Generation of Synthetic Images with Generative Adversarial Networks

Degree Project for Master of Science in Computer Science and Engineering

by

Mousa Zeid Baker

Supervisor: Professor Håkan Grahn
Blekinge Institute of Technology
Department of Computer Science and Engineering

January 2018
SE-371 79 Karlskrona, Sweden

Abstract

Machine Learning is a fast growing area that revolutionizes computer programs by providing systems with the ability to automatically learn and improve from experience. In most cases, the training process begins with extracting patterns from data. The data is a key factor for machine learning algorithms, without data the algorithms will not work. Thus, having sufficient and relevant data is crucial for the performance.

In this thesis, the researcher tackles the problem of not having a sufficient dataset, in terms of the number of training examples, for an image classification task. The idea is to use Generative Adversarial Networks to generate synthetic images similar to the ground truth, and in this way expand a dataset. Two types of experiments were conducted: the first was used to fine-tune a Deep Convolutional Generative Adversarial Network for a specific dataset, while the second experiment was used to analyze how synthetic data examples affect the accuracy of a Convolutional Neural Network in a classification task. Three well known datasets were used in the first experiment, namely MNIST, Fashion-MNIST and Flower photos, while two datasets were used in the second experiment: MNIST and Fashion-MNIST.

The results of the generated images of MNIST and Fashion-MNIST had good overall quality. Some classes had clear visual errors while others were indistinguishable from ground truth examples. When it comes to the Flower photos, the generated images suffered from poor visual quality. One can easily tell the synthetic images from the real ones. One reason for the bad performance is due to the large quantity of noise in the Flower photos dataset. This made it difficult for the model to spot the important features of the flowers.

The results from the second experiment show that the accuracy does not increase when the two datasets, MNIST and Fashion-MNIST, are expanded with synthetic images. This is not because the generated images had bad visual quality, but because the accuracy turned out to not be highly dependent on the number of training examples.

It can be concluded that Deep Convolutional Generative Adversarial Networks are capable of generating synthetic images similar to the ground truth and thus can be used to expand a dataset. However, this approach does not completely solve the initial problem of not having adequate datasets because Deep Convolutional Generative Adversarial Networks may themselves require, depending on the dataset, a large quantity of training examples.

Keywords: classification, deep learning, generative adversarial network, machine learning

Preface

This thesis is submitted to the Department of Computer Science and Engineering at Blekinge Institute of Technology (BTH) in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering. The Degree Programme in Computer Science and Engineering is composed of 300 ECTS whereof the thesis is equivalent to 30 ECTS. This research was conducted in collaboration with *Sony Mobile Communications* in Lund.

I would like to extend my sincerest thanks and appreciation to the people who helped me accomplish this study. Firstly, I would like to gratitude many thanks to my supervisor from BTH Professor Håkan Grahn for his advices, support and guidance that contributed to the quality of this thesis. Secondly, I would like to thank Jim Rasmusson and Daniel Lönnblad from Sony Mobile Communications for their experience in Deep Learning and guidance during the research. A special thanks is also extended to Dr Abbas Cheddad from BTH for his feedback on the report, and to Stefan Petersson from BTH for providing me with an office and equipment. Lastly, I would like to thank friends and family for their support.

Mousa Zeid Baker

Karlskrona, January 2018

Table of Contents

List of Figures

List of Tables

1 INTRODUCTION

Within the last half-century, the computer has been the most successful invention in the world and it has not only changed our lives but also completely changed the world we live in. Computers and their uses have rapidly grew due to the huge potential they have to resolve problems that we encounter daily in our lives. They are deeply involved in almost every society and have contributed to the development of many areas such as communication, education, health care, utility facilities and many more. The changes that already have been made is with great certainty known by most people, it's more interesting to discuss the new changes computers can bring in the near future [1].

Computer scientists and inventors have long dreamed of creating machines that can think and become intelligent [2]. Artificial Intelligence (AI) is an active field that concerns this topic. It started when the nascent field of computer science started to ask if a computer could become intelligent or mimic cognitive abilities that lead to knowledge such as learning, problem solving and reasoning. In the beginning of the development of AI, software were hard-coded with knowledge about the world with a list of formal, mathematical rules. This approach never led to a major success due to the struggle of describing the complexity of the world with formal rules. Instead of relying on hard-coded knowledge, AI systems needed a capability to extract their own knowledge. Systems started to extract patterns from raw data, this capability come to be known as Machine Learning (ML) [3].

ML is a field with many different learning capabilities and it is still expanding. There are different types of learning problems, three popular types will be described here but the reader should know that they are not the only types. The first type is called **supervised learning**, a very well studied problem with many successful outcomes. Supervised learning is where all the data is labeled, that is when for every input variable (x), the output variable (y) is known. An algorithm learn to map the input to the output and since the output (correct answer) is known for every input, the algorithm is said to be supervised. Supervised learning can further be divided into two sections: the first is when the output is a real value, that is a **regression problem**, and the second is when the output variable is a category, that is a **classification problem**. For example, suppose a dataset with images of cats and dogs. Every image is either labeled with "cat" or "dog", an algorithm then learns to map each image to its corresponding label category, this is a classification problem [4, 1].

Another ML problem type is when only the input data (x) is known, this is refereed to **unsupervised learning**. The task here is to organize the data or to discover the structure or distribution of the data in order to learn more about it. Since there are no correct answers (y), algorithms work on their own. Unsupervised learning can further be divided into two sections: **clustering** and **association**. Clustering is used to discover the inherent groupings in the data while association is used to discover rules that describe the data [4, 1].

The last type is called semi-supervised machine learning and refers to problems where one part of the dataset is labeled and one part is unlabeled. This is very common because it is very expensive and time consuming to label big datasets. Suppose a classification problem where the data set is not fully labeled. Then unsupervised learning techniques can be used to discover the structure in the input variables. Or supervised learning techniques can be used to predict labels to every unlabeled x [4, 1].

One important factor regarding the performance of most ML algorithms is how the data is **represented**. A logistic regression algorithm used in, let's say health-care, for cesarean recommendations will not make useful predictions if it was given an magnetic resonance imaging (MRI) scan. The ML algorithm will not be able to extract useful features (information) from the MRI scan. Instead, a doctor can provide the ML algorithm with some relevant features about the patient. Many AI tasks can be solved if the correct set of features are extracted and then provided to an ML algorithm. For example, to identify if a speaker is a man,

women or child, the size of the speaker's vocal tract is an important feature that must be extracted from the audio that represents the data. However, it is difficult to know what features are relevant for many tasks. One solution is to let AI systems use a set of ML techniques to automatically discover the representation by themselves. This is known as **representation learning** and results in better performance with even less human intervention [1].

An important task for many real-world AI applications is to separate the **factors of variation** from the observed data. Consider for instance a facial expression recognition application (example from [5]). Then two images of different individuals with the same facial expression are most likely separated in pixel space. Meanwhile, two images of the same individual showing different expressions will be positioned close to each other in the pixel space. In this case, two factors are responsible for the variation in data: the first is the identity of the individual, and the second is the facial expression. For facial expression recognition the identity of the individuals are highly irrelevant. Yet this factor dominates the variation of the data so a facial expression recognition application that is based on the pixel space will suffer from poor performance. It is therefore important to **disentangle** the factors of variation and discard the one that is irrelevant. But factors of variation are tightly entangled and hard to separate. However, Deep Learning (DL) algorithms can tackle this problem [1].

DL is a subfield of ML and has a special style for learning representations from data. Instead of learning one representation, DL algorithms learn successive layers of increasingly meaningful representations of the data. In other words, representations are expressed in terms of other, simpler representations. With this approach a hierarchy of features is built and it is therefore possible to extract high-level features from raw data. The hierarchy or layers of features are built on top of each other which creates a deep graph, hence the name Deep Learning. The quintessential example of a DL model is an artificial neural network (ANN) [6]. The research around DL exploded in 2012 when Alex Krizhevsky achieved remarkable results in the ImageNet competition (ILSVRC2012) using a convolutional neural network (CNN) [7]. But the pioneer of CNNs goes to Yann LeCun when he in 1989 used a CNN to recognize handwritten digits [8]. At that time, DL algorithms were outperformed by other ML algorithms due to two factors: the first was because the lack of available data and the second due to bad performance in hardware. So researchers did not see the potential of DL until a few years ago when the amount of data and the hardware performance increased. Today, DL is used in facial recognition, robotics, object detection, self-driving cars, translation, classification, speech recognition, computational biology and a plenty of many other systems. The list is long and the contributions from DL has led to revolutionary advances in ML and AI, and the evolution has just started [1].

1.1 Objectives and Thesis Question

One interesting branch of unsupervised learning techniques is **Generative Models**. Usually it is difficult to analyze and understand data but generative models can do so. They are trained to discover the essence of data in some domain in order to generate similar data. This technique can be used in many tasks, for instance for image denoising, inpainting, super-resolution, structured prediction, exploration in reinforcement learning etc. It may also be possible to create for instance Photoshop commands such as "make my smile wider". In the long run the idea is to let computers automatically learn the natural features of data, and to get a better understanding of the world, which is an important contribution to AI.

A generative model that has recently achieved major success is called **Generative Adversarial Network** (GAN) [9] and it was introduced in 2014. A GAN has a special structure, it consists of two ANNs that work together. In this thesis, a GAN will be used to generate samples of pre-training data. The generated samples can be used to further train other algorithms, for instance an algorithm for classification or object detection. For all ML algorithms, having sufficient and relevant data is crucial because without enough data, the algorithms' performance will deteriorate significantly. In many cases it is a problem for researchers and enterprises to find adequate datasets. The main objective for this research is therefore to

investigate if GANs can solve this problem. At the request of *Sony Mobile Communications* in Lund, a GAN will be used to generate synthetic images. The idea is to use the generated images in a classification task.

Thesis question:

- Will additional samples of synthetic images generated by a GAN improve the accuracy in a classification task?

1.2 Delimitations

Three datasets were used for this research with the aim to understand the potential of a specific GAN model, and to investigate if generated images from that specific model can increase the accuracy in a classification task. In the classification task, a specific model was also used to classify images. This means that the results are restricted to the specific models and to the specific datasets used in this research. Thus, the results potentially may not be generalized.

1.3 Outline

The remainder of this thesis is organized as follows: chapter 2 gives the theoretical framework of Deep Learning. Chapter 3 introduces the Generative Adversarial Network and chapter 4 features the research methodology. The obtained results are presented and discussed in chapter 5 and 6, respectively. Finally, chapter 7 concludes the research and provides directions for possible future work.

2 ARTIFICIAL NEURAL NETWORK

Artificial neural networks (ANNs) [6] can be seen as algorithms that are supposed to extract multiple levels of representation from input data. An algorithm consists of many simple computations, when the computations are combined together they create a complex algorithm or network. ANNs are organized in layers, one input layer, one output layer and in between them hidden layers. A hidden layer just indicate that it is neither an input nor an output of the whole network. If there are more than one hidden layer then the network is called deep neural network (DNN). DNNs have a large number of **model parameters** which makes them very powerful. A model parameter is a variable within the network itself whose value can be estimated from data. Additionally, networks also have **hyperparameters** [1], that is variables external to the model and whose value cannot be estimated from data. It is crucial to configure and fine tune both the model parameters and the hyperparameters in order to obtain high performance [10, 11, 12].

Each layer is connected with nodes called **neurons** [13], the connections are different for different types of architectures. When every neuron in the network is connected to every neuron in adjacent layers, the network is called **fully connected**. Additionally, when the output from one layer is used as input only to the next layer the network is called **feedforward** [1]. This means that information is always fed forward and never fed back [12]. Figure 2.1 illustrates the architecture of a feedforward fully connected DNN.

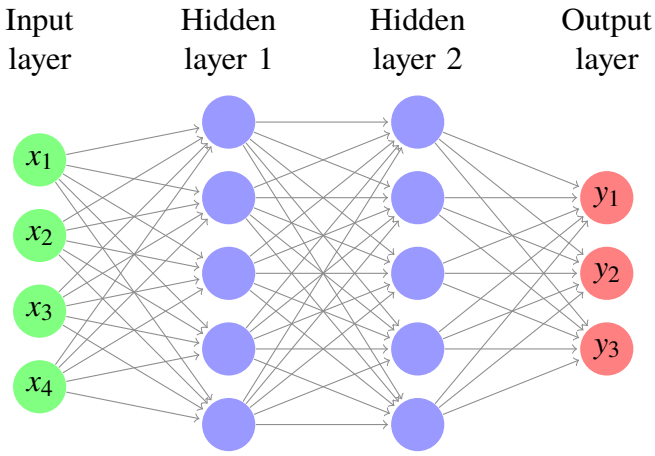

Figure 2.1: *An illustration of a feedforward fully connected deep neural network. The network consist of an input layer with 4 neurons, two hidden layers with 5 neurons in each, and an output layer with 3 neurons.*

2.1 Neuron

The first artificial neuron was developed in 1957 by Frank Rosenblatt which he named **perceptron** [13]. A neuron receives one or more inputs $(x_1, x_2, ..., x_n)$ and each input respectively have a weight $(w_1, w_2, ..., w_n)$. The weights determine the importance of the input to the output. The inputs and the weights can be seen as two vectors, x and w. The dot product of the two vectors gives a **weighted sum**. The weighted sum is then passed to an **activation function** that determines the output, more about this in the next section. This process is performed in each neuron so many neurons connected to each other creates a complex algorithm. Usually networks also have so called **biases**. A bias is another model parameter similar to a weight except that it is independent from the input [12]. Figure 2.2 shows how the output of an artificial neuron is produced.

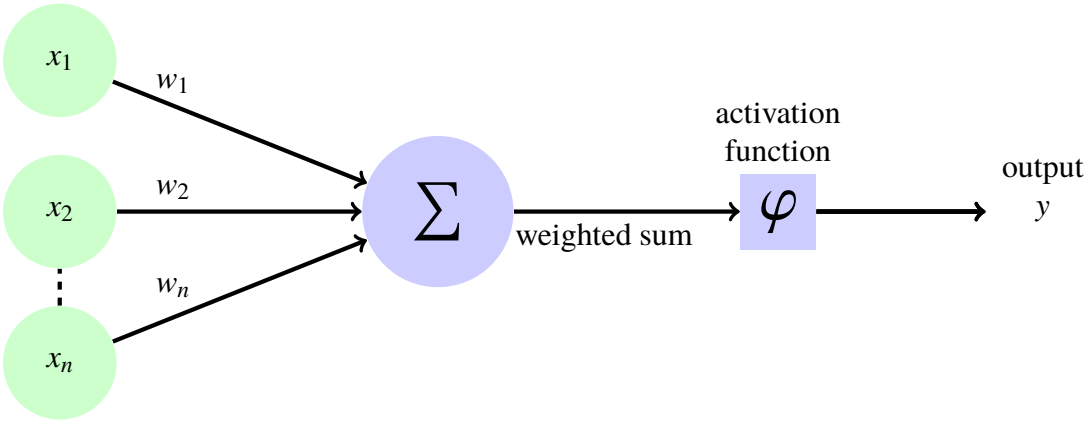

Figure 2.2: *An example of an artificial neuron. The neuron receives information through it's connections. First, the weighted sum is calculated as a dot product between the inputs x and their respective weights w. Then an activation function converts the weighted sum to an output y.*

2.2 Activation Function

The activation function converts the weighted sum to an output, we say the activation function decides if a neuron **fires** or not. The activation of the perceptron was the step function. As you probably already know, the output of a step function is based on a threshold and it is binary -that is, the output is either 1 or 0. Think about it as if the neuron is either **activated** or not. Now this is a good property for binary classification but what about the case where we want a network to classify more than two classes? In this case a property where a neuron can be "80% activated" or "60% activated" is needed. Because then for each class there can be a neuron in the output layer that determines the probability. If more than one neuron fires, simply find the one with the highest activation [12].

2.2.1 Linear Function

The output of a linear functions is proportional to the input, this way they can give a range of activations. But linear functions can actually not be used as activation functions because of two problems. The first is that the derivative of a linear function is constant which means that the gradient has no relationship with the input. With a constant gradient, learning will not be possible as the descent will be on constant gradient, this will be more described in later sections [14].

The second problem is that the activation of one layer will be the input into the next layer which will calculate the weighted sum on that input and the output will be just another linear function. This will be repeated for any number of layers so that the final activation is just the linear function of the first layer. In that case all layers can be replaced by one single layer which means that the important property of stacking layers together is lost [14].

2.2.2 Sigmoid and Tanh

Sigmoid is a logistic function and has a shape that is a smoothed version of the step function. But Sigmoid is **non-linear** which is great because it allow us to easily back-propagate errors. **Back-propagation** [15] is a common method used during the training phase of neural networks (NNs). It is also good for classification since it has a tendency to significantly change between x values -3 to 3 which will push the y values towards the extremes, see figure 2.3. This is very desirable when a network is trying to classify some input to a particular class [14, 1].

The sigmoid function were widely used but it had some drawbacks. As seen in figure 2.3, the derivatives show us that when the input is very positive or negative, the gradient becomes flat. So when the function falls in that region the gradient will approach to zero. This means that the network will refuse to further learn or it will become drastically slow. This is known as the **vanishing gradient problem** and for sigmoid only when the input is near 0 the sensitivity is strong. Another problem with Sigmoid is that it is not **zero-centered** meaning all data that goes out of a neuron is positive since it ranges from 0 to 1. This implicates the process of learning because during back-propagation the gradient on the weights will become either all positive or all negative. This problem has less severe consequences compared to the vanishing gradient problem however the problem is easily solved by scaling the Sigmoid function [14, 16].

Tanh is a scaled version of the Sigmoid function and ranges from -1 to 1 so it is zero-centered which solves one problem which makes training easier. But it also suffer from the vanishing gradient problem. Another difference is that the gradient is stronger, i.e. steeper derivatives, which can be required in some cases. This does not mean that the sigmoid is never used anymore, in some cases it is required especially when the network is not feedforward [14, 16].

