**Supervised Training of Recurrent Neural Networks with Simulated Detector Sequences**

**Roberto Santos, September 2016**

The goal of this study is utilize recurrent neural networks (RNNs) to label simulated liquid argon detector signals generated with LArSoft. The simulated signals represent the electrical pulses received by the three wire sets in the detector as sub atomic particles intersect the wires . Within an event window, a single wire will return 9600 sequential measurements. Where a particle intersects a wire corresponds to when the intersection pulses occur in the sequence of measurements. In general, these intersection events can produce electrical signals which are easily distinguishable from the background electrical measurements. However, as these events span several timesteps, the true beginning and end of the particle intersection can be obscured by the background noise. Also, smaller, less powerful intersections can be completely obscured by the pattern of background pulses. Given the temporal nature of the wire measurements, the problem of pulse classification is a natural fit for RNNs.

Through the experiments in this paper, we will strive to find an effective combination of recurrent model structure and data preparation which will be able to label the simulated wire signals accurately.

**Keras**

Several open-source neural network libraries are currently available. Keras (Chollet,2015) was chosen to run the experiments in this paper for the following reasons:

- Support for standard neural network structures and tuning parameters, including several flavors of RNNs.
- Dependencies are easily supported by current lab software.
- Runs on top of Theano or TensorFlow and can be accelerated with GPUs.
- Adequate online documentation available for starting basic structures.
- Simple, expressive Python syntax makes it easy to implement different models and present code that is easy to follow.
- As of this writing, the library is actively supported by the developer and an active online community.

Throughout this paper, sample Python code utilizing Keras is presented to describe RNN models. The actual experiment code utilized a YAML file to set experiment and model parameters. The RNN models were then constructed from these settings from a generic model generator. The code presented in this paper are exact hardcoded translations of the generated models and would produce identical results.

At the end of this document, additional code is presented to round out the examples should the reader wish to experiment with Keras on their own (Appendix B). Also included is a brief description of the common model parameters used in the experiments, but whose choice was not a focus of this paper (Appendix A). Please note the line numbers in the code are for identification only and are not part of Python or Keras. The original experiment code and setting files for this paper can be found at:

  https://github.com/roberto0179/AnalysisWork/tree/master/rnn_experiments

**Glossary of Machine Learning Terms**

Several accepted terms and formulas for evaluating the performance of machine learning models (Fawcett, 2005) are used throughout this paper. How these terms are defined and calculated are outlined below:

- **Signal vs Noise:** used to describe the data in a binary classification problem. Signal is considered the data point of interest that is to be separated or distinguished from the background data, or noise. Signal is commonly labeled with a '1' and noise labeled with a '0'. In our simulated wire data, the electrical pulses marking particle interaction with the wire are sigal, all other electrical pulses resulting from other sources are noise.
- **True Positive (TP):** the number of signal samples correctly identified as signal by the classifier.
- **True Negative (TN):** the number of noise samples correctly identified as noise by the classifier.
- **False Positive (FP):** the number of noise samples misclassified as signal.
- **False Negative (FN):** the number of signal samples misclassified as noise.
- **Recall or True Positive Rate (TPR)** = TP / ( TP + FN )
- **Specificity or True Negative Rate (TNR)** = TN / ( TN + FP )
- **Precision** = TP / ( TP + FP )
- **Accuracy** = ( TP + TN ) / ( TP + TN + FP + FN )
- **F1 Score** = 2 * ( Precision * Recall ) / ( Precision + Recall) = 2TP / ( 2TP + FP + FN ); F1 Score is the harmonic mean of precision and recall.
- **Area Under ROC Curve (AUC):** The Receiver Operating Characteristic is a plot of the performance of a binary classifier as the discrimination threshold is varied. In other words, a plot of the True Positive Rate versus the False Positive Rate. AUC is the area under this curve. Many classification metrics assume a 0.5 threshold for classification. AUC provides a measure of how a classifier performs over the entire threshold range.

**Description of the Data**

Our testing will consider a binary classification, where we will attempt to distinguish particle track signals from background noise in the wire sequence.

The simulated data represent the electrical pulses received by the three wire sets in the detector. Two of the wire sets (U and V) consist of 2400 wires. The third set Y, consists of 3456 wires. Each wire will record 9600 pulses in a single event window. This means a single simulated event window will consist of 8256 continuous sequences totalling just over 79 million electrical pulses.

Three separate, but related sets of electrical data were extracted for each event window. The first turned off the simulated background sources, isolating just the particle intersection signals for each wire. This served to build our labeling set, and these pulses were converted to 0 and 1 to represent background and signal respectively.

The second set laid simulated frequency noise on top of the intersection signals. The third set incorporated simulated white noise on top of the intersection signals. Separating the frequency and white noise doubled the sequences generated for each event window. Though each set is derived the from the same target set.

Initially, representing the electrical pulses as deltas was considered, where the value at each timestep represented the change from the previous signal. However it was decided using deviation from the median of the sequence would provide better overall context for each signal – allowing a smaller segment of wire to retain its relationship to the values of the parent wire. Also, when utilizing

bidirectional networks there would be no need to recalculate the deltas for the backward leg of the network.

For our initial dataset, we extracted the wire data from five simulated events from MicroBooNE . The data from four of these events was reserved for the training and validation of RNN models.  The fifth set of data was used exclusively for testing. Thus making 66048 thousand whole wire sequences available for training and validation, and 16512 whole wire sequences available for testing.

**Feedforward Networks**

To explain how recurrent neural networks work, we will first provide a very general description of feedforward network for contrast (Graves, 2008, 12-18).  In this simple example we have a network with 3 input nodes, a hidden layer with 4 nodes and an output layer with two nodes or classes, in other words a binary classifier (image from Wikipedia, 2016).

Between two layers, all nodes are interconnected with adaptive numeric weights which are initially set to random values. The input nodes are the values, or features, that describe a single sample. The hidden nodes contain sigmoid activation functions that take the sum of weighted inputs to that neuron and determine whether the neuron "fires" or not.  The output neurons represent the classes.  A particular sample will be represented by a value of 1 in the neuron that identifies its class and a value of 0 in the other output neurons.
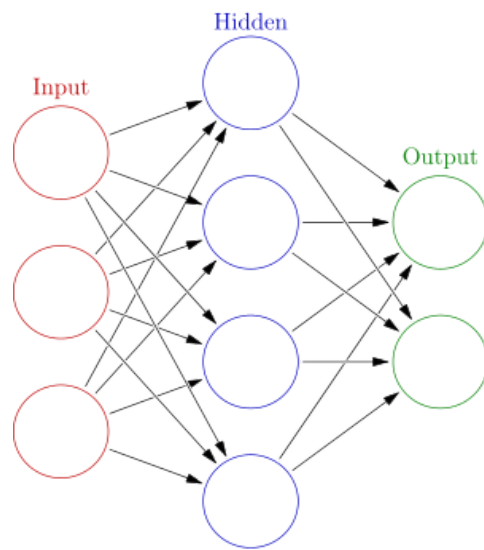


In training such a network, a single sample, represented by 3 features, is passed into the network. These values are multiplied by the weights as they pass to the hidden layer.  At each neuron in the hidden layer, the weighted values are summed and the activation function applied. If the neuron fires a 1 passes out of the neuron to each of the output nodes, where a weight is applied along the connecting edge.  At each output node the weighted inputs are summed and compared to the correct output for the sample.

At this point the output is likely different from what is should be, and this is because the weights have random values.  This is where the backpropagation algorithm is applied. The difference between the values at the output nodes and what they should be is the error for each node.  These error values are sent back to hidden nodes, with the same edge weights applied.  At the hidden nodes the error is calculated, and if there are additional hidden layers, these errors are sent back through the network until the error for each node (except the input nodes) is calculated. As a final step, the weights going into each node are adjusted slightly so that the error should decrease. Whether a weight should increase or decrease is determined by a gradient descent algorithm.

The above steps constitute a single round of training for a single sample.  As this is repeated with additional samples, the desired result is that the weights achieve an optimal value so that error for all samples is minimized. Each training sample that is passed into the network acts on the network independently, and weights are adjusted specifically to the error produced by that sample. There is no

implied relationship between features beyond their shared relationship to a single sample. If we identify the features in this example as A,B and C, it is important that for each sample, feature A enter the same input neurons as feature A for the other samples. However, whether the input neurons are ordered A-B-C or C-A-B in the model would have no impact on the resulting training.

**How RNNs work**

On the other hand sequential information, such as words in a sentence or signals in an electrical waveform, derive a significant portion of their meaning from their context. The individual discernible segments in a sequence have a temporal relationship with each other. What comes before, and sometimes what comes after, a particular point in a sequence helps to identify that point. C follows B follows A has meaning. Recurrent neural networks (Graves, 2008, 18-21) are ideally suited to sequential information in that they allow steps in a sequence to influence the evaluation of succeeding steps.

