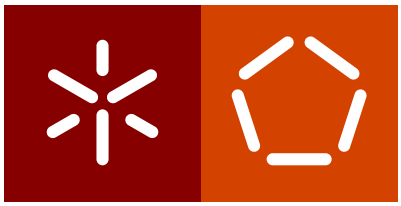


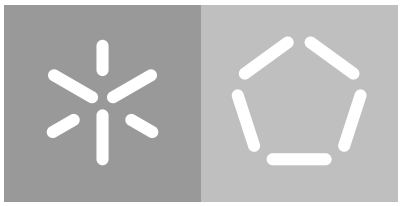
Universidade do Minho
Escola de Engenharia
Departamento de Informática

André Almeida Gonçalves

Learning to play DCSS with Deep Reinforcement Learning

Aprendendo a jogar DCSS com Deep Reinforcement Learning

December 2019



Universidade do Minho
Escola de Engenharia
Departamento de Informática

André Almeida Gonçalves

Learning to play DCSS with Deep Reinforcement Learning

Aprendendo a jogar DCSS com Deep Reinforcement Learning

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
Rui Mendes

December 2019

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ACKNOWLEDGEMENTS

To professor Rui Mendes, who suggested this project and helped me along the way.

To my parents for their support during all these years, and all the efforts they had made to provide me with the conditions necessary for me to fulfill my wishes.

To my brother who always motivated me, and made me stay focused and on track.

To all the friends that joined me in this adventure in 2014 and until today had never failed me, and always been there for me.

To Afonsina, Tuna de Engenharia da Universidade do Minho that was always a safe port and second home to me.

And to my girlfriend, Cláudia Serra, that always has been there for me, always helped me and shown her support, my personal best friend, my companion of many adventures that have passed, and many more to come.

To you all, my dearest thank you.

RESUMO

DCSS é um jogo *roguelike* no qual o jogador deve explorar e encontrar artefactos. A cada passo do jogo existem decisões a tomar e a complexidade do jogo reside na grande quantidade de opções disponíveis para o jogar a cada momento do jogo. Os comandos podem ser divididos em classes como: movimento, combate, gestão e utilização de inventário, magia e abilidades divinas.

O objetivo é implementar um *bot* inteligente capaz de jogar o jogo. Para tal usar-se-á *DRL*.

DRL é a área onde *deep learning* e *RL* se encontram. Usa os mesmos princípios de RL, para aprender a realizar uma tarefa, recebendo uma recompensa por cada ação efetuada. A diferença é que a ação vai ser escolhida por uma *DNN*.

Durante o projeto utilizar-se-à PyTorch[11] como a framework para implementar uma *NN* que irá ser capaz de encontrar uma solução para cada pequena decisão que pode ser efetuada ao longo do jogo. Tomadas todas essas decisões, o bot deve ser capaz de jogar o jogo com algum grau de sucesso.

O objetivo de um bot inteligente é aprender uma tarefa, neste caso, jogar o jogo, sem programar qualquer comportamento.

Keywords: Dungeon Crawler Stone Soup, Inteligência Artificial, Aprendizagem por Reforço, Deep Reinforcement Learning

ABSTRACT

DCSS is a roguelike game in which the player must explore and find artifacts. In every step of the game, there are decisions to make, and the complexity of the game resides in the vast amount of options available to the player at any given time. The commands can be divided into classes such as movement, combat, inventory management and usage, spell casting, and divine abilities.

We aim to implement an intelligent bot that will be able to play the game. To do so, we will use DRL.

DRL is where deep learning and RL meets. It uses the same principles of RL, to learn to perform a task, receiving rewards for every action made. The difference is that the action we will perform is chosen by a DNN, and therefore, we call it DRL

PyTorch[11] will be as the framework used to implement a NN that will be able to find the solution for every small decision that can be made during gameplay. If all of those decisions are merged, an intelligent bot should be able to play with some degree of success. The aim of the intelligent bot is to learn a task, which in this case, is playing the game, without programming any real behavior.

Keywords: Dungeon Crawler Stone Soup, Artificial Inteligence, Reinforcement Learning, Deep Reinforcement Learning

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

A

AI Artificial Inteligence

C

CNN Convolutional Neural Network

D

DCSS Dungeon Crawler Stone Soup

DNN Deep Neural Network

DRL Deep Reinforcement Learning

M

MDP Markov Decision Process

N

NN Neural Network

R

RL Reinforcement Learning

RNN Recurrent Neural Network

S

SL Supervised Learning

1

INTRODUCTION

1.1 CONTEXT

AI is a field that is currently under research, where the goal is to make a machine learn how to perform a task without the need to program its specific behavior. Instead, the machine will learn from data that is given to it, and by observing multiple cases of the same problem, and its outputs, learn how to predict the best action to perform.

RL[14] is one method of AI. In this approach, instead of giving data to our network, we will run our network on the environment, and it will learn by executing actions on it and from the reward received as feedback from that action. This is made by using one agent(or more) in an environment.

The agent is given the current state of the environment and it produces one action, the outcome of this action in the environment can be measured and used as feedback.

If the action puts us closer to the goal, we give it positive feedback: a reward, if not, we can give it negative feedback: a negative reward. Knowing this we can set out a goal and start training the agent to make more efficient decisions. In this case, our goal is to play the game with some degree of success.

After several steps of training, the agent will be able to make decisions on problems with some degree of complexity.

DRL[10] is a step further from RL, the difference between the two methods is that the first uses NN to compute our decision.

1.2 MOTIVATION

Every day more components of AI are being integrated in our lives, from facial recognition[7] on social media, to automatic music recommendation[16].

AI already exists since 1956 but it is been only a few years since we can use it to help us in our daily tasks, because it's been only a few years since we were able to use lots of computational power to perform these tasks, and this aligned with the tools that tensorflow[1] and other libraries give us, made AI available to anyone.

There used to be 2 main types of AI, supervised[6], and unsupervised [2] learning, RL came after these two.

In supervised learning, we use labels on our data. This means that if we want to build an image classifier that can identify a picture of a dog from a cat, we start by training the network with several samples of images labeled as "dog" and several as "cat". From this, our network will try to find patterns in the pictures from which it can differentiate the two classes and, when a new sample is presented, it will be able to classify it as one of the classes.

On the other hand, unsupervised learning does not use labels. It learns from unknown patterns in data. It can be used, for instance, to write text. To do this, we need to train our network with samples of sentences, after the train, it will start to recognize the structure needed to build a text and will start to apply it, therefore writing sentences.

RL takes a different approach, it uses states and actions, and with the concept of environment and agent, applies actions to the current state and learns from the changes in the environment.

The reason RL is so used in games is that a game is a controlled environment, we do not have more actions than those that the rules allow us to make, we have a state that is set, and there are no external variables that can influence our actions. For instance, in real life we can set a goal of walking 20 steps, if we train a network to do this, we have a lot of things to check first, from big, obvious things like checking if there is no obstacle or if next step is in a free space or an occupied one, to small things, the status of our shoes can put at risk the making of the task, the floor adherence, and so many more variables. In a game we can ignore all those small variables, if we say walk, we need to check if the direction is valid, all those extra steps are not made because we are working in a controlled environment and will not interfere in our process of learning. This makes the process that much easier, and as a proof of concept is an ideal scenario. Knowing this, learning how to play DCSS will be a perfect case for using this method, the game is complex enough to use RL in a demanding way since there are a large number of commands we can use and lots of variations of the game, since enemies are randomized in each run of the game, and the map level and its items are generated randomly every time.

1.3 Main aims

I propose to make a bot that is capable of learning. There is only 1 bot available online that can play the entire game and win the game, the bot was made in LUA [5] programming language and it was eight thousand lines of code, it is very hardcoded and specific to this game.

With DRL we can decrease the number of code lines necessary to make this task, and ultimately we will be able to use can use some parts of the bot to implement on another game if the data given addresses another game by making this a modular project.

So, as the main goal, we want to make a bot that can play the game to its fullest state without being hardcoded for it.

2

STATE OF THE ART

2.1 REINFORCEMENT LEARNING CONCEPTS

To fully understand all the content of this thesis we first need to understand the small concepts that make the big picture of deep reinforcement learning. In this section, we will cover the necessary concepts to gain full knowledge of the theme.

2.1.1 *Environment*

The environment is where we get our data, regardless of the problem in hands, the environment is what will give us the input for the problem.

This environment can be extended or minimized as needed to increase or decrease our problem complexity, respectively.

For instance, imagine a problem where our goal is to keep the temperature of a room constant.

If we want to simplify the problem we can set our environment as the temperature we get from the heater, but this may be a bit of overuse for a NN, the results that the network is going to give are very superficial.

Now imagine that we want to extend the problem, our world passed from being the temperature registered by the heater to be the thickness of the walls, number of persons in the room, space of the room, outside temperature, wind speed, number of windows, number of doors and the temperature of the heater.

This time the problem seems slightly more complex as we set a bigger environment, and this may lead to a rise in the efficiency of the algorithm.

2.1.2 *State*

Although our environment does not change, the state of it changes along with the experience. We can say that the environment is what we are watching, our state is the value of what we watched. For instance, if our environment is a value of the temperature of a room,

our state will be the value of the temperature at a given point of time. So, our environment will always be represented as a number, but the state of the environment is what tells us what number it is.

2.1.3 *Agent*

An agent is an entity that can take action in our environment. The agent has a set of actions that it can execute and change the state of the environment from the given action.

2.1.4 *Action*

Action is the way an agent can interact with the environment. From the start of our problem we can say that we have an N number of actions, from those, at every step, the agent needs to choose one. This action will change our state of the environment, the agent can analyze what consequences resulted from it, and decide if the action was good or not.

The actions taken by the agent can be split into two groups. The first one is when the agent still does not know enough information to make an intelligent decision and given that he does not have enough information, he will try to gather more by trial and error. To this process we call exploration. On the other hand, if the agent already knows enough about the environment he can start trying to use information that he has learned before. This is called exploitation.

2.1.5 *Markov Decision Process*

A state can be said to be Markovian[12] if, from it, we can predict out future states without the need to look at previous ones. Imagine an example of a car race, our state can be defined for the car position, speed, and orientation. If we know this we do not need to know the previous state for getting a future state, this state is representative enough for achieving future states. But instead, if our state only represents the car position, for instance, this information is not enough to get our next state, because the speed and orientation are also needed to get that. So this state is not considered Markovian.

Reinforcement learning is often modeled as a *MDP* which is a graph with states for nodes and edges that describe the probabilistic transition between Markov states.

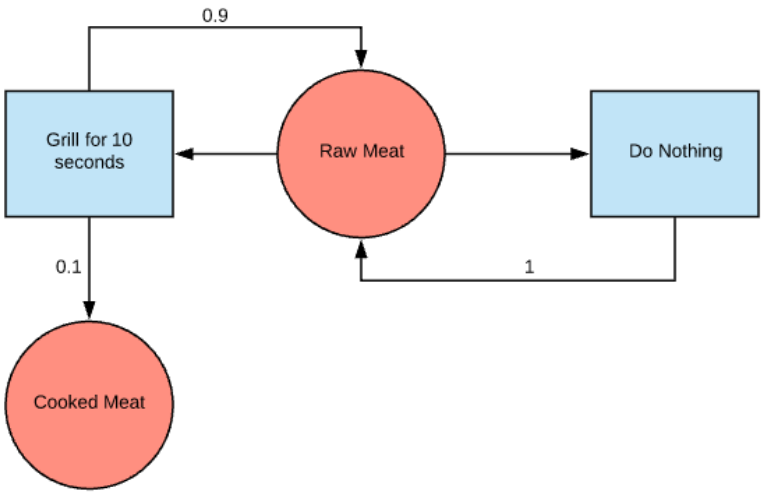


Figure 1.: Markov Decision Process on Cooking Meat.

On the example above we have 2 states(red) and 2 actions (blue). We start at state "Raw Meat", from there we can do 2 actions, we can "Do Nothing" and there is a probability of 1 of the meat remaining Raw, and we go back to the previous state.

Or we can execute the other action which is "Grill for 10 seconds", this has a 0.1 chance of cooking the meat and 0.9 change of the meat still be raw after that action.

The actions on a MDP are often non deterministic. We can execute an action and it can lead us always to the same resulting state or not.

The goal of a RL algorithm is to spend the most possible time in a more valuable state to increase our reward and our actions will be taken accordant to this law.

At the beginning of the problem we do not know all the states of the MDP and we do not know what actions will take us to each state. To this mapping, we call exploration as we try to explore the problem in all its complexity.

2.1.6 *Reward and Return*

Our goal in RL is to maximize reward, this includes immediate reward and reward that we may get a few steps away from an action taken in this step. The sum of all the rewards taken from now to the end of the episode is called return and it can be calculated by the formula:

$$R = r_{t+1} + r_{t+2} + r_{t+3} + r_{t+4} + r_{t+5} + ... = \sum_{n=1}^{\infty} r_{t+k}$$

It is also very common to use a future cumulative discounted reward given by the value:

$$R = \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \gamma^4 r_{t+4} + \gamma^5 r_{t+5} + ... = \sum_{n=0}^{\infty} \gamma^k r_{t+k+1}$$

$$0 < \gamma < 1$$

This means that if we set our discount as 1, we value the future rewards as much as the immediate rewards, and if we set it to 0 we do not value the future rewards at all, and therefore, our return is this step reward. We can adjust the value of γ accordingly to the value we want to give to future rewards.

2.1.7 *Policy*

A policy describes a way of acting. Given a state, an active policy and an optimal policy can be written in the form of :

$$\pi(s, a)$$

$$\pi^*(s, a)$$

The policy function takes in a state and an action and gives us the probability of taking a certain action. For example in a case where we are at a state and we are given 2 actions to choose from, a random policy would be:

$$\pi(State, Action1) = 0.5$$

$$\pi(State, Action2) = 0.5$$

Our goal is to find the optimal policy. This is a policy that tells us how to maximize returns in every state. An one optimal action to take in each state can be written as:

$$\pi^*(State) = Action$$

2.1.8 *QValue*

In a simple reinforcement learning problem, with a low amount of states and actions, it is possible to map every action to every state in a table, and therefore, mapping every reward of every action on every state.

To the values of reward mapped this way, we call QValues.

In a bigger environment, this mapping is not so trivial and therefore is not saved in a table. Instead to get the QValue, we use a function that approximates our QValue for the given action in the given state.

2.1.9 *Value Function and Q-Value Function*

Value and Q-Value functions are functions that tell us how good it is for an agent to be in a given state or how good it is for an agent to perform a given action in a given state.

We can set these values in the form of expected return, the value that we expect to receive. This value depends on the action the agent decides to take on a given state. Here is the formula for both respectively.

$$V^{\pi}(s) = \mathbb{E}_{\pi}[R_t | s_t = s]$$

Where the Value of state s is given by the reward R on time t, given that the state s is the state on the time t.

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}[R_t | s_t = s, a_t = a]$$

Where the Q-Value of state s and an action a is given by the reward R on time t, given that the state s and the action a are both on time t.

2.1.10 *Replay Memory*

To prevent the correlation between different samples of data, we save every step of the game to a stack of experiences, this is called the problem memory. In the going of the experience, instead of using a step of the game when it is given, we save it to our memory and then sample a random step, avoiding using samples obtained in a row and thus, breaking this correlation.

2.1.11 *Loss*

In a neural network, our loss is the error made in each step of the training. It is what tells us how good, or bad a network is at the current state.

The loss is calculated by the difference between the outputted value and the expected output value. For instance, in an image classifier where we have two classes represented by a zero and a one, and our input is labeled as a zero, if the network returns 0.25, the loss at the current step will be calculated by:

$|0.25 - 0| = 0.25$

Our goal is to minimize our loss, and therefore, minimize the error in our network predictions.

2.1.12 *Epsilon Greedy Strategy*

Epsilon greedy is what determines if we should use our knowledge of the problem to decide what action to take, or if instead, our knowledge is not wide enough and thus, the action chosen should be a random one to gain more knowledge and improve the network learning process.

2.1.13 *Exploration vs Exploitation*

From the epsilon greedy strategy function we can determine if our network is capable of making a decision.

If it is, the network will, and we call this an exploitation action, if it is not, the network needs to explore the environment and, to avoid making always the same action, and therefore, not gain knowledge from the environment, it chooses a random action to execute. We call this an exploration action.

2.2 GAME SPECIFIC CONCEPTS

To better understand our problem and the setup of our experiences, is it also needed to know some game specific concepts.

2.2.1 *Webtiles*

DCSS is distributed in some different formats. From text-only formats that run on a terminal, to visual versions where the user can see each cell of the game, or tile, represented on the screen. This version of the game is also available to play online in the browser, and it is known as webtiles.

2.2.2 *Lobby*

To join a game in the webtiles server, the user must pass through the lobby. This is the online menu of the game.

It enables the player to choose what version of the game he wants to play, editing the configuration file and seeing other players playing the game.

2.3 REINFORCEMENT LEARNING

2.3.1 *Supervised vs Reinforcement Learning*

The main difference between the approaches in *SL* learning and RL is the feedback. In SL we have instructive feedback. This means that the feedback teaches us how to reach our goal, while RL uses evaluative feedback that tells us how well we are achieving our goal.

Imagine, that we are building an image classifier, and we try to train it using both approaches.

In supervised learning, we try to teach the network that this image belongs to that class, and we can test the network by giving it a true or false value, and based on the percentage of correct guesses we can say that the network is, or not, well trained.

On the other hand, using the reinforcement approach, when we say, this image belongs to that class, our feedback is a number, for example, fifty points. Without any context we do not really know how good that is, we do not know how many points should a right answer receive. We need to train the network on other cases to see if the fifty is a good value, or if the next case actually gives us a thousand points, putting the previous fifty as a bad result.

The idea of evaluative feedback is more intuitive in some cases. Let's now imagine another case, where our output is not a class but a number, for example, a system that tries to balance our house temperature. If we use instructive feedback how can we tell the network that this is the right temperature? On the other hand with evaluative feedback, we can use the value of the temperature as points and using a simple formula like the difference between our wanted temperature and our temperature and simply trying to minimize the difference.

Formalizing Reinforcement Learning

The most important components of a RL are the agent and the environment. The agent exists in an environment and has control over it, by taking actions. The agent can change the state of the environment and by doing so, it will get a reward from that action[15].

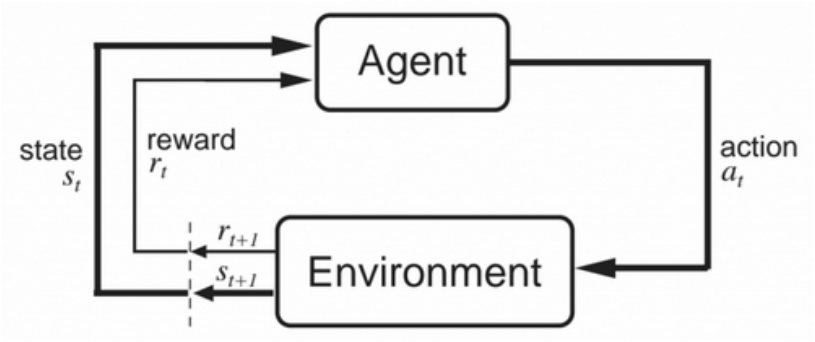


Figure 2.: Image from Sutton and Barto - Reinforcement Learning an Introduction.

The agent gets a state s(t), and based on it, decides to execute an action a(t), this action as an effect on the environment and gives us the next state, s(t+1) and a reward r(t+1). Although this reward is from time t, it is called t+1, because it is returned with the state s(t+1) and it makes sense to keep them together as seen above.

2.4 Deep Reinforcement Learning

Although the RL algorithm can solve examples with very simple cases, if we add more states and actions to our problem, is not that easy to map every state-action pair.

Because of the notorious number of cases that this produces the performance will drop significantly when adding a large number of cases. From now on, rather than using value iterations to directly compute Q values and find the optimal Q function, we will use a function approximator to estimate an optimal Q function.

To approximate a function we will use NN. We will use a DNN to estimate the Q values for each state-action pair in a given environment, by this process the network will approximate the optimal Q function.

2.5 Deep Neural Networks

The DNN will accept a state as input and will return as an output the estimated Q value for each action that can be taken from that state. The loss of this DNN can be calculated from comparing the outputted Q values to the target Q values from the right side of the Bellman equation, and, as normal in NN the objective is to minimize the loss. The network will be trained enough times for the loss to be minimal and stable so we can approximate an optimal Q function. The problem solved is the same in RL and DRL, the main change is the algorithm, rather than using value iteration, we are now using a deep neural network.

2.5.1 *What is a Deep Neural Network*

A DNN[13] is an NN with a few specific characteristics. Normally, for a model to be considered deep learning it needs to have a significant amount of layers, furthermore for a model to be considered DNN it must have hierarchical levels, in which we find simple features first and from those, we evolve into some more complex features.

2.5.2 *Convolutional Deep Neural Network*

A *CNN*[6] is a NN used for deep learning in which some layers feature convolutional levels. This means that this NN is a DNN where some levels represent data convolutions. This type of network is very often used for image treatment.

Images are a form of data that have a lot of useless information, the convolutional layers take care of that information, by reducing the amount of information we get to only keep the useful information and discard the rest. This layer does this discard action wisely, it learns what to value more or less when it comes to reducing information and keeps the most valuable data. By this process we achieve the goal of reducing our data and more easily pass it through the rest of our network, speeding up the process.

This concept can also be used for other types of data, such as music sheets, text, and we can also use it to read the state from our RL environment. In this last case, the network will learn what data to value more in our state and prioritize those values.

2.5.3 *Recurrent Deep Neural Network*

RNN[9] is a type of network with a feedback loop. This loop takes out the output of the network and uses it as input for the next step of the network. For instance, this type of network is being used a lot in text creation, where the network is trained on sequences of text sentences. Using the network will consist of giving it an initial character and from that, the network will produce the next one, after that this character that was generated in the previous step is used as input to generate a second character. This loop through this sequence will make the network produce text. This type of network can be used in DRL to preview our next state, given this state and this action the next state should be this.

2.6 USE CASES

2.6.1 *Atari Games*

DRL is already used to play Atari games, and it is able to beat even experts with some time to train.[15]

The process starts with feeding the NN with images, these images are frames from the game and represent our states from the game.

Figure 3.: Breakout game by Atari.

The prepossessing starts by converting the RBG data to a grey-scale image, knowing that a pixel is represented by 3 floats (Red, Green, and Blue), converting to a grey-scale gets our data size to about 1/3 of the original size.

The next step is cropping and scaling the images in order to minimize even more the data that will be fed to the network and cut unimportant information from the frame.

The state that is given to the network is not only 1 frame, because 1 frame cannot be considered a Markov, it is not sufficient to evaluate the state of the game at the current point, instead, we feed the NN a stack of consecutive frames, this will give us information on the speed of objects in the game, directions, and other important data.

This stack of frames is fed to a convolution DNN and our output will be given from a fully connected layer, and it will produce a Q value for each action that can be taken from a given state.

The agent takes the action given by the DNN and the reward is used to calculate the loss for that state, the DNN will try to maximize those rewards. Eventually, the agent will be able to play the game integrally.

2.7 DUNGEON CRAWLER STONE SOUP

2.7.1 *The game*

Dungeon crawl[3] is a roguelike game that began in 1995 as a project of Linley Henzell. In 2006 a variant of the game called DCSS was created. This game is still in development today and most of the Dungeon Crawl Players have switched to it.

The game is a turn-based game where you start by choosing a species and a background. This combination will determine the character that you will play with. You also get to pick your initial weapon. When this is done you see your character in a 2D tile game.

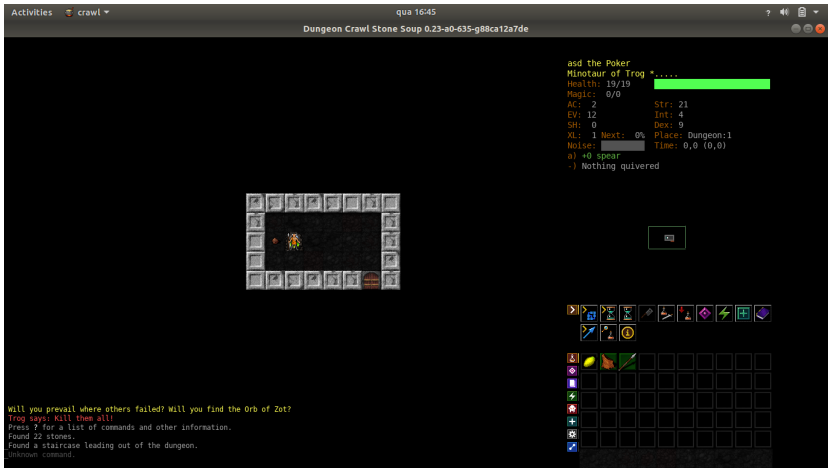


Figure 4.: Initial game state.

The game map is generated randomly in each game. Since the game is turn-based when you play a turn, the computer plays one turn too, the basic actions cost one turn, for instance moving one tile. Attacks and other actions can cost more or less turns. You can have a light or a heavy weapon, this can increase or reduce your speed.

The game is played by exploring the map, finding items, fighting enemies and gathering food to avoid starving. In the process, you level up and increase your game stats.

The game is a dungeon where you can use stairs to go one floor down or up, the stairs only connect two levels, in the next level you must find a new set of stairs, and in the process evolve your character.

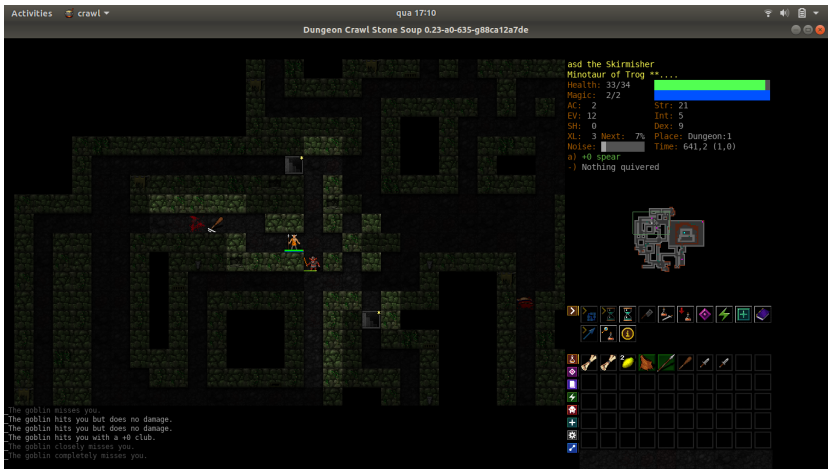


Figure 5.: Example of game environment after a few turns.

The goal of the game is to go down the dungeon, explore it, collect runes and steal the orb and come back to level 1 and exit the dungeon[4].

2.7.2 *The environment*

The game environment is composed of several parts. The core part is the map, this is where our character is. It tells us what ways we can follow, the enemies around us and items currently on the map. This is the core of the environment. Together with this we have the bag where we can carry items, spells, and equipment. And finally our player's current state, like his health points and stats.

2.7.3 *The actions*

The main actions on the game are moving, eating, attacking and use spells and items. The move action can be done via auto-explore, this is a feature of the game to avoid boredom, the game will start auto-explore if there are no monsters on the map near us, so with attacking and auto exploring we can make our first bot.

The rest of the action will come later and step by step we will implement more actions on the agent for it to have more chances to win the game.

2.7.4 *The goal*

Our goal is to train the agent to beat the game. Due to its complexity, the game is extremely difficult, and only a very small percentage of the players that have already played the game got to finish it.

To build a bot that can accomplish this is our main goal. There is already a bot that we can see online that can do this, but this bot has all of its behaviors programmed for all the situations it may encounter. This led to plenty of code lines, to avoid this we will use AI to learn what decisions to make in each step of the game.

3

THE PROBLEM AND ITS CHALLENGES

Being this a machine learning problem, there are some main concerns. To be able to run the network we need to read the state of the game, as this will be the input in our network. We also need to make sure the state of the game is updated correctly at each step.

In the network itself, we need to worry about the system architecture, how it fits the problem, and since we are using deep reinforcement learning, the reward from each step, and how to compute it.

The third main concern is the output of the network and how to execute the command. These series of commands will play the game and update the state to run the next step of the network.

3.1 THE NETWORK INPUT

The network input is the current state of the game. This state must be updated at each step of the network to have a valid input and generate a valid action for that state.

The state is composed of the game map and some important data about the character.

The map information is achieved by a JSON message that the game sends whenever the game map is updated. The rest of the data is not sent by the game every step.

To get it, it was needed to create a .rc file. This file is a minor script that is applied to the game. This file is a Lua script that runs each step of the game, and by this script, we can access data that otherwise would not be available. We print this data to the message board of the game, and since every message on the message board, is sent by JSON messages, we can now access them and update the state.

3.2 THE NETWORK

The network itself was made with the python library, PyTorch. This library provides arrays, that can be used in array multiplication, and by stacking multiple multiplications and filters

of the input, we can make a NN. To get a good architecture, several tests were made. By this process we were able to find a network that fits the problem.

To find a network that fits the problem and spends the least computer power possible, the tests began with networks of low complexity and a small number of layers, and then layers were added to increase the complexity of the network to be able to resolve the problem.

3.3 THE OUTPUT

The network outputs an array of values that represent the certainty of each command being the best command to play in this state. To get the command, an argmax command is applied to find the command index with the highest probability of being the best command. Given that command, it is used a python library that redirects our commands to the standard input of the game's process, so, in order to play the game, the user must select the tab where the webtiles version of the game is running, and the game plays itself as if it was a human player typing in the keyboard.

3.4 SYSTEM ARCHITECTURE

The architecture of the project is based on 4 packages that are described next.

3.4.1 *Game Connection*

This package enables the connection between the game and the network. It receives the JSON messages from webtiles, and sends all the messages relative to the current game to the main process, ignoring all the messages from the other games in the lobby.

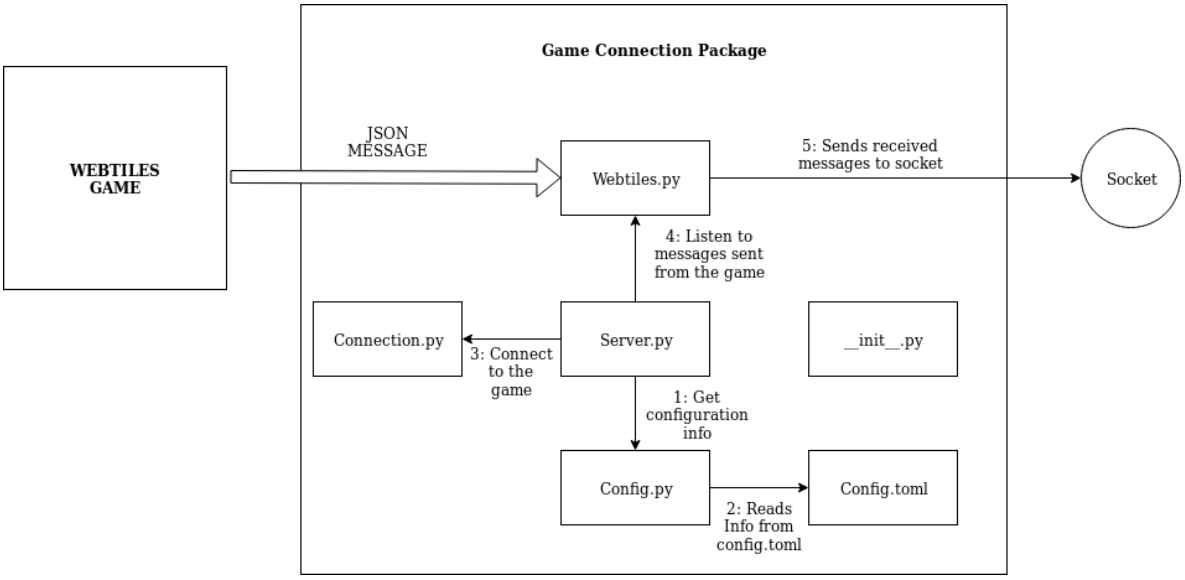


Figure 6.: Game Connection Package Architecture.

3.4.2 *Environment*

The environment package is composed of the environment of the game itself. This class will keep the information about the current state of the game. Besides that, this class also keeps the messages maps handler and the state updater.

This state update runs in a thread to receive messages from the game connection package and update the environment itself.

Finally, this package also includes a class that sends our commands to the game.

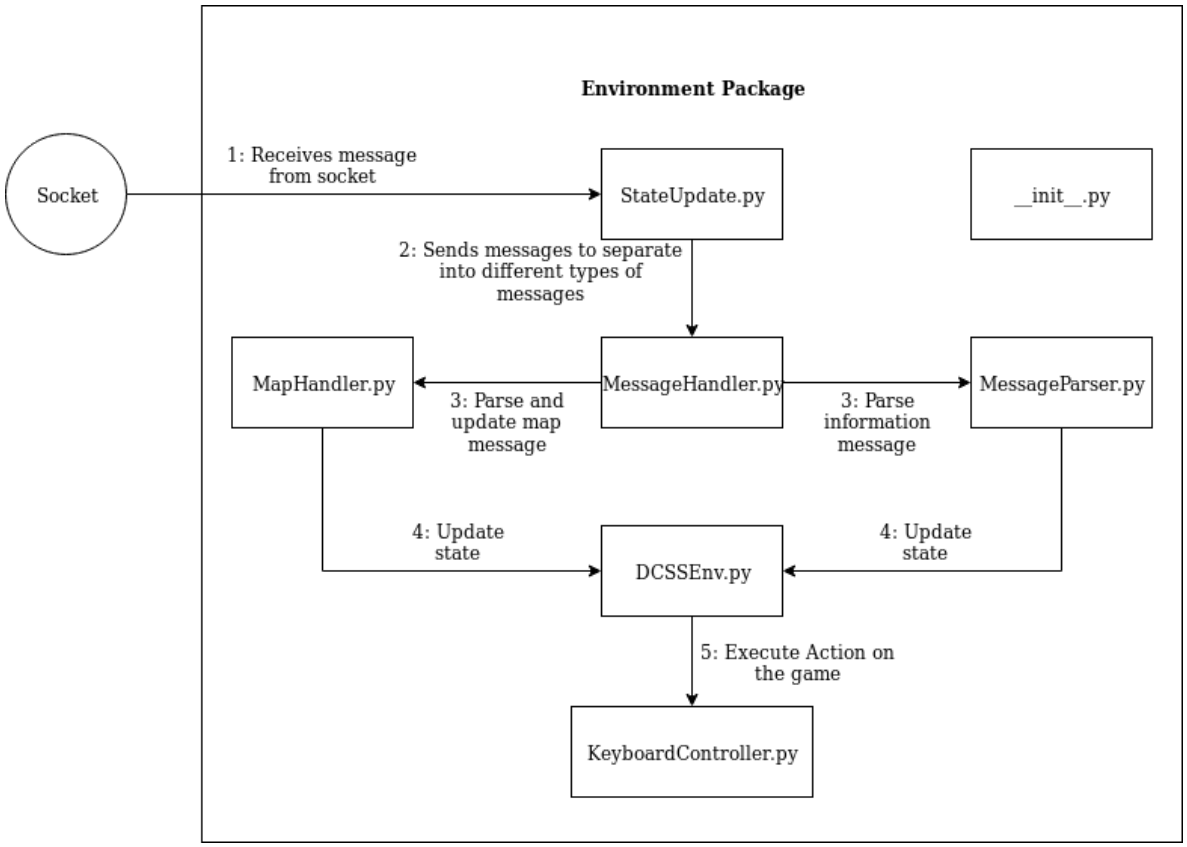


Figure 7.: Environment Package Architecture.

3.4.3 *Reinforcement*

The Reinforcement package is the package that holds all of the networks that were tested during this project.

This package also includes other classes that are important to train the network, such as the agent, that will run the network at each step of the game to get the action to execute, the epsilon greedy strategy, that decides if the network should make a decision about what action to play, or if it should play an random action in order to explore the environment and therefore, gain more knowledge of it. It also includes our replay memory where we are going to save our experiences during the training of the network.

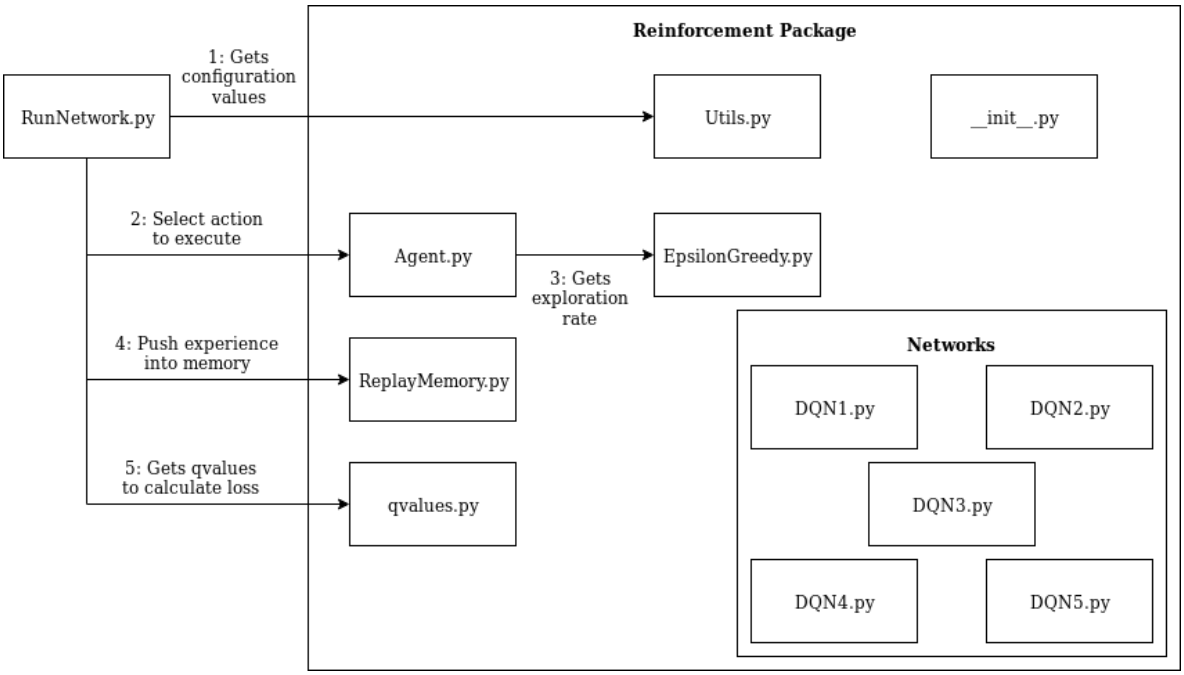


Figure 8.: Reinforcement Package Architecture.

3.4.4 *Webtiles Handler*

This class works as a post-training class. It has as responsibilities to treat the data that comes from training, to present it, and when the player dies, this is also the class that restarts the game.