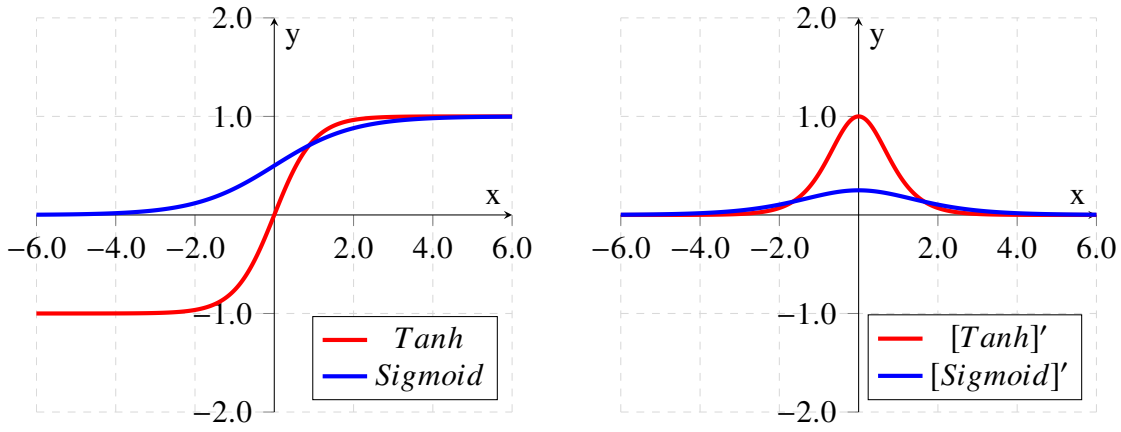

Figure 2.3: **Left:** *A plot of Sigmoid and Tanh.* **Right:** *A plot of the derivatives of Sigmoid and Tanh.*

2.2.3 ReLU

ReLU [17] stands for rectified linear unit and is, today, the most widely used activation function. ReLU is of course non-linear just like the Sigmoid and Tanh and it doesn't suffer from the vanishing gradient problem. The difference is that ReLU does not activate all neurons at the same time because, as seen in figure 2.4, when the input is negative it will convert it to zero and thus it will block some neurons from firing. This makes the network sparse and the computation becomes easier and more efficient which results in faster training [14, 18].

But ReLU also have a problem, for negative inputs the gradient is zero which means that the weights will not update. This can create dead neurons meaning they will no longer respond to any change and thus never fire anymore which makes a part of the network passive. This can be solved by making the negative side non-horizontal. Leaky ReLU is a variant where the negative side is slightly inclined, there are also other variants. It is important to know that ReLU is limited to only be used within the hidden layers of a NN [14, 18].

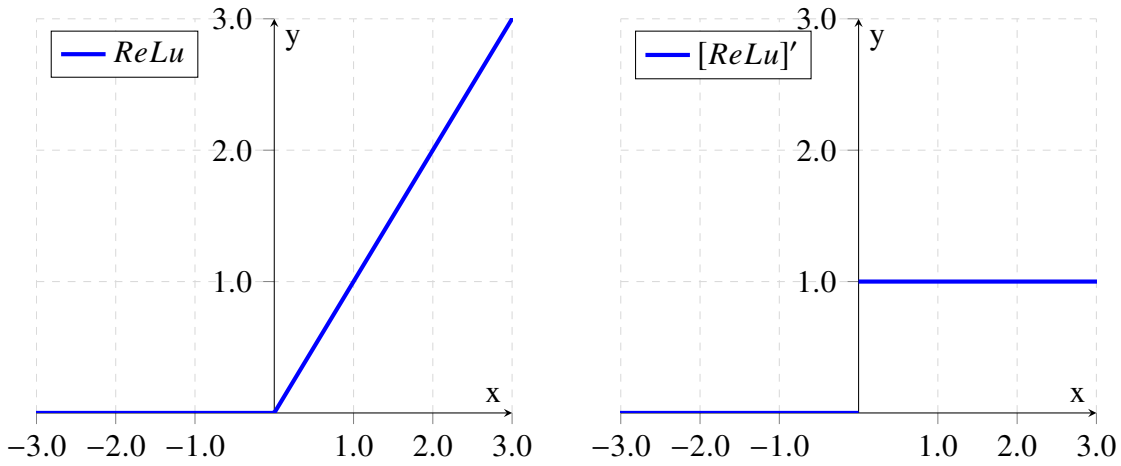

Figure 2.4: **Left:** *A plot of ReLU.* **Right:** *A plot of the derivatives of ReLU.*

2.2.4 Softmax

Softmax is used in the output layer for classifications. It simply gives the probability of the input being a particular class. The Softmax function is also a type of a Sigmoid function. Simply explained it takes an vector of arbitrary real values in order to produces another vector with real values in the range 0 to 1 that add up to 1, i.e. a vector with probabilities [1].

2.3 Loss Function

The loss function, also known as the cost function, computes how inconsistent the output of a network is compared to the intended ground truth. A higher loss value means less accuracy, thus the idea is to minimize the loss value. In order to minimize the loss function, i.e. get higher accuracy, the model parameters, e.g. the weights, in the network need to be adjusted. Why particularly the weights (assuming they are the only model parameters)? Simply because the loss is a value of how inconsistent the output is and the output depends on the activation function, which in turn depends on the weighted sum and the weighted sum is a dot product of the input x and the weights w. The input is fixed and can not be changed, thus only the weights are left [12, 1].

There are a number of loss functions but in most cases in classification the principle of maximum likelihood is used, meaning that the loss function is the **cross-entropy** between the training data and the model's predictions. Another well known loss function for classification is the **mean squared error** (MSE). However, in most cases the cross entropy is preferable and considered slightly better because during back-propagation the gradient can be smaller with MSE so the training is more likely to stall. It is worth noting that a current study highlights the importance of investigating how particular choices of loss functions affect NNs and their learning. The crucial aspects are the accuracy, the learning speed and the condition of the data, e.g. noisy data [19].

2.4 Gradient Descent

The model parameters are adjusted by an optimizer during training until the desired output is reached. The **gradient descent** [20] is a general algorithm used to minimize the loss function. The partial derivatives from the gradient of the loss function are useful because they tell how to change the model parameters in order to make a small improvement in the output. The gradient points directly uphill so a parameter is updated by taking a small step in the opposite direction of the gradient, the small step is known as the **learning rate**. The size of the learning rate is difficult to set and also very important. It must be small enough to avoid overshooting a minimum but at the same time big enough to in order to make progress.

The new updated parameter can algebraically be written as [12]:

$$\theta' = \theta - \eta \nabla f(\theta) \tag{2.1}$$

Where:

θ': is the new updated parameter, e.g. a weight
θ: is the old parameter
η: is the learning rate
$\nabla f(\theta)$: is the gradient of the loss function

The process of calculating the gradient of the loss function with respect to the network's parameters is usually made by a **back-propagation** algorithm [15]. The calculation of the gradient proceeds backwards through the network, thus the name back-propagation. Then an optimizer, for instance the gradient descent algorithm, uses the gradient to adjust the model parameters. This is repeated until the minimum is found or the gradient adequately converges [12, 21].

The gradient descent comes in three variants and they differ in the amount of data used to calculate the gradient. The trade-off is between the accuracy of the parameter to update and the time it takes to perform an update. First out is the regular gradient descent which is described by equation 2.1. This variant is slow because it calculates the gradient for the whole dataset to make just one update. The second variant is the **stochastic gradient descent** (SGD) that performs a parameter update for each training example x_i and is therefore usually faster. However the loss function tend to fluctuate heavily due to the frequent updates and it complicates the convergence to the exact minimum. This can be avoided by diminishing the learning rate during training. Lastly we have the **mini-batch gradient descent** which is a mix of the two first variants. It performs an update for every mini-batch of n training examples, the mini-batch sizes is usually chosen to be between 50-256 but it always depends on the dataset. This variant is typically the algorithm of choice because it is fast and has a more stable convergence. Usually when the SGD term is used, the intention is to refer to the mini-batch gradient descent variant. Even though SGD is fast, there are other variants that use optimized matrix optimizations in order to make the computation even more efficient [21].

The most important aspect during training is the cost so it is extremely important to minimize it as much as possible. When it comes to back-propagation, researchers have developed many good optimizers. However, the one that works well in practise and that is favorable is called **adaptive moment estimation** (Adam) [22]. Adam has an **adaptive learning rate** in order to make smaller or larger updates to each individual weight based on their importance. **RMSprop** [23] and **Adadelta** [24] are two other optimizers that are similar to Adam. The main difference is that Adam has bias-correction as well as momentum, a method that accelerate an optimizer to the relevant direction and dampens oscillations [21, 22].

2.5 Overfitting and Underfitting

An important aspect for all learning models is how well they generalize to new data. Models in the real world are not likely to be used on data they have been trained on, the purpose is to apply models on new data that they have never seen before. Overfitting and underfitting are terminologies used to address the performance of ML models. Underfitting refers to models that neither learns from training data nor generalize to new data. This simply means that the model is not suitable and one should try something else [1].

Overfitting on the other hand is when a model performs good on training data but bad on test or validation data, i.e. on new unseen data. Overfitting may occur if the dataset is to small or if the model capacity is not high enough. In these specific cases collecting more data and increasing the model capacity, e.g. number of parameters, respectively solves the problem. But overfitting can still occur even when both the dataset

and model capacity are enough. That happens when a model models the training data too well, that is when the model applies learning on unwanted details and noise to such an extent that it negatively impacts the performance of the model on new data. In this case increasing regularization prevents overfitting [25, 1].

2.5.1 Dropout

Dropout [25] is a stochastic regularization method often used due to its simplicity and high effectiveness. It improves the generalization performance in a wide variety of application domains including object classification, digit recognition, speech recognition, document classification and analysis of computational biology data. The idea is to approximate an equally weighted geometric mean of the predictions of an exponential number of learned models that share parameters. Because a combination of several models with different architectures nearly always improves the performance. But it is prohibitively expensive to do this with separately trained models. Dropout is a technique that addresses this problem and at the same time prevents overfitting [25, 1].

The trick is to sample many "thinned" networks from the original model. A network with n nodes can be seen as a collection of 2^n possible thinned networks and all networks share weights together. This is obtained by randomly selecting nodes to dropout, i.e. to temporarily remove them from the network as shown in figure 2.5. For each training iteration different nodes are randomly selected to dropout. In the simplest case each node is retained with a fixed probability p. For hidden nodes smaller p slows down the training and leads to underfitting. Large p may not produce enough dropout to prevent overfitting. For a wide range of networks it is optimal to set p to 0.5. For nodes in the input layer it is more optimal to choose the probability of retention to be closer to 1 than 0.5, for instance 0.8. For extremely small datasets (100, 500) dropout may not give any improvements and for larger datasets the improvements increases up to a point. It is important to note that dropout is intended to be used during training and should not be used during testing or validation [25, 1].

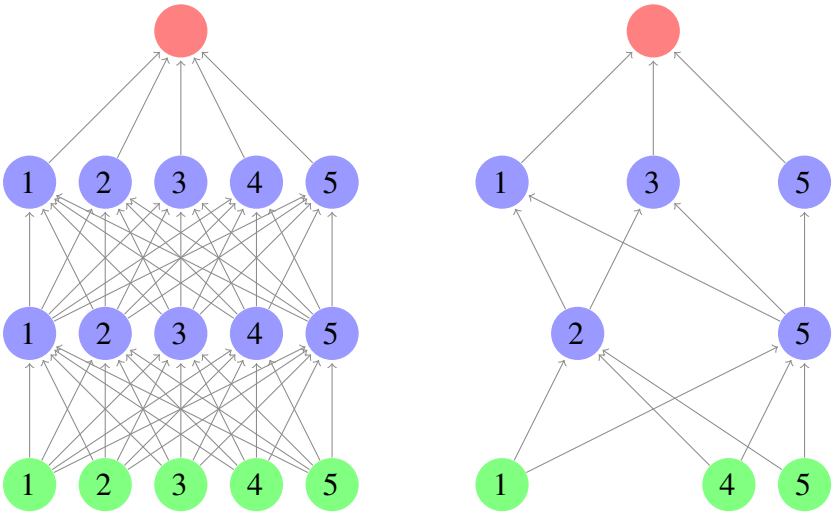

Figure 2.5: **Left:** *A standard network with 2 hidden layers.* **Right:** *An example of a thinned network after applying dropout to the network on the left.*

So training a neural network with dropout can be seen as training a collection of 2^n thinned networks. The drawback with this technique is the increasing training time. Typically it takes 2-3 times longer to train with dropout so the trade-off is between overfitting and training time. If longer time is not an issue, one can use higher dropout [25].

2.5.2 Early Stopping

Early stopping is another form of regularization used to avoid overfitting. With early stopping, the training of a model is the same as traditional training up to a point where the model simply quits training. The point is determined by computing the accuracy on the validation data at the end of each **training epoch**, one epoch corresponds to training on the whole training set once. When the validation accuracy stops improving, one should terminate the training. In this way the model stops training just before the generalization error starts to increase. By applying early stopping relevant number of training epochs is determined [26, 12, 1].

2.6 Convolutional Neural Network

Convolutional neural networks (CNNs) [8] have a special type of architecture and they have shown considerable success in in the field of computer vision. Today, many companies are using CNNs for their services, Facebook uses them for their automatic tagging algorithms, Google for their photo search and Instagram for their search infrastructure, just to mention some [12, 27, 1].

In fully connected layers the input is a vertical line of neurons, the vector x. In convolutional layers the input is a square of neurons that corresponds to the pixels of an image (assuming an grayscale image with the same width and height). This design is important for pictures because it allow convolutional layers, unlike fully connected layers, to take into account the spatial structure of images. The structure of a CNN does not only include convolutional layers, it may include a combination of convolutional, nonlinear (e.g. ReLU), pooling (downsampling), and fully connected layers in series [12, 27, 1].

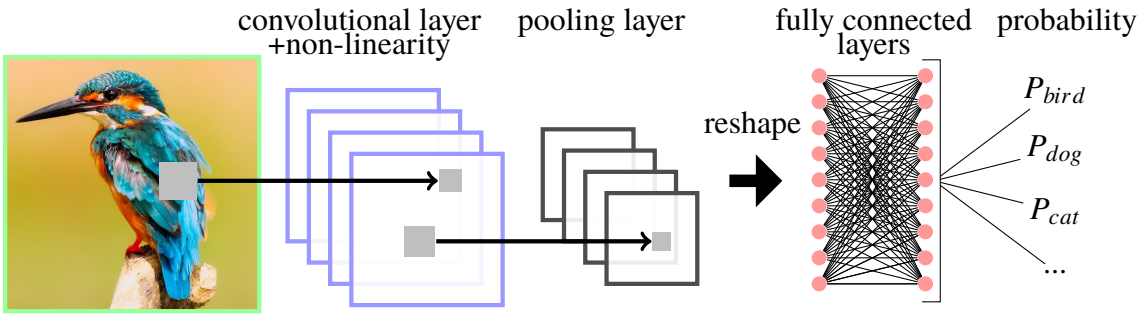

Figure 2.6: *The architecture of a CNN that performs classification. In the structure of the network one can see a combination of different layers that are common in CNNs. The image of the bird is from Max Pixel and licensed under the Creative Commons Zero (CC0) License, available at:* https://creativecommons.org/publicdomain/zero/1.0/deed.en

2.6.1 Convolutional Layer

As mentioned the input layer is the pixel dimensions of an image and the first hidden layer in a CNN is a convolutional layer (from now on convolutional is abbreviated to conv). The input neurons (pixels) will be connected to the hidden neurons in the conv layer but not as in the usual way. Every hidden neuron will be connected to a small, localized region of the input image. Let's say that the size of this region is 3x3, which corresponds to 9 input neurons. This means that a hidden neuron will connect to 9 input neurons. These 9 neurons of a region is called the **local receptive field** for that hidden neuron. The depth of the receptive field must match the depth of the input image. As mentioned before, for our example suppose a grayscale image, such images have the depth 1. RGB images have the depth 3 so the dimension of the receptive field would have been 3x3x3 [12, 27, 1]. See figure 2.7.

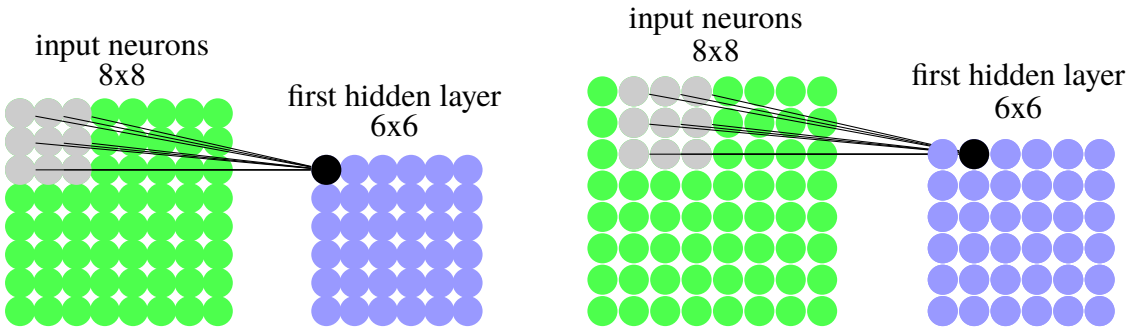

Figure 2.7: *An illustration of how a filter slides across an input and how the hidden neurons are connected to their local receptive field (the gray neurons on the input). The size of the filter is 3x3 which means that the receptive field corresponds to 9 neurons. When the filter convolves on the 8x8 input we get a hidden layer with 6x6 neurons.*

The hidden neurons learn their particular local receptive field. The receptive field slides across the entire input image. One can also say that a **filter** or **kernel** of the size 3x3 convolves around the input. The size of the shifting is determined by the **stride length**. A stride length of 1 corresponds to shifting the filter by one neuron as seen in figure 2.7. The size of the hidden layer (number of hidden neurons) depends on the image size, the filter size and the stride length. As seen in figure 2.7, an 8x8 image with a 3x3 filter and a stride of 1 corresponds to 6x6 neurons in the hidden layer. This is because the filter can maximum slide 5 neurons across (or down) before it collide with the right-hand side (or bottom) of the input image [12, 27, 1].

