“Neural Networks”

Shervin Halat

98131018

Homework 5

1.

DAE explanation:

Denoising autoencoders were not originally introduced to denoise images; instead, they were introduced to extract more robust features and avoid overfitting in simple autoencoders. In this case, noise was randomly added to the input data and simple autoencoder was trained to recover original denoised image from the noisy one. Hence, by this method, simple autoencoder converts to Denoising autoencoder which now is learned how to recover denoised image from the noisy input image. More precisely, to remove the noise from an image, it is important to reduce its dimensionality. Therefore, the autoencoders contain a bottleneck network (encoder) that performs compressed knowledge representation for the input. To investigate if the encoder has appropriately performed and extracted adequate features, decoder tries to carefully recreate the original image from the extracted features which will be penalized by the loss function known as reconstruction loss. Therefore, for Denoising autoencoder to perform well, it is necessary to extract generalized features in bottleneck which ignores low level features such as noise and focuses on higher level features. All this together, pushes the network towards first generate high level features from noisy images by encoder then generating denoised images by decoder using those high-level features. One last thing is that this training procedure is done by the back-propagation of the calculated reconstruction loss which may be computed by the difference between the expected value and generated value for each pixel of the original image.

Evaluation of the DAE performance:

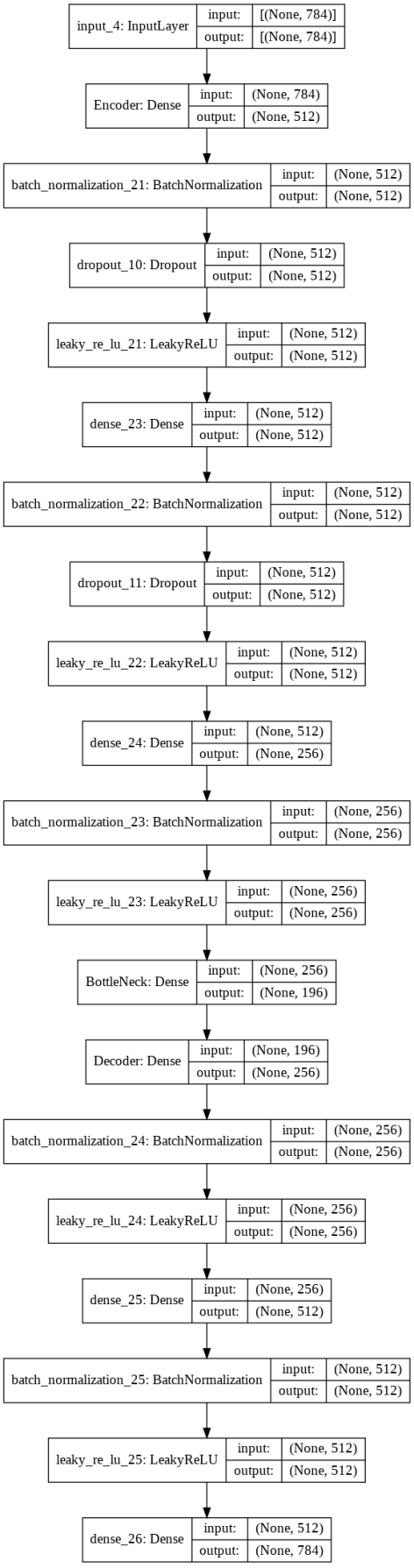
Generally, there are two types of evaluation for a DAE, namely, subjective and objective evaluation. In subjective evaluation, individuals will take part in a qualitative judgement on how well distorted images are recovered using a specific DAE. In objective evaluation, obtained feature vectors of latent space (which are generated by the encoder part of the DAE) from the dataset will be employed in a specific task. Also, the exactly the same procedure will be executed using the raw dataset. Then, the obtained performance of two operations will be compared in order to figure out if the generated feature vectors by DAE has performed well enough and are useful.

2.

Implemented ‘Denoising Autoencoder’ configuration:

In the following figures, the first layer of both encoder and decoder are marked by the names of ‘Encoder’ and ‘Decoder’ respectively. As it is shown in the following figures, most of the ‘Dense’ layers in both encoder and decoder are followed by three layers of ‘BatchNormalization’, ‘Dropout’, and ‘LeakyReLU’ activation function. The existence of two first mentioned layers (‘BatchNormalization’ and ‘Dropout’ layers) seemed critical in the case of generalization and regularization of the model’s training and therefore, in avoiding overfitting. Also, normal gaussian initial weights were considered for all ‘Dense’ layers. The bottle-neck layer has dimension of 196 which is one fourth of the flattened datapoints’ dimension (784). Lastly, sigmoid activation function was considered for the last layer of the decoder since all images were normalized into the range 0 and 1; therefore, output images had to have values between 0 and 1. Also, loss function of ‘mse’ was considered in order for it to be equal to the reconstruction loss. At the end, 100 epochs along with 256 batch size were considered for training phase. (It should be noted that the bottle-neck layer is marked by name of ‘BottleNeck’ in the following figures.)

```
Model: "model_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         [(None, 784)]             0
_________________________________________________________________
Encoder (Dense)              (None, 512)               401920
_________________________________________________________________
batch_normalization_21 (Batc (None, 512)               2048
_________________________________________________________________
dropout_10 (Dropout)         (None, 512)               0
_________________________________________________________________
leaky_re_lu_21 (LeakyReLU)   (None, 512)               0
_________________________________________________________________
dense_23 (Dense)             (None, 512)               262656
_________________________________________________________________
batch_normalization_22 (Batc (None, 512)               2048
_________________________________________________________________
dropout_11 (Dropout)         (None, 512)               0
_________________________________________________________________
leaky_re_lu_22 (LeakyReLU)   (None, 512)               0
_________________________________________________________________
dense_24 (Dense)             (None, 256)               131328
_________________________________________________________________
batch_normalization_23 (Batc (None, 256)               1024
_________________________________________________________________
leaky_re_lu_23 (LeakyReLU)   (None, 256)               0
_________________________________________________________________
BottleNeck (Dense)           (None, 196)               50372
_________________________________________________________________
Decoder (Dense)              (None, 256)               50432
_________________________________________________________________
batch_normalization_24 (Batc (None, 256)               1024
_________________________________________________________________
leaky_re_lu_24 (LeakyReLU)   (None, 256)               0
_________________________________________________________________
dense_25 (Dense)             (None, 512)               131584
_________________________________________________________________
batch_normalization_25 (Batc (None, 512)               2048
_________________________________________________________________
leaky_re_lu_25 (LeakyReLU)   (None, 512)               0
_________________________________________________________________
dense_26 (Dense)             (None, 784)               402192
=================================================================
Total params: 1,438,676
Trainable params: 1,434,580
Non-trainable params: 4,096
_________________________________________________________________
```


