN

Do you recall rich family members who do not have anything to do, except try to copy what another auntie did, and show off their life includes more luxury than others? If you think those behaviors belong to usual people, and big corporations don't do the same thing, you might be wrong. Anyway, this explanation has nothing to do with the rest of this section.

A few weeks after the release of Open AI's DALL E 2, Google released Imagen [Saharia '22], and few days later it releases Parti [Yu '22] both are text-to-image models.

Imagen architecture is significantly easier than DALL E 2 and in this paper authors also introduce an evaluation method, DrawBench, to evaluate the result of text-to-image algorithms. We do not go into the details of DrawBench. Figure 11-39 presents the architecture of Imagen.

Imagen uses a large language model, T5-XXL (we will explain T5 model in Chapter 12), pre-trained on text corpora (Check Chapter 12 to learn about language models) to map input text into a text embedding. Authors experiments show that using this language model outperforms CLIP accuracy.

Next, they adapt a conditional diffusion model provided in [Dhariwal '21] that gets the text embedding and constructs the image. Their diffusion model introduces a concept called

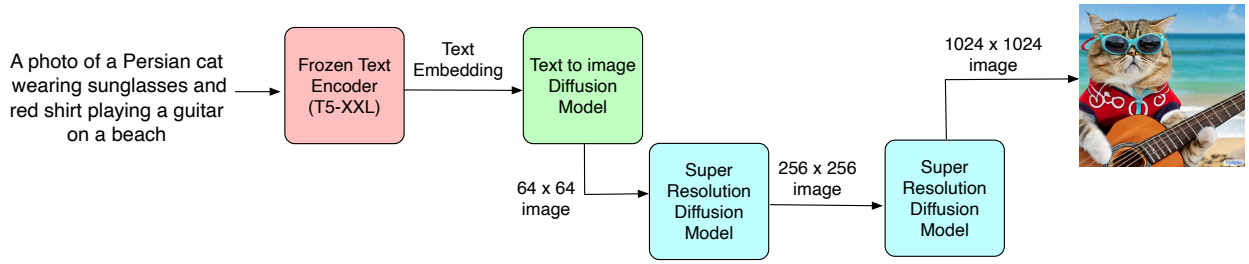


Figure 11-39: Google's Imagen architecture.

"dynamic thresholding", which is a sampling method to leverage weight guidance. This weight guidance results in having more photo-realistic images at the end.

Afterwards, they use two super-resolution diffusion models for generating 256×256 and 1024×1024 images. Their super-resolution architecture is based on their customized U-Net model (Efficient U-Net).

To train the T5 XXL they use 800 GB of textual data and the model has 11 billion parameters. Their 64×64diffusion model is trained on a 300 million parameters model, which is conditioned on text encoding. Their UNet models have 300 million to 2 billion parameters.

Parti

Parti [Yu '22] stayed for the "pathway autoregressive Text-to-Image model" and it was introduced few days after Imagen. Unlike Imagen, which uses a diffusion model, it is an autoregressive model. Recall that an autoregressive model takes a sequence of tokens as input and it predicts an upcoming token(s).

Parti treats text-to-image generation as a seq2seq model which is used for machine translation (check Chapter 12 for seq2seq models). Unlike DALL E 2 which provides a decoder only, it provides both image encoder and image decoder, by using ViT-VQGAN [Yu '21], and the authors claimed it has outperformed decoder only models.

We have explained VQGAN earlier here, ViT stayed for vision transformer which we explain in Chapter 12. Parti has a simple, but effective architecture. It has three components and all are based on the transformer.

Parti operates in two stages, an "image tokenizer" and an "autoregressive model". The first stage trains the image tokenizer (encoder) to convert an image into a sequence of discrete visual tokens (image embedding). As we have explained before in VQ-GAN, we should repeat that tokenizing an image into a sequence of pixels results in an extremely large sequence. VQ-GAN and DALL-E 2 handle this challenge with discrete types of VAE, and instead of using patches on the input image, they use codebook values. Parti authors downsample an input image with the size of 1024×1024 to 256×256. Then, gives it for training to ViT-VQGAN. The resulting codebook in this model has 8192 entries (codewords). The authors used ViT-VQGAN for encoding images as a sequence of discrete tokens and here tokens are codewords, which come from the codebook (not image patches). To train ViT-VQGAN authors follow the DALL E model. After training is done at this stage, the model freezes ViT-VQGAN.