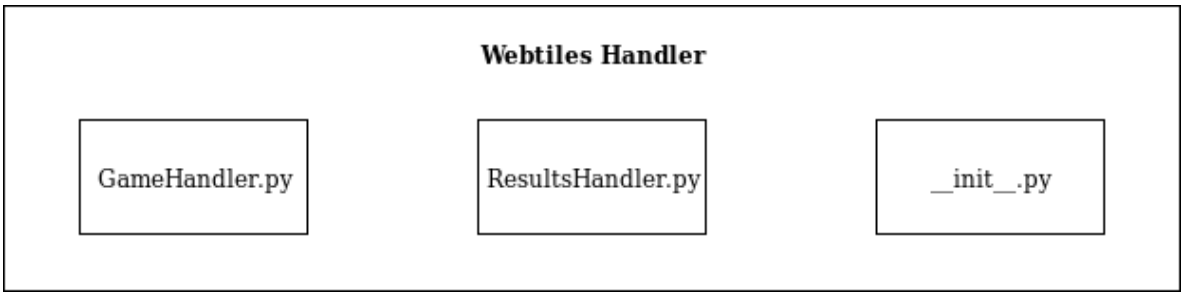


Figure 9.: Webtiles Package Architecture.

4

DEVELOPMENT

Almost all the development was made in the Python programming language. The game itself is made in Lua, and enables the user to add a little Lua script with configurations and some small functions. This configuration can be as simple as selecting what items the player should auto pick, to more complex functions that can implement fighting strategies, auto equip the characters, between others. It is also possible for the user to merge all of these functionalities and program the game to be self-played. This case was our starting point to make our intelligent bot. The most famous bot available online is called qw, and it was around eight thousand lines of code, making this a very verbose program to solve a specific problem.

In our implementation, our main focus was to make a bot that was capable of learning how to play the game and that could later be applied to a different problem.

To achieve this, we made our contact with the environment of the game as restricted as possible. By doing so, we guarantee that the code is reusable, and to solve another problem we only need to change the project parts that are specific to this problem.

4.1 DECISIONS

4.1.1 *Gathering Data*

Being this an artificial intelligence problem, the first main concern was how to get the data from the game.

Since we are using deep reinforcement learning, we do not need to worry about labeling data, but we must guarantee that the data that is fed to the neural network is up to date, because we are trying to find the relation between the state of the game, the action played and the next state of the game.

To make sure that the information is updated, an efficiency compromise was made. In every step of the game the bot needed to press a key, this key would produce a new state of the environment and at every step there is a new state created. Our first implementation would wait until receiving the new state from the game. The main problem here is that

sometimes a key passes multiple states at once, this means that only reading the first one will ignore all of the other messages, or, the next states will be received as the output of the next pressed key, this caused the environment to be not synchronized. To fix this, we need to slow down our bot. The solution was to wait a second between the action and reading the state. In our experiences we also tried to wait less that one second, but it was not enough to keep our environment updated to the current game state.

4.1.2 *Handling Data*

Our network input will be the current state of the environment. To have an environment where the network can learn, we need to choose which data will make part of it. This means knowing which data is relevant, and which is not.

To be a part of the environment we used the map of the game, because the network needs to know the character current position and its surroundings to be able to navigate through levels. In our representation of the map a single char represents a tile of the map of the game. To have a significant vision of the map, we feed the network with a range of sight of 10 tiles in every direction centered on the player, this means that there are 20 lines of 20 tiles, where each tile is a char, so the map length is 1600 units, this makes the major part of the environment.

In our map handler we decided to implement a easy way to change our map size. To do this we implemented a function that gets our map centered on our player, and returns a map with the size given in the arguments of the function.

The rest of it is some useful information such as, the health points of the player, that represent how much damage was the player taken, and from this, we can see if the player is in danger or not.

The player stats, intelligence, strength, and dexterity.

The level of hunger, because hunger is a very important concept in this game, where the player could starve and eventually die.

A boolean that represents if the player already reached the orb, which is the main objective of the game, to enter the dungeon, find the orb, and come back alive.

The time passed since the start of the game because since we have limited food and we must keep it until the end of the game to avoid starving, we are in a way running against time.

The current experience level of the player and the percentage of progression made to reach the next level.

And finally, if the current layer level is completely explored. When the level is explored to completion, the player must lower one level into the cave, since this level has nothing more to offer, and staying there is counter-productive in the hunger race.

4.1.3 *Multi threading vs Multi processing*

In this project, there is a need for constant message receiving and constant information update, to achieve this it is necessary to create new processes that can read and update our information.

There are some concerns we need to worry in the creation of the system. The first one is that the system must be failure tolerable, this means, if the game skips sending messages for any reason, the game must reset the connection with low effort. The second concern is sharing memory, the place where we write and read our memory must be the same, otherwise, the data read will not be updated.

In our project, we must have 3 things running at the same time. The first one is a message received from the game, the second is updating our information, and the third is the network itself.

Since the message receiver does not handle or save data, only sends it via socket, the second concern is not observed here, and since multiprocessing can be stooped more easily, making it failure tolerable, multiprocessing was implemented in the creating of the message receiver.

The network and the information updater must share information, so multiprocessing is not an option in this case. So multi-threading must be applied in this case. This created a problem, when we stop this thread, this effect is not instant, so the socket in the previous thread is still active, and trying to create a new socket with in the same port as the old one throws an error, making the program stop.

We were able to fix this by sending a message via a socket to the thread that causes the thread to close, but some times the thread is treating other messages, and while the thread is in this process of message handling, the socket is not listening for arriving messages, so it does not receive the exit message. To fix this we send messages to the thread in a loop, and wait for it to close. When it closes the message sent in the loop does not arrive, causing an error, so surrounding this piece of code in a try-catch block was the last step in order to make the thread failure tolerable.

To make sure the two conditions are accomplished, we implemented a system with multi-thread and multiprocessing.

4.1.4 *Problem Simplification*

The game itself is a very complex problem because of two factors, the complexity of the environment, and the number of possible actions.

To see results working with limited time and limited computing power, it was needed to create some simplifications to the problem.

By doing so we are treating the problem as a more simple one, excluding some decisions and some data.

When we start the game, the player is asked to choose one in twenty-seven species, each species has different abilities and general skills. The player can choose to be a Centaur, a humanoid with a horse lower half. This makes the player very fast, but very weak at everything except ranged combat, or a Minotaur, a bipedal bovine species, great in all forms of combat but very bad in magic casting, among many others.

After choosing the species of our player, we must choose one of twenty-four backgrounds, that are divided into 5 main classes, Warriors, Zealots, Warrior-Mages, Mages and Adventures. The background determines the character's life before entering the dungeon. The player can choose to be a Wizard, a broadly-educated magician, or a Berserker, a savage warrior, among many others.

After choosing our species and background it is possible to choose a god, but there are twenty-five gods in the game.

Gods will give players powerful abilities if the player chooses to worship them and make sacrifices during the game. But any violation of the god commandments will be punished. If the player worships the god Trog, the Wrathful, god of violence and rage it must kill as many enemies to please the god. If the enemies are magic-users, Trog will be very pleased, but if the user starts to use magic itself, Trog will count this as a break of his commandments, causing him to punish the player.

Players can abandon their god mid-game by worshiping to another god, but most gods will punish the player for abandoning their faith.

In the quest, a player will find many items on the floor, these items can be picked up and used. They vary from money to buy items in stores, food, equipment, scrolls, potions, and others.

During the game, our character can equip new things he may find in his way, making him stronger, read scrolls that can have several effects, like identification, healing and transportation, and drink potions to restore health points, or in some cases get poisoned. Since the player does not know the scrolls and potions effect before using them, the first time using an item is very risky.

The game also has abilities that grow stronger according to our faith to our god. One of the most known ability is Berserk, this fills the character with rage, making the player significantly stronger during a short time, and leaving the player handicapped for some turns after using it. If well used can be very useful, otherwise, it can leave the player in a weakened in a dangerous spot.

All of these are passive actions on the game, that influence a lot the gameplay. To focus on our problem, and make sure the network creates good results all of these actions were cut off our problem.

There are some species and backgrounds that are easier for beginner players, one of the common combinations is a Minotaur Berzerker worshiper of Trog, this combination avoids magic at all cost, and uses a lot a combat, so the main concern of the player is to explore and kill all the enemies we can, avoiding stronger ones. If we are not ready yet, worrying about fighting one enemy at once is also very important to avoid surrounding and taking damage from multiple places, so using walls and corners on the map is very important.

To our network we will use a Minotaur Berzerker, to avoid the use of magic and maintaining the same strategy in multiple games plays. We will also not implement equipping items, using spells and potions, praying to gods and using abilities, keeping the problem simple, and maintaining our focus on exploring the map and level up our character.

4.1.5 *Selected commands*

Since the problem was simplified we can now select which commands will be implemented. Since we are using Minotaur Berzerker, our main focus is to explore and kill all the enemies.

The player can move in eight different directions at every step of the game, left, right, up, down and all the four diagonals.

The player can also wait, this means seating in the same tile as the previous turn to restore health points or to wait for enemies to come to meet him if he is in a favorable spot instead of charging at them and losing the advantage of the field.

There is a shortcut to auto-explore the map, picking up some items like food and money automatically and stops when faces enemies or other items, to let the player decide what to do.

The player can also go up and down stairs to explore another level of the dungeon.

To attack an enemy the player must simply walk to the tile next to the enemy and move in the direction of the enemy.

The last implemented action was eating, to avoid starving. This means that the program must access the inventory to find where the food is in his backpack.

These thirteen actions are the output of our network, so the network takes as input an environment with one thousand six hundred and ten items and outputs the percentage of certainty of the thirteen actions, the action with more probability is selected and played.

4.2 IMPLEMENTATION

4.2.1 *Language*

It is very easy to write code in Python due to its low-level syntax, a huge community and plenty of libraries and frameworks. Being a scripting language and thus, not being com-

piled, makes Python slower than compiled languages like Java. Although this performance drop, most of the community in the artificial intelligence area still uses Python, so, there are plenty of papers and tutorials online based on the Python programming language, and based on that, we decided to build our project on python.

The only part of the project that it is not written Python is a small script, written in Lua, that is used to configure some aspects of the game and to be able to reach information about our character in every step of the game, that otherwise, it would not be available.

4.2.2 *Data Format*

All the messages received from the game come in the JSON format. JSON or JavaScript Object Notation is a file format that is ready for humans to read and write. It is completely language independent and it has been standardized to be easily used in all programming languages.

At every step the game sends some JSON messages, these messages contain information about the map of the game, about the player, the chat, and the message board. By reading all of these messages and parsing them we can build our environment and our state.

4.2.3 *Information messages*

Most of the messages received require a simple parse, just separating the message and reading the segment in the expected position is enough. Currently, we are only parsing a few messages, because only those are relevant to us. Most of those messages are automatically sent by the game, but to get all the information needed, two messages were added via the configuration file written in Lua. Those messages are written at every step in the message board, and since every message in the message board is sent automatically by the game, we can get that information outside of the game.

The two types of messages carry different types of information: the first tells the player stats, and the second one tells the inventory and what key corresponds to each item in the inventory.

```
{
    'msg': 'msgs',
    'messages': [
        {'text': '<darkgrey>Unknown command.<lightgrey>',
        'turn': 5, 'channel': 26},
        {'text': '<lightgrey>
        ReinforcementStats: Minotaur Berserker 19 9 4 21 false 4 5 D:1 0
        <lightgrey>', 'turn': 5, 'channel': 0},
        {'text': '<lightgrey>
        InventoryStats a weapon,InventoryStats b armour,InventoryStats c food,
        <lightgrey>', 'turn': 5, 'channel': 0}
    ]
}
```

The first message represents the Reinforcement Stats, and the information it contains represent, the race, background, the player health points, his dexterity, intelligence and strength, a Boolean representing if the player has or not the orb, (which is the goal of the game), his hunger in a scale from zero to six, (where zero means the player is fainting out of hunger, and six is engorged), the turns that have passed until now, the place in the dungeon where the player is. In this case, dungeon level one, and its experience progress until the next level.

The second message represents the inventory of the player. Right now, we can see that the player carries with him a weapon, armour, and some food. From this information, the only relevant information to us is the food and what key to press to reach it.

Besides those messages, the game itself also sends a few more that are parsed by us.

```
{
    'msg': 'msgs',
    'more': True,
    'messages': [
        {'text': '<green>
        You have reached level 2!
        <lightgrey>', 'turn': 404, 'channel': 12}
    ]
}
```

This message informs us the player has just leveled up.

```
{
    'msg': 'msgs',
    'messages': [
        {'text': '<lightgrey>
        Done exploring.
        <lightgrey>', 'turn': 1208, 'channel': 0}
    ]
}
```

This message lets the player know when the level is fully explored, and therefore, he can move on to the next level.

```
{
    'msg': 'msgs',
    'messages': [
        {'text': '<lightgrey>
        You climb downwards.
        <lightgrey>', 'turn': 1252, 'channel': 0}
    ]
}
```

This message indicates a change of level, by parsing this message we can start a new map to save the configuration of the next level.

```
{
    'msg': 'msgs',
    'more': False,
    'messages': [
        {'text': '<green>
        Your experience leads to an increase in your attributes!
        <lightgrey>', 'turn': 1308, 'channel': 12},
        {'text': '<cyan>
        Increase (S)trength, (I)ntelligence, or (D)exterity?
        <lightgrey>', 'turn': 1308, 'channel': 2}
    ]
}
```

This message appears sometimes when the player levels up his character, the game is stopped and the player is not able to perform any more actions until he chooses one stat to update, by pressing one of the keys *'S'*, *'I'* or *'D'*. So this parse is important in order not to stop the gameplay in the middle of an experience, after pressing one of the keys, the game proceeds as usual.

```
{
    'msg': 'msgs',
    'more': True,
    'messages': [
        {'text': '<lightgrey>
        You die...
        <lightgrey>', 'turn': 1367, 'channel': 0}
    ]
}
```

This is the last message received in each game.

When the player dies the game is over, and it is needed to reset the game to continue the experience.

Map Messages

The message that carries the information about the map, however, requires a more complex parser.

Whenever the game is started, a new dungeon, with a completely different map is generated. In a run, we can get this starting state.

Figure 10.: Initial Game State

This map sends this JSON.


```
{
'msg': 'map',
'clear': True,
'player_on_level': True,
'vgrdc': {'x': 0, 'y': 0},
'cells': [
{'x': 3, 'y': -6, 'mf': 26},
{'mf': 26},
{'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 968}},
{'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 968}},
{'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 965}},
{'x': 2, 'y': -5, 'mf': 26},
{'mf': 26},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 4, 'ov': [2205]}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 4, 'ov': [2204]}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 6, 'ov': [2204]}},
{'mf': 26},
{'x': -2, 'y': -4, 'mf': 26},
{'mf': 26},
{'mf': 26},
{'x': 2, 'y': -4, 'mf': 26},
```

```
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 5}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 7}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 3}},
{'mf': 26},
{'mf': 26},
{'x': -2, 'y': -3, 'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 968}},
{'f': 1, 'mf': 5, 'g': '+', 'col': 7, 't': {'bg': 2252, 'flv': {'f': 3}}},
{'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 968}},
{'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 965}},
{'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 966}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 6, 'ov': [2207, 2208]}},
{'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 965}},
{'mf': 26},
{'mf': 26},
{'x': -2, 'y': -2, 'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 965}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 5, 'ov': [2202, 2204]}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 2, 'ov': [2204]}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 2, 'ov': [2204]}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 9, 'ov': [2204]}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 5, 'ov': [2206, 2203]}},
{'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 967}},
{'x': -2, 'y': -1, 'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 965}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 2, 'ov': [2202]}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 4}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 2}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 3}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 5, 'ov': [2206]}},
{'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 967}},
{'x': -2, 'y': 0, 'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 967}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 6, 'ov': [2202]}},
{'f': 60, 'mf': 12, 'g': '@', 'col': 87, 't': {
    'fg': 527514, 'bg': 2412, 'flv': {'f': 3, 's': 235},
    'doll': [[3417, 32], [3375, 32], [3484, 32], [3544, 32], [4144, 32], [3872, 32]],
    'mcache': None}
},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 5}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 5}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 7, 'ov': [2206]}},
```

```
{'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 968}},
{'x': -2, 'y': 1, 'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 966}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 3, 'ov': [2202]}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 4}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 6}},
{'f': 33, 'mf': 6, 'g': '$', 'col': 14, 't': {
    'fg': 916, 'base': 0, 'bg': 1048584,
    'doll': None,
    'mcache': None}
},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 5, 'ov': [2206]}},
{'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 970}},
{'x': -2, 'y': 2, 'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 968}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 4, 'ov': [2202]}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 4}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 6}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 5}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 6, 'ov': [2206]}},
{'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 967}},
{'x': -2, 'y': 3, 'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 968}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 2, 'ov': [2202]}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 4}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 2}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 3}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 3, 'ov': [2206]}},
{'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 967}},
{'x': -3, 'y': 4, 'mf': 26},
{'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 967}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 7, 'ov': [2202]}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 6}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 3}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 2}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 7, 'ov': [2206]}},
{'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 971}},
{'x': -3, 'y': 5, 'mf': 26},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 3, 'ov': [2204]}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 4, 'ov': [2203]}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 2}},
```

```
{'f': 33, 'mf': 6, 'g': ')', 'col': 4, 't': {
    'fg': 195, 'base': 0, 'bg': 4, 'doll': None,
    'mcache': None}
},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 7}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 2, 'ov': [2206]}},
{'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 965}},
{'x': -4, 'y': 6, 'mf': 26},
{'mf': 26},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 6, 'ov': [2208]}},
{'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 967}},
{'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 965}},
{'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 971}},
{'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 966}},
{'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 965}},
{'f': 7, 'mf': 2, 'g': '#', 'col': 6, 't': {'bg': 968}},
{'x': -4, 'y': 7, 'mf': 26},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 3}},
{'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 6, 'ov': [2205]}},
{'mf': 26},
{'x': -4, 'y': 8, 'mf': 26},
{'mf': 26},
{'mf': 26},
{'mf': 26}
]
}
```


Although the first state is very simple, the message received is not that simple.

There are 2 main fields in this message, the *vgrdc*, that indicates the player position on the map, and the *cells* field, which carries all the current map information.

To parse the cells of the map we start by looking at the first cell, this cell besides carrying information about one cell also carries its coordinates in the map.

For example the cell *'x': 3, 'y': -6, 'mf': 26*, is a cell that as coordinates x:3 and y:-6. The x coordinates grow from left to right, and the y coordinates grow from top to bottom, so knowing that the player is currently at position x:0, y:0, counting 3 cells to the right and 6 up gives us the cell in the map.

The next cell does not have any coordinates on it, but we find out the coordinates by adding one to the x coordinate, so the next cell will be at x:4 y:-6, and we can continue to increase one in the x coordinate to find the rest of the coordinates of the next cells. This

system is used by the game to avoid data redundancy. But this system only lets us find one line of the map. Among the cells of the map there are some other cells that also contain coordinates, if a cell contains coordinates these are the new base coordinates and the next cell counting from there use the same system as above, this allows us to find other lines, and to avoid getting lots of cells with empty data in the case where a line only as a few cells separated by a big dark space. When this happens although the cells are in the same line, the game sends two messages with coordinates, because this is less redundant than to send several cells with no data in it. Because of this avoiding redundancy format, it is very important to read all cells in order, because otherwise, it would be impossible to know the position of a cell.

Now that we can find what position we are in, it is time to parse each cell.

Starting at the first the only information received is *'mf': 26*. This represents a dark cell, a cell that is not visible to the player yet. The rest of the relevant information can be found in the *'g'* field. The '#' symbol means a wall, the '.' is floor, '+' is a door, the '@' is the player, '£' is money, and '(' is an item.

Knowing all of this we are ready to parse all the information and group it in a bi-dimensional array.

This array starts as an empty one, and whenever it is needed it is added a line or column before of after the positions that are already on the array, by doing so we guarantee to keep the map format.

After parsing that message the resulting map is represented by a 20x20 map, where the spaces that are still to discover are represented by the char ' '(space).

```
          ###
         ...
        ...
 #+###.#
 #.....#
 #.....#
 #.@...#
 #...$.#
 #.....#
 #.....#
 #.....#
 ...)..#
 .######
..
```

After receiving the first map message the following ones are not so big, they only carry the information that has changed in the map, so if the player in the previous state chooses to go up, like this.

Figure 11.: Game State after 1 action

The message received is this.


```
{
'msg': 'map',
'vgrdc': {'x': 0, 'y': -1},
'cells': [
{'x': 3, 'y': -6, 'mf': 26},
{'mf': 26},
{'col': 8, 't': {'bg': 263112}},
{'x': 8, 'y': -6, 'mf': 26},
{'x': 2, 'y': -5, 'mf': 26},
{'mf': 26},
{'col': 8, 't': {'bg': 262148}},
{'x': 7, 'y': -5, 'f': 33, 'mf': 1, 'g': '.', 'col': 7, 't': {'bg': 7, 'ov': [2204]}},
{'mf': 26},
{'x': -2, 'y': -4, 'mf': 26},
{'mf': 26},
{'mf': 26},
{'x': 2, 'y': -4, 'mf': 26},
{'col': 8, 't': {'bg': 262149}},
{'x': 6, 'y': -4, 'mf': 26},
{'mf': 26},
{'mf': 26},
{'x': 5, 'y': -3, 'mf': 26},
```

```
{'mf': 26},
{'x': 0, 'y': -1, 'g': '@', 'col': 87, 't': {
    'fg': 527514,
    'doll': [[3417, 32],[3375, 32],[3484, 32],
             [3544, 32],[4144, 32],[3872, 32]],
'mcache': None}
},
{'x': 0, 'y': 0, 'g': '<', 'col': 9, 't': {'fg': 0, 'doll': None, 'mcache': None}},
{'x': -3, 'y': 4, 'mf': 26},
{'x': -3, 'y': 5, 'mf': 26},
{'x': -4, 'y': 6, 'mf': 26},
{'mf': 26},
{'x': -4, 'y': 7, 'mf': 26},
{'col': 8, 't': {'bg': 262147}},
{'col': 8, 't': {'bg': 262150}},
{'mf': 26},
{'x': -4, 'y': 8, 'mf': 26},
{'mf': 26},
{'mf': 26},
{'mf': 26}
]
}
```


In this state the player left the entrance of the dungeon, therefore, it is now shown on the map, it is represented by the symbol of $'<'$.

Updating the old state with this new one leads to this map.

```
        ###
       ....
      ...
 #+###.#
 #.....#
 #.@...#
 #.<...#
 #...$.#
 #.....#
 #.....#
 #.....#
 ...)..#
 .######
..
```

After parsing the entire map, every char is used as its index on the ASCII table, allowing us to use the map as a numpy array, that will be needed in the following tasks.

Enemies on the map

Whenever there is an enemy on the map, the game sends some information about it. Besides the character that represents the monster, it also sends some information about the monster. From this information we will focus on the name and threat. With the name of the monster it is possible to find more information about it. And the threat level is a good indicator of how dangerous a monster is and, if this will or will not be a successful fight.

```
{
    'f': 33,
    'mon': {
        'id': 4,
        'name': 'dart slug',
        'plural': 'dart slugs',
        'type': 623,
        'typedata': {'avghp': 10},
        'att': 0,
        'btype': 623,
        'threat': 1
    },
    'mf': 1,
    'g': 'w',
    'col': 3,
    't': {'fg': 2183, 'bg': 4, 'doll': [[2183, 32]], 'mcache': None}
}
```

```
{
    'f': 33,
    'mon': {
        'id': 1,
        'name': 'kobold',
        'plural': 'kobolds',
        'type': 187,
        'typedata': {'avghp': 3},
        'att': 0,
        'btype': 187,
        'threat': 1
    },
    'mf': 1,
    'g': 'K',
    'col': 6,
    't': {'fg': 2856, 'bg': 2, 'doll': [[2856, 32]], 'mcache': None}
}
```


These two monsters are level 1 monsters, which means that they are weak enemies and can be easily beaten.

```
{
    'f': 33,
    'mon': {
        'id': 63,
        'name': 'Jessica',
        'plural': 'Jessica',
        'type': 426,
        'typedata': {'avghp': 9},
        'att': 0,
        'btype': 426,
        'threat': 2,
        'clientid': 63
    },
    'mf': 1,
    'g': '@',
    'col': 7,
    't': {'fg': 2686, 'bg': 7, 'doll': [[2686, 32]], 'mcache': None}
}
```

```
{
    'f': 33,
    'mon': {
        'id': 94,
        'name': 'Pikel',
        'plural': 'Pikel',
        'type': 476,
        'typedata': {'avghp': 39},
        'att': 0,
        'btype': 476,
        'threat': 3,
        'clientid': 94
    },
    'mf': 1,
    'g': 'K',
    'col': 1,
    't': {'fg': 2855, 'bg': 7, 'doll': [[2855, 32]], 'mcache': None, 'ov': [2203]}
}
```


These last two monsters are a bit stronger than those first ones. But in the map representation of the game, the monster *Pikel* shares the same representation as a *kobold*, because of this, it is important to analyze the monster threat level and not his representation since the same character represents different levels of danger in battle.

4.2.4 *Modular System*

As said before, the system is divided into 4 modules, to be as abstract and simple to use in other projects, this was built to be modular. Although every part of the project is essential to a successful run, every part can be replaced if needed.

First of all, the system includes a connection to the game that runs on as a multi-process, this package is specific to this project, and, it can be deleted if the project is used to reach another goal.

If trying to reach another goal, the environment will be different, and therefore, all the environment package can be replaced, the user just needs to make sure that the initialization function of the environment is well built, and that this offers 3 main functions, the *step* function that will be called every time the agent needs to make an action on the environment, the *getState* function that returns the current state of the environment, and the *reset*

function that resets the environment to his initial state. By remaking the state, all the classes integrated with the *Environment* package will no longer be needed, because those classes were used to communicate with the game connection to update the environment state. The network itself is built using PyTorch, and it can also be replaced by a new one if the input layer of the network still has the same number of inputs, and the output layer the same number of outputs. This was important to test several network during this project. The last thing that can be tuned is the *Utils* class in the *Reinforcement* package. This class can be used to tune some hyperparameters, such as Batch size, learning rate, input and output size of the network, number of episodes, the size of the memory of the network and some other parameters about decisions to make in each episode.

4.2.5 *Libraries*

Time

From this library it is used the function *sleep* that stops a process for a certain amount of time, this allows us to wait for messages that will arrive from the game, and by waiting this time, we make sure that the state is always updated.

Pyautogui

This library provides functions that simulate the pressing of keys, it is used as a way to pass the action that decided to be taken to the game by pressing the correspondent key.

Namedtuple

Each step of the game is saved as an experience, this experience is composed of the current state, the action made on that state, the reward that resulted from that state and the next state. Each experience is saved under the structure of a namedtuple.

```
Experience = namedtuple(
'Experience',
('state', 'action', 'next_state', 'reward')
)
```

Random

To avoid data correlation, all the experiences are saved, and at each step one of them is selected randomly to be used for loss calculation.

Math

This class is used to use some math operations such as exponential.

Socket

In the project, socket communication is used to pass information between the game connection and the environment.

Json

All the information sent from the game is sent in the JSON format, so, to read these messages we need to import this library.

Re

Re stands for regular expressions, and it is used to split messages that are sent by the game to read each field in the message.

Os

To save log files of the network progression, and the current network weights and bias, it is needed to explore the operative system to find if it is possible to save the file on the needed directory. This library provides functions that help us with that task.

Numpy

Numpy is used to make our environment state cleaner before we passed it to a torch tensor.

Multiprocess

This provides the function *Process* needed to run a new process.

Threading

Similar to the above, with this library we can handle threads.

4.2.6 *Extensions*

When the game is over, it is needed to reset it. Since we are using the web tiles platform to run the game, to restart it we need to click on a button on the website to restart.

To avoid using the mouse to select this item we used a browser extension that passes through every link visible in the current page and sets a key to every one of them, since the

lobby page is always the same, the key given to the link that will start the game once again is always the same, and therefore we can simply simulate a press of the key to restart the game.

The extension used for this effect was *Vim Vixen*, which implements the use of keyboard shortcuts like in *vim* to help browser navigation.

4.2.7 *The algorithm*

The process starts by checking if there is a network saved locally if so, load that data to the current network. After this is loaded, the networks are created and all the data initialized and we are ready to start. The program runs in a loop where each cycle is an episode of the experience. At the beginning of the episode we start by getting the current state, given this state, the agent runs the network to find the best action to execute, this action can be achieved via exploration or exploitation.

The exploration is made when the agent does not know the environment, so it is needed to explore the actions available to find the rewards that will come from those actions, exploitation is applied when the agent already has some understanding of the environment and can make a decision by itself.

After the decision is taken for the action to follow, the action is applied to the environment, and this will result in a new state and a reward.

We are now ready to save our experience, by saving our old state, the action, the reward and the new state in our memory. After that, we test to see if there are enough samples in the memory to test a sample of experiences. If there are enough samples we get n samples, after that we get the current QValues by running our policy network and get the next QValues by running the target network. After that, we compare the two QValues to calculate the loss of the network. This will lead to the policy network trying to reach the target network and getting closer at every step, therefore improving. To keep it improving, we will update the weights of the target network with the weights of the policy network. This leads to an evolution of the target network, keeping the distance to the policy network, that will now improve its weights to reach the target network again.

When the number of episodes that were set to run end, we finish our experience and leave the cycle.

4.2.8 *The networks*

To first test the algorithm we used a simple test network that is not contemplated in these results. It was our network zero, and it served as a starting point to explore leaks and bugs in the algorithm and fix them.

Layer Input	Layer Output
Input	800
ReLU Activation Function	
800	400
ReLU Activation Function	
400	output

Table 1.: Network Zero Architecture.

We have used the input and output size from our *utils* file configuration to make changes in our input and output size whenever it was needed, automatically adapting the network to it. The final input size was 1610, and the output size has the size of the actions implemented, 13.

After that, we were ready to begin with the real testing of the network, to fully test the problem we started with a simple network and adding more layers from there.

Layer Input	Layer Output
Input	400
Sigmoid Activation Function	
400	100
Sigmoid Activation Function	
100	output

Table 2.: Network One Architecture.

Layer Input	Layer Output
Input	800
Sigmoid Activation Function	
800	400
Sigmoid Activation Function	
400	200
Sigmoid Activation Function	
200	100
Sigmoid Activation Function	
100	output

Table 3.: Network Two Architecture.

Layer Input	Layer Output
Input	900
Sigmoid Activation Function	
900	600
Sigmoid Activation Function	
600	300
Sigmoid Activation Function	
300	180
Sigmoid Activation Function	
180	60
Sigmoid Activation Function	
60	output

Table 4.: Network Three Architecture.

Layer Input	Layer Output
Input	2000
Sigmoid Activation Function	
2000	2500
Sigmoid Activation Function	
2500	3000
Sigmoid Activation Function	
3000	2500
Sigmoid Activation Function	
2500	2000
Sigmoid Activation Function	
2000	1500
Sigmoid Activation Function	
1500	1000
Sigmoid Activation Function	
1000	500
Sigmoid Activation Function	
500	250
Sigmoid Activation Function	
250	output

Table 5.: Network Four Architecture.

4.3 OUTCOMES

During the process of experimentation we were able to find some signs of evolution in the network, and some signs of learning.

To have good results, besides the base algorithm and good data. We need a good reward function, a NN that fits the problem and a lot of computational power.

During our tests, our main indicator of learning was the loss evolution, but this can be misleading, although the loss is decreasing the problem may not be going in the right direction, if the reward function is not adequate to our problem, our loss will decrease, but the solution to the problem will not be reached.

These tests were made to find a solution that reaches a good reward and a good loss evolution. Although the algorithm is well implemented, the results did not achieve the goal that has set for this project.

We can find some tests with some good loss evolution, but the reward was not optimized and our character ended up walking into walls and not being effective in passing the level.

After updating our reward function, our loss did not reach the levels that we were looking for, it did not stabilize. This may be due to an unfit neural network, or some bad initial hyperparameters.

4.4 SUMMARY

This project required programming skills from different areas, from simple parsing, to multi-threading, and obviously, artificial intelligence. So, given this, this project revealed itself as a more complex problem than what was expected.

The goal of achieving a modular system was reached, and therefore our algorithm can be used in other projects. The goal of implementing the algorithm and communicating with the game was also reached, and the network was able to play the game by itself.

But although the communication routes were stable and the data well parsed, the character played by the network did not go very far in the game, leaving this goal unchecked.

To achieve it several tests were made. Tests with several networks and several reward functions, but, due to our limit in time and computational power, we were unable to reach good results.

5

CASE STUDIES / EXPERIMENTS

5.1 EXPERIMENT SETUP

To test several architectures at once, a script was built that ran several test cases with different networks. The script starts by setting up the side threads that will communicate with the game and update our data.

In order to perform our simulation we need to send the actions to the game. To write the actions a python library that simulates keyboard input was used, but for the keys to affect the game, it is needed to select the game tab before starting. Although the *vim vixen* extension is used in the project, we need to disable it, because otherwise, some game commands can interfere with the extension.

After the extension is disabled, the script is running and the game tab is selected, the experiment is ready to start.

5.2 EXPERIMENT STATES AND ADJUSTS

In this section, we are going to talk about our process of evolution, and the alternatives we tried to improve during this project in order to obtain better results.

The graphs presented here are representative of mean values calculated at every twenty or fifty steps of the game, according to each situation in order to be easier to read the results. For more detailed results, see the appendix chapter.

5.2.1 *Original experiment - First State*

Our experiments started up from a state where we tested a reward function that only checks the level progression of the player, this means, when the player kills an enemy, it gains experiment, and by doing so it receives a reward accordingly.

In this state although there was a loss decrease, the player did not reach his goal, the network was good at deciding moves to walk around the map, explore and kill enemies,

but it failed when it needed to go to the next level, and when the map was fully explored the player would just walk around in the map, leading to a lack of food and eventually died by starvation.

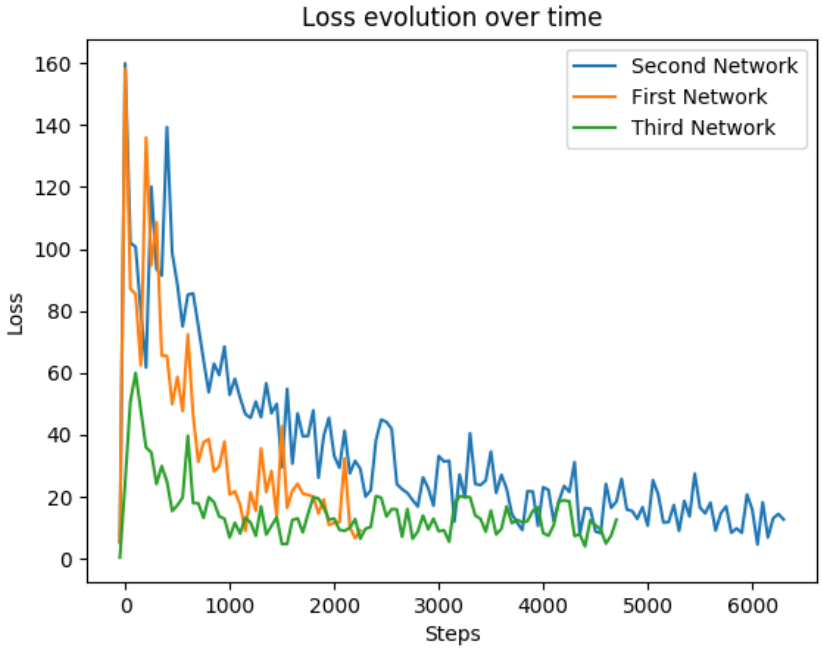
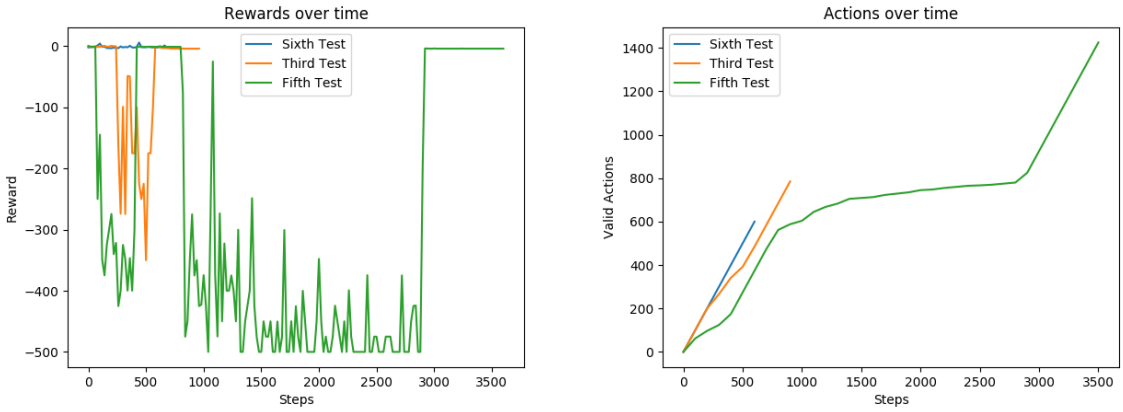


Figure 12.: State one loss evolution.

To speed up the player we started tweaking the reward function, analyzing the results with our second network. In our third test, we added a negative reward to try to fix our main problem: starving.

To do it, we added a very negative reward to some behaviors, like trying to eat when it was not needed, and making useless moves, to these two behaviours, we have given a reward of negative five hundred. This is the case that we can see in our third and fifth test.



(a) Rewards given in state one.

(b) Valid actions made in state one.

Figure 13.: These tests belong to state one, network two.

Since this leads to a very negative reward all the time, in our sixth test we have decided to change our reward function once again.

In this test, we replaced our negative rewards with smaller ones, instead of negative five hundred we passed down to negative one when the player did not made valid moves or ate out of is time, this leads to a balancing with the small positive rewards or exploring the map, and the result were much more favorable, but even after this readjustment, our player is not able to bypass the problem, and our loss had started to give us very high values. So, since the reward function changed our problem so much, to better solve it, we dedicated our next experiments to improve our reward function.

5.2.2 *Updating reward function - Second State*

This second set of experiments was built to try to change our reward function.

In our last state, adding very high negative rewards for some actions did not lead to good results, so the first thing we did was change our negative reward from negative five hundred to one hundred in the case of invalid moves. Because this was still our main problem to focus on.

After that, we also added a few more factors to the equation. A factor to control our players damage in each step was added in order for it to start fighting more carefully, thus, avoiding losing health points, and trying to regain them when the fight was over, which is a skill needed.

