Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Falco Winkler

Image Representation Learning with Generative Adversarial Networks

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Falco Winkler

Image Representation Learning with Generative Adversarial Networks

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Michael Neitzke
Zweitgutachter: Prof. Dr. Olaf Zukunft

Eingereicht am: 27. Juli 2018

Falco Winkler

Thema der Arbeit
Image Representation Learning with Generative Adversarial Networks

Stichworte
Maschinelles Lernen, Generierende Modelle, Bildverarbeitung, Neuronale Netze, Automatisierung

Kurzzusammenfassung
Mit generierenden Machine Learning - Modellen ist es möglich, neue Daten aus einem Datensatz zu gewinnen. Es gibt viele Anwendungen in der Bildverarbeitung, wie zum Beispiel Auflösungserhöhung und Bildvervollständigung. Aber auch in der Medizin oder in anderen Machine Learning - Algorithmen finden sie Anwendung. Ein generative adversarial network (GAN) wird trainiert, indem ein neuronales Netz, welches Daten aus einem zufälligen Rauschen generiert, gegen einen Gegner ausgespielt wird. Dieser versucht dann, falsche von echten Daten zu unterscheiden. Die Arbeit handelt von der Theorie der GAN's und deren Anwendung auf die Datensätze MNIST und CIFAR10 in sowohl unüberwachter als auch halb-überwachter Weise. Eine dem Stand der Technik entsprechende und vollautomatisierte Implementierung wird verwendet um hochqualitative Bilder zu generieren.
Falco Winkler

Title of the paper
Image Representation Learning with Generative Adversarial Networks

Keywords
machine learning, generative models, image processing, neural networks, automation

Abstract
Generative machine learning models make it possible to derive new data from a dataset. There are many applications in image processing, such as super-resolution and image completion. But they also find application in engineering and in other machine learning algorithms. A generative adversarial network (GAN) is trained by pitting a neural net that generates data out of a noise input, against an adversary, that tries to discriminate between fake and real data. This thesis is about the theory of GAN's and their application on the datasets MNIST and CIFAR10 in an unsupervised and semi-supervised fashion. A state of the art and fully automated implementation is used to achieve high quality image generation.

Contents

Listings

1 Introduction

Machine learning algorithms have seen most of their success in tasks that are easy for a human to perform, but are hardly expressible in a formal language. The location of a face in a picture for example, might be easily recognized by a human. For a machine this could only work with exact arithmetic rules, which makes it a hard problem. Every face looks different and every picture might have a different view angle, shadows, brightness, colors and so on. Early face recognition algorithms [5] used mostly manually extracted and coded features. As of today, it is common knowledge that problems like this can be solved more efficiently by learning those features on a set of training data.

Following the AI winter [8] there was a lot of progress on artificially classifying data. On many datasets, the classifier algorithms achieved human-level performance [23] [7]. The interest in data-generating models has been simultaneously growing. In this thesis, a specific algorithm for generation is examined.

Machine learning algorithms are categorized in a lot of ways. Generative neural networks fall under the category of deep learning algorithms. They learn a representation of a dataset in an unsupervised or semi-supervised fashion with the possibility to sample new data.

1.1 Deep Learning

Deep learning is an approach to artificial intelligence. It refers to algorithms that infer more complex representations out of simpler ones, using a computational graph of many layers [14, p. 1-8].

An image classifier for example, will learn to recognize edges in the first layer. In the second layer it can use the knowledge of how an edge in an image looks like and infer contours. In the third layer it may be able to infer even more complex shapes using knowledge about common contours and so on [35]. Feedforward neural networks are widely used for this purpose, and have seen huge success in inferring knowledge about a dataset without human intervention [28, chap. 6]. They are known to perform well on many classifying tasks. ImageNet, one of the biggest challenge for classifier algorithms, has been mostly approached with deep neural networks [23].

1.2 Representation Learning

To succeed in classifying huge datasets like ImageNet, the neural network has to somehow learn how the visual world looks like. This is done using inference rules encoded in the millions of its parameters. More specifically, the parameters in the network have to be adapted to some sort of generalization or representation of the dataset [14, p. 525]. This is process is called representation learning.

1.3 Unsupervised Learning

Unsupervised learning is a term that refers to machine learning algorithms that do not require additional annotation of the data that they learned from [14, chap. 5]. The GAN framework in its basic form also falls under this category. Supervised learning, in contrast, mostly refers to algorithms that learn a mapping from a data item to a hand-coded label. Neural network classifiers, where the expected class of an image is manually added to the data item, are the classic example for this. Some algorithms use both labeled data and unlabeled data. This is called semi-supervised learning. A semi-supervised GAN framework is examined in 4.8.3.

1.4 Objective

The main goal of this thesis is to train the classical GAN and its extended versions (Semi-Supervised GAN, Deep Convolutional GAN) on the datasets MNIST [25] and CIFAR10 [21] [22] and try to to maximize the visual image quality. Techniques that are used in GAN's should be carefully explored for their effect and understood. From the technical point of view, the aim is to create a reusable, maintainable and testable implementation. Experiments and results management should be automated as much as possible.

2 Generative Modelling

Generative models learn a training data distribution with the possibility to draw new, similar data out of the learned distribution.

They are especially interesting for image processing tasks, such as sharpening, superresolution, image completion and generation of new images, sometimes based on a textual description [13]. In medicine, they have been found useful to find new types of molecules that can be used in drugs [19]. There are also lots of use-cases in content-generation. For example, generative models can save a lot of working hours by generating game content based on some training data [31].

Generative Models can also be used as part of other machine learning algorithms. They can help to improve the training of a neural network classifier when only a part of the data is labeled 4.8.3.

This thesis examines the generative adversarial network. Apart from that, a variety of generative models with some similarities exist.

Autoencoders [4] consist of an encoder function e and a decoder function d modelled by a neural network. They are trained to approximate the identity function, so that $d(e(x)) = x$, but with some constraints on the network. The number of hidden layers between input and output layer (the output of e and the input of d) is set lower than the size of input and output. Consequently, the encoder learns to map data to low dimensional representations while the decoder learns to reconstruct data items out of the encoder output. In the classical autoencoder, training is achieved using a gradient based method with the mean squared error [14, p. 693-696] between real data and decoder output. Variational Autoencoders [10] add a constraint to the training process, which makes it possible to generate new images out of the learned representations. Apart from minimizing the loss between real and decoded images, they also minimize the KL-Divergence [14] between the representation and a unit gaussian distribution. In result, the distribution of the found representations will be close to a gaussian, and drawing new vectors from the gaussian distribution allows to generate new images using the decoder.

PixelRNN and its variant PixelCNN are in the category of autoregressive models, that means that their outputs depend on previous values [32]. Autoregressive Models are important for all

types of sequential generation. The look of a handwritten letter in a word for example, can vary dependent on what was written before and a music note only sounds good if it aligns well with the notes before it. An image can be seen as a pixel sequence from top to bottom, left to right. In this way it makes sense to model each pixel value in terms of the previous ones. To do that, PixelRNN and PixelCNN use various neural network architectures to model a conditioned probability distribution on each pixel value. This probability is calculated as the product of conditional probabilities of individual pixel values.

$$p(x) = \prod_{i=1}^{n^2} p(x_i|x1, ..., x_{i-1})$$

One approach is to use Recurrent Neural Networks [14, chap.10] to model this distribution. Empirically this model was found to capture local features of the training data relatively well. It also captured some global structure trained on large datasets, but most of the samples were not really recognizable.

A downside of the recurrent network approach is that it is slow in generation and training. The reason is that both training and generation are impossible to do in parallel. Each pixel value calculation requires a forward pass in the network with all the previous pixels already calculated. In the PixelCNN architecture [32] [33], a variant of the former, a convolutional neural network is used to model the conditional distribution of each pixel. The probability for each pixel will be calculated with a convolutional kernel on the image, and is thus not dependent on every single previos pixel. This leaves some options for parallel calculation.

3 The theory of generative adversarial networks

3.1 Architecture

Generative Adversarial Networks consist of two sub-networks. A generator $G(z)$, and a discriminator $D(x)$. The generator has the purpose to generate fake data from a random noise z. The discriminator distinguishes between real data from the training dataset and fake data from the generator.

Specifically, G takes random noise of arbitrary dimension as input and outputs a data element while D takes a data element as input and outputs a scalar denoting the probability estimation of the input being real. This estimation can then be used as a gradient signal for the generator. [15]

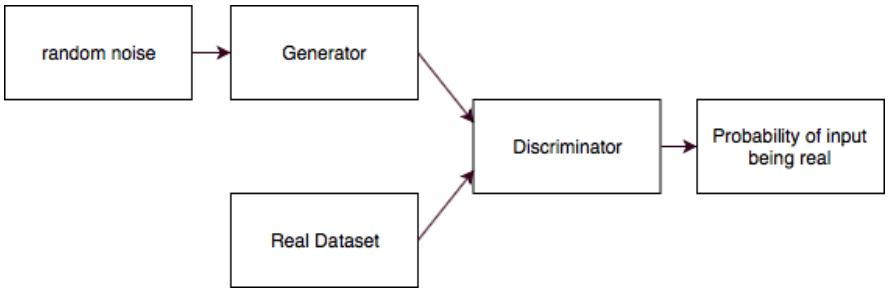

Figure 3.1: A high level overview of Generative Adversarial Networks.

