WILDML

Artificial Intelligence, Deep Learning, and NLP

SEPTEMBER 30, 2015 BY DENNY BRITZ

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

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')
    # Append SENTENCE_START and SENTENCE_END
    sentences = ["%s %s %s" % (sentence_start_token, x, sentence_end_toke
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 vec
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

# 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

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

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

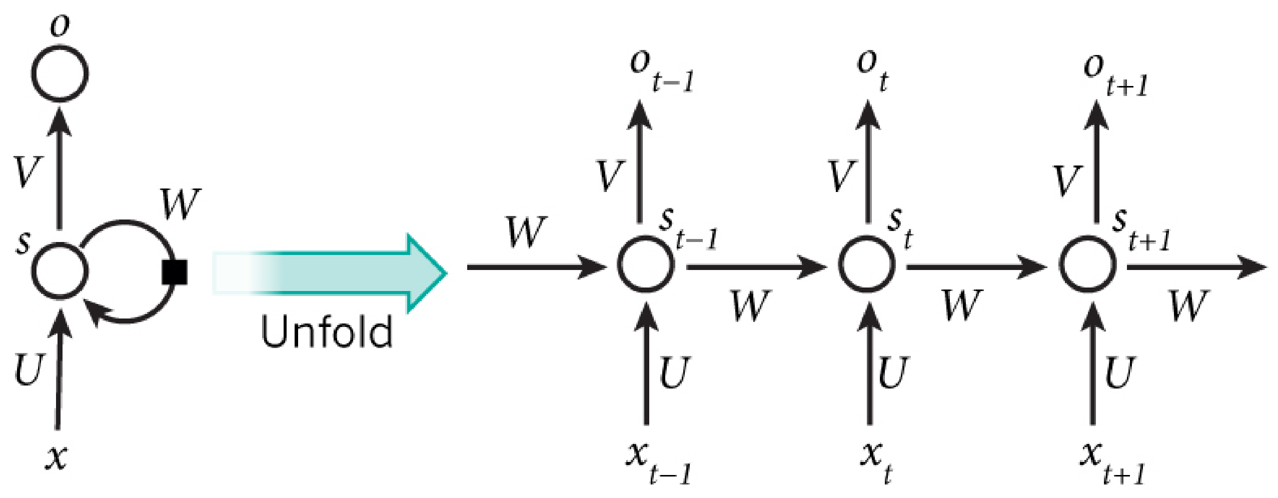
Here's an actual training example from our text:

```
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:

$x_t \in \mathbb{R}^{8000}$
$o_t \in \mathbb{R}^{8000}$
$s_t \in \mathbb{R}^{100}$
$U \in \mathbb{R}^{100\times8000}$
$V \in \mathbb{R}^{8000\times100}$
$W \in \mathbb{R}^{100\times100}$

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_
        self.V = np.random.uniform(-np.sqrt(1./hidden_dim), np.sqrt(1./hi
        self.W = np.random.uniform(-np.sqrt(1./hidden_dim), np.sqrt(1./hi
```

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 ne
    # We add one additional element for the initial hidden, which we set t
    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 multiply
        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 238 253
 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 357
```

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[:1
```

```
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 ste
        for bptt_step in np.arange(max(0, t-self.bptt_truncate), t+1)[::-
            # print "Backpropagation step t=%d bptt step=%d " % (t, bptt_
            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 i
    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." %
        # Iterate over each element of the parameter matrix, e.g. (0,0), (
        it = np.nditer(parameter, flags=['multi_index'], op_flags=['readwr
        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 backpropaga
            backprop_gradient = bptt_gradients[pidx][ix]
            # calculate The relative error: (|x - y|/(|x| + |y|))
            relative_error = np.abs(backprop_gradient - estimated_gradient
            # 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,
                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
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=1
    # 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" % (t
            # 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, e
```

```
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_thea

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.

DEEP LEARNING, GPU, LANGUAGE MODELING, RECURRENT NEURAL NETWORKS