Chapter 11: Self-Supervised Neural Networks

In the previous chapter, we have described CNN and RNN as two popular neural networks. These neural network architectures fall into the category of supervised learning. 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 neural networks*, a type of unsupervised neural network often associated with *generative models*. Most supervised models we have learned are doing either classification or regression, but self-supervised algorithms *reconstruct* the given input data. Previous neural network models we have explained get the input data, process them, and provide output. They include train (fit) and then test (predict). From now on, 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 be removed.

Recent advances in AI, including applications such as colorizing black-and-white photos, restoring old photographs, converting photos into videos, and generating images from text

Figure 11-1: An example of a black-and-white picture that is colorized by a generative neural network.

descriptions, leverage these algorithms. For instance, in Figure 11-1, we demonstrate an old photo to a neural network that uses a generative model, and it colorizes the old photo. Figure 11-2 showcases a text-to-image generation model, DALL-E v2 [Ramesh '22], which, as of 2023, can create detailed images based on text descriptions provided in captions. These examples highlight the capabilities of current models and contrast them with earlier, less sophisticated versions. At the time of reading this chapter, you might have wondered how the high quality of the text-2-image models or other models improved during the time and how dull are the versions you see in this chapter.

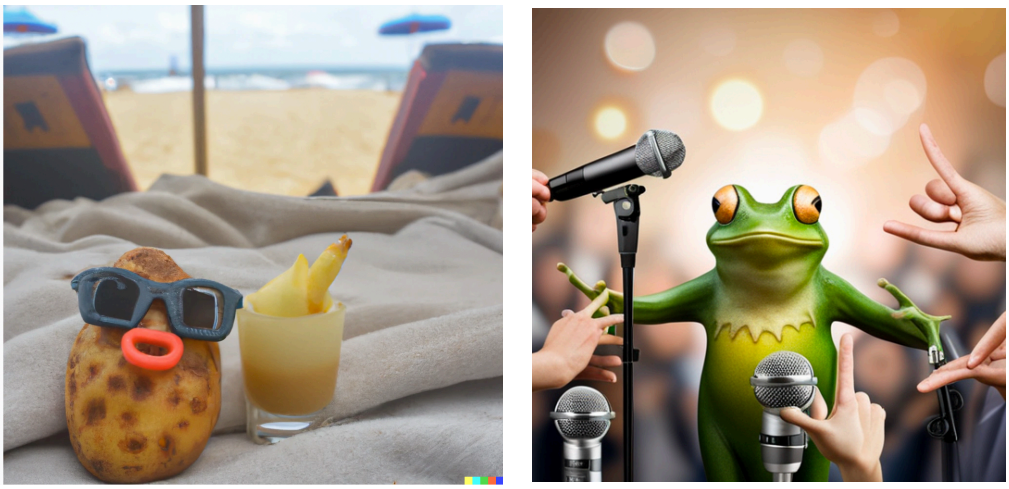


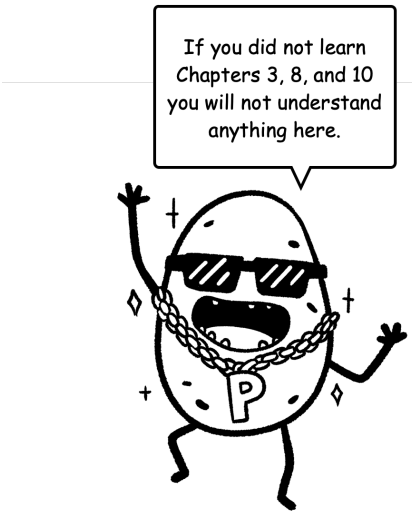
Figure 11-2: Text-to-image construction example. (right) DALL-E v2, text prompt: "A happy potato on the beach is drinking Pinacolada". (left) Stable Diffusion XL, text prompt: "A frog is talking behind the microphone for a crowd".

We start this chapter by describing Self Organizing Maps (SOM) or Kohonen Maps, an old neural network whose main objective is to change the representation of the input 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 problems[1]. Next, we describe the Restricted Boltzmann Machine (RBM) and its varieties as initial models of both RBM and SOM are fairly old, but 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 strongly recommend learning as many algorithms and models as you can.

Afterward, we describe Autoencoders (AE) and then Generative Adversarial Networks (GAN), which are generative models and very popular until the introduction of diffusion models. Next, we explain Contrastive Learning methods, followed by a discussion on text-to-image models and

[1] Traveling Salesman Problem (TSP) is a known problem in computer science that 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 return to the origin city?

their concepts and algorithms, such as diffusion models, zero-shot learning, CLIP, VQGAN, etc.



These models lead a new direction known as generative AI, e.g., generating text, illustrations, music, and videos. 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, there is another fat chapter about state-of-the-art architectures and models for neural networks. Nevertheless, by finishing this chapter and learning all its algorithms, you will be another person. Do not take it as an advertisement; the algorithms of this chapter are exciting to learn.

If you start this book from the beginning, at this point, you have a good understanding of distributions and probabilities. You are ready to learn new topics. If you skipped those chapters, we strongly recommend reading Chapter 3, Chapter 8, and Chapter 10 before starting to read this chapter. Otherwise, you will have trouble understanding this chapter.

Before we start our explanations, we should be familiar with a few basic terms, which we have tried not to explain in earlier chapters.

Representation Learning Concepts

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 learn a low-dimensional representation of the data (*latent space*) and train their model in the latent space.

It 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. Latent space includes *features that best describe the characteristics of data*. Features that are less important to describe the characteristics of the data are either eliminated or their impact is reduced in the latent space. 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 *observed variables*.

Generative vs. Discriminative Model

Machine learning and artificial intelligence algorithms 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 $p(x, y)$.

On 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, and these models try to learn the "distribution" of the training data, e.g., GMM (Chapter 4) and HMM (Chapter 5). Once a generative model learns the distribution of the train set, then it can decide on the labels for the test dataset based on the distribution. An easy explanation about generative models is algorithms that deal with data distribution.

In contrast, a discriminative model *discriminates between different data points, disregarding how they have been generated*. In a more technical sense, discriminative models do not need to understand how data is generated; instead, they focus on finding the optimal decision boundary to separate classes using the input features, e.g., logistic regression or SVM (Chapter 9).

Since generative models take into account dependencies between data points, they are very powerful algorithms. On the advantage of generative models, David Foster [Foster '19] stated that "*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. A human can not infer their relationships, but neural networks can. Discriminative models do not take into account feature dependencies, and thus, they are not very capable of generating synthetic data.

Discriminative models usually operate with labels, and thus, they have been categorized as supervised learning algorithms. However, generative models can be used in both supervised and unsupervised settings, and mainly, they are used in unsupervised settings. However, to keep them separate from traditional unsupervised learning algorithms, scientists call them self-supervised models, In other words, self-supervised learning refers to techniques where *models generate their own training signals from the data.*

Deterministic vs. Stochastic Model

A *deterministic model* produces consistent outputs for the same set of inputs and parameters, given fixed initial conditions. In contrast, a *stochastic model* incorporates elements of randomness, which means its outputs may vary with each execution, even under identical conditions. Therefore, every execution of a stochastic model might have different results. Examples of stochastic models are genetic algorithms (check Chapter 6) or manifold dimensionality reductions, tSNE, and UMAP (check Chapter 7). Most heuristics approaches that try to find the optimal answer, including deep neural networks, belong to stochastic models.

Generative models typically exhibit stochastic behavior because they sample from learned data distributions, making their outputs inherently variable. This stochasticity introduces challenges in using them for end-user applications, though there are techniques to mitigate these effects, which are beyond this basic explanation.

Self Organizing Maps (SOM)

One of the oldest practical artificial neural networks is Self Organizing Maps (SOM), which were invented back in 1982, by Teuvo Kohonen [Kohonen '82] in Finland. Finland is a small country in northern Europe where the Linux operating system and Nokia (the indestructible smartphones of early 2000) came from.

SOM changes the representation of the data, it can reduce the dimensionality of data, and its results can be used for clustering, classification, visualization, etc. We feed a multi-dimensional dataset into a SOM, it outputs a grid representation of the input dataset. Similar to other dimensionality reduction methods that we have described in Chapter 6, the SOM algorithm projects data into a different dimensional space while maintaining the topological properties of the data. Topological properties ensure that the relative relationships and neighborhood structures in the data are preserved during the projection of data into different dimensions.

SOM is a very special type of ANN. It does not have an activation function and hidden layers. 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 to adjusting weights.

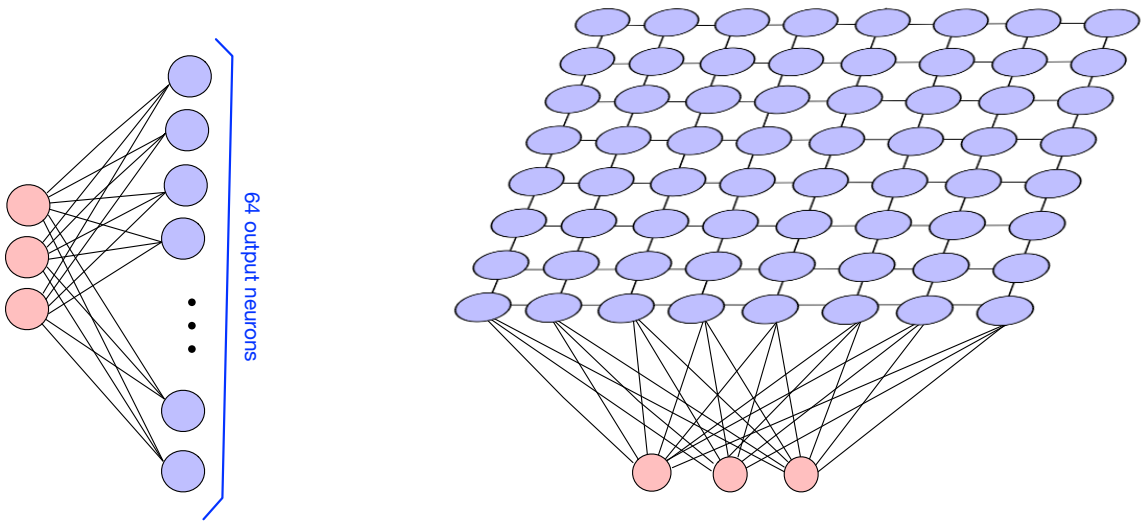


Figure 11-3: A sample SOM, whose red neurons are the input and blue ones are the output. Until now, we show neural networks like the left side, but SOM is usually shown as the right side of this figure, which is a rotated version of the left side (matrix of neurons).

The input layer of SOM is a vector equal to the number of features. For example, if we have a five-dimensional dataset, e.g., a table with five columns, the input vector includes five 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 $5 \times \sqrt{164} \approx 64$ output neurons. Take a look at Figure 11-3; there, we have a dataset with

three features (three input neurons), it includes 164 records, and the number of output neurons is $5 \times \sqrt{164} \approx 64$.

The output layer of SOM can also be visualized as a *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 11-4). The patterns of color correspond to the distribution of data. Usually, by looking at the topographical map, we could identify clusters of data, or compare two or more topographical maps to identify some correlations. For example, Figure 11-4 shows that by collecting many parents' advice, we realize that there is a correlation between life satisfaction and education.

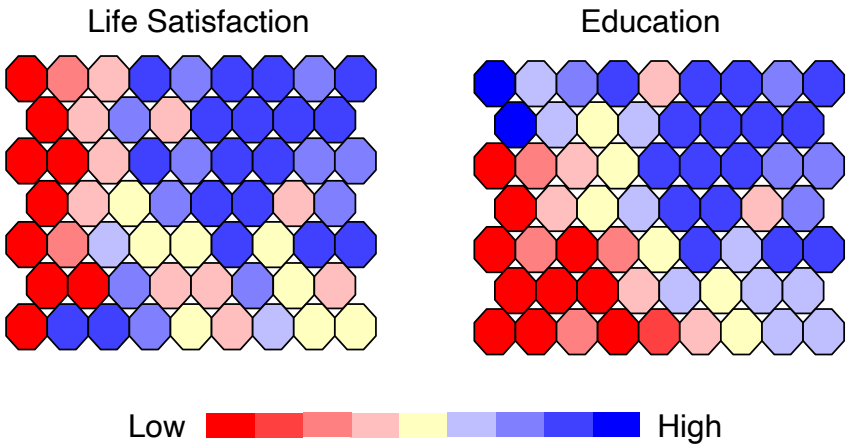


Figure 11-4: Two hexagonal topographical maps, which is the outputs of the SOM algorithm on a multi-feature dataset.

The result is shown in two hexagonal topological maps. By looking at these two images, we might see that higher education correlates with life satisfaction. Higher education is usually correlated with life satisfaction, but as 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 a few points with low education but high life satisfaction.

Higher dimensional SOM is also possible, but it is not common. The blue layers in Figure 11-3 are the lower dimensional representation of the input dataset. The output layer of the SOM is called a *lattice*, a specific mathematical structure. In simple words, the 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. For example, a 2D grid lattice can be seen as a graph where each node has edges connecting it to its immediate neighbors (up, down, left, right). Therefore, while not every graph is a lattice, every lattice can be represented as a graph.

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

(i) *Initialization:* All weights of the neural network are initialized with a small 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, called the discriminant function.

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

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 11-5) neuron. The final w is the presentation of data points in the projected dataset.

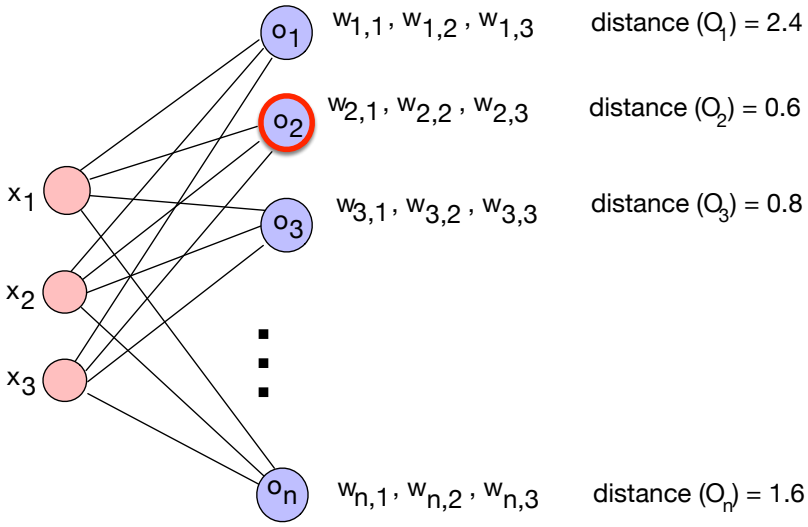


Figure 11-5: For each output node in the SOM, a distance between the given input vector and the weights of each node is calculated. Then, the node with the shortest distance is selected as the winner, i.e., here it is, and we mark it around with red.

To better understand this process, let's take a look at Figure 11-5. There, we fed one record from a table with three columns (for every column, assume one input neuron) 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 most similar one because its distance is the shortest one). The winning node is also called the *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 neighbors of the BMU. The radius size around the BMU

starts large, potentially covering the entire output layer's lattice, and decreases exponentially with each iteration, helping to fine-tune the learning process as the algorithm progresses, as shown in Figure 11-6. Each time a new record is fed into a network, the radius size will be changed (reduced) by the exponential rate.

(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, which are closer to BMU, get higher

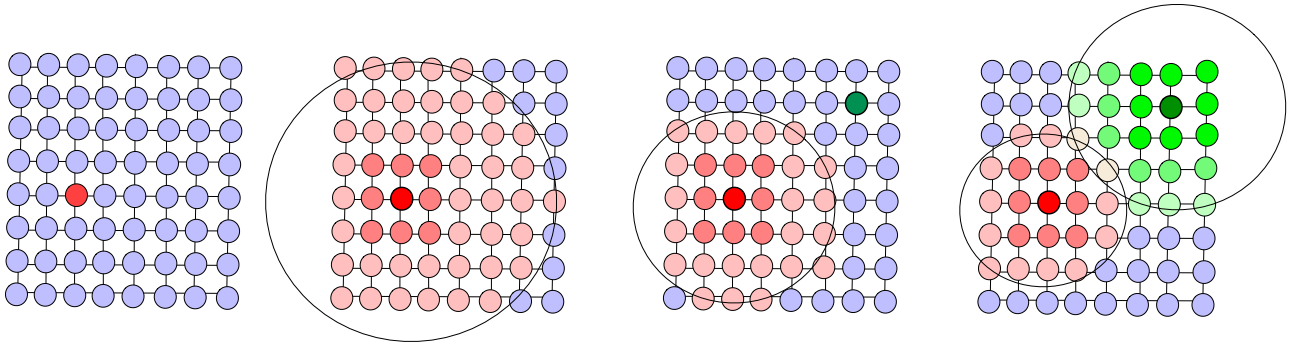


Figure 11-6: Output layer of SOM. From left to right, the first input record is processed, a BMU is identified with dark red, and with a large radius, neighbor nodes get its colors. Then, for the second input record, the green dot is identified. The red circle gets smaller (with an exponential decay rate), and this process continues for other records.

weight updates, and neighbor nodes apart from the BMU get lower weight updates. This process is visualized in Figure 11-6 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 11-6), and again, the 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 are processed.

One common approach to present SOM output is using hexagonal topological maps (Hex map), as shown in Figure 11-4. 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, it changes the representation of the data, and this result can be considered clustering. In particular, each input data point is assigned to the cluster represented by the winning neuron in the SOM.

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

Boltzmann Machines

In the neural networks we've studied previously, data flows in a specific direction, starting from the input layer and ending at the output layer. However, the Boltzmann Machine operates differently—it is directionless. This means that a neuron can function as an input at one moment and as an output at another, with no clear distinction between input and output layers. Each neuron can both send and receive signals to and from any other neuron in the network. Unlike previous networks, the Boltzmann Machine lacks a defined output layer, and all neurons, including those typically considered inputs, are interconnected.

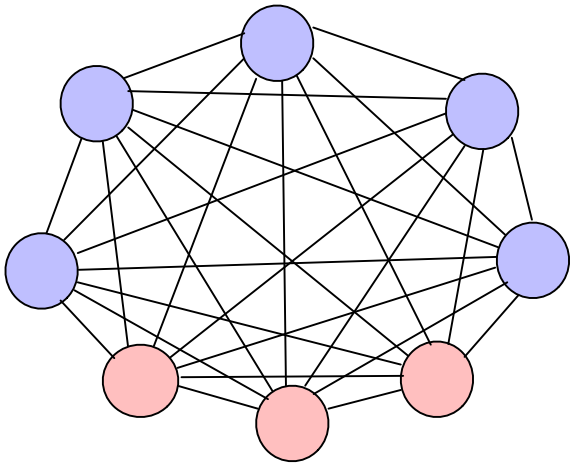


Figure 11-7: a Boltzmann machine, hidden nodes are presented in blue and visible nodes (input neurons) in red color.

Boltzmann machines, similar to SOM, have two types of layers. A Boltzmann machine includes a visible (observed) layer, and unlike SOM, it has a *hidden layer(s)*. The visible neurons are the information that we can measure, and the hidden layer neurons are information that we can not measure. Figure 11-7 presents a simple Boltzmann Machine.

The most prominent difference between Boltzmann Machine and other neural networks is that 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 odd, but the output of one epoch will be considered as the input of the next epoch. In other words, we can say neurons are typically considered units that update their states based on the states of other units they are connected to. The concept of distinct input and output does not apply as it would in feedforward networks. A unique characteristic of Boltzmann Machines is their generative nature, where the network's state is updated stochastically based on a probabilistic model, thereby generating new sample data from the learned distribution.

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 need to remember that while working with EBMs, we use the energy function instead of the cost function. The purpose of the energy function is to determine how likely each possible arrangement of the inputs is. 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 probability of a particular configuration is computed using the energy function and the Boltzmann distribution. When a system is in thermal equilibrium, its energy is usually

minimized, and the energy function reaches its lowest value. Thermal equilibrium is about achieving a distribution where transitions between states are stable (i.e., the rate of entering any given state equals the rate of leaving it).

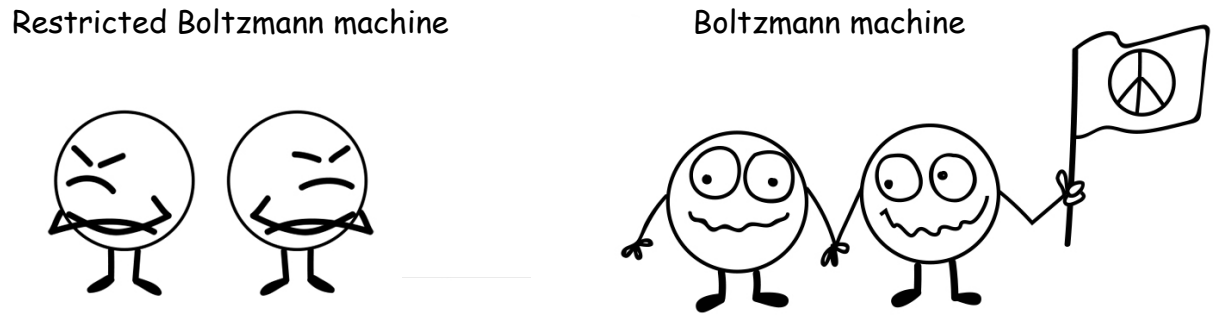
An energy function models the system, but a cost/objective/loss function is something we give to the algorithm as input to calculate the differences between the predicted value to the correct output. LeCun et al. [LeCun '06] stated that "*A loss score is minimized at the training (learning) phase, and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

Restricted Boltzmann Machine (RBM)

Restricted Boltzmann Machine (RBM) is an undirected 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 versions 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 layers (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 11-7, which is a Boltzmann machine to Figure 11-8, which is an RBM. These small differences between RBM and the Boltzmann machine make the energy function of the RBM much simpler than the Boltzmann machine's energy function.

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



Usually, we use a Binary RBM (Bernoulli RBM), its input in the visible layer is 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.

An RBM adjusts its weights and biases through multiple iterations of interaction between the visible and hidden layers. The *neurons in the hidden layer represent features that the model learns to capture from the data*. It's important to note that the RBM algorithm does not assign specific labels or meanings to these hidden neurons. However, by analyzing the activation

patterns of these neurons in response to various inputs, we can infer what features each neuron represents and selectively activate specific neurons based on these insights.

Let's do a brief review of what we have explained. RBM is an EBM model, and EBM models are generative models that learn the underlying distribution of the data (training set). Once the model has learned this distribution, it is capable of generating new data points that resemble the original data, effectively reproducing the characteristics of the training set's distribution. In other words, the neural networks that we have learned are identifying a mapping between input and output neurons. Still, RBM defines the probabilistic distribution instead of mapping between input and 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, 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*.

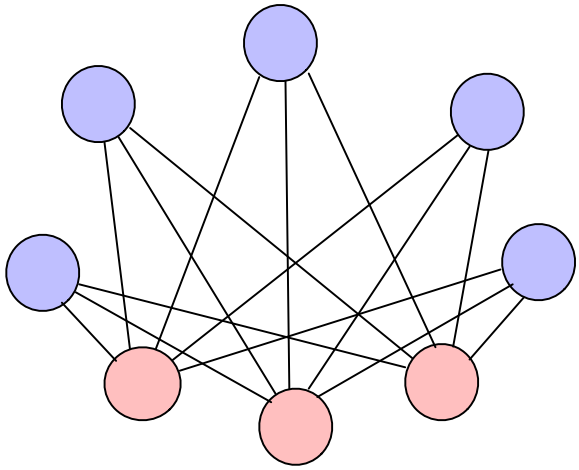


Figure 11-8: A sample Restricted Boltzmann Machine, hidden neurons are presented in blue and visible neurons (observation) in red. It is very similar to Figure 11-7; the only difference is that nodes are disconnected.

