keras autoencoder vs PCA

Asked 4 years, 1 month ago Active 1 year, 10 months ago Viewed 6k times

4

4

I am playing with a toy example to understand PCA vs keras autoencoder

I have the following code for understanding PCA:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn import decomposition
from sklearn import datasets

iris = datasets.load_iris()
X = iris.data
pca = decomposition.PCA(n_components=3)
pca.fit(X)

pca.explained_variance_ratio_
array([ 0.92461621,  0.05301557,  0.01718514])

pca.components_
array([[ 0.36158968, -0.08226889,  0.85657211,  0.35884393],
       [ 0.65653988,  0.72971237, -0.1757674 , -0.07470647],
       [-0.58099728,  0.59641809,  0.07252408,  0.54906091]])
```

I have done a few readings and play codes with keras including this one.

However, the reference code feels too high a leap for my level of understanding.

Does someone have a short auto-encoder code which can show me

(1) how to pull the first 3 components from auto-encoder

(2) how to understand what amount of variance the auto-encoder captures

(3) how the auto-encoder components compare against PCA components

python-2.7 keras pca autoencoder Edit tags

Share Edit Follow Close Flag

asked Aug 16 '17 at 20:15
Zanam
3,806 11 45 96

stackoverflow.com/questions/47842931/... any suggestions? – Dexter Dec 17 '17 at 16:22

3 Answers

Active Oldest Votes

2

The earlier answer cover the whole thing, however I am doing the analysis on the Iris data - my code comes with a slightly modificiation from this post which dives further into the topic. As it was request, lets load the data

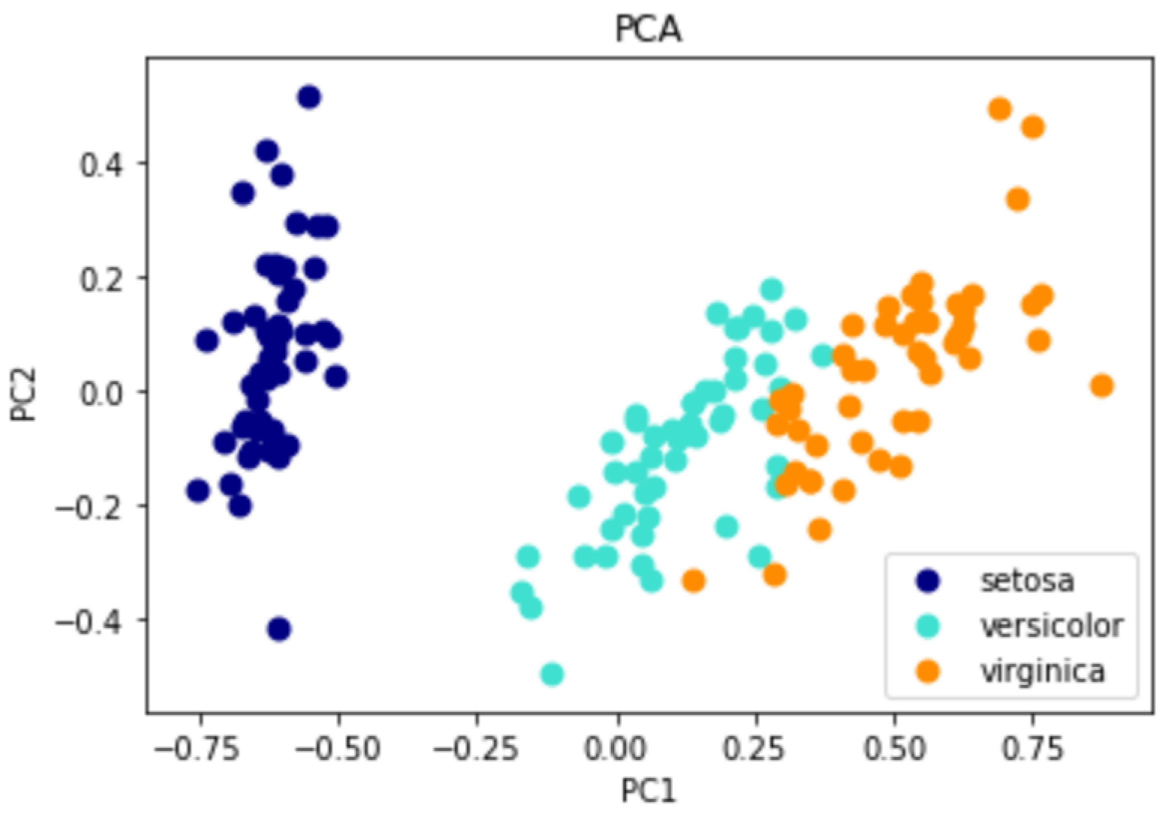
```
from sklearn.datasets import load_iris
from sklearn.preprocessing import MinMaxScaler

