Pose_data_with_keras

October 14, 2017

1 Modeling Pose Data with Keras

This is a collection of findings on modeling 3D point clouds with keras. All data is raw (*until processed*) and obtained using a Kinect sensor. A standard C# library obtains joint-point clouds which can be worked with.

1.1 Introduction & Specific Case

In this situation, I will be trying to predict *yoga poses*, specifically the poses concerning **suryanamaskaar**. First, let us load the initial. Note that the data is already *pickled* and just has to be loaded from the concerned files.

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import pickle
        filename_X = 'X.pkl'
        filename_y = 'y.pkl'
        filename_labels = 'labels.pkl'
        labels = pickle.load(open(filename_labels, 'rb'))
        X = pickle.load(open(filename_X, 'rb'))
        y = pickle.load(open(filename_y, 'rb'))
        X.shape, y.shape, labels.shape
```

```
Out[1]: ((161, 77), (161, 1), (25, 3))
```

1.2 Data Cleaning & Pruning

Herein, the data contains multiple joints and other details that can be pruned according to our use case. In this situation, analysis of the diagrams has shown that some joints are prone more to occlusion and can affect accuracy hence can be removed. In addition to this, the data can be normalized for better accuracy.

```
In [2]: # removing the last two angles
        X = X[:, :-2]
        # applying normalization
        X = (X - np.mean(X, 0)) / np.std(X, 0)
        assert(X.shape[0] == y.shape[0])
```

```
In [3]: poses = []
        for x in y:
            if x[0, 0] not in poses:
                poses.append(x[0, 0])
        print("Full sample space", sorted(poses))
```

```
Full sample space [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0]
```

Now, in the list of poses, a few of the poses are repeated, ie, output, say *5.0* and *3.0* refer to the same pose. Let us reduce the y dataset size by mapping the larger sample space to a reduced space.

```
In [4]: mapping = { 12: 1, 10: 3, 9: 4, 8: 5, 11: 8 }
        for ex in y:
            try:
                ex[0, 0] = mapping[ex[0, 0]]
            except KeyError:
                pass
        exec(In[3])
```

```
Full sample space [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0]
```

1.2.1 Data finalization

Now that we have pruned our data, let us recap.

```
In [5]: print("X Dimens ", X.shape)
        print("y dimens ", y.shape)
        print(mapping)
        print("Data labels", labels)
```

```
X Dimens  (161, 75)
y dimens  (161, 1)
{12: 1, 10: 3, 9: 4, 8: 5, 11: 8}
Data labels [['head x' 'head y' 'head z']
 ['AnkleL X' 'AnkleL Y' 'AnkleL Z']
 ['AnkleR X' 'AnkleR Y' 'AnkleR Z']
 ['ElbowL X' 'ElbowL Y' 'ElbowL Z']
 ['ElbowR X' 'ElbowR Y' 'ElbowR Z']
 ['FootL X' 'FootL Y' 'FootL Z']
 ['FootR X' 'FootR Y' 'FootR Z']
 ['HandL X' 'HandL Y' 'HandL Z']
 ['HandR X' 'HandR Y' 'HandR Z']
 ['HandTipL X' 'HandTipL Y' 'HandTipLZ']
 ['HandTipR X' 'HandTipR Y' 'HandTipR Z']
 ['HipL X' 'HipL Y' 'HipL Z']
 ['HipR X' 'HipR Y' 'HipR Z']
 ['KneeL X' 'KneeL Y' 'KneeL Z']
 ['KneeR X' 'KneeR Y' 'KneeR Z']
 ['Neck X' 'Neck Y' 'Neck Z']
 ['ShoulderL X' 'ShoulderL Y' 'ShoulderL Z']
 ['ShoulderR X' 'ShoulderR Y' 'ShoulderR Z']
 ['SpineBase X' 'SpineBase Y' 'SpineBase Z']
 ['SpineMid X' 'SpineMid Y' 'SpineMid Z']
 ['SpineShoulder X' 'SpineShoulder Y' 'SpineShoulder Z']
 ['ThumbL X' 'ThumbL Y' 'ThumbL Z']
 ['ThumbR X' 'ThumbR Y' 'ThumbR Z']
 ['WristL X' 'WristL Y' 'WristL Z']
 ['WristR X' 'WristR Y' 'WristR Z']]
```

1.3 Model Training and Pose Estimation process

Now that we have our data cleaned, we can proceed to training the actual model and getting some test data. We will use *Keras* with a *tensorflow* backend. We can train different models to get the best perfomance

```
In [6]: # this will be the store for all the models
        models = []
        # import necessary keras packages
        from keras.models import Sequential
        from keras.layers import Dense
```

```
Using TensorFlow backend.
```

The first model we will be training is a simple model with **reLu** activation functions and single node output. This model will have 3 hidden layers with a *X.shape* * 2, *X.shape*, *X.shape* * 2, network layout

```
In [7]: simple_model = Sequential()
        simple_model.add(Dense(X.shape[-1] * 2, input_dim=X.shape[-1], activation='relu'))
        simple_model.add(Dense(X.shape[-1], activation='relu'))
        simple_model.add(Dense(X.shape[-1] * 2, activation='relu'))
        # adding the final output node
        simple_model.add(Dense(1, activation='sigmoid'))
        # compiling the made model
        from keras.metrics import binary_accuracy
        simple_model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'
        simple_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 150)               11400
_________________________________________________________________
```

