Not you?

Human Activity Recognition (HAR) Tutorial with Keras and Core ML (Part 1)

Nils Ackermann
Aug 9, 2018 · 13 min read

Keras and Apple's Core ML are a very powerful toolset if you want to quickly deploy a neural network on any iOS device. Most other tutorials focus on the popular MNIST data set for image recognition. We will go beyond this widely covered machine learning example. Instead, you will learn how to process time-sliced, multi-dimensional sensor data.

To be more specific, we will train a deep neural network (DNN) to recognize the type of movement (Walking, Running, Jogging, etc.) based on a given set of accelerometer data from a mobile device carried around a person's waist. We will use a WISDM data set for this tutorial (WISDM).

The approach presented in this article should work well for any other sensor data that you might come across within the Internet of Things (IOT). This article walks you through the following steps:

- Load accelerometer data from the WISDM data set
- Convert and reformat accelerometer data into a time-sliced representation
- Visualize the accelerometer data
- Reshape the multi-dimensional tabular data so that it is accepted by Keras
- Split up the data set into training, validation, and test set
- Define a deep neural network model in Keras which can later be processed by Apple's Core ML
- Train the deep neural network for human activity recognition data
- Validate the performance of the trained DNN against the test data using learning curve and confusion matrix
- Export the trained Keras DNN model for Core ML
- Ensure that the Core ML model was exported correctly by conducting a sample prediction in Python
- Create a playground in Xcode and import the already trained Keras model
- Use Apple's Core ML library in order to predict the outcomes for a given data set using Swift

Prerequisites in order to conduct all steps explained in this article (including the version number that the code was tested with):

- Python (version 3.6.5)
- Keras (version 2.1.6)
- TensorFlow (version 1.7.0)

- Coremltools (version 2.0)

Out of scope for this article: The creation of the perfect machine learning model with the highest possible performance for this type of problem statement is not the focus of this walkthrough.

You might wonder why Keras was chosen for this article over other frameworks, namely TensorFlow. There are two key reasons:

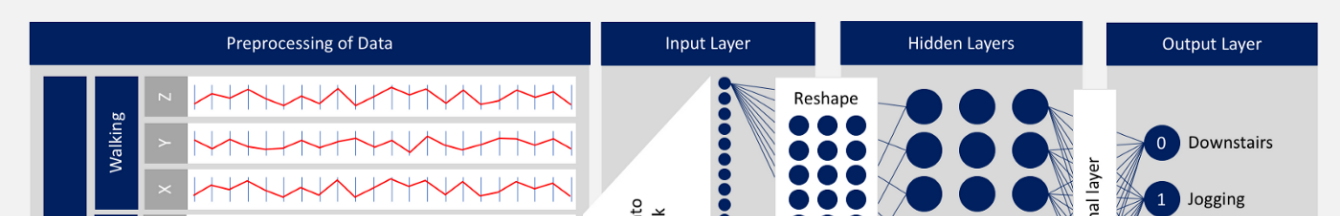
- Keras is very simple to learn and has a modern, more intuitive API than TensorFlow while still leveraging the capabilities of TensorFlow in the backend
- There are multiple TensorFlow APIs; while trying to use the more convenient estimator API (which is also recommended by the TensorFlow team — you can find more information here) I ran into compilation issues when converting the trained estimator to Core ML

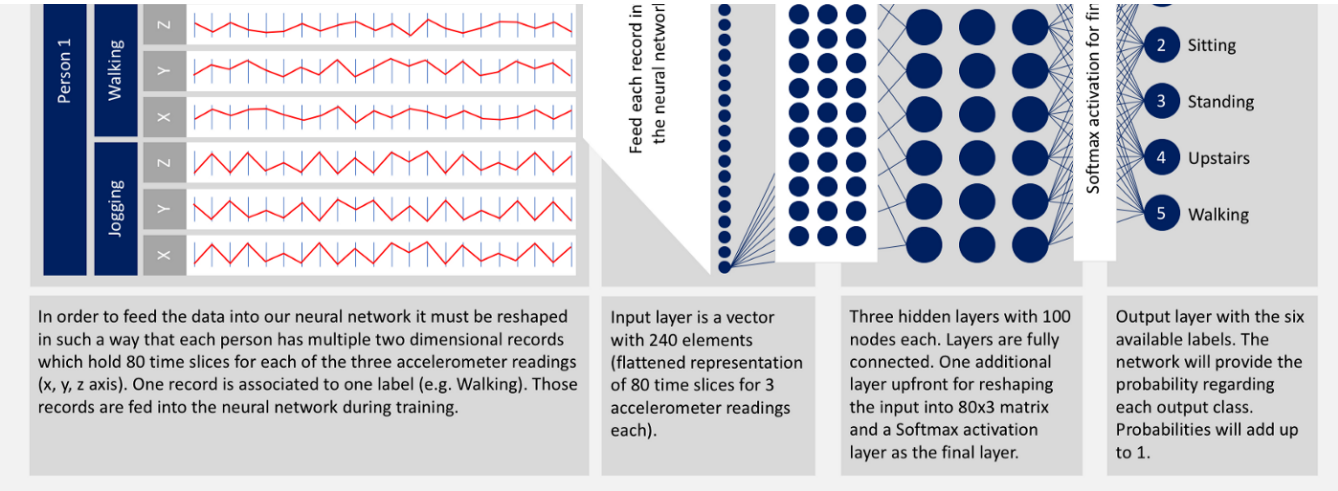
Conceptual Overview

Before we walk through the different steps in Python and Xcode, let's take a brief look at the problem statement and our solution approach. The data set that we are using is a collection of accelerometer data taken from a smartphone that various people carried with them while conducting six different exercises (Downstairs, Jogging, Sitting, Standing, Upstairs, Walking). For each exercise the acceleration for the x, y, and z axis was measured and captured with a timestamp and person ID.

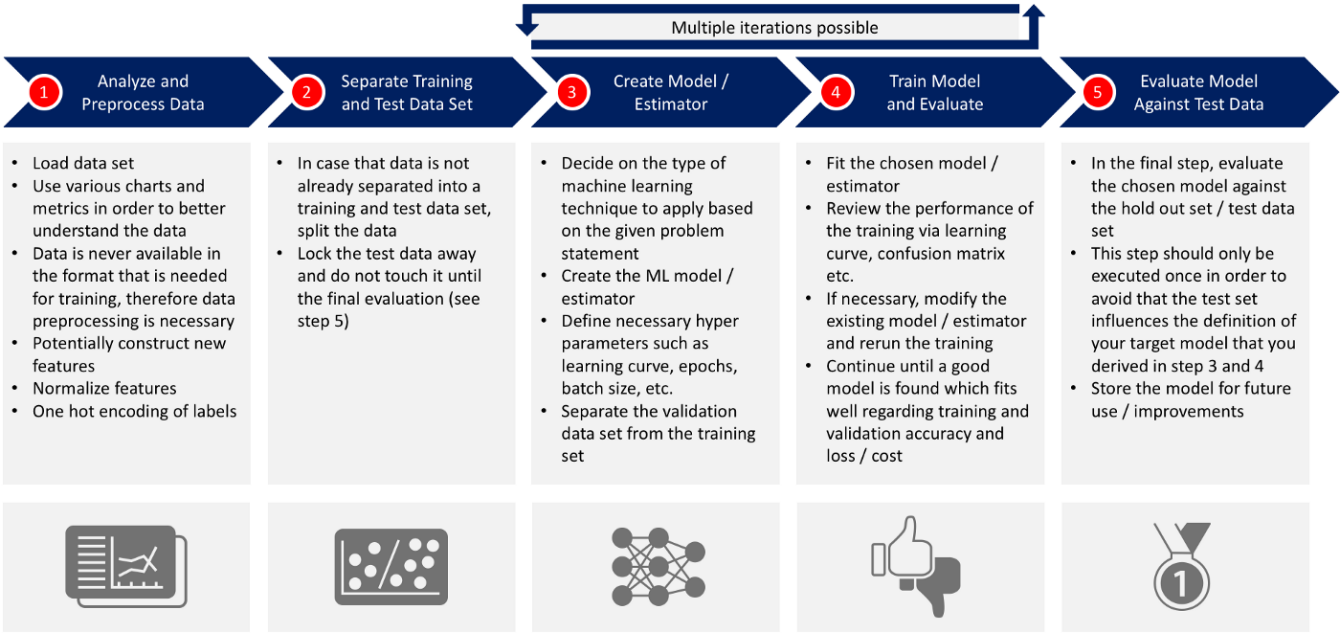
With this available data, we would like to train a neural network in order to understand if a person carrying a smartphone is performing any of the six activities. Once the neural network has been trained on the existing data, it should be able to correctly predict the type of activity a person is conducting when given previously unseen data.

The solution to this problem is a deep neural network. Based on the available data it will learn how to differentiate between each of the six activities. We can then show new data to the neural network and it will tell us what the user is doing at any particular point in time. The solution to this problem is depicted in the figure below.





The typical steps for solving a machine learning problem are depicted below. We will run through a very similar process throughout this article.



Import Libraries

First we need to import all necessary python libraries. If you are missing some of them, install them using the pip installer.

```
from __future__ import print_function
from matplotlib import pyplot as plt
%matplotlib inline
import numpy as np
import pandas as pd
import seaborn as sns
import coremltools
```

```
from scipy import stats
from IPython.display import display, HTML

from sklearn import metrics
from sklearn.metrics import classification_report
from sklearn import preprocessing

import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Reshape
from keras.layers import Conv2D, MaxPooling2D
from keras.utils import np_utils
```

20180808_har_1.py hosted with ❤ by **GitHub** **view raw**

After importing the libraries, let's set some standard parameters and print out the Keras version that we have installed. The WISDM dataset contains six different labels (Downstairs, Jogging, Sitting, Standing, Upstairs, Walking). Since we will use the list of labels multiple times, we create a constant for them (LABELS). The next constant TIME_PERIODS stores the length of the time segment. The constant STEP_DISTANCE determines the amount of overlap between two consecutive time segments.

```
# Set some standard parameters upfront
pd.options.display.float_format = '{:.1f}'.format
sns.set() # Default seaborn look and feel
plt.style.use('ggplot')
print('keras version ', keras.__version__)
# Same labels will be reused throughout the program
LABELS = ['Downstairs',
          'Jogging',
          'Sitting',
          'Standing',
          'Upstairs',
          'Walking']
# The number of steps within one time segment
TIME_PERIODS = 80
# The steps to take from one segment to the next; if this value is equal to
# TIME_PERIODS, then there is no overlap between the segments
STEP_DISTANCE = 40
```

20180808_har_2 hosted with ❤ by **GitHub** **view raw**

```
keras version  2.1.6
```

Load, Inspect and Transform the Accelerometer Data

Next, you need to download the dataset form here and store it locally. The important file is WISDM_ar_v1.1_raw.txt. Before doing the import, let's define a few convenience functions in order to read the data and show some basic information about the data.

```
def read_data(file_path):

    column_names = ['user-id',
                    'activity',
                    'timestamp',
                    'x-axis',
                    'y-axis',
                    'z-axis']
    df = pd.read_csv(file_path,
                     header=None,
                     names=column_names)
    # Last column has a ";" character which must be removed ...
    df['z-axis'].replace(regex=True,
      inplace=True,
      to_replace=r';',
      value=r'')
    # ... and then this column must be transformed to float explicitly
    df['z-axis'] = df['z-axis'].apply(convert_to_float)
    # This is very important otherwise the model will not fit and loss
    # will show up as NAN
    df.dropna(axis=0, how='any', inplace=True)

    return df

def convert_to_float(x):

    try:
        return np.float(x)
    except:
        return np.nan

def show_basic_dataframe_info(dataframe):

    # Shape and how many rows and columns
    print('Number of columns in the dataframe: %i' % (dataframe.shape[1]))
    print('Number of rows in the dataframe: %i\n' % (dataframe.shape[0]))

# Load data set containing all the data from csv
df = read_data('WISDM_ar_v1.1_raw.txt')
```

The data is loaded into the dataframe successfully. Now we can display the first 20 records of the dataframe and get some more insight regarding the distribution of the data.

```
# Describe the data
show_basic_dataframe_info(df)
df.head(20)
```

20180808_har_4.py hosted with ❤ by **GitHub** **view raw**

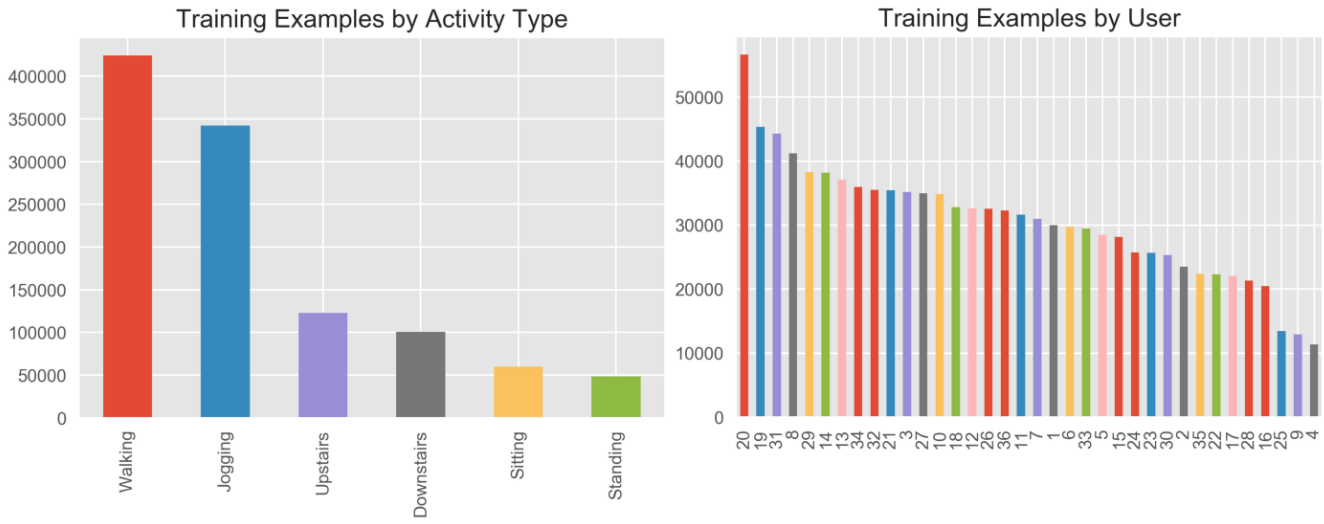
```
Number of columns in the dataframe: 6
Number of rows in the dataframe: 1098203
```

	user-id	activity	timestamp	x-axis	y-axis	z-axis
0	33	Jogging	491059623260000	-0.7	12.7	0.5
1	33	Jogging	491060622710000	5.0	11.3	1.0
2	33	Jogging	491061121670000	4.9	10.9	-0.1
3	33	Jogging	491062223050000	-0.6	18.5	3.0
4	33	Jogging	491063322900000	-1.2	12.1	7.2
5	33	Jogging	491064423060000	1.4	-2.5	-6.5
6	33	Jogging	491065423120000	-0.6	10.6	5.7
7	33	Jogging	491066523890000	-0.5	13.9	7.1
8	33	Jogging	491067623130000	-8.4	11.4	5.1
9	33	Jogging	491068722990000	1.0	1.4	1.6
10	33	Jogging	491069823150000	-8.2	19.6	2.7
11	33	Jogging	491070923300000	1.4	5.8	3.0
12	33	Jogging	491072023160000	-1.9	-3.0	-0.3
13	33	Jogging	491073123320000	6.1	6.9	8.2

13	33	Jogging	49107312332000	-6.1	6.9	-8.2
14	33	Jogging	49107422348000	5.8	18.0	8.5
15	33	Jogging	49107522293000	6.3	3.0	2.9
16	33	Jogging	49107632339000	-1.6	8.3	-1.5
17	33	Jogging	49107742355000	3.5	13.6	9.4
18	33	Jogging	49107852340000	-2.0	-5.7	-10.2
19	33	Jogging	49107962326000	2.8	10.3	-9.7

```
# Show how many training examples exist for each of the six activities
df['activity'].value_counts().plot(kind='bar',
                                   title='Training Examples by Activity Type')
plt.show()
# Better understand how the recordings are spread across the different
# users who participated in the study
df['user-id'].value_counts().plot(kind='bar',
                                  title='Training Examples by User')
plt.show()
```

20180808_har_5.py hosted with ❤ by **GitHub** **view raw**

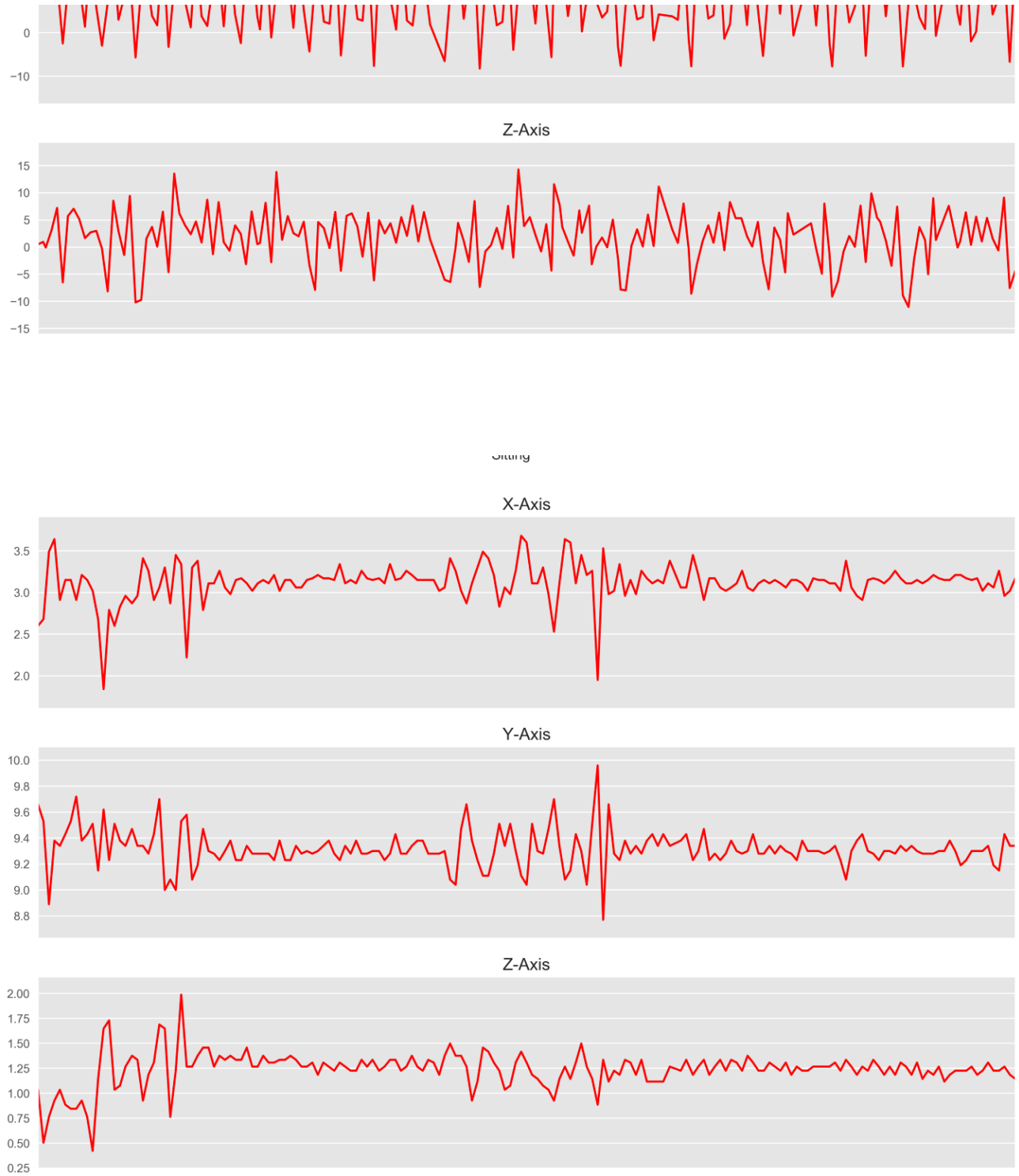


As we can see, we have more data for walking and jogging activities than we have for the other activities. Also we can see that 36 persons have participated in the experiment.

Next, let's take a look at the accelerometer data for each of the three axis for all six possible activities. The data is recorded at a sampling rate of 20 Hz (20 values per second). Since we show the first 180 records, each chart shows a 9 second interval for each of the six activities (calculation: 0.05 * 180 = 9 seconds). We will use two functions (which I have borrowed from here) to plot the data.

```python
def plot_activity(activity, data):

    fig, (ax0, ax1, ax2) = plt.subplots(nrows=3,
         figsize=(15, 10),
         sharex=True)
    plot_axis(ax0, data['timestamp'], data['x-axis'], 'X-Axis')
    plot_axis(ax1, data['timestamp'], data['y-axis'], 'Y-Axis')
    plot_axis(ax2, data['timestamp'], data['z-axis'], 'Z-Axis')
    plt.subplots_adjust(hspace=0.2)
    fig.suptitle(activity)
    plt.subplots_adjust(top=0.90)
    plt.show()

def plot_axis(ax, x, y, title):

    ax.plot(x, y, 'r')
    ax.set_title(title)
    ax.xaxis.set_visible(False)
    ax.set_ylim([min(y) - np.std(y), max(y) + np.std(y)])
    ax.set_xlim([min(x), max(x)])
    ax.grid(True)

for activity in np.unique(df['activity']):
    subset = df[df['activity'] == activity][:180]
    plot_activity(activity, subset)
```

20180808_har_5.py hosted with ❤ by **GitHub** **view raw**



As expected, there is a higher acceleration for activities such as jogging and walking compared to sitting. Before we continue, we will add one more column with the name "ActivityEncoded" to the dataframe with the encoded value for each activity: Downstairs, Jogging, Sitting, Standing, Upstairs, Walking

This is needed since the deep neural network cannot work with non-numerical labels. With the LabelEncoder, we are able to easily to convert back to the original label text.

```
# Define column name of the label vector
```

```
# Define column name of the label vector
LABEL = 'ActivityEncoded'
# Transform the labels from String to Integer via LabelEncoder
le = preprocessing.LabelEncoder()
# Add a new column to the existing DataFrame with the encoded values
df[LABEL] = le.fit_transform(df['activity'].values.ravel())
```

20180808_har_6.py hosted with ❤ by GitHub view raw

Split Data into Training and Test Set

It is important to separate the whole data set into a training set and a test set. Often times, you see mistakes on how the data is split. However you decide to split the data, you never want information from the test set to bleed into your training set. This might be great for the overall performance of your model during training and then validation against the test set. But your model is very unlikely to generalize well for data it has not seen yet.

The idea behind splitting: We want our neural network to learn from a few persons which have been through the experiment. Next, we then want to see how well our neural network predicts the movements of persons it has not seen before.

Data Splitting to Avoid

Only worrying about having at least a few example records per activity is not sufficient. You will run the risk that you have maybe three records of activity “Walking” for person 5 in the training set and one record for activity “Walking” for person 5 in the test set. Of course, with this type of situation, your model will perform great because it has already seen the movement pattern of person 5 during training. Always be critical about the performance of your DNN — it might be because of the wrong data split in the first place.

Better Splitting Approach

In our case, let’s split based on the user IDs. We will keep users with ID 1 to 28 for training the model and users with ID greater than 28 for the test set.

```
# Differentiate between test set and training set
df_test = df[df['user-id'] > 28]
df_train = df[df['user-id'] <= 28]
```

20180808_har_7.py hosted with ❤ by GitHub view raw

Normalize Training Data

Next, we need to normalize our features within our training data. Of course there are various ways on how to normalize. Please keep in mind that you use the same normalization algorithm later when feeding new data into your neural network. Otherwise your preditions will be off. On top of the normalization we will also apply rounding to the three features.

```
# Normalize features for training data set (values between 0 and 1)
# Surpress warning for next 3 operation
pd.options.mode.chained_assignment = None  # default='warn'
df_train['x-axis'] = df_train['x-axis'] / df_train['x-axis'].max()
df_train['y-axis'] = df_train['y-axis'] / df_train['y-axis'].max()
df_train['z-axis'] = df_train['z-axis'] / df_train['z-axis'].max()
# Round numbers
df_train = df_train.round({'x-axis': 4, 'y-axis': 4, 'z-axis': 4})
```

20180808_har_8.py hosted with ❤ by **GitHub** **view raw**

Reshape Data into Segments and Prepare for Keras

The data contained in the dataframe is not ready yet to be fed into a neural network. Therefore we need to reshape it. Let's create another function for this called "create_segments_and_labels". This function will take in the dataframe and the label names (the constant that we have defined at the beginning) as well as the length of each record. In our case, let's go with 80 steps (see constant defined earlier). Taking into consideration the 20 Hz sampling rate, this equals to 4 second time intervals (calculation: 0.05 * 80 = 4). Besides reshaping the data, the function will also separate the features (x-acceleration, y-acceleration, z-acceleration) and the labels (associated activity).

```
def create_segments_and_labels(df, time_steps, step, label_name):

    # x, y, z acceleration as features
    N_FEATURES = 3
    # Number of steps to advance in each iteration (for me, it should always
    # be equal to the time_steps in order to have no overlap between segments)
    # step = time_steps
    segments = []
    labels = []
    for i in range(0, len(df) - time_steps, step):
        xs = df['x-axis'].values[i: i + time_steps]
        ys = df['y-axis'].values[i: i + time_steps]
```

```
        zs = df['z-axis'].values[i: i + time_steps]
        # Retrieve the most often used label in this segment
        label = stats.mode(df[label_name][i: i + time_steps])[0][0]
        segments.append([xs, ys, zs])
        labels.append(label)

    # Bring the segments into a better shape
    reshaped_segments = np.asarray(segments, dtype= np.float32).reshape(-1, time_steps,
    labels = np.asarray(labels)

    return reshaped_segments, labels

x_train, y_train = create_segments_and_labels(df_train,
                                              TIME_PERIODS,
                                              STEP_DISTANCE,
                                              LABEL)
```

20180808_har_9.py hosted with ❤ by **GitHub** **view raw**

By now, you should have both 20.868 records in x_train as well as in y_train. Each of the 20.868 records in x_train is a two dimensional matrix of the shape 80x3.

```
print('x_train shape: ', x_train.shape)
print(x_train.shape[0], 'training samples')
print('y_train shape: ', y_train.shape)
```

20180808_har_10.py hosted with ❤ by **GitHub** **view raw**

```
x_train shape:  (20868, 80, 3)
20868 training samples
y_train shape:  (20868,)
```

For constructing our deep neural network, we should now store the following dimensions:

- Number of time periods: This is the number of time periods within one record (since we wanted to have a 4 second time interval, this number is 80 in our case)
- Number of sensors: This is 3 since we only use the acceleration over the x, y, and z axis
- Number of classes: This is the amount of nodes for our output layer in the neural network. Since we want our neural network to predict the type of activity, we will

take the number of classes from the encoder that we have used earlier.

```
# Set input & output dimensions
num_time_periods, num_sensors = x_train.shape[1], x_train.shape[2]
num_classes = le.classes_.size
print(list(le.classes_))
```

20180808_har_11.py hosted with ❤ by GitHub view raw

```
['Downstairs', 'Jogging', 'Sitting', 'Standing', 'Upstairs',
'Walking']
```

The data that we would like to feed into our network is two dimensional (80x3). Unfortunately, Keras and Core ML in conjunction are not able to process multi-dimensional input data. Therefore we need to "flatten" the data for our input layer into the neural network. Instead of feeding a matrix of shape 80x3 we will feed in a list of 240 values.

```
input_shape = (num_time_periods*num_sensors)
x_train = x_train.reshape(x_train.shape[0], input_shape)
print('x_train shape:', x_train.shape)
print('input_shape:', input_shape)
```

20180808_har_12.py hosted with ❤ by GitHub view raw

```
x_train shape: (20868, 240)
input_shape: 240
```

Before continuing, we need to convert all feature data (x_train) and label data (y_train) into a datatype accepted by Keras.

```
x_train = x_train.astype('float32')
y_train = y_train.astype('float32')
```

20180808_har_13.py hosted with ❤ by GitHub view raw

We are almost done with the preparation of our data. One last step we need to do is to conduct one-hot-encoding of our labels. Please only execute this line once!

```
y_train_hot = np_utils.to_categorical(y_train, num_classes)
print('New y_train shape: ', y_train_hot.shape)
```

20180808_har_14.py hosted with ❤ by GitHub view raw

```
New y_train shape:  (20868, 6)
```

Create Deep Neural Network Model in Keras

By now, you have completed all the heavy-lifting on your side. The data is ready in such a format that Keras will be able to process it. I have decided to create a neural network with 3 hidden layers of 100 fully connected nodes each (feel free to play around with the shape of the network or even switch to more complex ones like a convolutional neural network).

Important remark: as you remember, we have reshaped our input data from a 80x3 matrix into a vector of length 240 so that Apple's Core ML can later process our data. In order to reverse this, our first layer in the neural network will reshape the data into the "old" format. The last two layers will again flatten the data and then run a softmax activation function in order to calculate the probability for each class. Remember, that we are working with six classes in our case (Downstairs, Jogging, Sitting, Standing, Upstairs, Walking).

```
model_m = Sequential()
# Remark: since coreml cannot accept vector shapes of complex shape like
# [80,3] this workaround is used in order to reshape the vector internally
# prior feeding it into the network
model_m.add(Reshape((TIME_PERIODS, 3), input_shape=(input_shape,)))
model_m.add(Dense(100, activation='relu'))
model_m.add(Dense(100, activation='relu'))
model_m.add(Dense(100, activation='relu'))
model_m.add(Flatten())
model_m.add(Dense(num_classes, activation='softmax'))
print(model_m.summary())
```

20180808_har_15.py hosted with ❤ by GitHub view raw

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
reshape_2 (Reshape)          (None, 80, 3)             0
```

```
_________________________________________________________________
dense_5 (Dense)              (None, 80, 100)           400
_________________________________________________________________
dense_6 (Dense)              (None, 80, 100)           10100
_________________________________________________________________
dense_7 (Dense)              (None, 80, 100)           10100
_________________________________________________________________
flatten_2 (Flatten)          (None, 8000)              0
_________________________________________________________________
dense_8 (Dense)              (None, 6)                 48006
=================================================================
Total params: 68,606
Trainable params: 68,606
Non-trainable params: 0
_________________________________________________________________
None
```

Fit the DNN Model in Keras

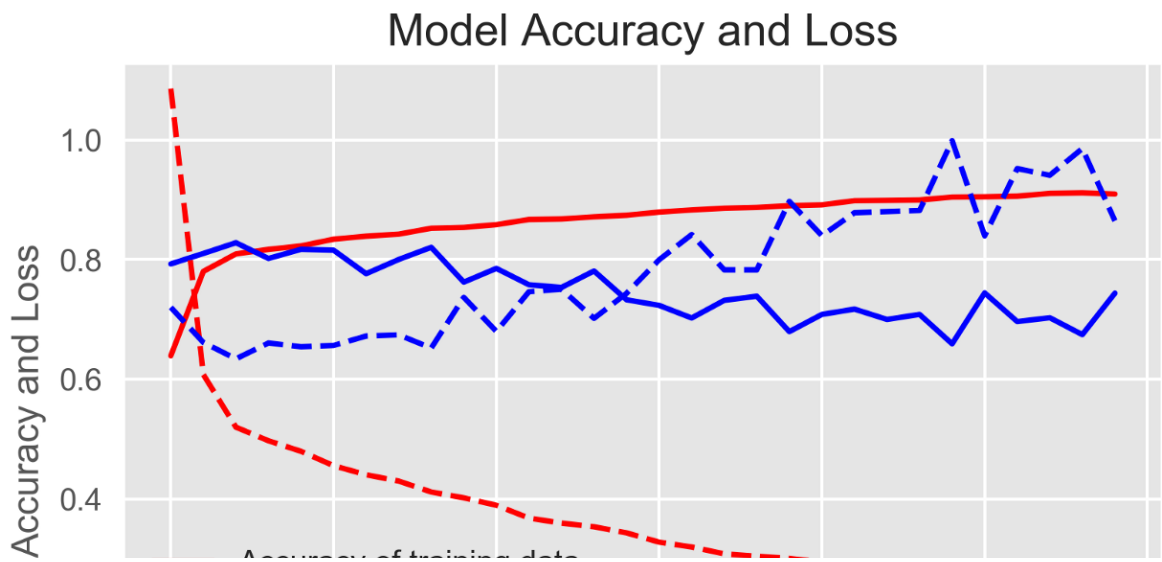
Next, we will train the model with our training data that we have prepared earlier. We will define an early stopping callback monitor on training accuracy: if the training fails to improve for two consecutive epochs, then the training will stop with the best model. The hyperparameter used for the training are quite simple: We will use a batch size of 400 records and will train the model for 50 epochs. For model training, we will use a 80:20 split to separate training data and validation data. It is that simple. So let's go ahead and train our model. There are some good explanations out there on the different hyperparameters, for instance here.

```
callbacks_list = [
    keras.callbacks.ModelCheckpoint(
        filepath='best_model.{epoch:02d}-{val_loss:.2f}.h5',
        monitor='val_loss', save_best_only=True),
    keras.callbacks.EarlyStopping(monitor='acc', patience=1)
]

model_m.compile(loss='categorical_crossentropy',
                optimizer='adam', metrics=['accuracy'])

# Hyper-parameters
BATCH_SIZE = 400
EPOCHS = 50

# Enable validation to use ModelCheckpoint and EarlyStopping callbacks.
history = model_m.fit(x_train,
                      y_train_hot,
                      batch_size=BATCH_SIZE,
                      epochs=EPOCHS,
```

```
                    callbacks=callbacks_list,
                    validation_split=0.2,
                    verbose=1)
```


The performance of this simple DNN is OK. We have validation accuracy of approximately 74%. This could definitely improved, maybe with further hyperparameter tuning and especially with a modified neural network design. Before we go on with the test validation, we will print the learning curve for both the training and validation data set.

```
plt.figure(figsize=(6, 4))
plt.plot(history.history['acc'], 'r', label='Accuracy of training data')
plt.plot(history.history['val_acc'], 'b', label='Accuracy of validation data')
plt.plot(history.history['loss'], 'r--', label='Loss of training data')
plt.plot(history.history['val_loss'], 'b--', label='Loss of validation data')
plt.title('Model Accuracy and Loss')
plt.ylabel('Accuracy and Loss')
plt.xlabel('Training Epoch')
plt.ylim(0)
plt.legend()
plt.show()

# Print confusion matrix for training data
y_pred_train = model_m.predict(x_train)
# Take the class with the highest probability from the train predictions
max_y_pred_train = np.argmax(y_pred_train, axis=1)
print(classification_report(y_train, max_y_pred_train))
```


```
precision    recall  f1-score   support

        0.0       0.70      0.42      0.53      1864
        1.0       0.98      0.98      0.98      6567
        2.0       0.99      0.99      0.99      1050
        3.0       0.99      0.99      0.99       833
        4.0       0.66      0.63      0.64      2342
        5.0       0.85      0.93      0.89      8212

avg / total       0.87      0.87      0.87     20868
```

Check Against Test Data

Let's continue with this model and see how it performs against the test data that we have held back earlier. In our case, we will check the performance against the movements of the six users that the model has not yet seen.