So when applying a conv layer the output volume shrinks, i.e. the spatial dimensions decreases. This means for every conv layers, the size will continuously decrease. One can maintain the same volume by applying **zero padding**, that is to pad the input volume with zeros around the border as illustrated in figure 2.8. Zero padding is sometimes applied in early layers in order to preserve as much information about the original input as possible. The formula for calculating the spatial size of the output volume for any conv layer is [28]:

$$o = \frac{i + 2p - k}{s} + 1 \tag{2.2}$$

Where:

o: is the size of the output volume
i: is the size of the input volume
k: is the size of the kernel/filter
k: is the amount of zero padding
s: is the stride length

2.6.2 Shared Weights

Let's continue with the same example, a 3x3 filter that convolves over a 8x8 input image which will result in a 6x6 hidden layer. Now the first position of the filter is, for example, the top left corner and this region is the receptive field for the first hidden neuron in the top left corner of the hidden layer. And the size of this region is 3x3 which means that the hidden neuron will have a total of 9 connections. Each of the connections have a weight and one bias for all 9 connections. This means that this hidden neuron have 3x3 weights and a single bias connected to its local receptive field. Now the next hidden neuron have exactly the same 3x3 weights and bias. Actually all 6x6 hidden neurons have the same weights and bias since the parameters in a conv layer is shared between all neurons. This is because the hidden neurons detect the exact same **feature** but in different locations in the input image. A feature is some kind of a pattern that will cause the neuron to activate. The pattern could be for instance an edge or any other shape, colour,

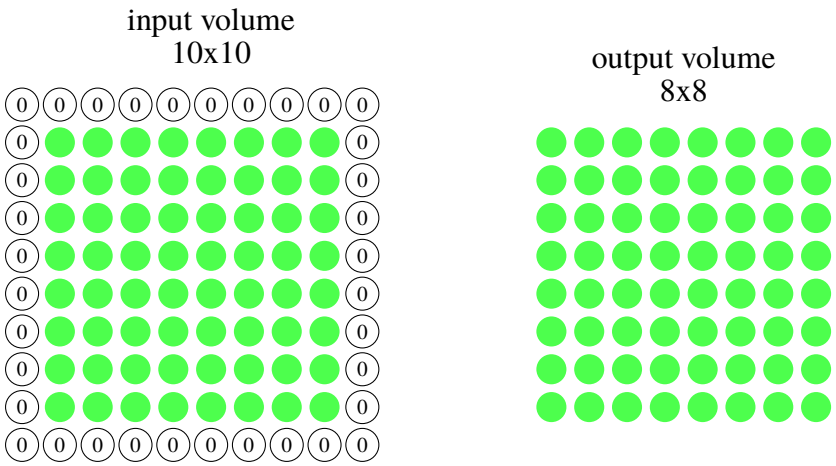

Figure 2.8: *An illustration of how zero padding is applied to an 8x8 input in order to maintain the volume. The input volume increases to 10x10 after zero padding and with a 3x3 filter the output volume will be 8x8, thus the input dimensions are maintained.*

shade etc. By sharing weights the number of parameters in a model greatly reduces. That is an advantage because it will result in faster training [12, 27, 1].

So what is happening when a filter convolves around the input image? The answer is element wise multiplications because the filter we have been talking about is actually the shared weights (plus bias) which can be stored in a vector. The values of the 3x3 weights are multiplied with the original pixel values of the image in each receptive field. Eventually, when the filter has convolved over the whole input, the multiplications for each receptive field are summed up and stored into another vector that is called activation map or **feature map**. The size of the feature map is equivalent to the size of the hidden layer, so for our example that is 6x6. One filter creates one feature map for a single kind of feature, in order to detect more features and preserve the spatial dimensions better there must be more filters. Let's say that we use 4 filters instead of one, then we will get 4 feature maps (6x6x4 vector) from the first convolution, see figure 2.9. As we go deeper, i.e. more conv layers, we get feature maps that represent more and more complex features [12, 27, 1].

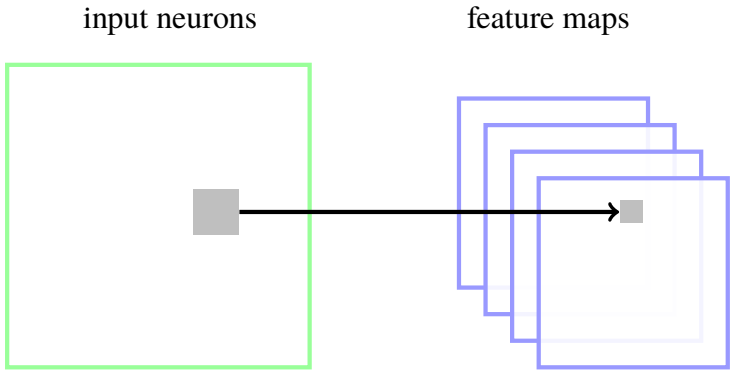

Figure 2.9: *Shows that for each filter applied on the input there will be a feature map.*

2.6.3 Pooling Layer

Immediately after a conv layer there is a nonlinear layer, e.g. ReLU, but before a second conv layer one can apply a pooling layer. This layer is intended to simplify the information outputted by the conv layer, so

it produces a condensed feature map. There are different procedures for pooling, one common procedure is the max-pooling. It simply outputs the maximum activation in a region, for instance in figure 2.10 for every 2x2 region only the maximum activation is forwarded. Pooling is applied for each feature map separately [12, 27, 1].

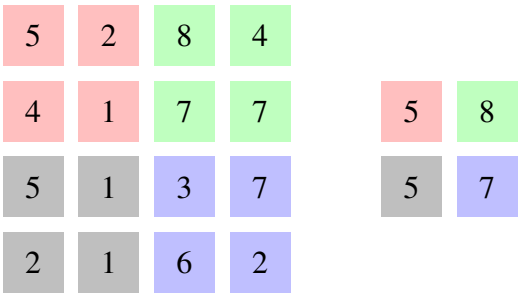

Figure 2.10: *An illustration of how a 2x2 max-pooling is applied on a 4x4 feature map. In each 2x2 region (red, green, black, and blue) only the maximum activation is forwarded.*

A Pooling layer actually throws away the exact location of a feature. This is because the exact location is not important, only the relative location to the other features are more important. This approach will drastically reduce the number of parameters and in turn the training will become faster. Another purpose gained from using pooling is that it will control overfitting [12, 27, 1].

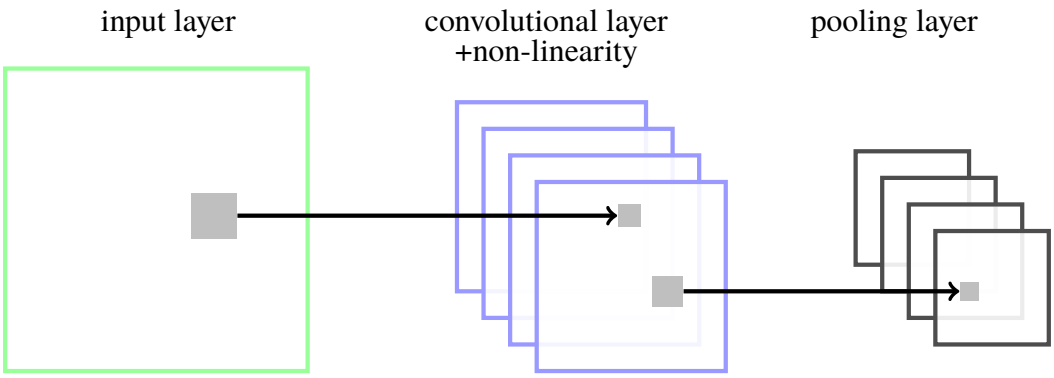

Figure 2.11: *A typical combination of layers used in a CNN. The three layers are convolutional, nonlinear and pooling. This combination is usually used in series before one or several fully connected layers are applied to the end of the network.*

3 GENERATIVE ADVERSARIAL NETWORK

The most impressive success in DL have, so far, involved discriminative models, i.e. models that map the dependence of unobserved target variables (y) on observed variables (x). In simple terms, discriminative models infer outputs based on inputs without caring about how the input was generated. Generative models, as opposed to discriminative models, maps how the input data was generated. They are a branch of unsupervised learning techniques [9, 29].

Generative Adversarial Networks (GANs) [9] are a class of generative models. GANs are taught to generate synthetic data similar to some known input data. A GAN model consists of two types of neural networks, a generative model and a discriminative model. The two networks have an adversarial relationship where they compete against each other. The generative model learns to generate data while the discriminative model learns to determine whether a sample is from the model distribution (p_{model}) or the data distribution (p_{data}). During training both models improve their methods until the artificially generated data are indistinguishable from real data[9, 29].

3.1 The Structure of a GAN

The two networks are represented by differentiable functions with respect to each network's input and parameters. The discriminator is defined by a function $D(x)$ where x (observed variable) is the input. $D(x)$ gives the probability that x came from p_{data} rather than p_{model}. It is a binary classifier with two classes, when x is real the probability is 1 and when x is synthetic (fake) the probability is 0. The discriminator can be seen as a typical CNN that transforms a 2- or 3-dimensional matrix of pixels into probabilities [9, 29].

The generator is represented by a function $G(z)$ where z (latent variable) is the input. $G(z)$ outputs a sample of x drawn from p_{model}. The input z is sampled from some simple prior distribution. The inputs to the function G do not necessary need to correspond to the inputs of the first layer in the generator network, they may be provided to any layer. For instance, z can be partitioned into two vectors z_1 and z_2, then z_1 can be fed to the first layer while z_2 can be added to the last layer. Other popular strategies are applying additive or multiplicative or concatenate noise to hidden layers. The restrictions are few on the design of the generator. The requirements are that the dimension of z must be at least as large as the dimension of x, and that G is differentiable. The generator can be seen as a kind of reverse CNN. It takes an z-dimensional vector of noise and upsamples it to an image [9, 29].

As mentioned the discriminator and generator takes respectively x and z as inputs but they can be extended to take additional inputs. For instance, they can both be conditioned on some extra information y, where y can be any kind of auxiliary information such as class labels, in order to generate samples that are based on some specific condition. This kind of model is known as a conditional Generative Adversarial Network (cGAN) [30]. The functions that defines the discriminator and generator respectively becomes $D(x|y)$ and $G(z|y)$.

3.2 The Training Process

The training process for GANs means training the generator and discriminator simultaneously. The first step is to sample two mini-batches: one of x values from p_{data}, the dataset, and one of z values from p_{model}, the model's prior over latent variables. Then for each network a gradient step is made, one updating the discriminator's parameters $\theta^{(D)}$ to reduce its lost function $J^{(D)}(\theta^{(D)}, \theta^{(G)})$. And the other gradient updates the generator's parameters $\theta^{(G)}$ to reduce its lost function $J^{(G)}(\theta^{(G)}, \theta^{(D)})$. The tricky part here is that there are two loss functions, they are defined in terms of both players parameters but each network can only control its own parameters. This training process is repeated for a number of training iterations [9, 29].

As you have noticed, GANs are complicated to train. There must be a balance during training between the discriminator and generator otherwise one can overpower the other. It is therefore important to have the correct hyper-parameters, network architecture and training procedure. The discriminator can overpower the generator when it classifies synthetic data with absolute certainty. In this case the generator is left with no gradient so learning will not be possible. The solution is to have unscaled output from the last layer in the discriminator. Another common problematic scenario is when the generator overpowers the discriminator. This problem is known as **mode collapse**, it occurs when the generator discover some weakness in the discriminator. This problem is recognized when the generator produces many very similar data regardless of variation in the input z. It can be avoided by making the discriminator better, for instance by adjusting its training rate. The discriminator can be trained for K steps for each step the generator trains. Algorithm 1 presents a pseudocode example of a training process where training rate can be adjusted if $K > 1$ [9, 29].

Algorithm 1: Pseudocode example of a training process for GANs. The hyperparameter K determines the number of steps to apply to the discriminator. When $K = 1$, the training is simultaneously, with one step for each network.

```
foreach number of training iterations do
    foreach K steps do
        sample mini-batch of m noise samples (x^(1), ..., x^(m)) from p_data;
        sample mini-batch of m noise samples (z^(1), ..., z^(m)) from p_model;
        update the discriminator;
    end
    sample mini-batch of m noise samples (z^(1), ..., z^(m)) from p_model;
    update the generator;
end
```

During training the discriminator receives one of two possible inputs. The first is when x is mapped from the dataset p_{data}, i.e. when x is real data. In this scenario, the discriminator's goal is for $D(x)$ to be near 1. In the second scenario both networks participate so that $x = G(z)$, i.e. the discriminator receives synthetic data. Here, the discriminator strives to make $D(G(z))$ near 0 while the generator strives to make it near 1. This is how the generator learn to produce better data. Eventually, if the models have sufficient capacity, they will achieve Nash equilibrium which in this context corresponds to $G(z)$ being drawn from the data distribution, i.e. $p_{model} = p_{data}$. When this is achieved, $D(x) = \frac{1}{2}$ will be true for any x [9, 29].

3.3 Deep Convolutional GAN

Deep Convolutional Generative Adversarial Networks (DCGAN) [31] is a class of GANs with a special architecture that many are applying today. The special architecture has resulted in a more stable training as well as the possibility to generate higher resolution images. The authors behind DCGAN adopted recently demonstrated changes to the CNN architecture [31].

The first approach was to let the overall network structure be all-convolutional. The pooling layers were replaced with strided convolutions (discriminator) or transposed convolutions, also called fractionally-strided convolutions, (generator) with a stride greater than 1. This allowed the generator to learn its own spatial upsampling. The fully connected hidden layers were eliminated. The generators input z was reshaped and the last conv layer in the discriminator was flattened and then fed into a single sigmoid output [31].

The second approach was to use batch normalization [32] in the layers so that each neuron had zero mean and unit variance. This helped to stabilize learning and allowed the gradient to flow deeper and prevented

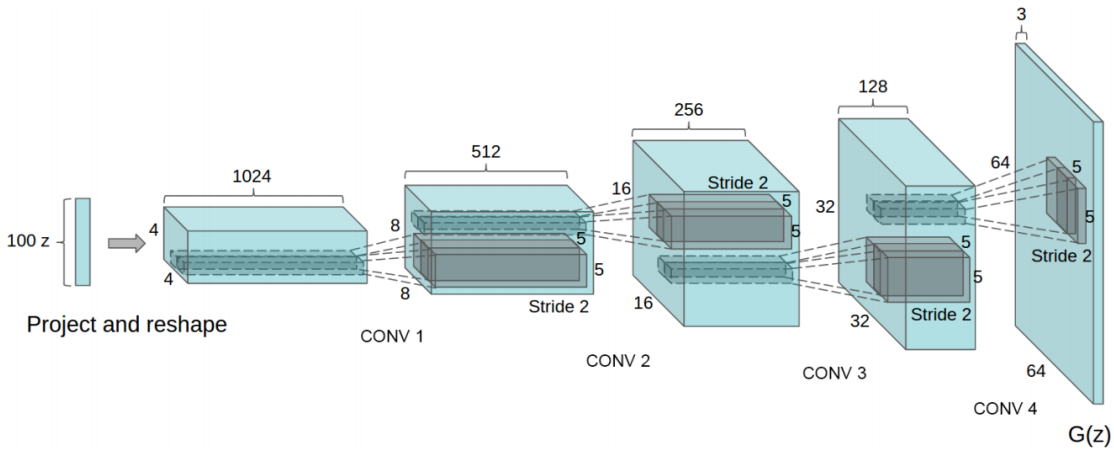

Figure 3.1: *An example of a generator network in a DCGAN. A 100 dimensional z is fed into the first convolutional layer. The high level representation is converted into a 64x64 pixel image by a series of four transposed convolutions, also called fractionally-strided convolutions. The structure does not include any polling or fully connected layers. Figure borrowed from the DCGAN paper [31].*

the generator from collapsing. However, the generator's output layer and the discriminator's input layer were not batch normalized in order to let the model learn the correct mean and scale of the data distribution and to avoid sample oscillation and instability [29, 31].

The activation functions in the generator were different from the ones in the discriminator. The discriminator had Leaky ReLU in all layers while the generator had ReLU in all layers except for the output, which used Tanh [31].

4 RELATED WORK

There has been a large interest in deep generative models in recent years. Two popular deep generative models other than the GAN are the Variational Autoencoders (VAEs) [33] and the Pixel Recurrent Neural Networks (PixelRNN) [34]. VAEs are probabilistic graphical models that consists of an encoder and a decoder. The high level features of a data distribution is encoded into a low level representation. This low level representation is known as a *latent vector*. The decoder can produce a high level representation given a latent vector. The produced results from VAEs are often blurry.

PixelRNNs are autoregressive deep neural networks that sequentially predicts the pixels in an image along the two spatial dimensions. The network scans an image pixel for pixel, and for each pixel it predicts the conditional distribution over the possible pixel values given the scanned context. This also means that an image is generated pixel by pixel, which is inefficient compared to the other models. The different approaches have their own advantages and disadvantages, VAEs have a nice probabilistic formulation, PixelRNNs have a simple and stable training process, and GANs seems to generate the sharpest images. In this research the visual quality of the generated images was the most important concern, thus GAN was selected among the deep generative models.

There have also been numerous of studies on how the vanilla GAN can be extended. In [35] the authors introduces an approach for semi-supervised learning with GANs. The discriminator is modified to produce an additional output so that it does not only give the probability that an input is real, but it also predicts the label of the input. The discriminator can predict $K + 1$ classes: the original K classes and a new class for synthetic images. This extension is promising because a small labeled dataset can be augmented with a larger unlabeled set of real and synthetic examples. However, in this research the real examples are fully labeled and the synthetic examples are sorted in directories when generated by a conditional GAN and later labeled with the original K classes.

Apple conducted an interesting research [36] where they addressed the problem of not achieving a desired performance when learning from synthetic images due to the gap between synthetic and real image distributions. The researchers propose a Simulated+Unsupervised (S+U) learning model, the model is an adversarial network similar to a GAN but a refiner neural network (R) is used instead of a generator. The input to the refiner is synthetic images instead of a vector of noise, the discriminator classifies the real and the refined (synthetic) images into two classes. Eventually the discriminator will no longer be able to distinguish the real images from the refined (synthetic) ones. This enables generation of highly realistic images.

