

Artificial Intelligence Nanodegree

Voice User Interfaces

Project: Speech Recognition with Neural Networks

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

Note: Once you have completed all of the code implementations, you need to finalize your work by exporting the Jupyter Notebook as an HTML document. Before exporting the notebook to html, all of the code cells need to have been run so that reviewers can see the final implementation and output. You can then export the notebook by using the menu above and navigating to "\n", "**File -> Download as -> HTML (.html)**". Include the finished document along with this notebook as your submission.

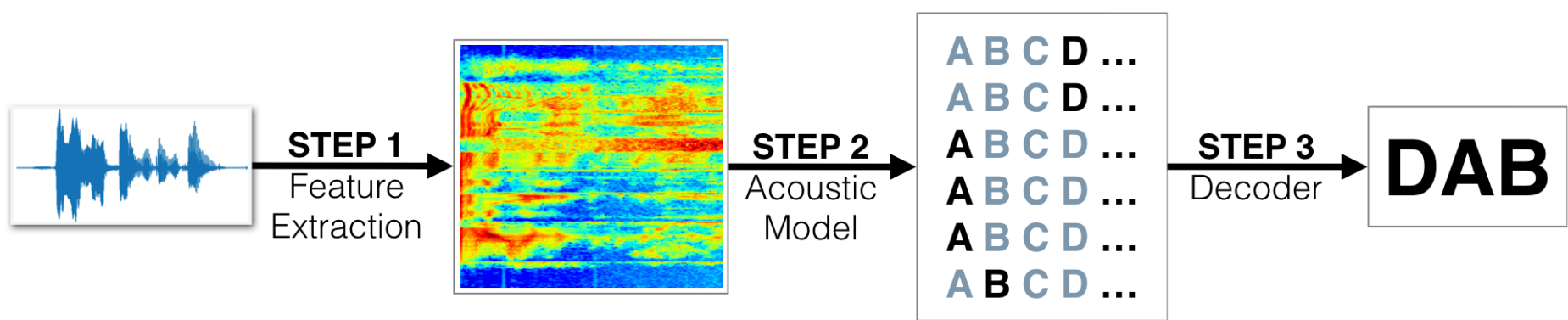
In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a '**Question X**' header. Carefully read each question and provide thorough answers in the following text boxes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

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

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

Introduction

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



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

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

- [The Data](#)
- **STEP 1**: Acoustic Features for Speech Recognition
- **STEP 2**: Deep Neural Networks for Acoustic Modeling
 - [Model 0](#): RNN
 - [Model 1](#): RNN + TimeDistributed Dense
 - [Model 2](#): CNN + RNN + TimeDistributed Dense
 - [Model 3](#): Deeper RNN + TimeDistributed Dense
 - [Model 4](#): Bidirectional RNN + TimeDistributed Dense
 - [Models 5+](#)
 - [Compare the Models](#)
 - [Final Model](#)
- **STEP 3**: Obtain Predictions

The Data

We begin by investigating the dataset that will be used to train and evaluate your pipeline. [LibriSpeech](http://www.danielpovey.com/files/2015_icassp_librispeech.pdf) (http://www.danielpovey.com/files/2015_icassp_librispeech.pdf) is a large corpus of English-read speech, designed for training and evaluating models for ASR. The dataset contains 1000 hours of speech derived from audiobooks. We will work with a small subset in this project, since larger-scale data would take a long while to train. However, after completing this project, if you are interested in exploring further, you are encouraged to work with more of the data that is provided [online](http://www.openslr.org/12/) (<http://www.openslr.org/12/>).

In the code cells below, you will use the `vis_train_features` module to visualize a training example. The supplied argument `index=0` tells the module to extract the first example in the training set. (You are welcome to change `index=0` to point to a different training example, if you like, but please **DO NOT** amend any other code in the cell.) The returned variables are:

- `vis_text` - transcribed text (label) for the training example.
- `vis_raw_audio` - raw audio waveform for the training example.

- `vis_mfcc_feature` - mel-frequency cepstral coefficients (MFCCs) for the training example.
- `vis_spectrogram_feature` - spectrogram for the training example.
- `vis_audio_path` - the file path to the training example.

In [1]:

```
from data_generator import vis_train_features

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

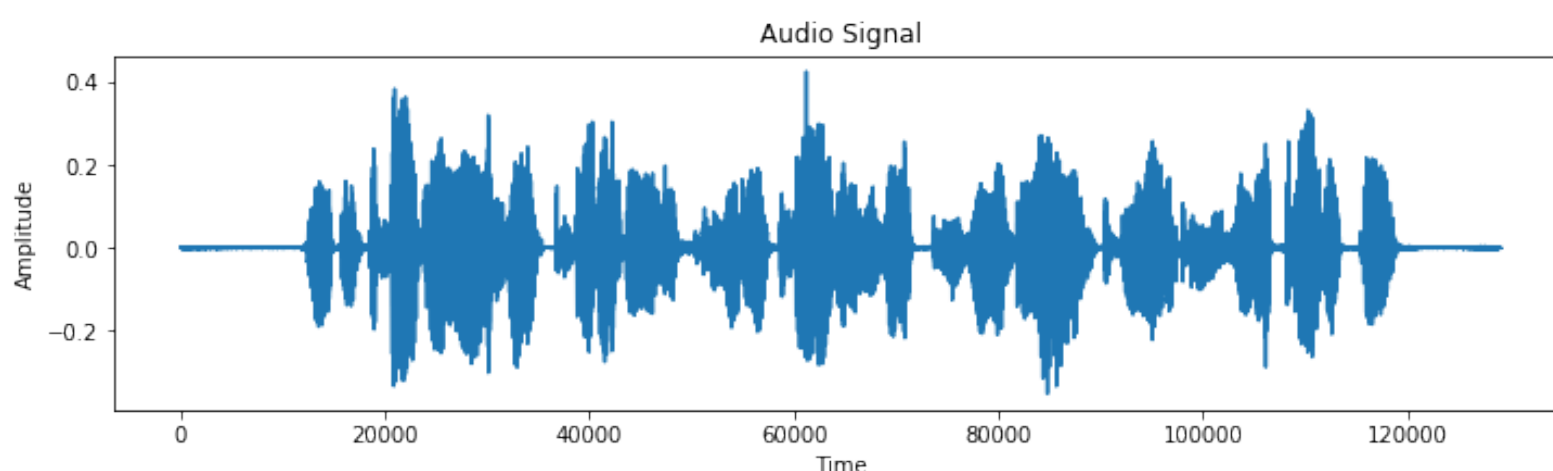
There are 2023 total training examples.

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

In [2]:

```
from IPython.display import Markdown, display
from data_generator import vis_train_features, plot_raw_audio
from IPython.display import Audio
%matplotlib inline

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



<IPython.core.display.Markdown object>

<IPython.core.display.Markdown object>

Out[2]:



STEP 1: Acoustic Features for Speech Recognition

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

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

Spectrograms

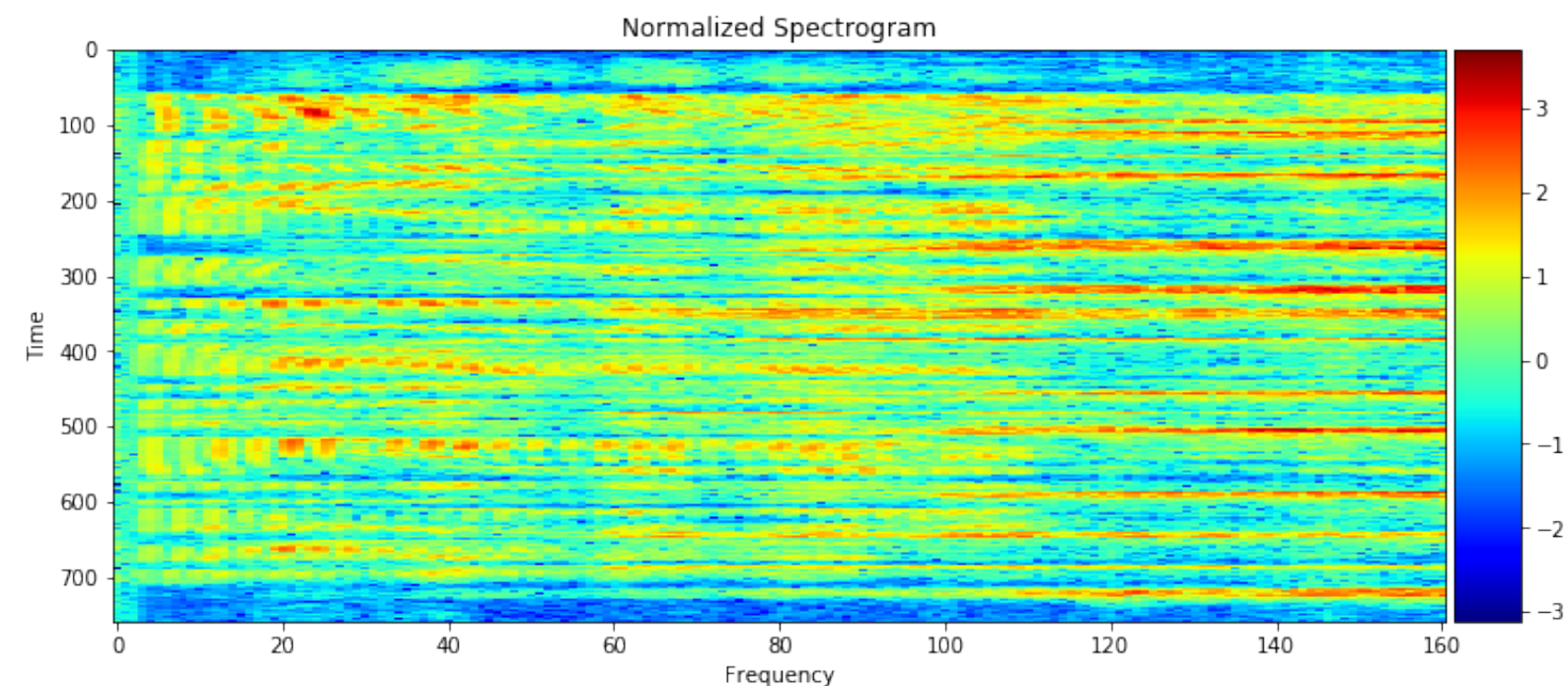
The first option for an audio feature representation is the spectrogram (https://www.youtube.com/watch?v=_FatxGN3vAM). In order to complete this project, you will **not** need to dig deeply into the details of how a spectrogram is calculated; but, if you are curious, the code for calculating the spectrogram 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)))
```



<IPython.core.display.Markdown object>

Mel-Frequency Cepstral Coefficients (MFCCs)

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

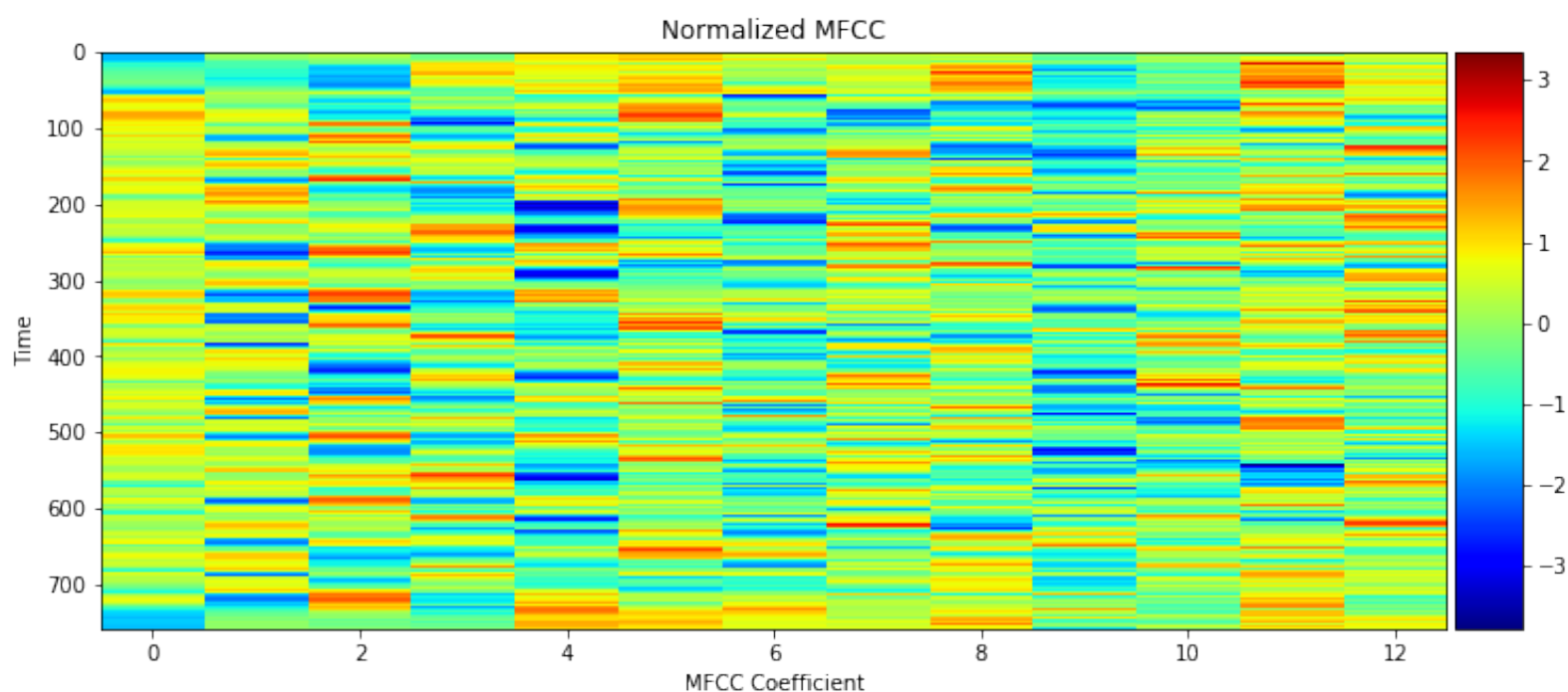
The main idea behind MFCC features is the same as spectrogram features: at each time window, the MFCC feature yields a feature vector that characterizes the sound within the window. Note that the MFCC feature is much lower-dimensional than the spectrogram feature, which could help an acoustic model to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

to the training dataset.

In [4]:

```
from data_generator import plot_mfcc_feature

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



<IPython.core.display.Markdown object>

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

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

STEP 2: Deep Neural Networks for Acoustic Modeling

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

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

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

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

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

In [3]:

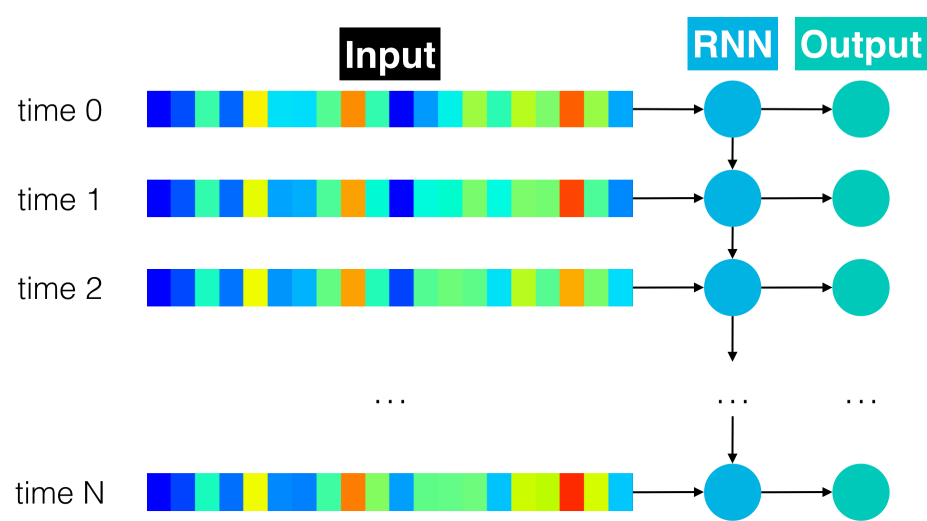
```
#####
# RUN THIS CODE CELL IF YOU ARE RESUMING THE NOTEBOOK AFTER A BREAK #
#####

# 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
```

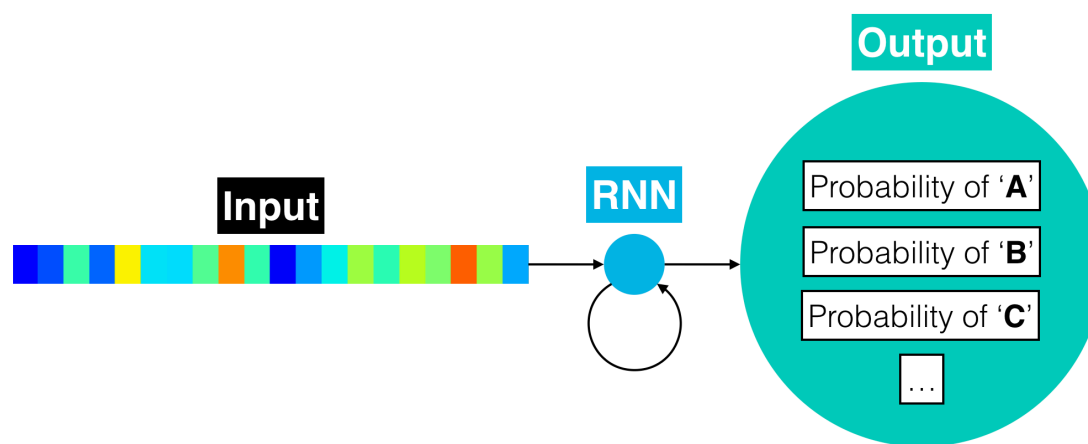
Model 0: RNN

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 [11]:

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

Layer (type)	Output Shape	Param #
the_input (InputLayer)	(None, None, 161)	0
rnn (GRU)	(None, None, 29)	16617
softmax (Activation)	(None, None, 29)	0

=====
Total params: 16,617
Trainable params: 16,617
Non-trainable params: 0
=====
None

As explored in the lesson, you will train the acoustic model with the CTC loss (http://www.cs.toronto.edu/~graves/icml_2006.pdf) criterion. Custom loss functions take a bit of hacking in Keras, and so we have implemented the CTC loss function for you, so that you can focus on trying out as many deep learning architectures as possible :). If you'd like to peek at the implementation details, look at the `add_ctc_loss` function within the `train_utils.py` file in the repository.

