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.

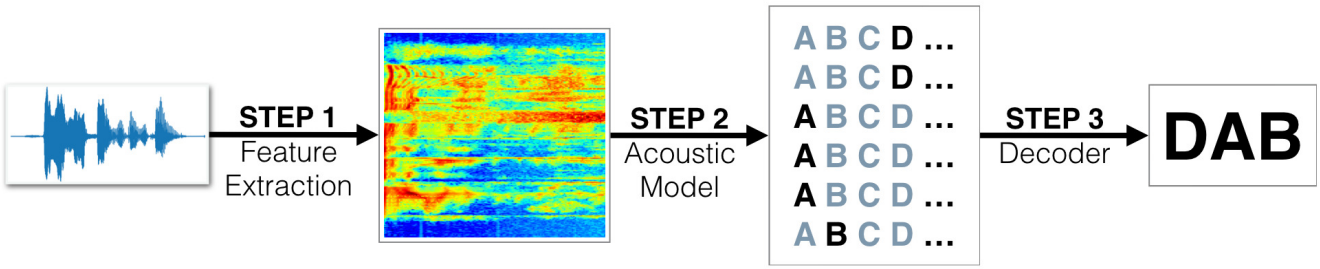
In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes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

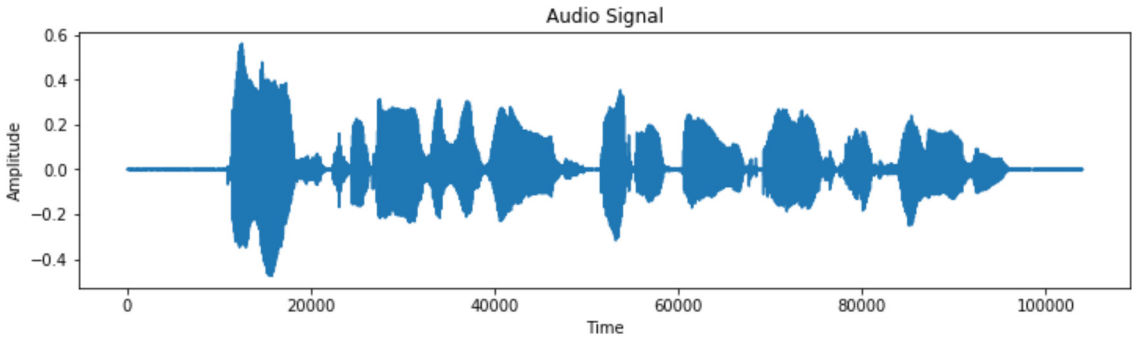
Feel free to use the links below to navigate the notebook:

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_pa
th = 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)
```



Shape of Audio Signal : (103966,)

Transcript : the last two days of the voyage bartley found almost intolerable

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.

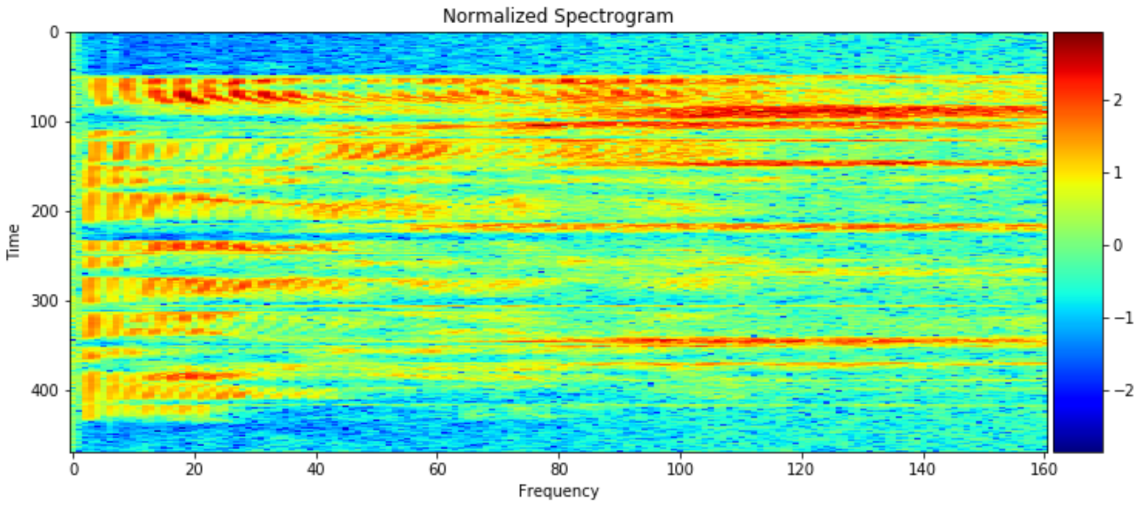
In this project, you will explore two possible feature representations. *After completing the project*, if you'd like to read more about deep learning architectures that can accept raw audio input, you are encouraged to explore this research paper (https://pdfs.semanticscholar.org/a566/cd4a8623d661a4931814d9dffc72ecbf63c4.pdf).

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_feature.shape)))
```



