Convolutional Neural Networks and Random Forests Classification Applied to Sequential 1-Dimensional Data

Michael Samarco
Department of Electrical and Computer Engineering, Temple University
michael.samarco@temple.edu

Introduction: This paper explores the process of solving a classification problem for a one-dimensional signal using a machine learning approach. The signal data is provided through a multitude of file captures representing independent trials or realizations of this stochastic process. Specifically, there are 2,000 data files labeled as evaluation data and represent the critical files that the machine learning implementation will ultimately be scored and evaluated on in the assessment of how well it performs. Note that these datasets are "blind", meaning that they do not have any samples within them labeled for events that may arise (i.e. the data is completely unknown to a machine learning system and its developer). Now, in order to develop a system to detect these events in the first place, there must be known data to match to events as they occur in the files. This is the purpose of the "training" (10,000 files provided) and "development" (2,000 files provided) dataset. The training dataset is actually employed in the learning aspect and is fed into the algorithm for it to develop features (some algorithms), become familiar with the statistical distributions of the events, and create decision boundaries. The evaluation data is used sort of like a sample evaluation set, but with known labels (i.e. this dataset can be predicted on, after training, to get an idea of how performance could turn out.

Each file contains amplitude values of some signal vs. time steps. The signal captured has four events that can arise within it and that need to be predicted and detected. The signals are mostly sparse in the sense that most of the data recorded in a given file will be predominantly background noise and the events will occur here and there. Note that although, prediction only needs to be made on non-background events (events 1-4), background noise is very important to a model when trying to distinguish an event from noise and ought to be accounted for in model training of each algorithm.

When tackling this classification problem for the specific dataset given, the thought of obtaining more features may arise. More features—if distinguishable and statistically separated amongst classes/events—means more information gathered and more knowledge that a machine learning model can apply to better separate classes and more accurately make predictions. However, in the case of this specific problem, finding features can be a real challenge and may require an expertise in certain areas or fields of engineering. As an example, see Figure 1, which shows that the distributions [and averages (dashed lines)] for the statistical moments during each event's occurrence appears to be very similar and overlap much. This was the same case when trying to look at frequency content (appeared DC for all event occurrences) or when trying time durations of events to amplitudes (no correlation; covariance was 0). A

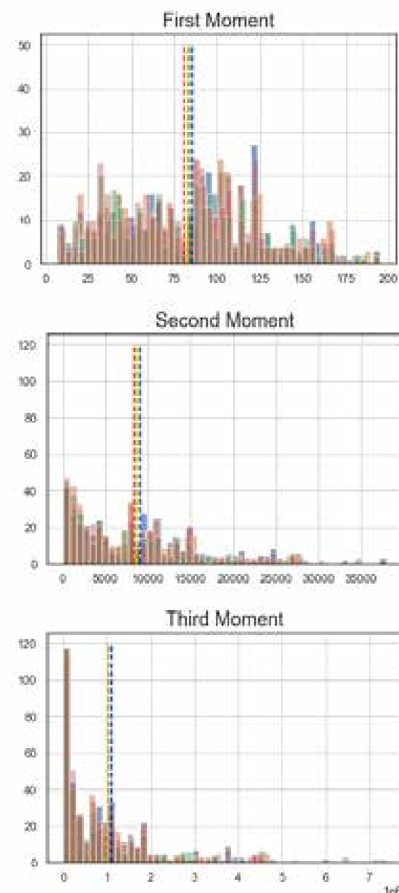

Figure 1. This shows the distributions/histograms among the events for their 1st moment (mean), 2nd moment (variance), and 3rd moment. Note: these were taken for each duration of an occurrence for an event.

reasonable approach would be to disregard any feature learning and apply an algorithm to the raw measurements that is good at learning or discovering features on its own. Two that come to mind are a neural network and random forests classification (RNF). In this case, the neural network chosen is a Convolutional Neural Network (CNN).

Random Forests Classification (RNF): A Random Forrest Classifier works by essentially taking data and randomly placing samples in decision trees (forests) to ultimately perform several feature classifications in each tree before coming to a majority vote on how it is decided the samples collected are best classified. The Python library *Sklearn* has a some nice machine learning functions that can help to build a RNF classifier: the function *RandomForestClassifier*() in conjunction with *RandomizedSearchCV* and *GridSearchCV* can be used to instantiate the model and optimize its parameters to achieve best possible performance. The "Randomized Search" function is nice in that it will apply a grid of parameters and score the fit and track the score for the parameter ranges specified. This will avoid doing the search in an exhaustive way that can be time consuming. It will instead choose random combinations of parameters and track score. Then, after doing that, a more narrow (tight range) can be exhaustively iterated over with "Grid Search CV" to ultimately arrive at the best parameters for the model when applied to this particular dataset. This is exactly what was done for this problem.

Convolutional Neural Network (CNN): Convolutional Neural Networks have this concept of hidden layers as well as an activation function to ultimately activate (or "fire the neuron"). In the hidden layers, weights are applied to the input samples and the samples are propagated through to the outputs. The outputs sum all of the incoming weighted samples from each network (think a feature)—and if the sum surpasses a threshold or meets some criteria, it will produce an output or prediction. The prediction will get better and better as back propagation and gradient descent are performed to optimize the weights and other parameters for the model. In the case of a CNN, the weights are applied to a convolutional window that slides across the sequential amplitudes (i.e. vs time) and tries to describe the signal as summed impulses over time.

