CS 224d: Assignment #2

Sunday 8th May, 2016

Due date: 5/5 11:59 PM PST (You are allowed to use three (3) late days maximum for this assignment)

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.

We encourage students to discuss in groups for assignments. However, each student finish the problem set and programming assignment individually, and must turn in her/his assignment. We ask that you abide the university Honor Code and that of the Computer Science department, and make sure that all of your submitted work are done by yourself.

Please review any additional instructions posted on the assignment page at http://cs224d.stanford.edu/assignments.html. When you are ready to submit, please follow the instructions on the course website.

1 Tensorflow Softmax (20 points)

In this question, we will implement a linear classifier with loss function

$$J_{softmax-CE}(\boldsymbol{W}) = CE(\boldsymbol{y}, \text{softmax}(\boldsymbol{x}\boldsymbol{W}))$$

(Here the rows of $\boldsymbol{x}$ are feature vectors). We will use TensorFlow's automatic differentiation capability to fit this model to provided data.

(a) (4 points) Implement the softmax function using TensorFlow in `q1_softmax.py`. Remember that

$$softmax(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{1}$$

Note that you may **not** use `tf.nn.softmax` or related built-in functions. You can run basic (non-exhaustive tests) by running `python q1_softmax.py`.

(b) (4 points) Implement the cross-entropy loss using TensorFlow in `q1_softmax.py`. Remember that

$$CE(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_{i=1}^{N_c} y_i \log(\hat{y}_i) \tag{2}$$

where $\boldsymbol{y} \in \mathbb{R}^5$ is a one-hot label vector and N_c is the number of classes. Note that you may **not** use TensorFlow's built-in cross-entropy functions for this question. You can run basic (non-exhaustive tests) by running `python q1_softmax.py`.

(c) (4 points) Carefully study the `Model` class in `model.py`. Briefly explain the purpose of placeholder variables and feed dictionaries in tensorflow computations. Fill in the implementations for the `add_placeholders`, `create_feed_dict` in `q1_classifier.py`. **Hint:** Note that configuration variables are stored in the `Config` class. You will need to use these configuration variables in the code.

Solution: Placeholder variables act as input nodes in the tensorflow computational graph. Feed dictionaries are how users set values for placeholder (or other) variables when running a computation.

(d) (4 points) Implement the transformation for a softmax classifier in function `add_model` in `q1_classifier.py`. Add cross-entropy loss in function `add_loss_op` in the same file. Use the implementations from the earlier parts of the problem, **not** TensorFlow built-ins.

(e) (4 points) Fill in the implementation for `add_training_op` in `q1_classifier.py`. Explain how TensorFlow's automatic differentiation removes the need for us to define gradients explicitly. Verify that your model is able to fit to synthetic data by running `python q1_classifier.py` and making sure that the tests pass. **Hint:** Make sure to use the learning rate specified in `Config`.

Solution: So long as the computational graph is defined properly, Tensorflow can automatically apply back-propagation to compute gradients.

2 Deep Networks for Named Entity Recognition (35 points)

In this section, we'll get to practice backpropagation and training deep networks to attack the task of Named Entity Recognition: predicting whether a given word, in context, represents one of four categories:

- Person (`PER`)
- Organization (`ORG`)
- Location (`LOC`)
- Miscellaneous (`MISC`)

We formulate this as a 5-class classification problem, using the four above classes and a null-class (`O`) for words that do not represent a named entity (most words fall into this category).

The model is a 1-hidden-layer neural network, with an additional representation layer similar to what you saw with `word2vec`. Rather than averaging or sampling, here we explicitly represent context as a "window" consisting of a word concatenated with its immediate neighbors:

$$\boldsymbol{x}^{(t)} = [\boldsymbol{x}_{t-1}\boldsymbol{L},\ \boldsymbol{x}_t\boldsymbol{L},\ \boldsymbol{x}_{t+1}\boldsymbol{L}] \in \mathbb{R}^{3d} \tag{3}$$

where the input $\boldsymbol{x}_{t-1}, \boldsymbol{x}_t, \boldsymbol{x}_{t+1}$ are one-hot row vectors into an embedding matrix $\boldsymbol{L} \in \mathbb{R}^{|V| \times d}$, with each row $\boldsymbol{L}_i$ as the vector for a particular word $i = \boldsymbol{x}_t$. We then compute our prediction as:

$$\boldsymbol{h} = \tanh(\boldsymbol{x}^{(t)}\boldsymbol{W} + \boldsymbol{b}_1) \tag{4}$$

$$\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{h}\boldsymbol{U} + \boldsymbol{b}_2) \tag{5}$$

