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) 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/).

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]:

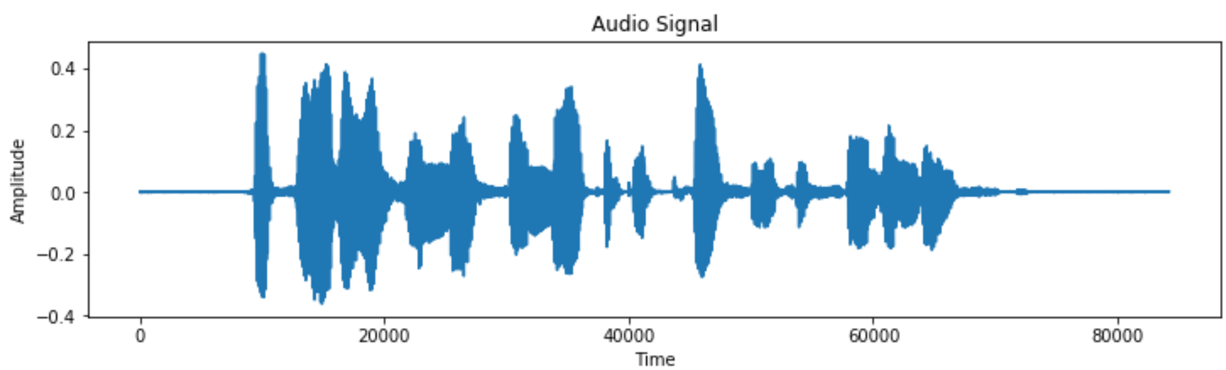
```
from data_generator import vis_train_features

# extract label and audio features for a single training example
vis_text, vis_raw_audio, vis_mfcc_feature, vis_spectrogram_feature, vi
s_audio_path = vis_train_features()
```

```
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.sh
ape)))
# 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]: 0:00 -0:03

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.

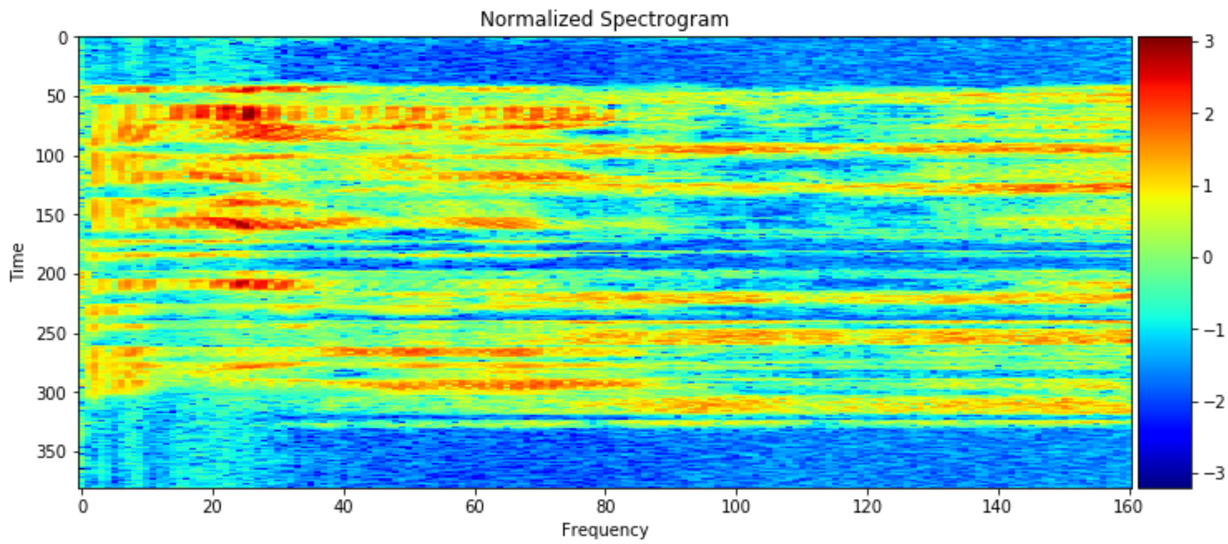
In this project, you will explore two possible feature representations. *After completing the project*, if you'd like to read more about deep learning architectures that can accept raw audio input, you are encouraged to explore this research paper (https://pdfs.semanticscholar.org/a566/cd4a8623d661a4931814d9dffc72ecbf63c4.pdf).

Spectrograms

The first option for an audio feature representation is the spectrogram (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). 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_f
eature.shape)))
```



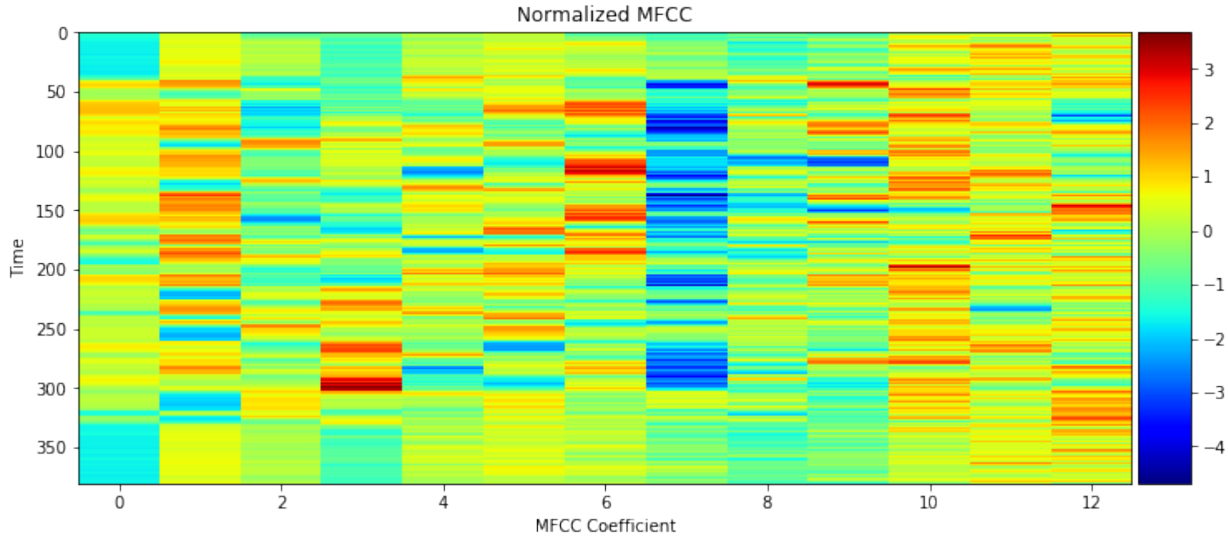
Shape of Spectrogram : (381, 161)