To train your architecture, you will use the `train_model` function within the `train_utils` module; it has already been imported in one of the above code cells. The `train_model` function takes three **required** arguments:

- `input_to_softmax` - a Keras model instance.
- `pickle_path` - the name of the pickle file where the loss history will be saved.
- `save_model_path` - the name of the HDF5 file where the model will be saved.

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

There are several **optional** arguments that allow you to have more control over the training process. You are welcome to, but not required to, supply your own values for these arguments.

- `minibatch_size` - the size of the minibatches that are generated while training the model (default: 20).
- `spectrogram` - Boolean value dictating whether spectrogram (True) or MFCC (False) features are used for training (default: True).
- `mfcc_dim` - the size of the feature dimension to use when generating MFCC features (default: 13).
- `optimizer` - the Keras optimizer used to train the model (default: `SGD(lr=0.02, decay=1e-6, momentum=0.9, nesterov=True, clipnorm=5)`).
- `epochs` - the number of epochs to use to train the model (default: 20). If you choose to modify this parameter, make sure that it is *at least* 20.
- `verbose` - controls the verbosity of the training output in the `model.fit_generator` method (default:

1).

- `sort_by_duration` - Boolean value dictating whether the training and validation sets are sorted by (increasing) duration before the start of the first epoch (default: `False`).

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

We have chosen to use GRU units in the supplied RNN. If you would like to experiment with LSTM or SimpleRNN cells, feel free to do so here. If you change the GRU units to SimpleRNN cells in `simple_rnn_model`, you may notice that the loss quickly becomes undefined (nan) - you are strongly encouraged to check this for yourself! This is due to the [exploding gradients problem](http://www.wildml.com/2015/10/recurrent-neural-networks-tutorial-part-3-backpropagation-through-time-and-vanishing-gradients/) (<http://www.wildml.com/2015/10/recurrent-neural-networks-tutorial-part-3-backpropagation-through-time-and-vanishing-gradients/>). We have already implemented [gradient clipping](https://arxiv.org/pdf/1211.5063.pdf) (<https://arxiv.org/pdf/1211.5063.pdf>) in your optimizer to help you avoid this issue.

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

In [12]:

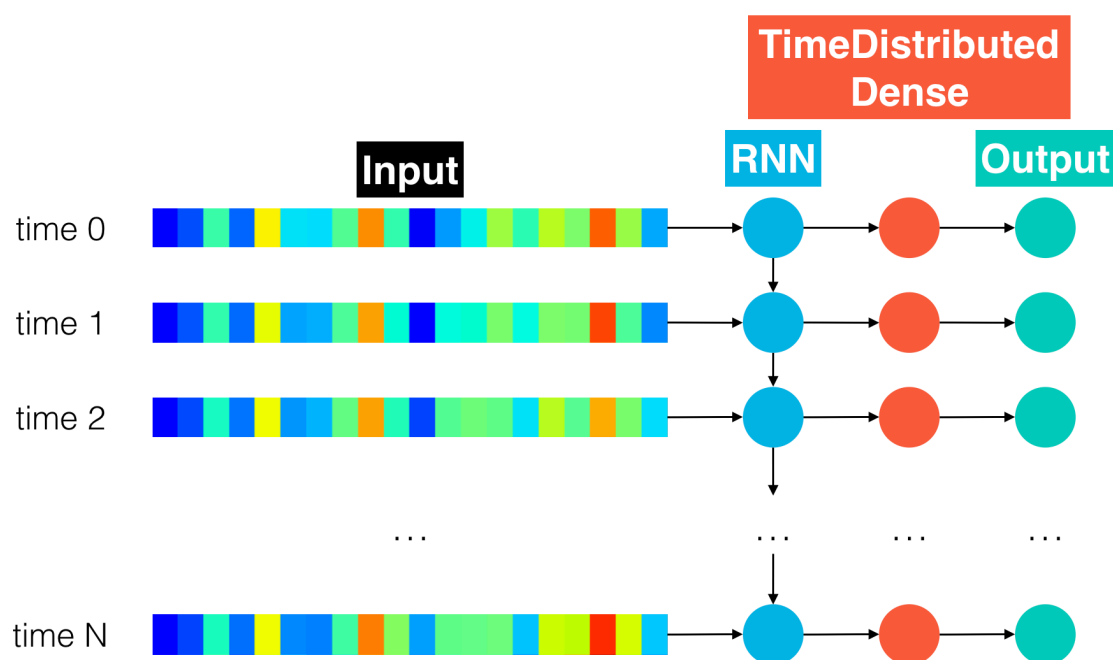
```
train_model(input_to_softmax=model_0,
            pickle_path='model_0.pickle',
            save_model_path='model_0.h5',
            spectrogram=True) # change to False if you would like to use MFCC features
```

```
Epoch 1/20
106/106 [=====] - 199s - loss: 837.9429 - val
_loss: 727.4035
Epoch 2/20
106/106 [=====] - 200s - loss: 751.0107 - val
_loss: 727.9180
Epoch 3/20
106/106 [=====] - 197s - loss: 752.1553 - val
_loss: 733.4376
Epoch 4/20
106/106 [=====] - 199s - loss: 751.2011 - val
_loss: 726.8646
Epoch 5/20
106/106 [=====] - 198s - loss: 750.5815 - val
_loss: 715.2178
Epoch 6/20
106/106 [=====] - 199s - loss: 750.8945 - val
_loss: 735.9968
Epoch 7/20
106/106 [=====] - 198s - loss: 752.0700 - val
_loss: 717.1776
Epoch 8/20
106/106 [=====] - 198s - loss: 750.4161 - val
```

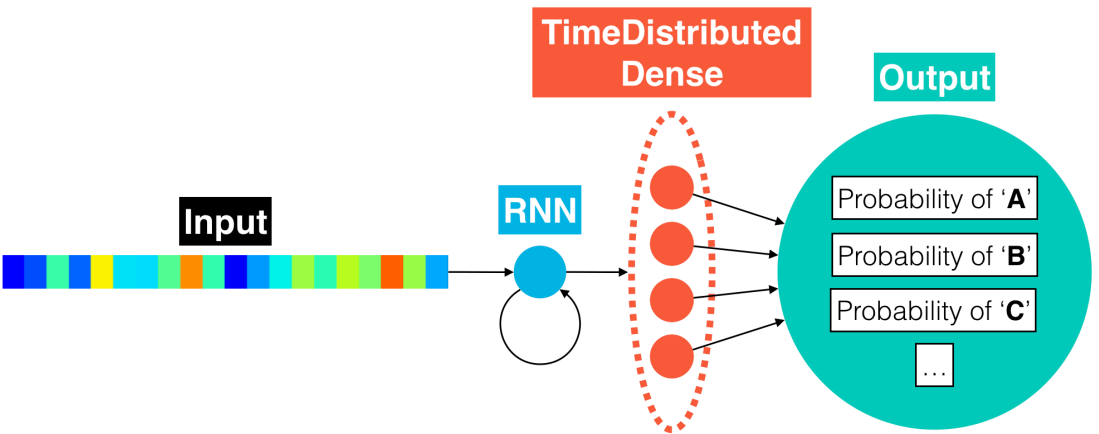
```
_loss: 732.7772
Epoch 9/20
106/106 [=====] - 198s - loss: 751.6664 - val
_loss: 721.1024
Epoch 10/20
106/106 [=====] - 199s - loss: 750.5927 - val
_loss: 728.8627
Epoch 11/20
106/106 [=====] - 198s - loss: 750.8361 - val
_loss: 724.9845
Epoch 12/20
106/106 [=====] - 197s - loss: 751.2437 - val
_loss: 723.4409
Epoch 13/20
106/106 [=====] - 199s - loss: 751.1496 - val
_loss: 727.2992
Epoch 14/20
106/106 [=====] - 197s - loss: 752.0769 - val
_loss: 726.2188
Epoch 15/20
106/106 [=====] - 197s - loss: 749.9494 - val
_loss: 726.7854
Epoch 16/20
106/106 [=====] - 197s - loss: 750.1326 - val
_loss: 729.1722
Epoch 17/20
106/106 [=====] - 198s - loss: 750.2834 - val
_loss: 726.2177
Epoch 18/20
106/106 [=====] - 199s - loss: 750.7481 - val
_loss: 729.5681
Epoch 19/20
106/106 [=====] - 199s - loss: 750.3988 - val
_loss: 722.2303
Epoch 20/20
106/106 [=====] - 196s - loss: 750.6582 - val
_loss: 726.0246
```

(IMPLEMENTATION) Model 1: RNN + TimeDistributed Dense

Read about the [TimeDistributed](https://keras.io/layers/wrappers/) (<https://keras.io/layers/wrappers/>) wrapper and the [BatchNormalization](https://keras.io/layers/normalization/) (<https://keras.io/layers/normalization/>) layer in the Keras documentation. For your next architecture, you will add [batch normalization](https://arxiv.org/pdf/1510.01378.pdf) (<https://arxiv.org/pdf/1510.01378.pdf>) to the recurrent layer to reduce training times. The `TimeDistributed` layer will be used to find more complex patterns in the dataset. The unrolled snapshot of the architecture is depicted below.



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



Use your research to complete the `rnn_model` function within the `sample_models.py` file. The function should specify an architecture that satisfies the following requirements:

- The first layer of the neural network should be an RNN (`SimpleRNN`, `LSTM`, or `GRU`) that takes the time sequence of audio features as input. We have added `GRU` units for you, but feel free to change `GRU` to `SimpleRNN` or `LSTM`, if you like!
- Whereas the architecture in `simple_rnn_model` treated the RNN output as the final layer of the model, you will use the output of your RNN as a hidden layer. Use `TimeDistributed` to apply a `Dense` layer to each of the time steps in the RNN output. Ensure that each `Dense` layer has `output_dim` units.

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

In [2]:

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

Layer (type)	Output Shape	Param #
the_input (InputLayer)	(None, None, 161)	0
rnn (GRU)	(None, None, 200)	217200
bn_simp_rnn (BatchNormalizat	(None, None, 200)	600
timedist (TimeDistributed)	(None, None, 29)	5829
softmax (Activation)	(None, None, 29)	0
Total params: 223,629		
Trainable params: 223,229		
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>) 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 [3]:

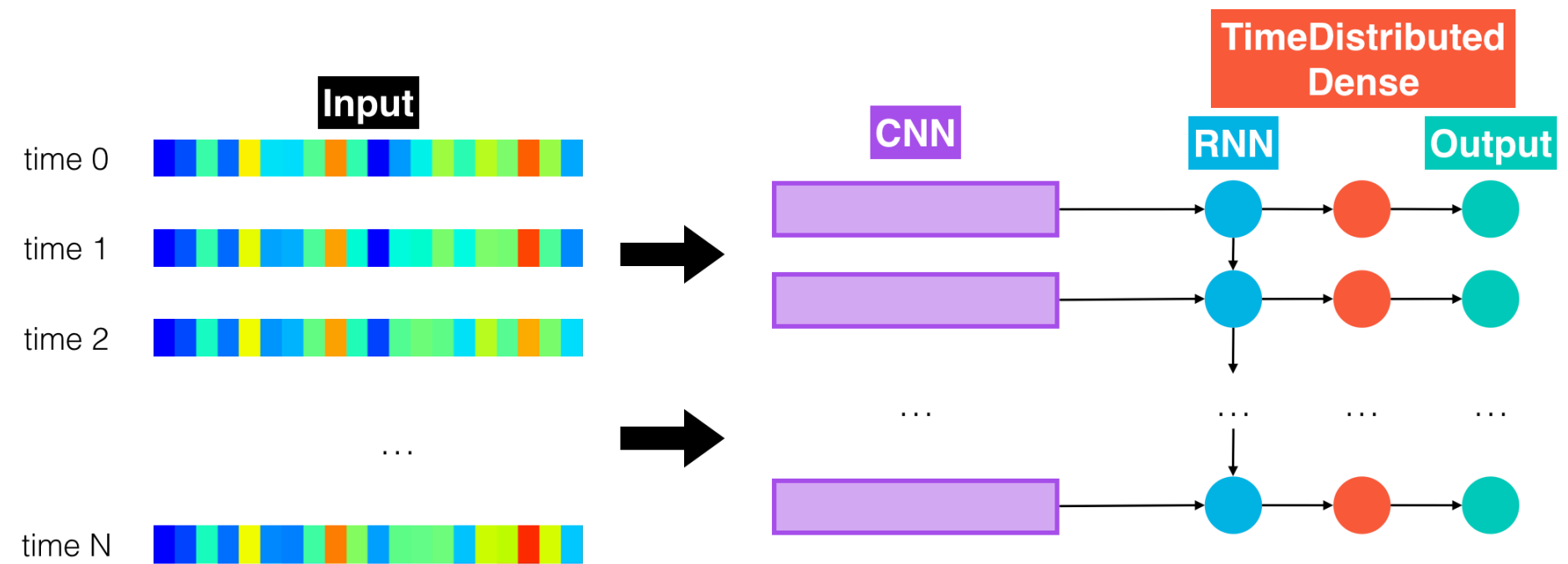
```
train_model(input_to_softmax=model_1,
            pickle_path='model_1.pickle',
            save_model_path='model_1.h5',
            spectrogram=True) # change to False if you would like to use MFCC features
```

```
Epoch 1/20
106/106 [=====] - 212s - loss: 272.2415 - val_loss: 252.6724
Epoch 2/20
106/106 [=====] - 198s - loss: 225.1904 - val_loss: 218.8738
Epoch 3/20
106/106 [=====] - 196s - loss: 218.8017 - val_loss: 206.6109
Epoch 4/20
106/106 [=====] - 195s - loss: 190.5303 - val_loss: 179.8811
```

```
Epoch 5/20
106/106 [=====] - 195s - loss: 170.3120 - val
_loss: 166.8822
Epoch 6/20
106/106 [=====] - 194s - loss: 158.1836 - val
_loss: 155.1859
Epoch 7/20
106/106 [=====] - 195s - loss: 151.0108 - val
_loss: 153.5003
Epoch 8/20
106/106 [=====] - 192s - loss: 146.3073 - val
_loss: 147.6480
Epoch 9/20
106/106 [=====] - 194s - loss: 142.8682 - val
_loss: 148.5094
Epoch 10/20
106/106 [=====] - 193s - loss: 141.8698 - val
_loss: 144.0977
Epoch 11/20
106/106 [=====] - 194s - loss: 138.7458 - val
_loss: 142.3574
Epoch 12/20
106/106 [=====] - 194s - loss: 135.4850 - val
_loss: 145.3829
Epoch 13/20
106/106 [=====] - 194s - loss: 134.9257 - val
_loss: 146.1889
Epoch 14/20
106/106 [=====] - 195s - loss: 135.1737 - val
_loss: 144.1117
Epoch 15/20
106/106 [=====] - 195s - loss: 132.5263 - val
_loss: 141.9348
Epoch 16/20
106/106 [=====] - 195s - loss: 131.3686 - val
_loss: 144.4337
Epoch 17/20
106/106 [=====] - 195s - loss: 131.6741 - val
_loss: 144.5315
Epoch 18/20
106/106 [=====] - 196s - loss: 131.0054 - val
_loss: 145.0428
Epoch 19/20
106/106 [=====] - 193s - loss: 129.3676 - val
_loss: 139.9224
Epoch 20/20
106/106 [=====] - 196s - loss: 129.5425 - val
_loss: 147.3098
```

(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 [9]:

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

Layer (type)	Output Shape	Param #
the_input (InputLayer)	(None, None, 161)	0
conv1d (Conv1D)	(None, None, 200)	354400
bn_conv_1d (BatchNormalizati	(None, None, 200)	800
rnn (SimpleRNN)	(None, None, 200)	80200
bn_simp_rnn (BatchNormalizat	(None, None, 200)	800
timedist (TimeDistributed)	(None, None, 29)	5829
softmax (Activation)	(None, None, 29)	0
Total params: 442,029		
Trainable params: 441,229		
Non-trainable params: 800		

None

Please execute the code cell below to train the neural network you specified in `input_to_softmax`. After the model has finished training, the model is saved (<https://keras.io/getting-started/faq/#how-can-i-save-a-keras-model>) in the HDF5 file `model_2.h5`. The loss history is saved (<https://wiki.python.org/moin/UsingPickle>) in `model_2.pickle`. You are welcome to tweak any of the optional parameters while calling the `train_model` function, but this is not required.