3.2 The adversarial process

An analogy to the generative adversarial learning process is the fight of a money counterfeiter (generator) with a police officer (discriminator). The counterfeiter wants to produce real looking money, while the police officer wants to be able to tell real money and fake money apart. Both are locked in a battle and will develop techniques to better produce or identify

fake money. At some point, the counterfeiter will produce perfect looking samples, which the police officer cannot tell apart from the real money. Neither police officer nor counterfeiter can improve their strategy [15].

In order to simultaneously train the discriminator and generator network, they are alternately trained using a gradient based update rule. During each training step on either D or G, a batch of samples (from the real dataset or from the generator) is used to calculate the loss and the gradient for the corresponding network. The parameters of the network are then updated using the gradient. Training the discriminator can be repeated k - times, but $k = 1$ is the only value used in this paper.

The two training steps repeated in a loop are defined as follows.

Update D by ascending its stochastic gradient

$$\nabla_{\theta_d} \frac{1}{n} \sum_{i=1}^{n} \log D(x_i) + \log(1 - D(G(z_i)))$$

Update G by descending its stochastic gradient

$$\nabla_{\theta_g} \frac{1}{n} \sum_{i=1}^{n} \log(1 - D(G(z_i)))$$

where x_i is a real sample and z_i is a noise distribution from the minibatch [15].

3.3 Loss functions

Mathematically, the GAN framework aims to reduce some distance measure of two probability distributions, the real data distribution and the distribution estimated by G. The cross-entropy function is commonly used to define this distance in neural networks [28], and it is the only distance measure used in this paper. However, most of the extensions to the classical GAN framework are attempts to define a better distance measure (or loss function) [6] [3].

3.3.1 Cross Entropy

In the classical GAN framework, the discriminator and generator loss are defined using the cross-entropy. There are however some different variations and notations that require explanation. As shown in the following, the cross entropy is equivalent to the loss seen in the training algorithm. The cross entropy between to discrete probability distributions P and Q is defined as

$$H(P, Q) = -\sum_{i} P(i) log(Q(i)) \tag{3.1}$$

The sum is over all classes for which P and Q are defined, in case of the discriminator these are 'real' and 'fake'. P denotes a 'true' and Q an empirical distribution.

As stated earlier, D outputs a scalar y', the probability of the input being real. Conclusively, the probability of the input being fake is $1 - y'$. For a datapoint x, D can be modeled as a probability distribution Q where $D(x) = Q(real) = y'$ and $Q(fake) = 1 - y' = 1 - Q(real) = 1 - D(x)$. Likewise, the expected perfect output of D can be modelled as a distribution P where $P(real) = y$ and $P(fake) = 1 - y$. Here y denotes the assigned label, hence it will be either 1 or 0.

Applying 3.1 for a single datapoint x yields

$$H(P, Q) = -(P(real)log(Q(real)) + P(fake)log(Q(fake)) =$$
$$-y \log(y') - (1 - y) \log(1 - y')$$
$$= -y \log(D(x)) - (1 - y) \log(1 - D(x))$$

Averaging over the entire batch of size n, this becomes

$$\frac{1}{n} \sum_{i=1}^{n} -y_i \log(D(x_i))) - (1 - y_i) \log(1 - D(x_i))) = \tag{3.2}$$
$$-\frac{1}{n} \sum_{i=1}^{n} y_i \log(D(x_i))) + (1 - y_i) \log(1 - D(x_i)))$$

Here it can be seen that this is equivalent to the loss in the training algorithm. If the sample is from the real data distribution, the second term will be 0. So the first x_i is always sampled from P_{data}. If the sample is from the generator, the first term is 0, so the second x_i's can be only from the generators distribution. This is $G(z_i)$ where z is sampled from an random noise, usually the random uniform distribution. Applying this to the sum yields the equation from chapter 1, only that it is now a loss function, hence the sign is flipped and it needs to be minimized:

$$-\frac{1}{n} \sum_{i=1}^{n} \log D(x_i) + \log(1 - D(G(z_i))) \tag{3.3}$$

There is another notation for the discriminator loss which is not obvious to be equivalent to the cross-entropy.

$$-\frac{1}{2} \mathbb{E}_{x \sim p_{data}} \Big[\log D(x) \Big] - \frac{1}{2} \mathbb{E}_z \Big[\log(1 - D(G(z))) \Big] \tag{3.4}$$

[13] [15]

This is just the cross entropy cost in the notation of a probabilistic expectation. To understand this notation, one can rewrite 3.2 as follows.

$$-\frac{1}{n}\sum_{i=1}^{n} y_i \log(D(x_i))) - \frac{1}{n}\sum_{i=1}^{n}(1-y_i)\log(1-D(G(z_i)))$$

The term y_i will be 1 for the real data samples and 0 for the generator samples, vice versa for $(1-y_i)$. Therefore, they will in average be $\frac{1}{2}$. Data samples from the real data and generator distribution will be drawn uniformly, which means that each data sample will be drawn with the same probability. Hence the expectations of $\log(D(x))$ and $(1-\log(D(G(z))))$ will be just the average function value, and the sums can be replaced with expectations, resulting in the notation of formula 3.4.

3.3.2 Minimax game

One way to specify the generator loss ℓ_G is to view GAN training as a minimax game [15].

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

The factors of $\frac{1}{2}$ found in 3.4 are omitted here, as they just scale down the value function. Defining the GAN training as minimax game means setting up the networks with opposite objectives. Conclusively ℓ_G can be specified as the negative discriminator loss ℓ_D.

$$\ell_G = -\ell_D$$

Using this loss, the generator will effectively minimize $\log(1-D(G(z_i)))$, as G is not present in the first expectation.

3.3.3 Heuristic, non-saturating game

In practice, it has some advantages to let G maximize the loss $\log(D(G(z)))$ instead of minimizing $\log(1-D(G(z_i)))$. The reason being that $\log(D(G(z)))$ has a high derivative near $x=0$ and therefore gives a strong gradient signal when the discriminator thinks that the given sample is fake. As a result, the early learning process will be more stable and efficient. Mathematically, this means that the opposite objective is used to construct the cross-entropy cost. The generator G will maximize the likelihood of D being wrong instead of minimizing

the likelihood of D being right. This changes nothing about the learning process because the goal of making D fail stays the same [15].

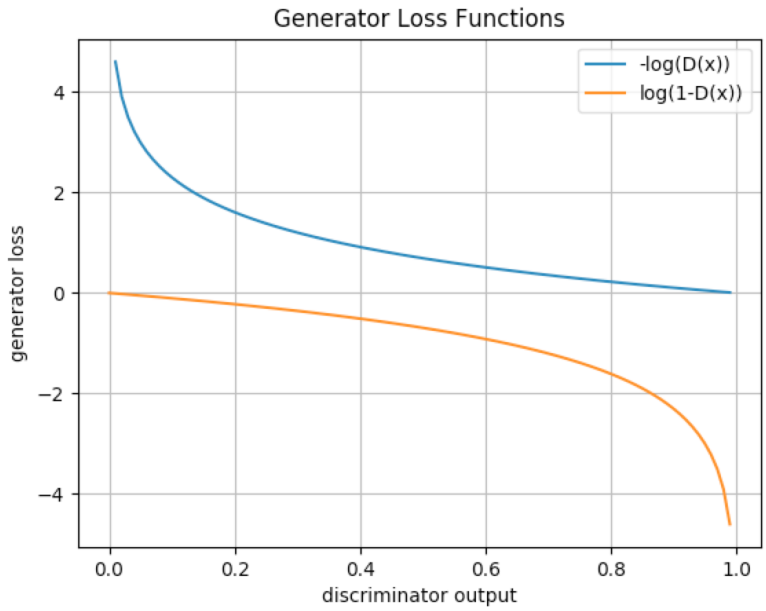

Figure 3.2: Using $-log(D(G(z)))$ for the generator loss has the advantage of strong gradients in early learning

3.4 Advanced techniques

This section describes techniques that are used in the experiments to the extent in which they are relevant. The practical impact of using these techniques will be understandable from the experiment results.

3.4.1 Rectifier activation function

The rectifier activation function addresses the problem of neuron saturation. It is defined as

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

This function is linear in the positive region and constant for negative inputs. In comparison to the logistic sigmoid, it is easier to compute and cannot saturate in the positive region. A neuron activated by this function is also called rectified linear unit, in short form 'ReLU'.

3.4.2 Leaky rectifier activation function

Rectified linear units still face the saturation problem in negative regions. The leaky rectifier activation function solves this. It is defined as

$$f(x) = \max(\epsilon \cdot x, x)$$

The multiplication of x with a small factor (for instance $\epsilon = 0.01$) results in a non-zero gradient for negative inputs, avoiding the neuron saturation in the region below 0.

3.4.3 Adam Optimizer Algorithm

ADAM is an optimization algorithm that can be used instead of the stochastic gradient method. The main difference is that ADAM optimizes a learning rate for every single weight parameter in the network and that these learning rates can adapt over time. An intuitive explanation for why this speeds up learning is to imagine the optimization goal of a weight parameter as a landscape. The goal is then to find the lowest spot in the landscape, though it might be that the landscape is a huge flat desert and the optimum lies in a rather complex part of the landscape with rocks and mountains. In order to get to the minimum, the classical stochastic gradient decent algorithm would take a lot of same-sized steps through the desert and then find the optimum. With ADAM, the step size through the desert can be picked large, so that it will arrive at the more complex gradient landscape quicker, where it can pick a smaller step size to find the optimum [9].