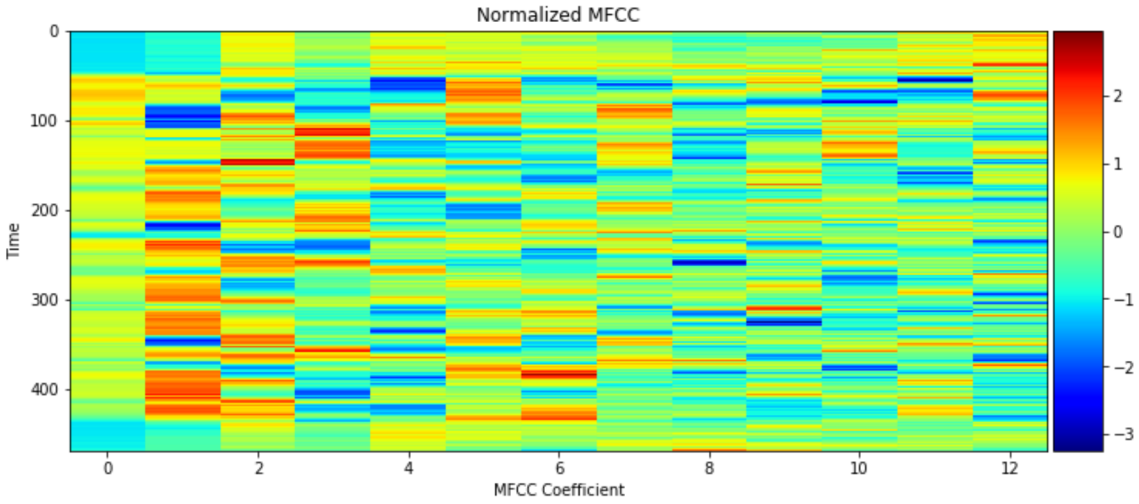
Shape of Spectrogram : (470, 161)

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 : (470, 13)

When you construct your pipeline, you will be able to choose to use either spectrogram or MFCC features. If you would like to see different implementations that make use of MFCCs and/or spectrograms, please check out the links below:

- This repository (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 [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
```

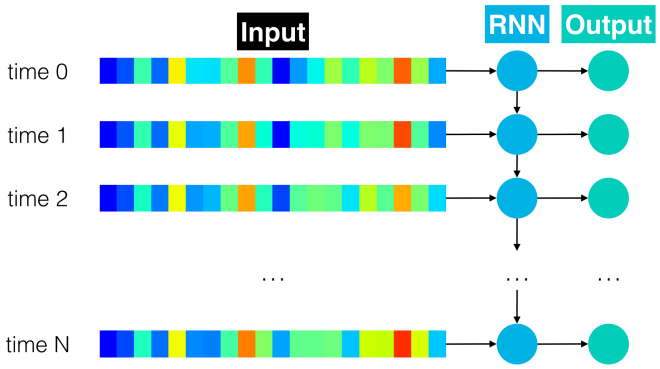
```
Using TensorFlow backend.
```

In [2]:
```
from tensorflow.python.client import device_lib
print(device_lib.list_local_devices())
```

```
[name: "/cpu:0"
device_type: "CPU"
memory_limit: 268435456
locality {
}
incarnation: 11891606997997480372
, name: "/gpu:0"
device_type: "GPU"
memory_limit: 5616697344
locality {
  bus_id: 1
}
incarnation: 502264720863821424
physical_device_desc: "device: 0, name: Tesla K80, pci bus id: 0000:00:1e.0"
]
```

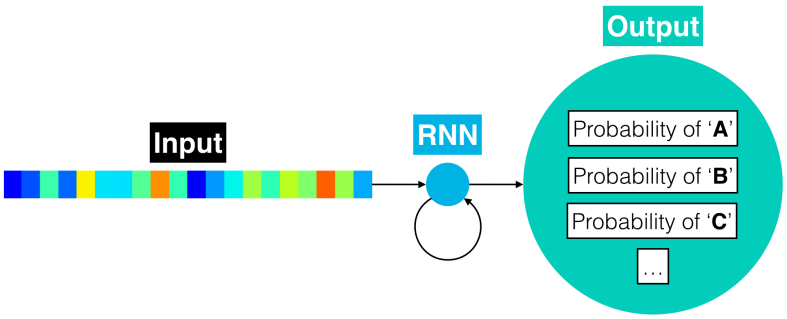
Model 0: RNN

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



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

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



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

In [7]:
```
model_0 = simple_rnn_model(input_dim=161) # change to 13 if you would like to us
e 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/). We have already implemented gradient clipping (https://arxiv.org/pdf/1211.5063.pdf) in your optimizer to help you avoid this issue.

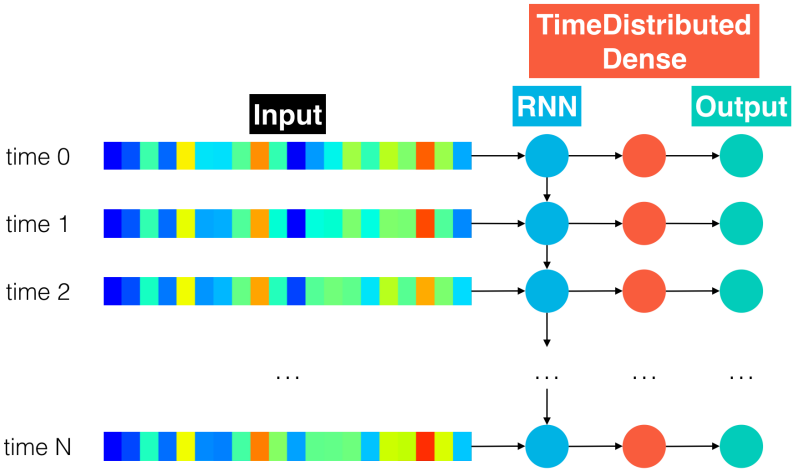
IMPORTANT NOTE: If you notice that your gradient has exploded in any of the models below, feel free to explore more with gradient clipping (the `clipnorm` argument in your optimizer) or swap out any `SimpleRNN` cells for `LSTM` or `GRU` cells. You can also try restarting the kernel to restart the training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

