SAPIENZA
UNIVERSITÀ DI ROMA

Machine Learning for Generative Models

Facoltà di Ingegneria Informatica e Automatica

Corso di Laurea in Ingegneria Informatica e Automatica

Candidate

Daniele Paliotta
ID number 1708835

Thesis Advisor

Prof.ssa Fiora Pirri

Academic Year 2017/2018

The source code is available at `https://github.com/dpstart/thesis`.

Machine Learning for Generative Models
Bachelor thesis. Sapienza – University of Rome

Author's email: danielepaliotta96@gmail.com

Dedicated to
my family, for the
unquestioned support.

Abstract

"What I cannot create, I do not understand."

– Richard Feynman

Generative models try to capture some data distribution in order to learn how to generate new samples from scratch.
In the scenario of supervised learning, a generative model estimates the joint probability distribution of data $P(X, Y)$ between the observed data X and corresponding labels Y.
Another requirement of a generative model, which is less frequently stated, is that it provides a way to sample X, Y pairs. This simply means that we want our model to understand how a certain type of data is distributed, and to be able to generate data from that distribution.
However, this is easier said than done. Some data distributions can be extremely difficult to infer, especially in high-dimensions: imagine having to estimate the probability distribution of certain types of images. Each sample of the dataset can have a huge number of dimensions (one for each pixel), and the relationships between the features can seem quite arbitrary, or very difficult to estimate anyway. This is just one of the reasons why generative models haven't been very successful for many, many years.

The advent of deep learning, with the subsequent great advances in hardware and in the training process of deep neural networks, has spiked research on generative models, which outperformed every other approach.
The most successful framework proposed takes the name of *Generative Adversarial Networks* (*GAN*s in short).
Since their first appearance in a popular 2014 paper by Ian Goodfellow *et al.* [1], many advances where made, and several different probabilistic approaches where tested.
GANs are now used for a large amount of tasks, and are considered state-of-the-art for many purposes, from image generation to image-to-image and video-to-video translation, from drug discovery to text-to-image synthesis and style transfer.
The main goal of my work is to look into modern generative models that make use of deep learning to get some insight into their power, going deep into their inner workings.

In Chapter 1, I introduce the basic concepts that are required to properly understand modern generative models.
In Chapter 2, I go deep in explaining and dissecting *Generative Adversarial Networks*, from both a theoretical and practical point of view.
In Chapter 3, I go over a family of models known as *Variational Autoencoders*, which provide an extremely interesting and effective framework to capture data distribution using Bayesian inference.
In Chapter 4, I discuss a novel application of GANs known as *image-to-image translation*.

I also provide a Python implementation of the most significant models and architectures.

Contents

1

Machine learning background

In order to properly understand generative models, one needs to have a quite solid foundation on several fields of study, from probability theory and statistics, to multivariate calculus, to linear algebra.
The intent of this chapter is to give the reader an introduction to the most important theoretical concepts standing at the foundation of generative models in general. This is a necessary step on our path to deeply understand the inner workings of these powerful models, and mastering the basics is also vital if we want to be able to fit these types of models to our needs.

1.1 Supervised, Unsupervised, Semi-supervised

1.1.1 Supervised learning

Supervised learning is all about learning a the mapping between our data, that we call X, and a set of labels for our data, that we call Y. The goal is to learn the mapping so well that we able to make predictions on data that our models has never seen before.
Example of supervised learning are text classification, image classification, spam detection, and many others.

1.1.2 Unsupervised learning

Unsupervised learning comes into play when we only have some data X, but we do not have labels or output variables. Unsupervised learning algorithm are therefore used to gain insight into our data, find relationships between samples, and find hidden structures in the distributions of our dataset. An example of an unsupervised learning technique are *clustering* algorithms, which are employed to discover the inherent groupings in the data

1.1.3 Semi-supervised learning

We often have a very large amount of data to work with, but very few labels for it. This is why we use semi-supervised learning. This is because it can be expensive or time-consuming to label data as it may require access to domain experts. Whereas unlabeled data is cheap and easy to collect and store.
You can use unsupervised learning techniques to discover and learn the structure in the input

variables.
You can also use supervised learning techniques to make best guess predictions for the unlabeled data, feed that data back into the supervised learning algorithm as training data and use the model to make predictions on new unseen data. [2]

1.2 Neural Networks

Neural networks are nothing but *universal function approximators*.
They are made of several *layers* of computation, and each layer is made of several *nodes*.
Every node performs a smalls step of computation to its input.
Each successive layer receives the output from the layer preceding it. The last layer produces the output of the system.
The computations are dependent on the parameters, or *weights*, of the network.
The end goal of a neural network, as stated above, is to approximate some function, which means learning some mapping between input and output. It does so by trying to find the best weights that, under that network architecture, approximate that function as precisely as possible.
To achieve this, we need to provide the network with a metric of how well (or, usually, how bad), it is doing. This comes in the form of a *loss function*.
During the *training* phase, the network will look at the input data we provide, make its computations, and then evaluate the loss function. It will then self-modify its weights in order to decrease the loss function. When the loss is reasonably low, we can have a certain confidence that our model has learned to properly approximate the function.
For brevity, I won't dive into the technical, mathematical details of how neural networks work. A more detail explanation is available at [21].

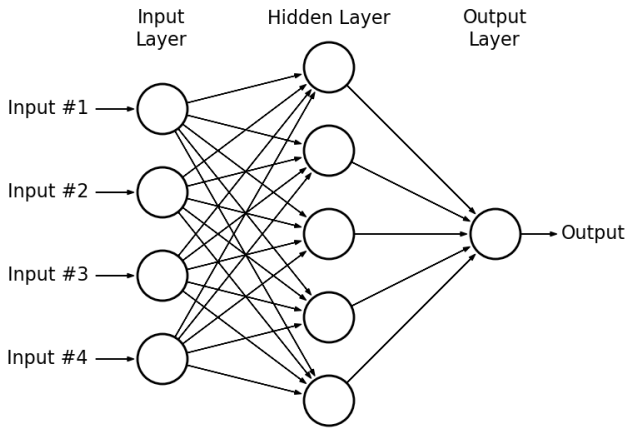


Figure 1.1. A simple neural network with three layers.

1.3 Backpropagation

Backpropagation is the algorithm that neural networks use to learn from data.
Instead of diving into the equations for the general case, I will give a simple intuition of the algorithm, and I will then provide a detailed example in Appendix B.
Refer to [21] for a more rigorous approach.
The general idea of backpropagation is to iteratively update the networks weights in order to decrease its loss.
Let's take as an example a neural network training on a labeled dataset: at every training iteration, the networks processes the input and calculate the loss by confronting the output with the target label.
Now it needs to find a way to decrease the loss. It does so by calculating the gradient of the loss with respect to the weights of the network.
The gradient is the vector of partial derivatives, and it represent the direction of steepest ascent of the loss function. We update our parameters so that our loss follows the direction opposed to the gradient, in order to try to reach a minimum in which the network has properly learn the mapping between X and Y.
The name of the general algorithm is gradient descent, which takes the name of backpropagation when applied to a neural network architecture.

$$\text{Repeat until convergence } \{$$

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

$$\}$$

Figure 1.2. Parameters update in gradient descent. J is the cost function.

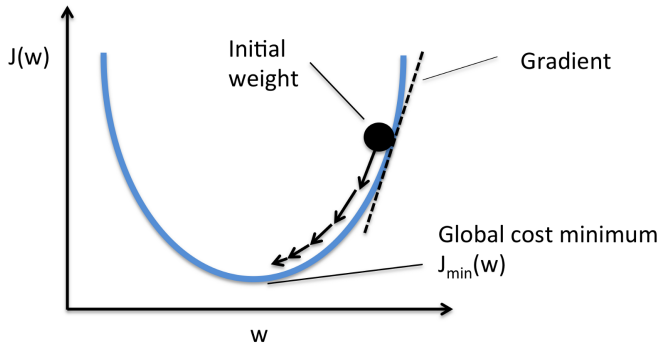


Figure 1.3. Visualization of gradient descent

See Appendix B for the example of a full iteration of the algorithm.

1.4 Variational Inference

Latent variables

A *latent variable* is a variable that is not directly observed from the data, but is *inferred*. There is, however, some kind of relationship that links a latent variable to a certain observed variable. A latent variable Z is related to an observed variable X through its conditional distribution $P(X|Z)$.

Bayes theorem

Bayes theorem is one of the fundamental theorems in probability theory.

$$P(Z \mid X) = \frac{P(X \mid Z)\, P(Z)}{P(X)}$$

- $P(Z \mid X)$ is called *posterior probability*. In the context of a classifier, it tells us the probability that a certain sample of data X belongs to a certain class $Z = z$.
- $P(X \mid Z)$ is the *likelihood*. It answers the following question: "given a value of Z, how 'probable' is a certain sample X to belong to that category?"
- $P(Z)$ is called *prior probability*, and it captures any prior knowledge we have about the distribution of Z.

KL-divergence

The Kullback-Leibler Divergence (KL-Divergence in short) is a metric[1] which is widely used to compare probability distribution. It is commonly used when we want to approximate a complex distribution with a simpler one, since it tell us how much information we lose due to this approximation.
Variational Autoencoders are based on the KL-Divergence, as we will see in chapter 4.
The meaning of the KL-Divergence is rooted in information theory. We begin by defining a very important metric in information theory called *entropy*, denoted as H:

$$H = -\sum_{i=1}^{N} p(x_i) \cdot \log p(x_i)$$

If we use $\log_2$, entropy tells us the minimum number of bits it would take us to encode our information [4], where our information is some probability distribution.
KL-Divergence directly derives from our definition of entropy:

$$D_{KL}(p||q) = -\sum_{i=1}^{N} p(x_i) \cdot (\log p(x_i) - \log q(x_i))$$

[1] The KL-Divergence is not technically a distance *metric*, since it's not symmetric, and because it does not satisfy the triangle inequality.
For convenience, I will use the term distance metric when talking about the KL-Divergence

Thus, the KL-Divergence can be defined as the expectation of the log-difference between two probability distribution. In the context of variational inference, one of the two distribution is a complex distribution we want to approximate, and the other one is the approximating distribution.
The standard way of writing the formula is:

$$D_{KL}(p||q) = -\sum_{i=1}^{N} p(x_i) \cdot \frac{\log p(x_i)}{\log q(x_i)}$$

since $\log a - \log b = \frac{\log a}{\log b}$.

2

Generative Adversarial Nets

2.1 Vanilla GANs

*GAN*s have arguably been the most successful among all generative models.
They provide a framework that is now commonly used for generating data, allowing to capture data distributions in an innovative way.
Since the first 2014 paper on the subject [1], a lot of AI research has been focusing on developing and improving this kind of models.
Figure 2.1 shows the trend of GAN research since 2014 in terms of number of papers produced.

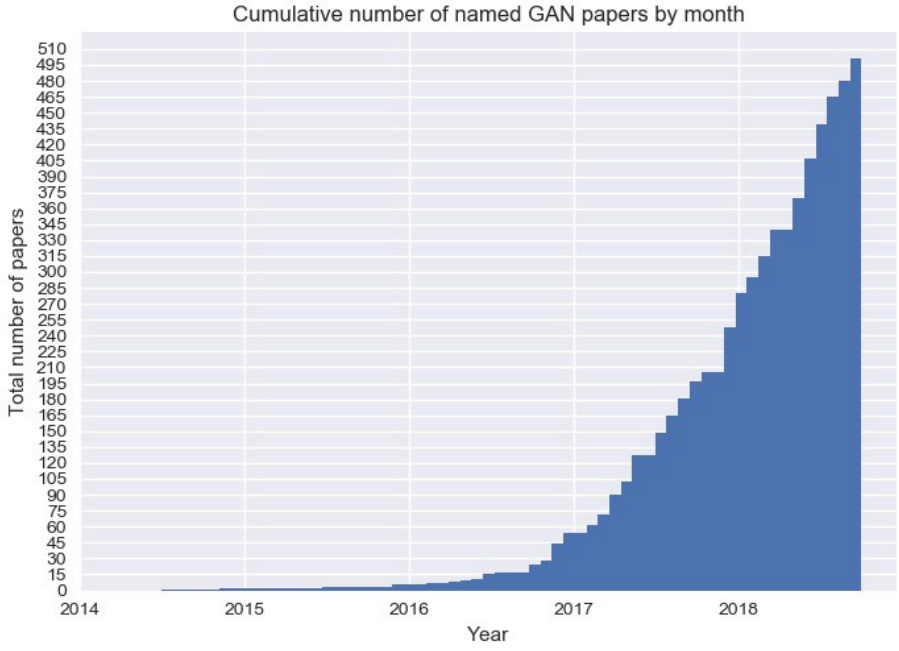


Figure 2.1. Number of GANs papers since 2014

2.1.1 Theory and Architecture

Put it simply, a *GAN* is composed of two neural networks: a **generator** G and a **discriminator** D. The goal of the discriminator is to tell whether a data sample came from the real data distribution, or whether it was instead generated by G. The goal of the generator is to generate data samples such as to fool the discriminator.
Let's get into more detail: we want to learn the generator distribution p_g over data **x**.
We first define an input noise variable $p_z(z)$, and a mapping to data space $G(z;\theta_g)$, where G is a neural network with parameters θ_g. This is our generator. We now define our discriminator network as a mapping $D(x;\theta_d)$ that outputs a single scalar: the probability that x came from the real data rather that the generator distrubution p_g.
We now train D to maximize the probability of assigning correct labels to both training examples and samples generated from G, and we train G at the same time to minimize $\log\left(1 - D(G(z))\right)$