In an RNN the same set of calculations are performed on each and every element of a sequence, however the output at each timestep in the sequence is dependent on a portion of the prior calculation from the previous timestep . In general, a recurrent network  can be visualized as follows:
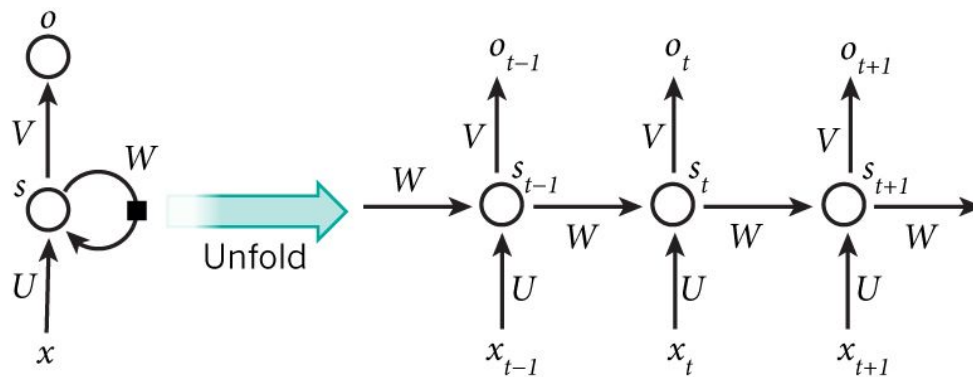


(from Nature through Britz)

In the condensed model at left, **x** is the input sequence and **o** is the output. **U,V** and **W** are the parameters or weights. **S** is the hidden state or hidden layers of the network. All of this is fairly analogous to our previous example of the feedforward network except for one important term:  the **W** parameter that loops back to hidden layer.  To visualize what the **W** term is doing here we can expand the model, or unfold it.  In the diagram on the right, **x** represents each of timesteps in the sequence. Each timestep in the sequence produces an output **o**. After the hidden state **s** is calculated for the first timestep, a weighted value **W** becomes part of the calculations for the hidden state of the next time step, and so on.

A slightly modified backpropagation algorithm, called backpropagation through time, is applied to the unfolded network. Similar to the feedforward network, the error for each node at each hidden state is propagated back to the very first element in the sequence.  However, the major difference from the feedforward network is that the gradient updates are averaged and applied equally to the corresponding weights at each timestep.  Remember, each hidden state in the sequence, with the all the weights, biases and activation functions, is identical for each element in the sequence. So are the **U,V** and **W** parameters. This process is repeated for each new training sequence sample.

**Simple RNN**

In our first experiment we will attempt to train with and classify small segments of our wire sequence. We will slice the 9600 wires into sequences only 100 timesteps long and train on a two layer deep RNN. This model will return a classification on each of the 100 timesteps in the testing sequences.
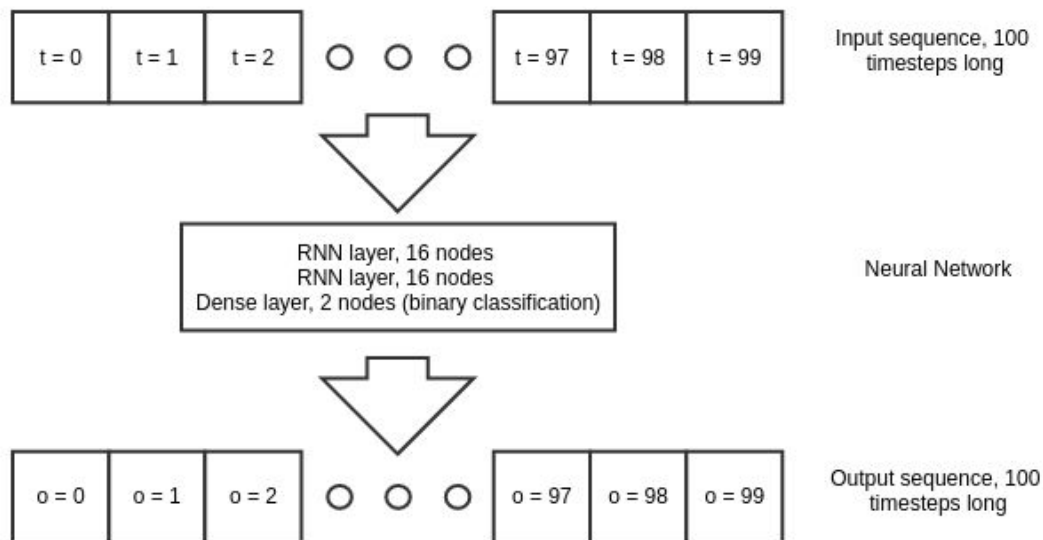


The Keras implementation for this model:

```
01  model = Sequential()
02  model.add(SimpleRNN(16, input_shape=(100,1), return_sequences=True,
                activation='softsign', dropout_W=0.2, dropout_U=0.2))
03  model.add(SimpleRNN(16, return_sequences=True,activation='softsign',
                dropout_W=0.2, dropout_U=0.2))
04  model.add(TimeDistributed(Dense(2)))
05  model.add(Activation('softmax'))
06  model.compile(loss='binary_crossentropy', optimizer=Nadam())
```

So that the reader can become familiar with how Keras implements a model and its parameters, here is a brief description of each line of code:

1. Instantiate model.
2. Add a 16 node, recurrent neural network layer. The first layer requires a description of the input sequence, in this case it's a sequence 100 timesteps long where each timestep consists of a single attribute. Since we want every output from the sequence and not just the last one, return_sequence is set to True. The activation can be any number of typical functions such as tanh or relu. In this case we are using softsign. Dropout helps to keep the network from overfitting to the training data. Here we are applying dropout to the input weights(W) and recurrent weights(U).
3. Here we add a second RNN layer, however the input description is not required.
4. This is our output layer with 2 nodes, one for each class. The TimeDistributed wrapper allows us to return the output at each timestep. Otherwise just the output from the last timestep would be returned.

5. This adds the activation function softmax to the output layer. Softmax is a typical activation for binary classification.
6. We compile our model to ready it for use. Our loss function is binary cross entropy, Keras' version of logistic loss. The optimizer is applied to our gradient descent algorithm to assist in updating weights during back propagation. Nadam is a version of RMSprop (which adapts learning rates during training and is commonly used in training recurrent nets) that adds Nesterov's Momentum.

After training for just 10 epochs we achieve the following overall classification performance for our model on the frequency noise simulation for the U wire set in our test dataset:
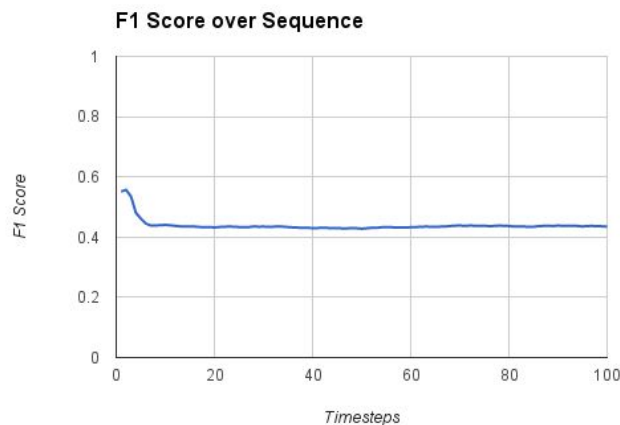


| | |
|---|---|
| True Positive: | 955897 |
| False Positive: | 1896306 |
| True Negative: | 19623173 |
| False Negative: | 564624 |
| Recall (TPR): | 0.6287 |
| Specificity (TNR): | 0.9119 |
| Precision: | 0.3351 |
| Accuracy: | 0.8932 |
| F1 Score: | 0.4372 |
| AUC: | 0.8547 |

Our F1 score throughout the sequence is unimpressive, though fairly consistent after an initial drop. Since the particle intersection events on a wire can be mapped to a location on the wire based on its occurrence in the time sequence, it is fairly straightforward to visualize the performance of our trained model by mapping the predictions on a grid of wires and timesteps. We can look at a small window from this grid taken from one of the frequency noise wire sets in our test data. It is important to note that the sequences start on the left and end on the right:
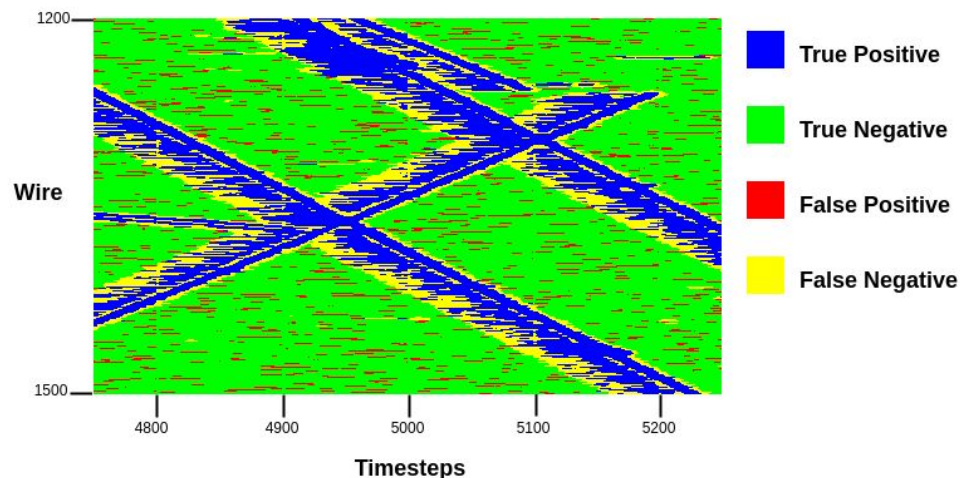
Visually, we can see the main tracks of intersection where particles cross the wire plane. Given the background in this test sample is comprised of frequency noise, it is easy to imagine that the network is mistaking the false structures of the frequency noise (red lines) as particle interaction. We can also see that while the model is generally finding the main structures of the particle intersection signals, it is missing the large parts of these structures, often on the left or beginning of these structures as it trains left to right. The results looks even more chaotic on a single section of wire:
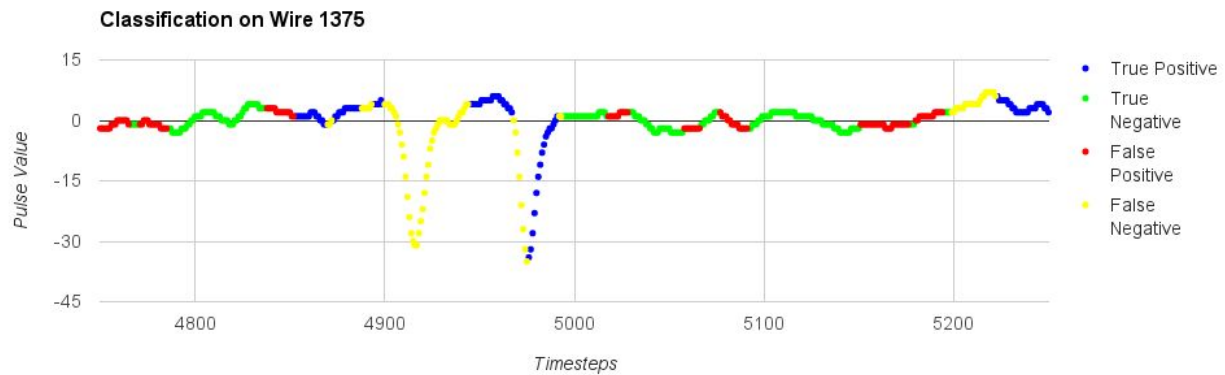