In [8]:
```
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 fe
atures
```

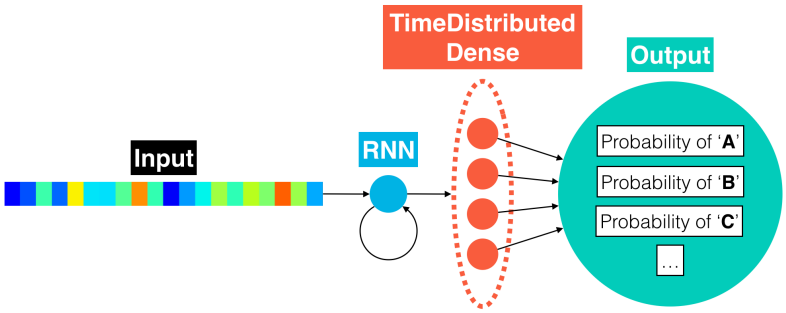
```
Epoch 1/20
106/106 [==============================] - 219s - loss: 836.7680 - val_loss: 7
29.3299
Epoch 2/20
106/106 [==============================] - 223s - loss: 752.6065 - val_loss: 7
26.3149
Epoch 3/20
106/106 [==============================] - 220s - loss: 750.6660 - val_loss: 7
28.9359
Epoch 4/20
106/106 [==============================] - 218s - loss: 750.7634 - val_loss: 7
23.1285
Epoch 5/20
106/106 [==============================] - 220s - loss: 752.0362 - val_loss: 7
27.9438
Epoch 6/20
106/106 [==============================] - 219s - loss: 750.9550 - val_loss: 7
26.3542
Epoch 7/20
106/106 [==============================] - 218s - loss: 750.8023 - val_loss: 7
33.1484
Epoch 8/20
106/106 [==============================] - 219s - loss: 751.0868 - val_loss: 7
25.4525
Epoch 9/20
106/106 [==============================] - 219s - loss: 750.6617 - val_loss: 7
19.5768
Epoch 10/20
106/106 [==============================] - 219s - loss: 752.3876 - val_loss: 7
24.8154
Epoch 11/20
106/106 [==============================] - 219s - loss: 750.8444 - val_loss: 7
25.3032
Epoch 12/20
106/106 [==============================] - 217s - loss: 751.5759 - val_loss: 7
26.8564
Epoch 13/20
106/106 [==============================] - 218s - loss: 750.9542 - val_loss: 7
33.0465
Epoch 14/20
106/106 [==============================] - 218s - loss: 752.0414 - val_loss: 7
17.7849
Epoch 15/20
106/106 [==============================] - 219s - loss: 751.1584 - val_loss: 7
29.0329
Epoch 16/20
106/106 [==============================] - 219s - loss: 750.2183 - val_loss: 7
30.5870
Epoch 17/20
106/106 [==============================] - 217s - loss: 751.5248 - val_loss: 7
19.8137
Epoch 18/20
106/106 [==============================] - 220s - loss: 751.0261 - val_loss: 7
23.3342
Epoch 19/20
106/106 [==============================] - 218s - loss: 751.2195 - val_loss: 7
26.5089
Epoch 20/20
106/106 [==============================] - 216s - loss: 752.2744 - val_loss: 7
27.1031
```

(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 [3]:
```
model_1 = rnn_model(input_dim=13, # change to 13 if you would like to use MFCC features
                    units=290,
                    activation='relu')
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
the_input (InputLayer)       (None, None, 13)          0
_________________________________________________________________
rnn (GRU)                    (None, None, 290)         264480
_________________________________________________________________
batch_normalization_1 (Batch (None, None, 290)         1160
_________________________________________________________________
time_distributed_1 (TimeDist (None, None, 29)          8439
_________________________________________________________________
softmax (Activation)         (None, None, 29)          0
=================================================================
Total params: 274,079
Trainable params: 273,499
Non-trainable params: 580
_________________________________________________________________
None
```

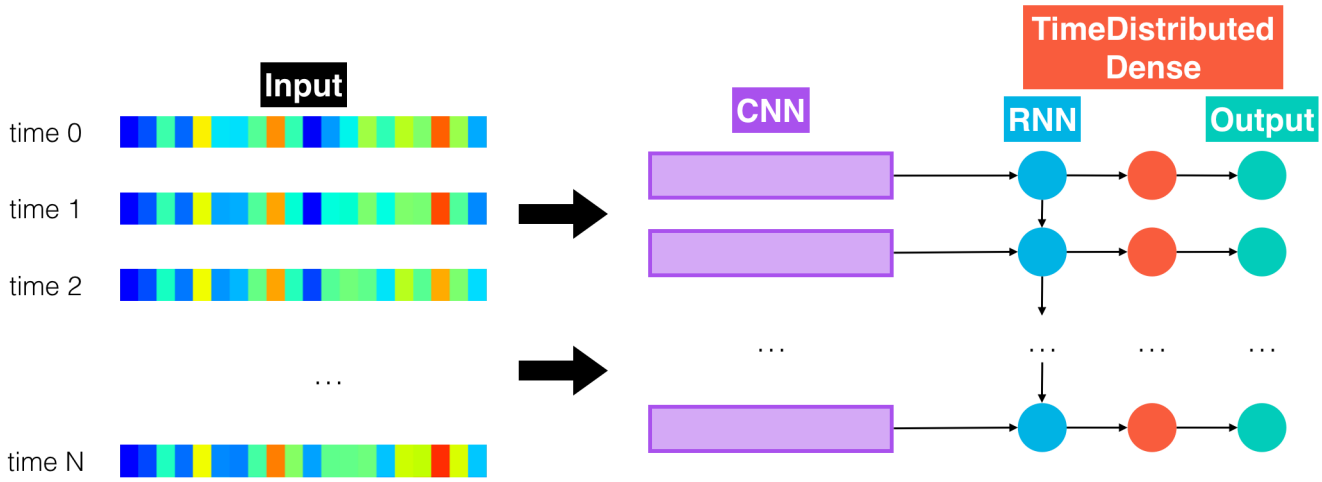
Please execute the code cell below to train the neural network you specified in `input_to_softmax`. After the model has finished training, the model is saved (https://keras.io/getting-started/faq/#how-can-i-save-a-keras-model) in the HDF5 file `model_1.h5`. The loss history is saved (https://wiki.python.org/moin/UsingPickle) 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 [4]:
```
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 f
eatures
```

```
Epoch 1/20
106/106 [==============================] - 262s - loss: 294.1912 - val_loss: 2
54.6544
Epoch 2/20
106/106 [==============================] - 251s - loss: 216.0005 - val_loss: 2
11.1898
Epoch 3/20
106/106 [==============================] - 247s - loss: 196.5010 - val_loss: 2
10.1823
Epoch 4/20
106/106 [==============================] - 246s - loss: 181.7126 - val_loss: 1
75.9134
Epoch 5/20
106/106 [==============================] - 246s - loss: 169.9244 - val_loss: 1
73.2393
Epoch 6/20
106/106 [==============================] - 245s - loss: 159.7307 - val_loss: 1
56.1874
Epoch 7/20
106/106 [==============================] - 246s - loss: 149.9815 - val_loss: 1
48.0650
Epoch 8/20
106/106 [==============================] - 246s - loss: 143.4072 - val_loss: 1
46.1457
Epoch 9/20
106/106 [==============================] - 246s - loss: 137.9817 - val_loss: 1
41.2213
Epoch 10/20
106/106 [==============================] - 247s - loss: 133.7196 - val_loss: 1
41.2503
Epoch 11/20
106/106 [==============================] - 248s - loss: 130.2216 - val_loss: 1
39.0630
Epoch 12/20
106/106 [==============================] - 246s - loss: 126.8203 - val_loss: 1
34.7727
Epoch 13/20
106/106 [==============================] - 247s - loss: 124.0081 - val_loss: 1
35.2370
Epoch 14/20
106/106 [==============================] - 248s - loss: 121.5541 - val_loss: 1
32.5228
Epoch 15/20
106/106 [==============================] - 248s - loss: 118.7180 - val_loss: 1
31.7590
Epoch 16/20
106/106 [==============================] - 248s - loss: 116.7131 - val_loss: 1
31.3717
Epoch 17/20
106/106 [==============================] - 247s - loss: 114.8191 - val_loss: 1
28.2194
Epoch 18/20
106/106 [==============================] - 247s - loss: 112.8423 - val_loss: 1
30.1607
Epoch 19/20
106/106 [==============================] - 247s - loss: 110.9899 - val_loss: 1
28.7145
Epoch 20/20
106/106 [==============================] - 246s - loss: 109.6401 - val_loss: 1
27.8896
```

