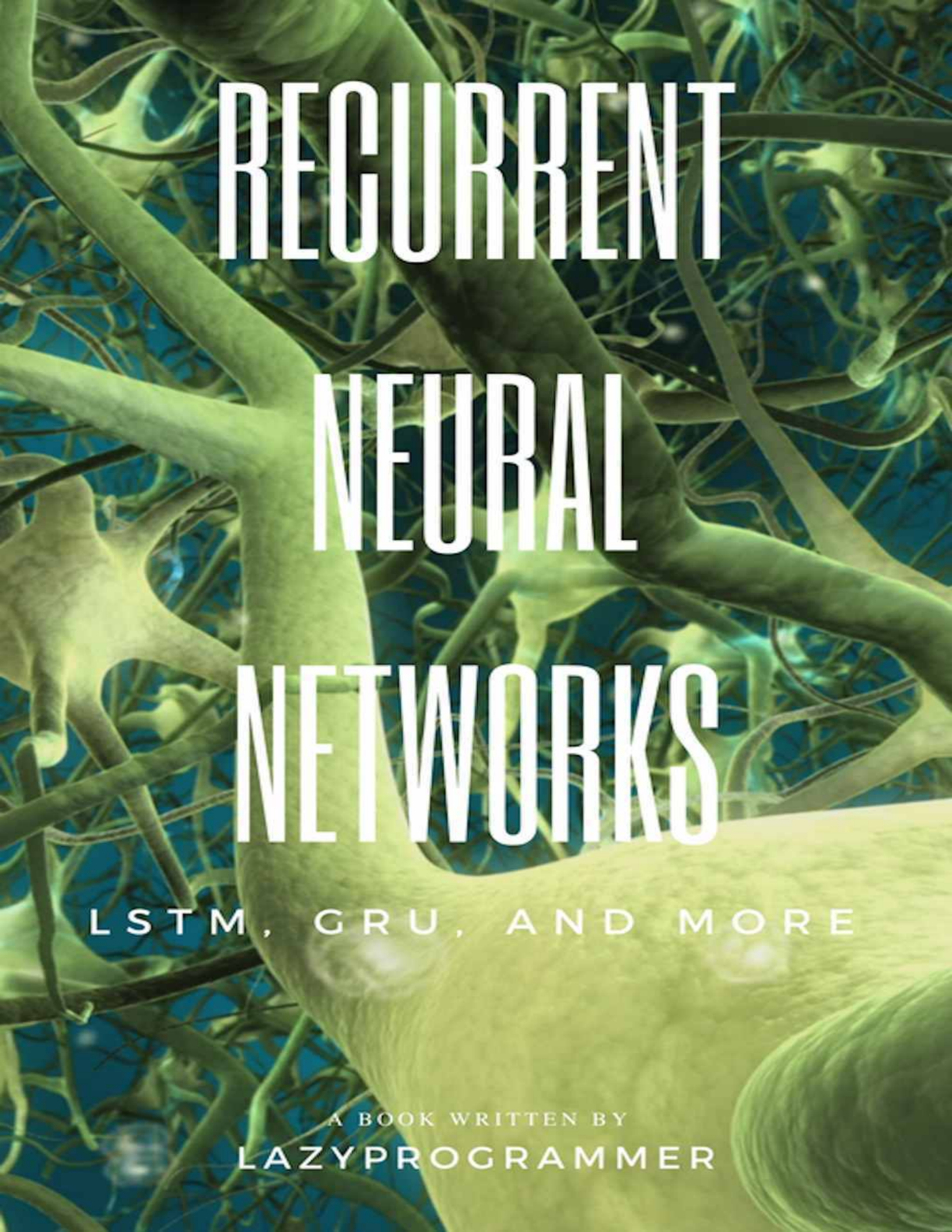

RECURRENT
NEURAL
NETWORKS
LSTM, GRU, AND MORE
A BOOK WRITTEN BY
LAZYPROGRAMMER

Deep Learning: Recurrent Neural Networks in Python

LSTM, GRU, and more RNN machine learning architectures in Python and Theano

By: The LazyProgrammer (https://lazyprogrammer.me)

Introduction

Recurrent Neural Networks are all about learning sequences.

You may have already learned about Markov models for sequence modeling, which make use of the Markov assumption, p{ x(t) | x(t-1), ..., x(1) } = p{ x(t) | x(t-1) }. In other words, the current value depends only on the last value. While easy to train, one can imagine this may not be very realistic. Ex. the previous word in a sentence is “and”, what’s the next word?

Whereas Markov Models are limited by the Markov assumption, Recurrent Neural Networks are not. As a result, they are more expressive, and more powerful than anything we’ve seen on tasks that we haven’t made progress on in decades.

In the first section of the book we are going to add time to our neural networks. I’ll introduce you to the Simple Recurrent Unit, also known as the Elman unit.

The Simple Recurrent Unit will help you understand the basics of recurrent neural networks - the types of tasks they can be used for, how to construct the objective functions for these tasks, and backpropagation through time.

We are going to revisit a classical neural network problem - the XOR problem, but we’re going to extend it so that it becomes the parity problem - you’ll see that regular feedforward neural networks will have trouble solving this problem but recurrent networks will work because the key is to treat the input as a sequence.

In the next section of the course, we are going to revisit one of the most popular applications of recurrent neural networks - language modeling, which plays a large role in natural language processing or NLP.

Another popular application of neural networks for language is word vectors or word embeddings. The most common technique for this is called

Word2Vec, but I'll show you how recurrent neural networks can also be used for creating word vectors.

In the section after, we'll look at the very popular LSTM, or long short-term memory unit, and the more modern and efficient GRU, or gated recurrent unit, which has been proven to yield comparable performance.

We'll apply these to some more practical problems, such as learning a language model from Wikipedia data and visualizing the word embeddings we get as a result.

One tip for getting through this book:

Understand the mechanics first, worry about the "meaning" later.

When we talk about LSTMs we're going to discuss its ability to "remember" and "forget" things - keep in mind these are just convenient names by way of analogy.

We're not actually building something that's "remembering" or "forgetting" - they are just mathematical formulas.

So worry about the math, and let the meaning come naturally to you.

What you especially don't want to do is the opposite - try to understand the meaning without understanding the mechanics.

When you do that, the result is usually a sensationalist media article or a pop-science book - this book is the opposite of that.

We want to understand on a technical level what's happening - explaining things in layman terms or thinking of real-life analogies is icing on the cake *only if* you understand the technicalities.

All the code for this class is hosted on Github, and you can get it from https://github.com/lazyprogrammer/machine_learning_examples in the folder rnn_class.

Git is a version control system that allows me to push updates and keep a history of all the changes.

You should always do a “git pull” to make sure you have the latest version.

Formatting

I know that the e-book format can be quite limited on many platforms. If you find the formatting in this book lacking, particularly for the code or diagrams, please shoot me an email at info@lazyprogrammer.me along with a proof-of-purchase, and I will send you the original ePub from which this book was created.

Chapter 1: The Simple Recurrent Unit

In this chapter we are going to talk about the “Simple Recurrent Unit”, also known as the “Elman Unit”. This is the most basic recurrent unit.

Sequence Representation

Before we go into the recurrent unit itself - let's talk about sequences. Why? Because these will look slightly different from what we're used to.

Recall that our input data X is usually represented with an NxD matrix. That's N samples and D features.

But that's when we weren't working with sequences.

Well let's suppose we did have a sequence. How many dimensions would that require?

Call the length of the sequence T.

If the observation is a D-dimensional vector, and we have T of them, then one sequence of observations will be a TxD matrix.

If we have N training samples, that will be an NxTxD matrix - a 3-dimensional object.

Sometimes, our sequences are not of equal length. This can happen when we're talking about sentences, which are obviously of arbitrary length, or perhaps sounds and music, or someone's credit history. How can we handle this?

Instead of a 3-D matrix, we'll have a length-N list, where each element is a 2-D observation sequence of size T(n)xD.

Since Python lists can contain any object as an element, this is ok.

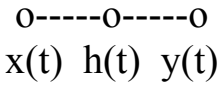
In general, we can represent our sequence as: x(1), x(2), …, x(t), …, x(T-1), x(T)

The Recurrent Unit

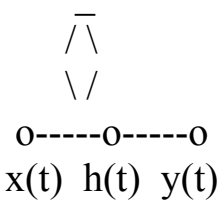
Back to our simple recurrent unit.

The way it works is this. Take a simple feedforward neural network with 1 hidden layer.

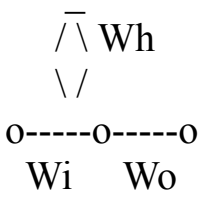
The input is a D-dimensional vector, x(t).



What we want to do is create a feedback connection from the hidden layer to itself.



Here's the picture again but showing the weights.



Notice that the feedback loop implies there's a delay of 1 time unit.

So the input into h(t) is now not just x(t), but h(t-1) also.

In math, we can represent h(t) as (assuming row vectors):

$h(t) = f(x(t)W_i + h(t-1)W_h + b_h)$

What is the size of W_h?

Just like the other layers, we connect "everything-to-everything".

Let M = number of dimensions of h(t).

So if there are M hidden units, the first hidden unit connects back to all M hidden units, the second hidden unit connects back to all M hidden units, and so on.

So in total there will be M^2 hidden-to-hidden weights.

Another way to think of this is that h(t) and h(t-1) must be the same size, therefore, W_h must be of size MxM for the equation to be valid.

Notice that the "f" function can be any one of the usual hidden layer nonlinearities - usually sigmoid, tanh, or relu.

It's a hyperparameter, just like with other types of neural networks.

Question for the reader to think about: Why does the RNN not make the Markov assumption?

Due to the recursive formulation, h(t) needs to have an initial state, h(0).

That is, assuming each sequence starts at the index 1.

Sometimes, researchers just set this to 0, and other times, they treat it as a parameter that we can use backpropagation on.

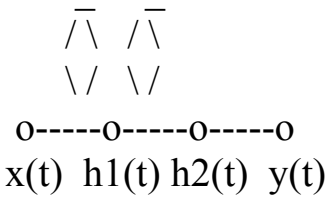
Since Theano automatically differentiates things for us, we'll treat it as an updeatable parameter.

Note that you may add multiple recurrent layers to your recurrent neural network, just like with feedforward neural networks.

The number of recurrent layers is a hyperparameter just like how the number of hidden layers is for a regular feedforward neural network.

The question of "how many" is the best depends on what's best for your specific problem.

We can do this layering for all types of the recurrent units we'll see - so that's the Elman, the GRU, and the LSTM.



And that's all there is to it!

Just by adding that one recurrent connection, we've already created a recurrent neural network.

We'll see in the coding lectures how this can already do some amazing things, like exponentially decrease the number of hidden units we would have needed in a feedforward neural network.