In [4]:

```
train_model(input_to_softmax=model_2,
            pickle_path='model_2.pickle',
            save_model_path='model_2.h5',
            spectrogram=True) # change to False if you would like to use MFCC featur
```

Epoch 1/20

106/106 [=====] - 110s - loss: 233.7630 - val

_loss: 218.8145

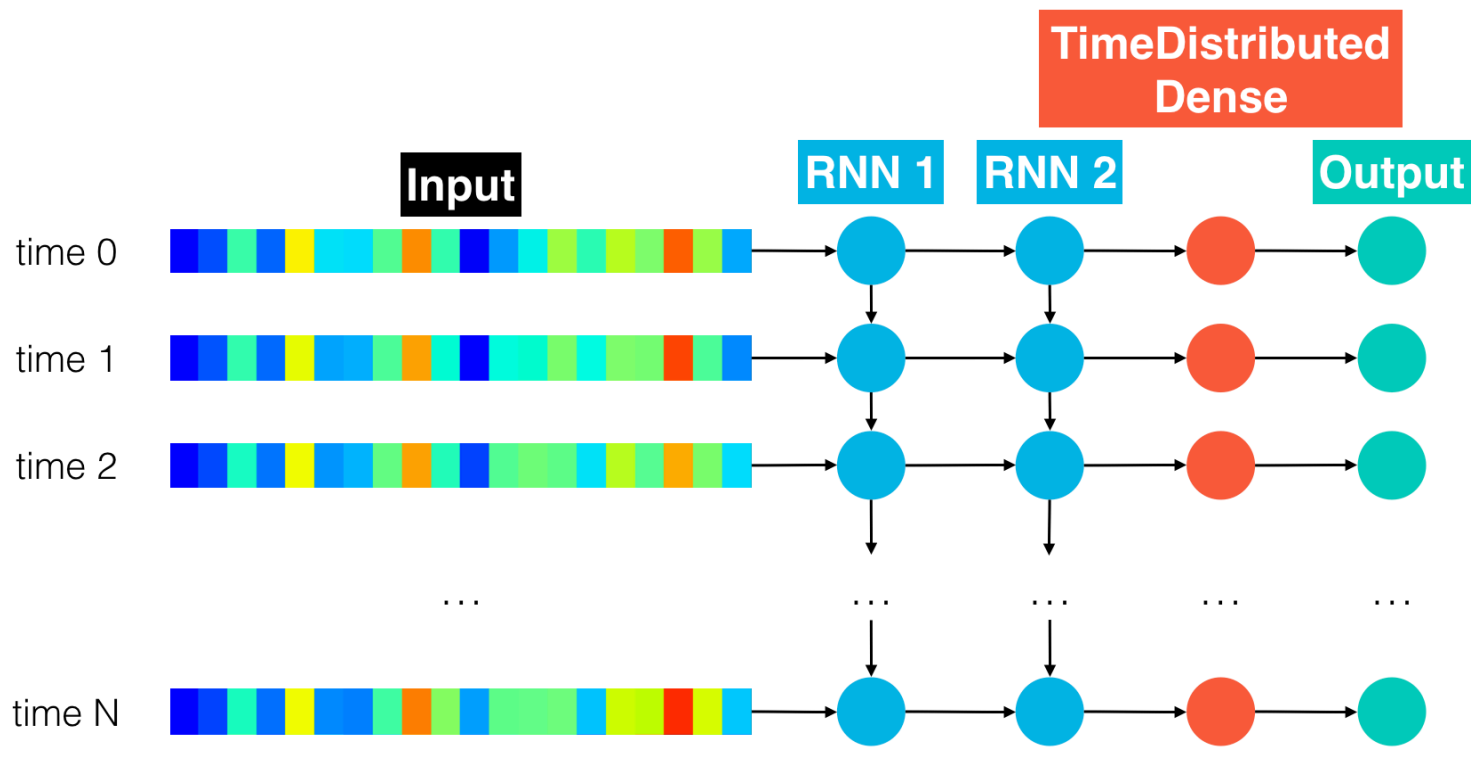
Epoch 2/20

106/106 [=====] - 50s - loss: 170.4182 - val_

loss: 157.9461
Epoch 3/20
106/106 [=====] - 50s - loss: 147.3617 - val_
loss: 143.7890
Epoch 4/20
106/106 [=====] - 50s - loss: 135.7705 - val_
loss: 139.8621
Epoch 5/20
106/106 [=====] - 50s - loss: 127.8209 - val_
loss: 138.0670
Epoch 6/20
106/106 [=====] - 49s - loss: 121.6368 - val_
loss: 132.0378
Epoch 7/20
106/106 [=====] - 49s - loss: 116.7462 - val_
loss: 129.6041
Epoch 8/20
106/106 [=====] - 50s - loss: 112.1723 - val_
loss: 131.2263
Epoch 9/20
106/106 [=====] - 49s - loss: 108.6944 - val_
loss: 132.1518
Epoch 10/20
106/106 [=====] - 49s - loss: 104.6465 - val_
loss: 132.4430
Epoch 11/20
106/106 [=====] - 49s - loss: 101.6462 - val_
loss: 128.6907
Epoch 12/20
106/106 [=====] - 49s - loss: 98.4104 - val_l
oss: 133.5354
Epoch 13/20
106/106 [=====] - 49s - loss: 95.9496 - val_l
oss: 134.8186
Epoch 14/20
106/106 [=====] - 49s - loss: 92.9457 - val_l
oss: 132.6184
Epoch 15/20
106/106 [=====] - 49s - loss: 89.8461 - val_l
oss: 132.4060
Epoch 16/20
106/106 [=====] - 49s - loss: 87.5273 - val_l
oss: 134.0788
Epoch 17/20
106/106 [=====] - 49s - loss: 85.7108 - val_l
oss: 138.9038
Epoch 18/20
106/106 [=====] - 49s - loss: 83.3296 - val_l
oss: 138.2181
Epoch 19/20
106/106 [=====] - 49s - loss: 80.6394 - val_l
oss: 139.8742
Epoch 20/20

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

Layer (type)	Output Shape	Param #
the_input (InputLayer)	(None, None, 161)	0
rnn (SimpleRNN)	(None, None, 200)	72400
bn_simp_rnn (BatchNormalizat	(None, None, 200)	800
rnn_2 (SimpleRNN)	(None, None, 200)	80200
bn_rnn_2 (BatchNormalization	(None, None, 200)	800
timedist (TimeDistributed)	(None, None, 29)	5829
softmax (Activation)	(None, None, 29)	0
Total params: 160,029		
Trainable params: 159,229		
Non-trainable params: 800		
None		

Please execute the code cell below to train the neural network you specified in input_to_softmax. After the model has finished training, the model is saved (<https://keras.io/getting-started/faq/#how-can-i-save-a-keras-model>) in the HDF5 file model_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 [3]:
train_model(input_to_softmax=model_3,
            pickle_path='model_3.pickle',
            save_model_path='model_3.h5',
            spectrogram=True) # change to False if you would like to use MFCC featur
```

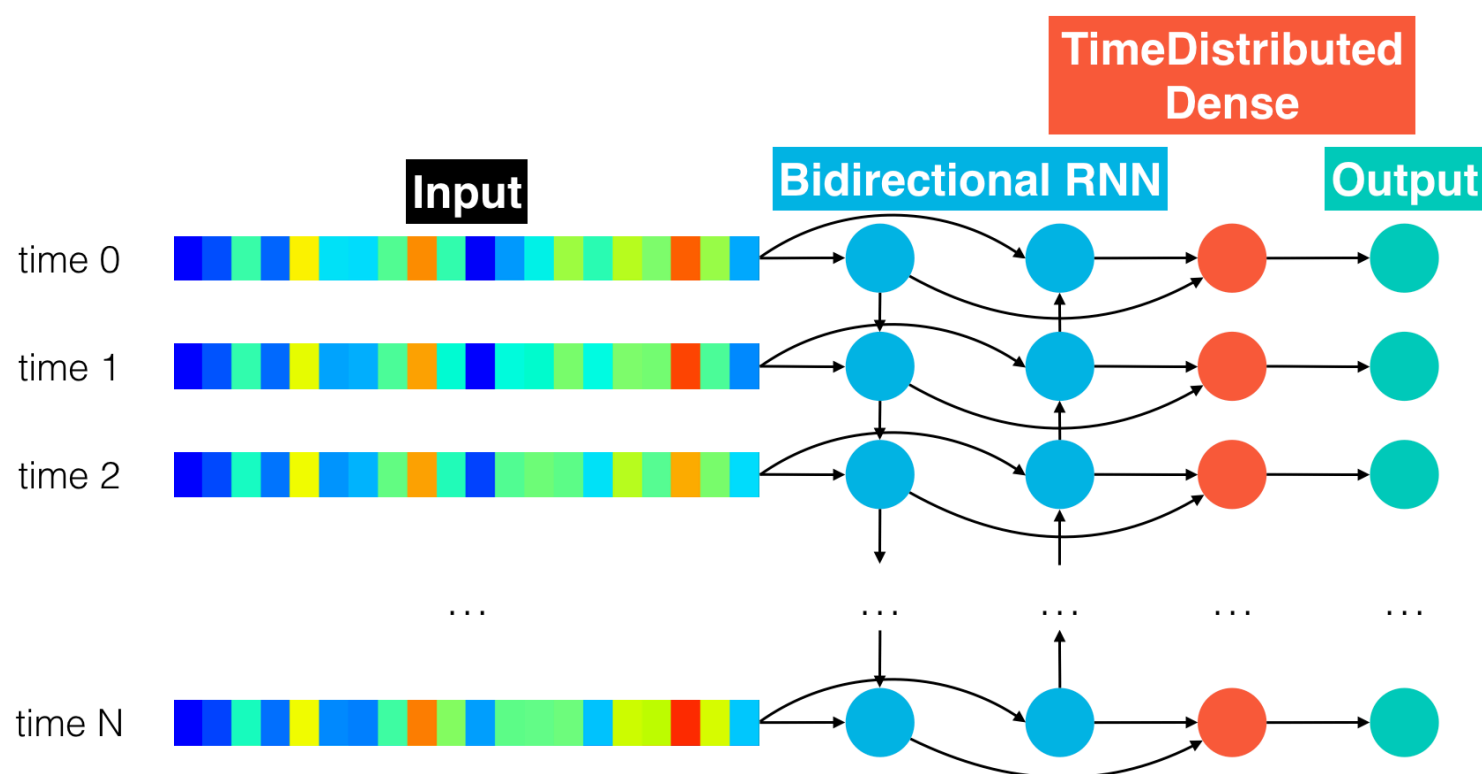
```
Epoch 1/20
106/106 [=====] - 287s - loss: 283.7298 - val
_loss: 232.4670
Epoch 2/20
106/106 [=====] - 290s - loss: 238.7994 - val
_loss: 239.9302
Epoch 3/20
106/106 [=====] - 291s - loss: 253.6563 - val
```

_loss: 217.0923
Epoch 4/20
106/106 [=====] - 288s - loss: 231.5879 - val
_loss: 209.4049
Epoch 5/20
106/106 [=====] - 290s - loss: 217.9090 - val
_loss: 207.1177
Epoch 6/20
106/106 [=====] - 289s - loss: 214.1400 - val
_loss: 199.9455
Epoch 7/20
106/106 [=====] - 288s - loss: 200.0583 - val
_loss: 187.7565
Epoch 8/20
106/106 [=====] - 289s - loss: 185.0693 - val
_loss: 176.5390
Epoch 9/20
106/106 [=====] - 290s - loss: 176.2793 - val
_loss: 169.1836
Epoch 10/20
106/106 [=====] - 291s - loss: 169.0741 - val
_loss: 162.8963
Epoch 11/20
106/106 [=====] - 292s - loss: 163.4966 - val
_loss: 161.6967
Epoch 12/20
106/106 [=====] - 292s - loss: 158.8893 - val
_loss: 156.6357
Epoch 13/20
106/106 [=====] - 291s - loss: 154.8503 - val
_loss: 157.2503
Epoch 14/20
106/106 [=====] - 291s - loss: 151.7182 - val
_loss: 153.8424
Epoch 15/20
106/106 [=====] - 291s - loss: 149.6728 - val
_loss: 148.8828
Epoch 16/20
106/106 [=====] - 291s - loss: 149.3257 - val
_loss: 154.3663
Epoch 17/20
106/106 [=====] - 292s - loss: 145.4884 - val
_loss: 149.4131
Epoch 18/20
106/106 [=====] - 289s - loss: 144.0371 - val
_loss: 149.5720
Epoch 19/20
106/106 [=====] - 291s - loss: 141.7089 - val
_loss: 147.1147
Epoch 20/20
106/106 [=====] - 291s - loss: 139.9915 - val
_loss: 146.6255

(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) (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 [9]:

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

Layer (type)	Output Shape	Param #
the_input (InputLayer)	(None, None, 161)	0
bidir_rnn (Bidirectional)	(None, None, 400)	144800
timedist (TimeDistributed)	(None, None, 29)	11629
softmax (Activation)	(None, None, 29)	0

Total params: 156,429
Trainable params: 156,429
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 [10]:

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

Epoch 1/20
106/106 [=====] - 113s - loss: 307.9029 - val_loss: 216.3605
Epoch 2/20
106/106 [=====] - 113s - loss: 202.6891 - val_loss: 187.4023
Epoch 3/20
106/106 [=====] - 113s - loss: 183.5768 - val_loss: 175.4280
Epoch 4/20
106/106 [=====] - 114s - loss: 171.5725 - val_loss: 168.2633
Epoch 5/20
106/106 [=====] - 113s - loss: 163.5857 - val_loss: 162.8504

