Eberhard Karls University of Tübingen
Faculty of Science
Wilhelm-Schickard-Institute for Computer Science

Thesis

Bachelor of Science

Bring the GANs into Action
Extending DeepOBS with novel test problems

Vanessa Tsingunidis

Tübingen, 16.07.2020

Examiner

Prof. Dr. Philipp Hennig
Wilhelm-Schickard-Institute for Computer Science
University of Tübingen

Thesis Advisor

Frank Schneider
Wilhelm-Schickard-Institute for Computer Science
University of Tübingen

Tsingunidis, Vanessa:
Bring the GANs into Action
Extending DeepOBS with novel test problems
Bachelor Thesis Computer Science
Eberhard Karls University of Tübingen

Abstract

The research field of deep learning actively develops new optimizers in order to train deep neural networks. However, it is becoming increasingly difficult to classify what can be considered state-of-the-art in deep learning optimization. Reproducible and comparable results have to be obtained in a time-consuming process, testing a new optimizer on various tasks and deep neural networks, completed by tuning the hyperparameter, to achieve a better performance.

Therefore the optimizer benchmark library DeepOBS is offering a highly automatized and standardized process for comparison with unified performance measures but still flexible hyperparameter tuning. The library offers to choose between different data sets and network architectures for different tasks, which constitute simplified representatives of more complex systems, to help understand the performance of optimizers.

By now, the deep neural networks in DeepOBS mainly solve classification tasks. Yet, the diversity of tasks is an important issue to stay meaningful as a benchmark and offer a product, which can be used for optimizer comparison in research. This work extends the DeepOBS library with a special family of deep generative models, the Generative Adversarial Networks (GANs).

GANs are a rather new approach, though gained high popularity in the deep learning field by means of remarkable achievements in generation tasks. Not only the capabilities of this architecture make it interesting in an optimizer benchmark, but the fact that training these until convergence comes with various new challenges. In this thesis, a deep convolutional GAN is introduced, addressing image generation tasks on different data sets, within the `PyTorch` framework. The architecture and the adapted training process form the ground for the integration of further varieties and the comparison of an optimizer's performance across different generation tasks in DeepOBS. The implementation of the training process comes with a qualitative evaluation method for the beginning. Allowing the user, to save the images generated by the model, at every epoch of the training process, to visually investigate the performance of an optimizer.

In order to investigate the capabilities of the test problems and the characteristics of the training process, several experiments have been done. This work, illustrates the most realistic outcomes, along with examples that emphasize the hyperparameter sensitivity of GAN models and the pitfalls that can arise during the training process.

Zusammenfassung

Auf dem Forschungsgebiet des Deep Learning wird aktiv an neuen Optimierern zum Training tiefer neuronaler Netze gearbeitet. Jedoch wird es immer schwieriger einzuordnen, was als Stand der Technik in der Deep Learning Optimierung angesehen werden kann. Reproduzierbare und vergleichbare Ergebnisse müssen in einem zeitaufwändigen Prozess erarbeitet werden, indem ein neuer Optimierer an verschiedenen Aufgaben und tiefen neuronalen Netzen getestet wird, ergänzt durch die Abstimmung der Hyperparameter.

DeepOBS bietet dafür eine Benchmark-Bibliothek für Deep Learning Optimierer, mit einem hochautomatisierten und standardisierten Prozess für den Vergleich, zusammen mit einheitlichen Leistungsmerkmalen und dennoch flexiblem Hyperparameter-Tuning. Die Bibliothek bietet die Möglichkeit, zwischen verschiedenen Datensätzen und Netzwerkarchitekturen, für verschiedene Aufgaben zu wählen. Diese stellen vereinfachte Repräsentanten komplexerer Systeme dar, um die Leistung eines Optimierers besser zu verstehen.

Bisher lösen die tiefen neuronalen Netze in DeepOBS hauptsächlich Klassifikationsaufgaben. Dennoch ist die Vielfalt der Aufgaben ein wichtiges Thema, um als Benchmark aussagekräftig zu bleiben und ein Produkt anzubieten, das in der Forschung für Vergleiche von Optimierern genutzt werden kann. Mit dieser Arbeit wird die DeepOBS Bibliothek um eine spezielle Familie von tiefen generativen Modellen, den Generative Adversarial Networks (GANs) erweitert. Diese sind zwar ein eher neuer Ansatz sind, haben aber im Deep Learning durch bemerkenswerte Erfolge bei Generierungsaufgaben hohe Popularität erlangt. Nicht nur die Fähigkeiten dieser Architektur machen GANs interessant für ein Benchmark von Optimierern, sondern auch die Tatsache, dass das Training dieser mit verschiedenen neuen Herausforderungen einhergeht.

In dieser Arbeit, wird ein Deep Convolutional GAN eingeführt, welches mit verschiedenen Datensätzen zur Generierung von Bildern trainiert wird. Die Architektur und der angepasste Trainingsprozess bilden die Grundlage für die Integration weiterer Varianten in DeepOBS und den Vergleich der Leistung eines Optimierers über verschiedene Generierungsaufgaben hinweg. Die Implementierung des Trainingsprozesses kommt für den Anfang mit einer qualitativen Evaluationsmethode. Diese erlaubt es dem Benutzer, die vom Modell erzeugten Bilder, in jeder Epoche des Trainingsprozesses, zu speichern und die Leistung eines Optimierers visuell zu untersuchen.

Um die Fähigkeiten der Testprobleme und die Eigenschaften des Trainingsprozesses zu veranschaulichen, wurden mehrere Experimente durchgeführt. In dieser Arbeit werden die realistischsten Ergebnisse präsentiert, zusammen mit Beispielen, die die Hyperparameter-Sensitivität von GANs und die Fallstricke, die während des Trainingsprozesses auftreten können, hervorheben.

Acknowledgments

I very much appreciate the opportunity to write my bachelor thesis in the friendly and welcoming atmosphere of Philipp Hennig's research group and therefore having the chance to study state-of-the-art techniques of machine learning.

Further, I would especially like to thank my thesis advisor, Frank Schneider, who made this possible with his exemplary guidance throughout this work.

In addition, I thank Felix Dangel for setting up a meeting with researchers, from the Max Planck Institute, in order to support this thesis with special expertise on GANs.

In the same way, I thank Robin Schmidt, who helped me with his experience and technical understanding of the DeepOBS library and the Tübingen ML Cloud.

Last but not least, I want to thank my family and friends for emotional support during my studies.

Contents

List of Figures

List of Tables

List of Abbreviations

ADAM	Adaptive Moment Estimation
BCE	Binary Cross-Entropy
DCGAN	Deep Convolutional Generative Adversarial Network
DeepOBS	Deep Learning Optimizer Benchmark Suite
FID	Fréchet Inception Score
GAN	Generative Adversarial Network
MLP	Multilayer Perceptron
NLP	Natural Language Processing
SGD	Stochastic Gradient Descent
VAE	Variational Autoencoder

Chapter 1

Introduction

The field of deep generative neural networks evolved from the continuous progress in research on deep learning and deep neural networks. Some of the models within this family gained wide popularity through remarkable achievements. Especially two architectures have to be named in this context, Variational Autoencoders (VAEs) and Generative Adversarial Networks (GANs). These introduce new applications for deep neural networks, to not just detect and classify real-world objects (further explained in Chapter 2), but to reconstruct image representations such that the results can even deceive human perception. Figure 1.1 shows some representative examples to give an impression of the capabilities of today's generative models.

Figure 1.1: Representative samples from generative models. The two images on the left are the result of a three-level hierarchical Vector Quantized VAE model trained on FFHQ-1024x1024 data set. Adopted from [RvdOV19]. The last four images are samples from a GAN model trained on the 1024x1024 CelebA HQ data set. Adopted from [KALL18].

In general, this is accomplished by reducing the huge amount of input data into a significantly smaller dimensioned space, to force a model to discover and internalize the essential features and properties of the input data, in order to generate it by itself. This process is not limited to the image generation tasks shown in the example but also includes natural language generation, which is briefly explained in Chapter 2 or even 3D object generation [FSG16], among various other applications.

Just like with classical deep neural network tasks the problem of learning is cast

as an optimization problem and in most scenarios, optimization means using gradient descent methods. Established gradient descend methods used for optimization include SGD [RM51], its momentum variants (MOMENTUM) [Pol64, Nes83] and ADAM [Die14] (see for example [Kar17]). Although research on new techniques for optimization and improvements of the existing ones, has not ceased (see following papers for further information [Mat12, Tim16, Yi 16, Zey17, Ily18]), by now no optimizer seems to outperform the other ones significantly. However, there is this notion in the deep learning community, that some optimizers perform not better in general, but only on certain tasks (see for example [SSH20]).

The reproduction of an optimizer's performance and behavior from empirical reports is constrained by the lack of unified implementation environment for a proper and significant benchmark process. Researchers decide for themselves how detailed, in which environment, and which hyperparameter setting the performance of a new optimizer or variants of existing ones are reported. This often means that the exact results cannot be reproduced what makes the comparison with other developments even harder. Additionally, the models, to train on, get heavily simplified, as real-world applications ordinarily have to be omitted, due to limitations of time for training and execution. Considering all the above it is difficult to keep an overview of state-of-the-art optimizer and make grounded statements on their relations to one another.

Just recently Schneider et al. [Fra19] developed the DeepOBS library (Section 2.1), which directly addresses the evaluation and comparison of deep learning optimizers. The authors provide unified performance measures and baselines for a more straightforward comparison with the popular optimizer, SGD, MOMENTUM, and ADAM mentioned above, among with a great variety of test problems (Section 2.2), which is basically a neural network (Section 2.4) within a fixed setting. What makes it even more attractive for researchers in the field is that the two most common and widely used frameworks `TensorFlow` [Ma15] and `PyTorch` [PGC+17] are provided (see for example the statistics in [He19]). Overall DeepOBS automates and simplifies the process of downloading and preparing data sets, as well as logging relevant metrics when running an optimizer and afterward reporting and visualizing the results. With the standardized procedures Schneider et al. target to help streamlining the analysis, making results reproducible and comparisons fair as possible.

Providing a sufficient environment to create expressive benchmark results requires the continuous development of such an environment, guided by current research and improvements of the field, in order to maintain the integrity of the objectives. Two main aspects prevent DeepOBS to follow up with state-of-the-art conditions for a significant benchmark in research:

- **Contribute to both frameworks equally:** Taking into account the performance margins between the two machine learning frameworks, that

Table 1.1: Overview of the test problems included in the DeepOBS library, separated by data set and model, and whether there exists an implementation in the `TensorFlow` or the `PyTorch` framework. The colors denote the task to which a test problem belongs to. For DeepOBS these tasks are image classification ● the most common task, image generation ●, and one test problem solving a natural language processing task. Additionally, DeepOBS offers small two-dimensional problems, for sanity checks, where the loss function is explicitly given ●.

