Optimizing Recurrent Neural Networks for Financial Time-Series Predictions

By Aaron Ronald Lux

Abstract:
Recurrent Neural Networks (RNNs) have become the state-of-the-art choice for sequence learning by extracting patterns from sequential datasets (financial time-series, spatial, text, sensor time-series, healthcare, etc). Here we introduce a Recurrent Neural Network machine learning model called Long Short-Term Memory (LSTM)[1][2], which consists of an LSTM cell that processes multiple sequences of an input at a time and computes probabilities of the possible values for the next output in the sequence[3]. However, current RNN models require advanced maintenance to minimize cross entropy loss and increase accuracy. In this paper we will make changes to a TensorFlow RNN LSTM model and monitor cross entropy loss and increase accuracy using TensorBoard.

RNNs have multiple moving parts which can cause issues during training resulting in poor accuracy. Debugging problems in a RNN often start with investigating how the network is changing after every iteration. Each weight is constantly changing as the network attempts to learn the most accurate set of weights to use based on the train method. These weights activate over a task and the output of the activation (feature map) is random as well. Over many iterations, the network becomes more stable as the weights are adjusted to fit the training feedback. As the network converges, the network resembles distinct small patterns which can be found in the input data set.

Introduction:
Interest in recurrent neural networks (RNNs) has greatly increased in recent years, since larger datasets, increasing computing resources, and better training algorithms have enabled breakthroughs in both processing and predictive modeling of sequential datasets. RNNs model predictions well on a variety of sequential tasks in many domains. Applications include speech recognition, natural language processing, and attention-based models for structured prediction, financial time-series predictions, and predictions of oil & gas reservoirs locations from moving sensors mounted on ocean-liners. In all of these applications, the order of observations or sequence is important.

In this paper we are using a financial time-series prediction which is a sequence generation task. Sequence generation has a single starting point (I.E. class label) that we want to generate sequences from. To generate sequences, we feed the output back into the network as next input. This may cause the actual output to different from the neural network generated output. This is because the network is generating an output a distribution over all classes but we only choose the most likely one. This is the opposite of Sequence Classification. In Sequence

---

1 cell = tf.nn.rnn_cell.LSTMCell(cellsize) Long short-term memory unit (LSTM) recurrent network cell with 3 gates (input gate, output gate, forget gate). The LSTMCell class can be used as a drop-in replacement for BasicRNNCell but also provides some additional switches. Despite its name LSTMCell represents a whole LSTM layer which can be connected to other network layers to form larger architectures. https://www.tensorflow.org/versions/r0.11/api_docs/python/rnn_cell/rnn_cells_for_use_with_tensorflow_s_core_rnn_methods#LSTMCell

2 [In 1997 MIT rejected a research paper on LSTM. By 2015, Google announced it had managed to improve the error rate of its voice recognition software by almost 50% using LSTM. It is the system that powers Amazon's Alexa, and Apple announced last year that is using LSTM to improve the iPhone. The inventor of LSTM, Jürgen Schmidhuber, has worked at Google Deepmind and has recently founded a new company called nnaisense.]

3 Artificial neural networks are remarkably adept at pattern recognition, sequence learning and reinforcement learning, but are limited in their ability to represent patterns which occur over varying timescales. We solved this problem in this paper by using a dataloader which loads variable sequence length timescales using a multi-sequences loader. A multi-sequence dataloader was found to significantly improve accuracy (For a comparison of multi-sequence and sequence loader see https://github.com/Element-Research/dataload).

Classification we encode the sequence into a dense vector to predict a class. In Sequence Generation we decode a dense vector back into a sequence.

Recurrent Neural Networks (RNN) are a family of networks that specifically model sequence. RNNs build on the same neurons summing up weighted inputs from other neurons. However, neurons are allowed to form cycles by connecting forward to higher layers and backward to lower layers. RNNs are attractive because they equipping neural networks with memories, and the introduction of gating units such as LSTM and gated recurrent units (GRU)[4] have greatly increased prediction accuracy from these networks compared to older prediction methods such as Markov models. The hidden activations of the network are remembered between inputs of the same sequence. In contrast to LSTM, GRU has a simpler architecture and requires less computation while yielding very similar results. GRU has no output gate and combines the input and forget gates into a single update gate.

