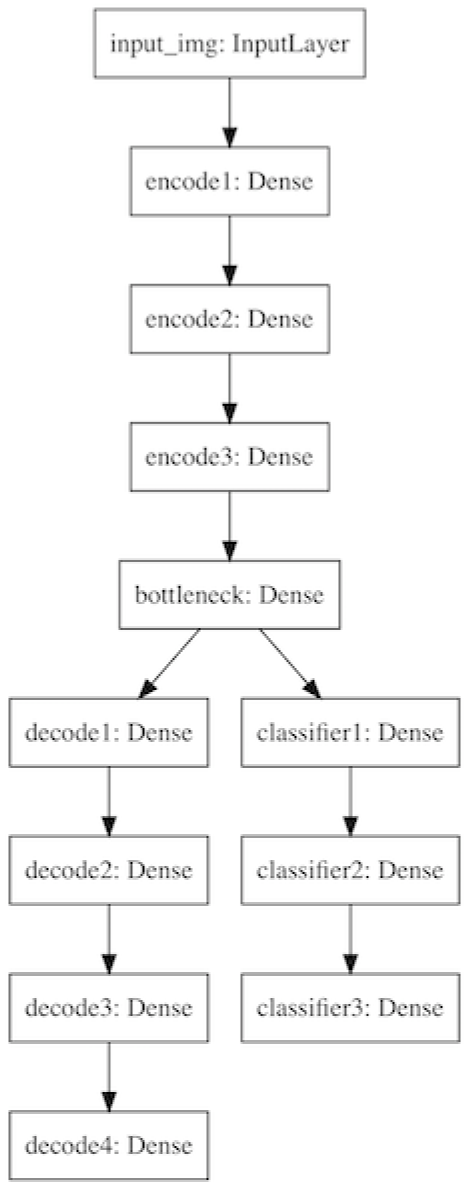
PCA and the standard autoencoder are unsupervised dimensionality reduction methods, and their learned features are not discriminative. If you build a classifier upon the low-dimenional features extracted by PCA and autoencoder, you will find the classification accuracy very poor.

Linear discriminant analysis (LDA) is a traditionally supervised dimensionality reduction method for learning low-dimensional features which are highly discriminative. Likewise, can we extend autoencoder to supervised leanring?

You are required to build and train a supervised autoencoder look like the following. You are required to add other layers properly to alleviate overfitting.



0. You will do the following:

1. Build a standard dense autoencoder, visual the low-dim features and the reconstructions, and evaluate whether the learned low-dim features are discriminative.
2. Repeat the above process by training a supervised autoencoder.

1. Data preparation

1.1. Load data

In [1]:
```
from keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.reshape(60000, 28*28).astype('float32') / 255.
x_test = x_test.reshape(10000, 28*28).astype('float32') / 255.

print('Shape of x_train: ' + str(x_train.shape))
print('Shape of x_test: ' + str(x_test.shape))
print('Shape of y_train: ' + str(y_train.shape))
print('Shape of y_test: ' + str(y_test.shape))
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-data
sets/mnist.npz (https://storage.googleapis.com/tensorflow/tf-keras-datasets/m
nist.npz)
11490434/11490434 [==============================] - 1s 0us/step
Shape of x_train: (60000, 784)
Shape of x_test: (10000, 784)
Shape of y_train: (60000,)
Shape of y_test: (10000,)
```

1.2. One-hot encode the labels

In the input, a label is a scalar in $\{0, 1, \cdots, 9\}$. One-hot encode transform such a scalar to a 10-dim vector. E.g., a scalar `y_train[j]=3` is transformed to the vector `y_train_vec[j]=[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]` .

1. Define a function `to_one_hot` that transforms an $n \times 1$ array to a $n \times 10$ matrix.
2. Apply the function to `y_train` and `y_test` .

In [2]:
```
import numpy as np

def to_one_hot(y, num_class=10):
    results = np.zeros((len(y), num_class))
    for i, label in enumerate(y):
        results[i, label] = 1.
    return results

y_train_vec = to_one_hot(y_train)
y_test_vec = to_one_hot(y_test)

print('Shape of y_train_vec: ' + str(y_train_vec.shape))
print('Shape of y_test_vec: ' + str(y_test_vec.shape))

print(y_train[0])
print(y_train_vec[0])
```

```
Shape of y_train_vec: (60000, 10)
Shape of y_test_vec: (10000, 10)
5
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

1.3. Randomly partition the training set to training and validation sets

Randomly partition the 60K training samples to 2 sets:

- a training set containing 10K samples;
- a validation set containing 50K samples. (You can use only 10K to save time.)

In [3]:
```
rand_indices = np.random.permutation(60000)
train_indices = rand_indices[0:10000]
valid_indices = rand_indices[10000:20000]

x_val = x_train[valid_indices, :]
y_val = y_train_vec[valid_indices, :]

x_tr = x_train[train_indices, :]
y_tr = y_train_vec[train_indices, :]

print('Shape of x_tr: ' + str(x_tr.shape))
print('Shape of y_tr: ' + str(y_tr.shape))
print('Shape of x_val: ' + str(x_val.shape))
print('Shape of y_val: ' + str(y_val.shape))
```

```
Shape of x_tr: (10000, 784)
Shape of y_tr: (10000, 10)
Shape of x_val: (10000, 784)
Shape of y_val: (10000, 10)
```

2. Build an unsupervised autoencoder and tune its

hyper-parameters

1. Build a dense autoencoder model
2. Your encoder should contain 3 dense layers and 1 bottlenect layer with 2 as output size.
3. Your decoder should contain 4 dense layers with 784 as output size.
4. You can choose different number of hidden units in dense layers.
5. Do not add other layers (no activation layers), you may add them in later sections.
6. Use the validation data to tune the hyper-parameters (e.g., network structure, and optimization algorithm)

   - Do NOT use test data for hyper-parameter tuning!!!
7. Try to achieve a validation loss as low as possible.
8. Evaluate the model on the test set.
9. Visualize the low-dim features and reconstructions

2.1. Build the model (20 points)

In [4]:
```
from keras.layers import *
from keras import models

input_img = Input(shape=(784,), name='input_img')

encode1 = Dense(128, activation='relu', name='encode1')(input_img)
encode2 = Dense(32, activation='relu', name='encode2')(encode1)
encode3 = Dense(8, activation='relu', name='encode3')(encode2)

bottleneck = Dense(2, activation='relu', name='bottleneck')(encode3)

decode1 = Dense(8, activation='relu', name='decode1')(bottleneck)
decode2 = Dense(32, activation='relu', name='decode2')(decode1)
decode3 = Dense(128, activation='relu', name='decode3')(decode2)
decode4 = Dense(784, activation='relu', name='decode4')(decode3)

ae = models.Model(input_img, decode4)

ae.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_img (InputLayer)      [(None, 784)]             0

 encode1 (Dense)             (None, 128)               100480

 encode2 (Dense)             (None, 32)                4128

 encode3 (Dense)             (None, 8)                 264

 bottleneck (Dense)          (None, 2)                 18

 decode1 (Dense)             (None, 8)                 24

 decode2 (Dense)             (None, 32)                288

 decode3 (Dense)             (None, 128)               4224

 decode4 (Dense)             (None, 784)               101136

=================================================================
Total params: 210,562
Trainable params: 210,562
Non-trainable params: 0
_________________________________________________________________
```

In [5]:
```
# print the network structure to a PDF file

from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot, plot_model

SVG(model_to_dot(ae, show_shapes=False).create(prog='dot', format='svg'))

plot_model(
    model=ae, show_shapes=False,
    to_file='unsupervised_ae.pdf'
)