**How LSTMs work**

Training an RNN can suffer from the same problem as a deep multilayer network in that as error calculations travel back through multiple nodes, most of the correction occurs closer to the beginning of the back propagation, or toward the end of the sequence, and it becomes harder to adjust weights earlier in the unfolded sequence. This is termed the vanishing gradient problem. (LISA Lab, 2010) As the sequences in an RNN become longer, earlier timesteps have less influence on correcting the gradients in the hidden state. That means for long sequences, the most recent timesteps greatly influence training, whereas earlier timesteps affect training very little or none at all. If we view the influence of the value passed along the **W** edge in the RNN illustration as a kind of memory, then the RNN can only remember a few timesteps prior to the current step in the sequence.

Long Short Term Memory networks, or LSTM, (Graves, 2008, 31-38) are a modification of RNNs that allow the influence of timesteps to be passed further along a sequence than is possible with a simple RNN. This is accomplished through the addition of a separate cell state that is passed to each timestep. At each timestep this cell state can be altered or allowed to pass unchanged through the current timestep to the next one. The cell state can also be fed into the current hidden state calculations to provide its influence. In this way the value of hidden states occurring early in a sequence can find their way all the way through to last element in a sequence.
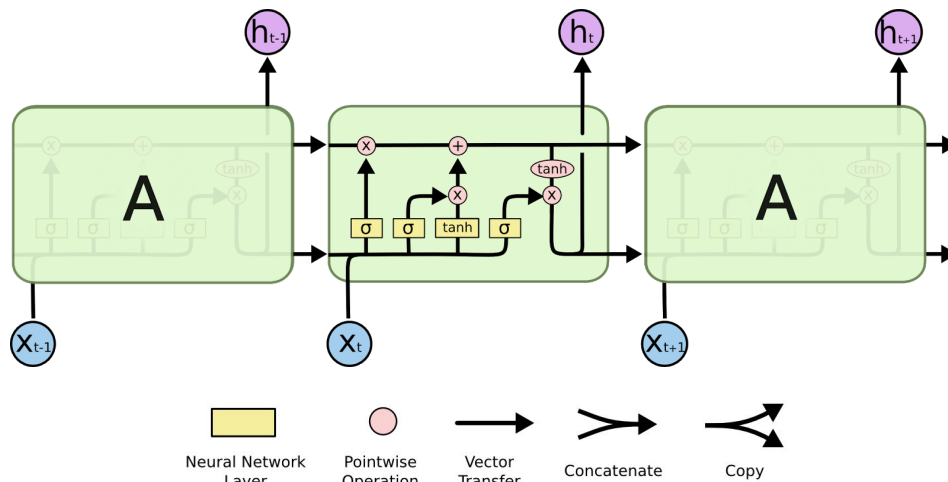
To explain this structure as simply as possible (LISA Lab, 2010) - Each rectangular area represents the processing of a single timestep in the sequence. The bottom horizontal path is similar to the hidden state portion of a regular RNN. The top horizontal path is the cell state which makes LSTMs different. Between these two paths there several sigmoids which act as gates through which the hidden state can affect the cell state. Toward the end of this structure the cell state provides its influence on the hidden state and the output for the timestep. Both the hidden state and cell state are then passed to the next timestep. It is easy to see how the addition of the cell state sidesteps the vanishing gradient problem by providing a potentially uncluttered path from timestep to timestep. In this way, the LSTM is able to 'remember' over a much longer sequence than a regular RNN.

**LSTM Model**

We can change our earlier RNN model to an LSTM model in Keras simply by changing the two hidden layer instantiations:

```
01  model = Sequential()
02  model.add(LSTM(16, input_shape=(100,1), return_sequences=True,
                 activation='softsign', dropout_W=0.2, dropout_U=0.2))
03  model.add(LSTM(16, return_sequences=True,activation='softsign',
                 dropout_W=0.2, dropout_U=0.2))
04  model.add(TimeDistributed(Dense(2)))
05  model.add(Activation('softmax'))
06  model.compile(loss='binary_crossentropy', optimizer=Nadam())
```

We left the other parameters unchanged and again trained for 10 epochs and achieved the following:

| | |
|---|---|
| True Positive: | 727361 |
| False Positive: | 146812 |
| True Negative: | 21372667 |

```
False Negative:        793160
Recall (TPR):          0.4784
Specificity (TNR):     0.9932
Precision:             0.8321
Accuracy:              0.9592
F1 Score:              0.6075
AUC:                   0.8760
```
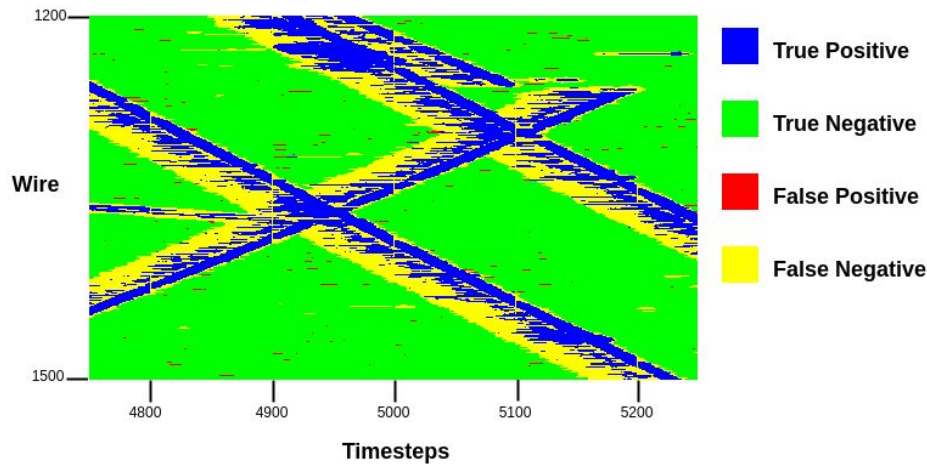


Aside from recall, all of our metrics have improved over the regular RNN. Here there is a significant improvement in the false positives. The network now appears to be learning to exclude the false structures of the frequency noise (at some expense to the true positive rate). The results from the LSTM network indicate that even in a sequence as short as 100 timesteps, a regular RNN is at a disadvantage in learning important structures in the sequence.

**Statefulness**

In the two previous experiments there is enough misclassification occurring at the start of the sequences that the our slicing boundaries are clearly visible in the visualizations of the results. Why is easy to understand if we reexamine our diagram of the unrolled RNN.
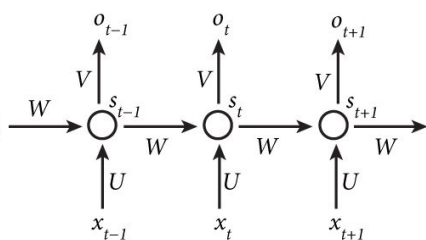


At the beginning of the sequence the **W** vector is empty. There is nothing being passed to the first timestep, and there is nothing the first timestep can learn about its place within the sequence. In fact, it can take several timesteps for the sequence to learn enough structure about itself that the hidden state becomes useful to the subsequent timesteps. A recurrent network missing this initial hidden state is described as being stateless (Brownlee, 2016).

Seeing that our test sequences are actually sliced from a much longer sequence, we know that these initial timesteps do exist within the context of previous data. It would be useful that when a new sequence enters the network, the hidden state (and cell state of an LSTM) from the end of the previous sequence could be inserted into the beginning of the next sequence. Thus rendering the network stateful for the first timestep of the new sequence:
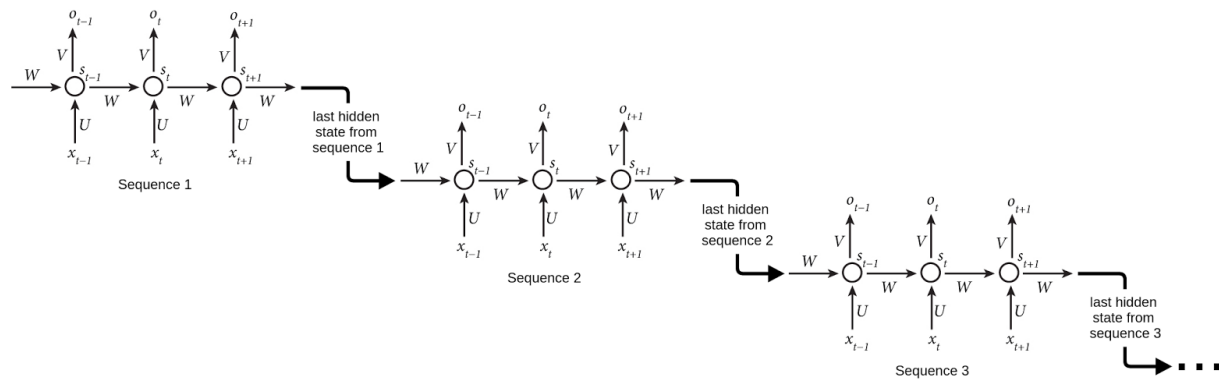
## Stateful LSTM

