DEEP SPEAKER: An End-to-End Neural Speaker Embedding System

INTRODUCTION

The idea of Deep Speaker revolves around the idea or the proposition of Statistical Language Modelling. The objective is to learn the joint probability function of sequences of words in a language. The biggest barrier in achieving this objective is due to the 'Curse of Dimensionality'. On an advanced level, we are trying to achieve Speaker Identification and Succesful Verification using proper Clustering and Sampling in order to achieve high percentage of accuracy.

One way of Speaker Identification is to collect sufficient samples of audio data in which there has been more than 95% of noise cancellation. Now, we make the training data set with certain necessary columns of 'Pitch' , 'Frequency', 'Amplitude' and voice characteristics such as these which vary from 1 person to the other or at least could help us in identifying properly the various accents of a particular language in different regions of the world. After successful training using ResCNN and GRE models, we could apply them into test datasets and check for accuracy before finally implementing it.

The model learns simultaneously:

(1) a distributed representation for each word along with.

(2) the probability function for word sequences, expressed in terms of these representations. Generalization is obtained because a sequence of words that has never been seen before gets high probability if it is made of words that are similar to words forming an already seen sentence.

For this, using the 'n-grams' model is a good choice.

For example, if one wants to model the joint distribution of 10 consecutive words in a natural language with a vocabulary V of size 100,000, there are potentially (100000)^10 − 1 = (10)^50 − 1 free parameters.

A statistical model of language can be represented by the conditional probability of the next word given all the previous ones, since

$$\hat{P}(w_1^T) = \prod_{t=1}^{T} \hat{P}(w_t | w_1^{t-1})$$

w(t): the 't' th word,

When building statistical models of natural language, one considerably reduces the difficulty of this modeling problem by taking advantage of word order, and the fact that temporally closer words in the word sequence are statistically more dependent. Thus, n-gram models construct tables of conditional probabilities for the next word, for each one of a large number of contexts, i.e. combinations of the last (n−1) words:

$$\hat{P}(w_t | w_1^{t-1}) \approx \hat{P}(w_t | w_{t-n+1}^{t-1}).$$

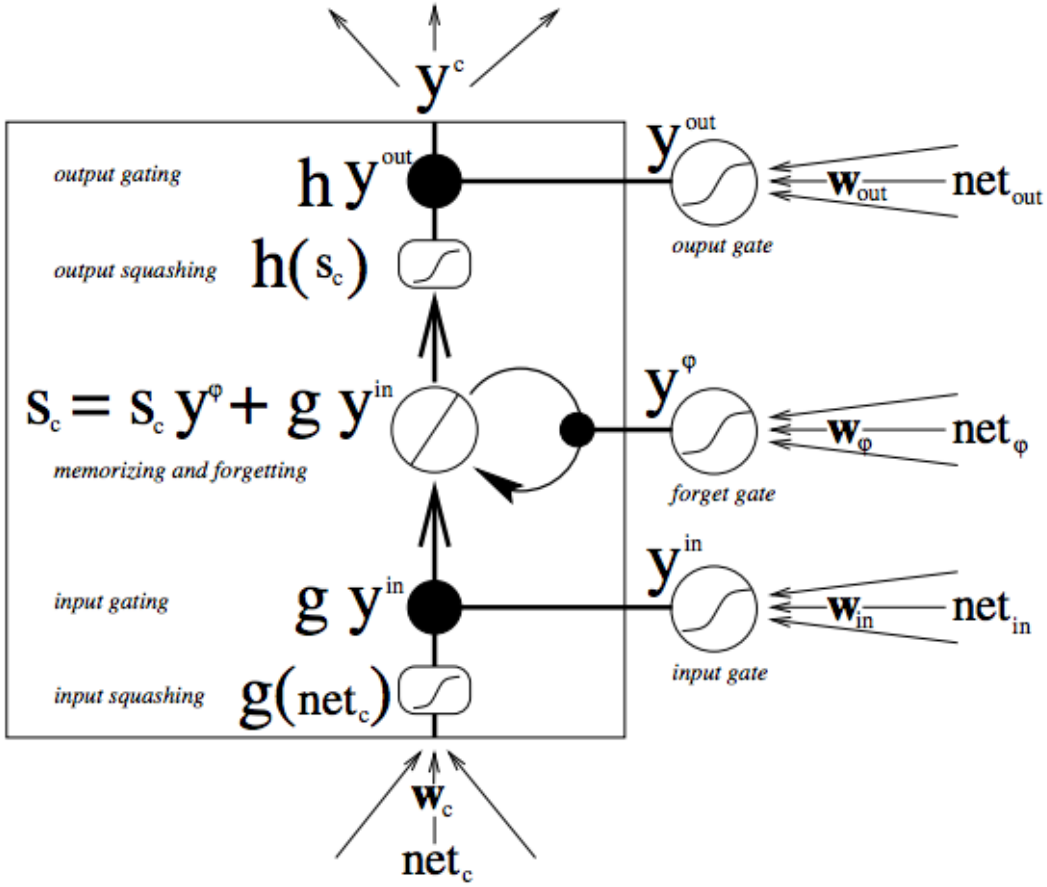
STEPS:

a) Collect sufficient statistics from training datasets.
b) Extract Speaker Embeddings
c) Classify

NETWORK ARCHITECTURE

The most basic neural network architecture that we use is LSTM (Long Short-Term Memory). One of the reasons training networks is difficult is that the errors computed in backpropoagation are multiplied by each other once per timestep. If the errors are small, the error quickly dies out, becoming very small; if the errors are large, they quickly become very large due to repeated multiplication. An alternative architecture built with *Long Short-Term Memory* (LSTM) cells attempts to negate this issue.

An LSTM Model Framework is as below:



But, LSTM is not a sufficiently good model for the objectives of the Deep Speaker Neural System. In simple words, continuous incoming words from a speaker or a person has to be captured and has to be remembered for a longer period of time due to the vastness of the vocabulary of the input and thhe output language. A short-term memory model system will never be designed for that vast

period of time so as to continuously train the words and compute the probability of certain words in a particular sentence. Also, if we increase the time difference between the input gate and the forget gate thus trying to increase the 'Remember Time', it will result in very high time complexity which can't be easily optimized and therefore reducing the accuracy to a great extent.

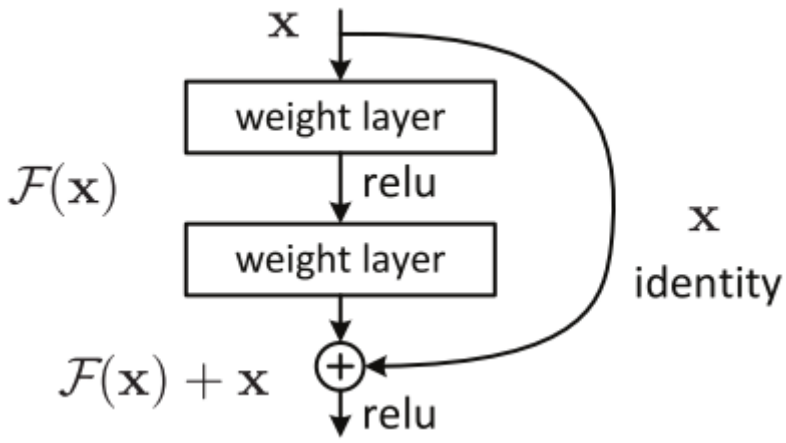
Therefore for solving this problem, we will be using primarily 2 architectures :

a) ResCNN
b) GRU

a) ResCNN:

This network architecture uses a concept which is known as 'Skip Connections'. Typically, the input matrix calculates in two linear transformation with ReLU activation function. In Residual network, it directly copies the input matrix to the second transformation output and sum the output in final ReLU function.