Data set	Model	Description	Framework
2D	● Noisy Beale	*Noisy version of Beale function*	TF
	● Noisy Branin	*Noisy version of the Branin function [Bra72]*	TF
	● Noisy Rosenbrock	*Noisy version of the Rosenbrock function [Ros60]*	TF
Quadratic	● Deep	*100dim ill-conditioned noisy quadratic [CCS+ 19]*	TF
MNIST [LBBH98]	● Log. Reg.	*Logistic regression*	TF PT
	● MLP	*Four layer fully-connected network*	TF PT
	● 2c2d	*Two conv. and two fully-connected layers*	TF PT
	● VAE	*Variational Autoencoder*	TF PT
Fashion MNIST [XRV17]	● Log. Reg.	*Logistic regression*	TF
	● MLP	*Four layer fully-connected network*	TF PT
	● 2c2d	Two conv. and two fully-connected layers	TF PT
	● VAE	*Variational Autoencoder*	TF PT
Cifar-10 [Kri09]	● 3c3d	*Three conv. and three fully-connected layers*	TF PT
	● VGG 16	*Adapted version of VGG16 [SZ15]*	TF
	● VGG 19	*Adapted version of VGG19*	TF
Cifar-100 [Kri09]	● 3c3d	*Three conv. and three fully-connected layers*	TF PT
	● VGG 16	*Adapted version of VGG16*	TF
	● VGG 19	*Adapted version of VGG19*	TF
	● All-CNN-C	*The all convolutional net from [SDBR14]*	TF PT
	● Wide ResNet-40-4	*Wide Residual Network [ZK16]*	TF
SVHN [NtWC+11]	● 3c3d	*Three conv. and three fully-connected layers*	TF
	● Wide ResNet-16-4	*Wide Residual Network*	TF PT
IMAGENET [DDS+09]	● VGG 16	*Adapted version of VGG16*	TF
	● VGG 19	*Adapted version of VGG19*	TF
	● Inception-v3	*Inception-v3 net as described in [SVI+ 16]*	TF
Tolstoi	● CharRNN	*Recurrent NN for character-level language modeling*	TF

Bahde showed in his work [Aar19] and the fact that most developers prefer to use only one of the two frameworks, allowing to either choose one of them or to take the opportunity to run experiments on both, is an important issue. Offering only one framework considerably reduces the target group and leaves the question open whether variations in behavior are due to the setting or the framework. Therefore striving a diverse selection of architectures and modules in both frameworks is fundamental. Table 1.1 shows which test problem is represented in each framework. By now, there are twice as many test problems in `TensorFlow` as in `PyTorch`. This can be attributed to the fact that DeepOBS was first published with the `TensorFlow` framework only [Fra19].

- **Provide expressive test problems:** As stated in the corresponding paper [Fra19] the provided test problems, shall offer a diversified range of tasks and difficulties that are as representative as possible for real-world applications and adapt to state-of-the-art configurations, while at the same time, enables less complex and time-consuming training. This work addresses both the range of diversity and the level of difficulty. Table 1.1 can be seen as a status report on the existing test problems at the time before this thesis and is later on supplemented by the contributions of this work in Table 5.1. What stands out is, that most models come under the image classification tasks whereas image generation and language processing problems are very limited.

To address these problems, this work strikes to extend the implementations supported by the `PyTorch` framework and as the main objective, extend the variety of tasks provided by introducing a meaningful new class of test problems to the DeepOBS library:

- **Support diversity for both frameworks:** Table 1.1 points out, that it is the variety of models in the `PyTorch` framework that is to be extended, to achieve equally distributed test problems across the frameworks. For `PyTorch`, there should no longer be only a slimmed-down version of DeepOBS available. Rather, it is intended to enlarge the possibilities for comparison between the frameworks on the same test problems, by implementing the missing ones into `PyTorch` in Section 3.1 as well as to provide new, more complex tasks.

- **Image generation tasks of higher complexity:** Considering the above described ongoing popularity of generative deep neural networks and the models in Table 1.1, it is out of the question that DeepOBS needs more image generation tasks, to stay competitive as a benchmark. As one of the two popular families in image generation, the implementation of a VAE, is already provided in DeepOBS, this work is dedicated to the second family, generative adversarial networks (GANs). Introducing GANs (Section 3.2) comes with a lot of new challenges in the training process (Section 3.2.3), making this project even more interesting.

Chapter 2

Background

When proposing a new optimizer, benchmarking the performance on deep neural networks, which are widely accepted by the research community, is crucial. Therefore the benchmark library DeepOBS (2.1) provides a variety of test problems on which an optimizer (2.3) can be tested and compared with state-of-the-art optimizer. These test problems (2.2) consist of deep neural networks (2.4) at their heart, combined with a data set (2.5) and a loss function (2.6).

2.1 DeepOBS

In the introduction, the fundamental objectives of the DeepOBS library are outlined but not the structure that is build up to achieve them. DeepOBS consists of a pipeline of modules, shown in Figure 2.1, build upon each other, to cover the whole benchmark process. At the ground level it offers to download and pre-process data sets from the source and directly batches them. Combined with the *Model*, the deep neural network, a test problem can be run. In the training process (*Runners*), the performance of the optimizer is calculated, by determining loss and accuracy as well as logging those for evaluation. After training, *Baselines* offer the possibility to compare one's results with competitors (currently these are SGD, MOMENTUM, ADAM) and a visualization module saves the performance measures as LaTeX-files. Introducing a new test problem usually only requires the modification of the *Model Loading* and potentially the *Data*

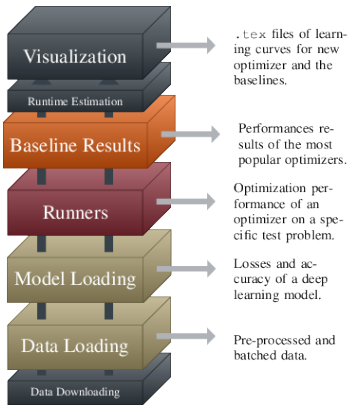


Figure 2.1: Module pipeline of DeepOBS, from bottom to top. Adopted from [Fra19]

Loading part of DeepOBS. However, for a novel class of test problems, more or less complex changes in the *Runners* part are required. This is the case for GANs, such that the introduction of these (3.2) demands new neural network architectures (3.2.1), a different training process (3.2.3) than all other existing test problems, and additional data sets (3.2.4).

2.2 Test Problems

The components that build up a test problem, on which an optimizer (2.3) is applied, consisting of the deep neural networks (2.4), each of them in many variations, the data set (2.5) for training, that directly denotes the domain of the test problem and the loss function (2.6) to calculate the error. The architecture of the neural network and the training process (3.2.3) varies for the task to solve. Combined with the specific training process, the components form a class of test problems. Figure 2.2 is a simplified illustration and serves exclusively to visualize the interrelationships of the essential modules. Note that this figure applies for classical test problems with one objective function, rather than GANs, as these consist of two DNNs (2.4.4) and therefore have more than one objective function.

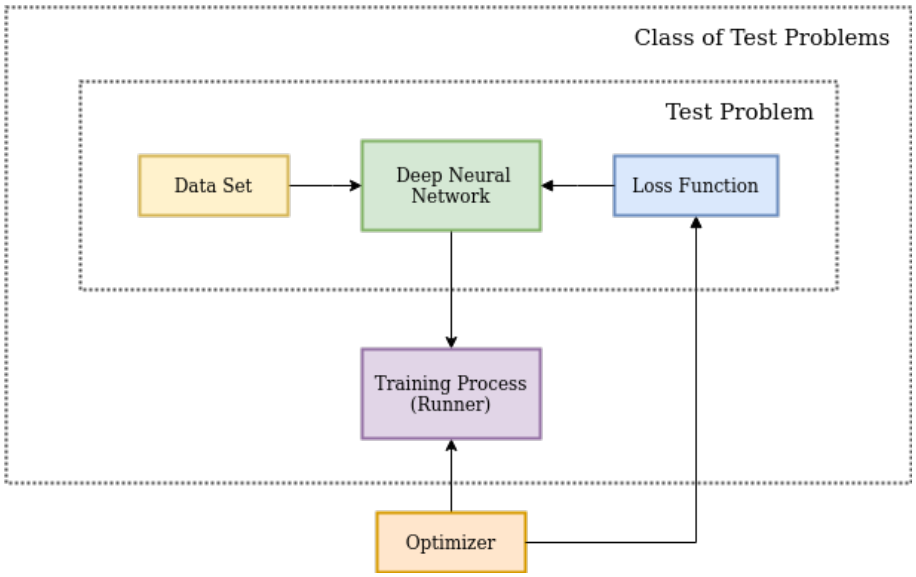


Figure 2.2: Illustration of the components of a test problem, with the deep neural network at its heart, getting input from the data set, and evaluating its performance with the loss function. Combined with the specific training process these components form a class of test problems. The optimizer applies directly to the loss function to minimize it in the training process.

2.2.1 Classes of Test Problems

Tasks or classes of test problems form the ground for the optimization problem. The better the optimizer (2.3), the faster and more accurate the test problem tends to solve the task. All of the classes below can be achieved with various types of test problems.

- **Regression** is a task that belongs to the area of supervised learning. Regression is used to determine relationships between a dependent variable and one or more independent variables. The input is a series of discrete or real values. The objective of regression models is to approximate the mapping function from inputs to continuous output variables. A common example would be predicting stock prices.

- **Classification tasks** are assigned to the area of supervised learning. This means the input data used for training comes with labels in the form of numeric keys. Here the labels can only have a finite number of values. The output of a classification task is a classifier, that predicts the classes, defined by the labels, of new, meaning unseen examples of the same data type used in training. This includes image recognition in Chapter 1 mentioned, for example, handwritten numbers from 0-9 or object recognition as described by Szegedy et al. [SRE+15].

- **Generation tasks** are a class of test problems, that covers the objective to generate new examples according to those seen in the training data. This means, there is no single correct output for a given input but various outputs striking to match the distribution of the input data or less formal of what appears to be perceived as realistic or natural. This leads to new challenges in training for deep neural networks, which will be discussed in Section 3.2.

Besides the classes mentioned above, there exist many more in the field, like anomaly detection, clustering, or denoising, which are not further explained here, as they are not represented in DeepOBS and therefore do not appear in the context of this thesis.

The above-mentioned tasks can be applied in different domains, referring to the type of input data that is processed. In DeepOBS two domains are represented among different tasks:

- **Image Vision** is the most represented domain in DeepOBS. It refers to all problems in which a DNN is applied to understand the content of a digital representation, such as images or videos. This can be for example a classification task in which the model has to predict numbers shown on pictures.

- **Natural Language Processing** (NLP) is broadly defined as the automatic comprehension and generation of natural language in speech or text by a system. NLP is applied, for example, to solve sequence-to-sequence or character prediction problems. This can be achieved with supervised as well as unsupervised learning. In DeepOBS the test problem in the NLP domain solves a character-level language modeling task (1.1).

2.3 Optimizer

The optimizer forms the basis for training neural networks. Therefore this section offers a more detailed view on the respective optimizer, to get a better notion for the differences in their performance. Only those optimizer are mentioned here whose results are used as baselines for comparison in DeepOBS and are therefore taken up again in Chapter 4. The selection of baselines covers three of the most popular and established optimizer in the research field.

2.3.1 SGD

The algorithm mentioned in the introduction was the first of the three optimizers to be introduced. With a batch x_i from the training data and the assigned labels y_i the respective loss $l(\theta_t|(x_i, y_i))$ is calculated at each iteration t through the network. The θ_t parameter denotes the current point and η the learning rate or step size for the calculation of the next point θ_{t+1}. Thereby the gradient tries to minimize the loss in the way of determining the direction for the next point.

