

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.

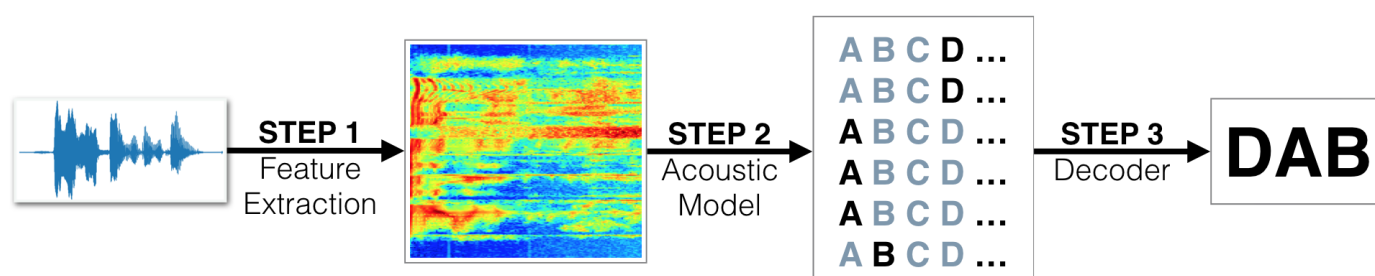
In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes 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](#)
- **STEP 1:** Acoustic Features for Speech Recognition
- **STEP 2:** Deep Neural Networks for Acoustic Modeling
 - [Model 0:](#) RNN
 - [Model 1:](#) RNN + TimeDistributed Dense
 - [Model 2:](#) CNN + RNN + TimeDistributed Dense
 - [Model 3:](#) Deeper RNN + TimeDistributed Dense
 - [Model 4:](#) Bidirectional RNN + TimeDistributed Dense
 - [Models 5+](#)
 - [Compare the Models](#)
 - [Final Model](#)
- **STEP 3:** Obtain Predictions

The Data

We begin by investigating the dataset that will be used to train and evaluate your pipeline. [LibriSpeech](http://www.danielpovey.com/files/2015_icassp_librispeech.pdf) (http://www.danielpovey.com/files/2015_icassp_librispeech.pdf) 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](http://www.openslr.org/12/) (<http://www.openslr.org/12/>).

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_features(
```

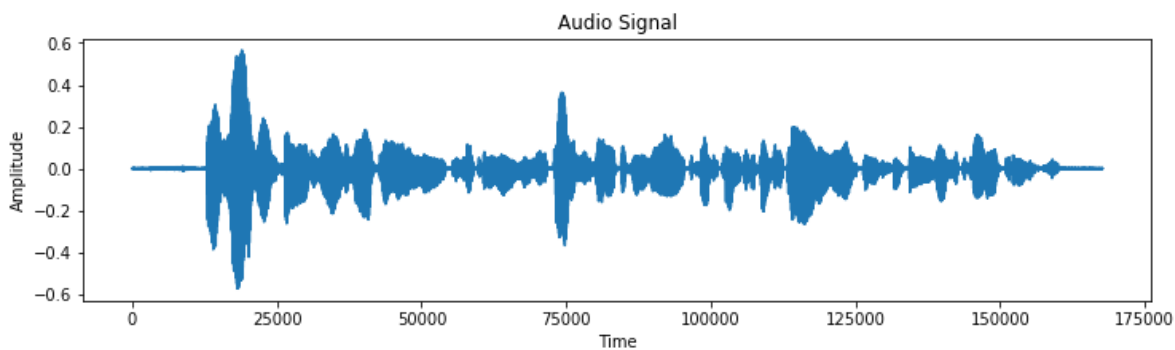
There are 2136 total training examples.

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)
```



<IPython.core.display.Markdown object>

<IPython.core.display.Markdown object>

Out[2]:



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 \(https://pdfs.semanticscholar.org/a566/cd4a8623d661a4931814d9dfc72ecbf63c4.pdf\)](https://pdfs.semanticscholar.org/a566/cd4a8623d661a4931814d9dfc72ecbf63c4.pdf).