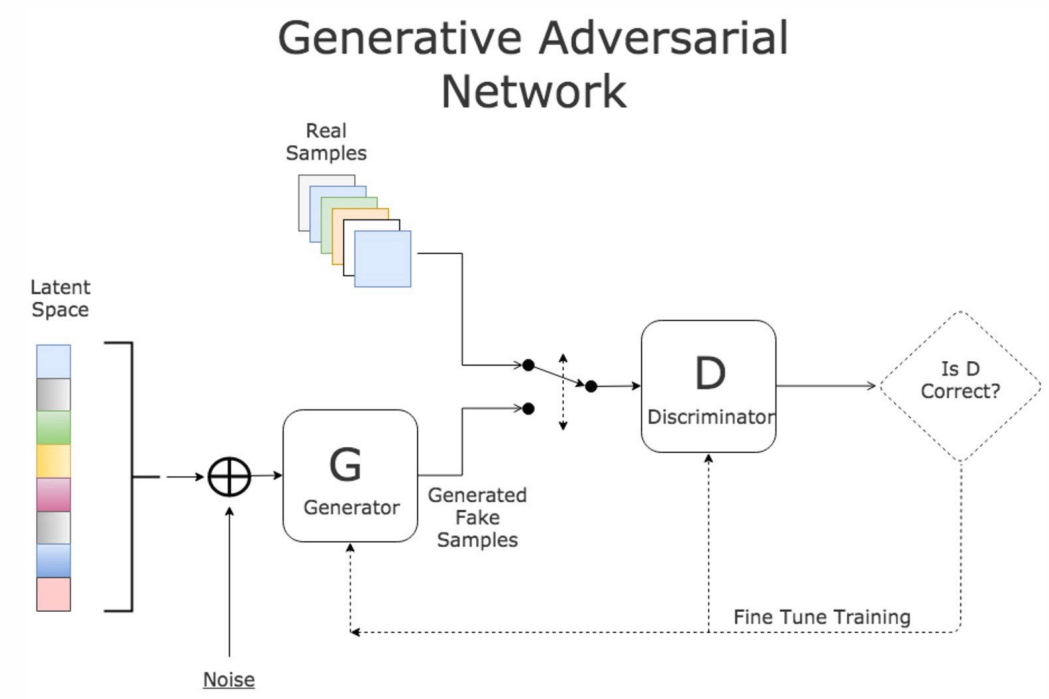


Figure 2.2. Overview of GANs architecture

We can write the training criterion in equation form, as a minimax game with a value function $V(D,G)$:

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

Optimal value for D

Now we have a well-defined loss function. Let's first examine what is the best value

for D.
First of all, we can rewrite the cost function as:

$$\min_G \max_D V(G,D) = E_{x\sim p_{data}}[\log D(x)] + E_{x\sim p_g}[\log(1-D(x)))]$$

(we omitted the generator changing the distribution of x in the second term), and we can expand the expectation to an integral form:

$$L(G,D) = \int_x \Big(p_{data}(x)\log D(x) + p_g(x)\log(1-D(x)\Big)dx$$

Now, let us label

$$\tilde{x} = D(x), A = p_{data}(x), B = p_g(x)$$

Then our cost function can be written as

$$f(\tilde{x}) = A\log\tilde{x} + B\log(1-\tilde{x})$$

Taking the derivative and changing the basis of log:

$$\frac{df(\tilde{x})}{dx} = A\frac{1}{\ln 10}\frac{1}{\tilde{x}} - B\frac{1}{\ln 10}\frac{1}{1-\tilde{x}} =$$

$$\frac{1}{\ln 10}(\frac{A}{\tilde{x}} - \frac{B}{1-\tilde{x}}) =$$

$$\frac{1}{\ln 10}(\frac{A-(A+B)\tilde{x}}{\tilde{x}(1-\tilde{x})}$$

If we set $\frac{df(\tilde{x})}{dx} = 0$, we get the best value for the discriminator:

$$D^*(x) = \tilde{x} = \frac{A}{A+B} = \frac{p_{data}(x)}{p_{data}(x)+p_g(x)}$$

Once the generator is trained to its optimum, $p_g(x) = p_{data}(x)$, thus $D^*(x)$ becomes $\frac{1}{2}$.

Global optimum

When both G and D are optimal, we have seen that $D = \frac{1}{2}$. Thus, the loss function becomes:

$$L(G,D^*) = \int_x \Big(p_{data}(x)\log D(x) + p_g(x)\log(1-D(x)\Big)dx =$$

$$\log\frac{1}{2}\int_x p_{data}(x)dx + \log\frac{1}{2}\int_x p_g(x)dx =$$

$$-\log 4$$

2.1.2 Training

The training of GANs has always proved to be very difficult and unstable. Optimizing D to its optimum on every training iteration would not only be extremely expensive computationally, but would also result in overfitting. Instead, the most commonly used method is to perform k steps of training for D and one single step of training for G on every iteration. This makes the training less prone to overfitting, but it also maintains D close to its optimal solution, as long as G does not change too fast.
We can see that this is not an optimal way of training a model. We shouldn't need to manually balance the training, and this is in fact a known issue with standard GANs. This problem was partially solved in WGANs, which we will discuss in Section 2.6

Here's the pseudocode for the training:

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \dots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
  • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \dots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  • Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

2.1.3 Implementation and results

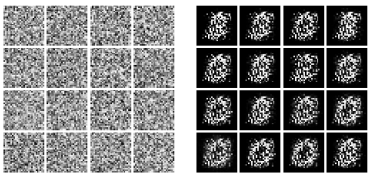
Here's the Tensorflow implementation of a basic **GAN**.
We use fully connected neural networks with only one hidden layer for the generator and discriminator, and we train it on the popular MNIST dataset of handwritten digits.

```
1 def xavier_init(size):
2     in_dim = size[0]
3     xavier_stddev = 1. / tf.sqrt(in_dim / 2.)
4     return tf.random_normal(shape=size, stddev=xavier_stddev)
5
6 def sample_z(n,m):
7     return np.random.uniform(-1.,1., size=[n,m])
```

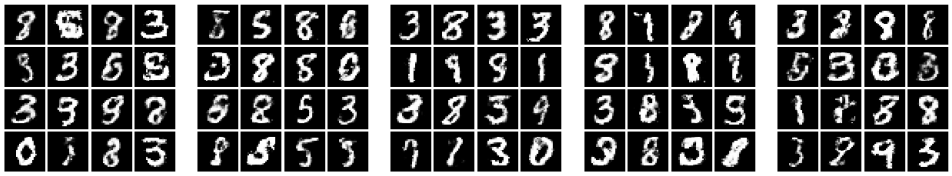
```
8
9
10 # Defining input placeholder and parameters for the generator
11
12 Z = tf.placeholder(tf.float32, shape=[None, 100])
13
14 G_W1 = tf.Variable(xavier_init([100, 128]))
15 G_b1 = tf.Variable(tf.zeros(shape=[128]))
16
17 G_W2 = tf.Variable(xavier_init([128, 784]))
18 G_b2 = tf.Variable(tf.zeros(shape=[784]))
19
20 theta_G = [G_W1, G_W2, G_b1, G_b2]
21
22
23 # Defining input placeholder and parameters for the discriminator
24
25 X = tf.placeholder(tf.float32, shape=[None, 784])
26
27 D_W1 = tf.Variable(xavier_init([784, 128]))
28 D_b1 = tf.Variable(tf.zeros(shape=[128]))
29
30 D_W2 = tf.Variable(xavier_init([128, 1]))
31 D_b2 = tf.Variable(tf.zeros(shape=[1]))
32
33 theta_D = [D_W1, D_W2, D_b1, D_b2]
34
35
36 # Defining the generator network
37
38 def generator(z):
39     G_h1 = tf.nn.relu(tf.matmul(z, G_W1) + G_b1)
40     G_log_prob = tf.matmul(G_h1, G_W2) + G_b2
41     G_prob = tf.nn.sigmoid(G_log_prob)
42
43     return G_prob
44
45
46
47 # Defining the discriminator network
48
49 def discriminator(x):
50     D_h1 = tf.nn.relu(tf.matmul(x, D_W1) + D_b1)
51     D_log_prob = tf.matmul(D_h1, D_W2) + D_b2
52     D_prob = tf.nn.sigmoid(D_log_prob)
53
54     return D_prob
55
56
57
58 G_sample = generator(Z)
59 D_real = discriminator(X)
60 D_fake = discriminator(G_sample)
61
62
63 # Losses
64
```

```
D_loss = -tf.reduce_mean(tf.log(D_real) + tf.log(1. - D_fake))
G_loss = -tf.reduce_mean(tf.log(D_fake))


# Optimizers for Discriminator and Generator

D_solver = tf.train.AdamOptimizer().minimize(D_loss, var_list = theta_D)
G_solver = tf.train.AdamOptimizer().minimize(G_loss, var_list = theta_G)


z_size = 100
mb_size = 16


# Training loop

sess = tf.Session()
sess.run(tf.initialize_all_variables())

i = 0

for it in range(1000000):

    X_mb, _ = mnist.train.next_batch(mb_size)

    _, D_loss_curr = sess.run([D_solver, D_loss], feed_dict={X: X_mb, Z: sample_z(mb_size, z_siz
    _, G_loss_curr = sess.run([G_solver, G_loss], feed_dict={Z: sample_z(mb_size, z_size)})
```

During the first iterations of the training loop, the generator merely takes as input a randomly initialized set of pixels, and uses randomly initialized parameters as the weights and biases for the network. Here's how the generator's output looks like at the first iteration, and after several hundreds iterations:



And here is the output of the generator network towards the end of the training loop:



2.1.4 Improved training techniques

The training of GANs is extremely unstable.
The reason for this is that we are trying to find a Nash equilibrium to a non-convex, two-player, non-cooperative game with continuous, high-dimensional parameters, in which each

player wants to minimize its cost function. However, GANs are typically trained using gradient-descent techniques, that are designed to find the low value of a cost function, but are not designed to find the Nash equilibrium of a game [3].
We will take a closer look at the training issues of GANs in Section 2.5.

However, several new techniques and improvements were proposed for more stability in the training of GANs. All of these improvements, though, still fail to guarantee convergence. They are mostly motivated by empirical, heuristic understanding of the training process of these models, but they were nonetheless successful in generating better, more realistic samples of data.

Feature matching

Feature matching is used to prevent the generator from overtraining on the current discriminator.
This is achieved by changing the objective of the generator. We don't train it anymore using the output of the discriminator as the objective. Instead, we now training using as objective the intermediate features of the discriminator.
This is motivated by the fact that the discriminator should learn the features that are most discriminative of real data versus the data coming from the generator.

Minibatch discrimination

Minibatch discrimination [3] is used to avoid a condition known as *mode collapse* (we will discuss it in more detail in Section 2.5), which prevents the model to correctly capture the data distribution.
Essentially, mode collapse happens when the generator starts to output the same samples over and over. This can happen because, as we know, the generator wants to minimize $\log(1 - D(G(x))$. So, when the generator gets high probability from D for certain samples, it might happen that it will produce those samples again and again, avoiding those samples that will produces low probability from D. As a consequence, the model will fail to capture the entire data distribution.
One of the reasons why this happens is because the discriminator always looks at one point in isolation. If the discriminator was able to look at multiple example in combination, the loss would be more representative of the actual quality of the generator over the whole distribution.
Thus, *minibatch discrimination* is nothing but a technique for incorporating information about a whole minibatch of data in what the discriminator sees.
It works in the following way: let $f(x_i)$ denote a vector of features for input x_i, produced by some intermediate layer in the discriminator.
We then multiply the vector $f(x_i)$ by a tensor $T \in R^{A \times B \times C}$, which results in a matrix $M_i \in R^{B \times C}$.
We then compute the $L1$-distance between the rows of the resulting matrix M_i across samples $i \in \{1, 2, ..., n\}$ and apply a negative exponential.
We then take the resulting values for each row, sum them up and now we have our minibatch discrimination values. We will concatenate these to our normal outputs from the intermediate

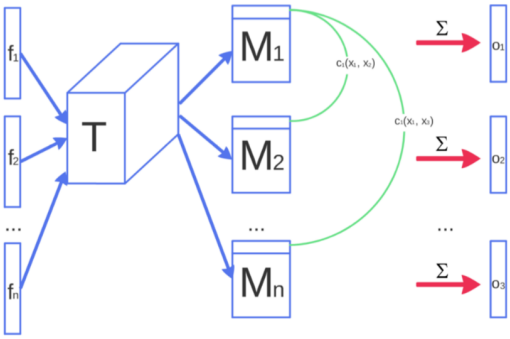


Figure 2.3. Diagram of how minibatch discrminator works.

layer, as show in figure 2.2.

Historical averaging

In historical averaging, we keep track of the model parameters for the last t iterations.
Alternatively, we update a running average of the model parameters if we need to keep a long sequence of models.
Using these parameters, we add an $L2$ cost to the cost function to penalize model different from the historical average:

$$\|\theta - \frac{1}{t}\sum_{i=1}^{t}\theta[i]\|_2$$

For GANs with non-convex object function, historical averaging may stop models circle around the equilibrium point and act as a damping force to converge the model. [3]

One-sided label smoothing

Label smoothing was proposed in [16] for encouraging the model to be less confident.
It consist on replaceing the 0 and 1 targets for a classifier with smoothed values, like 0.9 or 0.1.
While this may not be desired if the goal is to maximize the log-likelihood of training labels, it does regularize the model and makes it more adaptable.
In the context of GANs, we generally smooth only the positive labels, leaving negative labels set to 0.

2.2 DCGAN

DCGAN stands for *Deep Convolutional GANs*.
DCGANs add the architectural constraints of Convolutional Neural Networks to the GANs, achieving better results [5].
Previous attempts to scale up GANs with CNNs had been unsuccessful, primarily due to the instability of the training process. However, with the introduction of state-of-the-art techniques such as dropout and batch normalization [15], we can now train Convolutional

GANs in a stable way in most settings.
The main constraints and improvemenets that were applied to stabilize training where:

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use batchnorm in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except for the output, which uses Tanh.
- Use LeakyReLU activation in the discriminator for all layers.

