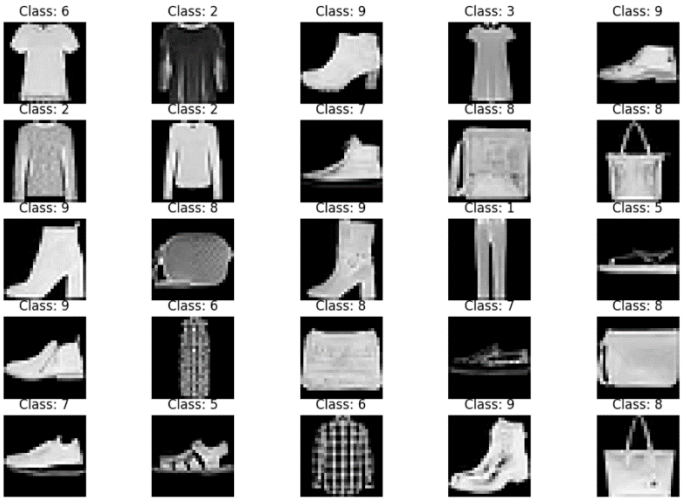
This Project is used as an introduction to generative models. With the help of a Variational Autoencoder I will generate new images as well as use the latent space vector to transform pictures from one to another. The dataset I'm going to use is fashion mnist. It consists of 60k pictures of 10 pieces of cloth (shirt, T-shirt, trousers, sandals, etc.).



The architecture of the VAE:

Encoder

```
class VariationalAutoEncoder():
    def __init__(self):
        self._build()

    def _build(self):

        #Encoder

        def sampling(args):
            mu,log_var = args
            epsilon = k.random_normal(shape = k.shape(mu),mean = 0,stddev = 1)
            return mu + k.exp(log_var/2) * epsilon

        image_input = Input(shape = (img_height,img_width,1), name = "vae_input")

        x_enc = Conv2D(784,(3,3),(2,2),padding = "same", name = "conv_784_encoder")(image_input)
        x_enc = BatchNormalization(name = "batch_norm_784_encoder")(x_enc)
        x_enc = Activation("relu",name = "activation_784_encoder")(x_enc)

        x_enc = Conv2D(256,(3,3),(2,2),padding = "same", name = "conv_256_encoder")(x_enc)
        x_enc = BatchNormalization(name = "batch_norm_256_encoder")(x_enc)
        x_enc = Activation("relu",name = "activation_256_encoder")(x_enc)

        x_enc = Conv2D(32,(3,3),(1,1),padding = "same", name = "conv_32_encoder")(image_input)
        x_enc = BatchNormalization(name = "batch_norm_32_encoder")(x_enc)
        x_enc = Activation("relu",name = "activation_32_encoder")(x_enc)

        x_enc = Conv2D(4,(3,3),(1,1),padding = "same", name = "conv_4_encoder")(image_input)
        x_enc = BatchNormalization(name = "batch_norm_4_encoder")(x_enc)
        x_enc = Activation("relu",name = "activati132on_4_encoder")(x_enc)

        x_enc = Flatten(name = "flatten_encoder")(x_enc)
        self.mu = Dense(latent_space, activation = "linear",name = "mu")(x_enc)
        self.log_var = Dense(latent_space, activation = "linear", name = "log_var")(x_enc)

        encoder_output = Lambda(sampling, name = "encoder_output")([self.mu,self.log_var])

        self.encoder = Model(inputs = [image_input], outputs = [encoder_output])
```

Decoder

```
#Decoder

latent_space_input = Input(shape=(latent_space,), name = "decoder_input")

x_dec = Dense(3136, activation = "linear",name = "decoder_dense")(latent_space_input)
x_dec = Reshape(target_shape = (7,7,64)) (x_dec)

x_dec = Conv2DTranspose(784,(3,3),(1,1),padding = "same", name = "conv_784_decoder")(x_dec)
x_dec = BatchNormalization(name = "batch_norm_784_decoder")(x_dec)
x_dec = Activation("relu",name = "activation_784_decoder")(x_dec)

x_dec = Conv2DTranspose(256,(3,3),(1,1),padding = "same", name = "conv_256_decoder")(x_dec)
x_dec = BatchNormalization(name = "batch_norm_256_decoder")(x_dec)
x_dec = Activation("relu",name = "activation_256_decoder")(x_dec)

x_dec = Conv2DTranspose(32,(3,3),(2,2),padding = "same", name = "conv_13_31decoder")(x_dec)
x_dec = BatchNormalization(name = "batch_norm_32_decoder")(x_dec)
x_dec = Activation("relu",name = "activation_32_decoder")(x_dec)


x_dec = Conv2DTranspose(4,(3,3),(2,2),padding = "same", name = "conv_4_decoder")(x_dec)
x_dec = BatchNormalization(name = "batch_norm_4_decoder")(x_dec)
x_dec = Activation("relu",name = "activation_4_decoder")(x_dec)

x_dec = Conv2DTranspose(1,(3,3),(1,1),padding = "same", name = "conv_1_decoder")(x_dec)
x_dec = BatchNormalization(name = "batch_norm_1_decoder")(x_dec)
decoder_output = Activation("sigmoid",name = "activation_1_decoder")(x_dec)

self.decoder = Model(inputs = [latent_space_input], outputs = [decoder_output])
```