In this paper we will focus on LSTMs. LSTMs correct the problem of vanishing and exploding gradients which is more prominent in RNNs than in forward networks. They work significantly better for learning long-term dependencies and have become a standard for RNNs. To correct the vanishing and exploding gradients problem the LSTM architecture replaces the normal neurons in an RNN with so-called cells that have a little memory inside. Those cells are wired together as they are in a usual RNN but they have an internal state that helps to remember errors over many time steps.

The state of an RNN depends on the current input and the previous state, which depends on the input and state before that. Therefore, the state has indirect access to all previous inputs of the sequence and can be interpreted as a working memory. The key to LSTMs is that this internal state has a self-connection with a fixed weight of one and a linear activation function, so that its local derivative is always one. During backpropagation, this so called constant error carousel can carry errors over many time steps without having the gradient vanish or explode.

While the purpose of the internal state is to deliver errors over many time steps, the LSTM architecture leaves learning to the surrounding gates that have non-linear, usually sigmoid, activation functions. Linear activation functions do not improve the performance because LSTMs were designed with a gating scheme in mind to better deal with vanishing/exploding gradients. In the original LSTM cell, there are two gates: One learns to scale the incoming activation and one learns to scale the outgoing activation. The cell can thus learn when to incorporate or ignore new inputs and when to release the feature it represents to other cells. The input to a cell is fed into all gates using individual weights. The sigmoid activation works like a percentage which determines how many percent of the input signal should be stored in a cell. It doesn't make sense to amplify a signal and write 110% of the current cell signal to the output as is the case with RELU activation.

RNNs are similar to forward networks in that they are optimized using gradient descent. To optimize an RNN the same way as forward networks it must first be unfolded in time (unrolled). We then apply standard backpropagation through this unrolled RNN to compute the gradient of the error with respect to the weights. This algorithm is called Back-Propagation Through Time (BPTT). It will return a derivative for each weight in time, including those that are tied together.

A popular extension to LSTM is to add a forget gate scaling the internal recurrent connection, allowing the network to learn to forget. In TensorFlow the forget gate bias values are initialized by specifying the forget_bias parameter to the LSTM layer. The default is the value one and usually its best to leave it that way.

4 cell = tf.nn.rnn_cell.GRUCell(hidden_size) Gated recurrent unit (GRU) - same as lstm but with 2 gates insead of 3 (reset gate, update gate). Faster to compute than LSTM. class tf.nn.rnn.GRUCell(cellsize). The TensorFlow GRU layer is called GRUCell and have no parameters other than the number of cells in the layer. https://www.tensorflow.org/versions/r0.11/api_docs/python/rnn_cell/rnn_cells_for_use_with_tensorflow_s_core_rnn_methods#GRUCell

Another extension to LSTM are peephole connections, which allows the gates to look at the cell state. In Tensorflow peephole connections can be activated by passing the use_peepholes=True flag to the LSTM layer.

Related work:
Research into newer RNNs such as Neural Turing Machines (NTM) and differentiable neural computers (DNC) have proven these models increase prediction accuracy by another order of magnitude over LSTM and GRU. The source of this increase of accuracy was from adding attention to the memory in these new neural networks.

Model Description:

LSTM cell layers are used to process a financial time-series problem. We will make changes to this model monitor cross entropy loss and increase accuracy using TensorBoard.

Results and Discussion:

We will analyze Accruacy and cross entropy loss in TensorBoard. results from with 4 common problems when training RNNs LSTMs

1. RNN LSTM Vanishing Gradients. Can be corrected in TensorFlow by trying different activation units. LSTM architecture leaves learning to the surrounding gates that have non-linear, usually sigmoid, activation functions. In these cases you will not use linear activation functions like relu.

- tf.nn.relu
- tf.nn.relu6
- tf.nn.crelu
- tf.nn.elu
- tf.nn.softplus
- tf.nn.softsign
- tf.nn.bias_add
- tf.sigmoid
- tf.tanh

2. RNN LSTM Exploding Gradients can be corrected in TensorFlow by cliping gradients with tf.clip_by_global_norm

```
# take gradients of cost during training
gradients = tf.gradients(cost, tf.trainable_variables())

# clip the gradients by a predefined max norm
clipped_gradients, _ =  tf.clip_by_global_norm(gradients, max_grad_norm)

# add the clipped gradients to the optimizer
optimizer = tf.train.AdamOptimizer(learning_rate)
train_op = optimizer.apply_gradients(zip(gradients, trainables))
```