In the experiments, the stochastic gradient optimizer performed poorly and sometimes failed to converge. The ADAM optimizer is a good replacement and improves convergence speed and training stability, especially for very high dimensional problems.

3.4.4 Transposed convolutional layer

Convolution is a well known technique for classifying neural networks. But in a GAN, the generator has to learn a transformation from a noise that goes in the opposite direction of a classical convolutional layer. Transposed convolutional layers do exactly that. For example, a 3×3 convolution over a 4×4 input has an output of dimension 2×2. The equivalent

transposed convolution takes a 2×2 input, adds a padding of size 2 to the borders and computes a 3×3 convolution, resulting in a 4×4 output.

A convolution can be expressed as a matrix operation with the matrix C. Forward and backward passes in the network can then be computing by multiplying with C and C^T. A transposed convolution is then the same operation as the normal convolution, except that the multiplication is done with C^T in the forward pass and C in the backward pass [11]. Transposed convolutional layer are used in deep convolutional GAN's [2].

3.4.5 Batch normalization

Batch normalization layers can be added between neuron layers to aid gradient flow, neuron saturation problems and generalization. To understand batch normalization, it is helpful to look at some methods that speed up the learning process by transforming the input in a non-destructive way. Consider a neural network classifier that has been trained to recognize simple RGB images, where the images have been de-colorized (meaning that the classifier works with black-and-white images in RGB format). If this classifier is exposed to the same images it has successfully been trained on, just with color, it will fail to classify these images correctly. The reason is that the input data distribution (or the statistics of the data) has changed dramatically, commonly referred to as covariance shift [18].

Making the classifier work again could be done by performing a normalization operation on the input (in this case a de-colorization). This works, since after the normalization operation the training data and test data distributions are close again and the network can better deal with the input. Common and long known optimizations to the network input have the same goal of reducing the shift between the training data and test data distribution. For example, transforming images to have zero mean and a variance of 1 has been found to improve both training speed and generalisation of the learned model.

With this in mind, consider for example a hidden layer of a deep and (for simplicity) fully-connected neural network. The activation of a neuron in this layer is calculated as

$$o = \sigma(w \cdot x + b)$$

with σ the activation function, x the activations of the previous layer, w the learned weight matrix and b the bias [28]. During the optimization of the weights of previous layers, the statistics of x change constantly, which is referred to as internal covariance shift. This can slow down learning or even make it impossible, just as different training and test data distributions on a classifier. The idea behind batch normalization is to remedy internal covariance shift by

normalizing x. Specifically, the activations of a hidden layer are normalized to have the mean of zero and the variance of 1 using the formula

$$x' = \frac{x - u}{\sqrt{v + \epsilon}}$$

where x are the activations, u is the mean and v is the variance. The epsilon term is added for numerical stability in computation.

Simply forcing the activations to change would lead to problems, as this change affects all layers behind the batch-normalized one. This would deform features that the network has just learned to represent. That is why for each activation x^k that is normalized, two additional parameters γ^k, β^k are introduced that are also learned by the network. These parameters are then used to scale and shift the normalized activation:

$$y^k = \gamma^k x^k + \beta^k$$

This gives the network the ability to mitigate the effects of the normalization. It may even change the batch normalization to an identity transformation by learning $\gamma^k = \sqrt{v + \epsilon}$ and $\beta = -u$. Informally, this implies that the network can discard any effects of the layer if it is interfering with the learning process.

Batch normalization layers are adopted largely and are especially useful in deep neural network with many layers. They improve the gradient flow during backpropagation and keep neurons from saturating. In the experiments for GAN training with convolutional architecture, they were necessary for a stable training success.

3.4.6 Semi-Supervised GAN training

Classifier networks usually output a class probability distribution for an input data item. The discriminator in the described GAN framework does the same, just with one class. To make use of labels that distinguish between K classes in the input data, this can be extended to $K + 1$ classes, one of which being a newly introduced class named 'generated'.

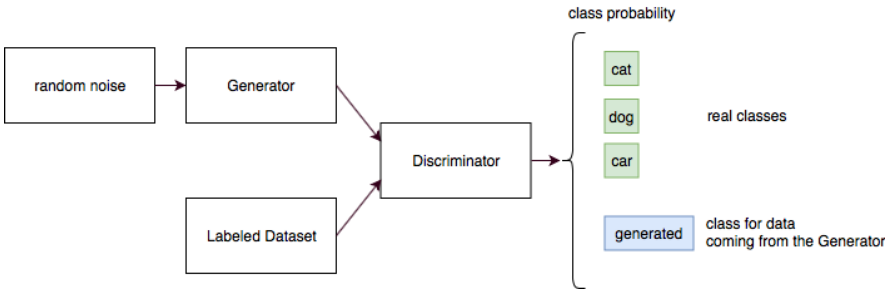

Figure 3.3: The discriminator from the classical GAN framework is extended to classify the real classes and a "generated" class.

The ideal discriminator changes accordingly. When presented a real image, the ideal discriminator should output 1 for the correct class label and 0 for every other class including the 'generated' class. When presented a fake image from the generator, it should should output 1 for the 'generated' class. The generator now has the goal to get the assigned probability for the 'generated' class as low as possible for it's outputs.

In comparison to the original GAN framework, the loss function changes in terms of which distance measures are minimized. The generator minimizes the distance between the discriminators output on label $K + 1$ for generated pictures and the one-hot label for the 'generated' class. Also a supervised loss term is added to the discriminator.

The supervised loss, similar to a classifier network, is the distance between the one-hot labels for the real class and the output without the additional class. This loss is only calculated on real images with disregard of the extended 'generated' class.

If the goal is to train a classifier, this method can improve the generalization of the learned model [16]. It is also useful to learn from very small datasets. Experiments with the MNIST dataset restricted to a very small amount of training images have shown an improvement of about 5% in classification compared to a normal classifier [29]. In the case of a partially labelled dataset, this method makes it possible to train classifiers even if just a small percentage of the training data is annotated. Consequently, the loss has to be extended to unlabeled data. The goal for the unlabeled images is to assign a low probability in the 'generated' label [20]. For generative models, this method appears to improve image quality and training stability. The reason for the quality improvement is not yet fully understood [16]. It could be explained with the fact that a classifier takes features of the whole image in consideration. Only if all the identified features together form an object, the classifier will assign a high probability to that class. In the original GAN framework there is no restrictions on this, which causes the

generator to learn local features well, but sometimes they don't harmonize with the rest of the image to form a whole object [16].

The classification accuracy can act as a measure of training success but only for the discriminator itself. As described previously, a strong discriminator can outperform the generator which would cause the network to produce no reasonable images.

4 Experiments

In this chapter, multiple GAN architectures will be applied to the problem of generating images out of the MNIST and CIFAR-10 Datasets.

4.1 Implementation

The implementation of the GAN framework follows the object oriented paradigm. It also adheres to the commonly known principle of dependency inversion. That means that every responsibility in the training and sampling process is extracted into a class. Objects of those classes are then passed as a dependency to constructors of other classes that need their functionality. This approach aligns well with the single responsibility principle [27] and vastly improves readability due to the separation of different abstraction layers. To build the computation graph, the machine learning framework tensorflow [1] is used.

```
1    def build_graph(self):
2        x, labels = self.input_queue
3        d_logit_real = self.D.inference(self.preprocessing(x))
4        d_logit_fake = self.D.inference(self.G.inference(self.z))
5        self.d_loss, self.g_loss =
6            self.loss_functions(d_logit_real, d_logit_fake)
7        self.d_solver, self.g_solver = self.optimizers
8        self.d_solver.minimize_operation(self.d_loss)
9        self.g_solver.minimize_operation(self.g_loss)
10
11   def train(self):
12       for it in range(self.training_iterations):
13           self.g_solver.minimize()
14           for _ in range(self.k):
15               self.d_solver.minimize()
```

Listing 4.1: At the heart of the implementation, the network is trained in very few lines of code.

4.2 Testing strategy

Another advantage of the object oriented approach is that each component is easier to test as a small object rather than a long script. Tests for the GAN algorithm for example, can be written without the use of any library. Owing to the fact that executing the algorithm is completely decoupled from dependencies to the machine learning framework and other functions such as logging and image saving, dependencies that are unimportant in the current test can be exchanged with mock objects. These mock objects can then make assertions, or count method calls without actually training a full network, or writing images to disk.

The implementation contains mainly numeric computations. While the numeric correctness of computations done by tensorflow [1] is already under unit-test, it has proven useful to test the correctness of chained computations such as the loss function. As it would not be feasible to check for every possible function input, most of the tests are boundary value analysis tests. This is a well-known testing strategy in the field and arises from the observation that functions often behave incorrectly with inputs at the bounds of their input space, or inputs around zero. For instance, it makes sense to write a test for the case that the discriminator loss is very high. In the case of two classes, the discriminator outputs a vector containing (in order) the probability of these classes and the fake probability. If the discriminator's output is $(0, 0, 10)$ for real and $(0, 10, 0)$ for fake data, the error should be very high. In fact, it should be close to 20, as one can easily derive. The semi-supervised loss function for the discriminator is the cross-entropy function applied on the softmax of discriminator logits and labels. Given the softmax function $softmax(z)_j = \frac{e^z_j}{\sum_k e^k}$ and the cross-entropy 3.1 the output for the label $(0, 1, 0)$ is

$$H((0,1,0), softmax((0,0,10))) =$$

$$H((0,1,0), (\frac{e^0}{2+e^{10}}, \frac{e^0}{2+e^{10}}, \frac{e^{10}}{2+e^{10}})$$

$$-log(\frac{e^0}{2+e^{10}})$$

$$-(0 - log(2+e^{10})) \approx log(e^{10}) = 10.$$

This is the same for both real and fake data and as the error is the sum of real and fake data loss, the test can check the result against 20 with a small tolerance value.