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta l(\theta_t|(x_i, y_i)) \tag{2.1}$$

2.3.2 SGD with MOMENTUM

The momentum variant of SGD was introduced, to steam oscillations and accelerate the gradient vectors in a direction that minimizes the error [Sut86]. This was achieved by adding the momentum term Δv_t with hyperparameter β, that takes into account the weight change already made at step t when calculating the change for the next step $t + 1$.

$$\begin{aligned} \Delta v_{t+1} &= \beta \Delta v_t + \eta \nabla_\theta l(\theta_t|(x_i, y_i)), \beta \epsilon [0, 1] \\ \theta_{t+1} &= \theta_t - \Delta v_t \end{aligned} \tag{2.2}$$

2.3.3 ADAM

The comparatively young Adaptive Moment Estimation (Adam) is founded on the adaptive learning rate method RMSProp[1] and has shown great performance in research [Rud17]. It extends the update rule, by taking into account a decaying average of prior squared gradients v_t as well as an exponentially decaying average of past gradients m_t similar to the above-mentioned MOMENTUM. The gradient of the objective function to the parameter θ_i at time step t, is denoted as g_t.

$$\begin{aligned} m_{t+1} &= \beta_1 m_t + (1-\beta_1) g_t \\ v_{t+1} &= \beta_2 v_t + (1-\beta_2) g_1^t \end{aligned} \tag{2.3}$$

Due to the fact, that m_t and v_t are initialized as zero vectors, these tend to be biased towards zero, which is particularly noticeable at the initial time steps when β_1 and β_2 are close to 1. The authors counteract this weakness by computing a bias correction for the first and second moment estimates $\hat{m}_t$ and $\hat{v}_t$. Then these are applied to the update rule.

$$\begin{aligned} \hat{m}_{t+1} &= \frac{m_t}{1-\beta_t^1} \\ \hat{v}_{t+1} &= \frac{v_t}{1-\beta_2^t} \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \end{aligned} \tag{2.4}$$

2.4 Deep Neural Networks

Deep neural networks are the core elements of a test problem. Therefore offering a diverse selection of architectures that cover state-of-the-art approaches and allow test problems to act as representatives for real-world applications is essential. Deep neural networks are distinguished according to their basic structure, despite the fact, that the exact architecture may also vary within this distinction. Basically, the objective of the network is to turn the input into the desired output by pushing it through a network of layers, which in turn consists of several weights.

2.4.1 Multilayer perceptron

Multilayer perceptrons (MLP) are considered the most basic model of a deep neural network. They consist of an input and an output layer along with multiple hidden layers. The inputs given are pushed forward through the network taking the dot product with the weights between the input and the

[1] http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

hidden layer. At each of the layers, an activation function is applied to calculate the output at the respective weights before it is pushed forward to the next layer. Usually, either the sigmoid function, tanh, or rectified linear units are chosen for the activation function. This procedure is repeated until the last layer. In the training process, backpropagation is applied that corresponds to the selected activation function, pushing the error back into the network to adjust the weights. In the case of testing, the MLP makes a prediction based on the output pushed through the activation function.

For example, in classification tasks every layer learns to map a different characteristic of the input data, similarly to our brain, which uses the V1 area for the detection of edges and corners [Ros58].

2.4.2 Convolutional Neural Networks

With convolutional neural networks, the research field of deep learning proceeds to use the organization of the visual cortex in the human brain as an inspiration [SDBR14]. Here the architecture tries to imitate the connectivity pattern of neurons in the way that only certain nodes in the network react to stimuli in a limited area of the image, thus successfully capturing the spatial dependencies of the image.

Convolutional neural networks achieve this by convoluting and reducing the image, without losing features that are critical for an accurate prediction. Basically, they are performing a many-to-one relationship operation. This is done by applying a filter with a much smaller size than the original input, shifting it through the picture and performing a matrix multiplication operation with the pixels over which the filter is hovering, till the entire image is traversed, outlined in Figure 2.3. The filter is referred to as kernel with the main objective to extract features, starting with low-level features, such as edges and color, then proceeding with additional convolutions to extract more high-level features [DV16].

The downscaling is usually done by either max pooling, which replaces the values in the filter area with the maximum value or average pooling, which works analogous but with the average value and mainly decreases the computational power needed to process the data. Among these various other layers can be applied, such as batch normalization or skip connections. After the convolutional layers, a fully-connected one is applied, to get a vector that allows classification in the form of probability calculations.

Transposed Convolutions. The process of convoluting images can be applied in the backward direction as well, to produce an image out of, for example randomly sampled values, as it is done with deep convolutional GANs [AR16]. This comes under the term of transposed convolution, deconvolution or fractionally stridden convolutions, performing a one-to-many relationship operation,

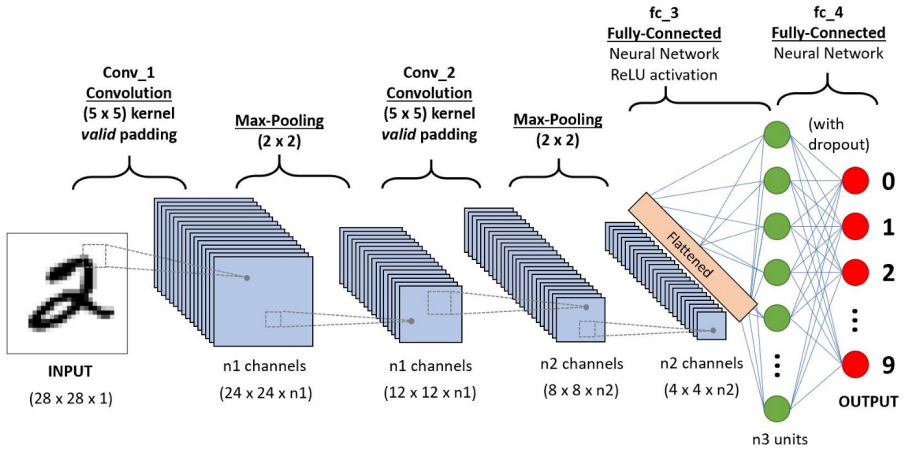


Figure 2.3: The basic architecture of a convolutional neural network training on the `mnist`[LBBH98] data set. The input image gets convoluted and downscaled to a size of 4x4 pixel maintaining the positional connectivity before it is proceeded into a fully-connected layer to get the one-dimensional vector to classify the digits from 0-9. Adopted from [Sah15]

with the same course as above but the other way around. For example, when the values of a 2x2 matrix are broadly speaking, upsampled to a 4x4 matrix, it results in a 1-to-9 relation, which is maintained due to the weight layout, when further deconvolutions are applied with the same filter or kernel size, further illustrated in Figure 3.1. What is notable here is, that a transposed convolution can always be emulated with a direct convolution but this would mean adding various columns and rows of zeros to the input, which results in a far less efficient implementation.

2.4.3 Variational Autoencoder

As stated in Chapter 1, VAEs are one of the most popular representatives from the area of generative models. The architecture of a VAE, depicted in Figure 2.4, basically consists of an encoder processing the input data X in space R^n to an encoded form with lower dimensionality Z in the latent space R^m with $m < n$ [Doe16], which is usually done with the above-mentioned convolutional layers. Most noteworthy here is, that within the encoder two vectors are constructed, to create objects that follow a Gaussian distribution. From there on, a decoder takes Z and reconstructs it to $\hat{X}$, by trying to match the distribution $p_{\hat{x}}$ with the original distribution p_x of the input data. This is usually done with the above-mentioned transposed convolutions aiming that the following equation applies $X = \hat{X}$. VAEs are special in the sense that their loss function consists of a reconstruction term, accelerating the decoding theme

and a regularization term, that makes the distributions from the encoder close to a normal distribution. The regularization is needed to obtain a latent space with good properties, in the meaning of forcing the encoded distributions p_z to rather overlap for reasons of continuity and completeness than fall apart.

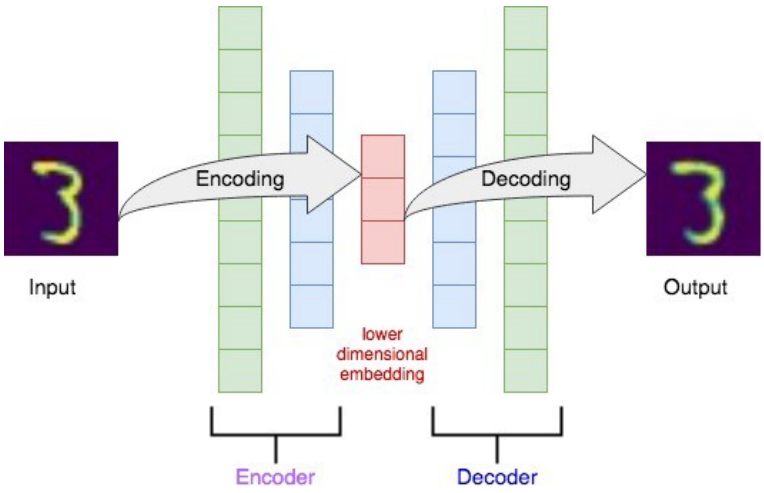


Figure 2.4: Straightforward illustration of the architecture of a VAE training on the `mnist`[LBBH98] data set to reconstruct the distributions of handwritten numbers. Here the input denotes X, the output $\hat{X}$, and the red-colored bar display the Z vector that is fed into the decoder. Adopted from [Han19]

2.4.4 Generative Adversarial Networks

GANs were first introduced by Ian Goodfellow et al. [GPAM+14] addressing image generation tasks and their popularity has exploded since then. What is special in regards to GANs is that they consist of two distinct models, each is a deep neural network depending on the other one in the training process. The first model acts as the generator, which in generating, for example, images, from a given noise. Second is the discriminator, whose job is to look at a given input, without any knowledge whether it was generated by the generator or it is genuine training data. The basic idea is that during the training process the generator tries to outwit the discriminator by generating more and more realistic looking data. On the other hand, the discriminator tries to detect and correctly classify real from fake, as it is illustrated in Figure 2.5. The generator and the discriminator compete with each other in something that is referred to as a minimax game.

The generator G is a differentiable function generating outputs $G(z)$ with a distribution p_g, from an input noise vector Z. G strikes to map the distribution of its outputs to the distribution of the input data X of the discriminator. The discriminator D outputs a single scalar, the probability $logD(x)$, by classifying

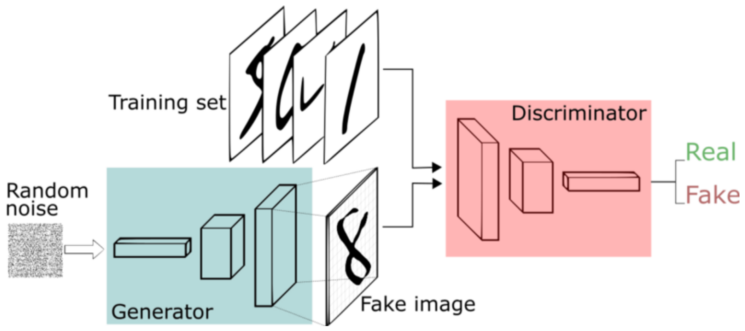


Figure 2.5: The basic architecture of a GAN. In this case, the discriminator is trained with `mnist`[LBBH98] to force the generator to output images looking like handwritten numbers. Adopted from [Sil18]

the given input X as part of the training data or as a creation of the generator $G(z)$. G is then trained to fool the discriminator, which means to minimize the probability of D to designate its outputs as fake ($log(1-D(G(x)))$). Conversely, the discriminator D is trained to maximize its probability to differentiate between X and $G(z)$ ($logD(x)$). Both terms together define the following function that is to be optimized,

$$min\ max\ V(G,D) = E_{z\sim p_z(z)}[log(1-D(G(z)))] + E_{x\sim_{data}(x)}[logD(x)]. \quad (2.5)$$

In theory, the end of the training process is reached when both networks' loss functions are at a level that can be defined as equally minimized. This scenario is referred to as the Nash equilibrium, illustrated in Figure 2.6. This is achieved, when the generator creates images looking like the distribution of the given training data, so the discriminator's guess is at a robust 50 percent confidence.