(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 [5]:
```
model_2 = cnn_rnn_model(input_dim=13, # change to 13 if you would like to use MFCC features
                        filters=256,
                        kernel_size=5,
                        conv_stride=2,
                        conv_border_mode='same',
                        units=290)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
the_input (InputLayer)       (None, None, 13)          0
_________________________________________________________________
conv1d (Conv1D)              (None, None, 256)         16896
_________________________________________________________________
bn_conv_1d (BatchNormalizati (None, None, 256)         1024
_________________________________________________________________
rnn (SimpleRNN)              (None, None, 290)         158630
_________________________________________________________________
batch_normalization_2 (Batch (None, None, 290)         1160
_________________________________________________________________
time_distributed_2 (TimeDist (None, None, 29)          8439
_________________________________________________________________
softmax (Activation)         (None, None, 29)          0
=================================================================
Total params: 186,149
Trainable params: 185,057
Non-trainable params: 1,092
_________________________________________________________________
None
```

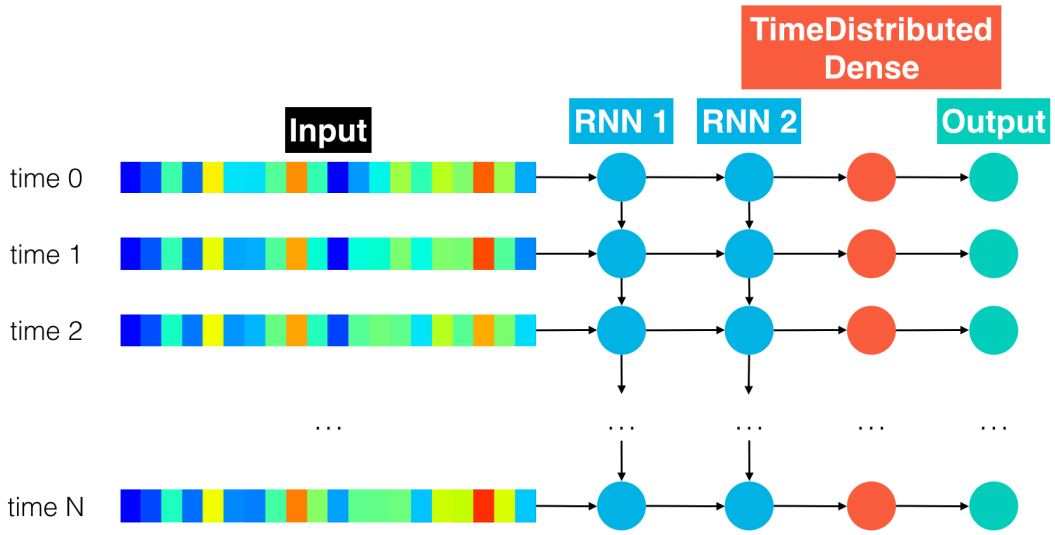
Please execute the code cell below to train the neural network you specified in `input_to_softmax`. After the model has finished training, the model is saved (https://keras.io/getting-started/faq/#how-can-i-save-a-keras-model) in the HDF5 file `model_2.h5`. The loss history is saved (https://wiki.python.org/moin/UsingPickle) 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 [ ]:
```
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 f
eatures
```

```
Epoch 1/20
106/106 [==============================] - 119s - loss: 237.3705 - val_loss: 3
13.9894
Epoch 2/20
106/106 [==============================] - 110s - loss: 173.1344 - val_loss: 2
05.8838
Epoch 3/20
106/106 [==============================] - 110s - loss: 150.4712 - val_loss: 1
78.4606
Epoch 4/20
106/106 [==============================] - 110s - loss: 138.8248 - val_loss: 1
69.6250
Epoch 5/20
106/106 [==============================] - 109s - loss: 131.0650 - val_loss: 1
69.6449
Epoch 6/20
106/106 [==============================] - 112s - loss: 125.5799 - val_loss: 1
61.7264
Epoch 7/20
106/106 [==============================] - 112s - loss: 121.3194 - val_loss: 1
60.6534
Epoch 8/20
106/106 [==============================] - 110s - loss: 117.5797 - val_loss: 1
56.4396
Epoch 9/20
106/106 [==============================] - 111s - loss: 114.2150 - val_loss: 1
58.8303
Epoch 10/20
106/106 [==============================] - 110s - loss: 111.2563 - val_loss: 1
51.1114
Epoch 11/20
106/106 [==============================] - 112s - loss: 108.9397 - val_loss: 1
51.3016
Epoch 12/20
106/106 [==============================] - 112s - loss: 107.4238 - val_loss: 1
42.3869
Epoch 13/20
106/106 [==============================] - 111s - loss: 104.7226 - val_loss: 1
37.7175
Epoch 14/20
106/106 [==============================] - 110s - loss: 102.9788 - val_loss: 1
33.5074
Epoch 15/20
106/106 [==============================] - 110s - loss: 101.2581 - val_loss: 1
25.6800
Epoch 16/20
106/106 [==============================] - 108s - loss: 99.6398 - val_loss: 12
6.1068
Epoch 17/20
106/106 [==============================] - 110s - loss: 98.2410 - val_loss: 12
3.3138
Epoch 18/20
106/106 [==============================] - 109s - loss: 95.9268 - val_loss: 12
2.9961
Epoch 20/20
106/106 [==============================] - 108s - loss: 94.4538 - val_loss: 12
3.2947
```

