Generative Adversarial Networks

Bálint Gersey

Supervisor: Prof. S. Bacallado

University of Cambridge

This dissertation is submitted for the degree of
Master of Advanced Study in Mathematics

May 2018

I would like to dedicate this thesis to my loving parents Beáta Gerseyné Botka and Zsolt Gersey.

I am extremely grateful to you for giving me the opportunity to study at the University of Cambridge. There are no better words to use today than "thank you," Mom and Dad. You have given me the greatest gift of all: an education, the best legacy that parents can pass on to their child. Thank you so much for believing in me. You are the best parents in the world, and I owe my success to you.

Mom, I feel so honored and blessed to have you as my mom, and want to express my gratitude for your care and support over the years. Thank you for instilling me with a strong passion for learning and for doing everything possible to put me on the path to greatness. You are my best friend, my confidante, but first my momma. You have been there for me through everything that has ever happened in my life.

Declaration

I declare that this essay is work done as part of the Part III Examination. I have read and understood the Statement on Plagiarism for Part III and Graduate Courses issued by the Faculty of Mathematics, and have abided by it. This essay is the result of my own work, and except where explicitly stated otherwise, only includes material undertaken since the publication of the list of essay titles, and includes nothing which was performed in collaboration. No part of this essay has been submitted, or is concurrently being submitted, for any degree, diploma or similar qualication at any university or similar institution.

Bálint Gersey
May 2018

Acknowledgements

I would like to acknowledge the professors of the Statslab from the Department of Pure Mathematics and Mathematical Statistics for the extremely high level education they have provided me during Part III of the Mathematical Tripos at the University of Cambridge. Part III was certainly the most challenging but also the most rewarding year of my studies. I have never had the opportunity to learn so much in one year. Namely, I would like to thank the following professors:

- Michael Tehranchi (Advanced Financial Models)
- Rajen Shah (Modern Statistical Methods)
- Thomas Berrett (Topics in Statistics)
- Tengyao Wang (Statistical Learning in Practice)
- Sergio Bacallado (Bayesian Modelling and Computation)
- Kaisey Mandel (Astrostatistics)
- Marcin Lis (Advanced Probability)
- Roland Bauerschmidt (Stochastic Calculus and Applications)

I would also like to thank once more Professor Sergio Bacallado for supervising this master thesis. Without his helpful guidance this work would have never been possible.

I have to thank Tamás Jakab as well. He was my math teacher in high school and it is thanks to him that I became passionate about mathematics. Tamás Jakab, you are an inspiration to students and other teachers alike. You have activated a hunger for knowledge and wisdom, inspiring me to plan my future and become a better individual. You have been so much more than a teacher for me. You have been my mentor, my support and my guide. If there was a Nobel Prize for best teacher, I am sure you would get it. Thank you for being an amazing person and a true inspiration.

Abstract

This essay (master thesis) is organised as follows.

Chapter 1 introduces the notions of feed-forward, convolutional and recurrent neural networks. This first chapter is optional (not submitted for examination) and readers having had previous exposure to deep learning can feel free to skip it.
Chapter 2 describes different gradient based algorithms and back-propagation. These algorithms are essential to train a neural network.
Chapter 3 is the main chapter of this thesis and is describing Generative Adversarial Networks (GANs). First we explain the intuition behind GANs before proving some theoretical results. We also introduce various versions of GANs like the DCGAN. In Section 3.6 we present some pathological behaviours that occur when training a GAN in practice. Section 3.7 gives some tricks to avoid these behaviours. Finally in Section 3.8 we present the Wasserstein GAN which is using a smooth distance called the Wasserstein distance to measure the similarity between the true distribution and the implicit distribution defined by the generator.
In Chapter 4 (and Appendixes A-C) we reproduce three experiments. We first train a GAN on sinusoidal curves. Then we train another GAN on the MNIST dataset. Finally we also implement a DCGAN and train it on the CIFAR-10 dataset.
In Chapter 5, we propose a new way to generate artificial financial time series using Recurrent Generative Adversarial Networks.

Table of contents

1 Basic concepts of Deep Learning **1**

1.1 Feed-forward networks 4

1.2 Convolutional Neural Networks 7

1.2.1 Convolutional operation 7

1.2.2 How does CNN work? 9

1.3 Sequence Modelling: Recurrent Neural Networks 15

1.3.1 Introduction to RNNs 15

1.3.2 Long-Term-Dependencies 16

1.3.3 Long Short Time Memory cells (LSTMs) 18

2 Updating the weights of a neural network **25**

2.1 Gradient-Based Optimisation 25

2.1.1 Gradient Descent 25

2.1.2 Stochastic Gradient Descent 26

2.1.3 Momentum 30

2.1.4 Algorithms with Adaptive Learning Rates 32

2.1.5 Adam optimiser 33

2.2 Back propagation 35

2.2.1 Practical aspects 38

3 Generative adversarial networks **41**

3.1 Objective of the GAN 41

3.2 How to train a GAN? A game theoretical approach 43

3.3 Theoretical results 44

3.3.1 Optimal discriminator 45

3.3.2 Global optimum for $p_g = p_{data}$ 48

3.3.3 Convergence of Algorithm 12 50

3.4 Heuristic, non saturating game 52

3.5 The DCGAN architecture . . . 54
3.5.1 Transpose convolution . . . 54
3.5.2 Batch normalisation . . . 55
3.5.3 Optimisers . . . 56
3.6 Pathological behaviours in practice . . . 56
3.6.1 Difficulty of finding the Nash equilibrium of game with Gradient Descent . . 56
3.6.2 Low dimensional supports . . . 58
3.6.3 Vanishing gradient . . . 59
3.6.4 Mode collapse . . . 61
3.7 Improved GAN training . . . 63
3.7.1 Training with labels . . . 63
3.7.2 One sided label smoothing . . . 64
3.7.3 Virtual Batch Normalization . . . 65
3.7.4 Feature matching . . . 67
3.7.5 Minibatch Features . . . 67
3.7.6 Adding noise . . . 67
3.7.7 Using better metric to measure the similarity between two distributions . . . 67
3.8 Wasserstein GAN . . . 68
3.8.1 The Wasserstein distance . . . 68
3.8.2 Advantages of the Wasserstein distance . . . 70
3.8.3 Using the Wasserstein distance as GAN loss function . . . 71

4 Experiments 73
4.1 Generating sinusoidal functions . . . 73
4.1.1 Model . . . 73
4.1.2 Results . . . 74
4.2 Training a GAN on the MNIST dataset . . . 76
4.2.1 Model . . . 76
4.2.2 Results . . . 77
4.3 Implementing a DCGAN for image generation . . . 78
4.3.1 Model . . . 79
4.3.2 Results . . . 79

5 Financial time series generation using Recurrent Generative Adversarial Networks 83
5.1 Machine Learning in Finance . . . 83
5.2 Augmenting the financial datasets . . . 84

5.2.1 Training a GAN on financial time series 85
5.3 Training a GAN on financial time series 85
5.3.1 RGAN architecture 86
5.3.2 Experiment 87
5.3.3 Comparing the real and generated samples 88
5.4 Future research 90
5.4.1 Theoretical works 90
5.4.2 Experimental works 92

Appendix A Python tutorials for training a GAN on sinusoidal curves 93

Appendix B Python tutorial for training a GAN on MNIST 99

Appendix C Python tutorial for training a DCGAN on CIFAR-10 105

Appendix D Python tutorial for Monte Carlo simulation 111

Appendix E Tutorial for the implementation of the RGAN 113

References 125

Chapter 1

Basic concepts of Deep Learning

Chapter 1 introduces some basic concepts of deep learning including feed forward, convolutional and recurrent neural networks. The first section about feed forward neural networks is mainly based on the Deep Learning Book [22] and the Statistical Learning in Practice Part III course [6].

Artificial intelligence and in particular deep learning is impacting all industries from healthcare to finance. Deep learning surrounds us everywhere. Voice activated assistants like Apple's Siri or Amazon Echo, self driving cars, Google translate, face recognition on Facebook are just a few of the most well known technologies that depend on deep learning. But what is deep learning?

To understand what deep learning is, we first need to explain the relations between artificial intelligence (AI), machine learning (ML) and deep learning. The connections between these notions is represented in the following Venn diagram.

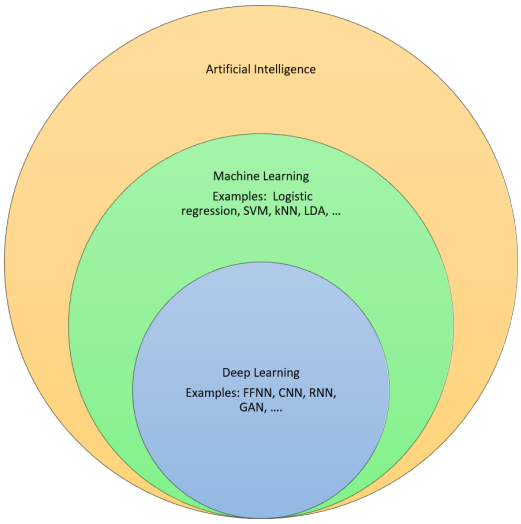


Fig. 1.1 Venn diagram representing the relations between Artificial Intelligence, Machine Learning and Deep Learning

As we can see, AI is a broad class of algorithms that include machine learning. Machine learning is thus a particular approach to achieve artificial intelligence. The difference between a standard AI and a machine learning algorithm is that ML algorithms are able to learn over time without explicitly being programmed. The way ML algorithms work is the following:

1. Train the algorithm on a training set
   For each element of the training set, the algorithm compares its output to the expected output (by calculating a loss/cost function).

2. Updates
   Once the loss function is calculated, the algorithm modifies the way it calculates it's output in order to minimise the loss.

Machine learning is a very broad class of algorithms that include in particular Generalised Linear Models (GLM), Elastic net and other regularised regressions, Linear Discriminant Analysis (LDA), Logistic regression classifier, Support Vector Machines (SVM), k-Nearest Neighbour classifiers (kNN), and many more algorithms.

Deep learning is a particular approach to machine learning. The idea behind deep learning is to learn very complicated mappings between the input and the output space by introducing multiple layers of neural networks. This enables the network to represent the complicated mapping as a series of nested simpler mappings, where each layer of the network corresponds to one of these simpler mappings (see Book [22]).

Deep networks can be represented using graphs, as shown in figure 1.2. The first layer is called the input layer and the last one is the output layer. Intermediate layers are refereed to as hidden layers. The role of hidden layers is to extract features from the previous layer. As we go deeper and deeper into the neural network, the extracted features become more and more abstract as illustrated in figure 1.3.

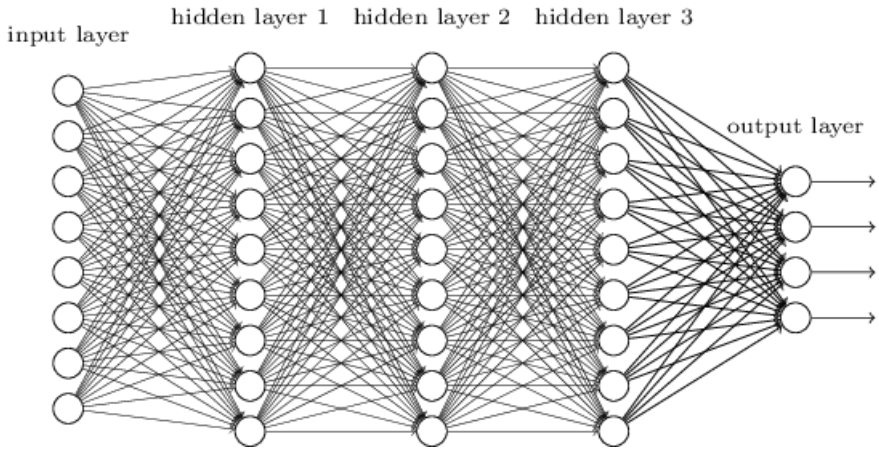


Fig. 1.2 Illustration of a deep learning model (*Source: https://hackernoon.com*)

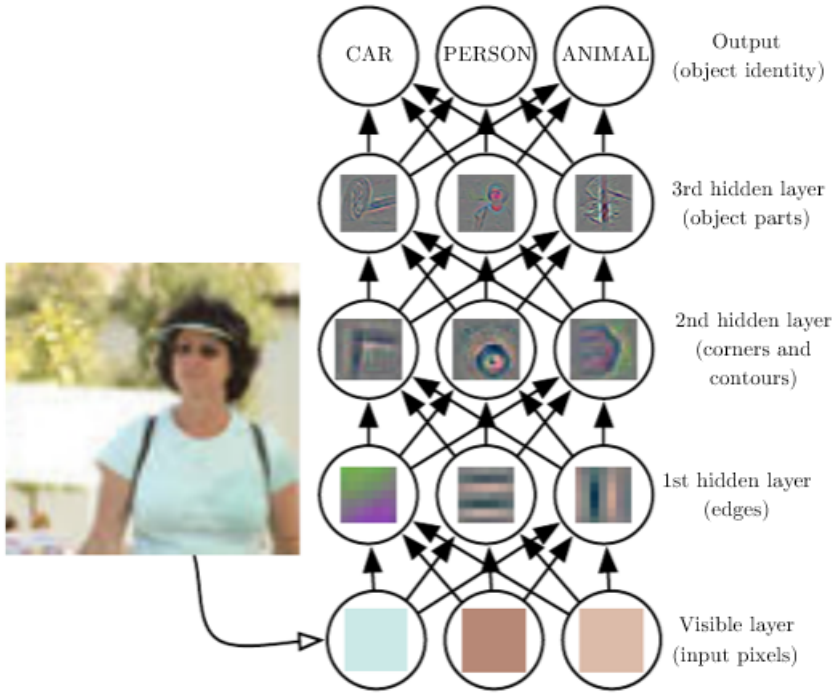


Fig. 1.3 Illustration of the way that layers of a deep learning model extract more and more complicated features as we go deeper in the model (*Source: [22]*)

In the following sections we will briefly discuss three different architectures of neural networks: Feed Forward Neural Networks, Convolutional Neural Networks and Recurrent Neural Networks.

1.1 Feed-forward networks

This section introduces feed-forward neural networks (FFNNs) and is mainly based on the Statistical Learning in Practice course lectured by Prof. Tengyao Wang at the University of Cambridge [6].

Recall that our objective is to learn a potentially very complicated function f^*. In order to achieve this, we define a mapping $y = f(x;\theta)$ depending on some parameters θ representing the weights of the different connections between the different layers of the network. The goal of the FFNN is to learn the optimal parameter value θ^*, for which $f(x;\theta^*) \approx f^*(x)$. To learn these parameters, we will train the FFNN in a forward-backward process as explained below.

First we randomly initialise the parameters θ. We then use the predictors of our training set as inputs, and forward-propagate the information through the network. This gives us predicted outputs that we can compare to the true responses via a loss function. We can then backpropagate the losses to update the weights of the network. By repeating this forward-backward process a large number of times (epochs) we can find the optimal parameters of the artificial neural network. In this section we explain how to forward-propagate information through a FFNN. In Chapter 2, we explain how to update the weights using gradient based optimisation algorithms and backpropagation.

To explain the mathematics behind forward-propagation we have to introduce some notations. Let $x^{(i)} = (x_1^{(i)}, ...x_p^{(i)})^\top$ (i=1,..,n), denote the ith observation. For each $i = 1, ..., n$, we will forward propagate the ith input through the network. This will give us a predicted output $\hat{y}_i$ that we will compare to the true response y_i.
During the forward-propagation, all nodes v in the network will be assigned values x_v. The input nodes receive values from the coordinates of the input vector $x^{(i)} = (x_1^{(i)}, ...x_p^{(i)})^\top$. Note that each observation consists of p values so the input layer of the FFNN will have p neurons; one for each observed value. For a hidden node v, we first calculate the assigned linear score by taking the weighted sum $s_v = \sum_{u\to v} \theta_{u,v} x_u$, where $\theta_{u,v}$ represents the weights of the connections between neurons u and v. The value x_v assigned to node v is then obtained by applying an activation function ϕ to this linear score.

$$x_v = \phi\left(\sum_{u\to v} \theta_{u,v} x_u\right)$$

The five most used activation functions are given in the next table.

Activation functions	
Threshold function	$\phi(x) = \mathbb{1}_{x \geq 0}$
Sigmoid function	$\phi(x) = \frac{1}{1+\exp^{-x}}$
Rectifier Linear Unit (ReLu)	$\phi(x) = max(x, 0)$
Modified Rectifier Linear Unit (here α is a small positive constant)	$\phi(x) = \begin{cases} x & x \geq 0 \\ \alpha x & otherwise \end{cases}$
Hyperbolic tangent function	$\phi(x) = \frac{1-\exp^{-2x}}{1+\exp^{-2x}}$

For instance if the FFNN is trained to classify objects, we can use the rectifier activation function in the hidden layers and the sigmoid function for the output layer. The sigmoid activation forces the outputs to be between 0 and 1 and hence we can interpret the outputs as the probability of the input belonging to a particular class.

Finally, once we have propagated the information through all hidden layers, we need to calculate the values of the output nodes. For an output node v we will calculate again the linear score

$$s_v = \sum_{u \to v} \theta_{u,v} x_u$$

However, we do not necessarily use an activation function for the output layer. Depending on the aim of the FFNN (classification, regression, ...), we use different output rules summarising information across the linear scores of the output nodes $\{s_v : v \in \{\text{output node}\}\}$ to generate a single or a vector output value.

Remark 1. *We can include intercepts terms in our model by adding an extra node with constant value 1 (known as bias node) in each layer as shown in figure 1.4.*

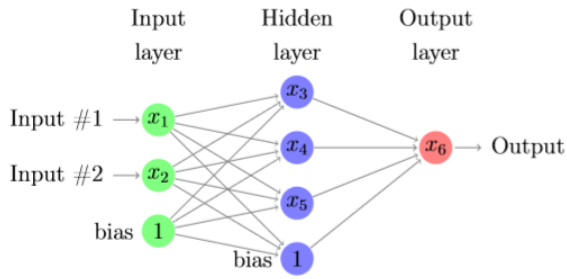


Fig. 1.4 Adding bias nodes to a Feed Forward Neural Network (*Source: Statistical Learning in Practice*

To better understand this process of forward propagation, consider the following basic example. Suppose that our input is a two dimensional vector ($x^{(i)} = (x_1^{(i)}, x_2^{(i)})^\top$) and that the FFNN has only one hidden layer with three neurons (x_3, x_4, x_5) and one output layer with two neurons (x_5, x_6). Suppose also that the FFNN is fully connected, i.e. for each layer, all neurons of that layer are connected to all neurons of the next layer. The FFNN can then be represented by the Directed Acyclic Graph (DAG) shown in figure 1.5

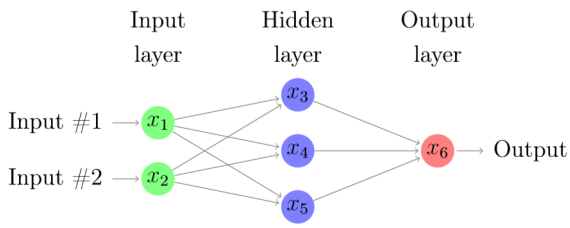


Fig. 1.5 Graphical representation of a Feed Forward Neural Network (*Source: Statistical Learning in Practice [6]*)

Writing f for the activation function, the feed forward process in this example is described by the following equations.

$$\begin{cases} x_3 = f(\theta_{1,3}x_1 + \theta_{2,3}x_2) \\ x_4 = f(\theta_{1,4}x_1 + \theta_{2,4}x_2) \\ x_5 = f(\theta_{1,5}x_1 + \theta_{2,5}x_2) \\ x_6 = f(\theta_{3,6}x_3 + \theta_{4,6}x_4 + \theta_{5,6}x_5) \end{cases}$$

Once the information is forward propagated through the network, we calculate the value of the cost function by using the output and the expected results. Different cost functions can be used. For example we might use the quadratic loss function defined as:

$$l(\mathbf{y}, \hat{\mathbf{y}}) = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

where $\hat{y}_i$ is the output for the ith observation and y_i is the true response for the ith observation.

The aim of the training process is to update the weights θ in order to minimise this cost function. This is done using backpropagation and some gradient based optimisation algorithms like Stochastic Gradient Descent. For more details see Chapter 2.

Finally we repeat this whole procedure a large number a times (epochs) in order to learn the optimal parameters.

1.2 Convolutional Neural Networks

This section introduces a special kind of neural network architecture called Convolutional Neural Networks (CNNs). CNNs play an important role in a different ML research ares like Natural Language Processing.

1.2.1 Convolutional operation

This section introduces the mathematical notion of convolutional operation and is mainly based on Section 9.1 of the Deep Learning book [22].

In 1989, Yann LeCun introduced the concept of Convolutional Neural Networks (CNNs). They are a special kind of neural networks that were designed to work particularly well with data that has a known grid like topology. Examples include time series data (1D grid taking samples at regular time intervals) and image data (2D grip of pixels). As the name convolutional neural networks suggests, CNNs are based on a mathematical operation called convolution.

Definition 1 (Continuous time convolution)**.** *Given two real valued function f and k defined on $\mathbb{R}$, the convolution of f and k, denoted by $f * k$, is the function defined by*

$$(f * k)(t) = h(t) = \int f(y)k(t-y)dy$$

This convolution operation can be interpreted as a weighted average of the $f(y)$ with more weights (given by $k(t-y)$) on the recent values of f. In the context of convolutional neural networks, the function f is called the *input* while k is the *kernel*. The input will correspond to a tensor (multidimensional array) of data and the kernel to a tensor of parameters that we are trying to learn during the training process. The result of the convolutional operation is refereed to as the *feature map*. Usually, in practice we do not work with continuous time data. We thus need to define a discrete time version of our convolutional operation.

Definition 2 (Discrete time convolution)**.** *Given two real valued function f and k defined over the integers, the convolution of f and k, denoted by $f * k$, is the function defined by*

$$(f * k)(t) = h(t) = \sum_{-\infty}^{\infty} f(y)k(t-y)$$

We are often working with higher dimensional inputs (e.g. two dimensional images). We thus need to extend our definitions to higher dimension as well. The next definition defines the convolutional operation between a two dimensional input and a two dimensional kernel. The definition can easily be generalised to higher dimensions (≥ 3) as well.

Definition 3 (Two-dimensional convolution)**.** *The convolution of a two dimensional input I with a two dimensional kernel K is defined as*

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

It is obvious from the definition that the convolutional operation is commutative.

Proposition 1. *Convolution is commutative, i.e,*

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

Proof.

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) = \sum_a \sum_b I(i-a, j-b)K(a,b)$$

where we have used the change of summation $a = i-m$, $b = j-n$. □

Remark 2. *In practice many machine learning libraries actually implement a slightly different "convolution" called the cross-correlation.*

$$\tilde{S}(i,j) = (I \tilde{*} K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)$$

Now that we have defined the convolutional operation, we can define the notion of Convolutional Neural Network.

Definition 4 (Convolutional Neural Network)**.** *A Convolutional Neural Network (CNN) is a neural network that uses convolution instead of a general matrix multiplication in at least one of his layers.*

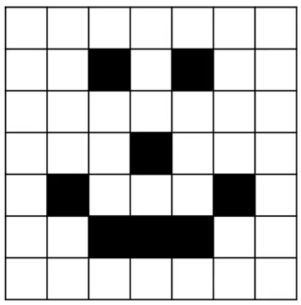
In the next section, we will briefly discuss how to use convolution in a neural network.

1.2.2 How does CNN work?

This section follows closely the Deep Learning and Computer Vision course [3] from Udemy.

Convolution layer

For simplicity, we will explain the concept on a particular example. Suppose that we want to train a network that is able to classify human faces into two classes: happy and sad. For simplicity we will also suppose that we only deal with black and white images so that the smiling face on the left of figure 1.6 can be represented in memory by the matrix on the right.



0	0	0	0	0	0	0
0	1	0	0	0	1	0
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	1	0	0	0	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0

Fig. 1.6 Representation of a smiling face in memory (*Source: Deep Learning and Computer Vision course [3]*)

Consider the following 3x3 kernel:

0	0	1
1	0	0
0	1	1

Fig. 1.7 Kernel matrix (*Source: Deep Learning and Computer Vision course [3]*)

Note that although 3×3 matrices are a common choice for kernels, it is not compulsory for the dimension to be 3×3. For instance, AlexNet uses a 7×7 kernel matrix.

We can interpret the convolutional operation between the input and the kernel as follows. First we overlap the convolution kernel on top of the input image starting in the top left 3×3 submatrix of the input as shown in figure 1.8. We then calculate the element wise product of this submatrix and the kernel matrix. This gives us a new 3×3 matrix. Finally, we add all the coefficients of this new matrix together to get the coefficient $(1,1)$ of our feature map: $0 = 0 \cdot 0 + 0 \cdot 0 + 0 \cdot 1 + 0 \cdot 1 + 1 \cdot 0 + 0 \cdot 0 + 0 \cdot 0 + 0 \cdot 1 + 0 \cdot 1$.

We then move the kernel to the next submatrix of the input and repeat this procedure as shown in figure 1.9.

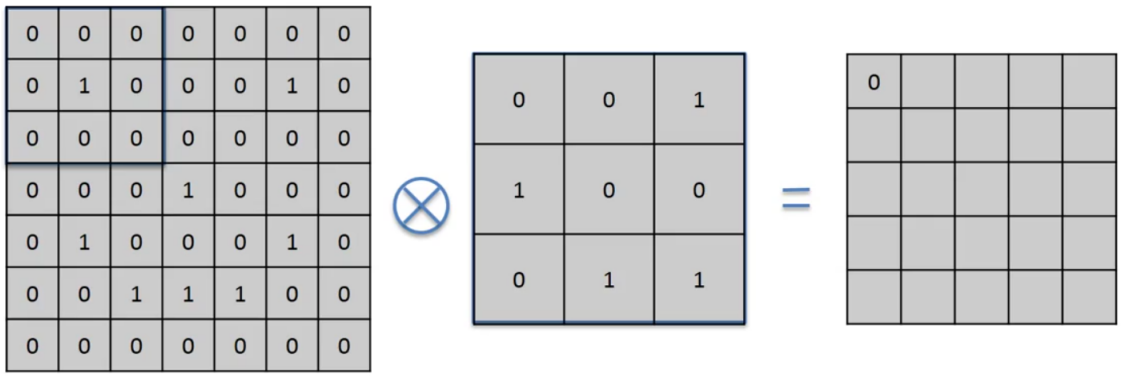


Fig. 1.8 Intuition behind convolution operation (1) (*Source: Deep Learning and Computer Vision course [3]*

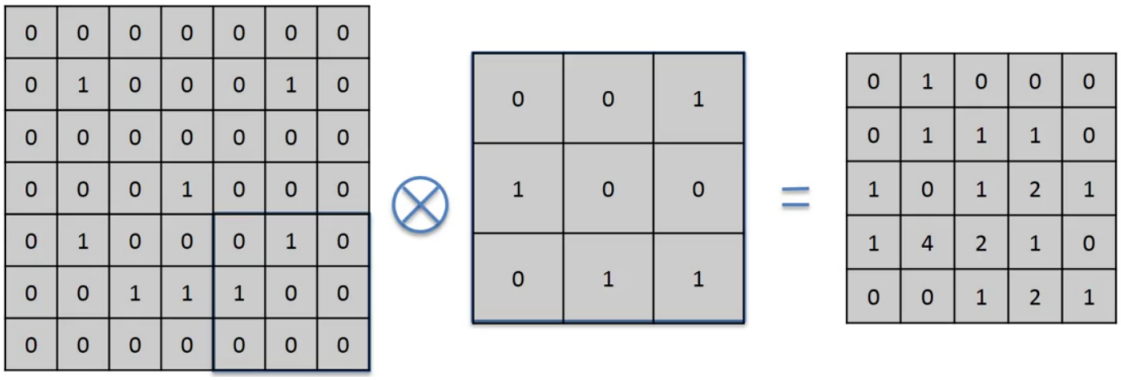


Fig. 1.9 Intuition behind convolution operation (2) (*Source: Deep Learning and Computer Vision course [3]*)

The step at which we move the kernel over the input image is called the stride. In figure 1.9 we have used a stride of 1. A common choice in practice is to use a stride of 2.

Notice that with the convolutional operation, we managed to downsample the input image. The amount of downsampling depends on the stride; the bigger the stride is, the smaller the feature map will be. In practice downsampling is very important since it enables the algorithm to focus only on the main features and process the data faster. The purpose of the kernel is to detect some features in the input. Different kernels can be used to detect different features. In the following figures we represent the effect of different kernels on an input image.

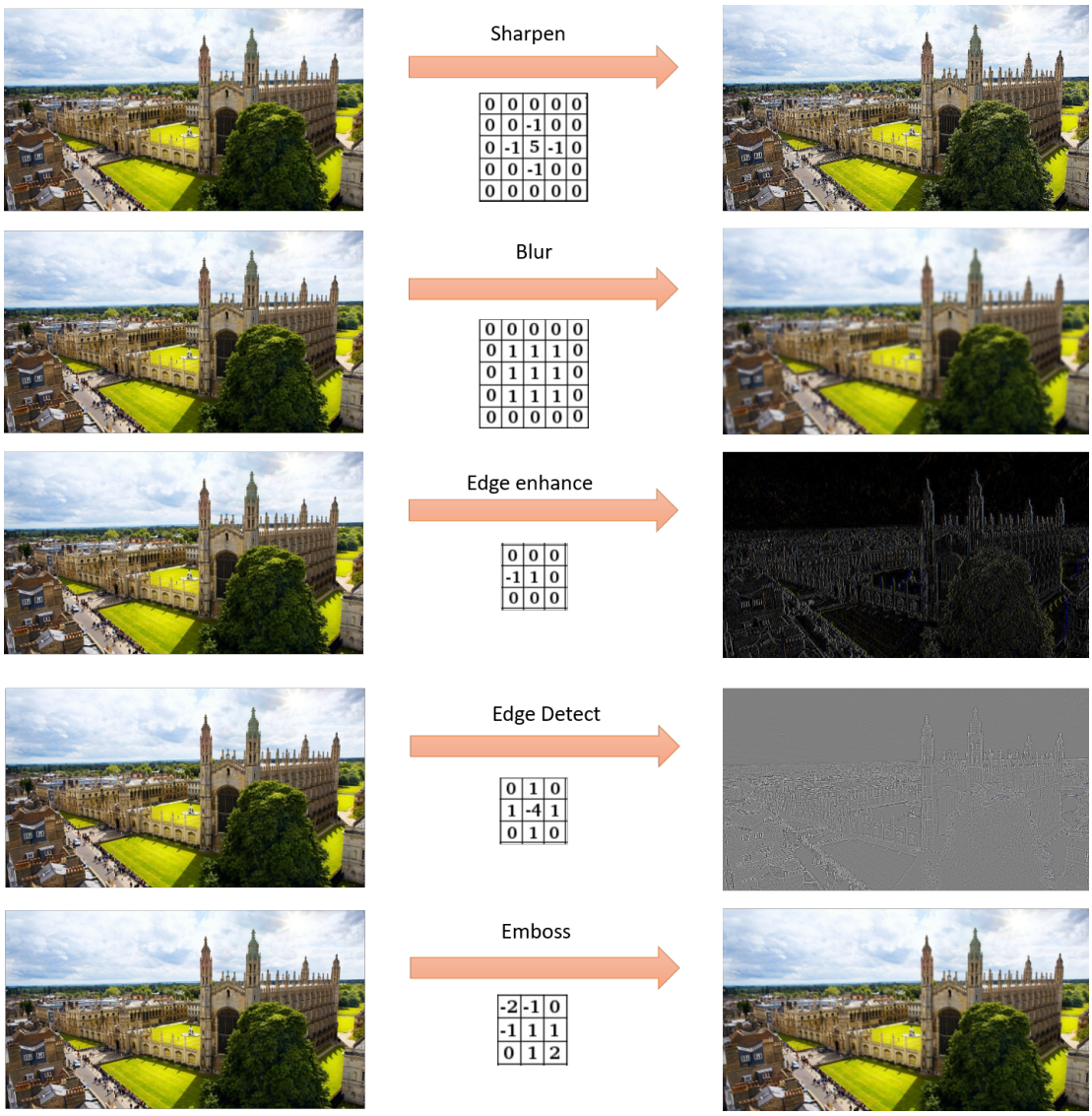


Fig. 1.10 Effects of different kernels on an input image [3]

Instead of just applying one kernel to the input, we will create several feature maps using different kernels. This enables us to detect and preserve the important features of the input image. Moreover, instead of using fixed kernels, we will let the computer decide which features are important by letting the algorithm learn the kernels.

To resume, the main purpose of convolution is to find features in the input using different kernels. This enables to downsample the input and focus only on the important features.
In practice CNN include some additional operations. These are discussed in the next paragraphs.

Non-linear activation function

If we want to use a CNN on non-linear inputs such as images, we have to increase the non linearity of the feature maps. To break linearity, we will apply a non linear activation function, such as the ReLU, on each feature map. This step is know as the detector stage. For more details about why non linear activation functions are important, one might read [30] which is a very good paper summarizing convolutional neural networks .

Pooling layer

Recall that we are interested in detecting whether a feature is present or not in the input, but not so much interested in knowing where exactly the given feature is. This is why we will encourage invariance under local translations by applying a pooling function on the outputs. The role of the pooling function is to replace the feature maps of the CNN by a statistics summary of them. In the case of the MaxPooling introduced in 1988 by Zhou and Chellappa (**reference**), we only report the maximum output value within a rectangular neighbourhood. This is illustrated in the figure 1.10.

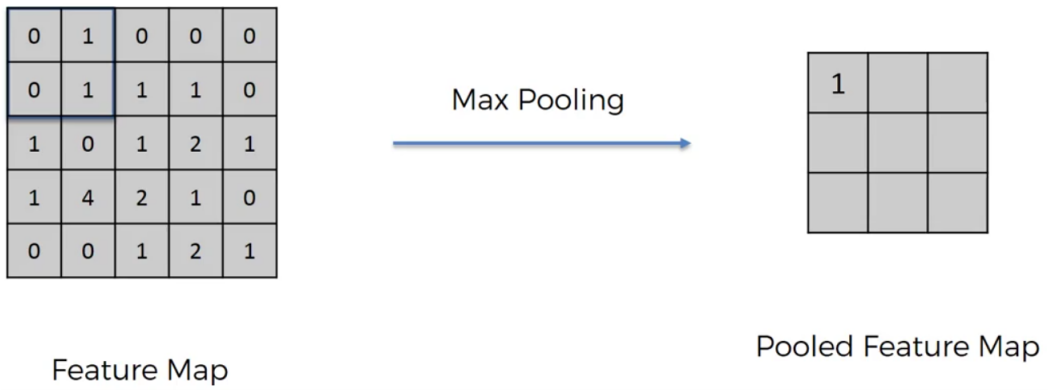


Fig. 1.11 Max Pooling (*Source: Deep Learning and Computer Vision course [3]*)

Other popular pooling functions include the average of a rectangular neighbourhood, the L^2 norm of a rectangular neighbourhood, or a weighted average based on the distance from the central pixel. For more details about pooling, the interested readers can refer to [41].

