Artificial Intelligence Nanodegree

Voice User Interfaces

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 \n", "**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.

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:

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]:

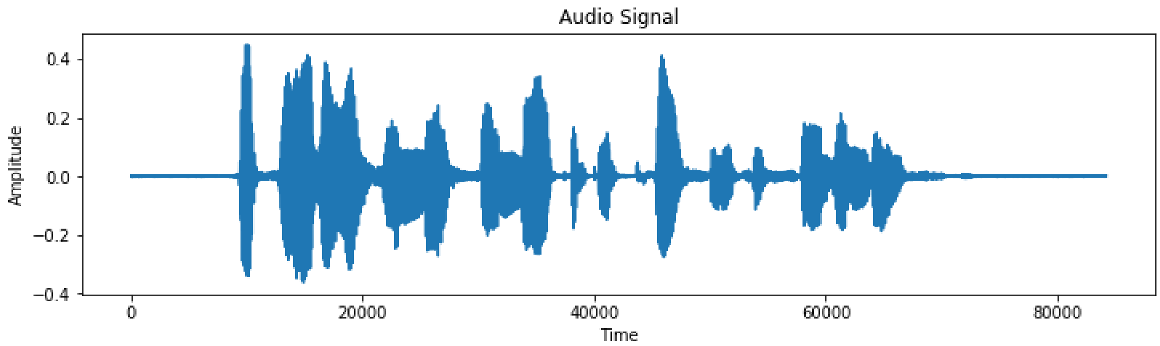
```
from data_generator import vis_train_features

# extract label and audio features for a single training example
vis_text, vis_raw_audio, vis_mfcc_feature, vis_spectrogram_feature, vis_audio_path = vis_train_feat
ures()
```

```
There are 2023 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 : (84231,)

Transcript : her father is a most remarkable person to say the least

Out[2]:

Your browser does not support the audio element.

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.

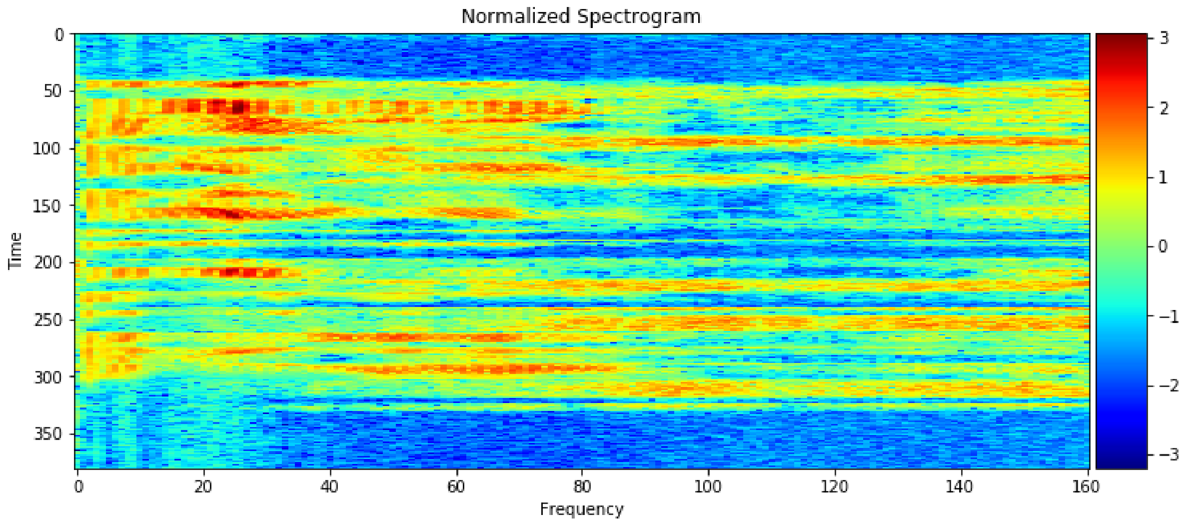
Spectrograms

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 : (381, 161)

Mel-Frequency Cepstral Coefficients (MFCCs)

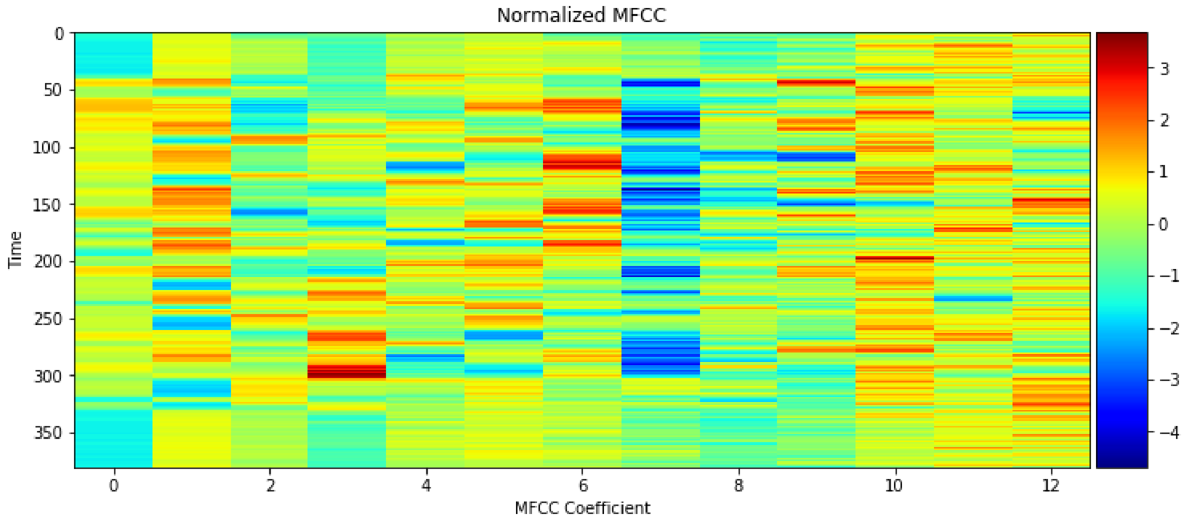
The second option for an audio feature representation is MFCCs. You do **not** need to dig deeply into the details of how MFCCs are calculated, but if you would like more information, you are welcome to peruse the documentation of the `python_speech_features` Python package. Just as with the spectrogram 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 : (381, 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 [1]:

```
#####################################################################
# 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
```

```
Using TensorFlow backend.
```

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 [7]:

```
model_0 = simple_rnn_model(input_dim=161)
```

```
_________________________________________________________________
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 [8]:

```
train_model(input_to_softmax=model_0,
            pickle_path='model_0.pickle',
            save_model_path='model_0.h5',
            minibatch_size=128,
            spectrogram=True)
```

```
Epoch 1/20
15/15 [==============================] - 69s 5s/step - loss: 1170.8660 - val_loss: 926.4780
Epoch 2/20
15/15 [==============================] - 64s 4s/step - loss: 838.8605 - val_loss: 772.3344
Epoch 3/20
15/15 [==============================] - 65s 4s/step - loss: 799.7342 - val_loss: 785.3805
Epoch 4/20
15/15 [==============================] - 65s 4s/step - loss: 793.4777 - val_loss: 767.1880
Epoch 5/20
15/15 [==============================] - 65s 4s/step - loss: 790.5130 - val_loss: 759.2594
Epoch 6/20
15/15 [==============================] - 65s 4s/step - loss: 781.3630 - val_loss: 764.6610
Epoch 7/20
15/15 [==============================] - 65s 4s/step - loss: 780.3437 - val_loss: 765.0483
Epoch 8/20
15/15 [==============================] - 66s 4s/step - loss: 780.0391 - val_loss: 763.3446
Epoch 9/20
15/15 [==============================] - 65s 4s/step - loss: 781.0038 - val_loss: 762.7876
Epoch 10/20
15/15 [==============================] - 65s 4s/step - loss: 778.5666 - val_loss: 757.8520
Epoch 11/20
15/15 [==============================] - 65s 4s/step - loss: 777.9522 - val_loss: 757.4281
Epoch 12/20
15/15 [==============================] - 65s 4s/step - loss: 780.7982 - val_loss: 756.8777
Epoch 13/20
15/15 [==============================] - 66s 4s/step - loss: 778.1554 - val_loss: 764.5061
Epoch 14/20
15/15 [==============================] - 65s 4s/step - loss: 776.0609 - val_loss: 758.5332
Epoch 15/20
15/15 [==============================] - 65s 4s/step - loss: 781.5360 - val_loss: 758.7222
Epoch 16/20
15/15 [==============================] - 64s 4s/step - loss: 775.5248 - val_loss: 756.6181
Epoch 17/20
15/15 [==============================] - 64s 4s/step - loss: 777.3770 - val_loss: 763.2734
Epoch 18/20
15/15 [==============================] - 64s 4s/step - loss: 776.9154 - val_loss: 757.8164
Epoch 19/20
15/15 [==============================] - 64s 4s/step - loss: 777.5046 - val_loss: 754.5046
Epoch 20/20
15/15 [==============================] - 63s 4s/step - loss: 776.7096 - val_loss: 755.9316
```