2.5 Data Sets

The term data sets covers every type of input data, that is used to train and test a deep learning model and in this case a deep neural network. This can range from a function to images or text and is specific to the given task of the model.

In the supervised learning field the input data X or at least a sufficient subset of it comes with labels Y. Labels can be specified as a kind of numeric property describing a feature or class, for which the model learns to map, the respective input to.

No matter, which type of input data is used and what the task of the neural network is, before starting the training, the input data needs to be divided into subsets, for several reasons:

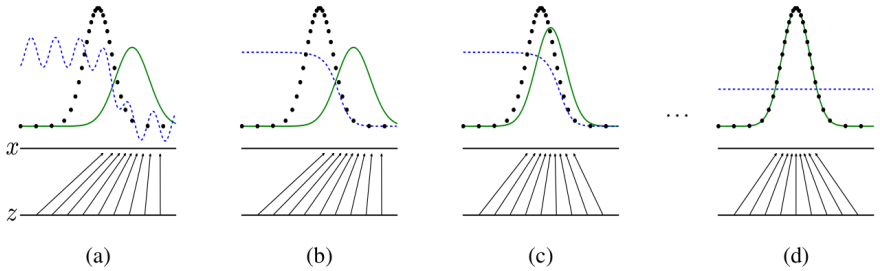


Figure 2.6: Development of the distributions over the training process of a GAN. With the discriminator's distribution D (*blue, dashed line*), to distinguish the generative distribution p_g (*green, solid line*) from the sample distribution p_x (*black, dotted line*). The horizontal lines below the graphs show the domains from which x and z are sampled. In this case, z is uniformly sampled and the upward arrows represent how the mapping $x = G(z)$ imposes the non-uniform distribution p_g on the generated samples. Starting from a point (a), at which the discriminator is capable of partly accurate classifications. Before an update step for G is made (b), D converges to $\frac{p_x(x)}{p_x(x)+p_g(x)}$. During training $p_g(G)$ approximates p_x step by step with every update of the generator G (c), due to guidance of D. In the optimal case (d) the adversarial pair reaches a state at which no further improvement can be done, because now both p_x and p_g are equivalent $p_g = p_x$. The discriminator cannot distinguish among both distributions anymore. Adopted from [GPAM+14]

- **Training Set:** This subset of the input data is used to adjust the weights on the neural network during training, to improve its performance.

- **Validation Set:** With this subset, one does not adjust the weights of the network but evaluate it in every epoch to verify that an increase of accuracy over the training data set yields an increase in accuracy over new, unseen data. This is important to avoid overfitting the neural network. For example, in the event, that the accuracy over the training set increases but the accuracy over the validation data stays the same or decreases, then the network is overfitting, and the training parameter should be reconsidered. DeepOBS separates the validation from the test set to provide the possibility to tune the hyperparameter for training. With the validation set, one can for example test several learning rates before deciding for the one who leads to the best performance.

- **Test Set:** For the final solution of the network, meaning no further adjustments will be made, another subset of the input data is coming into action. This subset only contains data the network has never seen before, not even for any validation cycle, and therefore, is used to confirm the actual predictive power of the network.

2.6 Loss Functions

The optimization process in training deep learning models with gradient descent methods requires the loss function to repeatedly calculate the respective error l_n in order to be minimized. This section covers only the functions that are used in the context of this thesis and are therefore listed in more detail.

2.6.1 Cross-Entropy Loss

The cross-entropy loss is a very common loss function to use when a classification problem has various classes Y assigned to the input data X. The function creates a criterion that measures the cross-entropy between the distribution of the respective target y_n and the distribution of the output of the propagation. In the `PyTorch` implementation, the function allows to add a weight argument, that is assigned to each of the classes to overcome the difficulties, described in the subsection on data sets, of an unbalanced training data set.

2.6.2 Binary Cross Entropy Loss

The binary cross entropy-loss, as the name suggests, allows only two classes Y and therefore also a binary output. With the way, the loss function is defined,

$$l_n = -w_n[y_n log x_n + (1 - y_n) log(1 - x_n)] \tag{2.6}$$

with $l(x, y) = \{l_1, ..., l_N\}\top$ and the fact that Y is either 0 or 1, one of the terms in the equation turns out to be 0. This circumstance is taken up in the implementation of GANs as a new class of test problems in Section 3.2.2.

Chapter 3

Implementation

The main contribution of this work is, to extend DeepOBS with novel test problems, therefore this chapter gives detailed insights on the implementation process of such. As obtained in the introduction, a balanced selection of test problems would be of great value for the optimizer benchmark suite. Before setting the focus on introducing a whole new class of test problems to DeepOBS (3.2), this work starts with a few test problems existing in the `TensorFlow` framework that can be adapted for `PyTorch` by modifying the *Model Loading* module of the pipeline, shown in Figure 2.1, only.

3.1 Translate test problems to `PyTorch`

The following list gives a brief explanation of the test problems, most often have been directly translated from `TensorFlow` to `PyTorch`. All of these are classification tasks of varying complexity.

- **Logarithmic regression with** f-mnist: Analogous to the existing logarithmic regression for mnist in `PyTorch`, this test problem was created for f-mnist. The authors Xiao et al. consider fmnist to slightly raise the level of difficulty for test problems([XRV17]).
- **Convolutional neural network with** SVHN: This test problem already exists for the data sets Cifar-10 and Cifar-100, which both have a size of 32x32 pixel and as the names suggest, one has 10 and the other 100 classes to distinguish. SVHN may resemble Cifar-10 in size and classes but it includes some distractors in the background of the images, making the task more challenging. The architecture is kept as implemented with three convolutional and three fully-connected layers.
- **VGG variants with** Cifar-10 **and** Cifar-100: For each data set two variants of the VGG architecture reference have been created respectively,

one with 16 and the other with 19 layers. The architecture of the model is adapted from the original VGG `TensorFlow` implementation that was designed for the ImageNet data set [SZ15].

- **Variants of the wide residual network with** Cifar-100: In the Tensorflow implementation two versions of the wide residual network are provided. One is using six residual blocks (16-4) and the other two (40-4). For Cifar-100 both the 16-4 and a 40-4 wide residual network have been adopted into the `PyTorch` framework, to offer direct comparison with the same test problem for the SVHN data set.

3.2 Introducing GANs to DeepOBS

Now the core part of the contribution of this work outsets, the implementation of a new class of test problems. Considering the recent impressive achievements of generative models stated in the introduction and the popularity of GANs in special, there are strong enough reasons for the need to implement such in a deep learning optimizer benchmark suite. However, the training process differs greatly from other test problems (3.2.3) and the evaluation comes with a lot of new challenges (3.2.5).

Since being introduced there have been many variations of GAN to improve their performance along with the attempt to stabilize the training process [MGN18, SGZ+16]. This results in a broad selection of possible architectures. To provide a good starting point for the decision, the weaknesses that appear in the training process of GAN, are to be considered.

- **Non-Convergence:** This happens when the minimization of cost for the discriminator and the generator with gradient descent methods leads the gradient to rather enter a stable orbit than maintaining a Nash equilibrium. This results in parameter oscillation and therefore the model destabilizes and never converges.

- **Mode collapse:** This is considered one of the harder problems to solve in GANs and describes the case in which the generator produces only a few or even just one single mode of outputs, meaning that the generator collapsed to a certain parameter setting. This can happen easily when the discriminator is exposed to only one example in an iteration, because it does not take into account any relations between its gradients for each example. In more detail, the gradient of the discriminator points in similar directions for similar points and therefore at first predicts the examples shown as realistic, while the generator emits the same point every time. After collapse, the discriminator learns which point comes from the generator and no convergence with sufficient entropy is achieved.

- **Overfitting:** In the original GAN paper, Goodfellow et al. state that a balance between the discriminator and generator updates is crucial to avoid overfitting and the resulting non-convergence, for training the discriminator to completion on a finite data set.
- **Hyperparameter sensitivity:** A small change in the hyperparameters can result in huge variations of a GAN's performance and therefore may cause one of the above problems. This effect is visualized in Section 4.2.1.
- **Cost function and image quality:** When classifiers are trained, the loss and the resulting accuracy are used to monitor the progress of a model. Yet for GAN the loss function estimates the performance of the network compared with the opponent. This means that even though the generator cost function increases, the image quality may be improving, which makes the evaluation and comparison between different models even harder along with complicating the tuning process.

For the purpose of an optimizer benchmark, the architecture of the model should consist of standardized and established elements, that keep the training process in appropriate limited time but still offer a significant problem representation for state-of-the-art tasks to solve by generative models. Therefore developments that seem quite promising but are of a rather exotic kind do not apply for the DeepOBS environment.

3.2.1 Setting up the architecture

For the implementation, the deep convolutional GAN (DCGAN[1]) architecture is adopted, introduced by Radford et al. [AR16]. It is a direct extension of the GAN described in Chapter 2 and consists exclusively of convolutional and transposed convolutional layers. The general structure of the network is shown in Figure 3.1. In this case, the generator consists of five transposed convolutions, each of them is followed by a batch normalization layer using ReLU as the activation function. An exception is the last layer, which is activated by the Tanh function without batch normalization. The image gets scaled up from the input vector Z to a 3x64x64 pixel image.

The discriminator network works analogously to the generator but in reverse. Using classical convolutional layers, explained in Chapter 2, followed by batch normalization with LeakyReLU as the activation function. The input layer does not use batch normalization, for the reasons of stability (see the original paper for a detailed explanation [MGN18]). Just as with the generator, the last layer makes an exception, as it is activated by the Sigmoid function. All convolutions, whether they are transposed or not, use a filter or kernel size of four.