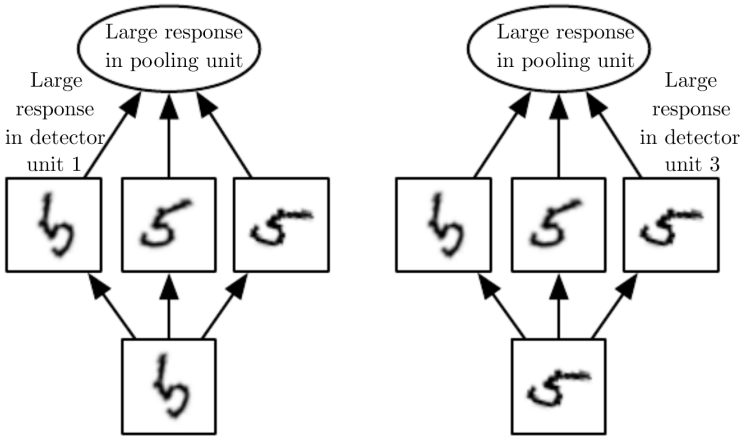


Fig. 1.12 Example of learned invariance (*Source: [22]*)

Note that pooling not only makes the CNN invariant under local translations, but it also reduces the number of parameters that are going to the next layer of the network by summarising the responses over a whole neighbourhood. This improves the computational efficiency of the network because the next layer has fewer inputs to process. Moreover, by reducing the number of parameters, pooling also prevents the network from overfitting to the training set.

Before passing to the next section, let's quickly summaries this section about CNN.
We started with an input to which we applied multiple kernels in order to detect the important features. This gave us several feature maps. Then to break linearity, we applied a non linear activation function (e.g. ReLU). Then from every feature map we created a pooling layer by applying a pooling operation. Finally, in order to pass the output of the CNN as an input to another artificial neural network, we flatten all the pooled outputs into one large column vector. The ANN can then take this vector as input to process all the features detected by the CNN. The whole network (CNN + ANN) is trained together in a forward and back-propagation process. It is very important to note that not only the weights of the ANN are trained, but also the kernels of the CNN are adjusted in the gradient descent process. This enables the CNN to learn the important features of the inputs during the training process. Figure 1.13 summarizes the major ideas behind CNNs.

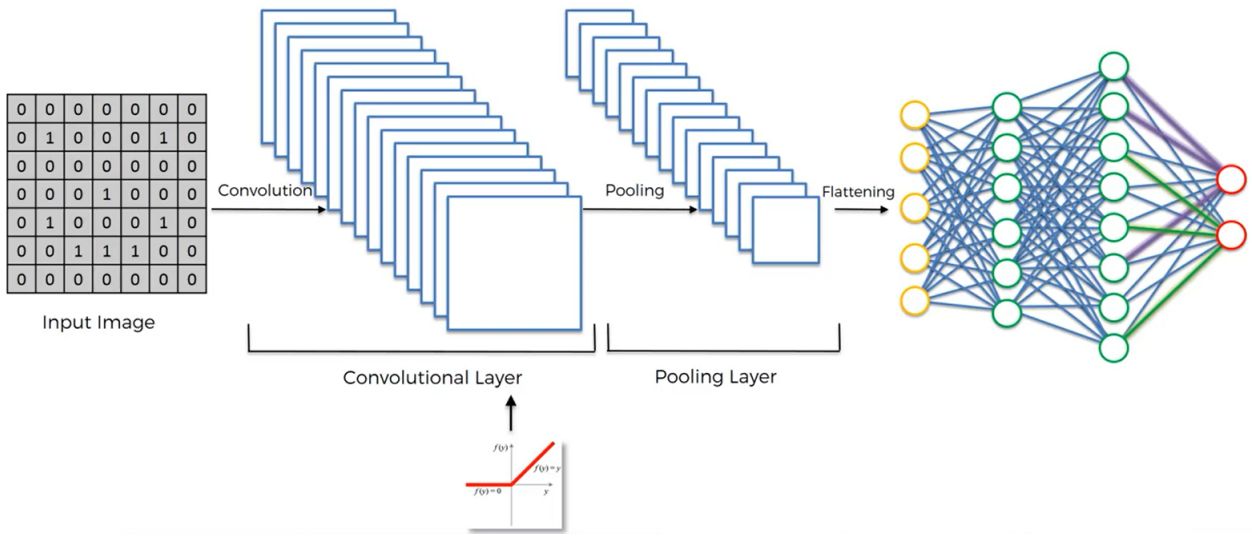


Fig. 1.13 Summary of Convolutional Neural Networks (*Source: Deep Learning and Computer Vision course [3]*)

1.3 Sequence Modelling: Recurrent Neural Networks

Section 1.3.1 is mainly based on the "Deep Learning: Hands-On Artificial Neural Networks" online course [2], but also on on Chapter 10 of book [22]. Section 1.3.3 about LSTMs is inspired by Colah's blog [7] and Chapter 10 of the book [22].

1.3.1 Introduction to RNNs

Recurrent neural networks (RNNs) are a particular class of neural networks designed to process sequential data. In particular RNNs can be applied to analyse time-series data like the log-returns of an asset. In the previous years, a lot of work have been made to compare the efficiency of RNN based models against traditional time series analysis models like ARIMA or GARCH models. Most of the research show that RNNs are outperforming the traditional time series analysis methods. For more details see [45], [33] and [28]. In Chapter 5, we will see how to combine RNNs (in particular the LSTM network) with Generative Adversarial Networks to generate fake, but real looking time series.

To illustrate why we need a new type of network to deal with sequential data, consider the following example. Imagine that we want to train an ANN to classify what is happening at every moment in a movie. Traditional ANNs will fail this task since they do not have access to previous events in the film. However, to understand what is currently happening in the film, it is clear that we not only need to have information about the current state of the film, but we also need to know what happened in the previous scenes (the previous few seconds). RNNs solve this problem by enabling the neurons to be connected to themselves through time.

How do RNNs work?

To explain the way RNNs work, we first need to introduce some notations. For all $t \in \{1,2,..\tau\}$, let

- h_t represent the state of a hidden unit at time t
- x_t be the external signal at time t

With these notations, a RNN can be represented by the following equation:

$$h_t = f(h_{t-1}, x_t; \theta) \tag{1.1}$$

where f is the transition function and θ represents the parameters of the RNN.

Equation 1.1 can be visualised in two different ways. For visualisation purposes, in both representations, we will represent one layer by only one node. Figure 1.14 represents a section A of a neural network. We can see that A receives an input x_t and outputs a value h_t. Then the output h_t is not only passed through the next layer, but is is also used (together with x_{t+1}) as input for A at time

$t+1$. This representation is called a circuit diagram. RNNs can also be represented using unfolded computational graphs as shown in figure 1.15.

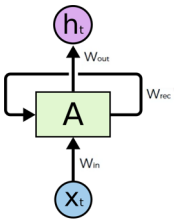


Fig. 1.14 Circuit diagram (*Source: Inspired by [7]*)

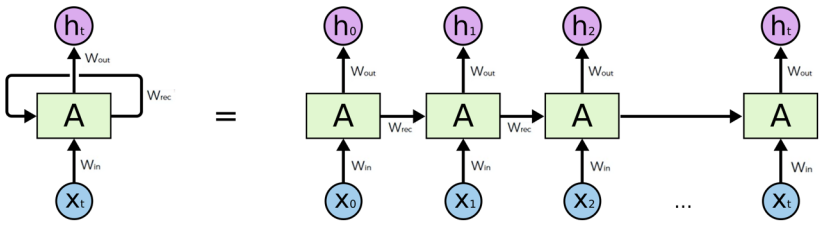


Fig. 1.15 Unrolled Recurrent Neural Network (*Source: Inspired by [7]*)

We can interpret figure 1.15 as showing several copies of the same network through time. Mathematically, we represent the unfolded recurrence after t steps with a function g_t

$$h_t = g_t(x_t, x_{t-1}, x_{t-2}, ..., x_2, x_1) = f(h_{t-1}, x_t; \theta)$$

In the last equality, we used that the unfolded reccurent structure allows us to factorise g_t into repeated applications of a function f. This parameter sharing is very practical since it simplifies considerably the learning process by allowing us to learn only a single model f. Moreover it also allows the network to deal wit sequence lengths that it has not seen during the training process.

1.3.2 Long-Term-Dependencies

In the previous section we have seen that RNNs construct very deep computational graphs by recursively applying the same operation at every time step of a long temporal sequence. This arises the so called vanishing and exploding gradient problem which was first discovered by Sepp Hochreiter in 1991 [25]. To understand this problem, let us represent a RNN using the unfolded computational graph as shown in figure 1.15.

In figure 1.15, the parameters W_{in} and W_{out} correspond to the input and output weights respectively.

The matrix W_{rec} represents the weights used to connect the hidden layers to themselves through time. Recall that every node in the graph actually represents a full layer of neurons.

At each time step $t \in \{1,2,..\tau\}$ we calculate the cost function $\varepsilon_{t-3}, \varepsilon_{t-2}, \varepsilon_{t-1}, ...$ by comparing the output h_t and the expected results. To understand the problem of gradient vanishing/exploding, let's just focus on the cost function at time t. As discussed in the Chapter 1, once we have calculated the loss function ε_t, we have to back propagate the error through the network in order to optimise the parameters. In particular, we need to adapt the weights of all the neurons that participated into the calculation of ε_t. In other words we need to back-propagate the error through time as shown in figure 1.16.

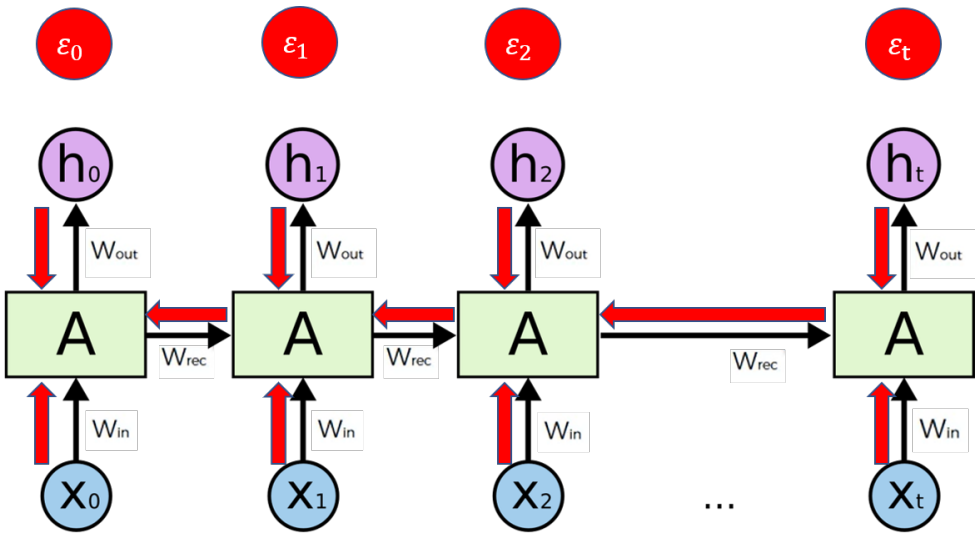


Fig. 1.16 Backpropagation in a Reccurent Neural Network (*Source: Inspired by [7]*)

Intuitively to get from the state of a layer at time t to the state of the same layer at time $t+1$, we multiply the outputs at time t by the matrix W_{rec}. This is not completely true since at each time step we also add some new external input. However for simplicity, we will discard from these external outputs. Multiplying by the same W_{rec} between all time steps is problematic since it means that we will multiply by W_{rec} as many times as we want to see back in the past. Let T be this number. After T steps we have multiplied the outputs of the layer at time $t-T$ by W_{rec}^T to get the output of the layer at time t. Suppose that the matrix W_{rec} has an eigendecomposition $W_{rec} = V diag(\lambda) V^{-1}$. In this case, it is straightforward to calculate W_{rec}^T.

$$\begin{aligned} W_{rec}^T &= \left[V diag(\lambda) V^{-1}\right]^T \\ &= V diag(\lambda)^T V^{-1} \end{aligned}$$

The problem is that any eigenvalue λ_i which is not near in absolute value to 1 will either explode

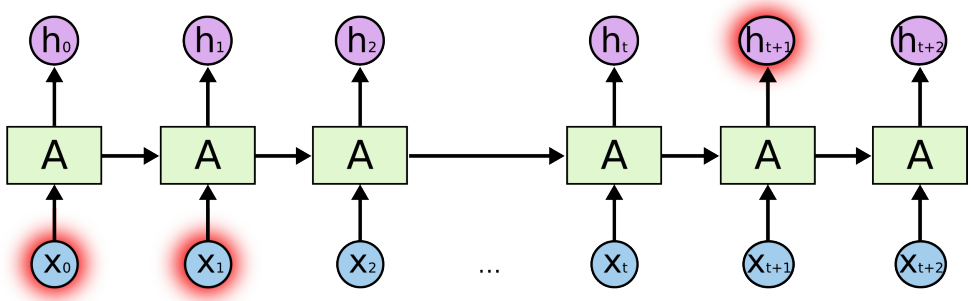
if $\lambda_i > 1$ or vanish if $\lambda_i < 1$. As we will see in Chapter 2, to avoid the problem of saturated nodes, we have to initialise the weights (including W_{rec}) to random numbers close to zero. This means that at the beginning of the training, $|\lambda_i| < 1$ for nearly all indices i. Thus most of the gradients will get smaller and smaller ($\lambda_i < 1$) as they are back-propagated though time. This leads to the vanishing gradient problem. Moreover if $\lambda_i > 1$, the gradients will get larger and larger leading to the exploding gradient problem. Vanishing gradients make the learning of the network impossible for large values of T, while exploding gradients make the learning unstable.

For more details, including mathematical explanations of this effect I recommend the reader to have a look at [25] , [36], [12] and Section 10.2.2 of the Deep Learning book [22].

1.3.3 Long Short Time Memory cells (LSTMs)

In the previous section we have seen that standard RNNs are able to remember information for a short period of time, but they struggle to remember information for over long time periods due to the vanishing gradient problem. Long Short Time Memory cells (LSTMs) were designed to overcome this problem. They have a particular architecture enabling them to avoid the long term dependency problem.

In this section we will explain this special architecture behind LSTMs. To see the difference with standard RNNs, let's quickly recall the structure of the repeating module in the standard RNN.



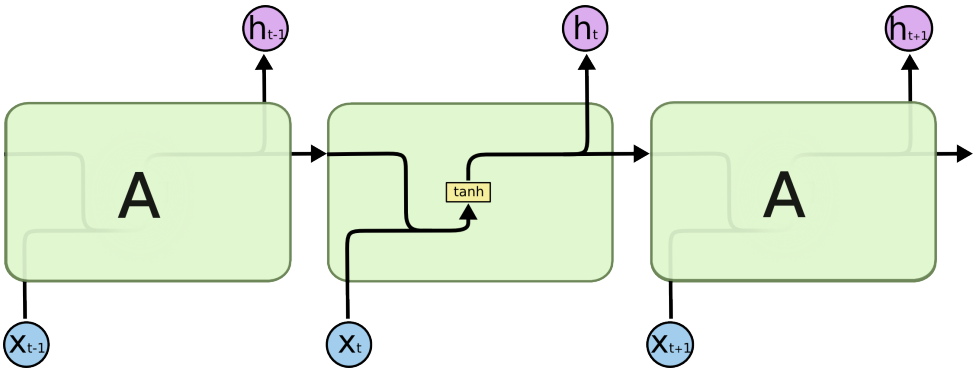


Fig. 1.17 Standard Reccurent Neural Network. We can see that the repeating module in a standard RNN contains a single *tanh* layer (*Source: [7]*)

The vanishing gradient problem lies in the *tanh* layer operation. Indeed, the W_{rec} from the previous section corresponds to weights of this *tanh* layer operation.

In contrast, the LSTM's repeating module has a very different structure. Instead of using just one ANN layer, LSTMs use four layers as shown in figure 1.19.

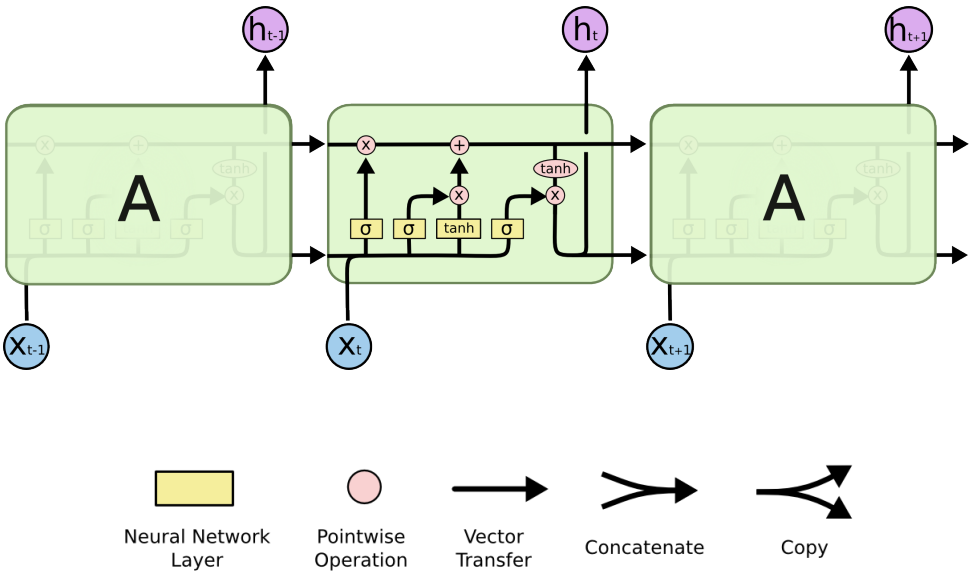


Fig. 1.18 The repeating module in an LSTM contains four interacting layers (*Source: [7]*)

The key part of the LSTM is the horizontal line running through the top of the diagram. It is called the memory cell. Contrarily to the standard RNN architecture (where the memory cell is affected by a *layer* operation), the LSTM's memory cell is only affected by two *pointwise* operations enabling the information to flow very easily along the memory cell. In other words, there is no complex neural network layer operation going on this pipeline. All the complex neural network layer operations are done in the bottom part of figure 1.19. Intuitively, having no neural network layer operations in the

memory cell can be though of taking $\lambda_i = 1 \forall i \in \{1, ..., n\}$. This is not completely true, but it gives the intuition. For a rigorous mathematical description of LSTMs, I recommend the reader the read the original research paper in which LSTMs were introduces [42].

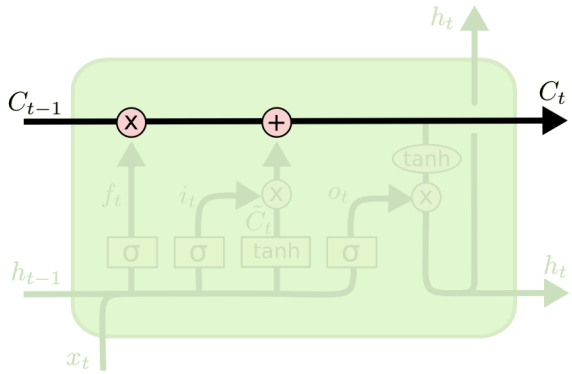


Fig. 1.19 Memory cell of an LSTM (*Source: [7]*)

We have seen that the memory cell does not have layer operations. The LSTM adds/removes information to/from the memory cell using so called **gates**.

Definition 5. *A sigmoid neural network layer followed by a pointwise multiplication operation, as shown in figure 1.20, is called a gate*

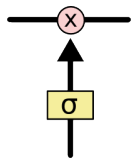


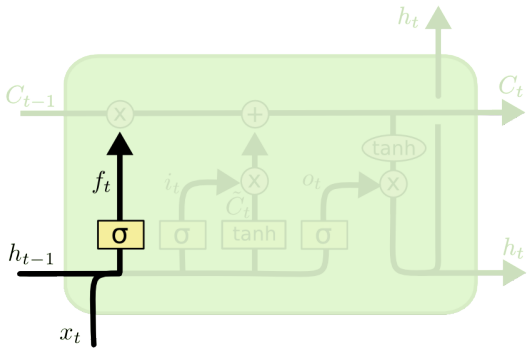
Fig. 1.20 Gate of an LSTM (*Source:[7]*)

The role of gates is to add or remove information to/from the memory cell. Since gates use a sigmoid layer (output between 0 and 1), we can interpret the output of a gate as the proportion of information that should be passed through. An LSTM has three gates. The first (forget gate) is responsible of forgetting data from the memory cell, the second (external input gate) is used to add new information to the memory cell, and the final (output) gate decides what to output.

In the next paragraphs we will discuss each of these gates.

Forget gate layer

We first begin by the forget gate layer, shown in figure 1.21.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

Fig. 1.21 Forget Gate of an LSTM (*Source:[7]*)

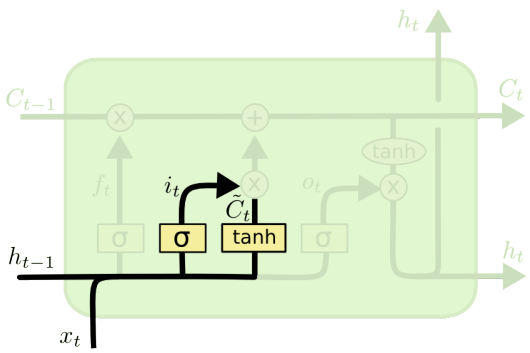
The role of the forget gate is to decide what information to keep in the memory cell. The sigmoid layer at time t takes as input an external input vector x_t and the output h_{t-1} of the LSTM at the previous time step. The output at time t of the sigmoid layer represents the proportion of the information that the memory cell C_{t-1} should keep at time t.
Mathematically we can represent this via the following equation:

$$f_t^i = \sigma\left(b_f^i + \sum_j U_f^{i,j} x_t^j + \sum_j W_f^{i,j} h_{t-1}^j\right) \tag{1.2}$$

where x_t is the input vector at time t, h_{t-1} is the hidden layer vector at time $t-1$ containing the outputs of the LSTM at time $t-1$ and b_f, U_f, W_f are respectively the biases, the input weights and the recurrent weights of the forget gate.

External input gate layer

After deciding what information to keep, the LSTM has to decide what new information to add to the memory cell. This is done by the external input gate in two parts as shown in figure 1.22.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

Fig. 1.22 External Input Gate of an LSTM (*Source: [7]*)

First the sigmoid layer of the input gate (called the input gate layer) decides which values to

update.

$$i_t^j = \sigma\left(b_g^j + \sum_k U_g^{j,k} x_t^k + \sum_k W_g^{j,k} h_{t-1}^k\right) \tag{1.3}$$

where x_t is the input vector at time t, h_{t-1} is the hidden layer vector at time $t-1$ containing the outputs of the LSTM at time $t-1$ and b_g, U_g, W_g are respectively the biases, the input weights and the recurrent weights of the external input gate.
This equation is very similar to the one for the forget gate, but it has his own parameters.

Next, a *tanh* layer creates a vector of new candidate values $\tilde{C}_t$ that could be added to the memory cell (similarly to standard RNN).

$$\tilde{C}_t = \tanh\left(b_C^i + \sum_j U_C^{i,j} x_t^j + \sum_j W_C^{i,j} h_{t-1}^j\right) \tag{1.4}$$

where x_t is the input vector at time t, h_{t-1} is the hidden layer vector at time $t-1$ containing the outputs of the LSTM at time $t-1$ and b_C, U_C, W_C are respectively the biases, the input weights and the recurrent weights of the LSTM cell.

We can now combine the outputs of the forget and input gates to update the old memory cell state C_{t-1} into it's new state C_t. To forget the information we decided to forget, we multiply the old state C_{t-1} by the output of the forget gate f_t. We then add the new information that we have decided to add by adding the output of the input gate i_t multiplied by the candidate value $\tilde{C}_t$.

$$C_t = f_t C_{t-1} + i_t \tilde{C}_t \tag{1.5}$$

i.e.

$$C_t^j = f_t^j C_{t-1}^j + i_t^j \tanh\left(b_C^i + \sum_k U_C^{j,k} x_t^k + \sum_k W_C^{j,k} h_{t-1}^k\right) \tag{1.6}$$

Equation 1.5 can be represented by the following figure.

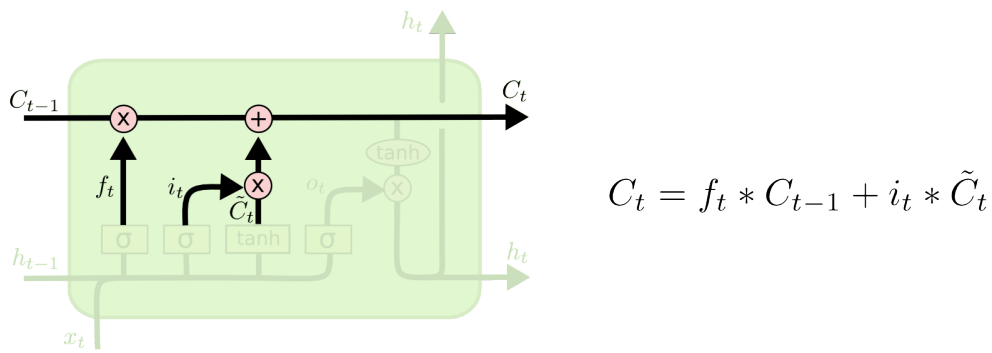
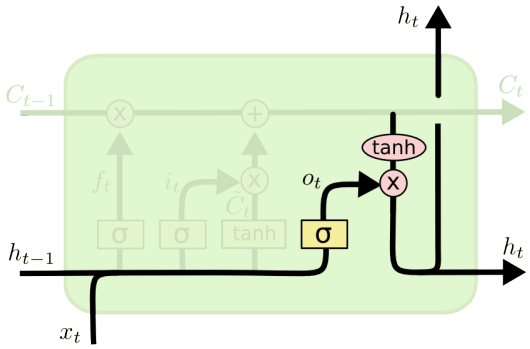


Fig. 1.23 Updating the memory cell (*Source: [7]*)

The output gate

The final step is to decide which part of the memory cell to output. This is done by the sigmoid layer of the output gate as shown in figure 1.24



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

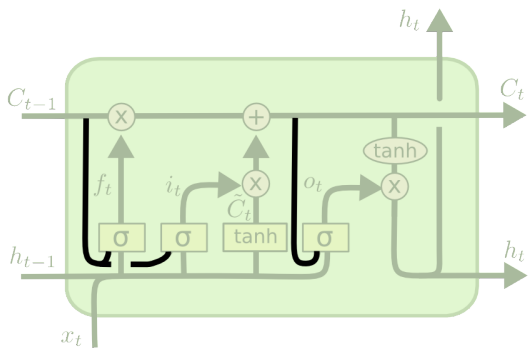
Fig. 1.24 Output Gate of an LSTM (*Source: [7]*)

After passing the inputs x_t and h_{t-1} through a sigmoid layer, we rescale the value of the memory cell to be between -1 and 1 with a pointwise tanh operation. The final output of the LSTM is then obtained by multiplying (pointwise) the information we decided to output by this rescaled value of the memory cell state.

$$\begin{cases} o_t^i = \sigma\left(b_o^i + \sum_j U_o^{i,j} x_t^j + \sum_j W_o^{i,j} h_{t-1}^j\right) \\ h_t^i = o_t^i \tanh(C_t^i) \end{cases}$$

where x_t is the input vector at time t, h_{t-1} is the hidden layer vector at time $t-1$ containing the outputs of the LSTM at time $t-1$ and b_o, U_o, W_o are respectively the biases, the input weights and the recurrent weights of the output gate.

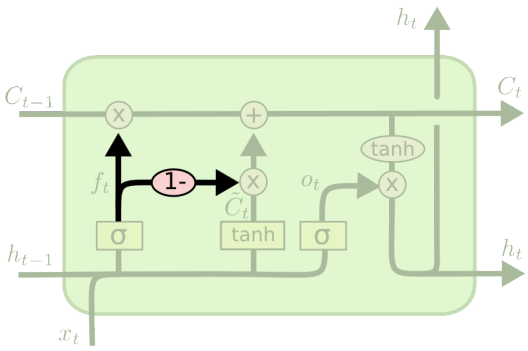
In practice several other variants of LSTMs are used. For example one of the most popular variant of the LSTM cell enables the gate layers to access the memory cell state value by adding new connections. This is illustrated in figure 1.25.



$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t]\ +\ b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t]\ +\ b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t]\ +\ b_o\right)$$

Fig. 1.25 Augmented LSTM (*Source: [7]*)

Another variant of the LSTM couples the forget and input gate into one single gate. The idea is that instead of separatly deciding what to forget from the memory cell and what to add to it, we make these decisions together.



$$C_t = f_t * C_{t-1} + (1 - \boldsymbol{f_t}) * \tilde{C}_t$$

Fig. 1.26 Coupled LSTM (*Source: [7]*)

These are only two of the most popular variants of the LSTM. In practice a lot of other LSTMs are used and I recommend the interested reader to read the research papers [31] and [44](each introducing one variant of the LSTM).

Chapter 2

Updating the weights of a neural network

In the previous chapter we have discussed different architectures of neural networks. In all of these networks we had to update the parameters of the network somehow. In this chapter we discuss how we can backpropagate the errors through the network and how we can minimise the cost function using some gradient based optimisation method.

2.1 Gradient-Based Optimisation

In the following sections, we introduce some of the most used gradient based optimisation algorithms like Gradient Decsent and Stochastic Gradient Descent. Note that these algorithms are called gradient based since they only use the gradient of the loss function to update the parameters. These sections are mainly based on Chapter 4 and 8 of the Deep Learning Book [22].

2.1.1 Gradient Descent

Under some regularity conditions, we may approximate a function f using it's first order Taylor approximation:

$$f(x+\varepsilon) \approx f(x) + \varepsilon \nabla f(x)$$

To find a local minimum of f we can thus descend the function in the direction of it's negative gradient. This is exactly what gradient descend does. In practice however, the true value of the gradient is usually unknown, so we replace it by an estimate of the gradient. A pseudo-code of gradient descent is shown in Algorithm 1.

Figure 2.1 illustrates how gradient descent finds a local minimum of a function.

Algorithm 1 Gradient Descent

Require: Learning rate ε
Require: Initial parameter θ
1: $k \leftarrow 1$
2: **while** stopping criterion not met **do**
3: compute gradient estimate : $\hat{g} \leftarrow \frac{1}{n}\sum_1^n \nabla_\theta L(f(x^{(i)};\theta), y^{(i)})$
4: Apply update: $\theta \leftarrow \theta - \varepsilon\hat{g}$
5: $k \leftarrow k+1$.
6: **end while**

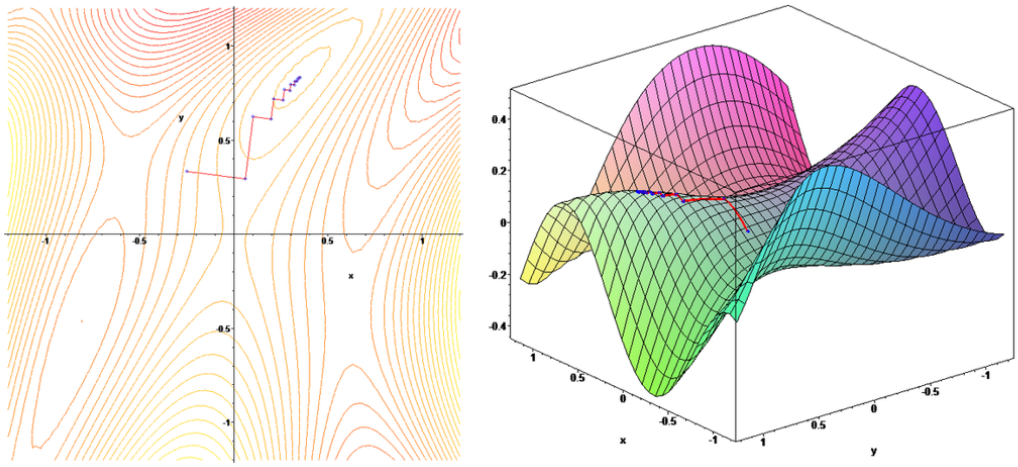


Fig. 2.1 Gradient Descent applied to the function $F(x,y) = \sin\left(\frac{x^2}{2} - \frac{y^2}{4} + 3\right)\cos(2x+1-\exp^{-y})$ (*Source: Wikipedia*)

2.1.2 Stochastic Gradient Descent

As seen in the previous section, (batch/deterministic) gradient descent computed the gradient over the whole training sample. In other words, we took all the rows of our dataset, run the neural network, computed the loss function and then we adjusted the weights using back propagation. This approach can become computationally very expensive for large training sets since computing a sum over n training examples takes $O(n)$ operations. Stochastic gradient descent does not involve computing this sum which allows the algorithm to converge even for very large datasets.

In stochastic gradient descent, we take the rows one by one (selecting them randomly), run the neural network, compute the loss for that row and adjust the weights before moving to the next row. Instead of just choosing one row at a time, we can also choose a minibatch of m examples from the training set for each step of the stochastic gradient descent algorithm.

Stochastic gradient descent replaces thus the gradient by the average gradient over a minibatch

of m examples, which is an unbiased estimate of the true gradient. The pseudo-code of stochastic gradient descent can be thus written as:

Algorithm 2 Stochastic Gradient Descent

Require: Sequence of Learning rate ε_1, ε_2, ...
Require: Initial parameter θ
1: **while** stopping criterion not met **do**
2: Sample a mini-batch of m examples from the training set $x^{(1)},...,x^{(m)})$ with corresponding outputs $y^{(i)}$
3: compute gradient estimate : $\hat{g} \leftarrow \frac{1}{m}\sum_1^m \nabla_\theta L(f(x^{(i)};\theta), y^{(i)})$
4: Apply update: $\theta \leftarrow \theta - \varepsilon_k \hat{g}$
5: **end while**

Remark 3. *In practice, minibatch sizes are driven by the following factors:*

- *Larger batch size provide a more accurate estimate of the gradient but these estimates are computationally more expensive to calculate. Usually the examples in the batch are processed in parallel so the memory size gives an upper bound on the batch size.*
- *For technical reasons, some hardwares and especially GPUs, achieve better runtime with arrays of specific size. It is common to take power of 2 batch sizes when using GPUs to train the network (usually the case).*
- *Stochastic gradient descent adds noise to the learning process by random sampling m training examples. The smaller the batch size m is, the more noise is added to the model. This noise prevents the network from overfitting to the training data. However using too small batch sizes means that at each epoch we will have to evaluate a lot's of gradient estimates which can make the algorithm slow.*

Note that stochastic gradient descent requires a sequence of decreasing learning rates $\varepsilon_1, \varepsilon_2, ...$ This was not the case for gradient descent, where we only used a fixed learning rate ε. Intuitively, this is because, by random sampling m training examples, the stochastic gradient descent introduces a source of noise. The following theorem gives a sufficient condition to guarantee the convergence of stochastic gradient descent.

Theorem 2.1.1. *Let C be an objective function satisfying:*

- *C has a unique minimiser* x^*
- $\forall \varepsilon > 0$, $\inf_{||x-x^*||_2^2 > \varepsilon} \langle x - x^*, \nabla C(x) \rangle > 0$
- $\mathbb{E}\left[||H_n(x)||_2^2\right] \leq A + B||x - x^*||_2^2$ *for some A, B* ≥ 0 *independent of n. Here* $H_n(x)$ *denotes an unbiased estimate of* $\nabla C(x)$.

Suppose that we use an iterative algorithm to find the minimiser x^* using the update rule

$$s_{n+1} = s_n - \varepsilon_n H_n(s_n)$$

where the sequence of learning rates ε_1, ε_2,... satisfies the following two conditions:

- $\sum_{k=1}^{\infty} \varepsilon_k = \infty$
- $\sum_{k=1}^{\infty} \varepsilon_k^2 < \infty$

Then $s_n \to x^*$ in L^2.