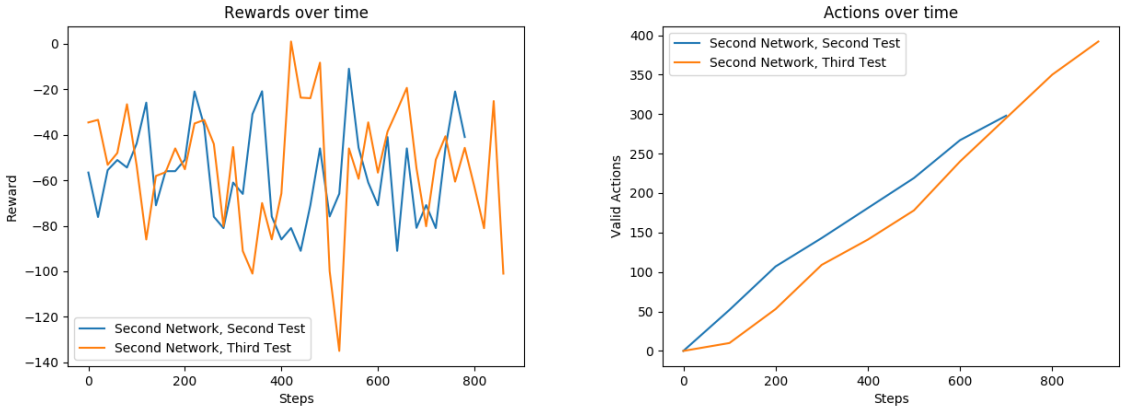
To control our hunger level it was said that at every step of the game, our player would receive a negative reward if it was hungry.

We also added a factor to improve map exploration, to avoid just making a random walker, because this leads to starving.

To implement the map exploration factor, we added one value of positive reward at every step for each tile of the game that had changed since the last step.

To encourage our player to find more ladders down, we added the reward of one hundred positive points when the player goes down a level.

This lead to this new reward evolution.



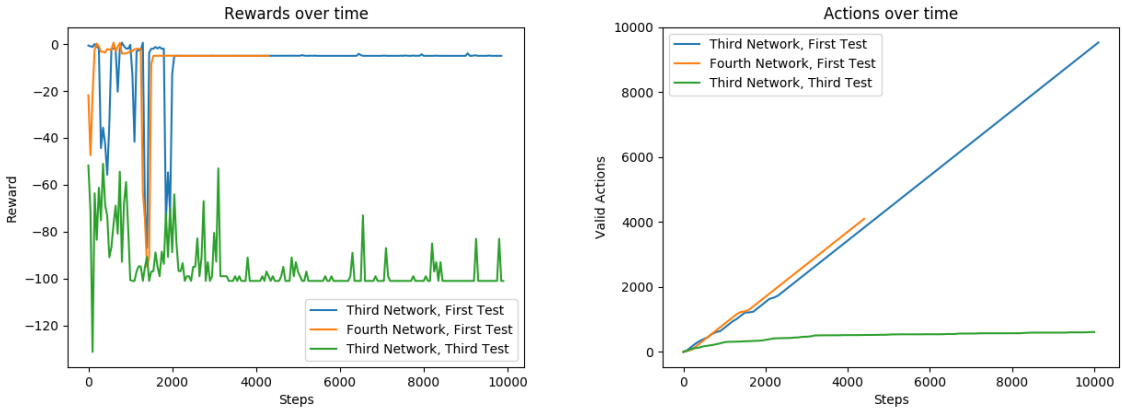
(a) Rewards given in state two.

(b) Valid actions made in state two.

Figure 14.: State two, first reward function.

Quickly we realized that adding a reward for being hungry was not a good policy, the player was being very badly rewarded every step, not because of its action, but because of its current state that was caused by a previous action. So we started by removing this factor and replacing with a negative reward when our player would eat if not necessary.

The changes made lead to the player trying to execute some commands too many times, like going up and down the ladders in order to find its reward, this lead to a lot of invalid movements, resulting in very negative rewards. To solve this we removed our high negative rewards, except the reward for invalid movement.



(a) Rewards given in state two.

(b) Valid actions made in state two.

Figure 15.: State two, second reward function.

In this experiment, we got more valid actions, but this did not lead to a more favorable result since the loss evolution did not stabilize at a lower level.

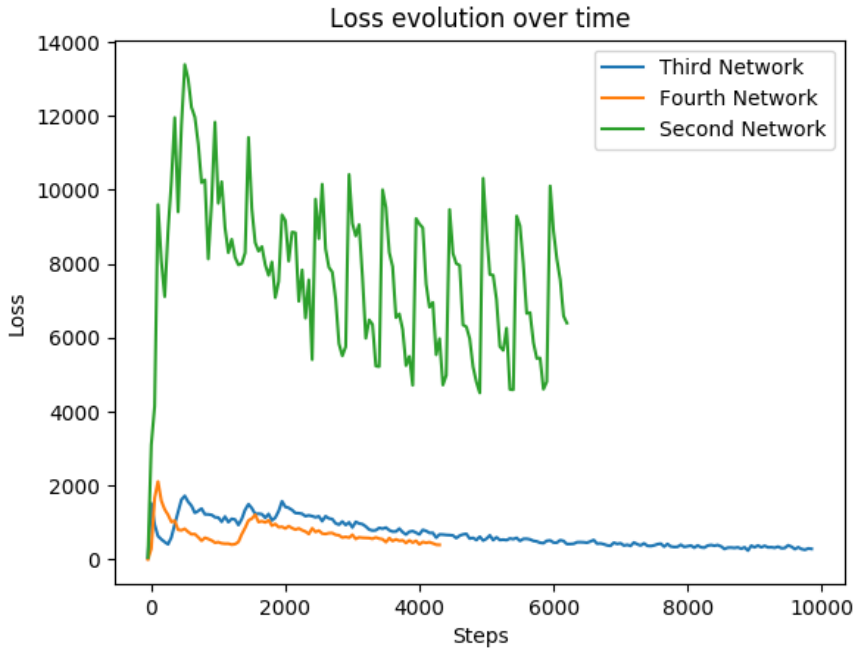


Figure 16.: State two loss evolution.

Since we have already tested several networks and updated our rewards function, it was needed to look at our hyperparameters to find some correlation with the results.

5.2.3 *Epsilon greedy strategy - Third State*

As we now know, the network takes two phases, the first one is the exploration phase when the network does not know the problem and it needs to explore. If the exploration is not wide enough, when it gets to the phase where the network needs to make decisions, it may not behave as expected.

In this test and in the followings, all the conditions were the same for the three tested networks, and the only different between them, was their own architecture, which is described in detail in chapter four.

In this test, we have increased the exploration phase.

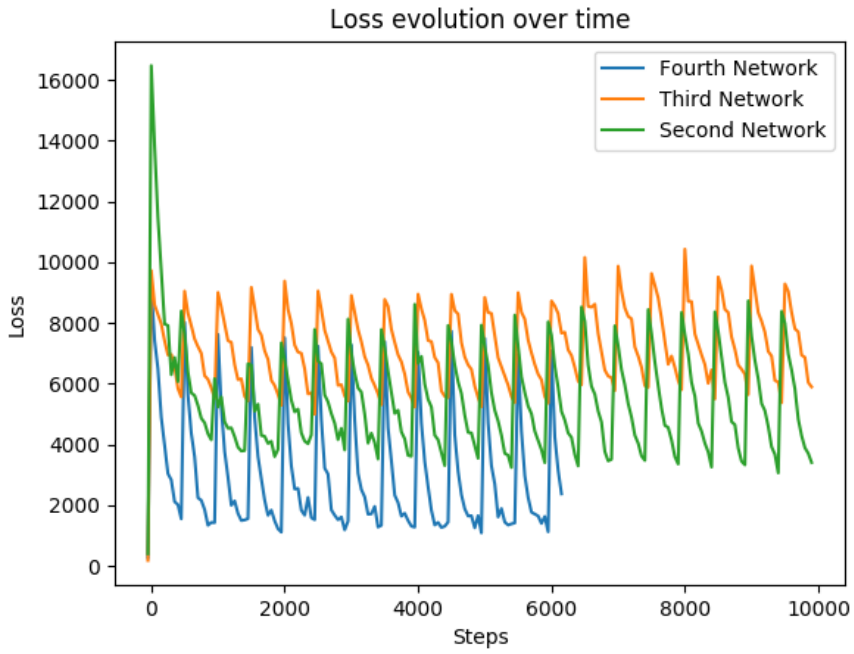
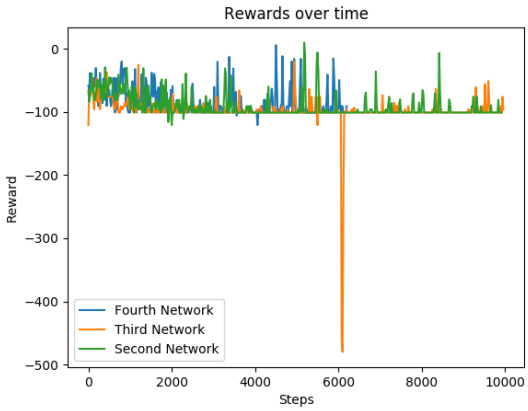
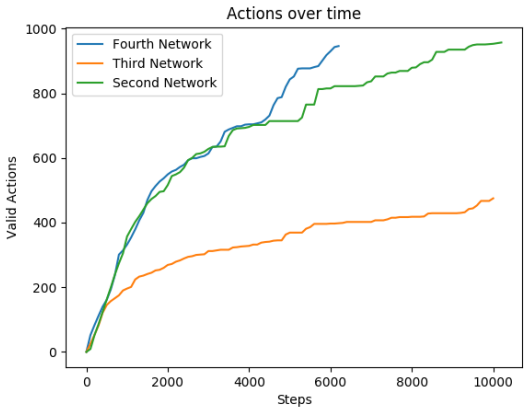


Figure 17.: State three loss evolution.

Our loss is now, not stabling, and therefore we can assume that our network is not being able to encounter a solution to our problem.



(a) Rewards given in state three.



(b) Valid actions made in state three.

The reward level is also very low, this is mainly because our character is not making valid moves. Since our loss is now not stable, this leads to our program making plenty of invalid moves. This leads us to conclude that increasing our exploration rate did not help, we can try to adjust other hyperparameters. In our case, the problem can be due to a too fast learning rate.

5.2.4 *Learning Rate Reduction - Forth and Fifth State*

Adjusting our learning rate leads to a network that learns at a slower pace. This means that every step of the game is less meaningful to change our network weights, and therefore, our evolution is slower.

In this experiment we trade our learning rate of 0.001 to 0.0001.

Slowing down our evolution leads to a more stable result in terms of loss. But unfortunately, our loss had stabilized in at very high values, this means that at every step our network is far away from the correct decision.

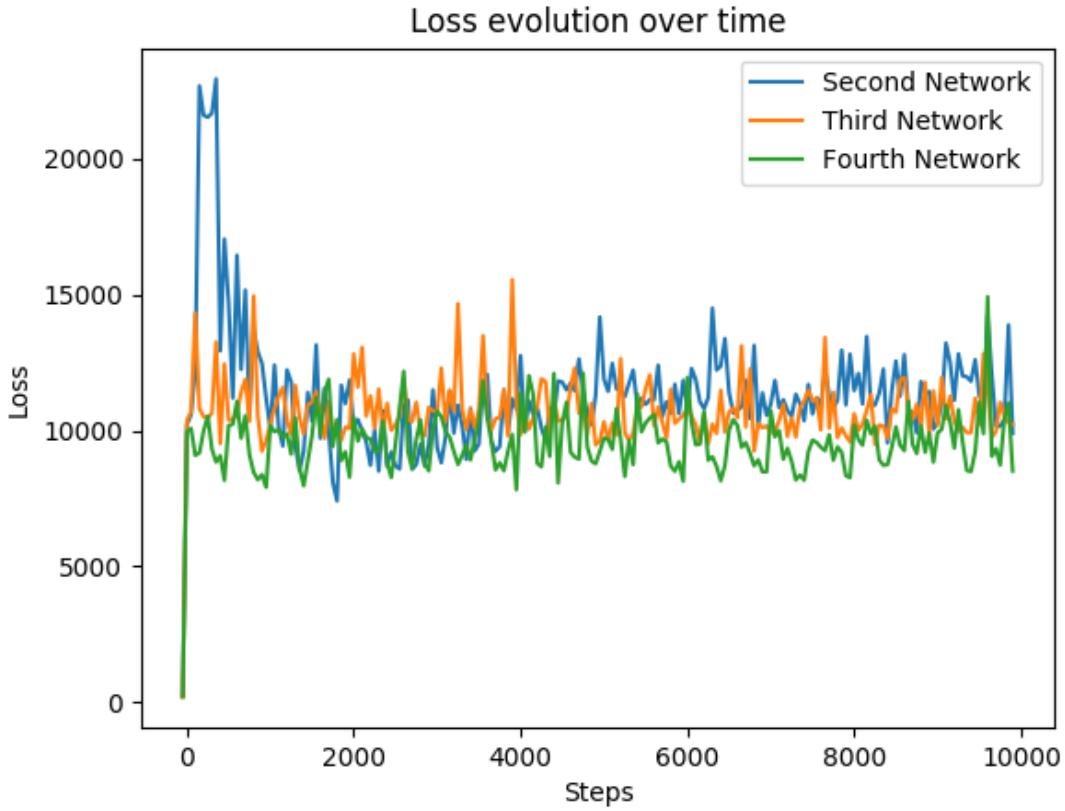
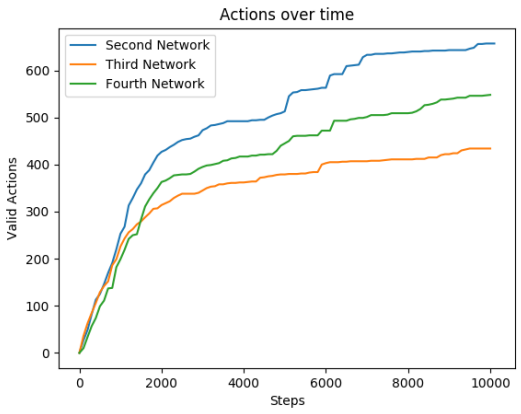


Figure 19.: State four loss evolution.

We can relate this very high loss to plenty of invalid moves during our experiment.

(a) Rewards given in state four.



(b) Valid actions made in state four.

In our last experiment, we decided to tweak a bit more with our learning rate and changed it to 0.00001. Our loss decreased compared to our last state but is still very high.

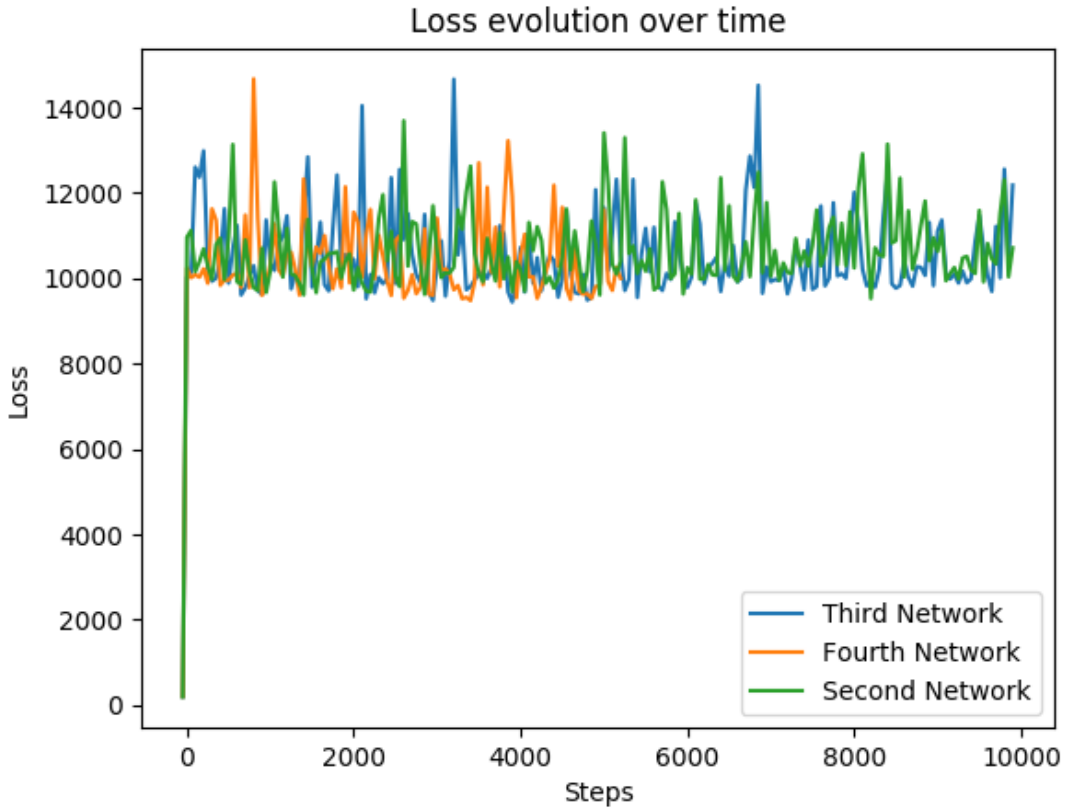
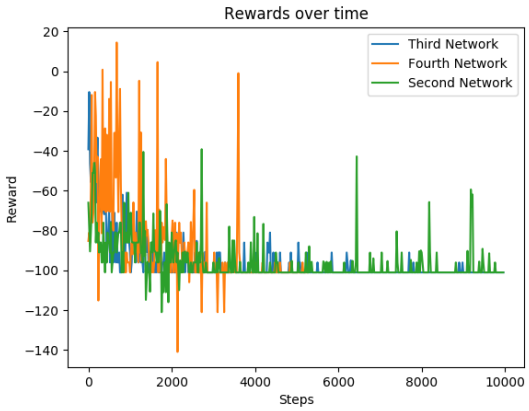
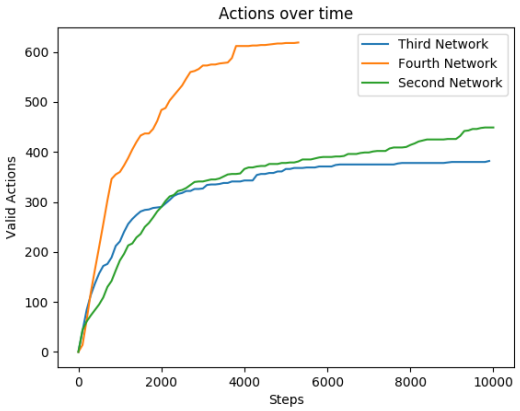


Figure 21.: State five loss evolution.



(a) Rewards given in state five.



(b) Valid actions made in state five.

5.3 RESULTS

During all of our tests, our results were not what we first expected. In most of our tests our bot only died on level one, and in some of them reached level two or tree.

To improve this result, our last test was a simplification of the problem. Since our cause of death was starvation, this key was removed from the output of the network and made automatic, every time the level of hunger of the player drops to a certain level, the eat command was automatically used.

The next simplification was to get to the next level automatically, since this process was when the player got lost most of the times, so to achieve this we waited until the player had explored all the level, and then used a command from the game to send it to the next level, this also removed two keys from the network output, going up, and downstairs.

After all these optimizations our network was left with only ten keys, all the directions, including diagonals and resting in place, and auto-explore.

This leads the player from reaching level three to reach level four of the game and level six of experiment. This was our best result, although it was not as good as expected at the beginning of the project.

5.4 DISCUSSION

After reaching the end of the project, we have now realized our goals were not realistic for the time of the project. Reaching a good artificial intelligence takes time, and lots of processing units, since all the work ran on CPU and it was needed to wait at every step of the game for the game feedback, this made the learning process a lot longer.

Passing the entire game would require a deeper understanding of the game to better understand the environment and reaching a better reward function.

This problem has proven itself as not trivial. The huge amount of states, possible actions, enemies, and even passive choices, like our character race and background and equipment, makes it a very difficult problem.

During our training, we have made five major changes in our problem, and this leads to our five states. In our first state, our reward function was really simple, and therefore, in a complex problem like this, it was normal not to see stunning results. But this was the state where our network achieved lower loss stability. This means that our reward function was fit to our networks, and we could use it as a solution, unfortunately, this was only a solution to explore and make a sort of random walking on the map. And consequently, this leads to plenty of deaths by starvation.

In our second state, our reward function was updated, and in most of our tests, our results presented plenty of valid movements, but, since our loss did not find stable values,

this means that our problem was in another part of the project, and although tunning more our reward function could lead to better results, they would never be good results.

This lead to changing our hyperparameters. And this is made in our third, fourth and fifth states, where we change our exploration rate and learning rate. Unfortunately, we did not find a good combination of them all, and therefore, our final results were not the best.

Since our network could do some work in map exploration, making some tasks automatic was a good improvement, and allowed our network to focus on the task of exploring the map, leading to better results.

5.5 SUMMARY

As the main result, we focused on building an architecture that can be used in future projects.

This will allow further research to be made in the deep reinforcement learning area of the project without the need to worry with all of the complex interpreting problems that the game carries with him, such as reading our game environment and making sure it is updated at every step of the problem.

Considering the time length of the project, and due to our main part of the project was trying to communicate with the game, making sure it was failure tolerable, and making sure all the data was updated in time because this was an essential step to train the network with real-time data. And since we could only start our tests when this process was over, we cannot consider our results as very bad. But since the algorithm was well implemented, better results could be achieved, unfortunately, due to lack of experiment in neural networks hyperparameters adjusting and architecture planning, the project had lead to these results.

Given this, we can present as our main result our architecture to solve any problem of reinforcement learning and our study for others to improve from it.

All the code made for this project can be found here: https://github.com/simbs38/tese

6

CONCLUSION

6.1 CONCLUSIONS

After all the work is done, we can say that although artificial intelligence area is growing, is still an area with a lot to discover and improve. Our main adversity was the creation of the neural network, for someone without years of experience in the field, is very difficult to come up with a new network and find out the results that it will bring, to decide if a new layer should be added or removed, and the configuration of each layer is not a trivial problem to solve, as well as the activation function for each layer.

Even the input of what is relevant to the network is not so linear, although we only have so much information, the way it is passed to the network matters and some adjustments may help to solve this problem.

A third set of options that matters to the problem is the hyperparameters adjustment. To adjust hyperparameters it is also required some experience and trial and error, but there are already many clues online that can help users to read our loss results and improve our hyperparameters. However, spending too much time adjusting the hyperparameters of a network that is not fit to our problem may be counter-productive, some networks just do not have a large enough complexity to solve our problem, and adjusting hyperparameters will not make us reach our goal.

Although there are many aspects of this field to discover, there is a huge community working on them, making some aspects really straight forward, libraries like the gym openai toolkit help us to dive in the problems and easily get some instant experience in more simple problems, that helps us to have some base knowledge to work in this area. Given this, anyone can work with artificial intelligence, and eventually, this makes more people do so.

As a result of this work, it presents a framework to solve our problem, and deals with all the parsing and data handling that the problem requires. Together with this, it is also presented as an implementation for deep reinforcement learning algorithms, and since it was built to be modular, it can be used on some other projects to solve other problems.

However, our main goals have not been reached, our goal was to play the game in some sort of success, which we can not say that was obtained. Our character had explored some levels of the game, and we could observe some behaviors that represented real learning, for example, when the reward function had punishment for losing health points, our character started to fight in a more conscious way, it would take advantage of the terrain and find tunnels or corners in order to minimize the damage taken. But this was not enough, and our complex problem was not solved after all.

Along with the results from the project, this was also good to develop some knowledge in some areas besides artificial intelligence. From multi-threading and multiprocessing systems to failure tolerant systems in the area of distributed systems, to data parsing and scripting, not ignoring our main focus and the experience obtained in the main area of the project, and deep reinforcement learning.

6.2 Prospect for future work

Since we failed to achieve our main results, the future work for this project passes through that. To find better results there are two main things to optimize in the project.

The first one is the reward function, this function has to be tweeted and tested in different cases in order to be optimized, since we are working on a game, and therefore, a controlled environment with access to all of the state of the game.

The second is the network architecture. Building a network from scratch is difficult, and takes many tries to get it right for the problem in question. Our main limitation to test our networks is computer power, to test more networks in an efficient way. As future work, in order to find the network that most fits the problem, a neural architecture search algorithm can be used. Automatic neural architecture design has shown its potential in discovering power-ful neural network architectures[8]. The problem with this is once again computer power since the most basic algorithms simply run lots of neural networks and then say what is the best one, but to test them, it is necessary to run them all and find results for them all. In this very recent area of neural architecture search there are already some improvements, such as reusing the weights of a network to initialize another, and by doing so, the process of training the second network can be much faster, but these algorithms still consume a very large amount of computer power, and because of that, it was not possible to implement such a strategy in this project.

BIBLIOGRAPHY

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[2] Horace B Barlow. Unsupervised learning. *Neural computation*, 1(3):295–311, 1989.

[3] DCSS Devteam. Dungeon Crawl. http://crawl.chaosforge.org/Dungeon_Crawl, 2016.

[4] DCSS Devteam. Dungeon Crawl Walktrough. http://crawl.chaosforge.org/Walkthrough, 2018.

[5] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[6] Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, 160:3–24, 2007.

[7] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[8] Renqian Luo, Fei Tian, Tao Qin, Enhong Chen, and Tie-Yan Liu. Neural architecture optimization. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 7816–7827. Curran Associates, Inc., 2018.

[9] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[11] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. Pytorch. *Computer software. Vers. 0.3*, 1, 2017.

[12] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[13] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[14] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[15] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[16] Aaron Van den Oord, Sander Dieleman, and Benjamin Schrauwen. Deep content-based music recommendation. In *Advances in neural information processing systems*, pages 2643–2651, 2013.

Appendices

A

APPENDIX

A.1 FIRST STATE EXPERIENCES

A.1.1 *First Network Results*

First Test

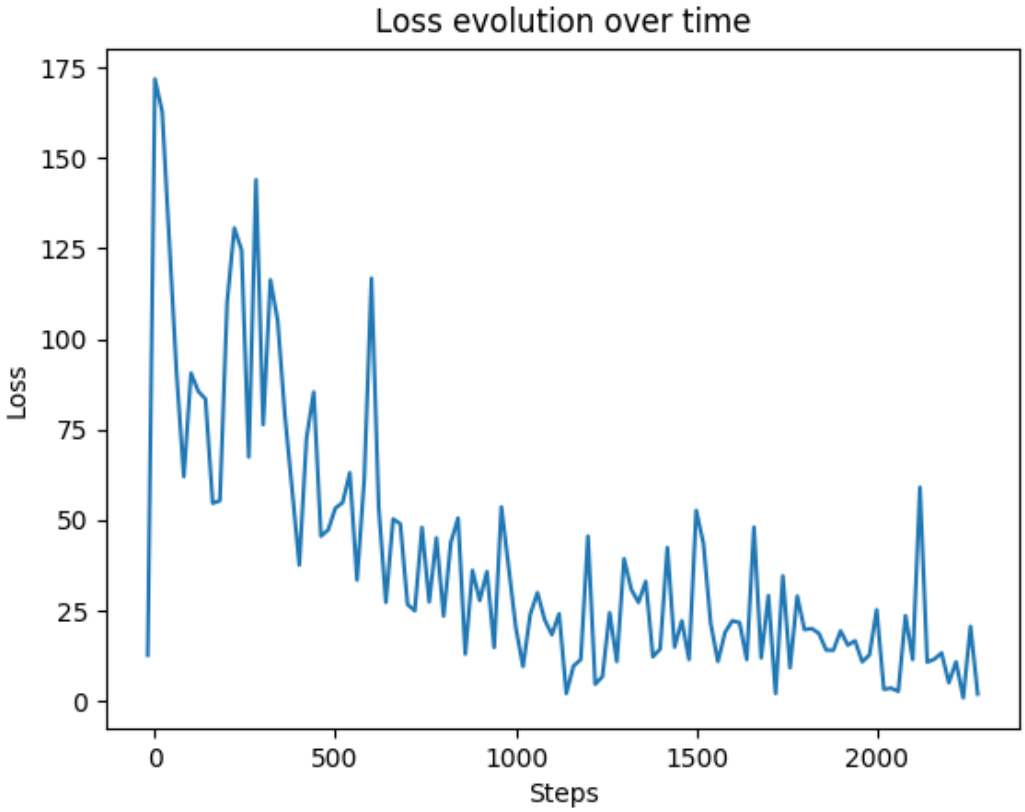


Figure 23.: Loss evolution of first network, first state, first test.

Second Test

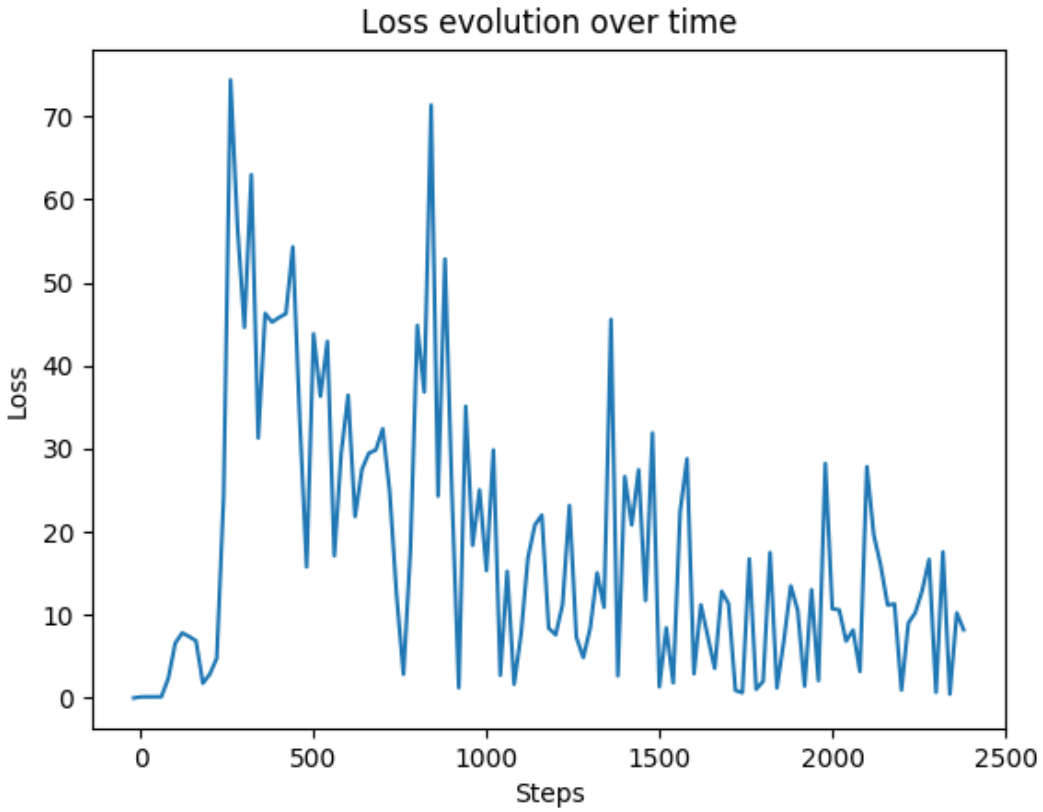


Figure 24.: Loss evolution of first network, first state, second test.

A.1.2 *Second Network Results*

First Test

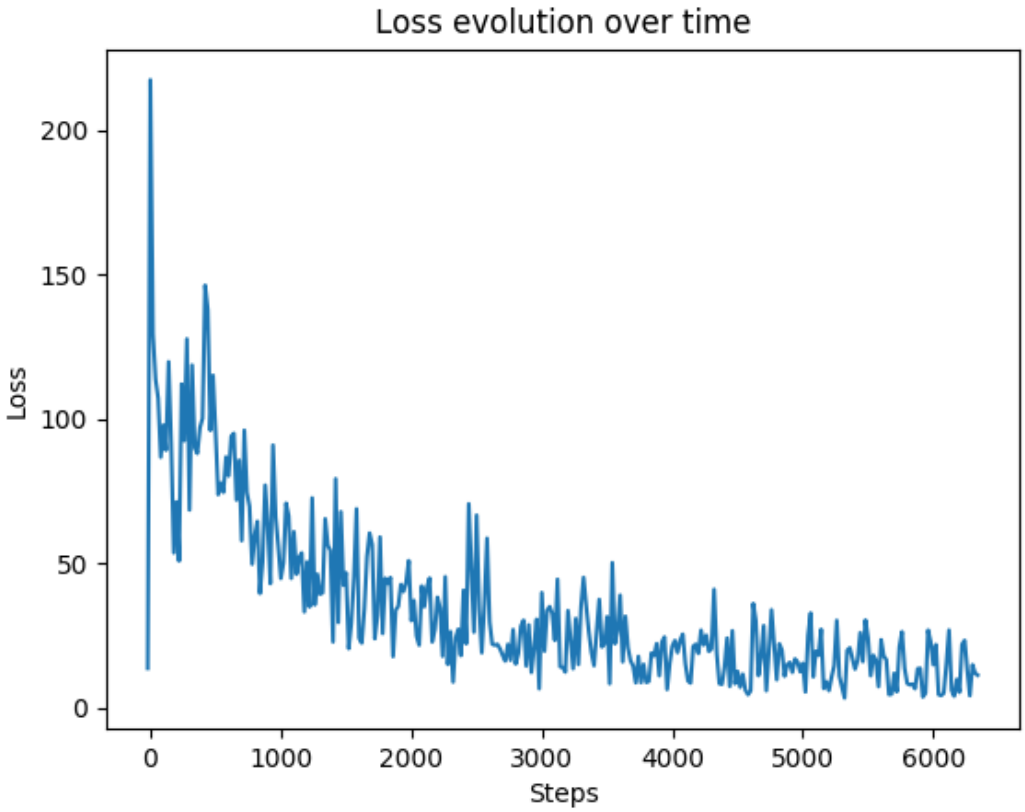


Figure 25.: Loss evolution of second network, first state, first test.

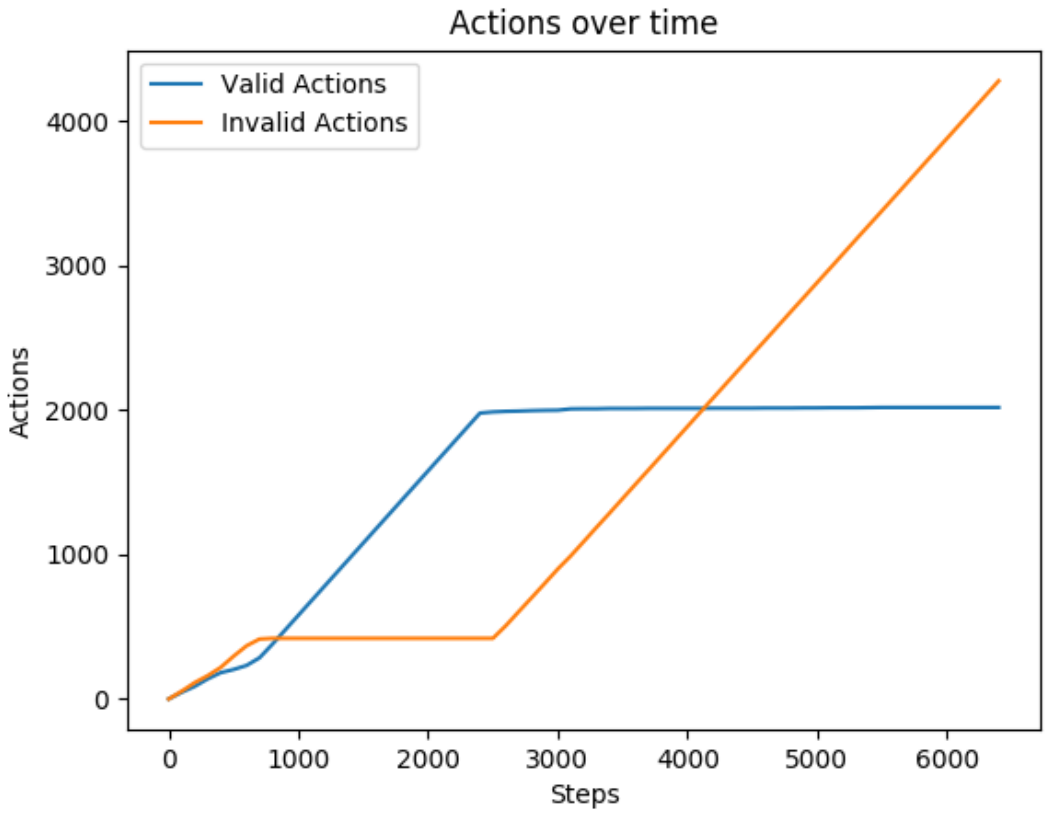


Figure 26.: Actions analysis of second network, first state, first test.

Second Test

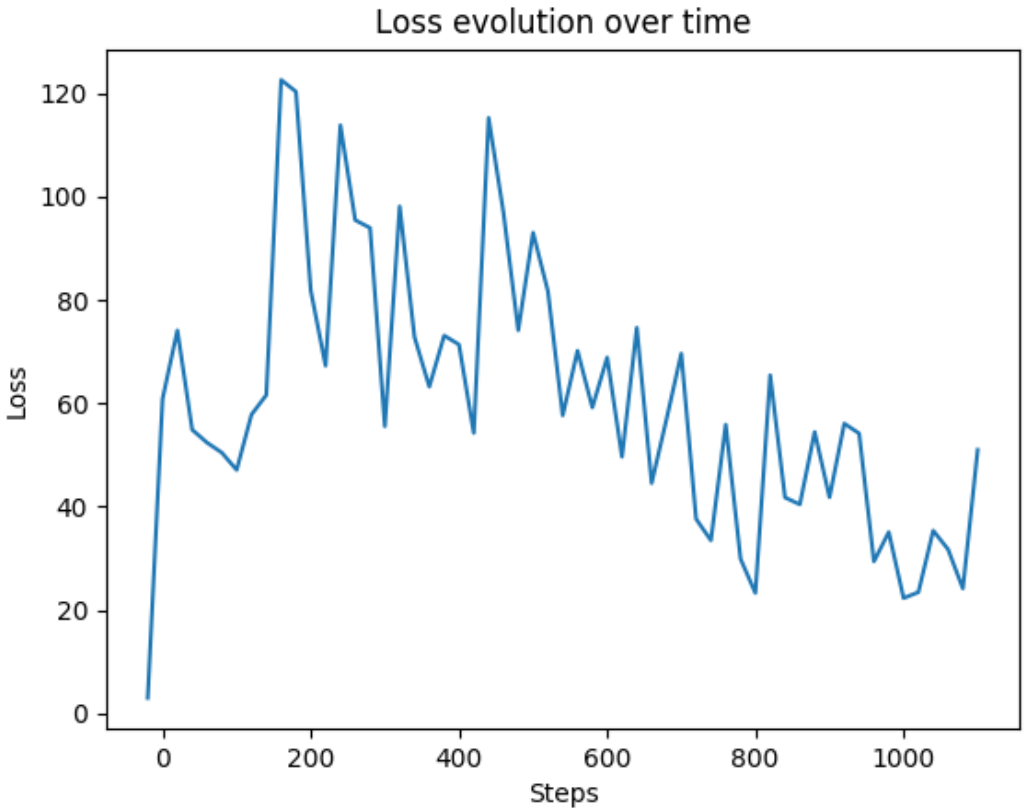


Figure 27.: Loss evolution of second network, first state, second test.

