TRABAJO FIN DE GRADO

INGENIERÍA INFORMÁTICA

Aprendiendo a Seleccionar Objetivos con Deep Q-Learning en GVGAI

Anexo - Documentación Técnica y Tutoriales

Autor
Carlos Núñez Molina

Director
Juan Fernández Olivares

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, julio de 2020

Contents

This document provides a description of the code structure as well as several use tutorials, which will allow anyone interested in the code to install and execute it.

1 Code Structure

1.1 GitHub Repository and Branches

The entirety of the code has been made available in the following **GitHub repository**. This repository contains a large number of branches, each one representing a different *version* of the code. This way, it has been easy to implement and test each version separately, without having to worry about the interactions with the other versions. For example, the Greedy Model and DQP Model correspond to different versions and, thus, are kept on separate branches. The different branches will now be explained.

modelo-greedy

This branch was used to implement and test the Greedy Model. Firstly, we did some testing using a Jupyter Notebook and, then, integrated the Greedy Model into the agent architecture. In this version of the code, the agent performs *online training*, i.e., the data needed to train the model is collected in real-time during the training phase.

modelo-DeepQLearning

The same as the *modelo-greedy* branch but with the DQP Model instead of the Greedy Model.

fast-Greedy

This branch significantly speeds up the process of training and testing different architectures of the Greedy Model. Firstly, it performs *offline learning*. This means that, instead of populating the dataset as the model is trained, it trains the model on an already-created, static dataset. This way, the dataset only needs to be populated once and the training is quicker. Secondly, we added auxiliary scripts to automatize the process of training and testing the Greedy Model: *ejecutar_pruebas.py*, *ejecutar_test_modelos.py* and *test_output_to_csv.py*.

fast-DQP

The same as the *fast-Greedy* branch but with the DQP Model instead of the Greedy Model.

tests-random

This branch implements the Random Model. This model selects subgoals in a random way and has been used as a baseline to compare the other two models against.

planningtimes-Greedy

This branch is used to calculate how long, on average, it takes for the planner of the Greedy Model to find a plan to a given subgoal. For that task, the *medir_tiempos_test.py* script is used.

planningtimes-DQP

The same as the *planningtimes-Greedy* branch but for the DQP Model.

Sokoban-base

In this branch we started to test the architecture on another GVGAI game known as *Sokoban*. However, it is unfinished and outside the scope of this project.

IceAndFire-base

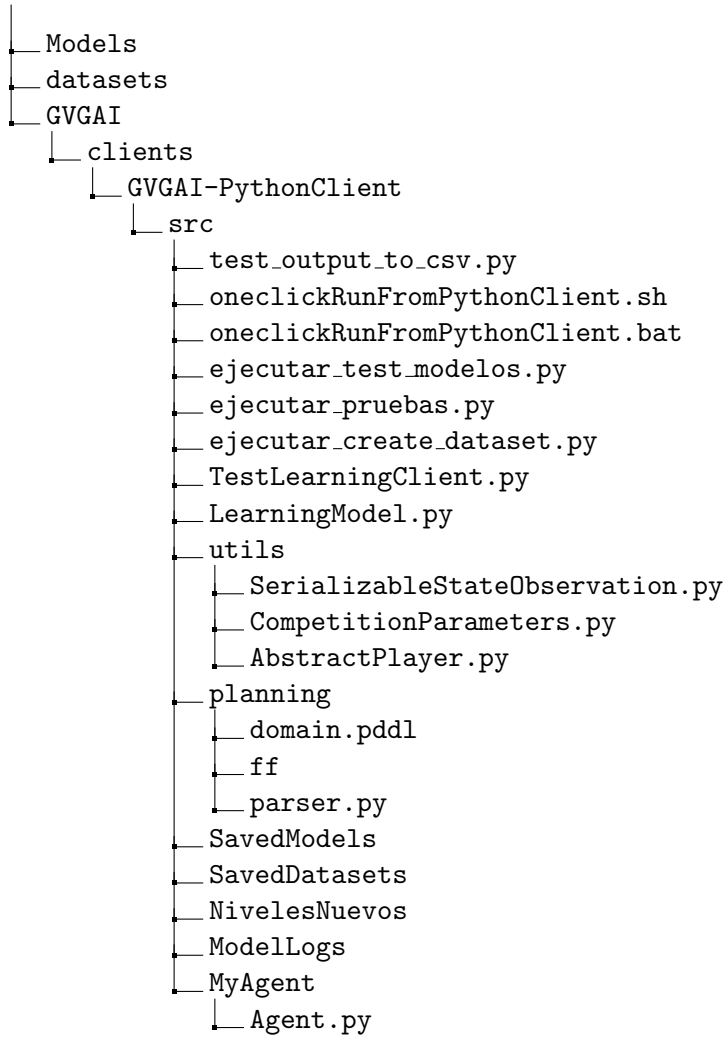
The same as the *Sokoban-base* branch but for the GVGAI game called *Ice and Fire*.