Remark 4. *The third condition on the objective function enables us to control the errors induced by the approximation of the gradient $\nabla C(\cdot)$ by an unbiased estimate $H(\cdot)$.*

Proof. The proof comes from a course at the University of Cambridge available at:
http://cbl.eng.cam.ac.uk/pub/Intranet/MLG/ReadingGroup/sgd.pdf
The proof is divided into five steps.

Step 1: Define the Lyapunov process

$$h_n = ||s_n - x^*||_2^2$$

Step 2: Consider the process

$$\begin{aligned} h_{n+1} - h_n &= ||s_{n+1} - x^*||_2^2 - ||s_n - x^*||_2^2 \\ &= ||s_{n+1}||_2^2 + ||x^*||_2^2 - 2\langle s_{n+1}, x^*\rangle - ||s_n||_2^2 - ||x^*||_2^2 + 2\langle s_{n+1}, x^*\rangle \\ &= \varepsilon_n^2 ||H_n(s_n)||_2^2 - 2\varepsilon_n \langle s_n - x^*, H_n(s_n)\rangle \end{aligned}$$

where in the last equality we used that the update rule is $s_{n+1} = s_n - \varepsilon_n H_n(s_n)$ by assumption.

Taking the conditional expectation with respect to the natural filtration generated by the stochastic process s_n, we get

$$\mathbb{E}\left[h_{n+1} - h_n | \mathscr{F}_n\right] = \varepsilon_n^2 \mathbb{E}\left[||H_n(s_n)||_2^2 | \mathscr{F}_n\right] - 2\varepsilon_n \langle s_n - x^*, \nabla C(s_n)\rangle$$

The last equality holds since $H_n(\cdot)$ is assumed to be an unbiased estimate of $\nabla C(\cdot)$.

Step 3: Show that h_n converges almost surely

To show the almost sure convergence of h_n, we will use the quasimartingale convergence theorem, which we recall below:

Theorem 2.1.2 (Quasimartingale convergence (Fisk, 1965))**.** *Let X_n be a positive stochastic process satisfying*

$$\sum_{i=1}^{\infty} \mathbb{E}\left[(\mathbb{E}\left[X_{n+1}|\mathscr{F}_n\right] - X_n)\mathbb{1}_{\mathbb{E}[X_{n+1}|\mathscr{F}_n]-X_n>0}\right] < \infty$$

where $\mathscr{F}_n$ is a given filtration with respect to which X_n is measurable.

Then there exits a random variable $X_\infty \in L_1$ such that $X_n \to X_\infty$ almost surely.

By assumption, $\mathbb{E}\left[||H_n(x)||_2^2\right] \leq A + B||x - x^*||_2^2$ for some A, $B \geq 0$ independent of n. Using this inequality and the result of Step 2, we get

$$\begin{aligned}\mathbb{E}\left[h_{n+1} - h_n|\mathscr{F}_n\right] + 2\varepsilon_n \left\langle s_n - x^*, H_n(s_n)\right\rangle &\leq A\varepsilon_n^2 + B\varepsilon_n^2||s_n - x^*||_2^2 \\ &= A\varepsilon_n^2 + B\varepsilon_n^2 h_n\end{aligned}$$

This is equivalent to

$$\mathbb{E}\left[h_{n+1} - (1 - \varepsilon_n^2 B)h_n|\mathscr{F}_n \leq A\varepsilon_n^2 (2.1)\right.$$

Let $\mu_n = \prod_{i=1}^{n} \frac{1}{1+\varepsilon_i^2 B}$ and consider $\tilde{h}_n = \mu_n h_n$. Equation 2.1 implies that

$$\mathbb{E}\left[\tilde{h}_{n+1} - \tilde{h}_n|\mathscr{F}_n\right] \leq \varepsilon_n^2 \mu_n A$$

Thus

$$\mathbb{E}\left[\tilde{h}_{n+1} - \tilde{h}_n \mathbb{1}_{\mathbb{E}\left[\tilde{h}_{n+1}-\tilde{h}_n|\mathscr{F}_n\right]>0}|\mathscr{F}_n\right] \leq \varepsilon_n^2 \mu_n A$$

which implies that

$$\sum_{n=1}^{\infty} \mathbb{E}\left[\tilde{h}_{n+1} - \tilde{h}_n \mathbb{1}_{\mathbb{E}\left[\tilde{h}_{n+1}-\tilde{h}_n|\mathscr{F}_n\right]>0}|\mathscr{F}_n\right] < \infty$$

since we supposed that $\sum_{n=1}^{\infty} \varepsilon_n^2 < \infty$.

We can thus apply the quasimarintgale convergence theorem to conclude that $(\tilde{h}_n)_{n=1}^{\infty}$ converges almost surely and thus (h_n) also converges almost surely.

Step 4: Show that the limit has to be zero

Recall that

$$\mathbb{E}\left[h_{n+1} - (1 - \varepsilon_n^2 B)h_n|\mathscr{F}_n = -2\varepsilon_n \left\langle s_n - x^*, \nabla C(s_n)\right\rangle + \varepsilon_n^2 A (2.2)\right.$$

Since the sequence $(h_n)_{n=1}^{\infty}$ converges almost surely, the left hand side of equation 2.2 has to be summable. Moreover by assumption $\sum_{n=1}^{\infty} \varepsilon_n^2 < \infty$, so

$$\sum_{n=1}^{\infty} \varepsilon_n \langle s_n - x^*, \nabla C(s_n) \rangle < \infty$$

almost surely. (2.3)

Moreover we have also assumed that $\sum_{n=1}^{\infty} \varepsilon_n = \infty$ which implies together with equation 2.3 that

$$\langle s_n - x^*, \nabla C(s_n) \rangle \to 0$$

almost surely.

We conclude that $(h_n)_{n=1}^{\infty}$ must converge to 0 since we supposed that $\forall \varepsilon > 0$

$$\inf_{||x-x^*||_2^2 > \varepsilon} \langle x - x^*, \nabla C(x) \rangle > 0$$

The stochastic process h_n thus converges to 0 almost surely.

Step 5: Conclude The process $h_n = ||s_n - x^*||_2^2$ converges to 0 almost surely, we conclude that $s_n \to x^*$ in L^2.

□

We thus need to choose not only one but a full sequence of hyperparameters. In practice, we will reduce the numbers of parameters to choose by decreasing the learning rate linearly until a certain iteration τ:

$$\varepsilon_k = (1-\alpha)\varepsilon_0 + \alpha\varepsilon_\tau$$

with $\alpha = \frac{k}{\tau}$. After iteration τ, one can leave ε constant. This reduces considerably the number of hyperparameters to choose. Instead of having to choose an infinite sequence of parameters, we now have to choose only three parameters (ε_0, ε_τ, τ).

In the following sections, we discuss some more advanced algorithms that are based on stochastic gradient descent and widely used in practice.

2.1.3 Momentum

Suppose that the objective is an ill-conditioned quadratic function. Such an objective has the form of a long flat valley leading to the optimum and with steep walls on the sides. In this case the negative of the gradient will point towards the steep sides instead of pointing towards the optimum.

Stochastic gradient descent algorithm will thus oscillate between the walls and hence converge very slowly. The momentum method is designed to solve this problem and accelerate the convergence of stochastic gradient descent. The idea is to accumulate an exponentially decaying moving average of past gradients and move in their direction. The following figure compares stochastic gradient descent with and without momentum.

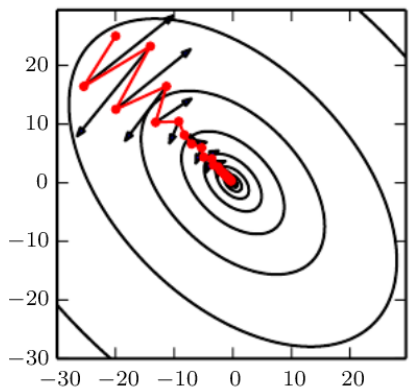


Fig. 2.2 The red path represents the path followed by stochastic gradient descent using Momentum. At each step, the black arrows indicate the step that gradient descent would take at that point (*Source: [22]*)

Mathematically, we introduce a new variable v which represents the velocity and corresponds to an exponentially decaying average of the negative gradient. The velocity v should thus accumulate the gradient elements $\nabla_\theta(\frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};theta),y^{(i)}))$. We also introduce a new hyperparameter $\alpha \in [0,1)$ to control the speed at which the contributions of the previous gradients decay. The update rule can then be written as:

$$\begin{cases} v \leftarrow \alpha v - \varepsilon \nabla_\theta \left[\frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};\theta),y^{(i)})\right] \\ \theta \leftarrow \theta + v \end{cases}$$

The parameter α has to be chosen relative to ε. The larger it is, the more previous gradients affect the current direction. In practice common values for α are $0.5, 0.9$ and 0.99. It is also common to adapt α over the iterations of the algorithm. Typically we chose an increasing sequence of α values.

Putting everything together, we can summarise the stochastic gradient descent with momentum by the following pseudo-code:

Nesterov Momentum

By evaluating the gradient after applying the current velocity, we can add a correction factor to the standard momentum algorithm. The update rules then become:

Algorithm 3 Stochastic Gradient Descent with momentum

Require: Sequence of Learning rate ε, momentum parameter α
Require: Initial parameter θ, initial velocity v
1: **while** stopping criterion not met **do**
2: Sample a mini-batch of m examples from the training set $x^{(1)},...,x^{(m)})$ with corresponding outputs $y^{(i)}$
3: compute gradient estimate : $\hat{g} \leftarrow \frac{1}{m}\sum_1^m \nabla_\theta L(f(x^{(i)};\theta),y^{(i)})$
4: compute velocity: $v \leftarrow \alpha v - \varepsilon\hat{g}$
5: Apply update: $\theta \leftarrow \theta + v$
6: **end while**

$$\begin{cases} v \leftarrow \alpha v - \varepsilon\nabla_\theta\left[\frac{1}{m}\sum_{i=1}^m L(f(x^{(i)};\theta+\alpha v),y^{(i)})\right] \\ \theta \leftarrow \theta + v \end{cases}$$

Stochastic gradient descent using these updates is called stochastic gradient descent with Nesterov momentum. The pseudo-code of this algorithm is given in Algorithm 4.

Algorithm 4 Stochastic Gradient Descent with Nesterov momentum

Require: Sequence of Learning rate ε, momentum parameter α
Require: Initial parameter θ, initial velocity v
1: **while** stopping criterion not met **do**
2: Sample a mini-batch of m examples from the training set $x^{(1)},...,x^{(m)})$ with corresponding outputs $y^{(i)}$
3: Apply interim update $\tilde{\theta} \leftarrow \theta + \alpha v$
4: Compute gradient estimate (at interim point) : $\hat{g} \leftarrow \frac{1}{m}\sum_1^m \nabla_{\tilde{\theta}} L(f(x^{(i)};\tilde{\theta}),y^{(i)})$
5: Compute velocity: $v \leftarrow \alpha v - \varepsilon\hat{g}$
6: Apply update: $\theta \leftarrow \theta + v$
7: **end while**

2.1.4 Algorithms with Adaptive Learning Rates

In this section we present a few algorithms using adaptive learning rates. RMSProp and Adam algorithms are the most used in practice and most of the generative adversarial networks studied in the following chapters are implemented using Adam optimiser.

The main idea behind adaptive learning rate algorithms is to use a separate learning rate for each parameter and adapt these learning rates throughout the learning procedure. This enables the algorithm to learn the directions in parameter space which highly influence the cost function.

The first optimisation algorithm using adaptive learning rates was the delta-bar-delta algorithm introduced by Jacobs in 1988. Delta-bar-delta algorithm increases the learning rate when the partial derivative of the loss, with respect to a given model parameter, remains of the same sign and decreases

the learning rate when the partial derivative changes sign. The main disadvantage of this method is that, by definition, it can only by applied to a full batch optimisation algorithm.

More recently a number of adaptive learning rate algorithms that work with mini-batch based optimisers have been developed. In the next sections we discuss some of these algorithms.

AdaGrad

The AdaGrad algorithm was proposed in 2011 by Duchi et al. To adapt the learning rates individually for all the parameters of the model, the AdaGrad algorithm scales the model parameters inversely proportional to the square root of the sum of all the historical squared values of the gradient as shown in the next algorithm (*Source:* [22]):

Algorithm 5 AdaGrad algorithm

Require: Global Learning rate ε
Require: Initial parameter θ
Require: Small constant δ for numerical stability
1: Initialise gradient accumulation variable $\mathbf{r} = \mathbf{0}$
2: **while** stopping criterion not met **do**
3: Sample a mini-batch of m examples from the training set $\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)}\}$ with corresponding outputs $\mathbf{y}^{(i)}$
4: Compute gradient: $g \leftarrow \frac{1}{m} \nabla_\theta \sum_{i=1}^{m} L(f(\mathbf{x}^{(i)}; \tilde{\theta}), \mathbf{y}^{(i)})$
5: Accumulate squared gradient: $\mathbf{r} \leftarrow$r + g $\odot \mathbf{g}$ ▷ $\odot$ represents the element wise multiplication (Hadamard product)
6: Compute update: $\Delta\theta \leftarrow$ - $\varepsilon \frac{}{\delta + \sqrt{\mathbf{r} \odot \mathbf{g}}}$ ▷ Division and square root is applied element wise
7: Apply update: $\theta \leftarrow \theta + \Delta\theta$
8: **end while**

RMSProp

AdaGrad algorithm was designed to perform well when applied to a convex function. In 2012 Hinton proposed a modification of the AdaGrad algorithm, where he replaced the gradient accumulation by an exponentially weighted moving average. This new algorithm (Algorithm 6 - RMSProp;) converges much faster than the AdaGrad algorithm when applied to non-convex functions (*Source:* [22]).

RMSProp optimiser can also be combined with Nesterov momentum to get the following algorithm (*Source:* [22]).

2.1.5 Adam optimiser

The adaptive moments (Adam) algorithm was discovered in 2014 by Kingma and Ba. Intuitively it can be seen as a combination of RMSProp and momentum.

Algorithm 6 RMSProp algorithm

Require: Global Learning rate ε
Require: Decay rate ρ
Require: Initial parameter θ
Require: Small constant δ for numerical stability
Initialise accumulation variables $\mathbf{r} = \mathbf{0}$
while stopping criterion not met **do**
    Sample a mini-batch of m examples from the training set $\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)}\}$ with corresponding outputs $\mathbf{y}^{(i)}$
    Compute gradient: $g \leftarrow \frac{1}{m} \nabla_\theta \sum_{i=1}^{m} L(f(\mathbf{x}^{(i)}; \tilde{\theta}), \mathbf{y}^{(i)})$
    Accumulate squared gradient: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1-\rho)\mathbf{g} \odot \mathbf{g}$ ▷ $\odot$ represents the element wise multiplication (Hadamard product)
    Compute update: $\Delta\theta \leftarrow$ - $\varepsilon \frac{}{\delta + \sqrt{\mathbf{r}} \odot \mathbf{g}}$ ▷ Division and square root is applied element wise
    Apply update: $\theta \leftarrow \theta + \Delta\theta$
end while

Algorithm 7 RMSProp algorithm with Nesterov momentum

Require: Global Learning rate ε, momentum coefficient α
Require: Decay rate ρ
Require: Initial parameter θ, initial velocity $\mathbf{v}$
Initialise accumulation variable $\mathbf{r} = \mathbf{0}$
while stopping criterion not met **do**
    Sample a mini-batch of m examples from the training set $\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(m)}\}$ with corresponding outputs $\mathbf{y}^{(i)}$
    Compute interim update: $\hat{\theta} \leftarrow \theta + \alpha \mathbf{v}$
    Compute gradient: $g \leftarrow \frac{1}{m} \nabla_{\hat{\theta}} \sum_{i=1}^{m} L(f(\mathbf{x}^{(i)}; \tilde{\hat{\theta}}), \mathbf{y}^{(i)})$
    Accumulate squared gradient: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1-\rho)\mathbf{g} \odot \mathbf{g}$ ▷ $\odot$ represents the element wise multiplication (Hadamard product)
    Compute velocity update: $\mathbf{v} \leftarrow \alpha \mathbf{v} - \frac{\varepsilon}{\sqrt{\mathbf{r}}} \odot \mathbf{g}$ ▷ Division and square root is applied element wise
    Apply update: $\theta \leftarrow \theta + \mathbf{v}$
end while

Algorithm 8 Adam optimiser

Require: Step size ε (Suggested default: 0.001)
Require: Exponential decay rates for momentum estimates ρ_1 and ρ_2 in [0,1) (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ for numerical stability (Suggested default: 10^{-8}
Require: Initial parameter θ

```
Initialise first and second moment variables: s = 0 and r = 0
Initialise time step t = 0
while stopping criterion not met do
    Sample a mini-batch of m examples from the training set {x^(1), ..., x^(m)} with corresponding outputs y^(i)
    Compute gradient: g ← (1/m) ∇_θ Σ_{i=1}^{m} L(f(x^(i); θ̃), y^(i))
    t ← t + 1        Updatebiasedfirstmomentestimate :s←ρ_1 s + (1 − ρ_1) g
    Update biased second moment estimate: r ←ρ_2 r + (1 − ρ_2) g ⊙ g
    Correct bias in first moment: ŝ ←s/(1−ρ_1^t)
    Correct bias in second moment: r̂ ←r/(1−ρ_2^t)
    Compute update: Δθ ← -ε ŝ/(δ+√r̂)        ▷ Division and square root is applied element wise
    Apply update: θ ←θ + Δθ
end while
```

2.2 Back propagation

This section describes the back-propagation algorithm and is mainly based on the Statistical Learning in Practice course [6] lectured by Prof. Tengyao Wang at the University of Cambridge.

In the last section we have presented some gradient based algorithms that we can use to train a neural network. In particular in all of the discussed algorithms we needed to calculate the gradient of the loss function with respect to the weights of the neural network. In this section we show how to take advantage of the composition structure of neural networks to derive these gradients using the chain rule. This technique is called back-propagation.

Before explaining how back-propagation works, lets recall the chain rule of differentiation.

Theorem 2.2.1. *Let* $\mathbf{x} \in \mathbb{R}^m$ *and* $\mathbf{y} \in \mathbb{R}^n$. *Consider the following two functions:*

$$g\colon \mathbb{R}^m \to \mathbb{R}^n$$
$$f\colon \mathbb{R}^n \to \mathbb{R}$$

If $\mathbf{y} = g(\mathbf{x})$ *and* $z = f(\mathbf{y})$, *then*

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

or equivalently using vector notation,

$$\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^{\top} \nabla_{\mathbf{y}} z$$

where $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ *is the* $n \times m$ *Jacobian matrix of* g.

First consider a simple example of a three layered fully-connected feed forward neural network, as shown in figure 2.3.

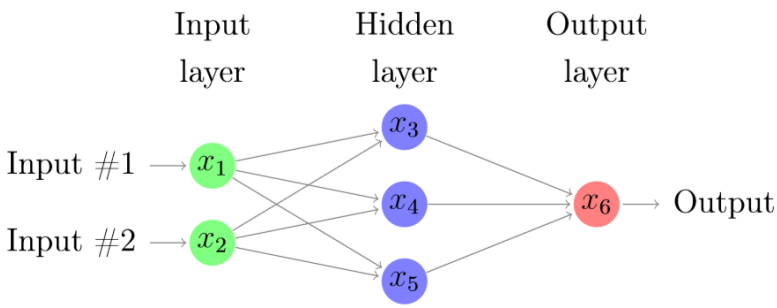


Fig. 2.3 Graphical representation of a Feed Forward Neural Network (*Source: Statistical Learning in Practice*)

Given a training set $\{x^{(i)} = (x_1^{(i)}, x_2^{(i)})^{\top}, i = 1, \ldots, n\}$ and some initial weights $\theta = \{\theta_{u,v} : u \to v, u, v \in \{nodes\}\}$, we can forward propagate the input values to get the output values and the value of the cost function as described in Chapter 1. Recall that the value of node v was calculated as $x_v = f(s_v)$ where f is the activation function and

$$s_v = \sum_{u \to v} \theta_{u,v} x_u$$

is the linear score.

In this simple example, the forward propagation consists of the following steps:

Algorithm 9 Forward propagation in the network shown in figure 2.3

1: $s_3 \leftarrow \theta_{1,3} x_1 + \theta_{2,3} x_2$
2: $x_3 \leftarrow \mathrm{f}(\mathrm{s}_3)$
3: $s_4 \leftarrow \theta_{1,4} x_1 + \theta_{2,4} x_2$
4: $x_4 \leftarrow \mathrm{f}(\mathrm{s}_4)$
5: $s_5 \leftarrow \theta_{1,5} x_1 + \theta_{2,5} x_2$
6: $x_5 \leftarrow \mathrm{f}(\mathrm{s}_5)$
7: $s_6 \leftarrow \theta_{3,6} x_3 + \theta_{4,6} x_4 + \theta_{5,6} x_5$
8: $x_6 \leftarrow \mathrm{f}(\mathrm{s}_6)$

We can always write the loss function l as a sum over i of the contributions of the ith observation:

$$l(\mathbf{y}, \hat{\mathbf{y}}) = \sum_{i=1}^{n} l_i(\hat{y}_i, y_i)$$

Backpropagation enables us to compute the gradients of l_i with respect to the weigh parameters $\theta = \{\theta_{u,v} : u \to v, u, v \in nodes\}$ using the chain rule as shown in Algorithm 10.

Algorithm 10 Example of back-propagation

1: Compute $\frac{\partial l_i}{\partial x_6}$ using the definition of l_i

2: $\frac{\partial l_i}{\partial s_6} = \frac{\partial l_i}{\partial x_6}\frac{\partial x_6}{\partial s_6} = \frac{\partial l_i}{\partial x_6} f'(s_6)$

3: $\frac{\partial l_i}{\partial \theta 3{,}6} = \frac{\partial l_i}{\partial s_6}\frac{\partial s_6}{\partial \theta_{3,6}} = \frac{\partial l_i}{\partial s_6} x_3$

4: $\frac{\partial l_i}{\partial \theta 4{,}6} = \frac{\partial l_i}{\partial s_6}\frac{\partial s_6}{\partial \theta_{4,6}} = \frac{\partial l_i}{\partial s_6} x_4$

5: $\frac{\partial l_i}{\partial \theta 5{,}6} = \frac{\partial l_i}{\partial s_6}\frac{\partial s_6}{\partial \theta_{5,6}} = \frac{\partial l_i}{\partial s_6} x_5$

6: $\frac{\partial l_i}{\partial x_3} = \frac{\partial l_i}{\partial s_6}\frac{\partial s_6}{\partial x_3} = \frac{\partial l_i}{\partial s_6} \theta_{3,6}$

7: $\frac{\partial l_i}{\partial x_4} = \frac{\partial l_i}{\partial s_6}\frac{\partial s_6}{\partial x_4} = \frac{\partial l_i}{\partial s_6} \theta_{4,6}$

8: $\frac{\partial l_i}{\partial x_5} = \frac{\partial l_i}{\partial s_6}\frac{\partial s_6}{\partial x_5} = \frac{\partial l_i}{\partial s_6} \theta_{5,6}$

9: $\frac{\partial l_i}{\partial s_3} = \frac{\partial l_i}{\partial x_3}\frac{\partial x_3}{\partial s_3} = \frac{\partial l_i}{\partial x_3} f'(s_3)$

10: $\frac{\partial l_i}{\partial s_4} = \frac{\partial l_i}{\partial x_4}\frac{\partial x_4}{\partial s_4} = \frac{\partial l_i}{\partial x_4} f'(s_4)$

11: $\frac{\partial l_i}{\partial s_5} = \frac{\partial l_i}{\partial x_5}\frac{\partial x_5}{\partial s_5} = \frac{\partial l_i}{\partial x_5} f'(s_5)$

12: $\frac{\partial l_i}{\partial \theta_{1,3}} = \frac{\partial l_i}{\partial s_3}\frac{\partial s_3}{\partial \theta_{1,3}} = \frac{\partial l_i}{\partial s_3} x_1$

13: $\frac{\partial l_i}{\partial \theta_{2,3}} = \frac{\partial l_i}{\partial s_3}\frac{\partial s_3}{\partial \theta_{2,3}} = \frac{\partial l_i}{\partial s_3} x_2$

14: $\frac{\partial l_i}{\partial \theta_{1,4}} = \frac{\partial l_i}{\partial s_4}\frac{\partial s_4}{\partial \theta_{1,4}} = \frac{\partial l_i}{\partial s_4} x_1$

15: $\frac{\partial l_i}{\partial \theta_{2,4}} = \frac{\partial l_i}{\partial s_4}\frac{\partial s_4}{\partial \theta_{2,4}} = \frac{\partial l_i}{\partial s_4} x_2$

16: $\frac{\partial l_i}{\partial \theta_{1,5}} = \frac{\partial l_i}{\partial s_5}\frac{\partial s_5}{\partial \theta_{1,5}} = \frac{\partial l_i}{\partial s_5} x_1$

17: $\frac{\partial l_i}{\partial \theta_{2,5}} = \frac{\partial l_i}{\partial s_5}\frac{\partial s_5}{\partial \theta_{2,5}} = \frac{\partial l_i}{\partial s_5} x_2$

More generally, if the network has M layers $S_1, S_2, \ldots, S_M$ where S_1 is the input layer, S_M the output

layer and $\{S_m : 2 \leq m \leq M-1\}$ are the hidden layers, then the gradient based algorithms discussed in the previous sections can be implemented using the algorithm 11, known as back-propagation.

Algorithm 11 Back-propagation

Require: Learning rate ε

Require: Initial parameter $\theta^{(0,n)} = \{\theta_{u,v}^{(0,n)} : u \to v, u, v \in nodes\}$

1: **for** t **do** = 1, 2, ... , num_{epochs}

2: **for** i **do** = 1, 2, ..., n

3: Feed-forward the input ith input x^i using the current parameters $\theta^{(t,i-1)}$ (with the convention that $\theta^{(t,0)} = \theta^{(t-1,n)}$). During the forward pass calculate all intermediate quantities, s_v and $x_v$for $v \in S_2 \cup \cdots \cup S_M$

4: Compute the partial derivatives of the loss function with respect to (in this order):

- $\{z_v : v \in S_M\}$, $\{s_v : v \in S_M\}$, $\{\theta_{u,v} : (u,v) \in S_{M-1} \times S_M\}$
- $\{x_v : v \in S_{M-1}\}$, $\{s_v : v \in S_{M-1}\}$, $\{\theta_{u,v} : (u,v) \in S_{M-2} \times S_{M-1}\}$
- ...
- $\{x_v : v \in S_2\}$, $\{s_v : v \in S_2\}$, $\{\theta_{u,v} : (u,v) \in S_1 \times S_2\}$

5: **end for**

6: Use the obtained partial derivatives of the loss function with respect to the weights of the network to update the weights $\{\theta_{u,v}^{(t,i-1)}\}$ using the chosen gradient based optimizer. Note that this step might be in or outside the second for loop depending on which optimizer we use. For example in the case of stochastic gradient descent with a bacth size of 1, the update is inside the for loop and is given by:

$$\theta_{u,v}^{(t,i)} \leftarrow \theta_{u,v}^{(t,i-1)} + \varepsilon_t \frac{\partial l_i}{\partial \theta_{u,v}}$$

where l_i is the contribution to the loss function of the ith observation.

7: **end for**

2.2.1 Practical aspects

Saturation

Suppose that $|s_v| = |\sum_{u \to v} \theta_{u,v} x_u|$ becomes large during the training process for some node v (we say that the node v is saturated). If we use the sigmoid activation function $f(z) = \frac{1}{1+\exp^{-z}}$, then $f'(s_v) \approx 0$. This means that due to the chain rule, all the coefficients $\theta_{u,v}$ for $u \to v$ will be updated very slowly. To avoid saturation at the beginning of the training process, we typically

- standardise inputs so that each covariate has mean zero and unit variance
- choose the initial parameters randomly close to zero (but not equal to zero otherwise the algorithm will not move since the partial derivatives to the left of the null weights will always remain zero)

Vanishing gradient

Rectifier activation function is preferred over the sigmoid activation for deep networks. The reason for this is that the gradient of the sigmoid function is always bounded below by $1/4$ Note that due to the chain rule, the partial derivatives with respect to weights in the earlier layers have many factors of $f'(s_v)$ multiplied together. In the case of the sigmoid activation function this can lead to vanishing gradient for the earlier layer nodes even if no nodes are saturated.

To avoid the problem of vanishing gradient, some newer activation functions have also been proposed like the Leaky Rectified Linear Unit defined as

$$f(z) = \begin{cases} z & \text{if z} > 0 \\ c \cdot z & \text{otherwise} \end{cases}$$

where c is a constant close to zero.

Chapter 3

Generative adversarial networks

"There are many interesting recent development in deep learning. The most important one, in my opinion, is adversarial training (also called GAN for Generative Adversarial Networks). This, and the variations that are now being proposed is the most interesting idea in the last 10 years in ML, in my opinion."

Yann LeCun

After having introduced some important concepts of deep learning such as FFNNs, CNNs and RNNs in Chapter 1 and having discussed some advanced gradient based optimisation methods in Chapter 2, we are finally ready to jump into the hearth of this thesis: Generative Adversarial Networks (GANs). Sections 3.1 to 3.2 are mainly based on the NIPS 2016 paper [21]. Section 3.3 explains the proof of paper [23] in more details. Section 3.4 is based on [21] and Lecture 13 of the CS231 course at Stanford University. Section 3.5 is again mainly based on [21]. Sections 3.6 and 3.7 closely follow ideas from [4] and [21], and Section 3.8 is mainly based on [4].

3.1 Objective of the GAN

This section is mainly based on [21].

We would like to be able to sample from a potentially high-dimensional and complicated distribution p_{data} corresponding to the underlying distribution of the training set. Since this distribution might be very complicated, it is a hard task to sample from it.

Sampling form basic distribution like the uniform or the Gaussian distribution have been studied for a long time and efficient algorithms are already implemented in packages for the most used programming languages (random for Python, <cstdlib> and more recently <random> for C++, ...). For more details about these algorithms the reader can refer to "Random Number Generation and Monte Carlo Methods" by James E.Gentle [16].

The idea behind GANs is to learn a mapping G from these simple distributions to the training/-model distribution p_{data}

$$G\colon \text{latent/noise distributions} \to \text{training distribution}$$
$$z \mapsto G(z)$$

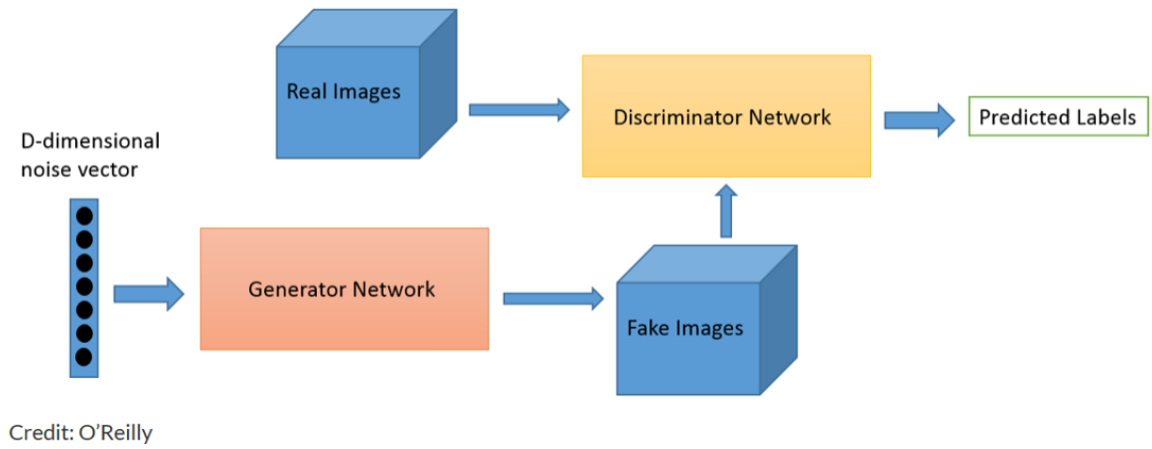


Fig. 3.1 Basic idea behind GANs (*Source: O'Reilly*)

How can we represent and learn that complicated differentiable function G? The next theorem motivates us to use a deep neural network to represent the transformation G.

Theorem 3.1.1 (The Universal Approximation Theorem (Homik et al 1989, Cybenko 1989))**.** *A feed-forward network with a single hidden layer and a "squashing" activation function (e.g. the logistic sigmoid activation function) can approximate any Borel measurable function from one finite dimensional space to another with any non zero amount of error, provided that the network is given enough neurons (hidden units).*

Proof. See Approximation Capabilities of Multilayer Feedforward Networks by Kurt Hornik [26]. □

As a consequence of Theorem 3.1.1 we will use a deep neural network with parameters θ_g to represent the function G. This network will be called the **generator**, and the aim will be to learn the parameters θ_g of the generator network.

3.2 How to train a GAN? A game theoretical approach

In this section, we will explain how to learn the parameters θ_g of the generator network representing the differentiable function G.

As the title of the section suggests, we are going to train the network through a game theoretical approach. First we introduce a second neural network called the **discriminator**. The discriminator will receive as inputs both real samples from the training set, and samples generated by the network. It will be trained to classify the inputs into two classes: real and fake. The output $D(y)$ can thus be interpreted as the probability that y comes from the true training distribution p_{data}.

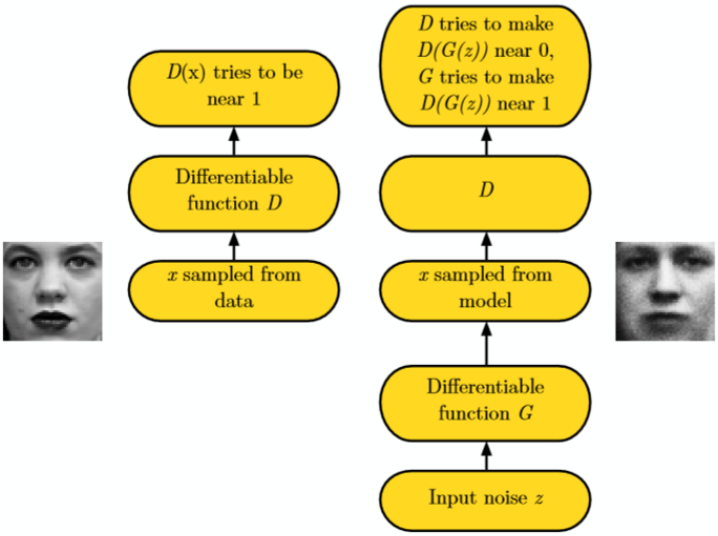


Fig. 3.2 Game theoretical approach to GANs (*Source: [21]*)

In the following discussions we will need some notations. Let

- $p_z(z)$ denote the prior distribution on the input noise variable
- $G_{\theta_g}(z)$ be the output of the generator network for an input $z \sim p_z(z)$. The vector θ_g represents the parameters of the generator network.
- $D_{\theta_d}(x)$ be the output of the discriminator network for an input x. The output $D_{\theta_d}(x)$ should be interpreted as the probability that x was sampled from p_{data} and not from p_g, where p_g is the implicit distribution defined by the generator as the distribution of the samples $G(z)$ for $z \sim p_z(z)$. The vector θ_d represents the parameters of the discriminator network.

