GENERATING IMAGES USING GENERATIVE ADVERSARIAL NEURAL NETWORKS

Simon Felix Seeger

ZUSAMMEN.WACHSEN.

Supervising Teacher: Tim Storck

Rupprecht Gymnasium München

Bavaria, Germany

November 7, 2022

Contents

List of Figures

1 Introduction

With DALL·E2 being able to generate images of stunning quality (Ramesh et al., 2022), AI (Artificial Intelligence) generated imagery has increased in popularity[1]. This is why this Paper discusses multiple Generative Adversarial Neural Networks (GAN) architectures and explains a model generating faces. GANs are, compared to for example diffusion model, an easy way to generate images.

2 Definitions

- The $\times$ symbol represents n-Dimensional shapes. For example a two dimensional array with a shape of 8×8 holds 64 elements.
- Although machine learning algorithms use n-Dimensional Tensors[2] for all calculations and data storage, this paper will use vectors and matrices interchangeably for simplicity.
- Long indexes are symbolized using square brackets to make equations more readable; short indices are represented by subscripts

3 Basic Neural Network Architecture

A neural Networks consists of layers, each passing their output to the next layer until the last one, called the output layer, is reached.

3.1 Perceptron

A Perceptron takes one or more inputs and returns a value between 0 and 1 based on those inputs. The connection between the input x_i and the Perceptron holds a weight w_i, which assigns a priority or importance to the input. The Perceptron itself also holds a bias b. The bias enables the Perceptron to reach the threshold easier or harder. To make calculating the values of the Perceptron easier, both the inputs and weights are represented by a vector containing the corresponding values. The vectors are called x and w accordingly. Both are of the same site m. Each Perceptron calculates the function seen in equation 1 (Nielsen, 2015, Chapter *Using neural nets to recognize handwritten digits - Perceptrons*).

$$z(x) = \begin{cases} 0, & \text{if } b + \sum_{i=0}^{m} w_i \cdot x_i \leq \textit{threshold} \\ 1, & \text{if } b + \sum_{i=0}^{m} w_i \cdot x_i > \textit{threshold} \end{cases} \tag{1}$$

3.2 Neuron

Neurons are modified Perceptrons. An example representation can be seen in figure 2. Here, the white circle is a Perceptron connected with the inputs x using blue and red arrows. In this example the weights are represented, by the color. Red means that w_i is a positive value, while a negative value has a blue connection. The stronger the color, the larger the value of $|w_i|$. Like

[1] Based on Google Trends https://trends.google.de/

[2] **Tensor:** Matrix optimized for machine learning arithmetic

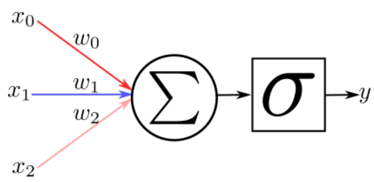

Figure 2: Example Structure Neuron

the Perceptron, the neuron calculates the weighted sum as seen in equation 2. In addition, the Neuron has an activation function σ

$$y = \sigma\left(b + \sum_{i=0}^{m} w_i \cdot x_i\right) \tag{2}$$

3.3 Activation functions

The activation function of an layer is used to ensure that small changes to the weights and biases of a neuron don't affect the whole outcome in an unpredictable way (Nielsen, 2015, Chapter *Using neural nets to recognize handwritten digits* - *Sigmoid Neurons*). We will simplify this function as $\sigma(z)$. The parameter z describes the "raw" output of the neuron. Commonly used functions can be seen in figure 3. Each of them serves an specific purpose depending on the type of model. The commonly used sigmoid neuron calculates:

$$\frac{1}{1 + e^{-\left(b + \sum_{i=1}^{n} w_i \cdot x_i\right)}}$$

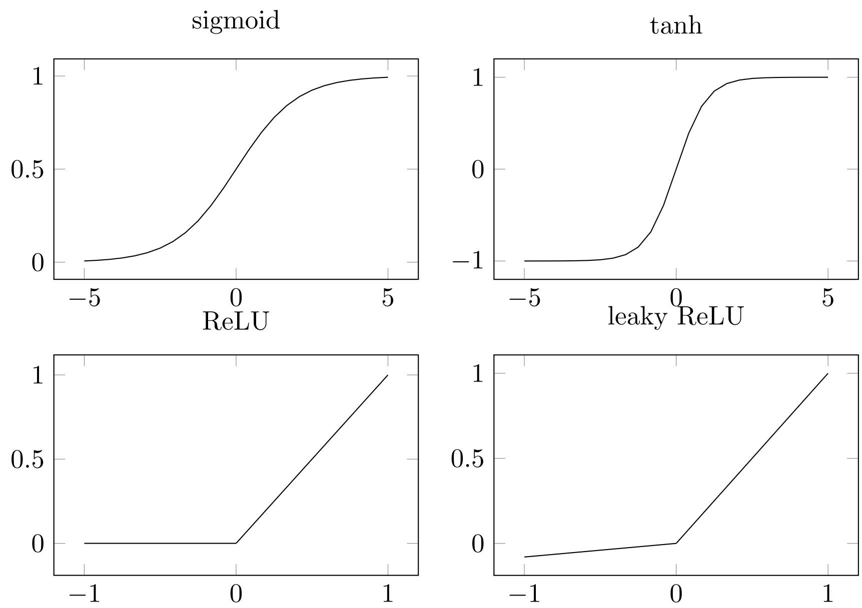

Figure 3: Different activation functions commonly used in machine learning

The other functions in this plot can be calculated as followed:

tanh:

$$\sigma(x) = tanh(x)$$

ReLU (Rectified Linear Unit):

$$\sigma(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{if } x \leq 0 \end{cases}$$

leaky ReLU[3]**:**

$$\sigma(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha x, & \text{if } x \leq 0 \end{cases}$$

3.4 Loss function

A loss function, also called error function, assigns a value to an output $\hat{y}$ of the model. This Value $Loss(\hat{y})$ depends on the expected output of the model. It is needed to optimize the model. The simplest of loss functions is the mean squared error seen in equation 3.

$$Loss(y) = (y - \hat{y})^2 \tag{3}$$

Let's take a model M build to classify images into two categories: Cats ($y_{cat} = 0$) and Dogs($y_{dog} = 1$) as an example. If the model classifies a image of a cat with the value $\hat{y} = 0.1$ then the loss of this image would be really small ($Loss(0.1) = 0.01$) since $\hat{y}$ is close to the expected value. But if the model classifies the same image as a dog e.g $\hat{y} = 1.8$, the loss increases ($Loss(1.8) = 3.24$). The goal of a model is to minimize loss.

4 Common Machine Learning Terms

4.1 Convolutional layer

Convolutional layers enable finding detail across an image by calculating a part of the input with a filter F. The filter is often called the kernel. This process is called a discrete convolution. The trainable parameters are the values inside F and the bias b. The hyperparameters of a convolutional layer are the size and amount of filters, the stride (The amount of pixels the filter gets shifted each step) and the zero padding of the input image. (Gu et al., 2017) In equation 4 a function for two dimensional convolution using one filter of uneven size can be seen. Here w_s and h_s are the width and height of the strides and w_F and h_F the width and height of the filter respectively. Using this equation, the padding is dependent of the filter size and the index of the output image.

$$O[m, n] = I[m \cdot w_s - w_F : m \cdot w_s + w_F, n \cdot h_s - h_F : n \cdot h_s + h_F] \cdot F + b \tag{4}$$

When the layer uses strides, the image is scaled down. Therefore the output size of a quadratic image can be calculated as followed with o, i and f being the size of the output image, input image and filter correspondingly.

$$o = \frac{i + 2p - f}{s} + 1 \tag{5}$$

Convolutional layers proof as exceptionally useful in classification problems, where the object is not strictly centered but can be in any place of the image. Informal speaking a convolutional

[3] usually $\alpha = 0.01$

layer creates a map showing how much a part of an image represents the object. In image 4

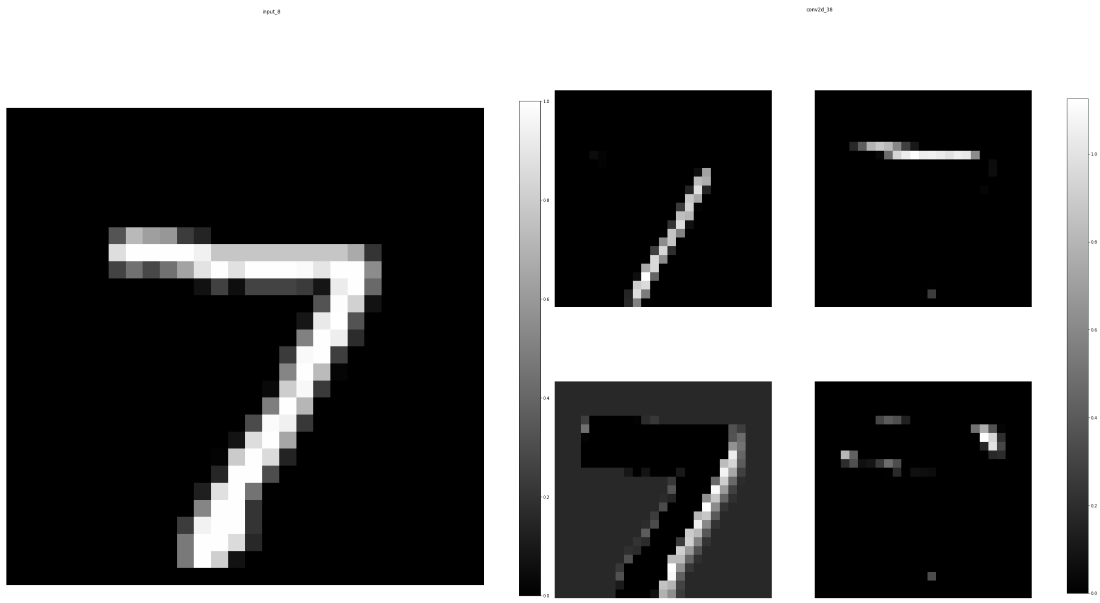

Figure 4: **Left:** input image out of the MNIST Dataset **Right:** Feature maps after the first convolution

you can see an input image (left) out of the MNIST dataset[4]. On the right, you can see the feature maps generated by a filter. It is visible how the models filter pics up on key details of the number like the diagonal and top line of the seven.

