HW4

Gabriel ROMON

1 Autoencoders

1. Here is a code snippet for the MLP architecture:

```
#encoder
input_img = Input(shape=(self.img_rows,self.img_cols,self.img_channels))
z = Flatten()(input_img)
z = Dense(self.z_dim)(z)
z = LeakyReLU(alpha=0.2)(z)
#decoder
output_img = Dense(n_pixels)(z)
output_img = Activation('sigmoid')(output_img)
output_img = Reshape((self.img_rows,self.img_cols,self.img_channels))(output_img)
```

2. Here is a code snippet for the convolutional architecture:

```
#encoder
input_img = Input(shape=(self.img_rows,self.img_cols,self.img_channels))
z = Conv2D(8, (3,3), strides=(2,2), padding='same')(input_img)
z = LeakyReLU(alpha=0.2)(z)
z = Conv2D(4, (3,3), strides=(2,2), padding='same')(z)
z = LeakyReLU(alpha=0.2)(z)
z = Flatten()(z)
z = Dense(self.z_dim)(z)
#decoder
output_img = Dense(196)(z)
output_img = LeakyReLU(alpha=0.2)(output_img)
output_img = Reshape((7,7,4))(output_img)
output_img = Conv2DTranspose(4, (3,3), strides=(2,2), padding='same')(output_img)
output_img = LeakyReLU(alpha=0.2)(output_img)
output_img = Conv2DTranspose(1, (3,3), strides=(2,2), padding='same')(output_img)
output_img = Activation('sigmoid')(output_img)
```

3. In each architecture the autoencoder was trained for 4000 epochs with a batch size of 64 and Adadelta optimizer. Results for $d = 10$ are shown in Figure 1. The convolutional architecture clearly achieves better, less blurry outputs. Besides, its final loss is around 0.15, while it is around 0.17 for the MLP architecture.

   The MLP network has $\sim$ 16.000 parameters while the convolutional one has $\sim$ 5.000 parameters. The better performance of the latter may be explained by the convolution operation which is more complex than a Dense layer and yields better representations in the latent space.

   Results for $d = 100$ are shown in Figure 2. For each architecture, higher d results in better outputs, but the convolutional network is still ahead in terms of visual sharpness.

Figure 1: Results of the autoencoder for $d = 10$

Figure 2: Results of the autoencoder for $d = 100$

4. To build a denoising autoencoder it suffices to modify the training loop with the following code:

```
noise = np.expand_dims(np.random.normal(scale=20/255,
                                        size=(self.img_rows,self.img_rows)), axis=2)
loss = self.ae.train_on_batch(curr_batch+noise,curr_batch)
```

Gaussian noise is added to each batch, so the network learns to get rid of it. Figure 3 shows the results for $d = 10$ and Figure 4 for $d = 100$. The removal of noise is already satisfying for $d = 10$.

