Introduction

In this notebook we will reproduce the results of Deep Speech: Scaling up end-to-end speech recognition (http://arxiv.org/abs/1412.5567). The core of the system is a bidirectional recurrent neural network (BRNN) trained to ingest speech spectrograms and generate English text transcriptions.

Let a single utterance x and label y be sampled from a training set $S = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots\}$. Each utterance, $x^{(i)}$ is a time-series of length $T^{(i)}$ where every time-slice is a vector of audio features, $x_t^{(i)}$ where $t = 1, \dots, T^{(i)}$. We use spectrograms as our features; so $x_{t,p}^{(i)}$ denotes the power of the p-th frequency bin in the audio frame at time t. The goal of our BRNN is to convert an input sequence x into a sequence of character probabilities for the transcription y, with $\hat{y}_t = \mathbb{P}(c_t \mid x)$, where $c_t \in \{a, b, c, \dots, z, space, apostrophe, blank\}$. (The significance of $blank$ will be explained below.)

Our BRNN model is composed of 5 layers of hidden units. For an input x, the hidden units at layer l are denoted $h^{(l)}$ with the convention that $h^{(0)}$ is the input. The first three layers are not recurrent. For the first layer, at each time t, the output depends on the spectrogram frame x_t along with a context of C frames on each side. (We typically use $C \in \{5, 7, 9\}$ for our experiments.) The remaining non-recurrent layers operate on independent data for each time step. Thus, for each time t, the first 3 layers are computed by:

$$h_t^{(l)} = g(W^{(l)} h_t^{(l-1)} + b^{(l)})$$

where $g(z) = \min\{\max\{0, z\}, 20\}$ is the clipped rectified-linear (ReLu) activation function and $W^{(l)}$, $b^{(l)}$ are the weight matrix and bias parameters for layer l. The fourth layer is a bidirectional recurrent layer[1 (http://www.di.ufpe.br/~fnj/RNA/bibliografia/BRNN.pdf)]. This layer includes two sets of hidden units: a set with forward recurrence, $h^{(f)}$, and a set with backward recurrence $h^{(b)}$:

$$\begin{aligned} h_t^{(f)} &= g(W^{(4)} h_t^{(3)} + W_r^{(f)} h_{t-1}^{(f)} + b^{(4)}) \\ h_t^{(b)} &= g(W^{(4)} h_t^{(3)} + W_r^{(b)} h_{t+1}^{(b)} + b^{(4)}) \end{aligned}$$

Note that $h^{(f)}$ must be computed sequentially from $t = 1$ to $t = T^{(i)}$ for the i-th utterance, while the units $h^{(b)}$ must be computed sequentially in reverse from $t = T^{(i)}$ to $t = 1$.

The fifth (non-recurrent) layer takes both the forward and backward units as inputs

$$h^{(5)} = g(W^{(5)} h^{(4)} + b^{(5)})$$

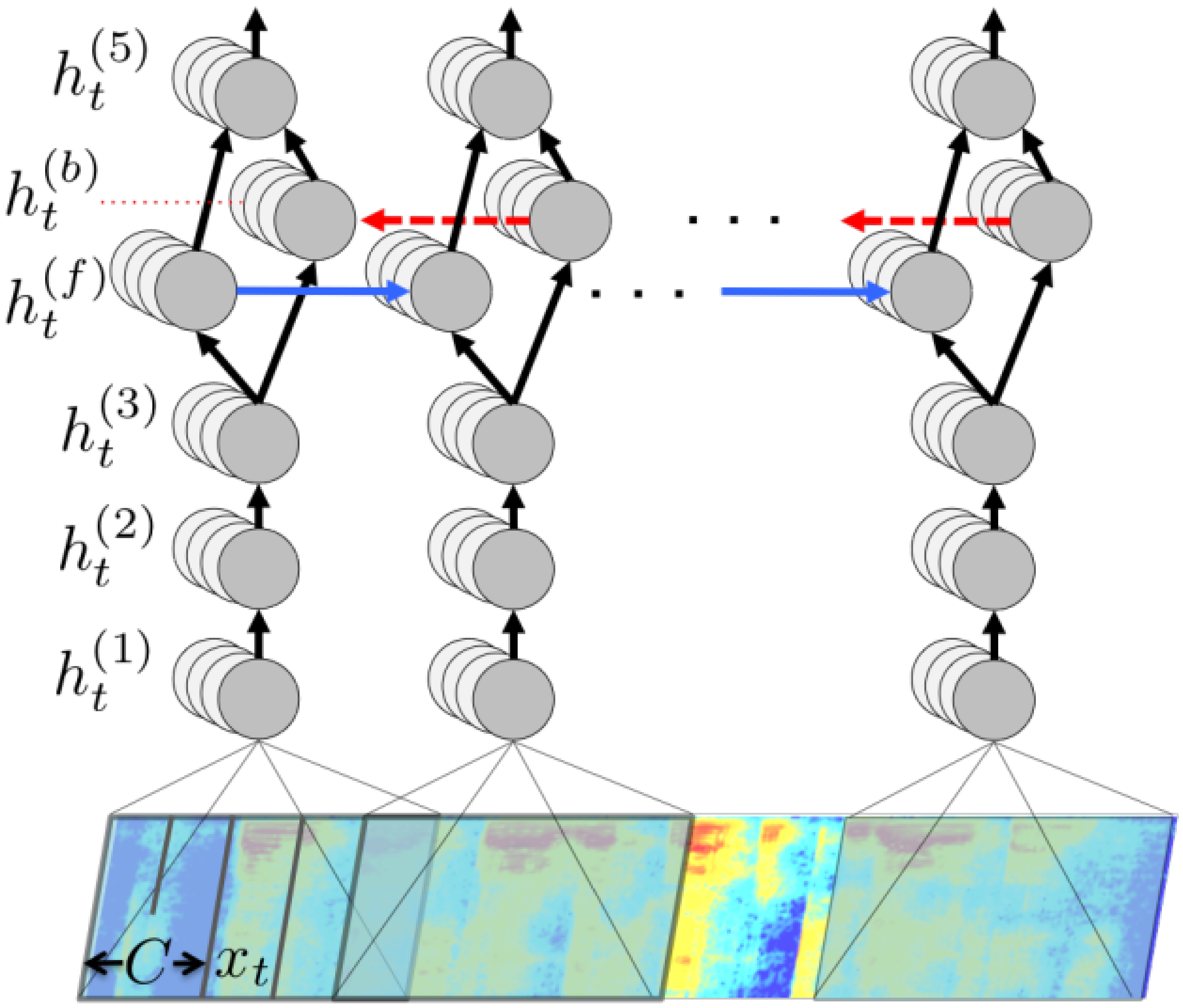
where $h^{(4)} = h^{(f)} + h^{(b)}$. The output layer is a standard softmax function that yields the predicted character probabilities for each time slice t and character k in the alphabet:

$$h_{t,k}^{(6)} = \hat{y}_{t,k} \equiv \mathbb{P}(c_t = k \mid x) = \frac{\exp\left((W^{(6)} h_t^{(5)})_k + b_k^{(6)} \right)}{\sum_j \exp\left((W^{(6)} h_t^{(5)})_j + b_j^{(6)} \right)}$$

Here $b_k^{(6)}$ denotes the k-th bias and $(W^{(6)} h_t^{(5)})_k$ the k-th element of the matrix product.