[1]Code available at https://github.com/pytorch/examples

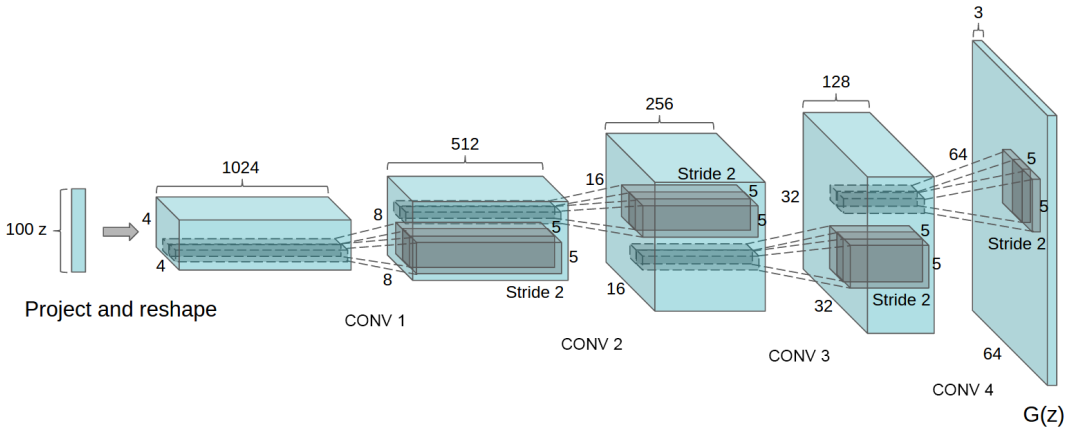


Figure 3.1: The architecture of the generator network. An input vector Z is fed into four transposed convolutional layers, scaling it up to a two-dimensional image $G(z)$ of size 64x64 with three channels according to the given noise vector. Adopted from [AR16]

3.2.2 Implementing the loss functions

As stated in the original paper [GPAM+14], the generator is trained by minimizing $log(1 - D(G(z)))$ in an effort to generate better fakes. This was shown by means of the authors to not provide sufficient gradients, especially early in the learning process. As a fix, instead $log(D(G(z)))$ is maximized. In the code, this is accomplished, by classifying the generator output with the discriminator and computing the generator's loss using real labels. On that basis then the gradients are computed in a backward pass, and finally the generator's parameters are updated with an optimizer step. It may seem counter-intuitive to use the real labels as labels for the generator loss function but as the BCELoss in PyTorch is defined as

$$l(x, y) = L = \{l_1, \ldots, l_N\}^\top, l_n = -w_n[y_n log(x_n) + (1 - y_n) log(1 - x_n)], \quad (3.1)$$

it allows, to use the $log(x)$ part of the function for the generators objective rather than the $log(1 - x)$ part. Therefore now the desired part to use from the BCE equation can be directly addressed with the y input.

3.2.3 Training process

The discriminator and the generator compete against each other, trying to minimize their loss function and thereby maximize that of the opponent, this is referred to as a minimax game. Therefore, a GAN uses two optimizers, one for the discriminator updates and one for the generator updates. In state-of-the-art approaches of GANs, often different optimizers or at least the same optimizer, with different training parameters is used. Due to the

high tuning and benchmarking effort, that comes with this adaptation, the same configuration for both optimizers has been used, in this work. Yet future extensions, which support the selection of different optimizers for a DCGAN test problem, are possible.

The training process starts with the discriminator iterating over the training data set for each batch in the data loader, one epoch denoting a whole cycle through the data set. In each iteration, the discriminator sets its gradients to zero, performs a forward pass with the real batches, evaluates the loss for them, and then computes the new gradients in a backward pass.

In the next step, the discriminator is trained with fake batches and the generator is fed with noise, to generate fake image batches. The output is then classified by the discriminator. The gradients for the fake batch are added to the gradients from all real batches to perform an update step by the optimizer.

Now the generator is about to be updated. Therefore the gradients of the generator are set to zero before the fake images get assigned to real labels to calculate the costs and then perform another forward pass with the all fake batch through the discriminator. Afterward, the gradients get calculated to perform an update step for G.

The values of both loss and accuracy function get saved for measuring training statistics in every iteration.

3.2.4 Setting up instances for the data sets

Now that the model architecture and the training schedule are defined, building up test problems within this new class is due. For a broader range of comparisons, four test problems, classified in different levels of difficulty for the optimizer, are created. Two of these are considered as less complex because of the fact, that they are trained on data sets with images that have only one color channel, MNIST, and F-MNIST. In this case, both instances already existed in the DEEPOBS library only not for the DCGAN. For test problems that generate colored images, two data sets have been introduced to DEEPOBS. One is the quite popular Celebrity Faces CELEBA [LLWT15] data set, which comes with over 200.000 images and rises the level of difficulty because of the fact, that humans have a sensitized perception of human faces. The other is the Animal Faces HQ AFHQ [CUYH20] data set, which only contains around 16.000 images in total, of dogs, cats, and wildlife animals, such as tigers or foxes. The size of the images is not taken into account here, because for the DCGAN implementation every input is resized, before being presented to the discriminator.

Below the four test problems of that new class of generative models are listed according to their expected level of difficulty from top to bottom:

- `mnist_dcgan`

- `fmnist_dcgan`
- `afhq_dcgan`
- `celebA_dcgan`

3.2.5 Methods for Evaluation

"While several measures have been introduced, as of yet, there is no consensus as to which measure best captures strengths and limitations of models and should be used for fair model comparison."[Bor18]. A GAN has two models, the discriminator, and the generator, training each other in a minimax game. Minimizing one loss function results in increasing the other, meaning the accuracy denotes how one model is performing compared to its opponent. Therefore the generator, that is trained by the discriminator through classification of real and generated images, has no objective function to directly measure and compare performance [SGZ+16] as well as finding a low value of a loss function is not quite helpful for the evaluation of such a minimax game. Furthermore, only local Nash equilibria can be found, as gradient descent is a local optimization method.

Qualitative Evaluation: In the rather short research history of GANs, there have been several approaches on how to evaluate a GAN or more generally the quality of a generated image. Intuitively it seems more reliable to measure the quality and diversity of the generated images. This can be achieved through several procedures:

- Rating and Preference Judgment: This is a quite popular way, where participants are asked to rank samples in terms of fidelity of the images, though the judgment is not fixed as it usually progresses over time, by cause of the judges getting more sensitive to the features, that indicate a fake or real sample.
- Rapid Scene Categorization: This procedure works analogous to the one above, with participants distinguish real from fake images but within a short presentation time. In this case, the samples are set next to each other for the evaluation. The variance in the judgments is tried to overcome by averaging the ratings. Thus this procedure is labor-intensive.
- Nearest Neighbors: Here representatives from the generated images are placed next to their nearest neighbors or at least the most similar-looking real version of samples in the training data set. This procedure can therefore help to detect whether the model is overfitting and additionally promises a good context for evaluating how realistic a generated image appears.

For all empirical subjects in the deep learning field, the success of an objective is determined by using evaluation metrics for performance measures which are developed and widely accepted by the research community. In the case of GANs, convergence theory is still highly researched and no performance metric is solidly established in the deep learning field [Bor18, LKM+17, BS18]. Considering these issues and the limited time frame of this work, the decision was made to provide a way to visually evaluate the results of the DCGAN test problems, yet not as a part of the automated benchmark process. The qualitative evaluation method used is most related to the nearest neighbors procedure, plus providing a way of visually tracking the progress of the model, the reason for this occurrence is, that in a fixed setting this depends on the optimizer, which is the true objective of the evaluation here. Following implementations have been done:

- Creating a vector in every run, that serves as input noise for the generators evaluation. Feeding this input vector into the generator, when the evaluation starts and generate images out of it. As a visual evaluation of one image is not as reliable as of various fakes, representatives of these are arranged in an 8x8 grid. Which results in 64 generated images for visual comparison.

- This grid of images is then saved as portable network graphics file in every intended epoch to visually track the evolvement of the generator. The images are saved in an extra folder within the path that is created to save the output file in an organized folder hierarchy. The folder naming provides a direct distinction between the test problem, the optimizer, the number of epochs a model is trained for, along with the configuration of the hyperparameters.

- On account of the number of epochs being relative to the amount of training data available, the possibility to leave it the user's decision at which epoch a test problem shall be evaluated. This can be handed over directly to the `run()` function, when starting the training for a test problem.

- At the end of the training, an additional image is created, which offers the opportunity to compare the final products of the generator directly next to samples from the data set.

Quantitative Evaluation: The above evaluation method mainly serves to detect the general functionality of a DCGAN test problem and therefore mode collapses or overfitting. However, it can not sufficiently meet the requirements of the DeepOBS library, which demands a quantitative metric, to provide a

reliable comparison of the optimizer. Therefore additional measurements are taken into account.

Goodfellow used the average Log-Likelihood method in his original paper [GPAM+14], which has generally been found not to be effective: "Parzen windows estimation of likelihood favors trivial models and is irrelevant to visual fidelity of samples. Further, it fails to approximate the true likelihood in high dimensional spaces or to rank models"[Bor18].

Another approach that has to be named in this context because it seems very promising and is still used widely in research, is the **Inception score** [SGZ+16]. Applying this metric requires a pre-trained model, for the image classification of the generated images, which gives the probability of an image belonging to each class. The inception score measures the variety of images and how much the images "look" like the distribution of a known class. In the event, that both arguments are sufficient the score is high otherwise it will be low, meaning the higher the inception score, the better is the quality of the generated images. The actual output is a conditional label distribution $p(y|\mathbf{x})$, with low entropy in cases where the image contains meaningful information and a high entropy for the marginal $\int p(y|\mathbf{x} = G(z))dz$ when the model creates varied images. Salimans et al. combine these requirements and propose to exponentiate results for easier comparison: $exp(\mathbb{E}_x \mathbf{KL}(p(y|\mathbf{x})||p(y)))$. Therefore they use the Kullback-Leibler (KL) divergence, a formula to measure the similarity or difference between to probability distributions.

Despite the appealing properties of this evaluation method, it suffers from several limitations. Ali Borji investigated these in his paper [Bor18]. He states that the Inception score lacks in the detection of overfitting and mode collapse, along with the constraint of sensitivity to variations in image resolution. All these are typical challenges that arise when training GANs.

The **Frechet inception distance** [HRU+17] is an improvement over the inception score: "FID performs well in terms of discriminability, robustness, and computational efficiency. [...] It has been shown that FID is consistent with human judgments and is more robust to noise than IS."[Bor18]. This metric uses the Inception-V3 model to extract features from a hidden layer to model the data distribution of these features. This is done by using a multivariate Gaussian distribution with mean m and covariance C to measure the distance d between the distribution of the real (m_{data}, C_{data}) and the generated images (m_z, C_z),

$$d^2((m_z, C_z), (m_{data}, C_{data})) = ||m_z - m_{data}||_2^2 + Tr(C_z + C_{data} - 2(C_z C_{data})^{1/2}). \tag{3.2}$$

The Tr denotes the sum of all diagonal elements. The lower the FID score, the more the images match the statistical properties of real images.

The FID has some limitations, as it is by means of its definition assuming that features in an image are of Gaussian distribution, which can not be

guaranteed in principle or that the detection of overfitting is not solved with the original FID implementation. Still, it is a very promising approach to evaluate the performance of an optimizer in DeepOBS. It is possible to integrate it in the automatized benchmark process and preparations to simplify the implementation have been done, however, the time frame of this thesis did not allow further investigations in this first release of GANs in DeepOBS. The evolvement of quantitative evaluation methods for GANs is a critical direction for scientists. It directly impacts the development of the DeepOBS library, in the sense of the need to continuously improve the benchmark process regarding state-of-the-art approaches on the evaluation of generative models.

Chapter 4

Experiments

4.1 Examining the novel test problems

For a wholesome introduction of the novel class of test problems, the basic functionality of the DCGAN concept within the DeepOBS architecture is to be proved (4.1). The performance of the test problems is visually measured from their respective output, indicating the correctness of all aspects of the implementation (4.1.2). Not only the results of the generator within the DeepOBS environment are illustrated, but a direct comparison with representatives from the training data set is provided (4.1, 4.2) and in one case is supplemented by a comparison (4.3), with results from the `PyTorch` tutorial [NI17].