3. RNN LSTM annealing the learning rate. While training any deep learning network, it is usually helpful to anneal the learning rate over time. Good intuition to have in mind is that with a high learning rate, the system contains too much kinetic energy and the parameter vector bounces around chaotically, unable to settle down into deeper, but narrower parts of the loss function. Knowing when to decay the learning rate can be tricky: Decay it slowly and you'll be wasting computation bouncing around chaotically with little improvement for a

long time. But decay it too aggressively and the system will cool too quickly, unable to reach the best position it can. We will adjust TensorFlow Optimizers to anneal the learning rate:

```
# Optimizer accept both scalars and tensors as learning rate
learning_rate = tf.train.exponential_decay(init_lr, global_step, decay_steps, decay_rate, staircase=True)
optimizer = tf.train.AdamOptimizer(learning_rate)
```

4. RNN LSTM Overfitting. We will add dropouts[5] during training to increase noise and therefore prevent overfitting. Do not add dropouts outside of training or problems will occur. For example adding dropouts during testing will cause the model to report impossibly high accuracies.

Use dropout through tf.nn.dropout or DropoutWrapper for cells

tf.nn.dropout

```
hidden_layer = tf.nn.dropout(hidden_layer, keep_prob)
```

● DropoutWrapper

```
cell = tf.nn.rnn_cell.GRUCell(hidden_size)
cell = tf.nn.rnn_cell.DropoutWrapper(cell,
output_keep_prob=keep_prob)
```

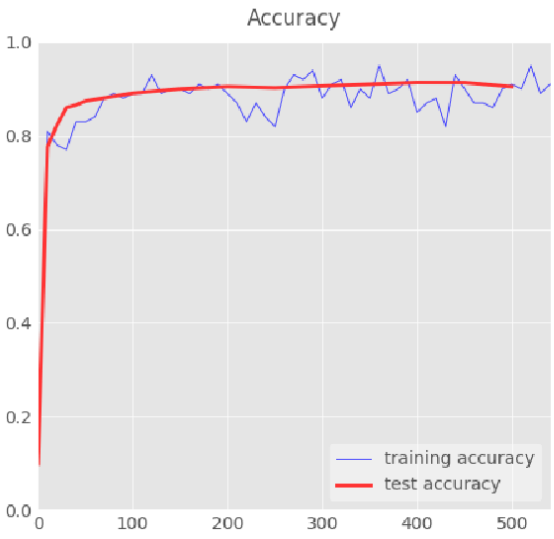
Results and Discussion:

Test#1 Results:

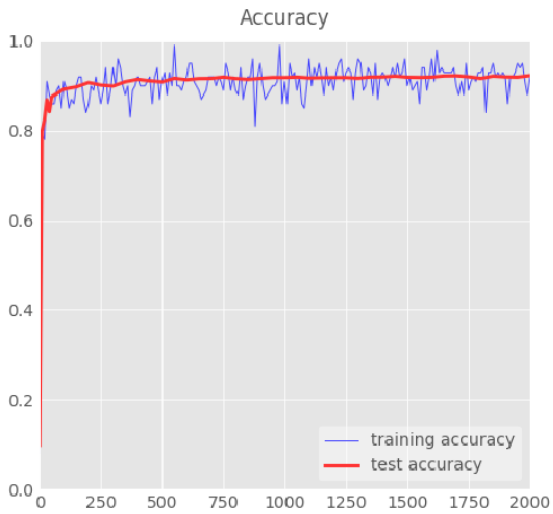
Results of RNN LSTM using sigmoid activation function. All sigmoid biases are initialized at 0.

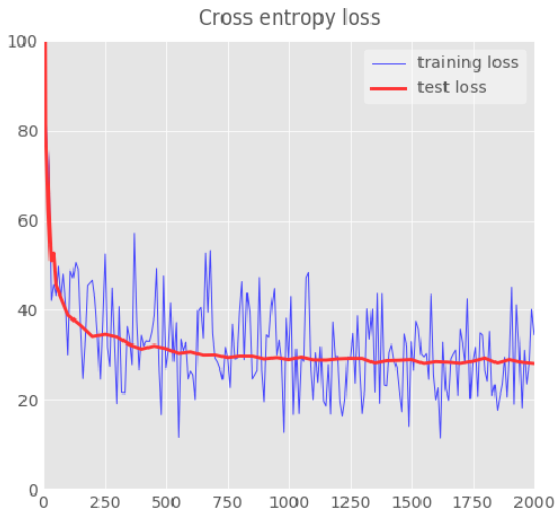
final max test accuracy = 0.91 (5K iterations). Accuracy peaked above 0.90 in the first 200 iterations.

