EXAMENSARBETE INOM TEKNIK,
GRUNDNIVÅ, 15 HP

STOCKHOLM, *SVERIGE 2019*

Increasingly Complex Environments in Deep Reinforcement Learning

OSKAR ERIKSSON

MATTIAS LARSSON

KTH
SKOLAN FÖR ELEKTROTEKNIK OCH DATAVETENSKAP

Increasingly Complex Environments in Deep Reinforcement Learning

OSKAR ERIKSSON & MATTIAS LARSSON

Degree Project in Computer Science
Date: June 17, 2019
Supervisor: Jörg Conradt
Examiner: Örjan Ekeberg
School of Electrical Engineering and Computer Science
Swedish title: Miljöer med ökande komplexitet i deep reinforcement learning

Abstract

In this thesis, we used deep reinforcement learning to train autonomous agents and evaluated the impact of increasing the complexity of the training environment over time. This was compared to using a fixed complexity. Also, we investigated the impact of using a pre-trained agent as a starting point for training in an environment with a different complexity, compared to an untrained agent. The scope was limited to only training and analyzing agents playing a variant of the 2D game Snake. Random obstacles were placed on the map, and complexity corresponds to the amount of obstacles. Performance was measured in terms of eaten fruits.

The results showed benefits in overall performance for the agent trained in increasingly complex environments. With regard to previous research, it was concluded that this seems to hold generally, but more research is needed on the topic. Also, the results displayed benefits of using a pre-trained model as a starting point for training in a different complexity environment, which was hypothesized.

Sammanfattning

I denna studie använde vi deep reinforcement learning för att träna autonoma agenter och utvärderade inverkan av att använda miljöer med ökande komplexitet över tid. Detta jämfördes med att använda en fixerad komplexitet. Utöver detta jämförde vi att använda en tränad agent som startpunkt för träning i en miljö med en annan komplexitet, jämfört med att använda en otränad agent. Studien avgränsades till att bara träna och analysera agenter på en variant av 2D-spelet Snake. Hinder placerades slumpmässigt ut på kartan, och komplexiteten motsvarar antalet hinder. Prestationen mättes i antal frukter som agenten lyckades äta.

Resultaten visade att agenten som tränades i miljöer med ökande komplexitet presterade bättre totalt sett. Med hänsyn till tidigare forskning drogs slutsatsen att detta verkar vara ett generellt fenomen, men att mer forskning behövs på ämnet. Vidare visade resultaten att det finns fördelar med att använda en redan tränad agent som startpunkt för träning i en miljö med en annan komplexitet, vilket var en del av författarnas hypotes.

Acknowledgements

Thank you to our supervisor Jörg Conradt, opponents Shapour Jahanshahi and Simon Jäger, and our friends Adrian Westerberg and Gustav Ung for the supportive feedback.

Contents

Chapter 1

Introduction

In this chapter, an introduction to the topic is given, followed by the purpose, the research questions and the scope of the thesis. This aims to present the context for the thesis and why it is an interesting topic.

The field of artificial intelligence is on the rise and autonomous agents display a large potential, e.g. in the field of robotics. However, creating an autonomous agent which functions well in a complex environment is not an easy task [1].

Reinforcement learning, the practice of letting the agent do the exploration itself and learning from experience, has shown impressive results for some tasks, e.g. playing computer games [2]. However, it can still be difficult for an agent to learn in complex environments where the goal is not simple.

1.1 Purpose

The purpose of this project is to investigate the impact of transfer learning in incrementally complex environments.

In real-world scenarios, e.g. in robotics applications, it could be cheaper/easier to let an agent train in a less complex environment before advancing to a more complex task. If there are benefits of using increasingly complex environments, then such a strategy could lead to reduced costs and perhaps also an overall improved performance.

Furthermore, there could be scenarios where there exists a pre-trained agent which has been trained in an environment with a different complexity. Our

hypothesis is that there can be benefits in using such an agent as a starting point, compared to starting the training from scratch in a new environment.

