Arsen Ignatosyan (s4034538)
Andrew Caruana (s4014049)
Van Nguyen (s3726266)

# Assignment 1: Building MLPs, CNNs and Generative Models with TensorFlow

Introduction to Deep Learning

December 19, 2023

## 1 Task 1

For this part of this assignment we were tasked to explore the MNIST and CIFAR-10 dataset. First we explored MLP implementation and then CNN implementation on MNIST data set. To give a little background information about the MNIST dataset.

The MNIST dataset consists of 28x28 images of handwritten numbers between 0-9. There are about 60 000 labeled images. It is considered or used to be considered as standard benchmark to evaluate machine learning and deep learning algorithms.

### 1.1 Task 1.1

In this task we simply run code provided from repository of keras-team. First we run MLP code which consisted of 5 layers. 3 dense layers and 2 Dropout layers. The MLP model has achieved **0.9821 Test accuracy** and **0.136 Test loss** with batch sizes of 128 and 12 epochs.

For CNN model consisting of 8 layers (convolutional, pooling, dropout, flatten and dense layers) had test accuracy of **0.8464** and test loss of **0.699**.

### 1.2 Tuning MLP for MNIST

Tuning part was divided into parameters and hyper parameters for parameters we tested for different initialization methods, activation functions, optimizers and regularization techniques. For finding the best parameters we simply looped the model through given parameters to find the best performing ones. For reproducibility we set the seed at 42.

Example:

```
activation_methods = ['relu', 'sigmoid', 'tanh', 'linear', 'softmax']

for method in activation_methods:
    model = Sequential()
    model.add(Dense(512, activation=method, input_shape=(784,)))
    model.add(Dropout(0.2))
    model.add(Dense(512, activation=method))
    model.add(Dropout(0.2))
    ...
```

In initialization tuning we found that most of initialization methods perform similarly except Zeros initialization method. All of them are performing around **0.98** range. Best performing one was **RandomUniform** with accuracy of **0.9845**.

For activation methods we tested *relu, sigmoid, tanh, linear and softmax.* Where the results were close for first places **sigmoid** having **0.9829** and **tahn** having *0.9820.*

When tuning optimizesr we tested for SGD, RMSprop and Adam. Where Adam optimizer performed slightly better than default RMSprop at accuracy of **0.9843**.

For regularization methods we opted out for *None, L1, L2 and Dropout.* With Dropout being the best performing one at **0.9828**.

For hyper parameters we decided on learning rate, momentum values, epsilon, nesterov, batch sizes and given optimizers. An for loop where each combination is tested with each other and then top 5 ranking combinations were given at the top of the output. The best 3 performing configurations were:

Table 1: Results for best performing parameters MLP

| Configuration | Test Loss | Test Accuracy |
|---|---|---|
| Initialization method: GlorotUniform | 0.119634 | 0.9824 |
| Activation function: Relu | 0.137991 | 0.9828 |
| Optimizer: Adam | 0.076241 | 0.9843 |
| Regularization technique: Dropout | 0.130384 | 0.9843 |

Table 2: Results for best configuration of hyperparameters

| Optimizer | Learning Rate | Momentum | Epsilon | Nesterov | Batch Size | Test Loss | Test Accuracy |
|---|---|---|---|---|---|---|---|
| SGD | 0.1 | 0 | $1.0 \times 10^{-4}$ | False | 64 | 0.065562 | 0.9795 |
| SGD | 0.1 | 0.9 | $1.0 \times 10^{-4}$ | False | 64 | 0.068581 | 0.9783 |
| SGD | 0.1 | 0 | $1.0 \times 10^{-8}$ | False | 64 | 0.073224 | 0.9772 |

## 1.3 Tuning CNN for MNIST

Unlike in MLP model there was room for improvement as default model had only accuracy of 0.8464. First we dove into finding best initialization method for which we got **HeNormal** with accuracy of **0.8987**.

Similarly to MLP we also tested the same activation functions with **linear** performing the best at the accuracy of **0.8779**.

For different optimizers we tested the same and we have noted the biggest difference in accuracy as all of chosen optimizers improved the accuracy to 0.95+ values. With **Adam** optimizer performing the best at **0.9915**.

Regularization technique that worked the best was **L2** at **0.8767** but the differences between different techniques was negligible.

