[Team 31] Proj-C: Terrain Recognition Using a CNN-LSTM Neural Network

Derik Muñoz Solis
dfmunzos@ncsu.edu

Alec Brewer
ambrewe3@ncsu.edu

Nisan Chhetri
nchhetr@ncsu.edu

I. Methodology

We propose a deep CNN-LSTM model for terrain classification from accelerometer and gyroscope values. For this task, the dataset came from the data collected by the Active Robotic Sensing (ARoS) Laboratory at NCSU[1]. This data was collected in an attempt to classify the terrain that an individual is walking on so that this can be incorporated into new prosthetic designs that can detect the surface being walked on and facilitate the change in gait as a person walks on these different surfaces. The data consists of six total values at each timestamp with three coming from an accelerometer and three from a gyroscope in the form of x, y, and z values. The data for the sensor was sampled at a rate of 40 Hertz while the labels for the training data were assigned at a rate of 10 Hertz, with a label of '0' indicating standing or walking on solid ground, '1' indicating going down the stairs, '2' indicating going up the stairs, and '3' indicating walking on grass. The data is imbalanced however, with the majority assigned to the '0' label, as seen in Figure 1 below.

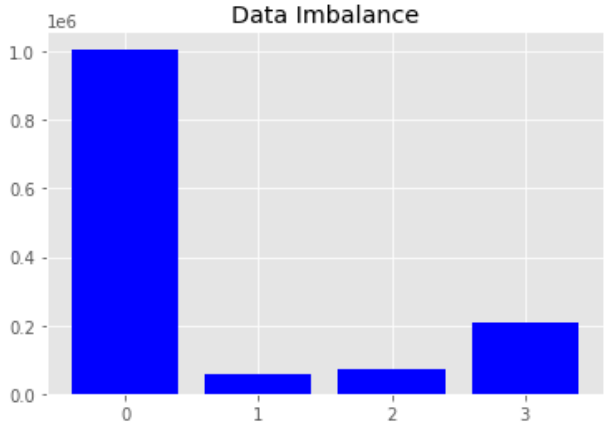


Fig. 1: Distribution of 0, 1, 2, and 3 labels in the provided training set. The data is imbalanced, leaning heavily towards the 0 class label.

A. Data Pre-processing

As stated previously, the data came from the accelerometer and gyroscope sensors of an IMU located at the subject's knee region that collected the data as the subject walked. In order to facilitate the task of classifying the terrain from this data, the first step taken was to scale the data as it has been proven that machine learning algorithms perform better on numerical data when it has been standardized. This is especially true when the data has different units for different portions of the data, as this arbitrary unit can skew learning algorithms.

In order to achieve this scaling, it is typical to subtract the mean and divide by the standard deviation, but in the presence of outliers in the data, it can help to remove the outliers from the calculation of the mean and standard deviation, then use these new calculated values to scale the data. This robust scaling was done with the help of the *sklearn.preprocessing* python module and it's *RobustScaler* class. This robust scaler proved to be more effective at scaling the data than simply making the data zero mean and unity variance. Another issue with the raw data was the data imbalance in terms of the label class distribution, since the data is heavily skewed toward the '0' label and thus a model trained on this data would most likely be biased towards this label.

Two attempts were made to try to balance the data, with the first being simply appending the underrepresented labels to the end of the original signal and adding some Gaussian noise to the data points, but this proved to be difficult to achieve from scratch and not very efficient in the way that it was implemented. The other attempt as discussed by P. Branco et al. [2] was to undersample the labels that were over-represented in order to make the distribution more equal. This was done with the help of the *RandomUnderSampler* class from the *imblearn* Python module. This admittedly might not be the best way to equalize the distribution of the data since undersampling causes a loss of data points that could have been used to train, so it might be worth investigating other data augmentation methods such as Synthetic Minority Oversampling Technique (SMOTE) that can be implemented using the same *imblearn* module.

Since the feature extraction was done using the convolutional layers of the network, no further processing of the data was necessary other than separating the training windows into batches to be passed into the network for training. In previous attempts at this task, the feature extraction was done by creating handcrafted features consisting of the mean and standard deviation for a window of the training data. Using the network to do the feature extraction gave much better results than using these handcrafted features for the classification task.

B. Model Description

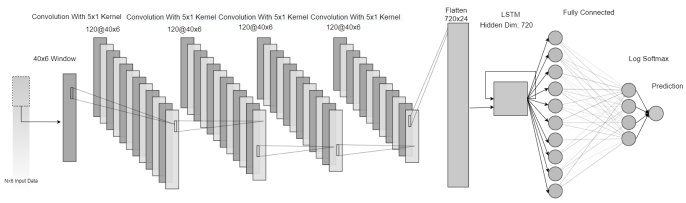


Fig. 2: Simplified Diagram of the CNN-LSTM Network. The network includes 4 convolutional layers used for feature extraction and an LSTM and fully connected layer used for the classification task. (Note that the dimensions and number of neurons are not necessarily to scale and simply for illustrative purposes.)