(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 [6]:
```
model_3 = deep_rnn_model(input_dim=13, # 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, 13)          0
_________________________________________________________________
rnn_0 (GRU)                  (None, None, 200)         128400
_________________________________________________________________
batch_normalization_3 (Batch (None, None, 200)         800
_________________________________________________________________
rnn_1 (GRU)                  (None, None, 200)         240600
_________________________________________________________________
batch_normalization_4 (Batch (None, None, 200)         800
_________________________________________________________________
time_distributed_2 (TimeDist (None, None, 29)          5829
_________________________________________________________________
softmax (Activation)         (None, None, 29)          0
=================================================================
Total params: 376,429
Trainable params: 375,629
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.

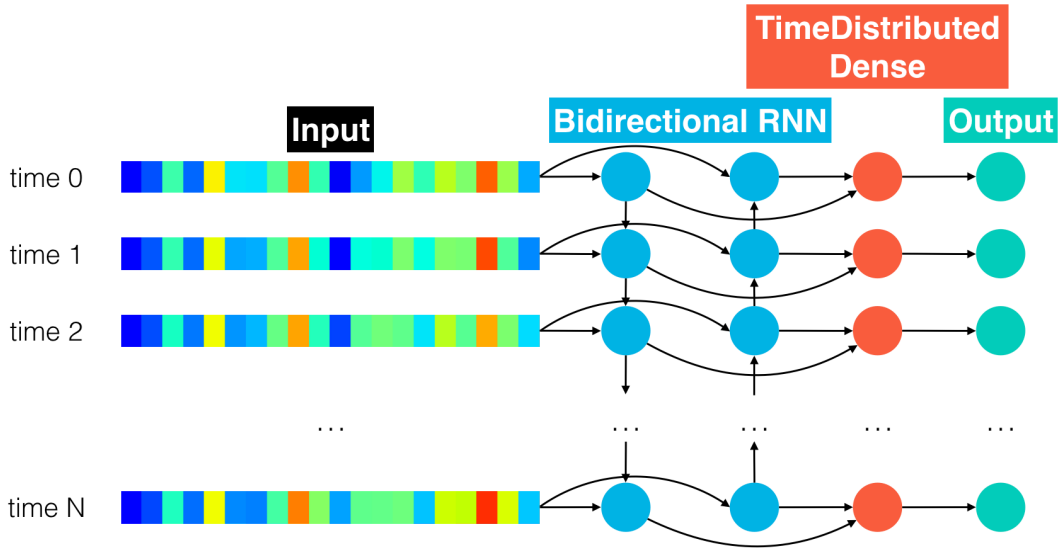
In [10]:
```
from keras.optimizers import Adam

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 f
eatures
            optimizer=Adam(lr=0.001, beta_1=0.9, beta_2=0.999, decay=0.0, clipno
rm=8))
```