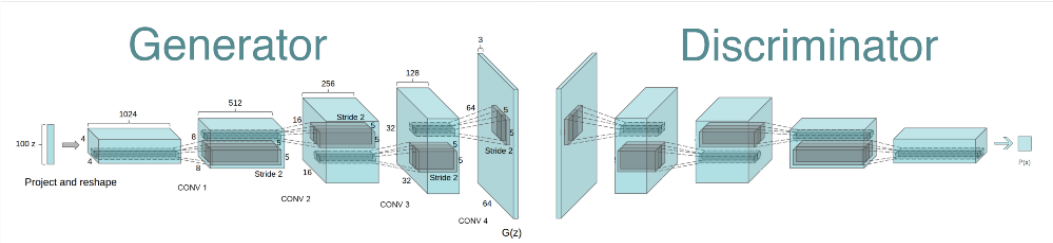


Figure 2.4. DCGAN Architecture

Here are some great results from the original DCGAN paper [5]:

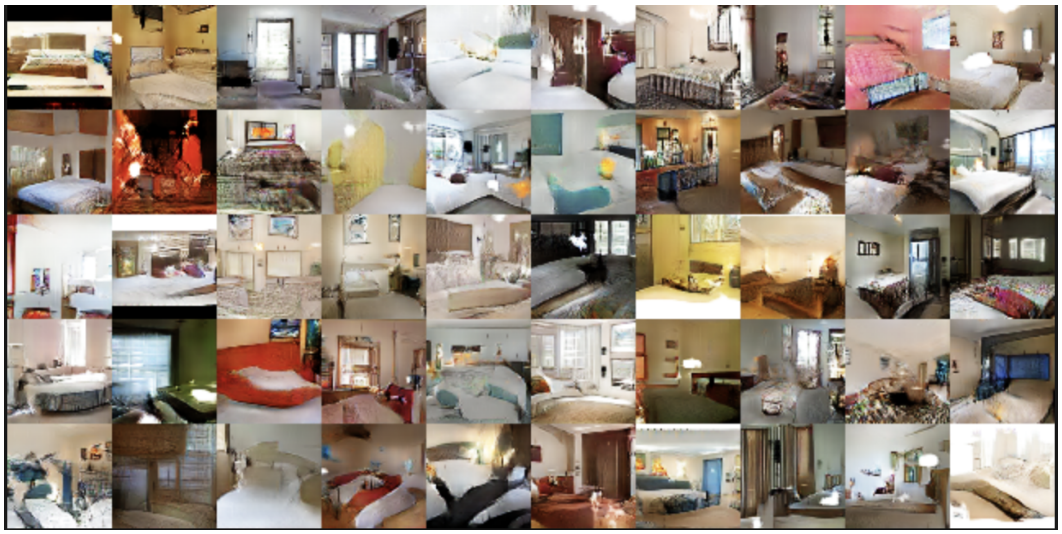


Figure 2.5. Results from DCGANs.

2.3 InfoGAN

2.3.1 Theory and architecture

The standard formulation of GANs imposes no restrictions on the manner in which the generator may use the input noise z. As a result, it is possible that the noise will be used

by the generator in a highly entangled way, causing the individual dimensions of z to not correspond to semantic features of the data, such as stroke width, angle etc.
InfoGANs were introduced to learn more meaningful, disentangled representations of data in the context of unsupervised learning.
The end goal is to divide the input variables into a small subset of latent variables c and noise variables z. The hope is that by forcing high information content in the latent variables, we cram the most interesting aspects of the representation into c.
If we are successful, c ends up representing the most salient and most meaningful sources of variation in the data, while the rest of the noise variables z will account for additional, meaningless sources of variation and can essentially be dismissed as uncompressible noise [6].
We first define an information-theoretic concept called *mutual information*:

$$I(X;Y) = H(X) - H(X \mid Y)$$

where H represents the entropy (See Section 1.4).
An intuitive explanation is that mutual information between X and Y tells us how much we learn about X when Y is observed.
This definition makes it easy to formulate a cost so that the information in the latent code c isn't lost in the generation process.
This cost direcly derives from the classical GAN cost function, but it adds a regularization term that accounts for the new constraint on mutual information:

$$\min_G \max_D V_I(D,G) = V(D,G) + \lambda I(c; G(z,c))$$

To make the problem more tractable, we define an auxiliary distribution $Q(c \mid x)$, that allows us to obtain a lower bound for $I(c; G(z,c))$.

$$I(c; G(z,c)) = H(c) - H(c; G(z,c) =$$
$$E_{z\sim G(z,c)}[E_{c'\sim P(c|x}[\log P(c' \mid x)]] + H(c) =$$
$$E_{z\sim G(z,c)}[D_{KL}(P(\cdot \mid x) \mid\mid Q(\cdot \mid x)) + E_{c'\sim P(c|x}[\log Q(c' \mid x)]] + H(c) \geq$$
$$E_{z\sim G(z,c)}[E_{c'\sim P(c|x}[\log Q(c' \mid x)]] + H(c)$$

Through this technique of lower bounding, we avoid having to compute the posterior $P(c \mid x)$.
For simplicity, entropy $H(c)$ is treated as a constant.
The auxiliary distribution Q is instead parametrized as a neural network.

2.3.2 Implementation and results

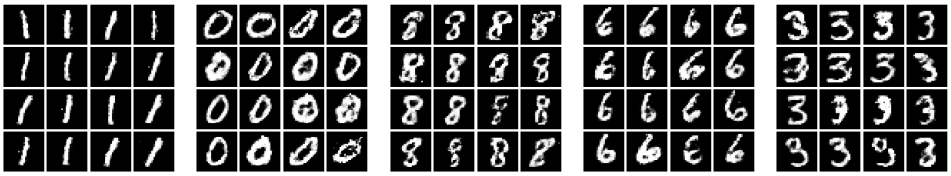
```
1 def xavier_init(size):
2     in_dim = size[0]
3     xavier_stddev = 1. / tf.sqrt(in_dim / 2.)
4     return tf.random_normal(shape=size, stddev=xavier_stddev)
5
```

```
# Input placeholder
X = tf.placeholder(tf.float32, shape=[None, 784])

# Discriminator parameters
D_W1 = tf.Variable(xavier_init([784, 128]))
D_b1 = tf.Variable(tf.zeros(shape=[128]))

D_W2 = tf.Variable(xavier_init([128, 1]))
D_b2 = tf.Variable(tf.zeros(shape=[1]))

theta_D = [D_W1, D_W2, D_b1, D_b2]


# Noise placeholder
Z = tf.placeholder(tf.float32, shape=[None, 16])

c = tf.placeholder(tf.float32, shape=[None, 10])

# Generator parameters
G_W1 = tf.Variable(xavier_init([26, 256]))
G_b1 = tf.Variable(tf.zeros(shape=[256]))

G_W2 = tf.Variable(xavier_init([256, 784]))
G_b2 = tf.Variable(tf.zeros(shape=[784]))

theta_G = [G_W1, G_W2, G_b1, G_b2]

# Q parameters
Q_W1 = tf.Variable(xavier_init([784, 128]))
Q_b1 = tf.Variable(tf.zeros(shape=[128]))

Q_W2 = tf.Variable(xavier_init([128, 10]))
Q_b2 = tf.Variable(tf.zeros(shape=[10]))

theta_Q = [Q_W1, Q_W2, Q_b1, Q_b2]


def sample_Z(m, n):
    return np.random.uniform(-1., 1., size=[m, n])


def sample_c(m):
    return np.random.multinomial(1, 10*[0.1], size=m)


def generator(z, c):
    inputs = tf.concat(axis=1, values=[z, c])
    G_h1 = tf.nn.relu(tf.matmul(inputs, G_W1) + G_b1)
    G_log_prob = tf.matmul(G_h1, G_W2) + G_b2
    G_prob = tf.nn.sigmoid(G_log_prob)

    return G_prob


def discriminator(x):
    D_h1 = tf.nn.relu(tf.matmul(x, D_W1) + D_b1)
    D_logit = tf.matmul(D_h1, D_W2) + D_b2
```

```
    D_prob = tf.nn.sigmoid(D_logit)

    return D_prob


def Q(x):
    Q_h1 = tf.nn.relu(tf.matmul(x, Q_W1) + Q_b1)
    Q_prob = tf.nn.softmax(tf.matmul(Q_h1, Q_W2) + Q_b2)

    return Q_prob



G_sample = generator(Z, c)
D_real = discriminator(X)
D_fake = discriminator(G_sample)
Q_c_given_x = Q(G_sample)

# Losses

D_loss = -tf.reduce_mean(tf.log(D_real + 1e-8) + tf.log(1 - D_fake + 1e-8))
G_loss = -tf.reduce_mean(tf.log(D_fake + 1e-8))
cross_ent = tf.reduce_mean(-tf.reduce_sum(tf.log(Q_c_given_x + 1e-8) * c, 1))
Q_loss = cross_ent

D_solver = tf.train.AdamOptimizer().minimize(D_loss, var_list=theta_D)
G_solver = tf.train.AdamOptimizer().minimize(G_loss, var_list=theta_G)
Q_solver = tf.train.AdamOptimizer().minimize(Q_loss, var_list=theta_G + theta_Q)

batch_size = 32
Z_dim = 16

sess = tf.Session()
sess.run(tf.global_variables_initializer())

for it in range(1000000):
    if it % 1000 == 0:
        Z_noise = sample_Z(16, Z_dim)

        idx = np.random.randint(0, 10)
        c_noise = np.zeros([16, 10])
        c_noise[range(16), idx] = 1

        samples = sess.run(G_sample,
                           feed_dict={Z: Z_noise, c: c_noise})

    X_batch, _ = mnist.train.next_batch(batch_size)
    Z_noise = sample_Z(batch_size, Z_dim)
    c_noise = sample_c(batch_size)

    _, D_loss_curr = sess.run([D_solver, D_loss],
                              feed_dict={X: X_batch, Z: Z_noise, c: c_noise})

    _, G_loss_curr = sess.run([G_solver, G_loss],
                              feed_dict={Z: Z_noise, c: c_noise})

    sess.run([Q_solver], feed_dict={Z: Z_noise, c: c_noise})
```

On line 68, we define our Q-network, which models the auxiliary distribution $Q(c \mid x)$.

On lines 102-104, we generate a zero-vector for c and assign 1 to a random index.
In this experiment, our c will encode label property nicely, as you can see from how homogeneous the results are for certain settings of c such as $[[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]]$:



2.4 CGAN

2.4.1 Theory and architecture

Classical GANs are unconditioned generative models, which means that there is no control on modes of the data being generated.
However, by conditioning the model on additional information, it is possible to direct the data generation process.
Such conditioning could be based, for example, on class labels, or on some other characteristic of the data [7].
For example, if we are training our model on handwritten digits, we might want to tell our model which specific digit we want it to generate.
This is where Conditional GANs come into play, since they allow us to *condition* our model so that it generates a specific class of our data distribution.

The intuition is extremely simple: all we do is feed some extra information y to both our discriminator and generator.
y can be, for example, a class label.

In the generator the prior input noise $p_z(z)$, and y are combined in joint hidden representation, and the adversarial training framework allows for considerable flexibility in how this hidden representation is composed.
In the discriminator x and y are presented as inputs to a discriminative function (embodied again by a neural network).

We end up with an objective function that is much similar to the one of classic GANs:

$$\min_G \max_D V(D, G) = E_{x \sim p_{data(x)}}[\log D(x \mid \mathbf{y})] + E_{z \sim p_z(z)}[\log(1 - D(G(z \mid \mathbf{y})))]$$

The nice thing about this architecture is that y can really be anything.
As I will discuss in Chapter 4, we can even condition our GAN on another picture, so that we can learn a mapping from image to image.

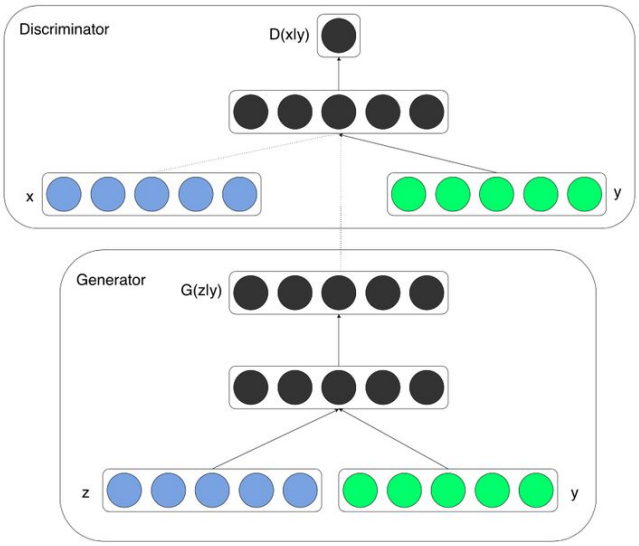


Figure 2.6. Overview of CGAN architecture.

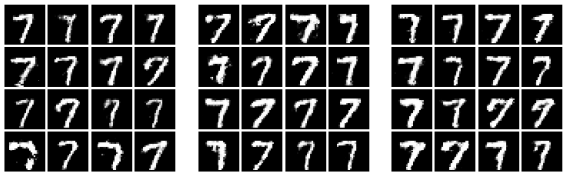
2.4.2 Implementation and results

The implementation of a simple CGAN architecture is quite straightforward.
All we have to do is add a vector y to the input of both the discriminator and the generator.

```
def generator(z,y):
    with tf.name_scope("generator"):
        x = tf.concat([z,y],axis=1)
        G_h1 = tf.nn.relu(tf.matmul(x, G_W1) + G_b1)
        G_log_prob = tf.matmul(G_h1, G_W2) + G_b2
        G_prob = tf.nn.sigmoid(G_log_prob)
    return G_prob

def discriminator(x,y):
    with tf.name_scope("discriminator"):
        x = tf.concat([x,y],axis=1)
        D_h1 = tf.nn.relu(tf.matmul(x, D_W1) + D_b1)
        D_log_prob = tf.matmul(D_h1, D_W2) + D_b2
        D_prob = tf.nn.sigmoid(D_log_prob)
    return D_prob
```

When training, we will feed the models the one-hot encoded vector for the current sample, so that the generator and discriminator condition on class labels.
After training, we can easily generate samples belonging to the class we want by feeding the corresponding one-hot encoded vector to the generator.