(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, units=200, activation='relu')
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
the_input (InputLayer)       (None, None, 161)         0         
_________________________________________________________________
rnn (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 [3]:

```
train_model(input_to_softmax=model_1,
            pickle_path='model_1.pickle',
            save_model_path='model_1.h5',
            minibatch_size=128,
            spectrogram=True)
```

```
Epoch 1/20
15/15 [==============================] - 78s 5s/step - loss: 550.5407 - val_loss: 305.6824
Epoch 2/20
15/15 [==============================] - 69s 5s/step - loss: 306.5675 - val_loss: 357.0929
Epoch 3/20
15/15 [==============================] - 70s 5s/step - loss: 282.1942 - val_loss: 481.2026
Epoch 4/20
15/15 [==============================] - 70s 5s/step - loss: 249.2612 - val_loss: 363.5119
Epoch 5/20
15/15 [==============================] - 70s 5s/step - loss: 234.1625 - val_loss: 235.3942
Epoch 6/20
15/15 [==============================] - 71s 5s/step - loss: 226.1521 - val_loss: 242.8939
Epoch 7/20
15/15 [==============================] - 71s 5s/step - loss: 218.8096 - val_loss: 222.2963
```

```
Epoch 8/20
15/15 [==============================] - 70s 5s/step - loss: 215.4314 - val_loss: 221.8104
Epoch 9/20
15/15 [==============================] - 71s 5s/step - loss: 211.5367 - val_loss: 218.5556
Epoch 10/20
15/15 [==============================] - 70s 5s/step - loss: 207.1756 - val_loss: 205.9382
Epoch 11/20
15/15 [==============================] - 70s 5s/step - loss: 203.1748 - val_loss: 208.7455
Epoch 12/20
15/15 [==============================] - 70s 5s/step - loss: 200.1258 - val_loss: 196.5948
Epoch 13/20
15/15 [==============================] - 69s 5s/step - loss: 195.5935 - val_loss: 193.7864
Epoch 14/20
15/15 [==============================] - 71s 5s/step - loss: 194.0328 - val_loss: 197.5228
Epoch 15/20
15/15 [==============================] - 70s 5s/step - loss: 189.8282 - val_loss: 187.6411
Epoch 16/20
15/15 [==============================] - 70s 5s/step - loss: 187.1169 - val_loss: 184.4520
Epoch 17/20
15/15 [==============================] - 69s 5s/step - loss: 184.5100 - val_loss: 182.1227
Epoch 18/20
15/15 [==============================] - 71s 5s/step - loss: 181.7277 - val_loss: 180.4456
Epoch 19/20
15/15 [==============================] - 70s 5s/step - loss: 178.5316 - val_loss: 175.6820
Epoch 20/20
15/15 [==============================] - 71s 5s/step - loss: 175.7741 - val_loss: 175.0228
```

Notes for Reviewer

As recommended, I have increased the GRU cells to 200. Performance have improved. Thanks!

(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](https://keras.io/layers/convolutional/#conv1d).