Table 3: Results for best performing parameters CNN

| Parameter | Test Loss | Test Accuracy |
|---|---|---|
| Initialization method: HeNormal | 0.411512 | 0.8987 |
| Activation function: Linear | 0.477094 | 0.8779 |
| Optimizer: Adam | 0.031210 | 0.9915 |
| Regularization technique: L2 | 0.836787 | 0.8767 |

Testing different hyperparameters was rather computationally heavy but used same methodology as we did for tuning hyperparameters for MLP.

Table 4: Results for the Best Configuration of Hyperparameters

| Optimizer | LR | Momentum | Epsilon | Nesterov | Batch | Regularization | Test Loss | Test Accuracy |
|---|---|---|---|---|---|---|---|---|
| Adadelta | 0.1 | 0.9 | $1.0 \times 10^{-4}$ | False | 64 | Dropout | 0.028825 | 0.9912 |
| Adadelta | 0.1 | 0.9 | $1.0 \times 10^{-4}$ | True | 64 | Dropout | 0.029777 | 0.9910 |
| Adadelta | 0.1 | 0 | $1.0 \times 10^{-8}$ | True | 64 | None | 0.036239 | 0.9899 |

## 1.4 Conclusion on tuning MLP and CNN

We have noticed that not necessarily using or tuning all parameter or parameters for both model make a significant difference. Sometimes only using different optimizers is sufficient enough to improve the model as is the case with Adam optimizer in CNN.

## 1.5 Using best performing configurations on CIFAR-10

The CIFAR-10 dataset consists of 32x32 color images that are categorized into ten different classes. Each image is labeled with one of the following ten categories: airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The dataset contains a total of 60,000 images, with 6,000 images per class.

When applying the best performing parameters on MLP model using CIFAR-10 we achieved accuracy of **0.4697**. When using the best performing the best hyperparameters we achieved 49% accuracy on (SGD, 0.1, 0.9, $1.0 \times 10^{-4}$, True, 64 and SGD, 0.1, 0.9,$1.0 \times 10^{-4}$ , True, 64) When testing for hyperparameter Nesterov set as **false** we get accuracy only of 10% with no improvement.

When applying best performing parameters on CNN model we get initial accuracy of **57%** and for hyperparameters we get only one actually performing model out of three. The model consists of learning rate': 0.1, 'momentum': 0, 'epsilon': 0.0001, 'nesterov': True, 'batch size': 64, 'regularization': 'None'. Accuracy achieved with **12 epoch is 65%** also (SGD). With increased number of epoch to 24 we can further improve the model to **69%** accuracy. The only difference between this model and other two which reached 10% is that the momentum was set to 0.9 and Dropout regularization technique was used.

To explain why the parameters didn't perform as good as in MNIST dataset. The logical explanation would be that the CIFAR-10 dataset is much more complex as it actually has colors unlike MNIST dataset. Also the increased size to 32x32 alters the results. The possibility of over fitting also might occur on simple dataset like MNIST where as CIFAR10 varies much more.

## 2 Task 2

### 2.1 Classification

For the classification task, the data was imported into the notebook, where the labels were each grouped into categories. The amount of categories was not a fixed value, but was experimented with. Furthermore, other parameters were tinkered with such as the amount of epochs, the number of convolutional layers, dense layers, the filter size, the kernel size and the batch size. The results of the best performing models for each category can be seen below.

| Number of Categories | Loss | Common Sense Accuracy (minutes) |
|---|---|---|
| 24 | 0.9045 | 23.94 |
| 48 | 1.6057 | 47.29 |
| 96 | 0.6742 | 10.12 |
| 192 | 0.7533 | 6.14 |
| 360 | 5.9096 | 181.36 |
| 720 | 6.6205 | 179.61 |

Table 5: Best Performing Models per Category

As can be seen in the table, the sweet spot for the amount of categories seems to be around 192, with 96 performing the second best, however, 360 categories performed extremely poorly, as did 720. It was not possible to train the model with this many categories, this could be because at this many categories, the model found it hard to differentiate between one category and another since they could be so close to each other in terms of resemblance. It could also be that there weren't enough samples per category. Since with more categories, one is further splitting the training set for each category, resulting in less of an opportunity to correctly "learn" that category.

Below, please find the model specifications for each of the number of categories. It should be noted that the batch size was set to 128 for 24 categories, and 256 for the rest, and each model was trained over 100 epochs.