Very noisy cross entropy loss.



5 http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf





Test#2 Results:

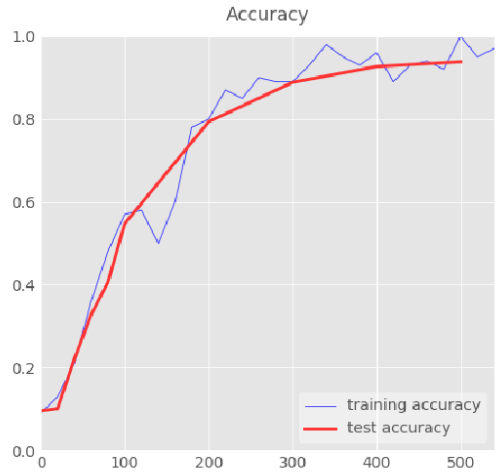
Results of RNN LSTM using cliping gradients with tf.clip_by_global_norm

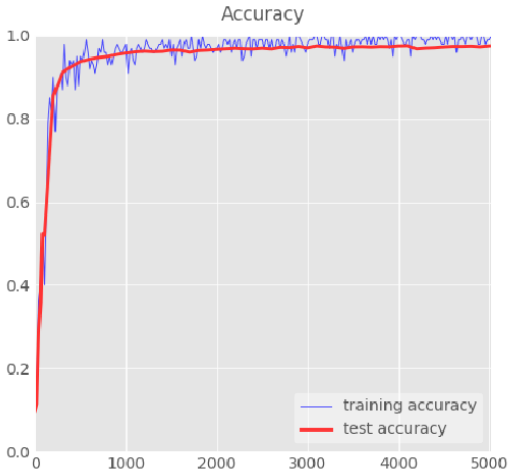
Slow start, training cross-entropy not stabilized in the end

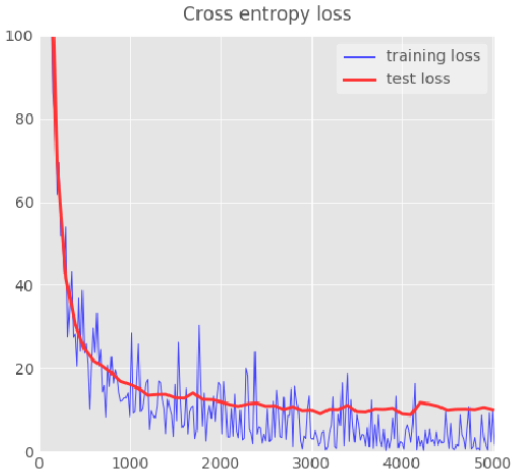
Cross entropy loss noise has improved from Test#1.

Cross entropy loss is lower than Test #1

Accuracy peaked after 1000 iterations. Final test accuracy was 0.97





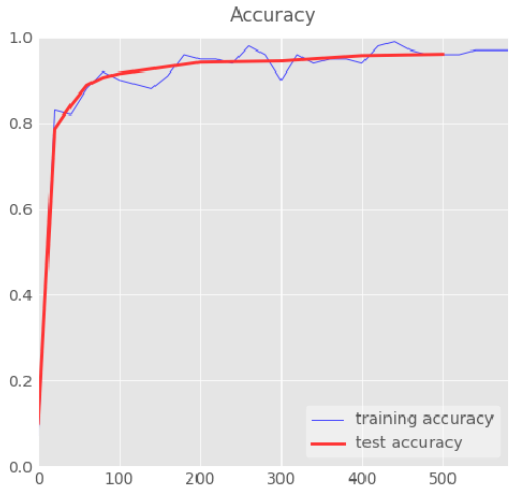


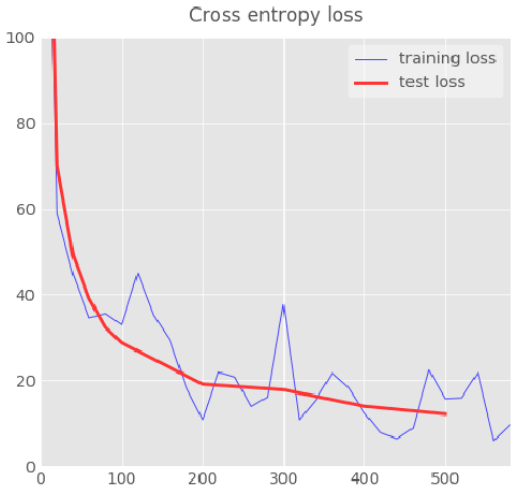
Test#3 Results:

RNN LSTM annealing the learning rate.