4.2 Transpose layer

While convolutional layers often reduce the size of the input matrix, transpose layers, also called deconvolution layers, increase the size of the input matrix using its filter F when strided or padded. The resulting output size can be calculated like seen in equation 6

$$o = (i - 1) \cdot s + k - 2p \tag{6}$$

4.3 Binary Cross Entropy

Binary Cross Entropy is a loss function commonly used for classification problems where data can be classified into a binary choice (e.g. yes or no). Given the label or expected value y and the probability of a prediction, generated by the model, p, the loss can be calculated like seen in equation 7.

$$BCELoss(y, p) = -\left(y \cdot log(p) + (1 - y) \cdot log(1 - p)\right) \tag{7}$$

When having multiple labels and predictions the loss can be calculated like seen in equation 8. N in this case is the amount of labels.

$$BCECost(y, p) = -\frac{1}{N} \sum_{i=1}^{N} y_i \cdot log(\hat{y_i}) + (1 - y_i) \cdot log(1 - \hat{y_i}) \tag{8}$$

[4]http://yann.lecun.com/exdb/mnist/

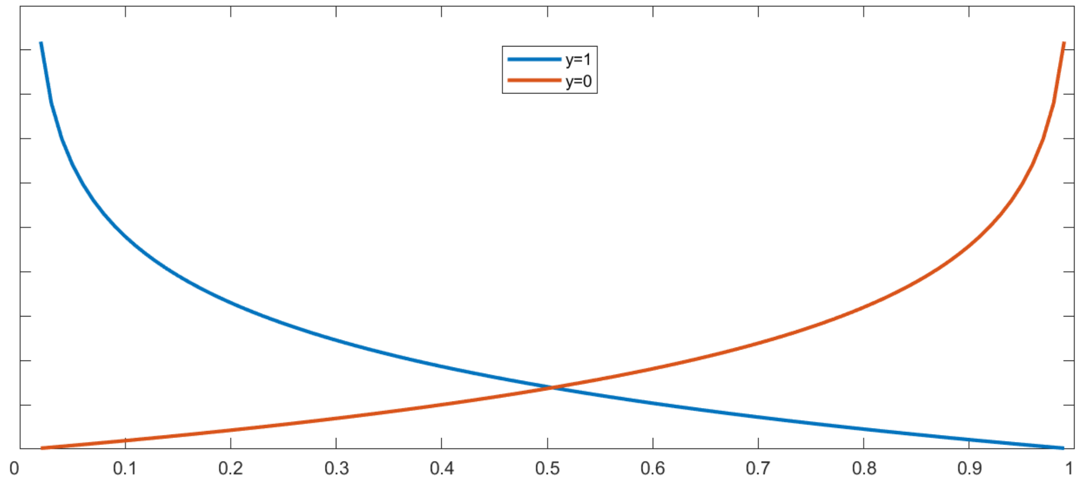

Figure 5: Binary Cross Entropy Loss

The function 7 has some problems:
First, $\lim_{x \to 1} Loss(1, x) = \infty$ and $\lim_{x \to 0} Loss(0, x) = \infty$ meaning that $Loss(a, a); a \in \{0, 1\}$ is undefined. A undefined loss means the model is unable to learn and adjust. Also, since the loss grows rapidly with values nearing the limit of the functions domain, optimizing the model gets uncontrollable. The easiest way to counter these effects is to clamp p to a arbitrary range $[eps, -eps]$ with $0 < eps < 1$. The python package scikit-learn uses a value of $eps = 10^{-15}$. Another problem is that models usually don't output probabilities, but logits, which are incompatible with the above defined loss function. To convert the floats into probabilities the sigmoid function can be used.[5] With $\sigma(x) = \frac{1}{1+e^{-x}}$ and $\hat{y}$ being the output of the model, the probability can be calculated as follows:

$$p = \sigma(\hat{y}) \tag{9}$$

This updates the BCE Loss function to the one seen below.

$$BCELoss(y, \hat{y}) = -\left(y \cdot log(\sigma(\hat{y})) + (1 - y) \cdot log(1 - \sigma(\hat{y})\right) \tag{10}$$

5 Preprocessing the training data

To make the training of the models easier, the training data needs to be preprocessed. First of all, all images need to be scaled to the same size. After resizing, the images have to be converted to tensors. The Tensor has the shape $w \times h \times 3$ with w being the image width and h the height. Since an RGB image has three color channels, the tensor has a depth of 3. The last step is to clamp the color values of the image between -1 and 1 since Machine Learning Algorithms learn better with smaller numbers.

6 Generative Adversarial Networks

A Generative Adversarial Network (GAN) consists out of two models which are trained simultaneously: a Generator G and a Discriminator D. While the Generator generates images, the

[5]See 3.3 for more information on this particular activation function.

Discriminator gets trained to differentiate between real images from the dataset and the images made by the Generator or simply put learns to distinguish real from fake images. D's output is used to backpropate through G and thus enables it's training process. The Generators goal is to deceive the Discriminator while D's goal is to classify images perfectly.
All GANs need an input, for example a random vector of numbers to generate an output. The following subsections outline different kinds of GAN architectures, each with a different purpose or different advantages and disadvantages.

6.1 Vanilla GAN

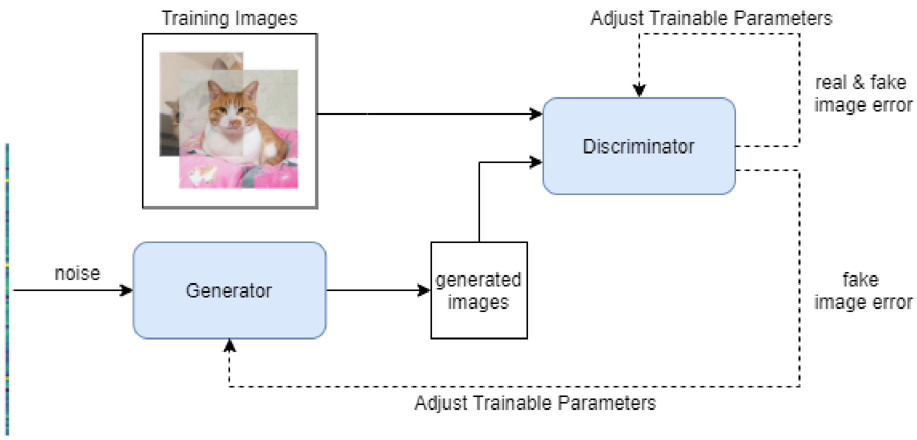

Figure 6: Black box training diagram of a GAN; Cat Images from https://www.microsoft.com/en-us/download/details.aspx?id=54765

Vanilla GANs mostly use dense layers for both the Generator and Discriminator. Since dense layers are just matrix operations, this kind of GAN is the fastest out of all the following.

6.1.1 The Generator Model

The Generator uses a latent vector z containing normal distributed random values as its input. this vector gets passed through the network producing an image as the result. In a Vanilla GAN architecture, the network is composed of dense layers. Since the training data was normalized to values between -1 and 1, the tanh activation function is used for the Generators output.

The first dense layers has $w_I \cdot h_i \cdot n$ Neurons with $w_I, h_I, n \in \mathbb{N}_{\neq 0} \cap w_I < w_O, h_I < h_O$.[6] n is an arbitrary number. Continuing this process, the output of the previous layer is computed sequentially with multiple fully connected hidden layers, until the desired output shape is reached.

6.1.2 The Discriminator

The Discriminator classifies images into how likely it is that the image is real. The closer this probability p approaches 1, the more certain is the Discriminator that the image is out of the training set. The Discriminator is not able to output bigger probabilities than 1 since p is a probability $1 \geq p \geq 0$ applies.

[6] w_0 and h_O being the size of the Output

6.1.3 Training the GAN

Since GAN training involves two models competing, a custom training loop is needed. In figure 6 a black box training diagram for a GAN can be seen.
First noise needs to be generated which gets passed to the Generator. G uses this latent vector the produce images O_G. The Discriminator evaluates O_G and outputs a vector evaluating how real each image in the batch seems. We will call this $O_{D(G(z))}$.

While it is the Discriminators goal to maximize assigning the right values to images x, the Generator tries to minimize $\log(1 - D(G(z))$. This minimax game between G and D gives this loss function the name Minimax Loss which was introduced in the paper by Goodfellow et al. (2014) and can be seen in equation 11. Here E represents the expected value over all occurrences of real data or generated data.

$$\min_G \max_D V(D, G) = E_x[\log(D(x))] + E_z[\log(1 - D(G(z))] \tag{11}$$

A problem with this function is, that the gradient in the early training process, in which D has an easy job telling real and fake samples apart, is not sufficient to train the Generator. They propose letting G maximize $\log(D(G(z))$ instead of minimizing $\log(1 - D(G(z))$. This is the reason why D uses the probability 1 for real images.

Using equation 11, the gradients for the Discriminator can be calculated like seen in equation 12 with the batch size m and the trainable parameters θ.

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=0}^{m} \log(D(x_i)) + \log(1 - D(G(z_i)) \tag{12}$$

In the same sense, the Generators gradients can be calculated visible in equation 13

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=0}^{m} \log(1 - D(G(z_i)) \tag{13}$$

Because minimax loss is an alternation of the binary cross entropy loss, another way to rewrite it is using the $BCECost$ function. Therefore the Generators loss $Loss_G = BCECost(1, O_{D(G(z))})$. In consideration of the Discriminator also having to learn the structure of real images, it is also shown a batch of real images, with the fake image batch and the real image batch being the same size. After producing the vector $O_{D(x)}$ for the real images, the Discriminator loss can be calculated as followed: $Loss_D = BCECost(0, O_{D(G(z))}) + BCECost(1, O_{D(x)}$.
It is important to note, that the Generator never sees real images, but only improves via D's feedback.

6.2 Deep Convolutional GAN

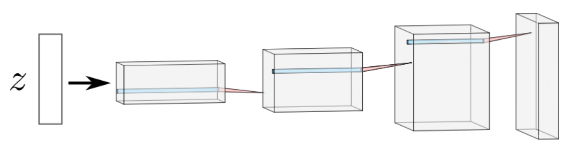

Figure 7: AlexNet representation of a DCGAN's Generator

Deep Convolutional GANs or DCGANs generate images using a convolutional architecture. The architecture was first introduced in a paper called *Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks* (Radford et al., 2016). Following this publication, there are some guidelines for building a DCGAN. First both G and D should use batch normalisation. Secondly the Architecture should follow the Convolutional Neural Network architecture specification. Third, the Adam optimizer should be used when possible and last, the Generator should use ReLu activation functions for all layers except the output which should be passed through the tanh function. The Discriminator should use leaky ReLU for all layers. This means that the network uses transpose layers instead of dense layers. A example representation can be seen in figure 7

6.2.1 The Generator

The Generator of an DCGAN begins similar to a Vanilla GAN. The input layer has a shape of n_z, this gets passed down to a dense layer with $w \cdot h \cdot n$ Neurons and a ReLU activation layer. The vector output of the dense layer gets reshaped to a $h \times w \times n$ sized matrix. This way we can pass it into transpose layers until the desired Output shape $h_O \times w_O \times 3$ is reached. Each transpose layer has an ReLU activation function. More on transpose layers see section 4.2.

6.2.2 The Discriminator Model

The Discriminator has an input layer in the same shape as the training images. This image gets passed through multiple convolutional layers and gets flattened[7] afterwards to enable processing the feature maps created by the Convolutional layers. The flattened vector is fed into a dense layer with only one neuron. This is the output of the Discriminator.