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))
)
```



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 (https://github.com/baidu-research/ba-dls-deepspeech) uses spectrograms.
- This repository (https://github.com/mozilla/DeepSpeech) uses MFCCs.
- This repository (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) 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.

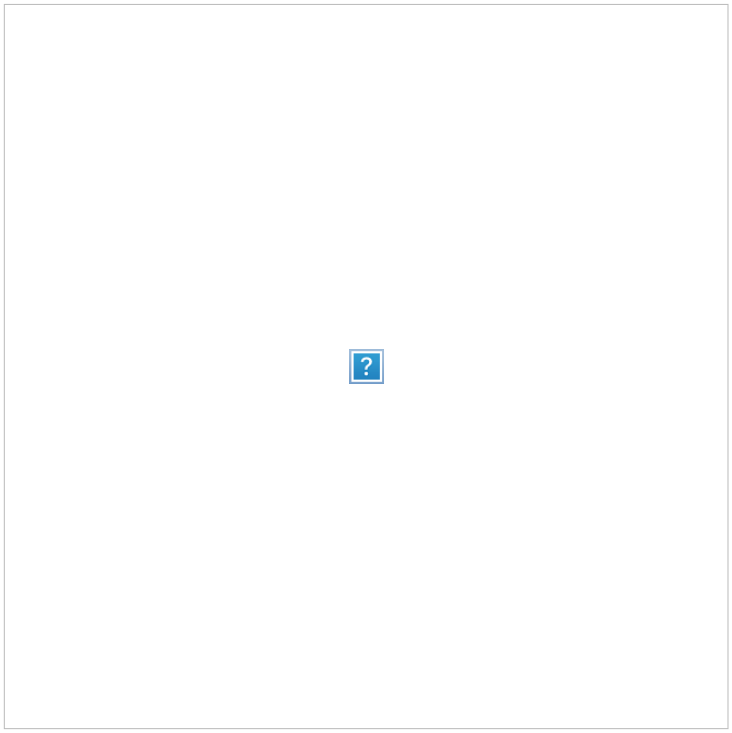
For your convenience, we have designed the notebook so that each model can be specified and trained on separate occasions. That is, say you decide to take a break from the notebook after training **Model 1**. Then, you need not re-execute all prior code cells in the notebook before training **Model 2**. 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 [5]:
```
#################################################################
# 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 aut
omatically
%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.
```

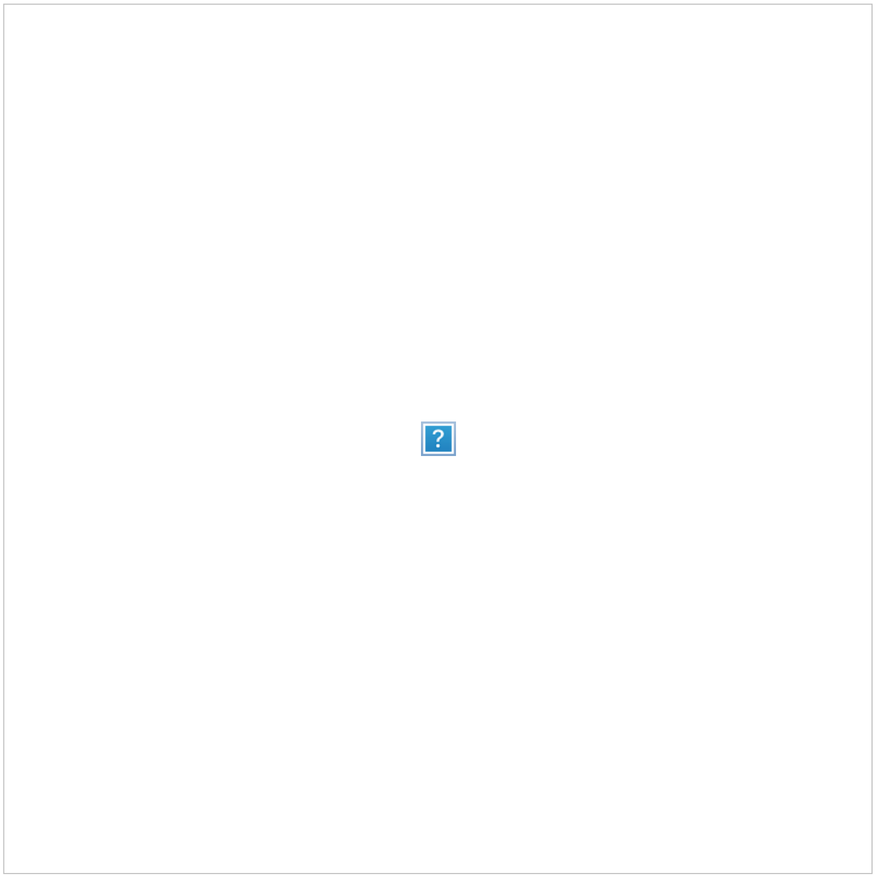
Model 0: RNN

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=13) # change to 13 if you would like to use MFCC features, otherwise 161
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
the_input (InputLayer)       (None, None, 13)          0         
_________________________________________________________________
rnn (GRU)                    (None, None, 29)          3741      
_________________________________________________________________
softmax (Activation)         (None, None, 29)          0         
=================================================================
Total params: 3,741
Trainable params: 3,741
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)](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/). We have already implemented gradient clipping (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.

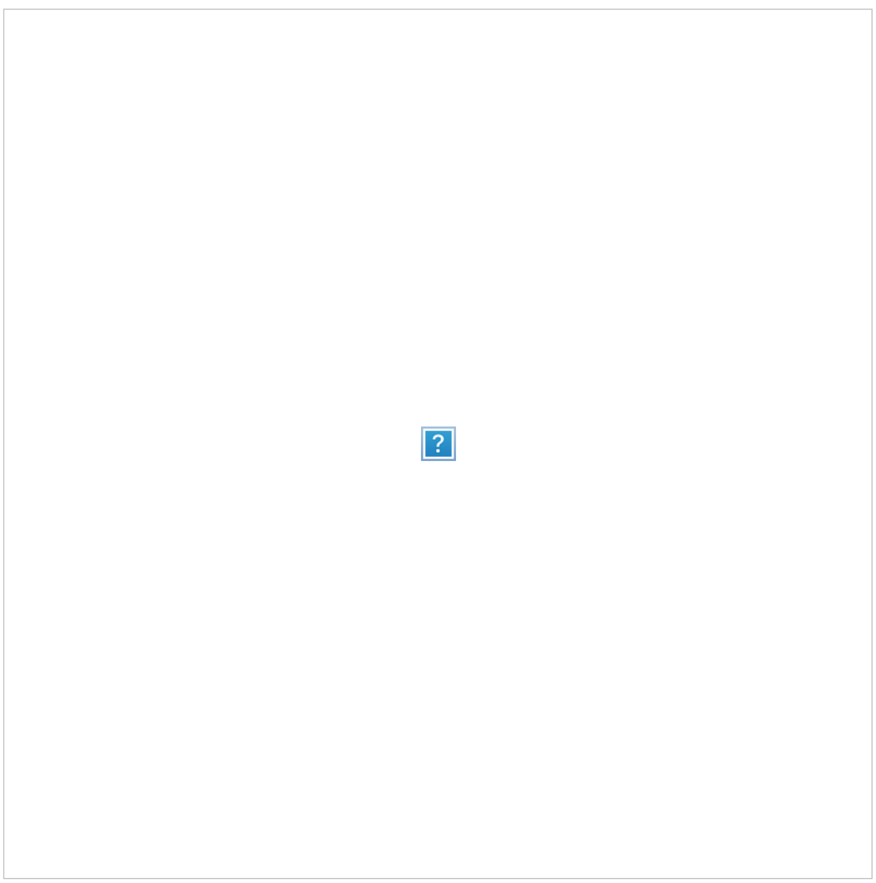
In [7]:
```
train_model(input_to_softmax=model_0,
            pickle_path='model_0.pickle',
            save_model_path='model_0.h5',
            spectrogram=False) # change to False if you would like to
use MFCC features
```