Spectrograms

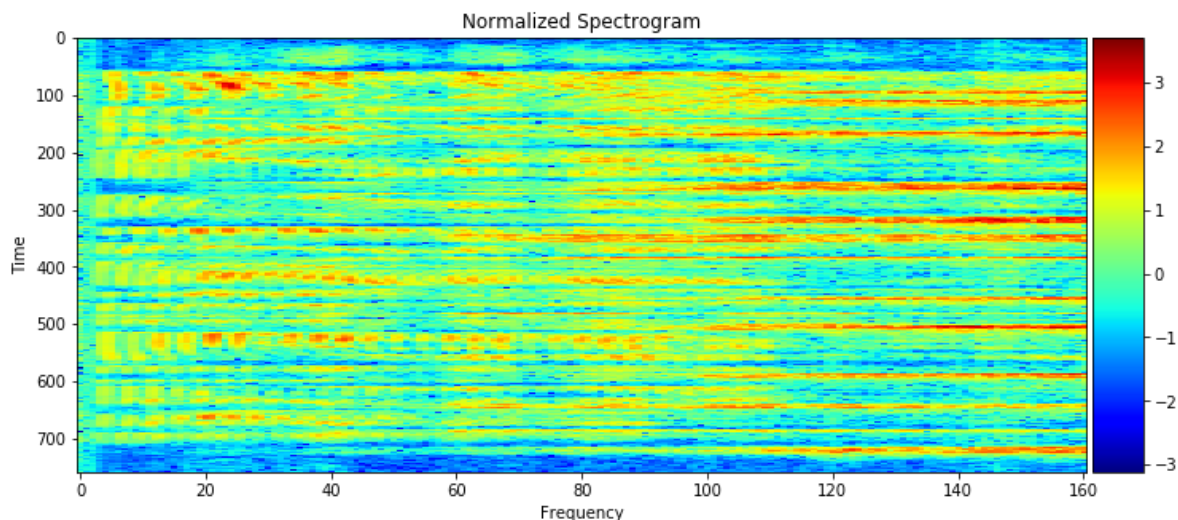
The first option for an audio feature representation is the [spectrogram \(https://www.youtube.com/watch?v=FatxGN3vAM\)](https://www.youtube.com/watch?v=FatxGN3vAM). In order to complete this project, you will **not** need to dig deeply into the details of how a spectrogram is calculated; but, if you are curious, the code for calculating the spectrogram was borrowed from [this repository \(https://github.com/baidu-research/ba-dls-deepspeech\)](https://github.com/baidu-research/ba-dls-deepspeech). 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)))
```



<IPython.core.display.Markdown object>

Mel-Frequency Cepstral Coefficients (MFCCs)

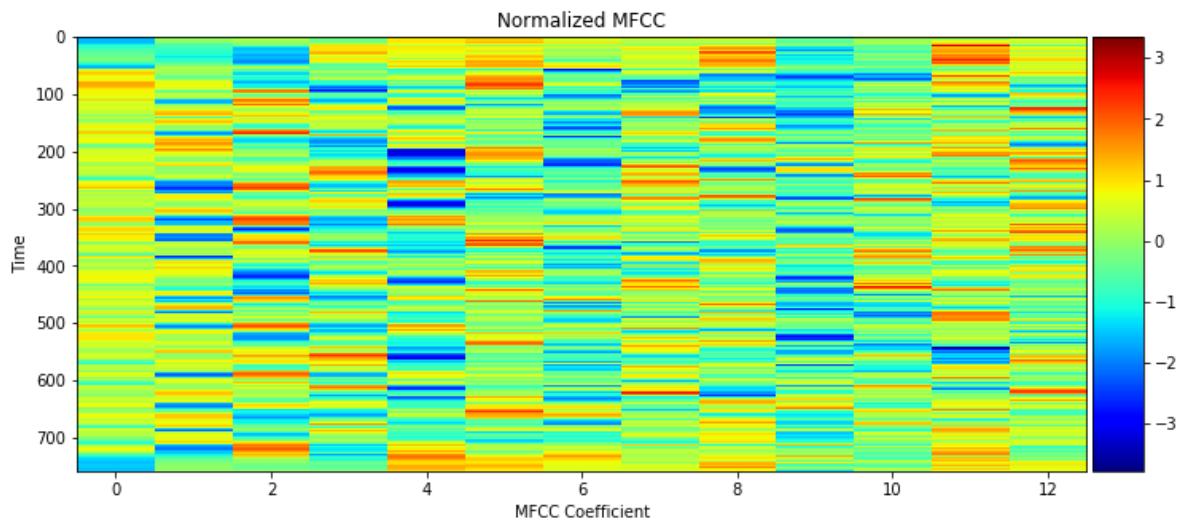
The second option for an audio feature representation is MFCCs (https://en.wikipedia.org/wiki/Mel-frequency_cepstrum). You do **not** need to dig deeply into the details of how MFCCs are calculated, but if you would like more information, you are welcome to peruse the documentation (https://github.com/jameslyons/python_speech_features) 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)))
```



<IPython.core.display.Markdown object>

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 \(https://github.com/baidu-research/ba-dls-deepspeech\)](https://github.com/baidu-research/ba-dls-deepspeech) uses spectrograms.
- This [repository \(https://github.com/mozilla/DeepSpeech\)](https://github.com/mozilla/DeepSpeech) uses MFCCs.
- This [repository \(https://github.com/buriburisuri/speech-to-text-wavenet\)](https://github.com/buriburisuri/speech-to-text-wavenet) also uses MFCCs.
- This [repository \(https://github.com/pannous/tensorflow-speech-recognition/blob/master/speech_data.py\)](https://github.com/pannous/tensorflow-speech-recognition/blob/master/speech_data.py) 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**.

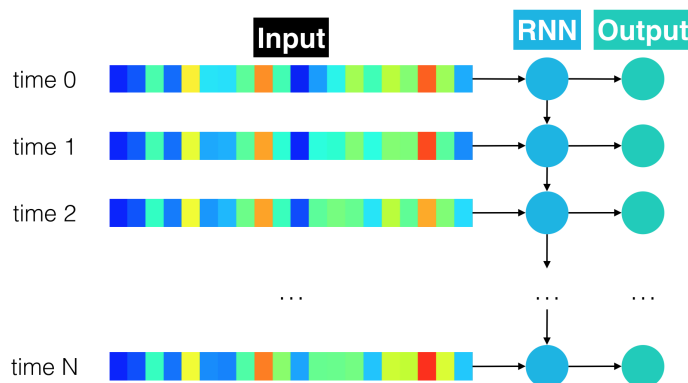
In [1]:

```
#####  
# 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.

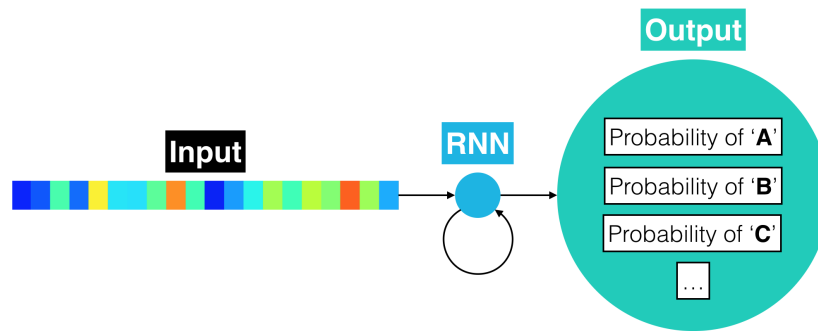
Model 0: RNN

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 [6]:

```
model_0 = simple_rnn_model(input_dim=161) # change to 13 if you would like to use MFCC features
```

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 (http://www.cs.toronto.edu/~graves/icml_2006.pdf) 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](http://www.wildml.com/2015/10/recurrent-neural-networks-tutorial-part-3-backpropagation-through-time-and-vanishing-gradients/) (<http://www.wildml.com/2015/10/recurrent-neural-networks-tutorial-part-3-backpropagation-through-time-and-vanishing-gradients/>). We have already implemented [gradient clipping](https://arxiv.org/pdf/1211.5063.pdf) (<https://arxiv.org/pdf/1211.5063.pdf>) 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 [=====] - 123s - loss: 833.7112 - val_loss: 730.717
7

Epoch 2/20

106/106 [=====] - 123s - loss: 752.4846 - val_loss: 725.628
6

Epoch 3/20

106/106 [=====] - 123s - loss: 751.3888 - val_loss: 721.714
9

Epoch 4/20

106/106 [=====] - 122s - loss: 751.0512 - val_loss: 724.766
5

Epoch 5/20

106/106 [=====] - 121s - loss: 749.7467 - val_loss: 722.204
1

Epoch 6/20

106/106 [=====] - 122s - loss: 751.1429 - val_loss: 732.212
1

Epoch 7/20

106/106 [=====] - 121s - loss: 751.3491 - val_loss: 715.403
1

Epoch 8/20

106/106 [=====] - 121s - loss: 749.7985 - val_loss: 719.208
9

Epoch 9/20

106/106 [=====] - 122s - loss: 750.0915 - val_loss: 734.033
8

Epoch 10/20

106/106 [=====] - 121s - loss: 750.6122 - val_loss: 723.016
3

Epoch 11/20

106/106 [=====] - 122s - loss: 752.1047 - val_loss: 725.314
5

Epoch 12/20

106/106 [=====] - 121s - loss: 750.5696 - val_loss: 725.562
8

Epoch 13/20

106/106 [=====] - 122s - loss: 750.0208 - val_loss: 731.471
3

Epoch 14/20

106/106 [=====] - 121s - loss: 750.7347 - val_loss: 726.922
7

Epoch 15/20

106/106 [=====] - 121s - loss: 750.6250 - val_loss: 725.945
7

Epoch 16/20

106/106 [=====] - 122s - loss: 750.9712 - val_loss: 731.742
9

Epoch 17/20

106/106 [=====] - 122s - loss: 751.7016 - val_loss: 723.413
1

Epoch 18/20

106/106 [=====] - 122s - loss: 750.5052 - val_loss: 723.950
4

Epoch 19/20

106/106 [=====] - 122s - loss: 750.8899 - val_loss: 724.722

0

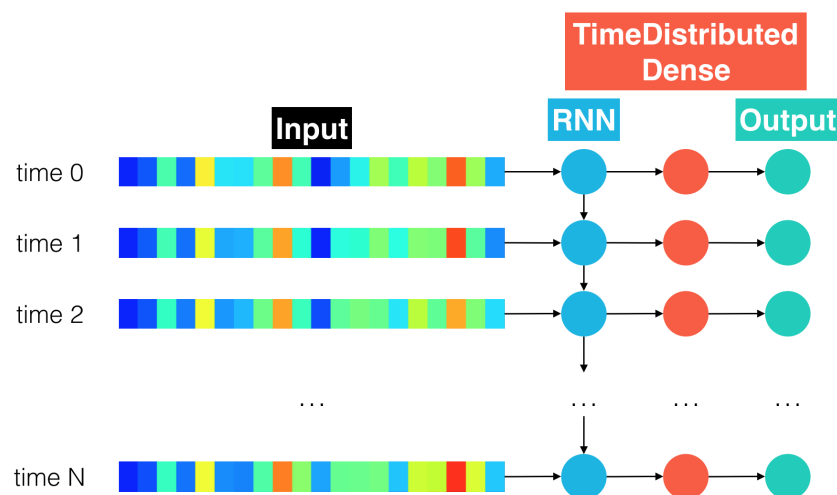
Epoch 20/20

106/106 [=====] - 122s - loss: 751.1083 - val_loss: 724.083

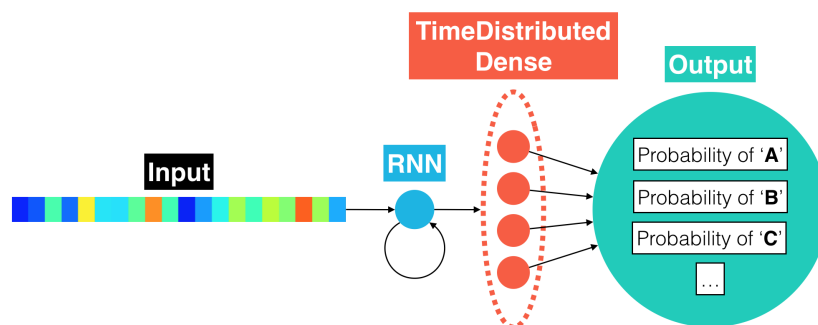
3

(IMPLEMENTATION) Model 1: RNN + TimeDistributed Dense

Read about the [TimeDistributed](https://keras.io/layers/wrappers/) (<https://keras.io/layers/wrappers/>) wrapper and the [BatchNormalization](https://keras.io/layers/normalization/) (<https://keras.io/layers/normalization/>) layer in the Keras documentation. For your next architecture, you will add [batch normalization](https://arxiv.org/pdf/1510.01378.pdf) (<https://arxiv.org/pdf/1510.01378.pdf>) 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 (TimeDistributed) 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 [8]:

```
model_1 = rnn_model(input_dim=161, # change to 13 if you would like to use MFCC features
                    units=200,
                    activation='relu')
```

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

rnn (GRU)	(None, None, 200)	217200

bn_gru (BatchNormalization)	(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 \(https://keras.io/getting-started/faq/#how-can-i-save-a-keras-model\)](https://keras.io/getting-started/faq/#how-can-i-save-a-keras-model) in the HDF5 file `model_1.h5`. The loss history is [saved \(https://wiki.python.org/moin/UsingPickle\)](https://wiki.python.org/moin/UsingPickle) 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 [9]:

```
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 features
```

Epoch 1/20

106/106 [=====] - 377s - loss: 307.3622 - val_loss: 362.883
3

Epoch 2/20

106/106 [=====] - 382s - loss: 207.4450 - val_loss: 191.121
4

Epoch 3/20

106/106 [=====] - 379s - loss: 176.8163 - val_loss: 174.522
0

Epoch 4/20

106/106 [=====] - 379s - loss: 161.0910 - val_loss: 155.061
0

Epoch 5/20

106/106 [=====] - 380s - loss: 152.3975 - val_loss: 150.740
7

Epoch 6/20

106/106 [=====] - 379s - loss: 145.1585 - val_loss: 149.539
8

Epoch 7/20

106/106 [=====] - 378s - loss: 140.3606 - val_loss: 142.131
3

Epoch 8/20

106/106 [=====] - 380s - loss: 137.1150 - val_loss: 143.629
9

Epoch 9/20

106/106 [=====] - 381s - loss: 134.6399 - val_loss: 144.103
5

Epoch 10/20

106/106 [=====] - 379s - loss: 131.3202 - val_loss: 141.078
7

Epoch 11/20

106/106 [=====] - 379s - loss: 130.1770 - val_loss: 139.180
8

Epoch 12/20

106/106 [=====] - 379s - loss: 127.9020 - val_loss: 140.884
6

Epoch 13/20

106/106 [=====] - 379s - loss: 127.8329 - val_loss: 139.083
3

Epoch 14/20

106/106 [=====] - 380s - loss: 125.6396 - val_loss: 136.280
6

Epoch 15/20

106/106 [=====] - 378s - loss: 123.2946 - val_loss: 138.345
1

Epoch 16/20

106/106 [=====] - 380s - loss: 125.0926 - val_loss: 136.036
0

Epoch 17/20

106/106 [=====] - 379s - loss: 123.3483 - val_loss: 151.266
6

Epoch 18/20

106/106 [=====] - 379s - loss: 125.9754 - val_loss: 143.405
4

Epoch 19/20

106/106 [=====] - 380s - loss: 127.5122 - val_loss: 139.801

9

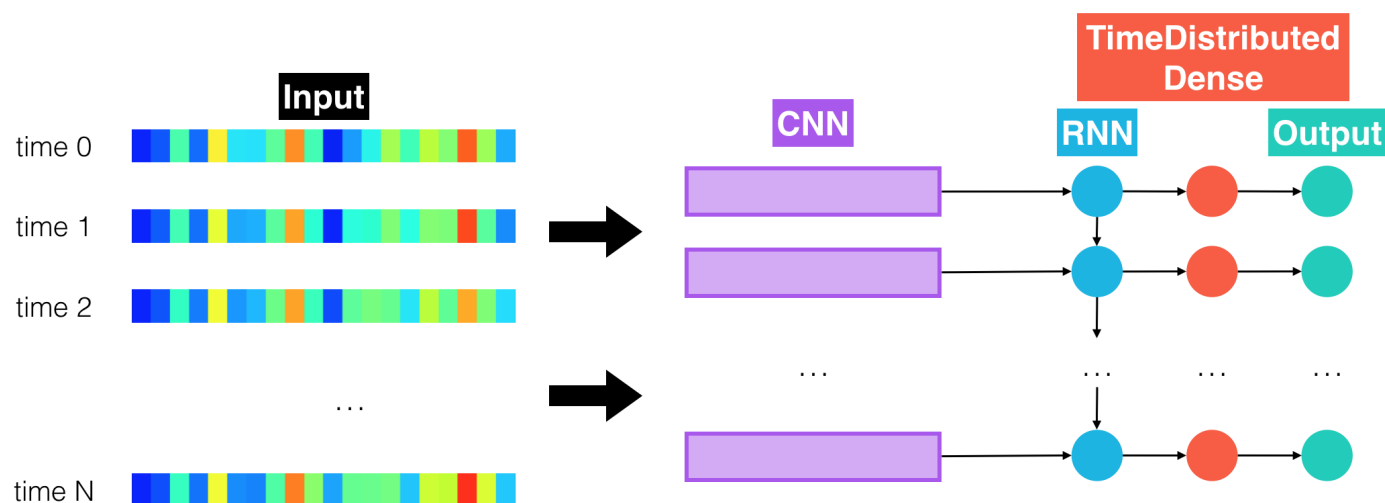
Epoch 20/20

106/106 [=====] - 380s - loss: 131.9956 - val_loss: 145.816

2

(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 [10]:

```
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
bn_rnn (BatchNormalization)	(None, None, 200)	800
time_distributed_2 (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 (<https://keras.io/getting-started/faq/#how-can-i-save-a-keras-model>) in the HDF5 file `model_2.h5`. The loss history is saved (<https://wiki.python.org/moin/UsingPickle>) 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 [11]:

```
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 features
```