Prediction and Relation to Markov Models

In this section we are going to look more closely at what a recurrent neural network can predict and talk about how, under certain circumstances, we can relate it to what we know about Markov models.

Adding a time component gives us a few more options in terms of the objective, or in other words, what we're trying to predict.

Let's start with a simple example.

Suppose you wanted to classify between male and female voices.

Each sound sample would thus correspond to a single label.

Out of every data point in the sequence, x(1), ..., x(T), you would probably want to consider the output at time T, since that's the one that takes into account the whole sequence.

But don't forget that for every h(t) there is a y(t).

y(t) is just the final layer calculated from h(t). $p\{y(t) \mid x(t)\} = \text{softmax}(h(t)W_o + b_o)$ (assuming row vectors).

Let's try to imagine some situations where this might be useful.

Think about brain-computer interfaces.

These are systems that are constantly reading electrical signals from your brain. Suppose the purpose of this brain-computer interface is for controlling a wheelchair.

Considering that the wheelchair is how you would get around, you'll need fine-grain control on the device.

So it moves forward when you want it to move forward, stops when you want it to stop, turns left when you want it to turn left. These would be the

different output classes.

Ideally you could control this motion in real-time. So the action happens *as you think of it*.

Then we would need a y(t) for every x(t), so this is precisely a situation where we would want to have not just one label for one sequence, but one label for every moment in time.

Let's now think of a third situation.

Suppose we're looking at sequences of words, or in other words, sentences.

As is typical, we want to predict the next word given all the previous words.

What's interesting about that is that it's essentially unsupervised learning, because there is no target label, the target is just the input, similar to autoencoders.

In other words, we're trying to model the probability:

$P\{x(t) \mid x(t-1), x(t-2), \ldots, x(1)\}$

Remember, this is not the Markov assumption.

Now what happens when we try to make the whole input sequence the target sequence?

We are then trying to maximize the following probabilities:
$P\{x(1)\}$
$P\{x(2)|x(1)\}$
$P\{x(3)|x(2), x(1)\}$
$P\{x(4)|x(3), x(2), x(1)\}$
...
$P\{x(t)|x(t-1), ..., x(1)\}$

What happens if we join the first two terms together? Using Bayes rule, we get:

$p\{x(2),x(1)\} = P\{x(2)|x(1)\}\ P\{x(1)\}$

What happens if we join the first three terms together? We get:

$P\{x(3), x(2), x(1)\} = P\{x(3)|x(2), x(1)\}\ P\{x(2)|x(1)\}\ P\{x(1)\}$

Remember this is called the "chain rule" of probability.

Eventually what we end up with is just the joint probability of the sequence!

Recall that this is also what we try to optimize when using Hidden Markov Models (in my Hidden Markov Models class), except we run into the underflow problem as the probability approaches 0.

With recurrent nets, we optimize this joint probability implicitly, so we never have to actually calculate it, thus avoiding any underflow.

In addition, recurrent nets need not make the Markov assumption.

This could lead us to the intuition that these recurrent neural networks might be more powerful.

So to conclude, we've identified 3 different ways of using recurrent neural networks for prediction.

1) We can predict a label over an entire sequence. The example we used was differentiating between male and female voice samples.

2) We can predict a label for every step of an input sequence. The example we used was controlling an accessibility device using a brain-computer interface.

3) We can predict the next value in a sequence, given the past values of a sequence. The example we used was learning to predict the next word in a

sentence.

Backpropagation Through Time

In this section we are going to return to the idea of backpropagation.

This is just a mental exercise only - in code we're going to use Theano and TensorFlow to calculate the gradients and do our neural network training.

In my previous books and courses, we learned that "backpropagation" is actually just a fancy name for gradient descent.

It has some interesting properties, but the method behind it was exactly the same as what we did with logistic regression, just simply calculating the gradient and moving in that direction.

Similarly, Backpropagation Through Time, usually abbreviated as BPTT, is just a fancy name for backpropagation, which itself is just a fancy name for gradient descent.

So what does this mean for us?

That means in the code, updating all the weights is going to look exactly the same, which makes things very easy for us.

However, we're still interested in taking the gradient.

First, let's consider what our neural network output would be calculated as (assuming column vectors):

$$y(t)=\text{softmax}(W_o^T h(t))$$

$$y(t)=\text{softmax}(W_o^T f(W_h^T h(t-1)+W_x^T x(t)))$$

$$y(t)=\text{softmax}(W_o^T f(W_h^T f(W_h^T h(t-2)+W_x^T x(t-1))+W_x^T x(t)))$$

$$y(t)=\text{softmax}(W_o^T f(W_h^T f(W_h^T f(W_h^T h(t-3)+W_x^T x(t-2))+W_x^T x(t-1))+W_x^T x(t)))$$

I've dropped the biases to make the pattern more obvious.

The output weight of course occurs after the recurrence, so we don't need to consider time in that case.

But let's say we wanted to go back in time 3 steps for W_h. We see that W_h occurs multiple times! So we need to be very careful taking our derivative.

I'll assume you already know how to take the derivative of the cross-entropy and softmax, which we did in Deep Learning part 1, so the important part to focus on here is what happens after that.

The important thing to realize is that there is a product where both terms depend on W_h, so you need to use the product rule from calculus.

$d[W_h^T h(t-1)] / dW_h$

We won't dive too deep into this problem since we won't ever have to use the solution, but instead I just wanted to outline a strategy for solving it if you ever wanted to.

One important pattern you want to try to see is that the same things are going to be multiplied together over and over again, due to the chain rule of calculus.

This will happen for both the hidden-to-hidden weights as well as the input-to-hidden weights.

The result is that you'll either get something that goes down to 0, or something that gets very large very quickly.

These problems are called the vanishing gradient problem and the exploding gradient problem, respectively.

One solution that has been proposed for the vanishing gradient problem is gradient clipping.

Gradient clipping works as follows. Let g = gradient of cost wrt variable of interest, and |g| be its L2 norm.

if |g| > threshold:
  g = threshold * g / |g|

If you're used to adding things like momentum and adaptive learning rates to your backpropagation this should be a pretty simple change in your Theano code.

Another modification to backpropagation through time is Truncated backpropagation through time.

Because the derivatives with respect to W_h and W_x depend on every single time step of the sequence so far, the calculation will take a very long time for very long sequences.

One common approximation is just to stop after a certain number of time steps.

One disadvantage of this is that it won't incorporate the error at longer periods of time. But if you don't care about dependencies past, say, 3 time steps, then you can just truncate at 3 time steps.

An alternative would just be to never train with sequences longer than T=3.

Chapter 2: The Parity Problem

In this chapter we are going to go back to our old friend, the XOR problem, but with a new twist.

Recall that the XOR problem is simply to implement the XOR logic gate using a neural network.

The result is a 0 if the 2 inputs are the same, and the result is 1 if the 2 inputs are different.

00 -> 0
01 -> 1
10 -> 1
11 -> 0

One obvious way to extend this problem is to add more inputs.

The label is 1 if the number of bits that are 1 is odd, and the label is 0 if the number of bits that are 1 is even.

000 -> 0
001 -> 1
010 -> 1
011 -> 0
100 -> 1
101 -> 0
110 -> 0
111 -> 1

Parity appears in a few places in computer science.

One application is data transmission in communication systems.

If you imagine the source is trying to send a bitstream to a receiver, but one of the bits gets reversed via noise - how can we detect this?

One simple way to do this is to add a parity bit to the end of a message. We set a requirement that each message must have an even number of 1 bits let's say. Then if the actual message has an odd number of 1s, we'll make the parity bit a 1 to make the total even.

If the actual message already has an even number of 1s, we'll make the parity bit a 0 so that the total is still even.

This way, when the receiver gets the message, it can check how many 1s there are, and if that's odd, then we know something has gone wrong in the transmission of the message.

You'll notice that when you try to solve the parity problem with a regular feedforward neural network, the number of hidden units required is going to grow very fast compared to the number of bits.

Think about the geometry of the problem to understand why that is.

Try it on your own for D=12 bits. You'll need thousands of hidden units.

That's a lot of hidden units for such a small data problem.

We can mitigate that problem somewhat by creating deeper networks with less units per layer.

Now think about how you would solve this problem using recurrent neural networks.

The basic idea is we would *like* the neural network to keep track of its state.

So if the output is 0, and we see another 1 in the input, we would like to switch to a 1.

If the output is currently a 1, and we see another 1 in the input, we would like to switch to a 0.

If we see a 0 in the input, we would like to just maintain whatever our current output is.

That's what we would *like* to happen - but can the neural network actually learn to do this? I'll show you that it can.

Of our 3 different ways to construct the error or objective - which is this?

This is of course the version where we care about the output at every point in time, just like with the brain-computer interface.

To do this, we'll have to discard the original label, which is just 1 number, and create a sequence of labels, which will be a sequence of the same size as the input that just keeps a tally of whether it's seen an even or odd number of ones so far.

Let's first write the code to generate the data:

```
def all_parity_pairs(nbit):
  N = 2**nbit
  X = np.zeros((N, nbit))
  Y = np.zeros(N)
  for i in xrange(N):
    for j in xrange(nbit):
      if i % (2**(j+1)) != 0:
        i -= 2**j
        X[i,j] = 1
    Y[i] = X[i].sum() % 2
  return X, Y
```

Our simple RNN can be defined as follows:

```
import theano
import theano.tensor as T
import numpy as np
import matplotlib.pyplot as plt
```

```
class SimpleRNN:
  def __init__(self, M):
    self.M = M # hidden layer size

  def fit(self, X, Y, learning_rate=10e-5, mu=0.99, reg=1.0,
activation=T.tanh, epochs=200, show_fig=False):
    D = X[0].shape[1] # X is of size N x T(n) x D
    K = len(set(Y.flatten()))
    N = len(Y)
    M = self.M
    self.f = activation

    # initial weights
    Wx = init_weight(D, M)
    Wh = init_weight(M, M)
    bh = np.zeros(M)
    h0 = np.zeros(M)
    Wo = init_weight(M, K)
    bo = np.zeros(K)

    # make them theano shared
    self.Wx = theano.shared(Wx)
    self.Wh = theano.shared(Wh)
    self.bh = theano.shared(bh)
    self.h0 = theano.shared(h0)
    self.Wo = theano.shared(Wo)
    self.bo = theano.shared(bo)
    self.params = [self.Wx, self.Wh, self.bh, self.h0, self.Wo, self.bo]

    thX = T.fmatrix('X')
    thY = T.ivector('Y')

    def recurrence(x_t, h_t1):
      # returns h(t), y(t)
      h_t = self.f(x_t.dot(self.Wx) + h_t1.dot(self.Wh) + self.bh)
      y_t = T.nnet.softmax(h_t.dot(self.Wo) + self.bo)
      return h_t, y_t
```

```
    [h, y], _ = theano.scan(
      fn=recurrence,
      outputs_info=[self.h0, None],
      sequences=thX,
      n_steps=thX.shape[0],
    )

    py_x = y[:, 0, :]
    prediction = T.argmax(py_x, axis=1)

    cost = -T.mean(T.log(py_x[T.arange(thY.shape[0]), thY]))
    grads = T.grad(cost, self.params)
    dparams = [theano.shared(p.get_value()*0) for p in self.params]

    updates = [
      (p, p + mu*dp - learning_rate*g) for p, dp, g in zip(self.params,
dparams, grads)
    ] + [
      (dp, mu*dp - learning_rate*g) for dp, g in zip(dparams, grads)
    ]

    self.predict_op = theano.function(inputs=[thX], outputs=prediction)
    self.train_op = theano.function(
      inputs=[thX, thY],
      outputs=[cost, prediction, y],
      updates=updates
    )

    costs = []
    for i in xrange(epochs):
      X, Y = shuffle(X, Y)
      n_correct = 0
      cost = 0
      for j in xrange(N):
        c, p, rout = self.train_op(X[j], Y[j])
        cost += c
```

```
        if p[-1] == Y[j,-1]:
          n_correct += 1
      print "i:", i, "cost:", cost, "classification rate:", (float(n_correct)/N)
      costs.append(cost)

    if show_fig:
      plt.plot(costs)
      plt.show()
```

We go through a Theano scan tutorial in my course if you don't let know how it works.