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 [3]:

```python
model_2 = cnn_rnn_model(input_dim=161, # change to 13 if you would like to use MFCC features
                        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
_________________________________________________________________
batch_normalization_1 (Batch (None, None, 200)         800
_________________________________________________________________
time_distributed_1 (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 [4]:

```
train_model(input_to_softmax=model_2,
            pickle_path='model_2.pickle',
            save_model_path='model_2.h5',
            minibatch_size=64,
            spectrogram=True)
```

```
Epoch 1/20
31/31 [==============================] - 43s 1s/step - loss: 302.2831 - val_loss: 331.7823
Epoch 2/20
31/31 [==============================] - 39s 1s/step - loss: 235.2240 - val_loss: 232.5838
Epoch 3/20
31/31 [==============================] - 39s 1s/step - loss: 225.2363 - val_loss: 215.2478
Epoch 4/20
31/31 [==============================] - 39s 1s/step - loss: 211.1869 - val_loss: 209.2183
Epoch 5/20
31/31 [==============================] - 39s 1s/step - loss: 193.6253 - val_loss: 198.0187
Epoch 6/20
31/31 [==============================] - 39s 1s/step - loss: 184.1736 - val_loss: 174.3401
Epoch 7/20
31/31 [==============================] - 39s 1s/step - loss: 173.3146 - val_loss: 170.4550
Epoch 8/20
31/31 [==============================] - 39s 1s/step - loss: 166.8874 - val_loss: 163.8313
Epoch 9/20
31/31 [==============================] - 39s 1s/step - loss: 160.4162 - val_loss: 155.8048
Epoch 10/20
31/31 [==============================] - 38s 1s/step - loss: 155.3215 - val_loss: 151.0986
Epoch 11/20
31/31 [==============================] - 39s 1s/step - loss: 151.1434 - val_loss: 149.6317
Epoch 12/20
31/31 [==============================] - 39s 1s/step - loss: 147.4253 - val_loss: 144.8530
Epoch 13/20
31/31 [==============================] - 39s 1s/step - loss: 143.2347 - val_loss: 142.7647
Epoch 14/20
31/31 [==============================] - 39s 1s/step - loss: 140.5943 - val_loss: 139.5369
Epoch 15/20
31/31 [==============================] - 39s 1s/step - loss: 137.1362 - val_loss: 138.1659
Epoch 16/20
31/31 [==============================] - 39s 1s/step - loss: 134.2712 - val_loss: 137.2657
Epoch 17/20
31/31 [==============================] - 38s 1s/step - loss: 132.0699 - val_loss: 135.1913
Epoch 18/20
31/31 [==============================] - 39s 1s/step - loss: 130.5735 - val_loss: 132.5744
Epoch 19/20
31/31 [==============================] - 38s 1s/step - loss: 128.2459 - val_loss: 132.1118
Epoch 20/20
31/31 [==============================] - 39s 1s/step - loss: 126.0906 - val_loss: 133.5253
```

Notes for Reviewer

As recommended,To avoid overfitting, I have implemented of a drop out of 0.3 in the model. The losses are closer now.

(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 [14]:

```
model_3 = deep_rnn_model(input_dim=161, # change to 13 if you would like to use MFCC features
                         units=200,
                         recur_layers=2)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
the_input (InputLayer)       (None, None, 161)         0         
_________________________________________________________________
rnn_0 (GRU)                  (None, None, 200)         217200    
_________________________________________________________________
batch_normalization_4 (Batch (None, None, 200)         800       
_________________________________________________________________
rnn_1 (GRU)                  (None, None, 200)         240600    
_________________________________________________________________
batch_normalization_5 (Batch (None, None, 200)         800       
_________________________________________________________________
time_distributed_4 (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 [15]:

```
train_model(input_to_softmax=model_3,
            pickle_path='model_3.pickle',
            save_model_path='model_3.h5',
            minibatch_size=128,
            spectrogram=True)
```

```
Epoch 1/20
15/15 [==============================] - 98s 7s/step - loss: 410.4804 - val_loss: 629.8792
Epoch 2/20
15/15 [==============================] - 95s 6s/step - loss: 255.5295 - val_loss: 406.0931
Epoch 3/20
15/15 [==============================] - 95s 6s/step - loss: 246.1954 - val_loss: 311.4794
Epoch 4/20
15/15 [==============================] - 96s 6s/step - loss: 238.3861 - val_loss: 389.2603
Epoch 5/20
15/15 [==============================] - 97s 6s/step - loss: 234.2477 - val_loss: 297.2931
Epoch 6/20
15/15 [==============================] - 98s 7s/step - loss: 231.8927 - val_loss: 277.7884
Epoch 7/20
15/15 [==============================] - 96s 6s/step - loss: 230.3102 - val_loss: 306.1784
Epoch 8/20
15/15 [==============================] - 96s 6s/step - loss: 226.6000 - val_loss: 254.5398
Epoch 9/20
15/15 [==============================] - 97s 6s/step - loss: 225.3247 - val_loss: 275.3212
Epoch 10/20
15/15 [==============================] - 96s 6s/step - loss: 225.2680 - val_loss: 312.7149
Epoch 11/20
15/15 [==============================] - 95s 6s/step - loss: 222.5771 - val_loss: 263.7283
Epoch 12/20
```

```
Epoch 12/20
15/15 [==============================] - 96s 6s/step - loss: 221.0419 - val_loss: 264.0083
Epoch 13/20
15/15 [==============================] - 94s 6s/step - loss: 213.0403 - val_loss: 247.9674
Epoch 14/20
15/15 [==============================] - 95s 6s/step - loss: 207.9791 - val_loss: 269.9445
Epoch 15/20
15/15 [==============================] - 95s 6s/step - loss: 200.8797 - val_loss: 207.9378
Epoch 16/20
15/15 [==============================] - 95s 6s/step - loss: 194.8085 - val_loss: 202.0403
Epoch 17/20
15/15 [==============================] - 95s 6s/step - loss: 188.8308 - val_loss: 196.4535
Epoch 18/20
15/15 [==============================] - 94s 6s/step - loss: 182.2800 - val_loss: 191.3098
Epoch 19/20
15/15 [==============================] - 94s 6s/step - loss: 175.6224 - val_loss: 186.6953
Epoch 20/20
15/15 [==============================] - 95s 6s/step - loss: 169.8018 - val_loss: 181.6143
```

(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, 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',
            minibatch_size=128,
            spectrogram=True) # change to False if you would like to use MFCC features
```

```
Epoch 1/20
```

```
15/15 [==============================] - 98s 7s/step - loss: 602.9467 - val_loss: 290.0039
Epoch 2/20
15/15 [==============================] - 95s 6s/step - loss: 280.2031 - val_loss: 256.7066
Epoch 3/20
15/15 [==============================] - 97s 6s/step - loss: 255.2691 - val_loss: 247.4606
Epoch 4/20
15/15 [==============================] - 96s 6s/step - loss: 246.3378 - val_loss: 239.4781
Epoch 5/20
15/15 [==============================] - 96s 6s/step - loss: 244.5001 - val_loss: 234.2449
Epoch 6/20
15/15 [==============================] - 96s 6s/step - loss: 240.8276 - val_loss: 234.8181
Epoch 7/20
15/15 [==============================] - 96s 6s/step - loss: 238.2385 - val_loss: 233.0005
Epoch 8/20
15/15 [==============================] - 96s 6s/step - loss: 236.3213 - val_loss: 227.1221
Epoch 9/20
15/15 [==============================] - 96s 6s/step - loss: 234.3875 - val_loss: 228.5222
Epoch 10/20
15/15 [==============================] - 96s 6s/step - loss: 230.1766 - val_loss: 226.2652
Epoch 11/20
15/15 [==============================] - 96s 6s/step - loss: 228.1078 - val_loss: 221.5178
Epoch 12/20
15/15 [==============================] - 96s 6s/step - loss: 224.5866 - val_loss: 216.3594
Epoch 13/20
15/15 [==============================] - 95s 6s/step - loss: 220.8755 - val_loss: 207.3491
Epoch 14/20
15/15 [==============================] - 95s 6s/step - loss: 216.1122 - val_loss: 209.4018
Epoch 15/20
15/15 [==============================] - 95s 6s/step - loss: 210.4078 - val_loss: 205.9368
Epoch 16/20
15/15 [==============================] - 96s 6s/step - loss: 208.0853 - val_loss: 199.3195
Epoch 17/20
15/15 [==============================] - 96s 6s/step - loss: 204.8666 - val_loss: 195.5171
Epoch 18/20
15/15 [==============================] - 95s 6s/step - loss: 201.6870 - val_loss: 195.5126
Epoch 19/20
15/15 [==============================] - 95s 6s/step - loss: 199.3690 - val_loss: 195.2551
Epoch 20/20
15/15 [==============================] - 95s 6s/step - loss: 197.5467 - val_loss: 190.1450
```

(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 [ ]:

```
## (Optional) TODO: Try out some more models!
### Feel free to use as many code cells as needed.
```

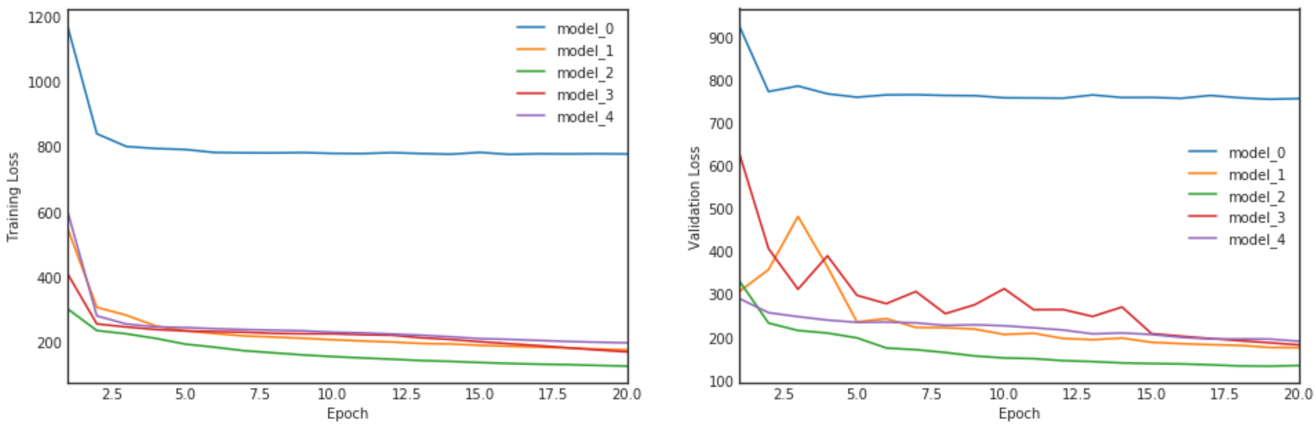
Compare the Models

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 [13]:

```
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()
```



In [15]:

```
all_pickles.pop(0)
```

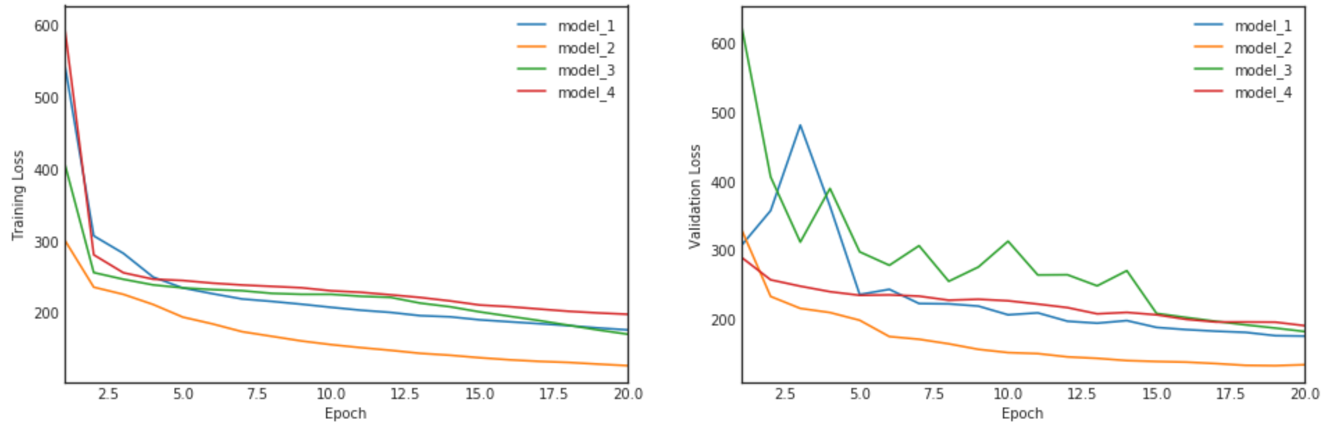
Out[15]:

```
'results/model_0.pickle'
```

In [16]:

```
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:

Ranking Architectures

1. Model 2
2. Model 1 and 3
3. Model 4 close to rank 2.
4. Model 0

Model 0

- Comments: The worst of all.
- Overfitting? : No
- How is the model created? : 29 GRU cells. Basic RNN
- Why : It is too basic to generalise the data points properly.A complex task as ASR will need deeper models for better performance.
- Takeaways to building final model : Basic RNN models can't solve the problem well

Model 1 (200 GRU cells --> 29 TimeDistributed)

- Comments : Better than Model 0
- Overfitting? : No
- How is the model created? : (200 GRU cells --> 29 TimeDistributed) (Dropout of 0.3)
- Why : The number of GRU cells have significantly increased.TimeDistributed wrapper allows to apply the dense layer on each timestep. Better retention.
- Takeaways to building final model : Use the TimeDistributed wrapper and atleast 200 GRU cells. Explore how deeper models perform. If the model is overfitting use dropouts. Also batch normalization to the recurrent layer reduces training times.

Model 2 (200 CNN -> 200 GRU -> 29 TimeDistributed)

- Comments : Lowest Training loss and Validation Loss . Best of the set.
- Overfitting? : No
- How is the model created? : (200 CNN -> 200 GRU -> 29 TimeDistributed)
- Why : The 1D temporal convolution extract features very well
- Takeaways to building final model : CNN used with RNN gave the best performance so far. Try adding more layers without

- Takeaways to building final model : CNN used with RNN gave the best performance so far. Try adding more layers without overfitting.

Model 3 (200 GRU -> 200 GRU -> 29 TimeDistributed)

- Comments : Well enough. Validation errors are pretty high intially relativly speaking, but at the end evens out as 2nd smallest loss.
- Overfitting? : Slightly
- How is the model created ? : (200 GRU -> 200 GRU -> 29 TimeDistributed)
- Why: Reccurent layers increaded , Batch Normalaisation used
- Take aways to building final model - Increasing the depth of RNN model hasn't shown much improvement in performance. (200 GRU cells --> 29 TimeDistributed) performs close enough to (200 GRU -> 200 GRU -> 29 TimeDistributed). Deeper rnn models may not be the key.

Model 4 (200 Bi-directional GRU -> 29 TimeDistributed).

- Comments: Performs not the best but good
- Overfitting? : No
- How is the model created ? : (200 Bi-directional GRU -> 29 TimeDistributed).
- Why : Compared to Model 3, it had lesser GRU
- Take aways to building final model - With the same number of GRU cells as Model 1, but by applying B- directionality we weren't able to improve the model much.

(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.

`cnn_output_length` function, if it suits your model.

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](https://keras.io/getting-started/faq/#how-can-i-save-a-keras-model) in the HDF5 file `model_end.h5`. The loss history is [saved](https://wiki.python.org/moin/UsingPickle) 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 [2]:

```
# specify the model
model_end = final_model(input_dim=13,
                        filters=200,
                        kernel_size=11,
                        conv_stride=2,
                        conv_border_mode='valid',
                        units=200,
                        activation='relu',
                        cell=GRU,
                        dropout_rate=1,
                        number_of_layers=2)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
the_input (InputLayer)       (None, None, 13)          0
_________________________________________________________________
layer_1_conv (Conv1D)        (None, None, 200)         28800
_________________________________________________________________
conv_batch_norm (BatchNormal (None, None, 200)         800
_________________________________________________________________
bidirectional_1 (Bidirection (None, None, 400)         481200
_________________________________________________________________
bt_rnn_1 (BatchNormalization (None, None, 400)         1600
_________________________________________________________________
bidirectional_2 (Bidirection (None, None, 400)         721200
_________________________________________________________________
bt_rnn_final (BatchNormaliza (None, None, 400)         1600
_________________________________________________________________
time_distributed_1 (TimeDist (None, None, 29)          11629
_________________________________________________________________
softmax (Activation)         (None, None, 29)          0
=================================================================
Total params: 1,246,829
Trainable params: 1,244,829
Non-trainable params: 2,000
_________________________________________________________________
None
```

