Towards Polygonal Mesh Generation With Generative Adversarial Networks

Tom Quareme

Promoter: Prof. dr. Philippe Bekaert
Assessors: dr. Nick Michiels, dr. Jeroen Put

A thesis presented for the degree of
Master of Sciences: Computer Science,
option Visual Computing

Faculty of Sciences: Computer Science
Hasselt University
Belgium
2017-2018

Abstract

In the field of computer science, machine learning has been extensively used to learn new content based on existing examples. Due to the translational invariant characteristic of convolutional neural networks (CNNs) and derivations we see great results in fields such as computer vision, speech recognition and natural language processing. Since standard convolution in the spatial domain utilizes a regularly structured gridlike Euclidean structure such as pixels in 2D or voxels being the 3D equivalent, convolution is not suitable for employment on non-Euclidean data, unless for example for meshes a Euclidean structure such as a voxel grid is being utilized. Recent state of the art implementations of generalized convolution in the spatial domain such as Geodesic CNNs, Anisotropic CNNs or the more recent Mixture Model CNNs have proven to outperform convolution operators defined in the spectral domain, but are still not efficient enough in comparison to the GPU-friendly standard Euclidean convolution. They also lack a definition of an upsampling operation such as deconvolution necessary for generative tasks.

In this thesis, we will show that standard 2D or 3D convolutions clearly outperform novel geometric deep learning techniques of convolution and we will address the need for faster techniques which are more fine-tuned to the nature of 3D polygonal meshes.

We show that the need for a deconvolutional (i.e. upsampling) operation is necessary for the expansion of voxel-based 3D-GANs to a fully polygonal learning solution in order to generate state of the art multidimensional content of a non-Euclidean nature. We also define the criteria for such an operation. As of today, there are no known implementations which use the power of downsampling (convolutional-like) and upsampling (deconvolutional-like) operations on 3D polygonal mesh data.

We find that the simplest solution is the use of a voxel-based 3D-GAN using least squares loss, improved techniques for GAN stabilization and a marching cubes post-processing algorithm for converting voxels to polygons. GAN stabilization techniques developed after the invention of 3D-GANs are necessary, since instabilities and slow learning could still arise due to depending on the picked mini-batches. The use of higher resolution voxelizations and the increased memory consumption can be tackled by the use of sparse matrices instead of standard dense matrices, since voxel models are typically sparse. Also smaller mini-batches can be used in exchange for more execution time.

Acknowledgements

I would like to honor and thank everyone who supported me during the writing of my thesis. In particular, I want to thank my promotor Prof. dr. Philippe Bekaert and assessors dr. Nick Michiels and dr. Jeroen Put for their time spent on reading this thesis, their tips and their support and feedback whenever I had questions.

I also want to thank my family for supporting me through all these years, for their motivation, support and allowance to study for a masters degree. I also would like to thank my former girlfriend Iris for sticking several years with me in harder times and her countless motivational speeches and cheer ups whenever I was in the need for it. And last but not least, my best friends Tom and Sven for their support, necessary distractions, humor and philosophical conversations about artificial intelligence and the importance of unsupervised learning.

Dedication

I would like to dedicate this work to my dog Snoopy who passed away during the Easter holidays, 2015. Looks like ages ago, but you are still not forgotten.

Table of contents

Chapter 1: Introduction

1.1 Data-driven 3D polygonal mesh generation

Since computer games get more and more complex every year with an increasing amount of content, there is a need for automated content creation. The artistic process of creating new 3D meshes in particular for example is rather time consuming for artists and introduces a lot of additional production costs. It would be highly beneficial if game development companies, film studios or the make industry could use a system which will generate novel 3D models based on a database of existing 3D models with a certain similarity so that these objects belong to the same class. For example: given a set of cars, one has to use a system which will generate new cars. The newly generated content should be recognizable by humans to be classified as if the new objects are belonging to the same class as the given dataset, e.g. humans consider the novel objects as cars. This solution should work with every possible given classified dataset. In this thesis we want to generate novel 3D polygonal meshes based on existing given meshes of the same class. We will expand upon the existing state-of-the-art, as well as trying to improve it.

Since the increased performance of graphics cards and the rise of massively parallel General Purpose GPU programming and the availability of large datasets, even for 3D objects, it is useful to consider an artificial neural network solution, since 3D models can be described as a vector of high dimensionality. Low-dimensional feature vectors do not benefit from neural networks and are better of with more classical machine learning algorithms. An artificial neural network is an interconnected group of nodes called neurons. It resembles the large connection of neurons in a human brain. A neural network library such as TensorFlow [7] is optimized for high-performant GPU computing.

With respect to generating new content from existing similar content (text, sound, images, 3D objects,...) we see that Generative Adversarial Networks (GANs) [23] yield impressive results. GANs consist of two competing neural networks where one neural network analyses whether the given data is real or fake and another network generates novel content based on a given high-dimensional random noise vector. Depending on the input this generator will create different new data, but still with enough similarity to belong to the same class. They generate content that bears resemblance to the given training data, but also has enough creative differences to it. Being unsupervised is a huge benefit, because humans do not need to label data and the AI can learn its own features for classification purposes. However, because the two networks (a generator and a discriminator acting as a supervisor) try to outperform each other in an adversarial context, as if they are competing in a game, it can be proven in game theory that this system has trouble converging to an equilibrium where the generator will generate optimal content. Various methods combatting this instability have been developed over the past years since the introduction of GANs, with Deep Convolutional GANs (DCGANs) [65] being the most successful approach to date combined with other stabilization techniques including the definition of a least squares loss function [64] instead of the standard logarithmic one. We will address these improvements in more detail, but the introduction of convolution and deconvolution basically introduces a notion of translational invariance, i.e. it does not matter how 3D objects are positioned;

features such as curvature can still be detected implicitly. The introduction of another loss function ensures faster convergence to the equilibrium state of optimal generated content.

In the context of 3D computer graphics, 3D-GAN models [67] allow the generation of 3D voxel objects from similar examples. However, this technique is still restricted to a Euclidean domain: the 3D voxel grid, with the exception of learning on 3D to 2D parameterized surfaces such as Tutte embeddings [72] or geometry images [74]. However these parameterizations induce a loss of information which can be crucial for the reconstruction process in the generator network where one reconstructs a 3D object from a given noise vector.

While standard deep learning techniques such as convolutional neural networks [19] currently show impressive results in fields like computer visio or even speech recognition, they currently do not deliver satisfying results in a broad range of other scientific fields. The standard techniques and especially convolutional neural networks (CNNs) produce results of high accuracy in analysis tasks. More recently, they are combined with generative models in synthesis tasks. By their raster-like nature of operation, on data which is structured on n-dimensional Euclidean grids, their calculations map efficiently on GPU hardware. The past few years have seen the rise of geometric deep learning [76], which tries to generalize deep learning techniques to non-Euclidean manifolds and graphs. A lot of high-dimensional data has no Euclidean structure and non-Euclidean convolution in its standard form can therefore not be applied. One problem due to the field being so young is that current generalizations for convolution in geometric deep learning pale in performance compared to the standard techniques which are more GPU-friendly to execute, and which led to major breakthroughs in many areas because of their run-time performance.

Possible applications have roots in the manufacturing and computer games industry. We believe that in the future the employment of a novel designed layer, on top of performant existing deep learning frameworks, with the generalized convolution and deconvolution implemented as an efficient GPU-friendly algorithm and used in a generative context, will enable groundbreaking AI applications in a broad range of areas, including 3D data in computer vision, but also graphs in network analysis, social networks in social sciences, 3D medical imaging, molecule design in chemistry, genetics, fraud detection, and so on.

1.2 Research goals

In this thesis we will discuss how current state-of-the-art 3D Generative Adversarial Networks (3D-GANs) can be modified to generate desirable 3D polygonal objects from an existing set of example objects. We will also address shortcomings and possible future improvements and the need for a full generalization of convolution and deconvolution on non-Euclidean data such as 3D polygonal objects.

An underlying question is whether or not current state-of-the-art generalized convolution in non-Euclidean domains is speed-wise performant enough. In an experiment we will show that standard 2D or 3D convolutions clearly outperform novel geometric deep learning techniques of convolution and

we will address the need for faster techniques which are more fine-tuned to the nature of 3D polygonal meshes.

We show that the need for a deconvolutional (i.e. upsampling) operation is necessary for the expansion of voxel-based 3D-GANs to a fully polygonal learning solution in order to generate state of the art multidimensional content of a non-Euclidean nature. We also define the criteria for such an operation. As of today, there are no known implementations which use the power of downsampling (convolutional-like) and upsampling (deconvolutional-like) operations on 3D polygonal mesh data. We show that the simplest solution is the use of a 3D-GAN using least squares loss, improved techniques for GAN stabilization and a marching cubes post-processing algorithm for converting voxels to polygons. The use of higher resolution voxelizations and the increased memory consumption can be tackled by the use of sparse matrices instead of standard dense matrices, since voxel models are typically sparse. Also smaller mini-batches can be used in spite of more processing time.

1.3 Overview

While this chapter serves as an introduction with an additional Dutch summary, in the second chapter of this dissertation we explain the importance of artificial neural networks for our problem, how they work mathematically and how a basic neural network can be built. We also motivate their relevance for our problem. In the third chapter we introduce the concept of convolutional layers and pooling, which can be used for translational invariant feature detection; an idea we will use later on as well. In this chapter we also explain which hyperparameters are common in convolutional and pooling layers, provide a list of common architectures and present some time and space optimizations as well. The fourth chapter deals with possible performance improvements for neural networks in general. A large section in this chapter compares different optimization algorithms and gives advice about the choices which can be made. Another large section of this chapter gives five alternatives for hyperparameter tuning.

In chapter five we introduce Generative Adversarial Networks (GANs) which can be used to generate novel data. We also address possible problems we can encounter when training a GAN. To overcome these difficulties we also introduce improved techniques for training GANs, with Wasserstein GAN (WGAN) and Least Squares GAN (LSGAN) architectures being the most memorable. In the sixth chapter we include convolution and deconvolution into the GAN architecture, forming a Deep Convolutional GAN (DCGAN). Finally, in the seventh chapter we expand the DCGAN from 2D to 3D to form a 3D-GAN which can be used on voxel data. In chapter eight we introduce a post-processing technique we can use to convert voxel data to polygonal meshes called: the marching cubes algorithm starting from its 2D analogon. In chapter nine we discuss how to use convolution in non-Euclidean domains such as the mesh of the shape itself and or graphs.

In the next chapter we discuss the implementation details starting from a vanilla artificial neural network, working our way up from the theory to an implementation of a 3D-GAN with marching cubes as post-processing. We also discuss an alternative implementation for convolution in the non-Euclidean

domain. The use of different hyperparameters and possible improvements will be discussed as well based on the corresponding theory chapters.

In the sections of the next chapter we define our experiments based on the implementation details discussed in the previous chapter. We also discuss the results by comparing created samples, measuring performance in the time or accuracy domain and address shortcomings and possible future improvements. The thesis closes with a conclusion of the results within our predefined goals.

1.4 Dutch summary

Tegenwoordig worden computerspelletjes en animatiefilms steeds complexer ieder jaar. Er is steeds meer artistieke data nodig, waardoor er een hoge nood is aan automatisering voor de creatie van voorwerpen en dergelijke. Zo is de ontwikkeling van 3D objecten erg tijdrovend, waardoor de productiekosten stijgen. Het zou daarom voordelig zijn wanneer de industrie gebruik zou kunnen maken van een systeem dat nieuwe 3D modellen genereert gebaseerd op een database van bestaande 3D modellen. Deze modellen dienen een zekere gelijkaardigheid te hebben; bijvoorbeeld gegeven auto's zorgen voor het genereren van nieuwe auto's. Het is hierbij gewenst dat mensen geen onderscheid kunnen maken uit wat nu wel of niet door een computer gegenereerd is. In mijn thesis wil ik nieuwe 3D polygonale objecten genereren gebaseerd op een reeks bestaande objecten van dezelfde klasse. Ik start vanuit de huidige standaard en tracht deze te verbeteren.

Tegenwoordig slagen Generative Adversarial Networks (GAN's) erin om geloofwaardige teksten, geluid, afbeeldingen en recent ook 3D objecten te genereren. Een GAN bestaat uit twee competitieve neurale netwerken. Een neuraal netwerk is een verbonden groep van neuronen die de werking van menselijke hersenen op een kunstmatige manier nabootst. Eén neuraal netwerk staat in voor het analyseren van de gegeven data, waarbij het controleert of deze gemaakt is door mensen of door een computer gegenereerd is. Het andere neurale netwerk tracht het analyserende neurale netwerk te misleiden door zichzelf te verbeteren wanneer hij een berisping krijgt dat zijn creatie nep is. Variatie in creatie wordt aangebracht door willekeurige ruis aan de generator te geven.

Beide neurale netwerken trachten op basis van feedback naar elkaar toe zichzelf te verbeteren alsof ze een spel tegen elkaar spelen. Men kan bewijzen dat dit systeem problemen heeft om een evenwicht te bereiken waarin de generator optimale nieuwe objecten genereert. In mijn thesis onderzoek ik verschillende oplossingen om deze instabiliteit aan te vechten. Ik stel verder ook, naargelang mijn resultaten en discussie, de beste manier voor om 3D polygonale objecten te genereren.

3D-GAN's die inwerken op 3D modellen geconstrueerd uit blokjes (voxels) kunnen aangepast worden zodat ze 3D polygonale objecten kunnen maken uit een verzameling bestaande objecten. Verder is er ook een nood aan een volledige veralgemening van het omhoog en omlaag samplen van niet-Euclidische data zoals 3D polygonale objecten. Wanneer men omhoog samplet, dan tracht ben van een kleine verzameling startwaardes naar een volledig vergroot eindresultaat te gaan. Men start als het ware van een verkorte encodering. Als men omlaag samplet, dan doet men het omgekeerde: men tracht van een

volledig object naar een verkorte semantische codering te gaan die het object op een verkorte manier beschrijft. In het pure polygonale geval bestaat er geen definitie voor het omhoog samplen. Tot er geen oplossing is, is het dus aangewezen om te werken met voxels om deze nadien dan naar polygonen om te zetten. Een toekomstige oplossing dient deze operaties te implementeren op een manier zodat deze efficiënt door een grafische kaart uitgevoerd kunnen worden. Het is ook belangrijk dat zo'n oplossing nog steeds accuraat de semantiek van de objecten omvat. Uit de benchmarks blijkt dat grafische kaarten nog steeds beter presteren wanneer ze dichte data toegewezen krijgen om te verwerken. Vandaar dat de oplossingen binnen niet-Euclidische deep learning aanzienlijk trager verlopen dan hun Euclidische tegenhangers. Er is dus een nood aan snellere technieken die meer aangepast zijn aan de natuur van 3D polygonale objecten.

Het beste alternatief is om voorlopig te blijven werken met voxelmodellen van hoge resolutie. Een voxelrooster is meestal slechts voor een klein gedeelte gevuld met blokjes. Op deze manier wordt er veel videogeheugen verspild. Dit kan men tegenwerken door gebruik te maken van "dunne" matrices die zo'n gevuld rooster voorstellen. De berekeningen voor dunne matrices gaan efficiënter met geheugen om door alle lege cellen van het rooster niet te beschouwen. De voorstelling in het geheugen is verkort genoteerd. Ook kan men werken met kleinere verzamelingen van objecten tijdens het leerproces. Hierbij duurt het trainen wel langer, maar gaat er minder geheugen verloren. Snellere training kan op zijn beurt weer bekomen worden door gebruik te maken van alternatieve formuleringen voor het achterliggend optimalisatieprobleem die ik in mijn thesis aan bod breng om tot een evenwicht te komen. Als naverwerking wordt er nog gebruik gemaakt van een algoritme dat de voxels omzet naar polygonen. Verder viel het mij op dat in deep learning softwarebibliotheken soms implementatiedetails afgeschermd zijn en voor problemen zorgen wanneer je deze niet zou verwachten. In sommige gevallen is het dan nodig om zelf zoveel mogelijk berekeningen te doen zodat je exact weet wat er gebeurt.

Een toekomstige onderzoeksrichting is de ontwikkeling van benaderende algoritmes die inwerken op datastructuren die in harmonie zijn met de werking van een grafische kaart. Dit bedraagt grotendeels continue data. Er dient een afweging gemaakt te worden tussen accuraatheid en uitvoersnelheid. Als resultaat kan er heel wat rekentijd en geld bespaard worden in wetenschappelijke domeinen en industrie die ver reiken buiten de computerwetenschappen zoals bijvoorbeeld het synthetiseren van chemische stoffen binnen de chemie, productontwikkeling, kunst en zoveel meer.

Chapter 2: Artificial neural networks

2.1 Motivation

An artificial neural network (ANN) [1, 13], also called a multilayer perceptron, is a computational directed acyclic graph system based on the human brain. By providing training data to the system, it progressively learns to improve itself on a certain task. As a simple example one could train a neural network in order to identify certain objects in an image, i.e. image recognition. For a supervised learning system one could provide training images described by a high-dimensional feature vector $\vec{x}^{(i)}$ and an associated training label $y^{(i)}$ in order to form a training element $(\vec{x}^{(i)}, y^{(i)})$. In a binary classification system this could be an image and a label which tells whether the image is a dog or not for example. These learned features do not necessarily compare to an understanding humans have of an image of a dog. Furriness, having a black nose or a tail might be deeply encoded in another mathematically cryptic way. Other data could be more general signals such as sound, animations, video, meshes, graphs, text, etc.

The neural network consists of a set of connected nodes called neurons connected to other neurons by synapses to transmit signals to each other. Neurons are ordered in an input layer, different hidden layers and an output layer. The signals are expressed by real numbers called weights. When this signal "arrives" at the destined neuron, this information will be processed at the neuron by a non-linear activation function. As an input to this function a dot product is calculated between a weight vector, consisting of all the incoming weight connections, and an input vector of data. An additional real number called a bias can be added to this dot product. In order to use the model to transform input data to a certain desirable output, these weights must be learned. Initially the weights are randomly assigned. When providing training data to the system, the error between the resulting output label and the desired label is computed and weights are adjusted accordingly by a learning process.

When working with high dimensional data such as images, 3D meshes, neural networks and its subclasses are preferred over other machine learning models such as support vector machines (SVMs) [2], decision trees [3], naive Bayes [4], etc. When data is low dimensional neural networks would be an overkill, since they would most likely overfit the problem. If the neural network has more learnable parameters than inherent dimensionality of the data, then the neural network would not be able to generalize beyond the training data. On the other hand when data is high dimensional, it becomes much harder for a model to detect useful patterns. Plenty of machine learning models depend on a certain metric to measure distances between data points. When the dimensionality increases, the distance or metric between two data points becomes larger. Since 3D meshes translate to high dimensional feature vectors and neural networks appear to be unaffectable by the curse of dimensionality [5], they are a nice fit to the problem of learning from meshes. Furthermore the Manifold Hypothesis suggests that real-world high dimensional data lies on a low dimensional manifold embedded in a higher dimensional space. Thus high dimensional data has a lower dimensional underlying pattern which can be learned by a neural network. This explains the reason why neural networks are able to learn from their training

data.

For the remainder of this thesis we restrict ourselves to deep feedforward networks [1, 13] where information flows from the input in the direction of the output layer. No feedback is allowed.

2.2 Network topology

A standard deep feedforward neural network [1, 13] consists of an input layer which forms an n-dimensional feature vector $\vec{x}^{(i)}$ where each vector component corresponds to a node in the input layer. Visually one can view the input layer as a stack of nodes vertically ordered in the order of the basis vectors of the data. For an image these basis vectors correspond to the pixel positions in a Cartesian grid. The value of each component corresponds for example to a color or grayscale value in [0, 1]. For other types of input data the same reasoning applies, but basis vectors might have different properties or layouts. The input layer is connected by a sequence of hidden layers consisting of neurons and an output layer consisting of class scores in [0, 1]; i.e. the probability given input data corresponds to each class. The label of the class with the highest probability will be chosen to be the output of the system.

The value of a neuron corresponds to an activation which is a real number in [0, 1]. The activations of each neuron in the hidden layers correspond to a certain pattern the network has learned during training. When the dataflow goes from the last hidden layer to the output layer, we can view the result as which linear combination of subcomponents (patterns) of the previous layer corresponds to which class. These subcomponents or patterns in turn consist of subcomponents of the previous layer, etc.

Imagine one provides the neural network with certain input data. Then the neurons of each layer get activated when they recognize patterns which match certain neurons. The deeper the hidden layer in the network, i.e. the higher the sequential number of the layer, the more complex the learned pattern becomes.

Each connection in a neural network receives a certain weight. This is a real number which determines the strength of the connection. The higher the number, the stronger the connection and the more influence on the result it will have. For a certain neuron in the first hidden layer certain weights match the learned underlying pattern. When doing image recognition, one could view this learned pattern as lighting up pixel values of the pattern by increased weight values.

The activation itself is computed by a dot product which is calculated between a weight vector, consisting of all the incoming weight connections, and an input vector of data or a previous activation. An additional real number called a bias can be added to this dot product. A bias is a tendency for a neuron to be active (positive) or inactive (negative). Mathematically we express the activation by following equation:

$$a^{(j)} = \sum_i [w_i^{(j-1)} a_i^{(j-1)} + b_i^{(j-1)}].$$

Essentially this new activation could be a any number and should be normalized to fit in [0, 1]. To do this we employ an activation function. For the activation function we tend to use nonlinear functions as most real world data is nonlinear and the neurons in a neural network should learn these nonlinearities.

When performing classification, a softmax function [6] is generally employed as an activation function for the output layer of the neural network. This is a normalization which makes sure that the output components are probabilities of sum 1. This function has as an input a K-dimensional vector $\vec{z}$ of arbitrary real-valued scores, called logits, and the function "squashes" these scores to a vector $\sigma(\vec{z})$ of values in]0, 1] that sum to one. Its function is given by:

$$\sigma : \mathbb{R}^K, \sigma_i > 0, \sum_{i=1}^{K} \sigma_i = 1$$

$$\sigma(\vec{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, j = \{1, ..., K\}, j \in \mathbb{N}_0.$$

Logits or scores form a vector of raw (non-normalized) predictions created by the classification model. Performing the softmax on a certain dimension called the axis, in a framework such as TensorFlow [7] this can be expressed as:

```
softmax = tf.exp(logits) / tf.reduce_sum(tf.exp(logits), axis)
```

In the case when one wants the outputs to be ranging from 0 to 1, but the outputs do not need not sum to 1, one can use the sigmoid function [8] as the activation function in the output layer. The sigmoid function can be expressed as:

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

Addition of a small epsilon value might help to avoid taking the logarithm of zero in the cross-entropy computation [9]. In order to define an optimization criterium, one can compute and minimize the cross entropy between the activated output vector *S* and the one-hot encoded [10] desired output vector *L* which serve as discrete probability distributions:

$$H(S, L) = -\sum_{i=1}^{n} L_i \log(S_i).$$

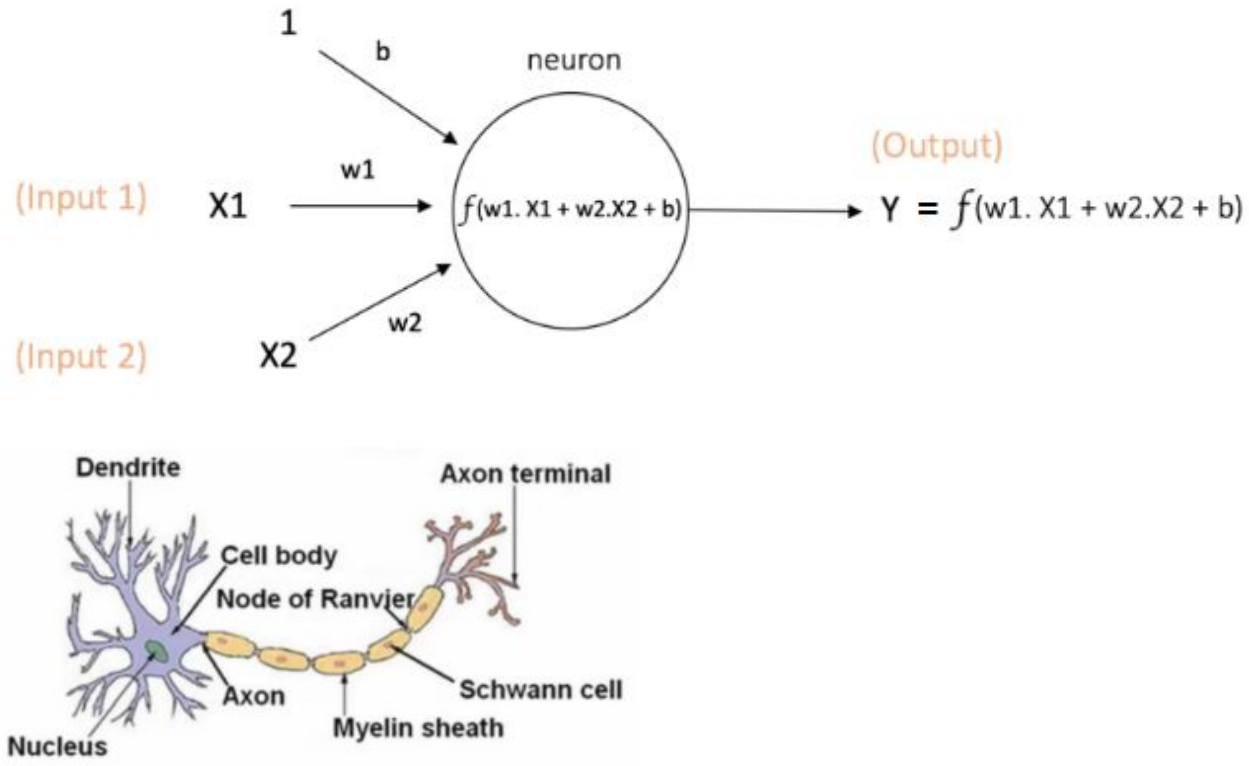

Figure 2.1: One neuron in isolation [1, 13]. The neuron accepts input numbers X1 and X2. Each input has an associate weight, w1 and w2 respectively. We also incorporate an additional input of value 1 with a weight b, which we call the bias. We can compute the output Y of the neuron by using a nonlinear activation function *f*. This is analogue to neurons in biology. Dendrites correspond to the input, neuron cells correspond to the nodes in our artificial neural network and the axon corresponds to the output of the network.

When combining multiple nodes, we can form a neural network with for example no hidden layers. This is called a single-layer perceptron [11]. Multi-layer perceptrons [11] contain one or more hidden layers. Because a single-layer perceptron can only learn linear functions, we will instead only consider multi-layer perceptrons which can learn nonlinear functions in general. Note that in a feedforward neural network [11] the information only flows from the input to the output in one direction as opposed to Recurrent Neural Networks (RNNs) [12] which can form cycles. An example of a feedforward neural network with one hidden layer, and thus a multilayer perceptron (MLP), is given below.

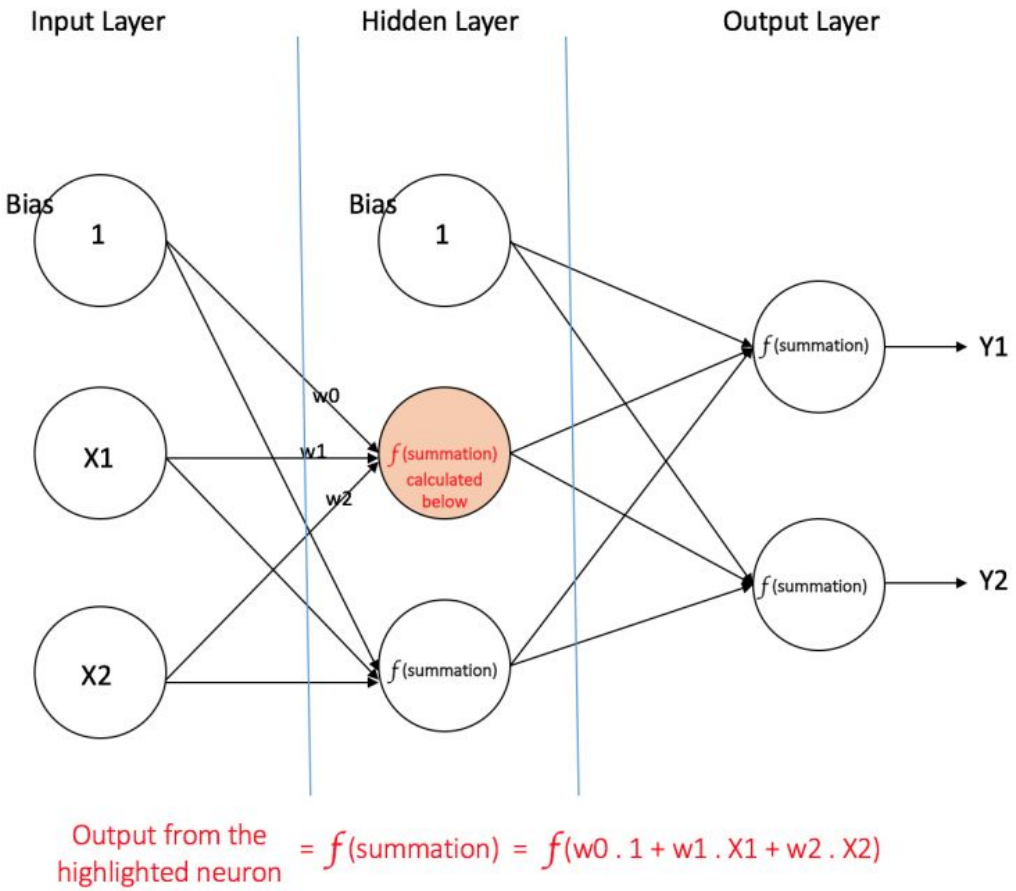

Figure 2.2: Example of a neural network. There are two input nodes and one additional bias node of value 1 in the input layer. The input nodes can read an input X1 and X2. These nodes are fed to the connected nodes of the next layer, which is called a hidden layer. No computations happen in the input layer. In the hidden layer of this example there are also three nodes of which one is a bias. The formula for the computation of the output from the highlighted neuron is also given. Here *f* is an arbitrary activation function. The final output layer consists of two nodes which compute the outputs Y1 and Y2.

2.2.1 Sigmoid activation function

Historically the first function used for this purpose was the sigmoid function (also called the logistic function) [8, 13] defined as:

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

Here *x* corresponds to a weighted summation and the function itself corresponds to a measure of how positive the weighted summation is. Large negative numbers map to 0, while large positive numbers map to 1.

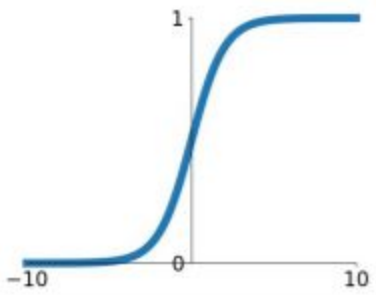

Figure 2.3: The S-shaped sigmoid function described above.

Throughout years of experience in the field, new activation functions which yield better results have been discovered and nowadays we can state that sigmoid is outdated. There are two reasons for this:

1) The use of sigmoid activation functions zeroes out the gradient magnitudes. A neuron with a sigmoid activation function gets saturated and the result will be either 0 or 1 and nothing in between. In these regions the gradient will turn out to reach a magnitude of 0. Consequently, after some iterations the weights are updated extremely slowly because of the low gradient magnitude. In order to prevent saturation and the effect of learning, one needs to carefully initialize the weights. Weights can not be initialized too high to prevent saturation. When weights do get initialized too high, no actual learning occurs.

2) As a second problem, but less severe than the one above, the output of a sigmoid function is has a center different from zero. This is unpleasant since neurons in succeeding layers require zero-centered data. As can be seen on the above figure, we get output which will always be positive. When input data for a neuron is always positive, the gradient operating on the weights during backpropagation will always be strictly positive or negative. This can cause unnecessary zig-zagging paths in the gradient descent optimization.

2.2.2 Tanh activation function

The tanh (hyperbolic tangent) function [13] fits any real number in the interval [-1, 1] in a non-linear fashion. Just as before with the sigmoid activation, saturation and vanishing of the gradient can occur. But as a plus the second problem does not occur; the output will be zero-centered and thus this method will be preferred over the sigmoid activation function. The tanh function is defined as:

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1.$$

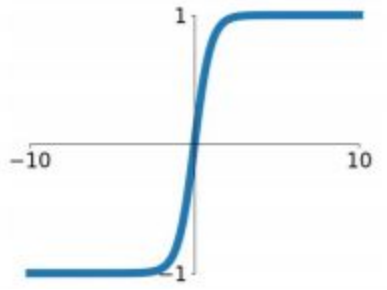

Figure 2.4: The tanh activation function described above.