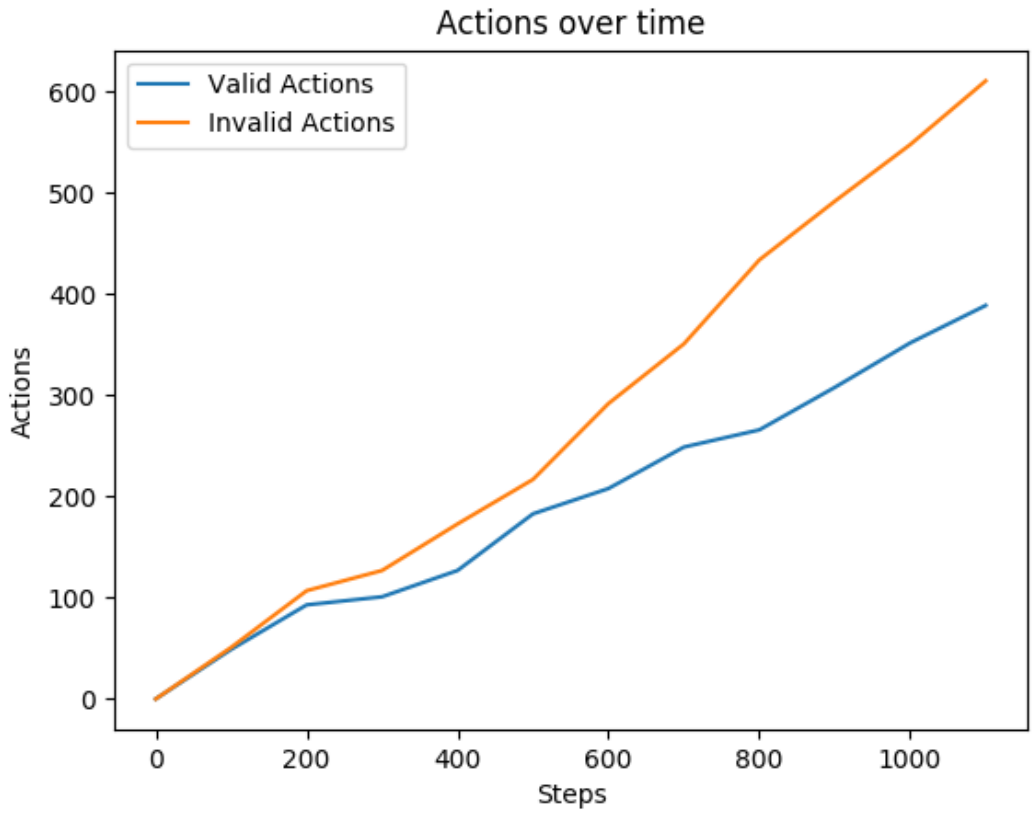


Figure 28.: Actions analysis of second network, first state, second test.

Third Test

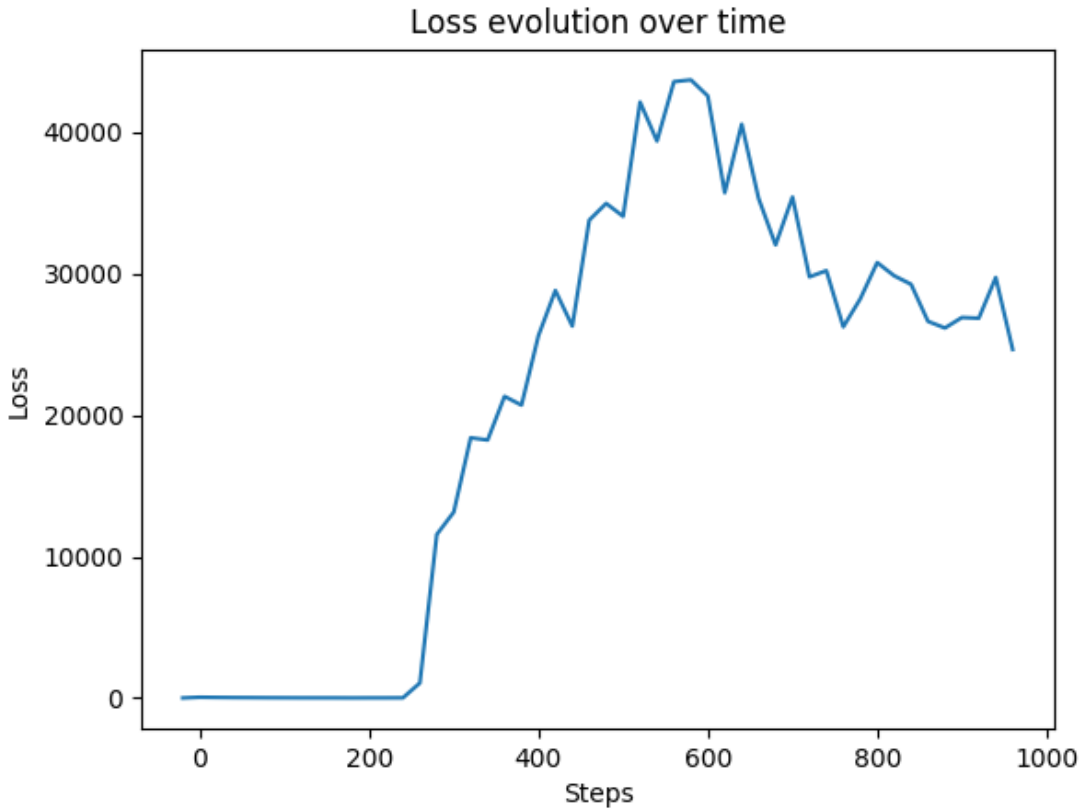


Figure 29.: Loss evolution of second network, first state, third test.

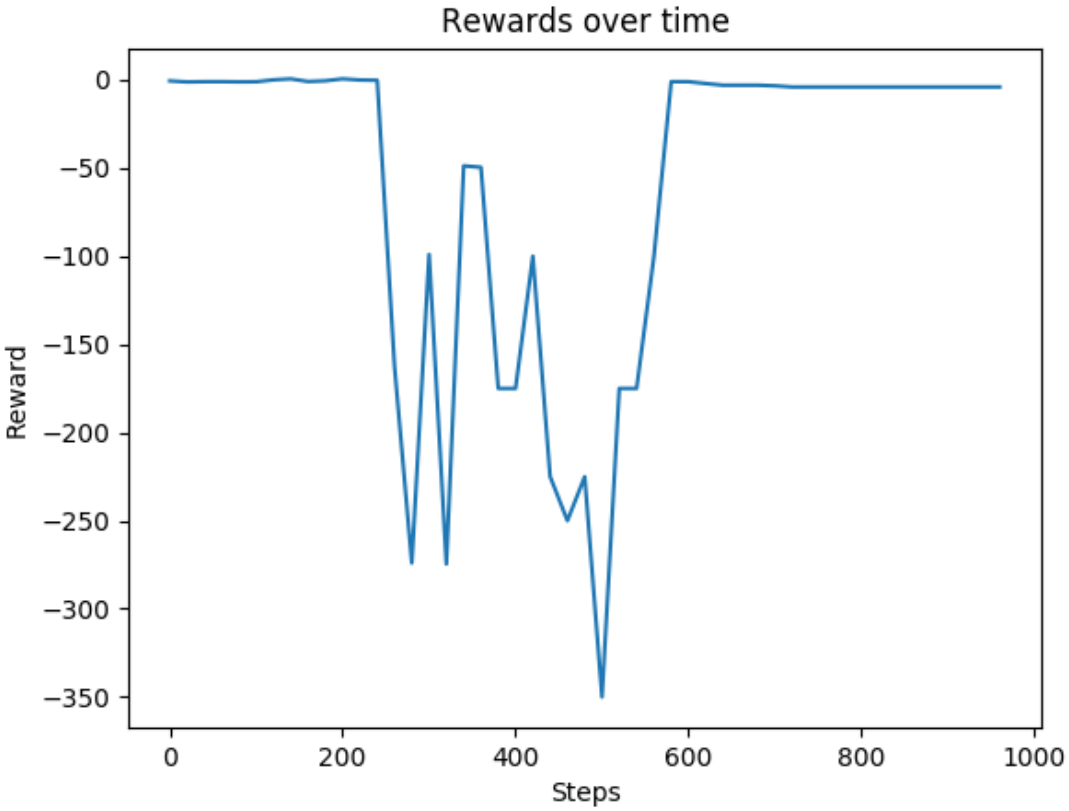


Figure 30.: Rewards given over time by second network, first state, third test.

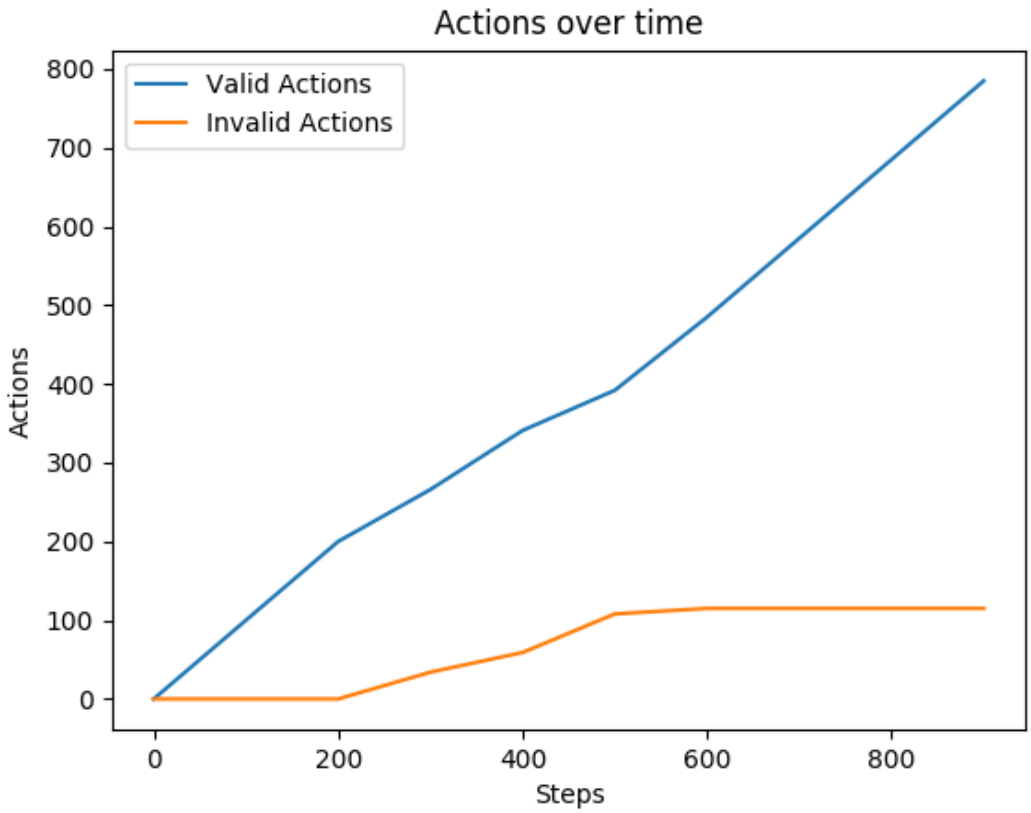


Figure 31.: Actions analysis of second network, first state, third test.

Fourth Test

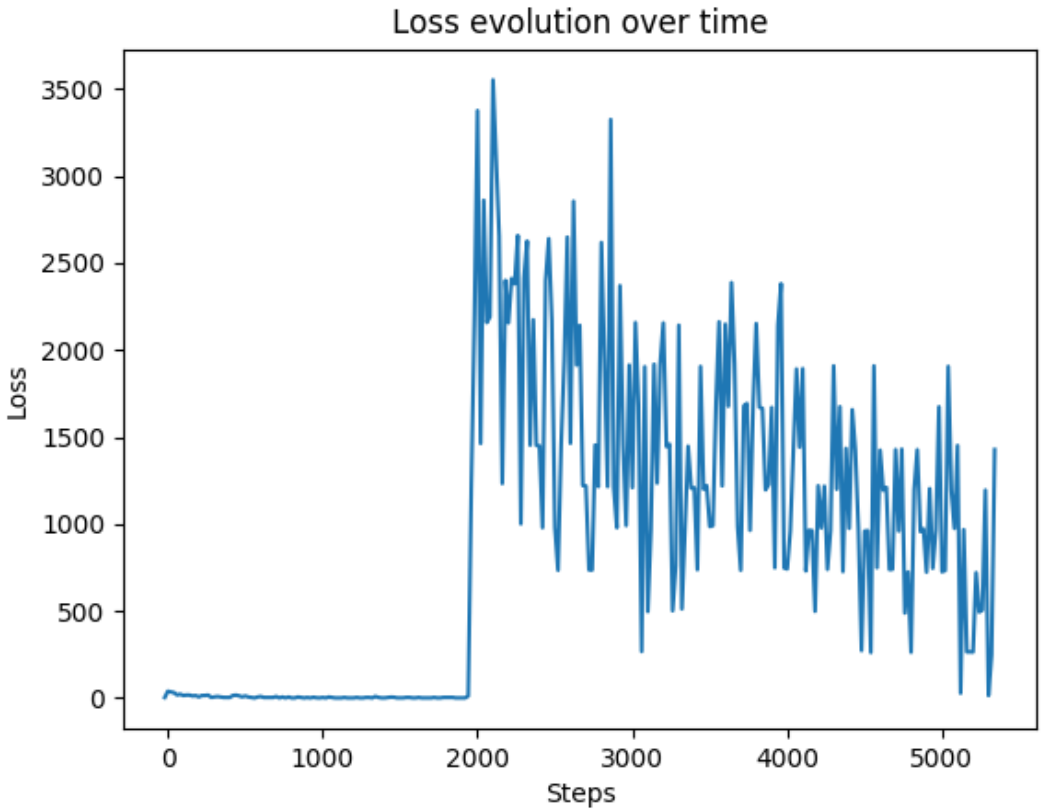


Figure 32.: Loss evolution of second network, first state, fourth test.

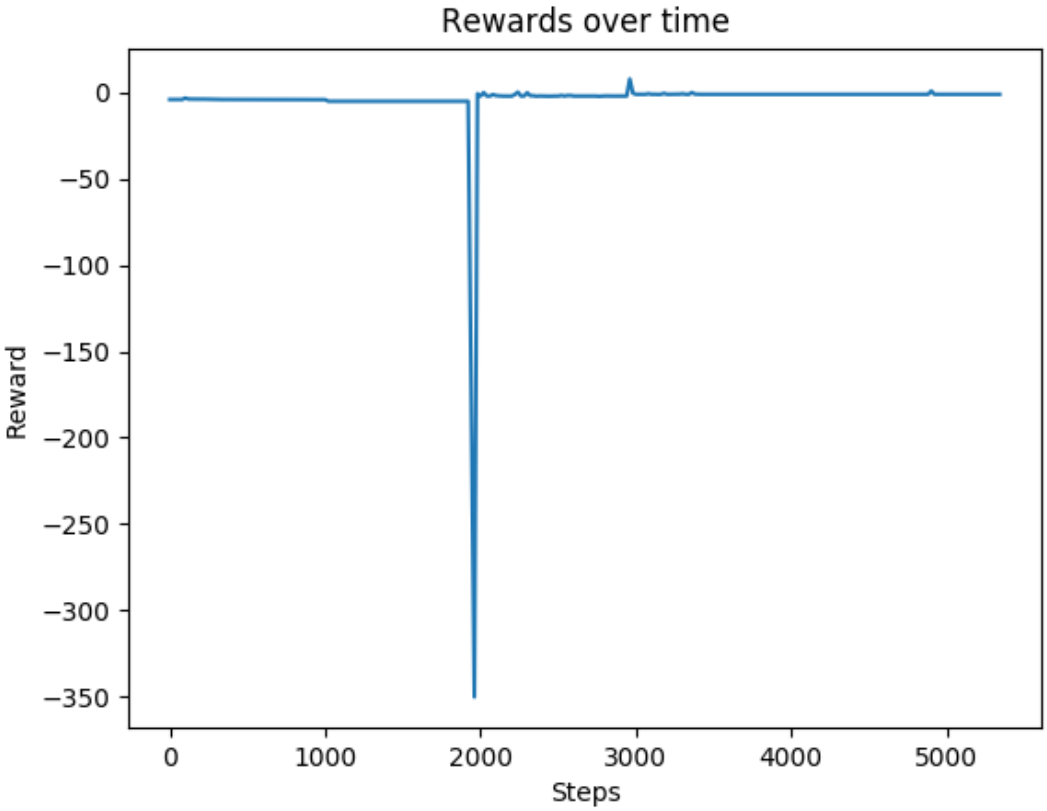


Figure 33.: Rewards given over time by second network, first state, fourth test.

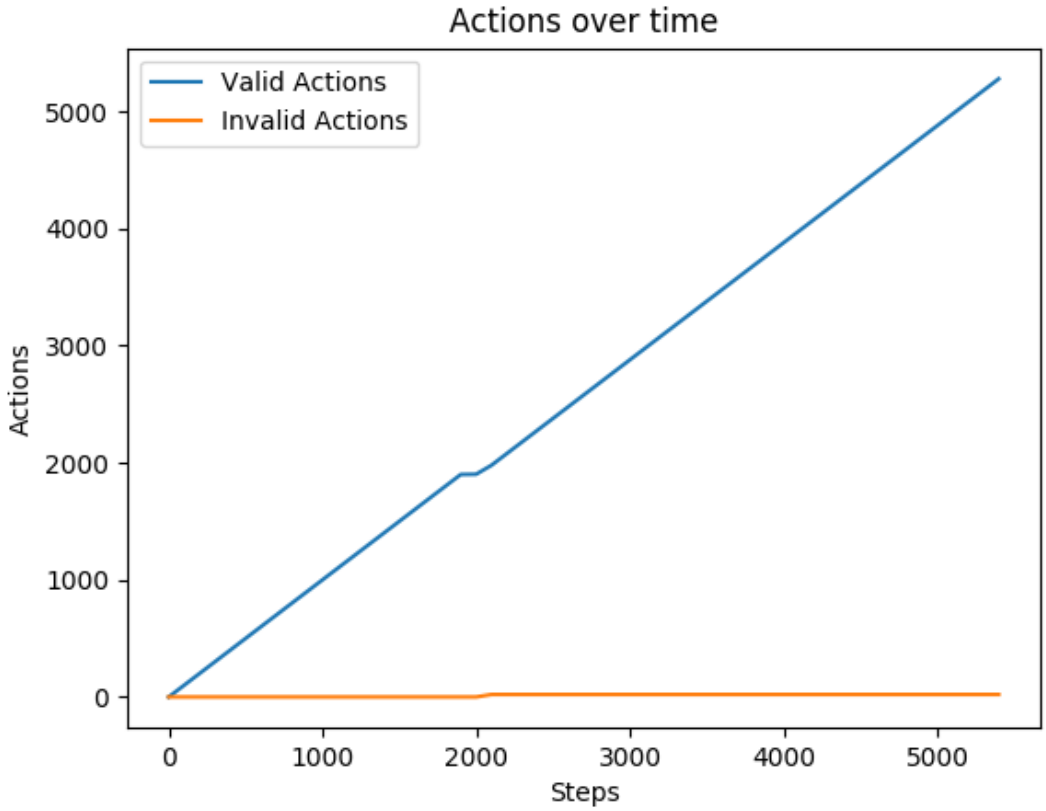


Figure 34.: Actions analysis of second network, first state, fourth test.

Fifth Test

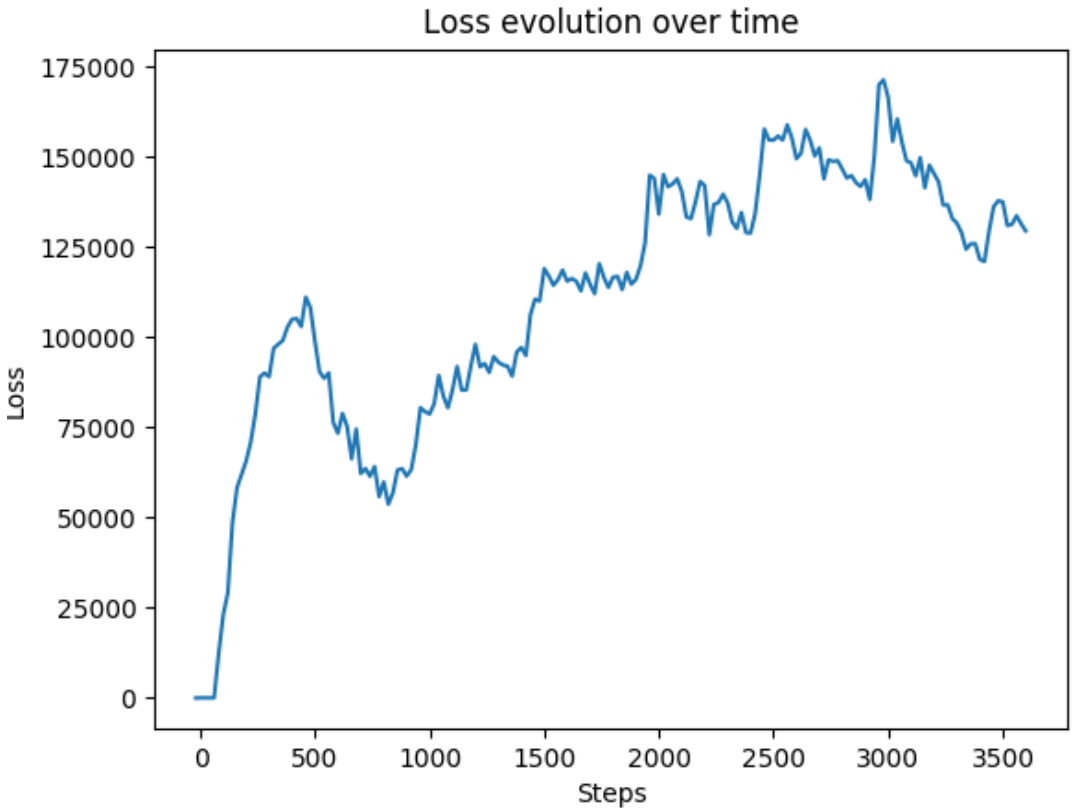


Figure 35.: Loss evolution of second network, first state, fifth test.

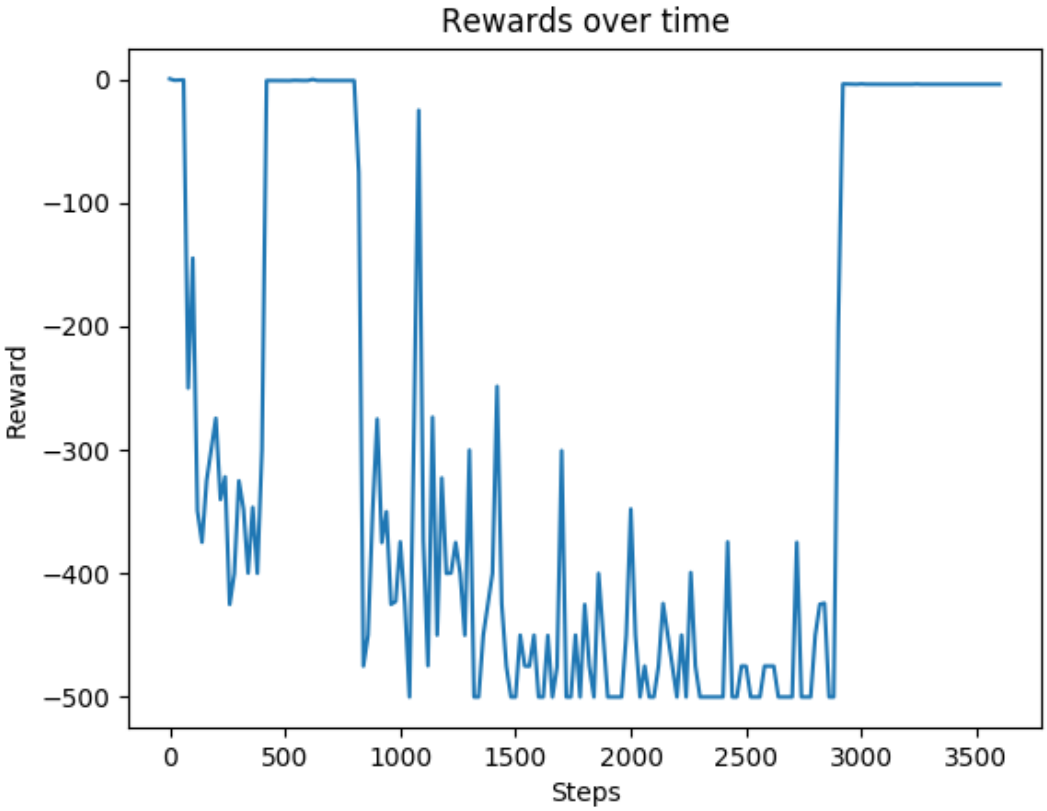


Figure 36.: Rewards given over time by second network, first state, fifth test.

Figure 37.: Actions analysis of second network, first state, fifth test.

Sixth Test

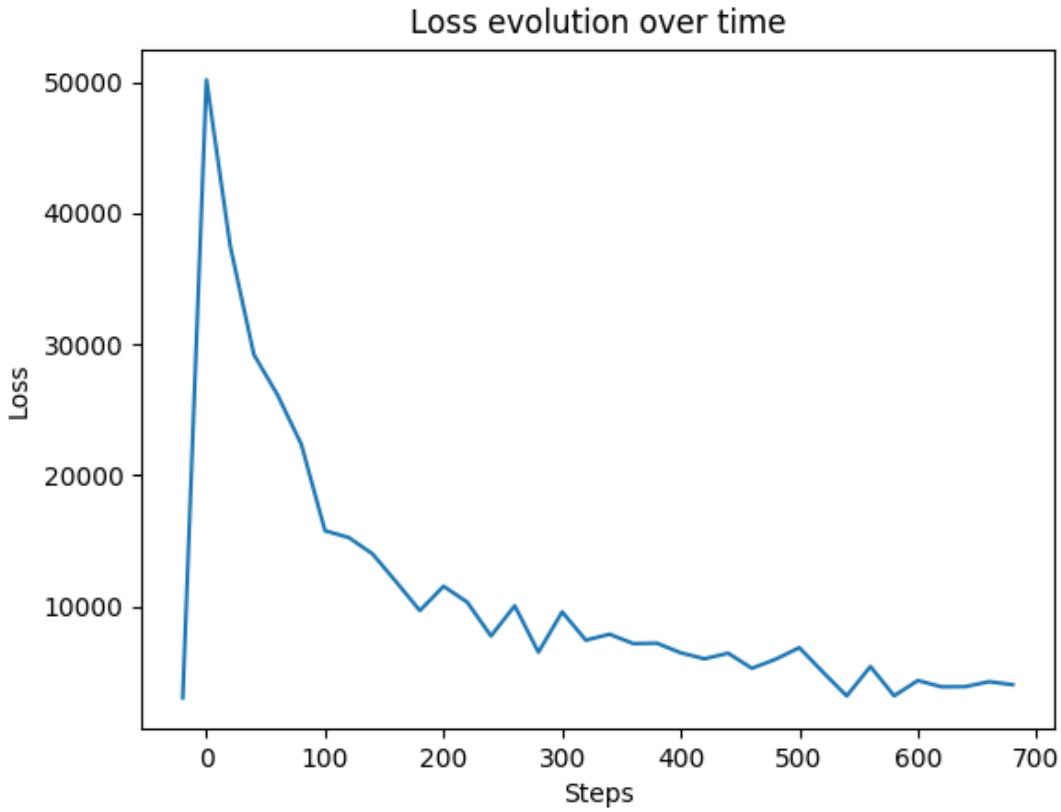


Figure 38.: Loss evolution of second network, first state, sixth test.

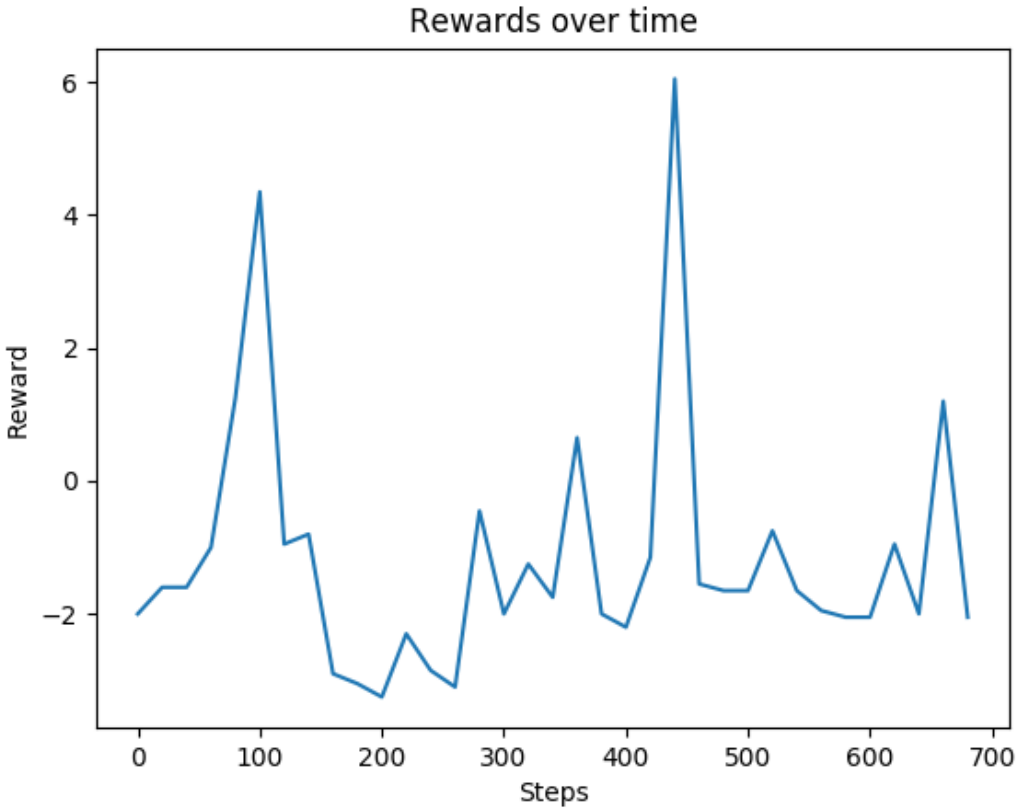


Figure 39.: Rewards given over time by second network, first state, sixth test.

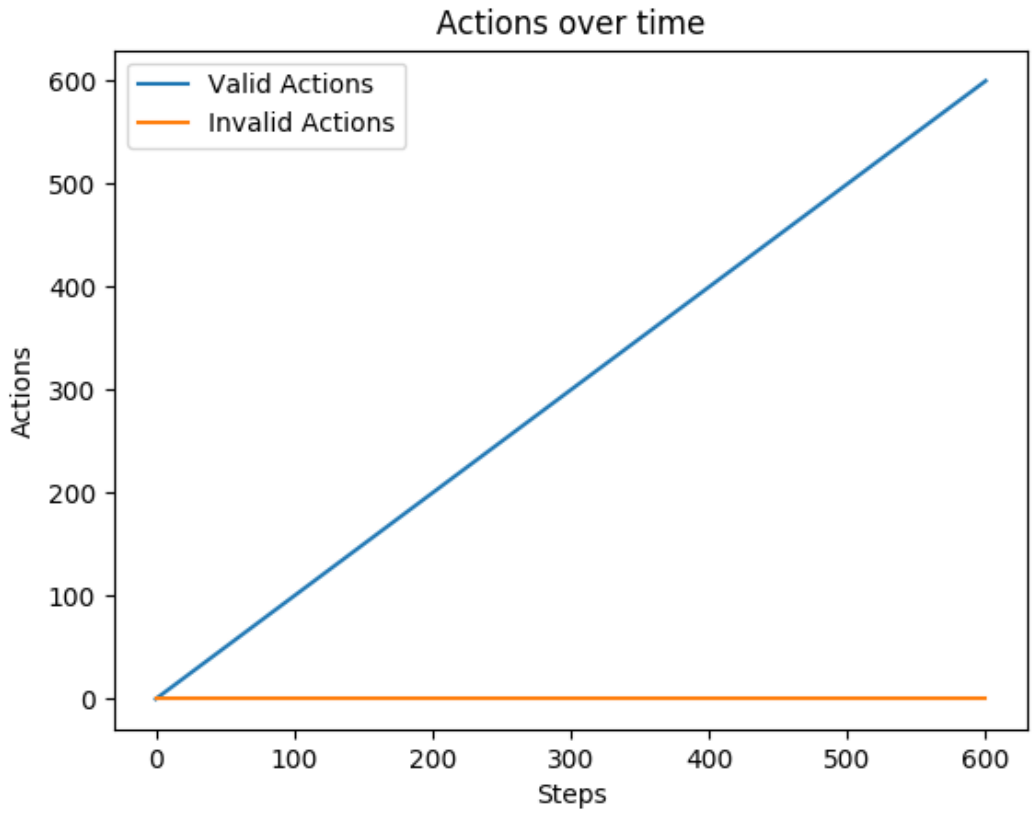


Figure 40.: Actions analysis of second network, first state, sixth test.

A.1.3 *Third Network Results*

First Test

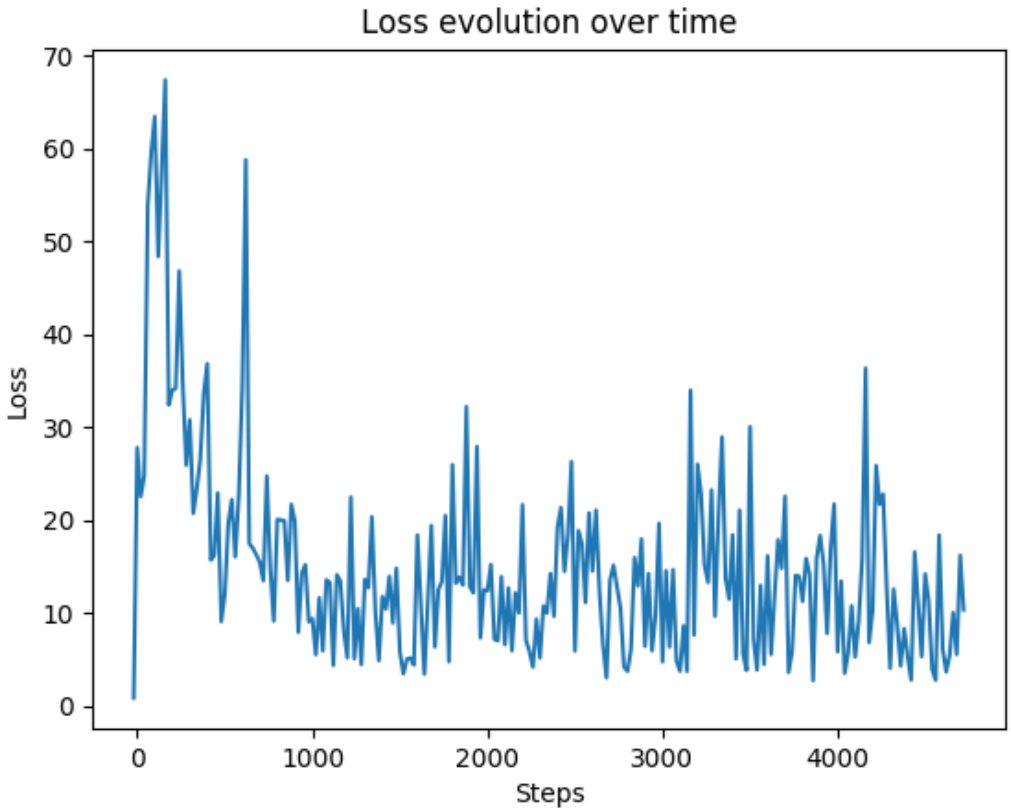


Figure 41.: Loss evolution of third network, first state, first test.

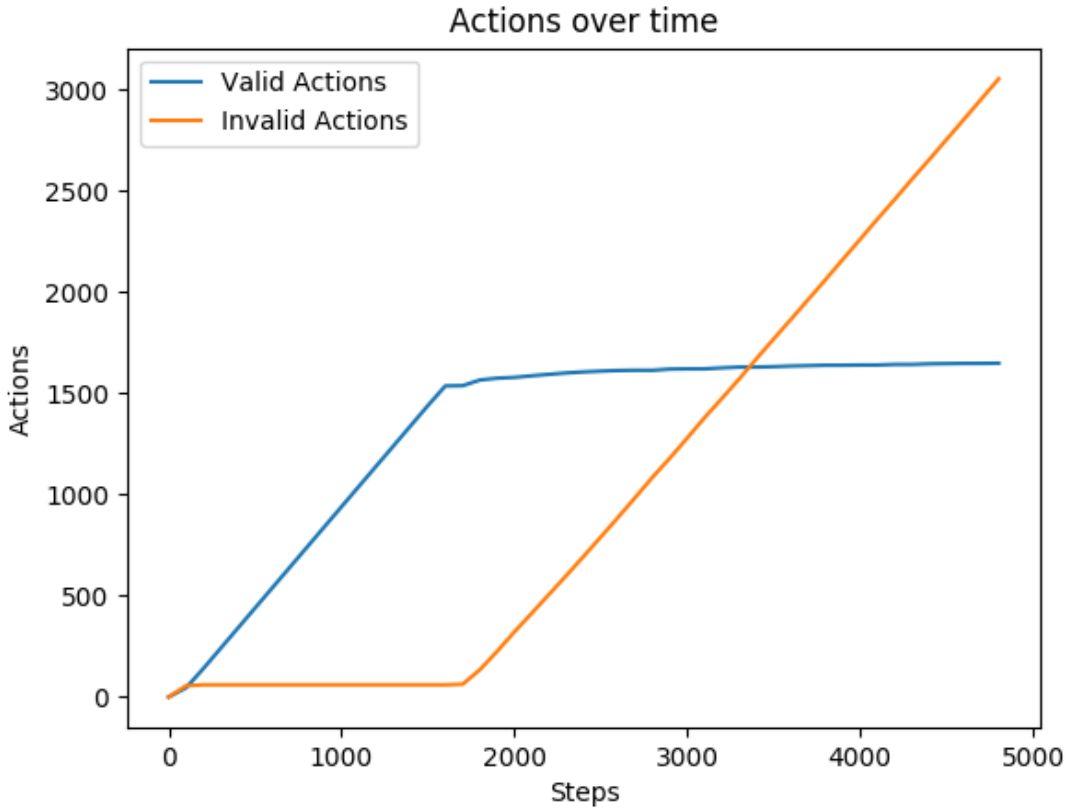


Figure 42.: Actions analysis of third network, first state, first test.

