Pose_data_with_keras

October 16, 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 **suryana-maskaar**. 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'
        all_labels = pickle.load(open(filename_labels, 'rb'))
        labels = np.array(all_labels['old_labels'])
        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,))
```

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' 'AnkleL' 'AnkleR' 'ElbowL' 'ElbowR' 'FootL' 'FootR' 'HandL' 'HandR'
 'HandTipL' 'HandTipR' 'HipL' 'HipR' 'KneeL' 'KneeR' 'Neck' 'ShoulderL'
 'ShoulderR' 'SpineBase' 'SpineMid' 'SpineShoulder' 'ThumbL' 'ThumbR'
 'WristL' 'WristR']
```

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 0x7f3c5813aef0>
```

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': [0.51429547750672633,
           0.093023255813953487,
           0.093023255813953487],
          '2test_data': [0.51569056510925293, 0.09375, 0.09375],
          '3dataset': [0.51457276773748928, 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', l
```

```
_________________________________________________________________
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 0x7f3c4f6c3be0>

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': [0.18188676238059998, 0.984375, 0.984375],
          '3dataset': [0.036151491197304063, 0.99689440993788825, 0.99689440993788825]}
```

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
159 [WRONG]  5 != 7.0
```

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('trained_model.h5')
```

1.5 Testing with new data

Now that the model has been trained, we can proceed to test it with new data. Let us load the model from the file, and proceed. The data is stored in a pickled format.

```
In [18]: from keras.models import load_model
         try:
             del new_model
             del simple_model
         except:
             pass
         newdata_file = 'dataset_new.pkl'

         cur_model = load_model('trained_model.h5')
         new_data = pickle.load(open(newdata_file, 'rb'))

In [19]: cur_model.summary()
         new_data = np.matrix(new_data)
         new_data = (new_data - np.mean(new_data, 0)) / np.std(new_data, 0)
```

```
_________________________________________________________________
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
_________________________________________________________________
```

Now that the data has been normalized, we can try prediction of a few random poses. However, the data isn't sorted the same way the previous data was, so we will have to take care of that. All the sorting has been done by using the pandas module. The data has been pickled and copied to a file. Let us load the data from the file:

In [20]:
```
import pandas as pd
datafile = 'all_data_X.pkl'
old_dataset = pd.read_pickle('old_data.pkl')
new_dataset = pd.read_pickle('new_data.pkl')
new_dataset[:10]
```

Out[20]:

	headX	headY	headZ	AnkleLX	AnkleLY	AnkleLZ	AnkleRX	\
0	0.280684	0.988059	3.263694	0.254301	-0.444253	3.343855	0.338832	
1	0.281372	0.987621	3.267906	0.255337	-0.442164	3.346285	0.342003	
2	0.282592	0.988507	3.273247	0.257084	-0.437238	3.348634	0.343773	
3	0.284382	0.990231	3.278028	0.257059	-0.437400	3.349086	0.343529	
4	0.285024	0.990090	3.278895	0.256005	-0.438547	3.349479	0.343443	
5	0.288270	0.991880	3.282324	0.252805	-0.440628	3.351120	0.344895	
6	0.290292	0.991635	3.283504	0.253375	-0.446780	3.348309	0.344739	
7	0.294620	0.991994	3.287684	0.254559	-0.441441	3.350679	0.344636	
8	0.295628	0.991814	3.287842	0.253678	-0.440837	3.351065	0.342701	
9	0.297428	0.991606	3.288206	0.253852	-0.440620	3.351242	0.342007	

	AnkleRY	AnkleRZ	ElbowLX	...	ThumbLZ	ThumbRX	ThumbRY	\
0	-0.448342	3.342944	0.059723	...	3.154000	0.526344	0.163979	
1	-0.443822	3.348526	0.063121	...	3.158571	0.522428	0.176023	
2	-0.441450	3.351580	0.066551	...	3.180250	0.524024	0.192591	
3	-0.441425	3.351696	0.070192	...	3.174875	0.514929	0.196622	
4	-0.441609	3.351665	0.075397	...	3.217080	0.519769	0.213355	
5	-0.435369	3.354405	0.078418	...	3.196308	0.518031	0.202581	
6	-0.435814	3.353754	0.080520	...	3.191308	0.518552	0.228914	
7	-0.435997	3.353716	0.081978	...	3.197600	0.516335	0.197266	
8	-0.437185	3.353600	0.082893	...	3.201401	0.520284	0.148180	
9	-0.438000	3.352500	0.084353	...	3.205637	0.516971	0.171947	