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). 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, and 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 in which each of its rows presents a visible neuron, and each of its columns presents a hidden neuron. Figure 11-9 visualizes both biases (presented as a and b) and weights on a simple RBM with three visible neurons and three hidden neurons, and it visualizes a forward pass and backward pass.

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 getting into thermal equilibrium and

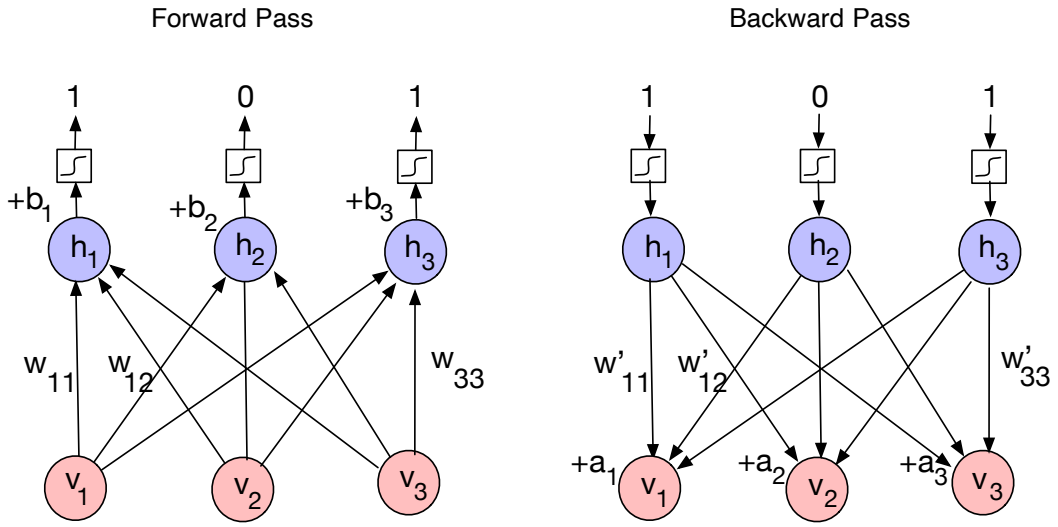


Figure 11-9: Visualizing the forward pass versus backward pass in RBM. The input and output of RBM is binary and its activation function is Sigmoid.

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) toward reaching thermal equilibrium.

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 computed 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 the problem again. 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 the joint distribution of $p(v,h)$, we can calculate conditional distributions $p(v\,|\,h)$, and $p(h\,|\,v)$, which is the basis of *Gibbs sampling*. By using Gibbs sampling (we will explain this sampling method in Chapter 16), we can approximate the $p(v,h)$, via conditional probabilities without directly computing the partition function Z.

Neuron Activation Probabilities

By using Gibbs sampling (we know v presents visible layers and h hidden layer), the probabilities of the visible layer and hidden layer in RBM can be decomposed as the following joint probabilities:

$$P(h\,|\,v) = \prod_i P(h_i\,|\,v)$$

$$P(v\,|\,h) = \prod_j P(v_j\,|\,h)$$

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 inactive with a value of 0. Therefore, given the visible vector v, the probability for a single hidden neuron h_j activated is written as follows:

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

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

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

RBM Training

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

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

However, since it includes Z and we can not calculate it, we use the energy of visible neurons $E(v)$ to approximate it, i.e., $E(v) = -\,log\ p(v)$. 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 that 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 (or actual), $\mathbb{E}_{data}$ minus the expectation provided by the model (or predicted), $\mathbb{E}_{model}$.

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

or

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

Here η is the learning rate. Recall from Chapter 3 that $\mathbb{E}$ denoted the expectation and $\mathbb{E}_{data}$ (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 $\mathbb{E}_{data}$(actual) and $\mathbb{E}_{model}$ (predicted) 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 with the same shape as the weight matrix` W.
for each v_t *in* $(1\ to\ T)$ {`#` T `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 $S \leftarrow S + v_t h^T$
}
$\mathbb{E}_{data} \leftarrow \frac{1}{T} S$

After calculating 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 *Contrastive Divergence (CD)* [Hinton '02]. The CD estimates the energy function's gradient using a set of model parameters and the training data as input. The CD makes a series of approximations to the true posterior distribution $p(h \mid v)$ over the hidden units using a *limited number* of Gibbs sampling steps.

Following is a simplified description of the CD algorithm. 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, the CD applies a small, subtle change that reduces the number of iterations. In summary, the CD algorithm continues to change visible and hidden neurons back and forth in fewer iterations than Gibbs sampling.

$Q = 0$ `# 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`

}

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

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:

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

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

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

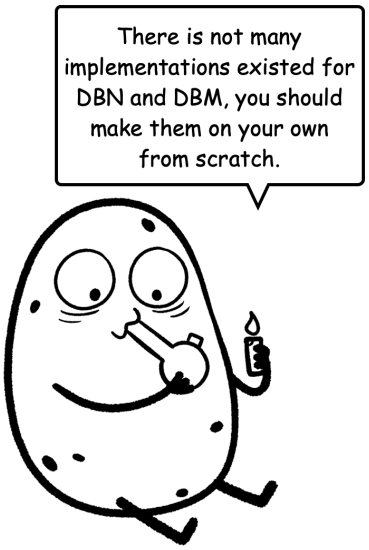
As we have explained, RBM is a historical algorithm. Through learning history, we cultivate an understanding of how later, more practical algorithms have been designed. Nevertheless, do not underestimate its capabilities. In some scenarios, to analyze data, we usually pass the dataset to SOM, RBM, and Autoencoder and decide which one provides the best accuracy.

Deep Belief Network and Deep Boltzmann Machine

Deep belief network (DBN) and Deep Boltzmann Machine (DBM) are two models inspired by RBM. DBN is composed of stacking multiple hidden layers, which can be trained independently (not necessarily using RBMs). DBM stacking RBM layers directly on top of each other. Each RBM's hidden layer becomes the visible layer for the next RBM in the stack (see Figure 11-10). 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 contains latent features that are better than random inputs. The hidden layer of the second RBM contains 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, and this step is used to correct weights in a supervised manner with Backpropagation. At the fine-tuning stage, the output layer is added (it is not present in Figure 11-10 because this figure focuses on pre-training only), and supervised learning is used to train the network with forward and backward propagations. Therefore, fine-tuning assigns the final weight and biases in a supervised manner.

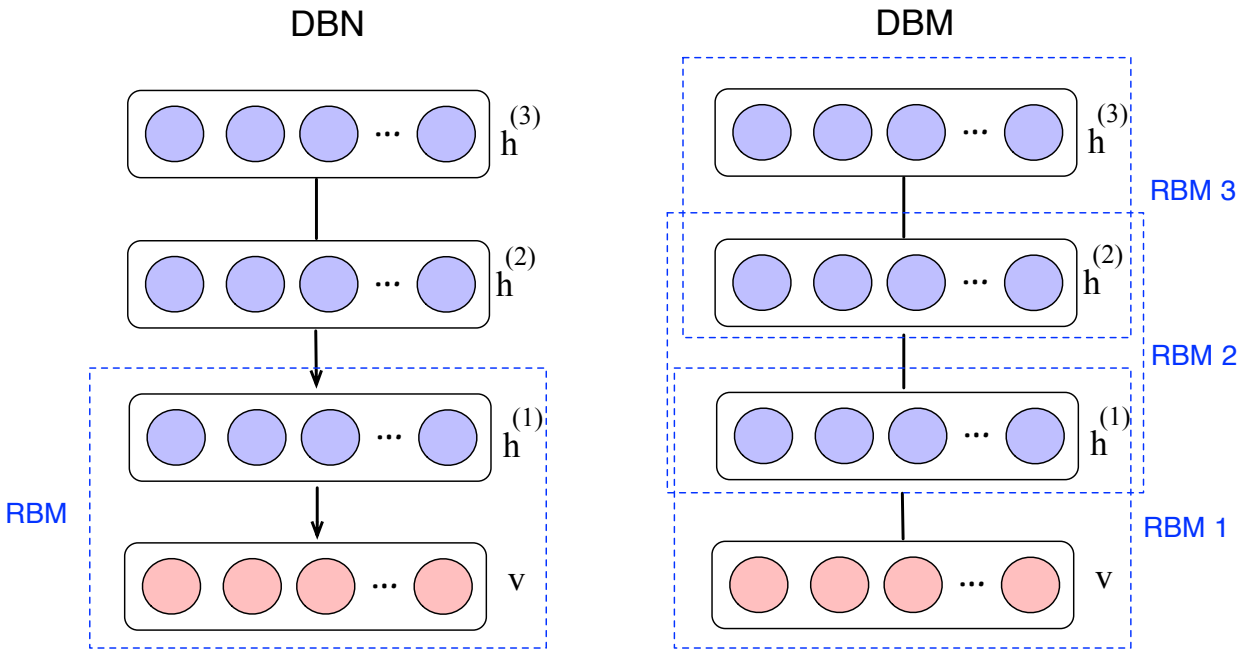


Figure 11-10: A sample DBN and DBM pre-training stage. The output of each RBM (hidden layer) is the visible layer of the next RBM. Arrows in this figure present the direction of the generative model (Gibbs sampling). For example, updated values in $h^{(1)}$ layer are used to reconstruct v layer, but in DBN, v layer can not be used to reconstruct $h^{(1)}$.

Keep in mind that both pre-training and fine-tuning stages in both DBN and DBM use a greedy approach to train and configure weights. Besides, in DBM, unlike DBNs, visible-to-hidden and hidden-to-hidden connections are trained simultaneously in an unsupervised manner.

By looking at Figure 11-10, we realize the arrowhead between the visible and the hidden layer. 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 DBN model can skip v to $h^{(1)}$, and how it can 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 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-tuning) start from the top layer until they reach the directed layers.

Why do we need to stack lots of RBM and train them (DBN or DBM)? Why not use a deep, feed-forward approach?

The motivation is that since RBM uses a contrastive divergence algorithm [Hinton ’95], there is no use of backpropagation in the pre-training stage while using a discriminative model. By not using backpropagation, it eliminates the vanishing or exploding gradient problem that existed at this stage. 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 11-10 However, unlike other deep learning algorithms, each layer learns to represent the data (input) passed to it from the previous layer. 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 learn from the input received from the previous layer.

To summarize their differences, DBN uses two phases of training, pretraining and fine-tuning, 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 operates better than DBN for learning complex models [Wang ’17], but, as always, we recommend experimenting with both architectures on your own and deciding which one to use.

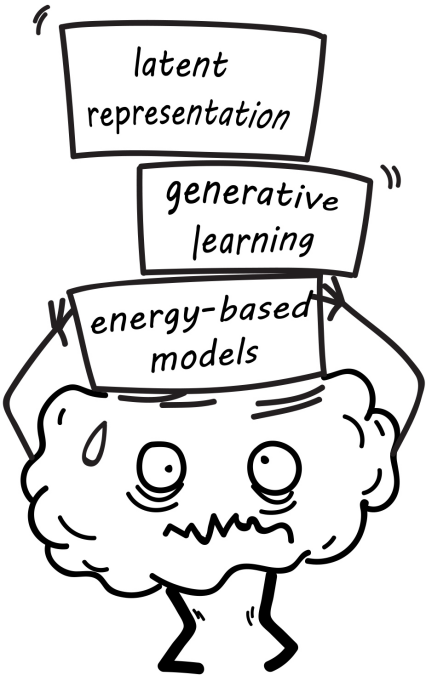
NOTES:

* A basic model of RBM is inspired by the 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 intend to learn more about Hopfield networks, you can

check Aggrawal's book [Aggarwal '18], which has a detailed explanation of 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 that we fed to it. RBM minimizes the energy or maximizes the probability. This feature of RBM enables both DBN and DBM to understand complex internal representations of the input data. Also, it can assist in improving 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 $\mathbb{E}_{model}$)*.

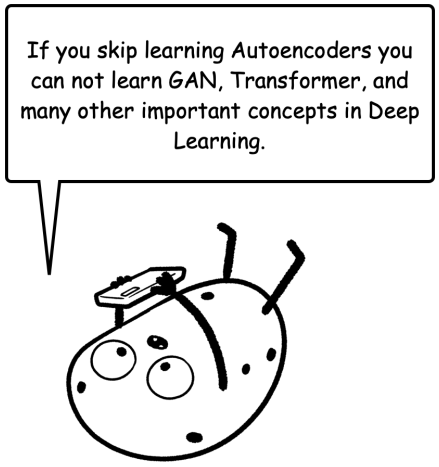


* DBN, DBM, and RBM are not widely used nowadays because Autoencoders outperform them. Nevertheless, learning them is useful for understanding the intuition behind the new models and algorithms. Some algorithms have not been in use for many years, and somebody extracts an idea from them and creates something useful, like Geoff Hinton, whose team created DBN, RBM, and tSNE and also applied backpropagation on DNN. Therefore, it is worth learning as much as we can, even though the algorithm seems not popular at the current time.

* 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 important to learn because they could inspire us while designing our algorithms. In this section, we discuss autoencoders, which, unlike SOM and RBM, are state-of-the-art data representation models.



Despite the superior accuracy of autoencoders due to resource consumption, we can not always use these complex neural network models, and using light algorithms such as SOM could resolve our need.

At the time of writing this chapter in 2021 and revising it back in 2024, autoencoders were one of the main components of most generative AI models. They are used in many applications, such as denoising data (e.g., watermark removal from watermarked images), better weight assignment in neural networks, image segmentation, text2image, synthetic video generation, and so forth.

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 noisy) while keeping the important features. The decoder uses latent space to reconstruct the data, similar to the original data, but without its non-descriptive features. In other words, autoencoders' outputs are not 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.

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 steps you say more than 10,000 steps). 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 "I ran more than 10,000 steps" and she compressed and decompressed it as you are an "athlete".

Take a look at Figure 11-11. We add noise to an image from the MNIST dataset and 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 unimportant features and only keeping the important ones. If there is no ‘4’ in the dataset, then the denoising was not successful.

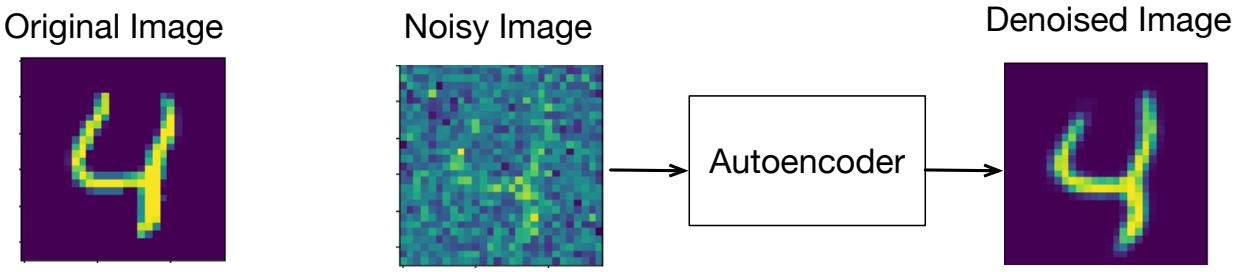


Figure 11-11: Autoencoder is used to Denoise a noisy image.

How do Autoencoders work?

The typical architecture of an autoencoder is presented in Figure 11-12. 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 dimensionality of the data (its features) is reduced. This reduction in the number of features has many useful applications and removes unnecessary features. An autoencoder has at least one hidden layer that includes the features used to encode the input data and change its representation (reducing its features).

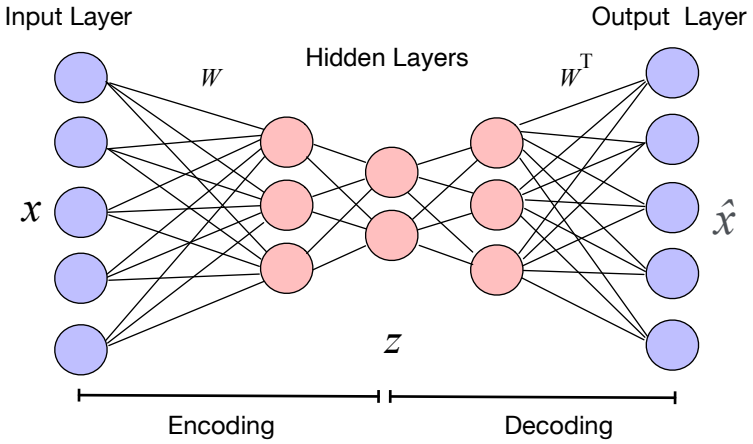


Figure 11-12: A simple one layer autoencoder which has only one layer of hidden layer.

Training an autoencoder 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), the 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 the 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 that is similar to the given input data.

We can interpret an autoencoder as a mathematical function. It has an encoder function $f(x)$, and 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 the 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 the 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 11-12. Autoencoders whose hidden layers have more neurons than input layer neurons are called *overcomplete autoencoders*, such as Figure 11-13. An autoencoder is called a *deep autoencoder* if it has more than one layer of hidden layers, such as Figure 11-12. Most of the time, we use deep autoencoders to solve our problems.

Autoencoders Can Cheat

The magic of an autoencoder occurs when it achieves an abstract representation of the input data, often facilitated by having a smaller number of activated neurons in the hidden layers. For example, the hidden layers in Figure 11-12 are smaller than the input layer. However, experiments have shown 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, the hidden layers typically contain more neurons than the input layers. On the other hand, having more neurons in the hidden layers than in the input layers might cause the network to simply copy the input to the output without any meaningful transformation (i.e., acting as an identity function). Figure 11-13 visualizes this problem. Such behavior, often referred to as cheating, should be avoided. Below, we describe three types of autoencoders that avoid this problem while still benefiting from having more hidden neurons than input neurons.

Autoencoder Types

Autoencoders are grouped into two main categories: regularized autoencoders and variational autoencoders. Common regularized autoencoders include sparse autoencoders, denoising autoencoders, and contractive autoencoders. They provide a regularization method to prevent the output layer from copying the input layer data (act as an identity function). To simplify the

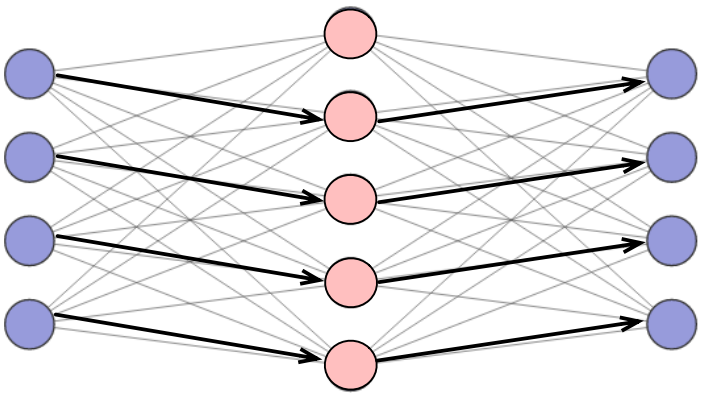


Figure 11-13: An over complete autoencoder that its hidden layer acts as identify function and input nodes are copied exactly as they are into the output node.

motivation behind using regularization in autoencoders, consider how we study for a mathematics exam. We need to learn concepts rather than memorize them because learning enables us to solve problems we haven't encountered before. In contrast, mere memorization is akin to copying and pasting answers, which is ineffective for addressing new or unfamiliar questions using our existing knowledge base.

After we are done with regularized autoencoders, we delve into a more advanced autoencoder, the variational autoencoder, which can generate data very close to the given input data [Kingma '13].

Sparse Autoencoder

Sparse autoencoders (SAE) [Makhzani '13, Ng '11] are typically overcomplete autoencoders, meaning their hidden layers are larger than the neurons of their input layer. When the number of hidden nodes exceeds that of the input, the autoencoder effectively increases the dimensionality of the data, which can lead to the extraction of more useful features. This characteristic is particularly beneficial for algorithms that aim to identify descriptive features of a dataset. Nevertheless, adding the extra hidden neurons requires certain measures to prevent the autoencoder from acting 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 11-14). In practice, it is similar to under-complete 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 $L(x, \hat{x}) + \Omega(f(x))$. Here, $L(x, \hat{x})$ is the reconstruction loss, and $\Omega(f(x))$ is the regularizer that receives the encoder output $f(x)$ as its input. Unlike regularizers, as explained in Chapter 8, these regularizers do not have weight decay. Instead, they measure the hidden layer activations for each training dataset and add a penalty to the loss function to penalize excessive changes in the activation.

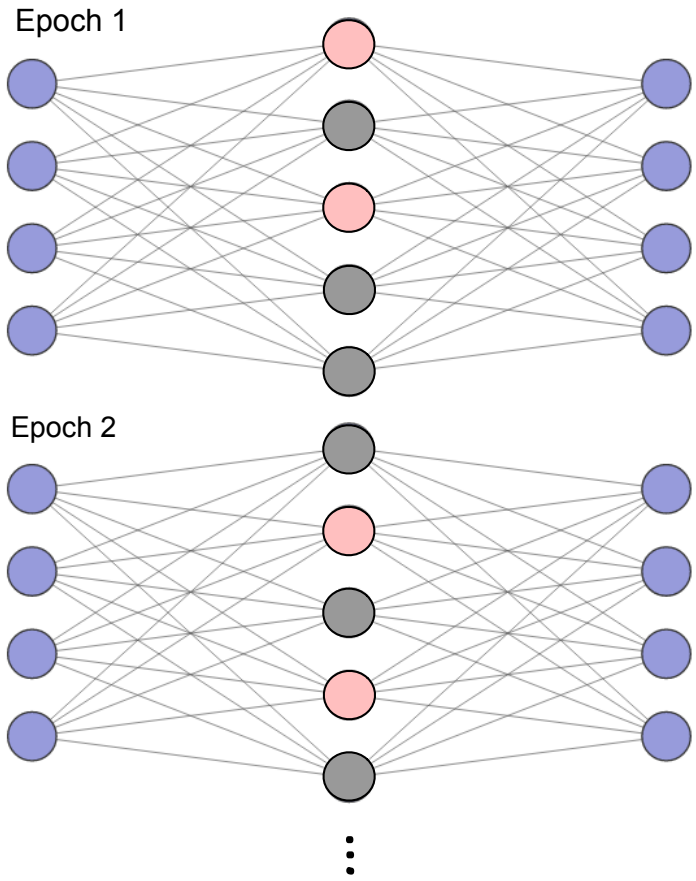


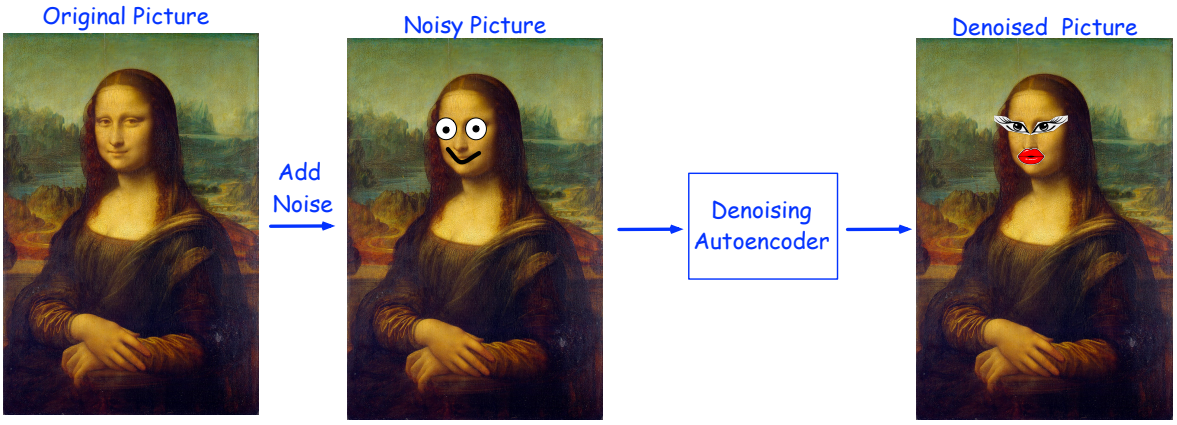
Figure 11-14: An example of the sparse autoencoder, which has two neurons of its hidden layer activated at each epoch. The grey neurons are regularized. Hence, their value is too small to have any significant impact on the output.

