WILDML

AI, DEEP LEARNING, NLP

MENU

RECURRENT NEURAL NETWORKS TUTORIAL, PART 1 – INTRODUCTION TO RNNS

September 17, 2015

Recurrent Neural Networks (RNNs) are popular models that have shown great promise in many NLP tasks. But despite their recent popularity I've only found a limited number of resources that throughly explain how RNNs work, and how to implement them. That's what this tutorial is about. It's a multi-part series in which I'm planning to cover the following:

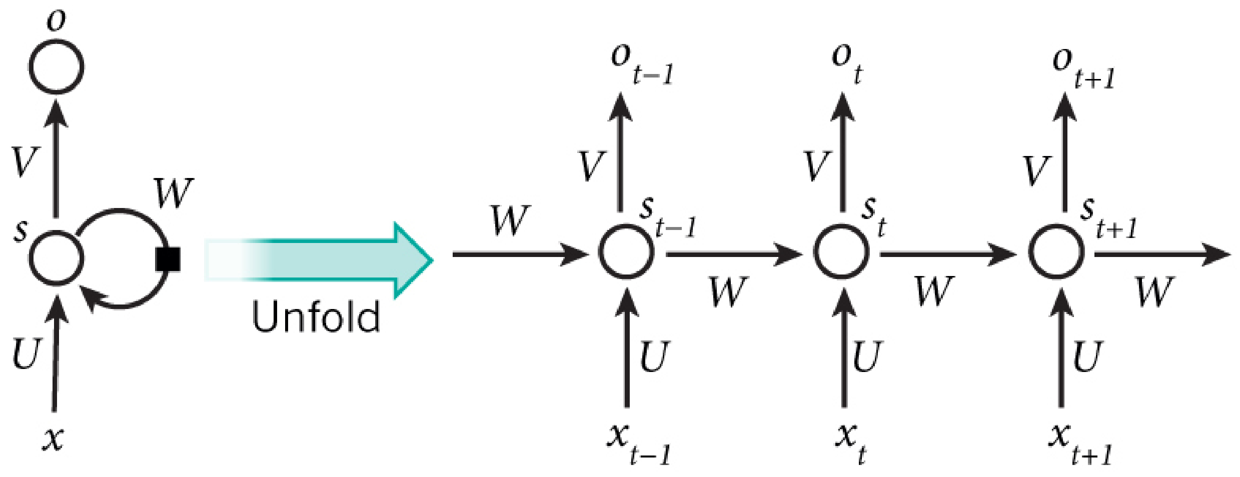
1. Introduction to RNNs (this post)
2. Implementing a RNN using Python and Theano
3. Understanding the Backpropagation Through Time (BPTT) algorithm and the vanishing gradient problem
4. Implementing a GRU/LSTM RNN

As part of the tutorial we will implement a recurrent neural network based language model. The applications of language models are two-fold: First, it allows us to score arbitrary sentences based on how likely they are to occur in the real world. This gives us a measure of grammatical and semantic correctness. Such models are typically used as part of Machine Translation systems. Secondly, a language model allows us to generate new text (I think that's the much cooler application). Training a language model on Shakespeare allows us to generate Shakespeare-like text. This fun post by Andrej Karpathy demonstrates what character-level language models based on RNNs are capable of.

I'm assuming that you are somewhat familiar with basic Neural Networks. If you're not, you may want to head over to Implementing A Neural Network From Scratch, which guides you through the ideas and implementation behind non-recurrent networks.

WHAT ARE RNNS?

The idea behind RNNs is to make use of sequential information. In a traditional neural network we assume that all inputs (and outputs) are independent of each other. But for many tasks that's a very bad idea. If you want to predict the next word in a sentence you better know which words came before it. RNNs are called *recurrent* because they perform the same task for every element of a sequence, with the output being depended on the previous computations. Another way to think about RNNs is that they have a "memory" which captures information about what has been calculated so far. In theory RNNs can make use of information in arbitrarily long sequences, but in practice they are limited to looking back only a few steps (more on this later). Here is what a typical RNN looks like:



A recurrent neural network and the unfolding in time of the computation involved in its forward computation. Source: Nature

The above diagram shows a RNN being *unrolled* (or unfolded) into a full network. By unrolling we simply mean that we write out the network for the complete sequence. For example, if the sequence we care about is a sentence of 5 words, the network would be unrolled into a 5-layer neural network, one layer for each word. The formulas that govern the computation happening in a RNN are as follows:

- x_t is the input at time step t. For example, x_1 could be a one-hot vector corresponding to the second word of a sentence.
- s_t is the hidden state at time step t. It's the "memory" of the network. s_t is calculated based on the previous hidden state and the input at the current step: $s_t = f(Ux_t + Ws_{t-1})$. The function f usually is a nonlinearity such as tanh or ReLU. s_{-1}, which is required to calculate the first hidden state, is typically initialized to all zeroes.
- o_t is the output at step t. For example, if we wanted to predict the next word in a sentence it would be a vector of probabilities across our vocabulary. $o_t = \mathrm{softmax}(Vs_t)$.

There are a few things to note here:

- You can think of the hidden state s_t as the memory of the network. s_t captures information about what happened in all the previous time steps. The output at step o_t is calculated solely based on the memory at time t. As briefly mentioned above, it's a bit more complicated in practice because s_t typically can't capture information from too many time steps ago.
- Unlike a traditional deep neural network, which uses different parameters at each layer, a RNN shares the same parameters (U, V, W above) across all steps. This reflects the fact that we are performing the same task at each step, just with different inputs. This greatly reduces the total number of parameters we need to learn.
- The above diagram has outputs at each time step, but depending on the task this may not be necessary. For example, when predicting the sentiment of a sentence we may only care about the final output, not the sentiment after each word. Similarly, we may not need inputs at each time step. The main feature of an RNN is its hidden state, which captures some information about a sequence.

WHAT CAN RNNS DO?

RNNs have shown great success in many NLP tasks. At this point I should mention that the most commonly used type of RNNs are LSTMs, which are much better at capturing long-term dependencies than vanilla RNNs are. But don't worry, LSTMs are essentially the same thing as the RNN we will develop in this tutorial, they just have a different way of computing the hidden state. We'll cover LSTMs in more detail in a later post. Here are some example applications of RNNs in NLP (by non means an exhaustive list).

LANGUAGE MODELING AND GENERATING TEXT

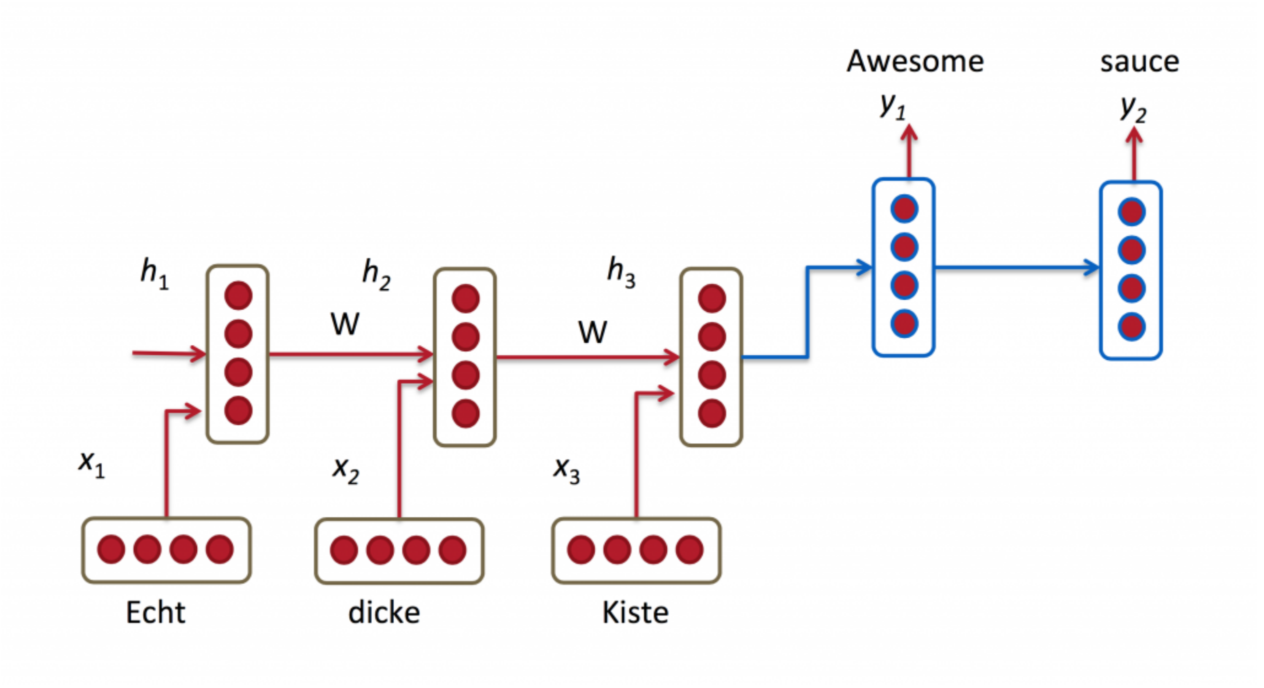
Given a sequence of words we want to predict the probability of each word given the previous words. Language Models allow us to measure how likely a sentence is, which is an important input for Machine Translation (since high-probability sentences are typically correct). A side-effect of being able to predict the next word is that we get a *generative* model, which allows us to generate new text by sampling from the output probabilities. And depending on what our training data is we can generate all kinds of stuff. In Language Modeling our input is typically a sequence of words (encoded as one-hot vectors for example), and our output is the sequence of predicted words. When training the network we set $o_t = x_{t+1}$ since we want the output at step t to be the actual next word.

Research papers about Language Modeling and Generating Text:

- Recurrent neural network based language model
- Extensions of Recurrent neural network based language model
- Generating Text with Recurrent Neural Networks

MACHINE TRANSLATION

Machine Translation is similar to language modeling in that our input is a sequence of words in our source language (e.g. German). We want to output a sequence of words in our target language (e.g. English). A key difference is that our output only starts after we have seen the complete input, because the first word of our translated sentences may require information captured from the complete input sequence.



RNN for Machine Translation. Image Source: http://cs224d.stanford.edu/lectures/CS224d-Lecture8.pdf

Research papers about Machine Translation:

- A Recursive Recurrent Neural Network for Statistical Machine Translation
- Sequence to Sequence Learning with Neural Networks
- Joint Language and Translation Modeling with Recurrent Neural Networks

SPEECH RECOGNITION

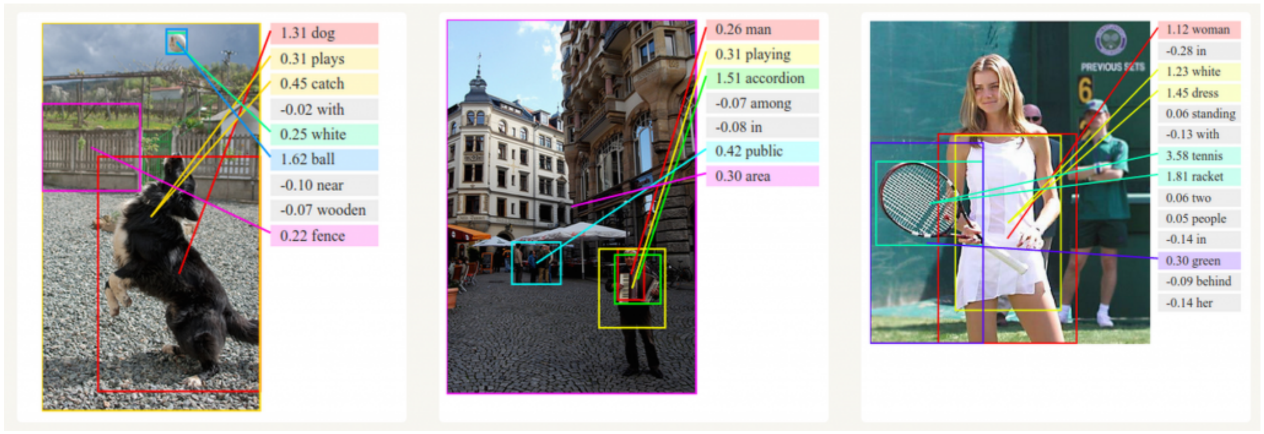
Given an input sequence of acoustic signals from a sound wave, we can predict a sequence of phonetic segments together with their probabilities.