2.2.3 ReLU activation function

Currently the most preferred activation function is the Rectified Linear Unit (ReLU) [13]. It is defined as:

$$f(x) = max(0, x).$$

All this function does is clamp all the negative values to zero and let the positive values be.

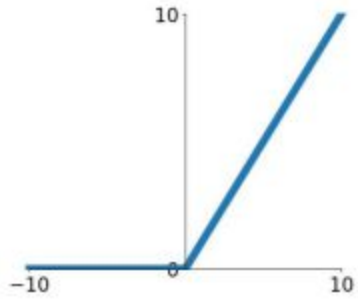

Figure 2.5: The ReLU activation function described above.

The benefits of this activation function are the following:

- Compared to the sigmoid and tanh activation functions Krizhevsky et al. [14] found that ReLU accelerates the convergence of stochastic gradient descent. Explanations can be found in the unsaturated nature of the function and its piecewise linearity. Faster convergence means faster learning.
- ReLU is computationally less intensive to perform than the exponent functions used in sigmoid or tanh.

Unfortunately it is possible for neurons with a ReLU activation function to end up in a dead state, i.e. it will never activate on data elements for the entire training set due to an undesirable weight update. From this point the gradient will be stuck at zero. We can overcome this problem significantly by setting

the initial learning rate not too high. Another alternative is the use of Leaky ReLU or Maxout instead of ReLU.

2.2.4 Leaky ReLU activation function

Leaky ReLU, introduced by Maas et al. in 2013 [15], tries to minimize the dead state of neurons due to the ReLU activation function. Normally negative values will get clamped to zero, but now we will have a tiny negative inclination. Leaky ReLU is defined as:

$$f(x) = max(\alpha x, x).$$

Here α is a constant around the order of 0.01 or smaller. A downside of this function is that it does not always yield consistent results, so it serves as a last resort fix for ReLU's possible dead neuron state [13].

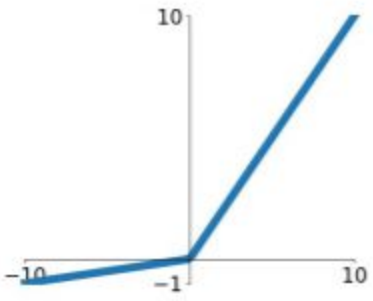

Figure 2.6: The Leaky ReLU activation function described above.

2.2.5 Maxout activation function

Goodfellow et al. [16] proposed a new sort of activation function which does not take a dot product between weights and data elements but insteads serves as a generalization of ReLU and Leaky ReLU. It is defined as:

$$f(x) = max(w_1^T x + b_1, w_2^T x + b_2).$$

For instance when w_1 and b_1 are zero we get plain old ReLU. Thus Maxout [16] has the same benefits as ReLU and instead does not suffer from dying neurons. Yet, its major downside is the doubling of parameters for each neuron, which makes the network more prone to overfitting.

When in doubt, we use the ReLU activation function. Next, we observe the amount of dead units in the network. When this is a significant problem, we might want to try Leaky ReLU or Maxout. Sigmoid is outdated and should never be used [13], unless necessary at the end of a neural network. One could try to use tanh, but it is expected to yield results worse than the ReLU variants or Maxout [13]. It is not so

common to mix different types of activation functions in one neural network, but it can be done whenever necessary.

2.2.6 Implementation of activation functions

When computing the activation of a neuron in a certain layer, the designer of a neural network library can implement this as a vector-matrix operation in order to calculate the activation for a whole hidden layer [13]. Let *f* be an activation function, *W* a matrix of weights. The rows of the matrix *W* correspond to the neurons of the previous layer, while the columns correspond to the neurons of the current layer. The activations of the previous layer are given by a vector $\vec{a}^{(j-1)}$ and the corresponding bias is given by a vector $\vec{b}^{(j-1)}$. The resulting activation vector for the current layer is given by: $\vec{a}^{(j)}$. This results in the following equation:

$$\vec{a}^{(j)} = f(W\vec{a}^{(j-1)} + \vec{b}^{(j-1)}).$$

A neural network can be considered a function approximator and the activation functions provide nonlinearity.

2.2.7 Implementation of forward propagation

Before starting the forward propagation step [1], we initialize all the weights and biases in the network. The weights get randomly assigned, while the biases are initialized to zero. Consider a node *V* in the hidden layer. The input connections to this node have weights w_1, w_2 and w_3. Here w_1 is considered a bias. A first training example gets loaded into the input nodes. This training example also has a given class label, which serves as the desired output of the neural network for this particular training example. The output of a node *V* in the hidden layer can be computed as:

$$V = f(1.w_1 + x_1w_2 + x_2w_3).$$

Outputs from each node in the hidden layer serve as inputs to the next layer, which in case of one hidden layer will be the output layer. From here it is possible to compute the output probabilities for each possible class. We then compute the error between the desired probabilities and the actual computed probabilities.

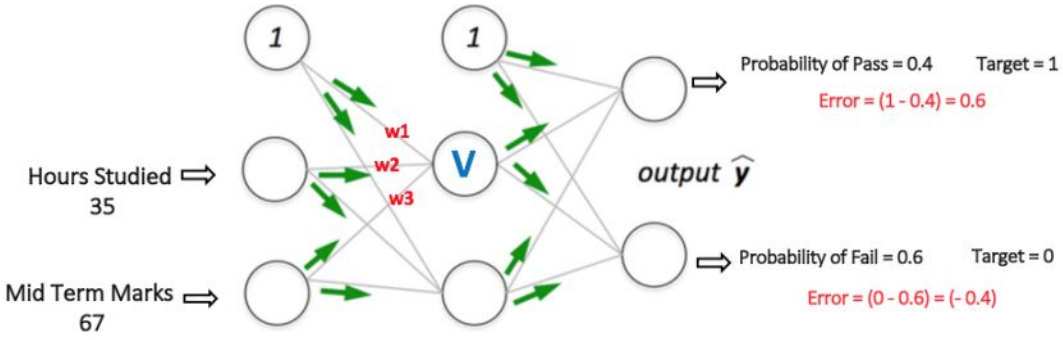

Figure 2.7: Forward propagation step [1] in a multi-layer perceptron which will deliver erroneous output.

2.2.8 Implementation of backpropagation

Learning by deep neural networks is the process of learning its weights and biases. This can be done by backpropagation [17]. As an input we provide training vectors $\vec{x}$ with some assigned label y. Together they form a training example $(\vec{x}, y)$. Weights get adjusted until the resulting labels conform to the desired given labels. One computes the (e.g. SSD, RMS,...) error between the desired output as a one-hot vector and resulting output vector. This corresponds to a certain cost, which is a scalar number. Recall that a one-hot vector is a vector where one component equals 1 and the others are 0.

Now we can construct a cost function which uses the weights and biases as an input together with a high amount of training examples. As an output we receive the cost. Training the neural network essentially boils down to an optimization problem where we need to minimize the cost function considering every sample of training data. In order to minimize the cost function we employ an algorithm called gradient descent [18] to search a local minimum in the cost space. Imagine that all the weights and biases are put together in a high dimensional vector $\vec{\theta}$. Weights are randomly initialized and biases are initialized to zero. Then the cost function is defined as a function $J(\vec{\theta})$, while the local minimum can be expressed as $min\{J(\vec{\theta})\}$.

When minimizing the cost function we use gradient descent to find the direction of steepest descent; comparable to a snowboarder descending a hill and trying to gain maximum speed. Computing the direction of steepest ascent in a certain point $\vec{w}$ can be done by using the gradient of the cost function in this point. The steepest descent can be computed by considering the negative direction. So in summary, gradient descent computes $\nabla J(\vec{\theta})$, takes a small step in the direction of $-\nabla J(\vec{\theta})$, where the magnitude of this vector is determined by a learning rate [18]. This process repeats itself until a local minimum is approached.

We do this gradient descent step per epoch (i.e. one pass over the training set), for every training example of the data set by applying the necessary gradient change $-\nabla J(\vec{\theta})$ to the weights. The actual

step size will be determined by a learning rate α. Consider θ_{old} to be the old position on the cost function before taking a step and θ_{new} to be the new position after the step. This step can be expressed as:

$$\theta_{new} = \theta_{old} - \alpha \nabla J(\vec{\theta}_{old}).$$

The necessary gradient change $-J(\vec{\theta})$ is a vector with a dimensionality corresponding to the total number of weights and biases. Each component in this vector corresponds to the average change the weights and biases require to produce minimal error. The average is taken over all the training examples or examples in the next mini-batch of examples to avoid computational overhead.

When using backpropagation, we need to adjust the weights and biases from the last layer to the first. Backpropagation in supervised learning is comparable to learning from mistakes and a supervisor will correct the network whenever it makes mistakes. We can observe that we need to adjust the weights connected to neurons in the previous layer which have the largest activation in particular. Adjustments to these weights cause stronger changes to the training result. Computed errors, which correspond to the gradient, for each output node get propagated back through the network by backpropagation. In order to receive the desired output or a valid approximation, we need to adjust the weights and biases accordingly by causing smaller errors. This algorithm then goes back recursively to the first layer.

Mathematically, calculating the derivatives of the cost function is expressed as follows: let $\delta_j^{(l)}$ and $a_j^{(l)}$ respectively be the error and activation of node *j* in layer *l*, while y_j is the desired probability of the *j*th class label. Then we have:

$$\delta_j^{(l)} = a_j^{(l)} - y_j.$$

For instance:

$$\delta_1^{(3)} = a_1^{(3)} - y_1$$
$$\delta_2^{(3)} = a_2^{(3)} - y_2.$$

We then for example backpropagate the errors $\delta_1^{(3)}$ and $\delta_2^{(3)}$ all the way towards $\delta_1^{(1)}$, $\delta_2^{(1)}$ and $\delta_3^{(1)}$. We then have the following vector equations where we employ element-wise multiplication:

$$\delta^{(2)} = \left(\Theta^{(2)}\right)^T \delta^{(3)} .* g'(z^{(2)})$$
$$\delta^{(1)} = \left(\Theta^{(1)}\right)^T \delta^{(2)} .* g'(z^{(1)})_{\cdot}.$$

Here *g* represents the activation function being applied to raw activations *z*.

In general, this corresponds to the following formula for the partial derivatives of the cost function:

$$\frac{\partial}{\partial \Theta_{ji}^{(l)}} J(\vec{\theta}) = a_j^{(l)} \delta_i^{(l+1)}.$$

Here $\Theta^{(l)}$ corresponds to the matrix of weights and biases from layer *l* to layer *l* + 1. Columns and rows in this matrix correspond to how the connections are formed. The column index is the index of the current node, while the row index corresponds to the index of a connected node from the previous layer.

In the end after all the epochs, the neural network has learned its weights and biases. After training, the network can be applied to a test set. Testing only will involve forward propagation.

Chapter 3: Convolutional neural networks

3.1 Motivation

Convolutional Neural Networks [19, 20], also abbreviated to CNNs or ConvNets are a variation of a standard neural network. Just as before they consist of a graph of neurons with learnable weights and biases. An activation function is present in each neuron. This function takes the input, applies a dot product between the input vector and a weight vector and finally applies a nonlinear function such as ReLU or tanh. All other principles and techniques for vanilla neural networks are still relevant: e.g. the presence of a loss function on the last layer, classification derived from scores, hyperparameter optimization [21, 22] and other improvements on neural networks which will be discussed later. The only alteration is the actual architecture, the meaning of the weights and the nature of the input data which usually consists of n-dimensional images, i.e. pixel- or voxel-like data.

A problem with standard neural networks is that they do not scale appropriately to larger sizes of input data. Consider for instance a fully-connected neuron in the first hidden layer. Imagine an input image of size *W*x*H* which consists of 3 color channels (R, G and B). The considered neuron would then have *W*x*H*x3 weights. For large *W* and *H* we observe quadratic scaling of the amount of weights for each neuron in the network. Since there would be a large amount of parameters, we could say that the full-connectivity of the standard neural network causes overfitting. Therefore a solution which greatly reduces the amount of weights and supplementary overfitting is desired. Another problem is that translation, rotation, scaling and other transformations are poorly handled by standard neural networks.

In a CNN all the weights that are being learned are the filter elements in the convolutional layers. In the first layer these filters detect primitive features such as edges, points, dark or bright planes. In the next layer more recognizable features are learned such as: eyes, noses, mouths,... for facial recognition tasks as an example [20]. In later layers features get more complex and could be entire faces or more intricate features. Other weights are the ones in the fully-connected layers which correspond to weights being learned in standard neural networks. For CNN learning we also employ backpropagation and a chosen variant of gradient descent for finding local minima.

Note that CNNs are only useful when data has an n-dimensional image-based form, so local spatial patterns can be recognized in the data. In other words: use of CNNs does not make sense if data continues to be useful after swapping rows or columns. This is important when we will construct our Generative Adversarial Network [23].

Just as standard neural networks before, we put an emphasis on the classification problem and discriminative models. In future chapters we will shift the context to generative models and thus the task of generating new content.

3.2 High-level architecture

A CNN [19, 20] assumes the input data is an n-dimensional variation of image data. In the 2D case this can be pixel data, in 3D voxel data and in 4D spatiotemporal voxel data (animations of 3D voxel models). For sake of simplicity let us first assume 2D pixel data. For a CNN this will constrain the architecture of the neural network. The layers of a CNN arrange the neurons in 3 dimensions: width, height and depth. For 2D pixel images the input layer will have size *W*x*H*x*D* where *W* is the width, *H* the height and *D* the number of color channels of the image. Instead of fully-connecting every neuron, connections of neurons in a layer will be more sparse. The output layer of a CNN will be a vector of size 1x1xC, where depth C is the number of classes. Each value will be the score for every class.

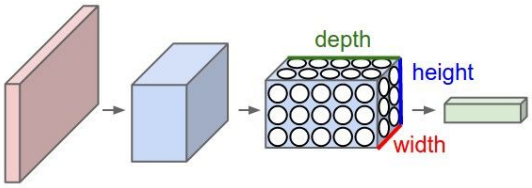

Figure 3.1: In a Convolutional Neural Network for 2D data we organize neurons in 3D blocks, which pose as a layer, with a certain width, height and depth. This idea can be extended to data of multiple dimensions. Here the input layer has a depth of 3 (R, G, B pixel values) and a width and height according to the dimensions of the input images. Each following layer essentially consumes an input hypervolume and spits out another output hypervolume of neuron activations, having the same dimensionality as the block which performs the transformation.

3.3 CNN layers

Comparable to standard neural networks a CNN consists of a sequence of layers. Each layer transforms a 3-dimensional input block of data/neurons to another 3-dimensional output block. This transformation process is performed by a differentiable function. The property of differentiability stems from the need of employing gradient descent. The four types of layers which will be alternated are:

- Convolutional layer (CONV)
- Rectified linear unit layer (RELU)
- Pooling layer (POOL)
- Fully-connected layer (FC)

We will discuss these layer types in depth in the following subsections. For now we will treat every layer type as a black box. A simple architecture which makes use of the aforementioned layers can be:

- *INPUT:* consists of the pixel data of images with dimensions *W*x*H*x*D*. Here *W* is the width, *H* the height and *D* the number of color channels (red, green or blue).

- *CONV:* consists of *F* number of filters. This layer uses the filter stack to compute a certain output of size of *W*x*H*x*F*. Here we place the convolution filter over local parts of the input block. Then one computes the dot product between the weights of the filter and the underlying input values. This is done for every filter and the results get stacked along the depth.
- *RELU:* applies a nonlinear activation function to each neuron in the layer.
- *POOL:* applies downsampling for each spatial dimension (in this case: width and height). For example max pooling with a 2x2 pooling window, a movement size (stride) of 2 pixels in each dimension will calculate the maximum value of each 2x2 matrix which appears under the 2x2 pooling window. The result is an output volume of size (*W*/2)x(*H*/2)x*F*.
- *FC:* calculates the class scores. For a classification task of *C* classes the final output result will have size 1x1x*C*. In this layer each neuron is fully-connected to every neuron in the previous layer.

We can also mention that only the convolutional and fully-connected layers have parameters and CONV, FC, POOL can have hyperparameters which the developer of the network can optimize. Unlike weights, these values are not learned by the network. In practice, more complex architectures can be built by alternating CONV, RELU and POOL layers followed by a final FC layer.

3.3.1 Convolutional layer (CONV)

A convolutional layer is basically a set of filters. Each weight of every filter can be considered a learnable parameter. Filters typically are much smaller than data or neuron volumes. The depth, for example for color channels, stays the same as in other volumes, yet width and height are very small in comparison. The most common filter used in a convolutional layer has a size of *F* = 5x5x3, yet other width and height values such as 3 or even 1 are possible. Since images can be considered as feature vectors of a high dimensionality and the same will hold for the amount of neurons in a large network, one only needs to connect neurons of a convolutional layer to local parts of its previous input layer. The filter size *F* will determine this local connectivity. This locality of connections takes place along every dimension except for the depth dimension. Depth is fully considered as connected.

When the neural network performs a forward pass, each filter moves across the width and height of its input volume. To complete actual convolution, we then compute the dot product between the learnable filter weights and the part of the input elements laying under the filter. For every filter in each convolutional layer and after each convolution we receive a 2D activation map. Activation maps get stacked in the depth to form a new output volume.

The goal is to learn filter weights which together form filters that will recognize features in the input data like edges or other low level features or even more high level features such as eyes, noses or even entire faces in deeper layers of the convolutional neural network.

As a rule of thumb it is advised to stack convolutional layers with smaller filters such as 3x3 or 5x5 instead of using 1 convolutional layer with large filters. This ensures learning of more meaningful

features with the benefit of using less parameters to reduce overfitting. However when memory is scarce, a settlement has to be made since more convolutional layers introduce memorization of intermediate results when performing the backward pass.

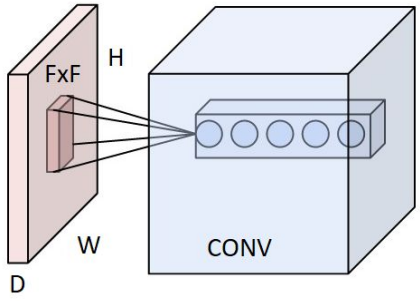

Figure 3.2: Imagine when an input volume, which in general can be a previous layer other than INPUT, has size *W*x*H*x*D* and a filter of size *FxF* is utilized in the convolutional layer. Each neuron in this layer has *F.F.D* local connections (weights) to the previous volume and one extra bias parameter if necessary.

A convolutional layer has four types of hyperparameters which determine the size of the output volume:

- *Depth K:* the depth of a convolutional layer determines the number of filters.
- *Filter size F:* the size of *F*x*F* filter kernels used in the convolutional layer.
- *Stride S:* is the movement size in pixels used as a step size when the filter is moved across the image. The larger the stride, the smaller the output volume along the spatial axes. In practice it is advised to use smaller strides for better results. When *S* = 1, downsampling will be handled only by the pooling layers.
- *Zero-padding size P:* is used when we want to control the spatial measurements of the output volumes when padding the input volume with zeros circling the boundaries. Padding also induces higher performance (accuracy), since boundary information of the input data will not be lost.

Typical values for these hyperparameters are: *F* = 3, *S* = 1 and *P* = *1*.

Parameter sharing [20] is used in a convolutional layer to bring down the amount of parameters to manageable proportions. It allows the sharing of multiple weights by every neuron in a certain feature map. As an example, one could take a volume of neurons and take depth slices of 1 single depth value each, essentially having *D* depth slices. As a restriction we could enforce the system to only use the same weights and bias in every slice. This divides the total number of parameters by a factor of *W*.*H*. Reducing the amount of parameters can help when one is faced with overfitting problems.

But it is important to know that parameter sharing is not always desirable. Imagine one has prior knowledge about how an image is composed, i.e. every image is taken from the same angle, in the same position and at the same scale. When some regions of neurons look for features at specific locations in the image, parameter sharing will rather work against us than helping us. Parameter sharing in a sense

forces the neurons to only look for the same parts and this is not what we want in such a scenario. An example of such a situation can be input data of human faces. Here different parts of the face can be learned in certain locations. In such a case parameter sharing should be disabled. If by design this is not a requirement, parameter sharing will be beneficial. A boolean for the option of parameter sharing can be used as an additional hyperparameter.

An extra hyperparameter can be a boolean which uses **dilated convolution** instead of regular convolution as suggested by Fisher Yu and Vladlen Koltun [24]. In TensorFlow's jargon this is also called **atrous convolution**. Dilation introduces spaces between every element laying under the filter. Regular convolution has a dilation of 0. The value of dilation can also be considered a hyperparameter instead of just using a boolean and a fixed value. When using dilation in combination with regular filters, the receptive field of the filter, i.e. its range on the image in a step, grows bigger faster after every convolutional layer. This means one can achieve the same result with fewer layers.

The convolutional layer operates on an input of size W_i x H_i x D_i and creates an output of size W_{i+1} x H_{i+1} x D_{i+1}.
The following equations apply:

$$W_{i+1} = (W_i - F + 2P)/S + 1$$
$$H_{i+1} = (H_i - F + 2P)/S + 1$$
$$D_{i+1} = K.$$

When parameter sharing is employed, there are $K.(F.F.D_i)$ weights and K biases in the convolutional layer.

The **forward pass** for the convolutional layer in a CNN [20] is implemented in practice as follows as one huge matrix multiplication:

- One uses an im2col operation which stretches out parts of the input image into columns. This means a part *F*x*F*x*D* gets stretched out into a *F*.*F*.*D* dimensional column vector. From the used stride, zero padding size and the above equations we calculate the width *W* and the height *H*. Then we calculate the product *W*.*H* which calculates the amount of column vectors and create an (*F*.*F*.*D*)x(*W*.*H*) matrix of column vectors. Due to strides these column vectors might have overlapping receptive fields, yet memory constraints put aside this is not problematic.
- An alike stretching scheme is applied for the weights of the filter. Instead of columns, we now obtain rows. For each row we obtain a matrix (second order tensor).
- To perform the actual convolution the network will calculate the dot product between a row of weights and the underlying column of data or output of a previous layer. Not that this is a dot product between matrices and not between vectors!
- Additionally the output volume will get scaled to the desired measurements when necessary.

An addition I/O caching approach can be taken when this method will take up too much RAM or VRAM.

The **backward pass (backpropagation)** [20] is just like the forward pass a convolution, however the filters are flipped in both spatial dimensions (bottom to top, left to right). This in fact is the mathematically correct form of convolution whereas the forward pass actually uses cross-correlation. When performing the backward pass in a CNN, each neuron in the convolutional layer calculates a gradient for its weights. When parameter sharing is used, the amount of computations will drop significantly.

3.3.2 Rectified linear unit layer (ReLU)

This layer performs a nonlinear activation function. Currently the best option is ReLU, which is defined as:

$$ReLU(x) = max(0, x).$$

In the past other activation functions such as sigmoid and tanh were used.

3.3.3 Pooling layer (POOL)

After convolution one can use pooling layers in order to reduce the number of parameters and cut the amount of computation significantly. By reducing the amount of parameters, one also reduces the chance of overfitting.

For the **forward pass** the pooling layer takes an input of size W_i x H_i x D_i and creates an output of size W_{i+1} x H_{i+1} x D_{i+1}. The size of the pooling window is *F*x*F* and the stride is *S*. One places the pooling window over the image and takes the maximum, minimum, average or L2-norm (Euclidean norm) of the underlying values and puts this new value in the corresponding position of the output volume for each depth slice. MAX pooling is currently the most common pooling method since it guarantees better results in practice. *F* and *S* can be considered as hyperparameters for the pooling layer. No zero-padding is employed for pooling. Common values for these hyperparameters are (*F*=2, *S*=2). Less common is the use of overlapping pooling where (*F*=3, *S*=2). Larger pooling windows are not advised, since they cause an undesirable high loss of information due to their aggressiveness.

The following equations apply for a pooled output W_{i+1} x H_{i+1} x D_{i+1}:

$$W_{i+1} = (W_i - F)/S + 1$$
$$H_{i+1} = (H_i - F)/S + 1$$
$$D_{i+1} = D_i.$$

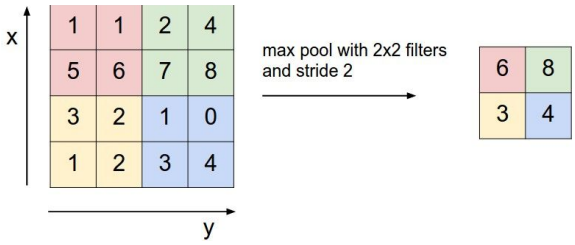

Figure 3.3: Max pooling for a depth slice of the image. Here a 2x2 pooling window with stride 2 is used.

For the **backward pass**, and assuming max pooling, we aim the gradient towards the input which has the maximal value in the forward pass. For min pooling we assume a minimal value and for average pooling we guide the gradient towards the input which has the average value. So additionally, in the forward pass it is necessary to keep a copy of the desired directions.

Whether or not pooling can be abandoned for highly performing generative models (e.g. GAN [23] or VAE [25]) will be discussed later on. When one discards pooling, one needs to use large strides in some succeeding convolutional layers occasionally.

3.3.4 Fully-connected layer (FC)

In this layer each neuron is fully connected to every neuron or data element of the previous layer. Activations are calculated by using the same matrix multiplication and bias addition used in standard neural networks. Hyperparameters for this layer are the same as in fully-connected standard neural networks and include the number of hidden layers and the amount of neurons in each layer.

3.4 CNN architecture

3.4.1 Layer sequences

There are many known layer sequences and architectures which stitch together CONV, RELU, POOL and FC layers in a CNN. The most common layout is the following [20]:

- 1 INPUT layer
- *N* CONV-RELU layers, with *N* a non-negative integer typically in [0,3]
- *M* optional POOL layers after *N* CONV-RELU combo's, with *M* a non-negative integer
- *K* FC-RELU layers, with *K* a non-negative integer typically in [0,2]
- 1 FC layer

To find the right kind of layer sequence it is advised to train and test on data from ImageNet [14]. It is also silly, unless it is a research topic, to make up an own architecture or layer sequence, since it is likely to perform worse than well-known high performing architectures. The best architecture tested on

ImageNet is the one which should be used as a starting point. Later adjustment, mainly through automated Bayesian Optimization [21], can be performed to improve the default architecture.

3.4.2 Common hyperparameter values

Next, we consider frequently chosen hyperparameter values which have proven to yield good results [20].

- **Input width or height:** should be an even number which can be divided by 2 multiple times. Ideally this should be a power of 2, e.g. 32, 64, 128,... or values like 96, 224, 384,... for *W* and *H*.

- **Convolutional layer:**
  - **Stride:** In practice it is advised to use smaller strides for better results. When *S* = 1, downsampling will be handled only by the pooling layers.
  - **Filter size:** Smaller filter sizes such as 3x3 or 5x5 yield better results. Larger filter sizes like 7x7 and greater are normally only used at the first convolutional layer.
  - **Zero-padding:** *P* = (*F* - 1)/2 maintains the input size. When *F* = 3, then *P* = 1. For *F* = 5 we use *P* = 2.

  Thus the most common typical values for these hyperparameters are: *F* = 3, *S* = 1 and *P* = *1*.
  But for 5x5 filters we must use *P* = 2.

- **Pooling layer:**
  - **Pooling method:** max-pooling has proven to yield the best results in practice.
  - **Pooling window size:** using 2x2 pooling windows, thus *F* = 2 is the most common setting. Less common is the use of overlapping pooling where (*F*=3, *S*=2). Larger pooling windows are not advised, since they cause an undesirable high loss of information due to their aggressiveness. Using a 2x2 window already reduces the amount of activations by a factor of 0.75.
  - **Stride:** when *F*=2, then we use a stride of *S*=2 for non-overlapping pooling. For overlapping pooling when *F*=3, we use a stride of *S*=2.

3.4.3 Common architectures

Since the inception of CNNs, multiple known architectures were given birth. The most famous ones were named after their creators. A short non-complete list in chronological order of the most popular architectures:

- LeNet [19]
- AlexNet [14]
- ZF Net [26]
- GoogleNet [27]
- VGGNet [28]
- ResNet [29]

These architectures will not be discussed in detail, but whenever ideas from these existing architectures will be used, we will elaborate more on the considered architectures.

3.5 Time and space complexity optimizations

One general problem with CNNs is the large amount of VRAM memory used on the graphics card which can easily overflow. Culprits of this bottleneck are:

- Volumes used in the CNN such as the input data and the amount of neurons and filters in the network.
- Parameter sizes of the weights, gradients while performing backprop, the learning rate, momentum values, caches when performing certain adaptive learning rate optimization methods,...
- Other memory like batches of image parts, adjusted data parts, generated data in a GAN,...

After pointing out the issues and manually making a calculated guess of al the culprits, before or while training, one needs to consider this amount of gigabytes in comparison to the available amount of memory. This can be done manually in code by listing all the values and calculating the total amount of bytes. Most values in a neural network are floating point values of 4 bytes each, when doubles are used instead of floats we multiply by 8 instead of 4. Now we know the amount of bytes, we can divide by 1024 successively to calculate the amount of kilobytes, megabytes and gigabytes (in this context also called kibibytes, mibibytes and gibibytes respectively). Or it can be monitored by profiling tools of the deep learning library while training. When we get a memory size greater than the available working memory, we keep on **lowering the batch size**, until everything will fit into memory.

Since the first convolutional layer has the most influence on the overall space complexity, it is a common approach to make a trade-off solely on this first convolutional layer. Here higher filter widths and heights of 7 or even 11 can be used instead of 1, 3 or 5, together with a stride of 2 for 7x7 filters and a 4 for 11x11 filters. The reason that the first convolutional layers contribute so much to the space complexity is that their activation results are kept in memory since they are necessary in the computations of backprop. [20]

Chapter 4: Improving a neural network

In this chapter we will present a number of general techniques which can be applied to a neural network to improve the accuracy and overall performance. These techniques will be general in the sense that they can be utilized in every domain; ranging from natural language programming to higher dimensional image classification or generation. Other techniques which only apply to our specific problem will be addressed later. It should be noted that these last techniques depend highly on the chosen domain and cannot be easily transferred to other domains. [30]

4.1 Input data

4.1.1 Splitting the data set

Traditionally one might take all the available data and carve off some portion of it to be the training set, another portion of it might be the development set and the final portion will be the test set. As a workflow one can use the training set for training a set of considered models. The development set will be used to see which model will perform best. The best model will then be chosen and for final evaluation one uses the test set in order to get an unbiased estimate of the algorithm performance.

As a first rule of thumb one has to make sure that the development and test sets come from the same data distribution, but to get a largely unbiased result the data distribution of the training set has to differ enough from the others sets in order to avoid overfitting of the model.

In the previous era of machine learning when large amounts of available data were unlikely, it was common practice to split all the available data according to a 70/30 percent split ratio which means that 70% of the available data will be used for training and the other 30% will be the test set. When one opts for a development set, this ratio will be 60/20/20 for instance. This is good practice when the data set has a manageable amount of entries; maximally in the order of 10 000 examples.

In the modern age of big data where millions of examples are available, one has to consider smaller percentages for the development and test sets. When development sets are becoming too large, comparison between different models will take up too much unnecessary time. One might see that on a scale of 1 million examples in total, one has enough with 1% of it for the development set and test sets.

4.1.2 Normalizing input data

Imagine a training set of feature vectors $\vec{x}$ and a cost function $J(\vec{\theta})$ for the neural network which needs to be optimized. For sake of simplicity assume the weights *w* are scalars instead of using an

n-dimensional vector. When normalizing training data, we calculate the mean of all the examples and subtract it from every training example:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x^{(i)}$$
$$x_{new} = x_{old} - \mu$$

After normalizing the mean to 0, we also normalize the variance. This is done by calculating the variance σ^2 of the updated training examples and dividing each updated training example by it. Here the mean is already subtracted from every $\vec{x}^{(i)}$:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}[x^{(i)}_{zero-mean}]^2$$
$$\vec{x} = \vec{x}_{zero-mean}/\sigma^2$$

When normalizing the test data, one should use the same μ and σ^2 calculated from the training set.

Normalizing input data, so each feature is on the same scale, can help to improve learning. When using unnormalized inputs, we can have a cost function which will take the form of an elongated bowl, due to different ranges of the features $\vec{x}^{(i)}$. Normalizing the input data on the other hand produces a more symmetric cost function. The elongations due to unnormalized features makes learning slower, since it takes longer for gradient descent to reach the minimum of the cost function, although this will depend on the starting point.