Two common methods to implement $\Omega(f(x))$ are L_1 regularization and KL-Divergence. The L_1 regularizer uses L_1 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 *desired* sparsity level. 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 11-15 is a denoising autoencoder. Similar to 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 the original input data, into the encoder. Many types of noise can be used to make an image noisy, like Gaussian noise, Perlin noise, etc. In Chapter 16, we will provide a more detailed description of different types of noise.

Figure 11-15 presents the architecture of the denoising autoencoder. Assuming the 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 shown in Figure 11-15, denoising autoencoders change the input neurons first, for example, setting some of them randomly to 0. It has a hyperparameter that specifies the percentage of neurons that should be randomly deactivated (turned off) during each pass by the denoising function. Since, in every pass, a random number of neurons are changed, this type of autoencoder is called a *stochastic autoencoder*.

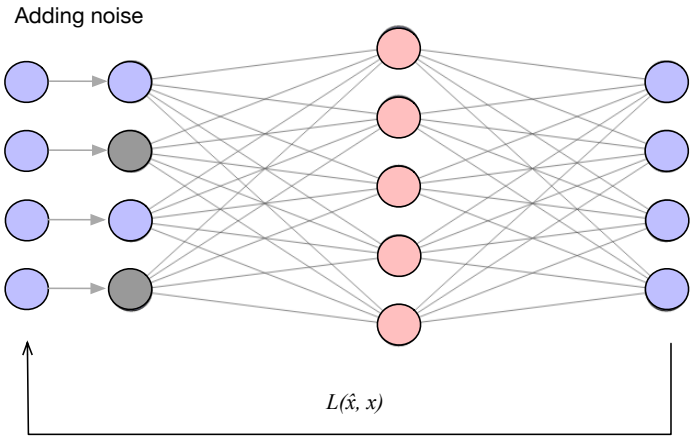


Figure 11-15: Example of denoising autoencoder. In each iteration the input layer with noise will be added to the network. Afterward, the cost function compares the output layer to the original input layer to tune weights and biases for the next iteration.

Contractive Autoencoder

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

It operates by incorporating a penalty into the 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 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 that controls the strength of the regularizer, $||J||_F$ is 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

---

[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.

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.* Each column refers to all hidden neurons' activation gradients with respect to one single input neuron. 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.

If the partial derivative is zero, 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.

A denoising autoencoder is straightforward to implement and does not require the computation of the Jacobian of hidden layers. However, Contractive autoencoders tend to be more stable than denoising autoencoders because they introduce penalties that measure the sensitivity of the hidden representation to small changes in the input. These penalties help maintain stability by ensuring the model's output does not vary significantly in response to minor input changes.

Stacked Autoencoder

Sometimes, the input dataset is complex, and one single autoencoder can not remove unnecessary information to provide 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* [Hinton '06, Bengio '07]. This pretraining is shortsighted, which means that each layer is trained based only on the output from the immediately preceding layer rather than on the outputs of all previous layers.

Why does greedy layer-wise training resolve 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, which can lead to inefficient training and poor convergence (issues often exacerbated by the vanishing gradient problem). Greedy layer-wise pretraining addresses this by training one hidden layer at a time. Each layer is trained using the output from the previous layer as input, effectively isolating the learning process for each layer.

For example, we have a small stacked autoencoder that gets input X, and by using the labeled data, it trains it, and the hidden layer of this autoencoder outputs Z_1. The next autoencoder receives Z_1 an input, and in the same fashion, it produces output Z_2. The third hidden layer receives Z_2 as input (not Z_1, only the immediate previous layer output). This sequential process repeats until the last layer produces Z_n , which is then fed into the output layer of the network.

Figure 11-16 presents a stacked autoencoder. A stacked autoencoder setup typically involves multiple autoencoders, each consisting of an encoder and a decoder. In a typical use-case for classification, only the encoder components might be utilized, and the last layer is usually a Softmax (or logistic regression), which has a randomized initial weight. Randomized initialized

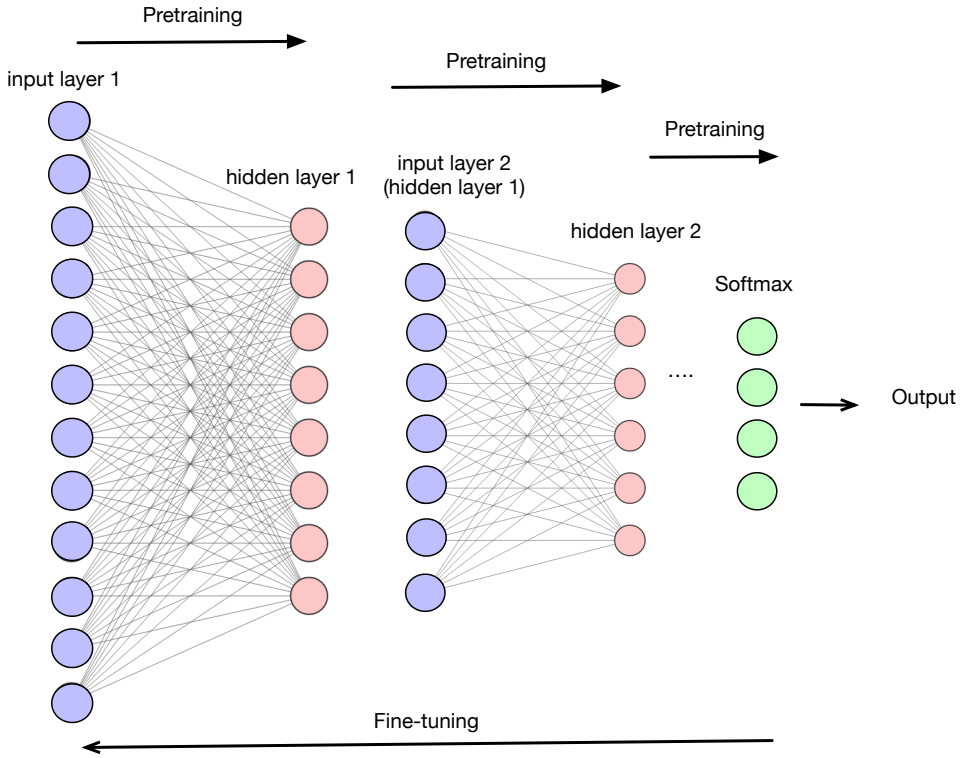
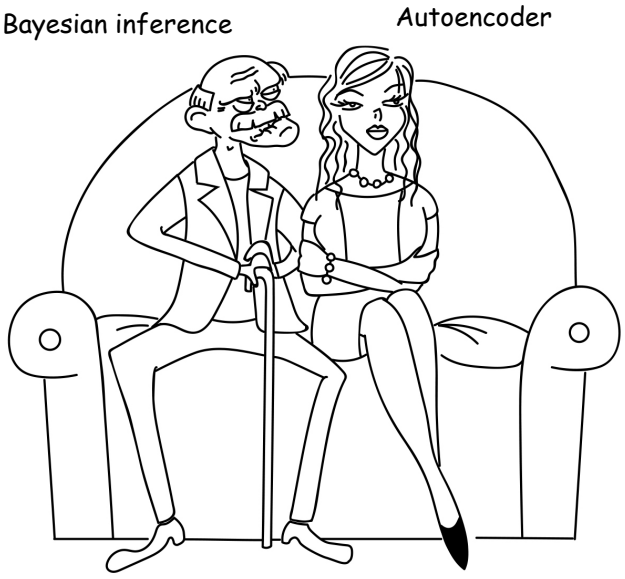


Figure 11-16: Stacked autoencoder with two autoencoders (this example has only an encoder, and there is no decoder), is followed by a logistic regression or a softmax layer for classification. Note that each encoder is trained separately in a greedy layer-wise manner using the labeled training data, and then fine-tuning stack them all together and train them.

weights here refer to the weights that must be corrected, and thus, we need to perform backpropagation. The backpropagation process adjusts the weights to approach optimal values, making the path of backpropagation less complex compared to other neural networks. This adjustment phase is known as *fine-tuning*. Fine-tuning is a supervised process, relying on labeled data to compute the errors necessary for backpropagation and to refine the model effectively. This step follows after all the hidden layers have been pre-trained.

Variational Auto-Encoder (VAE)

All autoencoders we have described until now are regularized autoencoders. 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, and in this book we only explained Bayesian inferences in different places, such as Naive Bayes in Chapter 9, probabilistically graphical model in Chapter 5, and LDA in Chapter 7.



Variational autoencoders (VAEs) are generative models that can be seen as an extension of latent variable models, which are similar in some respects to the expectation maximization (EM) algorithm (see Chapter 3). A VAE is specifically a type of latent variable model used to discover the underlying structure of a dataset containing hidden or unobserved variables. For example, if a document mentions terms like 'foreign policy', 'sanctions', and 'allies in the region', we might infer that its underlying topic is 'international politics'. This topic, not explicitly mentioned in the document, represents a latent variable. In contexts where the dataset is high-dimensional and noisy, it is common for classification algorithms to focus on extracting these latent features rather than attempting to utilize all existing features.

The Magic of Generative Models in Data Reconstruction

The VAE fits into the category of generative algorithms, and generative models are very successful in reconstructing synthetic 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. It leads to lots of success in generative AI applications.

How does such a thing happen? Assume we write an autoencoder to encode images of potatoes, carrots, and onions. A 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 bit and having 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 encode, the larger we 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 yellow potatoes. By transferring the data into the distribution, we can gain more flexibility in terms of learning the data's characteristics. In this example, assume an onion has a 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 comes 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) are used to perform this comparison.

How does VAE work?

The VAE extracts input features, but it does not assign a single value to each of the features (e.g., the face has an eyeglass: yes, the face has silicon in its lip: no). Instead, it presents them in a vector (e.g., the face has eyeglass: 0.7 probability, the face has silicon in its lip: 0.2 probability), and each extracted feature 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.

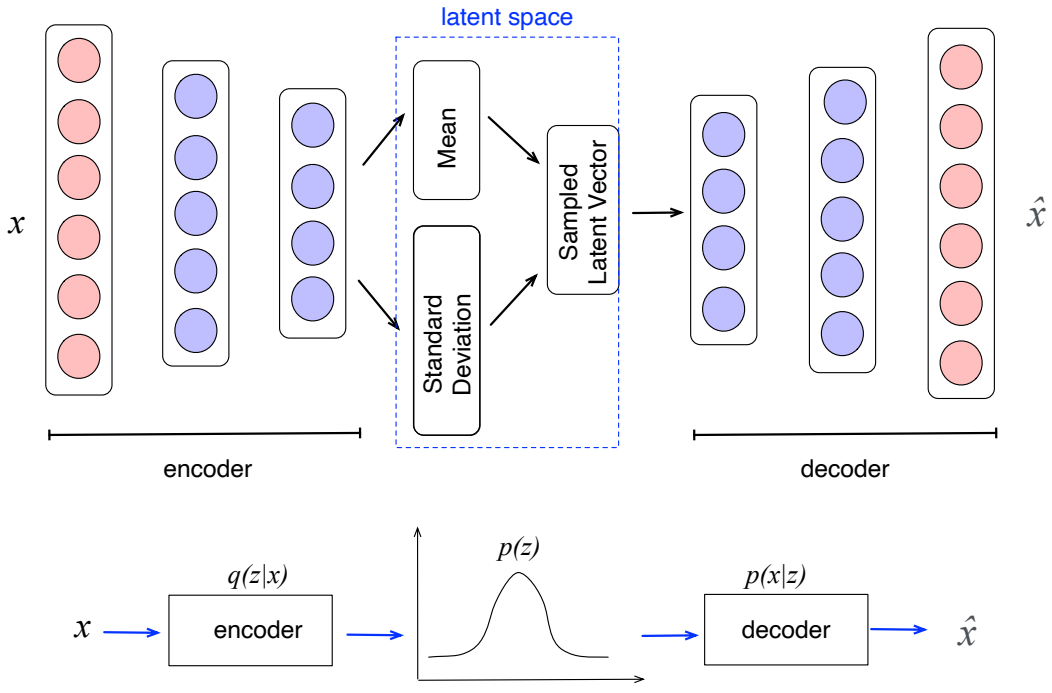


Figure 11-17: Schematic presentation of a Variational Autoencoder (VAE).

In particular, latent space, i.e., the output of the encoder, 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 of 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 11-17 visualizes the architecture of VAE.

Formally speaking, the output of the encoder in VAE is not a single vector z (e.g., a vector of one hot encoding). Instead, it computes $q(z|x)$, and results in a posterior that includes distribution parameters (e.g., mean and standard deviation if we deal with Gaussian distributions), i.e., $p(z)$. This posterior presents the PDF of z (Check Chapter 3 if you can't recall the PDF). In other words, VAE transforms the output of the encoder into *mean* and *variance*, which are parameters to describe a Gaussian distribution. Previously described autoencoders map input into a fixed-size vector, while VAEs map the input into a distribution $p(z)$, which is parameterized by the distribution parameters. The decoder then samples from the $p(z)$, and constructs $p(x|z)$. Afterward, $p(x|z)$ it is passed through an activation function, such as a 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, the final layer of the decoder typically uses a logit function to convert the outputs into parameters for a Bernoulli distribution, representing the probabilities of binary outcomes. These probabilities can then be sampled to generate binary data points $\hat{x}$. Figure 11-17 presents the VAE architecture.

For example, assume that the last layer of the encoder represents a two-dimensional Gaussian distribution[4]. We need two neurons to output the mean values (one for each dimension) and two additional neurons to output the log standard deviations. Thus, for a two-dimensional Gaussian, we have a total of four neurons, two for the means and two for the log standard deviations (refer to Chapter 3 for a review of Gaussian distribution parameters).

Another thing we need to remember is that variance values are always positive, and to increase the variance variability, the VAE uses the logarithm of the standard deviation, i.e., $log(\sigma)$, instead of the standard deviation itself. This is because the exponential of the logarithm will ensure that the standard deviation remains positive, and the logarithm function is capable of mapping a large domain of positive real numbers to a much smaller range, thus ensuring stability in the backpropagation process.

Assuming ϵ is a point sampled from a standard normal distribution, the encoding process converts the original input data into 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 the 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., $q(z|x) \rightarrow p(z)$.

---

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

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 its 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 between the latent posterior probability distribution, i.e., $q(z|x)$ and prior probability distribution, i.e., $p(z)$.

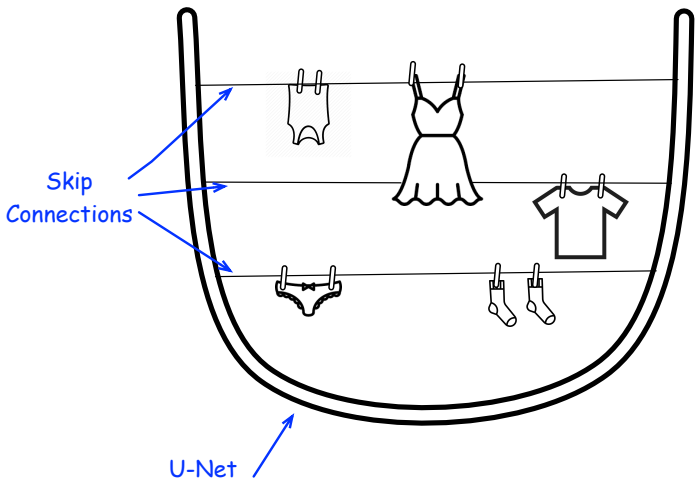
$$ELBO = \mathbb{E}[log\ p(x|z)] - D_{KL}(q(z|x)||p(z))$$

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 encoder distribution (encoder 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, backpropagation is used to minimize the cost function.

U-Net

We have briefly described two segmentation algorithms in Chapter 6, MSER and Watershed algorithms, several more segmentation algorithms will be described in Chapter 12. Nevertheless, it is a big sin not to explain the legendary U-Net model while discussing autoencoders.



One of the most popular image segmentation algorithms is U-Net[5], which has an autoencoder architecture. U-Net [Ronneberger '15] is a semantic segmentation model designed for

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

medical image segmentation, such as separating the 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 part (first in 2021 and revision in 2024), it is state-of-the-art semantic segmentation for medical images, and new models that provide a new segmentation usually compare their approach with U-Net, unless they train on a very large dataset, which we explain them later in Chapter 12. The authors of U-Net did not refer to it as an autoencoder, but it has a very similar architecture to autoencoders, and thus, we explain it in this section.

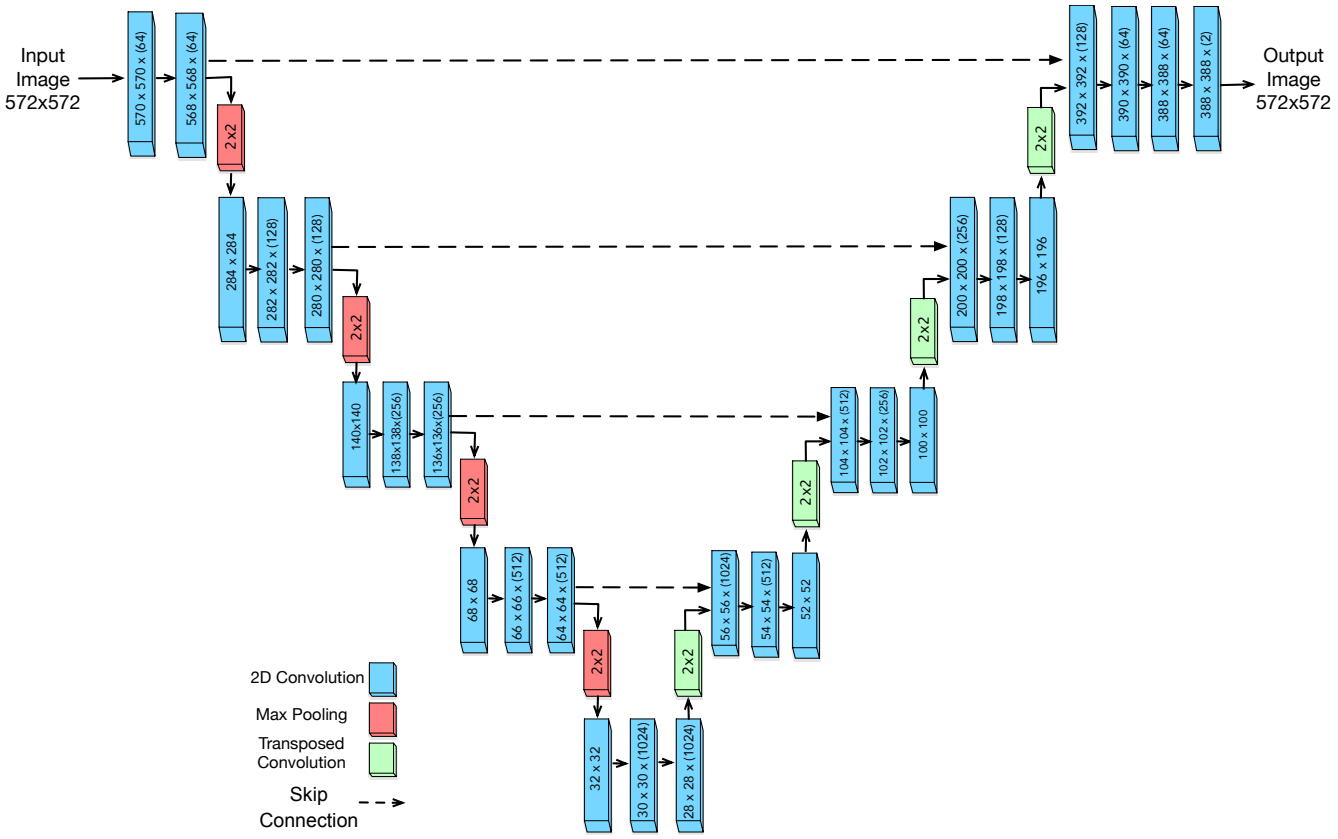


Figure 11-18: U-Net architecture. The left side presents the contraction/encoder, and the right side presents the extraction/decoder part of the network. The dotted arrows (copy and crop) present skip connections. The number of each channel is written on the top of its layer.

The original U-Net architecture is shown in Figure 11-18. It has a U-shape architecture, and it is composed of *contractor layers* (encoders) shown on the left side and *expansion layers* (decoders) shown on the right side of Figure 11-18. Note that not all implementations of U-Net should follow the standard U-Net numbers (or other neural network numbers) usually, we can customize these parameters based on our application needs.

U-Net moves 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 we can see in Figure 11-18, its contraction path (left side of U) *downsamples* the input image into a smaller size, and the expansion path (right side of U) *upsamples* the contracted data that it has received from the last layer of contraction path. In each downsampling stage of the contraction path, the number of feature channels doubled, and 2×2 max pooling with a stride of size 2 is 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 downsampled for the next level. In the next level, the convolution is applied again, and the result is 280×280, with 128 channels (the number of channels is doubled). This process continues in four levels of downsampling.

Next, the extractor path starts the process of upsampling, which we can read its numbers from the right side of Figure 11-18. The expansion path is a set of convolutional layers and transposed convolutions (instead of max pooling) to upsample the data. At each level of the expansion path, the number of feature channels will be halved.

Both the contraction path and expansion 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 (often referred to as the *where* information). The expansion path specifies *where* the information is located in the image, which means it handles the *localization of segments* in the image.

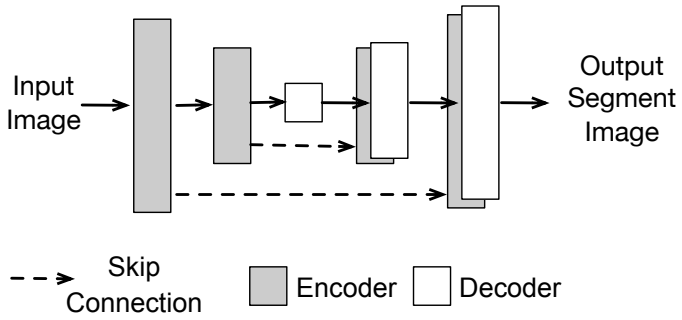


Figure 11-19: A simplified shape of U-Net architecture.