Once we have computed a prediction for $\mathbb{P}(c_t = k \mid x)$, we compute the CTC loss[2 (http://www.cs.toronto.edu/~graves/preprint.pdf)] $\mathcal{L}(\hat{y}, y)$ to measure the error in prediction. During training, we can evaluate the gradient $\nabla \mathcal{L}(\hat{y}, y)$ with respect to the network outputs given the ground-truth character sequence y. From this point, computing the gradient with respect to all of the model parameters may be done via back-propagation through the rest of the network. We use the Adam method for training[3 (http://arxiv.org/abs/1412.6980)].

The complete BRNN model is illustrated in the figure below.



Data Import

The import routines for the TED-LIUM (http://www-lium.univ-lemans.fr/en/content/ted-lium-corpus) have yet to be written.

In [1]:

```
#from ted_lium import input_data
#ted_lium = input_data.read_data_sets("./TEDLIUM_release2")
```

Preliminaries

Imports

Here we first import all of the packages we require to implement the DeepSpeech BRNN.

In [2]:

```
import tensorflow as tf
from tensorflow.python.framework.constant_op import constant
import numpy as np
```

Global Constants

Next we introduce several constants used in the algorithm below. In particular, we define

- `learning_rate` - The learning rate we will employ in Adam optimizer[3] (http://arxiv.org/abs/1412.6980)
- `training_iters`- The number of iterations we will train for
- `batch_size`- The number of elements in a batch
- `display_step`- The number of iterations we cycle through before displaying progress

In [3]:

```
learning_rate = 0.001   # TODO: Determine a reasonable value for this
training_iters = 100000 # TODO: Determine a reasonable value for this
batch_size = 128        # TODO: Determine a reasonable value for this
display_step = 10       # TODO: Determine a reasonable value for this
```

Note that we use the Adam optimizer[3] (http://arxiv.org/abs/1412.6980) instead of Nesterov's Accelerated Gradient [4] (http://www.cs.utoronto.ca/~ilya/pubs/2013/1051_2.pdf) used in the original DeepSpeech paper, as, at the time of writing, TensorFlow does not have an implementation of Nesterov's Accelerated Gradient [4] (http://www.cs.utoronto.ca/~ilya/pubs/2013/1051_2.pdf).

As we will also employ dropout on the feedforward layers of the network, we need to define a parameter `dropout_rate` that keeps track of the dropout rate for these layers

In [4]:

```
dropout_rate = 0.05
```

One more constant required of the non-recurrant layers is the clipping value of the ReLU. We capture that in the value of the variable `relu_clip`

In [5]:

```
relu_clip = 20 # TODO: Validate this is a reasonable value
```

Geometric Constants

Now we will introduce several constants related to the geometry of the network.

The network views each speech sample as a sequence of time-slices $x_t^{(i)}$ of length $T^{(i)}$. As the speech samples vary in length, we know that $T^{(i)}$ need not equal $T^{(j)}$ for $i \neq j$. However, BRNN in TensorFlow are unable to deal with sequences with differing lengths. Thus, we must pad speech sample sequences with trailing zeros such that they are all of the same length. This common padded length is captured in the variable `n_steps`

In [6]:

```
n_steps = 500 # TODO: Determine this programatically from the longest speech sample
```

Each of the `n_steps` vectors is the Fourier transform of a time-slice of the speech sample. The number of "bins" of this Fourier transform is dependent upon the sample rate of the data set. Generically, if the sample rate is 8kHz we use 80bins. If the sample rate is 16kHz we use 160bins... We capture the dimension of these vectors, equivalently the number of bins in the Fourier transform, in the variable `n_input`

In [7]:

```
n_input = 160 # TODO: Determine this programatically from the sample rate
```

As previously mentioned, the BRNN is not simply fed the Fourier transform of a given time-slice. It is fed, in addition, a context of $C \in \{5, 7, 9\}$ frames on either side of the frame in question. The number of frames in this context is captured in the variable `n_context`

In [8]:

```
n_context = 5 # TODO: Determine the optimal value using a validation data set
```

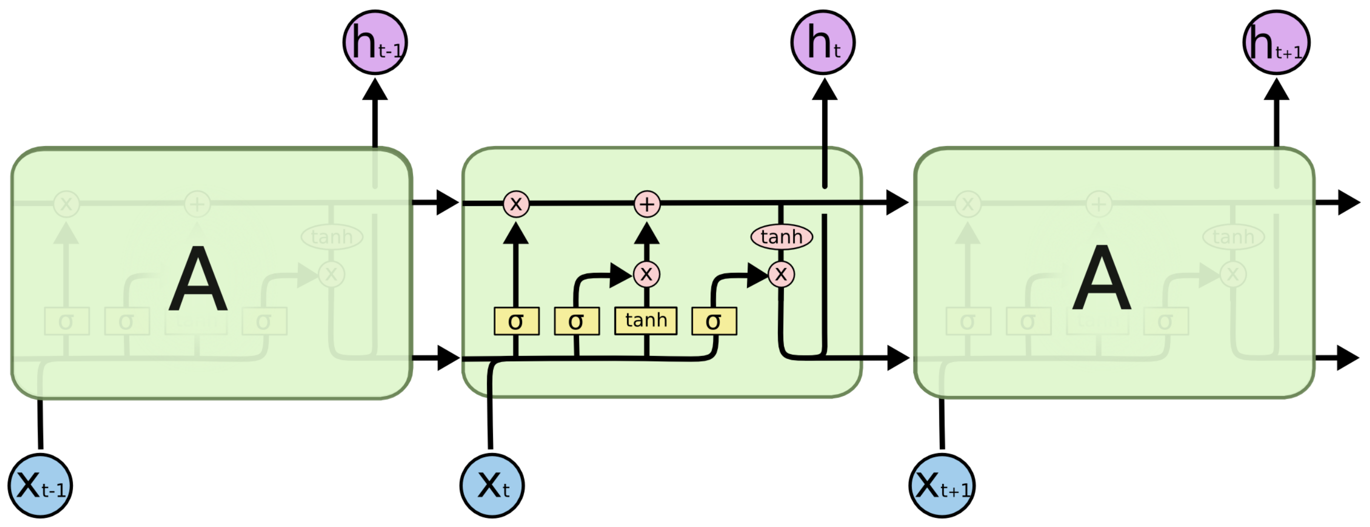
Next we will introduce constants that specify the geometry of some of the non-recurrent layers of the network. We do this by simply specifying the number of units in each of the layers

In [9]:

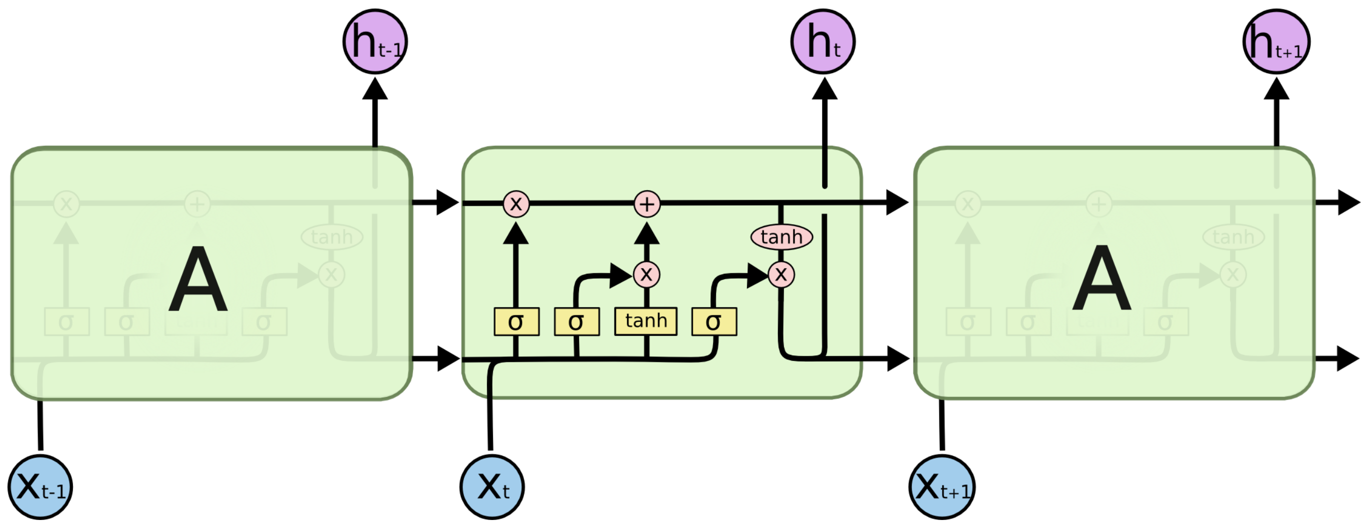
```
n_hidden_1 = n_input + 2*n_input*n_context # Note: This value was not specified in the
 original paper
n_hidden_2 = n_input + 2*n_input*n_context # Note: This value was not specified in the
 original paper
n_hidden_5 = n_input + 2*n_input*n_context # Note: This value was not specified in the
 original paper
```

where `n_hidden_1` is the number of units in the first layer, `n_hidden_2` the number of units in the second, and `n_hidden_5` the number in the fifth. We haven't forgotten about the third or sixth layer. We will define their unit count below.

A LSTM BRNN consists of a pair of LSTM RNN's. One LSTM RNN that works "forward in time"



and a second LSTM RNN that works "backwards in time"



The dimension of the cell state, the upper line connecting subsequent LSTM units, is independent of the input dimension and the same for both the forward and backward LSTM RNN.

Hence, we are free to choose the dimension of this cell state independent of the input dimension. We capture the cell state dimension in the variable `n_cell_dim`.

In [10]:

```
n_cell_dim = n_input + 2*n_input*n_context # TODO: Is this a reasonable value
```

The number of units in the third layer, which feeds in to the LSTM, is determined by `n_cell_dim` as follows

In [11]:

```
n_hidden_3 = 2 * n_cell_dim
```

Next, we introduce an additional variable `n_character` which holds the number of characters in the target language plus one, for the $blamk$. For English it is the cardinality of the set $\{a, b, c, \ldots, z, space, apostrophe, blank\}$ we referred to earlier.

In [12]:

```
n_character = 29 # TODO: Determine if this should be extended with other punctuation
```

The number of units in the sixth layer is determined by `n_character` as follows

In [13]:

```
n_hidden_6 = n_character
```

Graph Creation

Next we concern ourselves with graph creation.

First we create several place holders in our graph. The first two x and y are placeholders for our training data pairs.

In [14]:

```
x = tf.placeholder("float", [None, n_steps, n_input + 2*n_input*n_context])
y = tf.placeholder("string", [None, 1])
```

As y represents the text transcript of each element in a batch, it is of type "string" and has shape `[None, 1]` where the `None` dimension corresponds to the number of elements in the batch.

The placeholder x is a place holder for the the speech spectrograms along with their prefix and postfix contexts for each element in a batch. As it represents a spectrogram, its type is "float". The `None` dimension of its shape

```
[None, n_steps, n_input + 2*n_input*n_context]
```

has the same meaning as the `None` dimension in the shape of `y`. The `n_steps` dimension of its shape indicates the number of time-slices in the sequence. Finally, the `n_input + 2*n_input*n_context` dimension of its shape indicates the number of bins in Fourier transform `n_input` along with the number of bins in the prefix-context `n_input*n_context` and postfix-contex `n_input*n_context`.

The next placeholders we introduce `istate_fw` and `istate_bw` correspond to the initial states and cells of the forward and backward LSTM networks. As both of these are floats of dimension `n_cell_dim`, we define `istate_fw` and `istate_bw` as follows

In [15]:

```
istate_fw = (tf.placeholder("float", [None, n_cell_dim]), tf.placeholder("float", [None
, n_cell_dim]))
istate_bw = (tf.placeholder("float", [None, n_cell_dim]), tf.placeholder("float", [None
, n_cell_dim]))
```

As we will be employing dropout on the feedforward layers of the network we will also introduce a placeholder `keep_prob` which is a placeholder for the dropout rate for the feedforward layers

In [16]:

```
keep_prob = tf.placeholder(tf.float32)
```

We will define the learned variables through two dictionaries. The first dictionary `weights` holds the learned weight variables. The second `biases` holds the learned bias variables.

The `weights` dictionary has the keys `'h1'`, `'h2'`, `'h3'`, `'h5'`, and `'h6'` each keyed against the values of the corresponding weight matrix. In particular, the first key `'h1'` is keyed against a value which is the learned weight matrix that converts an input vector of dimension `n_input + 2*n_input*n_context` to a vector of dimension `n_hidden_1`. Similarly, the second key `'h2'` is keyed against a value which is the weight matrix converting an input vector of dimension `n_hidden_1` to one of dimension `n_hidden_2`. The keys `'h3'`, `'h5'`, and `'h6'` are similar. Likewise, the `biases` dictionary has biases for the various layers.

Concretely these dictionaries are given by

In [17]:

```
# Store layers weight & bias
# TODO: Is random_normal the best distribution to draw from?
weights = {
    'h1': tf.Variable(tf.random_normal([n_input + 2*n_input*n_context, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'h3': tf.Variable(tf.random_normal([n_hidden_2, n_hidden_3])),
    'h5': tf.Variable(tf.random_normal([(2 * n_cell_dim), n_hidden_5])),
    'h6': tf.Variable(tf.random_normal([n_hidden_5, n_hidden_6]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'b3': tf.Variable(tf.random_normal([n_hidden_3])),
    'b5': tf.Variable(tf.random_normal([n_hidden_5])),
    'b6': tf.Variable(tf.random_normal([n_hidden_6]))
}
```

Next we introduce a utility function `BiRNN` that can take our placeholders `x`, `istate_fw`, and `istate_bw` along with the dictionaries `weights` and `biases` and add all the apropos operators to our default graph.