There are many more GAN extensions, this paper [37] propose to learn the natural image manifold using a GAN. This research [38] propose a coupled GAN (CoGAN) to learn a joint distribution over images from multiple modalities without any tuple of corresponding images. InfoGAN [39], an information-theoretic extension of GAN, is able to learn disentangled representations in an unsupervised manner. This paper propose Style and Structure Generative Adversarial Network (S^2-GAN). The Structure-GAN generates a surface normal map and the Style-GAN takes the surface normal map as input and generates the 2D image.

There have also been a lot of research where synthetic data is used to train different models. Here [40], synthetic images are used for gaze estimation, i.e. estimate the eye gaze direction in images. A Fully-Convolutional Regression Network (FCRN) is trained on synthetic data for text localization in natural images [41]. 3D CAD models have been used to train CNNs in a target detection task [42]. A mix of real and synthetic data have been used for font recognition [43]. A mix of real and virtual data is used in [44], the authors shows that pre-training on virtual data improves performance. In this research a mix of real and synthetic data is used in a classification task.

Recently, a paper named "ARIGAN: Synthetic Arabidopsis Plants using Generative Adversarial Network" [45] was published while this research was still ongoing. The employed model in the ARIGAN paper is a conditional DCGAN that generates images of Arabidopsis plants where the condition is the number of leafs. The authors propose a new "Ax" dataset of artificial plants images, the Ax dataset is evaluated with a state-of-the-art leaf counting algorithm. When Ax is used as part of the training data, the testing error is reduced. In this research a cDCGAN is also used, however other datasets are used, the condition is the class/label of the input, and the synthetic examples are evaluated with a classifier.

Most recently, NVIDIA published a paper [46] were a new training methodology is used for a GAN. The idea is to progressively grow both the discriminator and the generator: starting from a low resolution, new layers are added in order to progressively increase the resolution. This allows the training to firstly concentrate on the large-scale structure of the image distribution, and then shift to the small-scale distribution, instead of having to learn all scales simultaneously. This approach results in synthetic images with an outstanding quality and at the same time the training process becomes faster and more stable.

5 METHOD

Research is a systematic process of collecting, analyzing, and interpreting data in order to understand a phenomenon [47]. Furthermore, research is defined as "a detailed study of a subject, especially in order to discover (new) information or reach a (new) understanding" [48]. Some characteristics of a research is that it originates with a question or problem, includes a plan for proceeding, requires collection and interpretation of data and is, by its nature, cyclic. Research tools are used to collect, manipulate, or interpret data and a research methodology is utilized to conduct a research. It is necessary that the research method is suitable for the chosen problem [47].

Three common research methodology approaches are the quantitative, qualitative, and mixed methods. They are adapted to the type of data needed to respond to the research question or solve the research problem. The quantitative method is based on the measurement of quantity or amount. It handles numerical, non-descriptive, data where statistics or mathematics are usually applied. The qualitative method is primarily used for exploratory research. It is a holistic approach that involves discovery and is used to understand reasons, opinions, or motivations. It provides insights into a problem and handles non-numerical, descriptive, data. The mixed method is a "mix" of both quantitative and qualitative. For instance, a qualitative method can be used to understand the meaning of quantitative data [49, 50, 51].

For this particular thesis the research question and the data needed to respond to it matches the characteristics of a quantitative research. This thesis tackles a problem that is a typical example of a quantitative research problem. Hence the research of this thesis proceeded with a quantitative method where the investigation included controlled experiments designed by the researcher.

Note that any hyperparameter that is changed from its default value is mentioned, non changed hyperparameters are not mentioned.

5.1 Research Tools

The following lists determines the research tools used for the experiments.

Hardware: Desktop PC
- RAM: 12GB
- Processor: Intel Core i7 CPU 950 @ 3.07GHz
- Graphic card: 4GB NVIDIA GeForce GTX 970/PCIe/SSE2
- Disk: 500GB HDD

Software:
- Operating system: Ubuntu 16.04.2 LTS
- OS type: 64-bit
- Programming language: Python 3.5.2 with Virtualenv (isolated Python environment)
  NVIDIA software for GPU support:
  - CUDA Toolkit 8.0 (Compute Unified Device Architecture)
  - cuDNN v6.0 (CUDA Deep Neural Network library)

  Deep Learning frameworks:
  - TensorFlow 1.4.1 with GPU support
  - Caffe 0.15.14
  - NVIDIA DIGITS 6.1.0 (Interactive Deep Learning GPU Training System)

5.1.1 CUDA and cuDNN

CUDA [52] is both a hardware and a software architecture for managing computations on a GPU as a data-parallel computing device. CUDA has been widely deployed in many applications since it was introduced in 2006. CUDA Toolkit [53] provides a development environment for creating high performance GPU-accelerated applications. During the experiments the DL frameworks use CUDA Toolkit in the background and the tool is therefore required to be installed for faster execution.

The cuDNN [54] is a library of DL primitives that optimizes DL workloads and provides optimized performance and memory usage. Since DL operations require intense computation and are very time consuming, an immediate efficiency improvement from this library is important. The cuDNN library supports most frameworks and transparent to the user through drop-in integration. This means there is no need to write parallel code manually, the library just need to be installed so that the GPU supported DL frameworks can take advantage of it.

5.1.2 TensorFlow

TensorFlow [55] is an interface for implementing machine learning algorithms including DL models introduced by Google. A TensorFlow computation is described by a graph of data flows. A graph has a set of nodes where each node represent a mathematical operation. Not all but some kind of nodes are allowed to maintain and update their persistent state. Data that flows between nodes are represented by edges. Data are expressed as tensors, arbitrary multidimensional arrays. Since the graph only describes computations, one need to lunch the graph in a distributed session in order to compute anything. A computational graph is constructed with one of the supported frontend languages such as Python. The TensorFlow library was used to implement a DCGAN.

5.1.3 Caffe

Caffe [56] is a DL framework created by Jia, et al., during his PhD at UC Berkeley. The framework is written with C++ and it has three interfaces: command line, Python, and MATLAB. Caffe is designed with high modularity where model definitions are written with separated config files using the Protocol Buffer language. Most parts of a neural network as well as loss functions, optimizers (solvers), back-propagation, learning rate policies, training and testing is already implemented and are in-built within Caffe. The user just needs to specify the type of layers, loss function etc. to use. This makes it very easy to use compared to other DL frameworks. Caffe is fast and mainly used with images. The Caffe framework was used together with DIGITS to specify a CNN classifier.

5.1.4 DIGITS

DIGITS [57] is a software system that runs in a web interface and is used for training DL models. It was developed by NVIDIA with a focus on image classification, segmentation and object detection tasks. It simplifies common tasks for DL model training such as formatting and loading images directly from a directory and monitors performance in real time. Caffe is required since it integrates with it, additionally it supports other framework such as TensorFlow. DIGITS was used to train the LeNet CNN classifiers.

5.2 Data Sets

The three following datasets were used to experiment on, all of them are publicly available.

Datasets:
- MNIST
- Fashion-MNIST
- Flower photos

The MNIST dataset [8] consists of 10-class handwritten digits where each digit have been size-normalized and centered in a fixed-size image. The dataset is divided into two sets: the first is the training set with 60,000 examples, and the second is the test set with 10,000 examples. Each image is a grayscale (bi-level) with a dimension of 28x28. This dataset was selected due to its simplicity and size that allows researchers to quickly check and prototype DL algorithms. Also, most ML libraries (e.g. scikit-learn) and DL frameworks (e.g. Tensorflow, Pytorch) provide functions that loads and prepares the dataset for usage. The MNIST dataset is available at http://yann.lecun.com/exdb/mnist/. The dataset is stored into four files with a special format. DIGITS can be used to download MNIST as regular images stored into directories based on their respective class, see DIGITS documentation.

The Fashion-MNIST dataset [58] is a new dataset introduced in 2017 with the aim to provide a good benchmark dataset with all the accessibility of the original MNIST. The dataset consists of 28x28 grayscale images of fashion products from 10 categories, with a training set of 60,000 examples and a test set of 10,000 examples. The motivation for the selection of this dataset was due to the advancement of today's DL models where the original MNIST simply is no longer a challenge at classification. The Fashion-MNIST was the ideal choice since it has all the accessibility of the original MNIST and is at the same time more challenging at classification. The Fashion-MNIST dataset is available at https://github.com/zalandoresearch/fashion-mnist and like MNIST it is stored in four files. DIGITS can also be used to download Fashion-MNIST but the URL links inside the "digits/download_data/mnist.py" file (line 21-24) must be changed to the URLs of the four files of Fashion-MNIST.

The third dataset is a collection of 3670 RGB images with different sizes. The images are of flowers from five categories. The five categories are the following: daisy, dandelion, roses, sunflowers and tulips. The collection is not preprocessed nor labeled but the categories are stored in separated directories. Furthermore, the collection contains a lot of noise in form of insects, unusual background and some images are not even of flowers. However, this dataset was selected due to a request from Sony where images of undesirable plants were in interest and this dataset was the nearest match that could be found. This dataset is only used to examine the quality of synthetic images generated by a DCGAN. The Flower photos dataset is available at `http://download.tensorflow.org/example_images/flower_photos.tgz`.

5.3 Data Preprocessing

The researcher wanted to test how different number of training examples from MNIST and Fashion-MNIST affects the DCGAN. Therefore, it was needed to "slice" the training sets into desired number of examples. The number of examples tested were: 10k, 20k, 30k and the full training set of 60k examples, k means thousands. The slicing were performed with a Python script that was implemented by the researcher. The script is named process_images.py and contains other useful functions that were used, the whole script can be found in appendix 9.

DIGITS inverts all images when the MNIST dataset is downloaded, this is done by the "digits/download_data/mnist.py" file in line 98. In order to ensure equivalently of the datasets among the different frameworks, this line must either be commented before downloading a dataset or one can use the inversion function that is implemented in process_images.py.

When it comes to the Flower photos no slicing were performed since the dataset is already small with only 3670 examples. As mentioned the images in this dataset have different sizes so all images were resized to: 64x64 and 128x128. This was performed with the resize_images function defined in process_images.py.

The last preprocessing step concerned the format of the images. DIGITS can directly operate on images when they are stored in directories. It is also possible to directly read data from directories with TensorFlow but it is not the most optimal solution since one would not want to load a whole dataset at once. It is more efficient to convert data into a supported format and then use a pipeline to read data from a file. All

datasets were converted to the standard TensorFlow format which is a TFRecords file containing Example protocol buffers where each buffer contains features about one data example. In process_images.py there is a function called convert_to_tfrecords that reads an image, gets its features, stuffs it in an Example protocol buffer, serializes the protocol buffer to a string and then writes the string to a TFRecords file. For more details read TensorFlows API.

5.4 DCGANs specifications

Two DCGANs, implemented with TensorFlow, were used during the experimentation, the first was a conditional DCGAN used on the MNIST and Fashion-MNIST datasets. The discriminator had two convolutional layers followed by two fully connected layers. Batch normalization was used in the second and third layer. Furthermore, the activation was LeakyReLU on all layers except the last one which was Sigmoid. The generator had two fully connected layers followed by two "deconvolution" layers, here deconvolution means the transpose of 2-D convolution. Batch normalization was used in all layers expect the last one. The first three layers used ReLU activation while the fourth layer used sigmoid. Both the discriminator and the generator had a 5x5 filters with a stride of 2. All details such as output shape and model parameters, i.e. weights and biases, are given by table 5.1 and table 5.2.

The second DCGAN was not conditioned and was used on the Flower photos dataset. The discriminator had four convolutional layers followed by one fully connected layer. Batch normalization was used in the second, third and fourth layer. Furthermore, the activation was LeakyReLU on all layers except the last one which was sigmoid. The generator started with a fully connected layer followed by four "deconvolution" (transpose) layers. Batch normalization was used in all layers expect the last one. The first four layers used ReLU activation while the fifth layer used Tanh. Both the discriminator and the generator had a 5x5 filters with a stride of 2. All details such as output shape and model parameters, i.e. weights and biases, are given by table 5.3 and table 5.4.

Weights in convolutional layers were initialized with a truncated normal distribution initializer with a standard deviation of 0.02, all other layers used a random normal initializer with a standard deviation of 0.02. All biases were initialized to 0. The decay for the moving average for the batch normalizations was set to 0.9, the epsilon was set to 10^{-5}. Every network used ADAM optimizer with the momentum term β_1 set to 0.5 and the learning set to 0.0002. Additionally, the loss functions were the sigmoid cross entropy with logits. In the LeakyReLU activation, the slope of the leak was set to 0.2. Lastly, the training process was balanced by making two training steps for the generator for each training step made for the discriminator. Most of the configurations were adopted both from the DCGAN paper and from implementations of DCGANs by its authors found in GitHub.

Table 5.1: *The details of the discriminator in the cDCGAN when the receptive field of the filters is 5x5 and the stride is 2, the batch size is excluded.*

Layer		Output Shape	Parameters
[x(28,28,1), y(10)] -> h_0			
	conv2d_1	14, 14, 11	3036 ($5 \times 5 \times 11 \times 11 + 11$)
	leaky_relu_1	14, 14, 11	0
	concat_1	14, 14, 21	0
h_0 -> h_1			
	conv2d_2	7, 7, 74	38924 ($5 \times 5 \times 21 \times 74 + 74$)
	batch_normalization_1	7, 7, 74	148 (74 + 74)
	leaky_relu_2	7, 7, 74	0
	reshape_1	3626	0
	concat_2	3636	0
h_1 -> h_2			
	fully_connected_1	1024	3724288 ($3636 \times 1024 + 1024$)
	batch_normalization_2	1024	2048 (1024 + 1024)
	leaky_relu_3	1024	0
	concat_3	1034	0
h_2 -> h_3			
	fully_connected_2	1	1035 ($1034 \times 1 + 1$)
	sigmoid_1	1	0

Total params: 3,769,479

Table 5.2: *The details of the generator in the cDCGAN when the receptive field of the filters is 5x5 and the stride is 2, the batch size is excluded.*

Layer		Output Shape	Parameters
[z(100), y(10)] -> h_0			
	fully_connected_1	1024	113664 ($110 \times 1024 + 1024$)
	batch_normalization_1	1024	2048 (1024 + 1024)
	relu_1	1024	0
	concat_1	1034	0
h_0 -> h_1			
	fully_connected_2	6272	6491520 ($1034 \times 6272 + 6272$)
	batch_normalization_2	6272	12544 (6272 + 6272)
	relu_2	6272	0
	reshape_1	7, 7, 128	0
	concat_2	7, 7, 138	0
h_1 -> h_2			
	conv2d_transpose_1	14, 14, 128	441728 ($5 \times 5 \times 128 \times 138 + 128$)
	batch_normalization_3	14, 14, 128	256 (128 + 128)
	relu_3	14, 14, 128	0
	concat_3	14, 14, 138	0
h_2 -> h_3			
	conv2d_transpose_2	28, 28, 1	3451 ($5 \times 5 \times 1 \times 138 + 1$)
	sigmoid_1	28, 28, 1	0

Total params: 7,065,211

Table 5.3: *The details of the discriminator in the DCGAN when the receptive field of the filters is 5x5 and the stride is 2, the batch size is excluded.*

Layer		Output Shape	Parameters
[x(64,64,3)] -> h_0			
	conv2d_1	32, 32, 64	4864 ($5 \times 5 \times 3 \times 64 + 64$)
	leaky_relu_1	32, 32, 64	0
h_0 -> h_1			
	conv2d_2	16, 16, 128	204928 ($5 \times 5 \times 64 \times 128 + 128$)
	batch_normalization_1	16, 16, 128	256 (128 + 128)
	leaky_relu_2	16, 16, 128	0
h_1 -> h_2			
	conv2d_3	8, 8, 256	819456 ($5 \times 5 \times 128 \times 256 + 256$)
	batch_normalization_2	8, 8, 256	512 (256 + 256)
	leaky_relu_3	8, 8, 256	0
h_2 -> h_3			
	conv2d_4	4, 4, 512	3277312 ($5 \times 5 \times 256 \times 512 + 512$)
	batch_normalization_3	4, 4, 512	1024 (512 + 512)
	leaky_relu_4	4, 4, 512	0
	reshape_1	8192	0
h_3 -> h_4			
	fully_connected_1	1	8193 ($8192 \times 1 + 1$)
	sigmoid_1	1	0

Total params: 4,316,545

Table 5.4: *The details of the generator in the DCGAN when the receptive field of the filters is 5x5 and the stride is 2, the batch size is excluded.*

DCGAN Generator with 5x5 filter and a stride of 2

Layer		Output Shape	Parameters
[z(100)] -> h_0			
	fully_connected_1	8192	827392 ($100 \times 8192 + 8192$)
	reshape_1	4, 4, 512	0
	batch_normalization_1	4, 4, 512	1024 (512 + 512)
	relu_1	4, 4, 512	0
h_0 -> h_1			
	conv2d_transpose_1	8, 8, 256	3277056 ($5 \times 5 \times 256 \times 512 + 256$)
	batch_normalization_2	8, 8, 256	512 (256 + 256)
	relu_2	8, 8, 256	0
h_1 -> h_2			
	conv2d_transpose_2	16, 16, 128	819328 ($5 \times 5 \times 128 \times 256 + 128$)
	batch_normalization_3	16, 16, 128	256 (128 + 128)
	relu_3	16, 16, 128	0
h_2 -> h_3			
	conv2d_transpose_3	32, 32, 64	204864 ($5 \times 5 \times 64 \times 128 + 64$)
	batch_normalization_4	32, 32, 64	128 (64 + 64)
	relu_4	32, 32, 64	0
h_4 <- h_3			
	conv2d_transpose_4	64, 64, 3	4803 ($5 \times 5 \times 3 \times 64 + 3$)
	tanh_1	64, 64, 3	0

Total params: 5,135,363

5.5 LeNet CNN specifications

A LeNet CNN classifier pre-implemented with Caffe's .prototxt format in DIGITS was used to classify both real and synthetic images of MNIST and Fashion-MNIST. An ADAM solver (optimizer) with base learning rate at 0.001 and an exponential decay learning rate policy was used, gamma was set to 0.95. Figure 5.1 gives all details about the network architecture.