Keras provides an easy mechanism to accomplish this. By setting stateful = True in each of the hidden LSTM layers in lines 2 and 3, we direct the recurrent network to initiate a new batch with states from the last batch.

```
01  model = Sequential()
02  model.add(LSTM(16, input_shape=(100,1), return_sequences=True,
                activation='softsign', dropout_W=0.2, dropout_U=0.2,
                stateful=True))
03  model.add(LSTM(16, return_sequences=True,activation='softsign',
                dropout_W=0.2, dropout_U=0.2, stateful=True))
04  model.add(TimeDistributed(Dense(2)))
05  model.add(Activation('softmax'))
06  model.compile(loss='binary_crossentropy', optimizer=Nadam())
```
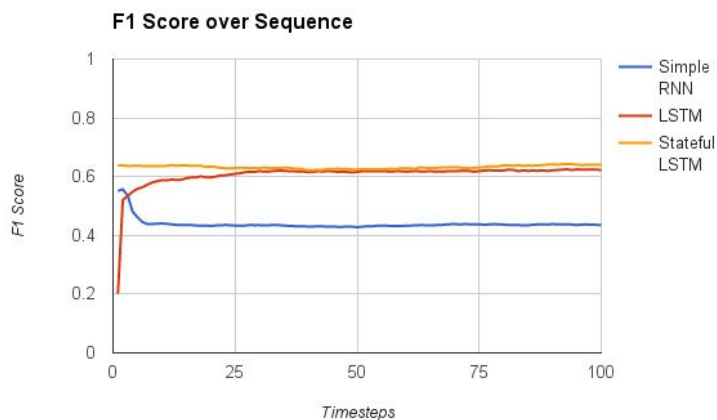


| | |
|---|---|
| True Positive: | 825181 |
| False Positive: | 264184 |
| True Negative: | 21255295 |
| False Negative: | 695340 |
| Recall (TPR): | 0.5427 |
| Specificity (TNR): | 0.9877 |
| Precision: | 0.7575 |
| Accuracy: | 0.9584 |
| F1 Score: | 0.6324 |
| AUC: | 0.8864 |
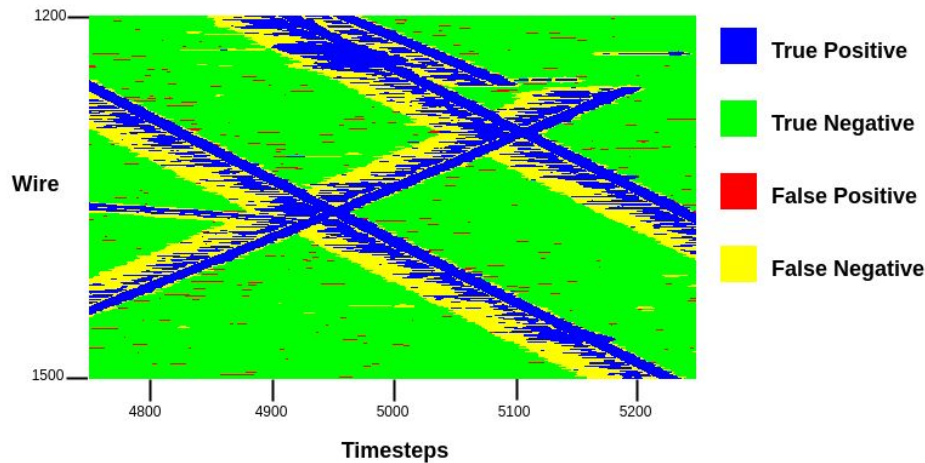
Adding statefulness to our model has not really improved the overall metrics. However the classification performance of the initial timesteps is much improved and is now on par with the rest of the sequence. The visualization confirms that the passing of hidden states from sequence to sequence is having a positive effect on training, as the sequence slices that were visible in previous tests have now disappeared.
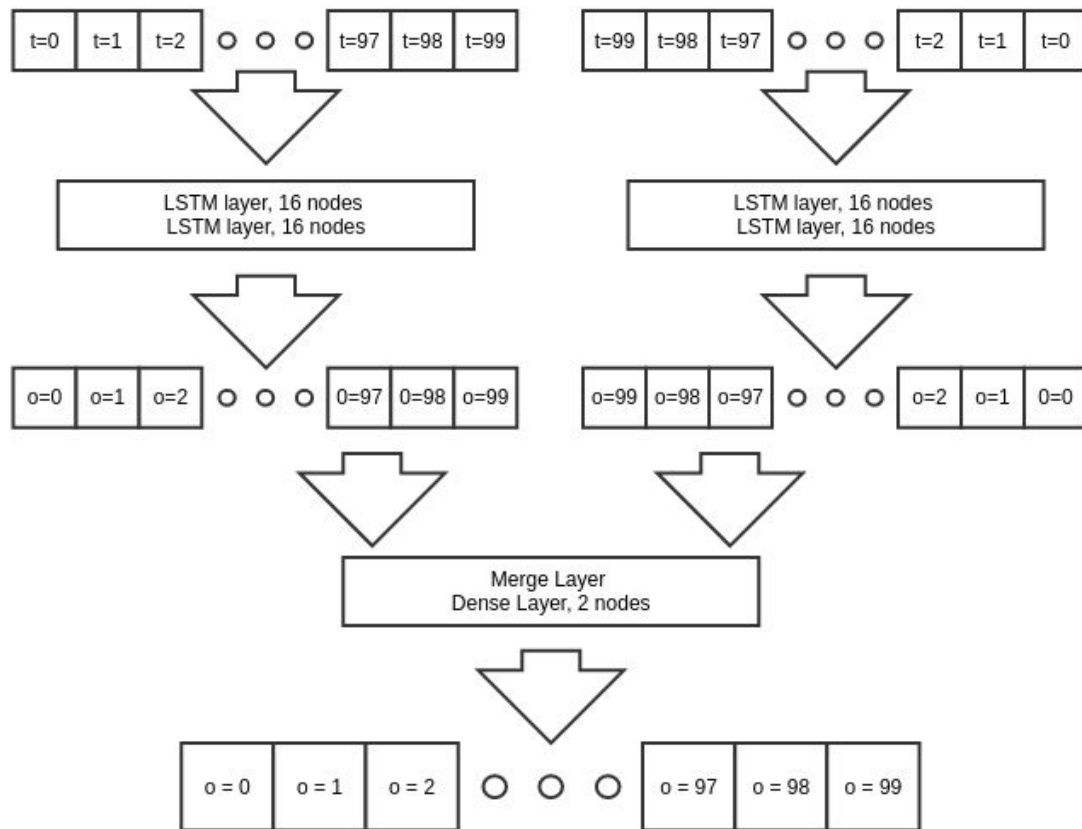
**Bidirectional Networks**
The three networks tested thus far have one thing in common in regards to how training and testing are handled. All of the sequences have a beginning and end, and these models process the timesteps from beginning to end. If memory in an RNN is passed to subsequent timesteps, that means the sequence after a specific timestep can have no bearing on its training or testing. Only the hidden states from the previous timesteps are used in calculation.

This limitation can be visualized in the three previous models by where the preponderance of false negatives are occurring, at the left or beginning of true positive segments. As the model encounters a distinct particle track, it is not correctly identifying this segment until it has processed a significant portion of it. The models appear to be more effective in latter half of these segments.

That the models are having difficulty in these specific pulses is not surprising. The electrical values in these sections are easily hidden within the range of background noise. Yet this is also true for the back end of these segments, and classification performance in these areas does not seem as weak. It can be hypothesized that once the model has correctly placed itself within one of these segments, it handles these more confusing sections differently. If the model could 'look ahead' to the pulses following a timestep, it may process these early particle segments more effectively.

Bidirectional recurrent networks (Graves, Jaitly, Mohamed, 2013) attempt to tackle this problem by including layers that process a sequence backwards. Essentially there are two sub-networks, each one acting as a typical recurrent network, except that one processes the timesteps in the normal forward manner, and the other processes the same sequence in reverse. The outputs from both of these subnetworks is then merged so that each timestep is influenced by the forward processing and the backward processing. The processing at the merge layer is handled timestep by timestep where any number of functions can be applied. For example, summing the forward and backward outputs, or taking the maximum value.

## Bidirectional Model in Keras

Keras provides a shorthand wrapper for bidirectional networks that builds this structure under the hood, however we are not going to use this and instead program it in the longer form so the basic structure of bidirectional network can be more easily visible:

```
01  left = Sequential()
02  left.add(LSTM(16, input_shape=(100,1), return_sequences=True,
              activation='softsign', dropout_W=0.2, dropout_U=0.2,
              stateful=True))
03  left.add(LSTM(16, return_sequences=True,activation='softsign',
              dropout_W=0.2, dropout_U=0.2, stateful=True))
04  right = Sequential()
05  right.add(LSTM(16, input_shape=(100,1), return_sequences=True,
              activation='softsign', dropout_W=0.2, dropout_U=0.2,
              stateful=False, go_backwards=True))
06  right.add(LSTM(16, return_sequences=True,activation='softsign',
              dropout_W=0.2, dropout_U=0.2, stateful=False,
              go_backwards=True))
07  model = Sequential()
08  model.add(Merge([left,right], mode = 'sum'))
09  model.add(TimeDistributed(Dense(2)))
```

```
10  model.add(Activation('softmax'))
11  model.compile(loss='binary_crossentropy', optimizer=Nadam())
```

There are several special things to note about this model:

1. This instantiates the forward processing portion as its own model.
2. This line and the next are the two hidden layers of the forward processing portion of the model. In all respects they are identical to the hidden layers of the stateful LSTM model presented earlier. This layer has the input shape which describes the input data.
3. The second hidden layer of the forward processing model.
4. This instantiates the backward processing model. At this moment it is an entirely separate model from the forward processing model.
5. This layer requires an input shape since it is the first hidden layer of a model. The attribute go_backwards=True instructs the model to process the input sequence backwards. This makes it unnecessary for us to prepare a separate set of training data with the timesteps reversed. You will also notice that stateful=False in the the backwards processing. Why will be explained a little later.
6. The second hidden layer of the backward processing model.
7. This instantiates the main model.
8. The layer merges the results from the forward and backward LSTMs into a single layer. The 'left' and 'right' models are passed in as arguments, along with the setting mode=sum. This directs that the outputs from the two models are summed together.
9. This is the output layer as before, which returns a classification for each timestep
10. The activation for the output layer.
11. Compile the model for use.



