O'REILLY®

Generative Deep Learning

Teaching Machines to Paint, Write, Compose and Play

Early Release

RAW & UNEDITED

David Foster

Generative Deep Learning

Teaching Machines to Paint, Write,
Compose and Play

David Foster

Beijing • Boston • Farnham • Sebastopol • Tokyo

Generative Deep Learning

by David Foster

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editors:Michele Cronin and Mike Loukides

Production Editor: Katherine Tozer

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

August 2019: First Edition

Revision History for the Early Release

- 2019-02-22: First Release
- 2019-04-04: Second Release
- 2019-04-25: Third Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492041948 for release details.

978-1-492-04194-8

Chapter 1. Generative Modeling

> **NOTE**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as he or she writes—so you can take advantage of these technologies long before the official release of these titles.

This chapter is a general introduction to the field of generative modeling. We shall first look at what it means to say that a model is *generative* and learn how it differs from the more widely studied *discriminative* modeling. Then, we will introduce the framework and core mathematical ideas that will allow us to structure our general approach to problems that require a generative solution.

With this in place, we will then build our first example of a generative model (Naive Bayes), that is probabilistic in nature. We shall see that this allows us to generate novel examples that are outside of our training dataset, but shall also explore the reasons why this type of model may fail as the size and complexity of the space of possible creations increases.

What is Generative Modeling?

A generative model can be broadly be defined as follows:

> *A generative model describes how a dataset is generated, in terms of a probabilistic model. By sampling from this model, we are able to generate new data.*

Suppose we have a dataset containing images of horses. We may wish to build a model that can generate a new image of a horse that has never existed but still looks real because the model has learnt the general rules that govern the appearance of a horse. This is the kind of problem that can be solved using generative modeling. A summary of a typical generative modeling process is shown in Figure 1-1.

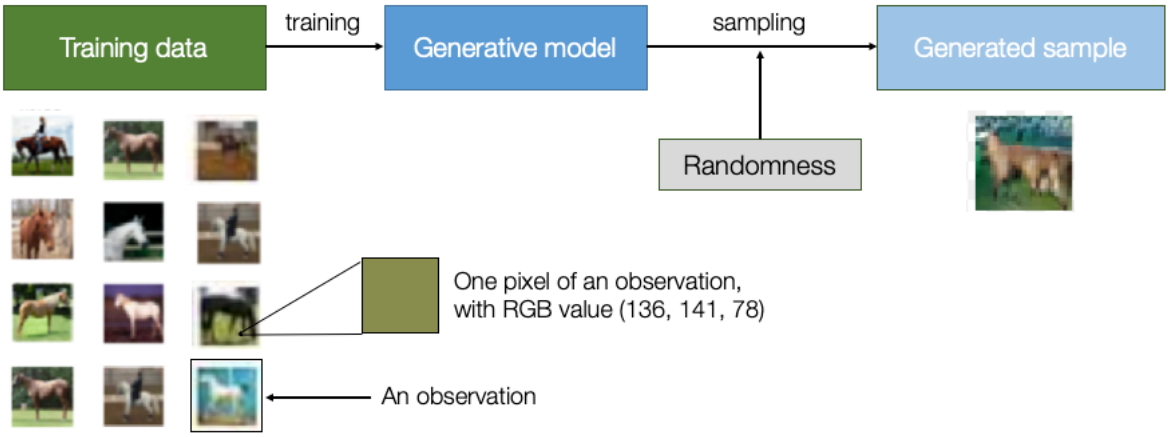


Figure 1-1. The generative modeling process

Firstly, we require a dataset consisting of many examples of the entity we are trying to generate. This is known as the *training data* and one such data point is called an *observation*.

Each observation consists of many *features*—for an image generation problem, the features are usually the individual pixel values. It is our goal to build a model that can generate new sets of features that look as if they have been created using the same rules as the original data. Conceptually, for image generation this is an incredibly difficult task, considering the vast number of ways that individual pixels values can be assigned and the relatively tiny number of such arrangements that consistute an image of the entity we are trying to simulate.

A generative model must also be *probabilistic* rather than *deterministic*. If our model is merely a fixed calculation, such as taking the average value of each pixel in the dataset, it is not generative because the model produces the same output every time. The model must include a *stochastic* (random) element that influences the individual samples generated by the model.

In other words, we can imagine that there is some unknown probabilistic distribution that explains why some images are likely to be found in the training dataset and other images are not. It is our job to build a generative model that mimics this distribution as closely as possible and then sample from it to generate new, distinct observations that look as if they could have been included in the original training set.

Generative vs discriminative modeling

In order to truly understand what generative modeling aims to achieve and why this is important, it is useful to compare it to its counterpart, *discriminative modeling*. If you have studied machine learning, most problems you will have faced will have most likely been discriminative in nature. To understand the difference, let's look at an example.

Suppose we have a dataset of paintings—some painted by Van Gogh and some by other artists. With enough data, we could train a *discriminative model* to predict if a given painting was painted by Van Gogh. Our model would learn that certain colours, shapes and textures are more likely to indiciate that a painting is by the Dutch master and for paintings with these features, the model would upweight its prediction accordingly. Figure 1-2 shows the discriminative modeling process—note how it differs from the generative modeling process shown in Figure 1-1.

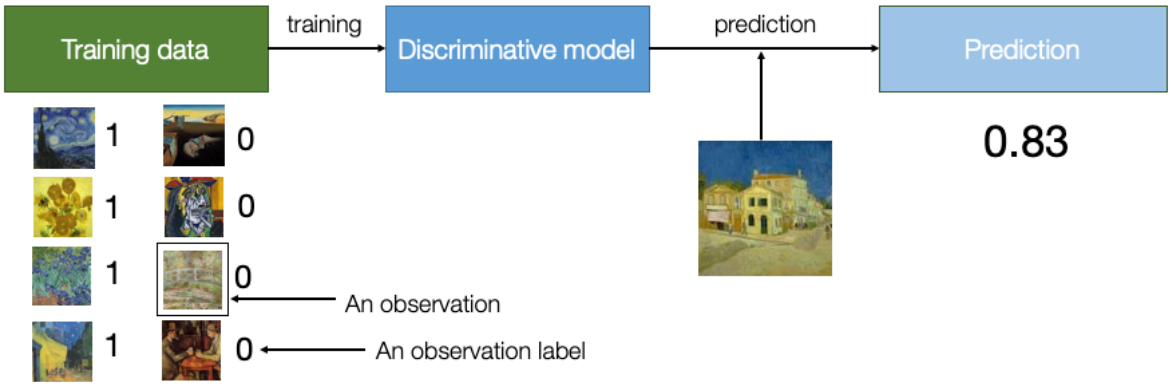


Figure 1-2. The discriminative modeling process

One key difference is that when performing discriminative modeling, each observation in the training data has a *label*—for a binary classification problem such as our artist discriminator, Van Gogh paintings would be labelled *1* and non-Van Gogh painting labelled *0*. Our model then learns how to discriminate between these two groups and outputs the probability that a new observation has label *1*—i.e. that it was painted by Van Gogh.

For this reason, discriminative modeling is synonymous with *supervised learning*—learning a function that maps an input to an output using a labelled dataset. Generative modeling is usually performed with an unlabelled dataset (i.e. as a form of unsupervised learning), though can also be applied to a labelled dataset to learn how to generate observations from each distinct class.

We'll now introduce some mathematical notation to describe the difference between generative and discriminative modeling.

Discriminative modeling estimates $p(y|\mathbf{x})$—the probability of a label y given observation $\mathbf{x}$.

Generative modeling estimates $p(\mathbf{x})$—the probability of observing observation $\mathbf{x}$.

If the dataset is labelled, we can also build a generative model that estimates the joint distribution $p(\mathbf{x}, y)$.

In other words, discriminative modeling attempts to estimate the probability that an observation $\mathbf{x}$ belongs to category y. Generative modeling doesn't care about labelling observations. Instead, it attempts to estimate the probability of seeing the observation at all.

The key point is that even if we were able to build a perfect discriminative model to identify Van Gogh paintings, it would still have no idea how to create a painting that looks like a Van Gogh. It can only output probabilities against existing images as this is what it has been trained to do. Instead, we would need to train a generative model, which can output sets of pixels that have a high chance of belonging to the original training dataset.

Advances in machine learning

To understand why generative modeling can be considered the next frontier for machine learning, we must first look at why discriminative modeling has been the driving force behind most progress in machine learning methodology in the last two decades, both in academia and industry.

From an academic perspective, progress in discrimintive modeling is certainly easier to monitor as we can measure performance metrics against certain high-profile classification tasks, to determine the current best in class methodology. Generative models are often more difficult to evaluate, especially when the quality of the output is largely subjective. Therefore, much emphasis in recent years has been placed on training discriminative models to reach human or super-human performance in a variety of image or text classification tasks.

For example, for image classification, the key breakthrough came in 2012 when a team, led by Geoff Hinton at the University of Toronto, won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) with a deep convolutional neural network. The competition involves classifying images into one of 1000 categories and is used as a benchmark to compare the latest state of the art techniques. The deep learning model had an error rate of 16%—a massive improvement on the next best model which only achieved a 26.2% error rate. This sparked a deep learning boom that has resulted in the error rate falling even further year after year. The 2015 winner achieved super-human performance for the first time, with an error rate of 4% and the current state-of-the-art model achieves an error rate of just 2%. Many would now consider the challenge a solved problem.

As well as being easier to publish measurable results within an academic setting, discriminative modeling has historically been more readily applicable to business problems than generative modeling. Generally, in a business setting, we generally do not care *how* data was generated, but instead want to know how a new example should be categorised or valued. For example:

- Given a satellite image, a government defence official would only care about the probability that it contains enemy units, not the probability that this particular image should appear.

- A customer relations manager would only be interested in knowing if the sentiment of an incoming email is positive or negative and wouldn't find much use in a generative model that could output examples of customer emails that don't yet exist.

- A doctor would want to know the chance that a given retinal image indicates glaucoma, rather than have access to a model that can generates novel pictures of the back an eye.

As most solutions required by businesses are in the domain of discriminative modeling, there has been a rise in the number of *machine-learning-as-a-service* (MLaaS) tools that aim to commodotise the use of discriminative modeling within industry, by largely automating the build, validation and monitoring processes that are common to almost all discrimiative modeling tasks.

The rise of generative modeling

Whilst discriminative modeling has so far provided the bulk of the impetus behind advances in machine learning, in last 3-5 years many of the most interesting advancements in the field have come through novel application of deep learning to generative modeling tasks.

In particular, there has been increased media attention on generative modeling projects such as StyleGAN (2018, Karras et al.)[1] from NVIDIA, that is able to create hyper-realistic images of human faces and the GPT-2 language model from OpenAI (2019, Radford, Wu et al.)[2] , that is able to complete a passage of text, given a short introductory paragraph.

Figure 1-3. Face generation using generative modeling has improved significantly in the last four years[3]

Figure 1-3 shows the striking progress that has already been made in facial image generation since 2014. There are clear positive applications here for industries such as game design and cinematography and improvements in automatic music generation will also surely start to resonate within these domains. It remains to be seen whether we will one day read news articles or novels written by a generative model, but the recent progress in this area is staggering and it is certainly not outrageous to suggest that this one day may be the case. Whilst exciting, this also raises ethical questions around the proliferation of fake content on the internet and mean it may become ever harder to trust what we see and read through public channels of communication.

As well as the practical uses of generative modeling (many of which are yet to be discovered) there are three deeper reasons why generative modeling can be considered the key to unlocking a far more form sophisticated form of artifical intelligence, that goes beyond what discriminative modeling alone can achieve.

Firstly, purely from a theoretical point of view, we should not be content with only being able to excel at categorising data but should also seek a more complete understanding of how the data was generated in the first place. This is undoubtedly a more difficult problem to solve, due to the high dimensionality of the space of feasible outputs and the relatively small number of creations that we would class as belonging to the dataset. However, as we shall see, many of the same techniques that have driven development in discriminative modeling, such as deep learning, can be utilised by generative models too.

Secondly, it is highly likely that generative modeling will be central to driving future developments in other fields of machine learning, such as reinforcement learning. Reinforcement learning is the study of teaching agents to optimise a goal in an environment through trial and error. For example, we could use reinforcement learning to train a robot to walk across a given terrain. The general approach would be to build a computer simulation of the terrain and then run many experiments where the agent tries out different strategies. Over time the agent would learn which strategies are more successful than others and therefore gradually improve. A typical problem with this approach is that the physics of the environment is often highly complex and would need be calculated at each time step in order to feed the information back to the agent to decide its next move. However, if the agent were able to simulate its environment through a generative model, it wouldn't need to test out the strategy in the computer simulation or in the real world, but instead could learn in its own *imaginary* environment. In [Link to Come] we shall see this idea in action by training a car to drive as fast as possible around a track by allowing it to learn directly from its own hallucinated environment.

Lastly, if we are to truly say that we have built a machine that has acquired a form of

intelligence that is comparable to a human, generative modeling must surely be part of the solution. One of the finest examples of a generative model in the natural world is the person reading this book. Take a moment to consider what an incredible generative model you are. You can close your eyes and imagine what an elephant would look like from any possible angle. You can imagine a number of plausible different endings to your favourite TV show and you can plan your week ahead by working through various futures in your minds eye and taking action accordingly. Current neuroscientific theory suggests that our perception of reality is not a highly complex discriminative model operating on our sensory input to produce predictions of what we are experiencing, but is instead a generative model that is trained from birth to produce simulations of our surroundings that accurately match the future. Some theories even suggest that the output from this generative model is what we directly perceive as reality. Clearly, a deep understanding of how we can build machines to acquire this ability will be central to our continued understanding of the workings of the brain and general artifical intelligence.

With this in mind, let's begin our journey into the exciting world of generative modeling. To begin with we shall look at the simplest examples of generative models and introduce some of the ideas that will help us to work through the more complex architectures that we will encounter later in the book.

The Generative Modeling Framework

Let's start by playing a generative modeling game in just 2 dimensions. I've chosen a rule that has been used to generate the set of points $\mathbf{X}$ in Figure 1-4. Let's call this rule p_{data}. Your challenge is to choose a different point $\mathbf{x} = (x_1, x_2)$ in the space that looks like it has been generated by the same rule.

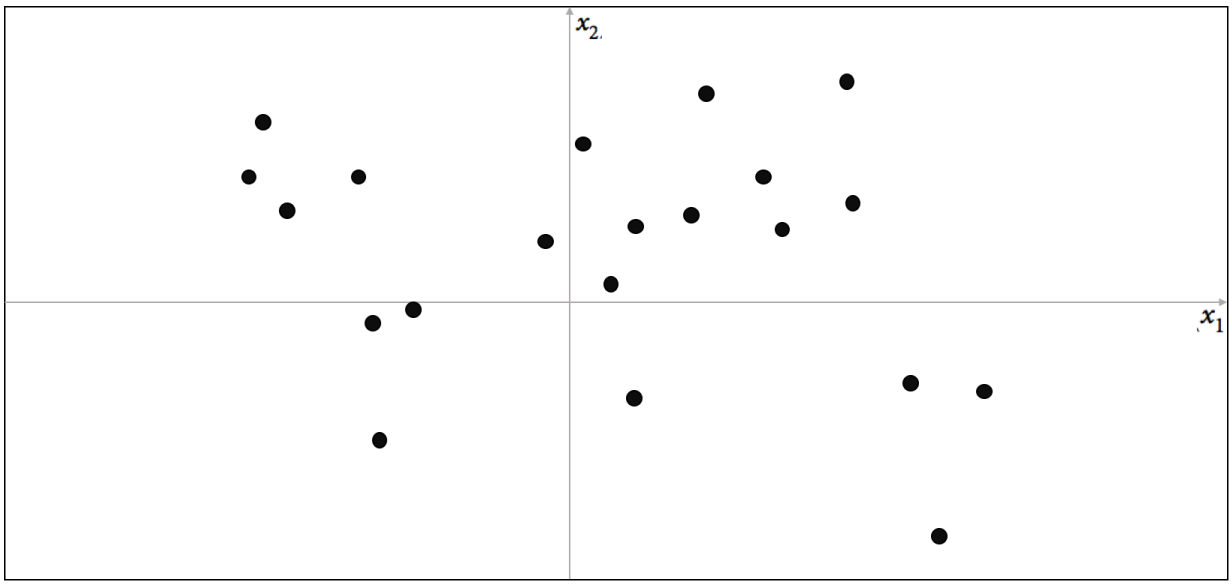


Figure 1-4. A set of points in 2-dimensions, generated by an unknown rule p_{data}

Where would you choose? You probably used your knowledge of the existing data points to construct a mental model, p_{model}, of whereabouts in the space the point is more likely to be found. In this respect, p_{model} is an *estimate* of p_{data}. Perhaps you decided that p_{model} should look like Figure 1-5—a rectangular box where points may be found and an area outside of the box where there is no chance of finding any points. To generate a new observation, you can simply choose a point at random within the box, or more formally, *sample* from the distribution p_{model}. Congratulations—you have just devised your first generative model!

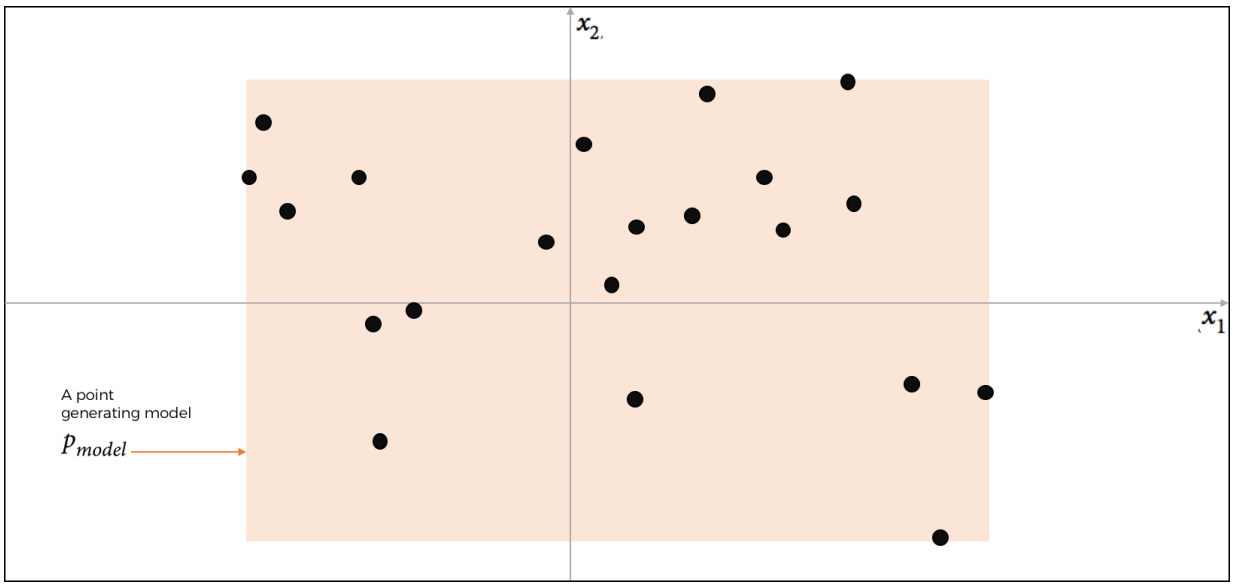


Figure 1-5. The orange box, p_{model} is an estimate of the true data generating distribution, p_{data}

Whilst this isn't the most most complex example, we can use it to understand what generative modeling is trying to achieve. The following framework sets out our motivations:

THE GENERATIVE MODELING FRAMEWORK

- We have a dataset of observations $\mathbf{X}$.
- We assume that the observations have been generated according to some unknown distribution, p_{data}.
- A generative model p_{model} tries to mimic p_{data}. If we achieve this goal, we can sample from p_{model} to generate observations that appear to have been drawn from p_{data}
- We are impressed by p_{model} if:
    - RULE 1: It can generate examples that appear to have been drawn from p_{data}
    - RULE 2: It can generate examples that are suitably different from the observations in $\mathbf{X}$. In other words, the model shouldn't simply reproduce things it has already seen.

Let's now reveal the true data generating distribution, p_{data} and see how the framework applies to this example.

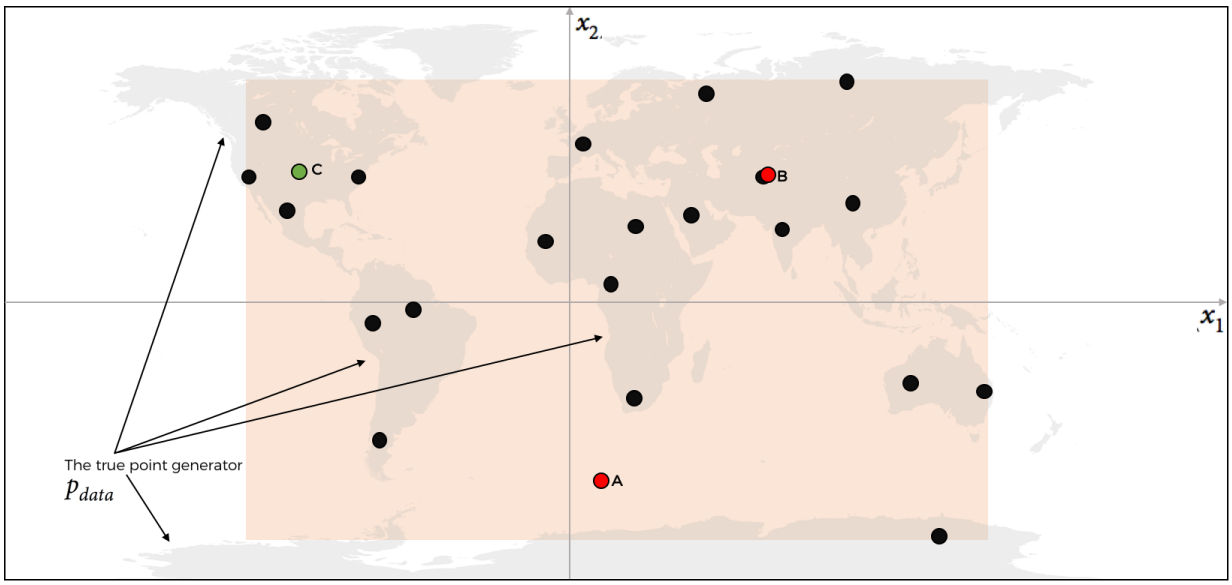


Figure 1-6. The orange box p_{model} is an estimate of the true data generating distribution p_{data} (the grey area).

As we can see from Figure 1-6, the data generating rule is simply a uniform distribution over the land mass of the world, with no chance of finding a point in the sea.

Clearly, our model p_{model} is an oversimplification of p_{data}. Points A, B and C show three observations generated by p_{model} with varying degress of success:

- **Point A** breaks Rule 1 of the Generative Modeling Framework—it does not appear to have been generated by p_{data} as it's in the middle of the sea
- **Point B** is so close to a point in the dataset that we shouldn't be impressed that our model can generate such a point. If all the examples generated by the model were like this, it would break Rule 2 of the Generative Modeling Framework
- **Point C** can be deemed a success because it could have been generated by p_{data} and is suitably different from any point in the original dataset.

The field of generative modeling is diverse and the problem definition can take a great variety of forms. However in most scenarios the Generative Modeling Framework captures how we should broadly think about tackling the problem.

Let's now build at our first non-trivial example of a generative model.

Probabalistic generative models

Firstly, if you have never studied probability—don't worry. To build and run many of the deep learning techniques that we shall see later in this book, it is not essential to have a deep understanding of statistical theory. However, to gain a full appreciation of the history of the task that we are trying to tackle, it's worth trying to build a generative model that doesn't rely on deep learning and instead is grounded purely in probabilistic theory. This way, you will have the foundations in place to understand all generative models, whether based on deep learning or not, from the same probabalistic standpoint.

If you already have a good understanding of probability, that's great and much of the next section may already be familiar to you. However, there is a fun example in the middle of this chapter, so be sure not to miss out on that…

As a first step, we shall define four key terms—*sample space*, *density function*, *parametric modeling* and *maximum likelihood estimation*.

SAMPLE SPACE

The *sample space* is the complete set of all values an observation $\mathbf{x}$ can take.

In our previous example, the sample space consists of all points of latitude and longitude $\mathbf{x} = (x_1, x_2)$ on the world map.

For example, $\mathbf{x} = (40.7306, -73.9352)$ is a point in the sample space (New York City)

PROBABILITY DENSITY FUNCTION

A *probability density function* (or simply *density function*) $p(\mathbf{x})$, is a function that maps a point $\mathbf{x}$ in the sample space to a number between 0 and 1. The sum[4] of the density function over all points in the sample space must equal 1, so that it is a well-defined probability distribution [5] .

In the world map example, the density function of our model is 0 outside of the orange box and constant inside of the box.

Whilst there is only one true density function $p_{data}(\mathbf{x})$ that is assumed to have generated the observable dataset, there are infintely many density functions $p_{model}(\mathbf{x})$ that we can use to estimate $p_{data}(\mathbf{x})$.

In order to structure our approach to finding a suitable $p_{model}(\mathbf{x})$ we can use a technique known as *parametric modeling*

PARAMETRIC MODELING

A *parametric model*, $p_\theta(\mathbf{x})$, is a family of density functions that can be described using a finite number of parameters, θ.

For example, the family of all possible boxes you could draw on Figure 1-5 is an example of a parametric model. In this case, there are four parameters—the coordinates of the bottom left (θ_1, θ_2) and top right (θ_3, θ_4) corners of the box.

Thus each density function $p_\theta(\mathbf{x})$ in this parametric model (i.e. each box) can be uniquely represented by four numbers, $\theta = (\theta_1, \theta_2, \theta_3, \theta_4)$.

LIKELIHOOD

The *likelihood*, $\mathscr{L}(\theta \mid \mathbf{x})$ of a parameter set θ is a function that measures the plausibility of θ, given some observed point, $\mathbf{x}$.

It is defined as follows:

$$\mathscr{L}(\theta \mid \mathbf{x}) = p_\theta(\mathbf{x})$$

That is, the likelihood of θ given some observed point $\mathbf{x}$ is defined to be the value of the density function parameterised by θ, at the point $\mathbf{x}$.

If we have a whole dataset $\mathbf{X}$ of independent observations then we can write:

$$\mathscr{L}(\theta \mid \mathbf{X}) = \prod_{\mathbf{x} \in \mathbf{X}} p_\theta(\mathbf{x})$$

Since this product can be quite computationlally difficult to work with, we often use the *log-likelihood* $\square$ instead:

$$\square(\theta \mid \mathbf{X}) = \sum_{\mathbf{x} \in \mathbf{X}} \log p_\theta(\mathbf{x})$$

There are statistical reasons why the likelihood is defined in this way, but it enough for us to understand why intuitively, this makes sense. We are simply defining the likelihood of a set of parameters θ to be equal to the probability of seeing the data under the model parameterised by θ.

In the world map example, an orange box that only covered the left half of the map would have a likelihood of 0—it couldn't have possibly generated the dataset as we have observed points in the right half of the map. The orange box in Figure 1-5 has a positive likelihood as the density function is positive for all data points under this model.

It therefore makes intuitive sense that the focus of parametric modeling should be to find the optimal value $\hat{\theta}$ of the parameter set that maximises the likelihood of observing the dataset $\mathbf{X}$. This technique is quite appropriatelly called *maximum likelihood estimation*.

MAXIMUM LIKELIHOOD ESTIMATION

Maximum Likelihood Estimation is the technique that allows us to estimate $\hat{\theta}$—the set of parameters θ of a density function, $p_\theta(\mathbf{x})$, that are most likely to explain some observed data $\mathbf{X}$.

More formally,

$$\hat{\theta} = \arg \max_{\theta} \mathscr{L}(\theta|X)$$

$\hat{\theta}$ is also called the *maximum likelihood estimate* (MLE).

We now have all the necessary terminology to start describing how we can build a probabalistic generative model.

Most chapters in this book will contain a short story that helps to describe a particular technique. In this chapter, we shall start by paying a trip to planet Wrodl where our first generative modeling assignment awaits…

Hello Wrodl!

The year is 2047 and you are delighted to have been appointed as the new Chief Fashion Officer (CFO) of planet Wrodl. As CFO, it is your sole responsibiltiy to create new and exciting fashion trends for the inhabitants of the planet to follow.

The Wrodlers are known to be quite particular when it comes to fashion, so your task is to generate new styles that are similar to those that already exist on the planet, but not identical.

On arrival, you are presented with a dataset featuring 50 observations of Wrodler fashion (Figure 1-7) and told that you have a day to come up with 10 new styles to present back to the Fashion Police for inspection. You're allowed to play around with hairstyle, hair colour, glasses, clothing type and clothing colour to create your masterpieces.

Figure 1-7. Headshots of 50 Wrodlers[6]

As you're a data scientist at heart, you decide to deploy a generative model to solve the problem. After a brief visit to the Intergalactic Library, you pick up a book called Generative Deep Learning and begin to read…

To be continued…

Your first probabilistic generative model

Let's take a closer look at the Wrodl dataset. It consists of $N = 50$ observations of fashions currently seen on the planet. Each observation can be described by five features, (topType, hairColor, accessoriesType, clotheType, clotheColour), as shown in Figure 1-8.

face_id	accessoriesType	clotheColor	clotheType	hairColor	topType
0	Round	White	ShirtScoopNeck	Red	ShortHairShortFlat
1	Round	White	Overall	SilverGray	ShortHairFrizzle
2	Sunglasses	White	ShirtScoopNeck	Blonde	ShortHairShortFlat
3	Round	White	ShirtScoopNeck	Red	LongHairStraight
4	Round	White	Overall	SilverGray	NoHair
5	Blank	White	Overall	Black	LongHairStraight
6	Sunglasses	White	Overall	SilverGray	LongHairStraight
7	Round	White	ShirtScoopNeck	SilverGray	ShortHairShortFlat
8	Round	Pink	Hoodie	SilverGray	LongHairStraight
9	Round	PastelOrange	ShirtScoopNeck	Blonde	LongHairStraight

Figure 1-8. The first 10 observations in the Wrodler face dataset

The possible values that each feature can take are as follows:

- 7 different hairstyles (topType)

    - NoHair, LongHairBun, LongHairCurly, LongHairStraight, ShortHairShortWaved, ShortHairShortFlat, ShortHairFrizzle

- 6 different kinds of hair colour (hairColor)
    - Black, Blonde, Brown, PastelPink, Red, SilverGrey
- 3 different kinds of glasses type (accessoriesType)
    - Blank, Round, Sunglasses
- 4 different kinds of clothing (clotheType)
    - Hoodie, Overall, ShirtScoopNeck, ShirtVNeck
- 8 different kinds of clothing colour (clotheColor)
    - Black, Blue01, Grey01, PastelGreen, PastelOrange, Pink, Red, White

There are 7 x 6 x 3 x 4 x 8 = 4032 different combinations of these features, so there are 4032 points in the **sample space**.

We can imagine that our dataset has been generated by some distribution p_{data} that favours some feature values over others. For example, we can see from the images in Figure 1-7 that white clothing seems to be a popular choice as well as silver hair and scoop neck T-shirts.

The problem is that we do not know p_{data} explicitly—all we have to work with is the sample of observations $\mathbf{X}$ that were generated by p_{data}. The goal of generative modeling is to use these observations to build a p_{model} that can accurately mimic the observations produced by p_{data}.

To achieve this, we could simply assign a probability to each possible combination of features, based on the data we have seen. Therefore, this parametric model would have $d = 4031$ parameters—one for each point in the sample space of possibilities, minus one since the value of the last parameter would be forced so that the total sums to 1. Thus the parameters of the model that we are trying to estimate are $(\theta_1, \ldots, \theta_{4031})$.

This particular class of parametric model is known as a *multinomial distribution*, and the **maximum likelihood estimate** $\widehat{\theta_j}$ of each parameter is given by:

$$\widehat{\theta_j} = \frac{n_j}{N}$$

where n_j is the number of times that combination j was observed in the dataset and $N = 50$ is

the total number of observations.

In other words, the estimate for each parameter is just the proportion of times that its correspoinding combination was observed in the dataset.

For example, the following combination (let's call it combination 1) appears twice in the dataset:

$$(LongHairStraight, Red, Round, ShirtScoopNeck, White)$$

Therefore:

$$\widehat{\theta_1} = 2/50 = 0.04$$

As another example, the following combination (let's call it combination 2) doesn't appear at all in the dataset:

$$(LongHairStraight, Red, Round, ShirtScoopNeck, Blue01)$$

Therefore:

$$\widehat{\theta_2} = 0/50 = 0$$

We can calculate all of the $\widehat{\theta_j}$ values in this way, to define a distribution over our sample space. Since we can sample from this distribution, our list could potentially be called a generative model. However, it fails in one major respect. It can never generate anything that it hasn't already seen, since $\widehat{\theta_j} = 0$ for any combination that wasn't in the original dataset $\mathbf{X}$.

To address this, we could assign an additional *pseudocount* of 1 to each possible combination of features. This is known as *additive smoothing*. Under this model, our MLE for the parameters would be:

$$\widehat{\theta_j} = \frac{n_j + 1}{N + d}$$

Now, every single combination has a non-zero probability of being sampled, including those that were not in the original dataset. However, this still fails to be a satisfactory generative model, because the probability of observing a point not in the original dataset is just a constant. If we tried to use such a model to generate Picasso paintings, it would assign just as much weight to a random collection of colourful pixels as to a replica of a Picasso painting that differs only very slightly from a genuine painting.

We would ideally like our generative model to upweight areas of the sample space that it believes are more likely, due to some inherent structure learnt from the data, rather that just

placing all probabilistic weight on the points that are present in the dataset.

To achieve this, we need to choose a different parametric model.

Naive Bayes

The Naive Bayes parametric model makes use of a simple assumption, that drastically reduces the number of parameters we need to estimate.

We make the *naive* assumption that each feature x_j is *independent* of every other feature x_k.[7] . Relating this to the Wrodl dataset, we are assuming that the choice of hair colour has no impact on the choice of clothing type, and the glasses that someone wears has no impact on their hairstyle, for example. More formally, for all features x_j, x_k:

Equation 1-1. The Naive Bayes assumption

$$p(x_j \mid x_k) = p(x_j)$$

To apply this assumption, we first make use of the chain rule of probability to write the density function as a product of conditional probabilities.

Equation 1-2. The Naive Bayes model

$$\begin{aligned} p(\mathbf{x}) &= p(x_1, \ldots, x_K) \\ &= p(x_2, \ldots, x_K \mid x_1)p(x_1) \\ &= p(x_3, \ldots, x_K \mid x_1, x_2)p(x_2 \mid x_1)p(x_1) \\ &= \prod_{k=1}^{K} p(x_k \mid x_1, \ldots, x_{k-1}) \end{aligned}$$

where K is the total number of features (i.e. 5 for the Wrodl example).