1.2 Research questions

- How does an agent trained in increasingly complex environments compare to an agent trained in an environment with fixed complexity, in terms of achieved performance?
- How does transferring a pre-trained agent to a different complexity environment, and performing additional training in the new environment, impact the agent's performance compared to starting with an untrained agent?

1.3 Scope

We have limited the scope of this thesis to training and analyzing agents playing the game Snake. The performance is measured in terms of the number of fruits eaten.

Snake was chosen because it can be modelled with both positive and negative rewards. It also has a finite action space, and an observation space where the agent is given perfect information. These are intrinsic features of many games making them suitable for reinforcement learning, given how well these features works with trial-and-error search and accumulation of rewards [3]. This makes Snake a reasonably general environment.

Chapter 2

Background

In this chapter, the background for the thesis is presented, with the aim of giving an understanding for the key components of the experiments.

2.1 Reinforcement learning

OpenAI [4] describes reinforcement learning as consisting of an agent acting in an environment. At every time step the agent is in a given state, and at every time step the agent performs an action. The action puts the agent in a new state, and also gives the agent a reward. This interaction loop is depicted in Figure 2.1.

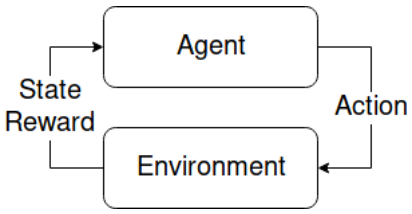

Figure 2.1: The reinforcement learning loop, an agent acting in an environment.

The agent might be able to see the entire state (i.e. the environment is fully observed) or only a partial observation of the state (i.e. the environment is partially observed) [4].

The point of the reward is to rate how good the current state is, and the agent's goal is to maximize this reward. Reinforcement learning is a collection of

methods which are used to make the agent learn how to make the best actions, which maximizes the reward based on previous experience [5].

The actions the agent can choose to perform are called the environment's action space, and different environments have different action spaces [5]. There are discrete action spaces (e.g. in Chess where the agent has a distinct number of possible moves to make) and continuous action spaces (e.g. a joint angle for a robot).

The rule used by the agent to choose between the available actions in the action space is called a policy. OpenAI [4] describes the policy as the brain of the agent. The key point of reinforcement learning is to find a policy which maximizes the expected cumulative reward, i.e. the reward the agent collects during its entire lifetime in the environment.

2.1.1 Q-Learning

OpenAI [4] describes the action-value function $Q^{\pi}(s, a)$ as the expected cumulative reward of taking the action a when the agent is in state s, and then acting according to the policy π after that action. That is,

$$Q^{\pi}(s, a) : S \times A \to \mathbb{R}$$

where S is the set of all possible states, A is the set of all possible actions, π is the policy that the agent will follow after the action has been taken, and the output is real-valued and gives the expected cumulative reward.

According to Sutton and Barto [5], Q-learning is a reinforcement learning method which was developed in 1989 and one of the early breakthroughs in the field. If the optimal action-value function was available, then we would know how to act in a way which would maximize the expected cumulative reward, which is the goal of reinforcement learning. The purpose of Q-learning is to find an approximation of the optimal action-value function.

2.2 Neural networks

A neural network is a function approximator which consists of a series of layers of neurons. The first layer is called the input layer and takes the input to the

function. Then there is a collection of hidden layers, the first one receives the output of the input layer, the second hidden layer receives the output of the first hidden layer, and so on. Finally, there is an output layer which outputs the result of the computation [6].

The neurons between the layers are connected with certain weights. The output of a neuron is calculated by going through each neuron in the previous layer and summing up its output multiplied by the weight of the connection. Finally, a bias value is added, and the result is passed through an activation function. This gives the output which is sent to the next layer [7].