There is something interesting in U-Net architecture that makes it different from other autoencoders. Between the contraction path and expansion path, there are *skip connections*, marked with dotted lines in Figure 11-18 and Figure 11-19. Skip connections are responsible for cropping the image (by using padding) in the contraction path to match the expansion path image at the same level. By using skip connections from the contraction path, it combines the high-level 'what' information with the spatial 'where' information to localize features within the image accurately. Using the skip connection enables the network to mitigate the risk of vanishing gradients. Another advantage of using skip connections is that 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 causes a decrease in the accuracy of the train and test. Using a skip connection is a good solution to avoid both issues and keep them in mind while designing your neural network architecture. We can also employ the use of skip connection for

our 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 11-18. A more abstract shape of U-Net is designed by the author of Pix2Pix [Isola '17], is shown in Figure 11-19, and it is easy to memorize.

NOTES:

* Autoencoders are unsupervised learning, and they are adapted as self-supervised learning, where the data itself provides the structure for 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. While it's common for autoencoder architectures to be symmetric, the symmetry pertains to the network's architecture rather than the exact mirroring of input and output data. Non-linear activation functions, such as hyperbolic tangents, enable the network to capture complex relationships in the data, resulting in meaningful compressed representations.
* 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. There may be reservations about adopting this approach. Despite this, it's important to present diverse perspectives. In environments where computational resources are not as constrained, such as within well-resourced corporations, following Aggarwal's advice could indeed be beneficial for model design.
* 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 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 from acting as an identity function. Besides, it enforces the model to have fewer parameters. By reducing the number of parameters, the likelihood of overfitting will be reduced.
* An autoencoder that consists of multiple hidden layers is termed a *deep autoencoder*. These models are particularly effective for information retrieval tasks, including image search and matching, because they can compress high-dimensional data into a more manageable, lower-dimensional space while retaining important features. According to the universal approximation theorem, a neural network with sufficient depth and neuron count can

theoretically represent a wide variety of complex functions. This capability suggests that deep autoencoders have the potential to capture intricate patterns and relationships within the data, although their practical performance depends on factors such as network architecture, training procedures, and data availability.

* U-Net does not use any fully connected layer and 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 the *Generative Adversarial Network (GAN)*. In 2013, a model for an adversarial approach was proposed for animal behavior modeling by Li et al. [Li '13]. Afterward, in 2014, Goodfellow et al. [Goodfellow '14] introduced and implemented GAN, popularized among the data science community.

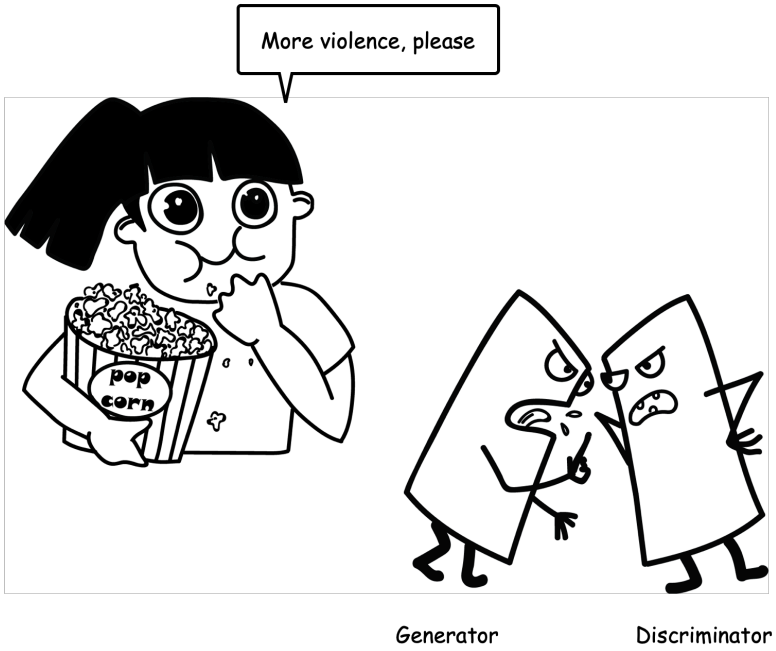
Figure 11-20: Synthetic image generated by NVIDIA's open source model, i.e., StyleGAN [Karras '19].

GAN employs generative models to construct fake (to be polite, we can say synthetic) data similar to the original input data. This is scary (for its deep fake applications), but an attractive approach, and it has 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 swapping and face image changes (e.g., aging the face or making it younger), image translation (e.g., converting day taken picture into night taken picture), etc.

GAN architecture is composed of two neural networks. One network is *Generator***,** and the other one is *Discriminator*. The generator network acts as a 'counterfeiter' that tries to generate synthetic data by using the original observed data. The generator network usually comprises a series of dense layers followed by transposed convolutional layers. 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 discriminator acts as a 'detective' and tries to identify the synthetic data from the original data. This process iteratively continues as a competition between the generator and the discriminator. In each iteration, the synthetic data gets more realistic because the generator learns to improve its model. Still, the discriminator also learns to recognize the synthetic data from the previous iteration, and it improves its recognition capability as well. Usually, the discriminator is a convolutional neural network that classifies the generated image.

The result of a GAN is synthetic data that is similar to the original data, like audio or images, that even humans can not recognize the differences from the original data. For example, Figure 11-20 presents a human face image constructed by StyleGAN [Karras '19]. The person in this figure does not exist in reality, but we can see that his face is very realistic, and we can not recognize that it is fake.



Both generator and discriminator networks compete with each other, and in every round, they get stronger. The generator is improving its synthetic data construction, and the discriminator encounters more synthetic data, which improves its discrimination capability. These two networks are playing a min-max game based on the objective function, which is cross entropy. Figure 11-21 presents the architecture of GAN.

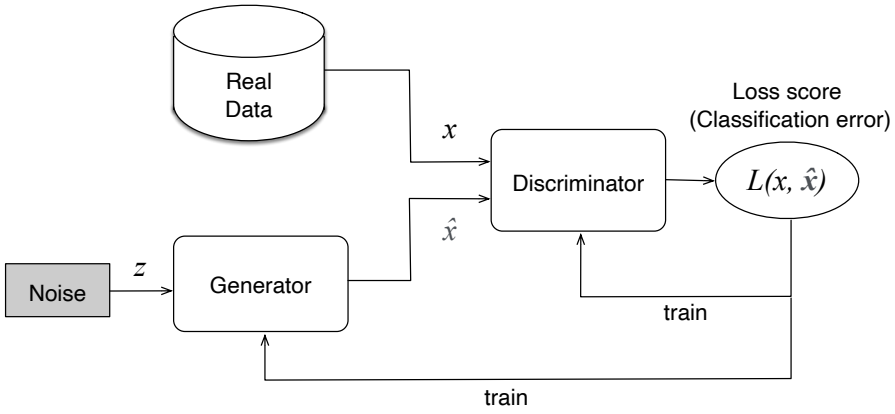


Figure 11-21: GAN architecture.

The dataset we use to train the GAN model is real data (x), and GAN uses this dataset to construct a synthetic version of this dataset ($\hat{x}$), with high similarity to the real data.

The generator performs the task of synthetic data construction and starts by getting a vector of random "noise" (z) and then refines it in each iteration with the backpropagation algorithm. When the network converges after several iterations, the random noise will be converted to data with a distribution similar to the real data.

The discriminator gets the real data (x) and the generator's output ($\hat{x}$) as input and tries 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 synthetic 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: $D(x) \approx 1$ (correctly identifies real data as real)
False Negative: $D(x) \approx 0$ (misclassifies real data as synthetic)
False Positive: $D(\hat{x}) \approx 1$ (misclassifies generated data as real)
True Negative: $D(\hat{x}) \approx 0$ (correctly identifies generated data as synthetic)

The result of the discriminator and generator is classification error (loss score), and backpropagation adjusts both generator and discriminator weights and biases. In particular, backpropagation is used by the discriminator to improve the accuracy of identifying real versus synthetic data. The backpropagation is used by the generator to increase the probability of the discriminator misclassifying synthetic 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 synthetic data.

Training GAN

The generator and discriminator are trained alternatively, and their training process is different. The discriminator training can be outlined in the following four steps:
1- Sample (random mini-batch) x from the real data.
2- Take a random noise z, and construct the output of the generator, 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.

The process generator training can be outlined in the following three steps:
1- Use a random noise z and feed it into the generator network to construct synthetic 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's weight and biases remain fixed while the algorithm is training the discriminator, and similarly, the discriminator's weight and biases remain fixed while the algorithm is training the generator.

A question might arise: When does the GAN iterative training stop? The GAN stops when both the discriminator and the generator reach a stage where neither the generator nor the discriminator can 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 synthetic examples that are not distinguishable from real data or (ii) the discriminator guess starts to get random for the generated synthetic data. In layman's terms, GAN stops when the discriminator is defeated and cannot recognize synthetic 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 results.

The training of other neural networks is an optimization problem, but in GAN, it is better to say its training is like a strategic game rather than an optimization for problem solving.

GAN Cost Function

The GAN training is a two-player game. One player is the Generator (one player) that tries to fool the discriminator by generating unreal data that looks like real data. The other player, the discriminator, tries to distinguish whether the data is real or not. 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 synthetic 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 the 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 synthetic data samples that come 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 synthetic data, i.e. $G(z)$.

In the cost function, the generator's objective is to minimize the loss score. The discriminator D tries to correctly classify real data x and generated data $G(z)$. Therefore, $D(x)$ should be close to 1 (indicating real data), and $D(G(z))$ should be close to 0 (indicating synthetic data). The generator G aims to fool the discriminator by generating data as realistic as possible. Thus, it wants $D(G(z))$ to be close to 1, indicating that the generated data is classified as real by the discriminator.

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

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

However, applying it directly to GANs requires adapting it to the context of discriminator and generator. Therefore we can write discriminator loss as follows:

$$L_D = -\frac{1}{n}\sum_{i=1}^{n}\left[\log D(x_i) + \log(1 - D(G(z_i)))\right]$$

Respectively, we can write generator loss as follows:

$$L_G = -\frac{1}{n}\sum_{i=1}^{n}\log D(G(z_i))$$

GAN Challenges

Training GAN is not trivial, and there are some major challenges associated with it. In the following, we list some common known challenges [Foster '19] of training GAN.

Loss Oscillation

Sometimes, the loss score for the generator and discriminator could oscillate and not converge. Some GAN models mitigate this oscillation, but it is still a common issue in training GAN. Oscillation in loss scores of the generator and the discriminator states that something in the GAN model is not correct, and thus, we should revise our GAN model.

Slow Convergence

Another common issue in GAN is 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 GPUs to train it, and this prevents small or medium size companies from being able to train a GAN from scratch. Usually, big corporation trains a GAN, and others use or fine-tune 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 synthetic 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 imitate real humans (the chatbot's conversational text is synthetic data, and the human conversational text is 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), it can fool the discriminator because the discriminator recognizes that 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 synthetic. 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 being 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 the discriminator and stick with those data points. Nevertheless, there are too few data points, which are not representative of the real data.

Uninformative Loss

Theoretically, a small loss score for a generator should translate to high-quality synthetic data. Nevertheless, the generator is compared with the discriminator, and sometimes, a decrease or an increase in generator loss is not correlated with the quality of the synthetic 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 many improvements in the GAN architecture to mitigate the described challenges. Later in this chapter, we explain some popular GAN architectures.

Evaluating GAN Result

GANs are a subset of generative models, and while some generative models are derived from the Maximum Likelihood Estimation (MLE) algorithm (see Chapter 4), GANs are distinct from MLE-based models in their approach and underlying theory. Unlike MLE-based models, the evaluation of GANs often relies on specific metrics that assess the quality and diversity of generated outputs rather than fitting to a known likelihood function. Two common metrics used to evaluate the results of GANs are the *Inception Score (IS)* and the *Fréchet Inception Distance (FID)*. These two methods are only used for synthetic image comparison, but the concepts behind these metrics can potentially be adapted for other types of data. Both the Inception Score and the Fréchet Inception Distance use the pre-trained Inception-v3 model.

Inception score (IS)

The IS [Salimans '16] was 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: 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, i.e., $KL(p(y|x) \,||\, p(y))$. In the last step, it exponentiates [6] the KL-Divergence score to make it easier 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, and thus we use the following equation.

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

We can see from this equation that 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 overall variety of images produced by the generator and how much each individual image looks like it belongs to a specific category.

Fréchet Inception Distance (FID)

IS does not take into account statistical characteristics (i.e., *mean* and *variance*) of 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 the IS score, and the 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 the 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 (paths or shapes) in a discrete manner, taking into account both the order and the positioning of data points along each curve. In Chapter 4, we explained DTW, which we use for time series similarity measurement, and in Chapter 3, we explained several distribution similarity measurements. Fréchet distance is generalizable to any shape and not only time series or distribution.

To understand the Fréchet distance, imagine a person walking a dog, where both start at the same point and move towards the same destination along different paths. They are not allowed to backtrack, meaning they can only move forward, though their paths might diverge within these constraints. The person and the dog can adjust their walking speeds independently but cannot reverse direction. The Fréchet distance between their paths is defined as the length of the shortest leash necessary for them to travel from start to finish without losing connection. This leash length represents the maximum distance between them at any point, assuming each can slow down or speed up as needed to minimize the distance along their paths, see Figure 11-22. In simple terms, the Fréchet distance specifies the minimum length of a dog leash required so that the owner and the dog can travel from the start to the finish points on their separate paths without losing connection.

---

[6] Exponentiating a number x means raising the constant e (Euler's number) to the power of x, resulting in e^x.

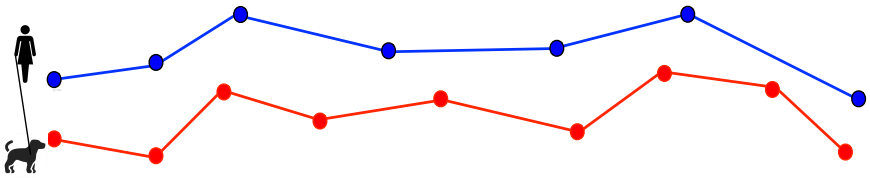


Figure 11-22: A toy example of two curves that have the same path but their trajectory is different.

Now that we understand Fréchet distance, let's get back to the FID score, unlike IS which uses the output (labels) of the Inception-v3 network, FID uses the activations from the last pooling layer (before the output layer) of the Inception-v3 network. This layer contains 2,048 activations, modeling each image as a vector of 2,048 features. Therefore, FID effectively uses the Inception-v3 network for feature extraction, allowing for a detailed comparison of the real and generated images based on these high-dimensional feature vectors.

Assuming the real data has a multidimensional Gaussian distribution of (μ and Σ), which μ presents the mean 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 generated images.

GAN Architectures

The GAN we have explained earlier in this chapter is called Min-Max GAN or vanilla GAN. It has very limited accuracy and 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. Later, after the revision in 2024, GAN is substituted by the Diffusion model, which we will explain later. The popularity of GAN has led to the introduction of too many GAN models, which are referred to as GAN Zoo. Therefore, we limit this section to a few well-known GAN architectures.

Note that most of the GAN architectures we describe here operate on image data we refer to the inputs more generally as data to acknowledge that these architectures may also be adaptable to other types of data, 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

[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.

assigned to real data, and this label helps the generator produce more realistic synthetic data. This label is incorporated in the cost function, and therefore, the Conditional GAN (CGAN) cost function is written as follows:

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

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

The baseline GAN cost is written as follows:

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

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

$$J_c(D, G) = \min_G \max_D \mathbb{E}_{x,\hat{x}}[log\ D(x, \hat{x})] + \mathbb{E}_{z,\hat{x}}[log(1 - D(G(z, \hat{x}), \hat{x})))]$$

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

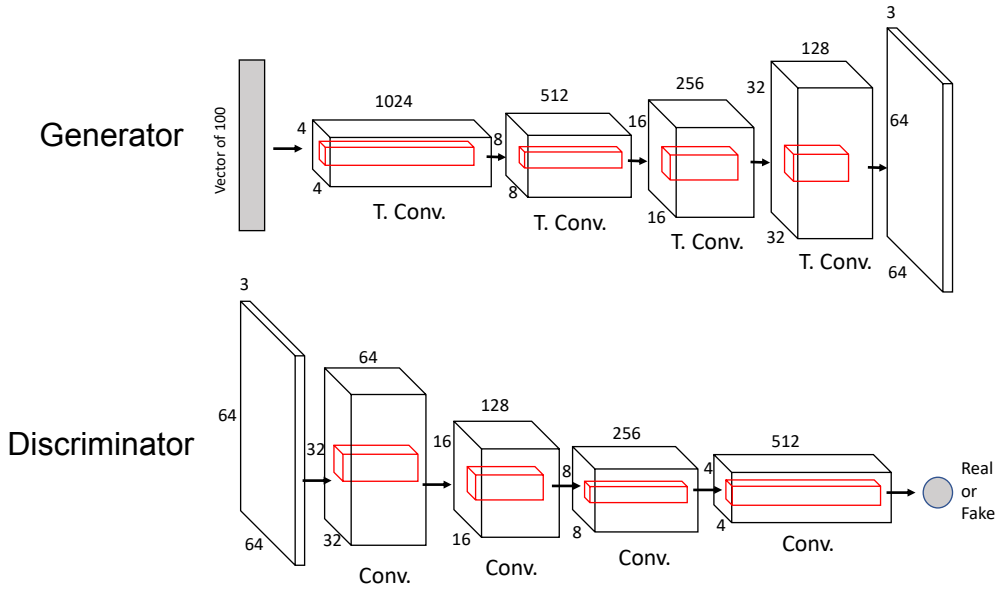
Deep Convolutional GAN (DCGAN)

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

In Chapter 10, we described 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 the generator and use a convolutional layer in the discriminator to classify the images.

As we can see in Figure 11-21, the generator starts with a vector of noise (in the original paper, the vector size is 100). It has four transposed convolutional layers, and each of them increases the height and width but decreases the channel, as is shown in Figure 11-23, in the original paper, the result is $64 \times 64 \times 3$. The discriminator includes four convolutional layers, and each convolutional layer decreases the height and width but increases the depth (the last layer of the 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 synthetic.

Both generator and discriminator 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:



DCGAN discriminator and generator architecture. T. Conv. stands for the transposed convolutional layer and Conv. Stands for the convolutional layer.

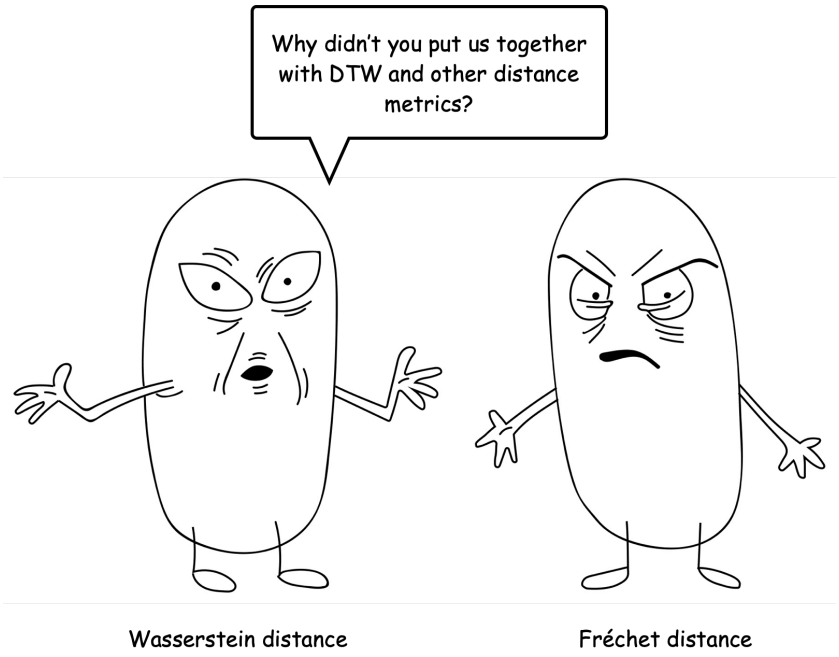
- It replaced any pooling layers with stride convolutions in the discriminator and *fractional-strided convolutions*[8] (often referred to as transposed convolutions) in the generator.
- Both the discriminator and the generator use Batch normalization.
- They use ReLU activation in the generator for all layers except for the output, which uses the hyperbolic tangent. They use LeakyReLU activation in the discriminator for all layers.
- The authors recommend initializing all weights 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 the discriminator and the generator, and it has a stable optimization process that mitigates the "mode collapse" and "vanishing gradient" problems. While using WGAN, we refer to the discriminator as a *critic* and not a discriminator. The reason is that the discriminator output is 0 or 1, which discriminates between real and synthetic, but in WGAN, it is a number because there is no logarithm, and the output is not a probability. Therefore, sigmoid activation is not applied to the output of the critic (discriminator). In other words, unlike traditional GANs, where the discriminator outputs a probability via a sigmoid activation, indicating whether inputs are real or synthetic, the WGAN critic provides a scalar score without using a sigmoid activation. This score is not limited to binary outcomes, reflecting how realistic or synthetic the critic deems the input rather than classifying it directly.

---

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



To understand WGAN, we should briefly describe two concepts: Wasserstein distance and continuity condition.

Wasserstein or Earth Mover Distance

The Wasserstein Distance, also known as the Kantorovich-Rubinstein [Rüschendorf '85] or Earth Mover's (EM) Distance, is a method used to compute the distance between two probability distributions. This distance measures how much work is required to transform one distribution into another, which is akin to calculating how to optimally relocate mass from one distribution to another. Imagine you have two piles of sand representing different distributions. The Wasserstein Distance calculates the minimum amount of work needed to reshape one pile to exactly resemble the other. This involves considering where each particle of sand is located and where it needs to go, which is why it's sometimes called the Earth Mover's Distance.

Figure 11-24 presents two distributions: P and Q. For the sake of simplicity, we present the area under each distribution as a set of numbered rectangles. The EM distance between these two distributions is calculated as the minimum cost of transporting the mass converting distribution P to distribution Q. Informally, we could 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 11-24 visualizes this figure in three steps for the sake of readability, and it does not cover all the details. Step 1, moves cubes 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 of this step is $2 \times d$. Step 3, moves cubes 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 cube 8 shifts one cube size less than d. The final cost of this simplified EM distance is:

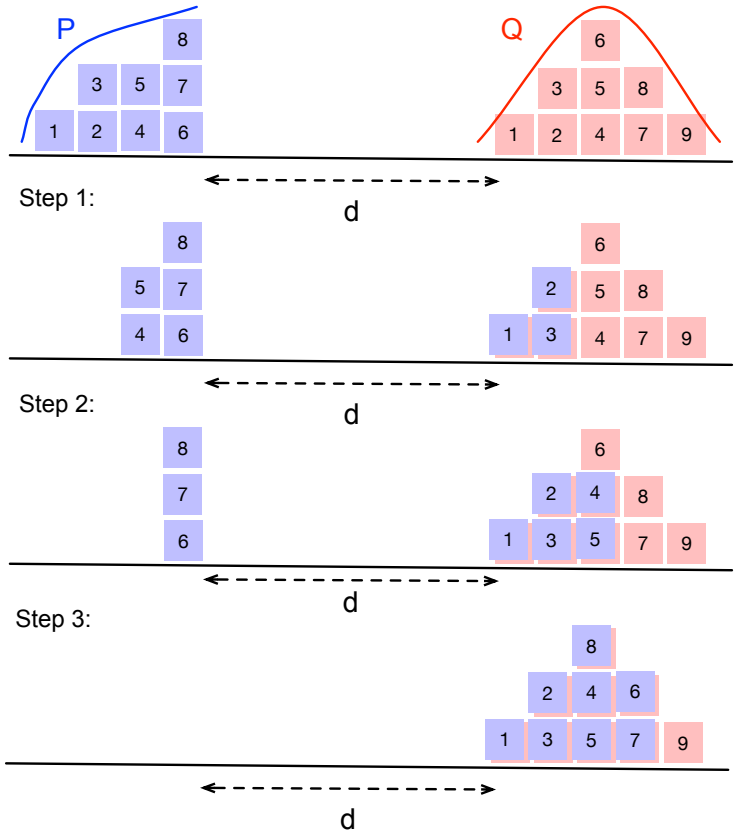


Figure 11-24: Two sample distributions, P and Q, that stay in distance of size *d* from each other. These steps try to simulate how the Earth Moving distance calculates the minimum distance between two distributions.

