Problem 1

1. **image = china[150:220, 130:250]** takes a specific portion of the image, specifically, pixels from row 150 to row 220 and from column 130 to column 250.

2. Lines 71-72 are setting up an operator for applying the filter **fmap** to the input image **X**, with strides being 1 in all four dimensions and with zero padding.

3. **X_reshaped = tf.reshape(X, shape=[-1, height, width, channels])** is reshaping the input **X** into four dimensional tensors, [batch size, height, width, channels]. This ensures all inputs have the same dimension. Note that -1 indicates that the first dimension needs to be computed.

4. Lines 47-53 are constructing two convolution layers. The arguments of **tf.layers.conv2d** are, respectively, input being fed into the convolution layer, number of filters, size of kernel, size of stride, padding method, activation function and the name of the convolution layer.

5. Lines 55-56 are constructing a max pooling layer then reshaping the result.

The arguments of **tf.nn.max_pool** are, respectively, input being fed into the max pooling layer, size of window for each dimension (**ksize**), size of stride for each dimension (**strides**) and padding method.

pool3 needs to be reshaped because following the max pooling layer is one or more fully connected layers in CNN.

6. **fc1** is the name of the first (and in **tfconv1_2.py** the only) fully connected layer in CNN.

7. **Y_proba** is a probability distribution of a given input belonging to different class labels.

8. **tf.nn.in_top_k(predictions, targets, k, name=None)** returns a boolean of whether targets are in the top k predictions. Lines 71-73 in **tfconv1_2.py**, for each input, first return true if the input is predicted to have the same label as the true label (false otherwise), and then compute accuracy as the percentage of correct labels.

9. The results of the pooling layer are fed into the fully connected layer, then units of this fully connected layer are randomly set to 0 by probability **fc1_dropout_rate** at each update during training time. Consequently, the active units are scaled by $\frac{1}{(1-rate)}$, so that their sum is unchanged during training/inference time. Dropout helps prevent overfitting.

10. In general, early stopping is a technique for controlling overfitting in machine learning models, especially in neural networks, by stopping training before the weights have converged. Usually training is stopped when the performance has stopped improving on the test set.

In **tfconv1_3.py**, during each epoch we check the loss value (**loss_eval**) against the best loss value so far (**best_loss_eval**). If the loss value has not improved for more than 20 (**max_checks_without_progress**) epochs, we stop the training.

Problem 2

1. Principal components are constructed in such a manner that the first principal component accounts for the largest possible variance in the dataset. Same calculation applies to the second principal component, such that it is uncorrelated with the first principal component and that it accounts for the next largest possible variance. The same goes on for the rest of the principal components.

s, u, v = tf.svd() is useful in implementing PCA since it computes the singular value decomposition (SVD) of the input matrix X.

$$X = U\Sigma W^T$$

Σ is an $n \times p$ rectangular diagonal matrix of positive numbers (singular values of X, or equivalently, square root of eigenvalues of matrix X^TX). U is a $n \times n$ matrix, where the columns are orthogonal unit vectors of length n (left singular vectors of X). W is a $p \times p$ matrix, where the columns are orthogonal unit vectors of length p (right singular vectors of X).

In fact,

$$X^TX = W\Sigma^TU^TU\Sigma W^T$$

$$= W\Sigma^T\Sigma W^T$$

$$= W\hat{\Sigma}^2 W^T$$

$\hat{\Sigma}$ is the square diagonal matrix with the singular values of X plus excess zeroes chopped off that satisfies $\hat{\Sigma}^2 = \Sigma^T\Sigma$. Note that the right singular vectors W of X are equivalent to the eigenvectors of X^TX.

