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

October 8, 2015

This the third part of the Recurrent Neural Network Tutorial.

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

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

BACKPROPAGATION THROUGH TIME (BPTT)

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

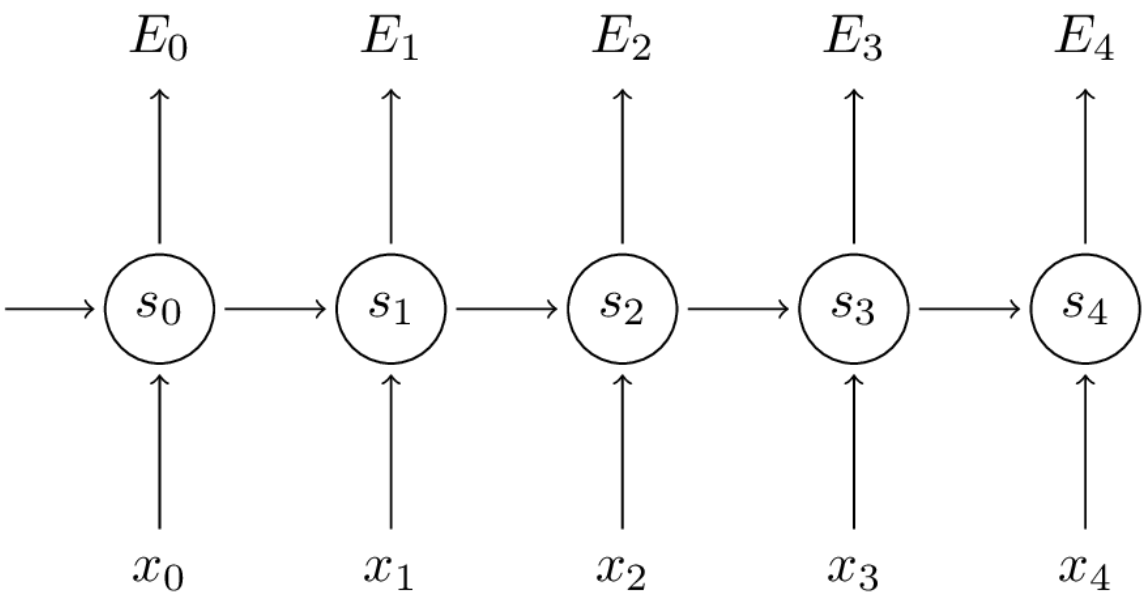
literature out there that I am referencing.

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

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

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

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



Remember that our goal is to calculate the gradients of the error with respect to our parameters U, V and W and then learn good parameters using Stochastic Gradient Descent. Just like we sum up the errors, we also sum up the gradients at each time step for one training example: $\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W}$.

To calculate these gradients we use the chain rule of differentiation. That's the backpropagation algorithm when applied backwards starting from the

error. For the rest of this post we'll use E_3 as an example, just to have concrete numbers to work with.