```
Epoch 1/20
101/101 [==============================] - 295s 3s/step - loss: 843.
7738 - val_loss: 756.6786
Epoch 2/20
101/101 [==============================] - 287s 3s/step - loss: 779.
3077 - val_loss: 753.1026
Epoch 3/20
101/101 [==============================] - 287s 3s/step - loss: 779.
8439 - val_loss: 752.2217
Epoch 4/20
101/101 [==============================] - 289s 3s/step - loss: 779.
6084 - val_loss: 759.6686
Epoch 5/20
101/101 [==============================] - 289s 3s/step - loss: 779.
2337 - val_loss: 758.8566
Epoch 6/20
101/101 [==============================] - 285s 3s/step - loss: 779.
6040 - val_loss: 754.1638
Epoch 7/20
101/101 [==============================] - 291s 3s/step - loss: 779.
1442 - val_loss: 759.8646
Epoch 8/20
101/101 [==============================] - 291s 3s/step - loss: 779.
2358 - val_loss: 761.1457
Epoch 9/20
101/101 [==============================] - 293s 3s/step - loss: 779.
0243 - val_loss: 756.0585
Epoch 10/20
101/101 [==============================] - 289s 3s/step - loss: 779.
3046 - val_loss: 758.9146
Epoch 11/20
101/101 [==============================] - 291s 3s/step - loss: 779.
0603 - val_loss: 757.4561
Epoch 12/20
101/101 [==============================] - 288s 3s/step - loss: 778.
6907 - val_loss: 756.6310
Epoch 13/20
101/101 [==============================] - 284s 3s/step - loss: 779.
3921 - val_loss: 758.1296
Epoch 14/20
101/101 [==============================] - 281s 3s/step - loss: 779.
5428 - val_loss: 755.7209
Epoch 15/20
101/101 [==============================] - 285s 3s/step - loss: 779.
0354 - val_loss: 754.8579
Epoch 16/20
101/101 [==============================] - 288s 3s/step - loss: 778.
```

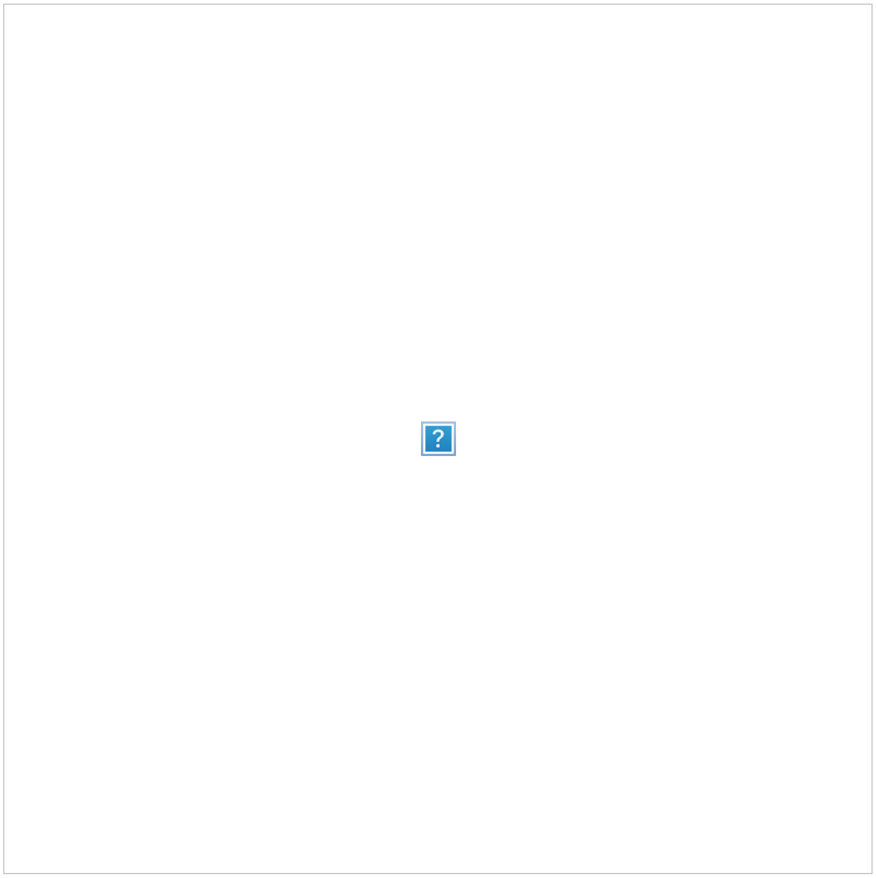
```
7810 - val_loss: 756.9104
Epoch 17/20
101/101 [==============================] - 289s 3s/step - loss: 779.
1732 - val_loss: 751.2092
Epoch 18/20
101/101 [==============================] - 283s 3s/step - loss: 778.
9381 - val_loss: 759.9166
Epoch 19/20
101/101 [==============================] - 283s 3s/step - loss: 779.
1585 - val_loss: 759.6989
Epoch 20/20
101/101 [==============================] - 290s 3s/step - loss: 779.
2482 - val_loss: 758.9405
```

(IMPLEMENTATION) Model 1: RNN + TimeDistributed Dense

Read about the TimeDistributed (https://keras.io/layers/wrappers/) wrapper and the BatchNormalization (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) 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 [6]:
```
from sample_models import *
model_1 = rnn_model(input_dim=13, # change to 13 if you would like to use MFCC features, input_dim=161
                    units=200,
                    activation='relu')
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
the_input (InputLayer)       (None, None, 13)          0         
_________________________________________________________________
rnn (GRU)                    (None, None, 200)         128400    
_________________________________________________________________
batch_normalization_1 (Batch (None, None, 200)         800       
_________________________________________________________________
time_distributed_1 (TimeDist (None, None, 29)          5829      
_________________________________________________________________
softmax (Activation)         (None, None, 29)          0         
=================================================================
Total params: 135,029
Trainable params: 134,629
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.