The Resnet CNN model looks like this:



The ResNet Block can be formulized as:

$$h = F(x, W_i) + x,$$

x : Input Layer
h : Output Layer
F : Mapping Function

Finding Optimal h(x_{l}) Function:

$$y = (w_2 + \lambda_2) * (w_1 + \lambda_1) * (w_0 + \lambda_0) * x$$

$$\frac{\partial E}{\partial x_o} = \frac{\partial E}{\partial x_3} * \frac{\partial x_3}{\partial x_2} * \frac{\partial x_2}{\partial x_1} * \frac{\partial x_1}{\partial x_0}$$

$$\frac{\partial E}{\partial x_o} = \frac{\partial E}{\partial x_3} * (w_2 + \lambda_2) * (w_1 + \lambda_1) * (w_0 + \lambda_0)$$

Backprop of the ResNet module

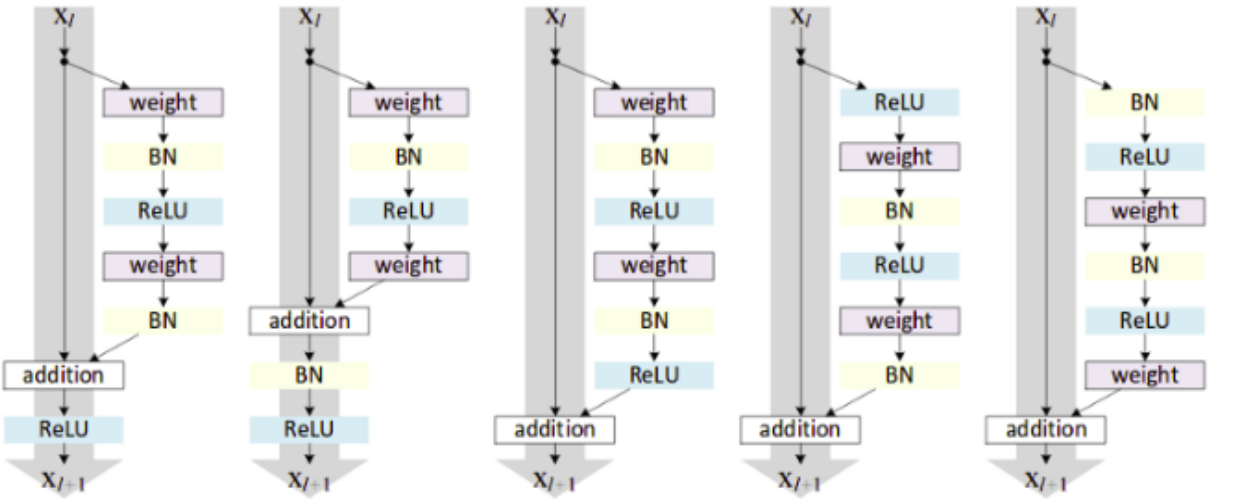
Case-1, Lambda = 0: This will be a plain network. Since w2, w1, w0 are all between {-1, 1}, the gradient vanishes as the network depth increases. This clearly shows vanishing gradient problem

Case-2, Lambda >1: In this case, The backprop value increases incrementally and lead to exploding of gradients.

Case-3, Lambda <1: For shallow networks this might not be problem. But for extra large networks, weight+lambda is still less than <1 in most cases and it achieves the same problem as case-1

case-4, Lambda =1: In this case, Every weight is incremented by 1, This eliminates the problem of multiplying with very large numbers as in case-2 and small numbers as in case-1 and acts as a good barrier.

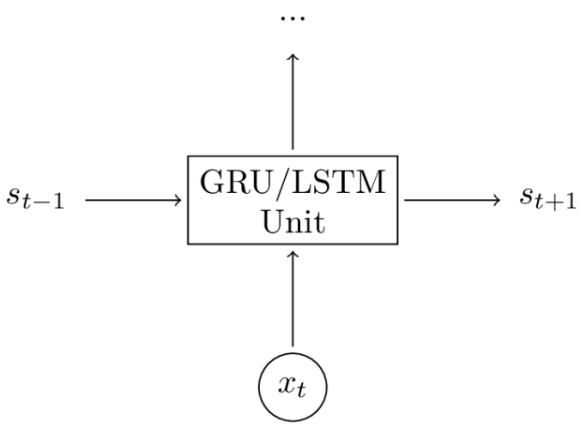
Finding Optimal f(y_{l}) Function:



Different Residual blocks

b) GRU Architecture:

Another important architecture that we use for the objectives of the Deep Speaker system is the GRU Architecture.



