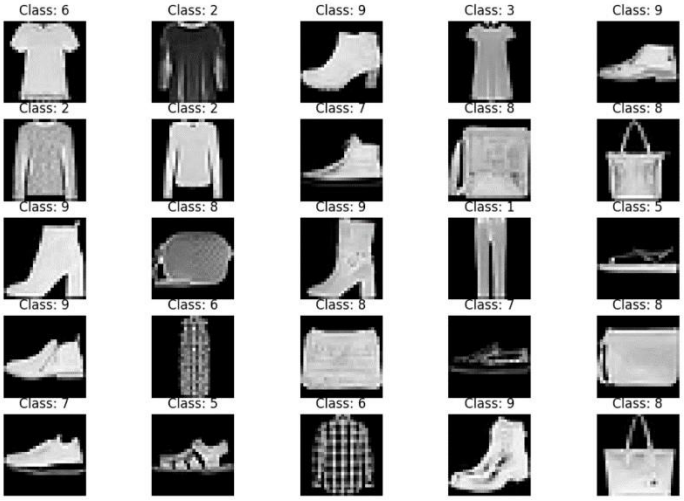
This project serves as an introduction to generative models. Utilizing a Variational Autoencoder (VAE), I will generate new images and use the latent space vector to transform pictures from one class to another. The dataset employed is Fashion MNIST, comprising 60k images of 10 clothing items (shirt, 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])
```

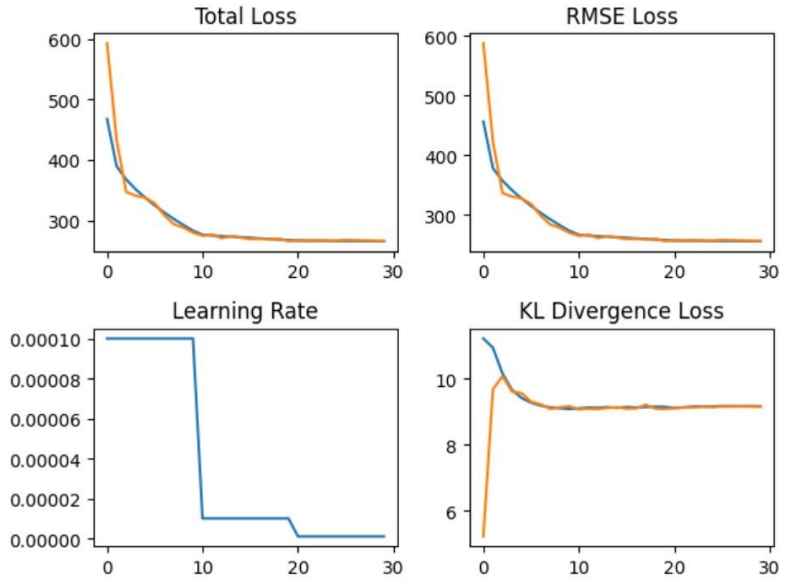
Autoencoder

```
#Variational AutoEncoder

model_input = image_input
model_output = self.decoder(encoder_output)