According to Csáji [7], a neural network with a finite number of neurons and at least one hidden layer can approximate any continuous function (on subsets of $\mathbb{R}^n$ which are compact). This fact, called the universality theorem, makes neural networks very powerful approximators. Even though the universality theorem tells us that a neural network with a single hidden layer can compute any function, Nielsen [6] claims that in practice it is often useful to use deep neural networks, i.e. neural networks with more hidden layers, to solve real world problems since they can "understand" more complex concepts. An example is image recognition where the input layer takes in the values from the raw pixels, while the hidden layers might recognize more complex concepts like edges and different shapes.

2.3 Convolutional neural networks

Nielsen [6] claims that a specific type of neural network, called a convolutional neural network (CNN), is especially useful when it comes to image recognition. A CNN is built up of convolutional layers, meaning neurons which only process data in a local receptive field (a subset of neurons in the previous layer). CNNs also include fully connected layers just as in regular neural networks and can also have other kinds of layers which performs certain operations, such as pooling. Pooling further reduces the number of parameters that needs to be trained [8].

According to Nielsen [6], an important aspect of CNNs is the local receptive field. A single neuron in a hidden layer may only be connected to a couple of adjacent neurons in the previous layer, and these neurons are called the local receptive field of the hidden neuron. This is also called a filter, which can have different sizes depending on the configuration of the model. How much the

local receptive field is moved when looking at the next neuron is called the stride, which also depends on the configuration of the model.

2.4 Deep reinforcement learning

According to OpenAI [4], deep reinforcement learning is a type of reinforcement learning which use so-called parameterized policies. A parameterized policy is a policy that depends on tunable parameters, which leads to a change in the policy's behaviour, and the tuning is performed by using an optimization algorithm. That is, the policy should be optimized such that the agent gains the maximum reward. A parameterized policy often consists of a neural network, which means that the weights and biases of the network are the parameters being adjusted.

2.4.1 Deep Q-Network

In 2013, Mnih et al. [9] proposed a novel approach of training neural networks using deep reinforcement learning. This approach is called Deep Q-learning, since the purpose is to approximate the action-value function using a neural network. They demonstrated this approach by training CNNs to play Atari games with the raw pixels of the game as input to the network. This type of network is called a Deep Q-Network (DQN). Mnih et al. [9] concluded that the DQN performed well in six of the seven Atari games that were tested.

In 2015, a group consisting of people from the same team of researchers (and others) presented a study where a DQN was used to achieve performance comparable to a human player in 49 Atari games. This exceeded the performance of any previously seen algorithms at the time [2].

2.4.2 Proximal Policy Optimization

Another approach to deep reinforcement learning is policy optimization methods. As opposed to Q-learning, policy optimization is not based on the action-value function. Instead, policy optimization is based on representing the policy as a function $\pi_\theta(a|s)$ that depends on some parameters θ, which should be tuned to optimize the policy [10].

For a stochastic policy $\pi(a|s)$, there is a certain probability that each action is taken in each state. This function returns the probability of action a being taken in state s [5].

In 2017, Schulman et al. [11] proposed a new policy optimization method for training in deep reinforcement learning called Proximal Policy Optimization (PPO). According to Schulman et al. [11], DQN does not perform well in environments with continuous action spaces and there are issues with other policy optimization such as bad data efficiency and complicated algorithms. However, PPO has a good balance between performance, data efficiency and being simple to implement.

The version of PPO that Schulman et al. [11] proposes is an actor-critic approach, which means that there are two components in the algorithm. One component is the actor, which acts in the environment according to the current policy. The other component is the critic, which is scoring the actor's actions. According to OpenAI [12], this is used for advantage estimation, which basically means checking if the performed action gave a better reward than the average expected reward in the current state.

Note that the critic is related to the action-value function $Q^{\pi}(s, a)$, i.e. it maps a state s and an action a to some expected reward. Both the actor and the critic can be represented with neural networks.