Epoch 1/20

106/106 [=====] - 164s - loss: 241.6114 - val_loss: 202.286
1

Epoch 2/20

106/106 [=====] - 164s - loss: 176.0550 - val_loss: 161.712
3

Epoch 3/20

106/106 [=====] - 165s - loss: 150.3102 - val_loss: 146.709
4

Epoch 4/20

106/106 [=====] - 164s - loss: 137.3677 - val_loss: 142.147
2

Epoch 5/20

106/106 [=====] - 165s - loss: 128.6957 - val_loss: 133.575
5

Epoch 6/20

106/106 [=====] - 165s - loss: 122.7939 - val_loss: 131.778
7

Epoch 7/20

106/106 [=====] - 165s - loss: 117.0275 - val_loss: 131.342
9

Epoch 8/20

106/106 [=====] - 164s - loss: 112.3145 - val_loss: 130.241
0

Epoch 9/20

106/106 [=====] - 165s - loss: 108.5031 - val_loss: 132.744
6

Epoch 10/20

106/106 [=====] - 164s - loss: 104.9052 - val_loss: 129.103
4

Epoch 11/20

106/106 [=====] - 164s - loss: 102.2028 - val_loss: 128.130
8

Epoch 12/20

106/106 [=====] - 164s - loss: 99.9386 - val_loss: 131.0216

Epoch 13/20

106/106 [=====] - 165s - loss: 95.7430 - val_loss: 130.1022

Epoch 14/20

106/106 [=====] - 164s - loss: 93.3914 - val_loss: 133.3422

Epoch 15/20

106/106 [=====] - 163s - loss: 90.2859 - val_loss: 132.5027

Epoch 16/20

106/106 [=====] - 164s - loss: 88.0552 - val_loss: 139.4990

Epoch 17/20

106/106 [=====] - 164s - loss: 85.9179 - val_loss: 131.9384

Epoch 18/20

106/106 [=====] - 164s - loss: 83.2544 - val_loss: 133.3030

Epoch 19/20

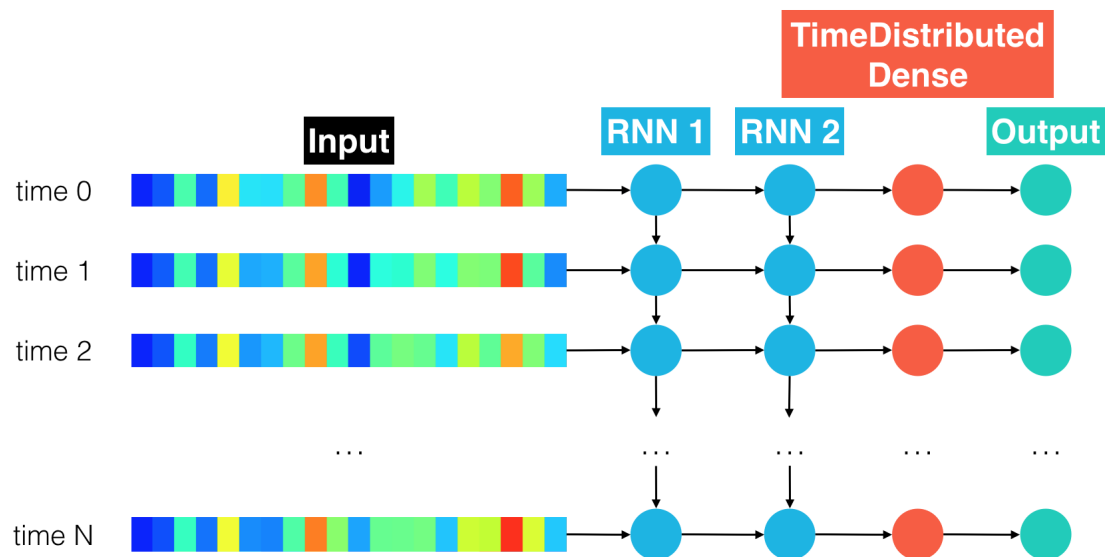
106/106 [=====] - 164s - loss: 80.6517 - val_loss: 137.7844

Epoch 20/20

106/106 [=====] - 165s - loss: 78.5715 - val_loss: 135.0023

(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 [12]:

```
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
gru_1 (GRU)	(None, None, 200)	217200
batch_normalization_1 (Batch Normalization)	(None, None, 200)	800
gru_2 (GRU)	(None, None, 200)	240600
batch_normalization_2 (Batch Normalization)	(None, None, 200)	800
time_distributed_3 (TimeDistributed Dense)	(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 (<https://keras.io/getting-started/faq/#how-can-i-save-a-keras-model>) in the HDF5 file `model_3.h5`. The loss history is saved (<https://wiki.python.org/moin/UsingPickle>) 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 [13]:

```
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 [=====] - 720s - loss: 265.3693 - val_loss: 222.822
0