6.3 Conditional GAN

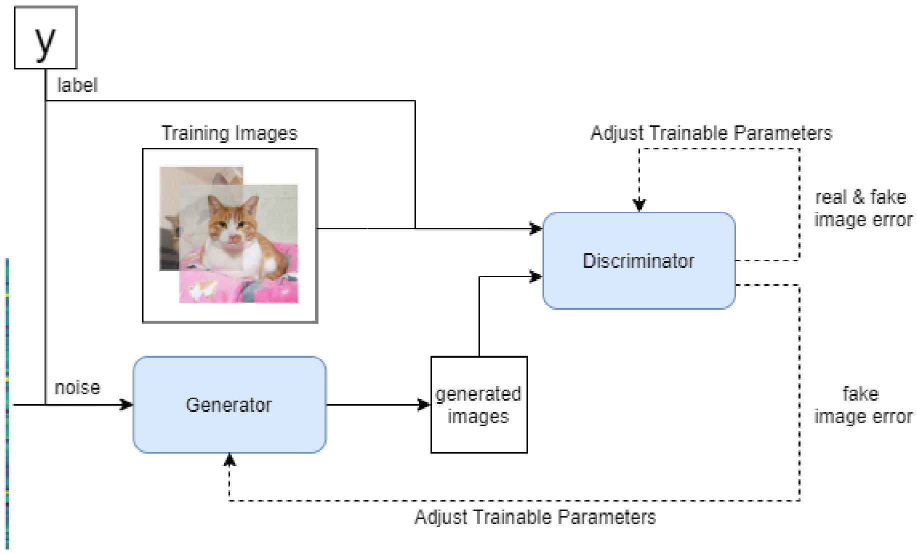

Figure 8: Black box training diagram of a CGAN; Cat Images from https://www.microsoft.com/en-us/download/details.aspx?id=54765

[7]Transforming the multi dimensional input into a vector Output

A conditional GAN or CGAN uses, in addition to the inputs described in section 6.1, labels provided through a secondary input in both Discriminator and Generator model. This architecture was introduced by Mirza and Osindero (2014). They were able to generate for example numbers from the *MNIST* dataset with the number generated resembling the label used while generating. Figure 8 shows a black box training diagram for CGANs.
The updated objective function $V(D, G)$ for a CGAN can be seen in equation 14

$$\min_G \max_D V(D, G) = E_x[\log(D(x|y))] + E_z[\log(1 - D(G(z|y))] \tag{14}$$

6.4 Problems

6.4.1 Mode Collapse

Figure 9: Output of an collapsed model

Mode collapse means that the Generator produces only similar looking images instead of the wanted variety and was already mentioned in the paper introducing Generative Adversarial Networks (Goodfellow et al., 2014). This is the case when G finds a output which is particularly good at fooling the Discriminator and thus learns to only produce one kind of image. Since D learns from generated images, it will reject all outputs generated by G. By doing so, the Discriminator gets stuck in a local minima leading to the Generator learning to generate a different set of homogeneous outputs. It is not possible to recover from mode collapse, since G will always outplay the Discriminator by "hopping" from one local minima to another.
In figure 9 mode collapse can be seen. Although there is some variety, many images look identical.

6.4.2 Vanishing Gradients

A vanishing gradient is a problem where while performing backpropagation on deeper models, the gradient used is so insignificantly small, that it does not have any effect on the models performance or progress while training. Small gradients occur when the Discriminator gets too good at identifying the images made by the Generator and thus giving not enough information to train G. The vanishing gradient problem is not reserved to Generative Adversarial Networks but can happen in any deep neural network. While optimizing a model the derivatives of the layers get multiplied with each other layer after layer. With very small derivatives this leads to the gradient of the model in the end becoming extremely small and hence learning becomes nearly impossible.

6.4.3 Failure to converge

When training a GAN, both the Generator and Discriminator compete against each other. This means, that a Generator performing exceptionally well makes the Discriminator less efficient and thus decreasing the value of the feedback generated by the Discriminator since D comes close to randomly guessing if an image is generated or not. Especially on long training periods this can become an issue with the Generator starting to produce useless images to satisfy the improper advice.

6.5 Improving image Quality

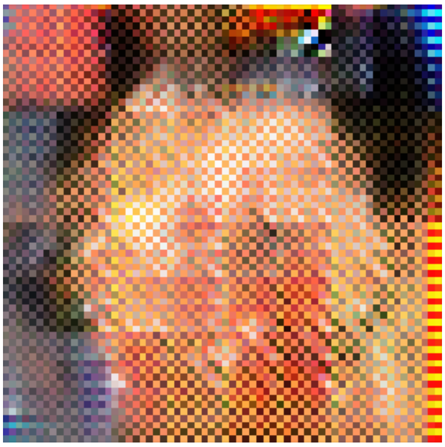

Figure 10: Checkboard pattern created by uneven convolutions

One of the big problems while using transposed convolutions is the creation of checkboard pattern artifacts like seen in the image on the left. This is caused by uneven overlapping convolutions. "In particular, deconvolution has uneven overlap when the kernel size [...] is not divisible by the stride [...]" (Odena et al., 2016). In an ideal scenario, the model would learn to adjust to the checkboard pattern, but quite the opposite is true. It is even possible for models with even overlap to learn this kind of artifact (Odena et al., 2016).
The article by Odena et al. (2016) proposes resizing the images using a interpolation function and then using a convolutional layer to add detail to the image. This is process called "resize-convolution".

7 Generating Faces using a Deep Convolutional Generative Adversarial Neural Network

Artificial images of faces can be useful in many different applications. This includes copyright free images of people which can be used in advertising, media or anything where a face should represent a non existent person. A prime example of face generation is https://thispersondoesnotexist.com/. It uses the network SyleGAN2 introduced by Karras et al. (2020).
The approach I chose was building a DCGAN in python and training it on the CelebA(Liu et al., 2015) dataset which was not changed except a resize to 64×64 pixels.

7.1 Tensorflow Models

Both models were created in the TensorFlow framework (Martín Abadi et al., 2015). TensorFlow allowed me to create and experiment with the models in a quick manner. As described in Section 6.1, a generator and a discriminator model is needed.

7.1.1 The Generator Model

The Generator uses a latent vector with the shape $n_z = 128$ to provide enough input variance as input. The input layer is connected to a dense layer with $4 \cdot 4 \cdot 256$ or 4096 neurons and a ReLU activation function. To continue in a convolutional fashion, the vector outputted by the fully connected layer needs to be reshaped into a three dimensional matrix of shape $4 \times 4 \times 256$. The dropout layer with a probability of 20% prevents mode collapse, since the generator is not able to become too adjusted to the current iterations discriminator. This layer is followed by 4 sets of transpose, leaky ReLU and batch normalisation layers. The transpose layers have a filter size of 2×2, a stride of 2. The first Transpose layer or $n = 0$ has 256, the second layer $n = 1$ has 128 filter. Generally speaking the filter amount is determined by $\frac{256}{n \cdot 2}$ The output layer of the generator is a convolutional layer with a filter amount of 3, a filter size of 1×1, a stride of 1 and the tanh activation function so that the Generator produces normalized image data.
The Generator plot can be seen in Figure A.1.

7.1.2 The Discriminator

The Discriminator's input has the shape $64 \times 64 \times 3$, since this is the shape of the images in the dataset. Following this layer is a convolutional layer with 8 filters, each with a size of 3×3 and no stride, an average pooling layer and a leaky ReLU layer with $\alpha = 0.02$. These three layers get repeated three times, with the amount of filters increasing by a factor of 2. The first set of layers out of this collection has in addition a batch normalisation layer. The second has a additional dropout layer with a dropout chance of 30%. The last set has no additional layers but is followed by a flatten layer to convert the three dimensional matrix into a vector. The vector gets passed through a dropout layer (30% Dropout probability) into a dense layer with 32 neurons followed by the output layer with one neuron.
A plot of the Discriminator can be seen in Figure A.2.

7.2 The training Loop

```
for i in range(self.d_steps):
    latent_vector = tf.random.normal(shape=(batch_size, self.latent_dim))
    with tf.GradientTape() as gt:
        generated_images = self.generator(latent_vector, training=True)
        prediction_fake = self.discriminator(generated_images, training=True)

        flipped_images = tf.image.random_flip_left_right(images)
        prediction_real = self.discriminator(flipped_images, training=True)

        d_loss = self.d_loss_fn(prediction_real, prediction_fake)
    d_gradients = gt.gradient(d_loss, self.discriminator.trainable_variables)
    self.d_optimizer.apply_gradients(zip(d_gradients, self.discriminator.trainable_variables))
```

Source Code 1: Discriminator training loop

The training loop was written using a class extending `keras.Model` and overwriting the `train_step()` function so that I could use the `model.fit()` function and additional tools provided by TensorFlow like TensorBoard for measuring progress and callbacks to save images and checkpoints while training.

The discriminator training step can get repeated multiple times if needed. In the current version of the model, this is not needed, since the discriminator slowly converges. Each time a latent vector for the generator with the shape $batch\_size \times 128$ is generated using the `tf.random.normal()` function. This vector is passed into the generator producing a vector of images which are evaluated by the discriminator. To limit the change of the discriminator overfitting and to introduce more variety, the training images get randomly flipped using the function `tf.image.random_flip_left_right()` before being judged by D. The code for the discriminator training loop can be seen in Source Code 1.

The Generator train step is as describes in Section 6.1.3.

The loss is calculated as seen in Source Code 2. It is important to note, that the binary crossentropy function has the `from_logits` attribute set to true, so that the logits produced by the Discriminator can be used for cross entropy loss.[8] The function `discriminator_loss(real_output, fake_output)` calculates $BCECost(1, O_{D(x)}) + BCECost(0, O_{D(G(z))})$. In the same code example it is also visible that the Adam optimizer with a learning rate of $1e-4$ is used for both D and G.

```python
cross_entropy = keras.losses.BinaryCrossentropy(from_logits=True)
def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)

generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

Source Code 2: custom loss functions for both D and G

7.3 Problems while training

Many training iterations failed due to mode collapse. To counter this I tried adding noise to the data passed into the Discriminator to decrease the chance of overfitting. This however did not help as much increasing the length of the latent vector and removing some complexity from the generator.

Another problem was that the Discriminator learned too fast to distinguish the images and thus giving the Generator no meaningful feedback. Fortunately, since the loss of D rapidly approaches zero, this issue can be identified early enough to avoid spending too much time training a model that is destined to fail. This time, reducing the filters by $\frac{1}{4}$ in all convolutional

[8] More on cross entropy loss can be read in Section 4.3.

Figure 11: 16 generated Images using imported model

layers of D resolved the problem.

7.4 Importing the Model into Matlab

The model can be imported into Matlab by using the `importKerasLayers` function. It is important to also load the weights of the model, else model has to be trained again.
An additional step which has to be taken when using my generator is to replace the `tf.keras.layer.Reshape` layer with a custom function layer, since Matlab does have its own corresponding layer.