Having introduced these notations, we can now explain the objectives of both the generator and the discriminator networks.

The discriminator D is trying to classify between real and fake samples. The discriminator's goal is thus to output $D(x)$ close to 1 for $x \sim p_{data}$ and $D(G(z))$ close to 0 for $z \sim p_z(z)$. We will achieve this by maximising over the parameters θ_d of the discriminator network the following objective function.

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

The generator network is trained to fool the discriminator. Let $z \sim p_z(z)$ and $G(z)$ be the corresponding generated sample. The generator's goal is then to make the discriminator's output for the fake/generated sample $D(G(z))$ close to 1. The parameters θ_g of the generator network will thus be learned by minimising over θ_g the same objective function $V(G,D)$. Since the first term of this objective function does not depend on θ_g, it is equivalent to minimise over θ_g

$$\mathbb{E}_{z_z}\left[\log\left(1 - D_{\theta_d}(G(z))\right)\right]$$

Putting the objective functions for the generator and the discriminator together, we see that the two networks are playing the following two player minimax game:

$$\min_{\theta_g}\max_{\theta_d} \mathbb{E}_{x_{data}}\left[\log D_{\theta_d}(x)\right] + \mathbb{E}_{z_z(z)}\left[\log\left(1 - D_{\theta_d}(G(z))\right)\right] \tag{3.1}$$

With the tools of Chapter 2, we can explain how to implement the training of these networks. Since we are minimising over θ_g and maximising over θ_d, we are going to alternate between gradient descent on the generator and gradient ascent on the discriminator. This leads to the following algorithm.

Remark 5. *The gradient based updates can be done using any of the standard gradient based optimisers presented in Chapter 2. Stochastic Gradient Descent with momentum for the discriminator and Adam optimiser for the generator work usually well in practice.*

3.3 Theoretical results

In this section we will work in a non-parametric setting, e.g we will study the convergence of Algortithm 12 towards p_{data} in the space of probability density functions. In particular this framework implies that we suppose that the networks have infinite capacity.

We will first find the optimal value for the discriminator (for a fixed generator). Then we will show that the minimax game 3.1 has a global optimum for $p_g = p_{data}$. Finally we prove that Algorithm 12 solves the optimisation problem 3.1.

Algorithm 12 Minibatch stochastic gradient ascent/descent training of a GAN. For every training of the generator, we train the discriminator k times, where k is a hyperparameter

1: **for** *number of epochs* **do**

2: **for** *k steps* **do** ▷ update the **discriminator**

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)},\ldots,\mathbf{z}^{(m)}\}$ from noise prior $p_z(\mathbf{z})$.
- Sample minibatch of m samples $\{\mathbf{x}^{(1)},\ldots,\mathbf{x}^{(m)}\}$ from the training data generating distribution p_{data}.
- Update the discriminator by ascending its stochastic gradient computed on these minibatches:

$$\nabla_{\theta_d}\frac{1}{m}\sum_{i=1}^{m}\left[\log D_{\theta_d}(\mathbf{x}^{(\mathbf{i})})+\log\left(1-D_{\theta_d}\left(G_{\theta_g}\left(\mathbf{z}^{(\mathbf{i})}\right)\right)\right)\right]$$

3: **end for**

▷ update the **generator**

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)},\ldots,\mathbf{z}^{(m)}\}$ from noise prior $p_z(\mathbf{z})$.
- Update the generator by descending its stochastic gradient computed on this minibatch:

$$\nabla_{\theta_g}\frac{1}{m}\sum_{i=1}^{m}\log\left(1-D_{\theta d}\left(G_{\theta_g}\left(\mathbf{z}^{(\mathbf{i})}\right)\right)\right)$$

4: **end for**

3.3.1 Optimal discriminator

Theorem 3.3.1. *For a fixed generator G, the optimal discriminator is given by*

$$D_G^*(x)=\frac{p_{data}(x)}{p_{data}(x)+p_g(x)} \tag{3.2}$$

Proof. Given any generator G, the discriminator is trained to maximise the objective function $V(G,D)$.

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

Using the implicit distribution p_g (defined as the distribution of the samples $G(z)$ when $z\sim p_z(z)$) and Random-Nikodym theorem, we can rewrite $V(G,D)$ as follows:

$$V(G,D)=\mathbb{E}_{x\sim p_{data}}\left[\log D_{\theta_d}(x)\right]+\mathbb{E}_{x\sim p_g}\left[\log\left(1-D_{\theta_d}(x)\right)\right]$$

Moreover writing the expectations explicitly, we get

$$V(G,D)=\int_x p_{data}(x)logD_{\theta_d}(x)+p_g(x)log\left(1-D_{\theta_d}(x)\right)dx \tag{3.3}$$

We want to find the value of $D(x)$ that maximises equation 3.3. First note that since we are

integrating over all possible values of x, it is enough to maximise the integrand. To simplify notations consider the change of variables $y = D(x)$, $a = p_{data}(x)$ and $b = p_g(x)$. Then the integrand can be rewritten as

$$p_{data}(x)\log D_{\theta_d}(x) + p_g(x)\log\left(1 - D_{\theta_d}(x)\right)) = a\log(y) + b\log(1-y) \tag{3.4}$$

Claim 1. *For any $(a,b) \in \mathbb{R}^2 \setminus \{(0,0)\}$, the function*

$$\begin{aligned} h\colon [0,1] &\to \mathbb{R} \\ y &\mapsto a\log(y) + b\log(1-y) \end{aligned}$$

achieves its maximum at $\frac{a}{a+b}$

Using the claim we conclude

$$D_G^*(x) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)}$$

It thus remains to show the claim.

The proof of the claim is very easy, we just have to differentiate h and set the derivative equal to 0.

$$\frac{dh}{dy} = \frac{a}{y} - \frac{b}{1-y} = 0 \iff y = \frac{a}{a+b}$$

□

Remark 6. *Note that the whole proof is based on the following equality:*

$$\mathbb{E}_{z\sim p_z(z)}\left[\log\left(1 - D_{\theta_d}(G(z))\right)\right] = \mathbb{E}_{x\sim p_g}\left[\log\left(1 - D_{\theta_d}(x)\right)\right] \tag{3.5}$$

Some people argue incorrectly that this equality follows from the change of variables formula. However this is not correct since to use the change of variables, we would need G to be invertible. This is not assumed and in practice G is usually non-invertible. However equation 3.5 still holds thanks to Randon-Nikodym theorem.

Remark 7. *Estimating the ratio $\frac{p_{data}(x)}{p_{data}(x)+p_g(x)}$ is the key idea behind GANs. Figure 3.3 illustrates how the discriminator network estimates this ratio of densities.*

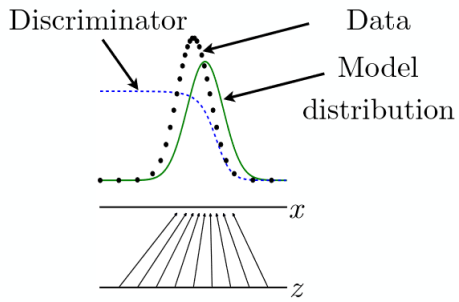


Fig. 3.3 Density ratio estimation (*Source: [23]*)

Generative adversarial networks are trained in a game theoretical approach by simultaneously updating the discriminator's distribution (blue, dashed line) and the generator's distribution p_g (green, solid line). The aim of the discriminator is to classify samples from p_g and from the true distribution p_{data} of the training set (black, dotted line). The black arrows represent the mapping from the latent space, from which z is sampled (in this case uniformly), to the training distribution: $x = G(z)$. Note that this mapping $x = G(z)$ establishes a non-uniform distribution p_g on the x values. More precisely, $p_g(x)$ is greater in regions where the latent z variables are brought together more densely by the mapping G. The discriminator estimates the ratio $\frac{p_{data}(x)}{p_{data}(x)+p_g(x)}$. In regions where the discriminator's distribution is high, the generator's density is too small, and whenever the discriminator's distribution is small, the generator's density is too large. The generator can thus improve himself by moving each value $G(z)$ in the direction that increases $D(G(z))$.

Using the previous proposition, we can reformulate the minimax game 3.1 as $\min_G C(G)$, where $C(G)$ is defined as

$$
\begin{aligned}
C(G) &= \max_D V(G,D) \\
&= V(G,D^*) \\
&= \mathbb{E}_{x\sim p_{data}}\left[\log D_G^*(x)\right] + \mathbb{E}_{z\sim p_z(z)}\left[\log\left(1 - D_G^*(G(z))\right)\right] \\
&= \mathbb{E}_{x\sim p_{data}}\left[\log D_G^*(x)\right] + \mathbb{E}_{z\sim p_z(z)}\left[\log\left(1 - D_G^*(G(z))\right)\right] \\
&= \mathbb{E}_{x\sim p_{data}}\left[\log D_G^*(x)\right] + \mathbb{E}_{x\sim p_g}\left[\log\left(1 - D_G^*(x)\right)\right] \\
&= \mathbb{E}_{x\sim p_{data}}\left[\log\left(\frac{p_{data}(x)}{p_{data}(x)+p_g(x)}\right)\right] + \mathbb{E}_{x\sim p_g}\left[\log\left(1 - \frac{p_{data}(x)}{p_{data}(x)+p_g(x)}\right)\right] \\
&= \mathbb{E}_{x\sim p_{data}}\left[\log\left(\frac{p_{data}(x)}{p_{data}(x)+p_g(x)}\right)\right] + \mathbb{E}_{x\sim p_g}\left[\log\left(\frac{p_g(x)}{p_{data}(x)+p_g(x)}\right)\right]
\end{aligned}
$$

3.3.2 Global optimum for $p_g = p_{data}$

In this section we will prove that the minimax game 3.1 has a global optimum for $p_g = p_{data}$. More precisely, we will prove the following theorem.

Theorem 3.3.2. *The global minimum of the virtual training criterion $C(G)$ is achieved if and only if $p_g = p_{data}$. Moreover at this minimum, $C(G)$ is equal to $-\log 4$*

Before giving the proof of Theorem 3.3.2, we need to recall how to quantify similarities between probability distributions using divergences.

Divergences

Definition 6. *A divergence D on a probability space S is a function $D(\cdot \parallel \cdot) : S \times S \to \mathbb{R}$ satisfying the following properties:*

1. $D(P \parallel Q) \geq 0, \forall P, Q \in S$
2. $D(P \parallel Q) = 0 \iff P = Q$ *almost surely*

Remark 8. *Note that divergences are not true distances since they are not necessarily symmetric.*

In this section of the thesis we will use the Kullback-Leibler (KL) and the Jensen-Shannon (JS) divergences. Later in section 3.8, where we will present the Wasserstein GAN, we will introduce the notion of Wasserstein distance between two distributions.

Definition 7. *The Kullback-Leibler divergence (KL divergence) between two probablility distributions P and Q with densities p and q respectively is defined as*

$$KL(P,Q) = D_{KL}(P \parallel Q) = \int p(x) log\left(\frac{p(x)}{q(x)}\right) dx$$

Definition 8. *The Jensen-Shannon divergence (JS divergence) between two probablility distributions P and Q is defined as*

$$JS(P,Q) = D_{JS}(P \parallel Q) = \frac{1}{2}KL\left(P \parallel \frac{P+Q}{2}\right) + \frac{1}{2}KL\left(Q \parallel \frac{P+Q}{2}\right)$$

Proof of Theorem 3.3.2

We are now ready to prove Theorem 3.3.2.

Proof. We want to prove that the $\min_G C(G)$ is achieved if and only if $p_g = p_{data}$, and that $\min_G C(G) = -\log 4$.

Recall that $C(G)$ was defined as

$$C(G) = \mathbb{E}_{x\sim p_{data}}\left[\log\left(\frac{p_{data}(x)}{p_{data}(x)+p_g(x)}\right)\right] + \mathbb{E}_{x\sim p_g}\left[\log\left(\frac{p_g(x)}{p_{data}(x)+p_g(x)}\right)\right]$$

First note that for $p_g = p_{data}$, we have

$$D_G^*(x) = \frac{p_{data}(x)}{p_{data}(x)+p_g(x)} = \frac{1}{2}$$

Hence for $p_g = p_{data}$,

$$C(G) = \log\frac{1}{2} + \log\frac{1}{2} = -\log 4$$

Now we have to prove that this is the minimal value of $C(G)$ and that it is reached only for $p_g = p_{data}$.

Note that

$$\begin{aligned}
C(G) &= \mathbb{E}_{x\sim p_{data}}\left[\log\left(\frac{p_{data}(x)}{p_{data}(x)+p_g(x)}\right)\right] + \mathbb{E}_{x\sim p_g}\left[\log\left(\frac{p_g(x)}{p_{data}(x)+p_g(x)}\right)\right] \\
&= \mathbb{E}_{x\sim p_{data}}\left[\log\left(\frac{\mathbf{2}p_{data}(x)}{\mathbf{2}(p_{data}(x)+p_g(x))}\right)\right] + \mathbb{E}_{x\sim p_g}\left[\log\left(\frac{\mathbf{2}p_g(x)}{\mathbf{2}(p_{data}(x)+p_g(x))}\right)\right] \\
&= \mathbb{E}_{x\sim p_{data}}\left[\log\left(\frac{2p_{data}(x)}{p_{data}(x)+p_g(x)}\right)\right] + \mathbb{E}_{x\sim p_g}\left[\log\left(\frac{2p_g(x)}{p_{data}(x)+p_g(x)}\right)\right] - \mathbb{E}_{x\sim p_{data}}[\log 2] - \mathbb{E}_{x\sim p_g}[\log 2] \\
&= -\log 4 + \mathbb{E}_{x\sim p_{data}}\left[\log\left(\frac{2p_{data}(x)}{p_{data}(x)+p_g(x)}\right)\right] + \mathbb{E}_{x_g}\left[\log\left(\frac{2p_g(x)}{p_{data}(x)+p_g(x)}\right)\right] \\
&= -\log 4 + KL\left(p_{data} \parallel \frac{p_{data}+p_g}{2}\right) + KL\left(p_g \parallel \frac{p_{data}+p_g}{2}\right) \\
&= -\log 4 + 2JS(p_{data} \parallel p_g)
\end{aligned}$$

Since the Jensen Shannon divergence is a divergence, $JS(p_{data} \parallel p_g) \geq 0$ with equality if and only if $p_{data} = p_g$. We thus conclude that

$$C^* = \min_G C(G) = -\log 4$$

and the minimum is attained if and only if $p_g = p_{data}$ □

Remark 9. *In the previous proof we have shown that*

$$C(G) = V(G, D^*) = -\log 4 + 2JS(p_{data} \parallel p_g)$$

This equation tells us that when there is no capacity limitation for D, and when D is optimal, the loss function of a GAN measures the similarity between the generative data distribution p_g and the true sample distribution p_{data} using the Jensen-Shannon divergence. More precisely, if we define

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

then computing

$$G^* = arg\min_G \max_D V(G,D)$$

amounts to compute

$$G^* = arg\min_G JS(p_{data} \parallel p_g)$$

Remark 10. *Note that although these results provide a nice theoretical framework, in practice the discriminator D is not fully optimal ($D \neq D^*_G$) when optimising the generator G.*

3.3.3 Convergence of Algorithm 12

We conclude this section about theoretical results by proving the convergence of Algorithm 12.

Theorem 3.3.3. *Suppose that both the generator network G and discriminator network D have enough capacity (i.e. the updates of the networks are made in a non-parametric space). Suppose moreover that at each step of Algorithm 12, the discriminator is allowed to reach its optimum D^* given G, and that p_g is updated in a way to improve the criterion*

$$\mathbb{E}_{x\sim p_{data}}\left[\log\left(\frac{p_{data}(x)}{(p_{data}(x)+p_g(x))}\right)\right] + \mathbb{E}_{x\sim p_g}\left[\log\left(\frac{p_g(x)}{(p_{data}(x)+p_g(x))}\right)\right]$$

Then p_g converges in distribution to p_{data}

Proof. It is important to assume that the updates are made directly to p_g in a non-parametrical setup (and not to the parameters θ_g of the neural network representing the generator). Consider the objective function $V(G,D)$ as a function of p_g.

$$\begin{aligned} V(G,D) = U(p_g,D) &= \int_x p_{data}(x) logD(x) + \int_z p_z(z) log(1 - D(G(z)))\,dz \\ &= \int_x p_{data}(x) logD(x) + p_g(x) log(1 - D(x)))\,dx \end{aligned} \tag{3.6}$$

In the last equality we used the definition of p_g and the Radom-Nikodym theorem as in equation 3.5. We first claim that the function $Up_g,D)$ is convex in p_g. Indeed, let $a \in [0,1]$ and p and q be any

probability densities. We have:

$$\begin{aligned}
U(ap+(1-a)q,D) &= \int_x p_{data}(x)logD(x) + [ap(x)+(1-a)q(x)]\,log\,(1-D(x)))\,dx \\
&= \int_x p_{data}(x)logD(x)dx + a\int_x p(x)log\,(1-D(x)))\,dx + (1-a)\int_x q(x)log\,(1-D(x)))\,dx \\
&= a\int_x p_{data}(x)logD(x) + p(x)log\,(1-D(x)))\,dx + (1-a)\int_x p_{data}(x)logD(x) + q(x)log\,(1-D(x \\
&= aU(p,D) + (1-a)U(q,D)
\end{aligned} \tag{3.7}$$

In particular $U(ap+(1-a)q,D) \leq aU(p,D)+(1-a)U(q,D)$ and thus $U(p_g,D)$ is convex in p_g.

Recall that for a convex function $f\colon \mathbb{R}^d \to \mathbb{R}$ we say that a vector $v \in \mathbb{R}^d$ is a subgradient of f at $x \in \mathbb{R}^d$ if

$$f(y) \geq f(x) + v^\top(y-x), \forall y \in \mathbb{R}^d$$

The set of subgradients of f at x is denoted $\partial f(x)$ and is called the subdifferential of f at x.

The proof of Theorem 3.3.3 is based on the following lemma.

Lemma 3.3.4. *Consider $f(x) = sup_{\alpha\in\mathscr{A}} f_\alpha(x)$, where the functions $f_\alpha(x)$ are convex subdifferentiable functions. Suppose the supremum in the definition of $f(x)$ is attained. Let $\beta = argsup_{\alpha\in\mathscr{A}} f_\alpha(x)$. Let g be a subgradient of f_β at x, i.e. $g \in \partial f_\beta(x)$. Then $g \in \partial f(x)$.*

We will apply this lemma with $sup_{\alpha\in\mathscr{A}} f_\alpha(x)$ replaced by $f(p_g) := sup_D U(p_g,D)$. Note that $f(\cdot)$ is convex as a supremum of convex functions. Let $D^* = argsup_D U(p_g,D)$. Lemma 3.3.4 tells us that any subgradient v of $U(p_g,D^*)$ (i.e. $v \in \partial U(p_g,D^*)$ is a subgradient of $f(p_g)$ (i.e. $v \in \partial f(p_g)$). To perform a gradient descent update for p_g at the optimal D^* (given the corresponding G), we need to compute a subgradient v of $U(p_g,D^*)$. By Lemma 3.3.4, this subgradient v is also a subgradient of $f(p_g)$. This means that in the gradient descent updates, we are also minimising $f(p_g) = sup_D U(p_g,D)$. Moreover we have proved in Theorem 3.3.2 that $f(p_g)$ has a unique global optimum given by $p_g = p_{data}$. Therefore, with sufficiently small learning rates, p_g will converge to p_{data}. □

Remark 11. *In practice we update only the parameters θ_g of the generator network $G_{\theta_g}(\mathbf{z})$ and not the implicit density p_g. Thus the updates are made in a parametric space and not in a non-parametric space as in the assumption of Theorem 3.3.3. This means that there is no theoretical guarantee that Algorithm 12 will actually converge in practice. However the Universal Approximation Theorem (Theorem 3.1.1) suggest that it is still reasonable to use Algorithm 12 in practice.*

3.4 Heuristic, non saturating game

Recall that the GAN is solving the following minimax optimisation problem:

$$\min_{\theta_g} \max_{\theta_d} \mathbb{E}_{x\sim p_{data}} [\log D_{\theta_d}(x)] + \mathbb{E}_{z\sim p_z(z)} [\log (1 - D_{\theta_d}(G(z)))]$$

In particular the cost function for the generator is $\log (1 - D_{\theta_d}(G(z)))$. In other words, the generator is minimising the likelihood of the discriminator being correct.

As seen in Section 3.3, this cost function is useful for theoretical analysis, but unfortunately it does not perform well in practice. To understand why, we plotted $\log (1 - D_{\theta_d}(G(z)))$ as a function of $D(G(z))$.

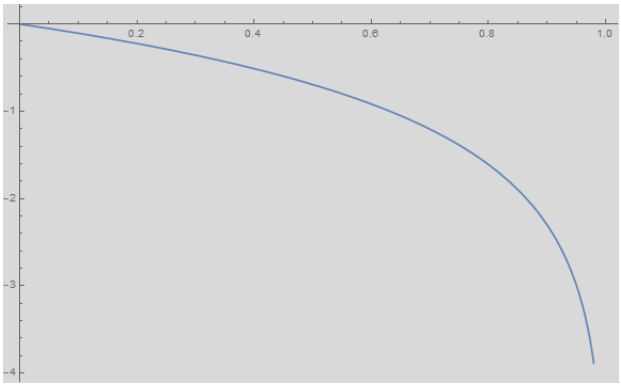


Fig. 3.4 $log\,(1 - D_{\theta_d}(G(z)))$ as a function of $D(G(z))$

Recall that $D(G(z))$ represents the probability that $G(z)$ comes from the true distribution p_{data}. In other words, the generator is doing a good job when $D(G(z))$ is close to one (and a poor job when $D(G(z))$ is close to zero). We can see from figure 3.4 that the slope of $\log (1 - D_{\theta_d}(G(z)))$ is high when $D(G(z))$ is close to one. This means that we are going to have a higher gradient when the generator is already good at fooling the discriminator. On the other hand, when $D(G(z))$ is close to zero, i.e when the generator still outputs bad samples, the gradient of $\log (1 - D_{\theta_d}(G(z)))$ is flat.

This is exactly the opposite of what we would like, since it would be great if the gradient was high when the generator has still a lot to learn.

To solve this problem, one approach is to continue to use the original cost function for the discriminator, but to introduce a slightly different objective function for the generator. **Instead of minimising the likelihood of our discriminator being correct, which is what we did earlier, we are going to maximise the likelihood of the discriminator being wrong.** The new objective for the generator then becomes:

$$\max_{\theta_g} \mathbb{E}_{z\sim p_z(z)} [\log (D_{\theta_d}(G(z)))]$$

Let's plot this new objective function and compare it to the original one.

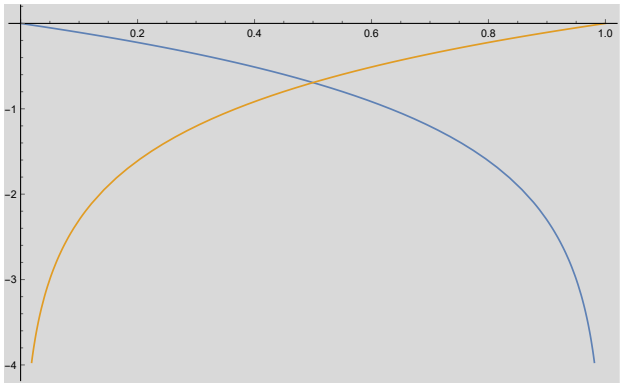


Fig. 3.5 $\log(1 - D_{\theta_d}(G(z)))$ (in blue) and $\log(D_{\theta_d}(G(z)))$ (in orange) as a function of $D(G(z))$

Figure 3.5 shows that with the new objective function (in orange), we have a high gradient in regions where the generator is still doing a bad job ($D(G(z)) \approx 0$) and the gradient vanishes only in regions where the generator is already sampling good quality fake images($D(G(z)) \approx 1$). So with this new objective function, the generator will learn more when it is still not well trained.

To train the GAN with this new objective function for the generator, we are going to alternate between gradient ascent on the discriminator to

$$\max_{\theta_d} \mathbb{E}_{x \sim p_{data}} [\log D_{\theta_d}(x)] + \mathbb{E}_{z \sim p_z(z)} [\log(1 - D_{\theta_d}(G(z)))] \tag{3.8}$$

and gradient ascent on the generator to

$$\max_{\theta_g} \mathbb{E}_{x \sim p_z(z)} [\log(D_{\theta_d}(G(z)))] \tag{3.9}$$

Algorithm 12 is thus changed into Algorithm 13:

Note that both in Algorithm 12 and Algorithm 13, for every training of the generator, the discriminator is trained k times. Optimal choices for k is a topic of debate amongst researchers. Some people argue that the discriminator should be trained more often than the generator ($k > 1$), while others say that the benefits of taking $k > 1$ are negligible compared to the additional computational cost that it induces. Since 2014 there has been a lot of theoretical work on GANs and one of the most promising research was the introduction of the Wasserstein GANs, which use the Wasserstein distance instead of the Jensen-Shannon divergence. In particular, Wasserstein GAN alleviates this problem. For more details see Section 3.8.

Algorithm 13 Minibatch stochastic gradient ascent training of a GAN with the modified objective function. For every training of the generator, we train the discriminator k times, where k is a hyperparameter

1: **for** *number of epochs* **do**
2: **for** *k steps* **do** ▷ update the **discriminator**

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(m)}\}$ from noise prior $p_z(\mathbf{z})$.
- Sample minibatch of m samples $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ from the training data generating distribution p_{data}.
- Update the discriminator by ascending its stochastic gradient computed on these minibatches:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D_{\theta_d}(\mathbf{x}^{(\mathbf{i})}) + \log \left(1 - D_{\theta_d} \left(G_{\theta_g} \left(\mathbf{z}^{(\mathbf{i})} \right) \right) \right) \right]$$

3: **end for**

▷ update the **generator**

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(m)}\}$ from noise prior $p_z(\mathbf{z})$.
- Update the generator by ascending its stochastic gradient computed on this minibatch:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log \left(D_{\theta d} \left(G_{\theta_g} \left(\mathbf{z}^{(\mathbf{i})} \right) \right) \right)$$

4: **end for**

3.5 The DCGAN architecture

Until the introduction of the DCGAN architecture by Radford et al. in their paper titled "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks" [37], the training of GANs were very unstable. The main advantage of the DCGAN architecture is that it provides a much more stable training. In this section we briefly discuss some of the key aspects of the DCGAN architecture.

3.5.1 Transpose convolution

The generator of a "standard" GAN is usually using simple fully connected neural network layers. To increase the spatial dimension of the inputs, the generator of the DCGAN uses transposed convolution with stride greater than one to up-sample the inputs. Recall that in section 1.2, we have discussed how we could down-sample the dimension of the input by applying a convolution with a kernel. Transposed convolution goes the other way round. It takes a low resolution input and outputs a high resolution output. Figure 3.6 explains how we can up-sample a 2×2 input matrix to a 4×4 output matrix using transposed convolution with 3×3 kernels and a stride of 2.

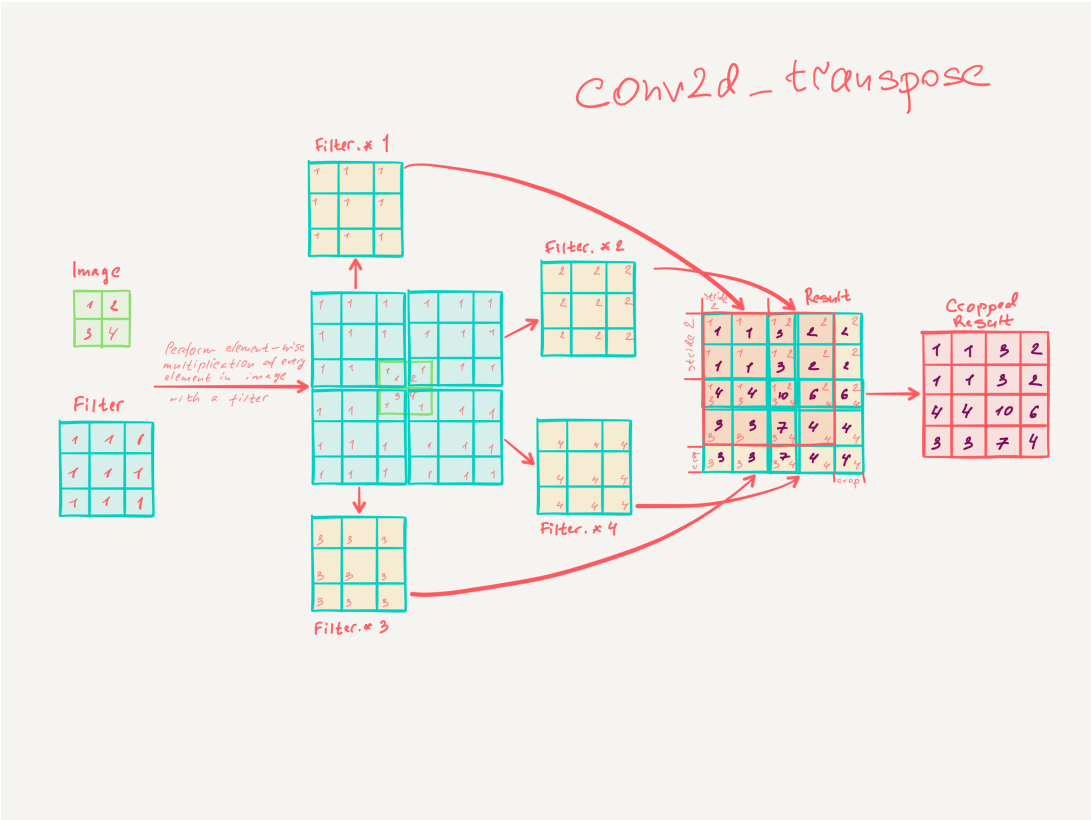


Fig. 3.6 Illustration of the transposed convolution (*Source: https://i.stack.imgur.com/GlqLM.png*)

For more details about the mathematics behind transposed convolution, the reader can refer to [46].

3.5.2 Batch normalisation

Another key aspect of the DCGAN architecture is to use batch normalisation in most layers of both the generator and the discriminator networks. However it is important not to batch normalise the input layer of the discriminator and the output layer of the generator to enable the GAN to learn the correct mean and standard deviation of the data distribution p_{data}. To understand batch normalisation, recall that before learning any neural network, we have to normalise the inputs in the pre-processing step. This is very important since usually inputs are measured on very different scales and we can't compare them without normalisation. The idea behind batch normalisation is to perform normalisation not only in the pre-processing step, but also in every layer of the neural network. Intuitively, the weights of certain features can get very large during the training process of a neural network. Other weights might get very small. If we don't use batch normalistaion, this results in outputs that are not comparable. Batch normalisation thus helps to maintain the outputs of the layers to be on a similar scale. In practice, neural networks using batch normalisation tend to achieve a higher accuracy rate

and the training process is also usually faster. For more details about batch normalisation, the reader can refer to the original paper [27] that introduced this concept.

3.5.3 Optimisers

Finally another key aspect of the DCGAN architecture is to use Adam optimiser instead of stochastic gradient descent with momentum. Although there is no theoretical explanation behind this, it seems that in practice Adam optimiser outperforms the other gradient based optimisers discussed in Chapter 2.

3.6 Pathological behaviours in practice

In this section we will discuss some of the difficulties encountered in practice when training a GAN.

Recall that all the theorems proved in Section 3.3, were proven in a non-parametric setup. In particular the poof of Theorem 3.3.3 about the convergence of Algorithm 12 relied on convexity properties that hold only if the parameter updates are made in function space. In practice however, the updates are made in parameter space meaning that we cannot guarantee the convergence of Algorithm 12. At the time of writing this essay, there is no theoretical results about the convergence of GANs when the updates are made to the parameters of the neural networks directly. This is an important open question and it would be interesting to study it in more details. In practice we usually observe that GAN tends to oscillate between different modes of the data distribution without converging to the equilibrium.

3.6.1 Difficulty of finding the Nash equilibrium of game with Gradient Descent

At NIPS 2016, Salimans et al. [40] discussed the problem of finding the Nash equilibrium of a game with gradient based optimisation algorithms. Recall that the generator and the discriminator of a GAN are trained together by alternating gradient steps on the generator and the discriminator networks. However, the cost function of the two networks are updated independently of each other. In particular this implies that the gradient-based algorithm is not guaranteed to converge to the Nash equilibrium of the game.

Consider the following example. Suppose that we have two players, each controlling a single scalar value of an objective function of the form:

$$V(x,y) = xy$$

Suppose that player 1 wants to minimise V with respect to x and player 2 wants to maximise the same objective with respect to y (i.e player 1 controls x to minimise $f_1(x) = xy$ while player 2 controls

y to minimise $f_2(y) = -xy$). This objective function $V(x,y)$ has a saddle point at $x = y = 0$ as shown in figure 3.7.

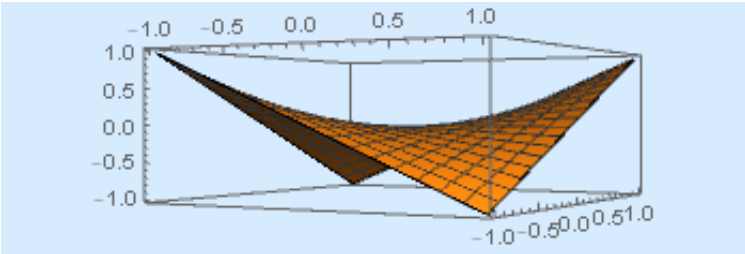


Fig. 3.7 $V(x,y) = xy$. This function has a saddle point at $x = y = 0$

In this example, the saddle point $(x,y) = (0,0)$ is the Nash equilibrium of the game since none of the players can do better by unilaterally changing their strategy. However gradient descent is not converging to this saddle point. Indeed the trajectory taken by two simultaneous gradient descent (one for player 1 and one for player 2) follow the derivatives

$$\frac{\partial x}{\partial t} = -y(t)$$
$$\frac{\partial y}{\partial t} = x(t)$$

Differentiating the second equation and plugging back into the first, we get the following differential equation:

$$\frac{\partial^2 y}{\partial t^2} = \frac{\partial x}{\partial t} = -y(t) \tag{3.10}$$

The differential equation 3.10 with initial conditions $x(0) = x_0$ and $y(0) = y_0$ has a solution given by:

$$\begin{cases} x(t) = x_0 \cos(t) - y_0 \sin(t) \\ y(t) = x_0 \sin(t) + y_0 \cos(t) \end{cases}$$

This solution can be visualised as a circular orbit as shown in figure 3.8. This implies that if we assume an infinitesimal learning rate, then simultaneously updating the weights of the two networks will result in orbiting around the equilibrium at a constant radius forever. If the learning rate is not assumed to be infinitesimal, then simultaneous gradient descent might spiral outward forever. In any cases, simultaneous gradient descent does not converge towards the equilibrium in this example.

In practice, we observe that simultaneous gradient descent does converge in some cases and does not in others. There is no theoretical guarantee for the convergence of simultaneous gradient descent in the case of GANs. Developping algorithms converging towards the Nash equilibrium of a non-cooperative game is still an open research problem.