Here are the results if we feed the vector $[0, 0, 0, 0, 0, 0, 1, 0, 0]$ to the generator:



2.5 GAN Training issues

Nash equilibrium

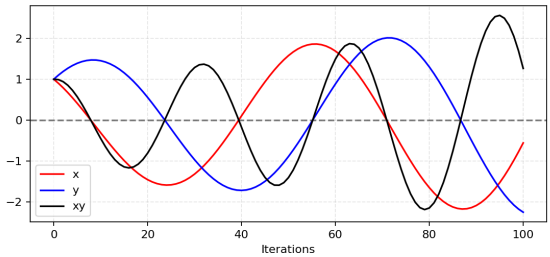
Ian J. Goodfellow *et al.* (2016) [3] discussed the problems with the training procedure of GANs.
As already mentioned in section 2.1.4 they showed that, when training GANs, two models are trained simultaneously to find a Nash equilibrium to a two-player non-cooperative game. However, each model updates its cost independently with no respect to another player in the game. Updating the gradient of both models concurrently cannot guarantee a convergence.

We can use a toy example to show why it is so difficult to find this kind of Nash equilibrium when using gradient-based methods.
Suppose we want one player x to minimize $f_1(x) = xy$, and at the same time we want player y to maximize $f_2 = -xy$.

Since $\frac{\partial f_1}{\partial x} = y$ and $\frac{\partial f_2}{\partial y} = -x$, and given learning rate η, the updates at every iterations will be the following:

$$x = x - \eta y$$
$$y = y + \eta x$$

Once x and y have different signs, every following gradient update causes huge oscillation and the instability gets worse in time, as shown in the figure below [8].



Vanishing gradient

When the discriminator is optimal or almost optimal, it will output 0, or values very close to 0, for every sample coming for p_g. This is very intuitive: an optimal discriminator will give 0 probability to samples that are not coming from p_{data} and at the very early phase of training, D is very easy to be confident in detecting G, so D will output almost always 0.

Mode collapse

Mode collapse happens when the generator generates a limited diversity of samples, or even the same sample, regardless of the input.
The name 'mode collapse' comes from the fact that, when trying to learn a multimodal distribution, the generator only outputs samples from a select number of these modes.
This happens because the optimal generator for a fixed discriminator maps every value $z \in Z$ to some $x \in X$ that the discriminator believes is most likely to be real rather than fake [9].

Towards more stable training

The first thing we have to analyze in order to understand how WGANs work are distance metrics.

I have already presented the KL-Divergence in chapter 1.
Another common metric is the Jensen–Shannon Divergence (JS-Divergence in short):

$$D_{JS}(p \,||\, q) = \frac{1}{2} D_{KL}(p \,||\, \frac{p+q}{2}) + \frac{1}{2} D_{KL}(q \,||\, \frac{p+q}{2})$$

The JS-Divergence is bounded by $[0, 1]$ and it is symmetric and smoother than KL-Divergence.

Previous GANs incorporate the JS-Divergence (and therefore the KL-Divergence) in their cost function.
To prove it, we will compute the JS-Divergence between the probability distribution of our generator, p_g, and p_{data}.

$$D_{JS}(p_{data} \,||\, p_g) = \frac{1}{2} D_{KL}(p_{data} \,||\, \frac{p_{data}+p_g}{2}) + \frac{1}{2} D_{KL}(p_g \,||\, \frac{p_{data}+p_g}{2}) =$$

$$\frac{1}{2}(\log 2 + \int_x p_{data}(x) \log \frac{p_{data}(x)}{p_{data}(x)+p_g(x)} dx) +$$

$$\frac{1}{2}(\log 2 + \int_x p_{data}(x) \log \frac{p_g(x)}{p_{data}(x)+p_g(x)} dx) =$$

$$\frac{1}{2}(\log 4 + L(G, D*))$$

which gives us

$$L(G, D*) = 2 D_{JS}(p_{data} \,||\, p_g) - 2 \log 2$$

Essentially the loss function of GAN quantifies the similarity between the generative data distribution p_g and the real sample distribution p_{data} by JS-Divergence when the discriminator is optimal. The best $G*$ that replicates the real data distribution leads to the minimum $L(G*, D*) = log4$ (as proved in Section 2.1) which is aligned with equations above.

2.6 WGAN

2.6.1 Theory and architecture

Earth Mover's distance (or Wasserstein metric)

We introduce a new metric to measure the distance of two probability distributions, called *Earth Mover's distance* (EM in short).
To get an intuitive understanding of this metric, let's start from the discrete case. We can imagine a discrete probability distribution as a way of piling up dirt over a certain region. The EM distance is then the minimum cost of turning one pile (representing the first distribution) into the pile that represents the second distribution, where the cost is the amount of dirt moved times the distance by which it is moved.

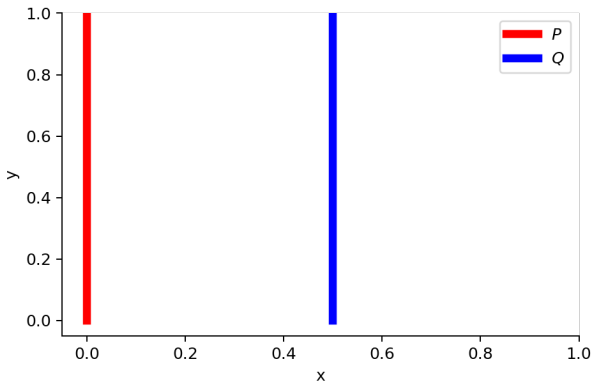
When dealing with continuous distribution, the formula for the EM distance is the following:

$$W(p,q) = \inf_{\gamma \sim \prod(p,q)} E_{(x,y)\sim\gamma}[\|x-y\|]$$

where $\prod(p,q)$ is the set of all join probability distributions between p and q. One joint distribution $\gamma \in \prod(p,q)$ describes one way of moving the dirt, but in a continous space.

I will now use a toy example to show why the EM distance makes a better loss function than KL-Divergence or JS-Divergence, as shown by Martin Arjovsky *et al.* [10]

Suppose we have two distributions, P and Q, and that $\forall(x,y) \in P, x = 0$ and $y \sim U(0,1)$ and $\forall(x,y) \in Q, x = \theta, 0 \leq \theta \leq 1$ and $y \sim U(0,1)$.



When $\theta \neq 0$:

$$D_{KL}(P \,||\, Q) = \sum_{x=0, y\sim U(0,1)} 1 \cdot \log \frac{1}{0} = +\infty$$

$$D_{KL}(Q \,||\, P) = \sum_{x=\theta, y\sim U(0,1)} 1 \cdot \log \frac{1}{0} = +\infty$$

$$D_{JS}(P \mid\mid Q) = \frac{1}{2}(\sum_{x=\theta, y\sim U(0,1)} 1 \cdot \log \frac{1}{\frac{1}{2}}) + \frac{1}{2}(\sum_{x=\theta, y\sim U(0,1)} 1 \cdot \log \frac{1}{\frac{1}{2}}) = \log 2$$

$$W(P, Q) = |\theta|$$

However, when $\theta = 0$ (and thus the two distributions are fully overlapped):

$$D_{KL}(P \mid\mid Q) = D_{KL}(Q \mid\mid P) = D_{JS}(P \mid\mid Q) = 0$$

$$W(P, Q) = |\theta| = 0$$

The KL-Divergence gives us infinity when two distributions are disjoint. The value of the JS-divergence has a sudden jump, and is not differentiable at $\theta = 0$. Only the Wasserstein metric provides a smooth measure, which is extremely helpful for a stable learning process using gradient descent [8].

EM distance as a loss function

The formula for the Wasserstein metric is computationally intractable, since it is unfeasible to exhaust all the possible joint distributions in $\prod(p_{data}, p_g)$ to compute $\inf_{\gamma\sim\prod(p_{data},p_g)}$.
The original paper shows an approximation, resulting from the Kantorovich-Rubinstein duality, that makes the problem tractable.
It shows that

$$W(p_{data}, p_g) = \sup_{\|f\|_L \leq K} E_{x\sim p_{data}}[f(x)] - E_{x\sim p_g}[f(x)]$$

where the *supremum* is taken over all **K-Lipschitz** functions.

Lipschitz continuity

The function f in the new form of the Wasserstein metric is demanded to satisfy $\|f\|_L \leq K$, meaning it should be **K-Lipschitz** continuous.

A real-valued function $f : \mathbb{R} \rightarrow \mathbb{R}$ is called K-Lipschitz continuous if there exists a real constant $K \geq 0$ such that, for all $x_1, x_2 \in \mathbb{R}$,

$$|f(x_1) - f(x_2)| \leq |x_1 - x_2|$$

Intuitively, this means that the derivative of f should be bounded.
Functions that are everywhere continuously differentiable are thus Lipschitz continuous.

The *supremum* in the equation above is still highly intractable, but we can now approximate it with the *maximum* of f:

$$\max_{w\in W} E_{x\sim p_{data}}[f_w(x)] - E_{x\sim p_g}[f_w(x)] \leq \sup_{\|f\|_L \leq K} E_{x\sim p_{data}}[f(x)] - E_{x\sim p_g}[f(x)]$$

where f is parametrized by some weights w, and the maximum is taken over the set of all possible weights W.

The discriminator is now in charge of computing our Lipschitz continuous function, to help compute the Wasserstein metric.
As the loss function decreases in the training, the Wasserstein distance gets smaller and the generator model's output grows closer to the real data distribution.
However, this entire derivation only works when the function family $\{f_w\}_{w \in \mathcal{W}}$ is K-Lipschitz.
To guarantee this is true, we use weight clamping: the weights w are constrained to lie within $[-c, c]$, by clipping w after every update.

2.6.2 Implementation and results

This is the full algorithm for the WGAN framework:

Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α, the learning rate. c, the clipping parameter. m, the batch size. n_{critic}, the number of iterations of the critic per generator iteration.
Require: : w_0, initial critic parameters. θ_0, initial generator's parameters.

while θ has not converged **do**
  for $t = 0, ..., n_{\text{critic}}$ **do**
    Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data.
    Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
    $g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))\right]$
    $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
    $w \leftarrow \text{clip}(w, -c, c)$
  end for
  Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
  $g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$
  $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
end while

Practically, all we have to do is change our loss function by removing the logarithms, remove the sigmoid activation from the output of our discriminator, and clip the weights on every iteration of the training loop.

Here is the commented Tensorflow implementation:

```
batch_size = 32
X_dim = 784
z_dim = 10
h_dim = 128


def xavier_init(size):
    in_dim = size[0]
    xavier_stddev = 1. / tf.sqrt(in_dim / 2.)
    return tf.random_normal(shape=size, stddev=xavier_stddev)
```

```
11
12 #Placeholder for the image input.
13 X = tf.placeholder(tf.float32, shape=[None, X_dim])
14
15 #Disciminator parameters
16 D_W1 = tf.Variable(xavier_init([X_dim, h_dim]))
17 D_b1 = tf.Variable(tf.zeros(shape=[h_dim]))
18
19 D_W2 = tf.Variable(xavier_init([h_dim, 1]))
20 D_b2 = tf.Variable(tf.zeros(shape=[1]))
21
22 theta_D = [D_W1, D_W2, D_b1, D_b2]
23
24 #Input noise
25 z = tf.placeholder(tf.float32, shape=[None, z_dim])
26
27 # Generator parameters
28 G_W1 = tf.Variable(xavier_init([z_dim, h_dim]))
29 G_b1 = tf.Variable(tf.zeros(shape=[h_dim]))
30
31 G_W2 = tf.Variable(xavier_init([h_dim, X_dim]))
32 G_b2 = tf.Variable(tf.zeros(shape=[X_dim]))
33
34 theta_G = [G_W1, G_W2, G_b1, G_b2]
35
36 # We use this to sample the Gaussian input variables for the generator.
37 def sample_z(m, n):
38     return np.random.uniform(-1., 1., size=[m, n])
39
40
41 # The generator networks is the same as in standard GANs.
42 def generator(z):
43     G_h1 = tf.nn.relu(tf.matmul(z, G_W1) + G_b1)
44     G_log_prob = tf.matmul(G_h1, G_W2) + G_b2
45     G_prob = tf.nn.sigmoid(G_log_prob)
46     return G_prob
47
48 # Disciminator network.
49 # We have removed the sigmoid output activation since we don't want to output a probability anyr
50 def discriminator(x):
51     D_h1 = tf.nn.relu(tf.matmul(x, D_W1) + D_b1)
52     out = tf.matmul(D_h1, D_W2) + D_b2
53     return out
54
55
56 G_sample = generator(z)
57 D_real = discriminator(X)
58 D_fake = discriminator(G_sample)
59
60 #Loss functions
61 D_loss = tf.reduce_mean(D_real) - tf.reduce_mean(D_fake)
62 G_loss = -tf.reduce_mean(D_fake)
63
64
65 # In the original paper, RMSProp is suggested as optimization algorithm.
66 D_solver = (tf.train.RMSPropOptimizer(learning_rate=1e-4)
67             .minimize(-D_loss, var_list=theta_D))
```

```
G_solver = (tf.train.RMSPropOptimizer(learning_rate=1e-4)
            .minimize(G_loss, var_list=theta_G))

# We will use this to clip the weights at every iteration of the training loop.
clip_D = [p.assign(tf.clip_by_value(p, -0.01, 0.01)) for p in theta_D]

sess = tf.Session()
sess.run(tf.global_variables_initializer())

for it in range(1000000):

    # 5 iterations of D optimization, 1 iteration of G optimization.
    for _ in range(5):
        X_batch, _ = mnist.train.next_batch(batch_size)

        # Optimize D
        _, D_loss_curr, _ = sess.run(
            [D_solver, D_loss, clip_D],
            feed_dict={X: X_batch, z: sample_z(batch_size, z_dim)}
        )
    # Optimize G
    _, G_loss_curr = sess.run(
      [G_solver, G_loss],
      feed_dict={z: sample_z(batch_size, z_dim)}
    )
```

2.6.3 Comparisons with standard GANs

In standard GANs, the discriminator output a probability $p \in (0, 1)$.
In WGANs, the discriminator outputs a score. The higher the score, the higher the confidence that the input comes from the real data distribution. This is why the discriminator was renamed *critic* by the authors of the original paper.