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

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 known as the *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:

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

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. The authors of the WGAN paper describe the Wasserstein distance for critic C and generator G as follows:

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

To recall what we have explained in Chapter 3, here $\mathbb{E}$ 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 a continuity condition. In layman's terms, a function is continuous if we can walk on it without lifting a pen from the paper (see Figure 11-25). 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, the function $f(x)$ is called continuous at point a if the following three conditions are met.

(i) $f(a)$ must exist

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

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

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

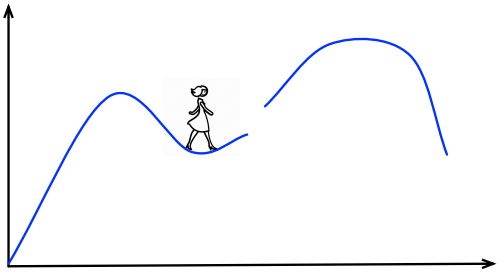


Figure 11-25: A simple illustration of non-continuous function, where we need to lift the pen to draw it, or the person walking on the function needs to jump to walk on it.

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 WGAN explained that if two distributions do not have overlap (disjoint), such as the example proposed in Figure 11-24, 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 is a number between 0 and 1. The mathematical explanation is fairly simple, and you can check the original paper for more details. 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 season, and good weather is too short here, so the author should go for his weekly half marathon and burn some of his excessive body fat.

Wasserstein is favored over KL-Divergence and JS-Divergence when distributions are disjoint, but computing the *inf* of $\gamma \in \Pi(P, Q)$ is intractable in EM distance. Therefore, WGAN authors propose a transformation using Kantorovich-Rubinstein duality[9] to the original Wasserstein distance, as follows:

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

[9] If you like to understand the details of this transformation this blogpost is helpful: www.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 $\mathbb{E}_{x\sim G}[f(x)]$[10] is the expected value of the generator function f (it is a 1-Lipschitz function, which its L is equal to 1), 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 the 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 remains to be explained.

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 the $[-c, c]$ 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 within the range of $[-c, c]$, and this guarantees its Lipschitz continuity.

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

WGAN with Gradient Penalty (WGAN-GP)

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

---

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

The gradient penalty is only applied to 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 follows:

$$W_{GP}(C,G) = \mathbb{E}_{x\sim C}[C(x)] - \mathbb{E}_{z\sim G}[C(G(z))] + \lambda \mathbb{E}_{x\sim P_x}[(||\nabla C(G(z))||_2 - 1)^2]$$

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 describes the interpolated data instead of real/synthetic data.

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

Pix2Pix

Explained GAN models are focused on generating images. One of the fascinating applications of GANs is the transformation of an image into another image. Pix2Pix [Isola '17] is a GAN model that uses a CGAN to translate an input image into a different image. For example, convert a satellite image into an outline map, color a sketch (See Figure 11-26), etc.

Pix2Pix, like CGAN, uses a conditional approach in its generator, but it conditions the entire input image rather than a label, making it suitable for tasks such as image-to-image translation. In addition to the adversarial loss, Pix2Pix incorporates a L_1 regularizer in its objective. This L_1 regularizer helps ensure that the generated images closely match the target images in terms of content and structure by penalizing the absolute differences between the output of the generator and the real target image. This feature makes Pix2Pix particularly effective for applications where precise alignment and details are crucial, such as photo editing or converting sketches to photographs. One popular application of Pix2Pix is to transfer sketches into photo realistic images. For example, we can see in Figure 11-26 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.

Figure 11-26: Using Pix2Pix trained on Pokemon image data to transfer a sketch we draw into photo realistic images. To construct this image, we use the following link https://zaidalyafeai.github.io/pix2pix.

If you do not read this section properly and skip our mathematical explanations, the monster of Figure 11-26 will hunt you in your dream and kiss your chick.

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. This architecture, which has skip layers, helps the output have the same structure as the input. In summary, Pix2Pix uses a U-Net architecture for its conditional GAN generator.

The discriminator of Pix2Pix is a *PatchGAN*. In short, a PatchGAN is patching an image into smaller patches (it is a disjoint sliding window we explained in Chapter 5), and instead of using a discriminator for the entire image, a discriminator is used for every single patch.

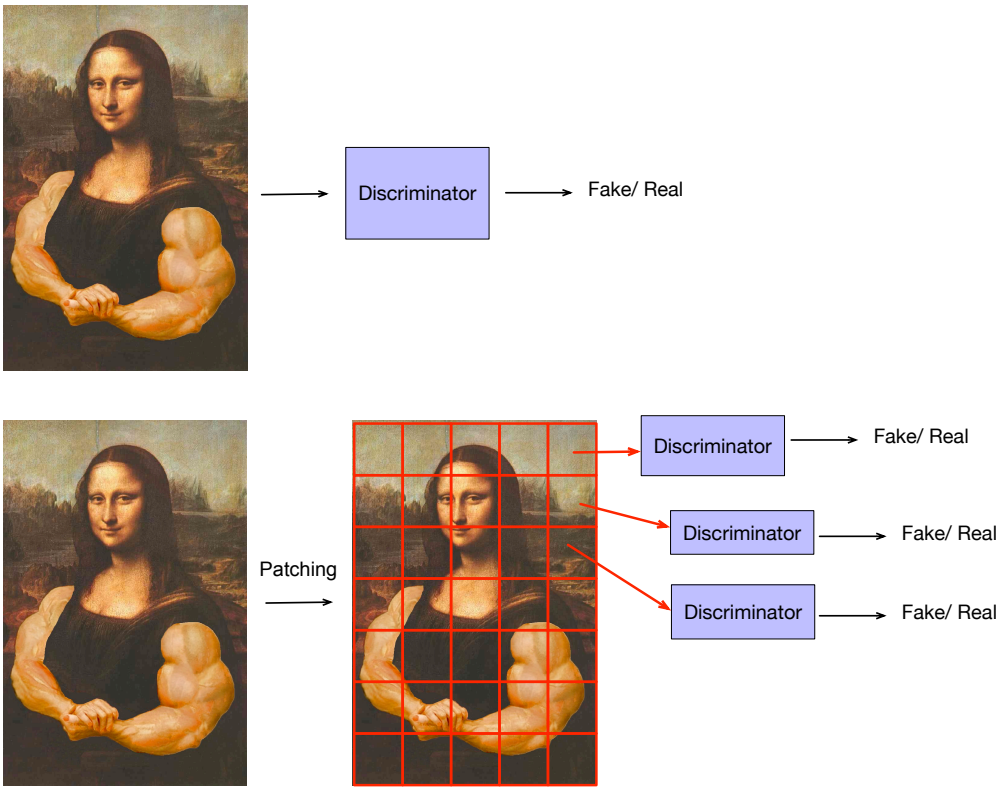


Figure 11-27: A vanilla GAN discriminator on top and PatchGAN discriminator on the bottom. The PatchGAN is feeding each patch into a discriminator, and at the end, averaging will be performed to decide if the image is fake or real.

Image source: https://www.deviantart.com/califjenni3/art/Mona-Lisa-the-Bodybuilder-62547732

Figure 11-27 visualizes the differences between vanilla GAN and PatchGAN. The PatchGAN Discriminator tries to classify each patch (70×70) in an image as real or synthetic. 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 (synthetic, 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 synthetic.

Both the generator and discriminator of Pix2Pix use Batch normalization after each convolutional layer and use the 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 core architecture.

CycleGAN

CycleGAN [Zhu '17] is a GAN architecture designed for image-to-image translation tasks where paired training data is not available. Unlike Pix2Pix, which requires precisely aligned pairs of source and target images, CycleGAN operates on unpaired sets of images from two different domains, such as horses and zebras. This is particularly useful when it's infeasible to obtain paired datasets that depict the exact same scene or object in two different styles or appearances. CycleGAN learns to translate an image from one domain to a corresponding image in another domain with reasonable accuracy by incorporating a *cycle consistency loss*. This loss ensures that an image translated from one domain to another can be translated back to the original domain, preserving content while adapting the style. We can see from Figure 11-28 that it can transform a zebra image into a horse image and transfer a landscape photo from winter into a photo in summer.

Figure 11-28: 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 $\hat{Y} = G(X)$.

In total, CycleGAN has two cost functions: cycle-consistency loss, which we have explained earlier, and adversarial loss.

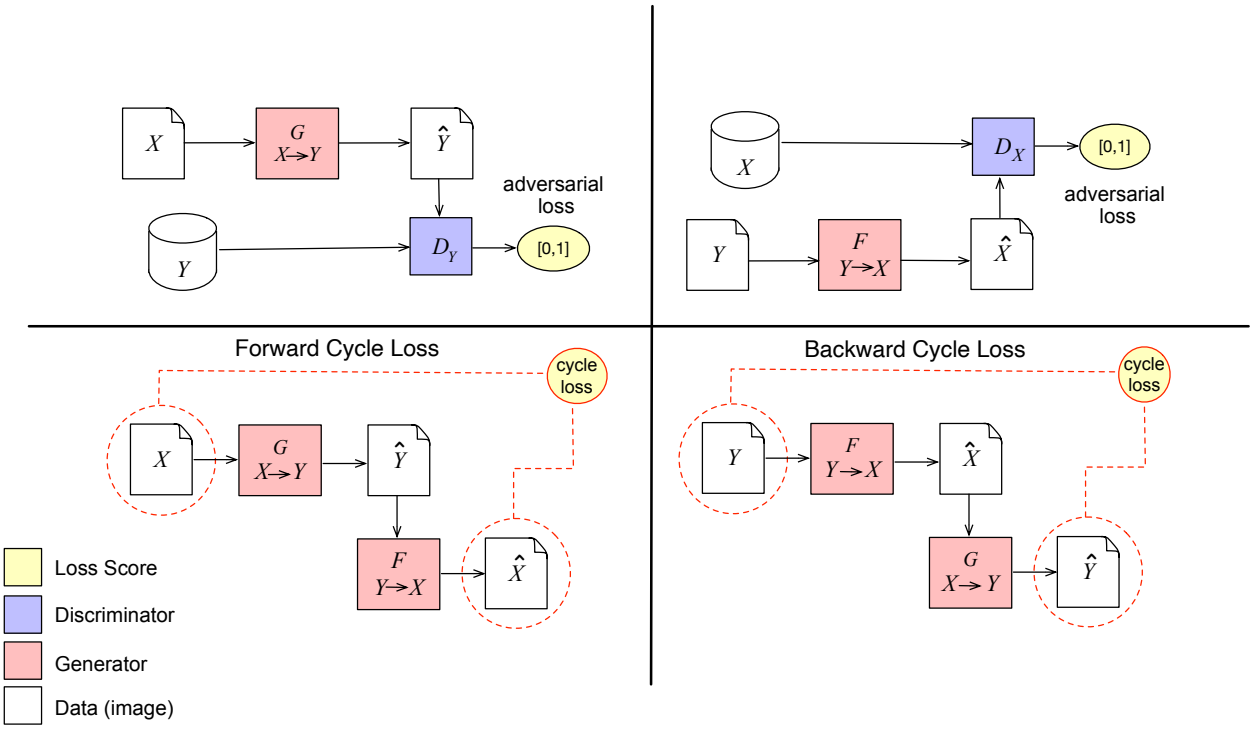


Figure 11-29: CycleGAN architecture is composed of two discriminators and two generators (We separate this figure into four subfigures for the sake of readability).

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.

Figure 11-29 presents the CycleGAN architecture. On 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., an image of a cat into a lion). Also, to maintain the cycle consistency, we need to have the image from domain Y translated back to domain X (e.g., a lion image back to a cat image). Therefore, two GAN networks are required, as shown at the top of Figure 11-29. 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 segment 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 stated before, there are two cycles that are shown in the bottom segments of Figure 11-29. They compare the translated image with the original images before translation and calculate the cycle loss.

For example, the 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 mapping $G : X \rightarrow Y$ 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 an image y. D_Y is the discriminator for domain Y, and tries to identify if the transformed data $G(x)$, which is shown with $\hat{y}$, and determines if $\hat{y}$ is real data or not (by comparing with y). For the second GAN ($F : Y \rightarrow X$), it is the same equation, just changing the input, output, and functions:

$$\min_{F} \max_{D_X} J(F, D_X, Y, X)$$

The cycle consistency cost is written as the sum of forward cycle cost and backward cycle loss, as follows:

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

The first part of this cost function calculates the forward cycle loss, and the second part calculates the backward cycle loss, here $\|\ldots\|_1$ 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. The authors of the paper set $\lambda = 10$ for their experiments.

Generator and Discriminator Networks

We describe the discriminator and generator architecture briefly. You also can check several open-source implementations, which could have 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: $64 \times 64 \times 256$) is fed into the transformer. The transformer is composed of a set of six Resnet block layers (we will explain them in Chapter 12). The decoder gets the image from the last layer of the transformer and reconstructs the image to its initial size by using three transposed convolutional layers. This means that the decoder gets the input of size $64 \times 64 \times 256$ and provides output of size $256 \times 256 \times 3$.

The CycleGAN discriminator uses the PatchGAN architecture [Isola '17], like Pix2Pix. By checking its original paper or open source implementations of CycleGAN, you can find more details about the details of architecture, such as activation functions, stride size, etc.

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 tired of reading about GAN architectures, take a look at Figure 11-20 and you can observe how realistic StyleGAN synthetic faces created are. That person does not exist in reality[11], and StyleGAN version 2 generated it.

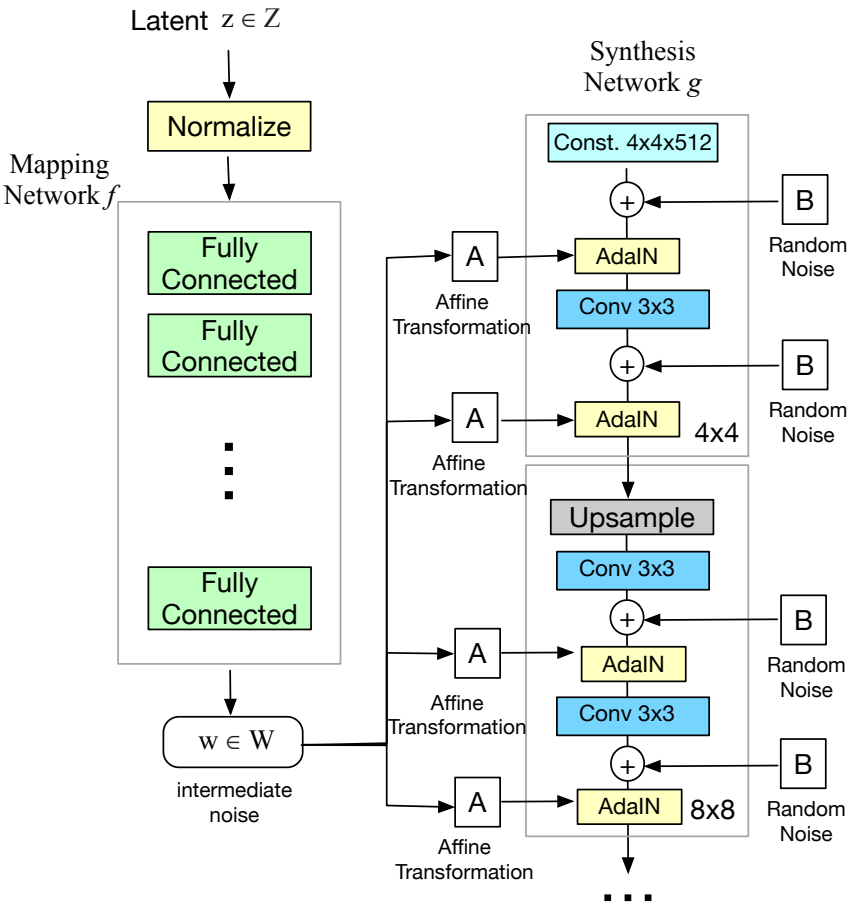


Figure 11-30: StyleGAN architecture.

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., generating only one particular

[11] 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

face).

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

StyleGAN generator has three architectural characteristics, which makes it different than vanilla GAN. In the following, we list these characteristics.

(i) Vanilla GAN generators start from a noise vector z and improve it until the generator provides synthetic data that is hard for the discriminator to distinguish from real data. The StyleGAN Generator does not start with plain noise. It has a mapping network architecture (a sequence of fully connected layers, as shown on the left side of Figure 11-30) that gets a set of plain noise vectors $z \in Z$ (not just one noise vector, more than one noise vector) and maps it to transforms it into an intermediate latent space, $w \in W$. This noise w is called an *intermediate noise*. This network is known as a mapping network and applies a non-linear transformation on noise z. The purpose of this mapping network is to create a more structured and manipulable representation of the latent variables.

(ii) StyleGAN introduces an Affine Transformation (Four A in Figure 11-30) and Adaptive Instance Normalization (AdaIN) [Huang '17], which incorporate *style (statistical characteristics of the image)* into the generated image (blue boxes in Figure 11-30 presents AdaIN).

(iii) After the w is transferred into the generator network, the model 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 11-30, we get something like Figure 11-31. In particular, two different types of noises are added to the generator: intermediate noise and random noise.

The generator got noise vector z with the size of 512 and passed it through eight fully connected (FC) layers of Multi Layer Perceptron, as we can see from the left side of Figure 11-30.

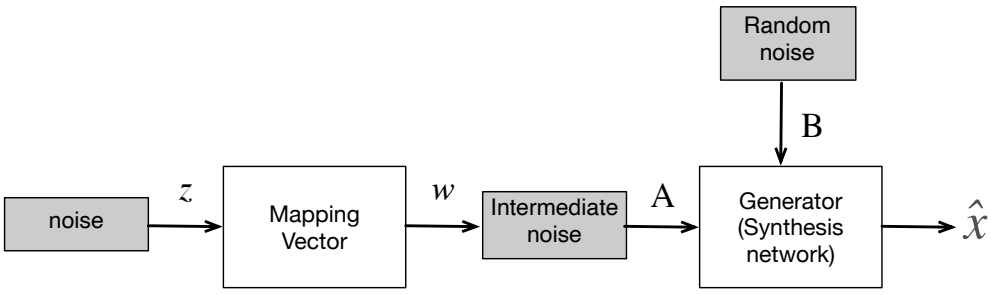


Figure 11-31: StyleGAN noises flow to the generator.

Why do the authors add eight additional Fully Connected layers? Mapping a noise vector into an intermediate noise 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, and nose). A noise vector has a Gaussian distribution, but features we need from output have a different 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 feature 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, with a dimensionality of 512. To summarize, the fully connected layers transform the initial noise vector z into a more disentangled intermediate latent representation w.

AdaIN injects the intermediate noise into the layers of the Generator. AdaIN normalization is a type of Instance Normalization (check Chapter 10) that adds style to the data. Style in this context refers to the statistical characteristics of the image components based on the incoming *intermediate noise (A)*. We can say, that 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 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.* AdaIN 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 the synthetic image. In Figure 11-30, y_s, y_b are presented as A. Therefore, we can say that AdaIN is responsible for transferring style information onto a 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 uses an approach called progressive growing, in which the generator gradually increases the quality of synthetic data. Progressive growing means that at the beginning layers, the model has a coarser component of a face, such as a nose, eyes, etc., and as it moves forward, it adds more details, such as hair color and eyebrow style.

The concept of progressive growing is proposed by Progressive GAN (ProGAN) architecture [Karras '17]. In progressive growing, the resolution grows slowly. The generator starts by constructing a tiny tensor with 4×4 pixel size, and the discriminator tries to analyze this image as well. Const. 4×4×512 in Figure 11-30 is a 4×4 grid where each cell contains a 512-dimensional vector. This tensor is the initial input to the StyleGAN generator network, which then progressively upsamples and transforms it to produce a high-resolution image. Then, the resolution of the image increases by simple upsampling (after 4×4 will be 8×8), and in parallel, it applies a convolution on the image until it ends up in an image with the size of 1024×1024 pixels. Other GAN architectures usually use transposed convolution for upsampling, but here, authors use a combination of nearest neighbor upsampling and convolution. On the right bottom side of each grey box in Figure 11-30, we 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. Briefly, this process is implemented with a combination of upsampling and convolutions. If you are interested in learning more details about progressive growing, you can check the ProGAN paper [Karras '17].

Style Mixing

One of the interesting characteristics of StyleGAN is mixing images and generating photo-realistic images based on the mixture of images. This is an amazing feature, and some recent mobile selfie apps have adopted it as well. Figure 11-32 presents an example of this feature.

The process of style mixing occurs by mixing several intermediate noises ('A' noises in Figure 11-30) along with stochastic noises ('B' noises in Figure 11-30). As we can see from Figure 11-30, different 'A' noises will be added to the generator (Synthesis network *g* Figure 11-30). 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 to the bottom layers of the generator).

Figure 11-32 presents two sources of images that are mixed (images from source 'A' and source 'B') and construct new images. As we 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'. We can see in the bottom row that the woman's face in source 'A' 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, which states that the number 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 we mentioned StyleGAN can control the fine-grained style of the generated image at the beginning of our explanation. It was not possible with the previously described GAN architectures.