Equations:

$$z = \sigma(x_t U^z + s_{t-1} W^z)$$
$$r = \sigma(x_t U^r + s_{t-1} W^r)$$
$$h = tanh(x_t U^h + (s_{t-1} \circ r) W^h)$$
$$s_t = (1 - z) \circ h + z \circ s_{t-1}$$

Example Python Notebook Code :

```
def forward_prop_step(x_t, s_t1_prev):
      # This is how we calculated the hidden state in a simple RNN. No longer!
      # s_t = T.tanh(U[:,x_t] + W.dot(s_t1_prev))

      # Get the word vector
      x_e = E[:,x_t]

      # GRU Layer
      z_t1 = T.nnet.hard_sigmoid(U[0].dot(x_e) + W[0].dot(s_t1_prev) + b[0])
      r_t1 = T.nnet.hard_sigmoid(U[1].dot(x_e) + W[1].dot(s_t1_prev) + b[1])
      c_t1 = T.tanh(U[2].dot(x_e) + W[2].dot(s_t1_prev * r_t1) + b[2])
      s_t1 = (T.ones_like(z_t1) - z_t1) * c_t1 + z_t1 * s_t1_prev

      # Final output calculation
      # Theano's softmax returns a matrix with one row, we only need the row
      o_t = T.nnet.softmax(V.dot(s_t1) + c)[0]

      return [o_t, s_t1]
```

```
cacheW = decay * cacheW + (1 - decay) * dW ** 2
W = W - learning_rate * dW / np.sqrt(cacheW + 1e-
6)
```

All the layers are here:

```
# GRU Layer 1
z_t1 = T.nnet.hard_sigmoid(U[0].dot(x_e) + W[0].dot(s_t1_prev) + b[0])
r_t1 = T.nnet.hard_sigmoid(U[1].dot(x_e) + W[1].dot(s_t1_prev) + b[1])
c_t1 = T.tanh(U[2].dot(x_e) + W[2].dot(s_t1_prev * r_t1) + b[2])
s_t1 = (T.ones_like(z_t1) - z_t1) * c_t1 + z_t1 * s_t1_prev

# GRU Layer 2
z_t2 = T.nnet.hard_sigmoid(U[3].dot(s_t1) + W[3].dot(s_t2_prev) + b[3])
r_t2 = T.nnet.hard_sigmoid(U[4].dot(s_t1) + W[4].dot(s_t2_prev) + b[4])
c_t2 = T.tanh(U[5].dot(s_t1) + W[5].dot(s_t2_prev * r_t2) + b[5])
s_t2 = (T.ones_like(z_t2) - z_t2) * c_t2 + z_t2 * s_t2_prev
```

c) N-Grams Model:

Core Codes:

i)
```
def ngrams(input, n):
    input = input.split(' ')
    output = []
    for i in range(len(input)-n+1):
        output.append(input[i:i+n])
    return output

ngrams('a b c d', 2) # [['a', 'b'], ['b', 'c'], ['c', 'd']]
```

ii)
```
for g in (' '.join(x) for x in ngrams(input, 2)):
```

```
    grams.setdefault(g, 0)
    grams[g] += 1
```

iii)
```
def ngrams(input, n):
  input = input.split(' ')
  output = {}
  for i in range(len(input)-n+1):
     g = ' '.join(input[i:i+n])
     output.setdefault(g, 0)
     output[g] += 1
  return output

ngrams('a a a a', 2) # {'a a': 3}
```

An n-gram is a contiguous sequence of n items from a given sequence of text. Given a sentence, s, we can construct a list of n-grams from (s) by finding pairs of words that occur next to each other. For example, given the sentence “The cat is walking in the bedroom” and “The dog was running in a room” ,we can construct bigrams (n-grams of length 3) by finding consecutive pairs of words.

Probability Model:

P(x1, x2, ..., xn) = P(x1)P(x2|x1)...P(xn|x1,...xn-1)

CONCLUSION

These models are therefore the perfect way to go on with the Deep Speaker Neural system. The system can be optimized using better algorithms based on the same models and architecture.