Finally, let's write a function to get the data and fit our SimpleRNN to it:

```
def parity(B=12):
  X, Y = all_parity_pairs(B)
  N, t = X.shape

  # we want every time step to have a label
  Y_t = np.zeros(X.shape, dtype=np.int32)
  for n in xrange(N):
    ones_count = 0
    for i in xrange(t):
      if X[n,i] == 1:
        ones_count += 1
      if ones_count % 2 == 1:
        Y_t[n,i] = 1

  X = X.reshape(N, t, 1).astype(np.float32)

  rnn = SimpleRNN(4)
  rnn.fit(X, Y_t, activation=T.nnet.sigmoid, show_fig=True)
```

On Adding Complexity

You've seen that we can use a simple recurrent unit to solve the parity problem with just one layer.

Of course the students in this course are interested in learning about GRUs and LSTMs, however, there is one important point to consider.

If a simpler model already solves your problem - then you don't *need* to use a more complex model.

If you do, you're going to add more training time, and possibly run the risk of overfitting.

LSTMs are much slower than simple recurrent units and they have many more components.

You always want to test against simpler models as benchmarks.

It's interesting to note that deep learning is not a common technique for Kaggle contests. Many of the contestants do a lot of feature engineering and use simpler models.

Deep learning touches a lot of different but related fields, including machine learning, computational neuroscience, and artificial intelligence - so that's what makes it interesting.

At the same time, for basic tasks, a very simple and fast model might be the best choice.

Chapter 3: Recurrent Neural Networks for NLP

NLP, or natural language processing, is basically machine learning for text, speech, and language.

It's tightly coupled with recurrent neural networks, and a lot of the RNN examples you'll see here and elsewhere use word sequences as examples, for several reasons.

First, language is an easy topic to comprehend. You use language all the time, whether you are speaking, reading, or writing.

You must understand language because you use it on a daily basis, and you're using those inherent abilities right now to understand this book.

Second, recurrent neural networks finally give us a way to avoid having to treat sentences as a bag of words.

A lot of examples and tutorials you'll see treat sentences as a bag of words, and what happens is you lose a lot of information that is vital to the meaning of the sentence.

For example, consider the sentence:

"Dogs love cats and I."

This sentence actually almost has correct grammatical structure, but its meaning is much different from the original sentence, which was: "I love dogs and cats."

So there is a lot of information, in the quantitative sense, that you throw away when you use bag of words.

To be clear, what do we mean by bag-of-words?

If you've taken my first NLP course or my logistic regression course, I did an example where we used logistic regression to perform sentiment

analysis.

The task is to be able to guess whether a sentence is positive or negative.

A positive sentence might be “Today is a great day!”, and a negative sentence might be, “This is the worst movie I’ve ever seen.”

To turn each sentence into an input vector for the classifier, we first start with a vector of 0s of size V, which is the vocabulary size. So there’s an entry for every individual word.

We would keep track of which word goes with which index using a dictionary.

Now, for every word in the sentence, we’ll set the corresponding index in the vector to 1, or perhaps some other frequency measure.

So there’s a nonzero value in the vector for every word that appears in the sentence, and everywhere else it’s 0.

You can see how given this vector, it wouldn’t be easy to determine the correct order of words in the sentence.

It’s not completely impossible, if the words in the sentence are such that there’s only one possible ordering, but generally speaking, there is some information lost.

Now, what happens when you have a sentence, “Today is a good day” versus “Today is not a good day”?

Well these both lead to almost the same vector. Bag-of-words models are known for not being able to handle negation.

You can imagine that a recurrent net would be ideal for this task - because they keep state - so you might be able to see how if the RNN saw a “not”, it would then negate everything that comes after it.

So how are words usually treated in deep learning?

The popular method at the moment, which has produced impressive results, is the use of word embeddings, or word vectors.

That means, given a vocabulary size V, we choose a dimensionality that is much smaller than that, call it D, and then map each word vector to somewhere in the D-dimensional space.

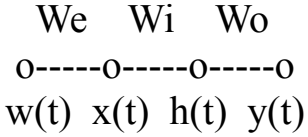
The word embedding matrix is thus a VxD matrix.

By training the model to do certain things, like try to predict the next word, or try to predict the surrounding words, we get word vectors can be manipulated using arithmetic to produce analogies, such as:

King - man ~= Queen - woman

How do we use word embeddings with recurrent neural networks?

To accomplish this, we simply create an embedding layer in the RNN.



In the figure above, the recurrent connection at h(t) is implicit.

w(t) represents a V dimensional one-hot encoded vector, and x(t) is a D dimensional vector.

2 questions arise here.

1) How do we train this model? The answer is of course gradient descent. Simply include W_e as a parameter when you do backpropagation with Theano. Another alternative is pretraining the word embedding using a method like word2vec or GLoVe.

2) What are the targets? This is a very good question, because language models don't necessarily have targets. You can attempt to learn word embeddings on a sentiment analysis task, so your targets could be movie ratings or some kind of sentiment score, or you can also do next-word prediction like we discussed earlier.

Representing a sequence of words as a sequence of word embeddings

In this section I am going to explain one small detail in the code that might be confusing.

We have a word embedding matrix W_e, which of size VxD, and we would like to get a sequence of word vectors that represent a sentence, which is a TxD vector.

The word embeddings must be part of the neural network, so that they can be updated via gradient descent with all the other weights.

This means the input to the neural network will just be a sequence of word indexes, and the indexes correspond to however we build our word2idx dictionary.

This also saves a lot of space because now we can represent each input by a Tx1 vector of ints, rather than a TxD matrix of floats.

Conceptually speaking, what we're trying to implement is the following for loop, where we take each word index in the input sequence, grab its corresponding word vector, and add it to a list of word vectors which is the output sequence.

```
list_of_word_vectors = []
for word_idx in sentence:
  v = We[word_idx, :]
  list_of_word_vectors.append(v)
```

Mathematically speaking, the way you would get a word vector is by multiplying your one-hot encoded word index vector by the word embedding matrix.

$x(t) = w(t)W_e$

You get a 1xV vector multiplied by a VxD matrix which results in a 1xD vector, as expected.

But since all of the one-hotted vectors can only have one location that's a one, and the rest must be 0, we can take a shortcut.

The key is to realize that multiplying the word embedding by a one-hot vector just gives you the row where that word embedding vector lives.

Therefore, we can say each row of the word embedding matrix represents a corresponding word. This should be pretty obvious already given that the word embedding matrix has V rows.

Furthermore, Numpy and Theano allow you to retrieve a row from a matrix very easily - simply index it like an array.

Not only that, but they allow us to index arrays using arrays.

E.g.

$x(t) = We[w(t)]$
$x(1), ..., x(T) = We[[w(1), ..., w(T)]]$

As an exercise, try this with Numpy arrays to prove to yourself that they can be indexed in this way.

Finding Word Analogies

In this section we are going to talk about how you can actually do calculations like show king - man + woman = queen.

It's quite simple but worth going through anyway.

I will describe it in 2 steps:

1) is to convert all 3 of the words on the left to their word embeddings, or word vectors. Once they're in vector form, you can subtract and add very easily.

Remember that we can just grab the word's corresponding word vector by indexing the word embedding matrix with the index of the word.

2) is to find the closest actual word in our vocabulary to the equation on the left.

Why is that? Because the result of king - man + woman just gives us a vector - there's no way to map from vectors to words, since a vector space is continuous and that would require an infinite number of words.

So the idea is we just find the closest word.

Now there are various ways of defining distance.

Sometimes you see just the plain squared distance used.

It is also common to use the cosine distance. In this latter form, since only the angle matters, during training we normalize all the word vectors so that their length is 1.

Then we can say that all the word embeddings lie on the unit sphere.

Once we have our distance function, how do we find the word? The simplest way is just to look at every word in the vocabulary, and get the

distance between each vector and your expression vector.

Keep track of the smallest distance and then return that word.

You may want to leave out the other words from the left side of the equation, namely king, man, and woman.

Pseudocode:

```
v1 = We[ word2idx[“king”] ]
v2 = We[ word2idx[“man”] ]
v4 = We[ word2idx[“woman”] ]
v3 = v1 - v2 + v4
best_word = None
best_dist = float(“inf”)
for word, idx in word2idx.iteritems():
  if word not in (“king”, “man”, “woman”):
    v = We[ word2idx[word] ]
    d = dist(v, v3)
    if d < best_dist:
      best_dist = d
      best_word = word
print v1, “-“, v2, “=“, v3, “-“, v4
```

Chapter 4: Generating and Classifying Poetry

In this chapter I'm going to give you an overview of how we'll use an RNN to create a language model and then generate poetry.

As we've discussed, this is an unsupervised model, and our softmax output is the probability of the next word given all the previous words in a line.

Because this is a language model, we'll also need word embeddings.

So this RNN is going to be a little different than the RNN we built for the parity problem.

So to enumerate the parameters:
We (word embedding of size VxD)
Wx (input-to-hidden weights of size DxM)
Wh (hidden-to-hidden weights of size MxM)
Wo (hidden-to-output weights of size MxK)

Wx, Wh, and Wo will have corresponding bias terms, and we are assuming 1 hidden layer.

Another difference is that the fit function will only take in an X, since there are no targets.

Within the fit function, we will however create our own targets. The target for words 1 to t-1 should be the word at time t.

But we also need to predict the end of a line, otherwise we would just create infinitely long lines.

