Reading 2

Read and write a summary of Amodei et al. (2016)'s Deep Speech 2. Note that although this reading is not due until the last day of class, you may want to do this earlier so that your weekend is free for studying. Our final is two days after the end of class.

Lab 2

TIMIT phoneme recognition with recurrent neural nets

In this lab, you will learn how to use recurrent neural networks in a phoneme recognizer. Experiments will be conducted on the TIMIT corpus (Garofolo et al., 1993). TIMIT is a speech corpus developed by Texas Instruments and the Massachusetts Institute of Technology designed to have balanced coverage of phonemes by a wide variety of speakers across the United States. **This corpus is licensed for educational use at San Diego State University and may not be redistributed**. You may access TIMIT from Blackboard. The doc directory contains files describing the corpus. For a history of reasonably recent performance benchmarks on TIMIT see (Lopes and Perdigão1, 2011).

You should not expect stellar performance in this lab. There are several confounding factors that make very good performance difficult. The first is the effect of coarticulation. Secondly, unless you want to write additional code, you will not be checking for correct labels in ambiguous frames. For example, if the transition between /ae/ and /t/ (as in the word bat) occurs in the middle of a frame, either label would really be correct.

This corpus is much larger than the corpus that you used in previous assignments and it is recommended that you develop your system with a reduced corpus; the first 100 sentences with 2-fold cross validation is enough to let you know that you are on track, and you should start with something even smaller. We will go through some basics on keras implementation details in class, but some of these are summarized here.

A Corpus class is provided for you that will make much of this assignment simpler. It provides methods for finding audio and label files, extracting features when given a feature extractor object, as well as converting labels from start and end samples to seconds or frame indices.

When training recurrent nets, it is possible to have variable length sequences, but for training and prediction, they must all be padded to a common size as the network execution graph expects data to be within a single tensor with the shape: $examples \times time \times features$.

This is accomplished by adding *time* columns to the feature matrix filled with a special value such as a zero. A Masking layer is inserted into the first layer of the network that detects frames filled with the special value and then ignores all time slices that are masked when computing loss, back-propagating, or computing error.

As an example, suppose we had the following code where `dim` is the number of feature dimensions:

```
model = Sequential()
model.add(Masking(mask_value = 0., input_shape=[None, dim])
model.add(LSTM, 150, return_sequences=True,
          kernel_regularizer=regularizers.L2(.01),
          recurrent_regularizer=regularizers.L2(.01))
more additions to the model…
```

This network would expect 3D tensors with an arbitrary number of examples, all zero-padded to the length of the longest sequence. If we had 10000 examples with up to 1200 time steps and 15 dimensional feature vectors, the tensor presented to this network would be $10000 \times 1200 \times 15$.

Suppose however that the 1200 time step example was an outlier, most of the sequences were of about length 100. Padding to the longest sequence in the entire training set will result in excessive memory use, a problem for large data sets. A strategy for dealing with this is to create minibatches. The network can handle tensors of varying time steps, but all examples within a given tensor must be the same length. Thus, we can pad our minibatches to the longest sequence in the batch. One of the mini batches will have examples all of length 1200, but most of them will only be padded to about a 100 examples. Batch generators are classes that can be passed to the model fitting function and generate minibatches. These can be used for things other than recurrent networks, such as iterating over very large data sets even when padding is not required.

A batch generator is a class that be iterated over. In addition to their constructor, they must support __next__() which will return the next minibatch of examples and labels. It is frequently helpful to include a method to return the number of minibatches per epoch, e.g. get_batches_per_epoch().