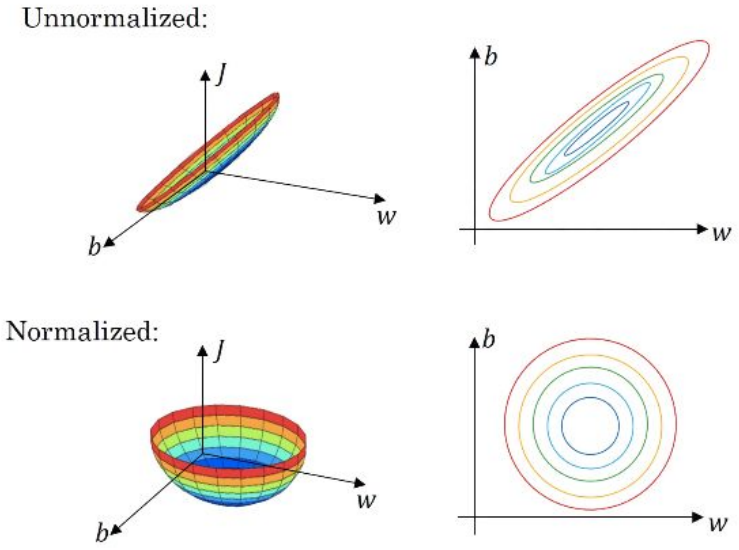

Figure 4.1: (Top) a cost function *J*(*w*, *b*) due to unnormalized training data. Notice the elongation in particular dimensions. At the right you can see a corresponding contour plot. (Bottom) a cost function *J*(*w*, *b*) due to normalized training data. Notice the more symmetric bowl shape of the cost function.

4.2 Bias and variance

For sake of simplicity consider a binary classifier which can classify two-dimensional data entries into one of two possible classes. When a model has a high bias, we can see that the model is underfitting for the data. This results in a rough separation of the data which has a high error rate at any data distribution. On the opposite end of the spectrum, one can employ a model which separates the data in a way that is not general enough for other data sets. We see that the model overfits the training data and has high variance; in other words the model is not general enough to consider data sets with other distributions. So on other data sets the errors will not be similar, hence the term high variance. Both groups of models are not desirable. A combination of high bias and high variance will be considered the worst. A model which ensures a correct fit for almost any sort of data distribution will be most favourable. Here one has low bias and low variance and error rates will be at a human level. This ensures an optimal error of approximately 0% for tasks tailored to humans. Higher errors are tolerable when the optimal error is higher as well, although striving for super-human behaviour can be highly beneficial for certain tasks.

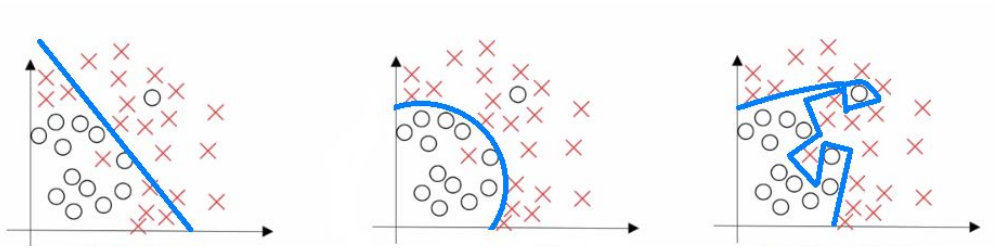

Figure 4.2: High bias and underfitting at the left, a correct fit in the middle and high variance and overfitting at the right.

We can state that when looking for bias problems, one has to consider a model which ensures low error rates for training and development sets. Possible solutions would be to increase the number of layers in a neural network, more advanced optimization algorithms or more complex architectures in general. When looking for variance problems after solving bias problems, one has to consider a model which ensures error rates that remain relatively stable across different data sets. Strategies to combat high invariance involve employing more complex architectures, provision of more data or regularization, which will be discussed in the next section. In the past when neural networks were not as common as today due to computing limitations, one had to consider a trade-off between bias and variance. Reducing the other would enlarge the other in a mutually exclusive fashion. This was true for classical machine learning models, but this is not longer for deep neural networks. The reason for this is that there exists a combination of techniques which provides the best of both worlds. For instance one could battle high variance by using more data. High bias could be tackled by employing a more hidden layers. Still high variance can be prevented by also applying regularization accordingly when a network takes greater proportions.

4.3 L2 Regularization

As mentioned before one can apply L2 regularization [30] to get rid of high variance and overfitting. Let λ be a hyperparameter for regularization which needs to be tuned appropriately using the development data set. Then we need to add an extra term to the cost function of the neural network. The regularization term is defined as:

$$\frac{\lambda}{2m}\sum_{l=1}^{L-1}\|\Theta^{(l)}\|_F^2 = \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{n^{(l)}}\sum_{j=1}^{n^{(l+1)}}(\Theta_{ji}^{(l)})^2 = \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sqrt{tr(\Theta^T\Theta)}$$

Here *w* can be considered as a ($n^{(l)}$ x $n^{(l+1)}$) matrix with $n^{(l)}$ hidden neurons in hidden layer *l* and $n^{(l+1)}$ hidden neurons in hidden layer *l*+1. The used norm is the Frobenius norm for real values. It is defined as the square root of the sum of the absolute squares of the matrix elements. Let us remind us of the fact that (*x*) in superscript does not stand for exponentiation, but instead serves as an index. When computing the gradient of the cost function $J(\vec{\theta})$, we need to add an extra term to the original

$$\frac{\partial}{\partial\Theta_{ji}^{(l)}}J(\vec{\theta}).$$

This extra term, according to the rules of partial differentiation, will be:

$$\frac{\lambda}{m}\Theta^{(l)}.$$

The reason why L2 regularization prevents overfitting is that one shrinks the Frobenius norm of the weights. This means that the use of a higher regularization parameter λ results in weights Θ_{ji} which are smaller and some might approach zero. When the weights get smaller for a larger λ, the result of the activation function also gets smaller and behaves more linear than non-linear. Less non-linearity induces less overfitting, since only more linear functions can be learned. This also means that the corresponding neurons with small weights would have less impact, so the network can be considered the same as a smaller one. A smaller neural network creates a shift from high variance to low variance and higher bias. Ideally the regularization parameter λ should be tuned so that we have a case where the variance gets low enough and the bias stays low enough.

4.4 Dropout regularization

Dropout regularization [31] is a technique which mimics the process in the human brain where random neurons tend to switch off. Each neuron has a probability *p* in [0,1] of being kept. The higher the probability the less dropout in the network. This probability is considered to be a hyperparameter which needs to be tuned.

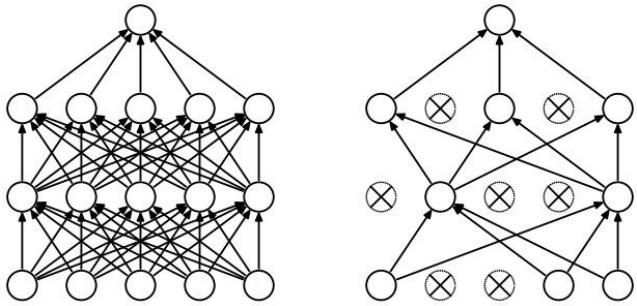

Figure 4.3: At the left we see a standard neural network, while at the right we see this network after application of dropout.

For a simple dropout algorithm we iterate through the hidden layers of the neural network while doing the forward pass in the training phase, calculate if each individual neuron drops out according to the keep-probability *p* and turn the neurons off when said. Ingoing and outgoing edges in the network will also be temporarily removed. The result again is a smaller network comparable to L2 regularization. During testing no dropout happens, but one needs to scale (multiply) the outputs of the hidden layers by the keep-probability *p*. The reason for doing this is because at test time all neurons are kept, and the outputs of neurons at test time need to be equal to the outputs of the training phase.

Another perspective of understanding tells us that the expected output from a neuron with activation function *x* during training will become:

$$px + (1 - p)0 = px.$$

In other words: there is a probability *p* of keeping the neuron and a probability (1-*p*) of dropping the neuron and the output will be 0. During the forward pass of the testing phase all neurons are being kept, so one must adjust *x* to *px* in order to generate a similar output.

The problem of the simple dropout algorithm introduced earlier is that one must scale the activation functions of the neurons by *p* while testing. Taking into account that machine learning models are being used in production where performance is critical, it is more desirable to move the scaling part into the training phase, but now as a division by *p* rather than a multiplication. As a consequence no additional operations are needed during the forward pass in test-time. The resulting algorithm is called inverted dropout and is used in practice.

Note that it is also possible to perform dropout on input nodes. This is the same as applying a binary mask to the input data, but is not as common. More common is the use of different keep-probabilities for different hidden layers. For layers where one does not worry about overfitting this probability can be chosen relatively higher or closer to 1.0, so less dropout occurs in those layers.

When using dropout a consequence is that the cost function $J(\vec{\theta})$ is not well-defined since one is turning on and off nodes randomly in every forward pass during training. In order to inspect monotonically

decreasing behavior of the cost function, it is advisable to set all keep-probabilities to 1.0 for debugging gradient descent and later when doing the actual training, one adjusts the keep-probabilities accordingly.

4.5 Weight and bias initialization

After building up a network and loading preprocessed data, we need to train the network. But before we can do this, the parameters such as weights and biases need to be initialized. [32, 33]

4.5.1 Random weight initialization

It is a common mistake to initialize all parameter values to zero. In fact when initializing parameters, the researcher will not know in advance what the values of the weights will become. When the normalized range is centered at zero it would seem a logical choice to initialize as zero, since a valid assumption can be that probabilistically the chances for weights to be negative is expected to be the same as to be positive. There is one major problem: when each neuron in a network will deliver the same output, then during the backward pass they will also produce the exact same gradients. Weights will be updated to the same values and in fact the network fails to learn since it is unlikely that the weights of a neural network will always be the same for a given problem.

A better approach is to use small random floating point values coming from a pseudo-random number generator which will sample in an interval around zero. When they are small in absolute value, they are in the vicinity of zero, but will still produce different gradients in the backpropagation phase and will deliver most likely unique updates. The chosen sampling distribution can be a uniform distribution or a Gaussian distribution with a mean of zero and unit standard deviation. Whether the samples are taken from a uniform or Gaussian distribution does not have implications on the accuracy of the network.

Since for vectorization purposes we use vector math, it is common to consider a weight vector for every neuron. Instead of taking samples from a one-dimensional Gaussian distribution, we employ its multidimensional variant. Again a uniform distribution is also common.

The problem with all of the above is that when we use random weight initialization, the larger the dimensionality of the input data, the smaller the weights need to be and this is not the case. The reason for this is that the dot product, which will serve as the input for the activation function, is not allowed to blow up in size. In order to ensure this will not happen and we will have a neuron output variance of 1, we will scale every weight vector by dividing its components by $\sqrt{n}$, where n is the number of input components. This will make sure that every neuron will originally have an equal output distribution and convergence will be faster.

We can explain the choice of the division factor $\sqrt{n}$. Consider a collection of input components x_i and corresponding weights w_i. Next we consider its dot product z which essentially is the activation of a

network before application of a non-linear activation function such as ReLU. Next, we compute the variance of *z*:

$$
\begin{aligned}
&Var(z) = Var(\sum_{i=1}^{n} w_i x_i) \\
&= \sum_{i=1}^{n} Var(w_i x_i) \\
&= \sum_{i=1}^{n} \mathbb{E}[(w_i x_i - \mu_{xw})^2] \ , \ \mu_{xw} = \mathbb{E}[w_i x_i] \\
&= \sum_{i=1}^{n} [[\mathbb{E}(w_i)]^2 Var(x_i) + [\mathbb{E}(x_i)]^2 Var(w_i) + Var(x_i) Var(w_i)] \\
&= \sum_{i=1}^{n} Var(x_i) Var(w_i) \\
&= n Var(w) Var(x)
\end{aligned}
$$

In the first step we substituted the dot product of *x* and *w* for *z*. Next, we see that the variance of a sum equals the sum of its variances. We also rewrite the variance formula in terms of expected value. Since we assume zero mean inputs and weights after input and batch normalization, we have

$$\mathbb{E}[x_i] = \mathbb{E}[w_i] = \mu_{xw} = 0.$$

In the last step, due to batch normalization, we expect an equal distribution of w_i and x_i values, so each term of the sum is the same, hence replacing the sum by a multiplication by *n*. Now we see that if we want to make sure that the variances of *x* and *z* are identical, then we need the variance of *w* to be 1/n. For each possible value *w* and scalar value *a* we have:

$$Var(aw) = a^2 Var(w).$$

Thus we need to randomly sample the weights *w* from a unit normal (Gaussian) distribution and divide each sample by $\sqrt{n}$. This way the variance will be:

$$Var(w) = 1/n.$$

In practice, most recently He et al. [34] suggest the use of ReLU activation functions and weights sampled from a unit normal (Gaussian) distribution as:

$$W = rand_{normal}(shape = (n_{in}, n_{out}), stddev = \sqrt{\frac{2}{n_{in}}}),$$

where n_{in} and n_{out} are the number of input and output of nodes. For tanh or sigmoid activation functions Xavier initialization [35] was empirically proven to be the best for weight initialization rather than the approach of He et al:

$$W = rand_{normal}(shape = (n_{in}, n_{out}), stddev = \sqrt{\frac{2}{n_{in} + n_{out}}}),$$

where n_{in} and n_{out} are the number of input and output of nodes.

4.5.2 Bias initialization

Just as before with weights it would seem to be common practice to initialize the biases to 0 or a tiny constant value like 0.01 or smaller. And this turns out to be true and it will suffice to use random initialization for the weights only since the weights themselves will already cause different gradients during the backpropagation. It is however best practice to just initialize the biases to 0, since performance tends to be better than using other small values as documented in some use-cases [32, 33]. Proponents of using small values argue that they will cause every ReLU activated neuron to signal right from the start and will produce a gradient during backprop. When in doubt, it is best to start with biases of 0 and change this later to another small value such as 0.01 whenever necessary. [33]

4.6 Choosing an optimization algorithm

In order to compute gradients for a cost function, which are used later to update the weights θ, one needs to use a suitable optimizer. In TensorFlow the following optimizers are currently the most popular:

- Gradient Descent [18]
- Momentum (with Nesterov Acceleration) as an extension to Stochastic Gradient Descent (SGD) [36]
- Adaptive learning rate methods [37]:
    - Adagrad [40]
    - Adadelta [41]
    - RMSprop [36]
    - Adam (RMSprop, bias-correction and momentum) [38]

Currently the Adam optimizer is being advised for deep neural networks used on sparse data, but one can use others to compare performance for verification. As an alternative when data is not sparse but dense one can use the Momentum optimizer with Nesterov Acceleration turned on. There is currently no unanimity which algorithm performs the best in every situation.

The Adam optimizer can be seen as an extension of RMSprop by addition of bias-correction and momentum. Although the Adam optimizer is generally preferred, we can see that the Adadelta, RMSprop, and Adam algorithms are alike. Therefor they can be exchanged when necessary. Due to the correction of bias by Adam, we see that the Adam optimizers dominates RMSprop at the end of the process when gradients turn out to be less dense. Thus Adam would be the best choice for sparse data.

By the work of Wilson et al. in 2017 [37] we know that for certain applications Stochastic Gradient Descent combined with momentum entails convergence to a more general optimum instead of the adaptive learning rate methods where there is a variety of suboptimal local optima. Therefor the reached optimum will not be ideal and performance will be worse depending on the application and the deviation from the actual global optimum. Thus one needs to bear in mind that the adaptive learning rate methods described here are not perfect and one eventually needs to switch to the other approaches when they perform better in a certain application. The promises of rapid convergence made by Adam are not the only factor to consider.

In short one should start by applying the Adam optimizer for rapid convergence on deep neural networks. Then one exchanges Adam for other adaptive learning rate methods when needed. When the found local minimum deviates excessively from the desired result one can change to SGD combined with momentum for accuracy in spite of convergence speed. For faster convergence we can add Nesterov Acceleration into the mix. The downsides are that it is sensitive to initialization of weights of biases, needs an annealing schedule for decrease in learning rate (learning rate decay), converges slower than Adam and can reach saddle points instead of local optima. The reason for slow convergence is that there are more long plateaus. Adaptive learning rate methods solve this problem by using a larger learning rate at the start and a smaller one at the end of a long plateau. If all of the above impose grave problems, we will stick to Adam. Although the writers of the Wasserstein GAN paper [39] suggested the use of RMSProp for the discriminator network instead of momentum centered optimization methods including Adam, yet I did not came across any theoretically formalized proof. So for the discriminator we will start by using RMSProp. For the generator we will stick to Adam as a starting point. Other people argue that the choice of the algorithm is free and will depend on the experience of the researcher with a certain algorithm and its hyperparameter tuning.

A more detailed explanation of the inner workings of the above optimization algorithms is given below avoiding mathematical formalism as much as possible in favour of intuition since TensorFlow does this for us.

4.6.1 Gradient descent

Instead of immediately guessing the best solution to a given objective, we guess an initial solution, and iteratively step in a direction closer to a better solution with lower cost. This direction is defined by computing the negative gradient of the cost function at the given point and moves by doing an update of the weights. The movement is determined by a vector which direction is in the negative gradient and its magnitude is determined by the learning rate. The algorithm just repeats this process with decreasing learning rate, until it converges to a solution that is considered suitable, i.e. a local minimum. [18]

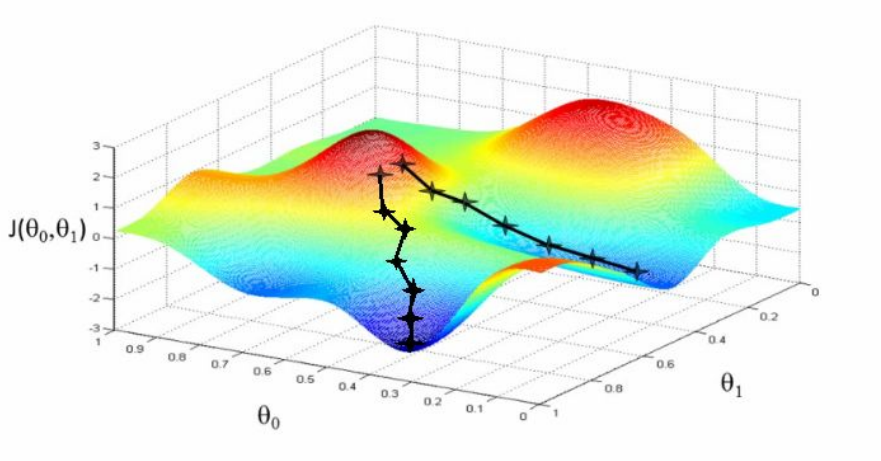

Figure 4.4: Gradient descent starting from an higher cost $J(\theta_0, \theta_1)$. By changing the two weights θ_0, θ_1 as described above we will reach a local minimum depending on the initialization of the weights, hence the different paths. This idea extends to the case of more than two weights.

Since an approximate finite difference calculation of the gradient is merely an approximation and is a costly operation, we will omit this method and explain the exact and less computationally intensive analytical solution. Here we apply the rules of calculus to partially differentiate the cost function, for one example data vector, with respect to the weights. We will omit the exact mathematical formalism, but in code this is translated as counting the amount of classes that contributed to the loss. Next, we scale the data vector by this amount and obtain the gradient.

Note that traditional gradient descent computes the gradients of the loss function respecting the parameters for the entire training data set for a given number of epochs. An **epoch** is one pass through the training data.

Since we need to compute the gradients for the whole data set for just one update, this is relatively slow and even unmanageable for data sets that do not fit in working memory. To get around this, we can turn over to stochastic gradient descent. [36]

4.6.2 Stochastic Gradient Descent

This is where we perform a parameter update for each training example and class label at a time. For each epoch we just add a loop over our example data vectors and compute the gradient with regards to each and every example. These more frequent updates with high variance cause the cost function to oscillate more intensely. It helps it jump to new and possibly better local minima. But SGD also complicates convergence to the exact minimum since it does not fix overshooting due to the high amount of oscillations. This can be solved by decreasing the learning rate or other techniques such as using momentum. Another downside is that we will lose the possible speedups from vectorization, since we are only processing one training example at a time.

4.6.3 Mini-batch gradient descent

An improvement would be to use mini-batch gradient descent [30], as it takes the best of both worlds by performing an update for every subset (mini-batch) of training examples. Thus the mini-batch size will serve as an hyperparameter which can be tuned. But normally this hyperparameter will not be optimized by tuning. Instead one needs to set it according to the available working memory and is usually a power of two typically ranging from 64 to 512. Other sizes are also possible, but a power of two ensures optimal vectorization at the processing unit. The larger the mini-batch in SGD, the more the variance of the stochastic gradient updates will get reduced, since we take the average of gradients in the mini-batch. This will result in larger step sizes, so the optimization process will reach a minimum faster.

When not using mini-batch gradient descent, we expect the cost to decrease monotonically after every iteration (epoch) through all the training examples. Using mini-batches, at every iteration we are training on a different subset (mini-batch) of training examples. Here we expect an overall decrease of the cost after every iteration, but in a more noisy oscillating pattern, yet not as much as SGD.

Training in mini-batches is usually the method of choice for training neural networks and we usually use the term stochastic gradient descent even when mini-batches are used, although SGD in its pure form is actually using a mini-batch with only one example data vector. On the other side of the spectrum, it is also possible to use a mini-batch size of the entire training set, which will result in the batch gradient descent algorithm. The sweet spot lies somewhere in between, since we will reach faster learning due to vectorization and we can still make progress without having to wait for the entire training set to be processed.

4.6.4 Momentum

The oscillations in plain old SGD make it hard to reach convergence though. Thus one needs to reduce the oscillations in order to reach faster convergence. Therefore a technique called momentum [42] was invented that lets us navigate along the relevant directions, ensures faster convergences and softens the oscillations. In short it does this by computing an exponentially weighted average or linear interpolation

of the gradients and using this averaged gradient to update the weights. In pseudo-code it is done as follows for each mini-batch:

$$\vec{v}_t = \beta\vec{v}_{t-1} + (1-\beta)\nabla_{\vec{\theta}}J(\vec{\theta})$$ (with β = 0.9 as a common value for optimal results)

$$\vec{\theta}_{new} = \vec{\theta}_{old} - \alpha\vec{v}_t$$ (with α being the learning rate).

Here we first compute the gradient of the cost function *J* for the weight and bias vectors on the current mini-batch. Before looping over the mini-batches, we initialize our v-vector at time t as a zero-vector. In other words: momentum adds a fraction of the update vector of the previous step to the other fraction of the current step. This propagating effect amplifies the speed in the correct direction. It is comparable to momentum from classical physics. When a ball is pushed down a hill, it builds up momentum, meaning its speed increases and it becomes harder to change directions. In the same way our momentum term increases for dimensions whose gradients point in the same direction, and reduces updates for dimensions whose gradients change direction, thus damping oscillations. The force acting on the ball equals the negative gradient of the potential energy of the ball.

4.6.5 Nesterov accelerated gradient (NAG)

Once we get close to our goal point the momentum is usually pretty high. The algorithm does not know that it should slow down and could miss the minima entirely. In the momentum method we compute the gradient, then make a jump in that direction amplified by the previous momentum. In this method we do the same process but in a different order. We first make a big jump based on our previous momentum (brown vector), calculate the gradient based on this change in parameters, then make a correction (first red vector), which forms a NAG update (first green vector) [43]. The process is then repeated. So essentially we look one step further in the future by computing the gradient using the future positions of the weights instead of the current weights. This technique prevents us from going too fast and we are more responsive to changes.

Figure 4.5: Nesterov Acceleration as described above for two update steps.

4.6.6 Adagrad (Adaptive Gradient)

Here we apply Nesterov acceleration to our learning rate and this allows the learning rate to adapt based on the weight parameters. This way Adagrad [40] will make large updates (with high learning rates) for parameters corresponding to infrequent features and small updates (with small learning rates) for the frequent case. It uses a different learning rate for every parameter at a given time step based on the past gradients that were computed for that parameter. In Adagrad, according to Duchi et al. in 2011 [40], the learning rates will be individually adapted for each parameter by performing a scaling

operation inversely proportional to the square root of the sum of every past squared gradient. When a parameter has a large partial derivative of loss, it will correspond to a fast decreasing learning rate and vice versa. As a result, there will be greater movement in flatter directions of the cost function.

In order to use all past squared gradients, we add them to a sum, which we call the cache:

$$cache_{new} = cache_{old} + [\nabla_{\vec{\theta}} J(\vec{\theta})]_t^2.$$

Next, we will use the cache for element-wise normalization in the parameter update step. This will ensure that parameters which have large gradients will have a smaller resulting learning rate and parameters with small or infrequent updates will have a larger resulting learning rate. The square root of the cache, instead of just a division by the cache, is needed because without it it is empirically shown that algorithm performance gets worse. The addition of an extra vector ε is again used to prevent division by a zero-vector and provide ensurement of numerical stability. Typically, component-wise $\varepsilon = 10^{-8}$.

$$\vec{\theta}_{t+1} = \vec{\theta}_t - [a[\nabla_{\vec{\theta}} J(\vec{\theta})]_t / (\sqrt{cache} + \varepsilon)]$$

The main intention behind the Adagrad algorithm is that it will induce fast convergence to an optimum on convex cost functions and therefore will have some favorable theoretical properties for convex optimization purposes. Application to nonconvex cost functions will not be as beneficial, since the chosen path will pass through many different parts of the landscape and will stop in a locally convex bowl. Adagrad will let the learning rate decay given a history of all the squared gradients, this decay can be too aggressive and before it can reach such a locally convex bowl, the learning rate can be too small, so fast convergence will not be possible or can even cause the model to stop learning. In general Adagrad can deliver good performance on some deep learning models, but certainly not in nonconvex cases.

As a plus this means that we do not have to manually tune the learning rate. The problem though is that the learning rate is always decreasing. At some point the learning rate could get so small that the model just stops learning entirely. This can be solved by Adadelta [41].

4.6.7 Adadelta

Adadelta [41] is an improvement on Adagrad. In Adagrad we were constantly causing a monotonically decreasing learning rate. This decrease should be less aggressive. Instead we will restrict the window size of accumulated past gradients to a fixed window size *w* instead of using all the past squared gradients. Since storing the *w* past squared gradients directly is inefficient, we will define the sum of gradients recursively via a decaying average of all past squared gradients. We call this a running average, which at a time step *t* will depend only on the previous average and the current gradient. This way only 2 variables need to be stored. Mathematically this is expressed as a linear interpolation:

$$\mathbb{E}[(\nabla_{\vec{\theta}} J(\vec{\theta}))^2]_t = \gamma \mathbb{E}[(\nabla_{\vec{\theta}} J(\vec{\theta}))^2]_{t-1} + (1-\gamma)[\nabla_{\vec{\theta}} J(\vec{\theta})]^2.$$

Here we use $\gamma = 0.9$ similar to the momentum term β. The rest of the algorithm is similar to Adagrad.

4.6.8 RMSprop

RMSprop, or Root Mean Square Prop [36], is another optimization algorithm which can speed up gradient descent. In pseudo-code it is done as follows for each mini-batch:

$$\vec{s}_t = \beta \vec{s}_{t-1} + (1-\beta)[\nabla_{\vec{\theta}} J(\vec{\theta})]^2$$ (here we use an element-wise power operation)

$$\vec{\theta}_{new} = \vec{\theta}_{old} - \alpha \nabla_{\vec{\theta}} J(\vec{\theta}) / \sqrt{\vec{s}_t + \varepsilon}$$ (with α being the learning rate).

This means that the RMSprop algorithm is a variation on the Adagrad method, where we try to reduce the aggressive monotonically decreasing learning rate α. RMSprop uses an exponentially weighted average of element-wise squared gradients. The learning decay rate β lets decay the old history. Therefore the algorithm will converge faster after finding a convex part of the cost function. In a sense it functions like it is performing Adagrad, starting from the convex part in order to find its minimum. Typical values for β are 0.9, 0.99 or 0.999. Here ε can be used as an extra small additive vector to prevent division by zero or to increase numerical stability, e.g. $\varepsilon = 10^{-8}$. The division by the square root will ensure that when the gradient of the weights is small, we will divide by a small number which will result in larger gradient descent steps and when the gradient of the biases is large, we will divide by a large number which will result in smaller gradient descent steps. This will reduce smaller oscillations and will provide faster convergence to a minimum. Due to its reduction of oscillations, it is also possible to use a higher learning rate α, so convergence can be even faster.

In practice RMSprop yields great results and it proves itself to be one of the preferred optimization algorithms for deep neural networks.

4.6.9 Adam (Adaptive Moment Estimation)

The Adam optimizer [38], or Adaptive Moment Estimation, combines the ideas of Momentum and RMSprop, although there are some differences. In pseudo-code the Adam optimization algorithm goes as follows:

$\vec{v}_0 = \vec{0}, \vec{s}_0 = \vec{0}$ (zero-vectors)

$\forall t \in [1, N]$: (with *t* a certain mini-batch)

$\vec{m}_t = \beta_1 \vec{m}_{t-1} + (1-\beta_1)\nabla_{\vec{\theta}} J(\vec{\theta})$ (β_1 is the Momentum decay rate)

$\vec{s}_t = \beta_2 \vec{s}_{t-1} + (1-\beta_2)[\nabla_{\vec{\theta}} J(\vec{\theta})]^2$ (β_2 is the RMSprop decay rate)

[bias correction]

[parameter update]

Here we are calculating learning rates for each parameter θ, while also storing momentum changes for each of them separately. We calculate the first moment estimation, i.e. the exponential weighted mean, and the 2nd moment, the uncentered variance, of the gradients respectively. Then we use those values to update the parameters just as in Adadelta. If we were to visualize these optimization algorithms during the learning process, we would see that the adaptive learning rate methods quickly find the right direction and converge faster, while momentum and the Nesterov accelerated gradient methods tend to go in less favorable directions.

Physically speaking Adam is the equivalent of rolling a heavy ball with friction down a hill. Since the first and second moments are biased towards zero in the beginning, the bias correction is implemented as follows:

$$\vec{m}_t^{(corrected)} = \vec{m}_t/(1-\beta_1^t)$$
$$\vec{s}_t^{(corrected)} = \vec{s}_t/(1-\beta_2^t).$$

The parameter update is implemented like so:

$$\vec{\theta}_{new} = \vec{\theta}_{old} - [\alpha m_t^{(corrected)}/(\sqrt{s_t}+\varepsilon)].$$

Typical values for the hyperparameters are:

β_1 = 0.9
β_2 = 0.999
ε = 10^{-8} .

In general, Adam is seen as being reasonably robust to the chosen hyperparameter values, yet the learning rate α might need some additional hyperparameter tuning.

4.6.10 Learning rate decay

One of the simplest ways to decrease the learning rate with each epoch (iteration through the training set), is to use the following formula for computing the decreased learning rate α:

$$\alpha = \frac{1}{1+(\beta.E)}\alpha_0.$$

Here α_0 is the initial learning rate, β is the decay rate and *E* is the current epoch number starting from 1. This results in a monotonically decreasing function.

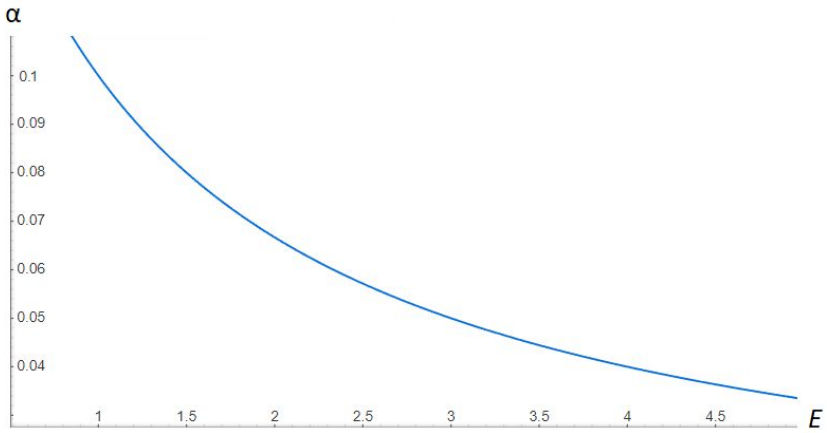

Figure 4.6: Monotonic learning rate decay as the number of epochs *E* increases. Here the following values are used: α_0 = 0.2 and β = 1.

There are also other possible implementations for learning rate decay, e.g. exponential learning rate decay which can be implemented using the following formula:

$\alpha = \beta^E \alpha_0$, where *E* is the epoch number starting from 1 and β = 0.95.

4.7 Order of training data

4.7.1 Random shuffling

As a general rule of thumb in order to reduce bias of the chosen optimization algorithm, one needs to avoid handing the training data to the model in a particular order every epoch. Therefore it can help to shuffle training data randomly before starting an epoch or after performing an epoch.

4.7.2 Curriculum training

Another way to improve convergence speed is to order the training data in a meaningful order. Curriculum learning does this ordering based on how humans learn. The learning process will be planned by starting to learn the simpler, more prototypical concepts first and later expanding to the more complex concepts, which are based on the previously learned simpler concepts.

