Deep speech

Vincent Rébiscoul, Stéphane Pouget and Florent Guépin

This document present our project in machine learning. We have implemented a voice recognition system i.e. our program is able to recognize spoken language and translate into text by computers. We use python3, Keras and Tensorflow.

Introduction

We have implemented this article (*1*) with python3, Keras and Tensorflow. The neural network used is not common. So we created our own neural network model.

1 Model

1.1 Topology of the model

The core of the system is a recurrent neural network trained to ingest speech spectograms and generate English text transcriptions.

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

We have seven layers of neuron. The three first layers are computed by :

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

where $g(z) = \min\{\max\ \{0, z\}, 20\}$ and $W^{(l)}, b^l$ are the weight matrix and bias parameters for layers l.

The fourth layer is a bi-directional reccurent layer. This layer includes two sets of hidden units : a set with forward reccurence, $h^{(f)}$, and a set with backward recurrence $h^{(b)}$:

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

$$h_t^{(b)} = g(W^{(4)} h_t^{(3)} + W_r^{(b)} h_{t+1}^{(b)} + b^{(4)})$$

The fifth (non-recurrent) layer takes both the forward and backward units as inputs $h_t^{(5)} = g(W^{(5)} h_t^{(4)} + b^{(5)}$ where $h_t^{(4)} = h_t^{(f)} + h_t^{(b)}$. The output layer is a standar softmax function that yields the predicted character probabilities for each time slice t and character k in the alphabet :

$$h_{t,k}^{(6)} \equiv \mathbb{P}(c_t = k|x) = \frac{exp(W_k^{(6)} h_t^{(5)} + b_k^{(6)})}{\sum_j exp(W_j^{(6)} h_t^{(5)} + b_j^{(6)})}$$

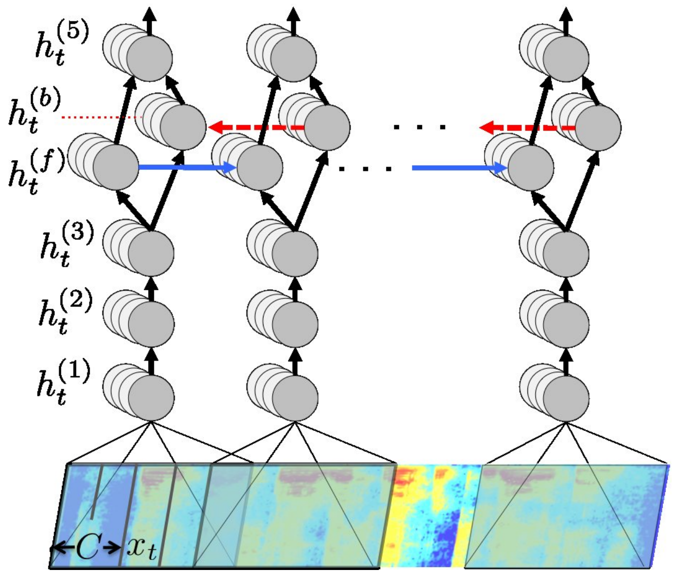


Figure 1: Structure of our RNN model

1.2 Analysis of the model

The model we use is a Recurrent Neural Network (RNN), the particularity of a recurrent neural network is that recurrent layer has a state and for to compute the value of a time slice, the layer can take the output value of the same layer one step before or one step after. In our case, we have two recurrent layers, with h_t^f depends on $t-1$ (so h_t^f depends on what happened previously). h_t^b depends on $t+1$ so it depends on what will happen next. The idea is that we cannot link a sound to a letter without putting it in context. For example, in the word ”the”, our neural network cannot predict that the first letter is t because t has several pronounciation depending on the letter preceding of following. This is why we need two recurrent layers, one with a forwad recurrence and one with a backward recurrence.

2 Our work

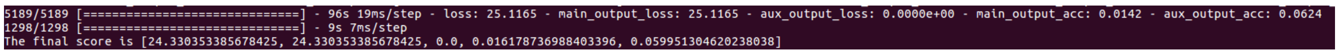
2.1 The implementation of the model

This article (*1*) create a new model and use a ctc loss function. So to create this model, we have customized our model so that it is like on the article. For that we had to work a lot on the documentation of keras and tensorflow. However our main problem was the ctc loss function. At the beginning, we had troubles to understand how the backend function ctc_batch_cost worked. First, one has to know that the CTC loss function is very peculiar. Indeed, our neural network slices an audio file in several time slices and then for each time slice, it tries to predict which letter it is. Now, suppose you give a recording of someone saying the expression “good morning”. If the neural network is correctly trained, it should output something like “gggoooodd moorrnnninng” (there are several time slices corresponding to the same letter), thus the cost $\mathcal{L}(\text{good morning}, \text{gggoooodd moorrnnninng})$ should be small when the cost $\mathcal{L}(\text{good morning}, \text{good mornint})$ should be higher. This makes the CTC function essential but complicated. Another problem is that the ctc_batch_cost function takes four parameters, but in Keras, the loss function only takes 2 parameters. To avoid this problem, we had to create a new lambda layer which outputs the loss of the batch and is connected to several other layers which feeds the different arguments that are needed. Apparently, this is a known trick in Keras to avoid a common problem.

