Autoencoder and Denoising Autoencoder on MNIST dataset

Deep Autoencoder on MNIST

In this section we will try to encode images from the MNIST dataset to a lower dimension and decode them as good as possible with the help of an autoencoder. We will see what is the lowest encoding dimension without loosing a lot of accuracy.

MNIST : It has a training set of $60,000$ examples, and a test set of $10,000$ examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image[1]. It contains images of handwritten digits, and their targets is the correct digit of the image.

Exe. 1 Import the mnist dataset from the `keras.datasets` and load it in `x_train, y_train, x_test, y_test` variables. Check the train and test shapes. In oder to be able using the sigmoid activation function, normalize `x_train` and `x test` according to the maximum and minimum elements of image set, for instance check `x_train[0]`.

Exe. 2 Plot some images to see your normalization results.

Notice The Encoder generally uses a series of Dense and/or Convolutional layers to encode an image into a fixed length vector that represents the image a compact form, while the Decoder uses Dense and/or Convolutional layers to convert the latent representation vector back into that same image or another modified image (see Figure 1).

Latent size is the size of the latent space: the vector holding the information after compression. This value is a crucial hyperparameter. If this value is too small, there won't be enough data for reconstruction and if the value is too large, overfitting can occur.

Exe. 3 Let's define the *LATENT_SIZE = 32*. Create an encoder model consists of a series of Dense layers, each layer is followed by a Dropout and a ReLU layer. The Dense Layers allow for the compression of the 28×28 input tensor down to the latent vector of size 32. The Dropout layers help prevent overfitting and ReLU, being the activation layer, introduces non-linearity into the mix. The final Dense(LATENT_SIZE) creates the final vector of size 32. This is how you define a encoder with a latent of size 32:

$$Dense, ReLU, Dropout, Dense, ReLU, Dropout, Dense, ReLU, Dropout, Dense, ReLU, Dropout$$

[1] http://yann.lecun.com/exdb/mnist/

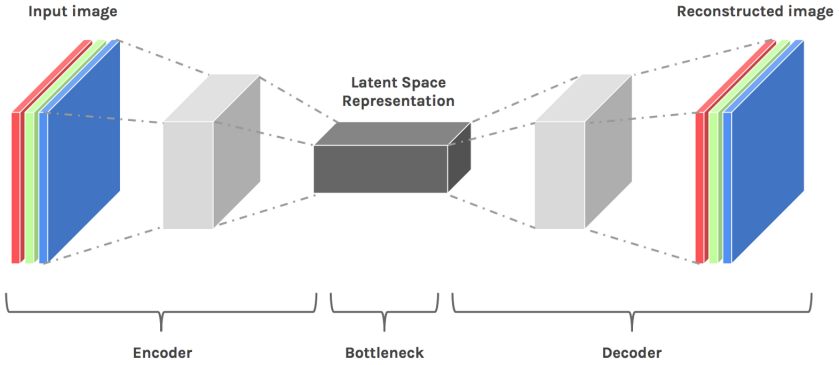

Figure 1: The image above shows an example of a simple autoencoder. In this autoencoder, you can see that the input is compressed into a latent vector (assume of size Z) and then decompressed into the same image of the same size.

Do not forget to add a final Dense layer for generating an output of *LATENT_SIZE*.
Exe. 4 Create a decoder model namely `decoder`. The decoder is essentially the same as the encoder but in reverse.

$$Dense, ReLU, Dropout, Dense, ReLU, Dropout, Dense, ReLU,$$

$$Dropout, Dense, ReLU, Dropout, Dense, Acivation, Reshape$$

Notice that the Dense layers have the same size in the reverse order as the encoder, and the final dense layer is a layer of size $28 \times 28 = 784$ that should be reshaped to 28×28 for reproducing the same image input. The sigmoid activation function output values in the range $[0, 1]$ to fit with the input scaled image data (use the Reshape from keras.layers).

To create the full model, the Keras Functional API must be used. The Functional API allows us to string together multiple models.

```
from tensorflow.keras.layers import Input
img = Input(shape = (28, 28))
```

This will create a placeholder tensor which we can feed into each network to get the output of the whole model.

```
latent_vector = encoder(img)
output = decoder(latent_vector)
```

The best part about the Keras Functional API is how readable it is. The Keras Functional API allows you to call models directly onto tensors and get the output from that tensor. By calling the `encoder` model onto the `img` tensor, we get the `latent_vector`. The same can be done with the decoder model onto the `latent_vector` which gives us the output.

```
model = Model(inputs = img, outputs = output)
model.compile("nadam", loss = "binary_crossentropy")
```

'nadam': Nesterov Adam optimizer. Much like Adam is essentially RMSprop with momentum, Nadam is RMSprop with Nesterov momentum. To create the model itself, we use the Model class and define what the inputs and outputs of the model are.

Exe. 5 implement the following code in your project:

```
from tensorflow.keras.layers import Input

img = Input(shape = (28, 28))
latent_vector = encoder(img)
output = decoder(latent_vector)
model = Model(inputs = img, outputs = output)
model.compile("nadam", loss = "binary_crossentropy")
```

To train a model, we must compile it. To compile a model, we have to choose an optimizer and a loss function. Binary Cross-Entropy is very commonly used with Autoencoders. Usually, however, binary cross-entropy is used with Binary Classifiers. Additionally, binary cross-entropy can only be used between output values in the range $[0, 1]$.

Exe. 6 The model can be trained as:

```
EPOCHS = 60
for epoch in range(EPOCHS):
  model.fit(x_train, x_train)
```

The value EPOCHS is a hyperparameter set to 60. Generally, the more epochs the better, at least until the model plateaus out.