Listing 1: 24 Categories Model

```
keras.Input(shape = in_shape),
layers.Conv2D(32, kernel_size=(4,4), activation="relu"),
layers.MaxPooling2D(pool_size=(2,2)),
layers.Conv2D(64, kernel_size=(4,4), activation="relu"),
layers.MaxPooling2D(pool_size=(2,2)),
layers.Conv2D(64, kernel_size=(5,5), activation="relu"),
layers.MaxPooling2D(pool_size=(2,2)),
layers.Flatten(),
layers.Dropout(0.5),
layers.Dense(num_categories, activation="softmax")
```

Listing 2: 48 Categories Model

```
keras.Input(shape = in_shape),
layers.Conv2D(32, kernel_size=(3,3), activation="relu"),
layers.MaxPooling2D(pool_size=(2,2)),
layers.Conv2D(64, kernel_size=(3,3), activation="relu"),
layers.MaxPooling2D(pool_size=(2,2)),
layers.Conv2D(64, kernel_size=(4,4), activation="relu"),
layers.MaxPooling2D(pool_size=(2,2)),
layers.Flatten(),
layers.Dropout(0.5),
layers.Dense(num_categories, activation="softmax")
```

Listing 3: 96-720 Categories Model

```
keras.Input(shape = in_shape),
layers.Conv2D(32, kernel_size=(4,4), activation="relu"),
layers.MaxPooling2D(pool_size=(2,2)),
layers.Conv2D(64, kernel_size=(4,4), activation="relu"),
layers.MaxPooling2D(pool_size=(2,2)),
layers.Conv2D(64, kernel_size=(5,5), activation="relu"),
layers.MaxPooling2D(pool_size=(2,2)),
layers.Flatten(),
layers.Dense(128, activation="relu"),
layers.Dropout(0.5),
layers.Dense(num_categories, activation="softmax")
```

## 2.2 Regression

Following the classification task, we adjusted the model to only have one output neuron and have the model attempt to tell the time using regression. As we were no longer using classification, the model had to be adjusted in several ways.

Firstly, we again experimented with different numbers of convolutional layers, dense layers, filter sizes and kernel sizes to find an optimal model. We quickly noticed that adding more dense layers helped the model quite a bit. Furthermore, the final activation function was changed from softmax to linear, as we no longer want a probability distribution among classes. Lastly, the loss function had to be experimented with as well, since categorical crossentropy is no longer applicable. Some experimentation results can be seen in the table below.

| Loss Function | Loss | Common Sense Accuracy (minutes) |
|---|---|---|
| MAE | 0.2636 | 14.26 |
| MSE | 0.3743 | 11.48 height |

Table 6: Best Performing Models per Category

As can be seen in the table above, using mean squared error yielded slightly better results than mean average error, however, neither of them were able to match the best results using classification. This is despite experimentation and tweaking the model to attempt to find the optimal parameters. The specific model used is listed below:

Listing 4: MSE Model

```
keras.Input(shape = in_shape),
layers.Conv2D(32, kernel_size=(4,4), activation="relu"),
layers.MaxPooling2D(pool_size=(2,2)),
layers.Conv2D(64, kernel_size=(5,5), activation="relu"),
layers.MaxPooling2D(pool_size=(2,2)),
layers.Conv2D(64, kernel_size=(5,5), activation="relu"),
layers.MaxPooling2D(pool_size=(2,2)),
layers.Flatten(),
layers.Dense(256, activation="relu"),
layers.Dense(256, activation="relu"),
layers.Dense(128, activation="relu"),
layers.Dense(128, activation="relu"),
layers.Dense(128, activation="relu"),
layers.Dropout(0.5),
layers.Dense(1, activation="linear")
```

### 2.3 Multi-Head Models

Following the implementation of two convolutional neural networks for both classification and regression, we implemented a multi head model that incorporates them both. Two heads were used in the model, one using classification for the hours, and the other using regression for the minutes.

The parameters for this model were also selected based on the best performing models described earlier. Incidentally, these parameters worked extremely well, and the results of the model can be seen below. (Note that CSA is hereby used to refer to common sense accuracy)

| Loss | CSA (Classification) | CSA (Regression) |
|---|---|---|
| 1.4991 | 0.8518 | 0.6472 |

Table 7: CSA of the Multi-Head Model

This model performed much better than both the classification, and regression individually. One reason as to why this could be the case, since within the images there are two clock hands, having one head for each allows the model to focus on a particular feature within the image. Furthermore, implementing both regression and classification allows the gives the model more variety in terms of how to correctly identify features, meaning that it might take the best aspects of both into consideration.