```

Epoch 6/20
106/106 [=====] - 113s - loss: 157.0874 - val
_loss: 160.4659
Epoch 7/20
106/106 [=====] - 113s - loss: 151.8453 - val
_loss: 157.5478
Epoch 8/20
106/106 [=====] - 113s - loss: 147.4596 - val
_loss: 154.5372
Epoch 9/20
106/106 [=====] - 113s - loss: 143.4066 - val
_loss: 155.5816
Epoch 10/20
106/106 [=====] - 113s - loss: 139.4311 - val
_loss: 152.9025
Epoch 11/20
106/106 [=====] - 113s - loss: 136.1765 - val
_loss: 148.6617
Epoch 12/20
106/106 [=====] - 113s - loss: 133.1076 - val
_loss: 151.6950
Epoch 13/20
106/106 [=====] - 113s - loss: 130.2432 - val
_loss: 149.7601
Epoch 14/20
106/106 [=====] - 112s - loss: 128.9840 - val
_loss: 151.5885
Epoch 15/20
106/106 [=====] - 113s - loss: 125.5640 - val
_loss: 147.5727
Epoch 16/20
106/106 [=====] - 113s - loss: 123.4390 - val
_loss: 147.7949
Epoch 17/20
106/106 [=====] - 113s - loss: 121.3167 - val
_loss: 146.9530
Epoch 18/20
106/106 [=====] - 113s - loss: 119.4886 - val
_loss: 147.3251
Epoch 19/20
106/106 [=====] - 113s - loss: 117.6043 - val
_loss: 147.3308
Epoch 20/20
106/106 [=====] - 113s - loss: 116.0765 - val
_loss: 147.1201

```

(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.

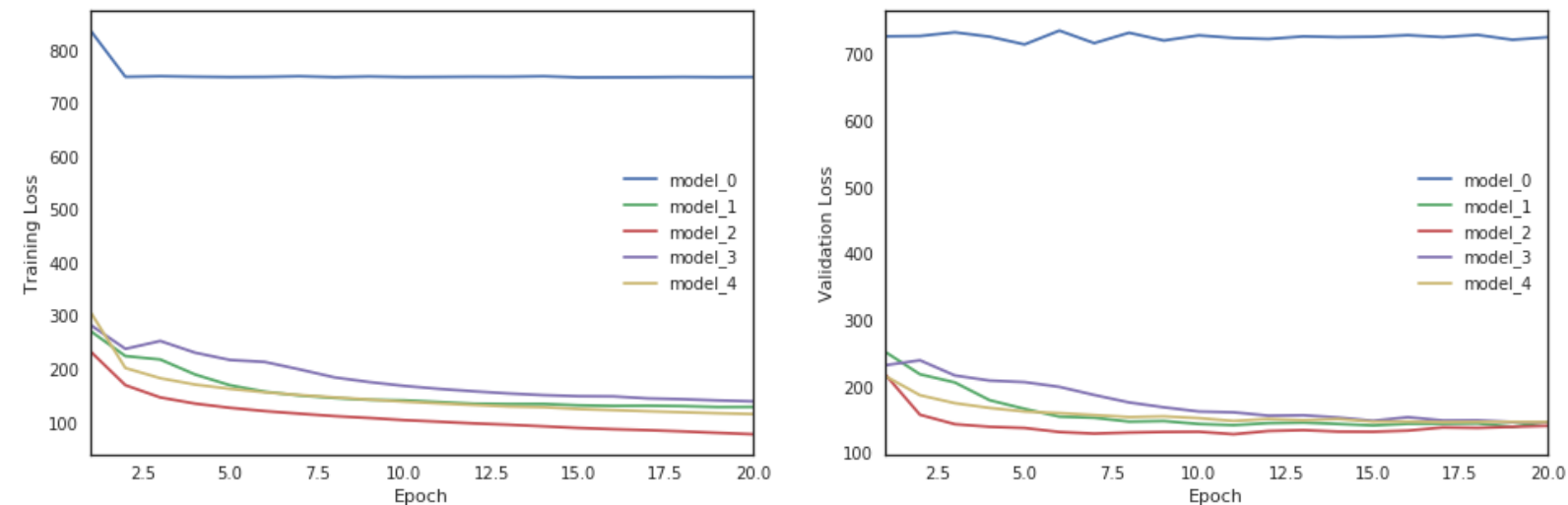
In [4]:

```
from glob import glob  
import numpy as np  
import _pickle as pickle  
import seaborn as sns  
import matplotlib.pyplot as plt  
%matplotlib inline  
sns.set_style(style='white')  
  
# obtain the paths for the saved model history  
all_pickles = sorted(glob("results/*.pickle"))  
# extract the name of each model  
model_names = [item[8:-7] for item in all_pickles]  
# extract the loss history for each model  
valid_loss = [pickle.load( open( i, "rb" ) )['val_loss'] for i in all_pickles]  
train_loss = [pickle.load( open( i, "rb" ) )['loss'] for i in all_pickles]  
# save the number of epochs used to train each model  
num_epochs = [len(valid_loss[i]) for i in range(len(valid_loss))]  
  
fig = plt.figure(figsize=(16,5))  
  
# plot the training loss vs. epoch for each model  
ax1 = fig.add_subplot(121)  
for i in range(len(all_pickles)):  
    ax1.plot(np.linspace(1, num_epochs[i], num_epochs[i]),  
            train_loss[i], label=model_names[i])  
# clean up the plot  
ax1.legend()  
ax1.set_xlim([1, max(num_epochs)])  
plt.xlabel('Epoch')  
plt.ylabel('Training Loss')  
  
# plot the validation loss vs. epoch for each model  
ax2 = fig.add_subplot(122)  
for i in range(len(all_pickles)):  
    ax2.plot(np.linspace(1, num_epochs[i], num_epochs[i]),  
            valid_loss[i], label=model_names[i])  
# clean up the plot  
ax2.legend()  
ax2.set_xlim([1, max(num_epochs)])
```



```
ax2.set_xlim([1, max(num_epochs)])
plt.xlabel('Epoch')
plt.ylabel('Validation Loss')
plt.show()
```

```
/home/aind2/anaconda3/envs/aind-vui/lib/python3.5/site-packages/matplotlib/font_manager.py:1297: UserWarning: findfont: Font family ['sans-serif'] not found. Falling back to DejaVu Sans
(prop.get_family(), self.defaultFamily[fonttext]))
```



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

Answer: (below)

Best performing: Model 2: CNN + RNN + TimeDistributed Dense

Model 0: RNN

Best validation loss: 715.2178

In this model, the RNN is only looking at the previous timestep's output in addition to the input. With this limited information, there is little for the model to work with and thus it is not able to make very good predictions.

Model 1: RNN + TimeDistributed Dense

Best validation loss: 139.9224

The RNN layer allows the network to include information from all the previous time steps. The TimeDistributed layer allows the network to find more complex patterns in the dataset which is very helpful given the complexity of the speech. This give much better performance than model 0.

Model 2: CNN + RNN + TimeDistributed Dense

Best validation loss: 128.6907

This network is just as accurate as Model 1 but it is much faster to train. Where Model 1 took 195 seconds per epoch, this model only took 50 seconds per epoch—25% of the time. This would be a huge advantage when training on the full LibriSpeech dataset.

Model 3: Deeper RNN + TimeDistributed Dense

Best validation loss: 146.6255

This model did slightly worse than Model 1. (I also tried 3 layers but from the first several epochs I could tell that the performance would not improve and terminated the training.) It seems that the additional layer or RNN does not benefit.

Model 4: Bidirectional RNN + TimeDistributed Dense

Best validation loss: 146.9530

The bidirectional RNN performs much better than the Deeper RNN (Model 3). It is as accurate as Model 1 but the training times are much better (113 secs per epoch vs. 195 secs per epoch).

(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 []:

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

In [8]:

```
model_end = wavenet_model(input_dim=13)
```

Layer (type)	Output Shape	Param #	Connected to
=====			
the_input (InputLayer)	(None, None, 13)	0	
=====			
conv_in (Conv1D)	(None, None, 128)	1792	the_input[0][0]
=====			
bn_conv_in (BatchNormalization)	(None, None, 128)	512	conv_in[0][0]
=====			
block_0_1_conv_tanh (Conv1D)	(None, None, 128)	114816	bn_conv_in[0][0]
=====			
block_0_1_conv_sigmoid (Conv1D)	(None, None, 128)	114816	bn_conv_in[0][0]
=====			

block_0_1_bn_tanh_out (BatchNorm)	(None, None, 128)	512	block_0_1_conv_tanh[0][0]
block_0_1_bn_sigmoid_out (BatchNorm)	(None, None, 128)	512	block_0_1_conv_sigmoid[0][0]
block_0_1_multi (Multiply)	(None, None, 128)	0	block_0_1_bn_tanh_out[0][0]
block_0_1_bn_sigmoid_out[0][0]			
block_0_1_conv_out (Conv1D)	(None, None, 128)	16512	block_0_1_multi[0][0]
block_0_1_bn_skip_out (BatchNorm)	(None, None, 128)	512	block_0_1_conv_out[0][0]
block_0_1_add (Add)	(None, None, 128)	0	block_0_1_bn_skip_out[0][0]
bn_conv_in[0][0]			
block_0_2_conv_tanh (Conv1D)	(None, None, 128)	114816	block_0_1_add[0][0]
block_0_2_conv_sigmoid (Conv1D)	(None, None, 128)	114816	block_0_1_conv_tanh[0][0]
block_0_2_bn_tanh_out (BatchNorm)	(None, None, 128)	512	block_0_2_conv_sigmoid[0][0]
block_0_2_bn_sigmoid_out (BatchNorm)	(None, None, 128)	512	block_0_2_bn_tanh_out[0][0]
block_0_2_multi (Multiply)	(None, None, 128)	0	block_0_2_bn_sigmoid_out[0][0]
block_0_2_bn_sigmoid_out[0][0]			
block_0_2_conv_out (Conv1D)	(None, None, 128)	16512	block_0_2_multi[0][0]

block_0_2_bn_skip_out (BatchNorm (None, None, 128)	512	blo
ck_0_2_conv_out[0][0]		
block_0_2_add (Add)	(None, None, 128)	0
ck_0_2_bn_skip_out[0][0]		blo
block_0_1_add[0][0]		
block_0_4_conv_tanh (Conv1D)	(None, None, 128)	114816
ck_0_2_add[0][0]		blo
block_0_4_conv_sigmoid (Conv1D)	(None, None, 128)	114816
ck_0_2_add[0][0]		blo
block_0_4_bn_tanh_out (BatchNorm (None, None, 128)	512	blo
ck_0_4_conv_tanh[0][0]		
block_0_4_bn_sigmoid_out (BatchN (None, None, 128)	512	blo
ck_0_4_conv_sigmoid[0][0]		
block_0_4_multi (Multiply)	(None, None, 128)	0
ck_0_4_bn_tanh_out[0][0]		blo
block_0_4_bn_sigmoid_out[0][0]		
block_0_4_conv_out (Conv1D)	(None, None, 128)	16512
ck_0_4_multi[0][0]		blo
block_0_4_bn_skip_out (BatchNorm (None, None, 128)	512	blo
ck_0_4_conv_out[0][0]		
block_0_4_add (Add)	(None, None, 128)	0
ck_0_4_bn_skip_out[0][0]		blo
block_0_2_add[0][0]		
block_0_8_conv_tanh (Conv1D)	(None, None, 128)	114816
ck_0_4_add[0][0]		blo
block_0_8_conv_sigmoid (Conv1D)	(None, None, 128)	114816
		blo

ck_0_4_add[0][0]			
block_0_8_bn_tanh_out (BatchNorm (None, None, 128) 512 blo			
ck_0_8_conv_tanh[0][0]			
block_0_8_bn_sigmoid_out (BatchN (None, None, 128) 512 blo			
ck_0_8_conv_sigmoid[0][0]			
block_0_8_multi (Multiply) (None, None, 128) 0 blo			
ck_0_8_bn_tanh_out[0][0]			
block_0_8_bn_sigmoid_out[0][0]			
block_0_8_conv_out (Conv1D) (None, None, 128) 16512 blo			
ck_0_8_multi[0][0]			
block_0_8_bn_skip_out (BatchNorm (None, None, 128) 512 blo			
ck_0_8_conv_out[0][0]			
block_0_8_add (Add) (None, None, 128) 0 blo			
ck_0_8_bn_skip_out[0][0]			
block_0_4_add[0][0]			
block_0_16_conv_tanh (Conv1D) (None, None, 128) 114816 blo			
ck_0_8_add[0][0]			
block_0_16_conv_sigmoid (Conv1D) (None, None, 128) 114816 blo			
ck_0_8_add[0][0]			
block_0_16_bn_tanh_out (BatchNor (None, None, 128) 512 blo			
ck_0_16_conv_tanh[0][0]			
block_0_16_bn_sigmoid_out (Batch (None, None, 128) 512 blo			
ck_0_16_conv_sigmoid[0][0]			
block_0_16_multi (Multiply) (None, None, 128) 0 blo			
ck_0_16_bn_tanh_out[0][0]			
block_0_16_bn_sigmoid_out[0][0]			

block_0_16_conv_out (Conv1D)	(None, None, 128)	16512	block_0_16_multi[0][0]
block_0_16_bn_skip_out (BatchNorm)	(None, None, 128)	512	block_0_16_conv_out[0][0]
block_0_16_add (Add)	(None, None, 128)	0	block_0_16_bn_skip_out[0][0]
block_0_8_add[0][0]			
block_1_1_conv_tanh (Conv1D)	(None, None, 128)	114816	block_0_16_add[0][0]
block_1_1_conv_sigmoid (Conv1D)	(None, None, 128)	114816	block_0_16_add[0][0]
block_1_1_bn_tanh_out (BatchNorm)	(None, None, 128)	512	block_1_1_conv_tanh[0][0]
block_1_1_bn_sigmoid_out (BatchNorm)	(None, None, 128)	512	block_1_1_conv_sigmoid[0][0]
block_1_1_multi (Multiply)	(None, None, 128)	0	block_1_1_bn_tanh_out[0][0]
block_1_1_bn_sigmoid_out[0][0]			
block_1_1_conv_out (Conv1D)	(None, None, 128)	16512	block_1_1_multi[0][0]
block_1_1_bn_skip_out (BatchNorm)	(None, None, 128)	512	block_1_1_conv_out[0][0]
block_1_1_add (Add)	(None, None, 128)	0	block_1_1_bn_skip_out[0][0]
block_0_16_add[0][0]			
block_1_2_conv_tanh (Conv1D)	(None, None, 128)	114816	block_1_1_add[0][0]

block_1_2_conv_sigmoid ck_1_1_add[0][0]	(Conv1D) (None, None, 128)	114816	blo
block_1_2_bn_tanh_out ck_1_2_conv_tanh[0][0]	(BatchNorm (None, None, 128)	512	blo
block_1_2_bn_sigmoid_out ck_1_2_conv_sigmoid[0][0]	(BatchN (None, None, 128)	512	blo
block_1_2_multi ck_1_2_bn_tanh_out[0][0]	(Multiply) (None, None, 128)	0	blo
block_1_2_bn_sigmoid_out[0][0]			
block_1_2_conv_out ck_1_2_multi[0][0]	(Conv1D) (None, None, 128)	16512	blo
block_1_2_bn_skip_out ck_1_2_conv_out[0][0]	(BatchNorm (None, None, 128)	512	blo
block_1_2_add ck_1_2_bn_skip_out[0][0]	(Add) (None, None, 128)	0	blo
block_1_1_add[0][0]			
block_1_4_conv_tanh ck_1_2_add[0][0]	(Conv1D) (None, None, 128)	114816	blo
block_1_4_conv_sigmoid ck_1_2_add[0][0]	(Conv1D) (None, None, 128)	114816	blo
block_1_4_bn_tanh_out ck_1_4_conv_tanh[0][0]	(BatchNorm (None, None, 128)	512	blo
block_1_4_bn_sigmoid_out ck_1_4_conv_sigmoid[0][0]	(BatchN (None, None, 128)	512	blo
block_1_4_multi ck_1_4_bn_tanh_out[0][0]	(Multiply) (None, None, 128)	0	blo
block_1_4_bn_sigmoid_out[0][0]			

block_1_4_conv_out (Conv1D)	(None, None, 128)	16512	block_1_4_multi[0][0]
block_1_4_bn_skip_out (BatchNorm)	(None, None, 128)	512	block_1_4_conv_out[0][0]
block_1_4_add (Add)	(None, None, 128)	0	block_1_4_bn_skip_out[0][0]
			block_1_2_add[0][0]
block_1_8_conv_tanh (Conv1D)	(None, None, 128)	114816	block_1_4_add[0][0]
block_1_8_conv_sigmoid (Conv1D)	(None, None, 128)	114816	block_1_4_add[0][0]
block_1_8_bn_tanh_out (BatchNorm)	(None, None, 128)	512	block_1_8_conv_tanh[0][0]
block_1_8_bn_sigmoid_out (BatchNorm)	(None, None, 128)	512	block_1_8_conv_sigmoid[0][0]
block_1_8_multi (Multiply)	(None, None, 128)	0	block_1_8_bn_tanh_out[0][0]
			block_1_8_bn_sigmoid_out[0][0]
block_1_8_conv_out (Conv1D)	(None, None, 128)	16512	block_1_8_multi[0][0]
block_1_8_bn_skip_out (BatchNorm)	(None, None, 128)	512	block_1_8_conv_out[0][0]
block_1_8_add (Add)	(None, None, 128)	0	block_1_8_bn_skip_out[0][0]
			block_1_4_add[0][0]
block_1_16_conv_tanh (Conv1D)	(None, None, 128)	114816	

ck_1_8_add[0][0]			
block_1_16_conv_sigmoid (Conv1D)	(None, None, 128)	114816	block_1_8_add[0][0]
ck_1_16_conv_tanh[0][0]			
block_1_16_bn_tanh_out (BatchNor	(None, None, 128)	512	block_1_16_conv_tanh[0][0]
ck_1_16_conv_sigmoid[0][0]			
block_1_16_bn_sigmoid_out (Batch	(None, None, 128)	512	block_1_16_conv_sigmoid[0][0]
ck_1_16_bn_tanh_out[0][0]			
block_1_16_multi (Multiply)	(None, None, 128)	0	block_1_16_bn_tanh_out[0][0]
ck_1_16_bn_sigmoid_out[0][0]			
block_1_16_conv_out (Conv1D)	(None, None, 128)	16512	block_1_16_multi[0][0]
ck_1_16_bn_skip_out[0][0]			
block_1_16_bn_skip_out (BatchNor	(None, None, 128)	512	block_1_16_conv_out[0][0]
ck_1_16_add[0][0]			
block_1_16_add (Add)	(None, None, 128)	0	block_1_16_bn_skip_out[0][0]
ck_1_16_add[0][0]			
block_2_1_conv_tanh (Conv1D)	(None, None, 128)	114816	block_1_8_add[0][0]
ck_1_16_add[0][0]			
block_2_1_conv_sigmoid (Conv1D)	(None, None, 128)	114816	block_2_1_conv_tanh[0][0]
ck_1_16_add[0][0]			
block_2_1_bn_tanh_out (BatchNorm	(None, None, 128)	512	block_2_1_conv_sigmoid[0][0]
ck_2_1_conv_tanh[0][0]			
block_2_1_bn_sigmoid_out (BatchN	(None, None, 128)	512	block_2_1_bn_tanh_out[0][0]
ck_2_1_conv_sigmoid[0][0]			
block_2_1_multi (Multiply)	(None, None, 128)	0	block_2_1_bn_sigmoid_out[0][0]
ck_2_1_bn_tanh_out[0][0]			

block_2_1_bn_sigmoid_out[0][0]			
block_2_1_conv_out (Conv1D)	(None, None, 128)	16512	blo
ck_2_1_multi[0][0]			
block_2_1_bn_skip_out (BatchNorm	(None, None, 128)	512	blo
ck_2_1_conv_out[0][0]			
block_2_1_add (Add)	(None, None, 128)	0	blo
ck_2_1_bn_skip_out[0][0]			
block_1_16_add[0][0]			
block_2_2_conv_tanh (Conv1D)	(None, None, 128)	114816	blo
ck_2_1_add[0][0]			
block_2_2_conv_sigmoid (Conv1D)	(None, None, 128)	114816	blo
ck_2_1_add[0][0]			
block_2_2_bn_tanh_out (BatchNorm	(None, None, 128)	512	blo
ck_2_2_conv_tanh[0][0]			
block_2_2_bn_sigmoid_out (BatchN	(None, None, 128)	512	blo
ck_2_2_conv_sigmoid[0][0]			
block_2_2_multi (Multiply)	(None, None, 128)	0	blo
ck_2_2_bn_tanh_out[0][0]			
block_2_2_bn_sigmoid_out[0][0]			
block_2_2_conv_out (Conv1D)	(None, None, 128)	16512	blo
ck_2_2_multi[0][0]			
block_2_2_bn_skip_out (BatchNorm	(None, None, 128)	512	blo
ck_2_2_conv_out[0][0]			
block_2_2_add (Add)	(None, None, 128)	0	blo
ck_2_2_bn_skip_out[0][0]			
block_2_1_add[0][0]			

block_2_4_conv_tanh (Conv1D) ck_2_2_add[0][0]	(None, None, 128)	114816	blo
block_2_4_conv_sigmoid (Conv1D) ck_2_2_add[0][0]	(None, None, 128)	114816	blo
block_2_4_bn_tanh_out (BatchNorm ck_2_4_conv_tanh[0][0]	(None, None, 128)	512	blo
block_2_4_bn_sigmoid_out (BatchN ck_2_4_conv_sigmoid[0][0]	(None, None, 128)	512	blo
block_2_4_multi (Multiply) ck_2_4_bn_tanh_out[0][0]	(None, None, 128)	0	blo
block_2_4_bn_sigmoid_out[0][0]			
block_2_4_conv_out (Conv1D) ck_2_4_multi[0][0]	(None, None, 128)	16512	blo
block_2_4_bn_skip_out (BatchNorm ck_2_4_conv_out[0][0]	(None, None, 128)	512	blo
block_2_4_add (Add) ck_2_4_bn_skip_out[0][0]	(None, None, 128)	0	blo
block_2_2_add[0][0]			
block_2_8_conv_tanh (Conv1D) ck_2_4_add[0][0]	(None, None, 128)	114816	blo
block_2_8_conv_sigmoid (Conv1D) ck_2_4_add[0][0]	(None, None, 128)	114816	blo
block_2_8_bn_tanh_out (BatchNorm ck_2_8_conv_tanh[0][0]	(None, None, 128)	512	blo
block_2_8_bn_sigmoid_out (BatchN ck_2_8_conv_sigmoid[0][0]	(None, None, 128)	512	blo

block_2_8_multi (Multiply)	(None, None, 128)	0	blo
ck_2_8_bn_tanh_out[0][0]			
block_2_8_bn_sigmoid_out[0][0]			
block_2_8_conv_out (Conv1D)	(None, None, 128)	16512	blo
ck_2_8_multi[0][0]			
block_2_8_bn_skip_out (BatchNorm	(None, None, 128)	512	blo
ck_2_8_conv_out[0][0]			
block_2_8_add (Add)	(None, None, 128)	0	blo
ck_2_8_bn_skip_out[0][0]			
block_2_4_add[0][0]			
block_2_16_conv_tanh (Conv1D)	(None, None, 128)	114816	blo
ck_2_8_add[0][0]			
block_2_16_conv_sigmoid (Conv1D)	(None, None, 128)	114816	blo
ck_2_8_add[0][0]			
block_2_16_bn_tanh_out (BatchNor	(None, None, 128)	512	blo
ck_2_16_conv_tanh[0][0]			
block_2_16_bn_sigmoid_out (Batch	(None, None, 128)	512	blo
ck_2_16_conv_sigmoid[0][0]			
block_2_16_multi (Multiply)	(None, None, 128)	0	blo
ck_2_16_bn_tanh_out[0][0]			
block_2_16_bn_sigmoid_out[0][0]			
block_2_16_conv_out (Conv1D)	(None, None, 128)	16512	blo
ck_2_16_multi[0][0]			
block_2_16_bn_skip_out (BatchNor	(None, None, 128)	512	blo
ck_2_16_conv_out[0][0]			
skip_sum (Add)	(None, None, 128)	0	blo
ck_0_1_bn_skip_out[0][0]			