iris = load_iris()
X = iris.data
y = iris.target
target_names = iris.target_names

scaler = MinMaxScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)
```

Let's do a regular PCA

```
from sklearn import decomposition
pca = decomposition.PCA()
pca_transformed = pca.fit_transform(X_scaled)
plot3clusters(pca_transformed[:,:2], 'PCA', 'PC')
```



A very simple AE model with linear layers, as the earlier answer pointed out with ... *the first reference, one linear hidden layer and the mean squared error criterion is used to train the network, then the k hidden units learn to project the input in the span of the first k principal components of the data.*

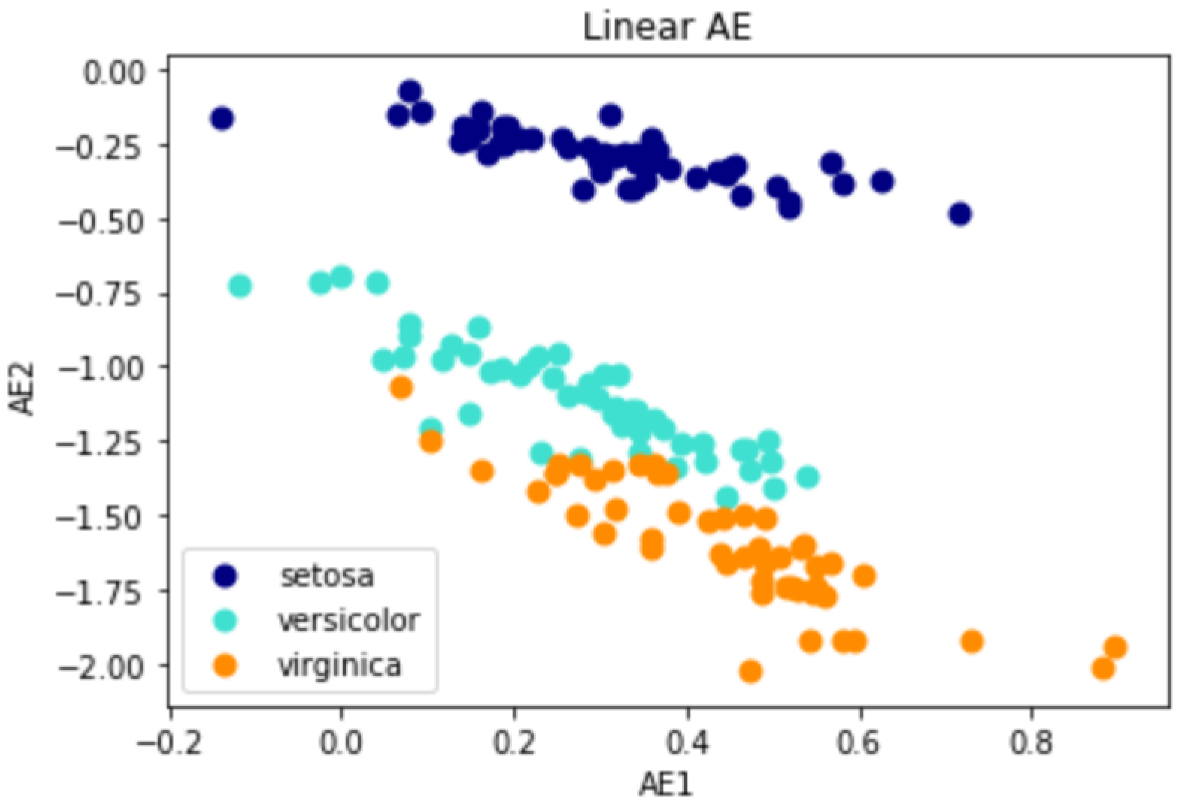
```
from keras.layers import Input, Dense
from keras.models import Model
import matplotlib.pyplot as plt
```