A.2 Second State Experiences

A.2.1 *Second Network Results*

First Test

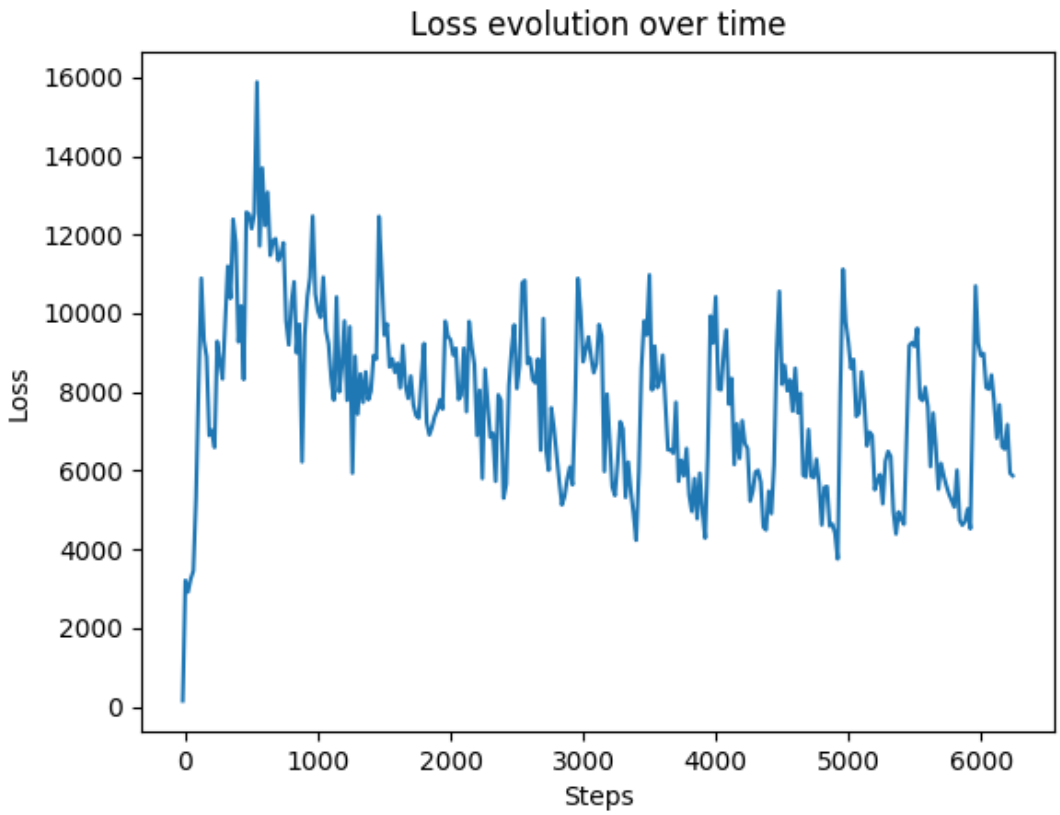


Figure 43.: Loss evolution of second network, second state, first test.

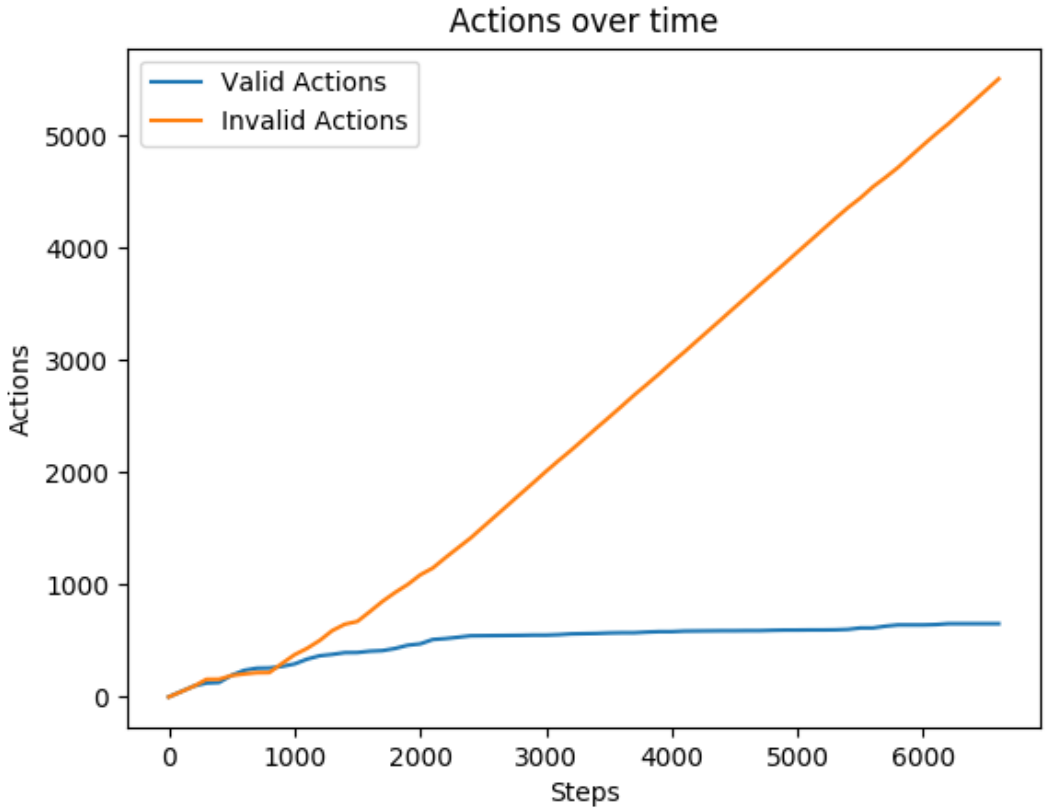


Figure 44.: Actions analysis of second network, second state, first test.

Second Test

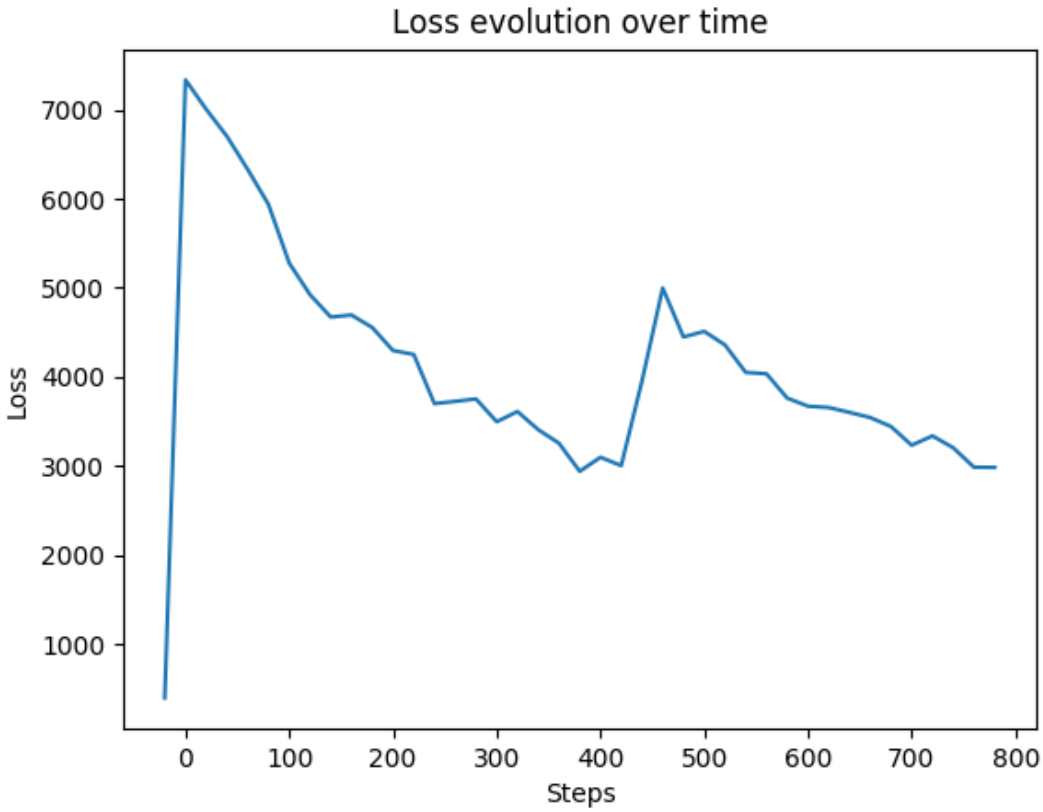


Figure 45.: Loss evolution of second network, second state, second test.

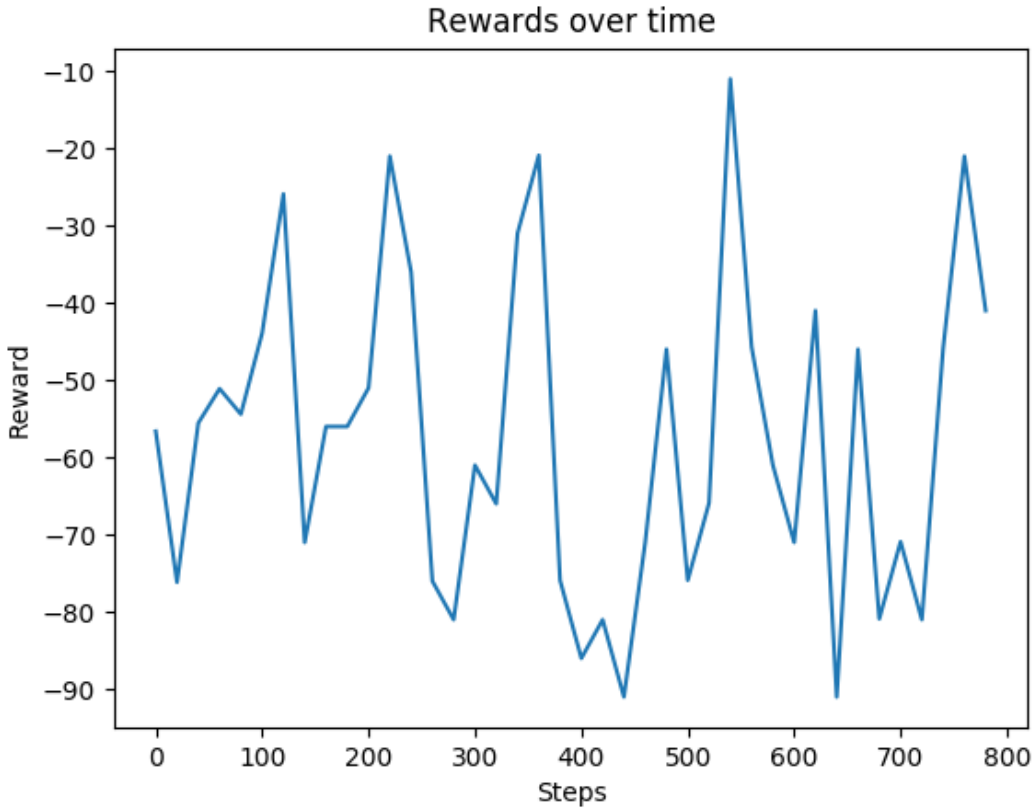


Figure 46.: Rewards given over time by second network, second state, second test.

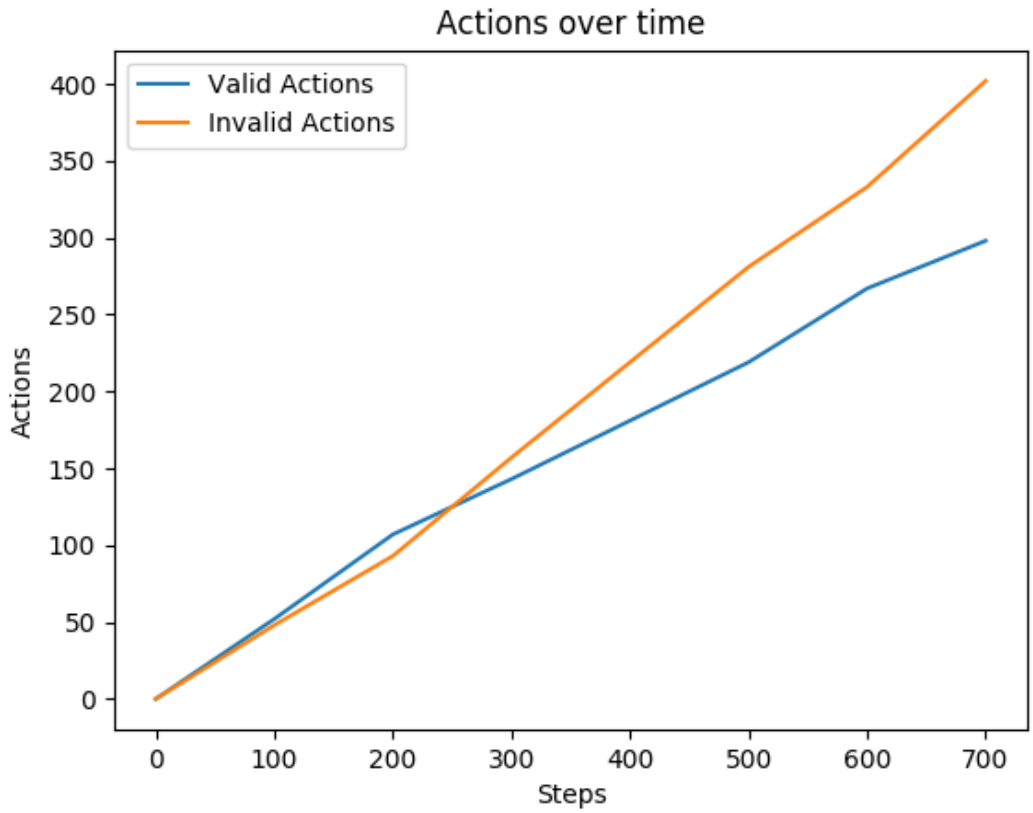


Figure 47.: Actions analysis of second network, second state, second test.

Third Test

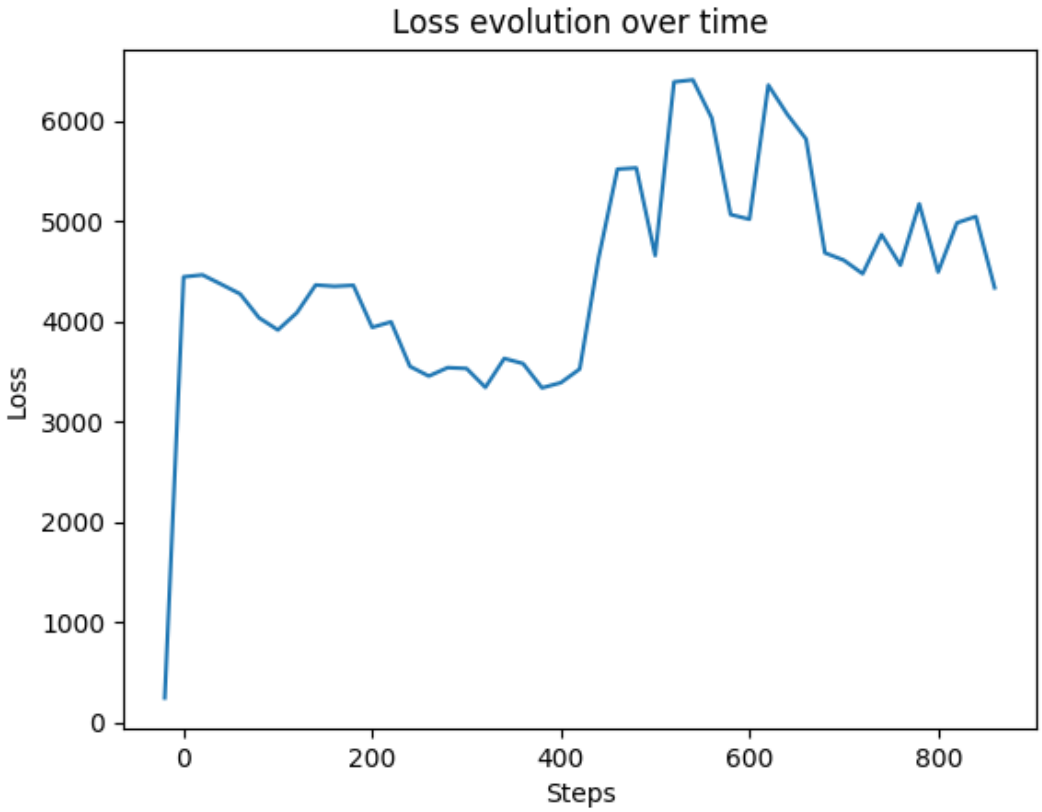


Figure 48.: Loss evolution of second network, second state, third test.

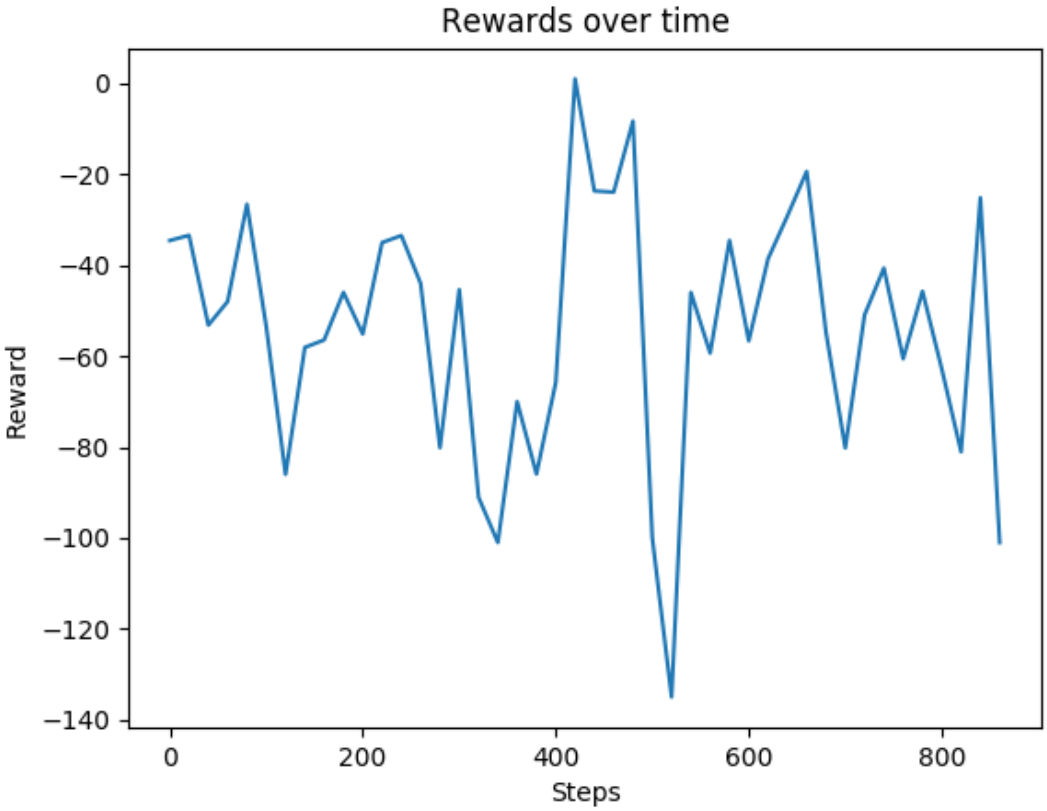


Figure 49.: Rewards given over time by second network, second state, third test.

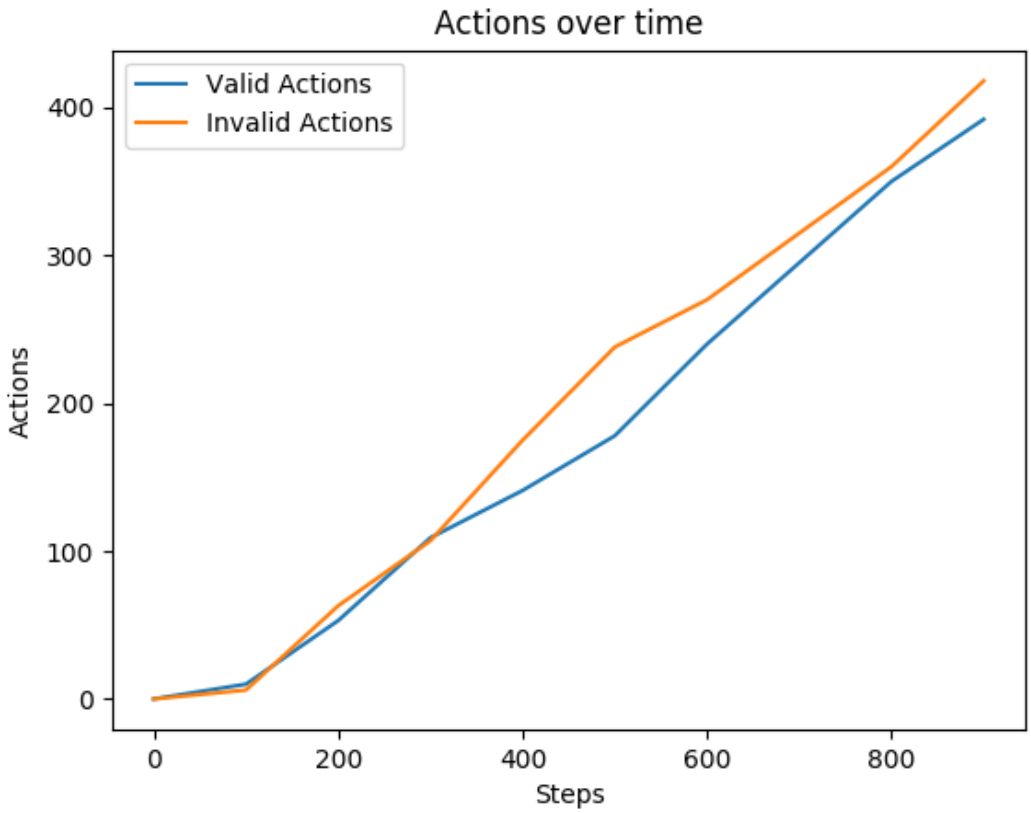


Figure 50.: Actions analysis of second network, second state, third test.

A.2.2 *Third Network Results*

First Test

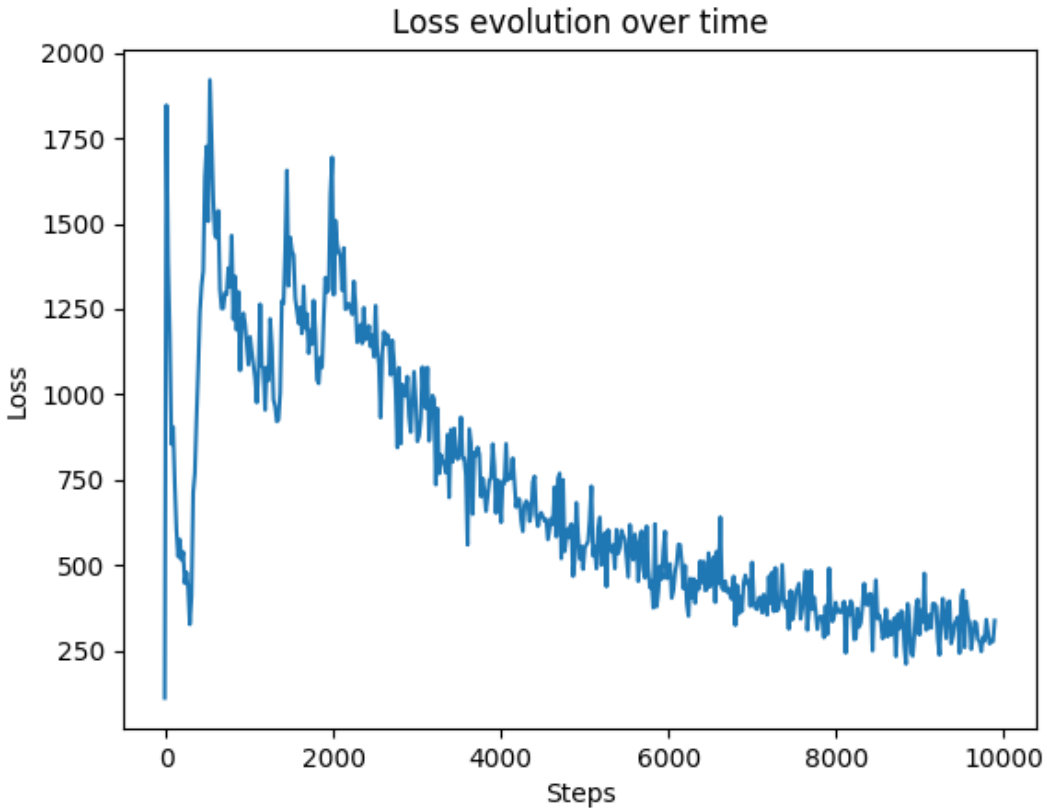


Figure 51.: Loss evolution of third network, second state, first test.

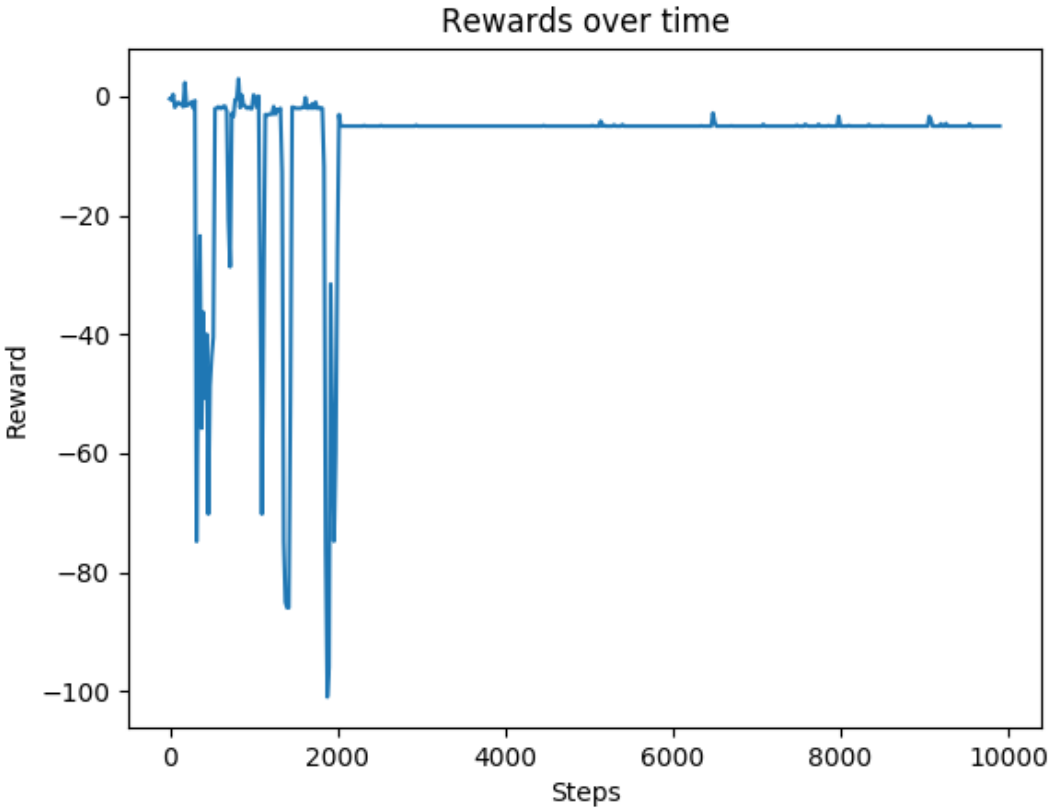


Figure 52.: Rewards given over time by third network, second state, first test.

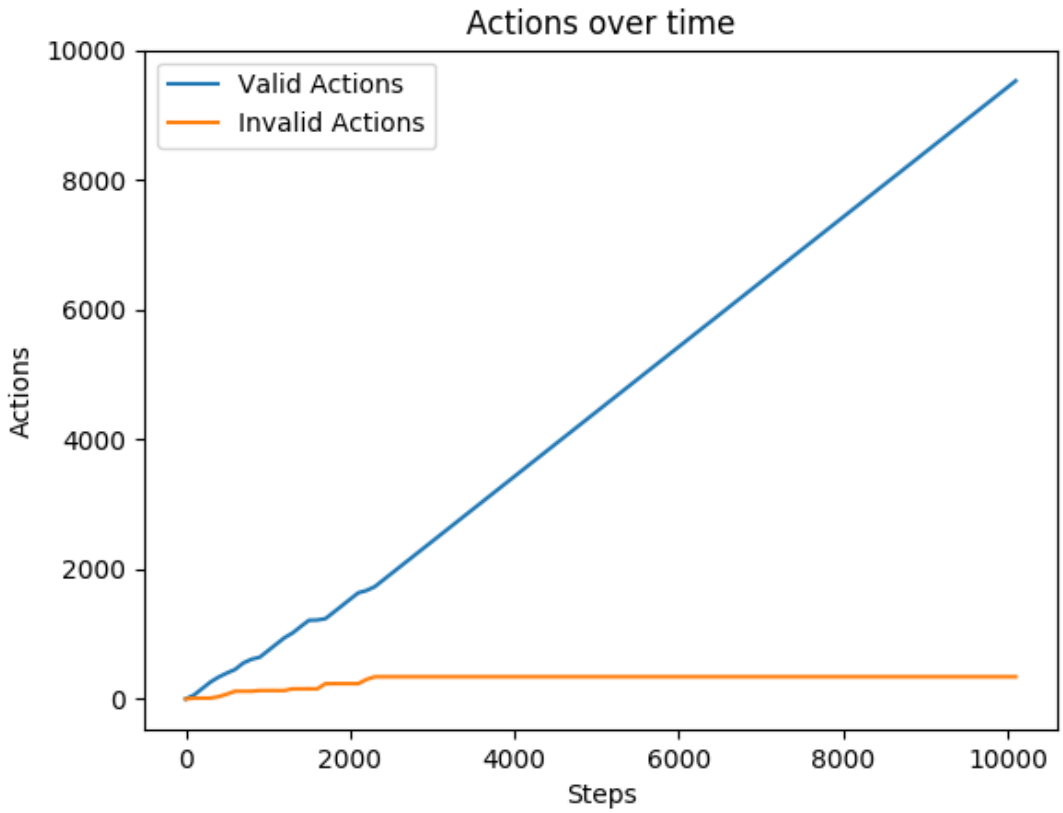


Figure 53.: Actions analysis of third network, second state, first test.

Second Test

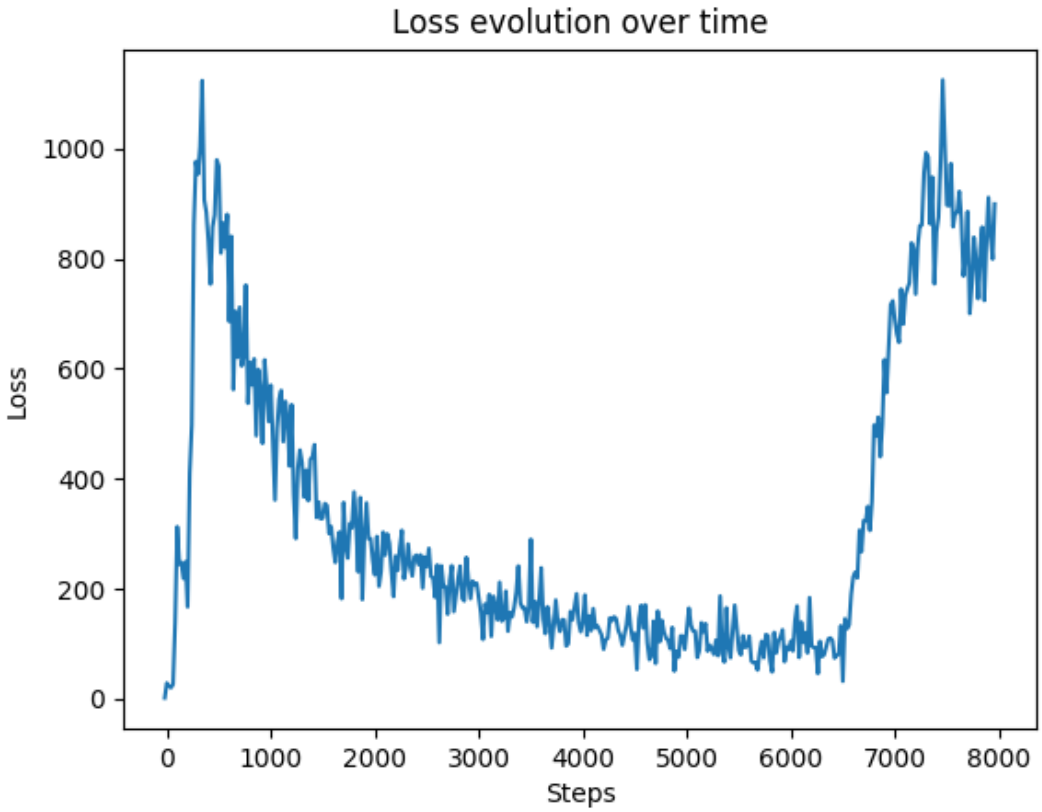


Figure 54.: Loss evolution of third network, second state, second test.

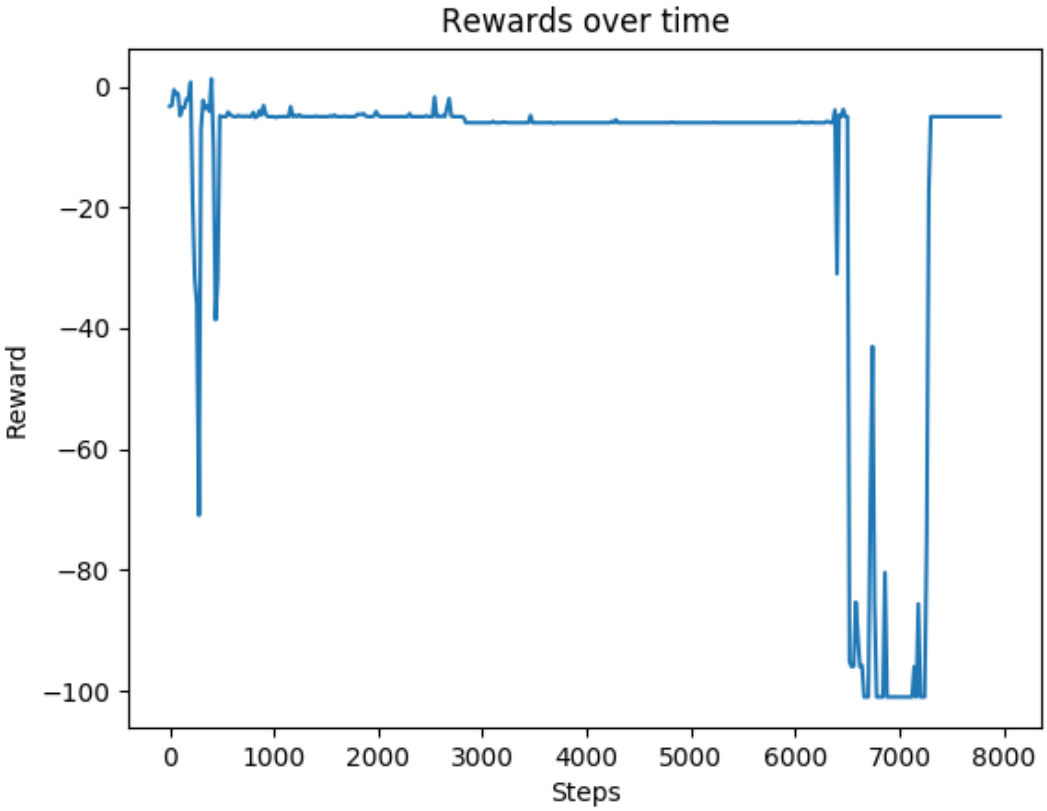


Figure 55.: Rewards given over time by third network, second state, second test.

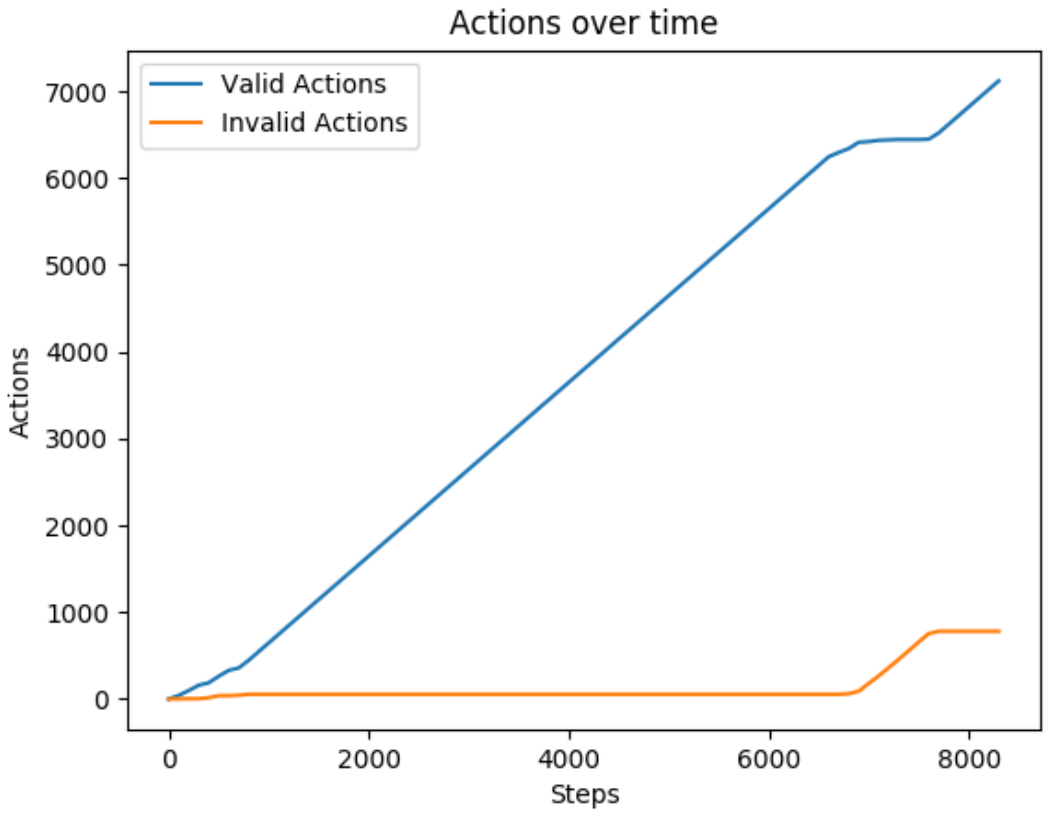


Figure 56.: Actions analysis of third network, second state, second test.

