BIOLOGICALLY INSPIRED ARTIFICIAL INTELLIGENCE

PROJECT REPORT FROM TOPIC:

FACE GENERATION USING VAE

Lecturer: Dr Inż. Grzegorz Baron

Date: 18.09.2023

Section:
Łukasz Filipek

Short introduction presenting the project topic

The goal of this project is to create and train a Variational Autoencoder type model, and showcase its various applications in the field of face generation. The model is trained on celebA dataset containing various labelled images of celebrities. The model consists of two sub models: *encoder* and *decoder* . The model was expanded with 2 hyper parameters α and β changing the behaviour of its loss function, which expands the model from a simple VAE to a disentangled one. The model has been trained with different approaches as to how to set the values of these parameters. The results of those training runs are showcased and explained in more detail in further sections of this report.

Analysis of the task

1. Possible approaches to solve the problem with its pros/cons, detailed description of selected methodology:

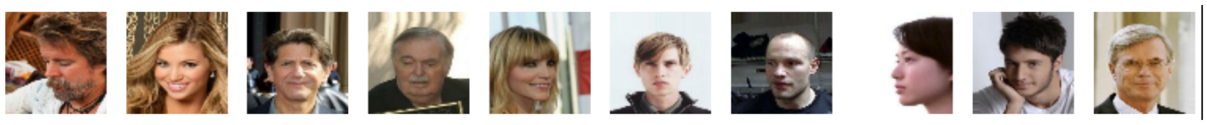
The presented problem is of exploratory nature, the goal from the very beginning was to employ a specific type of model in order to judge its performance, so as to the choice of a model no explanation is necessary.

What could possibly be a matter of argument is the choice of the specific architecture implemented. The model's architecture was hand picked in order to achieve a satisfactory performance whilst minimising the time necessary to complete the training run. The exact architecture used will be thoroughly explained in the internal specification section.

2. Presentation of possible/available datasets and detailed description of the chosen ones:

For this particular project the CelebA dataset was chosen, as it provides over 200 thousand small sized images of celebrity faces which are labelled as to which particular features they contain. This dataset is ideal for this problem as the labelling of the features allows showcasing the freedom of manipulating the latent space created by the encoder in order to influence the generated face.

This dataset is available here:
[CelebFaces Attributes (CelebA) Dataset | Kaggle]


Sample images from CelebA

No other dataset was taken into consideration, as this one fulfilled all the requirements for this particular problem.

3. Analysis of available tools/frameworks/libraries suitable for task solution; detailed presentation of selected tools/approach:

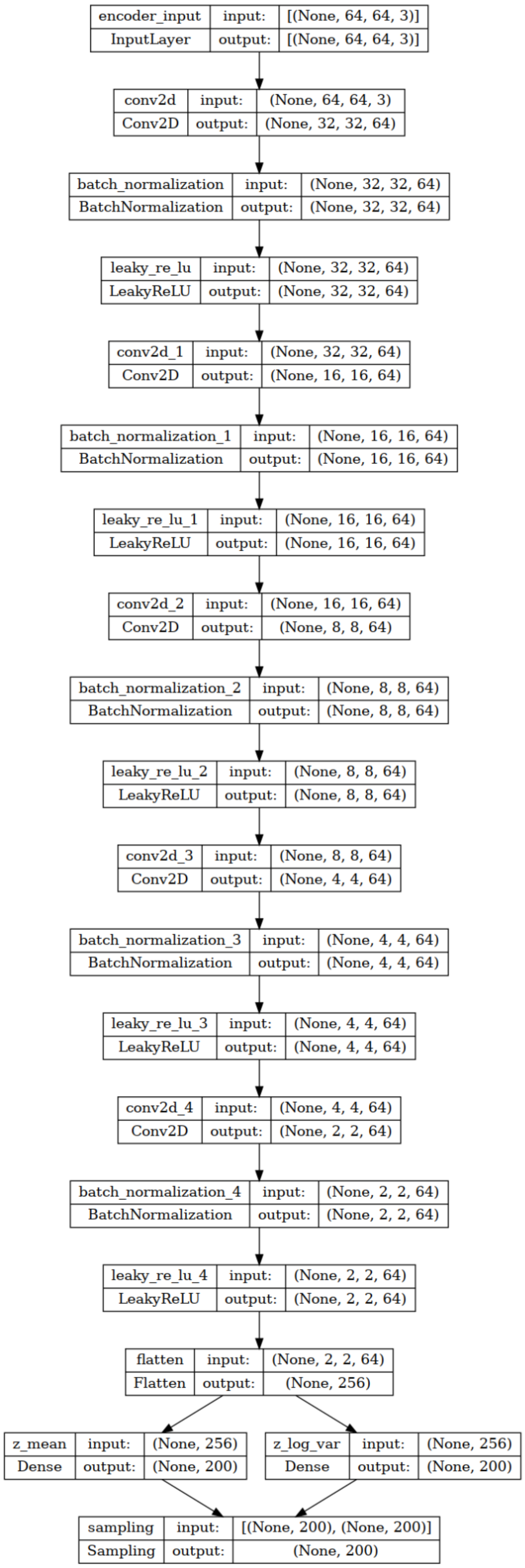
For this type of problem two most popular choices of tools are PyTorch and Keras. The chosen tool was Keras with TensorFlow backend, as the functional API provided by Keras simplifies the process of creating a neural network, without sacrificing performance, and is more widely used in commercial applications.