Suppose we wrote a batch generator that took a TIMIT corpus object, a list of utterances, and a batch size. We would create and use it as follows:

```
# Assume corpus is a Timit corpus, training_utterances is a list of
# utterances to be trained, and batch_size is the number of examples
# to be used.
gen = PaddedBatchGenerator(corpus, training_utterances, batch_size)
model.fit_generator(gen, steps_per_epoch=gen.get_batches_per_epoch(),
                    epochs=20, callbacks=…, validation_data=…)
```

Note the use of fit_generator instead of fit. We could also provide a batch generator for the validation_data, but as of this writing you cannot have TensorBoard write out gradients if use a validation_data generator. The last batch of an epoch is frequently smaller unless the size of the minibatches divides evenly into the number of training examples. Batch generators never raise

StopIteration. Instead, the model fitting routine is given a number of steps that can be calculated by taking the ceiling of the number of times the minibatch size divides into the number of training examples. If you wish to write out gradients, you can still use a padded batch generator to produce the test or evaluation set. Simply create a batch generator were the batch size is the same as the number of examples and you can produce a padded set of features and labels that can be passed as validation_data without the use of an iterator.

Several classes and functions are provided for your convenience in a zip file on Blackboard. Highlights:

- build_model – Creates a model from a list specification. This is very similar to the function you used previously, but it now supports tuples of length 3 or 4. Tuples of length 4 are for wrapper layers. When you have a recurrent network, you may wish to include some layers that do not have recurrences. These do not expect tensors without a time element but are in a network where time elements are expected. A TimeDistributed wrapper around the layer will resolve this e.g. TimeDistributed(Dense(2, activation="softmax"). In the network specification data structure that we have been using, this would be represented as: (Dense, [2], {'activation':'softmax'}, (TimeDistributed, [], {})).
- plot_confusion(predictions, truth, labels, masks=None) - Plots a confusion matrix. Predictions consist of a model output tensor ($examples \times time \times P(class)$) or a tensor where the class has already been decided and a 2D matrix ($examples \times time$) containing class indices, the phoneme index occurring in example *e* at time t. Truth is a ground-truth tensor that is similarly organized with the last dimension of the 3D tensor being a one-hot vector and the 2D tensor containing a class index. A labels vector provides a list of class names in the same order as the indices: class *i* has name labels[i]. An optional mask tensor is of type boolean ($examples \times time$) and is True anywhere that the utterance is valid (i.e. It is False for each frame that was added to an example for zero-padding). You may see warning produced by this function as we did not build in a check for cases where there are no examples of a specific class. There is also a ConfusionTensorBoard class that can produce a confusion matrix that can be viewed from TensorBoard's image viewer. It can be set up to show the performance of every minibatch (will slow things down considerably) or to show the last minibatch of an epoch. See the documentation for details as well as for how to install the required tfplot library
- The standard streamer modules that you have used before are present. Differences in the versions for this class: DFTStream now accepts arguments to produce Mel spectra. It also accounts for a well known problem that we have not had to deal with until we started using the TIMIT corpus. Occasionally, a frame will have all zeros or the energy will be so low as to result in a log transform to -Inf. This has disastrous results on most training methods as non-zero weights result in an infinite value being propagated through the network with disastrous consequences on the loss. See DFTStream for

details on how this is addressed. In general, it is a good idea to audit your data and make sure that you do not have NaNs or Infs. This has already been done for you, see skeleton code that is turned off in driver.py. It is also a good idea to check for all zeros which has not been done and is unlikely to be a problem in this dataset. Data cleaning is an important part of ensuring that your machine learner will learn the target distribution.

- Class Corpus is a class for manipulating speech corpora. It allows you to refer to both audio data and labels by relative paths specifying train (development data) or test (evaluation data), dialect region, the speaker, and the utterance indicator. An example is: train/dr8/mtcs0/sx82. There are methods for retrieving the audio filename, phoneme transcriptions with starts and stops of each phoneme specified in samples, time, or frame indices, and for retrieving features. Feature retrieval requires method set_feautre_extractor to be invoked first, providing an instance of the Features class. To retrieve lists of utterances labels associated with the development or evaluation sets, use get_utterances() with arguments of "train" or "test".
- The Features class is initialized with a frame rate specification and the path to the root of the audio directory. If you have already initialized your Corpus object, you can invoke get_audio_dir() on the corpus to obtain the audio directory.