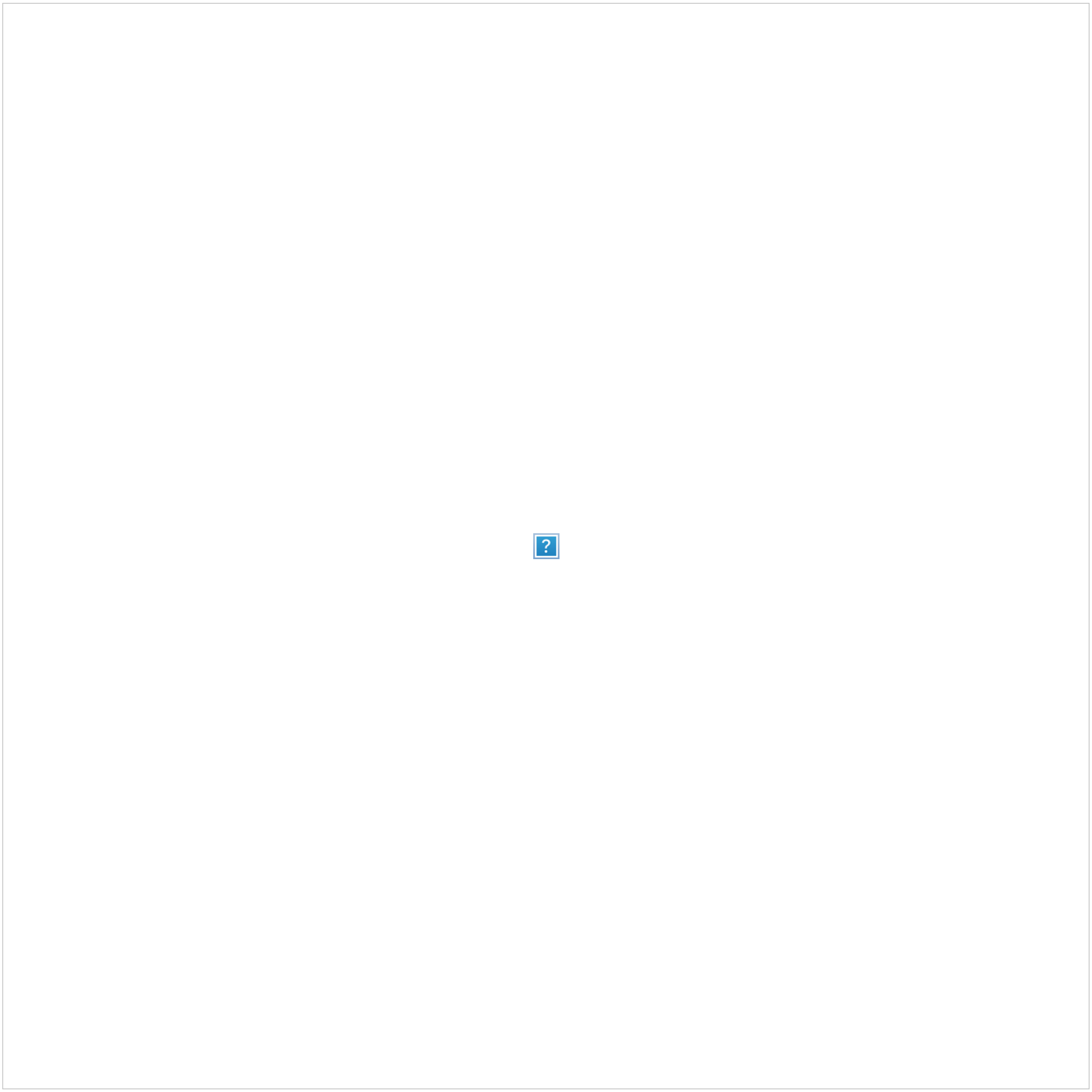
In [7]:
```
train_model(input_to_softmax=model_1,
            pickle_path='model_1.pickle',
            save_model_path='model_1.h5',
            spectrogram=False) # change to False if you would like to use MFCC features
```

```
Epoch 1/20
101/101 [==============================] - 787s 8s/step - loss: 325.5864 - val_loss: 228.9549
Epoch 2/20
101/101 [==============================] - 794s 8s/step - loss: 227.2834 - val_loss: 213.7306
Epoch 3/20
101/101 [==============================] - 796s 8s/step - loss: 206.8042 - val_loss: 204.3813
Epoch 4/20
101/101 [==============================] - 788s 8s/step - loss: 186.0086 - val_loss: 188.7432
Epoch 5/20
101/101 [==============================] - 789s 8s/step - loss: 170.
```

```
4808 - val_loss: 173.6189
Epoch 6/20
101/101 [==============================] - 793s 8s/step - loss: 158.
8490 - val_loss: 160.7199
Epoch 7/20
101/101 [==============================] - 790s 8s/step - loss: 150.
6959 - val_loss: 153.7383
Epoch 8/20
101/101 [==============================] - 783s 8s/step - loss: 144.
3737 - val_loss: 147.5350
Epoch 9/20
101/101 [==============================] - 795s 8s/step - loss: 139.
4791 - val_loss: 144.9805
Epoch 10/20
101/101 [==============================] - 790s 8s/step - loss: 135.
1057 - val_loss: 139.9948
Epoch 11/20
101/101 [==============================] - 787s 8s/step - loss: 131.
7902 - val_loss: 138.6287
Epoch 12/20
101/101 [==============================] - 790s 8s/step - loss: 128.
3992 - val_loss: 139.5652
Epoch 13/20
101/101 [==============================] - 784s 8s/step - loss: 125.
7113 - val_loss: 136.5069
Epoch 14/20
101/101 [==============================] - 794s 8s/step - loss: 123.
6629 - val_loss: 134.7047
Epoch 15/20
101/101 [==============================] - 793s 8s/step - loss: 121.
5452 - val_loss: 131.7925
Epoch 16/20
101/101 [==============================] - 787s 8s/step - loss: 119.
3919 - val_loss: 133.0718
Epoch 17/20
101/101 [==============================] - 793s 8s/step - loss: 117.
4842 - val_loss: 136.7920
Epoch 18/20
101/101 [==============================] - 788s 8s/step - loss: 116.
2350 - val_loss: 131.7407
Epoch 19/20
101/101 [==============================] - 783s 8s/step - loss: 114.
4609 - val_loss: 132.0875
Epoch 20/20
101/101 [==============================] - 789s 8s/step - loss: 113.
3263 - val_loss: 130.6053
```

(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 [8]:
```
model_2 = cnn_rnn_model(input_dim=13, # 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, 13)          0
_________________________________________________________________
conv1d (Conv1D)              (None, None, 200)         28800
_________________________________________________________________
bn_conv_1d (BatchNormalizati (None, None, 200)         800
_________________________________________________________________
rnn (SimpleRNN)              (None, None, 200)         80200
_________________________________________________________________
batch_normalization_2 (Batch (None, None, 200)         800
_________________________________________________________________
time_distributed_2 (TimeDist (None, None, 29)          5829
_________________________________________________________________
softmax (Activation)         (None, None, 29)          0
=================================================================
Total params: 116,429
Trainable params: 115,629
Non-trainable params: 800
_________________________________________________________________
None

/opt/conda/lib/python3.6/site-packages/keras/layers/recurrent.py:100
4: UserWarning: The `implementation` argument in `SimpleRNN` has bee
n deprecated. Please remove it from your layer call.
  warnings.warn('The `implementation` argument '
```

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.

