A (conditional) Generative Adversarial Network

Bastian Dittrich*

October 11, 2021

Abstract

In this report a Generative Adversarial Network (GAN) and a conditional Generative Adversarial Network (cGAN) are implemented with PyTorch and trained on MNIST and FashionMNIST. One focus is on explaining the implementation to give the reader some guidance in trying to implement their own GAN-project. In this context some tips and tricks from the literature are mentioned and evaluated through the experiments and tests within this project with the evaluation measures GAN-test and GAN-train.

1 Introduction

Machine Learning is one of the hot topics of todays research. One of its main big successes are the so called discriminative models which map high-dimensional data like pictures to low-dimensional data like words or numbers. The perhaps most popular version might be a classifier which sorts pictures into predefined classes like 'tree', 'cat', 'dog', 'house' etc. [23]. For a small number of classes which look very different, like dog and table, this is a very easy task for a classifier but for a huge number of partly similar looking classes this can be very hard. Popular databases for classification or other machine learning tasks are for example Cifar10/Cifar100 [22] and ImageNet [1].

In practice one often struggles to find enough data to train a classifier, i.e. with medical images [21]. The approach to increase the number of data is called data augmentation. One main technique is to slightly change the data one has, i.e. rotating a picture, changing some pixels etc. [23].

In 2014 Ian Goodfellow et al. [14] proposed a network structure which was not explicitly built for data augmentation but can be very well used for it, e.g. [5]. The network they invented is the so called Generative Adversarial Network (GAN), a pair of two networks working against each other. The goal of the first network, the generator, is to imitate data of the given data set while the second network, the discriminator, tries to distinguish between the fake data of the generator and the real data from the data set, hence the discriminator is a classifier with the two given classes real and fake.

During the learning process the generator tries to fool the discriminator into classifying the generated data as real while the discriminator continuously learns to differ

*Universität Würzburg, bastian.dittrich@stud-mail.uni-wuerzburg.de, Project for the AG "Theory and Implementation of Deep Learning Algorithms", Supervisor: Prof. Dr. Alfio Borzi. All codes are available at: `https://github.com/b-dittrich/cGAN`

between fake and real data. One difficulty lies in balancing the training process such that they rise together during training and in the end it would be the ideal case that the generator produces indistinguishable but new fake data.

Besides data augmentation this network structure has many different possible applications, i.e. generating music [10] or art [11], completing pictures, generating all kind of fake data like photo-realistic portraits of non-existing humans, etc. [8], the examples are nearly endless.
Due to the wide variety of applications in the following years this approach gained huge popularity and many different kinds of GANs were proposed (overview of some selected: [9]). Today this is "one of the most popular method[s] for generating images" [32], which does not mean it does not have problems. I will discuss some of them and explain how I tried to solve them in my implementation.

In this report I first explain in section 2 the structure of a GAN as well as of a cGAN, a slight modification of the original GAN structure. After that I describe in section 3 the basis of my experiments hence which data set I use and why, how I evaluate my experiments and how I choose my hyperparameters. Additionally I mention some problems that occurred to me during training a GAN and how I tried to solve them. In section 4 I walk you through the most important parts of a GAN implementation trying to give some indication. Lastly in section 5 I evaluate different GAN- but mainly cGAN-models based on the measures 'GAN-test' and 'GAN-train'.
In this paper I am not going to reach state-of-the-art performance but will give an insight into working with GANs as sort of proof of the concept and hopefully motivate the reader to implement their own GAN-project.

2 What is a (c)GAN?

2.1 GAN

A GAN consists of the two networks generator G and discriminator D. In one training step (with batch-size=1) a random input is fed into the generator and processed by the layers of the generator to produce an output which is called fake data, see Figure 1. Then this fake data, which in my case is one image, is fed into the discriminator and is processed by the layers to produce an output between 0 and 1. 0 stands for fake and 1 stands for real, hence the output can be interpreted as the certainty of the discriminator that the data is fake (closer to zero) or real (closer to 1).
Additionally for each fake data one real data piece from the data set the generator has to imitate is fed directly into the discriminator and gives another output between 0 and 1. With these two numbers the standard backpropagation-algorithm can be used to determine the weight updates for the generator and the discriminator by

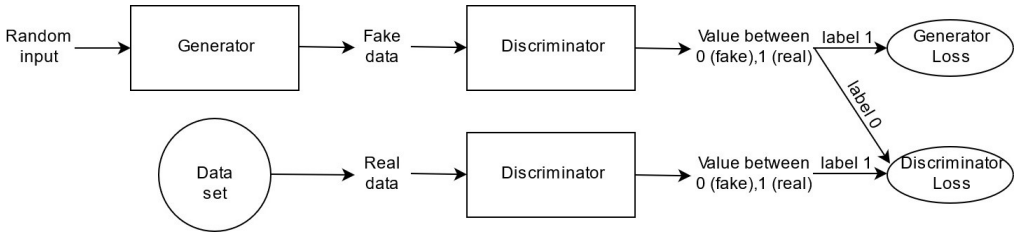

Figure 1: Structure of a GAN.

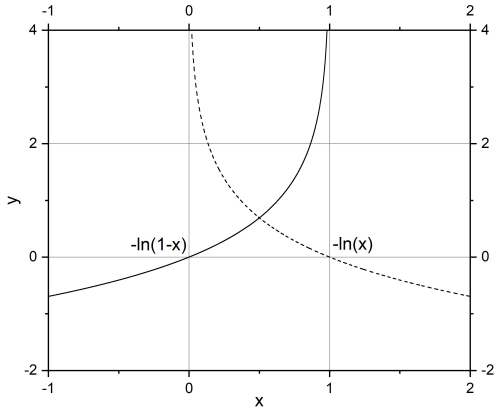

Figure 2: Functionparts of the generator and discriminator loss. Relevant is only the interval $[0, 1]$, because the output of the discriminator is always in this interval.

calculating the respective loss functions. The loss for the generator (GLoss) and for the discriminator (DLoss) are defined by

$$\begin{aligned} \text{GLoss}(z) &= -\log(D(G(z)) \\ \text{DLoss}(x, z) &= -\log(D(x)) - \log(1 - D(G(z))), \end{aligned}$$

where x is some real data and z is some random input, cf. [13].

