Aaron Jacobson

An RNN that Distinguishes Languages

Introduction

The recurrent neural network (RNN) model presented in this paper determines which language a word is in. Words are fed into the network character-by-character at a time and the output of the network results from the combination of all inputs over "time."

This RNN model is strictly different from a standard hidden-layer neural network receiving a vector of characters as input, for example. Each letter is only viewed once, in order, to determine the output. This is a "naïve" approach to recurrent learning, but is an excellent beginning.

The problem model can be seen as a small but useful testbed for more complex problems of this nature: the inputs are relatively short (no more than 12-deep); the problem, both input and output, is easily understood without domain knowledge; the size of both the one-hot input vectors (for characters) and one-hot output vectors (for languages) are relatively small; and there are hundreds of thousands of inputs available *for each language*, making testing and validation easier.

A well-trained algorithm has immediate usefulness: it's a necessary first step in auto-translation or other tools which rely on language detection.

Running the Code

The code which accompanies this paper requires Matlab R2016_b and Python 3.5. It contains four parts: a normalizer, in Python; 39 cleaned and normalized corpora for training purposes (and 36 un-normalized corpora), as text files; a corpus manager, in Matlab; and a recurrent neural network, in Matlab, specialized to learn on this input.

Finding a Corpus

Finding a corpus is relatively easy. It's cleaning it up that's difficult. I used corpora of the top 5000 words from E-Z-Glot. These were selected by criteria:

1. Written in the Latin script
2. More than 1000 distinct words
3. Easily cleanable

I screen-scraped them from each page with a snippet of JavaScript in the console:

```
$$('.topwords li span').reduce(function (acc, x, i) {
    return acc + ' ' + x.innerHTML;
}, '');
```

In the end, I had 36 distinct corpora containing 107,116 words.

Other corpora are available online or as text files, containing many more words per language, such as dictionaries. It pays to be careful when determining a corpus for languages you don't

speak, however. English is *not* a highly-conjugated or modal language, but other languages have, e.g., suffixes/compound words, etc., that, while well-represented in use, will not appear at all in dictionaries. I was able to avoid this problem with the E-Z-Glot corpora, because they were themselves scraped from Wikipedia.

Cleaning Up Corpora

Each corpus was first cleaned by hand, using text-editor tools, to separate words by newlines, remove obvious punctuation, numbers, *emojibake*, etc. I also removed one-letter words (and probably should have removed two-letter words as well.)

Regular expressions such as `[\x20-\xFF]+` do well to surface most typos. If the intent was clear, I repaired the word, if not, I removed it. Following that, each corpus was regularized with a very small Python program, **normalize.py**. To use this in Bash to normalize an entire directory at once:

```
for file in Corpora/*; do python normalize.py $(basename "$file"); done.
```

Combining all corpora into a single language-indexed corpus is handled within Matlab, by **Corpora.m**, and is described below.

Attached to this paper are three groups of corpora. The files in **Corpora/** contained un-normalized text, as source material. The files in **Normalized Corpora/** contain the cleaned and normalized corpora. Finally, the three files in **Mini Corpora/** contain 500-word selections from the English, German, and Spanish normalized corpora, and are useful for small experiments.

Combining Corpora

This is handled pretty simply. When creating a new Corpora object, pass the name of the folder as an argument. For example, to create a corpus from **Normalized Corpora/**,

```
feature('DefaultCharacterSet', 'UTF8'); % this _must_ be set correctly
c = Corpora('Normalized Corpora');
```

`c` now contains three properties: `c.corpus` is a dictionary of words to languages. `c.allChars` is a string containing all characters used in every language. `c.languages` is a list of all the languages, based on the filenames. The last two are used in the creation of one-hot vectors for the input and output layers of the neural network. If you get odd results on OS X, make sure that there's no **.DS_Store** or other hidden files in your corpora directory. Note, too, that the default character set *must* be UTF-8, as set above. This must be set on the Matlab REPL.

Running an RNN

Now that we have a corpus set up, let's set up an RNN:

```
r = RNN(W, U, b, V, c); % with previously trained weights
r.corpus = c; we still need to link a corpus
```

And we can run this on an input.

```
apple = r.feedforward({'apple'});
```

```
r.best_matches(apple, 3) % returns the 3 best guesses for 'apple'
```

Training the RNN

```
r = RNN(); % no parameters
r.corpus = c; each RNN has its own corpus to work on
```

The hidden layer of this recurrent neural network still needs to be made, and then we can run it on input.

```
r.initialize_weights(2000); % 2000 nodes in the single hidden layer
```

This creates weights to the hidden layer from the input layer (**U**), to the output layer from the hidden layer (**V**), between each generation of hidden layers, (**W**), and biases at each node (**b**, **c**). When choosing a size for the hidden layer, remember that the hidden-layer-to-hidden-layer weight is a square matrix: it gets big pretty fast.