# you can find the file "unsupervised_ae.pdf" in the current directory.
```

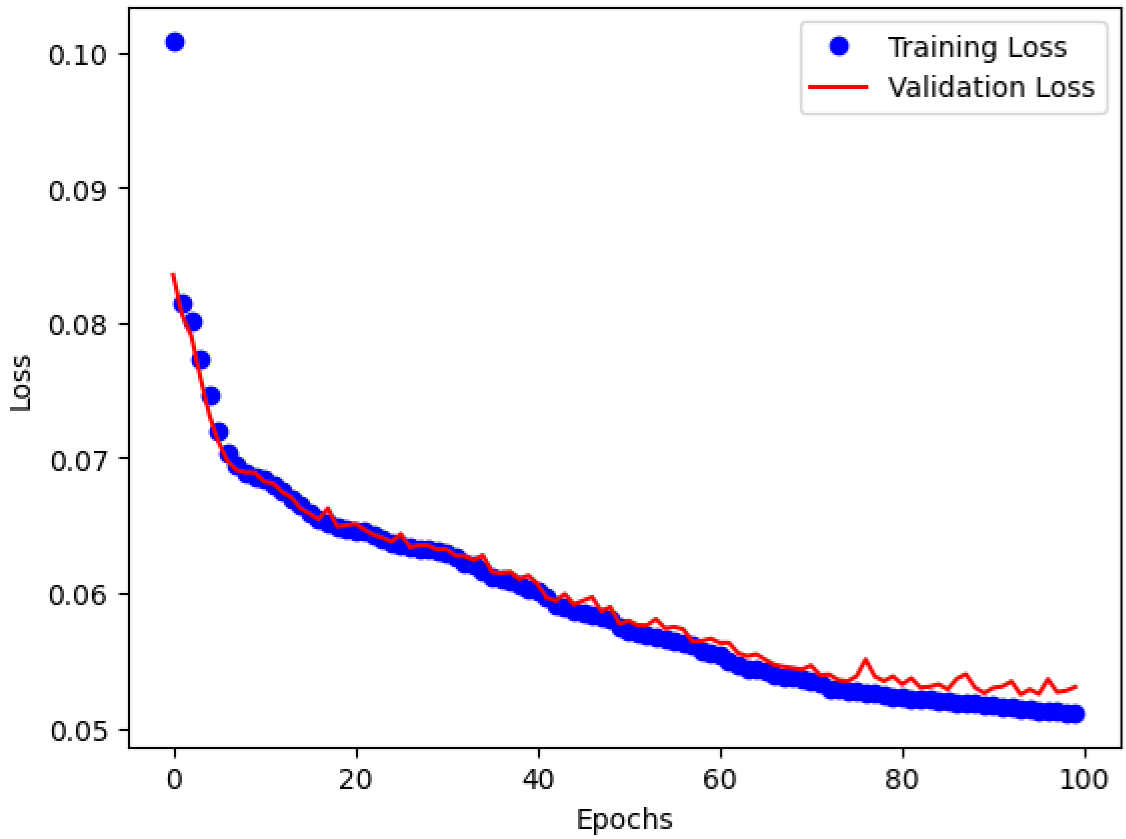
2.2. Train the model and tune the hyper-parameters (5 points)

In [6]:
```
from tensorflow.keras import optimizers

learning_rate = 1E-3 # to be tuned!

ae.compile(loss='mean_squared_error',
           optimizer=optimizers.RMSprop(learning_rate=learning_rate))
```

In [7]:
```
history = ae.fit(x_tr, x_tr,
                 batch_size=128,
                 epochs=100,
                 validation_data=(x_val, x_val))
```

```
Epoch 1/100
79/79 [==============================] - 7s 10ms/step - loss: 0.1009 - val
_loss: 0.0835
Epoch 2/100
79/79 [==============================] - 1s 9ms/step - loss: 0.0814 - val_
loss: 0.0806
Epoch 3/100
79/79 [==============================] - 1s 9ms/step - loss: 0.0801 - val_
loss: 0.0790
Epoch 4/100
79/79 [==============================] - 1s 9ms/step - loss: 0.0773 - val_
loss: 0.0759
Epoch 5/100
79/79 [==============================] - 1s 10ms/step - loss: 0.0747 - val
_loss: 0.0731
Epoch 6/100
79/79 [==============================] - 1s 7ms/step - loss: 0.0720 - val_
loss: 0.0712
Epoch 7/100
```

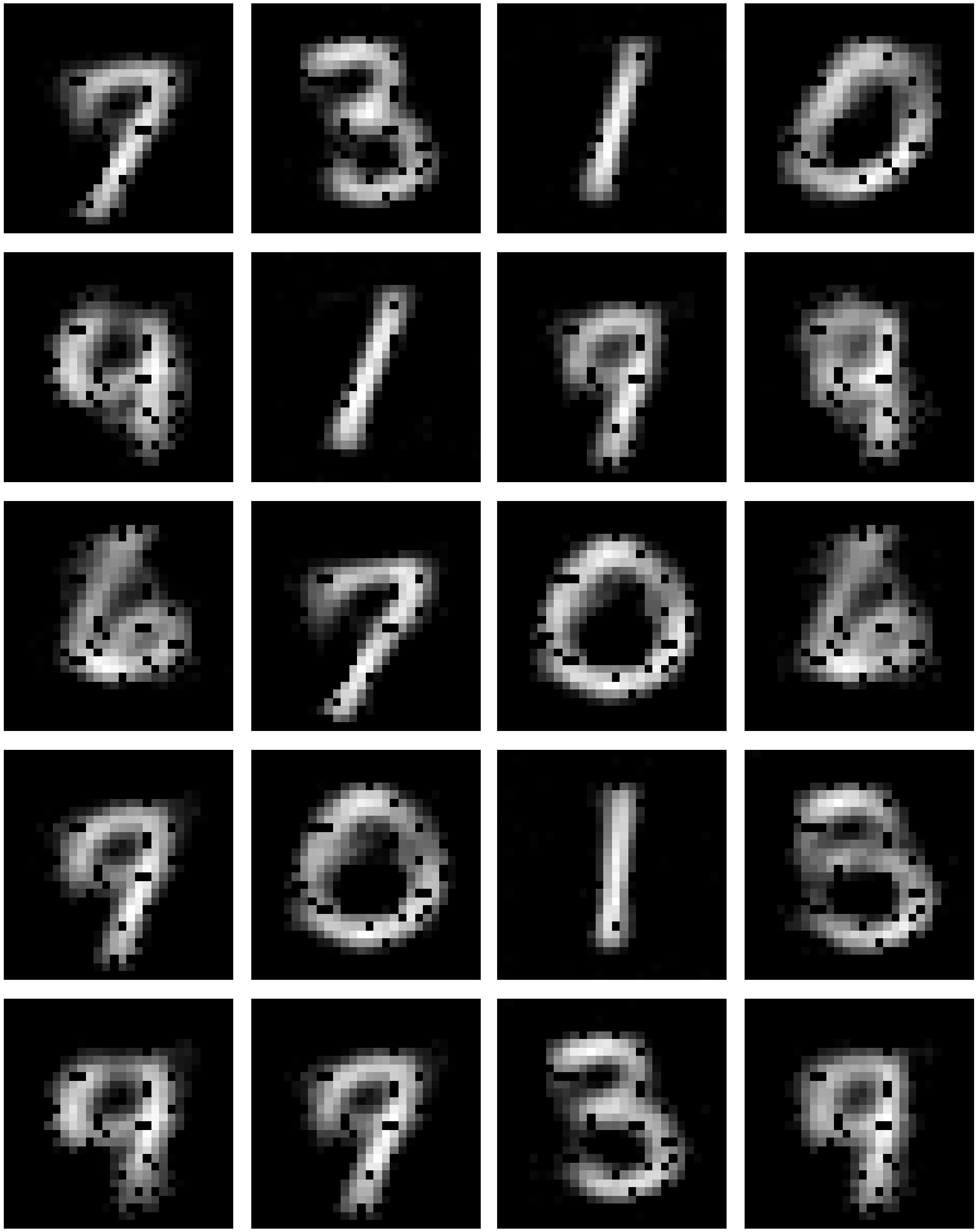
In [8]:
```
import matplotlib.pyplot as plt
%matplotlib inline

