vui_notebook

January 14, 2018

1 Artificial Intelligence Nanodegree

1.1 Voice User Interfaces

1.2 Project: Speech Recognition with Neural Networks

In this notebook, some template code has already been provided for you, and you will need to implement additional functionality to successfully complete this project. You will not need to modify the included code beyond what is requested. Sections that begin with **'(IMPLEMENTATION)'** in the header indicate that the following blocks of code will require additional functionality which you must provide. Please be sure to read the instructions carefully!

> **Note**: Once you have completed all of the code implementations, you need to finalize your work by exporting the Jupyter Notebook as an HTML document. Before exporting the notebook to html, all of the code cells need to have been run so that reviewers can see the final implementation and output. You can then export the notebook by using the menu above and navigating to",

File -> Download as -> HTML (.html). Include the finished document along with this notebook as your submission.

In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes that begin with **'Answer:'**. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide.

> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this Jupyter notebook.

1.3 Introduction

In this notebook, you will build a deep neural network that functions as part of an end-to-end automatic speech recognition (ASR) pipeline! Your completed pipeline will accept raw audio as input and return a predicted transcription of the spoken language. The full pipeline is summarized in the figure below.

- **STEP 1** is a pre-processing step that converts raw audio to one of two feature representations that are commonly used for ASR.
- **STEP 2** is an acoustic model which accepts audio features as input and returns a probability distribution over all potential transcriptions. After learning about the basic types of neural networks that are often used for acoustic modeling, you will engage in your own investigations, to design your own acoustic model!
- **STEP 3** in the pipeline takes the output from the acoustic model and returns a predicted transcription.

Feel free to use the links below to navigate the notebook: - Section **??** - Section **??**: Acoustic Features for Speech Recognition - Section **??**: Deep Neural Networks for Acoustic Modeling - Section **??**: RNN - Section **??**: RNN + TimeDistributed Dense - Section **??**: CNN + RNN + TimeDistributed Dense - Section **??**: Deeper RNN + TimeDistributed Dense - Section **??**: Bidirectional RNN + TimeDistributed Dense - Section **??** - Section **??** - Section **??** - Section **??**: Obtain Predictions

The Data

We begin by investigating the dataset that will be used to train and evaluate your pipeline. LibriSpeech is a large corpus of English-read speech, designed for training and evaluating models for ASR. The dataset contains 1000 hours of speech derived from audiobooks. We will work with a small subset in this project, since larger-scale data would take a long while to train. However, after completing this project, if you are interested in exploring further, you are encouraged to work with more of the data that is provided online.

In the code cells below, you will use the `vis_train_features` module to visualize a training example. The supplied argument `index=0` tells the module to extract the first example in the training set. (You are welcome to change `index=0` to point to a different training example, if you like, but please **DO NOT** amend any other code in the cell.) The returned variables are: - `vis_text` - transcribed text (label) for the training example. - `vis_raw_audio` - raw audio waveform for the training example. - `vis_mfcc_feature` - mel-frequency cepstral coefficients (MFCCs) for the training example. - `vis_spectrogram_feature` - spectrogram for the training example. - `vis_audio_path` - the file path to the training example.

```
In [1]: from data_generator import vis_train_features

        # extract label and audio features for a single training example
        vis_text, vis_raw_audio, vis_mfcc_feature, vis_spectrogram_feature, vis_audio_path = vi
```

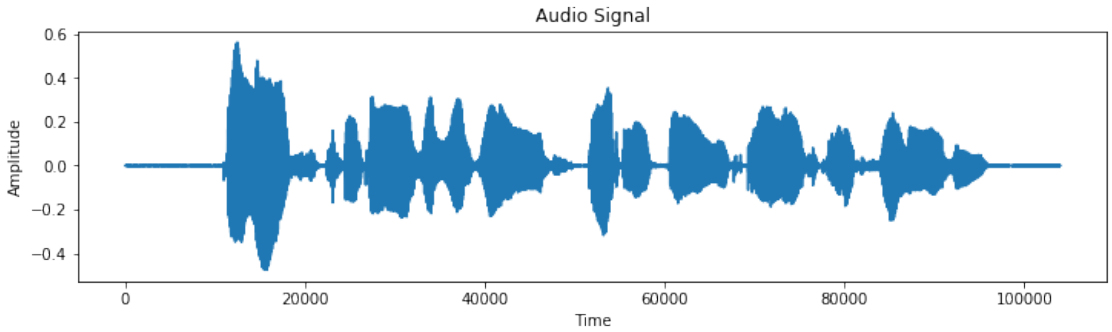
```
There are 2136 total training examples.
```

The following code cell visualizes the audio waveform for your chosen example, along with the corresponding transcript. You also have the option to play the audio in the notebook!

```
In [2]: from IPython.display import Markdown, display
        from data_generator import vis_train_features, plot_raw_audio
```

```
from IPython.display import Audio
%matplotlib inline

# plot audio signal
plot_raw_audio(vis_raw_audio)
# print length of audio signal
display(Markdown('**Shape of Audio Signal** : ' + str(vis_raw_audio.shape)))
# print transcript corresponding to audio clip
display(Markdown('**Transcript** : ' + str(vis_text)))
# play the audio file
Audio(vis_audio_path)
```



Shape of Audio Signal : (103966,)
Transcript : the last two days of the voyage bartley found almost intolerable

```
Out[2]: <IPython.lib.display.Audio object>
```

STEP 1: Acoustic Features for Speech Recognition

For this project, you won't use the raw audio waveform as input to your model. Instead, we provide code that first performs a pre-processing step to convert the raw audio to a feature representation that has historically proven successful for ASR models. Your acoustic model will accept the feature representation as input.

In this project, you will explore two possible feature representations. *After completing the project*, if you'd like to read more about deep learning architectures that can accept raw audio input, you are encouraged to explore this research paper.

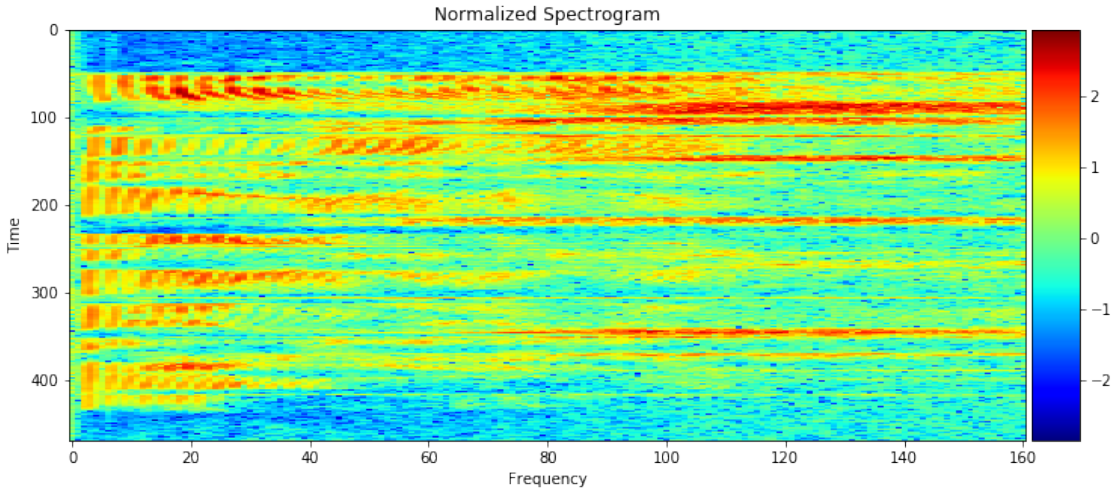
1.3.1 Spectrograms

The first option for an audio feature representation is the spectrogram. In order to complete this project, you will **not** need to dig deeply into the details of how a spectrogram is calculated; but, if you are curious, the code for calculating the spectrogram was borrowed from this repository. The implementation appears in the `utils.py` file in your repository.

The code that we give you returns the spectrogram as a 2D tensor, where the first (*vertical*) dimension indexes time, and the second (*horizontal*) dimension indexes frequency. To speed the convergence of your algorithm, we have also normalized the spectrogram. (You can see this quickly in the visualization below by noting that the mean value hovers around zero, and most entries in the tensor assume values close to zero.)

```
In [3]: from data_generator import plot_spectrogram_feature

        # plot normalized spectrogram
        plot_spectrogram_feature(vis_spectrogram_feature)
        # print shape of spectrogram
        display(Markdown('**Shape of Spectrogram** : ' + str(vis_spectrogram_feature.shape)))
```



Shape of Spectrogram : (470, 161)

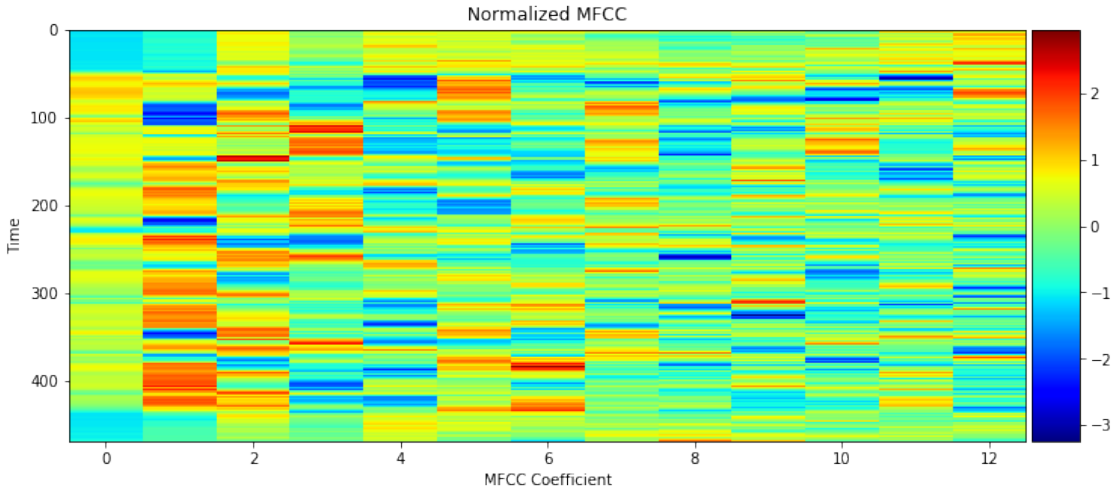
1.3.2 Mel-Frequency Cepstral Coefficients (MFCCs)

The second option for an audio feature representation is MFCCs. You do **not** need to dig deeply into the details of how MFCCs are calculated, but if you would like more information, you are welcome to peruse the documentation of the `python_speech_features` Python package. Just as with the spectrogram features, the MFCCs are normalized in the supplied code.