This method of learning was experimentally verified by Bengio et al. [44]. It is also shown by Khan et al. in 2011 [45] that curriculum learning corresponds to the way humans learn and teach. Results by Basu and Christensen in 2013 [46] also show that curriculum training yields better results than using a standard uniform example sampling.

Due to the fact that manually ordering the training data for the criteria set by curriculum learning is time consuming, we will not employ this method. Instead we will try random shuffling of training data after each epoch whenever benefits arise. Curriculum learning can be considered an extra hyperparameter when one wants to further improve the performance of the model.

4.8 Hyperparameter tuning

During training time there are a great amount of hyperparameters which can be set by the researcher and have an impact on the overall performance of the network. Some of the most popular hyperparameters in deep neural networks are in order of importance:

- The initial learning rate and the scheme or formula it uses for decay.
- The amount of filters and their size.
- The amount of convolutional layers.
- The amount of fully-connected layers or hidden layers.
- The amount of features or neurons in each of those layers.
- The mini-batch size.
- The amount of regularization or “keep probability” in dropout.
- The momentum decay rate β or β_1, although when using Adam 0.9 yields good results.
- The RMSprop decay rate β_2, although when using Adam 0.999 yields good results.
- ...

The hyperparameters listed above can have a significant impact on the performance but it is worth mentioning that there are still other hyperparameters which contribute less to the overall performance of the network. For instance hyperparameters involved in adaptive learning rate methods such as momentum have less impact, but can be tuned for extra performance.

A minor downside of hyperparameter optimization is that it can take up several hours, days or even weeks to find an optimal set of hyperparameters due to repeated training. It is therefore important to find an optimization strategy which searches for optimal hyperparameters as efficiently as possible. It is worth mentioning that due to it being highly costly to evaluate hyperparameter performance, we want an approach which tries to minimize the evaluation, only performing it when necessary. Inspection is very costly because one has to train the network first and performance needs to be measured on a test set.

4.8.1 Manual hyperparameter tuning

There are many methods for finding an optimal hyperparameter tuning. The easiest, yet most tedious way, is manually tuning a subset of the hyperparameters, trying to find an increase in performance, trying a new subset from there and repeating the process. This can be considered as a hand-executed greedy approach but can be fit into an algorithm as well. The main difference between manual and

automatic execution is that as a human we can form a more intricate intuition for guiding the search and can make better decisions than an automated greedy approach. The downside of this method is that it is a very annoying and tedious process for a researcher. The actual optimal hyperparameter values can also differ significantly from the build-up intuition, so they can be overlooked. Therefore we will omit this approach and try to find better alternatives.

4.8.2 Grid Search

In Grid Search [47] we consider a desired range for each hyperparameter and split these ranges into separate uniformly spaced values which essentially forms a grid of values. Next, we exhaustively try all possible combinations. This approach can be considered a brute-force approach and as such can take away too much processing time when dealing a reasonable amount of hyperparameters due to the exponentially increasing time complexity (cfr. curse of dimensionality [5]). As a general rule of thumb this method is not advised and serves as a last resort.

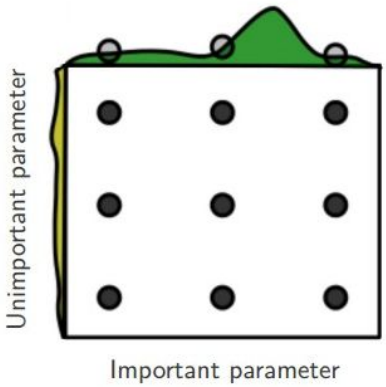

Figure 4.7: Grid Search for 2 hyperparameters in a 2D hyperparameter space. Note that using a strict layout, we can miss optimal values laying in between as indicated by the green peak in the horizontal dimension.

4.8.3 Random Search

In Random Search one tries a handful of possible combinations of hyperparameters in a random fashion. The problem is that it is comparable to searching for a polar bear in a snowstorm; for a large amount of parameters the probability for reaching an optimal solution will approach to zero. Yet it is still considered a better approach than Grid Search as suggested by Bergstra and Bengio [48]. If the sampling intervals are well-chosen from past experiences, chances will greatly improve and random search can even be considered king. Note that when sample points are not single numbers but actual vectors, we need to sample from a hypersphere instead of a one-dimensional interval.

As a plus the actual implementation is also less difficult compared to Grid Search and will deliver results faster. Another plus is that it is an embarrassingly parallel algorithm; we can simply start a Random

Search on each processing unit separately, evenly distributed over multiple workstations or computational cluster nodes.

The larger the neural network, the longer training will take. Each test of a combination of hyperparameters will induce another training and test phase and as such training and testing will extend from a few hours to a couple of days or even weeks. In comparison to Grid Search, Random Search can greatly reduce the considered search space.

Next to uniform random sampling we can also employ importance sampling by taking random samples from a known probability density function. This could for example be a Gaussian distribution, or in general a well-biased distribution.

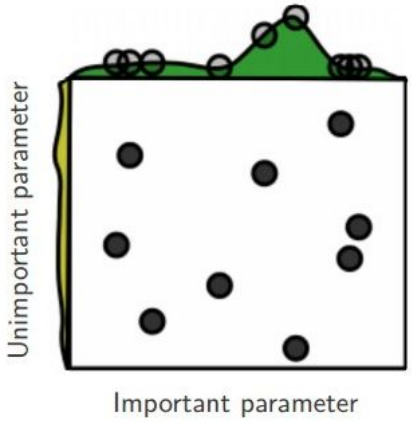

Figure 4.8: Random Search for 2 hyperparameters in a 2D hyperparameter space. Note that using a random sampling scheme, we are less likely to miss optimal values as indicated by the green peak in the horizontal dimension. A more dense scheme in the dimension of the important hyperparameters can be employed.

4.8.4 Hyperparameter intervals

It is considered best to search for optimal hyperparameters on a base 10 logarithmic scale when dealing with initial learning rate and regularization strength. Instead of sampling uniformly in an interval of the samples, we take a base of 10 and sample its exponent uniformly in an interval. The reason for using a logarithmic scale is because we want to adjust a decrement proportionally to the current amount. Thus the absolute value of the decrement for small values will be significantly smaller than the decrement for large values. The main cause here is that the learning rate gets multiplied by the gradient in an update step. For dropout strength, the number of hidden layers, filter sizes, etc. on the other hand we sample directly in the standard scale.

It is also important to note that it is desirable to have the optimal values for an hyperparameter to be in the center of the chosen interval or else they might get overlooked when they are not part of the interval. So the actual choice of the interval is key. Start from a coarse interval in the first epoch, and

sample in a more narrow interval around the best values from the previous more coarse interval for the next couple of epochs (e.g. 5 epochs). Later on a more detailed search can happen in the next epochs. [22]

4.8.5 Bayesian Hyperparameter Optimization

In Bayesian Hyperparameter Optimization [49] one tries to find algorithms for efficient exploration of the search space of hyperparameters and can be considered a research domain in itself. A major plus is that this strategy will take less computational time than Grid Search and even Random Search. Some consider Bayesian Hyperparameter Optimization to be superior in performance and execution time to the aforementioned search methods, but there are results which show that Convolutional Neural Networks can benefit more from Random Search in terms of more optimal hyperparameters despite its longer execution time. Therefore we need to make a trade-off. Even if Bayesian Hyperparameter Optimization will perform less accurate optimization than Random Search, we will still favour it because execution time will be faster. On the other hand it is more difficult for the researcher to set up. Hyperparameter optimization is also not the main focus of this research, so it will be more beneficial to have more time to focus on the actual intents and not on additional fine-tuning. Thus Bayesian Hyperparameter Optimization might be an excellent starting point when results are as expected, otherwise we will stick to Random Search.

The process will go cyclic as follows:

- One should create a neural network using some default setting of hyperparameters found by experience or hand-tuning.
- The neural network gets trained.
- One evaluates the performance (accuracy) of the network using a validation data set. Here we know the exact accuracy minus some inevitable noise.
- We take the negative classification accuracy as an objective value which needs to be minimized. Bayesian Optimizers often perform minimization instead of maximization, hence taking the negative.
- A test run on a test data set will take place to check whether the neural network does its job.
- Bayesian Hyperparameter Optimization takes place.
- The neural network with most optimal hyperparameters gets saved to disk together with its accuracy.
- Start again with updated hyperparameters until a fixed number of iterations is reached or until convergence reaches a negligible difference. The higher the number of epochs and overall iterations, the better the result in opposition to execution time.

In the Bayesian Hyperparameter Optimization phase we will form a new model of the hyperparameter search space behind the scenes. One can use a model called the Gaussian Process [49]. Here we get an estimate of the variation of performance when one adjusts the hyperparameter values. Another possible optimization method is Random Forests [50]. The reason that Gaussian Processes are so common, is that they try to infer a posterior distribution of the loss function f which can be updated

after the computation of new hyperparameters. *f* is in fact an unknown function for which no closed form is known, nor its gradients. We try to minimize this function as:

$$\vec{x}^* = argmin_{\vec{x}} f(\vec{x}).$$

We basically ask the Bayesian optimizer to give us a new suggestion for hyper-parameters in a region of the search-space that we haven't explored yet, or hyperparameters that the Bayesian optimizer thinks will bring us most improvement. We then repeat this process a number of times until the Bayesian optimizer has built a good model of how the performance varies with different hyperparameters, so we can choose the best parameters.

More formally, in Bayesian optimization we use previous or prior observations $x_1, \ldots, x_n$ to compute an optimal posterior expectation of the loss function *f*. Note that points are considered a collection of hyperparameter values in the hyperparameter space. Next we will sample the loss *f* at a new point $\vec{x}_{t+1}$. This new point will specify which parts of the domain of *f* are the most optimal to take samples from. One repeats these steps until a certain convergence to an optimal point in the hyperparameter space occurs or when a maximum number of iterations *T* is reached.

The posterior distribution can be computed by introducing a likelihood model for the previously evaluated prior samples $x_1, \ldots, x_n$ from *f*. For Bayesian optimization we use a normal distribution with added noise for sampling. So when we have a collection samples from *f* called $f(\vec{x})$, we can form a Gaussian function around the samples, where the noise will correspond to the standard deviation of the Gaussian. In the end we have modelled the function *f* as a mixture of Gaussians.

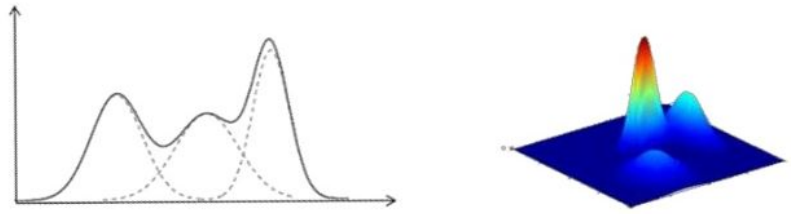

Figure 4.9: (Left) A mixture of Gaussian distributions in the univariate case. (Right) A mixture of multivariate Gaussian distributions used to model a function *f*.

In practice, we use a **Gaussian Process** for describing *f*. It will return the mean and variance of a normal distribution over the possible values of *f* at $\vec{x}$. A Gaussian Process can be seen as a generalization of the Gaussian probability distribution, but in a multivariate functional sense. We consider the GP as a multidimensional Gaussian distribution over functions instead of variables. Instead of using a mean and a variance, we now have a mean function $m(\vec{x})$ and a covariance function $k(\vec{x}, \vec{x}\,')$.
Let us say we have a set of data points $x_{1:n}$ and their loss values $f_{1:n}$. Each loss value can be described by a multivariate (normal) Gaussian distribution, to form a vector equation:

$$f_{1:n} = N(m(\vec{x}_{1:n}), K).$$

Here *K* is an *nxn* matrix, called the kernel of covariances where each element $K_{ij} = k(\vec{x}_i, \vec{x}_j)$ and $m(\vec{x}_{1:n})$ is a vector of means $[m(x_1), ..., m(x_n)]^T$.

In order to find the new best point to sample *f* from again, we need to choose a point which will maximize a certain **acquisition/utility function** $u(\vec{x})$. The acquisition function is a function of the posterior distribution over *f* and it describes to utility for each hyperparameter value. So we need to consider the values with the highest utility to compute the loss *f* from next.

In practice we gave a broad range of acquisition function to choose from and the most common one is the **Expected Improvement (EI)** function. Mathematically it is written as:

$$-EI(\vec{x}) = -E[max\{0, f(\vec{x}) - f(\vec{x}')\}].$$

Here $\vec{x}'$ is a vector which described the present optimal hyperparameters, i.e. the best point found so far. When we try to minimize this utility function, or maximize its negative, we will retrieve a point which is expected to improve *f* the best. Note that the acquisition function is simpler to optimize than the original loss *f*. It is also possible to compute its derivatives analytically, and use a gradient-based solver to maximize this function.

By using partial integration it is possible to compute the Expected Improvement (EI) as a closed-form expression.

$$\sigma(\vec{x}) = 0 \implies EI(\vec{x}) = 0$$
$$\sigma(\vec{x}) > 0 \implies EI(\vec{x}) = (\mu(\vec{x}) - f(\vec{x}'))\Phi(Z) + \sigma(\vec{x})\phi(Z) \text{ , } Z = [\mu(\vec{x}) - f(\vec{x}'))]/\sigma(\vec{x}).$$

Here $\Phi(Z)$ and $\phi(Z)$ are respectively the cumulative density function (CDF) and probability density function (PDF) of the multivariate Gaussian distribution.

We can now observe that:

- The Expected Improvement is high if the variance (uncertainty) around $\vec{x}$ is high. So when we maximize the EI, a high variance means we search in a part of *f* which has not yet been covered.
- The Expected Improvement is high is high if the posterior loss $\mu(\vec{x})$ is greater than the current best loss $f(\vec{x}')$. So when we maximize the EI, we can also sample from points which we expect to have higher utility.

In summary, this will give us the final Bayesian Hyperparameter Optimization algorithm using a Gaussian Process acquisition function. For each iteration *t* from 1 to *T,* or until convergence:

1) Using the observed values $(\vec{x}, f(\vec{x}_i))$, where *i* lies in [1, *t*], we build a probabilistic model for the loss *f* by using the Gaussian Process model. Although Random Forests is a possible alternative.

2) Next, we find a new sample point $\vec{x}_{t+1}$ which will maximize the EI, such that: $\vec{x}_{t+1} = argmax_{\vec{x}}\{EI(\vec{x})\} = argmin_{\vec{x}}\{-EI(\vec{x})\}$.
3) Finally, we will compute *f* for this new sample point $\vec{x}_{t+1}$, so we can add the new observation $\vec{y}_{t+1} = f(\vec{x}_{t+1})$ to our set of observed values.

Fortunately for us there are libraries such as scikit-optimize which have a Bayesian Optimization module, so we do not need to program all the mathematical details from scratch.

When using such a library, it is advised to consider the following guidelines:

- Like said earlier in this chapter, hyperparameters need to use a scale different from the standard uniform scale, whenever necessary. The initial learning rate and regularization strength for example need to be sampled from a logarithmic scale.
- Whenever necessary for performance improvements, one can experiment with using different Gaussian Process kernels such as the most common Matérn or White kernels for the covariance function *K*. We will not get into more mathematical details here, but for a Gaussian Process the Matérn kernel [51] is usually sufficient.
- In order to ensure that sampled hyperparameter vector components are unique, we need to add noise, in the same scale of the loss function, to the diagonal of the kernel matrix *K*. When hyperparameter values are too similar, the problem will not be well-conditioned.

4.9 Batch normalization

4.9.1 Normalizing activations

We have seen how input normalization can help gradient descent algorithms to reach convergence to a minimum much easier, since the cost function will transform from an elongated bowl to a much more spherical bowl shape. We can also apply the same idea to the activations so that the output of each hidden layer gets normalized to zero mean and unit variance of one as well. Using batch normalization [52] will yield a faster training of the weights and biases of the network. Here we normalize the dot product and bias sum *before* applying the activation function, e.g. ReLU, since in practice this order is more common, yet it is also possible to normalize the output after application of the activation function.

Given some intermediate values in the neural network, *before* application of the activation function $Z^{[l](i)}$, where we have:

$$Z^{[l](i)} = a^{[l-1]} \cdot W^{[l]} + b^{[l]}.$$

Here *l* is the hidden layer number and *i* the hidden unit number ranging from 1 to *m*. We calculate the mean and variance of the hidden units in the layer and use them for normalization of the values $Z^{[l](i)}$. After normalization we can scale the mean and increase or decrease the variance by applying a scaling and translation by certain other learnable parameters, γ and β respectively, as well to reach different

means and variances. The following formulas are applied:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} Z^{[l](i)}$$
$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(Z^{[l](i)} - \mu)^2$$
$$Z_{norm}^{[l](i)} = (Z^{[l](i)} - \mu)/\sqrt{\sigma^2 + \varepsilon}$$
$$Z'^{[l](i)} = \gamma Z_{norm}^{[l](i)} + \beta.$$

Here ε is again used for numerical stability. Later, we will use these new values $Z^{[l](i)}$ for further computations in the neural network as they get fed into the activation function.

In practice, batch normalization is applied to mini-batches of the training set. All the above calculations are applied to each mini-batch with vectorization instead of applying them to each example and further hidden unit separately. Gradient descent also gets applied to the γ and β parameters during backpropagation Instead of gradient descent, other optimization algorithms such as RMSprop, Momentum and Adam could be used as well.

Batch normalization is beneficial in the sense that after normalizing the input, one does not have to worry about the fact that the scale of the input features will be different for each component in the feature vector.

A downside of using batch normalization is that it induces regularization as an unintended side-effect. Each mini-batch will be scaled by the mean and variance computed in the current mini-batch. This will add a small noise to the Z-values in this mini-batch for each layer. After applying the activation functions, the noise will still be present and can have a regularization effect similar to dropout. These side-effects can be decreased by using larger mini-batch sizes when possible.

4.9.2 Batch normalization during testing

Since in training we use one mini-batch at a time and during testing we process one test example at a time, we need to adjust the above batch normalization scheme in the neural network in order to fit the testing scheme. In order to do this, we use an estimate of the mean and variance using exponentially weighted averages across all mini-batches. This is done for each layer separately. This exponentially weighted average is computed during training.

4.10 Summary

To finish up this chapter, in general we can list the following guidelines for improving performance:

- Start with biases of 0 and change this later to another small value such as 0.01 when needed.
- In practice, most recently He et al. [34] suggest the use of ReLU activation functions and weights sampled from a unit normal (Gaussian) distribution as:

  $$W = rand_{normal}(shape = (n_{in}, n_{out}), stddev = \sqrt{\frac{2}{n_{in}}}),$$

  where n_{in} and n_{out} are the number of input and output of nodes.

- Whenever sigmoid or tanh activation functions are used, it is best practice to use Xavier initialization.
- It can be helpful to plot the training accuracy and the cost function during training or validation. When using CNNs it is also helpful to plot the filter weights of the first convolutional layer to gain insights in what the network is learning.
- For the optimization algorithm it is advised to start with Adam or use Stochastic Gradient Descent with Nesterov Momentum.
- Use a decay scheme for the learning rate while training. This can be done with the use of an adaptive learning rate method or an external decay strategy where you can use a function such as exponential or hyperbolic learning rate decay.
- For reducing bias of the chosen optimization algorithm, one can shuffle training data randomly before starting an epoch or after performing an epoch.
- Pick an appropriate mini-batch size according to the memory capabilities of the graphics card. However, it is empirically shown that smaller batch sizes improve generalization.
- Use input and batch normalization for increased convergence speed.
- Introduce dropout or L2 regularization to resolve overfitting of the network when necessary.
- Hyperparameter tuning can be done by using Random Search or Bayesian Hyperparameter Optimization using Gaussian Processes. Grid Search should be avoided at all costs. When using Random Search, we can start using broad (coarse) hyperparameter intervals for 1 to 5 epochs and later adjust the intervals to be finer each epoch.
- For additional performance we can use different neural networks and take the average of their output. This is a technique called Ensemble Learning.

Chapter 5: Generative Adversarial Networks

5.1 Motivation

Generative Adversarial Networks [23], also abbreviated to GANs, are a form of generative models with an underlying neural network structure. GANs have the ability that they can learn to produce new data which is similar enough to given training data. When a GAN knows how to create new content, it has also learned about existing similar content. We say that the model has learned its own meaningful representation of the content in its own fictitious latent space. The use case of GANs is twofold: it can serve as a generator or as a discriminator. In fact this reflects in the implementation of GANs since they consist of two competing neural networks: a generator which generates new content and a discriminator which serves as a binary classifier to determine if the new content is valid or not. When one is interested only in the discriminator, we see that the generator serves as a mechanism to provide unsupervised training for the discriminator. The same thing applies for both networks. Training data does not have to be labeled. The GAN creates its own representation. The strongest point of the generator is that it has the ability to form an understanding in a fictitious latent space which describes the hidden structure of input data in the absence of class labels. Therefore the generator can be used for unsupervised learning where no manual labeling of the data is necessary and a lot of man hours get saved. Also due to using noise variable input, the resulting output will be more diverse and could be considered novel.

Currently GANs and their derivatives produce state of the art results for generating new content where they outperform other existing models at producing samples in some use cases such as Variational Autoencoders (VAEs) [25], Deep Boltzmann Machines [54], Deep Belief Networks (DBNs) [55], etc. They were first published in 2014 by a research group under Ian Goodfellow. Another benefit of GANs is that they are asymptotically consistent, in contrast to Variational Autoencoders (VAEs) which are not fully proven to be. The main issue with GANs however is that they are hard to train if the objective is the reach of a Nash equilibrium instead of the standard optimizing of an objective function.

5.2 High level description of GANs

The discriminative model or discriminator, like the classifiers discussed in previous chapters, learns essentially a nonlinear function which maps input data x to a desired output class y. Expressed in the language of probability theory this means that a discriminator tries to learn the distribution $P(y|x)$, i.e. given input data x, we determine the probability of y.

The generative model or generator will learn a joint probability $P(x, y)$, where x and y are the input data and the class label. By applying the chain rule of probability we can see that:

$$P(x, y) = P(y \mid x)P(x).$$

And so the generator can actually be converted into a classifier. Yet, for our purposes we use it to generate novel data samples (*x*, *y*). The strongest point of the generator is that it has the ability to form an understanding in a fictitious latent space which describes the hidden structure of input data in the absence of class labels. Therefore the generator can be used for unsupervised learning where no manual labeling of the data is necessary and a lot of man hours get saved.

As said before, a GAN is a combination of two competing neural networks trained simultaneously; a generator and a discriminator. The generator is a model which uses input noise to generate novel samples. The discriminator takes the newly generated samples and samples from the training set and then tries to label samples as real, i.e. coming from the training data, or fake, i.e. generated by the generator. Both networks will play an adversarial game. The generator will learn to create iteratively more realistic data, while the discriminator improves its ability to distinguish the real from the fake data. The adversarial game is expected to stop at a point where generated samples are similar to real samples coming from the training set.

In a GAN, we can backpropagate gradient information from the discriminator network to the generator. This way the generator will know how to adjust its weights so it can create data which can deceive the discriminator. This indicates the main difference with a reinforcement learning setting where the discriminator is a supervisor which will send a reward to the generator when it generates accurate novel data. GANs prevent the need for a direct supervisor and instead use the shared gradient information as a tool to introduce an unsupervised setting.

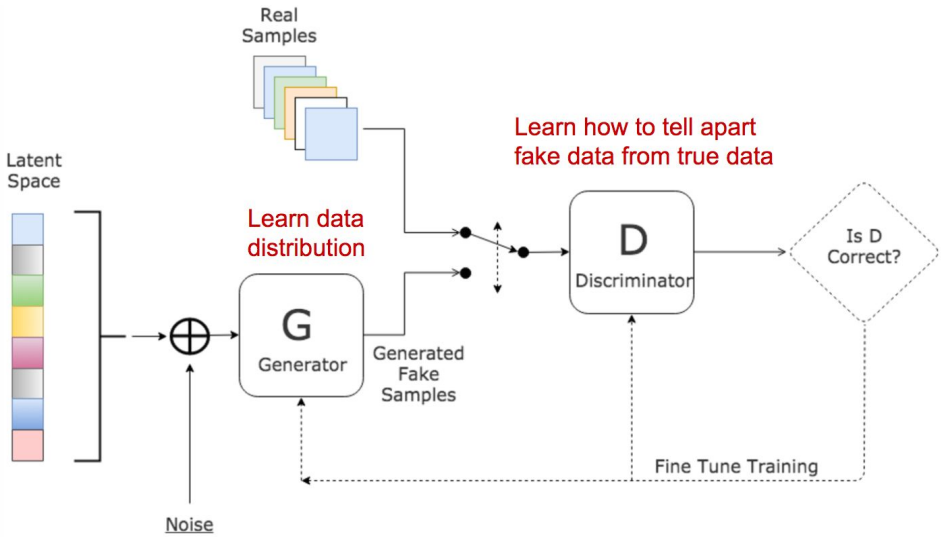

Figure 5.1: The high level topology of a Generative Adversarial Network (GAN) consisting of a discriminative and generative model. The objectives of each model are shown in red text above.

More formally, the discriminative model or discriminator, like the classifiers discussed in previous chapters, learns essentially a nonlinear function which maps input data *x* to a desired output class *y*. Expressed in the language of probability theory this means that a discriminator tries to learn a distribution $P(y \mid x)$, i.e. given input data, we determine the probability of the output class.

The generative model or generator will learn a joint probability where *x* and *y* are the input data and the class labels. By applying the chain rule of probability we can see that:

$$P(x, y) = P(y \mid x)P(x).$$

And so the generator can actually be converted into a classifier. Yet for our purposes we use it to generate novel data samples *x*.

The real input data coming from a data set can be thought of as having an underlying data distribution for the whole set of data. Samples, i.e. training and test examples, are samples taken from this distribution. The generator model will try to form a distribution which will try to approximate the real data distribution so it can produce valid examples.

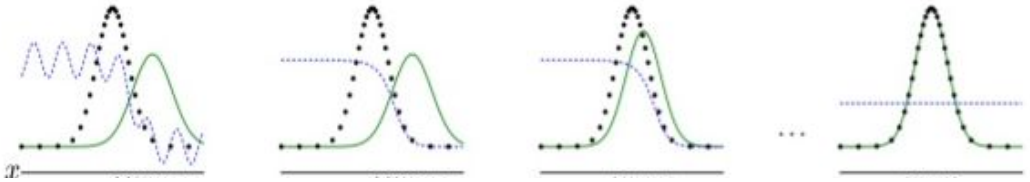

Figure 5.2: The black dotted curve is the real data distribution. The generator which tries to approximate the real curve is shown in green.

5.3 Mathematical description of GANs

In order to fully understand vanilla GANs and its shortcomings, one needs to form a mathematical description which provides more detail why GANs work, why they can be tricky to train and what can be done to improve them. [23]

We call the discriminator *D* and the generator *G*. The variable noise input is named by *z* and delivers diversity to the resulting output. We call p_z the data distribution for this noise value *z*. Most of the time we want uniform noise, so p_z is considered a uniform distribution unless considered otherwise. The real data distribution which produces real samples *x* is given by p_r. The data distribution of the generator which tries to approximate the real one is given by p_g.

When trying to improve the discriminator's decisions regarding the real data distribution we want to maximize the expectancy

$$\mathbb{E}_{x \sim p_r(x)}[logD(x)].$$

Here the logarithm has base 10, making it a Briggsian logarithm. This means we want to maximize its chance to correctly classify a sample coming from the real data distribution.

When given a sample $G(z)$ generated by the generator, where z is noise coming from the (uniform) distribution p_z, the discriminator will output a probability in [0, 1] given by $D(G(z))$. Therefore on fake data the discriminator will try to **maximize** the expectancy

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

Here we use 1 minus $D(G(z))$ because we want to minimize the chance that the discriminator will output a high probability for the given fake data to be real. In other words: we want the discriminator to maximize its chance to classify fake data from the generator as fake, hence improving the classification skills of the discriminator.

The generator is trying to fool the discriminator in trying to classify its data with a high probability. Therefore by means of minimization rather than maximization, the generator tries to **minimize**

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

This means the generator has actually the opposite goal of the discriminator.

When the discriminator and generator are taking turns, they form a two-player MinMax game where we perform the following optimization:

$$\min_G \max_D L(D, G) = \mathbb{E}_{x\sim p_r(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))]$$
$$= \mathbb{E}_{x\sim p_r(x)}[\log D(x)] + \mathbb{E}_{x\sim p_g(x)}[\log(1 - D(x)].$$

This essentially is the combination of the objectives of a generator and discriminator as described earlier.

It can be shown that if a generator is well-trained and optimal, the data distribution of the generator p_g approaches the real data distribution p_r. If $p_g = p_r$ it can be shown that the optimal value for the discriminator becomes 0.5. In other words: the discriminator has a fifty-fifty chance of classifying fake data as real data, i.e. it cannot tell if generated data is fake or not. We will omit the proof, but the reader can find it in the original paper on GANs by Ian Goodfellow et. al. In this paper you can also find that the global minimum that the generator tries to achieve is - log 4.

To measure if p_g and p_r are similar enough, we need to find a metric for measuring the similarity between these two distributions. In GANs we use two metrics:

1) **Kullback-Leibler divergence [56]:** or in short KL divergence, measures how a given probabilistic distribution p diverges from another distribution q. Divergency is the opposite of similarity, so the more a distribution diverges, the less similar they are. Here we want a minimal divergence of zero, which is essentially a similarity of 1, meaning 100% similarity so $p = q$. The KL divergence is defined as:

$$D_{KL}(p\|q) = \int_x p(x) \log \frac{p(x)}{q(x)} dx.$$

A downside of this metric is that when *p* is likely to be 0 and *q* is never 0, the effect of *q* will be left unconsidered. This is not what we want, since the effect of both distributions is key.

2) **Jensen-Shannon Divergence [57]:** or in short JS divergence is a better alternative since it is symmetric. It is formed using the average of two KL divergences. The arguments are chosen in such a way that the effect of both *p* and *q* are taken into account by comparing each distribution to the average distribution. As a downside it takes double the time to measure similarity.

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

We will not go excessively into mathematical detail, but when measuring similarity between p_g and p_r, we see that when a discriminator is optimal, the loss function of the entire network will encompass the similarity between real and fake data. An optimal generator will therefore lead to its global minimum of - log 4 and will create fake data which will be considered real enough by an optimal discriminator.

5.4 Complications of GANs

Training a standard GAN is hard and it is well known to have slow convergence speed and can suffer from instabilities. Yet when one manages to overcome the training difficulties, one can generate realistic content with a GAN. Typically we are using gradient descent based optimization algorithms which try to find the minimum of a cost function, rather than to find the Nash equilibrium of a two-player adversarial game, and therefore convergence to such a state might not be possible. The most common problems we will face are given below.

5.4.1 Problematic reach of a Nash equilibrium

In the GAN training setting we use two competing neural networks in an adversarial game. Both networks are trained concurrently in order to converge to a Nash equilibrium. The cost of each model will be updated independent from the other model in the game. The fact that the gradient will be updated for each model at the same time will not necessarily imply convergence. This was first documented by Salimans et al. in 2016 [58], who essentially blamed the gradient descent based approach used in GANs.

As an example to see this difficulty in reaching a Nash equilibrium, we can define a small game of two players where the first player tries to minimize $f_1(x) = xy$ and another player tries to minimize $f_2(y) = -xy$. The example serves as a simplification of the adversarial game in GANs. This way the first player has to update *x*, while the second player is updating *y*. We can compute the following partial derivatives, i.e. the simpler non-vector form of the gradient:

$$\frac{\partial f_1}{\partial x} = y$$
$$\frac{\partial f_2}{\partial y} = -x.$$

As a concurrent update step or iteration for each variable we use:

$$x_{new} = x_{old} - \alpha \frac{\partial f_1(x)}{\partial x}$$
$$y_{new} = y_{old} - \alpha \frac{\partial f_2(y)}{\partial y}.$$

Which can be simplified as:

$$x_{new} = x_{old} - \alpha y$$
$$y_{new} = y_{old} + \alpha x.$$

Here α is the learning rate. We can observer that once *x* and *y* will have a different sign, each next update will cause bigger oscillations. The system will lose its stability after each iteration and eventually will reach irreparable instability. Reaching a Nash equilibrium is not possible in this situation.