Let's train this:

```
r.train(200, 1, .01);
```

will train for 200 generations, with a mini-batch size of 1 (on-line), and a learning rate (η) of 0.01.

Results

Results from training can be got from **r.training_stats**

```
plot(mini_r.training_stats);
```

and results from different runs can be combined through standard Matlab matrix-handling functions, which I won't cover here.

Options and Helper Functions

The pros and cons of available options are discussed below, but there are many ways available to specialize your RNN included with the code. In general, cost, activation, and regularization functions are set with `r.cost_function_prime = @RNN.MSE_prime;`, etc. Note that these are class methods, hence independent of *your* RNN, its weights, etc..

Activation Functions

- `tanh`. The default.
- `sigmoid`
- `softmax`. This uses `tanh` in non-output layers
- `ReLU`

Cost Functions

- `log_likelihood`. The default.
- `MSE`

- `cross_entropy`
- `cosine_similarity`

Regularization Functions

- `none`. The default
- `L1`
- `L2`

Helper Functions

- `one_hot(vector, elems)` creates a one-hot matrix of length **vector** from the vectorized input and the total number of elements in the one-hot.
- `max_n(a, n)` returns a sorted vector of the **n** largest elements of **a**.
- `split_input(arr, part_a, part_b, part_c)` separates an input matrix into three parts of size **part_a, part_b, part_c**, for training, testing and validation.
- `shuffle(cell_arr)` shuffles a cell array.

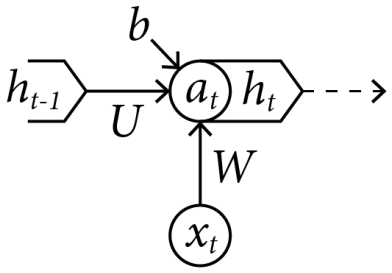
The System

The Corpus

The corpus, in **Corpora.md** handles loading and parsing of corpora, and encoding/decoding of words. Because an individual corpus is so tightly intertwined with its RNN, it's more useful to consider this a part of the RNN than as an independent and modifiable unit.

From here on out, then, the corpus should be thought of as a list of pairs: the first member is the word, encoded as a vector of one-hot vectors; the second member is the language it belongs in, encoded as a one-hot vector. We'll continue to refer to these as "words," "characters," "languages," etc., but remember that they are seen by the RNN as these tuples.

Feedforward



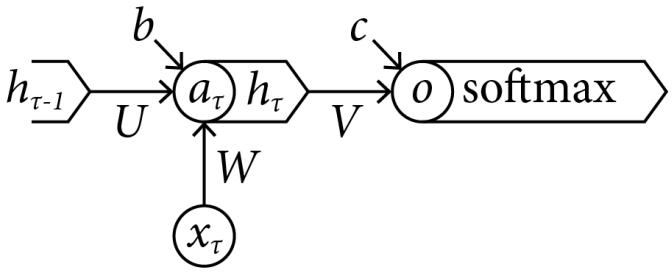
$$a_t = Uh_{t-1} + Wx_t + b$$

$$h_t = \sigma(a_t)$$

This particular version of an RNN is *almost* identical to a single hidden-layer neural network. The only difference is that each node receives input not just from the layer below it, but additionally from the previous state of the hidden layer. This is made clear by the feedforward equations shown in the diagram.

There is one additional term, Uh_{t-1}; this represents the previous state (or knowledge) of the network. a_t is the weighted input at timestep t, and h_t is the activation of the neuron at t.

At the output layer of the RNN, at timestep τ, we feed forward through a last, independent set of weights and use a softmax activation function to promote the "hottest" element of the output vector.



$$o = Uh_\tau + Wx_\tau + c$$

$$\hat{y} = \text{softmax}(o)$$

The Cost Function

The choice of cost function is the most important design decision in this model.

MSE is Horrible

Mean-squared error works incredibly poorly for this. The distance between two one-hot vectors is either 0 or $\sqrt{2}$. Averaging across the rank of these vectors leads to a fractional cost which has little correlation to the percentage of errors actually present. In training, we're compare a one-hot and an approximation of it, not two one-hots, but the MSE still ends up at approximates $\sqrt{||\text{output}||}$, no matter how good the results are.

Adding *any* sort of regularization to this reduces correctness even more: as the overall weights in the network decline, the chance of *any* output rounding up to 1 decreases.

The cosine-similarity cost function doesn't work for the same reasons; the *angle* between any two distinct one-hot vectors is exactly the same.