The model used can be found below:

Listing 5: MSE Model

```
input_layer = Input(shape=in_shape, name="input_layer")
conv1 = layers.Conv2D(32, kernel_size=(4,4), activation="relu", name="conv1")(input_layer)
pool1 = layers.MaxPooling2D(pool_size=(2,2), name="pool1")(conv1)
conv2 = layers.Conv2D(64, kernel_size=(4,4), activation="relu", name="conv2")(pool1)
pool2 = layers.MaxPooling2D(pool_size=(2,2), name="pool2")(conv2)
conv3 = layers.Conv2D(64, kernel_size=(5,5), activation="relu", name="conv3")(pool2)
pool3 = layers.MaxPooling2D(pool_size=(2,2), name="pool3")(conv3)

#Class Head
class_flatten = layers.Flatten(name="class_flatten")(pool3)
class_dense1 = layers.Dense(128, activation="relu", name="class_dense1")(class_flatten)
class_drop = layers.Dropout(0.5, name="class_drop")(class_dense1)
class_out = layers.Dense(num_categories, activation="softmax", name="class_out")(class_drop)

#Reg Head
reg_flatten = layers.Flatten(name="reg_flatten")(pool3)
reg_dense1 = layers.Dense(256, activation="relu", name="reg_dense1")(reg_flatten)
reg_dense2 = layers.Dense(256, activation="relu", name="reg_dense2")(reg_dense1)
reg_dense3 = layers.Dense(128, activation="relu", name="reg_dense3")(reg_dense2)
reg_dense4 = layers.Dense(128, activation="relu", name="reg_dense4")(reg_dense3)
reg_dense5 = layers.Dense(128, activation="relu", name="reg_dense5")(reg_dense4)
reg_drop = layers.Dropout(0.5, name="reg_drop")(reg_dense5)
reg_out = layers.Dense(1, activation="linear", name="reg_out")(reg_drop)

model = Model(inputs=input_layer, outputs=[class_out, reg_out])
```

## 3 Task 3

The goal of the task 3 is to train generative models, more specifically Variational Autoencoders (VAE) and Generative Adversarial Networks (GAN).

Figure 1: CelebA Dataset Sample Images

### 3.1 Dataset

For the task of training generative models the dataset of Large-scale CelebFaces Attributes (CelebA) Dataset [1] was chosen. The dataset contains 202,599 face images of celebrities. The sample of images from the dataset can be seen in Figure 1.

In contrast to widely known and used Flickr-Faces-HQ Dataset (FFHQ), which was also used in the provided Jupyter Notebook, the chosen dataset's images have a diverse set of poses and backgrounds, and do not have the level of uniformity that Flickr dataset has. The chosen dataset can be downloaded from the following Google Drive: https://drive.google.com/drive/folders/0B7EVK8r0v71pTUZsaXdaSnZBZzg?resourcekey=0-rJlzl934LzC-Xp28GeIBzQ. The dataset used by us is the `img_align_celeb.zip` compressed folder. As the authors state, these images are first roughly aligned using similarity transformation according to the two eye locations, and have already passed some processing steps, so this version of the dataset was chosen by us.

### 3.2 Theoretical Background

#### 3.2.1 Variational Autoencoders (VAE)

Variational Autoencoders (VAE) generally have the architecture presented in Figure 2. The architecture of VAE has an encoder and a decoder structure, which are neural networks, and its goal is to capture the underlying probability distribution of the dataset it is trained on and generate new images. The encoder transforms the input into a latent vector, and the decoder takes this latent vector and reconstructs the input image. The encoder takes input data and maps it to a probability distribution in latent space. Instead of producing a fixed encoding, VAE's output parameters of a probability distribution (commonly a Gaussian distribution). Once the encoder produces the parameters of the distribution (mean and standard deviation in the case of a Gaussian distribution), a point is sampled from this distribution. The decoder takes the sampled latent representation and reconstructs it into an image, based on which the error loss is calculated. The training of a VAE involves maximizing the evidence lower bound (ELBO), which is equivalent to minimizing the reconstruction error and the Kullback–Leibler divergence between the learned distribution in the latent space and a prior distribution.