Internal and external specification of the software solution

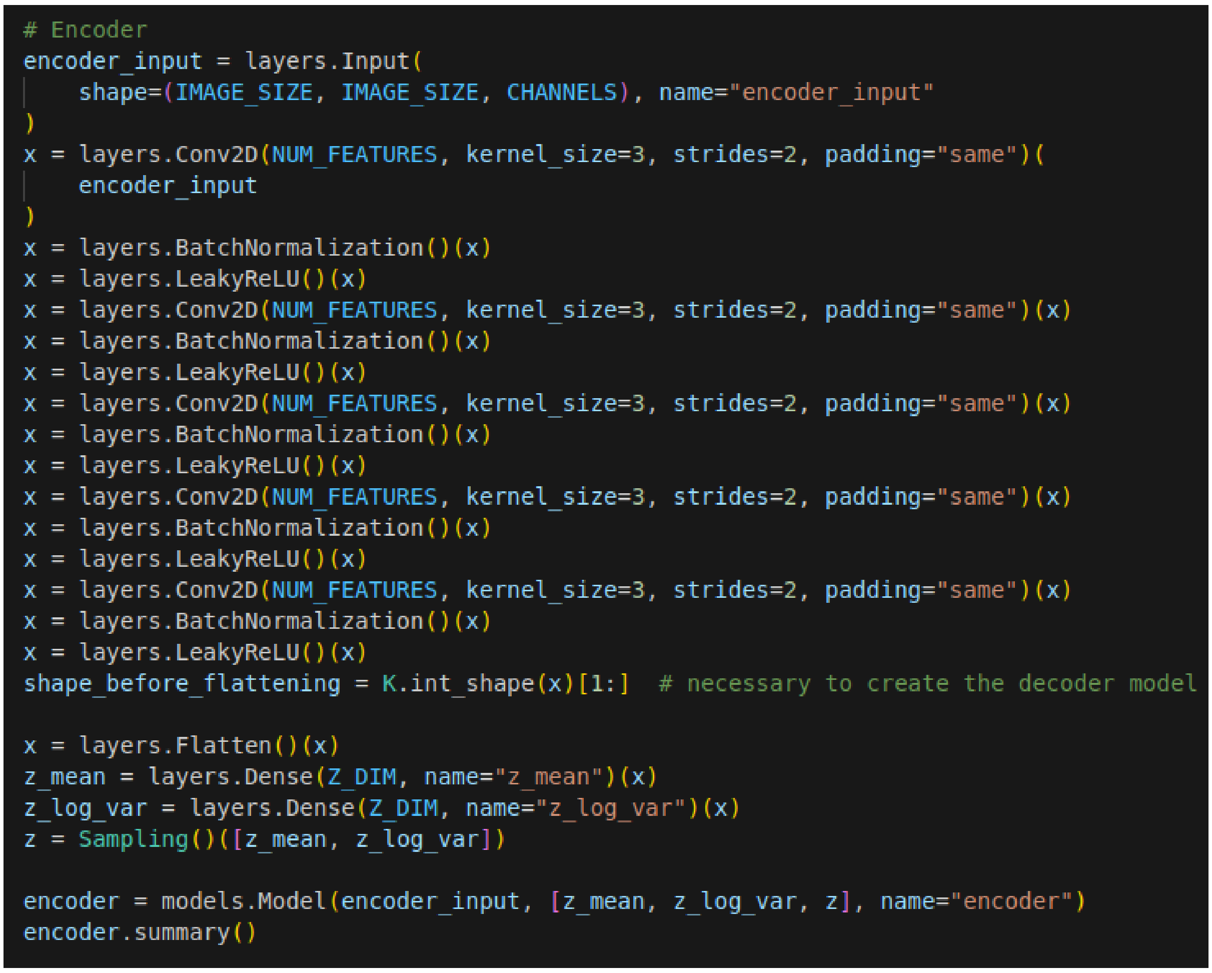
The model's encoder was built using convolutional layers followed by batch normalisation and would also benefit from using dropout layers. Convolutional layers are ideal for image processing applications as they allow for easier extraction of patterns from images. The reason for adding batch normalisation and dropout layers is to aid the training process. Batch norm layers make the learning more stable and even though they add computational overhead they make the learning process quicker as they influence the process in a way that allows the net to learn the data in less epochs than normally. Dropout layers randomly shut off neurons in a layer during the learning process which distributes the net's understanding of the data more evenly between the neurons instead of allowing it to depend too strongly on one individual neuron, this helps to prevent overfitting. The whole idea of a Variational Autoencoder is that the Encoder part maps images onto a distribution instead of fixed points in the latent space. In order to achieve this behaviour it is necessary to use a so called "Reparameterization Trick" the idea is that in order to produce an output out of the encoder we need to sample a distribution, this can be achieved using a stochastic node. This approach is problematic though as it is impossible to push backpropagation through a stochastic node. In order to be able to train the model we reorganise the nodes as follows:

1. We assume that the latent distribution is similar to a normal distribution
2. We change the network so that instead of encoding to a vector it encode to two vectors, one containing μ and the other containing $log(\sigma^2)$ (so called log_var trick) which are both parameters describing normal distribution
3. We get standalone σ by multiplying the log by 0.5 and taking the exponent of the result
4. We sample from our distribution as follows $z = \mu + \sigma\varepsilon$ where ε is sampled from a normal distribution