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

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

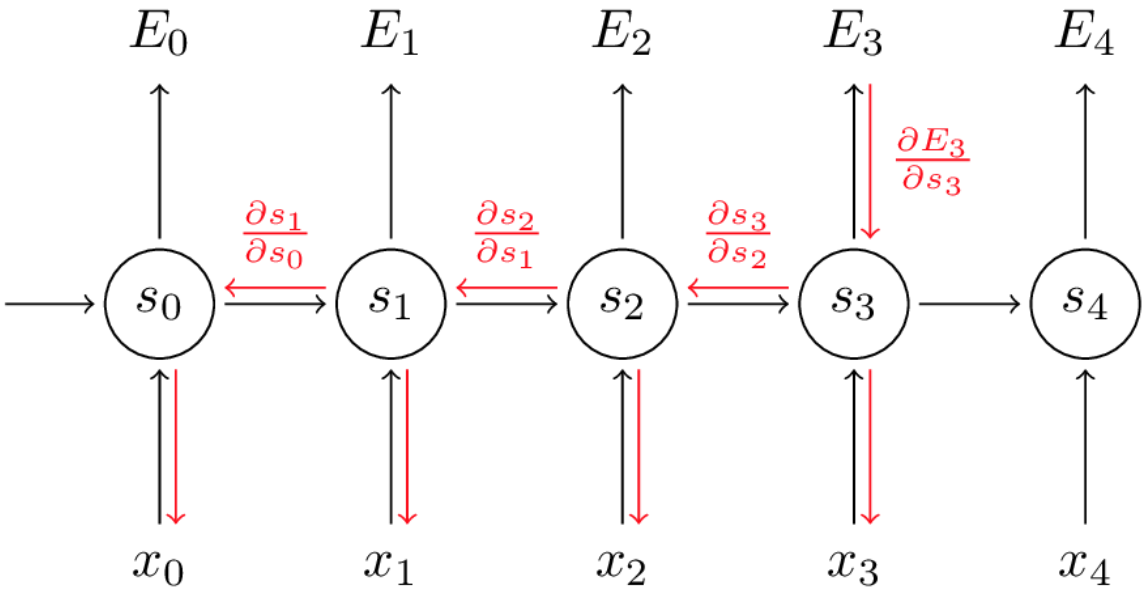
But the story is different for $\frac{\partial E_3}{\partial W}$ (and for U). To see why, we write out the chain rule, just as above:

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

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

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

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



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

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

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

```
            # print "Backpropagation step t=%d bptt step=%d " % (t, bptt_step)
            # Add to gradients at each previous step
            dLdW += np.outer(delta_t, s[bptt_step-1])
            dLdU[:,x[bptt_step]] += delta_t
            # Update delta for next step dL/dz at t-1
            delta_t = self.W.T.dot(delta_t) * (1 - s[bptt_step-1] ** 2)
    return [dLdU, dLdV, dLdW]
```

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

THE VANISHING GRADIENT PROBLEM

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

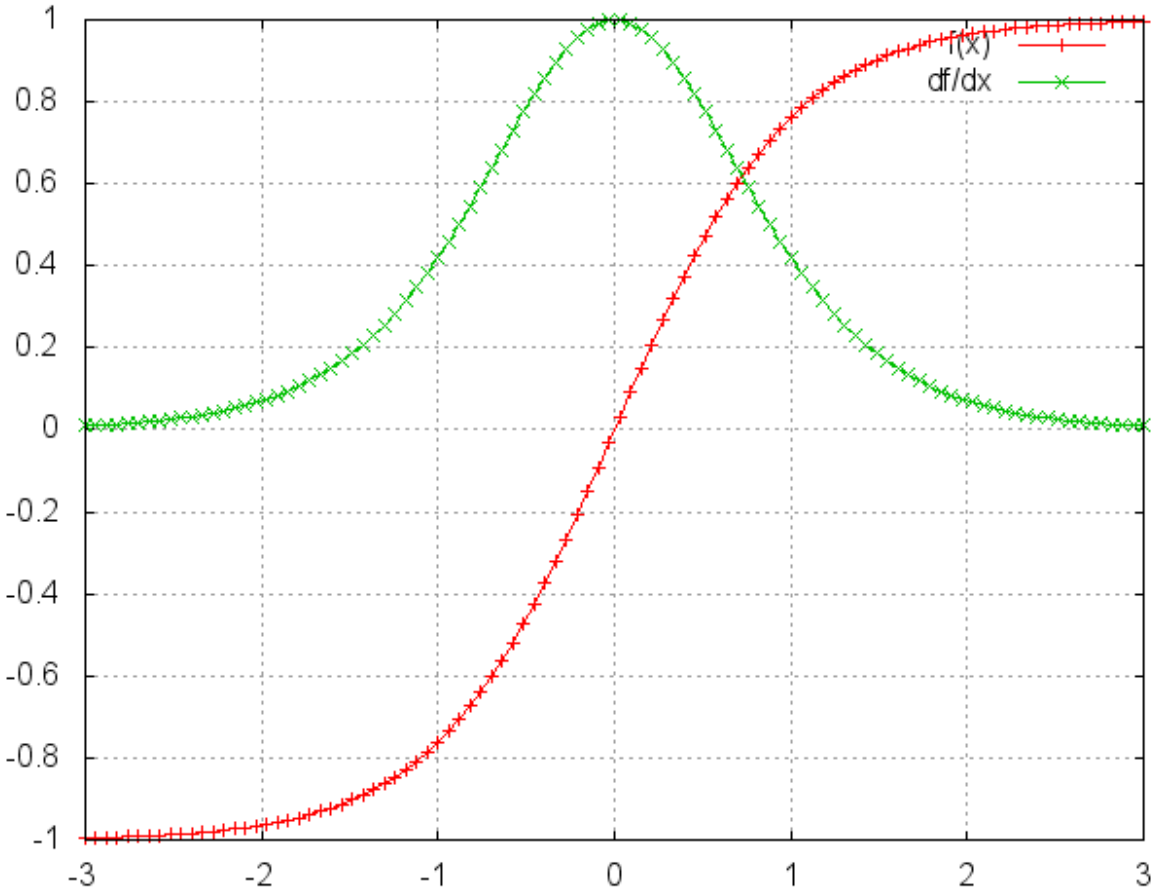
$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial s_3}\frac{\partial s_3}{\partial s_k}\frac{\partial s_k}{\partial W}$$