The main idea behind MFCC features is the same as spectrogram features: at each time window, the MFCC feature yields a feature vector that characterizes the sound within the window. Note that the MFCC feature is much lower-dimensional than the spectrogram feature, which could help an acoustic model to avoid overfitting to the training dataset.

```
In [4]: from data_generator import plot_mfcc_feature

        # plot normalized MFCC
        plot_mfcc_feature(vis_mfcc_feature)
        # print shape of MFCC
        display(Markdown('**Shape of MFCC** : ' + str(vis_mfcc_feature.shape)))
```



Shape of MFCC : (470, 13)

When you construct your pipeline, you will be able to choose to use either spectrogram or MFCC features. If you would like to see different implementations that make use of MFCCs and/or spectrograms, please check out the links below: - This repository uses spectrograms. - This repository uses MFCCs. - This repository also uses MFCCs. - This repository experiments with raw audio, spectrograms, and MFCCs as features.

STEP 2: Deep Neural Networks for Acoustic Modeling

In this section, you will experiment with various neural network architectures for acoustic modeling.

You will begin by training five relatively simple architectures. **Model 0** is provided for you. You will write code to implement **Models 1**, **2**, **3**, and **4**. If you would like to experiment further, you are welcome to create and train more models under the **Models 5+** heading.

All models will be specified in the `sample_models.py` file. After importing the `sample_models` module, you will train your architectures in the notebook.

After experimenting with the five simple architectures, you will have the opportunity to compare their performance. Based on your findings, you will construct a deeper architecture that is designed to outperform all of the shallow models.

For your convenience, we have designed the notebook so that each model can be specified and trained on separate occasions. That is, say you decide to take a break from the notebook after training **Model 1**. Then, you need not re-execute all prior code cells in the notebook before training **Model 2**. You need only re-execute the code cell below, that is marked with `RUN THIS CODE CELL IF YOU ARE RESUMING THE NOTEBOOK AFTER A BREAK`, before transitioning to the code cells corresponding to **Model 2**.

```
In [11]: ######################################################################
         # RUN THIS CODE CELL IF YOU ARE RESUMING THE NOTEBOOK AFTER A BREAK #
         ######################################################################

         # allocate 50% of GPU memory (if you like, feel free to change this)
         from keras.backend.tensorflow_backend import set_session
         import tensorflow as tf
```

```
        #config = tf.ConfigProto()
        #config.gpu_options.per_process_gpu_memory_fraction = 0.5
        #set_session(tf.Session(config=config))

        # watch for any changes in the sample_models module, and reload it automatically
        %load_ext autoreload
        %autoreload 2
        # import NN architectures for speech recognition
        from sample_models import *
        # import function for training acoustic model
        from train_utils import train_model
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

Model 0: RNN

Given their effectiveness in modeling sequential data, the first acoustic model you will use is an RNN. As shown in the figure below, the RNN we supply to you will take the time sequence of audio features as input.

At each time step, the speaker pronounces one of 28 possible characters, including each of the 26 letters in the English alphabet, along with a space character (" "), and an apostrophe (').

The output of the RNN at each time step is a vector of probabilities with 29 entries, where the i-th entry encodes the probability that the i-th character is spoken in the time sequence. (The extra 29th character is an empty "character" used to pad training examples within batches containing uneven lengths.) If you would like to peek under the hood at how characters are mapped to indices in the probability vector, look at the `char_map.py` file in the repository. The figure below shows an equivalent, rolled depiction of the RNN that shows the output layer in greater detail.

The model has already been specified for you in Keras. To import it, you need only run the code cell below.

```
In [6]: model_0 = simple_rnn_model(input_dim=161) # change to 13 if you would like to use MFCC
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
the_input (InputLayer)       (None, None, 161)         0
_________________________________________________________________
rnn (GRU)                    (None, None, 29)          16617
_________________________________________________________________
softmax (Activation)         (None, None, 29)          0
=================================================================
Total params: 16,617
Trainable params: 16,617
Non-trainable params: 0
_________________________________________________________________
None
```

As explored in the lesson, you will train the acoustic model with the CTC loss criterion. Custom loss functions take a bit of hacking in Keras, and so we have implemented the CTC loss function for you, so that you can focus on trying out as many deep learning architectures as possible :). If you'd like to peek at the implementation details, look at the `add_ctc_loss` function within the `train_utils.py` file in the repository.

To train your architecture, you will use the `train_model` function within the `train_utils` module; it has already been imported in one of the above code cells. The `train_model` function takes three **required** arguments: - `input_to_softmax` - a Keras model instance. - `pickle_path` - the name of the pickle file where the loss history will be saved. - `save_model_path` - the name of the HDF5 file where the model will be saved.

If we have already supplied values for `input_to_softmax`, `pickle_path`, and `save_model_path`, please **DO NOT** modify these values.

There are several **optional** arguments that allow you to have more control over the training process. You are welcome to, but not required to, supply your own values for these arguments. - `minibatch_size` - the size of the minibatches that are generated while training the model (default: `20`). - `spectrogram` - Boolean value dictating whether spectrogram (`True`) or MFCC (`False`) features are used for training (default: `True`). - `mfcc_dim` - the size of the feature dimension to use when generating MFCC features (default: `13`). - `optimizer` - the Keras optimizer used to train the model (default: `SGD(lr=0.02, decay=1e-6, momentum=0.9, nesterov=True, clipnorm=5)`). - `epochs` - the number of epochs to use to train the model (default: `20`). If you choose to modify this parameter, make sure that it is *at least* 20. - `verbose` - controls the verbosity of the training output in the `model.fit_generator` method (default: `1`). - `sort_by_duration` - Boolean value dictating whether the training and validation sets are sorted by (increasing) duration before the start of the first epoch (default: `False`).

The `train_model` function defaults to using spectrogram features; if you choose to use these features, note that the acoustic model in `simple_rnn_model` should have `input_dim=161`. Otherwise, if you choose to use MFCC features, the acoustic model should have `input_dim=13`.

We have chosen to use `GRU` units in the supplied RNN. If you would like to experiment with `LSTM` or `SimpleRNN` cells, feel free to do so here. If you change the `GRU` units to `SimpleRNN` cells in `simple_rnn_model`, you may notice that the loss quickly becomes undefined (`nan`) - you are strongly encouraged to check this for yourself! This is due to the exploding gradients problem. We have already implemented gradient clipping in your optimizer to help you avoid this issue.

IMPORTANT NOTE: If you notice that your gradient has exploded in any of the models below, feel free to explore more with gradient clipping (the `clipnorm` argument in your optimizer) or swap out any `SimpleRNN` cells for `LSTM` or `GRU` cells. You can also try restarting the kernel to restart the training process.

```
In [7]: train_model(input_to_softmax=model_0,
                    pickle_path='model_0.pickle',
                    save_model_path='model_0.h5',
                    spectrogram=True) # change to False if you would like to use MFCC features
```

```
Epoch 1/20
106/106 [==============================] - 145s 1s/step - loss: 843.5545 - val_loss: 736.6730
Epoch 2/20
106/106 [==============================] - 144s 1s/step - loss: 759.3367 - val_loss: 725.5323
Epoch 3/20
106/106 [==============================] - 144s 1s/step - loss: 753.1206 - val_loss: 727.7437
```