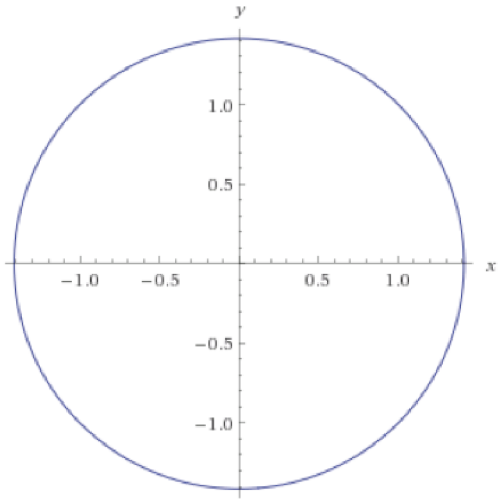


Fig. 3.8 Solutions of the PDE 3.10 with initial conditions $x_0 = y_0 = 1$ (*Source: [21]*)

3.6.2 Low dimensional supports

Recall that the support of a function f is defined to be the subset of the domain containing those elements which are not mapped to zero, i.e $\text{supp}(f) = \{x \in D_f : f(x) \neq 0\}$, where D_f is the domain of f. Recall also that an n-manifold is an n dimensional topological space that is locally euclidean, i.e around every point we can find a neighbourhood that is topologically equivalent to the open ball in $\mathbb{R}^n$. Figure 3.9 illustrates some compact manifolds.

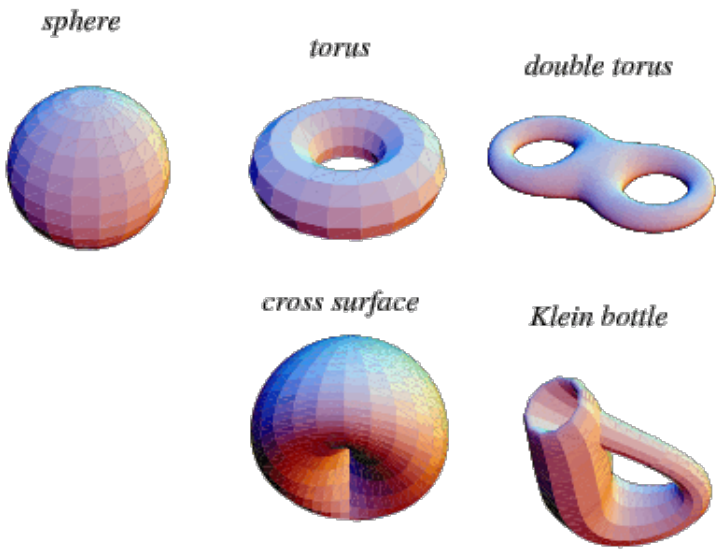


Fig. 3.9 Examples of (compact) manifolds (*Source: Wolframalpha*)

In 2017, Arjovsky and Bottou [8] proved that the supports of p_{data} and p_g lie on low dimensional manifolds and discussed how this contributes to the instability of GAN training. In this section we will only give the intuition behind their ideas and the reader is encouraged to read the original research paper for more details.

The main assumption behind manifold learning is that the dimensions of the real dataset represented by p_{data} are only artificially high. Indeed, suppose that we are interested in learning a GAN on real world images. Even though the underlying distribution of such a dataset might seem very high dimensional, once we fix the theme of the images, the distribution is enforced to live on a much smaller dimensional manifold. For instance a car should always have 4 wheels, dogs should have two ears, ... These restrictions force p_{data} to live on a low dimensional manifold. Moreover the implicit distribution p_g also lies on a low dimensional manifold. Indeed, even if the generator is trained to output a sample from a potentially high dimensional manifold (e.g. 128×128 images), in practice what happens is that the outputs of the generator hardly fill up the whole high dimensional space since the dimension of the latent space is usually very low compared to the high dimensional target space.

Intuitively, the fact that both p_{data} and p_g lie on low dimensional manifolds implies that the two distributions are going to be disjoint with high probability as illustrated in figure 3.10. This unfortunately leads to an unstable training of GANs. For more details, the reader can refer to [8].

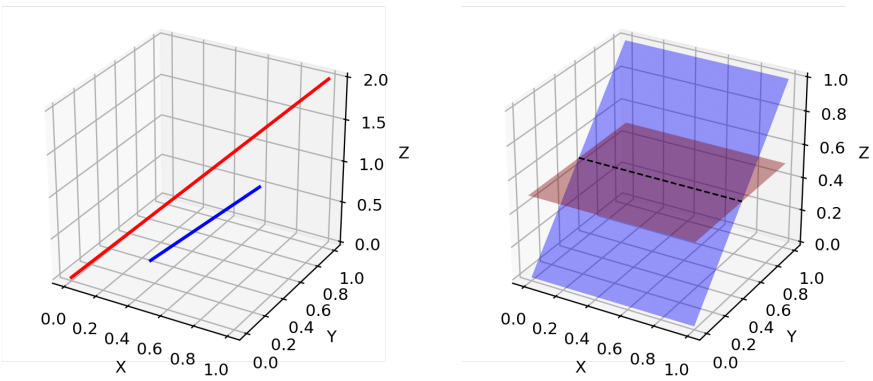


Fig. 3.10 Low dimensional manifolds in high dimensional spaces are usually disjoint. The left figure represents two lines, while the right figure shows two surfaces in a three-dimensional space. (*Source: [4]*)

3.6.3 Vanishing gradient

In this section we suppose that the objective of the GAN is defined as in the original zero-sum game defined by equation 3.1.

Suppose that the discriminator is near to optimality, i.e $D(x) \approx 1, \forall x$ sampled from the training

set and $D(G(z)) \approx 0$, $\forall z$ sampled from the latent variable. Recall that the original GAN objective is given by:

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

where

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

Note that when the discriminator is near to optimality, the value of this objective function and of it's gradient is close to zero. This is problematic since the generator is trained to minimise over θ_g

$$\mathbb{E}_{z\sim p_z}[log(1 - D_{\theta_d}(G(z)))]$$

using the gradients of the objective $V(G,D)$. When the discriminator is near to optimal, the gradients of the objective function $V(G,D)$ vanish and we cannot train the generator any more.

Figure 3.11 illustrates how the gradients vanish as the discriminator gets closer to it's optimal state.

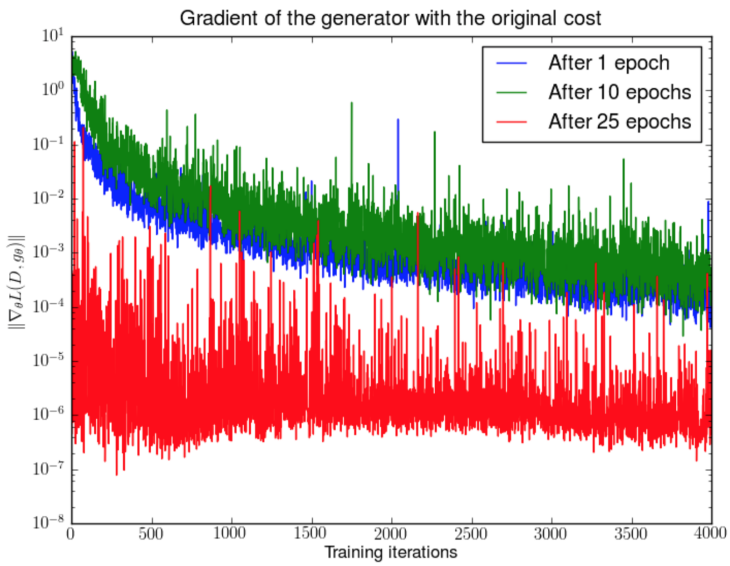


Fig. 3.11 First, we train a DCGAN for 1, 10 and 25 epochs. Then, we fix the generator, and train the discriminator from scratch. At each iteration, we record the gradients of the original cost function. The result shows a quick decay of the gradient norms. (*Source: Arjovsky and Bottou [8]*)

As a result training a GAN with the original objective functions is very challenging since if the discriminator behaves badly, then the generator does not get accurate feedback, but if the discriminator is too good, then the gradient of the objective function vanishes making the learning impossible.

One heuristic solution to this problem is to use an alternative objective function for the generator

as presented in Section 3.4. However this solution is just heuristic without any theoretical guarantees. A better approach is to use the Wasserstein GAN as we will discuss in Section 3.8.

3.6.4 Mode collapse

Mode collapse, also known as the Helvetica scenario, is one of the most common form of non-convergence. It refers to the situation when the generator maps several different latent variables z to the same output. This results in a generator that can generated very good quality fake samples from a few modes of the true data distribution p_{data}, but that struggles to generate samples from outside of these modes as illustrated in figure 3.12.

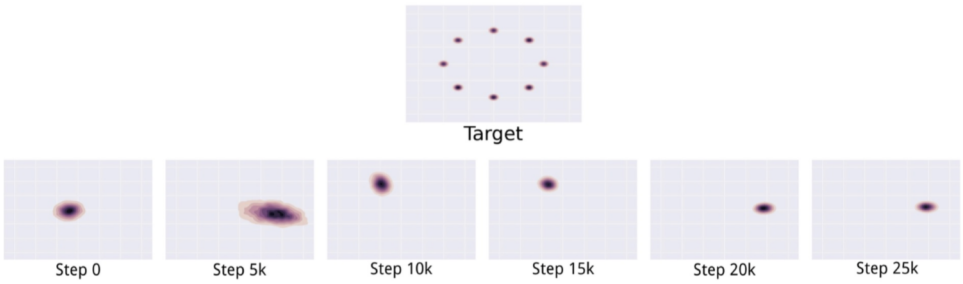


Fig. 3.12 Illustration of mode collapse. The top row represents the target distribution p_{data} (mixture of Gaussians in two dimensional space). In the lower row, we see the different distributions that the GAN has learned over time. Instead of converging to a distribution containing all modes of p_{data}, the generator only ever samples from one particular mode at a given time, and switches between the different modes as the discriminator learns to reject each of them. (*Source: [21]* and [34])

To understand intuitively the reasons behind mode collapse, suppose we want to train a GAN on a dataset having two very distinct modes. For examples we might use temperature records from Alice Springs in Australia and the South Pole during the summer as our training data (http://aiden.nibali.org/blog/2017-01-18-mode-collapse-gans/).

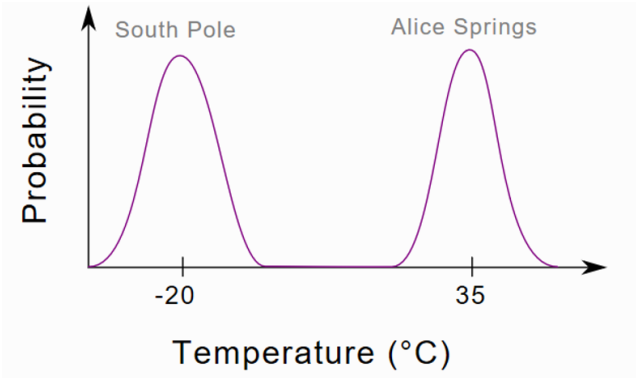


Fig. 3.13 True distribution p_{data} of the training data (*Source: http://aiden.nibali.org/blog/2017-01-18-mode-collapse-gans/)*

In this case the GAN is trained to output plausible temperatures. The two modes of the training set having the same probability, we would like the GAN to output temperatures from Australia and the South Pole with equal likelihood. Unfortunately this is not what we observe in practice. Instead of converging to a distribution containing all modes of p_{data}, the generator only ever samples from one particular mode at a given time, and switches between the different modes as the discriminator learns to reject each of them. To understand why, suppose that the generator learned that it can fool the discriminator by producing Antarctic temperatures. The discriminator will counter by learning that all Australian temperatures are real and randomly guessing for the Antarctic temperatures. The generator will then switch mode and produce only Australian temperatures until the discriminator counters by learning that all Antarctic temperatures are real...

Mathematically, mode collapse is due to the fact that we use simultaneous gradient descent on the generator and the discriminator in hope to find the solution of the following minimax game:

$$G^* = \min_G \max_D V(G,D) \tag{3.11}$$

However there is no theoretical guarantee that simultaneous gradient descent actually solves the minimax game 3.11 and not the following maximin game:

$$G^* = \max_D \min_G V(G,D) \tag{3.12}$$

Note that problems 3.11 and 3.12 are very different. In the minimax optimisation problem 3.11, the optimal generator G^* draws samples from the data distribution. However, when we change the minimax problem into a maximin problem, the minimization with respect to the generator gets into the inner loop of the optimization procedure. The generator is thus asked to map every latent variable z onto the single x value that the discriminator believes is most likely to be real.

One way to overcome mode collapse is to use minibatch features during the training. We will discuss this in section 3.7.5.

3.7 Improved GAN training

In this section we will discuss some practical techniques that help the training process of a GAN. To learn more about additional tips not included in this essay, one can check the GitHub repository associated with Soumith's "Improved Techniques for Training GANs" talk at NIPS 2016 (https://github.com/soumith/ganhacks) [40].

3.7.1 Training with labels

Instead of allowing the GAN to generate freely from any class of the training set, Denton et al. [15] proposed a class conditional GAN. The idea is to learn the GAN conditionally on the class labels. Suppose for instance, that we are interested in learning a GAN on the CIFA-10 dataset. This dataset consist of 60 000 colour images from ten classes (dog, horse, automobile, ...). In the "standard" GAN training, we would aim to learn one distribution for the whole dataset. Denton et al. propose to learn one separate distribution for every class of the dataset. Their experiments show that using labels during training results in an impressive improvement in the subjective quality of the samples generated by the generator. Figure 3.14 compares generated samples by a "standard" and a class conditional GAN trained on the CIFAR 10 dataset.

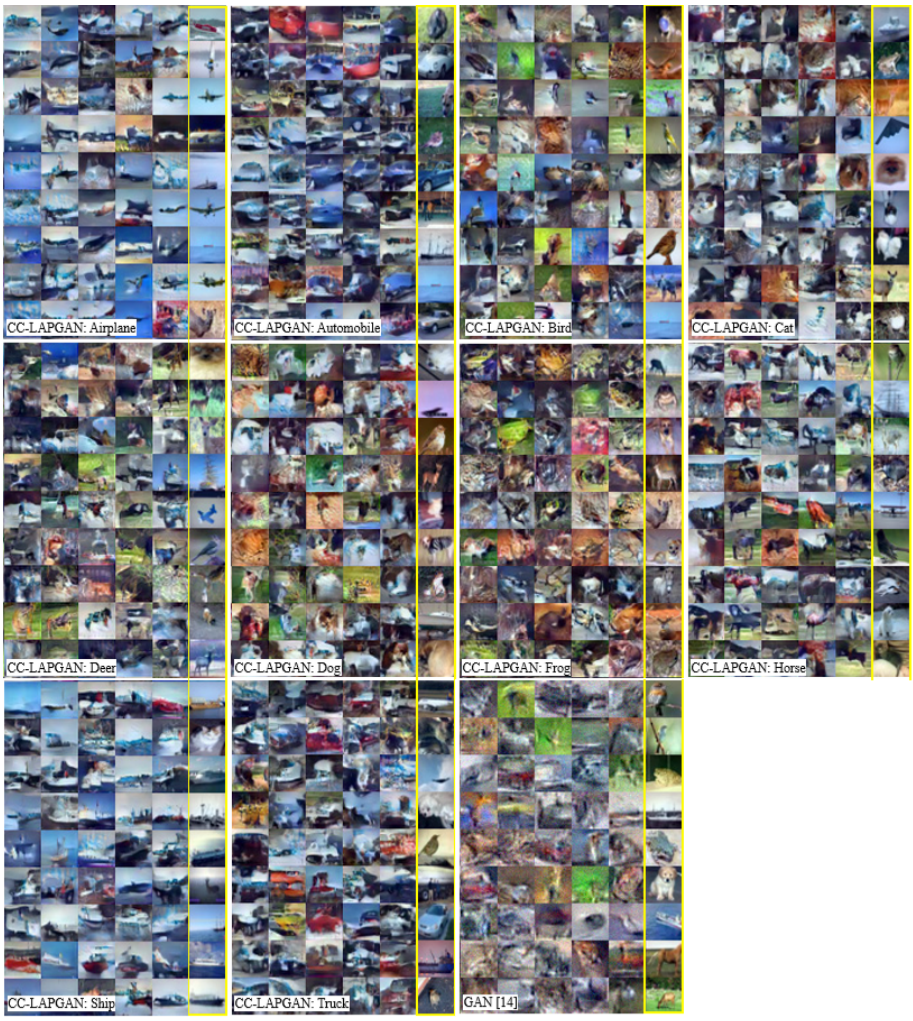


Fig. 3.14 CIFAR10 samples: class conditional CC-LAPGAN models of Denton et al. for each class and the standard GAN model of Goodfellow. The yellow column shows the nearest training example of the training set. We can see that both the CC-LAPGAN and the "standard" GAN did not only memorise the training set. (*Source: [15]*)

3.7.2 One sided label smoothing

In order to prevent the discriminator to produce too confident classifications, we can use a technique called one sided label smoothing introduced in 2016 by Salimans et al. [40]. When training the discriminator, instead of providing 0 and 1 labels for the fake and real samples respectively, we use a soften value such as 0.9 for the true samples.

In the "standard" GAN model, we train the discriminator with the objective function

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

which can be coded in TensorFlow as

```
d_on_data = discriminator_logits(data_minibatch) d_on_samples =
    discriminator_logits(samples_minibatch)
loss = tf.nn. sigmoid_cross_entropy_with_logits (d_on_data , 1.) +
    \ tf.nn. sigmoid_cross_entropy_with_logits(d_on_samples , 0.)
```

One sided label smoothing replaces the target for the real samples by a value slightly less than one (e.g 0.9):

```
loss = tf.nn. sigmoid_cross_entropy_with_logits (d_on_data , .9) +
    \ tf.nn. sigmoid_cross_entropy_with_logits(d_on_samples , 0.)
```

Remark 12. *Note that we did not smooth the labels for the fake samples. This is very important since if we did a two sided label smoothing then the shape of the optimal discriminator would be changed which could lead to incorrect behaviour of the generator. Indeed, suppose we use a target of* 1α *for the real data and a target of* $0+\beta$ *for the fake samples. Using the same arguments as in the proof of Theorem 3.3.1, we can derive that the optimal discriminator is given by*

$$D_G^*(x) = \frac{(1-\alpha)p_{data}(x) + \beta p_g(x)}{p_{data}(x) + p_g(x)} \tag{3.13}$$

In one sided label smoothing $\beta = 0$ *, so the smoothing by* α *just scales down the optimal value of the discriminator by a factor of* $(1-\alpha)$*. Suppose that we decided to do a two sided label smoothing, i.e we would take both* $\alpha \neq 0$ *and* $\beta \neq 0$*. In this case equation 3.13 tells us that the shape of the optimal discriminator function is changed compared to the optimal discriminator of the original game. This can lead to incorrect behaviour of the generator. For examples suppose that there exists a region where the true data distribution* $p_{data}(x)$ *is very small, but for some reason, the generator thinks that* $p_g(x)$ *should be relatively large. In such a region,* $D^*(x)$ *will have a peak near the artificial mode of* $p_g(x)$*. The discriminator will thus emphasis the incorrect behaviour of the generator. This means that if such a region exist, the generator could be actually trained to generate samples that look similar to the samples it already produces instead of being trained to generate samples that look similar to the training set. For more details see section 4.2 of [21].*

Remark 13. *It is also important to note that one sided label smoothing only encourages the discriminator to reduce the confidence in the correct class. In particular, it does not ever encourage the model to choose an incorrect class on the training set*

3.7.3 Virtual Batch Normalization

Since the introduction of the DCGAN architecture, most GANs use batch-normalization to learn faster and to achieve a better accuracy rate.

Unfortunately batch normalization has some negative side effects when used with GANs. Recall that during each epoch of the training, the inputs are fed into the model in minibatches. At each step of the training, batch-normalization will thus compute the normalization constants on different minibatches of data. This causes fluctuations of these normalization constants. These fluctuations, if they are too large, can have a greater impact on the generated samples than the latent noise z inputs. Typically the normalization constants have large fluctuations when the minibatch sizes are small. This is often the case in practice since we usually want to fit large generative models but have only access to limited GPU memory. Figure 3.15 illustrates this side effect.

Fig. 3.15 We train a GAN using batch-normalization and generate two minibatches of 16 samples each. We observe that the first minibatch contains only orange-tinted images while the second one contains only green-tinted samples. This is due to the fact that the fluctuations in the first two moments of the feature values in a minibatch have more effect on the generated samples than the individual latent noise z variables. Note that in theory the samples within a minibatch should be independent of each other since they are generated from independent latent variables and since a particular sample of the training set is used at most once within a given minibatch. We observe in this example that batch-normalization causes the samples within a minibatch to become correlated with each other. (*Source: [21]*)

One way to overcome this problem is to use a technique called Reference Batch Normalization (Salimans et al, 2016). The idea behind reference batch normalization is to chose a reference batch at the beginning of the training and use this same batch to normalise each data sample throughout the training process. For more details I recommend reading "Improved techniques for training gans" by Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., and Chen, X. (2016).

3.7.4 Feature matching

We can optimise the discriminator by giving him the opportunity to inspect whether the generated samples match some expected statistics of the real samples. Let f be a statistics (e.g. mean or median). Then we define a new loss function for the discriminator as the L_2-norm (squared) of the difference between the expected value of the statistic f under the real data distribution p_{data} and the implicit distribution p_g:

$$||\mathbb{E}_{x\sim p_{data}} f(x) - \mathbb{E}_{z\sim p_z(z)} f(G(z))||_2^2 = ||\mathbb{E}_{x\sim p_{data}} f(x) - \mathbb{E}_{x\sim p_g} f(x)||_2^2$$

3.7.5 Minibatch Features

One solution to avoid mode collapse is to encourage diversity of the generated samples. The idea behind minibatch features is to allow the discriminator to compare the input example to a minibatch of true samples and a minibatch of fake samples. The advantage is that using these minibatches, the discriminator can evaluate the similarities of the inputs and penalize samples that look too similar.

Another method to solve the problem of mode collapse is to use unrolled GANs (Metz et al, 2016). For more details please refer to the research paper [34].

3.7.6 Adding noise

In section 3.6.2 we have discussed that unfortunately p_{data} and p_g usually have low dimensional supports meaning that the two distributions will almost surely have disjoint supports. To spread out the supports of the distributions (and augment the probability of obtaining overlapping distributions) we can add continuous noise onto the inputs of the discriminator. Sometimes one might also wish to add noise to all the hidden layers of the discriminator as well. Note however that adding noise to the discriminator makes the training more difficult, and usually requires the networks to have a very deep architecture.

3.7.7 Using better metric to measure the similarity between two distributions

In section 3.3.2, we have proved that the loss function of a GAN measures the similarity between p_{data} and p_g using the Jensen-Shannon divergence. Unfortunately, the Jensen-Shannon divergence is not be smooth enough to ensure a stable training. In the next section we will introduce the Wasserstein metric which has the advantage of being much smoother than the Jensen-Shannon divergence, enabling a more stable training process.

3.8 Wasserstein GAN

In this section we will discuss the intuition behind the Wasserstein GAN. For a more rigorous mathematical analysis, see [10], [24] and [43]. This section closely follows the github blog of Lilian Weng [4].

3.8.1 The Wasserstein distance

In section 3.3.2, we have seen that we can use divergences to quantify the similarities between two distributions. In particular, the original GAN uses the Jensen-Shannon divergence to measure the similarity between p_{data} and p_g.

In this section we will introduce another way of measuring similarities between distributions using the Wasserstien (or Earth Mover's) distance.

Informally the Wasserstein distances between two distributions $\mathbb{P}$ and $\mathbb{Q}$ measures the minimal changes we have to make to transform the distribution $\mathbb{P}$ into the $\mathbb{Q}$. To understand this concept, consider the following simple example in discrete time. Suppose we have two distributions $\mathbb{P}$ and $\mathbb{Q}$, each having four modes, given by

$$\begin{aligned} \mathbb{P}(X=1) &= 0.3 & \mathbb{Q}(X=1) &= 0.1 \\ \mathbb{P}(X=2) &= 0.2 & \mathbb{Q}(X=2) &= 0.2 \\ \mathbb{P}(X=3) &= 0.1 & \mathbb{Q}(X=3) &= 0.4 \\ \mathbb{P}(X=4) &= 0.4 & \mathbb{Q}(X=4) &= 0.3 \end{aligned}$$

The distributions $\mathbb{P}$ and $\mathbb{Q}$ can be thought of as each representing four piles of dirt (with ten shovelfuls in total), where the number of shovelfuls in each dirt pile is given by:

$$\begin{aligned} P_1 &= 3 & Q_1 &= 1 \\ P_2 &= 2 & Q_2 &= 2 \\ P_3 &= 1 & Q_3 &= 4 \\ P_4 &= 4 & Q_4 &= 3 \end{aligned}$$

To calculate the Wasserstein distance between $\mathbb{P}$ and $\mathbb{Q}$, we need to change $\mathbb{P}$ to look like $\mathbb{Q}$. This process is illustrated in figure 3.16.

- **Step 1** move 2 shovelfuls from P_1 to P_2 to make P_1 and Q_1 match
- **Step 2** move 2 shovelfuls from P_2 to P_3 to make P_2 and Q_2 match

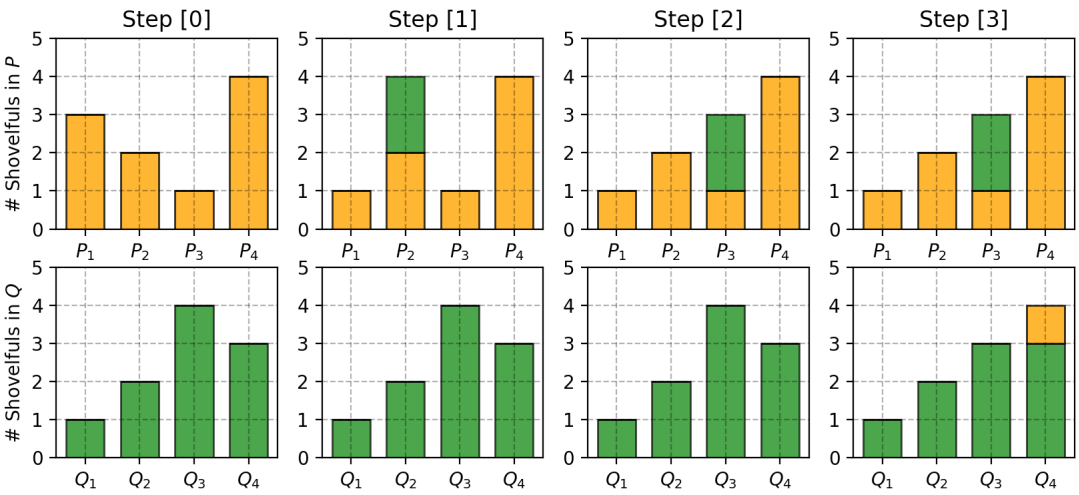


Fig. 3.16 Moving dirt between piles in $\mathbb{P}$ and $\mathbb{Q}$ to make the distribution match (*Source:*

- **Step 3** move 1 shovelfuls from Q_3 to Q_4 to make P_3 , Q_3 and P_4 , Q_4 match

Writing δ_i for the minimal cost to make P_i and Q_i match, the Wasserstein distance between $\mathbb{P}$ and $\mathbb{Q}$ is defined as

$$W(\mathbb{P},\mathbb{Q}) = \sum_i |\delta_i|$$

Note that the costs δ_i satisfy the recursion $\delta_{i+1} = \delta_i + P_i - Q_i$.

In our example we get

$$\begin{cases} \delta_0 = 0 \\ \delta_1 = 0+3-1 = 2 \\ \delta_2 = 2+2-2 = 2 \\ \delta_3 = 2+1-4 = -1 \\ \delta_4 = -1+4-3 = 0 \end{cases}$$

and thus the Wasserstein distance is equal to $W(\mathbb{P},\mathbb{Q}) = \sum |\delta_i| = 5$.

In the continuous case, the Wasserstein distance between two distributions is defined as follows.

Definition 9. *The Wasserstein distance between two probablility distributions $\mathbb{P}$ and $\mathbb{Q}$ with densities p and q respectively is defined as*

$$W(\mathbb{P},\mathbb{Q}) = \inf_{\gamma \sim \prod(p,q)} \mathbb{E}_{(x,y)\sim\gamma} ||x-y||$$

where $\prod(p,q)$ represents the set of all possible joint probability distributions whose marginal are respectively p and q.

Intuitively $\prod(p,q)$ corresponds to the set of all possible dirt transport plans. More precisely $\gamma(x,y) \in \prod(p,q)$ represents the percentage of dirt that should be moved from x to y to transform the distribution $\mathbb{P}$ into the distribution $\mathbb{Q}$. The cost of moving from x to y is given by the amount of dirt moved $(\gamma(x,y))$ multiplied by the distance between x and y $(||x-y||)$. The expected cost C is then obtained by averaging over all (x,y):

$$C = \sum_{x,y} \gamma(x,y)\,||x-y|| = \mathbb{E}_{(x,y)\sim\gamma}||x-y||$$

Finally we take the infimum over all dirt moving plans to get the definition of the Wasserstein distance.

3.8.2 Advantages of the Wasserstein distance

In the previous section, we have seen that the supports of p_{data} and p_g usually lie on low dimensional manifolds and do not overlap. In this case, the Jensen-Shannon divergence between the two distributions p_{data} and p_g is not smooth as illustrated in the example below. The main advantage of the Wasserstein distance is that it provides a smooth measure of the distance between p_{data} and p_g even when the supports of the distributions do not overlap. This makes the Wasserstein GAN much more stable during the training process.

To illustrate these results, consider the following example (see example 1 in the WGAN paper [10]). Let $Z \sim U[0,1]$ and let P_θ be the distribution of $(\theta, Z) \in \mathbb{R}^2$. Consider the following two distributions: $\mathbb{P} = P_0$ and $\mathbb{Q} = P_\theta$ for some $\theta \in \mathbb{R}$ as illustrated in figure 3.17.

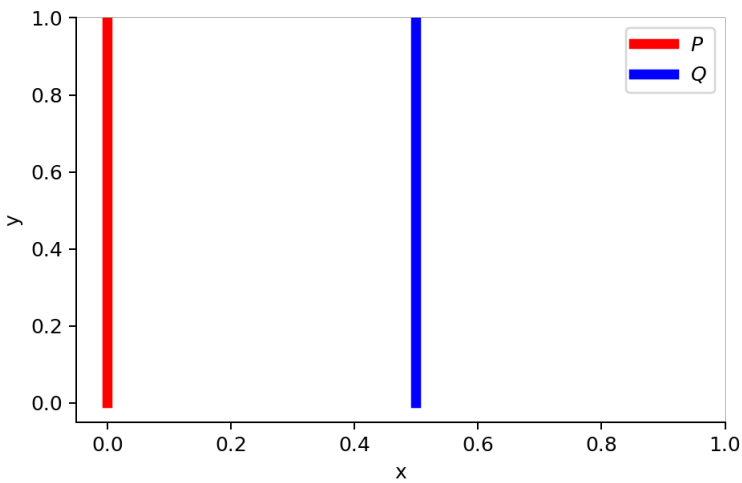


Fig. 3.17 There is no overlap between $\mathbb{P}$ and $\mathbb{Q}$ when $\theta \neq 0$ (*Source: [4]*)

We evaluate the Kulback-Leibler, the Jensen-Shannon divergence and the Wasserstein distance

between $\mathbb{P}$ and $\mathbb{Q}$.

$$D_{KL}(P\|Q) = D_{KL}(Q\|P) = \begin{cases} +\infty & \text{if } \theta \neq 0 \\ 0 \cdot z & \text{otherwise} \end{cases}$$

$$D_{JS}(P,Q) = \begin{cases} +\log 2 & \text{if } \theta \neq 0 \\ 0 \cdot z & \text{otherwise} \end{cases}$$

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

As we can see, both the KL and the JS divergences have jumps at $\theta = 0$. However, the Wasserstein distance provides a smooth measure of the distances between $\mathbb{P}$ and $\mathbb{Q}$.

This example shows that we can learn a probability distribution over a low dimensional manifold by using the Wassesrtein distance instead of the Kulback-Leibler or the Jensen-Shannon divergences.

3.8.3 Using the Wasserstein distance as GAN loss function

In the previous section we have seen that the Wasserstein distance $W(p_{data}, p_g)$ has nicer properties than the Jensen-Shannon divergence $JS(p_{data}, p_g)$. However the infinum in the definition of the Wasserstein distance is non-tractable since it is impossible to exhaust all posible elements of $\prod(p,q)$. Hopefully the Kantorovich-Rubinstein Duality enables us to transform the definition of the Wasserstein distance into

$$W(p_{data}, p_g) = \frac{1}{K} \sup_{\|f\|_L \leq K} \mathbb{E}_{x\sim p_{data}}[f(x)] - \mathbb{E}_{x\sim p_g}[f(x)] \tag{3.14}$$

where the supremum is taken over the K-Lipschtitz functions for some K. The proof of this duality result is beyond the scope of this essay and interested readers might refer to the WGAN paper [10] or to the following blog by Vincent Herrmann: https://vincentherrmann.github.io/blog/wasserstein/.

Now that we have transformed our original problem into this more tractable problem, we need to find f that solves the maximisation problem in equation 3.14. In the Wasserstein-GAN, we will suppose that the solution f belongs to a family of K-Lipscitz functions $f_{w_{w\in W}}$ parametrized by w. The "discriminator" is trained to learn the optimal w^*. We can then use $f^* = f_{w^*}$ as a solution of equation 3.14. The discriminator of the Wasserstein-GAN is thus no longer used to classify real and fake samples. It just helps to find the solution of equation 3.14. In that sense the word adversary is probably no longer correct since the generator and discriminator networks of a Wasserstein-GAN work more as partners than as adversaries. The loss function of the Wasserstein -GAN is defined to measure the Wasserstein distance between p_{data} and p_g.

$$V(G,D) = W(p_{data}, p_g) = \max_{w\in W} \mathbb{E}_{x\sim p_{data}}[f_w(x)] - \mathbb{E}_{z\sim p_r(z)}[f_w(g_\theta(z))]$$

One practical problem is to ensure the K-Lipshitz condition on f_w over the training process. The Wasserstein-GAN paper [10] proposes a heuristic solution which consists of clamping the weights w to a small compact space W (e.g: W = [-0.001,0.001]) after each gradient update. With this trick, the weigh parameter w lies in a compact space W which implies that all functions f_w will be K-Lipschitz for a certain K depending only on W.

The Wasserstein GAN algorithm is then finally given by the following pseudo-code:

Algorithm 14 Wasserstein-GAN

Require: learning rate α, clipping parameter c, batch size m, number of iterations of the discriminator per generator iteration n_d

Require: initial parameters of the discriminator network ω_0, initial parameters of the generator network θ_0

1: **while** θ has not converged **do**

2: **for** $t = 0, ..., n_d$ **do**

Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(m)}\}$ from noise prior $p_z(\mathbf{z})$.

Sample minibatch of m samples $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ from the training data generating distribution p_{data}.

$g_\omega \leftarrow \nabla_\omega \left[\frac{1}{m}\sum_{i=1}^m f_\omega(x^{(i)}) - \frac{1}{m}\sum_{i=1}^m f_\omega(g_\theta(z^{(i)}))\right]$

$\omega \leftarrow \omega + \alpha \cdot$ RMSProp(ω, g_ω)

$\omega \leftarrow clip(\omega, -c, c)$

3: **end for**

Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(m)}\}$ from noise prior $p_z(\mathbf{z})$.