```
layers = importKerasLayers("generator_model/generator.h5", "ImportWeights",true);

reshape_layer = functionLayer(@(x) dlarray(reshape(x, 4,4,256, []), "SSCB"), ...
                Formattable=true, Acceleratable=true);

layers_new = replaceLayer(layers, "reshape_1", reshape_layer);
generator = dlnetwork(layers_new);
```

Source Code 3: Code to import the Generator

Although I was able to import the model, Matlab had problems interpreting the weights, giving me useless results like in figure 11

7.5 Results

Figure 12: 100 artificial faces

The images visible in figure 12 were generated by the model described in section 7.1.1. This Generator was trained for 455 Iterations. More Images can be generated using the script A.3. This model has clearly some flaws. First of all, the image quality is not great. Some faces are not recognizable, others consist mostly out of black pixels. To improve this the generator network needs to be refined and more training epochs are needed. Second the images suffer from the check board pattern associated with transposed convolutions. More on how to prevent this effect can be read about in section 6.5.
Implementing the aforementioned changes, the image quality could be greatly improved.
Additional Images generated during training can be found on the USB-Stick or at https://github.com/SFSeeger/w-seminar. The code is also available in the appendix: B.1.

7.6 celebGAN2

CelebGAN2 is a project which I made to improve image Quality of the first model. I was not planning on showing it, since it was training while writing this paper and not producing any results showing improvement. This changed however as soon as i saw this paper as completed. This GAN was trained in 319 iterations. Both the generator and Discriminator Loss can be found in the appendix in section A.4.

7.6.1 Changes

Changes were only done to the generator. These changes are moving the Batch normalisation layer before the leaky ReLU layer and changing the last convolution layer to use a kernel size of 4×4 and zero-padding.

7.6.2 Results

Figure 13: 100 Images of celebGAN2

The Results are visible in Figure 13. As shown the faces are better recognisable than in Figure 12 which is promising. But this model still has some issues. First, the Images are less saturated and vibrant as the images produced by the first version of the model and the checkboard effect is still visible. In addition is the background stronger blurred. The code is available in the appendix: B.2.

8 Additional Experiments

8.1 conditional DCGAN

I also tried building a conditional Deep convolutional Neural Network to generate faces[9] based on a label specifying the gender of the generated person. The code for it can be found in the file *DCGAN.ipynb* or in the appendix: B.3. This model however suffered from mode collapse. The image seen in Section 6.4.1 was an output one of the many collapsed GANs produced and I was not able to make the model work reliably. An overhaul of the network architecture would be needed to resolve this issue. This would include changing the loss function to for example the Wasserstein loss (Arjovsky et al., 2017).

[9] using the CelebA Dataset(Liu et al., 2015)

8.2 Matlab DCGAN

This was an attempt to use Matlab's Deep Learning Toolbox to generate handwritten digits using the `digitTrain4DArrayData` Dataset. This attempt was unsuccessful, because the Generator did not converge. Here, a change in Discriminator architecture and the use of a better training loop could return satisfying results. The code can be found in the file *GAN.mlx* or in the appendix: B.5

9 Conclusion

In conclusion, Generative Adversarial Neural Networks are reliable network that can generate digital images with good quality. However, this requires overcoming the instability of the network using various techniques.

With the models introduced in this paper (**celebGAN** and **celebGAN2**) I was able to show some of what can be achieved using Generative Adversarial Neural Networks. Although the images generated by both models suffered from artifacts, faces were recognisable, which shows that both models understood which attributes a face needs to be identifiable. **CelebaGAN2** was able to generate images of higher quality in fewer iterations. This can probably be attributed to the convolutional layer with a larger kernel. Unfortunately, due to lack of time, I was not able to make the other models and processes described in this paper functional.

A future goal I set myself is to improve the **celebGAN2** architecture to generate images with higher quality. It is to be expected that AI generated imagery will play a greater role in our society by supporting the creative process in modern media like games, movies or advertising.

All code referenced here and the images generated by the models detailed above can be found on the included USB-Stick or at https://github.com/SFSeeger/w-seminar.

References

Arjovsky, M., Chintala, S., & Bottou, L. (2017, December 6). Wasserstein GAN. Retrieved November 7, 2022, from http://arxiv.org/abs/1701.07875

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). Generative Adversarial Networks [arXiv:1406.2661 [cs, stat]]. Retrieved October 24, 2022, from http://arxiv.org/abs/1406.2661

Gu, J., Wang, Z., Kuen, J., Ma, L., Shahroudy, A., Shuai, B., Liu, T., Wang, X., Wang, L., Wang, G., Cai, J., & Chen, T. (2017, October 19). Recent advances in convolutional neural networks. Retrieved November 7, 2022, from http://arxiv.org/abs/1512.07108

Karras, T., Laine, S., Aittala, M., Hellsten, J., Lehtinen, J., & Aila, T. (2020, March 23). Analyzing and improving the image quality of StyleGAN. Retrieved November 4, 2022, from http://arxiv.org/abs/1912.04958

Liu, Z., Luo, P., Wang, X., & Tang, X. (2015). Deep learning face attributes in the wild. *Proceedings of International Conference on Computer Vision (ICCV)*.

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Jia, Y., Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, ... Xiaoqiang Zheng. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems [Software available from tensorflow.org]. https://www.tensorflow.org/

Mirza, M., & Osindero, S. (2014, November 6). Conditional generative adversarial nets. Retrieved November 4, 2022, from http://arxiv.org/abs/1411.1784

Nielsen, M. A. (2015). *Neural networks and deep learning*. Determination Press.

Odena, A., Dumoulin, V., & Olah, C. (2016). Deconvolution and checkerboard artifacts. *Distill*. https://doi.org/10.23915/distill.00003

Radford, A., Metz, L., & Chintala, S. (2016). Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks [arXiv:1511.06434 [cs]]. Retrieved November 1, 2022, from http://arxiv.org/abs/1511.06434
Comment: Under review as a conference paper at ICLR 2016

Ramesh, A., Dhariwal, P., Nichol, A., Chu, C., & Chen, M. (2022, April 12). Hierarchical text-conditional image generation with CLIP latents. Retrieved November 7, 2022, from http://arxiv.org/abs/2204.06125

Glossary

B

backpropagation The Process of adjusting the hyperparameters of a model from output to input using the gradients of the model. 11

batch normalisation Takes the average of an inputted batch of training data and normalizes them accordingly. 10

D

dense layer Layer made out of multiple neurons; Also called hidden or fully connected layer. 8, 10, 13

dropout layer Layer that sets random inputs to 0. Prevents overfittitng. 13

H

hyperparameter Parameter which have to be chosen when designing the model. Example: Depth of network or learning rate. 5

O

overfitting A process occurring when the model becomes to specified on the training data thus losing all generality. 14

P

pooling Reduces the size of an input matrix using for example the highest value(max pooling) in an area (pooling size). 13

A Additional Content for Section 7

A.1 Generator Plot

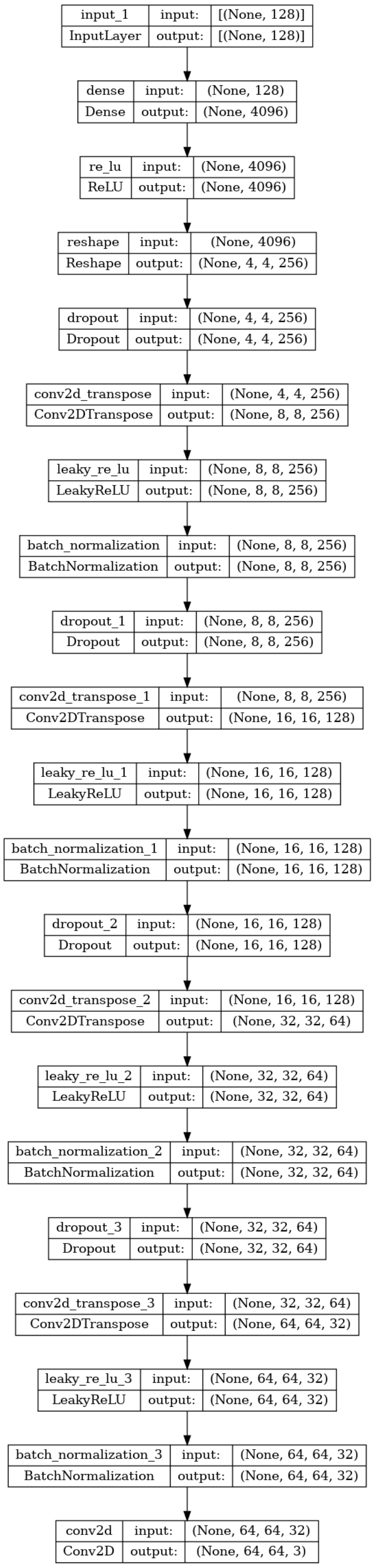

Figure 14: Generator used for celebGAN

A.2 Discriminator Plot

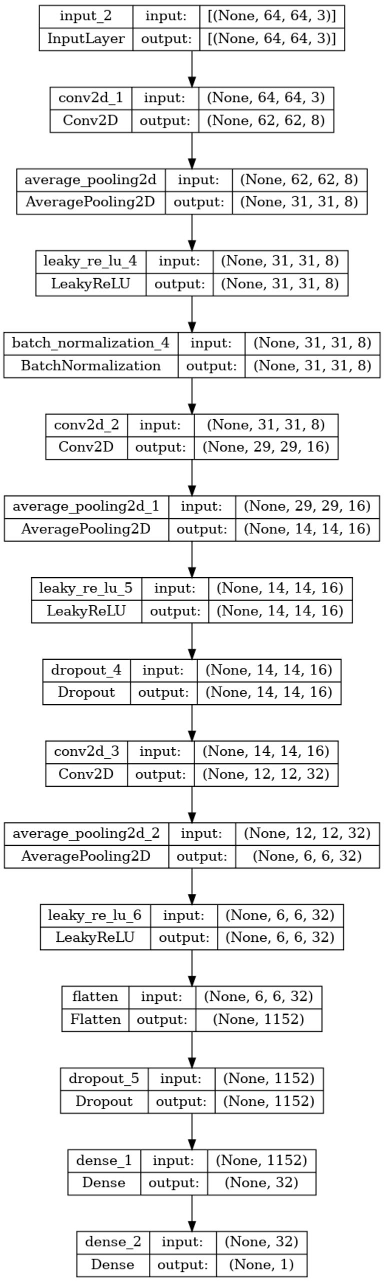

Figure 15: Discriminator used for celebGAN

A.3 Image Generation Code

```python
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np

generator = tf.keras.models.load_model('celebGAN/generator_model/generator.h5')

#for single image
lv = tf.random.normal([1, 128])
image = generator(lv)
imagenorm = image*127.5+127.5
imagenp = imagenorm.numpy()[0,:,:,:]
```

```
plt.imshow(imagenp.astype(dtype="int32"),
    interpolation='antialiased', interpolation_stage="rgba")

