6.835 Project 2:
Gesture Recognition using the Kinect

Due: 5:00 PM Monday, March 4, 2019

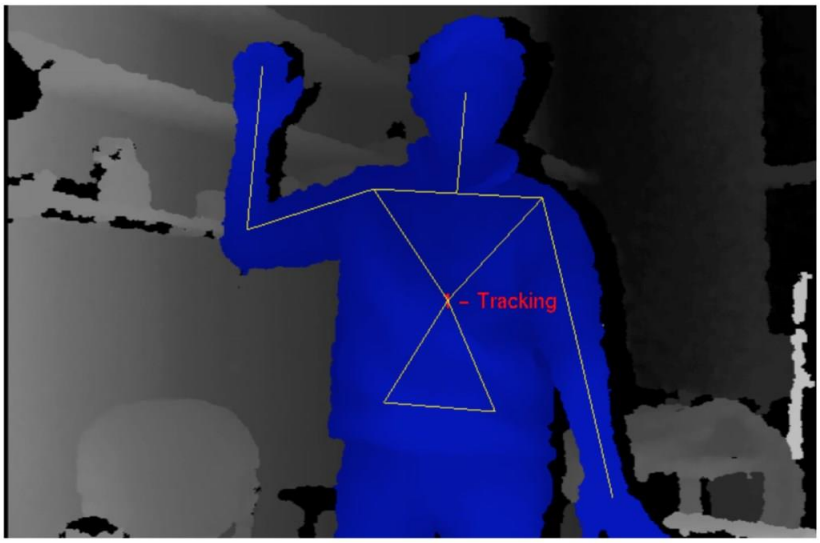

Figure 1: An example of the depth map from the Kinect camera and the corresponding skeletal body-pose estimation.

1 Introduction

The goal of this project is to explore the task of upper body gesture recognition. You will implement and compare several classification algorithms on a real 3D gesture dataset. Please start early and ask questions!

Figure 2: The kinect camera used to record the gesture data.

The dataset you will use in this project was recorded in front of a Microsoft Kinect camera (shown above) and consists of upper body gestures involving both arms. These nine gestures, listed below, are similar to the multi-touch gestures on devices like the iPhone or iPad, and could be used for tasks like map or picture manipulation.

Gesture	Description
0. pan left	right hand moving left
1. pan right	right hand moving right
2. pan up	right hand moving up
3. pan down	right hand moving down
4. zoom in	two hands moving outward
5. zoom out	two hands moving inward
6. rotate clockwise	two hands rotating clockwise
7. rotate counterclockwise	two hands rotating counterclockwise
8. point	right hand pointing

From each frame recorded by the Kinect camera, we have already estimated a skeletal body model consisting of 11 joint positions. The final data that you will use for this project consists of 33 pose features for each frame, encoding the 3D coordinates (x,y,z) of each joint. In Python notation (inclusive of the first index, exclusive of the last index):

Pose Features	Joint Data
frame[0:3]	head position (x,y,z)
frame[3:6]	neck position (x,y,z)
frame[6:9]	left shoulder position (x,y,z)
frame[9:12]	left elbow position (x,y,z)
frame[12:15]	left hand position (x,y,z)
frame[15:18]	right shoulder position (x,y,z)
frame[18:21]	right elbow position (x,y,z)
frame[21:24]	right hand position (x,y,z)
frame[24:27]	torso position (x,y,z)
frame[27:30]	left hip position (x,y,z)
frame[30:33]	right hip position (x,y,z)

For simplicity, we have already segmented the boundaries of each gesture, so the task in this project is the *labeling* of each sequence into one of known gesture classes, i.e., you don't have to solve the gesture *segmentation* and the gesture *spotting* problems.

2 Getting Started

Download the packaged .zip file from Stellar. After extracting the files, open them in your favorite text editor.