```
block_0_2_bn_skip_out[0][0]

block_0_4_bn_skip_out[0][0]

block_0_8_bn_skip_out[0][0]

block_0_16_bn_skip_out[0][0]

block_1_1_bn_skip_out[0][0]

block_1_2_bn_skip_out[0][0]

block_1_4_bn_skip_out[0][0]

block_1_8_bn_skip_out[0][0]

block_1_16_bn_skip_out[0][0]

block_2_1_bn_skip_out[0][0]

block_2_2_bn_skip_out[0][0]

block_2_4_bn_skip_out[0][0]

block_2_8_bn_skip_out[0][0]

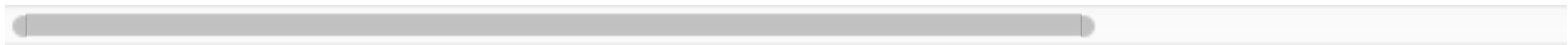
block_2_16_bn_skip_out[0][0]

conv_1 (Conv1D)                (None, None, 128)        16512        ski
p_sum[0][0]

bn_conv_1 (BatchNormalization) (None, None, 128)        512          con
v_1[0][0]

conv_2 (Conv1D)                (None, None, 29)         3741         bn_
conv_1[0][0]
=====
=====
Total params: 3,738,269
Trainable params: 3,726,237
Non-trainable params: 12,032

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 []:

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

```
Epoch 1/20
106/106 [=====] - 2547s - loss: 290.9299 - val_loss: 220.5923
Epoch 2/20
106/106 [=====] - 2569s - loss: 236.6306 - val_loss: 215.3467
Epoch 3/20
106/106 [=====] - 2557s - loss: 232.3798 - val_loss: 218.0521
Epoch 4/20
106/106 [=====] - 2562s - loss: 226.3389 - val_loss: 214.5569
Epoch 5/20
106/106 [=====] - 2567s - loss: 218.7075 - val_loss: 204.6410
Epoch 6/20
106/106 [=====] - 2568s - loss: 199.5556 - val_loss: 185.5145
Epoch 7/20
106/106 [=====] - 2547s - loss: 171.3264 - val_loss: 161.2014
Epoch 8/20
106/106 [=====] - 2562s - loss: 148.6898 - val_loss: 143.7042
Epoch 9/20
106/106 [=====] - 2566s - loss: 132.0965 - val_loss: 132.8907
Epoch 10/20
106/106 [=====] - 2553s - loss: 119.8539 - val_loss: 128.3805
Epoch 11/20
106/106 [=====] - 2565s - loss: 109.6234 - val_loss: 124.1635
Epoch 12/20
106/106 [=====] - 2575s - loss: 100.2456 - val_loss: 123.9358
Epoch 13/20
106/106 [=====] - 2557s - loss: 90.5858 - val_loss: 125.0632
Epoch 14/20
106/106 [=====] - 2550s - loss: 79.8350 - val_loss: 125.0632
```

```
_loss: 128.2734
Epoch 15/20
106/106 [=====] - 2581s - loss: 70.3025 - val
_loss: 130.0789
Epoch 16/20
106/106 [=====] - 2597s - loss: 60.7534 - val
_loss: 138.2626
Epoch 17/20
106/106 [=====] - 2552s - loss: 51.8431 - val
_loss: 143.2233
Epoch 18/20
106/106 [=====] - 2551s - loss: 43.9596 - val
_loss: 155.1319
Epoch 19/20
106/106 [=====] - 2563s - loss: 37.5090 - val
_loss: 161.7878
Epoch 20/20
106/106 [=====] - 2596s - loss: 32.3488 - val
_loss: 173.5343
```

In []:

In []:

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

Answer: (below)

Final Model Architecture

My final model is a simplified implementation of speech recognition based on DeepMind's WaveNet architecture. It is an adaptation of Kim and Park's Speech-to-Text-WaveNet implementation (<https://github.com/buriburisuri/> (<https://github.com/buriburisuri/>)). I refactored the code to use only Keras instead of sugartensor and made some changes to get it to work with our project code.

There are four main parts of the architecture: the front layer, stack of dilated casual convolutional layers, the residual block, and the final layers.

WaveNet Concept

Deep Mind's WaveNet is a deep neural network for generating raw audio waveforms. [1609.03499 WaveNet: A Generative Model for Raw Audio](https://arxiv.org/abs/1609.03499) (<https://arxiv.org/abs/1609.03499>)

WaveNet is based on dilated causal convolutions in order to deal with long-range temporal dependencies needed for raw audio generation. These dilated causal convolutions allow the network to economically have very large receptive fields.

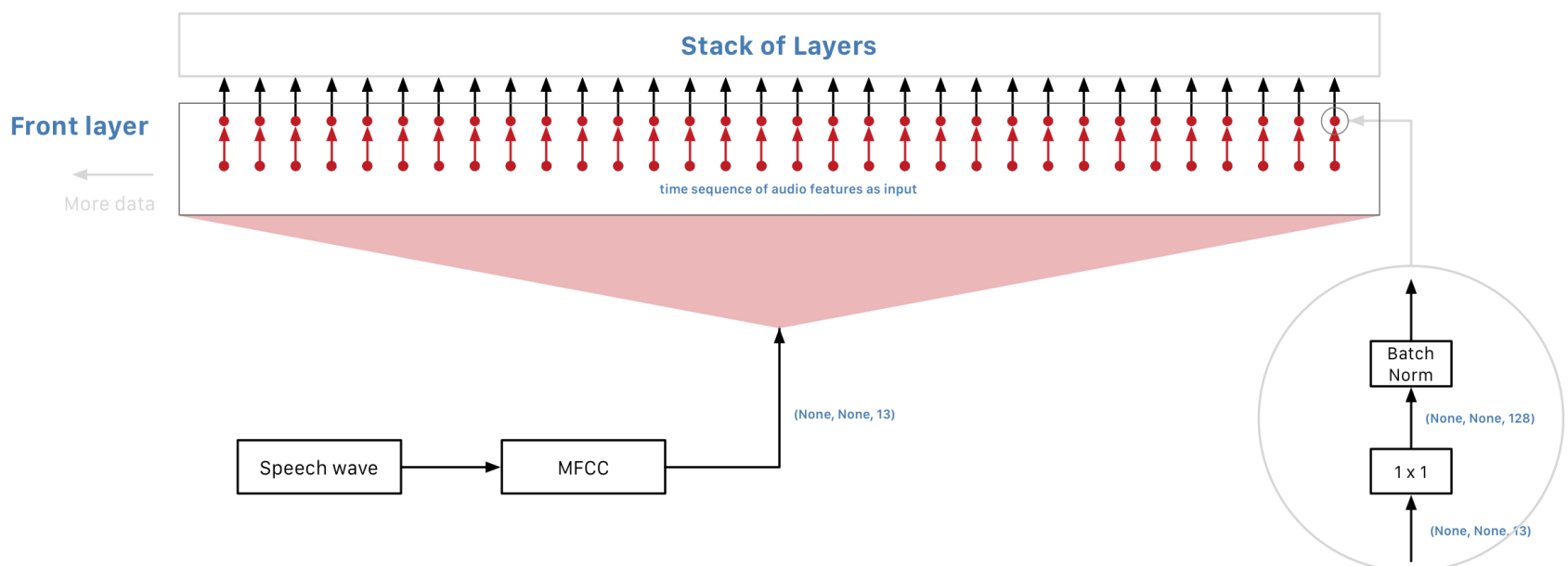
The joint probability of a waveform $\mathbf{x} = \{\mathbf{x}_1, \dots, \mathbf{x}_T\}$ is factorized as a product of conditional probabilities as follows:

$$p(\mathbf{x}) = \prod_{t=1}^T p(x_t \mid x_1, \dots, x_{t-1})$$

Each audio sample x_t is therefore conditioned on the samples at all previous timesteps.

The conditional probability distribution is modeled by a stack of convolutional layers. The model outputs a categorical distribution over the next value x_t with a softmax layer.

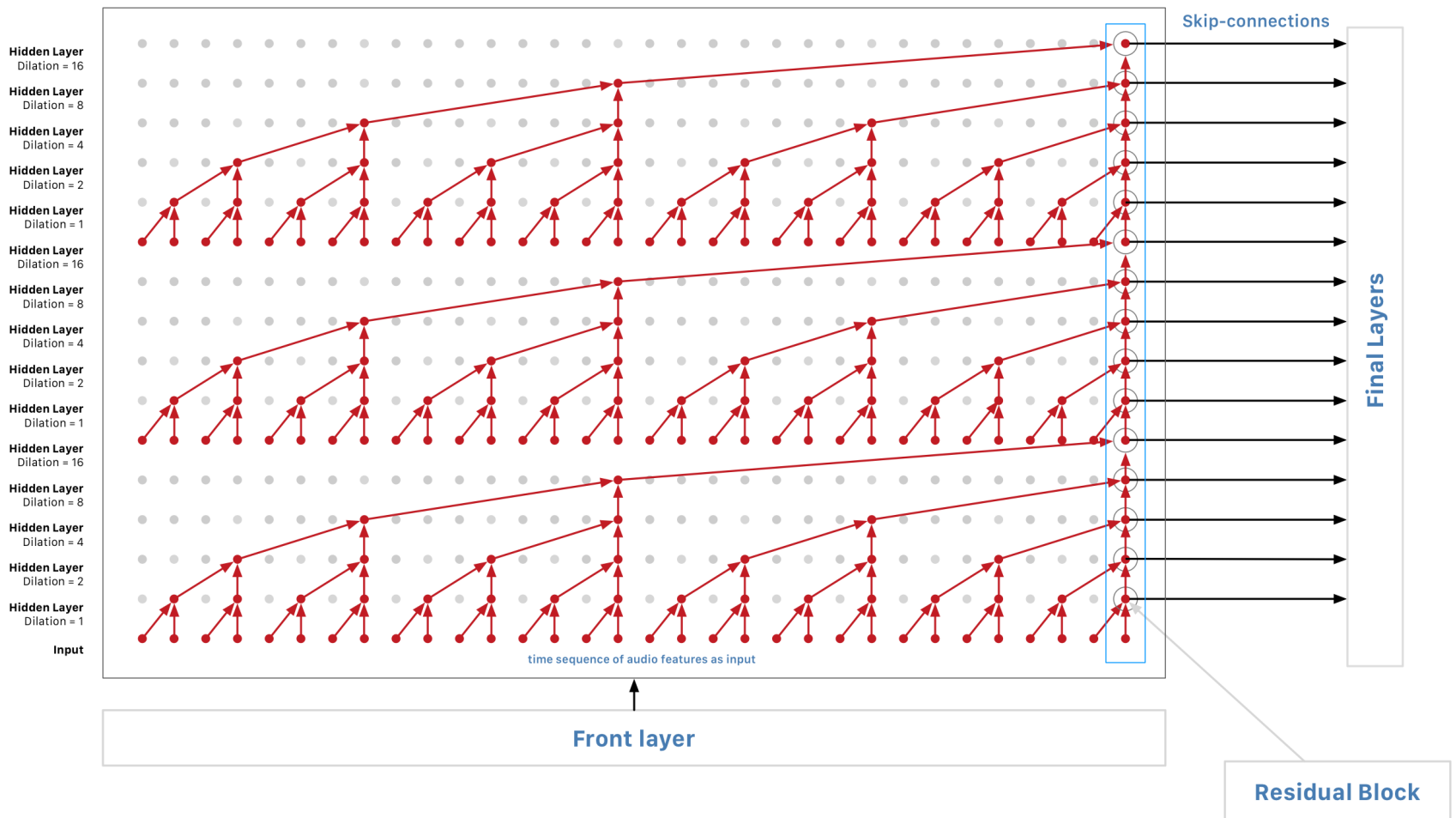
Front Layer



First, the MFCC input is fed into a 1x1 convolutional layer to increase the dimensionality from 13 to 128 to give the network a larger feature space. Then a batch normalization layer is used to solve the problem of vanishing/exploding gradients that are common in deep networks.