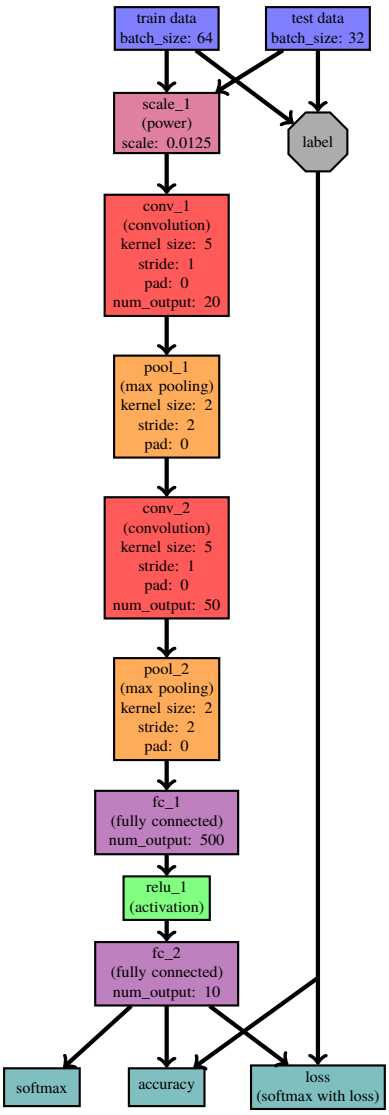

Figure 5.1: *The architecture of the LeNet CNN classifier used to classify MNIST and Fashion-MNIST.*

5.6 Experimental Setup

Two types of experiments were set up, one to generate synthetic images of the three datasets: MNIST, Fashion-MNIST and Flower photos. And a second to measure the accuracies of real, synthetic and a mix of real and synthetic images of MNIST and Fashion-MNIST.

5.6.1 Generation of synthetic images

With this experiment the researcher generated synthetic images using the cDCGAN and DCGAN. The experiment was run in phases, each phase corresponds to a change of the number of epochs to train, the size of a batch or the number of training examples (MNIST and Fashion-MNIST). The number of training

examples for the Flower photos was always the full set of 3670 images, instead the dimension of the images was different: 64x64x3 and 128x128x3. Table 5.5 and 5.6 details all phases. The dependent variable in this experiment is the visual quality of the generated images. The independent variable is the number of training examples (MNIST and Fashion-MNIST) or the dimensions of the images (Flower photos).

Table 5.5: *The different experiment phases regarding the generation of synthetic images of MNIST and Fashion-MNIST using a cDCGAN. The letter k indicates that the numbers are in thousands.*

Dataset	Phase	Training Examples	Epochs	Batch Size
MNIST, Fashion-MNIST				
	A	10k	25	64
	B	10k	100	64
	C	10k	100	100
	D	20k	25	64
	E	20k	100	64
	F	20k	100	100
	G	30k	25	64
	H	30k	100	64
	I	30k	100	100
	J	60k	25	64
	K	60k	100	64
	L	60k	100	100

Table 5.6: *The different experiment phases regarding the generation of synthetic images of Flower photos using a DCGAN.*

Dataset	Image Size	Epochs	Batch Size
Flowers photo			
	64x64x3	500	64
	64x64x3	1000	64
	64x64x3	500	128
	64x64x3	1000	128
	128x128x3	500	64
	128x128x3	1000	64

5.6.2 Measurement of accuracy

With this experiment the researcher measured the accuracies of LeNet CNN classifiers when they were trained on different datasets. While there are many different performance measures for classification tasks, the researcher decided to stick to the accuracy because it is the most common and simplest measure to evaluate a classifier. Additionally, the accuracy is a good measure to use when the class distributions in a dataset are balanced, which is the case for MNIST and Fashion-MNIST.

Six datasets of MNIST were used, the first dataset contained 30k real images, the second contained 30k synthetic images and the third contained 60k images (30k real + 30k synthetic). The three other datasets

contained, respectively, 60k real images, 60k synthetic images and 120k images (60k real + 60k synthetic). The synthetic images was derived from the previous experiment from a certain phase, as stated in table 5.7. The real and synthetic datasets were merged with a function called merge_datasets implemented in the process_images.py script. The exact same proceeding was performed with datasets of Fashion-MNIST. The test sets from MNIST and Fashion-MNIST was used to measure the accuracies, the classifiers were never trained on test sets. The dependent variable in this experiment is the accuracy while the independent variables are the number of training examples and the distribution of the images (real or synthetic).

Table 5.7: *The different datasets used to train LeNet CNN classifiers in order to measure the accuracy. Note that every dataset with synthetic images have a specified phase which indicates the training specifications of the cDCGAN, check table 5.5. The letter k indicates that the numbers are in thousands.*

Dataset		Training Examples	Epochs
MNIST:			
	real_30k	30k	500
	synthetic_30k_phase_H	30k	500
	real_30k_synthetic_30k_phase_H	60k	500
	real_60k	60k	500
	synthetic_60k_phase_K	60k	500
	real_60k_synthetic_60k_phase_K	120k	500
Fashion-MNIST:			
	real_30k	30k	500
	synthetic_30k_phase_H	30k	500
	real_30k_synthetic_30k_phase_H	60k	500
	real_60k	60k	500
	synthetic_60k_phase_K	60k	500
	real_60k_synthetic_60k_phase_K	120k	500

6 RESULTS

In this chapter the results from the experiments that are described in the previous chapter are presented. Firstly, the images from the "generation of synthetic images" experiment are shown, and secondly, the accuracies from the "measurement of accuracy" experiments are presented.

6.1 Generation of synthetic images

Samples of generated images are presented in the subsections below, each subsection corresponds to one of the datasets: MNIST, Fashion-MNIST and Flower photos. Additionally, samples of randomly selected ground truth images are added in the beginning of each subsection.

6.1.1 MNIST

Here, synthetic images of MNIST are presented. Figure 6.1 shows three samples of randomly selected ground truth images. Figure 6.2, 6.3, 6.4 and 6.5 shows samples of generated images when the number of training examples are, respectively, 10k, 20k, 30k and 60k.

Figure 6.1: *Three samples of randomly selected ground truth training examples from MNIST. Each row represents a class.*

Figure 6.2: *Three samples of synthetic images generated by the cDCGAN when trained on 10.000 training examples.* **Left:** *25 training epochs with a batch size of 64.* **Middle:** *100 training epochs with a batch size of 64.* **Right:** *100 training epochs with a batch size of 100.*

Figure 6.3: *Three samples of synthetic images generated by the cDCGAN when trained on 20.000 training examples.* **Left:** *25 training epochs with a batch size of 64.* **Middle:** *100 training epochs with a batch size of 64.* **Right:** *100 training epochs with a batch size of 100.*

Figure 6.4: *Three samples of synthetic images generated by the cDCGAN when trained on 30.000 training examples.* **Left:** *25 training epochs with a batch size of 64.* **Middle:** *100 training epochs with a batch size of 64.* **Right:** *100 training epochs with a batch size of 100.*

Figure 6.5: *Three samples of synthetic images generated by the cDCGAN when trained on 60.000 training examples.* **Left:** *25 training epochs with a batch size of 64.* **Middle:** *100 training epochs with a batch size of 64.* **Right:** *100 training epochs with a batch size of 100.*

6.1.2 Fashion-MNIST

Here, synthetic images of Fashion-MNIST are presented. Figure 6.6 shows three samples of randomly selected ground truth images. Figure 6.7, 6.8, 6.9 and 6.10 shows samples of generated images when the number of training examples are, respectively, 10k, 20k, 30k and 60k.

Figure 6.6: *Three samples of randomly selected ground truth training examples from Fashion-MNIST. Each row represents a class.*

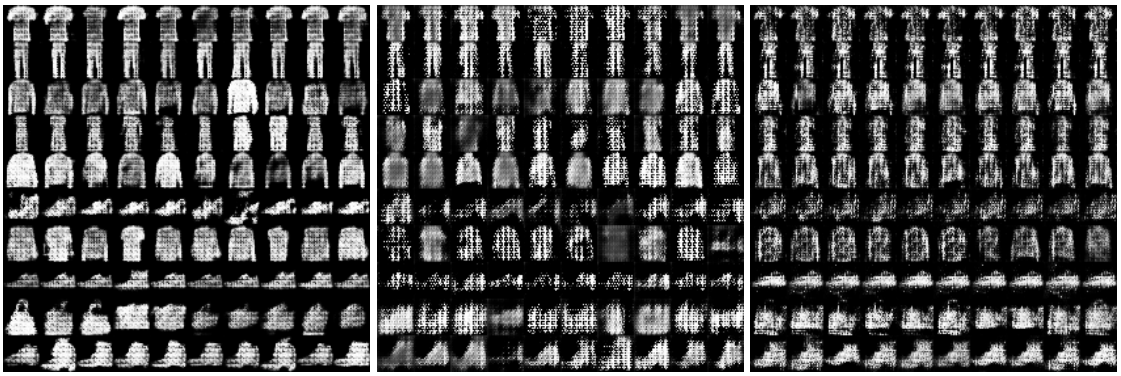

Figure 6.7: *Three samples of synthetic images generated by the cDCGAN when trained on 10.000 training examples.* **Left:** *25 training epochs with a batch size of 64.* **Middle:** *100 training epochs with a batch size of 64.* **Right:** *100 training epochs with a batch size of 100.*

Figure 6.8: *Three samples of synthetic images generated by the cDCGAN when trained on 20.000 training examples.* **Left:** *25 training epochs with a batch size of 64.* **Middle:** *100 training epochs with a batch size of 64.* **Right:** *100 training epochs with a batch size of 100.*

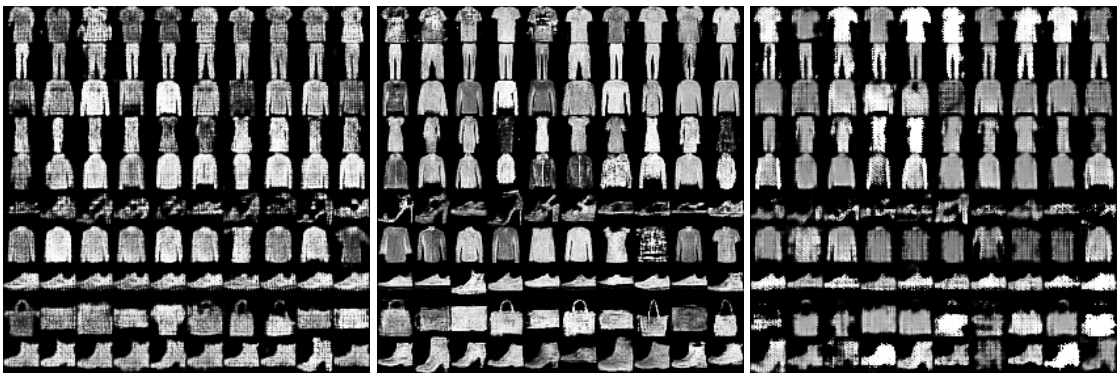

Figure 6.9: *Three samples of synthetic images generated by the cDCGAN when trained on 30.000 training examples.* **Left:** *25 training epochs with a batch size of 64.* **Middle:** *100 training epochs with a batch size of 64.* **Right:** *100 training epochs with a batch size of 100.*

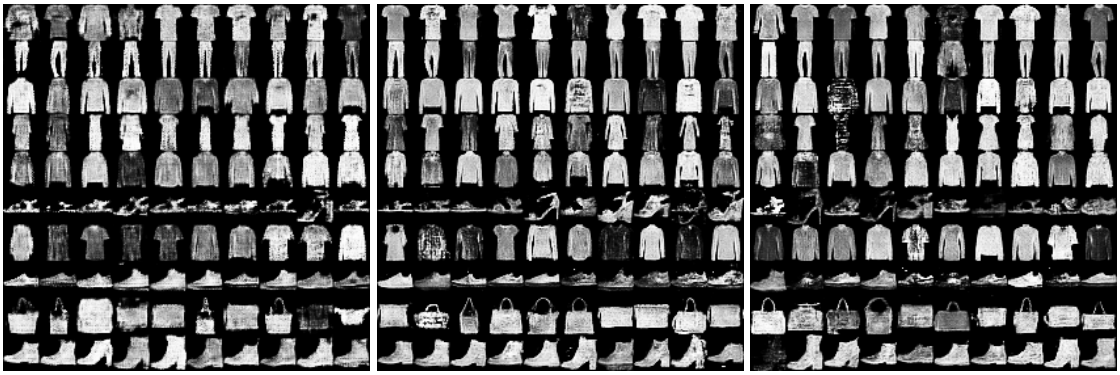

Figure 6.10: *Three samples of synthetic images generated by the cDCGAN when trained on 60.000 training examples.* **Left:** *25 training epochs with a batch size of 64.* **Middle:** *100 training epochs with a batch size of 64.* **Right:** *100 training epochs with a batch size of 100.*

6.1.3 Flower photos

Here, synthetic images of Flower photos are presented. Figure 6.11 shows 72 randomly selected ground truth images of size 64x64x3, figure 6.12 shows 18 randomly selected ground truth images of size 128x128x3. Figure 6.13, 6.14, 6.15 and 6.16 shows generated images from different training settings when trained on 64x64x3 images. Lastly, figure 6.17 and 6.18 shows generated images for different training settings when trained on 128x128x3 images.

Figure 6.11: *A sample of randomly selected ground truth examples from Flower photos, each image is of size 64x64x3. Flower photos are licensed under the Creative Commons By-Attribution License, available at:* https://creativecommons.org/licenses/by/2.0/

Figure 6.12: *A sample of randomly selected ground truth examples from Flower photos, each image is of size 128x128x3. Flower photos are licensed under the Creative Commons By-Attribution License, available at:* https://creativecommons.org/licenses/by/2.0/

Figure 6.13: *A sample of synthetic images generated by the DCGAN when trained on Flower photos, the image size was 64x64x3, the number of epochs was 500 and the batch size was 64.*

Figure 6.14: *A sample of synthetic images generated by the DCGAN when trained on Flower photos, the image size was 64x64x3, the number of epochs was 1000 and the batch size was 64.*

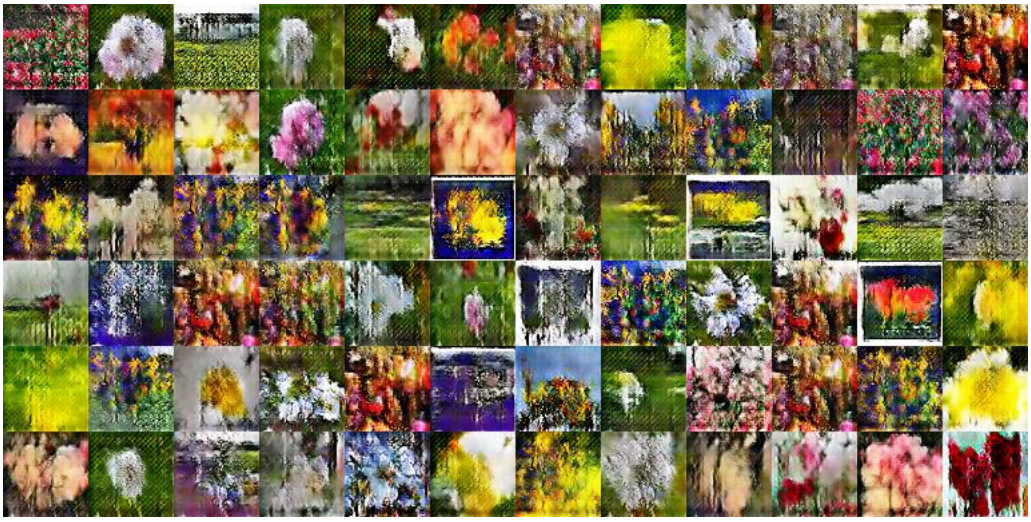

Figure 6.15: *A sample of synthetic images generated by the DCGAN when trained on Flower photos, the image size was 64x64x3, the number of epochs was 500 and the batch size was 128.*

Figure 6.16: *A sample of synthetic images generated by the DCGAN when trained on Flower photos, the image size was 64x64x3, the number of epochs was 1000 and the batch size was 128.*

Figure 6.17: *A sample of synthetic images generated by the DCGAN when trained on Flower photos, the image size was 128x128x3, the number of epochs was 500 and the batch size was 64.*

Figure 6.18: *A sample of synthetic images generated by the DCGAN when trained on Flower photos, the image size was 128x128x3, the number of epochs was 1000 and the batch size was 64.*

6.2 Measurement of accuracy

The test accuracies as well as the train & test losses for MNIST and Fashion-MNIST from the classification experiments are presented below. The accuracies and the test losses were measured using the test set from each dataset. The test set was never used for training, only for validation.

6.2.1 MNIST

Figure 6.19 shows the accuracies when 30k real, 30k synthetic and 60k (30 real + 30 synthetic) images are used for training. Figure 6.20 shows the losses for both the training and testing. Furthermore, figure 6.21 shows the accuracies when 60k real, 60k synthetic and 120k (60 real + 60 synthetic) images are used for training, and figure 6.22 shows their losses.

