In [7]:

```
# The code below is borrowed from Professor Zoran's Lecture 3 notebook
import numpy as np
import gensim
# Get the interactive Tools for Matplotlib
%matplotlib notebook
import matplotlib.pyplot as plt
plt.style.use('ggplot')
from sklearn.decomposition import PCA
from gensim.test.utils import datapath, get_tmpfile
from gensim.models import KeyedVectors
from gensim.scripts.glove2word2vec import glove2word2vec
```

Problem 1.

Examine examples of analogic reasoning we demonstrated in Lecture 03. Jupyter notebook with those examples is uploaded in the folder for Lecture 03 of the class site. One such example is “what is to Russia, what Paris is to France?”. Those four words (Russia, France, Paris and Moscow) should present a polygon with four edges, perhaps a romb or rectangle. Create three more similar analogies and present them in the same PCA plane. We are just curious whether the geometric shapes of those examples are identical or very similar one to another. Please select analogies of very similar nature: countries vs. capitals, people vs food, etc. Do this using the 100-dimensional Glove vectors transformed into Word2Vec format. Use Gensim API. If you are familiar with Spacy or some other NLP API, please be free to use it.

In [7]:

```
glove_file = datapath('/Users/ly/Desktop/Harvard Summer/glove/glove.6B.100d.txt')
word2vec_glove_file = get_tmpfile("glove.6B.100d.word2vec.txt")
glove2word2vec(glove_file, word2vec_glove_file)
model = KeyedVectors.load_word2vec_format(word2vec_glove_file)
```

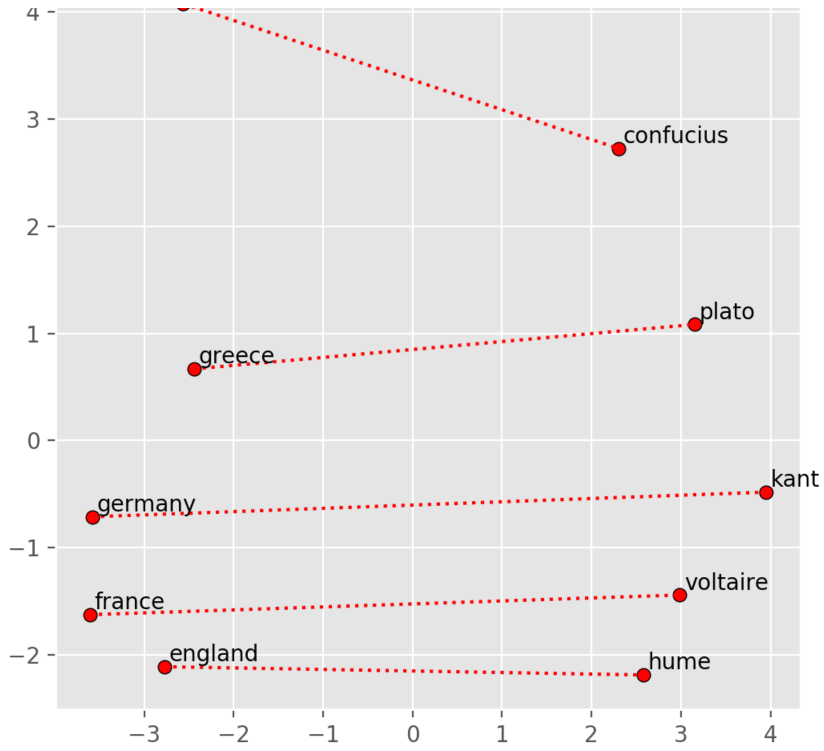
In [30]:

```
def display_pca_scatterplot(model, words=None, sample=0):
 if words == None:
  if sample > 0:
   words = np.random.choice(list(model.vocab.keys()), sample)
  else:
   words = [ word for word in model.vocab ]
 word_vectors = np.array([model[w] for w in words])
 twodim = PCA().fit_transform(word_vectors)[:,:2]
 plt.figure(figsize=(6,6))
 plt.scatter(twodim[:,0], twodim[:,1], edgecolors='k', c='r')
 for word, (x,y) in zip(words, twodim):
  plt.text(x+0.05, y+0.05, word)
 # Add connected lines to showoff the parallel relationship
 for i in range(0, len(twodim), 2):
  plt.plot(twodim[i:i+2,0],twodim[i:i+2,1],'r:')
```

Analogy 1: Country - Philosopher

In [31]:

```
display_pca_scatterplot(model,['france','voltaire',
                               'greece','plato',
                               'england','hume',
                               'china','confucius',
                               'germany','kant'])
```



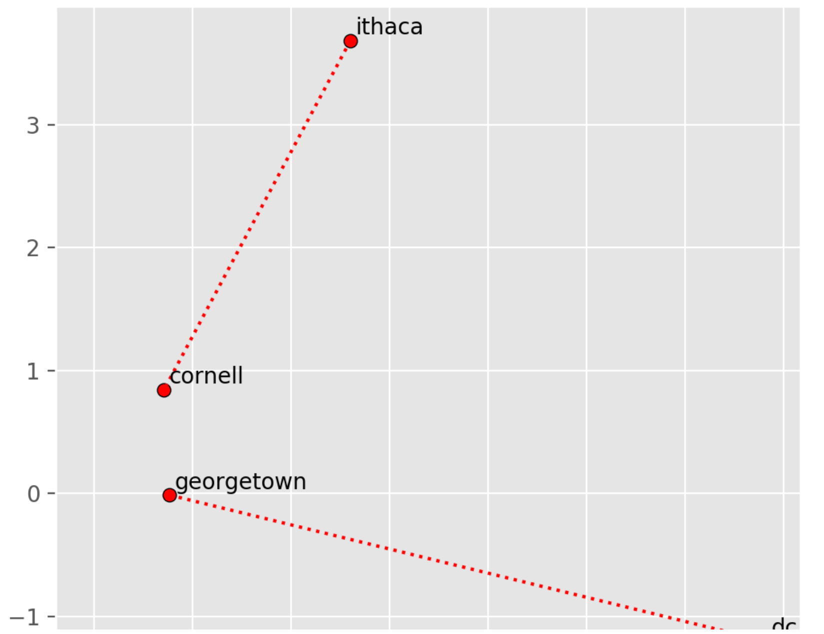
Comment:

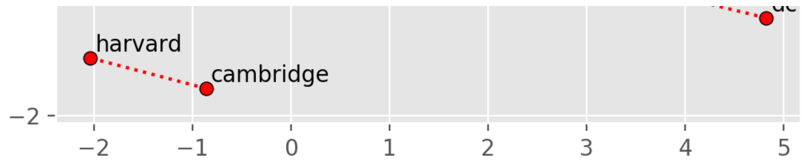
The analogy is apparent, all the countries lie on the left side and all the philosophers lie on the right side. Moreover, the connected the line between the country and the corresponding philosopher appear to be approximitaly parallel. The parallel relationship is stronger for European countries but weaker for between China and European countries.