Similar to the setup in mini-project 1, use the command prompt to create a virtual machine to hold the dependencies of the mini project. On a Mac, in Terminal you can type:

```
cd MiniProject2/
virtualenv venv -p python3
source venv/bin/activate
```

Instructions for Windows are slightly different[1]:

```
cd MiniProject2/
mkvirtualenv venv -p python3
```

Now the virtual machine should be up and running (you should see the (venv) at the front of the terminal line). Note: A virtual environment is not necessary, but is a good habit for developers to follow.

Next, install the dependencies using Pip.

```
pip install -r requirements.txt
```

You can load the gesture data by calling load `gestures`(). This will return a list of 9 `GestureSets`, one for each type of gesture. A `GestureSet` contains a label (an integer between 0 and 8 representing the gesture, as described in the table above) and a list of 30 Sequences. For example, the first `GestureSet` in the list returned by load `gestures`() contains all 30 "pan left" gesture examples (`Sequences`) recorded for all subjects.

[1] See this tutorial for additional info: http://timmyreilly.azurewebsites.net/python-pipvirtualenv-installation-on-windows/

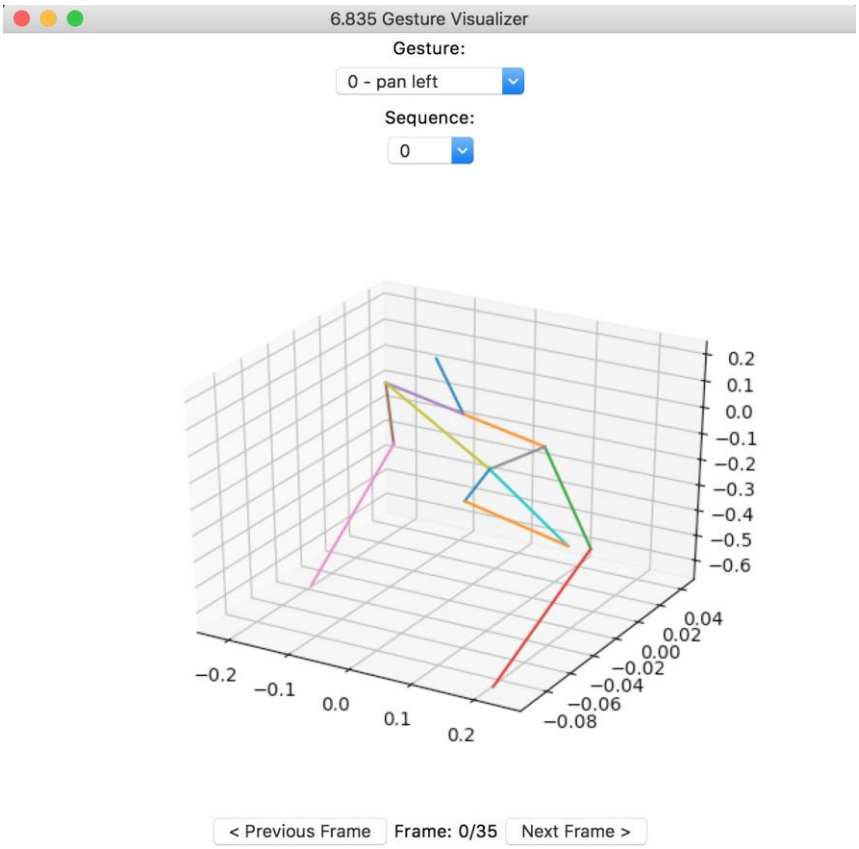

Figure 3: Visualization window showing gesture 0 (pan left).

A `Sequence` represents a single example of a gesture, and is composed of sequential Frames. Each of the 30 Sequences in a `GestureSet` contains N Frames (N varies between Sequences). Each Frame contains 33 features (i.e., the 33 xyz coordinates), which are described above. You can find the class definitions of the `GestureSet`, `Sequence`, and `Frame` classes in Gesture.py.

You can visualize all of the gestures using the command:

```
>> python show gui.py
```

A window like the one in Figure 3 should pop up. Select the gesture class and sequence number from the drop down menus and use the arrow buttons to step through the frames.

