Assignment 2, Problem 1 (Tensorflow Softmax)

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

$$J_{\texttt{softmax-CE}}(\mathbf{W}) = CE(\boldsymbol{y}, \texttt{softmax}(\boldsymbol{x}\mathbf{W})) \tag{1}$$

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

Solution to Assignment 2, Problem 1 (a)

James Le

Implement the softmax function using `tensorflow` in `q1_softmax.py`. Remember that,

$$\texttt{softmax}(\boldsymbol{x})_i = \frac{e^{\boldsymbol{x}_i}}{\sum_j e^{\boldsymbol{x}_j}} \tag{2}$$

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

Implementation in `numpy`:

```
import numpy as np

def softmax(x):
    c = np.max(x, axis=x.ndim - 1, keepdims=True)
    #for numerical stability
    y = np.sum(np.exp(x - c), axis=x.ndim - 1, keepdims=True)
    x = np.exp(x - c)/y
    return x
```

The implementation in `tensorflow`:

```
import numpy as np
import tensorflow as tf

def softmax(x):
  """
```

```
  Compute the softmax function in tensorflow.

  You might find the tensorflow functions tf.exp,
  tf.reduce_max, tf.reduce_sum, tf.expand_dims useful. (Many
  solutions are possible, so you may not need to use all of
  these functions). Recall also that many common tensorflow
  operations are sugared (e.g. x * y does a tensor
  multiplication if x and y are both tensors). Make sure to
  implement the numerical stability fixes as in the previous
  homework!

  Args:
    x:   tf.Tensor with shape (n_samples, n_features). Note
         feature vectors are represented by row-vectors. (For
         simplicity, no need to handle 1-d input as in the
         previous homework)
  Returns:
    out: tf.Tensor with shape (n_sample, n_features). You
         need to construct this tensor in this problem.
  """

  ### YOUR CODE HERE
  log_c = tf.reduce_max(x,
                        reduction_indices=[len(x.get_shape()) - 1],
                        keep_dims=True)
  y     = tf.reduce_sum(tf.exp(x - log_c),
                        reduction_indices=[len(x.get_shape()) - 1],
                        keep_dims=True)
  out   = tf.exp(x - log_c) / y
  ### END YOUR CODE

  return out
```

Solution to Assignment 2, Problem 1 (b)

James Le

Implement the cross-entropy loss using `tensorflow` in `q1_softmax.py`. Remember that:

$$CE(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_{i=1}^{\mathrm{N}_c} y_i \log(\hat{y}_i) \tag{3}$$

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`.

```
import numpy as np
```

Solution to Assignment 2, Problem 1 (c)

James Le

Carefully stude the `Model` class in `model.py`. Briefly explain the purpose of placeholder variables and feed dictionaries in `tensorflow` computations. Fill in the implementation for the `add_placeholders`, `create_feed_dict` in `q1_classifier.py`.

HINT: that configuration variables are stored in `Config` class. You will need to use these configuartion variables in the code.

```
import numpy as np
```

Solution to Assignment 2, Problem 1 (d)

James Le

Implement the transformatoin 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 implementation form earlier pars of the problem, **NOT** `tensorflow` built-ins.

```
import numpy as np
```

Solution to Assignment 2, Problem 1 (e)

James Le

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 make sure that the tests pass.

HINT: Make sure to use the learning rate specified in `Config`.

```
import numpy as np
```

Assignment 2, Problem 2 (Deep Networks for Named Entity Recognition (NER))

In this section, we'll get to practice backpropagation and training deep networks to attack the task of **Named Entity Recognition** (**NER**): predicting whether a given word, in context, represets 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 (**NOTE**: 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 neighbours:

$$\boldsymbol{x}^{(t)} = [\boldsymbol{x}_{t-1}\mathbf{L}, \boldsymbol{x}_t\mathbf{L}, \boldsymbol{x}_{t+1}\mathbf{L}] \in \mathbb{R}^{3\mathrm{d}} \tag{4}$$