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(len(loss))

plt.plot(epochs, loss, 'bo', label='Training Loss')
plt.plot(epochs, val_loss, 'r', label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



2.3. Visualize the reconstructed test images (5 points)

In [9]:
```
ae_output = ae.predict(x_test).reshape((10000, 28, 28))

ROW = 5
COLUMN = 4

x = ae_output
fname = 'reconstruct_ae.pdf'

fig, axes = plt.subplots(nrows=ROW, ncols=COLUMN, figsize=(8, 10))
for ax, i in zip(axes.flat, np.arange(ROW*COLUMN)):
    image = x[i].reshape(28, 28)
    ax.imshow(image, cmap='gray')
    ax.axis('off')

plt.tight_layout()
plt.savefig(fname)
plt.show()
```

```
313/313 [==============================] - 1s 3ms/step
```



2.4. Evaluate the model on the test set

Do NOT used the test set until now. Make sure that your model parameters and hyper-parameters are independent of the test set.

In [10]:
```
loss = ae.evaluate(x_test, x_test)
print('loss = ' + str(loss))
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.0532
loss = 0.05320274084806442
```

2.5. Visualize the low-dimensional features

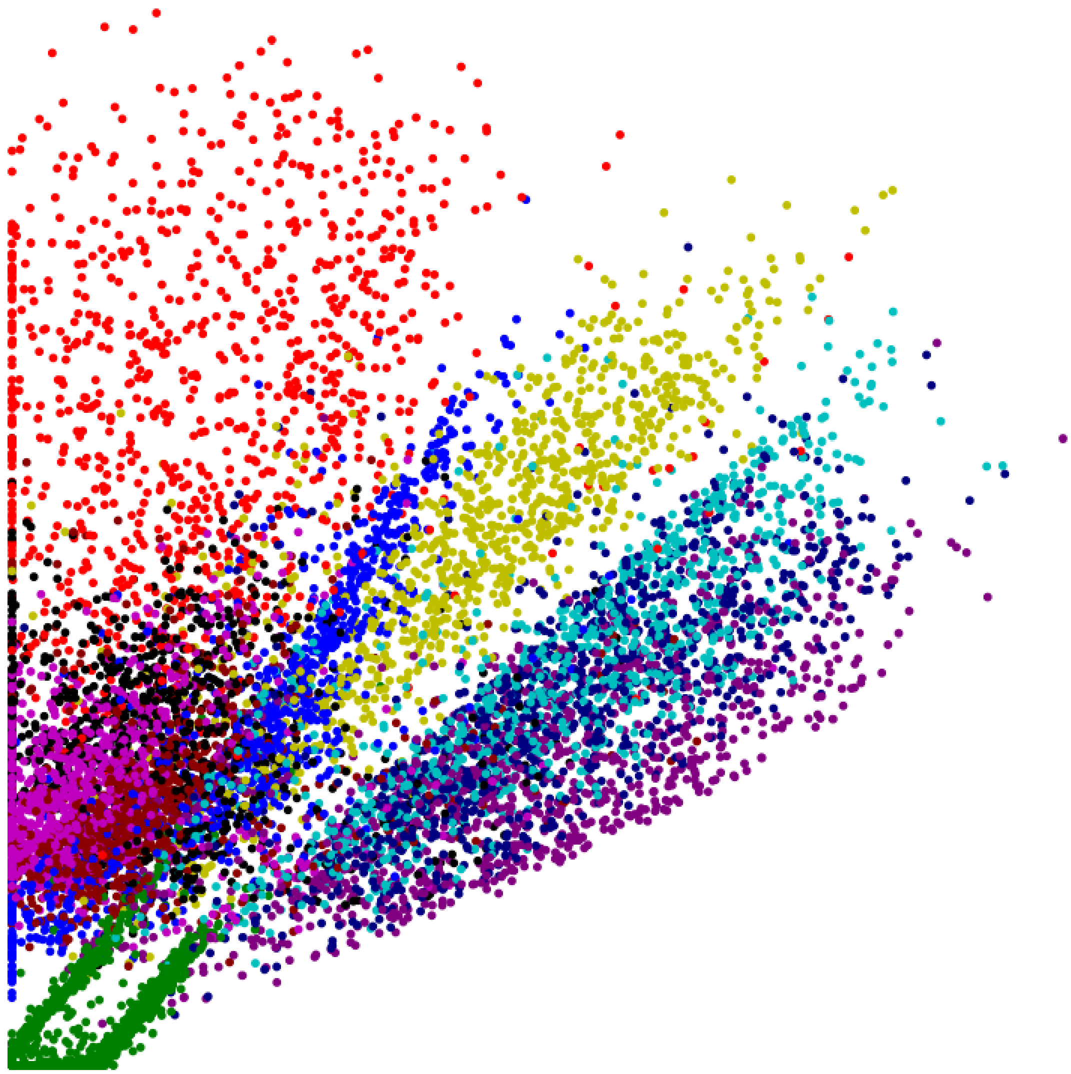
In [11]:
```
# build the encoder network
ae_encoder = models.Model(input_img, bottleneck)
ae_encoder.summary()
```