We now apply the Naive Bayes assumption to simplify the last line:

Equation 1-3. The Naive Bayes model

$$p(\mathbf{x}) = \prod_{k=1}^{K} p(x_k)$$

Therefore, the problem is reduced to estimating the parameters $\theta_{kl} = p(x_k = l)$ for each feature separately and multiplying these to find the probability for any possible combination.

How many parameters do we now need to estimate? For each feature, we need to estimate a parameter for each value that the feature can take. Therefore, in the Wrodl example, this model is defined by only 7+6+3+4+8-5=23 parameters [8]

The maximum likelihood estimates $\widehat{\theta_{kl}}$ are as follows:

Equation 1-4. The maximum likelihood estimate for the Naive Bayes model

$$\widehat{\theta_{kl}} = \frac{n_{kl}}{N}$$

where $\widehat{\theta_{kl}}$ is the number of times that feature k takes on the value l in the dataset and $N = 50$ is the total number of observations.

Let's see what these parameters are for the Wrodl dataset.

topType	n	$\hat{\theta}$
NoHair	7	0.14
LongHairBun	0	0.00
LongHairCurly	1	0.02
LongHairStraight	23	0.46
ShortHairShortWaved	1	0.02
ShortHairShortFlat	11	0.22
ShortHairFrizzle	7	0.14
Grand Total	50	1.00

hairColor	n	$\hat{\theta}$
Black	7	0.14
Blonde	6	0.12
Brown	2	0.04
PastelPink	3	0.06
Red	8	0.16
SilverGrey	24	0.48
Grand Total	50	1.00

clotheColor	n	$\hat{\theta}$
Black	0	0.00
Blue01	4	0.08
Grey01	10	0.20
PastelGreen	5	0.10
PastelOrange	2	0.04
Pink	4	0.08
Red	3	0.06
White	22	0.44
Grand Total	50	1.00

accessoriesType	n	$\hat{\theta}$
Blank	11	0.22
Round	22	0.44
Sunglasses	17	0.34
Grand Total	50	1.00

clotheType	n	$\hat{\theta}$
Hoodie	7	0.14
Overall	18	0.36
ShirtScoopNeck	19	0.38
ShirtVNeck	6	0.12
Grand Total	50	1.00

Figure 1-9. The MLEs for the parameters under the Naive Bayes model

To calculate the probability of the model generating some observation $\mathbf{X}$, we simply multiply together the individual feature probabilities. For example:

$$\begin{aligned} & p(LongHairStraight, Red, Round, ShirtScoopNeck, White) \\ = \; & p(LongHairStraight) \times p(Red) \times p(Round) \times p(ShirtScoopNeck) \times p(Blue01) \\ = \; & 0.46 \times 0.16 \times 0.44 \times 0.38 \times 0.08 \\ = \; & 0.00098 \end{aligned}$$

Notice that this combination doesn't appear in the original dataset, but our model still allocates it a non-zero probability, so it is still able to be generated. Also, it has a higher probability of being sampled than say, (LongHairStraight, Red, Round, ShirtScoopNeck, Blue01), because white clothing appears more than blue clothing in the dataset.

Therefore, a Naive Bayes model is able to learn some structure from the data and use this to generate new examples that were not seen in the original dataset. It has estimated the probability of seeing each feature value independently, so that under the Naive Bayes assumption we can multiply these probabilities to build our full density function $p_\theta(\mathbf{x})$

Let's take a look at ten observations sampled from the model.

Figure 1-10. Ten new Wrodl styles, generated using the Naive Bayes model

For this simple problem, the Naive Bayes assumption that each feature is independent of every other feature is reasonable and therefore produces a good generative model.

Let's see what happens when this assumption breaks down…

Hello Wrodl! continued…

You feel a certain sense of pride as you look upon the ten new creations generated by your Naive Bayes model. Glowing with success, you turn your attention to another planet's fashion dilemma, but this time the problem isn't quite as simple.

On the conveniently named Planet Pixel, the dataset you are provided with doesn't consist of the 5 high-level features that you saw on Wrodl (hairColor, accessoriesType etc.) , but instead contains just the values of the 32x32 pixels that make up each image. Thus each observation now has $32 * 32 = 1024$ features and each feature can take any of 256 values (the individual colours in the palette).

Images from the new dataset are shown in Figure 1-11 and a sample of the pixel values for the first 10 observations in Figure 1-12

Figure 1-11. Fashions on Planet Pixel

face_id	...	pixel_458	pixel_459	pixel_460	pixel_461	pixel_462	pixel_463	pixel_464	pixel_465	pixel_466	pixel_467	...
0	...	49	14	14	19	7	5	5	12	19	14	...
1	...	43	10	10	17	9	3	3	18	17	10	...
2	...	37	12	12	14	11	4	4	6	14	12	...
3	...	54	9	9	14	10	4	4	16	14	9	...
4	...	2	2	5	2	4	4	4	4	2	5	...
5	...	44	15	15	21	14	3	3	4	21	15	...
6	...	12	9	2	31	16	3	3	16	31	2	...
7	...	36	9	9	13	11	4	4	12	13	9	...
8	...	54	11	11	16	10	4	4	19	16	11	...
9	...	49	17	17	19	12	6	6	22	19	17	...

Figure 1-12. The values of pixels 458-467 from the first 10 observations in the Planet Pixel dataset

You decide to try your trusty Naive Bayes model once more, this time trained on the pixel dataset. The model will estimate the maximum likelihood parameters that govern the distribution of the colour of each pixel so that you are able to sample from this distribution to generate new observations. However, when you do so, it is clear that something has gone very wrong…

Figure 1-13. Ten new Planet Pixel styles, generated by the Naive Bayes model

Rather than producing novel fashions, the model outputs ten very similar images that have no distinguishable accessories or clear blocks of hair or clothing colour. Why is this?

The challenges of generative modeling

Firstly, since the Naive Bayes model is sampling pixels independently, it has no way of knowing that two adjacent pixels are probably quite similar in shade, as they are part of the same item of clothing, for example. The model can generate the facial colour and mouth, as all of these pixels in the training set are roughly the same shade in each observation; however for

the t-shirt pixels, each pixel is sampled at random from a variety of different colours in the training set, with no regard to the colours that have been sampled in neighbouring pixels. Additionally, there is no mechanism for pixels near the eyes to form circular glasses shapes, or red pixels near the top of the image to exhibit a wavy pattern to represent a particular hairstyle, for example.

Secondly, there are now an incomprehensibly vast number of possible observations in the sample space. Only a tiny proportion of these are recognisable faces and an even smaller subset are faces that adhere to the fasion rules on Planet Pixel. Therefore if our Naive Bayes model is working directly with the highly correlated pixel values, the chance of it finding a satisfying combination of values is incredibly small.

In summary, on planet Wrodl individual features are independent and the sample space is relatively small so Naive Bayes works well. On Planet Pixel, the assumption that every pixel value is independent of every other pixel value clearly isn't true. Pixel values are highly correlated and the sample space is vast, so finding a valid face by sampling pixels independently is almost impossible. This explains why Naive Bayes models cannot be expected to work well on raw image data.

This example highlights the two key challenges that a generative model must overcome in order to be successful:

GENERATIVE MODELING CHALLENGES

- How does the model cope with the high degree of conditional dependence between features?
- How does the model find one of the tiny proportion of satisfying possible generated observations in amongst a high dimensional sample space?

Deep learning is the key to unlocking both of these challenges.

We need a model that can infer relevant structure from the data, rather than being told which assumptions to make in advance. This is exactly where deep learning excels and is one of the key reasons why the technique has driven the major recent advances in generative modeling. We shall be exploring deep learning in detail in Chapter 2

The fact that deep learning can form its own features in a lower dimensional space, means that it is a form of *representation learning*. It is important to understand the key concepts of representation learning before we tackle deep learning in the next chapter.

Representation learning

The core idea behind representation learning is that instead of trying to model the high-dimensional sample space directly, we should instead describe each observation in the training set using some low-dimensional *latent* space and then learn a mapping function that can take a point in the latent space and map it to a point in the original domain. In other words, each point in the latent space is the *representation* of some high-dimensional image.

What does this mean in practice? Let's suppose we have a training set consisting of greyscale images of biscuit tins (Figure 1-14)

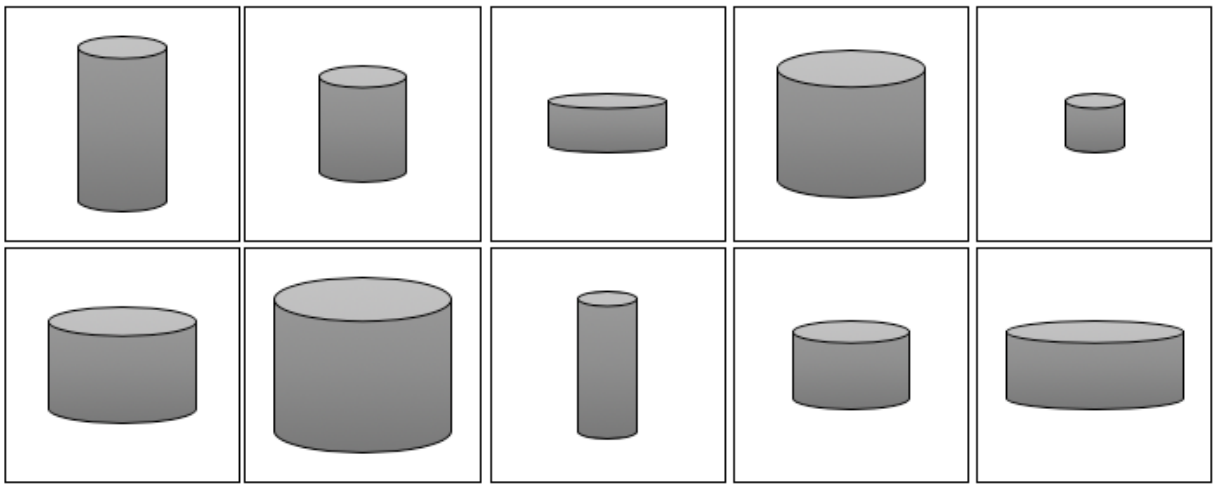


Figure 1-14. The biscuit tin dataset

To us, it is obvious that there are two features that can uniquely represent each of these tins—the height and width of the tin. Given a height and weight, we could draw the correpsonding tin, even if its image wasn't in the training set. However, this is not so easy for a machine—it would first need to establish that height and width are the two latent space dimensions that best describe this dataset, then learn the mapping function, f, that can take a point in this space and map it to a greyscale biscuit tin image. The resulting latent space of biscuit tins and generation process is shown in Figure 1-15.

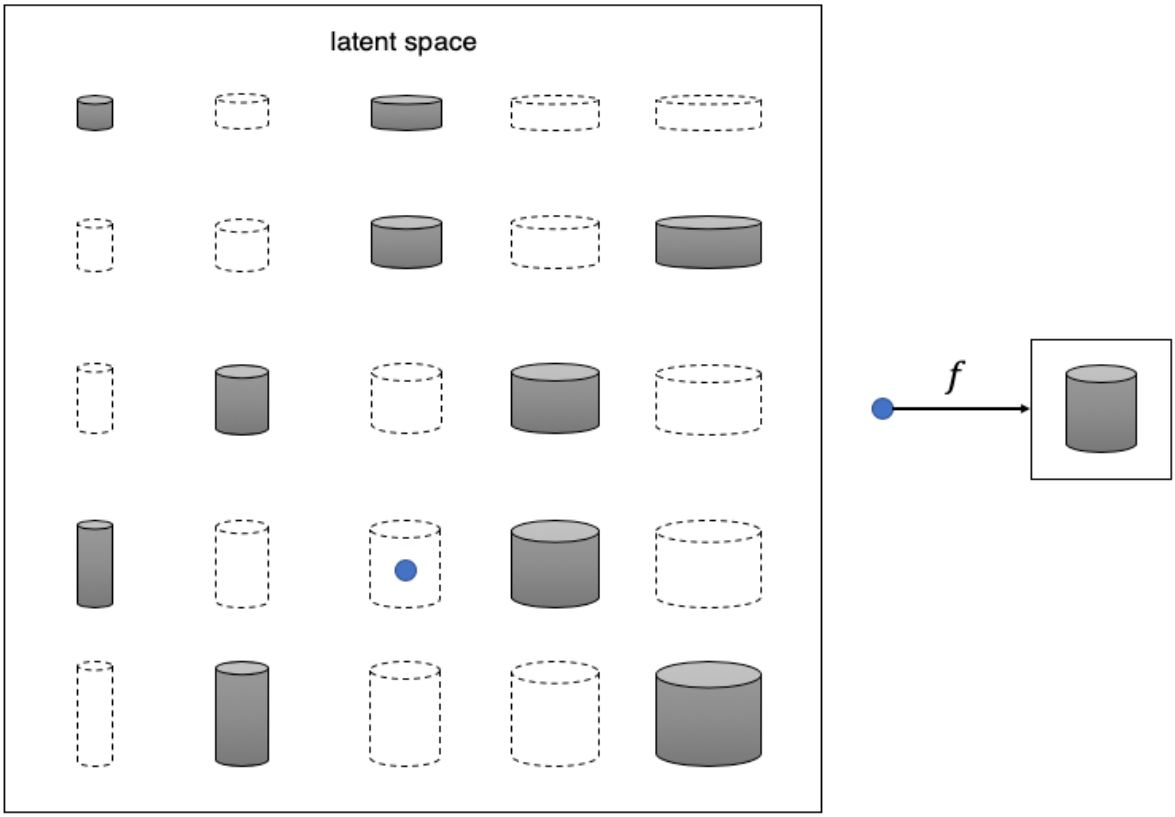


Figure 1-15. The latent space of biscuit tins and the function, f, that maps a point in the latent space to the original image domain.

Deep learning gives us the ability to learn the often highly complex mapping function f in a variety of ways. We shall explore some of the most important techniques in later chapters of this book. For now it is enough for us to understand at a high level what representaiton learning is trying to achieve.

One of the advantages of using representing learning is that we can perform operations within the more manageable latent space that affect high level properties of the image. It is not obvious how to adjust the shading of every single pixel to make a given biscuit tin image *taller*. However, in the latent space, it's simply a case of adding 1 to the *height* latent dimension, then applying the mapping function to return back to the image domain. We shall see an explicit example of this in the next chapter, applied not to biscuit tins, but to faces.

Representation learning comes so naturally to us as humans that you may never have stopped to think just how amazing it is that we can do it so effortlessless. Suppose you wanted to describe your appearance to someone who is looking for you in a crowd of people and doesn't know what you look like. You wouldn't start by stating the colour of pixel 1 of your hair, then pixel 2, then pixel 3 etc. Instead you would make the reasonable assumption that the other person has a general idea of what an average human looks like, then amend this baseline with features that describe groups of pixels such as *I have very blonde hair* or *I wear glasses*. With no more than ten or so of these statements, the person would be able to map the description back into pixels to

generate a image of you in their head. The image wouldn't be perfect, but it would be a close enough likeness to your actual appearance for them to find you in amongst possibly hundreds of other people, even though the person has never seen you before.

Note that representation learning doesn't just assign values to a given set of features such as *blondenesss of hair*, *height* etc, for some given image. The power of representation learning it that is actually learns which features are most important for it to describe the given observations and how to generate these features from the raw data. Mathematically speaking, it tries to find the highly non-linear *manifold* on which the data lies and then establish the dimensions required to fully describe this space—this is shown in Figure 1-16.

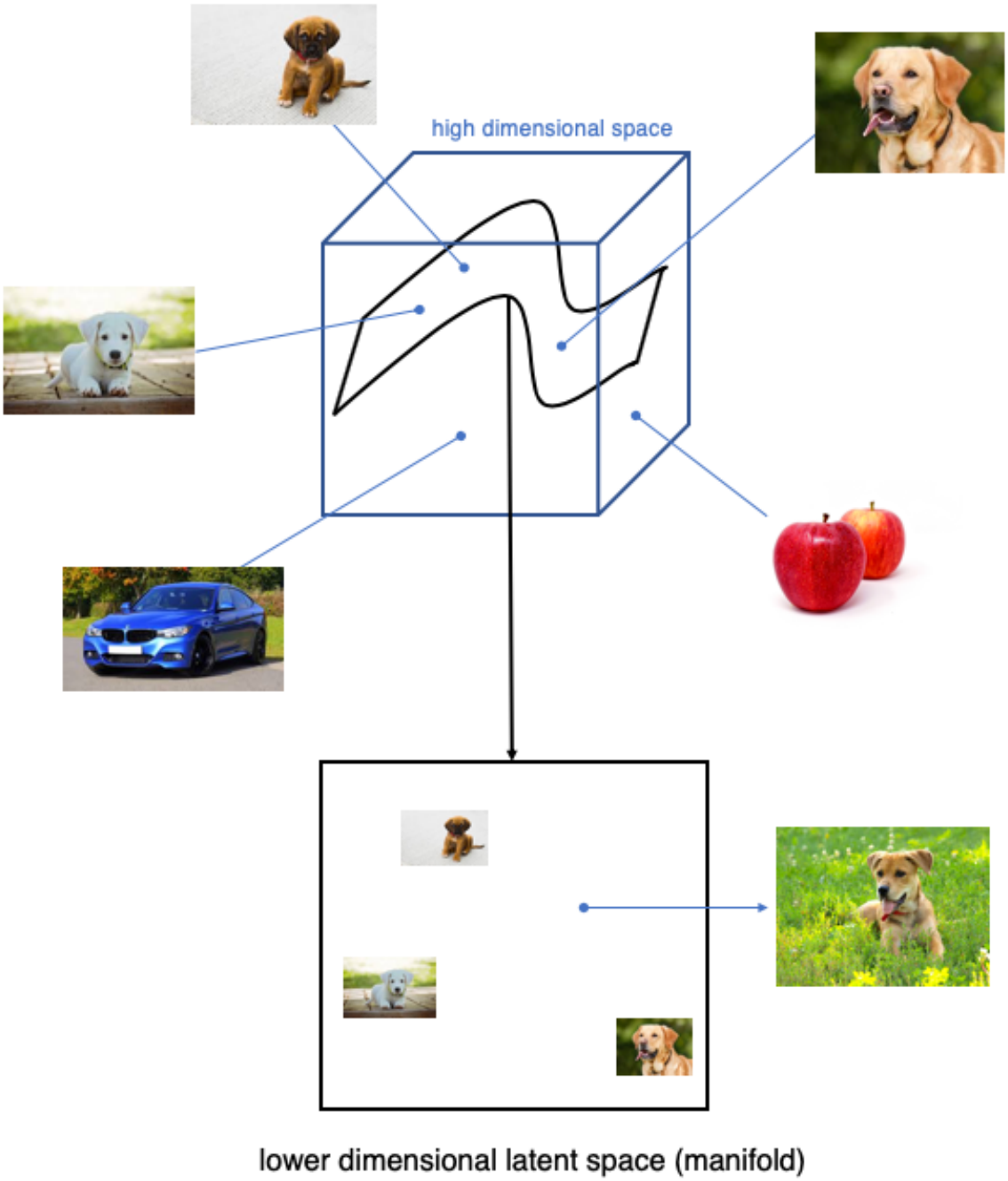


Figure 1-16. The cube represents the extremely high dimensional space of all images. Representation learning tries to find the smaller latent subspace or manifold on which particular kinds of image lie—for example, the dog manifold

In summary, representation learning establishes the most relevant high-level features that describe how groups of pixels are displayed so that is it likely that any point in the latent space is the representation of a well-formed image. By tweaking the value of features in the latent space we can produce novel representations that when mapped back to the original image domain, have a much better chance of looking *real* than if we tried to work directly with the individual raw pixels.

Now that we have introduced representation learning, which forms the backbone of many of the example generative deep learning in this book, it only remains to set up your environment so

that can begin building generative deep learning models of your own.

Setting up your environment

Throughout this book, there are many worked examples of how to build the models that we will be discussing in the text.

To get access to these worked examples, you'll need to clone the Git repository that accompanies this book. Git is an open source version control system and will allow you to copy the code locally so that you can run the notebooks on your own machine, or perhaps in a cloud-based environment. You may already have this installed, but if not, follow the instructions relevant to your operating system here [9] .

To clone the repository for this book, navigate to the folder where you would like to store the files and type the following into your terminal:

```
git clone https://github.com/davidADSP/GDL_code.git
```

Always make sure that you have the most up to date version of the codebase by running the following command.

```
git pull
```

You should now be able to see the files in a folder on your machine.

Next, we need to set up a virtual environment. This is simply a folder into which we install a fresh copy of Python and all of the packages that we will be using in this book. This way, you can be sure that your system version of Python isn't affected by any of the libraries that we will be using.

If you are using Anaconda, set up a virtual environment as follows:

```
conda install -n generative python=3.6 ipykernel
```

If not, you can install virtualenv and virtualenvwrapper with the following command[10]

```
pip install virtualenv virtualenvwrapper
```

You will also need to add the following lines to your shell startup script (e.g. .profile, .bashrc,

.bash_profile):

```
export WORKON_HOME=$HOME/.virtualenvs ❶
export VIRTUALENVWRAPPER_PYTHON=/usr/local/bin/python3 ❷
source /usr/local/bin/virtualenvwrapper.sh ❸
```

❶ This is where your virtual environments will be stored

❷ The default version of python to use when a virtual environment is created—make sure this points at Python 3, rather than Python 2.

❸ Reloads the virtualenvwrapper initialisation script

To create a virtual environment called *generative*, simply enter the following into your terminal:

```
mkvirtualenv generative
```

You'll know that you're inside the virtual environment because your terminal will show (generative) at the start of the prompt.

Now you can go ahead and install all the packages that we'll be using in this book with the following command:

```
pip install -r requirements.txt
```

Throughout this book, we will use Python 3. The requirements.txt file contains the names and version numbers of all the packages that you will need to run the examples.

To check everything works as expected, type *python* into your terminal and then try to import Keras (the deep learning library that we will be using extensively in this book). You should see a Python 3 prompt, with Keras reporting that it is using the Tensorflow backend as shown in Figure 1-17.

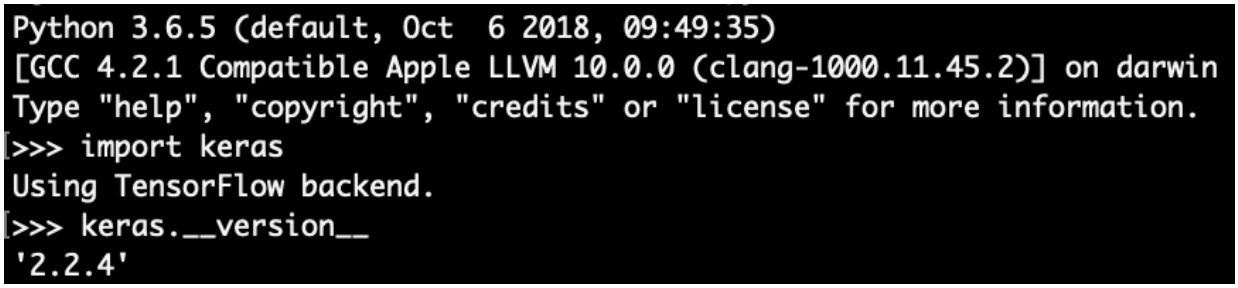


Figure 1-17. Setting up your environment

Lastly you will need to ensure you are set up to to access your virtual environment through a Jupyter notebooks on your machine. Jupyter is a way to interactively code in Python through your browser and is a great way to develop new ideas and share code. Most of the examples given in this book are written using Jupyter notebooks.

To do this, run the following command from your terminal inside your virtual environment.

```
python -m ipykernel install --user --name generative ❶
```

❶ This gives you access to the virtual environment that you've just set up (generative), inside jupyter notebooks.

To check that it has installed correctly, navigate in your terminal to the folder where you have cloned the book repository and type:

```
jupyter notebook
```

A window should open in your browser showing a screen similar to Figure 1-18. Then click the notebook you wish to run and from the *Kernel* > *Change kernel* dropdown, select the *generative* virtual environment.

Figure 1-18. Jupyter notebook

You are now ready to start building generative deep neural networks.

Summary

In this chapter we have introduced the field of generative modeling and seen how it is an important branch of machine learning that complements the more widely studied discriminative modeling. We have built our first simple generative model that utilises the Naive Bayes assumption to produce a smooth probability distribution that is able to represent some of the inherent structure in the data so that we can generate examples outside of the training set. We have also seen how these kinds of basic model can fail as the complexity of the generative task grows and have analysed the general challenges associated with generative modeling. Lastly, we have taken our first look at representation learning, a important concept that forms the core of many generative models.

In the next chapter, we will introduce deep learning and see how to use Keras to build models that can perform discriminative modeling tasks. This will give us the necessary foundations to go on to tackle generative deep learning in later chapters.

---

1 *https://arxiv.org/abs/1812.04948*

2 *https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf*

3 source: *https://www.eff.org/files/2018/02/20/malicious_ai_report_final.pdf*

4 Or integral if the sample space is continuous

5 If the sample space is discrete, $p(\mathbf{x})$ is simply the probability assigned to observing point $\mathbf{x}$.

6 Images sourced from *https://getavataaars.com/*

7 When a response variable y is present, the Naive Bayes assumption states that there is *conditional* independence between each pair of features, x_j, x_k, given y:.

8 The *-5* is due to the fact that the last parameter for each feature is forced to ensure that the sum of the parameters for this feature sums to 1.

9 *https://git-scm.com/download*

10 Full instructions here (*https://virtualenvwrapper.readthedocs.io/en/latest/*)

Chapter 2. Deep Learning

> **NOTE**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as he or she writes—so you can take advantage of these technologies long before the official release of these titles.

Let's start with a basic definition of deep learning:

> *Deep learning is a class of machine learning algorithm that uses multiple stacked layers of processing units to learn high level representations from unstructured data.*

To understand deep learning fully and particularly why it is so useful within generative modeling, we need to delve into this definition a bit further. Firstly, what do we mean by unstructured data, and its counterpart, structured data?

Structured and unstructured data

Many types of machine learning algorithm require **structured**, tabular data as input, arranged into columns of *features* that describe each observation. For example, a person's age, income and the number of website visits in the last month are all features that could help to predict if the person will subscribe to a particular online service in the coming month. We could use a structured table of these features to train a logistic regression, random forest or XGBoost model to predict the binary response variable—did the person subscribe (1) or not (0)? Here, each individual feature contains a nugget of information about the observation and the model would learn how these features interact to influence the response.

Unstructured data refers to any data that are not naturally arranged into columns of features, such as images, audio and text. There is of course spatial structure to an image, temporal structure to a recording and both spatial and temporal structure to video data, but since the data

do not arrive in columns of features, it is considered unstructured, as shown in Figure 2-1.

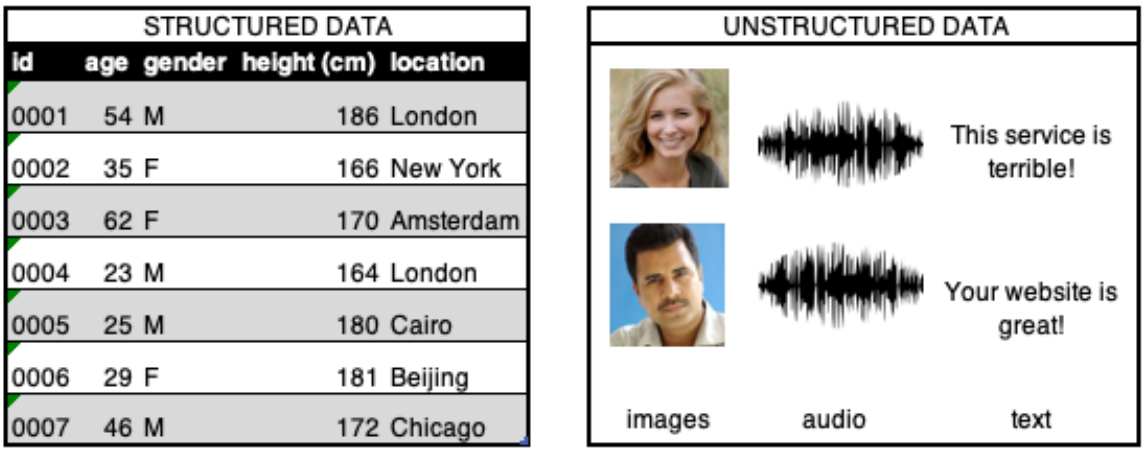


Figure 2-1. The difference between structured and unstructured data

When our data is unstructured, individual pixels, frequencies or characters are almost entirely uninformative. For example, knowing that pixel 234 of an image is a muddy shade of brown doesn't really help identify if the image is of a house or a dog. Knowing that character 24 of a sentence is an *e* doesn't help predict if the text is about football or politics.

Pixels or characters are really just the dimples of the canvas into which higher level informative features, such as an image of a chimney or the word *striker*, are embedded. If the chimney in the image was placed on the other side of the house, the image still contains a chimney, but this information is now carried by completely different pixels. If the word *striker* appears slightly earlier or later in the text, the text is still about football, but different character positions would provide this information. The granularity of the data combined with the high degree of spatial dependence destroys the concept of the pixel or character as an informative feature in its own right.

For this reason, if we train algorithms such as logistic regression, random forest or XGBoost on raw pixel values, the trained model will often perform poorly for all but the simplest of classification tasks. These models rely on the input features to be informative and not spatially dependent. Deep learning on the other hand, can learn how to build high level informative features by itself, directly from the unstructured data.

Deep learning can also be applied to structured data but its real power, especially with regards to generative modeling, comes from its ability to work with unstructured data. Most often, we want to generate unstructured data, such as new images, or original strings of text, which is why deep learning has had such a profound impact on the field of generative modeling.

Deep neural networks

The majority of deep learning systems are artificial neural networks (ANNs, or just *neural networks* for short) with multiple stacked hidden layers. For this reason *deep learning* has now almost become synonymous with *deep neural networks*. However, it is important to point out that any system that employs many layers to learn high level representations of the input data is also a form of deep learning (e.g. Deep Belief Networks / Deep Boltzmann Machines).

Let's start by taking a high-level look at how a deep neural network can make a prediction about a given input.

A deep neural network consists of a series of stacked **layers**. Each layer contains **units** (or nodes), that are connected to the previous layer's units through a set of **weights**. As we shall see, there are many different types of layer but one of the most common is the *dense* layer that connects all units to every unit in the previous layer. By stacking layers, the units in each subsequent layer can represent increasingly sophisticated aspects of the original input.

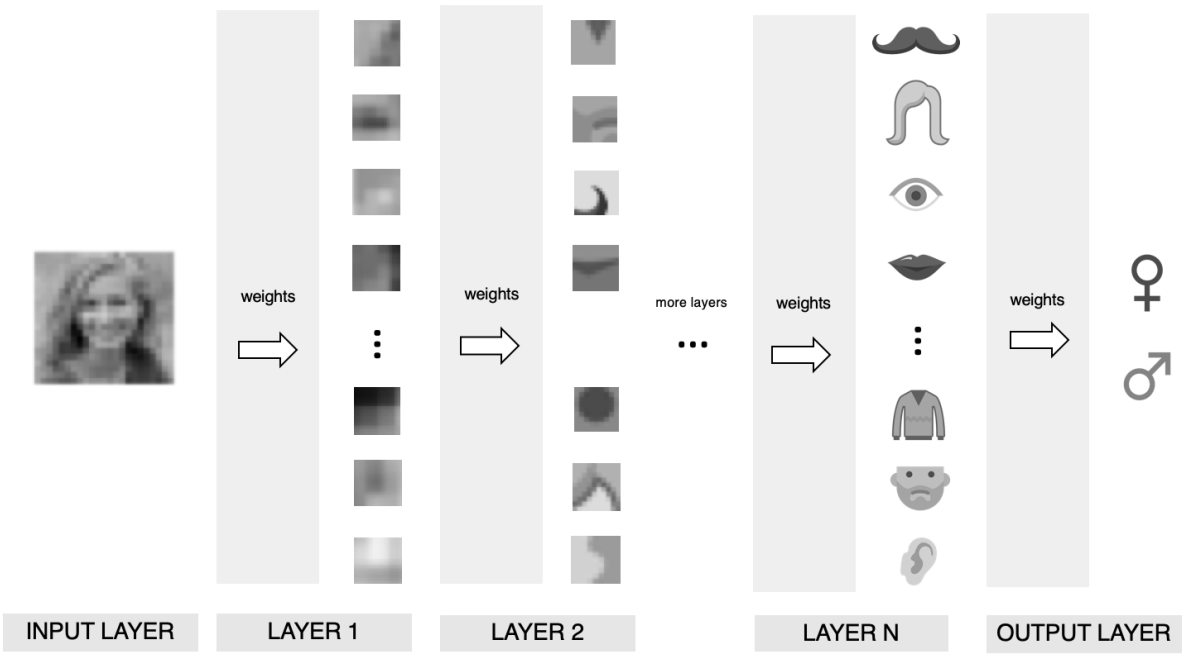


Figure 2-2. Deep learning conceptual diagram.

For example, in Figure 2-2, layer 1 consists of units that activate more strongly when they detect particular basic properties of the input image, such as edges. The output from these units is then passed to the units of layer 2, which are able to use this information to detect slightly more complex features—and so on, through the network. The final output layer is the culmination of this process, where the network outputs a set of numbers that can be converted into probabilities, to represent the chance that the original input belongs to one of *n* categories.

The magic of deep neural networks lies in finding the set of weights for each layer that result in the most accurate predictions. The process of finding these weights is what we mean by *training*

the network.

During the training process, batches of images are passed through the network and the output is compared to the ground truth. The error in the prediction is then propagated backwards through the network, adjusting each set of weights a small amount in the direction that improves the prediction most significantly. This process is appropriately called *back-propagation*. Gradually, each unit becomes skilled at identifying a particular feature that ultimately helps the network to make better predictions.

Deep neural networks can have any number of middle or *hidden* layers—for example, ResNet (2015, He et al.) designed for image recognition, contains 152 layers. We shall see in Chapter 3 that we can use deep neural networks to influence high-level features of an image such as the hair color or expression of a face, by manually tweaking the values of these hidden layers. This is only possible because the deeper layers of the network are capturing high-level features that we can work with directly.

Next, we'll dive straight into the practical side of deep learning and get set up with Keras and Tensorflow, the two libraries that will enable you to start building your own generative deep neural networks.

Keras and Tensorflow

Keras is a high-level Python library for building neural networks and is the core library that we shall be using in this book. It is extremely flexible and has a very user-friendly API, making it an ideal choice for getting started with deep learning. Moreover Keras provides numerous useful building blocks that can be plugged together to create highly complex deep learning architectures through its functional API.

Keras is not the library that performs the low-level array operations required to train neural networks. Instead Keras utilses one of three backend libraries for this purpose—Tensorflow, CNTK, or Theano. You are free to choose whichever you are most comfortable with, or whichever library works fastest for a particular network architecture. For most purposes, it doesn't matter which you choose as you usually won't be coding directly using the underlying backend framework. In this book we use Tensorflow as it is the most widely adopted and documented of the three.