The need for unit testing is generally recognized, but especially in GAN's, tests like these are of very high importance, as even small numeric errors can have a devastating effect and can be very hard to find.

4.3 Automatic Deployment

The setup for the GAN framework utilizes a variety of tools to automate the deployment and execution of the written algorithms. All code is under version control and is kept in synchronization with an online repository. Each time new code is published to this repository, scripts are automatically executed to perform the following tasks. In the first stage of the continuous integration, all written unit tests are executed. If the tests fail, all following stages will not be run. If they pass, the application is packaged in a container together with necessary dependencies. The container will be published to a registry, which provides easy access and allows for execution on demand. In the last step, the contained application is deployed to a worker node in a computation cluster. The nodes in the cluster that allow for GPU computing are preferred, as they allow for a much faster execution of the algorithm 4.5. After the training algorithm reaches the maximum amount of training iterations, the process is shut down automatically. Multiple instances of the GAN framework can run in parallel.

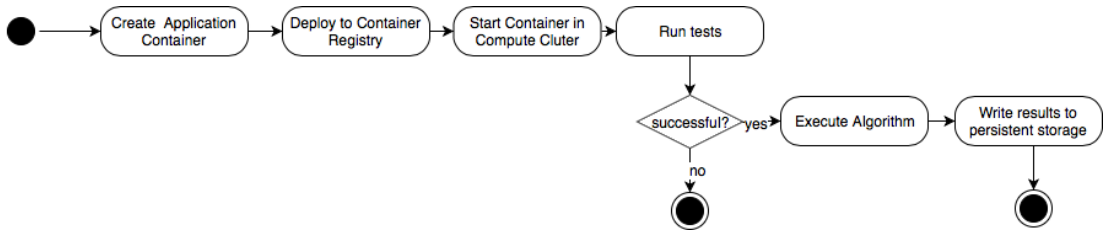

Figure 4.1: Every experiment is fully automated.

4.4 Monitoring and Results management

While the deployment and execution is fully automated, it has proven useful to create a central service to which all GAN framework applications can upload their results. Similar to the the GAN framework, this service is an application that is deployed automatically to the same computation cloud as a container. This service provides a REST Interface [12] to the GAN trainer. Every time the trainer generates a new image or new summary data, it publishes it to the service. The data is then persisted in separate folders for each training process, along with a configuration file that keeps track of the job and its progress. Along with this data, the GAN trainer also publishes the commit hash which can be used to review the exact code changes that triggered the automatic deployment. To persist an exact mapping of code and algorithm output was crucial for keeping an overview after many experiments have been run.

The monitoring service also exposes a website that shows training processes in overview, as well as an image results and training data overview for each job.

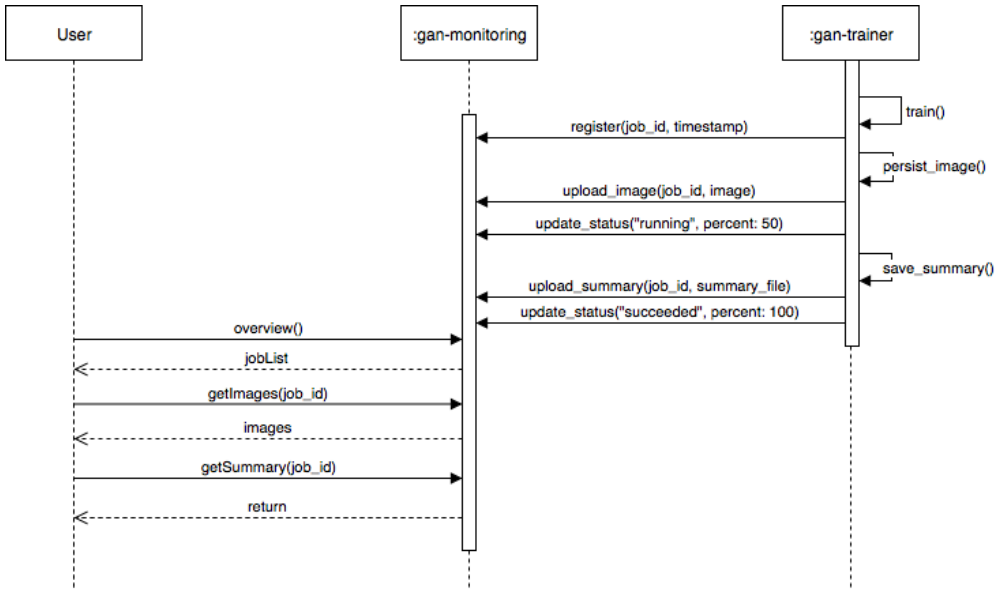

Figure 4.2: The GAN framework uploads results and status messages automatically

4.5 GPU Computing

Graphic Processing Units (GPU's) can run machine learning algorithms quicker than CPU's. That is because they are optimized for graphics computation, in which it is necessary to execute a high number of operations on vectors, matrices and tensors. They are also optimized for parallel computation. Most machine learning algorithms do the same and with GPU driver, one can make use of the optimized computation.

4.6 Pre- and post-processing

Image data for all experiments are the floating point format, meaning the RGB or brightness values are in the interval $[0, 1]$. These values have to be scaled to the range of $tanh$, the output activation of G. Therefore each image value i is pre-processed with $2i - 1$ and post-processed with $(i + 1)/2$. No other pre-processing is applied to the images.

4.7 MNIST dataset

4.7.1 One Layer Architecture

To find out what would be a minimal setup for a successful image generation and to examine some failure modes of the GAN framework, this first experiment uses a similar network architecture to the one that seems to be the minimum to accomplish a reasonable error rate on MNIST. A description of a very simple MNIST classifier network accomplishing a 6% error rate can be found in [28]. This architecture was copied for the discriminator and just inverted for the generator. Precisely, the test setup is as follows:

Generator

96-dimensional input
fully connected layer of size 30, sigmoid activation
fully connected layer of size 28×28, tanh activation

Discriminator

28×28-dimensional Input
fully connected layer of size 30, sigmoid activation
fully connected layer of size 1, linear activation

Stochastic gradient decent with a heuristically chosen learning rate of 0.3 and a batch size of 128 is used. From iteration 40.000 the loss values stagnate and learning freezes. Some generated images are shown in 4.3.

Generated images clearly represent handwritten digits but the generator outputs just items of one class. This class changes during learning, although at each parameter setting the outputs look almost equal for various noise distribution inputs. In the first training run, the generator first learned to output images of twos, and then switched to outputting zeros. In a second run, the generator learned to output first zeros, then switched two nines and back to zeros again. On repeatedly training the model, the same behaviour occurs, but with different numbers, where interestingly the number zero occurs very often. This problem is commonly known as mode collapse. In this failure mode, the GAN learns a parameter setting that maps every noise distribution to the same output. This usually occurs rarely and is triggered seemingly randomly. In this case however, the GAN collapses in every training attempt. This can be explained as follows.

The discriminator learns to output high values for samples close to the dataset. For one of the classes, the discriminator will learn fast that it is real. Usually this will be the case for a

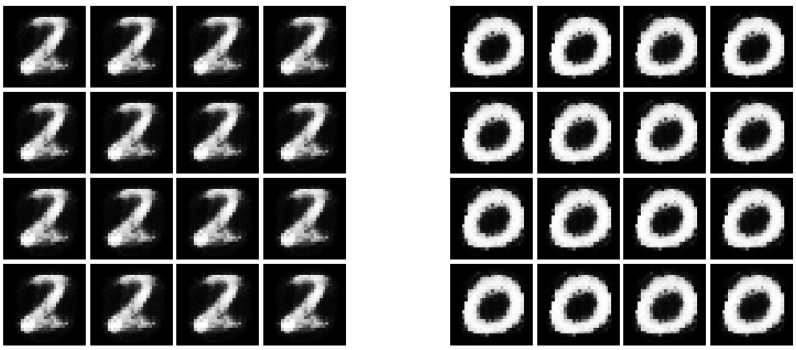

Figure 4.3: Image output at training iteration 6.000 and 18.000

class that is easily distinguishable from the others. This indicates that when presented a real image of that class, the discriminator does especially well and outputs a high probability of the input being real. The generator model lacks of expressive power to represent all 10 classes and learns that it can fool the discriminator by outputting exactly that class. This results in the generator learning to map every noise input to this class. After mode collapse occurred, the discriminator will learn that it has been fooled and it cannot be sure if an image of the collapsed class is real or not. Precisely, $D(x)$ for x resembling the collapsed class, will descend to 0.5. This increases the generator loss. In consequence, the generator will switch to outputting images of another class for which D outputs a high value. D will then learn again that it is fooled and the cycle will start over. Without countermeasures, this behavior will continue forever. The stochastic gradient descent optimizer can just minimize the loss of G but it will be unable to introduce more entropy and to separate the identical outputs [16].

Interestingly the quality of the outputs of the collapsed network are good and visually even better that samples generated with a fully connected architecture that does not collapse and shows all 10 classes. As the generator just focuses on learning to generate one class, it can use all its variables for just one representation. AdaGAN's exploit this behavior by training multiple GAN with a priority on the modes that are missing in each iteration, and adding them together to a mixture model [17].