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_img (InputLayer)      [(None, 784)]             0         
                                                                 
 encode1 (Dense)             (None, 128)               100480    
                                                                 
 encode2 (Dense)             (None, 32)                4128      
                                                                 
 encode3 (Dense)             (None, 8)                 264       
                                                                 
 bottleneck (Dense)          (None, 2)                 18        
                                                                 
=================================================================
Total params: 104,890
Trainable params: 104,890
Non-trainable params: 0
_________________________________________________________________
```

In [12]:
```
# extract low-dimensional features from the test data
encoded_test = ae_encoder.predict(x_test)
print('Shape of encoded_test: ' + str(encoded_test.shape))
```

```
313/313 [==============================] - 1s 1ms/step
Shape of encoded_test: (10000, 2)
```

In [13]:
```
colors = np.array(['r', 'g', 'b', 'm', 'c', 'k', 'y', 'purple', 'darkred', 'na
colors_test = colors[y_test]


import matplotlib.pyplot as plt
%matplotlib inline

fig = plt.figure(figsize=(8, 8))
plt.scatter(encoded_test[:, 0], encoded_test[:, 1], s=10, c=colors_test, edgec
plt.axis('off')
plt.tight_layout()
fname = 'ae_code.pdf'
plt.savefig(fname)
```



Remark:

Judging from the visualization, the low-dim features seems not discriminative, as 2D features from different classes are mixed. Let quantatively find out whether they are discriminative.

3. Are the learned low-dim features discriminative? (10 points)

To find the answer, lets train a classifier on the training set (the extracted 2-dim features) and evaluation on the test set.

In [14]:
```
# extract the 2D features from the training, validation, and test samples
f_tr = ae_encoder.predict(x_tr)
f_val = ae_encoder.predict(x_val)
f_te = ae_encoder.predict(x_test)

print('Shape of f_tr: ' + str(f_tr.shape))
print('Shape of f_te: ' + str(f_te.shape))
```

```
313/313 [==============================] - 0s 1ms/step
313/313 [==============================] - 0s 1ms/step
313/313 [==============================] - 0s 1ms/step
Shape of f_tr: (10000, 2)
Shape of f_te: (10000, 2)
```

In [15]:
```
from keras.layers import Dense, Input
from keras import models

input_feat = Input(shape=(2,))

hidden1 = Dense(128, activation='relu')(input_feat)
hidden2 = Dense(128, activation='relu')(hidden1)
output = Dense(10, activation='softmax')(hidden2)

classifier = models.Model(input_feat, output)

classifier.summary()
```

```
Model: "model_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 2)]               0

 dense (Dense)               (None, 128)               384

 dense_1 (Dense)             (None, 128)               16512

 dense_2 (Dense)             (None, 10)                1290

=================================================================
Total params: 18,186
Trainable params: 18,186
Non-trainable params: 0
_________________________________________________________________
```

In [16]:
```
classifier.compile(loss='categorical_crossentropy',
                  optimizer=optimizers.RMSprop(learning_rate=1E-4),
                  metrics=['acc'])

history = classifier.fit(f_tr, y_tr,
                        batch_size=32,
                        epochs=30,
                        validation_data=(f_val, y_val))
```

```
Epoch 1/30
313/313 [==============================] - 3s 6ms/step - loss: 1.9223 - acc:
0.2946 - val_loss: 1.7327 - val_acc: 0.3309
Epoch 2/30
313/313 [==============================] - 2s 7ms/step - loss: 1.6437 - acc:
0.3635 - val_loss: 1.5942 - val_acc: 0.3789
Epoch 3/30
313/313 [==============================] - 2s 6ms/step - loss: 1.5404 - acc:
0.4078 - val_loss: 1.5139 - val_acc: 0.4160
Epoch 4/30
313/313 [==============================] - 2s 5ms/step - loss: 1.4612 - acc:
0.4630 - val_loss: 1.4293 - val_acc: 0.4890
Epoch 5/30
313/313 [==============================] - 2s 5ms/step - loss: 1.3899 - acc:
0.5159 - val_loss: 1.3628 - val_acc: 0.5239
Epoch 6/30
313/313 [==============================] - 2s 5ms/step - loss: 1.3313 - acc:
0.5370 - val_loss: 1.3125 - val_acc: 0.5213
Epoch 7/30
313/313 [==============================] - 2s 5ms/step - loss: 1.2875 - acc:
0.5484 - val_loss: 1.2724 - val_acc: 0.5488
Epoch 8/30
313/313 [==============================] - 2s 5ms/step - loss: 1.2547 - acc:
0.5584 - val_loss: 1.2461 - val_acc: 0.5530
Epoch 9/30
313/313 [==============================] - 2s 6ms/step - loss: 1.2300 - acc:
0.5682 - val_loss: 1.2264 - val_acc: 0.5484
Epoch 10/30
313/313 [==============================] - 3s 9ms/step - loss: 1.2110 - acc:
0.5684 - val_loss: 1.2084 - val_acc: 0.5548
Epoch 11/30
313/313 [==============================] - 2s 5ms/step - loss: 1.1957 - acc:
0.5705 - val_loss: 1.1967 - val_acc: 0.5675
Epoch 12/30
313/313 [==============================] - 2s 5ms/step - loss: 1.1841 - acc:
0.5748 - val_loss: 1.1858 - val_acc: 0.5581
Epoch 13/30
313/313 [==============================] - 2s 5ms/step - loss: 1.1744 - acc:
0.5813 - val_loss: 1.1878 - val_acc: 0.5567
Epoch 14/30
313/313 [==============================] - 2s 5ms/step - loss: 1.1680 - acc:
0.5830 - val_loss: 1.1721 - val_acc: 0.5651
Epoch 15/30
313/313 [==============================] - 2s 5ms/step - loss: 1.1610 - acc:
0.5824 - val_loss: 1.1676 - val_acc: 0.5684
Epoch 16/30
313/313 [==============================] - 2s 6ms/step - loss: 1.1554 - acc:
0.5831 - val_loss: 1.1662 - val_acc: 0.5611
Epoch 17/30
313/313 [==============================] - 2s 7ms/step - loss: 1.1499 - acc:
0.5841 - val_loss: 1.1641 - val_acc: 0.5673
Epoch 18/30
313/313 [==============================] - 2s 5ms/step - loss: 1.1469 - acc:
0.5842 - val_loss: 1.1556 - val_acc: 0.5702
Epoch 19/30
313/313 [==============================] - 2s 5ms/step - loss: 1.1422 - acc:
0.5858 - val_loss: 1.1548 - val_acc: 0.5662
```