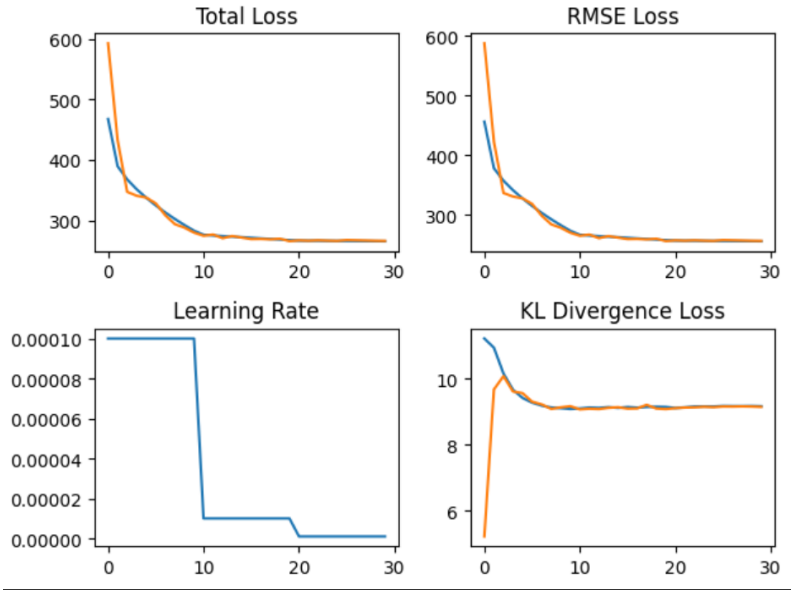
Autoencoder

```
#Variational AutoEncoder

model_input = image_input
model_output = self.decoder(encoder_output)

self.full_model = Model(model_input, model_output)
```

Losses

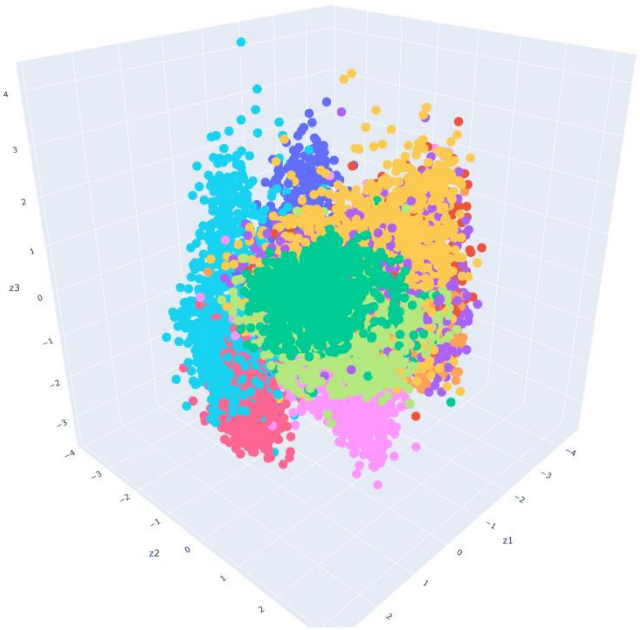
```
def custom_compile(self, learning_rate, rmse_multiplier):
    self.learning_rate = learning_rate

    def vae_r_loss(y_true, y_pred):
        r_loss = k.mean(k.square(y_true - y_pred), axis = [1])
        return rmse_multiplier * r_loss

    def vae_kl_loss(y_true, y_pred):
        kl_loss =  - 0.5 * k.sum(1 + self.log_var - k.square(self.mu) - k.exp(self.log_var), axis = 1)
        return kl_multiplier * kl_loss

    def vae_loss(y_true, y_pred):
        reconstruction_loss = vae_r_loss(y_true, y_pred)
        kl_loss = vae_kl_loss(y_true, y_pred)
        return  reconstruction_loss + kl_loss

    optimizer = Adam(learning_rate=learning_rate)
    self.full_model.compile(optimizer=optimizer, loss = vae_loss,  metrics = [vae_r_loss,vae_kl_loss])
```

Training Metrics with learning rate scheduler:



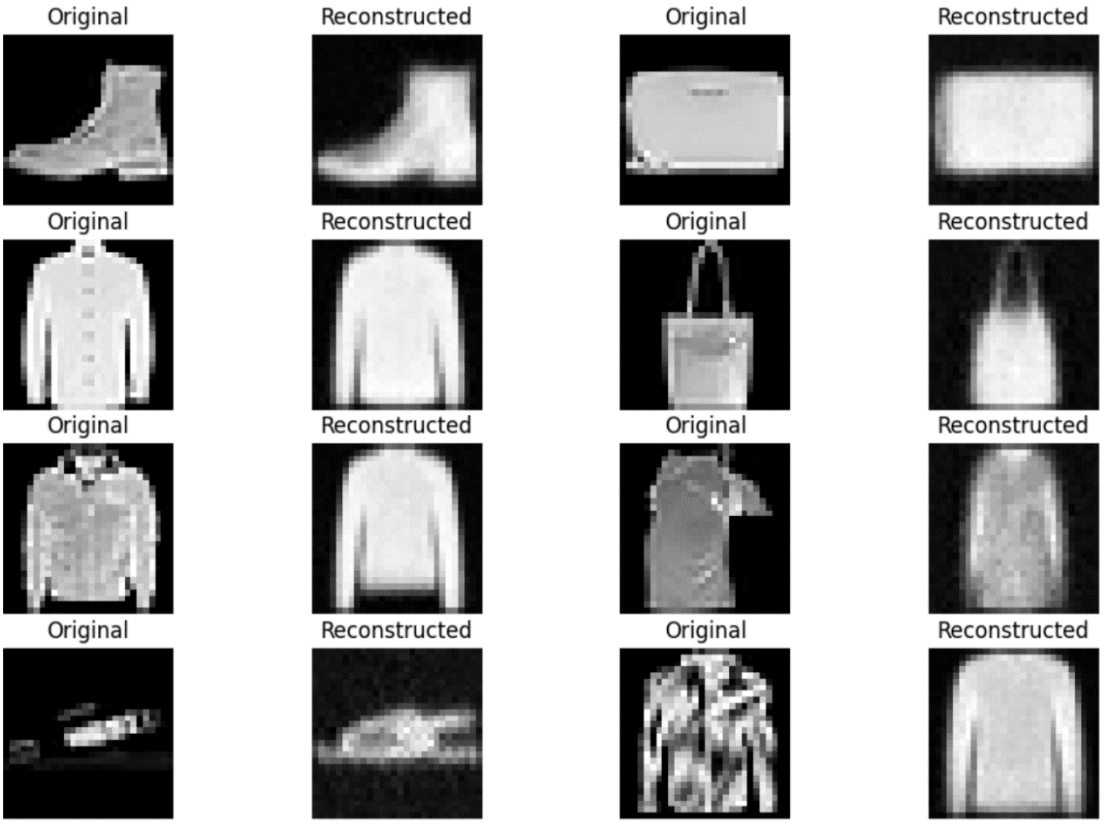
I tried multiple latent space dimensions with different multipliers for the reconstruction loss. I ended up with a latent space of 3 dimensions and multiplying the reconstruction loss by 5000, while leaving the KL divergence loss against a normal distribution untouched.

The VAE latent dimension after training, each colour representing each class:



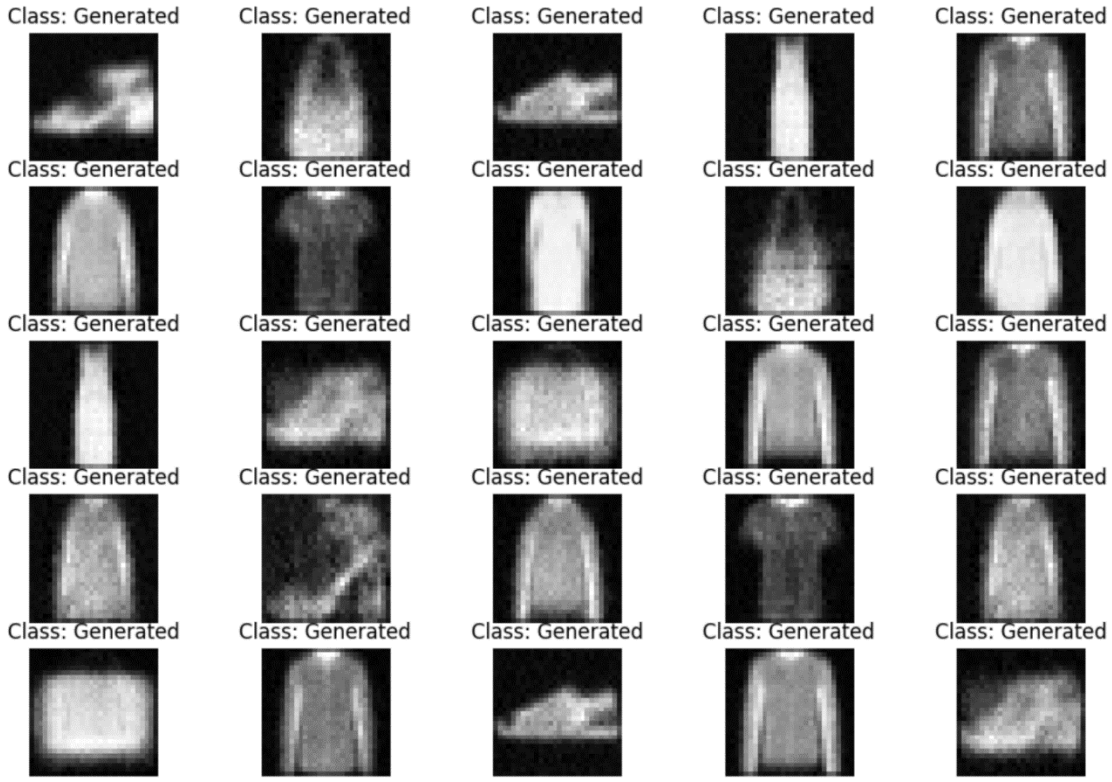
We can see that, although the distribution has more standard deviation than a normal distribution(which may cause problems in the generation of new images) pictures are all in a compressed space, not dispersed in the 3D space(which would cause even more problems when generating images, since a lot of the samples from a normal distribution would end up with a point in the latent space not represented by any class, therefore the decoder would reconstruct pictures with no sense)