Another big difference between standard GANs and WGANs is in the way the discriminator and generator are trained together.
The fact that the EM distance is continuous and differentiable amost everywhere means that we can (and should) train the critic till optimality. For the JS-divergence that appears in standard GANs, as the discriminator gets better the gradients get more reliable but the true gradient is 0 since the JS is locally saturated and we get vanishing gradients.

So, while it was extremely hard to balance the the training of the discriminator and generator together in standard GANs (adding to the instability of the process), we can now train the critic to its optimum, without affecting negatively the learning process of the model.

2.7 EBGAN

EBGAN stands for *Energy-based* GANs.
In EGBANs, we view the discriminator as an energy function that attributes low energies to the regions near the data manifold and higher energies to other regions [13]. Using energy functions instead of a probabilistic models allows for the use of different loss functions and

architectures.
For some choices of losses and architectures, it was shown by Junbo Zhao *et al.* that EBGAN are more stable than standard GANs during training.

The essence of the energy-based model (LeCun et al., 2006) is to build a function that maps each point of an input space to a single scalar, which is called "energy". The learning phase is a data-driven process that shapes the energy surface in such a way that the desired configurations get assigned low energies, while the incorrect ones are given high energies. This means that the discriminator should assign low energy to samples coming from p_{data} and high energy to samples from the generator distribution.

2.7.1 Theory and architecture

Loss function

Like in standard GANs, EBGANs use different loss functions for the generator and the discriminator.
Given a positive margin m, the loss function are:

$$L_D(x, z) = D(x) + [m - D(G(z))]^+$$

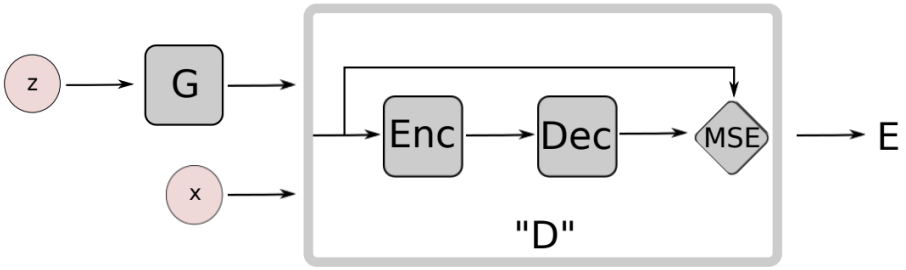
$$L_G(z) = D(G(z))$$

where $[\cdot]^+ = \max(0, \cdot)$.

Using Autoencoders

in EBGANs, the discriminator is structures as an autoencoder (see figure below), and its output is the reconstruction loss (Mean Squared Error):

$$D(x) = \|Dec(Enc(x)) - x\|$$



2.7.2 Implementation and results

To implement an EBGAN, we need to change the discriminator architecture, as well as the loss function.

The discriminator becomes a simple autoencoder architecture, with the output layer computing the mean squared error (often referred as reconstruction error) between the input and the output:

```
def discriminator(x):
    with tf.variable_scope("discriminator",reuse=tf.AUTO_REUSE):
        enc = tf.layers.dense(x, 128, activation=tf.nn.relu)
        dec = tf.layers.dense(enc, 784)
        mse = tf.reduce_mean(tf.reduce_sum((x - dec)**2, 1))
    return mse
```

The loss becomes:

```
m=5

G_sample = generator(Z)
D_real = discriminator(X)
D_fake = discriminator(G_sample)
D_loss = D_real + tf.maximum(0., m-D_fake)
G_loss = D_fake
```

with m as hyperparameter.

Below are some sample generated with an EBGAN.

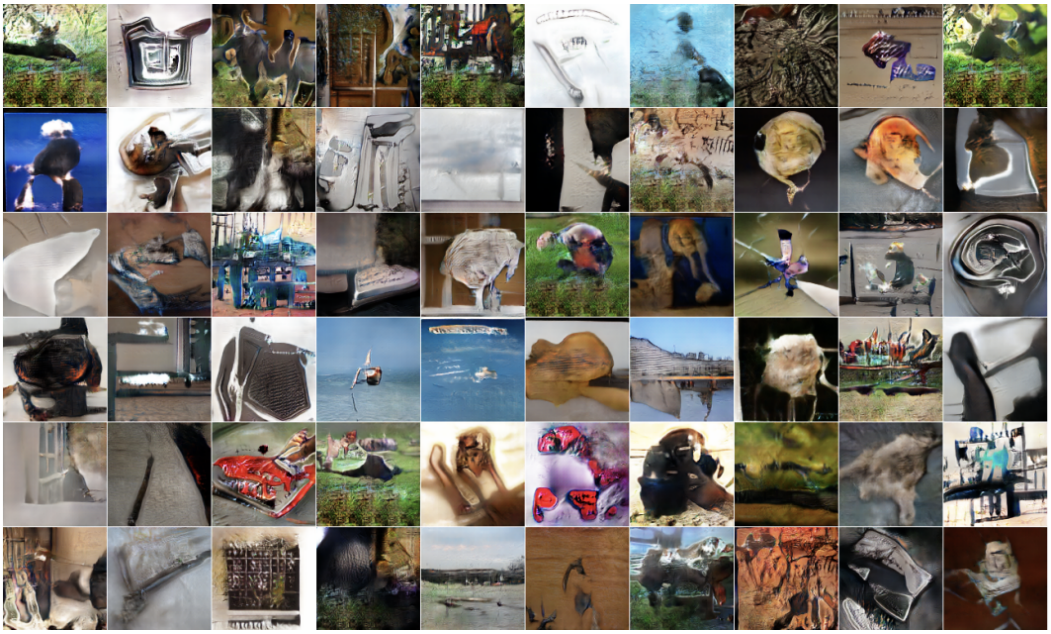


Figure 2.7. ImageNet 128 x 128 generation using EBGAN,

Figure 2.8. ImageNet 256 x 256 generation using EBGAN.

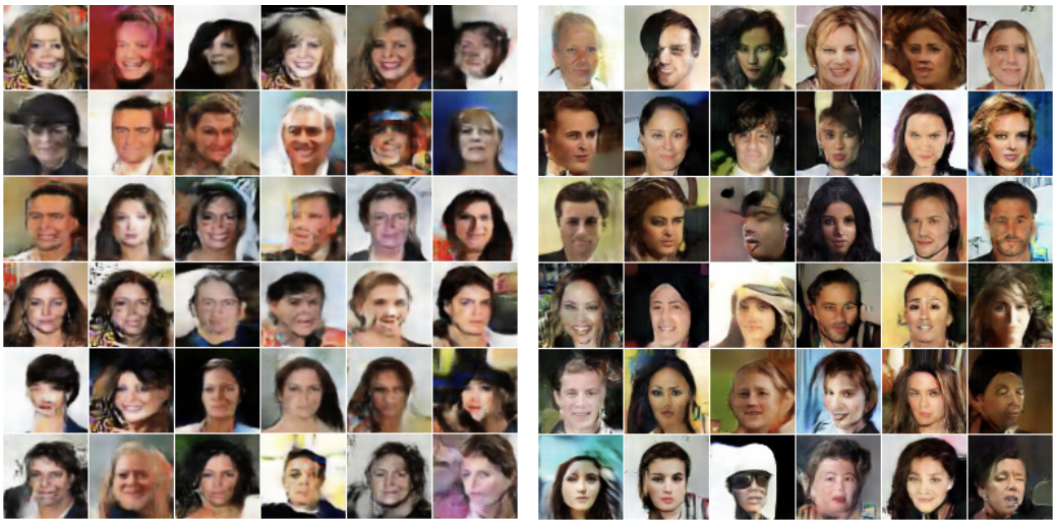


Figure 2.9. CelebA faces generation using EBGAN.

3

Variational Autoencoders

Variational Autoencoders where introduced to "*perform efficient inference and learning in directed probabilistic models, in the presence of continuous latent variables with intractable posterior distributions, and large datasets*". [11]
We have already discussed the fundamental concepts of variational inference in chapter 1. They will be the building blocks for a probabilistic description of how VAEs work.

3.0.1 Maximum Likelihood framework

As usual, we have some data X that we want to be able to generate without supervision, which means that we need to somehow capture its distribution.
VAEs use a latent variable model. Recall that latent variable are variables that we don't observe from data, but that give us useful information about it (see Section 1.4 for more details).
Before we can say that our model is representative of our dataset, we need to make sure that there exist settings of our latent variable that cause the model to reproduce something similar to X.
Formally, say that we have a vector of random variables z in some high-dimensional space Z, that we can sample according to some probability density function $P(z)$ defined over Z. Then, say we have a family of deterministic functions $f(z;\theta)$, parameterized by a vector θ in some space Θ, where $f : Z \times \Theta \to X$.
We wish to optimize θ such that, if we sample z from $P(z)$, $f(z;\theta)$ will be like the X in our dataset.

We can formulate this objecive in a maximum likelihood framework, by saying that we want to maximize $P(x)$, defined as

$$P(x) = \int P(X \mid z;\theta)P(z)dz \tag{3.1}$$

where $f(z;\theta)$ has been replaced by a distribution $P(X \mid z;\theta)$, which allows us to make the dependence of X on z explicit by using the law of total probability.
The intuition is that, if our model is likely to produce training set samples (by maximizing $P(X)$, it is also likely to produce similar samples, which is our end goal.
To formalize this intuition, we say that the output distribution should be, for example, Gaussian: $P(X \mid z;\theta) = \mathcal{N}(X \mid f(z;\theta), \sigma^2 * I)$

In general, our model will produce output that are similar but not identical to some particular X. By having a Gaussian distribution, we can use gradient descent (or any other optimization technique) to increase $P(X)$ by making $f(z;\theta)$ approach X for some z, i.e., gradually making the training data more likely under the generative model. [12]

3.0.2 Variational lower bound

Firstly, we need to decide which information we want our latent variables z to capture.
In the VAE framework, we simply assume that there is no simple interpretation of z that we can decide by hand (ex: angle of a digit, stroke width etc.), and we state that samples of z can me drawn from a normal distribution $\mathcal{N}(0, I)$ where I is the identity matrix.
This technique works because we know that any distribution in n dimensions can be generated by mapping a set of n normally distributed variables through a sufficiently complicated function (see Appendix A).

In practice, for most z, $P(X \mid z)$ will be nearly zero, and hence contribute almost nothing to our estimate of $P(X)$. The key idea behind the variational autoencoder is to attempt to sample values of z that are likely to have produced X, and compute $P(X)$ just from those. This means that we need a new function $Q(z \mid X)$ which can take a value of X and give us a distribution over z values that are likely to produce X. Hopefully the space of z values that are likely under Q will be much smaller than the space of all z's that are likely under the prior $P(z)$.

Now, in order to work with Equation 3.1, we first need to relate $E_{z\sim Q}P(X \mid z)$ with $P(X)$.
We can work this out starting with the definition of KL-Divergence between $P(z \mid X)$ and $Q(z)$ for some arbitrary Q.
Remember we have already discussed KL-Divergence in Chapter 1.

$$D_{KL}[Q(z) \mid\mid P(z \mid X)] = E_{z\sim Q}[\log Q(z) - \log P(z \mid X)] \tag{3.2}$$

We can now get $E_{z\sim Q}P(X \mid z)$ and $P(X)$ into this equation by applying Bayes rule.

$$D_{KL}[Q(z) \mid\mid P(z \mid X)] = E_{z\sim Q}[\log Q(z) - \log P(X \mid z) - \log P(z)] + \log P(X) \tag{3.3}$$

with $P(X)$ out of the expectation because it does not depend on z.

We can rearrange this equation in this way:

$$\log P(X) - D_{KL}[Q(z) \mid\mid P(z \mid X)] = E_{z\sim Q}[\log P(X \mid z)] - D_{KL}[\log Q(z) \mid\mid \log P(z)] \tag{3.4}$$

One last detail: Since we want $Q(z)$ to make a good job mapping X to zs that can reproduce X, we make the dependence on X explicit for Q:

$$\log P(X) - D_{KL}[Q(z \mid X) \mid\mid P(z \mid X)] = E_{z\sim Q}[\log P(X \mid z)] - D_{KL}[\log Q(z \mid X) \mid\mid \log P(z)] \tag{3.5}$$

This equation is extremely important, and we should take a closer look at it.

The left side of the equation is made by the quantity we want to maximize $\log P(X)$, and an error term given by the KL-Divergence.
The right hand side, instead, is something that we can optimize, as we will see.

Optimizing the objective

So, how can we optimize the right hand side of the equation using gradient descent?
The elements we have are:

- $Q(z|X)$ that project our data X into latent variable space
- z, the latent variable
- $P(X|z)$ that generate data given latent variable

This looks a lot like an autoencoder, where Q encodes the input into latent variable space, and P decodes it to data space, and this is in fact where the name *variational autoencoder* comes from.

If we see this framework as an autoencoder, we understand how we can implement $P(X \mid z)$ with a neural network. But what about $D_{KL}[\log Q(z \mid X) \mid\mid \log P(z)]$?
As stated at the beginning of this section, the easiest choice is to sample of z from a normal distribution $\mathcal{N}(0, I)$.

What about $Q(z \mid X)$?
The usual choice is to say that $Q(z \mid X) = (z \mid \mu(X), \sum(X))$, where μ and $\sum$ are arbitrary deterministic functions that can be learned from data.

Choosing Gaussian distributions is particularly wise: we can in fact compute the KL-Divergence of two multivariate Gaussian distributions in closed form.

$$\begin{aligned} &D_{KL}[\mathcal{N}(\mu(x), \sum(x)) \mid\mid \mathcal{N}(0, I)] = \\ &\tfrac{1}{2}(tr(\sum(X)) + (\mu(X))^T(\mu(X)) - k - \log\det(\sum(X))) \end{aligned} \tag{3.6}$$

This makes it possible to compute the gradient of the right hand side of equation 3.5, which means that we can use gradient descent to optimize our objective.