Question 1: Look at several sequences of each type of gesture using the GUI. What are some of the differences in how gestures of the same type are performed?

Note that the poses have already been normalized to have the same shoulder width and rotated so that the average direction is facing directly at the camera. What difficulties would you encounter if this were not the case? Describe the difficulties specifically faced by a Nearest Neighbor classifier.

Question 2: In order to test our classifiers for our gestures, we must split our data into training and testing sets. Assuming the training set reflects the distribution of gestures in the test set, what is the best performance you can obtain on the test set by simply guessing the labels (i.e. the chance performance, without looking at the test data)? Explain your answer.

3 Recognition using a nearest neighbor classifier

In this section, you will need to complete these files:

1. `normalize_frames.py`
2. `classify_nn.py`
3. `test_classify_nn.py`

One of the simplest ways to classify a gesture is to compare it to the training gestures using the $L2$[2] distance, and then select the gesture that matches best. Note that we can use this nearest neighbor algorithm only on fixed-length gestures (i.e., gestures with the same number of frames).

Question 3: Explain why we can use the nearest neighbor algorithm only on fixed-length gestures. Implement the following function that normalizes each gesture sequence so that it consists of a fixed number of frames (`num_frames`):

```
normalize_frames(gesture_sets, num_frames)
```

`gesture_sets` is the list of the 9 `GestureSets` loaded by load_gestures.py. Your output from this function should be a list of 9 new `GestureSets`, which each contain 30 `Sequences`, where each Sequence contains `num_frames` Frames. If the length of the original `Sequence` is greater than `num_frames`, you need to sample a subset of `num_frames` evenly spaced frames. For example, normalizing 15 frames to 10 could produce samples: [2, 3, 5, 6, 8, 9, 11, 12, 14, 15]. Explain how you implemented your function.

To test your function, you can run the following in your terminal:
`>> python test_normalize_frames.py <num_frames>`

[2] The $L2$ (also known as Euclidean) distance between two vectors X and Y is defined as $\sqrt{\sum_i (X_i - Y_i)^2}$, no matter what dimension X are Y are.

where num frames is the number of frames you want each Sequence to be. This function will raise an Assertion Error if your frames were not normalized by your normalize_frames function.

To view your normalized gestures in the GUI, go to the show gui.py script and uncomment line 8. Then, run:

```
>> python show gui.py
```

Question 4: Implement `classify_nn(test sequence, training gesture sets)` in `classify nn.py`. The first input, `test_sequence`, is a single `Sequence` from an unknown `GestureSet`. The second input, training gesture sets, is a list of 9 `GestureSets` that each contain a subset of the original 30 `Sequences` to train on.

This function should perform a simple NN (single nearest neighbor) classification, using the $L2$ distance described above. You should return a classification label for the test sequence as an integer between 0 and 8.

Explain and justify the pros and cons of the nearest neighbor classifier for gesture recognition.

Question 5: Now you will test your classifier by implementing `test_classify_nn(num_frames,ratio)` in `test_classify_nn.py`.

Split the data from load_gestures.py into training and testing sequences. The ratio is the percentage of the original data that will be used for training. You should create the testing and training splits, and then call your `classify_nn` function created in Question 4 for each of the test sequences. For each test sequence, compare the output of your `classify_nn` to the actual label of the test sequence to compute the accuracy of your `classify_nn` function. `test_classify_nn` should return this accuracy.

To run your function, type the following into your terminal:

```
>> python test_classify_nn.py <num frames> <ratio>
```

What is the accuracy when `num_frames = 20` and `ratio` = 0.4? Try a few different values for the `num_frames` and ratio parameters and report the values that give you the best accuracy.

4 Recognition using Decision Trees

In the remainder of this Mini Project, most of the code has been written for you. Take the time to understand the classification methods we use. You'll be asked to make optimizations to our implementations. In addition, we'll show you how to use modern, high level APIs to build complex models for your projects.

In this section, you will work with this file: `decision_tree.py`.