Analogy 2: City - University

In [45]:

```
display_pca_scatterplot(model,['cambridge','harvard',
                               'dc','georgetown',
                               'ithaca','cornell'])
```





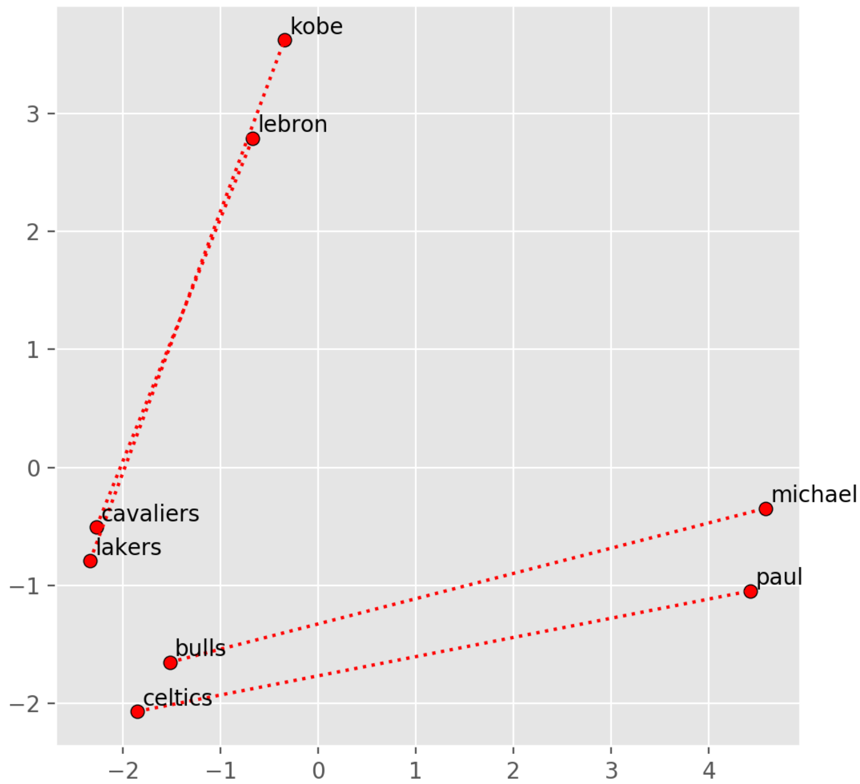
Comment:

The relationship holds for Georgetown University in DC and Harvard in Cambridge, but seems odd for Cornell in Ithaca. I also tested some city names with two words, such as "New York" and "Columbia", which turned out to be worse and was ignored.

Analogy 3: NBA Team - Leading Player

In [68]:

```
display_pca_scatterplot(model,['lakers','kobe',
                               'celtics','paul',
                               'cavaliers','lebron',
                               'bulls','michael'
                               ])
```



Comment:

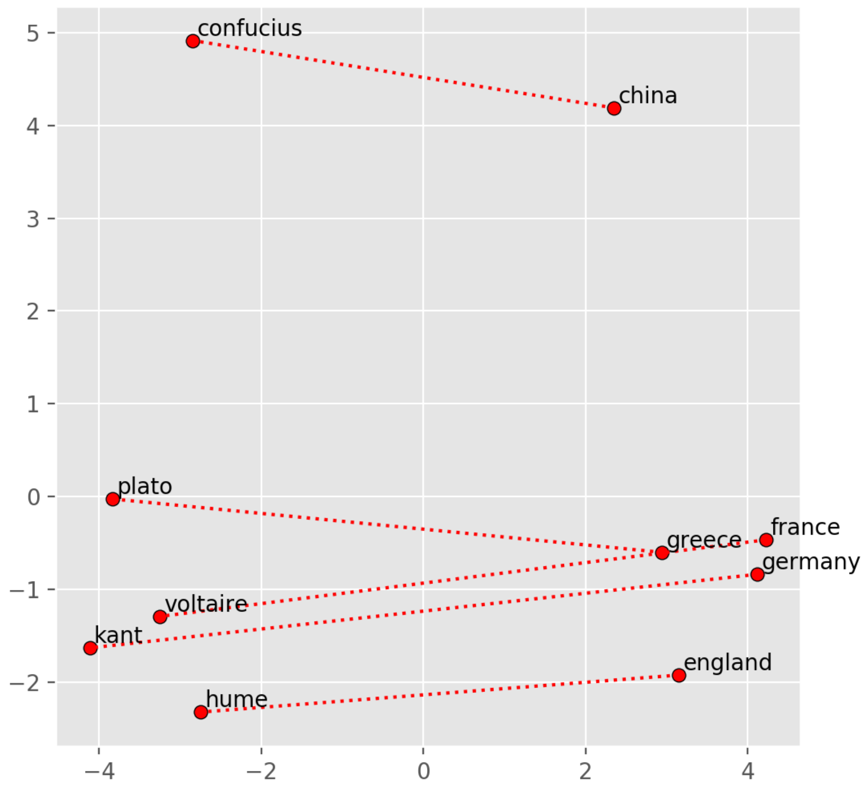
The relationship approximately holds, the team names lie on the left while the player names on the right. Kobe Bryant is to Lakers as Lebron James to Cavaliers, which is so true! For Celtics, it's a bit off but Paul is not disambiguated. It could be Paul Pierce or Paul George. The same reason might explain Chicago Bulls and Michael Jordan. Besides, there must be a lot of texts/reports comparing Lebron James and Kobe Bryant, thus the close location of the two pair of points.

Problem 2.

Repeat the above experiment with 300-dimensional Glove vectors transformed into Word2Vec format. We are curious whether the shape of above geometric shapes are preserved or modified in the higher dimensional space. Use PCA to make the projections.