The motivation of these loss functions is simple: As the generator G has to fool the discriminator D into classifying the fake data as real its goal is $D(G(z)) = 1$ which is equivalent to minimizing $\text{GLoss}(z)$, see Figure 2.
The loss for the discriminator consists of two parts because he has to perform on two different data. Minimizing the first part is equivalent to the goal of getting $D(x) = 1$ which corresponds to classifying the real data x as real, while minimizing the second part is equivalent to the goal $1 - D(G(z)) = 1$ respectively $D(G(z)) = 0$, see Figure 2, which corresponds to classifying the fake data $G(z)$ as fake.

It can be seen relatively easy, that the loss functions can be written with the Binary-Cross-Entropy-Loss

$$\text{BCE}(y, d) = -d \log(y) - (1 - d) \log(1 - y),$$

where y is the output of a network and d its target label [19]. As the discriminator is a classifier for real and fake data in our case d can be 0 or 1. For GLoss we have fake data but the goal of the generator is to get the fake data classified as real hence the target label is 1. This leads to

$$\text{GLoss}(z) = -\log(D(G(z)) = -1 \cdot \log(D(G(z)) - 0 = \text{BCE}(D(G(z)), 1).$$

For DLoss this works analogously but there are now two parts. At first we have real data which the discriminator wants to classify as real and secondly fake data which the discriminator wants to classify as fake. This leads to

$$\text{DLoss}(x, z) = -\log(D(x)) - \log(1 - D(G(z))) = \text{BCE}(D(x), 1) + \text{BCE}(D(G(z)), 0).$$

2.2 cGAN

A cGAN is a simple modification of a GAN and was introduced shortly after the GAN by Mirza and Osindero [27]. It just uses labels too, hence the generator in the end does not only generate fake representatives of the data set but fake representatives of specific classes.

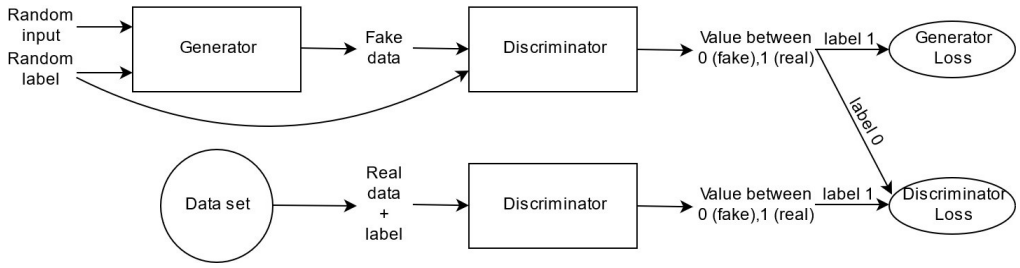

Figure 3: Structure of a cGAN.

For this purpose additionally to the random input a random (or fixed) label is fed into the generator to produce fake data related to this label, see Figure 3. Then both the fake data and the label are given to the discriminator to decide if it is real or fake. Obviously now besides real data the label from the data set belonging to this real data is given to the discriminator.

Practically the loss functions are the same as for the GAN, but theoretically written they now depend on the labels y_z of the random input and y_x of the real data too:

$$\begin{aligned} \text{GLoss}(z, y_z) &= -\log(D(G(z, y_z), y_z) \\ \text{DLoss}(x, z, y_x, y_z) &= -\log(D(x, y_x)) - \log(1 - D(G(z, y_z), y_z)), \end{aligned}$$

These loss functions can be written in form of the Binary-Cross-Entropy in the same way as for the GAN.

3 Setting of the Experiments

3.1 The Real Data

Most of the experiments in this report are done with MNIST [25], which consists of pictures of handwritten digits, hence 10 classes, and is the most standard data set for machine learning often used as a benchmark and is easy to handle [34]. As I do this project without a GPU it is important to have comparably small data which MNIST perfectly fits.
It contains 70.000 (splitted in 60.000 for training and 10.000 for testing) greyscale pictures with 28x28 pixels. Since a main problem of MNIST is that it is very easy for todays neural networks I use FashionMNIST too which was built to lessen this issue [34], [30]. It has the exact same structure as MNIST, but contains pictures of shoes, T-shirts etc., which makes it perfect to use with MNIST because one does not have to change any parameters but has significantly more complex pictures [34].

3.2 The Evaluation

Evaluating a GAN is one problem of this network structure and is discussed in many papers, i.e. [32], [7], [6], [26], [24]. There are many different measures that were proposed from which the most popular ones may be the Inception score (IS) and the Frechet-Inception-Distance (FID) with their variants, but nevertheless of the popularity they get criticized a lot for not measuring different aspects of a GAN like diversity and quality of the data well [7], [32], [26], [6], [24]. Additionally the computation is comparably expensive which does not fit my equipment.

The evaluation strategy proposed by Shmelkov et al. [32] gives a good insight in diversity and quality of the generated data which is why I choose this measure. They recommend to determine the two numbers GAN-test and GAN-train with an external classifier.
GAN-train is the accuracy of the external classifier trained on the fake data produced by the generator and tested on real data. If the accuracy is significantly lower than the accuracy of the classifier on real data trained on real data that means that the fake data can not train the classifier well enough to have a good performance on real data hence the fake data does not represent the real data well. This can be due to lack of diversity in the produced fake data or the fake data not being similar enough to real data, cf. [32].
GAN-test is the accuracy of a classifier trained on the real data and tested on the fake data produced by the generator. This accuracy can measure how close the fake data is to the real data regarding the quality. If the accuracy of the classifier is significantly lower than on the real data the fake data lacks in quality because to the classifier they have few similarity. If the accuracy is significantly higher the generator did only 'copy' some of the real data hence overfitted, cf. [32].
Therefore regarding both measures the GAN is better the closer GAN-train and GAN-test are to the accuracy of the classifier trained and tested on real data. As a proof of concept Shmelkov et al. [32] showed on the MNIST data set that it is possible to train a GAN such that it scores what you can name perfect in both GAN-train and GAN-test. A model like this would hence be perfect for data augmentation because it can generate data that is indistinguishable from the real data and not identical to the real data hence can actually enlarge the data set.

One disadvantage of this evaluation procedure is that one needs to have labels for the generated data as the accuracy of the external classifier is decisive. Hence it is literally not possible to evaluate a GAN with this measure but at least a cGAN is needed.
In section 5 I evaluate my cGAN models with GAN-test and GAN-train, for which I implemented an additional classifier. Obviously the comparison to other reports and papers is not possible unless they used the same external classifier. As I do not want to implement a state-of-the-art model and compare to others' performances but only compare my different models this is not a problem. Of course in general this is another disadvantage of this evaluation procedure.

3.3 Problems and Possible Solutions

In this subsection I will shortly explain some more or less typical problems when working with GANs and roughly present my solution approaches in this very special case.
For the cGAN evaluation in section 5 I first implemented a fully-connected classifier similar to the discriminator in the cGAN reaching about 97% accuracy on MNIST. When determining GAN-test and GAN-train it turns out classifying nearly everything as one number (mostly 5, 4 or 3) resulting in GAN-test and GAN-train accuracies of 9% to 20%. As this is not consistent with human eye perception, it seems that the classifier does not recognize the shape of the digits but only uses a few pixel values for classification which the cGAN accidentally sets similar for all classes.
This possibly demonstrates a problem both of fully-connected neural networks for image classification and MNIST. MNIST seems to be classifiable by a few pixel values [12] and especially by fully-connected neural networks, which only use the pixel

values and not the structure of the picture like convolutional neural networks. It may be possible to adapt and tune the fully-connected classifier to generalize better and solve this problem another way but implementing a convolutional external classifier for this purpose solved the problem for me.

A typical problem of GANs is called mode collaps. It describes the szenario that the generator has very few diversity in its outputs and hence produces only a few modes of the real data. For the MNIST case this would be if e.g. every 5 the generator produces mainly looks the same. In the worst case it outputs just one picture per class every time. The reasons for this problem are not very well understood [13] which results in that there is no unique solution formula when this problem shows up.
Mode collaps occurred to me when adding labels to my GAN structure hence when implementing my first cGAN. This could possibly be expected because I was adding new information (the label) to both generator and discriminator. Somehow it was too much weight on this new information during training such that the generator ended up mainly just taking the label and outputting nearly the same picture for each class every time.
I solved this problem by using an idea by Isola et al. [18] where they used dropout and batch normalization in the generator even when not training it, see section 5 for more details. Other possible solutions could be to get more randomness into the fake data generation or making the discriminators job harder by introducing random noise into the layers of the generator [16] or maybe having more random input nodes.

Another problem I stumbled on is a PyTorch problem. There are mainly two possibilities to use dropout in a network: torch.nn.Dropout or torch.nn.functional.dropout. The by PyTorch predefined evaluation mode eval() is designed to turn off layers like dropout and batch normalization during evaluation mode. This works fine with torch.nn.Dropout but does NOT work with torch.nn.functional.dropout which is not mentioned in the PyTorch documentation, cf. [4]. If one wants to use torch.nn.functional.dropout one has to manually set the parameter training=False when one wants to deactivate dropout.

3.4 The Base Model

Oriented at some models in the literature I choose some hyperparameters for the GAN and the cGAN I never change during my experiments, see Table 1. Also the basic architecture of the GAN and the cGAN stays roughly the same during experiments and is explained in section 4, see Figure 4.
Following [28], [9], [18], [29], [33], [13] I use the Adam optimizer [20] with $\beta_1 = 0.5$ [28], [9], [29], [33] and a learning rate of 0.0002 [28], [29], [33] and close to [9]. The parameter β_1 influences the exponential moving average of the gradient of the objective function [20]. Furthermore the batch-size is set to 100 [28], the scale for the leaky ReLU activation function is defined as 0.2 [28], [18], [29] and the dropout percentage is fixed to 0.5 [28], [27], [18]. The scale for leaky ReLU sets the slope for the line for negative values of the normal ReLU activation function, hence for a scale of 0 leaky ReLU and ReLU are the same function.

optimizer	β_1	learning rate	batch size	scale	dropout
Adam	0.5	0.0002	100	0.2	0.5

Table 1: Fixed hyperparameters for every experiment.

4 Implementation

Only main points of the implementation of the GAN are explained in this section. The complete implementation of the GAN as well as the cGAN and all other necessary files and already trained models are attached to this report. All the implementations are very basic and not designed to be as efficient as possible.

At first the hyperparameters of the GAN have to be set. Doing this right at the top makes it easier to change them when you want to tune your parameters.
Then I load the data for which I used the datasets/dataloader of PyTorch. A good explanation can be found in the tutorials [2]. There are some popular data sets inbuilt like MNIST and FashionMNIST. The DataLoader just groups as many tensors from the data as the batch size to one tensor. Another option is for example that it shuffles the data each epoch which is not chosen here.

```
1  training_data = datasets.MNIST(
2      root="data",
3      train=True,
4      download=True,
5      transform=ToTensor()
6  )
7
8  test_data = datasets.MNIST(
9      root="data",
10     train=False,
11     download=True,
12     transform=ToTensor()
13 )
14
15 train_dataloader = DataLoader(training_data, batch_size=
       batch_size)
16 test_dataloader = DataLoader(test_data, batch_size=
       batch_size)
```

The weight initialization is done automatically by PyTorch too, but He et al. [15] proposed the now called Kaiming-Initialization for neural networks with mostly ReLU-activation to prevent it from vanishing or exploding gradients. As I want to use the leaky ReLU activation function I use this weight initialization.

```
17 def init_weights(layer):
18     if type(layer) == nn.Linear:
19         torch.nn.init.kaiming_normal_(layer.weight)
```

Now I define the generator and the discriminator as classes, which are both fully-connected neural networks. The very similar cGAN-architecture can be found in Figure 4. The generator has 100 input nodes, which are mapped to 512 nodes with leaky ReLU activation function. A second layer with 512 nodes and leaky ReLU activation function is then followed by the output-layer with 784 nodes and tanh activation function. Alternatively one can use sigmoid activation function in the output layer too, but experience of many implementations seems to propose tanh

for better results [28], [9], [18], [29]. Additionally I add batch normalization and dropout, which is suggested for better results [28], [9], [27], [18], [29], [13], [17].

```
20 class Gen(nn.Module):
21     def __init__(self):
22         super(Gen, self).__init__()
23         self.architecture = nn.Sequential(
24             nn.Linear(100, 512),
25             nn.BatchNorm1d(512),
26             nn.LeakyReLU(scale),
27             nn.Dropout(p=pDropout),
28             nn.Linear(512, 512),
29             nn.BatchNorm1d(512),
30             nn.LeakyReLU(scale),
31             nn.Dropout(p=pDropout),
32             nn.Linear(512, 784),
33             nn.BatchNorm1d(784),
34             nn.Tanh()
35         )
36
37     def forward(self, input):
38         output = self.architecture(input)
39         return output
```

Now as I defined the class Gen for the generator I have to define one instance of this class to which I apply the predefined weight initialization. If one wants to use the automatic initialization from PyTorch one can omit the second line.

```
40 G = Gen()
41 G.apply(init_weights)
```

The discriminator has 784 nodes as inputs which are mapped to 512 nodes with, like in the generator, leaky ReLU activation. After a second 512 node leaky ReLU layer there is only one output node with sigmoid activation. I use dropout in the discriminator as well but no batch normalization. As for the generator I create one instance of this class and apply the Kaiming initialization.

```
42 class Dis(nn.Module):
43     def __init__(self):
44         super(Dis, self).__init__()
45         self.flatten = nn.Flatten()
46         self.architecture = nn.Sequential(
47             nn.Linear(784, 512),
48             nn.LeakyReLU(scale),
49             nn.Dropout(p=pDropout),
50             nn.Linear(512, 512),
51             nn.LeakyReLU(scale),
52             nn.Dropout(p=pDropout),
53             nn.Linear(512, 1),
54             nn.Sigmoid()
55         )
56
57     def forward(self, input):
58         input = self.flatten(input)
59         output = self.architecture(input)
60         return output
61
62 D = Dis()
63 D.apply(init_weights)
```

The reason why there are 784 nodes is that I will train the GAN on MNIST- and FashionMNIST-dataset which contains pictures with 28x28 pixels, hence 784 numbers. The numbers 100 for the input size of the generator and 512 for the size of the hidden layers can be changed/tuned if wanted but were used about this size in other implementations [27], [9], [29]. As the discriminator has to distinguish between real and fake images there is only one output here. The sigmoid activation function takes care of the scaling between 0 and 1.

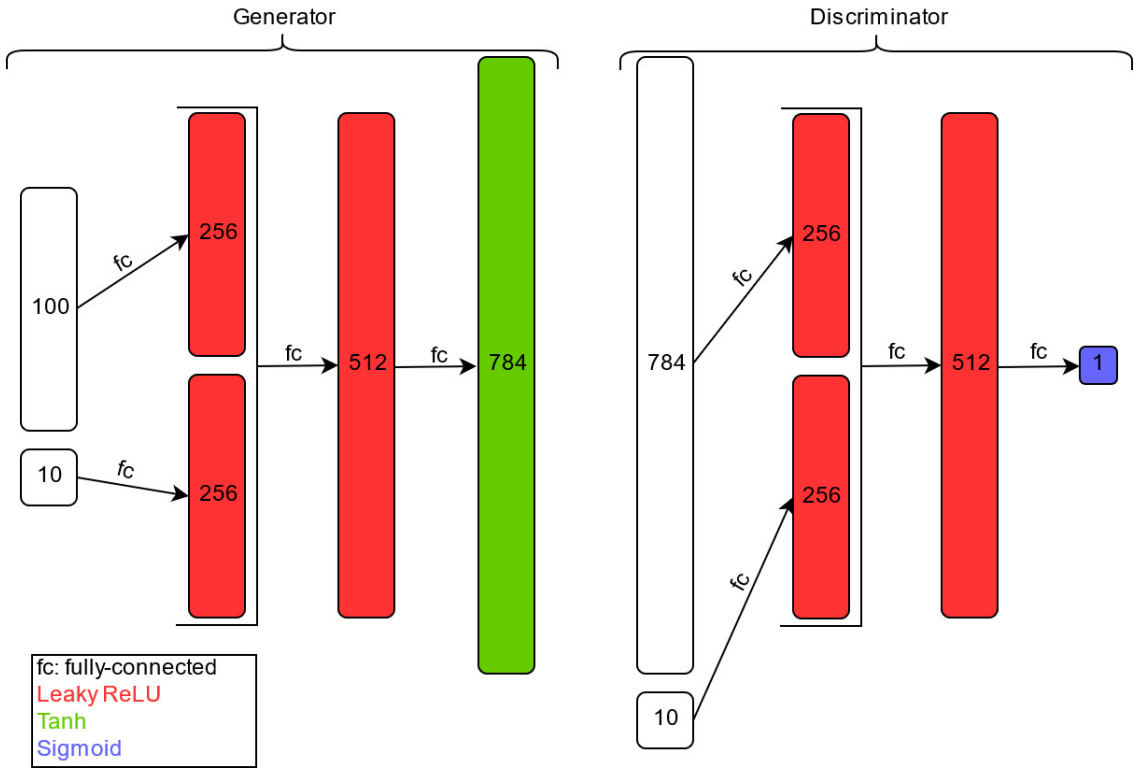

Figure 4: Basic architecture of my cGAN. I use one hot encoding for the labels which explains the input dimension. The GAN-architecture is the same except that there is no label input for G and D such that the first hidden layer is normally fully-connected with 512 nodes. Only some details of this architecture get changed during the experiments.

Next I choose the optimizer for the training. As previously discussed I use the Adam-optimizer for both generator and discriminator with $\beta_1 = 0.5$; $\beta_2 = 0.999$ is the default setting.

```
64 G_optimizer = torch.optim.Adam(G.parameters(), lr=
       learning_rate, betas=(0.5, 0.999))
65 D_optimizer = torch.optim.Adam(D.parameters(), lr=
       learning_rate, betas=(0.5, 0.999))
```

The loss is simply implemented in form of the BCE-Loss as discussed previously. Additionally a trick for training a GAN comes in at this point which is called one-sided label smoothing or one-sided softlabel [13], [31]. In case of the discriminator the labels of the real data get lowered hence instead of 1 the label of a real image is now for example 0.85. In this implementation it is a random number between 1 and 0.85. One-sided label smoothing has the effect that the discriminator learns to be not so sure about classifying real data as real [13]. In section 5 I show in my special case that this could indeed have a positive effect on GAN performance.

```
66 BCELoss = nn.BCELoss()
67
68 def GenLoss(fake_perc):
69     real_label = torch.ones(batch_size, 1, dtype=torch.float
       )
70     loss = BCELoss(fake_perc, real_label)
71     return loss
72
73 def DisLoss(real_perc, fake_perc):
74     real_label = torch.ones(batch_size, 1, dtype=torch.float
       ) - softlabel * torch.rand(batch_size, 1, dtype=torch
       .float)
75     fake_label = torch.zeros(batch_size, 1, dtype=torch.
       float)
76     loss = BCELoss(real_perc, real_label) + BCELoss(
       fake_perc, fake_label)
77     return loss
```

Now I implement the training loop for the GAN in which I iterate through the whole given data set respectively the real data. Because I use the tanh activation function in the generator and this function has the range $]-1,1[$ I have to rescale the real data to this range too. As the pixels of the pictures in MNIST and FashionMNIST are scaled between 0 and 1 the given code satisfies this goal.

```
78 def TrainingLoop(dataloader, D, G, epoch):
79     for real_data, _ in dataloader:
80         real_data = 2*real_data - 1
```

A general thing which should not be forgotten is the command *zero_grad()* which prevents PyTorch to average over the history of the gradients which is default [3].

```
81         D.zero_grad()
```

So at first I evaluate the discriminator on real data. Then I let the generator produce fake data out of random input and evaluate the discriminator on this fake data. Out of these I compute the loss of the discriminator with which I do a backward pass and the optimization step.

```
82         D_real = D(real_data)
83
84         x_rand = torch.randn(batch_size, 100, dtype=torch.
               float)
85         fake_data = G(x_rand)
86         D_fake = D(fake_data)
87         D_loss = DisLoss(D_real, D_fake)
88
89         D_loss.backward()
90         D_optimizer.step()
```

For the generator do not forget *zero_grad()* and then do analogous steps. Take random input and let the generator produce fake data, evaluate the discriminator on the fake data, compute the loss for the generator, do the backward pass and optimize.

```
91         G.zero_grad()
92         x_rand = torch.randn(batch_size, 100, dtype=torch.
               float)
93         fake_data = G(x_rand)
```

```
94            D_fake = D(fake_data)
95            G_loss = GenLoss(D_fake)
96
97            G_loss.backward()
98            G_optimizer.step()
```

For actually training the GAN I now just have to call the implemented functions in each epoch. PyTorch has the predefined functions train() and eval() which set the networks into train and evaluation mode. This has the effect that in evaluation mode layers like dropout and batch normalization get deactivated because one normally does not want them to be active when testing the already trained network. As during generator training the discriminator is not trained one normally puts the discriminator into evaluation mode and vice versa. In the next section I show that it could be advantageous to change this, e.g. like here, based on [18].

```
99  for epoch in range(1, max_epochs + 1):
100     G.train()
101     D.train()
102     TrainingLoop(train_dataloader, D, G, epoch)
103     G.eval()
104     D.eval()
105     TestLoop(test_dataloader, D, G, epoch)
```

5 Results

I determine the GAN-test accuracy as the average GAN-test accuracy of 10 runs with the sample standard deviation. For the GAN-train accuracy the external convolutional classifier is trained only once which gives only one accuracy on the real data test set. It scores a 99.11% test-accuracy on MNIST. If not explicitly mentioned otherwise the experiments were done with the MNIST data set.

5.1 GAN

Because the evaluation method with GAN-test and GAN-train requires labels there were not done many experiments with the GAN-architecture. In Table 2 an overview about some experiments with their settings is given while the Figures 5, 6, 7 and 8 show the outputs of these GAN-models.
The output of the model 1 is not shown here because this model does not learn to output more than black background with some noise on it. The reason possibly is that after about one third of the training process the discriminator manages to classify nearly every fake and real data right giving the generator nothing to learn from.
Putting the generator in evaluation mode for image generation has a positive effect on the smoothness of the generated numbers but also makes the images a bit blurry compared to Figure 5. Comparing Figure 6 and 7 training the discriminator twice seems to improve quality of the images while there seems to be no big difference between random input from a normal or uniform distribution, cf. Figures 7, 8.
Overall it seems that the GAN develops a favor for certain numbers. For example in the Figures 6 and 7 especially the numbers 0 and 3 are overrepresented while in Figure 8 this is the case for 0, 1 and 9. The sample size is of course too small but it is definitely noticable that maybe more difficult numbers like a 4 appear hardly ever.

#	prob. distr.	train D	D-training	G-training	image generation
1	normal	once	G.eval()	D.eval()	G.eval()
2	normal	once	G.train()	D.eval()	G.train()
3	normal	once	G.train()	D.train()	G.eval()
4	normal	twice	G.train()	D.train()	G.eval()
5	uniform	twice	G.train()	D.train()	G.eval()

Table 2: Settings of the GAN-experiments. Each GAN was trained with the hyperparameters epochs: 100, soft label: 0.15 + random.

Figure 5: Outputs of model 2.

Figure 6: Outputs of model 3.

Figure 7: Outputs of model 4.

Figure 8: Outputs of model 5.

5.2 cGAN

No evaluation mode: A very interesting idea for training GANs was used by Isola et al. in [18] when they never put the generator in evaluation mode hence used dropout and batch normalization at all times. I already used this method earlier for training the GAN-model 2 but adapting this method by, inter alia, putting the generator in evaluation mode for generating images after the training lead to better results for the GAN.
From Table 3 it is very clear that using this technique for the cGAN does not work and is leading to mode collaps, cf. GAN-test and GAN-train of model 6. Additionally putting the generator in evaluation mode during discriminator training leads to the same result as for the GAN. The reason is possibly the same too thus the discriminator being too good, see Figures 9 and 10, which does not allow the generator to learn properly.
It is of course possible to solve this issue with other techniques but never putting the generator in evaluation-mode (picking up the idea from Isola et al. [18] again) works kind of well for the cGAN and in my opinion has a positive effect on the

#	epochs	label	prob. distr.	soft label	eval()?	train D	GAN-test	GAN-train
6	100	64	normal	0.15 + random	GAN	once	99.55 ± 0.08%	55.29%
7	100	64	normal	0.15 + random	Yes	once	0.00 ± 0.00%	–
8	100	64	normal	0.15 + random	No	once	86.83 ± 0.34%	91.70%
9	100	64	normal	0.15 + random	No	twice	89.40 ± 0.29%	94.49%
10	100	256	normal	0.15 + random	No	once	90.93 ± 0.30%	92.94%
11	100	256	normal	0.15 + random	No	twice	92.54 ± 0.30%	87.00%

Table 3: Experiments on never using eval(), the number of neurons in the first hidden layer connected to the label and how often the discriminator is trained.
#: Model number; label: Number of nodes the input label gets mapped to in both G and D; prob. distr.: Probability distribution the random input for G gets sampled from; eval()?: If the generator gets set in evaluation mode during inference (GAN means using the technique from training the GAN); train D: How often the discriminator gets trained per generator training.

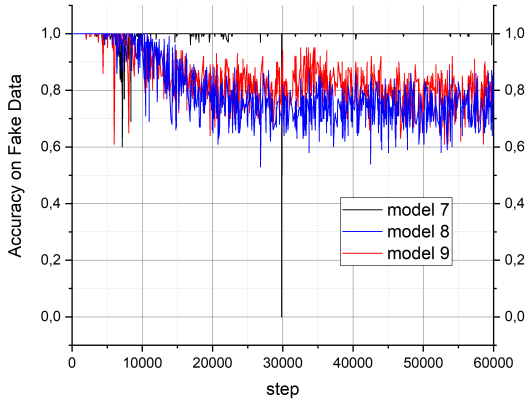

Figure 9: Discriminator performance of models 7, 8 and 9 during training on fake data.

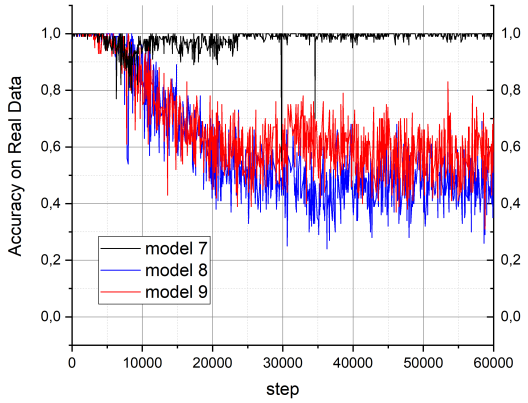

Figure 10: Discriminator performance of models 7, 8 and 9 during training on real data.

diversity of the images due to active dropout during inference.
Interestingly the model outputs higher quality numbers when dropout is reduced afterwards, see Figure 11, even though it was trained completely with active dropout. But on the other side this then leads to mode collaps, see Figure 12, where the diversity of the images is very small when dropout is completely deactivated.

Training D more often, cGAN architecture: Secondly it can be stated that when training the discriminator only once having the label mapped to 256 nodes instead of 64 leads to higher GAN-test and GAN-train accuracies indicating higher quality and more diversity in the generated images, see Table 3. Interestingly the effect of training the discriminator in the models twice instead of once is different. While the 64-model improves in both GAN-test and GAN-train, indicating an overall better GAN-performance, the 256-model improves only in GAN-test while rather drastically worsen in GAN-train. The improvement in GAN-test indicates that the image quality increases, because the external Classifier performs better than before, but the lower GAN-train accuracy indicates that the image diversity is poorer, because the external classifier can not learn to classify MNIST-digits that well based on this generated data. Figures 13 and 14 illustrate that there is not much visual difference compared to the relatively large difference in GAN-test and GAN-train. As the only difference between these models is the number of nodes the label gets mapped to it seems that in the 256 case there is more focus on the label resulting

Figure 11: Outputs of model 9. Starting from standard dropout percentage of 0.5 at the top row, the dropout percentage is reduced 0.05 each row till 0.05 at the bottom row.

Figure 12: Outputs of model 9 with deactivated dropout (dropout percentage of 0).

Figure 13: Outputs of model 10.

Figure 14: Outputs of model 11.

in a decrease in image diversity when the discriminator is trained twice and hence has better performance during training.

Uniform vs. normal distribution: Sampling the random input from a uniform distribution instead of a normal distribution increases the GAN-test accuracy slightly, see Table 4. The GAN-train accuracy instead lowers quite considerably. Judging overall GAN-performance it seems that the normal distribution is to be preferred in this special case but this of course can depend on the intended goal for which the cGAN is used.

#	epochs	label	prob. distr.	soft label	eval()?	train D	GAN-test	GAN-train
9	100	64	normal	0.15 + random	No	twice	89.40 ± 0.29%	94.49%
12	100	64	uniform	0.15 + random	No	twice	90.37 ± 0.22%	91.77%

Table 4: Using random input from a normal vs. a uniform probability distribution.

Whereas Tran et al. [33] state that the distribution the prior noise gets sampled from "is often Uniform or Gaussian distribution" in my sources only [27], [29] and [33] explicitly said that they used a uniform distribution. To my knowledge none of my sources explicitly wrote that they were using a normal distribution and I found nothing about how they reached this decision.

One-sided label smoothing: It seems that the one-sided label smoothing has no or a very small positive effect on GAN-test as the accuracies increase steadily but

very little the more the label gets lowered respectively the variable soft label gets increased, see Table 5. Choosing the label as a random number between the original label 1 and the smoothed label (here 0.85) seems to have no effect on GAN-test and GAN-train in this setting. The changes in GAN-train depending on the value of soft label are relatively big compared to GAN-test, where the highest GAN-train accuracy was scored with $softlabel = 0.15$.

#	epochs	label	prob. distr.	soft label	eval()?	train D	GAN-test	GAN-train
9	100	64	normal	0.15 + random	No	twice	89.40 ± 0.29%	94.49%
13	100	64	normal	–	No	twice	89.31 ± 0.38%	92.81%
14	100	64	normal	0.1	No	twice	89.69 ± 0.29%	93.76%
15	100	64	normal	0.15	No	twice	89.89 ± 0.26%	94.50%
16	100	64	normal	0.2	No	twice	90.04 ± 0.43%	92.83%

Table 5: Experiments on using one-sided label smoothing.

One effect of the label-smoothing is directly seen when looking at the discriminator performance during training, see Figures 15 and 16. Because of the huge variance and the basically same development of the discriminator performance during training only two exemplary runs are shown. The difference in performance can, because of the same development, well judged by looking at the average, see Table 6.
The more the original label gets lowered the better is the discriminator performance on fake data and the poorer is the discriminator performance on real data. Adding randomness seems to lower the performance on real data and increase the performance on fake data, cf. models 9 and 15, whereas this changed nothing with respect to GAN-test and GAN-train.
It is noticeable that the standard deviation of model 13 and especially 16 are big compared to the other standard deviations. This means the accuracy of the external convolutional classifier is relatively strongly fluctuating and hence the cGAN is not as stable in generating good pictures as in the other models. Therefore it seems that especially too much label smoothing but also no label smoothing destabilizes the cGAN.

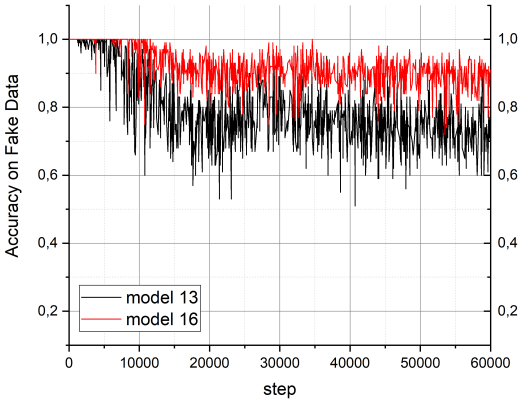

Figure 15: Discriminator performance of models 13 and 16 during training on fake data.

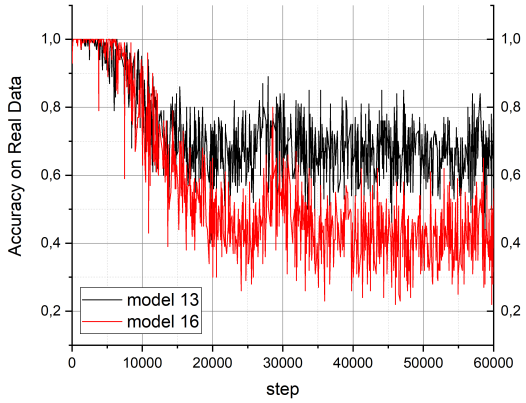

Figure 16: Discriminator performance of models 13 and 16 during training on real data.

Tanh vs. sigmoid: Most papers in my sources use tanh activation function in the output layer, e.g. [9], [28], [29] and [18], but some use the sigmoid function,

data	9 (0.15r)	13 (–)	14 (0.1)	15 (0.15)	16 (0.2)
real	84.29%	79.43%	85.66%	88.87%	92.02%
fake	67.00%	72.73%	63.94%	60.14%	55.39%

Table 6: Average discriminator performance of different model settings on fake and real data during training. Next to the model number the value of the variable soft label is written.

e.g. [33], including Goodfellow et al. in [14] and Mirza, Osindero in [27], where they originally introduced the GAN respectively the cGAN architecture. Especially newer publications seem to use tanh more often which maybe is a good hint that tanh works better in practice.
In my experiment tanh performs way better compared to sigmoid based on GAN-train and GAN-test, see Table 7. But these numbers are a bit misleading. Looking at the generated pictures and some example classifications of the external convolutional classifier reveals that the main problem is the number seven and partly the eight. Somehow the generator produces only pictures with a white background and some black pixels for the label seven and about half of the pictures look like this for the label eight. I can not explain why there is this problem with only these two numbers.

The difference between the sigmoid and the tanh function is that the tanh function is closer to a step function than sigmoid, see Figure 17, and hence for numbers close to zero the outputs are further apart than for the sigmoid function. The effect that the cGAN has problems with the numbers seven and eight remains unchanged if the dropout percentage gets reduced for generating images afterwards. For the other numbers the same effect as in Figure 11 for tanh is observed.

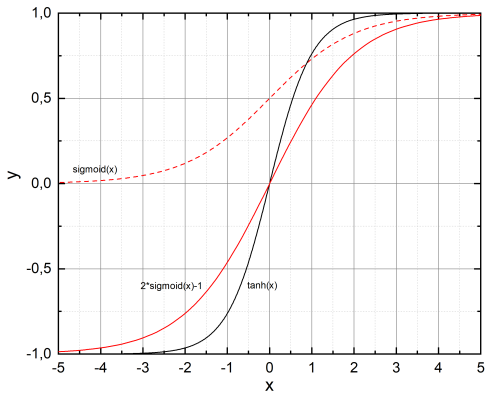

Figure 17: For better comparison the sigmoid function is rescaled.

#	activ.	label	prob. distr.	soft label	eval()?	train D	GAN-test	GAN-train
9	tanh	64	normal	0.15 + random	No	twice	89.40 ± 0.29%	94.49%
17	sigmoid	64	normal	0.15 + random	No	twice	75.50 ± 0.63%	82.32%

Table 7: Having tanh activation function vs. sigmoid activation function in the output layer of the generator. The epochs are set to 100 as in the last experiments. activ.: Activation function of the last generator layer.

Number of epochs: Increasing the number of training epochs from 100 to 200 leads to an relatively clear improvement in GAN-test accuracy, see Table 8, while GAN-train decreases only slightly which could also be due to stochastic reasons. This can be interpreted as that the image quality increases while the image diversity roughly stays the same. One has to mention that the difference in image quality is not really visible by eye, see Figures 18 and 19. One possible explanation could be that not really the quality of the pictures increases but that there are fewer really

#	epochs	label	prob. distr.	soft label	eval()?	train D	GAN-test	GAN-train
9	100	64	normal	0.15 + random	No	twice	89.40 ± 0.29%	94.49%
18	200	64	normal	0.15 + random	No	twice	93.04 ± 0.20%	93.99%

Table 8: Effect of number of training epochs on the GAN performance.

bad pictures which the classifier has nearly no chance of classifying right. This seems to be reasonable when improving the number of training epochs. This is also consistent with the size of the standard deviation as it is much smaller than for model 9 and also the smallest of all working cGAN-models depicted in this section. The quality of the images can possibly be further improved by training the cGAN even longer for which my technical equipment is not adequate. For example in [9] the models for MNIST are trained for 10.000 epochs. This number seems very high to me especially when comparing their generated images with mine.

Figure 18: Outputs of model 9.

Figure 19: Outputs of model 18.

FashionMNIST: The external convolutional classifier scores a 89.96% test-accuracy on FashionMNIST. This and the results in Table 9 compared to 8 clearly show that FashionMNIST is way more complex than MNIST. Interestingly the GAN-train performance of the models is still very good which seems to highlight the good diversity in the images generated by a model with this parameters. On the downside the low GAN-test accuracy seems to emphasize that the models lack quality of the images compared to the diversity.
Strangely enough the cGAN does not improve when training 200 epochs. This can have many reasons so that more experiments would be necessary. Figures 20 and 21 show well why FashionMNIST is much more difficult than MNIST as the objects are very noisy whereas the original images have slight patterns or designs.

#	epochs	label	prob. distr.	soft label	eval()?	train D	GAN-test	GAN-train
19	100	64	normal	0.15 + random	No	twice	43.29 ± 0.40%	80.24%
20	200	64	normal	0.15 + random	No	twice	43.57 ± 0.57%	78.46%

Table 9: Models from Table 8 trained on FashionMNIST.

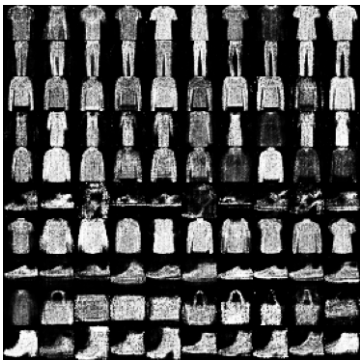

Figure 20: Outputs of model 19.

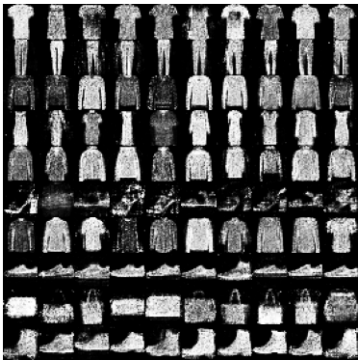

Figure 21: Outputs of model 20.

Summary

In this report the GAN framework was explained and some problems of GAN training were discussed with possible solutions. An indication for further implementations was given by explaining a PyTorch implementation used in this report and clarifying the choice of the hyperparameters, architecture and GAN-tricks based on the literature. The experiments confirmed that using dropout and batch normalization in the generator also for inference works well for a cGAN and in particular seems to support image diversity. Furthermore the tests especially showed that in this case one-sided label smoothing directly influences the discriminator performance during training and can, correctly chosen, improve overall GAN-performance based on GAN-test and GAN-train. Additionally some strange effects occurred when using sigmoid instead of tanh in the output layer of the generator which seems to verify the preferred usage of tanh in the literature.

References

[1] *ImageNet*. `https://image-net.org/`.

[2] *Pytorch Tutorials*. `https://pytorch.org/tutorials/`.

[3] *Zeroing out gradients in PyTorch*. `https://pytorch.org/tutorials/recipes/recipes/zeroing_out_gradients.html`.

[4] *Using Dropout in Pytorch: nn.Dropout vs. F.dropout*, 2018. `https://stackoverflow.com/questions/53419474/using-dropout-in-pytorch-nn-dropout-vs-f-dropout`.

[5] A. Antoniou, A. Storkey, and H. Edwards, *Data Augmentation Generative Adversarial Networks*, (2018).

[6] S. Barratt and R. Sharma, *A Note on the Inception Score*, (2018).

[7] A. Borji, *Pros and Cons of GAN Evaluation Measures: New Developments*, (2021).

[8] J. Brownlee, *18 Impressive Applications of Generative Adversarial Networks (GANs)*, June 2019. `https://machinelearningmastery.com/impressive-applications-of-generative-adversarial-networks/`.

[9] K. Cheng, R. Tahir, L. K. Eric, and M. Li, *An analysis of generative adversarial networks and variants for image synthesis on MNIST dataset*, Multimedia Tools and Applications, 79 (2020), pp. 13725–13752.

[10] H.-W. Dong, W.-Y. Hsiao, L.-C. Yang, and Y.-H. Yang, *MuseGAN: Demonstration of a convolutional GAN based model for generating multi-track piano-rolls*, (2017).

[11] A. Elgammal, B. Liu, M. Elhoseiny, and M. Mazzone, *CAN: Creative Adversarial Networks, Generating "Art" by Learning About Styles and Deviating from Style Norms*, (2017).

[12] L. Franceschini, *Distinguishing pairs of classes on MNIST and Fashion-MNIST with just one pixel*, Apr. 2021. `https://lucafrance.github.io/2021/04/05/mnist-pairwise-one-pixel.html`.

[13] I. Goodfellow, *NIPS 2016 Tutorial: Generative Adversarial Networks*, (2016).

[14] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, *Generative Adversarial Nets*, (2014).

[15] K. He, X. Zhang, S. Ren, and J. Sun, *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, (2015).

[16] F. Huszár, *Instance Noise: A trick for stabilising GAN training*, Oct. 2016. `https://www.inference.vc/instance-noise-a-trick-for-stabilising-gan-training/`.

[17] S. Ioffe and C. Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, (2015).

[18] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, *Image-to-Image Translation with Conditional Adversarial Networks*, (2016).

[19] P. Kim, *MATLAB Deep Learning*, Apress, 2017.

[20] D. P. Kingma and J. L. Ba, *Adam: A Method for Stochastic Optimization*, (2017).

[21] N. Kiryati and Y. Landau, *Dataset Growth in Medical Image Analysis Research*, Journal of Imaging, 7 (2021), p. 155.

[22] A. Krizhevsky, *Learning Multiple Layers of Features from Tiny Images*, (2009).

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *ImageNet classification with deep convolutional neural networks*, Communications of the ACM, 60 (2017), pp. 84–90.

[24] T. Kynkäänniemi, T. Karras, S. Laine, J. Lehtinen, and T. Aila, *Improved Precision and Recall Metric for Assessing Generative Models*, (2019).

[25] Y. LeCun, C. Cortes, and C. J. C. Burges, *THE MNIST DATABASE*. `http://yann.lecun.com/exdb/mnist/`.

[26] M. Lucic, K. Kurach, M. Michalski, S. Gelly, and O. Bousquet, *Are GANs Created Equal? A Large-Scale Study*, (2018).

[27] M. Mirza and S. Osindero, *Conditional Generative Adversarial Nets*, (2014).

[28] A. Odena, C. Olah, and J. Shlens, *Conditional Image Synthesis With Auxiliary Classifier GANs*, Proceedings of the 34th International Conference on Machine Learning, (2017).

[29] A. Radford, L. Metz, and S. Chintala, *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*, (2016).

[30] K. Rasul, H. Xiao, and R. Vollgraf, *FashionMNIST*. `https://research.zalando.com/project/fashion_mnist/fashion_mnist/`.

[31] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, *Improved Techniques for Training GANs*, (2016).

[32] K. Shmelkov, C. Schmid, and K. Alahari, *How good is my GAN?*, (2018).

[33] N.-T. Tran, V.-H. Tran, N.-B. Nguyen, T.-K. Nguyen, and N.-M. Cheung, *On Data Augmentation for GAN Training*, (2020).

[34] H. Xiao, K. Rasul, and R. Vollgraf, *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*, (2017).