Human Activity Recognition Using Deep Recurrent Neural Nets, LSTM and Tensorflow on Smartphones

Author: Girish Parameswarappa, Project Advisor: Dr. Julia Hua Fang

Department: Computer and Information Science

Abstract: In today's world with the increase widespread of mobile phones packed with respective onboard sensors such as GPS, gyroscope, accelerometer, etc., and Internet connectivity. It has become easier to track smartphone user activities and they can even start logging their activities to track their physical activities such as walking, jogging, climbing stairs, number of steps taken, calories burned, etc. In this context, we are discussing an activity recognition using supervised learning from a dataset which contains recordings of accelerometer data of different users with activity type. Results are obtained using Deep Learning Recurrent Neural Nets & LSTM using Tensorflow API to build the model and export the model to an android application for real-time predictions.

Introduction: Human activity recognition (**HAR**) aims to learn and find out the actions carried out by different users given a set of observations and in a controlled environment. Many applications have adapted motion sensors in different parts of the body such as wrist, waist, chest and thighs to achieve good performance in classifying activities. These sensors would not provide an efficient and long-term solution for activity recognition and monitoring.

On the other hand, handheld devices such as smartphones are bringing new research opportunities with computer-vision, human centered applications where the user is the main source of data from where the information can be retrieved and the phone is the user's sensing tool. We are using smartphone's triaxial accelerometer that measures acceleration in all three physical dimensions (x-axis, y-axis, z-axis), however accelerometers can detect device orientation too. In this project, we aim to train LSTM (Long-short-term-memory) Recurrent Neural network from the accelerometer data and build the model using Tensorflow library and later export the model to an Android application to showcase how well the model performs with real-life data.

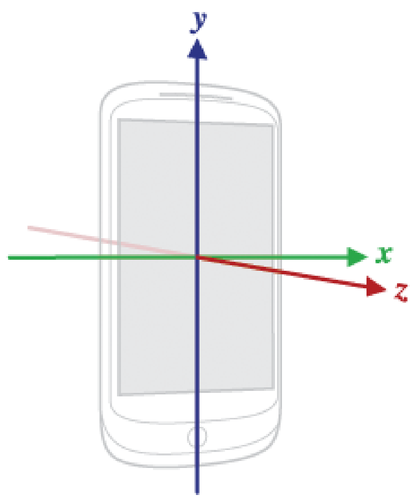


Fig 1 – Smartphone accelerometer Sensor

Data collection: For this project, the data is being extracted from Wireless Sensor Data Mining Lab (WISDM) site which contains a raw text file with 6 other attributes and around 1,098,207 samples of different user activities with column names as UserId, Activity, Timestamp, x-axis, y-axis and z-axis. This dataset was collected in a controlled environment where there were different users based on the user id and respective activities.

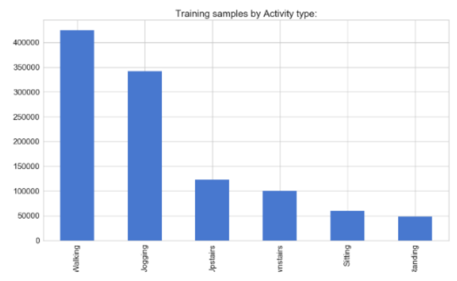


Fig 2 – Training samples by Activity type

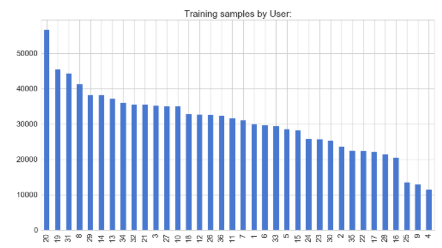


Fig 2.1 – Training samples by Users

For the data pre-processing, we used fixed-length sequences as a training data because the LSTM model expects to have it in a same format to maximize the probability results. So, we reshaped the data into the format that the LSTM model expects, using 200 training samples for respective generated sequences and for each generated sequence we added the corresponding x, y & z axes as number of features (x, y, z) and applied validation rules by splitting the shaped data into training (80%) and testing (20%) sets.