In [71]:

```
display_pca_scatterplot(model_300,['france','voltaire',
                                    'greece','plato',
                                    'england','hume',
                                    'china','confucius',
                                    'germany','kant'])
```



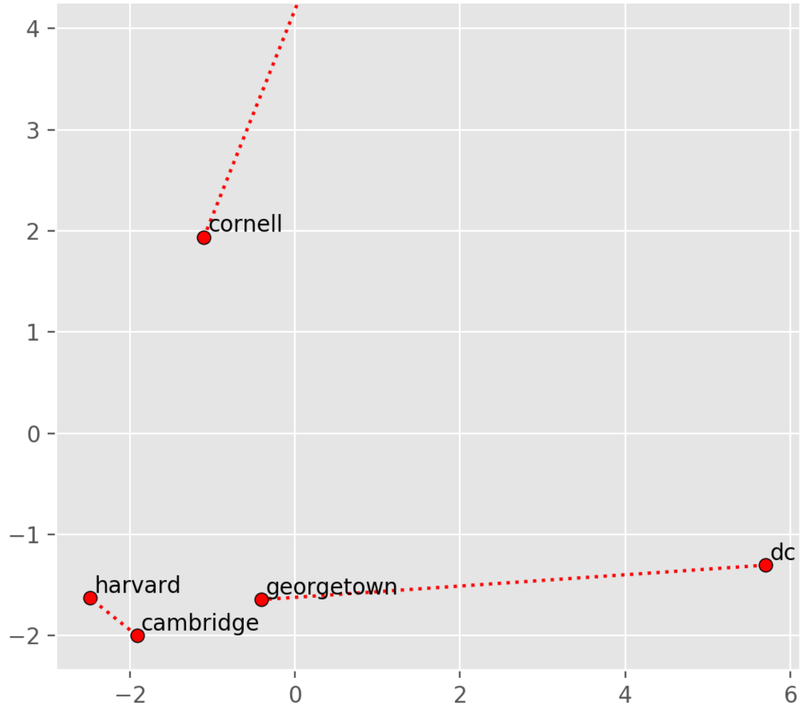
Comment:

The general shape or relative location of the lines holds, but for Greece - Plato, it becomes less parallel with the other European country - Philosophers, but more parallel with Chinese - Confucius, which actually makes sense. Becaues Plato and Confucius are philosophers a long time ago while those for the other European countries are more recent.

Analogy 2: City - University

In [72]:

```
display_pca_scatterplot(model_300,['cambridge','harvard',
                                    'dc','georgetown',
                                    'ithaca','cornell'])
```



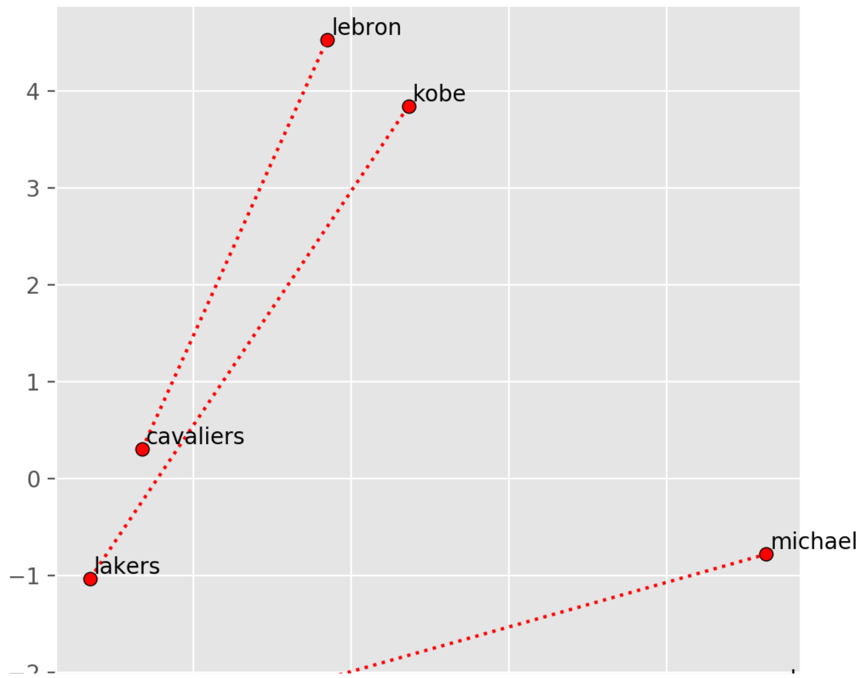
Comment:

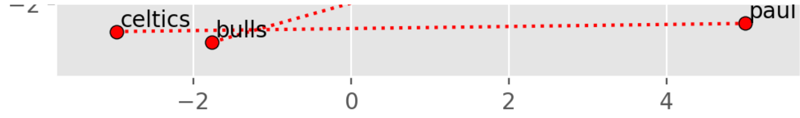
The relationships become worse somehow.

Analogy 3: NBA Team - Leading Player

In [74]:

```
display_pca_scatterplot(model_300,['lakers','kobe',
                                    'celtics','paul',
                                    'cavaliers','lebron',
                                    'bulls','michael'
                                    ])
```





Comment:

The relationships become worse somehow. Lakers - Koby and Cavaliers - Lebron becomes more separate while Celtics - Paul becomes almost flat. Paul could potentially refer to more person in the high dimensional place.

Problem 3.

Repeat the above experiment from problem 1 using Sci-kit Learn T-SNE projections, rather than PCA.

In [4]:

```
from sklearn.manifold import TSNE
```

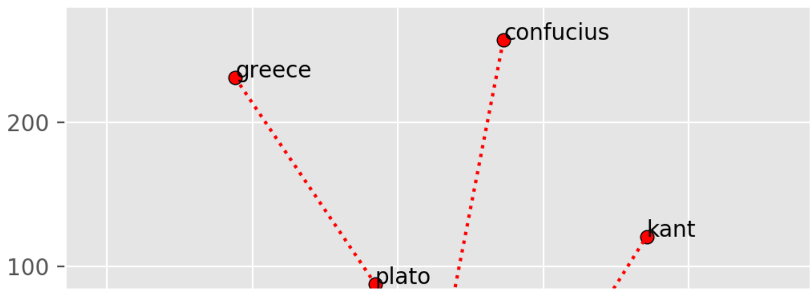
In [102]:

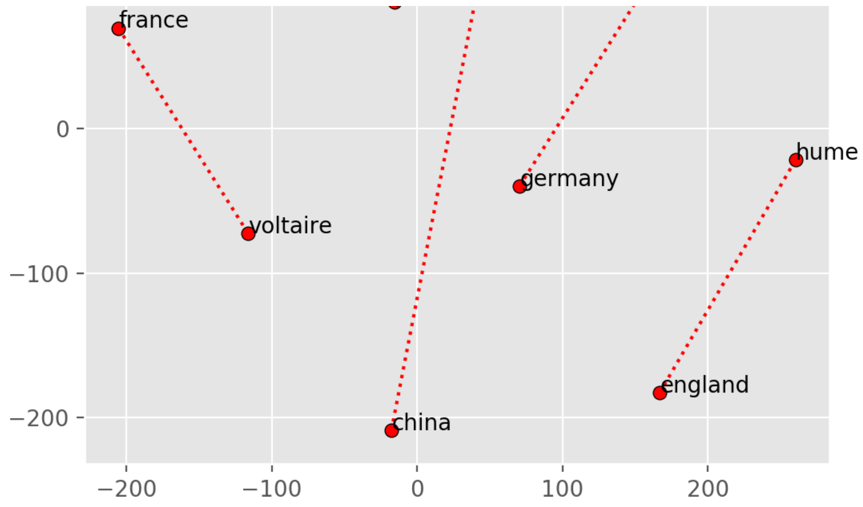
```
# Adapt pca function to tsne
# Add a parameter random_state for easy tuning
def display_tsne_scatterplot(model,random_state, words=None, sample=0):
    if words == None:
        if sample > 0:
            words = np.random.choice(list(model.vocab.keys()), sample)
        else:
            words = [ word for word in model.vocab ]
    word_vectors = np.array([model[w] for w in words])
    tsne = TSNE(perplexity=30, n_components=2, init='random', n_iter=5000,
                method='exact',random_state = random_state)
    twodim = tsne.fit_transform(word_vectors[:, :2])
    #twodim = PCA().fit_transform(word_vectors)[:,:2]
    plt.figure(figsize=(6,6))
    plt.scatter(twodim[:,0], twodim[:,1], edgecolors='k', c='r')
    for word, (x,y) in zip(words, twodim):
        plt.text(x+0.05, y+0.05, word)
    # Add connected lines to showoff the parallel relationship
    for i in range(0, len(twodim), 2):
        plt.plot(twodim[i:i+2,0],twodim[i:i+2,1],'r:')
```