And evaluate by cross-entropy loss

$$J(\theta) = \text{CE}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_{i=1}^{5} y_i \log \hat{y}_i \tag{6}$$

To compute the loss for the training set, we sum (or average) this $J(\theta)$ as computed with respect to each training example.

For this problem, we let $d = 50$ be the length of our word vectors, which are concatenated into a window of width $3 \times 50 = 150$. The hidden layer has a dimension of 100, and the output layer $\hat{y}$ has a dimension of 5.

(a) (5 points) Compute the gradients of $J(\theta)$ with respect to all the model parameters:

$$\frac{\partial J}{\partial \boldsymbol{U}} \qquad \frac{\partial J}{\partial \boldsymbol{b}_2} \qquad \frac{\partial J}{\partial \boldsymbol{W}} \qquad \frac{\partial J}{\partial \boldsymbol{b}_1} \qquad \frac{\partial J}{\partial \boldsymbol{L}_i}$$

where

$$\boldsymbol{U} \in \mathbb{R}^{100\times 5} \quad \boldsymbol{b}_2 \in \mathbb{R}^5 \quad \boldsymbol{W} \in \mathbb{R}^{150\times 100} \quad \boldsymbol{b}_1 \in \mathbb{R}^{100} \quad \boldsymbol{L}_i \in \mathbb{R}^{50}$$

In the spirit of backpropagation, you should express the derivative of activation functions (tanh, softmax) in terms of their function values (as with sigmoid in Assignment 1). This identity may be helpful:

$$\tanh(z) = 2\ \text{sigmoid}(2z) - 1$$

Furthermore, you should express the gradients by using an "error vector" propagated back to each layer; this just amounts to putting parentheses around factors in the chain rule, and will greatly simplify your analysis. All resulting gradients should have simple, closed-form expressions in terms of matrix operations. (*Hint: you've already done most of the work here as part of Assignment 1.*)

Solution: Note that

$$\tanh'(z) = 4\text{sigmoid}(2z)(1 - \text{sigmoid}(2z))$$

The requested partial derivatives are then

$$\begin{aligned}
\frac{\partial J}{\partial \boldsymbol{U}} &= \boldsymbol{h}^T(\boldsymbol{y} - \hat{\boldsymbol{y}}) \\
\frac{\partial J}{\partial \boldsymbol{b}_2} &= (\boldsymbol{y} - \hat{\boldsymbol{y}}) \\
\frac{\partial J}{\partial \boldsymbol{h}} &= (\boldsymbol{y} - \hat{\boldsymbol{y}})\boldsymbol{U}^T \\
\frac{\partial J}{\partial \boldsymbol{W}} &= (\boldsymbol{x}^{(t)})^T \left(\frac{\partial J}{\partial \boldsymbol{h}} \odot \tanh'(2(\boldsymbol{x}^{(t)}\boldsymbol{W} + \boldsymbol{b}_1))\right) \\
\frac{\partial J}{\partial \boldsymbol{b}_1} &= \left(\frac{\partial J}{\partial \boldsymbol{h}} \odot \tanh'(2(\boldsymbol{x}^{(t)}\boldsymbol{W} + \boldsymbol{b}_1))\right) \\
\frac{\partial J}{\partial \boldsymbol{x}^{(t)}} &= \left(\frac{\partial J}{\partial \boldsymbol{h}} \odot \tanh'(2(\boldsymbol{x}^{(t)}\boldsymbol{W} + \boldsymbol{b}_1))\right) \boldsymbol{W}^T
\end{aligned}$$