In [18]:

```
def BiRNN(_X, _istate_fw, _istate_bw, _weights, _biases):
    # Input shape: [batch_size, n_steps, n_input + 2*n_input*n_context]
    _X = tf.transpose(_X, [1, 0, 2])  # Permute n_steps and batch_size
    # Reshape to prepare input for first layer
    _X = tf.reshape(_X, [-1, n_input + 2*n_input*n_context]) # (n_steps*batch_size, n_i
nput + 2*n_input*n_context)

    #Hidden layer with clipped RELU activation and dropout
    layer_1 = tf.minimum(tf.nn.relu(tf.add(tf.matmul(_X, _weights['h1']), _biases['b1'
])), relu_clip)
    layer_1 = tf.nn.dropout(layer_1, keep_prob)
    #Hidden layer with clipped RELU activation and dropout
    layer_2 = tf.minimum(tf.nn.relu(tf.add(tf.matmul(layer_1, _weights['h2']), _biases[
'b2'])), relu_clip)
    layer_2 = tf.nn.dropout(layer_2, keep_prob)
    #Hidden layer with clipped RELU activation and dropout
    layer_3 = tf.minimum(tf.nn.relu(tf.add(tf.matmul(layer_2, _weights['h3']), _biases[
'b3'])), relu_clip)
    layer_3 = tf.nn.dropout(layer_3, keep_prob)

    # Define lstm cells with tensorflow
    # Forward direction cell
    lstm_fw_cell = tf.nn.rnn_cell.BasicLSTMCell(n_cell_dim, forget_bias=1.0)
    # Backward direction cell
    lstm_bw_cell = tf.nn.rnn_cell.BasicLSTMCell(n_cell_dim, forget_bias=1.0)

    # Split data because rnn cell needs a list of inputs for the BRNN inner loop
    layer_3 = tf.split(0, n_steps, layer_3)

    # Get lstm cell output
    outputs, output_state_fw, output_state_bw = tf.nn.bidirectional_rnn(cell_fw=lstm_fw
_cell,
                                                                         cell_bw=lstm_bw
_cell,
                                                                         inputs=layer_3,
                                                                         initial_state_f
w=_istate_fw,
                                                                         initial_state_b
w=_istate_bw)

    # Reshape outputs from a list of n_steps tensors each of shape [batch_size, 2*n_cel
l_dim]
    # to a single tensor of shape [n_steps*batch_size, 2*n_cell_dim]
    outputs = tf.pack(outputs[0])
    outputs = tf.reshape(outputs, [-1, 2*n_cell_dim])

    #Hidden layer with clipped RELU activation and dropout
    layer_5 = tf.minimum(tf.nn.relu(tf.add(tf.matmul(outputs, _weights['h5']), _biases[
'b5'])), relu_clip)
    layer_5 = tf.nn.dropout(layer_5, keep_prob)
    #Hidden layer with softmax function
    layer_6 = tf.nn.softmax(tf.add(tf.matmul(layer_5, _weights['h6']), _biases['b6']))

    # Reshape layer_6 from a tensor of shape [n_steps*batch_size, n_hidden_6]
    # to a tensor of shape [batch_size, n_steps, n_hidden_6]
    layer_6 = tf.reshape(layer_6, [n_steps, batch_size, n_hidden_6])
    layer_6 = tf.transpose(layer_6, [1, 0, 2])  # Permute n_steps and batch_size
    # Return layer_6
    return layer_6
```

The first few lines of the function `BiRNN`

```
def BiRNN(_X, _istate_fw, _istate_bw, _weights, _biases):
    # Input shape: [batch_size, n_steps, n_input + 2*n_input*n_context]
    _X = tf.transpose(_X, [1, 0, 2])  # Permute n_steps and batch_size
    # Reshape to prepare input for first layer
    _X = tf.reshape(_X, [-1, n_input + 2*n_input*n_context])
    ...
```

reshape `_X` which has shape `[batch_size, n_steps, n_input + 2*n_input*n_context]` initially, to a tensor with shape `[n_steps*batch_size, n_input + 2*n_input*n_context]`. This is done to prepare the batch for input into the first layer which expects a tensor of rank `2`.

The next few lines of `BiRNN`

```
#Hidden layer with clipped RELU activation and dropout
    layer_1 = tf.minimum(tf.nn.relu(tf.add(tf.matmul(_X, _weights['h1']), _biase
s['b1'])), relu_clip)
    layer_1 = tf.nn.dropout(layer_1, keep_prob)
    ...
```

pass `_X` through the first layer of the non-recurrent neural network, then apply dropout to the result.

The next few lines do the same thing, but for the second and third layers

```
#Hidden layer with clipped RELU activation and dropout
    layer_2 = tf.minimum(tf.nn.relu(tf.add(tf.matmul(layer_1, _weights['h2']), _
biases['b2'])), relu_clip)
    layer_2 = tf.nn.dropout(layer_2, keep_prob)
    #Hidden layer with clipped RELU activation and dropout
    layer_3 = tf.minimum(tf.nn.relu(tf.add(tf.matmul(layer_2, _weights['h3']), _
biases['b3'])), relu_clip)
    layer_3 = tf.nn.dropout(layer_3, keep_prob)
```

Next we create the forward and backward LSTM units

```
# Define lstm cells with tensorflow
    # Forward direction cell
    lstm_fw_cell = tf.nn.rnn_cell.BasicLSTMCell(n_cell_dim, forget_bias=1.0)
    # Backward direction cell
    lstm_bw_cell = tf.nn.rnn_cell.BasicLSTMCell(n_cell_dim, forget_bias=1.0)
```

both of which have inputs of length `n_cell_dim` and bias `1.0` for the forget gate of the LSTM.

The next line of the funtion `BiRNN` does a bit more data preparation.

```
# Split data because rnn cell needs a list of inputs for the RNN inner loop
    layer_3 = tf.split(0, n_steps, layer_3)
```

It splits `layer_3` in to `n_steps` tensors along dimension `0` as the LSTM BRNN expects its input to be of shape `n_steps *[batch_size, 2*n_cell_dim]`.

The next line of `BiRNN`

```
# Get lstm cell output
    outputs, output_state_fw, output_state_bw  = tf.nn.bidirectional_rnn(cell_fw
=lstm_fw_cell,
                                                                        cell_bw
=lstm_bw_cell,
                                                                        inputs=
layer_3,
                                                                        initial
_state_fw=_istate_fw,
                                                                        initial
_state_bw=_istate_bw)
```

feeds `layer_3` to the LSTM BRNN cell and obtains the LSTM BRNN output.

The next lines convert `outputs` from a list of rank two tensors into a rank two tensor in preparation for passing it to the next neural network layer

```
# Reshape outputs from a list of n_steps tensors each of shape [batch_size, 2*n_
cell_dim]
    # to a single tensor of shape [n_steps*batch_size, 2*n_cell_dim]
    outputs = tf.pack(outputs)
    outputs = tf.reshape(outputs, [-1, 2*n_cell_dim])
```

The next couple of lines feed `outputs` to the fifth hidden layer

```
#Hidden layer with clipped RELU activation and dropout
    layer_5 = tf.minimum(tf.nn.relu(tf.add(tf.matmul(outputs, _weights['h5']), _
biases['b5'])), relu_clip)
    layer_5 = tf.nn.dropout(layer_5, keep_prob)
```

The next line of `BiRNN`

```
#Hidden layer with softmax function
    layer_6 = tf.nn.softmax(tf.add(tf.matmul(layer_5, _weights['h6']), _biases[
'b6']))
```

Applies the weight matrix `_weights['h6']` and bias `_biases['h6']`to the output of `layer_5` creating `n_classes` dimensional vectors, then performs softmax on them.