Reconstructed images from test dataset:



With greater reconstruction loss multiplier and latent space dimensions, the reconstruction quality increases as the pictures have more representation, but at the spent of not decreasing KL Divergence loss sufficiently for approaching a latent space modeled by a multivariate normal distribution(and therefore not generating real images when passing random samplings from a multivariate normal distribution with the same dimension of the latent space to the decoder). Greater latent space dimensions mean less points in each quadrant of the dimensional space (dimensionality curse), so the greater the latent space dimension, the greater probability of choosing a point in a quadrant without sufficient trained datapoints.

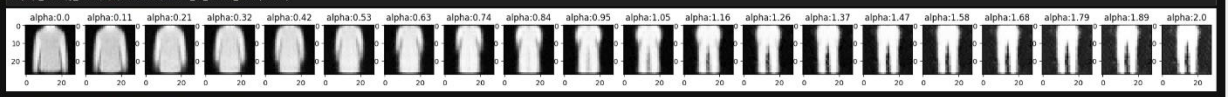
Sampling from a multivariate variable normal distribution of 3 variables to generate new images:



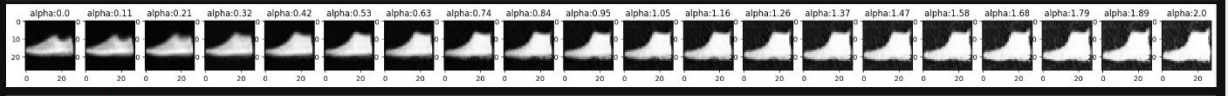
Finally, we can use the latent space vectors to get representation for the classes.

Suppose we grab all the pictures from the class dress, let's say 5k pictures. If we pass them through the encoder, we would get 5k latent space vectors. If we get the mean of these vectors, we would have a latent space that represents all the pictures from the class dress. If we pass this to the decoder, we would then have a picture of a dress that represents all the dress pictures in the data. With this idea, we can get the mean latent space vector of two classes, say dress and trousers, have the respective representation, and do vector algebra to morph these latent space vectors. We can grab the dress vector and sum the trousers vector; the result would be a representation of both dresses and trousers combined. The decoder would then reconstruct a picture that represents both these classes. We could, instead of summing, given a class vector, sum another class vector multiplied by a scalar alpha. Then we can adjust how much we want to morph our first vector to the second vector. Below are some examples of this. When alpha equals 0, we sum nothing to the original vector representing all the pictures of a class. As we increase this alpha, we morph the original vector (and therefore the original picture) to the second class representation more intensively.

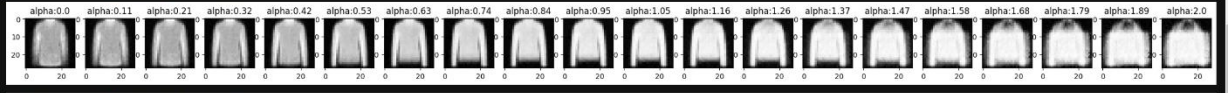
[20]: morph(x_train,y_train,4,1,estimator,amount_of_vector_multipliers)



[14]: morph(x_train,y_train,7,9,estimator,amount_of_vector_multipliers)



[15]: morph(x_train,y_train,6,8,estimator,amount_of_vector_multipliers)



[16]: morph(x_train,y_train,0,1,estimator,amount_of_vector_multipliers)