Third Test

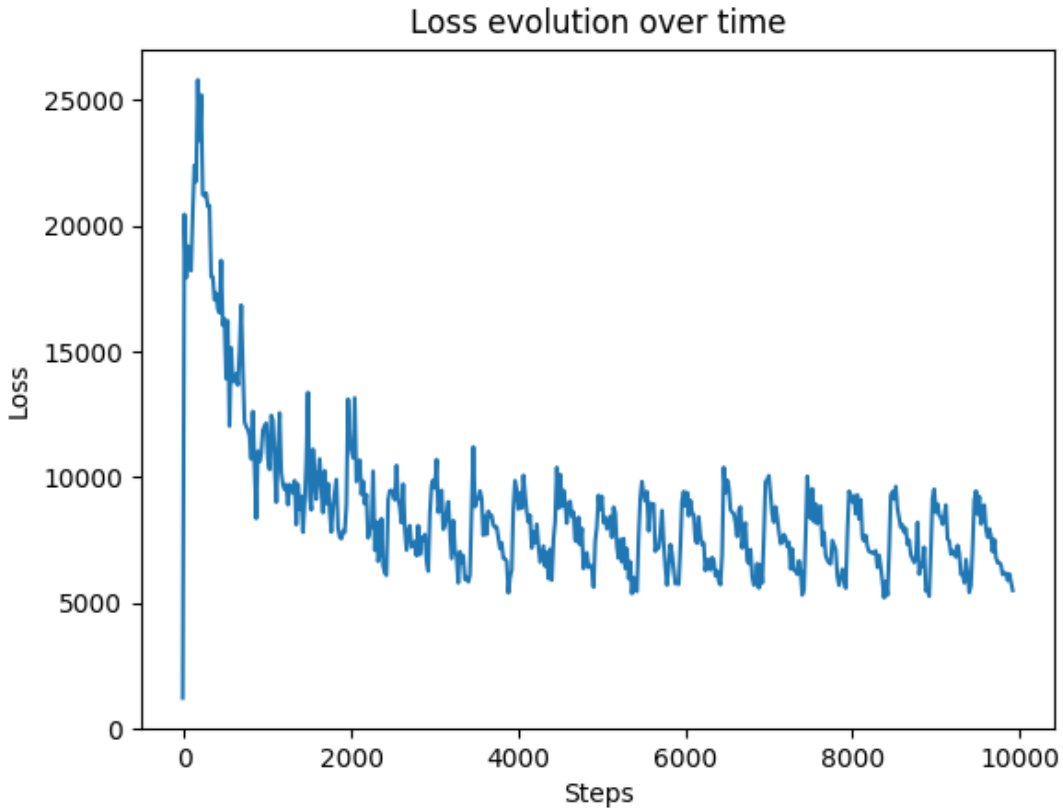


Figure 57.: Loss evolution of third network, second state, third test.

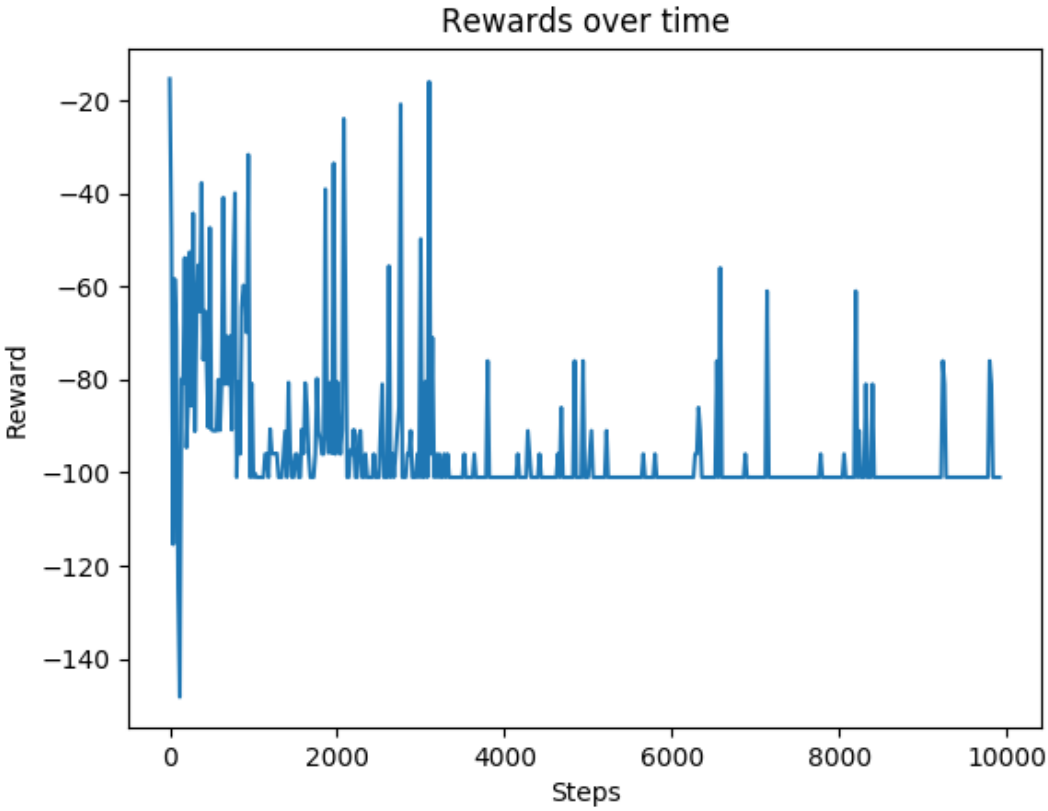


Figure 58.: Rewards given over time by third network, second state, third test.

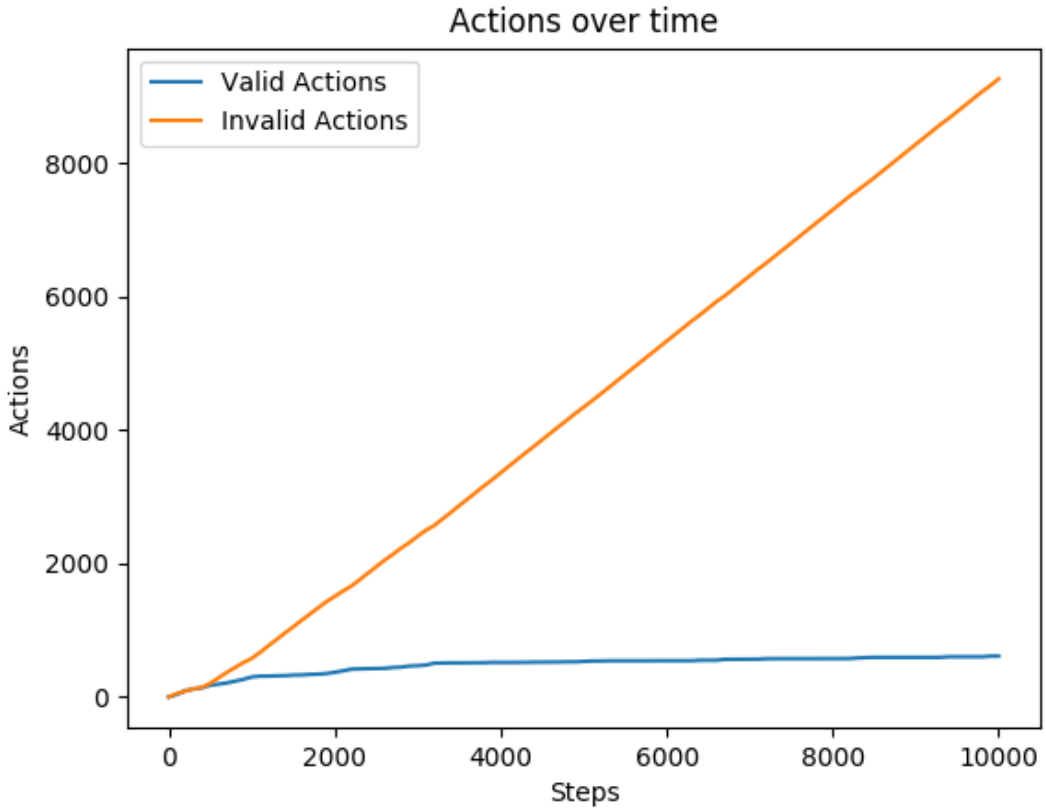


Figure 59.: Actions analysis of third network, second state, third test.

A.2.3 *fourth Network Results*

First Test

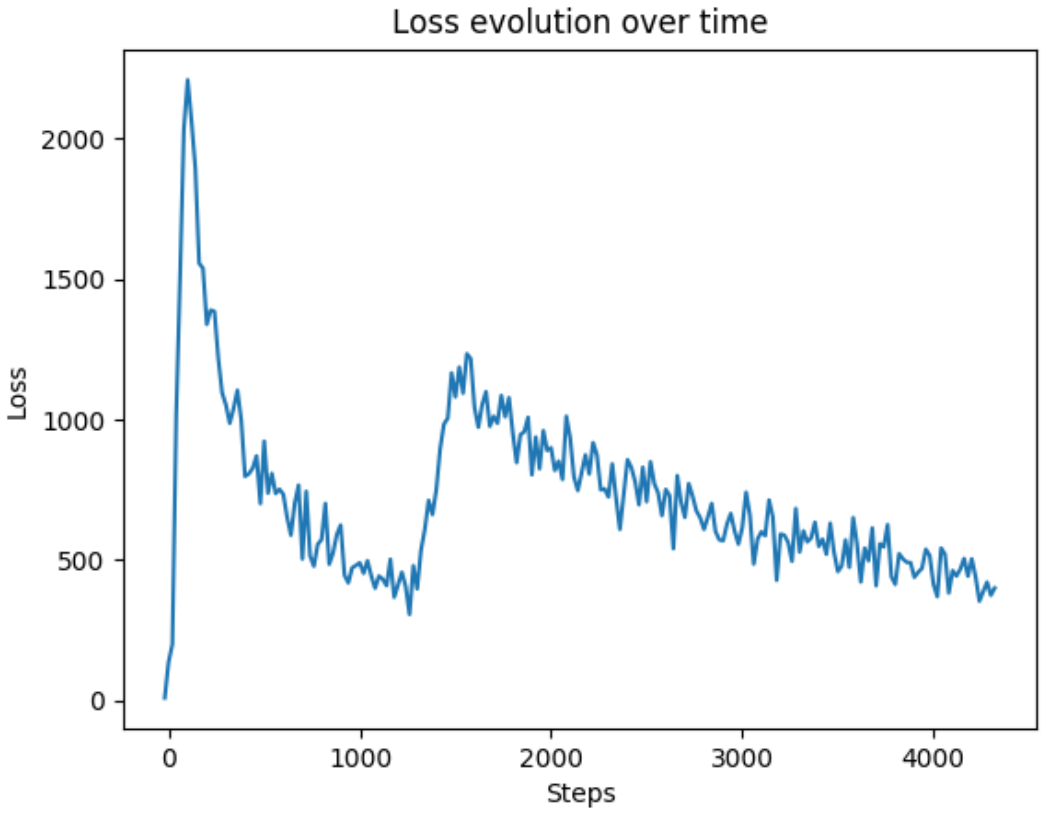


Figure 60.: Loss evolution of fourth network, second state, first test.

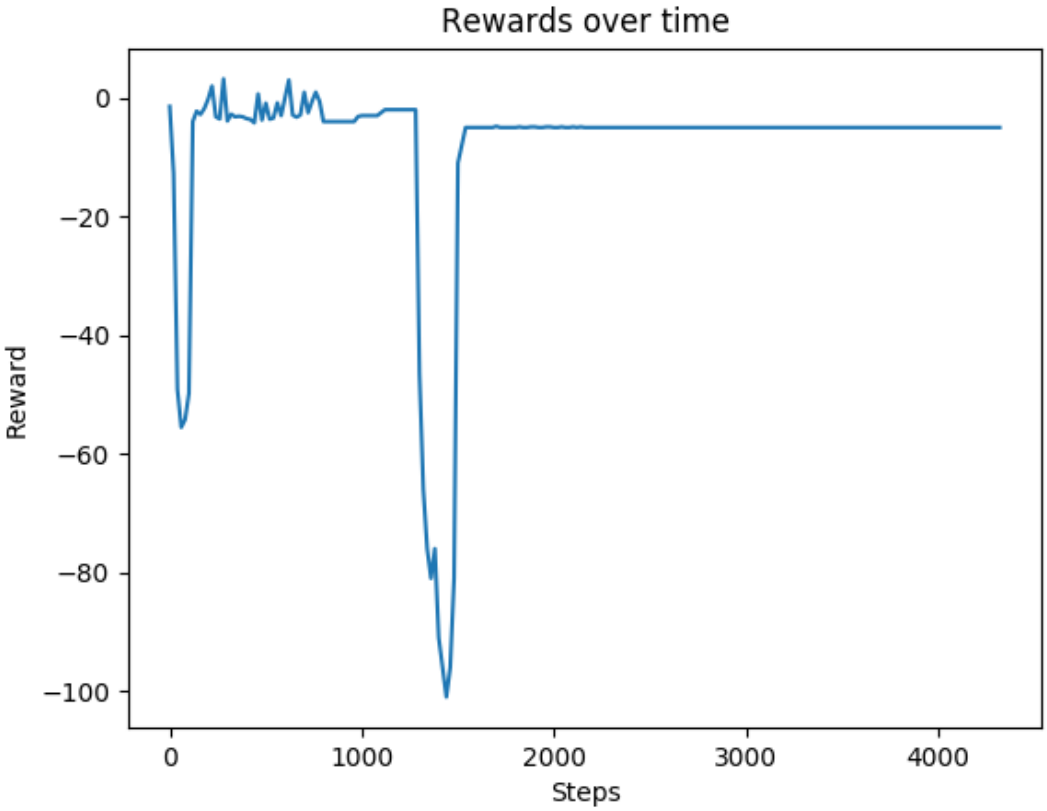


Figure 61.: Rewards given over time by fourth network, second state, first test.

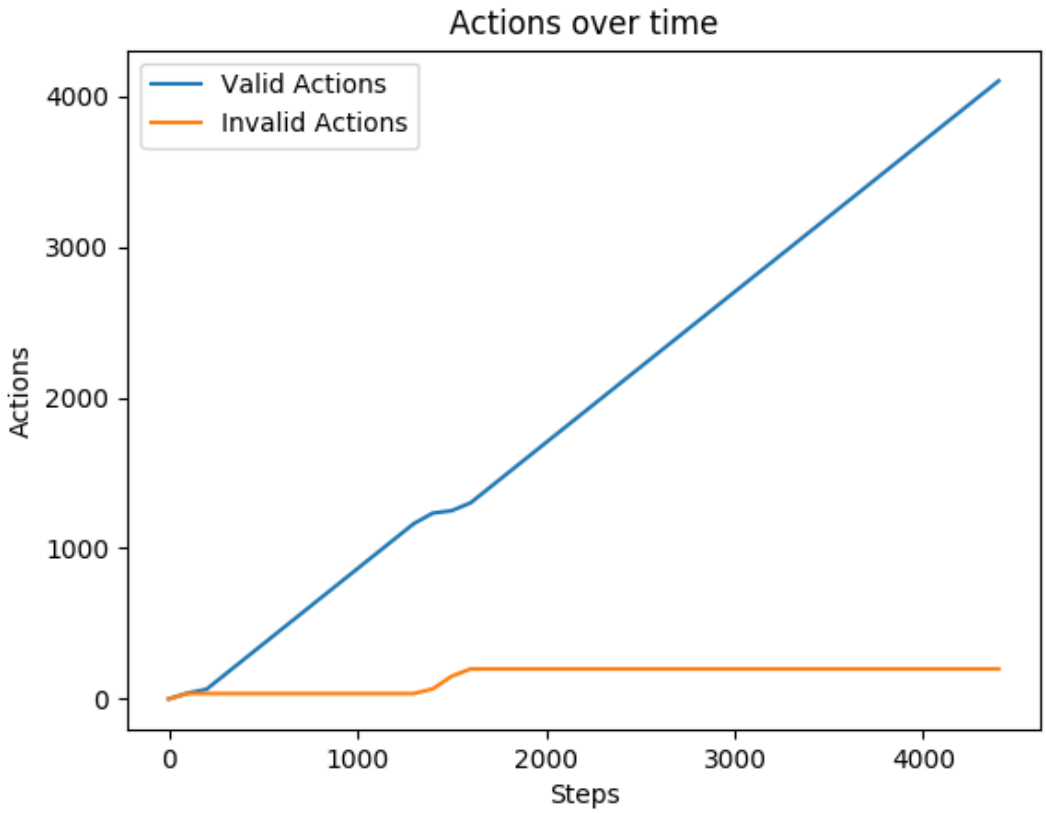


Figure 62.: Actions analysis of fourth network, second state, first test.

A.3 THIRD STATE EXPERIENCES

A.3.1 *Second Network Results*

First Test

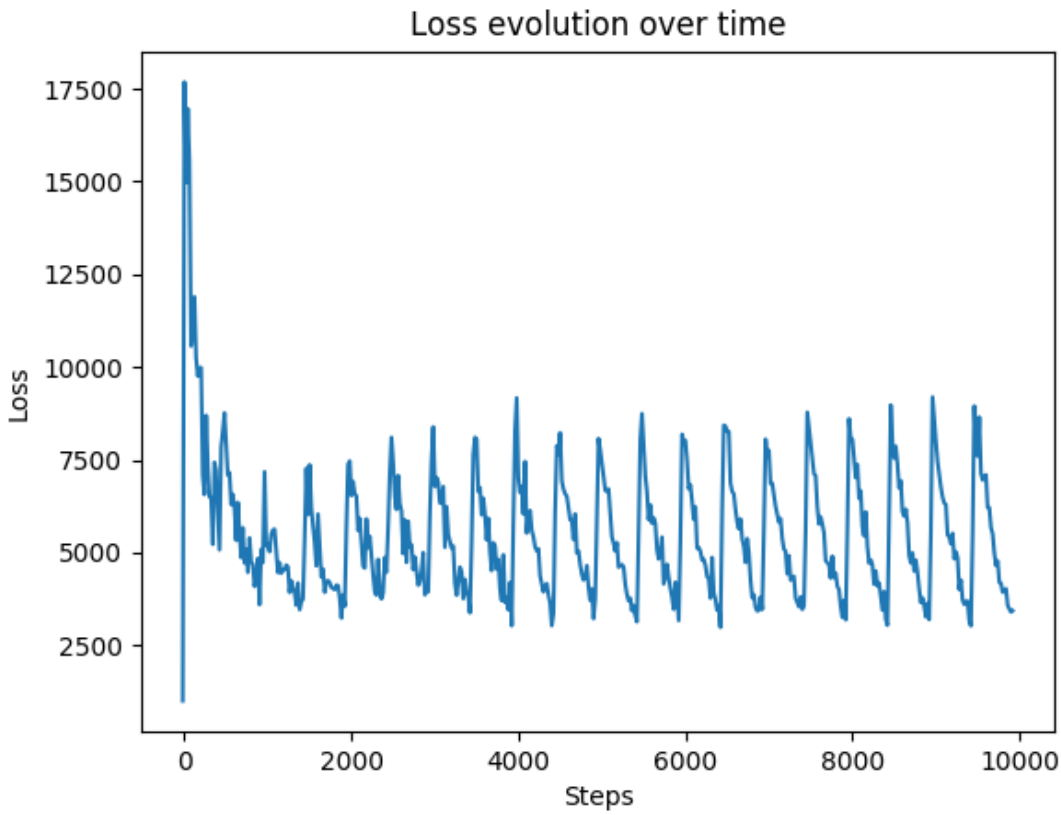


Figure 63.: Loss evolution of second network, third state, first test.

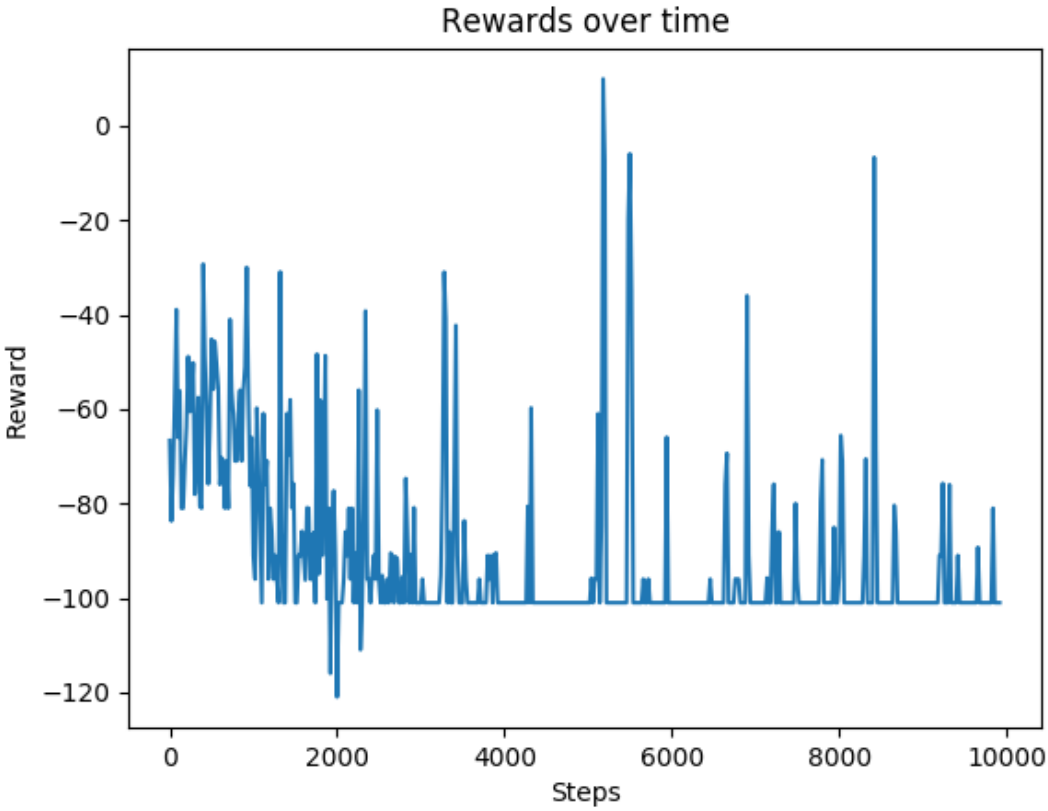


Figure 64.: Rewards given over time by second network, third state, first test.

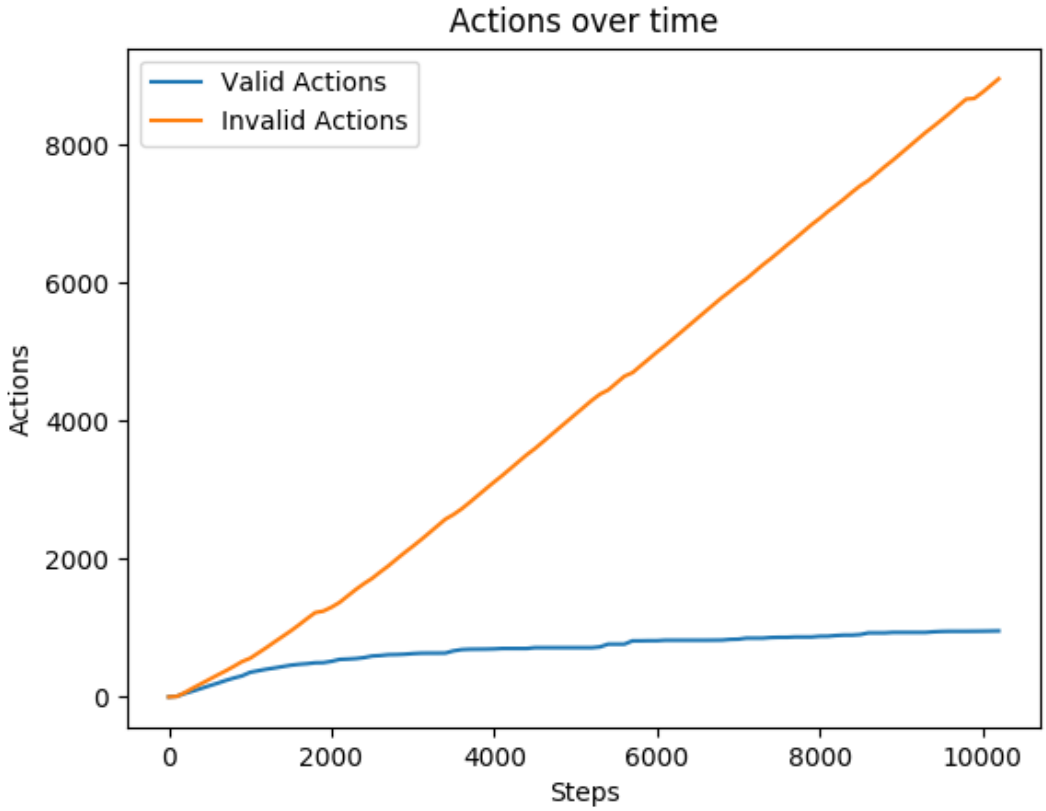


Figure 65.: Actions analysis of second network, third state, first test.

A.3.2 *Third Network Results*

First Test

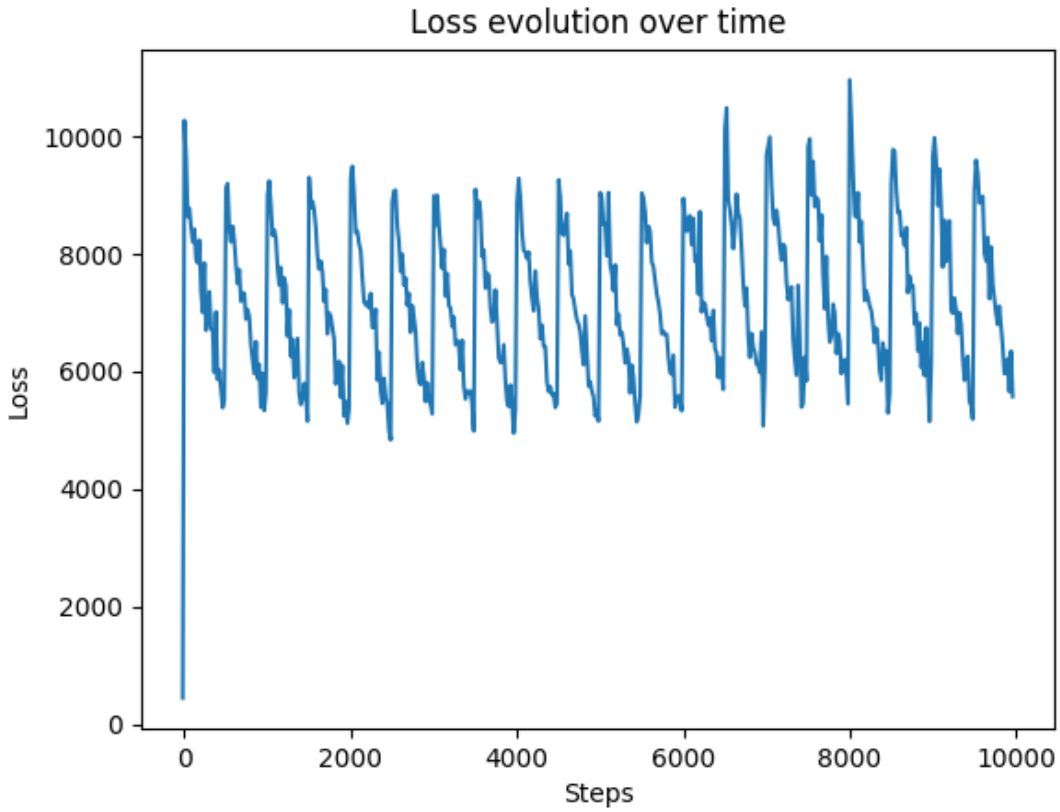


Figure 66.: Loss evolution of third network, third state, first test.

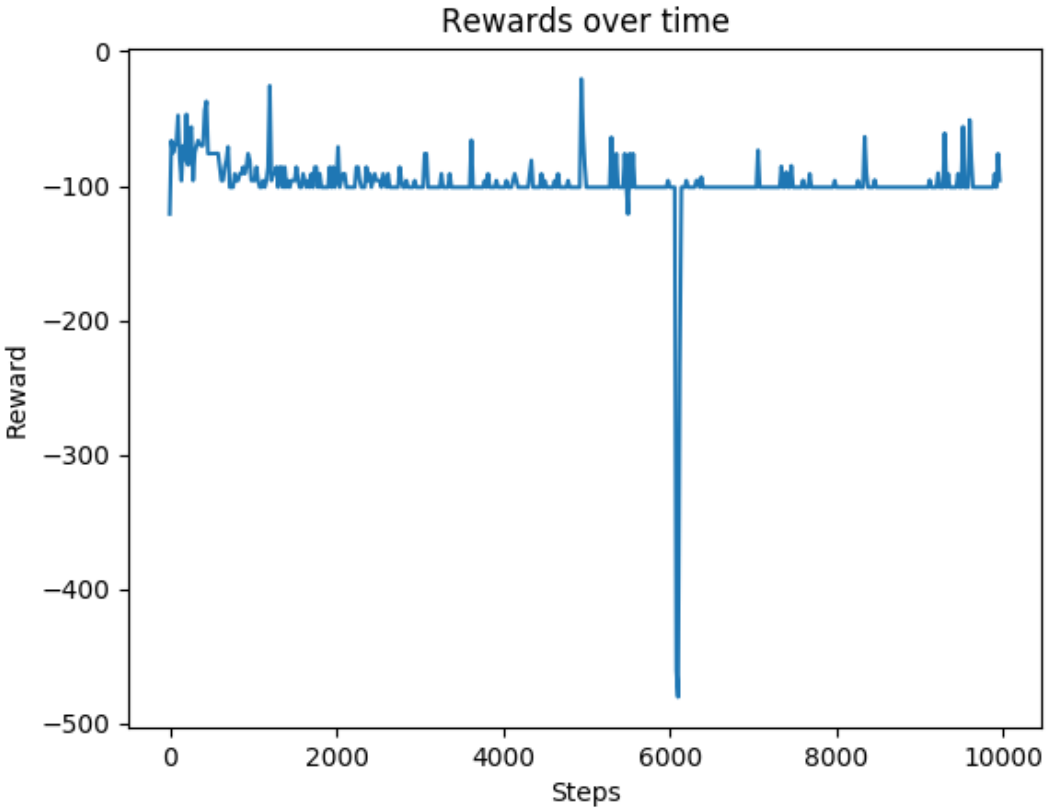


Figure 67.: Rewards given over time by third network, third state, first test.

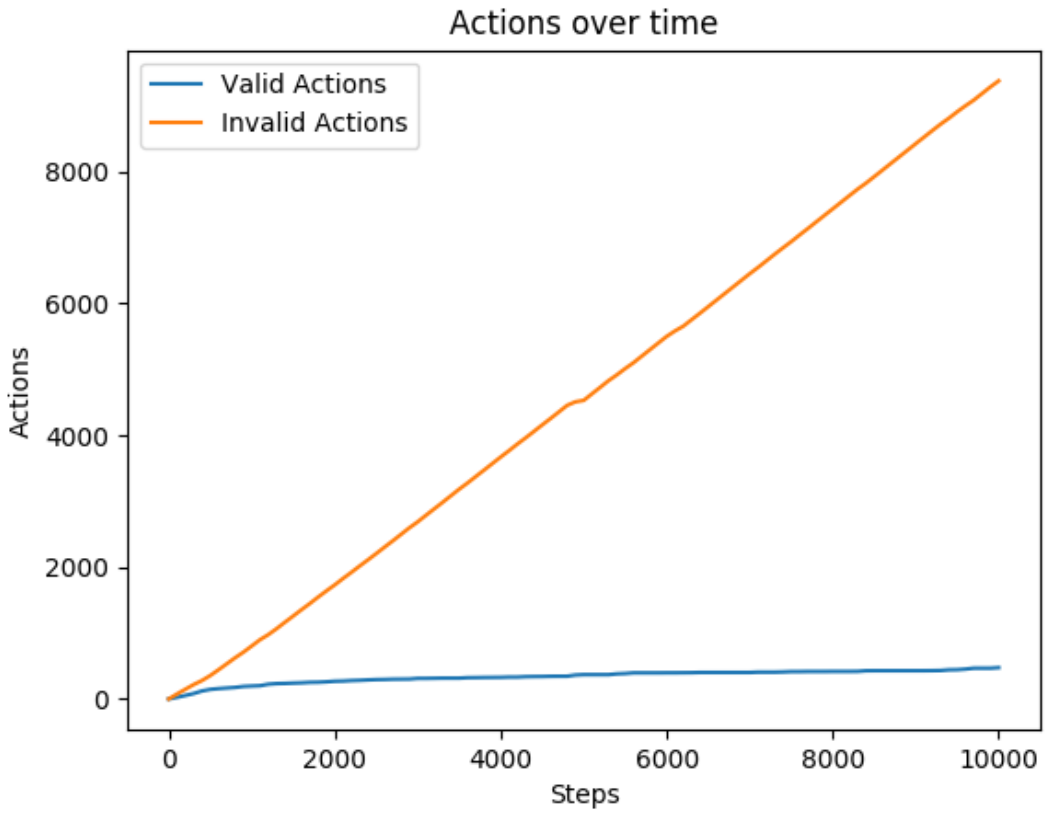


Figure 68.: Actions analysis of third network, third state, first test.

A.3.3 *Fourth Network Results*

First Test

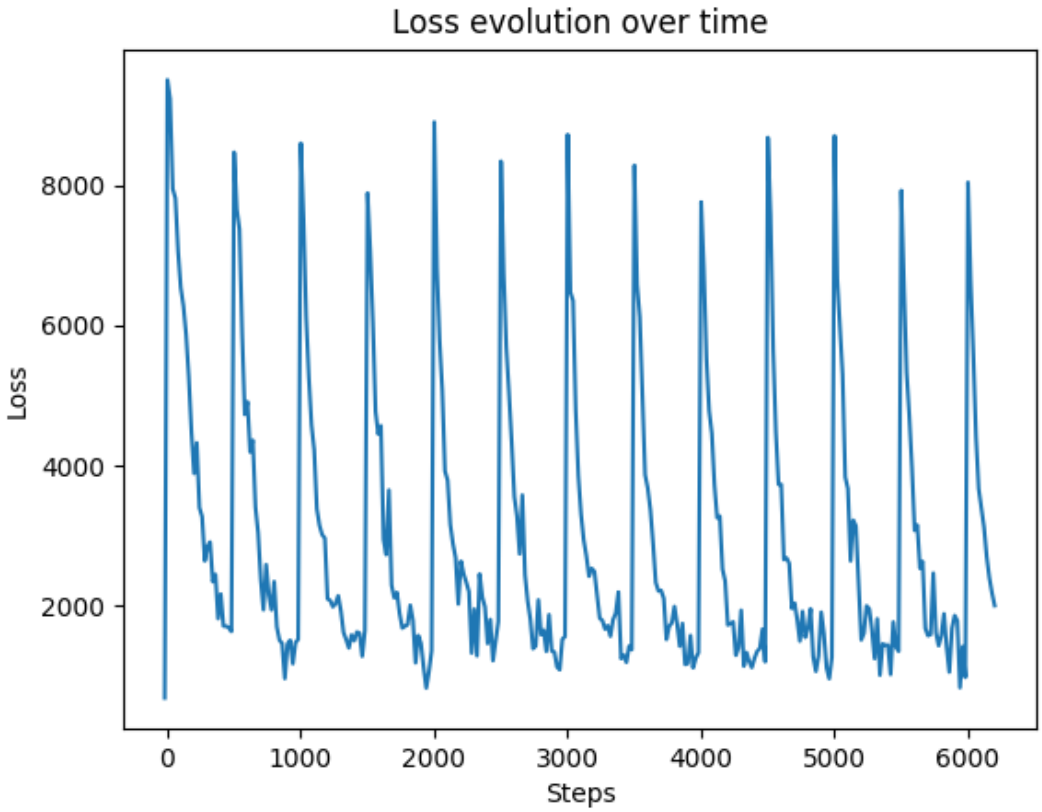


Figure 69.: Loss evolution of fourth network, third state, first test.

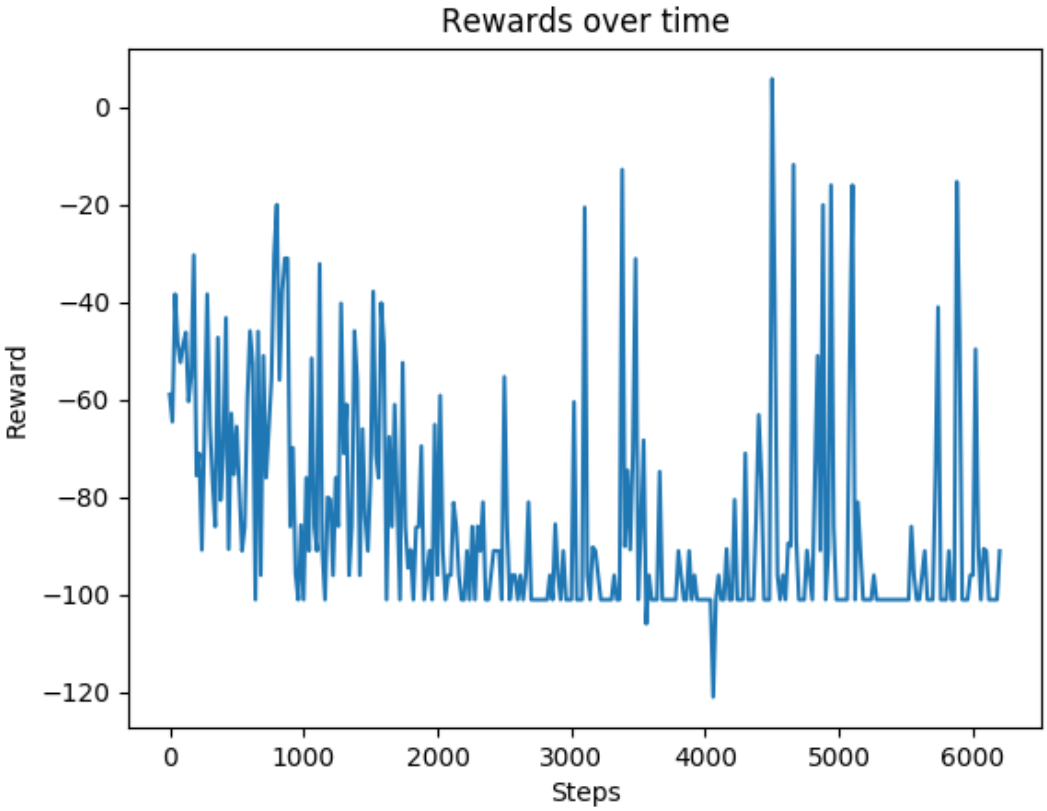


Figure 70.: Rewards given over time by fourth network, third state, first test.

