Department of Computer Science
Technical University of Cluj-Napoca

Intelligent Systems

Laboratory activity 2019-2020

Project title: Human action mimicking
Tool: Tensorflow RNN

Name: Kelemen Máté
Group: 30432
Email: mate.kelemen009@gmail.com

Contents

Chapter 1

Overview

A recurrent neural network (RNN) is a class of artificial network derived from regular feed-forward neural networks. The main idea behind RNNs is to make use of sequential information. In a traditional neural network we assume that all inputs (and outputs) are independent of each other. But for many tasks that's a very bad idea. Eg. if we wanted to predict the next word in a sentence, we better know which words came before it. RNNs are called recurrent because they perform the same task for every element of a sequence, with the output being depended on the previous computations.
Another way to think about RNNs is that they have a "memory" which captures information about what has been calculated so far.

Chapter 2

Main functionalities

The main functionality of an RNN lies in doing exceptionally well at tasks that require the capturing of long-term dependencies. RNNs have shown great success in many natural language processing tasks, but not only. Some examples include:

1. Language modeling and text generation (Given a sequence of words we want to predict the probabilty of each word given the previous words)
2. Machine translation (translating a sequence of words in a target language eg. German to English)
3. Speech recognition (Given an input sequence of acoustic signals from a sound wave, we can predict a sequence of phonetic segments together with their probabilities)
4. Time series prediction
5. Rhythm learning
6. Music composition
7. Grammar learning
8. Handwriting recognition
9. Human action recognition
10. Protein Homology Detection
11. Predicting subcellular localization of proteins
12. Several prediction tasks in the area of business process management
13. Prediction in medical care pathways

Chapter 3

Detailed descriptions of some components

3.1 LSTM

The problem with standard RNNs is that although they can keep track of arbitrarily long-term dependencies in the input sequence, when trained using backpropagation, the gradients which are backpropagated can vanish or explode. The reason behind this is the computational nature of the process, which use finite-precision numbers. RNNs using LSTM cells can partially solve the vanishing gradient problem, because LSTMs allow gradients to also flow unchanged.
LSTM (long-short term memory) units are at the core of most RNNs. Unline standard neural networks, LSTMs have feedback connections. A common LSTM cell is composed of a cell, an input gate, an output gate, and a forget gate. The unit is responsible for keeping track of the dependencies between the elements in the input sequence. The input gate controls the extent to which a new value flows into the cell, the forget gate controls the extent to which a value remains in the cell and the output gate controls the extent to which the value in the cell is used to compute the output activation of the LSTM unit. Figure 3.1 show the structure of an LSTM.

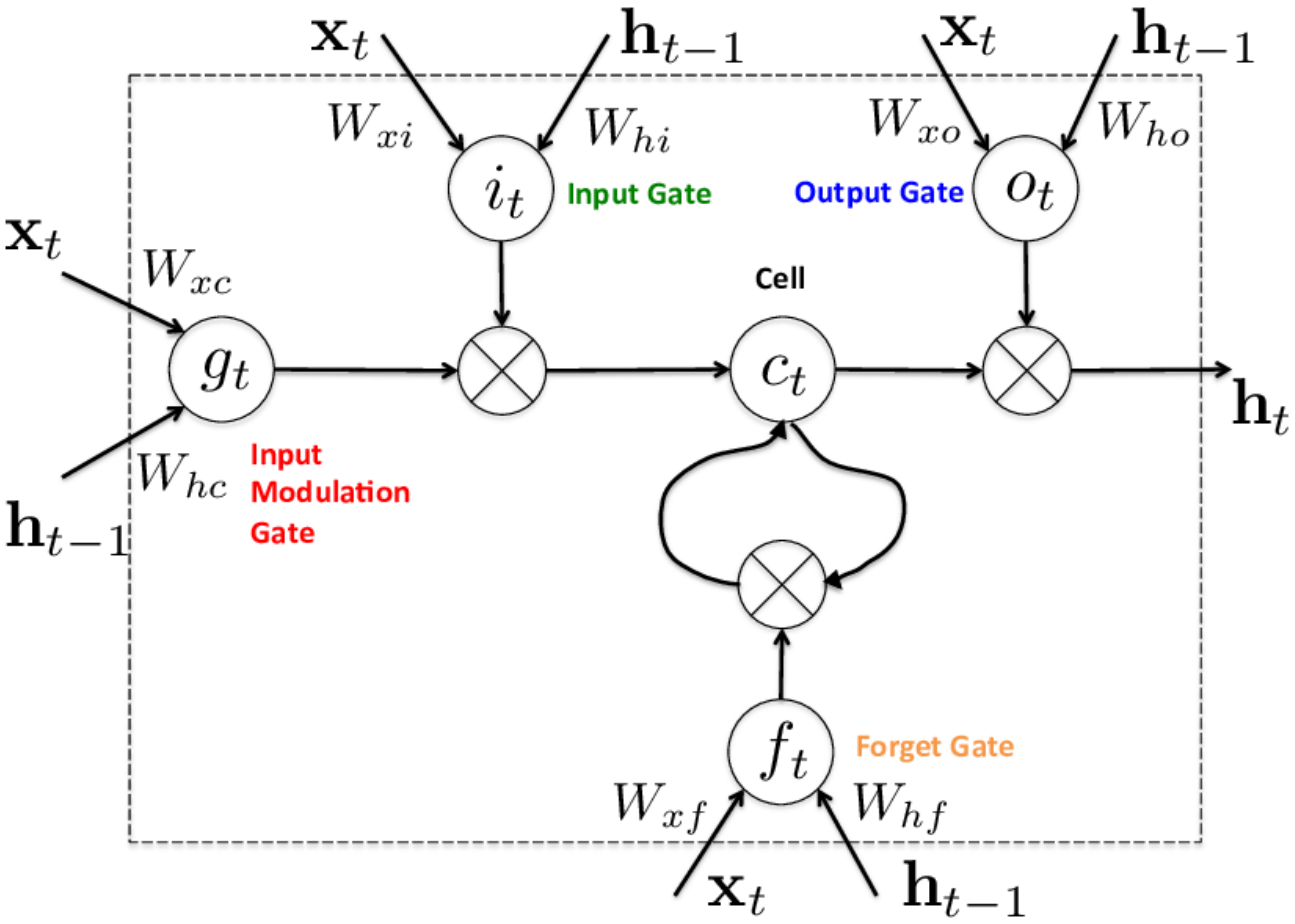

Figure 3.1: The structure of an LSTM

3.2 Backpropagation through time

Backpropagation Through Time, or BPTT, is the application of the Backpropagation training algorithm to recurrent neural network applied to sequence data like a time series.
A recurrent neural network is shown one input each timestep and predicts one output.
Conceptually, BPTT works by unrolling all input timesteps 3.2. Each timestep has one input timestep, one copy of the network, and one output. Errors are then calculated and accumulated for each timestep. The network is rolled back up and the weights are updated.
Spatially, each timestep of the unrolled recurrent neural network may be seen as an additional layer given the order dependence of the problem and the internal state from the previous timestep is taken as an input on the subsequent timestep.
We can summarize the algorithm as follows:

1. Present a sequence of timesteps of input and output pairs to the network.
2. Unroll the network then calculate and accumulate errors across each timestep.
3. Roll-up the network and update weights.
4. Repeat.

BPTT can be computationally expensive as the number of timesteps increases.

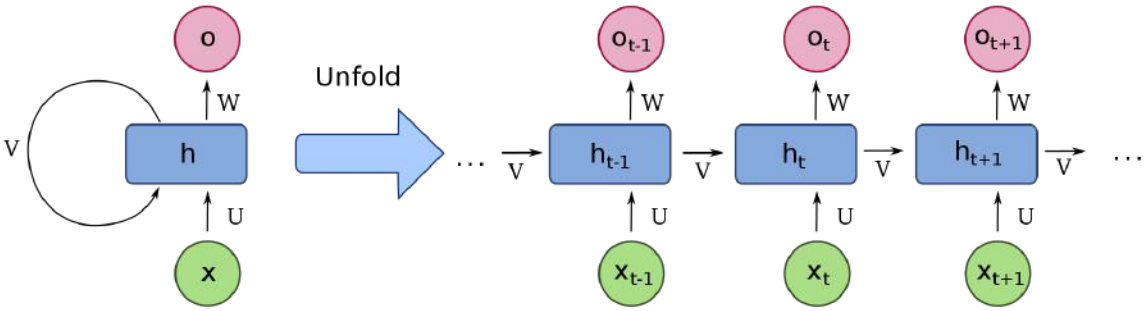

Figure 3.2: Unrolling an RNN

Chapter 4

Examples

4.1 Generating image descriptions

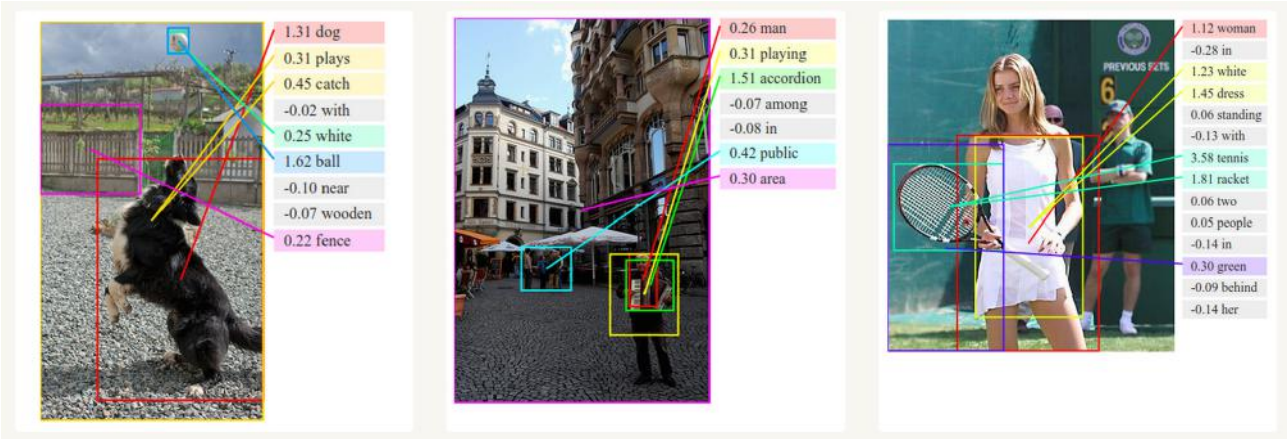

Figure 4.1: Deep Visual-Semantic Alignments for Generating Image Descriptions - Stanford 5.8

4.2 MariFlow

Figure 4.2: Self-Driving Mario Kart which copies its developer's play style 5.8

Chapter 5

Proposed problem

The main theme of this problem is human action copying. The goal of the project is to develop an AI using LSTM RNNs that is able to play a computer game in real-time by itself, and its play style to be indistinguishable from that of a human's. The goal of the AI is not necessarily to win the game, but to play like a human, knowing nothing about the details of the objects that it sees, but creating the connections between a series of input images and the courses of action, that a human takes, while playing that game.

5.1 General specifications

The AI will receive a fixed amount of downsized grayscale images as input during every second. Other inputs could include game state flags (eg. whether the game is finished). The outputs will be the key presses on a simulated controller (eg. keyboard, joystick).
Training the model will be done in the following way:
A human (user) has to record a long video footage of them playing the game, and feed this data into the model. After this, hypothetically, the AI should be able to mimick the human's play style with a high accuracy, when it is put to the test.
Sometimes the AI can get stuck on certain game states. The reason behind this, is the possibility of the AI encountering a state that is far from anything it has encountered so far. To fix this, the functionality to swap the human player with the AI in real-time will be added, so that the player can get the AI out of its bad situation and provide extra training data.

5.2 Design

The selected game is Mario Kart(SNES). The training data consists of a stream of 20x20 grayscale images, containing only a part of the full emulator screen. The focus was to record only the most important part of the game, the road. For this purpose, only the road part of the emulator screen was recorded, omitting some of the gameplay details. The second part of the training data are the button presses corresponding to their image. There are ten buttons on a SNES controller, and each of the buttons are either pushed down or released, meaning that the button data can easily be stored in a binary array. This is exactly how the button data was encoded. Thus, the complete training data is a stream of images together with their corresponding button presses represented as binary arrays.

The process for recording the data was the following:

1. The emulator was opened and Mario Kart was started
2. Only the main events of interest were recorded (the main race)

3. The user played one runthrough of the 50cc Mushroom Cup

The purpose of the AI is to predict the button presses that the player would make, given a series of images (input states of the game that led to the current state), thus the justification for LSTMs. A loader class takes care of putting the training data into a format that the LSTM can accept as input.

The input of an LSTM is an array of images ordered by time. The prediction is based on time, and this is how it can predict specific actions for a given user: It examines past events in steps until a given number of steps is reached. This number of steps is called the timestep. The current training data is arranged in timesteps of 5, although there is a possibility to vary this parameter. At runtime, not only the current frame, but a queue of images is passed as input.

As mentioned before, the predicted keypresses of the model will be the inputs for the emulator. Because of the expected erronious button-press-predictions of the model, the user may override the controls at any moment. The realtime button presses and the field of vision of the AI will be shown at runtime.

5.3 Model

5.3.1 Summary

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 5, 200)            480800
_________________________________________________________________
dropout (Dropout)            (None, 5, 200)            0
_________________________________________________________________
lstm_1 (LSTM)                (None, 200)               320800
_________________________________________________________________
dropout_1 (Dropout)          (None, 200)               0
_________________________________________________________________
dense (Dense)                (None, 200)               40200
_________________________________________________________________
dropout_2 (Dropout)          (None, 200)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                2010
=================================================================
Total params: 843,810
Trainable params: 843,810
Non-trainable params: 0
_________________________________________________________________
None
```

5.3.2 Details

The activation function for the output layer is the sigmoid function, so the output represents the likelihood of the corresponding button-press array being a correct one. Dropout layers were added to ensure that no overfitting occurs.

5.4 Results

5.4.1 Loss

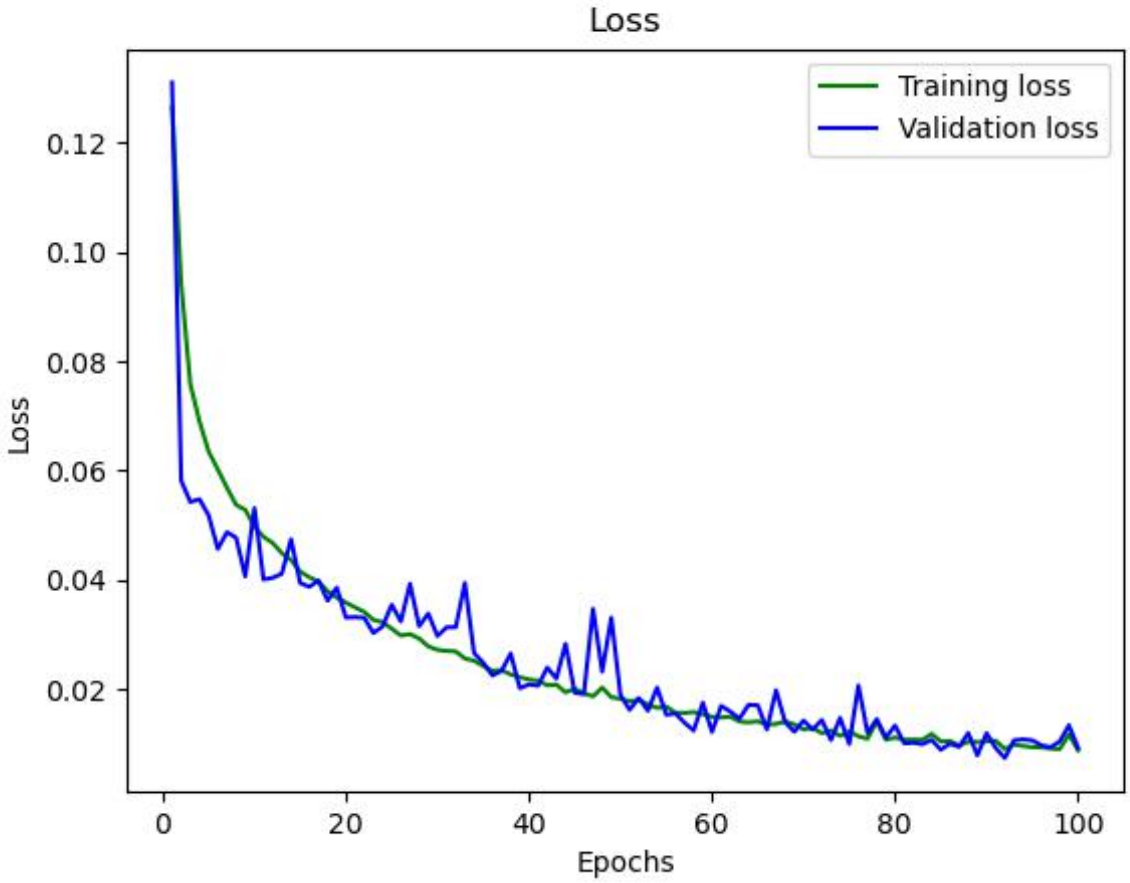

Figure 5.1: Loss (Validation set of part of the training set)

5.4.2 Accuracy

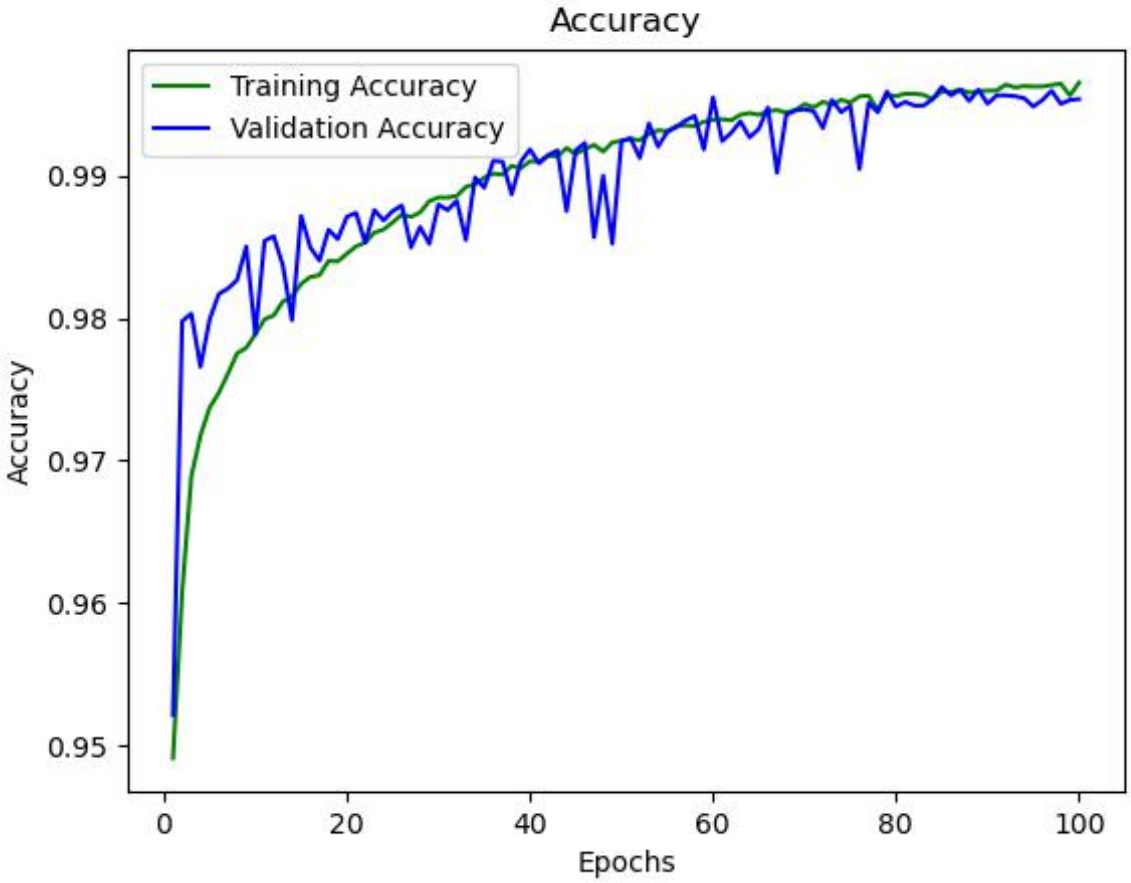

Figure 5.2: Accuracy (Validation set of part of the training set)

5.4.3 Observations

The most important observation when giving control to the AI at runtime is that when the race is started, it could drive around for a little bit, but then it encountered a state that it has never seen before and get stuck. This is most likely because of the training data. The AI performed well in situations, where there were no other racers around it. That is also because of the quality of the training data (In the training data the player was at the first place every time, and it was far from other racers).

5.5 Running the Software

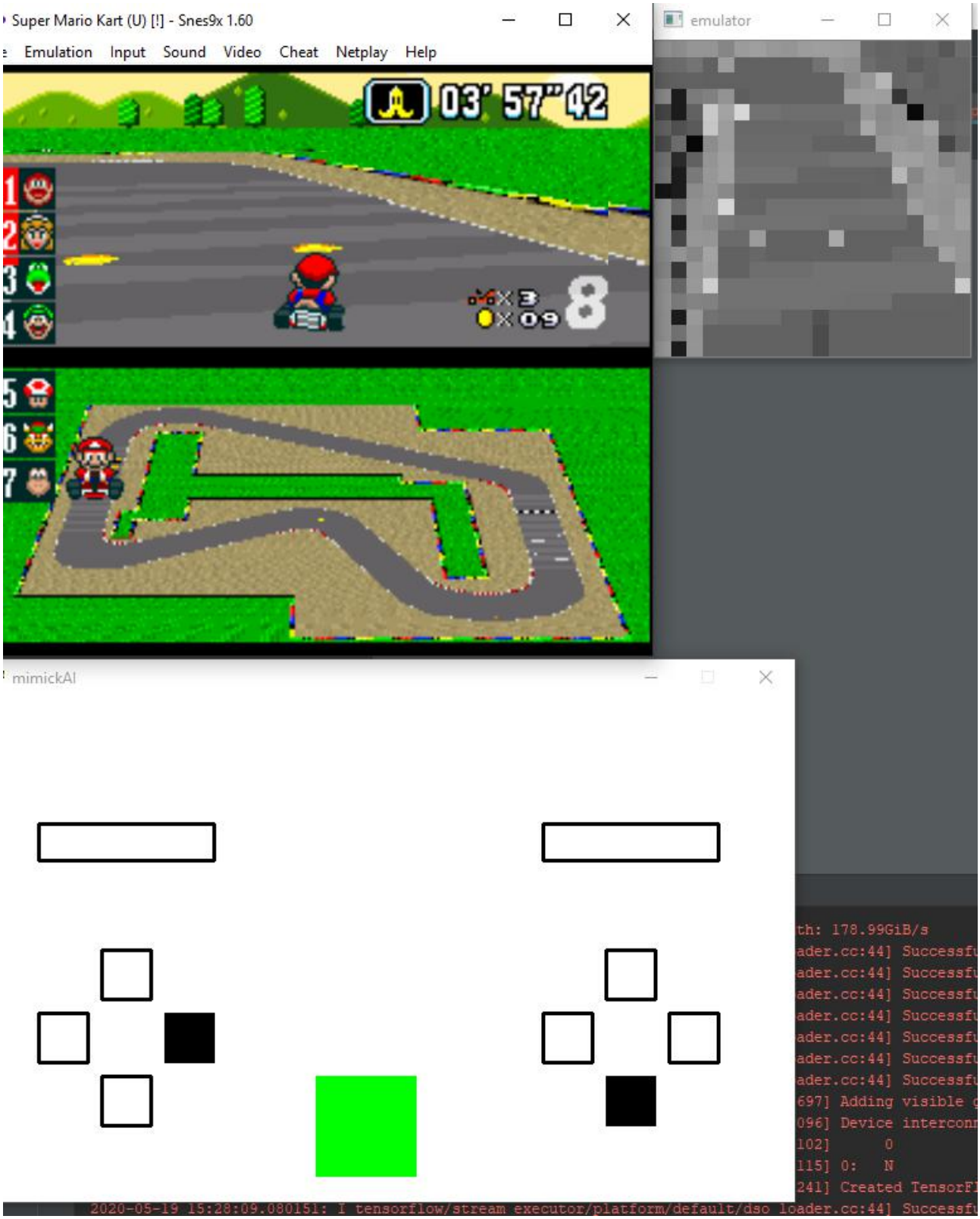

Figure 5.3: Emulator (upper left), AI's vision (upper right), button press predictions (bottom)

To run the software:

1. Open snes9x (emulator), start mario kart

2. Resize the snes9x window to be a square, and place in on the upper left corner of the screen
3. Execute runner.py
4. Click on the snes9x window to make it active
5. Press 'P' to alternate between user and AI (Square at the middle will be red if the user controls,)

5.6 Further development

The first major improvement should be providing better training data (eg. where there are more racers around). The hyperparameters of the model could also use some tuning. Furthermore, the training data could be downsampled. The reason for this is, the training data has been recorded at 50-60fps (Frames per second), but when predicting the keypresses in realtime, the framerate drops to 10-20 fps. This means that some information never makes sense because of how sudden the changes are.

5.7 Related work

MariFlow 5.8

5.8 Source of Data

MariFlow - `https://www.youtube.com/watch?v=Ipi40cb_RsI`
Generating image descriptions - `https://cs.stanford.edu/people/karpathy/deepimagesent/`
LSTM - `https://en.wikipedia.org/wiki/Long_short-term_memory`
BPTT - `https://en.wikipedia.org/wiki/Backpropagation_through_time`
Introduction to BPTT - `https://machinelearningmastery.com/gentle-introduction-backpropagati`
Tensorflow RNN - `https://www.educba.com/tensorflow-rnn/`
`http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial\-part-1-introduction-t`