The reparametrization trick

However, we are now facing a problem. How do we get z from the encoder outputs? Obviously we could sample z from a Gaussian for which parameters are the outputs of the encoder.
However, the sampling operation is not a differentiable one. This means that we can't backpropagate through a layer that performs sampling.
The only way to solve this is to somehow move the sampling operation outside of the network, so that we don't need to backpropagate through it.

The solution, called the *reparameterization trick* is to move the sampling to an input layer.

Given $\mu(X)$ and $\Sigma(X)$— the mean and covariance of $Q(z \mid X)$ — we can sample from $\mathcal{N}(\mu(X), \Sigma(X))$ by first sampling $\epsilon \sim \mathcal{N}(0, I)$, then computing $z = \mu(X) + \Sigma^{\frac{1}{2}}(X) \cdot \epsilon$.

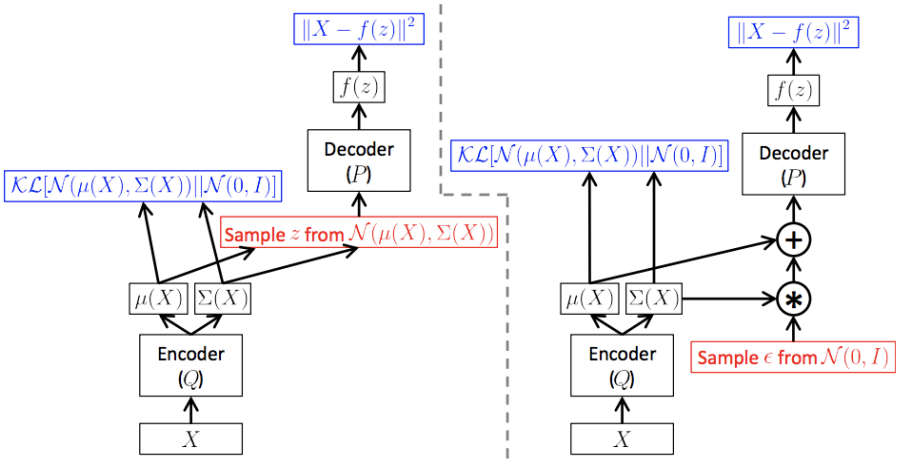


Figure 3.1. Left: VAE without the reparametrization trick. The red operation (sampling) is not differentiable, and is inside the network.
Right: VAE with the reparametrization trick. The sampling layer is now out of the network, so that we can backpropagate.

To summarize, here is the equation of which we take the gradient to optimize our objective:

$$E_{X \sim D}\Big[E_{\epsilon \sim \mathcal{N}(0,I)}[\log P(X \mid z = \mu(X) + \Sigma\frac{1}{2}(X) \cdot \epsilon)] - D_{KL}[\log Q(z \mid X) \mid\mid \log P(z)]\Big] \tag{3.7}$$

Summary

In short, this is what we have done: we stated the problem of capturing the data distribution in terms of a maximum likelihood problem conditioned on some latent variables z (Equation 3.1).
We then used variational inference (Equations 3.2, 3.3, 3.4) to express the quantity we want to maximize in term of $P(X \mid z)$ (our **decoder**), and a KL-Divergence.
Then, by forcing simple, Gaussian distributions on the latent variables and on the conditional distribution $Q(z \mid X)$ (our **encoder**), we made the optimization problem computationally tractable (and differentiable).
We can now use gradient descent to maximize $\log P(X)$, which is our objective.

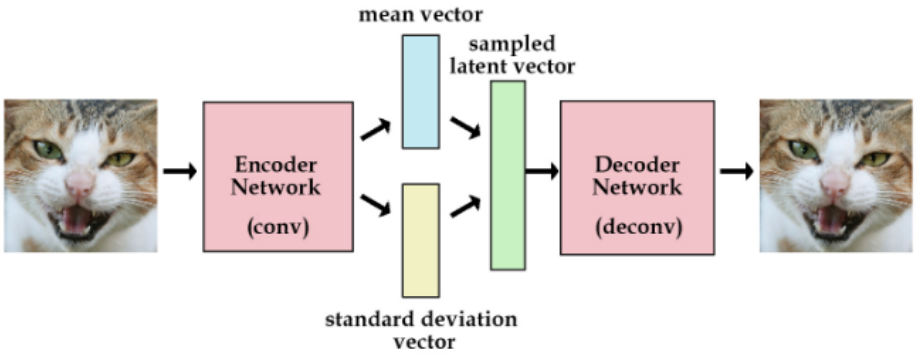


Figure 3.2. VAE architecture

3.0.3 Implementations and results

The first thing we need to implement is the encoder network, which corresponds to $Q(z \mid x)$. It should output a vector of means and a vector of variances (it is theoretically a diagonal matrix, which we can represent as a vector) for our latent, multivariate Gaussian distribution.

```
def Q(x):
    with tf.name_scope("encoder"):
        with tf.variable_scope("encoder", initializer=tf.contrib.layers
        .xavier_initializer(), reuse=tf.AUTO_REUSE):
            x = tf.layers.dense(x, H_DIM, activation=tf.nn.relu)
            mu = tf.layers.dense(x, LATENT_DIM)
            sigma = tf.layers.dense(x, LATENT_DIM)

    return mu, sigma
```

Next, we need to implement the reparametrization trick.

```
def sample_z(mu, sigma):
    eps = tf.random_normal(shape=tf.shape(mu))
    return mu + tf.exp(sigma / 2) * eps
```

The decoder network is the same as in standard GANs:

```
def P(z):
    with tf.name_scope("decoder"):
        with tf.variable_scope("decoder", initializer=tf.contrib.layers.xavier_initialize
            x = tf.layers.dense(z, H_DIM, activation = tf.nn.relu)
            logits = tf.layers.dense(x, IMG_DIM)
            prob = tf.nn.sigmoid(logits)

    return logits, prob
```

In the loss, we also want to minimize the expression for the KL-Divergence in closed form:

```
# Losses
mu, sigma = Q(X)
z_sample = sample_z(mu,sigma)
logits, _ = P(z_sample)

# Sampling from random z
X_samples, _ = P(z)

# E[log P(X|z)]
recon_loss = tf.reduce_sum(tf.nn.sigmoid_cross_entropy_with_logits(logits=logits, labels=X), 1)
KL_loss =  0.5 * tf.reduce_sum(tf.exp(sigma) + mu**2 - 1. - sigma, 1)
loss = tf.reduce_mean(recon_loss + KL_loss)
```

Here are some of the samples generated by a variational autoencoder trained on the MNIST dataset:

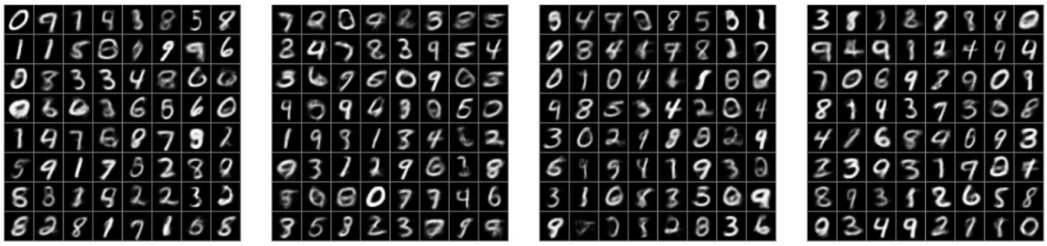


Figure 3.3. MNIST digits generated with a VAE.

4

Other applications

4.1 Image-to-image translation

Image-to-image translation refers to the process of mapping certain images to certain transformation of those images.
Examples of this are:

- mapping a black-and-white picture to its colored version
- transforming a geographical map into an aerial photo
- mapping the edges of an image into the original image
- mapping an image with some missing pixels to the original image

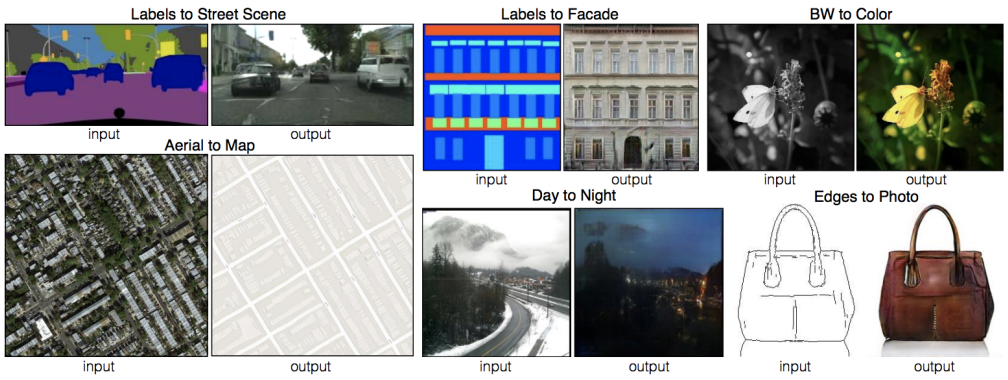


Figure 4.1. Examples of applications of image-to-image translation.

4.1.1 Theory and architecture

In previous attempts of image-to-image translation, simple CNN were commonly used. However, CNNs learn to minimize a loss function – an objective that scores the quality of results – and although the learning process is automatic, a lot of manual effort still goes into designing effective losses.
In other words, we still have to tell the CNN what we wish it to minimize. [14]

For example, if we simply use Euclidean distance between ground truth pixels and predicted result, the results tend to be blurry.
This is were GANs come into play for this task. The great advantage here is that with GANs, one only specifies a high-level goal, like “make the output indistinguishable from reality”, and the GAN automatically learn a loss function appropriate for satisfying this goal.

In this context, experiments were made using CGANs (Section 2.4), and they proved to be extremely successful as a general framework for image-to-image translation.

The CGANs are simply conditioned on the input image, and the discriminator learns to tell whether a sample comes from the generator or from ground truth.

The loss function of a CGAN can be expressed as:

$$L_{CGAN}(G, D) = E_{x,y}[\log D(x, y)] - E_{x,z}[1 - \log D(x, G(x, z))]$$

The authors of the original papers also decided to mix the GAN loss with a more traditional loss.
The best choice proved to be the standard $L1$ loss, which proved to encourage less blurring than $L2$ loss.

$$L_{L1}(G) = E_{x,y,z}[\|y - G(x, z)\|_1]$$

Thus, the final objective is

$$G^* = \arg\min_G \max_D L_{CGAN}(G, D) + \lambda L_{L1}(G)$$

The generator

The generator is an encoder-decoder architecture, which has previously been used for different approaches to this problem.
However, the best choice for the generator proved to be a U-Net model. [17].
It is quite similar to an autoencoder in the sense that it has several downsampling layers followed by a bottleneck layers, followed by several upsampling layers. The difference is in the fact that the U-Net uses skip connections to pass information between the input and output. This can be quite useful, since the image input and the generated one can share several feature such as the location of the edges.

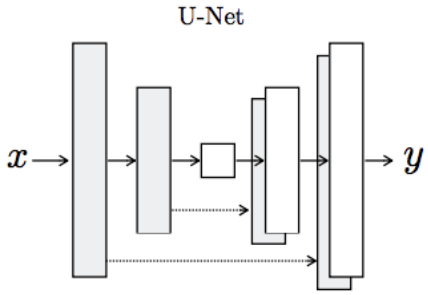


Figure 4.2. U-Net architecture: encoder-decoder with skip connections.

The discriminator

While the $L1$ loss is used to ensure high-frequency correctness (pixel-by-pixel), the discriminator is used to ensure lower frequency correctness, working on a wider scale.
This new model is called *PatchGAN*. It tries to classify if each N × N patch in an image is real or fake. The discriminator runs convolutionally across the image, averaging all responses to provide the ultimate output of D.

4.1.2 Implementation and results

Training this kind of model requires a great deal of computational energy, and it is practically unfeasible to run on a CPU.
However, the authors of the original papers released a software library that can be used to apply this model to any dataset of our choosing. All the information is available at `https://phillipi.github.io/pix2pix/`.
Here are some great results:

Figure 4.3. Results with different losses.

Figure 4.4. Map to aerial photo.

Figure 4.5. Sketch to picture.

Figure 4.6. Image-to-image for style transfer

5

Conclusions and future work

Generative models are a rapidly advancing area of research. As we continue to advance these models and scale up the training and the datasets, we can expect to eventually generate samples that depict entirely plausible scenarios in many different fields of applications.
The technology is fairly new, which means that there are several issues that need to be solved in order to scale these models properly. The main problem, as we've seen, is training instability and lack of certainty about training convergence.
Nevertheless, in recent papers researchers were able to apply these models to a large set of problems, and to generate incredible results, raising the expectation for the future of this field of research.
As an example, very recently a team at NVIDIA was able to perform incredibly realistic video-to-video translation [18], which was taught to be an almost impossible task with curren knowledge and resources.
Other successful applications of these models come, for example, from drugs research, where a variation of GANs called Adversarial Autoencoders (AAE) [20] was trained to generate new molecules with specific properties to use in new drugs and treatments [19].
Other crucial applications of GANs are, and will be, related to outlier detection (tumor detection is one example) and the generation of environments for agents in reinforcement learning.
Personally, I am confident that generative models will be crucial in solving a great amount of real-world problems and will have a wider and wider range of applications as they reach higher stability.
I am also convinced that they will play a central role in AI research in the coming years, and more in general in the future of science and technology.

