Author: Benjamin Smidt
Created: October 18th, 2022
Last Updated: October 18th, 2022

Assignment 4: RNNs, LSTM, and Attention with Image Captioning

Note to reader.

This is my work for assignment four (A4) of Michigan's course EECS 498: Deep Learning for Computer Vision. The majority of explanations and understanding are derived from Justin Johnson's Lectures and Stanford's CS 231N Lecture Notes. This document is meant to be used as a reference, explanation, and resource for the assignment, not necessarily a comprehensive overview of Neural Networks. If there's a typo or a correction needs to be made, feel free to email me at benjamin.smidt@utexas.edu so I can fix it. Thank you! I hope you find this document helpful.

Contents

1 Vanilla Recurrent Neural Networks

1.1 RNN Forward

Recurrent neural networks are very powerful in their ability to process and output variable length data. Said another way, RNNs can be fed different length inputs as well predict different length outputs making them a powerful and useful paradigm in many applications. We'll go into more depth as we go along but for now we'll start with vanilla recurrent neural networks.

$$h_t = f_w(h_{t-1}, x_t)$$

To achieve variable length inputs and outputs we need to change our neural network model a bit. Instead of having some predefined network size, we have a function that takes two inputs: the output of the previous computation (also known as the "hidden state") and some data input (usually interpreted as a sequence). See this picture for a visual. For vanilla neural networks (also known as "Elman RNNs") we use the following function.

$$h_t = tanh(W_{hh}h_{t-1} + W_{xh}x_t + b) \tag{1}$$

where h_t is the current state, x_t is the input data, W_{hh} is our (reused) weight matrix for the hidden state input, and W_{xh} is our (reused) weight matrix for the current input x_t. We also add a bias b. To be clear, W_{hh} and W_{xh} do not change at all beween time steps. They are the same set of parameters throughout the neural network's computation. For out first function, *rnn-forward*, we simply write down Eq. (1) in code and store our needed variables in *cache* for backpropagation.

If you're wondering about initialization and how to know when to stop computing h_t, keep reading. I'll answer those and other questions as we go along.

1.2 RNN Backward

Let's look at backpropagating a given time step given our function. Recall that

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Thus, by quotient rule, our derivative is as follows

$$\frac{\partial\ tanh(z)}{\partial z} = \frac{(e^z + e^{-z})(e^z + e^{-z}) - (e^z - e^{-z})(e^z - e^{-z})}{(e^z + e^{-z})^2}$$

$$\frac{\partial\ tanh(z)}{\partial z} = \frac{(e^z + e^{-z})^2 - (e^z - e^{-z})^2}{(e^z + e^{-z})^2}$$

$$\frac{\partial\ tanh(z)}{\partial z} = \frac{(e^z + e^{-z})^2}{(e^z + e^{-z})^2} - \frac{(e^z - e^{-z})^2}{(e^z + e^{-z})^2}$$

$$\frac{\partial\ tanh(z)}{\partial z} = 1 - tanh^2(z)$$

If we set $z = W_{hh}h_{t-1} + W_{xh}x_t + b$, we can get the first term in our backpropagation. I'll write our original function here as well for clarity.

$$h_t = tanh(z) \tag{2}$$

$$\frac{\partial loss}{\partial z} = \frac{\partial loss\ z}{\partial h_t} \odot (1 - tanh^2(z))$$

Where $\frac{\partial loss\ z}{\partial h_t}$ is passed down to us from some function upstream and $\odot$ indicates elementwise multiplication (work out the shapes!). Moving forward (or backward I guess) in our backpropagation, we'll next work on each of the variables inside the *tanh* function starting with W_{hh} and W_{xh}.

$$\frac{\partial\ z}{\partial W_{hh}} = h_{t-1} \quad \text{and} \quad \frac{\partial\ z}{\partial W_{xh}} = x_t$$

Thus

$$\frac{\partial\ h_t}{\partial z}\frac{\partial\ z}{\partial W_{hh}} = h_{t-1}^T\ [1 - tanh^2(z)]$$

$$\frac{\partial\ h_t}{\partial z}\frac{\partial\ z}{\partial W_{xh}} = x_t^T[1 - tanh^2(z)]$$

where $z = W_{hh}h_{t-1} + W_{xh}x_t + b$ and the transposes are derived by the shape convention. Next we have h_{t-1} and x_t.

$$\frac{\partial\ z}{\partial h_{t-1}} = W_{hh} \quad \text{and} \quad \frac{\partial\ z}{\partial x_t} = W_{xh}$$

Thus

$$\frac{\partial\ h_t}{\partial z}\frac{\partial\ z}{\partial h_{t-1}} = [1 - tanh^2(z)]\ W_{hh}^T$$

$$\frac{\partial\, h_t}{\partial z}\frac{\partial\, z}{\partial x_t} = [1 - tanh^2(z)]\, W_{xh}^T$$

And finally for our bias

$$\frac{\partial\, z}{\partial b} = 1$$

$$\frac{\partial\, h_t}{\partial z}\frac{\partial\, z}{\partial b} = [1 - tanh^2(z)]$$

Of course, we'll have to manipulate the bias b more when we program the backpropagation since b is actually broadcast over the the outputs which needs to be accounted for in our backpropagation (we sum over the rows, which is dimension 0).

1.3 RNN Forward

This is where some of our initial questions need some answers.

How do we initialize the network? (i.e. where does the first h_{t-1} come from?). We simply initialize it as a separate matrix and make it a learnable parameter of the network.

When do we stop our recursive calls to f_w? When the input sequence has run out. For instance, given T time steps for a set of data, we recursively compute f_w until we get to the last time step where we throw in the towel and compute our loss function.

Speaking of loss functions, how is it computed? Well, because recurrent neural networks are very malleable, how you compute the loss depends on the type of recurrent neural network you use. I really like the visuals shown on this website. Our network is a one to many relationship, meaning we have a loss computed for each hidden state h_i. Thus, we must compute the gradient with respect to the loss produced by the current state h_i as well as with respect to all the downstream states $h_d > h_i$. This sounds more difficult than it really is in practice.

This forward function isn't really anything new to use so I'll leave you to just read the code in the notebook. The only detail to note is that I chose to transpose the input x (shape N x T x D) for computing the forward pass to

make the computation more clear. This led me to have to take the transpose of the output h (tensor of hidden states with shape N x T x H).

1.4 RNN Backward

As I briefly mentioned above, computing the gradient for any given time step is a little more complicated than what we're used to. It's really not much different though if you're used to computing gradients already. First, see this website for a picture of what a *one-to-many* relationship looks like for recurrent neural networks.

In a one-to-many RNN, a loss function is computed for each time step. Thus, the loss computed for a given time step depends on all the time steps before it. This means, for a given time step, our gradient depends on the loss computed at that time step as well as the loss computed for all the time steps after it.

In practice this isn't too difficult. The time step h_{t+1} will pass back some gradient to time step h_t. Since this gradient is passed back from every time step ahead of h_t, it embeds all the gradient with respect to all the loss functions after h_t (so excluding the loss function calculated on time step h_t. Thus, we simply add the gradient passed back from h_{t+1} and the gradient with respect to the loss function computed at h_t and pass this value to our *rnn-step-backward* function. And that's it!

Again, the details here are pretty simply once you understand the high level concept so I won't explain the code. We're just backpropagating like normal with a couple extra steps added in between.