The encoder's architecture can be seen on the following picture


encoder_input | InputLayer | input: [(None, 64, 64, 3)] | output: [(None, 64, 64, 3)]
conv2d | Conv2D | input: (None, 64, 64, 3) | output: (None, 32, 32, 64)
batch_normalization | BatchNormalization | input: (None, 32, 32, 64) | output: (None, 32, 32, 64)
leaky_re_lu | LeakyReLU | input: (None, 32, 32, 64) | output: (None, 32, 32, 64)
conv2d_1 | Conv2D | input: (None, 32, 32, 64) | output: (None, 16, 16, 64)
batch_normalization_1 | BatchNormalization | input: (None, 16, 16, 64) | output: (None, 16, 16, 64)
leaky_re_lu_1 | LeakyReLU | input: (None, 16, 16, 64) | output: (None, 16, 16, 64)
conv2d_2 | Conv2D | input: (None, 16, 16, 64) | output: (None, 8, 8, 64)
batch_normalization_2 | BatchNormalization | input: (None, 8, 8, 64) | output: (None, 8, 8, 64)
leaky_re_lu_2 | LeakyReLU | input: (None, 8, 8, 64) | output: (None, 8, 8, 64)
conv2d_3 | Conv2D | input: (None, 8, 8, 64) | output: (None, 4, 4, 64)
batch_normalization_3 | BatchNormalization | input: (None, 4, 4, 64) | output: (None, 4, 4, 64)
leaky_re_lu_3 | LeakyReLU | input: (None, 4, 4, 64) | output: (None, 4, 4, 64)
conv2d_4 | Conv2D | input: (None, 4, 4, 64) | output: (None, 2, 2, 64)
batch_normalization_4 | BatchNormalization | input: (None, 2, 2, 64) | output: (None, 2, 2, 64)
leaky_re_lu_4 | LeakyReLU | input: (None, 2, 2, 64) | output: (None, 2, 2, 64)
flatten | Flatten | input: (None, 2, 2, 64) | output: (None, 256)
z_mean | Dense | input: (None, 256) | output: (None, 200)
z_log_var | Dense | input: (None, 256) | output: (None, 200)
sampling | Sampling | input: [(None, 200), (None, 200)] | output: (None, 200)

This architecture is implemented using the following Keras code:



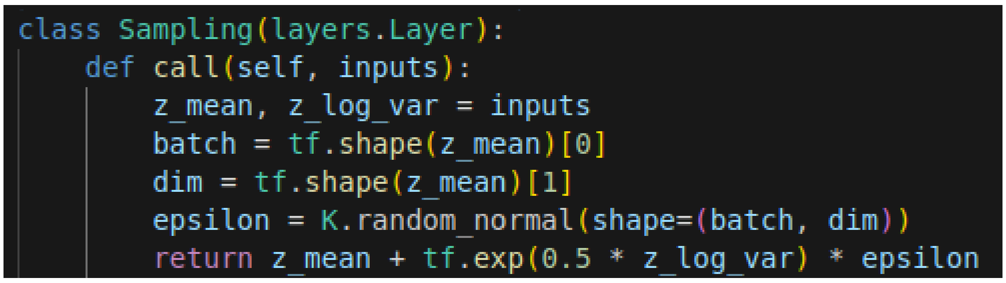
```
# Encoder
encoder_input = layers.Input(
    shape=(IMAGE_SIZE, IMAGE_SIZE, CHANNELS), name="encoder_input"
)
x = layers.Conv2D(NUM_FEATURES, kernel_size=3, strides=2, padding="same")(
    encoder_input
)
x = layers.BatchNormalization()(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(NUM_FEATURES, kernel_size=3, strides=2, padding="same")(x)
x = layers.BatchNormalization()(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(NUM_FEATURES, kernel_size=3, strides=2, padding="same")(x)
x = layers.BatchNormalization()(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(NUM_FEATURES, kernel_size=3, strides=2, padding="same")(x)
x = layers.BatchNormalization()(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(NUM_FEATURES, kernel_size=3, strides=2, padding="same")(x)
x = layers.BatchNormalization()(x)
x = layers.LeakyReLU()(x)
shape_before_flattening = K.int_shape(x)[1:]  # necessary to create the decoder model

x = layers.Flatten()(x)
z_mean = layers.Dense(Z_DIM, name="z_mean")(x)
z_log_var = layers.Dense(Z_DIM, name="z_log_var")(x)
z = Sampling()([z_mean, z_log_var])

encoder = models.Model(encoder_input, [z_mean, z_log_var, z], name="encoder")
encoder.summary()
```

This code creates a multilayered neural network containing layers like Conv2D with LeakyReLU activation function and batch normalisation layers.