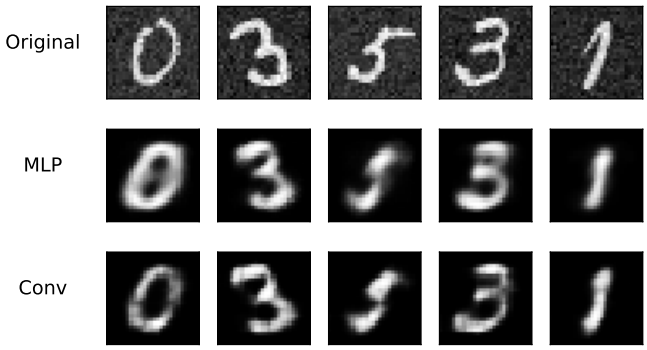

Figure 3: Results of the autoencoder for $d = 10$

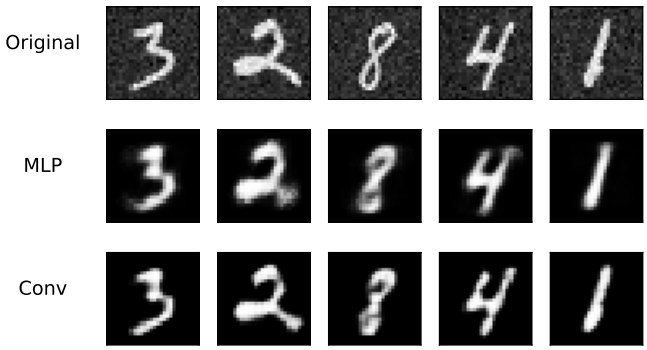

Figure 4: Results of the autoencoder for $d = 100$

2 Variational autoencoders

Encoding: the Gaussian case

Using the formula provided in the assignment,

$$\mathrm{KL}(P_1||P_2) = \frac{1}{2}\left(\mathrm{tr}(\Sigma) + \mu^T\mu - d - \log\det\Sigma\right)$$

In the special case where Σ is diagonal, this rewrites as

$$\begin{aligned}\mathrm{KL}(P_1||P_2) &= \frac{1}{2}\left(\sum_{i=1}^{d}\Sigma_{ii} - \log\prod_{i=1}^{d}\Sigma_{ii} + \mu^T\mu - d\right)\\ &= \frac{1}{2}\left(\sum_{i=1}^{d}\exp\log\Sigma_{ii} - \sum_{i=1}^{d}\log\Sigma_{ii} + \mu^T\mu - d\right)\\ &= \frac{1}{2}\sum_{i=1}^{d}\left(\exp\log\Sigma_{ii} - \log\Sigma_{ii} + \mu_i^2 - 1\right)\end{aligned}$$

Decoding: the Bernoulli case

Let $x_1, \ldots, x_n$ denote the individual pixels in the output of the decoder. They are assumed to be independent, hence

$$\begin{aligned}\log p_\theta(x|z) &= \log\prod_{i=1}^{n} p_\theta(x_i|z)\\ &= \log\prod_{i=1}^{n} y^{x_i}(1-y)^{1-x_i}\\ &= \sum_{i=1}^{n} x_i\log y + (1-x_i)\log(1-y)\end{aligned}$$

Implementing the VAE

The code for building the encoder and the decoder is shown below

```
#encoder
input_img = Input(shape=(self.img_rows,self.img_cols,self.img_channels))
input_img_flatten = Flatten()(input_img)
z = Dense(512)(input_img_flatten)
z = LeakyReLU(alpha=0.2)(z)
# mean and variance parameters
z_mean = Dense(self.z_dim)(z)
```

```
z_log_var = Dense(self.z_dim)(z)

#sample the latent vector
z_rand = Lambda(self.sampling, output_shape=(self.z_dim,))([z_mean, z_log_var])
#save the encoder
self.encoder = Model(input_img, [z_mean, z_log_var, z_rand], name='encoder')

#build decoder
latent_inputs = Input(shape=(self.z_dim,), name='z_sampling')
y = Dense(512)(latent_inputs)
output_img = LeakyReLU(alpha=0.2)(y)
output_img = Dense(n_pixels)(output_img)
output_img = Activation('sigmoid')(output_img)
self.decoder = Model(latent_inputs, output_img, name='decoder')
```

The code for computing the loss is shown below

```
# reconstruction loss
reconstruction_loss = K.sum(K.binary_crossentropy(x,y), axis=-1)
# KL divergence
kl_loss = K.sum(K.exp(z_log_var) - z_log_var + K.square(z_mean) -1, axis=-1)
# total loss
vae_loss = reconstruction_loss + kl_loss
```

Keep in mind that the loss to be **minimized** is $-\mathbb{E}_{q_\phi}\left[\log p_\theta(x|z)\right] + \mathrm{KL}\left(q_\phi(z|x)\|p_\theta(z)\right)$. The binary cross-entropy for a single sample is defined by $-(x\log y + (1-x)\log(1-y))$. Because of the extra minus sign, there is no need to put a minus sign before `reconstruction_loss` in the code.

Results of the VAE with MLP architecture trained for 30.000 epochs are shown in Figure 5. Reconstructed images are less blurry compared to the previous autoencoders. When the generator is fed with random Gaussian noise it outputs images that consistently look like hand-drawn digits, as shown in Figure 6.

Figure 5: Reconstruction results of the VAE for $d = 10$

Figure 6: Images created by the generator from Gaussian random noise