Tensorflow is an open-source Python library for machine learning, developed by Google. It is now one of the most utilized frameworks for building machine learning solutions, with particular emphasis on the manipulation of tensors, hence the name. Within the context of deep learning, tensors are simply multi-dimensional arrays that store the data as it flows through the network. As we shall see, understanding how each layer of a neural network changes the shape

of the data as it flows through the network is a key part of truly understanding the mechanics of deep learning.

If you are just getting started with deep learning, I highly recommend that you choose Keras with a Tensorflow backend as your toolkit. In combination, these two libraries are a powerful combination that will allow you to build any network that you can think of in a production environment, whilst also giving you an easy-to-learn API that is so important for rapid development of new ideas and concepts.

A step by step guide to building a deep neural network

Let's start by seeing how easy it is to build a deep neural network in Keras.

We will be working through the jupyter notebook in the book repository called `02_01_deep_learning_first_deep_neural_network.ipynb`.

Loading the data

We will be using the CIFAR-10 dataset—a collection of 60,000 32 x 32 pixel color images, that comes bundled in with Keras out of the box. Each image is classified into exactly one of 10 classes, as shown in Figure 2-3

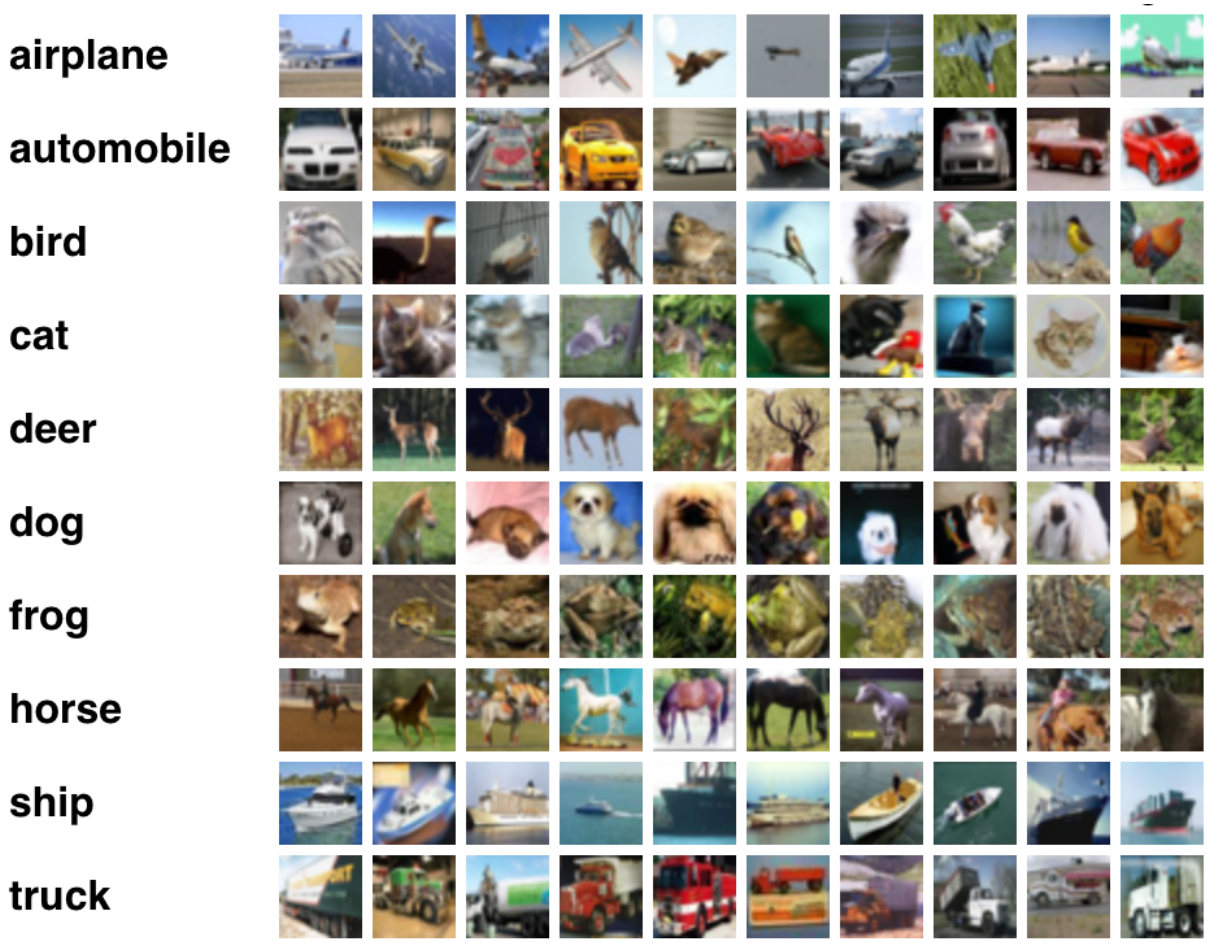


Figure 2-3. Example images from the CIFAR-10 dataset [1]

The following code loads and scales the data.

```
import numpy as np
from keras.utils import to_categorical
from keras.datasets import cifar10

(x_train, y_train), (x_test, y_test) = cifar10.load_data() ❶

NUM_CLASSES = 10

x_train = x_train.astype('float32') / 255.0 ❷
x_test = x_test.astype('float32') / 255.0

y_train = to_categorical(y_train, NUM_CLASSES) ❸
y_test = to_categorical(y_test, NUM_CLASSES)
```

❶ Loads the CIFAR-10 dataset. x_train and x_test are numpy arrays with shape (50000, 32, 32, 3) and (10000, 32, 32, 3) respectively. y_train and y_test are numpy arrays with shape (50000, 1) and (10000, 1) respectively, containing the integer labels in the range [0, 9] for the class of each image.

❷ By default the image data consists of integers between 0 and 255 for each pixel channel. Neural networks work best when each input is inside the range -1 to 1, so we need to divide by 255.

❸ We also need to change the integer labeling of the images to one-hot-encoded vectors. If the class integer label of an image is *i*, then its one-hot-encoding is a vector of length 10 (the number of classes) which has zeros in all but the 'i'th element, which is 1. The new shapes of y_train and y_test are therefore (50000, 10) and (10000, 10) respectively.

It's worth noting the shape of the image data in x_train—(50000, 32, 32, 3). The first dimension of this array references the index of the image in the dataset, the second and third relate to the size of the image and the last is the channel (i.e. red, green or blue, since these are RGB images). There are no *columns* or *rows* in this dataset—instead, this is a **tensor** with 4 dimensions. Each *slot* in this tensor contains a single number—for example, the following entry refers to the green channel (1) value of the pixel in the (12,13) position of image 54.

```
x_train[54, 12, 13, 1]
# 0.36862746
```

Building the model

In Keras there are two ways to define the structure of your neural network—as a Sequential model or using the Functional API.

A Sequential model is useful for quickly defining a linear stack of layers (i.e. one layer follows on directly from the previous layer without any branching). However, many of the models in this book require that the the output from a layer is passed to multiple separate layers beneath it, or conversely, that a layer receives input from multiple layers above it.

To be able to build networks with branches, we need to use the Functional API, which is a lot more flexible. I recommend that even if you are just starting out building linear models with Keras, you still use the Functional API rather than Sequential models, since it will serve you better in the long run as your neural networks become more architecturally complex. The Functional API will give you complete freedom over the design of your deep neural network.

To demonstrate the difference between the two methods, Example 2-1 shows the network coded using a Sequential model and the Functional API. Feel free to try both and observe that they give the same result:

Example 2-1. The architecture using a Sequential Model

```
from keras.models import Sequential
from keras.layers import Flatten, Dense

model = Sequential([
    Dense(200, activation = 'relu', input_shape=(32, 32, 3)),
    Flatten(),
    Dense(150, activation = 'relu'),
    Dense(10, activation = 'softmax'),
])
```

Example 2-2. The architecture using the Functional API

```
from keras.layers import Input, Flatten, Dense
from keras.models import Model

input_layer = Input(shape=(32,32, 3))

x = Flatten()(input_layer)

x = Dense(units=200, activation = 'relu')(x)
x = Dense(units=150, activation = 'relu')(x)

output_layer = Dense(units=10, activation = 'softmax')(x)

model = Model(input_layer, output_layer)>
```

Here, we are using three different types of layer—Input, Flatten and Dense.

The **Input** layer is an entry point into the network. We tell the network the shape of each data element to expect as a tuple. Notice that we do not specify the batch size—this isn't necessary as we can pass any number of images into the Input layer simultaneously. We do not need to explicitly state the batch size in the input layer definition.

Next we flatten this input into a 1D vector, using a **Flatten** layer. This results in a vector of length 3072 (= 32 x 32 x 3). The reason we do this is because the subsequent Dense layer requires that its input is flat, rather than a multidimensional array. As we shall see later, other layer types do require multidimensional arrays as input, so you need to be aware of the required input and output shape of each layer type to understand when it is necessary to use Flatten.

The **Dense** layer is perhaps the most fundamental layer type in any neural network. It is a layer that contains a given number of units, that are densely connected to the previous layer—that is, every unit in the layer is connected to every unit in the previous layer, through a single connection which carries a weight (which can be positive or negative). The output from a given unit is the weighted sum of the input it receives from the previous layer, that is then passed through a non-linear **activation function** before being sent to the following layer. The

activation function is critical to ensure the neural network is able to learn complex functions and doesn't just output a linear combination of its input.

There are many kinds of activation function—the three most important are ReLU, sigmoid and softmax.

The **ReLU** (rectified linear unit) activation function is defined to be 0 if the input is negative and is otherwise equal to the input. The **LeakyReLU** activation function, is very similar to the ReLU activation layer, with one key difference. Whereas the ReLU activation function returns zero for input values less than 0, the LeakyReLU function returns a small negative number, proportional to the input. ReLU units can sometimes *die* if they always output zero, because of a large bias towards negative values preactivation. In this case, the gradient is zero and therefore no error is propagated back through this unit. LeakyReLU activations fix the issue by always ensuring the gradient is non-zero. ReLU based functions are now established to be the most reliable activation to use between the layers of a deep network to encourage stable training.

The **sigmoid** activation is useful if you wish the output from the layer to be scaled between 0 and 1—for example, for a binary classification problems with one output unit or multilabel classification problems, where each observation can belong to more than one class. Figure 2-4 shows ReLU, LeakyReLU and sigmoid activation functions side by side for comparison.

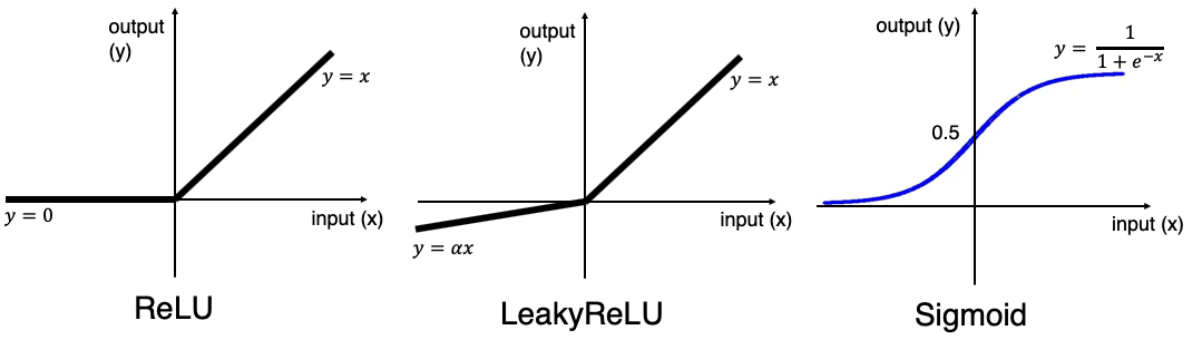


Figure 2-4. The ReLU, LeakyReLU and sigmoid activation functions

The **softmax** activation is useful if you want the total sum of the output from the layer to equal 1—for example, for a multiclass classification problems, where each observation only belongs to exactly one class. It is defined as:

$$y_i = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}}$$

Here, *J* is the total number of units in the layer. In our neural network, we use softmax activation in the final layer to ensure that the output is a set of 10 probabilities that sum to 1, which can be interpreted as the chance that the image belongs to each class.

In Keras, activation functions can also be defined in a separate layer as follows:

```
x = Dense(units=200)(x)
x = Activation('relu')(x)
```

This is equivalent to:

```
x = Dense(units=200, activation = 'relu')(x)
```

In our example, we pass the input through two dense hidden layers, the first with 200 units and the second with 150, both with ReLU activation functions. A diagram of the total network is shown in Figure 2-5.

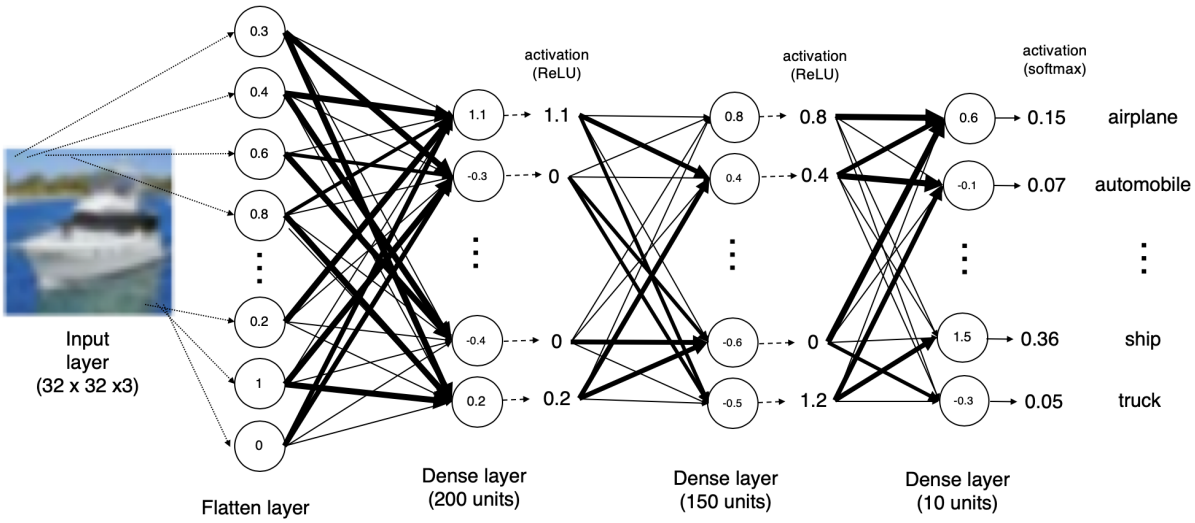


Figure 2-5. A diagram of the neural network trained on CIFAR-10 data

The final step is to define the model itself, using the **Model** class. In Keras a *Model* is defined by the input and output layers. In our case, we have one input layer that we defined earlier and the output layer is the final Dense layer of 10 units. It is possible to define models with multiple input and output layers—we shall see this in action later in the book.

In our example, as required, the shape of our Input layer matches the shape of *x_train* and the shape of our Dense output layer matches the shape of *y_train*. To see this we can use the *.summary()* method to see the shape of the network at each layer as shown in Figure 2-6.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 32, 32, 3)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 3072)              0
_________________________________________________________________
dense_1 (Dense)              (None, 200)               614600
_________________________________________________________________
dense_2 (Dense)              (None, 150)               30150
_________________________________________________________________
dense_3 (Dense)              (None, 10)                1510
=================================================================
Total params: 646,260
Trainable params: 646,260
Non-trainable params: 0
_________________________________________________________________
```

Figure 2-6. The summary of the model

Notice how Keras uses *None* as a marker to show that it doesn't yet know the number of observations that will be passed into the network. In fact it doesn't need to—we could just as easily pass one observation through the network at a time as 1,000. That's because tensor operations are conducted across all observations simultaneously using linear algebra—this is the part handled by Tensorflow. It is also the reason why you get a performance increase when training deep neural networks on GPUs instead of CPUs—GPUs are optimized for large array multiplications since these calculations are also necessary for complex graphics manipulation.

The *summary* method also gives the number of parameters (weights) that will be trained at each layer. If ever you find that your model is training too slowly, check the summary to see if there are any layers that contain a huge number of weights. If so, you should consider whether the number of units in the layer could be reduced to speed up training.

Compiling the model

In this step, we compile the model with an optimizer and a loss function.

```
from keras.optimizers import Adam

opt = Adam(lr=0.0005)
model.compile(loss='categorical_crossentropy', optimizer=opt,  metrics=['a
```

The loss function is used by the neural network to compare its predicted output to the ground truth. It returns a single number for each observation—the greater this number, the worse the network has performed for this observation.

Keras provides many inbuilt loss functions to choose from, or you can create your own. Three of the most commonly used are *mean_squared_error*, *categorical_crossentropy* and *binary_crossentropy*. It is important to understand when it is appropriate to use each.

If your neural network is designed to solve a regression problem (i.e. the output is continuous), then you can use the **mean_squared_error** loss. This is the mean of the squared difference between the ground truth y_i and predicted value p_i of each output unit, where the mean is taken over all n output units.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - p_i)^2$$

If you are working on a classification problem where each observation only belongs to one class, then **categorical_crossentropy** is the correct loss function. This is defined as follows:

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

Lastly, if you are working on a binary classification problem with one output unit, or a multi-label problem where each observation can belong to multiple classes simultaneously, you should use **binary_crossentropy**.

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

The optimizer is the algorithm that will be used to update the weights in the neural network based on the gradient of the loss function. One of the most commonly used and stable optimizers is **Adam** [2] . In most cases, you shouldn't need to tweak the default parameters of the Adam optimizer, except for the learning rate. The greater the learning rate, the larger the change in weights at each training step. Whilst training is initially faster with a large learning rate, the downside is that it may result in less stable training and may not find minima of the loss function. This is a parameter that you may want to tune or adjust during training.

Another common optimizer that you may come across is **RMSProp**. Again, you shouldn't need to adjust the parameters of this optimizer too much, but it is worth reading the Keras documentation to understand the role of each parameter.

We pass both the loss function and optimizer into the *compile* method of the model, as well as a *metrics* parameter where we can specify any additional metrics that we would like reporting during training, such as *accuracy*.

Training the model

Thus far, we haven't shown the model any data and have just set up the architecture and compiled the model with a loss function and optimizer.

To train the model, simply call the *fit* method, as shown below:

```
model.fit(x_train ❶
          , y_train ❷
          , batch_size= 32 ❸
          , epochs= 10 ❹
          , shuffle= True ❺
          )
```

❶ The raw image data

❷ The one-hot-encoded class labels

❸ The **batch_size** determines how many observations will be passed to the network at each training step

❹ The **epochs** determine how many times the network will be shown the full training data

❺ If shuffle = True, the batches will be drawn randomly without replacement from the training data at each training step

This will start training a deep neural network to predict the category of an image from the CIFAR-10 dataset.

The training process works as follows:

First, the weights of the network are initialized to small random values. Then the network performs a series of training steps.

At each training step, one batch of images is passed through the network and the errors are backpropagated to update weights. The **batch_size** determines how many images are in each training step batch. The larger the batch size, the more stable the gradient calculation, but the slower each training step. It would be far too time consuming and computationally intense to use the entire dataset to calculate the gradient at each training step, so generally a batch size between 32 and 128 is used. It is also now recommended practice to increase the batch size as training progresses [3] .

This continues until all observations in the dataset have been seen once. This completes the first **epoch**. The data is then passed through the network again in batches as part of the second epoch. This process repeats until the number of specified epochs have elapsed.

During training, Keras outputs the progress of the procedure, as shown in Figure 2-7. We can see that the training dataset of 50,000 observations has been shown to the network 10 times (i.e. over 10 epochs), at a rate of approximately 160 microseconds per observation. The categorical_crossentropy loss has fallen from 1.842 to 1.357, resulting in an accuracy increase from 33.5% after the first epoch to 51.9% after the 10th epoch.

```
model.fit(x_train
          , y_train
          , batch_size=BATCH_SIZE
          , epochs=EPOCHS
          , shuffle=True)
```

```
Epoch 1/10
50000/50000 [==============================] - 8s 164us/step - loss: 1.8424 - acc: 0.3354
Epoch 2/10
50000/50000 [==============================] - 8s 154us/step - loss: 1.6592 - acc: 0.4048
Epoch 3/10
50000/50000 [==============================] - 8s 153us/step - loss: 1.5733 - acc: 0.4381
Epoch 4/10
50000/50000 [==============================] - 8s 154us/step - loss: 1.5232 - acc: 0.4579
Epoch 5/10
50000/50000 [==============================] - 8s 155us/step - loss: 1.4874 - acc: 0.4698
Epoch 6/10
50000/50000 [==============================] - 8s 165us/step - loss: 1.4569 - acc: 0.4799
Epoch 7/10
50000/50000 [==============================] - 10s 208us/step - loss: 1.4281 - acc: 0.4887
Epoch 8/10
50000/50000 [==============================] - 8s 165us/step - loss: 1.4038 - acc: 0.4984
Epoch 9/10
50000/50000 [==============================] - 8s 153us/step - loss: 1.3797 - acc: 0.5084
Epoch 10/10
50000/50000 [==============================] - 8s 155us/step - loss: 1.3571 - acc: 0.5187
```

Figure 2-7. The output from the fit method

Evaluating the model

We know the model achieves an accuracy of 51.9% on the training set, but how does it perform on data it has never seen?

To answer this question we can use the *.evaluate()* method provided by Keras:

```
model.evaluate(x_test, y_test)
```

```
10000/10000 [==============================] - 1s 55us/step

[1.4358007415771485, 0.4896]
```

Figure 2-8. The output from the evaluate method

The output from this method is a list of the metrics we are monitoring—categorical_crossentropy and accuracy. We can see that model accuracy is still 49.0% even on images that it has never seen before. Note that if the model was guessing randomly, it would achieve 10%, since there are 10 classes, so 50% is still a good result given that we have used a very basic neural network.

We can view some of the predictions on the test set using the *predict* method.

```
CLASSES = np.array(['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog'

preds = model.predict(x_test) ❶
preds_single = CLASSES[np.argmax(preds, axis = -1)] ❷
actual_single = CLASSES[np.argmax(y_test, axis = -1)]
```

❶ preds is an array of shape (10000, 10)—i.e. a vector of 10 class probabilities for each observation.

❷ We convert this array of probabilities back into a single prediction using numpy's argmax function. Here, *axis = -1* tells the function to collapse the array over the last dimension (the classes dimension), so that the shape of preds_single is then (10000, 1).

We can view some the images alongside their label and prediction with the following code. As expected, around half are correct.

```
import matplotlib.pyplot as plt

n_to_show = 10
indices = np.random.choice(range(len(x_test)), n_to_show)

fig = plt.figure(figsize=(15, 3))
fig.subplots_adjust(hspace=0.4, wspace=0.4)

for i, idx in enumerate(indices):
    img = x_test[idx]
    ax = fig.add_subplot(1, n_to_show, i+1)
    ax.axis('off')
    ax.text(0.5, -0.35, 'pred = ' + str(preds_single[idx]), fontsize=10, h
    ax.text(0.5, -0.7, 'act = ' + str(actual_single[idx]), fontsize=10, ha
    ax.imshow(img)
```

Figure 2-9. Some predictions made by the model, alongside the actual label

Congratulations—you've just built your first deep neural network using Keras and used it to make predictions on new data. Even though this is a supervised learning problem, when we come to building generative models in future chapters, many of the core ideas from this network such as loss functions, activation functions and understanding layer shapes will still be extremely important. Next we'll look at ways of improving this model, by introducing a few new layer types.

Improving the model

One of the reasons our network isn't yet performing as well as it might is because there isn't anything in the network that takes into account the spatial structure of the input images. In fact, our first step is to flatten the image into a single vector, so that we can pass it to the first Dense layer!

To achieve this we need to use a *convolutional* layer.

Convolutional layers

Firstly, we need to understand what we mean by a *convolution*. To perform a convolution, we take a given *filter* (an array), multiply pixelwise with a portion of an input tensor of the same size, and sum the result. The output is large when the portion of the tensor closely matches the filter and small when the portion of the tensor is the negative of the filter.

In Figure 2-10, we show a simple example of a 3x3x1 filter being applied to portions of a greyscale image. A colour image has 3 channels (red, green, blue) rather than one and therefore a 3x3x3 filter would be applied instead.

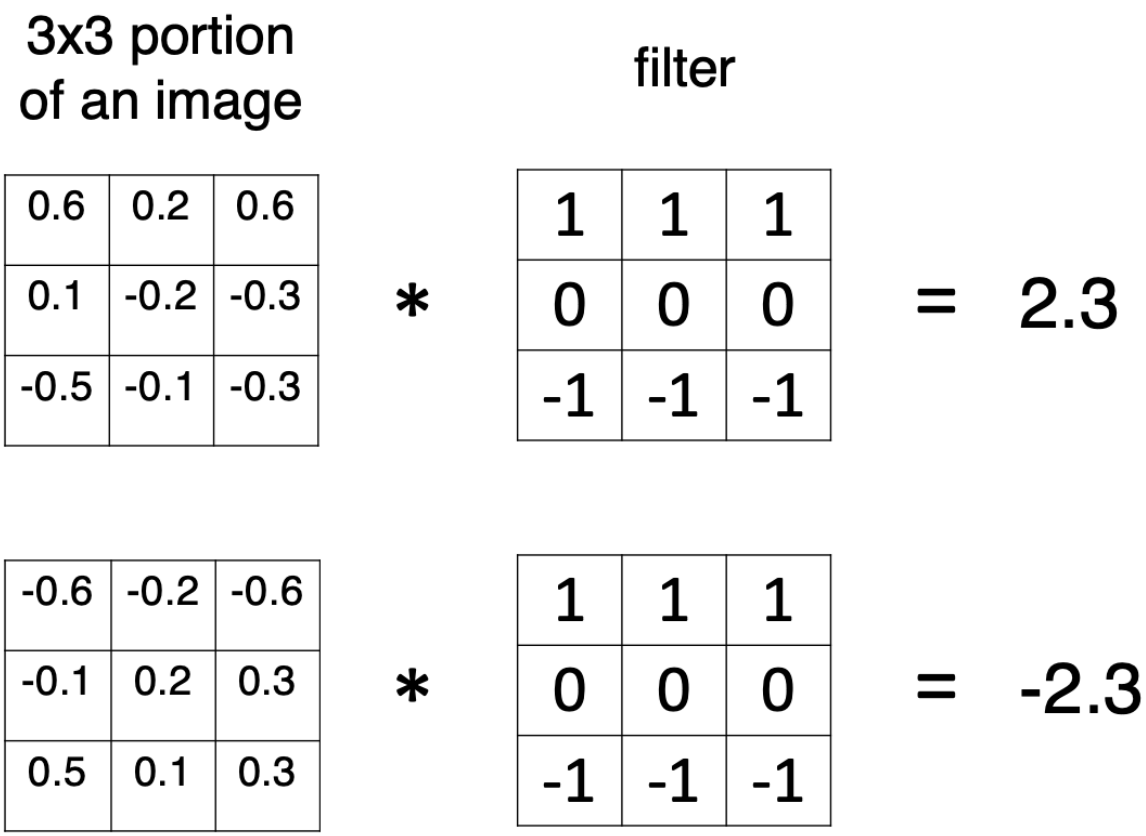


Figure 2-10. The convolution operation

If we move the filter across the entire image, from left to right and top to bottom, recording the convolutional output as we go, we obtain a new array that picks out a particular feature of the input, depending on the values in the filter.

This is exactly what a convolutional layer is designed to do, but with multiple filters rather than just one. For example, Figure 2-11 shows two filters that highlight horizontal and vertical edges. You can see this convolutional process worked through manually in the book repository notebook `02_02_deep_learning_convolutions.ipynb`.

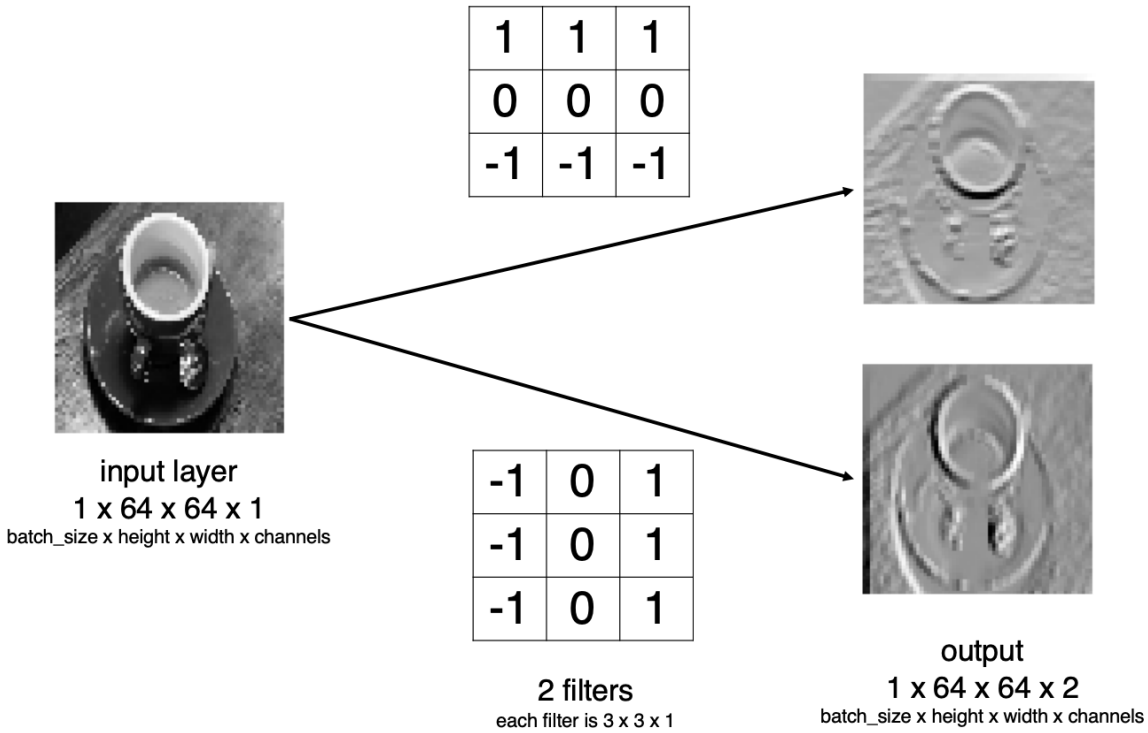


Figure 2-11. Two convolutional filters applied to a greyscale image

In Keras, the **Conv2D** layer can apply convolutions to an input tensor with 2 spatial dimensions (such as an image). For example, the Keras code corresponding to the diagram in Figure 2-11 is shown below:

```
input_layer = Input(shape=(64,64,1))

conv_layer_1 = Conv2D(
    filters = 2
    , kernel_size = (3,3)
    , strides = 1
    , padding = "same"
    )(input_layer)
```

STRIDES

The `strides` parameter tells the layer the step size to use when moving the filters across the input. Increasing the stride therefore reduces the size of the output tensor. For example when `strides = 2`, the height and width of the output tensor will be half the size of the input tensor.

This is useful for reducing the spatial size of the tensor as it passes through the network, whilst increasing the number of channels.

PADDING

The `padding = "same"` input parameter pads the input data with zeros so that the output size from the layer is exactly the same as the input size when `strides = 1`.

Figure 2-12 shows a 3x3 kernel being passed over a 5x5 input image, with `padding = "same"` and `strides = 1`. The output size from this convolutional layer would also be 5x5, as the padding allows the kernel to extend over the edge of the image, so that it fits 5 times in both directions. Without padding, the kernel could only fit 3 times along each directions, giving an output size of 3x3.

Setting `padding = "same"` is a good way to ensure that you are able to easily keep track the size of the tensor as it passes through many convolutional layers.

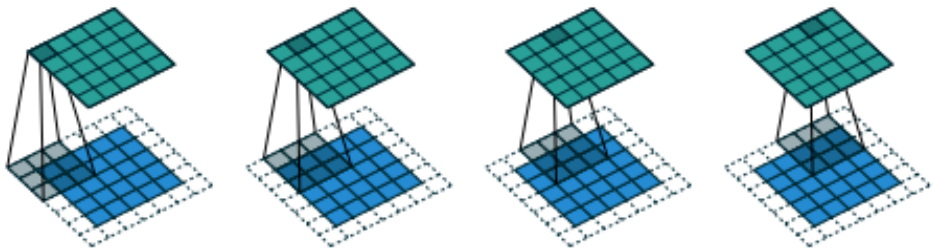


Figure 2-12. This example shows a 3x3x1 kernel (grey) being passed over a 5x5x1 input image (blue), with `padding="same"`, and `strides = 1` to generate the 5x5x1 output (green).[4]

The values stored in the filters are the weights that are learnt by the neural network through training. Initially these are random, but gradually the filters adapt their weights to start picking out interesting features such as edges, or particular colour combinations.

The output of a convolutional layer is another another 4-dimensional tensor (batch_size, height, width, filters), so we can stack Conv2D layers on top of each other to grow the depth of our neural network. It's really important to understand how the shape of the tensor changes as data flows through from one convolutional layer to the next. To demonstrate this, let's imagine we are applying Conv2D layers to the CIFAR-10 dataset. This time, instead of one input channel (greyscale) we have three (red, green and blue).

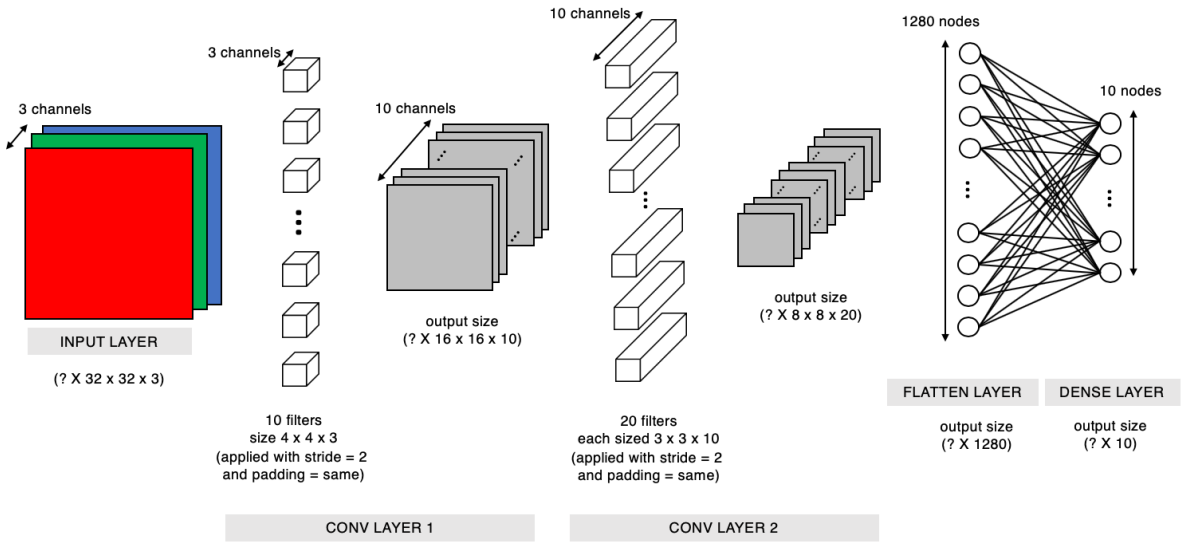


Figure 2-13. A diagram of a convolutional neural network

Figure 2-13 represents the following network in Keras:

```
input_layer = Input(shape=(32,32,3))

conv_layer_1 = Conv2D(
    filters = 10
    , kernel_size = (4,4)
    , strides = 2
    , padding = 'same'
    )(input_layer)

conv_layer_2 = Conv2D(
    filters = 20
    , kernel_size = (3,3)
    , strides = 2
    , padding = 'same'
    )(conv_layer_1)

flatten_layer = Flatten()(conv_layer_2)

output_layer = Dense(units=10, activation = 'softmax')(flatten_layer)

model = Model(input_layer, output_layer)
```

We can use the `model.summary()` method to see the shape of the tensor as it passes through the network.

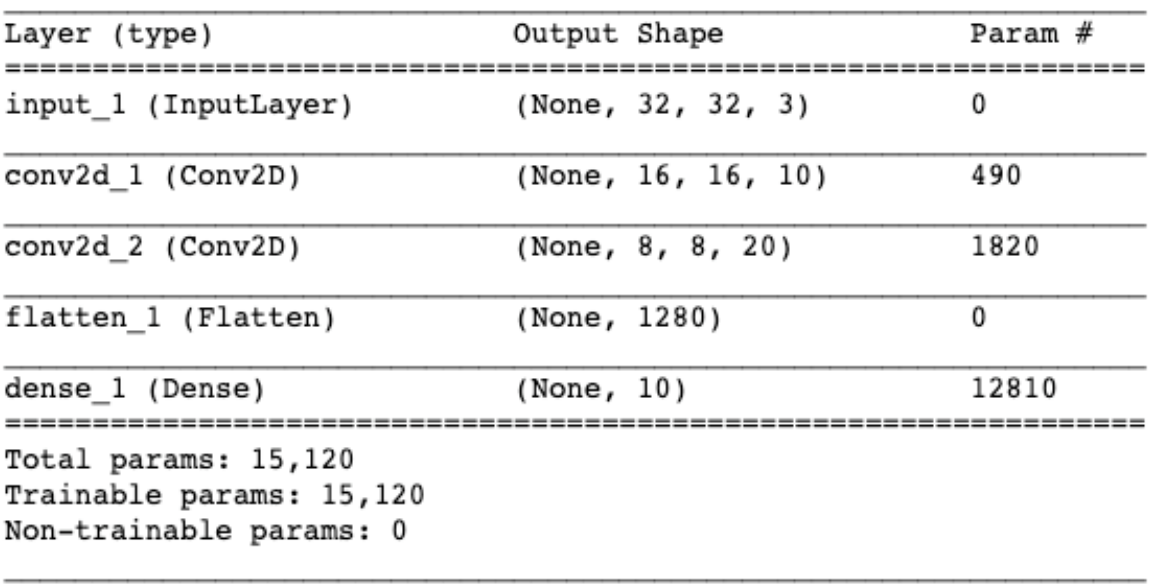


Figure 2-14. A convolutional neural network summary

Let's analyze this network from input through to output. Firstly, the input shape is (None x 32 x 32 x 3)—Keras uses `None` to represent the fact that we can pass any number of images through the network simultaneously. Since the network is just performing tensor algebra, we don't need to pass images through the network individually, but instead can pass them through together as a *batch*.

The shape of the filters in the first convolutional layer is 4 x 4 x 3. This is because we have chosen the filter to have height and width of 4 (kernel_size = (4,4)) and there are 3 channels in the preceding layer (red, green and blue). Therefore, the number of parameters (or weights) in the layer is (4 x 4 x 3 + 1) x 10 = 490, where the + 1 is due to the inclusion of a bias term attached to each of the filters. It's worth remembering that the depth of the filters in a layer is *always* the same as the number of channels in the preceding layer.

As before, the output from each filter when applied to each 4 x 4 x 3 section of the input image will be the pixelwise multiplication of the filter weights and the area of the image it is covering. As `strides = 2` and `padding = "same"`, the width and height of the output are both halved to 16 and since there are 10 filters, the output of the first layer is a tensor of shape None x 16 x 16 x 10.

In general, the size of the output from a convolutional layer with `padding = "same"` is:

$$\left(None, \frac{previous\ height}{stride}, \frac{previous\ width}{stride}, num\_filters\right)$$

In the second convolutional layer, we choose the filters to be 3 x 3 and they now have depth 10, to match the number of channels in the previous layer. Since there are 20 filters in this layer,

this gives the total number of parameters (weights) to be (3 x 3 x 10 + 1) x 20 = 1820. Again, we use a stride of 2, with `padding = "same"`, so the output width and height both halve, giving an overall output shape of None x 8 x 8 x 20.

After applying a series of Conv2D layers, we need to flatten the tensor, using the Keras *Flatten* layer. This results in a set of 8 x 8 x 20 = 1280 units that we can connect to a final 10-unit Dense layer with softmax activation, that represents the probability of each category in a 10-category classification task.

This examples demonstrates how we can chain convolutional layers together to create a convolutional neural network. Before we see how this compares in accuracy to our densely connected neural network, we shall introduce two more layer types that can also improve performance—BatchNormalization and Dropout.

Batch normalization

One common problem when training deep neural network is ensuring that the weights of the network remain within a reasonable range of values—if they start to become too large, this is a sign that your network is suffering from what is known as the *exploding gradient* problem. As errors are propagated backwards through the network, the calculation of the gradient in the earlier layers can sometimes grow exponentially large, causing wild fluctuations in the weight values. If your loss function starts to return NaN, chances are that your weights have grown large enough to cause an overflow error and thus your network has *exploded*.

This doesn't have to happen immediately as you start training the network. Sometimes your network can be happily training for hours when suddenly, the loss function = NaN and it's exploded. This can be incredibly annoying—especially when the network has been seemingly been training well for a long time. To prevent this from happening we should first understand the root cause of the exploding gradient problem.

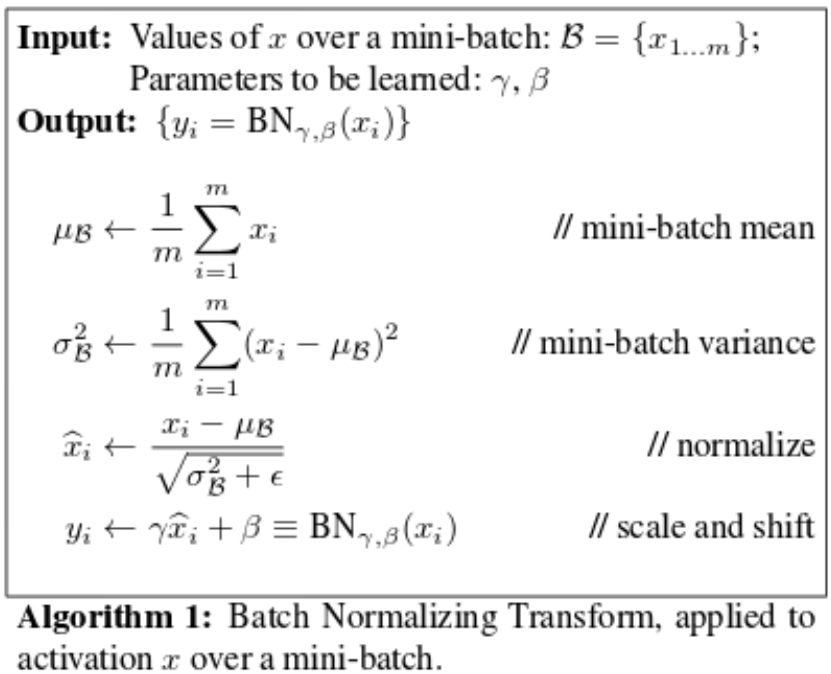
One of the reasons for scaling input data into a neural network is to ensure a stable start to training over the first few iterations. Since the weights of the network are initially randomized, unscaled input could potentially create huge activation values that immediately lead to exploding gradients. For example instead of passing pixel values from 0-255 into the nodes of the input layer, we usually scale these values to between 0 and 1.

Since the input is scaled, it's natural to expect the activations from all future layers to be relatively well scaled as well. Initially, this may well be true, but as the network trains and the weights move further away from their random initial values, this assumption can start to break down. This phenomena is known as *covariate shift*.

Imagine carrying a tall pile of books when you get hit by a gust of wind. You move the books in the direction opposite to the wind to compensate, but in doing so, some of the books shift so that the tower is slightly more unstable than before. Initially, this is OK, but with every gust, the pile becomes more and more unstable, until eventually the books have shifted so much that the pile collapses. This is covariate shift.

Relating this to neural networks, each layer is like a book in the pile. To remain stable, when the network updates the weights, each layer implicitly assumes that the distribution of its input from the layer beneath is approximately consistent across iterations. However, since there is nothing to stop any of the activation distributions shifting significantly in a certain direction, this can sometimes lead to runaway weight values and an overall collapse of the network.

Batch normalization [5] (2015, Ioffe, Szegedy) is a solution that drastically reduces this problem. The solution is surprisingly simple. A batch normalization layer calculates the mean and standard deviation of each of its input channels across the batch and normalizes by subtracting the mean and dividing by the standard deviation. There are then two learned parameters for each channel—the scale (gamma) and shift (beta). The output is simply the normalized input, scaled by gamma and shifted by beta. Figure 2-15 shows the whole process:



Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Figure 2-15. The batch normalization process [6]

We can place batch normalization layers after Dense or Convolutional layers to normalize the output from the layer. It's a bit like connecting the layers of books with small sets of adjustable springs that ensure there isn't any overall huge shifts in their position over time.

You might be wondering how this layer works at test time. When it comes to prediction, we

may only want to predict a single observation, so there is no *batch* over which to take averages. To get around this problem, during training a batch normalization layer also calculates the moving average of the mean and standard deviation of each channel and stores this value as part of the layer to use at test time.

How many parameters are contained within a batch normalization layer? For every channel in the preceding layer, 2 weights need to be learned—the scale (gamma) and shift (beta). These are the *trainable* parameters. The moving average and standard deviation also need to be calculated for each channel but since they are derived from the data passing through the layer, rather than trained through backpropagation, they are called *non-trainable* parameters. In total, this gives 4 parameters for each channel in the preceding layer, where 2 are trainable and 2 are non-trainable.

In Keras, the `BatchNormalization` layer implements the batch normalization functionality.

```
BatchNormalization(momentum = 0.9)
```

THe `momentum` parameter is the weight given to the previous value when calculating the moving average and moving standard deviation.

Dropout layer

When studying for an exam, it is common practice for students to use past papers and sample questions to improve their knowledge of the subject material. Some students try to memorise the answers to these questions, but then come unstuck in the exam because they haven't truly understood the subject matter. The best students use the practice material to further their general understanding, so that they are still able to answer correctly when faced with new questions that they haven't seen before.

Exactly the same principle is true for machine learning. Any successful machine learning algorithm must ensure that it generalizes to unseen data, rather than simply *remembering* the training dataset. If an algorithm performs well on the training dataset, but not the test dataset, we say that it is suffering from *overfitting*. To counteract this problem, we use *regularization* techniques, which ensure that the model is penalized if it starts to overfit.

There are many ways to regularize a machine learning algorithm, but for deep learning, one of the the most common is by using **Dropout** layers. This idea was introduced by Geoffrey Hinton in 2012 and presented in a 2014 paper by Srivastava et al., entitled 'Dropout: A Simple Way to Prevent Neural Networks from Overfitting'[7] .

Dropout layers are very simple. During training time, each dropout layer chooses a random set of units from the preceding layer and sets their output to zero, as shown in Figure 2-16. Incredibly, this simple addition drastically reduces overfitting, by ensuring that the network doesn't become overdependent on certain units or groups of units, that are just *remembering* the correct answer to observations in the training set. If we use dropout layers, the network cannot rely too much on any one unit and therefore knowledge is more evenly spread across the whole network. This makes the model much better at generalizing to unseen data, because the network has been trained to produce accurate predictions even under unfamiliar conditions, such as those caused by dropping random units. There are no weights to learn within a dropout layer, as the units to drop are decided stocastically. At test time, the dropout layer doesn't drop any units, so that the full network is used to make predictions.

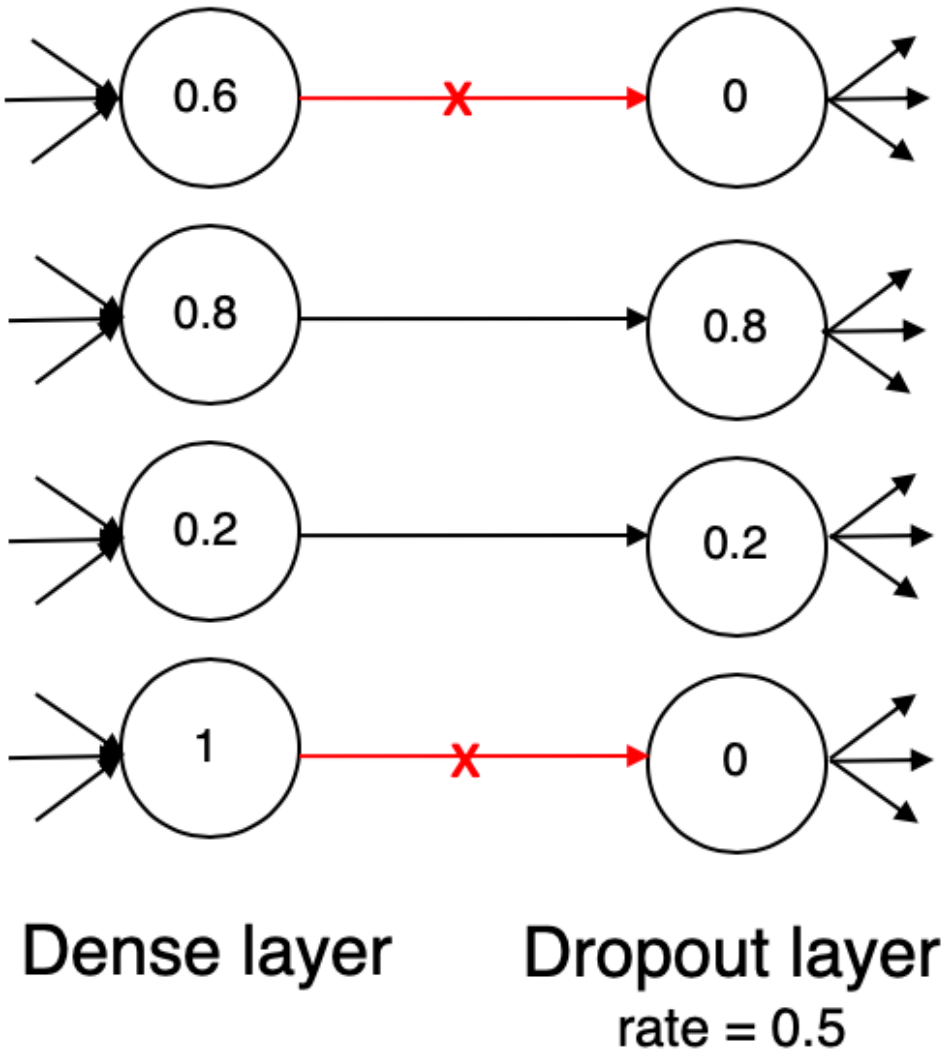


Figure 2-16. A dropout layer

Returning to our analogy, it's is a bit like a maths student practicing past papers with a random selection of key formulae missing from their formula book. This way, they learn how to answer

questions through an understanding of the core principles, rather than always looking up the formulae in the same place in the book. When it comes to test time, they will find it much easier to answer questions that they have never seen before, due to their ability to generalize beyond the training material.

The *Dropout* layer in Keras implements this functionality, with the *rate* parameter specifiying the proportion of units to drop from the preceding layer:

```
Dropout(rate = 0.25)
```

Dropout layers are used most commonly before Dense layers since these are most prone to overfitting due to the higher number of weights, though you can also use them before Convolutional layers as well. Batch normalization also has been shown to reduce overfitting and therefore many modern deep learning architectures are not using dropout at all, and relying solely on batch normalization for regularization. As with most deep learning principles, there is no golden rule that is true in every situation—the only way to know for sure is to test different architectures and see which performs best on a holdout set of data.

Putting it all together

We've now introduced three new Keras layer types—Conv2D, BatchNormalization and Dropout. Let's now put these pieces together into a new deep learning architecture and see how it performs on the CIFAR10 dataset.

You can run the following example in the jupyter notebook in the book repository called `02_03_deep_learning_conv_neural_network.ipynb`.

The model architecture we shall test is shown below:

```
input_layer = Input((32,32,3))

x = Conv2D(filters = 32, kernel_size = 3, strides = 1, padding = 'same')(i
x = BatchNormalization()(x)
x = LeakyReLU()(x)

x = Conv2D(filters = 32, kernel_size = 3, strides = 2, padding = 'same')(x
x = BatchNormalization()(x)
x = LeakyReLU()(x)

x = Conv2D(filters = 64, kernel_size = 3, strides = 1, padding = 'same')(x
x = BatchNormalization()(x)
x = LeakyReLU()(x)

x = Conv2D(filters = 64, kernel_size = 3, strides = 2, padding = 'same')(x
```

```
x = BatchNormalization()(x)
x = LeakyReLU()(x)

x = Flatten()(x)

x = Dense(128)(x)
x = BatchNormalization()(x)
x = LeakyReLU()(x)
x = Dropout(rate = 0.5)(x)

x = Dense(NUM_CLASSES)(x)
output_layer = Activation('softmax')(x)

model = Model(input_layer, output_layer)
```

We use four stacked Conv2D layers, each followed by a LeakyReLU and BatchNormalization layer. After flattening the resulting tensor, we pass the data through a Dense layer of size 128, again followed by a LeakyReLU and BatchNormalization layers. This is immediately followed by a Dropout layer for regularization and the network is concluded with an output Dense layer of size 10. The model summary is shown in Figure 2-17.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         (None, 32, 32, 3)         0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 32, 32, 32)        896
_________________________________________________________________
batch_normalization_1 (Batch (None, 32, 32, 32)        128
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 32, 32, 32)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 16, 16, 32)        9248
_________________________________________________________________
batch_normalization_2 (Batch (None, 16, 16, 32)        128
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 16, 16, 64)        18496
_________________________________________________________________
batch_normalization_3 (Batch (None, 16, 16, 64)        256
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 16, 16, 64)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 8, 8, 64)          36928
_________________________________________________________________
batch_normalization_4 (Batch (None, 8, 8, 64)          256
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 8, 8, 64)          0
_________________________________________________________________
flatten_2 (Flatten)          (None, 4096)              0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               524416
_________________________________________________________________
batch_normalization_5 (Batch (None, 128)               512
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 128)               0
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                1290
_________________________________________________________________
activation_1 (Activation)    (None, 10)                0
=================================================================
Total params: 592,554
Trainable params: 591,914
Non-trainable params: 640
_________________________________________________________________
```

Figure 2-17. Convolutional Neural Network for CIFAR10

Before moving on, make sure you are able to calculate the output shape and number of parameters for each layer by hand. It's a good exercise to prove to yourself that you have fully understood how each layer is constructed and how it is connected to the preceding layer! Don't forget to include the bias weights that are included as part of the Conv2D and Dense layers!

We compile and train the model in exactly the same way as before and call the *evaluate* method to determine its accuracy on the holdout set.

```
model.evaluate(x_test, y_test, batch_size=1000)
```

```
 10000/10000 [==============================] - 15s 1ms/step
[0.8423407137393951, 0.7155999958515167]
```

Figure 2-18. CNN

As you can see, this model is now achieving 71.5% accuracy—up from 49.0% previously. Much better! Figure 2-19 shows some predictions from our new convolutional model.

Figure 2-19. CNN

This improvement has been achieved simply by changing the architecture of the model to include convolutional, batch normalization and dropout layers. Notice that the number of parameters is actually fewer in our new model than the previous model, even though the number of layers is far greater. This demonstrates the importance of being experimental with your model design and being comfortable with how the different layer types can be used to your advantage. When building generative models, it becomes even more important to understand the inner workings of your model since it is the middle layers of your network that capture the high-level features that we are most interested in.

Summary

In this chapter we have introduced the core deep learning concepts that we will need to start building out first deep generative models.

A really important point to take away from this chapter is that deep neural networks are completely flexible by design, and that there really are no fixed rules when it comes to model architecture. There are guidelines and best practice, but you should feel free to experiment with layers and the order in which they appear. You will need to bear in mind that, like a set of building blocks, some layers will not fit together, simply because the input shape of one does not confirm to the output shape of the other, but this knowledge comes with experience and a solid understanding of how each layer changes the tensor shape as data flows through the network.

Another point to remember is that it is the layers in a deep neural networks that are

convolutional, rather than the network itself. When people talk about "convolutional neural networks", they really mean "neural networks that contains a convolutional layer". It is important to make this distinction, because you shouldn't feel constrained to only use the architectures that you have read about in this book or elsewhere, but instead should see them as just one way of piecing together the different layer types. Like a child with a set of building blocks, the design of your neural network is only limited by your own imagination and crucially, your understanding of how the various layers fit together.

In the next chapter, we shall see how we can use these building blocks to design a network that can generate images.

---

1 Image source: *https://www.cs.toronto.edu/~kriz/cifar.html*

2 *https://arxiv.org/abs/1412.6980v8*

3 *https://arxiv.org/abs/1711.00489*

4 source: *https://arxiv.org/pdf/1603.07285.pdf*

5 *https://arxiv.org/abs/1502.03167*

6 Image sourced from the original paper: *https://arxiv.org/abs/1502.03167*

7 *http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf*

Chapter 3. Variational Autoencoders

> **NOTE**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as he or she writes—so you can take advantage of these technologies long before the official release of these titles.

In 2013, D.P Kingma and Max Welling published a paper entitled `Auto-encoding Variational Bayes'[1] , that laid the foundations for a type of neural network known as a variational autoencoder (VAE). This technique is now one of the most fundamental and well known deep learning architectures for generative modeling. In this chapter, we shall start by building a standard autoencoder and see how we can extend this framework to develop a variational autoencoder—our first example of a generative deep learning model.

Along the way, we will pick apart both types of model, to understand how they work at a granular level. By the end of the chapter you should have a complete understanding of how to build and manipulate autoencoder based models and in particular, how to build a variational autoencoder from scratch to generate images based on your own training set.

Let's start by paying a visit to a strange art exhibition…

The Art Exhibition

Two brothers, Mr N. Coder and Mr D. Coder run an art gallery. One weekend, they host an exhibition focused on monochrome studies of single digit numbers.

The exhibition is particularly strange because it contains only one wall and no physical artwork. When a new painting arrives for display, Mr N. Coder simply chooses a point on the wall to represent the painting, places a marker at this point, then throws the original artwork away. When a customer requests to see the painting, Mr D. Coder attempts to recreate the artwork

using just the coordinates of the relevant marker on the wall.

The exhibition wall is shown in Figure 3-1, where each black dot is a marker placed by Mr N. Coder to represent a painting. We also show one of the paintings being marked on the wall at the point [-3.5, -0.5] by Mr N.Coder and then reconstructed using just these two numbers by Mr D.Coder. In Figure 3-2 you can see examples of other original paintings (top row), the coordinates of the point on the wall given by Mr N. Coder and the reconstructed painting produced by Mr D.Coder (bottom row).

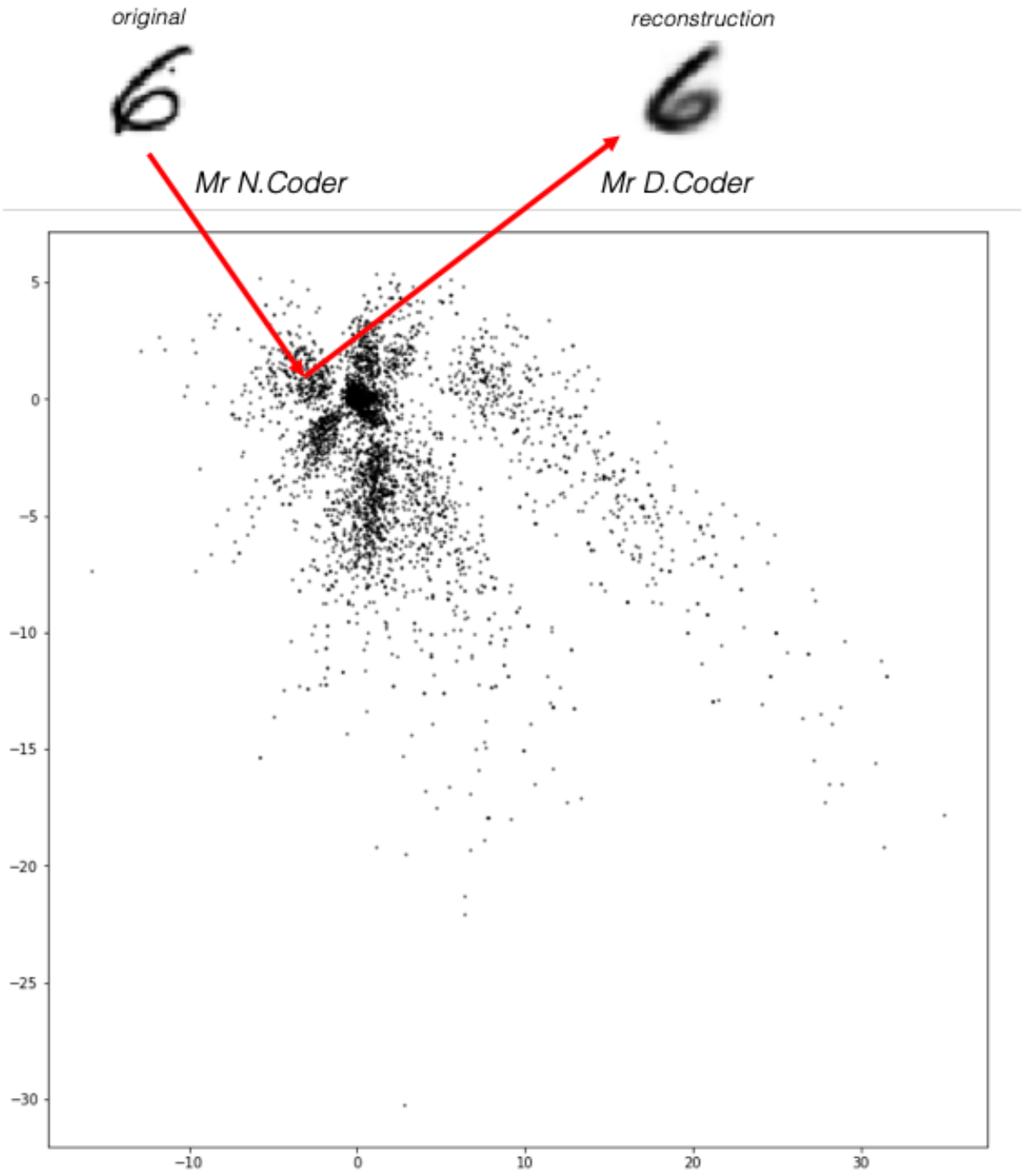


Figure 3-1. The wall at the Art Exhibition.

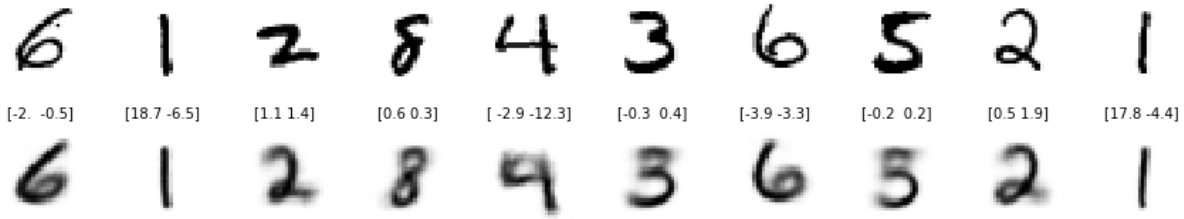


Figure 3-2. More examples of reconstructed paintings

So how does Mr N. Coder decide where to place the markers? The brothers carefully monitor the loss of revenue at the ticket office caused by customers asking for money back for badly reconstructed paintings and are constantly trying to find a system that minimizes this loss of earnings. By gradually tweaking the processes for marker placement and reconstruction, the system arises naturally through years of training and working together. As you can see from Figure 3-2, it works remarkably well—customers who come to view the artwork very rarely complain that Mr D. Coder's recreated paintings are significantly different from the original pieces they came to see.

One day, Mr N.Coder has an idea. What if he placed markers on empty parts of the wall that currently do not have a marker? Mr D. Coder could then recreate the artwork corresponding to these points and within a few days they would have their own exhibition of completely original, generated paintings.

The brothers set about their plan and open their new exhibition to the public. Some of the exhibits and corresponding markers are displayed in Figure 3-3

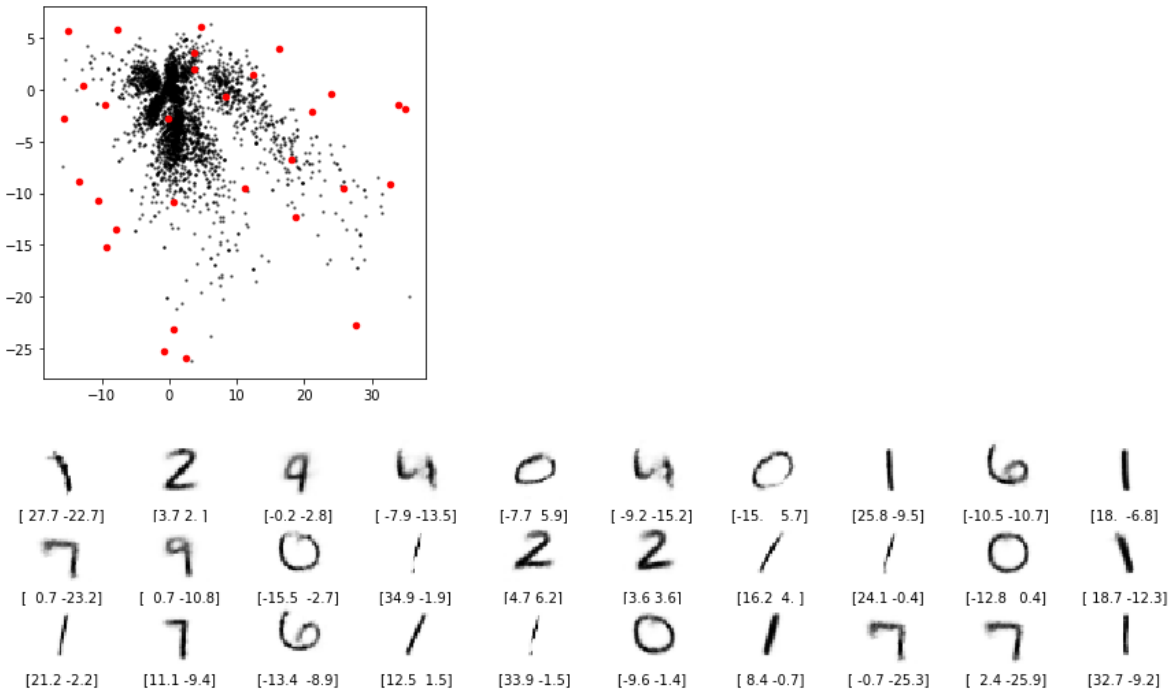


Figure 3-3. The new generative art exhibition

As you can see, the plan was not a great success. The overall variety is poor and some pieces of the artwork do not really resemble single digit numbers.

So, what went wrong and how can the brothers improve their scheme?

Autoencoders

The above story is an analogy for an autoencoder.

An *autoencoder* is a neural network made up of two parts:

- An *encoder* network that compresses high dimensional input data into a lower dimensional representation vector
- A *decoder* network that decompresses a given representation vector back to the original domain.

This process is shown in Figure 3-4.

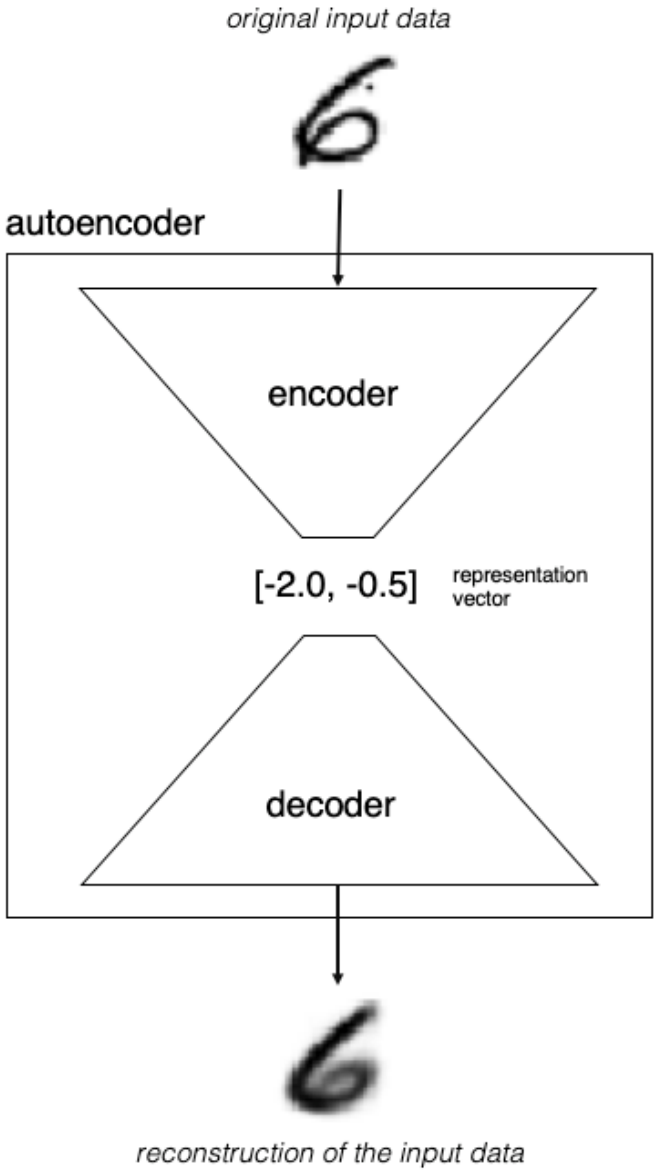


Figure 3-4. Diagram of an autoencoder

The network is trained to find weights for the encoder and decoder that minimize the loss between the original input and the reconstruction of the input after it has passed through the encoder and decoder.