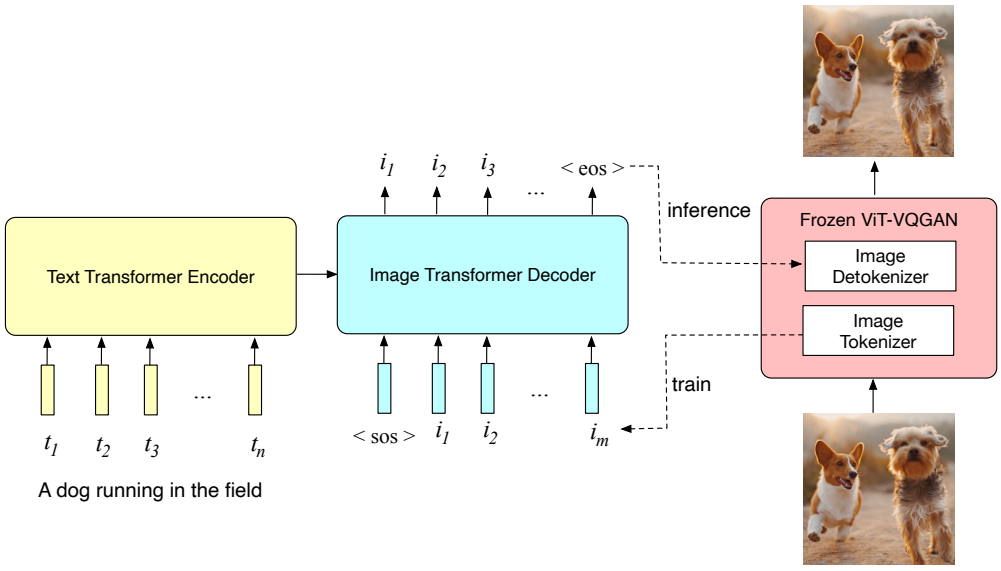


Figure 11-40: Training Autoregressive seq2seq model that generates images token from text token.

Since the frozen model of ViT-VQGAN operates with 256×256 image size, later the synthetic image that is reconstructed by ViT-VQGAN will be fed into a super-resolution layer and upsamples the reconstructed image to a size of 1024×1024.

The second stage focuses on training an autoregressive encoder and decoder transformers (the yellow and blue transformers in Figure 11-40), by treating text-to-image as a seq2seq modeling. The input for training is image and text pair. The Image Transformer Decoder, shown in Figure 11-40, gets the (i) sequence of image tokens (extracted by ViT-VQGAN from the image) and (ii) the encoded text prompt (the Transformer Encoder (blue color in Figure 11-40) converts it to text embedding), then it tries to predict the next image token. The text embedding acts as a target for attention (attention and transformer are explained in Chapter 12).