master

At the beginning, we started implementing all the code into a single branch (*master*). However, as development continued, we realized it was better to use several branches instead. For this reason, this branch was *left behind* in favour of the others.

1.2 Directory Structure and Files

Here we present the most important files for the development of this project and where they are located within the directory tree. Were this not enough, please refer to the official **GVGAI website** and **GitHub repository** for the GVGAI Learning Track. Have in mind that some files (like the ones of GVGAI) are present in all branches whereas some others are only in certain branches. Below is shown an overview of the directory tree of both the *fast-Greedy* and *fast-DQP* branches:



The *Models* directory contains the Jupyter Notebooks we used to test the Greedy Model before integrating it into the agent architecture. For that task, we used the datasets in the *datasets* directory.

The *GVGAI* directory contains all the code needed to execute the GVGAI Learning Track. It also contains the code of our agent and the auxiliary scripts we have implemented. GVGAI uses a client-server architecture: the (local) server executes all the logic of the game whereas the agent acts as a client which returns to the server the action to execute in each game state.

The *oneclickRunFromPythonClient.sh* and *oneclickRunFromPython Client.bat* scripts are used to execute the agent. The former is used in Linux and the latter, in Windows. These scripts call the *TestLearningClient.py* script which initalizes the server and connects the agent to it. The most important parameters of this call are:

- *gameId*: the game to play (11 for Boulder Dash).
- *agentName*: the name of the Python script implementing the agent, in the form of *directory.name*.
- *visuals*: if used, the game shows a window which lets the user see how the agent plays the game.

The *Agent.py* script implements the agent which plays the game. It uses the functionality provided by the *LearningModel.py* script and the *planning* folder.

The *LearningModel.py* script implements the TensorFlow code used to learn to select subgoals by the agent. This code is different for the DQP and the Greedy Model. The *SavedDatasets* directory contains the datasets used to train the learning model and *SavedModels* is the directory where all the trained models are saved. *ModelLogs* (called *DQNetworkLogs* in the *fast-DQP* branch) contains the training logs of the models in *SavedModels*, which can be visualized using TensorBoard.

Once a subgoal has been selected, a planner is used to find a suitable plan to achieve it. The implementation of the planner, as well as its integration into the agent architecture (the *parse_game_state* and *search_plan* methods of the *Agent* class), has been made by the student Vladislav Nikolav Vasilev. The *planning* folder contains all the code needed to perform the planning. The *domain.pddl* file contains the PDDL description of the planning domain of BoulderDash. The *parser.py* script is used to encode (parse) the inputs into a suitable representation for the planner. To finish, *ff* is the executable file of the Fast-Forward Planner, which is used to obtain every plan. The plans obtained are not always optimum in length, although they are usually close to it.

Inside the *NivelesNuevos* folder there are 10 new BoulderDash levels, different from the ones used for training and validation. These levels where used to test the final architecture of both the Greedy and DQP Model and, this way, obtain the final results which are shown in the paper. These levels were created using the VGDL language which GVGAI provides to create new games and levels.

The *utils* folder contains, as its name suggests, several utilities. *AbstractPlayer.py* contains the class which all agents must inherit from. *SerializableStateObservation.py* implements the code used to encode a game state and shows all the information which can be retrieved from it by the agent. In the *CompetitionParameters.py* script several execution parameters are set. The most important one is *TOTAL_LEARNING_TIME*, which is how long the training phase will last. After this time, the training phase ends and the validation phase begins.