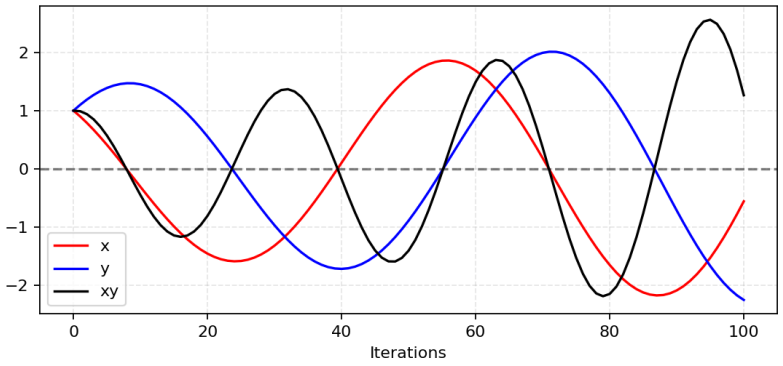

Figure 5.3: A plot of the above simplified example with a learning rate α = 0.1. Once *x* and *y* will have a different sign, each next update will cause bigger oscillations and instability as seen in the black curve of the function *xy*. [59]

5.4.2 Improper stop criterium

A decent stop criterium for the GAN would be the moment the models reach a Nash equilibrium. Yet, when using an optimization method based on gradient descent, reaching this state is impossible since the loss for both of the models in the GAN will oscillate and cause an increasing instability. In practice, there is no stop criterium. GANs do not have an objective function which informs the researcher about improvements during training. The only way one can stop the training phase, is when the generated images are visually pleasing enough and when no perceived improvement occurs. [58, 59]

5.4.3 Mode Collapse

When training a GAN, the most common problem can be that the generator will reach a state where it can only create the same samples or highly similar ones. In other words: the samples generated by the generator are not diverse. It violates the essence of GANs; i.e. high diversity in generated data. We call this phenomena Mode Collapse [58, 59]. The generator might still be able to deceive the discriminator, but it essentially loses its creativity. This is due to the fact that the generator can get stuck in a space with low diversity when trying to optimize its data distribution to fit the real one. [58, 59]

5.4.4 The vanishing gradient problem

When we reach a perfect discriminator, then it will classify real examples of p_r as real ($D(x) = 1$) and fake examples coming from the distribution p_g as fake ($D(x) = 0$). In such a state the loss function L will reach 0. Therefore we reach a point where there is no gradient anymore which can update the loss. Thus, the better the discriminator gets, the more we have to deal with the vanishing gradient problem. The deeper the neural network, the bigger the vanishing effect of the gradient. [58, 59]

This imposes the following dilemma when training a GAN:

- A perfect discriminator will cause a vanishing gradient, which slows down or stops learning.
- On the other hand, a malfunctioning discriminator will cause a generator to receive inaccurate feedback on its creations.

5.4.5 Low dimensional supports

In the work of Arjovsky and Bottou from 2017 [60] we can find a mathematical discussion on the problem where supports of p_g and p_r lie on different low dimensional manifolds and therefore will introduce instability during training. Recall that a support is the subset of a domain of a real function f which contains only elements which are not mapped to 0 by this function. The support of a real function $f : X \mapsto \mathbb{R}$ is defined as:

$$supp(f) = \{x \in X |\ f(x) \neq 0\}.$$

From manifold learning [61] we know that although the dimensionality of real-world data given by a distribution p_r is only imaginary high. Manifold learning assumes that the data of interest is concentrated on an embedded nonlinear manifold within a higher-dimensional space. When in practice we generate novel real world images, there are a lot of constraints which are implicitly learned: e.g. generated cars can only have four wheels, humans should have two legs and two arms, etc. Therefore the constraints will also put constraints in the possibility that data has a higher dimensional form. Because p_r and p_g are situated in lower dimensional manifolds, the probability that both distributions are disjoint is legitimate. If both distributions own disjoint supports, there can be a discriminator model which can separate real from fake data at full accuracy. Proofs can be found in the mentioned paper [60].

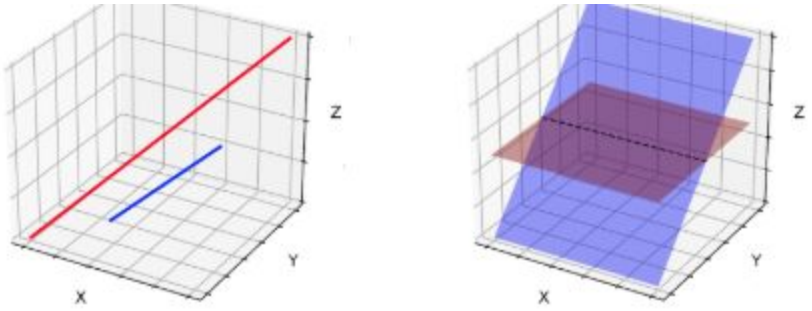

Figure 5.4: The probability that two low-dimensional manifolds will overlap in a high-dimensional space is low. (Left) Two lines, or one-dimensional manifolds, in a three-dimensional space with a low intersection probability. (Right) Two surfaces, or two-dimensional manifolds, in a three-dimensional space with a high intersection probability.

5.5 Improved techniques for training GANs

After the homonymous paper by Salimans et al. from 2016 [58], in this section we list a number of techniques for overcoming the problems faced by training and eventually using GANs. Improved convergence is a first goal, so we will list four techniques to achieve this. The last techniques that will be discussed can help us to overcome the problem of disjoint p_g and p_r distributions in high dimensional spaces.

5.5.1 Feature matching

Suppose we have a function $f(x)$, which can be an arbitrary function that calculates a certain statistic of the feature vector x, e.g. mean, median or variance. We will define a novel loss function where one can optimize the discriminator accordingly to the case when the generated output's statistics $f(G(z))$ match those of the real data. For simplicity, we wrote vectors without the typical arrow notation. The loss function can be formulated as a squared L2 distance (Euclidean norm):

$$\|\mathbb{E}_{x\sim p_r} f(x) - \mathbb{E}_{z\sim p_z(z)} f(G(z))\|_2^2.$$

5.5.2 History averaging

For both the generator and discriminator we can add to the loss function the following term:

$$\|\Theta - \frac{1}{t}\sum_{i=1}^{t}\Theta_i\|^2.$$

Here Θ is a vector of the current weights or biases of the model, while Θ_i represents the parameter values at a previous training batch i. The cost is the distance between the current parameters Θ and the

average of the parameters over the last *t* batches. Essentially, this is the addition of a penalty term which punishes parameters which largely differ from their historical average values. This can help the GAN to reach equilibria that standard gradient descent could not find.

5.5.3 One-sided label smoothing

As proposed by Salimans et al. [58], we can use a technique called one-sided label smoothing as a regularization method. Instead of using 1-valued labels for real data, we use less harsh values such as 0.9 when feeding the discriminator. As advised by Ian Goodfellow [62], it is important to not smooth the labels for the fake samples as this would impose mathematical problems which we will not discuss further. The reason for success can be found in the fact that it does not push the model to choose a false label for the training data. It will only try to reduce confidence in the true class.

5.5.4 Virtual batch normalization

As we will see in the next chapter about Deep Convolutional Generative Adversarial Networks (DCGANs), we will need batch normalization [52, 58] to improve optimization of the models. There are however a few side effects in doing so. When using a different mini-batch to compute the mean and variance on each training step, we get a fluctuation for these values. When the available GPU memory is low and mini-batch sizes are chosen to be small accordingly, these fluctuations will increase and they will have a bigger influence on the generated data than the noise vector will have.

In virtual batch normalization [58] each sample will be normalized with respect to samples of a chosen reference batch. The reference batch is chosen at the beginning of the training phase and will stay fixed during training. The purpose of virtual batch normalization is to reduce the dependence of each sample on the other samples in the mini-batch. The problem is that virtual batch normalization requires a fair amount of computation time. The reason is that we need to forward passing on two mini-batches of data instead of one. Therefore virtual batch normalization will only be applied on the generator.

5.5.5 Noise addition

When we have an high dimensional feature space, we see that the distributions of real and generated data (p_r and p_g) are disjoint. This results in the vanishing gradient problem. We can try to solve this by adding continuous noise to the discriminator inputs, i.e. both real and fake data. When doing so, the distributions will be more spread out and will have a higher chance that p_r and p_g will overlap. The mathematical proof can be found in [60].

5.5.6 Wasserstein distance as a similarity metric

When using a standard GAN, the loss function will measure the Jensen-Shannon divergence between the p_r and p_g data distributions. The problem with this metric is that it is undefined when p_r and p_g are disjoint distributions. We can solve this by using a Wasserstein distance [39] instead of the Jensen-Shannon divergence. The Wasserstein distance function measures a distance between two probabilities. Intuitively, each distribution can be viewed as a unit amount of clay in a metric space *M*, the metric is the minimum cost of turning one lump of clay into the other. The cost is given by the amount of clay moved times the moved distance. This Wasserstein distance is therefore also called the Earth Mover's distance.

As a first example we consider the discrete case where we have two distributions *P* and *Q* each consisting of four lumps of clay such that both distributions are the sets of its lumps P_i and Q_i with *i* an integer index from 1 to 4. Each distribution divides 10 unit pieces over its entire domain, so for example:

$$P = \{P_1 = 3, P_2 = 2, P_3 = 1, P_4 = 4\}$$
$$Q = \{Q_1 = 1, Q_2 = 2, Q_3 = 4, Q_4 = 3\}.$$

Then we need to find a scheme to change the look from *P* to *Q* so that they are indistinguishable. We can do this by using the following steps:

- Move 2 pieces from P_1 to P_2, such that:
  $$P_1 = Q_1.$$
- Move 2 pieces from P_2 to P_3, such that:
  $$P_2 = Q_2.$$
- Move 1 piece from P_4 to P_3, such that:
  $$P_3 = Q_3,\ P_4 = Q_4.$$

The total amount of moved pieces or the Earth Mover's distance is the sum of each step, and thus 5.

In the case of a continuous probability domain, the Wasserstein distance between the two data distributions will be:

$$W(p_r, p_g) = \inf_{\gamma \sim \Pi(p_r, p_g)} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|].$$

Here $\Pi(p_r, p_g)$ is the set of each possible joint probability distribution between the two distributions p_r and p_g. Imagine one such joint probability distribution γ. This distribution will describe one way to move the earth, but now in a continuous probability space instead of a discrete one. When we consider two points *x* and *y* which indicate two possible states of a clay model, $\gamma(x, y)$ will express the percentage of clay which should be transferred from *x* to *y* in order to make *x* resemble *y*. Or another way of seeing this is that the total amount of data moved from *x* to *y* is $\gamma(x, y)$. The transfer distance is $\|x - y\|$ and the cost is

$\gamma(x, y) \cdot \| x - y \|$. The expected cost averaged across each (x, y) pair can be expressed as:

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

As a last step we consider the minimal cost of all the movement solutions as the Wasserstein distance. The "inf" in the formula stands for infimum and is essentially the greatest lower bound, coming from mathematical analysis. The infimum indicates that we only consider the minimal cost.

5.5.7 Incorporating Wasserstein distance into the loss function

The above formulation is unmanageable as we would have to consider every possible joint distribution in the set $\Pi(p_r, p_g)$. The authors of the Wasserstein GAN paper [39] proposed a clever adapted formula based on the Kantorovich-Rubinstein duality:

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

Here "sup" is the supremum, which is the opposite of the infimum, i.e. we want to measure the least upper bound and thus the maximum value. We will omit the correctness proof of this transformation. The new function f in this dual form is K-Lipschitz continuous [39], which means there exists a real Lipschitz constant $K \geq 0$ for f such that for each two real numbers x_1 and x_2:

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

Let us assume the function f of the dual form of the Wasserstein distance is coming from a set of K-Lipschitz continuous functions $\{f_w\}_{w\in W}$, where W is a set of indexes for these functions.

In a WGAN the discriminator will learn an index w (which can be considered a weight) to find an optimal f_w and the discriminator loss function measures the Wasserstein distance between the data distributions p_r and p_g:

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

This means that the discriminator model is not directly separating the real data from the fake. Now it will learn a K-Lipschitz continuous function, which is used in the computation of the Wasserstein distance. When minimizing the loss function, the Wasserstein distance will decrease, which causes the generator to generate more real data as its distribution p_g approaches the real data distribution p_r.

The authors of the WGAN paper introduce a hack called "weight clipping", where they try to maintain K-Lipschitz continuity of f_w during training. After each update step, the weights w are clamped in a small

window, e.g. [-0.01, 0.01]. Therefore the weight space W will be smaller and f_w will now have an upper and lower bound and thus K-Lipschitz continuity is preserved.

In short, the WGAN algorithm makes the following adaptations to the GAN algorithm:

- After each gradient update for the discriminator, weights will be clipped in a small window [-c, c].
- The logarithmic loss function of the standard GAN is replaced by the one using the Wasserstein distance.
- The discriminator will estimate the Wasserstein distance between the real and fake data distributions.
- The RMSProp optimization algorithm is recommended by the authors of the WGAN paper, instead of using Adam which they claim to introduce instability. These results are only experimental and no theoretical proof has been given.

Using the Wasserstein metric instead of KL or JS divergence can improve stability to some extent. When two data distributions are situated on non-intersecting lower dimensional manifolds, the Wasserstein distance can still be used to smoothly measure the distance between the distributions.

But we can conclude that using a Wasserstein GAN is not the most elegant solution for stabilizing GANs, as the authors of the WGAN paper wrote that “weight clipping is a terrible way to enforce a Lipschitz constraint” [39]. WGANs still have some unstable training, slow convergence when using a large clipping window and the vanishing gradient problem when using a small clipping window. Gulrajani et al. [63] proposed a replacement in 2017 of weight clipping by the introduction of a gradient penalty. Another improvement is the Least Squares GAN [64], which will be discussed in the following section.

5.5.8 Least Squares GAN (LSGAN)

Although the Wasserstein GAN (WGAN) proves itself to be a worthy modification of the original GAN by replacing the Jensen-Shannon divergence by the Wasserstein distance, the theory behind it is quite mathematical and requires hacks such as weight clipping. Although its implementation is more straightforward than the theory, the training process is slow due to slow convergence speed when being compared to the original GAN. Only the stability and thus the network performance receives improvement, training speed on the other hand grows worse.

Least Squares GAN (LSGAN) [64] tries to overcome these problems as it is a faster and more intuitive alternative for the WGAN as it tries to embrace mathematical simplicity. Therefore adjusting an original GAN to an LSGAN will only require a few changes. The idea introduced by LSGAN is to use a loss function which will deliver a smooth and non-saturating gradient for the discriminator model. This will cause the discriminator to push the generated data distribution p_g towards the real data distribution p_r and therefore the generator can generate novel data with high realism.

In a vanilla GAN on the other hand, the discriminator utilizes a logarithmic loss function. In the sample space there is a decision boundary which separates real from fake data. When using a logarithmic loss function in the generator, the distance of a sample *x* to this decision boundary is easily ignored. This induces the problem where sampled points *x* that live far from the decision boundary will not get penalized for doing so. The logarithmic loss function does not pay attention to the distance and instead only considers the sign and therefore when samples lie further from the decision boundary, their gradients will vanish. This discriminator gradient is needed the generator. When this gradient has vanished to zero, the generator will not have enough information to learn its data distribution to resemble that of the real data.

When using an LSGAN, the logarithmic loss in the discriminator is replaced by an L2-based loss. Here we will penalize samples which lie from a far distance from the decision boundary, given they are at the correct side. The penalization is proportional to the regressive distance and will generate more gradients to update the generator, because there are less vanished gradients. The generator can use these gradients to improve its learning of the real data distribution.

When performing the optimization, the only way to minimize the L2 loss of the discriminator is to have the generator create samples which will lie close to the decision boundary, as this penalization is brought into the objective of the discriminator. When the generator generates samples which lie close to the decision boundary, it means that these novel samples are hard to distinguish between real and fake. Therefore the generator will learn to improve its data distribution p_g to match p_r better. Summarized, the objective function of an LSGAN is given by:

$$min_D\ V_{LSGAN}(D) = \frac{1}{2}\mathbb{E}_{\vec{x}\sim p_r(\vec{x})}[(D(\vec{x}) - b)^2] + \frac{1}{2}\mathbb{E}_{\vec{z}\sim p_g(\vec{z})}[(D(G(\vec{z})) - a)^2]$$
$$min_G\ V_{LSGAN}(G) = \frac{1}{2}\mathbb{E}_{\vec{z}\sim p_g(\vec{z})}[(D(G(\vec{z})) - c)^2].$$

The constant *b* is the output value for the real data. This value can be 1 or 0.9 when employing one-sided label smoothing. Constant *a* is chosen to be 0 for symbolizing the fake data. Because we want the generator to deceive the discriminator, we choose *c* = 1. Other values are still possible, but these are the most common ones. This resolves to:

$$min_D\ V_{LSGAN}(D) = \frac{1}{2}\mathbb{E}_{\vec{x}\sim p_r(\vec{x})}[(D(\vec{x})-1)^2] + \frac{1}{2}\mathbb{E}_{\vec{z}\sim p_g(\vec{z})}[(D(G(\vec{z})))^2]$$
$$min_G\ V_{LSGAN}(G) = \frac{1}{2}\mathbb{E}_{\vec{z}\sim p_g(\vec{z})}[(D(G(\vec{z}))-1)^2].$$

Summarized, the only modifications to the original GAN are removal of the logarithmic loss function and replacement with an L2-loss function.

Chapter 6: Deep Convolutional Generative Adversarial Networks (DCGANs)

6.1 Motivation

Instead of solely using a standard GAN or WGAN, we can combine techniques learned from CNNs in a supervised learning setting to the unsupervised learning of GANs. This will produce a new branch of CNNs which is called Deep Convolutional Generative Adversarial Networks (DCGANs) [65]. It will transfer the benefits of CNNs compared to vanilla neural networks over to the field of GANs. Although DCGANs were initially developed for learning to generate 2D images, the techniques from this chapter can be used in a 3D setting for voxel data as well. As mentioned earlier, the idea of convolution only makes sense on spatial data in contrast to other data such as text.

As we have discussed in the previous chapter, GANs turn out to be unstable while training, which causes the generator to produce subpar results. We therefore mentioned a number of improvements which can be made to improve stability, reduce noisiness and in particular we discussed the use of a Wasserstein distance metric. We can combine these improvements with the DCGAN architecture for increased performance. In the original DCGAN paper from 2015 [65], the authors propose a number of constraints on the architecture of GANs combined with convolution and deconvolution. The learned convolutional filters can be used to draw new images. Another interesting feature is that the generators have the ability to perform vector arithmetic operations such as addition or subtraction in the latent space to combine generated images or to manipulate them.

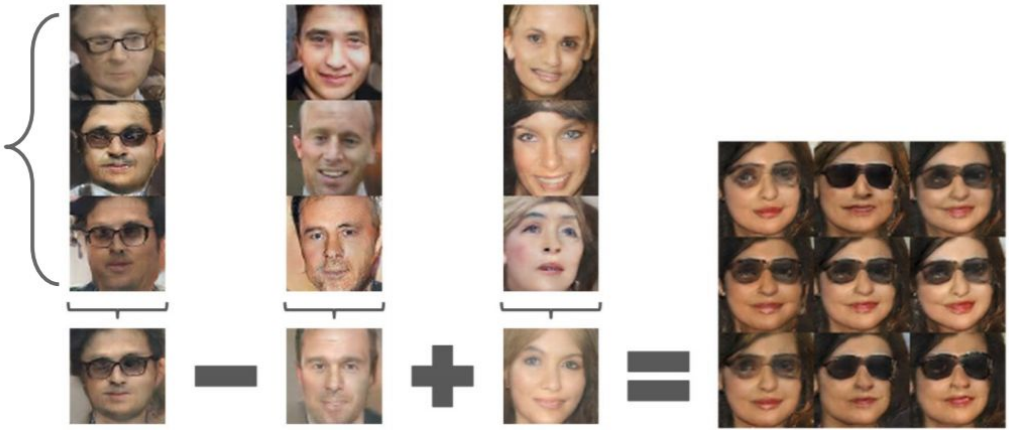

Figure 6.1: We take samples from the model where men have glasses, no glasses and where women have no glasses. From these samples we take the average *z* vectors in the latent space and perform arithmetic operations on them in this latent space. By subtracting a male face from the glasses, we only get the glasses. Later we add the glasses to the female face and thus can generate women with glasses.

As described by the authors, DCGANs were the first GAN model which has learned to generate high resolution images via one-shot learning. The generated images have a higher quality than the ones coming from a regular GAN. In this paper of DCGANs it is also shown that the latent space of GANs in general can be used to perform arithmetic as described above.

In short we can conclude that DCGANs improve the training stability in comparison to vanilla GANs. Yet there are still some instabilities; experiments show that for longer training times DCGANs can have oscillating filters which do not converge. A combination of DCGANs and the Wasserstein distance of the WGAN can help to battle these inconveniences.

6.2 DCGAN Architecture

Simply using standard CNN architectures in our DCGAN model will not help, since it is shown in the original DCGAN paper [65] that GANs are difficult to scale using standard CNNs. We therefore introduce a number of adjustments to standard CNN architectures and changes to GANs as well to form the DCGAN.
These changes can be listed as follows:

- We replace the max pooling layers with strided convolutions for the discriminator network, while using fractional strided transposed convolutions instead of integer strides for the generator network.
- Deeper architectures should employ fully-connected hidden layers.
- We employ batch normalization for the discriminator and apply this to all layers but the input layer.
- For the generator we use batch normalization on all layers but the output layer.
- The discriminator will use the Leaky ReLU activation function for each layer with a slope of e.g. 0.2.
- The generator will just use the standard ReLU activation function for each layer but the output layer, which instead will have a tanh activation function.
- No maxout activation like in the original GAN paper is used.

For the **discriminator** we use a CNN for image classification between real and fake image data. The discriminator of a DCGAN has 4 convolutional layers, but for faster training (yet slightly decreased quality) using 3 such layers is also an option. Each of those layers computes a convolution after which batch normalization and Leaky ReLU are performed. This will improve speed and accuracy. Then we use flattening of the output of the last layer. At the very end a sigmoid activation function is used to get a number between 0 and 1 for classification as real or fake data.

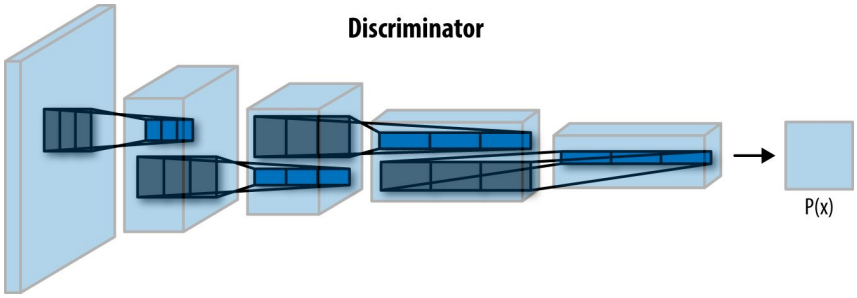

Figure 6.2: The discriminator of a DCGAN performs 4 convolutions, although more or less convolutional layers can be used whenever necessary.

The **generator** on the other hand has 4 deconvolutional layers. First we generate a random 100-dimensional uniform noise vector *Z* to serve as an input. We then project and reshape this noise vector to a small spatial extruded volume of 4x4x1024. The deconvolutional layers are used to transform this high level input into a 2D image (e.g. 64 x 64 pixels). After each deconvolutional layer, we use batch normalization and ReLU. After all the deconvolutions, we use the tanh function at the end.

Note that deconvolution is actually not the right term. In practice, we use 4 fractionally-strided convolutional layers, implemented as a transposed convolutional layer. The necessity for transposed convolutions comes from the wish to have a transformation going in the opposite direction of the standard convolution. Yet, the term deconvolution is used to indicate the opposite process of convolution. Instead of using convolution to transform a 2D image into numerical data, we now use fractionally-strided convolutions to transform numerical data into 2D image data. In fact this is identical to the discriminator, just in the opposite direction.

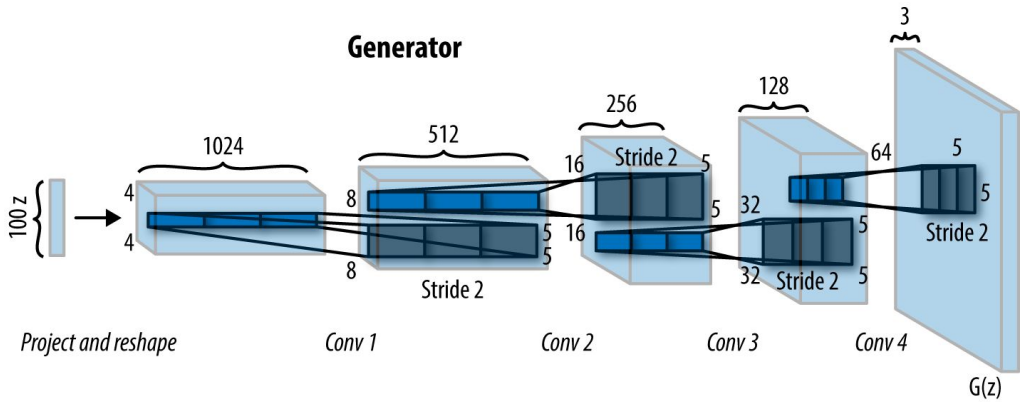

Figure 6.3: The generator performs 4 deconvolutions, i.e. fractionally-strided convolutions after projection and reshaping of the 100D noise vector *Z*. Note that we only use convolutional layers and no fully-connected layers or pooling layers.

The reason for using batch normalization after each convolutional or deconvolutional layer is the increased stability. The input of each layer will get normalized so each unit has zero mean and unit variance. The use of batch normalization proves itself worthy in preventing generator models from mode collapsing. It is shown that when batch normalization is performed on every layer, it would

ironically introduce instability. This can be avoided by performing batch normalization on every layer except for the discriminator input layer and the generator output layer.

As for **hyperparameters** used in this network we consider the following [66]:

- Use Stochastic Gradient Descent for the discriminator and Adam for the generator. Other than this using the Adam optimizer for both models can be useful as well.
- Use a mini-batch size of 128 as a starting point. This will depend on the available GPU memory. Each mini-batch contains either real or fake (generated) data.
- A better loss function than the standard logarithmic one is one which uses the Wasserstein distance or for faster convergence the L2 loss function used in LSGANs.
- Instead of sampling noise from a uniform distribution, it is better practice to sample noise from a zero-centered Gaussian (normal) distribution with a standard deviation of e.g. 0.02.
- The suggested learning rate is 0.001, but lower learning rates of e.g. 0.0002 are also documented.
- Practical results show that using a momentum term $\beta 1$ of value 0.5 yields better stability and less training oscillation than using a suggested value of 0.9.

The training process will be run in a session mini-batch per mini-batch. Every 400 or so mini-batches we can show current progress by showing the generated images and printing the loss of both the discriminator and generator models. Depending on the system specifications of the machine, training can take up to more than an hour for achieving decent results.

Instead of optimizing for one **loss function**, we can define two loss functions:

- The sum of discriminator loss for real data and discriminator loss for fake data.
- Generator loss.

For the loss function of the discriminator for real data, we compare the output of the discriminator to labels of value 1 (true). Yet, one-sided label smoothing proves to impose better results, so we compare to labels of value 0.9 as true. For the loss function of the discriminator for fake images, we compare the output of the discriminator to labels of value 0 (false). The sum of both losses needs to be minimized. We can use an optimizer such as Stochastic Gradient Descent or the Adam optimizer.

For the generator loss, the same procedure as in the last step is employed, but instead of comparing the output to zeros, we compare it to ones. The reason for this is that the generator wants to deceive the discriminator. The Adam optimizer will minimize this loss.

Chapter 7: 3D Generative Adversarial Networks (3D-GANs)

7.1 Motivation

Since the rise of DCGANs for near-perfect image generation at higher resolutions, the MIT Computer Science and Artificial Intelligence Laboratory (MIT CSAIL) introduced the 3D-GAN [67] to expand the same idea from 2D to 3D for near-perfect voxel model generation by the use of volumetric convolutions. As we know from GANs, their nature enforces data generation that has high variation, while still maintaining underlying realism. The high variational character reflects that the model is capable of being creative enough to go beyond memorization. It is essentially using different parts from the given data set without explicitly borrow exact parts and synthesizes novel content from there. Realism implies that the model has a sense for necessary detail. The higher the chosen resolution, the more detail can be observed.

There are a couple of advantages when one synthesizes 3D objects with a GAN:

- GANs are essentially a form of unsupervised learning, which does not require labeled data.
- 3D objects can be created as samples from a lower dimensional probabilistic latent space, e.g. a 200 dimensional vector from a Gaussian distribution or from a uniform distribution.
- As a possible extension we can couple the 3D-GAN with a trained Variational Autoencoder (VAE) [25], such that a 2D image can be encoded to serve as the high dimensional noise vector *z*. This way one can reconstruct a 3D object from a given 2D image.
- Next to generating novel content by the generator, one can also use the discriminator for shape classification in 3D object recognition tasks. The discriminator uses an unsupervisedly learned 3D shape descriptor.
- 3D-GANs have also empirically proven to work well when the amount of training data is limited.

When using the noise vectors in the latent space, we can perform arithmetic operations on them, as described earlier when we discussed DCGANs, to create novel objects. These operations can be addition and subtraction or even interpolation between two given objects from the same or even different classes, e.g. interpolation between a boat and an airplane. By training a generator model and later using it for testing or in an application, we can feed the generator a noise vector created by performing arithmetic operations on given noise vectors from which we know the the actual model in the voxel space. This is certainly not a costly operation and is highly intuitive compared to equivalent operations performed in the actual voxel space. Some dimensions of the latent space vectors bear semantic properties such as the width of the actual model or its thickness.

Figure 7.1: Creating novel 3D objects by performing arithmetic operations between latent space vectors in the same vein as in DCGAN vector arithmetic.

7.2 Architecture

The 3D-GAN is in its essence not much more than a 3D analogon of the DCGAN. As discussed in the DCGAN chapter, we do not use dense (fully connected) layers nor pooling layers in this model and the 3D-GAN is therefore fully convolutional.

A generator network *G* is used to transform a noise vector *z* sampled from a 200-dimensional latent space into a voxel-model which can have an enclosing grid space of 64x64x64 voxels. It upsamples feature maps to consistently lower channeled feature maps. The generator network can be constructed by the use of 5 fully deconvolutional layers with kernels of size 4x4x4 and strides of size 2x2x2. Each layer is followed by a batch normalization layer and a ReLU layer. At the end a sigmoid layer is used. We call the resulting generated 3D model *G*(*z*). Note that the use of batch normalization eliminates the need for additional biases, since they are already parameters in the batch normalization layer.

A discriminator network *D* is used to determine if the given 3D model is real or generated. The resulting output is a value *D*(*x*) which is a real value in [0, 1] indicating the percentage the discriminator is sure if the data is real, e.g. a 80% confidence in realness. The layers in the discriminator are built opposite to the generator, by using 5 fully convolutional (downsampling) layers and Leaky ReLU layers instead of regular ReLU.

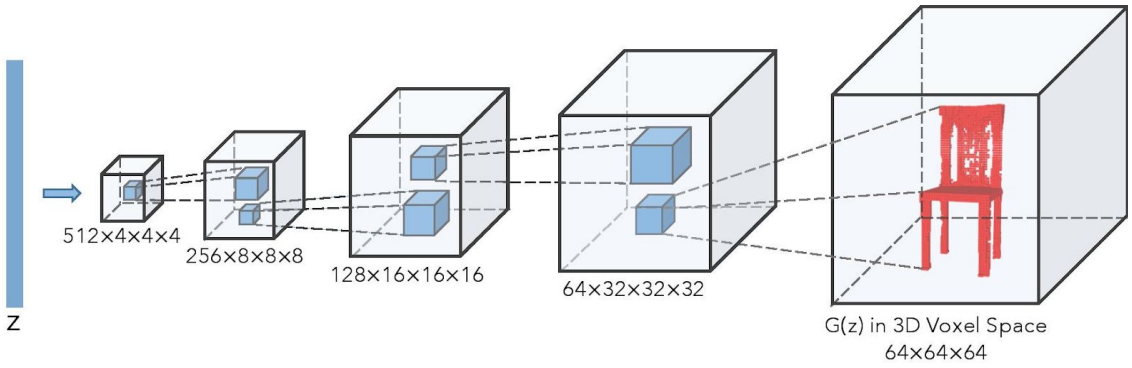

Figure 7.2: Graphical representation of the generator network. We first use 512 4x4x4 upsampling filters, followed by: 256 8x8x8 filters, 128 16x16x16 filters and 64 32x32x32 filters and a resulting 3D

voxel model *G*(*z*). Each filter operates on a feature map. More details are given in the implementation chapter.