2.2 The dataset

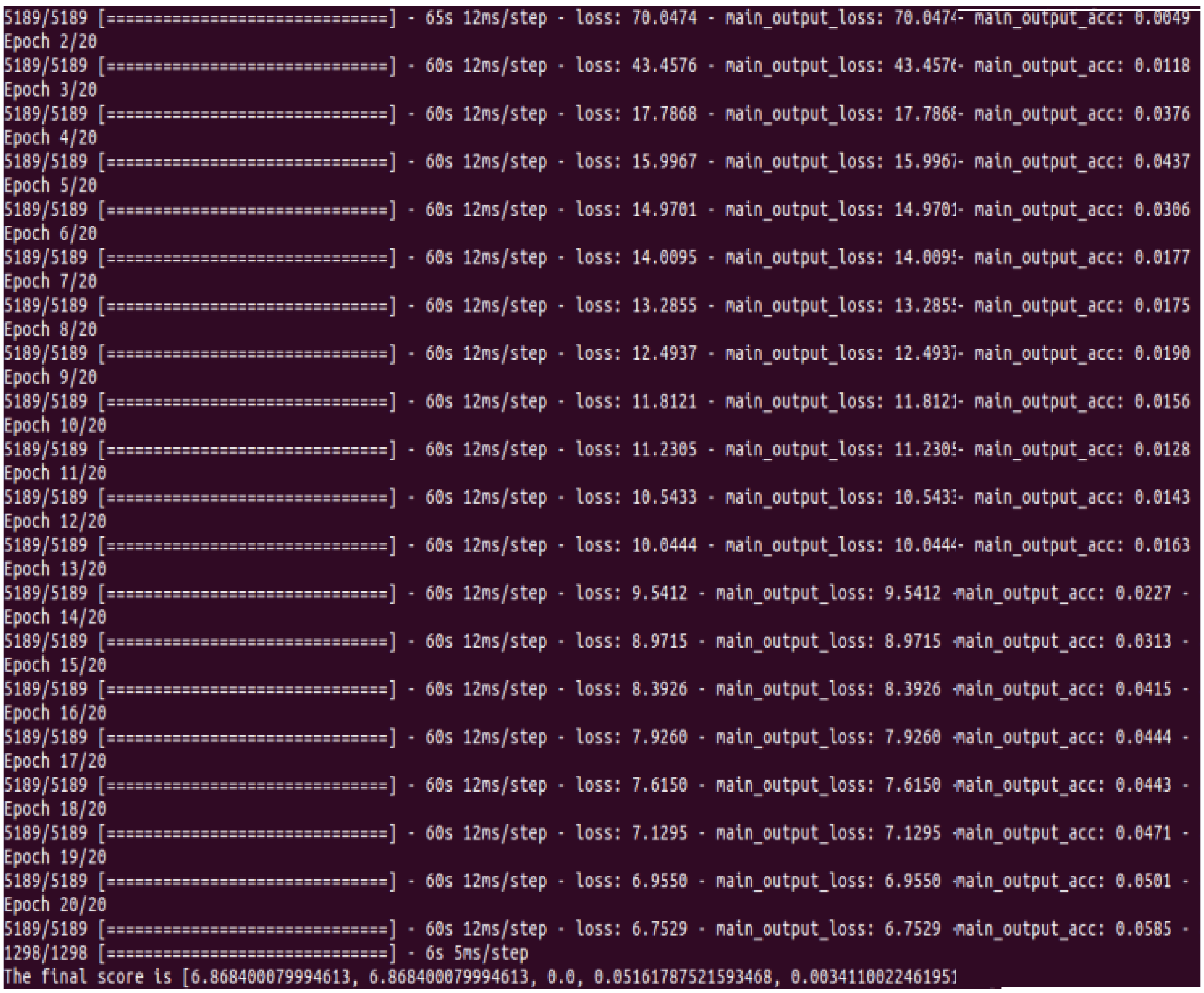
The dataset that we used is not composed of sentences or expressions but of words. The idea was to have easily trainable neural network that could work quickly and would be able to recognize at least some words. Indeed, in the article they say they managed to have a solid working neural network but using several optimisations and with 5000 hours of training. This seemed too much for machine learning rookies. Obviously this makes a neural network weak when you train it on sentences but it works better on words (at least the words where the neural network has been trained on).