Figure 11-32 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 generated image. This will be handled by stochastic noise (B noises), which is added before

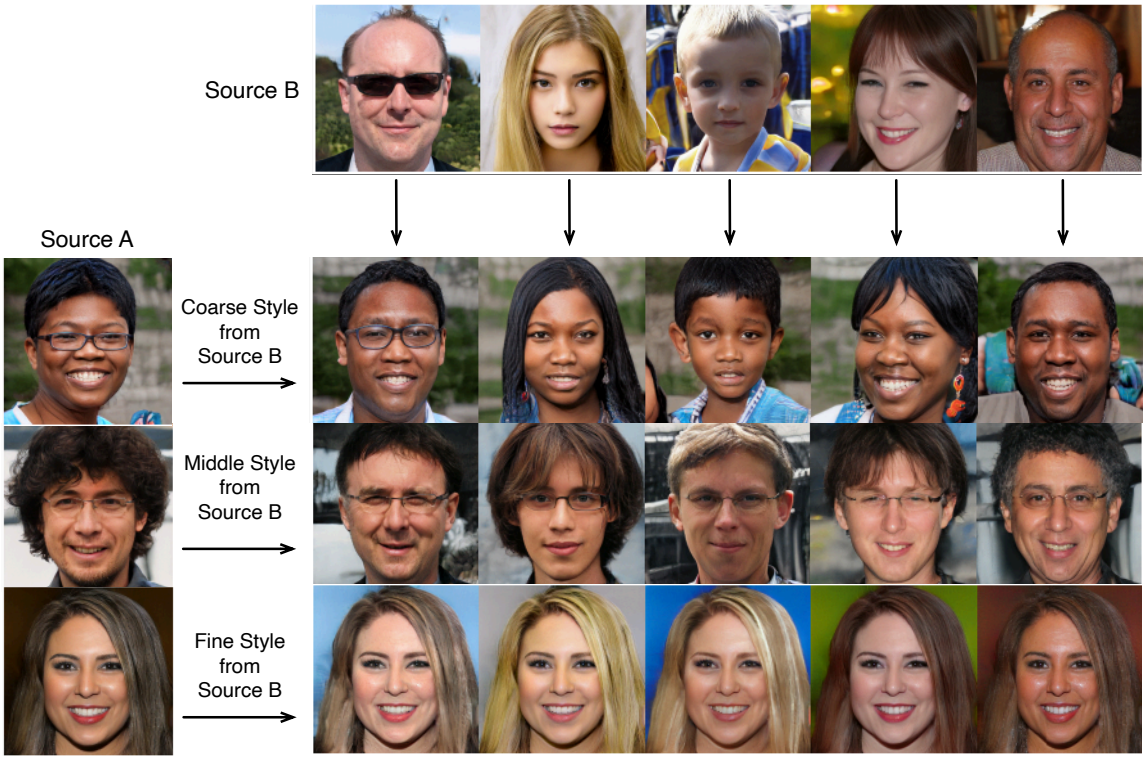


Figure 11-32: Mixing styles in StyleGAN: the impact of adding the intermediating noise vector into the Generator.

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 11-33 presents these features of StyleGAN, and we can see the impact of stochastic noise.

Figure 11-33: 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 section, NVIDIA is the largest GPU manufacturer, 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 the author of this book could afford 100 NVIDIA GPUs back in 2021, he definitely would

not write this book and enjoy a lavish life on a private island with tropical smoothies.

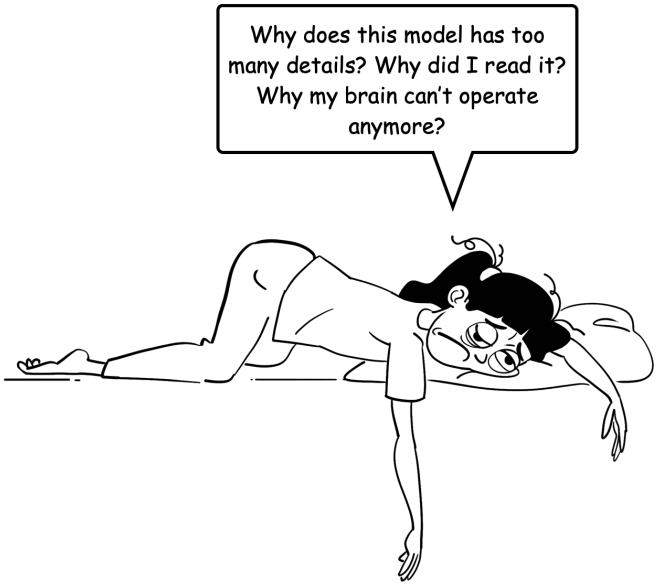
After the success of StyleGAN, the same group announced an even more accurate version of StyleGAN in 2020, which is known as StyleGAN2 [Karras '20], next to that StyleGAN3 is introduced as Alias-Free GAN [Karras '21]. Here, we briefly explain their differences from StyleGAN, for more information, you can check their papers.

Figure 11-34: Water droplet sign on generated images, because of AdaINST.

One known limitation of StyleGAN is the *water droplet effect* on the generated image (see Figure 11-34). 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 11-32; the region of teeth in all images is the same.

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 introducing StyleGAN2, the same team developed StyleGAN3, known as Alias-Free GAN [Karras '21]. While StyleGAN2 reduced but did not eliminate the fixed positioning of features in images, StyleGAN3 further addressed this issue. Although its FID score is similar to that of StyleGAN2, StyleGAN3 offers improved internal image representation by better handling transformations and rotations without texture sticking. This was achieved by addressing aliasing issues, which in previous models led to the generation of repetitive texture patterns due to inadequate suppression by upsampling filters. By eliminating aliasing, StyleGAN3 effectively resolved the texture sticking issue.



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 on arXiv.org, there are only more than 1,800 GAN models remaining that you can learn on your own[12].

All jokes aside, if you are young and a student, do not spend your valuable time making another GAN architecture; there are lots of interesting things that remain to be discovered and learned in artificial intelligence. At the time of revising this section, diffusion models are hype, but they also have a lifespan.

NOTES:

* There are claims that all generative models are derived from the maximum likelihood algorithm. The assertion, however, is an oversimplification, as many generative models employ a variety of foundational algorithms, such as variational inference, adversarial training, autoregressive models, etc.

* Training GAN is hard (still in late 2024), and again, it is something that we can not train on our own unless we have access to large GPU clusters of big corporations.

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

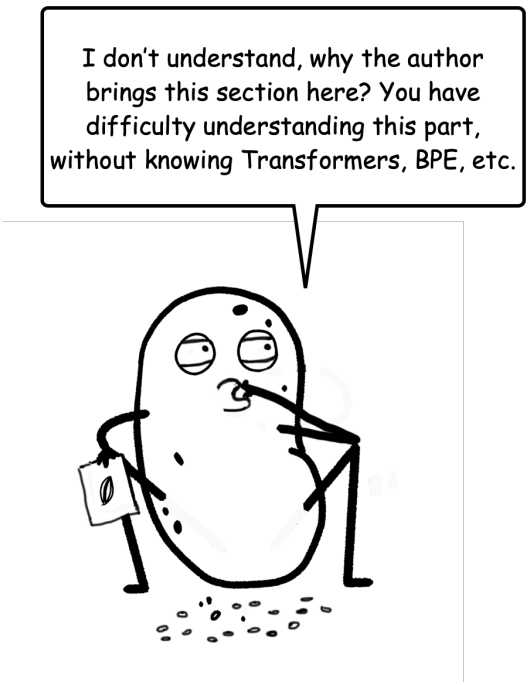
* While working with other autoencoders and other described neural network models, we have a single cost function that the optimizer needs to improve. However, GAN has two cost functions that need to compete with each other.
* To have a stable GAN, it is recommended not to mix synthetic and real data into a batch (stochastic gradient descent batch). Instead, use separate batches, and each batch only contains either synthetic or real data.
* Another approach to have a stable GAN is label smoothing, which means instead of 0 and 1 labels for synthetic and real data. Instead, the model can use probability that 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 synthetic video or synthetic audio generation, which is known as *deep fake,* and it is getting harder to distinguish if a real person saying something in a video is real or deep fake. There are interesting works where you can see the face of an old painting converted into a moving face and change his/her mimic, like Mona Lisa smiling, singing, or changing her facial expressions [Zakharov '19]. Some speculated that in the future, it will be possible to create an immortal version of a human in the digital world [Bell '10], known as digital immortality. By the time of reading this text, it is probably already implemented. Another extreme example is that criminals can simulate a synthetic digital avatar which we can distinguish if it is synthetic or real, e.g., swapping a face of a popular person with a porn actor in a porn movie.
* Only big and wealthy corporations such as NVIDIA, OpenAI, Google, and Meta can afford to build these large models. Even big universities fall behind the trend of making fancy generative AI models due to their limited GPUs.

Contrastive Representation Learning

There are different methods for preparing an object for comparison. Non-deep learning approaches use a similarity measurement (Chapter 4) to compare two digital objects. These approaches are used in the traditional image-matching, audio search, etc, but they are not flexible. For example, the face of a person who rotates his head in two different images is identified as two different images. Older approaches convert data into a lower dimension, such as applying PCA (Chapter 7) and then comparing them, and even after conversion, those methods are still inflexible.

Feature engineering in the context of deep learning is effectively managed through a process known as representation learning. This process involves mapping raw input data to a more useful representation or feature space through multiple layers of non-linear transformations. These transformations help convert the input data into a form where its features are more useful for machine learning algorithms, particularly for tasks referred to as *downstream tasks*. This automated feature discovery and transformation are key aspects of how deep learning models learn from data [Bengio '13]. A good representation of the data entails important features of the original data. For example, the word embedding methods described in Chapter 6, are

representation learning methods because they convert text data into vectors while maintaining the semantics of the text.



Contrastive Learning, a form of self-supervised learning also referred to as Contrastive Self-supervised Learning, involves using representation learning techniques to distinguish between similar and dissimilar data points. This approach, known as *Contrastive Representation Learning (CLR*), focuses on learning by comparison. CLR algorithms are particularly effective in learning robust features by contrasting pairs or sets of similar and dissimilar data, which enhances the model's ability to generalize from unlabelled data [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 model, and then we ask the model 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 contrastive learning and other supervised learning? While supervised learning typically processes individual data points or batches, assigning or predicting labels based on provided examples, Contrastive Learning differs by focusing on learning from the relationships between data points. Specifically, CLR algorithms enhance their learning by comparing and contrasting similar and dissimilar data points to construct robust representations. These representations do not directly classify objects into categories but rather enable the model to distinguish between similarities and differences effectively.

In the following, 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.

Contrastive Learning Loss Functions

Before we describe the 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:

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

In this equation, i specifies the index of the current data point. $Y = 1$ if two data points X_1 and X_2 are dissimilar, $Y = 0$ 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 that the G_W is a mapping function E_W between X_1 and X_2 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 enough amount of labels, which is common because acquiring labels for raw data is expensive.

Triplet Loss

The *triplet loss function* was proposed for the first time 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.

This loss function compares the baseline or anchor sample x, against the positive sample x^+ and a negative sample x^-. In this context, the positive sample means x and x^+ belongs to the same class (they have the same label), and the negative sample means 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^-. Triplet loss 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 that 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. There are 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 networks[13]. Unlike other neural networks, they do not train 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 a different angle (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 the 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 Siamese neural network commonly employs a convolutional neural network (CNN) architecture within each of its branches. This setup typically includes several convolutional layers followed by a few fully connected layers. Unlike traditional classification networks, Siamese networks do not use a softmax layer at the output because the primary task is not to classify inputs into categories but to assess their similarity. The output from each network branch, often called the encoding of the input data, represents a feature-rich representation used to measure similarity between pairs of inputs.

Training Siamese Network

First, the network should be fed with positive (1) and negative (0) pairs. For example, we fed the following dataset into a network: {(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 11-35 presents the architecture of training the 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). After obtaining the difference or distance between feature vectors, this value is often processed through one or more fully connected (FC) layers. The result of an FC layer(s) is given to a Sigmoid to transform the output into a probability and thus identify if x_1 and x_2 are similar or not (a threshold of 0.5 is used to make a binary decision).

---

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

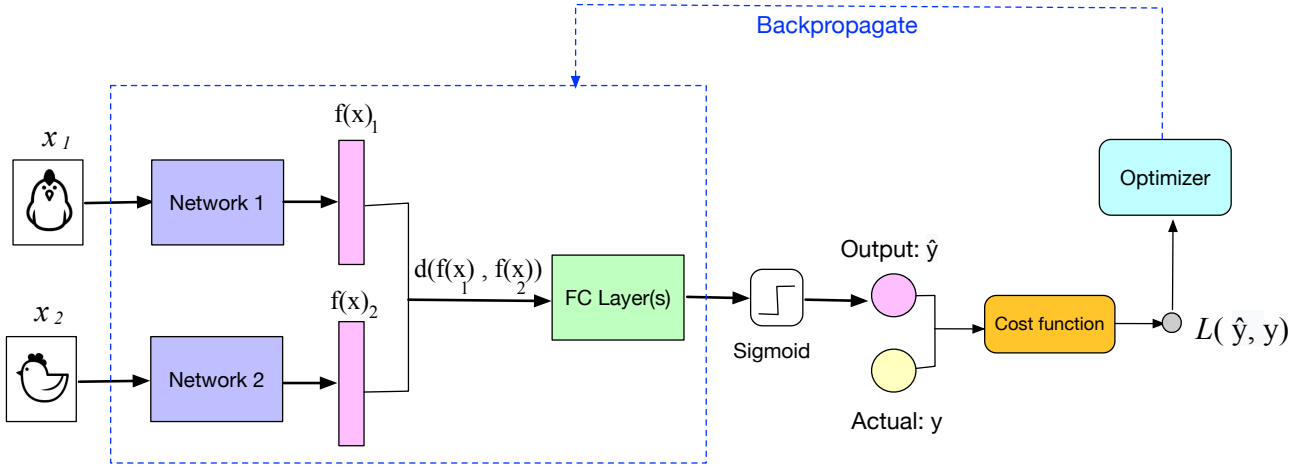


Figure 11-35: Training phase for Siamese network, in this example we show only one positive pair of examples, which is a chicken. In practice, the network will be trained for many positive and negative pairs.

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 both twin networks toward reducing the loss score.

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

Data Preparation for Training

To prepare data for training a contrastive model, we start by selecting a random input to serve as the anchor sample. We then choose additional samples that share the same label with the anchor to create positive pairs and samples with different labels to form negative pairs. This process ensures that there are corresponding positive and negative samples for every anchor. During the training process, every image in the dataset should have the chance to be used as an anchor, ensuring that the model learns from a diverse and representative set of comparisons. This selection process is typically repeated across training epochs to gurantee robust learning.

Testing Siamese Network

The test set will benefit from the model constructed in the training 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 images are passed 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

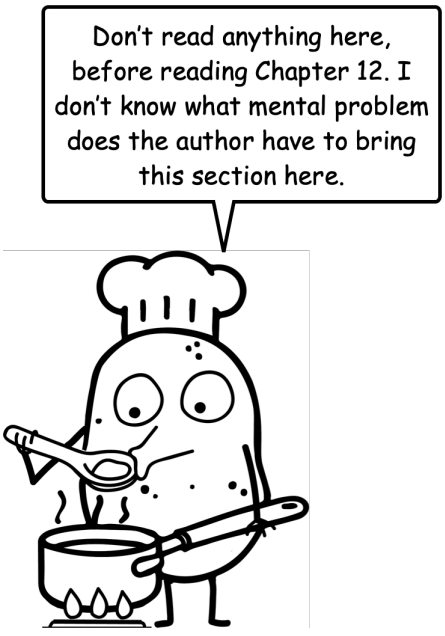
In the summer of 2022, when we revised this chapter to add Text-to-Image models, at that time, OpenAI released DALL-E, which was the top model at that time that got the text and constructed an image from the text, a.k.a. *text2image* or *text-to-image* models. Since the author of this book has a mental disorder and is obsessed with learning new models, we list some of the popular text-to-image models here that will be introduced and in use by the end of 2024.

You might have difficulties understanding the architecture of some of them, then, please come back here after reading Chapter 12. There, you will learn transformers and thus be able to understand these models. Before we explain models, we follow our tradition and explain some concepts used in text-to-image models.

Text-to-Image Related Concepts

In this section, we introduce key concepts and models used in text-to-image generation. Among these, diffusion models have gained significant popularity, surpassing GANs in synthetic data generation. Although we provide only a brief overview of diffusion models, this should not diminish their importance. Diffusion models have demonstrated remarkable success in text-to-image, text-to-audio, and even image-to-video generation.

Zero-Shot Learning



Zero-shot learning enables a model to label data it has not encountered during the training phase. However, based on given instructions and other auxiliary information, it generates a label for the unseen data not included 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 hasn't previously seen. 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 receives a text input. There are many descriptions of horses, and these texts are labeled as 'horse' during the training phase. Therefore, the model can easily label a text segment about a horse during its testing phase. Now, consider a text about a zebra. Although there is no label for 'zebra' provided, when it encounters a text about a zebra, it recognizes that the description is similar to that of a horse but with stripes (auxiliary information). By analyzing the text, it recognizes the horse and infers from other parts of the text that a horse with stripes is called a zebra. The model then assigns the label 'zebra' to that text based on this information.

In summary, zero-shot learning enables the model to assign labels to unseen data in the training phase. While large corporations have historically had access to such extensive datasets, there are now publicly available resources that can be used for zero-shot learning. Open-source contributions democratize access to the technology, enabling more entities beyond just large corporations to explore and implement zero-shot learning.

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 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 time 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 neural network models. RNN model connections can go forward and backward, but AR model connections are feedforward only. RNNs 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 to learn them to understand text-to-image generative models.

Diffusion Models

Another category of generative models is 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; recall that the Boltzmann machine we have explained earlier uses something similar.

For example, consider a spoonful of salt that we put into the water. The salt molecules move in the water (diffuses) until the salt concentration inside the water is equal at every point.

Diffusion models define a Markov chain (check Chapter 5), and in each Markov state, they gradually add noise (usually Gaussian noise, which will be explained in Chapter 16). A diffusion model starts with the original data, such as an image, and progressively adds noise over a series of steps until the image becomes indistinguishable from random noise, specifically following a Gaussian distribution with a mean of zero. Then, a neural network is trained to invert this process (get back from noise to the image). Once this network is trained, it can generate synthetic data with reasonable quality.

The process of adding noise gradually to data at each Markov state is referred to as a *diffusion process* or *forward process/path*. Then, the diffusion model tries to reconstruct the original data from the noise, which is known as the *reverse (denoising) path/process* or *generative path*. The process of Diffusion is presented in Figure 11-36. A trained diffusion model learns to generate new synthetic data by reversing a gradually added noise. The forward path is essential because it creates the training pairs (original data and noisy data) used in training the model. The model then learns how to reverse this process during the reverse path, where it learns to reconstruct the original data from the noisy data.

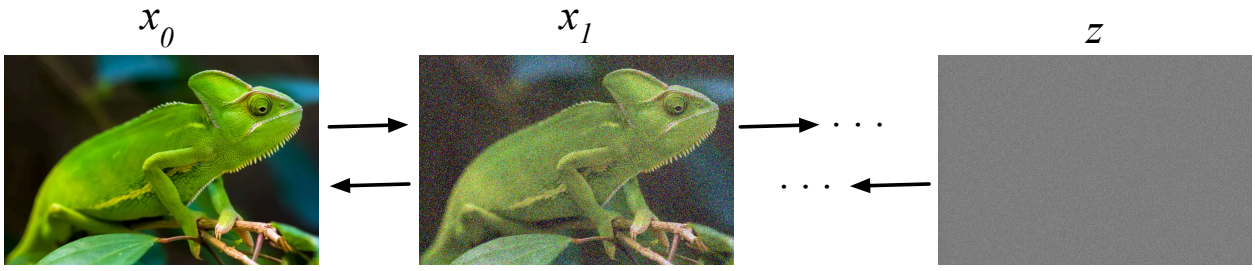


Figure 11-36: A diffusion model that gradually adds a noise at each Markov state (forward process), and then from *z* state, noise gradually reverse to reconstruct a data similar to the original data (reverse process).

Diffusion models differ from other generative approaches, including VAE or GAN models, which use latent spaces and adversarial dynamics. Diffusion models instead directly learn the transformation from noise to data, maintaining the original data dimensionality throughout the process. In other words, diffusion models are trained to de-noise the noise (z) and return to something similar to the original data (x_0).

Forward path: To formalize the forward process and reverse step, we assume the given input data as x_0 sampled from a real distribution, i.e., $x_0 \sim q(x)$. The forward process gradually adds a Gaussian noise $\mathcal{N}$ to the sampled data at T steps, which results in the sequence of noisy images, $x_1, x_2, \ldots x_T$, and x_T will be the pure Gaussian noise with a mean of 0 and variance of I. Assuming ϵ_t is sampled from this noise distribution, $\epsilon_t \sim \mathcal{N}(0,I)$, and α_t is a variance schedule, and decrease over time. Therefore, we can formalize the forward process as follows:

$$x_t = \sqrt{\alpha_t} x_{t-1} + \sqrt{1 - \alpha_t \epsilon_t}$$

In the context of the diffusion model, α_t is a sequence of values that dictate how much noise is added to the data at each step of the forward process. α_t is usually computed as a product of terms of the form $1 - \beta$, where β_t is a small value that defines the noise increment at each step. Usually α_t is computed as $\alpha_t = \Pi_{s=1}^{t}(1 - \beta_s)$.

Reverse path: The reverse process is where the diffusion model generates synthetic data by learning to reverse the noising process. This entails reconstructing the original data from its

noise-corrupted state step-by-step, based on learned predictions of the noise ϵ_t, that was added at each stage of the forward process. The reverse step can be formalized as follows:

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{1-\alpha_t}{\sqrt{1-\alpha_t}} \epsilon_\theta(x_t, t) \right)$$

Here, α_t is defined at the forward process, $\epsilon_\theta(x_t, t)$ is the predicted noise by the model at step t and parameterized by θ. In other words, θ is the trainable parameter of the neural network. The learning objective is to adjust θ so that $\epsilon_\theta(x_t, t)$ matches the actual noise, enabling effective reversal of the noise addition to reconstructing the original data.

Training: The training of the diffusion model involves optimizing θ that the model's predicted noise $\epsilon_\theta(x_t, t)$ matches the actual noise ϵ_t that was added during the forward process. This is usually achieved through an MSE between these two values:

$$L(\theta) = \frac{1}{N} \Sigma_{i=1}^{N} \| \epsilon_t - \epsilon_\theta(x_t, t) \|_2^2$$

Here, $L(\theta)$ represents the loss as a function of the model parameters θ and N represents the number of training examples. We have explained that $||.||_2$ means L_2 norm.

The diffusion process can also be seen from an energy-based perspective, where the model learns an energy function that assigns low energy to data-like samples and high energy to non-data-like samples. The training of diffusion models involves learning the reverse of the diffusion process. This is typically achieved by training the model to predict the noise added at each step of the forward process to undo the diffusion.

A common choice for the neural network architecture in diffusion models is the U-Net, which consists of a series of downsampling layers followed by upsampling layers, with skip connections between corresponding layers.

Inpainting and Outpainting

Inpainting generates or fills in content within a user-defined mask in a specific area of an image, typically to alter or complete parts of the image based on the surrounding context. Outpainting extends the existing content of an image beyond its original boundaries, using the provided prompt and context to generate coherent, expanded visuals. These features are integrated into image editing and image restoration tools.

CLIP (Contrastive Language–Image Pre-training)

CLIP [Radford '21] is a model that can understand images in the context of textual descriptions, proposed by OpenAI in 2021. In layman's terms, CLIP can match the given image to a range of textual descriptions, facilitating zero-shot learning. It is a very popular model and is used in many text-to-image models.