```
Epoch 4/20
106/106 [==============================] - 144s 1s/step - loss: 752.5001 - val_loss: 729.3181
Epoch 5/20
106/106 [==============================] - 143s 1s/step - loss: 752.0626 - val_loss: 726.7641
Epoch 6/20
106/106 [==============================] - 142s 1s/step - loss: 750.9767 - val_loss: 724.2145
Epoch 7/20
106/106 [==============================] - 141s 1s/step - loss: 752.1639 - val_loss: 731.0447
Epoch 8/20
106/106 [==============================] - 141s 1s/step - loss: 751.9760 - val_loss: 732.1049
Epoch 9/20
106/106 [==============================] - 141s 1s/step - loss: 752.2521 - val_loss: 723.7136
Epoch 10/20
106/106 [==============================] - 140s 1s/step - loss: 751.8776 - val_loss: 727.1290
Epoch 11/20
106/106 [==============================] - 142s 1s/step - loss: 751.2792 - val_loss: 725.9709
Epoch 12/20
106/106 [==============================] - 141s 1s/step - loss: 751.3884 - val_loss: 722.6850
Epoch 13/20
106/106 [==============================] - 140s 1s/step - loss: 752.1030 - val_loss: 733.1521
Epoch 14/20
106/106 [==============================] - 140s 1s/step - loss: 752.0450 - val_loss: 728.0485
Epoch 15/20
106/106 [==============================] - 141s 1s/step - loss: 751.6147 - val_loss: 726.2926
Epoch 16/20
106/106 [==============================] - 139s 1s/step - loss: 752.2462 - val_loss: 714.2499
Epoch 17/20
106/106 [==============================] - 142s 1s/step - loss: 752.1877 - val_loss: 740.3504
Epoch 18/20
106/106 [==============================] - 141s 1s/step - loss: 752.1010 - val_loss: 720.0252
Epoch 19/20
106/106 [==============================] - 141s 1s/step - loss: 752.9576 - val_loss: 721.2199
Epoch 20/20
106/106 [==============================] - 140s 1s/step - loss: 751.6413 - val_loss: 730.1493
```

(IMPLEMENTATION) Model 1: RNN + TimeDistributed Dense

Read about the TimeDistributed wrapper and the BatchNormalization layer in the Keras documentation. For your next architecture, you will add batch normalization to the recurrent layer to reduce training times. The `TimeDistributed` layer will be used to find more complex patterns in the dataset. The unrolled snapshot of the architecture is depicted below.

The next figure shows an equivalent, rolled depiction of the RNN that shows the (`TimeDistrbuted`) dense and output layers in greater detail.

Use your research to complete the `rnn_model` function within the `sample_models.py` file. The function should specify an architecture that satisfies the following requirements: - The first layer of the neural network should be an RNN (`SimpleRNN`, `LSTM`, or `GRU`) that takes the time sequence of audio features as input. We have added `GRU` units for you, but feel free to change `GRU` to `SimpleRNN` or `LSTM`, if you like! - Whereas the architecture in `simple_rnn_model` treated the RNN output

as the final layer of the model, you will use the output of your RNN as a hidden layer. Use `TimeDistributed` to apply a `Dense` layer to each of the time steps in the RNN output. Ensure that each `Dense` layer has `output_dim` units.

Use the code cell below to load your model into the `model_1` variable. Use a value for `input_dim` that matches your chosen audio features, and feel free to change the values for `units` and `activation` to tweak the behavior of your recurrent layer.

```
In [2]: model_1 = rnn_model(input_dim=161, # change to 13 if you would like to use MFCC featur
                            units=200,
                            activation='relu')
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
the_input (InputLayer)       (None, None, 161)         0
_________________________________________________________________
gru_1 (GRU)                  (None, None, 200)         217200
_________________________________________________________________
batch_normalization_1 (Batch (None, None, 200)         800
_________________________________________________________________
time_distributed_1 (TimeDist (None, None, 29)          5829
_________________________________________________________________
softmax (Activation)         (None, None, 29)          0
=================================================================
Total params: 223,829
Trainable params: 223,429
Non-trainable params: 400
_________________________________________________________________
None
```

Please execute the code cell below to train the neural network you specified in `input_to_softmax`. After the model has finished training, the model is saved in the HDF5 file `model_1.h5`. The loss history is saved in `model_1.pickle`. You are welcome to tweak any of the optional parameters while calling the `train_model` function, but this is not required.

```
In [11]: train_model(input_to_softmax=model_1,
                     pickle_path='model_1.pickle',
                     save_model_path='model_1.h5',
                     spectrogram=True) # change to False if you would like to use MFCC feature
```

```
Epoch 1/20
106/106 [==============================] - 140s 1s/step - loss: 284.4611 - val_loss: 215.4652
Epoch 2/20
106/106 [==============================] - 143s 1s/step - loss: 205.5376 - val_loss: 192.4191
Epoch 3/20
106/106 [==============================] - 144s 1s/step - loss: 184.6638 - val_loss: 180.3632
Epoch 4/20
106/106 [==============================] - 142s 1s/step - loss: 166.9435 - val_loss: 157.9452
```

```
Epoch 5/20
106/106 [==============================] - 141s 1s/step - loss: 156.1864 - val_loss: 155.0192
Epoch 6/20
106/106 [==============================] - 143s 1s/step - loss: 149.3683 - val_loss: 148.9709
Epoch 7/20
106/106 [==============================] - 144s 1s/step - loss: 145.5144 - val_loss: 152.3570
Epoch 8/20
106/106 [==============================] - 142s 1s/step - loss: 141.7052 - val_loss: 146.3004
Epoch 9/20
106/106 [==============================] - 143s 1s/step - loss: 139.0751 - val_loss: 142.3439
Epoch 10/20
106/106 [==============================] - 142s 1s/step - loss: 136.0708 - val_loss: 144.6307
Epoch 11/20
106/106 [==============================] - 142s 1s/step - loss: 134.3579 - val_loss: 146.5164
Epoch 12/20
106/106 [==============================] - 142s 1s/step - loss: 132.8003 - val_loss: 142.0703
Epoch 13/20
106/106 [==============================] - 141s 1s/step - loss: 133.6372 - val_loss: 144.7329
Epoch 14/20
106/106 [==============================] - 142s 1s/step - loss: 132.4990 - val_loss: 144.3944
Epoch 15/20
106/106 [==============================] - 143s 1s/step - loss: 129.7148 - val_loss: 142.8956
Epoch 16/20
106/106 [==============================] - 143s 1s/step - loss: 128.1105 - val_loss: 142.4843
Epoch 17/20
106/106 [==============================] - 142s 1s/step - loss: 130.2341 - val_loss: 145.6777
Epoch 18/20
106/106 [==============================] - 142s 1s/step - loss: 129.5090 - val_loss: 141.0543
Epoch 19/20
106/106 [==============================] - 143s 1s/step - loss: 127.6395 - val_loss: 140.1938
Epoch 20/20
106/106 [==============================] - 142s 1s/step - loss: 126.0189 - val_loss: 141.7793
```

(IMPLEMENTATION) Model 2: CNN + RNN + TimeDistributed Dense

The architecture in `cnn_rnn_model` adds an additional level of complexity, by introducing a 1D convolution layer.

This layer incorporates many arguments that can be (optionally) tuned when calling the `cnn_rnn_model` module. We provide sample starting parameters, which you might find useful if you choose to use spectrogram audio features.

If you instead want to use MFCC features, these arguments will have to be tuned. Note that the current architecture only supports values of `'same'` or `'valid'` for the `conv_border_mode` argument.

When tuning the parameters, be careful not to choose settings that make the convolutional layer overly small. If the temporal length of the CNN layer is shorter than the length of the transcribed text label, your code will throw an error.

Before running the code cell below, you must modify the `cnn_rnn_model` function in `sample_models.py`. Please add batch normalization to the recurrent layer, and provide the same

TimeDistributed layer as before.

```
In [13]: model_2 = cnn_rnn_model(input_dim=161, # change to 13 if you would like to use MFCC f
                                 filters=200,
                                 kernel_size=11,
                                 conv_stride=2,
                                 conv_border_mode='valid',
                                 units=200)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
the_input (InputLayer)       (None, None, 161)         0
_________________________________________________________________
conv1d (Conv1D)              (None, None, 200)         354400
_________________________________________________________________
bn_conv_1d (BatchNormalizati (None, None, 200)         800
_________________________________________________________________
rnn (SimpleRNN)              (None, None, 200)         80200
_________________________________________________________________
bn_rnn (BatchNormalization)  (None, None, 200)         800
_________________________________________________________________
time_distributed_4 (TimeDist (None, None, 29)          5829
_________________________________________________________________
softmax (Activation)         (None, None, 29)          0
=================================================================
Total params: 442,029
Trainable params: 441,229
Non-trainable params: 800
_________________________________________________________________
None
```

Please execute the code cell below to train the neural network you specified in `input_to_softmax`. After the model has finished training, the model is saved in the HDF5 file `model_2.h5`. The loss history is saved in `model_2.pickle`. You are welcome to tweak any of the optional parameters while calling the `train_model` function, but this is not required.

```
In [14]: train_model(input_to_softmax=model_2,
                     pickle_path='model_2.pickle',
                     save_model_path='model_2.h5',
                     spectrogram=True) # change to False if you would like to use MFCC feature
```

```
Epoch 1/20
106/106 [==============================] - 68s 638ms/step - loss: 233.8269 - val_loss: 201.560
Epoch 2/20
106/106 [==============================] - 34s 316ms/step - loss: 169.8235 - val_loss: 159.776
Epoch 3/20
106/106 [==============================] - 34s 317ms/step - loss: 149.2567 - val_loss: 145.420
```

```
Epoch 4/20
106/106 [==============================] - 33s 314ms/step - loss: 137.9520 - val_loss: 141.9096
Epoch 5/20
106/106 [==============================] - 33s 313ms/step - loss: 129.6079 - val_loss: 134.8034
Epoch 6/20
106/106 [==============================] - 33s 313ms/step - loss: 122.3582 - val_loss: 136.0375
Epoch 7/20
106/106 [==============================] - 33s 312ms/step - loss: 117.1146 - val_loss: 129.8337
Epoch 8/20
106/106 [==============================] - 33s 311ms/step - loss: 112.8061 - val_loss: 128.8583
Epoch 9/20
106/106 [==============================] - 33s 313ms/step - loss: 108.8604 - val_loss: 132.1610
Epoch 10/20
106/106 [==============================] - 33s 315ms/step - loss: 105.1657 - val_loss: 131.1508
Epoch 11/20
106/106 [==============================] - 33s 310ms/step - loss: 101.2671 - val_loss: 128.3527
Epoch 12/20
106/106 [==============================] - 33s 310ms/step - loss: 98.0596 - val_loss: 131.1613
Epoch 13/20
106/106 [==============================] - 33s 313ms/step - loss: 95.2325 - val_loss: 129.9068
Epoch 14/20
106/106 [==============================] - 33s 313ms/step - loss: 92.2111 - val_loss: 131.6800
Epoch 15/20
106/106 [==============================] - 33s 308ms/step - loss: 89.3839 - val_loss: 134.1484
Epoch 16/20
106/106 [==============================] - 33s 313ms/step - loss: 87.0646 - val_loss: 133.7997
Epoch 17/20
106/106 [==============================] - 33s 308ms/step - loss: 84.4525 - val_loss: 133.9794
Epoch 18/20
106/106 [==============================] - 33s 309ms/step - loss: 81.8300 - val_loss: 137.0172
Epoch 19/20
106/106 [==============================] - 33s 309ms/step - loss: 79.7719 - val_loss: 136.0165
Epoch 20/20
106/106 [==============================] - 33s 313ms/step - loss: 77.7044 - val_loss: 140.1428
```

(IMPLEMENTATION) Model 3: Deeper RNN + TimeDistributed Dense

Review the code in `rnn_model`, which makes use of a single recurrent layer. Now, specify an architecture in `deep_rnn_model` that utilizes a variable number `recur_layers` of recurrent layers. The figure below shows the architecture that should be returned if `recur_layers=2`. In the figure, the output sequence of the first recurrent layer is used as input for the next recurrent layer.

Feel free to change the supplied values of `units` to whatever you think performs best. You can change the value of `recur_layers`, as long as your final value is greater than 1. (As a quick check that you have implemented the additional functionality in `deep_rnn_model` correctly, make sure that the architecture that you specify here is identical to `rnn_model` if `recur_layers=1`.)

```
In [3]: model_3 = deep_rnn_model(input_dim=161, # change to 13 if you would like to use MFCC f
                                 units=200,
                                 recur_layers=2)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
the_input (InputLayer)       (None, None, 161)         0
_________________________________________________________________
gru_1 (GRU)                  (None, None, 200)         217200
_________________________________________________________________
batch_normalization_1 (Batch (None, None, 200)         800
_________________________________________________________________
gru_2 (GRU)                  (None, None, 200)         240600
_________________________________________________________________
batch_normalization_2 (Batch (None, None, 200)         800
_________________________________________________________________
time_distributed_1 (TimeDist (None, None, 29)          5829
_________________________________________________________________
softmax (Activation)         (None, None, 29)          0
=================================================================
Total params: 465,229
Trainable params: 464,429
Non-trainable params: 800
_________________________________________________________________
None
```

Please execute the code cell below to train the neural network you specified in `input_to_softmax`. After the model has finished training, the model is saved in the HDF5 file `model_3.h5`. The loss history is saved in `model_3.pickle`. You are welcome to tweak any of the optional parameters while calling the `train_model` function, but this is not required.

```
In [4]: train_model(input_to_softmax=model_3,
                    pickle_path='model_3.pickle',
                    save_model_path='model_3.h5',
                    spectrogram=True) # change to False if you would like to use MFCC features
```

```
Epoch 1/20
106/106 [==============================] - 374s - loss: 295.9878 - val_loss: 288.3476
Epoch 2/20
106/106 [==============================] - 371s - loss: 208.7503 - val_loss: 195.6717
Epoch 3/20
106/106 [==============================] - 372s - loss: 176.3261 - val_loss: 172.6310
Epoch 4/20
106/106 [==============================] - 372s - loss: 155.5491 - val_loss: 157.3287
Epoch 5/20
106/106 [==============================] - 368s - loss: 143.4091 - val_loss: 146.4786
Epoch 6/20
106/106 [==============================] - 368s - loss: 134.9470 - val_loss: 140.7857
Epoch 7/20
106/106 [==============================] - 366s - loss: 128.6154 - val_loss: 137.8512
```

```
Epoch 8/20
106/106 [==============================] - 368s - loss: 123.1451 - val_loss: 135.3723
Epoch 9/20
106/106 [==============================] - 373s - loss: 119.0034 - val_loss: 141.5648
Epoch 10/20
106/106 [==============================] - 369s - loss: 117.1938 - val_loss: 132.3246
Epoch 11/20
106/106 [==============================] - 373s - loss: 112.9313 - val_loss: 128.8022
Epoch 12/20
106/106 [==============================] - 372s - loss: 110.6216 - val_loss: 132.0946
Epoch 13/20
106/106 [==============================] - 372s - loss: 108.2203 - val_loss: 128.1300
Epoch 14/20
106/106 [==============================] - 375s - loss: 106.3406 - val_loss: 126.1110
Epoch 15/20
106/106 [==============================] - 368s - loss: 103.9524 - val_loss: 125.7774
Epoch 16/20
106/106 [==============================] - 369s - loss: 101.0038 - val_loss: 123.0792
Epoch 17/20
106/106 [==============================] - 368s - loss: 99.8780 - val_loss: 122.6394
Epoch 18/20
106/106 [==============================] - 370s - loss: 98.7415 - val_loss: 124.8694
Epoch 19/20
106/106 [==============================] - 369s - loss: 101.0782 - val_loss: 125.0774
Epoch 20/20
106/106 [==============================] - 369s - loss: 98.7292 - val_loss: 122.8388
```

(IMPLEMENTATION) Model 4: Bidirectional RNN + TimeDistributed Dense

Read about the Bidirectional wrapper in the Keras documentation. For your next architecture, you will specify an architecture that uses a single bidirectional RNN layer, before a (`TimeDistributed`) dense layer. The added value of a bidirectional RNN is described well in this paper. > One shortcoming of conventional RNNs is that they are only able to make use of previous context. In speech recognition, where whole utterances are transcribed at once, there is no reason not to exploit future context as well. Bidirectional RNNs (BRNNs) do this by processing the data in both directions with two separate hidden layers which are then fed forwards to the same output layer.

Before running the code cell below, you must complete the `bidirectional_rnn_model` function in `sample_models.py`. Feel free to use `SimpleRNN`, `LSTM`, or `GRU` units. When specifying the `Bidirectional` wrapper, use `merge_mode='concat'`.

```
In [5]: model_4 = bidirectional_rnn_model(input_dim=161, # change to 13 if you would like to u
                                          units=200)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
the_input (InputLayer)       (None, None, 161)         0
```

```
_________________________________________________________________
bidirectional_1 (Bidirection (None, None, 400)         434400
_________________________________________________________________
time_distributed_2 (TimeDist (None, None, 29)          11629
_________________________________________________________________
softmax (Activation)         (None, None, 29)          0
=================================================================
Total params: 446,029
Trainable params: 446,029
Non-trainable params: 0
_________________________________________________________________
None
```

Please execute the code cell below to train the neural network you specified in `input_to_softmax`. After the model has finished training, the model is saved in the HDF5 file `model_4.h5`. The loss history is saved in `model_4.pickle`. You are welcome to tweak any of the optional parameters while calling the `train_model` function, but this is not required.

```
In [6]: train_model(input_to_softmax=model_4,
                    pickle_path='model_4.pickle',
                    save_model_path='model_4.h5',
                    spectrogram=True) # change to False if you would like to use MFCC features
```

```
Epoch 1/20
106/106 [==============================] - 357s - loss: 255.7198 - val_loss: 205.5528
Epoch 2/20
106/106 [==============================] - 363s - loss: 202.8234 - val_loss: 188.7644
Epoch 3/20
106/106 [==============================] - 364s - loss: 189.3287 - val_loss: 180.1132
Epoch 4/20
106/106 [==============================] - 360s - loss: 180.0773 - val_loss: 174.9297
Epoch 5/20
106/106 [==============================] - 361s - loss: 172.4668 - val_loss: 165.7650
Epoch 6/20
106/106 [==============================] - 361s - loss: 165.0099 - val_loss: 161.9585
Epoch 7/20
106/106 [==============================] - 361s - loss: 158.6284 - val_loss: 159.2820
Epoch 8/20
106/106 [==============================] - 364s - loss: 153.0483 - val_loss: 154.9917
Epoch 9/20
106/106 [==============================] - 362s - loss: 147.8917 - val_loss: 153.1364
Epoch 10/20
106/106 [==============================] - 363s - loss: 142.2610 - val_loss: 147.3989
Epoch 11/20
106/106 [==============================] - 364s - loss: 137.1105 - val_loss: 143.5334
Epoch 12/20
106/106 [==============================] - 363s - loss: 132.9617 - val_loss: 143.5190
```

```
Epoch 13/20
106/106 [==============================] - 362s - loss: 128.9160 - val_loss: 140.2615
Epoch 14/20
106/106 [==============================] - 361s - loss: 125.4127 - val_loss: 141.8306
Epoch 15/20
106/106 [==============================] - 363s - loss: 122.1888 - val_loss: 136.4900
Epoch 16/20
106/106 [==============================] - 360s - loss: 119.1285 - val_loss: 139.3063
Epoch 17/20
106/106 [==============================] - 363s - loss: 116.3920 - val_loss: 138.0645
Epoch 18/20
106/106 [==============================] - 362s - loss: 113.2511 - val_loss: 134.0935
Epoch 19/20
106/106 [==============================] - 361s - loss: 110.6029 - val_loss: 132.3267
Epoch 20/20
106/106 [==============================] - 361s - loss: 108.1646 - val_loss: 134.0507
```

(OPTIONAL IMPLEMENTATION) Models 5+

If you would like to try out more architectures than the ones above, please use the code cell below. Please continue to follow the same convention for saving the models; for the *i*-th sample model, please save the loss at **`model_i.pickle`** and saving the trained model at **`model_i.h5`**.

In [26]:
```
# This is not the actual network that was trained in cell 27.
# Original code that generated model for cell 27 was lost (but model was saved to H5 )
# See Note in Answer 1.