In [3]:

```
from keras.optimizers import RMSprop
train_model(input_to_softmax=model_end,
            pickle_path='model_end.pickle',
            save_model_path='model_end.h5',
            spectrogram=False,
            epochs=10)
```

```
Epoch 1/10
101/101 [==============================] - 424s 4s/step - loss: 246.6407 - val_loss: 211.5678
Epoch 2/10
101/101 [==============================] - 417s 4s/step - loss: 182.0113 - val_loss: 179.1528
Epoch 3/10
101/101 [==============================] - 414s 4s/step - loss: 151.3001 - val_loss: 147.9998
Epoch 4/10
101/101 [==============================] - 410s 4s/step - loss: 132.3400 - val_loss: 133.6395
Epoch 5/10
101/101 [==============================] - 410s 4s/step - loss: 119.4389 - val_loss: 131.6267
Epoch 6/10
101/101 [==============================] - 413s 4s/step - loss: 109.8139 - val_loss: 130.6581
Epoch 7/10
101/101 [==============================] - 414s 4s/step - loss: 100.9434 - val_loss: 127.2910
Epoch 8/10
101/101 [==============================] - 416s 4s/step - loss: 92.8245 - val_loss: 127.3832
Epoch 9/10
```

```
101/101 [==============================] - 413s 4s/step - loss: 85.4056 - val_loss: 125.1933
Epoch 10/10
101/101 [==============================] - 410s 4s/step - loss: 78.4936 - val_loss: 128.6843
```

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

Answer:

Thoughts:

- As recommended by the reviewer, I noticed that my models were overfitting and added dropout to avoid the same of 0.5.
- As pointed out in a cell above Dilated Convolutions have been used.
- As recommended by both the reviewer and the cells above, Deep bidirectional RNN has been used.

This is the architecture I concluded after the reading and experimenting with different number of convolution layers, RNN layers, with and without dropout, changing the dilations etc. It's more of a combination of the best parts of all the trials done so far, especially from our first four models.

```
Layer (type)                Why?
```

```
layer_1_conv (Conv1D)         As seen from Model 1 above, shows a good performance.
```

```
conv_batch_norm (BatchNormal  reduce training times and avoids gradient issues as observed
from Model 1
```

```
bidirectional_1 (Bidirection  Deep RNN with Bidirectionality after exploring many different
architectures
```

```
bt_rnn_1 (BatchNormalization  reduce training times and avoids gradient issues as observed
from Model 1
```

```
bidirectional_2 (Bidirection  Deep RNN with Bidirectionality after exploring many different
architectures
```

```
bt_rnn_final (BatchNormaliza  reduce training times and avoids gradient issues as observed
from Model 1
```

```
time_distributed_1 (TimeDist TimeDistributed wrapper allows to apply the dense layer on
each timestep.
                                  Better retention of features which are complex and we want
the model to
                                  learn.When combined with CNN or GRU layers, we see from  a
ove models
                                  that it performs well.
```

```
softmax (Activation)
```

Inferenences:

The model may have overfitted a bit since we see that the validation loss values decrease to a minima and then go back up a bit. We could either reduce the epochs or add a dropout at every layer . That may avoids overfitting, but tests show this to have lesser accuracy . Maybe could be improved by tweaking some other params.

We have a fairly good performace as of now, so stopping here as far as the project is concerned.

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 [7]:

```
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):
    """ 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=False)
    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 [8]:

```
get_predictions(index=0,
                partition='train',
                input_to_softmax=model_end,
                model_path='results/model_end.h5')
```

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

her father is a most remarkable person to say the least
--------------------------------------------------------------------------------
Predicted transcription:

her fotere s om most rae markaabl kperscan t say the lea
--------------------------------------------------------------------------------
```

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

In [13]:

```
get_predictions(index=6,
                partition='validation',
                input_to_softmax=model_end,
                model_path='results/model_end.h5')
```

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

it was in fact the best weapon of its day
--------------------------------------------------------------------------------
Predicted transcription:

it was a fack te bess wept e no t staray
--------------------------------------------------------------------------------
```

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!