For CNNs, the Python Library *Keras* can be utilized. For a CNN, it is just a matter of describing the layers in python and then running a fit and evaluation, where the model iterates over the training datasets and discovers the best parameters to ultimately work its "loss" function down and accuracy [in prediction] up. The layers for the model for this application are described in Figure 2 below.

```
# specify CNN Layers
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=5, activation='relu',\
                 input_shape=(n_timesteps,n_features)))
model.add(Conv1D(filters=64, kernel_size=5,activation='relu'))
model.add(Dropout(0.4))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(100, activation='relu'))
model.add(Dense(n_outputs, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Figure 2. This snippet of code is the example CNN layer description that was applied for the dataset at hand. Note: this is "sequential" data and 1-dimensional (apply 1-D kernel/window size for the convolution). The model also uses some "dropout" and "pooling" layer to avoid overfitting and converging to fast to the solution (learn more and avoid local optima).

Results: For both algorithms described above, the process of fitting the data to a model and tuning parameters is described. Now, for actually applying that model to the sequential datasets, a windowing function is made that essentially slides a fixed window across the raw datasets and computes the probabilities for all of the datapoints that they belong to an event. The greatest number of events that produce the highest probabilities in a given time window will classify the first half of the window and the window will advance in time and slide by half, using overlapping samples (previous window's samples).

For this dataset, it was found that the smallest duration of any of the events lasted for 10 seconds—and so a window size of 10-time steps was chosen for that reason and 5 samples were classified at a time. The results for the prediction performance for the CNN algorithm are shown in **Error! Reference source not found.** below. This scoring is computed using the following equation:

$$P = avg[TPR\ for\ C1 - C4] - avg[FPR\ for\ C1 - C4]$$

Equation 1. The equation for the scoring metric of the classification algorithms.

Conclusions: The scoring results for the CNN are provided. The windowing function for that worked relatively fast compared to the CNN windowing function. The reason is that the CNN "out-of-the-box" algorithm already is set-up to compute an efficient computation on the probabilities for the events given the sampled fixed 10 second window. Therefore, each file could simply be partitioned into these 10 second overlapping windows and that array of windows could all be passed to the CNN predictor at once. Whereas, the RNF algorithm has to rely on sample by sample prediction so it is somewhat tougher to optimize a predictor on all of those files to work quickly on all of the data files. Unfortunately, due to a time constraint and the complexity of the issue, the scoring numbers for the RNF model are not provided. Though, as a comparison, a scoring of accuracy for the evaluation sample formats passed into each algorithm's predict function are given below in Figure 3 (this is the direct output from the modeling and predict code). The accuracy in the prediction for the RNF are much less than that for the CNN; therefore, it is not expected that the RNF model provide better scoring on the dataset file than that provided by CNN. Now, this is not to say the RNF model is fully optimized.

The RNF model parameters could have been swept for higher ranges; however, in the limited time frame for this investigation, it was best to sweep and guess on a smaller range limit of parameters. On top of this, the RNF model was trained on 100 random datafiles as this sped things up and considered less memory since—unlike the CNN, RNF does not train on fixed windows of time steps. 1000 files were also considered in a separate case and this actually showed worse accuracy.

	Data Set		
Algorithm	**Train**	**Dev Test**	**Eval**
RNF	X	X	X
CNF	56.38 %	55.09%	

Table 1. This is the scoring performance for the classifiers when applied to the datafiles. These are computed from Equation 1.

RNF Code IN:

```
# Fit the grid search to the data
grid_search.fit(xTrain, yTrain)
grid_search.best_params_

best_grid = grid_search.best_estimator_

# evaluate performance on 'optimized' param model
yPredBest = best_grid.predict(xDev)

best_acc = metrics.accuracy_score(yDev, yPredBest)
print(f"Accuracy (Optimized Model)={best_acc:.4f}")
```

RNF Code OUT:

```
Fitting 3 folds for each of 80 candidates, totalling 240 fits

{'bootstrap': False,
 'max_depth': 80,
 'max_features': 'sqrt',
 'min_samples_leaf': 1,
 'min_samples_split': 5,
 'n_estimators': 33}

Accuracy (Optimized Model)=0.4727
```

CNN Code IN:

```
# fit network
model.fit(trainX, trainY, epochs=10, batch_size=500, verbose=1)

print("Done Training...\n\n")

# evaluate model
_, accuracy = model.evaluate(devX, devY, batch_size=50, verbose=1)
```

CNN Code OUT:

```
Epoch 1/10
1503/1503 [==============================]- 29s 19ms/step - loss: 0.1715 - accuracy: 0.9383
Epoch 2/10
1503/1503 [==============================]- 28s 19ms/step - loss: 0.1363 - accuracy: 0.9412
Epoch 3/10
1503/1503 [==============================]- 28s 19ms/step - loss: 0.1345 - accuracy: 0.9413
Epoch 4/10
1503/1503 [==============================]- 28s 19ms/step - loss: 0.1340 - accuracy: 0.9413
Epoch 5/10
1503/1503 [==============================]- 28s 19ms/step - loss: 0.1335 - accuracy: 0.9415
Epoch 6/10
1503/1503 [==============================]- 28s 19ms/step - loss: 0.1332 - accuracy: 0.9417
Epoch 7/10
1503/1503 [==============================]- 28s 19ms/step - loss: 0.1330 - accuracy: 0.9417
Epoch 8/10
1503/1503 [==============================]- 28s 19ms/step - loss: 0.1330 - accuracy: 0.9418
Epoch 9/10
1503/1503 [==============================]- 28s 19ms/step - loss: 0.1327 - accuracy: 0.9415
Epoch 10/10
1503/1503 [==============================]- 29s 19ms/step - loss: 0.1326 - accuracy: 0.9417
Done Training...


2902/2902 [==============================]- 6s 2ms/step - loss: 0.1352 - accuracy: 0.9418
```

Figure 3. Code for Evaluation on the CNN and RNF Models and the output of that code.