Building the model: We used Recurrent Neural Nets (RNN) and Long-Short-Term Memory units (LSTM) for building the model.

Recurrent networks are a type of artificial neural network which supports machine learning. The word “recurrent” means occurring repeatedly, i.e. RNN allows the data to flow in both directions within the network units and forms a directed cycle. This allows RNN to perform temporal modeling. A basic RNN looks like a network of nodes (neurons) with feedback loops. Each node has a set of real valued activation functions and each connection with modifiable weight. Nodes could be an input data receiving data from outside the network, output data or hidden nodes that compute the data while passing from input node to output node. Additionally, RNN can use their internal memory for processing arbitrary input sequences and all the information usually circulates within the hidden states.

A process to carry memory forward mathematically:

$$\mathbf{h}_t = \phi\left(W\mathbf{x}_t + U\mathbf{h}_{t-1}\right),$$

LSTM or long short-term memory is an RNN architecture which holds all the values over certain arbitrary intervals. LSTM is very well suited to learn from various experiences such as classifying, providing predictions given unknown size and formed between important events. LSTM units have memory cells which can remember values for long or short periods of time. LSTM is a type of RNN that uses special units with the addition of standard units. LSTM uses a set of gates to control the flow of information into the memory, its output and when it should forget the information.

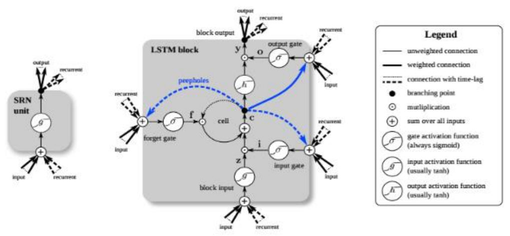


Figure 1. Detailed schematic of the Simple Recurrent Network (SRN) unit (left) and a Long Short-Term Memory block (right) as used in the hidden layers of a recurrent neural network.

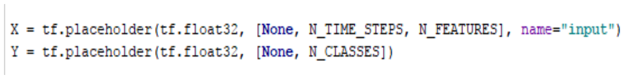
Fig 3 – A simple RNN & LSTM block

Step 1:

For this project, we are trying to build a deep network by using 2 fully connected RNN with 2 LSTM layers stacked on top of each other with 64 hidden units altogether. The reason we are stacking multiple layers of LSTM is because this allows for great complexity in our model and we can describe more complex patterns to recognize at each layer. When the first input is fed into the first layer, the output of the first layer will be the next input to the second layer and hence the computation will be more complex and the efficiency of the test accuracy will increase. In a single layer RNN we have only 1 hidden state which does all the work and it fails to capture all the structures of sequences. Using multi-layered RNN such structures of sequences are captured and it will result in better performance. For instance, the 1st layer might learn some of the activities as standing or walking and the 2nd layer would build on this to learn other activities such as walking upstairs or downstairs.

Step2: Tensorflow Placeholders

We create placeholders for our model using Tensorflow API library. A placeholder is simply a variable which will assign the data at a later point. This API will allow us to build operations and computation graphs without need of data and then feed the data into the graphs using these placeholders.



```
X = tf.placeholder(tf.float32, [None, N_TIME_STEPS, N_FEATURES], name="input")
Y = tf.placeholder(tf.float32, [None, N_CLASSES])
```

Step3: Loss function & Optimizer.

For calculating the loss, we use the L2-norm loss function which is also known as Least Squared Error (LSE). This function minimizes the difference between the target value (**Y**$_i$) and the estimated value (**f(x**$_i$**)**) by taking the sum of absolute differences between them.

L2-norm function is defined as follows:

$$S = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

The difference between L1-norm and L2-norm as follows:

L1 loss function (Least abs deviations) - LAD	**L2 Loss function (Least squared error) – LSE**
Unstable solution	Stable solutions
Robust	Not Robust
Multiple solutions	Single solution

For optimizing the model, we use Adam Optimization algorithm as a better choice for our deep learning model. This algorithm is an extension to stochastic gradient descent where it maintains a single learning rate for all the weights throughout the training period and only the learning rate is maintained for each network weight. This algorithm calculates moving average with more weights being fed to the recent data. This is usually called exponential moving average. This type reacts faster to recent changes than a simple moving average. So, taking both averages of the gradients and second moments of the gradients are used.