model_15_copy = deep_cnn_cudnngru_model(input_dim=13,
                                        filters=200,
                                        num_layers=3,
                                        kernel_size=11,
                                        conv_stride=1,
                                        conv_border_mode='valid',
                                        dropout_rate=0.3,
                                        units=200)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
the_input (InputLayer)       (None, None, 13)          0
_________________________________________________________________
conv1d_36 (Conv1D)           (None, None, 200)         28800
_________________________________________________________________
leaky_re_lu_45 (LeakyReLU)   (None, None, 200)         0
_________________________________________________________________
batch_normalization_45 (Batc (None, None, 200)         800
_________________________________________________________________
dropout_45 (Dropout)         (None, None, 200)         0
```

```
_________________________________________________________________
conv1d_37 (Conv1D)           (None, None, 200)         440200
_________________________________________________________________
leaky_re_lu_46 (LeakyReLU)   (None, None, 200)         0
_________________________________________________________________
batch_normalization_46 (Batc (None, None, 200)         800
_________________________________________________________________
dropout_46 (Dropout)         (None, None, 200)         0
_________________________________________________________________
conv1d_38 (Conv1D)           (None, None, 200)         440200
_________________________________________________________________
max_pooling1d_13 (MaxPooling (None, None, 200)         0
_________________________________________________________________
cu_dnngru_25 (CuDNNGRU)      (None, None, 200)         241200
_________________________________________________________________
leaky_re_lu_47 (LeakyReLU)   (None, None, 200)         0
_________________________________________________________________
batch_normalization_47 (Batc (None, None, 200)         800
_________________________________________________________________
dropout_47 (Dropout)         (None, None, 200)         0
_________________________________________________________________
cu_dnngru_26 (CuDNNGRU)      (None, None, 200)         241200
_________________________________________________________________
leaky_re_lu_48 (LeakyReLU)   (None, None, 200)         0
_________________________________________________________________
batch_normalization_48 (Batc (None, None, 200)         800
_________________________________________________________________
dropout_48 (Dropout)         (None, None, 200)         0
_________________________________________________________________
bidirectional_18 (Bidirectio (None, None, 400)         482400
_________________________________________________________________
time_distributed_13 (TimeDis (None, None, 29)          11629
_________________________________________________________________
activation_13 (Activation)   (None, None, 29)          0
=================================================================
Total params: 1,888,829
Trainable params: 1,887,229
Non-trainable params: 1,600
_________________________________________________________________
None
```

```
In [20]: train_model(input_to_softmax=model_15_copy,
                     pickle_path='model_15.pickle',
                     save_model_path='model_15.h5',
                     epochs=50,
                     minibatch_size=10,
                     spectrogram=False)
```

```
Epoch 1/50
213/213 [==============================] - 132s 618ms/step - loss: 109.4286 - val_loss: 114.952
Epoch 2/50
213/213 [==============================] - 123s 578ms/step - loss: 107.1405 - val_loss: 111.498
Epoch 3/50
213/213 [==============================] - 122s 574ms/step - loss: 104.9539 - val_loss: 108.210
Epoch 4/50
213/213 [==============================] - 122s 572ms/step - loss: 103.2891 - val_loss: 106.683
Epoch 5/50
213/213 [==============================] - 121s 569ms/step - loss: 101.5617 - val_loss: 107.037
Epoch 6/50
213/213 [==============================] - 120s 566ms/step - loss: 99.6830 - val_loss: 103.7146
Epoch 7/50
213/213 [==============================] - 121s 568ms/step - loss: 97.6260 - val_loss: 105.4807
Epoch 8/50
213/213 [==============================] - 122s 571ms/step - loss: 96.0584 - val_loss: 103.5952
Epoch 9/50
213/213 [==============================] - 121s 566ms/step - loss: 94.0791 - val_loss: 103.0310
Epoch 10/50
213/213 [==============================] - 122s 571ms/step - loss: 92.3493 - val_loss: 102.1253
Epoch 11/50
213/213 [==============================] - 121s 567ms/step - loss: 90.4744 - val_loss: 101.2623
Epoch 12/50
213/213 [==============================] - 121s 568ms/step - loss: 89.0567 - val_loss: 102.2585
Epoch 13/50
213/213 [==============================] - 120s 565ms/step - loss: 86.7161 - val_loss: 101.3316
Epoch 14/50
213/213 [==============================] - 121s 566ms/step - loss: 85.5967 - val_loss: 99.0625
Epoch 15/50
213/213 [==============================] - 120s 564ms/step - loss: 84.0955 - val_loss: 98.3844
Epoch 16/50
213/213 [==============================] - 120s 565ms/step - loss: 82.4830 - val_loss: 99.0238
Epoch 17/50
213/213 [==============================] - 121s 568ms/step - loss: 81.1143 - val_loss: 98.5495
Epoch 18/50
213/213 [==============================] - 121s 566ms/step - loss: 79.8793 - val_loss: 98.0829
Epoch 19/50
213/213 [==============================] - 121s 569ms/step - loss: 78.5824 - val_loss: 97.1214
Epoch 20/50
213/213 [==============================] - 120s 566ms/step - loss: 77.4555 - val_loss: 94.5170
Epoch 21/50
213/213 [==============================] - 121s 567ms/step - loss: 76.1335 - val_loss: 95.1506
Epoch 22/50
213/213 [==============================] - 121s 566ms/step - loss: 74.9732 - val_loss: 96.3938
Epoch 23/50
213/213 [==============================] - 121s 568ms/step - loss: 74.3944 - val_loss: 97.0851
Epoch 24/50
213/213 [==============================] - 120s 565ms/step - loss: 73.2983 - val_loss: 94.8676
```

```
Epoch 00024: early stopping
```

Compare the Models

Execute the code cell below to evaluate the performance of the drafted deep learning models. The training and validation loss are plotted for each model.

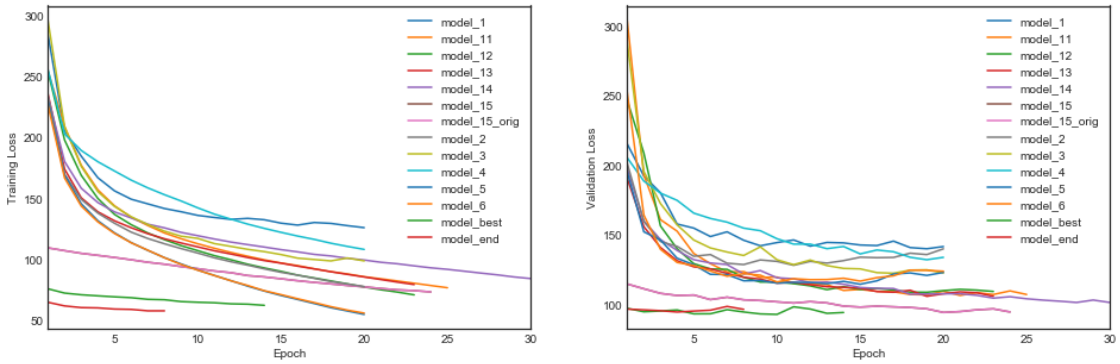
```
In [67]: from glob import glob
         import numpy as np
         import _pickle as pickle
         import seaborn as sns
         import matplotlib.pyplot as plt
         %matplotlib inline
         sns.set_style(style='white')

         # obtain the paths for the saved model history
         all_pickles = sorted(glob("results/*.pickle"))
         # extract the name of each model
         model_names = [item[8:-7] for item in all_pickles]
         # extract the loss history for each model
         valid_loss = [pickle.load( open( i, "rb" ) )['val_loss'] for i in all_pickles]
         train_loss = [pickle.load( open( i, "rb" ) )['loss'] for i in all_pickles]
         # save the number of epochs used to train each model
         num_epochs = [len(valid_loss[i]) for i in range(len(valid_loss))]

         fig = plt.figure(figsize=(16,5))

         # plot the training loss vs. epoch for each model
         ax1 = fig.add_subplot(121)
         for i in range(len(all_pickles)):
             ax1.plot(np.linspace(1, num_epochs[i], num_epochs[i]),
                     train_loss[i], label=model_names[i])
         # clean up the plot
         ax1.legend()
         ax1.set_xlim([1, max(num_epochs)])
         plt.xlabel('Epoch')
         plt.ylabel('Training Loss')