The authors explain 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

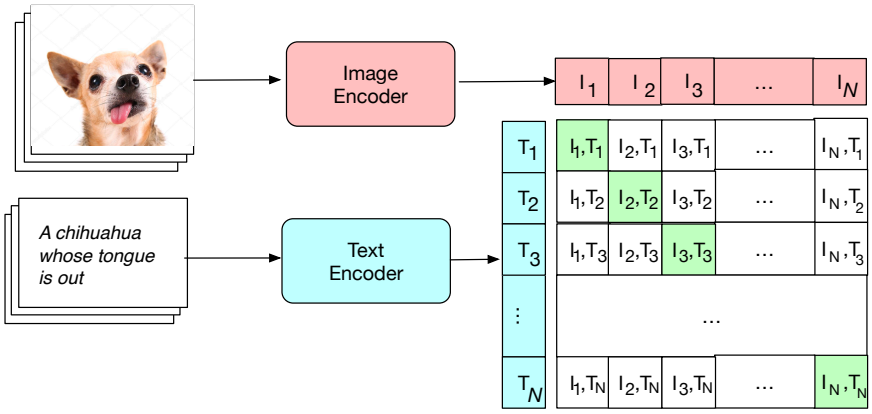


Figure 11-37: CLIP Pre-training phase, each pair of text and image gets into separate encoder and the green diagonal of the result matrix (it is a matrix of cosine similarity) includes positive samples and the white ones are negative samples.

example, in the MNIST dataset, we are limited to 10 labels, whereas ImageNet has 22,000 labels. 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, the labeling process in the classification task is 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 bot zero-shot learning and text prompt labeling by matching images with textual descriptions. It is trained on many images paired with their text captions, which are collected from the Internet[14]. Similar to CLR methods, CLIP input is also a set of image-text pairs.

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 (we will explain them in Chapter 12), 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, the representation (feature vectors that are the result of encoding) matches together.

Figure 11-35 shows the pre-training phase of the CLIP. Each image and text in the Image-text pairs set uses a separate encoding, resulting in a vector. The main diagonal of the matrix in this

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

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 11-37.

Training

To train the model, authors experiment with different ResNet and Transformer models with different resolutions and embedding (feature vector) sizes. Instead of pairing each image model with a specific text Transformer, CLIP uses one image encoder and one text encoder that work together. In other words, the authors complemented the image encoder with a Transformer-based text encoder. Through experiments, the authors identified that a larger Vision Transformer model (ViT-L) paired effectively with the text Transformer, yielding the highest accuracy.

The optimizer of the training is Adam, and it is trained for 32 epochs. CLIP has a hyperparameter, i.e., *temperature parameter*, 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 are 32,768.

For this setting, the authors trained their model on only 256 NVidia V100 GPUs for 12 days. Yes, it is very affordable, and you can train a small model with a small budget in your basement. Joke aside, such pre-training is only possible by a large corporation that has a large number of resources, and even at the time of describing this model, to our knowledge, even universities can not 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. This capability stems from its ability to understand images and textual descriptions through its dual-encoder architecture. For example, in Figure 11-38, when an image of a chicken is fed

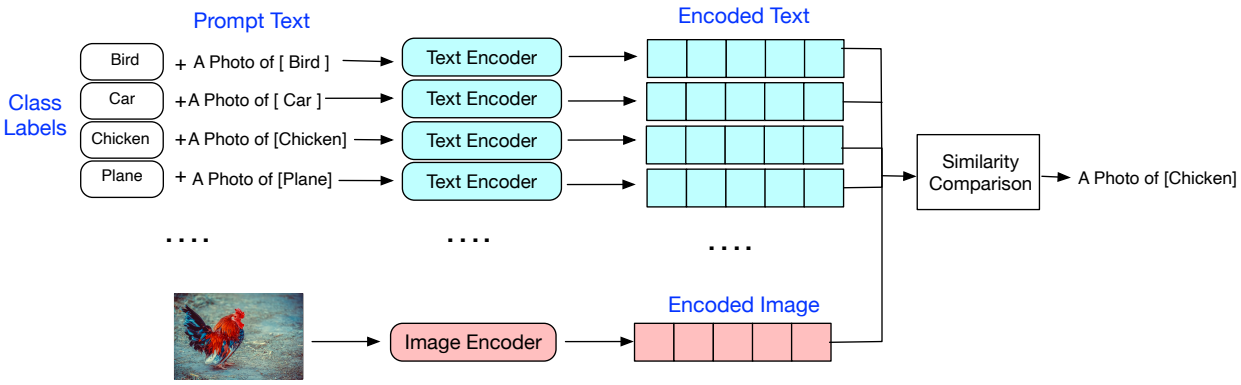


Figure 11-38: Zero-shot learning of CLIP for text prompt prediction. Note that text labels have no information about images. The label will be assigned based on comparing the similarity of the encoded text to the encoded image.

into CLIP, the model does not generate a caption. Instead, a predefined text prompt (e.g., 'A picture of'), manually provided by a model engineer, is used. The model then iterates through possible class labels, combines them with this prompt (like 'A picture of a chicken'), and computes the embeddings of these text phrases. CLIP then compares these text embeddings with the image's embedding. The text whose embedding is closest to that of the image is selected as the best description.

Note that zero training is needed on the actual task of matching the text caption to the image. In other words, the output of the text encoder in Figure 11-36 could be entirely different from the dataset built in Figure 11-37. CLIP identifies the best match by finding the closest text (blue part in Figure 11-38) to the image encoder (red part in Figure 11-38) based on similarity comparison.

In summary, CLIP ensures high similarity between the encoded image and text. This model is very popular in generative AI architecture, and its new variants, such as openCLIP[15], are trained on a larger number of data.

VQGAN (Vector Quantized GAN)

CNN architecture is capable of identifying local relationships correctly, but not able to capture complex higher-level information. In other words, CNN is good at capturing spatial information from pixels close to each other, such as the texture of the object.

VQGAN [Esser ’21] combines both architectures to produce high-resolution synthetic images. To explain it, we separate our explanation into two stages: one stage focuses on its GAN architecture, and the second stage focuses on its transformer.

Stage 1: VQGAN is inspired a lot by VQ-VAE [Van Den Oord ’17]. The primary motivation behind VQ-VAE was to replace the continuous latent space typical of Variational AutoEncoders (VAEs) with a discrete latent space. This discrete latent space provides several advantages, such as enabling more controlled and stable image generation.

To implement this discretization, the authors of VQ-VAE use a vector quantization and construct a codebook from training images. 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 exact dimensions. VQGAN’s codebook is a dictionary that is used to reconstruct images. For example, it has a vector for a sunny sky, a vector for the garden, a vector or more than a vector for a dog, etc. Then, if a query asks to build "a dog in the garden at the sunny sky," these vectors are retrieved from the codebook, and the decoder will use the codebook to construct the image by using these vectors. It is a simplified example, and keep in mind that vectors in the codebook typically represent various visual patterns or textures, not items or scenes.

Why use a 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 it can index them. Besides, the training images are fed into a CNN encoder

[15] https://laion.ai/blog/large-openclip

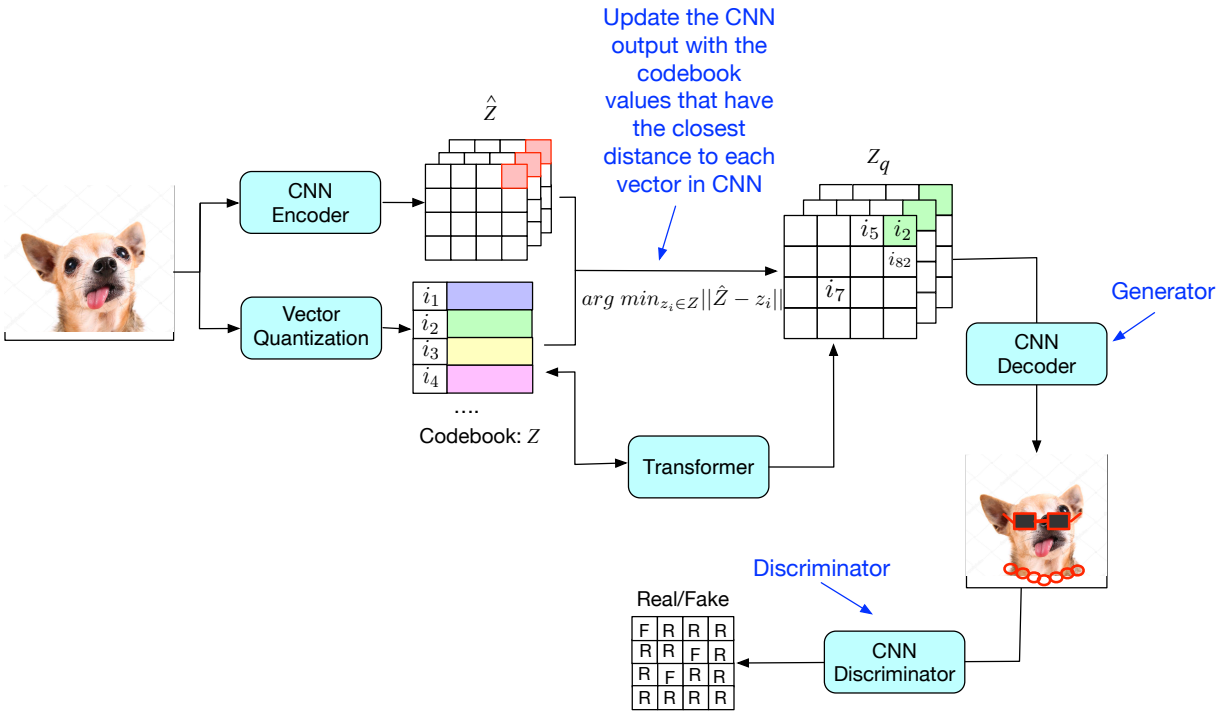


Figure 11-39: VQGAN architecture.

whose output ($\hat{Z}$) reduces the size of the input image while maintaining its spatial information and feature information.

Now, we have trained quantized images into codebook (Z) and reduced size into $\hat{Z}$. Next, 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 continuous output from the CNN encoder ($\hat{Z}$), we now have a discretized tensor Z_q of the same size as $\hat{Z}$. Note that Z_q includes vectors from the codebook rather than CNN Encoder.

Figure 11-39 Visualizes the construction of the codebook from the image. For 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 (i_1 the one with a light green color) will substitute the vector of $\hat{Z}$ in Z_q.

VQGAN, similar to other GAN models, includes a synthetic image generator and a discriminator. The generator is a CNN decoder, as shown in Figure 11-39.

Now, the generator (CNN decoder) constructs a synthetic image from the codebook. The encoder learns to map images to discrete codes, while the codebook learns to store useful feature vectors that can reconstruct the input effectively.

The backpropagation flows through the entire model, including the encoder, but the quantization step introduces a challenge because it is non-differentiable. Since Z_q is derived from discrete

codebook values, direct backpropagation through this step is not possible. Besides, Z_q includes discrete values, and discrete values are non-differentiable and do not have a meaningful gradient. This means the optimizer cannot backpropagated through the encoder during training. To address this, the authors use the *straight-through estimator* (proposed in VQ-VAE). This approach allows gradients from the decoder to be copied to the encoder as if the quantization step were differentiable, ensuring the encoder and decoder can be trained effectively.

For the CNN discriminator, there is not much to explain except that the authors recommend starting the CNN discriminator later in the training because the generator has time to learn. Otherwise, the discriminator can easily recognize the synthetic image, and thus, the generator can not learn.

Stage 2: Until now, we have only had a GAN reconstruct the data. However, the goal is to generate large synthetic images with high resolution. The encoder and decoder are typically invariant to image size, but the transformer can assist the decoder in building larger images. Therefore, a transformer will be used in an autoregressive manner.

We have explained that the codebook acts 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 and sends them 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 (check Chapter 12), it is computationally very hard to feed a large sequence to a transformer. Therefore, the codebook is fed into the transformer, and the authors use a sliding window for the attention model to reduce the size of the image and the computational cost. The transformer processes the sequences of discrete codebooks in an autoregressive manner. This autoregressive approach allows the transformer to sequentially predict different parts of the image, using the output of one step as the input for the next, thus enhancing the overall quality and coherence of the generated images.

The VQGAN 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.

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 ones.

If you would like to bluff about your deep understanding of VQGAN, say the following in front of others: Using the concept of the codebook and feeding it into a transformer makes VQGAN

very attractive and shows its high accuracy in generating high-resolution images. Jokes aside, using a codebook is a popular trick, and we will encounter it later in upcoming chapters.

We do not explain the cost functions of the VQGAN in detail; it uses two cost functions: one is discriminator cost, and the other one is codebook cost. The discriminator cost is a standard GAN cost function whose goal is to train the discriminator to successfully identify the generator's synthetic images from real training images. The codebook cost encourages the discrete latent code Z_q to stay close to the encoder outputs.

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. They make their trained model openly accessible, allowing it to be fine-tuned for other tasks. VQGAN-CLIP is an interaction between VQGAN and CLIP networks, which results in using the following neural networks: the generator and discriminator of VQGAN and a separate neural network of CLIP.

The objective of VQGAN-CLIP is to generate images from the given text prompt. Recall that the VQGAN network generates a high-quality image from the text prompt, and 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 image generated. 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 calculated by CLIP is backpropagated to the VQGAN generator. Then, VQGAN improves its image generation process. This process continues iteratively until the loss reaches the maximum threshold.

We skip the small details of the original paper, but one thing that is worth explaining is the challenge that existed with CLIP loss. The updated gradient 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 models are neural networks designed by OpenAI that generate images from textual descriptions, leveraging the power of diffusion models and transformers to create detailed and imaginative visuals.

DALL-E v1

DALL-E version 1 [Ramesh ’21] was introduced back in early 2021 by OpenAI. The authors use a transformer architecture and autoregressive model. They 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 going to sleep.

Earlier, we explained that autoregressive models are used for sequential data (token). The authors described that they could not consider the pixel of an image as a token of data for the autoregressive model. Because then the model focuses on capturing short-range dependencies between pixels, and thus, long-range relations between pixels will be neglected.

To capture long-range relations, DALL-E v1 operates in two stages. The first stage is focused on training a discrete variational autoencoder (dVAE) to compress image data. The resulting images are encoded and thus smaller but hold the feature of the original images. In particular, the resulting images are encoded into a sequence of 1024 tokens, each of which can take one of 8192 possible values. The codebook contains 8192 vectors.

The dVAE is similar to VQ-VAE, which builds a codebook. 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. In simple terms, the dVAE can output a distribution over codewords. However, since backpropagation is not able to work with discrete data (discrete data are not differentiable), to transform them into continuous forms and make them differentiable, authors use a method called Gumbel-softmax [Maddison '16, Jang ’16]. Gumbel-softmax enables gradient-based optimization despite having discrete data. It approximates the discrete distribution with a smooth one that can be sampled and differentiated. The result of this stage is to convert a 256×256 RGB image into a 32×32 grid of tokens.

The second stage first concatenates image tokens to BPE-encoded text tokens (in Chapter 14, we will explain BPE). This enables the model to process both text and image data in a similar manner. Then, they trained an autoregressive transformer to model the joint distribution over text and image tokens using their 12-billion parameter transformer.

As a result, a model is trained on a large dataset of image and text token pairs. Then, they can be used in an autoregressive fashion and predict the next image token in a sequence, and a decoder builds the entire image from a given text.

DALL-E v2

DALL-E version 2 [Ramesh ’22] was released in 2022, it can construct significantly more realistic and accurate photos than its version 1. DALL-E 2 uses a diffusion model for the decoder, while DALL-E v1 uses an autoregressive model for generating images.

DALL-E 2 is composed of two main components: Prior and Decoder. First, a CLIP model [Radford '21] is trained, and each pair of images and text will be encoded to match image

embeddings (z_i) and text embedding (z_t). The objective of CLIP is to learn a joint embedding space for images and text, enabling it to match text descriptions with relevant images. The top part of Figure 11-40 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 using the CLIP frozen model. The output of the prior is z_p which is an image embedding. The authors stated that they used both diffusion and autoregressive for the Prior.

Afterward, the diffusion decoder produces images conditioned on image embedding (from the prior) and optionally on the caption, i.e., $P(\hat{x} \mid z_p, y)$. This means that unlike the diffusion we have explained earlier, this diffusion model does not start from noise. Instead, it starts with the image embedding produced by the prior. The objective of the decoder is to generate an image that corresponds to the image embedding produced by the prior. Figure 11-40

Why does the decoder perform inverting (the process of reconstructing an image from its latent representation)? The goal is to generate a new image from a text prompt rather than build an identical image as it is encoded by the Prior. The 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 authors call the combination of prior and GLIDE-based decoders to *unCLIP* because it reverses the mapping learned by the CLIP image encoder. By stacking the Prior and Decoder, DALL-E v2 builds a generative model $P(\hat{x} \mid y)$ of images $\hat{x}$ given captions y.

Why do authors of DALL-E 2 use prior and not simply a CLIP encoder with a decoder? The reason for this design choice is that, through experiments, the authors found that having a Prior network in between produces better images. They stated that the Prior creates a summary of the image, and the decoder inverts images given their CLIP image embeddings. During this process, the decoder learns the useful details 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 the 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 details of BSR degradation here.

To train, the encoder authors use CLIP and DALL-E v1 datasets, which include about 650 million images. To train the encoder, upsamplers, and prior, they used 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 text). In the second step,

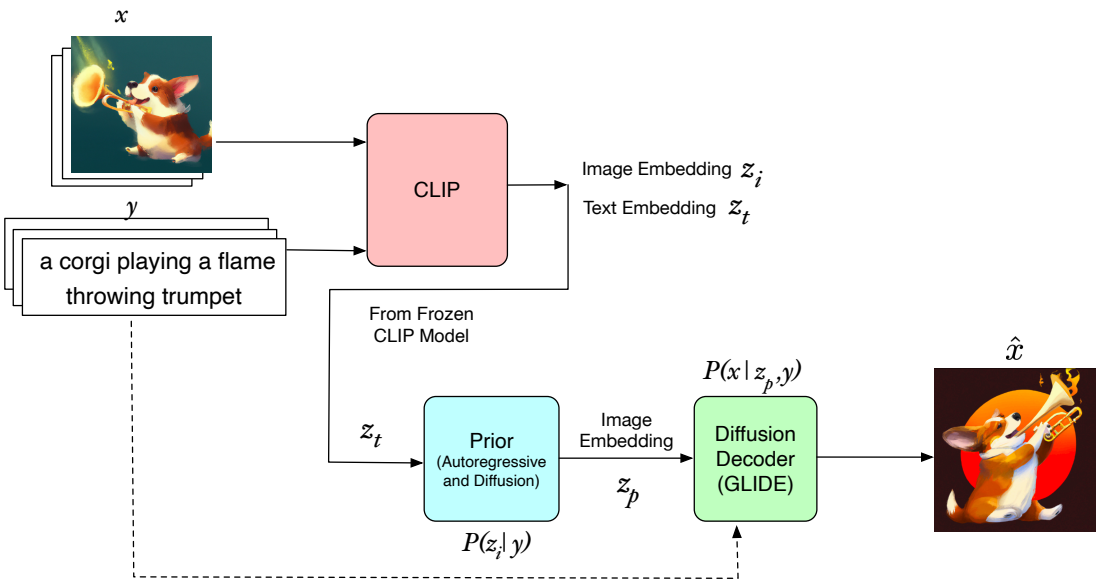


Figure 11-40: DALL-E 2 architecture, which is composed of three networks, CLIP, Prior and a decoder, which is a GLIDE model with some changes.

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[16].

DALL-E v3

In 2024, OpenAI released DALL-E 3 [Betker '23]. Authors claimed that one problem with the DALL-E 2 is the prompt text given by the user, and if there is a mechanism to improve the descriptiveness of the prompt, the text-to-image model can generate high quality images with more details related to the prompt. To handle this issue, they trained an *image captioner* and a text-to-image model to build DALL-E 3.

Image Captioner: The image captioner is a model that is similar to a language model (we will explain in Chapter 12) is used to predict text. It begins by using a tokenizer to break down strings of text into discrete tokens, representing these tokens as sequences, which can then be processed to generate captions for images. This approach is often used to address shortcomings in existing captions found on the internet, which may be incorrect or only tangentially related to the content of the image. The image captioner is jointly trained along with CLIP to leverage CLIP capabilities in measuring text and image similarity, thus ensuring that generated captions closely align with the visual content and human interpretation of images.

When an image captioner is trained alongside CLIP, it can optimize its captions to be more aligned with the actual content of the images. This is particularly useful because it enables the

[16] You can also check the Blogpost of one author to better understand DALL-E 2, http://adityaramesh.com/posts/dalle2/dalle2.html

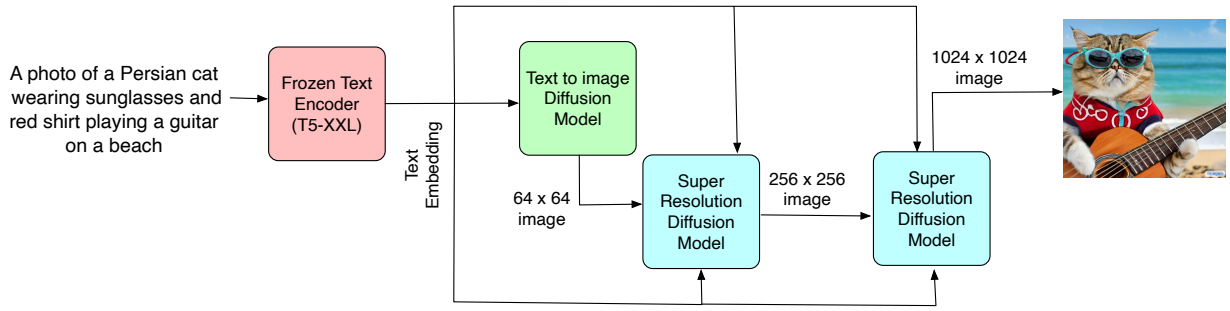


Figure 11-41: Imagen architecture.

model to generate captions that not only describe the images accurately but also in a way that resonates with how humans interpret the images.

Synthetic Captions: The authors stated that first, they build a small dataset (not clear what they mean by small) of captions that describe only the main subject of the image. Then, they trained their image captioner model on this dataset to produce what is referred to as *short synthetic captions*.

Afterward, they repeat the fine-tuning process for the second time with a new dataset that contains long, highly-descriptive captions. These captions include details about not only the main subject but also the surroundings, background, text found in the image, styles, and coloration. The result of the second fine-tuning is referred to as *descriptive synthetic captions*. GPT-4 is utilized to upsample these captions further, enhancing them to include even more detailed descriptions, thereby improving the model's ability to generate images

To train the DALL-E 3, at the end, they use the descriptive synthetic captions on their image dataset to build the final model. Authors claimed they used 95% synthetic captions and 5% ground truth captions. This blending of captions helps balance between creativity and accuracy, improving the model's ability to generate relevant images.

OpenAI does not release any information about where they get the data, and most probably, they use the copyrighted data available on the Internet.

Imagen

Do you recall rich family or friends who do not have anything to do except try to copy what other rich people did and show off their life, which includes more luxury than others? If you think those behaviors belong to people and big corporations don't do the same thing, you are wrong. Anyway, this explanation has nothing to do with the relationship between Google and OpenAI, and we wrote it accidentally here.

A few weeks after the release of Open AI's DALL-E 2, Google released Imagen [Saharia '22], and a few days later, it released Parti [Yu '22], both of which are text-to-image models. The Imagen architecture is significantly easier than DALL-E 2, and in its paper, the authors also introduce an evaluation method, DrawBench, to evaluate the result of text-to-image algorithms. DrawBench evaluates key aspects such as the fidelity of the generated images to the original text,

the creativity and diversity of the outputs, the detail and resolution of the images, and their internal consistency and coherence.