You will need to construct the following:

driver.py – Your experiment driver. Your model lists should be specified here. Suggestions for your experiments:

- Use of BatchNormalization layers (these have no arguments). For details, see 8.7.1 of Goodfellow et al. (2016) and the keras documentaion.
- Dropout
- L_P regularization : Note that for the recurrent layers, there is a recurrent_regularizer in addition to the kernel_regularizer. Avoid setting these too large as large penalties can sometimes result in problems during training for these networks. You might want to start with .001.
- The skeleton code shows an example of one recurrent network. The anonymous function has additional parameters when compared to the previous lab that allow you to vary the instantiation of the network. The network supports specifying the input feature dimensionality, the width of the layers, the dropout rate, and an L_2 regularization parameter. Example: models_rnn[0](input_dim, 45, .5, 0.001)
- Experiments on the full data set are long. Training one epoch of the development data takes a little unter 30 m on a Core I7-3820 with an NVIDIA 1080-Ti. There are a few suggestions for managing things:
    - Don't use a large number of folds. This increases the size of training data (usually a good thing, but not for a student project)

- You should implement grid and/or random search on a subset of the data. Avoid taking the first N elements as they are sorted by dialect region and speaker. If you use a random search, your network may not have enough hidden nodes for the log operator to be a reasonable heuristic for how many iterations to use. As an example, two layers of 50 hidden nodes each: $\log(100) \approx 4.6$. Once you have a feel for things that work and things that do not, run *at least one model specification* as a 3-fold or greater experiment on the development data. See how well it generalizes on the evaluation data (TIMIT's test directory). Abstract this into a function that takes a model and corpus. Depending on your computer and architecture, this may take a day or two.

myclassifier/crossvalider.py – A cross validator class geared at processing utterances from a corpus.

myclassifier/recurrent.py – A train_and_evaluate method for testing a specific fold.

- Use the Adam optimizer and categorical cross entropy as your loss function.
- Create a log directory and use TensorBoard to track your progress. Write the graph and gradients.
- Save your trained model to allow you to run it on the evaluation data (you could add this to the CrossValidator if you prefer).
- Create a ConfusionTensorBoard and add it as a callback to see the progress of the last minibatch on each epoch.
- There is currently what I believe to be an error in keras. Once a model is trained, attempting to classify data will result in an error indicating a bad tensor shape for BatchNorm layers. This can be resolved as follows:
  1. Save the model to disk
  2. Use keras's backend mechanism to tell keras that layers that are treated differently in training and testing should be executed in test mode:
     ```
     # Assumes import keras.backend as K
     K.set_learning_phase(False)
     ```
  3. Reload the model and use it.
- Create a confusion matrix for the test data of this fold and save it. Note that once you have the confusion matrix it is easy to compute the error rate:
  $1 - \frac{\sum diag(confusion)}{\sum\sum(confusion)}$

- myclassifier/batchgenerator – PaddedBatchGenerator – a batch generator that pads to the longest sequence.

As it is near the end of the semester, we will make the reporting for this lab simpler than the previous one. You do not need to write and introduction or methods section. Focus on the results and discussion section, making sure that you include enough detail to understand your

experimental plan, what it produced, and your observations/hypotheses about the experimental results. Consequently, code will count for 75% in this lab and the report will be 25%.

Literature Cited

Amodei, D., Ananthanarayanan, S., Anubhai, R., Bai, J., Battenberg, E., Case, C., Casper, J., Catanzaro, B., Cheng, Q., Chen, G. et al. (2016). Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin. In *Proceedings of The 33rd International Conference on Machine Learning*, vol. 48 eds. B. Maria Florina and Q. W. Kilian), pp. 173--182. Proceedings of Machine Learning Research: PMLR.

Garofolo, J. S., Lamel, L. F., Fisher, W. M., Fiscus, J. G., Pallett, D. S., Dahlgren, N. L. and Zue, V. (1993). Timit acoustic-phonetic continuous speech corpus, pp. 8. Philadelphia, PA: Linguistic Data Consortium, Univ. of Pennsylvania.

Goodfellow, I., Bengio, Y. and Courville, A. (2016). Deep learning. Cambridge, Massachusetts: The MIT Press.

Lopes, C. and Perdigão1, F. (2011). Phone Recognition on the TIMIT Database. In *Speech Technologies*, (ed. I. Ipsic), pp. 20. London: IntechOpen Limited.