For the model, we propose using a deep CNN-LSTM model that uses the CNN layers to extract features from the time series data given by the sensors. Those features are then used for the classification task by passing them through an LSTM layer. CNN networks are often used as feature extraction networks and LSTM networks are useful for time series data and forecasting because of their 'memory' component, so together they can be used in a sequence where the CNN layers in a network are used to extract features from the data and the LSTM layer is used to classify the features extracted by the CNN portion. This type of architecture has been explored by Tara N. Sainath et al. for vocabulary task [3], as well as by Xingjian Shi et al for precipitation prediction [4] and has shown to perform well on these types of tasks. We propose using this type of architecture that Tara N. Sainath et al. have called "CLDNN" for the task of classifying terrain from sensor data.

A simplified overview of the proposed network can be seen in Figure 2. The network consists of four convolutional layers that convolve a windowed input of 40 time steps with a 5×1 kernel, resulting in 120 output channels for the first layer. The second convolutional layer takes in the 120 channels and also applies a 5×1 kernel to the data and outputs 120 channels. The same is true for the third and fourth convolutional layers. As this convolution is done without any padding, the size of the window is reduced after every convolution by four rows. The results at the end of the convolutional layers are then flattened to three dimensions since the LSTM layer expects a three dimensional input. The LSTM has a hidden size of 768 and an output dimension of 128. Finally this is passed through a single fully connected layer with on output size of 4 corresponding to the 4 class labels. The final prediction is chosen by the applying the log softmax to the predictions and selecting the most likely label. The final predictions were also filtered to try to remove any predictions that consist of very short periods of a certain label, since it is assumed that there are no rapid changes between labels as any such predictions from the model would be erroneous. This filtering was done using a very simple mode filter with a window size of 20 datapoints.

II. Model Training and Selection

A. Model Training

In order to estimate how well the network might be able to generalize to the test data, the training data was split 80:20 for training and validation respectively. One session was also set aside as a test session to have a general idea of how any hyperparameter changes would affect the F1 score for the test set. The model was trained only on the training set. For the final results, no data augmentation was used since the label distribution was equalized using undersampling, but a further exploration into this task would be to examine various methods of data augmentation and determining how those would affect the performance of the model.

B. Model Selection

A window size of 40 datapoints corresponding to 1 second (since the data was collected at a rate of 40 hertz) was chosen for training. The session chosen to be the test session, as mentioned earlier, was the first session in the dataset titled *subject_001_01__x* for the sensor values and *subject_001_01__y* for the ground truth labels. Once the model had been trained with the chosen hyperparameters, this test session was then inferenced on and compared to the ground truth labels from the session. One of the first parameters that was optimized was the number of kernels for the convolutional layers. All of the convolutional layers output the same number of channels, and this was chosen after trials of the model were conducted with output channels ranging from 6 to 120. Ultimately 120 was chosen as this provided the best F1 score for the test session. The number of convolutional layers to use in the model was also varied from 1 to 4. While there is a positive correlation between the number of layers used for convolution and the gain in the model accuracy, the F1 on the test session seemed to not increase much past 4 layers of convolution and there was a significant increase in training time with more convolutional layers, so ultimately 4 was the number of convolutional layers chosen for the final model.

Number of output chanels for convolution	F1 on test session
12	.8696
24	.88995
48	.89842
96	.90377
120	.90616

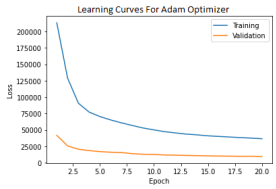
TABLE I: Comparison of the number of output channels for the convolutional layers and the effect this had on the F1 score on the testing session (this was done with one convolutional layers as this was optimized before the number of layers was).

Layers for convolution	F1 on test session	Model Accuracy
1	.90616	.94099
2	.90433	.95237
3	.91143	.95751
4	.912504	.97471

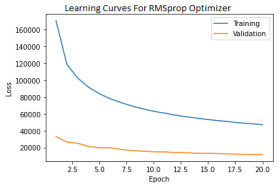
TABLE II: Comparison of the number of convolutional layers and the effect this had on the F1 score on the testing session and the model accuracy.

The model was ultimately trained with Adam optimizer as this was observed to give the steepest learning curve and better

results when training, as can be seen in Figure 3. For this comparison the network was trained with the same parameters and only the optimizer was changed. The optimizers tested were SGD with momentum, RMSprop, and Adam. Only RMSprop and Adam are shown in the figure as these were significantly better than SGD with momentum.