Stack of Dilated Causal Convolutional Layers

Stack of dilated causal convolutional layers



Next is a stack of dilated causal convolutional layers, the main component of the WaveNet architecture. By using causal convolutions, the model cannot violate the ordering of the data it's modeling. The prediction $p(x_{t+1} | x_1, \dots, x_t)$ produced by the model at timestep t depends only on the past timesteps.

A dilated convolution is a convolution where the filter is applied over an area larger than its length by skipping input values with a certain step. A dilated convolution effectively allows the network to operate on a coarser scale than with a normal convolution. This is similar to pooling or regular convolutions with strides, but here the output has the same size as the input.

The above visualization highlights the pattern of the dilation across the input. The kernel size of the convolutional layer is 7. The stack itself is represented by the nodes in the blue box.

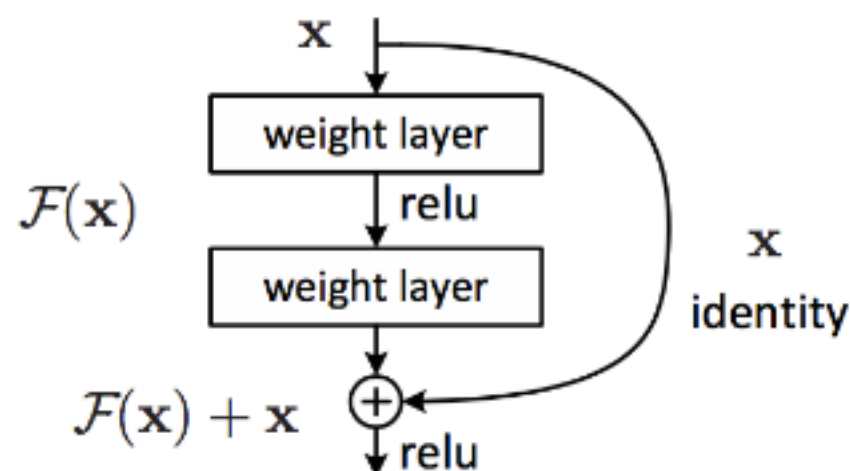
The WaveNet uses stacked dilated convolutions to have a very large receptive field with just a few layers, while preserving the input resolution throughout the network as well as computational efficiency.

Gated Activation Units (Residual and Skip Connections)

Each node in the above diagram is the Gated Activation Unit. These units produce both residual and parameterized skip connections. They are used throughout the network, to speed up convergence and enable training of much deeper models.

The reason for this is an insight from the paper, Deep Residual Learning for Image Recognition (1512.03385 [Deep Residual Learning for Image Recognition \(https://arxiv.org/abs/1512.03385\)](https://arxiv.org/abs/1512.03385)): When networks are made deeper, a degradation problem occurs where accuracy get saturated and then degrades rapidly. This

degradation is *not caused by overfitting* and adding more layers leads to *higher training error*. They found that when more layers are added, some layers were essentially doing identity mapping, $f(x) = x$. And, it was found that identity mapping was difficult for the current group of solvers to find solutions.



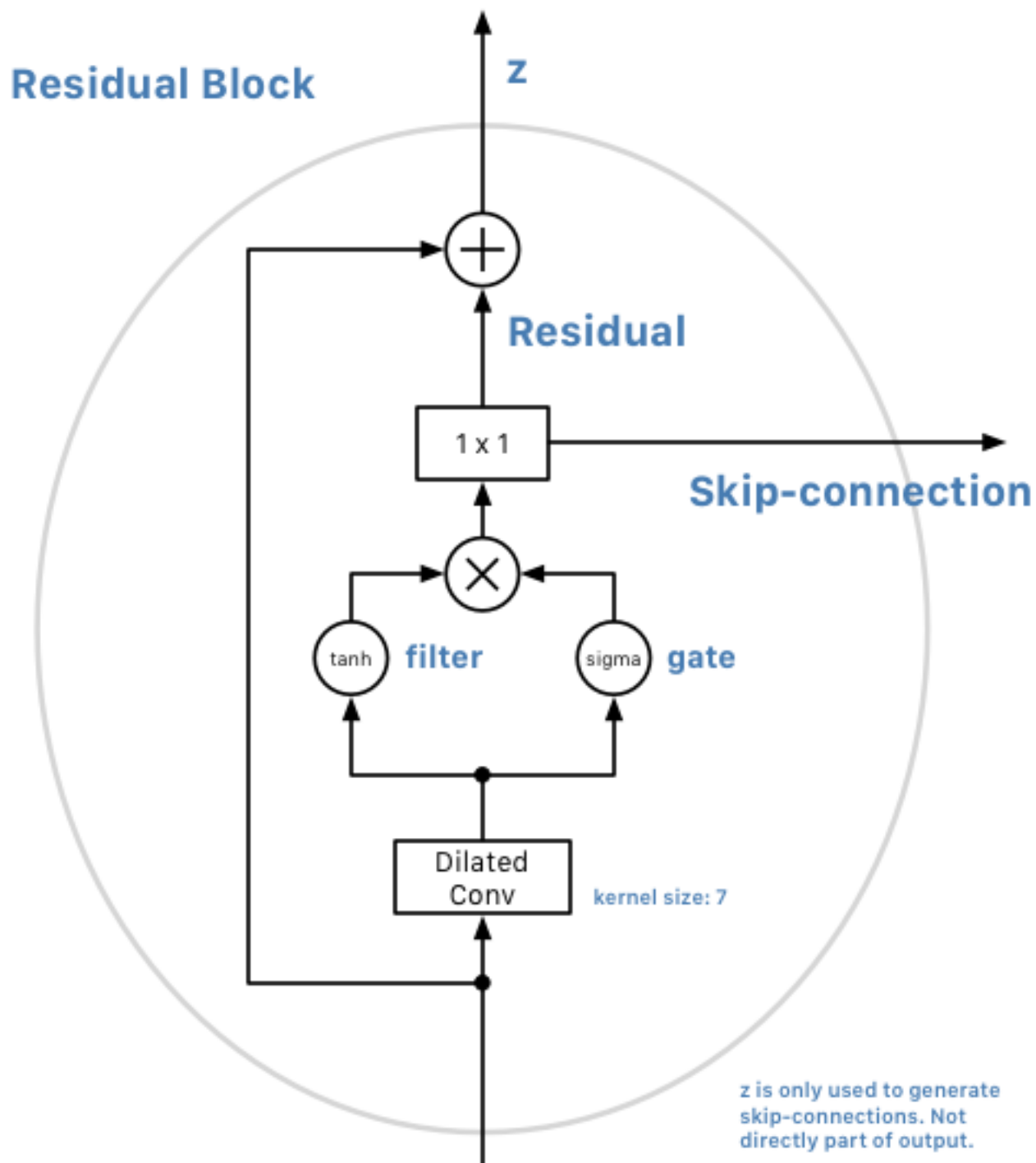
Their solution to the degradation problem was to instead of stacking layers that needed to eventually directly fit a desired underlying mapping, they let the layers fit a residual mapping. Here instead of modeling $H(x)$, the network modeled $F(x) = H(x) - x$. Then by adding x to $F(x)$, they are able to obtain $H(x)$. They found that it was easier “to optimize the residual mapping than to optimize the original, unreferenced mapping. To the extreme, if an identity mapping were optimal, it would be easier to push the residual to zero than to fit an identity mapping by a stack of nonlinear layers.”

$F(x) + x$ is implemented by feedforward neural networks with “shortcut connections.” Shortcut connections are connections that skip one or more layers. They perform identity mapping, and their outputs are added to the outputs of the stacked layers to give $H(x) = F(x) + x$.

The Gated Activation Units implement the function:

$$\mathbf{z} = \tanh(W_{f,k} * \mathbf{x}) \odot \sigma(W_{g,k} * \mathbf{x}),$$

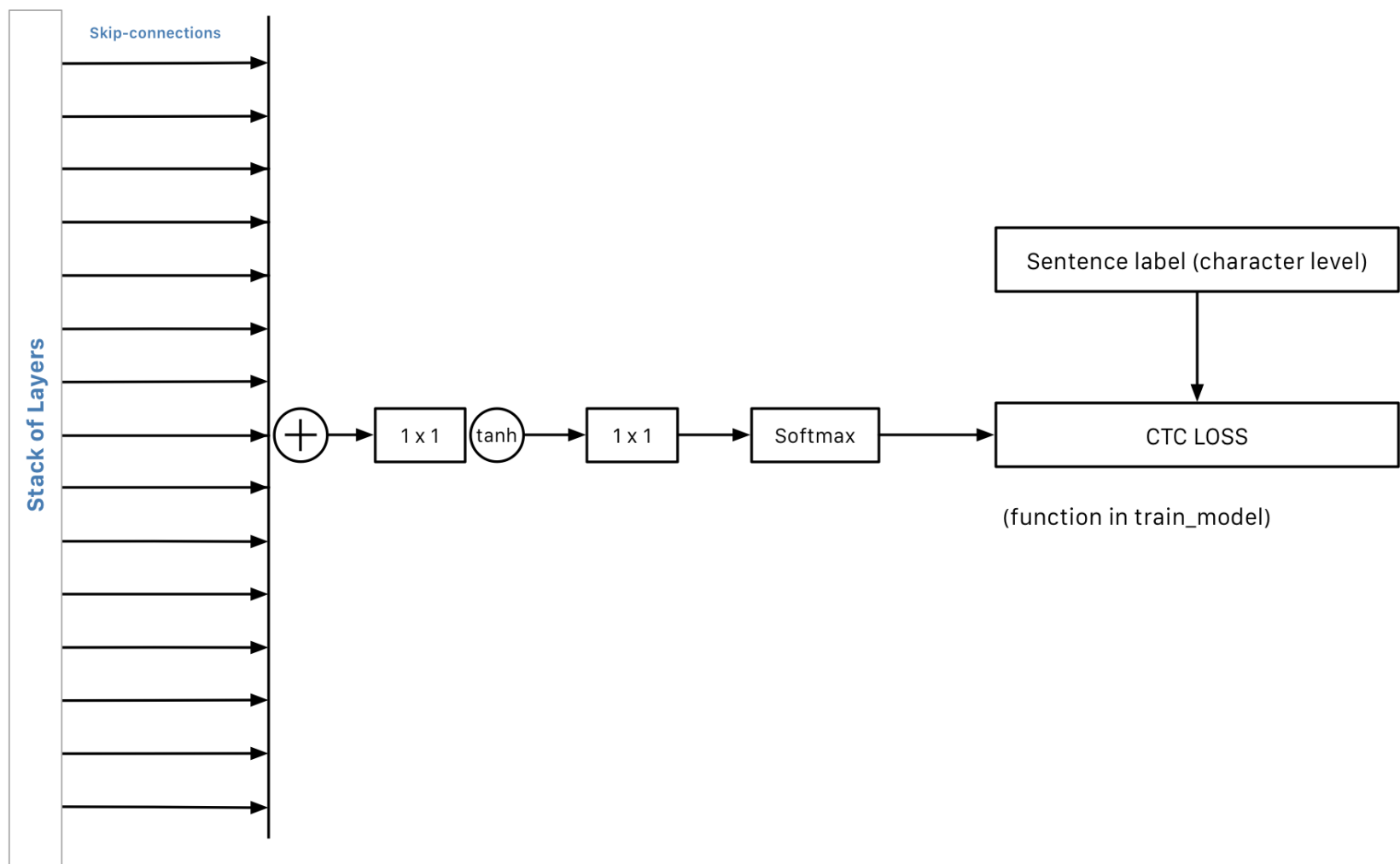
where $*$ denotes a convolution operator, \odot denotes an element-wise multiplication operator, $\sigma(\cdot)$ is a sigmoid function, k is the layer index, f and g denote filter and gate, respectively, and W is a learnable convolution filter. They observed that this non-linearity worked significantly better than the rectified linear activation function for modeling audio signals.



Final Layers

The final layers sums all of the skip connections (which add in $F(x)$ along the way as the data passes through the stack) to get $H(x) = F(x) + x$, our desired function. This is then passed through a 1×1 convolutional layer to reduce the dimensions to 1. We again batch normalize. And, finally, we add a 1×1 layer with softmax to get our predictions.

Final Layers



WaveNet uses a softmax distribution to model the conditional distributions $p(x_t \mid x_1, \dots, x_{t-1})$ over the individual audio samples because of its superior results compared to other methods. “One of the reasons is that a categorical distribution is more flexible and can more easily model arbitrary distributions because it makes no assumptions about their shape.”

Early Results

The model shows a lot of overfitting. (Did more research and found that this was a problem with using WaveNet for speech-to-text.)

In []:

Dropout?

Tried adding a Dropout layer to the final layers. This did not work. It caused an error.

Less Complex Model

Try reducing complextity of network.

In [2]:

```
model_end = wavenet_model(input_dim=13, num_layer_grps=2, filters=64)
```

Layer (type) nected to	Output Shape	Param #	Con
=====			
the_input (InputLayer)	(None, None, 13)	0	
conv_in (Conv1D) _input[0][0]	(None, None, 64)	896	the
bn_conv_in (BatchNormalization) v_in[0][0]	(None, None, 64)	256	con
block_0_1_conv_tanh (Conv1D) conv_in[0][0]	(None, None, 64)	28736	bn_
block_0_1_conv_sigmoid (Conv1D) conv_in[0][0]	(None, None, 64)	28736	bn_
block_0_1_bn_tanh_out (BatchNorm ck_0_1_conv_tanh[0][0]	(None, None, 64)	256	blo
block_0_1_bn_sigmoid_out (BatchN ck_0_1_conv_sigmoid[0][0]	(None, None, 64)	256	blo
block_0_1_multi (Multiply) ck_0_1_bn_tanh_out[0][0] block_0_1_bn_sigmoid_out[0][0]	(None, None, 64)	0	blo
block_0_1_conv_out (Conv1D) ck_0_1_multi[0][0]	(None, None, 64)	4160	blo
block_0_1_bn_skip_out (BatchNorm ck_0_1_conv_out[0][0]	(None, None, 64)	256	blo

block_0_1_add (Add)	(None, None, 64)	0	blo
ck_0_1_bn_skip_out[0][0]			
bn_conv_in[0][0]			
block_0_2_conv_tanh (Conv1D)	(None, None, 64)	28736	blo
ck_0_1_add[0][0]			
block_0_2_conv_sigmoid (Conv1D)	(None, None, 64)	28736	blo
ck_0_1_add[0][0]			
block_0_2_bn_tanh_out (BatchNorm	(None, None, 64)	256	blo
ck_0_2_conv_tanh[0][0]			
block_0_2_bn_sigmoid_out (BatchN	(None, None, 64)	256	blo
ck_0_2_conv_sigmoid[0][0]			
block_0_2_multi (Multiply)	(None, None, 64)	0	blo
ck_0_2_bn_tanh_out[0][0]			
block_0_2_bn_sigmoid_out[0][0]			
block_0_2_conv_out (Conv1D)	(None, None, 64)	4160	blo
ck_0_2_multi[0][0]			
block_0_2_bn_skip_out (BatchNorm	(None, None, 64)	256	blo
ck_0_2_conv_out[0][0]			
block_0_2_add (Add)	(None, None, 64)	0	blo
ck_0_2_bn_skip_out[0][0]			
block_0_1_add[0][0]			
block_0_4_conv_tanh (Conv1D)	(None, None, 64)	28736	blo
ck_0_2_add[0][0]			
block_0_4_conv_sigmoid (Conv1D)	(None, None, 64)	28736	blo
ck_0_2_add[0][0]			
block_0_4_bn_tanh_out (BatchNorm	(None, None, 64)	256	blo
ck_0_4_conv_tanh[0][0]			