The next lines of `BiRNN`

```
# Reshape layer_6 from a tensor of shape [n_steps*batch_size, n_hidden_6]
    # to a tensor of shape [batch_size, n_steps, n_hidden_6]
    layer_6 = tf.reshape(layer_6, [n_steps, batch_size, n_hidden_6])
    layer_6 = tf.transpose(layer_6, [1, 0, 2])  # Permute n_steps and batch_size
```

reshapes `layer_6` to the slightly more useful shape `[batch_size, n_steps, n_hidden_6]`.

The final line of `BiRNN` returns `layer_6`

```
# Return layer_6
    return layer_6
```

In [19]:

```
layer_6 = BiRNN(x, istate_fw, istate_bw, weights, biases)
```

Loss Function

In accord with Deep Speech: Scaling up end-to-end speech recognition (http://arxiv.org/abs/1412.5567), the loss function used by our network should be the CTC loss function[2] (http://www.cs.toronto.edu/~graves/preprint.pdf). Unfortunately, as of this writing, the CTC loss function[2] (http://www.cs.toronto.edu/~graves/preprint.pdf) is not implemented within TensorFlow[5] (https://github.com/tensorflow/tensorflow/issues/32). Thus we will have to implement it ourselves. The next few sections are dedicated to this implementation.

Introduction

The CTC algorithm was specifically designed for temporal classification tasks; that is, for sequence labelling problems where the alignment between the inputs and the target labels is unknown. Unlike hybrid approaches combining HMM and DNN, CTC models all aspects of the sequence with a single neural network, and does not require the network to be combined with a HMM. It also does not require pre-segmented training data, or external post-processing to extract the label sequence from the network outputs.

Generally, neural networks require separate training targets for every timeslice in the input sequence. This has two important consequences. First, it means that the training data must be pre-segmented to provide targets for every timeslice. Second, as the network only outputs local classifications, global aspects of the sequence, such as the likelihood of two labels appearing consecutively, must be modelled externally. Indeed, without some form of post-processing the final label sequence cannot reliably be inferred at all.

CTC avoids this problem by allowing the network to make label predictions at any point in the input sequence, so long as the overall sequence of labels is correct. This removes the need for pre-segmented data, since the alignment of the labels with the input is no longer important. Moreover, CTC directly outputs the probabilities of the complete label sequences, which means that no external post-processing is required to use the network as a temporal classifier.

From Outputs to Labellings

For a sequence labelling task where the labels are drawn from an alphabet A, CTC consists of a softmax output layer, our `layer_6`, with one more unit than there are labels in `A`. The activations of the first `|A|` units are the probabilities of outputting the corresponding labels at particular times, given the input sequence and the network weights. The activation of the extra unit gives the probability of outputting a $blank$, or no label. The complete sequence of network outputs is then used to define a distribution over all possible label sequences of length up to that of the input sequence.

Defining the extended alphabet $A' = A \cup \{blank\}$, the activation $y_{t,p}$ of network output p at time t is interpreted as the probability that the network will output element p of A' at time t, given the length T input sequence x. Let A'^T denote the set of length T sequences over A'. Then, if we assume the output probabilities at each timestep to be independent of those at other timesteps (or rather, conditionally independent given x), we get the following conditional distribution over $\pi \in A'^T$:

$$\Pr(\pi \mid x) = \prod_{t=1}^{T} y_{t,\pi_t}$$

From now on we refer to the sequences π over A' as *paths*, to distinguish them from the *label sequences* or *labellings* l over A. The next step is to define a many-to-one function $\mathcal{B} : A'^{T} \rightarrow A^{\leq T}$, from the set of paths onto the set $A^{\leq T}$ of possible labellings of x (i.e. the set of sequences of length less than or equal to T over A). We do this by removing first the repeated labels and then the blanks from the paths. For example,

$$\mathcal{B}(a-ab-)amp; = aab$$
$$\mathcal{B}(-aa--abb)amp; = aab.$$

Intuitively, this corresponds to outputting a new label when the network either switches from predicting no label to predicting a label, or from predicting one label to another. As $\mathcal{B}$ is many-to-one, the probability of some labelling $l \in A^{\leq T}$ can be calculated by summing the probabilities of all the paths mapped onto it by $\mathcal{B}$:

$$\Pr(l \mid x) = \sum_{\pi \in \mathcal{B}^{-1}(l)} \Pr(\pi \mid x)$$

This 'collapsing together' of different paths onto the same labelling is what makes it possible for CTC to use unsegmented data, because it allows the network to predict the labels without knowing in advance where they occur. In theory, it also makes CTC networks unsuitable for tasks where the location of the labels must be determined. However in practice CTC tends to output labels close to where they occur in the input sequence.

Role of the Blank Labels

In the original formulation of CTC there were no blank labels, and $\mathcal{B}(\pi)$ was simply π with repeated labels removed. This led to two problems. First, the same label could not appear twice in a row, since transitions only occurred when π passed between different labels. Second, the network was required to continue predicting one label until the next began, which is a burden in tasks where the input segments corresponding to consecutive labels are widely separated by unlabelled data (for example, in speech recognition there are often pauses or non-speech noises between the words in an utterance).

Forward-Backward Algorithm

So far we have defined the conditional probabilities $\Pr(l \mid x)$ of the possible label sequences. Now we need an efficient way of calculating them. At first sight, the previous equation suggests this will be problematic. The sum is over all paths corresponding to a given labelling. The number of these paths grows exponentially with the length of the input sequence. More precisely, for a length T input sequence and a length U labelling there are

$$2^{T-U^2+U(T-3)}\ 3^{(U-1)(T-U)-2}$$

paths.

Fortunately the problem can be solved with a dynamic-programming algorithm similar to the forward-backward algorithm for HMM's[6] (http://www.ee.columbia.edu/~dpwe/e6820/papers/Rabiner89-hmm.pdf). The key idea is that the sum over paths corresponding to a labelling l can be broken down into an iterative sum over paths corresponding to prefixes of that labelling.

To allow for blanks in the output paths, we consider a modified "label sequence" l', with blanks added to the beginning and the end of l, and inserted between every pair of consecutive labels. If the length of l is U, the length of l' is $U' = 2U + 1$. In calculating the probabilities of prefixes of l' we allow all transitions between blank and non-blank labels, and also those between any pair of distinct non-blank labels.

For a labelling l, the forward variable $\alpha(t, u)$ is defined as the summed probability of all length t paths that are mapped by $\mathcal{B}$ onto the length $\lfloor u/2 \rfloor$ prefix of l. (Note, $\lfloor u/2 \rfloor$ is the *floor* of $u/2$, the greatest integer less than or equal to $u/2$.) For some sequence s, let $s_{p:q}$ denote the subsequence s_p, s_{p+1}, ..., s_{q-1}, s_q, and define the set $V(t,u) \equiv \{\pi \in A'^t : \mathcal{B}(\pi) = l_{1:\lfloor u/2 \rfloor} \text{ and } \pi_t = l'_u\}$. We can then define $\alpha(t, u)$ as

$$\alpha(t,u) \equiv \sum_{\pi \in V(t,u)} \prod_{i=1}^{t} y_{i,\pi_i}$$

As we will see, the forward variables at time t can be calculated recursively from those at time $t-1$.