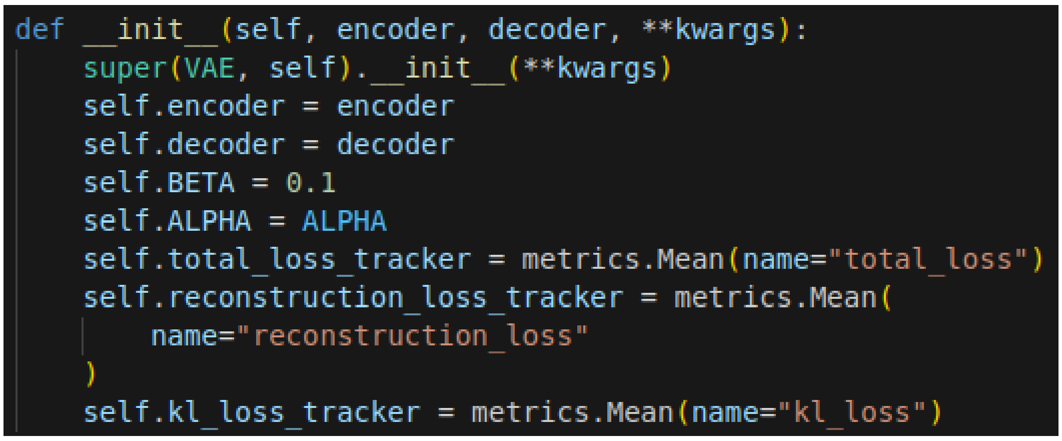
The sampling is done using a custom layer defined here like this:



```
class Sampling(layers.Layer):
    def call(self, inputs):
        z_mean, z_log_var = inputs
        batch = tf.shape(z_mean)[0]
        dim = tf.shape(z_mean)[1]
        epsilon = K.random_normal(shape=(batch, dim))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon
```

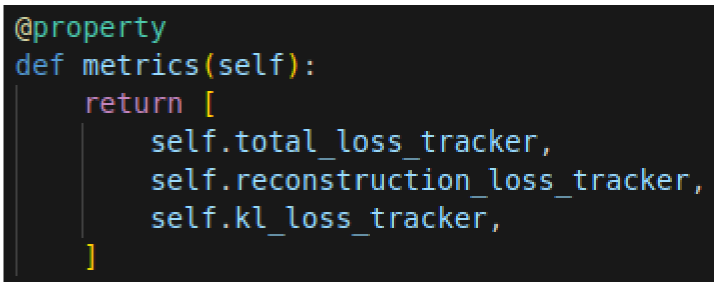
The decoder is a mirror image of the encoder but instead of using convolutional layers it uses transposed convolutional layers.

These two layers combined create a complete VAE whose specific parts are showcased below:



```
def __init__(self, encoder, decoder, **kwargs):
    super(VAE, self).__init__(**kwargs)
    self.encoder = encoder
    self.decoder = decoder
    self.BETA = 0.1
    self.ALPHA = ALPHA
    self.total_loss_tracker = metrics.Mean(name="total_loss")
    self.reconstruction_loss_tracker = metrics.Mean(
        name="reconstruction_loss"
    )
    self.kl_loss_tracker = metrics.Mean(name="kl_loss")
```

Model's constructor method



```
@property
def metrics(self):
    return [
        self.total_loss_tracker,
        self.reconstruction_loss_tracker,
        self.kl_loss_tracker,
    ]
```

Metric function returns 3 metrics:
Mean total loss
Mean reconstruction loss
Mean Kullback Leibler Divergence loss

Data Structures

1. Sampling - custom layer for sampling the latent space
2. Vae - class representing a variational autoencoder model
3. ImageGenerator - custom callback generating 10 images per training epoch
4. BetaScheduler - custom callback dynamically changing the model's beta hyperparameter during training (in theory)