Epoch 2/20

106/106 [=====] - 728s - loss: 219.0525 - val_loss: 200.234
4

Epoch 3/20

106/106 [=====] - 724s - loss: 194.3913 - val_loss: 177.546
0

Epoch 4/20

106/106 [=====] - 724s - loss: 165.8368 - val_loss: 161.895
2

Epoch 5/20

106/106 [=====] - 723s - loss: 147.7162 - val_loss: 156.771
8

Epoch 6/20

106/106 [=====] - 721s - loss: 138.1133 - val_loss: 145.423
4

Epoch 7/20

106/106 [=====] - 722s - loss: 131.4164 - val_loss: 139.634
3

Epoch 8/20

106/106 [=====] - 723s - loss: 125.7998 - val_loss: 135.832
7

Epoch 9/20

106/106 [=====] - 721s - loss: 120.8483 - val_loss: 137.019
2

Epoch 10/20

106/106 [=====] - 725s - loss: 118.1330 - val_loss: 133.109
0

Epoch 11/20

106/106 [=====] - 724s - loss: 115.3368 - val_loss: 131.960
9

Epoch 12/20

106/106 [=====] - 720s - loss: 112.2002 - val_loss: 129.648
8

Epoch 13/20

106/106 [=====] - 722s - loss: 109.7182 - val_loss: 127.946
9

Epoch 14/20

106/106 [=====] - 725s - loss: 107.5175 - val_loss: 129.734
1

Epoch 15/20

106/106 [=====] - 719s - loss: 106.3172 - val_loss: 125.733
6

Epoch 16/20

106/106 [=====] - 722s - loss: 104.5403 - val_loss: 129.228
3

Epoch 17/20

106/106 [=====] - 726s - loss: 102.9287 - val_loss: 127.136
5

Epoch 18/20

106/106 [=====] - 726s - loss: 101.7947 - val_loss: 126.966
6

Epoch 19/20

106/106 [=====] - 725s - loss: 98.9256 - val_loss: 124.8222

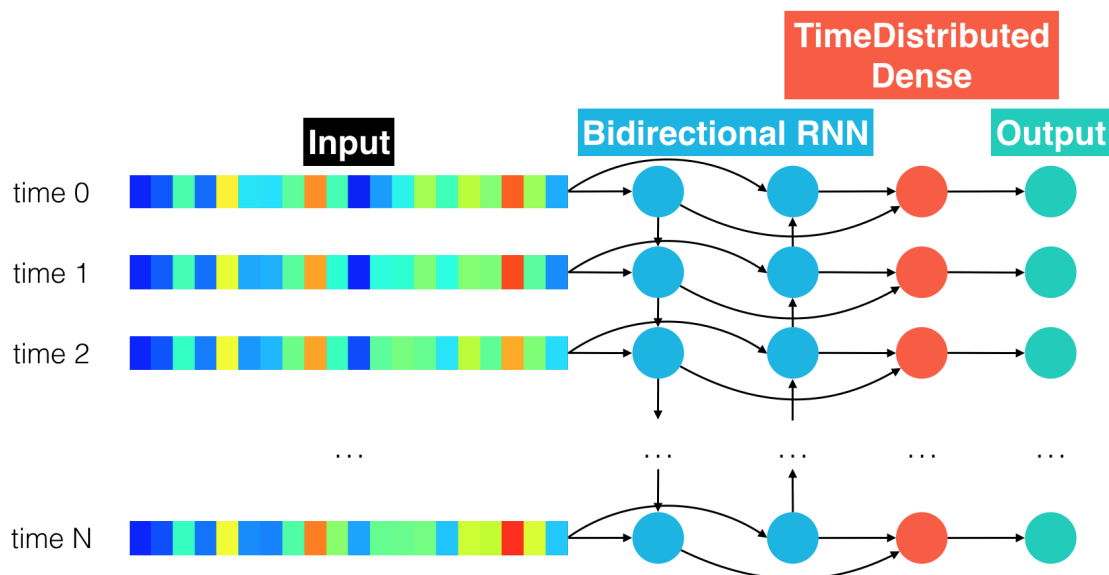
Epoch 20/20

106/106 [=====] - 725s - loss: 97.4351 - val_loss: 123.7975

(IMPLEMENTATION) Model 4: Bidirectional RNN + TimeDistributed Dense

Read about the [Bidirectional](https://keras.io/layers/wrappers/) (<https://keras.io/layers/wrappers/>) 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](http://www.cs.toronto.edu/~hinton/absps/DRNN_speech.pdf) (http://www.cs.toronto.edu/~hinton/absps/DRNN_speech.pdf).

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 [2]:

```
model_4 = bidirectional_rnn_model(input_dim=161, # change to 13 if you would like to use MFCC features
                                  units=200)
```

Layer (type)	Output Shape	Param #
the_input (InputLayer)	(None, None, 161)	0
bidirectional_1 (Bidirection	(None, None, 400)	434400
time_distributed_1 (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 \(https://keras.io/getting-started/faq/#how-can-i-save-a-keras-model\)](https://keras.io/getting-started/faq/#how-can-i-save-a-keras-model) in the HDF5 file `model_4.h5`. The loss history is [saved \(https://wiki.python.org/moin/UsingPickle\)](https://wiki.python.org/moin/UsingPickle) 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 [3]:

```
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 [=====] - 564s - loss: 284.6778 - val_loss: 231.8719

Epoch 2/20

106/106 [=====] - 569s - loss: 233.8033 - val_loss: 217.9033

Epoch 3/20

106/106 [=====] - 566s - loss: 225.6503 - val_loss: 211.6079

Epoch 4/20

106/106 [=====] - 565s - loss: 220.6682 - val_loss: 209.3788

Epoch 5/20

106/106 [=====] - 566s - loss: 207.9896 - val_loss: 197.2673

Epoch 6/20

106/106 [=====] - 563s - loss: 195.1585 - val_loss: 182.8128

Epoch 7/20

106/106 [=====] - 567s - loss: 184.0601 - val_loss: 176.9658

Epoch 8/20

106/106 [=====] - 564s - loss: 175.3509 - val_loss: 170.2264

Epoch 9/20

106/106 [=====] - 563s - loss: 168.5944 - val_loss: 165.2472

Epoch 10/20

106/106 [=====] - 565s - loss: 162.5265 - val_loss: 162.2642

Epoch 11/20

106/106 [=====] - 566s - loss: 157.1032 - val_loss: 160.1764

Epoch 12/20

106/106 [=====] - 566s - loss: 152.0662 - val_loss: 153.9161

Epoch 13/20

106/106 [=====] - 561s - loss: 146.7407 - val_loss: 150.6598

Epoch 14/20

106/106 [=====] - 562s - loss: 141.8075 - val_loss: 149.3377

Epoch 15/20

106/106 [=====] - 566s - loss: 137.4366 - val_loss: 145.5363

Epoch 16/20

106/106 [=====] - 564s - loss: 133.4242 - val_loss: 143.6832

Epoch 17/20

106/106 [=====] - 563s - loss: 129.4265 - val_loss: 141.3488

Epoch 18/20

106/106 [=====] - 564s - loss: 125.9808 - val_loss: 138.2864

Epoch 19/20

106/106 [=====] - 564s - loss: 122.4934 - val_loss: 137.7608

Epoch 20/20

106/106 [=====] - 564s - loss: 119.5276 - val_loss: 138.5695

(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 [4]:

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

model_5 = model_number5()

Layer (type)	Output Shape	Param #
the_input (InputLayer)	(None, None, 161)	0
bidirectional_2 (Bidirection	(None, None, 600)	831600
bidirectional_3 (Bidirection	(None, None, 600)	1621800
time_distributed_2 (TimeDist	(None, None, 29)	17429
softmax (Activation)	(None, None, 29)	0

Total params: 2,470,829
Trainable params: 2,470,829
Non-trainable params: 0

None

In [5]:

```
train_model(input_to_softmax=model_5,
            pickle_path='model_number5.pickle',
            save_model_path='model_number5.h5',
            spectrogram=True, # change to False if you would like to use MFCC features
            epochs=20)
```

Epoch 1/20

106/106 [=====] - 2292s - loss: 256.4023 - val_loss: 212.0587

Epoch 2/20

106/106 [=====] - 2317s - loss: 206.7268 - val_loss: 191.4868

Epoch 3/20

106/106 [=====] - 2319s - loss: 186.6125 - val_loss: 178.0895

Epoch 4/20

106/106 [=====] - 2307s - loss: 169.0169 - val_loss: 159.7341

Epoch 5/20

106/106 [=====] - 2314s - loss: 154.6515 - val_loss: 150.7287

Epoch 6/20

106/106 [=====] - 2311s - loss: 141.3910 - val_loss: 138.1920

Epoch 7/20

106/106 [=====] - 2298s - loss: 131.9008 - val_loss: 132.3804

Epoch 8/20

106/106 [=====] - 2308s - loss: 123.6825 - val_loss: 130.0969

Epoch 9/20

106/106 [=====] - 2317s - loss: 116.7140 - val_loss: 126.7146

Epoch 10/20

106/106 [=====] - 2314s - loss: 110.5800 - val_loss: 124.5112

Epoch 11/20

106/106 [=====] - 2321s - loss: 104.8673 - val_loss: 120.8287

Epoch 12/20

106/106 [=====] - 2321s - loss: 99.7867 - val_loss: 115.8781

Epoch 13/20

106/106 [=====] - 2312s - loss: 94.8191 - val_loss: 114.9650

Epoch 14/20

106/106 [=====] - 2310s - loss: 90.4332 - val_loss: 116.6450

Epoch 15/20

106/106 [=====] - 2307s - loss: 85.9304 - val_loss: 113.5043

Epoch 16/20

106/106 [=====] - 2292s - loss: 81.4097 - val_loss: 113.4503

Epoch 17/20

106/106 [=====] - 2304s - loss: 77.0592 - val_loss: 113.9277

Epoch 18/20

106/106 [=====] - 2298s - loss: 73.3/10 - val_loss: 118.104

8
Epoch 19/20

106/106 [=====] - 2312s - loss: 68.8833 - val_loss: 116.065

7
Epoch 20/20

106/106 [=====] - 2306s - loss: 65.2052 - val_loss: 116.856

2

In [6]:

model_6 = model_number6()

Layer (type)	Output Shape	Param #
the_input (InputLayer)	(None, None, 13)	0
bidirectional_4 (Bidirection	(None, None, 600)	565200
bidirectional_5 (Bidirection	(None, None, 600)	1621800
time_distributed_3 (TimeDist	(None, None, 29)	17429
softmax (Activation)	(None, None, 29)	0
Total params: 2,204,429		
Trainable params: 2,204,429		
Non-trainable params: 0		

None

In [7]:

```
train_model(input_to_softmax=model_6,  
            pickle_path='model_number6.pickle',  
            save_model_path='model_number6.h5',  
            spectrogram=False) # change to False if you would like to use MFCC features
```

```
Epoch 1/20  
106/106 [=====] - 2218s - loss: 268.0043 - val_loss: 208.28  
48  
Epoch 2/20  
106/106 [=====] - 2233s - loss: 212.0339 - val_loss: 196.53  
34  
Epoch 3/20  
106/106 [=====] - 2234s - loss: 202.3025 - val_loss: 183.78  
77  
Epoch 4/20  
106/106 [=====] - 2243s - loss: 193.9916 - val_loss: 177.39  
26  
Epoch 5/20  
106/106 [=====] - 2239s - loss: 186.9666 - val_loss: 168.34  
59  
Epoch 6/20  
106/106 [=====] - 2242s - loss: 180.3647 - val_loss: 163.04  
58  
Epoch 7/20  
106/106 [=====] - 2246s - loss: 174.2585 - val_loss: 158.44  
78  
Epoch 8/20  
106/106 [=====] - 2248s - loss: 168.3173 - val_loss: 153.89  
50  
Epoch 9/20  
106/106 [=====] - 2274s - loss: 162.8051 - val_loss: 146.09  
61  
Epoch 10/20  
106/106 [=====] - 2266s - loss: 157.2508 - val_loss: 140.11  
59  
Epoch 11/20  
106/106 [=====] - 2265s - loss: 151.8950 - val_loss: 135.75  
03  
Epoch 12/20  
106/106 [=====] - 2268s - loss: 147.8918 - val_loss: 131.47  
33  
Epoch 13/20  
106/106 [=====] - 2264s - loss: 143.7952 - val_loss: 128.27  
45  
Epoch 14/20  
106/106 [=====] - 2227s - loss: 140.9402 - val_loss: 127.41  
40  
Epoch 15/20  
106/106 [=====] - 2240s - loss: 137.7350 - val_loss: 124.54  
92  
Epoch 16/20  
106/106 [=====] - 2236s - loss: 135.0707 - val_loss: 118.88  
66  
Epoch 17/20  
106/106 [=====] - 2239s - loss: 132.8113 - val_loss: 118.69  
08  
Epoch 18/20  
106/106 [=====] - 2244s - loss: 130.5031 - val_loss: 114.84  
~
```

2/

Epoch 19/20

106/106 [=====] - 2238s - loss: 128.1432 - val_loss: 115.1679

Epoch 20/20

106/106 [=====] - 2239s - loss: 126.5853 - val_loss: 114.1502

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 [4]:

```
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))]

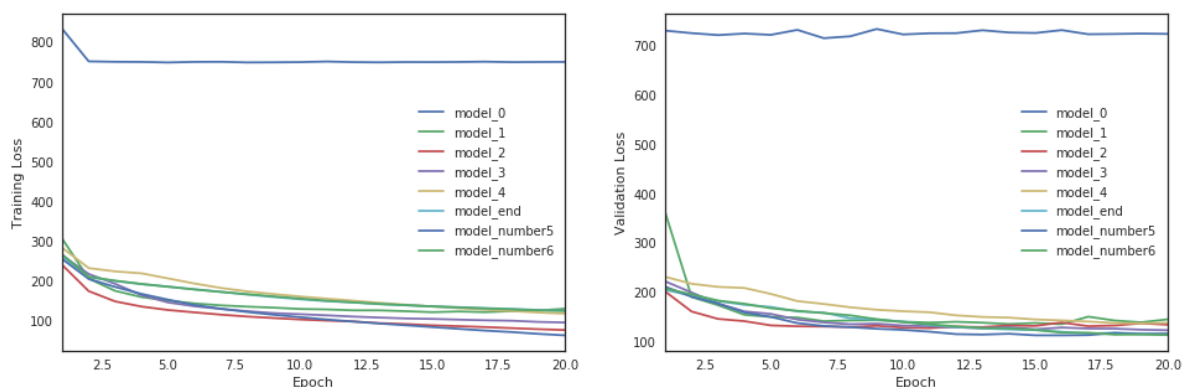
fig = plt.figure(figsize=(16,5))