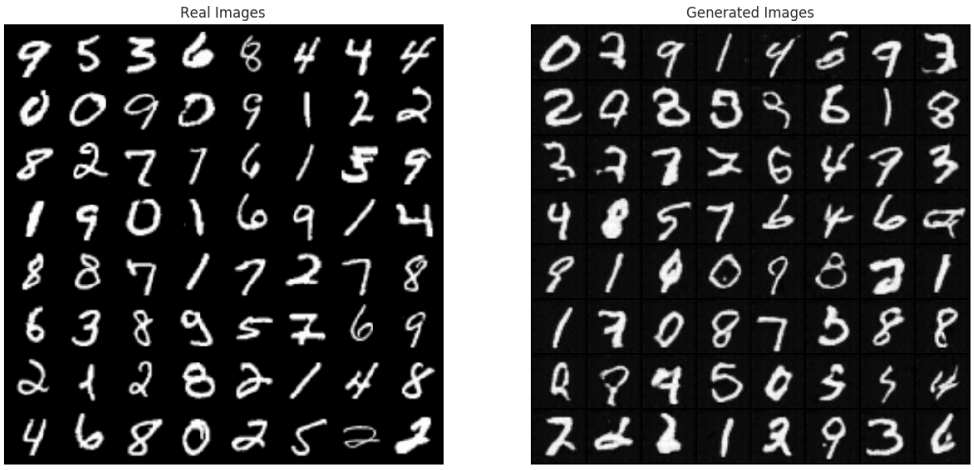
4.1.1 Setting for the evaluation

For a meaningful analysis of the DCGAN test problems, the optimizer, and the training parameters proposed in the original paper [AR16], are used for the training process. Though in the paper only data sets with colored images have been used, the setting may apply for the test problems using grey-scaled data sets as well. The beta parameters of the Adam optimizer, are adjusted to (0.5, 0.999), which is a slight variation to the default settings (0.9,0.999), for which the authors found out that it destabilizes the training and results in oscillation [AR16]. In addition, the learning rate (lr) is changed to 0.0002, as the default value of 0.001 is too high for good performance [AR16].

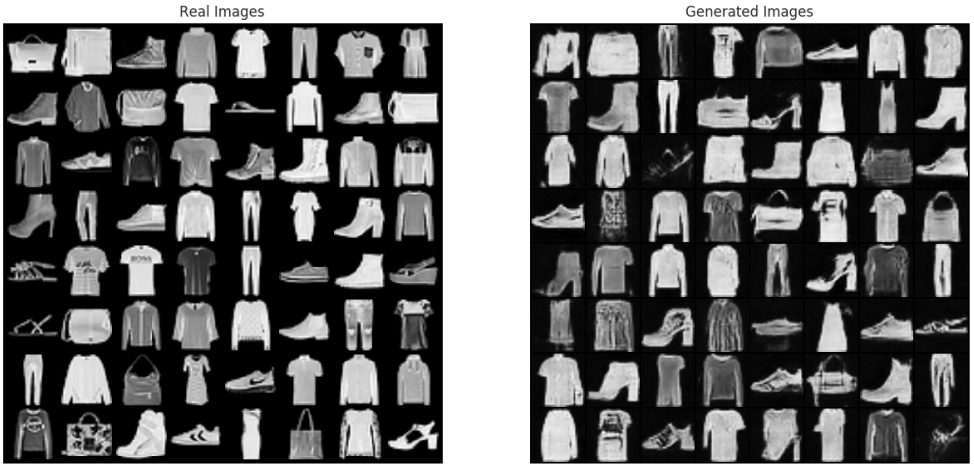
Further, the training of a DCGAN, and GANs, in general, is also highly sensitive to changes in the batch size. Thus the choice for this parameter is based on the guidelines of the paper as well, where a batch size of 128 turned out to be appropriate for stable training. The epochs at which the generator of a test problem is evaluated in this part of the section depends on the amount of available training data.

4.1.2 Results

The first two figures (Figure 4.1 and Figure 4.2) offer a direct comparison between an 8x8 grid of images from the training data set next to generated images of the DCGAN model. Figure 4.1 shows the test problems for the data sets MNIST and F-MNIST, which both have one color channel and have been trained for 20 epochs each.



(a) `mnist_dcgan`



(b) `fmnist_dcgan`

Figure 4.1: Representatives of the training data set and the generated images from the DCGAN model next to each other for the test problems a) `mnist_dcgan` and b) `fmnist_dcgan`. Both models have been trained for 20 epochs.

Based on the visual assessment, all DCGAN test problems respectively, create images that match the training data. Especially in Figure 4.1 distinguishing real from generated images is not an easy task for a human judge. The `mnist_dcgan` creates mostly clear numbers with a few slightly blurred exceptions and the

(a) `afhq_dcgan`

(b) `celebA_dcgan`

Figure 4.2: Representatives of the training data set and the generated images from the DCGAN model next to each other for the two data sets a) AFHQ and b) CELEBA, both with three color channels. The **`afhq_dcgan`** has been trained for 40 epochs, whereas the **`celebA_dcgan`** has been trained only for 5 epochs.

`fmnist_dcgan` can create sharp pictures of clothes, some of which leave the question unanswered whether some extravagant designer has been at work here or a GAN. However, the colored images from the `afhq_dcgan` and the `celeba_dcgan` can still easily be classified as fake. One can see the features that form the face of an animal or a human being, which allow to assign a picture to a dedicated class in the data set, for example, cats or dogs.

Figure 4.3 offers an additional way of visual comparison by putting the results of the implementation from the `PyTorch` tutorial in relation to the results from the `afhq_dcgan` test problem. Both models are trained for five epochs on the CelebA data set, to allow a reasonable comparison of the progress of the test problems. Though a visual assessment, can not offer an exact estimation, none of the test problems seems to outperform the other.

In the purpose of this section, the visual presentation of the capabilities of the DCGAN test problems does not need an additional quantitative method, in order to confirm the functionality in the DeepOBS environment. The images in the figures (Figure 4.1, Figure 4.2 and Figure 4.3) demonstrate, that each of the novel test problems can generate data, which tends to match the training data when using a specific optimizer and hyperparameter setting.

(a) `PyTorch` tutorial **(b)** DeepOBS implementation

Figure 4.3: Results from the `celeba_dcgan` test problem a) in the `PyTorch` tutorial [NI17] compared with b) the results from the DeepOBS implementation, both after five epochs of training.

4.2 Illustrating the training characteristics

In the implementation, the unique challenges and weaknesses of a GAN's training process, such as hyperparameter sensitivity (4.2.1) or mode collapse, have been discussed. In the experiments, with the optimizer SGD, MOMENTUM, and ADAM, different hyperparameters have been used, to visually track the progress of the respective optimizer for the four novel test problems. During these experiments, the results of the above-mentioned weaknesses showed up in the training process and can be observed, by interpreting the images (Figure 4.4, Figure 4.5, and Figure 4.6) created during the training.

4.2.1 Hyperparameter sensitivity

The high sensitivity towards the training parameters, mentioned in the implementation (Section 3.2), became particularly visible when using ADAM or MOMENTUM, with different values for the beta or momentum term.

The generated images are illustrated in grids of 8x8 pictures, starting with the first evaluations in the top row, from left to right, and proceeds with the next two evaluation steps in the bottom row, again from the left to the right side.

For ADAM, all hyperparameters are equal to the setting in Section 4.2, except for the beta parameter, which is changed back to the default value of 0.9. The `afhq_dcgan` seems to learn some features, that appear as blurred areas with high contrast. One can see, that there is no mode collapse, as every image in the twentieth epoch looks different (Figure 4.4a). The `celeba_dcgan` creates images that clearly show abstract representations of faces (Figure 4.4b). Though the images indicate an improvement of the optimizer, there is a big performance drop compared to the test problems, using ADAM with an adjusted beta parameter (Figure 4.4).

The MOMENTUM optimizer the learning rate has been adjusted to 0.001, as it has shown to accelerate performance, compared to a learning rate of 0.0002. All other hyperparameters remain at their default values for the beginning, which means the momentum term has a value of 0.9. Figure 4.5 shows the results, achieved with the DCGAN test problems.

The `mnist_dcgan` and the `fmnist_dcgan` hardly allow any interpretation of the patterns in the generated images. Though for both, the images show a slightly different pattern after the fifth and the fifteenth epoch, no features at all can be detected. A human judge is not able to tell, which images are generated early in the training or at the end.

The `afhq_dcgan` test problem (Figure 4.5b) appears to be stuck in the oscillations, described in the DCGAN paper to show up when the momentum term is not reduced [AR16]. Indeed the output of the tenth and the thirtieth epoch look similar, just like the twentieth and the fortieth output.

(a) `afhq_dcgan` at 5, 10, 15 and 20 epochs **(b)** `celeba_dcgan` at 2, 3, 4 and 5 epochs

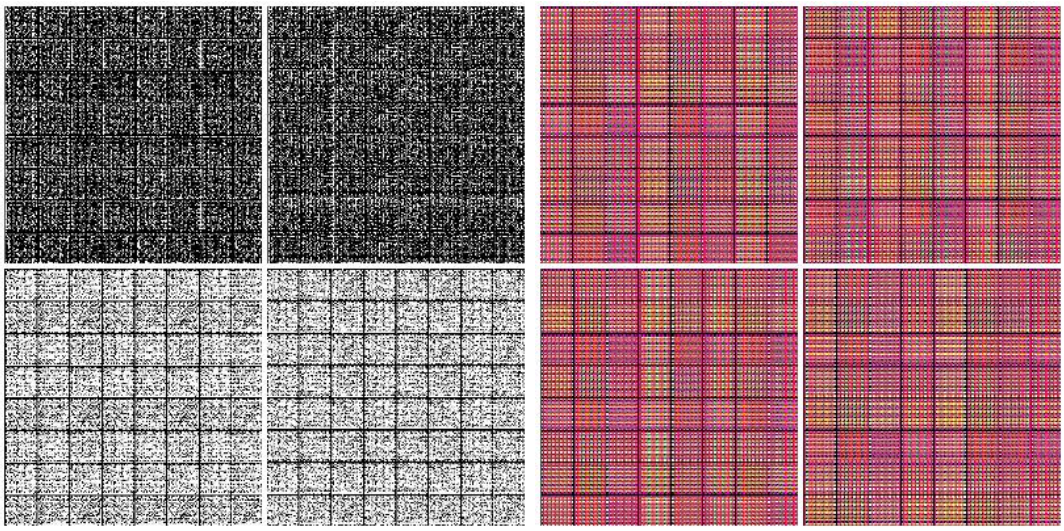
Figure 4.4: Running ADAM with default settings and a learning rate of 0.0002 on DCGAN test problems with three color channels `afhq_dcgan` and `celeba_dcgan`. Showing the generated images at different steps of the training process.

Now the momentum term has been adjusted to 0.5, to enhance the difference in performance. Looking at Figure 4.6, the improvement is huge. Not only do the test problems, learn the features from the training data, but the goal of the generators can be classified. Particularly the `mnist_dcgan` generates handwritten numbers (Figure 4.6a), mostly similar to the data set. The `afhq_dcgan` already generates images, in which animal faces can be identified, but a further classification of the classes of animals is not possible.

At last, SGD is run on the test problems, using the same learning rate as it is chosen for MOMENTUM above shown in Figure 4.7.

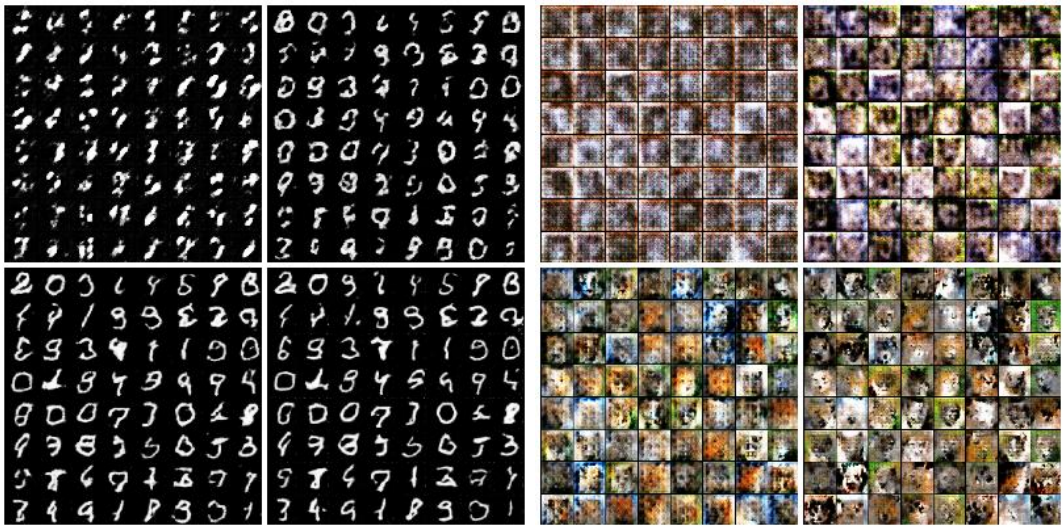
The `mnist_dcgan` and the `fmnist_dcgan`, which are trained for the same amount of epochs and are generally considered as the easier test problems, generate more or less noise. On the right side of the bottom row in Figure 4.7a, the pixels appear to be organized according to some structure, but as every image looks the same, it is not yet possible to decide whether the model learns very slow with SGD or whether this is the illustration of mode collapse.

What is unusual here, is that the `afhq_dcgan`, which is considered to be more complex, achieves to learn different features (Figure 4.7b). Although, it is not yet possible to identify what the generator tries to generate, without knowing the training data.



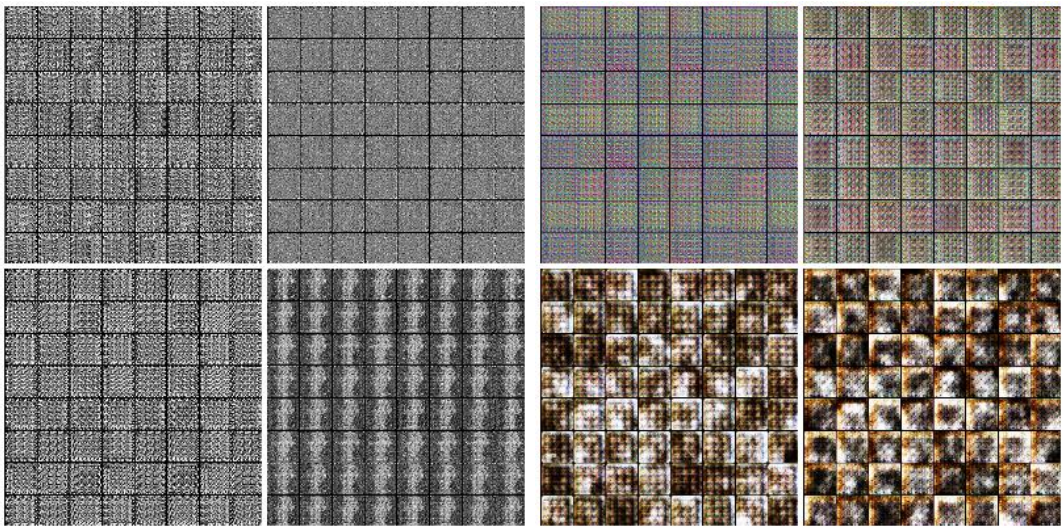
(a) MNIST top row and FMNIST bottom row **(b)** `afhq_dcgan` at 10, 20, 30 and 40 epochs

Figure 4.5: Running the MOMENTUM optimizer with a learning rate of 0.001 and default MOMENTUM parameter of 0.9 on DCGAN test problems. The image on the left a) shows the two test problems using grey-scaled images, at epoch 5 and 10. On the right side, the output of the `afhq_dcgan` is placed, trained till the fortieth epoch.



(a) MNIST at epochs 2, 5, 10, and 15. **(b)** `afhq_dcgan` at epochs 10, 20, 30, and 40

Figure 4.6: Running the MOMENTUM optimizer with a learning rate of 0.001 and an adjusted momentum term of 0.5 on DCGAN test problems. The image on the left a) shows the `fmnist_dcgan` problem at epoch 2, 5, 10, and 15. On the right side, the output of the `afhq_dcgan` is placed, trained till the fortieth epoch.



(a) MNIST top row and FMNIST bottom row **(b)** `afhq_dcgan` at 5, 10, 15 and 20 epochs

Figure 4.7: Running SGD with a learning rate of 0.001 on DCGAN test problems. The image on the left a) shows the two test problems using greyscaled images, at epoch 5 and 20, respectively. On the right side, the output of the `afhq_dcgan` is placed, trained until the twentieth epoch.

4.2.2 GANs can be deceptive

Another weakness of the training process of GANs, is that even though the output of a model appears to progress in the right direction, this can collapse within one epoch.

In the experiments, this effect was detected twice, with the `fmnist_dcgan` and the `celeba_dcgan`. Both have been trained with the MOMENTUM optimizer, with an adjusted learning rate of 0.001 and a momentum term of 0.5.

In the first thirteen evaluation steps, the generator of the `fmnist_dcgan` learns to map the distribution of the input data quite well (Figure 4.8). From there on, the model suddenly collapses and generates nothing but noise. The same effect, even though less severe, appeared with the `celeba_dcgan` (Figure 4.9). In the first epoch, the generator learned some patterns of different colors. With a more detailed look, one can see a pixelated shape of a head, in most images. The output of the first epoch is the initial noise vector for the generator, and is only presented to enhance the progress in the first epoch.

These experiments let the question arise, whether a model is able to rehabilitate from this stage, which is left open for future examinations.

Figure 4.8: The `fmnist_dcgan`, using the MOMENTUM optimizer. The top row shows the images generated at epoch 2, 4, 6, and 8. In the second row, epochs 10, 12, 13, and 14, are presented, from left to right. With a value of 0.5 for the momentum term, the model learns to match the distribution of the training data but collapses after the fourteenth epoch.

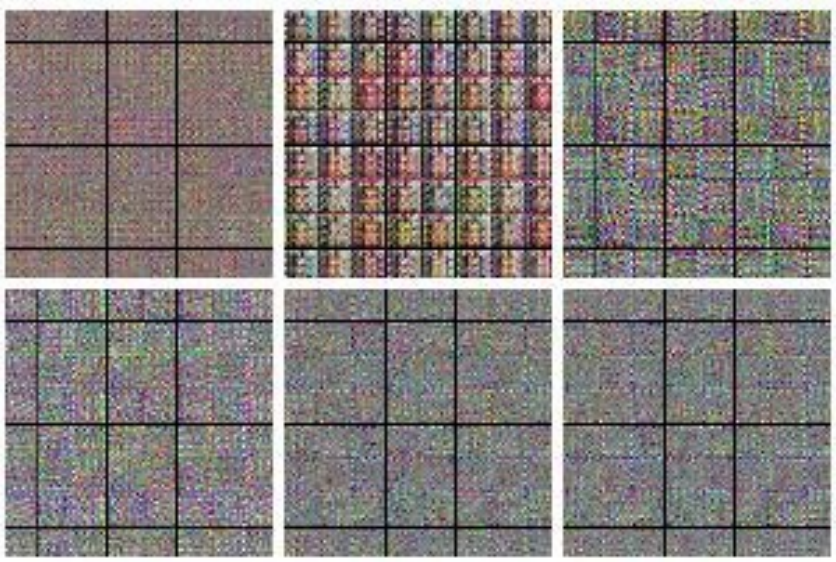


Figure 4.9: The `celeba_dcgan`, using the MOMENTUM optimizer. The top row, shows the images generated at epoch 0, 1, and 2. In the second row, epochs 3, 4, and 5 are presented, from left to right. With a value of 0.5 for the momentum term, the model first seems to learn some features, but then quickly collapses.

Chapter 5

Conclusion and Outlook

The main contribution of this work is the implementation of a novel class of test problems as an additional representation for deep generative models, within the deep learning optimizer benchmark, DeepOBS. This was achieved by several steps within this work:

- Finding and integrating a stable and standardized model architecture, that requires less time-consuming training, while striking a class of test problems that can serve as a representation for real-world applications and tasks.
- Creating four instances of this new class of test problems for the `PyTorch` framework in DeepOBS.
- Implementing the process for adversarial training of DCGANs in the DeepOBS environment, which is not trivial due to the fact, that the original training protocol does not apply for this architecture. The DCGAN demands alternating update steps for the two optimizers to improve the performance of the generator and the discriminator in concert, and therefore a whole new training procedure had to be integrated.
- Finding an appropriate evaluation method for the DCGAN test problems was a challenging part, in light of the fact that convergence theory of these, is still extensively researched. In this work, quantitative and qualitative methods have been investigated in detail, before deciding to stay with the implementation of qualitative methods at first.
- With the experiments [1], the functionality of the DCGAN concept, within the DeepOBS environment was proven, for each of the new test problems. Further, the hyperparameter sensitivity that is typical for

[1] For an overview of all experiments, done in the context of this work, see: https://github.com/Vanessa-Ts/DeepOBS_BA

GANs is illustrated by examples from the experiments. These come along with results, that enhance the fragility of the training process, in the form of models that collapse, despite of the fact, that these initially developed promisingly.

For a better overall insight on the implementations of this thesis, an updated version of Table 1.1, is shown in Table 5.1. It illustrates these new test problems, dedicated to the generation tasks marked in orange, as well as the far more equal distribution of test problems in general over the two frameworks. In total four new test problems have been introduced, for the new class within the generation tasks, along with eight classification problems for `PyTorch`.

The decision to implement particularly a version of a GAN, supplements the diversity of tasks in DeepOBS, with a unique model architecture as well as a completely different training and evaluation process.

Offering state-of-the-art models and tasks of the deep learning field, for the evaluation of new optimizer, is essential for the benchmark suite, as it indicates the significance of an optimizer's results within this environment. Therefore the transfer of the classification tasks to the `PyTorch` framework and especially the introduction of a novel class of test problems directly strengthens the competitiveness of the library.

Despite this progress, the implementation of this work has room for improvements. Though qualitative evaluation methods guarantee a reliable classification, whether an image appears realistic or not, they have several drawbacks within an optimizer benchmark. Qualitative techniques do allow a classification of optimizer when the performances differ greatly but not when an exact comparison of almost similar images is needed. Further, the process of human judges examining images to benchmark optimizer in the long run is neither realistic nor can be automated.

5.1 Future Work

The implementation of a GAN model in DeepOBS enables new challenges, on which an optimizer can be tested, in order to prove its quality. For example, researchers from the Max Planck Institute for Intelligent Systems, who work with GANs, suggest applying the RMSProp[2] optimizer [MGN18] for the training. Investigating the performance of this one with the DCGAN test problems, is an interesting setting, for future experiments.

Further, there is no unified measurement in DeepOBS by now, that allows an exact comparison of an optimizer's performance across all generation tasks. A more precise evaluation and comparison, of the respective optimizer, could be

[2]http://www.cs.toronto.edu/ tijmen/csc321/slides/lecture_slides_lec6.pdf

Table 5.1: Status update on the overview of the test problems included in the DeepOBS library, separated by data set and model, and whether there exists an implementation in the `TensorFlow` or the `PyTorch` framework. The colors illustrate the new distribution of classes of test problems. Image classification ● is still the most common, image generation ● is now represented six times with variable complexity. Natural language processing ● stays so far the least represented class of tasks and the 2D problems ● remain unchanged as well.

Data set	Model	Description	Framework
2D	● Noisy Beale	*Noisy version of Beale function*	
	● Noisy Branin	*Noisy verson of the Branin function*	
	● Noisy Rosenbrock	*Noisy version of the Rosenbrock function*	
Quadratic	● Deep	*100dim ill-conditioned noisy quadratic*	
MNIST [LBBH98]	● Log. Reg.	*Logistic regression*	
	● MLP	*Four layer fully-connected network*	
	● 2c2d	*Two conv. and two fully-connected layers*	
	● VAE	*Variational Autoencoder*	
	● DCGAN	*Deep convolutional generative adversarial net*	
Fashion MNIST [XRV17]	● Log. Reg.	*Logistic regression*	
	● MLP	*Four layer fully-connected network*	
	● 2c2d	Two conv. and two fully-connected layers	
	● VAE	*Variational Autoencoder*	
	● DCGAN	*Deep convolutional generative adversarial net*	
AFHQ	● DCGAN	*Deep convolutional generative adversarial net*	
CelebA	● DCGAN	*Deep convolutional generative adversarial net*	
Cifar-10 [Kri09]	● 3c3d	*Three conv. and three fully-connected layers*	
	● VGG 16	*Adapted version of VGG16 [SZ15]*	
	● VGG 19	*Adapted version of VGG19*	
Cifar-100 [Kri09]	● 3c3d	*Three conv. and three fully-connected layers*	
	● VGG 16	*Adapted version of VGG16*	
	● VGG 19	*Adapted version of VGG19*	
	● All-CNN-C	*The all convolutional net from [SDBR14]*	
	● Wide ResNet-16-4	*Wide Residual Network [ZK16]*	
	● Wide ResNet-40-4	*Wide Residual Network [ZK16]*	
SVHN [NtWC+11]	● 3c3d	*Three conv. and three fully-connected layers*	
	● Wide ResNet-16-4	*Wide Residual Network*	
IMAGENET [DDS+09]	● VGG 16	*Adapted version of VGG16*	
	● VGG 19	*Adapted version of VGG19*	
	● Inception-v3	*Inception-v3 net as described by Szegedy et al. [SVI+16]*	
Tolstoi	● CharRNN	*Recurrent NN for character-level language modeling*	

achieved with the addition of quantitative metrics, such as the FID [HRU+17], explained in Section 3.2.5. Applying this metric to the generated samples of a VAE would close the gap between the different evaluation metrics for generative models in DeepOBS.

In addition, combining this work with the FID calculation can extend the possibilities, for researchers in the deep learning field, to investigate performance and convergence of GANs, in a highly automated and unified environment. As many optimizers and training settings for GANs tend to fail, in the form of non-convergence (Section 4.2), it would be preferable to find reliable termination

conditions for the training process of DCGAN test problems in DeepOBS, that indicate this effect. Implementing such a quantification is left open for future extensions, as further analysis and experiments are demanded here, which is beyond the scope of this work.

In order to increase the value and significance and competition of the results of the benchmark, there are still various test problems and classes of these same to extend DeepOBS with. The DCGAN is only one of the various architectures within the family of GANs and there exist many more interesting classes within the deep learning field, such as reinforcement learning [SDG16].

With the extensions stated above, this work can help to further investigate the notion of the deep learning community that some optimizers perform better in certain kinds of tasks.

Bibliography

[Ma15] Martín Abadi and Ashish Agarwal and Paul Barham and Eugene Brevdo and Zhifeng Chen and Craig Citro and Greg S. Corrado and Andy Davis and Jeffrey Dean and Matthieu Devin and Sanjay Ghemawat and Ian Goodfellow and Andrew Harp and Geoffrey Irving and Michael Isard and Yangqing Jia and Rafal Jozefowicz and Lukasz Kaiser and Manjunath Kudlur and Josh Levenberg and Dandelion Mané and Rajat Monga and Sherry Moore and Derek Murray and Chris Olah and Mike Schuster and Jonathon Shlens and Benoit Steiner and Ilya Sutskever and Kunal Talwar and Paul Tucker and Vincent Vanhoucke and Vijay Vasudevan and Fernanda Viégas and Oriol Vinyals and Pete Warden and Martin Wattenberg and Martin Wicke and Yuan Yu and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[Aar19] Aaron Bahde. Towards Meaningful Deep Learning Optimizer Benchmarks. Master's thesis, 2019. Eberhard-Karls-University Tübingen, Germany.

[AR16] Soumith Chintala Alec Radford, Luke Metz. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, 2016.

[Bor18] Ali Borji. Pros and Cons of GAN Evaluation Measures, 2018.

[Bra72] F. H. Branin. Widely convergent method for finding multiple solutions of simultaneous nonlinear equations. *IBM J. Res. Dev.*, 16(5):504–522, September 1972.

[BS18] Shane Barratt and Rishi Sharma. A note on the inception score, 2018.

[CCS+19] Pratik Chaudhari, Anna Choromanska, Stefano Soatto, Yann LeCun, Carlo Baldassi, Christian Borgs, Jennifer Chayes, Levent

Sagun, and Riccardo Zecchina. Entropy-sgd: biasing gradient descent into wide valleys. *Journal of Statistical Mechanics: Theory and Experiment*, 2019(12):124018, Dec 2019.

[CUYH20] Yunjey Choi, Youngjung Uh, Jaejun Yoo, and Jung-Woo Ha. Stargan v2: Diverse image synthesis for multiple domains. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2020.

[DDS+09] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[Die14] Diederik P. Kingma, Jimmy Ba. Adam: A Method for Stochastic Optimization, 2014.

[Doe16] Carl Doersch. Tutorial on variational autoencoders, 2016.

[DV16] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning, 2016.

[Fra19] Frank Schneider, Lukas Balles, Philipp Hennig. DeepOBS: A Deep Learning Optimizer Benchmark Suite, 2019.

[FSG16] Haoqiang Fan, Hao Su, and Leonidas Guibas. A point set generation network for 3d object reconstruction from a single image, 2016.

[GPAM+14] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Nets, 2014.

[Han19] Fanghao Han. Tutorial on variational graph auto-encoders, 09 2019. Accessed: 04.07.2020.

[He19] Horace He. The state of machine learning frameworks in 2019. *The Gradient*, October 2019. Accessed: 23.06.2020.

[HRU+17] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium, 2017.

[Ily18] Ilya Loshchilov and Frank Hutter. Fixing Weight Decay Regularization in Adam, 2018.

[KALL18] Tero Karras, Timo Aila, Samuli Laine, and Jaako Lehtinen. Progressive growing of gans for improved quality, stability, and variation, 2018. ICLR.

[Kar17] Andrej Karpathy. A Peek at Trends in Machine Learning, April 2017. Accessed: 20.06.2020.

[Kri09] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009.

[LBBH98] Yann Lecun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 12 1998. 86:2278-2324, doi: 10.1109/5.726791.

[LKM+17] Mario Lucic, Karol Kurach, Marcin Michalski, Sylvain Gelly, and Olivier Bousquet. Are gans created equal? a large-scale study, 2017.

[LLWT15] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.

[Mat12] Matthew D. Zeiler. ADADELTA: An Adaptive Learning Rate Method, 2012.

[MGN18] Lars Mescheder, Andreas Geiger, and Sebastian Nowozin. Which training methods for gans do actually converge? In *International Conference on Machine learning (ICML)*, 2018.

[Nes83] Yurii Nesterov. A method of solving a convex programming problem with convergence rate $\mathcal{O}(1/k^2)$, 1983.

[NI17] PyTorch Nathan Inkawhich. DCGAN TUTORIAL, 2017. Accessed: 02.07.2020.

[NtWC+11] Yuval Netzer, tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. *NIPS workshop on deep learning and unsupervised feature learning*, 2011.

[PGC+17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. 2017. NIPS.

[Pol64] Boris T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1-17, 1964.

[RM51] Herbert Robbins and Sutton Monro. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3):400-407, 1951.

[Ros58] Frank F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65 6:386–408, 1958.

[Ros60] H. H. Rosenbrock. An Automatic Method for Finding the Greatest or Least Value of a Function. *The Computer Journal*, 3(3):175–184, 01 1960.

[Rud17] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.

[RvdOV19] Ali Razavi, Aaron van den Oord, and Oriol Vinyals. Generating diverse high-fidelity images with vq-vae-2, 2019.

[Sah15] Sumit Saha. A comprehensive guide to convolutional neural network - the eli5 way, 12 2015. Accessed: 04.07.2020.

[SDBR14] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net, 2014.

[SDG16] Tanmay Shankar, Santosha K. Dwivedy, and Prithwijit Guha. Reinforcement learning via recurrent convolutional neural networks. *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 2592–2597, 2016.

[SGZ+16] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alex Radford, and Xi Chen. Improved Techniques for Training GANs, 2016.

[Sil18] Thalles Silva. An intuitive introduction to generative adversarial networks (GANs), 01 2018. Accessed: 02.07.2020.

[SRE+15] Christian Szegedy, Scott Reed, Dumitru Erhan, Dragomir Anguelov, and Sergey Ioffe. Scalable, High-Quality Object Detection, 2015.

[SSH20] Robin M. Schmidt, Frank Schneider, and Philipp Hennig. Descending through a crowded valley - benchmarking deep learning optimizers. *ArXiv*, abs/2007.01547, 2020.

[Sut86] Richard S. Sutton. Two problems with backpropagation and other steepest-descent learning procedures for networks, 1986.

[SVI+16] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. in. *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. 8pp. 2818-2826.

[SZ15] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[Tim16] Timothy Dozat. Incorporating Nesterov Momentum into Adam, 2016.

[XRV17] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

[Yi 16] Yi Xu and Qihang Lin and Tianbao Yang. Accelerate Stochastic Subgradient Method by Leveraging Local Growth Condition, 2016.

[Zey17] Zeyuan Allen-Zhu. Natasha: Faster Non-Convex Stochastic Optimization Via Strongly Non-Convex Parameter, 2017.

[ZK16] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks, 2016.

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore,I declare that I have acknowledged the work of others by providing detailed references of said work.I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date Signature