Adam’s algorithm parameters:

$$m_w^{(t+1)} \leftarrow \beta_1 m_w^{(t)} + (1-\beta_1)\nabla_w L^{(t)}$$
$$v_w^{(t+1)} \leftarrow \beta_2 v_w^{(t)} + (1-\beta_2)(\nabla_w L^{(t)})^2$$

$$\hat{m}_w = \frac{m_w^{(t+1)}}{1-\beta_1^t}$$

$$\hat{v}_w = \frac{v_w^{(t+1)}}{1-\beta_2^t}$$

$$w^{(t+1)} \leftarrow w^{(t)} - \eta \frac{\hat{m}_w}{\sqrt{\hat{v}_w} + \epsilon}$$

Step 4: Training the Model

For training the model we again use Tensorflow API with 50 Epochs and set the batch size as 1024 (32 x 32) because sending this number of samples will be suitable for this model. Using a larger batch size might overfit the data. While training the model to know the value of **Y,** we must have knowledge about **X.** As the model learns about the input it defines all these values in the **feed_dict** argument and **history** is where all the training, testing accuracies and loss are stored.

Step 5: Experimental Results:

After the model has been trained for over 50 epochs which took more than 6 hrs. to train the model as per the computation power of the system. We got the test accuracy above 97% with loss of over 0.2%. Below is a graph which shows the iterations of training, testing over Epochs and a confusion matrix to show how the model predicted v/s true predictions.

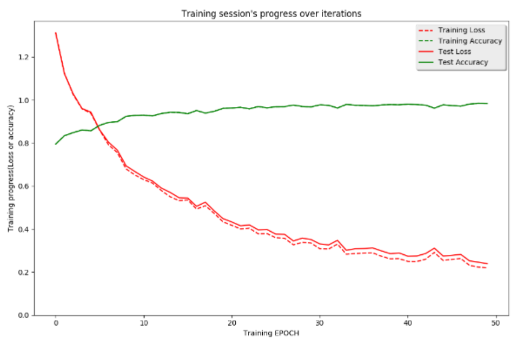


Fig 4 – Training session's progress

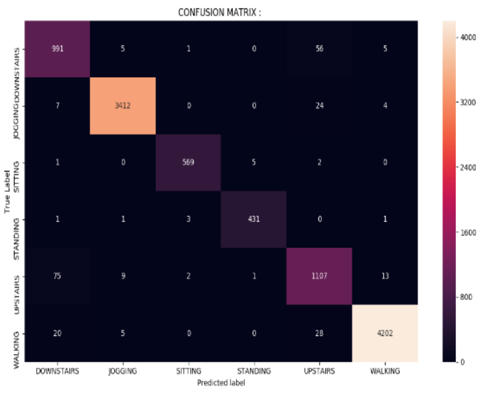
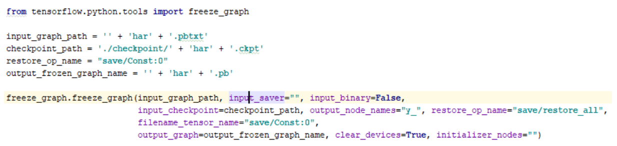


Fig 4.1 – Confusion matrix

Step 6: Exporting the model to An Android Application:

After the model has been trained, we have used Python packages such as **pickle** to store all the learned parameters such as history, predictions and checkpoint folder with all the model metadata.

Later we must freeze the graph using the Tensorflow API to save the graph and all the weights so that the Tensorflow for Android can understand it by merging all the files into a single protobuf file. Below is the small snippet for this function.