```
Epoch 1/20
106/106 [==============================] - 417s - loss: 403.4689 - val_loss: 4
31.7046
Epoch 2/20
106/106 [==============================] - 406s - loss: 327.7901 - val_loss: 4
23.8852
Epoch 3/20
106/106 [==============================] - 412s - loss: 514.7336 - val_loss: 6
91.2754
Epoch 4/20
106/106 [==============================] - 408s - loss: 242.6085 - val_loss: 6
18.8003
Epoch 5/20
106/106 [==============================] - 409s - loss: 216.6488 - val_loss: 3
05.3557
Epoch 6/20
106/106 [==============================] - 409s - loss: 195.0958 - val_loss: 2
05.3268
Epoch 7/20
106/106 [==============================] - 407s - loss: 180.8353 - val_loss: 1
90.1409
Epoch 8/20
106/106 [==============================] - 409s - loss: 167.8513 - val_loss: 1
87.8062
Epoch 9/20
106/106 [==============================] - 410s - loss: 156.0686 - val_loss: 1
83.3546
Epoch 10/20
106/106 [==============================] - 407s - loss: 146.3216 - val_loss: 1
63.6247
Epoch 11/20
106/106 [==============================] - 410s - loss: 139.5912 - val_loss: 1
50.3507
Epoch 12/20
106/106 [==============================] - 407s - loss: 133.7296 - val_loss: 1
43.1290
Epoch 13/20
106/106 [==============================] - 406s - loss: 129.0570 - val_loss: 1
36.4850
Epoch 14/20
106/106 [==============================] - 408s - loss: 123.7975 - val_loss: 1
33.2669
Epoch 15/20
106/106 [==============================] - 409s - loss: 120.5100 - val_loss: 1
31.0477
Epoch 16/20
106/106 [==============================] - 407s - loss: 117.1043 - val_loss: 1
31.8951
Epoch 17/20
106/106 [==============================] - 407s - loss: 114.5056 - val_loss: 1
26.0870
Epoch 18/20
106/106 [==============================] - 409s - loss: 111.7987 - val_loss: 1
27.0753
Epoch 19/20
106/106 [==============================] - 406s - loss: 109.2756 - val_loss: 1
27.8813
Epoch 20/20
106/106 [==============================] - 408s - loss: 107.3449 - val_loss: 1
26.5656
```

(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 [3]:
```
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 [4]:
```
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 f
eatures
```

```
Epoch 1/20
106/106 [==============================] - 414s - loss: 265.7775 - val_loss: 2
04.1835
Epoch 2/20
106/106 [==============================] - 407s - loss: 203.8705 - val_loss: 1
91.1697
Epoch 3/20
106/106 [==============================] - 407s - loss: 191.0002 - val_loss: 1
84.4619
Epoch 4/20
106/106 [==============================] - 402s - loss: 181.5477 - val_loss: 1
72.3306
Epoch 5/20
106/106 [==============================] - 404s - loss: 173.9652 - val_loss: 1
65.9417
Epoch 6/20
106/106 [==============================] - 405s - loss: 166.7944 - val_loss: 1
62.4791
Epoch 7/20
106/106 [==============================] - 404s - loss: 161.2586 - val_loss: 1
59.1542
Epoch 8/20
106/106 [==============================] - 404s - loss: 155.6194 - val_loss: 1
55.8756
Epoch 9/20
106/106 [==============================] - 405s - loss: 150.4641 - val_loss: 1
49.3443
Epoch 10/20
106/106 [==============================] - 408s - loss: 146.3055 - val_loss: 1
45.5644
Epoch 11/20
106/106 [==============================] - 408s - loss: 141.6571 - val_loss: 1
43.9932
Epoch 12/20
106/106 [==============================] - 407s - loss: 137.0738 - val_loss: 1
41.6694
Epoch 13/20
106/106 [==============================] - 405s - loss: 133.1703 - val_loss: 1
39.8378
Epoch 14/20
106/106 [==============================] - 405s - loss: 129.6341 - val_loss: 1
35.4649
Epoch 15/20
106/106 [==============================] - 410s - loss: 126.0948 - val_loss: 1
39.0950
Epoch 16/20
106/106 [==============================] - 409s - loss: 123.3921 - val_loss: 1
32.5197
Epoch 17/20
106/106 [==============================] - 409s - loss: 120.4250 - val_loss: 1
35.0312
Epoch 18/20
106/106 [==============================] - 408s - loss: 117.7728 - val_loss: 1
31.7822
Epoch 19/20
106/106 [==============================] - 411s - loss: 115.1869 - val_loss: 1
28.0137
Epoch 20/20
106/106 [==============================] - 409s - loss: 112.5118 - val_loss: 1
31.3392
```

(OPTIONAL IMPLEMENTATION) Models 5+

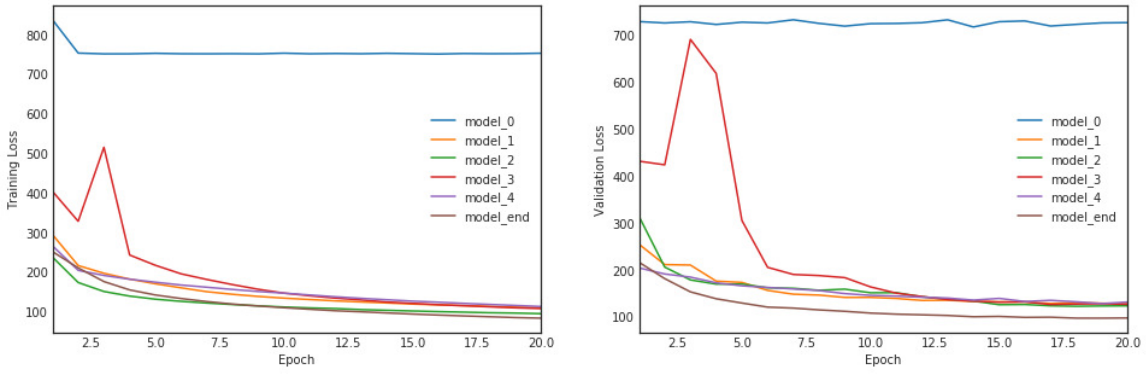
If you would like to try out more architectures than the ones above, please use the code cell below. Please continue to follow the same convention for saving the models; for the i-th sample model, please save the loss at **`model_i.pickle`** and saving the trained model at **`model_i.h5`**.

