A5

Part I – Questions worth 20 points each

1. Explain why the receptive field in a convolutional network grows as convolutional layers are stacked.
2. Would you best describe convolutional filters as a local or global analysis tool?
3. You have just been hired by *Contestador* Inc. to develop speech systems for an automated call center. You have been asked to develop an emotion recognition system that identifies whether or not people are getting frustrated with the automated agent that handles most of their calls. As such, you plan to develop an emotion classification system based on the client's speech that will route them to a real person if they appear unhappy, with a check at the end of each round of communication between the client and agent. You decide to use a recurrent neural network (RNN). Based on the problem, would you select a many-to-many or a many-to-one classifier? Justify your answer.
4. Justify why we use input and forget gates in LSTM networks.

Part II – Recurrent networks (100 points for code, 20 points for figures)

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 **Canvas on the assignment code tab**. The README.doc provides a high-level overview of the corpus with additional resources in the doc directory. TIMIT is small by today's standards (~ 5.4 hours of speech), but remains one of the best phonetically annotated corpora and is still used in research although scientists frequently use additional sources of training data.

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, and only a single phoneme label is typically assigned to a frame.

State of the art phoneme error rates (PERs) for this corpus are typically in the teens (e.g. 13-20%, see the PER section of https://github.com/syhw/wer_are_we for TIMIT) and frequently are achieved by using additional training data. The word error rates are much lower as sequences of phone hypotheses can be leveraged to correct some mislabeled phonemes. We will talk more about this when we study language models. You should aim to achieve a minimum of a 50% phone error rate, but you can certainly do much better, and error rates of under 40% are not too difficult to achieve.

This corpus is much larger than the corpus that you used in previous assignments and you do not need to perform cross validation. Data have been partitioned into train and test directories. It is highly recommended that you work to create a system that can complete all steps without failure prior to starting large scale experiments. When I develop these types of systems, I will frequently reduce the amount of training data and set the number of epochs to a very small number.

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 $10000 \times 1200 \times 15$.

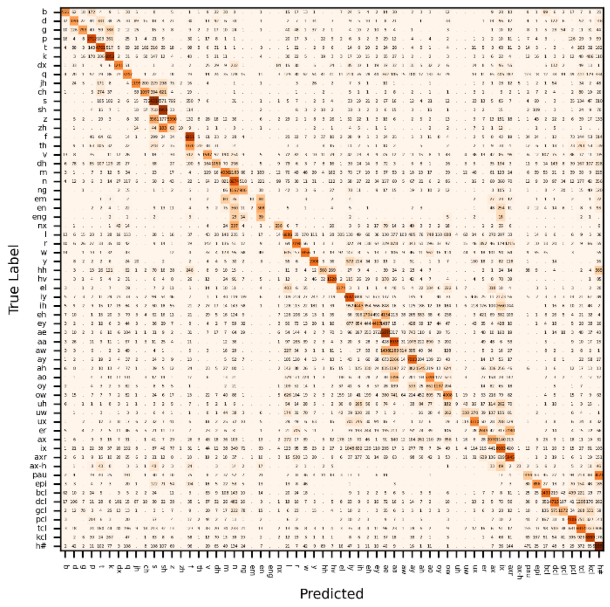
Suppose however that the 1200 time step example was an outlier and most of the sequences were of about length 100. Padding to the longest sequence in the entire training set will result in excessive memory use; a problem for large data sets. A strategy for dealing with this is to create minibatches. 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.

Note that when you classify novel data using a trained model and the predict method, you can have problems if you are using a batch generator. Your batch generator will be generating input tensors of different sizes (longest sequence in each batch). Predict will gleefully predict each batch and blindly attempt to concatenate the batch tensors into a single tensor. As the tensors are likely to be different sizes, your process will almost certainly throw an exception. There are two solutions to this:

1. Use predict_batch instead of predict. You will need to explicitly create a loop over your batches and call predict_batch each time.
2. Sacrifice memory and initialize your batch generator on the test data such that it generates a single batch. All samples will be of the same duration.

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; that is 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. The function produces plots like the one to the right (which has been scaled down to the point that the counts cannot be read, do not do this in a report).



- 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_feature_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". If you would like your features and labels to be cached, set the optional cache argument to True.
- 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. Once a feature object is initialized, the get_features method can be called with filenames that are relative to the root of the audio directory. For example, if you have decompressed the TIMIT corpus to D:\timit, providing an argument of "train\dr4\fcag0\sx63.WAV" would return feature data where each row of the tensor represents the Mel spectra of one frame.

You will need to construct the following:

driver.py – Your experiment driver. Your model lists should be specified here. You will not have time to look at all parameters extensively. The main things for this set of experiments are that you examine RNNs. You should run experiments on a minimum of 3 different networks, and it is suggested that you use the network in the code skeleton as a starting point. Things you could do:

- The use of BatchNormalization layers is recommended.
- Dropout.
- Use gated recurrent units (GRUs) instead of LSTM. GRUs are simpler, see the difference in your book. If you are using a GPU accelerator, NVIDIA's cuDNN library (which you may need to install, installing tensorflow-gpu does not necessarily install it) offers significant speedup for GRU and LSTM.
- Conv1D layer
- 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. Example initialization: models_rnn[0](input_dim, 45, .5, 0.001)
- Experiments on the full data set are long. Time per epoch depends greatly on your network architecture, but as an example a network that achieves reasonable

performance can be trained in about 5-6 minutes per epoch on a Core I7-9700K with an NVIDIA 2080-Ti. Note that if you are using a GPU and have the compute unified device architecture deep neural network library (CUDNN) installed, LSTM and GRU layers will use an optimized CUDA which significantly speeds training. These library routines are fairly specialized and will not be used if you change default parameters such as the activation or gating functions, or if you use dropout in your network.

- For many architectures, you should see performance stabilize in about 15-25 epochs.
- Use smaller batches. Remember that each example has many phonemes in it and these examples will be longer than what you have previously used. A batch size of 20 is not unreasonable.

myclassifier/recurrent.py – A train_and_evaluate method for testing a specific split of the training data (you only need one split, but write this function so that you could do cross validation if you had the time).

- Use the Adam optimizer and categorical cross entropy as your loss function.
- Create a log directory and use TensorBoard to track your progress.
- 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.py – PaddedBatchGenerator – a batch generator that pads to the longest sequence.

General hints:

- Spend time getting your PaddedBatchGenerator correct. Your model will perform poorly if you are not aligning phoneme labels with the proper spectrogram frames or padding incorrectly. It is worth writing some testing code to go through the entire data set to make sure that no batch fails due to edge cases.
- When you start your development, limit yourself to a restricted dataset. You will not generate good models, but you can quickly turn around errors. Do not train on larger data sets until you can train a bad model first.

What to turn in:

- Submit Python files including affidavit
- Submit a Word or PDF file with a table showing your architectures and performance. Provide a confusion matrix for each of the three architectures.

Literature Cited

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.