The representation vector is a compression of the original image into a lower dimensional, latent space. The idea is that by choosing *any* point in the latent space, we should be able to generate novel images by passing this point through the decoder, since the decoder has learned how to convert points in the latent space into viable images.

In our analogy, Mr N.Coder and Mr D.Coder are using representation vectors inside a two dimensional latent space (the wall) to encode each image. This helps us to visualize the latent space, since we can easily plot points in 2D. In practice, autoencoders usually have more than two dimensions in order to have more freedom to capture greater nuance in the images.

Autoencoders can also be used to clean noisy images, since the encoder learns that it is not useful to capture the position of the random noise inside the latent space. For tasks such as this, a 2D latent space is probably too small to encode sufficient relevant information from the input. However, as we shall see, increasing the dimensionality of the latent space quickly leads to problems if we want to use the autoencoder as a generative model.

Your first autoencoder

Let's now build an autoencoder in Keras. We shall follow the Jupyter notebook `03_01_autoencoder_train.ipynb` in the book repository.

Generally speaking, it is a good idea to create a class for your model in a seperate file. This way, you can instantiate an Autoencoder object with parameters that define a particular model architecture in the notebook, as shown below. This makes you model very flexible and able to be easily tested and ported to other projects as necessary.

Example 3-1.

```
from models.AE import Autoencoder

AE = Autoencoder(
    input_dim = (28,28,1)
    , encoder_conv_filters = [32,64,64, 64]
    , encoder_conv_kernel_size = [3,3,3,3]
    , encoder_conv_strides = [1,2,2,1]
    , decoder_conv_t_filters = [64,64,32,1]
    , decoder_conv_t_kernel_size = [3,3,3,3]
    , decoder_conv_t_strides = [1,2,2,1]
    , z_dim = 2)
```

Let's now take a look at the architecture of an autoencoder in more detail.

The encoder

An autoencoder consists of two parts—the encoder and the decoder. The encoder's job is to take the input image and map it to a point in the latent space. The architecture of the encoder we will be building is shown in Figure 3-5.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
encoder_input (InputLayer)   (None, 28, 28, 1)         0
_________________________________________________________________
encoder_conv_0 (Conv2D)      (None, 28, 28, 32)        320
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 28, 28, 32)        0
_________________________________________________________________
encoder_conv_1 (Conv2D)      (None, 14, 14, 64)        18496
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 14, 14, 64)        0
_________________________________________________________________
encoder_conv_2 (Conv2D)      (None, 7, 7, 64)          36928
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 7, 7, 64)          0
_________________________________________________________________
encoder_conv_3 (Conv2D)      (None, 7, 7, 64)          36928
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 7, 7, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 3136)              0
_________________________________________________________________
encoder_output (Dense)       (None, 2)                 6274
=================================================================
Total params: 98,946
Trainable params: 98,946
Non-trainable params: 0
_________________________________________________________________
```

Figure 3-5. Architecture of the encoder

To achieve this, we first create an input layer for the image and pass this through four convolutional layers in sequence, each of which captures increasingly high-level features. We use a stride of 2 on some of the layers to reduce the size of the output. The last convolutional layer is then flattened, and connected to a Dense layer of size 2, which represents our 2 dimensional latent space.

The following code shows how to build this in Keras:

Example 3-2.

```
### THE ENCODER
encoder_input = Input(shape=self.input_dim, name='encoder_input') ❶

x = encoder_input

for i in range(self.n_layers_encoder):
    conv_layer = Conv2D(
        filters = self.encoder_conv_filters[i]
        , kernel_size = self.encoder_conv_kernel_size[i]
        , strides = self.encoder_conv_strides[i]
        , padding = 'same'
```

```
            , name = 'encoder_conv_' + str(i)
            )

        x = conv_layer(x) ❷
        x = LeakyReLU()(x)

    shape_before_flattening = K.int_shape(x)[1:]
    x = Flatten()(x) ❸

    encoder_output= Dense(self.z_dim, name='encoder_output')(x) ❹

    self.encoder = Model(encoder_input, encoder_output) ❺
```

❶ Define the input to the encoder (the image)

❷ Stack convolutional layers sequentially on top of each other

❸ Flatten the last convolutional layer to a vector

❹ Dense layer that connects this vector to the 2D latent space

❺ The Keras model that defines the encoder—a model that take an input image and encodes it into the 2D latent space

You can change the number of convolutional layers in the encoder, simply by adding elements to the lists that define the model architecture in the notebook. I strongly recommend experimenting with the parameters that define the models in this book, to understand how the architecture affects the number of weights in each layer, model performance and model run time.

The decoder

The decoder is a mirror image of the encoder, except instead of convolutional layers, we use *convolutional transpose* layers, as shown in Figure 3-6

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
decoder_input (InputLayer)   (None, 2)                 0
_________________________________________________________________
dense_1 (Dense)              (None, 3136)              9408
_________________________________________________________________
reshape_1 (Reshape)          (None, 7, 7, 64)          0
_________________________________________________________________
decoder_conv_t_0 (Conv2DTran (None, 7, 7, 64)          36928
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 7, 7, 64)          0
_________________________________________________________________
decoder_conv_t_1 (Conv2DTran (None, 14, 14, 64)        36928
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 14, 14, 64)        0
_________________________________________________________________
decoder_conv_t_2 (Conv2DTran (None, 28, 28, 32)        18464
_________________________________________________________________
leaky_re_lu_7 (LeakyReLU)    (None, 28, 28, 32)        0
_________________________________________________________________
decoder_conv_t_3 (Conv2DTran (None, 28, 28, 1)         289
_________________________________________________________________
activation_1 (Activation)    (None, 28, 28, 1)         0
=================================================================
Total params: 102,017
Trainable params: 102,017
Non-trainable params: 0
_________________________________________________________________
```

Figure 3-6. Architecture of the decoder

CONVOLUTIONAL TRANSPOSE LAYERS

Standard convolutional layers allow us to half the size of an input tensor in both height and width, by setting `strides = 2`.

The convolutional transpose layer uses the same principal as a standard convolutional layer (passing a filter across the image), but is different in that setting `strides = 2` *doubles* the size of the input tensor in both height and width.

In a convolutional transpose layer, the *strides* parameter determines the internal zero padding between pixels in the image as shown in Figure 3-7.

In Keras, the **Conv2DTranspose** layer allows us to perform convolutional transpose operations on tensors.

By stacking these layers, we can gradually expand the size of each layer, using strides of 2, until we get back to the original image dimension of 28 x 28.

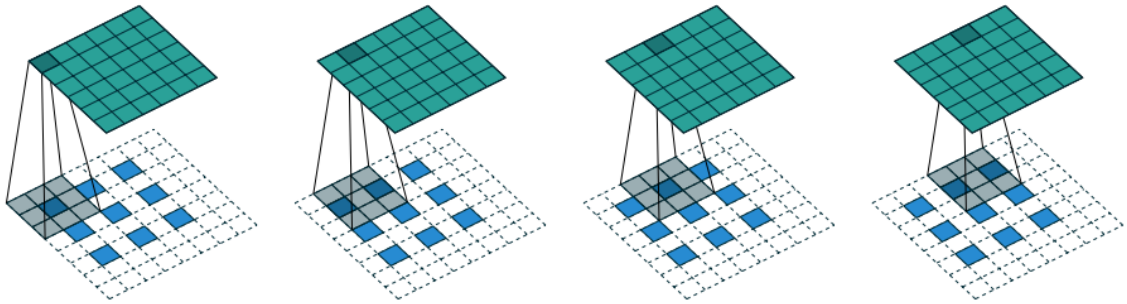


Figure 3-7. A convolutional transpose layer example—here, a 3x3x1 filter (grey) is being passed across a 3x3x1 image (blue) with `strides = 2`, to produce a 6x6x1 output tensor (green).[2]

Note that the decoder doesn't have to be a mirror image of the encoder. It can be anything you want, as long as the output from the last layer of the decoder is the same size as the input to the encoder (since our loss function will be comparing these pixelwise).

The following code shows how we build the decoder in Keras:

Example 3-3.

```
### THE DECODER
decoder_input = Input(shape=(self.z_dim,), name='decoder_input') ❶

x = Dense(np.prod(shape_before_flattening))(decoder_input) ❷
x = Reshape(shape_before_flattening)(x) ❸

for i in range(self.n_layers_decoder):
    conv_t_layer = Conv2DTranspose(
        filters = self.decoder_conv_t_filters[i]
        , kernel_size = self.decoder_conv_t_kernel_size[i]
        , strides = self.decoder_conv_t_strides[i]
        , padding = 'same'
        , name = 'decoder_conv_t_' + str(i)
        )

    x = conv_t_layer(x) ❹

    if i < self.n_layers_decoder - 1:
        x = LeakyReLU()(x)
    else:
        x = Activation('sigmoid')(x)

decoder_output = x

self.decoder = Model(decoder_input, decoder_output) ❺
```

❶ Define the input to the decoder (the point in the latent space)

❷ Connect the input to a Dense layer

❸ *Unflatten* this vector into a tensor that can be fed as input into the first convolutional transpose layer

❹ Stack convolutional transpose layers on top of each other

❺ The Keras model that defines the decoder—a model that takes a point in the latent space and decodes it into the original image domain.

Joining the encoder to the decoder

To train the encoder and decoder simultaneously, we need to define a model that will represent the flow of an image through the encoder and back out through the decoder. Luckily, Keras makes it extremely easy to do this:

Example 3-4.

```
### THE FULL AUTOENCODER
model_input = encoder_input # ❶
model_output = decoder(encoder_output) # ❷

self.model = Model(model_input, model_output) # ❸
```

❶ The input to the autoencoder is simply the same as the input to the encoder

❷ The output from the autoencoder is the output from the encoder passed through the decoder

❸ The Keras model that defines the full autoencoder—a model that takes an image, passes it through the encoder and back out through the decoder, to generate a reconstruction of the original image.

Now that we've defined our model, we just need to compile it with a loss function and optimizer.

The loss function is usually chosen to be either the root mean squared error (RMSE) or binary cross entropy between the individual pixels of the original image and the reconstruction. Binary cross entropy places heavier penalties on predictions at the extremes that are badly wrong, so tends to push pixel predictions to the middle of the range. This results in less vibrant images. For this reason, I generally prefer to use RMSE as the loss function. However, there is no right

or wrong choice— you should choose whichever works best for your use case.

Example 3-5.

```
### COMPILATION
optimizer = Adam(lr=learning_rate)

def r_loss(y_true, y_pred):
    return K.mean(K.square(y_true - y_pred), axis = [1,2,3])

self.model.compile(optimizer=optimizer, loss = r_loss)
```

We train the autoencoder by passing in the input images as both the input and output.

Example 3-6.

```
self.model.fit(
    x = x_train
    , y = x_train
    , batch_size = batch_size
    , shuffle = True
    , epochs = 10
    , callbacks = callbacks_list
)
```

Analysis of the autoencoder

Now that our autoencoder is trained, we can start to investigate how it is representing images in the latent space. We'll then see how variational autoencoders are the natural extension that fix the issues faced by autoencoders. The relevant code is included in the `03_02_autoencoder_analysis.ipynb` notebook in the book repository

Firstly let's take a set of new images that the model hasn't seen, pass them through the encoder and plot the 2D representations in a scatter plot. In fact, we've already seen this plot—it's just Mr N.Coder's wall from Figure 3-1. Coloring this plot by digit produces the chart in Figure 3-8. It's worth noting that even though the digit labels were never shown to the model during training, the autoencoder has naturally grouped digits that look alike into the same part of the latent space.

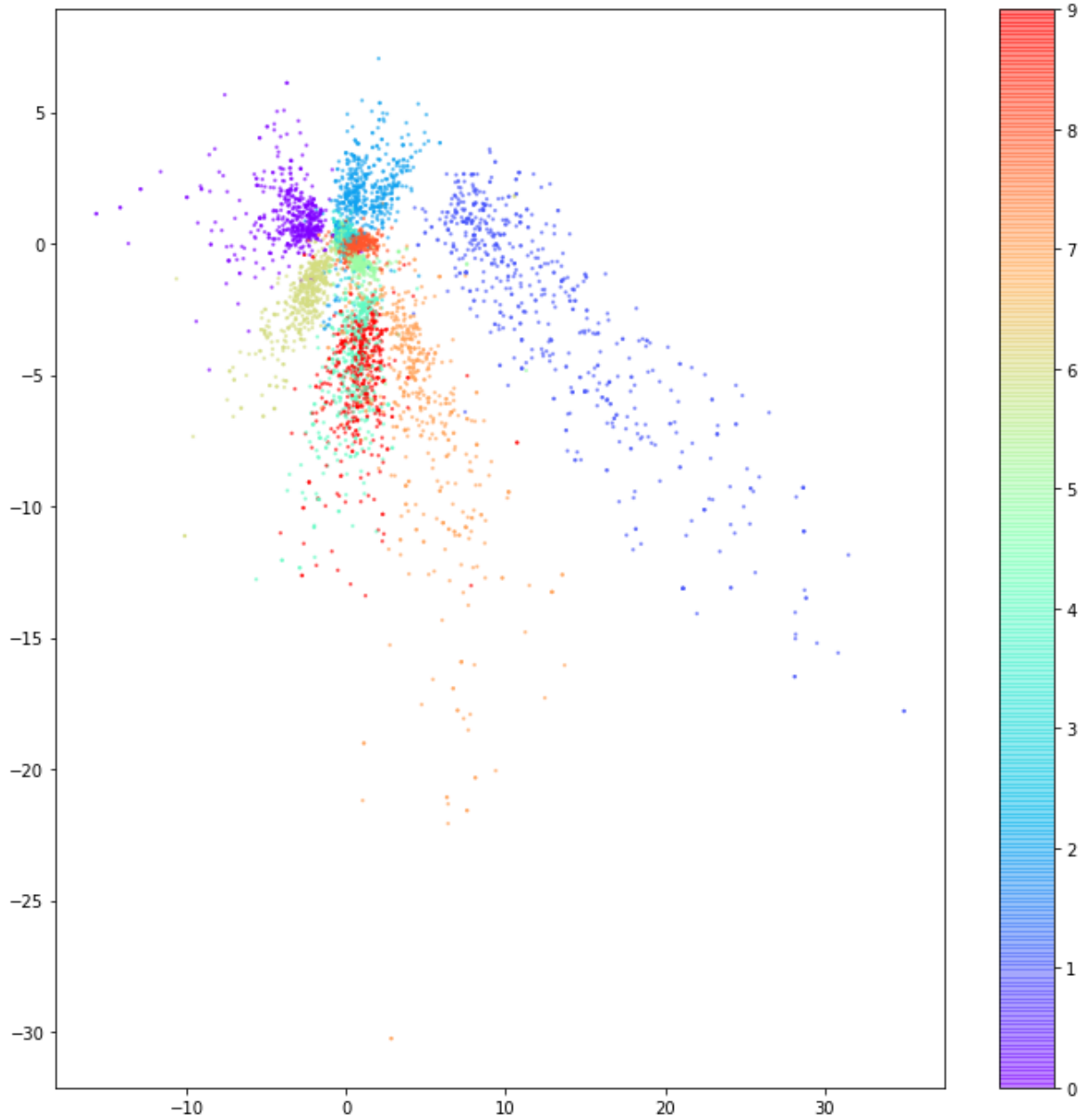


Figure 3-8. Plot of the latent space, coloured by digit

There are a few interesting points to note:

1. The plot is not symmetrical about the point (0, 0) or bounded—for example there are far more points with negative y-axis values than positive and some points even extend to a y-axis value of < -30.

2. Some digits are represented over a very small area and others over a much larger area.

3. There are large gaps between colors containing few points

Remember, our goal is to be able to choose a random point in the latent space, pass this through the decoder and obtain an image of a digit that looks real. If we do this multiple times we would also ideally like to get a roughly equal mixture of different kinds of digit (i.e. it shouldn't always

product the same digit). This was also the aim of the Coder brothers when they were choosing random points on their wall to generate new artwork for their exhibition.

Point 1 explains why its not obvious how we should even go about choosing a *random* point in the latent space, since the distribution of these points is undefined. Technically, we would be justified in choosing any point in the 2D plane! It's not even guaranteed that points will be centered around (0,0). This makes sampling from our latent space extremely problematic.

Point 2 explains the lack of diversity in the generated images. Ideally, we'd like to obtain a roughly equal spread of digits when sampling randomly from our latent space—however, with an autoencoder this is not guaranteed. For example, the area of 1's is far bigger than the area for 8's, so when we pick points randomly in the space, we're more likely to sample something that decodes to look like a *1* than an *8*.

Point 3 explains why some generated images are poorly formed. In Figure 3-9 we can see three points in the latent space and their decoded images, all of which are not particularly well-formed. Partly, this is because of the large spaces at the edge of the domain where there are few points—the autoencoder has no reason to ensure that points here are decoded to legible digits as very few images are encoded here. However, more worryingly, even points that are right in the middle of the domain may not be decoded into well-formed images. This is because the autoencoder is not forced to ensure that the space is *continuous*—for example, even though the point (2, -2) might be decoded to give a satisfactory image of a *4*, there is no mechanism in place to ensure that the point (2.1, -2.1) also produces a satisfactory *4*.

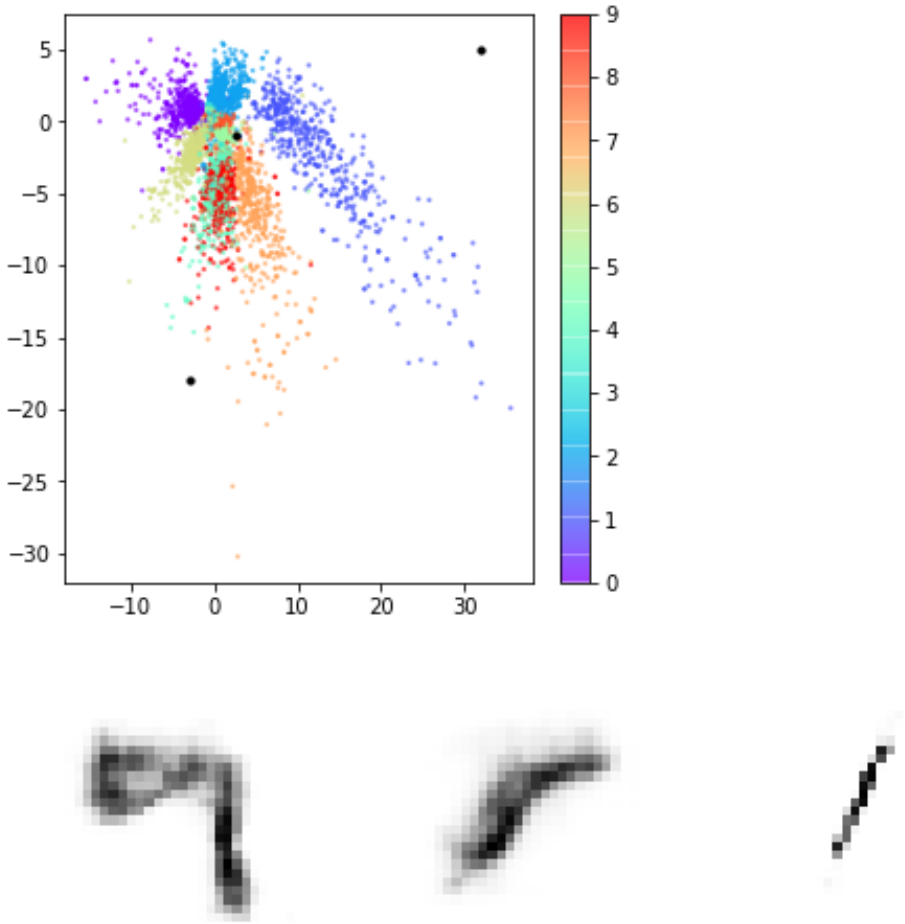


Figure 3-9. Some poorly generated images

In 2D this issue is subtle, because the autoencoder only has a small number of dimensions to work with, so naturally has to squash digit groups together, resulting in the space between digit groups being relatively small. However as we start to use more dimensions in the latent space to generate more complex images such as faces, this problem becomes even more apparent. If we give the autoencoder a free reign over how it uses the latent space to encode images, there will be huge gaps between groups of similar points with no incentive for the space between to generate well-formed images.

So how can we solve these three problems, so that our autoencoder framework is ready to be used as a generative model? To explain, let's revisit the Coder brothers' art exhibition, where a few changes have taken place since our last visit…

The Variational Art Exhibition

Determined to make the generative art exhibition work, Mr N.Coder recruits the help of his daughter, Epsilon. After a brief discussion, they decide to change the way that new paintings are marked on the wall. The new process works as follows:

When a new painting arrives at the exhibition, Mr N. Coder chooses a point on the wall where

he would like to place the marker to represent the artwork, as before. However now, instead of placing the marker on the wall himself, he passes his opinion of where it should go to Epsilon who decides where the marker shall be placed. She of course takes her father's opinion into account, so usually places the marker somewhere near the point that he suggests. Mr D.Coder then finds the marker where Epsilon placed it and never hears Mr N.Coder's original opinion.

Mr N. Coder also provides his daughter with an indication of how sure he is that the marker should be placed at the given point. The more certain he is, the closer Epsilon will generally place the point to his suggestion.

There is one final change to the old system. Before, the only feedback mechanism was the loss of earnings at the ticket office for poorly reconstructed images. If brothers saw that particular paintings weren't being recreated accurately, they would adjust their understanding of marker placement and image regeneration, to ensure revenue loss was minimized.

Now, there is another source of feedback. Epsilon is quite lazy and gets annoyed whenever her father tells her to place markers far away from the center of the wall, where the ladder rests. She also doesn't like it when he is too strict on where the markers should be placed, as then she feels she doesn't have enough responsibility. Equally, if her father provides little confidence in where the markers should go, she feels like she's the one doing all the work! The amount of confidence in marker placement that he provides has to be just right for her to be happy.

To compensate her annoyance, her father pays her more to do the job whenever he doesn't stick to these rules. On the balance sheet, this expense is listed as his kitty-loss (KL) divulgance. He therefore needs to be careful that he doesn't end up paying her too much, as well as also monitoring the loss of revenue at the ticket office.

With these simple changes, Mr N. Coder once again tries his strategy of placing markers on portions of the wall that are empty, so that Mr D. Coder can regenerate these points as original artwork.

Some of these points are shown in Figure 3-10, along with the images that were generated.

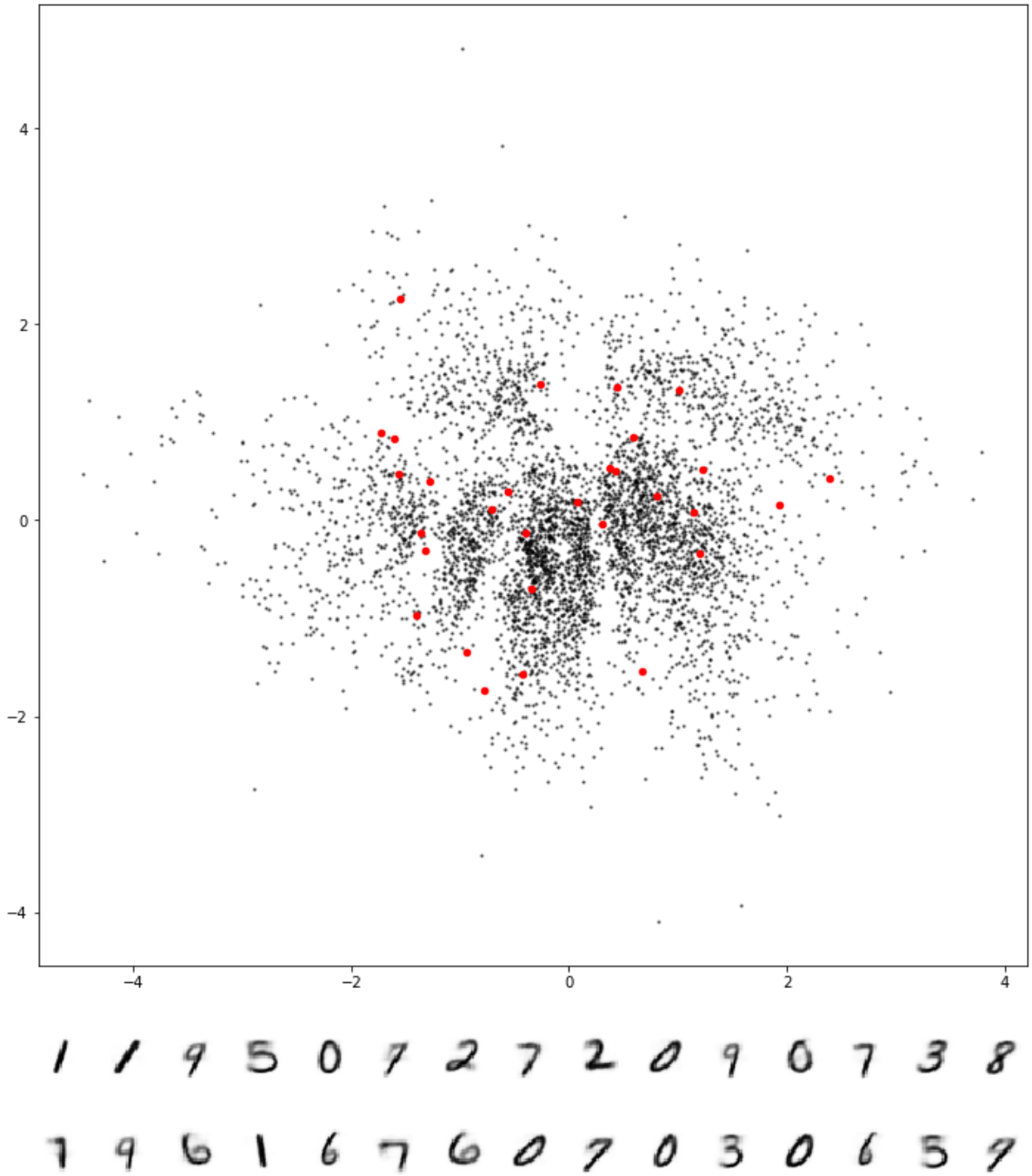


Figure 3-10. Artwork from the new Exhibition

Much better! The crowds arrive in great waves to see this new, exciting generative art and are amazed by the originality and diversity of the paintings. The Coder brothers retire rich and famous and Epsilon returns to her previous job as a Free Space Permittivity officer.

Variational Autoencoders

The previous story explains how with a few simple changes, the art exhibition could be transformed into a successful generative process. Let's now try to understand mathematically what we need to do to our autoencoder to convert it into a variational autoencoder and thus make it a truly generative model.

There are actually only two parts that we need to change—the encoder and the loss function.

The encoder

In an autoencoder, each image is mapped directly to one point in the latent space. In a variational autoencoder, each image is instead mapped to a multivariate normal distribution around a point in the latent space, as shown in Figure 3-11

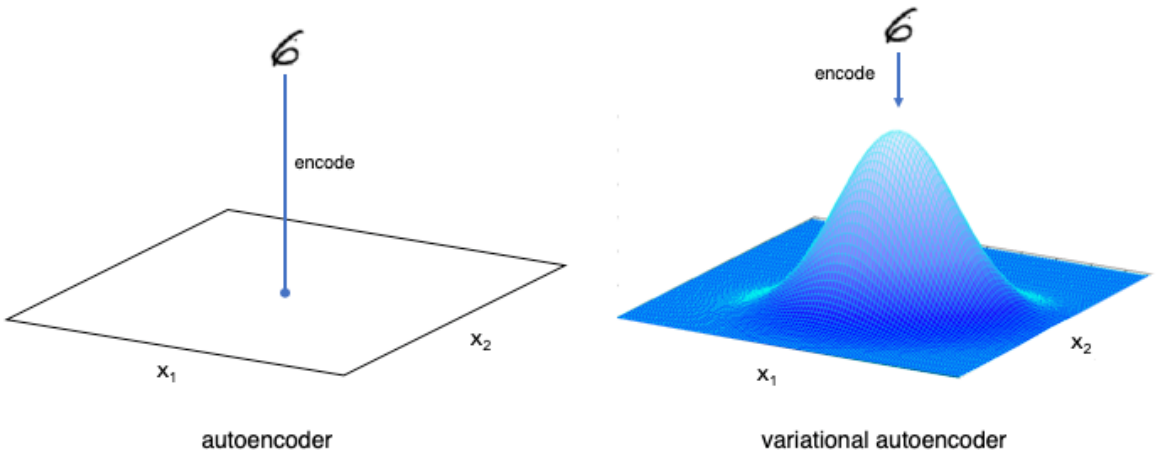


Figure 3-11. Encoders—the difference between an autoencoder and a variational autoencoder

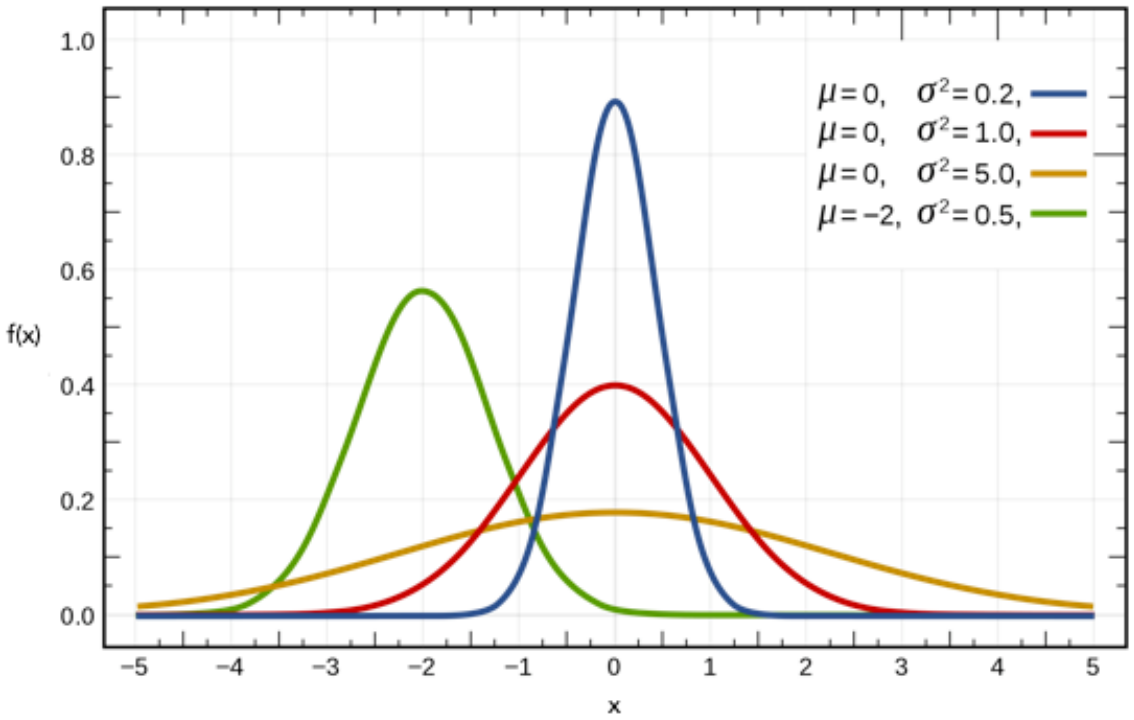


Figure 3-12. The normal distribution in one dimension[3]

THE NORMAL DISTRIBUTION

A normal distribution is a probability distribution characterized by a distinctive *bell-curve* shape. In one dimension, it is defined by two variables—the *mean* (μ, pronounced “mu”) and the *variance* (σ^2 pronounced “sigma squared”). The *standard deviation* (σ) is the positive square root of the variance.

The probability density function of the normal distribution in one dimension is given below:

$$f\left(x \mid \mu, \sigma^2\right) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Figure 3-12 shows several normal distributions in one-dimension, for different values of the mean and variance. The red curve is the *standard normal*—the normal distribution with mean equal to 0 and variance equal to 1.

We can sample a point z from a normal distribution with mean μ and variance σ using the following equation:

$$z = \mu + \sigma\epsilon$$

where ϵ is a point sampled from a standard normal distribution.

The concept of a normal distribution extends to more than one dimension—the probability density function for a general multivariate normal distribution in k dimensions is given below:

$$f\left(x_1, \ldots, x_k\right) = \frac{\exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^{\mathrm{T}} \Sigma^{-1}(\mathbf{x}-\mu)\right)}{\sqrt{(2\pi)^k |\Sigma|}}$$

In 2D, the mean vector μ and the symmetric covariance matrix Σ are defined as follows:

$$\mu = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}, \quad \Sigma = \begin{pmatrix} \sigma_1^2 & \rho\sigma_1\sigma_2 \\ \rho\sigma_1\sigma_2 & \sigma_2^2 \end{pmatrix}$$

where ρ is the correlation between the two dimensions x_1 and x_2.

Variational autoencoders assume that there is no correlation between any of the dimensions in the latent space and therefore that the covariance matrix is diagonal. This means the encoder only needs to map each input to a mean vector and a variance vector and not worry about covariance between dimensions. We also choose to map to the *logarithm* of the variance, rather than the variance directly, as log-variance can take any real number in the range $(-\infty, \infty)$, matching the natural output range from a neural network unit, whereas variance values are always positive.

To summarise, the encoder will take each input image and encode it to two vectors, that together define a multivariate normal distribution on the latent space:

- `mu`—the mean point of the distribution
- `log_var`—the logarithm of the variance of each dimension

To encode an image into a specific point `z` in the latent space, we can sample from this distribution, using the following equation[4] :

```
z = mu + exp(log_var / 2) * epsilon
```

where `epsilon` is a point sampled from the standard normal distribution.

Relating this back to our story, `mu` represents Mr N. Coder's opinion of where the marker should appear on the wall. `epsilon` is his daughter's random choice of how far away from `mu` the marker should be placed, scaled by `log_var`, Mr N.Coder's confidence in the marker's position.

So why does this small change to the encoder help?

Previously, we saw how there was no requirement for the latent space to be continuous—even if the point (-2,2) decodes to a well formed image of a 4, there was no requirement for (-2.1,2.1) to look similar. Now, since we are sampling at random from an area around `mu`, the decoder must ensure that all points in the same neighborhood produce very similar images when decoded, so that the reconstruction loss remains small. This is a very nice property that ensures that even when we choose a point in the latent space that has never been seen by the decoder, it is likely to decode to an image that is well-formed.

Let's now see how we build this new version of the encoder in Keras. You can train your own variational autoencoder on the digits dataset by running the notebook `03_03_vae_digits_train.ipynb` in the book repository.

