Homework 3: End-to-end Speech Recognition
Due: 10th May 2017 at 11:59 PM PST

CS224S/LINGUIST285

Prof. Andrew Maas

Please read this entire homework handout before beginning. We advise you to *start early* and to make use of the TAs by coming to office hours and asking questions! For collaboration and the late day policy, please refer to the course website.

Starter Code To obtain the starter code, run

```
scp -r <SUNet ID>@corn.stanford.edu:/afs/.ir/class/cs224s/hw/hw3 .
// or if you're on corn: cp -r /afs/.ir/class/cs224s/hw/hw3 .
```

1 Connectionist Temporal Classification

In the first part of the homework, we will implement the Connectionist Temporal Classification (CTC) loss function, building on the material covered in the lecture on end-to-end neural network speech recognizers.

Define an utterance as being T timesteps long and having f speech features i.e. the feature vector at each time step comes from $\mathbb{R}^f$, for a $T \times f$-sized input feature matrix. Also let the output space contain L symbols and the true output labeling for the utterance to be a vector $o = (o_1, o_2, \ldots o_U)$ of length U, where each element of the vector is one of $|L|$ output symbols i.e. $o \in L^U$, with $U \leq T$.

Now, recall from the lecture that to apply CTC loss to this utterance, we train a sequence-to-sequence model whose output is a real number matrix of shape $T \times (|L| + 1)$, where the $(i, j)^{th}$ entry of this matrix equals the activation for predicting the j^{th} output symbol at timestep i. The additional 1 in $|L|+1$ is to allow the model to predict the blank symbol at that time step, needed for CTC to work, as discussed in the lecture.

In this part of the assignment, we will start with this $T \times (|L| + 1)$ activations matrix, and compute the negative log of the probability of obtaining our true output labeling given these activations.

Note that the inputs to the forward and backward computations are valid probability distributions, but Tensorflow's implementation of CTC loss takes in unnormalized activations and does the softmax step itself. Thus, you will have to include a softmax step at the start of your code.

1. Implement the following functions

   - `compute_forward_variables`, which computes the $\alpha_t(s)$ values
   - `compute_backward_variables`, which computes the $\beta_t(s)$ values
   - `compute_ctc_loss` which returns the negative log of the probability $p(l|x')$, where l and x' are your target labeling and **normalized** input activations respectively
   - `compute_gradients` which returns the gradient of the CTC loss w.r.t. the **unnormalized** activations x given as input to your CTC implementation

   See the starter code, all of which is in `q1.py`, for further details.

2. Run your code on our sample inputs with

   ```
   python q1.py
   ```

1.1 Deliverables

1. Include all of your code.
2. Include the computed CTC loss and gradient matrix on the inputs `input1` in the code.
3. What is the time complexity (big-O) of forward-backward algorithm?
4. Compute the CTC loss on the inputs `input2` in the code and explain the reason behind your observation.

2 End-To-End CTC in Tensorflow

2.1 Tensorflow Installation

For this assignment, you will need at least Tensorflow version 0.10 (for CTC). If you've already satisfied this requirement, you can skip this section. Else, we recommend installing this version of Tensorflow in a Python virtualenv.

1. Install virtualenv, if you haven't already.

   ```
   pip install virtualenv
   ```

2. Create a new virtualenv.

   ```
   virtualenv -p /usr/bin/python2.7 cs224s_venv
   ```

3. Start your python virtual environment.

   ```
   source cs224s_venv/bin/activate
   // Or, if you're on corn: source cs224s_venv/bin/activate.csh
   ```

   Verify that:

   ```
   which python
   ```

   gives you the path to your fresh new virtualenv.

4. Within the virtual environment, install all required dependencies, including the correct Tensorflow version, by running the following.

   ```
   // if you're using corn, first run "pip install --upgrade pip"
   pip install -r requirements.txt
   ```

5. That's it! If you have any issues setting up, try StackOverflow, or please come to office hours.

2.2 Starter Code and Dataset

We will be using a subset of the TIDIGITS dataset for this exercise. The file containing the features and target sequences is located on AFS. We've preprocessed this data for you, and it should already be in the data folder bundled with the starter code.