#### 3.2.2 Generative Adversarial Networks (GAN)

Generative Adversarial Networks (GAN) generally have the architecture presented in Figure 3. GANs have two components: generator and discriminator. The generator network takes random noise as input and generates synthetic data. The discriminator network evaluates the compared the generated data from the generator to the real data sample, attempting to distinguish between them. Throughout the training iterations of the model, the discriminator tries to distinguish better between the synthetic data and real data, while the generator tries learning to fool the discriminator and generate images that are really close to real data. GAN uses two losses, the discriminator loss and the generator loss. The loss of discriminator encourages the discriminator to correctly classify real samples as real and generated samples as fake. The loss of generator encourages the generator to generate samples that the discriminator classifies as real.

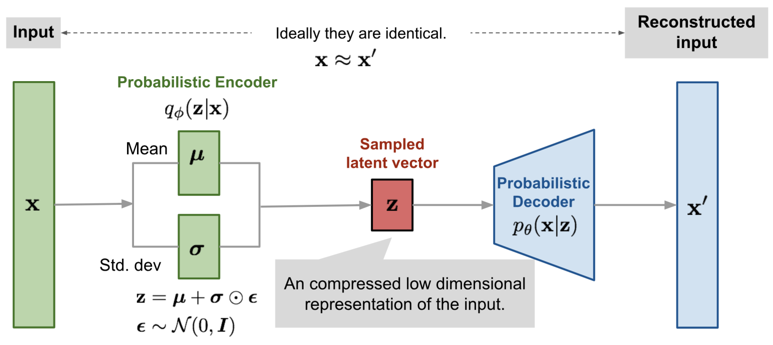

Input
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Reconstructed input
Probabilistic Encoder
$q_\phi(\mathbf{z}|\mathbf{x})$
Mean
Std. dev
$\boldsymbol{\mu}$
$\boldsymbol{\sigma}$
$\mathbf{x}$
Sampled latent vector
$\mathbf{z}$
Probabilistic Decoder
$p_\theta(\mathbf{x}|\mathbf{z})$
$\mathbf{x}'$
$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$
$\boldsymbol{\epsilon} \sim \mathcal{N}(0, \boldsymbol{I})$
An compressed low dimensional representation of the input.

Figure 2: Variational Autoencoder Model Architecture

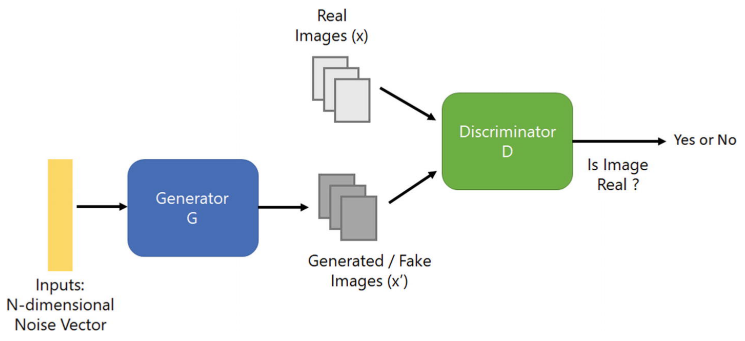


Figure 3: Generative Adversarial Network Architecture

### 3.2.3 Differences of GANs and VAEs

Firstly, VAEs and GANs differ notably in their training approaches. VAEs employ an unsupervised training method, while GANs utilize a supervised technique. During VAEs' training, the objective is to maximize the probability of the generated output with respect to the input, achieving an output from a target distribution by compressing the input into a latent space. Conversely, GANs aim to find a balance between the generator and discriminator in a two-player game, where the generator attempts to fool the discriminator. Additionally, VAEs employ KL-divergence as their loss function, while GANs use two distinct loss functions for the generator and discriminator, respectively.

## 3.3 Experiments and Results

For the task of training generative models on the chosen image dataset of CelebA, VAEs and GANs were trained with multiple setups. Some experiments were conducted on the architecture of these models and dataset experiments were also done.

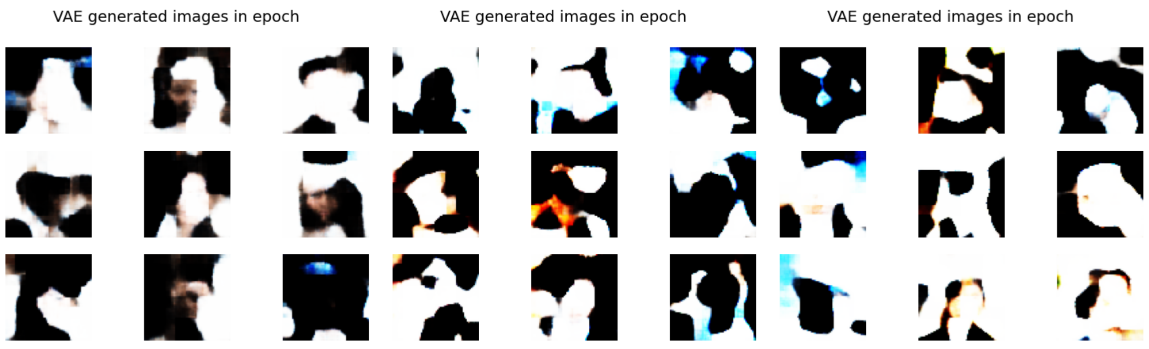
### 3.3.1 Training VAEs

The setups for VAE model architectures can be seen in Table 8. These setups were passed to `build_conv_net(in_shape, out_shape, filters)` and `build_deconv_net(latent_dim, filters)` functions, and the according model architectures were created by these functions.

| Setup # | Number of Filters | Latent Dimension Size |
|---|---|---|
| 1 | 128 | 32 |
| 2 | 128 | 64 |
| 3 | 256 | 64 |

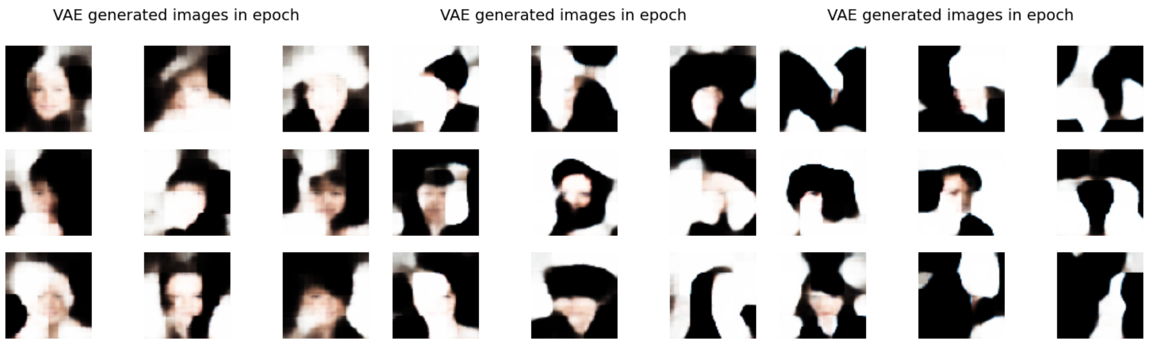
Table 8: Setups of VAE Model Architectures

The model was trained on the complete CelebA image dataset, consisting of 202,599 samples, across various architectures. The training involved 30 epochs, with batches of 128 samples processed in each epoch. Following training, for evaluation of each model's generative capabilities, an array of random samples drawn from a


VAE generated images in epoch
VAE generated images in epoch
VAE generated images in epoch

(a) VAE Setup 1 (b) VAE Setup 2 (c) VAE Setup 3

Figure 4: VAE Generated Images


VAE generated images in epoch
VAE generated images in epoch
VAE generated images in epoch

(a) VAE Setup 1 (b) VAE Setup 2 (c) VAE Setup 3

Figure 5: VAE Generated Images (Trained on Limited Data)

Gaussian distribution. These samples, sized according to the model's latent dimension, were then fed into the model's decoder to generate images based on these randomly generated inputs. The generation of results for each setup can be found in Figure 4. It can be seen that the generated images are note satisfactory. This can be due to the diversity of the chosen dataset, variation in backgrounds and poses specifically. To treat this case, and to test this assumption, we sampled 20,000 images from the original dataset, and used these samples to train the models once again with limited data (presumably with limited variation). The results of the training was also evaluated in the same way as training process with full dataset and the results can be seen in Figure 5. By comparing the generated results from models trained on full data, and models trained on limited data it can be seen that the background parts of images are less colorful for limited data models. Also more facial features are seen with limited data models, which is also a good thing. Maybe the number of trainable parameters is not enough for VAE to capture the training data's probability distribution, so that it can generate synthetic data that can resemble the real one.