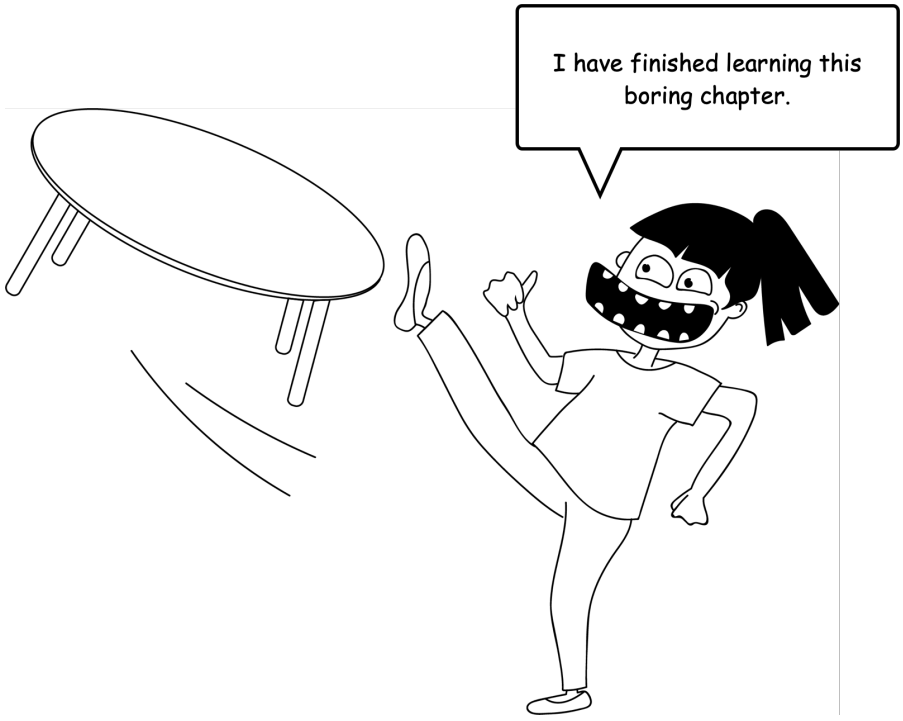
After the model is trained, it can be used to generate images, by feeding it the text prompt.

After the second stage of training, they freeze the encoder and codebook and fine-tune the model, which results in a 600 million parameter model.

Parti model that provides most photo-realistic images has 20 billion parameters. The super-resolution module has about 15 million parameters for the 512×512 version and about 30 million parameters for the 1024×1024 version. Their image-text paired datasets are vaguely explained, but it seems they use several internal to Google and public datasets, and they use about 6 billion text-image pairs.

Guided Video Synthesis

https://research.runwayml.com/gen2



Summary

This section focuses on unsupervised neural networks. It starts by describing concepts that convert the data into a low dimensional representation of the original data.

We explained latent variables, which are constructed by hidden layer neurons is hidden, which is in contrast to observed variables that are either input or output variables of a neural network. Next, we describe generative (joint probability) vs. discriminative (conditional probability) models. Afterward, we define deterministic (the output is determined by model parameters) and stochastic (the output is random) models.

Next, we describe SOM as one of the oldest unsupervised neural networks that is still in use, mainly for dimensionality reduction.

Then, we describe Autoencoders. An autoencoder has two main components, an encoder which reduces the dimensionality of the data, and a decoder which tries to reconstruct the original data from the given data in the low dimension, i.e., latent space. The process of training in Autoencoders includes finding weights for the encoder that minimize the error of data reconstruction.

There are two types of Autoencoders, regularized Autoencoders (denoising, sparse and contractive Autoencoders) and Variational Autoencoder. Regularized Autoencoders map a set of data points directly to one single data point in latent space, but Variational Autoencoder Map a set of data points to a multivariate Gaussian distribution around a point in the latent space. This makes them very flexible in constructing synthetic data that they have never encountered before.

Afterward, we describe U-Net, which is a state-of-the-art algorithm for medical image segmentation. U-Net is not called Autoencoder, but it is also composed of an encoder, and then the encoded compressed data will be retrieved and reconstructed by the decoder. Nevertheless, between layers, there is a concept called skip connection, which concatenates the encoder output (that includes "what information") to the decoder output at the same level (that includes "where information").

Lastly, we introduce GANs. GAN is the min-max game between two neural networks, Discriminator and Generator. Each of these neural networks has its cost function and its own parameters. During the training phase, their parameters were tuned, similar to other neural networks.

The Generator starts from noise and improves it until it builds fake data, which has a very similar distribution to the original data. In particular, it takes a point from latent space as input and generates new fake data. The Discriminator employs Backpropagation to minimize the loss score for real and fake data examples it receives. The Generator tries to minimize the Discriminator's loss for fake data (motivates the discriminator to consider the fake data as real data). In other words, Generator tries to increase the false-positive errors of the Discriminator, while Discriminator tries to minimize the false-positive and false-negative of its classification.