2.3 Model Overview

Our main task for this part of the homework is to build a digit recognizer, similar to the system built in homework 2 - except this one will be trained end-to-end. At a high level, the model works as follows:

1. For each example, say with T timesteps and $f = 13$ (13 MFCC features per frame) for each timestep, our feature vector is of size $T \times f$. We will be doing minibatch training. Thus, for a batch size of b, our final input tensor is of size $(b \times T \times f)$. Note that we need to pad all sequences to the same number of time steps, since this tensor needs to be dense, but do not worry - the starter code does this for you.

2. This dense $(b \times T \times f)$-sized tensor becomes our input to a single-depth recurrent neural network with Gated Recurrent Units (GRUs) for the individual cells. Assume the GRUs have a hidden layer size of h. Then, the GRU-RNN maintains an h-sized hidden state at each time step, which for T time steps and b examples in the batch becomes a $(b \times T \times h)$-sized output tensor.

3. We apply a single affine transformation $(y = xW + b)$ to this output tensor to convert it to activations for our labels. For $L = 11$ labels (0-9 plus the 'oh' utterances), the activations tensor must be of size $b \times T \times L+1$ (the extra 1 accounts for the blank label needed for CTC to work). You'll need to initialize the W and b variables accordingly.

   Note here that these activations are not normalized/do not constitute a probability distribution; the Tensorflow CTC implementation itself computes the softmax.

4. Finally, from these unnormalized activations, we compute the CTC loss, which is largely what we will seek to optimize during training. We will be using Tensorflow's CTC implementation for the purpose.

The guidelines here are brief, and we expect you to rely on the starter code comments to get a better understanding of the requirements.

2.4 Construct Computational Graph

Behind the scenes, Tensorflow builds a computation graph consisting of several types of nodes: placeholders (specifying inputs to the model), variables (for weights and bias terms), constants (exactly what they sound like), and functions (result of performing an operation, like "muliply" or "sigmoid"). We'll specify those nodes now.

The advantage with explicitly stating the inputs, weights, operations performed, and other components of the model is that we can let Tensorflow worry about computing the gradients and updating all of our weights, in a process known as automatic differentiation, and we get free graph visualization tools such as Tensorboard.

1. Implement `add_placeholders()`, to provide the inputs to our end-to-end model.

2. Implement `create_feed_dict()`, which actually creates the dictionary of inputs that is provided to the model at compute time.

3. Implement `add_prediction_op()` - in this function you will write code to create the GRU-RNN and the affine transformation.

4. Implement `add_loss_op()` - from the (unnormalized) activations that will be obtained after a Tensorflow session run (we'll explain) on an input minibatch, we will write code to compute the loss function. We will use L2 regularization loss in addition to CTC loss.

5. Implement `add_training_op()` - here, we create the Tensorflow optimizer object that will update our trainable variable weights in order to decrease the loss as defined in `add_loss_op()`.

6. Implement `add_decoder_and_wer_op()` - Obtaining the model's output sequences on different inputs also involve a decoding step, which decodes the unnormalized output of the RNN to an output sequence. For this purpose, we will be using (one of) Tensorflow's decoder implementations.

   Additionally, to gauge how well our model is doing, you will also compute in this function the word error rate of our model's prediction (obtained using the decoder) with the ground truth sequence, and average it across all examples.

7. Now that we've built the comptuational graph, we're ready to run our model. We've given you this code for free. Run the following command to see your model in action (this may take up to half an hour to run):

   ```
   // Make sure you've copied / sym-linked your data folder from HW2
   python q2.py
   ```

8. Open Tensorboard to visualize performance of the model.

   ```
   tensorboard --logdir=tensorboard/
   ```

2.5 Deliverables

1. Include all your code - this should all be in `q2.py`.
2. Report your final train edit distance, and include a plot of edit distance per training epoch (from Tensorboard). How does this edit distance compare to your training set performance in homework 2? Assume the edit distance is very similar to WER for your newly-trained model.
3. Include plots of your training cost as a function of epochs using Tensorboard. In the config, reduce your learning rate to 1e-4 and plot the training cost again. How do the two runs of the model compare?
4. Include a screenshot of the compute graph from Tensorboard.
5. Modern state-of-the-art ASR systems now use neural networks, attached with CTC loss. What are some advantage of these systems over traditional Hidden Markov Model (HMM) based models? What are some disadvantages?