Part 2: Acceptor Capabilities: try to make it fail

Lior Shimon 341348498
Lev Levin 342480456

We tried to define languages over an alphabet, all of them separable into two classes, that the RNN cannot succeed in learning the class separations, even after many iterations, many examples and many epochs.

In order to make the acceptor to fail in learning, we noticed several features in the RNN model:
Firstly, the model is "strong" for learning tasks linked to the element's position in the sentence (as we saw in the acceptor we implement in part 1).
Furthermore, we know that with enough time, enough computing power, enough epochs and enough training examples the RNN can be seen as equivalent to Turing Machine (and therefore, equivalent to pushdown automaton with two stacks).
Finally, we know RNN learn long-term dependencies features over sequences.

Therefore, we decided to build languages that outcome those insights.
We thought to build languages where sentences are in positive class if a feature of those sentences follows the sequences rule. The key was to define a specific order for sequence so that the RNN would hardly discover it despite his abilities.

Among all the language we thought, the three more challenging cases were:

Case 1:

Description of the language:

the alphabet of the language is composed of three signs: {'a', 'b', 'c'}
there is no limit for the length of the sentences.
The "positive" examples of the language are all the sentences with the number of occurrences of the letter "a" is a prime number.
All other cases are in "negative" class.
If the number of occurrences of "a" is 0, the sentence is in class negative.
If the number of occurrences of "a" is 1, the sentence is in class positive.

Examples

abcbcabcaaba – Number of occurences of "a": 5 – is a prime number – classification: Positive
aaabc – Number of occurences of "a": 3– is a prime number – classification: Positive

bcbcaa – Number of occurences of "a": 2 – is not a prime number – classification: Negative
aaaaaa – Number of occurences of "a": 6 – is not a prime number – classification: Negative

Why did we thought the language will be hard to distinguish?

As we explained at the top of this report, we want to define a specific sequence's order that the RNN would hardly discover.

To find if a given number is prime, our LSTM model needs to learn a complex function, which behaves 'unpredictably' and 'randomly'. It would require too much training data to make it learn. If there is given a list of the previous prime number (training data), it is really hard to find the rule for finding the next one.
We saw at the lecture that LSTM can count. Counting the occurrences of "a" in the sentence is a must, but discovering if this number is a prime by decomposition requires much more information that cannot be discovered from a limited set of examples by counting or any other features of our acceptor described before.
Another more theoretical explanation to it is that, as we said before, RNN is equivalent to Turing Machine only if it has unlimited computational resources. Our LSTM model has limited resources, therefore, despite that Turing Machine can distinguish such a language, LSTM would fail in this task.

LSTM acceptor results:
we run the LSTM acceptor on a training set of 1000 examples with number of occurrence of "a" where the 500th first prime number and 500 non prime number and on a dev set of 300 examples. The accuracy after 5 epochs was still around 44%

Case 2:

Description of the language:
The alphabet of the language is composed of three signs: {'a', 'b', 'c'} there is no limit for the length of the sentences.
The "positive" examples of the language are all the sentences with the number of occurrences of the letter "a" which is in the Fibonacci sequence. All other sentences are in "negative" class. If the number of occurrences of "a" is 0 or 1, the sentence is in the positive class.

Examples
aacbaaabcaaba – Number of occurences of "a":8 – is in fibonacci seq.– class: Positive
aaabc – Number of occurences of "a": 3– is in fibonacci seq.– class: Positive

aabcbcaa – Number of occurences of "a": 4 – is not in fibonacci – classification: Negative
aaaaaa – Number of occurences of "a": 6 – is not in fibonacci – classification: Negative

Why did we thought the language will be hard to distinguish?
This time again and for the same reason, putting aside the formal definition, the Fibonacci sequence is "hard" to predict: given specific number – the number of occurrences of 'a'- predicting if this number is in Fibonacci sequence is difficult, the model will require too many examples to learn the rule of the sequence.

LSTM acceptor results:
This time again we run the LSTM acceptor on a training set of 3000 examples and on a dev set of 800 examples. The accuracy after 7 epochs was still around 54%

Case 3:

Description of the language:

The alphabet of the language is composed of the English alphabet: {'a', 'b', …., 'z'}
there is no limit for the length of the sentences.
The "positive" examples of the language are all sentences that contain a palindrome in it.
A palindrome is a sequence which reads the same backward as forward.
All other sentences are in "negative" class.

Examples

aaaaaaabcbddddddddd – bcb is a palindrome – this sentence is positive
fdedf – all the sequence is a palindrome – this sentence is positive
kayak – all the sentence is a palindrome – this sentence is positive

igotit – is not a palindrome – this sentence is negative
aboringexample – is not a palindrome – this sentence is negative

Why did we thought the language will be hard to distinguish?

This case is different than the two previous one.
Using a Bidirectional-LSTM, the acceptor would have learnt the features related to positive class without too may difficulties since it "reads" the regular order and in the reversed order at the same time, then concatenate the two outputs.
hence, if any output contain any symmetry, there is a palindrome in the sequence.
However, since our acceptor is a simple LSTM, our model has not enough computational power.
Finding this symmetry feature on the overall sentence is maybe theatrically doable, but requires a lot of examples and times.
Here the "rule" of the classification is easy to learn, but is hidden inside a lot of possibilities.
Therefore, given a sentence with fixed length -n- and a lot of example and iterations, our model would eventually find the feature.
However, since the length of each sentences is not fixed, we potentially need an infinite number of examples and iterations.
Our LSTM model is clearly not equivalent to an automate with two stacks,
therefore we thought the language would be hard to distinguish.

We run our model over 5000 examples of positive and 5000 negatives for train, and 1000 on each for the dev set. We reached 58% accuracy after 15 epochs