# 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()
```

/home/aind2/anaconda3/envs/aind-vui/lib/python3.5/site-packages/matplotlib/font_manager.py:1316: UserWarning: findfont: Font family ['sans-serif'] not found. Falling back to DejaVu Sans

(prop.get_family(), self.defaultFamily[fonttext]))



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: I think the model 6 is the best. Actually, model 5 and model 6 show the similar result from this experiment. but model 5 use spectrogram and it seems to suffer from overfitting. model 6 is using MFCC and its validation loss is getting down with 20 epochs. If I run it more than 20 epochs, It will give us the better result I guess.

(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 (<https://faroit.github.io/keras-docs/1.0.2/layers/recurrent/>), pay special attention to the `dropout_W` and `dropout_U` arguments. This paper (<http://arxiv.org/abs/1512.05287>) 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 (<https://arxiv.org/abs/1609.03499>). For an example of a speech-to-text system that makes use of dilated convolutions, check out this GitHub repository (<https://github.com/buriburisuri/speech-to-text-wavenet>). You can work with dilated convolutions in Keras (<https://keras.io/layers/convolutional/>) 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 (<https://arxiv.org/pdf/1701.02720.pdf>) 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 (https://www.cs.toronto.edu/~graves/asru_2013.pdf)!

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 [2]:

```
# specify the model
model_end = final_model()
```

Layer (type)	Output Shape	Param #
the_input (InputLayer)	(None, None, 13)	0
bidirectional_1 (Bidirection	(None, None, 600)	565200
bidirectional_2 (Bidirection	(None, None, 600)	1621800
time_distributed_1 (TimeDist	(None, None, 29)	17429
softmax (Activation)	(None, None, 29)	0
Total params: 2,204,429		
Trainable params: 2,204,429		
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 (<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 [3]:

```
train_model(input_to_softmax=model_end,
            pickle_path='model_end.pickle',
            save_model_path='model_end.h5',
            spectrogram=False,
            epochs=20) # change to False if you would like to use MFCC features
```

Epoch 1/20

106/106 [=====] - 801s - loss: 263.7715 - val_loss: 205.549
0

Epoch 2/20

106/106 [=====] - 804s - loss: 211.2218 - val_loss: 193.855
4

Epoch 3/20

106/106 [=====] - 806s - loss: 201.3234 - val_loss: 182.749
4

Epoch 4/20

106/106 [=====] - 804s - loss: 193.7344 - val_loss: 175.345
7

Epoch 5/20

106/106 [=====] - 797s - loss: 187.5301 - val_loss: 170.914
0

Epoch 6/20

106/106 [=====] - 801s - loss: 180.6526 - val_loss: 161.840
0

Epoch 7/20

106/106 [=====] - 792s - loss: 174.0384 - val_loss: 159.477
8

Epoch 8/20

106/106 [=====] - 803s - loss: 167.5589 - val_loss: 148.000
1

Epoch 9/20

106/106 [=====] - 805s - loss: 161.2548 - val_loss: 145.486
4

Epoch 10/20

106/106 [=====] - 804s - loss: 155.7316 - val_loss: 141.273
6

Epoch 11/20

106/106 [=====] - 803s - loss: 151.0142 - val_loss: 134.366
7

Epoch 12/20

106/106 [=====] - 805s - loss: 147.7436 - val_loss: 130.098
1

Epoch 13/20

106/106 [=====] - 803s - loss: 143.6440 - val_loss: 127.500
1

Epoch 14/20

106/106 [=====] - 803s - loss: 140.5674 - val_loss: 125.407
7

Epoch 15/20

106/106 [=====] - 799s - loss: 137.9723 - val_loss: 124.176
6

Epoch 16/20

106/106 [=====] - 790s - loss: 135.8021 - val_loss: 120.597
8

Epoch 17/20

106/106 [=====] - 801s - loss: 133.3755 - val_loss: 117.071
4

Epoch 18/20

106/106 [=====] - 805s - loss: 131.4642 - val_loss: 118.622
.

1

Epoch 19/20

106/106 [=====] - 800s - loss: 128.7345 - val_loss: 116.856

7

Epoch 20/20

106/106 [=====] - 798s - loss: 126.5120 - val_loss: 115.698

7

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

Answer:

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 [8]:

```
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) # spectrogram argument must be given because mfcc
    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' + 'Wn' + transcr)
    print('-'*80)
    print('Predicted transcription:\n' + 'Wn' + ''.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 [12]:

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

Layer (type)	Output Shape	Param #
the_input (InputLayer)	(None, None, 13)	0
bidirectional_13 (Bidirectio	(None, None, 600)	565200
bidirectional_14 (Bidirectio	(None, None, 600)	1621800
time_distributed_7 (TimeDist	(None, None, 29)	17429
softmax (Activation)	(None, None, 29)	0
Total params: 2,204,429		
Trainable params: 2,204,429		
Non-trainable params: 0		

None

True transcription:

the houses seemed miserable in the extreme especially to an eye accustomed to the smiling neatness of english cottages

Predicted transcription:

the has asin do mes erobon the ixtran mi spasly ton ni casten to the smilin ne nesot hin los coitans

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

In [10]:

```
get_predictions(index=0,
                 partition='validation',
                 input_to_softmax=final_model(),
                 model_path='model_end.h5')
```

Layer (type)	Output Shape	Param #
the_input (InputLayer)	(None, None, 13)	0
bidirectional_9 (Bidirection	(None, None, 600)	565200
bidirectional_10 (Bidirectio	(None, None, 600)	1621800
time_distributed_5 (TimeDist	(None, None, 29)	17429
softmax (Activation)	(None, None, 29)	0
Total params: 2,204,429		
Trainable params: 2,204,429		
Non-trainable params: 0		

None

True transcription:

the bogus legislature numbered thirty six members

Predicted transcription:

the bo as lidislater novert tarti six mevr

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 \(http://www.openslr.org/12/\)](http://www.openslr.org/12/) 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 \(https://arxiv.org/pdf/1512.02595v1.pdf\)](https://arxiv.org/pdf/1512.02595v1.pdf) model would take 3-6 weeks on a single GPU!