Let $i = \boldsymbol{x}_t, j = \boldsymbol{x}_{t-1}, k = \boldsymbol{x}_{t+1}$. Note that

$$\frac{\partial J}{\partial \boldsymbol{x}^{(t)}} = \left[\frac{\partial J}{\partial \boldsymbol{L}_i}, \frac{\partial J}{\partial \boldsymbol{L}_j}, \frac{\partial J}{\partial \boldsymbol{L}_k}\right]$$

(b) (5 points) To avoid parameters from exploding or becoming highly correlated, it is helpful to augment our cost function with a Gaussian prior: this tends to push parameter weights closer to zero, without constraining their direction, and often leads to classifiers with better generalization ability.

If we maximize log-likelihood (as with the cross-entropy loss, above), then the Gaussian prior becomes a quadratic term [1] (L2 regularization):

$$J_{reg}(\theta) = \frac{\lambda}{2}\left[\sum_{i,j} W_{ij}^2 + \sum_{i'j'} U_{i'j'}^2\right] \tag{7}$$

and we optimize the combined loss function

$$J_{full}(\theta) = J(\theta) + J_{reg}(\theta) \tag{8}$$

[1] **Optional (not graded):** The interested reader should prove that this is indeed the maximum-likelihood objective when we let $W_{ij} \sim N(0, 1/\lambda)$ for all i, j.

Update your gradients from part (a) to include the additional term in this loss function (i.e. compute $\frac{dJ_{full}}{dW}$, etc.).

Solution: Note that

$$\partial J_{full} = \partial J + \partial J_{reg}$$
$$\frac{\partial J_{reg}}{\partial \boldsymbol{W}} = \lambda \boldsymbol{W}$$
$$\frac{\partial J_{reg}}{\partial \boldsymbol{U}} = \lambda \boldsymbol{U}$$

Thus these gradients for W and U have to be added to the corresponding gradients from the previous question.

(c) (5 points) In order to avoid neurons becoming too correlated and ending up in poor local minimina, it is often helpful to randomly initialize parameters. One of the most frequent initializations used is called Xavier initialization[2].

Given a matrix $\boldsymbol{A}$ of dimension $m \times n$, select values A_{ij} uniformly from $[-\epsilon, \epsilon]$, where

$$\epsilon = \frac{\sqrt{6}}{\sqrt{m+n}} \tag{9}$$

Implement the initialization for use in `xavier_weight_init` in `q2_initialization.py` and use it for the weights $\boldsymbol{W}$ and $\boldsymbol{U}$.

(d) (20 points) In `q2_NER.py` implement the NER window model by filling in the appropriate sections. The gradients you derived in (a) and (b) will be computed for you automatically, showing the benefits that automatic differentiation can provide for rapid prototyping.

Run `python q2_NER.py` to evaluate your model's performance on the dev set, and compute predictions on the test data (make sure to turn off debug settings when doing final evaluation). Note that the test set has only dummy labels; we'll compare your predictions against the ground truth after you submit.

Deliverables:

- Working implementation of the NER window model in `q2_NER.py`. (We'll look at, and possibly run this code for grading.)
- In your writeup (i.e. where you're writing the answers to the written problems), *briefly* state the optimal hyperparameters you found for your model: regularization, dimensions, learning rate (including time-varying, such as annealing), SGD batch size, etc. Report the performance of your model on the validation set. **You should be able to get validation loss below** 0.2**.**
- List of predicted labels for the `test` set, one per line, in the file `q2_test.predicted`.
- **Hint:** When debugging, set `max_epochs = 1`. Pass the keyword argument `debug=True` to the call to `load_data` in the `__init__` method.
- **Hint:** This code should run within 15 minutes on a GPU and 1 hour on a CPU.

[2]This is also referred to as Glorot initialization and was initially described in `http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf`

3 Recurrent Neural Networks: Language Modeling (45 points)

In this section, you'll implement your first recurrent neural network (RNN) for building a language model.