#Accuracy above 0.97 in the first 1500 iterations but noisy curves.

training cross-entropy down to 0

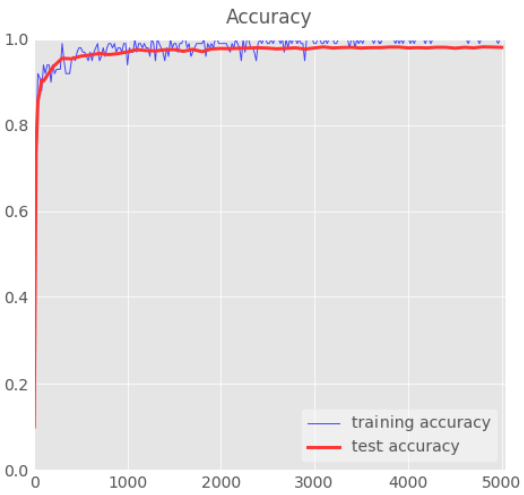


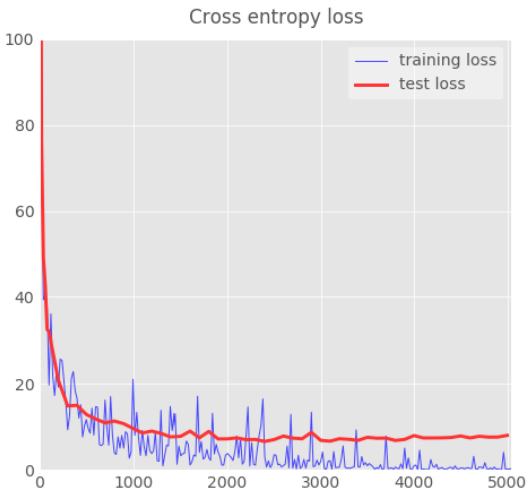


Training cross-entropy still a bit noisy

Test cross-entropy stable

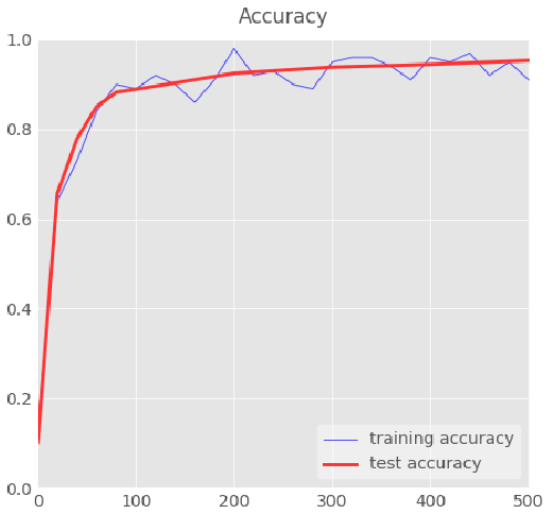
test accuracy stable just under 98

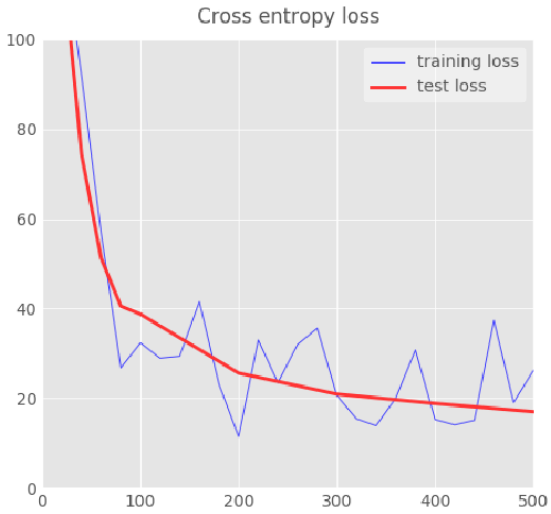


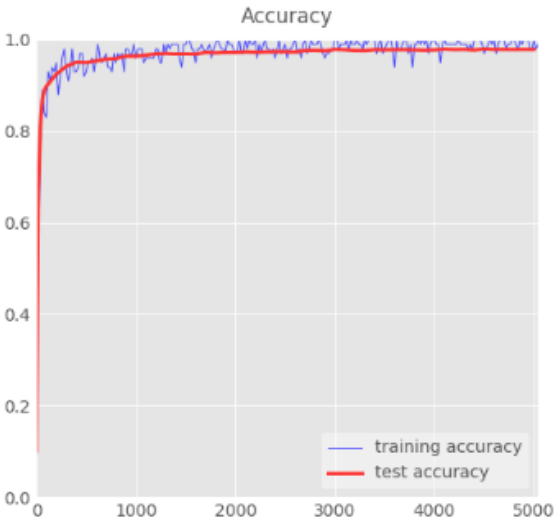


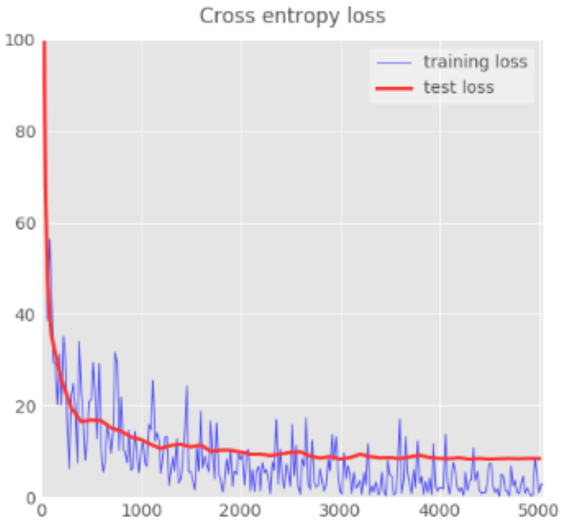
Test#4 Results:
RNN LSTM Overfitting. We will add dropouts during training to increase noise and therefore prevent overfitting.
training set fully learned, test accuracy stable

#Accuracy above 0.98 in the first 2000 iterations but noisy curves.
Training cross-entropy still a bit noisy
Test cross-entropy stable
test accuracy stable just under 98









Conclusion:

RNN LSTM are powerful sequential predictive models that are applicable to a wide range of problems and are responsible for state-of-the-art results. We learned how to optimize RNN LSTMs, what problems arise doing so, and how architectures like LSTM and GRU help to overcome them. Using these building blocks, we solved a financial time-series task with several hidden states.

Appendix:

Cross Entropy Loss:

loss function: cross-entropy = - sum(Y_i * log(Yi))

Y: the computed output vector

Y_: the desired output vector

cross-entropy

log takes the log of each element, * multiplies the tensors element by element

reduce_mean will add all the components in the tensor.
We end up with the total cross-entropy for all inputs in the batch

Accuracy:
accuracy of the trained model, between 0 (worst) and 1 (best)