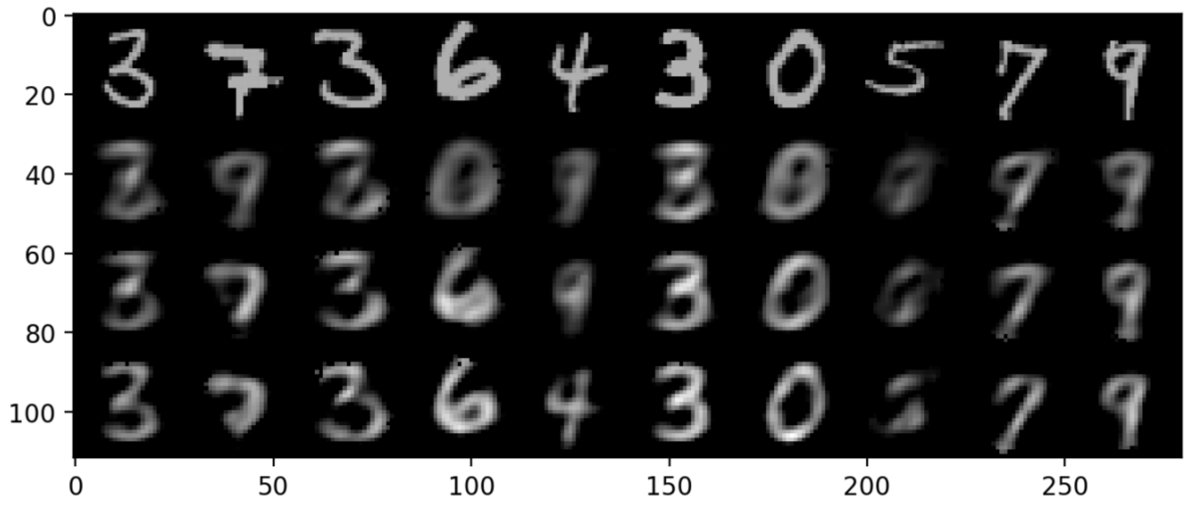
In PCA, the projection matrix is constructed by selecting the $\boldsymbol{k}$ eigenvectors that corresponds to the $\boldsymbol{k}$ largest eigenvalues of X^TX, where $\boldsymbol{k}$ is the number of dimensions of the new feature subspace. There $\boldsymbol{k}$ eigenvectors are given by the last return value of **s, u, v = tf.svd()**.

Note that although the 2 representations can both be used to compute the projection matrix in PCA, the SVD approach is more computationally efficient than the covariance matrix approach.

2. See **prob2_2.py**.

For better illustration, the results below use 10 test examples, and 4, 8, 16 neurons in the hidden layer, respectively. Feel free to modify the parameters.

Also, the images are combined together into one single image, instead of 8 images in total.



It can be observed that the more neurons there are in the hidden layer, the clearer the reconstructed images are, since with more neurons the hidden layer is able to learn more details about the input images.

3. Both PCA and autoencoder are unsupervised learning methods, and can be used for dimensionality reduction. In fact, a single layer autoencoder with linear activation function is nearly equivalent to PCA, in a sense that the subspace spanned by autoencoder is the same as that of PCA.

Obviously autoencoder can achieve non-linearity, but this is not possible with PCA.

4. Convolutional autoencoder can be created by replacing the fully connected layers of linear autoencoder with convolutional layers (i.e., filters). Along with pooling layers, this helps the network extract visual features from the input images, and therefore obtain a much more accurate latent space representation, compared with linear autoencoder.

Convolutional autoencoder scales well to high dimensional images because the number of parameters required to product an activation map remains the same regardless of the size of the input. In contrast, linear autoencoder ignores the 2D image structure since input images must be unrolled into a single vector during training.

5. Both denoising autoencoder and variational autoencoder are variations of autoencoder, and are composed of an encoder followed by a decoder.

Variational autoencoder is a generative model that models the underlying probability distribution of the input data and samples new data from it. The main difference between VAE and AE is that the latent space is continuous in VAE whereas it is discontinuous in AE. This difference allows VAE to generate new data instead of repeating existing input data.

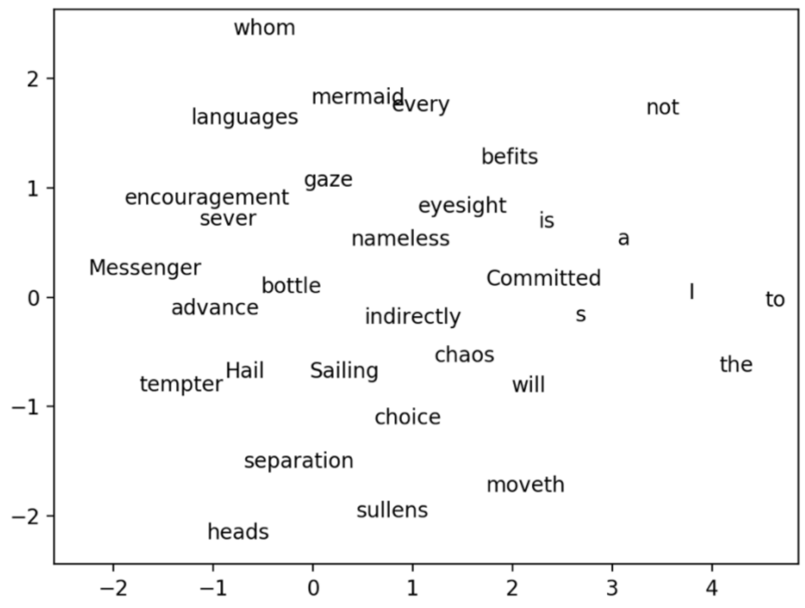
Denoising autoencoder corrupts the input by setting some of inputs to zero and trains to recover the original undistorted input. This helps the denoising autoencoder avoid learning the identity function. By doing this, the denoising autoencoder is able to learn to extract essential features that are useful for modeling the input distribution.