input_4: InputLayer
input: [(None, 784)]
output: [(None, 784)]
Encoder: Dense
input: (None, 784)
output: (None, 512)
batch_normalization_21: BatchNormalization
input: (None, 512)
output: (None, 512)
dropout_10: Dropout
input: (None, 512)
output: (None, 512)
leaky_re_lu_21: LeakyReLU
input: (None, 512)
output: (None, 512)
dense_23: Dense
input: (None, 512)
output: (None, 512)
batch_normalization_22: BatchNormalization
input: (None, 512)
output: (None, 512)
dropout_11: Dropout
input: (None, 512)
output: (None, 512)
leaky_re_lu_22: LeakyReLU
input: (None, 512)
output: (None, 512)
dense_24: Dense
input: (None, 512)
output: (None, 256)
batch_normalization_23: BatchNormalization
input: (None, 256)
output: (None, 256)
leaky_re_lu_23: LeakyReLU
input: (None, 256)
output: (None, 256)
BottleNeck: Dense
input: (None, 256)
output: (None, 196)
Decoder: Dense
input: (None, 196)
output: (None, 256)
batch_normalization_24: BatchNormalization
input: (None, 256)
output: (None, 256)
leaky_re_lu_24: LeakyReLU
input: (None, 256)
output: (None, 256)
dense_25: Dense
input: (None, 256)
output: (None, 512)
batch_normalization_25: BatchNormalization
input: (None, 512)
output: (None, 512)
leaky_re_lu_25: LeakyReLU
input: (None, 512)
output: (None, 512)
dense_26: Dense
input: (None, 512)
output: (None, 784)

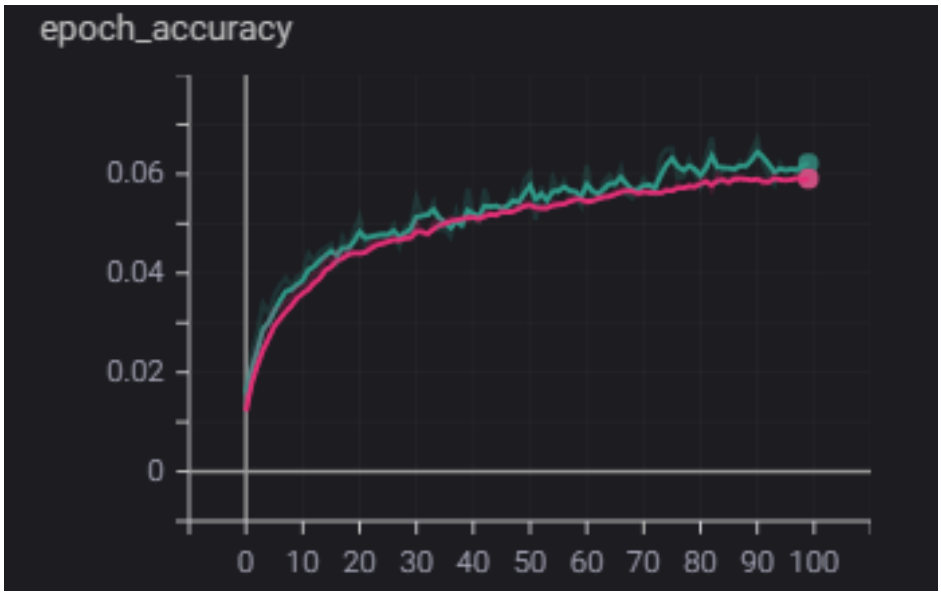
3.

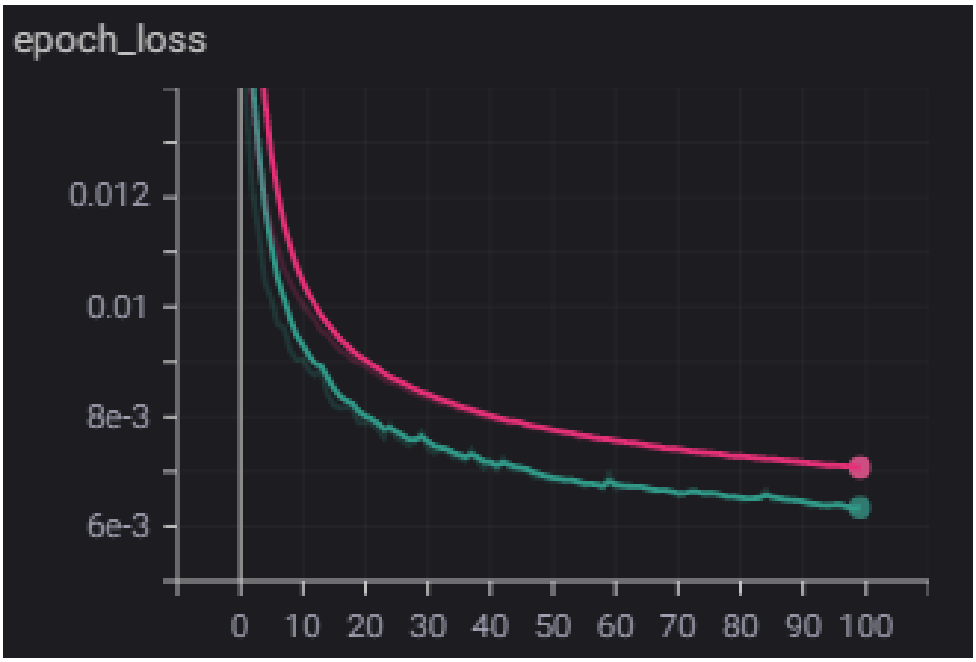
The mentioned model above was trained using training data set which was noised by gaussian noise with the following properties.

```
noise_factor = 0.1
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_train.shape)
x_test_noisy = x_test + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_test.shape)

x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)
```

