Author: Benjamin Smidt
Created: October 18th, 2022
Last Updated: October 23rd, 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.

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

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.

2 RNN for Image Captioning

Initially we implement some functions that we need to do our image captioning. Namely, we look at MobileNet v2 architecture, create vectors for our word embeddings, and look at a temporal affine layer. I'm not going go over these because they're pretty straightforward and we can just use PyTorch's API's for a lot of it to make things simple.

2.1 RNN Captioning Forward

2.1.1 Overview

This architecture was a little confusing to me at first I'm not going to lie. However, it makes perfect sense once I went through it step by step. Let's do that.

Remember that we're given an image and associated captions for a given number of examples. Our job is to use this data to train a network such that, if we only fed it an input picture, it would give us a caption. To do this we're using a many-to-many RNN architecture. Our initial state is a feature vector extracted from our image, and each time step is fed the vector representation of the previous word. Then, at each time step, it predicts the next word which we compute softmax loss on and use for backpropagation.

2.1.2 Input Image Feature Vector

There are quite a few operational details here but that's the big picture. Let's start with extracting our feature vector from our input image. We'll use the *FeatureExtractor* class (implemented for us) to do this. What this class does is use the MobileNet v2 model to extract the features for us (as opposed to training a whole neural network for this purpose only).

We'll adjust the network a little bit by chopping off the last two layers (FC-1000 and softmax) leaving us with a feature tensor of 1280 x 4 x 4 as the output. We'll then use average pooling on dimensions 1 and 2 to get our feature vector of length 1280. Before we talk about how we input this into our RNN, let's quickly review our RNN architecture.

Each step of our RNN takes two inputs: the previous hidden state h_t and the current input x_t. We use $h_{t+1} = tanh(W_{hh}h_t + W_{xh}x + b)$ where W_{hh} and W_{xh} are the same matrix for all time steps. The output, h_{t+1}, is then fed into the next step along with the next input $x_t + 1$. In our case, each x is a vector representation of a word.

Staying on topic though let's look at some shapes. x will be an N x W dimensional tensor where N is the number of examples in our current batch and W is the length of our vector representations of a given word (so we have N words represented each as vectors of length W). A given h_t will be of shape N x H where H is the vector length we choose to represent a given state for a given example.

Okay, so back to our input feature vector of length 1280. Remember, because we always do training in batches, we'll have N input feature vectors of length 1280. Thus, if we tried to use this as our current input as h_0 it wouldn't work because we're trying to input a tensor of shape N x 1280 which isn't the same shape as our general hidden state h_t of N x H (I mean, guess it could be but it isn't for our implementation).

Thus, we use an affine (linear) transformation (matrix multiplication) to convert it to the shape we want. In particulary, we multiply our feature tensor of shape N x 1280 by our affine transformation tensor (matrix) of shape 1280 x H to get our initial state h_0 of shape N x H. This affine transformation matrix is a learnable part of our network meaning we will perform backprop on it (although we let PyTorch do that for us this assignment).

2.1.3 RNN, Loss, and Backprop

I'm not going to go super in depth on the details since I already explained them above when implementing the loss and backpropagation for RNNs, but there are some things I need to clear up. First, I've stated before, each time step is associated with a predicted next word given the previous hidden state and the previous word (represented by a vector). How does this work exactly though?

Well, we again use an affine transformation to turn a given example's hidden vector of length H into a hidden vector of length N. That is, we matrix multiply the vector by a matrix of shape H x V where V is the length of our vocabulary (more on this in a second). Thus, a given time step h_t with N hidden vectors can be matrix multiplied to produce a *score* tensor of shape N x V (N x H @ H x V).

Next, we can compute our softmax loss function on each example (so the V dimension, dimension 1 in this case) using the ground truth labels of what the next word should actually be. This is done for every time step in our RNN. Thus, our backpropagation needs to include the loss function at the current time step and all the time steps after it for a given time step h_t (as discussed in the previous section, the idea is the exact same).

Also, please note that the affine function used to transform h_t into our scores tensor is another learnable parameter of the network (meaning we again do backpropagation on it). It is the same function used at every time step though, similar to W_{hh} and W_{xh}.

2.1.4 Word Embeddings

That last part, that I've kind of beat around the bush, is our word embeddings. This part isn't very difficult to understand but I do want to make sure it's clear. Essentially, the best method to represent words (so far) is using vectors. In our case, a vector of length W represents a single word. I won't get into why this is the case or how we come up with those vectors.

For this assignment, the vector representations don't actually mean anything. They're just unique identifiers for words. Typically though, word vectors *do* have meaning associated with them. The numbers in the vectors actually indicate something about semantic meaning or relationship to other words. We use vectors to represent words because this is standard practice in NLP and we need to do so to make our neural network work (x being a vector instead of just a scalar gives the network more parameters with which to store information and make better predictions).

3 LSTM

Onto LSTMs, which stands for Long Short Term Memory. I don't think this the greatest name (seems kind of misleading I guess) but I see where the name came from. In essence, LSTMs solve the problem of vanishing and exploding gradients that are pervasive in long RNNs. LSTM architecture is *very* prevalent, so get used to seeing it and understand it well. What is it? It's a little complicated at first but we'll take step by step.

3.1 LSTM Forward

First, in addition to having a hidden state at a given time step h_t, we have a cell state c_t. The hidden state and cell state are the same dimension, in particular N x H. Our x_t doesn't change, it's still of shape N x D. Our weight matrices however, W_{xh} and W_{hh}, are different from previously though. We'll use D x $4H$ for W_{xh} and H x $4H$ for W_{hh}. This leaves us with the following (very similar from before) equation.

$$a = x_t W_{xh} + h_{t-1} W_{hh} + b$$

to compute the *activation vector* a of shape N x $4H$ (instead of h_t as we did previously). We then split up this activation vector into four vectors: a_i, a_f, a_o, and a_g where all four vectors are of shape N x H. Each of them are pushed through the following nonlinearities:

$$i = \sigma(a_i) \quad f = \sigma(a_f) \quad o = \sigma(a_o) \quad g = tanh(a_g)$$

where $\sigma(x)$ is the sigmoid function. We then use these to compute the next cell state

$$c_t = f \odot c_{t-1} + i \odot g$$

and finally the next hidden state.

$$h_t = o \odot tanh(c_t)$$

Okay, so that's how we compute a forward pass of the LSTM architecture. But why? Well, we know it solves the vanishing and exploding gradient problem. But how? Another good question. For that, let's talk about LSTM RNN backpropagation. The implementation of the LSTM forward pass is pretty cut and dry. It's quite similar to our Elman RNN implementation just with a few more gadgets and gizmos.

3.2 LSTM Backward

Ah, the reason the LSTM architecture exists: gradient flow. Before we talk about how LSTMs solve gradient flow, let's review why Vanilla RNNs struggle to have good gradient flow for long sequences. The EECS 498 Lecture Slides pg. 86 give a nice visual for this.

Essentially, during backpropagation we must propagate through each time step h_t. By calculus, the gradient will be whatever weight matrix we multiply by h_t. In our case, we have

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

Thus, our gradient with respect to h_t at any time step is W_{hh} (with the $tanh()$ derivative in their as well). This results in a continual multiplication of matrix W_{hh} on itself for every time step in the network to propagate all the way to the beginning. As such, values greater than 1 quickly explode and those less than 1 quickly vanish, leaving us with a serious gradient flow problem.

So, how do LSTMs fix this issue? Well if you go to EECS 498 Lecture Slides pgs. 94 - 97, you can visually see the difference in gradient flow using LSTMs. We could go through the entire mechanics of backpropagation but you don't need to do so to see that LSTMs have many different connections and pathways. The information flow has a lot of options including a fair amount of additions (which play especially nice since they distribute the gradient, can't decrease, and don't explode).

This is essentially why LSTMs improve gradient flow: the allow more connections and don't repeatedly rely on a given matrix W_{hh}. This is not dissimilar to Residual Networks and their use of skip connections to improve gradient flow. In fact, befor ResNet was a thing, an LSTM like architecture was proposed to solve the same gradient issues ResNet solved. So, you can see they very much share the same intuition.

3.3 Attention

3.3.1 Intuition

Attention! Okay I'll try not to make any silly attention jokes. In all honesty, attention is one of the most intruiguing, yet difficult (the generalization to tranformers especially) concepts we've studied so far in the course.

Let's begin with some intuition. We have some data, a picture in our case, that we want to use to predict something, captions in our case. Our output is a sequence and while we could use all the data at each time step to make a prediction (for that time step), it's reasonable to want to be able to "focus" on something. That is, it's reasonable to give more attention (weight) to some parts of our data than others and letting this be a learnable parameter of our network. Hopefully, our network can then focus on what is most relevant in the data to predicting the next time step and make a better prediction because of it.

3.3.2 Basic Attention Layer

This is all fine and dandy, but how do we actually do this? Well, we compute alignment scores over our input data (our picture) and normalize them using softmax. See EECS 498 Lecture Slides pg. 15 for a picture. The computation of our alignment scores are usually given by

$$e = f_{att}(q, X)$$

where q (shape D_Q) is our *query vector*, often a time step of the encoder (which we don't have in this network) or some data (which we do have in this network). In our case, q is the image we're captioning. X is a matrix of input vectors (shape N_x x D_Q). As a note, D_Q is the same dimension as what we've been referring to as H throughout this assignment (often times q is h_{t-1} with shape H using our previous notation).

So, what's our f_{att} function? Generally just a dot product. It's simple, computationally efficient and quite robust. In particular we have

$$e = \frac{qX^T}{\sqrt{D_Q}} \quad e_i = \frac{q \cdot X_i}{\sqrt{D_Q}}$$

where e is a vector of length N_x. You may be wondering why we're dividing by $\sqrt{D_Q}$. Remember that large values don't play well with softmax, so we normalize by $\sqrt{D_Q}$. Now, our vector e is a vector of alignment scores. We apply softmax to it so convert those alignment scores to weights upon which use again on our vector X to produce an output vector Y that we'll be able to use as an input into a given time step of our network.

3.3.3