The proposed loss function is comparable to the logarithmic cross entropy used in a standard GAN:

$$L_{3D-GAN} = \log D(x) + \log (1 - D(G(z))).$$

Here *x* is a real object in a 64×64×64 space, and *z* is a randomly sampled noise vector from a distribution *p*(*z*). However in practice, it is still possible to stick to a cross-entropy formulation. In this work, each dimension of *z* is a uniform distribution over [0, 1].

The necessary training and test data can be downloaded from Princeton's website ShapeNet. Since in a regular 3D-GAN we use 64x64x64 models and the ShapeNet website only has 32x32x32 models, one needs to upsample the models by a factor of 2 in each dimension as a means of preprocessing.

Typically, the initial learning rate of the generator is 0.0025, while that of the discriminator is 0.00001. The mini-batch size is set to 100 and we use an Adam optimizer with β_1 = 0.5. For each mini-batch, we update the discriminator when the accuracy of the previous mini-batch was 0.8 or lower, as suggested by empirical results of the original paper. A total of around 20 000 epochs is a good start, but longer training is always possible.

As we will see in the implementation chapter, training a GAN and in particular a 3D-GAN is widely considered to be a notoriously difficult problem. To start we use common and known heuristic values for the hyperparameters and well-known GAN hacks and improvements summarized by Soumith [66], as described in the GAN chapter. The use of common heuristics is the best starting point. However it can also be beneficial to do hyperparameter tuning by a random search or even Bayesian Hyperparameter Optimization [49], whenever enough resources are available to train for a few days or even weeks.

Results can be visualized by using Tensorboard or Visdom. We preferred the latter due to its simplicity and we can also use Visdom for plotting the generated samples.

Chapter 8: 3D voxel object to mesh conversion

8.1 Motivation

In order to transform a novel 3D voxel model generated by a 3D-GAN to a 3D polygonal mesh consisting of triangular surfaces, we can use the iterative marching cubes algorithm [69]. In general it works for 3D functions, but in the case of a voxel model it is a pointwise 3D function. The algorithm essentially marches over the 3D voxel model which consists of cubes. Voxels are the 3D volumetric analogon of 2D pixels, where the vertices of the cubes are considered the voxels, instead of the actual volume,and are lying on a 3D grid. The marching cubes algorithm will compute if a triangle goes through the cube. When each corner voxel value is 1, this means that the actual cube is lying fully inside the mesh without intersection. When we consider the opposite where each corner voxel value is 0, this means that the actual cube does not intersect the mesh and is lying outside of it. At the surface level some voxels might be 0 while others are 1. Marching cubes tries to compute the triangular polygons by computing intersection points and normals. Since cubes have 8 vertices, we consider 8 voxels per actual cube. Each voxel is a boolean value which can be either 0 or 1. So in total there will be 2^8 = 256 possible cube settings. Marching cubes manages to compute these 256 cube settings by reducing it to 16 cases by observing symmetry such as mirroring or rotations. For higher computational speed it makes use of a dictionary, which serves as a lookup table.

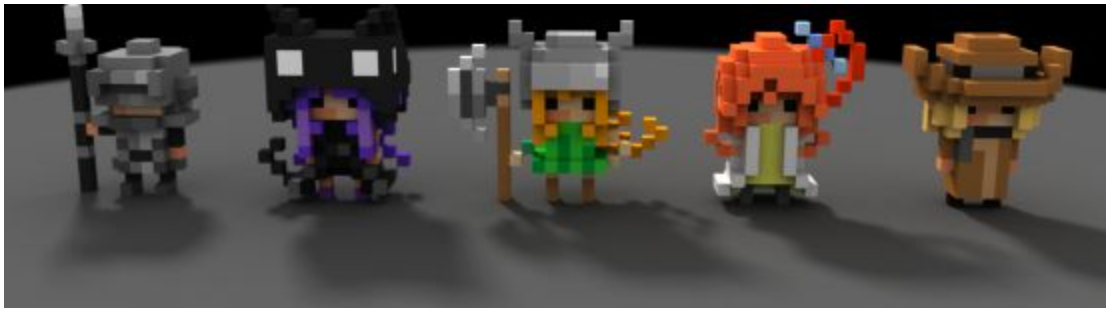

Figure 8.1: Low resolution 3D voxel models consisting of cubes.

8.2 The 2D equivalent: marching squares

When trying to understand the 3D marching squares algorithm in more detail, it can be helpful to first understand its 2D variant, i.e. marching squares. Suppose we have a 2D voxel mesh living in a planar space which we have to divide into 2D triangles to form a desired 2D polygonal mesh. The 2D voxel mesh is lying on a 2D grid of squares.

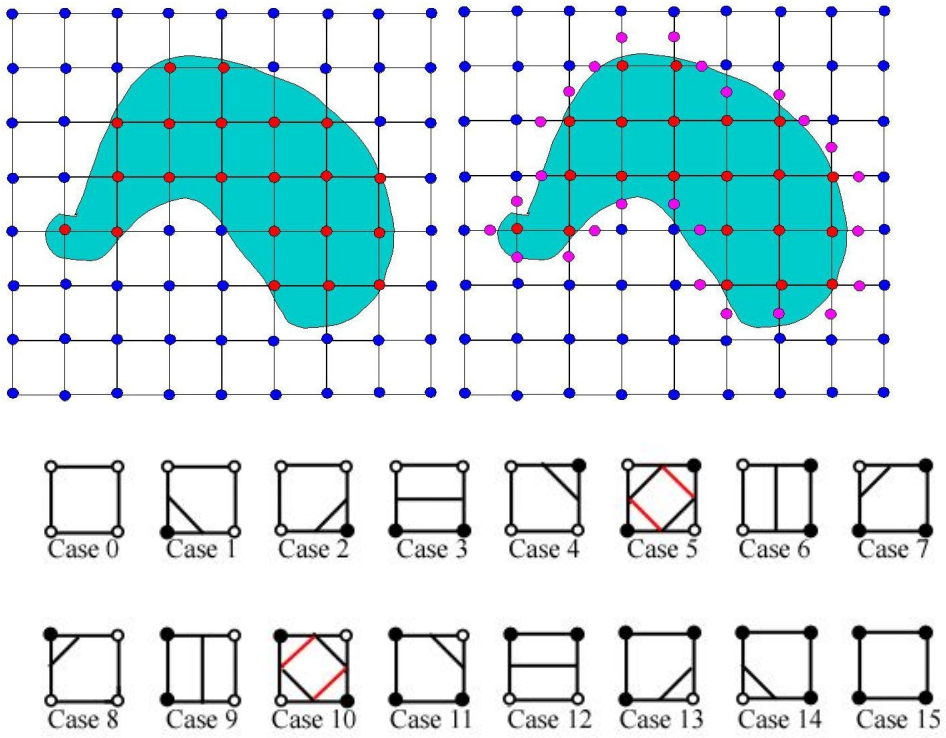

Figure 8.2: The cyan shape is the desired 2D mesh. The red dots represent the given voxels. The 16 possible cases of boundary-square intersections in the 2D marching squares algorithm can be used to determine the fuschia dots. Note that the filled dots represent a 1 and the empty white dots represent a 0.

Let us now consider 16 square configurations, since there are 4 vertices in 2D which means $2^4 = 16$ cases. Squares in the grid which have all vertices in red are fully lying inside the desired mesh and will resolve to case 15. Squares in the grid which have all vertices in blue are lying fully outside the mesh and will resolve to case 0. For each other case, we will hash how the surface borders overlap the squares. We can now use these 16 cases to determine possible intersection points. We will draw them in fuschia. The only step left is the connection of these dots which can be done by a mesh rendering system which uses Delaunay triangulation [71] for instance.

The accuracy of these estimations of the intersection points can be improved by decreasing the size of the squares in the grid. The only downside is that this increases the computation time. Other improvements can be made by using a better interpolation scheme. As we observe the 16 configuration cases, we can see that the intersection points are always estimated to lie in the center of the edges. We can improve this by choosing the intersection points via a linear function, e.g. computing a mass point average so that intersection points will lay closer to the given voxel points.

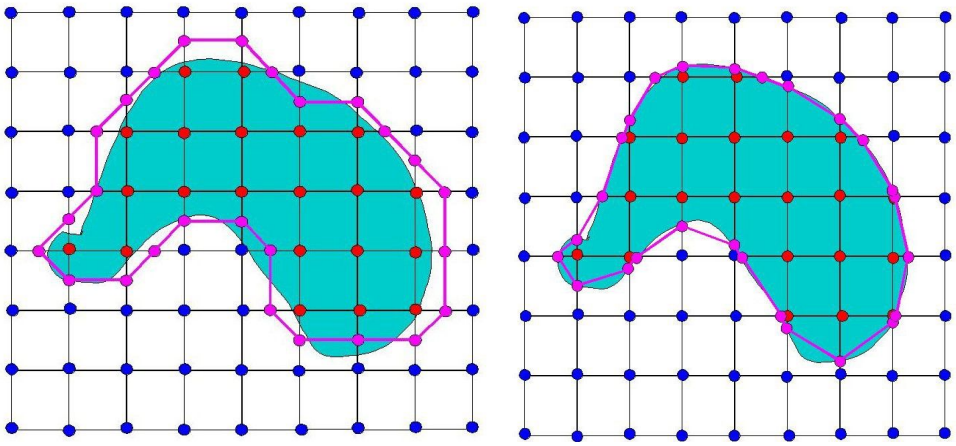

Figure 8.3: Comparison between the standard midpoint estimation at the left and a better interpolation scheme such as the mass point average given at the right.

8.3 The 3D case: marching cubes

The marching cubes algorithm [69, 70] can be used to render a polygonal mesh from volumetric data such as voxels. This algorithm thus produces a triangular mesh by computing isosurfaces from discrete voxel data. The starting point of this algorithm is that it subdivides the space into a set of tiny cubes. Then the algorithm will march through each cube, while testing for corner points as described earlier. Depending on the configuration of corner points, the algorithm then selects an appropriate set of polygons to replace the cube with. As a result, all the generated polygons will form a surface which is an approximation of the surface that was intended by its voxel representation.

We will now gradually describe the underlying algorithm to generate a surface from 3D voxel data. First, we march through the voxel space. We construct an imaginary cube from 8 neighbor locations.

Next, we compute and index for the cube. We do this by comparing 8 density values at the cube vertices to a surface constant. These 8 values can be treated as a bit in a 1-byte-sized integer. When a density value is higher than the surface constant, thus meaning it lies inside the surface, the corresponding bit will be set to 1 and 0 otherwise. Note that we have 8 vertices on a cube and an inside or outside state, so there will be 2^8 or 256 possibilities for the intersection of a surface with a cube. Due to symmetry, these 256 cases can be reduced to 16 unique cases. Next, we create an index for each case. Later on, we can use the computed 1-byte index to lookup a list of precomputed edges from an array. This lookup returns an index into an edge table which returns a value of 12-bits, which could be represented by 2 bytes. This value has bits corresponding to each edge of the cube and represent which edges need to be cut (i.e. a bit-value of 1). At the end, each vertex from these polygons will be positioned on the edge of the cube by using linear interpolation between the 2 density values connected by the edge. Let P_1 and P_2 be vertices or points of an edge on which we need to compute the intersection, while V_1 and V_2 are the density values 0 or 1.

Then the intersection point can be computed by using linear interpolation as follows:

$$P = P_1 + (0.5 - V_1)(P_2 - P_1)/(V_2 - V_1)$$

Using Delaunay triangulation [71] for instance, we can compute a mesh from the resulting point cloud.

For the construction of normals in the interest of applying shaders and illumination, it can be observed that the gradient in each point in the grid corresponds to the normal vector of the constructed polygon in this point. Normals can be interpolated on the cube edges. In our use-case, normals are not so interesting.

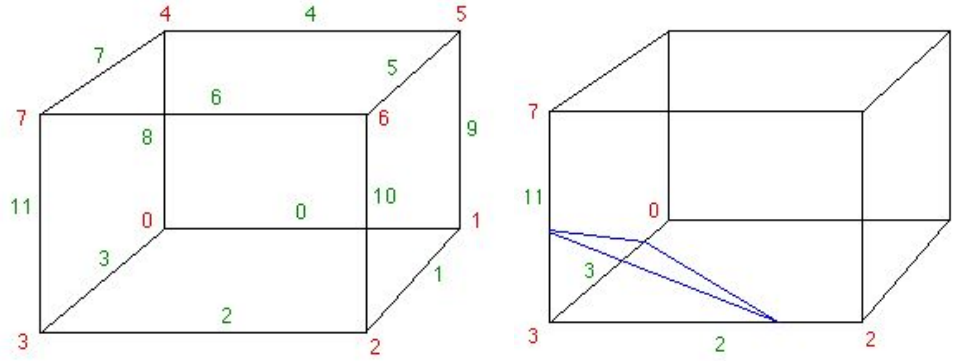

Figure 8.4: Green numbers are edge indexes, while red numbers indicate vertex indexes. Imagine only the voxel at vertex 3 has a value of 1. Then the surface which passes through the cube will intersect edges 3, 2 and 11 in a certain point on each edge. Symmetry can be observed in the inverse case, where all other vertices are 1 and vertex 3 has a value of 0; this results in the same surface. This halves the number of configurations. Using other symmetries such as rotational symmetry, the total of cases can be reduced to 16 cases. By changing the bits of a 1-byte cube index, we can encode which vertices are inside or outside the volume.

Chapter 9: Non-Euclidean Convolution

9.1 Motivation

Deep learning models such as Convolutional Neural Networks (CNNs) allowed us to make progress in a wide range of problems from a different fields (computer vision, speech recognition, natural language processing). Geometric deep learning [76] is a collection of novel techniques attempting to generalize (structured) deep neural models to non-Euclidean domains such as graphs and manifolds, where 3D meshes are a subtype of manifolds. Manifolds are spaces which are locally Euclidean. This generalization is necessary, since a significant amount of fields such as biology, chemistry, physics, computer graphics and network sciences deal with non-Euclidean data. In order to generalize, we need to adjust the Euclidean convolution to a non-Euclidean domain where they were originally not well defined. An important goal of geometric deep learning is to generalize CNN-like constructions such as convolution and pooling to these non-Euclidean domains. In the case of meshes we speak of 2D surfaces embedded in a 3D real space. When comparing Euclidean convolution to non-Euclidean convolution, we see that the former is extrinsic and the latter is an intrinsic operation operating in the embedded space and not in the surrounding space. A recent state-of-the-art method in the spatial domain is the use of Mixture Model CNNs. Since the original paper [77] is very theoretical, we will start from TensorFlow code in this chapter instead for enhanced intuition. Feel free to first read the implementation chapter for simpler neural network architectures.

9.2 Implementation of geometric convolution

We will now define our computational graph for the implementation of 2D geometric convolution as described by Mixture Model CNNs. First, we build a sparse adjacency matrix having a certain connectivity or topology to resemble the graph of our mesh. For simplicity we only implement the 2D case, but the same idea can be applied to 3D as well. We construct two lists: one for the indices and another for the values. Since meshes do not have a certain function defined on their surface domain, we only consider a constant value of 1.0. This is necessary, since geometric convolutions are defined for functions defined on a manifold. Our two lists are built using the following API given below.

```
adjmat_indices_list = []
adjmat_values_list  = []

adjmat_indices_list.append(index_a, index_b)
adjmat_values_list.append(1.0)
```

Next, we start building the computational graph by building a sparse convolutional network. We first create a placeholder input tensor for the input features *X*. Typically, its values will be initialized to zero. It also has the following shape: `(batch_count, grid_sz * grid_sz, feature_count)`. We also introduce a placeholder input tensor *X_xyz* of shape: `(batch_count, grid_sz * grid_sz, 3)`.

We use Numpy to create initial values for our alpha and sigma parameters:

```
mu, stdev = 0, 1
Alpha_init = np.float32(np.random.normal(mu, stdev, (filter_count,3)))
Alpha_init[0:5, 0] = -1
Alpha_init[5:, 1] = 1
Sigma_init = np.float32(np.ones((filter_count, 3)))
Sigma_init[0:5] = 1.0
Sigma_init[5:] = 2.0
```

Now we can introduce these values into the computational graph:

```
Alpha = tf.constant(Alpha_init, tf.float32, (filter_count,3))
Sigma = tf.constant(Sigma_init, tf.float32, (filter_count,3))
```

We can use both `adjmat_indices_list` and `adjmat_values_list` to create a sparse tensor with `tf.SparseTensor()` having a dense shape `[grid_sz * grid_sz, grid_sz * grid_sz]`. We call this a sparse adjacency matrix Adj_sp.

Our weight matrix can be built als follows using an init and inclusion into the computational graph:

```
W_init = np.float32(np.ones((filter_count*feature_count,
      feature_count)))
W = tf.constant(W_init, tf.float32, (filter_count*feature_count,
      feature_count))
```

Next, we build a list of geodesic sparse weight matrices for every mini-batch using the following algorithm [77]:

```
f_count = filter_count
W_sparse_list = []
for b in range(batch_count):
  for dim in range(3):
      Wsp = tf.sparse_add((X_xyz[b,:, dim:(dim+1)]*Adj_sp),
            Adj_sp*tf.transpose(X_xyz[b,:,dim:(dim+1)]*(-1.0)))
      Wsp = tf.sparse_reshape(Wsp, (grid_sz*grid_sz,grid_sz*grid_sz,1))

      Wsp = tf.sparse_concat(axis=2, sp_inputs=[Wsp for i in range(f_count)])

      Adj_sp_rshp = tf.sparse_reshape(Adj_sp,
            (grid_sz*grid_sz,grid_sz*grid_sz,1))
      Adj_sp_rshp = tf.sparse_concat(axis = 2,
            sp_inputs = [Adj_sp_rshp for i in range(f_count)])

      a = tf.reshape(Alpha[:,dim],(1,1,-1))
      s = tf.reshape(Sigma[:,dim],(1,1,-1))
      Wsp = tf.sparse_add(Wsp,Adj_sp_rshp*(-a))
      Wsp = tf.SparseTensor(indices = Wsp.indices,
            values = Wsp.values**2,
            dense_shape=[grid_sz*grid_sz, grid_sz*grid_sz, f_count])
      Wsp = Wsp*(-0.5/(s**2))

      W_sparse = Wsp  if dim == 0 else tf.sparse_add(W_sparse, Wsp)

 # for each batch
 W_sparse = tf.SparseTensor(indices = W_sparse.indices,
      values = tf.exp(W_sparse.values),
      dense_shape=[grid_sz*grid_sz*grid_sz*grid_sz,
            grid_sz*grid_sz,f_count])

 W_sparse = tf.sparse_transpose(W_sparse,(1,0,2))
 W_sparse = tf.sparse_reshape(W_sparse,(grid_sz*grid_sz,
            grid_sz*grid_sz*f_count))
 W_sparse = tf.sparse_transpose(W_sparse,(1,0))

 W_sparse_list.append(W_sparse)
```

Finally, we run a number of succeeding geodesic convolutions given our input features and the list of geodesic sparse weight matrices for every mini-batch. Such a geodesic convolution can be written as follows: for every mini-batch we compute the geodesic features and use them to compute output features using a matrix multiplication with the weight matrix. Geodesic features are computed by using a matrix multiplication between the sparse weight matrices and the dense input features *X*. Output features get appended to the list of all output features for every successive mini-batch. Later, we use the stack operation which stacks a list for R^{th} order tensors such that they form an $(R+1)^{th}$ order tensor of mini-batches along a certain given axis.

```
def geometric_conv(X, W_sparse_list, W, batch_count):
      output_features = []
      for b in range(batch_count):
            geod_features =
                  tf.sparse_tensor_dense_matmul(W_sparse_list[b], X[b])
            geod_features = tf.reshape(geod_features,(grid_sz*grid_sz),
                  filter_count*feature_count))
            output_features.append(tf.matmul(geod_features, W))

      return tf.stack(output_features, axis = 0)
```

Chapter 10: Implementation

In this chapter we will discuss the full implementation of the neural network models and algorithms discussed in the previous chapters. Since the process of developing a 3D-GAN or 3D-GAN for non-Euclidean manifolds requires a lot of understanding from the reader, we will start with a basic neural network topology in a supervised setting for 2D image data. From there we will gradually introduce the expansion to supervised Convolutional Neural Networks, unsupervised Generative Adversarial Networks and combine CNNs and GANs to form Deep Convolutional GANs. DCGANs will serve as the base for expansion from 2D image data to 3D voxel data by using the 3D-GAN architecture. Using the marching cubes algorithm as post processing we can transform generated 3D voxel models into regular polygonal 3D meshes. When replacing the extrinsic convolution on 3D voxel data with the intrinsic non-Euclidean convolution from the previous chapter, we can learn from polygonal meshes instead of voxelized meshes and no post processing step is required since the generated meshes will also be polygonal. We will first start with a discussion concerning the development setup.

10.1 Development setup

10.1.1 Software

The programming language of choice for our experiments is Python. We used Python version 3.5.3 of the Anaconda 4.4.0 (64-bit) distribution on a Windows 10 64-bit operating system. The reason that we use Python is that it is a high-level, general-purpose, dynamically typed multiparadigm programming language. Code written in Python looks remarkably like pseudocode, since it allows us to express very powerful ideas in very few lines of code, while still being very readable. Python allows us to use a number of libraries such as numpy, scipy or matplotlib, which can serve as a powerful environment for scientific computing. We can also use a fair amount of machine learning libraries as well such as:

- TensorFlow: a high-level neural network library;
- scikit-learn: used for data mining, data analysis and machine learning;
- Pylearn2: does the same as scikit-learn, but is more flexible;
- ...

The weaker performance of Python does not matter that much, since we are basically declaring the computational graph in Python, while the TensorFlow backend runs the neural network code and computations on the GPU. Furthermore, Python only requires basic programming knowledge. We spend less time writing or debugging code in comparison to C/C++, Java or C#. Instead we can spend more time on the algorithms, data structures and mathematics related to artificial intelligence and machine learning. It also helps that most tutorials for deep learning are written for Python and TensorFlow.

An important reason for using TensorFlow is that it is available on Windows, my main platform, unlike PyTorch which was only supported for Mac OSX and Linux at the start of this research. Currently it is officially supported on Windows. PyTorch was also pre-1.0 at the time of writing, so the API might not have been stable across multiple versions. A major downside is that TensorFlow uses a static computational graph instead of a dynamic one. This means that in code we cannot change the architecture of the neural network; this a choice made by the designers of TensorFlow for the sake of better optimization possibilities. PyTorch would have been better for research actually. The reason we are not using scikit-learn for deep learning is that it has no neural networks and no GPU support at the time of writing and it is not planned for the near future as well. Torch, the precursor of PyTorch, uses Lua instead of Python, which I am not familiar with. In the Lua programming ecosystem we also do not have the library support (numpy, matplotlib,...) that we have in Python.

The TensorFlow API is also similar to numpy and has much better and easier support for saving and loading models across different environments and even programming languages. Higher level TensorFlow frameworks such as Keras, and TFLearn are not low-level enough for expansion and tweaking of training loops, so we will stick to the lower level API.

Caffe is of the table as well, since it is not flexible. Each node in the computational graph is a layer, so if the researcher wants a new layer type, he/she has to define the full forward, backward propagation and gradient updates. Caffe is also verbose; if you want to support both the CPU and the GPU, you need to implement extra functions for each. Models are also defined in plain text configuration files. In TensorFlow on the other hand each node is a tensor operation (matrix multiplication, convolution,...). Layers can be defined as a composition of these operations and thus the building blocks are more similar and offer more modularity. Models in TensorFlow are also defined programmatically instead of using plain text configuration files.

Theano in combination with Lasagne as a higher level API was another possible alternative. Yet, I found it to have a fairly hard installation process compared to TensorFlow. Theano outperforms TensorFlow on a single GPU, while TensorFlow is more performant on parallel execution across multiple GPUs.

MXNet and its higher API Gluon was actually a better choice than TensorFlow from a coding perspective as it allows us to use the declarative model of TensorFlow and the imperative model of PyTorch in unison. The major downside was that most useful tutorials for our purpose are written in TensorFlow. Documentation for MXNet is in my opinion also not as good as TensorFlow's documentation. So in the end TensorFlow seems to be the best choice for our needs.

10.1.2 Hardware

Our experiments are being performed on one NVidia GeForce GTX 970 graphics card with 1664 available CUDA cores. The CPU is an Intel Core i5-4690K of 4 cores running overclocked at 3.5 GHz. There is 16.0 GB RAM available and 4 GB video memory.

10.2 TensorFlow's computational graph

The entire purpose of TensorFlow is to have a so-called computational graph that can be executed much more efficiently than if the same calculations were to be performed directly in Python. TensorFlow can be more efficient than NumPy because TensorFlow knows the entire computational graph that must be executed, while NumPy only knows the computation of a single mathematical operation at a time.

TensorFlow can also automatically compute the gradients that are needed to optimize the variables of the graph so as to make the model perform better. This is because the graph is a combination of simple mathematical expressions, so the gradient of the entire graph can be calculated using the chain-rule for derivatives.

TensorFlow can also take advantage of multi-core CPUs as well as GPUs - and Google has even built special chips especially for TensorFlow which are called TPUs (Tensor Processing Units), which are even faster than GPUs for deep learning purposes.

A TensorFlow graph consists of the following parts:

- placeholder variables used for input and output data of the graph;
- variables that are going to be optimized so as to make the neural network perform better;
- the mathematical formulas for the neural network;
- a cost measure that can be used to guide the optimization of the variables;
- an optimization method which updates the variables;
- the TensorFlow graph may also contain various debugging statements, e.g. for logging data to be displayed using TensorBoard.

As a simple example, first we build the computational graph. Note that no calculations are being performed yet. We only create abstract tensors in the computational graph.

```
x1 = tf.constant(8)
x2 = tf.constant(9)

result_tensor = tf.multiply(x1, x2)
```

In order to actually compute the result, we need to run a session and automatically close it.

```
with tf.Session() as sess:
      output = sess.run(result_tensor)
      print(output)
```

Next, we can open the Anaconda command prompt and activate the TensorFlow (GPU) environment and running the Python script as usual and observe the output.

10.3 Vanilla neural networks in TensorFlow

Let us say we have the following neural network topology:

- an input layer with weights connected to the next layer;
- a first hidden layer of 500 neurons with weights connected to the next layer;
- a second hidden layer of 500 neurons with weights connected to the next layer;
- a third hidden layer of 500 neurons with weights connected to the next layer;
- an output layer.

Each hidden layer has a ReLU activation function used in its neurons. We compare the output to the desired output via cost function which involves the Softmax cross entropy function. Next, we use an optimization function (optimizer) to minimize the cost. Here we use the Adam optimizer, but Stochastic Gradient Descent, RMSProp, AdaGrad,... are also possible. We use backpropagation to learn the weights. An epoch is one cycle which involves a feed forward pass and a backpropagation pass. After each epoch we lower the cost function.

First we load the data. To start and get our architecture on point, we will use the MNIST data set of the numbers 0 to 9 for training and test data. MNIST images are fairly low resolution images of 28x28 pixels. Later on in the more complex architectures to come we will move on to 3D data. Since we need to store these 2D images in a 1D array as an input feature vector, the flattened size is 784. We load the associated labels to each image as one-hot data. One-hot means that if we have 10 possible output classes, each class label is a 10-dimensional vector with a value of 1 for the associated class, while the rest of the components are 0. For instance: 0 = [1,0,0,0,0,0,0,0,0,0], 1 = [0,1,0,0,0,0,0,0,0,0], etc.

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("../data/MNIST/", one_hot=True)
img_size = 28
img_size_flat = img_size * img_size
img_shape = (img_size, img_size)
class_count = 10
```

The hyperparameters for our network are the number of nodes in each hidden layer, the batch size and the number of epochs. The higher the batch size, the less precision but the faster the calculations will be if the GPU supports higher batch sizes.

```
hl1_node_count = 500
hl2_node_count = 500
hl3_node_count = 500
```

```
	batch_size = 128
	epoch_count = 10
```

Before we start building the computational graph for the model, we will define a few functions. Our first function will build the necessary neural network topology described earlier. We initialize the weight matrices and bias vectors as random values from a normal (Gaussian) distribution and constants of 0.1. Since we perform matrix and vector math to improve SIMD and SIMT computations performed by the GPU, we perform matrix multiplication of the weights and inputs from the different layers as a whole. Later on the bias vectors will be added to the resulting vectors to form the raw activation. Later ReLU will be performed. At the end we return the output vector.

```
def neural_network(data):
	hidden_layer_1 = {
		'weights': tf.Variable(tf.random_normal([784, hl1_node_count])),
		'biases': tf.Variable(tf.constant(0.1, shape=[hl1_node_count]))}

	hidden_layer_2 = {
		'weights': tf.Variable(tf.random_normal([hl1_node_count,
				hl2_node_count])),
		'biases': tf.Variable(tf.constant(0.1, shape=[hl2_node_count]))}

	hidden_layer_3 = {
		'weights': tf.Variable(tf.random_normal([hl2_node_count,
				hl3_node_count])),
		'biases': tf.Variable(tf.constant(0.1, shape=[hl3_node_count]))}

	output_layer = {
		'weights': tf.Variable(tf.random_normal([hl3_node_count,
				class_count])),
		'biases': tf.Variable(tf.random_normal([class_count]))}

	l1 = tf.add(tf.matmul(data, hidden_layer_1['weights']),
			hidden_layer_1['biases'])
	l1 = tf.nn.relu(l1)
	l2 = tf.add(tf.matmul(l1, hidden_layer_2['weights']),
			hidden_layer_2['biases'])
	l2 = tf.nn.relu(l2)
	l3 = tf.add(tf.matmul(l2, hidden_layer_3['weights']),
			hidden_layer_3['biases'])
	l3 = tf.nn.relu(l3)
	return tf.add(tf.matmul(l3, output_layer['weights']),
			output_layer['biases'])
```

We now define a function to compute the cost based on the actual predicted output and the desired output. The cross-entropy is a performance measure used in classification which we want to minimize. As seen earlier it is a positive continuous function which turns to zero whenever the predicted output equals the desired output. We can minimize to cross-entropy to be close to zero by changing the parameters of the model across the different layers. TensorFlow has a built-in function for calculating the cross-entropy. This will serve as our loss. We then compute the mean of the elements across the dimensions of the loss tensor.

```
def compute_cost(predicted_output, desired_output):
      loss = tf.nn.softmax_cross_entropy_with_logits_v2(
            logits=predicted_output, labels=desired_output)
      return tf.reduce_mean(loss)
```

After defining our helper functions, we will build the entire computational graph and specify the cost function and optimizer of the neural network which will be run on our GPU device. The flattened input vector and desired output vectors will also be initialized.

```
      device = '/gpu:0'
      device_config = tf.ConfigProto(allow_soft_placement=True)
      device_config.gpu_options.allow_growth = True
      with tf.device(device):
            g_input = tf.placeholder(tf.float32, shape=[None, img_size_flat])
            g_desired_output = tf.placeholder(tf.float32)
            predicted_output = neural_network(g_input)
            cost = compute_cost(predicted_output, g_desired_output)
            optimizer = tf.train.AdamOptimizer().minimize(cost)
```

When the computational graph is defined, we can start to run our training session. We do a total of 10 epochs, but more epochs are possible as it is one of our hyperparameters. We iterate over each mini-batch every epoch and run the optimizer to minimize the loss. The optimizer also needs the desired output in order to compute the gradients like we discussed in the theory. This will give us a batch loss which we will at to the total loss for each epoch. The epoch loss should decrease after each epoch, since the network will be learning and minimizing the loss.

```
with tf.Session(config = device_config) as sess:
      sess.run(tf.global_variables_initializer())

      for epoch in range(epoch_count):
            epoch_loss = 0
            batch_count = int(mnist.train.num_examples/batch_size)

            for i in range(batch_count):
                  epoch_input, epoch_desired_output =
                        mnist.train.next_batch(batch_size)
                  i, batch_loss = sess.run(
                        [optimizer, cost],
                        feed_dict = {
                              g_input: epoch_input,
                              g_desired_output: epoch_desired_output})
                  epoch_loss += batch_loss
```

At the end we can print some statistics about the performance of the neural network. The argmax operation returns the index with the largest value across a certain axis of a tensor. Here we have matrices, so for each row we will pick the index of the largest element in that row. This way we can receive the actual class with the highest probability. From there we can calculate the training accuracy. This accuracy is expected to be high; around 99% for this model.

```
predicted_class = tf.argmax(predicted_output, 1)
desired_class = tf.argmax(g_desired_output, 1)
correctness = tf.equal(predicted_class, desired_class)
accuracy = tf.reduce_mean(tf.cast(correctness, 'float'))

accuracy_calculated = accuracy.eval({g_input: mnist.train.images,
      g_desired_output: mnist.train.labels})
print(accuracy_calculated)
```

The model can also be evaluated on a test data batch. On test data, the accuracy is expected to be lower; around 95% for this model.