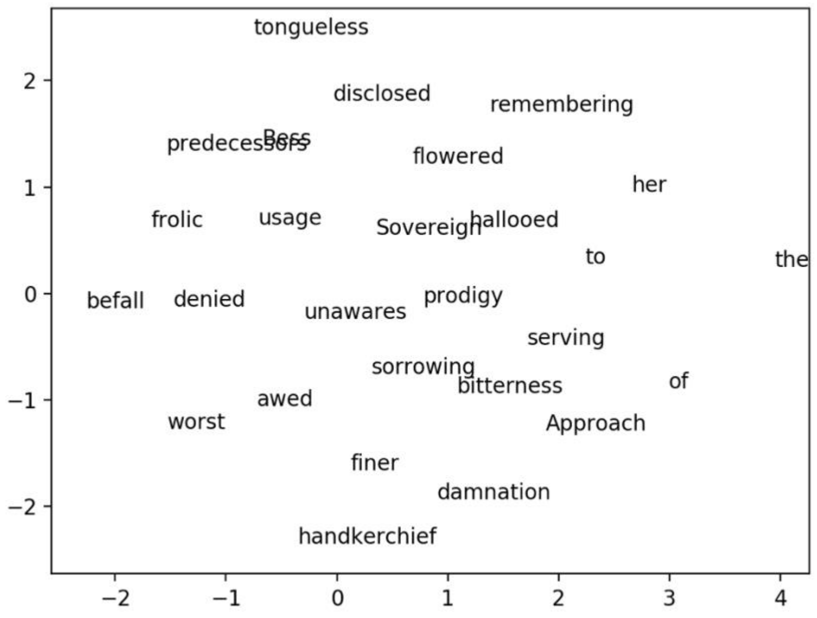
Problem 3

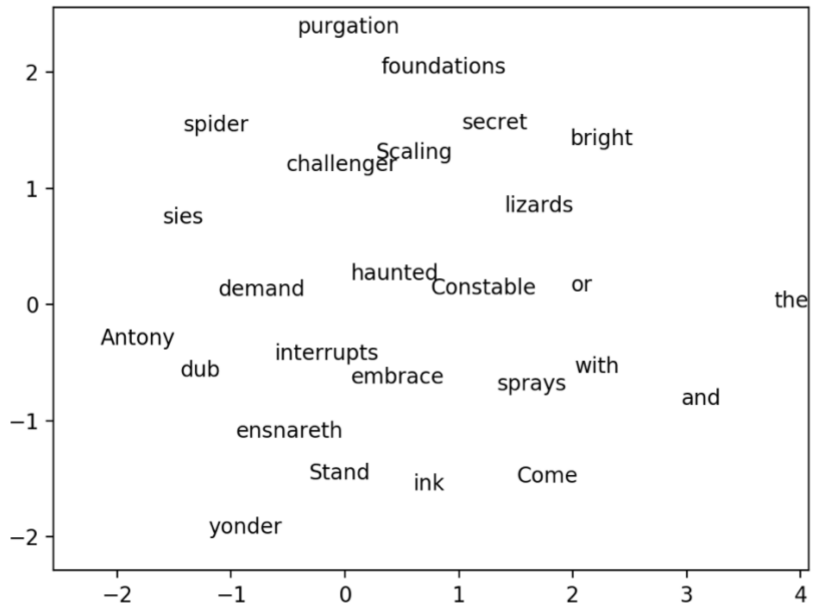
1 – 5. See **rnn.py**. The input text is from Shakespeare. It can be found on Andrej Karpathy's Github.

https://github.com/karpathy/char-rnn/blob/master/data/tinyshakespeare/input.txt

Plot randomly selected words for a couple of time.







From the 3rd plot, it can be observed that prepositions (with), conjunctions (and, or) and articles (the) appear to be close with each other, whereas a top cluster consists of nouns (purgation, foundations, spider, secret, etc.), and finally somewhere around the bottom left corner, there are a few verbs (dub, demand, interrupts, embrace, sprays, stand, etc.).

The same pattern can be observed in the 1st plot and the 2nd plot as well.