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

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

It turns out (I won't prove it here but this paper goes into detail) that the 2-norm, which you can think of it as an absolute value, of the above Jacobian matrix has an upper bound of 1. This makes intuitive sense because our $tanh$

(or sigmoid) activation function maps all values into a range between -1 and 1, and the derivative is bounded by 1 (1/4 in the case of sigmoid) as well:



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

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

It is easy to imagine that, depending on our activation functions and network parameters, we could get exploding instead of vanishing gradients if the values of the Jacobian matrix are large. Indeed, that's called the *exploding*

gradient problem. The reason that vanishing gradients have received more attention than exploding gradients is two-fold. For one, exploding gradients are obvious. Your gradients will become NaN (not a number) and your program will crash. Secondly, clipping the gradients at a pre-defined threshold (as discussed in this paper) is a very simple and effective solution to exploding gradients. Vanishing gradients are more problematic because it's not obvious when they occur or how to deal with them.

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

Please leave questions or feedback in the comments!

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

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

Recurrent Neural Network Tutorial, Part 4 – Implementing a GRU/LSTM RNN with Python and Theano →

114 Comments WildML Login

Recommend 10 Share Sort by Best

Join the discussion…

OD • a year ago

Hi Denny,
I am not sure the chain rule for delta(E_3)/ W is actually correct since just above you stated that delta(E_3)/W = delta(E_3) / delta(y_hat) delta(y_hat) / delta(s_3) delta(s_3) /W, if k = 3 you have the initial expression in the summation again.
Could you please check ?

3 • Reply • Share ›

martian ➚ OD • a month ago

i personally second your doubt... he's missing a term here.
the post also missed the same term in the python code.

• Reply • Share ›

Boussad ➚ martian • a month ago

Me too. I think there are some mistakes out there...

1 • Reply • Share ›

martian ➚ Boussad • a month ago

see more

• Reply • Share ›

martian ➚ Boussad • a month ago

it's when I recursively derive the S3 over state weights

everytime, WSt is depending on WSt-1, so in doing the t=3, you have to know all the t=2, t=1 derivatives, as shown in my deduction process in that image. so i'm wondering if i'm doing this wrong or something...
https://github.com/martianm...

Reply • Share ›

Boussad martian • a month ago

see more

1 Reply • Share ›

martian Boussad • a month ago

trying to figure out what this part is saying here..... sorry i have a little difficulty reading your hand written words, but other than this little part it looks great! studying it still.

Reply • Share ›

martian martian • a month ago

just this letter here...

Reply • Share ›

martian ↗ Boussad • a month ago

Lol, need more samples to learn your hand writing letters.... hold on. let me study over it.

Reply • Share ›

Boussad ↗ martian • a month ago

It is just the greek Psi letter. This is used to denote dSdU just like Lambda was used to denote dSdW. I was going to give you only calculations related to dW but I added this at the last minute to calculate also dU. For dV, nothing else to add was necessary!