Figure 11-41 presents the architecture of Imagen. Imagen uses a large language model, T5-XXL (we will explain the T5 model in Chapter 12), pre-trained on a text corpora (Check Chapter 12 to learn about language models) to map input text into a text embedding (word embedding). Authors experimented and observed 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 *dynamic thresholding*, which is a sampling method to leverage weight guidance. This weight guidance results in having more photo-realistic images at the end.

Afterward, 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 64×64 diffusion 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 "Pathways autoregressive text-to-image model", and it was introduced a 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 predicts an upcoming token(s).

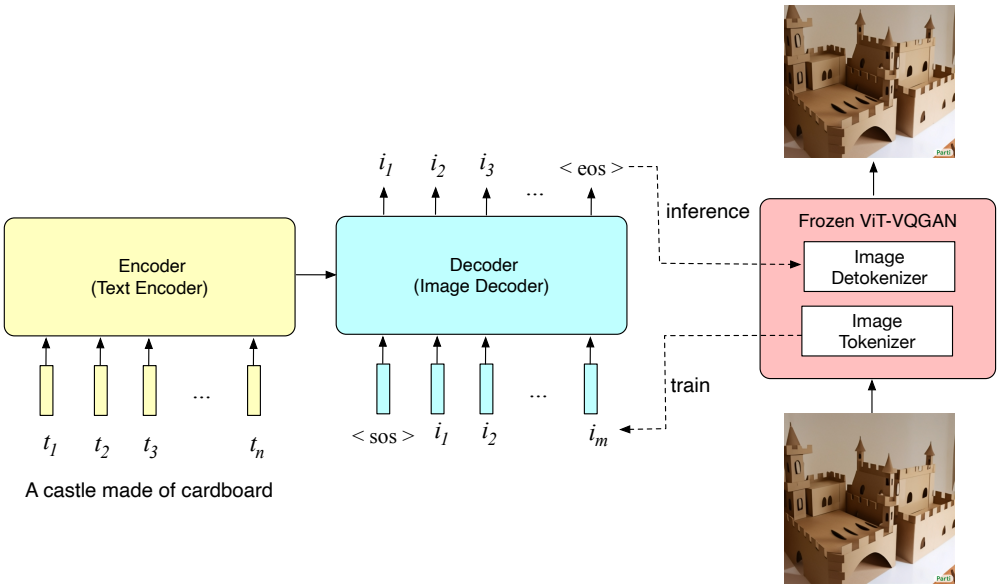


Figure 11-42: Parti architecture.

Parti treats text-to-image generation as a seq2seq model used for machine translation (check Chapter 12 for seq2seq models). Unlike DALL-E 2, which provides a decoder only, it provides both an image encoder and image decoder by using ViT-VQGAN [Yu ’21], and the authors claimed it had outperformed models only with the decoder. We have already explained VQGAN earlier here, and ViT (we will explain in Chapter 12) stayed for the vision transformer. Parti has a transformer-based architecture, and it operates in two stages: an image tokenizer and an autoregressive model.

Image Tokenizer: The first stage trains the image tokenizer to convert an image into a sequence of discrete visual tokens (image embedding). We could assume this as an encoder because the tokenizer's role is to compress the image into tokens effectively, which are then used for generating images.

As explained in the VQGAN section, tokenizing an image into a sequence of pixels results in an extremely large sequence. VQGAN 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. Authors of Parti downsample an input image of the size 1024×1024 to 256×256. Then, they give it for training to ViT-VQGAN. ViT-VQGAN plays a role similar to an encoder in Seq2Seq models (check Chapter 12), as it encodes images into a tokenized format suitable for the autoregressive transformer. The resulting codebook in this model has 8192 entries (codewords). ViT-VQGAN encodes images as a sequence of discrete tokens (codewords), which come from the codebook (not image patches). To train ViT-VQGAN, the authors follow the DALL-E model. After training is done at this stage, the model freezes ViT-VQGAN, which is presented as the red component of Figure 11-42.

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.

Autoregressive model: In the second stage, an autoregressive model Transformer Decoder (presented as a blue box in Figure 11-42) receives the sequence of embedded tokens from the text encoder (yellow box in Figure 11-42) and generates an image token-by-token, predicting each subsequent token based on all previous ones. In other words, this stage focuses on training autoregressive encoder and decoder transformers (the yellow and blue transformers in Figure 11-42) by treating text-to-image as a seq2seq modeling. The input for training is the image and text pair. The Image Decoder, shown in Figure 11-42, gets the (i) sequence of image tokens (extracted by ViT-VQGAN from the image) and (ii) the encoded text prompt that the Text Encoder (yellow box in Figure 11-42) converts to text embedding. Then, it tries to predict the next image token. The text embedding acts as a target for attention (attention and transformer will be explained in Chapter 12).

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

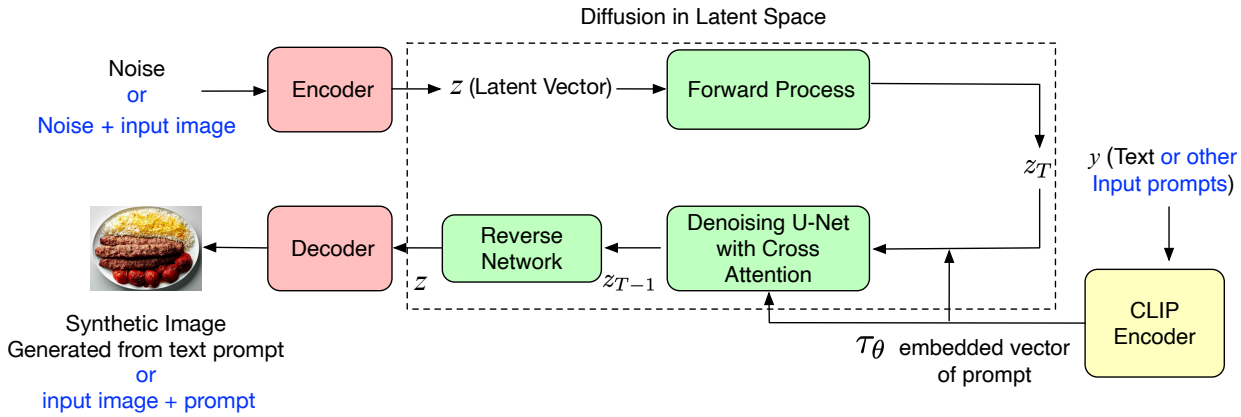


Figure 11-43: The simplified architecture of LDM training, which is the first version of the Stable Diffusion. The blue texts on this figure are used for inferences, and they mean LDM can get an image as input and generate another output, such as editing the given input based on the given text prompt.

The 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 Google and public datasets, and they use about 6 billion text-image pairs.

Stable Diffusion Models

In 2022, Rombach et al. [Rombach '22] provided a text-to-image model, later known as Stable diffusion. Unlike Google and OpenAI, released their model weights open so that others can benefit from their model and further tune it. Their approach leads to the introduction and advancement of many research and applications that can benefit from text-to-image.

Their explanation of their architecture is not easy to understand; we spent reasonable time simplifying their approach, and the simplified version of their architecture is presented in Figure 11-43. In this figure, we use the blue color in the text to emphasize that Stable Diffusion can indeed be used to edit or modify an existing image based on a given text prompt, a capability often referred to as t*ext-guided image editing* or *inpainting*.

Latent Diffusion Model

The first version of Stable Diffusion is called the *Latent Diffusion Model (LDM)*. Authors described that previous diffusion models operate on the original image data, which is too large to process and thus computationally expensive. LDM utilizes an autoencoder, which compresses high-dimensional data into a lower-dimensional space (light red encoder in Figure 11-43). Later, after the diffusion process is trained, the decoder (light red decoder in Figure 11-43) translates the clean vector z into the output (synthetic image). By using autoencoder architecture, the computational cost for both training and inferences is reduced significantly.

The reverse process (diffusion process) includes a U-Net with a cross-attention model. The use of cross-attention allows the model to integrate additional contextual information (e.g., text descriptions, bounding boxes) directly into the diffusion process. We will describe some attention mechanisms in Chapter 12. Similar to self-attention, the cross-attention mechanism involves three main components: queries (Q), keys (K), and values (V). In the context of LDMs, the diffusion model's intermediate layers generate queries (Q), while the encoded conditioning inputs (like text or bounding box descriptions) generate keys (K) and values (V). Note that this U-Net works only with the latent representation of the data and not the original data.

The encoder shown in yellow in Figure 11-41 is used to convert different modalities (text, image, or even audio), but to our understanding and by checking other resources, this is a CLIP encoder that aligns text with the image. This means the encoder constructs the 'conditioning signal' $\tau(\theta)$ that represents the transformation of text input θ into a *conditioning signal* τ, which is used to guide the image generation process via U-Net.

The result of the denoised U-Net is passed to the Reverse Network (shown in green in Figure 11-43), which implements the reverse process of a diffusion model. The output of the reverse network will be given to the decoder that converts the latent space data into pixel space, which means it constructs the output (i.e., synthetic image).

Training LDM

We explain the training process with respect to the figure because there are different components to train, some trained in a sequence and some in parallel.

Encoder Training: The training starts with an encoder that compresses high-resolution images into a lower-dimensional latent space. The encoder is usually a part of a denoising autoencoder (DAE) or a variational autoencoder (VAE) that learns to encode the image into a latent vector z, which retains the essential information but in a more compact form.

Forward Diffusion Process: Next is the training of the forward process. During this phase, Gaussian noise is incrementally added to the latent vectors (encoded input image) over several steps, transforming the data from its structured latent representation to a completely noisy state. This creates a sequence of increasingly noisier versions of the latent vector, represented by $z_1, z_2, \ldots, z_T$, where T is the final time step.

Reverse Network (Denoising Network): Parallel to the forward process, the Reverse Network is trained. This network learns to perform the reverse of the diffusion process, denoising the latent vectors. The network starts from noise z_T (the noisiest state) and trains to predict the less noisy previous state z_{T-1}. By training on many such pairs, the network learns to reverse the entire diffusion process, recovering the original latent vector z from noise.

Integration with Text Prompts: If the LDM is conditioned on text prompts, a separate component, such as the CLIP Encoder, is used to convert text prompts into embedded vectors τ. These embeddings are used during the reverse diffusion process. The Denoising U-Net with Cross Attention incorporates these embeddings to ensure that the features of the generated image match the text prompt's content.

Decoder for Image Generation: Finally, the decoder will be trained to convert latent vectors back into images, is used to generate synthetic images from the denoised latent vectors.

Stable Diffusion XL

After the introduction of Stable Diffusion v1 or LDM, some minor changes were introduced, and later, Stable Diffusion XL (SDXL) [Podell '23] was introduced. SDXL has a larger training set and leverages Reinforcement Learning with Humans in the Loop (RLHF), which we will describe in Chapter 13. Besides, it leverages a three times larger U-Net model than LDM, which has 2.6 Billion parameters, and it can produce images at a resolution of 1024×1024 pixels.

It includes a second U-Net model known as the *refinement model* to improve the visual fidelity of synthetic data generated by the first U-Net model. It also includes two text encoders (CLIP ViT-L and OpenCLIP ViT-bigG) that enhance the model's ability to interpret and synthesize images from textual prompts accurately. The two text encoders allow the system to capture more nuanced details from the text prompts, presumably enabling better text-to-image translation and higher fidelity in the generated images. Figure 11-44 presents the architecture of SDXL, which seems too abstract, in contrast to their previous architectural drawing, which was unnecessarily complex.

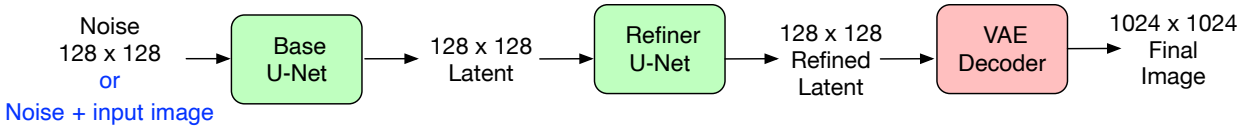


Figure 11-44: SDXL architecture (very similar to the same figure that is presented in the paper).

Additionally, SDXL can handle varying image sizes, as the model needs to adapt to different resolutions and aspect ratios to produce coherent and visually acceptable results. To address this, SDXL employs several image size conditioning strategies, which include:

- It allows users to explicitly specify the desired output image size and aspect ratio in the prompt.
- During training, the model learns to associate specific size embeddings with different image resolutions. At generation time, the appropriate size embedding is selected based on the desired output size, and it is used to condition the image generation process.
- The model starts by generating a low-resolution image and then iteratively increases the resolution through multiple stages. At each stage, the model takes the previous stage's output as input and generates a higher-resolution version of the image. This progressive refinement allows SDXL to generate high-quality images at larger sizes while maintaining detail.

Stable Diffusion 3

Stable Diffusion 3 or SD3 [Esser '24] medium size mode has been released open, but not its large version. SD3 introduced two new components in their architecture: (i) re-weighing of the

noise in Rectified Flow and (ii) employing the Multi-Modal Diffusion Transformer (MM-DiT), along with some other minor improvements that we will learn while we describe the architecture.

Re-weighing of the noise in Rectified Flow: Forward paths from data to noise in the diffusion process can be implemented in different approaches. One common one is known as *Rectified Flow*. Rectified flow optimizes the noise addition in a diffusion model by adjusting the noise to simplify the reverse path. This maintains key structural elements of the original data throughout the process. Rectified flow implements this simplification by taking pairs of data points, one with noise (like a noisy image) and the other with the original clean data. Then, it learns the Ordinary Differential Equation (ODE)[17], which connects these points through a straight path. This involves iteratively refining the path to make it straighter.

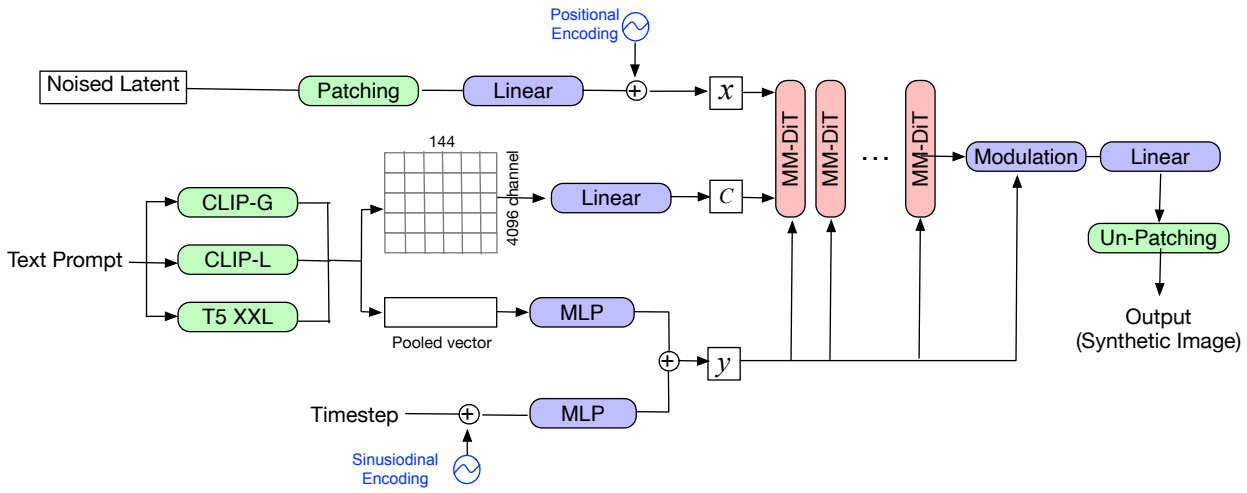


Figure 11-45: SD3 architecture.

The authors introduce *re-weighing of the noise in rectified flow*, which reduces error accumulation during the sampling phase due to increasing the possibility of getting to noise in fewer steps. Sampling in this context refers to generating new data points from the learned model distribution. Besides, the re-weight of the noise improves the fidelity and clarity of the generated images, and ensures that the sampling process is in line with the underlying data distribution.

Multi-modal Diffusion Transformer (MM-DiT): Authors employ diffusion transformers in their architecture. Their architecture uses two separate sets of weights, one for text and one for image. This refers to multi-modality in their architecture. This separation of weights for different modalities improves the model's capacity to synthesize images from textual descriptions by enhancing text comprehension and visual representation. These transformer blocks are referred to as MM-DiT blocks, and they are stacked on top of each other in the model architecture, as

[17] An Ordinary Differential Equation (ODE) is a type of mathematical equation that involves derivatives of one or more functions with respect to a single independent variable. It is called "ordinary" to distinguish it from partial differential equations, which involve partial derivatives with respect to multiple independent variables.

shown in Figure 11-45. Each MM-DiT block includes modulated attention and multi-layer perceptrons (MLPs), which are designed to merge information from text and image effectively.

SD3 Architecture: In the architecture, CLIP-G/14 and CLIP-L/14 provide semantic alignment between text and images, while T5 XXL (we will explain T5 in Chapter 12) processes detailed textual input, ensuring the generated images reflect the content described in the text. The authors described the text prompt passes to these listed models, and the output feature map has 4096 channels.

Having 4096 channels implies a very high-dimensional feature space, allowing the model to capture and process a vast amount of information from the data. This rich feature map passes to a linear layer to apply a linear transformation to it. The result is shown as 'c', which is a transformed feature matrix.

Besides, their architecture shows a pooled vector. Pooling refers to an operation that aggregates or summarizes information from a broader set of data into a more compact form. Therefore, this pooled vector includes a rich feature vector of text and images, which is passed to an MLP.

To incorporate the sequential notation into the model, they use *Timestep* combined with *Sinusoidal encoding* (a type of positional encoding that we will describe in Chapter 12), which it helps the model understand the order or sequence of data. This approach introduces a notion of progression or evolution within the model's processing steps, which assists the model in creating temporally coherent outputs. The raw timestep, along with its Sinusoidal encoding, passes to an MLP to transform the raw timestep data (a numerical indicator of the current phase in the generative process) into a more complex, feature-rich representation. The output of this MLP is combined with the output of the MLP of the 'pooled vector' and constructs the 'y' matrix. This matrix is a feature fusion of textual, visual (from one MLP), and sequential (from the other MLP) features altogether.

The diffusion process starts with a 'noise latent'. Noise latent is not pure noise; the authors apply an intermediary step where the latent representation of an image (initially derived from real data) is systematically infused with noise through a controlled process. We have described this earlier as a re-weighing of the noise in rectified flow. Then, they get the noise latent and apply patching. The patching process is spatially restructured by breaking down the noise into smaller segments. Then, it passes to a linear layer, positional encoding is added, and it results in matrix *x*.

x, y, and *c* go through *n* number of sequential MM-DiT modules. After these modules, there is a 'Modulation' component. Modulation is a mechanism that adjusts the behavior of the model based on certain inputs (timestep and a class vector). It alters parameters or activations in the network dynamically as it processes through the steps of generating an image, ensuring that the network's outputs are appropriately influenced by the inputs and the stage of the diffusion process. The result is passed to another linear layer, and later, they will be unpatched to build the final synthetic image.

Explaining these models are: Finished, تمام, finalizada, خلاص, 完成的, Fertig, завершены, खत्म

The weights of stable diffusion models are openly accessible, which leads us to dedicate serious time to explaining them here. If you find the architecture of SD3 too complex and it reminds you of StyleGAN3, we do agree with you. It seems that the diffusion model is squeezed, and similar to StyleGAN, which was one of the final GAN models; this might be one of the final models of diffusion. The same group of scientists also implemented knowledge distillation to make the distillation process faster, such as SDXL-Trubo [Sauer '24], which introduces Latent Adversarial Diffusion Distillation (LADD), which is a simple distillation approach. In Chapter 16, we explain what is knowledge distillation.

Later, they released another group of models known as FLUX models[18] (the largest one is not open-source), and at the time of revising this chapter for the 100th time, no technical details were shared.

Other models that we didn't explain

In addition to these promising text-to-image models, there are (i) music generation models, (ii) text/image-to-video models, and (iii) face swap (or deep fake) models. We skip explaining them in detail. If you are interested in learning more about music generation models, the authors and his students did a survey [Zhu '23] on AI music generation frameworks, which you can read and get familiar with. On the other hand, there are promising efforts to build video from a text, a.k.a, text-to-video model, such as Make-A-Video [Singer '22], Text2Video-Zero [Khachatryan '23], and Sora[19], KLIG[20], and LumaLab[21] that shows many interesting videos. Besides, there are efforts to construct a video from a single image or a GIF file from a single image, such as [Kandala '24].

---

[18] https://blackforestlabs.ai/#get-flux

[19] https://openai.com/index/sora

[20] https://klingai.org

[21] https://lumalabs.ai/dream-machine

Another group of models, known in layman's terms as deep fake, can swap faces on videos or images, make the image of the person older, combine the two face images, and build a new one. We have explained diffusion models that are capable of doing so. However, there should be used with lots of caution because they are capable of faking a talk of a person or claims, making realistic porn videos from the face of a human, etc. However, there are lots of interesting models in this direction, and we recommend you check Github to learn more about them.

Perhaps, in the next version of this very short book, we will add them as well to be sure nobody is able to finish this book.

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, 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.

After describing concepts, we start explaining SOM, which is one of the oldest unsupervised neural networks that is still in use, mainly for dimensionality reduction. Then, we describe the Boltzmann machine to prepare your poor brain for RBM. Afterward, we focused on 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 VAEs. Regularized autoencoders map a set of data points directly to one single data point in latent space, but VAE maps 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. We conclude the autoencoder discussion by explaining the legendary U-Net, which is a state-of-the-art algorithm for medical image segmentation. U-Net is not called an 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").

Next, we introduce GANs. GAN is the min-max game between two neural networks, a discriminator and a generator. Each of these neural networks has its cost function and its 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 synthetic 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 synthetic data. The discriminator employs Backpropagation to

minimize the loss score for real and synthetic data examples it receives. The generator tries to minimize the discriminator's loss for synthetic data (motivates the discriminator to consider the synthetic data as real data). In other words, the generator tries to increase the false-positive errors of the discriminator, while the 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 GAN architectures that try to mitigate those challenges.

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 finalized this chapter by describing Text-to-Image and inpainting approaches that rely on diffusion models.

Further Reading and Watching

* Frank Noe, has a good RBM tutorial and we used it to explain the details of RBM algorithm, https://www.youtube.com/watch?v=wfFf5Fj-rzE.
* Another tutorial to learn RBM and DBN is the monograph of Bengio [Bengio '09]; that tutorial is among the few ones that explain 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 in understanding 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 is a good online tutorial. His autoencoder tutorial is also very detailed and useful.
* 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.
* Lilian Wang has a brief article on Autoencoders: https://lilianweng.github.io/lil-log/2018/08/12/from-autoencoder-to-beta-vae.html. Also, she has a tutorial on WGAN: https://lilianweng.github.io/lil-log/2017/08/20/from-GAN-to-WGAN.html#wasserstein-gan-wgan
* At the time of writing this section, two books have been 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 would like to have a more hands-on experiment with GAN, we 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 blog post: https://ml.berkeley.edu/blog/posts/dalle2.
* We benefit from the Yannic Kilcher online videos for text-to-image models (https://www.youtube.com/c/YannicKilcher) to learn these models. His explanation is fairly ok, or at least better than the papers, which seem to all be written in haste.
* Umar Jamil has an excellent explanation of the Stable Diffusion model, which we recommend checking it: https://youtu.be/ZBKpAp_6TGI.