```
#create an AE and fit it with our data using 3 neurons in the dense layer using keras'
functional API
input_dim = X_scaled.shape[1]
encoding_dim = 2
input_img = Input(shape=(input_dim,))
encoded = Dense(encoding_dim, activation='linear')(input_img)
decoded = Dense(input_dim, activation='linear')(encoded)
autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='mse')
print(autoencoder.summary())

history = autoencoder.fit(X_scaled, X_scaled,
                epochs=1000,
                batch_size=16,
                shuffle=True,
                validation_split=0.1,
                verbose = 0)

# use our encoded layer to encode the training input
encoder = Model(input_img, encoded)
encoded_input = Input(shape=(encoding_dim,))
decoder_layer = autoencoder.layers[-1]
decoder = Model(encoded_input, decoder_layer(encoded_input))
encoded_data = encoder.predict(X_scaled)

plot3clusters(encoded_data[:,:2], 'Linear AE', 'AE')
```



You can look into the loss if you want

```
#plot our loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model train vs validation loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper right')
plt.show()
```

The function to plot the data

```
def plot3clusters(X, title, vtitle):
    import matplotlib.pyplot as plt
    plt.figure()
    colors = ['navy', 'turquoise', 'darkorange']
    lw = 2

    for color, i, target_name in zip(colors, [0, 1, 2], target_names):
        plt.scatter(X[y == i, 0], X[y == i, 1], color=color, alpha=1., lw=lw,
label=target_name)

    plt.legend(loc='best', shadow=False, scatterpoints=1)
    plt.title(title)
    plt.xlabel(vtitle + "1")
    plt.ylabel(vtitle + "2")
    return(plt.show())
```

Regarding explaining the variability, using non-linear hidden function, leads to other approximation similar to ICA / TSNE and others. Where the idea of variance explanation is not there, still one can look into the convergence.

Share Edit Follow Flag

answered Nov 28 '19 at 13:08
user702846
4,345 4 37 61

0

This post is hidden. It was deleted 3 years ago by Bhargav Rao ♦.

mega super example ! I have look over the internet for this type of example for a long time ! (classification with autoencoders)

One good thing It would be nice to see (and applied to MNIST) is using 2D autoencoders.

Just one detail in line:

```
y = Dense(height * width//256, activation='relu')(x)
```

should not be:

```
y = Dense(height * width//256, activation='relu')(encoder)
```

also just for theoretical application after "unsupervised train" should not use something like:

```
y.trainable=False
```

Share Edit Follow Flag

edited Dec 19 '17 at 1:56
Arun Vinoth - MVP
20.9k 14 49 144

answered Dec 19 '17 at 1:34
rjpg
134 1 11

This does not provide an answer to the question. Once you have sufficient reputation you will be able to comment on any post; instead, provide answers that don't require clarification from the asker. - From Review – Tom Aranda Dec 19 '17 at 2:04

Comments disabled on deleted / locked posts / reviews

6

First of all, the aim of an autoencoder is to learn a representation (encoding) for a set of data, typically for the purpose of dimensionality reduction. So, the target output of the autoencoder is the autoencoder input itself.

It is shown in [1] that If there is one linear hidden layer and the mean squared error criterion is used to train the network, then the `k` hidden units learn to project the input in the span of the `first k principal components` of the data. And in [2] you can see that If the hidden layer is nonlinear, the autoencoder behaves differently from PCA, with the ability to capture multi-modal aspects of the input distribution.

Autoencoders are data-specific, which means that they will only be able to compress data similar to what they have been trained on. So, the usefulness of features that have been learned by hidden layers could be used for evaluating the efficacy of the method.

For this reason, one way to evaluate an autoencoder efficacy in dimensionality reduction is cutting the output of the middle hidden layer and compare the accuracy/performance of your desired algorithm by this reduced data rather than using original data. Generally, PCA is a linear method, while autoencoders are usually non-linear. Mathematically, it is hard to compare them together, but intuitively I provide an example of dimensionality reduction on MNIST dataset using Autoencoder for your better understanding. The code is here:

```
from keras.datasets import mnist
from keras.models import Model
from keras.layers import Input, Dense
from keras.utils import np_utils
import numpy as np

num_train = 60000
num_test = 10000

height, width, depth = 28, 28, 1 # MNIST images are 28x28
num_classes = 10 # there are 10 classes (1 per digit)

(X_train, y_train), (X_test, y_test) = mnist.load_data()

X_train = X_train.reshape(num_train, height * width)
X_test = X_test.reshape(num_test, height * width)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

X_train /= 255 # Normalise data to [0, 1] range
X_test /= 255 # Normalise data to [0, 1] range

Y_train = np_utils.to_categorical(y_train, num_classes) # One-hot encode the labels
Y_test = np_utils.to_categorical(y_test, num_classes) # One-hot encode the labels

input_img = Input(shape=(height * width,))

x = Dense(height * width, activation='relu')(input_img)

encoded = Dense(height * width//2, activation='relu')(x)
encoded = Dense(height * width//8, activation='relu')(encoded)

y = Dense(height * width//256, activation='relu')(x)

decoded = Dense(height * width//8, activation='relu')(y)
decoded = Dense(height * width//2, activation='relu')(decoded)

z = Dense(height * width, activation='sigmoid')(decoded)
model = Model(input_img, z)

model.compile(optimizer='adadelta', loss='mse') # reporting the accuracy

model.fit(X_train, X_train,
      epochs=10,
      batch_size=128,
      shuffle=True,
      validation_data=(X_test, X_test))

mid = Model(input_img, y)
reduced_representation =mid.predict(X_test)

out = Dense(num_classes, activation='softmax')(y)
reduced = Model(input_img, out)
reduced.compile(loss='categorical_crossentropy',
          optimizer='adam',
          metrics=['accuracy'])

reduced.fit(X_train, Y_train,
      epochs=10,
      batch_size=128,
      shuffle=True,
      validation_data=(X_test, Y_test))


 scores = reduced.evaluate(X_test, Y_test, verbose=1)
 print("Accuracy: ", scores[1])
```

It produces a $y\in \mathbb{R}^{3}$ (almost like what you get by `decomposition.PCA(n_components=3)`). For example, here you see the outputs of layer `y` for a digit `5` instance in dataset:

```
  class  y_1    y_2     y_3
  5      87.38  0.00    20.79
```

As you see in the above code, when we connect layer `y` to a `softmax` dense layer:

```
mid = Model(input_img, y)
reduced_representation =mid.predict(X_test)
```

the new model `mid` give us a good classification accuracy about `95%`. So, it would be reasonable to say that `y`, is an efficiently extracted feature vector for the dataset.

References:

[1]: Bourlard, Hervé, and Yves Kamp. "Auto-association by multilayer perceptrons and singular value decomposition." Biological cybernetics 59.4 (1988): 291-294.

[2]: Japkowicz, Nathalie, Stephen Jose Hanson, and Mark A. Gluck. "Nonlinear autoassociation is not equivalent to PCA." Neural computation 12.3 (2000): 531-545.

Share Edit Follow Flag

answered Sep 12 '17 at 10:09

moh
750 1 9 22

stackoverflow.com/questions/47842931/... any suggestions? – Dexter Dec 17 '17 at 16:19