Research papers about Speech Recognition:

- Towards End-to-End Speech Recognition with Recurrent Neural Networks

GENERATING IMAGE DESCRIPTIONS

Together with convolutional Neural Networks, RNNs have been used as part of a model to generate descriptions for unlabeled images. It's quite amazing how well this seems to work. The combined model even aligns the generated words with features found in the images.



Deep Visual-Semantic Alignments for Generating Image Descriptions. Source: http://cs.stanford.edu/people/karpathy/deepimagesent/

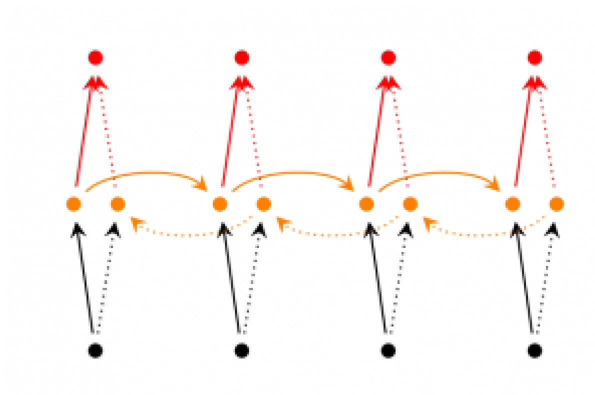
TRAINING RNNS

Training a RNN is similar to training a traditional Neural Network. We also use the backpropagation algorithm, but with a little twist. Because the parameters are shared by all time steps in the network, the gradient at each output depends not only on the calculations of the current time step, but also the previous time steps. For example, in order to calculate the gradient at $t = 4$ we would need to backpropagate 3 steps and sum up the gradients. This is called Backpropagation Through Time (BPTT). If this doesn't make a whole lot of sense yet, don't worry, we'll have a whole post on the gory details. For now, just be aware of the fact that vanilla RNNs trained with BPTT have difficulties learning long-term dependencies (e.g. dependencies between steps that are far apart) due to what is called the vanishing/exploding gradient problem. There exists some machinery to deal with these problems, and certain types of RNNs (like LSTMs) were specifically designed to get around them.

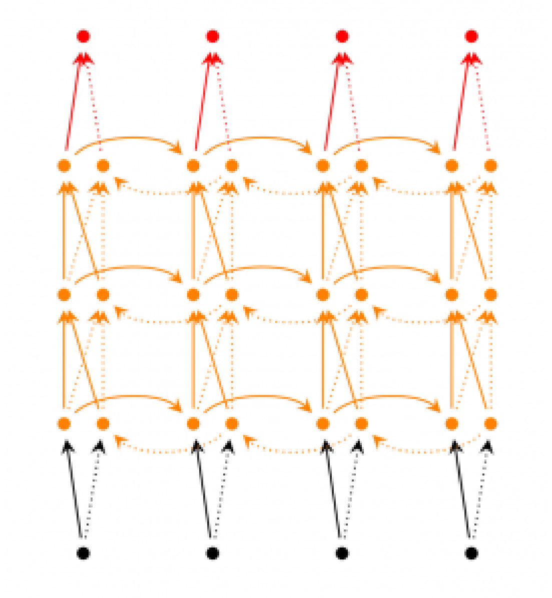
RNN EXTENSIONS

Over the years researchers have developed more sophisticated types of RNNs to deal with some of the shortcomings of the vanilla RNN model. We will cover them in more detail in a later post, but I want this section to serve as a brief overview so that you are familiar with the taxonomy of models.

Bidirectional RNNs are based on the idea that the output at time t may not only depend on the previous elements in the sequence, but also future elements. For example, to predict a missing word in a sequence you want to look at both the left and the right context. Bidirectional RNNs are quite simple. They are just two RNNs stacked on top of each other. The output is then computed based on the hidden state of both RNNs.



Deep (Bidirectional) RNNs are similar to Bidirectional RNNs, only that we now have multiple layers per time step. In practice this gives us a higher learning capacity (but we also need a lot of training data).



LSTM networks are quite popular these days and we briefly talked about them above. LSTMs don't have a fundamentally different architecture from RNNs, but they use a different function to compute the hidden state. The memory in LSTMs are called *cells* and you can think of them as black boxes that take as input the previous state h_{t-1} and current input x_t. Internally these cells decide what to keep in (and what to erase from) memory. They then combine the previous state, the current memory, and the input. It turns out that these types of units are very efficient at capturing long-term dependencies. LSTMs can be quite confusing in the beginning but if you're interested in learning more this post has an excellent explanation.

CONCLUSION

So far so good. I hope you've gotten a basic understanding of what RNNs are and what they can do. In the next post we'll implement a first version of our language model RNN using Python and Theano. Please leave questions in the comments!

Posted in: Deep Learning, Neural Networks, Recurrent Neural Networks

← *Speeding up your Neural Network with Theano and the GPU*

Recurrent Neural Networks Tutorial, Part 2 – Implementing a RNN with Python, Numpy and Theano →

WILDML

AI, DEEP LEARNING, NLP

MENU

RECURRENT NEURAL NETWORKS TUTORIAL, PART 2 – IMPLEMENTING A RNN WITH PYTHON, NUMPY AND THEANO

September 30, 2015

This the second part of the Recurrent Neural Network Tutorial. The first part is here.

Code to follow along is on Github.

In this part we will implement a full Recurrent Neural Network from scratch using Python and optimize our implementation using Theano, a library to perform operations on a GPU. The full code is available on Github. I will skip over some boilerplate code that is not essential to understanding Recurrent Neural Networks, but all of that is also on Github.

LANGUAGE MODELING

Our goal is to build a Language Model using a Recurrent Neural Network. Here's what that means. Let's say we have sentence of m words. A language model allows us to predict the probability of observing the sentence (in a given dataset) as:

$$P(w_1, ..., w_m) = \prod_{i=1}^{m} P(w_i \mid w_1, ..., w_{i-1})$$

In words, the probability of a sentence is the product of probabilities of each word given the words that came before it. So, the probability of the sentence "He went to buy some chocolate" would be the probability of "chocolate" given "He went to buy some", multiplied by the probability of "some" given "He went to buy", and so on.

Why is that useful? Why would we want to assign a probability to observing a sentence?

First, such a model can be used as a scoring mechanism. For example, a Machine Translation system typically generates multiple candidates for an input sentence. You could use a language model to pick the most probable sentence. Intuitively, the most probable sentence is likely to be grammatically correct. Similar scoring happens in speech recognition systems.

But solving the Language Modeling problem also has a cool side effect. Because we can predict the probability of a word given the preceding words, we are able to generate new text. It's a *generative model*. Given an existing sequence of words we sample a next word from the predicted probabilities, and repeat the process until we have a full sentence. Andrej Karparthy has a great post that demonstrates what language models are capable of. His models are trained on single characters as opposed to full words, and can generate anything from Shakespeare to Linux Code.

Note that in the above equation the probability of each word is conditioned on **all** previous words. In practice, many models have a hard time representing such long-term dependencies due to computational or memory constraints. They are typically limited to looking at only a few of the previous words. RNNs can, in theory, capture such long-term dependencies, but in practice it's a bit more complex. We'll explore that in a later post.

TRAINING DATA AND PREPROCESSING

To train our language model we need text to learn from. Fortunately we don't need any labels to train a language model, just raw text. I downloaded 15,000 longish reddit comments from a dataset available on Google's BigQuery. Text generated by our model will sound like reddit commenters (hopefully)! But as with most Machine Learning projects we first need to do some pre-processing to get our data into the right format.

1. TOKENIZE TEXT

We have raw text, but we want to make predictions on a per-word basis. This means we must *tokenize* our comments into sentences, and sentences into words. We could just split each of the comments by spaces, but that wouldn't handle punctuation properly. The sentence "He left!" should be 3 tokens: "He", "left", "!". We'll use NLTK's `word_tokenize` and `sent_tokenize` methods, which do most of the hard work for us.

2. REMOVE INFREQUENT WORDS

Most words in our text will only appear one or two times. It's a good idea to remove these infrequent words. Having a huge vocabulary will make our model slow to train (we'll talk about why that is later), and because we don't have a lot of contextual examples for such words we wouldn't be able to learn how to use them correctly anyway. That's quite similar to how humans learn. To really understand how to appropriately use a word you need to have seen it in different contexts.

In our code we limit our vocabulary to the `vocabulary_size` most common words (which I set to 8000, but feel free to change it). We replace all words not included in our vocabulary by `UNKNOWN_TOKEN`. For example, if we don't include the word "nonlinearities" in our vocabulary, the sentence "nonlineraties are important in neural networks" becomes "UNKNOWN_TOKEN are important in Neural Networks". The word `UNKNOWN_TOKEN` will become part of our vocabulary and we will predict it just like any other word. When we generate new text we can replace `UNKNOWN_TOKEN` again, for example by taking a randomly sampled word not in our vocabulary, or we could just generate sentences until we get one that doesn't contain an unknown token.

3. PREPEND SPECIAL START AND END TOKENS

We also want to learn which words tend start and end a sentence. To do this we prepend a special `SENTENCE_START` token, and append a special `SENTENCE_END` token to each sentence. This allows us to ask: Given that the first token is `SENTENCE_START`, what is the likely next word (the actual first word of the sentence)?

4. BUILD TRAINING DATA MATRICES

The input to our Recurrent Neural Networks are vectors, not strings. So we create a mapping between words and indices, `index_to_word`, and `word_to_index`. For example, the word "friendly" may be at index 2001. A training example x may look like `[0, 179, 341, 416]`, where 0 corresponds to `SENTENCE_START`. The corresponding label y would be `[179, 341, 416, 1]`. Remember that our goal is to predict the next word, so y is just the x vector shifted by one position with the last element being the `SENTENCE_END` token. In other words, the correct prediction for word `179` above would be `341`, the actual next word.

```
vocabulary_size = 8000
unknown_token = "UNKNOWN_TOKEN"
sentence_start_token = "SENTENCE_START"
sentence_end_token = "SENTENCE_END"

# Read the data and append SENTENCE_START and SENTENCE_END tokens
print "Reading CSV file..."
with open('data/reddit-comments-2015-08.csv', 'rb') as f:
    reader = csv.reader(f, skipinitialspace=True)
    reader.next()
    # Split full comments into sentences
    sentences = itertools.chain(*[nltk.sent_tokenize(x[0].decode('utf-8').lower()) for x in
    # Append SENTENCE_START and SENTENCE_END
    sentences = ["%s %s %s" % (sentence_start_token, x, sentence_end_token) for x in sente
print "Parsed %d sentences." % (len(sentences))

# Tokenize the sentences into words
tokenized_sentences = [nltk.word_tokenize(sent) for sent in sentences]

# Count the word frequencies
word_freq = nltk.FreqDist(itertools.chain(*tokenized_sentences))
print "Found %d unique words tokens." % len(word_freq.items())

# Get the most common words and build index_to_word and word_to_index vectors
vocab = word_freq.most_common(vocabulary_size-1)
index_to_word = [x[0] for x in vocab]
index_to_word.append(unknown_token)
word_to_index = dict([(w,i) for i,w in enumerate(index_to_word)])

print "Using vocabulary size %d." % vocabulary_size
print "The least frequent word in our vocabulary is '%s' and appeared %d times." % (vocab[-

# Replace all words not in our vocabulary with the unknown token
for i, sent in enumerate(tokenized_sentences):
    tokenized_sentences[i] = [w if w in word_to_index else unknown_token for w in sent]

print "\nExample sentence: '%s'" % sentences[0]
print "\nExample sentence after Pre-processing: '%s'" % tokenized_sentences[0]

# Create the training data
X_train = np.asarray([[word_to_index[w] for w in sent[:-1]] for sent in tokenized_sentence
y_train = np.asarray([[word_to_index[w] for w in sent[1:]] for sent in tokenized_sentences
```

