1 Tensorflow Softmax (20 points)

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

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

(Here the rows of $x$ 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

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

(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

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

(2)

where $y \in \mathbb{R}^5$ 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.
(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.

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:

\[ x(t) = [x_{t-1}L, x_tL, x_{t+1}L] \in \mathbb{R}^{3d} \quad (3) \]

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

\[
\begin{align*}
    h &= \tanh(x^{(t)}W + b_1) \quad (4) \\
    \hat{y} &= \text{softmax}(hU + b_2) \quad (5)
\end{align*}
\]

And evaluate by cross-entropy loss

\[ J(\theta) = CE(y, \hat{y}) = -\sum_{i=1}^{5} y_i \log \hat{y}_i \quad (6) \]

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:

\[
\begin{align*}
    \frac{\partial J}{\partial U} & \quad \frac{\partial J}{\partial b_2} & \quad \frac{\partial J}{\partial W} & \quad \frac{\partial J}{\partial b_1} & \quad \frac{\partial J}{\partial L_i}
\end{align*}
\]
where
\[
U \in \mathbb{R}^{100 \times 5} \quad b_2 \in \mathbb{R}^{5} \quad W \in \mathbb{R}^{150 \times 100} \quad b_1 \in \mathbb{R}^{100} \quad L_i \in \mathbb{R}^{50}
\]

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.)*

(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 (L2 regularization):

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

and we optimize the combined loss function

\[
J_{\text{full}}(\theta) = J(\theta) + J_{\text{reg}}(\theta)
\]

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

(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 \(A\) of dimension \(m \times n\), select values \(A_{ij}\) uniformly from \([-\epsilon, \epsilon]\), where

\[
\epsilon = \frac{\sqrt{6}}{\sqrt{m+n}}
\]

Implement the initialization for use in `xavier_weight_init` in `q2_initialization.py` and use it for the weights \(W\) and \(U\).

(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:**

\(^1\) 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 \(i,j\).

\(^2\) This is also referred to as Glorot initialization and was initially described in `http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf`
• 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.

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 $x_1, \ldots, x_t$, a language model predicts the following word $x_{t+1}$ by modeling:

$$P(x_{t+1} = v_j | x_t, \ldots, x_1)$$

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 $t = 1, \ldots, n - 1$:

$$e^{(t)} = x^{(t)}L$$

$$h^{(t)} = \text{sigmoid} \left( h^{(t-1)}H + e^{(t)}I + b_1 \right)$$

$$\hat{y}^{(t)} = \text{softmax} \left( h^{(t)}U + b_2 \right)$$

$$\bar{P}(x_{t+1} = v_j | x_t, \ldots, x_1) = \hat{y}^{(t)}_j$$

where $h^{(0)} = h_0 \in \mathbb{R}^{D_h}$ is some initialization vector for the hidden layer and $x^{(t)}L$ is the product of $L$ with the one-hot row-vector $x^{(t)}$ representing index of the current word. The parameters are:

$$L \in \mathbb{R}^{|V| \times d}, \quad H \in \mathbb{R}^{D_h \times D_h}, \quad I \in \mathbb{R}^{d \times D_h}, \quad b_1 \in \mathbb{R}^{D_h}, \quad U \in \mathbb{R}^{D_h \times |V|}, \quad b_2 \in \mathbb{R}^{|V|}$$

where $L$ is the embedding matrix, $I$ the input word representation matrix, $H$ the hidden transformation matrix, and $U$ is the output word representation matrix. $b_1$ and $b_2$ are biases. $d$ is the embedding dimension, $|V|$ is the vocabulary size, and $D_h$ is the hidden layer dimension.

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

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

where $y^{(t)}$ 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 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:

$$PP^{(t)}(y^{(t)}, \hat{y}^{(t)}) = \frac{1}{\bar{P}(x_{t+1} = x_{t+1} | x_t, \ldots, x_1)} = \frac{1}{\sum_{j=1}^{|V|} y_j^{(t)} \cdot \hat{y}_j^{(t)}}$$ (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 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.

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

$$\frac{\partial J^{(t)}}{\partial U}, \frac{\partial J^{(t)}}{\partial b_2}, \frac{\partial J^{(t)}}{\partial L_{x^{(t)}}}, \frac{\partial J^{(t)}}{\partial I}_i^{(t)}, \frac{\partial J^{(t)}}{\partial H}_i^{(t)}, \frac{\partial J^{(t)}}{\partial b_1}_i^{(t)}$$

where $L_{x^{(t)}}$ is the column of $L$ corresponding to the current word $x^{(t)}$, and $\big|_{(t)}$ denotes the gradient for the appearance of that parameter at time $t$. (Equivalently, $h^{(t-1)}$ 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:

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

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

```
      \hat{y}^{(t)}
        \downarrow
\hat{h}^{(t-1)} \rightarrow h^{(t)} \rightarrow \ldots
        \downarrow
x^{(t)}
```

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

$$\frac{\partial J^{(t)}}{\partial L_{x^{(t-1)}}}, \frac{\partial J^{(t)}}{\partial H}_{(t-1)}, \frac{\partial J^{(t)}}{\partial I}_{(t-1)}, \frac{\partial J^{(t)}}{\partial b_1}_{(t-1)}$$

---

4We use the tensorflow function `sequence_loss` to do this.
where $\delta^{(t-1)}$ 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

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

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 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.*

(d) (3 points) Given $h^{(t-1)}$, 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 $\tau$ 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?

(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 $x_{t+1}$ from the distribution $\hat{y}^{(t)}$ 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/](http://kingjamesprogramming.tumblr.com/) for a great one 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!