Batch Size:

```
def predict_sequences_multiple(model, data, window_size, prediction_len):
    #Predict sequence of 50 steps before shifting prediction run forward by 50 steps
    prediction_seqs = []
    for i in range(int(len(data)/prediction_len)):
        curr_frame = data[i*prediction_len]
        predicted = []
        for j in range(prediction_len):
            predicted.append(model.predict(curr_frame[newaxis,:,:])[0,0])
            curr_frame = curr_frame[1:]
            curr_frame = np.insert(curr_frame, [window_size-1], predicted[-1], axis=0)
        prediction_seqs.append(predicted)
    return prediction_seqs
```

Neural Network **Activation** Functions
The model is:
Y = softmax(X * W + b)
X: matrix
W: weight matrix
b: bias vector
+: add with broadcasting: adds the vector to each line of the matrix (numpy)
softmax(matrix) applies softmax on each line
softmax(line) applies an exp to each value then divides by the norm of the resulting line
Y: output matrix with 100 lines and 10 columns

```
model.add(Activation("sigmoid")) #keras activation functions https://keras.io/activations/

model.add(Activation("relu"))
```

Misc

RNNs LSTMs appear to automate feature engineering (using feature learning) and parameter optimization (using gradient-based optimization). The machine learning benefits derived from this automation (E.G. reduced latency of calculations, proper handling of anomalies, increased inefficiency, etc), along with the elimination of predictive bias and variability, may cause the elimination of certain jobs including quantitative analysts, actuaries, and statisticians.