         # plot the validation loss vs. epoch for each model
         ax2 = fig.add_subplot(122)
         for i in range(len(all_pickles)):
             ax2.plot(np.linspace(1, num_epochs[i], num_epochs[i]),
                     valid_loss[i], label=model_names[i])
         # clean up the plot
         ax2.legend()
         ax2.set_xlim([1, max(num_epochs)])
         plt.xlabel('Epoch')
         plt.ylabel('Validation Loss')
         plt.show()
```



Question 1: Use the plot above to analyze the performance of each of the attempted architectures. Which performs best? Provide an explanation regarding why you think some models perform better than others.

Answer:

1.3.3 Model 0: RNN

Epoch 16/20 | loss: 752.2462 - val_loss: 714.2499

This model had huge validation loss because it has a single layer, which was trying to map input directly to the output. This network was trying to map each input directly to a letter, which is of couse not possible.

1.3.4 Model 1: RNN + TimeDistributed

Epoch 18/20 | loss: 129.5090 - val_loss: 141.0543 * Total params: 223,829 * Trainable params: 223,429 * Non-trainable params: 400

Adding TimeDistributed to the network actually helped a lot. This is how network became capable of making sense of sliced continious input. The network is still too simple to learn complex patterns.

1.3.5 Model 2: CNN + RNN + TimeDistributed

Epoch 11/20 | loss: 101.2671 - val_loss: 128.3527 * Total params: 442,029 * Trainable params: 441,229 * Non-trainable params: 800

Adding Conv1D layer increased the ability of the network to extract patterns from the input. CNNs gain popularity in Sequence-to-Sequence modeling tasks. They allow us to increase the depth and expressive power of a network while reducing the total number of parameters that would have been needed otherwise to build such deeper models. (This paper cited here). This model started to overfit quite early (Epoch 11). It needs to be tuned, for example by adding Dropout, in order to show its full potential.

1.3.6 Model 3: Deeper RNN + + TimeDistributed

Epoch 20/20 | loss: 98.7292 - val_loss: 122.8388 * Total params: 465,229 * Trainable params: 464,429 * Non-trainable params: 800

This model deepens Model 2 by adding more RNN layers. In terms of number of parameters and performance it is very similar to Model 2. It is, however, didn't overfit the data as Model 2 did. Better performance could be achieved by simply letting it train longer than 20 epochs.

1.3.7 Model 4: Bidirectional RNN + TimeDistributed

Epoch 19/20 | 110.6029 - val_loss: 132.3267

Simply wrapping RNN layer in Bidirectional layer helped a little bit, but this alone is not enough to get significant performance boost. It is the slowest converging network out of 4 predefined models (discounting Model 0).

1.3.8 Model 15: Deep CNN + Deep Bidirectional RNN + TimeDistributed

Epoch 24/50 | loss: 73.2983 - val_loss: 94.8676 * Total params: 1,888,829 * Trainable params: 1,887,229 * Non-trainable params: 1,600

By combining together all good features from the above and taking measures to avoid overfitting, this network achieved whopping 94.8676 val_loss. Note, how many parameters this network has to train. But by using CuDNNGRU Keras layers and p2 AWS instance, training is very affordable, just 2 minutes per step. This network converged in just 20 epochs.

Note 1: I accidentally removed original code, which generates model_15. I, however, had H5 file and with a few tricks was able to restore the model from there (see cells below). H5 file is also added to GitHub repo, so this notebook is fully runnable if someone wants to validate this result.

Note 2: Notebook which contains a lot of experiments around this net can be found here

In [24]:
```
# Run these cells in order to restore model_15, which produced val_loss < 100

pool_size = 2
num_layers = 3
kernel_size = 11
conv_border_mode = 'valid'
conv_stride = 1
dilation_rate = (2,)

def cnn_output_length(input_length, filter_size, border_mode, stride,
                      dilation=1):
    """ Compute the length of the output sequence after 1D convolution along
        time. Note that this function is in line with the function used in
        Convolution1D class from Keras.
    Params:
        input_length (int): Length of the input sequence.
        filter_size (int): Width of the convolution kernel.
        border_mode (str): Only support `same` or `valid`.
        stride (int): Stride size used in 1D convolution.
        dilation (int)
    """
    if input_length is None:
        return None
    assert border_mode in {'same', 'valid'}
```

```
            dilated_filter_size = filter_size + (filter_size - 1) * (dilation - 1)
            if border_mode == 'same':
                output_length = input_length
            elif border_mode == 'valid':
                output_length = input_length - dilated_filter_size + 1
            return (output_length + stride - 1) // stride

        def calc_output_length(input_length):
            output_length = input_length
            for _ in range(num_layers):
                output_length = cnn_output_length(output_length, kernel_size,
                                                  conv_border_mode, conv_stride,
                                                  dilation=dilation_rate[0])
            return output_length / pool_size

        from keras.models import load_model

        best_model = load_model('model_15_orig.h5',
                                custom_objects={'cnn_output_length': cnn_output_length},
                                compile=False)
        best_model.output_length = calc_output_length
        print(best_model.summary())
```

```
_________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
=================================================================================================
the_input (InputLayer)       (None, None, 13)          0
_________________________________________________________________________________________________
conv1d_24 (Conv1D)           (None, None, 200)         28800       the_input[0][0]
_________________________________________________________________________________________________
leaky_re_lu_31 (LeakyReLU)   (None, None, 200)         0           conv1d_24[0][0]
_________________________________________________________________________________________________
batch_normalization_31 (BatchNo (None, None, 200)      800         leaky_re_lu_31[0][0]
_________________________________________________________________________________________________
dropout_31 (Dropout)         (None, None, 200)         0           batch_normalization_31[0][0]
_________________________________________________________________________________________________
conv1d_25 (Conv1D)           (None, None, 200)         440200      dropout_31[0][0]
_________________________________________________________________________________________________
leaky_re_lu_32 (LeakyReLU)   (None, None, 200)         0           conv1d_25[0][0]
_________________________________________________________________________________________________
batch_normalization_32 (BatchNo (None, None, 200)      800         leaky_re_lu_32[0][0]
_________________________________________________________________________________________________
dropout_32 (Dropout)         (None, None, 200)         0           batch_normalization_32[0][0]
_________________________________________________________________________________________________
conv1d_26 (Conv1D)           (None, None, 200)         440200      dropout_32[0][0]
_________________________________________________________________________________________________
max_pooling1d_9 (MaxPooling1D)  (None, None, 200)      0           conv1d_26[0][0]
_________________________________________________________________________________________________
```

```
cu_dnngru_15 (CuDNNGRU)         (None, None, 200)    241200      max_pooling1d_9[0][0]
_________________________________________________________________________________________
leaky_re_lu_33 (LeakyReLU)      (None, None, 200)    0           cu_dnngru_15[0][0]
_________________________________________________________________________________________
batch_normalization_33 (BatchNo (None, None, 200)    800         leaky_re_lu_33[0][0]
_________________________________________________________________________________________
dropout_33 (Dropout)            (None, None, 200)    0           batch_normalization_33[0][0]
_________________________________________________________________________________________
cu_dnngru_16 (CuDNNGRU)         (None, None, 200)    241200      dropout_33[0][0]
_________________________________________________________________________________________
leaky_re_lu_34 (LeakyReLU)      (None, None, 200)    0           cu_dnngru_16[0][0]
_________________________________________________________________________________________
batch_normalization_34 (BatchNo (None, None, 200)    800         leaky_re_lu_34[0][0]
_________________________________________________________________________________________
dropout_34 (Dropout)            (None, None, 200)    0           batch_normalization_34[0][0]
_________________________________________________________________________________________
bidirectional_12 (Bidirectional (None, None, 400)    482400      dropout_34[0][0]
_________________________________________________________________________________________
time_distributed_9 (TimeDistrib (None, None, 29)     11629       bidirectional_12[0][0]
_________________________________________________________________________________________
input_length (InputLayer)       (None, 1)            0
_________________________________________________________________________________________
activation_9 (Activation)       (None, None, 29)     0           time_distributed_9[0][0]
_________________________________________________________________________________________
the_labels (InputLayer)         (None, None)         0
_________________________________________________________________________________________
lambda_6 (Lambda)               (None, 1)            0           input_length[0][0]
_________________________________________________________________________________________
label_length (InputLayer)       (None, 1)            0
_________________________________________________________________________________________
ctc (Lambda)                    (None, 1)            0           activation_9[0][0]
                                                                 the_labels[0][0]
                                                                 lambda_6[0][0]
                                                                 label_length[0][0]
=========================================================================================
Total params: 1,888,829
Trainable params: 1,887,229
Non-trainable params: 1,600
_________________________________________________________________________________________
None
```

```
In [29]: train_model(input_to_softmax=best_model,
                     pickle_path='model_best.pickle',
                     save_model_path='model_best.h5',
                     epochs=50,
                     minibatch_size=10,
                     should_add_ctc_loss=False,
```

```
                    spectrogram=False)
```

```
Epoch 1/50
213/213 [==============================] - 131s 617ms/step - loss: 75.8554 - val_loss: 97.4926
Epoch 2/50
213/213 [==============================] - 123s 577ms/step - loss: 72.3491 - val_loss: 94.9903
Epoch 3/50
213/213 [==============================] - 122s 573ms/step - loss: 70.9712 - val_loss: 95.5792
Epoch 4/50
213/213 [==============================] - 122s 572ms/step - loss: 70.0864 - val_loss: 96.2840
Epoch 5/50
213/213 [==============================] - 121s 567ms/step - loss: 69.2675 - val_loss: 93.5092
Epoch 6/50
213/213 [==============================] - 122s 572ms/step - loss: 68.5595 - val_loss: 93.5667
Epoch 7/50
213/213 [==============================] - 122s 571ms/step - loss: 67.2267 - val_loss: 96.5023
Epoch 8/50
213/213 [==============================] - 122s 572ms/step - loss: 66.8884 - val_loss: 94.8412
Epoch 9/50
213/213 [==============================] - 120s 561ms/step - loss: 65.3730 - val_loss: 93.4666
Epoch 10/50
213/213 [==============================] - 120s 564ms/step - loss: 64.8291 - val_loss: 93.1120
Epoch 11/50
213/213 [==============================] - 122s 572ms/step - loss: 64.4042 - val_loss: 98.5606
Epoch 12/50
213/213 [==============================] - 120s 564ms/step - loss: 63.5424 - val_loss: 96.9593
Epoch 13/50
213/213 [==============================] - 121s 566ms/step - loss: 63.2401 - val_loss: 93.8743
Epoch 14/50
213/213 [==============================] - 121s 569ms/step - loss: 62.3107 - val_loss: 94.5014
Epoch 00014: early stopping
```

(IMPLEMENTATION) Final Model

Now that you've tried out many sample models, use what you've learned to draft your own architecture! While your final acoustic model should not be identical to any of the architectures explored above, you are welcome to merely combine the explored layers above into a deeper architecture. It is **NOT** necessary to include new layer types that were not explored in the notebook.

However, if you would like some ideas for even more layer types, check out these ideas for some additional, optional extensions to your model:

- If you notice your model is overfitting to the training dataset, consider adding **dropout**! To add dropout to recurrent layers, pay special attention to the `dropout_W` and `dropout_U` arguments. This paper may also provide some interesting theoretical background.
- If you choose to include a convolutional layer in your model, you may get better results by working with **dilated convolutions**. If you choose to use dilated convolutions, make sure that you are able to accurately calculate the length of the acoustic model's output in the `model.output_length` lambda function. You can read more about dilated convolutions

in Google's WaveNet paper. For an example of a speech-to-text system that makes use of dilated convolutions, check out this GitHub repository. You can work with dilated convolutions in Keras by paying special attention to the `padding` argument when you specify a convolutional layer.

- If your model makes use of convolutional layers, why not also experiment with adding **max pooling**? Check out this paper for example architecture that makes use of max pooling in an acoustic model.
- So far, you have experimented with a single bidirectional RNN layer. Consider stacking the bidirectional layers, to produce a deep bidirectional RNN!

All models that you specify in this repository should have `output_length` defined as an attribute. This attribute is a lambda function that maps the (temporal) length of the input acoustic features to the (temporal) length of the output softmax layer. This function is used in the computation of CTC loss; to see this, look at the `add_ctc_loss` function in `train_utils.py`. To see where the `output_length` attribute is defined for the models in the code, take a look at the `sample_models.py` file. You will notice this line of code within most models:

```
model.output_length = lambda x: x
```

The acoustic model that incorporates a convolutional layer (`cnn_rnn_model`) has a line that is a bit different:

```
model.output_length = lambda x: cnn_output_length(
        x, kernel_size, conv_border_mode, conv_stride)
```

In the case of models that use purely recurrent layers, the lambda function is the identity function, as the recurrent layers do not modify the (temporal) length of their input tensors. However, convolutional layers are more complicated and require a specialized function (`cnn_output_length` in `sample_models.py`) to determine the temporal length of their output.

You will have to add the `output_length` attribute to your final model before running the code cell below. Feel free to use the `cnn_output_length` function, if it suits your model.

```
In [30]: # specify the model
         model_end = best_model
         print(best_model.summary())
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
the_input (InputLayer)       (None, None, 13)          0
_________________________________________________________________
conv1d_24 (Conv1D)           (None, None, 200)         28800
_________________________________________________________________
leaky_re_lu_31 (LeakyReLU)   (None, None, 200)         0
_________________________________________________________________
batch_normalization_31 (BatchNo (None, None, 200)      800
_________________________________________________________________
dropout_31 (Dropout)         (None, None, 200)         0
_________________________________________________________________
```

```
conv1d_25 (Conv1D)              (None, None, 200)    440200      dropout_31[0][0]
_________________________________________________________________________________________
leaky_re_lu_32 (LeakyReLU)      (None, None, 200)    0           conv1d_25[0][0]
_________________________________________________________________________________________
batch_normalization_32 (BatchNo (None, None, 200)    800         leaky_re_lu_32[0][0]
_________________________________________________________________________________________
dropout_32 (Dropout)            (None, None, 200)    0           batch_normalization_32[0][0]
_________________________________________________________________________________________
conv1d_26 (Conv1D)              (None, None, 200)    440200      dropout_32[0][0]
_________________________________________________________________________________________
max_pooling1d_9 (MaxPooling1D)  (None, None, 200)    0           conv1d_26[0][0]
_________________________________________________________________________________________
cu_dnngru_15 (CuDNNGRU)         (None, None, 200)    241200      max_pooling1d_9[0][0]
_________________________________________________________________________________________
leaky_re_lu_33 (LeakyReLU)      (None, None, 200)    0           cu_dnngru_15[0][0]
_________________________________________________________________________________________
batch_normalization_33 (BatchNo (None, None, 200)    800         leaky_re_lu_33[0][0]
_________________________________________________________________________________________
dropout_33 (Dropout)            (None, None, 200)    0           batch_normalization_33[0][0]
_________________________________________________________________________________________
cu_dnngru_16 (CuDNNGRU)         (None, None, 200)    241200      dropout_33[0][0]
_________________________________________________________________________________________
leaky_re_lu_34 (LeakyReLU)      (None, None, 200)    0           cu_dnngru_16[0][0]
_________________________________________________________________________________________
batch_normalization_34 (BatchNo (None, None, 200)    800         leaky_re_lu_34[0][0]
_________________________________________________________________________________________
dropout_34 (Dropout)            (None, None, 200)    0           batch_normalization_34[0][0]
_________________________________________________________________________________________
bidirectional_12 (Bidirectional (None, None, 400)    482400      dropout_34[0][0]
_________________________________________________________________________________________
time_distributed_9 (TimeDistrib (None, None, 29)     11629       bidirectional_12[0][0]
_________________________________________________________________________________________
input_length (InputLayer)       (None, 1)            0
_________________________________________________________________________________________
activation_9 (Activation)       (None, None, 29)     0           time_distributed_9[0][0]
_________________________________________________________________________________________
the_labels (InputLayer)         (None, None)         0
_________________________________________________________________________________________
lambda_6 (Lambda)               (None, 1)            0           input_length[0][0]
_________________________________________________________________________________________
label_length (InputLayer)       (None, 1)            0
_________________________________________________________________________________________
ctc (Lambda)                    (None, 1)            0           activation_9[0][0]
                                                                 the_labels[0][0]
                                                                 lambda_6[0][0]
                                                                 label_length[0][0]
=========================================================================================
Total params: 1,888,829
```

```
Trainable params: 1,887,229
Non-trainable params: 1,600
_________________________________________________________________________________________________
None
```

Please execute the code cell below to train the neural network you specified in `input_to_softmax`. After the model has finished training, the model is saved in the HDF5 file `model_end.h5`. The loss history is saved in `model_end.pickle`. You are welcome to tweak any of the optional parameters while calling the `train_model` function, but this is not required.

```
In [32]: train_model(input_to_softmax=best_model,
                     pickle_path='model_end.pickle',
                     save_model_path='model_end.h5',
                     epochs=50,
                     minibatch_size=10,
                     should_add_ctc_loss=False,
                     spectrogram=False)
```

```
Epoch 1/50
213/213 [==============================] - 120s 562ms/step - loss: 64.8009 - val_loss: 96.9212
Epoch 2/50
213/213 [==============================] - 121s 568ms/step - loss: 61.8176 - val_loss: 96.3646
Epoch 3/50
213/213 [==============================] - 120s 565ms/step - loss: 60.4488 - val_loss: 95.7638
Epoch 4/50
213/213 [==============================] - 120s 565ms/step - loss: 60.2009 - val_loss: 94.8111
Epoch 5/50
213/213 [==============================] - 121s 567ms/step - loss: 59.2460 - val_loss: 95.5178
Epoch 6/50
213/213 [==============================] - 120s 565ms/step - loss: 58.9642 - val_loss: 96.1138
Epoch 7/50
213/213 [==============================] - 122s 571ms/step - loss: 57.7871 - val_loss: 98.8419
Epoch 8/50
213/213 [==============================] - 121s 567ms/step - loss: 57.7969 - val_loss: 96.7423
Epoch 00008: early stopping
```

Question 2: Describe your final model architecture and your reasoning at each step.

Answer: This model is a very deep neural network with a stack of Conv1D layers and a stack of GRU layers. Conv1D layers isolate patterns in sequential data. GRU layers learn to map sequences of those patterns to letters. Bidirectional wrapper is somewhat helpful, but doesn't make much difference by itself. Such a deep network wouldn't learn without several important tweaks. Batch Normalization and LeakyReLU help to keep gradients flowing. Dropout helps to prevent overfitting, especially when Convolutional stack is deep. This network leverages CuDNNGRU layers, which work very well on "AWD Deep Learning" AMI. Using this special layer provides significant boost for network training. My observation is that network with CuDNNGRU layers trains ~4 times faster than one with regular GRU layers.

Further improvements: * This network is very sensitive to hyperparameters, like number of layers, dropout rate etc. Tuning them is endless process, but my goal was to push val_loss under 100. * Experimenting with using different activation functions for different Convolutional and Recurrent layers * Experimenting with GRU vs LSTM * Experimenting with different optimizers and learning rates. One popular choice for sequence-to-sequence is RMSprop * Try to remove/reduce Dropout in RNN stack. RNN stack is less prone to overfitting. * Use different filters number/units number in different layers within each stack. * As was suggested in this notebook, plugging-in some Language model will be useful to produce meaningful words instead of sequences of letters that sound similar to real words.

STEP 3: Obtain Predictions

We have written a function for you to decode the predictions of your acoustic model. To use the function, please execute the code cell below.