Since ‘mse’ loss was considered as the models’ loss function, the obtained loss plot is the same as reconstruction loss. In the following both plots of epoch_accuracy and epoch_loss are shown below:



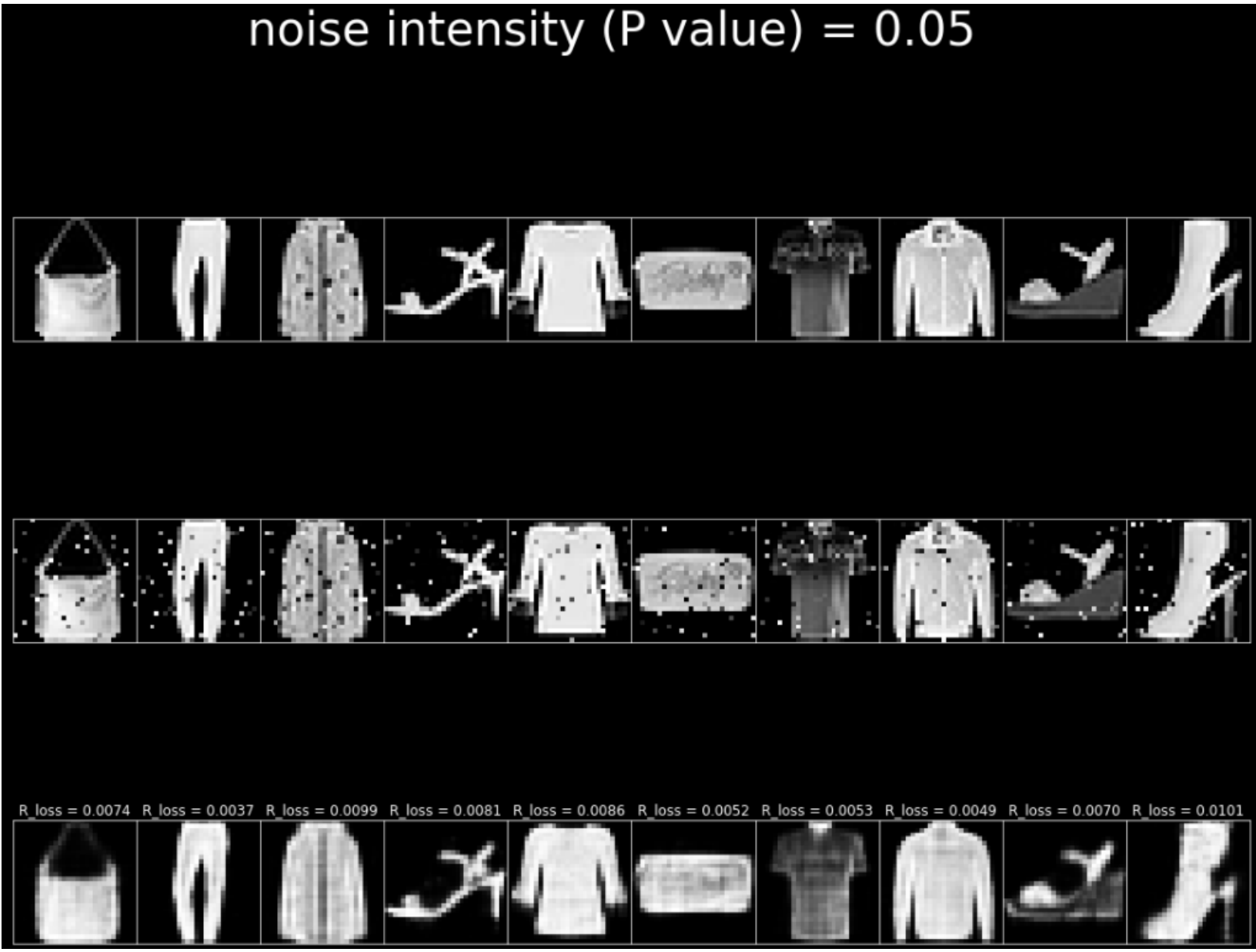


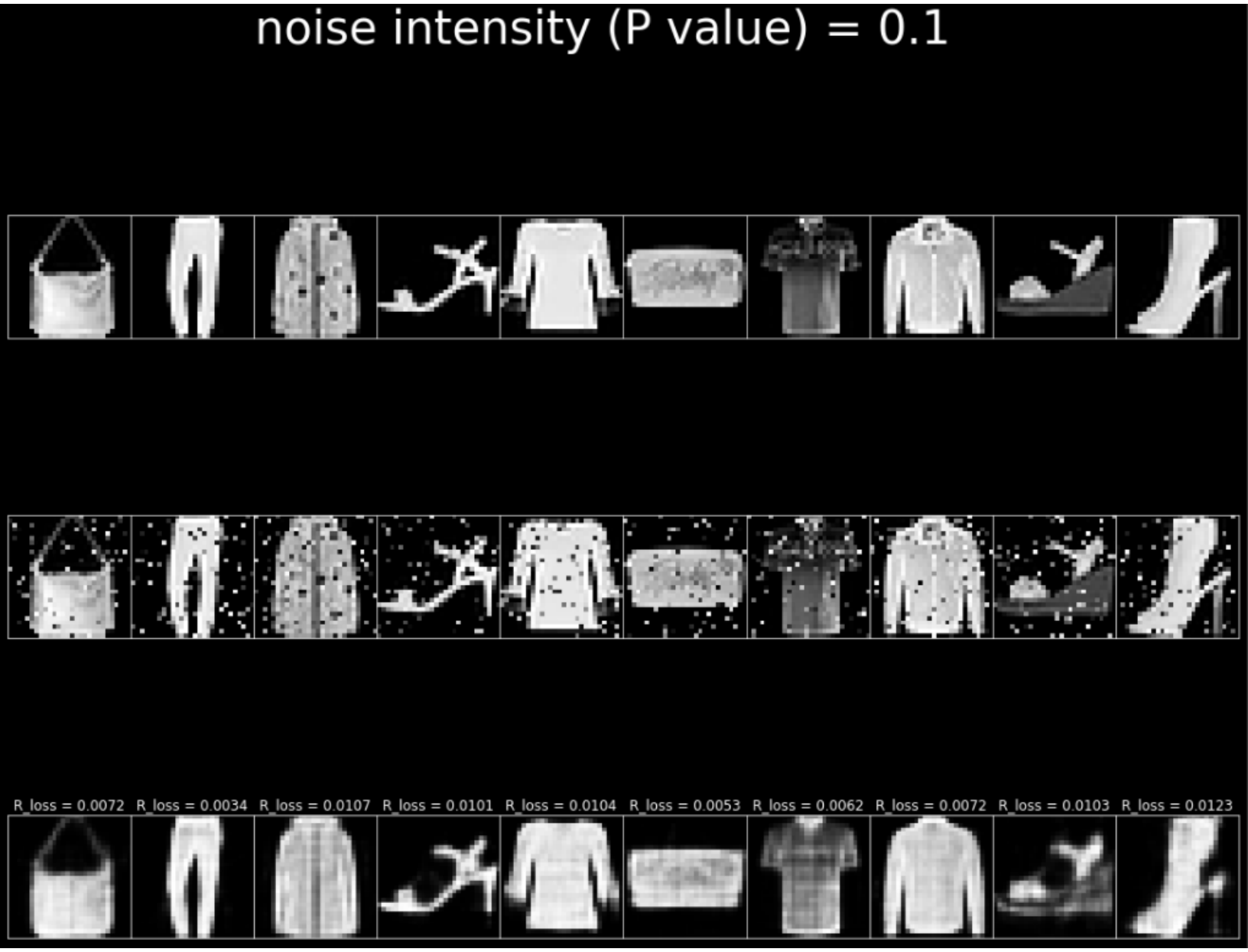