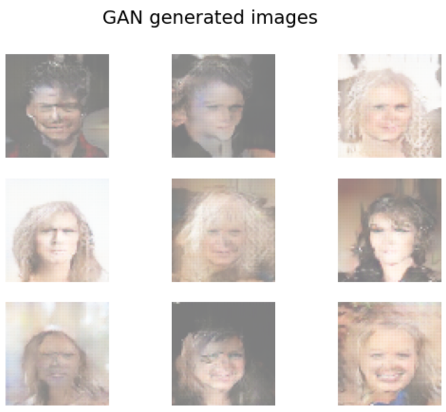
### 3.3.2 Training GANs

The setups for GAN model architectures can be seen in Table 9. These setups were passed to `build_conv_net(in_shape, out_shape, filters)` and `build_deconv_net(latent_dim, filters)` functions, and the according model architectures were created by these functions.

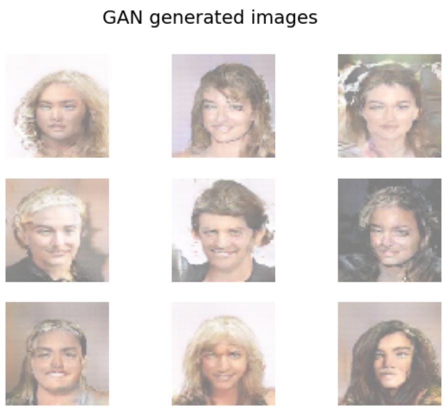
| Setup # | Number of Filters | Latent Dimension Size |
|---|---|---|
| 1 | 64 | 128 |
| 2 | 128 | 128 |

Table 9: Setups of GAN Model Architectures

The differences of the training process of GAN models, compared to VAEs, is that the discriminator was trained with 128 synthetic images generated by generator part of the model, combined with 128 real images, so the

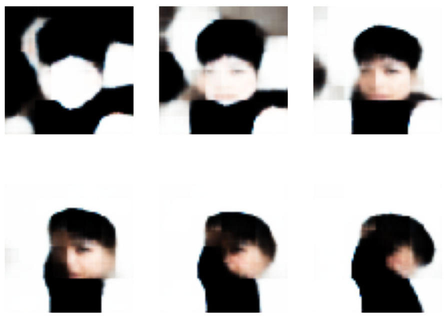


(a) GAN Setup 1

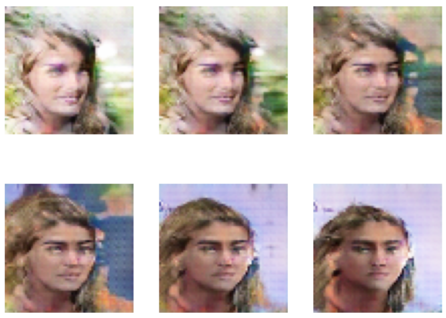


(b) GAN Setup 2

Figure 6: GAN Generated Images



(a) VAE Result



(b) GAN Result

Figure 7: Linear Interpolation of Random Vectors Passed to VAE and GAN

batch size for training becomes 256 images. Also, the epochs chosen for training are 20, as the GAN's training is more computationally expensive then VAE training. The trained model was evaluated the same way, as the VAE models were evaluated, which means that we generated random noise from Gaussian distribution with size of latent dimension, which is passed to the generator network of the model. The generated results of the models can be seen in the Figure 6. From the generated images it can be seen than GAN results outperform VAE results by a mile, as GAN is able to generate human-like features, including different hair styles, different poses, etc. The second setup of GAN gets better outputs as the features are sharper compared to the first setup. Facial features are more visible and better defined.

### 3.3.3 Linear Interpolation Between Two Latent Vectors

In this subsection a linear interpolation will be conducted on two randomly generated vectors, that will be passed to VAE and GAN separately. The function that generates vectors and interpolates them is the following:

Listing 6: Linear Interpolation Function

```
def linear_interpolation(latent_dim, ts):
    latent_vectors = np.random.randn(2, latent_dim)
    latent_vector1, latent_vector2 = latent_vectors[0], latent_vectors[1]

    interpolated_vecs = []
    for t in ts:
        interpolated_vec = np.array([(1 - t) * v1 + t * v2 for v1, v2 in zip(latent_vector1, latent_vector2)])
        interpolated_vecs.append(interpolated_vec)

    return np.array(interpolated_vecs)
```

The VAE with setup 3 trained on the limited data is selected for this subsection, and GAN with setup 2 is selected. The results for both can be seen in Figure 7.

## References

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