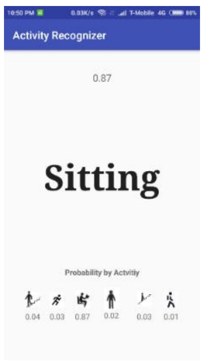


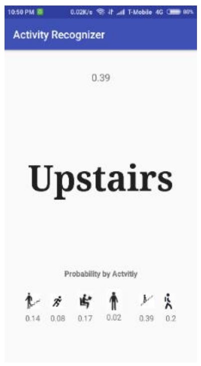
```
from tensorflow.python.tools import freeze_graph

input_graph_path = '' + 'har' + '.pbtxt'
checkpoint_path = './checkpoint/' + 'har' + '.ckpt'
restore_op_name = "save/Const:0"
output_frozen_graph_name = '' + 'har' + '.pb'

freeze_graph.freeze_graph(input_graph_path, input_saver="", input_binary=False,
                          input_checkpoint=checkpoint_path, output_node_names="y_", restore_op_name="save/restore_all",
                          filename_tensor_name="save/Const:0",
                          output_graph=output_frozen_graph_name, clear_devices=True, initializer_nodes="")
```

For testing the model on real-time data, we use Android studio IDE to build a test application which uses our pre-trained model stored in the main assets folder, with all the Tensorflow inference interface libraries, and uses placeholders from the model to connect with the application for predictions from the smartphone accelerometer sensor.

- **Results:**

  A demo application which shows the activities using a live smartphone when worn in your waist pocket can start predicting the basic activities as it starts to learn about recent changes in the accelerometer sensor and speaks it to the user using text-to-speech API in a time frame of 3 seconds. Below are some of the results shown:





Implementation: All the model building was done using python 3.5 and using many python packages such as sklearn, scipy, pandas, numpy, tensorflow, matplotlib, seaborn and pickle. After the model was built, all the graphs where saved using *plt.savefig()*. Later the model was exported to an android application using Java, xml and Android Studio IDE to create an application and start live predictions. Additionally, a text-to-speech API was used for speaking out the results from the phone.

Conclusion: We tested a machine learning algorithm using deep learning and implemented an Recurrent Neural Network (RNN) with LSTM memory units and Tensorflow API to build this project. This produced an overall test accuracy of 97% with 200 step sequences and when exported to an android application, it is much more efficient when used more often.

Future Work: This project could be used in various fields such as tracking Human Activity Recognition (HAR), health-care monitoring, old-age homes, etc. With more data coming in from the user’s activities it can be used as a recommender system for predicting various activities. With the use of more sensors, or a dataset with more sensor feature samples, supervised learning could make it more reliable and require less learning time for an application.

References:

[1]. Dataset - http://www.cis.fordham.edu/wisdm/
[2]. https: //en.wikipedia.org/wiki/Recurrent_neural_network#Long_short-term_memory
[3]. https://medium.com/@curiousily/human-activity-recognition-using-lstms-on-android-tensorflow-for-hackers-part-vi-492da5adef64
[4]. https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2013-84.pdf
[5]. http://cs229.stanford.edu/proj2015/100_report.pdf
[6]. https://deeplearning4j.org/lstm.html
[7]. https://learningtensorflow.com/lesson4/
[8]. https://aqibsaeed.github.io/2017-05-02-deploying-tensorflow-model-andorid-device-human-activity-recognition/
[9]. http://www.chioka.in/differences-between-l1-and-l2-as-loss-function-and-regularization/
[10]. https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/
[11]. https://en.wikipedia.org/wiki/Stochastic_gradient_descent#Adam
[12]. https://stats.stackexchange.com/questions/184448/difference-between-gradientdescentoptimizer-and-adamoptimizer-tensorflow.
[13]. https://github.com/fchollet/keras/issues/4149
[14]. https://www.tensorflow.org/
[15]. https://www.python.org/
[16]. http://scikit-learn.org/stable/modules/neural_networks_supervised.html
[17]. https://arxiv.org/pdf/1206.5533.pdf
[18]. https://deeplearning4j.org/recurrentnetwork
[19]. https://deeplearning4j.org/updater
[20]. http://www.androidauthority.com/android-studio-tutorial-beginners-637572/