Assignment#3

Due date: 07/06 11:59 PM

This handout consists of several homework problems, as well as instructions on the "deliverables" associated with the coding portions of this assignment.

These questions require thought, but do not require long answers. Please be as concise as possible.

Not all questions will be checked.

What to expect in this assignment: In the first two assignments, we had you take tons of gradients and really dig deep into the backpropogation algorithm. In this assignment, I will treat you as the back-prop ninjas you are and turn our focus to making these models really work. Parameter exploration and understanding is crucial to master Neural Networks. This assignment should not take you long, but it is meant to give you the skills to succeed for your research projects.

1 RNN's (Recursive Neural Network)

Welcome to Technion NLP Lab (TNL): Congrats! You have just been given a Research Assistantship in TNL. Your task is to discover the power of Recursive Neural Networks (RNNs). So you plan an experiment to show their effectiveness on Positive/Negative Sentiment Analysis. In this part, you will derive the forward and backpropogation equations, implement them, and test the results.

We will assume the RNN has one ReLU layer and one softmax layer, and uses Cross Entropy loss as its cost function. We follow the parse tree given from the leaf nodes up to the top of the tree and evaluate the cost at each node. During backprop, we follow the exact opposite path. Figure 1 shows an example of such a RNN applied to a simple sentence "I love this assignment". These equations are sufficient to explain our model:

$$CE(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_i y_i \log(\hat{y}_i)$$

where y is the one-hot label vector, and $\hat{y}$ is the predicted probability vector for all classes. In our case, $y \in R^5$ and $\hat{y} \in R^5$ to represent our 5 sentiment classes: Really Negative, Negative, Neutral, Positive, and Really Positive. Furthermore,

$$h^{(1)} = \max(W^{(1)} \begin{bmatrix} h_{Left}^{(1)} \\ h_{Right}^{(1)} \end{bmatrix} + b^{(1)}, 0)$$

$$\hat{y} = \text{softmax}(Uh^{(1)} + b^{(s)})$$

where $h_{Left}^{(1)}$ is the output of the layer beneath it on the left (and could be a word vector), the same for $h_{Right}^{(1)}$ but coming from the right side. Assume $L_i \in R^d, \forall i \in [1 \dots |V|], W^{(1)} \in \mathbb{R}^{dx2d}, b^{(1)} \in \mathbb{R}^d, U \in \mathbb{R}^{5xd}$, and $b^{(s)} \in \mathbb{R}^5$

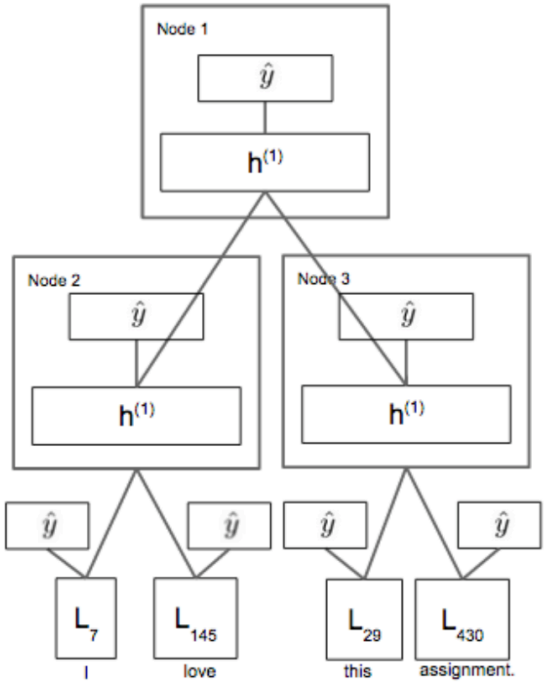

Figure 1: RNN (Recursive Neural Network) example

(a) Follow the example parse tree in Figure 1 in which we are given a parse tree and truth labels y for each node. Starting with Node 1, then to Node 2, finishing with Node 3, write the update rules for $W^{(1)}, b^{(1)}, U, b^{(s)}$, and L after the evaluation of $\tilde{y}$ against our truth, y. This means for at each node, we evaluate:

$$\delta_3 = \tilde{y} - y$$

as our first error vector and we backpropogate that error through the network, aggregating gradient at each node for:

$$\frac{\partial J}{\partial U} \quad \frac{\partial J}{\partial b^{(s)}} \quad \frac{\partial J}{\partial W^{(1)}} \quad \frac{\partial J}{\partial b^{(1)}} \quad \frac{\partial J}{\partial L_i}$$

Points will be deducted if you do not express the derivative of activation functions (ReLU) in terms of their function values (as with Assignment 1 and 2) or do not express the gradients by using an "error vector" (δ_i) propagated back to each layer. Tip on notation: δ_{below} and δ_{above} should be used for error that is being sent down to the next node, or came from an above node. This will help you think about the problem in the right way. Note you should not be updating gradients for L_i in Node 1. But error should be leaving Node 1 for sure!

(b) Implementation time! Now that you have a feel for how to train this network:

- (a) Download, unzip, and have a look at the code base.
- (b) From the command line, run ./setup.sh. This should download the labeled parse tree dataset and setup the environment for you (model folders, etc).
- (c) Now lets peruse the code base. You should start with runNNet.pyto get a grasp for how the network is run and tested. You should also take a peek at tree.pyto understand what the Node class is, and how we traverse a parse tree and what fields we update during forward pass and backward pass (you need to know what hActs is!). Next, take a look at run.sh. This shell script contains all the parameters needed to train the model. It is important to get this right. You should update these environment parameters here and only here when you are ready.
  (d) Finally, open rnn.py. There are two functions left for you to implement, forwardProp and backProp. Implement them!
- (e) When you are ready to test your implementation, run python rnn.py to perform gradient check. You can make use of pdb and set trace as you code to give you insights into what's happening under the hood! **It is expected of you to take the time to fully understand how this code base functions!** This way you can perhaps use it as a starting point for your projects (or any other codebase one might give you). Also, if you are unfamiliar with pdb.set trace, definitely take 5 minutes and learn about it!

(c) Test time! From the command line, run ./run.sh. This will train the model with the parameters you specified in run.shand produce a pickled neural network that we can test later via ./test.sh. Note the training could take about an hour or more pending how many epochs you allow. Once the training is done, you should open test.shchange the test data type to devand run ./test.shfrom the command line. This should output your dev set accuracy.

Your task here is to produce four plots.

- (a) First, provide a plot showing the training error and dev error over epochs for a $wvecdim = 30$. You should see a point where the dev error actually begins to increase. If you do not, you probably did not run for enough epochs. Note the number of epochs that is best on your dev set.
- (b) Provide a one-sentence intuition why this happens.
- (c) Next, produce 2 confusion matrices on the above environment parameters (with your optimal number of epochs), one for the training set, and one for the dev set. The function *makeconf* might be handy.
  A confusion matrix is a great way to see what you are getting wrong. In Figure 2, we see that the model is very accurate for matching a 1 label with a 1. However, it confuses a 4 for a 0 very often. You should take a second to make sense of this plot.

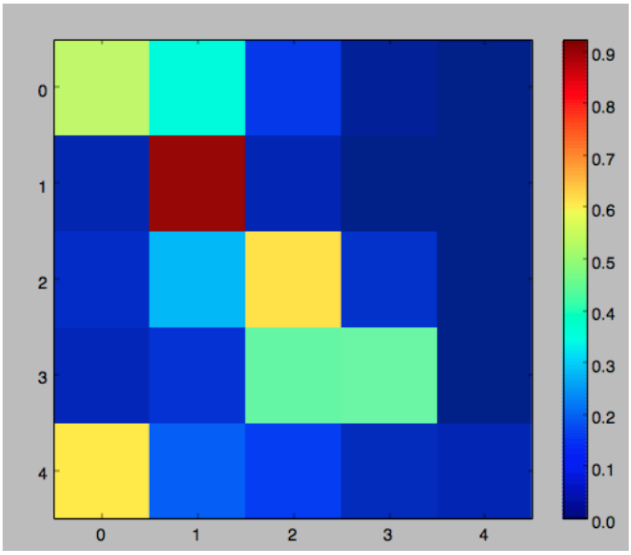

Figure 2: Confusion Matrix with truth down the y axis, and our models guess across the x axis

- (d) Finally, provide a plot of dev accuracy vs wvecdim with the same epochs as above. Reasonable values for wvecdim would be 5, 15, 25, 35, 45. Note to generate all of this data for the plots, you must train a number of models that will take some time, so it would NOT be wise to run one at a time but rather, let many of them train over night on different prompts. These are the real pains data scientists must deal with! If you are running this on myth, corn or your own server somewhere, look up the linux command: screen. It will help tremendously.

2 2-Layer Deep RNN's

Lets go deeper: Your advisor is impressed with your results. She mentions in your conversation that perhaps the model is just not expressive enough. This gets you thinking. What if we add a layer in between the first layer and the softmax layer to help increase score accuracy!Assume the same assumptions as in the first RNN, but now we have one more layer such that $W^{(2)} \in \mathbb{R}^{d_{middle}xd}$, $b^{(2)} \in \mathbb{R}^{d_{middle}}$ and $U \in \mathbb{R}^{5xd_{middle}}$. The equations below should be sufficient to explain the model

$$h^{(1)} = \max(W^{(1)} \begin{bmatrix} h^{(1)}_{Left} \\ h^{(1)}_{Right} \end{bmatrix} + b^{(1)}, 0)$$

$$h^{(2)} = \max(W^{(2)} h^{(1)} + b^{(2)}, 0)$$

$$\hat{y} = \text{softmax}(U h^{(2)} + b^{(s)})$$

(a) Perform the same analysis for the example in Figure 3. The updates starting at Node 1, to Node 2, and finally for Node 3 for:

$$\frac{\partial J}{\partial U} \quad \frac{\partial J}{\partial b^{(s)}} \quad \frac{\partial J}{\partial W^{(1)}} \quad \frac{\partial J}{\partial b^{(1)}} \quad \frac{\partial J}{\partial W^{(2)}} \quad \frac{\partial J}{\partial b^{(2)}} \quad \frac{\partial J}{\partial L_i}$$

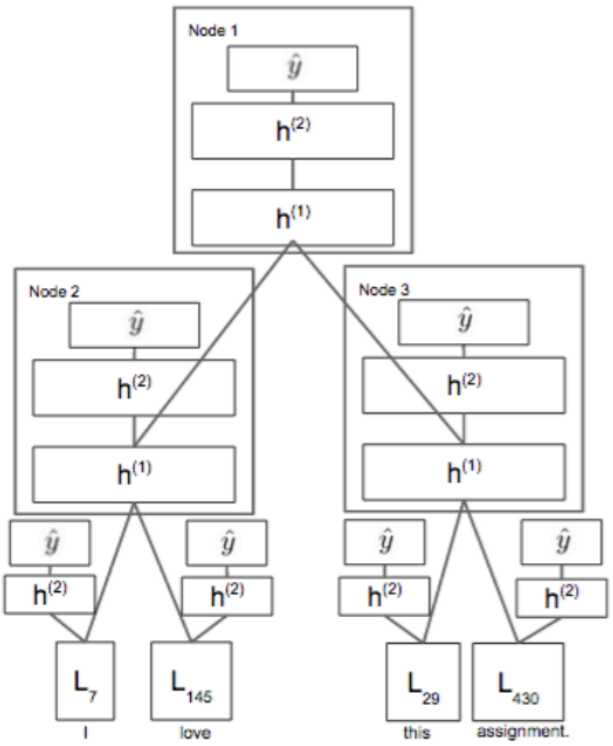

Figure 3: 2-Layer RNN example

Points will be deducted if you do not express the derivative of activation functions (ReLU) in terms of their function values (as with Assignment 1 and 2) or do not express the gradients by using an "error vector" (δ_i) propagated back to each layer. Tip on notation: δ_{below} and δ_{above} should be used for error that is being sent down to the next node, or came from an above node. This will help you think about the problem in the right way.

(b) Implementation time again

a. First, open rnn2deep.py. There are two functions left for you to implement, forwardProp and backProp

b. When you are ready run python rnn2deep.py to run gradient check. You can make use of pdb and set trace as you code to give you insights into what's happening under the hood!

c. Now, lets take a look at run.sh and the middleDim parameter. This is what you will use to augment the size of the middle layer in your 2-Layer Network. Also, remember to update model to be RNN2 instead of RNN.

(c) Test time! From the command line, run ./run.sh. This will train the model with the parameters you specified in run.sh and produce a pickled neural network that we can test later. Note the training could take about an hour or more. Once the training is done, you should open test.sh change the test data type to dev and model to RNN2 and run ./test.sh. This should give you your dev set accuracy.

Your task here is to produce four more plots.

(a) First, a plot showing the training error and dev error over epochs for a wvecdim=30 and middleDim=30.

(b) Second, a plot showing a confusion matrix on the train and another one on the dev sets using the number of epochs from above. You might find makeconf in runNNet.py useful.

(c) Provide a two sentence intuition for why the model is doing better or worse than the first RNN.

(d) Next, provide a plot of dev accuracy vs middleDim. Reasonable values for middleDim would be 5, 15, 25, 35, 45 while wvecdim=30 and a constant number of epochs (found in part (c)(a) above). Note to generate all of this data for the plots, you must train a number of models that will take some time, so it would NOT be wise to run one

at a time but rather, let many of them train over night on different prompts.

(d) Suggest a change! Take a moment and observe the errors that your model is making. Does your model do better on Negative scores? Positive scores? Suggest a change that would correct for that error. One example change would be, add a "depth level index" to the model input making your input to the neural network $\in R^{2d+1}$, so the model knows where in the tree it is. i.e. at the base level this index would be 0, at the next level it would be 1, etc, etc. Another example change could be to add ANOTHER layer. What might that do for the model? A final suggestion would be to make the error from above flow into $h^{(2)}$ rather than $h^{(1)}$. This question is asking you to dig deep in the data, think outside the box and come up with a scheme that will boost performance. A complete answer will be original, based on data observation, and should have a reasonable expectation to boost performance.

(e) Extra Credit: Implement that change! Take your change you suggested, open rnn changed.py file, and implement a new model called RNN3. Run the model and optimize it, as done in the previous problems of this assignment. Report your findings with a train and dev accuracy plot. And comment on your findings.

(f) Extra Credit: Implement Dropout on the softmax layer. Dropout is a regularization technique where we randomly "drop" nodes in the Neural Network during training. When we apply our input, we can just randomly select certain nodes to not fire, by setting there output to 0 even though it might not have been. We just need to remember at backprop time, NOT to update those weights. Hinton describes this method as training 2^N separate neural networks (if N is the number of nodes in the network), and we take the average of the results. This method has been shown to improve neural network's performance by ensuring that each node is useful on its own. Apply this method on the RNN2 model and report your results as you did in the previous portions of the assignment.

3 Extra Credit: Recursive Neural Tensor Networks

Derive the gradients and updates as done in part 1 and 2 of this assignment. Then, using the starter codein rntn.py, implement the model detailed in Richard's paper titled "Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank":

Socher, Richard, et al. "Recursive deep models for semantic compositionality over a sentiment treebank." Proceedings of the

conference on empirical methods in natural language processing (EMNLP). Vol. 1631. 2013.

Report your findings in similar fashion to the previous 2 problems.