#for multiple images
images = 25
predictions = np.empty([100,64,64,3])
for i in range(4):
    seed = tf.random.normal([images, 128])
    label_seed = np.random.randint(0,2, images)
    pred = generator(seed, training=False).numpy()
    predictions[25*i:25*(i+1), :, :, :] = pred

print(predictions.shape)
figsize = 10
fig = plt.figure(figsize=(figsize, figsize))
for i in range(predictions.shape[0]):
    plt.subplot(figsize, figsize, i+1)
    plt.imshow((predictions[i, :, :, :]*127.5+127.5).astype("int32"),
        interpolation='antialiased', interpolation_stage="rgba")
    plt.axis('off')
plt.show()
```

A.4 Loss for celebGAN2

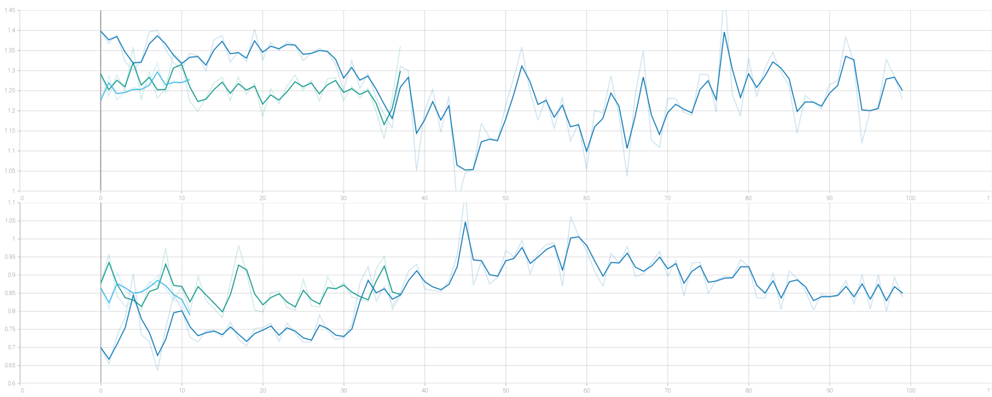

Figure 16: **Top:** Discriminator Loss. **Bottom:** Generator Loss; Each line represents a training process at a different time

B Code

celebGAN

November 7, 2022

[ ]:
```
# basic imports
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

from tensorflow.keras.datasets import mnist

import numpy as np
import matplotlib.pyplot as plt

import os
import io

import datetime

%load_ext tensorboard
```

[ ]:
```
# Set base Parameters
IMAGE_HEIGHT = 64
IMAGE_WIDTH = 64
BATCH_SIZE = 256
```

[ ]:
```
# Load Dataset from storage
AUTOTUNE = tf.data.AUTOTUNE
dataset = tf.keras.utils.image_dataset_from_directory("dataset/preprocessed",
  image_size=(IMAGE_HEIGHT, IMAGE_WIDTH),
  batch_size=BATCH_SIZE)
normalization_layer = tf.keras.layers.Rescaling(1./127.5, offset=-1)
dataset = dataset.map(lambda x, y: (normalization_layer(x), y))
```

[ ]:
```
# Define generator model
def make_generator(latent_vector_shape, dense_shape):
    latent_input = layers.Input(shape=latent_vector_shape)
    gen = layers.Dense(4*4*dense_shape)(latent_input)
    gen = layers.ReLU()(gen)
    gen = layers.Reshape((4,4,dense_shape))(gen)
    gen = layers.Dropout(0.2)(gen)
```

```
    gen = layers.Conv2DTranspose(dense_shape, (2, 2), 2, use_bias=False)(gen)
    gen = layers.LeakyReLU()(gen)
    gen = layers.BatchNormalization()(gen)
    gen = layers.Dropout(0.25)(gen)

    gen = layers.Conv2DTranspose(dense_shape/2, (2, 2), 2, use_bias=False)(gen)
    gen = layers.LeakyReLU()(gen)
    gen = layers.BatchNormalization()(gen)
    gen = layers.Dropout(0.25)(gen)

    gen = layers.Conv2DTranspose(dense_shape/4, (2, 2), 2, use_bias=False)(gen)
    gen = layers.LeakyReLU()(gen)
    gen = layers.BatchNormalization()(gen)
    gen = layers.Dropout(0.25)(gen)

    gen = layers.Conv2DTranspose(dense_shape/8, (2, 2), 2, use_bias=False)(gen)
    gen = layers.LeakyReLU()(gen)
    gen = layers.BatchNormalization()(gen)

    out = layers.Conv2D(3, (1, 1), strides=(1,1), activation='tanh')(gen)

    model: keras.Model = keras.Model(latent_input, out)
    print(model.output_shape)
    assert model.output_shape == (None, 64, 64, 3)
    return model

generator = make_generator(128, 256)
generator.summary()
```

[ ]:
```
#define discriminator model
def make_discriminator(input_shape):
    image_input = layers.Input(input_shape)

    disc = layers.Conv2D(8, (3, 3))(image_input)
    disc = layers.AveragePooling2D()(disc)
    disc = layers.LeakyReLU(alpha=0.02)(disc)
    disc = layers.BatchNormalization()(disc)


    disc = layers.Conv2D(16, (3, 3))(disc)
    disc = layers.AveragePooling2D()(disc)
    disc = layers.LeakyReLU(alpha=0.02)(disc)
    disc = layers.Dropout(0.3)(disc)

    disc = layers.Conv2D(32, (3, 3))(disc)
    disc = layers.AveragePooling2D()(disc)
```

```
    disc = layers.LeakyReLU(alpha=0.02)(disc)

    disc = layers.Flatten()(disc)
    disc = layers.Dropout(0.3)(disc)
    disc = layers.Dense(32)(disc)
    out = layers.Dense(1)(disc)

    model = keras.Model(image_input, out)

    return model

discriminator = make_discriminator((64, 64, 3))
discriminator.summary()
```

[ ]:
```
# define GAN model
class GAN(keras.Model):
    def __init__(self, discriminator, generator, latent_dim=128,␣
 ↪disc_extra_steps=3):
        super(GAN, self).__init__()
        self.discriminator = discriminator
        self.generator = generator
        self.latent_dim = latent_dim
        self.d_steps = disc_extra_steps

    def compile(self, d_optimizer, g_optimizer, d_loss_fn, g_loss_fn):
        super(GAN, self).compile()
        self.d_optimizer = d_optimizer
        self.g_optimizer = g_optimizer
        self.d_loss_fn = d_loss_fn
        self.g_loss_fn = g_loss_fn

    def train_step(self, data):
        images, _ = data
        #calculate bacth size of current batch
        batch_size = tf.shape(images)[0]

        for i in range(self.d_steps):
            #generate new latent vector
            latent_vector = tf.random.normal(shape=(batch_size, self.
 ↪latent_dim))
            with tf.GradientTape() as gt:
                # generate and predict images while being observed by gradient␣
 ↪tape
                # this allows backpropagation and automatic taking of the␣
 ↪derivative
                generated_images = self.generator(latent_vector, training=True)
```

```
                prediction_fake = self.discriminator(generated_images,␣
↪training=True)

                #flip images randomly to inttroduce variety
                flipped_images = tf.image.random_flip_left_right(images)
                prediction_real = self.discriminator(flipped_images,␣
↪training=True)

                 #calculate discriminator loss
                d_loss = self.d_loss_fn(prediction_real, prediction_fake)

            #calculate discriminator gradients
            d_gradients = gt.gradient(d_loss, self.discriminator.
↪trainable_variables)
            #apply gradients using Adam
            self.d_optimizer.apply_gradients(zip(d_gradients, self.
↪discriminator.trainable_variables))

            #generate new latent vector for generator training
            latent_vector = tf.random.normal(shape=(batch_size, self.
↪latent_dim))
            with tf.GradientTape() as gt:
                # generate and predict images while being observed by gradient␣
↪tape
                # this allows backpropagation and automatic taking of the␣
↪derivative
                generated_images = self.generator(latent_vector, training=True)
                prediction_fake = self.discriminator(generated_images,␣
↪training=True)
                #calculate generator loss
                g_loss = self.g_loss_fn(prediction_fake)

            #calculate gradients and apply them using Adam
            g_gradients = gt.gradient(g_loss, self.generator.
↪trainable_variables)
            self.g_optimizer.apply_gradients(zip(g_gradients, self.generator.
↪trainable_variables))

        return {"d_loss": d_loss, "g_loss": g_loss}
```

[ ]:
```
# define Loss functions
cross_entropy = keras.losses.BinaryCrossentropy(from_logits=True)
def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
```

```
    return total_loss

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)

#initalize optimizers
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

[ ]:
```
# Enable Checkpoint saving if training gets interrupted
checkpoint_dir = 'celebGAN/training_checkpoints'
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                                 discriminator_optimizer=discriminator_optimizer,
                                 generator=generator,
                                 discriminator=discriminator)
manager = tf.train.CheckpointManager(checkpoint, checkpoint_dir, max_to_keep=5)
```

[ ]:
```
log_dir = "celebGAN/logs/fit/" + datetime.datetime.now().
 ↪strftime("%Y%m%d-%H%M%S")
img_log_dir = "celebGAN/logs/images/" + datetime.datetime.now().
 ↪strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir,␣
 ↪histogram_freq=1)

file_writer = tf.summary.create_file_writer(img_log_dir)

# Method converting matplotlib figures to images usable by tensorboard
# Source https://www.tensorflow.org/tensorboard/image_summaries
def plot_to_image(figure):
    """Converts the matplotlib plot specified by 'figure' to a PNG image and
    returns it. The supplied figure is closed and inaccessible after this call.
 ↪"""
    # Save the plot to a PNG in memory.
    buf = io.BytesIO()
    plt.savefig(buf, format='png')
    # Closing the figure prevents it from being displayed directly inside
    # the notebook.
    plt.close(figure)
    buf.seek(0)
    # Convert PNG buffer to TF image
    image = tf.image.decode_png(buf.getvalue(), channels=4)
    # Add the batch dimension
    image = tf.expand_dims(image, 0)
    return image

# Save Image every epoch
class GANMonitor(keras.callbacks.Callback):
```

```
    def __init__(self, num_img=16, latent_dim=100, start_epoch=0, seed=None):
        self.num_img = num_img
        if not seed == None:
            self.seed = seed
        else:
            self.seed = tf.random.normal(shape=(num_img, latent_dim))
        self.start_epoch = start_epoch

    def on_epoch_end(self, epoch, logs=None):
        generated_images = self.model.generator(self.seed, training=False)
        generated_images = (generated_images * 127.5) + 127.5
        generated_images = generated_images.numpy()

        fig = plt.figure(figsize=(4, 4))

        for i in range(generated_images.shape[0]):
            plt.subplot(4, 4, i+1)
            plt.imshow(generated_images[i, :, :, :].astype("int32"))
            plt.axis('off')

        plt.savefig(os.path.join("celebGAN/", "images/",'image_at_epoch_{:04d}.
 ↪png'.format(self.start_epoch+epoch)))
        with file_writer.as_default():
            tf.summary.image("Output", plot_to_image(fig), step=epoch)

# Save Checkpoint every 2 epochs
class GANSaver(keras.callbacks.Callback):
    def __init__(self, manager, num_epochs=15):
        self.num_epochs = num_epochs
        self.manager = manager

    def on_epoch_end(self, epoch, logs=None):
        if (epoch + 1) % self.num_epochs == 0:
            self.manager.save()


ckp = GANSaver(manager, 2)
```