	ThumbRZ	WristLX	WristLY	WristLZ	WristRX	WristRY	WristRZ
0	3.212111	0.060925	0.271715	3.224301	0.527751	0.265957	3.274888
1	3.200111	0.066423	0.269686	3.223008	0.522764	0.277916	3.266220
2	3.208750	0.074040	0.280027	3.225738	0.521314	0.278699	3.263463
3	3.204286	0.079813	0.286167	3.227628	0.519758	0.277824	3.262696
4	3.213834	0.087647	0.288218	3.229130	0.518375	0.278476	3.260859
5	3.211932	0.108080	0.250589	3.224464	0.518165	0.278385	3.260657
6	3.229000	0.100447	0.287098	3.232600	0.517889	0.278280	3.260650
7	3.215000	0.107225	0.267964	3.234044	0.517034	0.278482	3.261563
8	3.198500	0.106880	0.267847	3.237896	0.516086	0.278092	3.262226
9	3.216100	0.103504	0.284026	3.257442	0.513883	0.276644	3.265942

[10 rows x 75 columns]

In [21]:
```
new_data = np.matrix(new_dataset.get_values())
new_data = (new_data - np.mean(new_data, 0)) / np.std(new_data, 0)

# setup pose storing data
```

```
poses = {}
pose_indices = {}
for _ in range(1, 9):
    poses[_] = 0
    pose_indices[_] = []

# storing the poses and other data
for ind, _ in enumerate(new_data):
    prediction = cur_model.predict(_)
    if np.any(prediction > 9e-2) and np.max(prediction) > 0.5:
            prediction = np.ndarray.tolist(prediction)
            pose_ind = prediction[-1].index(max(prediction[-1])) + 1
            pose_indices[pose_ind].append(ind)
            poses[pose_ind] += 1

poses, "Classified : ", sum([poses[_] for _ in poses]) / new_data.shape[0] * 100
```

```
Out[21]: ({1: 939, 2: 458, 3: 62, 4: 720, 5: 500, 6: 129, 7: 54, 8: 1},
          'Classified : ',
          87.92997542997543)
```

1.6 Verifying the estimated poses

Now that out data has been collected, we can randomly visualize some of the predicted poses to make sure they have been classified correctly.

```
In [22]: from mpl_toolkits.mplot3d import Axes3D
         def draw_person(person, pose_no, labels):
                 plt.clf()
                 fig = plt.figure()
                 ax = fig.gca(projection = '3d')
                 for _ in range(person.shape[0]):
                         ax.scatter(person[_, 0], person[_, 2],person[_, 1])
                         ax.text(person[_, 0], person[_, 2], person[_, 1], labels[_],\
                          size = 12, zorder = 1)
                 ax.set_title(pose_no)
                 ax.set_ylim(1, 3)
                 plt.show()

         def get_person(ind, dataset):
             return dataset[ind].reshape(25, 3)
```

```
In [23]: %matplotlib
         # visualize one pose for every type of pose
         for i in range(1, 2):
             ind = random.choice(pose_indices[i])
             person = get_person(ind, new_data)
             draw_person(person, i, labels)
```

```
Using matplotlib backend: TkAgg
```

As seen from the visualizations, the poses have been classified correctly and now we can further train the model to improve it's accuracy.