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

Compare the Models

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

In [3]:
```
from glob import glob
import numpy as np
import _pickle as pickle
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
sns.set_style(style='white')

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

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

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

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



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

Answer: The following were key observations regarding the results by using various models.

- The best performing models are in this order:
  - model_2 (CNN + RNN + TimeDistributed Dense) = 123
  - model_3 (Deeper RNN + TimeDistributed Dense) = 126
  - model_1 (RNN + TimeDistributed Dense) = 127
  - model_4 (Bidirectional RNN + TimeDistributed Dense) = 131
  - model_0 (RNN) = 727
    - The best performing model was the one that combines CNNs + RNNs. It seems that the model is learning patterns in a better way when we introduce a CNN layer initially.
    - This obviously makes sense because the model is finding patterns and extracting features using the CNN layer. Then, it uses the RNN layer to connect these patterns in a recurrent way.
- Additional points observed
  - Training with the MFCC features is giving better results as compared to the raw spectogram inputs. (around 10% better)
  - Each of the above models are subject to overfitting. This is evident because of a lack of measures like Dropouts.
  - Deeper RNNs seem to yield better results though we need to try with deeper layers in the final model.
  - Bidirectional convolutions only give marginal improvements.

(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 [3]:
```
model_end = final_model(input_dim=13, # change to 13 if you would like to use MFCC features
                        filters=200,
                        kernel_size=5,
                        conv_stride=1,
                        conv_border_mode='causal',
                        units=200,
                        output_dim=29,
                        recur_layers=4)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
the_input (InputLayer)       (None, None, 13)          0
_________________________________________________________________
conv1d (Conv1D)              (None, None, 200)         13200
_________________________________________________________________
bn_conv_1d (BatchNormalizati (None, None, 200)         800
_________________________________________________________________
bidirectional_1 (Bidirection (None, None, 400)         481200
_________________________________________________________________
batch_normalization_1 (Batch (None, None, 400)         1600
_________________________________________________________________
bidirectional_2 (Bidirection (None, None, 400)         721200
_________________________________________________________________
batch_normalization_2 (Batch (None, None, 400)         1600
_________________________________________________________________
bidirectional_3 (Bidirection (None, None, 400)         721200
_________________________________________________________________
batch_normalization_3 (Batch (None, None, 400)         1600
_________________________________________________________________
bidirectional_4 (Bidirection (None, None, 400)         721200
_________________________________________________________________
batch_normalization_4 (Batch (None, None, 400)         1600
_________________________________________________________________
time_distributed_1 (TimeDist (None, None, 29)          11629
_________________________________________________________________
softmax (Activation)         (None, None, 29)          0
=================================================================
Total params: 2,676,829
Trainable params: 2,673,229
Non-trainable params: 3,600
_________________________________________________________________
None
```

Please execute the code cell below to train the neural network you specified in `input_to_softmax`. After the model has finished training, the model is saved (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 [4]:
```
from keras.optimizers import Adam

train_model(input_to_softmax=model_end,
            pickle_path='model_end.pickle',
            save_model_path='model_end.h5',
            spectrogram=False, # change to False if you would like to use MFCC f
eatures
            optimizer=Adam(lr=0.001, beta_1=0.9, beta_2=0.999, decay=0.0, clipno
rm=6) )
```

```
Epoch 1/20
106/106 [==============================] - 1538s - loss: 284.1660 - val_loss:
242.3701
Epoch 2/20
106/106 [==============================] - 1548s - loss: 213.3268 - val_loss:
210.5785
Epoch 3/20
106/106 [==============================] - 1544s - loss: 179.2643 - val_loss:
156.3663
Epoch 4/20
106/106 [==============================] - 1550s - loss: 157.0097 - val_loss:
140.0854
Epoch 5/20
106/106 [==============================] - 1541s - loss: 141.2417 - val_loss:
126.3523
Epoch 6/20
106/106 [==============================] - 1550s - loss: 129.9805 - val_loss:
121.9597
Epoch 7/20
106/106 [==============================] - 1554s - loss: 122.0799 - val_loss:
112.9084
Epoch 8/20
106/106 [==============================] - 1556s - loss: 116.4391 - val_loss:
111.2053
Epoch 9/20
106/106 [==============================] - 1540s - loss: 111.7682 - val_loss:
104.7955
Epoch 10/20
106/106 [==============================] - 1544s - loss: 106.9278 - val_loss:
104.8252
Epoch 11/20
106/106 [==============================] - 1554s - loss: 104.0302 - val_loss:
101.6594
Epoch 12/20
106/106 [==============================] - 1551s - loss: 101.9616 - val_loss:
100.6103
Epoch 13/20
106/106 [==============================] - 1555s - loss: 99.0360 - val_loss: 9
9.9524
Epoch 14/20
106/106 [==============================] - 1548s - loss: 96.9942 - val_loss: 9
8.4548
Epoch 15/20
106/106 [==============================] - 1568s - loss: 94.6818 - val_loss: 9
4.9515
Epoch 16/20
106/106 [==============================] - 1560s - loss: 91.6827 - val_loss: 9
6.0808
Epoch 17/20
106/106 [==============================] - 1541s - loss: 89.4326 - val_loss: 9
3.4115
Epoch 18/20
106/106 [==============================] - 1556s - loss: 88.8395 - val_loss: 9
3.0651
Epoch 19/20
106/106 [==============================] - 1543s - loss: 86.5407 - val_loss: 9
2.3146
Epoch 20/20
106/106 [==============================] - 1556s - loss: 84.8383 - val_loss: 9
0.7013
```

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

Answer: Based on the observations from the results of previous models, I have strucutured this model in the following way.

- The first layer is a convolutional layer with the following specifics:
  - Kernel size of 5 and stride size of 1
  - Dilated convolution (Keras supports it by a border model of 'causal') with a dilation of 2
  - The activaiton is a 'relu'
  - Added BatchNormalization to the output
  - Dilated convolutions seem to perform better based on this article : WaveNet paper (https://arxiv.org/abs/1609.03499)
    - Hence, added a dilated convolution with a kernel size of 5 and a stride size of 1 (required)
    - The dilation rate was set at 2. Anything larger was leading to gradients exploding and the model was not learning.
    - Tuned the model output length function accordingly.
- Added RNN layers as the next set of deep layers, with the following specifics:
  - Used a 'GRU' unit, activation is 'relu' by default
  - Added a dropout (recurrent and input) as 0.2
  - Wrapped it in a bidirectional RNN layer
  - Added BatchNormalization to the output
  - Noted that a layer with 4 RNN layers was giving better results.
  - This was of course taking a lot of time to train as can be seen in each of the steps below. (8 hrs / epochs=20)
- Plugged this into a TimeDistributed Dense layer with an activation of softmax

Additional details:

- Used 200 units for all the hidden layers
- Used the MFCC features as this was giving a better result over the Spectogram (less overfitting)
- The relus help in preventing the vanishing gradient problem to an extent.
- Used an Adam optimizer with an LR of 0.001 with an added clipnorm to prevent gradient explosion.
- Reduced overfitting by using dropouts.

Results/Conclusion:

- Overall, observed a significant improvement in the accuracy as compared to the earlier models. (90%)
- This resulted in a validation loss of 90 as compared to the best case of 123 earlier with Model-2 (CNN-RNN model)
- Ran this only for 20 iterations, but I think if we tune this further and run for a few more iterations, it should improve much more. (Was running out of available credits in AWS)
- Also, the overfitting is the least as compared to the other models, as the difference between the training loss and the validation loss is very small.

Future enhancement thoughts: Overall, my conclusion is that if we fine tune the final model a bit more and run it for a larger number of iterations and also with a larger datasets, we will get much better results. Thinking of doing this after I submit the project, as a future enhancement.

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 [9]:
```
import numpy as np
from data_generator import AudioGenerator
from keras import backend as K
from utils import int_sequence_to_text
from IPython.display import Audio

