Implementing Artificial Neural Networks with Tensorflow 2020/21

Documentation of our Final Project

Linus kleine Kruthaup, Mara Rehmer

Contents

Choosing a topic

The first challenge of the project was of course deciding on a topic. The numerous applications of the different artificial neural network (ANN) architectures can be quite overwhelming. Our first idea was to use the state of the art transformer architecture to do a NLP task. Either classifying a reddit article to its subreddits, or using Eminem lyrics to create a new song. But as we looked a little closer into the ideas, we discovered that the runtime of the model increases quadratically with respect to the sequence length [1]. For that reason we looked around and took a different path. We were intrigued by the amazing results in generating fake faces produced by Karras et al. (2019) with their Style Generative Adversarial Network (Style-GAN) architecture [2].
Sadly, one of our group members had to leave our group, which is why we finally decided to reimplement a version of the Progressive Growing GAN described by Karras et al. (2018) [3], which is the base model for the StyleGAN architecture and tackles some impairments of previous GAN architectures.

Progressive Growing GAN (ProGAN)

General GAN Architecture

In the course we learned that there are ANN architectures that can generate data, so called Generative Adversarial Networks (GANs). They consist of two connected sub-architectures: a generator and a discriminator. The generator tries to create convincing samples similar to real data from a latent input, whereas the discriminator has to decide whether a given sample is real or fake/generated. Both architectures are basically trained via the same loss function. To be more specific, the discriminator uses binary cross entropy and aims to maximize the probability of the right classification - true or fake sample. The generator uses the negative loss of the discriminator, but only using the term for the generated samples. So in the best case, the generator captures the underlying distribution of the real data while training and can therefore create samples which are indistinguishable from real samples for the discriminator (and sometimes also for humans).

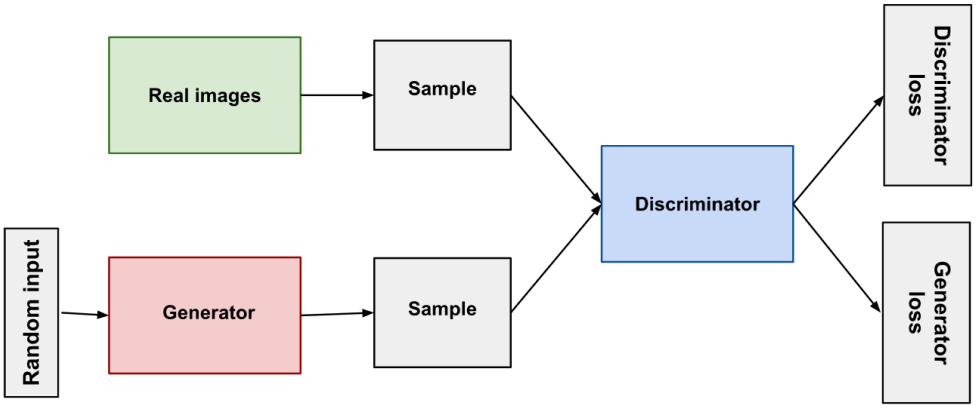

Figure 1 [4]: *General architecture of a GAN.*

Problems with GANs

Nowadays, we are used to high quality pictures, so of course we want a generator to create images with a large resolution. The problem is, that the larger the generated images, the easier it is for the discriminator to tell them apart from real samples. Generating high quality images also requires more memory to train the GAN, which in turn also means that we have to reduce the batch size, causing the model to be unstable [6]. Since both the generator and the discriminator
change the same loss in opposite directions, it can be hard to achieve the so-called “Nash Equilibrium” - a state where both, the discriminator and the generator, are optimal with respect to each other. This can also cause the training to be unstable.
Another problem is mode collapse: if the generator does not manage to capture enough variation from the training data, it can happen that the generator creates a convincing sample and from there on only produces this or very similar samples independent of its input.
If the discriminator is too strong, ergo it always classifies the real and fake samples as such, it does not give the generator enough information to become better. So the probability that the discriminator wrongly classifies a created sample as a real one is very close to zero, the gradients vanish.