The simplicity of PPO comes from the main objective function

$$L^{CLIP} = \mathbf{E}_t\Big[\min(r_t A_t, \text{clip}(r_t, 1-\epsilon, 1+\epsilon)A_t\Big]$$

where $r_t = \frac{\pi(a_t|s_t)}{\pi_{old}(a_t|s_t)}$ is the probability ratio, i.e. the ratio between how probable a given action is according to the current policy and the old policy. $\mathbf{E}_t$ is the expected value, and A_t is the advantage estimates. The subscripts t says that this is for time step t. The clip-function clips the first argument between the second and third argument, the reason for this is to make sure that the update to the policy is not too large, i.e. we only want to take small steps [11]. High-level

pseudocode for PPO based on OpenAI [12] is presented in Algorithm 1.

algorithm *PPO*:
  while *iterations remaining* **do**
    Run the old policy π_{old} in the environment
    Compute the rewards
    Compute the advantage estimates A_t
    Find a new policy by maximizing L^{CLIP} (update actor)
    Update the action-value function (update critic)

Algorithm 1: Proximal Policy Optimization (PPO)

2.5 Transfer learning

According to Torrey and Shavlik [13], transfer learning means improving the learning process of a new task using already learned experiences from other environments. This can be particularly useful in cases where there is a lack of training data for the target problem. In these cases, training in a source environment where there exists much training data and then transferring to a target environment could be a viable solution.

An example where transfer learning could be used is in reinforcement learning, where the starting point of a new model could be an old model. This could lead to an improvement of the learning processes, since something that was learned in the old model could be applied to the new problem, and therefore does not need to be re-learned. The improvement could mean a jump start in how the trained model performs in the target environment, without having started training on it. There could also be improvements in how the model behaves asymptotically. The existence and magnitude of a jump start and the model behaviour in the asymptotic case are some of the ways the benefits of transfer learning can be measured [14].

Chapter 3

Related work

In this chapter, related work is presented. This aims to give an understanding for what has been done before and what our thesis builds upon.

3.1 Snake and reinforcement learning

There have been many studies done on how to train agents on the game Snake with reinforcement learning. For example, Ma, Tang, and Zhang [15] used a straightforward neural network and chose a specific feature vector as the input. Furthermore, there are versions using Q-learning as demonstrated by Ovidiu Chelcea and Ståhl [16]. There are also examples of deep reinforcement learning where CNNs are used, and thereby taking in the visual pixel-based representation of the game as input [17]. Mostly these examples are only looking at playing the most basic implementation of Snake, but there are also cases where research has been done on more complex Snake environments. For example, Örnberg and Nylund [18] did research on their own version of Snake, where they added randomly placed obstacles to the game.

All of the above-mentioned examples has been shown to succeed in the way that the agents learned to play Snake. Hence, there are multiple different methods that are applicable for training a skillful Snake agent.

3.2 Increasingly complex environments

Örnberg and Nylund [18] also looked at incrementally increasing the size of the map, thereby transfer learning from simpler to more complex environments, and compared it to just training on the full-scale map directly. Their results showed that their incremental method performed just as good as the conventional method and in some cases, where the learning rate was larger, even better. This could point to it being more efficient to use this kind of incrementally complex learning, instead of just training on the fully complex environment straight away. Our research builds upon the concept of increasing complexity, but instead of varying the size of the map, we vary the number of obstacles on the map.

Wang et al. [19] introduced an algorithm called Paired Open-Ended Trailblazer (POET), that "pairs the generation of environmental challenges and the optimization of agents to solve these challenges". This aims at automatically creating the training curriculum in order to improve learning. They showed that agents which are trained using transfer learning from a simpler environment to a more complex environment, and then back to the first one, can end up performing better than agents only trained in the original environment (given equal amount of training between the two runs). Our research complements this study since we use a different deep reinforcement learning method (PPO) and another environment (Snake) in our experiments. We also further investigate the impact of a fixed training curriculum and compare this to using no curriculum.

Chapter 4

Method

In this chapter, the method used is described. This aims to give an understanding of what we have done, and how we obtained our results.