$g_\theta \leftarrow$- $\nabla_\theta \frac{1}{m}\sum_{i=1}^m f_\omega(g_\theta(z^{(i)}))$

$\theta \leftarrow \theta - \alpha \cdot$ RMSProp(θ, g_θ)

4: **end while**

Remark 14. *The authors of the Wasserstein-GAN paper recommend using RMSProp instead of the momentum based Adam optimiser for the training of the WGAN.*

Unfortunately, in practice, Wasserstein-GAN can still be unstable if the clipping window is too large. Moreover taking too small clipping window is not a good idea either since it leads to the problem of vanishing gradient.

Since the introduction of Wasserstein-GAN in 2017, several researchers have proposed alternatives to weight clipping. In particular, Gulrajni et al proposed the improved Wasserstein GAN in which the Lipshitz constraint is replaced by a smooth gradient penalty. These newer improvements go beyond the scope of this essay and the interested reader is encouraged to read [24] and [43].

Chapter 4

Experiments

In this chapter we will train three different GANs, each on a different training set. For illustration purposes, we will use different deep learning frameworks (TensorFlow, Keras and Pytorch) for each of them.

The first GAN will be a toy example coded in Keras where the GAN will be trained to generated sinusoidal curves (code source : http://www.rricard.me [39]). In the second example we will train a GAN on MNIST, which is a large dataset of handwritten digits, using TensorFlow directly (code source: Complete Guide to TensorFlow with Python online course [1]). Finally in the third example, we will implement a DCGAN using Pytorch and train it on the CIFAR-10 dataset (code source: Deep Learning and Computer Vision A-Z online course [3]). For each implementation, the reader can refer to a Python tutorial in Appendixes A, B and C respectively.

4.1 Generating sinusoidal functions

In this section, we train a GAN on simulated sinusoidal functions. The implementation is done using Keras and is available online on Robin Riccard's blog about Machine Learning [39]. For an explanation of the implementation, please refer to Appendix A.

4.1.1 Model

The training data is obtained by randomly generating sinusoidal curves. As discussed in Chapter 3, the generator will take random noise as input and will and try to generate sinusoidal curves from it. In this basic example, the generator is a feed-forward neural network containing one input layer, one fully connected dense hidden layer with *tanh* activation and one output layer. We also define the loss function of the generator and the optimiser that we are going to use to train the generator. In this case, we will use stochastic gradient descent with a learning rate of 0.001. The generator

network will be trained only via the GAN by freezing the weights of the discriminator. Similarly we create the discriminator network who will try to classify real and fake samples. In this example the discriminator consists of a convolutional neural network with two hidden layers. The first hidden layer is a convolutional layer using a ReLU activation function and a dropout rate of 0.25 to avoid overfitting. The second hidden layer is a fully connected layer. The output layer of the discriminator uses a sigmoid activation function to force the outputs to be between 0 and 1 (necessary since the output represents the probability that the input came from the true data distribution). We also define the loss function for the discriminator and the optimiser that we will use during the training of the discriminator. In this case we will use Adam optimiser. The discriminator will be trained directly. The architecture of the GAN is represented in figure 4.1.

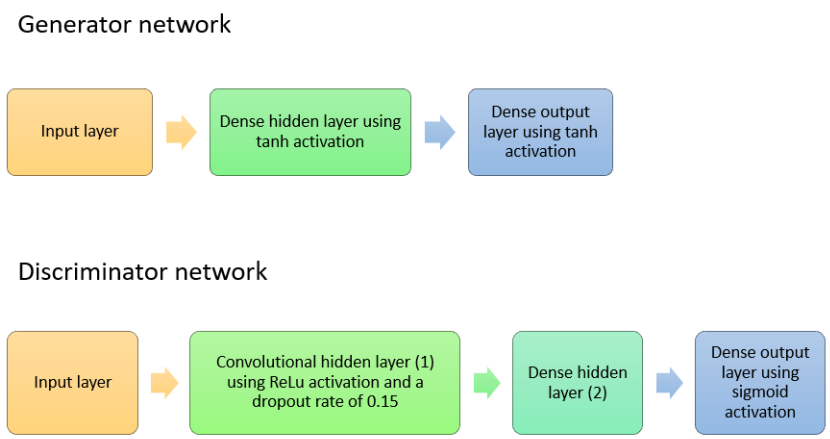


Fig. 4.1 Architecture of the generator and dicriminator networks

For both networks, we use the binary-cross entropy loss, which is equivalent to the objective function $V(G,D)$ defined in Chapter 3. Moreover, both networks will be updated using Adam optimiser.

4.1.2 Results

We trained the network for 500 epochs. Figure 4.2 shows the evolution of the loss functions for both the discriminator and generator networks.

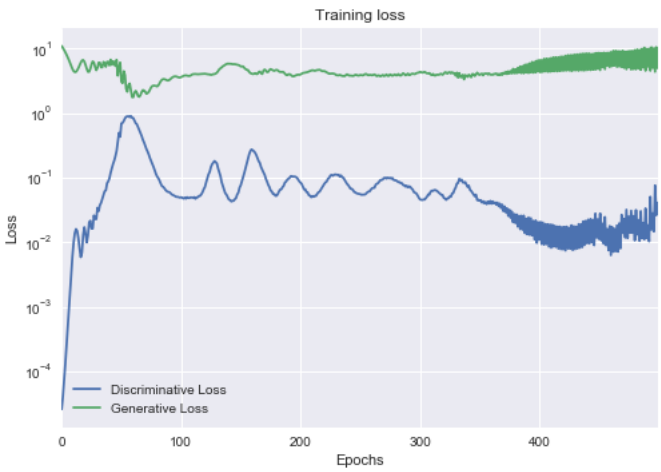


Fig. 4.2 Losses of the two networks

As one can see, the losses do not seem to converge towards zero. We would probably need a better tuning of the hyper parameters to obtain a better convergence.

We can visually inspect whether the GAN has learned to create sinusoidal curves by plotting the generator's output for two different latent variables.

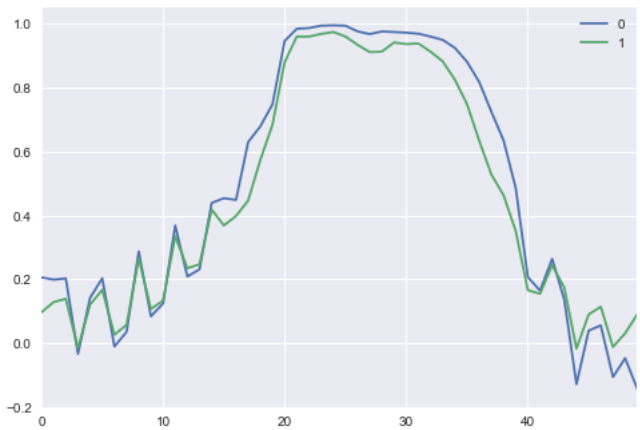


Fig. 4.3 Samples generated by the generator after training

Obviously, the results are not very good. We would need more training and a better tuning of the hyper-parameters to generate samples of greater quality.

Finally, note that the GAN did not avoid the problem of mode collapse. Indeed we see that the generator outputs very similar curves for different noise inputs. To overcome this problem we should use minibatch features to access the diversity of the generated samples (as dicussed in section 3.7.5).

4.2 Training a GAN on the MNIST dataset

In this section we will train a GAN on the MNIST dataset using TensorFlow. The MNIST database is a large dataset of handwritten digits as shown in figure 4.4.

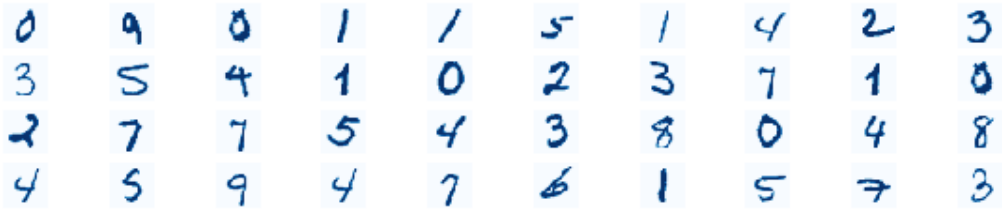


Fig. 4.4 Samples from the MNIST dataset

Our goal is thus to create a generator capable of producing fake handwritten digits. The implementation closely follows a code written by Jose Portilla and available online as part of the *Complete guide to TensorFlow for deep learning wih Python* online course [1]. The implementation is also explained in Appendix B.

4.2.1 Model

We define the two networks as follows. The input layer of the generator network is a densely connected layer having 128 neurons. We use the Leaky Rectified Linear Unit (LeakyRelu) activation function for this first layer. The generator also has a densely connected hidden layer with 128 neurons. This hidden layer also uses the LeakyRelu activation function. Finally, the output layer of the generator is a fully connected layer with 784 neurons (since we want to output a 28×28 image). Experimentally, the *tanh* activation function seems to work well for the output layer. The discriminator takes as input real or fake samples. The goal of the discriminator is to predict the probability that the input comes from the real data distribution. We will use the same architecture as for the generator with the only difference that the output layer will have only one neuron with sigmoid activation function to force the output to be between 0 and 1. The architecture of the GAN is represented in figure 4.5.

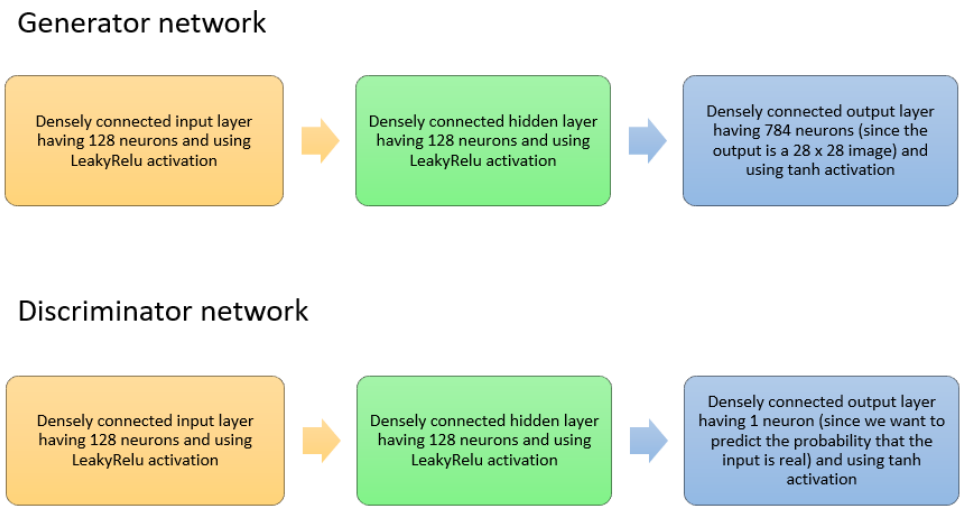


Fig. 4.5 Architecture of the generator and dicriminator networks

For both networks, we use the sigmoid-cross entropy loss. Moreover, both networks are updated using Adam optimiser with a learning rate of 0.001.

4.2.2 Results

We trained the network for 500 epochs. Figure 4.6 shows the evolution of the loss functions for both the Discriminator and Generator networks.

Fig. 4.6 Losses of the two networks

As one can see, the losses do not seem to converge towards zero. We would probably need a better tuning of the hyper parameters to obtain a better convergence.

We can visually inspect whether the GAN has learned to create fake hand-written digits by plotting the generator's output for two different latent variables.

Fig. 4.7 Generated samples after 500 epochs

4.3 Implementing a DCGAN for image generation

In this section, we will train a DCGAN using Pytorch on the CIFAR10 dataset. The CIFAR-10 dataset consists of 60 000 32×32 colour images from 10 classes, with 6000 images per class. The ten classes are airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. Figure 4.8 shows a random sample from the CIFAR 10 dataset.

Fig. 4.8 Examples of the CIFAR 10 training set (*Source: [3]*)

Remark 15. *In this thesis, we only implement a "standard" DCGAN, but it would be interesting to implement another class conditional DCGAN and compare the results of the two models.*

The implementation of the DCGAN is done using Pytorch (only available on Linux and IOS). The code was written by Kirill Eremenko and Hadelin de Ponteves and is available online as part of the *Computer Vision A-Z* course [3] on Udemy. An explanation of the code is also available in Appendix C.

4.3.1 Model

In this example, we used a more advanced architecture for both networks. The generator uses several transposed convolutional layers together with batch-normalisation and ReLU activation function. The final output layer uses a *tanh* activation. We also define the discriminator network. The discriminator uses several convolutional hidden layers with batch-normalisation and leaky rectified linear unit activation functions. The output layer of the discriminator uses a sigmoid activation (since we want to predict the probability that the input comes from the true data distribution p_{data}). The architecture of the DCGAN's networks is illustrated in figure 4.9.

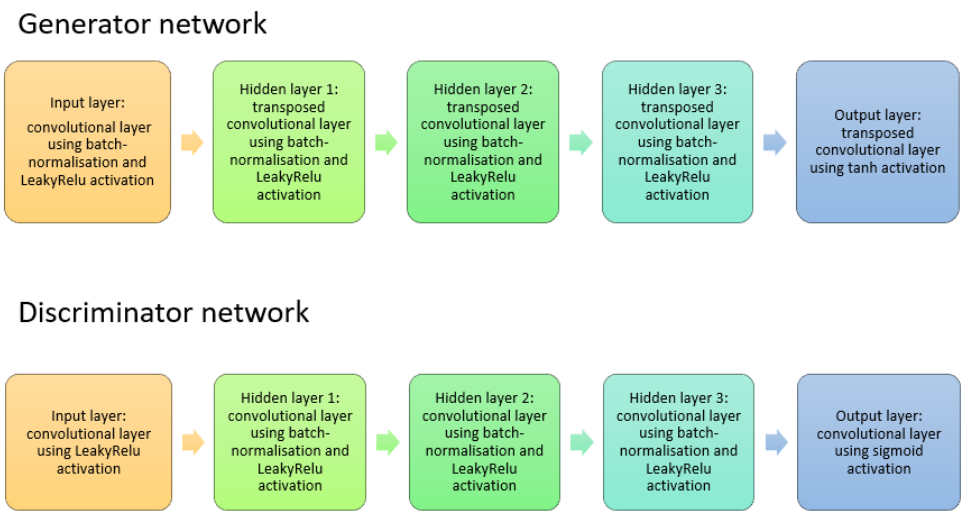


Fig. 4.9 Architecture of the generator and dicriminator networks

For both networks, we use the binary-cross entropy loss as objective function and Adam optimiser with a learning rate of 0.0002 to update the parameters.

4.3.2 Results

Due to limited access to computer power, we only trained the network for 25 epochs.

We can visually inspect how the DCGAN gets better and better at creating fake images by plotting the generator's output for different epochs.

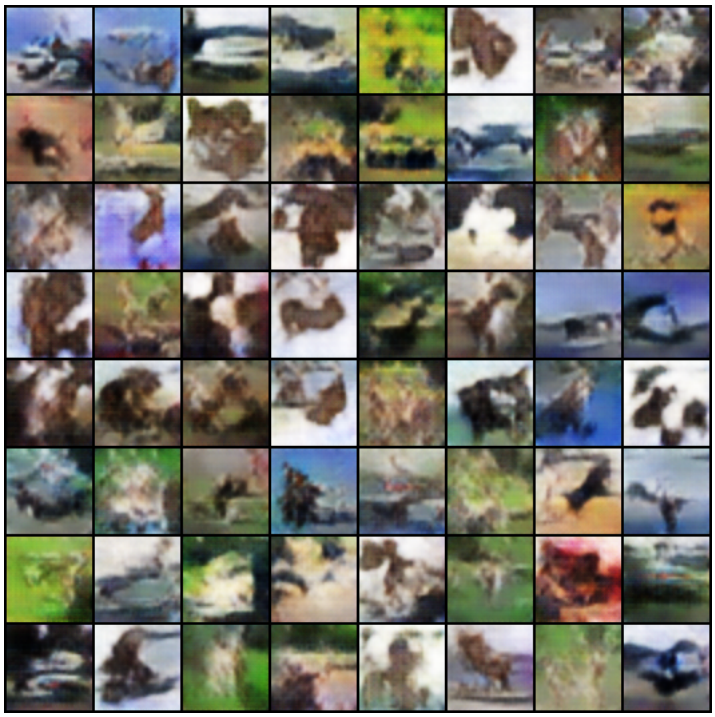


Fig. 4.10 Fake samples after 5 epochs
(*Source: [3]*)

Fig. 4.11 Fake samples after 10 epochs (*Source: [3]*)

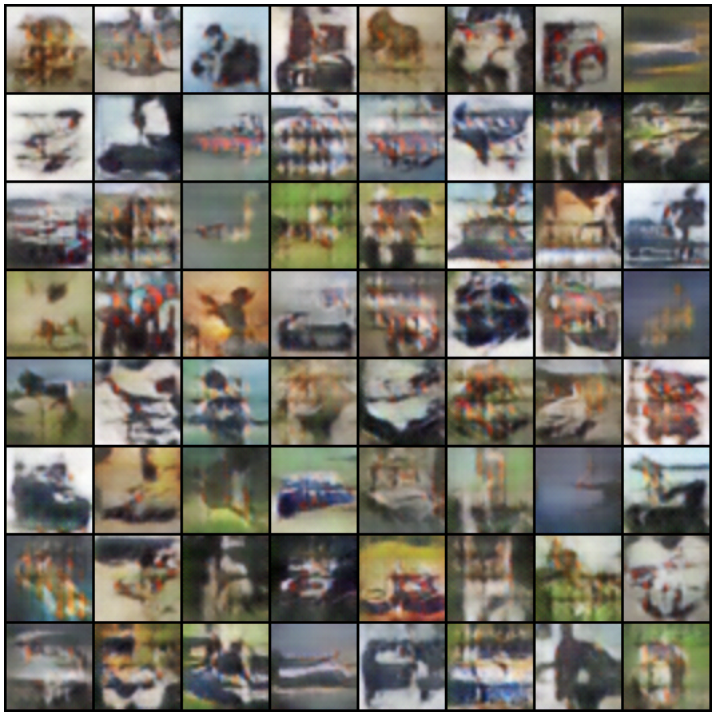


Fig. 4.12 Fake samples after 15 epochs (*Source: [3]*)

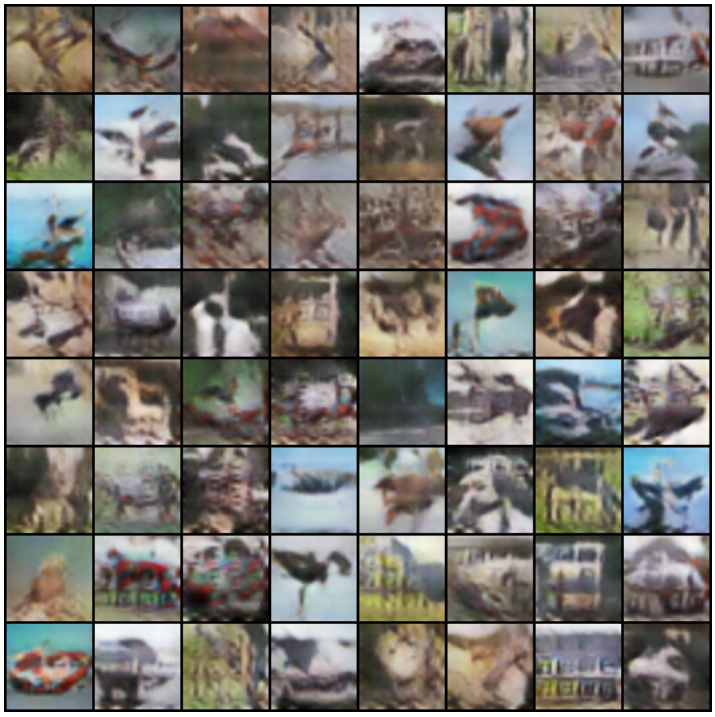


Fig. 4.13 Fake samples after 20 epochs (*Source: [3]*)

Fig. 4.14 Fake samples after 25 epochs (*Source: [3]*)

Chapter 5

Financial time series generation using Recurrent Generative Adversarial Networks

5.1 Machine Learning in Finance

In the recent years there has been a lot of research about how one could apply machine learning algorithms in a financial context. Due to limitations in the length of this thesis, I cannot explain these in details, but the interested readers can refer to the following papers (amongst others):

Optimal stopping

- Sebastian Becker, Patrick Cheridito, Arnulf Jentzen: Deep optimal stopping [11]

Model calibration with neural networks (i.e. how to find the parameters, of a given stochastic model, that correspond to the observed data)

- Andres Hernandez: Model Calibration with neural networks [9]
- Christa Cuchiero, Josef Teichmann, et al: Bayesian Finance – a machine learning approach to a simple calibration problem [14]

Hedging a portfolio of derivatives

- Hans Buehler, Lukas Gonon, Josef Teichmann, and Ben Wood: Deep hedging [13]

Pricing high dimensional American Options by neural networks

- Michael Kohler, Adam Krzyzak, Nebjosa Todorovic: Pricing of high-dimensional American Options by neural networks [29]

Forecasting exchange rates

- Tan Elliot, Patrick Cheridito: Forecasting Foreign Exchange Rates in Response to Federal Reserve Communication: A Machine Learning Approach [18]

Solving high-dimensional parabolic partial differential equations numerically using deep learning

- Weinan E, Jiequn Han, Arnulf Jentzen: Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations [17]

Filtering signals from noisy observations

- Terry Lyons, Rough paths, Signatures and the modeling of functions on streams [32]

Most of these research papers have been discussed at a Seminar in Computational Finance at ETH Zurich in 2017 (https://people.math.ethz.ch/ jteichma/index.php?content=teach_seminar17).

5.2 Augmenting the financial datasets

Although there has been an expansion of the number of publications in machine learning applied to finance, one of the main drawbacks to apply ML in finance is still the lack of access to sufficiently large training sets. Different authors have proposed different ways to artificially augment the available financial datasets (include examples). In practice, one of the most used method to create artificial data relies on Monte Carlo simulations. In this chapter, I propose a different approach using GANs. The advantage of the GAN approach is that unlike Monte Carlo simulations, it is not necessary to make any statistical assumptions about the future evolution of the market since the generator automatically learns the model describing the price evolution. Before explaining how GANs can be used to create artificial financial data, let's first recall how Monte Carlo simulations are made.

Monte Carlo simulations

Suppose for example that we wanted to simulate the price evolution of a particular stock using traditional Monte Carlo simulations. We could for instance estimate the mean return (by calculating the compound annual grow rate) and the volatility of the returns using the past price evolutions and then simulate new prices under the assumption that the returns follow a Gaussian distribution with the estimated parameters. However this normality assumption is not true in practice where we tend to observe higher probabilities for the tail events than those predicted by the Gaussian distribution. We could change our assumption (e.g into a student distribution), but neither would that assumption

completely describe the reality. The code in Appendix A (inspired by a code on pythonforfinance.net) implements this Monte Carlo simulation. Figure 1 shows 1000 Monte Carlo simulations of the price evolution of the Apple stock price over the next financial year (253 days).

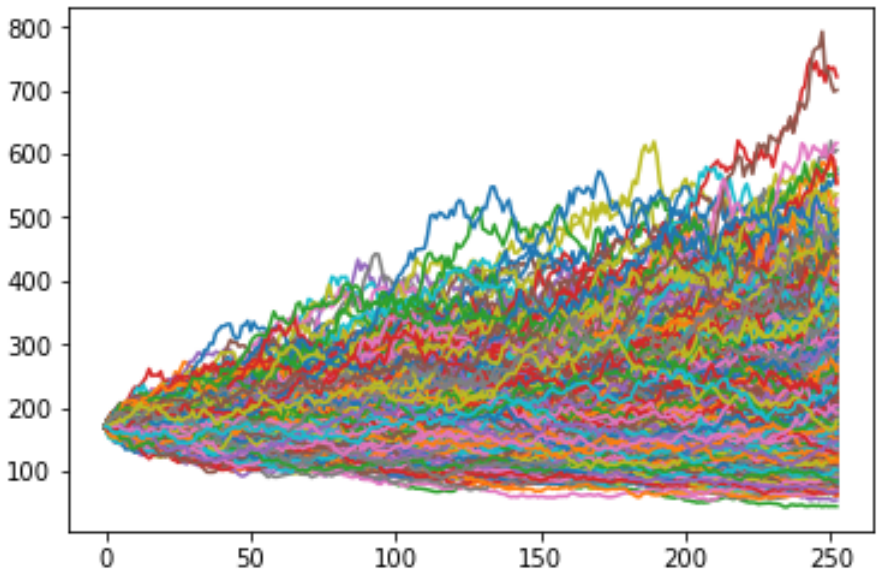


Fig. 5.1 Monte Carlo simulation of the future price evolution of the Apple stock price

5.2.1 Training a GAN on financial time series

In this thesis, instead of using Monte Carlo simulations to augment the available financial datasets, we will try to train a GAN on financial time series as discussed in the next section.

5.3 Training a GAN on financial time series

As we have seen in the previous section, access to financial data is the main barriers in the development of machine learning solutions to finance specific problems. We have also seen the disadvantages of using Monte Carlo simulations to create new artificial datasets. In this work, we propose a new approach via Generative Adversarial Networks. Since we want to generate time series, it is tempting to propose a GAN architecture using Recurrent Neural Networks (in particular LSTMs) in both the discriminator and the generator. To my knowledge there has been only two preliminary work using GANs to generate sequences. In 2016, Mogren [35] trained a recurrent GAN using LSTM cells for both networks. He used it to produce artificial music. In 2017 researchers at ETH Zurich introduced another the Recurrent GAN (RGAN) architecture to generate real-valued medical time series [19]. There are only minor architectural differences between these two RGANs.

In the next section we will discuss the architecture of the RGAN as proposed by the ETH Zurich group in [19]. In Appendix B, we implement an RGAN using TensorFlow. The code is mainly based on the code released with the RGAN paper [19].

5.3.1 RGAN architecture

This section follows very closely section 3 of the Real-valued (Medical) Time Series Generation with Recurrent Conditional GANs paper [19].

The main difference between the RGAN and a "normal" GAN, is that in the RGAN both the discriminator and the generator networks are using recurrent neural networks. The use of RNNs and in particular LSTM cells make the RGAN's generator particularly well suited for generation of real valued sequential data.

As shown in figure 2 the generator RNN takes a different random seed at each time step, and produces a synthetic signal. The discriminator RNN takes real and synthetic sequences as input and produces a classification into real/synthetic classes for each time step of the input sequence.

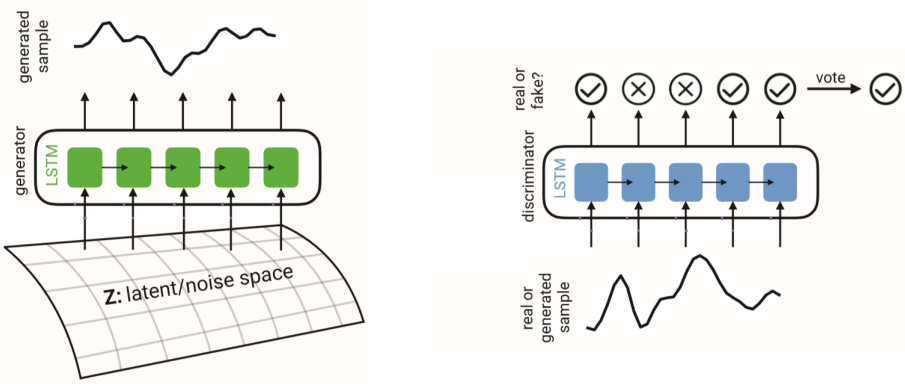


Fig. 5.2 Architecture of the generator and discriminator of a RGAN (Source: Real-valued (Medical) Time Series Generation with Recurrent Conditional GANs [19])

More precisely, the discriminator is trained to minimize the average negative cross-entropy between its predictions per time-step and the labels of the sequence.

Let's introduce some notations. Let $RNN(X)$ be the vector or matrix comprising the T outputs from a RNN receiving a sequence of T vectors $\{\mathbf{x}_t\}_{t=1}^{T}$ ($\mathbf{x}_t \in \mathbb{R}^d$). Let $CE(\mathbf{a},\mathbf{b})$ the average cross-entropy between two sequences **a** and **b**.

Using these notations, we can define the discriminator loss for a pair $\{\mathbf{X}_n, \mathbf{y}_n\}$ (with $\mathbf{X}_n \in \mathbb{R}^{T\times d}$ and $\mathbf{y}_n \in \{0,1\}^T$ as:

$$D_{loss}(\mathbf{X}_n, \mathbf{y}_n) = -CE(RNN_D(\mathbf{X}_n), \mathbf{y}_n) \tag{5.1}$$

Here the vector $\mathbf{y}_n$ is a vector of 1s for real sequences and a vector of 0s for synthetic sequences. As usual with GANs, the discriminator sees both real and synthetic sequences in each mini batch training step.

The objective of the generator is again to fool the discriminator. It wants the discriminator to think that its outputs are real. The generator will thus minimize the average negative cross-entropy between the discriminator's prediction on generated sequences and the vector of 1s corresponding to the "true" label (denoted by $\mathbf{1}$). Let Z_n be a sequence of T points $\{\mathbf{z}_t\}_{t=1}^{T}$ (where $\mathbf{z}_t \in \mathbb{R}^m$) sampled independently from the m-dimensional latent space $\mathbf{Z} = \mathbb{R}^m$. Thus for all n, $Z_n \in \mathbb{R}^{T \times m}$. The loss function of the generator is thus defined as:

$$G_{loss}(Z_n) = D_{loss}(RNN_G(Z_n), \mathbf{1}) = -CE(RNN_D(RNN_G(Z_n)), \mathbf{1}) \tag{5.2}$$

5.3.2 Experiment

I implemented a Recurrent GAN (based on the code published with the original RGAN for Medical Time Series paper [19]). The implementation is explained in Appendix B.

As a training set I used 50 000 Monte Carlo simulations of the log-returns of the Australian Foundation Investment Company (AFI.AX). The Monte Carlo simulations are made under the assumption that the daily returns follow a Gaussian distribution with estimated parameters.

I trained the RGAN for 50 epochs (limited computer power) and after training used the generator to generate the evolution of the prices of AFI.AX for ten days (one price per day). Despite the lack of proper training (larger epoch number), the results look quite convincing.

I then generated a test set using Monte Carlo again, and compared the predicted prices for the 10th day by the Monte Carlo and RGAN methods (on a sample of 50 000 for each method). I used a Kolmogorov Smirnov test to test the null hypothesis that the two samples are drawn from the same distribution and hopefully the obtained p-value lead to the non-rejection of H0 at 5% level.

Remark 16. *To train the RGAN we should use real financial data, however, due to the difficult access to large financial datasets and due to the limited amount of time available for this master thesis, I trained the RGAN on 50000 Monte Carlo simulations.*

Remark 17. *I have manually chosen the hyper-parameters of the networks (number of hidden units in both networks, learning rate, dimension of the latent space, number of times we train the discriminator for one training of the generator, ...). If I had access to more computer power, these parameters should be chosen via Cross-Validations. Unfortunately, this heuristic choice for the parameters is not optimal as suggested by figure 3.*

Remark 18. *I have tried different combinations of optimizers but Stochastic Gradient Descent for the*

discriminator and Adam optimizer for the generator seemed to work the best. The choice of optimizers should be studied in more details.

Results

I trained the network for 50 epochs. Figure 3 shows the evolution of the loss functions for both the discriminator and generator networks.

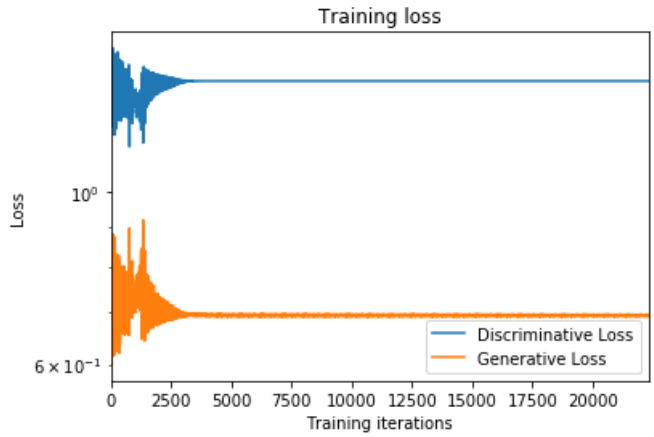


Fig. 5.3 Losses of the two networks

As one can see, unfortunately the losses do not seem to converge towards zero. Even worse, the seem to stay constant after only a few training. In order to obtain a better convergence, we would need a better tuning of the hyper parameters via cross-validations.

We can visually inspect whether the GAN has learned to create fake time series by comparing the generator's output for 50 000 different latent variables with a test set of 50 000 new Monte Carlo simulations generated independently from the training set.

5.3.3 Comparing the real and generated samples

In this section, for simplicity, we will suppose that we are only interested in the prediction of the stock price for the 10th day. We will then compare the vector of 50 000 predicted prices using Monte Carlo with the vector of 50 000 predicted prices using the RGAN.

To test statistically whether the distribution of the generated prices are similar to the distribution of the prices obtained via Monte Carlo, we first compare the major statistics of the two empirical distributions shown in figure 5.6.

We can also visualize these results via boxplots and histograms as shown in figures 5.7 and 5.8 respectively.