So we'll make the target for the full sequence the END token.

Similarly, we'll add a START token at the beginning of the input sequence and its target will be the first word.

To summarize, the input sequence will be *prepended* with the start token, and the output sequence will be *appended* with the end token.

Unlike the parity problem, we want to measure our accuracy rate by every predicted word, not just the last word.

To that end, will accumulate the number of correct words guessed, and divide it by the total number of words, NOT the total number of sentences, to get the final accuracy.

Next, because we might want to generate new poetry without training the model every time we do it, we'll want to save our model after it's trained, and also have a way to load the saved model.

Here's the API for that:

def save(self, filename)

@staticmethod
def load(filename)

Notice one is a static method and the other is an instance method.

Because Theano functions need to be compiled, you can't simply just set the weights to the saved numpy arrays either - we'll need to re-initialize the object with all the required Theano functions in order to make predictions.

Let's discuss the data we'll be looking at. We're going to try to generate poetry by learning from Robert Frost poems (already provided in the course repo).

We've got about 1500 lines of Robert Frost, and we'll treat each line as a sequence.

The basic algorithm is, for each line, lowercase all the text, remove all the punctuation, and split by whitespace to get a list of tokens.

Then for each token, if it's not in our word2idx map, add it and give it an index.

Then we'll save each sentence as a sequence of word indexes, and return both that and the word2idx map.

This is generally the same process we'll follow for building any sort of language model, but you'll see how we can introduce further modifications when we look at more complicated datasets.

Finally, the code for this exercise is in srn_language.py in the course repo.

Generating Poetry Code

```
import theano
import theano.tensor as T
import numpy as np
import matplotlib.pyplot as plt

from sklearn.utils import shuffle
from util import init_weight, get_robert_frost, get_wikipedia_data


class SimpleRNN:
    def __init__(self, D, M, V):
        self.D = D # dimensionality of word embedding
        self.M = M # hidden layer size
        self.V = V # vocabulary size

    def fit(self, X, learning_rate=10e-1, mu=0.99, reg=1.0, activation=T.tanh, epochs=500,
show_fig=False):
        N = len(X)
        D = self.D
        M = self.M
        V = self.V
        self.f = activation

        # initial weights
        We = init_weight(V, D)
        Wx = init_weight(D, M)
        Wh = init_weight(M, M)
        bh = np.zeros(M)
        h0 = np.zeros(M)
        Wo = init_weight(M, V)
        bo = np.zeros(V)

        # make them theano shared
        self.We = theano.shared(We)
        self.Wx = theano.shared(Wx)
        self.Wh = theano.shared(Wh)
        self.bh = theano.shared(bh)
        self.h0 = theano.shared(h0)
        self.Wo = theano.shared(Wo)
        self.bo = theano.shared(bo)
        self.params = [self.We, self.Wx, self.Wh, self.bh, self.h0, self.Wo, self.bo]

        thX = T.ivector('X')
        Ei = self.We[thX] # will be a TxD matrix
        thY = T.ivector('Y')

        # sentence input:
```

```
# [START, w1, w2, ..., wn]
# sentence target:
# [w1,    w2, w3, ..., END]

def recurrence(x_t, h_t1):
    # returns h(t), y(t)
    h_t = self.f(x_t.dot(self.Wx) + h_t1.dot(self.Wh) + self.bh)
    y_t = T.nnet.softmax(h_t.dot(self.Wo) + self.bo)
    return h_t, y_t

[h, y], _ = theano.scan(
    fn=recurrence,
    outputs_info=[self.h0, None],
    sequences=Ei,
    n_steps=Ei.shape[0],
)

py_x = y[:, 0, :]
prediction = T.argmax(py_x, axis=1)

cost = -T.mean(T.log(py_x[T.arange(thY.shape[0]), thY]))
grads = T.grad(cost, self.params)
dparams = [theano.shared(p.get_value()*0) for p in self.params]

updates = [
    (p, p + mu*dp - learning_rate*g) for p, dp, g in zip(self.params, dparams, grads)
] + [
    (dp, mu*dp - learning_rate*g) for dp, g in zip(dparams, grads)
]

self.predict_op = theano.function(inputs=[thX], outputs=prediction)
self.train_op = theano.function(
    inputs=[thX, thY],
    outputs=[cost, prediction],
    updates=updates
)

costs = []
n_total = sum((len(sentence)+1) for sentence in X)
for i in xrange(epochs):
    X = shuffle(X)
    n_correct = 0
    cost = 0
    for j in xrange(N):
        # problem! many words --> END token are overrepresented
        # result: generated lines will be very short
        # we will try to fix in a later iteration
        # BAD! magic numbers 0 and 1...
        input_sequence = [0] + X[j]
        output_sequence = X[j] + [1]
```

```
            # we set 0 to start and 1 to end
            c, p = self.train_op(input_sequence, output_sequence)
            cost += c
            for pj, xj in zip(p, output_sequence):
                if pj == xj:
                    n_correct += 1
        print "i:", i, "cost:", cost, "correct rate:", (float(n_correct)/n_total)
        costs.append(cost)

    if show_fig:
        plt.plot(costs)
        plt.show()

def save(self, filename):
    np.savez(filename, *[p.get_value() for p in self.params])

@staticmethod
def load(filename, activation):
    # TODO: would prefer to save activation to file too
    npz = np.load(filename)
    We = npz['arr_0']
    Wx = npz['arr_1']
    Wh = npz['arr_2']
    bh = npz['arr_3']
    h0 = npz['arr_4']
    Wo = npz['arr_5']
    bo = npz['arr_6']
    V, D = We.shape
    _, M = Wx.shape
    rnn = SimpleRNN(D, M, V)
    rnn.set(We, Wx, Wh, bh, h0, Wo, bo, activation)
    return rnn

def set(self, We, Wx, Wh, bh, h0, Wo, bo, activation):
    self.f = activation

    # redundant - see how you can improve it
    self.We = theano.shared(We)
    self.Wx = theano.shared(Wx)
    self.Wh = theano.shared(Wh)
    self.bh = theano.shared(bh)
    self.h0 = theano.shared(h0)
    self.Wo = theano.shared(Wo)
    self.bo = theano.shared(bo)
    self.params = [self.We, self.Wx, self.Wh, self.bh, self.h0, self.Wo, self.bo]

    thX = T.ivector('X')
    Ei = self.We[thX] # will be a TxD matrix
    thY = T.ivector('Y')
```

```
        def recurrence(x_t, h_t1):
            # returns h(t), y(t)
            h_t = self.f(x_t.dot(self.Wx) + h_t1.dot(self.Wh) + self.bh)
            y_t = T.nnet.softmax(h_t.dot(self.Wo) + self.bo)
            return h_t, y_t

        [h, y], _ = theano.scan(
            fn=recurrence,
            outputs_info=[self.h0, None],
            sequences=Ei,
            n_steps=Ei.shape[0],
        )

        py_x = y[:, 0, :]
        prediction = T.argmax(py_x, axis=1)
        self.predict_op = theano.function(
            inputs=[thX],
            outputs=prediction,
            allow_input_downcast=True,
        )

    def generate(self, word2idx):
        # convert word2idx -> idx2word
        idx2word = {v:k for k,v in word2idx.iteritems()}
        V = len(word2idx)

        # generate 4 lines at a time
        n_lines = 0

        X = [ 0 ]
        while n_lines < 4:
            PY_X, _ = self.predict_op(X)
            PY_X = PY_X[-1].flatten()
            P = [ np.random.choice(V, p=PY_X)]
            X = np.concatenate([X, P]) # append to the sequence
            P = P[-1] # just grab the most recent prediction
            if P > 1:
                # it's a real word, not start/end token
                word = idx2word[P]
                print word,
            elif P == 1:
                # end token
                n_lines += 1
                X = [0]
                print ''


def train_poetry():
    # students: tanh didn't work but you should try it
```

```
    sentences, word2idx = get_robert_frost()
    rnn = SimpleRNN(30, 30, len(word2idx))
    rnn.fit(sentences, learning_rate=10e-5, show_fig=True, activation=T.nnet.relu, epochs=2000)
    rnn.save('RNN_D30_M30_epochs2000_relu.npz')

def generate_poetry():
    sentences, word2idx = get_robert_frost()
    rnn = SimpleRNN.load('RNN_D30_M30_epochs2000_relu.npz', T.nnet.relu)
    rnn.generate(word2idx)


if __name__ == '__main__':
    train_poetry()
    generate_poetry()
```

Classifying Poetry

In this section we are going to look at a problem we first encountered in my Hidden Markov Model class - discriminating between Robert Frost and Edgar Allan Poe poems given only sequences of parts-of-speech tags.

We were able to get around 60-70% accuracy on the validation set using HMMs, and we'll try to do better with RNNs (> 90%).

First, since the data processing part of this exercise is a little more complex than usual, we'll discuss that here.

I mentioned parts-of-speech tags. These tags are basically things like noun, adverb, adjective, and so on, that tell us what the role of each word is in a sentence.

So given a sentence, or, a sequence of words, we can get a sequence of POS tags of the same length.

To do this we'll use a library called NLTK, which stands for natural language toolkit.

If you've already taken my Easy NLP class, you should already have it installed, but just in case you don't, it's very easy, just use "sudo pip install nltk". There may be some external packages you'll need to install that NLTK will require, but it will prompt you to do so if it needs them, in addition to giving you instructions for how to do it. It's a very simple command, just nltk.download() inside the IPython console.

Next, we just do the same thing as we did in the generating poetry exercise - turn each sequence of POS tags into a sequence of indexes that represent those POS tags.

I've added some caching ability since NLTK's POS tagging and word tokenization is extremely slow, and you have the ability to input the number of desired samples per class. All it does is save the data, targets, and vocabulary size to a numpy blob.

Note that we don't actually need the POS tag to index mapping since we don't care about what the actual POS tags are, we just need to be able to differentiate them in order to do classification.

```
from nltk import pos_tag, word_tokenize

def get_tags(s):
    tuples = pos_tag(word_tokenize(s))
    return [y for x, y in tuples]

def get_poetry_classifier_data(samples_per_class, load_cached=True,
save_cached=True):
    datafile = 'poetry_classifier_data.npz'
    if load_cached and os.path.exists(datafile):
        npz = np.load(datafile)
        X = npz['arr_0']
        Y = npz['arr_1']
        V = int(npz['arr_2'])
        return X, Y, V

    word2idx = {}
    current_idx = 0
    X = []
    Y = []
    for fn, label in zip(('../hmm_class/edgar_allan_poe.txt',
'../hmm_class/robert_frost.txt'), (0, 1)):
        count = 0
        for line in open(fn):
            line = line.rstrip()
            if line:
                print line
                # tokens = remove_punctuation(line.lower()).split()
                tokens = get_tags(line)
                if len(tokens) > 1:
                    # scan doesn't work nice here, technically could fix...
                    for token in tokens:
```

```
                if token not in word2idx:
                    word2idx[token] = current_idx
                    current_idx += 1
            sequence = np.array([word2idx[w] for w in tokens])
            X.append(sequence)
            Y.append(label)
            count += 1
            print count
            # quit early because the tokenizer is very slow
            if count >= samples_per_class:
                break
    if save_cached:
        np.savez(datafile, X, Y, current_idx)
    return X, Y, current_idx
```

Next let's look at the classifier itself. It's again going to be slightly different than the previous RNNs we built.