```
6584/6584 [==============================] - 1s 128us/step

Accuracy on test data: 0.76

Loss on test data: 1.39
```

The accuracy on the test data is 76%. This means that our model generalizes well for persons it has not yet seen. Let's see where our model wrongly predicted the labels.

```
def show_confusion_matrix(validations, predictions):

    matrix = metrics.confusion_matrix(validations, predictions)
    plt.figure(figsize=(6, 4))
    sns.heatmap(matrix,
                cmap='coolwarm',
                linecolor='white',
                linewidths=1,
                xticklabels=LABELS,
                yticklabels=LABELS,
```

```
                    annot=True,
                    fmt='d')
    plt.title('Confusion Matrix')
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.show()

y_pred_test = model_m.predict(x_test)
# Take the class with the highest probability from the test predictions
max_y_pred_test = np.argmax(y_pred_test, axis=1)
max_y_test = np.argmax(y_test, axis=1)

show_confusion_matrix(max_y_test, max_y_pred_test)

print(classification_report(max_y_test, max_y_pred_test))
```

20180808_har_18.py hosted with ❤ by **GitHub** **view raw**

	precision	recall	f1-score	support
0	0.61	0.27	0.37	650
1	0.80	0.95	0.87	1990
2	0.82	0.99	0.90	452
3	0.91	0.74	0.81	370
4	0.48	0.42	0.45	725

```
          5       0.77      0.79      0.78      2397
avg / total       0.74      0.76      0.74      6584
```

As you can see, the precision of the model is good for predicting jogging (1), sitting (2), standing (3), and walking (5). The model has problems for clearly identifying upstairs and downstairs activities.

There is of course still great potential for improving the model, e.g. by using more advanced neural network designs like convolutional neural networks (CNN) or Long Short Term Memory (LSTM). I might explore this in a later article. For our purpose of showing the end to end process, the result is good enough.

Freeze Keras Model for Core ML

If you are happy with the model and its performance, you should convert it now to be used with Core ML. The convert function only takes in a few arguments:

- The reference to your Keras model
- The name that you want to give the input data; in our case we are feeding acceleration data into the network
- The name that you want to assign to the outputs
- The “human-readable” label names; you can again use our LABEL constant that we defined in the beginning

```
coreml_model = coremltools.converters.keras.convert(model_m,
                                                    input_names=['acceleration'],
                                                    output_names=['output'],
                                                    class_labels=LABELS)

print(coreml_model)
coreml_model.author = 'Nils Ackermann'
coreml_model.license = 'N/A'
coreml_model.short_description = 'Activity based recognition based on WISDM dataset'
coreml_model.output_description['output'] = 'Probability of each activity'
coreml_model.output_description['classLabel'] = 'Labels of activity'

coreml_model.save('HARClassifier.mlmodel')
```

```
input {
  name: "acceleration"
  type {
    multiArrayType {
      shape: 240
      dataType: DOUBLE
    }
  }
}
output {
  name: "output"
  type {
    dictionaryType {
      stringKeyType {
      }
    }
  }
}
output {
  name: "classLabel"
  type {
    stringType {
    }
  }
}
predictedFeatureName: "classLabel"
predictedProbabilitiesName: "output"
```

Compare Keras Prediction to Core ML Prediction

Before using your Core ML model, let's make sure that the export was successful and that both our Keras model as well as the Core ML model provide the same prediction when given a random data set.

```
print('\nPrediction from Keras:')
test_record = x_test[1].reshape(1,input_shape)
keras_prediction = np.argmax(model_m.predict(test_record), axis=1)
print(le.inverse_transform(keras_prediction)[0])
print('\nPrediction from Coreml:')
coreml_prediction = coreml_model.predict({'acceleration': test_record.reshape(input_shap
print(coreml_prediction["classLabel"])
```

20180808_har_20.py hosted with ❤ by **GitHub** **view raw**

```
Prediction from Keras:
Jogging
```

```
Prediction from Coreml:
Jogging
```

Great news! For the record with index 1, both Keras and Core ML predict the same label which is Jogging. We are now ready to use our Core ML model on any iOS device.

Summary and What's Next

In this article you have learned how to load and transform complex accelerometer data and run it through a deep neural network in Keras. You exported the trained model into a Core ML file. In my next article, I will walk you through the steps necessary in order to use this trained Core ML model within a simple Swift program. You can then use this knowledge to deploy the DNN to any iOS device.

The Jupyter notebook for this article is available on github.

Links and References

- Official Anaconda website
- Official Keras website
- Official TensorFlow website
- Official Apple Core ML documentation
- Official Apple coremltools github repository
- Good overview to decide which framework is for you: TensorFlow or Keras
- Good article by Aaqib Saeed on convolutional neural networks (CNN) for human activity recognition (also using the WISDM dataset)
- Another article also using the WISDM dataset implemented with TensorFlow and a more sophisticated LSTM model written by Venelin Valkov

Disclaimer

The postings on this site are my own and do not necessarily represent the postings, strategies or opinions of my employer.