Language modeling is a central task in NLP, and language models can be found at the heart of speech recognition, machine translation, and many other systems. Given words $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t$, a language model predicts the following word $\boldsymbol{x}_{t+1}$ by modeling:

$$P(\boldsymbol{x}_{t+1} = \boldsymbol{v}_j \mid \boldsymbol{x}_t, \ldots, \boldsymbol{x}_1)$$

where v_j is a word in the vocabulary.

Your job is to implement a recurrent neural network language model, which uses feedback information in the hidden layer to model the "history" $x_t, x_{t-1}, \ldots, x_1$. Formally, the model[3] is, for $t = 1, \ldots, n-1$:

$$\begin{aligned}
\boldsymbol{e}^{(t)} &= \boldsymbol{x}^{(t)}\boldsymbol{L} && (10)\\
\boldsymbol{h}^{(t)} &= \text{sigmoid}\left(\boldsymbol{h}^{(t-1)}\boldsymbol{H} + e^{(t)}\boldsymbol{I} + \boldsymbol{b}_1\right) && (11)\\
\hat{\boldsymbol{y}}^{(t)} &= \text{softmax}\left(\boldsymbol{h}^{(t)}\boldsymbol{U} + \boldsymbol{b}_2\right) && (12)\\
\bar{P}(\boldsymbol{x}_{t+1} = \boldsymbol{v}_j \mid \boldsymbol{x}_t, \ldots, \boldsymbol{x}_1) &= \hat{y}_j^{(t)} && (13)
\end{aligned}$$

where $\boldsymbol{h}^{(0)} = \boldsymbol{h}_0 \in \mathbb{R}^{D_h}$ is some initialization vector for the hidden layer and $\boldsymbol{x}^{(t)}\boldsymbol{L}$ is the product of $\boldsymbol{L}$ with the one-hot row-vector $\boldsymbol{x}^{(t)}$ representing index of the current word. The parameters are:

$$\boldsymbol{L} \in \mathbb{R}^{|V| \times d} \quad \boldsymbol{H} \in \mathbb{R}^{D_h \times D_h} \quad \boldsymbol{I} \in \mathbb{R}^{d \times D_h} \quad \boldsymbol{b}_1 \in \mathbb{R}^{D_h} \quad \boldsymbol{U} \in \mathbb{R}^{D_h \times |V|} \quad \boldsymbol{b}_2 \in \mathbb{R}^{|V|} \tag{14}$$

where $\boldsymbol{L}$ is the embedding matrix, $\boldsymbol{I}$ the input word representation matrix, $\boldsymbol{H}$ the hidden transformation matrix, and $\boldsymbol{U}$ is the output word representation matrix. $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are biases. d is the embedding dimension, $|V|$ is the vocabulary size, and D_h is the hidden layer dimension.

The output vector $\hat{\boldsymbol{y}}^{(t)} \in \mathbb{R}^{|V|}$ is a probability distribution over the vocabulary, and we optimize the (unregularized) cross-entropy loss:

$$J^{(t)}(\theta) = \text{CE}(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = -\sum_{i=1}^{|V|} y_i^{(t)} \log \hat{y}_i^{(t)} \tag{15}$$

where $\boldsymbol{y}^{(t)}$ is the one-hot vector corresponding to the target word (which here is equal to x_{t+1}). As in Q 2, this is a point-wise loss, and we sum (or average) the cross-entropy loss across all examples in a sequence, across all sequences[4] in the dataset in order to evaluate model performance.

(a) (5 points) Conventionally, when reporting performance of a language model, we evaluate on *perplexity*, which is defined as:

$$\text{PP}^{(t)}\left(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}\right) = \frac{1}{\bar{P}(\boldsymbol{x}_{t+1}^{\text{pred}} = \boldsymbol{x}_{t+1} \mid \boldsymbol{x}_t, \ldots, \boldsymbol{x}_1)} = \frac{1}{\sum_{j=1}^{|V|} y_j^{(t)} \cdot \hat{y}_j^{(t)}} \tag{16}$$

i.e. the inverse probability of the correct word, according to the model distribution $\bar{P}$. Show how you can derive perplexity from the cross-entropy loss (*Hint: remember that $y^{(t)}$ is one-hot!*), and thus argue that