Even though this is sort of a language model, it's not going to have any word embeddings since we're not using words. The one-hot vector representing the POS tag goes straight to the recurrent unit.

Note that this means Wx will now be VxM instead of DxM, there's no D anymore since there are no word embeddings.

The number of output classes is now 2 instead of V since we're not predicting a word, we're predicting the poet.

Of our 3 ways of doing prediction using RNNs you should recognize this as the first one - one class per sequence.

This means we only care about the classification at the end of the sequence, or in other words, after we have seen the entire sequence.

One more small note - I'm going to use a variable learning rate here because I noticed the cost jumping around a lot in the later epochs.

To do this I’m just going to make the learning rate smaller by a factor of 0.9999 after every epoch.

Classifying Poetry Code

Note: The relevant file in the class repo is poetry_classifier.py

```
import theano
import theano.tensor as T
import numpy as np
import matplotlib.pyplot as plt

from sklearn.utils import shuffle
from util import init_weight, get_poetry_classifier_data


class SimpleRNN:
    def __init__(self, M, V):
        self.M = M # hidden layer size
        self.V = V # vocabulary size

    def fit(self, X, Y, learning_rate=10e-1, mu=0.99, reg=1.0, activation=T.tanh, epochs=500,
show_fig=False):
        M = self.M
        V = self.V
        K = len(set(Y))

        X, Y = shuffle(X, Y)
        Nvalid = 10
        Xvalid, Yvalid = X[-Nvalid:], Y[-Nvalid:]
        X, Y = X[:-Nvalid], Y[:-Nvalid]
        N = len(X)

        # initial weights
        Wx = init_weight(V, M)
        Wh = init_weight(M, M)
        bh = np.zeros(M)
        h0 = np.zeros(M)
        Wo = init_weight(M, K)
        bo = np.zeros(K)

        thX, thY, py_x, prediction = self.set(Wx, Wh, bh, h0, Wo, bo, activation)

        cost = -T.mean(T.log(py_x[thY]))
        grads = T.grad(cost, self.params)
        dparams = [theano.shared(p.get_value()*0) for p in self.params]
        lr = T.scalar('learning_rate')

        updates = [
            (p, p + mu*dp - lr*g) for p, dp, g in zip(self.params, dparams, grads)
        ] + [
            (dp, mu*dp - lr*g) for dp, g in zip(dparams, grads)
```

```
        ]

        self.train_op = theano.function(
            inputs=[thX, thY, lr],
            outputs=[cost, prediction],
            updates=updates,
            allow_input_downcast=True,
        )

        costs = []
        for i in xrange(epochs):
            X, Y = shuffle(X, Y)
            n_correct = 0
            cost = 0
            for j in xrange(N):
                c, p = self.train_op(X[j], Y[j], learning_rate)
                cost += c
                if p == Y[j]:
                    n_correct += 1
            # update the learning rate
            learning_rate *= 0.9999

            # calculate validation accuracy
            n_correct_valid = 0
            for j in xrange(Nvalid):
                p = self.predict_op(Xvalid[j])
                if p == Yvalid[j]:
                    n_correct_valid += 1
            print "i:", i, "cost:", cost, "correct rate:", (float(n_correct)/N),
            print "validation correct rate:", (float(n_correct_valid)/Nvalid)
            costs.append(cost)

        if show_fig:
            plt.plot(costs)
            plt.show()

    def save(self, filename):
        np.savez(filename, *[p.get_value() for p in self.params])

    @staticmethod
    def load(filename, activation):
        # TODO: would prefer to save activation to file too
        npz = np.load(filename)
        Wx = npz['arr_0']
        Wh = npz['arr_1']
        bh = npz['arr_2']
        h0 = npz['arr_3']
        Wo = npz['arr_4']
        bo = npz['arr_5']
        V, M = Wx.shape
```

```
        rnn = SimpleRNN(M, V)
        rnn.set(Wx, Wh, bh, h0, Wo, bo, activation)
        return rnn

    def set(self, Wx, Wh, bh, h0, Wo, bo, activation):
        self.f = activation

        # redundant - see how you can improve it
        self.Wx = theano.shared(Wx)
        self.Wh = theano.shared(Wh)
        self.bh = theano.shared(bh)
        self.h0 = theano.shared(h0)
        self.Wo = theano.shared(Wo)
        self.bo = theano.shared(bo)
        self.params = [self.Wx, self.Wh, self.bh, self.h0, self.Wo, self.bo]

        thX = T.ivector('X')
        thY = T.iscalar('Y')

        def recurrence(x_t, h_t1):
            # returns h(t), y(t)
            h_t = self.f(self.Wx[x_t] + h_t1.dot(self.Wh) + self.bh)
            y_t = T.nnet.softmax(h_t.dot(self.Wo) + self.bo)
            return h_t, y_t

        [h, y], _ = theano.scan(
            fn=recurrence,
            outputs_info=[self.h0, None],
            sequences=thX,
            n_steps=thX.shape[0],
        )

        py_x = y[-1, 0, :] # only interested in the final classification of the sequence
        prediction = T.argmax(py_x)
        self.predict_op = theano.function(
            inputs=[thX],
            outputs=prediction,
            allow_input_downcast=True,
        )
        return thX, thY, py_x, prediction


def train_poetry():
    X, Y, V = get_poetry_classifier_data(samples_per_class=500)
    rnn = SimpleRNN(30, V)
    rnn.fit(X, Y, learning_rate=10e-7, show_fig=True, activation=T.nnet.relu, epochs=1000)

if __name__ == '__main__':
    train_poetry()
```

Chapter 5: Advanced RNN Units - GRU and LSTM

Rated Recurrent Unit

In this section I'm going to show you a very simple modification to the simple recurrent unit. It will help us bridge the gap to GRU and LSTM.

The idea is we want to weight 2 things:
1) f(x, h(t-1)), which was the output we would've gotten in a simple recurrent unit, and
2) h(t-1), the previous value of the state.

$h\_hat(t) = f(x(t)W_x + h(t-1)W_h + b_h)$
$z(t) = sigmoid(x(t)W_{xz} + h(t-1)W_{hz} + b_z)$
$h(t) = (1 - z(t)) * h(t-1) + z(t) * h\_hat(t)$

z(t) is known as the rate.

You can infer the sizes of the new weights by the equations (we use * to represent element-wise multiplication).

Wxz is DxM
Whz is MxM
bz is size M

Notice how this looks a lot like the low-pass filter we built in the Theano scan tutorial (free on YouTube, but also included in my course) - so that should give you some intuition about the effect of this unit.

If you've been following along with the code you can probably imagine that this is a very simple change - just add the new params and include the above equation in the recurrence function.

RRNN in Code (only the recurrence function needs to change + obviously the weights need to be initialized):

```
def recurrence(x_t, h_t1):
  # returns h(t), y(t)
  hhat_t = self.f(x_t.dot(self.Wx) + h_t1.dot(self.Wh) + self.bh)
```

```
z_t = T.nnet.sigmoid(x_t.dot(self.Wxz) + h_t1.dot(self.Whz) + self.bz)
h_t = (1 - z_t) * h_t1 + z_t * hhat_t
y_t = T.nnet.softmax(h_t.dot(self.Wo) + self.bo)
return h_t, y_t
```

Gated Recurrent Unit

In this section we are going to talk about a more powerful recurrent unit than the rated recurrent unit. It's called the GRU or gated recurrent unit.

Gated recurrent units were introduced in 2014, whereas LSTMs were introduced in 1997.

Usually when you learn about recurrent networks you learn about the LSTM first - which could be due to the fact that it's way more popular, or possibly because it was invented first.

You can think of the GRU as a simpler version of the LSTM. It incorporates a lot of the same concepts, but it has a much smaller number of parameters, and so it can train faster at a constant hidden layer size. At the same time, it's more complex than the rated recurrent unit, which we just discussed.

So these lectures, instead of introducing the LSTM unit first, just go in order of complexity.

Recent research has also shown that the accuracy between the LSTM and GRU is comparable and even better with the GRU in some cases.

So there isn't any hard rule that you should choose one over the other. It's just like how you would choose the best nonlinearity for a regular neural network - you just have to try and see what works better for your particular data.

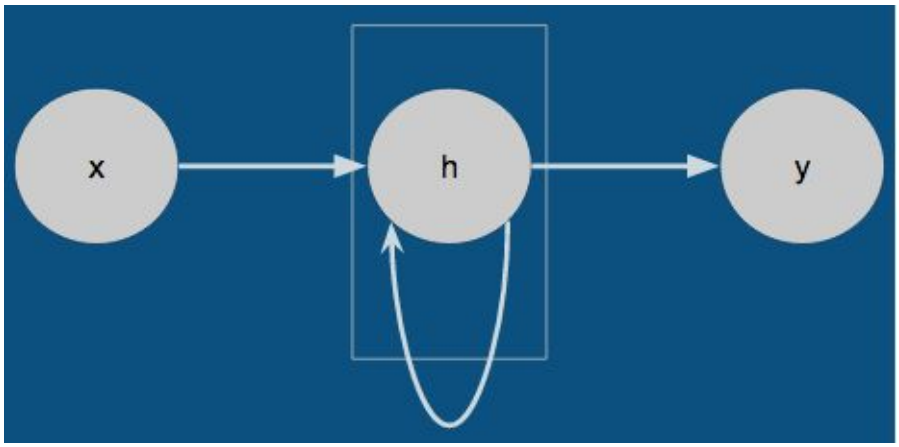
Let's describe the architecture of the GRU.

The first thing we want to do is take a compartmental point of view.