Improvements: from GAN to Progressive Growing GAN (ProGAN)

The paper on ProGAN addresses some of the problems mentioned above and tries to combat them with various approaches.
The main idea of the ProGAN architecture is to introduce layers with increasingly more detail as the training advances. This change in the architecture allows for “2-6 times faster [training], depending on the output resolution” [3]. It also helps to cope with the instability problem and uses less memory because it spends less training time with the high resolutions. Looking at Figure 2, the generator and the discriminator grow stepwise and in synchrony, starting from a low 4×4 resolution going up to 1024×1024. This allows the network to focus on general concepts first and then continuously attend to more and more detail.

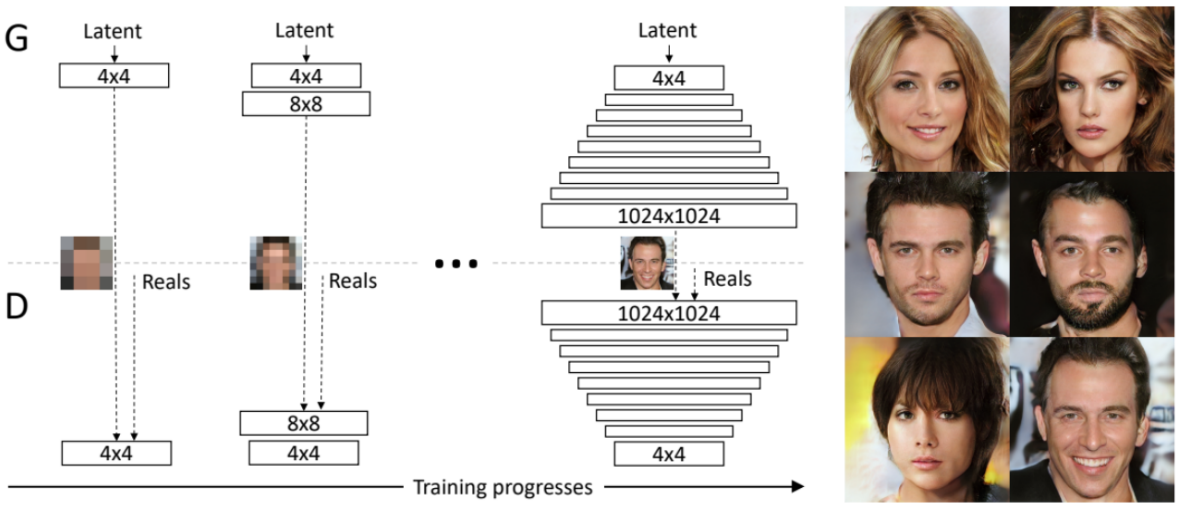

Figure 2 [3]: *Stepwise growing generator and discriminator on the left, generated pictures with 1024x1024 resolution on the right.*

The number of filters for the convolutional layers increases from 16 to 512 in the discriminator and respectively decreases from 512 to 16 in the generator.

Generator	Act.	Output shape	Params
Latent vector	–	512 × 1 × 1	–
Conv 4 × 4	LReLU	512 × 4 × 4	4.2M
Conv 3 × 3	LReLU	512 × 4 × 4	2.4M
Upsample	–	512 × 8 × 8	–
Conv 3 × 3	LReLU	512 × 8 × 8	2.4M
Conv 3 × 3	LReLU	512 × 8 × 8	2.4M
Upsample	–	512 × 16 × 16	–
Conv 3 × 3	LReLU	512 × 16 × 16	2.4M
Conv 3 × 3	LReLU	512 × 16 × 16	2.4M
Upsample	–	512 × 32 × 32	–
Conv 3 × 3	LReLU	512 × 32 × 32	2.4M
Conv 3 × 3	LReLU	512 × 32 × 32	2.4M
Upsample	–	512 × 64 × 64	–
Conv 3 × 3	LReLU	256 × 64 × 64	1.2M
Conv 3 × 3	LReLU	256 × 64 × 64	590k
Upsample	–	256 × 128 × 128	–
Conv 3 × 3	LReLU	128 × 128 × 128	295k
Conv 3 × 3	LReLU	128 × 128 × 128	148k
Upsample	–	128 × 256 × 256	–
Conv 3 × 3	LReLU	64 × 256 × 256	74k
Conv 3 × 3	LReLU	64 × 256 × 256	37k
Upsample	–	64 × 512 × 512	–
Conv 3 × 3	LReLU	32 × 512 × 512	18k
Conv 3 × 3	LReLU	32 × 512 × 512	9.2k
Upsample	–	32 × 1024 × 1024	–
Conv 3 × 3	LReLU	16 × 1024 × 1024	4.6k
Conv 3 × 3	LReLU	16 × 1024 × 1024	2.3k
Conv 1 × 1	linear	3 × 1024 × 1024	51
Total trainable parameters			**23.1M**