Given the above formulation, the probability of l can be expressed as the sum of the forward variables with and without the final blank at time T.

$$\Pr(l \mid x) = \alpha(T, U') + \alpha(T, U' - 1)$$

All correct paths must start with either a blank (b) or the first symbol in l (l_1), yielding the following initial conditions:

$$\begin{aligned} \alpha(1,1) amp;&= y_{1,b} \\ \alpha(1,2) amp;&= y_{1,l_1} \\ \alpha(1,u) amp;&= 0, \ \forall u \quad gt; 2 \end{aligned}$$

Thereafter the variables can be calculated recursively:

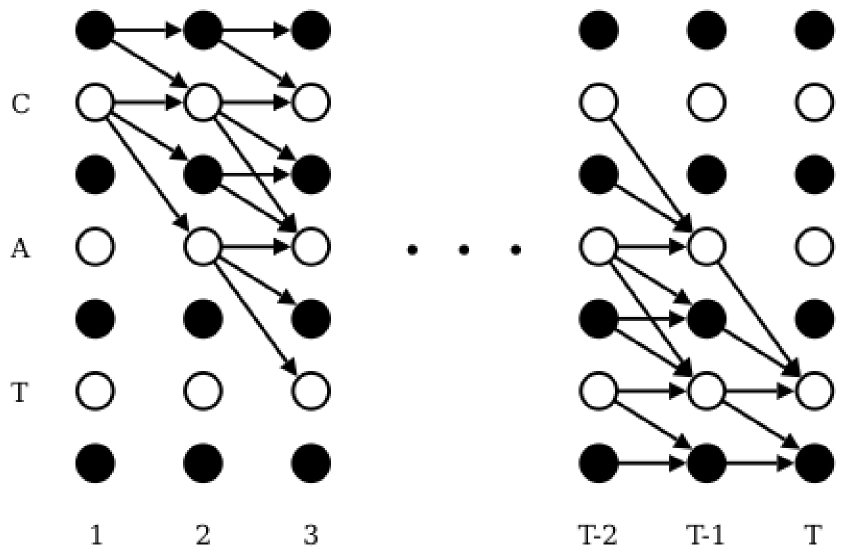
$$\alpha(t,u) = y_{t,l'_u} \sum_{i=f(u)}^{u} \alpha(t-1, i)$$

where

$$f(u) = \begin{cases} u-1, & amp; \text{if } l'_u = blank \text{ or } l'_{u-2} = l'_u \\ u-2, & amp; \text{otherwise} \end{cases}$$

which one can derive by expanding $\alpha(t, u)$ and substituting $\alpha(t-1, u)$ into the expansion.

Graphically we can express the recurrence relation for $\alpha(t, u)$ as follows



where t runs along the x axis and u runs along the y axis. The black circles of the diagram represent $blank$ elements of l' while the white circles represent non-$blank$ elements of l'. The arrows represent computational dependencies derived from our recursion relation for $\alpha(t, u)$. So, for example, the value of $\alpha(2, 3)$, corresponding to the $blank$ at $t = 2$ and $u = 3$, is derived from $\alpha(1, 2)$. Similarly, the value of $\alpha(2, 2)$, corresponding to the letter c at $t = 2$ and $u = 2$, is derived from $\alpha(1, 2)$ and $\alpha(1, 1)$.

Note also that

\alpha(t, u) = 0 \,\, \forall u &lt; U′ − 2(T − t) − 1

because these variables correspond to states for which there are not enough timesteps left to complete the sequence. We also impose the boundary condition

$$\alpha(t, 0) = 0 \ \forall t$$

The backward variables $\beta(t, u)$ are defined as the summed probabilities of all paths starting at $t + 1$ that "complete" l when appended to any path $\hat{\pi}$ contributing to $\alpha(t, u)$. Define $W(t, u) \equiv \{\pi \in A'^{T-t} : \mathcal{B}(\hat{\pi} + \pi) = l \ \forall \hat{\pi} \in V(t, u)\}$. Then

$$\beta(t, u) \equiv \sum_{\pi \in W(t,u)} \prod_{i=1}^{T-t} y_{t+i, \pi_i}$$

The rules for initialisation of the backward variables are as follows

$$\beta(T, U')amp; = 1$$
$$\beta(T, U' - 1)amp; = 1$$
$$\beta(T, u)amp; = 0, \ \forall u \qquad lt; U' - 1$$

The rules for recursion are as follows

$$\beta(t, u) = \sum_{i=u}^{g(u)} \beta(t+1, i) y_{t+1, l'_i}$$

where

$$g(u) = \begin{cases} u + 1, & amp; \text{if } l'_u = blank \text{ or } l'_{u+2} = l'_u \\ u + 2, & amp; \text{otherwise} \end{cases}$$

Note that

\beta(t, u) = 0 \, \, \forall u &gt; 2t

and

$$\beta(t, U' + 1) = 0 \ \forall t$$

Log Scale

In practice, the above recursions will soon lead to underflows on any digital computer. A good way to avoid this is to work in the log scale, and only exponentiate to find the true probabilities at the end of the calculation. A useful equation in this context is

$$\ln(a + b) = \ln(a) + \ln(1 + e^{\ln b - \ln a})$$

which allows the forward and backward variables to be summed while remaining in the log scale.

Loss Function

The CTC loss function $\mathcal{L}(S)$ is defined as the negative log probability of correctly labelling all the training examples in some training set S:

$$\mathcal{L}(S) = -\ln \prod_{(x,z) \in S} \Pr(z \mid x) = -\sum_{(x,z) \in S} \ln \Pr(z \mid x)$$

Because the function is differentiable, its derivatives with respect to the network weights can be calculated with backpropagation through time, and the network can then be trained with any gradient-based non-linear optimisation algorithm.

We also define the *example loss*

$$\mathcal{L}(x,z) \equiv -\ln \Pr(z \mid x)$$

Obviously

$$\mathcal{L}(S) = \sum_{(x,z)\in S} \mathcal{L}(x,z)$$

Now if we identify l and z and define $X(t,u) \equiv \{\pi \in A'^{T} : \mathcal{B}(\pi) = z,\ \pi_t = z'_u\}$, then our definition of $\alpha(t,u)$ and $\beta(t,u)$ imply

$$\alpha(t,u)\beta(t,u) = \sum_{\pi\in X(t,u)} \prod_{t=1}^{T} y_{t,\pi_t}$$

thus substituting our previous expression for $\Pr(\pi \mid x)$

$$\alpha(t,u)\beta(t,u) = \sum_{\pi\in X(t,u)} \Pr(\pi \mid x)$$

From our expression for $\Pr(l \mid x)$ we can see that this is the portion of the total probability of $\Pr(z \mid x)$ due to those paths going through z'_u at time t. For any t, we can therefore sum over all u to get

$$\Pr(z \mid x) = \sum_{u=1}^{|z'|} \alpha(t,u)\beta(t,u)$$

Thus the *example loss* is given by

$$\mathcal{L}(x,z) = -\ln \sum_{u=1}^{|z'|} \alpha(t,u)\beta(t,u)$$

As

$$\mathcal{L}(S) = \sum_{(x,z)\in S} \mathcal{L}(x,z)$$

the gradient of $\mathcal{L}(S)$ can be computed by computing the gradient of $\mathcal{L}(x,z)$. This gradient can be computed using the formulas above and TensorFlow's automatic differentiation.

In [20]:

```
# cost = .... TODO: Compute the cost using the above formula
```

Decoding

Once the network is trained, we would ideally label some unknown input sequence x by choosing the most probable labelling l^*:

$$l^* \equiv \underset{l}{\operatorname{argmax}} \Pr(l \mid x)$$

Using the terminology of HMM's, we refer to the task of finding this labelling as *decoding*. Unfortunately, we do not know of a general, tractable decoding algorithm for CTC. However we now present two approximate methods that work well in practice.