(a) Using Adam optimizer for training gave the best learning curve compared to RMSprop and SGD



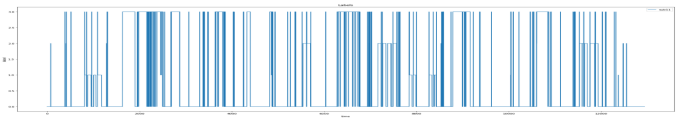
(b) RMSprop also gave good results compared to SGD but not as good as Adam taking a lot longer to reach lower training loss

Fig. 3: Comparison of the two best optimizers the network was trained on. Ultimately Adam was chosen to train the network as it gave the best results.

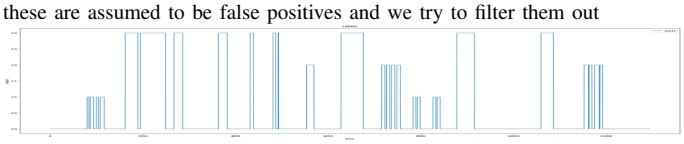
III. Evaluation

Once the parameters were chosen as the ones mentioned above, the model was trained for 100 epochs and resulted in a final model with an accuracy of 96.74%. Training and validation plots were generated and can be seen in Figure 6. One thing to note is that the validation loss was significantly less than the training loss. The validation set was selected by using a percentage of the data from the training set and so the portion of data that was selected might have not been as challenging for the network.

As mentioned in section I-B, the resulting predictions from the model were filtered to try and eliminate any predictions that did not fit in with the surrounding predictions. The rational was that there would not be any short periods of a label in between longer periods of a single label, so these sorts of predictions from the model are assumed to be erroneous and are filtered out. This proved to be key in terms of the final F1 accuracy score for the test set. It was observed when comparing the predictions of a test set session to those of the ground truth that the predictions seemed to have the general shape of the ground truth labels, but due to the multiple false detections, this resulted in a low F1 score. A filter was devised to attempt to eliminate these poor predictions and when applied to the prediction, the results were clear that this filtering improved the F1 score significantly. An example of the filtered outputs compared to the initial predictions straight from the model can be seen in Figure 4 below. The predictions significantly improved from their unfiltered state (Fig. 4a) once they were filtered (Fig. 4b).



(a) Example of an unfiltered prediction from the network. There are several short pulses of a label in between longer sections of another, these are assumed to be false positives and we try to filter them out



(b) Example of a filtered prediction. The short pulses have been filtered out by comparing them against the surrounding labels using a mode filter

Fig. 4: Comparison of the label predictions for test data subject 9 before and after filtering. This filtering proved to be key in terms of improving the F1 score for the predictions.

The final precision, recall, accuracy and F1 scores for each class and the average were calculated for the final model and can be seen in Table III

Class:	Zero Label	One Label	Two Label	Three Label	Average
Precision	.9831	.8344	.8611	.8387	.8793
Recall	.9430	.9553	.9441	.9442	.9467
Accuracy	.9430	.9553	.9441	.9442	.9467
F1 Score	0.9626	0.8907	0.9006	0.8883	0.9106

TABLE III: Table showing the final precision, recall, accuracy and F1 scores for each class and the average

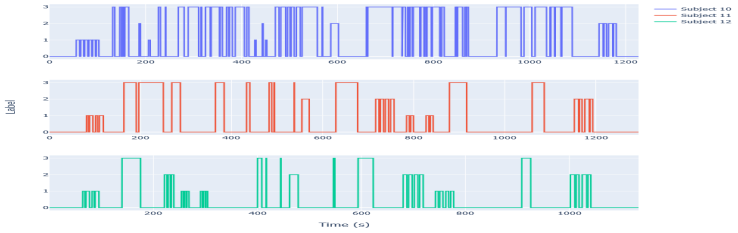


Fig. 5: Final class label predictions for test data subjects 10, 11, and 12.

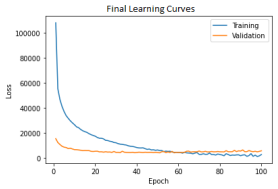


Fig. 6: Final learning curves for the training and validation error.

References

[1] B. Zhong, R. L. da Silva, M. Li, H. Huang, and E. Lobaton, "Lower limb prostheses environmental context dataset," 2020. [Online]. Available: https://dx.doi.org/10.21227/d3z5-n231

[2] P. Branco, L. Torgo, and R. Ribeiro, "A survey of predictive modelling under imbalanced distributions," 2015.

[3] T. N. Sainath, O. Vinyals, A. Senior, and H. Sak, "Convolutional, long short-term memory, fully connected deep neural networks," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015, pp. 4580–4584.

[4] X. Shi, Z. Chen, H. Wang, D.-Y. Yeung, W. kin Wong, and W. chun Woo, "Convolutional lstm network: A machine learning approach for precipitation nowcasting," 2015.