F1 Score over Sequence
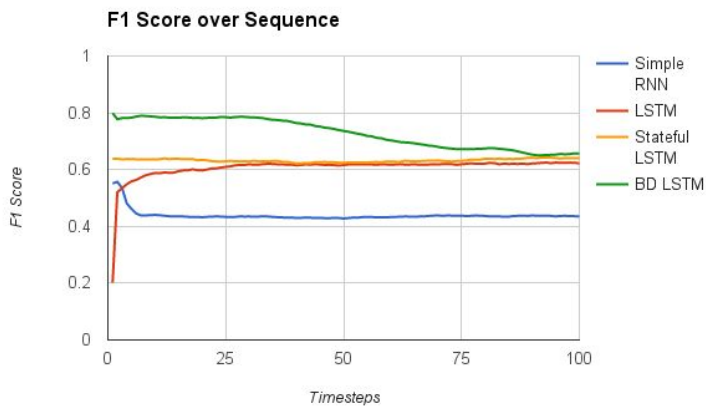
| | |
|---|---|
| True Positive: | 1174592 |
| False Positive: | 529919 |
| True Negative: | 20989560 |
| False Negative: | 345929 |
| Recall (TPR): | 0.7725 |
| Specificity (TNR): | 0.9754 |
| Precision: | 0.6891 |
| Accuracy: | 0.9620 |
| F1 Score: | 0.7284 |
| AUC: | 0.9571 |
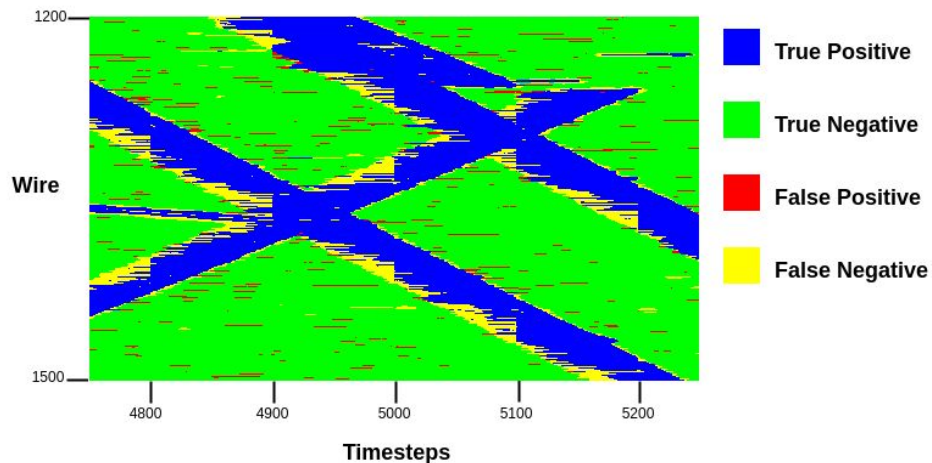
The metrics with this model have improved. In the visual we can see marked improvement in recognizing the beginning of particle tracks on a sequence, though not everywhere. In fact performance in this particular area is worse at the end (or right side) of the sequence slices. This is apparent in the F1 score over the timesteps in the sequences and in our wire plane visual, where the divisions between sequences are once again visible.

This model is bidirectional over the 100 timestep sequences only, not over the entire wirelength. The reason for this is that this model processes the samples one sequence at a time and has to combine the forward and backward evaluations after each sequence. So while the forward sequences are handled in order from the beginning of the wire to the end. The backward sequences are actually out of order.

This is why in the Keras code the backward evaluations have statefulness set to false, whereas statefulness can still be used in the forward processing. This lack of statefulness in the backwards processing means the performance in the beginning of the backward sequences is the weakest. And this is reflected by a drop in performance in the latter timesteps of combined result.

To utilize statefulness for the backward direction it would be necessary to process all of the backward sequences for a single wire in their correct order from last sequence to first and store the states so that the correct backward state can be combined with the corresponding forward state once it has been processed. While this is possible programmatically and would add to the training time, it may be unnecessary.

Statefulness allows important states to pass from sequence to sequence, in effect turning a series of sequences into a single long sequence. It may be that an LSTM network would retain enough wire structure to process sequences much longer than our designated 100 timesteps (Graves, 2008, 32).

**LSTM and Long Sequences**
To test if an LSTM network can in fact handle the entirety of single wire's 9600 timesteps, we modify our bidirectional model slightly to accept sequences of the full wire length. Also, statefulness is now unnecessary as each sample now represents a full time series from beginning to end rather than a subset of the wire.
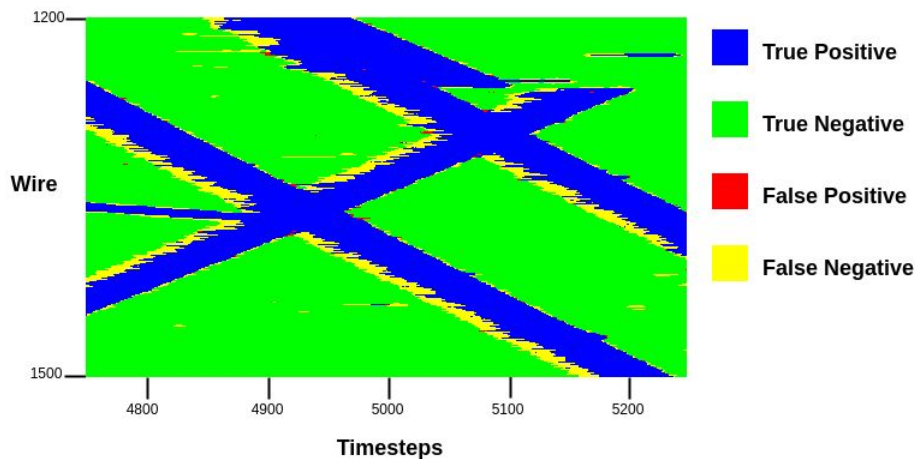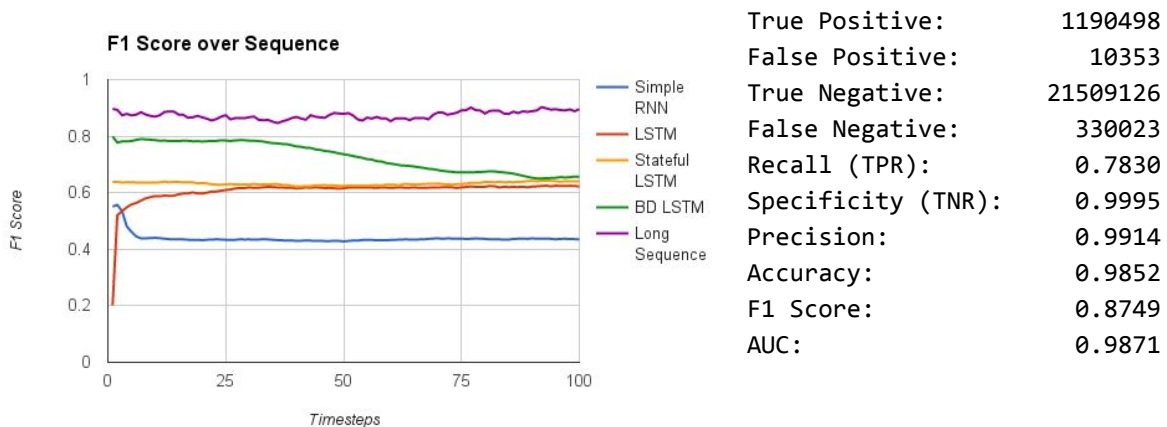
```
01  left = Sequential()
```

```
02  left.add(LSTM(16, input_shape=(9600,1), return_sequences=True,
              activation='softsign', dropout_W=0.2, dropout_U=0.2))
03  left.add(LSTM(16, return_sequences=True,activation='softsign',
              dropout_W=0.2, dropout_U=0.2))
04  right = Sequential()
05  right.add(LSTM(16, input_shape=(9600,1), return_sequences=True,
              activation='softsign', dropout_W=0.2, dropout_U=0.2,
              go_backwards=True))
06  right.add(LSTM(16, return_sequences=True,activation='softsign',
              dropout_W=0.2, dropout_U=0.2, go_backwards=True))
07  model = Sequential()
08  model.add(Merge([left,right], mode = 'sum'))
09  model.add(TimeDistributed(Dense(2)))
10  model.add(Activation('softmax'))
11  model.compile(loss='binary_crossentropy', optimizer=Nadam())
```

Since the results from this model is classification over the entire wire of 9600 timesteps, we present the F1 scores sampled from the 100 timesteps in the center of the wires (timesteps 4750 through 4849) for comparison on our chart. The metrics are for all timesteps in the wireset:



| | |
|---|---|
| True Positive: | 1190498 |
| False Positive: | 10353 |
| True Negative: | 21509126 |
| False Negative: | 330023 |
| Recall (TPR): | 0.7830 |
| Specificity (TNR): | 0.9995 |
| Precision: | 0.9914 |
| Accuracy: | 0.9852 |
| F1 Score: | 0.8749 |
| AUC: | 0.9871 |