As we can see in the plots above, lastly, evaluation loss (green line) has reached to lower value compared to training loss (red line) after 100 epochs which expresses the defined models' appropriate generalization.

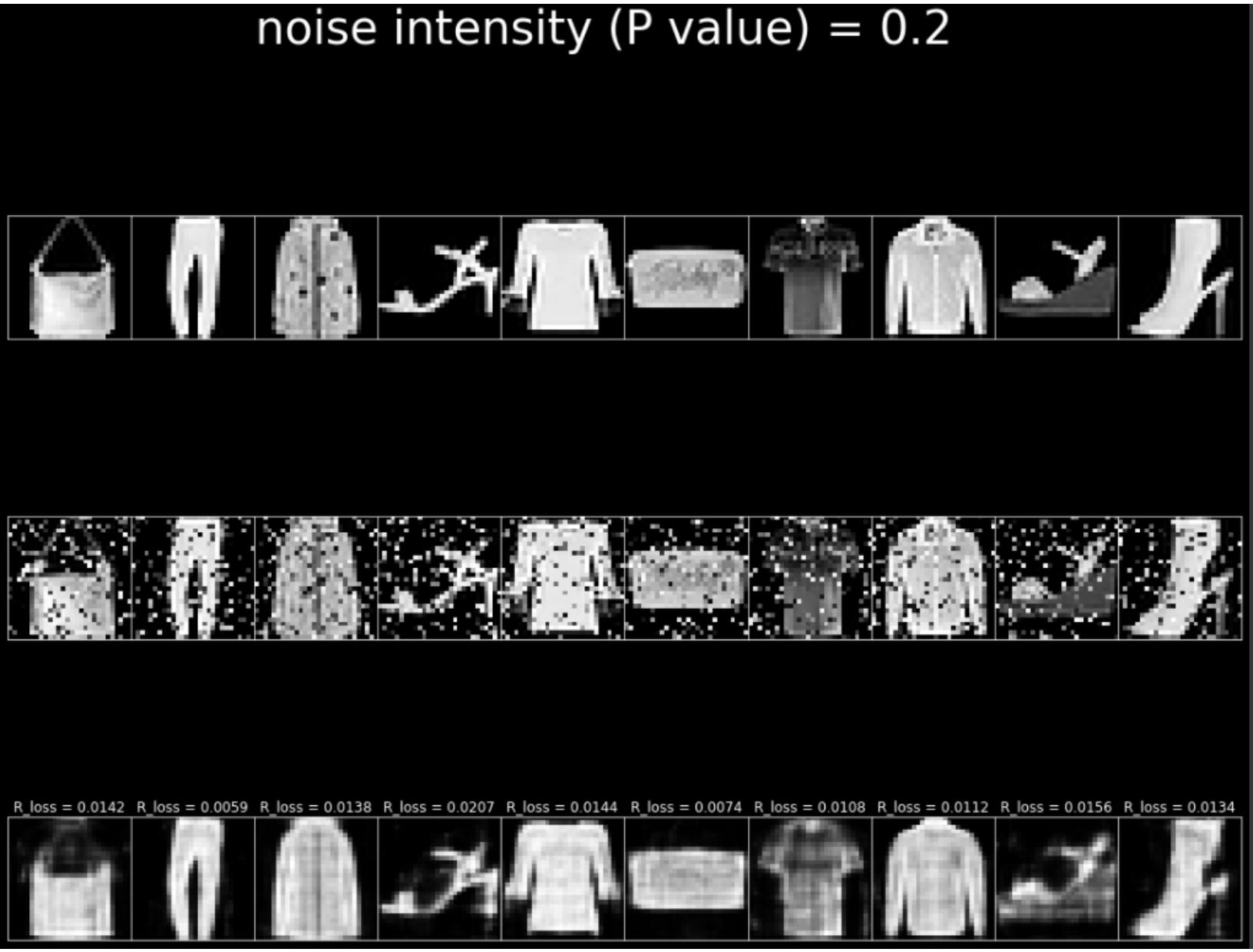
4.