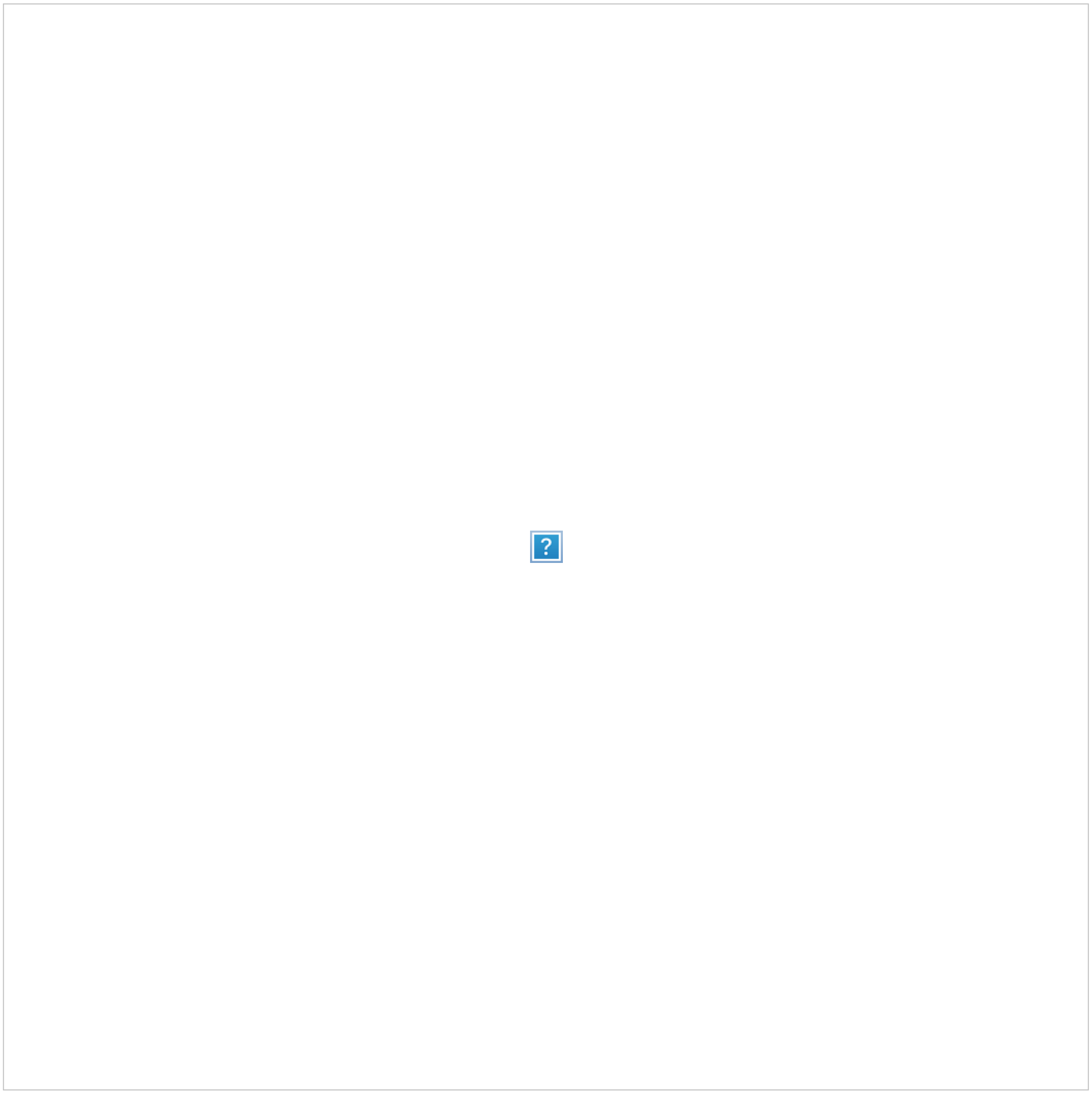
In [9]:
```
train_model(input_to_softmax=model_2,
            pickle_path='model_2.pickle',
            save_model_path='model_2.h5',
            spectrogram=False) # change to False if you would like to
use MFCC features
```

```
Epoch 1/20
101/101 [==============================] - 303s 3s/step - loss: 238.
9334 - val_loss: 203.3130
Epoch 2/20
101/101 [==============================] - 304s 3s/step - loss: 174.
9127 - val_loss: 171.5165
Epoch 3/20
101/101 [==============================] - 300s 3s/step - loss: 152.
7532 - val_loss: 151.0703
Epoch 4/20
101/101 [==============================] - 301s 3s/step - loss: 141.
8524 - val_loss: 147.3402
Epoch 5/20
101/101 [==============================] - 299s 3s/step - loss: 134.
2622 - val_loss: 138.3947
Epoch 6/20
101/101 [==============================] - 300s 3s/step - loss: 128.
6329 - val_loss: 138.5197
Epoch 7/20
101/101 [==============================] - 301s 3s/step - loss: 124.
0926 - val_loss: 136.6730
Epoch 8/20
101/101 [==============================] - 301s 3s/step - loss: 120.
3222 - val_loss: 131.6259
Epoch 9/20
101/101 [==============================] - 302s 3s/step - loss: 116.
8157 - val_loss: 133.9919
Epoch 10/20
101/101 [==============================] - 297s 3s/step - loss: 114.
2222 - val_loss: 132.7398
Epoch 11/20
101/101 [==============================] - 298s 3s/step - loss: 111.
7701 - val_loss: 130.5447
Epoch 12/20
101/101 [==============================] - 298s 3s/step - loss: 109.
3637 - val_loss: 130.9868
Epoch 13/20
101/101 [==============================] - 299s 3s/step - loss: 107.
4162 - val_loss: 130.2533
Epoch 14/20
101/101 [==============================] - 299s 3s/step - loss: 105.
9381 - val_loss: 132.1536
Epoch 15/20
101/101 [==============================] - 297s 3s/step - loss: 103.
7256 - val_loss: 130.9695
Epoch 16/20
101/101 [==============================] - 300s 3s/step - loss: 102.
```

```
2856 - val_loss: 130.8891
Epoch 17/20
101/101 [==============================] - 301s 3s/step - loss: 100.
9630 - val_loss: 132.9525
Epoch 18/20
101/101 [==============================] - 299s 3s/step - loss: 99.3
810 - val_loss: 129.9318
Epoch 19/20
101/101 [==============================] - 295s 3s/step - loss: 98.7
046 - val_loss: 131.3387
Epoch 20/20
101/101 [==============================] - 300s 3s/step - loss: 97.2
362 - val_loss: 130.5514
```

(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 [7]:
```
model_3 = deep_rnn_model(input_dim=13, # change to 13 if you would lik
e to use MFCC features
                         units=200,
                         recur_layers=2)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
the_input (InputLayer)       (None, None, 13)          0         
_________________________________________________________________
lstm_1 (LSTM)                (None, None, 200)         171200    
_________________________________________________________________
batch_normalization_1 (Batch (None, None, 200)         800       
_________________________________________________________________
lstm_2 (LSTM)                (None, None, 200)         320800    
_________________________________________________________________
batch_normalization_2 (Batch (None, None, 200)         800       
_________________________________________________________________
time_distributed_1 (TimeDist (None, None, 29)          5829      
_________________________________________________________________
softmax (Activation)         (None, None, 29)          0         
=================================================================
Total params: 499,429
Trainable params: 498,629
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 [8]:
```
train_model(input_to_softmax=model_3,
            pickle_path='model_3.pickle',
            save_model_path='model_3.h5',
            spectrogram=False) # change to False if you would like to
use MFCC features
```

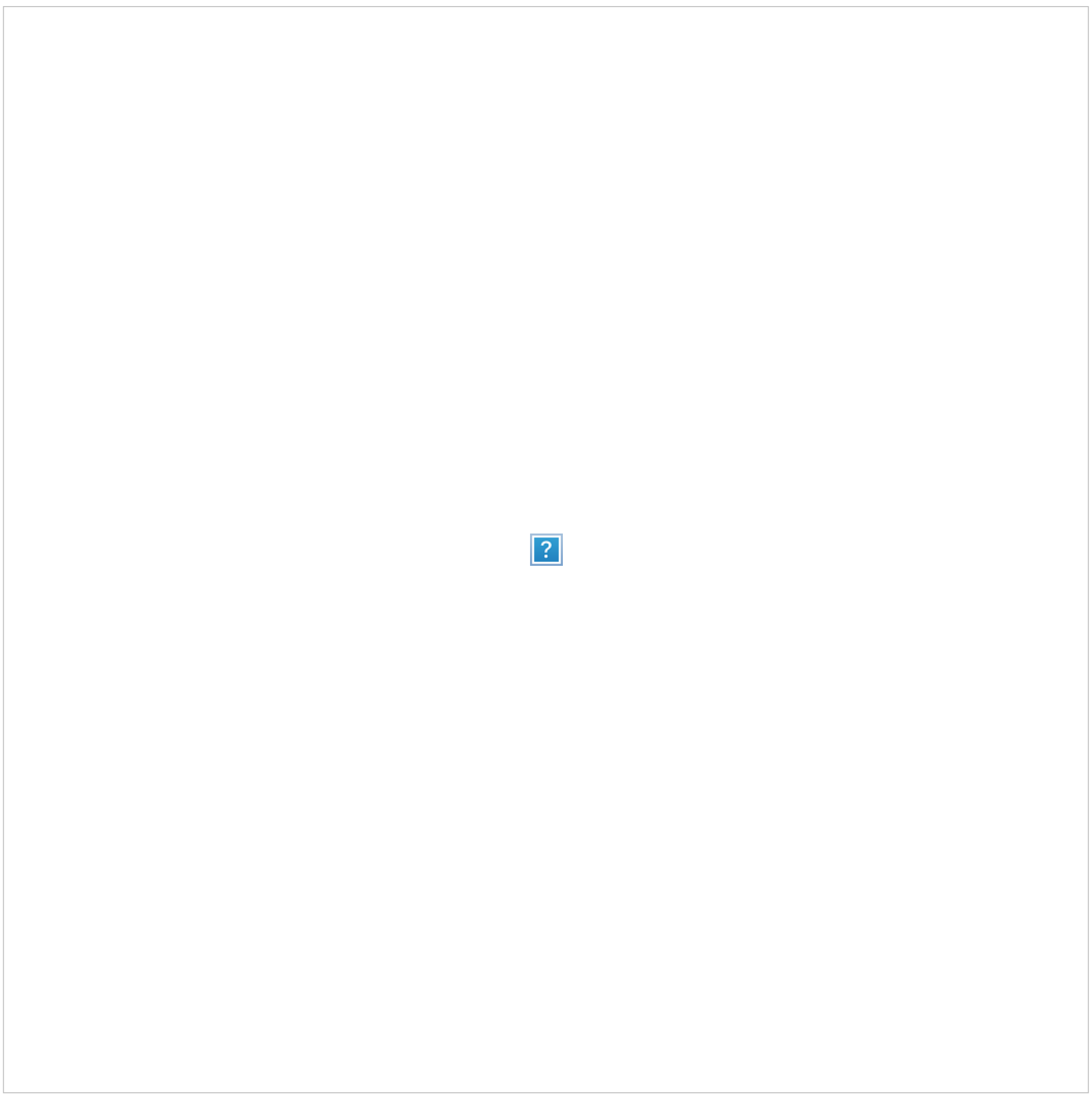
```
Epoch 1/20
101/101 [==============================] - 447s 4s/step - loss: 354.
1638 - val_loss: 268.3433
Epoch 2/20
101/101 [==============================] - 443s 4s/step - loss: 242.
3879 - val_loss: 244.7227
Epoch 3/20
101/101 [==============================] - 443s 4s/step - loss: 226.
4498 - val_loss: 213.1777
Epoch 4/20
101/101 [==============================] - 443s 4s/step - loss: 223.
```