Discriminator	Act.	Output shape	Params
Input image	–	3 × 1024 × 1024	–
Conv 1 × 1	LReLU	16 × 1024 × 1024	64
Conv 3 × 3	LReLU	16 × 1024 × 1024	2.3k
Conv 3 × 3	LReLU	32 × 1024 × 1024	4.6k
Downsample	–	32 × 512 × 512	–
Conv 3 × 3	LReLU	32 × 512 × 512	9.2k
Conv 3 × 3	LReLU	64 × 512 × 512	18k
Downsample	–	64 × 256 × 256	–
Conv 3 × 3	LReLU	64 × 256 × 256	37k
Conv 3 × 3	LReLU	128 × 256 × 256	74k
Downsample	–	128 × 128 × 128	–
Conv 3 × 3	LReLU	128 × 128 × 128	148k
Conv 3 × 3	LReLU	256 × 128 × 128	295k
Downsample	–	256 × 64 × 64	–
Conv 3 × 3	LReLU	256 × 64 × 64	590k
Conv 3 × 3	LReLU	512 × 64 × 64	1.2M
Downsample	–	512 × 32 × 32	–
Conv 3 × 3	LReLU	512 × 32 × 32	2.4M
Conv 3 × 3	LReLU	512 × 32 × 32	2.4M
Downsample	–	512 × 16 × 16	–
Conv 3 × 3	LReLU	512 × 16 × 16	2.4M
Conv 3 × 3	LReLU	512 × 16 × 16	2.4M
Downsample	–	512 × 8 × 8	–
Conv 3 × 3	LReLU	512 × 8 × 8	2.4M
Conv 3 × 3	LReLU	512 × 8 × 8	2.4M
Downsample	–	512 × 4 × 4	–
Minibatch stddev	–	513 × 4 × 4	–
Conv 3 × 3	LReLU	512 × 4 × 4	2.4M
Conv 4 × 4	LReLU	512 × 1 × 1	4.2M
Fully-connected	linear	1 × 1 × 1	513
Total trainable parameters			**23.1M**

Figure 3 [3]: *Generator and Discriminator architectures.*

The new layers are faded in smoothly and all existing layers remain trainable. As demonstrated in Figure 4, the output from the prior layer with lower resolution is upsampled for the generator (respectively downsampled for the discriminator). Then two pathways are created. One pathway goes through a newly introduced layer block with the higher resolution input, followed by a 1×1 convolution layer which transfers the input to a $N \times N \times 3$ image. The other pathway goes right through a 1×1 convolution layer, without another layer block. Now both activations are weighted with α and $(\alpha - 1)$ and then merged. In other words, "the layers that operate on the higher resolution [are] like a residual block, whose weight α increases linearly from 0 to 1" [3]. At first, when $\alpha = 0$, only the up/downsampled output from the layer block before is considered. With increasing α, this bias is progressively shifted towards the new output [5].

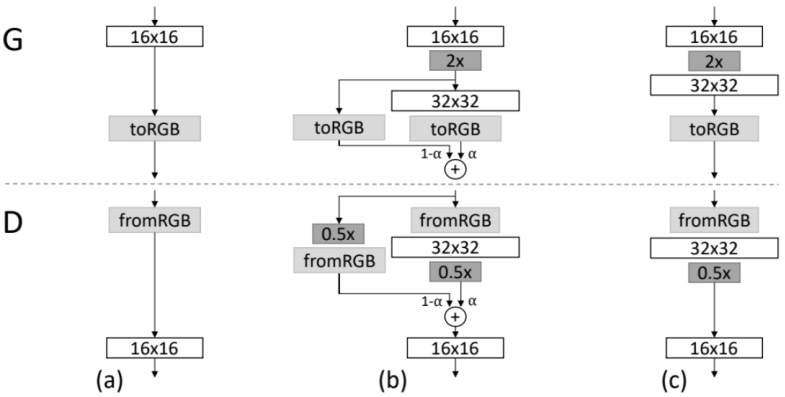

Figure 4 [3]: *Generator and discriminator (a) before, (b) during and (c) after transition from 16x16 to 32x32 resolution.*

There are different approaches to increase variance to combat mode collapse. The one introduced in Karras et al. (2018) is called minibatch standard deviation. The idea is to add a simple constant feature map so that the batch of generated samples will share similar features with the batch of real samples. This is accomplished by calculating the standard deviation of each feature across channels and then averaging it over the minibatch [6]. This layer is inserted towards the end of the network. It is a rather simple method which does not require any additional trainable parameters [3].
Usually, batch normalization is used after each convolutional layer in the generator and discriminator to prevent exploding gradients. Karras et al. (2018) introduce a different normalization technique called “pixelwise feature vector normalization” which is only used in the generator after each convolutional layer and defined as: $b_{x,y} = \frac{a_{x,y}}{\sqrt{\frac{1}{N}\sum_{j=0}^{N-1}(a_{x,y}^{j})^2 + \epsilon}}$, where $\epsilon = 10^{-8}$ is added to deal with zero means, “N is the number of feature maps, and $a_{x,y}$ and $b_{x,y}$ are the original normalized feature vector in pixel (x, y), respectively” [3].
So this layer normalizes each pixel in the activation maps to unit length, which effectively prevents exploding gradients.
Instead of carefully initializing the network's weights, the paper initializes the weights using $\mathcal{N}(0, 1)$ and then introduces an equalized learning rate. This scales the weights w_i at each layer with a constant c: $w'_i = w_i/c$, where c is the per-layer normalization constant from He’s initializer [8]. It is performed during runtime in order to keep the network's weights at a similar scale during training. It ensures that the learning speed is the same for all weights [3].
The ProGAN uses improved Wasserstein loss WGAN-GP, where the GP stands for gradient penalty, and the Adam optimizer with $\alpha = 0.001$, $\beta_1 = 0$, $\beta_2 = 0.99$, $\varepsilon = 10^{-8}$.

Our Approach

As our computational resources are limited and our aim for this project is not recreating high resolution results, but rather understanding the underlying architecture and its parts, we simplify some of the mentioned aspects. We align our implementation with Jason Brownlee’s article on implementing ProGANs [5].
In general, we tried to implement our version of the ProGAN as described in the original paper. But as mentioned, due to limited computational resources, we produce images with a lower resolution (64×64), use less training images (10,000) and also less training epochs.
We use the CelebA dataset [9] which contains 202599 pictures of celebrity faces. The original dataset also holds information about different features of the face displayed in an image e.g. if the person is bald or wears glasses. For our purposes we are only interested in the images though. As a starting point we used the img_align_celeba version of the dataset which already cropped and aligned all of the images to the same dimensions. But this preprocessing is not sufficient for the task at hand. We additionally need to extract only the faces of each picture, since we are not interested in the background. In order to extract the faces we used a pre-trained Neural Network called MTCNN to perform face detection on the images [10].
Our loss function differs in the way that we use normal Wasserstein loss without gradient penalty. Lastly, for simplicity sake, we decided to define sequential models.

Our results

As expected due to our simplifications made to the task our results can not match those of the paper. During our training process we first struggled with extremely high losses for resolutions higher than (16×16) that lead to NaN values in our losses and thus stopping our network from learning. We were able to counteract this by implementing an equalized learning rate also featured in the paper. With this improvement our model was able to tackle higher resolutions without collapsing.

Figure 5: *Generated images with 4x4 (a), 8x8 (b), 16x16 (c), 32x32 (d)*

Throughout the progression of our models training the underlying idea of a ProGAN can be observed. Our result images in Figure 5 slowly increase in detail and as the resolution increases more detailed areas of the face are improved. At first, the images only capture the general shape of a face and we can see darker areas where the eyes would be (a,b). In (c) and (d), the eyes are further detailed and other facial features like eyebrows, noses, mouths and even teeth become visible.

With increasing resolution we also notice that the generated images share many features. It seems like the generator does not capture enough variance of the data distribution. This tendency is most notable in the shift from (d) to (e).
Although the ProGAN paper suggests using MinibatchStdev in order to introduce more variation to the data, this change does not seem sufficient for our implementation. The paper we referred to in our introduction however, amongst other new ideas, considers various new methods to introduce even more variance to our model.