We see a marked improvement in all our metrics. Notice in the visual that the false positives that likely marked frequency structures in the noise have all but disappeared. Also, the problems caused by shorter sequences and lack of statefulness in backwards training are no longer relevant. If we return to our example of the single wire we see better organization as well:
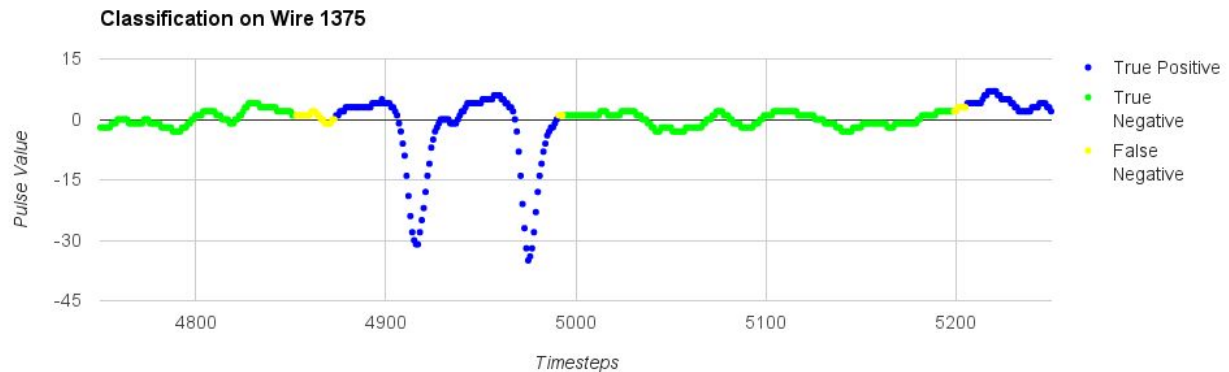


We still see that the model has difficulty at the very ends of the particle intersection signal, but this area is much reduced in size. Of the several modifications to the structure of the RNN model that were tested, the bidirectional LSTM training over the full wire length has so far proven to be the most effective, despite the length of the sequences at 9600 timesteps.

**Multiple Attributes**
In this experiment we will attempt to enhance the feature set over what has been presented before to see if we can address the difficulty the model is having at the very ends of the signals. The current single feature for each timestep we are attempting to classify is the value of the electrical pulse itself at that timestep. The electrical pulses before and after a particular point are evaluated by the RNN and provide their influence in the classification of that time step and should be redundant information if added as features. When we examine the visual map of how particles interact with the entire wire set, we see a particle track often intersects the corresponding timesteps on neighboring wires as it intersects the wire plane. If we treat these neighboring pulses across wires as features of a single timestep, we can see if this provides additional learnable structure for the RNN network.
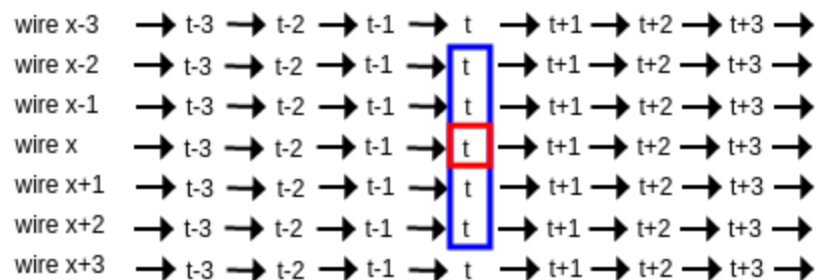


The data arrays are modified to now include the additional features. In our Keras model, the input shape needs to account for these features as well:

```
01  left = Sequential()
02  left.add(LSTM(16, input_shape=(9600,5), return_sequences=True,
```

```
            activation='softsign', dropout_W=0.2, dropout_U=0.2))
03  left.add(LSTM(16, return_sequences=True,activation='softsign',
            dropout_W=0.2, dropout_U=0.2))
04  right = Sequential()
05  right.add(LSTM(16, input_shape=(9600,5), return_sequences=True,
            activation='softsign', dropout_W=0.2, dropout_U=0.2,
            go_backwards=True))
06  right.add(LSTM(16, return_sequences=True,activation='softsign',
            dropout_W=0.2, dropout_U=0.2, go_backwards=True))
07  model = Sequential()
08  model.add(Merge([left,right], mode = 'sum'))
09  model.add(TimeDistributed(Dense(2)))
10  model.add(Activation('softmax'))
11  model.compile(loss='binary_crossentropy', optimizer=Nadam())
```

Since this experiment uses the full 9600 timestep sequence, we again present the F1 scores from the 100 timesteps in the center (timesteps 4750 through 4849) for comparison:
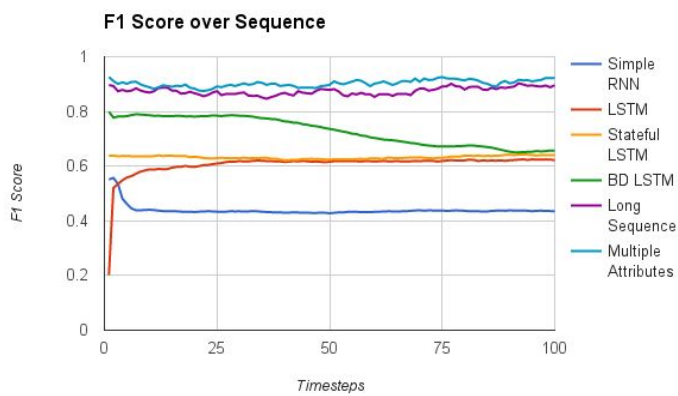


```
True Positive:            1260769
False Positive:             17218
True Negative:           21502261
False Negative:            259752
Recall (TPR):              0.8292
Specificity (TNR):         0.9992
Precision:                 0.9865
Accuracy:                  0.9880
F1 Score:                  0.9010
AUC:                       0.9887
```
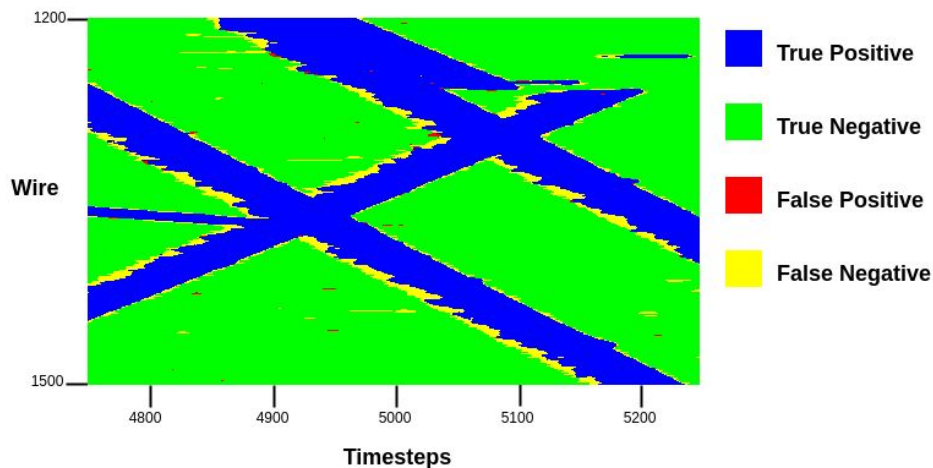
Overall, there is a modest gain in the metrics. Comparing this visual to the previous experiment, there appears to be a kind of "clumping" among neighboring wires within the leading edge of the particle intersections, an obvious result of including pulses from neighboring wires as features to a single timestep.
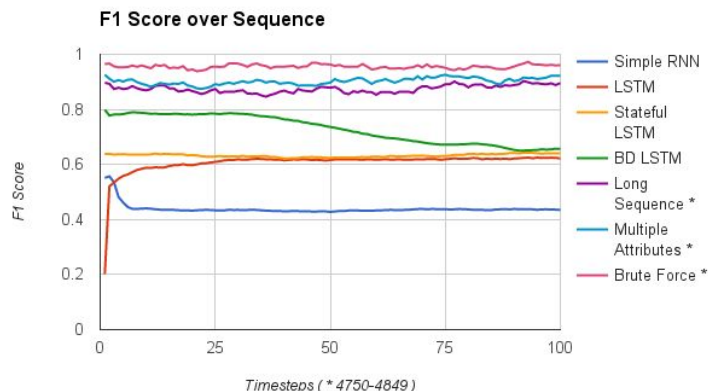
**Brute Force**

Seeing the slight benefit provided by including neighboring wires in evaluating a single timestep, we can try to expand on this with a couple of modifications to the model. We will increase the window of features to include nine wires. We will also double the nodes in our hidden layers to see if there is any unexplored complexity in the wire data. This is a rather brute force approach but is not bad to try after settling on model structure:

```
01  left = Sequential()
02  left.add(LSTM(32, input_shape=(9600,9), return_sequences=True,
               activation='softsign', dropout_W=0.2, dropout_U=0.2))
03  left.add(LSTM(32, return_sequences=True,activation='softsign',
               dropout_W=0.2, dropout_U=0.2))
04  right = Sequential()
05  right.add(LSTM(32, input_shape=(9600,9), return_sequences=True,
               activation='softsign', dropout_W=0.2, dropout_U=0.2,
               go_backwards=True))
06  right.add(LSTM(32, return_sequences=True,activation='softsign',
               dropout_W=0.2, dropout_U=0.2, go_backwards=True))
07  model = Sequential()
08  model.add(Merge([left,right], mode = 'sum'))
09  model.add(TimeDistributed(Dense(2)))
10  model.add(Activation('softmax'))
11  model.compile(loss='binary_crossentropy', optimizer=Nadam())
```



F1 Score over Sequence

Legend: Simple RNN, LSTM, Stateful LSTM, BD LSTM, Long Sequence *, Multiple Attributes *, Brute Force *