```
Epoch 20/30
313/313 [==============================] - 2s 5ms/step - loss: 1.1391 - acc:
0.5850 - val_loss: 1.1508 - val_acc: 0.5723
Epoch 21/30
313/313 [==============================] - 2s 5ms/step - loss: 1.1355 - acc:
0.5865 - val_loss: 1.1457 - val_acc: 0.5715
Epoch 22/30
313/313 [==============================] - 2s 5ms/step - loss: 1.1314 - acc:
0.5866 - val_loss: 1.1460 - val_acc: 0.5686
Epoch 23/30
313/313 [==============================] - 2s 5ms/step - loss: 1.1288 - acc:
0.5914 - val_loss: 1.1445 - val_acc: 0.5742
Epoch 24/30
313/313 [==============================] - 3s 9ms/step - loss: 1.1259 - acc:
0.5880 - val_loss: 1.1398 - val_acc: 0.5741
Epoch 25/30
313/313 [==============================] - 2s 5ms/step - loss: 1.1227 - acc:
0.5904 - val_loss: 1.1372 - val_acc: 0.5727
Epoch 26/30
313/313 [==============================] - 2s 5ms/step - loss: 1.1190 - acc:
0.5872 - val_loss: 1.1348 - val_acc: 0.5742
Epoch 27/30
313/313 [==============================] - 2s 5ms/step - loss: 1.1161 - acc:
0.5914 - val_loss: 1.1344 - val_acc: 0.5703
Epoch 28/30
313/313 [==============================] - 2s 5ms/step - loss: 1.1137 - acc:
0.5927 - val_loss: 1.1368 - val_acc: 0.5724
Epoch 29/30
313/313 [==============================] - 2s 5ms/step - loss: 1.1126 - acc:
0.5905 - val_loss: 1.1319 - val_acc: 0.5776
Epoch 30/30
313/313 [==============================] - 2s 5ms/step - loss: 1.1098 - acc:
0.5927 - val_loss: 1.1287 - val_acc: 0.5703
```

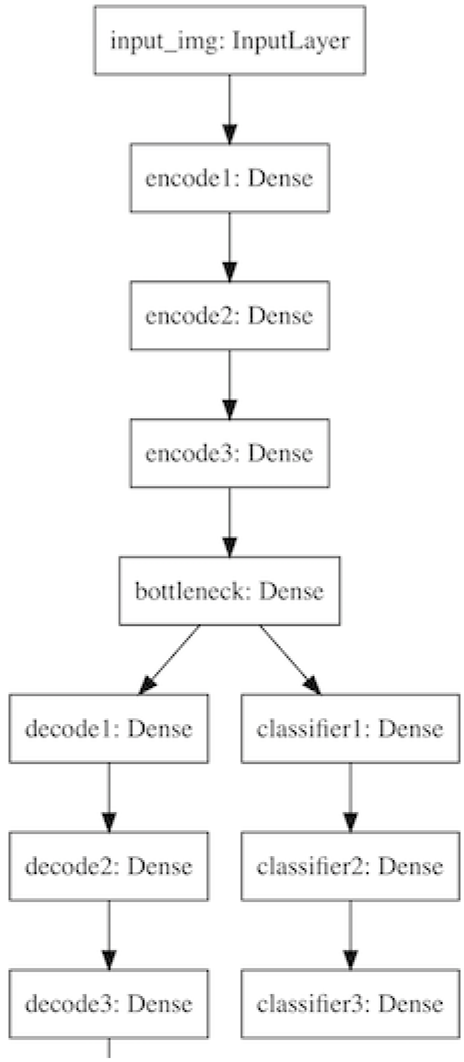
Conclusion

Using the 2D features, the validation accuracy is 60~70%. Recall that using the original data, the accuracy is about 97%. Obviously, the 2D features are not very discriminative.

We are going to build a supervised autoencode model for learning low-dimensional discriminative features.

4. Build a supervised autoencoder model

You are required to build and train a supervised autoencoder look like the following. (Not necessary the same. You can use convolutional layers as well.) You are required to add other layers properly to alleviate overfitting.


input_img: InputLayer
encode1: Dense
encode2: Dense
encode3: Dense
bottleneck: Dense
decode1: Dense
classifier1: Dense
decode2: Dense
classifier2: Dense
decode3: Dense
classifier3: Dense

4.1. Build the network (30 points)

In [17]:
```
# build the supervised autoencoder network
from keras.layers import Dense, Input, Activation, BatchNormalization, Dropout
from keras import models

input_img = Input(shape=(784,), name='input_img')

# encoder network
encode1 = Dense(128, name = 'encode1')(input_img)
encode1 = BatchNormalization()(encode1)
encode1 = Activation('relu')(encode1)
encode1 = Dropout(0.2)(encode1)

encode2 = Dense(32, name = 'encode2')(encode1)
encode2 = BatchNormalization()(encode2)
encode2 = Activation('relu')(encode2)
encode2 = Dropout(0.2)(encode2)

encode3 = Dense(8, name = 'encode3')(encode2)
encode3 = BatchNormalization()(encode3)
encode3 = Activation('relu')(encode3)

# The width of the bottleneck layer must be exactly 2.
bottleneck = Dense(2, name = 'bottleneck')(encode3)
bottleneck = BatchNormalization()(bottleneck)
bottleneck = Activation('relu')(bottleneck)

# decoder network
input_dec = Input(shape=(2,))
decode1 = Dense(8, name='decode1')(input_dec)
decode1 = BatchNormalization()(decode1)
decode1 = Activation('relu')(decode1)
#decode1 = Dropout(0.2)(decode1)

decode2 = Dense(32, name='decode2')(decode1)
decode2 = BatchNormalization()(decode2)
decode2 = Activation('relu')(decode2)
decode2 = Dropout(0.4)(decode2)

decode3 = Dense(128, name='decode3')(decode2)
decode3 = BatchNormalization()(decode3)
decode3 = Activation('relu')(decode3)
decode3 = Dropout(0.2)(decode3)

decode4 = Dense(784, name='decode4')(decode3)
decode4 = BatchNormalization()(decode4)
decode4 = Activation('relu')(decode4)


# build a classifier upon the bottleneck layer
input_feat = Input(shape=(2,))
hidden1 = Dense(128)(input_feat)
hidden1 = BatchNormalization()(hidden1)
hidden1 = Activation('relu')(hidden1)
hidden1 = Dropout(0.2)(hidden1)

hidden2 = Dense(128)(hidden1)
hidden2 = BatchNormalization()(hidden2)
```

```
hidden2 = Activation('relu')(hidden2)
hidden2 = Dropout(0.5)(hidden2)

output = Dense(10, activation='softmax')(hidden2)



sae_encoder = models.Model(input_img, bottleneck)
sae_decoder = models.Model(input_dec, decode4)
classifer = models.Model(input_feat, output)

sae_bottleneck = sae_encoder(input_img)
final_decode = sae_decoder(sae_bottleneck)
final_classifier = classifer(sae_bottleneck)
```

In [18]:

```
# connect the input and the two outputs
sae = models.Model(input_img, [final_decode, final_classifier])

sae.summary()
```

```
Model: "model_6"
_____________________________________________________________________________
_____________________
 Layer (type)                   Output Shape         Param #     Connected to
=============================================================================
=====================
 input_img (InputLayer)         [(None, 784)]        0           []

 model_3 (Functional)           (None, 2)            105570      ['input_img
[0][0]']

 model_4 (Functional)           (None, 784)          109480      ['model_3[0]
[0]']

 model_5 (Functional)           (None, 10)           19210       ['model_3[0]
[0]']

=============================================================================
=====================
Total params: 234,260
Trainable params: 231,504
Non-trainable params: 2,756
_____________________________________________________________________________
_____________________
```

In [19]:
```
# print the network structure to a PDF file

from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot, plot_model

SVG(model_to_dot(sae, show_shapes=False).create(prog='dot', format='svg'))

plot_model(
    model=sae, show_shapes=False,
    to_file='supervised_ae.pdf'
)