The basic form of GAN is associated with several challenges, and later we introduce some GAN architectures that try to mitigate those challenges. We have explained seven GAN models, but

there are still ongoing efforts to build new GAN systems, and the ocean of GAN models has no end.

Afterward, we introduced Contrastive learning representations and their common cost function (triplet loss and contrastive loss), and later, we explained how the Siamese network operates.

We finished this chapter with Text-to-Image approaches which three of the top ones are introduced by big corporations and are not accessible to the public yet. Nevertheless, since data existing on the Internet is biased, racist, and against minorities, releasing these models without proper restriction might be challenging for them as well. Besides, they might keep it to sell access to their trained models and benefit from it financially. We would respect approaches such as Cariyon[18] or StableDiffusion[19] [Rombach '22] that create something similar with a limited amount of resources but release their model as open source.

Further Reading and Watching

* There are several online videos for learning RBM and tutorials. To our understanding, either they go into too many details or they require a deep mathematical understanding. One of them, which we found to be helpful is the video of Frank Noe and we used it to explain details of RBM algorithm, https://www.youtube.com/watch?v=wfFf5Fj-rzE

* Another good tutorial to learn RBM and DBN is the monograph of Bengio [Bengio '09], that tutorial is among the few ones, that explained these algorithms in simple terms and does not need significant prior knowledge, assuming you have read Chapter 3 and Chapter 8.

* If you are interested to understand the details of RBM and DBM, a good tutorial with lots of mathematical detail is for Hugo Larochelle under this link: https://www.youtube.com/watch?v=35MUlYCColk. Nevertheless, beware that it goes fairly deep into the mathematical explanations but by far the best available online tutorial at the time of writing this book. His autoencoder tutorial is also very detailed and useful.

* A short but good explanation for some AutoEncoder is provided by B.K. Biswas from IIT Kharagpur University (India), under this link https://www.youtube.com/watch?v=w8xbd8XI7U8.

* Andrew Ng has a good explanation about Sparse Autoencoder and delves deep into the math detail in his lectures: https://web.stanford.edu/class/cs294a/sparseAutoencoder.pdf Nevertheless, we need to be familiar with other types of Autoencoder as well.

---

[18] https://github.com/borisdayma

[19] https://github.com/CompVis/stable-diffusion

* Lilian Wang has an excellent brief article on Autoencoders for further reading: https://lilianweng.github.io/lil-log/2018/08/12/from-autoencoder-to-beta-vae.html. Also, she has a good tutorial on WGAN, which briefly explains how WCGAN works. https://lilianweng.github.io/lil-log/2017/08/20/from-GAN-to-WGAN.html#wasserstein-gan-wgan

* At the time of writing this chapter, two books are published about GAN, *Generative Deep Learning: Teaching Machines to Paint, Write, Compose, and Play* [Foster '19], by Foster and *GANs in Action: Deep learning with Generative Adversarial Networks* [Langr '19] by Langr and Bok. Both books are well written, and if you like to have a more hands-on experiment with GAN, we highly recommend checking them.

* Soheil Feizi has a good introduction to GAN in mathematical language. His video lectures are available as well: https://www.youtube.com/watch?v=IzaerbzSB64

* Jason Brown Lee has a good introduction and implementation example on FID score https://machinelearningmastery.com/how-to-implement-the-frechet-inception-distance-fid-from-scratch/

* There is an online course that teaches some of the common GAN models. Especially we used its explanation about StyleGAN, https://www.coursera.org/learn/build-better-generative-adversarial-networks-gans

* We skip explaining DALL-E 1, and dedicate most of our explanation to DALL-E 2. If you are curious to see how dVAE works in detail, you can check this blogpost: https://ml.berkeley.edu/blog/posts/dalle2

* At the time of writing text-to-image models, there are very new and very few auxiliary explanations available for them. We benefit a lot from the Yannic Kilcher online videos (https://www.youtube.com/c/YannicKilcher) to learn these models. His explanation is fairly ok, or at least better than the DALL-E 2, Imagen, and Parti papers, which seem they all be written in haste.