```
4778 - val_loss: 220.5848
Epoch 5/20
101/101 [==============================] - 440s 4s/step - loss: 234.
9024 - val_loss: 220.0530
Epoch 6/20
101/101 [==============================] - 442s 4s/step - loss: 229.
1068 - val_loss: 211.6871
Epoch 7/20
101/101 [==============================] - 441s 4s/step - loss: 223.
7445 - val_loss: 211.4021
Epoch 8/20
101/101 [==============================] - 440s 4s/step - loss: 225.
8091 - val_loss: 209.3851
Epoch 9/20
101/101 [==============================] - 440s 4s/step - loss: 220.
5481 - val_loss: 203.4252
Epoch 10/20
101/101 [==============================] - 439s 4s/step - loss: 218.
8051 - val_loss: 203.4375
Epoch 11/20
101/101 [==============================] - 438s 4s/step - loss: 216.
4746 - val_loss: 202.8055
Epoch 12/20
101/101 [==============================] - 439s 4s/step - loss: 219.
2454 - val_loss: 205.4968
Epoch 13/20
101/101 [==============================] - 443s 4s/step - loss: 219.
0237 - val_loss: 205.6666
Epoch 14/20
101/101 [==============================] - 442s 4s/step - loss: 221.
3313 - val_loss: 208.2998
Epoch 15/20
101/101 [==============================] - 448s 4s/step - loss: 220.
6614 - val_loss: 202.9964
Epoch 16/20
101/101 [==============================] - 433s 4s/step - loss: 218.
0527 - val_loss: 205.1905
Epoch 17/20
101/101 [==============================] - 438s 4s/step - loss: 215.
8668 - val_loss: 203.5983
Epoch 18/20
101/101 [==============================] - 441s 4s/step - loss: 216.
8521 - val_loss: 199.9050
Epoch 19/20
101/101 [==============================] - 437s 4s/step - loss: 218.
5613 - val_loss: 216.3182
Epoch 20/20
101/101 [==============================] - 438s 4s/step - loss: 219.
7533 - val_loss: 205.8332
```

(IMPLEMENTATION) Model 4: Bidirectional RNN + TimeDistributed Dense

Read about the Bidirectional (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).

> 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 [6]:
```
model_4 = bidirectional_rnn_model(input_dim=13, # change to 13 if you
would like to use MFCC features
                                  units=200)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
the_input (InputLayer)       (None, None, 13)          0
_________________________________________________________________
bidirectional_1 (Bidirection (None, None, 400)         256800
_________________________________________________________________
time_distributed_1 (TimeDist (None, None, 29)          11629
_________________________________________________________________
softmax (Activation)         (None, None, 29)          0
=================================================================
Total params: 268,429
Trainable params: 268,429
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_4.h5`. The loss history is saved (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 [7]:
```
train_model(input_to_softmax=model_4,
            pickle_path='model_4.pickle',
            save_model_path='model_4.h5',
            spectrogram=False) # change to False if you would like to
use MFCC features
```

```
Epoch 1/20
101/101 [==============================] - 409s 4s/step - loss: 333.
3424 - val_loss: 228.6528
Epoch 2/20
101/101 [==============================] - 412s 4s/step - loss: 221.
3261 - val_loss: 199.3798
Epoch 3/20
101/101 [==============================] - 413s 4s/step - loss: 201.
5601 - val_loss: 188.2407
Epoch 4/20
101/101 [==============================] - 407s 4s/step - loss: 187.
9250 - val_loss: 177.4040
Epoch 5/20
101/101 [==============================] - 412s 4s/step - loss: 175.
7377 - val_loss: 167.8395
Epoch 6/20
101/101 [==============================] - 416s 4s/step - loss: 164.
2226 - val_loss: 156.8984
```

```
Epoch 7/20
101/101 [==============================] - 416s 4s/step - loss: 153.
7073 - val_loss: 150.1483
Epoch 8/20
101/101 [==============================] - 409s 4s/step - loss: 145.
2970 - val_loss: 143.0757
Epoch 9/20
101/101 [==============================] - 409s 4s/step - loss: 138.
6296 - val_loss: 143.3127
Epoch 10/20
101/101 [==============================] - 412s 4s/step - loss: 133.
2169 - val_loss: 136.9851
Epoch 11/20
101/101 [==============================] - 410s 4s/step - loss: 128.
2420 - val_loss: 135.2069
Epoch 12/20
101/101 [==============================] - 413s 4s/step - loss: 123.
8780 - val_loss: 132.2997
Epoch 13/20
101/101 [==============================] - 410s 4s/step - loss: 120.
2052 - val_loss: 131.8366
Epoch 14/20
101/101 [==============================] - 409s 4s/step - loss: 116.
4509 - val_loss: 129.0648
Epoch 15/20
101/101 [==============================] - 414s 4s/step - loss: 113.
4083 - val_loss: 127.3201
Epoch 16/20
101/101 [==============================] - 409s 4s/step - loss: 110.
3783 - val_loss: 126.0399
Epoch 17/20
101/101 [==============================] - 404s 4s/step - loss: 107.
4399 - val_loss: 126.5143
Epoch 18/20
101/101 [==============================] - 409s 4s/step - loss: 104.
6633 - val_loss: 126.9361
Epoch 19/20
101/101 [==============================] - 412s 4s/step - loss: 102.
0917 - val_loss: 126.9972
Epoch 20/20
101/101 [==============================] - 405s 4s/step - loss: 99.7
134 - val_loss: 127.5758
```