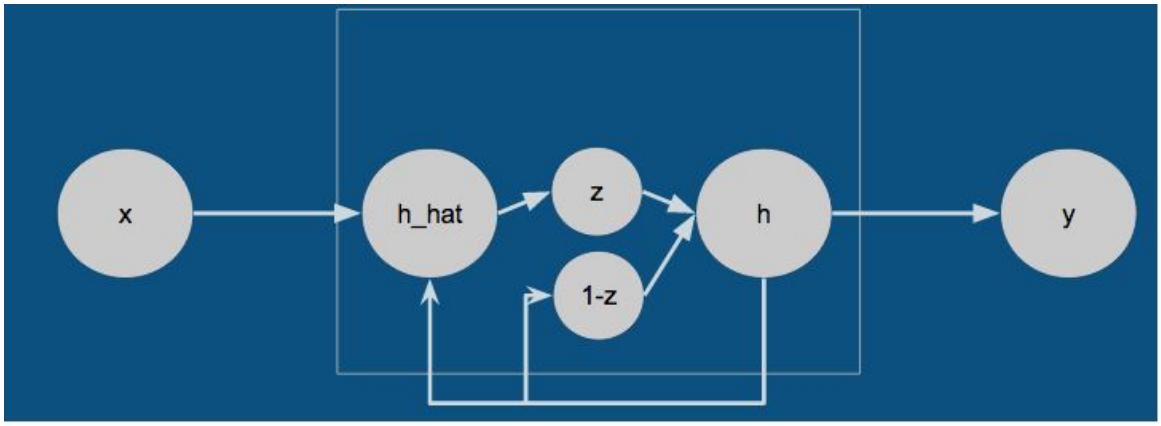
Think of everything between the previous layer and the next layer as a black box.

In the simplest feedforward neural network, this black box just contains some nonlinear function like tanh or relu.

In a simple recurrent network, we just connect the output of the black box back to itself, with a time delay of one.



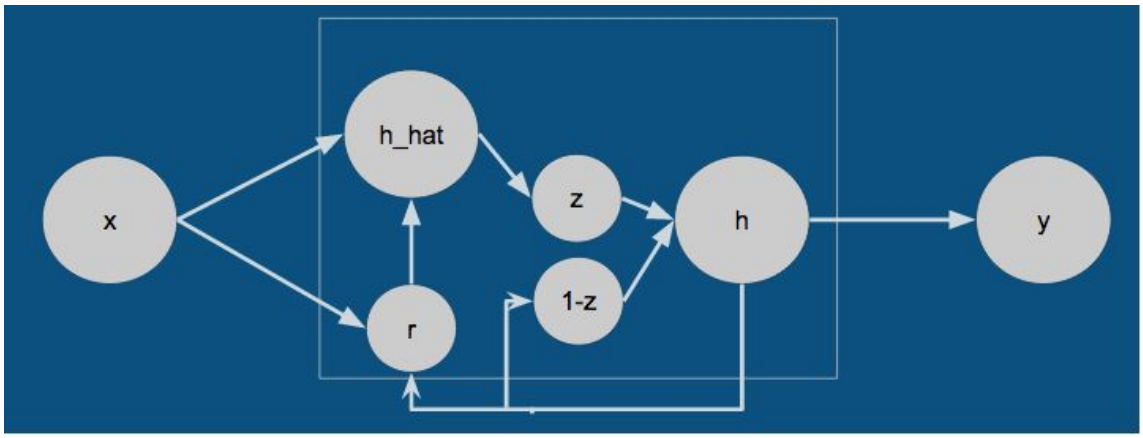
In the rated recurrent network, we add a rating operation between what would've been the output of the simple recurrent network, and the previous output value.



You can think of this new operation as a gate. Since it has to take on a value between 0 and 1, and the other gate has to take on 1 minus that value, it's a gate that is choosing between 2 things - taking on the old value, or taking on the new value.

The result is we get a mixture of both.

Now let's go to the gated recurrent unit or GRU. The architecture simply requires that we add one more gate.



For some it's easier to understand with a picture, for others it's easier to understand with a formula. Notice there is no new math here, just more of the same stuff we already know about - weight matrices multiplied by inputs and passed through nonlinear functions and gates.

$$
\begin{aligned}
r_t &= \sigma(x_t W_{xr} + h_{t-1} W_{hr} + b_r) \\
z_t &= \sigma(x_t W_{xz} + h_{t-1} W_{hz} + b_z) \\
\hat{h}_t &= g\left(x_t W_{xh} + (r_t \odot h_{t-1}) W_{hh} + b_h\right) \\
h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \hat{h}_t.
\end{aligned}
$$

Note that the circle with the dot in it is another way of representing element-wise multiplication.

Here we see again the "update gate" z(t) that we saw in the rated recurrent unit. It still balances how much of the previous hidden value and how much of the new candidate hidden value combines to get the new hidden value.

The extra thing here is r(t), or the "reset gate". It has the exact same functional form as the "update gate", and all of its weights are the same

size, but its position in the black box is different.

The “reset gate” is multiplied by the previous hidden state value - it controls how much of the previous hidden state we will consider when we create the new candidate hidden value.

In other words, it has the ability to “reset” the hidden value.

If r(t) = 0, then we get h_hat(t) = f(x(t) Wxh + bh), which would be as if x(t) were the beginning of a new sequence.

Note that this is not the full picture since h_hat(t) is only a candidate for the new h(t), since h(t) will be a combination of h_hat(t) and h(t-1), controlled by the “update gate” z(t).

If this all sounds very complicated, another way to think of this is more pragmatically. What’s happening is we’re simply adding more parameters to our model and making it more expressive.

Adding more parameters allows us to fit more complex patterns.

In terms of code, writing a GRU should be relatively trivial if you already understand the simple recurrent unit and the rated recurrent unit. We just have to add more weights and modify the recurrence.

The next step to making our code better is to modularize it.

I talked about the GRU being a black box.

In my previous classes, we put the hidden layer and convolutional pooling layer in a class, so that we could re-use them and stack them.

With the GRU we will do the same - make it into a class, so that it can be “abstracted away”.

By doing this, you’ll be able to stack GRUs, and anywhere you could have put a vanilla hidden layer you can now insert a GRU.

It’s just a “thing” that takes an input and produces an output - the fact that it contains a memory of previous inputs is just an internal detail of the black box.

GRU in Code

```
import numpy as np
import theano
import theano.tensor as T

from util import init_weight


class GRU:
    def __init__(self, Mi, Mo, activation):
        self.Mi = Mi
        self.Mo = Mo
        self.f  = activation

        # numpy init
        Wxr = init_weight(Mi, Mo)
        Whr = init_weight(Mo, Mo)
        br  = np.zeros(Mo)
        Wxz = init_weight(Mi, Mo)
        Whz = init_weight(Mo, Mo)
        bz  = np.zeros(Mo)
        Wxh = init_weight(Mi, Mo)
        Whh = init_weight(Mo, Mo)
        bh  = np.zeros(Mo)
        h0  = np.zeros(Mo)

        # theano vars
        self.Wxr = theano.shared(Wxr)
        self.Whr = theano.shared(Whr)
        self.br  = theano.shared(br)
        self.Wxz = theano.shared(Wxz)
        self.Whz = theano.shared(Whz)
        self.bz  = theano.shared(bz)
        self.Wxh = theano.shared(Wxh)
        self.Whh = theano.shared(Whh)
        self.bh  = theano.shared(bh)
        self.h0  = theano.shared(h0)
        self.params = [self.Wxr, self.Whr, self.br, self.Wxz, self.Whz, self.bz, self.Wxh, self.Whh,
self.bh, self.h0]

    def recurrence(self, x_t, h_t1):
        r = T.nnet.sigmoid(x_t.dot(self.Wxr) + h_t1.dot(self.Whr) + self.br)
        z = T.nnet.sigmoid(x_t.dot(self.Wxz) + h_t1.dot(self.Whz) + self.bz)
        hhat = self.f(x_t.dot(self.Wxh) + (r * h_t1).dot(self.Whh) + self.bh)
        h = (1 - z) * h_t1 + z * hhat
        return h

    def output(self, x):
```

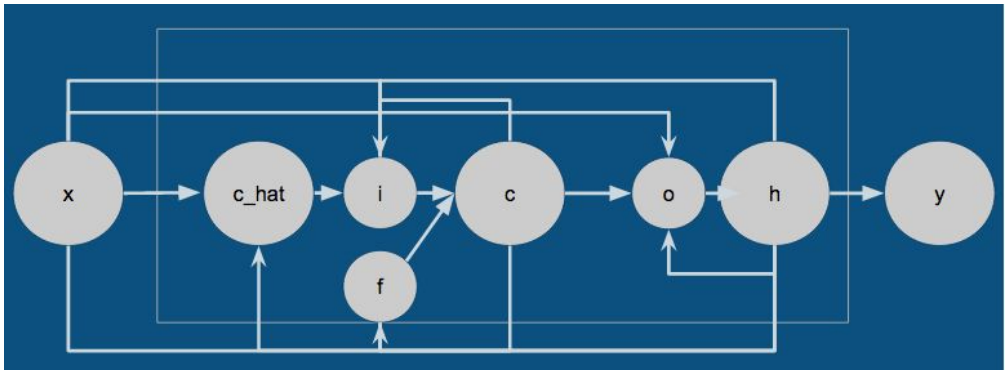
```
# input X should be a matrix (2-D)
# rows index time
h, _ = theano.scan(
    fn=self.recurrence,
    sequences=x,
    outputs_info=[self.h0],
    n_steps=x.shape[0],
)
return h
```

LSTM

In this section we are going to go from GRU to LSTM. LSTM stands for "long short-term memory" and it's a type of recurrent unit that has become very popular in recent years due to its superior performance and the fact that it doesn't as easily succumb to the vanishing gradient problem. You'll see that all the recurrent units you've learned about up until now got progressively more complex, and each incorporated concepts from the previous unit.

The LSTM is the most complex recurrent unit you'll learn about in this book - however - that "complexity" is not in the way of a hard-to-understand idea or hard math, it's just that the number of components is higher.

The gist of it is: we will now have 3 different gates, and we're going to add yet another "internal unit", that will exist alongside the hidden state, which is sometimes called a "memory cell" or just a "cell".



The 3 gates are called the input gate, the output gate, and the forget gate.

Again, some may find looking at the formulas easier to understand:

$$
\begin{aligned}
i_t &= \sigma(x_t W_{xi} + h_{t-1} W_{hi} + c_{t-1} W_{ci} + b_i) \\
f_t &= \sigma(x_t W_{xf} + h_{t-1} W_{hf} + c_{t-1} W_{cf} + b_f) \\
c_t &= f_t c_{t-1} + i_t \tanh(x_t W_{xc} + h_{t-1} W_{hc} + b_c) \\
o_t &= \sigma(x_t W_{xo} + h_{t-1} W_{ho} + c_t W_{co} + b_o) \\
h_t &= o_t \tanh(c_t)
\end{aligned}
$$

One way to think of the cell is that it takes the place of what we called "h_hat" or the "candidate hidden value" when we talked about the GRU.

The input gate and forget gate should remind you of the "rate gate" or "update gate" from the GRU.

Before we used z(t) and 1 - z(t) but now we just have 2 separate gates.

The input gate controls how much of the new value goes into the cell, and the forget gate controls how much of the previous cell value goes into the current cell value.

The candidate for the new cell value looks a lot like what would be the simple recurrent unit's value, right before it gets multiplied by the input gate.

