A Recurrent Approach to Connect 4

Riccardo Zangarini

Abstract

Recurrent Neural Networks are widely used today for various tasks: text classification, text generation, audio or video processing, time-series forecasting and many others.
The main advantage of Recurrent Networks is the ability of the recurrent neurons - fundamental components of the network - to be able to intercept patterns within data sequences with a precise ordering. Indeed, the applications mentioned above are characterised by sequential data.
This project stems precisely from the realisation that most games are made up of sequences of actions performed by players. These actions follow a precise order and, therefore, it is assumed that there is a pattern that determines future actions. This observation gave rise to the idea of applying a Neural Network with a hybrid convolutional-recurrent architecture to games, in particular the popular board game *Connect 4*. After an initial phase of dataset realisation, various experiments were conducted to evaluate the network's performance: the results show a certain degree of learning on the part of the network, despite the severe computational limitations of the available hardware. Possible future developments are proposed at the end of the paper.

1 Task

The nature of the data used to train a RNN is strictly sequential: time-series, texts, audio or video files are some of the most common examples. [1]
Sequential data can also be produced from less conventional sources, such as games: let us think of one of the simplest possible game, *rock, paper, scissors*. This is a game that everyone has played at least once in their life, especially as a child. Each game is independent, and the possibilities for each player are reduced to three moves. Yet even in this simple game, it is possible to get sequences of actions with an identifiable pattern: a player's future move may in fact be determined by his observation of previous games.
A very similar but more complex structure emerges from matches of other popular games, such as *Tic-Tac-Toe*, *Connect 4*, *Chess* and so on.
I decided to take *Connect 4* as an example since it's not very hard to code, the rules are simple and it is very well-known.
The main objective of the project is training a RNN using sequences of moves from simulated games of *Connect 4*, in order to make the network learn winning strategies that outperform, at least, a player that plays random moves.

[1] Robin M. Schmidt: *Recurrent Neural Networks (RNNs): A gentle Introduction and Overview*, 2019

2 Methods

To train the RNN for this task we need to build a dataset from a large number of simulated games of *Connect 4*: it is necessary to implement the game in such a way that these simulations can take place in the shortest possible time, collecting all the necessary data efficiently, but at the same time allowing the user to observe a game or to play it himself.
For all the coding process, I used Python 3.11.3, as long as the most recent versions of the libraries: Numpy, Pandas, Tensorflow and Keras, MatplotLib.
The complete and documented code can be found on my personal GitHub page.

2.1 Players' Creation

The *agents* of the game are derived from the `Player()` class: this is a parent class for all the future *player* objects, providing the essential information required to update the board state, to distinguish the two players and to check who has won the game.
Every sub-class will inherit its features, as well as an additional function that determines the *child* player's next move.
This function is called `move()` and it contains the decision process used by the *player* to make its move.
We have 4 types of players (children classes of the `Player()` class): the `SimplePlayer()` is the main player used to simulate multiple games to train the network with. Its strategy is efficient but very simple: it places its tokens randomly until it finds that either a row, a column or a diagonal is filled up with exactly 3 identical tokens and an empty space. Then, it places the token inside this empty space, giving priority to the winning move.

2.2 Simulation

Using the `simulation()` function, we can simulate n independent games and store them inside a large Pandas' Dataframe. Note that this function does not save games that end in draw. The Dataframe's index corresponds to the game's number: thus, for the first game, all the rows will have the same `index=0`. This will be useful for future data manipulations.
This dataset's structure will be used to **train** the network.

```
[6]  dataset.head()                                                                    Python
```

	player	move	choice	col_0	col_1	col_2	col_3	col_4	col_5	col_6
0	RandomAI_2	0	1	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
0	RandomAI_1	1	4	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]	[0.0, 0.0, 0.0, 0.0, 0.0, -1.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
0	RandomAI_2	2	1	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]	[0.0, 0.0, 0.0, 0.0, 0.0, -1.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 1.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
0	RandomAI_1	3	6	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]	[0.0, 0.0, 0.0, 0.0, -1.0, -1.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 1.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
0	RandomAI_2	4	3	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]	[0.0, 0.0, 0.0, 0.0, -1.0, -1.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 1.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0]	[0.0, 0.0, 0.0, 0.0, 0.0, 1.0]

Figure 1: *Simulation of games between two identical* `RandomPlayer()` *players that, as the name suggests, play random moves.*

2.3 Training Data Structure

Now it's time to get to the core of the project: training a Recurrent Neural Network to learn to play the game with a simulated dataset. We are going to use the dataset produced by two autonomous players

playing against each other.
To make the Network learn a robust winning strategy, we want to train it using only the winning moves: the `df_to_tensor()` function takes care of exactly that.
Briefly, this function manipulates the Dataframe as follows:

1. For each row (player's move) it concatenates the grid's columns along the first axis, obtaining a *grid* feature that is just a 1D array with $height_{board} \times width_{board}$ items.
2. It computes the mean duration (μ) of the games (`int` number) and then it takes only the last μ moves from each game.
   Some games may have a shorter number of moves, therefore this function also expands them by using an auxiliary `padding()` function, which appends the required number of padding rows at the beginning of the sequence. [2]
3. It transforms the dataset into a Tensorflow's tensor, batching it into equally-sized batches, each of them corresponding to the last μ moves of a game.

Finally, the `quickdraw_dataset()` function caches, shuffles and batches the resulting dataset into batches of size 32.

Before proceeding with the RNN training, I would like to emphasise the data structure we have obtained so far.
The batches' shape can be represented as follows:

$$shape = (32, \mu, h \times w) \tag{1}$$

where h and w are, respectively, the height and the width of the grid.

3 Results

Before building and training the RNN, we have to check that the results from the simulations with autonomous hard-coded players are in agreement with our expectations. This way, we make sure that we are going to use non-biased, well-built data to train our model.
As expected, the `SimplePlayer()` outperforms the `RandomPlayer()` (player playing random moves). Furthermore, from the game length distribution, we can notice that the bell-shaped curve from the Simple-Random Simulation is more asymmetric towards the left side and the mean and standard deviation are lower with respect to the Random-Random simulation: in this simulated dataset the games last less moves in average, and the outcomes' consistency is higher. [3]

First Dataset: the first dataset we are using to train the model with is obtained simulating 10^5 games between the `SimplePlayer()` amd the `RandomPlayer()`. I have decided to evaluate the model's performance not on another test set, but in a simulation where one of the two players is the `RNNPlayer()` using the model that we just trained to make the predictions. The results are quite promising. [4]
The RNN definitely learned some sort of strategy: it is able to outperform the `RandomPlayer()` but not the `SimplePlayer()`. If we try to play a game against this trained `RNNPlayer()` we observe that it becomes stronger after a certain number of moves, probably because at the start of the game the data to make the predictions with is not yet as large as the average game duration (μ).
For example, if we play letting the RNN make its own moves without interfering with them (e.g., putting our tokens on the opposite side of the grid), we notice that it follows a pattern that tries to put all the tokens in a very small region, but it does not always complete a line at the first try (so it's worse than the simple strategy).
However, after a couple of wrong moves, it puts the token in the correct place and it also manages to

[2] Padding Row: a row with an empty grid associated to the 0^{th} column as a choice. In this case, we are basically supposing that it's fine to place a token in the first column of an empty grid.
[3] For further details on these results, check the brief analysis in the code.
[4] See the Appendix for the results' details.

interrupt our 3-token line.

Second Dataset: I also tried to train the RNN using simulated games between the `SimplePlayer()` and the `RandomPlayer()` to analyze the results obtained with a more biased dataset. The first difference we notice is the slightly higher accuracy on the validation set.
Here, the results are even better and the Recurrent Network even outperforms the `SimplePlayer()`.

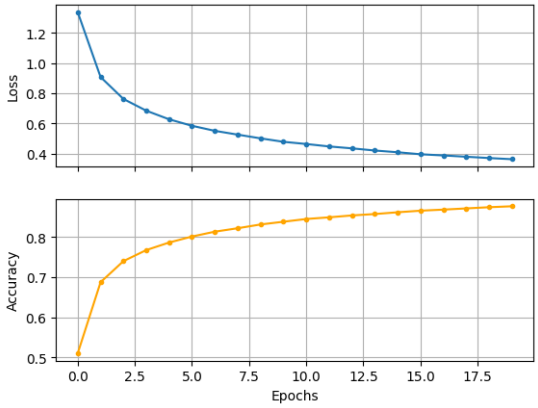

Figure 2: *learning curve of the model on the second training dataset. Total of 20 epochs.*

Model's performance during simulated games:

```
RNN-Random:                          RNN-Simple:
RecurrentAI_1 winrate =  0.932       RecurrentAI_1 winrate = 0.513
RandomAI_2 winrate    =  0.068       SimpleAI_2 winrate    = 0.487
```

The network eventually learned the best way to replicate the `SimplePlayer()` strategy, but looking only at winning moves. This fact probably gives an advantage to the RNN against a player using the simple strategy, since this player can not use the entire sequence of moves to make its decision.
When playing against this RNN, we notice that it gives priority to the interruption of the opponent's lines. Despite being quite effective against both of the other two players, we can conclude that this is not an efficient strategy at all, since it often does not see that it has the opportunity to win.

4 Conclusion

Unfortunately, the available hardware for this project was not excellent, therefore I had to strongly limit the number of training instances, as well as the number of training epochs. At the same time, I could not do many attempts to tweak all the hyperparameters, since they were very time-consuming and very expensive in terms of computational power.
However, the results I have obtained show an interesting potential for this recurrent approach to the game of *Connect 4*, since it was sufficient to give the RNN a dataset to make it learn a strategy that outperformed both the random and simple one.
Despite being quite promising, these results do not justify this kind of Artificial Intelligence in games, especially when compared to an algorithm of *Reinforcement Learning* that is often able to outperform the human player.

Possible Developments: one of the most natural and interesting possible developments would be to incorporate the *Reinforcement Learning* approach, making the network train itself also while playing, and giving it a reward or a penalty (decreasing or increasing the cost function) for good or bad moves. Another possible improvement would be training the network on a much larger dataset (perhaps 10^6 games) in order to make it learn from a wider variety of board states.

5 Appendix

5.1 Model's Architecture

We begin the training of the RNN using the dataset obtained with the simulation of 10^5 games between two `SimplePlayer()` players.
During various experiments, I have tried different architectures, from very simple ones (just one recurrent layer plus the output layer), to very complex sequential models, stacking multiple recurrent layers with convolutional ones.
After comparing the performances, I concluded that, with the dataset's dimension being not very large (10^5 total games), the best model to use has the architecture shown below.
For the training process, I used **Stochastic Gradient Descent**, with a learning rate $\eta = 0.05$ and `clipnorm=1` (the latter gives the opportunity to use a larger η).
The loss function I used is the **Sparse Categorical Cross-Entropy** that, based on the Cross-Entropy equation, produces a category index of the most likely matching category. Therefore, the output will be an array containing the probabilities associated to each class.
I also implemented *Early Stopping* for regularization and computation time efficiency.

This model's architecture gave good results in terms of learning accuracy and loss function:

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d_6 (Conv1D)           (None, None, 32)          6752

 batch_normalization_6 (Batc  (None, None, 32)         128
 hNormalization)

 conv1d_7 (Conv1D)           (None, None, 64)          6208

 batch_normalization_7 (Batc  (None, None, 64)         256
 hNormalization)

 lstm_3 (LSTM)               (None, 64)                33024

 dense_3 (Dense)             (None, 7)                 455

=================================================================
Total params: 46,823
Trainable params: 46,631
Non-trainable params: 192
_________________________________________________________________
```

Brief description of the model's layers:

1. `tf.keras.layers.Conv1D`: it is a common 1D convolutional layer, that transforms the original 1D input into a collection of feature maps, whose weights will be updated during training. Its main purpose is to make each neuron learn from a small portion of the input (in the first layer) and then expand this field of view in order to detect various patterns using the feature maps.
   It uses the *ReLU* activation function.
2. `tf.keras.layers.BatchNormalization`: used to make the training process more stable thanks to the normalization of the layers' inputs by re-centering and re-scaling. It also acts as a regularizer.

3. `tf.keras.layers.LSTM`: this is the actual recurrent layer. It stands for "Long-Short Term Memory" and it is widely used in any recurrent network's architecture.
4. `tf.keras.layers.Dense`: the output layer, using the *Softmax* as activation function on 7 different classes (number of columns).

5.2 Results Data

First Dataset

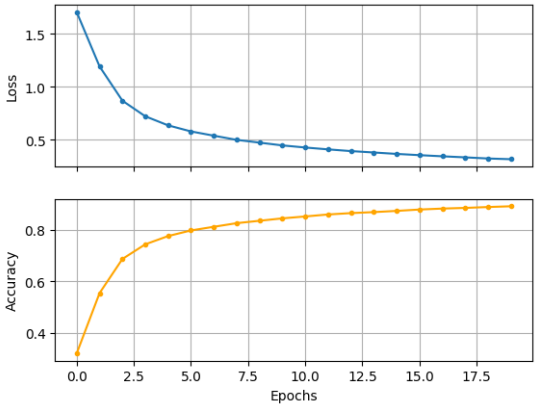

Figure 3: *learning curve of the model on the first training dataset. Total of 20 epochs.*

```
RNN-Random:
RecurrentAI_1 winrate =  0.741
RandomAI_2 winrate    =  0.259
```

```
RNN_Simple:
RecurrentAI_1 winrate =  0.399
SimpleAI_2 winrate    =  0.601
```