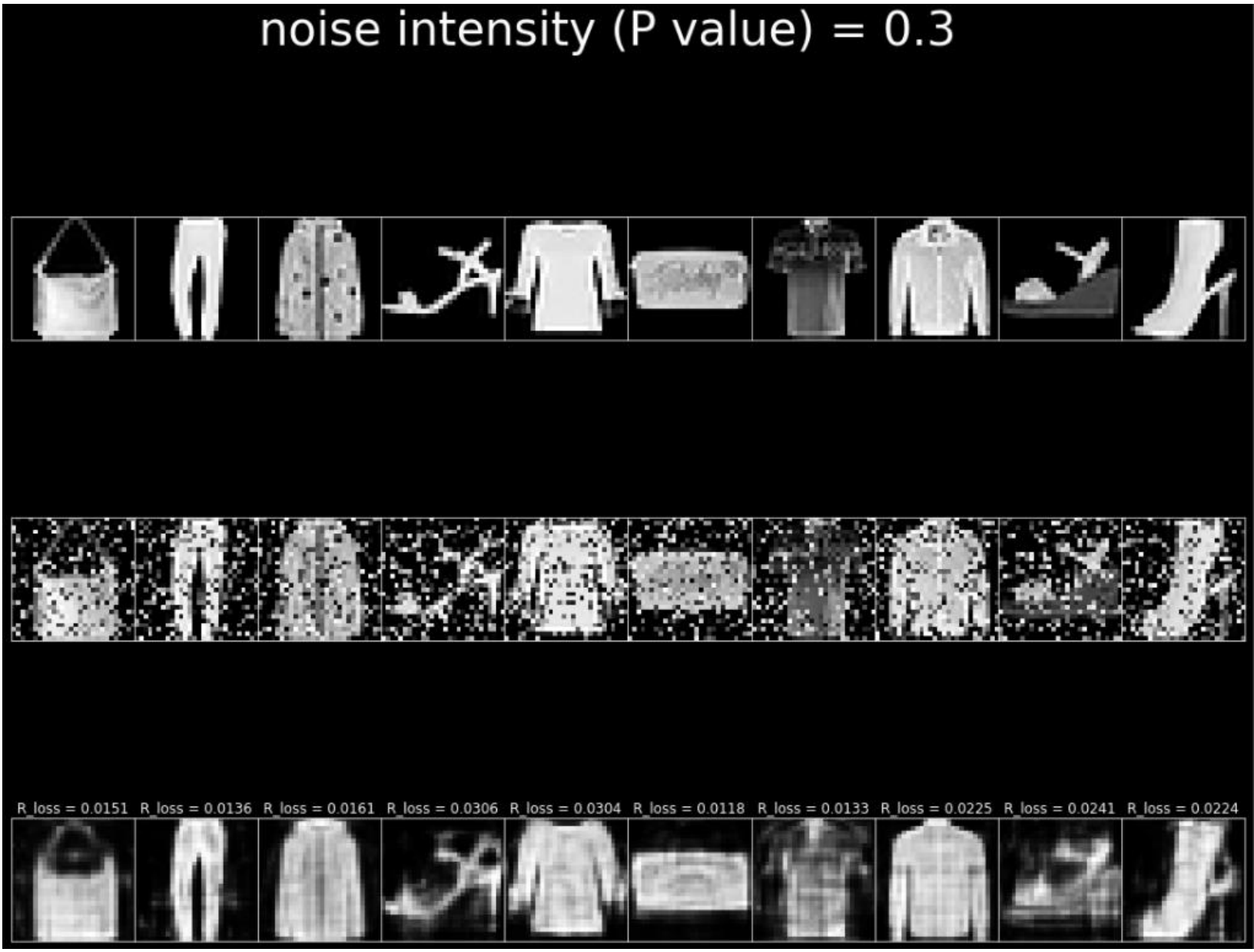
Now we are going to evaluate the performance of the defined Denoising-Autoencoder in denoising noisy input images; although the DAE model was trained by adding gaussian noise to training data set, the performance is evaluated on the images which are noised by uniform noise (therefore we expect less performance in denoising uniformly noised images compared to gaussian noised images).