(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 [9]:
```
## (Optional) TODO: Try out some more models!
### Feel free to use as many code cells as needed.
model_5 = lstm_model(input_dim=13, # change to 13 if you would like to
use MFCC features
                                  units=200)
train_model(input_to_softmax=model_5,
            pickle_path='model_5.pickle',
            save_model_path='model_5.h5',
            spectrogram=False) # change to False if you would like to
use MFCC features
```

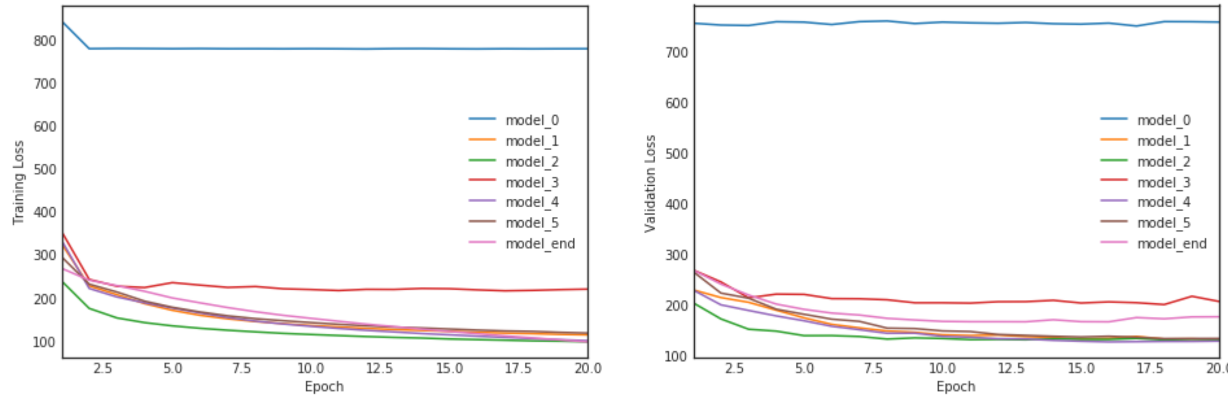
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
the_input (InputLayer)       (None, None, 13)          0
_________________________________________________________________
lstm_1 (LSTM)                (None, None, 200)         171200
_________________________________________________________________
batch_normalization_2 (Batch (None, None, 200)         800
_________________________________________________________________
time_distributed_2 (TimeDist (None, None, 29)          5829
_________________________________________________________________
softmax (Activation)         (None, None, 29)          0
=================================================================
Total params: 177,829
Trainable params: 177,429
Non-trainable params: 400
_________________________________________________________________
None
Epoch 1/20
101/101 [==============================] - 655s 6s/step - loss: 295.
0561 - val_loss: 264.9514
Epoch 2/20
101/101 [==============================] - 656s 6s/step - loss: 231.
3127 - val_loss: 222.6353
Epoch 3/20
101/101 [==============================] - 657s 7s/step - loss: 213.
0211 - val_loss: 212.4489
Epoch 4/20
101/101 [==============================] - 654s 6s/step - loss: 191.
7213 - val_loss: 190.6504
Epoch 5/20
101/101 [==============================] - 654s 6s/step - loss: 177.
4434 - val_loss: 180.6805
Epoch 6/20
101/101 [==============================] - 657s 7s/step - loss: 166.
5867 - val_loss: 171.0039
Epoch 7/20
101/101 [==============================] - 654s 6s/step - loss: 157.
5857 - val_loss: 166.6267
Epoch 8/20
101/101 [==============================] - 654s 6s/step - loss: 151.
0241 - val_loss: 153.2543
```

```
Epoch 9/20
101/101 [==============================] - 655s 6s/step - loss: 146.
2553 - val_loss: 152.4142
Epoch 10/20
101/101 [==============================] - 649s 6s/step - loss: 141.
8850 - val_loss: 147.7338
Epoch 11/20
101/101 [==============================] - 651s 6s/step - loss: 137.
8681 - val_loss: 146.5116
Epoch 12/20
101/101 [==============================] - 652s 6s/step - loss: 134.
4149 - val_loss: 140.7149
Epoch 13/20
101/101 [==============================] - 654s 6s/step - loss: 131.
3531 - val_loss: 138.8734
Epoch 14/20
101/101 [==============================] - 655s 6s/step - loss: 129.
3119 - val_loss: 137.1019
Epoch 15/20
101/101 [==============================] - 646s 6s/step - loss: 126.
9322 - val_loss: 135.7626
Epoch 16/20
101/101 [==============================] - 658s 7s/step - loss: 124.
4475 - val_loss: 137.1899
Epoch 17/20
101/101 [==============================] - 656s 6s/step - loss: 122.
4750 - val_loss: 135.2799
Epoch 18/20
101/101 [==============================] - 653s 6s/step - loss: 121.
4160 - val_loss: 132.4657
Epoch 19/20
101/101 [==============================] - 652s 6s/step - loss: 119.
1266 - val_loss: 132.6831
Epoch 20/20
101/101 [==============================] - 658s 7s/step - loss: 117.
4715 - val_loss: 132.8148
```

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 [1]:
```
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_pick
les]
# 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: MFCCs features (shorter training time) are used for all these models. Bidirderectional+RNN and CNN+RNN models performed the best in terms of both training and validation loss. It is better in compare with signle directional RNN since it provides more information as input and the output layer will use information information of past and future together. CNN+RNN is slightly better and faster than Bidirderectional+RNN model. CNN is a powerful feature detector which operates on a static spatial signal input. DeepRNN model has more parameters than others. Itried to use dropout to avoid parameter overflow and reduce the potential of ovefitting. DeepRNN is the worst better all the models, it might need more iterations, but eventually much higher number of parameters tahn input can cause overflow. Relu activation function has been used in models to prevent gradient descent. DeepRNN has multiple GRU levels which adds up more parameters to be tuned and enables more complex sequences to be captured, but at the same time high number of parameters makes training hypothesis more indicative and prone to overfitting. I added high dropout to control it. On the other hand, high number of parameters in DeepRNN adds up epoch time.
To compare simple RNN model0 with the rest of models, the batch normalization and the time distributed layers effectively improve the loss.
I bet better to try a suitable language model for the final model to make the captured speech sequence more suitable for the context. As course videos mentioned, the tried ML models do not have any idea of the vocabulary/language context and will transform the generated sppech text into something meaningful.

