Terrain Identification from Time Series Data

Hosung Hwang
North Carolina State University
hwang4@ncsu.edu

Alex Carruth
North Carolina State University
agcarrut@ncsu.edu

Utsab Ray
North Carolina State University
uray@ncsu.edu

I. Methodology

The goal of our model was to properly classify terrains given data taken from an accelerometer and a gyroscope attached to multiple test subjects walking along various terrains. There were 8 subjects in the training data and each subject was split into the individual trials for the subject. For each trial, the data was split into the gyroscope and accelerometer data, the time for the gyroscope and accelerometer samples, the classification of the terrain, and the time for the classification of the terrain. The terrain dataset contains classification numbers 0 through 3, with each number indicating a different terrain. 0 indicates standing or walking in solid ground, 1 indicates going down the stairs, 2 indicates going up the stairs, and 3 indicates walking on grass. The accelerometer and gyroscope data were sampled at 40Hz starting at 0s and the classification data was sampled at 10Hz starting at 20ms.

In previous work in this area, *Lee et al.* [1] create a robust 1D Convolutional Neural Network (CNN) which utilizes tri-axial accelerometer data from a smart phone to identify when a subject is walking, running, and staying still. In this paper, three window sizes of 3, 4, and 5 were used with a total of 128 filters to create the feature vectors. Max pooling, dropout, and output with a softmax activation function was also used to finish this model. Using this model as a basis, we created our own 1D CNN model to analyze our data

For this project, our team decided to add on an additional CNN layer with four convolutional layers (using the Keras function 'Conv1D'), all with a kernel size of 2. Each convolutional layer had a filter of size 64 and was followed by a pooling layer (using the Keras function 'MaxPooling1D' with a pool size of 2) and a batch normalization layer (using the Keras function 'BatchNormalization'). Two dense layers (using the Keras function 'Dense') were also added, with a batch normalization layer placed between them. An RNN LSTM model with seven layers, data standardization, and one hot encoding was also trained and ran on the dataset but did not perform as well as the CNN.

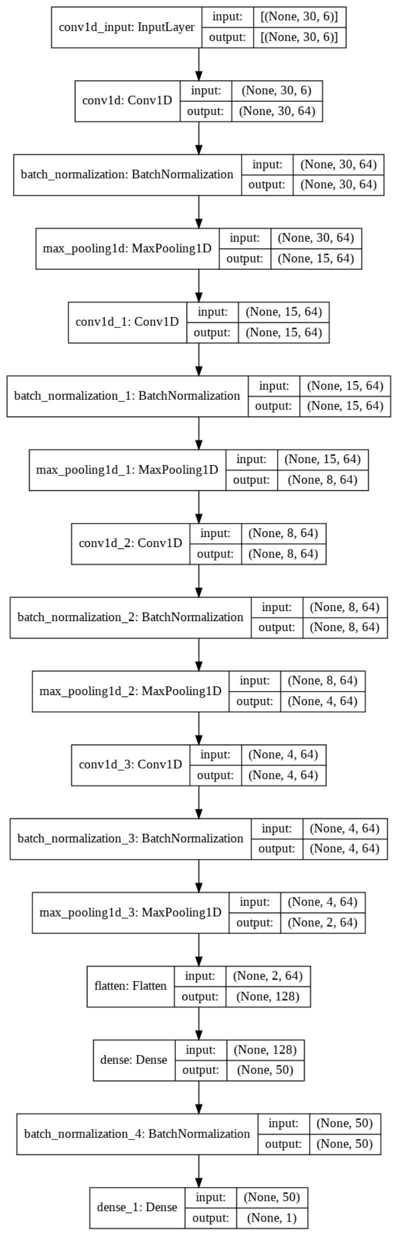

conv1d_input: InputLayer | input: [(None, 30, 6)] | output: [(None, 30, 6)]
conv1d: Conv1D | input: (None, 30, 6) | output: (None, 30, 64)
batch_normalization: BatchNormalization | input: (None, 30, 64) | output: (None, 30, 64)
max_pooling1d: MaxPooling1D | input: (None, 30, 64) | output: (None, 15, 64)
conv1d_1: Conv1D | input: (None, 15, 64) | output: (None, 15, 64)
batch_normalization_1: BatchNormalization | input: (None, 15, 64) | output: (None, 15, 64)
max_pooling1d_1: MaxPooling1D | input: (None, 15, 64) | output: (None, 8, 64)
conv1d_2: Conv1D | input: (None, 8, 64) | output: (None, 8, 64)
batch_normalization_2: BatchNormalization | input: (None, 8, 64) | output: (None, 8, 64)
max_pooling1d_2: MaxPooling1D | input: (None, 8, 64) | output: (None, 4, 64)
conv1d_3: Conv1D | input: (None, 4, 64) | output: (None, 4, 64)
batch_normalization_3: BatchNormalization | input: (None, 4, 64) | output: (None, 4, 64)
max_pooling1d_3: MaxPooling1D | input: (None, 4, 64) | output: (None, 2, 64)
flatten: Flatten | input: (None, 2, 64) | output: (None, 128)
dense: Dense | input: (None, 128) | output: (None, 50)
batch_normalization_4: BatchNormalization | input: (None, 50) | output: (None, 50)
dense_1: Dense | input: (None, 50) | output: (None, 1)

Fig. 1. 1D CNN Network Architecture

II. Model Training and Selection

A. Model Training

For training and validation, we reserved 80% of our training data set for training and 20% of our training data set for our validation data set. To deal with the issue around the fact that the training data set was sampled at differing sampling rates our team decided to up-sample the y data and fill in the values in between each y classification with the previous classification. There was also a class imbalance, with there being a lot more of 0 and 3 classifications than the other 2 classifications. To avoid the prediction issues this might cause, our team decided to use an oversampling technique known as Synthetic Minority Oversampling Technique (SMOTE). Using SMOTE, the two minority classes (in this case, the y classes for 1 and 2) were over-sampled by creating ”synthetic” examples as opposed to over-sampling with replacement [2].

B. Model Selection

Before settling on an enhanced CNN model, our team did extensive work to try and create a working RNN LSTM model in an effort to increase the accuracy of our initial model. The LSTM model that we had been working on had seven LSTM layers, with a hidden neuron size of 100, batch size of 64, and dropout rate of 0.2. The training and validation data was split based on a 4:1 ratio, respectively. Both our training and validation data sets were one-hot encoded and a batch generator was used to enable our massive data set to be used by our potential model. However, even after fitting our model for several epochs, the validation accuracy was never higher than 25%. Our team even tried to use training and validation data without one-hot encoding and without standardization, but the validation accuracy did not amount to anything noticeably above a success rate higher than if the validation data had been classified through random guessing. Hence, this is why our team used an enhanced variant of the CNN model that we had initially used during the preliminary part of the competition project.

The optimized model that we used was a CNN model that used four Conv1D layers, each with batch normalization, padding, filter size of 64, and kernel size of 2. ReLu activation was used for each of the convoluational layers and the Adam optimizer and MSE loss function was used when compiling the model. Figure 2 displays the loss, training accuracy and validation accuracy for models with different optimizers. As we can see from the plots, the model with the Adam optimizer yielded the most consistently high validation accuracy. Additionally, Each convolutional layer included a MaxPooling1D layer each with a pool size of 2. The model also included two dense layers preceded by batch normalization. The validation loss and accuracy of our optimized CNN model after fitting for twelve epochs can be seen in Figure 2.

From the standpoint of validation loss and accuracy, the optimized CNN model was a drastic improvement over our team’s initial CNN model, with the results of fitting over fifteen epochs shown in Figure 1. Our initial CNN model only included three convolutional layers without padding and fitting was done over fifteen epochs rather than twelve, which caused over-fitting of the data, leading to a decrease in the final validation accuracy measurement. When comparing the validation loss and accuracy between the initial and optimized CNN models, it is clear that the optimized CNN model was a significant improvement over its predecessor. Using the optimized model, the validation loss went down 7.19% to 1.0392 while the validation accuracy increased by 9.80% to 66.19%.

Table 1: Validation Loss and Accuracy for Initial CNN Model

Epoch	Validation Loss	validation Accuracy
1	1.7448	0.3936
2	1.3415	0.4686
3	1.2221	0.5348
4	1.2720	0.5260
5	1.2479	0.5463
6	1.3850	0.5096
7	1.2789	0.5590
8	1.3063	0.5526
9	1.2527	0.5673
10	1.1759	0.6033
11	1.2770	0.6008
12	1.3386	0.5805

Table 2: Validation Loss and Accuracy for Optimized CNN Model

Epoch	Validation Loss	validation Accuracy
1	0.9998	0.6032
2	0.9724	0.6085
3	1.0817	0.6422
4	0.9707	0.6607
5	1.0037	0.6547
6	0.9507	0.6770
7	1.0130	0.6467
8	1.0120	0.6496
9	1.0060	0.6711
10	0.9606	0.6700
11	1.0328	0.6496
12	1.0392	0.6619

III. Evaluation

In Figure 3 we show a plot for the training and validation accuracy along with the loss function for our model. We can see that the validation accuracy has multiple local maximas and minimas as the number of epochs increase.

Table 3 shows the precision, recall, accuracy and F1 scores for our model. Table 4 shows the class wise accuracy scores for each of the 4 classes.

Table 3: Precision, Recall, Accuracy and F1 scores

Precision	Recall	Accuracy	F1
0.70	0.63	0.63	0.66

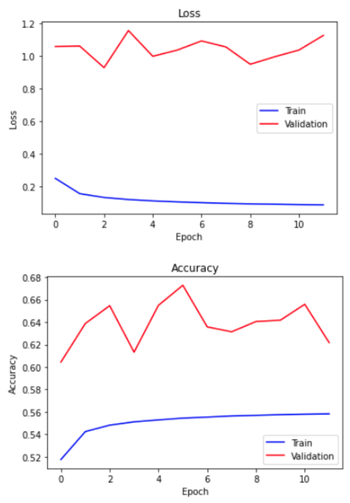
Table 4: Class-wise accuracy scores

	Class 0	Class 1	Class 2	Class 3
Accuracy	0.79	0.67	0.35	0.38

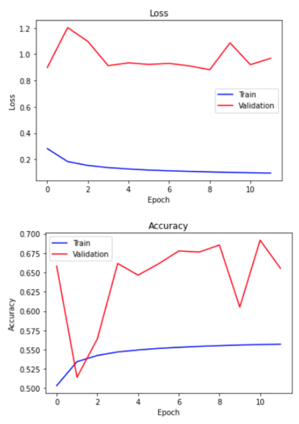
References

[1] Song-Mi Lee, Sang Min Yoon, and Heeryon Cho, “Human activity recognition from accelerometer data using Convolutional Neural Network,” Feb. 2017, pp. 131–134, doi: 10.1109/BIGCOMP.2017.7881728.

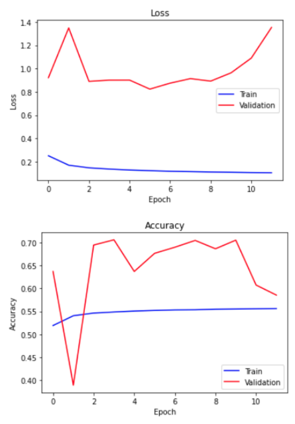
[2] Jack Tan. 2020. How to deal with imbalanced data in Python



(a) Model with Adam optimizer



(b) Model with SGD optimizer



(c) Model with RMSprop optimizer

Fig. 2. Accuracy and Loss plots for different optimizers.

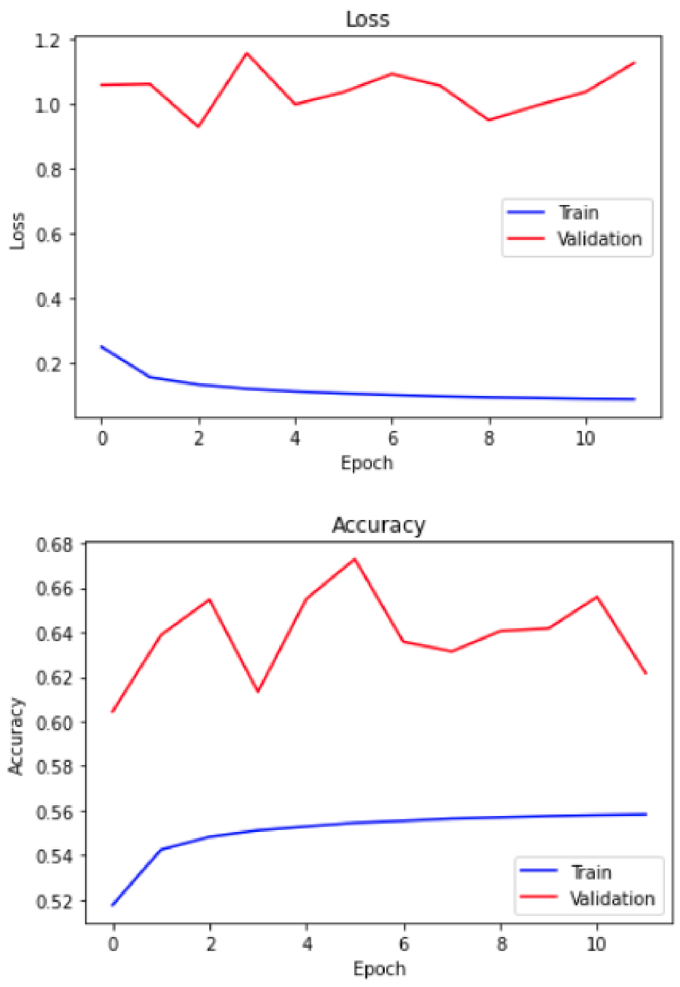


Fig. 3. Accuracy and Loss plot for our model trained on all subjects