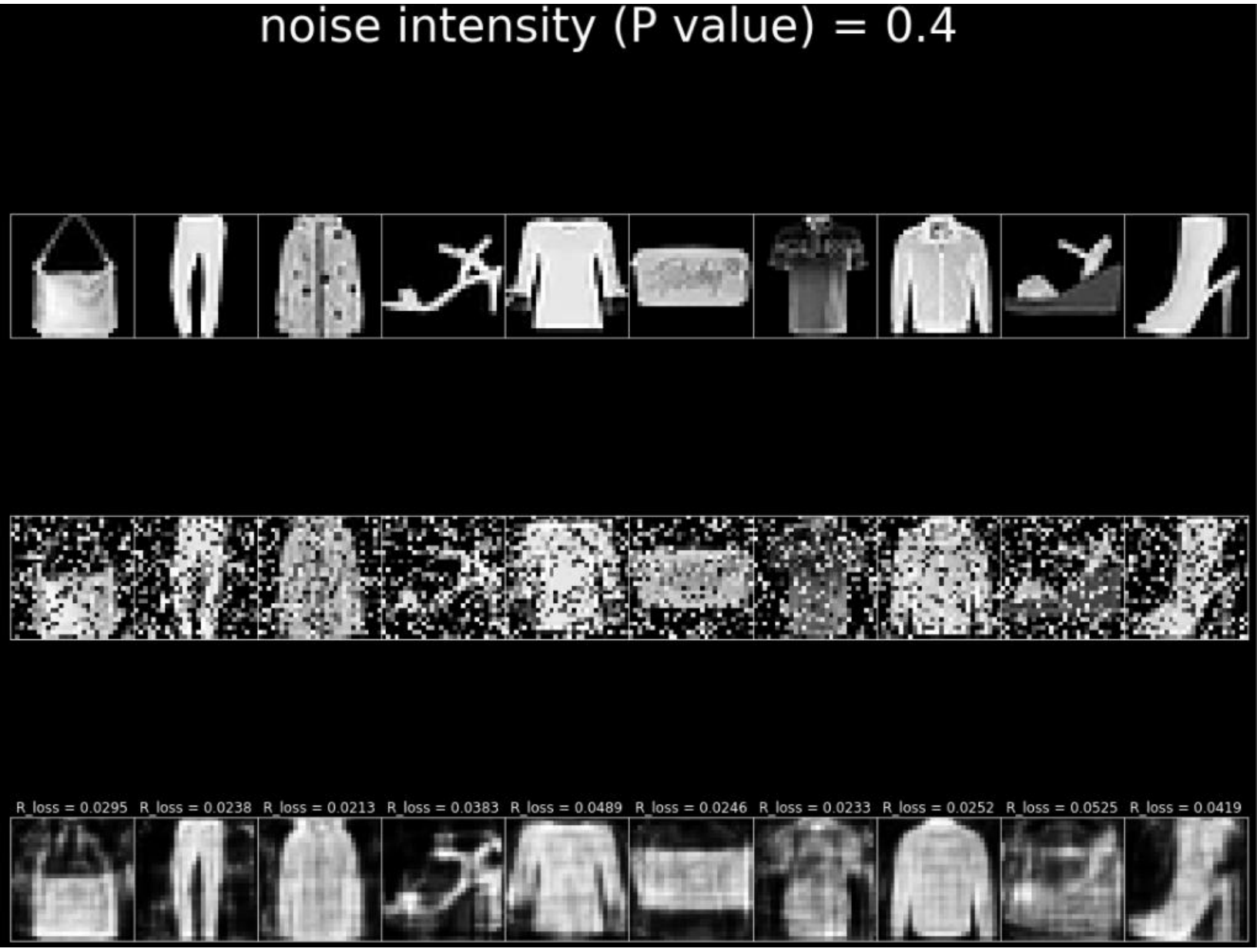
In the following figures, the obtained results in denoising images with five different noise intensities (‘P’ value) are shown. For each ‘P’ value, raw image, noisy image, and denoised image of 10 randomly selected images are shown. The reconstruction loss of each image is shown on top of the corresponding reconstructed image in the figure.


noise intensity (P value) = 0.05
R_loss = 0.0074
R_loss = 0.0037
R_loss = 0.0099
R_loss = 0.0081
R_loss = 0.0086
R_loss = 0.0052
R_loss = 0.0053
R_loss = 0.0049
R_loss = 0.0070
R_loss = 0.0101


noise intensity (P value) = 0.1
R_loss = 0.0072
R_loss = 0.0034
R_loss = 0.0107
R_loss = 0.0101
R_loss = 0.0104
R_loss = 0.0053
R_loss = 0.0062
R_loss = 0.0072
R_loss = 0.0103
R_loss = 0.0123


noise intensity (P value) = 0.2
R_loss = 0.0142
R_loss = 0.0059
R_loss = 0.0138
R_loss = 0.0207
R_loss = 0.0144
R_loss = 0.0074
R_loss = 0.0108
R_loss = 0.0112
R_loss = 0.0156
R_loss = 0.0134


noise intensity (P value) = 0.3
R_loss = 0.0151
R_loss = 0.0136
R_loss = 0.0161
R_loss = 0.0306
R_loss = 0.0304
R_loss = 0.0118
R_loss = 0.0133
R_loss = 0.0225
R_loss = 0.0241
R_loss = 0.0224