```
accuracy_calculated = accuracy.eval({g_input: mnist.test.images,
      g_desired_output: mnist.test.labels})
print(accuracy_calculated)
```

10.3 CNNs in TensorFlow

Let us say we have the following Convolutional Neural Network topology:

- an input layer;
- convolutional and pooling layers;
- a fully-connected layer where neurons from neighboring layers are all connected to each other;
- an output layer for the classes.

As for convolution, we use 5x5 filters which create a feature map from the image. We use a movement stride of 1 pixel in each dimension (y, x). We also use zero-padding so the output size of each convolution is the same. For pooling we use max pooling with a 2x2 pooling window and a movement size (stride) of 2 pixels in each dimension. We calculate the maximum value of each 2x2 matrix which appears under 2x2 pooling window. We will also introduce dropout where we are mimicking failing neurons. Sometimes at a probability of 0.2, the neurons can drop out. As discussed in the theory, dropout will help for large datasets and serves as a regularization technique used to improve generalization.

The MNIST data and properties will be loaded as discussed in the previous section and likewise for GPU configurations. The hyperparameters are also the same, except since we are using dropout we will use a keep probability of 0.8, i.e. a dropout probability of 20% per neuron.

```
keep_prob = 0.8
```

We will again introduce some helper functions. Note that for the convolution and pooling operations the first and last stride always need to be 1, because the first value is used for the image-number and the last one stands for the input-channel. The padding is set to 'SAME' which means the input image is padded with zeros, so the size of the output is the same as the input and this requires zero-padding as the filter needs to look beyond the image borders. The cost will be computed as in the previous section.

```
def weight_variable(shape):
    return tf.Variable(tf.truncated_normal(shape, stddev=0.05))

def bias_variable(count):
    return tf.Variable(tf.constant(0.1, shape=[count]))

def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1,1,1,1], padding='SAME')

def max_pool_2x2(x):
      return tf.nn.max_pool(x, ksize=[1,2,2,1], strides=[1,2,2,1],
                  padding='SAME')
```

The structure of our convolutional neural network model can be expressed by setting up the weight and bias tensors for each layer:

```
weights = {'conv_1': weight_variable([5,5,1,32]),
           'conv_2': weight_variable([5,5,32,64]),
           'fc': weight_variable([64*7*7, 1024]),
           'output': weight_variable([1024, class_count])}

biases = {'conv_1': bias_variable(32),
          'conv_2': bias_variable(64),
          'fc': bias_variable(1024),
          'output': bias_variable(class_count)}
```

The first convolutional layer consists of 32 5x5 filters and 1 input channel (monochromatic color). The second convolutional layer has 32 inputs and consists of 64 5x5 filters and thus 2 filters per input. Next we have a fully connected layer which takes 64 inputs of 7x7 pixels as a flattened vector connected to 1024 nodes. As a result there will be 64 outputs. The final output layer takes the 1024 inputs, which are the outputs of the previous layer and map them to the total number of classes. For each output value of each layer there is a bias.

Before a 2D image can be used, one needs to reshape the flattened 1D array back to a 2D image. Here we use -1 to infer the amount of images in the mini-batch automatically.

```
data = tf.reshape(data, [img_count, img_size, img_size, channel_count])
```

When the data and the weights are set up, one can define the actual operations used in the network. It should be noted that normally a ReLU layer would be defined before a pooling layer, but since both operations are commutative we can save 75% of the ReLU operations by doing max pooling first. Next, we use a fully connected layer which uses a ReLU activation function in every node. Depending on the cost computation, we can use raw logits or apply Softmax at the final output layer, but here we prefer the first option since `tf.nn.softmax_cross_entropy_with_logits_v2()` accepts raw logits.

```
conv_1 = conv2d(data, weights['conv_1'])
conv_1 = max_pool_2x2(conv_1)
conv_1 = tf.nn.relu(conv_1)

conv_2 = conv2d(conv_1, weights['conv_2'])
conv_2 = max_pool_2x2(conv_2)
conv_2 = tf.nn.relu(conv_2)

fc_node_count = 64*7*7
fc = tf.reshape(conv_2, [-1, fc_node_count])
fc = tf.nn.relu(tf.add(tf.matmul(fc, weights['fc']), biases['fc']))
fc = tf.nn.dropout(fc, keep_prob)
```

```
output_logit = tf.add(tf.matmul(fc, weights['output']),
                                    biases['output'])
```

The training process and use of the Adam optimizer is identical to that of the previous section. As a side note, we can plot MNIST images in grayscale using matplotlib. Here we use a batch size of 1 to fetch the next image. The index of 0 corresponds to the training data rather than the training label, which has an index of 1. Since the training data is already shaped into a 784-dimensional column vector, we reshape it to a 28x28 pixel image in order to plot it.

```
import matplotlib.pyplot as plt
example_image = mnist.train.next_batch(1)[0]
img_size = 28
print(example_image.shape)
example_image = example_image.reshape([img_size, img_size])
plt.imshow(example_image, cmap='Greys')
plt.show()
```

Or a GridSpec can be used to create a grid of 4 by 4 images from a mini-batch of e.g. 16 images.

```
def plot(images, img_width, img_height):
      figure = plt.figure(figsize=(4, 4))
      gs = gridspec.GridSpec(4, 4)
      gs.update(wspace=0.04, hspace=0.04)

      for index, img in enumerate(images):
            axis = plt.subplot(gs[index])
            axis.set_xticklabels([])
            axis.set_yticklabels([])
            axis.set_aspect('equal')
            plt.axis('off')
            plt.imshow(img.reshape(img_width,img_height), cmap='Greys')

      plt.show()

plot(mnist.train.next_batch(16)[0], img_width, img_height)
```

10.4 GANs in TensorFlow

Loading the data happens in the same fashion as the previous sections. To save some generated image examples after a certain amount of epochs, we create an additional folder.

```
if not os.path.exists("../results/"):
      os.makedirs("../results/")
```

We also define some hyperparameters for the GAN model.

```
learning_rate = 0.001
batch_size = 128
Z_size = 100
epoch_count = 1000000
D_h1_node_count = 128
G_h1_node_count = 128
```

The chosen weight initialization is the one described by He et al., since we are dealing with ReLU and Leaky ReLU activation functions, as discussed earlier in the theory. Whenever we use tanh or sigmoid activation functions, we will use Xavier initialization instead. Biases will get initialized to zero.

```
def he_et_al_initializer(shape):
      n_in = shape[0]
      stddev = 1.0 / tf.sqrt(0.5 * n_in)
      return tf.random_normal(shape=shape, stddev=stddev)

def weight_variables(name, shape):
      return tf.get_variable(name,
                  shape=shape,initializer=he_et_al_initializer())

def bias_variables(name, count):
      return tf.get_variable(name, [count],
                  initializer=tf.constant_initializer(0.0))
```

A basic discriminator network can be built by setting up weight and bias matrices for each hidden layer. The shape of the matrices uses the [input size, output size] pattern. We use a ReLU activation function for each node in the first hidden layer, but now again written as function operating on an entire matrix to exploit vectorization on the GPU. The output layer only contains one node. This will be a logit value which can be turned into a probability, with 0 as fake and 1 as real data, by the use of a sigmoid function. We return a raw logit, because our chosen cross entropy function uses the sigmoid function internally already.

```
def discriminator_model(x):
      with tf.variable_scope("discriminator", reuse=tf.AUTO_REUSE):
            # Parameters from input layer to hidden layer 1
            D_W1 = weight_variables("D_W1", [img_size_flat, D_h1_node_count])
            D_b1 = bias_variables("D_b1", D_h1_node_count)

            # Parameters from hidden layer 1 to output layer
            D_W2 = weight_variables("D_W2", [D_h1_node_count, 1])
            D_b2 = bias_variables("D_b2", 1)

            D_h1 = tf.nn.relu(tf.add(tf.matmul(x, D_W1), D_b1))
            D_output_logit = tf.add(tf.matmul(D_h1, D_W2), D_b2)

      return D_output_logit
```

Before we can use a generator model, we need to pass it a randomly sampled uniform noise vector *z* with real values in [-1, 1]. The size of this vector, i.e. its dimensionality, is typically 100-dimensional, but higher dimensionality is possible for more complex data such as 3D models. In our training loop we have a tendency to operate on entire mini-batches. As such this adds another dimensionality to our tensors. This implies that we need to generate a noise matrix *Z* rather than a noise vector *z*. Its row size typically corresponds to the mini-batch size and the column size corresponds to the dimensionality of the noise vector, e.g. 100.

```
      def Z_random_noise(row_count, col_count):
            return np.random.uniform(-1.0, 1.0, size=[row_count, col_count])
```

The generator takes a row from this random noise matrix as its noise vector *z* to generate novel samples from. We feed the noise vector as an input to the generator. The generator model consists of one hidden layer in this example, while each neuron in this layer has a ReLU activation function. At the output layer we added a sigmoid activation function. This function squeezes pixel values that would appear grey toward either black or white values, resulting in a sharper image quality.

```
def generator_model(z):
      with tf.variable_scope("generator", reuse=tf.AUTO_REUSE):
            # Parameters from input layer to hidden layer 1
            G_W1 = weight_variables("G_W1", [Z_size, G_h1_node_count])
            G_b1 = bias_variables("G_b1", G_h1_node_count)

            # Parameters from hidden layer 1 to output layer
            G_W2 = weight_variables("G_W2", [G_h1_node_count, img_size_flat])
            G_b2 = bias_variables("G_b2", img_size_flat)

            G_h1 = tf.nn.relu(tf.add(tf.matmul(z, G_W1), G_b1))
            G_sample = tf.nn.sigmoid(tf.add(tf.matmul(G_h1, G_W2), G_b2))

      return G_sample
```

When performing the actual training, one needs to pre-allocate two placeholder variables. X is the placeholder for the input data, while Z serves as the placeholder for the random noise vector. Both are (flattened) vectors.

```
      X = tf.placeholder(tf.float32, shape=[None, img_size_flat])
      Z = tf.placeholder(tf.float32, shape=[None, Z_size])
```

To build the computational graph we create a generator model which will generate a novel image based on the noise vector. We also use two discriminator models: one which classifies real data from the dataset and another which classifies the generated images. Since we use automatic reuse of variables in the scope of the generator model, the weights and biases are shared by both models, so essentially we can treat the discriminator as one model.

```
      G_sample = generator_model(Z)
      D_logit_real = discriminator_model(X)
      D_logit_fake = discriminator_model(G_sample)
```

Instead of using the original logarithmic loss functions from the theory, we use a sigmoid cross entropy with logits version of the loss. Using the original log loss functions we got warnings and bad output.

```
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)))
```

Since we need to pass the weights and biases of both the discriminator and the generator to the Adam optimizer, we need to get them from these scopes. Both the discriminator and the generator have a separate optimizer and use the same initial learning rate, although it is common to use different values. The optimizers minimize the defined discriminator and generator losses by adjusting their parameters appropriately.

```
      with tf.variable_scope("discriminator", reuse=True):
            D_params = [
                  tf.get_variable("D_W1"),
                  tf.get_variable("D_W2"),
                  tf.get_variable("D_b1"),
                  tf.get_variable("D_b2")
            ]

      with tf.variable_scope("generator", reuse=True):
            G_params = [
                  tf.get_variable("G_W1"),
                  tf.get_variable("G_W2"),
                  tf.get_variable("G_b1"),
                  tf.get_variable("G_b2")
            ]

      D_optimizer =
      tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(D_loss,
            var_list=D_params)
      G_optimizer =
      tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(G_loss,
            var_list=G_params)
```

When the computational graph and its loss functions and optimizers are defined, we can start a training session which will execute the graph.

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

When developing and testing our Generative Adversarial Networks, we train for a given number of epochs. Each 2000 epochs for example, we can print our loss and save 16 generated images to one large grid of 4 by 4 images using our plotting function defined earlier. These images are printed as means of a test run for the generator. At the first epoch, only noise will be produced, since the generator has not been trained yet. At the end of the epoch we print the loss of both the generator and discriminator to inspect any invalid behavior every 2000 epochs. In the middle, we perform the actual training. We gather the data from the next mini-batch. Labels are discarded because we now run unsupervised models. Both the discriminator and generator are trained by running their Adam optimizers over the mini-batches, trying to minimize their losses.

```
index = 0
epoch_print_count = 2000
for epoch in range(epoch_count):
      if epoch % epoch_print_count == 0:
            samples = sess.run(G_sample,
                        feed_dict={Z: Z_random_noise(4*4, Z_size)})
            figure = plot(samples, img_size, img_size)
            plt.savefig("../results/{}.png".format(str(index).zfill(5)),
                  bbox_inches="tight")
            plt.close(figure)
            index += 1

      X_batch, _ = mnist.train.next_batch(batch_size)
      _, D_loss_epoch = sess.run([D_optimizer, D_loss], feed_dict={X:
            X_batch, Z: Z_random_noise(batch_size, Z_size)})
      _, G_loss_epoch = sess.run([G_optimizer, G_loss], feed_dict={Z:
            Z_random_noise(batch_size, Z_size)})

      if epoch % epoch_print_count == 0:
            print("Epoch: {}".format(epoch))
            print("Discriminator loss: {:.5}". format(D_loss_epoch))
            print("Generator loss: {:.5}\n".format(G_loss_epoch))
```

10.5 WGANs in TensorFlow

As seen in the theory, Wasserstein GANs (WGANs) use the same architecture as the vanilla GAN, but they differ in a few aspects. The only differences between WGAN and GAN are the definitions of the loss functions, the use of the RMSProp optimizer instead of the Adam optimizer and the use of weight clipping. We clip the weights of the discriminator to a range of [-0.01, 0.01] to enforce a Lipschitz constraint on the discriminator. The clipped weights get also fed to the session runner for the discriminator optimizer. The initial learning rate is also set to 0.0001 instead of 0.001. This can be expressed in TensorFlow as:

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

      D_optimizer =
tf.train.RMSPropOptimizer(learning_rate=learning_rate).minimize(-D_loss,
            var_list=D_params)

      G_optimizer =
tf.train.RMSPropOptimizer(learning_rate=learning_rate).minimize(G_loss,
            var_list=G_params)

      clip_D = [param.assign(tf.clip_by_value(param, -0.01, 0.01))
                  for param in D_params]
```

10.6 LSGANs in TensorFlow

As we discussed in the theory, the only differences between LSGAN and GAN are the loss functions.

```
      D_loss = 0.5 * (tf.reduce_mean((D_real - 1)**2)
                  + tf.reduce_mean(D_fake**2))
      G_loss = 0.5 * tf.reduce_mean((D_fake - 1)**2)
```

10.7 DCGANs in TensorFlow

In the DCGAN setup, the only difference is the definition of the models. In this example we also show how to apply a sigmoid at the output layers instead of using raw logits for cross entropy computation. First we define a few helper functions as described in the previous sections. We create an additional function named lrelu() which computes the Leaky ReLU of a value using a given slope.

```
def lrelu(x, slope=0.2):
      return tf.maximum(slope * x, x)
```

The He et al. initializer can be used by a built in function of TensorFlow using custom parameters but is given here for completeness.

```
def he_init(shape):
      row_count = shape[0]
      stddev = 1.0 / tf.sqrt(0.5 * row_count)
      return tf.random_normal(shape=shape, stddev=stddev)

def weight_variables(name, shape):
      return tf.get_variable(name, shape=shape,
      initializer=tf.contrib.layers.variance_scaling_initializer(
            factor=2.0, mode='FAN_IN', uniform=False))
```

For the actual simplified implementation of the discriminator model, we create 3 convolutional layers, each followed by an additional Leaky ReLU layer. In the first layers, we use 2D convolution with a stride of 2 in each dimension and zero-padding. In the last layer, we use a fully connected layer for simplicity and no additional Leaky ReLU activation. Later when developing our 3D-GAN, we will go fully convolutional and use an additional convolutional layer. At the end, we return the sigmoid of the raw logits as described by the function signature. As far as the choice of the filter stack in each layer goes, we use the following setup:

- we reshape the input to a mini-batch of 28x28 monochromatic pixel images,
- for convolutional layer 1 we use 64 5x5 filters with strides of 2x2 on 1 input channel to retrieve a mini-batch of 64 14x14 feature maps as an output shape of size [-1, 14, 14, 64],
- for convolutional layer 2 we use 128 5x5 filters with strides of 2x2 on 64 input feature maps to retrieve a mini-batch of 128 7x7 feature maps as an output shape of size [-1, 7, 7, 128],
- a dense (fully connected) layer of 256 Leaky ReLU nodes of 50% dropout probability per mini-batch resulting in 1 output, fed to a sigmoid.

Note that strides of 2x2 will halve the width and the height of the given image. We also used no further convolutional layers with strides of 2x2, since 7 is an odd number. The function which creates the discriminator model has the following signature: it takes an input in the form of real or fake data and

returns a raw logit value (not a probability), which will later be used in the cross entropy computation.

```
def discriminator_model(x):
      with tf.variable_scope("discriminator", reuse=tf.AUTO_REUSE):
      # ...
      return output
```

The body of this function is written below:

```
channel_count = 1
F = 5       # 5x5 filters
x_rs = tf.reshape(x, [-1, 28, 28, channel_count])

# 1st Convolutional layer: input shape = [-1, 28, 28, 1]
# and output shape = [-1, 14, 14, 64]
K1 = 64
D_W1 = weight_variables("D_W1", [F, F, channel_count, K1])
D_conv1 = tf.nn.conv2d(x_rs, D_W1, strides=[1,2,2,1], padding="SAME")
D_conv1 = tf.contrib.layers.batch_norm(D_conv1)
D_conv1 = lrelu(D_conv1)

# 2nd Convolutional layer: input shape = [-1, 14, 14, 64]
# and output shape = [-1, 7, 7, 128]
K2 = 128
D_W2 = weight_variables("D_W2", [F, F, K1, K2])
D_conv2 = tf.nn.conv2d(D_conv1,D_W2,strides=[1,2,2,1], padding="SAME")
D_conv2 = tf.contrib.layers.batch_norm(D_conv2)
D_conv2 = lrelu(D_conv2)

# Dense layer (fully connected): input shape = [-1, 7, 7, 128]
# and output shape = [-1, 1]
# Node count = [-1, 256]
fc_node_count = 256
D_fc_nodes = tf.reshape(D_conv2, [-1, fc_node_count])
D_fc_nodes = lrelu(D_fc_nodes)
input = tf.nn.dropout(D_fc_nodes, 0.5)

D_Wfc = weight_variables("D_Wfc", [fc_node_count, 1])
D_bfc = bias_variables("D_bfc", 1)
logit = tf.matmul(input, D_Wfc) + D_bfc
output = tf.nn.sigmoid(logit)
```

For the actual simplified implementation of the generator model, we create 2 fully connected batch-normalized layers each followed by an additional ReLU layer.
In the first transposed convolutional layer, we use 2D transposed convolution with a stride of 2 in each dimension and zero-padding. As far as the choice of the filter stack in each layer goes, we use the following setup:

- for the first dense layer we use 1024 nodes,
- for the second dense layer we use 7*7*128 nodes,
- for transposed convolutional layer 1 we use 64 5x5 filters with strides of 2x2 on 128 inputs to retrieve a mini-batch of 64 14x14 feature maps as an output shape of size [-1, 14, 14, 64],
- for transposed convolutional layer 2 we use 1 5x5 filter with strides of 2x2 on 64 input feature maps to retrieve a mini-batch of 28x28 output images.

The function which creates the generator model has the following signature: it takes a random noise vector *z* as an input and returns an upscaled version which is a novel 2D image.

```
def generator_model(z):
      with tf.variable_scope("generator", reuse=tf.AUTO_REUSE):
      # ...
      return output
```

The body of this function is given below. Note that since we are not using a deep network, we do not need to go fully convolutional, hence the use of two dense layers at the start.

```
# 1st FC layer: [-1, 1024]
G_fc1_input_size = Z_size
G_fc1_node_count = 1024
G_W1 = weight_variables("G_W1", [G_fc1_input_size, G_fc1_node_count])
G_fc1 = tf.matmul(z, G_W1)
G_fc1 = tf.contrib.layers.batch_norm(G_fc1)
G_fc1 = tf.nn.relu(G_fc1)

# 2nd FC layer: [-1, 7*7*128]
G_fc2_input_size = G_fc1_node_count
G_fc2_node_count = 7*7*128
G_W2 = weight_variables("G_W2", [G_fc2_input_size, G_fc2_node_count])
G_fc2 = tf.matmul(G_fc1, G_W2)
G_fc2 = tf.contrib.layers.batch_norm(G_fc2)
G_fc2 = tf.nn.relu(G_fc2)
G_fc2 = tf.reshape(G_fc2, [batch_size, 7, 7, 128])
```

```
# 1st Deconvolutional layer
G_W3 = weight_variables("G_W3", [5, 5, 64, 128])
G_deconv3 = tf.nn.conv2d_transpose(G_fc2, G_W3,
      (-1,14,14,64), strides=[1,2,2,1], padding="SAME")
G_deconv3 = tf.contrib.layers.batch_norm(G_deconv3)
G_deconv3 = tf.nn.relu(G_deconv3)

# 2nd Deconvolutional layer
G_W4 = weight_variables("G_W4", [5, 5, 1, 64])
G_deconv4 = tf.nn.conv2d_transpose(G_deconv3, G_W4,
      (-1,28,28,1), strides=[1,2,2,1], padding="SAME")
G_deconv4 = tf.reshape(G_deconv4, [-1, 28*28])
output = tf.nn.sigmoid(G_deconv4)
```

When not using raw logits and a sigmoid function upfront, we can use the logarithmic loss functions as discussed in the theory:

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

Here we train the discriminator to correctly judge whether the data is real or not and the generator to fool the discriminator as if it was producing real data. The training process of the DCGAN is identical to the other GAN models we have discussed earlier.

10.8 3D-GANs in TensorFlow

The data we read is 3D volumetric voxel data, which can be represented by a 3D array structure and so our TensorFlow tensors will involve an order increase by one. We define some hyperparameters for the 3D-GAN model. We intentionally set the initial learning rate of the discriminator lower, so the generator will not be criticized too hard in the beginning of the training process. 3D models consist of a 64x64x64 voxel grid. Since volumetric data takes up more working memory, we lower the mini-batch size to 32. 3D models are more complex data, so we follow the guidelines of the 3D-GAN paper and increase the dimensionality of the noise vector *z* from 100 to 200.

```
D_learning_rate = 1e-5  # or 1e-3
G_learning_rate = 0.0025
beta1 = 0.5
cubic_size = 64
batch_size = 32
z_size = 200
epoch_count = 10000
```

As before the weights get initialized by the He et al. initializer, since we will use ReLU and Leaky ReLU activation functions. Leaky ReLU also gets a custom implementation, since it is not directly included in the TensorFlow library. We do not use biases because batch normalization already uses the addition of a bias. The He et al. initializer can be used by a built in function of TensorFlow using custom parameters.

```
def lrelu(x, slope=0.2):
      return tf.maximum(slope * x, x)

def he_init(shape):
      row_count = shape[0]
      stddev = 1.0 / tf.sqrt(0.5 * row_count)
      return tf.random_normal(shape=shape, stddev=stddev)

def weight_variables(name, shape):
      return tf.get_variable(name, shape=shape,
      initializer=tf.contrib.layers.variance_scaling_initializer(
            factor=2.0, mode='FAN_IN', uniform=False))
```

The function which creates the discriminator model has the following signature: it takes an input in the form of real or fake data and returns a raw logit value (not a probability), which will later be used in the cross entropy computation.

```
def discriminator_model(input, is_training=True):
      with tf.variable_scope("discriminator", reuse=tf.AUTO_REUSE):
      # ...
      return raw_logit
```

For the actual implementation of the discriminator model, we create 5 volumetric, batch-normalized convolutional layers, each followed by an additional Leaky ReLU layer. In the first layers, we use 3D convolution with a stride of 2 in each dimension and zero-padding. In the last layer, we use strides of 1 in each dimension, no padding and no additional Leaky ReLU activation. At the end, we return the raw logits as described by the function signature. As far as the choice of the filter stack in each layer goes, we use the following setup:

- for convolutional layer 1 we use 64 4x4x4 filters on 1 input,
- for convolutional layer 2 we use 128 4x4x4 filters on 64 input feature maps,
- for convolutional layer 3 we use 256 4x4x4 filters on 128 input feature maps,
- for convolutional layer 4 we use 512 4x4x4 filters on 256 input feature maps,
- and for convolutional layer 5 we use 1 4x4x4 filter on 512 input feature maps to go back to one value.

```
D_W1 = weight_variables("D_W1", [4, 4, 4, 1, 64])
D_conv1 = tf.nn.conv3d(input, D_W1, strides=[1,2,2,2,1],
      padding="SAME")
D_conv1 = tf.contrib.layers.batch_norm(D_conv1,
      is_training=is_training)
D_conv1 = lrelu(D_conv1)

D_W2 = weight_variables("D_W2", [4, 4, 4, 64, 128])
D_conv2 = tf.nn.conv3d(D_conv1, D_W2, strides=[1,2,2,2,1],
      padding="SAME")
D_conv2 = tf.contrib.layers.batch_norm(D_conv2,
      is_training=is_training)
D_conv2 = lrelu(D_conv2)

D_W3 = weight_variables("D_W3", [4, 4, 4, 128, 256])
D_conv3 = tf.nn.conv3d(D_conv2, D_W3, strides=[1,2,2,2,1],
      padding="SAME")
D_conv3 = tf.contrib.layers.batch_norm(D_conv3,
      is_training=is_training)
D_conv3 = lrelu(D_conv3)

D_W4 = weight_variables("D_W4", [4, 4, 4, 256, 512])
D_conv4 = tf.nn.conv3d(D_conv3, D_W4, strides=[1,2,2,2,1],
      padding="SAME")
```

```
D_conv4 = tf.contrib.layers.batch_norm(D_conv4,
      is_training=is_training)
D_conv4 = lrelu(D_conv4)


D_W5 = weight_variables("D_W5", [4, 4, 4, 512, 1])
D_conv5 = tf.nn.conv3d(D_conv4, D_W5, strides=[1,1,1,1,1],
      padding="VALID")
raw_logit = D_conv5
```

The function which creates the generator model has the following signature: it takes a random noise vector *z* as an input and returns an upscaled version which is a novel 3D object.

```
def generator_model(z, is_training=True):
      with tf.variable_scope("generator", reuse=tf.AUTO_REUSE):
      # ...
      return output
```

We first reshape the noise vector *z* into the following tensor. By using -1, the mini-batch size will automatically be inferred, so there is no need to pass it as a function argument for the generator model.

```
z_rs = tf.reshape(z, (-1, 1, 1, 1, z_size))
```

For the actual implementation of the generator model, we create 5 volumetric, batch-normalized transposed convolutional layers for upsampling, each followed by an additional ReLU layer. In the first layer, we use 3D transposed convolution with a stride of 1 in each dimension and no padding. In the next layers, we use strides of 2 in each dimension, zero-padding and no additional ReLU activation. Instead, we use a tanh activation function with values between -1 and 1, although a sigmoid function could be used as well, but is not recommended since it has values between 0 and 1 which clashes with the normalization of the data between -1 and 1. At the end, we receive a mini-batch of 64x64x64 3D voxel models. As far as the choice of the filter stack in each layer goes, we use the following setup:

- for transposed convolutional layer 1 we use 512 4x4x4 filters on the 200 input components to create an output volume over the whole mini-batch size of 512 4x4x4 feature maps,
- for transposed convolutional layer 2 we use 256 4x4x4 filters on 512 input feature maps to create an output volume over the whole mini-batch size of 256 4x4x4 feature maps,
- for transposed convolutional layer 3 we use 128 4x4x4 filters on 256 input feature maps to create an output volume over the whole mini-batch size of 128 4x4x4 feature maps,
- for transposed convolutional layer 4 we use 64 4x4x4 filters on 128 input feature maps to create an output volume over the whole mini-batch size of 64 4x4x4 feature maps,
- and for transposed convolutional layer 5 we use 1 4x4x4 filter on 64 input feature maps to create a whole mini-batch of 64x64x64 3D voxel models.

```
G_W1 = weight_variables("G_W1", [4, 4, 4, 512, 200])
G_deconv1 = tf.nn.conv3d_transpose(z_rs, G_W1,
      (-1,4,4,4,512), strides=[1,1,1,1,1], padding="VALID")
G_deconv1 = tf.contrib.layers.batch_norm(G_deconv1,
      is_training=is_training)
G_deconv1 = tf.nn.relu(G_deconv1)

G_W2 = weight_variables("G_W2", [4, 4, 4, 256, 512])
G_deconv2 = tf.nn.conv3d_transpose(G_deconv1, G_W2,
      (-1,8,8,8,256), strides=[1,2,2,2,1], padding="SAME")
G_deconv2 = tf.contrib.layers.batch_norm(G_deconv2,
      is_training=is_training)
G_deconv2 = tf.nn.relu(G_deconv2)

G_W3 = weight_variables("G_W3", [4, 4, 4, 128, 256])
G_deconv3 = tf.nn.conv3d_transpose(G_deconv2, G_W3,
      (-1,16,16,16,128), strides=[1,2,2,2,1], padding="SAME")
G_deconv3 = tf.contrib.layers.batch_norm(G_deconv3,
      is_training=is_training)
G_deconv3 = tf.nn.relu(G_deconv3)

G_W4 = weight_variables("G_W4", [4, 4, 4, 64, 128])
G_deconv4 = tf.nn.conv3d_transpose(G_deconv3, G_W4,
      (-1,32,32,32,64), strides=[1,2,2,2,1], padding="SAME")
G_deconv4 = tf.contrib.layers.batch_norm(G_deconv4,
      is_training=is_training)
G_deconv4 = tf.nn.relu(G_deconv4)

G_W5 = weight_variables("G_W5", [4, 4, 4, 1, 64])
G_deconv5 = tf.nn.conv3d_transpose(G_deconv4, G_W5,
      (-1,64,64,64,1), strides=[1,2,2,2,1], padding="SAME")
output = tf.nn.tanh(G_deconv5)
```

To set up the computational graph, we first define the placeholders in which we load the data and the random noise vector respectively. We allocate for a whole mini-batch at a time. In our setup, we allocate for a mini-batch of 64x64x64 voxel models, with one input channel (no typical color), and a mini-batch of 200-dimensional noise vectors. We also set up our generator and discriminator models. Weights are shared between the discriminator models due to automatic reuse. So essentially we only have one generator model and one discriminator model.

```
cs = cubic_size
X = tf.placeholder(tf.float32, shape=[batch_size, cs, cs, cs, 1])
Z = tf.placeholder(tf.float32, shape=[batch_size, z_size])

G_sample = generator_model(Z, is_training=True)
D_output_x = discriminator_model(X, is_training=True)
D_output_z = discriminator_model(G_sample, is_training=True)
```

Next, we compute the discriminator and generator loss using cross entropy, which will be optimized by the Adam optimizer given an initial learning rate and a beta1 parameter, which is the exponential decay rate for the first moment estimates. Here, everything is identical to the other GAN models discussed earlier which use cross entropy.

```
D_loss = tf.reduce_mean(
tf.nn.sigmoid_cross_entropy_with_logits(logits=D_output_x,
      labels=tf.ones_like(D_output_x)) +
tf.nn.sigmoid_cross_entropy_with_logits(logits=D_output_z,
      labels=tf.zeros_like(D_output_z)))

G_loss = tf.reduce_mean(
tf.nn.sigmoid_cross_entropy_with_logits(logits=D_output_z,
      labels=tf.ones_like(d_output_z)))

T_params = tf.trainable_variables()
D_params = [v for v in T_params if v.name.startswith("discriminator")]
G_params = [v for v in T_params if v.name.startswith("generator")]

D_optimizer =
      tf.train.AdamOptimizer(learning_rate=D_learning_rate,beta1=beta1)
      .minimize(D_loss,v ar_list=D_params)

G_optimizer =
      tf.train.AdamOptimizer(learning_rate=G_learning_rate,beta1=beta1)
      .minimize(G_loss, var_list=G_params)
```

Next, we need some code for reading and writing data. To load our 3D voxelized volumes we open the .mat MATLAB files from ShapeNet and convert them to a voxel format, i.e. a 3D array of boolean values. Since the data from ShapeNet contains 32x32x32 voxel models, we upsample to 64x64x64, i.e. by a factor of 2 in each dimension. Volumes are padded or surrounded by zeros as well.

```
import scipy.io as io
import numpy as np
import os

dataset_path = "../data/"
results_path = "../results/"
object_type = "chair"

volumes = load_data(object_type=object_type, is_training=True,
      cubic_size=cubic_size, keep_fraction=0.70)
volumes = volumes[..., np.newaxis]
volume_count = len(volumes)

if not os.path.exists(results_path):
      os.makedirs(results_path)

def matlab_to_voxels(path, cubic_size=64):
      vox = io.loadmat(path)["instance"]
      width = 1
      vox = np.pad(vox,(width,width),"constant",constant_values=(0,0))
      if cube_len != 32:
            f = cubic_size / 32.0
            vox = nd.zoom(voxels, (f,f,f), mode='constant', order=0)
      return vox

def load_data(object_type="chair", is_training=True,
      cubic_size=64, keep_fraction=0.75):
      path = dataset_path + object_type + "/30/"
      if is_training:
            path += "train/"
      else:
            path += "test/"
      files = [file for file in os.listdir(path)
                  if file.endswith(".mat")]
      object_count = len(files)
      indexes = 0:int(object_count*keep_fraction)
      kept_files = files[indexes]
```

```
        return np.asarray(
              [matlab_to_voxels(path + file, cubic_size)
                    for file in kept_files],
              dtype=np.bool)
                    .astype(np.float)
```

For each training epoch we pick a random set of indexes for the mini-batch and put the mini-batch in placeholder x.

```
vis = visdom.Visdom()
sess = tf.Session()
sess.run(tf.global_variables_initializer())

for epoch in range(epoch_count):
      indexes = np.random.randint(volume_count, size=batch_size)
      x = volumes[indexes]
      z = np.random.normal(0.0, 0.2, size=[batch_size, z_size])

      _, D_loss_curr = sess.run([D_solver, D_loss], {Z: z, X: x})
      _, G_loss_curr = sess.run([G_solver, G_loss], {Z: z})

      # Plotting of generated 3D models
```

Plotting of generated models can be done by using our `plot_voxels()`function. Novel models can be generated by using a trained generator for testing.

```
batch_count = 20
Z = tf.placeholder(tf.float32, shape=[batch_size, z_size])
G_sample = generator_model(Z, is_training=False)

vis = visdom.Visdom()
sess = tf.Session()
sess.run(tf.global_variables_initializer())

for i in range(batch_count):
      z = np.random.normal(0, 0.2, size=[batch_size, z_size])
      samples = sess.run(G_sample, feed_dict={Z: z)})
      indices = np.random.randint(0, batch_size, 4)
      for i in range(4):
            plot_voxels(np.squeeze(samples[indices[i]] > 0.5), vis)
```

Voxels can be plotted as a polygonal mesh as follows:

```
def plot_voxels(voxels, vis, title = ""):
      v,f = voxels_to_polygons(voxels)
      vis.mesh(X=v, Y=f, opts=dict(opacity=0.7, title=title))
```

Here the conversion from voxels to a polygonal mesh of vertices and faces is done by the marching cubes algorithm, using the skimage library:

```
def voxels_to_polygons(voxels, level=0.5):
      v,f =  skimage.measure.marching_cubes(voxels, level=level)
      return v,f
```

Summarized, we observe the following improvements over the standard implementation:

- When using a maximization instead of a minimization for the logarithmic loss of the generator, we encounter less problems with vanishing gradients and better convergence. For the logarithmic definition this means an optimization of $max(\log(D(G(z))))$ instead of $min(\log(1 - D(G(z)))$.

  Instead, we sticked to the cross-entropy formulation discussed in the previous GAN model sections, since the results can be observed as being identical.
- Sampling noise vectors from a normal distribution instead of a uniform distribution yields a faster convergence and higher quality results for the same amount of epochs.
- Use of a formulation of the loss function in terms of least squares as in LSGAN ensures even faster convergence and increased stability. The WGAN formulation yields a slower convergence, but solves mode collapse and vanishing gradients.
- In order to increase performance of the discriminator to an accuracy of 1.0, one can use an initial learning rate of 0.001 instead of 0.00001, however the generator can suffer from its inability to keep up with the discriminator.

Chapter 11: Results

11.1 Experiment 1: Benchmarks convolution

In our first experiment, we compare the execution speed of both a regular Euclidean convolution and the convolution used in Mixture Model CNNs. For the Euclidean convolution we build our computational graph as follows:

- We declare and allocate a constant 4th order input tensor *X*. The axes correspond to: the mini-batch, width, height and number of features which we will all set to 50. Therefore, we will consider a basic 2D Euclidean convolution. 5th order tensors can be used for a 3D convolution.
- We will also allocate the variable tensor for the weights called *w*.
- The computational graph consists of the input layer and 5 tf.nn.conv2d() layers with strides of 1 in each dimension with zero-padding as well.
- We run this computational graph for a total of 100 epochs where we measure the execution time in each epoch.
- At the end we compute the average execution time and the standard deviation of these results.

On an NVidia GeForce GTX 970 graphics card we achieve an average execution time of 0.000784 seconds per epoch with a standard deviation of 0.00000392 for 2D Euclidean convolution in TensorFlow. This high execution speed on the GPU is due to the regular grid of an Euclidean domain.

Next, we define our computational graph for the implementation of 2D geometric convolution as described by Mixture Model CNNs as follows:

- First, we build a sparse adjacency matrix having a grid connectivity to resemble a Euclidean grid. The grid has the same hyperparameters and measurements as the Euclean one above.
- Next, we start building the computational graph by building a list of geodesic sparse weight matrices for every mini-batch.
- Finally, we run 5 successive geodesic convolutions as described in the implementation chapter given our input features and the list of geodesic sparse weight matrices for every mini-batch.

On an NVidia GeForce GTX 970 graphics card we achieve an average execution time of 1.343968 seconds per epoch with a standard deviation of 0.0075262 for 2D geometric convolution in TensorFlow as used in Mixture Model CNNs. This is significantly slower than the Euclidean variant. Apparently, it is not possible for TensorFlow to utilise the GPU for certain sparse matrix operations. These operations correspond to the creation of the sparse geometric weight matrices used in Mixture Model CNNs. This way, we can conclude that better results can be obtained when this implementation of geometric convolution utilizes a sparse GPU implementation.

11.2 Experiment 2: 3D-GAN stabilization

Since the training of GANs is known in literature to be a highly difficult problem due to instabilities such as mode collapse, vanishing gradients or difficulties reaching a Nash equilibrium, in this section we try to improve the standard architecture of 3D-GANs with regards to higher stability and faster training convergence.

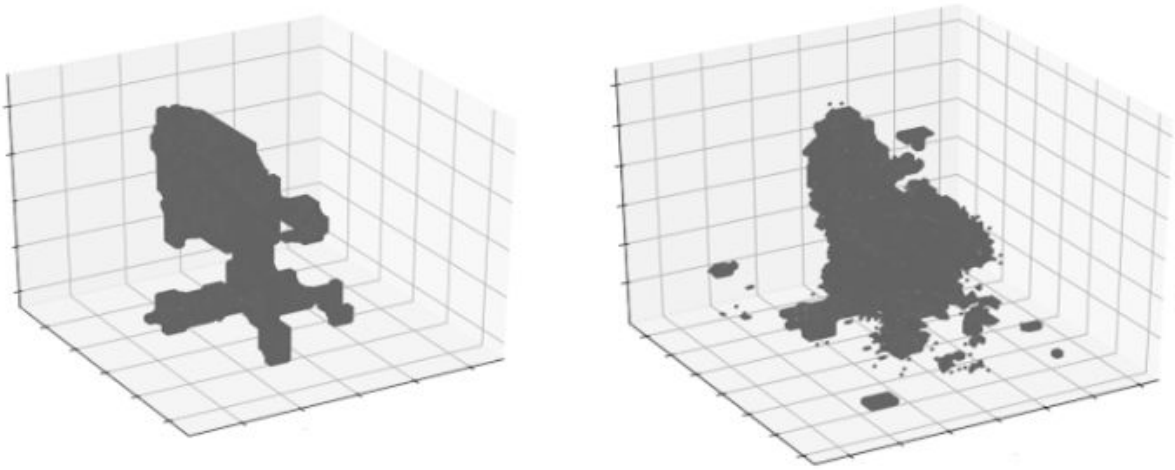

Figure 11.1: Using a standard implementation for 3D-GAN can sometimes cause instabilities at an unpredictable pace, most likely depending on the inputs. At the left we see a generated desk chair after 2000 epochs. At the right we see mode collapse appearing after 5000 epochs.

Vanishing gradients also seem to occur sometimes, depending on the picked mini-batches for the input. We see that the generator no longer improves at a non-optimal content generation stage due to the discriminator being to good compared to the generator.

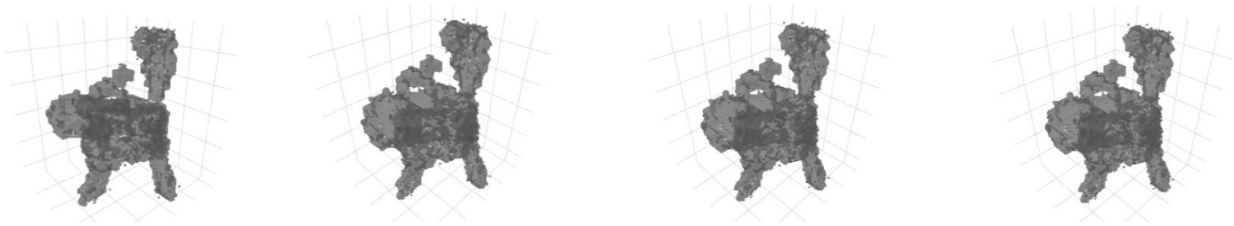

Figure 11.2: Vanishing gradients can occur as well. From left to right we see a generated chair-like object after 2000, 3000, 4000 and 5000 epochs. This is clearly a case of a vanishing gradient, where the generator prematurely cannot improve itself anymore, because the discriminator is outperforming the generator.

When using a maximization instead of a minimization for the logarithmic loss of the generator, we encounter less problems with vanishing gradients, less chances of mode collapse and better convergence. For the logarithmic definition this means an optimization of $max(log(D(G(z))))$ instead of $min(log(1 - D(G(z)))$. Instead, we sticked to the cross-entropy formulation of this maximisation as discussed in the previous GAN model sections, since the results can be observed as being identical.

Use of a formulation of the loss function in terms of least squares as in LSGAN ensures even faster convergence and increased stability and avoidance of mode collapse. The WGAN formulation yields a slower convergence, but is still more stable than the standard formulation of the loss. Since both the LSGAN and WGAN formulations solve mode collapse and instabilities to a large extent and LSGAN being faster and more intuitive, we stick to the LSGAN formulation.
Faster training can be achieved by replacing the standard loss function of a 3D-GAN by its least squares formulation used in LSGANs, this way faster convergence to the equilibrium state of optimal content generation can be achieved. The use of a Wasserstein distance is inferior to the least squares formulation in terms of runtime performance.

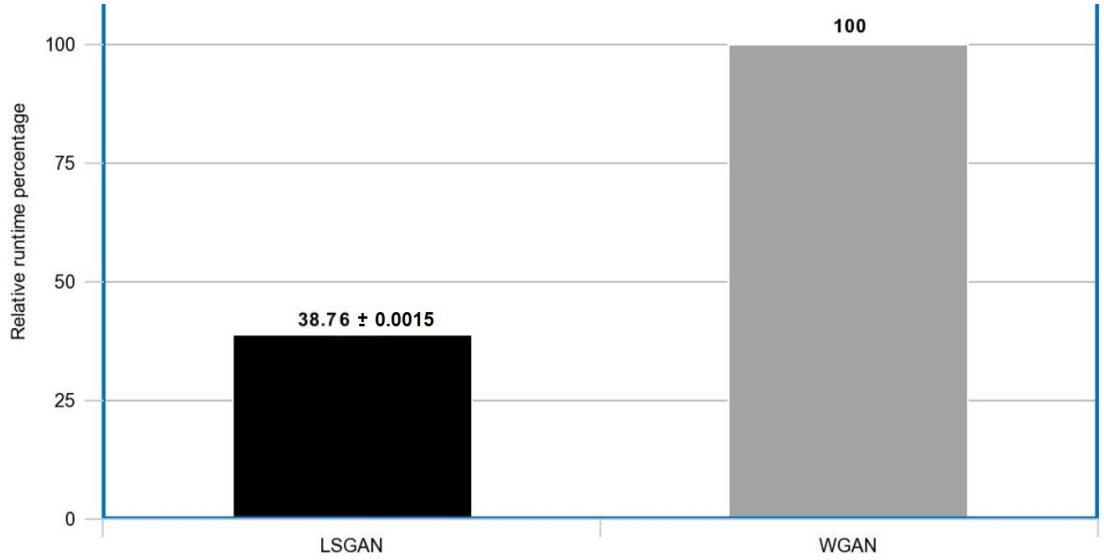

Figure 11.3: Relative runtime percentage comparison between a baseline of the Wasserstein GAN (WGAN) implementation and the LSGAN. The WGAN is a baseline of 100% runtime, while the LSGAN only occupies 38.76 ± 0.0015% of the runtime. This is tested on our 3D-GAN for chair models on multiple mini-batches.

We have also seen that improved techniques for GAN training such as input normalization or more general batch normalization, sampling noise from Gaussian distributions instead of uniform ones for faster convergence and the use of a dropout of 50% in the generator cause results of higher quality. As implementation details might differ between different frameworks, it is wise to sometimes implement custom layers, so we know exactly what to expect. For example, we had to replace tf.contrib.layers.batch_norm with tf.layers.batch_normalization for better training results. We do not reduce on the axis for the mini-batch, i.e. axis 0. This function has the following correct implementation:

```
def batch_normalization(x, gamma, beta, epsilon=1e-8):
      mean = tf.reduce_mean(x, [1,2,3,4])
      stdev = tf.reduce_mean(tf.square(x - mean), [1,2,3,4])
      x = (x - mean) / tf.sqrt(stdev + epsilon)
      gamma = tf.reshape(gamma, [-1,1,1,1,1])
      beta = tf.reshape(beta, [-1,1,1,1,1])
      return gamma*x + beta
```

Here gamma is a 5th order tensor of learnable weights initialized to 1 and beta is a 5th order tensor of learnable biases initialized to 0. In order to increase performance of the discriminator to an accuracy of 1.0, one can theoretically use an initial learning rate of 0.001 instead of 0.00001, however the generator can suffer from its inability to keep up with the discriminator, so we will not use such an adaption.

As post-processing an algorithm such as marching cubes can be used for converting voxels to polygons. Hyperparameter optimization can later be applied for better results, yet is expected to take up several days. Now we can see better results.

Figure 11.4: After 4000 epochs we can already see more clear results. Instabilities are rare now and convergence speed has improved as well. Artefacts at the edges are unavoidable, since marching cubes causes a pixelated effect.

11.3 Experiment 3: 3D generator without deconvolution

When implementing the graph or manifold generalization provided by Mixture Model CNNs, we noticed that there is no theoretical equivalent formulation of upsampling (deconvolution). Since our introduction to GANs, we have seen that it is possible to construct a generator which does not consist of deconvolutional layers. Instead we can create a generator of fully or non-fully connected layers. Since convolution is still defined in this framework, we can keep our convolutional layers in the discriminator. Since the discriminator functions as a supervisor for the generator and delivering direct feedback so the generator can improve itself, the generator still will have a supervisor which has a notion of translational invariance and high-resolution analyses skills due to the convolutional layers in the discriminator. However, the generator is now in a clear disadvantage compared to the discriminator. Since it has no notion of translational invariance, it will produce inferior results. Also, since we now have a supervisor which is too harsh on an underperforming generator, the generator itself is not capable enough to correct itself in the right way, so more epochs are necessary and its improved results are more akin to luck. While setting the initial learning rate of the discriminator a bit lower than the initial learning rate of the generator might help in the beginning, the discriminator will catch-up before the generator has a chance to produce reasonable results; the generator always has a clear disadvantage.

It is also hard to find an equivalent vanilla neural network architecture which emulates multiple deconvolutional layers. When one considers the node count of each hidden layer, the amount of hidden layers and their connectivity as hyperparameters, one could in theory employ a technique such as Bayesian Hyperparameter Optimization or a random search. Yet, in practice this can take up to days and will still be inferior to the actual use of deconvolutional layers if they existed in a purely polygonal learning setting. In figure 11.2 a typical result for such an underperforming generator is shown.

While one could argue that the use of regular 2D Euclidean convolution on projected images such as Orbifold Tutte Embeddings [72, 73], geometry images [74] or the use of convolutional neural networks via an embedding of a surface on a torus [75], these are actually parameterizations of 3D surfaces on a 2D plane. The downside of these methods is that they create inevitable distortions. From differential geometry it is known that for general surfaces there exists no isometric (distance preserving) parameterization in the plane. These parameterizations only exist for smooth surfaces of zero Gaussian curvature, which means they are developable. Distortion can be reduced by division of surfaces in smaller charts, yet it is inevitable. Distortions can either be angular, stretching, shearing, length or area distortion and different trade-offs can be made between them, yet distortion avoidance can cause an increased cost in performance. Furthermore, these distortions make it hard to perform decent reconstructions from 2D back to 3D in a deconvolutional layer and the transformations are not always necessarily invertible. Still, trade-offs between angle preservation or area preservation need to be made, but even when a certain object-to-plane transformation is invertible, the result will still involve distortions. We can in fact tolerate a small amount of distortion in some applications, but it still requires significant effort to find a trade-off between each kind of distortion for optimal content generation.

Therefore we can conclude the current best alternative is to stick to volumetric 3D-GANs used on voxel models of the highest possible resolution for more detail. Voxel grids tend to be sparsely filled with actual object content and most values are zero, so large quantities of VRAM are wasted on these empty cells. Therefore, one can make a trade-off between memory consumption and runtime performance. To reduce memory usage one can use sparse matrices in TensorFlow at the expense of faster training when the voxel grid might not be sparse enough, although most of the time when sparsity is so that we have more than 75% zeros , we can gain a significant speed-wise performance gain as well.

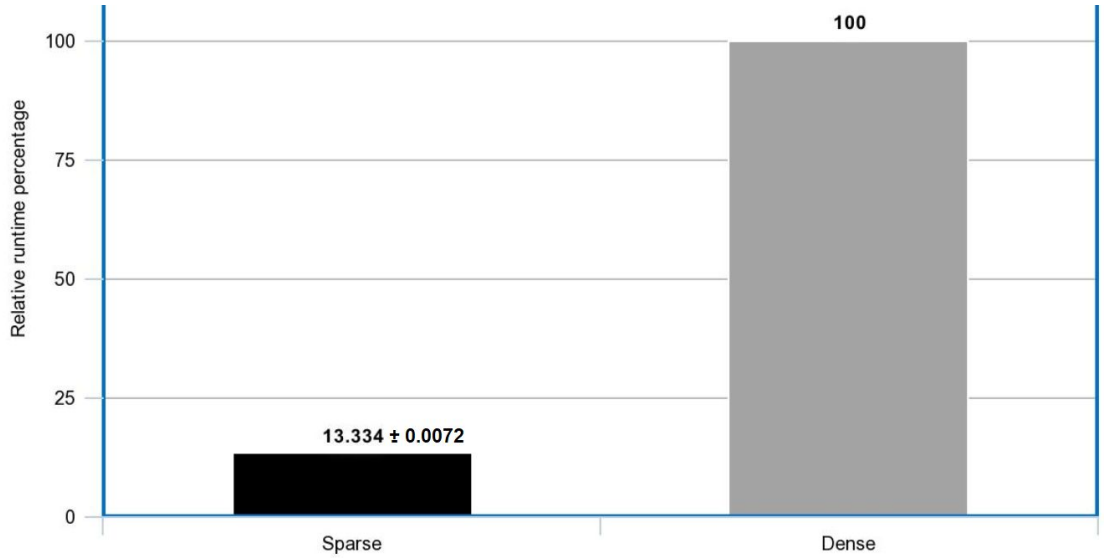

Figure 11.5: Relative runtime percentage comparison between a baseline of dense matrix operations occupying a runtime of 100% and its sparse competitor, when dealing with a sparsity where around 77.25 ± 0.002% of the values in the grid are zero depending on the chosen mini-batches. For sparse matrix operations this is 13.334 ± 0.0072%. This is tested on our 3D-GAN for chair models on multiple mini-batches.

Conclusion

In this thesis we have seen that state-of-the-art 3D-GANs developed for volumetric data can be modified to generate desirable 3D polygonal objects from an existing set of example objects. We also addressed shortcomings and possible future improvements. We have seen that we need a full generalization of convolution and deconvolution on non-Euclidean data such as 3D polygonal objects. Current state-of-the-art non-Euclidean convolutional layers are only defined for the actual convolution, i.e. the downsampling case. Upsampling or deconvolutional operations, which are essential for content generation, are currently not defined in the theory of geometric deep learning. Hence there is a need for novel generalized convolutional and deconvolutional operations, which contain weights that can be learned in both directions of a generative adversarial setting, as it is necessary for the expansion of voxel-based 3D-GANs to a fully polygonal learning solution. These up- and downsampling operations should be implemented in a GPU friendly manner, while still capturing the semantics of its input data. The earlier non-Euclidean convolution generalizations require sparse linear algebra and algorithms on graphs, which are known to be difficult to map on massively parallel SIMD style computing architectures such as GPUs. As a result as we have seen in our experiments, the runtime performance of these techniques is over a large order of magnitude lower in practice than standard CNN methods on Euclidean data. Even though GPU architectures are evolving, they still excel at dense linear algebra and grid-based data structures and algorithms, and will remain doing so in the foreseeable future. There is a need for faster techniques which are more fine-tuned to the nature of 3D polygonal meshes.

The current best alternative is to stick to volumetric 3D convolutional and deconvolutional layers used on a voxel model of the highest possible resolution for more detail. Voxel grids tend to be sparsely filled with actual object content, so large quantities of VRAM are wasted. To reduce memory usage one can use sparse matrices and computations happen faster in sparsity. Also smaller mini-batches can be used in spite of more processing time. Faster training can be achieved by replacing the standard loss function of a 3D-GAN by its least squares formulation used in LSGANs, this way faster convergence to the equilibrium state of optimal content generation can be achieved. The use of a Wasserstein distance is inferior to the least squares formulation in terms of runtime performance. We have also seen that improved techniques for GAN training such as input normalization or more general batch normalization, sampling from Gaussian distributions instead of uniform ones and the use of a dropout of 50% in the generator cause better results. As post-processing an algorithm such as marching cubes can be used for converting voxels to polygons. As implementation details might differ between different frameworks, it is wise to sometimes implement custom layers, so we know exactly what to expect. Hyperparameter optimization can later be applied for better results, yet is expected to take up several days.
A future direction of research is the development of approximation algorithms, operating on GPU-friendly data structures (i.e. mostly array- or grid-based contiguous data), leading to better hardware utilization. An optimal trade-off between mathematical correctness and efficient training and testing will need to be established. The immediate benefit in applications will consist of highly reduced model training and trained model application costs in science and industry, even beyond the field of computer science.

References

[1] Karn, U. (2016, August 9). A Quick Introduction to Neural Networks. Retrieved from https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/

[2] Support vector machine. (2018, August 13). Retrieved from https://en.wikipedia.org/wiki/Support_vector_machine

[3] Decision tree. (2018, August 16). Retrieved from https://en.wikipedia.org/wiki/Decision_tree

[4] Naive Bayes classifier. (2018, July 24). Retrieved from https://en.wikipedia.org/wiki/Naive_Bayes_classifier

[5] Curse of dimensionality. (2018, July 23). Retrieved from https://en.wikipedia.org/wiki/Curse_of_dimensionality

[6] Softmax function. (2018, August 13). Retrieved from https://en.wikipedia.org/wiki/Softmax_function

[7] TensorFlow. (2018, August 2). Retrieved from https://www.tensorflow.org/

[8] Sigmoid function. (2018, August 13). Retrieved from https://en.wikipedia.org/wiki/Sigmoid_function

[9] Cross entropy. (2018, August 15). Retrieved from https://en.wikipedia.org/wiki/Cross_entropy

[10] One-hot. (2018, May 22). Retrieved from https://en.wikipedia.org/wiki/One-hot

[11] Feedforward neural network. (2018, July 8). Retrieved from https://en.wikipedia.org/wiki/Feedforward_neural_network

[12] Recurrent neural network. (2018, August 9). Retrieved from https://en.wikipedia.org/wiki/Recurrent_neural_network

[13] CS231n Convolutional Neural Networks for Visual Recognition. (2018, August 5). Retrieved from http://cs231n.github.io/neural-networks-1/

[14] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097-1105).

[15] Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013, June). Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml* (Vol. 30, No. 1, p. 3).

[16] Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., & Bengio, Y. (2013). Maxout networks. *arXiv preprint arXiv:1302.4389*.

[17] Intuitive understanding of backpropagation. (2018, August 5). Retrieved from http://cs231n.github.io/optimization-2/#intuitive

[18] Gradient descent. (2018, August 5). Retrieved from http://cs231n.github.io/optimization-1/#gd

[19] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE, 86*(11), 2278-2324.

[20] Convolutional Neural Networks (CNNs / ConvNets). (2018, July 20). Retrieved from http://cs231n.github.io/convolutional-networks/

[21] Hyperparameter optimization. (2018, July 19). Retrieved from https://en.wikipedia.org/wiki/Hyperparameter_optimization

[22] Hyperparameter optimization. (2018, July 20). Retrieved from http://cs231n.github.io/neural-networks-3/#hyper

[23] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... & Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems* (pp. 2672-2680).

[24] Yu, F., & Koltun, V. (2015). Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*.

[25] Kingma, D. P., & Welling, M. (2013). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.

[26] Zeiler, M. D., & Fergus, R. (2014, September). Visualizing and understanding convolutional networks. In *European conference on computer vision* (pp. 818-833). Springer, Cham.

[27] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... & Rabinovich, A. (2014). Going deeper with convolutions, CoRR abs/1409.4842. *URL http://arxiv. org/abs/1409.4842*.

[28] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[29] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).

[30] Ng, A. [Andrew Ng] (2017, August 25). Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization [Video file]. Retrieved from https://www.youtube.com/watch?v=1waHlpKiNyY&list=PLkDaE6sCZn6Hn0vK8co82zjQtt3T2Nkqc

[31] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research, 15*(1), 1929-1958.

[32] Mishkin, D., & Matas, J. (2015). All you need is a good init. *arXiv preprint arXiv:1511.06422*.

[33] Weight initialization. (2018, May 24). Retrieved from http://cs231n.github.io/neural-networks-2/#init

[34] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision* (pp. 1026-1034).

[35] Glorot, X., & Bengio, Y. (2010, March). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (pp. 249-256).

[36] Schaul, T., Antonoglou, I., & Silver, D. (2013). Unit tests for stochastic optimization. *arXiv preprint arXiv:1312.6055*.

[37] Wilson, A. C., Roelofs, R., Stern, M., Srebro, N., & Recht, B. (2017). The marginal value of adaptive gradient methods in machine learning. In *Advances in Neural Information Processing Systems* (pp. 4148-4158).

[38] Kingma, Diederik P and Ba, Jimmy Lei. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[39] Arjovsky, M., Chintala, S., & Bottou, L. (2017). Wasserstein GAN. *arXiv preprint arXiv:1701.07875*.

[40] Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research, 12*(Jul), 2121-2159.

[41] Zeiler, M. D. (2012). ADADELTA: An Adaptive Learning Rate Method. Retrieved from http://arxiv.org/abs/1212.5701

[42] Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural networks, 12*(1), 145-151.

[43] Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence O (1/k^ 2). In *Doklady AN USSR* (Vol. 269, pp. 543-547).

[44] Bengio, Y., Louradour, J., Collobert, R., & Weston, J. (2009, June). Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning* (pp. 41-48). ACM.

[45] Khan, F., Mutlu, B., & Zhu, X. (2011). How do humans teach: On curriculum learning and teaching dimension. In *Advances in Neural Information Processing Systems* (pp. 1449-1457).

[46] Basu, S., & Christensen, J. (2013, July). Teaching Classification Boundaries to Humans. In *AAAI*.

[47] Bergstra, J. S., Bardenet, R., Bengio, Y., & Kégl, B. (2011). Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems* (pp. 2546-2554).

[48] Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research, 13*(Feb), 281-305.

[49] Brochu, E., Cora, V. M., & De Freitas, N. (2010). A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*.

[50] Breiman, L. (2001). Random forests. *Machine learning, 45*(1), 5-32.

[51] Matérn covariance function. (2018, July 21). Retrieved from https://en.wikipedia.org/wiki/Mat%C3%A9rn_covariance_function

[52] Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

[53] Zhou, Z. H. (2009, June). When semi-supervised learning meets ensemble learning. In *International Workshop on Multiple Classifier Systems* (pp. 529-538). Springer, Berlin, Heidelberg.

[54] Salakhutdinov, R., & Larochelle, H. (2010, March). Efficient learning of deep Boltzmann machines. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (pp. 693-700).

[55] Hinton, G. E., Osindero, S., & Teh, Y. W. (2006). A fast learning algorithm for deep belief nets. *Neural computation, 18*(7), 1527-1554.

[56] Kullback-Leibler divergence. (2018, July 12). Retrieved from https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence

[57] Jensen–Shannon divergence. (2018, July 12). Retrieved from https://en.wikipedia.org/wiki/Jensen%E2%80%93Shannon_divergence

[58] Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., & Chen, X. (2016). Improved techniques for training GANs. In *Advances in Neural Information Processing Systems* (pp. 2234-2242).

[59] Weng, L. (2017, August 20). From GAN to WGAN. Retrieved from https://lilianweng.github.io/lil-log/2017/08/20/from-GAN-to-WGAN.html

[60] Arjovsky, M., & Bottou, L. (2017). Towards principled methods for training generative adversarial networks. *arXiv preprint arXiv:1701.04862*.

[61] Manifold learning. (2017). Retrieved from http://scikit-learn.org/stable/modules/manifold.html

[62] Goodfellow, I. (2016). NIPS 2016 tutorial: Generative adversarial networks. *arXiv preprint arXiv:1701.00160.*

[63] Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., & Courville, A. C. (2017). Improved training of Wasserstein GANs. In *Advances in Neural Information Processing Systems* (pp. 5767-5777).

[64] Mao, X., Li, Q., Xie, H., Lau, R. Y., Wang, Z., & Smolley, S. P. (2017, October). Least squares generative adversarial networks. In *Computer Vision (ICCV), 2017 IEEE International Conference on* (pp. 2813-2821). IEEE.

[65] Radford, A., Metz, L., & Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*.

[66] How to Train a GAN? Tips and tricks to make GANs work. (2016). Retrieved from https://github.com/soumith/ganhacks

[67] Wu, J., Zhang, C., Xue, T., Freeman, B., & Tenenbaum, J. (2016). Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling. In *Advances in Neural Information Processing Systems* (pp. 82-90).

[68] ShapeNet. (2018, April 14) Retrieved from https://www.shapenet.org/

[69] Lorensen, W. E., & Cline, H. E. (1987, August). Marching cubes: A high resolution 3D surface construction algorithm. In *ACM siggraph computer graphics* (Vol. 21, No. 4, pp. 163-169). ACM.

[70] Voxel to Mesh Conversion: Marching Cube Algorithm. (2017, July 19). Retrieved from https://medium.com/zeg-ai/voxel-to-mesh-conversion-marching-cube-algorithm-43dbb0801359

[71] Lee, D. T., & Schachter, B. J. (1980). Two algorithms for constructing a Delaunay triangulation. *International Journal of Computer & Information Sciences, 9*(3), 219-242.

[72] Aigerman, N., & Lipman, Y. (2015). Orbifold Tutte embeddings. *ACM Trans. Graph., 34*(6), 190-1.

[73] Aigerman, N., Kovalsky, S. Z., & Lipman, Y. (2017). Spherical orbifold Tutte embeddings. *ACM Transactions on Graphics (TOG), 36*(4), 90.

[74] Gu, X., Gortler, S. J., & Hoppe, H. (2002). Geometry images. *ACM Transactions on Graphics (TOG), 21*(3), 355-361.

[75] Maron, H., Galun, M., Aigerman, N., Trope, M., Dym, N., Yumer, E., ... & Lipman, Y. (2017). Convolutional neural networks on surfaces via seamless toric covers. *ACM Trans. Graph, 36*(4), 71.

[76] Bronstein, M. M., Bruna, J., LeCun, Y., Szlam, A., & Vandergheynst, P. (2017). Geometric deep learning: going beyond Euclidean data. *IEEE Signal Processing Magazine, 34*(4), 18-42.p. 1). ACM.

[77] Monti, F., Boscaini, D., Masci, J., Rodola, E., Svoboda, J., & Bronstein, M. M. (2017, July). Geometric deep learning on graphs and manifolds using Mixture Model CNNs. In *Proc. CVPR* (Vol. 1, No. 2, p. 3).

[78] Adjacency matrix. (2018, May 29). Retrieved from https://en.wikipedia.org/wiki/Adjacency_matrix