Reply • Share ›

martian ↗ Boussad • a month ago

oh of course, i should have guessed. anyways, i looked through your deduction process and basically we agree that it's a recursive process starting from the beginning of time, not 'backwards'. i will work on the python code after a few days and will be nice to share it with you again.

Reply • Share ›

Boussad ↗ martian • a month ago

Hi Martian,
Did you take some time to implement the RNN using my equations ? Actually i did but only by neglecting term d(S-1)/dw as often done. I got the same results as the supposed BPTT. As a matter of fact, I neglected the term because I don't know how to put everything in a compact form using matrices. Which product to use, outer product ? dot product ? or vectorial product ? I think tensors are rather needed !
By the way, you can join me in linkedin at https://www.linkedin.com/in... if you want.

Reply • Share ›

go2carter ↗ Boussad • 24 days ago

That's great to hear, Boussad! I will have to check it out once I get more time from school again. Tensors are very useful, as is Einstein summation notation, which I found tremendously helpful when figuring out outer vs inner products etc.

Check out https://en.wikipedia.org/wi... and http://luc.edu/faculty/dsla... if not familiar.

Reply • Share ›

xianghang • a year ago

Hi Denny, thanks for such a great post.
Just a minor suggestion, it would better to keep the definition of z consistent: z_3 is

defined as vs_3 but in definition of \delta_2^(3), z_2 is defined as Ux_2 + Ws_1. I think this is what z represents in the code comment. Thus, it would be nice to have a clear definition of z in the code comments.

3 ^ v • Reply • Share ›

Denny Britz Mod ➚ xianghang • a year ago

You are probably right, I may have somehow mixed up the definition in the formula here and the code (which I took from the previous post). It has been a while since I wrote this. I don't have time to go back and double check right now, but will do that.

^ v • Reply • Share ›

Nipun Agarwal • 8 months ago

Awesome I really loved this series. Would like to thank you for that.

2 ^ v • Reply • Share ›

go2carter • 3 months ago

hi denny,

thanks a bunch for this site, these tutorials have helped a bunch.

I found that for BPTT going through the math helped a bunch, so if anybody else wants to see the math in depth, I threw this together: https://github.com/go2carte...

1 ^ v • Reply • Share ›

Chandresh Maurya ➚ go2carter • a day ago

Hi, do you have implementation for your BPTT derivation as well? Thanks for your gradient derivation. It was awesome and crystal clear.

^ v • Reply • Share ›

go2carter ➚ Chandresh Maurya • a day ago

Thanks, Chandresh. Unfortunately, don't have an implementation yet--been a much busier semester than anticipated... Will upload to the same repo once done though

^ v • Reply • Share ›

haha ➚ go2carter • 20 days ago

Really Great! I like the Einstein summation very much. I did not know this before. THANKS

^ v • Reply • Share ›

go2carter ➚ haha • a day ago

glad to hear it, haha. Einstein summation changes the game

^ v • Reply • Share ›

Weikeng Qin ➚ go2carter • 25 days ago

Hi go2carter,

I lost track starting from (17) -- what does the "arrow" mean?

Reply • Share ›

go2carter ➚ Weikeng Qin • 24 days ago

Hi Weikeng,

The arrow is signifying "going to" or "updating" the variable. The left side is the derivative we're calculating, but since multiple terms in s_t depend on W_{ij}, we must apply chain rule. It got confusing writing an equal sign since d s_{t_m}/d W_{ij} would appear on both sides. The arrow helped get around that. So, the left side is what we want to calculate, the right side is what that actually looks like after applying chain rule, so each partial derivative we can actually treat as a partial derivative (i.e. no more worrying about chain rule and just assume that all other variables are constants). Since this is recursive, we must continue to apply this technique, which leads us to (18), then the more general (20)

Appreciate the feedback--thinking about how I can make this clearer going forward. Let me know if this helps.

Reply • Share ›

JianYi Yang ➚ go2carter • a month ago

Hi go2carter, your post helps a lot!
One place I don't understand is that when you do dV_lm/dV_ij, what does delta_il and delta_jm mean in (12b)?