The nearest neighbor classifier you just wrote should have a decent accuracy, but we can do better with more advanced classification methods. Decision Trees allow us to predict a label Y from inputs $X_1,X_2,...X_n$. We do this by growing a binary tree. At each internal node in the tree, a test is applied to one of the inputs, say X_i. Depending on the outcome of the test, we either go to the left or the right sub-branch of the tree. Eventually we come to a leaf node, where we make a prediction. This prediction in effect aggregates or averages all the training data points which reach that leaf.

The Decision Tree algorithm belongs to the family of supervised learning algorithms. Unlike other supervised learning algorithms, the decision tree algorithm can be used for solving both regression and classification problems. Since our gesture sequences should be classified into a finite set of labels, we will use a classification tree[3]. The decision tree we implemented for you is located in `decision_tree.py`. To build, train and test the model, just run the script from the command line:

```
>> python decision_tree.py
```

Let's walk through the code together so you can understand and then optimize it.

Question 5: In our implementation, we use `DecisionTreeClassifier` from `sklearn`[4] which builds the binary tree for us according to certain criteria. Run decision_tree.py multiple times (e.g. 10). You'll see that the test accuracy changes each time. Why is this? (Hint: Looking at the source code may be helpful.)

Question 6: Dissect the code and determine how we formatted the gesture data to input into the Decision Tree Classifier. Explain what our inputs, the features, X, and, labels, Y are and describe their shape.

We split our data using the train test split method from `sklearn`. The parameter train _size is given value 0.7; it means training set will be 70% of whole dataset and test set will be 30% of the entire dataset. You can change this parameter by changing the ratio variable in `decision_tree.py`. The random state variable is a pseudo-random number generator state used for random sampling. If you want to split the same way every time use the same value of random state. Try different test/train ratios and report the best one you find. Since the `DecisionTreeClassifier` returns a different test accuracy on each run, you will need to run the script several times to understand what the best `train_size` is.

Question 7: In just two lines of code we can create a Decision Tree Classifier model and train the model on our training data. The model takes in a criterion parameter used to

[3] For more information on classification trees please see the following lecture notes: http://www.stat.cmu.edu/ cshalizi/350/lectures/22/lecture-22.pdf

[4] sklearn is a popular machine learning library for Python. We encourage you to explore it for your final projects.

measure the quality of a split. Our model uses Gini Impurity, as seen in the DecisionTree constructor. Give its equation and describe what it measures.

Question 8: Now it's time to visualize the decision tree. We do this by using the built in export `graphviz` function. As the function saves a .dot file, we must convert it to a .png to view it. This can be done in the command prompt by installing `graphviz`. On a Mac, use Homebrew[5] to install `graphviz`:

```
>> brew install graphviz
```

On Windows, you will need to download the stable release and assure dot is a path variable[6]. To convert a .dot file to .png, run

```
>> dot -Tpng tree.dot -o tree.png
```

Inspect the tree. Report this tree's accuracy. What do the parameters *gini*, *samples*, *value*, *class* refer to? Why do all of the leaves have *gini* = 0?

Extra Credit 1 (10 points max): Extend our implementation by implementing new features for the classifier. Write the code to compute the feature and add it to the system. Some example features might be joint velocity, joint angle, and angular velocity[7]. Report what you've added and any improvements you were able to make.

5 Recognition using Recurrent Neural Networks

In this section, you will work with this file: `rnn.py`.

So far we've seen two different classification methods for classifying our data. In this section we'll jump into a deep learning approach for classification: recurrent neural networks (RNN). RNNs have loops within the network which allows information about the data to persist. Unraveling this loop reveals a chain-like structure mirroring a sequence, which is why RNNs are a natural neural network architecture to use when processing sequences.