Finally, the output gate takes into account everything - the input at time t, the previous hidden state, and the current cell value.

The new hidden state is just the tanh of the cell value multiplied by the output gate.

It might be useful to look at all the parameters of this model, because there are many.

It will helps us organize them so that it doesn't seem so complex.

For the input gate, we have Wxi, Whi, Wci, and bi, which map x(t), h(t-1) and c(t-1) to the input gate.

For the forget gate, we have matching parameters, so we have Wxf, Whf, Wcf, and bf, which again map x(t), h(t-1), and c(t-1) to the forget gate.

Next, to calculate the new cell value, we have Wxc, Whc, and bc, which map x(t) and h(t-1) to the new candidate cell value.

Finally we have the output gate, which adds 4 more parameters, Wxo, Who, Wco, and bo, which map x(t), h(t-1), and c(t) to the output gate.

Notice that the output gate is different from the input and forget gates in that it depends on the cell value at time t, not t-1.

In total, that's 15 new weights and biases for this black box. We've come a long way from the feedforward neural network, which has just 1 weight and 1 bias.

If you find the complexity overwhelming, I'll mention again that another way to think of this is that we're simply just adding more parameters, which is making the model more expressive.

Similar to the GRU, what we're going to do in the code is create a class and make the LSTM modular. I'll show you that since they are both black boxes, they can be plugged in and out interchangeably and used for the same problems.

At this point, you should be VERY familiar with how to create these recurrent units. We are again just adding some more variables and updating the recurrence function. In fact you should be able to take the formulas I presented and implement the LSTM class yourself.

I would highly recommend doing so before moving on to the code.

In the next chapter we will build a new model for Wikipedia data, treating the GRU and LSTM as "plug and play" components.

LSTM in Code

```
import numpy as np
import theano
import theano.tensor as T

from util import init_weight


class LSTM:
    def __init__(self, Mi, Mo, activation):
        self.Mi = Mi
        self.Mo = Mo
        self.f  = activation

        # numpy init
        Wxi = init_weight(Mi, Mo)
        Whi = init_weight(Mo, Mo)
        Wci = init_weight(Mo, Mo)
        bi  = np.zeros(Mo)
        Wxf = init_weight(Mi, Mo)
        Whf = init_weight(Mo, Mo)
        Wcf = init_weight(Mo, Mo)
        bf  = np.zeros(Mo)
        Wxc = init_weight(Mi, Mo)
        Whc = init_weight(Mo, Mo)
        bc  = np.zeros(Mo)
        Wxo = init_weight(Mi, Mo)
        Who = init_weight(Mo, Mo)
        Wco = init_weight(Mo, Mo)
        bo  = np.zeros(Mo)
        c0  = np.zeros(Mo)
        h0  = np.zeros(Mo)

        # theano vars
        self.Wxi = theano.shared(Wxi)
        self.Whi = theano.shared(Whi)
        self.Wci = theano.shared(Wci)
        self.bi  = theano.shared(bi)
        self.Wxf = theano.shared(Wxf)
        self.Whf = theano.shared(Whf)
        self.Wcf = theano.shared(Wcf)
        self.bf  = theano.shared(bf)
        self.Wxc = theano.shared(Wxc)
        self.Whc = theano.shared(Whc)
        self.bc  = theano.shared(bc)
        self.Wxo = theano.shared(Wxo)
        self.Who = theano.shared(Who)
        self.Wco = theano.shared(Wco)
```

```
        self.bo  = theano.shared(bo)
        self.c0  = theano.shared(c0)
        self.h0  = theano.shared(h0)
        self.params = [
            self.Wxi,
            self.Whi,
            self.Wci,
            self.bi,
            self.Wxf,
            self.Whf,
            self.Wcf,
            self.bf,
            self.Wxc,
            self.Whc,
            self.bc,
            self.Wxo,
            self.Who,
            self.Wco,
            self.bo,
            self.c0,
            self.h0,
        ]

    def recurrence(self, x_t, h_t1, c_t1):
        i_t = T.nnet.sigmoid(x_t.dot(self.Wxi) + h_t1.dot(self.Whi) + c_t1.dot(self.Wci) + self.bi)
        f_t = T.nnet.sigmoid(x_t.dot(self.Wxf) + h_t1.dot(self.Whf) + c_t1.dot(self.Wcf) + self.bf)
        c_t = f_t * c_t1 + i_t * T.tanh(x_t.dot(self.Wxc) + h_t1.dot(self.Whc) + self.bc)
        o_t = T.nnet.sigmoid(x_t.dot(self.Wxo) + h_t1.dot(self.Who) + c_t.dot(self.Wco) + self.bo)
        h_t = o_t * T.tanh(c_t)
        return h_t, c_t

    def output(self, x):
        # input X should be a matrix (2-D)
        # rows index time
        [h, c], _ = theano.scan(
            fn=self.recurrence,
            sequences=x,
            outputs_info=[self.h0, self.c0],
            n_steps=x.shape[0],
        )
        return h
```

Chapter 6: Learning from Wikipedia Data

In this chapter we are going to talk about how we're going to create a model for the Wikipedia data dumps.

The goal is to create a language model just like we did for poetry, both are just sequences of words. The difference is that the Wikipedia dataset is much larger and has many more words.

You'll find that there is not much new here in terms of RNN concepts, just that we're going to plug the GRU and LSTM instead of using a simple recurrent unit like before.

Most of the work will go into getting and processing the data. Then it's just a matter of creating a language model, which we have already done.

First let's talk about how you're going to get the data.

You'll want to go to https://dumps.wikimedia.org/ and click on "Database Data Dumps".

Look for enwiki and we're interested in the files that have "pages-articles" in the name. You can download the entire data dump which is about 12 or 13 GB or you can download it in parts.

Note that you won't need ALL the parts to obtain a meaningful result. When I visualize the data later I'm just going to be using the first part.

The data is in bz2 format but instead of directly unzipping it we're going to use a tool to extract the data in a more convenient format, since it's currently in XML.

The next step is to get the data into a flat file format. To do this we're going to use a tool called wp2txt.

Just go to https://github.com/yohasebe/wp2txt and follow the instructions.

To install it, you'll want to use the command "sudo gem install wp2txt".

Next, go to the folder "large_files", which should be adjacent to the rnn_class folder, and put the bz2 file in there. Then run the command "wp2txt -i <the filename>".

It should output text files into the same folder.

Finally let's talk about how we are going to take these text files and get it into the right format for our neural network.

This code is of course inside util.py, where all our other data gathering code is. The relevant function is get_wikipedia_data.

This function takes in 2 parameters: n_files and n_vocab.

We take in n_files because there are going to be a lot of input files - too many to fit into memory if you use all the data. I've been limiting it to 100 or less, and that seems to be enough to learn a meaningful representation of the data.

You also want to limit the vocabulary, which is a lot larger than the poetry dataset. We're going to have on the order of 500,000 and over 1 million words.

Remember that the output target is the next word, so that's 1 million output classes, which is a lot of output classes. This will make it hard to get good accuracy, and it's also going to make our output weight huge.

To remedy this, we'll restrict the vocabulary size to n_vocab. Usually this is set to around 2000 words.

Note that the 2000 words we want are the 2000 most common words, not just 2000 random words.

Because of this, we'll need to do a few things.

First we need to keep a count of how many times a word has appeared in a dictionary.

Next, we need to sort that dictionary by value in reverse so that the highest count word comes first.

This will give us the indexes for the top 2000 words.

However, we're still not done, because our word embedding matrix needs to be of size V=2000, and so the word indexes must be from 0..2000.

But the word indexes we currently have are just 2000 random numbers from 0..1 million, assuming our total vocabulary is 1 million words.

Therefore, we need to create a new mapping, from old word index to new word index, where the old word index is any number from 0.. 1 million and the new word index is a number from 0..2000.

This also means we need to create a new word2idx dictionary as well.

If you want to train your model on the entire dataset, you'll probably run out of memory, but there are some ways around this.

First, you don't want to load all your data into memory all at the same time.

One strategy would be to train on each separate text file one at a time within each epoch.

So within the epoch, you loop through each file, open the file, convert all the sentences into arrays of word indexes, and then train on those sentences. Open the next file, and so on. You would have to of course keep a dictionary of the word to index mapping handy so that it remains consistent between each file.

You can imagine that re-processing the sentences after each file open would be slow. Another option would be to convert each sentences into lists of

word indexes before running your neural network, and to save your word to index mapping to a file as well.

That way, your input data can be just a bunch of arrays of word indexes. This would still need to be in multiple files since it would take up too much RAM, but at least you won't have to convert the data each time you open the file.

Yet another option might be to use a simple database like SQLite to store the arrays of word indexes. This way, you can have the entire dataset in the database, and retrieve rows at random without having to store it in memory.

We won't do this since it's not the focus of this course, but if you're interested in learning about techniques like this then you'll want to check out my course on SQL.

Let us now look at the code.

First, for loading the Wiki data:

```
def get_wikipedia_data(n_files, n_vocab, by_paragraph=False):
    prefix = '../large_files/'
    input_files = [f for f in os.listdir(prefix) if f.startswith('enwiki') and
f.endswith('txt')]

    # return variables
    sentences = []
    word2idx = {'START': 0, 'END': 1}
    idx2word = ['START', 'END']
    current_idx = 2
    word_idx_count = {0: float('inf'), 1: float('inf')}

    if n_files is not None:
        input_files = input_files[:n_files]

    for f in input_files:
        print "reading:", f
```

```
        for line in open(prefix + f):
            line = line.strip()
            # don't count headers, structured data, lists, etc...
            if line and line[0] not in ('[', '*', '-', '|', '=', '{', '}'):
                if by_paragraph:
                    sentence_lines = [line]
                else:
                    sentence_lines = line.split('. ')
                for sentence in sentence_lines:
                    tokens = my_tokenizer(sentence)
                    for t in tokens:
                        if t not in word2idx:
                            word2idx[t] = current_idx
                            idx2word.append(t)
                            current_idx += 1
                        idx = word2idx[t]
                        word_idx_count[idx] = word_idx_count.get(idx, 0) + 1
                    sentence_by_idx = [word2idx[t] for t in tokens]
                    sentences.append(sentence_by_idx)

    # restrict vocab size
    sorted_word_idx_count = sorted(word_idx_count.items(),
key=operator.itemgetter(1), reverse=True)
    word2idx_small = {}
    new_idx = 0
    idx_new_idx_map = {}
    for idx, count in sorted_word_idx_count[:n_vocab]:
        word = idx2word[idx]
        print word, count
        word2idx_small[word] = new_idx
        idx_new_idx_map[idx] = new_idx
        new_idx += 1
    # let 'unknown' be the last token
    word2idx_small['UNKNOWN'] = new_idx
    unknown = new_idx

    assert('START' in word2idx_small)
```

```
    assert('END' in word2idx_small)
    assert('king' in word2idx_small)
    assert('queen' in word2idx_small)
    assert('man' in word2idx_small)
    assert('woman' in word2idx_small)

    # map old idx to new idx
    sentences_small = []
    for sentence in sentences:
        if len(sentence) > 1:
            new_sentence = [idx_new_idx_map[idx] if idx in idx_new_idx_map
else unknown for idx in sentence]
            sentences_small.append(new_sentence)

    return sentences_small, word2idx_small
```