4.7.2 Neuron Saturation

In about half of the training runs, it can be observed that only one number is learned. During every training run, learning freezes completely at a high number of iterations. Inspecting the outputs of the discriminators hidden layer during the freeze shows the problem cause.

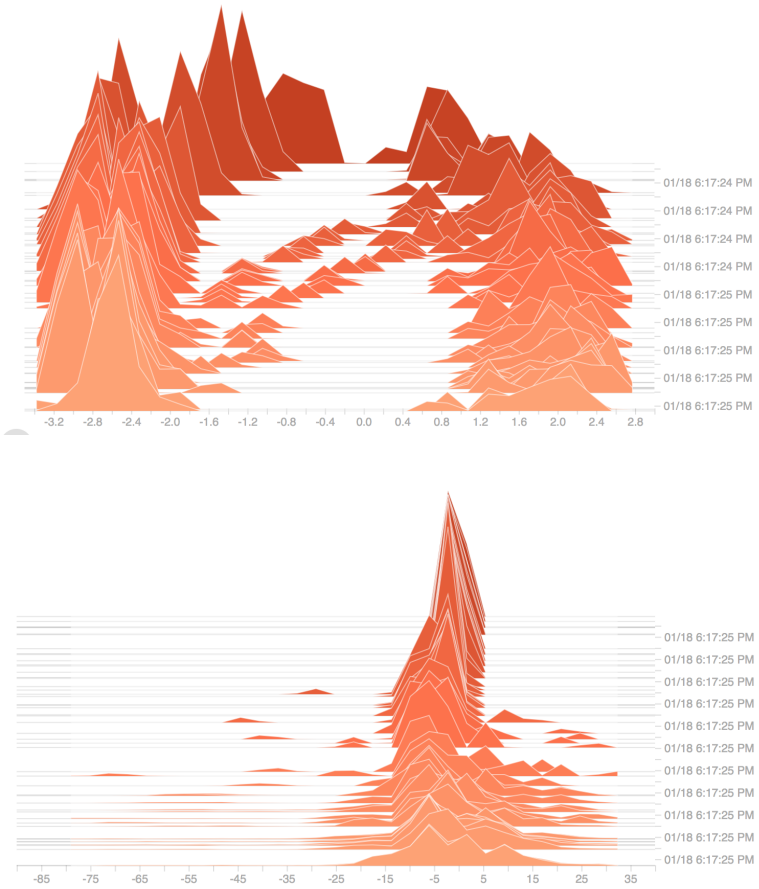

Figure 4.4: The two graphs show the distributions of outputs of the discriminators fully connected layer before applying the sigmoid activation function. The distributions are shown over time (back to front), from the beginning to iteration 1500 (left) and from iteration 13.500 to 15.000 (right). The logits in the discriminator tend very early towards the saturated domain of the sigmoid. In later iterations, logits of both the discriminator and generator tend towards high values spread between -75 and 35.

For extremely positive or negative values, the derivative of the sigmoid converges to zero. As a result, the weights in the discriminator will hardly change, as the gradient during back-

propagation is too small. This also causes the generator to receive a small gradient signal (the loss function includes the discriminator) with the effect that learning freezes completely.

4.7.3 Improved Architectures

The described problem is well known as neuron saturation and can be addressed by using batch normalization 3.4.5 or a different activation function that does not saturate such as ReLU 3.4.1 or leaky ReLU 3.4.2.

In the next experiment, the activation functions are replaced first by ReLU and then by leaky ReLU with a leak parameter of 0.01. This makes a change in the learning rate required, empirically 0.1 is a good fit.

This change is already enough to remedy mode collapse and neuron saturation. The outputs get much more diverse, and even at iterations beyond 40.000 the training does not stop. The error rates of both discriminator and generator are steady, in other words the training process is balanced. This serves as an example how crucial the activation function can be for training success.

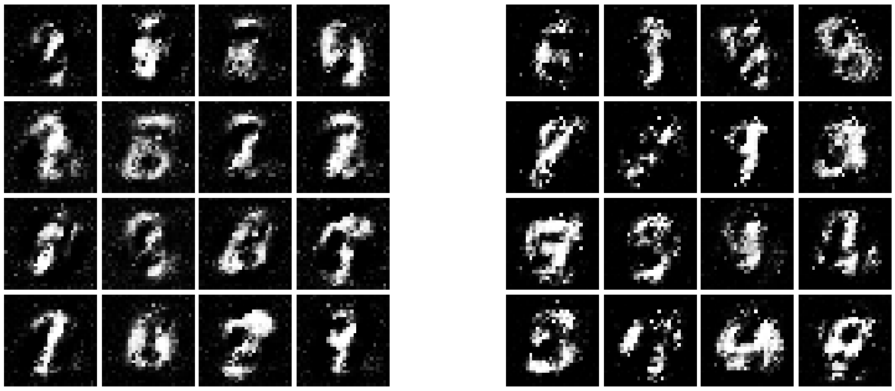

Figure 4.5: Results with the minimal fully connected architecture after replacing sigmoid activations with ReLU. (Iterations 4000 and 15800)

Due to less saturated neurons in the model, both generator and discriminator can make full use of their representative power, which explains the avoidance of mode collapse.

On the other hand the results present a lack of representative power in the networks, especially in the generator.

Restructuring D and G to use more parameters results in another problem: training with a high amount of parameters is not stable anymore and yields unrecognizable results. By the error rates it can be observed that the discriminator is outperforming the generator by a big margin.

The stochastic gradient decent method is the main cause for that. Compared to newer optimization techniques it is quite slow [9] and therefore it is hard for the networks to find equilibrium. Introducing an Adam Optimizer [9] instead with a learning rate of 1^{-3}, $\beta_1 = 0.5$, $\beta_2 = 0.999$ and $\epsilon = 1^{-8}$ a vast improvement of learning speed can be observed. Empirically, two dense layers (with 256 units in the discriminator and 1024 units in the generator) work well for a fully-connected architecture. Giving the discriminator more hidden units throws the training process out of balance. Interestingly, this requires more parameters than the classifier network needed, in order to achieve a reasonable error rate of 6%. Apparently, much less knowledge is required to keep image classes apart than to represent human-readable numbers. With this architecture, the results become human-recognizable after 3000 iterations.

The GAN architecture, which uses a very minimal number of neurons, performs poorly. It suffers from mode collapse, neuron saturation and slow learning, and results in low sample quality. By adding another hidden layer with more nodes, applying the ADAM optimizer and introducing the ReLU activation to the network, these problems have been solved.

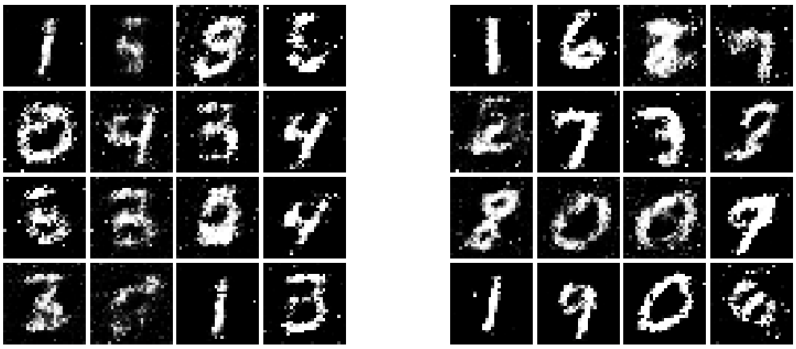

Figure 4.6: Results with the fully connected architecture at iteration 2000 and 3000.

4.7.4 Convolutional GAN's

Training with the previously described architecture is stable and yields human-recognizable samples. However, the samples show two flaws: the numbers appear deformed and not in a

shape as a human would write them, and there are small white spots around the numbers. The structure of the networks, which are all fully connected, makes it hard to learn spatial structure, and causes the problem mentioned above.

The performance of classifying neural networks on image datasets can be drastically improved using convolutional layers. Using this technique on the MNIST dataset, over 99 % classification accuracy has been achieved [26]. On the CIFAR10 Dataset, a convolutional network produces a state of the art performance of 95.56 % accuracy [30].

The benefits of using convolution on datasets with spatial structure can also be exploited in GAN architectures. For architectures of this kind, some guidelines have been proven to work well [2].

- Use ReLU's in the generator, leaky ReLU's in the discriminator.
- Use batch-normalization in every layer except the generator output and discriminator input layer.
- Use convolutional layers in the discriminator and transposed convolutional layer in the generator.
- Do not use pooling layers, to minimize parameters, instead use strided (transposed) convolutional layers.
- Use dense layers only to connect to discriminators output and generators input.

The next experiments are attempts to bring these guidelines into practice and infer a good architecture for MNIST with them.

4.7.5 Developing a convolutional architecture for MNIST

The guidelines help on deciding, which kind of layers and activations should be used. They do not however answer the question of how many layers, of what size are performing well. The best network structure varies for each dataset, and can only be empirically constructed, following some guidelines. A basis for empirical research can be an architecture that performs well on classifying the dataset. For MNIST, a simple convolutional architecture reaches 99.6 % classification accuracy [28].