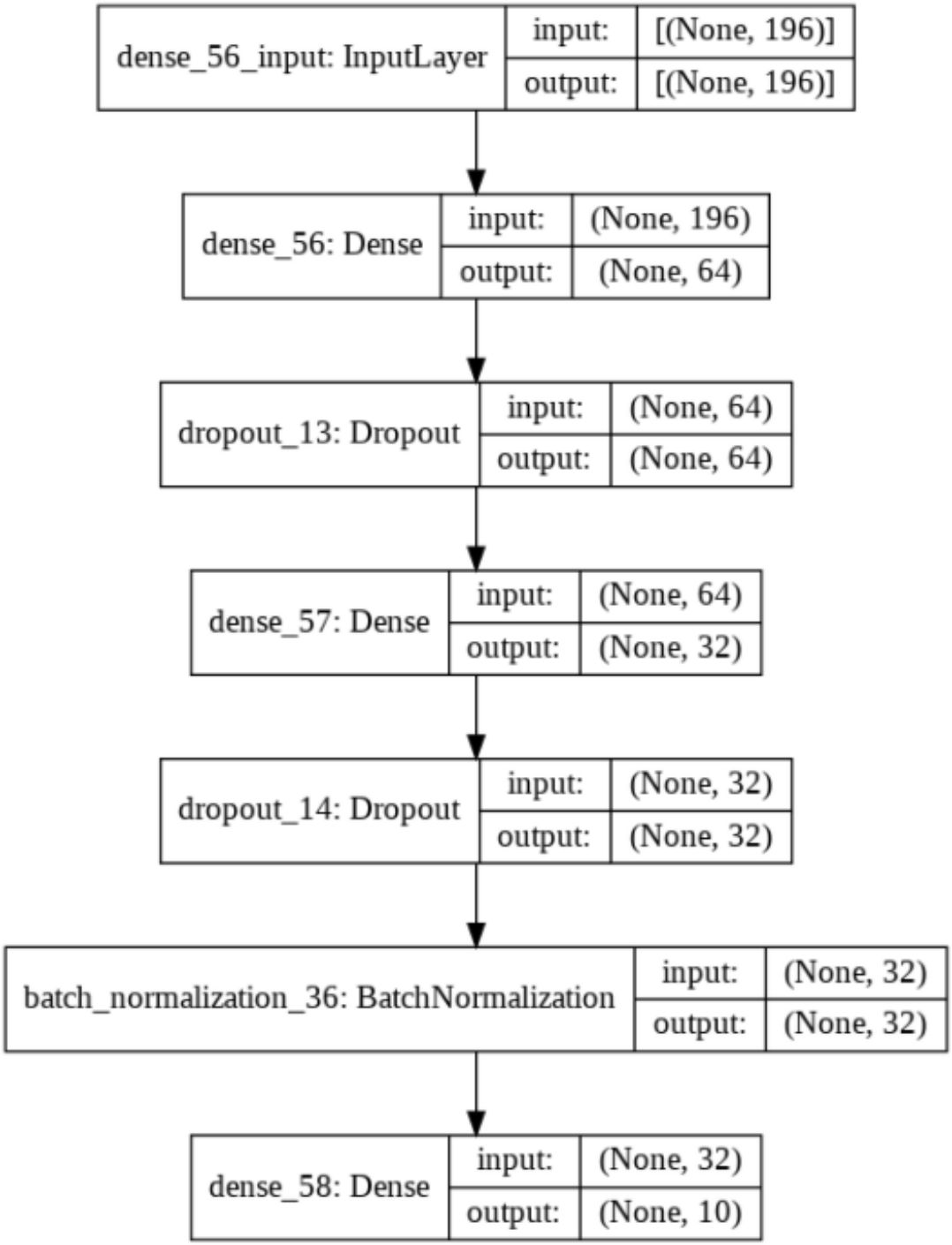

noise intensity (P value) = 0.4
R_loss = 0.0295
R_loss = 0.0238
R_loss = 0.0213
R_loss = 0.0383
R_loss = 0.0489
R_loss = 0.0246
R_loss = 0.0233
R_loss = 0.0252
R_loss = 0.0525
R_loss = 0.0419

5 & 6.

Now we are going to discard the decoder and only use the encoder in order to compress data points into codes with dimension of latent space as new feature vectors and train a classifier to classify the dataset by their corresponding generated codes.

Implemented classifier configuration:

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_56 (Dense)             (None, 64)                12608
_________________________________________________________________
dropout_13 (Dropout)         (None, 64)                0
_________________________________________________________________
dense_57 (Dense)             (None, 32)                2080
_________________________________________________________________
dropout_14 (Dropout)         (None, 32)                0
_________________________________________________________________
batch_normalization_36 (Batc (None, 32)                128
_________________________________________________________________
dense_58 (Dense)             (None, 10)                330
=================================================================
Total params: 15,146
Trainable params: 15,082
Non-trainable params: 64
_________________________________________________________________
```


dense_56_input: InputLayer
input: [(None, 196)]
output: [(None, 196)]
dense_56: Dense
input: (None, 196)
output: (None, 64)
dropout_13: Dropout
input: (None, 64)
output: (None, 64)
dense_57: Dense
input: (None, 64)
output: (None, 32)
dropout_14: Dropout
input: (None, 32)
output: (None, 32)
batch_normalization_36: BatchNormalization
input: (None, 32)
output: (None, 32)
dense_58: Dense
input: (None, 32)
output: (None, 10)

```
log_dir = "classifier/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
cb4 = keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)
cb3 = keras.callbacks.EarlyStopping(monitor='val_accuracy', verbose=1, patience=20)
classifier2 = keras.models.Sequential([
                                keras.layers.Dense(units = 64 , activation = 'relu', input_shape = ((x_train.shape[1],))),
                                keras.layers.Dropout(0.4),
                                keras.layers.Dense(units = 32 , activation = 'relu' ),
                                keras.layers.Dropout(0.4),
                                keras.layers.BatchNormalization(),
                                keras.layers.Dense(units = 10 , activation = 'softmax'),
])