block_0_4_bn_sigmoid_out (BatchNorm)	(None, None, 64)	256	block_0_4_conv_sigmoid[0][0]
block_0_4_multi (Multiply)	(None, None, 64)	0	block_0_4_bn_tanh_out[0][0]
block_0_4_bn_sigmoid_out[0][0]			
block_0_4_conv_out (Conv1D)	(None, None, 64)	4160	block_0_4_multi[0][0]
block_0_4_bn_skip_out (BatchNorm)	(None, None, 64)	256	block_0_4_conv_out[0][0]
block_0_4_add (Add)	(None, None, 64)	0	block_0_2_add[0][0]
block_0_8_conv_tanh (Conv1D)	(None, None, 64)	28736	block_0_4_add[0][0]
block_0_8_conv_sigmoid (Conv1D)	(None, None, 64)	28736	block_0_4_add[0][0]
block_0_8_bn_tanh_out (BatchNorm)	(None, None, 64)	256	block_0_8_conv_tanh[0][0]
block_0_8_bn_sigmoid_out (BatchNorm)	(None, None, 64)	256	block_0_8_conv_sigmoid[0][0]
block_0_8_multi (Multiply)	(None, None, 64)	0	block_0_8_bn_tanh_out[0][0]
block_0_8_bn_sigmoid_out[0][0]			
block_0_8_conv_out (Conv1D)	(None, None, 64)	4160	block_0_8_multi[0][0]
block_0_8_bn_skip_out (BatchNorm)	(None, None, 64)	256	

ck_0_8_conv_out[0][0]			
block_0_8_add (Add)			
ck_0_8_bn_skip_out[0][0]	(None, None, 64)	0	blo
block_0_4_add[0][0]			
block_0_16_conv_tanh (Conv1D)			
ck_0_8_add[0][0]	(None, None, 64)	28736	blo
block_0_16_conv_sigmoid (Conv1D)			
ck_0_8_add[0][0]	(None, None, 64)	28736	blo
block_0_16_bn_tanh_out (BatchNor			
ck_0_16_conv_tanh[0][0]	(None, None, 64)	256	blo
block_0_16_bn_sigmoid_out (Batch			
ck_0_16_conv_sigmoid[0][0]	(None, None, 64)	256	blo
block_0_16_multi (Multiply)			
ck_0_16_bn_tanh_out[0][0]	(None, None, 64)	0	blo
block_0_16_bn_sigmoid_out[0][0]			
block_0_16_conv_out (Conv1D)			
ck_0_16_multi[0][0]	(None, None, 64)	4160	blo
block_0_16_bn_skip_out (BatchNor			
ck_0_16_conv_out[0][0]	(None, None, 64)	256	blo
block_0_16_add (Add)			
ck_0_16_bn_skip_out[0][0]	(None, None, 64)	0	blo
block_0_8_add[0][0]			
block_1_1_conv_tanh (Conv1D)			
ck_0_16_add[0][0]	(None, None, 64)	28736	blo
block_1_1_conv_sigmoid (Conv1D)			
ck_0_16_add[0][0]	(None, None, 64)	28736	blo

block_1_1_bn_tanh_out (BatchNorm (None, None, 64))	256	blo
ck_1_1_conv_tanh[0][0]		
block_1_1_bn_sigmoid_out (BatchNorm (None, None, 64))	256	blo
ck_1_1_conv_sigmoid[0][0]		
block_1_1_multi (Multiply)	(None, None, 64)	0
ck_1_1_bn_tanh_out[0][0]		blo
block_1_1_bn_sigmoid_out[0][0]		
block_1_1_conv_out (Conv1D)	(None, None, 64)	4160
ck_1_1_multi[0][0]		blo
block_1_1_bn_skip_out (BatchNorm (None, None, 64))	256	blo
ck_1_1_conv_out[0][0]		
block_1_1_add (Add)	(None, None, 64)	0
ck_1_1_bn_skip_out[0][0]		blo
block_0_16_add[0][0]		
block_1_2_conv_tanh (Conv1D)	(None, None, 64)	28736
ck_1_1_add[0][0]		blo
block_1_2_conv_sigmoid (Conv1D)	(None, None, 64)	28736
ck_1_1_add[0][0]		blo
block_1_2_bn_tanh_out (BatchNorm (None, None, 64))	256	blo
ck_1_2_conv_tanh[0][0]		
block_1_2_bn_sigmoid_out (BatchNorm (None, None, 64))	256	blo
ck_1_2_conv_sigmoid[0][0]		
block_1_2_multi (Multiply)	(None, None, 64)	0
ck_1_2_bn_tanh_out[0][0]		blo
block_1_2_bn_sigmoid_out[0][0]		
block_1_2_conv_out (Conv1D)	(None, None, 64)	4160
ck_1_2_multi[0][0]		blo

block_1_2_bn_skip_out (BatchNorm (None, None, 64))	256	blo
ck_1_2_conv_out[0][0]		
block_1_2_add (Add) (None, None, 64)	0	blo
ck_1_2_bn_skip_out[0][0]		
block_1_1_add[0][0]		
block_1_4_conv_tanh (Conv1D) (None, None, 64)	28736	blo
ck_1_2_add[0][0]		
block_1_4_conv_sigmoid (Conv1D) (None, None, 64)	28736	blo
ck_1_2_add[0][0]		
block_1_4_bn_tanh_out (BatchNorm (None, None, 64))	256	blo
ck_1_4_conv_tanh[0][0]		
block_1_4_bn_sigmoid_out (BatchN (None, None, 64))	256	blo
ck_1_4_conv_sigmoid[0][0]		
block_1_4_multi (Multiply) (None, None, 64)	0	blo
ck_1_4_bn_tanh_out[0][0]		
block_1_4_bn_sigmoid_out[0][0]		
block_1_4_conv_out (Conv1D) (None, None, 64)	4160	blo
ck_1_4_multi[0][0]		
block_1_4_bn_skip_out (BatchNorm (None, None, 64))	256	blo
ck_1_4_conv_out[0][0]		
block_1_4_add (Add) (None, None, 64)	0	blo
ck_1_4_bn_skip_out[0][0]		
block_1_2_add[0][0]		
block_1_8_conv_tanh (Conv1D) (None, None, 64)	28736	blo
ck_1_4_add[0][0]		
block_1_8_conv_sigmoid (Conv1D) (None, None, 64)	28736	blo
ck_1_4_add[0][0]		

block_1_8_bn_tanh_out (BatchNorm (None, None, 64)	256	blo
ck_1_8_conv_tanh[0][0]		
block_1_8_bn_sigmoid_out (BatchN (None, None, 64)	256	blo
ck_1_8_conv_sigmoid[0][0]		
block_1_8_multi (Multiply) (None, None, 64)	0	blo
ck_1_8_bn_tanh_out[0][0]		
block_1_8_bn_sigmoid_out[0][0]		
block_1_8_conv_out (Conv1D) (None, None, 64)	4160	blo
ck_1_8_multi[0][0]		
block_1_8_bn_skip_out (BatchNorm (None, None, 64)	256	blo
ck_1_8_conv_out[0][0]		
block_1_8_add (Add) (None, None, 64)	0	blo
ck_1_8_bn_skip_out[0][0]		
block_1_4_add[0][0]		
block_1_16_conv_tanh (Conv1D) (None, None, 64)	28736	blo
ck_1_8_add[0][0]		
block_1_16_conv_sigmoid (Conv1D) (None, None, 64)	28736	blo
ck_1_8_add[0][0]		
block_1_16_bn_tanh_out (BatchNor (None, None, 64)	256	blo
ck_1_16_conv_tanh[0][0]		
block_1_16_bn_sigmoid_out (Batch (None, None, 64)	256	blo
ck_1_16_conv_sigmoid[0][0]		
block_1_16_multi (Multiply) (None, None, 64)	0	blo
ck_1_16_bn_tanh_out[0][0]		
block_1_16_bn_sigmoid_out[0][0]		
block_1_16_conv_out (Conv1D) (None, None, 64)	4160	blo

ck_1_16_multi[0][0]			
block_1_16_bn_skip_out (BatchNor	(None, None, 64)	256	blo
ck_1_16_conv_out[0][0]			
skip_sum (Add)	(None, None, 64)	0	blo
ck_0_1_bn_skip_out[0][0]			
block_0_2_bn_skip_out[0][0]			
block_0_4_bn_skip_out[0][0]			
block_0_8_bn_skip_out[0][0]			
block_0_16_bn_skip_out[0][0]			
block_1_1_bn_skip_out[0][0]			
block_1_2_bn_skip_out[0][0]			
block_1_4_bn_skip_out[0][0]			
block_1_8_bn_skip_out[0][0]			
block_1_16_bn_skip_out[0][0]			
conv_1 (Conv1D)	(None, None, 64)	4160	ski
p_sum[0][0]			
bn_conv_1 (BatchNormalization)	(None, None, 64)	256	con
v_1[0][0]			
conv_2 (Conv1D)	(None, None, 29)	1885	bn_
conv_1[0][0]			
=====			
Total params: 631,453			
Trainable params: 627,357			
Non-trainable params: 4,096			
None			

```

In [3]:

train_model(input_to_softmax=model_end,
            pickle_path='model_end_2.pickle',
            save_model_path='model_end_2.h5',

```

epochs=20,
spectrogram=False) # change to False if you would like to use MFCC featur

```
Epoch 1/20
101/101 [=====] - 627s - loss: 301.4913 - val
_loss: 294.8876
Epoch 2/20
101/101 [=====] - 604s - loss: 241.8142 - val
_loss: 225.6550
Epoch 3/20
101/101 [=====] - 605s - loss: 226.4872 - val
_loss: 238.6076
Epoch 4/20
101/101 [=====] - 611s - loss: 200.8732 - val
_loss: 190.1827
Epoch 5/20
101/101 [=====] - 601s - loss: 171.7161 - val
_loss: 168.0978
Epoch 6/20
101/101 [=====] - 617s - loss: 152.6980 - val
_loss: 150.5077
Epoch 7/20
101/101 [=====] - 621s - loss: 141.3795 - val
_loss: 142.3021
Epoch 8/20
101/101 [=====] - 613s - loss: 132.7585 - val
_loss: 142.0057
Epoch 9/20
101/101 [=====] - 610s - loss: 125.7593 - val
_loss: 135.3764
Epoch 10/20
101/101 [=====] - 617s - loss: 119.4498 - val
_loss: 134.7948
Epoch 11/20
101/101 [=====] - 622s - loss: 113.9031 - val
_loss: 134.5714
Epoch 12/20
101/101 [=====] - 608s - loss: 108.1584 - val
_loss: 133.1006
Epoch 13/20
101/101 [=====] - 617s - loss: 103.0697 - val
_loss: 133.5844
Epoch 14/20
101/101 [=====] - 619s - loss: 97.7298 - val_
loss: 134.5668
Epoch 15/20
101/101 [=====] - 605s - loss: 93.0563 - val_
loss: 132.7639
Epoch 16/20
101/101 [=====] - 605s - loss: 87.8832 - val_
loss: 137.0102
Epoch 17/20
101/101 [=====] - 605s - loss: 83.7116 - val_
```

loss: 140.0509
Epoch 18/20
101/101 [=====] - 603s - loss: 78.7918 - val_
loss: 144.7398
Epoch 19/20
101/101 [=====] - 604s - loss: 74.3941 - val_
loss: 147.0726
Epoch 20/20
101/101 [=====] - 604s - loss: 70.1260 - val_
loss: 150.6451

In []:

Let's try more data.

In [4]:

model_end_3 = wavenet_model(input_dim=13)

Layer (type) connected to	Output Shape	Param #	Con
=====			
=====			
the_input (InputLayer)	(None, None, 13)	0	
=====			
conv_in (Conv1D) _input[0][0]	(None, None, 128)	1792	the
=====			
bn_conv_in (BatchNormalization) v_in[0][0]	(None, None, 128)	512	con
=====			
block_0_1_conv_tanh (Conv1D) conv_in[0][0]	(None, None, 128)	114816	bn_
=====			
block_0_1_conv_sigmoid (Conv1D) conv_in[0][0]	(None, None, 128)	114816	bn_
=====			
block_0_1_bn_tanh_out (BatchNorm ck_0_1_conv_tanh[0][0]	(None, None, 128)	512	blo
=====			
block_0_1_bn_sigmoid_out (BatchN ck_0_1_conv_sigmoid[0][0]	(None, None, 128)	512	blo

block_0_1_multi (Multiply)	(None, None, 128)	0	blo
ck_0_1_bn_tanh_out[0][0]			
block_0_1_bn_sigmoid_out[0][0]			
block_0_1_conv_out (Conv1D)	(None, None, 128)	16512	blo
ck_0_1_multi[0][0]			
block_0_1_bn_skip_out (BatchNorm	(None, None, 128)	512	blo
ck_0_1_conv_out[0][0]			
block_0_1_add (Add)	(None, None, 128)	0	blo
ck_0_1_bn_skip_out[0][0]			
bn_conv_in[0][0]			
block_0_2_conv_tanh (Conv1D)	(None, None, 128)	114816	blo
ck_0_1_add[0][0]			
block_0_2_conv_sigmoid (Conv1D)	(None, None, 128)	114816	blo
ck_0_1_add[0][0]			
block_0_2_bn_tanh_out (BatchNorm	(None, None, 128)	512	blo
ck_0_2_conv_tanh[0][0]			
block_0_2_bn_sigmoid_out (BatchN	(None, None, 128)	512	blo
ck_0_2_conv_sigmoid[0][0]			
block_0_2_multi (Multiply)	(None, None, 128)	0	blo
ck_0_2_bn_tanh_out[0][0]			
block_0_2_bn_sigmoid_out[0][0]			
block_0_2_conv_out (Conv1D)	(None, None, 128)	16512	blo
ck_0_2_multi[0][0]			
block_0_2_bn_skip_out (BatchNorm	(None, None, 128)	512	blo
ck_0_2_conv_out[0][0]			
block_0_2_add (Add)	(None, None, 128)	0	blo

ck_0_2_bn_skip_out[0][0]			
block_0_1_add[0][0]			
<hr/>			
block_0_4_conv_tanh (Conv1D)	(None, None, 128)	114816	blo
ck_0_2_add[0][0]			
<hr/>			
block_0_4_conv_sigmoid (Conv1D)	(None, None, 128)	114816	blo
ck_0_2_add[0][0]			
<hr/>			
block_0_4_bn_tanh_out (BatchNorm	(None, None, 128)	512	blo
ck_0_4_conv_tanh[0][0]			
<hr/>			
block_0_4_bn_sigmoid_out (BatchN	(None, None, 128)	512	blo
ck_0_4_conv_sigmoid[0][0]			
<hr/>			
block_0_4_multi (Multiply)	(None, None, 128)	0	blo
ck_0_4_bn_tanh_out[0][0]			
block_0_4_bn_sigmoid_out[0][0]			
<hr/>			
block_0_4_conv_out (Conv1D)	(None, None, 128)	16512	blo
ck_0_4_multi[0][0]			
<hr/>			
block_0_4_bn_skip_out (BatchNorm	(None, None, 128)	512	blo
ck_0_4_conv_out[0][0]			
<hr/>			
block_0_4_add (Add)	(None, None, 128)	0	blo
ck_0_4_bn_skip_out[0][0]			
block_0_2_add[0][0]			
<hr/>			
block_0_8_conv_tanh (Conv1D)	(None, None, 128)	114816	blo
ck_0_4_add[0][0]			
<hr/>			
block_0_8_conv_sigmoid (Conv1D)	(None, None, 128)	114816	blo
ck_0_4_add[0][0]			
<hr/>			
block_0_8_bn_tanh_out (BatchNorm	(None, None, 128)	512	blo
ck_0_8_conv_tanh[0][0]			
<hr/>			
<hr/>			

block_0_8_bn_sigmoid_out (BatchNorm)	(None, None, 128)	512	block_0_8_conv_sigmoid[0][0]
block_0_8_multi (Multiply)	(None, None, 128)	0	block_0_8_bn_tanh_out[0][0]
block_0_8_bn_sigmoid_out[0][0]			
block_0_8_conv_out (Conv1D)	(None, None, 128)	16512	block_0_8_multi[0][0]
block_0_8_bn_skip_out (BatchNorm)	(None, None, 128)	512	block_0_8_conv_out[0][0]
block_0_8_add (Add)	(None, None, 128)	0	block_0_4_add[0][0]
block_0_8_bn_skip_out[0][0]			
block_0_16_conv_tanh (Conv1D)	(None, None, 128)	114816	block_0_8_add[0][0]
block_0_16_conv_sigmoid (Conv1D)	(None, None, 128)	114816	block_0_16_conv_tanh[0][0]
block_0_16_bn_tanh_out (BatchNorm)	(None, None, 128)	512	block_0_16_conv_sigmoid[0][0]
block_0_16_bn_sigmoid_out (BatchNorm)	(None, None, 128)	512	block_0_16_bn_tanh_out[0][0]
block_0_16_multi (Multiply)	(None, None, 128)	0	block_0_16_bn_sigmoid_out[0][0]
block_0_16_conv_out (Conv1D)	(None, None, 128)	16512	block_0_16_multi[0][0]
block_0_16_bn_skip_out (BatchNorm)	(None, None, 128)	512	block_0_16_conv_out[0][0]