20 5×5 convolutions, stride 1
2×2 max-pooling, stride 1, ReLU activation
40 5×5 convolutions, stride 1
2×2 max-pooling, stide 1, ReLU activation
fully connected layer of size 1000, 0.5 % dropout, ReLU activation
fully connected layer of size 1000, 0.5 % dropout, ReLU activation
softmax layer, 0.5 % dropout

Strictly applying the previously stated guidelines, the following discriminator architecture can be inferred.

20 5×5 convolutions, stride 1
20 2×2 convolutions, stride 1, leaky ReLU activation, batch normalization
40 5×5 convolutions, stride 1, batch normalization
40 2×2 convolutions, stride 1, leaky ReLU activation, batch normalization
fully connected layer, size 1, 0.5% dropout

Finding a good architecture for the generator can require a bit more research. As a guideline, G should have the same number of layers in roughly the same structure. A good starting point is to just mirror the discriminator architecture. Instead of downsampling an image to a probability, the generator will upsample images from a noise distribution, therefore using transposed convolutional layers in place of convolutional layers. The inferred discriminator architecture transforms the input to 40 7×7 layers. In order to mirror that, the inputs will be connected fully to a layer with $7 \times 7 \times 40$ neurons and upsampled with the same strides and kernel sizes as in the discriminator. Specifically, the mirrored architecture for the generator is:

fully connected layer of size $7 \times 7 \times 40$, ReLU activation, batch normalization, 0.5 % dropout
40 5×5 transposed convolutions, stride 1, ReLU activation, batch normalization
40 2×2 transposed convolutions, stride 2, ReLU activation, batch normalization
20 5×5 transposed convolutions, stride 1, ReLU activation
one 2×2 transposed convolution, stride 2, tanh activation

Using the described setup, the following results are obtained.

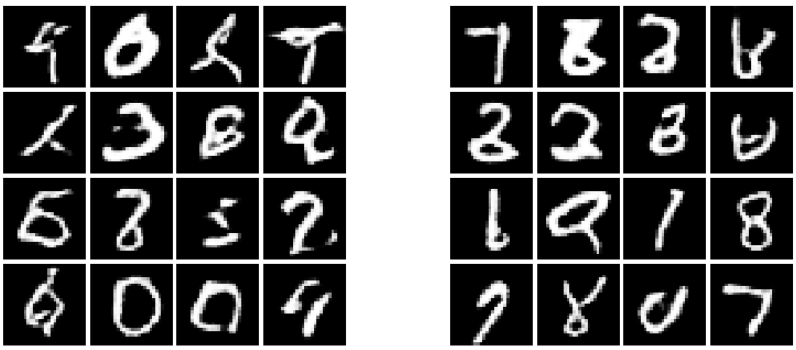

Figure 4.7: Results with the convolutional architecture at iteration 2000 and 3000

It can be observed that the generated samples show sharp lines and little noise. Some of the samples clearly resemble a handwritten number, but most of the samples are hardly recognizable by a human. The cause for this behaviour has to be found heuristically and empirically. For GAN's, this process can be very experimental. The two loss functions used are dependent on the performance of the other network, therefore they are just a rough measure of training success. An unusual behavior observed is that the generator loss alternates quickly between high and low values. This suggests that the generator has problems fitting the dataset. Training is stable and converges, so it is unlikely that hyperparameters need to be adapted.

Contrary to the guidelines, adding a dense layer of size 1024 right after the noise input to the generator seems to aid the training stability. However, the poor visual quality remains.

Another point to improve seems to be the number of convolutional layers. Common convolutional architectures for MNIST seldom use more than three convolutional layers. Therefore, the convolution is simplified into two transposed convolutions with stride two and kernel size 4 x 4, with the filter count of 64 and 1 in the output layer. Consequently, the jitter of both loss functions vanishes and most of the samples are readable, but still worth improving. The architectural change causes the discriminator error rate to increase, which shows that the generator now performs much better after the simplification has been made.

Increasing the filter amount from 20 and 40 in the first and to 32 and 64 in the second convolutional layer, lowers the discriminators error value. Combined with an additional dense layer before the output layer, the best visual results so far are achieved (shown below).

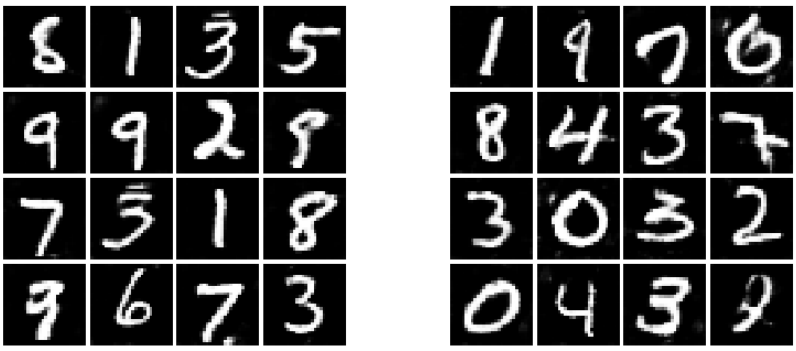

Figure 4.8: Final results on the MNIST Dataset at iteration 2000 and 3000.

Another attempt using the exact MNIST classifier architecture for D (described at the beginning of the chapter) with one dense layer yields the same quality of images. At this point, it gets increasingly harder to compare the approaches for quality, as there is no exact metric defining the performance of the generator. The human eye is a very inaccurate measure of quality, as it is subject to individual judgement. Another approach could be to let classifier networks determine the quality of generated images by checking that there are diverse classes generated that can be recognized with a high confidence score [16].

Interestingly, dropout does not make any recognizable difference, neither in image quality nor in training stability. Dropout is seldom used in popular GAN architectures, as overfitting is rarely a problem [34].

Batch normalization seems to be crucial in the generator architecture. Removing it results in a very high generator loss, a completely unstable training process and unusable images. Surprisingly, this is not the same for the discriminator. Another training run with batch normalization removed in the discriminator produces high quality images in a stable training process like before. This can be explained by the fact that the discriminator uses leaky ReLU as activation function, which does not saturate for negative values as the ReLU does, and this could be the reason that removing batch norm does not change much about the gradient flow in the discriminator. In fact, repeating the experiment with no batch norm in the generator, but leaky ReLU activation instead of ReLU, leads to well recognizable results. The discriminator is still stronger in this setup, but training is stable and convergent.

It is difficult to compare these subtle changes for image quality, but the best results seem to be obtained by the below architecture, which will also be evaluated and improved on CIFAR10 in the following chapter.

fully connected layer of size 1024, ReLU activation, batch normalization
fully connected layer of size $7 \times 7 \times 128$, ReLU activation, batch normalization
64 4×4 transposed convolutions, stride 2, ReLU activation, batch normalization
one 4×4 transposed convolution, stride 2, tanh activation

32 5×5 convolutions, stride 1, leaky ReLU activation
32 2×2 convolutions, stride 2, leaky ReLU activation
64 5×5 convolutions, stride 1, leaky ReLU activation
64 2×2 convolutions, stride 2, leaky ReLU activation
fully connected layer, size $4 \times 4 \times 64$
fully connected layer, size 1

In short, it was possible to empirically find a GAN architecture that works well for the MNIST dataset. Following guidelines can help to accomplish a stable training process and good results. But often one has to deviate from these to achieve better visual quality of the samples.

4.8 CIFAR10 dataset

The CIFAR10 dataset consists of 60000 32 x 32 color images (10000 images for validation, and 50000 images for training) from 10 different classes. Compared to the MNIST dataset, the structure of the data samples is far more complex. In comparison, one should expect a larger network architecture to work well. GAN training on more complex datasets is usually harder to balance.

4.8.1 Grayscale CIFAR images

The first training setup evaluates the performance of the convolutional architecture for MNIST, described previously. In order to fit the data to the architecture, every RGB value will be transferred to a single grayscale value using the luminosity formula.

$$y = 0.2989r + 0.5870g + 0.1140b$$

Some changes are required to fit the image size. The generator is changed to upsample from a $8 \times 8 \times 40$ instead of $7 \times 7 \times 40$ dense layer, resulting in an output size of 32×32 instead of 28×28. The discriminator is solely changed to reshape into the correct output dimension.

Figure 4.9: Grayscale cifar results at iteration 20000 and 22000 with the convolutional architecture previously used on MNIST

From some results, it seems like the network performs reasonably well, as some shapes and forms become visible. On the other hand, what becomes clear in most of the images is that the network has problems representing the more complex features of a CIFAR image.

4.8.2 Colored CIFAR images

In the next experiment, the same architecture will be applied to colored images. This requires just the change of the color dimension from one to three. As the colored version of the images contains considerably more information, the network is found to perform slightly worse on them.

Figure 4.10: Results at iteration 20000 and 22000 with the same architecture applied to the colorized version of the dataset. A slight tendency towards mode collapse can be seen.

The generator is able to express some features, but they seem a little bit less distinct than before. In some training runs, the image tends towards a certain color and structure, which seems like a slightly less critical version of the collapsing mode failure that was explored previously on the MNIST dataset.

An obvious limitation of this architecture is the number of filters. As described in [35], every filter in a convolutional network learns to detect a certain meaningful feature. In the current setup, the network is not capable of learning every feature that a CIFAR image has. It can detect straight and round edges, but simply has no more filters available to learn more complex features like for example the face of a dog.

In the next experiment, one more (transposed) convolutional layer is added to the generator and discriminator. Specifically, a layer with kernel size 5, stride 1 and 128 features is added right after the last dense layer of the generator and right before the first dense layer of the discriminator. In this case, the padding is chosen in a way that it keeps the input dimensions, so the other layers do not have to be modified. Using this approach, the visuals become more clear, the outputs become more diverse, and one can observe actual objects in some of the images.