Analogy 1: Country - Philosopher

In [101]:

```
display_tsne_scatterplot(model,10,['france','voltaire',
                                   'greece','plato',
                                   'england','hume',
                                   'china','confucius',
                                   'germany','kant'])
```





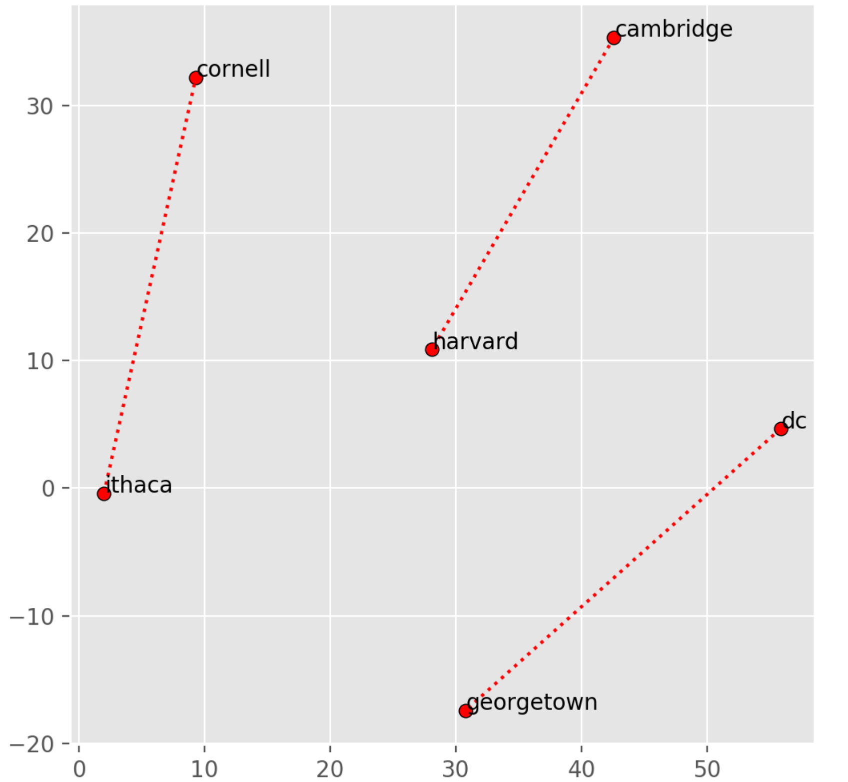
Comment:

The relationship now looks weird, which could be due to random initialization. The chart looks somewhat symmetric with random state = 10.

Analogy 2: City - University

In [103]:

```
display_tsne_scatterplot(model,1, ['cambridge','harvard',
                                   'dc','georgetown',
                                   'ithaca','cornell'])
```



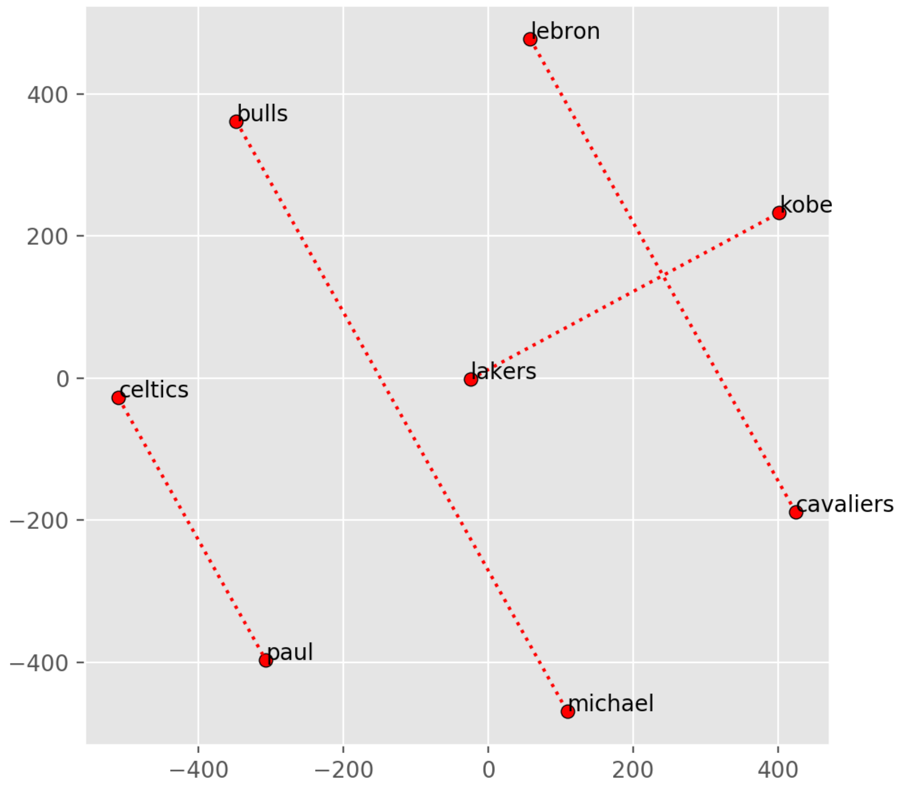
Comment:

The relationship almost holds. But Cornell-Ithaca should have reversed. Note that random state actually affects the result.

Analogy 3: NBA Team - Leading Player

In [107]:

```
display_tsne_scatterplot(model,102, ['lakers','kobe',
                                     'celtics','paul',
                                     'cavaliers','lebron',
                                     'bulls','michael'
                                     ])
```



Comment:

The relationship appears different from before. Celtics - Paul and Bulls - Michael are parallel. Lebron - Cavaliers seems reversed while Kobe - Lakers crosses with Lebron - Cavaliers. Still, random state actually affects the result.