2.3 Our results



```
5189/5189 [==============================] - 96s 19ms/step - loss: 25.1165 - main_output_loss: 25.1165 - aux_output_loss: 0.0000e+00 - main_output_acc: 0.0142 - aux_output_acc: 0.0624
1298/1298 [==============================] - 9s 7ms/step
The final score is [24.330353385678425, 24.330353385678425, 0.0, 0.016178736988403396, 0.059951304620238038]
```

Figure 2: Result after a training of 4 epochs



```
5189/5189 [==============================] - 65s 12ms/step - loss: 70.0474 - main_output_loss: 70.0474- main_output_acc: 0.0049
Epoch 2/20
5189/5189 [==============================] - 60s 12ms/step - loss: 43.4576 - main_output_loss: 43.4576- main_output_acc: 0.0118
Epoch 3/20
5189/5189 [==============================] - 60s 12ms/step - loss: 17.7868 - main_output_loss: 17.7868- main_output_acc: 0.0376
Epoch 4/20
5189/5189 [==============================] - 60s 12ms/step - loss: 15.9967 - main_output_loss: 15.9967- main_output_acc: 0.0437
Epoch 5/20
5189/5189 [==============================] - 60s 12ms/step - loss: 14.9701 - main_output_loss: 14.9701- main_output_acc: 0.0306
Epoch 6/20
5189/5189 [==============================] - 60s 12ms/step - loss: 14.0095 - main_output_loss: 14.0095- main_output_acc: 0.0177
Epoch 7/20
5189/5189 [==============================] - 60s 12ms/step - loss: 13.2855 - main_output_loss: 13.2855- main_output_acc: 0.0175
Epoch 8/20
5189/5189 [==============================] - 60s 12ms/step - loss: 12.4937 - main_output_loss: 12.4937- main_output_acc: 0.0190
Epoch 9/20
5189/5189 [==============================] - 60s 12ms/step - loss: 11.8121 - main_output_loss: 11.8121- main_output_acc: 0.0156
Epoch 10/20
5189/5189 [==============================] - 60s 12ms/step - loss: 11.2305 - main_output_loss: 11.2305- main_output_acc: 0.0128
Epoch 11/20
5189/5189 [==============================] - 60s 12ms/step - loss: 10.5433 - main_output_loss: 10.5433- main_output_acc: 0.0143
Epoch 12/20
5189/5189 [==============================] - 60s 12ms/step - loss: 10.0444 - main_output_loss: 10.0444- main_output_acc: 0.0163
Epoch 13/20
5189/5189 [==============================] - 60s 12ms/step - loss: 9.5412 - main_output_loss: 9.5412 -main_output_acc: 0.0227 -
Epoch 14/20
5189/5189 [==============================] - 60s 12ms/step - loss: 8.9715 - main_output_loss: 8.9715 -main_output_acc: 0.0313 -
Epoch 15/20
5189/5189 [==============================] - 60s 12ms/step - loss: 8.3926 - main_output_loss: 8.3926 -main_output_acc: 0.0415 -
Epoch 16/20
5189/5189 [==============================] - 60s 12ms/step - loss: 7.9260 - main_output_loss: 7.9260 -main_output_acc: 0.0444 -
Epoch 17/20
5189/5189 [==============================] - 60s 12ms/step - loss: 7.6150 - main_output_loss: 7.6150 -main_output_acc: 0.0443 -
Epoch 18/20
5189/5189 [==============================] - 60s 12ms/step - loss: 7.1295 - main_output_loss: 7.1295 -main_output_acc: 0.0471 -
Epoch 19/20
5189/5189 [==============================] - 60s 12ms/step - loss: 6.9550 - main_output_loss: 6.9550 -main_output_acc: 0.0501 -
Epoch 20/20
5189/5189 [==============================] - 60s 12ms/step - loss: 6.7529 - main_output_loss: 6.7529 -main_output_acc: 0.0585 -
1298/1298 [==============================] - 6s 5ms/step
The final score is [6.868400079994613, 6.868400079994613, 0.0, 0.05161787521593468, 0.003411002246195
```

Figure 3: Result after a training of 20 epochs

This were 2 examples of a training session of our neural network. On the second one, several intersting things happend, we can notice : our neural network is able to learn efficiency after the 12th epochs, before he's not able to distinguish efficiently. We also have to make a test over more epochs (something like 200), to see a good learning from our neural network. Here, we used 3 words, with more than 5100 files. At the end, our neural network is able to know 5/100 words of people. It's pretty bad for a neural network, but it's only with 20 epochs! Now, we will try to do it with more than 200 epochs.

References and Notes

1. A. Y. Hannun, *et al.*, *CoRR* **abs/1412.5567** (2014).