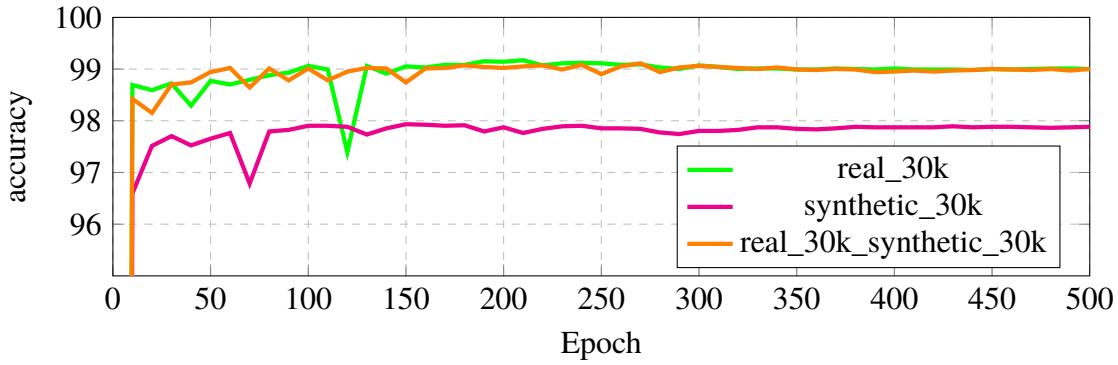

Figure 6.19: *Plots of the accuracies during testing of LeNet CNN classifiers when trained on MNIST.* **Green:** *training on 30.000 real images.* **Magenta:** *training on 30.000 synthetic images.* **Orange:** *training on a mix of 30.000 real and 30.000 synthetic images.*

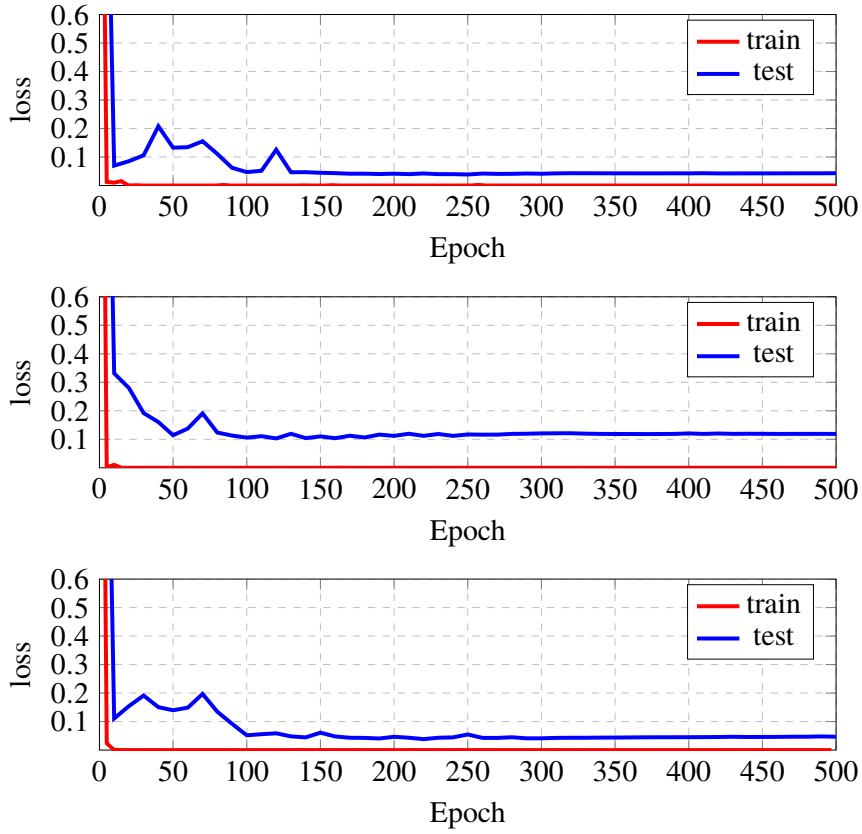

Figure 6.20: *Plots of the loss values during training and testing of LeNet CNN classifiers when trained on MNIST.* **Top:** *training on 30.000 real images.* **Middle:** *training on 30.000 synthetic images.* **Bottom:** *training on a mix of 30.000 real and 30.000 synthetic images.*

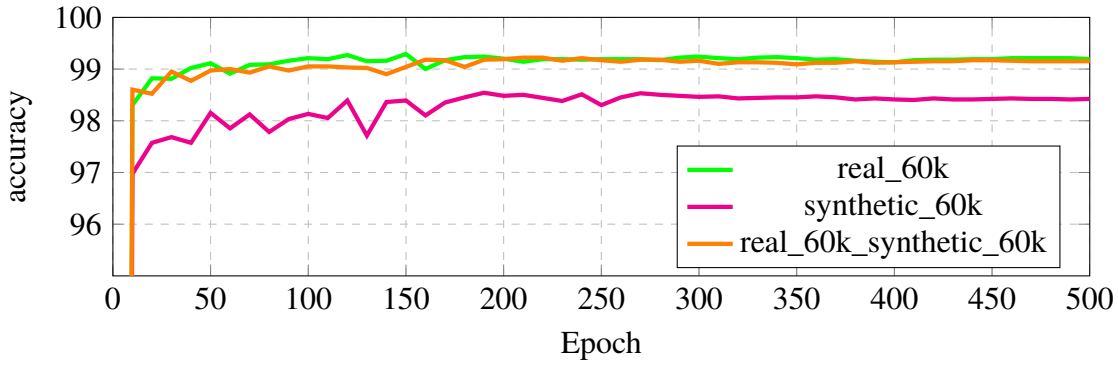

Figure 6.21: *Plots of the accuracies during testing of LeNet CNN classifiers when trained on MNIST.* **Green:** *training on 60.000 real images.* **Magenta:** *training on 60.000 synthetic images.* **Orange:** *training on a mix of 60.000 real and 60.000 synthetic images.*

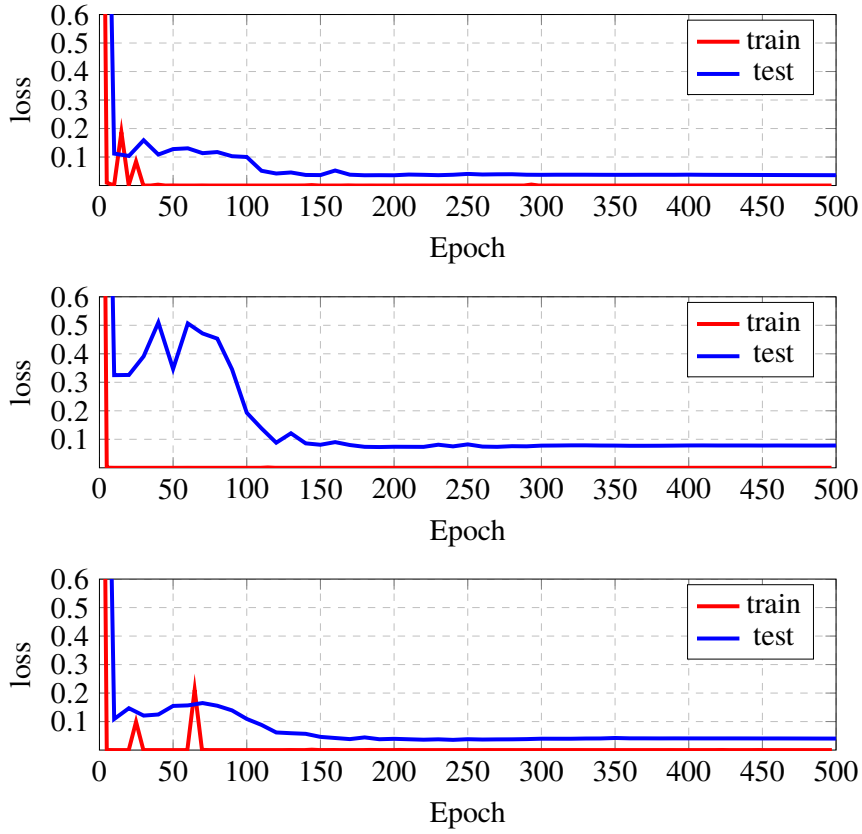

Figure 6.22: *Plots of the loss values during training and testing of LeNet CNN classifiers when trained on MNIST.* **Top:** *training on 60.000 real images.* **Middle:** *training on 60.000 synthetic images.* **Bottom:** *training on a mix of 60.000 real and 60.000 synthetic images.*

6.2.2 Fashion-MNIST

Figure 6.23 shows the accuracies when 30k real, 30k synthetic and 60k (30 real + 30 synthetic) images are used for training. Figure 6.24 shows the losses for both the training and testing. Furthermore, figure 6.25 shows the accuracies when 60k real, 60k synthetic and 120k (60 real + 60 synthetic) images are used for training, and figure 6.26 shows their losses.

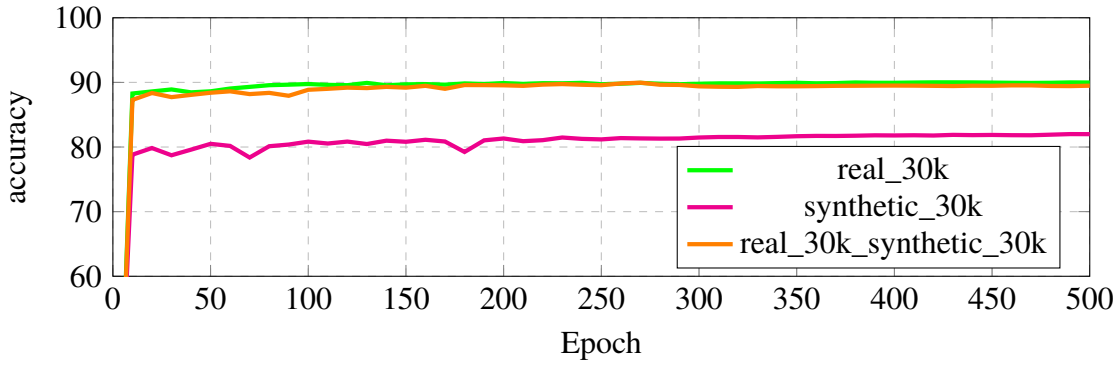

Figure 6.23: *Plots of the accuracies during testing of LeNet CNN classifiers when trained on Fashion-MNIST.* **Green:** *training on 30.000 real images.* **Magenta:** *training on 30.000 synthetic images.* **Orange:** *training on a mix of 30.000 real and 30.000 synthetic images.*

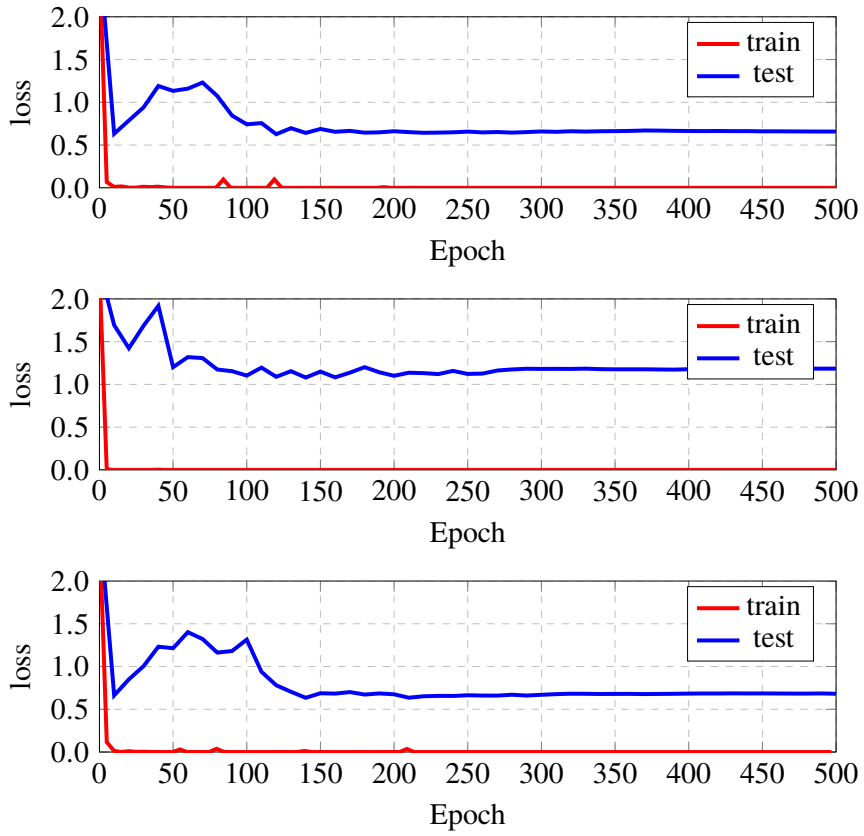

Figure 6.24: *Plots of the loss values during training and testing of LeNet CNN classifiers when trained on Fashion-MNIST.* **Top:** *training on 30.000 real images.* **Middle:** *training on 30.000 synthetic images.* **Bottom:** *training on a mix of 30.000 real and 30.000 synthetic images.*

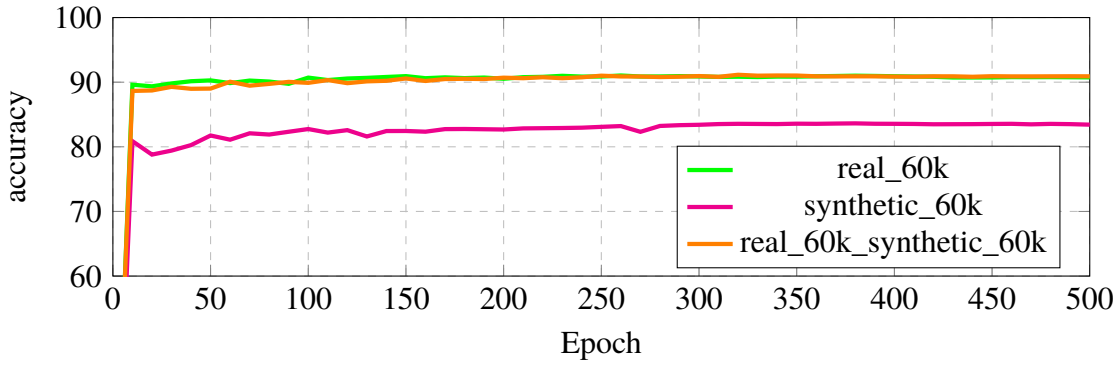

Figure 6.25: *Plots of the accuracies during testing of LeNet CNN classifiers when trained on Fashion-MNIST.* **Green:** *training on 60.000 real images.* **Magenta:** *training on 60.000 synthetic images.* **Orange:** *training on a mix of 60.000 real and 60.000 synthetic images.*

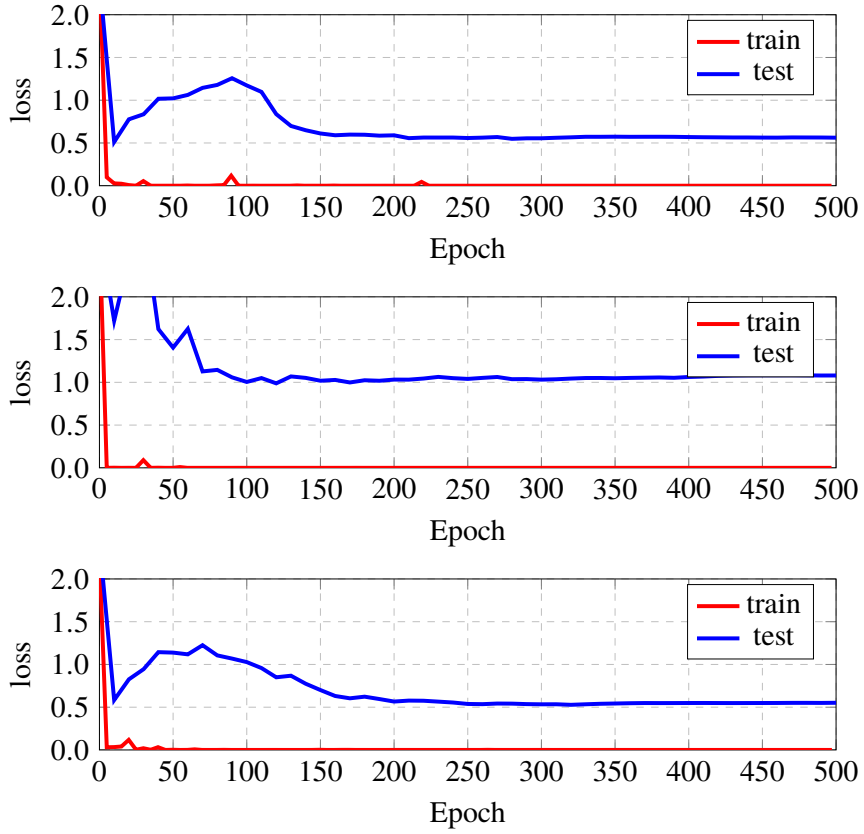

Figure 6.26: *Plots of the loss values during training and testing of LeNet CNN classifiers when trained on Fashion-MNIST.* **Top:** *training on 60.000 real images.* **Middle:** *training on 60.000 synthetic images.* **Bottom:** *training on a mix of 60.000 real and 60.000 synthetic images.*

7 ANALYSIS AND DISCUSSION

During the experiments a lot of data could be observed, the data were logged in order to be carefully analyzed after the executions of the experiments. The interpretation of the data and other important aspects are discussed below.

7.1 Generation of synthetic images

As expected, the MNIST dataset was easy to generate synthetic samples of. Actually, the results were surprisingly good even when the cDCGAN only had 10k examples to train on. As seen in figure 6.2 the generated samples are quite alike the original samples but there are still visible errors that distinguish the samples. Of course when the training examples increases the cDCGAN becomes better at generating synthetic samples. The researcher observed that setting the training epochs to 100 with a batch size of 64 results in a cDCGAN that generates samples that are very similar to ground truth samples. Best results were obtained when those settings were used together with the full training set of 60k examples, some generated classes had minor visible errors but not all.

The Fashion-MNIST was also a relatively easy dataset for the cDCGAN to generate similar samples of. Not surprisingly, the same settings that were used with the regular MNIST provided very good results. However, the researcher noticed that the batch size could greatly affect the visual quality of the generated images. This can be seen in figure 6.9 when training for 100 epochs, it is clear that when the batch size is set to 64 the results have superior quality compared to when the batch size is set to 100. Additionally, when the training set is small the number of training epochs also greatly affects the quality of the results, this can be examined in figure 6.7.

When it comes to the Flower photos dataset, the DCGAN had some complications to generate synthetic images identical to the ground truth. Several reasons are behind the complications. First of all, many images in the dataset have a lot of noise in terms of other objects than flowers are included in the images. If the dataset is explored, one can find objects such as a human, animal, tractor, bike, spoon, buildings, statue, cake and many other objects. Moreover, some images contain one flower while others can contain many flowers or a field of flowers. Furthermore, the angles of the flowers as well as the background of the images differ very much from image to image. All these aspects make it difficult for the DCGAN to collect the features of the flowers. Secondly, the size of the dataset also plays a part. For the MNIST and Fashion-MNIST datasets, 60.000 examples are available while for the Flower photos dataset there are only 3670 examples. The last reason concerns the capacity of the DCGAN assuming that it is not already sufficient. The DCGAN may need some more fine-tuning in order to become better but due to lack of time the researcher did not carry on with it. Comparing the results from the two different sizes used, 64x64x3 and 128x128x3, training a DCGAN on larger images seemed to increase the visual quality of the images. One reason can be because details are neglected in images with smaller size.