(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/)](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)](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)](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)](https://github.com/buriburisuri/speech-to-text-wavenet). You can work with dilated convolutions [in Keras (https://keras.io/layers/convolutional/)](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)](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)](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 [19]:
```
# specify the model
#model_end = final_model(input_dim=161,units=200)
model_end = final_model(input_dim=161, # change to 13 if you would lik
e to use MFCC features
                        filters=200,
                        kernel_size=11,
                        conv_stride=2,
                        conv_border_mode='valid',
                        units=200,
                        dropout=0.0)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
the_input (InputLayer)       (None, None, 161)         0         
_________________________________________________________________
conv1d (Conv1D)              (None, None, 200)         354400    
_________________________________________________________________
max_pooling1d_6 (MaxPooling1 (None, None, 200)         0         
_________________________________________________________________
batch_normalization_13 (Batc (None, None, 200)         800       
_________________________________________________________________
bidirectional_6 (Bidirection (None, None, 400)         641600    
_________________________________________________________________
batch_normalization_14 (Batc (None, None, 400)         1600      
_________________________________________________________________
time_distributed_6 (TimeDist (None, None, 29)          11629     
_________________________________________________________________
softmax (Activation)         (None, None, 29)          0         
=================================================================
Total params: 1,010,029
Trainable params: 1,008,829
Non-trainable params: 1,200
_________________________________________________________________
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_end.h5`. The loss history is [saved (https://wiki.python.org/moin/UsingPickle)](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 [7]:
```
train_model(input_to_softmax=model_end, pickle_path='model_end.pickle'
,  save_model_path='model_end.h5', spectrogram=True) # change to False
if you would like to use MFCC features
```

```
Epoch 1/20
101/101 [==============================] - 191s 2s/step - loss: 268.
3370 - val_loss: 268.1924
Epoch 2/20
101/101 [==============================] - 190s 2s/step - loss: 240.
```

```
5049 - val_loss: 240.3884
Epoch 3/20
101/101 [==============================] - 191s 2s/step - loss: 227.
8163 - val_loss: 218.8672
Epoch 4/20
101/101 [==============================] - 189s 2s/step - loss: 214.
6863 - val_loss: 200.8132
Epoch 5/20
101/101 [==============================] - 192s 2s/step - loss: 199.
2142 - val_loss: 190.4388
Epoch 6/20
101/101 [==============================] - 193s 2s/step - loss: 187.
8496 - val_loss: 183.0437
Epoch 7/20
101/101 [==============================] - 195s 2s/step - loss: 176.
7977 - val_loss: 179.1765
Epoch 8/20
101/101 [==============================] - 194s 2s/step - loss: 167.
1646 - val_loss: 172.6453
Epoch 9/20
101/101 [==============================] - 195s 2s/step - loss: 158.
7894 - val_loss: 169.4001
Epoch 10/20
101/101 [==============================] - 197s 2s/step - loss: 151.
8215 - val_loss: 166.6044
Epoch 11/20
101/101 [==============================] - 196s 2s/step - loss: 144.
9730 - val_loss: 166.0107
Epoch 12/20
101/101 [==============================] - 198s 2s/step - loss: 138.
2756 - val_loss: 165.9915
Epoch 13/20
101/101 [==============================] - 196s 2s/step - loss: 132.
5264 - val_loss: 165.8516
Epoch 14/20
101/101 [==============================] - 195s 2s/step - loss: 126.
6459 - val_loss: 169.8320
Epoch 15/20
101/101 [==============================] - 195s 2s/step - loss: 121.
2108 - val_loss: 166.0661
Epoch 16/20
101/101 [==============================] - 194s 2s/step - loss: 116.
0700 - val_loss: 165.7873
Epoch 17/20
101/101 [==============================] - 196s 2s/step - loss: 110.
9354 - val_loss: 174.2050
Epoch 18/20
101/101 [==============================] - 194s 2s/step - loss: 106.
2719 - val_loss: 171.7623
Epoch 19/20
101/101 [==============================] - 194s 2s/step - loss: 101.
5592 - val_loss: 175.4567
Epoch 20/20
```

```
101/101 [==============================] - 191s 2s/step - loss: 96.9
870 - val_loss: 175.7240
```

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

Answer: I used one CNN layer as it made good results in the previous model. Max pooling was recomended in the instructions and was incorporated to CNN layer. I added LSTM model as model5 and observed good results in compare with RNN. The reason is mostly related to its memorizing process. I used bidirectional LSTM (BDLSTM) instead of LSTM due to processing the data in both directions with two separate hidden layers, so it has practically two LSTMs. After both CNN and BDLSTM, batch normalization is accomplished. CNN+LSTM can potentially help with eliminating the issue of long term dependences which RNN cannot. In the last layer, we have single TimeDistibuted Dense layer. I used spectrogram feature representations for this final model since of better compatibility with Convolutional NN and max pooling. Overall, results of training loss is better in compare with every single previous models. However, the evaluation loss is higher than previousely investigated models except for DRNN and simpleRNN. The reason can be related to overfitting which is due to high number of parameters in compare with training data, or stucking into local optima and gradient vanishing. Also we have huge number of parameters in compare with previous simpler models which might need more than 20 epochs. The GPU time was not enough to investigate paper's 10 level deep bidirectional LSTM model. But I tried to use bidirectional LSTM and max pooling layer together with the mindset of getting better loss value. For previous models, I used MFCCs features (shorter training time), but for the final model I used spectrogram.

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 [14]:

```
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()
    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 "valid
ation"')

    # 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, a
xis=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 transcri
ptions
    print('-'*80)
    Audio(audio_path)
    print('True transcription:\n' + '\n' + transcr)
    print('-'*80)
    print('Predicted transcription:\n' + '\n' + ''.join(int_sequence_t
o_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 [15]:
```
get_predictions(index=0,
                partition='train',
                input_to_softmax=final_model(input_dim=161, # change t
o 13 if you would like to use MFCC features
                        filters=200,
                        kernel_size=11,
                        conv_stride=2,
                        conv_border_mode='valid',
                        units=200,
                        dropout=0.0),
                model_path='results/model_end.h5')
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
the_input (InputLayer)       (None, None, 161)         0
_________________________________________________________________
conv1d (Conv1D)              (None, None, 200)         354400
_________________________________________________________________
max_pooling1d_4 (MaxPooling1 (None, None, 200)         0
_________________________________________________________________
batch_normalization_9 (Batch (None, None, 200)         800
_________________________________________________________________
bidirectional_4 (Bidirection (None, None, 400)         641600
_________________________________________________________________
batch_normalization_10 (Batc (None, None, 400)         1600
_________________________________________________________________
time_distributed_4 (TimeDist (None, None, 29)          11629
_________________________________________________________________
softmax (Activation)         (None, None, 29)          0
=================================================================
Total params: 1,010,029
Trainable params: 1,008,829
Non-trainable params: 1,200
_________________________________________________________________
None
--------------------------------------------------------------------
------------
True transcription:

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

hr fathrris am mos rmr prsnd to saytie la
--------------------------------------------------------------------
------------
```

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

In [18]:
```
get_predictions(index=0,
                partition='validation',
                input_to_softmax=final_model(input_dim=161, # change t
o 13 if you would like to use MFCC features
                        filters=200,
                        kernel_size=11,
                        conv_stride=2,
                        conv_border_mode='valid',
                        units=200,
                        dropout=0.0),
                model_path='results/model_end.h5')
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
the_input (InputLayer)       (None, None, 161)         0
_________________________________________________________________
conv1d (Conv1D)              (None, None, 200)         354400
_________________________________________________________________
max_pooling1d_5 (MaxPooling1 (None, None, 200)         0
_________________________________________________________________
batch_normalization_11 (Batc (None, None, 200)         800
_________________________________________________________________
bidirectional_5 (Bidirection (None, None, 400)         641600
_________________________________________________________________
batch_normalization_12 (Batc (None, None, 400)         1600
_________________________________________________________________
time_distributed_5 (TimeDist (None, None, 29)          11629
_________________________________________________________________
softmax (Activation)         (None, None, 29)          0
=================================================================
Total params: 1,010,029
Trainable params: 1,008,829
Non-trainable params: 1,200
_________________________________________________________________
None
--------------------------------------------------------------------
------------
True transcription:

the bogus legislature numbered thirty six members
--------------------------------------------------------------------
------------
Predicted transcription:

the bo s irdslurd nvir  theres oeers
--------------------------------------------------------------------
------------
```

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/) 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) model would take 3-6 weeks on a single GPU!