[3] This model is adapted from a paper by Toma Mikolov, et al. from 2010: `http://www.fit.vutbr.cz/research/groups/speech/publi/2010/mikolov_interspeech2010_IS100722.pdf`

[4] We use the tensorflow function `sequence_loss` to do this.

minimizing the (arithmetic) mean cross-entropy loss will also minimize the (geometric) mean perplexity across the training set. ***This should be a very short problem - not too perplexing!***

For a vocabulary of $|V|$ words, what would you expect perplexity to be if your model predictions were completely random? Compute the corresponding cross-entropy loss for $|V| = 2000$ and $|V| = 10000$, and keep this in mind as a baseline.

Solution: Using the fact that $y^{(t)}$ is one-hot, suppose that $y_i^{(t)}$ is the only nonzero element of $y^{(t)}$. Then, note that

$$\mathrm{CE}(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = -\log \hat{y}_i^{(t)} = \log \frac{1}{\hat{y}^{(t)}}$$
$$\mathrm{PP}^{(t)}(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = \frac{1}{\hat{y}_i^{(t)}}$$

Thus, it follows that

$$\mathrm{CE}(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = \log \mathrm{PP}^{(t)}(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)})$$

It follows that minimizing the arithmetic mean of the cross-entropy is identical to minimizing the geometric mean of the perplexity. If the model predictions are completely random, $E[\hat{y}_i^{(t)}] = \frac{1}{|V|}$. Given that $y^{(t)}$ is one-hot, it follows that the expected value of the perplexity is $|V|$. Since the cross-entropy is the logarithm of perplexity, the expected cross-entropies are $\log 2000 \approx 7.6$ and $\log 10000 \approx 9.21$.

(b) (5 points) As you did in Q 2, compute the gradients with for all the model parameters at a single point in time t:

$$\frac{\partial J^{(t)}}{\partial \boldsymbol{U}} \qquad \frac{\partial J^{(t)}}{\partial \boldsymbol{b}_2} \qquad \frac{\partial J^{(t)}}{\partial \boldsymbol{L}_{x^{(t)}}} \qquad \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{I}}\right|_{(t)} \qquad \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{H}}\right|_{(t)} \qquad \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{b}_1}\right|_{(t)}$$

where $\boldsymbol{L}_{\boldsymbol{x}^{(t)}}$ is the column of $\boldsymbol{L}$ corresponding to the current word $\boldsymbol{x}^{(t)}$, and $|_{(t)}$ denotes the gradient for the appearance of that parameter at time t. (Equivalently, $\boldsymbol{h}^{(t-1)}$ is taken to be fixed, and you need not backpropagate to earlier timesteps just yet - you'll do that in part (c)).

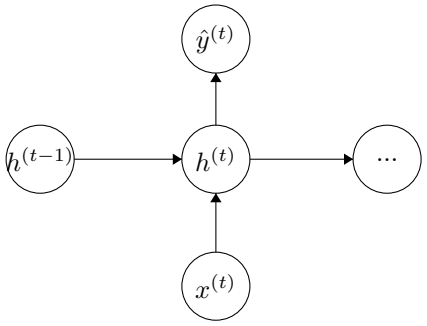
Additionally, compute the derivative with respect to the *previous* hidden layer value:

$$\frac{\partial J^{(t)}}{\partial \boldsymbol{h}^{(t-1)}}$$

Solution: Recall that $\frac{d}{dz}\text{sigmoid}(z) = \text{sigmoid}(z)(1 - \text{sigmoid}(z))$.