Best Path Decoding

The first method, which refer to as *best path decoding*, is based on the assumption that the most probable path corresponds to the most probable labelling

$$l^* \approx \mathcal{B}(\pi^*)$$

where

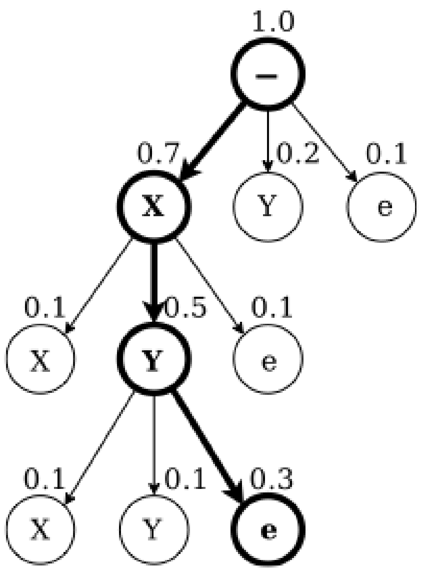
$$\pi^* \equiv \underset{\pi}{\operatorname{argmax}} \Pr(\pi \mid x)$$

Best path decoding is trivial to compute, since π^* is just the concatenation of the most active outputs at every timestep. However it can lead to errors, particularly if a label is weakly predicted for several consecutive timesteps.

Prefix Search Decoding

The second method (prefix search decoding) relies on the fact that, by modifying the forward variables $\alpha(t, u)$, we can efficiently calculate the probabilities of successive extensions of labelling prefixes.

Prefix search decoding is a best-first search through the tree of labellings, where the children of a given labelling are those that share it as a prefix. At each step the search extends the labelling whose children have the largest cumulative probability. This is illustrated in the following diagram



Here each node either ends (‘e’) or extends the prefix at its parent node. The number above an extending node is the total probability of all labellings beginning with that prefix. The number above an end node is the probability of the labelling ending at its parent. At every iteration the extensions of the most probable remaining prefix are explored. Search ends when a single labelling (here ‘XY’) is more probable than any remaining prefix.

Let $\gamma(p_n, t)$ be the probability of the network outputting prefix p by time t such that a non-blank label is output at t, let $\gamma(p_b, t)$ be the probability of the network outputting prefix p by time t such that the blank label is output at t, and let the set $Y = \{\pi \in A'^t : \mathcal{B}(\pi) = p\}$. Then

$$\gamma(p_n, t) amp; = \sum_{\pi \in Y : \pi_t = p_{|p|}} \Pr(\pi \mid x)$$
$$\gamma(p_b, t) amp; = \sum_{\pi \in Y : \pi_t = blank} \Pr(\pi \mid x)$$

Thus, for a length T input sequence x, $\Pr(p \mid x) = \gamma(p_n, T) + \gamma(p_b, T)$. Also let $\Pr(p\ldots \mid x)$ be the cumulative probability of all labellings not equal to p of which p is a prefix, then

$$\Pr(p\ldots \mid x) = \sum_{l \neq \emptyset} \Pr(p + l \mid x)$$

where $\emptyset$ denotes the empty sequence. With these definitions is mind, the pseudocode for prefix search decoding is given as follows:

```
**Initialisation:**
$1 \leq t \leq T \begin{cases} \gamma(\emptyset_n, t) = 0 \\ \gamma(\emptyset_b, t) = \prod_{t'=1}^{t} y_b^{t'} \end{cases}$
$p(\emptyset|\mathbf{x}) = \gamma(\emptyset_b, T)$
$p(\emptyset\ldots|\mathbf{x}) = 1 - p(\emptyset|\mathbf{x})$
$\mathbf{l}^* = \mathbf{p}^* = \emptyset$
$P = \{\emptyset\}$

**Algorithm:**
**while** $p(\mathbf{p}^*\ldots|\mathbf{x}) > p(\mathbf{l}^*|\mathbf{x})$ **do**
    $probRemaining = p(\mathbf{p}^*\ldots|\mathbf{x})$
    **for** all labels $k \in A$ **do**
        $\mathbf{p} = \mathbf{p}^* + k$
        $\gamma(\mathbf{p}_n, 1) = \begin{cases} y_k^1 \text{ if } \mathbf{p}^* = \emptyset \\ 0 \text{ otherwise} \end{cases}$
        $\gamma(\mathbf{p}_b, 1) = 0$
        $prefixProb = \gamma(\mathbf{p}_n, 1)$
        **for** $t = 2$ to $T$ **do**
            $newLabelProb = \gamma(\mathbf{p}_b^*, t-1) + \begin{cases} 0 \text{ if } \mathbf{p}^* \text{ ends in } k \\ \gamma(\mathbf{p}_n^*, t-1) \text{ otherwise} \end{cases}$
            $\gamma(\mathbf{p}_n, t) = y_k^t\,(newLabelProb + \gamma(\mathbf{p}_n, t-1))$
            $\gamma(\mathbf{p}_b, t) = y_b^t\,(\gamma(\mathbf{p}_b, t-1) + \gamma(\mathbf{p}_n, t-1))$
            $prefixProb \mathrel{+}= y_k^t\; newLabelProb$
        $p(\mathbf{p}|\mathbf{x}) = \gamma(\mathbf{p}_n, T) + \gamma(\mathbf{p}_b, T)$
        $p(\mathbf{p}\ldots|\mathbf{x}) = prefixProb - p(\mathbf{p}|\mathbf{x})$
        $probRemaining \mathrel{-}= p(\mathbf{p}\ldots|\mathbf{x})$
        **if** $p(\mathbf{p}|\mathbf{x}) > p(\mathbf{l}^*|\mathbf{x})$ **then**
            $\mathbf{l}^* = \mathbf{p}$
        **if** $p(\mathbf{p}\ldots|\mathbf{x}) > p(\mathbf{l}^*|\mathbf{x})$ **then**
            add $\mathbf{p}$ to $P$
        **if** $probRemaining \leq p(\mathbf{l}^*|\mathbf{x})$ **then**
            break
    remove $\mathbf{p}^*$ from $P$
    $\mathbf{p}^* = \arg\max_{\mathbf{p} \in P}\; p(\mathbf{p}\ldots|\mathbf{x})$

**Termination:**
output $\mathbf{l}^*$
```

Given enough time, prefix search decoding always finds the most probable labelling. However, the maximum number of prefixes it must expand grows exponentially with the input sequence length. If the output distribution is sufficiently peaked around the mode, it will still finish in reasonable time. But for many tasks, a heuristic is required to make its application feasible.

Observing that the outputs of a trained CTC network tend to form a "series of spikes separated by strongly predicted blanks", we can divide the output sequence into sections that are very likely to begin and end with a blank. We do this by choosing boundary points where the probability of observing a blank label is above a certain threshold. We then apply the prefix search decoding algorithm to each section individually and concatenate these to get the final transcription.

In practice, prefix search works well with this heuristic, and generally outperforms best path decoding. However it still makes mistakes in some cases, for example if the same label is predicted weakly on both sides of a section boundary.