Bibliography

[1] Ian J. Goodfellow *et al*: Generative Adversarial Networks
https://arxiv.org/abs/1406.26611

[2] https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms

[3] Ian J. Goodfellow *et al*: Improved Techniques for Training GANs
https://arxiv.org/abs/1606.03498

[4] https://www.countbayesie.com/blog/2017/5/9/kullback-leibler-divergence-explained

[5] Alec Radford Luke Metz, Soumith Chintala: Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks
https://arxiv.org/pdf/1511.06434.pdf

[6] https://www.inference.vc/infogan-variational-bound-on-mutual-information-twice/

[7] Mehdi Mirza, Simon Osindero: Conditional Generative Adversarial Nets
https://arxiv.org/pdf/1511.06434.pdf

[8] https://lilianweng.github.io/lil-log/2017/08/20/from-GAN-to-WGAN.htmlkullbackleibler-and-jensenshannon-divergence

[9] S. A. Barnett: Convergence Problems with Generative Adversarial Networks (GANs)
https://arxiv.org/pdf/1806.11382.pdf

[10] Martin Arjovsky *et al.*: Wasserstein GAN
https://arxiv.org/pdf/1701.07875.pdf

[11] Diederik P. Kingma, Max Welling: Auto-Encoding Variational Bayes
https://arxiv.org/pdf/1312.6114.pdf

[12] Carl Doersch: Tutorial on Variational Autoencoders
https://arxiv.org/pdf/1606.05908.pdf

[13] Junbo Zhao, Michael Mathieu and Yann LeCun: Energy-based Generative Adversarial Networks
https://arxiv.org/pdf/1609.03126.pdf

[14] Phillip Isola *et al.*: Image-to-Image Translation with Conditional Adversarial Networks
https://arxiv.org/pdf/1611.07004.pdf

[15] Sergey Ioffe, Christian Szegedy: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.
https://arxiv.org/pdf/1502.03167.pdf

[16] Christian Szegedy *et al*: Rethinking the Inception Architecture for Computer Vision
https://arxiv.org/pdf/1512.00567.pdf

[17] Olaf Ronneberger, Philipp Fischer, Thomas Brox: U-Net: Convolutional Networks for Biomedical Image Segmentation
https://arxiv.org/pdf/1505.04597.pdf

[18] Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Guilin Liu, Andrew Tao1, Jan Kautz, Bryan Catanzaro: Video-to-Video Synthesis
https://tcwang0509.github.io/vid2vid/paper$_v id2vid.pdf$

[19] Artur Kadurin, Sergey Nikolenko, Kuzma Khrabrov, Alex Aliper, and Alex Zhavoronkov: druGAN: An Advanced Generative Adversarial Autoencoder Model for de Novo Generation of New Molecules with Desired Molecular Properties in Silico
https://pubs.acs.org/doi/10.1021/acs.molpharmaceut.7b00346

[20] Alireza Makhzani, Jonathon Shlens, Navdeep Jaitly, Ian Goodfellow, Brendan Frey: Adversarial Autoencoders
https://arxiv.org/pdf/1511.05644.pdf

[21] Michael Nielsen: Neural Networks and Deep Learning
http://neuralnetworksanddeeplearning.com/

Appendix A

Example of generating complex distributions from a simple multivariate Gaussian

Here, I want to give an intuition on the fact that any distribution in n dimensions can be generated by mapping a set of n normally distributed variables through a sufficiently complicated function.
This concept is extremely important, since it poses the basis for why variational inference is effective.
I'll draw a simple example from [12].
Say we wanted to construct a 2D random variable whose values lie on a ring.
If z is 2D and normally distributed, $g(z) = z/10 + z/\|z\|$ is roughly ring-shaped, as shown in the figure below.

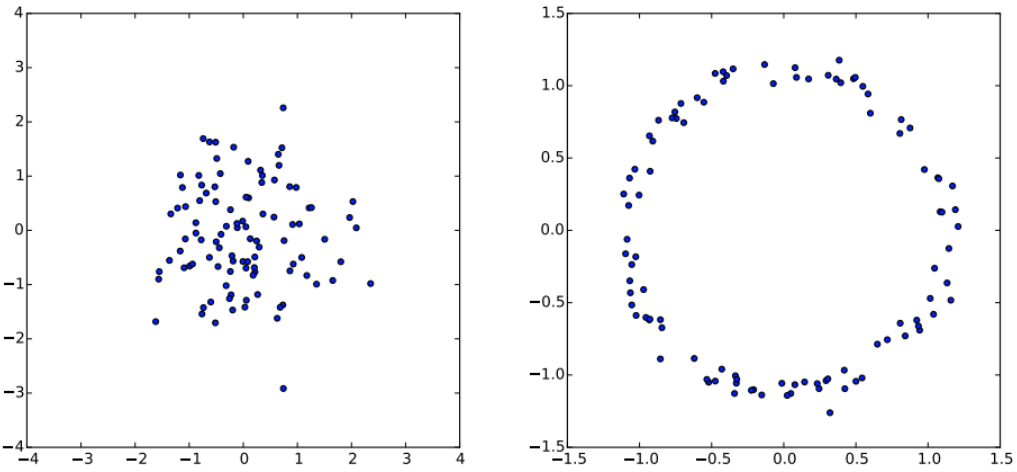


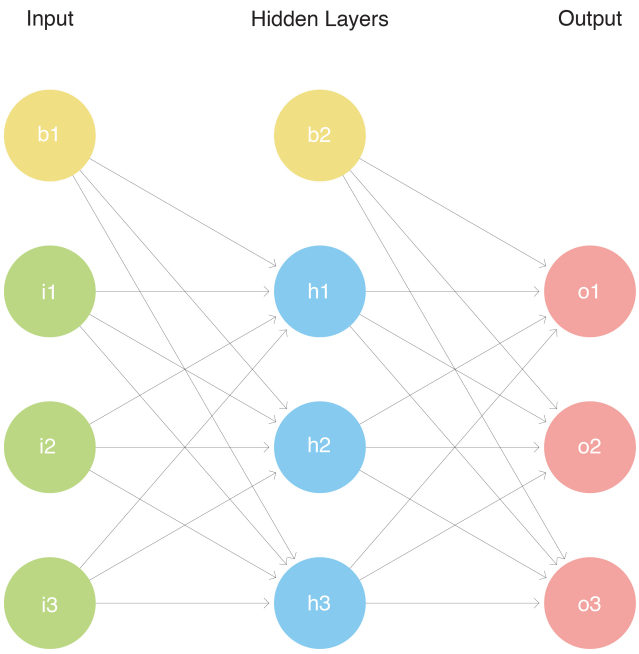
Figure A.1. Left: samples from a Gaussian distribution. Right: those same samples mapped through the function $g(z) = z/10 + z/\|z\|$ to form a ring.

Appendix B

Full iteration of the backpropagation algorithm

The Network

We are going to work on a feedforward neural network with 3 layers.
Each layer consists of 3 neurons, plus the biases.



Notation

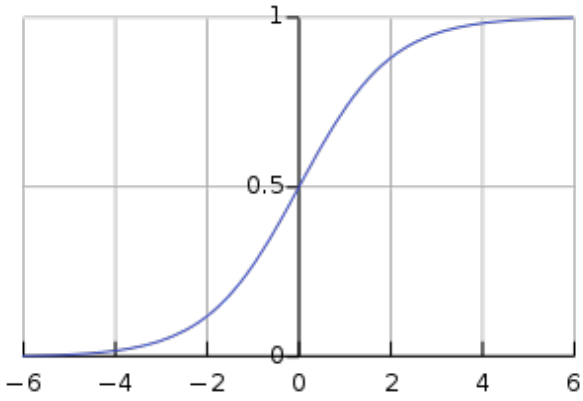
- w_{i_k,h_j} represents the weight of the edge linking the *k*th neuron in layer *i* to the *j*th neuron in layer *h*.
- net_{h_i} represents the *i*th input of layer *h*, before applying the activation function.

- $\boldsymbol{out}_{h_i}$ represents the activated value of the ith neuron in layer h.

Forward pass

For the forward pass, all we need to do is multiply the weight matrices by the input vector, and add the biases. We then apply the activation function. As our activation function, we use the sigmoid.

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$



$$\begin{bmatrix} w_{i_1,h_1} & w_{i_2,h_1} & w_{i_3,h_1} \\ w_{i_1,h_2} & w_{i_2,h_2} & w_{i_3,h_2} \\ w_{i_1,h_3} & w_{i_2,h_3} & w_{i_3,h_3} \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} + \begin{bmatrix} b_{1,1} \\ b_{1,2} \\ b_{1,3} \end{bmatrix} = \begin{bmatrix} net_{h_1} \\ net_{h_2} \\ net_{h_3} \end{bmatrix}$$

$$\begin{bmatrix} out_{h_1} \\ out_{h_2} \\ out_{h_3} \end{bmatrix} = sigmoid(\begin{bmatrix} net_{h_1} \\ net_{h_2} \\ net_{h_3} \end{bmatrix})$$

$$\begin{bmatrix} w_{h_1,o_1} & w_{h_2,o_1} & w_{h_3,o_1} \\ w_{h_1,o_2} & w_{h_2,o_2} & w_{h_3,o_2} \\ w_{h_1,o_3} & w_{h_2,o_3} & w_{h_3,o_3} \end{bmatrix} \begin{bmatrix} out_{h_1} \\ out_{h_2} \\ out_{h_3} \end{bmatrix} + \begin{bmatrix} b_{2,1} \\ b_{2,2} \\ b_{2,3} \end{bmatrix} = \begin{bmatrix} net_{o_1} \\ net_{o_2} \\ net_{o_3} \end{bmatrix}$$

$$\begin{bmatrix} out_{o_1} \\ out_{o_2} \\ out_{o_3} \end{bmatrix} = sigmoid(\begin{bmatrix} net_{o_1} \\ net_{o_2} \\ net_{o_3} \end{bmatrix})$$

Let's put the numbers in:

$$\begin{bmatrix} w_{i_1,h_1} & w_{i_2,h_1} & w_{i_3,h_1} \\ w_{i_1,h_2} & w_{i_2,h_2} & w_{i_3,h_2} \\ w_{i_1,h_3} & w_{i_2,h_3} & w_{i_3,h_3} \end{bmatrix} = \begin{bmatrix} 0.45 & 0.17 & 0.79 \\ 0.14 & 0.82 & 0.33 \\ 0.37 & 0.62 & 0.34 \end{bmatrix}$$

$$\begin{bmatrix} w_{h_1,o_1} & w_{h_2,o_1} & w_{h_3,o_1} \\ w_{h_1,o_2} & w_{h_2,o_2} & w_{h_3,o_2} \\ w_{h_1,o_3} & w_{h_2,o_3} & w_{h_3,o_3} \end{bmatrix} = \begin{bmatrix} 0.15 & 0.21 & 0.75 \\ 0.22 & 0.56 & 0.66 \\ 0.97 & 0.32 & 0.53 \end{bmatrix}$$

$$b_1 = 0.45$$
$$b_2 = 0.12$$
$$target = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

And here's the input vector:

$$\begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} = \begin{bmatrix} 0.42 \\ 0.55 \\ 0.12 \end{bmatrix}$$

Now, we do the forward pass from input to hidden layer.

$$\begin{bmatrix} w_{i_1,h_1} & w_{i_2,h_1} & w_{i_3,h_1} \\ w_{i_1,h_2} & w_{i_2,h_2} & w_{i_3,h_2} \\ w_{i_1,h_3} & w_{i_2,h_3} & w_{i_3,h_3} \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_1 \\ b_1 \end{bmatrix} = \begin{bmatrix} net_{h_1} \\ net_{h_2} \\ net_{h_3} \end{bmatrix} = \begin{bmatrix} 0.83 \\ 1.0 \\ 0.99 \end{bmatrix}$$

$$\begin{bmatrix} out_{h_1} \\ out_{h_2} \\ out_{h_3} \end{bmatrix} = \text{sigmoid}(\begin{bmatrix} net_{h_1} \\ net_{h_2} \\ net_{h_3} \end{bmatrix}) = \begin{bmatrix} 0.7 \\ 0.73 \\ 0.73 \end{bmatrix}$$

And we repeat the process for the next layer:

$$\begin{bmatrix} w_{h_1,o_1} & w_{h_2,o_1} & w_{h_3,o_1} \\ w_{h_1,o_2} & w_{h_2,o_2} & w_{h_3,o_2} \\ w_{h_1,o_3} & w_{h_2,o_3} & w_{h_3,o_3} \end{bmatrix} \begin{bmatrix} out_{h_1} \\ out_{h_2} \\ out_{h_3} \end{bmatrix} + \begin{bmatrix} b_2 \\ b_2 \\ b_2 \end{bmatrix} = \begin{bmatrix} net_{o_1} \\ net_{o_2} \\ net_{o_3} \end{bmatrix} = \begin{bmatrix} 0.93 \\ 1.16 \\ 1.42 \end{bmatrix}$$

$$\begin{bmatrix} out_{o_1} \\ out_{o_2} \\ out_{o_3} \end{bmatrix} = \text{sigmoid}(\begin{bmatrix} net_{o_1} \\ net_{o_2} \\ net_{o_3} \end{bmatrix}) = \begin{bmatrix} 0.72 \\ 0.76 \\ 0.81 \end{bmatrix}$$

The Cost Function

We are going to use the squared error function as our cost function. As a consequence, the error in the *i*th output neuron is going to be:

$$E_i = \frac{1}{2}(target_i - out_{o_i})^2$$

The total error will just be the sum of the errors for each of the output neurons.

$$E_{total} = \sum_{k=1}^{n} E_k$$

$$E_k = \frac{1}{2}(target_k - out_{o_k})^2$$

$$\begin{bmatrix} E_1 \\ E_2 \\ E_3 \end{bmatrix} = \begin{bmatrix} 0.26 \\ 0.29 \\ 0.02 \end{bmatrix}$$

$$E_{total} = \sum_{k=1}^{3} E_k = 0.57$$

Backward pass

The goal here is to compute the derivatives of the total error with respect to the weights and biases of the network.

$$\frac{\partial E_{total}}{\partial w_{h_k,o_j}} \tag{B.1}$$

To calculate this, we apply the chain rule.

$$\frac{\partial E_{total}}{\partial w_{h_k,o_j}} = \frac{\partial E_{total}}{\partial out_{o_j}} \frac{\partial out_{o_j}}{\partial net_{o_j}} \frac{\partial net_{o_j}}{\partial w_{h_k,o_j}} \tag{B.2}$$