# you can find the file "supervised_ae.pdf" in the current directory.
```

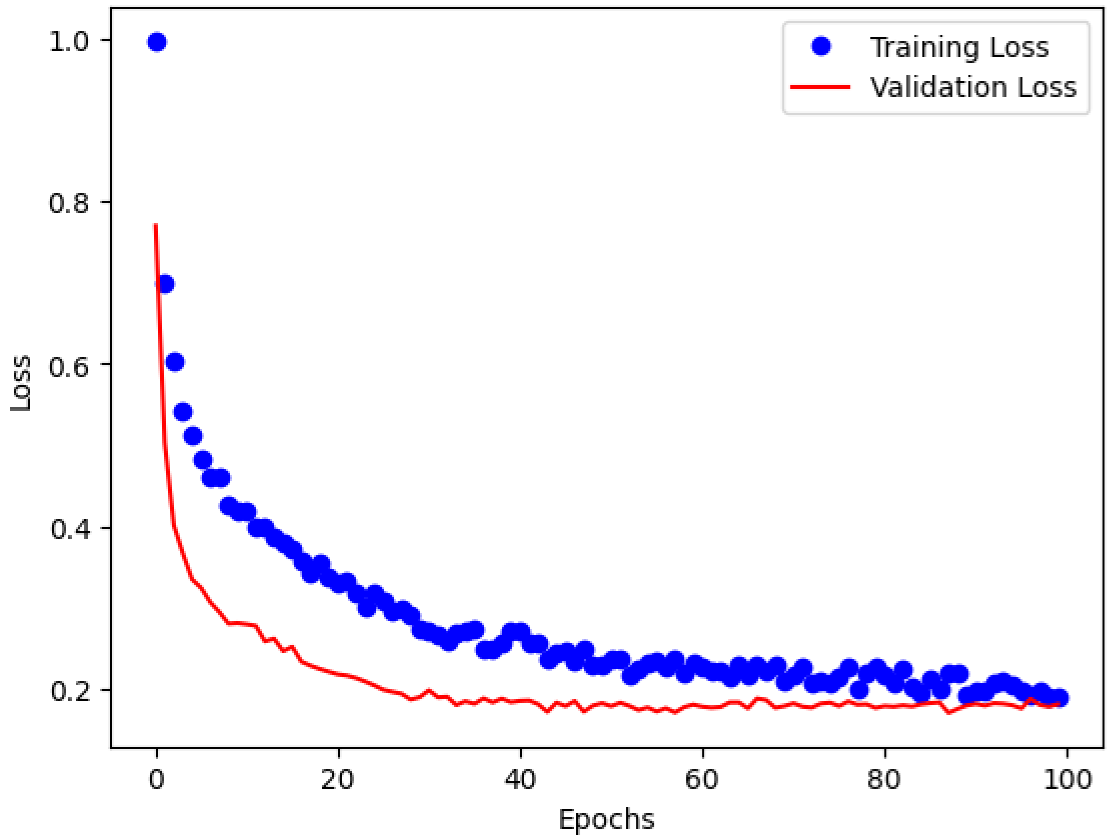
4.2. Train the new model and tune the hyper-parameters

The new model has multiple output. Thus we specify **multiple** loss functions and their weights.

In [20]:
```
from tensorflow.keras import optimizers

sae.compile(loss=['mean_squared_error', 'categorical_crossentropy'],
            loss_weights=[1, 0.5], # to be tuned
            optimizer=optimizers.RMSprop(learning_rate=1E-3))

history = sae.fit(x_tr, [x_tr, y_tr],
                  batch_size=32,
                  epochs=100,
                  validation_data=(x_val, [x_val, y_val]))
```

```
Epoch 1/100
313/313 [==============================] - 13s 19ms/step - loss: 0.9972 -
model_4_loss: 0.1120 - model_5_loss: 1.7704 - val_loss: 0.7697 - val_model
_4_loss: 0.0713 - val_model_5_loss: 1.3968
Epoch 2/100
313/313 [==============================] - 5s 16ms/step - loss: 0.6995 - m
odel_4_loss: 0.0725 - model_5_loss: 1.2541 - val_loss: 0.5027 - val_model_
4_loss: 0.0685 - val_model_5_loss: 0.8683
Epoch 3/100
313/313 [==============================] - 6s 19ms/step - loss: 0.6032 - m
odel_4_loss: 0.0703 - model_5_loss: 1.0659 - val_loss: 0.4011 - val_model_
4_loss: 0.0671 - val_model_5_loss: 0.6679
Epoch 4/100
313/313 [==============================] - 5s 16ms/step - loss: 0.5420 - m
odel_4_loss: 0.0691 - model_5_loss: 0.9458 - val_loss: 0.3661 - val_model_
4_loss: 0.0658 - val_model_5_loss: 0.6007
Epoch 5/100
313/313 [==============================] - 6s 19ms/step - loss: 0.5126 - m
odel_4_loss: 0.0683 - model_5_loss: 0.8885 - val_loss: 0.3346 - val_model_
```

In [21]:
```
import matplotlib.pyplot as plt
%matplotlib inline

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(len(loss))

plt.plot(epochs, loss, 'bo', label='Training Loss')
plt.plot(epochs, val_loss, 'r', label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



Question (10 points)

Do you think overfitting is happening? If yes, what can you do? Please make necessary changes to the supervised autoencoder network structure.