Timesteps ( * 4750-4849 )

| | |
|---|---|
| True Positive: | 1397764 |
| False Positive: | 68122 |
| True Negative: | 21451357 |
| False Negative: | 122757 |
| Recall (TPR): | 0.9193 |
| Specificity (TNR): | 0.9968 |
| Precision: | 0.9535 |
| Accuracy: | 0.9917 |
| F1 Score: | 0.9361 |
| AUC: | 0.9899 |
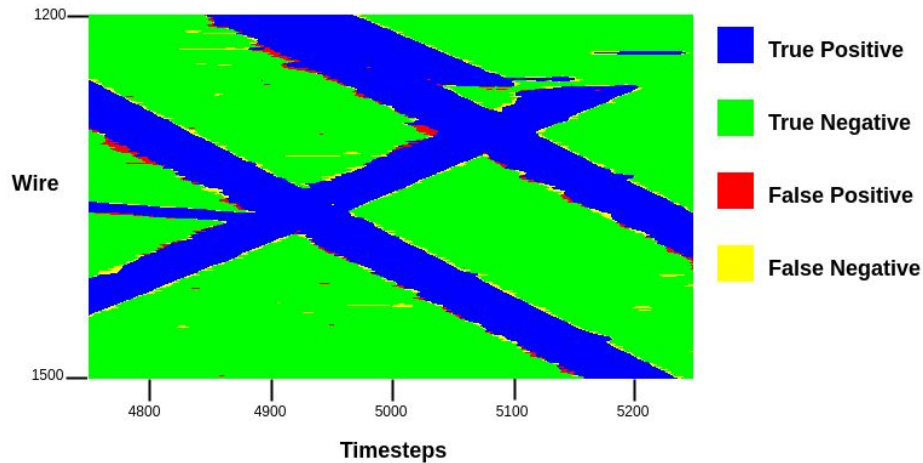
Again, there are modest overall gains in the metrics, though these results are achieved at a cost. The primary difficulty with adding nodes and features is the added computational complexity. These experiments use a data generator and various GPUs, so direct comparisons between models cannot be 100% accurate. However the chart below gives a rough approximation of the growth in resource requirements for each model:

| Experiment | Keras Model Parameters (weights and bias terms) | F1 Score Achieved | AUC Achieved | Approximate Training Time for 10 Epochs (hours) |
|---|---|---|---|---|
| Simple RNN | 850 | .4372 | .8547 | 14 |
| LSTM | 3298 | .6075 | .8760 | 30 |
| Stateful LSTM | 3298 | .6324 | .8864 | 30 |
| Bidirectional LSTM | 6562 | .7284 | .9571 | 40 |
| Long Sequence LSTM | 6562 | .8749 | .9871 | 33 |
| Multiple Attributes | 7074 | .9010 | .9887 | 47 |
| Brute Force | 27458 | .9361 | .9899 | 138 |

The slight gain achieved by the Brute Force experiment over the other models hardly seems worth the extra training time. However, this model not only performs the best overall, the ends of the particle intersection signals which proved troublesome for the other models appear less difficult for this classifier.

**A Faster Model**

In this experiment we will attempt reduce the training time of the Brute Force approach without sacrificing too much of discrimination power it was able to achieve.

Two things that distinguished the Brute Force model from the others was the doubling of hidden nodes in the LSTM layers and the increase of features per timestep. The growth in training time correlates directly with the increase in the model parameters (weights and bias terms) that need to be updated with each round of training. As a compromise, we will cut this down to 20 hidden nodes and use 7 features per timestep, 3 neighboring wires on each side.
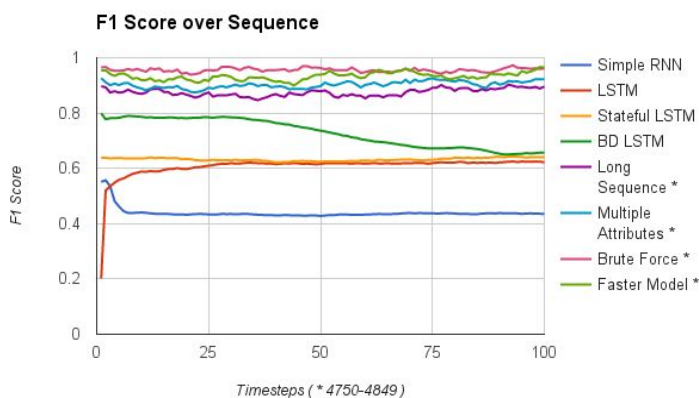
There are also a RNN option that may assist in reducing computational complexity. Gated Recurrent Units (GRUs) are a variation of LSTMs that combine the cell state and hidden state into one internal structure. The output gate is also eliminated, meaning the entire unit output is passed to the next GRU unit instead of being regulated. Despite these simplifications, GRUs have been shown to perform on par with LSTMs (Chung, Gulcehre, Cho, Bengio, 2014). The net result of all these changes is a reduction in the number of calculations per training round. In fact the Keras parameters for this model number 8322 as compared to 27458 for the Brute Force model.

Another obvious contributor to training time is the size of the training data set. So far we've used all of the wire data from four simulated events. It could be there is enough similarity among the simulated waveforms that this training set could be reduced in size without losing the variation that is desireable in training data. We will randomly sample groups of wires from the four events and drastically reduce our training set to approximately ⅓ its previous size.

```
01  left = Sequential()
02  left.add(GRU(20, input_shape=(9600,7), return_sequences=True,
                activation='softsign', dropout_W=0.2, dropout_U=0.2))
03  left.add(GRU(20, return_sequences=True,activation='softsign',
                dropout_W=0.2, dropout_U=0.2))
04  right = Sequential()
05  right.add(GRU(20, input_shape=(9600,7), return_sequences=True,
                activation='softsign', dropout_W=0.2, dropout_U=0.2,
                go_backwards=True))
06  right.add(GRU(20, return_sequences=True,activation='softsign',
                dropout_W=0.2, dropout_U=0.2, go_backwards=True))
07  model = Sequential()
08  model.add(Merge([left,right], mode = 'sum'))
09  model.add(TimeDistributed(Dense(2)))
10  model.add(Activation('softmax'))
11  model.compile(loss='binary_crossentropy', optimizer=Nadam())
```



F1 Score over Sequence

Legend: Simple RNN, LSTM, Stateful LSTM, BD LSTM, Long Sequence *, Multiple Attributes *, Brute Force *, Faster Model *

Timesteps ( * 4750-4849 )

| | |
|---|---:|
| True Positive: | 1371610 |
| False Positive: | 50571 |
| True Negative: | 21468908 |
| False Negative: | 148911 |
| Recall (TPR): | 0.9021 |
| Specificity (TNR): | 0.9976 |
| Precision: | 0.9644 |
| Accuracy: | 0.9913 |
| F1 Score: | 0.9322 |
| AUC: | 0.9899 |
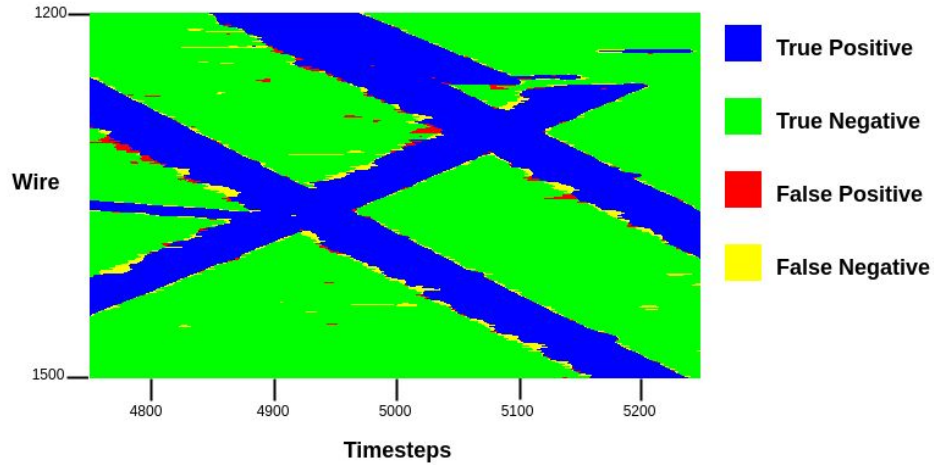
With only 47 hours of training, we appear to achieve relatively favorable scores. If we compare the AUC scores among the models, the Faster Model provides a very good return on our use of computation resources. Also of note is the high score achieved by the Long Sequence LSTM with only 33 hours of training:



**Frequency Noise vs. White Noise Simulations**

Throughout this paper experiment results have been presented using the frequency noise wire sets from the U plane. While all classification tests were performed on both types of background noise simulations, the frequency noise wire sets consistently performed worse than the white noise wiresets. Therefore we chose to present the detailed results using the more challenging frequency noise test data. For comparison, below are some of the metrics for the white noise wire set from the U plane tested using the same trained models:

| Experiment | Frequency Noise, U plane | | White Noise, U plane | |
|---|---|---|---|---|
| | F1 Score | AUC | F1 Score | AUC |
| Simple RNN | .4372 | .8547 | .7536 | .9273 |

| | | | | |
|---|---|---|---|---|
| LSTM | .6075 | .8760 | .7590 | .9379 |
| Stateful LSTM | .6324 | .8864 | .7638 | .9356 |
| Bidirectional LSTM | .7284 | .9571 | .8863 | .9829 |
| Long sequence LSTM | .8749 | .9871 | .9039 | .9888 |
| Multiple Attributes | .9010 | .9887 | .9197 | .9914 |
| Brute Force | .9361 | .9899 | .9486 | .9921 |
| Faster Model (GRU) | .9322 | .9899 | .9467 | .9911 |

What is interesting is that the training data for all models consisted of both types of distinct simulations, yet this did not appear to adversely affect the performance of the models when classifying. A few tests were conducted in which the training and testing data were limited to just frequency or white noise. However no appreciable advantage was noted between the classification performance of models trained on just one type of noise and those trained with mixed data.