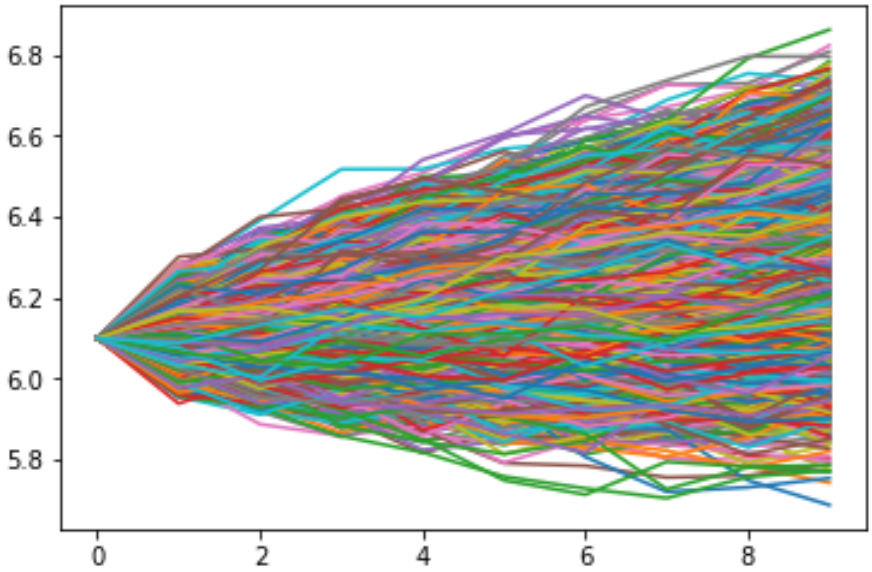


Fig. 5.4 Real samples

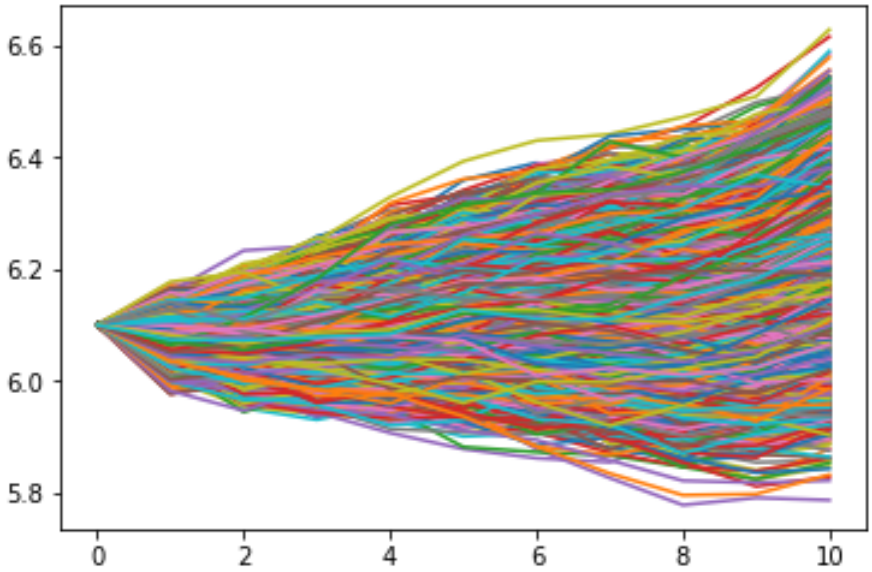


Fig. 5.5 Fake samples

Finally to formally test whether the two vectors are from the same distribution, we use a Kolmogorov Smirnov test. The Kolmogorov Smirnov test is testing the null hypothesis that the two samples are from the same distribution. The obtained p-value leads to the non rejection of H0 at 5% level . In other words, the two vectors are likely to come from the same distribution.

```
In [4]: print(results.describe())
                Fake          Real
count  50000.000000  50000.000000
mean       6.255305      6.255051
std        0.137163      0.138182
min        5.698330      5.681760
25%        6.162344      6.161181
50%        6.254092      6.253195
75%        6.346743      6.348192
max        6.864863      6.880436
```

Fig. 5.6 Statistics summary for fake and real samples

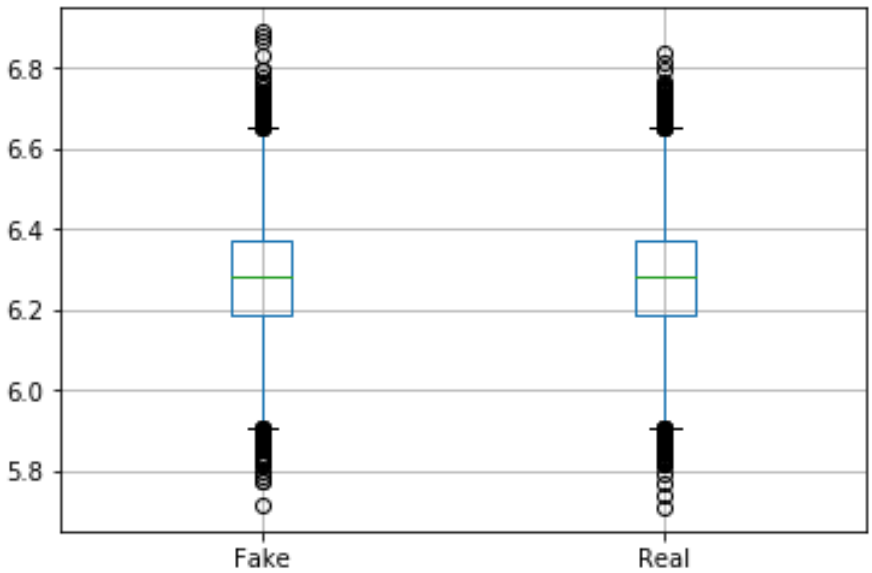


Fig. 5.7 Boxplot

5.4 Future research

In this section, we briefly mention some open problems that could lead to interesting future results.

5.4.1 Theoretical works

- Study theoretically the optimal choice for the optimizers (see Remark 3). Moreover, as mentioned by Ian Goodfellow gradient based optimizers were not designed to find the Nash equilibrium of a game. Updating the weights of the generator and discriminator networks independently does not guarantee the convergence of the algorithm towards the Nash equilibrium (see Chapter 3.6.1). Designing an algorithm that converges more reliably to the Nash equilibrium of a game is an open problem that would have great impact not only on GANs.

- In Chapter 3.8, we have discussed the advantages of using the Wasserstein distance instead of

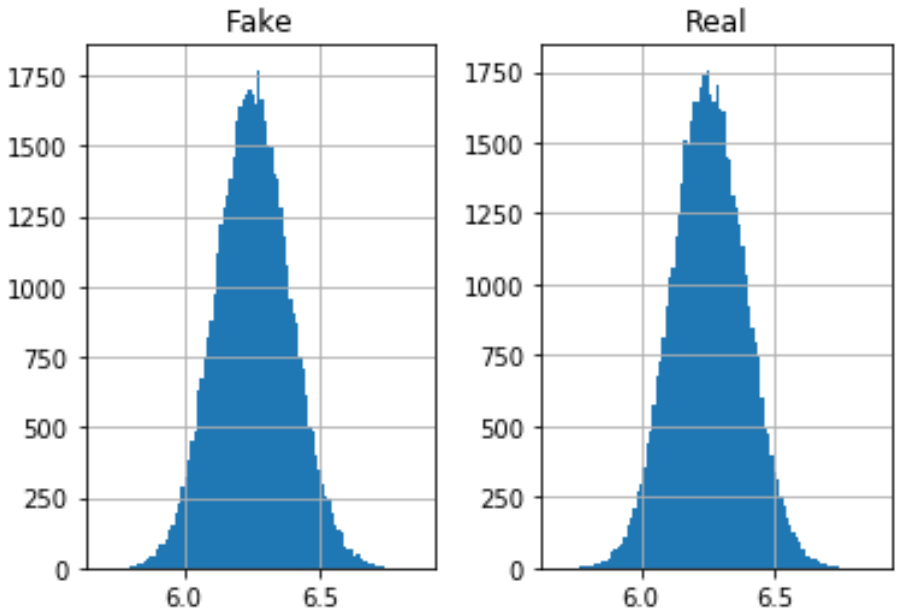


Fig. 5.8 Histogram

the Jenson-Shannon divergence to measure the similarity between the true training distribution and the implicit distribution defined by the generator as the distribution of the samples $G(z)$ for z drawn from the latent space. We have also seen that to train a Wasserstein GAN, we need to ensure a Lipschitz constraint over the training process. Unfortunately, there is no theoretical result yet on how to ensure such constraints on a recurrent neural network. If one could find a way to ensure the Lipschitz condition over the training of an RNN, we could define a Wasserstein Recurrent Generative Adversarial Network (WRGAN) by slightly changing the average cross-entropy into an average Wasserstein distance in the objective functions of the RGAN. The smoothness of the Wasserstein distance would probably lead to a much more stable training.

- To generate artificial time series we could have proceeded in a lots of different ways. For instance we could have used different time series prediction models to create predicted values from the observed historical values of the stock prices. Instead of using traditional time series forecasting methods, we could also train an RNN using LSTM cells to predict the evolution of the market. In some sense the RGAN approach proposed in this thesis can be seen as an alternative way of training the LSTM network. Instead of directly training the RNN, we train the network in a game theoretical setup by introducing an additional network (the discriminator). It would be interesting to compare these training methods both theoretically and experimentally.

5.4.2 Experimental works

- If we want to use the RGAN to replace usual Monte Carlo simulations, we should train the RGAN on real financial time series and not on an artificial dataset (see Remark 16).
- To obtain better convergence of the loss functions, we should tune the hyper-parameters of the networks using cross-validations as mentioned in Remark 17.
- It would be interesting to use the RGAN with multivariate time series. For example we might try to learn the RGAN on different stocks that are highly correlated and see whether the RGAN is able to learn the correlation structure of the training set. For instance we could use as a training set the stock prices of the Australian Foundation Investment Company (AFI.AX), Argo Investment Limited (ARG.AX) and Milton Corporation Limited (MLT.AX). All of these companies are listed investment companies that invest in similar Australian equities.

Appendix A

Python tutorials for training a GAN on sinusoidal curves

We first define a function *sample_data* to generate sinusoidal curves. We will use this function to create the training set of the GAN.

```
def sample_data(n_samples=10000, x_vals=np.arange(0, 5, .1),
    max_offset=100, mul_range=[1, 2]):
     vectors = []
     for i in range(n_samples):
         offset = np.random.random() * max_offset
         mul = mul_range[0] + np.random.random() * (mul_range[1] -
             mul_range[0])
         vectors.append(
             np.sin(offset + x_vals * mul) / 2 + .5
         )
     return np.array(vectors)
ax = pd.DataFrame(np.transpose(sample_data(5))).plot()
```

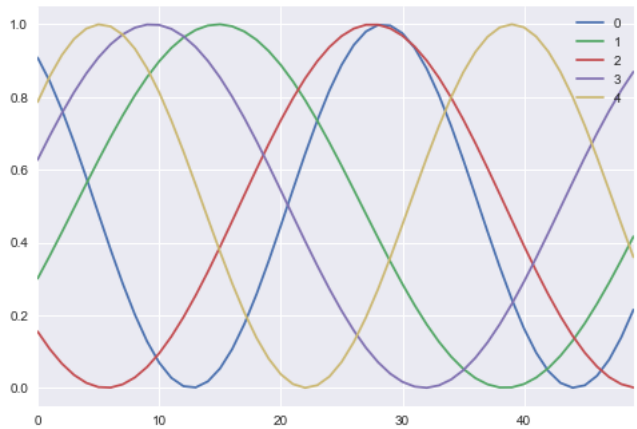


Fig. A.1 5 samples from the training set

We then define the generator network of the GAN. As discussed in Chapter 3, the generator will take random noise as input and will and try to generate sinusoidal curves from it. In this basic example, the generator is a feed-forward neural network containing one input layer, one fully connected dense hidden layer with *tanh* activation and one output layer. We also define the loss function of the generator and the optimiser that we are going to use to train the generator. In this case, we will use stochastic gradient descent with a learning rate of 0.001. The generator network will be trained only via the GAN by freezing the weights of the discriminator.

```
def get_generative(G_in, dense_dim=200, out_dim=50, lr=1e-3):
    x = Dense(dense_dim)(G_in)
    x = Activation('tanh')(x)
    G_out = Dense(out_dim, activation='tanh')(x)
    G = Model(G_in, G_out)
    opt = SGD(lr=lr)
    G.compile(loss='binary_crossentropy', optimizer=opt)
    return G, G_out

G_in = Input(shape=[10])
G, G_out = get_generative(G_in)
```

Similarly we create the discriminator network who will try to classify real and fake samples. In this example the discriminator consists of a convolutional neural network with two hidden layers. The first hidden layer is a convolutional layer using a ReLU activation function and a dropout rate of 0.25 to avoid overfitting. The second hidden layer is a fully connected layer. The output layer of the discriminator uses a sigmoid activation function to force the outputs to be between 0 and 1 (necessary since the output represents the probability that the input came from the true data distribution). We also define the loss function for the discriminator and the optimiser that we will use during the training of the discriminator. In this case we will use Adam optimiser. The discriminator will be trained directly.

```
def get_discriminative(D_in, lr=1e-3, drate=.25, n_channels=50,
   conv_sz=5, leak=.2):
    x = Reshape((-1, 1))(D_in)
    x = Conv1D(n_channels, conv_sz, activation='relu')(x)
    x = Dropout(drate)(x)
    x = Flatten()(x)
    x = Dense(n_channels)(x)
    D_out = Dense(2, activation='sigmoid')(x)
    D = Model(D_in, D_out)
    dopt = Adam(lr=lr)
    D.compile(loss='binary_crossentropy', optimizer=dopt)
    return D, D_out

D_in = Input(shape=[50])
D, D_out = get_discriminative(D_in)
```

We then merge the two networks into a GAN. The function *set_trainability* is used to freeze the weights of the discriminator while training the generator.

```
def set_trainability(model, trainable=False):
    model.trainable = trainable
    for layer in model.layers:
        layer.trainable = trainable

def make_gan(GAN_in, G, D):
    set_trainability(D, False)
    x = G(GAN_in)
    GAN_out = D(x)
    GAN = Model(GAN_in, GAN_out)
    GAN.compile(loss='binary_crossentropy', optimizer=G.optimizer)
    return GAN, GAN_out

GAN_in = Input([10])
GAN, GAN_out = make_gan(GAN_in, G, D)
```

Finally we train the GAN in the following way. First we pre-train the discriminator for one epoch using real and fake samples. The real samples (XT) come from our *sample_data* function defined earlier, while the fake samples are generated by the generator as follows: we first sample some random noise (*XN_noise*) and then use these latent variables as input of the generator network to create the fake samples (*XN*).

```
def sample_data_and_gen(G, noise_dim=10, n_samples=10000):
    XT = sample_data(n_samples=n_samples)
    XN_noise = np.random.uniform(0, 1, size=[n_samples, noise_dim])
    XN = G.predict(XN_noise)
    X = np.concatenate((XT, XN))
    y = np.zeros((2*n_samples, 2))
    y[:n_samples, 1] = 1
    y[n_samples:, 0] = 1
    return X, y

def pretrain(G, D, noise_dim=10, n_samples=10000, batch_size=32):
    X, y = sample_data_and_gen(G, n_samples=n_samples, noise_dim=
        noise_dim)
    set_trainability(D, True)
    D.fit(X, y, epochs=1, batch_size=batch_size)

pretrain(G, D)
```

Once the discriminator is pre trained, we train the GAN by alternating the training of the discriminator and the training of the generator. As mentionned above, the generator is trained by training the chained GAN model with discriminator's weights freezed.

```
def sample_noise(G, noise_dim=10, n_samples=10000):
    X = np.random.uniform(0, 1, size=[n_samples, noise_dim])
    y = np.zeros((n_samples, 2))
    y[:, 1] = 1
    return X, y
```

```
def train(GAN, G, D, epochs=500, n_samples=10000, noise_dim=10,
    batch_size=32, verbose=False, v_freq=50):
    d_loss = []
    g_loss = []
    e_range = range(epochs)
    if verbose:
        e_range = tqdm(e_range)
    for epoch in e_range:
        X, y = sample_data_and_gen(G, n_samples=n_samples,
            noise_dim=noise_dim)
        set_trainability(D, True)
        d_loss.append(D.train_on_batch(X, y))

        X, y = sample_noise(G, n_samples=n_samples, noise_dim=
            noise_dim)
        set_trainability(D, False)
        g_loss.append(GAN.train_on_batch(X, y))
        if verbose and (epoch + 1) % v_freq == 0:
            print("Epoch #{}: Generative Loss: {}, Discriminative
                Loss: {}".format(epoch + 1, g_loss[-1], d_loss[-1]))
    return d_loss, g_loss

d_loss, g_loss = train(GAN, G, D, verbose=True)
```

We can plot the evolution of the two networks losses as follows:

```
ax = pd.DataFrame(
    {
        'Generative Loss': g_loss,
        'Discriminative Loss': d_loss,
    }
).plot(title='Training loss', logy=True)
ax.set_xlabel("Epochs")
ax.set_ylabel("Loss")
```

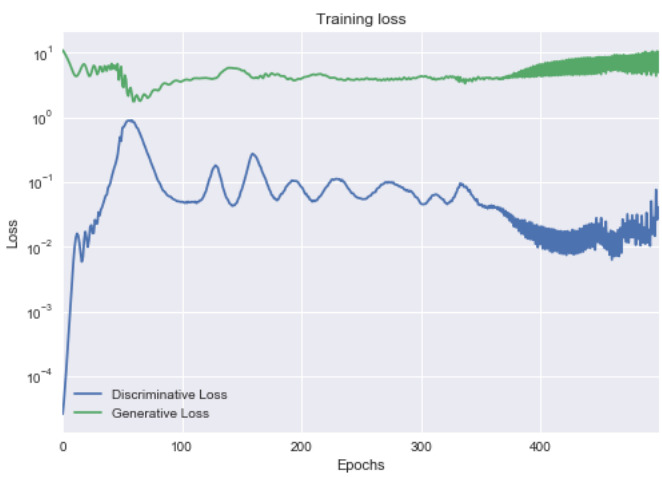


Fig. A.2 Losses of the two networks

Finally once the GAN is trained, we can drop the discriminator and use the generator to sample fake sinusiodal curves.

```
N_VIEWED_SAMPLES = 2
data_and_gen, _ = sample_data_and_gen(G, n_samples=N_VIEWED_SAMPLES
    )
pd.DataFrame(np.transpose(data_and_gen[N_VIEWED_SAMPLES:])).plot()
```

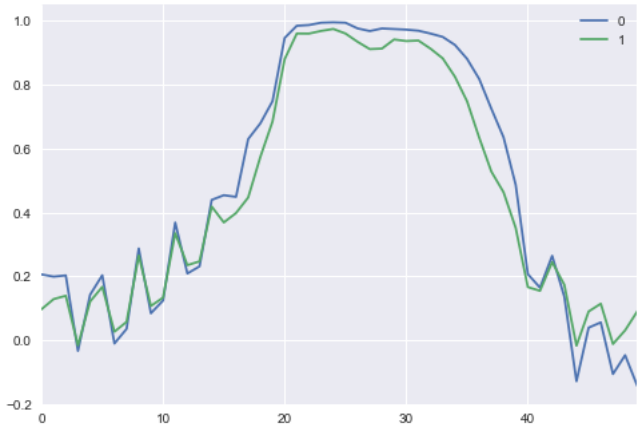


Fig. A.3 Samples generated by the generator after training

Appendix B

Python tutorial for training a GAN on MNIST

We first import TensorFlow and other important packages.

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import pandas as pd
import seaborn as sns
```

We then import the MNIST dataset.

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("../03-Convolutional-Neural-
    Networks/MNIST_data/",one_hot=True)
```

We can visualise an element of the dataset with the following command. The reshape function is used to reshape the vector data into a 28×28 picture.

```
fig, axes = plt.subplots(6, 10, figsize=(10, 2))
for i in range(6):
    for j in range(10):
        axes[i][j].imshow(mnist.train.images[np.random.randint(0,
            500)].reshape(28, 28), cmap = 'Blues', interpolation='
            nearest')
        axes[i][j].set_xticks([])
        axes[i][j].set_yticks([])
```

We then define the two networks as follows. The input layer of the generator network is a densely connected layer having 128 neurons. We use the Leaky Rectified Linear Unit (LeakyRelu) activation function for this first layer. The generator also has a densely connected hidden layer with 128 neurons. This hidden layer also uses the LeakyRelu activation function. Finally, the output layer of the generator

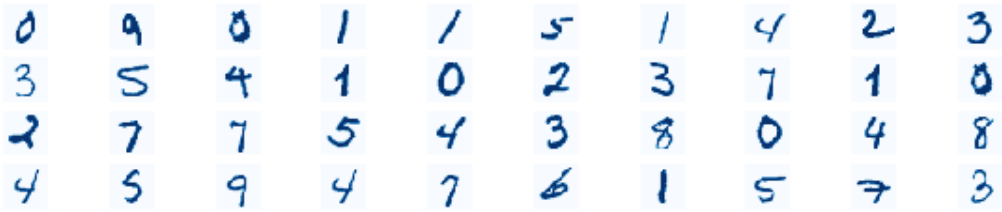


Fig. B.1 Samples from the MNIST dataset

is a fully connected layer with 784 neurons (since we want to output a 28×28 image). Experimentally, the *tanh* activation function seems to work well for the output layer.

The discriminator takes as input real or fake samples. The goal of the discriminator is to predict the probability that the input comes from the real data distribution. We will use the same architecture as for the generator with the only difference that the output layer will have only one neuron with sigmoid activation function to force the output to be between 0 and 1.

```
# Define the generator
def generator(z,reuse=None):
    with tf.variable_scope('gen',reuse=reuse):
        input_g = tf.layers.dense(inputs=z,units=128)
        # Leaky Relu
        alpha = 0.01
        input_g = tf.maximum(alpha*hidden1,hidden1)
        hidden1 = tf.layers.dense(inputs=input_g,units=128)

        hidden1 = tf.maximum(alpha*hidden1,hidden1)
        output = tf.layers.dense(hidden1,units=784,activation=tf.nn
            .tanh)
        return output

# Define the discriminator
def discriminator(X,reuse=None):
    with tf.variable_scope('dis',reuse=reuse):
        input_d = tf.layers.dense(inputs=X,units=128)
        # Leaky Relu
        alpha = 0.01
        input_d = tf.maximum(alpha*input_d,input_d)

        hidden1 = tf.layers.dense(inputs=input_d,units=128)
        hidden1 = tf.maximum(alpha*hidden1,hidden1)

        logits = tf.layers.dense(hidden1,units=1)
        output = tf.sigmoid(logits)

        return output, logits
```

Next we create two placeholders: one for the real images that will serve as input for the discrimi-

nator and one for the latent/noise sample that will serve as input for the generator. Here we will use $\mathbb{R}^{100}$ as our latent space.

```
real_images = tf.placeholder(tf.float32,shape=[None,784])
z = tf.placeholder(tf.float32,shape=[None,100])
```

We then define the outputs for the generator and the discriminator networks. Just as in the previous example, we will first pretrain the discriminator on the real images, so that the discriminator learns what the real dataset looks like. We will thus call the discriminator function twice: once on the real images and once on the fake generated samples. In the second case we need to set reuse=True so that we can reuse the same variable names twice.

```
# Generator output
G = generator(z)

# Discriminator output
D_output_real , D_logits_real = discriminator(real_images)
D_output_fake , D_logits_fake = discriminator(G,reuse=True)
```

We also have to define the loss functions for the two networks. When defining the loss function of the discriminator for real samples, we should pass *D_logits_real* and a vector of ones corresponding to the "real" label as input to our *loss_func* function. However we will use one sided label smoothing (as discussed in section 3.7.2) so instead of using a vector of ones, we are going to use a vector of 0.9's for the labels. We define the loss function of the discriminator for fake samples in a very similar way, but now the labels are 0. The total loss function of the discriminator is then defined as the sum of the losses for real and fake samples.

The generator is trained to fool the discriminator, so for *D_logits_fake* we will use a vector of one's as label.

```
def loss_func(logits_in,labels_in):
    return tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
        logits=logits_in,labels=labels_in))7

D_real_loss = loss_func(D_logits_real,tf.ones_like(D_logits_real)*
    (0.9))
D_fake_loss = loss_func(D_logits_fake,tf.zeros_like(D_logits_real))
D_loss = D_real_loss + D_fake_loss

G_loss = loss_func(D_logits_fake,tf.ones_like(D_logits_fake))
```

Finally we define the optimisers that we will use to train the networks. Both the discriminator and the generator will be updated using Adam optimiser with a learning rate of 0.001. To create the optimisers, we first create a list of the variable names (one for the generator and one for the discriminator). The function *tf.trainable_variables* returns the variables created.

The optimiser of the discriminator (*D_trainer*) is used to minimise the discriminator's loss with respect to the discriminator variables.

The generator's optimiser (*G_trainer*) minimises the generator loss with respect to the generator variables.

```
learning_rate = 0.001

tvars = tf.trainable_variables()

d_vars = [var for var in tvars if 'dis' in var.name]
g_vars = [var for var in tvars if 'gen' in var.name]

print([v.name for v in d_vars])
print([v.name for v in g_vars])

D_trainer = tf.train.AdamOptimizer(learning_rate).minimize(D_loss,
    var_list=d_vars)
G_trainer = tf.train.AdamOptimizer(learning_rate).minimize(G_loss,
    var_list=g_vars)
```

We then create a training session. We will train the GAN over 500 epochs with a batchsize of 100.

We first initialise all the variables with the *tf.trainable_variables* function. We will also create a saver function so that we can access the trained model after training. In the samples list, we will save one sample per epoch.

Next, we initialise the TensorFlow session inside which we will train the GAN. For every epoch, we calculate the number of batches needed to go through the training data once. Then for each batch, we reshape the images so that we can pass them to the discriminator. Next we create some random uniform noise using numpy. We then run the optimisers by using feed-dictionaries to feed in the placeholders created earlier.

Finally in each epoch we append the sample list with one generated sample. To create the generated sample, we first draw some random noise from a uniform distribution, and then use this latent noise as input of the trained generator to create the fake sample.

```
batch_size = 100
epochs = 500
init = tf.global_variables_initializer()
saver = tf.train.Saver(var_list=g_vars)

# Save a sample per epoch
samples = []

with tf.Session() as sess:

    sess.run(init)

    d_loss = []
    g_loss = []

    # Recall an epoch is an entire run through the training data
    for e in range(epochs):
```

```
        # // indicates classic division
        num_batches = mnist.train.num_examples // batch_size

        for i in range(num_batches):

            # Grab batch of images
            batch = mnist.train.next_batch(batch_size)

            # Get images, reshape and rescale to pass to D
            batch_images = batch[0].reshape((batch_size, 784))
            batch_images = batch_images*2 - 1

            # Z (random latent noise data for Generator)
            # -1 to 1 because of tanh activation
            batch_z = np.random.uniform(-1, 1, size=(batch_size,
                100))

            # Run optimizers, no need to save outputs, we won't use
                 them
            _ = sess.run(D_trainer, feed_dict={real_images:
                batch_images, z: batch_z})
            _ = sess.run(G_trainer, feed_dict={z: batch_z})
            g_loss.append(sess.run(G_loss, feed_dict={z: batch_z}))
            d_loss.append(sess.run(D_loss, feed_dict={real_images:
                batch_images, z: batch_z}))


        print("Currently on Epoch {} of {} total...".format(e+1,
            epochs))

        # Sample from generator as we're training for viewing
            afterwards
        sample_z = np.random.uniform(-1, 1, size=(1, 100))
        gen_sample = sess.run(generator(z ,reuse=True),feed_dict={z
            : sample_z})

        samples.append(gen_sample)
```

We can plot the evolution of the loss function using

```
ax = pd.DataFrame(
    {
        'Generative Loss': g_loss,
        'Discriminative Loss': d_loss,
    }
).plot(title='Training loss', logy=True)
ax.set_xlabel("Training iteration")
ax.set_ylabel("Loss")
```

Finally, after training, we use the generator to sample fake handwritten digits.

```
fig, axes = plt.subplots(6, 10, figsize=(10, 2))
for i in range(6):
    for j in range(10):
```

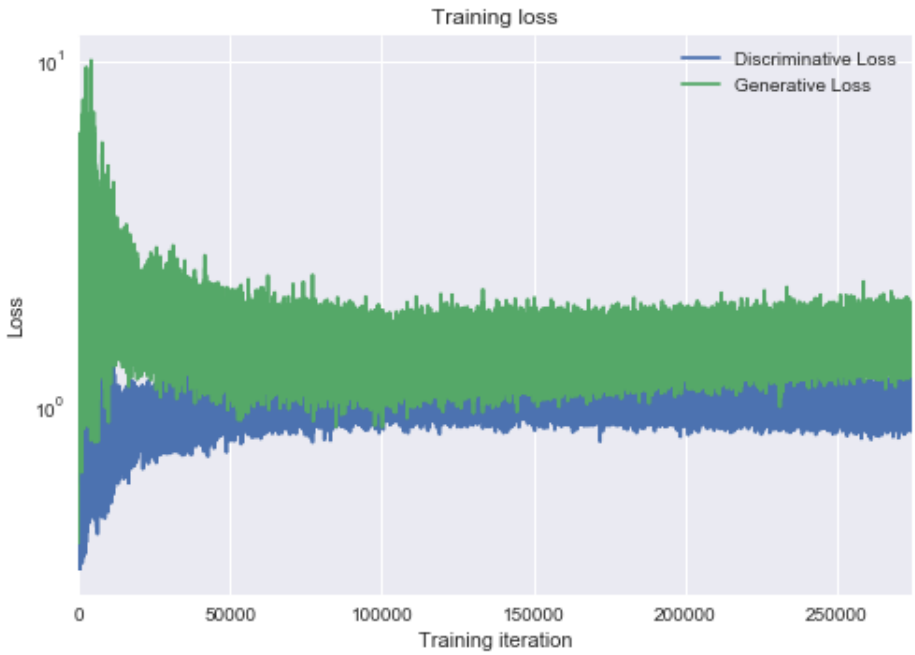


Fig. B.2 Evolution of the loss functionc

```
        axes[i][j].imshow(new_samples[np.random.randint(0, 500)].
            reshape(28, 28), cmap = 'Blues', interpolation='nearest'
            )
        axes[i][j].set_xticks([])
        axes[i][j].set_yticks([])
```

Fig. B.3 Generated samples after 500 epochs

Appendix C

Python tutorial for training a DCGAN on CIFAR-10

As always, we begin by importing the required libraries.

```
# Deep Convolutional GANs

# Importing the libraries
from __future__ import print_function
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
from torch.autograd import Variable
```

We then set the hyperparameters and create a variable called transform to make the images compatible with the inputs of the generator. We also import the CIFAR-10 dataset and create a function *weight_init* that takes as input a neural network and initializes all the weights of the network.

```
# Setting some hyperparameters
batchSize = 64 # We set the size of the batch.
imageSize = 64 # We set the size of the generated images (64x64).

# Creating the transformations
transform = transforms.Compose([transforms.Scale(imageSize),
    transforms.ToTensor(), transforms.Normalize((0.5, 0.5, 0.5),
    (0.5, 0.5, 0.5)),]) # We create a list of transformations (
    scaling, tensor conversion, normalization) to apply to the input
     images.

# Loading the dataset
```

```
dataset = dset.CIFAR10(root = './data', download = True, transform
    = transform) # We download the training set in the ./data folder
     and we apply the previous transformations on each image.
dataloader = torch.utils.data.DataLoader(dataset, batch_size =
    batchSize, shuffle = True, num_workers = 2) # We use dataLoader
    to get the images of the training set batch by batch.

# Defining the weights_init function that takes as input a neural
    network m and that will initialize all its weights.
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        m.weight.data.normal_(0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        m.weight.data.normal_(1.0, 0.02)
        m.bias.data.fill_(0)
```

We then define and create the generator network. The generator uses several transposed convolutional layers together with batch-normalisation and ReLU activation function. The final output layer uses a *tanh* activation.

```
# Defining the generator

class G(nn.Module): # We introduce a class to define the generator.
    We use inheritence from nn.Module

    def __init__(self): # We introduce the __init__() function that
        will define the architecture of the generator.
        super(G, self).__init__() # We inherit from the nn.Module
            tools.
        self.main = nn.Sequential( # We create a meta module of a
            neural network that will contain a sequence of modules (
            convolutions, full connections, etc.).
             nn.ConvTranspose2d(100, 512, 4, 1, 0, bias = False), #
                 We start with an inversed convolution.
             nn.BatchNorm2d(512), # We normalize all the features
                 along the dimension of the batch.
             nn.ReLU(True), # We apply a ReLU rectification to break
                 the linearity.
             nn.ConvTranspose2d(512, 256, 4, 2, 1, bias = False), #
                 We add another inversed convolution.
             nn.BatchNorm2d(256), # We normalize again.
             nn.ReLU(True), # We apply another ReLU.
             nn.ConvTranspose2d(256, 128, 4, 2, 1, bias = False), #
                 We add another inversed convolution.
             nn.BatchNorm2d(128), # We normalize again.
             nn.ReLU(True), # We apply another ReLU.
             nn.ConvTranspose2d(128, 64, 4, 2, 1, bias = False), #
                 We add another inversed convolution.
             nn.BatchNorm2d(64), # We normalize again.
             nn.ReLU(True), # We apply another ReLU.
```

```
            nn.ConvTranspose2d(64, 3, 4, 2, 1, bias = False), # We
                add another inversed convolution.
            nn.Tanh() # We apply a Tanh rectification to break the
                linearity and stay between -1 and +1.
        )

    def forward(self, input): # We define the forward function that
         takes as argument an input that will be fed to the neural
        network, and that will return the output containing the
        generated images.
        output = self.main(input) # We forward propagate the signal
             through the whole neural network of the generator
            defined by self.main.
        return output # We return the output containing the
            generated images.

# Creating the generator
netG = G() # We create the generator object.
netG.apply(weights_init) # We initialize all the weights of its
    neural network.
```

We also define the discriminator network. The discriminator uses several convolutional hidden layers with batch-normalisation and leaky rectified linear unit activation functions. The output layer of the discriminator uses a sigmoid activation (since we want to predict the probability that the input comes from the true data distribution p_{data}).

```
# Defining the discriminator

class D(nn.Module): # We introduce a class to define the
    discriminator.

    def __init__(self): # We introduce the __init__() function that
         will define the architecture of the discriminator.
        super(D, self).__init__() # We inherit from the nn.Module
            tools.
        self.main = nn.Sequential( # We create a meta module of a
            neural network that will contain a sequence of modules (
            convolutions, full connections, etc.).
            nn.Conv2d(3, 64, 4, 2, 1, bias = False), # We start
                with a convolution.
            nn.LeakyReLU(0.2, inplace = True), # We apply a
                LeakyReLU.
            nn.Conv2d(64, 128, 4, 2, 1, bias = False), # We add
                another convolution.
            nn.BatchNorm2d(128), # We normalize all the features
                along the dimension of the batch.
            nn.LeakyReLU(0.2, inplace = True), # We apply another
                LeakyReLU.
            nn.Conv2d(128, 256, 4, 2, 1, bias = False), # We add
                another convolution.
            nn.BatchNorm2d(256), # We normalize again.
```

```
            nn.LeakyReLU(0.2, inplace = True), # We apply another
                LeakyReLU.
            nn.Conv2d(256, 512, 4, 2, 1, bias = False), # We add
                another convolution.
            nn.BatchNorm2d(512), # We normalize again.
            nn.LeakyReLU(0.2, inplace = True), # We apply another
                LeakyReLU.
            nn.Conv2d(512, 1, 4, 1, 0, bias = False), # We add
                another convolution.
            nn.Sigmoid() # We apply a Sigmoid rectification to
                break the linearity and stay between 0 and 1.
        )

    def forward(self, input): # We define the forward function that
         takes as argument an input that will be fed to the neural
        network, and that will return the output which will be a
        value between 0 and 1.
        output = self.main(input) # We forward propagate the signal
             through the whole neural network of the discriminator
            defined by self.main.
        return output.view(-1) # We return the output which will be
             a value between 0 and 1.

# Creating the discriminator
netD = D() # We create the discriminator object.
netD.apply(weights_init) # We initialize all the weights of its
    neural network.
```

We then define the optimisation criterion and the optimisers for both networks. The BCELoss stands for Binary Cross Entropy and corresponds to the objective function defined in section 3.2. In this example we will use Adam optimiser for both the discriminator and the generator.

```
# Define the optimisation criterion and the optimisers

criterion = nn.BCELoss() # We create a criterion object that will
    measure the error between the prediction and the target.
optimizerD = optim.Adam(netD.parameters(), lr = 0.0002, betas =
    (0.5, 0.999)) # We create the optimizer object of the
    discriminator.
optimizerG = optim.Adam(netG.parameters(), lr = 0.0002, betas =
    (0.5, 0.999)) # We create the optimizer object of the generator.
```