Problem 4.

Try to use the simplest architecture we discussed in Lecture 4 with two Dense layers to create a de-noising autoencoder. Use the latent space of dimension 64. Report on your findings.

1) Create Noisy Data

In [298]:

```
from keras.datasets import mnist
import numpy as np
```

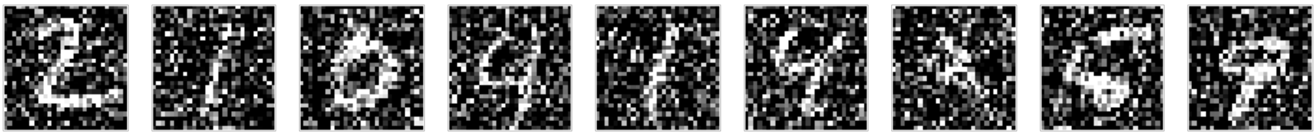
```
(x_train, _), (x_test, _) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

noise_factor = 0.5
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.)
print("train.shape: ", x_train_noisy.shape)
print("test.shape: ", x_test_noisy.shape)
```

```
train.shape:  (60000, 784)
test.shape:  (10000, 784)
```

2) Visualize Noisy Data

In [121]:

```
# use Matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

n = 10
plt.figure(figsize=(20, 2))
for i in range(1,n,1):
    ax = plt.subplot(1, n, i)
    plt.imshow(x_test_noisy[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



3) Create a 2-layer autoencoder

Autoencoder

In [122]:

```
import keras
from keras.layers import Input, Dense
from keras.models import Model

# set encoding dimension as 64 64-D Latent Space
encoding_dim = 64   # assuming the input is 784 floats

# this is our input placeholder
input_img = Input(shape=(784,))
# add first layer
first_layer = Dense(encoding_dim, activation='relu')(input_img)
# "encoded" is the encoded representation of the input
encoded = Dense(encoding_dim, activation='relu')(first_layer)
# "decoded" is the lossy reconstruction of the input
decoded = Dense(784, activation='sigmoid')(encoded)

# this model maps an input to its reconstruction
autoencoder = Model(input_img, decoded)
```

Encoder

In [123]:

```
# this model maps an input to its encoded representation
encoder = Model(input_img, encoded)
```

Decoder

In [124]:

```
# create a placeholder for an encoded (32-dimensional) input
encoded_input = Input(shape=(encoding_dim,))
# retrieve the last layer of the autoencoder model
decoder_layer = autoencoder.layers[-1]
# create the decoder model
decoder = Model(encoded_input, decoder_layer(encoded_input))
```

Compile autoencoder

In [125]:

```
autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')
```

In [128]:

```
history = autoencoder.fit(x_train_noisy, x_train,
                            epochs=50,
                            batch_size=128,
                            shuffle=True,
                            validation_data=(x_test_noisy, x_test))
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1215 - val_loss: 0.1206
Epoch 2/50
60000/60000 [==============================] - 2s 40us/step - loss: 0.1212 - val_loss: 0.1212
Epoch 3/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1211 - val_loss: 0.1212
Epoch 4/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1208 - val_loss: 0.1212
Epoch 5/50
60000/60000 [==============================] - 2s 40us/step - loss: 0.1207 - val_loss: 0.1198
Epoch 6/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1204 - val_loss: 0.1204
Epoch 7/50
60000/60000 [==============================] - 2s 38us/step - loss: 0.1203 - val_loss: 0.1204
Epoch 8/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1200 - val_loss: 0.1196
Epoch 9/50
60000/60000 [==============================] - 2s 40us/step - loss: 0.1199 - val_loss: 0.1217
Epoch 10/50
60000/60000 [==============================] - 2s 38us/step - loss: 0.1197 - val_loss: 0.1194
Epoch 11/50
60000/60000 [==============================] - 2s 38us/step - loss: 0.1195 - val_loss: 0.1197
Epoch 12/50
60000/60000 [==============================] - 2s 40us/step - loss: 0.1194 - val_loss: 0.1187
Epoch 13/50
60000/60000 [==============================] - 2s 38us/step - loss: 0.1191 - val_loss: 0.1189
Epoch 14/50
60000/60000 [==============================] - 2s 38us/step - loss: 0.1190 - val_loss: 0.1187
Epoch 15/50
60000/60000 [==============================] - 2s 38us/step - loss: 0.1189 - val_loss: 0.1185
Epoch 16/50
60000/60000 [==============================] - 2s 38us/step - loss: 0.1187 - val_loss: 0.1187
Epoch 17/50
60000/60000 [==============================] - 2s 38us/step - loss: 0.1186 - val_loss: 0.1181
Epoch 18/50
60000/60000 [==============================] - 2s 41us/step - loss: 0.1183 - val_loss: 0.1180
Epoch 19/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1181 - val_loss: 0.1182
```

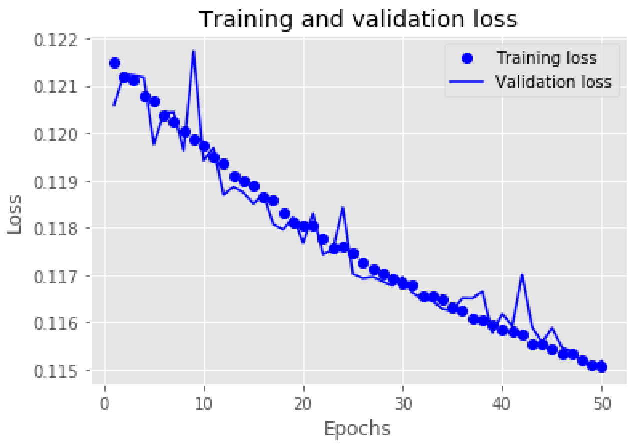
```
Epoch 20/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1181 - val_loss: 0.1177
Epoch 21/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1181 - val_loss: 0.1183
Epoch 22/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1178 - val_loss: 0.1174
Epoch 23/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1176 - val_loss: 0.1175
Epoch 24/50
60000/60000 [==============================] - 2s 40us/step - loss: 0.1176 - val_loss: 0.1184
Epoch 25/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1175 - val_loss: 0.1170
Epoch 26/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1173 - val_loss: 0.1169
Epoch 27/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1171 - val_loss: 0.1170
Epoch 28/50
60000/60000 [==============================] - 2s 40us/step - loss: 0.1170 - val_loss: 0.1169
Epoch 29/50
60000/60000 [==============================] - 2s 38us/step - loss: 0.1169 - val_loss: 0.1168
Epoch 30/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1168 - val_loss: 0.1170
Epoch 31/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1168 - val_loss: 0.1166
Epoch 32/50
60000/60000 [==============================] - 2s 38us/step - loss: 0.1166 - val_loss: 0.1165
Epoch 33/50
60000/60000 [==============================] - 2s 38us/step - loss: 0.1165 - val_loss: 0.1165
Epoch 34/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1165 - val_loss: 0.1163
Epoch 35/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1163 - val_loss: 0.1162
Epoch 36/50
60000/60000 [==============================] - 2s 38us/step - loss: 0.1162 - val_loss: 0.1165
Epoch 37/50
60000/60000 [==============================] - 2s 38us/step - loss: 0.1161 - val_loss: 0.1165
Epoch 38/50
60000/60000 [==============================] - 2s 38us/step - loss: 0.1160 - val_loss: 0.1166
Epoch 39/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1159 - val_loss: 0.1158
Epoch 40/50
60000/60000 [==============================] - 2s 38us/step - loss: 0.1159 - val_loss: 0.1162
Epoch 41/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1158 - val_loss: 0.1159
Epoch 42/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1157 - val_loss: 0.1170
Epoch 43/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1155 - val_loss: 0.1159
Epoch 44/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1155 - val_loss: 0.1156
Epoch 45/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1154 - val_loss: 0.1159
Epoch 46/50
60000/60000 [==============================] - 2s 40us/step - loss: 0.1154 - val_loss: 0.1155
Epoch 47/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1153 - val_loss: 0.1154
Epoch 48/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1152 - val_loss: 0.1152
Epoch 49/50
60000/60000 [==============================] - 2s 38us/step - loss: 0.1151 - val_loss: 0.1150
Epoch 50/50
60000/60000 [==============================] - 2s 39us/step - loss: 0.1151 - val_loss: 0.1152
```

Show training and validation loss change by epochs

In [129]:

```python
import matplotlib.pyplot as plt
%matplotlib inline

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

epochs = range(1, len(loss) + 1)
```

```
# "bo" is for "blue dot"
plt.plot(epochs, loss, 'bo', label='Training loss')
# b is for "solid blue line"
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```



Comment:

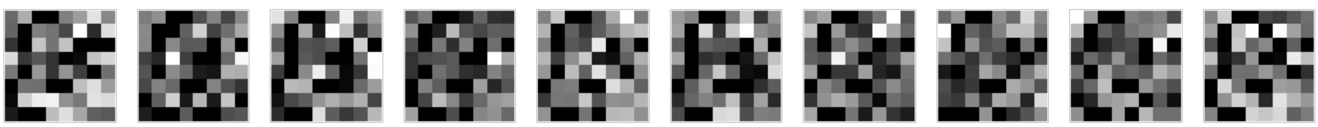
Training loss decreases as we train more epochs while validation loss also shows a downward trend with occasionally fluctuation. Yet, if we notice the y axis scale, within 50 epochs, the training loss and the validation loss only decreases by around 0.007, so the model doesn't really improve much. But the loss is overall low at around 0.12.

Show signal intensity and encodings

In [299]:

```
# note that we take them from the *test* set
encoded_imgs = encoder.predict(x_test_noisy)
print("Signal intensity: ", encoded_imgs.mean())
n = 10  # how many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    # display codings as 4x8 array
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(encoded_imgs[i].reshape(8, 8))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

```
Signal intensity:  5.68109
```



Comment:

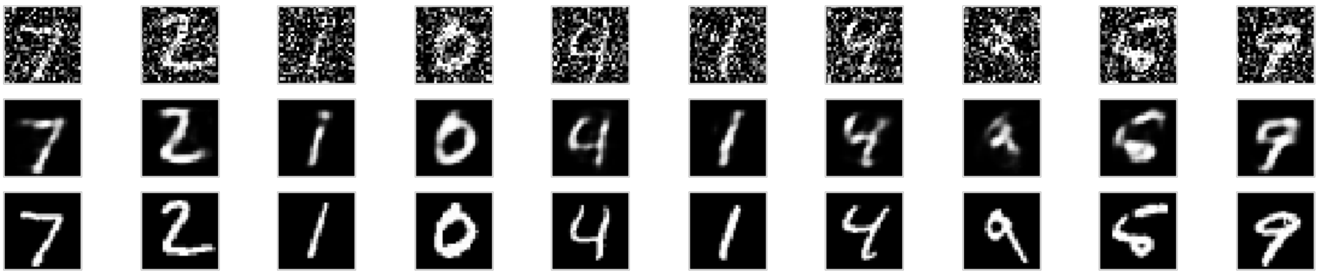
As usual, it's hard to interpret the encodings.

Show noisy and denoised images

Show noisy and denoised images

In [141]:

```
decoded_imgs = autoencoder.predict(x_test_noisy)

n = 10  # how many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    # display original noisy digits
    ax = plt.subplot(3, n, i + 1)
    plt.imshow(x_test_noisy[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction
    ax = plt.subplot(3, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28)) # decoded_img
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display original digits
    ax = plt.subplot(3, n, i + 1 + 2*n)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



Top row: noisy image
Middle row:denoised image
Bottom row: original clean image

Comment:

The denoised image turn out to be pretty good.

Problem 5.

Consider image denoising auto decoder described on slide 54 of the notes for Lecture 4. Make an experiment by reducing the number of filters (channels) contained in all Conv2D layers from 32 to 16. Compare results of autoencoders with 32 and 16 channels visually. Could you come up with a technique to compare the quality of denoising more accurately. In either case present the effect of removing noise from handwritten digits 3, 6, and 8.

Reload, reshape data, add noise and clip

In [232]:

```
from keras.datasets import mnist
import numpy as np

(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))  # adapt this if using `channels_first` im
age data format
```

```
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))  # adapt this if using `channels_first` image data format

noise_factor = 0.1 # Only add a small fraction of noise
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.)
```

In [234]:

```
from keras.layers import Input, Dense, Conv2D, MaxPooling2D, UpSampling2D
from keras.models import Model
from keras import backend as K

def build_autoencoder(dim):
    input_img = Input(shape=(28, 28, 1))  # adapt this if using `channels_first` image data format

    x = Conv2D(dim, (3, 3), activation='relu', padding='same')(input_img)
    x = MaxPooling2D((2, 2), padding='same')(x)
    x = Conv2D(dim, (3, 3), activation='relu', padding='same')(x)
    encoded = MaxPooling2D((2, 2), padding='same')(x)

    # at this point the representation is (7, 7, 32)

    x = Conv2D(dim, (3, 3), activation='relu', padding='same')(encoded)
    x = UpSampling2D((2, 2))(x)
    x = Conv2D(dim, (3, 3), activation='relu', padding='same')(x)
    x = UpSampling2D((2, 2))(x)
    decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

    # Extract encoder
    encoder = Model(input_img, encoded)

    # Build autoencoder
    autoencoder = Model(input_img, decoded)
    autoencoder.compile(optimizer='adadelta', loss='binary_crossentropy')

    return autoencoder,encoder
```