We wanted to investigate the impact of using increasingly complex environments compared to a fixed complexity environment. Also, we wanted to investigate the impact of using a pre-trained agent as a starting point in a different complexity environment, compared to using an untrained agent. To do this, we conducted experiments where we trained agents on the game Snake using the deep reinforcement learning method PPO, and compared the agents to each other. We chose PPO because we could not find any previous research where it has been used in environments with increasing complexity.

Initial experiments were performed to find a suitable configuration for the models, with a satisfactory balance between training time and performance. The same configuration was used for each of the models. We have not seen any evidence that the configuration would impact the comparison between the trained agents in any way.

4.1 Snake

For all our experiments, we chose to look at a variant of the game Snake, where the player plays as a snake that moves around in a 2D environment trying to collect as many fruits as possible, while avoiding obstacles and the player's own tail. For every fruit, the tail of the snake grows in size by one square, and

it becomes harder and harder to avoid the obstacles. The game is illustrated in Figure 4.1.

Figure 4.1: The snake game on a map with obstacles. The S is the snake's head, the X is the fruit, the @:s are the body of the snake and the #:s are the obstacles.

We chose the game Snake because it has several interesting aspects. There is a visual aspect, i.e. from a visual representation of the entire game world the player needs to identify obstacles (object detection), where the snake itself is (self-awareness) and where the fruits are (goal). These aspects are intrinsic features of many games, and also present in many real-world scenarios.

Furthermore, there is a clear goal to the game, i.e. collecting as many fruits as possible, and the player needs to be aware of the location of the fruits. There is also a clear terminal state, a state where the game ends, which is when the player collides with an obstacle. There is also a finite amount of actions which the player can choose from, these are moving up, down, left or right. Having a finite action space, as well as a finite state space, is common in many games including this one.

4.1.1 Complexity

We define the complexity of an environment as the number of obstacles on the map. For every environment, the outer walls are always covered with obstacles like in classic Snake. More complex environments also have obstacles within the level itself. This means that there are more obstacles for the player to collide with, and thus survival in the game becomes more challenging.

We compared an agent trained in an empty environment (no obstacles except for the outer walls), an agent trained in a complex environment, and an agent in an increasingly complex environment, i.e. trained on increasingly complex maps as training proceeds.

4.1.2 Actions

At each timestep, the agent is able to choose between four different actions. These actions correspond to moving the snake's head in either of the directions up, down, left or right. If the agent tries to move the head in opposite direction of where the snake is currently moving (that is, moving the snake's head back to the position where it was located in the previous timestep), the action is ignored. From our experience with other Snake games, simply ignoring the action is the most common way to handle this type of action.

On the first move, the snake does not yet have a tail, and therefore all directions are available. This means that the agent itself is able to choose its initial direction.

Machado et al. [20] recommends using so called *sticky actions* when training agents on these types of games. This means that a form of stochasticity is introduced, and this in turn leads to a more robust policy. According to Machado et al. [20], sticky actions are recommended in deterministic games such that the agent does not simply "memorize" a good sequence of actions which is repeated every game. This was implemented by ignoring the agent's decision with a probability of 10% each time, and thus letting the snake keep moving in the direction that it is currently going. This percentage was determined from initial experiments.

Another approach we considered was a three-action procedure, where the agent could turn clockwise, counter-clockwise or do nothing. However, this would require the agent to have information about the current direction of the snake. The four-action approach does not rely as much on this information, since the snake's tail can give the snake information about its direction.

4.1.3 World

The game world is a grid of size 12×12 (called the map), where each square can contain either an obstacle, the snake's head or a fruit. Note that the snake's tail is considered as an obstacle at each timestep.

For every game world, the outer edges of the map are set to obstacles. This is illustrated in Figure 4.2.

In Figure 4.3, a more complex map is depicted. The map in this figure has a complexity of 0.09, meaning that obstacles are randomly placed on 9% of the

Figure 4.2: The empty map.

available squares (not counting the outer walls).