User Interface/GUI/Console

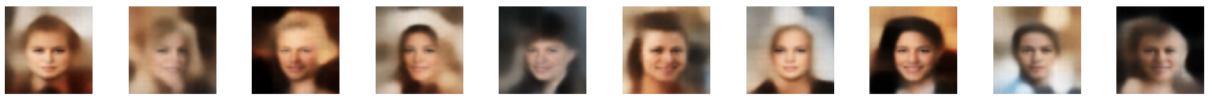
In this project I decided to use a very popular data science tool, the Jupyter Notebook. The Notebook is divided into individual cells, each cell contains either python code or markdown. Within the notebook we can find plots, graphs, images representing the model's performance and python code outputs. The notebook is built using a notebook kernel which is a "computational engine" that executes the code contained in a Notebook document.

Experiments

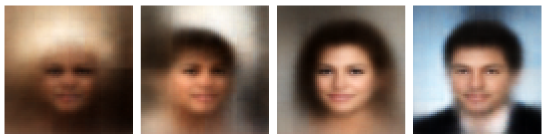
The experimentation phase was focused at training the model with different beta hyperparameter values. This particular parameter has been selected based on the fact that hyperparameters are considered generally and in the literature as the main points of model optimization and performance. In the case of a VAE type model beta parameter is changing the focus of the model between quality of reconstruction and latent space disentanglement. What is more, the mentioned hypothesis was confirmed by the results presented below.

Generation of new faces using different models

1. Model with very low beta parameter, faces are visibly sharper

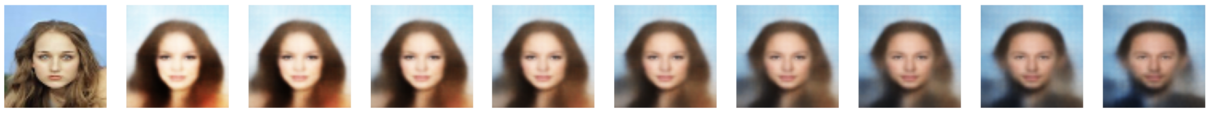


2. Model with a much higher beta parameter after the same training procedure

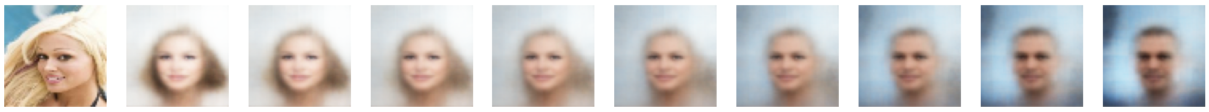


Vectorisation of latent space, in this example adding and subtracting a vector of MALE attribute. The feature vector is computed by calculating the centroid of data points labelled as containing chosen feature and calculating the centroid of data points not containing said feature and calculating a vector pointing from one to the other.

1. Model with very low beta, higher quality of reconstruction

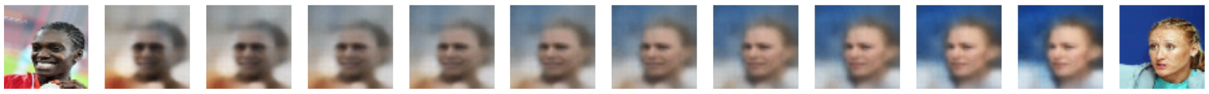


2. Model with higher beta, more focus on disentanglement resulting in more blurred reconstruction

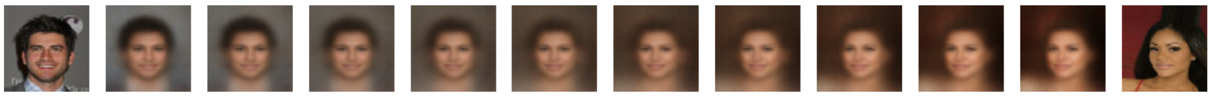


Morphing of faces using models, faces are first encoded then a vector from one face to another is calculated

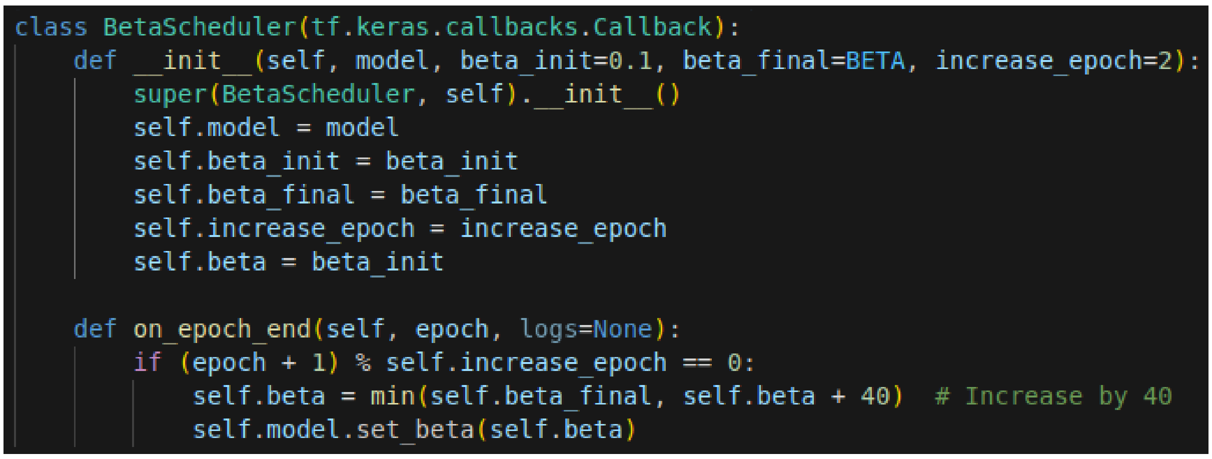
1. Model with very low beta



2. Model with higher beta



There was also an attempt to dynamically tune the beta parameter during the learning period using a custom callback written like this



```python
class BetaScheduler(tf.keras.callbacks.Callback):
    def __init__(self, model, beta_init=0.1, beta_final=BETA, increase_epoch=2):
        super(BetaScheduler, self).__init__()
        self.model = model
        self.beta_init = beta_init
        self.beta_final = beta_final
        self.increase_epoch = increase_epoch
        self.beta = beta_init

    def on_epoch_end(self, epoch, logs=None):
        if (epoch + 1) % self.increase_epoch == 0:
            self.beta = min(self.beta_final, self.beta + 40)  # Increase by 40
            self.model.set_beta(self.beta)
```

BetaScheduler implementation

However, this approach did not work as intended as the beta, despite being correctly updated inside the callback class, did not seem to change inside the model itself, during the learning process. This is a potential area open to more experimentation and an issue to which a solution needs to be found as it would probably yield interesting results if implemented correctly.

Summary, overall conclusions, possible improvements, future work

To summarise the research, a Variational Autoencoder is an interesting model which can be used in many applications. The architecture guarantees a continuous latent space which allows generation of new images by sampling points from it. Controllable disentanglement also allows to vectorise certain features seen in the data which makes uses such as controlling the presence of a feature in a generated image or image morphing. These features make VAE a versatile model that could be applied in many circumstances.

When it comes to improving the performance of the model, the first thing that comes to mind is improving the process of hyperparameter tuning and picking an optimal value of the beta parameter. This particular model is also implemented on a relatively small neural net to keep the training time realistic. This is another angle of potential improvement for future use, expanding the amount of filters and layers used in the network.

Future work could also include implementation of a user-friendly GUI in order to make using the model more intuitive and easier.

References – list of sources used during the work on the project:

Variational autoencoder - Wikipedia
[1312.6114] Auto-Encoding Variational Bayes
[1606.05579] Early Visual Concept Learning with Unsupervised Deep Learning

Project repository:

https://github.com/lukaszkacperfilipek/BIAI.git