In [289]:

```
def fit_autoencoder(autoencoder, epochs):
    history = autoencoder.fit(x_train_noisy, x_train,
                    epochs=epochs,
                    batch_size=256,
                    shuffle=True,
                    validation_data=(x_test_noisy, x_test))
    return history
```

Build 32D Autoencoder and Extract Encoder

In [250]:

```
# Limit to 15 epochs considering the running time
autoencoder_32, encoder_32 = build_autoencoder(32)
```

Fit 32D Autoencoder

In [251]:

```
history_32 = fit_autoencoder(autoencoder_32, 15)
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/15
60000/60000 [==============================] - 48s 793us/step - loss: 0.1653 - val_loss: 0.0983
Epoch 2/15
60000/60000 [==============================] - 47s 786us/step - loss: 0.0975 - val_loss: 0.0921
```

```
Epoch 3/15
60000/60000 [==============================] - 48s 807us/step - loss: 0.0890 - val_loss: 0.0871
Epoch 4/15
60000/60000 [==============================] - 49s 823us/step - loss: 0.0848 - val_loss: 0.0816
Epoch 5/15
60000/60000 [==============================] - 46s 763us/step - loss: 0.0821 - val_loss: 0.0821
Epoch 6/15
60000/60000 [==============================] - 47s 779us/step - loss: 0.0803 - val_loss: 0.0774
Epoch 7/15
60000/60000 [==============================] - 49s 809us/step - loss: 0.0788 - val_loss: 0.0775
Epoch 8/15
60000/60000 [==============================] - 45s 749us/step - loss: 0.0778 - val_loss: 0.0761
Epoch 9/15
60000/60000 [==============================] - 47s 790us/step - loss: 0.0769 - val_loss: 0.0776
Epoch 10/15
60000/60000 [==============================] - 46s 764us/step - loss: 0.0762 - val_loss: 0.0751
Epoch 11/15
60000/60000 [==============================] - 47s 777us/step - loss: 0.0756 - val_loss: 0.0761
Epoch 12/15
60000/60000 [==============================] - 47s 785us/step - loss: 0.0751 - val_loss: 0.0735
Epoch 13/15
60000/60000 [==============================] - 47s 777us/step - loss: 0.0747 - val_loss: 0.0743
Epoch 14/15
60000/60000 [==============================] - 47s 784us/step - loss: 0.0743 - val_loss: 0.0736
Epoch 15/15
60000/60000 [==============================] - 49s 814us/step - loss: 0.0740 - val_loss: 0.0743
```

Build 16D Autoencoder and Extract Encoder

In [236]:

```
autoencoder_16,encoder_16 = build_autoencoder(16)
```

Fit 16D Autoencoder

In [238]:

```
history_16 = fit_autoencoder(autoencoder_16, 15)
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/15
60000/60000 [==============================] - 30s 504us/step - loss: 0.1280 - val_loss: 0.1133
Epoch 2/15
60000/60000 [==============================] - 30s 499us/step - loss: 0.1033 - val_loss: 0.0958
Epoch 3/15
60000/60000 [==============================] - 29s 485us/step - loss: 0.0943 - val_loss: 0.0908
Epoch 4/15
60000/60000 [==============================] - 29s 487us/step - loss: 0.0899 - val_loss: 0.0861
Epoch 5/15
60000/60000 [==============================] - 29s 485us/step - loss: 0.0870 - val_loss: 0.0845
Epoch 6/15
60000/60000 [==============================] - 29s 483us/step - loss: 0.0852 - val_loss: 0.0841
Epoch 7/15
60000/60000 [==============================] - 28s 474us/step - loss: 0.0838 - val_loss: 0.0825
Epoch 8/15
60000/60000 [==============================] - 29s 490us/step - loss: 0.0826 - val_loss: 0.0803
Epoch 9/15
60000/60000 [==============================] - 29s 477us/step - loss: 0.0817 - val_loss: 0.0810
Epoch 10/15
60000/60000 [==============================] - 32s 528us/step - loss: 0.0810 - val_loss: 0.0791
Epoch 11/15
60000/60000 [==============================] - 29s 483us/step - loss: 0.0800 - val_loss: 0.0786
Epoch 12/15
60000/60000 [==============================] - 30s 502us/step - loss: 0.0795 - val_loss: 0.0779
Epoch 13/15
60000/60000 [==============================] - 29s 484us/step - loss: 0.0789 - val_loss: 0.0785
Epoch 14/15
60000/60000 [==============================] - 29s 489us/step - loss: 0.0783 - val_loss: 0.0774
Epoch 15/15
60000/60000 [==============================] - 30s 493us/step - loss: 0.0779 - val_loss: 0.0772
```

Comment:

16D CNN spends about half time of training in each epoch compared with 32D CNN.
The performace of the two models are similar after 15 epochs.
The training loss at the end of 10 epochs of 16D CNN is 0.0779, while that of 32D CNN is 0.074.
The validation loss at the end of 10 epochs of 16D CNN is 0.0772, while that of 32D CNN is 0.0743.

Visually Compare decoded images of digits 3, 6, 8 for 32D-CNN and 16D-CNN

In [276]:

```
# Subset indexes for digits 3,6,8
y_test_3_inx = list(np.where(y_test == 3)[0])
y_test_6_inx = list(np.where(y_test == 6)[0])
y_test_8_inx = list(np.where(y_test == 8)[0])
```

In [277]:

```
plt_368_inx = y_test_3_inx[:3] + y_test_6_inx[:3] + y_test_8_inx[:3]
print("Plot two 3-digit, two 6-digit, two 8-digit at index:", plt_368_inx)
```

```
Plot two 3-digit, two 6-digit, two 8-digit at index: [18, 30, 32, 11, 21, 22, 61, 84, 110]
```