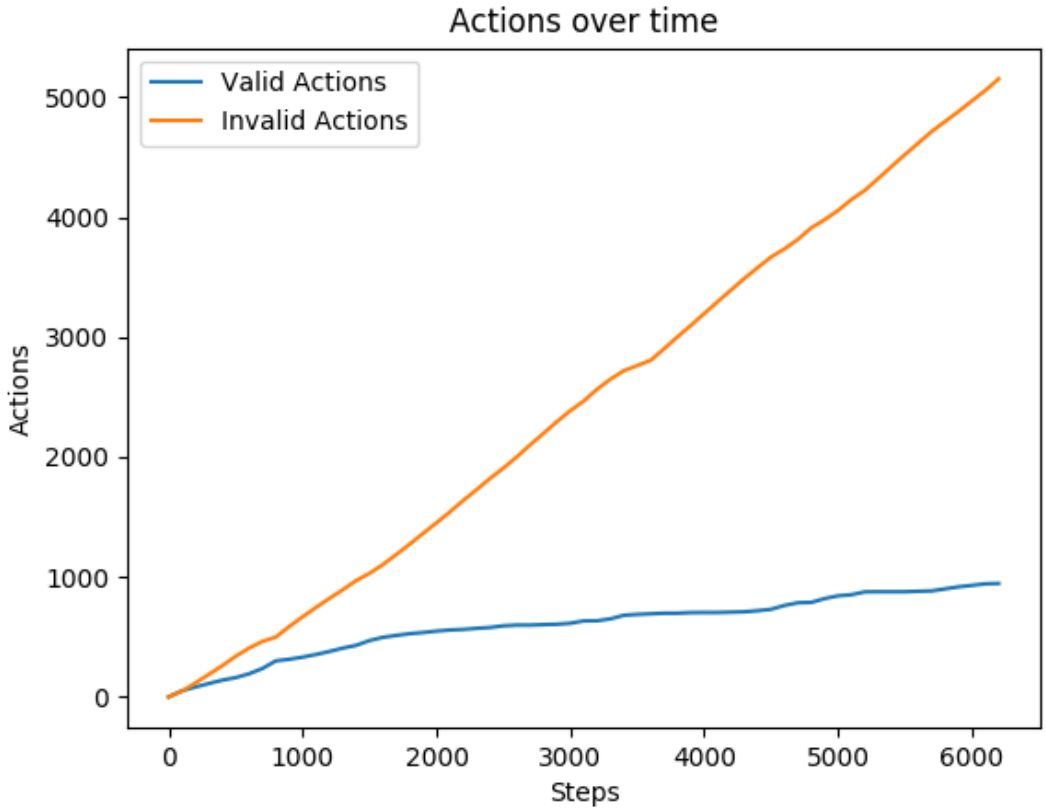


Figure 71.: Actions analysis of fourth network, third state, first test.

A.4 Fourth State Experiences

A.4.1 *Second Network Results*

First Test

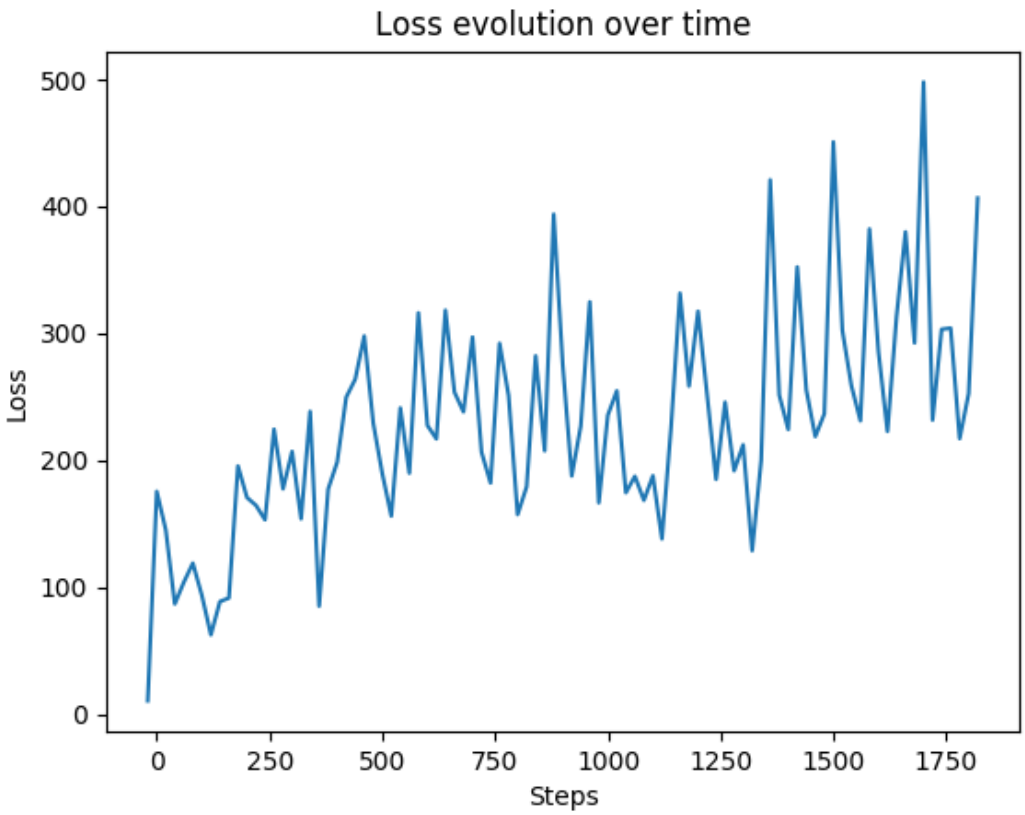


Figure 72.: Loss evolution of second network, fourth state, first test.

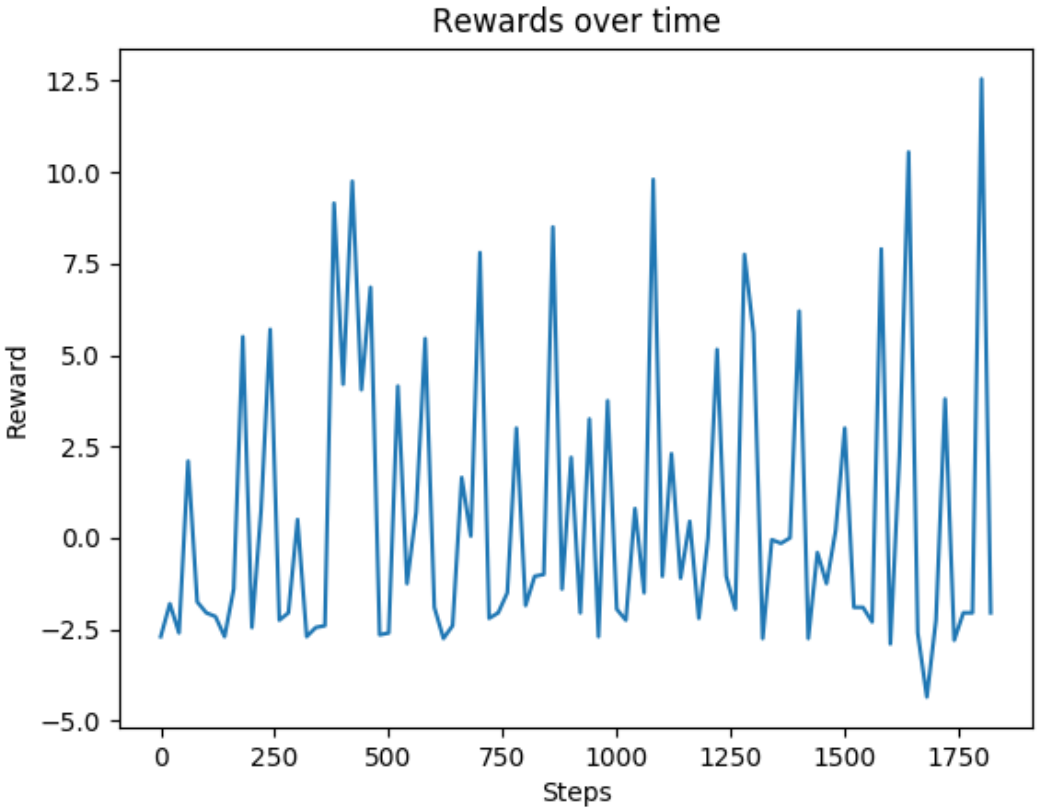


Figure 73.: Rewards given over time by second network, fourth state, first test.

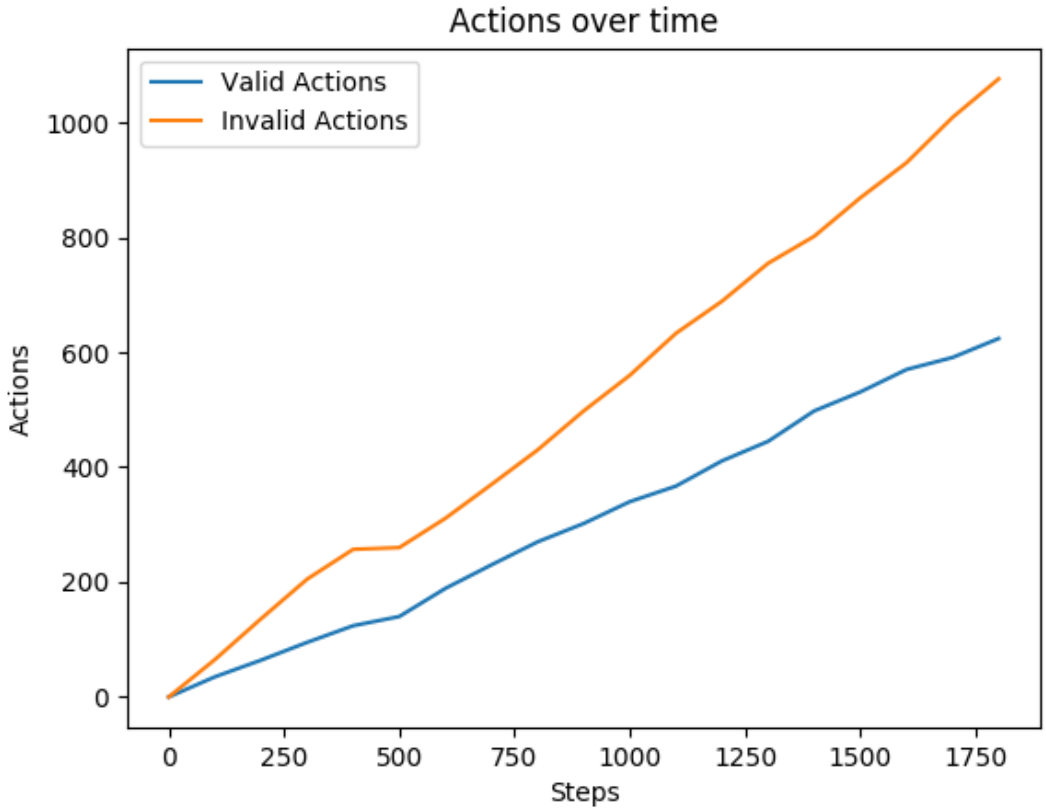


Figure 74.: Actions analysis of second network, fourth state, first test.

Second Test

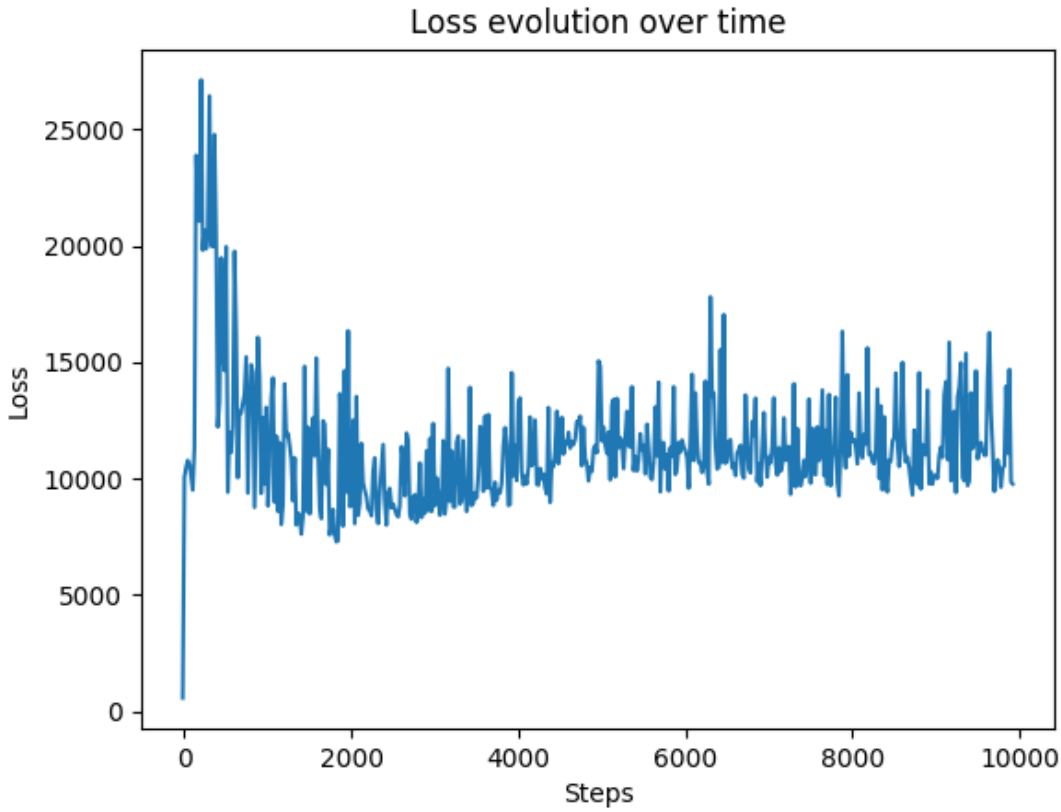


Figure 75.: Loss evolution of second network, fourth state, second test.

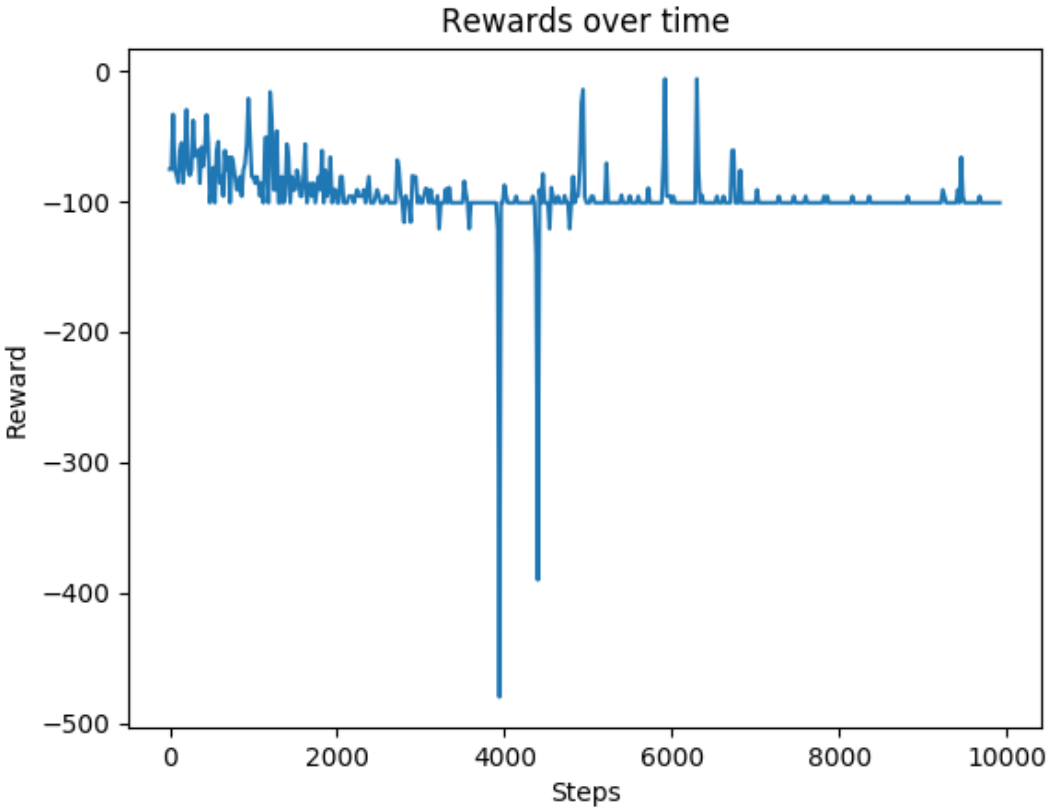


Figure 76.: Rewards given over time by second network, fourth state, second test.

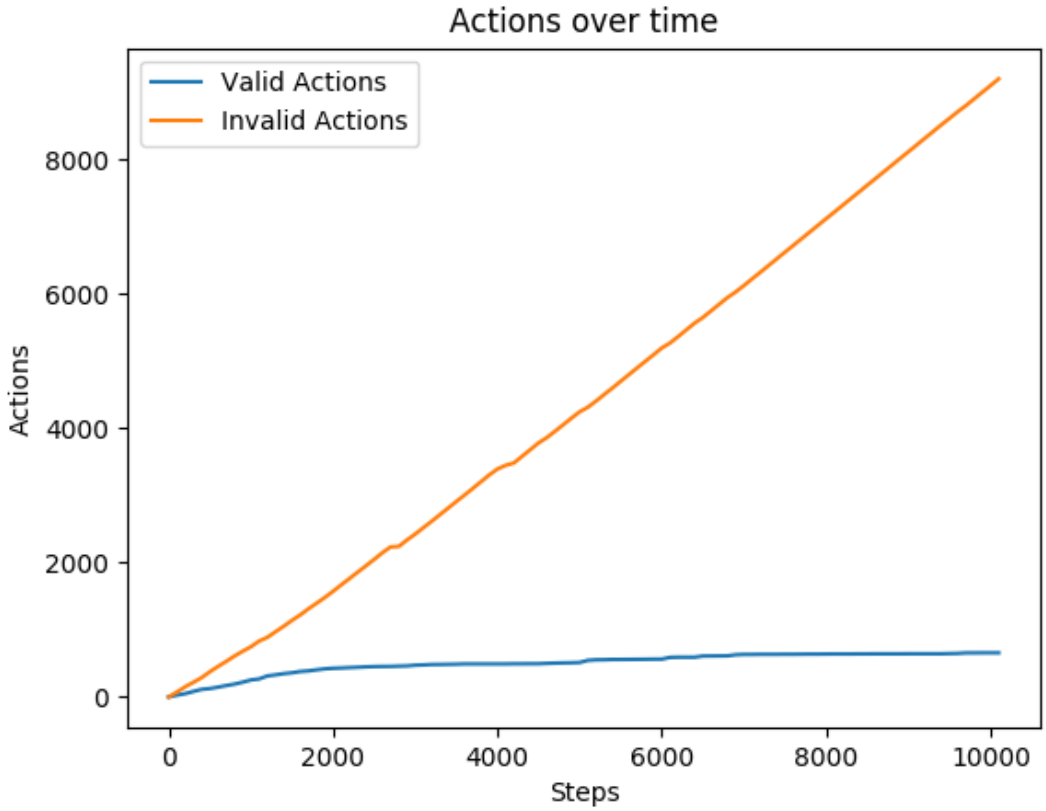


Figure 77.: Actions analysis of second network, fourth state, second test.

A.4.2 *Third Network Results*

First Test

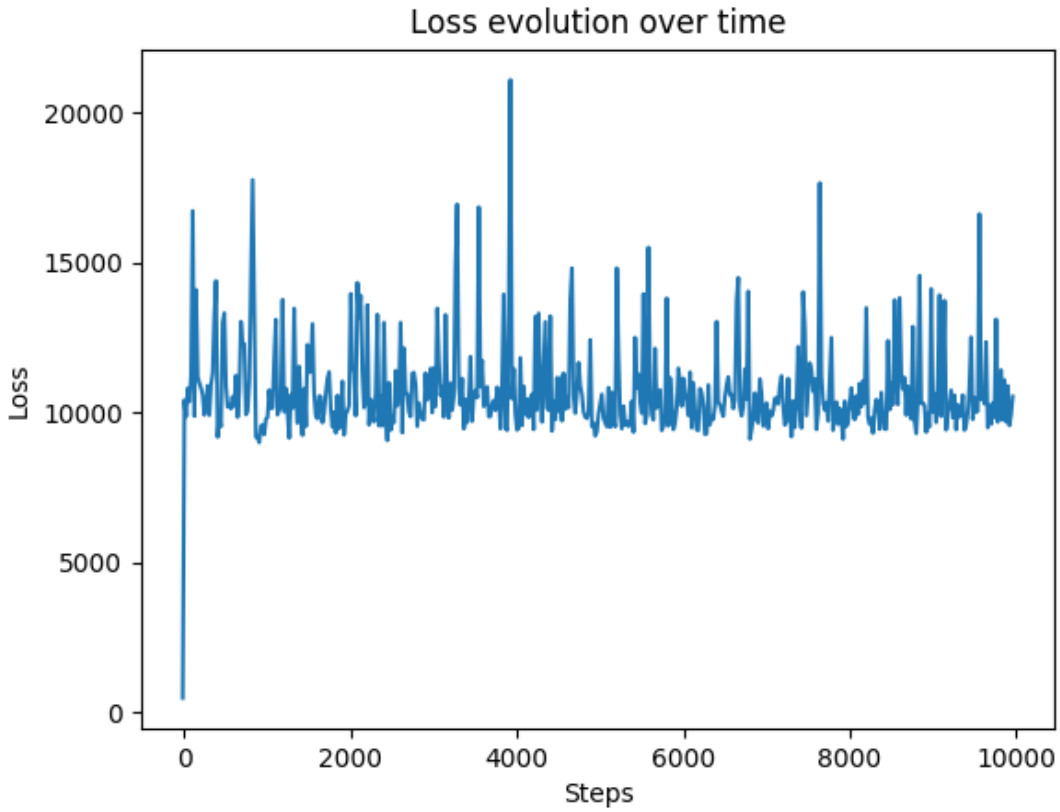


Figure 78.: Loss evolution of third network, fourth state, first test.

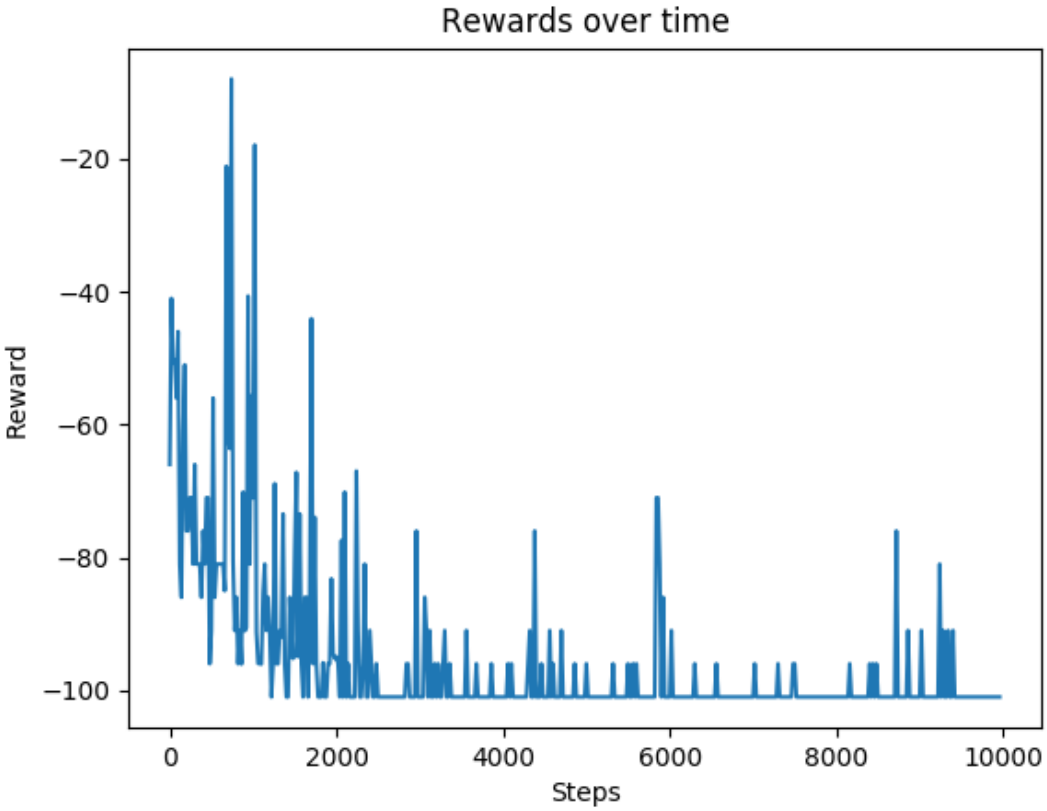


Figure 79.: Rewards given over time by third network, fourth state, first test.

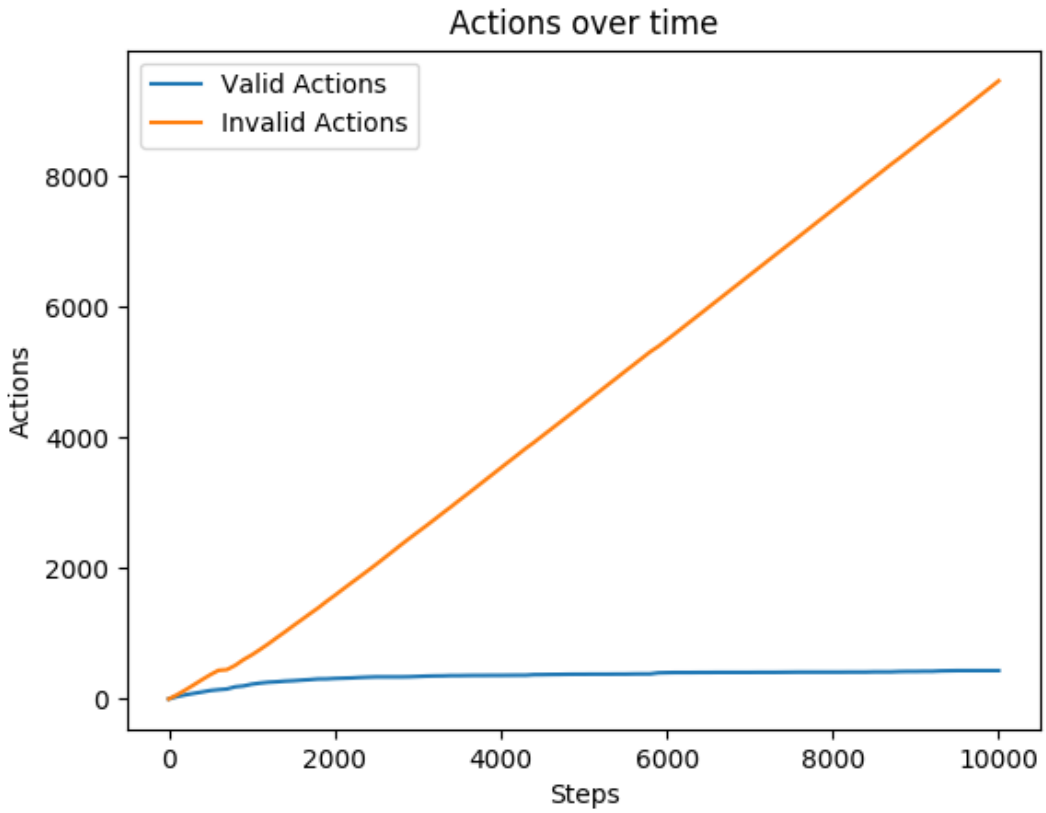


Figure 80.: Actions analysis of third network, fourth state, first test.

Second Test

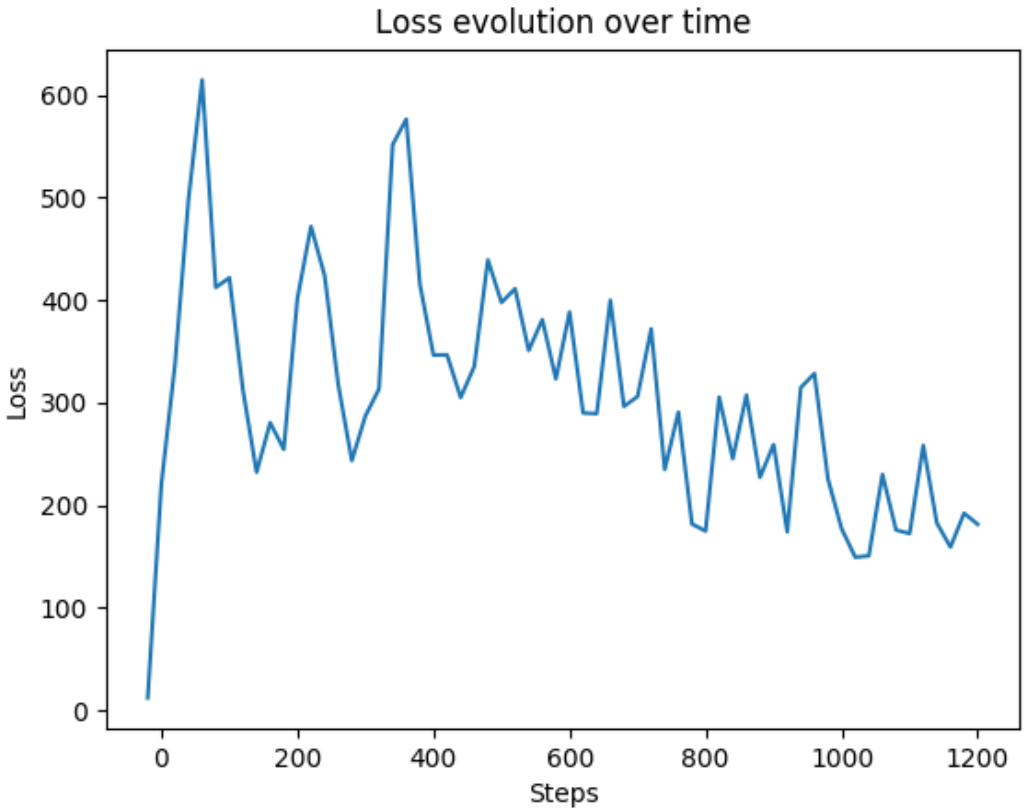


Figure 81.: Loss evolution of third network, fourth state, second test.

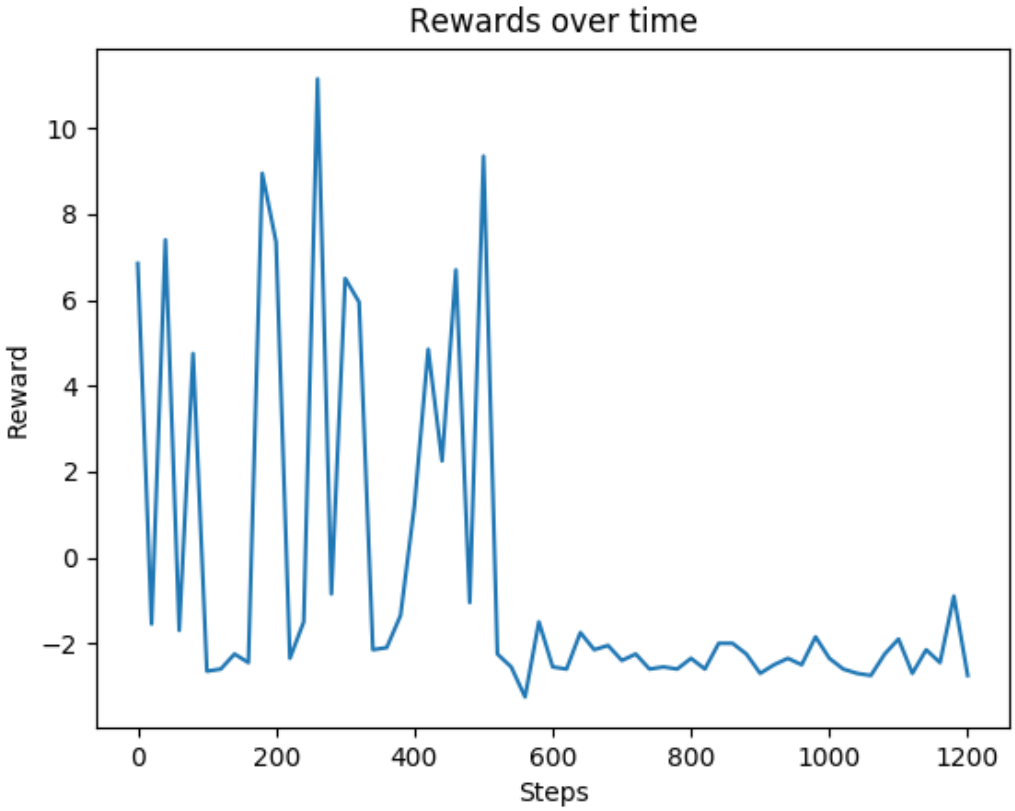


Figure 82.: Rewards given over time by third network, fourth state, second test.

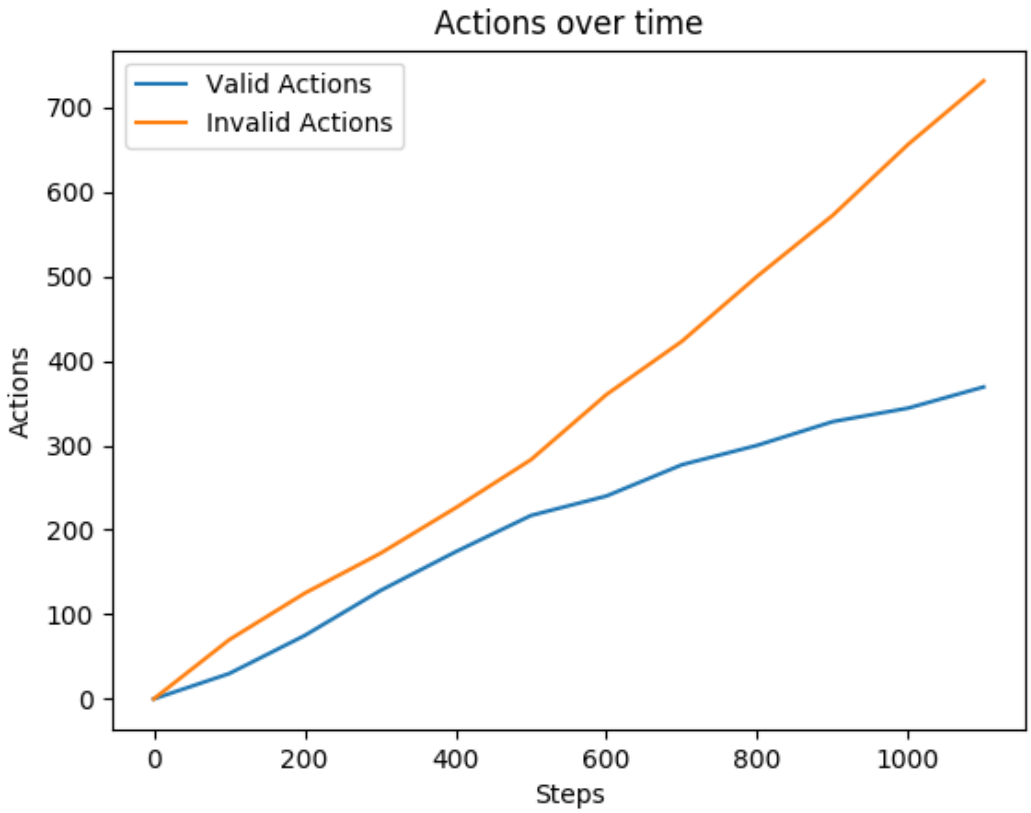


Figure 83.: Actions analysis of third network, fourth state, second test.

Third Test

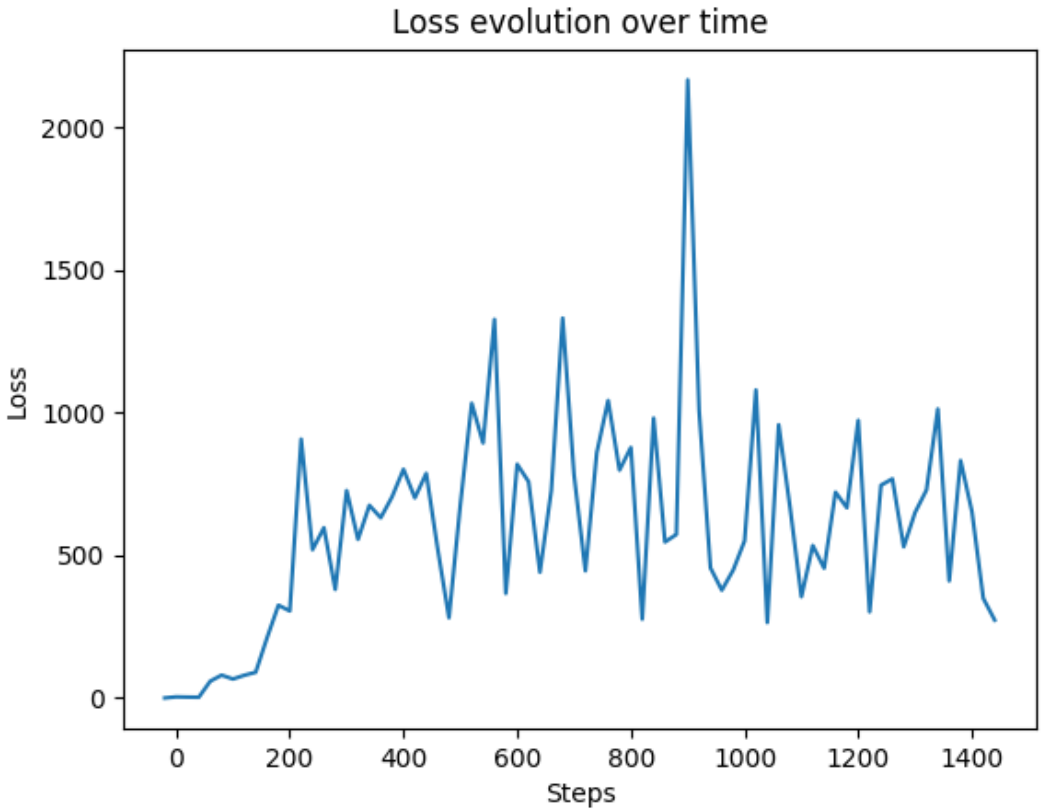


Figure 84.: Loss evolution of third network, fourth state, third test.