block_0_16_add (Add)	(None, None, 128)	0	blo
ck_0_16_bn_skip_out[0][0]			
block_0_8_add[0][0]			
block_1_1_conv_tanh (Conv1D)	(None, None, 128)	114816	blo
ck_0_16_add[0][0]			
block_1_1_conv_sigmoid (Conv1D)	(None, None, 128)	114816	blo
ck_0_16_add[0][0]			
block_1_1_bn_tanh_out (BatchNorm	(None, None, 128)	512	blo
ck_1_1_conv_tanh[0][0]			
block_1_1_bn_sigmoid_out (BatchN	(None, None, 128)	512	blo
ck_1_1_conv_sigmoid[0][0]			
block_1_1_multi (Multiply)	(None, None, 128)	0	blo
ck_1_1_bn_tanh_out[0][0]			
block_1_1_bn_sigmoid_out[0][0]			
block_1_1_conv_out (Conv1D)	(None, None, 128)	16512	blo
ck_1_1_multi[0][0]			
block_1_1_bn_skip_out (BatchNorm	(None, None, 128)	512	blo
ck_1_1_conv_out[0][0]			
block_1_1_add (Add)	(None, None, 128)	0	blo
ck_1_1_bn_skip_out[0][0]			
block_0_16_add[0][0]			
block_1_2_conv_tanh (Conv1D)	(None, None, 128)	114816	blo
ck_1_1_add[0][0]			
block_1_2_conv_sigmoid (Conv1D)	(None, None, 128)	114816	blo
ck_1_1_add[0][0]			
block_1_2_bn_tanh_out (BatchNorm	(None, None, 128)	512	blo
ck_1_2_conv_tanh[0][0]			

block_1_2_bn_sigmoid_out (BatchNorm)	(None, None, 128)	512	block_1_2_conv_sigmoid[0][0]
block_1_2_multi (Multiply)	(None, None, 128)	0	block_1_2_bn_tanh_out[0][0]
block_1_2_bn_sigmoid_out[0][0]			
block_1_2_conv_out (Conv1D)	(None, None, 128)	16512	block_1_2_multi[0][0]
block_1_2_bn_skip_out (BatchNorm)	(None, None, 128)	512	block_1_2_conv_out[0][0]
block_1_2_add (Add)	(None, None, 128)	0	block_1_2_bn_skip_out[0][0]
block_1_1_add[0][0]			
block_1_4_conv_tanh (Conv1D)	(None, None, 128)	114816	block_1_2_add[0][0]
block_1_4_conv_sigmoid (Conv1D)	(None, None, 128)	114816	block_1_4_conv_tanh[0][0]
block_1_4_bn_tanh_out (BatchNorm)	(None, None, 128)	512	block_1_4_conv_sigmoid[0][0]
block_1_4_bn_sigmoid_out (BatchNorm)	(None, None, 128)	512	block_1_4_bn_tanh_out[0][0]
block_1_4_multi (Multiply)	(None, None, 128)	0	block_1_4_bn_sigmoid_out[0][0]
block_1_4_bn_sigmoid_out[0][0]			
block_1_4_conv_out (Conv1D)	(None, None, 128)	16512	block_1_4_multi[0][0]
block_1_4_bn_skip_out (BatchNorm)			

ck_1_4_conv_out[0][0]			
block_1_4_add (Add)	(None, None, 128)	0	blo
ck_1_4_bn_skip_out[0][0]			
block_1_2_add[0][0]			
block_1_8_conv_tanh (Conv1D)	(None, None, 128)	114816	blo
ck_1_4_add[0][0]			
block_1_8_conv_sigmoid (Conv1D)	(None, None, 128)	114816	blo
ck_1_4_add[0][0]			
block_1_8_bn_tanh_out (BatchNorm	(None, None, 128)	512	blo
ck_1_8_conv_tanh[0][0]			
block_1_8_bn_sigmoid_out (BatchN	(None, None, 128)	512	blo
ck_1_8_conv_sigmoid[0][0]			
block_1_8_multi (Multiply)	(None, None, 128)	0	blo
ck_1_8_bn_tanh_out[0][0]			
block_1_8_bn_sigmoid_out[0][0]			
block_1_8_conv_out (Conv1D)	(None, None, 128)	16512	blo
ck_1_8_multi[0][0]			
block_1_8_bn_skip_out (BatchNorm	(None, None, 128)	512	blo
ck_1_8_conv_out[0][0]			
block_1_8_add (Add)	(None, None, 128)	0	blo
ck_1_8_bn_skip_out[0][0]			
block_1_4_add[0][0]			
block_1_16_conv_tanh (Conv1D)	(None, None, 128)	114816	blo
ck_1_8_add[0][0]			
block_1_16_conv_sigmoid (Conv1D)	(None, None, 128)	114816	blo
ck_1_8_add[0][0]			

block_1_16_bn_tanh_out (BatchNor ck_1_16_conv_tanh[0][0]	(None, None, 128)	512	blo
block_1_16_bn_sigmoid_out (Batch ck_1_16_conv_sigmoid[0][0]	(None, None, 128)	512	blo
block_1_16_multi (Multiply) ck_1_16_bn_tanh_out[0][0]	(None, None, 128)	0	blo
block_1_16_bn_sigmoid_out[0][0]			
block_1_16_conv_out (Conv1D) ck_1_16_multi[0][0]	(None, None, 128)	16512	blo
block_1_16_bn_skip_out (BatchNor ck_1_16_conv_out[0][0]	(None, None, 128)	512	blo
block_1_16_add (Add) ck_1_16_bn_skip_out[0][0]	(None, None, 128)	0	blo
block_1_8_add[0][0]			
block_2_1_conv_tanh (Conv1D) ck_1_16_add[0][0]	(None, None, 128)	114816	blo
block_2_1_conv_sigmoid (Conv1D) ck_1_16_add[0][0]	(None, None, 128)	114816	blo
block_2_1_bn_tanh_out (BatchNorm ck_2_1_conv_tanh[0][0]	(None, None, 128)	512	blo
block_2_1_bn_sigmoid_out (BatchN ck_2_1_conv_sigmoid[0][0]	(None, None, 128)	512	blo
block_2_1_multi (Multiply) ck_2_1_bn_tanh_out[0][0]	(None, None, 128)	0	blo
block_2_1_bn_sigmoid_out[0][0]			
block_2_1_conv_out (Conv1D) ck_2_1_multi[0][0]	(None, None, 128)	16512	blo

block_2_1_bn_skip_out (BatchNorm (None, None, 128))	512	blo
ck_2_1_conv_out[0][0]		
block_2_1_add (Add)	(None, None, 128)	0
ck_2_1_bn_skip_out[0][0]		
block_1_16_add[0][0]		
block_2_2_conv_tanh (Conv1D)	(None, None, 128)	114816
ck_2_1_add[0][0]		
block_2_2_conv_sigmoid (Conv1D)	(None, None, 128)	114816
ck_2_1_add[0][0]		
block_2_2_bn_tanh_out (BatchNorm (None, None, 128))	512	blo
ck_2_2_conv_tanh[0][0]		
block_2_2_bn_sigmoid_out (BatchN (None, None, 128))	512	blo
ck_2_2_conv_sigmoid[0][0]		
block_2_2_multi (Multiply)	(None, None, 128)	0
ck_2_2_bn_tanh_out[0][0]		
block_2_2_bn_sigmoid_out[0][0]		
block_2_2_conv_out (Conv1D)	(None, None, 128)	16512
ck_2_2_multi[0][0]		
block_2_2_bn_skip_out (BatchNorm (None, None, 128))	512	blo
ck_2_2_conv_out[0][0]		
block_2_2_add (Add)	(None, None, 128)	0
ck_2_2_bn_skip_out[0][0]		
block_2_1_add[0][0]		
block_2_4_conv_tanh (Conv1D)	(None, None, 128)	114816
ck_2_2_add[0][0]		
block_2_4_conv_sigmoid (Conv1D)	(None, None, 128)	114816
ck_2_2_add[0][0]		

block_2_4_bn_tanh_out (BatchNorm (None, None, 128)	512	blo
ck_2_4_conv_tanh[0][0]		
block_2_4_bn_sigmoid_out (BatchN (None, None, 128)	512	blo
ck_2_4_conv_sigmoid[0][0]		
block_2_4_multi (Multiply) (None, None, 128)	0	blo
ck_2_4_bn_tanh_out[0][0]		
block_2_4_bn_sigmoid_out[0][0]		
block_2_4_conv_out (Conv1D) (None, None, 128)	16512	blo
ck_2_4_multi[0][0]		
block_2_4_bn_skip_out (BatchNorm (None, None, 128)	512	blo
ck_2_4_conv_out[0][0]		
block_2_4_add (Add) (None, None, 128)	0	blo
ck_2_4_bn_skip_out[0][0]		
block_2_2_add[0][0]		
block_2_8_conv_tanh (Conv1D) (None, None, 128)	114816	blo
ck_2_4_add[0][0]		
block_2_8_conv_sigmoid (Conv1D) (None, None, 128)	114816	blo
ck_2_4_add[0][0]		
block_2_8_bn_tanh_out (BatchNorm (None, None, 128)	512	blo
ck_2_8_conv_tanh[0][0]		
block_2_8_bn_sigmoid_out (BatchN (None, None, 128)	512	blo
ck_2_8_conv_sigmoid[0][0]		
block_2_8_multi (Multiply) (None, None, 128)	0	blo
ck_2_8_bn_tanh_out[0][0]		
block_2_8_bn_sigmoid_out[0][0]		
block_2_8_conv_out (Conv1D) (None, None, 128)	16512	blo

ck_2_8_multi[0][0]			
block_2_8_bn_skip_out	(BatchNorm	(None, None, 128)	512
ck_2_8_conv_out[0][0]			blo
block_2_8_add	(Add)	(None, None, 128)	0
ck_2_8_bn_skip_out[0][0]			blo
block_2_4_add[0][0]			
block_2_16_conv_tanh	(Conv1D)	(None, None, 128)	114816
ck_2_8_add[0][0]			blo
block_2_16_conv_sigmoid	(Conv1D)	(None, None, 128)	114816
ck_2_8_add[0][0]			blo
block_2_16_bn_tanh_out	(BatchNor	(None, None, 128)	512
ck_2_16_conv_tanh[0][0]			blo
block_2_16_bn_sigmoid_out	(Batch	(None, None, 128)	512
ck_2_16_conv_sigmoid[0][0]			blo
block_2_16_multi	(Multiply)	(None, None, 128)	0
ck_2_16_bn_tanh_out[0][0]			blo
block_2_16_bn_sigmoid_out[0][0]			
block_2_16_conv_out	(Conv1D)	(None, None, 128)	16512
ck_2_16_multi[0][0]			blo
block_2_16_bn_skip_out	(BatchNor	(None, None, 128)	512
ck_2_16_conv_out[0][0]			blo
skip_sum	(Add)	(None, None, 128)	0
ck_0_1_bn_skip_out[0][0]			blo
block_0_2_bn_skip_out[0][0]			
block_0_4_bn_skip_out[0][0]			
block_0_8_bn_skip_out[0][0]			
block_0_16_bn_skip_out[0][0]			

block_1_1_bn_skip_out[0][0]
block_1_2_bn_skip_out[0][0]
block_1_4_bn_skip_out[0][0]
block_1_8_bn_skip_out[0][0]
block_1_16_bn_skip_out[0][0]
block_2_1_bn_skip_out[0][0]
block_2_2_bn_skip_out[0][0]
block_2_4_bn_skip_out[0][0]
block_2_8_bn_skip_out[0][0]
block_2_16_bn_skip_out[0][0]

conv_1 (Conv1D)	(None, None, 128)	16512	skip
p_sum[0][0]			
bn_conv_1 (BatchNormalization)	(None, None, 128)	512	conv_1[0][0]
conv_2 (Conv1D)	(None, None, 29)	3741	bn_conv_1[0][0]
=====			
=====			
Total params: 3,738,269			
Trainable params: 3,726,237			
Non-trainable params: 12,032			

None

In []:

Use: train_json='train-clean-100_corpus.json'

In [3]:

```
train_model(input_to_softmax=model_end_3,
            pickle_path='model_end_3.pickle',
            save_model_path='model_end_3.h5',
```

```
train_json='train-clean-100_corpus.json',  
epochs=20,  
spectrogram=False) # change to False if you would like to use MFCC featur
```

```
Epoch 1/20  
259/260 [=====>.] - ETA: 14s - loss: 302.9868Epoch 00000: val_loss improved from inf to 242.92232, saving model to results/model_end_3.h5  
260/260 [=====] - 4209s - loss: 302.8925 - val_loss: 242.9223  
Epoch 2/20  
259/260 [=====>.] - ETA: 14s - loss: 200.3781Epoch 00001: val_loss improved from 242.92232 to 143.32914, saving model to results/model_end_3.h5  
260/260 [=====] - 4168s - loss: 200.2383 - val_loss: 143.3291  
Epoch 3/20  
259/260 [=====>.] - ETA: 14s - loss: 155.4101Epoch 00002: val_loss improved from 143.32914 to 125.44872, saving model to results/model_end_3.h5  
260/260 [=====] - 4172s - loss: 155.4190 - val_loss: 125.4487  
Epoch 4/20  
259/260 [=====>.] - ETA: 14s - loss: 138.3119Epoch 00003: val_loss improved from 125.44872 to 113.41151, saving model to results/model_end_3.h5  
260/260 [=====] - 4142s - loss: 138.3171 - val_loss: 113.4115  
Epoch 5/20  
259/260 [=====>.] - ETA: 14s - loss: 127.6338Epoch 00004: val_loss improved from 113.41151 to 110.36013, saving model to results/model_end_3.h5  
260/260 [=====] - 4185s - loss: 127.6479 - val_loss: 110.3601  
Epoch 6/20  
259/260 [=====>.] - ETA: 14s - loss: 119.8854Epoch 00005: val_loss improved from 110.36013 to 108.91213, saving model to results/model_end_3.h5  
260/260 [=====] - 4182s - loss: 119.9216 - val_loss: 108.9121  
Epoch 7/20  
259/260 [=====>.] - ETA: 14s - loss: 113.1806Epoch 00006: val_loss improved from 108.91213 to 105.35011, saving model to results/model_end_3.h5  
260/260 [=====] - 4168s - loss: 113.1065 - val_loss: 105.3501  
Epoch 8/20  
259/260 [=====>.] - ETA: 14s - loss: 106.9797Epoch 00007: val_loss improved from 105.35011 to 102.61770, saving model to results/model_end_3.h5  
260/260 [=====] - 4173s - loss: 106.9537 - val_loss: 102.6177  
Epoch 9/20
```

```
259/260 [=====>.] - ETA: 14s - loss: 99.5487Epoch 00008: val_loss improved from 102.61770 to 102.04600, saving model to results/model_end_3.h5
260/260 [=====] - 4143s - loss: 99.5354 - val_loss: 102.0460
Epoch 10/20
259/260 [=====>.] - ETA: 14s - loss: 92.9767Epoch 00009: val_loss did not improve
260/260 [=====] - 4121s - loss: 92.9467 - val_loss: 103.4206
Epoch 11/20
259/260 [=====>.] - ETA: 14s - loss: 86.4353Epoch 00010: val_loss did not improve
260/260 [=====] - 4108s - loss: 86.4439 - val_loss: 102.3125
Epoch 12/20
259/260 [=====>.] - ETA: 14s - loss: 79.9540Epoch 00011: val_loss did not improve
260/260 [=====] - 4109s - loss: 79.9836 - val_loss: 103.8346
Epoch 00011: early stopping
```

In []:

Modifications

To solve the overfitting problem:

- Tried adding a Dropout layer to the final layers. This did not work. It caused an error.
- Tried a less complex model. Did not show much improvement.
- Trained on a larger dataset. Used the LibriSpeech-100 dataset.
- Added early stopping by adding `save_best_only=True` to `ModelCheckpoint` and added `EarlyStopping`.

More data and early stopping got the validation error to 103.8346.

In []:

In []:

In []:

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 [1]:

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

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

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

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

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

Using TensorFlow backend.

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

In [11]:

```
get_predictions(index=0,
                 partition='train',
                 input_to_softmax=model_end_3,
                 model_path='results/model_end_3.h5')
```

True transcription:

mister quilter is the apostle of the middle classes and we are glad to
welcome his gospel

Predicted transcription:

theste coer se appis of te tht le cises and wir glad poon hisgispe

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

In [7]:

```
get_predictions(index=0,
                 partition='validation',
                 input_to_softmax=model_end_3,
                 model_path='results/model_end.h5')
```

True transcription:

stuff it into you his belly counselled him

Predicted transcription:

stoffi int yu his vely caungeltham

In []:

In []:

In []:

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!

In []:

In []: