LAB 05 - Autoencoders

In this laboratory we will cover Autoencoders and Variational Autoencoders. Implementations will cover basic usage and denoising. Aditionally, we will start to build our own custom models and introduce Tensorflow Probability. This knowledge will prove helpful in upcoming laboratories.

Custom TF Keras Model – Vanilla Autoencoder

We will define a custom model that largely simluates the behavior of a regular tf.keras model. This should prove to be very simple. Firstly, we will define a class that inherits tf.keras.Model and calls it's initializer.

```
class Autoencoder(tf.keras.Model):
    def __init__(self):
        super(Autoencoder, self).__init__()
        ...
```

Afterwards, inside __init__ we will define all needed components. In our case we shall define and encoder network, a decoder network, a loss function and an optimizer. Considering time efficiency, we will work with the MNIST dataset. The encoder will consist in two convolutional layers and a dense layer that outputs the encoding which we will set to be of length 10. The decoder will apply a fully connected layer on top of the encoding and reshape the output to generate a small picture in order to start a series of upconvolutions. Those can be implemented in one of two fashions: by using transposed convolutions with stride 2 or by upsampling the image and applying a regular convolution on top. The code presented will showcase the first approach while the latter will be left as an exercise.

```
class Autoencoder(tf.keras.Model):
    def __init__(self):
        super(Autoencoder, self).__init__()

        self.encoder = tf.keras.Sequential([
            tf.keras.layers.InputLayer(input_shape = (28, 28, 1)),
            tf.keras.layers.Conv2D(
                filters = 32,
                kernel_size = 3,
                strides = 2,
                activation = 'relu'
            ),
            tf.keras.layers.Conv2D(
                filters = 64,
                kernel_size = 3,
                strides = 2,
                activation = 'relu'
            ),
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(10),
        ])

        self.decoder = tf.keras.Sequential([
            tf.keras.layers.InputLayer(input_shape = (10)),
            tf.keras.layers.Dense(7 * 7 * 32, activation = 'relu'),
            tf.keras.layers.Reshape((7, 7, 32)),
            tf.keras.layers.Conv2DTranspose(
                filters = 64,
                kernel_size = 3,
                strides = (2, 2),
                padding = "SAME",
                activation = 'relu'
            ),
            tf.keras.layers.Conv2DTranspose(
                filters = 32,
                kernel_size = 3,
                strides = 2,
```

```
                padding = "SAME",
                activation = 'relu'
            ),
            tf.keras.layers.Conv2DTranspose(
                filters = 1,
                kernel_size = 3,
                strides = 1,
                padding = "SAME",
                activation = 'sigmoid'
            )
        ])


        self.loss_function = tf.keras.losses.binary_crossentropy
        self.optimizer = tf.keras.optimizers.Adam()

        self.create_checkpoint()
```

Since our images will be normalized and have pixel values within the [0..1] interval, our final activation will be a sigmoid function and we will use binary_crossentropy as our loss. Notice that lastly we call a create_checkpoint function that we will define in order to save and restore our model. It will look as follows:

```
def create_checkpoint(self, path = './model/checkpoint'):
    self.path = path
    self.ckpt = tf.train.Checkpoint(model = self)
    self.ckpt_manager = tf.train.CheckpointManager(
        self.ckpt, self.path, max_to_keep = 1
    )
```

Before jumping into the *fitting* procedure we will define some simple helper functions that we will use later.

```
def save(self):
    self.ckpt_manager.save()
```

```
def restore(self):
    self.ckpt.restore(self.ckpt_manager.latest_checkpoint)

def decode(self, encodings):
    return self.decoder(encodings)

def encode(self, inputs):
    return self.encoder(inputs)

def reconstruct(self, inputs):
    return self.decoder(self.encoder(inputs))
```

In order to train our model we will define two functions: *fit* and *fit_ iteration*. The *fit* function will just call the *fit_ iteration* function the neccesary amount of times so the latter is the one that represents the core of the training process. It will receive as input a batch of samples, compute the gradients for the iteration and apply them. In tensorflow this is done using a **Gradient Tape** that records gradients for specified variables during computations. More specific, we will define a gradient tape, tell it what variables it needs to watch the gradient for, perform all computations and then apply the gradients using our optimizer.

```
def fit_iteration(self, inputs):
    with tf.GradientTape() as tape:
        tape.watch(self.trainable_variables)

        reconstructions = self.reconstruct(inputs)
        loss = tf.reduce_sum(
            self.loss_function(inputs, reconstructions)
        )

    self.optimizer.apply_gradients(zip(
        tape.gradient(loss, self.trainable_variables),
        self.trainable_variables
```

```
    ))

    return loss
```

In this scenario, self.trainable_variables contains all trainable variables from our model since it extends tf.keras.Model. After defining the tape we compute our loss. Outside the gradient tape context we will apply our gradients; this should always be done as such for computational efficiency.

Finally, our *fit* function will roughly simulate the behavior of a regular tf.keras fit function. It shall receive as a first parameter a tensorflow dataset which we will convert to an iterator in order to generate batches.

```
def fit(self, dataset, epochs = 1, steps_per_epoch = 1):
    dataset = iter(dataset)
    for epoch_n in range(epochs):
        for iteration_n in range(steps_per_epoch):
            inputs = next(dataset)
            loss = self.fit_iteration(inputs)
            print(
                f'Epoch {epoch_n}: ' +
                f'{np.round((iteration_n + 1) / steps_per_epoch * 100, 2)}%, ' +
                f'loss: {loss}',
                end = '\r'
            )
        print('')
```

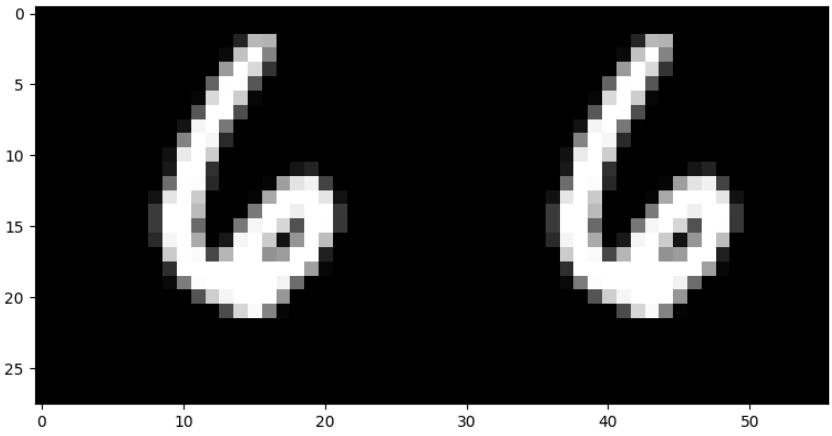
In order to evaluate our model we will load the dataset as we have done in our previous labs, instantiate a new model and run the training procedure for two epochs. We have went through this numerous times so the code will be attached here for simplicity:

```
dataset, meta = tfds.load(
    'mnist',
```

```
    as_supervised = True,
    with_info = True
)

def normalize(image, label):
    processed_image = tf.cast(image, tf.float32) / 255.
    return processed_image

batch_size = 64
train_ds = dataset['train'].map(normalize)
n_samples = meta.splits['train'].num_examples
steps_per_epoch = n_samples // batch_size

model = Autoencoder()
model.fit(
    train_ds.batch(batch_size).shuffle(100000).repeat(),
    epochs = 2,
    steps_per_epoch = steps_per_epoch
)
model.save()
# model.restore()

sample_dataset = iter(dataset['test'].map(normalize).batch(1))
for sample_n in range(5):
    original = next(sample_dataset)
    reconstruction = model.reconstruct(original)
    original = np.reshape(original, (28, 28))
    reconstruction = np.reshape(original, (28, 28))

    plt.imshow(
        np.concatenate([original, reconstruction], axis = -1),
        cmap = 'gray'
    )
    plt.show()
```

Notice two things. Firstly, there is a commented model.restore() call. After you have trained your model there is no need to fit it again and you can comment the fitting and

saving lines, uncommenting the model.restore() call. Secondly, this code features plots of reconstructions performed on images from the test dataset in order to visually evaluate the models performance. They should look something like this:



Exercise: Replace the Conv2DTranspose layers with upsampling and convolutional layers and repeat the experiment.

Variational Autoencoders

In order to transform our model into a variational autoencoder we should alter the encoding process and our loss. This can be done in a straight forward manner, by generating means and variances with de last dense layer from the encoder network, generating a random variable, scaling it accordingly and writing the KL term for the loss. You can tackle this approach as an exercise. In this laboratory we will showcase combining tf.keras layers with **tensorflow_probability** layers.

You should install the package and import it.

```
import tensorflow_probability as tfp
```

Firstly, the second to last layer of the encoder should compute the parameters for encoding sampling.

```
class Autoencoder(tf.keras.Model):
    def __init__(self):
        super(Autoencoder, self).__init__()
        ...
        self.encoder = tf.keras.Sequential([
            ...
            tf.keras.layers.Flatten(),
            tf.keras.layers.Dense(
                tfp.layers.MultivariateNormalTriL.params_size(encoding_size)
            ),
            self.sampling_layer
        ])
```

It will be a dense layer generating the required amount of parameters in order to sample an encoding according to a normal distribution. tfp.layers.MultivariateNormalTriL is the name of the layer that does the sampling. Calling params_size(encoding_size) will return the number of parameters neccesary to sample a vector of length encoding_size. We will define our last layer, the sampling layers as follows:

```
prior = tfp.distributions.Independent(
    tfp.distributions.Normal(loc = tf.zeros(encoding_size), scale = 1),
    reinterpreted_batch_ndims = 1
)
self.sampling_layer = tfp.layers.MultivariateNormalTriL(
    encoding_size,
    activity_regularizer = tfp.layers.KLDivergenceRegularizer(
        prior, weight = 1.0
    )
)
```

The prior is defined as a normal distribution with a zero mean and a variance of 1. The sampling layer is the aforementioned tfp.layers.MultivariateNormalTriL. It is regularized using KL Divergence, so, instead of manually implementing the sampling and the KL Divergence formula, those layers will do the job for you. The weight specified there is the weight associated with the KL loss; it is the **beta** parameter in the context of a disentangled autoencoder [1].

Finally, the fit_iteration function should be adapted to include the loss coming from the activity regularizer of the sampling layer with respect to the generated encodings.

```
def fit_iteration(self, inputs):
    with tf.GradientTape() as tape:
        tape.watch(self.trainable_variables)

        encodings = self.encode(inputs)
        reconstructions = self.decode(encodings)
        loss = tf.reduce_sum(
            self.loss_function(inputs, reconstructions)
        ) + tf.reduce_sum(
            self.sampling_layer.activity_regularizer(encodings)
        )

    self.optimizer.apply_gradients(zip(
        tape.gradient(loss, self.trainable_variables),
        self.trainable_variables
    ))

    return loss
```

The rest of the code and the training procedure remain unchanged. Set an encoding size of 10, train the model and use the following code to visualize the effect of jittering the first two components of the encoding.

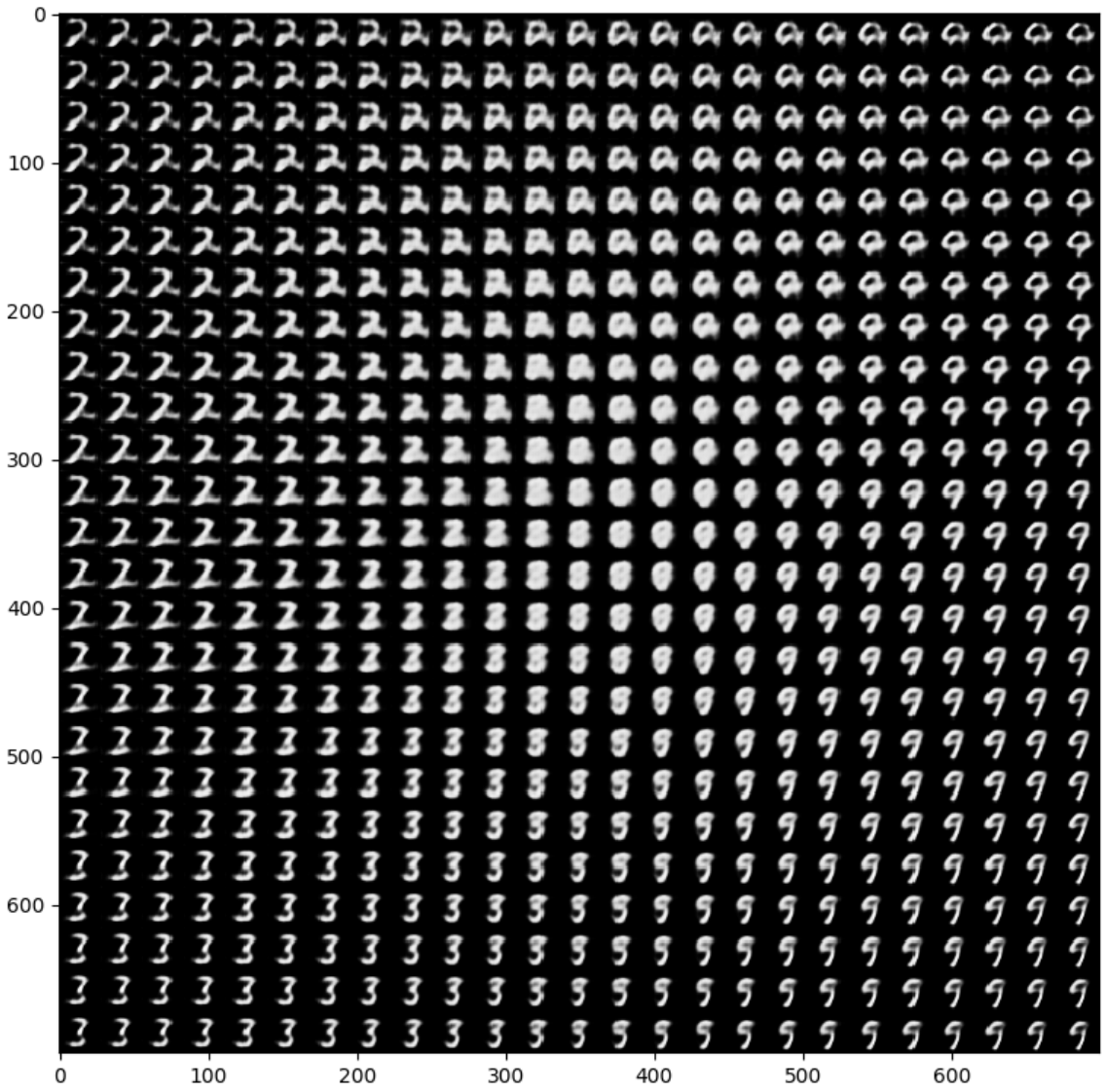
[1] https://openreview.net/pdf?id=Sy2fzU9gl

```
n_samples = 25
limit = 2.5
all_samples = np.zeros(
    shape = (n_samples * 28, n_samples * 28),
    dtype = np.float32
)
for i, x in enumerate(np.linspace(-limit, limit, n_samples)):
    for j, y in enumerate(np.linspace(-limit, limit, n_samples)):
        im = model.decode(np.array([[x, y, 0, 0, 0, 0, 0, 0, 0, 0]]))
        im = np.reshape(im, (28, 28))
        all_samples[
            i * 28: (i + 1) * 28,
            j * 28: (j + 1) * 28
        ] = im
plt.imshow(all_samples, cmap = 'gray')
plt.show()
```

The output should look similar to the last figure.

Exercise: with the help of you supervisor train a denoising autoencoder.

Exercise: with the help of you supervisor design an MNIST classifier based on stacked autoencoders.



Variational autoencoder generated samples.