Reply • Share ›

JianYi Yang ➚ JianYi Yang • a month ago

does it mean if i=l then the value is 1, else the value is 0?

Reply • Share ›

go2carter ➚ JianYi Yang • a month ago

exactly, that's a kronecker delta: https://en.wikipedia.org/wi...

Reply • Share ›

JianYi Yang ➚ go2carter • a month ago

got it, thanks!

Reply • Share ›

martian ➚ go2carter • a month ago

uh, the process described in the link is kind of crazy.. too much.. but overall i think it's pointing out that the equations in this post is missing something.....

Reply • Share ›

go2carter ➚ martian • a month ago

Had to go through the derivations to find exactly what I wasn't following. After calculating the chain rule, it turns out get another outer product when computing loss for W and U which aren't

explicitly mentioned until the code. It also made the sum over the s's easier to understand.

What are you thinking the post is missing?

Reply • Share ›

martian → go2carter • a month ago

it didn't seem to be backward at all!

Reply • Share ›

martian → go2carter • a month ago

it's when I recursively derive the S3 over state_weights... everytime, WSt is depending on WSt-1, so in doing the t=3, you have to know all the t=2, t=1 derivatives, as shown in my deduction process in that image. so i'm wondering if i'm doing this wrong or something...

Reply • Share ›

go2carter → martian • a month ago

sorry for late response getting back..

that's right, the backprop through time demonstrates itself as this recursive definition

This is what Denny does in the last line of code right before he returns the gradient in the bptt function

Reply • Share ›

martian → go2carter • a month ago

can you post your handwritten deduction somewhere and let me take a look?
Here's mine...
https://github.com/martianm...

Reply • Share ›

go2carter → martian • a month ago

thx for sharing, will look over

unfortunately my handwritten notes are on the other side of the country, but what I typed up follows them very closely

Reply • Share ›

martian → go2carter • a month ago

also, don't you think the equations in this posts are missing something?

Reply • Share ›

martian → go2carter • a month ago

this is super nice, you should do a blog for just this

Reply • Share ›

go2carter ➚ martian • a month ago

Thx, martian. In the future, maybe I'll tone down the math a bit, then :)

Reply • Share ›

jzhang • a year ago

Hi Denny,

thanks for your nice post.
I am wondering in your demo code of bptt, there might exist duplicated computation. i.e., inner loop may partially do same computation when t moves to t-1.

I am actually confused by the inner loop.
how about calculate delta of s_t-1 in step t, and in step t-1 simply add it with delta of s_t-1 from E_t-1, then using this new delta of s_t-1 to compute grads of W and U in t-1.

1 Reply • Share ›

mgl888 ➚ jzhang • 10 months ago

I second this. You can write a BPTT function that only uses one loop. You just need to add the ds[t-1] from the previous time step.
Code here (passes gradient check): https://gist.github.com/m-l...

P.S.: I think truncation can also be applied here by wrapping this function call in another loop.

1 Reply • Share ›

Ashutosh Modi • a year ago

Hi Denny,

Very nice post. I have a question regarding, weight update in RNN. Basically, we collect all weight gradients in a sequence using BPTT (by summing individual grads) and then do one big weight update at the end of the sequence. My question is that why don't we take the average of the collected grads (instead of just the sum) and then update. Since in the train data different seq would be of diff lengths, so the update might be affected by seq length.

By the way in your bptt code, you have:
dLdV += np.outer(delta_o[t], s[t].T)
but it should be :
dLdV += np.outer(delta_o[t], s[t])

s[t] should not be transposed, right?

1 Reply • Share ›

Denny Britz Mod ➚ Ashutosh Modi • a year ago

Hi,

We are making multiple predictions per sequence, one per time step, that's

basically a minibatch. So yes,we should divide the loss (and the gradients) by sequence length. In practice we often pad sequences to be of the same length so we can do efficient batching of examples. In that case all sequences will have the same length and the normalization isn't required. But you're right, we should divide by sequence length.

Re the transpose. I don't have time to go back and look at the code now, but the gradient check for the code passed, so I'm assuming it's right. Maybe it was transposed previously. I could be mistaken though.

Reply • Share ›

Igor Chernobaev ➚ Denny Britz • a month ago

Hi, Denny,
you gradient check returns error for these parameters:
vocabulary_size=8
np.random.seed(10)
model = RNNNumpy(vocabulary_size, 2, bptt_truncate=4)
model.gradient_check([0, 1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6, 7])
And the error is:
Performing gradient check for parameter U with size 16.
Gradient Check ERROR: parameter=U ix=(0, 0)
+h Loss: 14.451545254711402
-h Loss: 14.452114350897709
Estimated_gradient: -0.28454809315370255
Backpropagation gradient: -0.2721150178481872
Relative Error: 0.022335008481410117

But with bptt_truncate > 5 everything is ok. I'm completely confused.

Reply • Share ›

martian ➚ Denny Britz • a month ago

lol, this is not the attitude you should carry on with.. it's misleading people if there's any such kind of error

Reply • Share ›

Scott • a year ago

Denny, how would you go about modifying bptt to account for a dynamic network? That is, if x1 = f(y0,x0). My bptt works great if x is a static vector, but as soon as x changes as a function of the previous states the gradient check fails.

1 Reply • Share ›

Denny Britz Mod ➚ Scott • a year ago

Hi Scott,

If I understand correctly, this would be equivalent to having a RNN with only one input (x_0), but an output y_t at each time step. If x_1 is a function of the previous time step I wouldn't call it an "input" the network, if that makes sense. This mean the gradients will change and you would need to recalculate them. In practice, you really want to use a library that supports auto-differentiation to compute the gradients, something like Theano, Torch,

Tensorflow, etc.

However, what is your use case for this? In theory, dependencies on x_{t-1} and y_{y-1} are implicitly captured by the hidden state, so I'm not sure how much you would gain by making the dependencies explicit. What I'm saying is that x1 = f(s_0) should be able to represent almost the same as x1 = f(y0, x0). Does that make sense?

There's also a decision around what the y's are in this case. If x1 = f(y0,x0), is y0 the output of the network of at step 0, or is the correct training label (regardless of the network output) at step 0 from your training data? The latter would be called "teacher forcing".

Reply • Share ›

Scott → Denny Britz • a year ago

Hi Denny,

Thanks for the response.

I am attempting to train a RNN controller where X(t) would represent the current state of the system. Y(t) would be the input to the "plant" or the dynamics of the system(which is differentiable). After calculating Y0, X1 can be calculated based on the current state X0 and the control input. The training data contains a sequence of states that were a result of using a specified control sequence. Where Error is defined as (x-x(training))^2. Basicaly trying to find weights that can perform the same as the trainer controller.

Any thoughts you have on this would be great.

Thanks.

Reply • Share ›

Sowmya Jonnada • a year ago

Hi Denny, I would like to understand why the derivative of the activation function of output layer(softmax) is missing in the delta_o equation.

delta_o[np.arange(len(y)), y] -= 1.

1 Reply • Share ›

Denny Britz Mod → Sowmya Jonnada • a year ago

If you have z = Vs_t, yhat = softmax(z) and L(yhat) = cross entropy loss, then the derivative dL/dz turns out to be (yhat - y), which is what the above is. By the chain rule, dL/dV is then (delta * s_t), which is the next step in the code. If you need a more detailed derivation you can take a look at this: http://cs224d.stanford.edu/...

Does that make sense? Let me know if not.

2 Reply • Share ›

OB ➚ Denny Britz • a year ago

Hi. I still don't understand how dL/dz is (yhat - y). I checked the link, too. In the handout, it is used maximum margin objective function. Additionally, if we would have been used quadratic cost function, I agree that dL/dz is (yhat - y). But here, I couldn't derive it.

 • Reply • Share ›

- CONNECT -

- RECENT POSTS -

- LINKS -

- ARCHIVES -

- CATEGORIES -

RNNs

Tensorflow

- SUBSCRIBE TO BLOG VIA EMAIL -

Enter your email address to subscribe to this blog and receive notifications of new posts by email.

Email Address

SUBSCRIBE

- META -

Log in

Entries RSS

Comments RSS

WordPress.org