7.2 Measurement of accuracy

Regarding the accuracies, none of the experiments lead to an increased accuracy. This applies to both the MNIST and Fashion-MNIST dataset. At best, the accuracies from the mix datasets reached the accuracies of the real datasets. This was not because the generated samples were not alike the ground truth, the generated samples of MNIST and Fashion-MNIST were very good. The main reason is that the datasets are relatively easy for the CNN to classify, especially the MNIST. The researcher initially assumed that the accuracy is highly dependent on the number of training examples, but figure 7.1 and 7.2 clearly show that this is not the case on the LeNet CNN classifier when trained on MNIST and Fashion-MNIST. The figures show a comparison of accuracies when the number of real training examples of MNIST and Fashion-MNIST respectively are doubled. The accuracy does not majorly increase even when the number

of training examples are doubled with real examples, so why would synthetic examples increase the accuracy? A better case would be if the accuracy is dependent on the number of training examples, then one can compare how an increment of both real and synthetic examples would affect the accuracy.

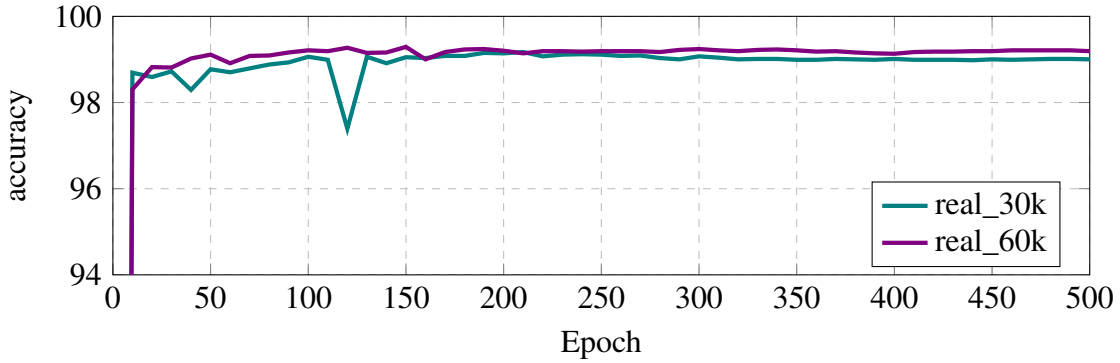

Figure 7.1: *Plots of the accuracies of LeNet CNN classifiers when trained on only real images of MNIST.* **Teal:** *training on 30.000 real images.* **Violet:** *training on 60.000 real images. The plots shows that the accuracy is not highly dependent on the number of training examples.*

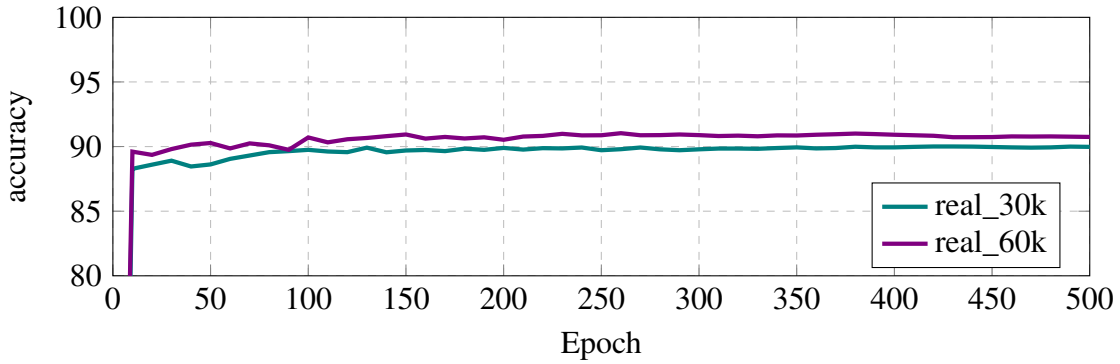

Figure 7.2: *Plots of the accuracies of LeNet CNN classifiers when trained on only real images of Fashion-MNIST.* **Teal:** *training on 30.000 real images.* **Violet:** *training on 60.000 real images. The plots shows that the accuracy is not highly dependent on the number of training examples.*

Comparing the accuracies from real and synthetic datasets is a better way to measure how similar the generated samples are to the ground truth samples. The accuracies of the synthetic datasets of MNIST were approximately 1% lower than the accuracies of the real and mix datasets. This is another indication that the cDCGAN generated very similar samples to the ground truth samples. For Fashion-MNIST the difference was about approximately 7%, not as good as MNIST but still good enough.

When it comes to the loss functions, nothing very special about MNIST was spotted. The loss is of course higher for synthetic datasets since the accuracy is lower. The test loss functions were initially unstable for whatever dataset used but after approximately 120 epochs all loses stabilized expect for the MNIST 30k synthetic dataset which stabilized earlier.

The test loss values for Fashion-MNIST does not converge to zero even when only real training examples are used meanwhile the training loss values do converge to zero. This means that the classifier performs good at training but not when it is tested on unseen data, which is a sign of the overfitting problem described in section 2.5. However, in this case it is not a big issue since the test accuracies are actually high, around 90%.

Again here, the same behaviour can be seen with the test loss values being initially unstable but stabilizes after approximately 150 epochs, the synthetic datasets however stabilizes much earlier especially the one with 30k synthetic examples. So why do the synthetic datasets have faster stabilization and less oscillation? Because the synthetic datasets are more noisy than the real datasets especially synthetic datasets of Fashion-MNIST. The researcher also discovered that around epoch 230 the test loss for the mix of 60k real + 60k synthetic examples of Fashion-MNIST becomes slightly lower than the loss of the 60k real examples, see figure 7.3. Again here, the reason is that the synthetic images includes noise, and noise is known to prevent overfitting. This can mean that a GAN may be trained to generate noisy samples in order to prevent overfitting or to make, for instance, classifiers more stable. This is of course something that must be further investigated but is outside the scope of this work.

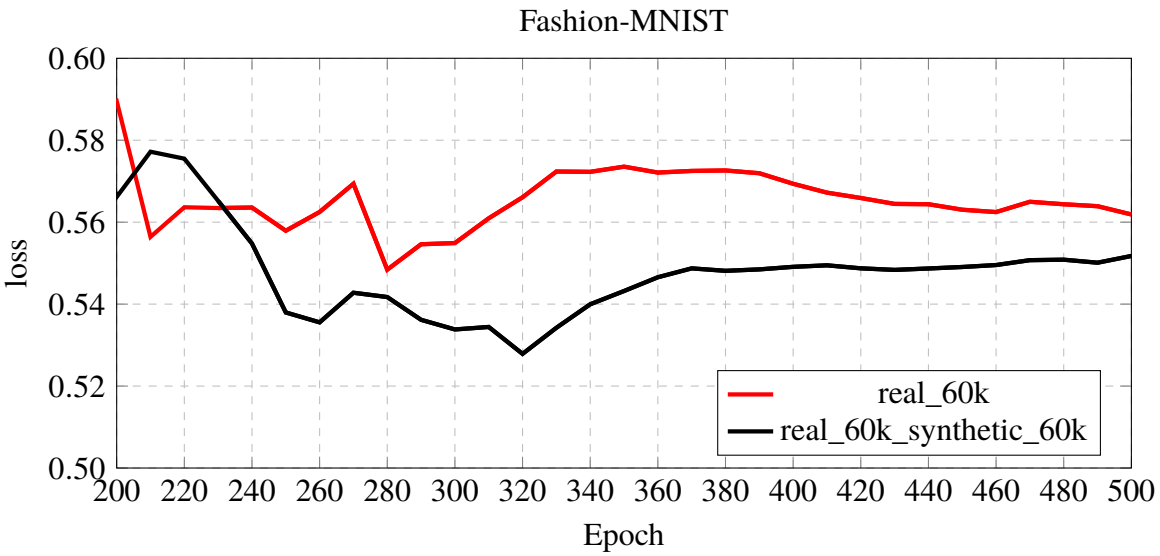

Figure 7.3: *Plots of the loss values during testing of two LeNet CNN classifiers when trained on Fashion-MNIST.* **Red:** *training on 60.000 real images.* **Black:** *training on a mix of 60.000 real and 60.000 synthetic images. The mix of real and synthetic examples results in a lower loss.*

7.3 Challenges and Limitations

The challenges with this study was mainly to build decent DCGANs and make them generate high quality samples that are similar to the ground truth samples. Generally, deep learning models have a lot of model parameters as well as a plenty of hyper-parameters to be tuned. Not only do appropriate layers with relevant number of nodes, type of activations for each layer, the optimizer etc. all affects the performance of a DL model, a DCGAN contains two DL models which further complicates the already complicated training process of a DL model. While the researcher followed the recommendations from the DCGAN paper [31], different datasets may require different settings. This made the fine tuning of a DL model very time consuming due to the many tests that needed to be performed.

Speaking of time, the time required for each training session was the foremost limitation to this study. One training session for MNIST or Fashion-MNIST could require 2 hours while for the Flower photos the time could rise up to 40 hours per session. The extremely time consuming training sessions took up most of the time dedicated for this study. This further complicated the fine-tuning of the DCGAN on the Flower photos.

Another challenge was the data preprocessing, the datasets needed to be resized, sliced, merged and transformed to an appropriate format. The process_images.py script made it straightforward to process the datasets but the implementation of it took time.

7.4 Reliability and Validity

A fundamental issue that concerns the results from an experiment is that they must be both reliable and valid. Reliability refers to the repeatability of findings, that is when a phenomenon is measured twice, the outcome shall be very similar. Additionally, objective measures are more reliable than subjective measures [59]. The researcher ensure that the results from the experiments are reliable since every experiment was run at least twice, and every time an experiment was run very similar outcomes were obtained. Anyone that performs the experiments that have been performed in this study using the same equipment (hardware and software), datasets and configurations should obtain very similar outcome.

The validity can be divided into two types, internal and external validity [60]. The internal validity refers to threats that can affect the independent variable with respect to causality. For instance, instruments or procedures used in a measurement. The external validity is concerned with generalization, that is can the results of a study be generalized beyond the immediate study? The threats for different research methods vary with regard to these two types of validity. In this study, structured and controlled experiments were conducted which often ensures high internal validity. At the same time, this approach may result in low external validity due to the strong structure and control. The results from this study is limited to the datasets used together with the specific configurations used for the different models. The results can be generalized to this limitation but not beyond it.

8 CONCLUSIONS AND FUTURE WORK

In this chapter, the conclusions drawn from the research as well ass possible proposals for future work are presented. Section 8.1 gives the conclusions while section 8.2 proposes future work.

8.1 Conclusions

In this research, the problem of not having sufficient datasets was tackled using GANs to generate synthetic images in order to expand datasets. The lesson learned is that DCGANs are highly capable of generating synthetic samples similar to the ground truth. In some cases the generated samples are even indistinguishable from the ground truth. At the same time DCGANs may require, depending on the dataset, a large quantity of training examples in order to be that good. This does not completely solve the initial problem of not having adequate datasets. For that, a GAN that require a small quantity of examples and at the same time generates high quality synthetic samples similar to the ground truth is needed. However, DCGANs are fully capable of expanding datasets and can be used to complement existing datasets.

Another important aspect to point out is that the data preprocessing plays a big roll on a DCGANs performance, as for almost any ML algorithm. It was clearly observed from the results that the DCGANs could easily generate high quality samples of MNIST and Fashion-MNIST while for the Flower photos there was complications because this dataset is not preprocessed and contains much noise.

When it comes to the accuracy, no improvements could be spotted. This was not because the generated samples were not alike the ground truth, contrariwise, the generated samples of MNIST and Fashion-MNIST were very good. The reason was that the LeNet CNN classifier could easily classify MNIST and Fashion-MNIST. It turned out that even when the real training examples were doubled from 30 to 60 thousands, the accuracy did not excessively increase and thus the accuracy was not highly dependent on the number of training examples. This made the comparison between the only real examples and the mix of real & synthetic examples not good. The comparison would had been better if the accuracy would had been more dependent on the number of training examples.

The drawbacks of a DCGAN are all related to the training process. First of all, DCGANs contains two models which complicates an already complicated situation since the two models must be trained simultaneously. Secondly, the training process is very sensitive and problems can easily occur if the training process is not balanced between the discriminator and the generator. The discriminator can overpower the generator if it classifies synthetic data with absolute certainty. And the generator can overpower the discriminator if it discovers some weakness in the discriminator. Thirdly, DL models performs massive amounts of computations during the training process, a DCGAN contains two models which means that even more computations are performed so large computational power is required. And even with large computational power the training process will be very time-consuming. This makes it difficult to fine-tune a model.

8.2 Future Work

While many different adaptions, tests and experiments have been performed, there are still many left for the future. Especially the fine-tuning of the DCGAN on Flower photos or other similar but less noisy datasets is interesting since there were no time left for the researcher to continue. Also finding other ML tasks than classification where the accuracy is very dependent to the number of examples available in order to investigate how synthetic examples generated by a GAN affects the accuracy. The author suggests that before a dataset or a task is selected, one should examine if the accuracy is highly dependent on the number of training examples. Another proposal is to investigate whether a GAN can be trained to generate noisy samples in order to prevent overfitting.

Other proposals for future work concerns the analysis of other GAN models then the one used in this research. The development of GANs is still ongoing and the area is expanding so new models and classes are published every now and then. A recently published class of GAN called BEGAN (Boundary Equilibrium GAN) [61] is interesting, the researchers behind it promises a more balanced training, a new approximate convergence measure and a fast & stable training where the generated samples have high visual quality. Another very interesting class is the Triple-GAN [62] where the two-player formulation is expanded to a three-player formulation containing a discriminator, generator and a classifier.

REFERENCES

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[2] A. M. Turing, "Computers & thought," ch. Computing Machinery and Intelligence, pp. 11–35, Cambridge, MA, USA: MIT Press, 1995.

[3] T. M. Mitchell, *Machine Learning*. New York, NY, USA: McGraw-Hill, Inc., 1 ed., 1997.

[4] M. Welling, *"A First Encounter with Machine Learning"*. Donald Bren School of Information and Computer Science University of California Irvine, November 2011.

[5] S. Rifai, Y. Bengio, A. C. Courville, P. Vincent, and M. Mirza, "Disentangling factors of variation for facial expression recognition," pp. 808–822, 2012.

[6] R. Rojas, *Neural Networks: A Systematic Introduction*. New York, NY, USA: Springer-Verlag New York, Inc., 1996.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," NIPS'12, (USA), pp. 1097–1105, Curran Associates Inc., 2012.

[8] Y. LeCun, L. D. Jackel, B. Boser, J. S. Denker, H. P. Graf, I. Guyon, D. Henderson, R. E. Howard, and W. Hubbard, "Handwritten Digit Recognition: Applications of Neural Net Chips and Automatic Learning," *IEEE Communication*, pp. 41–46, November 1989. invited paper.

[9] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," pp. 2672–2680, Curran Associates, Inc., 2014.

[10] L. Deng and D. Yu, "Deep Learning: Methods and Applications," *Foundations and Trends in Signal Processing*, vol. 7, no. 3-4, pp. 197–387, 2014.

[11] G. E. Hinton, "Learning multiple layers of representation," *Trends in Cognitive Sciences*, vol. 11, pp. 428–434, Oct 2007.

[12] M. A. Nielsen, "Neural Networks and Deep Learning," *Determination Press*, 2015.

[13] F. Rosenblatt, *The Perceptron, a Perceiving and Recognizing Automaton Project Para*. Cornell Aeronautical Laboratory, Jan 1957.

[14] Stanford University CS231n Convolutional Neural Networks for Visual Recognition, "Neural networks part 1: Setting up the architecture." [Online].Available: `http://cs231n.github.io/neural-networks-1`.

[15] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Neurocomputing: Foundations of research," ch. Learning Representations by Back-propagating Errors, pp. 696–699, Cambridge, MA, USA: MIT Press, 1988.

[16] Y. LeCun, L. Bottou, G. B. Orr, and K. R. Müller, *Efficient BackProp*, pp. 9–50. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998.

[17] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," pp. 807–814, 2010.

[18] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," pp. 1097–1105, Curran Associates, Inc., 2012.

[19] K. Janocha and W. M. Czarnecki, "On loss functions for deep neural networks in classification," vol. 1/2016, pp. 49–59, 2017. Theoretical Foundations of Machine Learning, Jagiellonian University, 13-17 February 2017, Krakow, Poland.

[20] A.-L. Cauchy, "Méthode générale pour la résolution des systèmes d'équations simultanées," *Compte Rendu des S'eances de L'Acad'emie des Sciences XXV*, vol. S'erie A, pp. 536–538, Oct. 1847.

[21] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016.

[22] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *CoRR*, vol. abs/1412.6980, 2014. 3rd International Conference on Learning Representations (ICLR 2015), May 7-9, 2015, San Diego, California, United States.

[23] T. Tieleman and G. Hinton, "Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude." COURSERA: Neural Networks for Machine Learning [Online].Available: `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`, 2012.

[24] M. D. Zeiler, "ADADELTA: an adaptive learning rate method," *CoRR*, vol. abs/1212.5701, 2012.

[25] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[26] P. Lutz, *Early Stopping - But When?*, pp. 53–67. Neural Networks: Tricks of the Trade. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2012.

[27] Stanford University CS231n Convolutional Neural Networks for Visual Recognition, "Convolutional neural networks: Architectures, convolution / pooling layers." [Online].Available: `http://cs231n.github.io/convolutional-networks/`.

[28] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *CoRR*, vol. abs/1603.07285, 2016.

[29] I. J. Goodfellow, "NIPS 2016 Tutorial: Generative Adversarial Networks," *CoRR*, vol. abs/1701.00160, 2017.

[30] M. Mirza and S. Osindero, "Conditional Generative Adversarial Nets," *CoRR*, vol. abs/1411.1784, 2014.

[31] A. Radford, L. Metz, and S. Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," *CoRR*, vol. abs/1511.06434, 2015. 4th International Conference on Learning Representations (ICLR 2016), May 2-4, 2016, San Juan, Puerto Rico.