To conclude, there are the auxiliary scripts, which are used to automatize the creation of datasets, training, validation and test process. *ejecu-*

tar_create_datasets.py is used to automatically generate new training datasets, which are saved in the *SavedDatasets* directory. *ejecutar_pruebas.py* is used to repeat the training and validation of a given DQP or Greedy architecture a certain number of times. Repetition reduces the variance of the results obtained so it is needed in order to obtain robust results. *ejecutar_test_modelos.py* is used to test a series of already-trained models on the 10 new levels used for testing. These scripts use regular expressions to change the *Agent.py* and *CompetitionParameters.py* scripts and, then, execute *oneclickRunFromPythonClient.sh*. We have chosen this way of automation because, since all the code is in the form of Python scripts, it is really easy to change them using regular expresions and call them from other scripts. Also, the GVGAI documentation is a little confusing and we had issues trying to find, for example, how to increase the number of validation levels from 2 to 10. *test_output_to_csv.py* reads *test_output.txt* (file which contains the performance obtained by the models on the validation or test levels) and encodes it in CSV.

1.3 Class Structure and Methods

As explained before, we have split our code across several Python scripts. The most important ones are *Agent.py*, which implements the logic of the agent, and *LearningModel.py*, which implements the TensorFlow code the agent uses to learn to select subgoals. Here we will describe those scripts for both the *fast-DQP* and *fast-Greedy* branches.

Agent.py defines the *Agent* class, which inherits from the *AbstractPlayer* class defined in *AbstractPlayer.py*. All agents must inherit from *AbstractPlayer* to be able to connect to the GVGAI server. The main methods of this class are the following:

__init__

Constructor. It is called before any other method.

What it does depends on the *execution mode* of the agent. There are three different ones: *create_dataset*, *train* and *test*. In *create_dataset*, the agent simply plays the training levels, selecting subgoals randomly and using the collected experience to populate the training datasets. In *train*, the agent loads the datasets and trains a learning model on them, which learns to select subgoals. In *test*, the agent loads an already-trained model and uses it to play the validation or test levels, measuring how many actions it uses to beat each level.

init

Similar to *__init__*, this method is called at the start of every level.

act

The most important method of the *Agent* class. It is called at each game state and it must return the action the agent will execute.

If the *execution mode* is *create_dataset*, the agent plays the training levels (levels 0-2). Every time it needs to select a subgoal (gem), it chooses it randomly. While the agent plays the game, it saves the collected experience and, when the number of samples is big enough, it saves the dataset to disk and the execution ends.

If the *execution mode* is *train*, the agent performs the training of the learning model (which can be either the DQP Model or the Greedy Model). It loads datasets of different sizes and, for each dataset, it trains an independent model (but with the same network architecture). Once a model has been trained, it is saved to disk. After all the models have been trained, execution ends. All this process happens at the very first turn of the agent. This is why, in this *execution mode*, the agent doesn't play the game.

If the *execution mode* is *test*, the agent plays the validation/test levels (levels 3 and 4). Every time it needs to select a subgoal, it uses the loaded, already-trained model to select the best one. In order to test a model on the 10 test levels instead of just 2, the *ejecutar_test_modelos.py* script is used, which repeats the execution of the agent using a different pair of test levels each time.

get_gems_positions

Given a game state, it returns the positions of the gems which the agent have not picked yet.

encode_game_state

It transforms the encoding of a game state from a matrix of observations (instance of the *SerializableStateObservation* class) to a *one-hot matrix*, i.e., the state representation the learning model works with.

choose_next_subgoal

Given a game state, the positions of the available gems and the position of the agent, it uses the learning model to choose the best subgoal (gem).

parse_game_state

It encodes (parses) a game state and subgoal as a suitable representation for the planner.

search_plan

Given the current game state and a selected subgoal, it uses the planner to find a suitable plan that achieves it.

save_dataset

It uses the *pickle* module to save the data collected by the agent (experience replay) to disk as a new dataset.

load_dataset

It uses the *pickle* module to load a dataset with a given number of samples to be used for training the learning model.

result

This method is called when the current level of the game ends. It must return which level to play next, which we have chosen to be simply the next one. If the *execution mode* is *test*, it outputs to the *test_output.txt* file the number of actions used to complete each validation/test level. Once done, execution ends.

LearningModel.py defines the *CNN* class in the *fast-Greedy* branch and the *DQNetwork* class in the *fast-DQP* branch. The main methods of both classes are described above.

__init__

Constructor. It initializes the learning model. It defines the network architecture and hyperparameters and also initializes the parameters, log writer and TensorFlow session.

close_session

It closes the current TensorFlow session, freeing the resources.

predict

Given an input x (a selected subgoal and a game state), it predicts the length y of the associated plan.

train

It performs a certain number of training iterations on the given (X, Y) batch of training samples.

save_logs

It calculates the training loss on the given (X, Y) batch and saves it to disk to be later visualized using TensorBoard.

save_model

It saves the trained model to disk.

load_model

It loads an already-trained model from disk.