[ ]:
```
gan = GAN(discriminator, generator, latent_dim=128, disc_extra_steps=1)
gan.compile(discriminator_optimizer, generator_optimizer, discriminator_loss,␣
 ↪generator_loss)
```

[ ]:
```
#restore latest state of training
if manager.latest_checkpoint:
    checkpoint.restore(manager.latest_checkpoint)
    latest_epoch = int(manager.latest_checkpoint.split('-')[1])
    last_epoch = latest_epoch * 2
```

```
    print ('Latest checkpoint of epoch {} restored!!'.format(last_epoch))
else:
    last_epoch = 0
    print ('No latest checkpoint found!')
#initialize image saver with start epoch variable to keep existing images after␣
↪restart
ick = GANMonitor(num_img=16, latent_dim=128, start_epoch=last_epoch)

#train model on the dataset for 100 epochs
%tensorboard --logdir celebGAN/logs
#train model on the dataset for 100 epochs
gan.fit(dataset, epochs=100, batch_size=256, callbacks=[ick, ckp,␣
↪tensorboard_callback])
```

[ ]:
```
# save model to harddrive
manager.save()
# store seed in variable to keep faces consistent when rerun
seed = ick.seed
```

[ ]:
```
#generate 100 sample images in batches
images = 25
predictions = np.empty([100,64,64,3])
for i in range(4):
    seed = tf.random.normal([images, 128])
    label_seed = np.random.randint(0,2, images)
    pred = generator(seed, training=False).numpy()
    predictions[25*i:25*(i+1), :, :, :] = pred

print(predictions.shape)
figsize = 10
fig = plt.figure(figsize=(figsize, figsize))
for i in range(predictions.shape[0]):
    plt.subplot(figsize, figsize, i+1)
    plt.imshow((predictions[i, :, :, :]*127.5+127.5).astype("int32"))
    plt.axis('off')
plt.show()
```

[ ]:
```
# output the Generator and Discriminator as Image
tf.keras.utils.plot_model(generator, "celebGAN/Generator.png", show_shapes=True)
tf.keras.utils.plot_model(discriminator, "celebGAN/Discriminator.png",␣
↪show_shapes=True)
```

[ ]:
```
# save Generator as h5 model to use in e.g. matlab
generator.save("celebGAN/generator_model/generator.h5")
```

celebGAN2

November 7, 2022

[ ]:
```
# basic imports
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

from tensorflow.keras.datasets import mnist

import numpy as np
import matplotlib.pyplot as plt

import os
import io

import datetime

%load_ext tensorboard
```

[ ]:
```
# Set base Parameters
IMAGE_HEIGHT = 64
IMAGE_WIDTH = 64
BATCH_SIZE = 256
```

[ ]:
```
# Load Dataset from storage
AUTOTUNE = tf.data.AUTOTUNE
dataset = tf.keras.utils.image_dataset_from_directory("dataset/preprocessed",
  image_size=(IMAGE_HEIGHT, IMAGE_WIDTH),
  batch_size=BATCH_SIZE)
normalization_layer = tf.keras.layers.Rescaling(1./127.5, offset=-1)
dataset = dataset.map(lambda x, y: (normalization_layer(x), y))
```

[ ]:
```
# Define generator model
def make_generator(latent_vector_shape, dense_shape):
    latent_input = layers.Input(shape=latent_vector_shape)
    gen = layers.Dense(4*4*dense_shape)(latent_input)
    gen = layers.ReLU()(gen)
    gen = layers.Reshape((4,4,dense_shape))(gen)
    gen = layers.Dropout(0.2)(gen)
```

```
    gen = layers.Conv2DTranspose(dense_shape, (2, 2), 2, use_bias=False)(gen)
    gen = layers.BatchNormalization()(gen)
    gen = layers.LeakyReLU()(gen)
    gen = layers.Dropout(0.25)(gen)

    gen = layers.Conv2DTranspose(dense_shape/2, (2, 2), 2, use_bias=False)(gen)
    gen = layers.BatchNormalization()(gen)
    gen = layers.LeakyReLU()(gen)
    gen = layers.Dropout(0.25)(gen)

    gen = layers.Conv2DTranspose(dense_shape/4, (2, 2), 2, use_bias=False)(gen)
    gen = layers.BatchNormalization()(gen)
    gen = layers.LeakyReLU()(gen)
    gen = layers.Dropout(0.25)(gen)

    gen = layers.Conv2DTranspose(dense_shape/8, (2, 2), 2, use_bias=False)(gen)
    gen = layers.BatchNormalization()(gen)
    gen = layers.LeakyReLU()(gen)

    out = layers.Conv2D(3, (4, 4), strides=(1,1), padding="same",␣
↪activation='tanh')(gen)

    model: keras.Model = keras.Model(latent_input, out)
    print(model.output_shape)
    assert model.output_shape == (None, 64, 64, 3)
    return model

generator = make_generator(128, 256)
generator.summary()
```

[ ]:
```
#define discriminator model
def make_discriminator(input_shape):
    image_input = layers.Input(input_shape)

    disc = layers.Conv2D(8, (3, 3))(image_input)
    disc = layers.AveragePooling2D()(disc)
    disc = layers.BatchNormalization()(disc)
    disc = layers.LeakyReLU(alpha=0.02)(disc)


    disc = layers.Conv2D(16, (3, 3))(disc)
    disc = layers.AveragePooling2D()(disc)
    disc = layers.LeakyReLU(alpha=0.02)(disc)
    disc = layers.Dropout(0.3)(disc)

    disc = layers.Conv2D(32, (3, 3))(disc)
```

```
    disc = layers.AveragePooling2D()(disc)
    disc = layers.LeakyReLU(alpha=0.02)(disc)

    disc = layers.Flatten()(disc)
    disc = layers.Dropout(0.3)(disc)
    disc = layers.Dense(32)(disc)
    out = layers.Dense(1)(disc)

    model = keras.Model(image_input, out)

    return model

discriminator = make_discriminator((64, 64, 3))
discriminator.summary()
```

[ ]:
```
# define GAN model
class GAN(keras.Model):
    def __init__(self, discriminator, generator, latent_dim=128,␣
 ↪disc_extra_steps=3):
        super(GAN, self).__init__()
        self.discriminator = discriminator
        self.generator = generator
        self.latent_dim = latent_dim
        self.d_steps = disc_extra_steps

    def compile(self, d_optimizer, g_optimizer, d_loss_fn, g_loss_fn):
        super(GAN, self).compile()
        self.d_optimizer = d_optimizer
        self.g_optimizer = g_optimizer
        self.d_loss_fn = d_loss_fn
        self.g_loss_fn = g_loss_fn

    def train_step(self, data):
        images, _ = data
        #calculate bacth size of current batch
        batch_size = tf.shape(images)[0]

        for i in range(self.d_steps):
            #generate new latent vector
            latent_vector = tf.random.normal(shape=(batch_size, self.
 ↪latent_dim))
            with tf.GradientTape() as gt:
                # generate and predict images while being observed by gradient␣
 ↪tape
                # this allows backpropagation and automatic taking of the␣
 ↪derivative
                generated_images = self.generator(latent_vector, training=True)
```

```
                prediction_fake = self.discriminator(generated_images,␣
↪training=True)

                #flip images randomly to inttroduce variety
                flipped_images = tf.image.random_flip_left_right(images)
                prediction_real = self.discriminator(flipped_images,␣
↪training=True)

                #calculate discriminator loss
                d_loss = self.d_loss_fn(prediction_real, prediction_fake)
            #calculate discriminator gradients
            d_gradients = gt.gradient(d_loss, self.discriminator.
↪trainable_variables)
            #apply gradients using Adam
            self.d_optimizer.apply_gradients(zip(d_gradients, self.
↪discriminator.trainable_variables))

            #generate new latent vector for generator training
            latent_vector = tf.random.normal(shape=(batch_size, self.
↪latent_dim))
            with tf.GradientTape() as gt:
                # generate and predict images while being observed by gradient␣
↪tape
                # this allows backpropagation and automatic taking of the␣
↪derivative
                generated_images = self.generator(latent_vector, training=True)
                prediction_fake = self.discriminator(generated_images,␣
↪training=True)
                #calculate generator loss
                g_loss = self.g_loss_fn(prediction_fake)

            #calculate gradients and apply them using Adam
            g_gradients = gt.gradient(g_loss, self.generator.
↪trainable_variables)
            self.g_optimizer.apply_gradients(zip(g_gradients, self.generator.
↪trainable_variables))

        return {"d_loss": d_loss, "g_loss": g_loss}
```

[ ]:

```
# define Loss functions
cross_entropy = keras.losses.BinaryCrossentropy(from_logits=True)
def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss
```

```
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)

#initalize optimizers
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

[ ]:
```
# Enable Checkpoint saving if training gets interrupted
checkpoint_dir = 'celebGAN2/training_checkpoints'
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                                 discriminator_optimizer=discriminator_optimizer,
                                 generator=generator,
                                 discriminator=discriminator)
manager = tf.train.CheckpointManager(checkpoint, checkpoint_dir, max_to_keep=5)
```

[ ]:
```
log_dir = "celebGAN2/logs/fit/" + datetime.datetime.now().
 ↪strftime("%Y%m%d-%H%M%S")
img_log_dir = "celebGAN2/logs/images/" + datetime.datetime.now().
 ↪strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir,␣
 ↪histogram_freq=1)

file_writer = tf.summary.create_file_writer(img_log_dir)

# Method converting matplotlib figures to images usable by tensorboard
# Source https://www.tensorflow.org/tensorboard/image_summaries
def plot_to_image(figure):
    """Converts the matplotlib plot specified by 'figure' to a PNG image and
    returns it. The supplied figure is closed and inaccessible after this call.
 ↪"""
    # Save the plot to a PNG in memory.
    buf = io.BytesIO()
    plt.savefig(buf, format='png')
    # Closing the figure prevents it from being displayed directly inside
    # the notebook.
    plt.close(figure)
    buf.seek(0)
    # Convert PNG buffer to TF image
    image = tf.image.decode_png(buf.getvalue(), channels=4)
    # Add the batch dimension
    image = tf.expand_dims(image, 0)
    return image

# Save Image every epoch
class GANMonitor(keras.callbacks.Callback):
    def __init__(self, num_img=16, latent_dim=100, start_epoch=0, seed=None):
```

```
        self.num_img = num_img
        if not seed == None:
            self.seed = seed
        else:
            self.seed = tf.random.normal(shape=(num_img, latent_dim))
        self.start_epoch = start_epoch

    def on_epoch_end(self, epoch, logs=None):
        generated_images = self.model.generator(self.seed, training=False)
        generated_images = (generated_images * 127.5) + 127.5
        generated_images = generated_images.numpy()

        fig = plt.figure(figsize=(4, 4))

        #generate a subplot
        for i in range(generated_images.shape[0]):
            plt.subplot(4, 4, i+1)
            plt.imshow(generated_images[i, :, :, :].astype("int32"))
            plt.axis('off')