[32] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning* (F. Bach and D. Blei, eds.), vol. 37 of *Proceedings of Machine Learning Research*, (Lille, France), pp. 448–456, PMLR, 07–09 Jul 2015.

[33] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *CoRR*, vol. abs/1312.6114, 2013.

[34] A. van den Oord, N. Kalchbrenner, and K. Kavukcuoglu, "Pixel recurrent neural networks," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pp. 1747–1756, 2016.

[35] T. Salimans, I. J. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans," in *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pp. 2226–2234, 2016.

[36] A. Shrivastava, T. Pfister, O. Tuzel, J. Susskind, W. Wang, and R. Webb, "Learning from simulated and unsupervised images through adversarial training," in *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pp. 2242–2251, 2017.

[37] J. Zhu, P. Krähenbühl, E. Shechtman, and A. A. Efros, "Generative visual manipulation on the natural image manifold," in *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part V*, pp. 597–613, 2016.

[38] M. Liu and O. Tuzel, "Coupled generative adversarial networks," in *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pp. 469–477, 2016.

[39] X. Chen, X. Chen, Y. Duan, R. Houthooft, J. Schulman, I. Sutskever, and P. Abbeel, "Infogan: Interpretable representation learning by information maximizing generative adversarial nets," in *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pp. 2172–2180, 2016.

[40] E. Wood, T. Baltrusaitis, L. Morency, P. Robinson, and A. Bulling, "Learning an appearance-based gaze estimator from one million synthesised images," in *Proceedings of the Ninth Biennial ACM Symposium on Eye Tracking Research & Applications, ETRA 2016, Charleston, SC, USA, March 14-17, 2016*, pp. 131–138, 2016.

[41] A. Gupta, A. Vedaldi, and A. Zisserman, "Synthetic data for text localisation in natural images," in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pp. 2315–2324, 2016.

[42] X. Peng, B. Sun, K. Ali, and K. Saenko, "Learning deep object detectors from 3d models," in *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*, pp. 1278–1286, 2015.

[43] Z. Wang, J. Yang, H. Jin, E. Shechtman, A. Agarwala, J. Brandt, and T. S. Huang, "Deepfont: Identify your font from an image," in *Proceedings of the 23rd Annual ACM Conference on Multimedia Conference, MM '15, Brisbane, Australia, October 26 - 30, 2015*, pp. 451–459, 2015.

[44] A. Gaidon, Q. Wang, Y. Cabon, and E. Vig, "Virtual worlds as proxy for multi-object tracking analysis," *CoRR*, vol. abs/1605.06457, 2016.

[45] M. V. Giuffrida, H. Scharr, and S. A. Tsaftaris, "ARIGAN: synthetic arabidopsis plants using generative adversarial network," *CoRR*, vol. abs/1709.00938, 2017.

[46] T. Karras, T. Aila, S. Laine, and J. Lehtinen, "Progressive growing of gans for improved quality, stability, and variation," *CoRR*, vol. abs/1710.10196, 2017.

[47] P. D. Leedy and J. E. Ormrod, *Practical Research: Planning and Design*. Pearson Education, 2013.

[48] *Cambridge Advanced Learner's Dictionary*. Shaftesbury Road, Cambridge CB2 8BS, UK: Cambridge University Press, 2008.

[49] Bhawna and Gobind, "Research methodology and approaches," vol. 5, Issue 3 Version 4, pp. 48–51, 2015.

[50] C. Williams, "Research Methods," *Journal of Business Economics Research (JBER)*, vol. 5 No 3, Feb 2011.

[51] S. Rajasekar, P. Philominathan, and V. Chinnathambi, "Research Methodology," *arXiv:physics/0601009*, Jan 2006. arXiv: physics/0601009.

[52] NVIDIA, "NVIDIA CUDA - Compute Unified Device Architecture Programming Guide." [Online].Available: `http://developer.download.nvidia.com/compute/cuda/1.0/NVIDIA_CUDA_Programming_Guide_1.0.pdf`.

[53] NVIDIA, "CUDA Toolkit." [Online].Available: `https://developer.nvidia.com/cuda-toolkit`.

[54] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient Primitives for Deep Learning," *CoRR*, vol. abs/1410.0759, 2014.

[55] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," *CoRR*, vol. abs/1603.04467, 2016.

[56] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," MM '14, (New York, NY, USA), pp. 675–678, ACM, 2014.

[57] NVIDIA, "NVIDIA DIGITS - Interactive Deep Learning GPU Training System." [Online].Available: `https://developer.nvidia.com/digits`.

[58] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms," *arXiv:1708.07747 [cs, stat]*, Aug 2017. arXiv: 1708.07747.

[59] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, and B. Regnell, *Experimentation in Software Engineering*. Springer Berlin Heidelberg, 2012.

[60] D. T. Campbell and J. C. Stanley, *Experimental and Quasi-Experimental Designs for Research*. Houghton Mifflin Company, Boston, 1963.

[61] D. Berthelot, T. Schumm, and L. Metz, "BEGAN: boundary equilibrium generative adversarial networks," *CoRR*, vol. abs/1703.10717, 2017.

[62] C. LI, T. Xu, J. Zhu, and B. Zhang, "Triple generative adversarial nets," in *Advances in Neural Information Processing Systems 30* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), pp. 4091–4101, Curran Associates, Inc., 2017.

PROCESS_IMAGES.PY

```
# Created 2017 by Mousa Zeid Baker at Blekinge Institute of Technology

'''
Description:
        This script was used during a thesis research to process image datasets.
        Six operations can be selected: FUNCTION_MAP = ['convert_to_tfrecords', 'crop_images', 'invert_images', 'merge_datasets', 'reize_images', 'slice_dataset']

        The data directory is expected to have this structure:
                data_dir/label_0/An_image
                data_dir/label_0/image0235
                ...
                data_dir/label_N/Last_image

        where the sub-directory is the unique label associated with the images.
        Use "Python process_images.py --help" to show a help message.
'''

from random import shuffle
from PIL import Image
import PIL.ImageOps
import scipy.misc
import tensorflow as tf
import os
import sys
import numpy as np
import random
import shutil
import argparse

FUNCTION_MAP = ['convert_to_tfrecords', 'crop_images', 'invert_images', 'merge_datasets', 'resize_images', 'slice_dataset']

parser = argparse.ArgumentParser(description="Process your image dataset.")
parser.add_argument('data_dir', default='', help='Path to data directory.')
parser.add_argument('-dd2', '--data_dir2', default='', help='Path to second data directory.', metavar='')
parser.add_argument('-ops', '--operation', choices=FUNCTION_MAP, help='Select an operation {convert_to_tfrecords, crop_images, invert_images, merge_datasets, resize_images, slice_dataset}.', metavar='')
parser.add_argument('-d', '--dest', default='', help='Destination for the new dataset.', metavar='')
parser.add_argument('-s', '--size', default=-1, help='The number of images from each label to copy into the new dataset.', type=int, metavar='')
args = parser.parse_args()
```

```
def find_files(data_dir):
        """
        Build a list of all images files and labels in the data set.
        Args:
                data_dir: string, path to the root directory of images.
        Returns:
                filenames: list of strings; each string is a path to an image
↪   file.
                labels: list of integer; each integer in index i is the ground
↪   truth of filenames in index i.
                unique_labels: list of strings; each string is a label.
        """
        print('Determining list of input files and labels from "%s".' %
        ↪  data_dir)
        unique_labels = []
        for dirname in os.listdir(data_dir):
                if os.path.isdir(os.path.join(data_dir, dirname)):
                        unique_labels.append(dirname)

        unique_labels.sort()
        filenames = []
        labels = []
        for label in unique_labels:
                for file_ in os.listdir(os.path.join(data_dir, label)):
                        filenames.append(os.path.join(data_dir, label, file_))
                        labels.append(unique_labels.index(label))
        print('Found %d files across %d labels inside "%s".' %(len(filenames),
        ↪  len(unique_labels), data_dir))

        # Shuffle the ordering of all image files in order to guarantee
        # random ordering of the images with respect to label in the
        # saved TFRecord files.
        random.seed(12345)
        permutation = np.random.permutation(len(filenames))
        tmp1 = []
        tmp2 = []
        for i in range(len(filenames)):
                tmp1.append(filenames[permutation[i]])
                tmp2.append(labels[permutation[i]])
        filenames = tmp1
        labels = tmp2

        return filenames, labels, unique_labels

def convert_to_tfrecords(filenames, labels, output_name):
        '''
        Process and save list of images in TFRecord file type.
        Args:
                filenames: list of strings; each string is a path to an image
↪   file.
                labels: list of integer; each integer in index i is the ground
↪   truth of filenames in index i.
```

```
                output_name: string, the filename of the TFRecord to create.
        '''
        def _bytes_feature(value):
                return
                 ↪  tf.train.Feature(bytes_list=tf.train.BytesList(value=value))
        def _int64_feature(value):
                return
                 ↪  tf.train.Feature(int64_list=tf.train.Int64List(value=value))

        writer = tf.python_io.TFRecordWriter(output_name)

        count=0
        print('Converting to TFRecords..')
        n=len(filenames)

        for i in range(n):
                count+=1
                # open image
                img = scipy.misc.imread(filenames[i])
                # optional! All preprocessing goes here
                #img = scipy.misc.imresize(img, [64, 64])

                # get image dimensions
                height, width = img.shape[:2]

                im_raw = img.tostring()

                # construct the Example proto object to store our features
                example = tf.train.Example(features=tf.train.Features(
                feature={
                        'image_raw': _bytes_feature([im_raw]),
                        "height": _int64_feature([height]),
                        "width": _int64_feature([width]),
                        "label": _int64_feature([labels[i]])
                }))
                # serialize the example to a string and write it to the disk
                writer.write(example.SerializeToString())

                percentage = (float(i+1)/n)*100
                percentage = round(percentage, 2)
                print('%s%%\r' %(percentage), end='')

        writer.close()
        print('\n\nConverted %s images to TFRecords' %(count))
        print('Saved results to "%s"' % (output_name))

def crop_images(filenames, height=28, width=28):
        '''
        Crop a list of images.
        Args:
                filenames: list of strings; each string is a path to an image
 ↪  file.
```

```
                height: integer; the desired height.
                width: integer, the desired width.
        '''
        ext = 'png'
        left = 0
        upper = 0
        right = width
        lower = height
        print('Cropping images..')
        count=0
        for i in range(len(filenames)):
                percentage = (float(i+1)/len(filenames))*100
                percentage = round(percentage, 2)
                print('%s%%\r' %(percentage), end='')

                left = 0
                upper = 0
                right = width
                lower = height
                img = Image.open(filenames[i])
                imgwidth, imgheight = img.size
                for k in range(int(imgheight/height)):
                        for j in range(int(imgwidth/width)):
                                box = (left, upper, right, lower)
                                cropped_img = img.crop(box)
                                name =
                                ↪  os.path.join(os.path.dirname(filenames[i]),
                                ↪  'crop%s_%s-%s.%s' %(i, k, j, ext))
                                cropped_img.save(name)
                                count+=1

                                left += width
                                right += width
                        upper += height
                        lower += height
                        left = 0
                        right = width
                os.remove(filenames[i])
        print('')
        print('Cropped into %s images.' %(count))

def invert_images(filenames):
        '''
        Invert a list of images.
        Args:
                filenames: list of strings; each string is a path to an image
↪  file.
        '''
        print('Inverting images..')

        for i in range(len(filenames)):
                percentage = (float(i+1)/(len(filenames)))*100
```

```
                percentage = round(percentage, 2)
                print('%s%%\r' %(percentage), end='')

                image = Image.open(filenames[i])
                inverted_image = PIL.ImageOps.invert(image)
                os.remove(filenames[i])
                inverted_image.save(filenames[i])

        print('')
        print('Inverted %s images' %(len(filenames)))

def merge_datasets(data_dir1, data_dir2, dest):
        '''
        Merges two "datasets".
        Args:
                data_dir1: string; the path to the first dataset.
                data_dir2: string; the path to the second dataset.
                dest: string, optional destination path.
        '''
        datasetName = 'merged_' + os.path.basename(data_dir1) + '_' +
        ↪  os.path.basename(data_dir2)
        dir_ = os.path.join(dest, datasetName)
        if not os.path.exists(dir_):
                os.system("mkdir -p %s" %(dir_))

        print('Copying first dataset..')
        unique_labels = []
        for dirname in os.listdir(data_dir1):
                if os.path.isdir(os.path.join(data_dir1, dirname)):
                        unique_labels.append(dirname)
        count=0
        count_progress=0
        for label in unique_labels:
                count_progress+=1
                subDir = os.path.join(dir_, label)
                if not os.path.exists(subDir):
                        os.makedirs(subDir)
                for file_ in os.listdir(os.path.join(data_dir1, label)):
                        file_path = os.path.join(data_dir1, label, file_)
                        shutil.copy(file_path, subDir)
                        print('%s/%s\r' %(count_progress, len(unique_labels)),
                        ↪  end='')
                        count+=1
        print('')

        print('Copying second dataset..')
        unique_labels = []
        for dirname in os.listdir(data_dir2):
                if os.path.isdir(os.path.join(data_dir2, dirname)):
                        unique_labels.append(dirname)
        count_progress=0
        for label in unique_labels:
```

```
                count_progress+=1
                subDir = os.path.join(dir_, label)
                if not os.path.exists(subDir):
                        os.makedirs(subDir)
                for file_ in os.listdir(os.path.join(data_dir2, label)):
                        file_path = os.path.join(data_dir2, label, file_)
                        shutil.copy(file_path, subDir)
                        print('%s/%s\r' %(count_progress, len(unique_labels)),
                        ↪  end='')
                        count+=1
        print('')

        unique_labels = []
        for dirname in os.listdir(dir_):
                if os.path.isdir(os.path.join(dir_, dirname)):
                        unique_labels.append(dirname)
        tmp = []
        for label in unique_labels:
                for file_ in os.listdir(os.path.join(dir_, label)):
                        tmp.append(os.path.abspath(os.path.join(dir_, label,
                        ↪  file_)) + ' ' + label + '\n')

        np.random.shuffle(tmp)
        fh = open(os.path.join(dir_, datasetName+'.txt'), 'w')
        for i in range(len(tmp)):
                fh.write(tmp[i])
        fh.close()

        fh = open(os.path.join(dir_, 'labels.txt'), 'w')
        for label in unique_labels:
                fh.write(label+'\n')
        fh.close()
        print('\nCreated new dataset "%s" with %s images' %(os.path.join(dest,
        ↪  datasetName), count))

def resize_images(filenames, height=64, width=64):
        '''
        Resize a list of images.
        Args:
                filenames: list of strings; each string is a path to an image
↪  file.
                height: integer; the desired height.
                width: integer, the desired width.
        '''
        print('Resizing images..')
        for i in range(len(filenames)):
                percentage = (float(i+1)/len(filenames))*100
                percentage = round(percentage, 2)
                print('%s%%\r' %(percentage), end='')

                img = Image.open(filenames[i])
                os.remove(filenames[i])
```

```
                resized_img = img.resize([height, width])
                resized_img.save(filenames[i])
        print('')
        print('Resized %s images.' %(len(filenames)))

def slice_dataset(data_dir, unique_labels, newSize, dest):
        '''
        Slice a "dataset" into a new .
        Args:
                data_dir: string; the path to the dataset.
                unique_labels: list of strings; each string is a label.
                newSize: integer, the desired number of files to copy from each
↪  label.
                dest: string, optional destination path.
        '''
        if (newSize != -1):
                datasetName = os.path.basename(data_dir) + '_' +
                ↪  str(newSize*len(unique_labels))
                dir_ = os.path.join(dest, datasetName)
                if not os.path.exists(dir_):
                        os.system("mkdir -p %s" %(dir_))

                print('Creating new dataset..')
                count=0
                count_progress=0
                tmp = []
                for label in unique_labels:
                        count_progress+=1
                        tmp =  os.listdir(os.path.join(data_dir, label))
                        np.random.shuffle(tmp)
                        subDir = os.path.join(dir_, label)
                        if not os.path.exists(subDir):
                                os.makedirs(subDir)
                        for i in range(newSize):
                                try:
                                        file_path = os.path.join(data_dir,
                                        ↪  label, tmp[i])
                                        shutil.copy(file_path, subDir)
                                        print('%s/%s\r' %(count_progress,
                                        ↪  len(unique_labels)), end='')
                                        count+=1
                                except IndexError:
                                            print('', end='')
                print('')

                tmp = []
                for label in unique_labels:
                        for file_ in os.listdir(os.path.join(dir_, label)):
                                tmp.append(os.path.abspath(os.path.join(dir_,
                                ↪  label, file_)) + ' ' + label + '\n')
```

```
                np.random.shuffle(tmp)
                fh = open(os.path.join(dir_, datasetName+'.txt'), 'w')
                for i in range(len(tmp)):
                        fh.write(tmp[i])
                fh.close()

                fh = open(os.path.join(dir_, 'labels.txt'), 'w')
                for label in unique_labels:
                        fh.write(label+'\n')
                fh.close()
                print('\nCreated new dataset "%s" with %s images'
                ↪  %(os.path.join(dest, datasetName), count))

def main():
        # Get filenames, labels..
        filenames, labels, unique_labels = find_files(args.data_dir)

        if (args.operation=='convert_to_tfrecords'):
                # The TFRecord filename to output
                output_name = os.path.basename(args.data_dir) + '.tfrecords'
                output_name = os.path.join(args.dest, output_name)
                # Write to TFRecords
                convert_to_tfrecords(filenames, labels, output_name)

        elif (args.operation=='crop_images'):
                crop_images(filenames)

        elif (args.operation=='invert_images'):
                invert_images(filenames)

        elif (args.operation=='merge_datasets'):
                merge_datasets(args.data_dir, args.data_dir2, args.dest)

        elif (args.operation=='resize_images'):
                resize_images(filenames)

        elif (args.operation=='slice_dataset'):
                slice_dataset(args.data_dir, unique_labels, args.size,
                ↪  args.dest)

        else:
                print('Please specify which operation you want to use. Use "-h"
                ↪  flag for help.')

if __name__ == '__main__':
        main()
```

Blekinge Institute of Technology, Campus Gräsvik, 371 79 Karlskrona, Sweden