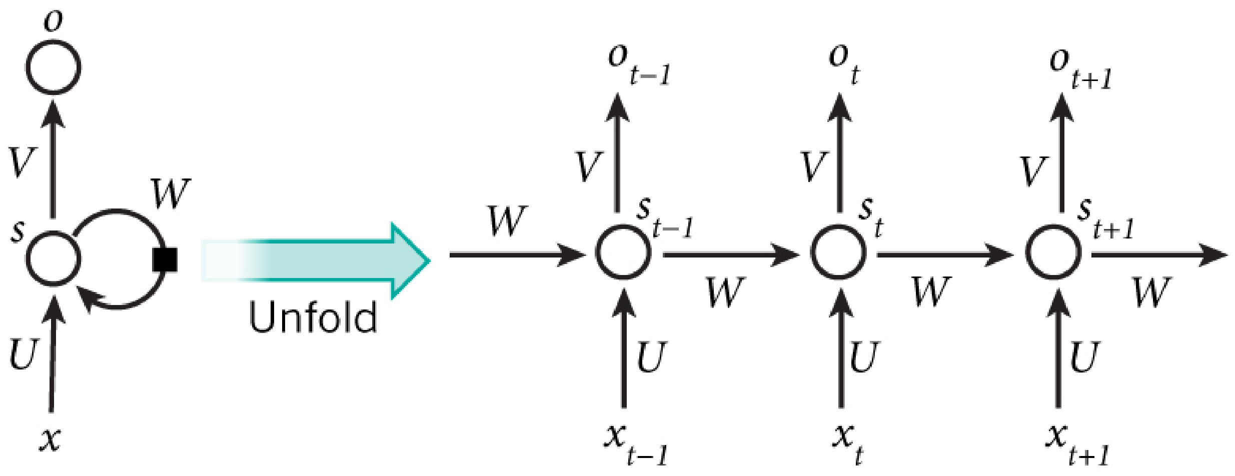
Here's an actual training example from our text:

```
x:
SENTENCE_START what are n't you understanding about this ? !
[0, 51, 27, 16, 10, 856, 53, 25, 34, 69]

y:
what are n't you understanding about this ? ! SENTENCE_END
[51, 27, 16, 10, 856, 53, 25, 34, 69, 1]
```

BUILDING THE RNN

For a general overview of RNNs take a look at first part of the tutorial.



A recurrent neural network and the unfolding in time of the computation involved in its forward computation.

Let's get concrete and see what the RNN for our language model looks like. The input x will be a sequence of words (just like the example printed above) and each x_t is a single word. But there's one more thing: Because of how matrix multiplication works we can't simply use a word index (like 36) as an input. Instead, we represent each word as a *one-hot vector* of size `vocabulary_size`. For example, the word with index 36 would be the vector of all 0's and a 1 at position 36. So, each x_t will become a vector, and x will be a matrix, with each row representing a word. We'll perform this transformation in our Neural Network code instead of doing it in the pre-processing. The output of our network o has a similar format. Each o_t is a vector of `vocabulary_size` elements, and each element represents the probability of that word being the next word in the sentence.

Let's recap the equations for the RNN from the first part of the tutorial:

$$\begin{aligned} s_t &= \tanh(Ux_t + Ws_{t-1}) \\ o_t &= \mathrm{softmax}(Vs_t) \end{aligned}$$

I always find it useful to write down the dimensions of the matrices and vectors. Let's assume we pick a vocabulary size $C = 8000$ and a hidden layer size $H = 100$. You can think of the hidden layer size as the "memory" of our network. Making it bigger allows us to learn more complex patterns, but also results in additional computation. Then we have:

$$\begin{aligned} x_t &\in \mathbb{R}^{8000} \\ o_t &\in \mathbb{R}^{8000} \\ s_t &\in \mathbb{R}^{100} \\ U &\in \mathbb{R}^{100\times 8000} \\ V &\in \mathbb{R}^{8000\times 100} \\ W &\in \mathbb{R}^{100\times 100} \end{aligned}$$

This is valuable information. Remember that U, V and W are the parameters of our network we want to learn from data. Thus, we need to learn a total of $2HC + H^2$ parameters. In the case of $C = 8000$ and $H = 100$ that's 1,610,000. The dimensions also tell us the bottleneck of our model. Note that because x_t is a one-hot vector, multiplying it with U is essentially the same as selecting a column of U, so we don't need to perform the full multiplication. Then, the biggest matrix multiplication in our network is Vs_t. That's why we want to keep our vocabulary size small if possible.

Armed with this, it's time to start our implementation.

INITIALIZATION

We start by declaring a RNN class an initializing our parameters. I'm calling this class `RNNNumpy` because we will implement a Theano version later. Initializing the parameters U, V and W is a bit tricky. We can't just initialize them to 0's because that would result in symmetric calculations in all our layers. We must initialize them randomly. Because proper initialization seems to have an impact on training results there has been lot of research in this area. It turns out that the best initialization depends on the activation function ($\tanh$ in our case) and one recommended approach is to initialize the weights randomly in the interval from $\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right]$ where n is the number of incoming connections from the previous layer. This may sound overly complicated, but don't worry too much it. As long as you initialize your parameters to small random values it typically works out fine.

```
class RNNNumpy:

    def __init__(self, word_dim, hidden_dim=100, bptt_truncate=4):
        # Assign instance variables
        self.word_dim = word_dim
        self.hidden_dim = hidden_dim
        self.bptt_truncate = bptt_truncate
        # Randomly initialize the network parameters
        self.U = np.random.uniform(-np.sqrt(1./word_dim), np.sqrt(1./word_dim), (hidden_dim,
        self.V = np.random.uniform(-np.sqrt(1./hidden_dim), np.sqrt(1./hidden_dim), (word_di
        self.W = np.random.uniform(-np.sqrt(1./hidden_dim), np.sqrt(1./hidden_dim), (hidden_
```

Above, `word_dim` is the size of our vocabulary, and `hidden_dim` is the size of our hidden layer (we can pick it). Don't worry about the `bptt_truncate` parameter for now, we'll explain what that is later.

FORWARD PROPAGATION

Next, let's implement the forward propagation (predicting word probabilities) defined by our equations above:

```
def forward_propagation(self, x):
    # The total number of time steps
    T = len(x)
    # During forward propagation we save all hidden states in s because need them later.
    # We add one additional element for the initial hidden, which we set to 0
    s = np.zeros((T + 1, self.hidden_dim))
    s[-1] = np.zeros(self.hidden_dim)
    # The outputs at each time step. Again, we save them for later.
    o = np.zeros((T, self.word_dim))
    # For each time step...
    for t in np.arange(T):
        # Note that we are indxing U by x[t]. This is the same as multiplying U with a one-h
        s[t] = np.tanh(self.U[:,x[t]] + self.W.dot(s[t-1]))
        o[t] = softmax(self.V.dot(s[t]))
    return [o, s]

RNNNumpy.forward_propagation = forward_propagation
```

We not only return the calculated outputs, but also the hidden states. We will use them later to calculate the gradients, and by returning them here we avoid duplicate computation. Each o_t is a vector of probabilities representing the words in our vocabulary, but sometimes, for example when evaluating our model, all we want is the next word with the highest probability. We call this function `predict`:

```
def predict(self, x):
    # Perform forward propagation and return index of the highest score
    o, s = self.forward_propagation(x)
    return np.argmax(o, axis=1)

RNNNumpy.predict = predict
```

Let's try our newly implemented methods and see an example output:

```
np.random.seed(10)
model = RNNNumpy(vocabulary_size)
o, s = model.forward_propagation(X_train[10])
print o.shape
print o
```

```
(45, 8000)
[[ 0.00012408  0.0001244   0.00012603 ...,  0.00012515  0.00012488
   0.00012508]
 [ 0.00012536  0.00012582  0.00012436 ...,  0.00012482  0.00012456
   0.00012451]
 [ 0.00012387  0.0001252   0.00012474 ...,  0.00012559  0.00012588
   0.00012551]
 ...,
 [ 0.00012414  0.00012455  0.0001252  ...,  0.00012487  0.00012494
   0.0001263 ]
 [ 0.0001252   0.00012393  0.00012509 ...,  0.00012407  0.00012578
   0.00012502]
 [ 0.00012472  0.0001253   0.00012487 ...,  0.00012463  0.00012536
   0.00012665]]
```

For each word in the sentence (45 above), our model made 8000 predictions representing probabilities of the next word. Note that because we initialized U, V, W to random values these predictions are completely random right now. The following gives the indices of the highest probability predictions for each word:

```
predictions = model.predict(X_train[10])
print predictions.shape
print predictions
```

```
(45,)
[1284 5221 7653 7430 1013 3562 7366 4860 2212 6601 7299 4556 2481 238 2539
 21 6548 261 1780 2005 1810 5376 4146 477 7051 4832 4991 897 3485 21
 7291 2007 6006 760 4864 2182 6569 2800 2752 6821 4437 7021 7875 6912 3575]
```

CALCULATING THE LOSS

To train our network we need a way to measure the errors it makes. We call this the loss function L, and our goal is find the parameters U, V and W that minimize the loss function for our training data. A common choice for the loss function is the cross-entropy loss. If we have N training examples (words in our text) and C classes (the size of our vocabulary) then the loss with respect to our predictions o and the true labels y is given by:

$$L(y, o) = -\frac{1}{N} \sum_{n \in N} y_n \log o_n$$

The formula looks a bit complicated, but all it really does is sum over our training examples and add to the loss based on how off our prediction are. The further away y (the correct words) and o (our predictions), the greater the loss will be. We implement the function `calculate_loss`:

```
def calculate_total_loss(self, x, y):
    L = 0
    # For each sentence...
    for i in np.arange(len(y)):
        o, s = self.forward_propagation(x[i])
        # We only care about our prediction of the "correct" words
        correct_word_predictions = o[np.arange(len(y[i])), y[i]]
        # Add to the loss based on how off we were
        L += -1 * np.sum(np.log(correct_word_predictions))
    return L

def calculate_loss(self, x, y):
    # Divide the total loss by the number of training examples
    N = np.sum((len(y_i) for y_i in y))
    return self.calculate_total_loss(x,y)/N

RNNNumpy.calculate_total_loss = calculate_total_loss
RNNNumpy.calculate_loss = calculate_loss
```

Let's take a step back and think about what the loss should be for random predictions. That will give us a baseline and make sure our implementation is correct. We have C words in our vocabulary, so each word should be (on average) predicted with probability $1/C$, which would yield a loss of $L = -\frac{1}{N} N \log \frac{1}{C} = \log C$:

```
# Limit to 1000 examples to save time
print "Expected Loss for random predictions: %f" % np.log(vocabulary_size)
print "Actual loss: %f" % model.calculate_loss(X_train[:1000], y_train[:1000])
```

```
Expected Loss for random predictions: 8.987197
Actual loss: 8.987440
```

Pretty close! Keep in mind that evaluating the loss on the full dataset is an expensive operation and can take hours if you have a lot of data!

TRAINING THE RNN WITH SGD AND BACKPROPAGATION THROUGH TIME (BPTT)

Remember that we want to find the parameters U, V and W that minimize the total loss on the training data. The most common way to do this is SGD, Stochastic Gradient Descent. The idea behind SGD is pretty simple. We iterate over all our training examples and during each iteration we nudge the parameters into a direction that reduces the error. These directions are given by the gradients on the loss: $\frac{\partial L}{\partial U}, \frac{\partial L}{\partial V}, \frac{\partial L}{\partial W}$. SGD also needs a *learning rate*, which defines how big of a step we want to make in each iteration. SGD is the most popular optimization method not only for Neural Networks, but also for many other Machine Learning algorithms. As such there has been a lot of research on how to optimize SGD using batching, parallelism and adaptive learning rates. Even though the basic idea is simple, implementing SGD in a really efficient way can become very complex. If you want to learn more about SGD this is a good place to start. Due to its popularity there are a wealth of tutorials floating around the web, and I don't want to duplicate them here. I'll implement a simple version of SGD that should be understandable even without a background in optimization.

But how do we calculate those gradients we mentioned above? In a traditional Neural Network we do this through the backpropagation algorithm. In RNNs we use a slightly modified version of the this algorithm called *Backpropagation Through Time (BPTT)*. Because the parameters are shared by all time steps in the network, the gradient at each output depends not only on the calculations of the current time step, but also the previous time steps. If you know calculus, it really is just applying the chain rule. The next part of the tutorial will be all about BPTT, so I won't go into detailed derivation here. For a general introduction to backpropagation check out this and this post. For now you can treat BPTT as a black box. It takes as input a training example (x, y) and returns the gradients $\frac{\partial L}{\partial U}, \frac{\partial L}{\partial V}, \frac{\partial L}{\partial W}$.

```
def bptt(self, x, y):
    T = len(y)
    # Perform forward propagation
    o, s = self.forward_propagation(x)
    # We accumulate the gradients in these variables
    dLdU = np.zeros(self.U.shape)
    dLdV = np.zeros(self.V.shape)
    dLdW = np.zeros(self.W.shape)
    delta_o = o
    delta_o[np.arange(len(y)), y] -= 1.
    # For each output backwards...
    for t in np.arange(T)[::-1]:
        dLdV += np.outer(delta_o[t], s[t].T)
        # Initial delta calculation
        delta_t = self.V.T.dot(delta_o[t]) * (1 - (s[t] ** 2))
        # Backpropagation through time (for at most self.bptt_truncate steps)
        for bptt_step in np.arange(max(0, t-self.bptt_truncate), t+1)[::-1]:
            # print "Backpropagation step t=%d bptt step=%d " % (t, bptt_step)
            dLdW += np.outer(delta_t, s[bptt_step-1])
            dLdU[:,x[bptt_step]] += delta_t
            # Update delta for next step
            delta_t = self.W.T.dot(delta_t) * (1 - s[bptt_step-1] ** 2)
    return [dLdU, dLdV, dLdW]

RNNNumpy.bptt = bptt
```

GRADIENT CHECKING

Whenever you implement backpropagation it is good idea to also implement *gradient checking*, which is a way of verifying that your implementation is correct. The idea behind gradient checking is that derivative of a parameter is equal to the slope at the point, which we can approximate by slightly changing the parameter and then dividing by the change:

$$\frac{\partial L}{\partial \theta} \approx \lim_{h \to 0} \frac{J(\theta + h) - J(\theta - h)}{2h}$$

We then compare the gradient we calculated using backpropagation to the gradient we estimated with the method above. If there's no large difference we are good. The approximation needs to calculate the total loss for *every* parameter, so that gradient checking is very expensive (remember, we had more than a million parameters in the example above). So it's a good idea to perform it on a model with a smaller vocabulary.

```
def gradient_check(self, x, y, h=0.001, error_threshold=0.01):
    # Calculate the gradients using backpropagation. We want to checker if these are correct
    bptt_gradients = self.bptt(x, y)
    # List of all parameters we want to check.
    model_parameters = ['U', 'V', 'W']
    # Gradient check for each parameter
    for pidx, pname in enumerate(model_parameters):
        # Get the actual parameter value from the mode, e.g. model.W
        parameter = operator.attrgetter(pname)(self)
        print "Performing gradient check for parameter %s with size %d." % (pname, np.prod(
        # Iterate over each element of the parameter matrix, e.g. (0,0), (0,1), ...
        it = np.nditer(parameter, flags=['multi_index'], op_flags=['readwrite'])
        while not it.finished:
            ix = it.multi_index
            # Save the original value so we can reset it later
            original_value = parameter[ix]
            # Estimate the gradient using (f(x+h) - f(x-h))/(2*h)
            parameter[ix] = original_value + h
            gradplus = self.calculate_total_loss([x],[y])
            parameter[ix] = original_value - h
            gradminus = self.calculate_total_loss([x],[y])
            estimated_gradient = (gradplus - gradminus)/(2*h)
            # Reset parameter to original value
            parameter[ix] = original_value
            # The gradient for this parameter calculated using backpropagation
            backprop_gradient = bptt_gradients[pidx][ix]
            # calculate The relative error: (|x - y|/(|x| + |y|))
            relative_error = np.abs(backprop_gradient - estimated_gradient)/(np.abs(backpro
            # If the error is to large fail the gradient check
            if relative_error &gt; error_threshold:
                print "Gradient Check ERROR: parameter=%s ix=%s" % (pname, ix)
                print "+h Loss: %f" % gradplus
                print "-h Loss: %f" % gradminus
                print "Estimated_gradient: %f" % estimated_gradient
                print "Backpropagation gradient: %f" % backprop_gradient
                print "Relative Error: %f" % relative_error
                return
            it.iternext()
        print "Gradient check for parameter %s passed." % (pname)

RNNNumpy.gradient_check = gradient_check

# To avoid performing millions of expensive calculations we use a smaller vocabulary size fo
grad_check_vocab_size = 100
np.random.seed(10)
model = RNNNumpy(grad_check_vocab_size, 10, bptt_truncate=1000)
model.gradient_check([0,1,2,3], [1,2,3,4])
```

SGD IMPLEMENTATION

Now that we are able to calculate the gradients for our parameters we can implement SGD. I like to do this in two steps: 1. A function `sdg_step` that calculates the gradients and performs the updates for one batch. 2. An outer loop that iterates through the training set and adjusts the learning rate.

```
# Performs one step of SGD.
def numpy_sdg_step(self, x, y, learning_rate):
    # Calculate the gradients
    dLdU, dLdV, dLdW = self.bptt(x, y)
    # Change parameters according to gradients and learning rate
    self.U -= learning_rate * dLdU
    self.V -= learning_rate * dLdV
    self.W -= learning_rate * dLdW
```

```
RNNNumpy.sgd_step = numpy_sdg_step
```

```
# Outer SGD Loop
# - model: The RNN model instance
# - X_train: The training data set
# - y_train: The training data labels
# - learning_rate: Initial learning rate for SGD
# - nepoch: Number of times to iterate through the complete dataset
# - evaluate_loss_after: Evaluate the loss after this many epochs
def train_with_sgd(model, X_train, y_train, learning_rate=0.005, nepoch=100, evaluate_loss_a
    # We keep track of the losses so we can plot them later
    losses = []
    num_examples_seen = 0
    for epoch in range(nepoch):
        # Optionally evaluate the loss
        if (epoch % evaluate_loss_after == 0):
            loss = model.calculate_loss(X_train, y_train)
            losses.append((num_examples_seen, loss))
            time = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
            print "%s: Loss after num_examples_seen=%d epoch=%d: %f" % (time, num_examples_
            # Adjust the learning rate if loss increases
            if (len(losses) &gt; 1 and losses[-1][1] &gt; losses[-2][1]):
                learning_rate = learning_rate * 0.5
                print "Setting learning rate to %f" % learning_rate
            sys.stdout.flush()
        # For each training example...
        for i in range(len(y_train)):
            # One SGD step
            model.sgd_step(X_train[i], y_train[i], learning_rate)
            num_examples_seen += 1
```

Done! Let's try to get a sense of how long it would take to train our network:

```
np.random.seed(10)
model = RNNNumpy(vocabulary_size)
%timeit model.sgd_step(X_train[10], y_train[10], 0.005)
```

Uh-oh, bad news. One step of SGD takes approximately 350 milliseconds on my laptop. We have about 80,000 examples in our training data, so one epoch (iteration over the whole data set) would take several hours. Multiple epochs would take days, or even weeks! And we're still working with a small dataset compared to what's being used by many of the companies and researchers out there. What now?

Fortunately there are many ways to speed up our code. We could stick with the same model and make our code run faster, or we could modify our model to be less computationally expensive, or both. Researchers have identified many ways to make models less computationally expensive, for example by using a hierarchical softmax or adding projection layers to avoid the large matrix multiplications (see also here or here). But I want to keep our model simple and go the first route: Make our implementation run faster using a GPU. Before doing that though, let's just try to run SGD with a small dataset and check if the loss actually decreases:

```
np.random.seed(10)
# Train on a small subset of the data to see what happens
model = RNNNumpy(vocabulary_size)
losses = train_with_sgd(model, X_train[:100], y_train[:100], nepoch=10, evaluate_loss_after=
```

```
2015-09-30 10:08:19: Loss after num_examples_seen=0 epoch=0: 8.987425
2015-09-30 10:08:35: Loss after num_examples_seen=100 epoch=1: 8.976270
2015-09-30 10:08:50: Loss after num_examples_seen=200 epoch=2: 8.960212
2015-09-30 10:09:06: Loss after num_examples_seen=300 epoch=3: 8.930430
2015-09-30 10:09:22: Loss after num_examples_seen=400 epoch=4: 8.862264
2015-09-30 10:09:38: Loss after num_examples_seen=500 epoch=5: 6.913570
2015-09-30 10:09:53: Loss after num_examples_seen=600 epoch=6: 6.302493
2015-09-30 10:10:07: Loss after num_examples_seen=700 epoch=7: 6.014995
2015-09-30 10:10:24: Loss after num_examples_seen=800 epoch=8: 5.833877
2015-09-30 10:10:39: Loss after num_examples_seen=900 epoch=9: 5.710718
```

Good, it seems like our implementation is at least doing something useful and decreasing the loss, just like we wanted.

TRAINING OUR NETWORK WITH THEANO AND THE GPU

I have previously written a tutorial on Theano, and since all our logic will stay exactly the same I won't go through optimized code here again. I defined a `RNNTheano` class that replaces the numpy calculations with corresponding calculations in Theano. Just like the rest of this post, the code is also available Github.

```
np.random.seed(10)
model = RNNTheano(vocabulary_size)
```

```
%timeit model.sgd_step(X_train[10], y_train[10], 0.005)
```

This time, one SGD step takes 70ms on my Mac (without GPU) and 23ms on a g2.2xlarge Amazon EC2 instance with GPU. That's a 15x improvement over our initial implementation and means we can train our model in hours/days instead of weeks. There are still a vast number of optimizations we could make, but we're good enough for now.

To help you avoid spending days training a model I have pre-trained a Theano model with a hidden layer dimensionality of 50 and a vocabulary size of 8000. I trained it for 50 epochs in about 20 hours. The loss was was still decreasing and training longer would probably have resulted in a better model, but I was running out of time and wanted to publish this post. Feel free to try it out yourself and trian for longer. You can find the model parameters in `data/trained-model-theano.npz` in the Github repository and load them using the `load_model_parameters_theano` method:

```
from utils import load_model_parameters_theano, save_model_parameters_theano

model = RNNTheano(vocabulary_size, hidden_dim=50)
# losses = train_with_sgd(model, X_train, y_train, nepoch=50)
# save_model_parameters_theano('./data/trained-model-theano.npz', model)
load_model_parameters_theano('./data/trained-model-theano.npz', model)
```

GENERATING TEXT

Now that we have our model we can ask it to generate new text for us! Let's implement a helper function to generate new sentences:

```
def generate_sentence(model):
    # We start the sentence with the start token
    new_sentence = [word_to_index[sentence_start_token]]
    # Repeat until we get an end token
    while not new_sentence[-1] == word_to_index[sentence_end_token]:
        next_word_probs = model.forward_propagation(new_sentence)
        sampled_word = word_to_index[unknown_token]
        # We don't want to sample unknown words
        while sampled_word == word_to_index[unknown_token]:
            samples = np.random.multinomial(1, next_word_probs[-1])
            sampled_word = np.argmax(samples)
        new_sentence.append(sampled_word)
    sentence_str = [index_to_word[x] for x in new_sentence[1:-1]]
    return sentence_str

num_sentences = 10
senten_min_length = 7

for i in range(num_sentences):
    sent = []
    # We want long sentences, not sentences with one or two words
    while len(sent) &lt; senten_min_length:
        sent = generate_sentence(model)
    print " ".join(sent)
```

A few selected (censored) sentences. I added capitalization.

- Anyway, to the city scene you're an idiot teenager.
- What ? ! ! ! ! ignore!
- Screw fitness, you're saying: https
- Thanks for the advice to keep my thoughts around girls.
- Yep, please disappear with the terrible generation.

Looking at the generated sentences there are a few interesting things to note. The model successfully learn syntax. It properly places commas (usually before and's and or's) and ends sentence with punctuation. Sometimes it mimics internet speech such as multiple exclamation marks or smileys.

However, the vast majority of generated sentences don't make sense or have grammatical errors (I really picked the best ones above). One reason could be that we did not train our network long enough (or didn't use enough training data). That may be true, but it's most likely not the main reason. **Our vanilla RNN can't generate meaningful text because it's unable to learn dependencies between words that are several steps apart**. That's also why RNNs failed to gain popularity when they were first invented. They were beautiful in theory but didn't work well in practice, and we didn't immediately understand why.

Fortunately, the difficulties in training RNNs are much better understood now. In the next part of this tutorial we will explore the Backpropagation Through Time (BPTT) algorithm in more detail and demonstrate what's called the *vanishing gradient problem*.

This will motivate our move to more sophisticated RNN models, such as LSTMs, which are the current state of the art for many tasks in NLP (and can generate much better reddit comments!). **Everything you learned in this tutorial also applies to LSTMs and other RNN models, so don't feel discouraged if the results for a vanilla RNN are worse then you expected.**

That's it for now. **Please leave questions or feedback in the comments!** and don't forget to check out the /code.

Posted in: Deep Learning, GPU, Language Modeling, Recurrent Neural Networks

← *Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs*

Recurrent Neural Networks Tutorial, Part 3 – Backpropagation Through Time and Vanishing Gradients →

- CONNECT -

- RECENT POSTS -

Learning Reinforcement Learning (with Code, Exercises and Solutions)

WILDML

AI, DEEP LEARNING, NLP

MENU

RECURRENT NEURAL NETWORKS TUTORIAL, PART 3 – BACKPROPAGATION THROUGH TIME AND VANISHING GRADIENTS

October 8, 2015

This the third part of the Recurrent Neural Network Tutorial.

In the previous part of the tutorial we implemented a RNN from scratch, but didn't go into detail on how Backpropagation Through Time (BPTT) algorithms calculates the gradients. In this part we'll give a brief overview of BPTT and explain how it differs from traditional backpropagation. We will then try to understand the *vanishing gradient problem*, which has led to the development of LSTMs and GRUs, two of the currently most popular and powerful models used in NLP (and other areas). The vanishing gradient problem was originally discovered by Sepp Hochreiter in 1991 and has been receiving attention again recently due to the increased application of deep architectures.

To fully understand this part of the tutorial I recommend being familiar with how partial differentiation and basic backpropagation works. If you are not, you can find excellent tutorials here and here and here, in order of increasing difficulty.

BACKPROPAGATION THROUGH TIME (BPTT)

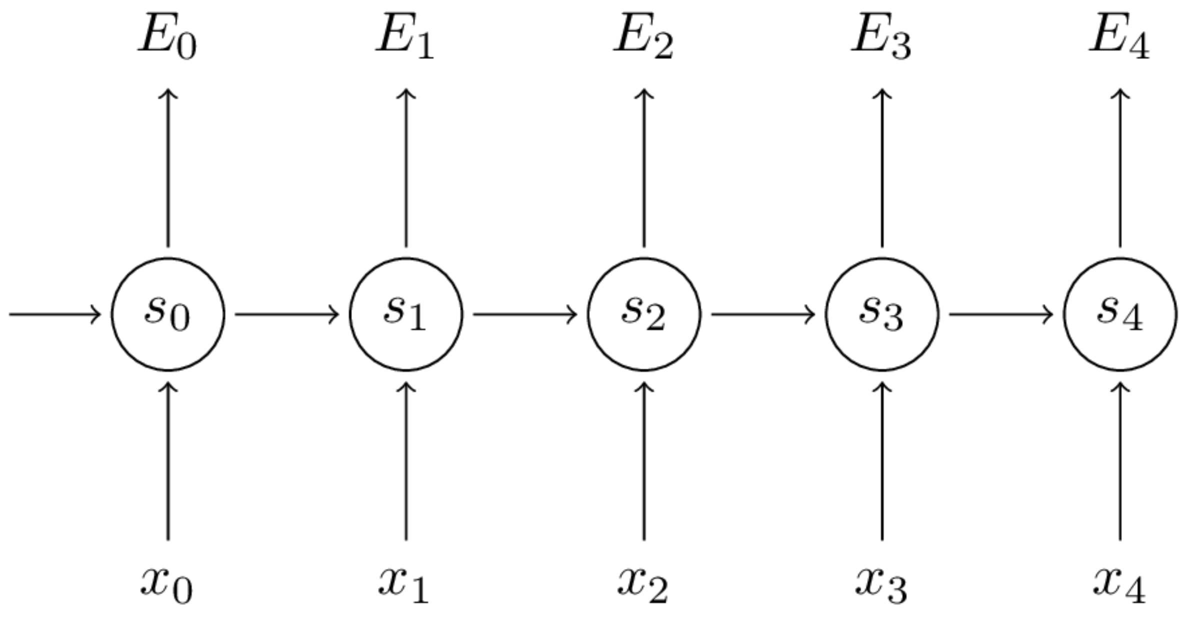
Let's quickly recap the basic equations of our RNN. Note that there's a slight change in notation from o to $\hat{y}$. That's only to stay consistent with some of the literature out there that I am referencing.

$$\begin{aligned} s_t &= \tanh(Ux_t + Ws_{t-1}) \\ \hat{y}_t &= \mathrm{softmax}(Vs_t) \end{aligned}$$

We also defined our *loss*, or error, to be the cross entropy loss, given by:

$$\begin{aligned} E_t(y_t, \hat{y}_t) &= -y_t \log \hat{y}_t \\ E(y, \hat{y}) &= \sum_t E_t(y_t, \hat{y}_t) \\ &= -\sum_t y_t \log \hat{y}_t \end{aligned}$$

Here, y_t is the correct word at time step t, and $\hat{y}_t$ is our prediction. We typically treat the full sequence (sentence) as one training example, so the total error is just the sum of the errors at each time step (word).


$$\begin{tikzcd} & E_0 & E_1 & E_2 & E_3 & E_4 \\ {} \arrow[r] & s_0 \arrow[u] \arrow[r] & s_1 \arrow[u] \arrow[r] & s_2 \arrow[u] \arrow[r] & s_3 \arrow[u] \arrow[r] & s_4 \arrow[u] \\ & x_0 \arrow[u] & x_1 \arrow[u] & x_2 \arrow[u] & x_3 \arrow[u] & x_4 \arrow[u] \end{tikzcd}$$

Remember that our goal is to calculate the gradients of the error with respect to our parameters U, V and W and then learn good parameters using Stochastic Gradient

Descent. Just like we sum up the errors, we also sum up the gradients at each time step for one training example: $\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W}$.

To calculate these gradients we use the chain rule of differentiation. That's the backpropagation algorithm when applied backwards starting from the error. For the rest of this post we'll use E_3 as an example, just to have concrete numbers to work with.

$$\begin{aligned}\frac{\partial E_3}{\partial V} &= \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial V}\\ &= \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial z_3}\frac{\partial z_3}{\partial V}\\ &= (\hat{y}_3 - y_3) \otimes s_3\end{aligned}$$

In the above, $z_3 = Vs_3$, and $\otimes$ is the outer product of two vectors. Don't worry if you don't follow the above, I skipped several steps and you can try calculating these derivatives yourself (good exercise!). The point I'm trying to get across is that $\frac{\partial E_3}{\partial V}$ only depends on the values at the current time step, $\hat{y}_3, y_3, s_3$. If you have these, calculating the gradient for V a simple matrix multiplication.

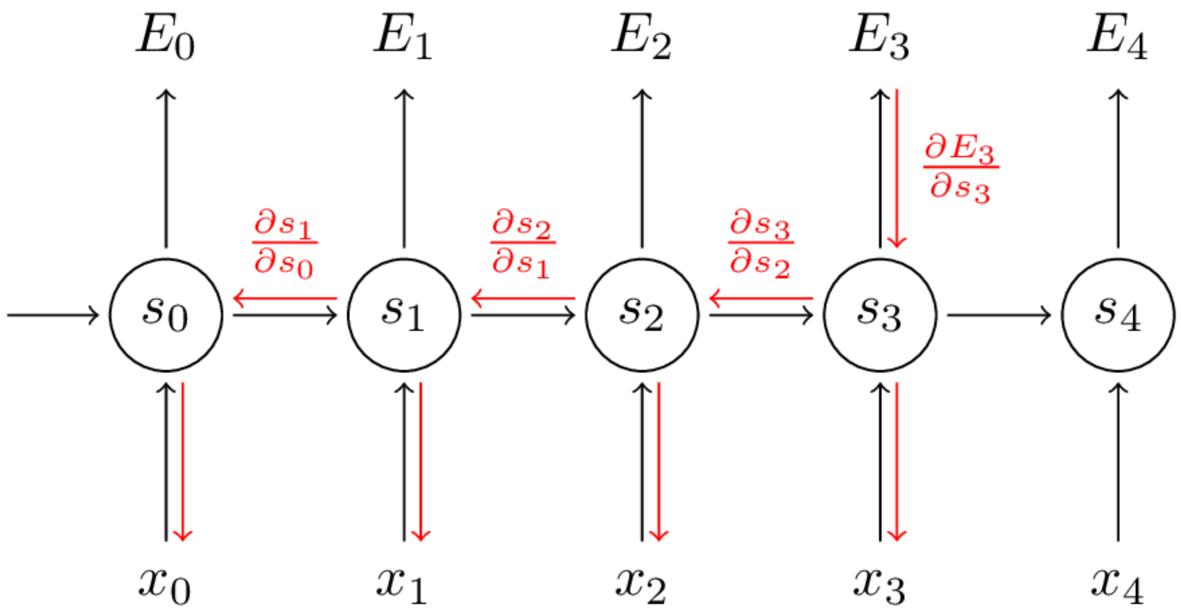
But the story is different for $\frac{\partial E_3}{\partial W}$ (and for U). To see why, we write out the chain rule, just as above:

$$\frac{\partial E_3}{\partial W} = \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial s_3}\frac{\partial s_3}{\partial W}$$

Now, note that $s_3 = \tanh(Ux_t + Ws_2)$ depends on s_2, which depends on W and s_1, and so on. So if we take the derivative with respect to W we can't simply treat s_2 as a constant! We need to apply the chain rule again and what we really have is this:

$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial s_3}\frac{\partial s_3}{\partial s_k}\frac{\partial s_k}{\partial W}$$

We sum up the contributions of each time step to the gradient. In other words, because W is used in every step up to the output we care about, we need to backpropagate gradients from $t = 3$ through the network all the way to $t = 0$:


E_0
E_1
E_2
E_3
E_4
$\frac{\partial E_3}{\partial s_3}$
$\frac{\partial s_1}{\partial s_0}$
$\frac{\partial s_2}{\partial s_1}$
$\frac{\partial s_3}{\partial s_2}$
s_0
s_1
s_2
s_3
s_4
x_0
x_1
x_2
x_3
x_4

Note that this is exactly the same as the standard backpropagation algorithm that we use in deep Feedforward Neural Networks. The key difference is that we sum up the gradients for W at each time step. In a traditional NN we don't share parameters across layers, so we don't need to sum anything. But in my opinion BPTT is just a fancy name for standard backpropagation on an unrolled RNN. Just like with Backpropagation you could define a delta vector that you pass backwards, e.g.: $\delta_2^{(3)} = \frac{\partial E_3}{\partial z_2} = \frac{\partial E_3}{\partial s_3}\frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial z_2}$ with $z_2 = Ux_2 + Ws_1$. Then the same equations will apply.

In code, a naive implementation of BPTT looks something like this:

```
def bptt(self, x, y):
    T = len(y)
    # Perform forward propagation
    o, s = self.forward_propagation(x)
    # We accumulate the gradients in these variables
    dLdU = np.zeros(self.U.shape)
    dLdV = np.zeros(self.V.shape)
    dLdW = np.zeros(self.W.shape)
    delta_o = o
    delta_o[np.arange(len(y)), y] -= 1.
    # For each output backwards...
    for t in np.arange(T)[::-1]:
        dLdV += np.outer(delta_o[t], s[t].T)
        # Initial delta calculation: dL/dz
        delta_t = self.V.T.dot(delta_o[t]) * (1 - (s[t] ** 2))
        # Backpropagation through time (for at most self.bptt_truncate steps)
        for bptt_step in np.arange(max(0, t-self.bptt_truncate), t+1)[::-1]:
            # print "Backpropagation step t=%d bptt step=%d " % (t, bptt_step)
            # Add to gradients at each previous step
            dLdW += np.outer(delta_t, s[bptt_step-1])
            dLdU[:,x[bptt_step]] += delta_t
            # Update delta for next step dL/dz at t-1
            delta_t = self.W.T.dot(delta_t) * (1 - s[bptt_step-1] ** 2)
    return [dLdU, dLdV, dLdW]
```

This should also give you an idea of why standard RNNs are hard to train: Sequences (sentences) can be quite long, perhaps 20 words or more, and thus you need to back-propagate through many layers. In practice many people *truncate* the backpropagation to a few steps.

THE VANISHING GRADIENT PROBLEM

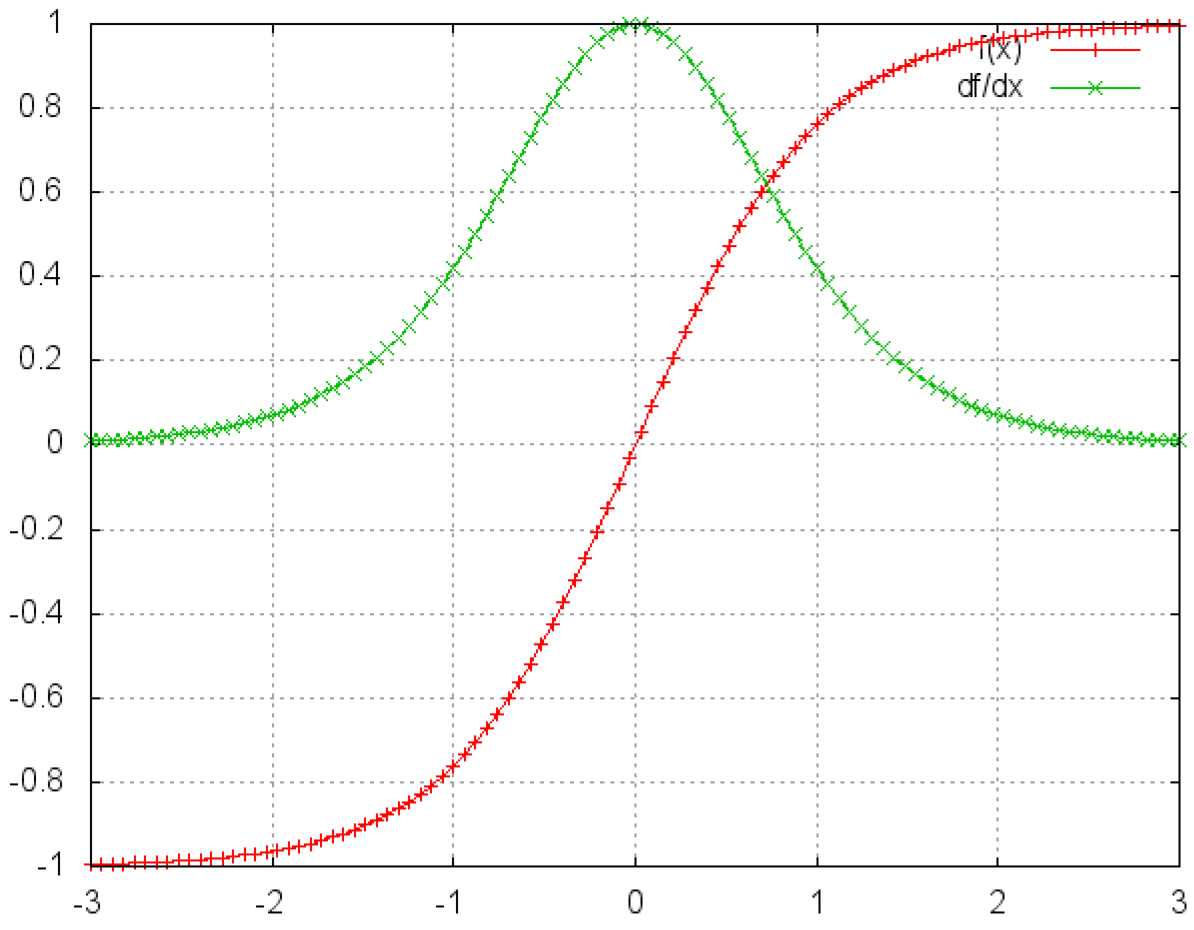
In previous parts of the tutorial I mentioned that RNNs have difficulties learning long-range dependencies – interactions between words that are several steps apart. That's problematic because the meaning of an English sentence is often determined by words that aren't very close: "The man who wore a wig on his head went inside". The sentence is really about a man going inside, not about the wig. But it's unlikely that a plain RNN would be able capture such information. To understand why, let's take a closer look at the gradient we calculated above:

$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial s_3}\frac{\partial s_3}{\partial s_k}\frac{\partial s_k}{\partial W}$$

Note that $\frac{\partial s_3}{\partial s_k}$ is a chain rule in itself! For example, $\frac{\partial s_3}{\partial s_1} = \frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial s_1}$. Also note that because we are taking the derivative of a vector function with respect to a vector, the result is a matrix (called the Jacobian matrix) whose elements are all the pointwise derivatives. We can rewrite the above gradient:

$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial s_3}\left(\prod_{j=k+1}^{3} \frac{\partial s_j}{\partial s_{j-1}}\right)\frac{\partial s_k}{\partial W}$$

It turns out (I won't prove it here but this paper goes into detail) that the 2-norm, which you can think of it as an absolute value, of the above Jacobian matrix has an upper bound of 1. This makes intuitive sense because our $tanh$ (or sigmoid) activation function maps all values into a range between -1 and 1, and the derivative is bounded by 1 (1/4 in the case of sigmoid) as well:



tanh and derivative. Source: http://nn.readthedocs.org/en/rtd/transfer/

You can see that the $\tanh$ and sigmoid functions have derivatives of 0 at both ends. They approach a flat line. When this happens we say the corresponding neurons are saturated. They have a zero gradient and drive other gradients in previous layers towards 0. Thus, with small values in the matrix and multiple matrix multiplications ($t-k$ in particular) the gradient values are shrinking exponentially fast, eventually vanishing completely after a few time steps. Gradient contributions from "far away" steps become zero, and the state at those steps doesn't contribute to what you are learning: You end up not learning long-range dependencies. Vanishing gradients aren't exclusive to RNNs. They also happen in deep Feedforward Neural Networks. It's just that

RNNs tend to be very deep (as deep as the sentence length in our case), which makes the problem a lot more common.

It is easy to imagine that, depending on our activation functions and network parameters, we could get exploding instead of vanishing gradients if the values of the Jacobian matrix are large. Indeed, that's called the *exploding gradient problem*. The reason that vanishing gradients have received more attention than exploding gradients is two-fold. For one, exploding gradients are obvious. Your gradients will become NaN (not a number) and your program will crash. Secondly, clipping the gradients at a pre-defined threshold (as discussed in this paper) is a very simple and effective solution to exploding gradients. Vanishing gradients are more problematic because it's not obvious when they occur or how to deal with them.

Fortunately, there are a few ways to combat the vanishing gradient problem. Proper initialization of the W matrix can reduce the effect of vanishing gradients. So can regularization. A more preferred solution is to use ReLU instead of $tanh$ or sigmoid activation functions. The ReLU derivative is a constant of either 0 or 1, so it isn't as likely to suffer from vanishing gradients. An even more popular solution is to use Long Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU) architectures. LSTMs were first proposed in 1997 and are the perhaps most widely used models in NLP today. GRUs, first proposed in 2014, are simplified versions of LSTMs. Both of these RNN architectures were explicitly designed to deal with vanishing gradients and efficiently learn long-range dependencies. We'll cover them in the next part of this tutorial.

Please leave questions or feedback in the comments!

Posted in: Deep Learning, Language Modeling, Recurrent Neural Networks, RNNs

WILDML

AI, DEEP LEARNING, NLP

MENU

RECURRENT NEURAL NETWORK TUTORIAL, PART 4 – IMPLEMENTING A GRU/LSTM RNN WITH PYTHON AND THEANO

October 27, 2015

The code for this post is on Github. This is part 4, the last part of the Recurrent Neural Network Tutorial. The previous parts are:

- Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs
- Recurrent Neural Networks Tutorial, Part 2 – Implementing a RNN with Python, Numpy and Theano
- Recurrent Neural Networks Tutorial, Part 3 – Backpropagation Through Time and Vanishing Gradients

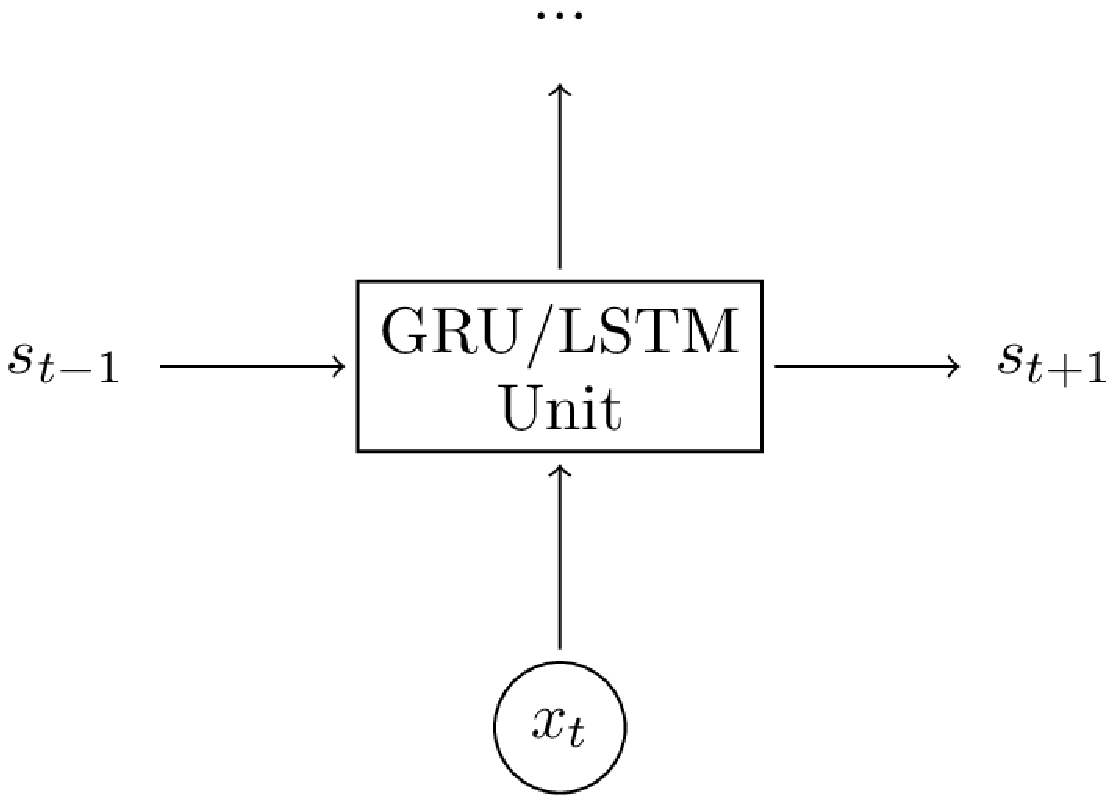
In this post we'll learn about LSTM (Long Short Term Memory) networks and GRUs (Gated Recurrent Units). LSTMs were first proposed in 1997 by Sepp Hochreiter and Jürgen Schmidhuber, and are among the most widely used models in Deep Learning for NLP today. GRUs, first used in 2014, are a simpler variant of LSTMs that share many of the same properties. Let's start by looking at LSTMs, and then we'll see how GRUs are different.

LSTM NETWORKS

In part 3 we looked at how the vanishing gradient problem prevents standard RNNs from learning long-term dependencies. LSTMs were designed to combat vanishing gradients through a *gating* mechanism. To understand what this means, let's look at how a LSTM calculates a hidden state s_t (I'm using $\circ$ to mean elementwise multiplication):

$$
\begin{aligned}
i &= \sigma(x_t U^i + s_{t-1} W^i) \\
f &= \sigma(x_t U^f + s_{t-1} W^f) \\
o &= \sigma(x_t U^o + s_{t-1} W^o) \\
g &= \ tanh(x_t U^g + s_{t-1} W^g) \\
c_t &= c_{t-1} \circ f + g \circ i \\
s_t &= \tanh(c_t) \circ o
\end{aligned}
$$

These equations look quite complicated, but actually it's not that hard. First, notice that a LSTM layer is just another way to compute a hidden state. Previously, we computed the hidden state as $s_t = \tanh(Ux_t + Ws_{t-1})$. The inputs to this unit were x_t, the current input at step t, and s_{t-1}, the previous hidden state. The output was a new hidden state s_t. A LSTM unit does the exact same thing, just in a different way! **This is key to understanding the big picture.** You can essentially treat LSTM (and GRU) units as a black boxes. Given the current input and previous hidden state, they compute the next hidden state in some way.

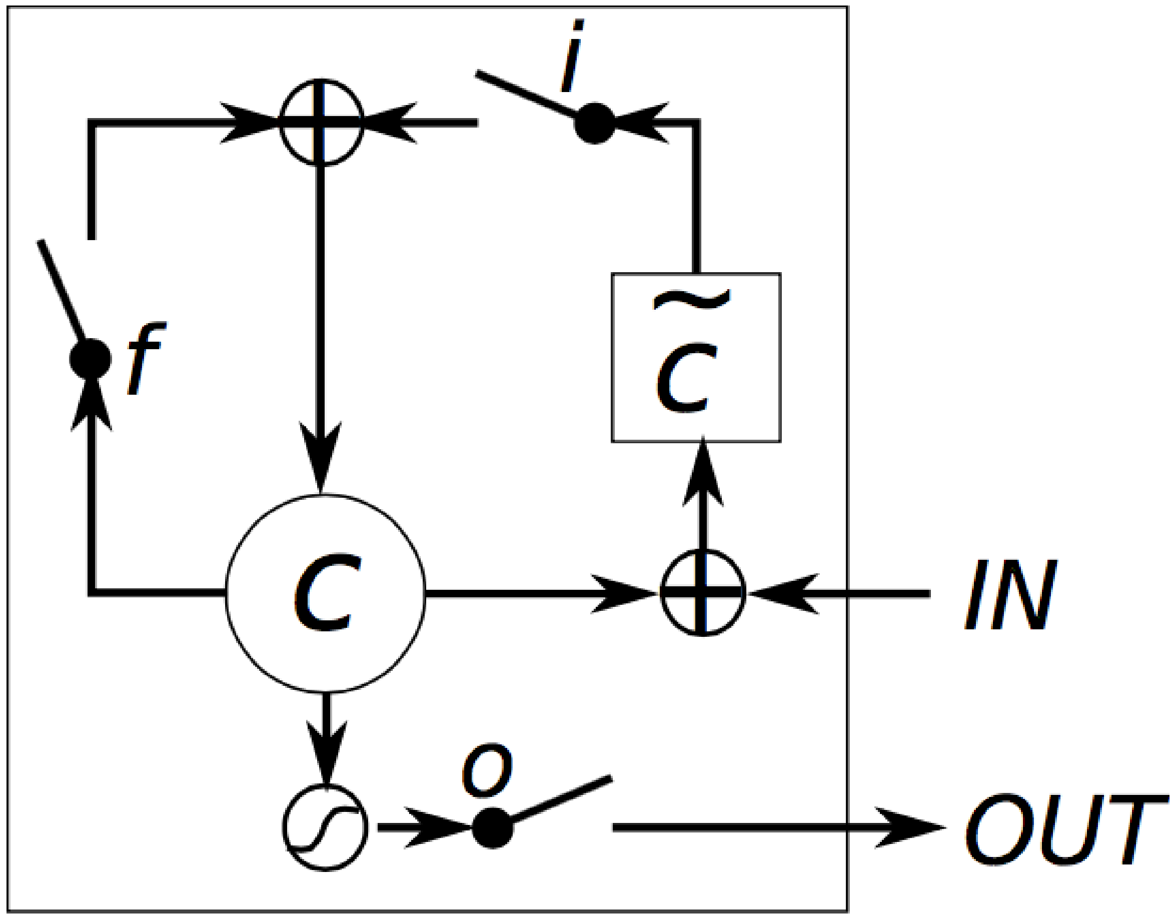


With that in mind let's try to get an intuition for *how* a LSTM unit computes the hidden state. Chris Olah has an excellent post that goes into details on this and to avoid duplicating his effort I will only give a brief explanation here. I urge you to read his post to for deeper insight and nice visualizations. But, to summarize:

- i, f, o are called the input, forget and output *gates*, respectively. Note that they have the exact same equations, just with different parameter matrices. They care called *gates* because the sigmoid function squashes the values of these vectors between 0 and 1, and by multiplying them elementwise with another vector you define how much of that other vector you want to "let through". The input gate defines how much of the newly computed state for the current input you want to let through. The forget gate defines how much of the previous state you want to let through. Finally, The output gate defines how much of the internal state you want to expose to the external network (higher layers and the next time step). All the gates have the same dimensions d_s, the size of your hidden state.
- g is a "candidate" hidden state that is computed based on the current input and the previous hidden state. It is exactly the same equation we had in our vanilla RNN, we

just renamed the parameters U and W to U^g and W^g. However, instead of taking g as the new hidden state as we did in the RNN, we will use the input gate from above to pick some of it.

- c_t is the internal memory of the unit. It is a combination of the previous memory c_{t-1} multiplied by the forget gate, and the newly computed hidden state g, multiplied by the input gate. Thus, intuitively it is a combination of how we want to combine previous memory and the new input. We could choose to ignore the old memory completely (forget gate all 0's) or ignore the newly computed state completely (input gate all 0's), but most likely we want something in between these two extremes.
- Given the memory c_t, we finally compute the output hidden state s_t by multiplying the memory with the output gate. Not all of the internal memory may be relevant to the hidden state used by other units in the network.



LSTM Gating. Chung, Junyoung, et al. "Empirical evaluation of gated recurrent neural networks on sequence modeling." (2014)

Intuitively, plain RNNs could be considered a special case of LSTMs. If you fix the input gate all 1's, the forget gate to all 0's (you always forget the previous memory) and the output gate to all one's (you expose the whole memory) you almost get standard RNN. There's just an additional $\tanh$ that squashes the output a bit. The *gating mechanism* is what allows LSTMs to explicitly model long-term dependencies. By learning the parameters for its gates, the network learns how its memory should behave.

Notably, there exist several variations on the basic LSTM architecture. A common one is creating *peephole* connections that allow the gates to not only depend on the previous hidden state s_{t-1}, but also on the previous internal state c_{t-1}, adding an additional term in the gate equations. There are many more variations. LSTM: A Search Space Odyssey empirically evaluates different LSTM architectures.

GRUS

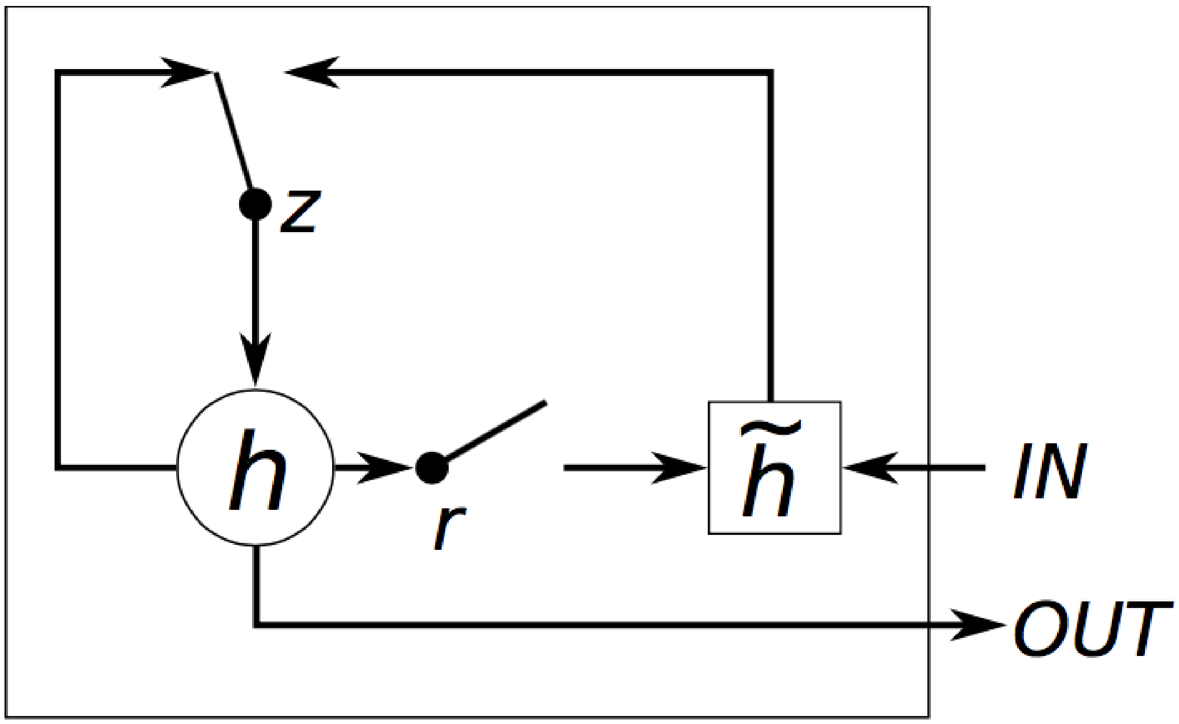
The idea behind a GRU layer is quite similar to that of a LSTM layer, as are the equations.

$$\begin{aligned}
z &= \sigma(x_t U^z + s_{t-1} W^z) \\
r &= \sigma(x_t U^r + s_{t-1} W^r) \\
h &= tanh(x_t U^h + (s_{t-1} \circ r) W^h) \\
s_t &= (1 - z) \circ h + z \circ s_{t-1}
\end{aligned}$$

A GRU has two gates, a reset gate r, and an update gate z. Intuitively, the reset gate determines how to combine the new input with the previous memory, and the update gate defines how much of the previous memory to keep around. If we set the reset to all 1's and update gate to all 0's we again arrive at our plain RNN model. The basic idea of using a gating mechanism to learn long-term dependencies is the same as in a LSTM, but there are a few key differences:

- A GRU has two gates, an LSTM has three gates.
- GRUs don't possess and internal memory (c_t) that is different from the exposed hidden state. They don't have the output gate that is present in LSTMs.
- The input and forget gates are coupled by an update gate z and the reset gate r is applied directly to the previous hidden state. Thus, the responsibility of the reset gate in a LSTM is really split up into both r and z.

- We don't apply a second nonlinearity when computing the output.



GRU Gating. Chung, Junyoung, et al. "Empirical evaluation of gated recurrent neural networks on sequence modeling." (2014)

GRU VS LSTM

Now that you've seen two models to combat the vanishing gradient problem you may be wondering: Which one to use? GRUs are quite new (2014), and their tradeoffs haven't been fully explored yet. According to empirical evaluations in Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling and An Empirical Exploration of Recurrent Network Architectures, there isn't a clear winner. In many tasks both architectures yield comparable performance and tuning hyperparameters like layer size is probably more important than picking the ideal architecture. GRUs have fewer parameters (U and W are smaller) and thus may train a bit faster or need less data to generalize. On the other hand, if you have enough data, the greater expressive power of LSTMs may lead to better results.

IMPLEMENTATION

Let's return to the implementation of the Language Model from part 2 and let's use GRU units in our RNN. There is no principled reason why I've chosen GRUs instead LSTMs in this part (other that I also wanted to become more familiar with GRUs). Their implementations are almost identical so you should be able to modify the code to go from GRU to LSTM quite easily by changing the equations.

We base the code on our previous Theano implementation. Remember that a GRU (LSTM) layer is just another way of computing the hidden state. So all we really need to do is change the hidden state computation in our forward propagation function.

```
def forward_prop_step(x_t, s_t1_prev):
      # This is how we calculated the hidden state in a simple RNN. No longer!
      # s_t = T.tanh(U[:,x_t] + W.dot(s_t1_prev))

      # Get the word vector
      x_e = E[:,x_t]

      # GRU Layer
      z_t1 = T.nnet.hard_sigmoid(U[0].dot(x_e) + W[0].dot(s_t1_prev) + b[0])
      r_t1 = T.nnet.hard_sigmoid(U[1].dot(x_e) + W[1].dot(s_t1_prev) + b[1])
      c_t1 = T.tanh(U[2].dot(x_e) + W[2].dot(s_t1_prev * r_t1) + b[2])
      s_t1 = (T.ones_like(z_t1) - z_t1) * c_t1 + z_t1 * s_t1_prev

      # Final output calculation
      # Theano's softmax returns a matrix with one row, we only need the row
      o_t = T.nnet.softmax(V.dot(s_t1) + c)[0]

      return [o_t, s_t1]
```

In our implementation we also added bias units b, c. It's quite typical that these are not shown in the equations. Of course we also need to change the initialization of our parameters U and W because they now have a different sizes. I don't show the initialization code here, but it is on Gitub. I also added a word embedding layer E, but more on that below.

That was pretty simple. But what about the gradients? We could derive the gradients for E, W, U, b and c by hand using the chain rule, just like we did before. But in practice most people use libraries like Theano that support auto-differenation of expressions. If you are for somehow forced to calculate the gradients yourself, you probably want to modularize different units and have your own version of auto-differentiation using the chain rule. We let Theano calculate the gradients for us:

```
# Gradients using Theano
dE = T.grad(cost, E)
dU = T.grad(cost, U)
dW = T.grad(cost, W)
db = T.grad(cost, b)
dV = T.grad(cost, V)
dc = T.grad(cost, c)
```

That's pretty much it. To get better results we also use a few additional tricks in our implementation.

USING RMSPROP FOR PARAMETER UPDATES

In part 2 we used the most basic version of Stochastic Gradient Descent (SGD) to update our parameters. It turns out this isn't such a great idea. If you set your learning rate low enough, SGD is guaranteed to make progress towards a good solution, but in practice that would take a very long time. There exist a number of commonly used variations on SGD, including the (Nesterov) Momentum Method, AdaGrad, AdaDelta and rmsprop. This post contains a good overview of many of these methods. I'm also planning to explore the implementation of each of these methods in detail in a future post. For this part of the tutorial I chose to go with rmsprop. The basic idea behind rmsprop is to adjust the learning rate **per-parameter** according to the a (smoothed) sum of the previous gradients. Intuitively this means that frequently occurring features get a smaller learning rate (because the sum of their gradients is larger), and rare features get a larger learning rate.

The implementation of rmsprop is quite simple. For each parameter we keep a cache variable and during gradient descent we update the parameter and the cache as follows (example for W):

```
cacheW = decay * cacheW + (1 - decay) * dW ** 2
W = W - learning_rate * dW / np.sqrt(cacheW + 1e-6)
```

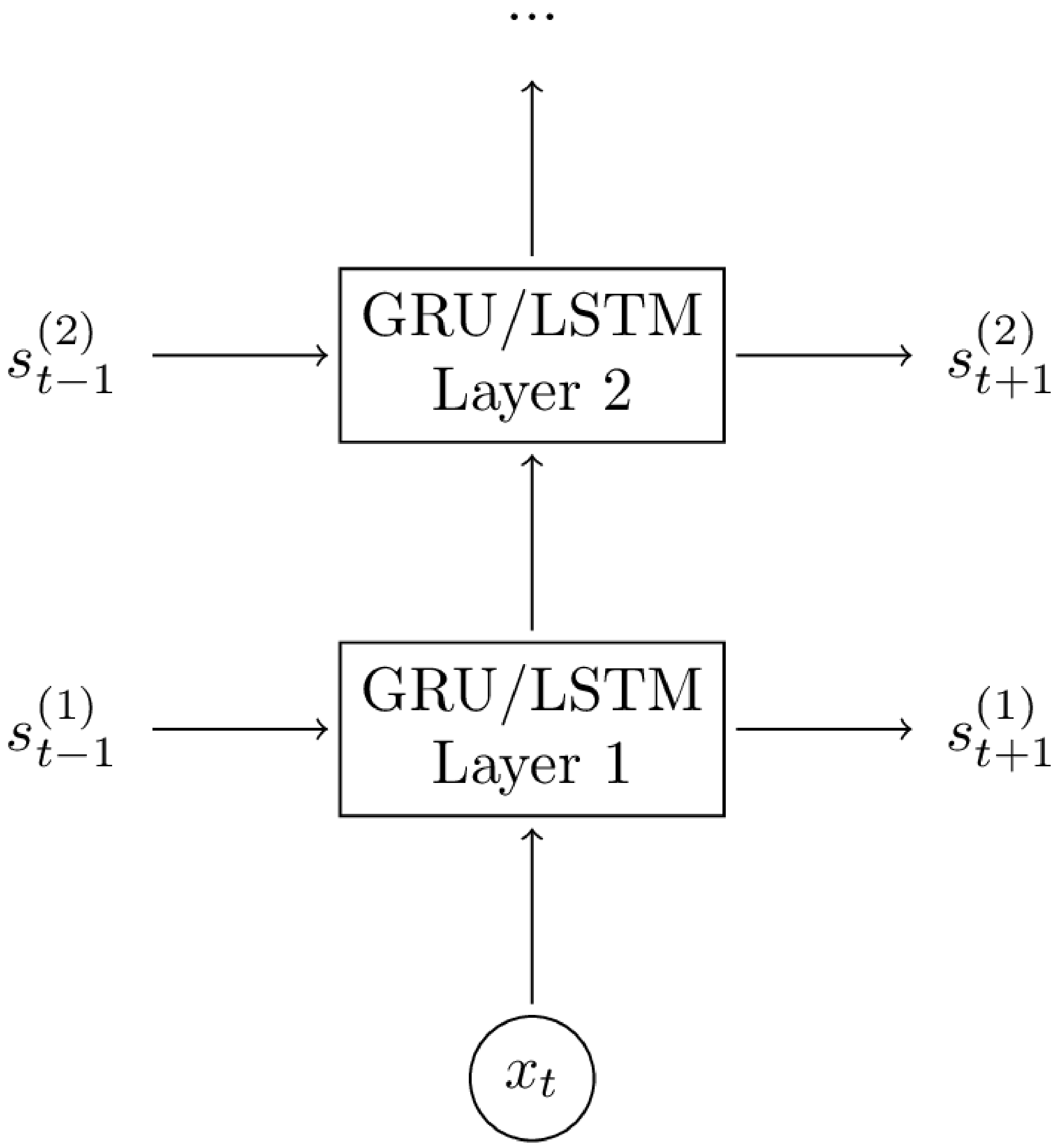
The decay is typically set to 0.9 or 0.95 and the 1e-6 term is added to avoid division by 0.

ADDING AN EMBEDDING LAYER

Using word embeddings such as word2vec and GloVe is a popular method to improve the accuracy of your model. Instead of using one-hot vectors to represent our words, the low-dimensional vectors learned using word2vec or GloVe carry semantic meaning – similar words have similar vectors. Using these vectors is a form of *pre-training.* Intuitively, you are telling the network which words are similar so that it needs to learn less about the language. Using pre-trained vectors is particularly useful if you don't have a lot of data because it allows the network to generalize to unseen words. I didn't use pre-trained word vectors in my experiments, but adding an embedding layer (the matrix E in our code) makes it easy to plug them in. The embedding matrix is really just a lookup table – the ith column vector corresponds to the ith word in our vocabulary. By updating the matrix E we are learning word vectors ourselves, but they are very specific to our task (and data set) and not as general as those that you can download, which are trained on millions or billions of documents.

ADDING A SECOND GRU LAYER

Adding a second layer to our network allows our model to capture higher-level interactions. You could add additional layers, but I didn't try that for this experiment. You'll likely see diminishing returns after 2-3 layers and unless you have a huge amount of data (which we don't) more layers are unlikely to make a big difference and may lead to overfitting.



Adding a second layer to our network is straightforward, we (again) only need to modify the forward propagation calculation and initialization function.

```
# GRU Layer 1
z_t1 = T.nnet.hard_sigmoid(U[0].dot(x_e) + W[0].dot(s_t1_prev) + b[0])
r_t1 = T.nnet.hard_sigmoid(U[1].dot(x_e) + W[1].dot(s_t1_prev) + b[1])
c_t1 = T.tanh(U[2].dot(x_e) + W[2].dot(s_t1_prev * r_t1) + b[2])
s_t1 = (T.ones_like(z_t1) - z_t1) * c_t1 + z_t1 * s_t1_prev

# GRU Layer 2
z_t2 = T.nnet.hard_sigmoid(U[3].dot(s_t1) + W[3].dot(s_t2_prev) + b[3])
```

```
    r_t2 = T.nnet.hard_sigmoid(U[4].dot(s_t1) + W[4].dot(s_t2_prev) + b[4])
    c_t2 = T.tanh(U[5].dot(s_t1) + W[5].dot(s_t2_prev * r_t2) + b[5])
    s_t2 = (T.ones_like(z_t2) - z_t2) * c_t2 + z_t2 * s_t2_prev
```

The full code for the GRU network is available here.

A NOTE ON PERFORMANCE

I've gotten questions about this in the past, so I want to clarify that the code I showed here isn't very efficient. It's optimized for clarity and was primarily written for educational purposes. It's probably good enough to play around with the model, but you should not use it in production or expect to train on a large dataset with it. There are many tricks to optimize RNN performance, but the perhaps most important one would be to batch together your updates. Instead of learning from one sentence at a time, you want to group sentences of the same length (or even pad all sentences to have the same length) and then perform large matrix multiplications and sum up gradients for the whole batch. That's because such large matrix multiplications are efficiently handled by a GPU. By not doing this we can get little speed-up from using a GPU and training can be extremely slow.

So, if you want to train a large model I highly recommended using one of the existing Deep Learning libraries that are optimized for performance. A model that would take days/weeks to train with the above code will only take a few hours with these libraries. I personally like Keras, which is quite simple to use and comes with good examples for RNNs.

RESULTS

To spare you the pain of training a model over many days I trained a model very similar to that in part 2. I used a vocabulary size of 8000, mapped words into 48-dimensional vectors, and used two 128-dimensional GRU layers. The iPython notebook contains code to load the model so you can play with it, modify it, and use it to generate text.

Here are a few good examples of the network output (capitalization added by me).

- I am a bot , and this action was performed automatically .

- I enforce myself ridiculously well enough to just youtube.
- I've got a good rhythm going !
- There is no problem here, but at least still wave !
- It depends on how plausible my judgement is .
- (with the constitution which makes it impossible)

It is interesting to look at the semantic dependencies of these sentences over multiple time steps. For example, bot and automatically are clearly related, as are the opening and closing brackets. Our network was able to learn that, pretty cool!

That's it for now. I hope you had fun and please leave questions/feedback in the comments!

Posted in: Deep Learning, Language Modeling, Neural Networks, Recurrent Neural Networks, RNNs

← *Recurrent Neural Networks Tutorial, Part 3 – Backpropagation Through Time and Vanishing Gradients*

Understanding Convolutional Neural Networks for NLP →

- CONNECT -

- RECENT POSTS -

Learning Reinforcement Learning (with Code, Exercises and Solutions)

RNNs in Tensorflow, a Practical Guide and Undocumented Features

- META -

Log in

Entries RSS

Comments RSS

WordPress.org