Output layer

$$\frac{\partial E_{total}}{\partial w_{h_k,o_j}} = \frac{\partial E_{total}}{\partial out_{o_j}} \frac{\partial out_{o_j}}{\partial net_{o_j}} \frac{\partial net_{o_j}}{\partial w_{h_k,o_j}} = -(target_{o_j} - out_{o_j}) out_{o_j}(1 - out_{o_j}) out_{h_k} \tag{B.3}$$

where

$$\frac{\partial E_{total}}{\partial out_{o_j}} = -(target_{o_j} - out_{o_j}), \qquad \frac{\partial out_{o_j}}{\partial net_{o_j}} = out_{o_j}(1 - out_{o_j}), \qquad \frac{\partial net_{o_j}}{\partial w_{h_k,o_j}} = out_{h_k}$$

(B.4)

At this point, we can apply our step of Gradient Descent.
η is our learning rate.
We choose η=0.05

$$\begin{bmatrix} w^*_{h_1,o_1} & w^*_{h_2,o_1} & w^*_{h_3,o_1} \\ w^*_{h_1,o_2} & w^*_{h_2,o_2} & w^*_{h_3,o_2} \\ w^*_{h_1,o_3} & w^*_{h_2,o_3} & w^*_{h_3,o_3} \end{bmatrix} = \begin{bmatrix} w_{h_1,o_1} & w_{h_2,o_1} & w_{h_3,o_1} \\ w_{h_1,o_2} & w_{h_2,o_2} & w_{h_3,o_2} \\ w_{h_1,o_3} & w_{h_2,o_3} & w_{h_3,o_3} \end{bmatrix} - \eta \begin{bmatrix} \frac{\partial E_{total}}{\partial w_{h_1,o_1}} & \frac{\partial E_{total}}{\partial w_{h_2,o_1}} & \frac{\partial E_{total}}{\partial w_{h_3,o_1}} \\ \frac{\partial E_{total}}{\partial w_{h_1,o_2}} & \frac{\partial E_{total}}{\partial w_{h_2,o_2}} & \frac{\partial E_{total}}{\partial w_{h_3,o_2}} \\ \frac{\partial E_{total}}{\partial w_{h_1,o_3}} & \frac{\partial E_{total}}{\partial w_{h_2,o_3}} & \frac{\partial E_{total}}{\partial w_{h_3,o_3}} \end{bmatrix}$$

$$\begin{bmatrix} \frac{\partial E_{total}}{\partial w_{h_1,o_1}} & \frac{\partial E_{total}}{\partial w_{h_2,o_1}} & \frac{\partial E_{total}}{\partial w_{h_3,o_1}} \\ \frac{\partial E_{total}}{\partial w_{h_1,o_2}} & \frac{\partial E_{total}}{\partial w_{h_2,o_2}} & \frac{\partial E_{total}}{\partial w_{h_3,o_2}} \\ \frac{\partial E_{total}}{\partial w_{h_1,o_3}} & \frac{\partial E_{total}}{\partial w_{h_2,o_3}} & \frac{\partial E_{total}}{\partial w_{h_3,o_3}} \end{bmatrix} = \begin{bmatrix} 0.1008 & 0.1051 & 0.1051 \\ 0.0958 & 0.0999 & 0.0999 \\ -0.0199 & -0.0208 & -0.0208 \end{bmatrix}$$

where

$$\frac{\partial E_{total}}{\partial out_{o_j}} = -(target_{o_j} - out_{o_j}), \qquad \begin{bmatrix} \frac{\partial E_{total}}{\partial out_{o_1}} \\ \frac{\partial E_{total}}{\partial out_{o_2}} \\ \frac{\partial E_{total}}{\partial out_{o_3}} \end{bmatrix} = \begin{bmatrix} 0.72 \\ 0.76 \\ -0.19 \end{bmatrix}$$

$$\frac{\partial out_{o_j}}{\partial net_{o_j}} = out_{o_j}(1 - out_{o_j}), \qquad \begin{bmatrix} \frac{\partial out_{o_1}}{\partial net_{o_1}} \\ \frac{\partial out_{o_2}}{\partial net_{o_2}} \\ \frac{\partial out_{o_3}}{\partial net_{o_3}} \end{bmatrix} = \begin{bmatrix} 0.2 \\ 0.18 \\ 0.15 \end{bmatrix}$$

$$\frac{\partial net_{o_j}}{\partial w_{h_k,o_j}} = out_{h_k} \qquad \begin{bmatrix} \frac{\partial net_{o_j}}{\partial w_{h_1,o_j}} \\ \frac{\partial net_{o_j}}{\partial w_{h_2,o_j}} \\ \frac{\partial net_{o_j}}{\partial w_{h_3,o_j}} \end{bmatrix} = \begin{bmatrix} 0.7 \\ 0.73 \\ 0.73 \end{bmatrix}$$

$$\begin{bmatrix} w^*_{h_1,o_1} & w^*_{h_2,o_1} & w^*_{h_3,o_1} \\ w^*_{h_1,o_2} & w^*_{h_2,o_2} & w^*_{h_3,o_2} \\ w^*_{h_1,o_3} & w^*_{h_2,o_3} & w^*_{h_3,o_3} \end{bmatrix} = \begin{bmatrix} w_{h_1,o_1} & w_{h_2,o_1} & w_{h_3,o_1} \\ w_{h_1,o_2} & w_{h_2,o_2} & w_{h_3,o_2} \\ w_{h_1,o_3} & w_{h_2,o_3} & w_{h_3,o_3} \end{bmatrix} -$$

$$\eta \begin{bmatrix} \frac{\partial E_{total}}{\partial w_{h_1,o_1}} & \frac{\partial E_{total}}{\partial w_{h_2,o_1}} & \frac{\partial E_{total}}{\partial w_{h_3,o_1}} \\ \frac{\partial E_{total}}{\partial w_{h_1,o_2}} & \frac{\partial E_{total}}{\partial w_{h_2,o_2}} & \frac{\partial E_{total}}{\partial w_{h_3,o_2}} \\ \frac{\partial E_{total}}{\partial w_{h_1,o_3}} & \frac{\partial E_{total}}{\partial w_{h_2,o_3}} & \frac{\partial E_{total}}{\partial w_{h_3,o_3}} \end{bmatrix} =$$

$$\begin{bmatrix} 0.1449 & 0.2047 & 0.7447 \\ 0.2152 & 0.5550 & 0.6550 \\ 0.9709 & 0.3210 & 0.5310 \end{bmatrix}$$

We also need to update the bias.

$$b^*_2 = b_2 - \eta \frac{\partial E_{total}}{\partial b_2} = 0.1073$$

Hidden layer

$$\frac{\partial E_{total}}{\partial w_{i_k,h_j}} = \frac{\partial E_{total}}{\partial out_{h_j}} \frac{\partial out_{h_j}}{\partial net_{h_j}} \frac{\partial net_{h_j}}{\partial w_{i_k,h_j}} = \frac{\partial E_{total}}{\partial out_{h_j}} out_{h_j}(1 - out_{h_j}) i_k$$

(B.5)

where

$$\frac{\partial E_{total}}{\partial out_{h_j}} = \sum_{k=1}^{3} \frac{\partial E_{o_k}}{\partial out_{h_j}} = \sum_{k=1}^{3} \frac{\partial E_{o_k}}{\partial out_{o_k}} \frac{\partial out_{o_k}}{\partial net_{o_k}} \frac{\partial net_{o_k}}{\partial out_{h_j}} = -\sum_{k=1}^{3} (target_{o_k} - out_{o_k}) out_{o_k}(1 - out_{o_k}) w_{h_j,o_k}$$

(B.6)

$$\frac{\partial out_{h_j}}{\partial net_{h_j}} = out_{h_j}(1 - out_{h_j}), \qquad \frac{\partial net_{h_j}}{\partial w_{i_k,h_j}} = i_k$$

$$\begin{bmatrix} w^*_{i_1,h_1} & w^*_{i_2,h_1} & w^*_{i_3,h_1} \\ w^*_{i_1,h_2} & w^*_{i_2,h_2} & w^*_{i_3,h_2} \\ w^*_{i_1,h_3} & w^*_{i_2,h_3} & w^*_{i_3,h_3} \end{bmatrix} = \begin{bmatrix} w_{i_1,h_1} & w_{i_2,h_1} & w_{i_3,h_1} \\ w_{i_1,h_2} & w_{i_2,h_2} & w_{i_3,h_2} \\ w_{i_1,h_3} & w_{i_2,h_3} & w_{i_3,h_3} \end{bmatrix} - \eta \begin{bmatrix} \frac{\partial E_{total}}{\partial w_{i_1,h_1}} & \frac{\partial E_{total}}{\partial w_{i_2,h_1}} & \frac{\partial E_{total}}{\partial w_{i_3,h_1}} \\ \frac{\partial E_{total}}{\partial w_{i_1,h_2}} & \frac{\partial E_{total}}{\partial w_{i_2,h_2}} & \frac{\partial E_{total}}{\partial w_{i_3,h_2}} \\ \frac{\partial E_{total}}{\partial w_{i_1,h_3}} & \frac{\partial E_{total}}{\partial w_{i_2,h_3}} & \frac{\partial E_{total}}{\partial w_{i_3,h_3}} \end{bmatrix}$$

$$\begin{bmatrix} \frac{\partial E_{total}}{\partial w_{i_1,h_1}} & \frac{\partial E_{total}}{\partial w_{i_2,h_1}} & \frac{\partial E_{total}}{\partial w_{i_3,h_1}} \\ \frac{\partial E_{total}}{\partial w_{i_1,h_2}} & \frac{\partial E_{total}}{\partial w_{i_2,h_2}} & \frac{\partial E_{total}}{\partial w_{i_3,h_2}} \\ \frac{\partial E_{total}}{\partial w_{i_1,h_3}} & \frac{\partial E_{total}}{\partial w_{i_2,h_3}} & \frac{\partial E_{total}}{\partial w_{i_3,h_3}} \end{bmatrix} = \begin{bmatrix} 0.0026 & 0.003 & 0.0008 \\ 0.0076 & 0.0099 & 0.0022 \\ 0.0143 & 0.0187 & 0.0041 \end{bmatrix}$$

where

$$\frac{\partial E_{total}}{\partial out_{h_j}} = \sum_{k=1}^{3} \frac{\partial E_{o_k}}{\partial out_{h_j}} = \sum_{k=1}^{3} \frac{\partial E_{o_k}}{\partial out_{o_k}} \frac{\partial out_{o_k}}{\partial net_{o_k}} \frac{\partial net_{o_k}}{\partial out_{h_j}} = -\sum_{k=1}^{3} (target_{o_k} - out_{o_k}) out_{o_k}(1 - out_{o_k}) w_{h_j,o_k}$$

$$\begin{bmatrix} \frac{\partial E_{total}}{\partial out_{h_1}} \\ \frac{\partial E_{total}}{\partial out_{h_2}} \\ \frac{\partial E_{total}}{\partial out_{h_3}} \end{bmatrix} = \begin{bmatrix} 0.03 \\ 0.09 \\ 0.17 \end{bmatrix}$$

$$\frac{\partial out_{h_j}}{\partial net_{h_j}} = out_{h_j}(1 - out_{h_j}) \qquad \begin{bmatrix} \frac{\partial out_{h_1}}{\partial net_{h_1}} \\ \frac{\partial out_{h_2}}{\partial net_{h_2}} \\ \frac{\partial out_{h_3}}{\partial net_{h_3}} \end{bmatrix} = \begin{bmatrix} 0.21 \\ 0.2 \\ 0.2 \end{bmatrix}$$

$$\frac{\partial net_{h_j}}{\partial w_{i_k,h_j}} = i_k \qquad \begin{bmatrix} \frac{\partial net_{h_j}}{\partial w_{i_1,h_j}} \\ \frac{\partial net_{h_j}}{\partial w_{i_2,h_j}} \\ \frac{\partial net_{h_j}}{\partial w_{i_3,h_j}} \end{bmatrix} = \begin{bmatrix} 0.42 \\ 0.55 \\ 0.12 \end{bmatrix}$$

$$\begin{bmatrix} w^*_{i_1,h_1} & w^*_{i_2,h_1} & w^*_{i_3,h_1} \\ w^*_{i_1,h_2} & w^*_{i_2,h_2} & w^*_{i_3,h_2} \\ w^*_{i_1,h_3} & w^*_{i_2,h_3} & w^*_{i_3,h_3} \end{bmatrix} = \begin{bmatrix} w_{i_1,h_1} & w_{i_2,h_1} & w_{i_3,h_1} \\ w_{i_1,h_2} & w_{i_2,h_2} & w_{i_3,h_2} \\ w_{i_1,h_3} & w_{i_2,h_3} & w_{i_3,h_3} \end{bmatrix} \text{-}\eta \begin{bmatrix} \frac{\partial E_{total}}{\partial w_{i_1,h_1}} & \frac{\partial E_{total}}{\partial w_{i_2,h_1}} & \frac{\partial E_{total}}{\partial w_{i_3,h_1}} \\ \frac{\partial E_{total}}{\partial w_{i_1,h_2}} & \frac{\partial E_{total}}{\partial w_{i_2,h_2}} & \frac{\partial E_{total}}{\partial w_{i_3,h_2}} \\ \frac{\partial E_{total}}{\partial w_{i_1,h_3}} & \frac{\partial E_{total}}{\partial w_{i_2,h_3}} & \frac{\partial E_{total}}{\partial w_{i_3,h_3}} \end{bmatrix} =$$

$$\begin{bmatrix} 0.4499 & 0.1698 & 0.7899 \\ 0.1396 & 0.8195 & 0.3298 \\ 0.3693 & 0.6191 & 0.3398 \end{bmatrix}$$

Finally, we update the bias.

$$b^*_1 = b_1 - \eta \frac{\partial E_{total}}{\partial b_1} = 0.4469.$$