Log-Likelihood is Best (But Not Good)

Although log-likelihood is the best cost function for one-hot outputs, it's still not great at distinguishing between results that are incorrect proportionally, and those that are totally different. The immediate counterargument to this complaint is that this is what training is for, although with large output vectors ($n > 150$, in this application), the learning proceeds *very* slowly.

Backpropagation

Backpropagation over the last set of weights is the same as in a hidden-layer neural network, so I won't discuss it in detail. It's performed by applying the chain rule to the cost function and pushing back through to the hidden layer.

Backpropagation of the error from generation to generation of hidden layers is done in regards to the changes from both the current generation's input layer and the previous generation's hidden layer. Mathematically, it, too, is an application of the chain rule.

$$\partial_t = W'\partial_{t+1} + Ux_t \odot \sigma(a_t)$$

The total error for each U, W, and b is the average over all gradients in every timestep.

$$\partial U = \eta \sum_{t \in \tau} \partial_t x_t$$

$$\partial W = \eta \sum_{t \in \tau} \partial_t a_{t\text{-}1}$$

$$\partial b = \eta \sum_{t \in \tau} \partial_t$$

Deep backpropagation like this can lead to exploding or disappearing gradients. (This is something that I struggled with throughout the entire project.) There are two immediate solutions: the first is to limit backpropagation; the second is to clip the deltas. (Tweaking η doesn't work: exploding/disappearing gradients grow at superlinear rates.) Both of these are poor solutions. Limiting backpropagation reduces the abilities of the network to use previous solutions, which leads to overfitting on the last character(s) of the input string. It's a very blunt instrument. In my experiments on this model, gradient explosions would occur only in a few elements of the weight matrices, but limiting backprop prevents all of them from learning.

Clipping is another very blunt instrument, which limits the deltas to the range [$-clip$, $clip$]. Choosing too-small of a range weakens the model unnecessarily. A combination of limiting backprop and clipping works best.

Both of these approaches, as it turns out, are commonplace in RNN's, and have been since the 90's. They are among the many reasons that RNN's are so difficult to train. However, using them means the RNN *will work*, albeit poorly, and thus they become yet another pair of hyperparameters to tweak.

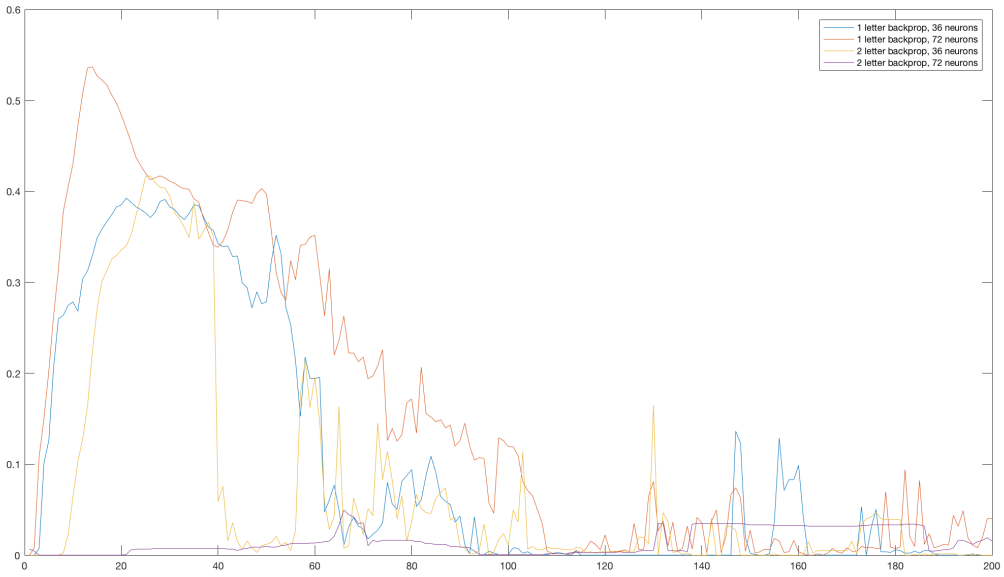
Batch Size

Typically, mini-batches of variable-length inputs are zero-padded at the end to fill vectors of the max set size. I wasn't able to implement mini-batching in my code, but I think that I would need to do the opposite: pad zeros on the *beginning* of the input, so that I could back-propagate over meaningful input and still receive useful gradients to update on.

Hyperparameters

In a system this complex, the choice of hyperparameters matters a great deal. One conclusion that became clear in testing was that the high-dimensional space that this model was exploring was full of sharp nonlinearities. In many training sessions, the model would learn nothing for the first twenty epochs, then jump to 20 or 30 percent correctness at once. Sudden shifts of correctness, both up and down, were commonplace in training, even after hundreds of epochs of relatively linear change.