We've implemented a Long-Short Term Memory (LSTM) network[8] (a type of RNN) to classify our gestures using `Keras`, a high-level neural network API, written in Python and capable of running multiple backends (i.e. multiple deep learning frameworks such as Tensorflow or Theanos). Here we use the Tensorflow backend. We took advantage

[5] https://brew.sh/

[6] See: https://graphviz.gitlab.io/ pages/Download/Download windows.html

[7] See Zhang, Hao, Wen Xiao Du, and Haoran Li. "Kinect gesture recognition for interactive system." Stanford University Term Paper for CS 299 (2012) for more information

[8] See http://colah.github.io/posts/2015-08-Understanding-LSTMs/ for a thorough explanation of LSTM networks

of Keras' Model functional API. You can find this model in rnn.py. To build, train and test the model, just call the script from the command line.

A simple depiction of the model we've created is in Figure 4.

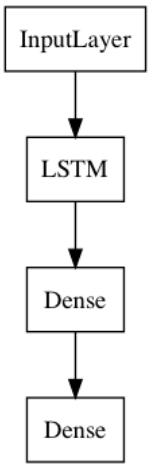

Figure 4: LSTM network

The first layer is an Input Layer, which defines the shape of the input data to our model. The next layer is the LSTM layer which is a recurrent layer that is quite complex. We do not expect you to understand everything about LSTMs, but encourage you to understand them more. The following layers are two Dense layers which apply activation functions to the data and make a final prediction. There are three parameters that go along with the model:

- **batch_size**: the number of samples to be shown to the network before a weight update can be performed. An *iteration* describes the number of times a batch of data passed through the algorithm
- **epochs**: the number of times the algorithm sees the entire data set
- **latent_dim**: specifies how many nodes we should have in the hidden layer

When running the network, you will see a few different outputs in the terminal: two values for loss and two values for accuracy. The first loss and accuracy values refer to how the network performs on the training data. The validation loss (`val_loss`) and validation accuracy (`val_acc`) refer to how the network performs on the test data. The lower the loss, the better a model, usually. The loss is calculated on training and validation and its interpretation is how well the model is doing for these two sets. Unlike accuracy, loss is not a percentage.

It is a summation of the errors made for each example in training or validation sets.

Question 9: LSTM networks require data to be formatted in a specific way. Analyze and describe how we format the gesture data for this network.

Run the LSTM with varying batch sizes (i.e. 8-24), latent dim (i.e. 8-24) and number of epochs (i.e. 40-400). You can change these parameters in the code in rnn.py, and then run the network from the command line:

```
>> python rnn.py
```

Report the best validation accuracy you are able to attain and your values for each variable. Pay attention to the training and validation (test) loss to ensure that you are not underfitting or overfitting your model. An overfit model memorizes the training data instead of learning from it; as a result, the training accuracy is high but the validation accuracy is low. An underfit model performs poorly on the training data because it is unable to capture the relationship between the input examples and their classifications.

Question 10: In both examples, our data has been limited: we have 30 examples of 9 gestures. In order to increase the amount of input information available to the network, we can modify the existing model into a *Bidirectional* LSTM. The Bidirectional LSTM is created by duplicating the first recurrent layer in the network so that there are now two layers side-by-side, then providing the input sequence as input to the first layer and providing a reversed copy of the input sequence to the second.

Modify the model by adding a second LSTM layer and making the model into a Bidirectional LSTM. Draw a network diagram representing this new network, similar to the original network drawing in Figure 4. Run your new model, report its performance as compared to the Unidirectional LSTM. (Hint: use the go backwards parameter in the LSTM constructor)

Extra Credit 2 (10 points max): Improve the model further to gain a better accuracy on the test set. You can do this by adding new layers such as different recurrent layers (Gated Recurrent Units, SimpleRNN, etc.), additional dense layers, etc. Explain your architecture to us and report its accuracy and loss.

Question 11: Provide the staff with feedback. Let us know what you think of the new Mini Project. What did you like about it? What did you not like about it? Were there any ambiguities that should be cleared up?

6 Submission

You should submit to Stellar a zip file containing the following items:

1. A folder with all the files needed to run your code (including all supporting files given as part of the starter code).

2. The PNG image of your Decision Tree from Question 8.

3. A PDF of your write-up containing all of your answers to the questions in this assignment (including your drawing from Question 10).