The architecture can be further simplified. In the current state, the network contains two convolutional layers with kernel size and stride 2, and no batch normalization. Another training run without those layers (with 2 convolutional layers of kernel size 5 and stride 2) shows

comparable results. With this setup, it has been necessary to add batch normalization to the discriminator.

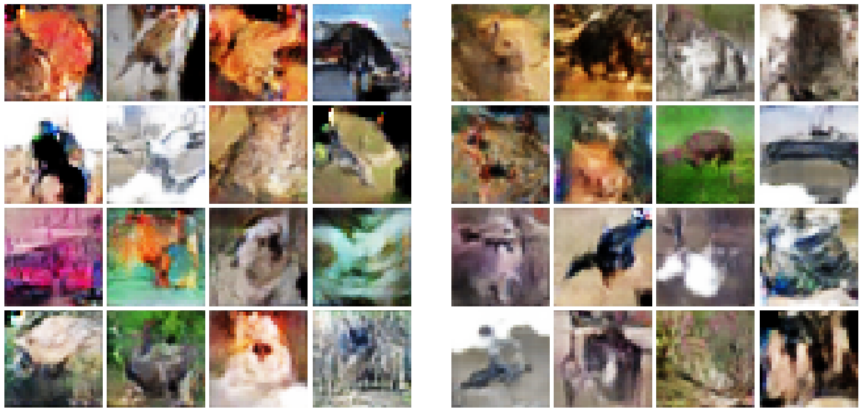

Figure 4.11: Final results at iteration 20000 and 21500.

For completeness, the generator and discriminator architecture for obtaining the final results on CIFAR10 are specified as follows.

fully connected layer of size 1024, ReLU activation, batch normalization
fully connected layer of size $8 \times 8 \times 128$, ReLU activation, batch normalization
128 5×5 transposed convolutions, stride 1, ReLU activation, batch normalization
64 4×4 transposed convolutions, stride 2, ReLU activation, batch normalization
one 4×4 transposed convolution, stride 2, tanh activation

32 5×5 convolutions, stride 2, leaky ReLU activation
64 5×5 convolutions, stride 2, leaky ReLU activation, batch normalization
128 5×5 convolutions, stride 1, leaky ReLU activation, batch normalization
fully connected layer, size $4 \times 4 \times 64$, batch normalization
fully connected layer, size 1

The quality of the images can hardly be improved with the classical GAN framework, as it is comparable to state of the art results [16].

4.8.3 Semi-Supervised GAN

In this experiment, the architecture and hyperparameters are the same as in the previously described setup. The only changes are that the discriminator outputs a probability distribution

over 11 classes instead of 1, and that the error function changes as described in 3.4.6. To get a rough measure of the disciminator quality, the accuracy is computed by taking the average over correct predictions in one minibatch. The accuracy goes up to 100% in some iterations, but that is not to be confused with the test error of a classifier. The test error determines the accuracy over a whole dataset and not just a minibatch. Therefore, it is a metric for determining training success, but not a good measure of the actual quality of the discriminator-classifier.

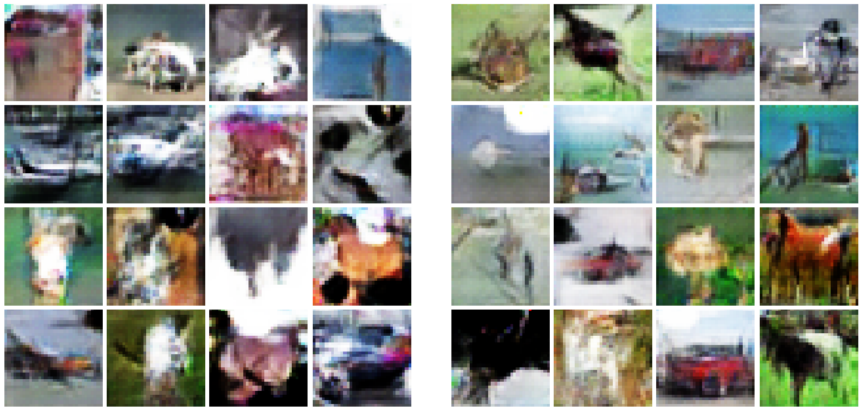

Figure 4.12: CIFAR 10 results at iteration 10000 and 14000. Some images clearly resemble some objects. Results in earlier iterations also look promising, but the algorithm seems to need more time to train well.

The results do not seem to differ much from the previous ones, though for some images it can be said that they are a bit more recognizable. This aligns with the statement that human annotators seem to recognize images generated with this approach a bit better [16].

From the error rate and the average of correct classifications it is apparent that the discriminator training is successful, though that does not necessarily mean that the GAN training is stable. The discriminator is still just an aid for the generator to find a parameter setting, which captures the training data distribution best. In other words, the fact that the discriminator does well on classifying real and fake data does not necessarily mean that the generator has found a good representation of the dataset.

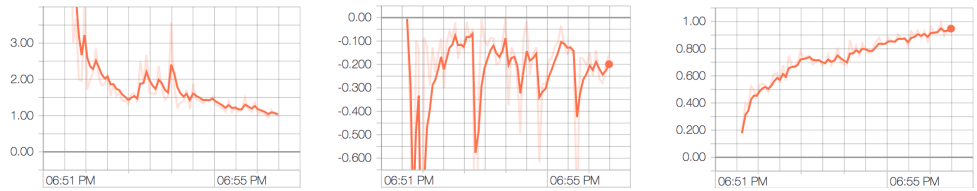

Figure 4.13: Graphs for (left-to-right) the discriminator error, the generator error, and the training classification accuracy.

The output of generative models are also of interest if one wants to examine problems with classifier architecture. Previous experiments with fully connected networks have shown that the impossibility to learn neighbourhood information about pixels is problematic. Convolutional networks can learn the spatial structure of features. Though, as a result of the feature map having high activation, regardless of how many times the feature is present, the network has problems learning the count of features. For example, it is unable to learn that a dog has just two eyes.

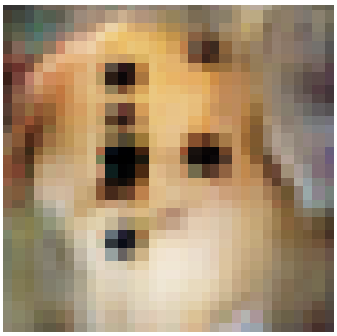

Figure 4.14: This picture seems to resemble a dog, but with a incorrect number of eyes.

Similar problems were also reported on the larger IMAGENET dataset [16].

5 Summary

5.1 Results

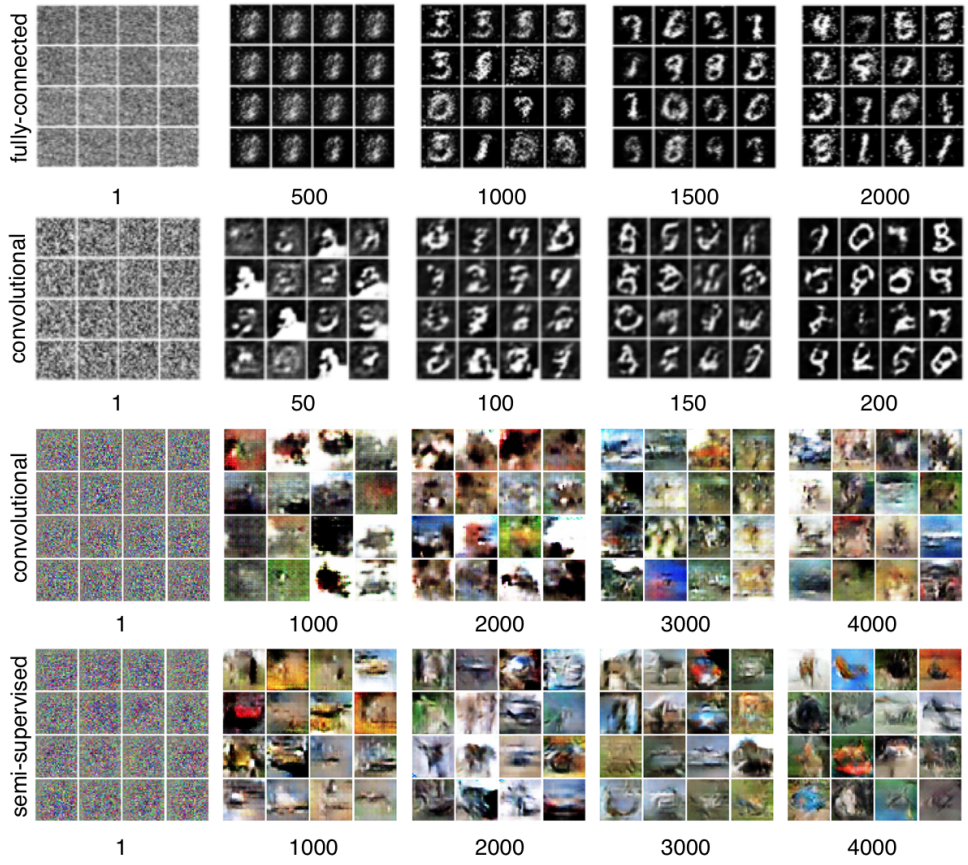

Figure 5.1: Visualization of the described training processes. The label under each picture denotes the iteration.

Every implemented architecture was sufficient to generate images that, more or less, resemble the dataset. As there were not only large differences in image quality but also in convergence speed, is is necessary to make a comparison on the training processes.

Convolutional GAN's show what has been visualized in classifier networks before [35]: during early training, only the most basic features like edges and contours are learned in layers close to the input. Deriving the more complex features to generate complete images takes more time and requires previous layers to already have learned some basic features. The semi-supervised GAN generates better images faster, because the discriminator is forced to learn features that are important to keep the classes apart. This forces the generator to create more distinct features [16].

Differing the most were the fully-connected and the convolutional architecture on MNIST. The fully-connected architecture learned a recognizable representation after 2000 iterations. In comparison, the convolutional architecture managed to generate clearly better results in a tenth of the iterations. While this shows the suitability of convolution for image processing with neural networks, the results have also pointed out the problem that a count of a feature is hard to learn.

To determine if the approach was successful, it is of interest how well some generated examples are represented. Or, in other words, if it is possible to find similar images in the dataset. Even though the goal of the generator was not to generate good copies of real images, this experiment shows what features have been learned and what similar images in the dataset look like.

An arithmetic measure for image similarity can be defined as

$$l_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

where I is the image pixel value matrix and the iteration is over all pixel indices p. In short, l_1 defines the distance between image samples as the difference between pixel values. The images with a minimal l_1 distance to a sample image are thus similar.

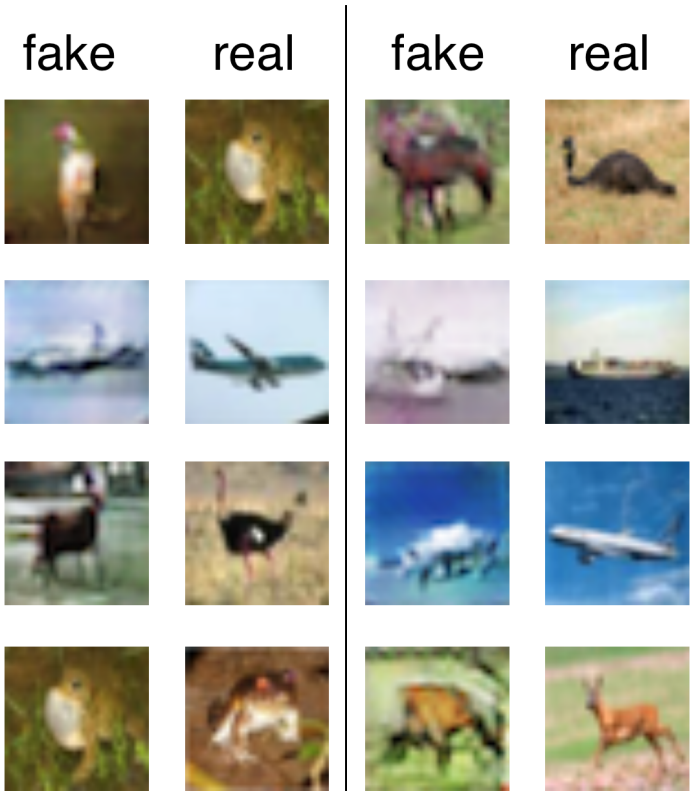

Figure 5.2: Some cherry-picked generated examples of the semi-supervised training runs are compared with their nearest l_1 distance neighbor in the training dataset of CIFAR10 (50000 images).

The results clearly show a resemblance between learned distribution and training data. Although the fake pictures are cherry-picked from multiple training runs, it can be said that random pictures also have nearest neighbors that are quite similar, but they are not so well human recognizable.

5.2 Conclusion

For an inexperienced reader most of the results may seem pretty bad. On most of the images there are just seemingly random blobs of color. There is a big difference when comparing them with a random selection of CIFAR10 images. Generated images are not nearly as sharp as the original images.

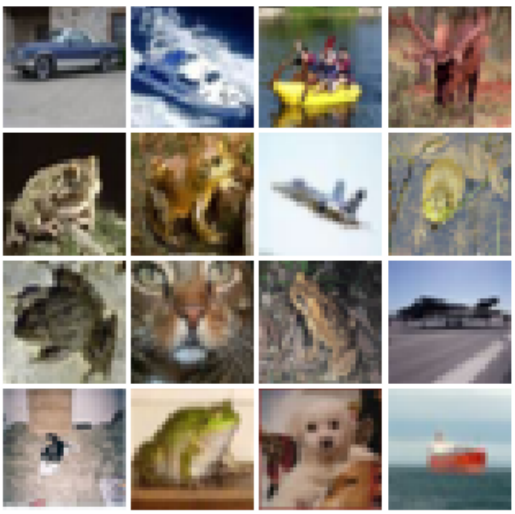

Figure 5.3: 16 randomly picked CIFAR10 images.

To see why learning a representation of CIFAR10 is difficult, one has to understand that the described models had to learn to represent a dataset containing 50000 images with extremely complex relationships in the data. Images from one class can be numerically vastly different in CIFAR10 [21] [22]. In addition to that, some original images from the dataset are of bad quality and hardly recognizable.

Seen under these aspects, it is amazing what generative models are capable of. They learn an accurate representation of datasets and can show what the visual world looks like to a neural network. Although these algorithms are based on well defined mathematical rules, generative models are a sort of artificial creativity; they can combine visual features in new ways that are unseen in the dataset.

The results in this paper are comparable with state of the art implementations of the classical and semi-supervised GAN framework [16]. GAN's are easy to sample from and require less computing power than other approaches [33] [10]. The downside is that the probabilities of pixel values are not explicitly learned. Conclusively, GAN's are impossible to validate by an exact metric. Training GAN's also has to be carefully balanced. That is why, depending on the dataset, GAN's can be a very poor choice of a generative model. In content generation and image processing they have been proven useful, and their use-cases extend even into other domains, such as medicine [19]. In 2016, Yann LeCun described GAN's as 'the most interesting idea in the last 10 years in ML' [24] which supports my own conclusion that GAN's are highly intriguing to study.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[2] Soumith Chintala Alec Radford, Luke Metz. Unsupervised representation learning with deep convolutional generative adversarial networks.

[3] Martín Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein GAN. *CoRR*, abs/1701.07875, 2017.

[4] Pierre Baldi. Autoencoders, unsupervised learning, and deep architectures, 2012.

[5] Roberto Brunelli and Tomaso Poggio. Face recognition: features versus templates. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(10):1042–1052, 1993.

[6] Xi Chen, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, and Pieter Abbeel. Infogan: Interpretable representation learning by information maximizing generative adversarial nets. *CoRR*, abs/1606.03657, 2016.

[7] Yangqing Jia Pierre Sermanet Scott Reed Dragomir Anguelov Dumitru Erhan Vincent Vanhoucke Andrew Rabinovich Christian Szegedy, Wei Liu. Going deeper with convolutions, 2015.

[8] Daniel Crevier. *Ai: The Tumultuous History Of The Search For Artificial Intelligence*. Basic Books, 1993.

[9] Jimmy Ba Diederik P. Kingma. Adam: A method for stochastic optimization.

[10] Carl Doersch. Tutorial on variational autoencoders, 2016.

[11] V. Dumoulin and F. Visin. A guide to convolution arithmetic for deep learning. *ArXiv e-prints*, March 2016.

[12] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. AAI9980887.

[13] Ian Goodfellow. Nips 2016 tutorial: Generative adversarial networks.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[15] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014.

[16] Wojciech Zaremba Vicki Cheung Alec Radford Xi Chen Ian J. Goodfellow, Tim Salimans. Improved techniques for training gans.

[17] Olivier Bousquet Carl-Johann Simon-Gabriel Bernhard Schölkopf Ilya Tolstikhin, Sylvain Gelly. Adagan: Boosting generative models.

[18] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[19] Artur Kadurin, Sergey Nikolenko, Kuzma Khrabrov, Alex Aliper, and Alex Zhavoronkov. drugan: An advanced generative adversarial autoencoder model for de novo generation of new molecules with desired molecular properties in silico. *Molecular Pharmaceutics*, 14(9):3098–3104, 2017. PMID: 28703000.

[20] Diederik P. Kingma, Danilo Jimenez Rezende, Shakir Mohamed, and Max Welling. Semi-supervised learning with deep generative models. *CoRR*, abs/1406.5298, 2014.

[21] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[22] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[24] Yann LeCun. What are some recent and potentially upcoming breakthroughs in deep learning?, 2016.

[25] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[26] Zhibin Liao and Gustavo Carneiro. Competitive multi-scale convolution. *CoRR*, abs/1511.05635, 2015.

[27] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008.

[28] Michael Nielsen. Neural networks and deep learning, 2010.

[29] Augustus Odena. Semi-supervised learning with generative adversarial networks, 2016.

[30] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin A. Riedmiller. Striving for simplicity: The all convolutional net. *CoRR*, abs/1412.6806, 2014.

[31] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgard, Amy K. Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. Procedural content generation via machine learning (pcgml), 2017.

[32] Aaron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks, 2016.

[33] Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. Conditional image generation with pixelcnn decoders, 2016.

[34] Yuhuai Wu, Yuri Burda, Ruslan Salakhutdinov, and Roger B. Grosse. On the quantitative analysis of decoder-based generative models. *CoRR*, abs/1611.04273, 2016.

[35] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks, 2013.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 27. Juli 2018 Falco Winkler