$$\begin{aligned}
\frac{\partial J^{(t)}}{\partial \boldsymbol{U}} &= (\boldsymbol{h}^{(t)})^T(\boldsymbol{y} - \hat{\boldsymbol{y}}) \\
\frac{\partial J^{(t)}}{\partial \boldsymbol{h}^{(t)}} &= (\boldsymbol{y} - \hat{\boldsymbol{y}})\boldsymbol{U}^T \\
\frac{\partial J^{(t)}}{\partial \boldsymbol{b}_2} &= (\boldsymbol{y} - \hat{\boldsymbol{y}}) \\
\left.\frac{\partial J^{(t)}}{\partial \boldsymbol{b}_1}\right|_{(t)} &= \left(\frac{\partial J^{(t)}}{\partial \boldsymbol{h}^{(t)}} \odot \text{sigmoid}'(\boldsymbol{h}^{(t-1)}\boldsymbol{H} + e^{(t)}\boldsymbol{I} + \boldsymbol{b}_1)\right) \\
\frac{\partial J^{(t)}}{\partial \boldsymbol{L}_{x^{(t)}}} &= \frac{\partial J^{(t)}}{\partial \boldsymbol{e}^{(t)}} = \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{b}_1}\right|_{(t)} \boldsymbol{I}^T \\
\left.\frac{\partial J^{(t)}}{\partial \boldsymbol{I}}\right|_{(t)} &= (\boldsymbol{e}^{(t)})^T \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{b}_1}\right|_{(t)} \\
\left.\frac{\partial J^{(t)}}{\partial \boldsymbol{H}}\right|_{(t)} &= (h^{(t-1)})^T \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{b}_1}\right|_{(t)} \\
\frac{\partial J^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} &= \boldsymbol{H}^T \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{b}_1}\right|_{(t)}
\end{aligned}$$

(c) (5 points) Below is a sketch of the network at a single timestep:



Draw the "unrolled" network for 3 timesteps, and compute the backpropagation-through-time gradients:

$$\frac{\partial J^{(t)}}{\partial \boldsymbol{L}_{x^{(t-1)}}} \qquad \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{H}}\right|_{(t-1)} \qquad \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{I}}\right|_{(t-1)} \qquad \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{b}_1}\right|_{(t-1)}$$

where $\big|_{(t-1)}$ denotes the gradient for the appearance of that parameter at time $(t-1)$. Because parameters are used multiple times in feed-forward computation, we need to compute the gradient for each time they appear.

You should use the backpropagation rules from Lecture 5 [5] to express these derivatives in terms of error term

$$\boldsymbol{\delta}^{(t-1)} = \frac{\partial J^{(t)}}{\partial \boldsymbol{h}^{(t-1)}}$$

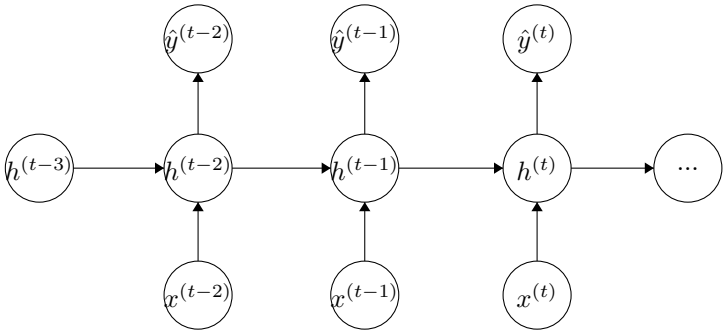
computed in the previous part. (Doing so will allow for re-use of expressions for $t-2$, $t-3$, and so on).

Note that the true gradient with respect to a training example requires us to run backpropagation all

[5] http://cs224d.stanford.edu/lectures/CS224d-Lecture5.pdf

the way back to $t = 0$. In practice, however, we generally truncate this and only backpropagate for a fixed number $\tau \approx 3 - 5$ timesteps.