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

Losses

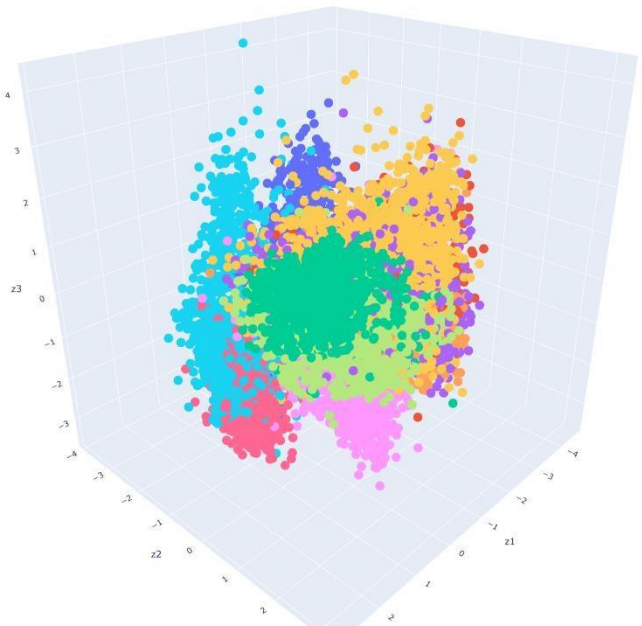
```
def custom_compile(self, learning_rate, rmse_multiplier):
    self.learning_rate = learning_rate

    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:



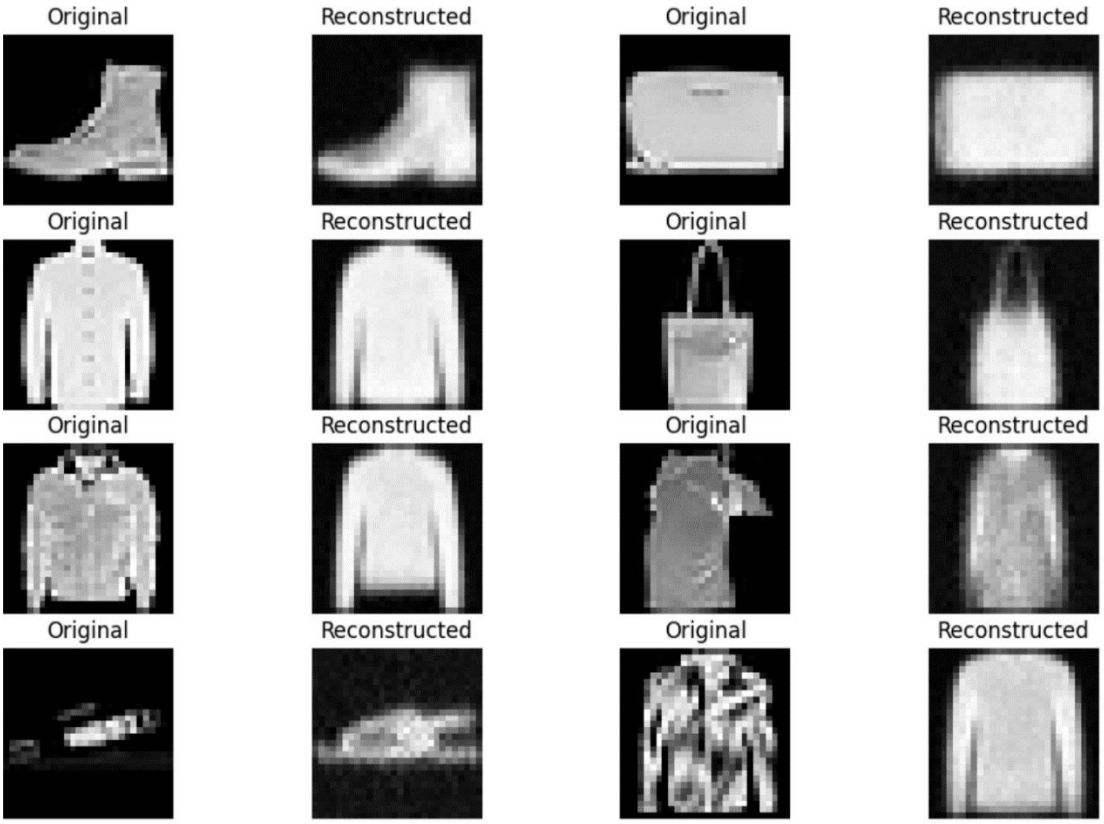
I experimented with various latent space dimensions and different multipliers for the reconstruction loss. Eventually, I settled on a latent space of 3 dimensions, multiplying the reconstruction loss by 5000, while leaving the KL divergence loss against a normal distribution unaffected.

Post-training, the VAE latent dimension is represented visually, with each color corresponding to a distinct class:



Observably, though the distribution exhibits more standard deviation than a normal distribution (which may pose challenges in generating new images), all pictures are confined to a compressed space, avoiding dispersion in the 3D space. Such dispersion could lead to issues in image generation, as samples from a normal distribution might end up in a latent space not represented by any class, resulting in nonsensical picture reconstruction by the decoder.

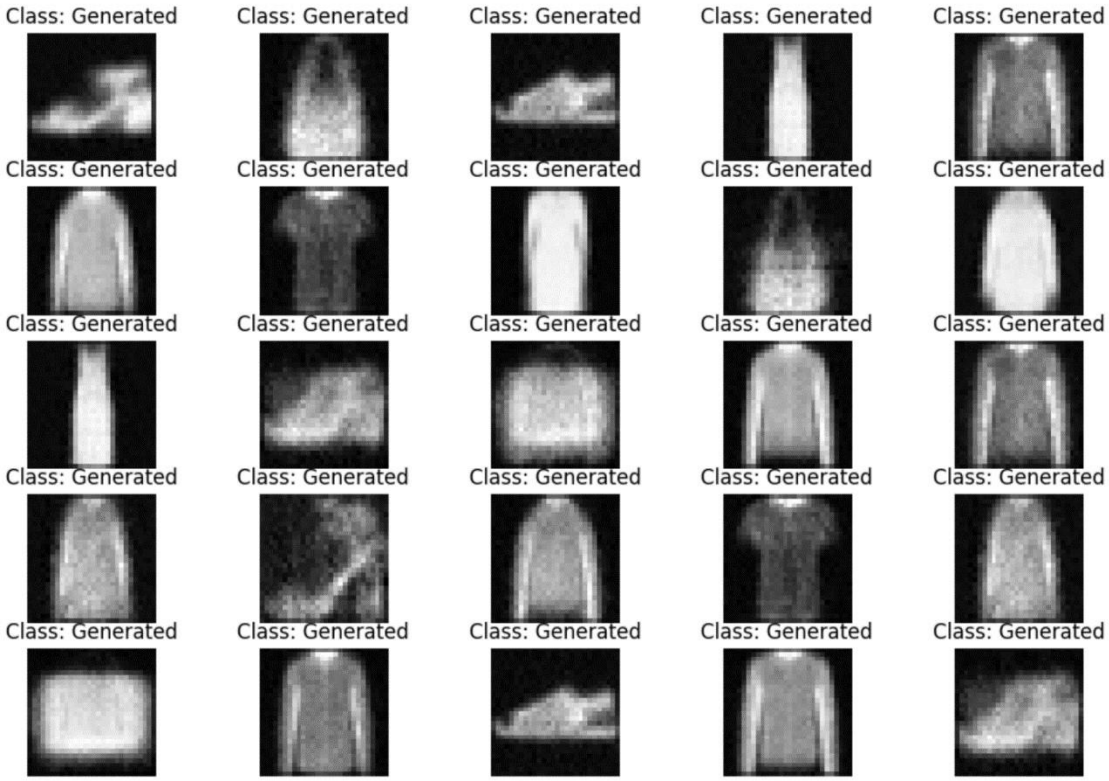
Reconstructed images from test dataset:



Increasing the reconstruction loss multiplier and latent space dimensions enhances reconstruction quality by providing more representation. However, this comes at the expense of insufficiently reducing KL Divergence loss, preventing the approach of a latent space modeled by a multivariate normal distribution. This hinders the generation of realistic images when passing random samples from a multivariate normal distribution with the same dimension as the latent space to the decoder.