**Areas of Further Exploration**
Despite the high scores achieved in overall metrics and the ability of the tested models to identify the larger particle tracks on the wire planes, the smaller particle intersections, which may cover only a few timesteps on a single or a few wires, are largely invisible to the trained classifiers. Different model structures and parameters will need to be tested to effectively classify these more subtle signals in the data.

As more realistic simulation data is made available, it would be useful to train the more effective models on this data and then test against actual detector output. Recurrent neural network performance against real waveforms could be a useful metric in evaluating the tools and methods used to produce simulated sequential data.

**Appendix A: Neural Network Parameters**
The primary focus of this paper has been to explore variations in overall recurrent neural network structure and data structure to improve classification performance on our test data. There are also a multitude of activations, tuning parameters, cost functions, and back propagation algorithms which can affect model performance to varying degrees. Though most traditional and several new model options are implemented in the Keras libraries, they are not fully explored in these experiments. Following are brief descriptions of the parameters used throughout these experiments.
**Number of Layers and Nodes**
The choice of network size (number of layers and nodes) can often be arbitrary. After several informal trainings, a modestly sized network was chosen as it seemed capable of capturing the complexity within this specific dataset and highlighting the results achieved by the different types of RNN structures.

**Inner Activations**
In a typical feed-forward network, each node in a neural network contains an activation function that determines the output of that node based on the sum of the inputs. In recurrent neural networks, specifically LSTMs, the activations serve to control the gates within the LSTM structure. These activations are typically sigmoids. **Softsign** is similar to Tanh but can be better in avoiding saturation, where the network fails to learn, or respond to new inputs. (Glarot, Bengio, 2010)

**Dropout**
Ideally, a trained network should be generalized, meaning it can produce correct output for input it has not yet seen. An overfit network has had its weights so tuned that it only performs well on the training data. **Dropout** is a technique where node connections are randomly ignored (or dropped) at each training epoch. When training is complete all node connections are used during classification. This technique has been shown to help in avoiding overfitting and preserve a network's ability to generalize. (Srivastava, Hinton, Krizhevsy, Sutskever, 2014)

**Output Activation**
**Softmax** is a logistic function often used in probabilistic classification that takes a vector of inputs and returns vector of real number outputs from 0 to 1 that add up to 1. Each class would have an output that represents the probability of belonging to that class. Typically the highest probability represents the decision of the classifier, or a threshold is chosen as a determinant. (Theodoridis, Koutroubmas, 2009, 174)

**Loss (or cost) Function**
The loss function returns a value representing a penalty for incorrect classification. The goal then is to minimize the loss function when training. Mean square error is a common loss function used in neural networks where equal value is placed on the error for each class. **Log loss** (or binary cross entropy in Keras) places different levels of penalty based on confidence. So a wrong classification of low confidence is penalized less where a wrong classification with high confidence is penalized harshly (Rosasco, Caponnetto, Piana, 2003).

**Optimizers**
The optimizer is the backpropagation algorithm used to propagate error through the network and adjust the weights. **Nadam** is the Keras implementation of the RMSprop algorithm (Hinton, 2014), a gradient descent algorithm with an adaptive learning rate, combined with **Nesterov momentum** (Sutskever, Martens, Dahl, Hinton, 2013).

**Appendix B: Using Keras**
In the event the reader is interested in exploring the Keras library further, the following sections provide some brief notes and additional Keras code. For clarity just the minimum code is presented. Full documentation, installation and dependency instructions, additional tuning parameters, default settings, and more examples are available at the developer's website https://keras.io/ .

**Training a Model**
Once the model has been instantiated and compiled, a single line is needed to initiate training:

```
01  model.fit(X, Y, batch_size=96, nb_epoch=10,
              validation_data=(valX,valY), verbose=2)
```

X and Y are array like objects that represent the training data and training labels respectively. Here we have a batch size of 96, which means 96 samples will be handled at a time until the entire training set has been processed. The number of epochs is 10, which is the number of times the full training set will be processed. We have prepared a separate, smaller set of data for validation at each epoch. The data and labels are also array like objects. The verboset=2 setting is a flag for Keras to output to console

simple metrics after each epoch while it is training. The samples or instances in booth data and validation data sets need to be shaped the same as the input shape specified in the model.

**Testing a Model**
The following command will return an array of predictions on the dataset X:

```
01  predictions = model.predict(X, batch_size=96, verbose=0)
```

The prediction probabilities can also be obtained using the following command:

```
01  probabilities = model.predict_proba(X, batch_size=96, verbose=0)
```

**Preparing the Data**
The data and labels that are passed into the model need to match the shape specified in the model instantiation. If we want to train 1000 instances, each with 100 timesteps, and each time step is represented by one feature, the shape of that array is (1000, 100, 1). The input shape for the model covers the sequence length and number of attributes only. So in the model we would specify input_shape = (100, 1).

A few of the loss functions in Keras require the classes for the samples to be submitted in categorical format. For example, if there are three possible classes, classes 1, 2 and 3 are labeled as [1,0,0], [0,1,0] and [0,0,1] respectively. Keras provide a utility to create these labels if needed:

```
01  from keras.utils import np_utils

02  newlabels = np_utils.to_categorical(oldlabels, 3)
```

In the example above we are converting an array of single value labels (oldlabels), where a label can be one of three classes, into an array of categorical labels (newlabels) where each label is represented by a three element vector. The output from a model accepting categorical labels will also be in the form of categorical labels.

**Data Generator in Python**
A python generator was used during these experiments due to the size of the training set. While training is faster when a complete dataset is provided to model.fit, you can sacrifice speed for memory when necessary. A data generator can get data from the external files as needed and pass it to the model:

```
01  def data_generator():
02      continue = True
03      while continue:
04          # code to get data and labels from file
05          # code to shape data and labels into x and y
```

```
06          # set continue flag when needed
07          yield x, y
```

Then when training, place model.fit in a loop that runs until the generator is empty. This can run within another loop of epochs:

```
01  for i in range(number_epochs):
02      for X,Y in data_generator():
03          model.fit(X, Y, batch_size=96, nb_epoch=1)
```

**Stateful RNN**

It is important that during stateful training and testing, the sequences are fed into the network in order. Sample x should be the 100 timesteps that follow sample x-1, and so on, since the last timestep in sample x-1 is the immediate predecessor to the first timestep in sample x.

If we happen to be processing a sequence that is truly the beginning of a larger sequence, and we do not want to pass the state of the last sequence into the network (since it is unrelated to the new sequence) we can reset the state of the network, making it initially stateless for that specific sample:

```
01  model.reset_states()
```

This method is usually invoked on a per batch or per epoch basis as long as larger sequences are mapped to begin at a new batch or epoch. The processing of next sequence after this command will immediately return the network to stateful. This method is usually placed in a loop where each iteration of the loop corresponds to new initial sequences requiring the model state to be reset.

**References**
- Artificial Neural Network. (2016, August 9). In Wikipedia. Retrieved August 9, 2016. https://en.wikipedia.org/wiki/Artificial_neural_network
- Britz, Denny. (2015, September 17). Recurrent Neural Networks Tutorial Part 1 - Introduction to RNNs. Retrieved August 9, 2016. http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/
- Brownlee, Jason. (2016, July 28). Understanding Stateful LSTM Recurrent Neural Networks in Python with Keras. Retrieved August 15, 2016. http://machinelearningmastery.com/understanding-stateful-lstm-recurrent-neural-networks-python-keras/

- Chollet, Francois. (2015). Keras. http://github.com/fchollet/keras
- Chung, Junyoung. Gulcehre, Caglar. Cho, KyungHyun. Bengio, Yoshua. (2014). Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. https://arxiv.org/pdf/1412.3555v1.pdf
- Fawcett, Tom. (2005, December 19). An introduction to ROC analysis. Pattern Recognition Letters, Volume 27, Issue 8, 861-874. http://www.sciencedirect.com/science/article/pii/S016786550500303X
- Glarot, Xavier. Bengio, Yoshua. (2010). Understanding the difficulty of training deep feedforward neural networks. Journal of Machine Learning Research, Volume 9, 249-256. http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf
- Graves, Alex. (2008, July). Supervised Sequence Labeling with Recurrent Neural Networks. Retrieved June 15, 2016. http://www.cs.toronto.edu/~graves/preprint.pdf
- Graves, Alex. Jaitly, Navdeep. Mohamed, Abdel-rahman. (2013) Hybrid Speech Recognition with Deep Bidirectional LSTM. Retrieved August 16, 2016. http://www.cs.toronto.edu/~graves/asru_2013.pdf
- Hinton, Geoffrey (2014, February 11). Neural Networks for Machine Learning, Lecture 6A. Retrieved September 1, 2016. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf
- LISA Lab. (2010). LSTM Networks for Sentiment Analysis. Retrieved August 11, 2016. http://www.deeplearning.net/tutorial/lstm.html#lstm
- Olah, Christopher. (2015, August 27). Understanding LSTM Networks. Retrieved August 11, 2016. http://colah.github.io/posts/2015-08-Understanding-LSTMs/
- Rosasco, L. De Vito, E. Caponnetto, A. Piana, M. (2003, September 30) Are Loss Functions All the Same? Retrieved September 1, 2016. http://web.mit.edu/lrosasco/www/publications/loss.pdf
- Srivastava, Nitish. Hinton, Geoffrey. Krizhevsy, Alex. Sutskever, Ilya. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research, Volume 15, 1929-1958. https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf
- Sutskever, Ilya. Martens, James. Dahl, George. Hinton, Geoffrey. (2013) On the importance of initialization and momentum in deep learning. JMLR: W&CP volume 28. http://www.jmlr.org/proceedings/papers/v28/sutskever13.pdf