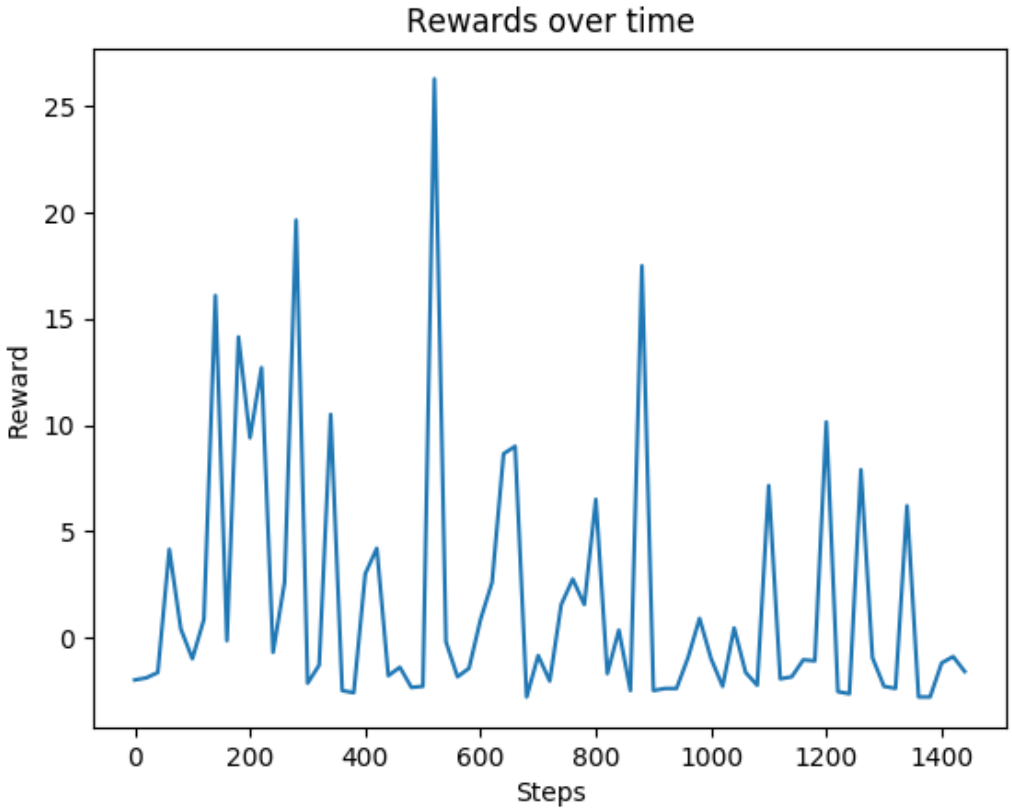


Figure 85.: Rewards given over time by third network, fourth state, third test.

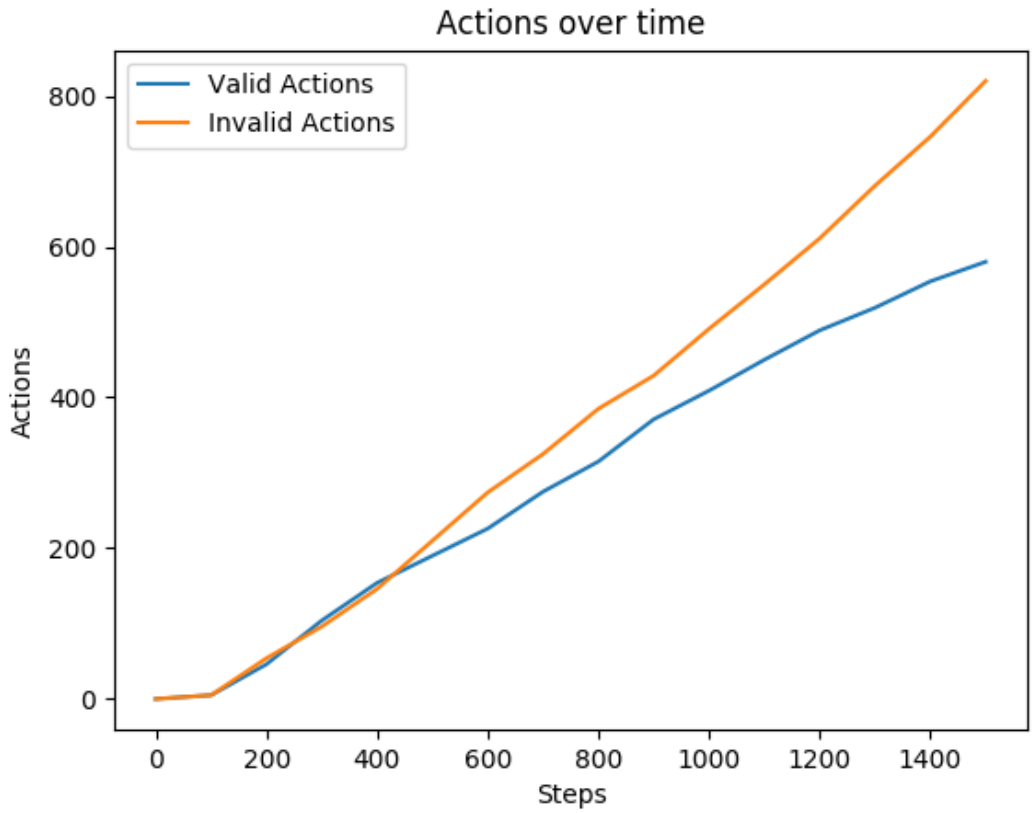


Figure 86.: Actions analysis of third network, fourth state, third test.

A.4.3 *Fourth Network Results*

First Test

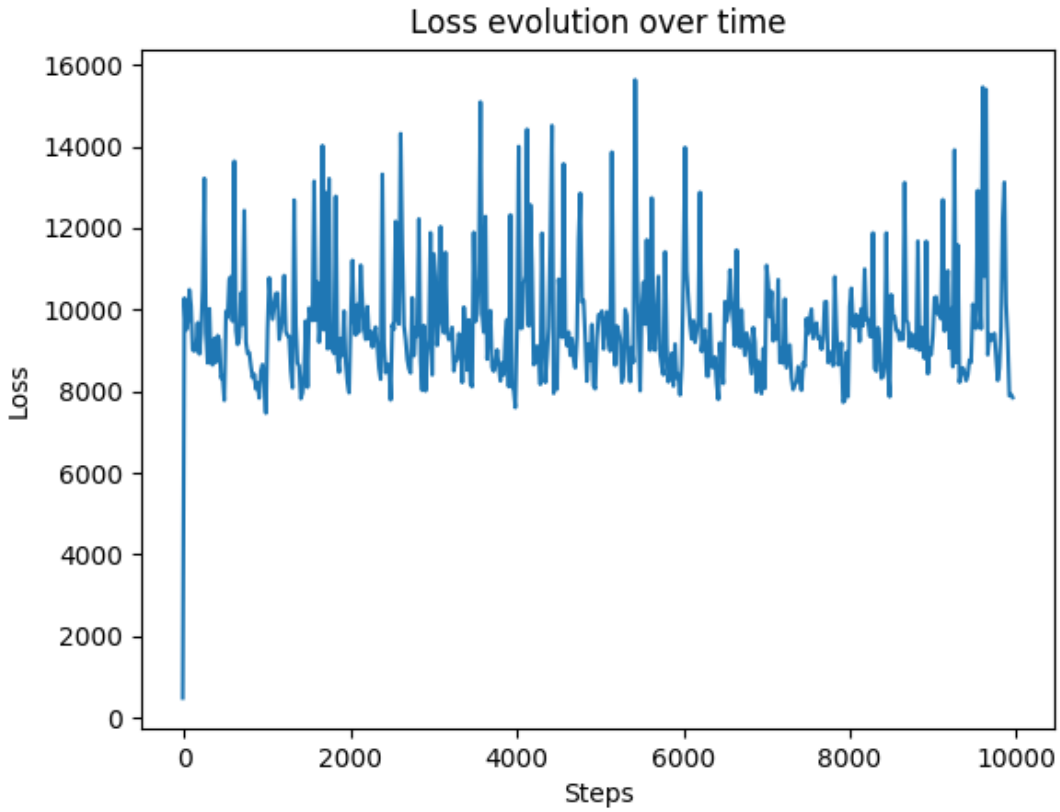


Figure 87.: Loss evolution of forth network, fourth state, first test.

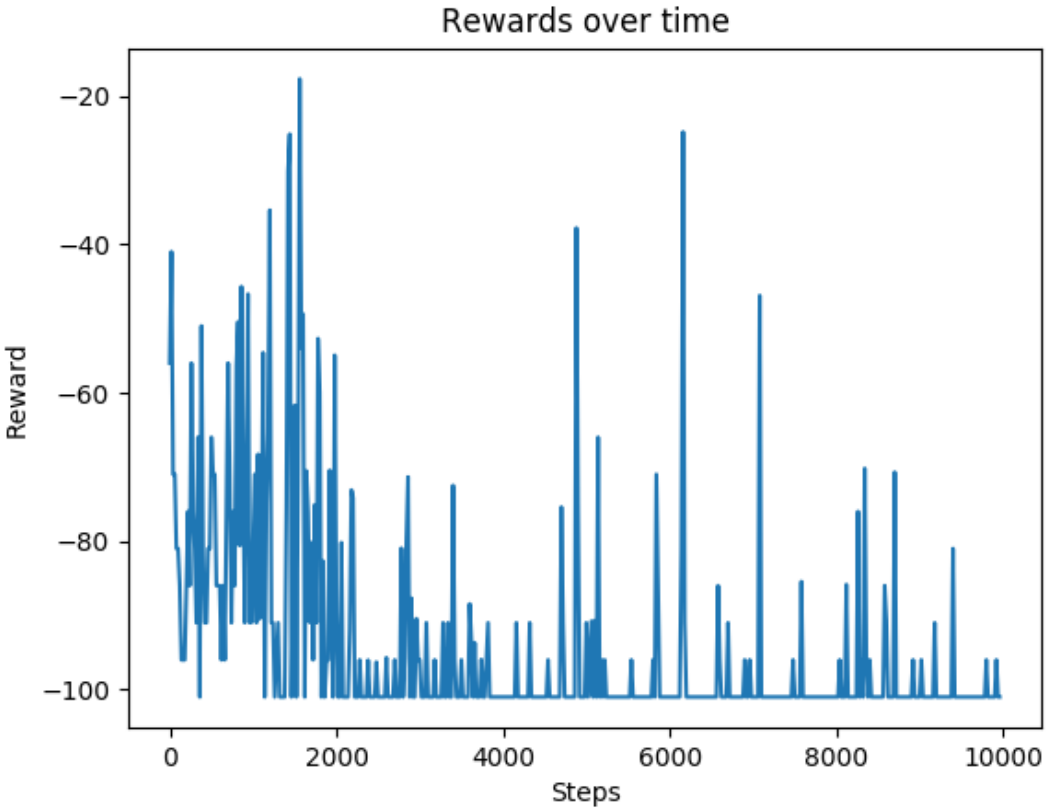


Figure 88.: Rewards given over time by fourth network, fourth state, first test.

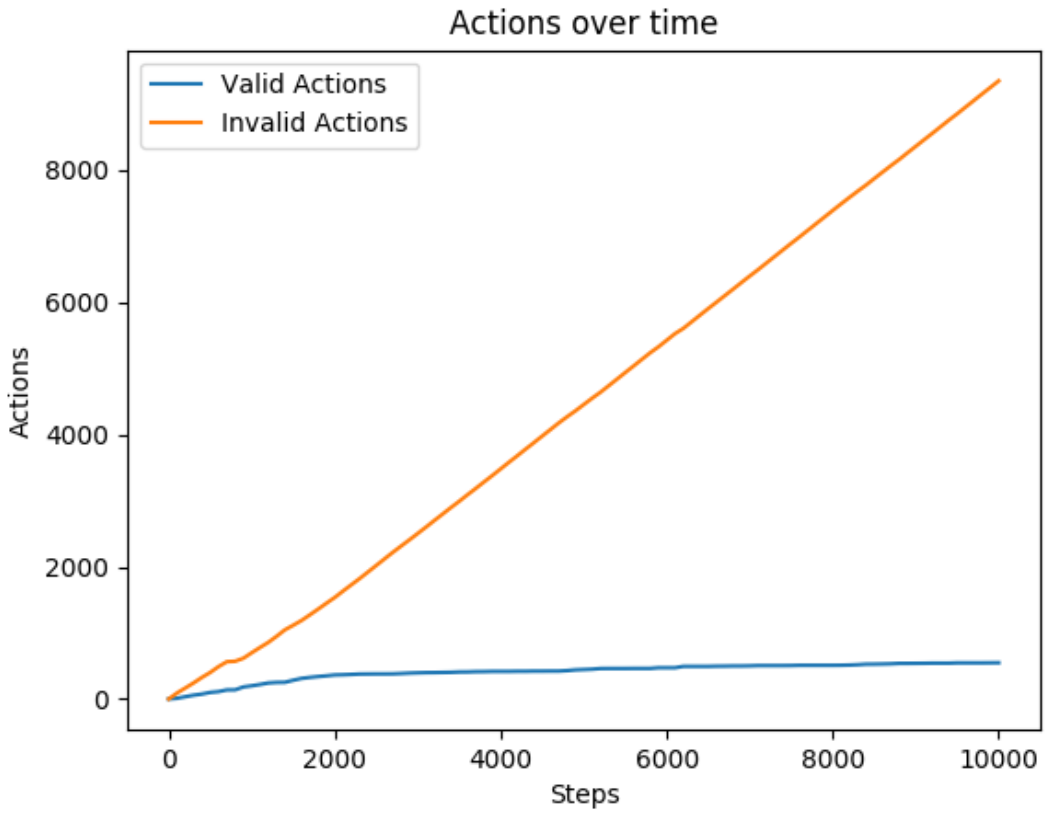


Figure 89.: Actions analysis of fourth network, fourth state, first test.

A.5 Fifth State Experiences

A.5.1 *Second Network Results*

First Test

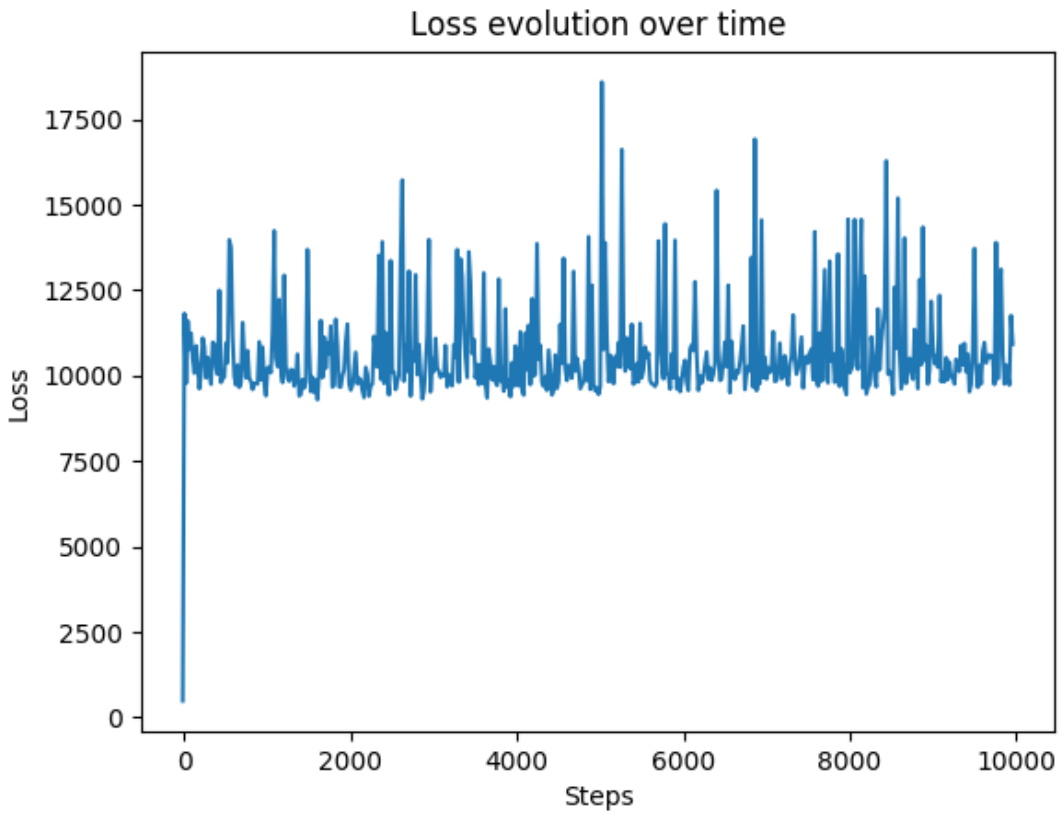


Figure 90.: Loss evolution of second network, fifth state, first test.

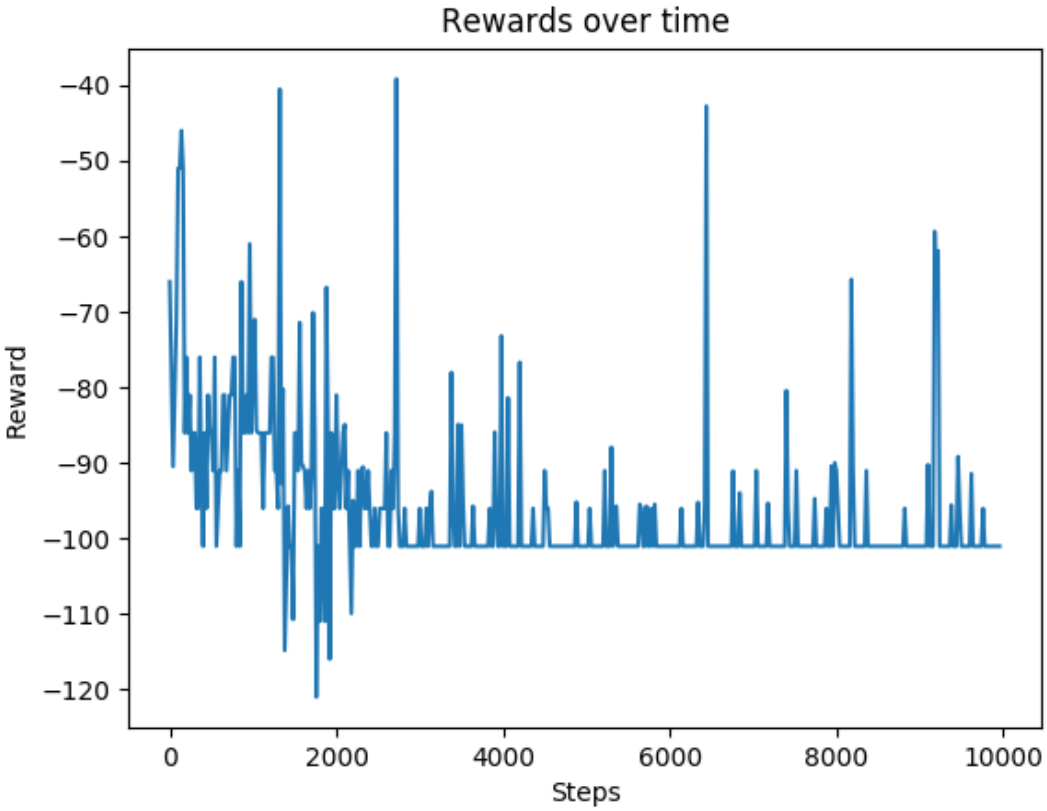


Figure 91.: Rewards given over time by second network, fifth state, first test.

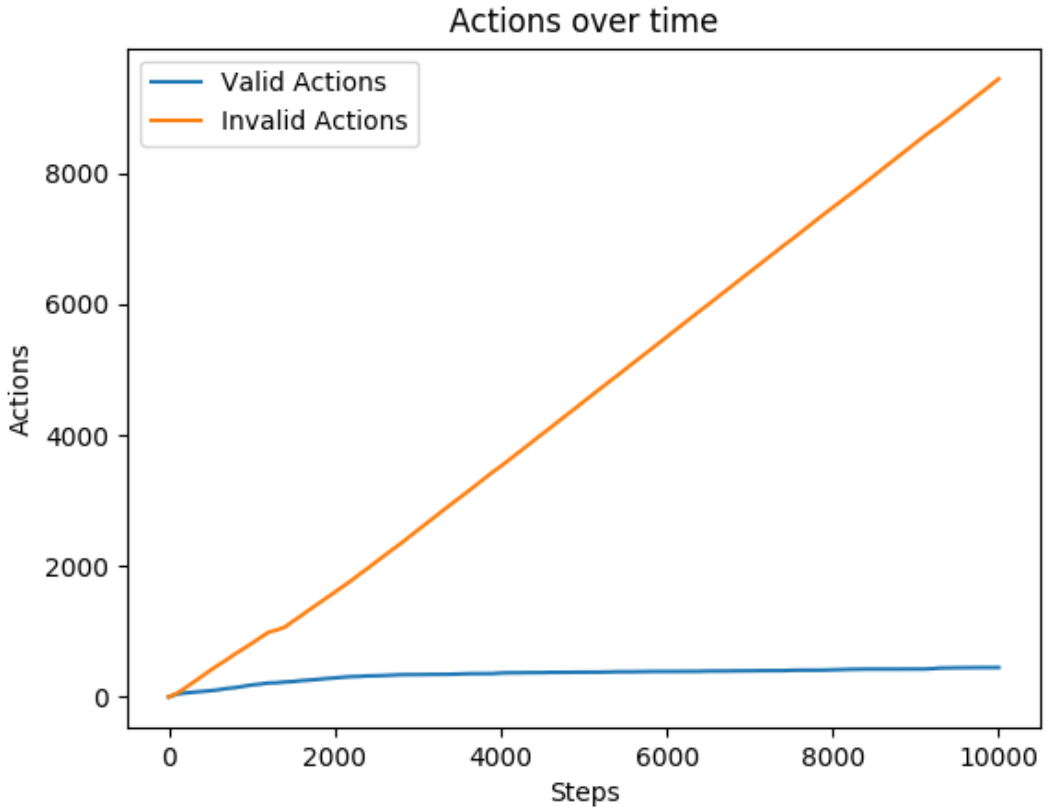


Figure 92.: Actions analysis of second network, fifth state, first test.

A.5.2 *Third Network Results*

First Test

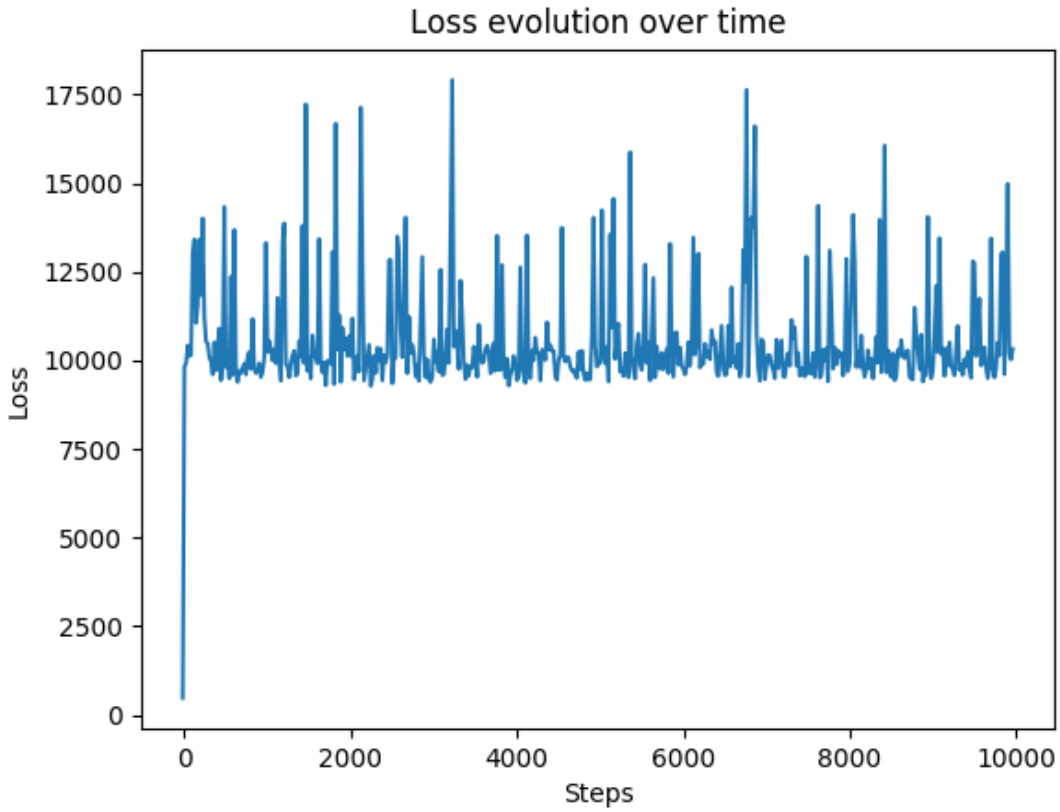


Figure 93.: Loss evolution of third network, fifth state, first test.

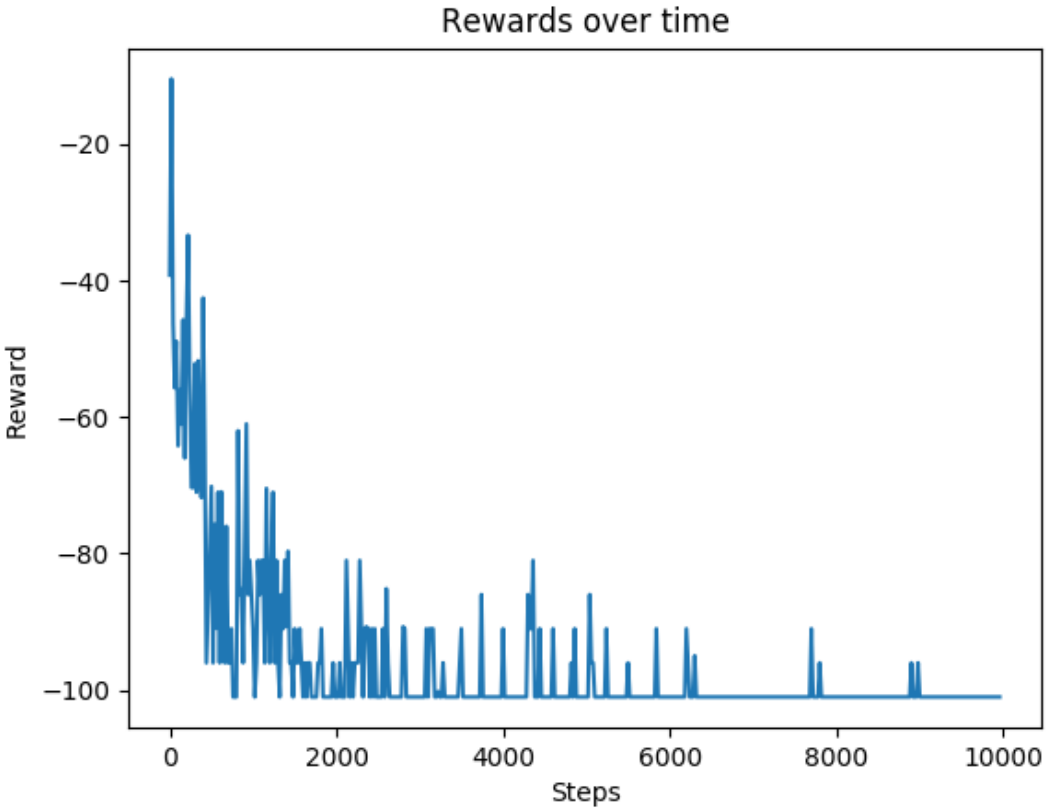


Figure 94.: Rewards given over time by third network, fifth state, first test.

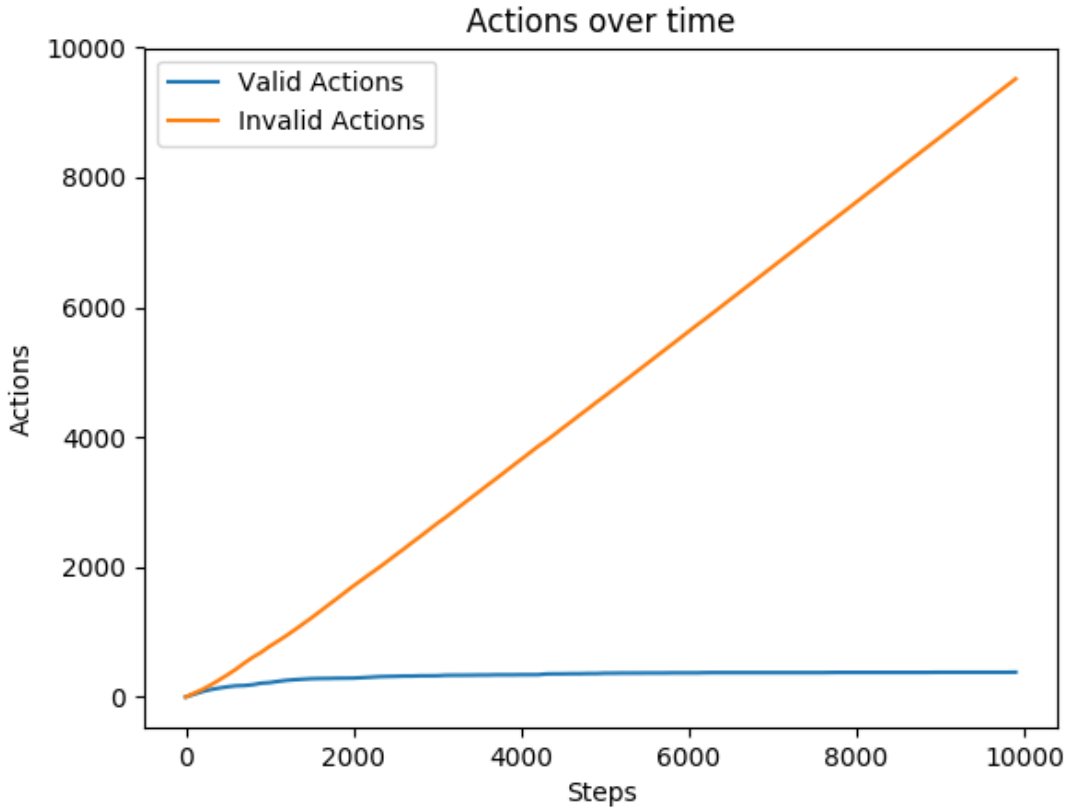


Figure 95.: Actions analysis of third network, fifth state, first test.

A.5.3 *Fourth Network Results*

First Test

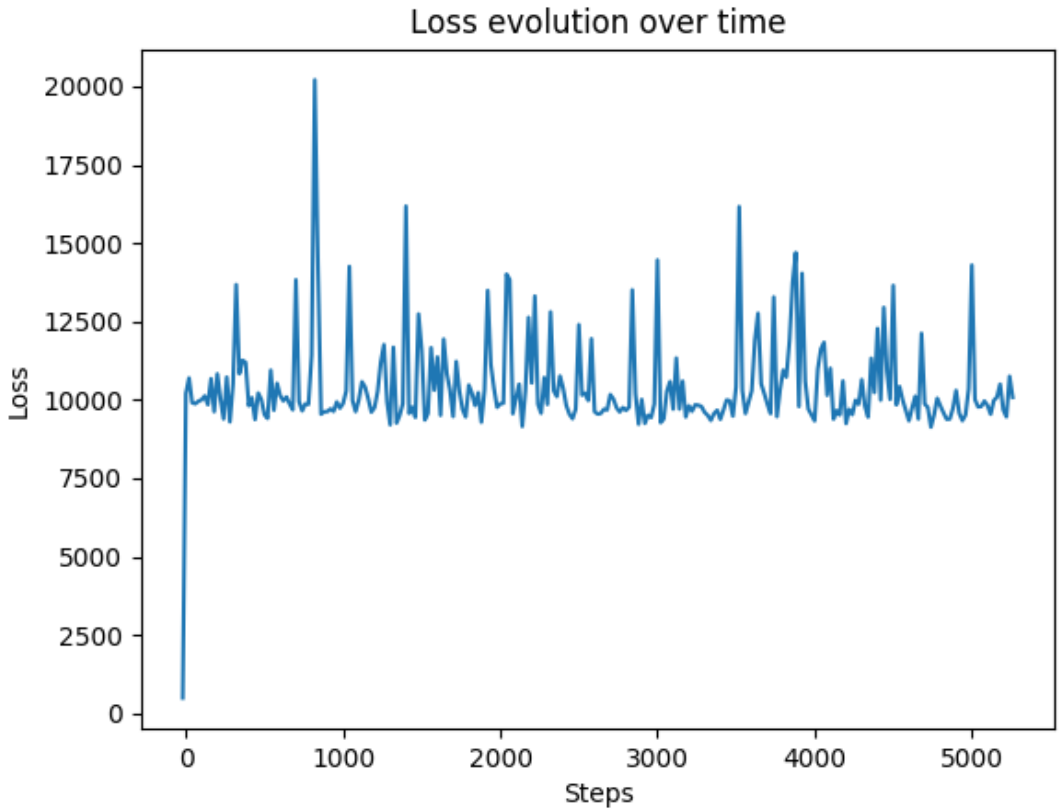


Figure 96.: Loss evolution of fourth network, fifth state, first test.

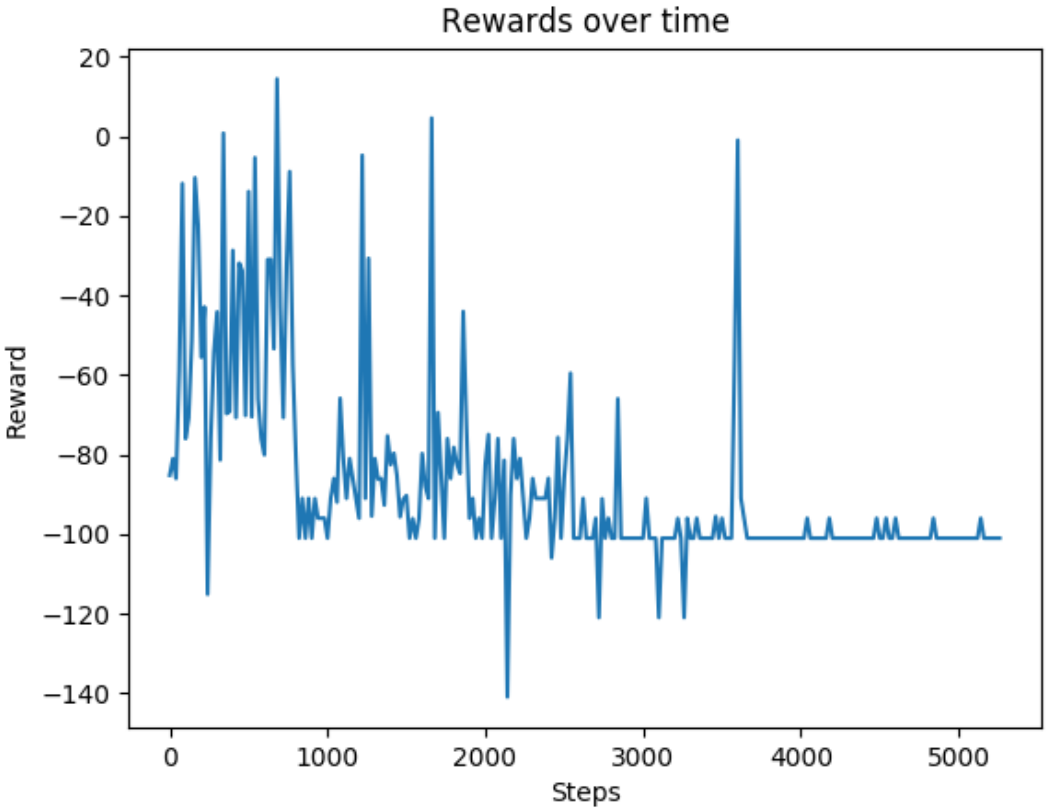


Figure 97.: Rewards given over time by fourth network, fifth state, first test.

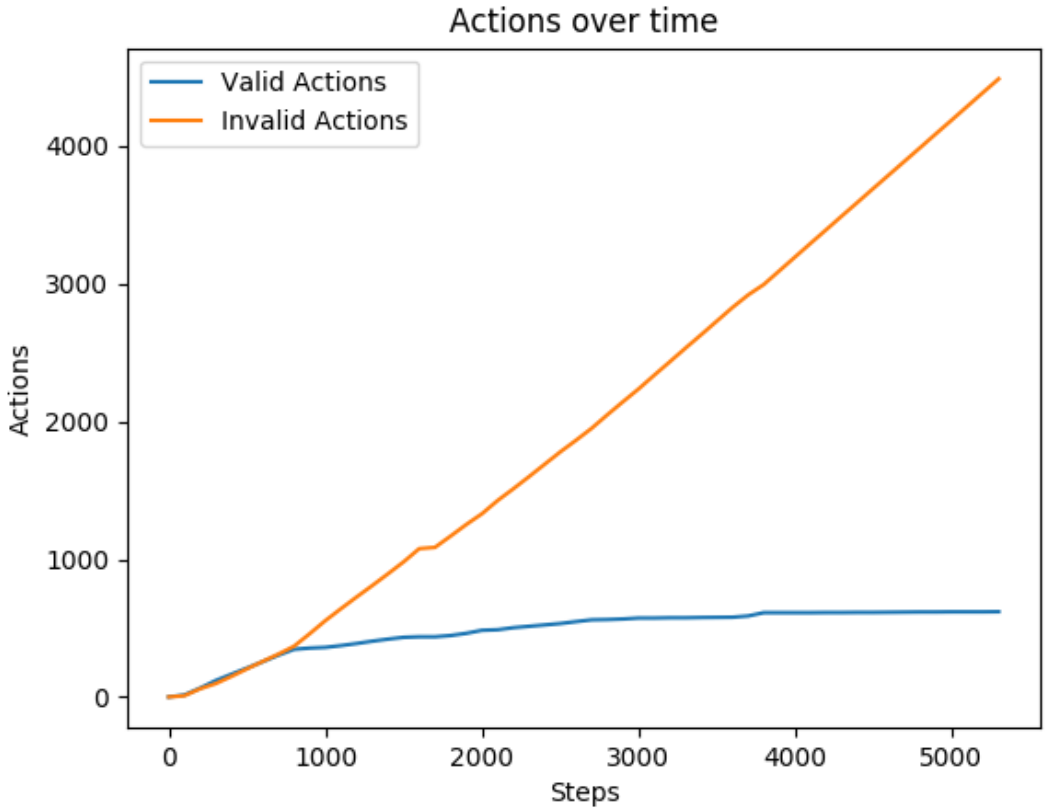


Figure 98.: Actions analysis of fourth network, fifth state, first test.