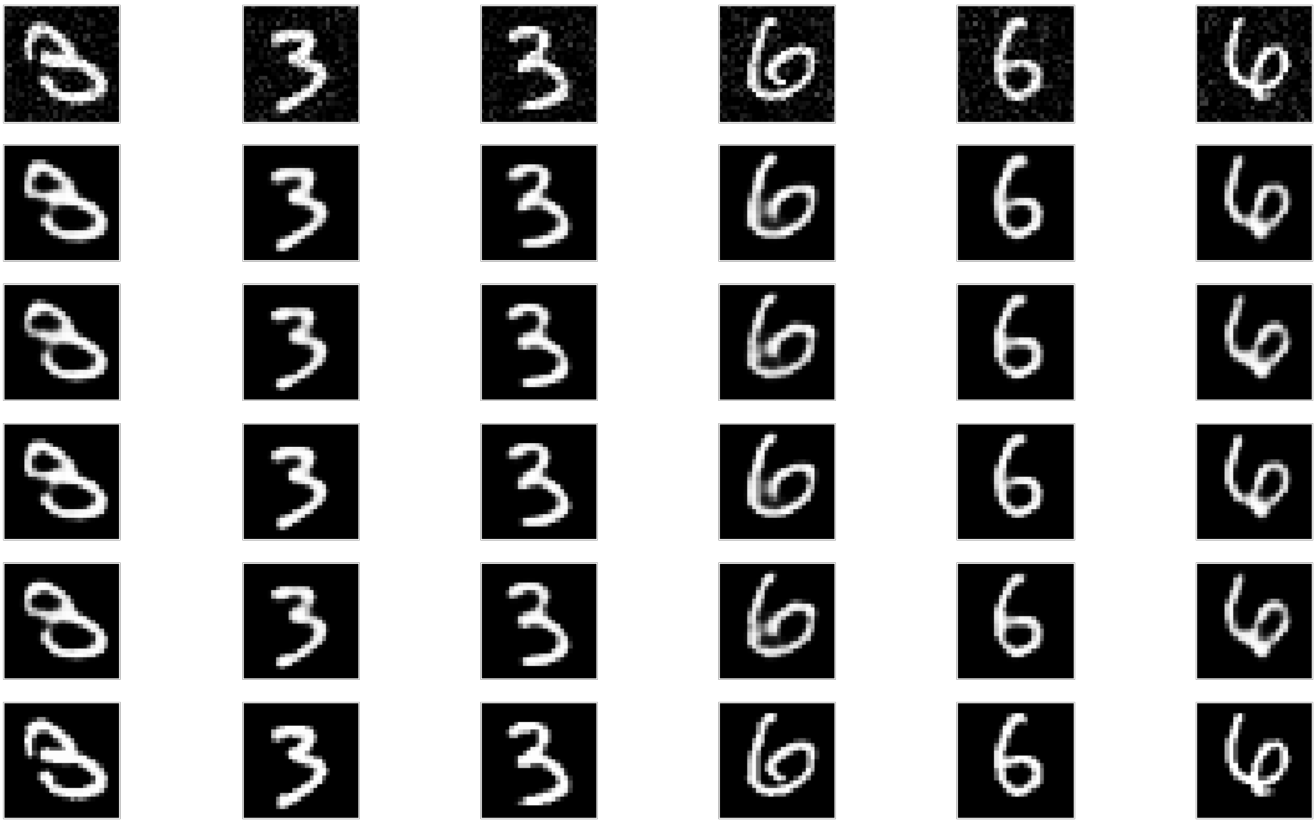
In [291]:

```
# Get decoded images
decoded_imgs_32 = autoencoder_32.predict(x_test_noisy)
decoded_imgs_32_noiseless = autoencoder_32.predict(x_test)
decoded_imgs_16 = autoencoder_16.predict(x_test_noisy)
decoded_imgs_16_noiseless = autoencoder_16.predict(x_test)
```

In [292]:

```
n = 6 # how many digits we will display
plt.figure(figsize=(20, 12))
for i in range(6):
    # display original noisy images
    ax = plt.subplot(6, n, i + 1)
    plt.imshow(x_test_noisy[plt_368_inx[i]].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction of 32D
    ax = plt.subplot(6, n, i + 1 + n)
    plt.imshow(decoded_imgs_32[plt_368_inx[i]].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction of 16D
    ax = plt.subplot(6, n, i + 1 + 2*n)
    plt.imshow(decoded_imgs_16[plt_368_inx[i]].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction of 32D on images without noise
    ax = plt.subplot(6, n, i + 1 + 3*n)
    plt.imshow(decoded_imgs_32_noiseless[plt_368_inx[i]].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction of 16D on images without noise
    ax = plt.subplot(6, n, i + 1 + 4*n)
    plt.imshow(decoded_imgs_16_noiseless[plt_368_inx[i]].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
```

```
    # display original digits
    ax = plt.subplot(6, n, i + 1 + 5*n)
    plt.imshow(x_test[plt_368_inx[i]].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

plt.show()
```



1st row: noisy image
2nd row: denoised image of 32D CNN
3rd row: denoised image of 16D CNN
4th row: reconstructed image of 32D CNN on clean images
5th row: reconstructed image of 16D CNN on clean images
6th row: original clean image

Comment:

Since we added only small fraction of noise, it's hard to tell which autoencoder performs better. But the results for 16D-CNN and 32D-CNN are comparable to the original images. It's also hard to determine the effect of removing noise from handwritten digits 3, 6, and 8.

Quantitatively compare the quality of denoising with signal intensity

In [294]:

```
# Get average signal intensity by digits and encoder
def getDigitSignalIntensity(encoder,digit,noise):
    indices = list(np.where(y_test == digit)[0])
    if noise == True:
        encoded_imgs = encoder.predict(np.take(x_test_noisy, indices, axis = 0))
    else:
        encoded_imgs = encoder.predict(np.take(x_test, indices, axis = 0))
    return encoded_imgs.mean()
```

In [295]:

```
import pandas as pd
```

```python
import pandas as pd
# Create a data frame to showcase the results
digits = list(range(10))
digit_signal_16 = {digit:getDigitSignalIntensity(encoder_16,digit,True) for digit in digits}
digit_signal_32 = {digit:getDigitSignalIntensity(encoder_32,digit,True) for digit in digits}
digit_signal_16_clean = {digit:getDigitSignalIntensity(encoder_16,digit,False) for digit in digits}
digit_signal_32_clean = {digit:getDigitSignalIntensity(encoder_32,digit,False) for digit in digits}
```

In [296]:

```python
digit_signal_df = pd.DataFrame([digit_signal_16,digit_signal_32,
                                digit_signal_16_clean,digit_signal_32_clean],
                                index = ['16D-CNN-Noisy','32D-CNN-Noisy',
                                         '16D-CNN-Clean','32D-CNN-Clean'])
digit_signal_df
```

Out[296]:

	0	1	2	3	4	5	6	7	8	9
16D-CNN-Noisy	0.525209	0.348249	0.491905	0.485120	0.442398	0.468604	0.471544	0.426098	0.491441	0.442848
32D-CNN-Noisy	0.396922	0.299583	0.380786	0.377439	0.352754	0.368156	0.367764	0.343289	0.379044	0.352243
16D-CNN-Clean	0.529994	0.340455	0.494217	0.487181	0.441563	0.469346	0.472268	0.423497	0.493262	0.441237
32D-CNN-Clean	0.404399	0.293407	0.386574	0.383122	0.355082	0.372437	0.370913	0.343362	0.383416	0.353019

Comment:

The data frame above shows the signal intensity by digits (each column represents a digit) with 2 different encoders on both noisy and clean data. We can find out that on clean images, the signal intensity is stronger than noisy images using the same encoder.