        #save to harddrive
        plt.savefig(os.path.join("celebGAN2/", "images/",'image_at_epoch_{:04d}.
 ↪png'.format(self.start_epoch+epoch)))
        with file_writer.as_default():
            tf.summary.image("Output", plot_to_image(fig), step=epoch)

# Save Checkpoint every 2 epochs
class GANSaver(keras.callbacks.Callback):
    def __init__(self, manager, num_epochs=15):
        self.num_epochs = num_epochs
        self.manager = manager

    def on_epoch_end(self, epoch, logs=None):
        if (epoch + 1) % self.num_epochs == 0:
            self.manager.save()


ckp = GANSaver(manager, 2)
```

[ ]:
```
gan = GAN(discriminator, generator, latent_dim=128, disc_extra_steps=1)
gan.compile(discriminator_optimizer, generator_optimizer, discriminator_loss,␣
 ↪generator_loss)
```

[ ]:
```
#restore latest state of training
if manager.latest_checkpoint:
    checkpoint.restore(manager.latest_checkpoint)
    latest_epoch = int(manager.latest_checkpoint.split('-')[1])
```

```
    last_epoch = latest_epoch * 2
    print ('Latest checkpoint of epoch {} restored!!'.format(last_epoch))
else:
    last_epoch = 0
    print ('No latest checkpoint found!')
#initialize image saver with start epoch variable to keep existing images after␣
 ↪restart
ick = GANMonitor(num_img=16, latent_dim=128, start_epoch=last_epoch)

#start tensorboard
%tensorboard --logdir celebGAN2/logs
#train model on the dataset for 100 epochs
gan.fit(dataset, epochs=100, batch_size=256, callbacks=[ick, ckp,␣
 ↪tensorboard_callback])
```

[ ]:
```
# save model to harddrive
manager.save()
# store seed in variable to keep faces consistent when rerun
seed = ick.seed
```

[ ]:
```
#generate 100 sample images in batches
images = 25
predictions = np.empty([100,64,64,3])
for i in range(4):
    seed = tf.random.normal([images, 128])
    label_seed = np.random.randint(0,2, images)
    pred = generator(seed, training=False).numpy()
    predictions[25*i:25*(i+1), :, :, :] = pred

print(predictions.shape)
figsize = 10
fig = plt.figure(figsize=(figsize, figsize))
for i in range(predictions.shape[0]):
    plt.subplot(figsize, figsize, i+1)
    plt.imshow((predictions[i, :, :, :]*127.5+127.5).astype("int32"))
    plt.axis('off')
plt.show()
```

[ ]:
```
# output the Generator and Discriminator as Image
tf.keras.utils.plot_model(generator, "celebGAN/Generator.png", show_shapes=True)
tf.keras.utils.plot_model(discriminator, "celebGAN/Discriminator.png",␣
 ↪show_shapes=True)
```

[ ]:
```
# save Generator as h5 model to use in e.g. matlab
generator.save("celebGAN/generator_model/generator.h5")
```

DCGAN

November 7, 2022

[ ]:
```
# importing
import IPython
from IPython.display import clear_output

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import tensorflow_datasets as tfds
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import pydot

import kaggle
import os
import glob
import shutil
import time
print("Num GPUs Available: ", len(tf.config.experimental.
 ↪list_physical_devices('GPU')))
print(tf.test.gpu_device_name())
tf.get_logger().setLevel('INFO')
#test GPUs
gpus = tf.config.experimental.list_physical_devices('GPU')
if gpus:
  try:
    tf.config.experimental.set_virtual_device_configuration(
        gpus[0],[tf.config.experimental.
 ↪VirtualDeviceConfiguration(memory_limit=5120)])
  except RuntimeError as e:
    print(e)

%load_ext tensorboard
!rm -rf ./cDCGAN/logs/

#base directory for execution
```

```
BASE_DIR = "/home/simon/Documents/W-Seminar/"
```

[ ]:
```
# set parameters
IMAGE_HEIGHT = 64
IMAGE_WIDTH = 64
BATCH_SIZE = 256

#initialize optimizers
generator_optimizer = keras.optimizers.Adam(2e-4)
discriminator_optimizer = keras.optimizers.Adam(2e-4)

#initialize loss
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
```

[ ]:
```
# load dataset
AUTOTUNE = tf.data.AUTOTUNE
dataset = tf.keras.utils.image_dataset_from_directory(
  os.path.join(BASE_DIR, "dataset/preprocessed"),
  image_size=(IMAGE_HEIGHT, IMAGE_WIDTH),
  batch_size=BATCH_SIZE)
class_names = dataset.class_names
normalization_layer = tf.keras.layers.Rescaling(1./127.5, offset=-1)
dataset = dataset.map(lambda x, y: (normalization_layer(x), y))
```

[ ]:
```
plt.figure(figsize=(10, 10))
for images, labels in dataset.take(1).cache():
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow((images[i].numpy()*127.5+127.5).astype("uint8"))
        plt.title(class_names[labels[i]])
        plt.axis("off")
```

[ ]:
```
#build generator model
def make_generator_model():
    # latent vector input
    gen_input = layers.Input((100,), name="latent_vector")

    gen = layers.Dense(8*8*100, use_bias=False)(gen_input)
    gen = layers.BatchNormalization()(gen)
    gen = layers.LeakyReLU(alpha=0.2)(gen)
    gen = layers.Reshape((8, 8, 100))(gen)

    # label input
    label_input = layers.Input((1,), name="label")
    label = layers.Embedding(2, 50)(label_input)
    label = layers.Dense(8*8)(label)
    label = layers.Reshape((8,8,1))(label)
```

```
    # convert two inputs to one tensor
    gen = layers.Concatenate()([gen, label])

    gen = layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same',␣
↪use_bias=False)(gen)
    gen = layers.BatchNormalization()(gen)
    gen = layers.LeakyReLU(alpha=0.2)(gen)
    gen = layers.Dropout(0.5)(gen)

    gen = layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same',␣
↪use_bias=False)(gen)
    gen = layers.BatchNormalization()(gen)
    gen = layers.LeakyReLU(alpha=0.2)(gen)

    gen = layers.Conv2DTranspose(32, (5, 5), strides=(2, 2), padding='same',␣
↪use_bias=False)(gen)
    gen = layers.BatchNormalization()(gen)
    gen = layers.LeakyReLU(alpha=0.2)(gen)
    gen = layers.Dropout(0.5)(gen)

    gen = layers.Conv2DTranspose(16, (5, 5), strides=(2, 2), padding='same',␣
↪use_bias=False)(gen)
    gen = layers.BatchNormalization()(gen)
    gen = layers.LeakyReLU(alpha=0.2)(gen)

    out_layer = layers.Conv2D(3, (5, 5), padding="same", use_bias=False, ␣
↪activation='tanh')(gen)

    model = keras.Model([gen_input, label_input], out_layer)
    assert out_layer.shape == (None, 64, 64,3)

    return model
```

[ ]:
```
# get summary
generator = make_generator_model()
generator.summary()
tf.keras.utils.plot_model(generator, "cDCGAN/cDCGenerator.png",␣
↪show_shapes=True)
```

[ ]:
```
#define generator
def make_discriminator_model(optimizer):
    # label input
    label_input = layers.Input((1,), name="label")
    label = layers.Embedding(2, 50)(label_input)
    label = layers.Dense(64*64)(label)
```

```python
    label = layers.Reshape((64,64,1))(label)

    #image input
    image_input = layers.Input((64, 64, 3), name="image")

    # convert two inputs to one tensor
    disc = layers.Concatenate()([image_input, label])

    disc = layers.Conv2D(64, (5, 5), strides=(2,2), padding='same')(disc)
    disc = layers.LeakyReLU(alpha=0.2)(disc)
    disc = layers.Dropout(0.3)(disc)

    disc = layers.Conv2D(128, (5, 5), strides=(2,2), padding='same')(disc)
    disc = layers.LeakyReLU(alpha=0.2)(disc)
    disc = layers.Dropout(0.3)(disc)

    disc = layers.Flatten()(disc)
    disc = layers.Dropout(0.3)(disc)
    output_layer = layers.Dense(1, activation='leaky_relu')(disc)

    model = keras.Model([image_input, label_input], output_layer)
    model.compile(optimizer, loss=keras.losses.
↪BinaryCrossentropy(from_logits=True), metrics=["acc"])
    return model
```

[ ]:
```python
discriminator = make_discriminator_model(discriminator_optimizer)
generator.summary()
tf.keras.utils.plot_model(generator, "cDCGAN/cDCDiscriminator.png",␣
↪show_shapes=True)
```

[ ]:
```python

```

[ ]:
```python
# make gan model
def make_gan_model(discrimiator: keras.Model, generator: keras.Model):
    discrimiator.trainable = False
    gen_noise_input, gen_label_input = generator.input
    gen_output = generator.output

    gan_output = discrimiator([gen_output, gen_label_input])
    model = keras.Model([gen_noise_input, gen_label_input], gan_output)
    model.compile(generator_optimizer, loss=cross_entropy)
    return model
```

[ ]:
```python
gan = make_gan_model(discriminator, generator)
gan.summary()
tf.keras.utils.plot_model(gan, "cDCGAN/cDCGAN.png", show_shapes=True)
```

[ ]:
```
# seed for image generation
seed = tf.random.normal([16, 100])
label_seed = np.random.randint(0,2, 16)
```

[ ]:
```
# define checkpoint constants
checkpoint_dir = 'cDCGAN/training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                                 ␣
 ↪discriminator_optimizer=discriminator_optimizer,
                                 generator=generator,
                                 discriminator=discriminator,)
manager = tf.train.CheckpointManager(checkpoint, checkpoint_dir, max_to_keep=3)
```

[ ]:
```
# noise layer for reducing overfitting
noise_layer = layers.GaussianNoise(0.2)

def train_step(images, labels, candidate):
    batch_size = images.shape[0]
    noise = tf.random.normal([batch_size, 100])
    fake_labels = np.random.randint(0,2, batch_size)

    generated_images = generator.predict({"latent_vector":noise, "label":
 ↪fake_labels}, verbose=0)

    noisy_generated_images = noise_layer(generated_images, training=False)
    flipped_images = tf.image.random_flip_left_right(images)
    noisy_images = noise_layer(flipped_images, training=False)


    disc_loss_fake = 0.0
    disc_loss_real = 0.0
    gen_loss = 0.0

    # logic to train either generator, discriminator or both
    if candidate == 0:
        disc_loss_fake, _ = discriminator.
 ↪train_on_batch([noisy_generated_images, fake_labels], tf.
 ↪zeros([batch_size,1]))
        disc_loss_real, _ = discriminator.train_on_batch([noisy_images,␣
 ↪labels], tf.ones([batch_size,1]))
    elif candidate == 1:
        gen_loss = gan.train_on_batch([noise, fake_labels], tf.
 ↪ones([batch_size,1]))
    else:
```