```
dense_2 (Dense)              (None, 75)                11325
_________________________________________________________________
dense_3 (Dense)              (None, 150)               11400
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 151
=================================================================
Total params: 34,276
Trainable params: 34,276
Non-trainable params: 0
_________________________________________________________________
```

Now we have to scale our y = data for this model, since it's output function is a sigmoid function which returns values between [0, 1]

```
In [8]: model_X = X
        model_y = y / 8
        test_size = int( (20 / 100 ) * X.shape[0])
        training_size = int(X.shape[0] - test_size)
        batch_size = int(0.2 * training_size)
        iterations = int(1e4)
```

```
In [9]: simple_model.fit(model_X[:training_size], model_y[:training_size], verbose=0, epochs=i
```

```
Out[9]: <keras.callbacks.History at 0x7f29c3f38048>
```

Now that our model has been trained, time to do some analytics. We have to analyse the performance on the test set, training set, as well as the entire dataset. Let us define a function to do this testing

```
In [10]: def test_data(test_model, X, y, training_set_size, test_set_size):
             scores = {
                 '0test_types' : test_model.metrics_names,
                 '1training_data' : test_model.evaluate(X[:training_set_size], y[:training_set
                 '2test_data' : test_model.evaluate(X[-test_set_size:], y[-test_set_size:], ve
                 '3dataset' : test_model.evaluate(X, y, verbose=0)
             }
             return scores
```

```
In [11]: test_data(simple_model, model_X, model_y, training_size, test_size)
```

```
Out[11]: {'0test_types': ['loss', 'acc', 'binary_accuracy'],
          '1training_data': [0.51405878931052928,
           0.093023255813953487,
           0.093023255813953487],
          '2test_data': [0.51576805114746094, 0.09375, 0.09375],
          '3dataset': [0.51439851670531755, 0.093167701863354033, 0.093167701863354033]}
```

As seen the accuracy isn't that great, average around *9 - 10%* irrespective of the dataset size. One thing we can do is convert the y label into a 8 feature vector and set the last function to be a **sigmoid activation** function.

```
In [12]: new_y = np.zeros((X.shape[0], len(poses)))
         for i, ex in enumerate(y):
             new_y[i, int(ex[0, 0] - 1)] = 1
         print(new_y)

[[ 1.  0.  0. ...,  0.  0.  0.]
 [ 0.  0.  0. ...,  0.  0.  0.]
 [ 1.  0.  0. ...,  0.  0.  0.]
 ...,
 [ 0.  0.  0. ...,  0.  0.  0.]
 [ 0.  0.  0. ...,  0.  1.  0.]
 [ 0.  0.  1. ...,  0.  0.  0.]]
```

```
In [13]: new_model = Sequential()
         new_model.add(Dense(X.shape[-1] * 2, input_dim=X.shape[-1], activation='relu'))
         new_model.add(Dense(X.shape[-1], activation='relu'))
         new_model.add(Dense(X.shape[-1] * 2, activation='relu'))
         new_model.add(Dense(len(poses), activation='sigmoid'))
         new_model.summary()
         new_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy', 

_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_5 (Dense)              (None, 150)               11400     
_________________________________________________________________
dense_6 (Dense)              (None, 75)                11325     
_________________________________________________________________
dense_7 (Dense)              (None, 150)               11400     
_________________________________________________________________
dense_8 (Dense)              (None, 8)                 1208      
=================================================================
Total params: 35,333
Trainable params: 35,333
Non-trainable params: 0
_________________________________________________________________
```

```
In [14]: new_model.fit(X[:training_size], new_y[:training_size], verbose=0, epochs=iterations,

Out[14]: <keras.callbacks.History at 0x7f29c3b54978>

In [15]: test_data(new_model, X, new_y, training_size, test_size)

Out[15]: {'0test_types': ['loss', 'acc', 'binary_accuracy'],
          '1training_data': [1.0240115966200759e-07, 1.0, 1.0],
          '2test_data': [0.12658296525478363, 0.98828125, 0.98828125],
          '3dataset': [0.025159432141241184, 0.99767080745341619, 0.99767080745341619]}
```

As seen, just by changing our output array to become a *binary vector* which stores probability of pose, we have gotten a *tremendous increase*, all the way to **93 - 96%** across the board. Is is evident that our new model is much better than the previous one.

1.4 Testing for errors

Now let us test the examples manually to find patterns in the errors.

```
In [16]: # function to get easy prediction format
         def get_prediction(X, i):
             f = np.ndarray.tolist(new_model.predict(X[i]))[0]
             return f.index(max(f)) + 1
         import random
         predictions = [(get_prediction(X, i), y[i, 0]) for i in range(X.shape[0])]
         for i, (predicted, actual) in enumerate(predictions):
             if predicted != actual:
                 print("%d [WRONG] " % i, predicted, "!=", actual)
```

```
140 [WRONG]  1 != 2.0
144 [WRONG]  4 != 2.0
```

Now that this model is pretty satisfactory, we can save it for future use.

```
In [17]: import json
         j = json.loads(new_model.to_json())
         json.dump(j, open('trained_model.json', 'w'))
         new_model.save_weights('trained_model.h5')
```