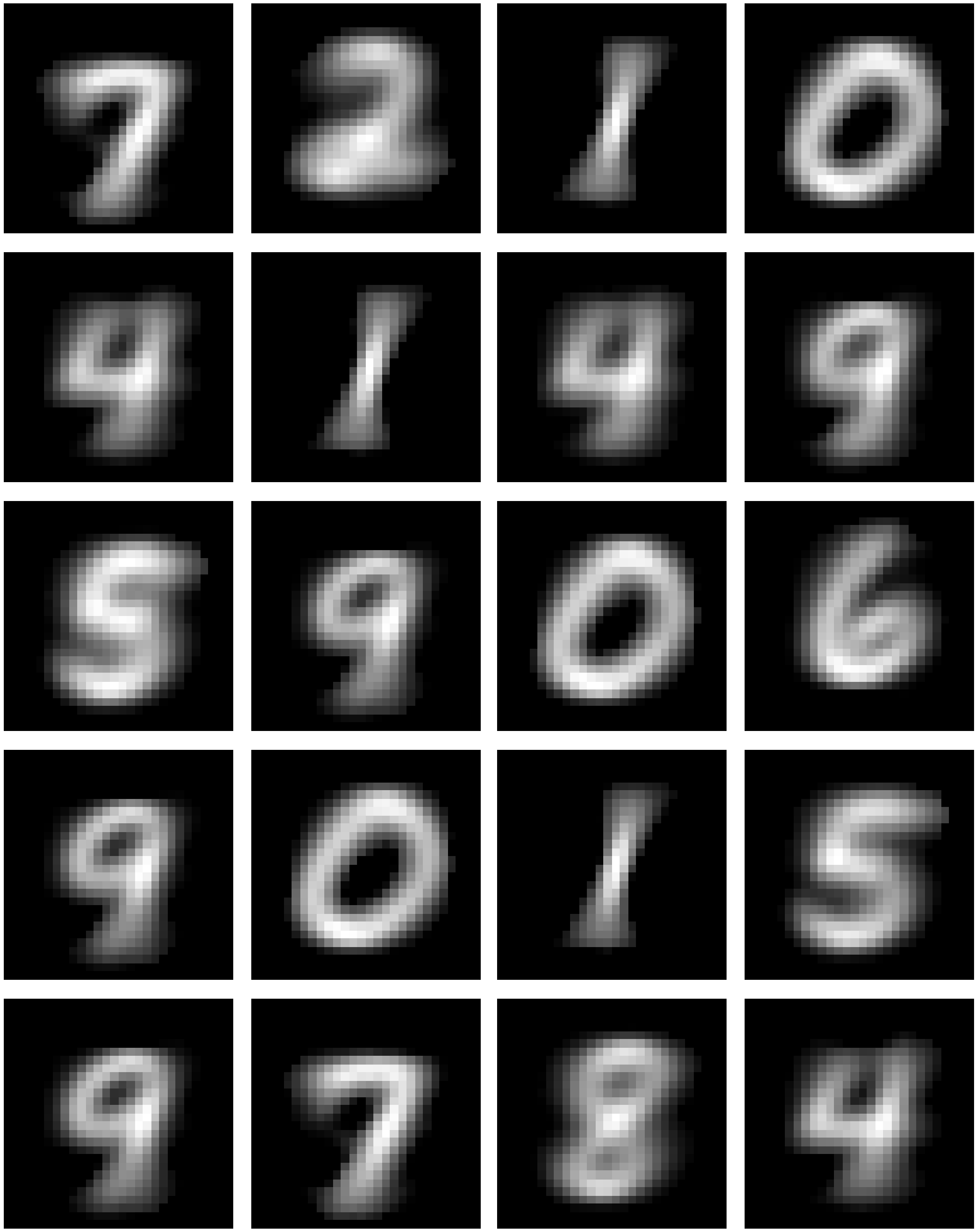
You can use the new model without overfitting for the following sections.

In [21]:
```
```

4.3. Visualize the reconstructed test images

In [22]:
```
sae_output = sae.predict(x_test)[0].reshape((10000, 28, 28))

ROW = 5
COLUMN = 4

x = sae_output
fname = 'reconstruct_sae.pdf'

fig, axes = plt.subplots(nrows=ROW, ncols=COLUMN, figsize=(8, 10))
for ax, i in zip(axes.flat, np.arange(ROW*COLUMN)):
    image = x[i].reshape(28, 28)
    ax.imshow(image, cmap='gray')
    ax.axis('off')

plt.tight_layout()
plt.savefig(fname)
plt.show()
```

```
313/313 [==============================] - 1s 3ms/step
```



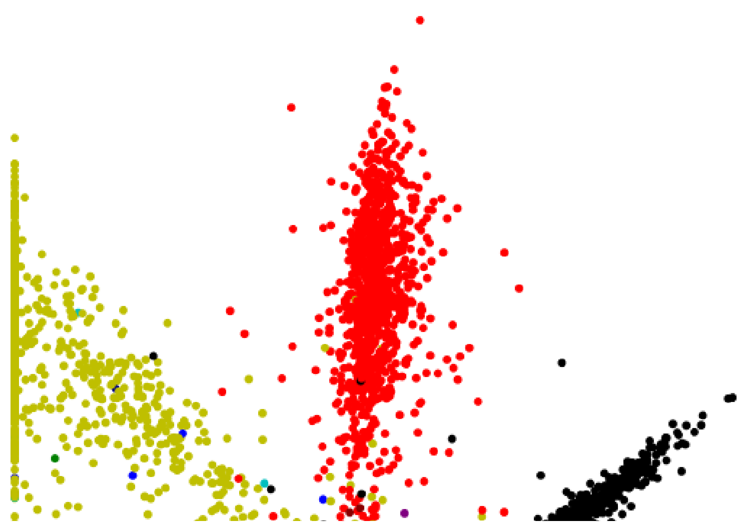
4.4. Visualize the low-dimensional features

In [23]:
```
# build the encoder model
sae_encoder = models.Model(input_img, bottleneck)
sae_encoder.summary()
```

```
Model: "model_7"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_img (InputLayer)      [(None, 784)]             0         
                                                                 
 encode1 (Dense)             (None, 128)               100480    
                                                                 
 batch_normalization (BatchN  (None, 128)              512       
 ormalization)                                                   
                                                                 
 activation (Activation)     (None, 128)               0         
                                                                 
 dropout (Dropout)           (None, 128)               0         
                                                                 
 encode2 (Dense)             (None, 32)                4128      
                                                                 
 batch_normalization_1 (Batc  (None, 32)               128       
 hNormalization)                                                 
                                                                 
 activation_1 (Activation)   (None, 32)                0         
                                                                 
 dropout_1 (Dropout)         (None, 32)                0         
                                                                 
 encode3 (Dense)             (None, 8)                 264       
                                                                 
 batch_normalization_2 (Batc  (None, 8)                32        
 hNormalization)                                                 
                                                                 
 activation_2 (Activation)   (None, 8)                 0         
                                                                 
 bottleneck (Dense)          (None, 2)                 18        
                                                                 
 batch_normalization_3 (Batc  (None, 2)                8         
 hNormalization)                                                 
                                                                 
 activation_3 (Activation)   (None, 2)                 0         
                                                                 
=================================================================
Total params: 105,570
Trainable params: 105,230
Non-trainable params: 340
_________________________________________________________________
```

In [24]:
```
# extract test features
encoded_test = sae_encoder.predict(x_test)
print('Shape of encoded_test: ' + str(encoded_test.shape))

colors = np.array(['r', 'g', 'b', 'm', 'c', 'k', 'y', 'purple', 'darkred', 'na
colors_test = colors[y_test]


import matplotlib.pyplot as plt
%matplotlib inline

fig = plt.figure(figsize=(8, 8))
plt.scatter(encoded_test[:, 0], encoded_test[:, 1], s=10, c=colors_test, edgec
plt.axis('off')
plt.tight_layout()
fname = 'sae_code.pdf'
plt.savefig(fname)
```

```
313/313 [==============================] - 1s 2ms/step
Shape of encoded_test: (10000, 2)
```



4.5. Are the learned low-dim features discriminative? (10 points)

To find the answer, lets train a classifier on the training set (the extracted 2-dim features) and evaluation on the validation and test set.