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

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

    # obtain and decode the acoustic model's predictions
    input_to_softmax.load_weights(model_path)
    prediction = input_to_softmax.predict(np.expand_dims(data_point, axis=0))
    output_length = [input_to_softmax.output_length(data_point.shape[0])]
    pred_ints = (K.eval(K.ctc_decode(
                prediction, output_length)[0][0])+1).flatten().tolist()

    # play the audio file, and display the true and predicted transcriptions
    print('-'*80)
    Audio(audio_path)
    print('True transcription:\n' + '\n' + transcr)
    print('-'*80)
    print('Predicted transcription:\n' + '\n' + ''.join(int_sequence_to_text(pre
d_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 [11]:
```
final_mod = final_model(input_dim=13, # change to 13 if you would like to use MF
CC features
                        filters=200,
                        kernel_size=5,
                        conv_stride=1,
                        conv_border_mode='causal',
                        units=200,
                        output_dim=29,
                        recur_layers=4)
get_predictions(index=0,
                partition='train',
                input_to_softmax=final_mod,
                model_path='results/model_end.h5')
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
the_input (InputLayer)       (None, None, 13)          0
_________________________________________________________________
conv1d (Conv1D)              (None, None, 200)         13200
_________________________________________________________________
bn_conv_1d (BatchNormalizati (None, None, 200)         800
_________________________________________________________________
bidirectional_17 (Bidirectio (None, None, 400)         481200
_________________________________________________________________
batch_normalization_17 (Batc (None, None, 400)         1600
_________________________________________________________________
bidirectional_18 (Bidirectio (None, None, 400)         721200
_________________________________________________________________
batch_normalization_18 (Batc (None, None, 400)         1600
_________________________________________________________________
bidirectional_19 (Bidirectio (None, None, 400)         721200
_________________________________________________________________
batch_normalization_19 (Batc (None, None, 400)         1600
_________________________________________________________________
bidirectional_20 (Bidirectio (None, None, 400)         721200
_________________________________________________________________
batch_normalization_20 (Batc (None, None, 400)         1600
_________________________________________________________________
time_distributed_5 (TimeDist (None, None, 29)          11629
_________________________________________________________________
softmax (Activation)         (None, None, 29)          0
=================================================================
Total params: 2,676,829
Trainable params: 2,673,229
Non-trainable params: 3,600
_________________________________________________________________
None
--------------------------------------------------------------------------------
--
True transcription:

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

the last to dae of the vogh bartly fond amostim talerable
--------------------------------------------------------------------------------
--
```

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

In [15]:
```
final_mod = final_model(input_dim=13, # change to 13 if you would like to use MF
CC features
                        filters=200,
                        kernel_size=5,
                        conv_stride=1,
                        conv_border_mode='causal',
                        units=200,
                        output_dim=29,
                        recur_layers=4)
get_predictions(index=0,
                partition='validation',
                input_to_softmax=final_mod,
                model_path='results/model_end.h5')
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
the_input (InputLayer)       (None, None, 13)          0
_________________________________________________________________
conv1d (Conv1D)              (None, None, 200)         13200
_________________________________________________________________
bn_conv_1d (BatchNormalizati (None, None, 200)         800
_________________________________________________________________
bidirectional_25 (Bidirectio (None, None, 400)         481200
_________________________________________________________________
batch_normalization_25 (Batc (None, None, 400)         1600
_________________________________________________________________
bidirectional_26 (Bidirectio (None, None, 400)         721200
_________________________________________________________________
batch_normalization_26 (Batc (None, None, 400)         1600
_________________________________________________________________
bidirectional_27 (Bidirectio (None, None, 400)         721200
_________________________________________________________________
batch_normalization_27 (Batc (None, None, 400)         1600
_________________________________________________________________
bidirectional_28 (Bidirectio (None, None, 400)         721200
_________________________________________________________________
batch_normalization_28 (Batc (None, None, 400)         1600
_________________________________________________________________
time_distributed_7 (TimeDist (None, None, 29)          11629
_________________________________________________________________
softmax (Activation)         (None, None, 29)          0
=================================================================
Total params: 2,676,829
Trainable params: 2,673,229
Non-trainable params: 3,600
_________________________________________________________________
None
--------------------------------------------------------------------------------
--
True transcription:

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

ohon tho wode stod in nice slitl firtry
--------------------------------------------------------------------------------
--
```

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!