1.7 Adding second round of training

To add a second round of training, all we need to do is setup the corresponding *y vectors* for the correctly classified new poses, and run another `model.fit`, after which we can store the model for future use.

```
In [24]: # first creating the y vector
         newdata_y = []
         labeled_x = []
         for i in pose_indices:
             y_vec = np.zeros((8, 1))
             y_vec[i - 1, 0] = 1
             for x_vals in pose_indices[i]:
                 labeled_x.append(new_data[x_vals])
                 newdata_y.append(y_vec)
         new_train_labels = np.array(newdata_y).reshape(len(newdata_y), len(newdata_y[0]))
         new_train_X = np.array(labeled_x).reshape(len(labeled_x), new_data.shape[-1])
         new_train_labels.shape, new_train_X.shape

Out[24]: ((2863, 8), (2863, 75))

In [25]: training_size = int(0.6 * new_train_labels.shape[0])
         test_size = new_train_labels.shape[0] - training_size
         iterations = int(1e4)
         batch_size = int(0.05 * new_train_labels.shape[0])

In [26]: # permuting the data before training
         random_order = np.random.permutation(new_train_X.shape[0])
         new_train_X = new_train_X[random_order]
         new_train_labels = new_train_labels[random_order]

In [27]: cur_model.fit(new_train_X[:training_size], new_train_labels[:training_size], verbose=

Out[27]: <keras.callbacks.History at 0x7f3c4f6c3908>
```

1.7.1 Testing (Once Again!)

Now that the new model has been trained, let us shuffle the data and test the model. First we will test on the current large dataset and then test on the original X, y dataset.

```
In [28]: random_order = np.random.permutation(new_train_X.shape[0])
         new_train_X = new_train_X[random_order]
         new_train_labels = new_train_labels[random_order]
```

```
In [29]: test_data(cur_model, new_train_X, new_train_labels, training_size, test_size)
```

```
Out[29]: {'0test_types': ['loss', 'acc', 'binary_accuracy'],
          '1training_data': [0.022695497494558958,
           0.99803436225975539,
           0.99803436225975539],
          '2test_data': [0.032186949419580918,
           0.99705497382198949,
           0.99705497382198949],
          '3dataset': [0.026494730274943611, 0.99764233321690532, 0.99764233321690532]}
```

Testing it on the X, y original dataset, after perumutation we get:

```
In [30]: random_order = np.random.permutation(X.shape[0])
         X = X[random_order]
         new_y = new_y[random_order]
```

```
In [31]: training_size = int(0.6 * X.shape[0])
         test_size = X.shape[0] - training_size
         test_data(cur_model, X, new_y, training_size, test_size)
```

```
Out[31]: {'0test_types': ['loss', 'acc', 'binary_accuracy'],
          '1training_data': [1.6491459608078003, 0.88671875, 0.88671875],
          '2test_data': [1.3273274950962861, 0.90576923076923077, 0.90576923076923077],
          '3dataset': [1.5192192510484934, 0.89440993788819878, 0.89440993788819878]}
```

1.8 Saving information

Finally, all the relevant information can be saved for future use. This includes the trained model, the new X and y data along with the old one.

```
In [32]: # save the model with the weights, optimizer etc
         cur_model.save('trained_model.h5')
```

```
In [34]: X.shape, new_y.shape, new_train_X.shape, new_train_labels.shape
```

```
Out[34]: ((2863, 75), (2863, 8))
```

```
In [38]: # make the complete x dataset
         complete_dataset = pd.DataFrame(new_train_X, columns=old_dataset.columns).append(old_
         assert(complete_dataset.shape[0] == old_dataset.shape[0] + new_train_X.shape[0])
```

```
In [39]: # make the complete y dataset
         y_labels = pd.DataFrame(new_train_labels, columns=[_ for _ in range(1, 9)])
         new_y_labels = pd.DataFrame(new_y, columns = [_ for _ in range(1, 9)])
         complete_y = y_labels.append(new_y_labels)
         assert(complete_y.shape[0] == new_y.shape[0] + new_train_labels.shape[0])
```

```
In [40]: complete_y.shape[0], complete_dataset.shape[0]
```

```
Out[40]: (3024, 3024)
```

```
In [41]: complete_dataset[complete_y.columns] = complete_y
```

```
In [45]: complete_dataset.to_pickle('complete_dataset.pkl')
```