Constrained Decoding

For certain tasks we want to constrain the output labellings according to some predefined grammar. For example, in speech and handwriting recognition, the final transcriptions are usually required to form sequences of dictionary words. In addition it is common practice to use a language model to weight the probabilities of particular sequences of words.

We can express these constraints by altering the label sequence probabilities in

$$l^* \equiv \underset{l}{\operatorname{argmax}} \Pr(l \mid x)$$

to be conditioned on some probabilistic grammar G, as well as the input sequence x.

$$l^* \equiv \underset{l}{\operatorname{argmax}} \Pr(l \mid x, G)$$

Absolute requirements, for example that l contains only dictionary words, can be incorporated by setting the probability of all sequences that fail to meet them to 0.

At first sight, conditioning on G would seem to contradict a basic assumption of CTC: that the labels are conditionally independent given the input sequences. Since the network attempts to model the probability of the whole labelling at once, there is nothing to stop it from learning inter-label transitions direct from the data, which would then be skewed by the external grammar. Indeed, when we tried using a biphone model to decode a CTC network trained for phoneme recognition, the error rate increased. However, CTC networks are typically only able to learn local relationships such as commonly occurring pairs or triples of labels. Therefore as long as G focuses on long range label dependencies (such as the probability of one word following another when the outputs are letters) it doesn’t interfere with the dependencies modelled internally by CTC.

Applying the basic rules of probability we obtain

$$\Pr(l \mid x, G) = \frac{\Pr(l \mid x)\Pr(l \mid G)\Pr(x)}{\Pr(x \mid G)\Pr(l)}$$

where we have used the fact that x is conditionally independent of G given l. If we assume that x is independent of G, this reduces to

$$\Pr(l \mid x, G) = \frac{\Pr(l \mid x)\Pr(l \mid G)}{\Pr(l)}$$

This assumption is in general false, since both the input sequences and the grammar depend on the underlying generator of the data, for example the language being spoken. However it is a reasonable first approximation, and is particularly justifiable in cases where the grammar is created using data other than that from which x was drawn (as is common practice in speech and handwriting recognition, where separate textual corpora are used to generate language models).

If we further assume that, prior to any knowledge about the input or the grammar, all label sequences are equally probable, then

$$l^* \equiv \operatorname*{argmax}_l \Pr(l \mid x, G)$$

reduces to

$$l^* \equiv \operatorname*{argmax}_l \Pr(l \mid x)\Pr(l \mid G)$$

Note that, since the number of possible label sequences is finite (because both A and S are finite), assigning equal prior probabilities does not lead to an improper prior.

CTC Token Passing Algorithm

We now describe an algorithm, based on the *token passing algorithm* for HMMs[7] (ftp://mi.eng.cam.ac.uk/pub/reports/auto-pdf/young_tr38.pdf), that finds an approximate solution to the previous equation for a simple grammar.

Let G consist of a dictionary D containing W words, and an optional set of W^2 bigrams $\Pr(w \mid \hat{w})$ that define the probability of making a transition from word $\hat{w}$ to word w. The probability of any label sequence that does not form a sequence of dictionary words is 0.

For each word w, define the modified word w' as w with blanks added at the beginning and end and between each pair of labels. Therefore $|w'| = 2|w| + 1$. Define a token $tok = (score, history)$ to be a pair consisting of a real valued 'score' and a 'history' of previously visited words. The history corresponds to the path through the network outputs the token has taken so far, and the score is the log probability of that path. The basic idea of the token passing algorithm is to pass along the highest scoring tokens at every word state, then maximise over these to find the highest scoring tokens at the next state. The transition probabilities are used when a token is passed from the last state in one word to the first state in another. The output word sequence is then given by the history of the highest scoring end-of-word token at the final timestep.

At every timestep t of the length T output sequence, each segment s of each modified word w' holds a single token $tok(w, s, t)$. This is the highest scoring token reaching that segment at that time. Define the input token $tok(w, 0, t)$ to be the highest scoring token arriving at word w at time t, and the output token $tok(w, -1, t)$ to be the highest scoring token leaving word w at time t. $\emptyset$ denotes the empty sequence.

The pseudocode for the algorithm is here

```
**Initialisation:**
for all words $w \in D$ do
    $tok(w,1,1) = (\ln y^1_b, (w))$
    $tok(w,2,1) = (\ln y^1_{w_1}, (w))$
    if $|w| = 1$ then
        $tok(w,-1,1) = tok(w,2,1)$
    else
        $tok(w,-1,1) = (-\infty, \emptyset)$
    $tok(w,s,1) = (-\infty, \emptyset)$ for all other $s$

**Algorithm:**
for $t = 2$ to $T$ do
    if using bigrams then
        sort output tokens $tok(w,-1,t-1)$ by ascending score
    else
        find single highest scoring output token
    for all words $w \in D$ do
        if using bigrams then
            $w^* = \arg\max_{\hat{w}} [tok(\hat{w},-1,t-1).score + \ln p(w|\hat{w})]$
            $tok(w,0,t) = tok(w^*,-1,t-1)$
            $tok(w,0,t).score \mathrel{+}= \ln p(w|w^*)$
        else
            $tok(w,0,t) =$ highest scoring output token
        add $w$ to $tok(w,0,t).history$
        for segment $s = 1$ to $|w'|$ do
            $P = \{tok(w,s,t-1), tok(w,s-1,t-1)\}$
            if $w'_s \neq blank$ and $s > 2$ and $w'_{s-2} \neq w'_s$ then
                add $tok(w,s-2,t-1)$ to $P$
            $tok(w,s,t) =$ token in $P$ with highest score
            $tok(w,s,t).score \mathrel{+}= \ln y^t_{w'_s}$
        $tok(w,-1,t) =$ highest scoring of $\{tok(w,|w'|,t), tok(w,|w'|-1,t)\}$

**Termination:**
$w^* = \arg\max_w tok(w,-1,T).score$
output $tok(w^*,-1,T).history$
```

Computational Complexity

If bigrams are used, the CTC token passing algorithm has a worst-case complexity of $\mathcal{O}(TW^2)$, since line 19 requires a potential search through all W words. However, because the output tokens $tok(w, -1, T)$ are sorted in order of score, the search can be terminated when a token is reached whose score is less than the current best score with the transition included. The typical complexity is therefore considerably lower, with a lower bound of $\mathcal{O}(TW \log W)$ to account for the sort.

If no bigrams are used, the single most probable output token at the previous timestep will form the new input token for all the words, and the worst-case complexity reduces to $\mathcal{O}(TW)$.

Single Word Decoding

If the number of words in the target sequence is fixed, the previous algorithm can be constrained by forbidding all tokens whose history already contains that many words from transitioning to new words. In particular, if the target sequences are constrained to be single words, then all word-to-word transitions are forbidden (and bigrams are clearly not required).

In general the extension from finding the single best transcription to the N-best transcriptions is complex. However, in the special case of single word decoding, the N-best transcriptions are simply the (single word) histories of the N-best output tokens when the algorithm terminates.

Another straightforward extension to single word decoding occurs when the same word has several different label transcriptions. This happens, for example, when pronunciation variants are considered in speech recognition, or spelling variants are allowed in handwriting recognition. In that case all variants should be considered separate words until the termination of the previous algorithm (lines 34 and 34); at that point the scores of all variant transcriptions of each word should be added together in the log scale; thereafter the best or N-best words should be found as usual.