Solution:



Using terms from the previous part

$$\left.\frac{\partial J^{(t)}}{\partial \boldsymbol{b}_1}\right|_{(t-1)} = \boldsymbol{\delta}^{(t-1)} \odot \text{sigmoid}'(\boldsymbol{h}^{(t-2)}\boldsymbol{H} + e^{(t-1)}\boldsymbol{I} + \boldsymbol{b}_1)$$

$$\frac{\partial J^{(t)}}{\partial \boldsymbol{L}_{x^{(t-1)}}} = \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{b}_1}\right|_{(t-1)} \boldsymbol{I}^T$$

$$\left.\frac{\partial J^{(t)}}{\partial \boldsymbol{I}}\right|_{(t-1)} = (\boldsymbol{e}^{(t-1)})^T \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{b}_1}\right|_{(t-1)}$$

$$\left.\frac{\partial J^{(t)}}{\partial \boldsymbol{H}}\right|_{(t-1)} = (\boldsymbol{h}^{(t-2)})^T \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{b}_1}\right|_{(t-1)}$$

(d) (3 points) Given $\boldsymbol{h}^{(t-1)}$, how many operations are required to perform one step of forward propagation to compute $J^{(t)}(\theta)$? How about backpropagation for a single step in time? For τ steps in time? Express your answer in big-O notation in terms of the dimensions d, D_h and $|V|$(Equation 14). What is the slow step?

Recall that $\boldsymbol{h}^{(t-1)}$ is of size D_h, $\boldsymbol{e}^{(t)}$ is of size d, and $\hat{\boldsymbol{y}}^{(t)}$ is of size $|V|$. Computing $\boldsymbol{e}^{(t)}$ only takes time $O(d)$ since it is a lookup into matrix $\boldsymbol{L}$. Computing $\boldsymbol{h}^{(t)}$ then takes time $O(D_h^2 + dD_h)$, and computing $\hat{\boldsymbol{y}}^{(t)}$ takes time $O(D_h|V|)$. The backward pass and the forward pass have the same time-complexity, and computing τ steps of forward or backward propagation simply multiplies the previous complexities by τ. The slow step is the computation of $\hat{\boldsymbol{y}}^{(t)}$ taking $O(D_h|V|)$ since $|V|$ will be large.

(e) (20 points) Implement the above model in `q3_RNNLM.py`. Data loaders and other starter code are provided. Follow the directions in the code to understand which parts need to be filled in. Running `python q3_RNNLM.py` will run the model. Note that you may **not** use built-in tensorflow functions such as those in the `rnn_cell` module.

Train a model on the `ptb-train` data, consisting of the first 20 sections of the WSJ corpus of the Penn Treebank. As in Q 2, you should tune your model to maximize generalization performance (minimize cross-entropy loss) on the dev set. We'll evaluate your model on an unseen, but similar set of sentences.

Deliverables:

- In your writeup, include the best hyperparameters you used (training schedule, number of iterations, learning rate, backprop timesteps), and your perplexity score on the `ptb-dev` set. You should be able to get validation perplexity below 175.
- Include your saved model parameters for your best model; we'll use these to test your model.
- **Hint:** When debugging, set `max_epochs = 1`. Pass the keyword argument `debug=True` to the call to `load_data` in the `__init__` method.
- **Hint:** On a GPU, this code should run quickly (below 30 minutes). On a CPU, the code may take up to 4 hrs to run.

(f) (7 points) The networks that you've seen in Assignment 1 and in q2 of this assignment are discriminative models: they take data, and make a prediction. The RNNLM model you've just implemented is a *generative* model, in that it actually models the distribution of the *data* sequence $x_1, \ldots, x_n$. This means that not only can we use it to evaluate the likelihood of a sentence, but we can actually use it to generate one!

After training, in `q3_RNNLM.py`, implement the `generate_text()` function. This should run the RNN forward in time, beginning with the index for the start token `<eos>`, and sampling a new word $\boldsymbol{x}_{t+1}$ from the distribution $\hat{\boldsymbol{y}}^{(t)}$ at each timestep. Then feed this word in as input at the next step, and repeat until the model emits an `end` token (index of `</eos>`).

Deliverables:

- Include 2-3 generated sentences in your writeup. See if you can generate something humorous!

Completely optional, not graded: If you want to experiment further with language models, you're welcome to load up your own texts and train on them - sometimes the results can be quite entertaining! (See `http://kingjamesprogramming.tumblr.com/` for a great one[6] trained on a mix of the King James Bible and the Structure and Interpretation of Computer Programs.)

[6] This one just uses a simple n-gram Markov model, but there's no reason an RNNLM can't compete!