Next, for fitting the RNN and doing some word analogies (notice how we can plug and play LSTM and GRU):

```
import sys
import theano
import theano.tensor as T
import numpy as np
import matplotlib.pyplot as plt
import json

from datetime import datetime
from sklearn.utils import shuffle
from gru import GRU
from lstm import LSTM
from util import init_weight, get_wikipedia_data


class RNN:
    def __init__(self, D, hidden_layer_sizes, V):
        self.hidden_layer_sizes = hidden_layer_sizes
        self.D = D
        self.V = V

    def fit(self, X, learning_rate=10e-5, mu=0.99, epochs=10, show_fig=True, activation=T.nnet.relu,
RecurrentUnit=GRU, normalize=True):
        D = self.D
        V = self.V
        N = len(X)
```

```
We = init_weight(V, D)
self.hidden_layers = []
Mi = D
for Mo in self.hidden_layer_sizes:
    ru = RecurrentUnit(Mi, Mo, activation)
    self.hidden_layers.append(ru)
    Mi = Mo

Wo = init_weight(Mi, V)
bo = np.zeros(V)

self.We = theano.shared(We)
self.Wo = theano.shared(Wo)
self.bo = theano.shared(bo)
self.params = [self.Wo, self.bo]
for ru in self.hidden_layers:
    self.params += ru.params

thX = T.ivector('X')
thY = T.ivector('Y')

Z = self.We[thX]
for ru in self.hidden_layers:
    Z = ru.output(Z)
py_x = T.nnet.softmax(Z.dot(self.Wo) + self.bo)

prediction = T.argmax(py_x, axis=1)
# let's return py_x too so we can draw a sample instead
self.predict_op = theano.function(
    inputs=[thX],
    outputs=[py_x, prediction],
    allow_input_downcast=True,
)

cost = -T.mean(T.log(py_x[T.arange(thY.shape[0]), thY]))
grads = T.grad(cost, self.params)
dparams = [theano.shared(p.get_value()*0) for p in self.params]

dWe = theano.shared(self.We.get_value()*0)
gWe = T.grad(cost, self.We)
dWe_update = mu*dWe - learning_rate*gWe
We_update = self.We + dWe_update
if normalize:
    We_update /= We_update.norm(2)

updates = [
    (p, p + mu*dp - learning_rate*g) for p, dp, g in zip(self.params, dparams, grads)
] + [
    (dp, mu*dp - learning_rate*g) for dp, g in zip(dparams, grads)
```

```
        ] + [
            (self.We, We_update), (dWe, dWe_update)
        ]

        self.train_op = theano.function(
            inputs=[thX, thY],
            outputs=[cost, prediction],
            updates=updates
        )

        costs = []
        for i in xrange(epochs):
            t0 = datetime.now()
            X = shuffle(X)
            n_correct = 0
            n_total = 0
            cost = 0
            for j in xrange(N):
                if np.random.random() < 0.01 or len(X[j]) <= 1:
                    input_sequence = [0] + X[j]
                    output_sequence = X[j] + [1]
                else:
                    input_sequence = [0] + X[j][:-1]
                    output_sequence = X[j]
                n_total += len(output_sequence)
                c, p = self.train_op(input_sequence, output_sequence)
                cost += c
                for pj, xj in zip(p, output_sequence):
                    if pj == xj:
                        n_correct += 1
                if j % 200 == 0:
                    sys.stdout.write("j/N: %d/%d correct rate so far: %f\r" % (j, N, float(n_correct)/n_total))
                    sys.stdout.flush()
            print "i:", i, "cost:", cost, "correct rate:", (float(n_correct)/n_total), "time for epoch:",
(datetime.now() - t0)
            costs.append(cost)

        if show_fig:
            plt.plot(costs)
            plt.show()


def train_wikipedia(we_file='word_embeddings.npy', w2i_file='wikipedia_word2idx.json',
RecurrentUnit=GRU):
    sentences, word2idx = get_wikipedia_data(n_files=50, n_vocab=2000)
    print "finished retrieving data"
    print "vocab size:", len(word2idx), "number of sentences:", len(sentences)
    rnn = RNN(30, [30], len(word2idx))
    rnn.fit(sentences, learning_rate=10e-6, epochs=10, show_fig=True, activation=T.nnet.relu)
```

```
    np.save(we_file, rnn.We.get_value())
    with open(w2i_file, 'w') as f:
        json.dump(word2idx, f)

def find_analogies(w1, w2, w3, we_file='word_embeddings.npy',
w2i_file='wikipedia_word2idx.json'):
    We = np.load(we_file)
    with open(w2i_file) as f:
        word2idx = json.load(f)

    king = We[word2idx[w1]]
    man = We[word2idx[w2]]
    woman = We[word2idx[w3]]
    v0 = king - man + woman

    def dist1(a, b):
        return np.linalg.norm(a - b)
    def dist2(a, b):
        return 1 - a.dot(b) / (np.linalg.norm(a) * np.linalg.norm(b))

    for dist, name in [(dist1, 'Euclidean'), (dist2, 'cosine')]:
        min_dist = float('inf')
        best_word = ''
        for word, idx in word2idx.iteritems():
            if word not in (w1, w2, w3):
                v1 = We[idx]
                d = dist(v0, v1)
                if d < min_dist:
                    min_dist = d
                    best_word = word
        print "closest match by", name, "distance:", best_word
        print w1, "-", w2, "=", best_word, "-", w3

if __name__ == '__main__':
    train_wikipedia()
    we = 'lstm_word_embeddings2.npy'
    w2i = 'lstm_wikipedia_word2idx2.json'
    train_wikipedia(we, w2i, RecurrentUnit=LSTM)
    find_analogies('king', 'man', 'woman', we, w2i)
    find_analogies('france', 'paris', 'london', we, w2i)
    find_analogies('france', 'paris', 'rome', we, w2i)
    find_analogies('paris', 'france', 'italy', we, w2i)
```

Conclusion

I really hope you had as much fun reading this book as I did making it.

Did you find anything confusing? Do you have any questions?

I am always available to help. Just email me at: info@lazyprogrammer.me

I do 1:1 coaching and consulting as well.

Do you want to learn more about deep learning? Perhaps online courses are more your style. I happen to have a few of them on Udemy.

My first course in deep learning is a lot like the book, but you get to see me derive the formulas and write the code live:

Data Science: Deep Learning in Python

https://udemy.com/data-science-deep-learning-in-python

Are you comfortable with this material, and you want to take your deep learning skillset to the next level? Then my follow-up Udemy course on deep learning is for you. Similar to this book, I take you through the basics of Theano and TensorFlow - creating functions, variables, and expressions, and build up neural networks from scratch. I teach you about ways to accelerate the learning process, including batch gradient descent, momentum, and adaptive learning rates. I also show you live how to create a GPU instance on Amazon AWS EC2, and prove to you that training a neural network with GPU optimization can be orders of magnitude faster than on your CPU.

Data Science: Practical Deep Learning in Theano and TensorFlow

https://www.udemy.com/data-science-deep-learning-in-theano-tensorflow

When you've got the basics of deep learning down, you're ready to explore alternative architectures. One very popular alternative is the convolutional

neural network, created specifically for image classification. These have promising applications in medical imaging, self-driving vehicles, and more. In this course, I show you how to build convolutional nets in Theano and TensorFlow.

Deep Learning: Convolutional Neural Networks in Python

https://www.udemy.com/deep-learning-convolutional-neural-networks-theano-tensorflow

In part 4 of my deep learning series, I take you through unsupervised deep learning methods (that's this book!). We study principal components analysis (PCA), t-SNE (jointly developed by the godfather of deep learning, Geoffrey Hinton), deep autoencoders, and restricted Boltzmann machines (RBMs). I demonstrate how unsupervised pretraining on a deep network with autoencoders and RBMs can improve supervised learning performance.

Unsupervised Deep Learning in Python

https://www.udemy.com/unsupervised-deep-learning-in-python

Would you like an introduction to the basic building block of neural networks - logistic regression? In this course I teach the theory of logistic regression (our computational model of the neuron), and give you an in-depth look at binary classification, manually creating features, and gradient descent. You might want to check this course out if you found the material in this book too challenging.

Data Science: Logistic Regression in Python

https://udemy.com/data-science-logistic-regression-in-python

To get an even simpler picture of machine learning in general, where we don't even need gradient descent and can just solve for the optimal model parameters directly in "closed-form", you'll want to check out my first Udemy course on the classical statistical method - linear regression:

Data Science: Linear Regression in Python

https://www.udemy.com/data-science-linear-regression-in-python

If you are interested in learning about how machine learning can be applied to language, text, and speech, you'll want to check out my course on Natural Language Processing, or NLP:

Data Science: Natural Language Processing in Python

https://www.udemy.com/data-science-natural-language-processing-in-python

If you are interested in learning SQL - structured query language - a language that can be applied to databases as small as the ones sitting on your iPhone, to databases as large as the ones that span multiple continents - and not only learn the mechanics of the language but know how to apply it to real-world data analytics and marketing problems? Check out my course here:

SQL for Marketers: Dominate data analytics, data science, and big data

https://www.udemy.com/sql-for-marketers-data-analytics-data-science-big-data

Are you interested in stock prediction, time series, and sequences in general? My Hidden Markov Models course is where you want to be. I teach you not only all the classical theory of HMMs, but I also show you how to write them in Theano using gradient descent! This is great practice for writing deep learning models and it will prepare you well for its sequel, Deep Learning Part 5: Recurrent Neural Networks in Python. You can get the HMM course here:

Unsupervised Machine Learning: Hidden Markov Models in Python

https://udemy.com/unsupervised-machine-learning-hidden-markov-models-in-python

Recurrent Neural Networks, the topic covered in this book, is covered even more in-depth in the corresponding video course:

Deep Learning: Recurrent Neural Networks in Python

https://udemy.com/deep-learning-recurrent-neural-networks-in-python

Finally, I am *always* giving out **coupons** and letting you know when you can get my stuff for **free**. But you can only do this if you are a current student of mine! Here are some ways I notify my students about coupons and free giveaways:

My newsletter, which you can sign up for at http://lazyprogrammer.me (it comes with a free 6-week intro to machine learning course)

My Twitter, https://twitter.com/lazy_scientist

My Facebook page, https://facebook.com/lazyprogrammer.me (don't forget to hit "like"!)