classifier2.compile(optimizer = 'Adam' , loss = 'categorical_crossentropy' , metrics = ['accuracy'])
classifier2.fit(x_train, y_train , epochs=100, validation_split = 2/9, callbacks = [cb4],batch_size=128)
```

The only difference in two classifiers (question 5 and 6) is the dimension of input vector. For the classifier of codes its equal to the dimension of latent dimension (i.e. 196) and 784 for the classifier of raw data (28*28).

As we can see, ‘softmax’ activation function was considered for the last ‘Dense’ layer in order to represent the output as probabilities of each of the 10 classes; therefore, loss function of ‘categorical_crossentropy’ was considered.

Obtained results:

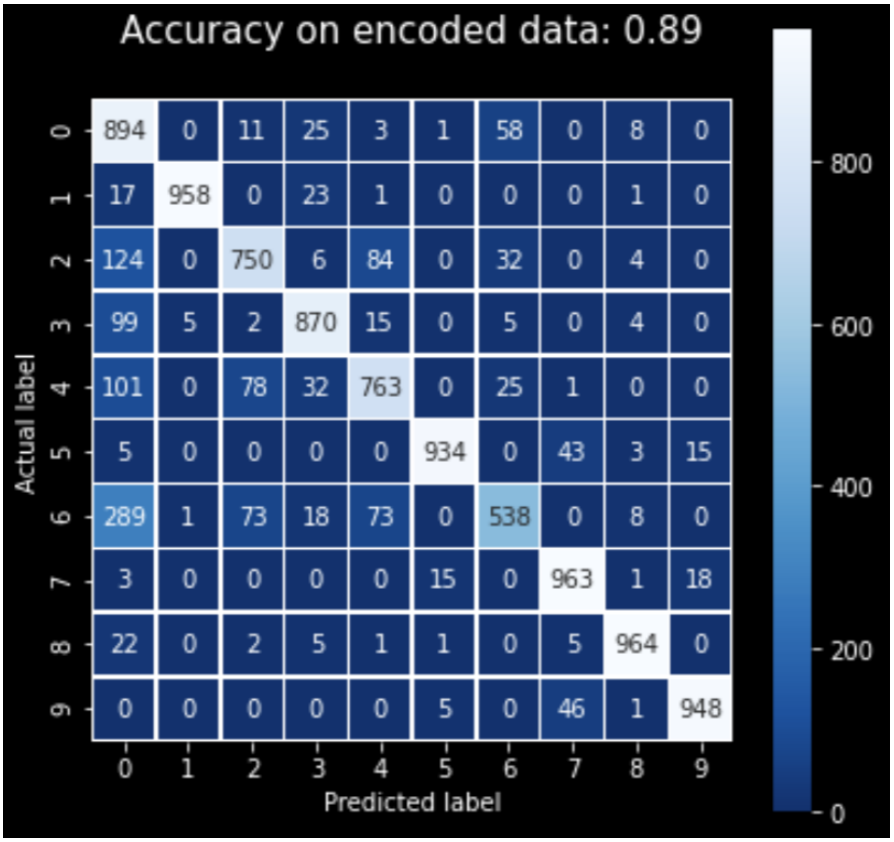
In the following, obtained results of the trained classifiers on both encoded dataset and raw dataset are shown respectively:

Classifiers’ results on encoded dataset:

```
Epoch 100/100
365/365 [==============================] - 1s 4ms/step - loss: 0.3834 - accuracy: 0.8658 - val_loss: 0.3234 - val_accuracy: 0.8841
<tensorflow.python.keras.callbacks.History at 0x7f409828abe0>
```

```
results = classifier.evaluate(codes_test,y_test)
print('classifier accuracy on encoded test data = {0:.2f}'.format(results[1]))
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.3506 - accuracy: 0.8752
classifier accuracy on encoded test data = 0.89
```

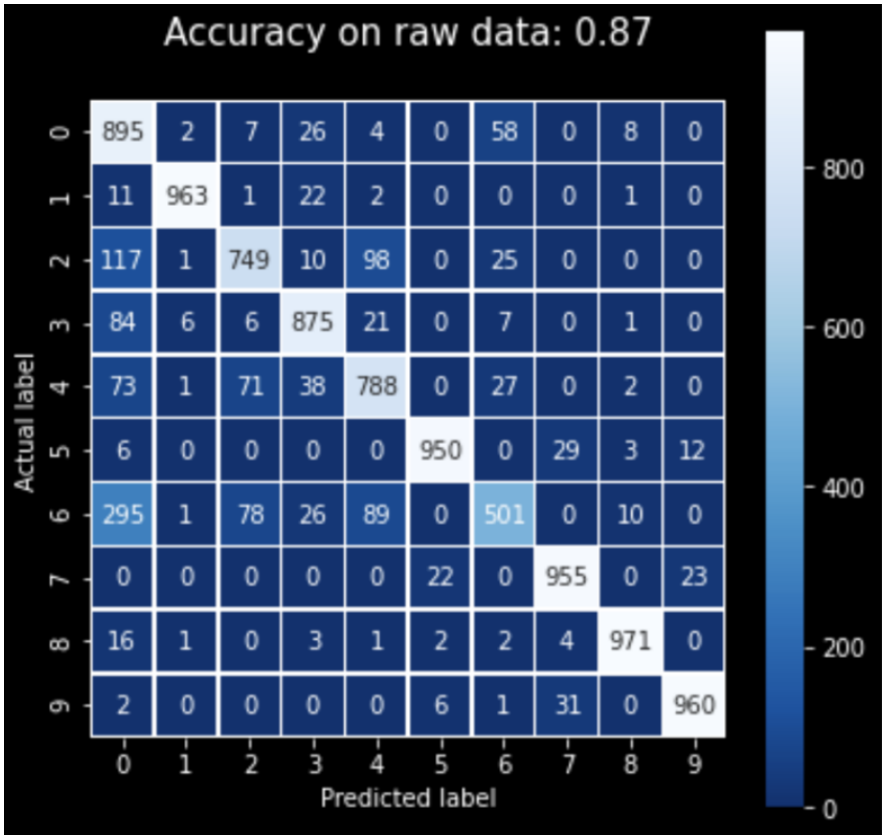


Classifiers' results on raw dataset:

```
Epoch 100/100
365/365 [==============================] - 2s 4ms/step - loss: 0.3650 - accuracy: 0.8683 - val_loss: 0.3461 - val_accuracy: 0.8791
<tensorflow.python.keras.callbacks.History at 0x7f40a7e4bc50>
```

```
1 results2 = classifier2.evaluate(x_test,y_test)
2 print('classifier accuracy on raw test data = {0:.2f}'.format(results2[1]))
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.3673 - accuracy: 0.8764
classifier accuracy on raw test data = 0.87
```



In our case, the classifier on encoded dataset has achieved a classification accuracy of 89 percent as against to the classifier on raw dataset which has achieved an accuracy of 87 percent. This suggests that although the encoder compresses the raw datapoints' dimension (784) into one fourth of their original dimension (196), encoded feature vectors adequately represent the dataset; hence, the encoder is helpful in the case of our defined classifier.