```
In [37]: import numpy as np
         from data_generator import AudioGenerator
         from keras import backend as K
         from utils import int_sequence_to_text
         from IPython.display import Audio

         def get_predictions(index, partition, input_to_softmax, model_path, spectrogram=True)
             """ Print a model's decoded predictions
             Params:
                 index (int): The example you would like to visualize
                 partition (str): One of 'train' or 'validation'
                 input_to_softmax (Model): The acoustic model
                 model_path (str): Path to saved acoustic model's weights
             """
             # load the train and test data
             data_gen = AudioGenerator(spectrogram=spectrogram)
             data_gen.load_train_data()
             data_gen.load_validation_data()

             # obtain the true transcription and the audio features
             if partition == 'validation':
                 transcr = data_gen.valid_texts[index]
                 audio_path = data_gen.valid_audio_paths[index]
                 data_point = data_gen.normalize(data_gen.featurize(audio_path))
             elif partition == 'train':
                 transcr = data_gen.train_texts[index]
                 audio_path = data_gen.train_audio_paths[index]
                 data_point = data_gen.normalize(data_gen.featurize(audio_path))
             else:
                 raise Exception('Invalid partition!  Must be "train" or "validation"')

             # obtain and decode the acoustic model's predictions
             input_to_softmax.load_weights(model_path)
             prediction = input_to_softmax.predict(np.expand_dims(data_point, axis=0))
             output_length = [input_to_softmax.output_length(data_point.shape[0])]
```

```
            pred_ints = (K.eval(K.ctc_decode(
                        prediction, output_length)[0][0])+1).flatten().tolist()

            # play the audio file, and display the true and predicted transcriptions
            print('-'*80)
            Audio(audio_path)
            print('True transcription:\n' + '\n' + transcr)
            print('-'*80)
            print('Predicted transcription:\n' + '\n' + ''.join(int_sequence_to_text(pred_ints
            print('-'*80)
```

Use the code cell below to obtain the transcription predicted by your final model for the first example in the training dataset.

```
In [39]: get_predictions(index=0,
                        partition='train',
                        input_to_softmax=final_model(),
                        spectrogram=False,
                        model_path='results/model_end.h5')
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
the_input (InputLayer)       (None, None, 13)          0
_________________________________________________________________
conv1d_7 (Conv1D)            (None, None, 200)         28800
_________________________________________________________________
leaky_re_lu_8 (LeakyReLU)    (None, None, 200)         0
_________________________________________________________________
batch_normalization_8 (Batch (None, None, 200)         800
_________________________________________________________________
dropout_8 (Dropout)          (None, None, 200)         0
_________________________________________________________________
conv1d_8 (Conv1D)            (None, None, 200)         440200
_________________________________________________________________
leaky_re_lu_9 (LeakyReLU)    (None, None, 200)         0
_________________________________________________________________
batch_normalization_9 (Batch (None, None, 200)         800
_________________________________________________________________
dropout_9 (Dropout)          (None, None, 200)         0
_________________________________________________________________
conv1d_9 (Conv1D)            (None, None, 200)         440200
_________________________________________________________________
max_pooling1d_3 (MaxPooling1 (None, None, 200)         0
_________________________________________________________________
cu_dnngru_6 (CuDNNGRU)       (None, None, 200)         241200
_________________________________________________________________
leaky_re_lu_10 (LeakyReLU)   (None, None, 200)         0
```

```
-----------------------------------------------------------------
batch_normalization_10 (Batc (None, None, 200)         800
-----------------------------------------------------------------
dropout_10 (Dropout)         (None, None, 200)         0
-----------------------------------------------------------------
cu_dnngru_7 (CuDNNGRU)       (None, None, 200)         241200
-----------------------------------------------------------------
leaky_re_lu_11 (LeakyReLU)   (None, None, 200)         0
-----------------------------------------------------------------
batch_normalization_11 (Batc (None, None, 200)         800
-----------------------------------------------------------------
dropout_11 (Dropout)         (None, None, 200)         0
-----------------------------------------------------------------
bidirectional_4 (Bidirection (None, None, 400)         482400
-----------------------------------------------------------------
time_distributed_3 (TimeDist (None, None, 29)          11629
-----------------------------------------------------------------
activation_3 (Activation)    (None, None, 29)          0
=================================================================
Total params: 1,888,829
Trainable params: 1,887,229
Non-trainable params: 1,600

-----------------------------------------------------------------
None
-------------------------------------------------------------------------------------
True transcription:

the last two days of the voyage bartley found almost intolerable
-------------------------------------------------------------------------------------
Predicted transcription:

the last to days of the voige bartley found almost in tolarable
-------------------------------------------------------------------------------------
```

Use the next code cell to visualize the model's prediction for the first example in the validation dataset.

```
In [41]: get_predictions(index=0,
                         partition='validation',
                         input_to_softmax=final_model(),
                         spectrogram=False,
                         model_path='results/model_end.h5')

-----------------------------------------------------------------
Layer (type)                 Output Shape              Param #
=================================================================
the_input (InputLayer)       (None, None, 13)          0
```

```
_________________________________________________________________
conv1d_13 (Conv1D)           (None, None, 200)         28800
_________________________________________________________________
leaky_re_lu_16 (LeakyReLU)   (None, None, 200)         0
_________________________________________________________________
batch_normalization_16 (Batc (None, None, 200)         800
_________________________________________________________________
dropout_16 (Dropout)         (None, None, 200)         0
_________________________________________________________________
conv1d_14 (Conv1D)           (None, None, 200)         440200
_________________________________________________________________
leaky_re_lu_17 (LeakyReLU)   (None, None, 200)         0
_________________________________________________________________
batch_normalization_17 (Batc (None, None, 200)         800
_________________________________________________________________
dropout_17 (Dropout)         (None, None, 200)         0
_________________________________________________________________
conv1d_15 (Conv1D)           (None, None, 200)         440200
_________________________________________________________________
max_pooling1d_5 (MaxPooling1 (None, None, 200)         0
_________________________________________________________________
cu_dnngru_12 (CuDNNGRU)      (None, None, 200)         241200
_________________________________________________________________
leaky_re_lu_18 (LeakyReLU)   (None, None, 200)         0
_________________________________________________________________
batch_normalization_18 (Batc (None, None, 200)         800
_________________________________________________________________
dropout_18 (Dropout)         (None, None, 200)         0
_________________________________________________________________
cu_dnngru_13 (CuDNNGRU)      (None, None, 200)         241200
_________________________________________________________________
leaky_re_lu_19 (LeakyReLU)   (None, None, 200)         0
_________________________________________________________________
batch_normalization_19 (Batc (None, None, 200)         800
_________________________________________________________________
dropout_19 (Dropout)         (None, None, 200)         0
_________________________________________________________________
bidirectional_6 (Bidirection (None, None, 400)         482400
_________________________________________________________________
time_distributed_5 (TimeDist (None, None, 29)          11629
_________________________________________________________________
activation_5 (Activation)    (None, None, 29)          0
=================================================================
Total params: 1,888,829
Trainable params: 1,887,229
Non-trainable params: 1,600
_________________________________________________________________
None
```

```
--------------------------------------------------------------------------------
True transcription:

out in the woods stood a nice little fir tree
--------------------------------------------------------------------------------
Predicted transcription:

od in the wod stoiden nice wil firh me
--------------------------------------------------------------------------------
```

```
In [46]: get_predictions(index=1,
                         partition='validation',
                         input_to_softmax=final_model(),
                         spectrogram=False,
                         model_path='results/model_end.h5')
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
the_input (InputLayer)       (None, None, 13)          0
_________________________________________________________________
conv1d_28 (Conv1D)           (None, None, 200)         28800
_________________________________________________________________
leaky_re_lu_36 (LeakyReLU)   (None, None, 200)         0
_________________________________________________________________
batch_normalization_36 (Batc (None, None, 200)         800
_________________________________________________________________
dropout_36 (Dropout)         (None, None, 200)         0
_________________________________________________________________
conv1d_29 (Conv1D)           (None, None, 200)         440200
_________________________________________________________________
leaky_re_lu_37 (LeakyReLU)   (None, None, 200)         0
_________________________________________________________________
batch_normalization_37 (Batc (None, None, 200)         800
_________________________________________________________________
dropout_37 (Dropout)         (None, None, 200)         0
_________________________________________________________________
conv1d_30 (Conv1D)           (None, None, 200)         440200
_________________________________________________________________
max_pooling1d_10 (MaxPooling (None, None, 200)         0
_________________________________________________________________
cu_dnngru_27 (CuDNNGRU)      (None, None, 200)         241200
_________________________________________________________________
leaky_re_lu_38 (LeakyReLU)   (None, None, 200)         0
_________________________________________________________________
batch_normalization_38 (Batc (None, None, 200)         800
_________________________________________________________________
```

```
dropout_38 (Dropout)         (None, None, 200)         0
_________________________________________________________________
cu_dnngru_28 (CuDNNGRU)      (None, None, 200)         241200
_________________________________________________________________
leaky_re_lu_39 (LeakyReLU)   (None, None, 200)         0
_________________________________________________________________
batch_normalization_39 (Batc (None, None, 200)         800
_________________________________________________________________
dropout_39 (Dropout)         (None, None, 200)         0
_________________________________________________________________
bidirectional_11 (Bidirectio (None, None, 400)         482400
_________________________________________________________________
time_distributed_10 (TimeDis (None, None, 29)          11629
_________________________________________________________________
activation_10 (Activation)   (None, None, 29)          0
=================================================================
Total params: 1,888,829
Trainable params: 1,887,229
Non-trainable params: 1,600
_________________________________________________________________
None
--------------------------------------------------------------------------------
True transcription:

but this was what the tree could not bear to hear
--------------------------------------------------------------------------------
Predicted transcription:

ot fis was with the tre could not the t here
--------------------------------------------------------------------------------
```

One standard way to improve the results of the decoder is to incorporate a language model. We won't pursue this in the notebook, but you are welcome to do so as an *optional extension*.

If you are interested in creating models that provide improved transcriptions, you are encouraged to download more data and train bigger, deeper models. But beware - the model will likely take a long while to train. For instance, training this state-of-the-art model would take 3-6 weeks on a single GPU!