```
        disc_loss_fake, _ = discriminator.
↪train_on_batch([noisy_generated_images, fake_labels], tf.
↪zeros([batch_size,1]))
        disc_loss_real, _ = discriminator.train_on_batch([noisy_images,␣
↪labels], tf.ones([batch_size,1]))
        gen_loss = gan.train_on_batch([noise, fake_labels], tf.
↪ones([batch_size,1]))

    return gen_loss, disc_loss_real, disc_loss_real
```

[ ]:
```
BATCHES = int(214692 / BATCH_SIZE)
def train(dataset, epochs, start_epoch=0, save_checkpoints=True):
    for epoch in range(start_epoch, epochs + start_epoch):
        start = time.time()
        batch = 1
        train_condition = np.random.random([1,1])
        #choose which model should be trained
        if train_condition > 0 and train_condition < 0.25:
            candidate = 0
        elif train_condition > 0.25 and train_condition < 0.44:
            candidate = 1
        else:
            candidate = 2

        for image_batch, labels in dataset:
            gen_loss, disc_loss_real, disc_loss_fake = train_step(image_batch,␣
↪labels, candidate)
            print(f'{batch}/{BATCHES}: d_real={disc_loss_real}␣
↪d_fake={disc_loss_fake} gan={gen_loss}', end="\r")
            batch+=1

        print(f'd_real={disc_loss_real} d_fake={disc_loss_fake} gan={gen_loss}')
        print("Time for epoch {}: {}".format(epoch, time.time()-start))

        #generate images each epoch
        generate_and_save_images(generator,epoch,seed, label_seed)

        clear_output(wait=True)
        # Save the model every 15 epochs
        if (epoch + 1) % 15 == 0 and save_checkpoints:
            manager.save()
            save_latest_epoch(epoch)


def generate_and_save_images(model, epoch, test_input, labels):
    predictions = model([test_input, labels], training=False).numpy()
```

```
    fig = plt.figure(figsize=(4, 4))

    for i in range(predictions.shape[0]):
        plt.subplot(4, 4, i+1)
        plt.imshow((predictions[i, :, :, :]*127.5+127.5).astype("int32"))
        plt.axis('off')

    plt.savefig(os.path.join(checkpoint_dir, "images/",'image_at_epoch_{:04d}.
 ↪png'.format(epoch)))
    plt.show()
```

[ ]:
```
# load latest training checkpoint and get latest epoch
if manager.latest_checkpoint:
    checkpoint.restore(manager.latest_checkpoint)
    latest_epoch = int(manager.latest_checkpoint.split('-')[1])
    last_epoch = latest_epoch * 15
    print ('Latest checkpoint of epoch {} restored!!'.format(last_epoch))
else:
    last_epoch = 0
```

[ ]:
```

```

[ ]:
```
# enable tensorboard for logging
LOG_DIR = "cDCGAN/logs/fit"
tb_callback = tf.keras.callbacks.TensorBoard(os.path.join(LOG_DIR, "gen"))
tb_callback.set_model(generator)
tb_disc_callback = tf.keras.callbacks.TensorBoard(os.path.join(LOG_DIR, "disc"))
tb_callback.set_model(discriminator)

%tensorboard --logdir cDCGAN/logs

#train GAN for 5000 epochs
train(dataset.prefetch(AUTOTUNE), 5000, last_epoch)
```

DS Creator

November 7, 2022

[ ]:
```
from ipywidgets import IntProgress
from IPython.display import display

import tensorflow as tf
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import kaggle
import os
import glob
import shutil
print("Num GPUs Available: ", len(tf.config.experimental.
↪list_physical_devices('GPU')))


BASE_DIR = os.getcwd()
```

[ ]:
```
# Download CelebA Dataset from Kaggle
!kaggle  datasets download -d jessicali9530/celeba-dataset
!unzip -d dataset -u -q celeba-dataset.zip
```

[ ]:
```
#Sort images into categories
df = pd.read_csv('dataset/list_attr_celeba.csv')
df.head()
is_male_df = df["Male"]

ds_PATH = "dataset/img_align_celeba/img_align_celeba/"

for file in glob.glob(os.path.join(ds_PATH, "*.jpg")):
    file_number = file.strip(ds_PATH).strip(".jpg")
    if is_male_df[int(file_number)-1] == 1:
        shutil.copy2(file, os.path.join(BASE_DIR, "dataset/preprocessed/male",␣
↪file_number+".jpg"))
    else:
        shutil.copy2(file, os.path.join(BASE_DIR, "dataset/preprocessed/
↪female", file_number+".jpg"))
```

```
[ ]: IMAGE_WIDTH = 64
     IMAGE_HEIGHT = 64
     BATCH_SIZE = 128
```

```
[ ]: # generate and test dataset
     AUTOTUNE = tf.data.AUTOTUNE
     train_ds = tf.keras.utils.image_dataset_from_directory(
         os.path.join(BASE_DIR, "dataset/preprocessed"),
         image_size=(IMAGE_HEIGHT, IMAGE_WIDTH),
         batch_size=BATCH_SIZE)
     train_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

```
[ ]: # plot example images
     class_names = train_ds.class_names
     plt.figure(figsize=(10, 10))
     for images, labels in train_ds.take(1):
         for i in range(9):
             ax = plt.subplot(3, 3, i + 1)
             plt.imshow(images[i].numpy().astype("uint8"))
             plt.title(class_names[labels[i]])
             plt.axis("off")
```

```
%  load dataset
[XTrain,YTrain,anglesTrain] = digitTrain4DArrayData;

% Hyperparameters
num_epochs = 100;
minibatchsize = 256;

learnRate = 0.0002;
gradientDecayFactor = 0.5;
squaredGradientDecayFactor = 0.999;

%Define Generator
layers = [
    featureInputLayer(128,"Name","sequence")
    fullyConnectedLayer(12544,"Name","fc")
    batchNormalizationLayer("Name","batchnorm")
    leakyReluLayer(0.01,"Name","leakyrelu")

    functionLayer(@(x) dlarray(reshape(x, 7,7,256, []), "SSCB"), ...
    Formattable=true, Acceleratable=true)

    transposedConv2dLayer([5 5],128,"Name","transposed-conv", ...
    "BiasLearnRateFactor",0,"Cropping","same")
    batchNormalizationLayer("Name","batchnorm_1")
    leakyReluLayer(0.01,"Name","leakyrelu_1")
    transposedConv2dLayer([5 5],64,"Name","transposed-conv_1", ...
    "BiasLearnRateFactor",0,"Cropping","same","Stride",[2 2])
    batchNormalizationLayer("Name","batchnorm_2")
    leakyReluLayer(0.01,"Name","leakyrelu_2")
    transposedConv2dLayer([5 5],1,"Name","transposed-conv_2", ...
    "BiasLearnRateFactor",0,"Cropping","same","Stride",[2 2])
    sigmoidLayer()
    ];
generator = dlnetwork(layers)

% define Discriminator
disc_layers = [
    imageInputLayer([28 28 1],"Name","imageinput", ...
    "Normalization","none")

    convolution2dLayer([5 5],4,"Name","conv", ...
    "Padding","same","Stride",[2 2])
    leakyReluLayer(0.01,"Name","leakyrelu")
    batchNormalizationLayer()
    dropoutLayer(0.5,"Name","dropout")

    convolution2dLayer([5 5],8,"Name","conv_1", ...
    "Padding","same","Stride",[2 2])
    leakyReluLayer(0.01,"Name","leakyrelu_1")
    batchNormalizationLayer()
```

```
    dropoutLayer(0.5,"Name","dropout_1")

    flattenLayer("Name","flatten")

    fullyConnectedLayer(1)
    sigmoidLayer("Name","sigmoid")];
discriminator = dlnetwork(disc_layers)

latent_vector = dlarray(randn([128, 1], "single"), "CB");
pred = predict(generator, latent_vector)*255;
image(extractdata(pred))
title("Untrained model Output")

x = XTrain(:, :, 1);
image(x*255)
title("Sample Out of the dataset")

XTrain_datastore = arrayDatastore(XTrain, ...
    "IterationDimension",4)

mbq = minibatchqueue(XTrain_datastore, ...
    minibatchsize=minibatchsize, ...
    PartialMiniBatch="discard", ...
    MiniBatchFormat="SSBC");

% parameters for training
iteration = 0;
averageGradDisc = [];
averageSqGradDisc = [];

averageGradGen = [];
averageSqGradGen = [];

val_latent_vector = dlarray(randn([128, 16]), "CB");

if canUseGPU
    val_latent_vector = gpuArray(val_latent_vector);
end

%bar = waitbar(0,"training...")

for epoch = 1:num_epochs
    shuffle(mbq);

    epochloss_d = zeros([minibatchsize,1]);
    epochloss_g = zeros([minibatchsize,1]);

    while mbq.hasdata
        iteration = iteration + 1;
        data = next(mbq);
```

```
        % generate training Latent vector
        latent_vactor = dlarray(randn([128,minibatchsize]), "CB");
        if canUseGPU
            latent_vactor = gpuArray(latent_vactor);
        end
        fake_images = forward(generator, latent_vactor);

        % calulate gradients
        [lossD,gradientsD] = dlfeval(@modelLossD, discriminator, ...
            data, fake_images);
        %apply discriminator gradients
        [discriminator, averageGradDisc, averageSqGradDisc] = adamupdate(...
            discriminator, ...
            gradientsD, averageGradDisc, ...
            averageSqGradDisc, iteration, ...
            learnRate, gradientDecayFactor, ...
            squaredGradientDecayFactor);

        latent_vactor = dlarray(randn([128,minibatchsize]), "CB");
        % generate training Latent vector
        if canUseGPU
            latent_vactor = gpuArray(latent_vactor);
        end

        [fake_images, stateG] = forward(generator, latent_vactor);
        % calculate Generator gradients
        [lossG,gradientsG] = dlfeval(@modelLossG, generator, ...
            discriminator, fake_images);
        % apply generator gradients
        [generator, averageGradGen, averageSqGradGen] = adamupdate( ...
            generator, ...
            gradientsG, averageGradGen, ...
            averageSqGradGen, iteration, ...
            learnRate, gradientDecayFactor, ...
            squaredGradientDecayFactor);

        generator.State = stateG;
        % save loss
        epochloss_d(epoch) = lossD;
        epochloss_g(epoch) = lossG;
    end
    %generate validation images and print loss
    val_images = extractdata(forward(generator, val_latent_vector)*255);
    fprintf("Discriminator: %0.5f; Generator: %0.5f\n", mean(epochloss_d), ...
        mean(epochloss_g))
    f = figure;
    for i = 1:16
        subplot(4, 4, i)
        image(val_images(:,:,i))
```

```
    end
    %waitbar(epoch/num_epochs, bar, "training...")
end
```

```
function [loss, grad] = modelLossD(net, images, generated_images)
    y_real = forward(net, images);
    loss_real = log(y_real);

    y_fake = forward(net, generated_images);
    loss_fake = log(1-y_fake);

    loss = - mean(loss_real) - mean(loss_fake);
    grad = dlgradient(loss, net.Learnables, RetainData=true);
end

function [loss, grad] = modelLossG(net, discriminator, generated_images)
    y_fake = forward(discriminator, generated_images);
    loss = - mean(log(y_fake));
    grad = dlgradient(loss, net.Learnables);
end
```