Example 3-7.

```
### THE ENCODER
encoder_input = Input(shape=self.input_dim, name='encoder_input')

x = encoder_input

for i in range(self.n_layers_encoder):
    conv_layer = Conv2D(
        filters = self.encoder_conv_filters[i]
```

```
            , kernel_size = self.encoder_conv_kernel_size[i]
            , strides = self.encoder_conv_strides[i]
            , padding = 'same'
            , name = 'encoder_conv_' + str(i)
            )

        x = conv_layer(x)

        if self.use_batch_norm:
            x = BatchNormalization()(x)

        x = LeakyReLU()(x)
        if self.use_dropout:
            x = Dropout(rate = 0.25)(x)

    shape_before_flattening = K.int_shape(x)[1:]
    x = Flatten()(x)

    self.mu = Dense(self.z_dim, name='mu')(x) ❶
    self.log_var = Dense(self.z_dim, name='log_var')(x) #

    encoder_mu_log_var = Model(encoder_input, (self.mu, self.log_var)) ❷

    def sampling(args):
        mu, log_var = args
        epsilon = K.random_normal(shape=K.shape(mu), mean=0., stddev=1.)
        return mu + K.exp(log_var / 2) * epsilon

    encoder_output = Lambda(sampling, name='encoder_output')([self.mu, sel
    f.log_var]) ❸

    encoder = Model(encoder_input, encoder_output) ❹
```

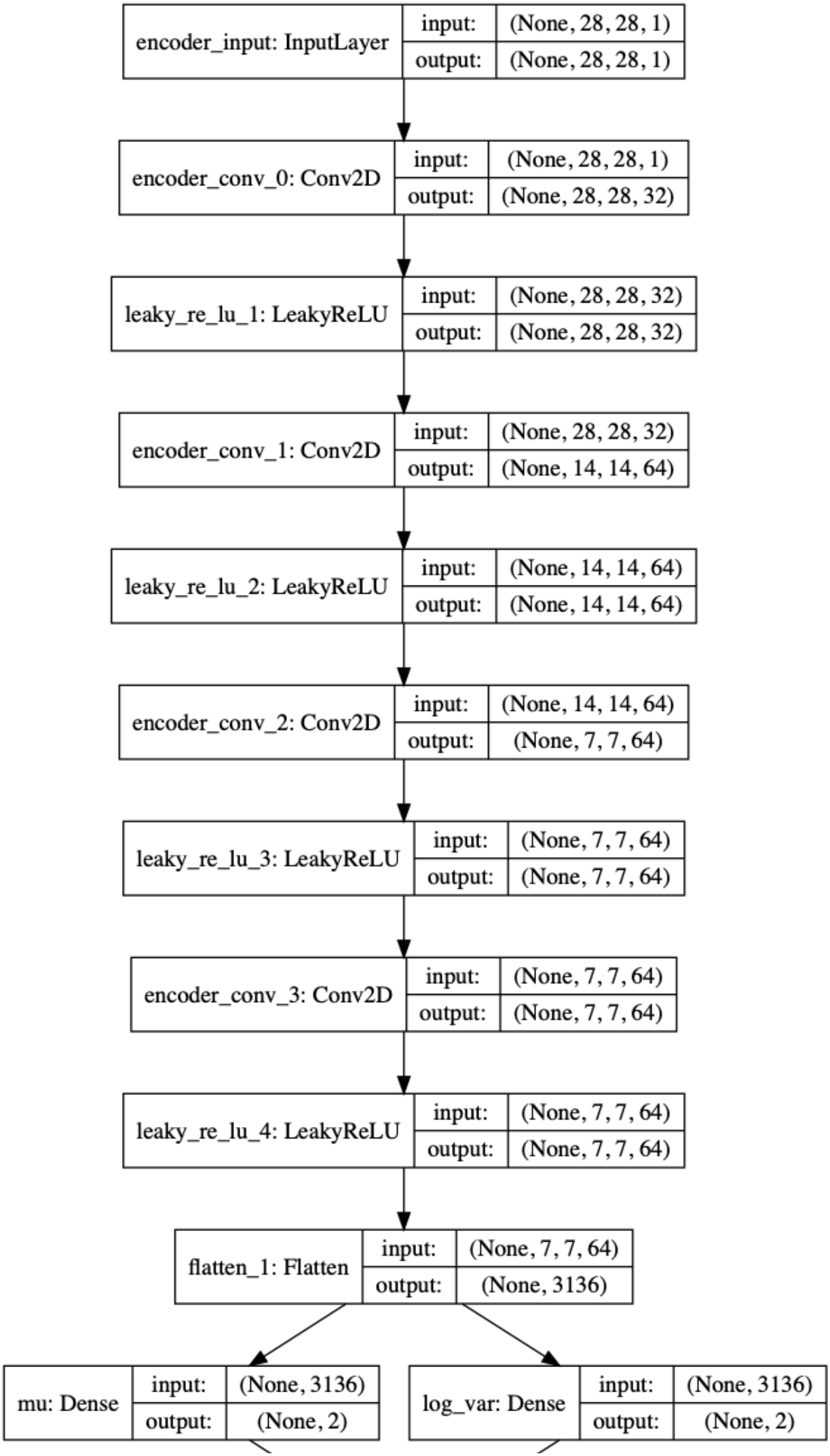
❶ Instead of connecting the flattened layer directly to the 2D latent space, we instead connect it to two layers, `mu` and `log_var`

❷ The Keras model that outputs the values of `mu` and `log_var` for a given input image

❸ This Lambda layer samples a point `z` in the latent space from the normal distribution defined by the parameters `mu` and `log_var`.

❹ The Keras model that defines the encoder—a model that take an input image and encodes it into the 2D latent space, by sampling a point from the normal distribution defined by `mu` and `log_var`.

A diagram of the encoder is shown in Figure 3-13.


encoder_input: InputLayer
input: (None, 28, 28, 1)
output: (None, 28, 28, 1)
encoder_conv_0: Conv2D
input: (None, 28, 28, 1)
output: (None, 28, 28, 32)
leaky_re_lu_1: LeakyReLU
input: (None, 28, 28, 32)
output: (None, 28, 28, 32)
encoder_conv_1: Conv2D
input: (None, 28, 28, 32)
output: (None, 14, 14, 64)
leaky_re_lu_2: LeakyReLU
input: (None, 14, 14, 64)
output: (None, 14, 14, 64)
encoder_conv_2: Conv2D
input: (None, 14, 14, 64)
output: (None, 7, 7, 64)
leaky_re_lu_3: LeakyReLU
input: (None, 7, 7, 64)
output: (None, 7, 7, 64)
encoder_conv_3: Conv2D
input: (None, 7, 7, 64)
output: (None, 7, 7, 64)
leaky_re_lu_4: LeakyReLU
input: (None, 7, 7, 64)
output: (None, 7, 7, 64)
flatten_1: Flatten
input: (None, 7, 7, 64)
output: (None, 3136)
mu: Dense
input: (None, 3136)
output: (None, 2)
log_var: Dense
input: (None, 3136)
output: (None, 2)

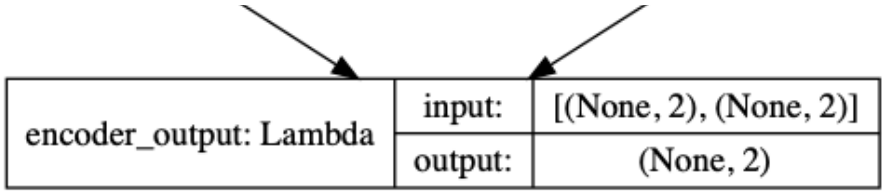


Figure 3-13. Diagram of the VAE encoder

As mentioned previously, the decoder of a variational autoencoder is identical to the decoder of a plain autoencoder. The only other part we need to change is the loss function.

The loss function

Previously, our loss function only consisted of the RMSE loss between images and their reconstruction after being passed through the encoder and decoder. This loss also appears in a variational autoencoder, but we also require one extra component—the KL divergence.

KL divergence (Kullback–Leibler divergence) is actually a way of measuring of how one probability distribution is different from another. In a VAE, we want to measure how different our normal distribution with parameters `mu` and `log_var` is from the standard normal distribution. In this special case, the KL divergence has the closed form given below.

```
kl_loss = -0.5 * sum(1 + log_var - mu ^ 2 - exp(log_var))
```

The sum is taken over all the dimensions in the latent space. `kl_loss` is minimized (0) when `mu = 0` and `log_var = 0` for all dimensions. As these two terms start to differ from 0, `kl_loss` increases.

In summary, the KL divergence term penalizes the network for encoding observations to `mu` and `log_var` variables that differ significantly from the parameters of a standard normal distribution, namely: `mu = 0` and `log_var = 0`.

Again, relating this back to our story, this term represents Epsilon's annoyance at having to move the ladder away from the middle of the wall (`mu` different from 0) and also if Mr N. Coder's confidence in the marker position isn't just right (`log_var` different from 0), both of which incur a cost.

Why does this addition to the loss function help?

Firstly, we now have a well-defined distribution that we can use for choosing points in the latent space—the standard normal distribution. If we sample from this distribution, we know that

we're very likely to get a point that lies within the limits of what the VAE is used to seeing. Secondly, since this term tries to force the cloud of points in the latent space to look as if it were generated by the standard normal distribution, there is less chance that large gaps will form between point clusters. Instead, the encoder will try to use the space around the origin symmetrically and efficiently.

In the code, the loss function for a VAE is simply the addition of the reconstruction loss and the KL divergence loss term. We weight the reconstruction loss with a term `r_loss_factor`, that ensures that it is well balanced with the KL divergence loss. If we weight the reconstruction loss too heavily, the KL loss will not have the desired regulatory effect and we will see the same problems that we experienced with the plain autoencoder. If the weighting term is too small, the KL divergence loss will dominate and the reconstructed images will be poor. This weighting term is one of the parameters to tune when you're training your VAE.

We can include the KL divergence term in our loss function as follows:

Example 3-8.

```
### COMPILATION
optimizer = Adam(lr=learning_rate)

def vae_r_loss(y_true, y_pred):
    r_loss = K.mean(K.square(y_true - y_pred), axis = [1,2,3])
    return r_loss_factor * r_loss

def vae_kl_loss(y_true, y_pred):
    kl_loss =  -0.5 * K.sum(1 + self.log_var - K.square(self.mu) - K.e
xp(self.log_var), axis = 1)
    return kl_loss

def vae_loss(y_true, y_pred):
    r_loss = vae_r_loss(y_true, y_pred)
    kl_loss = vae_kl_loss(y_true, y_pred)
    return  r_loss + kl_loss

optimizer = Adam(lr=learning_rate)
self.model.compile(optimizer=optimizer, loss = vae_loss,  metrics = [v
ae_r_loss, vae_kl_loss])
```

Analysis of the VAE

All of the following analysis is available in the book repository, in the notebook `03_04_vae_digits_analysis.ipynb`.

Referring back to Figure 3-10, we can see several changes in how the latent space is organised.

Firstly, it looks a lot more like a cloud of points that would be generated from a 2D standard normal distribution. This is because the KL divergence loss term ensures that the encoded points never stray too far from this underlying distribution. We can therefore sample from the standard normal distribution to generate new points in the space to be decoded.

Secondly, there are not so many generated digits that are badly formed, since the latent space is now locally continuous due to fact that the encoder is now stochastic, rather than deterministic.

Lastly, by coloring points in the latent space by digit (Figure 3-14), we can see that there is no preferential treatment of any one type. The right hand plot shows the space transformed into p-values and we can see that each color is approximately equally represented. Again, it's important to remember that the labels were not used at all during training—the VAE has learned the various forms of digits by itself in order to help minimize reconstruction loss.

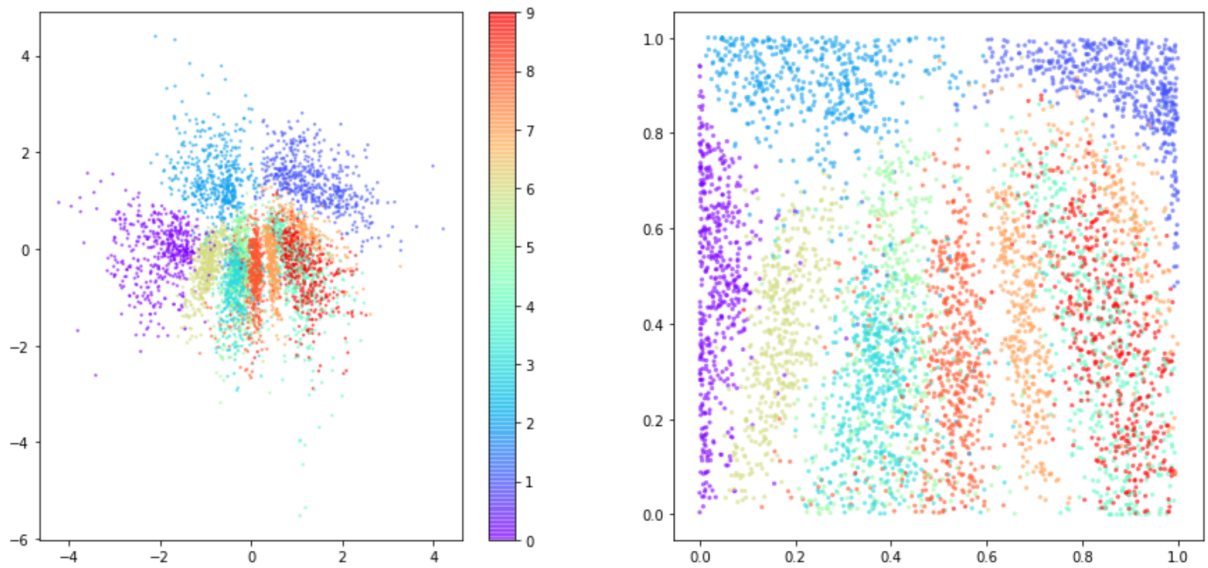


Figure 3-14. The latent space of the VAE colored by digit

So far, all of our work on autoencoders and variational autoencoders has been limited to a latent space with 2 dimensions. This has helped us to visualize the inner workings of a VAE on the page and understand why the small tweaks that we made to the architecture of the autoencoder transform it into a more powerful class of network that can be used for generative modeling.

Let's now turn our attention to a more complex dataset and see the amazing things that variational autoencoders can achieve when we increase the dimensionality of the latent space.

Using VAEs to generate faces

We shall be using the CelebFaces Attributes (CelebA) dataset [5] to train our next variational autoencoder. This is a collection of over 200,000 colour images of celebrity faces, each annotated with various labels (e.g. *wearing hat*, *smiling* etc.). A few examples are shown in

Figure 3-15 :

Figure 3-15. Some examples from the CelebA dataset

Of course, we don't need the labels to train the VAE, but these will come in useful later when we start exploring how these features are captured in the multidimensional latent space. Once our VAE is trained, we can sample from the latent space to generate new examples of celebrity faces.

Training the VAE

The network architecture for the faces model is similar to the digits example, with a few slight differences.

1. Our data now has three input channels (RGB) instead of 1 (greyscale). This means we need to change the number of channels in the final convolutional transpose layer to 3.

2. We shall be using a latent space with 200 dimensions instead of 2. Since faces are much more complex than digits, we increase the dimensionality of the latent space so that the network can encode a satisfactory amount of detail from the images.

3. There are batch normalization layers after each convolution layer to speed up training. Even though each batch takes a longer time to run, the number of batches required to reach the same loss is greatly reduced. Dropout layers are also used.

4. We increase the reconstruction loss factor to 10,000. This is a parameter that requires tuning—for this dataset and architecture a value of 10,000 was found to generate good results.

5. We use a generator to feed images to the VAE from a folder, rather than loading all images into memory up front. Since the VAE trains in batches, there is no need to load all images into memory first, so instead we use the built in `fit_generator()` method that Keras provides to read in images only when they are required for training.

The full architecture of the encoder and decoder are shown in Figure 3-16 and Figure 3-17.

```
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
encoder_input (InputLayer)   (None, 128, 128, 3)  0
_________________________________________________________________________________________________
encoder_conv_0 (Conv2D)      (None, 64, 64, 32)   896         encoder_input[0][0]
_________________________________________________________________________________________________
batch_normalization_1 (BatchNor (None, 64, 64, 32)   128         encoder_conv_0[0][0]
_________________________________________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 64, 64, 32)   0           batch_normalization_1[0][0]
_________________________________________________________________________________________________
dropout_1 (Dropout)          (None, 64, 64, 32)   0           leaky_re_lu_1[0][0]
_________________________________________________________________________________________________
encoder_conv_1 (Conv2D)      (None, 32, 32, 64)   18496       dropout_1[0][0]
_________________________________________________________________________________________________
batch_normalization_2 (BatchNor (None, 32, 32, 64)   256         encoder_conv_1[0][0]
_________________________________________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 32, 32, 64)   0           batch_normalization_2[0][0]
_________________________________________________________________________________________________
dropout_2 (Dropout)          (None, 32, 32, 64)   0           leaky_re_lu_2[0][0]
_________________________________________________________________________________________________
encoder_conv_2 (Conv2D)      (None, 16, 16, 64)   36928       dropout_2[0][0]
_________________________________________________________________________________________________
batch_normalization_3 (BatchNor (None, 16, 16, 64)   256         encoder_conv_2[0][0]
_________________________________________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 16, 16, 64)   0           batch_normalization_3[0][0]
_________________________________________________________________________________________________
dropout_3 (Dropout)          (None, 16, 16, 64)   0           leaky_re_lu_3[0][0]
_________________________________________________________________________________________________
encoder_conv_3 (Conv2D)      (None, 8, 8, 64)     36928       dropout_3[0][0]
_________________________________________________________________________________________________
batch_normalization_4 (BatchNor (None, 8, 8, 64)     256         encoder_conv_3[0][0]
_________________________________________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 8, 8, 64)     0           batch_normalization_4[0][0]
_________________________________________________________________________________________________
dropout_4 (Dropout)          (None, 8, 8, 64)     0           leaky_re_lu_4[0][0]
_________________________________________________________________________________________________
flatten_1 (Flatten)          (None, 4096)         0           dropout_4[0][0]
_________________________________________________________________________________________________
mu (Dense)                   (None, 200)          819400      flatten_1[0][0]
_________________________________________________________________________________________________
log_var (Dense)              (None, 200)          819400      flatten_1[0][0]
_________________________________________________________________________________________________
encoder_output (Lambda)      (None, 200)          0           mu[0][0]
                                                              log_var[0][0]
=================================================================================================
Total params: 1,732,944
Trainable params: 1,732,496
Non-trainable params: 448
_________________________________________________________________________________________________
```

Figure 3-16. The VAE encoder for the faces dataset

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
decoder_input (InputLayer)   (None, 200)               0
_________________________________________________________________
dense_1 (Dense)              (None, 4096)              823296
_________________________________________________________________
reshape_1 (Reshape)          (None, 8, 8, 64)          0
_________________________________________________________________
decoder_conv_t_0 (Conv2DTran (None, 16, 16, 64)        36928
_________________________________________________________________
batch_normalization_5 (Batch (None, 16, 16, 64)        256
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 16, 16, 64)        0
_________________________________________________________________
dropout_5 (Dropout)          (None, 16, 16, 64)        0
_________________________________________________________________
decoder_conv_t_1 (Conv2DTran (None, 32, 32, 64)        36928
_________________________________________________________________
batch_normalization_6 (Batch (None, 32, 32, 64)        256
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 32, 32, 64)        0
_________________________________________________________________
dropout_6 (Dropout)          (None, 32, 32, 64)        0
_________________________________________________________________
decoder_conv_t_2 (Conv2DTran (None, 64, 64, 32)        18464
_________________________________________________________________
batch_normalization_7 (Batch (None, 64, 64, 32)        128
_________________________________________________________________
leaky_re_lu_7 (LeakyReLU)    (None, 64, 64, 32)        0
_________________________________________________________________
dropout_7 (Dropout)          (None, 64, 64, 32)        0
_________________________________________________________________
decoder_conv_t_3 (Conv2DTran (None, 128, 128, 3)       867
_________________________________________________________________
activation_1 (Activation)    (None, 128, 128, 3)       0
=================================================================
Total params: 917,123
Trainable params: 916,803
Non-trainable params: 320
_________________________________________________________________
```

Figure 3-17. The VAE decoder for the faces dataset

To train the VAE on the faces dataset, run the jupyter notebook `03_05_vae_faces_train.ipynb` from the book repository. After around 5 epochs of training your VAE should be able to produce novel images of celebrity faces!

Analysis of the VAE

You can replicate the analysis below by running the notebook `03_06_vae_faces_analysis.ipynb`, once you have trained the VAE. Many of the ideas in this section have been inspired by the paper 'Deep Feature Consistent Variational Autoencoder'[6] by Xianxu Hou, Linlin Shen, Ke Sun and Guoping Qiu.

Firstly, let's take a look at a sample of reconstructed faces. The top row in Figure 3-18 shows the original image and the bottom row shows the reconstruction once it has passed through the encoder and decoder.

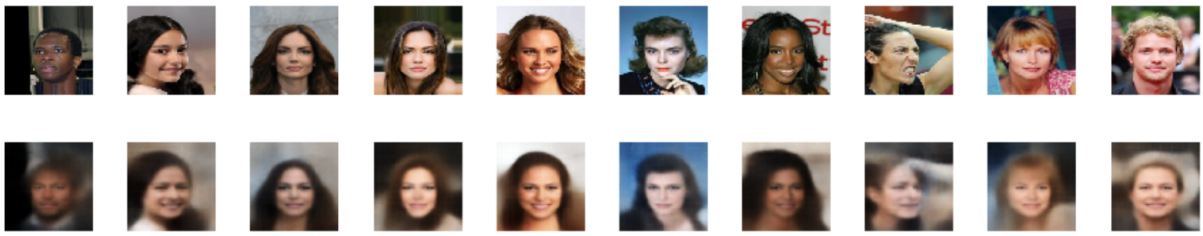


Figure 3-18. Reconstructed faces, after passing through the encoder and decoder.

We can see that the VAE has successfully captured the key features of each face—the angle of the head, the hairstyle, the expression etc. Some of the fine detail is missing, but it is important to remember that the aim of building variational autoencoders isn't to achieve perfect reconstruction loss. Our end goal is to sample from the latent space in order to generate new faces.

For this to be possible we must check that the distribution of points in the latent space is approximately distributed like a multivariate standard normal distribution. Since we cannot view all dimensions simultaneously, we can instead check the distribution of each latent dimension individually—if we see any dimensions that are significantly different from a standard normal, we should probably reduce the reconstruction loss factor, since the KL divergence term isn't having enough effect.

The first 50 dimensions in our latest space are shown in Figure 3-19. There aren't any distributions that stand out as being significantly different from the standard normal, so we can move on to generating some faces!

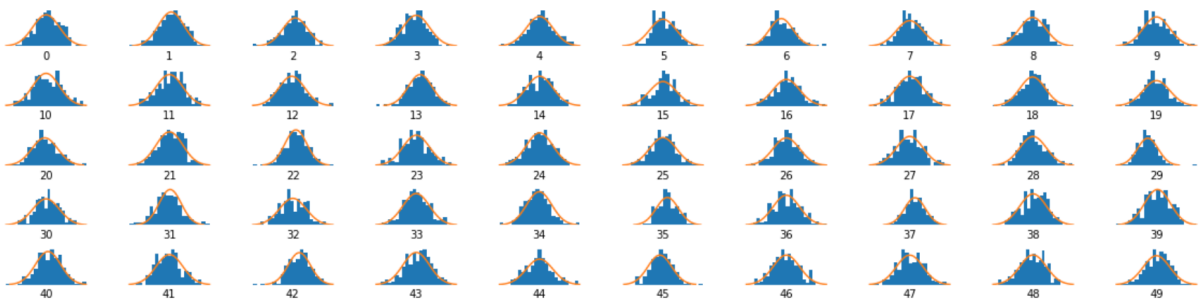


Figure 3-19. Distributions of points for the first 50 dimensions in the latent space.

Generating new faces

To generate new faces, we can use the following code:

Example 3-9.

```
n_to_show = 30

znew = np.random.normal(size = (n_to_show,VAE.z_dim)) ❶

reconst = VAE.decoder.predict(np.array(znew)) ❷

fig = plt.figure(figsize=(18, 5))
fig.subplots_adjust(hspace=0.4, wspace=0.4)
for i in range(n_to_show):
    ax = fig.add_subplot(3, 10, i+1)
    ax.imshow(reconst[i, :,:,:]) ❸
    ax.axis('off')

plt.show()
```

❶ We sample 30 points from a standard normal distribution with 200 dimensions…

❷ … then pass these points to the decoder

❸ The resulting output is a 128 x 128 x 3 image that we can view

The output is shown in Figure 3-20

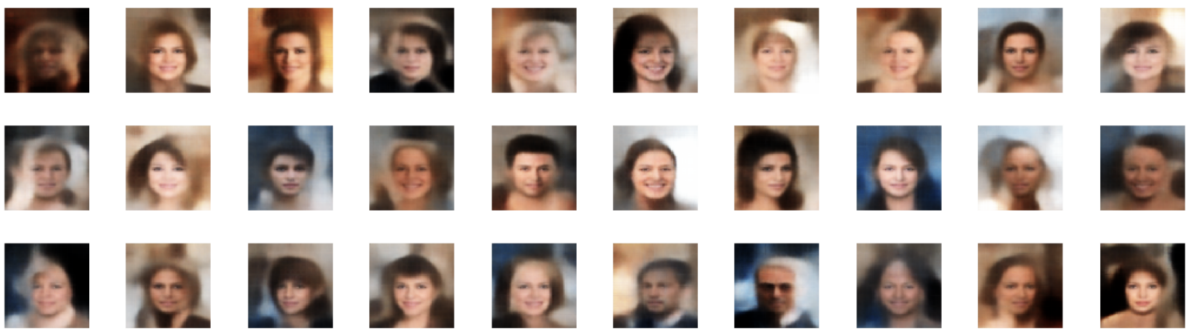


Figure 3-20. New generated faces

Amazingly, the VAE is able to take the set of points that we sampled and convert each into a convincing image of a person's face. Whilst the images are not perfect, they are a giant leap forward from the Naive Bayes model that we started exploring in Chapter 1. The Naive Bayes model faced the problem of not being able to capture dependency between adjacent pixels, since it had no notion of higher level features such as *sunglasses* or *brown hair*. The VAE doesn't suffer from this problem, since the convolutional layers of the encoder are designed to translate

low level pixels into high level features and the decoder is trained to perform the opposite task of translating the high level features in the latent space, back to raw pixels.

Latent space arithmetic

One benefit of mapping images into a lower dimensional space is that we can perform arithmetic on vectors in this latent space that has a visual analogue when decoded back into the original image domain.

For example, suppose we want to take an image of somebody who looks sad and give them a smile. To do this we first need to find a vector in the latent space that points in the direction of increasing smile. Adding this vector to the encoding of the original image in the latent space will give us a new point which when decoded, should give us a more smiley version of the original image.

So how can we find the *smile* vector? Each image in the CelebA dataset is labeled with attributes, one of which is *smiling*. If we take the average position of an encoded image in the latent space with the attribute *smiling* and subtract the average position of encoded images that do not have the attribute *smiling*, we will obtain the vector that points from *not smiling* to *smiling*, which is exactly what we need.

Conceptually, we are performing the following vector arithmetic in the latent space, where alpha is multiple that determines how much of the feature vector is added or subtracted.

```
z_new = z + alpha * feature_vector
```

Let's see this in action. Figure 3-21 shows several images that have been encoded into the latent space. We then add or subtract multiples of a certain vector (e.g. smile, blonde, male, eyeglasses) to obtain different versions of the image, with only the relevant feature changed.

Figure 3-21. Adding and subtracting features to faces

It is quite remarkable that even though we are moving the point a significantly large distance in the latent space, the core image barely changes, except for the one feature that we want to manipulate. This demonstrate the power of variational encoders for capturing and adjusting high level features in images.

Morphing between faces

We can use a similar idea to morph between two faces. Imagine two points in the latent space, A and B, that represent two images. If you start at point A and walk towards point B in a straight line, decoding points on the line as you go, you would see a gradual transition from the starting face to the end face.

Mathematically, we are traversing a straight line, which can be described by the following equation.