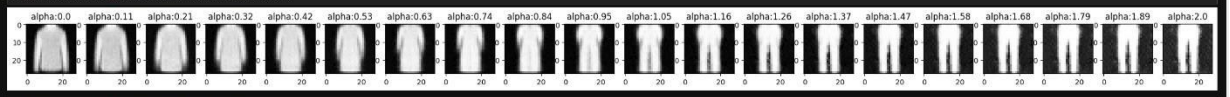
Greater latent space dimensions imply fewer points in each quadrant of the dimensional space (dimensionality curse). Consequently, a higher latent space dimension increases the probability of choosing a point in a quadrant without sufficient trained data points.

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

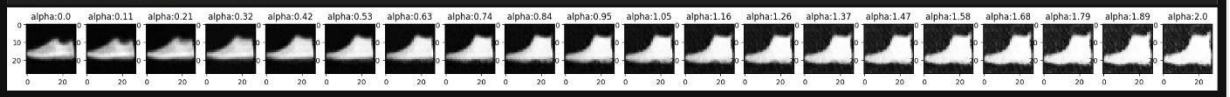


Lastly, latent space vectors can be employed to represent classes. Taking the example of the "dress" class with 5k pictures, passing them through the encoder produces 5k latent space vectors. Calculating the mean of these vectors yields a latent space representing all "dress" pictures. Passing this to the decoder results in a picture representing all dress pictures in the dataset. This concept extends to obtaining the mean latent space vector of two classes, such as "dress" and "trousers," and using vector algebra to morph these latent space vectors. Examples of this morphing, achieved by adjusting a scalar alpha, are provided below. When alpha is 0, nothing is added to the original vector representing all pictures of a class. As alpha increases, the original vector (and picture) morphs more intensively toward the representation of the second class.

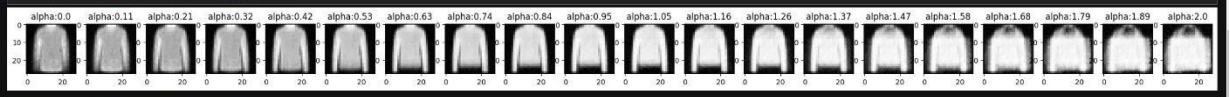
```
[20]: morph(x_train,y_train,4,1,estimator,amount_of_vector_multipliers)
```




```
[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)
```