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

$$1 = \tag{5}$$

$$2 = \tag{6}$$

And evaluate by cross-entropy loss,

$$J(\theta) = CE(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_{i=1}^{\mathrm{d}} y_i \log(\hat{y}_i) \tag{7}$$

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

Solution to Assignment 2, Problem 2 (a)

James Le

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

$$\frac{\partial J}{\partial \boldsymbol{U}} \quad \frac{\partial J}{\partial \boldsymbol{b}_2} \quad \frac{\partial J}{\partial \boldsymbol{W}} \quad \frac{\partial J}{\partial \boldsymbol{b}_1} \quad \frac{\partial J}{\partial \boldsymbol{L}_i} \tag{8}$$

where,

$$\boldsymbol{U} \in \mathbb{R}^{100\times 5} \quad \boldsymbol{b}_2 \in \mathbb{R}^5 \quad \boldsymbol{W} \in \mathbb{R}^{150\times 100} \quad \boldsymbol{b}_1 \in \mathbb{R}^{100} \quad \boldsymbol{L}_i \in \mathbb{R}^{50} \tag{9}$$

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:

$$\tanh(z) = 2\sigma(2z) - 1 \tag{10}$$

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 to Assignment 2, Problem 2 (b)

James Le

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):

$$J_{\text{reg}}(\theta) = \frac{\lambda}{2}\big[\sum_{i,j} \boldsymbol{W}_{ij}^2 + \sum_{i',j'} \boldsymbol{U}_{i'j'}^2\big] \tag{11}$$

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

---

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

Solution to Assignment 2, Problem 2 (c)

James Le

In order to avoid neurons becoming too correlated and ending up in poor local minima, 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 dimensions m $\times$ n, select values $\boldsymbol{A}_{ij}$ uniformly from $[-\epsilon, -\epsilon]$, where

$$\epsilon = \frac{\sqrt{6}}{\sqrt{m+n}} \tag{12}$$

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

---

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

Solution to Assignment 2, Problem 2 (d)

James Le

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 turn off debug settings when doing final evaluation). **Note**: the test set has only dummy labels; we'll compare your predictions against ground truth after you submit.

Deliverables:

- Working implementation of the **NER** window model in `q2_NER.py`.
- In your write-up, *briefly* state the optimal hyperparameters you found for your model: **regularization**, **dimensions**, **learning rate** (including time-varying, such as **annealing**), **SGC batch size**, etc. Report the performance of your model on the **validiation** set. You should be able to get a 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**: The code should run within 15 minutes on a GPU and 1 hour on a CPU.

Assignment 2, Problem 3 (Recurrent Neural Networks: Language Modeling)

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. Give $\boldsymbol{x}_1, ..., \boldsymbol{x}_t$, a language model prdicts the following word $\boldsymbol{x}_{t+1}$ by modeling:

$$P(\boldsymbol{x}_{t+1} = v_j | \boldsymbol{x}_1, ..., \boldsymbol{x}_t) \tag{13}$$

where v_j is a word in the vocabulary.

Your job is to implement a recurrent neural network language model, which use feedback informatoin in the hidden layer to model the 'history' $x_t, x_{t-1}, ..., x_1$. Formally, the model[3] is, for $t \in \{1, ..., \mathrm{n}-1\}$:

$$\boldsymbol{e}^{(t)} = \boldsymbol{x}^{(t)}\boldsymbol{L} \tag{14}$$

$$\boldsymbol{h}^{(t)} = \sigma(\boldsymbol{h}^{(t-1)}\boldsymbol{H} + e^{(t)}\boldsymbol{I} + \boldsymbol{b}_1) \tag{15}$$

$$\hat{\boldsymbol{y}}^{(t)} = \mathrm{softmax}(\boldsymbol{h}^{(t)}\boldsymbol{U} + \boldsymbol{b}_2) \tag{16}$$