```
z_new = z_A * (1- alpha) + z_B * alpha
```

Here, alpha is number between 0 and 1 that determines how far along the line we are, away from point A.

Figure 3-21 shows this is process in action. We take two images, encode them into the latent space and then decode points along the straight line between them, at regular intervals.

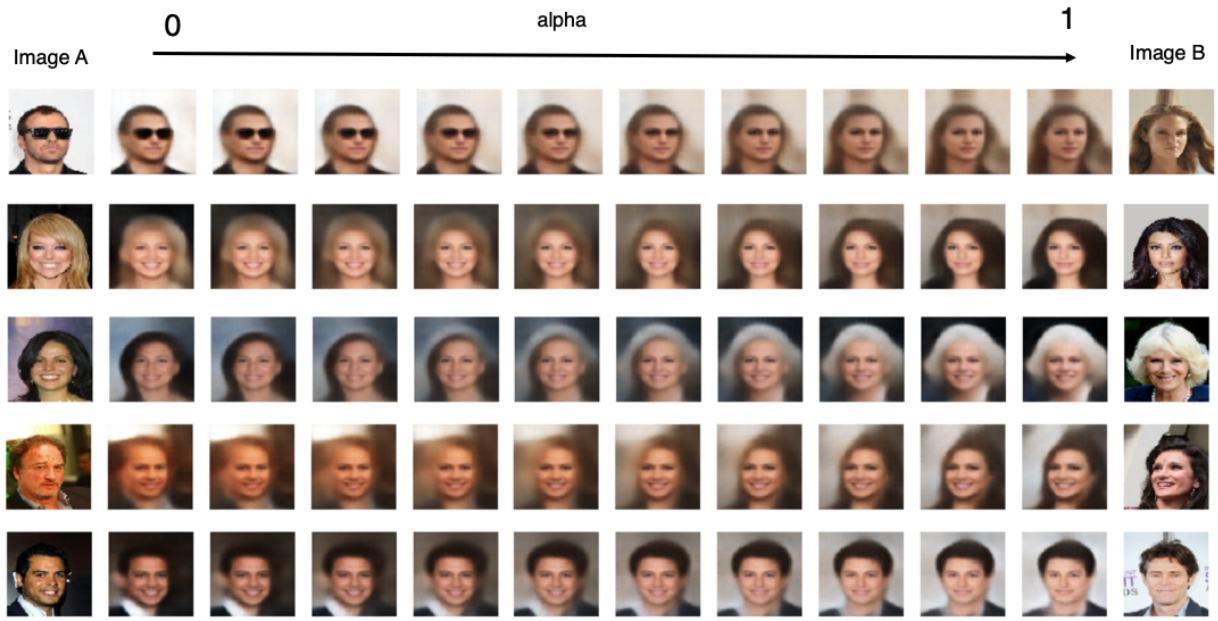


Figure 3-22. Morphing between two faces

It is worth noting the smoothness of the transition—even where there are multiple features to change simultaneously (e.g. removal of eyeglasses, hair color, gender etc.), the VAE manages to achieve this fluidly, showing that the latent space of the VAE is truly a continuous space that can be traversed and explored to generate a multitude of different human faces.

Summary

In this chapter we have seen how autoencoders are a powerful tool in the generative modeling toolbox. We started by exploring how plain autoencoders can be used to map high dimensional images into a low dimensional latent space, in order that high level features can be extracted from the individually uninformative pixels. However, like the Coder brother's art exhibition, we quickly found that there were some drawbacks to using plain autoencoders as a generative model—sampling from the learned latent space was problematic for a number of reasons.

Variational autoencoders solve these problems, by introducing randomness into the model and constraining how points in the latent space are distributed. We saw that with a few minor adjustments, we can transform our autoencoder into a variational autoencoder, thus giving it the power to be a generative model.

Lastly, we applied our new technique to the problem of face generation and saw how we can

simply choose points from a standard normal distribution to generate new faces. Moreover, by performing vector arithmetic within the latent space, we can achieve some amazing effects, such as face morphing and feature manipulation. With these features, it is easy to see why VAEs have become a prominent technique for generative modeling in recent years.

In the next chapter, we shall explore a different kind of generative model, that has attracted an equal amount of attention—the Generative Adversarial Network.

---

1 *https://arxiv.org/abs/1312.6114*

2 source: *https://arxiv.org/pdf/1603.07285.pdf*

3 source: *https://en.wikipedia.org/wiki/Normal_distribution*

4 $\exp\left(\log\left(\sigma^2\right)/2\right) = \exp\left(2\log\left(\sigma\right)/2\right) = \exp\left(\log\left(\sigma\right)\right) = \sigma$

5 *http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html*

6 *https://arxiv.org/abs/1610.00291*

Chapter 4. Generative Adversarial Networks

> **NOTE**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as he or she writes—so you can take advantage of these technologies long before the official release of these titles.

On Monday 5th December 2016 at 2.30 p.m., Ian Goodfellow of Google Brain presented a tutorial entitled *Generative Adversarial Networks* to the delegates of the Neural Information Processing Systems (NIPS) conference in Barcelona [1] [2] . The ideas presented in the tutorial are now regarded as one of the key turning points for generative modeling and have spawned a wide variety of variations on his core idea that have pushed the field to even greater heights.

In this chapter we shall first lay out the theoretical underpinning of GANs. You will then learn how to use the Python library Keras to start building your own Generative Adversarial Networks (GANs).

GANimals

One afternoon, whilst walking through the local jungle, Gene sees a woman thumbing through a set of black and white photographs, looking worried. He goes over to ask if he can help.

The woman introduces herself as Di, a keen explorer, and explains that she is hunting for the elusive *ganimal*, a mythical creature that is said to roam around the jungle. Since the creature is nocturnal, she only has a collection of night-time photos of the beast that she once found lying on the floor of the jungle, dropped by another ganimal enthusiast—some of these photos are shown in Figure 4-1. Di makes money by selling the images to collectors but is starting to worry, as she hasn't actually ever seen the creature and is worried that her business will falter if

she can’t produce more original photographs soon.

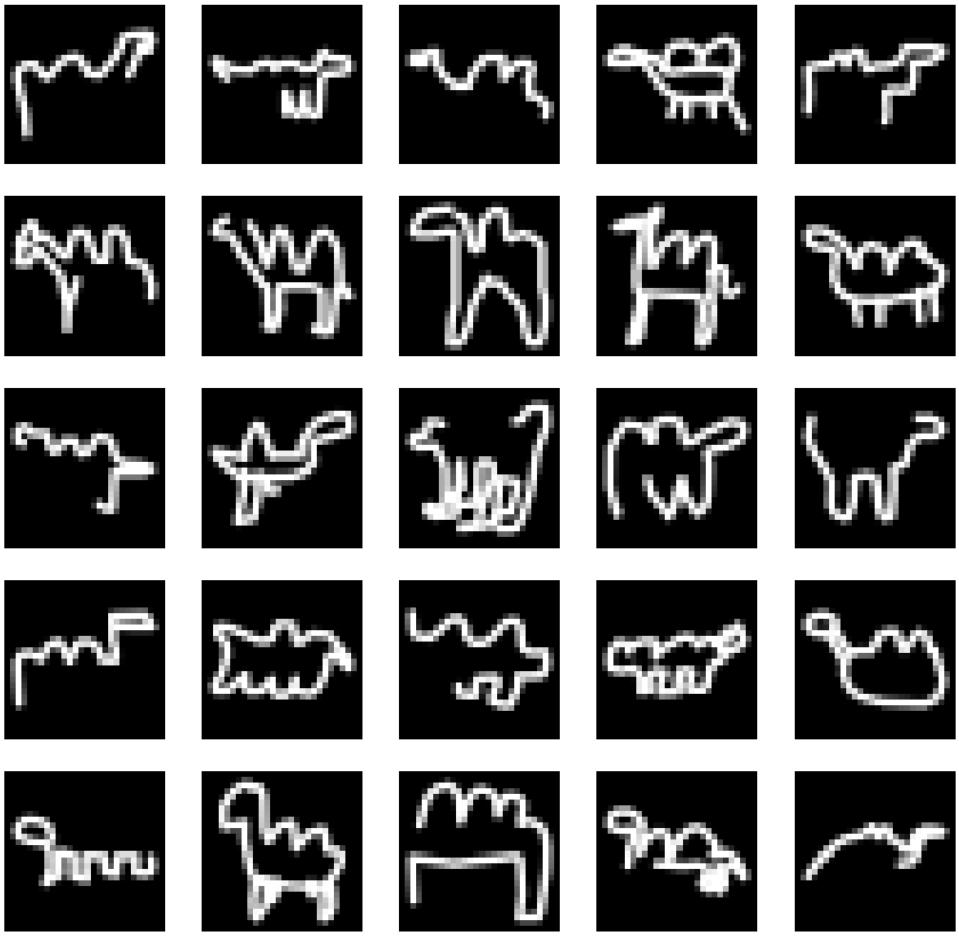


Figure 4-1. Original ganimal photographs

Being a keen photographer, Gene decides to help Di. He agrees to search for the ganimal himself and give her any photographs of the nocturnal beast that he manages to take.

However, there is a problem. Since Gene has never seen a ganimal, he doesn’t know how to produce good photos of the creature and also, since Di has only ever sold the photos she found, she cannot even tell the difference between a good photo of a ganimal and photo of nothing at all.

Starting from this state of naïvity, how can they work together to ensure Gene is eventually able to produce impressive ganimal photography?

They come up with following process. Each night, Gene takes 64 photographs, each in a

different location with different random moonlight readings, and mixes them with 64 ganimal photos from the original collection. Di then looks at this set of photos and tries to guess which are taken by Gene and which were originals. Based on her mistakes, she updates her own understanding of how to discriminate between Gene's attempts and the original photos. Afterwards, Gene takes another 64 photos and shows them to Di. Di gives each photo a score between 0 and 1, indicating how good she thinks each photo is. Based on this feedback, Gene updates the settings on his camera to ensure that next time, he takes photos that Di is more likely to rate highly.

This process continues for many days. Initially, Gene doesn't get any useful feedback from Di, since she is randomly guessing which photos are genuine. However, after a few weeks of her training ritual, she gradually gets better at this, which means that she can provide better feedback to Gene so that he can adjust his camera accordingly in his training session. This makes Di's task harder, since now Gene's photos aren't quite as easy to distinguish from the real photos, so she must again learn how to improve. This back-and-forth process continues, over many days and weeks.

Over time, Gene gets better and better at producing ganimal photos, until eventually, Di is once again resigned to the fact that she cannot tell the difference between Gene's photos and the originals. They take Gene's generated photos to the auction and the experts cannot believe the quality of the new sightings—they are just as convincing as the originals. Some examples of Gene's work are shown in Figure 4-2.

Figure 4-2. Samples of Gene's ganimal photography

Generative Adversarial Networks

The adventures of Gene and Di hunting elusive nocturnal camel-like ganimals are a metaphor for one of the most important deep learning advancements of recent years—Generative Adversarial Networks (or GANs for short).

Simply put, a GAN is a battle between two adversaries—the generator and the discriminator.

The generator tries to convert random noise into observations that look as if they have been sampled from the original dataset and the discriminator tries to predict whether an observation comes from the original dataset or is one of the generator's forgeries. The inputs and outputs to the two networks are shown in Figure 4-3.

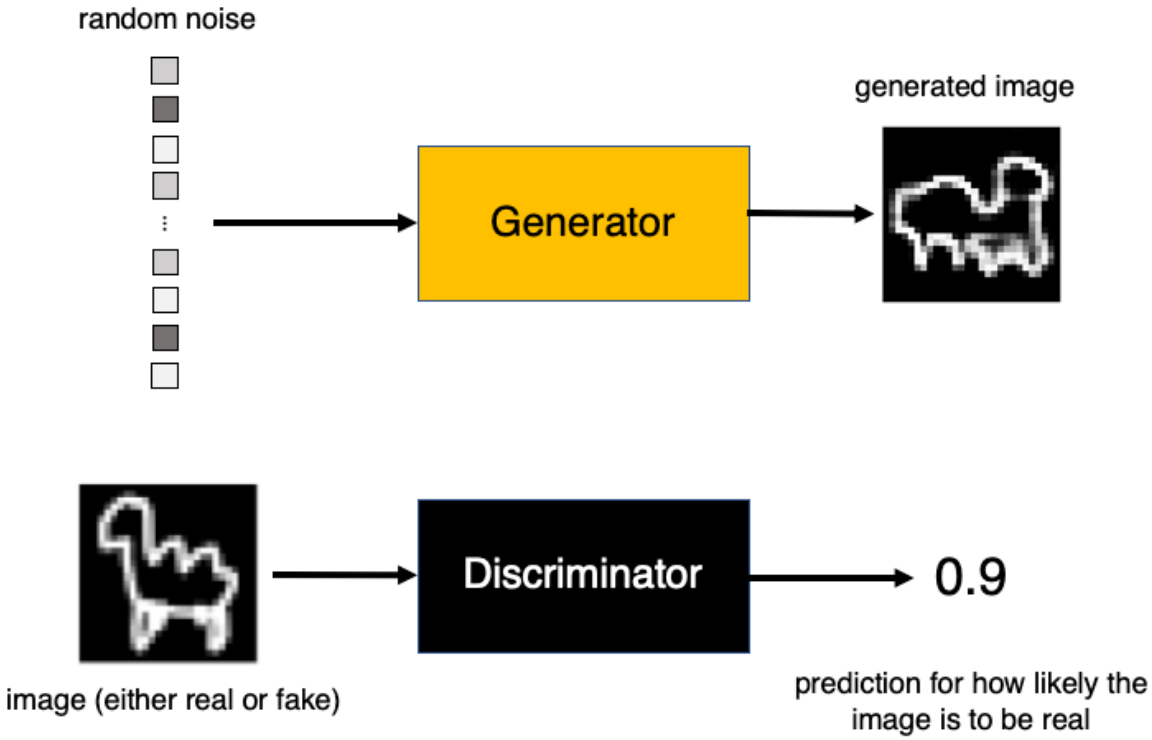


Figure 4-3. Inputs and outputs of the two networks in a GAN

At the start of the process, the generator outputs noisy images and the discriminator predicts randomly. The key to GANs lies in how we alternate the training of the two networks, so that as the generator becomes more adept at fooling the discriminator, the discriminator must adapt in order to maintain its ability to correctly identify which observations are fakes. This drives the generator to find new ways to fool the discriminator and so the cycle continues.

To see this in action, let's start building our first GAN in Keras, to generate pictures of nocturnal ganimals.

Your first GAN

Firstly, you'll need to download the training data. We'll be using the *Quick, Draw!* dataset[3] from Google. This is a crowd-sourced collection of 28 x 28 greyscale doodles, labelled by subject. The dataset was collected as part of an online game that challenged players to draw a picture of an object or concept, whilst a neural network tries to guess the subject of the doodle. It's a really useful and fun dataset for learning the fundamentals of deep learning. For this task you'll need to download the *camel* numpy file[4] and save it into the ./data/camel/ folder in the

book repository. The original data is scaled in the range [0, 255] to denote the pixel intensity. For this GAN we rescale the data to the range [-1, 1].

Running the notebook `04_01_gan_camel_train.ipynb` in the book repository will start training the GAN. The architecture of the GAN is stored in `./models/GAN.py`. Similarly to the previous chapter on VAEs, you can instantiate a GAN object in the notebook and play around with the parameters to see how it affects the model.

Example 4-1.

```
gan = GAN(input_dim = (28,28,1)
            , discriminator_conv_filters = [64,64,128,128]
            , discriminator_conv_kernel_size = [5,5,5,5]
            , discriminator_conv_strides = [2,2,2,1]
            , discriminator_batch_norm_momentum = None
            , discriminator_activation = 'relu'
            , discriminator_dropout_rate = 0.4
            , discriminator_learning_rate = 0.0008
            , generator_initial_dense_layer_size = (7, 7, 64)
            , generator_upsample = [2,2, 1, 1]
            , generator_conv_filters = [128,64, 64,1]
            , generator_conv_kernel_size = [5,5,5,5]
            , generator_conv_strides = [1,1, 1, 1]
            , generator_batch_norm_momentum = 0.9
            , generator_activation = 'relu'
            , generator_dropout_rate = None
            , generator_learning_rate = 0.0004
            , optimiser = 'rmsprop'
            , z_dim = 100
            )
```

Let's first take a look at how we build the discriminator.

The discriminator

The goal of the discriminator is to predict if an image is real or fake. This is an supervised image classification problem so we can use the same network architecture as in Chapter 2—stacked convolutional layers, followed by a dense output layer.

In the original GAN paper, dense layers were used in place of the convolutional layers—however, since then, it has been shown that convolutional layers give greater predictive power to the discriminator. You may see this type of GAN called a DCGAN (Deep Convolutional Generative Adversarial Network) in the literature, but now essentially all GAN architectures contain convolutional layers, so the 'DC' is implied when we talk about GANs. It is also common to see batch normalization layers in the discriminator for vanilla GANs, though we

choose not to use them here for simplicity.

The full architecture of the discriminator we will be building is shown in Figure 4-4.

Layer (type)	Output Shape	Param #
discriminator_input (InputLa	(None, 28, 28, 1)	0
discriminator_conv_0 (Conv2D	(None, 14, 14, 64)	1664
activation_1 (Activation)	(None, 14, 14, 64)	0
dropout_1 (Dropout)	(None, 14, 14, 64)	0
discriminator_conv_1 (Conv2D	(None, 7, 7, 64)	102464
activation_2 (Activation)	(None, 7, 7, 64)	0
dropout_2 (Dropout)	(None, 7, 7, 64)	0
discriminator_conv_2 (Conv2D	(None, 4, 4, 128)	204928
activation_3 (Activation)	(None, 4, 4, 128)	0
dropout_3 (Dropout)	(None, 4, 4, 128)	0
discriminator_conv_3 (Conv2D	(None, 4, 4, 128)	409728
activation_4 (Activation)	(None, 4, 4, 128)	0
dropout_4 (Dropout)	(None, 4, 4, 128)	0
flatten_1 (Flatten)	(None, 2048)	0
dense_1 (Dense)	(None, 1)	2049

Total params: 720,833
Trainable params: 720,833
Non-trainable params: 0

Figure 4-4. The discriminator of the GAN

The Keras code to build the discriminator is provided below.

Example 4-2.

```
discriminator_input = Input(shape=self.input_dim, name='discriminator_
input') ❶
x = discriminator_input

for i in range(self.n_layers_discriminator): ❷
```

```
        x = Conv2D(
            filters = self.discriminator_conv_filters[i]
            , kernel_size = self.discriminator_conv_kernel_size[i]
            , strides = self.discriminator_conv_strides[i]
            , padding = 'same'
            , name = 'discriminator_conv_' + str(i)
            )(x)

        if self.discriminator_batch_norm_momentum and i > 0:
            x = BatchNormalization(momentum = self.discriminator_batch_nor
    m_momentum)(x)

        x = Activation(self.discriminator_activation)(x)

        if self.discriminator_dropout_rate:
            x = Dropout(rate = self.discriminator_dropout_rate)(x)

    x = Flatten()(x) ❸
    discriminator_output= Dense(1, activation='sigmoid', kernel_initialize
    r = self.weight_init)(x) ❹

    discriminator = Model(discriminator_input, discriminator_output) ❺
```

❶ Define the input to the discriminator (the image)

❷ Stack convolutional layers on top of each other

❸ Flatten the last convolutional layer to a vector

❹ Dense layer of one unit, with a sigmoid activation function that transforms the output from the dense layer to the range [0,1]

❺ The Keras model that defines the discriminator—a model that take an input image and outputs a single number between 0 and 1

Notice how we use a stride of 2 in some of the convolutional layers to reduce the size of the tensor as it passes through the network, but increase the number of channels (1 in the greyscale input image, then 64, then 128).

The sigmoid activation in the final layer ensures that the output is scaled between 0 and 1. This will be the predicted probability that the image is real.

The generator

Now let's build the generator. The input to the generator is a vector, usually drawn from a multivariate standard normal distribution. The output is an image of the same size as an image in the original training data.

This description may remind you of the decoder in a Variational Autoencoder (VAE). In fact, the generator of a GAN fulfills exactly the same purpose as the decoder of a VAE—converting a vector in the latent space to an image. The concept of mapping from a latent space back to the original domain is very common in generative modeling as it gives us the ability to manipulate vectors in the latent space to change high level features of images in the original domain.

The architecture of the generator we will be building is shown in Figure 4-5.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
generator_input (InputLayer) (None, 100)               0
_________________________________________________________________
dense_9 (Dense)              (None, 3136)              316736
_________________________________________________________________
batch_normalization_10 (Batc (None, 3136)              12544
_________________________________________________________________
activation_36 (Activation)   (None, 3136)              0
_________________________________________________________________
reshape_4 (Reshape)          (None, 7, 7, 64)          0
_________________________________________________________________
up_sampling2d_10 (UpSampling (None, 14, 14, 64)        0
_________________________________________________________________
generator_conv_0 (Conv2D)    (None, 14, 14, 128)       204928
_________________________________________________________________
batch_normalization_11 (Batc (None, 14, 14, 128)       512
_________________________________________________________________
activation_37 (Activation)   (None, 14, 14, 128)       0
_________________________________________________________________
up_sampling2d_11 (UpSampling (None, 28, 28, 128)       0
_________________________________________________________________
generator_conv_1 (Conv2D)    (None, 28, 28, 64)        204864
_________________________________________________________________
batch_normalization_12 (Batc (None, 28, 28, 64)        256
_________________________________________________________________
activation_38 (Activation)   (None, 28, 28, 64)        0
_________________________________________________________________
generator_conv_2 (Conv2D)    (None, 28, 28, 64)        102464
_________________________________________________________________
batch_normalization_13 (Batc (None, 28, 28, 64)        256
_________________________________________________________________
activation_39 (Activation)   (None, 28, 28, 64)        0
_________________________________________________________________
generator_conv_3 (Conv2D)    (None, 28, 28, 1)         1601
_________________________________________________________________
activation_40 (Activation)   (None, 28, 28, 1)         0
=================================================================
Total params: 844,161
Trainable params: 837,377
Non-trainable params: 6,784
```

Figure 4-5. The generator

UPSAMPLING

In the decoder of the variational autoencoder that we built in the previous chapter, we doubled the width and height of the tensor at each layer using `Conv2DTranspose` layers with stride 2. This inserted zero values in between pixels before performing the convolution operations.

In this GAN, we instead use the Keras `Upsampling2D` layer to double the *width* and *height* of the input tensor—this simply repeats each row and column of its input in order to double the size. We then follow this with a normal convolutional layer with stride 1 to perform the convolution operation. It is a similar idea to convolutional transpose, but instead of filling the gaps between pixels with zeros, upsampling just repeats the existing pixel values.

Both of these two methods—`Upsampling` + `Conv2D`, and `Conv2DTranspose` are acceptable ways to transform back to the original image domain. It really is a case of testing both methods in your own problem setting and see which produces better results. It has been reported[5] that the `Conv2DTranspose` method can lead to *artifacts*—small checkboard patterns in the output image (Figure 4-6), that spoil the quality of the output. However, they are still used in many of the most impressive GANs in the literature and have proven to be a powerful tool in the deep learning practitioners toolbox—I suggest you experiment with both methods and see which works best for you.

Figure 4-6. Artifacts - source (https://distill.pub/2016/deconv-checkerboard/)

The code for building the generator is given below:

Example 4-3. The generator

```
generator_input = Input(shape=(self.z_dim,), name='generator_input') ❶
x = generator_input
```

```
x = Dense(np.prod(self.generator_initial_dense_layer_size))(x) ❷

if self.generator_batch_norm_momentum:
    x = BatchNormalization(momentum = self.generator_batch_norm_moment
um)(x)
x = Activation(self.generator_activation)(x)

x = Reshape(self.generator_initial_dense_layer_size)(x) ❸

if self.generator_dropout_rate:
    x = Dropout(rate = self.generator_dropout_rate)(x)

for i in range(self.n_layers_generator): ❹

    x = UpSampling2D()(x)
    x = Conv2D(
        filters = self.generator_conv_filters[i]
        , kernel_size = self.generator_conv_kernel_size[i]
        , padding = 'same'
        , name = 'generator_conv_' + str(i)
        )(x)

    if i < n_layers_generator - 1: ❺
        if self.generator_batch_norm_momentum:
            x = BatchNormalization(momentum = self.generator_batch_nor
m_momentum))(x)
        x = Activation('relu')(x)
    else:
        x = Activation('tanh')(x)


generator_output = x
generator = Model(generator_input, generator_output) ❻
```

❶ Define the input to the generator—a vector of length 100.

❷ We follow this with a `Dense` layer consisting of 3136 units…

❸ …which, after applying batch normalization and a ReLU activation function, is reshaped to a 7 x 7 x 64 tensor.

❹ We pass this through four `Conv2D` layers. The first two preceded by `Upsampling2D` layers, to reshape the tensor to 14 x 14 then, 28 x 28 (the original image size). In all but the last layer, we use batch normalization and ReLU activation (LeakyReLU could also be used).

❺ After the final `Conv2D` layer, we use a tanh activation to transform the output to the range [-1, 1], to match the original image domain.

❻ The Keras model that defines the generator—a model that accepts a vector of length 100 and outputs an image of size 28 x 28 x 1.

Training the GAN

As we have seen, the architecture of both the generator and discriminator in a GAN is very simple and not so different from the models that we have already seen. The key to understanding GANs is in understanding the training process.

We can train the discriminator by creating a training set where some of the images are randomly selected *real* observations from the training set and some are outputs from the generator. The response would be *1* for the true images and *0* for the generated images. If we treat this as a supervised learning problem, we can train the discriminator to learn how to tell the difference between the the original and generated images, outputting values near *1* for the true images and values near *0* for the fake images.

Training the generator is more difficult as there is no training set that tells us the *true* image that a particular point in the latent space should be mapped to. Instead, we only wish that the image that is generated fools the discriminator—that is, when the image is fed as input to the discriminator, the output is close to *1*.

Therefore, to train the generator, we must first connect it to the discriminator to create a Keras model that we can train. Specifically, we feed the output from the generator (a 28 x 28 x 1 image) into the discriminator so that the output from this combined model is the probability that the generated image is *real*, according to the discriminator. We can train this combined model, by creating training batches consisting of randomly generated 100-dimensional latent vectors as input and a response which is always *1*, since we want to train the generator to produce images that the discriminator thinks are real.

Crucially, we must freeze the weights of the discriminator whilst we are training the combined model, so that only the generator's weights are updated. If we do not freeze the discriminator's weights, the discriminator will adjust so that it is more likely to predict generated images as real, which is not the desired outcome. We want generated images to be predicted close to *1* (real) because the generator is strong, not because the discriminator is weak.

A diagram of the training process for the discriminator and generator is shown in Figure 4-7.

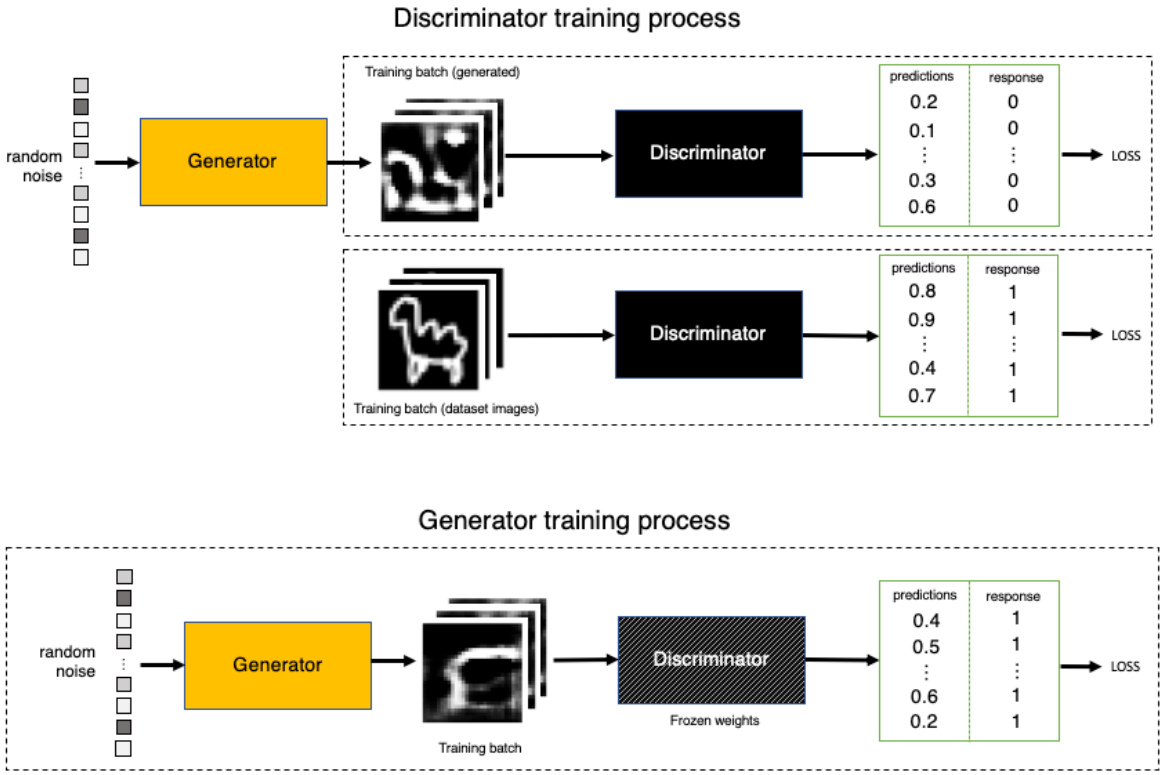


Figure 4-7. Training the GAN

Let's see what this looks like in code. First we need to compile the discriminator model and the model that trains the generator:

Example 4-4. Compiling the GAN

```
### COMPILE MODEL THAT TRAINS THE DISCRIMINATOR

self.discriminator.compile(
    optimizer= RMSprop(lr=0.0008)
    , loss = 'binary_crossentropy'
    ,  metrics = ['accuracy']
) ❶

### COMPILE MODEL THAT TRAINS THE GENERATOR

self.discriminator.trainable = False ❷
model_input = Input(shape=(self.z_dim,), name='model_input')
model_output = discriminator(self.generator(model_input))
self.model = Model(model_input, model_output) ❸

self.model.compile(
    optimizer=RMSprop(lr=0.0004)
    , loss='binary_crossentropy'
    , metrics=['accuracy']
) ❹
```

❶ The discriminator is compiled with binary cross-entropy loss, as the response is binary and we have one output node with sigmoid activation.

❷ Next, we freeze the discriminator weights—this doesn't affect the existing discriminator model that we have already compiled.

❸ We define a new model whose input is a 100-dimensional latent vector—this is passed through the generator and frozen discriminator to produce the output probability.

❹ Again we use a binary cross-entropy loss for the combined model—the learning rate is slower that the discriminator as generally, we would like the discriminator to be stronger than the generator. The learning rate is a parameter that should be tuned carefully for each GAN problem setting.

Then we train the GAN by alternating training of the discriminator and generator.

Example 4-5. Training the GAN

```
def train_discriminator(x_train, batch_size):

    valid = np.ones((batch_size,1))
    fake = np.zeros((batch_size,1))

    # TRAIN ON REAL IMAGES
    idx = np.random.randint(0, x_train.shape[0], batch_size)
    true_imgs = x_train[idx]

    self.discriminator.train_on_batch(true_imgs, valid) ❶

    # TRAIN ON GENERATED IMAGES
    noise = np.random.normal(0, 1, (batch_size, z_dim))
    gen_imgs = generator.predict(noise)

    self.discriminator.train_on_batch(gen_imgs, fake) ❷


def train_generator(batch_size):

    valid = np.ones((batch_size,1))

    noise = np.random.normal(0, 1, (batch_size, z_dim))
    self.model.train_on_batch(noise, valid) ❸

epochs = 2000
batch_size = 64

for epoch in range(epochs):
```

```
        train_discriminator(x_train, batch_size)
        train_generator(batch_size)
```

❶ One batch update of the discriminator involves first training on a batch of true images with the response *1*…

❷ …then on a batch of generated images, with the response *0*.

❸ One batch update of the generator involves training on a batch of generated images, with the response *1*. As the discriminator is frozen, its weights will not be affected—instead the generator weights will move in the direction that allows it to better generate images that are more likely to fool the discriminator (i.e. make the discriminator predict values close to *1*).

After a suitable number of epochs, the discriminator and generator will have found an equilibrium that allows the generator to learn meaningful information from the discriminator and the quality of the images will start to improve(Figure 4-8).

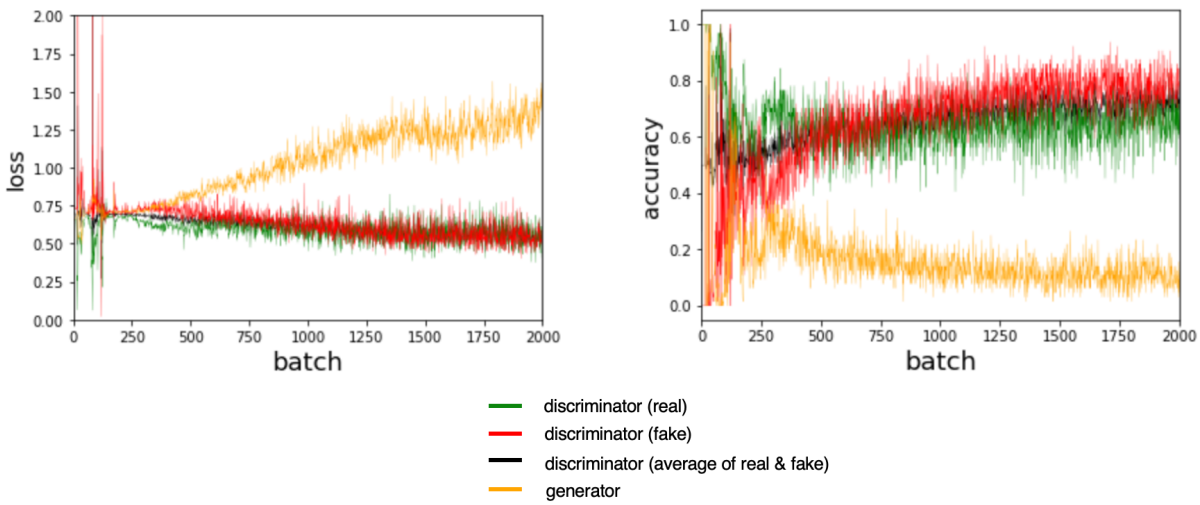


Figure 4-8. Loss and accuracy of the discriminator and generator during training

By observing images produced by the generator at specific epochs during training (Figure 4-9), it is clear that the generator is becoming increasingly adept at producing images that could have been drawn from the training set.

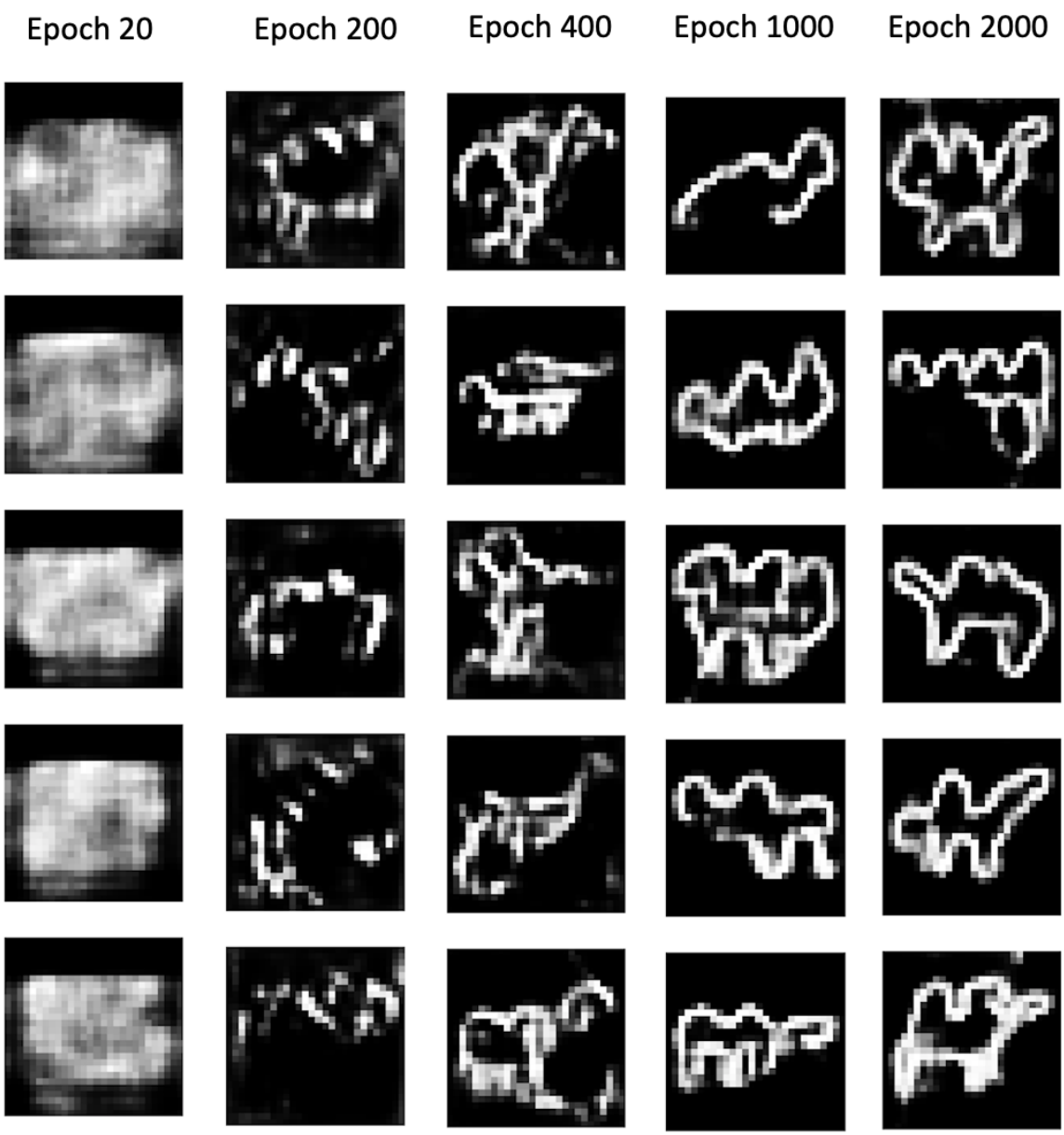


Figure 4-9. Output from the generator at specific epochs during traiing

It is somewhat miraculous that a neural network is able to convert random noise into something meaningful. It is worth remembering that we haven't provide the model with any additional features beyond the raw pixels, so it has to work out high-level concepts such as how to draw a *hump*, *legs* or a *head*, entirely by itself. The Naive Bayes models that we saw in Chapter 1 wouldn't be able to achieve this level of sophistication since they cannot model the interdependencies between pixels that are crucial to forming these high level features.

Another requirement of a successful generative model is that is doesn't only reproduce images from the training set. To test this, we can find the image from the training set that is 'closest' to a particular generated example. A good measure for distance is the L1 distance, shown below:

```
def l1_compare_images(img1, img2):
    return np.mean(np.abs(img1 - img2))
```

Figure 4-10 shows the closest observation in the training set for a selection of generated images. We can see that whilst there is some degree of similarity between the generated images and the training set, they are not identical and it is also able to complete some of the unfinished drawings by, for example, adding legs or a head. This shows that the generator has understood these high level features and can generate examples that are distinct from those it has already seen.

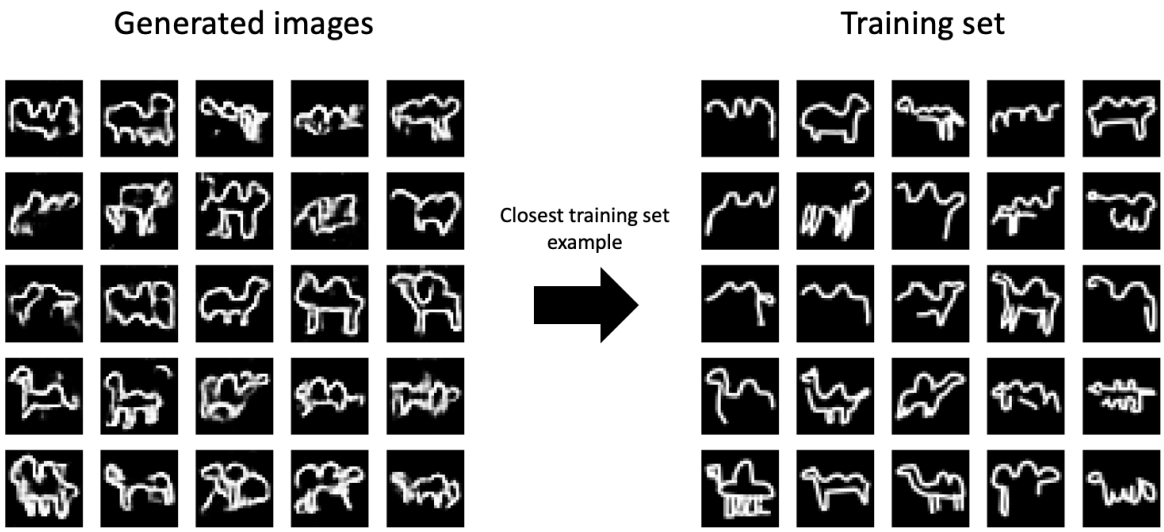


Figure 4-10. Cloest matches of generated images from the training set

GAN challenges

Whilst GANs are a major breakthrough for generative modeling, the downside is that they are notoriously difficult to train. When applied to more complex problems, there are several things that can go wrong.