The most important hyperparameter that affected training was the learning rate. A tiny learning rate was best able to maneuver to a lower cost. (This, of course, depends a great deal on the cost function.) The next most important hyperparameters were the clipping bounds and the depth of backpropagation. Clipping works best when set as high as possible, and I haven't been able to fully explore that. Limiting backpropagation depth works best when set to only 1 or 2 generations. I don't know why this is, at all, and find it very surprising. Another major factor was the size of the hidden layer. It worked best when set to what I would consider a very small size: a small multiple of the input vector.



Conclusion

In general, a naive approach to recurrency such as this doesn't work very well. My best results came on training a mini-corpus of 500 words from three languages each, and never rose above 60% correctness. Ironically, *smaller* batches, *fewer* hidden nodes, and *less* backpropagation through time do better. Even with the best hyperparameters I could discover, there was still a great deal of "jitter" in the results, and almost all training sessions eventually declined to 0% correct, although this can be allayed to an extent with a very small learning rate.

Further, training on a large corpus, such as my 110,000-word one, takes minutes per epoch. A big disappointment in my model is that hyperparameters are still sensitive to the size of the dataset, and I haven't been able to tell what the relationship is (one-hot size?).

The poorness of these training results did not allow me to study the structure of a working RNN, or perform meaningful tests on made-up words, as I proposed at the beginning.

I've learned a great deal from this. Hyperparameter tweaking is definitely an art form, one which is handwaved in introductions to machine learning, but becomes immediately apparent in the creation of one's own ML routine, and that's something I wouldn't have learned without writing this. In addition, because this version of an RNN is a less-popular shape for recurrent neural nets, I had to derive the equations for backpropagation and updating on my own, and although this is "just calculus," I did it, and it *worked*.

I spent the vast majority of my time on this project fixing what turned out to be an exploding gradients problem, and not a bug, and learned a lot about Matlab, debugging code, and working in a REPL alongside a separately editable program (which Matlab handles extraordinarily well).

Ideas for enhancement

Here are five quick ideas:

1. Input vectors as matrices: The inputs are currently variable-length cells of one-hot vectors. The *corpus* which contains them should handle conversion to matrix form and all the associated complications (dynamic lengths, dynamic charset-size, etc.). Having input as matrices means true mini-batching is possible; it probably means a drastic speedup as well.
2. Better outputs, better output methods: There's one helper method which formats results pretty well, but as soon as this can be trained to higher accuracy, I'll need better and more useful I/O routines.
3. Hyperparameters need to scale with input size: This will make training larger corpora possible. Hopefully, this RNN is just simple enough to allow for experimentation of this sort. I do realize that this is not unique to my version, but I can explore it here.
4. Refactor the code: Even at 600 lines, there's plenty to refactor. One thing I definitely need to do is specialize my error-handling.
5. Add non-Latin texts: Testing if a word's in Korean should be a slam dunk. No other language uses that alphabet! Adding corpora for Cyrillic languages will be trickier, because the languages are generally related by more than script, but should be possible, even if training slows down.

Bibliography

Karpathy, Andrej. char-rnn (2015). https://github.com/karpathy/char-rnn.

An excellent, very small reference model of an RNN written in Torch/Lua.

Juraj Koščák, Rudolf Jakša, Peter Sinčák. Stochastic Weight Update in Neural Networks (2015). http://neuron.tuke.sk/koscak/documents/publications/Stochastic_Weight_Update_In_Neural_Networks.pdf.

An overview of RNNs in theory and practice. Most of my other references handwaved equations for backpropagation and updating weights; this contains them.

Sutskever, I., Martens, J., and Hinton, G. E. (2011). Generating text with recurrent neural networks. In ICML'2011, pages 1017–1024.

A primary text for the field. Contains a nice overview and history of RNNs while making clear their promises and difficulties.

Gillick, D., Brunk, C., Vinyals, O., and Subramanya, A. (2015). Multilingual language processing from bytes. arXiv preprint arXiv:1512.00103.

This research also uses character-by-character inputs for the same puropse of reducing input size and preprocessing of data.

Bengio, Y., Ducharme, R., and Vincent, P. (2001). A neural probabilistic language model. In T. K. Leen, T. G. Dietterich, and V. Tresp, editors, NIPS'2000 , pages 932–938. MIT Press.

A good background paper as well, which contains specific recommendations for model choices and hyperparameters for RNNs.

Bengio, Y., Simard, P., and Frasconi, P. (1994) Learning long-term dependencies with gradient descent is difficult. IEEE Transactions on Neural Networks, 5(2):157–166, 1994.