If you code with Ipython, Jupyter or google Colab, repeatedly plotting is recommended, but the matplotlib plots are inline and can not create individual plots repeatedly. You can we create plots with 4 rows and 4 columns of subplots and choose 16 random testing data images to check how the network performs after each iteration. To check the the model in each iteration, use the following code as help in part of your code:

```
rand = x_test[np.random.randint(0, 10000, 16)].reshape((4, 4, 1,
    28, 28))
model.predict(rand[i, j])[0]
```

Check your predicted image after the training. How close is it to the original images?

Denoising autoencoder on MNIST

Denoising is one of the classic applications of autoencoders. The denoising process removes unwanted noise that corrupted the true signal.

Noise + Data $- - - >$ Denoising Autoencoder $- - - >$ Data

Given a training dataset of corrupted data as input and true signal as output, a denoising autoencoder can recover the hidden structure to generate clean data. This example has modular design. The encoder, decoder and autoencoder are 3 models that share weights. For example, after training the autoencoder, the

encoder can be used to generate latent vectors of input data for low-dim visualization like PCA or TSNE.

Exe. 7 Generate corrupted MNIST images by adding noise with normal distribution (mean = 0.5 and std= 0.5) to your `x_train` and `x_test` dataset. Fix the random seed with your student number.

Exe. 8 After adding the random generated noises to the x sets, keep only those among 0 and 1 using `np.clip()`/.

Exe. 9 Print some of your noisy images to see how they are noisy now.

Exe. 10 Check the new noisy data with the previous model. How are the results? How they are close to the real images?

Exe. 11 This time design the encoder using Conv2D networks. The model is as following:

$$Input(Inputshape), Conv2d(32), Conv2d(64), Flatten, Dense(Latent\ \ vector\ \ size)$$

Define a latent vector as 16, conv2d values with filters 32 and 64 respectively. While each conv2d has a `kernel_size = 3`, strides 2,activation of relu and 'padding' of 'same'. For instance:

```
# input should be modified by you
output =  Conv2D(filters=filter,
                 kernel_size= 3,
                 strides=2,
                 activation='relu',
                 padding='same')(input)
```

Notice check `x_train[0]`, `x_train_noisy[0]` shapes. They probably has a shape as: (28, 28). Since that we use a conv2d network, we should add an extra dimension to the input shape tensor. Because we work with images containing pixels of black and white, reshape `x_train`, `x_test`, `x_train_noisy` and `x_test_` to shape (28, 28, 1) using the following code:

```
image_size = x_train.shape[1]
x_train = np.reshape(x_train, [-1, image_size, image_size, 1])
```

To check your encoder model, make a summary of the model after implementing it.

Exe. 12 Build decoder model, based on the following model:

$$Inout(latent\ \ size), Dense(28*28*1), Conv2DTranspose(64),$$

$$Conv2DTranspose(32), Conv2DTranspose, Activation('sigmoid')$$

To generate the encoder in the reverse way, first define an Input of latent size (16), then we need a dense layer of the same size as Flatten in the encoder, in order to define a dense layer of size $28*28*1$. In order to generate the Conv2D networks in reverse, we use Conv2DTranspose with the similar parameters of the previous Exe. Pay attention to their order (they are reversed in comparison to the encoder).

Notice that, we will need to reshape the output of the Dense (28*28*1) layer to (28,28,1). We can use the following code:

```
from tensorflow.keras.layers import Reshape
x = Reshape((28, 28, 1]))(x)
```

The one before the final Conv2DTranspose gives an output tensor of size (28, 28, 32), the final Conv2DTranspose should be added in order to generate a same size image as (28, 28, 1). In order to do that, define the Conv2DTranspose as:

```
from tensorflow.keras.layers import Activation

outputs = Conv2DTranspose(filters=1,
                    kernel_size=kernel_size,
                    activation = 'sigmoid',
                    padding='same')(x)
```

After making the encoder model, print a summary of the model.
Exe. 13 Make the final model including the encoder and decoder models, such as

```
from tensorflow.keras.models import Model

autoencoder = Model(inputs, decoder(encoder(inputs)), name='
    autoencoder')
autoencoder.summary()
```

print the autoencoder model summary.
Exe. 14 Compile the model using two classical 'mse' loss function and 'adam' optimiser.

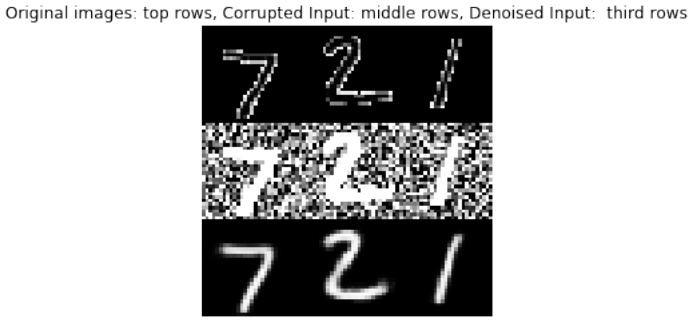

Figure 2: •

Exe. 15 Now train the autoencoder. Notice that the to be trained data here is `x_train_noisy`, while the exact data is *x_train*. It is the same for the test data set:

```
autoencoder.fit(x_train_noisy,
                x_train,
                validation_data=(x_test_noisy, x_test),
                epochs=30,
                batch_size=batch_size)
```

Exe. 16 predict the `x_test_noisy` using the trained autoencoder model.
Exe. 17 Get some of the `x_test`, `x_test_noisy` and predicted `x_test_noisy` from the trained model and show compare them in different figures. For instance similar to figure 2.
Exe. 18 Implement an encoder decoder on Labeled Faces in the Wild dataset `http://vis-www.cs.umass.edu/lfw/`.