Finally we train the DCGAN for 25 epochs. For each epoch, the training can be decomposed into three steps. First pre-train the discriminator network. To do this we first need to pre-train the discriminator on real samples so that it can learn how the true samples look like. Then we pre-train the discriminator on fake images as well. We compute the loss of the discriminator on both the real and the fake samples and we define the total loss of the discriminator as the sum of these two losses. This total loss is then used to update the discriminator by backpropagating the errors using Adam optimiser.

```
# Training the DCGANs

for epoch in range(25): # We iterate over 25 epochs.

    for i, data in enumerate(dataloader, 0): # We iterate over the
        images of the dataset.

        netD.zero_grad() # We initialize to 0 the gradients of the
            discriminator with respect to the weights.

        # Training the discriminator with a real image of the
            dataset
        real, _ = data # We get a real image of the dataset which
            will be used to train the discriminator.
        input = Variable(real) # We wrap it in a variable.
        target = Variable(torch.ones(input.size()[0])) # We get the
            target.
        output = netD(input) # We forward propagate this real image
            into the neural network of the discriminator to get the
            prediction (a value between 0 and 1).
        errD_real = criterion(output, target) # We compute the loss
            between the predictions (output) and the target (equal
            to 1).

        # Training the discriminator with a fake image generated by
            the generator
        noise = Variable(torch.randn(input.size()[0], 100, 1, 1)) #
            We make a random input vector (noise) of the generator.
        fake = netG(noise) # We forward propagate this random input
            vector into the neural network of the generator to get
            some fake generated images.
        target = Variable(torch.zeros(input.size()[0])) # We get
            the target.
        output = netD(fake.detach()) # We forward propagate the
            fake generated images into the neural network of the
            discriminator to get the prediction (a value between 0
            and 1).
        errD_fake = criterion(output, target) # We compute the loss
            between the prediction (output) and the target (equal
            to 0).

        # Backpropagating the total error
        errD = errD_real + errD_fake # We compute the total error
            of the discriminator.
        errD.backward() # We backpropagate the loss error by
            computing the gradients of the total error with respect
            to the weights of the discriminator.
        optimizerD.step() # We apply the optimizer to update the
            weights according to how much they are responsible for
            the loss error of the discriminator.
```

In the second step, we update the weights of the generator network. To do this we first forward

propagate the generated (fake) sample through the discriminator network. This gives us a prediction of the probability that the fake sample came from the true distribution. Using this prediction and the generator's target (equal to 1), we calculate the loss and backpropagate it through the network by computing the gradients of the total error with respect to the weights of the generator using Adam optimiser.

```
        output = netD(fake) #forward propagation (gives a
            prediction between 0 and 1) )
        errG = criterion(output, target) # We compute the loss
            between the prediction and the target
        errG.backward() # We backpropagate the loss error
        optimizerG.step() # update the weights
```

Finally in the third and last step, we save the real and the generated samples of the minibatch every 100 steps.

```
            vutils.save_image(real, '%s/real_samples.png' % "./
                results", normalize = True) # We save the real
                images of the minibatch.
            fake = netG(noise) # We get our fake generated images.
            vutils.save_image(fake.data, '%s/fake_samples_epoch_%03
                d.png' % ("./results", epoch), normalize = True) #
                We also save the fake generated images of the
                minibatch.
```

Appendix D

Python tutorial for Monte Carlo simulation

This Appendix contains the Monte Carlo simulation explained in Chapter 5.1.1 (Source of the code: Python for Finance blog: http://www.pythonforfinance.net/2016/11/28/monte-carlo-simulation-in-python/).

```
#import necessary packages
import numpy as np
import math
import matplotlib.pyplot as plt
# from scipy.stats import norm
from pandas_datareader import data

#download Apple price data into DataFrame
apple = data.DataReader('AAPL', 'yahoo',start='1/1/2000')

#### ESTIMATION OF THE MEAN AND VOLATILITY #####

#calculate the compound annual growth rate (CAGR) which
#will give us our mean return input (mu)
days = (apple.index[-1] - apple.index[0]).days
cagr = ((((apple['Adj Close'][-1]) / apple['Adj Close'][1])) **
    (365.0/days)) - 1
print ('CAGR =',str(round(cagr,4)*100)+"%")
mu = cagr
#create a series of percentage returns and calculate the annual
    volatility of returns
apple['Returns'] = apple['Adj Close'].pct_change()
vol = apple['Returns'].std()*math.sqrt(252)
print ("Annual Volatility =",str(round(vol,4)*100)+"%")

#### MONTE CARLO ####

#Define Variables
```

```
S = apple['Adj Close'][-1] #starting stock price (i.e. last
    available real stock price)
T = 252 #Number of trading days
mu = 0.2309 #Return
vol = 0.4259 #Volatility

for i in range(1000):
    #create list of daily returns using random normal distribution
    daily_returns=np.random.normal(mu/T,vol/math.sqrt(T),T)+1

    #set starting price and create price series generated by above
        random daily returns
    price_list = [S]

    for x in daily_returns:
        price_list.append(price_list[-1]*x)

    #plot data from each individual run which we will plot at the
        end
    plt.plot(price_list)

#show the plot of multiple price series created above
plt.show()
```

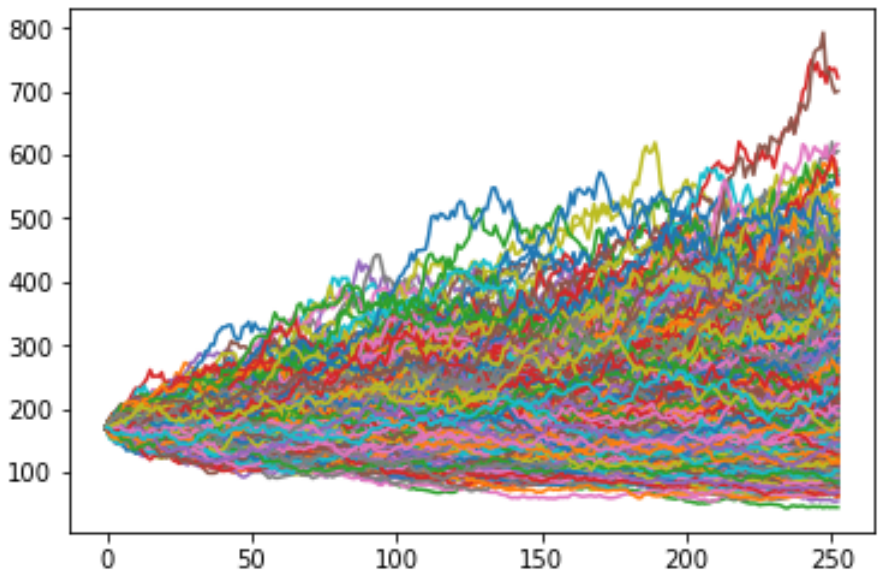


Fig. D.1 Monte Carlo simulation of the future price evolution of the Apple stock price

Appendix E

Tutorial for the implementation of the RGAN

In this section we briefly discuss the RGAN implementation. First we import the required libraries

```
#Imports
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
#%matplotlib inline
from pandas_datareader import data
import tensorflow as tf
#from tensorflow.contrib.rnn import LSTMCell
tf.logging.set_verbosity(tf.logging.ERROR)
from time import time
from matplotlib.colors import hsv_to_rgb
from scipy.stats import ks_2samp
```

We then download the historical prices of the AFI.AX and simulate artificial log-returns via Monte Carlo simulation under the assumption that the daily returns follow a Gaussian distribution with mean and volatility parameters estimated on the historical prices. These artificial log-returns will be used as training set for the RGAN.

```
# STEP 1 SIMULATE LOG RETURNS VIA MONTE CARLO SIMULATIONS (GENERATE
    SOME THE DATA USED TO LEARN THE GAN)
# ESTIMATION OF THE MEAN AND VOLATILITY

T = 9 #Number of trading days

#download Apple price data into DataFrame
apple = data.DataReader('AFI.AX', 'yahoo',start='1/1/2013')

#calculate the compound annual growth rate (CAGR) which
#will give us our mean return input (mu)
days = (apple.index[-1] - apple.index[0]).days
```

```
cagr = ((((apple['Adj Close'][-1]) / apple['Adj Close'][1])) **
    (365.0/days)) - 1
print ('CAGR =',str(round(cagr,4)*100)+"%")
mu = cagr

#create a series of percentage returns and calculate the annual
    volatility of returns
apple['Returns'] = apple['Adj Close'].pct_change()
vol = apple['Returns'].std()*math.sqrt(T)
print ("Annual Volatility =",str(round(vol,4)*100)+"%")


# GENERATION OF THE DATA USED TO TRAIN THE GAN

#Define Variables
S = apple['Adj Close'][-1] #starting stock price (i.e. last
    available real stock price)

n_samples = 50000
seq_length = 10 # change line 202 and 227 as well

#log returns
def sample_data(n_samples=n_samples):
    vectors = []
    ###### MONTE CARLO #######
    for i in range(n_samples):

        #create list of daily returns using random normal
            distribution
        daily_returns=np.random.normal(mu/T,vol/math.sqrt(T),
            seq_length) #T+1 since we work with the vectors = [] and
             do not initilase it by [S]
        vectors.append(np.log(daily_returns+1))

    dataset = np.array(vectors)
    #dataset = np.expand_dims(dataset, axis=0)
    #dataset = dataset.reshape(38, 253, 1)

    dataset.reshape(-1, seq_length, 1)

    return dataset
```

We then choose the hyper parameters of the RGAN. As mentionned above, here we only tested some parameters but the parameters should be chosen via Cross-Validations (not done for computational power reasons).

```
learning_rate = 0.001
lr = learning_rate
batch_size = 28
latent_dim = 20
num_epochs = 50
vis_freq = 2
```

```
labels = None
D_rounds = 3
G_rounds = 1
hidden_units_g = 150
hidden_units_d = 150
num_generated_features = 1
```

We then initialize the TensorFlow placeholders. For more details about TensorFlow, we recommend the CS20 Tensorflow for Deep Learning Research course from Stanford University.

```
CG = tf.placeholder(tf.float32, [batch_size, seq_length]) #
    Placeholder 0 (shape: (50,253)) is it 0 or seq_length
CD = tf.placeholder(tf.float32, [batch_size, seq_length]) #
    Placerholder 1
Z = tf.placeholder(tf.float32, [batch_size, seq_length, latent_dim
    ]) #Placeholder 2
W_out_G = tf.Variable(tf.truncated_normal([hidden_units_g,
    num_generated_features]))
b_out_G = tf.Variable(tf.truncated_normal([num_generated_features])
    )

X = tf.placeholder(tf.float32, [batch_size, seq_length,
    num_generated_features])
W_out_D = tf.Variable(tf.truncated_normal([hidden_units_d,1]))
b_out_D = tf.Variable(tf.truncated_normal([1]))
```

Next we define a function $sample_z$ which enables us to sample the latent variables from a normal distribution.

```
def sample_Z(batch_size, seq_length, latent_dim):
    sample = np.float32(np.random.normal(size=[batch_size,
        seq_length, latent_dim]))
    if use_time:
        print('WARNING: use_time has different semantics')
        sample[:, :, 0] = np.linspace(0, 1.0/seq_length, num=
            seq_length)

    return sample
```

We then define the generator and discriminator networks of the RGAN using LSTM cells. This is probably the most important part of the RGAN, since this is where we introduce Recurrent Neural Networks (LSTM cells) that are particularly adapted for time series (see Chapter 1).

```
def generator(z, c):
    with tf.variable_scope("generator") as scope:

        # each step of the generator takes a random seed + the
            conditional embedding
        repeated_encoding = tf.tile(c,[1, tf.shape(z)[1]])
        repeated_encoding = tf.reshape(repeated_encoding, [tf.shape
            (z)[0], tf.shape(z)[1],10])
        generator_input = tf.concat([repeated_encoding, z], 2)
```

```
        cell = tf.contrib.rnn.LSTMCell(num_units=hidden_units_g,
            state_is_tuple=True)
        rnn_outputs, rnn_states = tf.nn.dynamic_rnn(
                cell=cell,
                dtype=tf.float32,
                sequence_length=[seq_length]*batch_size,
                inputs=generator_input)
        rnn_outputs_2d = tf.reshape(rnn_outputs, [-1,
            hidden_units_g])
        logits_2d = tf.matmul(rnn_outputs_2d, W_out_G) + b_out_G
        output_2d = tf.nn.tanh(logits_2d)
        output_3d = tf.reshape(output_2d, [-1, seq_length,
            num_generated_features])
    return output_3d


def discriminator(x, c, reuse=False):
    with tf.variable_scope("discriminator") as scope:
        if reuse:
            scope.reuse_variables()

        # each step of the generator takes one time step of the
            signal to evaluate +
        # its conditional embedding
        repeated_encoding = tf.tile(c,[1, tf.shape(x)[1]])
        repeated_encoding = tf.reshape(repeated_encoding, [tf.shape
            (x)[0], tf.shape(x)[1],10])
        decoder_input = tf.concat([repeated_encoding, x], 2)

        cell = tf.contrib.rnn.LSTMCell(num_units=hidden_units_d,
            state_is_tuple=True)
        rnn_outputs, rnn_states = tf.nn.dynamic_rnn(
                cell=cell,
                dtype=tf.float32,
                inputs=decoder_input)
        rnn_outputs_flat = tf.reshape(rnn_outputs, [-1,
            hidden_units_g])
        logits = tf.matmul(rnn_outputs_flat, W_out_D) + b_out_D
        output = tf.nn.sigmoid(logits)
    return output, logits
```

Next we define the loss functions for the generator and the discriminator of the RGAN as discussed in Section 3.1.

```
G_sample = generator(Z, CG)
D_real, D_logit_real = discriminator(X, CD)
D_fake, D_logit_fake = discriminator(G_sample, CG, reuse=True)

generator_vars = [v for v in tf.trainable_variables() if v.name.
    startswith('generator')]
discriminator_vars = [v for v in tf.trainable_variables() if v.name
    .startswith('discriminator')]
```

```
D_loss_real = tf.reduce_mean(tf.nn.
    sigmoid_cross_entropy_with_logits(logits=D_logit_real,

    labels=tf.ones_like(D_logit_real)))

D_loss_fake = tf.reduce_mean(tf.nn.
    sigmoid_cross_entropy_with_logits(logits=D_logit_fake,
labels=tf.zeros_like(D_logit_fake)))

D_loss = D_loss_real + D_loss_fake

G_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
    logits=D_logit_fake,
labels=tf.ones_like(D_logit_fake)))
```

We then choose the optimisers for the two networks. We have tried different combinations but stochastic gradient descent for the discriminator and Adam optimiser for the generator seemed to work the best. Again this should be studied in more details and is left for a future work.

```
D_solver = tf.train.GradientDescentOptimizer(learning_rate=lr).
    minimize(D_loss, var_list=discriminator_vars)
#D_solver = tf.train.AdamOptimizer().minimize(D_loss, var_list=
    generator_vars)
#G_solver = tf.train.GradientDescentOptimizer(learning_rate=lr).
    minimize(G_loss, var_list=discriminator_vars)
G_solver = tf.train.AdamOptimizer().minimize(G_loss, var_list=
    generator_vars)
```

Finally we define the training process as explained in Chapter 3.

```
# Train the GAN on the Monte Carlo simulations

# Starting a tensorflow session

sess = tf.Session()
sess.run(tf.global_variables_initializer())
vis_Z = sample_Z(batch_size, seq_length, latent_dim)


t0 = time()

def train_generator(batch_idx, offset):
    # update the generator
    for g in range(G_rounds):
        #X_mb = get_batch(samples, batch_size, batch_idx + g +
            offset)
        Y_mb = get_batch(samples, batch_size, batch_idx + g +
            offset)
        _, G_loss_curr = sess.run([G_solver, G_loss],
                                  feed_dict={CG:Y_mb,
                                             Z: sample_Z(batch_size
                                                , seq_length,
                                                latent_dim)})
```

```
    return G_loss_curr


def train_discriminator(batch_idx, offset):
    # update the discriminator
    for d in range(D_rounds):
    # using same input sequence for both the synthetic data and the
        real one,
    # probably it is not a good idea...
        X_mb = get_batch(samples, batch_size, batch_idx + d +
           offset)
        X_mb = X_mb.reshape(batch_size,seq_length,1)
        Y_mb = get_batch(samples, batch_size, batch_idx + d +
           offset)
        _, D_loss_curr = sess.run([D_solver, D_loss],
                                  feed_dict={CD:Y_mb, CG:Y_mb, X:
                                     X_mb,
                                             Z: sample_Z(batch_size
                                                , seq_length,
                                                latent_dim)})
    return D_loss_curr
```

We are then finally ready to train the RGAN.

```
print('num_epoch\t D_loss_curr\t G_loss_curr\t time elapsed')

d_loss = []
g_loss = []
for num_epoch in range(num_epochs):

    for batch_idx in range(0, int(len(samples)/batch_size) - (
       D_rounds + G_rounds), D_rounds + G_rounds):

        if num_epoch % 2 == 0:

            G_loss_curr = train_generator(batch_idx,0)
            D_loss_curr = train_discriminator(batch_idx, G_rounds)

        else:

            D_loss_curr = train_discriminator(batch_idx, 0)
            G_loss_curr = train_generator(batch_idx, D_rounds)

        d_loss.append(D_loss_curr)
        g_loss.append(G_loss_curr)
        t = time() -t0


        print(num_epoch,'\t', D_loss_curr, '\t', G_loss_curr, '\t',
            t)

    # save synthetic data
    if num_epoch % 5 == 0:
```

```
    # generate synthetic dataset
        gen_samples = []
        for batch_idx in range(int(len(samples) / batch_size)):
            X_mb = get_batch(samples, batch_size, batch_idx)
            Y_mb = get_batch(samples, batch_size, batch_idx)
            z_ = sample_Z(batch_size, seq_length, latent_dim)
            gen_samples_mb = sess.run(G_sample, feed_dict={Z: z_,
                CG:Y_mb})
            gen_samples.append(gen_samples_mb)
            print (batch_idx)

        gen_samples = np.vstack(gen_samples)
```

We can plot the evolution of the training losses for both network.

```
ax = pd.DataFrame(
    {
        'Generative Loss': g_loss,
        'Discriminative Loss': d_loss,
    }
).plot(title='Training loss', logy=True)
ax.set_xlabel("Training iteration")
ax.set_ylabel("Loss")
```

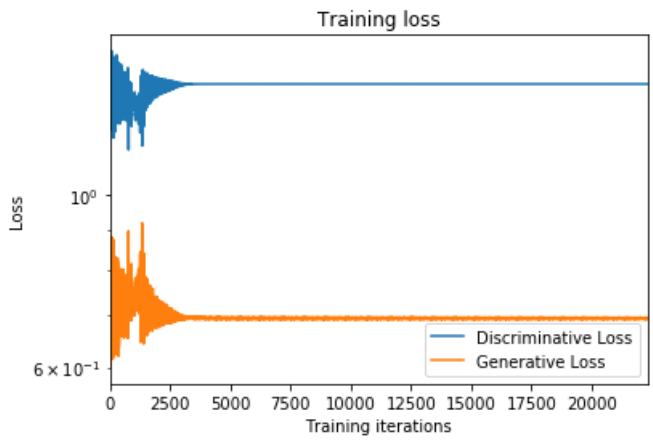


Fig. E.1 Losses of the two networks

Now that we have trained our RGAN, let's compare some fake (RGAN generated) and real (Monte Carlo generated) samples. To get the prices from the log-returns we need to define some functions.

```
generated_data = np.transpose(gen_samples)
#plot the log-returns
gen_ind = 1 # change in function price as well
pd.DataFrame(generated_data[0,:,gen_ind]).plot()  # 1 is the index
    of the plotted sample out of the 1000 generated

# get the prices from the log returns
def price_gen(ind_gen_sample = 1):
```

```
    daily_log_returns=generated_data[0,:,ind_gen_sample]

    price_list = [S] #initial price (today)

    for x in np.transpose(daily_log_returns):

        price_list.append(price_list[-1]*np.exp(x))

    #price_list = np.asarray(price_list)

    return price_list

def price_real():

    daily_returns=np.random.normal(mu/T,vol/math.sqrt(T),T)+1

    price_list = [S]

    for x in daily_returns:

        price_list.append(price_list[-1]*x)

    return price_list


def prices_gen_data_frame(num_samples = 1):
    df = pd.DataFrame([])

    for i in range(num_samples):

        df[i] = price_real()

    return df

def prices_real_data_frame(num_samples = 1):
    df = pd.DataFrame()

    for i in range(num_samples):

        df[i] = price_real()

    return df

# Plot the price evolution of the generated sample
def plot_price_gen(num_gen_sample = 1):

    for i in range(num_gen_sample):
        daily_log_returns = generated_data[0,:,i]

        price_list = [S]
```

```
        for x in daily_log_returns:
            price_list.append(price_list[-1]*np.exp(x))

        plt.plot(price_list)

    return plt.show()

# Plot one real evolution
def plot_price_real(num_gen_samples = 1):

    for i in range(num_gen_samples):
        #create list of daily returns using random normal
            distribution
        daily_returns=np.random.normal(mu/T,vol/math.sqrt(T),T)+1

        #set starting price and create price series generated by
            above random daily returns
        price_list = [S]

        for x in daily_returns:
            price_list.append(price_list[-1]*x)

        #plot data from each individual run which we will plot at
            the end
        plt.plot(price_list)
        #pd.DataFrame(price_list).plot() #(if we want plots on
            separate figures)

    return plt.show() # return None (if we want plots on separate
        figures)
```

To get a qualitative idea of whether the RGAN has learned to generate fake time series, we plot 20 000 fake and 20 000 real price evolutions of the AFI.AX over the next 10 days using the functions defined above.

```
plot_price_gen(20000)
plot_price_real(20000)
```

Statistical Analysis of the generated samples

Finally we study statistically whether the predicted prices for the 10th day generated by the RGAN follows a similar distribution to the same quantity generated by Monte Carlo.

```
def transform2darray(vector):
    v = []
    for i in range(len(vector)):
        v.append(vector[i][0])
    return v

num_sample = 50000
result_gen = prices_gen_data_frame(num_sample).iloc[[seq_length-1]]
```

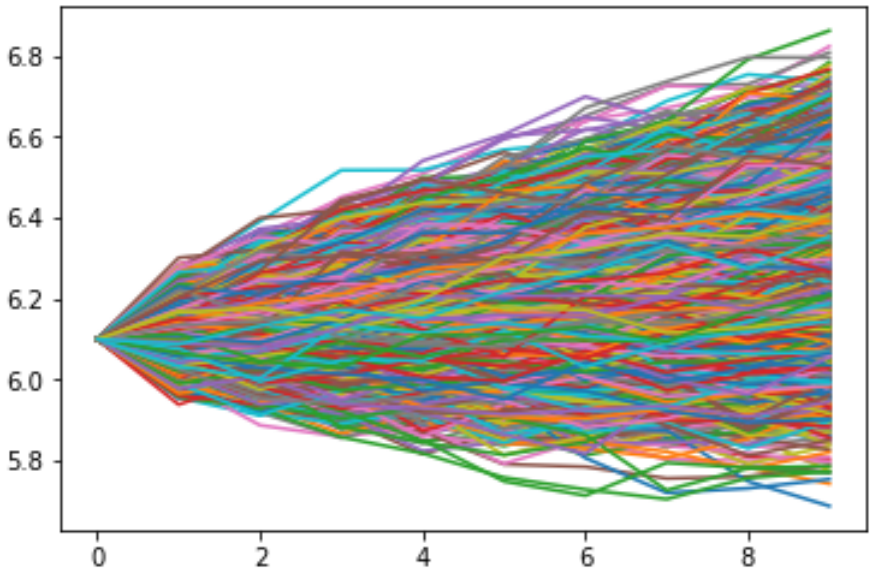


Fig. E.2 Real samples

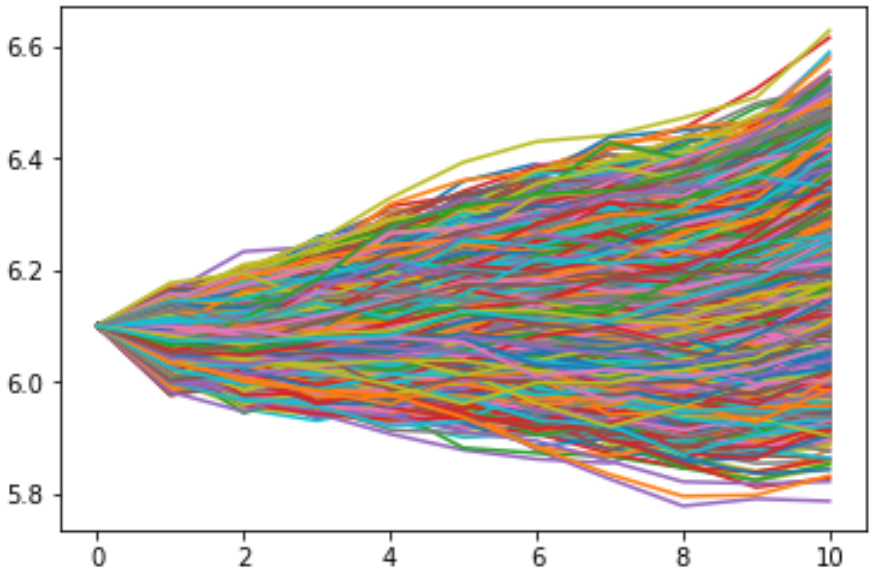


Fig. E.3 Fake samples

```
result_gen = np.asarray(result_gen)
result_gen = np.transpose(result_gen)
result_gen = transform2darray(result_gen)
result_real = prices_real_data_frame(num_sample).iloc[[seq_length
    -1]]
result_real = np.asarray(result_real)
result_real = np.transpose(result_real)
result_real = transform2darray(result_real)
```

```
results = pd.DataFrame({'Real':result_real, 'Fake': result_gen})
```

We can also print out statistics summaries and plot boxplots and histograms.

```
# statistics summary
print(results.describe())
# boxplot
results.boxplot()
results.hist(bins=100)
```

```
In [4]: print(results.describe())
                Fake          Real
count  50000.000000  50000.000000
mean       6.255305      6.255051
std        0.137163      0.138182
min        5.698330      5.681760
25%        6.162344      6.161181
50%        6.254092      6.253195
75%        6.346743      6.348192
max        6.864863      6.880436
```

Fig. E.4 Statistics summary for fake and real samples

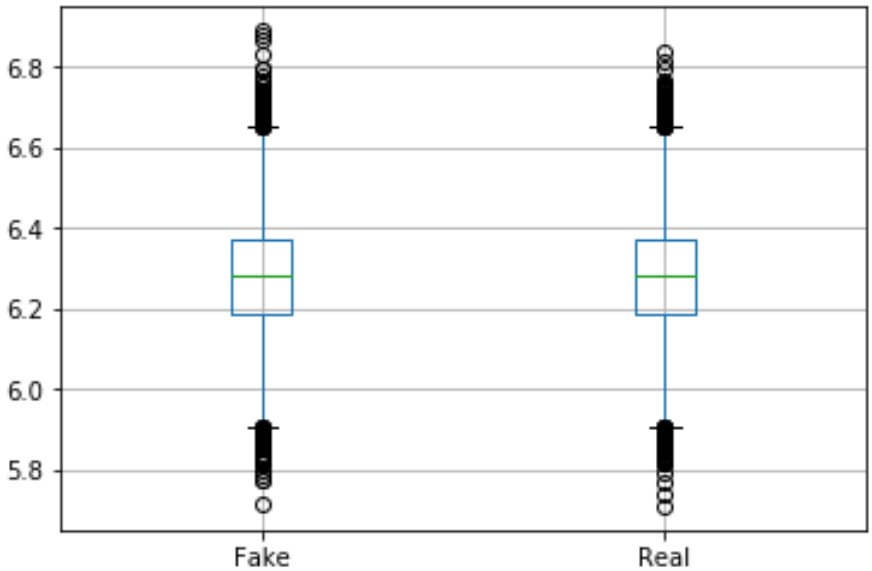


Fig. E.5 Boxplot

We finally conduct a Kolmogorov Smirnov test to test the null hypothesis that the two samples are drawn from the same distribution.

```
value, pvalue = ks_2samp(result_real, result_gen)
print("test statistic =", value, "  ", "pvalue = ", pvalue)
if pvalue > 0.05:
    print('Samples are likely drawn from the same distributions (
        not reject H0)')
```

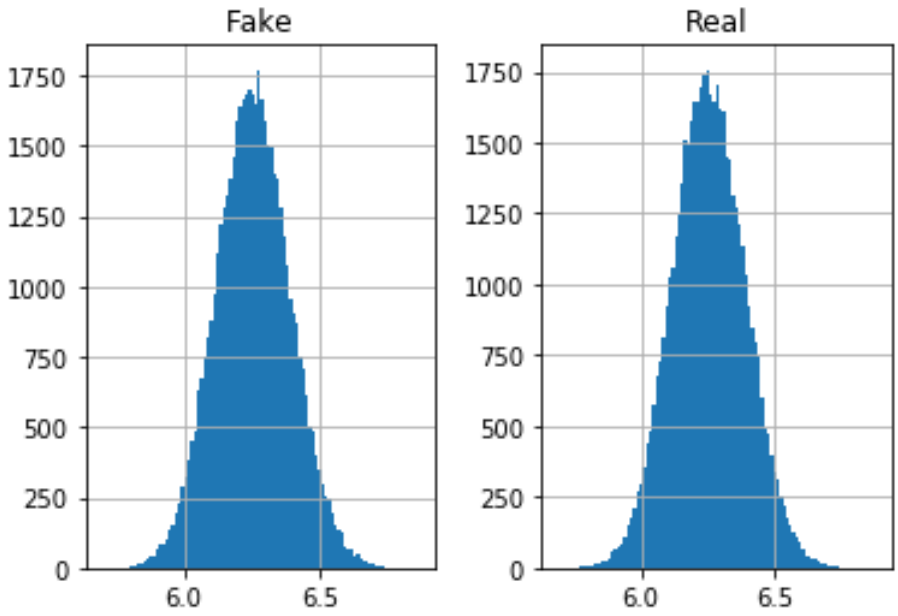


Fig. E.6 Histogram

```
else:
    print('Samples are likely drawn from different distributions (
        reject H0)')
```

Hopefully the test does not reject H0 at level 5% and thus we can conclude that samples are likely drawn from the same distributions.

References

[1] *Complete Guide to TensorFlow for Deep Learning with Python Jose Portilla*.

[2] *Deep Learning A-Z: Hands-On Artificial Neural Networks Kirill Eremenko*.

[3] *Deep Learning and Computer Vision Hadelin de Ponteves and Kirill Eremenko*.

[4] *Lilian Weng: From GAN to WGAN*.

[5] *Scott Rome: An Annotated Proof of Generative Adversarial Networks with Implementation Notes*.

[6] *Statistical Learning in Practice Tengyao Wang*.

[7] *Understanding LSTM Networks http://colah.github.io/posts/2015-08-Understanding-LSTMs/*.

[8] Martin Arjovsky an Leon Bottou. Towards principled methods for training generative adversarial networks. *arXiv:1701.04862*, 2017.

[9] Hernandez Andres. Model calibration with neural networks. *SSRN: https://ssrn.com/abstract=2812140 or http://dx.doi.org/10.2139/ssrn.2812140*, 2016.

[10] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan. *arXiv:1701.07875*, 2017.

[11] Sebastian Becker, Patrick Cheridito, and Arnulf Jentzen. Deep optimal stopping. *arXiv:1804.05394*, 2018.

[12] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE TRANSACTIONS ON NEURAL NETWORKS, VOL. 5, NO. 2, MARCH 1994*, 1994.

[13] Hans Bühler, Lukas Gonon, Josef Teichmann, and Ben Wood. Deep hedging. *arXiv:1802.03042*, 2018.

[14] Christa Cuchiero, Irene Klein, and Josef Teichmann. Bayesian finance. In *https://people.math.ethz.ch/ jteichma/talk$_o$bergurgl*2017.*pdf*, 2017.

[15] Emily L. Denton, Soumith Chintala, Arthur Szlam, and Robert Fergus. Deep generative image models using a laplacian pyramid of adversarial networks. *CoRR*, abs/1506.05751, 2015.

[16] Gentle James E. *Random Number Generation and Monte Carlo Methods*. Springer, 2003.

[17] Weinan E., Jiequn Han, and Arnulf Jentzen. Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations. *arXiv:1706.04702*, 2017.

[18] Tan Elliot and Patrick Cheridito. Forecasting foreign exchange rates in response to federal reserve communication: A machine learning approach. 2016.

[19] Cristóbal Esteban, Stephanie L. Hyland, and Gunnar Rätsch. Real-valued (medical) time series generation with recurrent conditional gans. *arXiv:1706.02633v2*, 2017.

[20] Aurelien Geron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reilly Media, 2017.

[21] Ian Goodfellow. Nips 2016 tutorial: Generative adversarial networks. In *arXiv:1701.00160v4*, 2016.

[22] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[23] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *arXiv:1406.2661v1*, 2014.

[24] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. Improved training of wasserstein gans. *arXiv:1704.00028*, 2017.

[25] Josef Hochreiter. *Untersuchungen zu dynamischen neuronalen Netzen*. PhD thesis, Technische Universitat Munchen, 1991.

[26] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Elsevier*, 1991.

[27] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[28] Kyoung jae Kim. Financial time series forecasting using support vector machines. *Neurocomputing*, 55(1):307 – 319, 2003. Support Vector Machines.

[29] Michael Kohler, Adam Krzyzak, and Nebojsa Todorovic. Pricing of high-dimensional american options by neural networks. *Mathematical Finance*, 2010.

[30] C.-C. Jay Kuo. Understanding convolutional neural networks with a mathematical model. *arXiv:1609.04112v2*, 2016.

[31] KyunghyunCho, Bart van Merrienboer, CaglarGulcehr, DzmitryBahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learningphraserepresentationsusingrnnencoder–decoder forstatisticalmachinetranslation. *arXiv:1406.1078v3*, 2014.

[32] Terry Lyons. Rough paths, signatures and the modelling of functions on streams. *arXiv:1405.4537*, 2014.

[33] Leandro S. Maciel and Rosangela Ballini. Design a neural network for time series financial forecasting: Accuracy and robustness analysis.

[34] Luke Metz, Ben Poole, David Pfau, and Jascha Sohl-Dickstein. Unrolled generative adversarial networks. *CoRR*, abs/1611.02163, 2016.

[35] Olof Mogren. C-rnn-gan: Continuous recurrent neural networks with adversarial training. *arXiv:1611.09904v1*, 2016.

[36] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. *arXiv:1211.5063v2*, 2013.

[37] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *CoRR*, abs/1511.06434, 2015.

[38] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv:1511.06434v2*, 2015.

[39] Robin Ricard. Generative adversarial networks part 2 - implementation with keras 2.0.

[40] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, Xi Chen, and Xi Chen. Improved techniques for training gans. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 2234–2242. Curran Associates, Inc., 2016.

[41] Dominik Scherer, Andreas Muller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *20th International Conference on Artificial Neural Networks (ICANN)*, 2010.

[42] Hochreiter Sepp and Schmidhuber Jürgen. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[43] Xiang Wei, Zixia Liu, Liqiang Wang, and Boqing Gong. Improving the improved training of wasserstein GANs. In *International Conference on Learning Representations*, 2018.

[44] Kaisheng Yao, Trevor Cohn, Katerina Vylomova, Kevin Duh, and Chris Dyer. Depth-gatedrecurrentneuralnetworks. *arXiv:1508.03790v2*, 2015.

[45] Lean Yu, Shouyang Wang, and Kin Keung Lai. A neural-network-based nonlinear metamodeling approach to financial time series forecasting. *Applied Soft Computing*, 9(2):563 – 574, 2009.

[46] Matthew D. Zeiler, Dilip Krishnan, Graham W. Taylor, and Rob Fergus. Deconvolutional networks. 2010.