Oscillating loss

The loss of the discriminator and generator can start to oscillate wildly, rather than exhibiting long-term stability. Typically, there is some small oscillation of the loss between batches, but in the long-term you should be looking for loss that stablilizes or gradually increases or decreases (see Figure 4-8), rather than erratically fluctuating, to ensure your GAN converges and improves over time. Figure 4-11 shows an example of a GAN where the loss of the discriminator and generator has started to spiral out of control, around batch 1400. It is difficult to establish if or when this might occur as vanilla GANs are prone to this kind of instability.

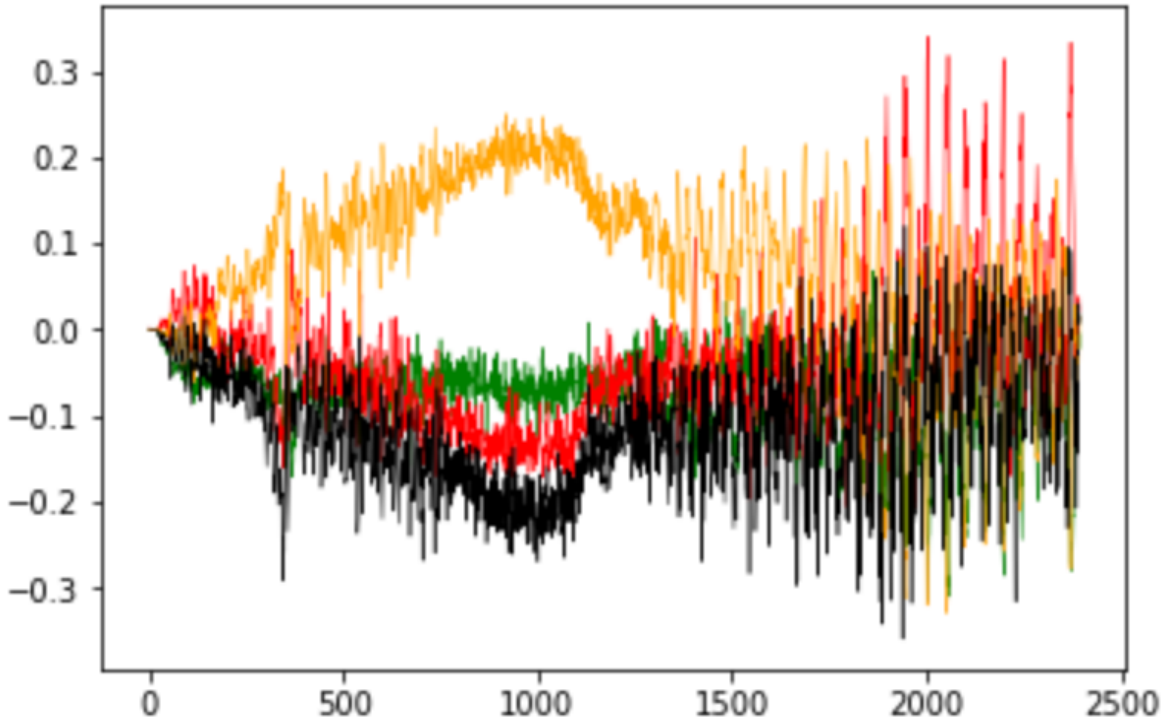


Figure 4-11. Oscillating loss in an unstable GAN

Mode collapse

Mode collapse occurs when the generator finds a small number of samples that fool the discriminator and therefore isn't able to produce any examples other than this limited set of images. Let's think about how this might occur. Suppose we train the generator over several batches without updating the discriminator in between. The generator would be inclined to find the single point (also know as a *mode*), that always fools the discriminator, no matter what the latent input. This means that the gradient of the loss function collapses to near 0. Even if we then try to retrain the discriminator to stop it being fooled by this one point, the generator will simply find another mode that fools the discriminator, since it is has already become numb to its input and therefore has no incentive to diversify its output. The effect of mode collapse can be seen in Figure 4-12.

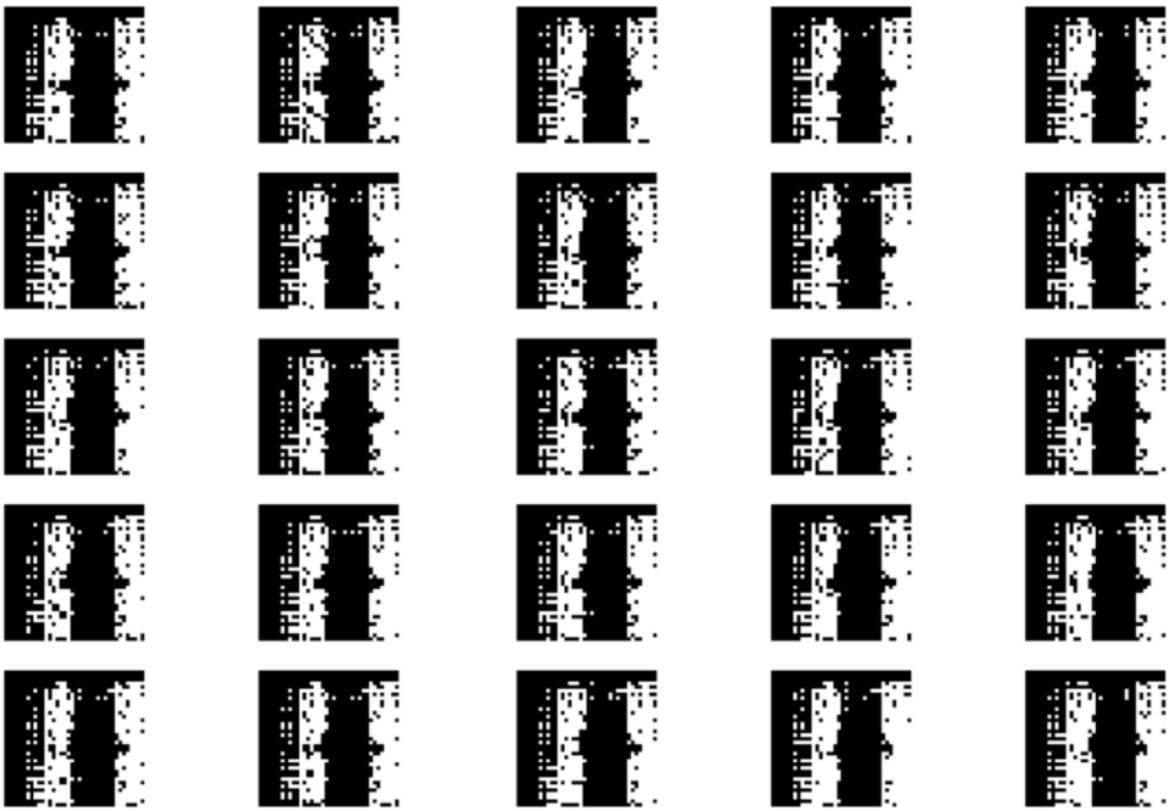


Figure 4-12. Mode collapse

Uninformative loss

Since the deep learning model is compiled to minimize the loss function, it would be natural to think that the smaller the loss function of the generator, the better the quality of the images produced. However, since the generator is only graded against the current discriminator, we cannot compare the loss function evaluated at different points in the training process, since the discriminator is constantly improving over time. Indeed, in Figure 4-8, the loss function of the generator actually increases over time, even though the quality of the images is clearly improving. This lack of correlation between the generator loss and image quality makes GAN training difficult to monitor and in need of constant supervision.

Hyperparameters

As we have seen, even with very similar GANs, there are a large number of hyperparameters to tune. As well as the overall architecture of both the discriminator and generator, there are the parameters that govern the batch normalization, dropout, learning rate, activation layers, convolutional filters, kernel_sizes, striding, batch size, and latent space size to consider. GANs are highly sensitive to very slight changes in all of these parameters and finding a set of parameters that work is often a case of educated trial and error, rather than following an established set of guidelines.

This is why it is important to understand the inner workings of the GAN and know how to interpret the loss function—so that you can identify sensible adjustments to the hyperparameters that might improve the stability of the model.

Improving the GAN

In recent years, several key advancements have drastically improved the overall stability of GAN models and diminish the likelihood of some of the problems listed above, such as mode collapse.

For the remainder of this chapter we shall explore two such advancements, the Wasserstein GAN (WGAN) and Wasserstein Gradient Penalty GAN (WGAN-GP). Both are only minor adjustments to the framework we have explored thus far. The latter is now considered best practice for training the most sophisticated GANs available today.

Wasserstein GAN

A 2017 paper from Arjovsky, Chintala and Bottou entitled 'Wasserstein GAN'[6] was one of the first big steps towards stablising GAN training. With a few changes, they were able to show how to train GANs that have the following two properites (quoted from the paper):

- a meaningful loss metric that correlates with the generator's convergence and sample quality
- improved stability of the optimization process

Specifically, the paper introduces a new loss function for both the discriminator and generator—the Wasserstein loss. Using this loss function instead of binary cross-entropy results in a more stable convergence of the GAN—the mathematical explanation for this is beyond the scope of this book, but there some excellent resources available online that explain the rationale behind switching to this loss function.

Let's take a look at the definition of the Wasserstein loss.

Wasserstein loss

Let's first remind ourselves of the definition of binary cross-entropy loss that we are currently using to train the GAN:

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

This assumes that the response values are binary (0/1).

The Wasserstein loss requires that we use *1* and *-1* to label real and fake observations, so that we can define the loss function as follows:

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

We also remove the sigmoid activation from the final layer of the discriminator. This means that for real images (*1*) in the training set, the discriminator will strive to output a value that is as large as possible, in order to minimize the Wasserstein loss function. Conversely, for generated images (*-1*), the discriminator will try to output a value that is as small as possible. For this reason, the discriminator in WGANs is usually referred to as a *critic* in the literature, since it is now not outputting a probability, but instead outputting a number that can fall anywhere in the range $-\infty, \infty$.

As before, when training the generator, we use a response value of *1* (real), to encourage it to produce images that fool the discriminator.

(Figure 4-13) compares the loss functions of the GAN and the WGAN. If you are mathematically inclined, take a moment to understand why the functions above are equivalent to the equations below.

Discriminator Loss

GAN $\min_D -\left(\mathbb{E}_{x\sim p_X}[\log D(x)] + \mathbb{E}_{z\sim p_Z}\left[\log\left(1 - D\left(G(z)\right)\right)\right]\right)$

WGAN $\min_D -\left(\mathbb{E}_{x\sim p_X}[D(x)] - \mathbb{E}_{z\sim p_Z}\left[D\left(G(z)\right)\right]\right)$

Generator Loss

GAN $\min_D -\left(\mathbb{E}_{z\sim p_Z}\left[\log D\left(G(z)\right)\right]\right)$

WGAN $\min_D -\left(\mathbb{E}_{z\sim p_Z}\left[D\left(G(z)\right)\right]\right)$

Figure 4-13. Comparison between the GAN and WGAN loss functions

In the code, the Wasserstein loss function can be defined as follows:

Example 4-6.

```
def wasserstein(y_true, y_pred):
    return -K.mean(y_true * y_pred)
```

When we compile the models that train the critic and the generator, we can specific that we want to use the Wasserstein loss instead of the binary crossentropy. We also tend to use smaller learning rates for WGANs.

Example 4-7.

```
critic.compile(
    optimizer= RMSprop(lr=0.00005)
    , loss = wasserstein
)

...

model.compile(
    optimizer= RMSprop(lr=0.00005)
    , loss = wasserstein
)
```

The Lipschitz constraint

It may surprise you that we are now allowing the critic to output any number in the range $-\infty, \infty$, rather than apply a sigmoid function to restrict the output to the usual $0, 1$ range. The Wasserstein loss can therefore be very large, which is unsettling—usually, large numbers in neural networks are to be avoided!

In fact, the authors of the WGAN paper show that for the Wasserstein loss function to work, we also need to place an additional constraint on the critic. Specifically, it is required that the critic is a 1-Lipschitz continuous function. Let's pick this apart to understand what it means in more detail.

We can think of the critic as a function, $f : X \to Y$, that converts an image (domain X) into a prediction (domain Y). We say that this function is 1-Lipschitz, if it satisfies the following inequality for any two input images, x_1 and x_2.

$$\frac{d_Y\left(f\left(x_1\right) - f\left(x_2\right)\right)}{d_X\left(x_1 - x_2\right)} \leq 1$$

Here, $d_X\left(x_1 - x_2\right)$ refers to the pixelwise absolute difference between two images and $d_Y\left(f\left(x_1\right) - f\left(x_2\right)\right)$ is the absolute difference between the critic predictions. Essentially, we

require a limit on the rate at which the predictions of the critic can change between two images (i.e. the absolute value of the gradient must be at most 1 everywhere). We can see this applied to a Lipschitz continuous 1D function in Figure 4-14—at no point does the line enter the cone, wherever you place the cone on the line.

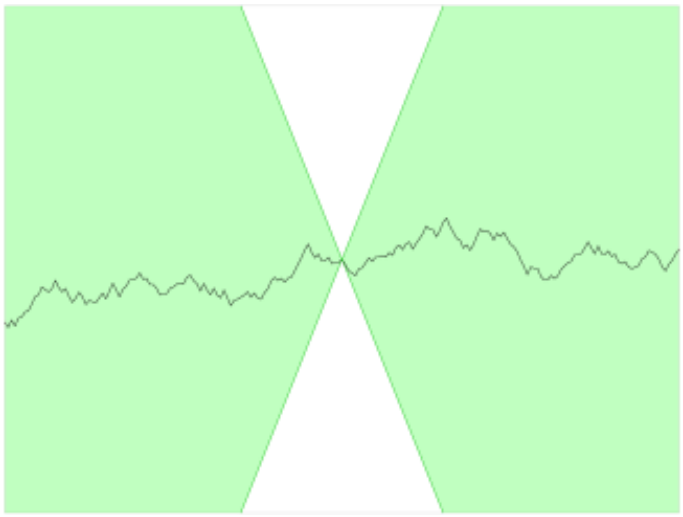


Figure 4-14. A Lipschitz continuous function—there exists a double cone (white) such that wherever it is placed on the line, the function always remains entirely outside the cone.[7]

For those who want to delve deeper into the mathematical rationale behind why the Wasserstein loss only works when this constraint is enforced, there is an excellent explanation available here [8] .

Weight clipping

In the WGAN paper, the authors show how it is possible to enforce the Lipschitz constraint by clipping the weights of the critic to lie within a small range $[-0.01, 0.01$] after each training batch.

We can include this clipping process in our WGAN critic training function as follows:

Example 4-8. Training the critic of the WGAN

```
def train_critic(x_train, batch_size, clip_threshold):

    valid = np.ones((batch_size,1))
    fake = -np.ones((batch_size,1))

    idx = np.random.randint(0, x_train.shape[0], batch_size)
    true_imgs = x_train[idx]

    noise = np.random.normal(0, 1, (batch_size, self.z_dim))
    gen_imgs = self.generator.predict(noise)
```

```
        self.critic.train_on_batch(true_imgs, valid)
        self.critic.train_on_batch(gen_imgs, fake)

        for l in critic.layers:
            weights = l.get_weights()
            weights = [np.clip(w, -clip_threshold, clip_threshold) for w i
n weights]
            l.set_weights(weights)
```

Training the WGAN

When using the Wasserstein loss function, we should train the critic to convergence, in order that the gradients for the generator update are accurate. This is in contrast to a standard GAN, where it is important not to let the discriminator get too strong, to avoid vanishing gradients.

Therefore, using the Wasserstein loss removes one of the key difficulties of training GANs—how to balance the training of the discriminator and generator. With WGANs, we can simply train the critic several times between generator updates, to ensure it is close to convergence. A typical ratio used is 5 critic updates to 1 generator update.

The training loop of the WGAN is coded as follows:

Example 4-9. Training the WGAN

```
for epoch in range(epochs):

    for _ in range(5):
        train_critic(x_train, batch_size = 128, clip_threshold = 0.01)

    train_generator(batch_size)
```

We have now covered all of the key differences between a standard GAN and WGAN. To recap:

- A WGAN uses the Wasserstein loss
- The WGAN is trained using labels of *1* for real and *-1* for fake
- There is no need for the sigmoid activation in the final layer of the WGAN critic
- Clip the weights of the critic after each update
- Train the critic multiple times for each update of the generator

You can train your own WGAN using code from the Jupyter notebook `04_02_wgan_cifar_train.ipynb` in the book repository. This will train a WGAN to generate images of horses from the CIFAR-10 dataset.

Analysis of the WGAN

In Figure 4-15 we show some of the samples generated by the WGAN.

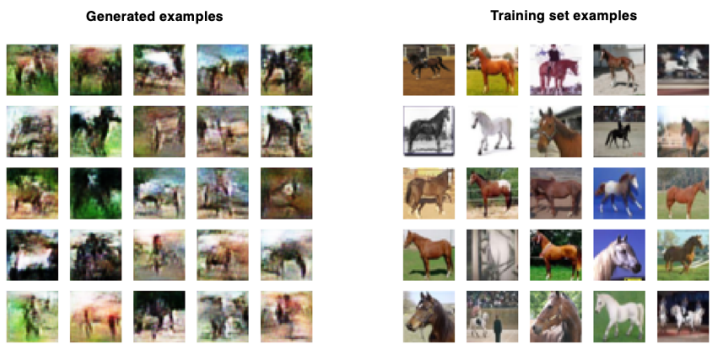


Figure 4-15. Examples from the generator of a WGAN trained on images of horses.

Clearly, this is a much more difficult problem than our previous example. As well as being in color, there are many varying angles, shapes and backgrounds for the GAN to deal with. Therefore whilst the image quality isn't yet perfect, we should be encouraged by the fact that our WGAN is clearly learning the high level features that make up a color photograph of a horse.

One of the main criticisms of WGANs is that since we are clipping the weights in the critic, its capacity to learn is greatly diminished. In fact, even in the original WGAN paper, the authors write 'weight clipping is a clearly terrible way to enforce a Lipschitz constraint'!

A strong critic is pivotal to the success of a WGAN, since without accurate gradients, the generator cannot learn how to adapt its weights to produce better samples.

Therefore, other researchers have looked for alternative ways to enforce the Lipschitz constraint and improve the capacity of the WGAN to learn complex features—we shall explore one such breakthrough in the next section.

WGAN-GP

One of the most recent extensions to the WGAN literature is the Wasserstein GAN—Gradient Penalty framework of Gulrajani et al., in their 2017 paper, 'Improved Training of Wasserstein GANs'[9] .

The WGAN-GP generator is defined and compiled in exactly the same way as the WGAN generator. It is only the definition and compilation of the critic that we need to change.

In the paper, the authors propose an alternative way to enforce the Lipschitz constraint on the critic. Rather than clip the weights of the critic, the authors show how the constraint can be enforced directly, by including a term in the loss function that penalizes the model if the gradient norm of the critic deviates from 1. This is a much more natural way to achieve the constraint and results in a far more stable training process.

In total, there are three changes we need to make to our WGAN critic to convert it to a WGAN-GP critic.

- Include a gradient penalty term in the critic loss function
- Don't clip the weights of the critic
- Don't use batch normalisation layers in the critic

Let's start by seeing how we can build the gradient penalty term into our loss function.

The gradient penalty loss

Figure 4-16 is a diagram of the training process for the critic. If we compare this to the original discriminator training process from Figure 4-7, we can see that the key addition is the gradient penalty term included as part of the loss function, alongside the Wasserstein loss from the real and fake images.

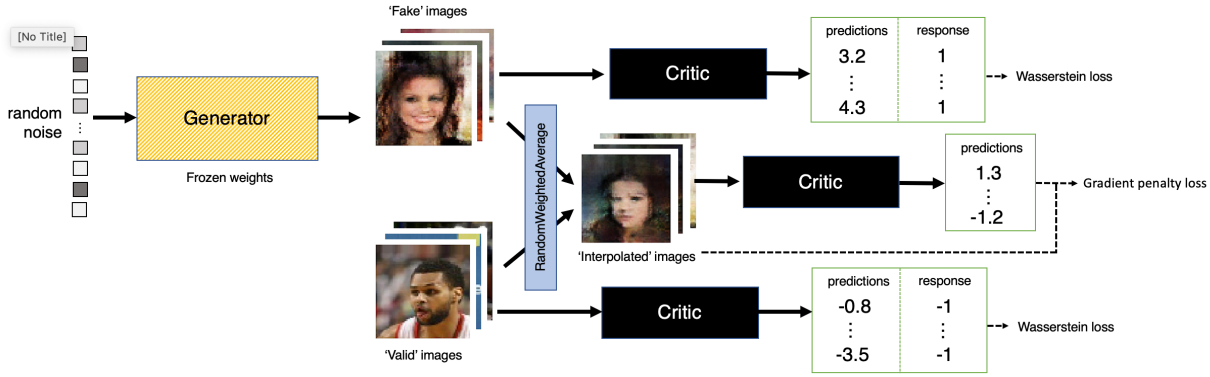


Figure 4-16. The WGAN-GP critic training process

This term measures the absolute difference between the norm of the gradient of the predictions with respect to the input images and 1. The model will naturally be inclined to find weights that ensure the gradient penalty term is minimized, thereby encouraging the model to conform to the Lipschitz constraint.

It is intractable to calculate this gradient everywhere during the training process, so instead the WGAN-GP evaluates the gradient at only a handful of points. To ensure a balanced mix, we use a set of interpolated images that lie at randomly chosen points along lines connecting the batch of real images to the batch of fake images pairwise, as shown in Figure 4-17.

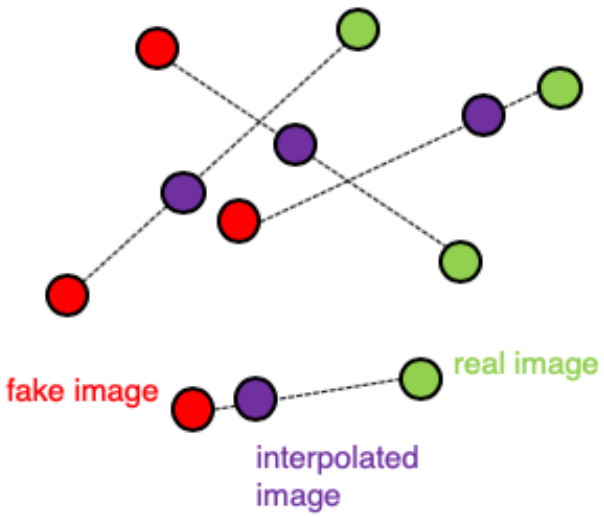


Figure 4-17. Interpolating between images

In Keras, we can create a `RandomWeightedAverage` layer to perform this interpolating operation, by inheriting from the inbuilt `_Merge` layer:

Example 4-10. The RandomWeightedAverage layer

```
class RandomWeightedAverage(_Merge):
    def __init__(self, batch_size):
        super().__init__()
        self.batch_size = batch_size
    def _merge_function(self, inputs):
        alpha = K.random_uniform((self.batch_size, 1, 1, 1)) ❶
        return (alpha * inputs[0]) + ((1 - alpha) * inputs[1]) ❷
```

❶ Each image in the batch gets a random number, between 0 and 1, stored as the vector `alpha`.

❷ The layer returns the set of pixelwise interpolated images that lie along the lines connecting the real images (`inputs[0]`) to the fake images (`inputs[1]`), pairwise, weighted by the `alpha` value for each pair.

The `gradient_penalty_loss` function below returns the squared difference between the gradient calculated at the interpolated points and 1.

Example 4-11. The Gradient penalty loss function

```
def gradient_penalty_loss(y_true, y_pred, interpolated_samples):

    gradients = K.gradients(y_pred, interpolated_samples)[0] ❶

    gradient_l2_norm = K.sqrt(
        K.sum(
            K.square(gradients),
            axis=[1:len(gradients.shape)]
            )
        )
    ) ❷
    gradient_penalty = K.square(1 - gradient_l2_norm)
    return K.mean(gradient_penalty) ❸
```

❶ The Keras `gradients` function calculates the gradients of the predictions for the interpolated images (`y_pred`) with respect to the input (`interpolated_samples`)

❷ We calculate the L2 norm of this vector (i.e. its Euclidean length)

❸ The function returns the squared distance between the L2 norm and 1.

Now that we have the `RandomWeightedAverage` layer that can interpolate between two images and the `gradient_penalty_loss` that can calculate the gradient loss for the interpolated images, we can use both of these in the model compilation of the critic.

In the WGAN example, we compiled the critic directly, to predict if a given image was real or fake. To compile the WGAN-GP critic, we need to use the interpolated images in the loss function—however, Keras only permits a custom loss function with two parameters; the predictions and the true labels. To get around this issue, we use a Python `partial` function.

See below for the full compilation of the WGAN-GP critic in code.

Example 4-12. Compiling the WGAN-GP critic

```
from functools import partial

### COMPILE CRITIC MODEL

self.generator.trainable = False ❶

real_img = Input(shape=self.input_dim) ❷
z_disc = Input(shape=(self.z_dim,))
fake_img = self.generator(z_disc)

fake = self.critic(fake_img) ❸
valid = self.critic(real_img)

interpolated_img = RandomWeightedAverage(self.batch_size)([real_img, f
ake_img]) ❹
validity_interpolated = self.critic(interpolated_img)

partial_gp_loss = partial(self.gradient_penalty_loss, interpolated_sam
ples = interpolated_img) ❺
partial_gp_loss.__name__ = 'gradient_penalty' # Keras requires the fun
ction to be named

self.critic_model = Model(inputs=[real_img, z_disc],
                          outputs=[valid, fake, validity_interpolated]) ❻

self.critic_model.compile(
    loss=[self.wasserstein,self.wasserstein, partial_gp_loss]
    ,optimizer=Adam(lr=self.critic_learning_rate, beta_1=0.5)
    ,loss_weights=[1, 1, self.grad_weight]
    ) ❼
```

❶ Freeze the weights of the generator. The generator forms part of the model that we are using to train the critic, as the interpolated images are now actively involved in the loss function, so we need to freeze its weights.

❷ There are two inputs to the model, a batch of real images and a set of randomly generated numbers that are used to generate a batch of fake images.

❸ The real and fake images are passed through the critic in order to calculate the Wasserstein loss.

❹ The RandomWeightedAverage layer creates the interpolated images, which are then also passed through the critic.

❺ Keras is expecting a loss function with only two inputs—the predictions and true labels, so we define a custom loss function `partial_gp_loss`, using the Python `partial` function to pass the interpolated images through to our `gradient_penalty_loss` function.

❻ The model that trains the critic is defined to have two inputs—the batch of real images and the random input that will generate the batch of fake images. The model has three outputs —*1* for the real images, *-1* for the fake images and a dummy *0* vector, which isn't actually used, but is required by Keras as every loss function must map to an output. Therefore we create the dummy *0* vector to map to the `partial_gp_loss` function.

❼ We compile the critic with three loss functions—two Wasserstein losses for the real and fake images and the gradient penalty loss. The overall loss is the sum of these three losses, with the gradient loss weighted by a factor of 10, in line with the recommendations from the original paper. We use the Adam optimizer, which is generally regarded to be the best optimizer for WGAN-GP models.

Batch normalization in WGAN-GP

One last consideration we should note before building a WGAN-GP is that batch normalization shouldn't be used in the critic. This is because batch normalization creates correlation between images in the same batch, which makes the gradient penalty loss less effective. Experiments have shown that WGAN-GPs can still produce excellent results even without batch normalization in the critic.

Analysis of the WGAN-GP

Running the Jupyter notebook `04_03_wgangp_faces_train.ipynb` in the book repository will train a WGAN-GP model on the CelebA dataset of celebrity faces.

Firstly, let's take a look at some uncurated example outputs from the generator, after 3000 training batches (Figure 4-18):

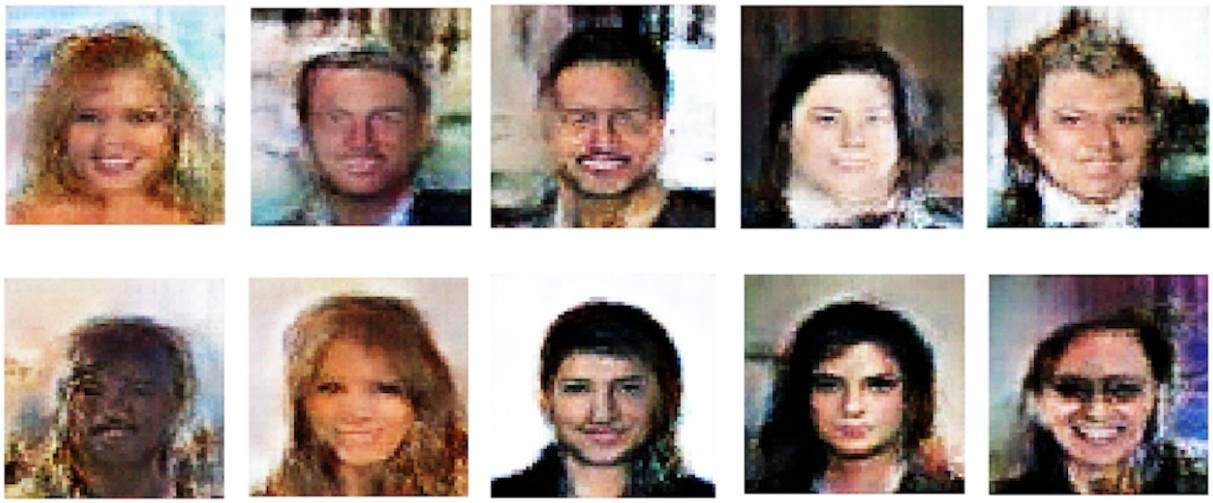


Figure 4-18. WGAN-GP CelebA examples

Clearly the model has learnt the significant high level attributes of a face and there is no sign of mode collapse.

We can also see how the loss functions of the model evolve over time (Figure 4-19)—the loss functions of both the discriminator and generator are highly stable and convergant.

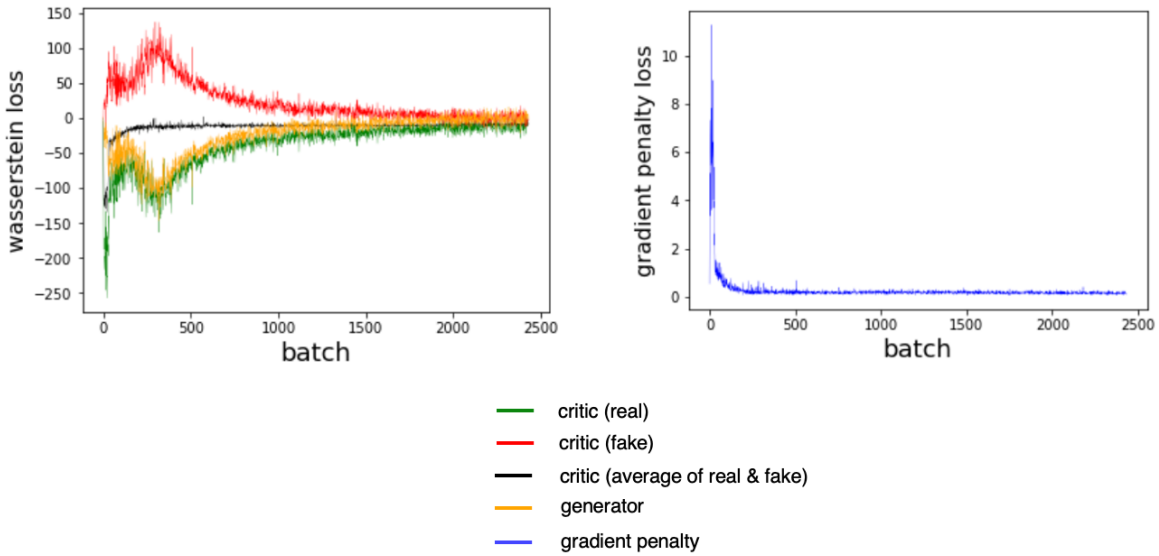


Figure 4-19. WGAN-GP loss

If we compare the WGAN-GP output to the Variational Autoencoder output from the previous chapter we can see that the GAN images are generally sharper—especially the definition between the hair and the background. This is true in general—VAEs tend to produce softer images that blur color boundaries, whereas GANs are known to produce sharper, more well-

defined images.

It is also true that GANs are generally more difficult to train than VAEs and take longer to reach a satisfactory quality. However, most of the state of the art generative models today are GAN based, as the rewards for training large scale GANs on GPUs of a longer period of time are significant.

Summary

In this chapter we have explored three distinct flavors of Generative Adversarial Networks—from the most fundamental vanilla GANs, through the Wasserstein GAN (WGAN), to the current state of the art WGAN-GP models.

All GANS are characterizted by a generator vs discriminator (or critic) architecture, with the discriminator trying to 'spot the difference' between real and fake images and the generator aiming to fool the discriminator. By balancing how these two adversaries are trained, the GAN generator can gradually learn how to produce similar observations to those in the training set.

We saw how vanilla GANs can suffer from several problems including mode collapse and unstable training, and how the Wasserstein loss function remedied many of these problems and made GAN training more predictable and reliable. The natural extension of the WGAN is the WGAN-GP, which places the 1-Lipschitz requirement at the heart of the training process, by including a term in the loss function to pull the gradient norm towards 1.

Lastly, we applied our new technique to the problem of face generation and saw how by simply choosing points from a standard normal distribution, we can generate new faces. This sampling process is very similar to a VAE, though the faces produced by a GAN are quite different—often sharper, with greater distinction between different parts of the image. When trained on a large number of GPUs, this property allows GANs to produce extremely impressive results and has taken the field of generative modeling to ever greater heights.

Overall, we have seen how the GAN framework—a generator and critic, each vying for domination over the other—is extremely flexible and able to be adapted to many interesting problem domains. We'll look at one such application in the next chapter and explore how we can teach machines to paint.

---

1 *https://channel9.msdn.com/Events/Neural-Information-Processing-Systems-Conference/Neural-Information-Processing-Systems-Conference-NIPS-2016/Generative-Adversarial-Networks*

2 *https://arxiv.org/abs/1701.00160v4*

3 *https://console.cloud.google.com/storage/browser/quickdraw_dataset/full/numpy_bitmap*

4 By happy coincidence, ganimals look exactly like camels

5 *https://distill.pub/2016/deconv-checkerboard/*

6 *https://arxiv.org/pdf/1701.07875.pdf*

7 source: *https://en.wikipedia.org/wiki/Lipschitz_continuity*

8 *https://medium.com/@jonathan_hui/gan-wasserstein-gan-wgan-gp-6a1a2aa1b490*

9 *https://arxiv.org/abs/1704.00028*

About the Author

David Foster is a cofounder of Applied Data Science, a data science consultancy delivering bespoke solutions for clients. He holds an MA in Mathematics from Trinity College, Cambridge, UK and an MSc in Operational Research from the University of Warwick.

David has won several international machine learning competitions, including the Innocentive Predicting Product Purchase challenge and was awarded first prize for a visualisation that enables a pharmaceutical company in the US to optimize site selection for clinical trials.

He is an active participant in the online data science community and has authored several successful blog posts on deep reinforcement learning including “How To Build Your Own AlphaZero AI.”