Sequential Decoding Using Neural and Non-Neural Algorithms

Arpita Das
Department of Electrical and Computer Engineering, Temple University
tup33422@temple.edu

Introduction: The aim of this task is to use machine learning techniques to analyze raw time-series data, perform segmentation to identify events in the data, and use sequential decoding to assign each event a value between 0 and 4. The provided data includes annotated files containing start and end times for events and their corresponding class values, as well as the raw time-series data. The data comprises 10,000 files for training and 2,000 files for development, while an additional 2,000 files represent the evaluation data that the machine learning system will be evaluated on. The evaluation data is blind, meaning that it does not contain any labeled event samples, and the machine learning system and its developer are unaware of the data within this dataset. The training and development datasets serve the purpose of providing known data to match events as they occur in the files and enable the development of a system to detect these events.

After examining the raw data, it was found that the duration of each class pulse follows a uniform distribution, which is a maximum entropy distribution that limits the assumptions that can be made about the duration. Additionally, the pulses are uniformly distributed with similar shapes, making Fourier analysis of limited benefit. Instead, the information is encoded in the amplitudes of each class's initial pulse, which are not consistent between files. As more instances of the event occur, the amplitudes increase, leading to the crucial realization that the class is dependent on both the current observed amplitude of the event and its previous occurrences. As a result, methods such as Hidden Markov Models may not be effective as they do not consider all previous states. Using these observations, two methods will be utilized: a hybrid model (K-Means + LSTM) and a Deep Neural Network (DNN) (autoencoder + LSTM).

K-Means + LSTM: The first model employs K-means for segmentation and an LSTM for seq2seq decoding. To prepare the data, normalization and smoothing are applied, and each file is processed through K-means with K=2, converting the signal to a binary form of zeros and ones. The algorithm assumes that one centroid represents event amplitudes while the other represents non-event amplitudes (DC offset), given the signal's sparsity. Empirical thresholding is applied to fine-tune these clusters: data points above the lowest centroid plus half a standard deviation are assigned as event class, while those below are assigned non-event class. This thresholding accounts for small amplitude pulses. The binary signal is differentiated to extract positive impulses (start times) and negative impulses (end times). Seq2seq decoding is performed using an LSTM, requiring quantization of the raw data to one-hot encodings. The LSTM contains an encoder step, a context vector, and a decoding step. The LSTM's long-term memory enables it to remember previous amplitudes from earlier in the sequence. The output sequence is compared to the true sequence using the Baum-Welsh algorithm.

Autoencoder + LSTM: This approach is like the previous one, but instead of using K-means for segmentation, an autoencoder is employed. The autoencoder is used to map a normalized input to a binary signal that represents events/non-events. The autoencoder consists of an encoder, a latent space, and a decoder. The rest of the algorithm remains the same: using the first derivative to extract timestamps, quantization, applying the LSTM, and evaluating using Baum-Welsh.

Results: To start with the analysis, LSTM and autoencoder models are optimized to ensure optimal performance. This is accomplished by varying the amount of quantization and the number of internal nodes used in the LSTM, as shown in Table 1. After experimentation, the optimal configuration is found to be a quantization value of 100 and an internal node structure of 32.

Table 1. Optimization on LSTM model in terms of Quantized values

LSTM		
Quantization	**# of Nodes**	**Error**
100	32	7.38
100	16	8.92
50	32	8.97
25	32	8.10

Based on these results, the autoencoder is tuned by varying the cost function, shown in Table 2. It is seen that BCE has lower error rates than MSE, thus for further analysis BCE cost function has been chosen.

Table 2. Optimization on Autoencoder in terms of Loss Functions

Autoencoder	
Parameter	**Error**
MSE	16.46
BCE	14.66

After optimization, the results for the training and development sets were reported and shown in Table 3. Overall, the K-means approach was found to be slightly better than the autoencoder approach in terms of error rates and false positive rates (FPR). However, these results were not statistically significant with a confidence level above 45%, so it is difficult to determine conclusively which algorithm is superior. When examining the false positive rates, there was a larger dissimilarity between the two approaches, which was found to be significant up to 80%. Therefore, K-means can be considered superior only when using this metric. In terms of complexity, K-means is much simpler and does not require supervised training like the autoencoder, which makes it a more viable option.

Table 3. Comparison of Performance

Algorithm	**Data Set**					
	Train		**Dev Test**		**Eval**	
	Error	**FPR**	**Error**	**FPR**	**Error**	**FPR**
Kmeans+LSTM	13.12	9.01	14.66	11.22		
AUT+LSTM	14.72	17.23	14.67	18.33		

LSTM

In [210]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
```

In [237]:
```
# Define LSTM model
model = Sequential()
model.add(LSTM(units=32, activation='relu', input_shape=(frame_size, 2)))
model.add(Dense(units=5, activation='softmax'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])


# Reshape the input data to match LSTM input shape
# X = np.reshape(X, (X.shape[0], 1, X.shape[1]))

# Predict the label and probability for each segment
# for segment in segments[0:1]:
    # Reshape the input data to match LSTM input shape
#     X = np.array(segment.iloc[:, :2]).reshape(frame_size, 2)
n_features = 2
X = np.asarray(segments_v2).astype('float32')
X.reshape((X.shape[0], X.shape[1], n_features))
# extract the event label data for the current segment
# y = pd.get_dummies(segment['event labels']).values
y = np.asarray(label).astype('int64')

# Train the model on the segmented data
model.fit(X, y, epochs=10, batch_size=32)

#predict the model
y_pred = model.predict(X)
y_pred_labels = np.argmax(y_pred, axis=1)
y_pred_probs = np.max(y_pred, axis=1)
print(f"Predicted label: {y_pred_labels}, Probability: {y_pred_probs}")

# compute the error for each frame in the current segment
y_true = np.argmax(y, axis=1)
y_pred = np.argmax(y_pred, axis=1)
acc = np.sum(y_true == y_pred) / len(y_true)
# print the error for the current segment
print(f'Accuracy for segment: {acc:.4f}')
    # Print the predicted labels and probabilities for each segment
#     for i in range(len(y_pred_labels)):
#         print(f'Segment {i+1}: Predicted label={y_pred_labels[i]}, Probability={y_pred_probs[i]}')
```

```
Epoch 1/10
```

Fig 01. Snippet of the trained LSTM model

Conclusions: The project involved segmenting a time-series signal and performing sequential decoding. A K-means + LSTM approach is used as a hybrid model followed by a purely DNN method using an autoencoder + LSTM. Many of the metrics resulted in insignificant differences, hence resulted in an increase in false-positive rates. Overall, based on the Dev and Train results, the K-Means approach with LSTM appears to be a better performing model.