$$\bar{P}(\boldsymbol{x}^{(t)} = \boldsymbol{v}_j | \boldsymbol{x}_{(t)}, ..., \boldsymbol{x}_{(1)}) = \hat{\boldsymbol{y}}_j^{(t)} \tag{17}$$

$$\tag{18}$$

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:

$$\boldsymbol{L} \in \mathbb{R}^{|V| \times d} \tag{19}$$

where $\boldsymbol{L}$ is the embedding matrix, $\boldsymbol{I}$ 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 $\hat{\boldsymbol{y}}^{(t)} \in \mathbb{R}^{|\mathrm{V}|}$ is a probability distribution over the vocabulary, and we optimize the (unregularized) cross-entropy loss:

$$J(\theta) = CE(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = -\sum_{t=1}^{|\mathrm{V}|} \boldsymbol{y}_i{}^{(t)} \log(\hat{\boldsymbol{y}}_i^{(t)} \tag{20}$$

[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`

where $\boldsymbol{y}^{(t)}$ is the one-hot vector corresponding to the target word (which here is equal to x_{t+1}). As in question 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.

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

Solution to Assignment 2, Problem 3 (a)

James Le

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

$$PP \tag{21}$$

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 (arithemtic) mean cross-entropy loss will also minimize the (geometric) mean perplexity across the training set. [5]

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 $|\mathrm{V}| = 2000$ and $|\mathrm{V}| = 10,000$ and keep this in mind as a baseline.

---

[5] **NOTE**: this should be a very short problem - not too perplexing!

Solution to Assignment 2, Problem 3 (b)

James Le

As you did in question 2, compute the gradients with respect to all the model parameters at a single point in time t:

$$\frac{\partial J^(t)}{\partial \boldsymbol{U}} \tag{22}$$

where $\boldsymbol{L}_{\boldsymbol{x}^{(t)}}$ is the column of $\boldsymbol{L}$ corresponding to the current word $\boldsymbol{x}^{(t)}$, and $\big|_{(t)}$ 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 this in 3 (c).

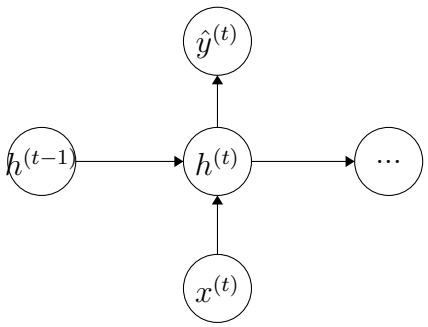
Additionally compute the derivative with respoect to the **previous** hidden layer value:

$$\frac{\partial J^(t)}{\partial \boldsymbol{h}^{(t-1)}} \tag{23}$$

Solution to Assignment 2, Problem 3 (c)

James Le

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:

$$\frac{\partial J^(t)}{\partial \boldsymbol{L}_{\boldsymbol{x}^{(t-1)}}} \tag{24}$$

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[6] to express these derivatives in terms of error term

$$\delta^{(t-1)} = \frac{\partial}{\partial} \tag{25}$$

computed in the previous part. Doing so will allow for re-use of the 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 of $\tau \approx 3 - 5$ timesteps.

---

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

Solution to Assignment 2, Problem 3 (d)

James Le

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 $|\mathrm{V}|$ (Equation 19). What is the slow step?

Solution to Assignment 2, Problem 3 (e)

James Le

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 question 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 (train schedule, number of iterations, learning rate, backprop timesteps), and you perplexity score on the `ptb-dev` set. You should be able to get validiation 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 hours to run.

Solution to Assignment 2, Problem 3 (f)

James Le

The networks that you've seen in Assignment 1 and in question 2 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 $x_1, ..., x_n$. 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 $\hat{\boldsymbol{y}}_{t+1}$ 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 write up. See if you can generate something humorour!

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 entertaining. See `http://kingjamesprogramming.tumblr.com/` for a great one[7] trained on a mix of the King James Bible and the Structure and Interpretation of Computer Programs.

---

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