In [25]:
```
# extract 2D features from the training, validation, and test samples
f_tr = sae_encoder.predict(x_tr)
f_val = sae_encoder.predict(x_val)
f_te = sae_encoder.predict(x_test)
```

```
313/313 [==============================] - 1s 3ms/step
313/313 [==============================] - 1s 3ms/step
313/313 [==============================] - 1s 2ms/step
```

In [26]:
```
# build a classifier which takes the 2D features as input
from keras.layers import *
from keras import models

classifier = models.Model(input_feat, output)

classifier.summary()
```

```
Model: "model_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_3 (InputLayer)        [(None, 2)]               0         
                                                                 
 dense_3 (Dense)             (None, 128)               384       
                                                                 
 batch_normalization_8 (Batc  (None, 128)              512       
 hNormalization)                                                 
                                                                 
 activation_8 (Activation)   (None, 128)               0         
                                                                 
 dropout_4 (Dropout)         (None, 128)               0         
                                                                 
 dense_4 (Dense)             (None, 128)               16512     
                                                                 
 batch_normalization_9 (Batc  (None, 128)              512       
 hNormalization)                                                 
                                                                 
 activation_9 (Activation)   (None, 128)               0         
                                                                 
 dropout_5 (Dropout)         (None, 128)               0         
                                                                 
 dense_5 (Dense)             (None, 10)                1290      
                                                                 
=================================================================
Total params: 19,210
Trainable params: 18,698
Non-trainable params: 512
_________________________________________________________________
```

In [27]:
```
classifier.compile(loss='categorical_crossentropy',
                  optimizer=optimizers.RMSprop(learning_rate=1E-4),
                  metrics=['acc'])

history = classifier.fit(f_tr, y_tr,
                        batch_size=32,
                        epochs=30,
                        validation_data=(f_val, y_val))
```

```
Epoch 1/30
313/313 [==============================] - 5s 9ms/step - loss: 0.2512 - acc:
0.9270 - val_loss: 0.2897 - val_acc: 0.9466
Epoch 2/30
313/313 [==============================] - 2s 7ms/step - loss: 0.2471 - acc:
0.9335 - val_loss: 0.3003 - val_acc: 0.9451
Epoch 3/30
313/313 [==============================] - 2s 7ms/step - loss: 0.2372 - acc:
0.9368 - val_loss: 0.3033 - val_acc: 0.9452
Epoch 4/30
313/313 [==============================] - 3s 9ms/step - loss: 0.2568 - acc:
0.9348 - val_loss: 0.3065 - val_acc: 0.9439
Epoch 5/30
313/313 [==============================] - 3s 9ms/step - loss: 0.1995 - acc:
0.9448 - val_loss: 0.3023 - val_acc: 0.9433
Epoch 6/30
313/313 [==============================] - 3s 9ms/step - loss: 0.1698 - acc:
0.9502 - val_loss: 0.3079 - val_acc: 0.9418
Epoch 7/30
313/313 [==============================] - 2s 7ms/step - loss: 0.1979 - acc:
0.9443 - val_loss: 0.3070 - val_acc: 0.9412
Epoch 8/30
313/313 [==============================] - 3s 9ms/step - loss: 0.1759 - acc:
0.9495 - val_loss: 0.3050 - val_acc: 0.9402
Epoch 9/30
313/313 [==============================] - 3s 9ms/step - loss: 0.1980 - acc:
0.9440 - val_loss: 0.3115 - val_acc: 0.9414
Epoch 10/30
313/313 [==============================] - 2s 7ms/step - loss: 0.1901 - acc:
0.9462 - val_loss: 0.3106 - val_acc: 0.9404
Epoch 11/30
313/313 [==============================] - 2s 7ms/step - loss: 0.2072 - acc:
0.9451 - val_loss: 0.3168 - val_acc: 0.9393
Epoch 12/30
313/313 [==============================] - 2s 7ms/step - loss: 0.2203 - acc:
0.9385 - val_loss: 0.3072 - val_acc: 0.9404
Epoch 13/30
313/313 [==============================] - 2s 7ms/step - loss: 0.1819 - acc:
0.9504 - val_loss: 0.3057 - val_acc: 0.9397
Epoch 14/30
313/313 [==============================] - 3s 9ms/step - loss: 0.1746 - acc:
0.9519 - val_loss: 0.3123 - val_acc: 0.9391
Epoch 15/30
313/313 [==============================] - 2s 7ms/step - loss: 0.2019 - acc:
0.9413 - val_loss: 0.3090 - val_acc: 0.9393
Epoch 16/30
313/313 [==============================] - 2s 7ms/step - loss: 0.1773 - acc:
0.9493 - val_loss: 0.3124 - val_acc: 0.9395
Epoch 17/30
313/313 [==============================] - 2s 7ms/step - loss: 0.1775 - acc:
0.9521 - val_loss: 0.3082 - val_acc: 0.9399
Epoch 18/30
313/313 [==============================] - 2s 7ms/step - loss: 0.1673 - acc:
0.9537 - val_loss: 0.3164 - val_acc: 0.9397
Epoch 19/30
313/313 [==============================] - 3s 9ms/step - loss: 0.1965 - acc:
0.9463 - val_loss: 0.3185 - val_acc: 0.9397
```

```
Epoch 20/30
313/313 [==============================] - 3s 8ms/step - loss: 0.1613 - acc:
0.9543 - val_loss: 0.3165 - val_acc: 0.9382
Epoch 21/30
313/313 [==============================] - 2s 7ms/step - loss: 0.1799 - acc:
0.9498 - val_loss: 0.3126 - val_acc: 0.9399
Epoch 22/30
313/313 [==============================] - 2s 7ms/step - loss: 0.1683 - acc:
0.9508 - val_loss: 0.3151 - val_acc: 0.9387
Epoch 23/30
313/313 [==============================] - 3s 9ms/step - loss: 0.1727 - acc:
0.9506 - val_loss: 0.3203 - val_acc: 0.9379
Epoch 24/30
313/313 [==============================] - 2s 7ms/step - loss: 0.1854 - acc:
0.9492 - val_loss: 0.3160 - val_acc: 0.9380
Epoch 25/30
313/313 [==============================] - 3s 9ms/step - loss: 0.1709 - acc:
0.9531 - val_loss: 0.3166 - val_acc: 0.9395
Epoch 26/30
313/313 [==============================] - 3s 9ms/step - loss: 0.1599 - acc:
0.9550 - val_loss: 0.3129 - val_acc: 0.9399
Epoch 27/30
313/313 [==============================] - 2s 7ms/step - loss: 0.1879 - acc:
0.9466 - val_loss: 0.3175 - val_acc: 0.9387
Epoch 28/30
313/313 [==============================] - 2s 7ms/step - loss: 0.1944 - acc:
0.9465 - val_loss: 0.3166 - val_acc: 0.9386
Epoch 29/30
313/313 [==============================] - 2s 7ms/step - loss: 0.1548 - acc:
0.9555 - val_loss: 0.3232 - val_acc: 0.9372
Epoch 30/30
313/313 [==============================] - 3s 9ms/step - loss: 0.1803 - acc:
0.9468 - val_loss: 0.3248 - val_acc: 0.9390
```

Remark: (10 points)

The validation accuracy must be above 90%. It means the low-dim features learned by the supervised autoencoder are very effective.

In [28]:
```
# evaluate your model on the never-seen-before test data
# write your code here:
lossAccuracy = classifier.evaluate(f_te, y_test_vec)

print('Accuracy:', lossAccuracy[1] * 100)
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.3387 - acc:
0.9404
Accuracy: 94.04000043869019
```