2 Tutorials

This section provides several tutorials which will teach the user how to use the above-mentioned functionalities.

2.1 Basic Tutorial

This tutorial will show how to install the GVGAI framework and use it to execute a test agent which plays the BoulderDash game.

Prerequisites

The code of this project has been developed using the following tools:

- **Java - OpenJDK 11.0.6**
- **Anaconda - 4.7.12**. In the *master* branch of the project repository, within the parent directory, there is a file called *conda_env.yml*. This file contains the conda environment used for this project.
- **Git**. In order to upload the training datasets (*SavedDatasets* folder), which are too big for normal git, **git lfs** is needed. Simply associate the *.dat* format to git lfs with the following command: `git lfs track "*.dat"`.
- **Ubuntu - 18.04.3 LTS x86_64**

Clone the repository

Clone the **GitHub repository** of this project and switch to the *master* branch, where the conda environment file is located.

Import the conda environment

Use the *conda_env.yml* file to create a new conda environment using the following command: `conda env create -f conda\_env.yml`. This will create the conda environment and install all the dependencies, including Python 3.5.6. Activate the environment with `conda activate PlanningWithSubgoals`. Then, if you execute `conda list`, you should be able to see Python and all the packages installed, such as TensorFlow.

Create a new agent

In the `GVGAI/clients/GVGAI-PythonClient/src` directory, create a new directory called *TestAgent*. Inside it, create a Python script called *Agent.py* with the following code:

```
from AbstractPlayer import AbstractPlayer

from Types import *
from utils.Types import LEARNING_SSO_TYPE

class Agent(AbstractPlayer):

    def __init__(self):
        AbstractPlayer.__init__(self)
        self.lastSsoType = LEARNING_SSO_TYPE.JSON

    def init(self, sso, elapsedTimer):
        pass

    def act(self, sso, elapsedTimer):
        return 'ACTION_UP'

```

This simple agent will simply execute *ACTION UP* in each turn, which means it will go up until it hits a wall.

Change CompetitionParameters.py

Inside the *utils* directory, open the *CompetitionParameters.py* file. Set the class attribute named `TOTAL_LEARNING_TIME` equal to `3*MILLIS_IN_MIN`. This way, the agent will play the training levels for 3 minutes, before entering the validation phase.

Change oneclickRunFromPythonClient.sh

Edit the *oneclickRunFromPythonClient.sh* as follows:

```
#!/bin/bash

game_id=11
server_dir=../../..
```

```
agent_name=TestAgent.Agent
sh_dir=utils


DIRECTORY='./logs'
if [ ! -d "$DIRECTORY" ]; then
    mkdir ${DIRECTORY}
fi

# Run the client with visualisation on
python TestLearningClient.py -gameId ${game_id} -agentName ${
  agent_name} -serverDir ${server_dir} -shDir ${sh_dir} -
  visuals

```

Execute the agent

Still with the conda environment activated, execute `bash oneclickRunFromPythonClient.sh`. The agent will connect to the server and a window showing the agent playing the first level of BoulderDash should open, as shown in the figure below.

Figure 1: The agent playing BoulderDash.

2.2 Creating Datasets

In this tutorial we will explain how to create new datasets to train the model on. The experience used to populate the datasets is collected by the agent while playing the three training levels GVGAI provides. *fast-DQP* is the branch we will be using for this tutorial. However, the process is identical for the *fast-Greedy* branch.

Switch Branches and Activate Conda Environment

Switch to the *fast-DQP* branch by executing `git checkout fast-DQP` on the terminal. Then, activate the conda environment using `conda activate PlanningWithSubgoals`.

Edit ejecutar_create_dataset.py

Move to the *src* directory and open *ejecutar_create_dataset.py*. `id_dataset_ini` and `id_dataset_fin` are the indexes of the first and last dataset to generate. Let's set `id_dataset_ini=11` and `id_dataset_fin=12` to generate two additional datasets to the 10 already created.

Edit Agent.py

In *Agent.py*, set `self.EXECUTION_MODE="create_dataset"`.

Edit CompetitionParameters.py

In *CompetitionParameters.py*, set `TOTAL_LEARNING_TIME=100*60*MILLIS_IN_MIN`.

Edit oneclickRunFromPythonClient.py

If the *visuals* parameter is set, it is advised to remove it, since the execution will be faster without any GUI. However, if you want to observe the agent play the game, you can use this parameter.

Start the Execution

On a terminal, execute `python ejecutar_create_dataset.py`. This will create all the datasets given in *ejecutar_create_dataset.py*, each one containing 1000 samples. This number can be changed in *Agent.py*, using the attribute `self.num_samples_for_saving_dataset`.

After finishing the execution, open the *SavedDatasets* directory. You should be able to see two new datasets: *dataset_1000_11.dat* and *dataset_1000_12.dat*.

2.3 Training and Validating a Model

In this tutorial we'll cover how to train a given model architecture and, after training, validate it on the two validation levels GVGAI provides. *fast-DQP* is the branch we will be using for this tutorial. However, the process is identical for the *fast-Greedy* branch.

Switch Branches and Activate Conda Environment

Switch to the *fast-DQP* branch by executing `git checkout fast-DQP` on the terminal. Then, activate the conda environment using `conda activate PlanningWithSubgoals`.

Edit ejecutar_pruebas.py

Move to the *src* directory and open *ejecutar_pruebas.py*. In this script, you can set the training options as well as choose the network hyperparameters.

Let's set `num_rep` to 1, since we want to train and validate the model only once. Then, set `model_id = curr_rep` and choose the model hyperparameters, e.g., `num_its = 500`, `alfa = 0.005`, `dropout = 0.2` and `batch_size = 16`. These hyperparameters can be easily changed from this script because they are the ones we tested most during the development of this project. However, if you want to change the CNN architecture, you can do so by editing the *LearningModel.py* script. Finally, if you don't want your computer to shutdown after the script has finished executing, substitute the line `os.system("shutdown -h +1")` with `pass`.

Edit oneclickRunFromPythonClient.sh

In case the *visuals* parameter is set, remove it. Remember that, during training, the agent does not really play the game, it just loads several datasets and trains the model on them. For that reason, it's better to turn the visualization off.

Start the Execution

On a terminal, execute `python ejecutar_pruebas.py`. This will train the model on several datasets and then will evaluate each of the trained models on the two validation levels. The number of actions used to complete the validation levels by each model are output to *test_output.txt*.

After the execution has finished, *test_output.txt* should show something like this:

```
--------------------------

Model Name: DQN_alfa-0.005_dropout-0.2_batch-16_its-500_0

500 - level 0 - 82, level 1 - 148
1000 - level 0 - 99, level 1 - 125
2500 - level 0 - 116, level 1 - 121
5000 - level 0 - 117, level 1 - 197
7500 - level 0 - 101, level 1 - 158
10000 - level 0 - 91, level 1 - 162

```

2.4 Testing a Model

This tutorial will show how to easily test a trained model on the 10 test levels using the *ejecutar_test_modelos.py* script. We will be using the *fast-DQP* branch, although the steps are the same for the *fast-Greedy* branch.

Switch Branches and Activate Conda Environment

Switch to the *fast-DQP* branch by executing `git checkout fast-DQP` on the terminal. Then, activate the conda environment using `conda activate PlanningWithSubgoals`.

Edit ejecutar_test_modelos.py

Let's use *ejecutar_test_modelos.py* to test the model we trained in the previous tutorial. The name of this model is *DQN_alfa-0.005_dropout-0.2_batch-16_its-500_0* and was saved inside the *SavedModels* directory.

Then, in *ejecutar_test_modelos.py*, set `nombre_modelos = "DQN_alfa-0.005_dropout-0.2_batch-16_its-500_"`, `id_ini = 0` and `id_fin = 0`. `id_ini` and `id_fin` are used to test several versions of the same model architecture. However, this is not the case here, since in the previous tutorial we only trained one version of the model (we set `num_rep` to 1 in *ejecutar_pruebas.py*).

Edit Agent.py

In *Agent.py*, set `self.EXECUTION_MODE="test"`.

Edit CompetitionParameters.py

In *CompetitionParameters.py*, set `TOTAL_LEARNING_TIME=0.1*MILLIS_IN_MIN`. The agent cannot directly jump into the test levels without playing at least one training level before. This way, by setting `TOTAL_LEARNING_TIME` to a very small amount of time, we make sure that, as the agent finishes the first training level, the training phase ends and it starts playing the test levels.

Edit oneclickRunFromPythonClient.sh

In *oneclickRunFromPythonClient.sh*, set the *visuals* parameter on if you want to see the agent play the test levels and set it off otherwise.

Clean test_output.txt

Delete the contents of *test_output.txt*, since the performance of the model on the test levels will be saved in this file.

Start the Execution

On a terminal, execute `python ejecutar_test_modelos.py`. This script will start testing the trained model on all the test levels and will save the number of actions to *test_output.txt*. After execution, open *test_output.txt* and you should be able to see something like this:

```
  --------------------------------
-   Nivel 5 y nivel 6   -
--------------------------------

-------------------------

Model Name: DQN_alfa-0.005_dropout-0.2_batch-16_its-500_0

500 - level 0 - 86, level 1 - 69
1000 - level 0 - 136, level 1 - 83
2500 - level 0 - 58, level 1 - 56
5000 - level 0 - 115, level 1 - 55
7500 - level 0 - 64, level 1 - 77
10000 - level 0 - 128, level 1 - 60

--------------------------------
-   Nivel 7 y nivel 8   -
--------------------------------

-------------------------

Model Name: DQN_alfa-0.005_dropout-0.2_batch-16_its-500_0

500 - level 0 - 49, level 1 - 77
1000 - level 0 - 47, level 1 - 89
2500 - level 0 - 47, level 1 - 83
5000 - level 0 - 58, level 1 - 98
7500 - level 0 - 69, level 1 - 86
10000 - level 0 - 37, level 1 - 83

--------------------------------
-   Nivel 9 y nivel 10   -
--------------------------------

-------------------------

Model Name: DQN_alfa-0.005_dropout-0.2_batch-16_its-500_0

500 - level 0 - 82, level 1 - 111
1000 - level 0 - 100, level 1 - 106
2500 - level 0 - 81, level 1 - 98
5000 - level 0 - 79, level 1 - 101
7500 - level 0 - 53, level 1 - 89
10000 - level 0 - 65, level 1 - 96

--------------------------------
-   Nivel 11 y nivel 12   -
```

```
--------------------------------

-------------------------

Model Name: DQN_alfa-0.005_dropout-0.2_batch-16_its-500_0

500 - level 0 - 67, level 1 - 55
1000 - level 0 - 74, level 1 - 59
2500 - level 0 - 76, level 1 - 81
5000 - level 0 - 74, level 1 - 59
7500 - level 0 - 75, level 1 - 80
10000 - level 0 - 71, level 1 - 77

--------------------------------
-   Nivel 13 y nivel 14   -
--------------------------------

-------------------------

Model Name: DQN_alfa-0.005_dropout-0.2_batch-16_its-500_0

500 - level 0 - 82, level 1 - 148
1000 - level 0 - 99, level 1 - 125
2500 - level 0 - 116, level 1 - 121
5000 - level 0 - 117, level 1 - 197
7500 - level 0 - 101, level 1 - 158
10000 - level 0 - 91, level 1 - 162

```

Parse the output

In case you want to encode the contents of *test_output.txt* as CSV, you can use the *python test_output_to_csv.py* script.

Open it and edit the following: `pruebas_por_modelo = 6`, `num_modelos = 1`, `num_niveles_test = 10`. `num_niveles_test` is the number of test levels used, `num_modelos` is how many models we have tested and `pruebas_por_modelo` is how many datasets we have trained each model on.

Execute `python test_output_to_csv.py` and, within the same directory, you should be able to see a new file called *test_output_parseado.csv*. If you open it as plain text, you will see something like this:

```
- Nivel 5 -
86
136
58
115
64
128

- Nivel 6 -
69
83
```

```
56
55
77
60

- Nivel 7 -
49
47
47
58
69
37

- Nivel 8 -
77
89
83
98
86
83

- Nivel 9 -
82
100
81
79
53
65

- Nivel 10 -
111
106
98
101
89
96

- Nivel 11 -
67
74
76
74
75
71

- Nivel 12 -
55
59
81
59
80
77

- Nivel 13 -
```

```
82
99
116
117
101
91

- Nivel 14 -
148
125
121
197
158
162

```

Each - *Nivel X* - section corresponds to the performance of the model on test level number *X*. The numbers of actions are listed starting from the smallest dataset. For instance, the model trained on the smallest dataset has used 86 actions to complete level 5 and 128 actions when it was trained on the biggest dataset.[1]

[1] As can be observed, the results obtained are really bad. This is because, in order to speed up the execution of the train and test tutorial, the model was trained for only 500 training iterations. To improve the performance, it is advised to train it for 5000 training iterations.