Improvements for ProGAN

The Style-GAN implementation by Karras et al. (2018) applies some innovative architectural changes which further improve the quality of generated portraits. We will shortly summarize these improvements because we think that they produce amazing results.
Generally Style-GAN uses the ProGAN architecture as its base, but they also use “style” vectors coming from style transfer - e.g. applying a certain painting style to a photo. There are five major differences which contribute to the impressingly realistic looking generated images as seen in Figure 5.

Figure 6 [2]: *Images generated with Style-GAN using FFHQ dataset*

1. Bilinear Sampling:

Instead of using the nearest neighbor method for upsampling like in the ProGAN, the Style-GAN uses bilinear upsampling layers which are implemented “by low-pass filtering the activations with a separable 2nd order binomial filter after each upsampling layer and before each downsampling layer” [2]. While nearest neighbor sampling copies the value from the nearest pixel, bilinear sampling uses the surrounding pixels to calculate the pixel’s value using linear interpolations. This can improve the image quality and makes it look smoother.

2. Mapping network and AdaIN:

The latent space input for the generator is first fed through a mapping network which comprises 8 fully connected layers instead of feeding it directly into the generator. The output of this network is then converted into “styles” with learned affine transformations. These styles control the coarse components (pose, face shape, glasses, etc.) and the soft aspects (eyes, hair, lighting, etc.) of the image. “The style vector is then transformed and incorporated into each block of the generator model after the convolutional layers via an operation called adaptive instance normalization or AdaIN” [7]. AdaIN is defined as: $AdaIN(x_i, y) = y_{s,i}\frac{x_i - \mu(x_i)}{\sigma(x_i)} + y_{b,i}$, where x_i is each feature map which is

normalized separately with its mean μ and standard deviation σ, “and then scaled and biased using the corresponding scalar components from style y” [2].
The mapping network is added to avoid the inevitable entanglement of the latent input space which arises because it “must follow the probability density of the training data” [2].

3. Removal of traditional input:
   The input of the “synthesis network” - basically the next component of the generator after the mapping network - is simplified to a constant 4x4x512 tensor, instead of the latent vector input as in the ProGAN.
4. Addition of noise
   To account for stochastic variation, like “exact placement of hairs, stubble, freckles, or skin pores” [2] noise inputs are included.
   After each convolutional layer in the synthesis network, a single channel image of Gaussian noise is added before the AdaIN layer. “A different sample of noise is generated for each block and is interpreted using per-layer scaling factors” [7].
5. Mixing regularization
   For this method, two latent codes are fed through the mapping network. The resulting style y_1 is applied up to a random point in the synthesis network, from thereon style y_2 is applied. “This encourages the layers and blocks to localize the style to specific parts of the model and corresponding level of detail in the generated image” [7].

Literature

[1] Alexandre Matton and Amaury Sabran. Faster Transformers for Text Summarization. Stanford CS224N Project Report, 2019.

[2] Tero Karras, Samuli Laine, Timo Aila. A Style-Based Generator Architecture for Generative Adversarial Networks. arXiv:1812.04948v3 [cs.NE], 2019.

[3] Tero Karras, Timo Aila, Samuli Laine, Jaakko Lehtinen. Progressive Growing of GANs for Improved Quality, Stability and Variation. arXiv:1710.10196v3 [cs.NE], 2018.

[4] Overview of GAN Structure. Google Developer Guide, (last updated) 2019.

[5] Jason Brownlee. How to implement Progressive Growing GAN Models in Keras. Machine Learning Mastery, 2019.

[6] Kang & Atul. An Introduction To The Progressive Growing of GANs. The AI Learner, 2019.

[7] Jason Brownlee. A Gentle Introduction to StyleGAN the Style Generative Adversarial Network. Machine Learning Mastery, 2019.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. arXiv:1502.01852v1 [cs.CV], 2015.

[9] CelebA Dataset from Kaggle

[10] GitHub repository for the MTCNN face detector, based on:
Kaipeng Zhang, Zhanpeng Zhang, Zhifeng Li, and Yu Qiao. Joint Face Detection and Alignment using Multi-task Cascaded Convolutional Networks. arXiv:1604.02878v1 [cs.CV], 2016.