A constraint on the placement of obstacles are that there can be no obstacles adjacent to each other (also not diagonally). This is because we want to avoid trapping the snake or fruits, which could lead to undesirable effects when training the agent. Placing the obstacles according to these constraints always allows the snake to be able to eat fruits without being forced to collide with an obstacle afterwards (except if the snake is forced to collide with its own tail). We tried generating maps by placing obstacles fully at random, but this led to "traps" which was an unwanted aspect of the game, and therefore we switched to the aforementioned method.

After the map has been generated, the starting point of the snake is placed randomly on an available square, and the first fruit is placed on an available square. When a fruit has been eaten, the fruit is removed, and another fruit is spawned on an available square. This means that there is always exactly one fruit on the map at any given timestep. The reason why we used randomly generated maps was to create versatile agents that performs well on any given map.

Figure 4.3: A map with complexity 0.09.

4.1.4 Observation space

Egorov [21] proposed a way to perform deep reinforcement learning in a 2D environment. In this approach, the environment is considered a 2D space and is split up into different channels, each one represented as a 2D image. The channels could be thought of as different colours of the pixels in an image. Egorov [21] used four different channels, the background channel which contains obstacles in the environment, the opponent channel which contains all the opponents, the ally channel containing information about the allies in the environment and finally the self channel contains information about where the agent is located. We used this approach since Egorov [21] experienced satisfactory results using channels.

We modified this approach to fit our experiments. Our obstacle space consists of an obstacle channel containing information about every obstacle that the player must evade, including the player's own tail. We also have a fruit channel, where information about the fruit is contained. Finally, we have the same type of self channel that Egorov [21] used.

All of these channels are 12×12 "images", where each pixel can have a value of 0 or 1. If an obstacle is placed on a given square, the pixel corresponding to this square will be 1 in the obstacle channel. The same goes for the fruit, which is represented in the fruit channel, and for the snake's head which is represented in the self channel. This means that the agent gets information about the entire game world at each decision.

We chose this representation of the game because it is very general and can be used for any snake game (the image of the game needs to be pre-processed to our format first). Also, if we were to use some other features to send to the agent, these would have to be engineered by us. This could for example be the distance to the fruit, the distance to the nearest obstacle etc. We did not want to engineer any of these features, but instead let the agent figure out by itself what important aspects of the game there are, making the chosen representation suitable.

4.1.5 Initial tail length

We also conducted some experiments with the initial tail length of the Snake. We found that a tail length of zero gave the best final performance after training, so we used this tail length for training.

However, when we evaluated the models, by letting them play the game and saving their scores, we used a tail length of one. The snake can use this information to know in which direction it is going, since its tail would be on the square the snake's head was on at the previous timestep.

4.2 Learning

In this section, the learning process is described.

4.2.1 CNN

Mnih et al. [2] used a CNN as the underlying neural network for their DQN, which they achieved good results with. We used the same type of architecture as this, but we modified the sizes of our layers to fit our game world. We used a CNN because it allowed us to use a deeper neural network than a fully connected neural network would, because of fewer trainable parameters.

The first layer that we used consists of 16 filters of size 3×3 with a stride of 1. The second and third layers both consist of 32 filters of size 3×3 with stride 1. The fourth layer is a fully connected layer with 512 neurons. All of these layers use the ReLU activation function, which computes $\max(0, x)$ (it only keeps the positive part of the input). Finally, the last layer has one output neuron for each valid action (four in our case, since the available actions are up/down/right/left) without an activation function. The action represented by the node with the highest output value is then chosen.

We conducted experiments to find a good network structure, and the structure that we chose provided a good trade-off between performance and training speed. With less limited computing and time resources, one might be able to find a better network structure that could lead to a better overall performance.

4.2.2 Reward functions

We conducted some initial experiments to find a good reward function to use. For each experiment, we gave a reward of 10 for eating a fruit and a reward of -10 for dying. We tried giving the inverse of the change in Manhattan distance (a natural way to measure distance in a grid) between the snake's head and the

fruit as a reward at every timestep, we tried a negative reward of -0.01 at every time step and also no reward at every time step. We noticed that the approach using Manhattan distance led to the fastest learning of the game's goal, so we chose to use this approach for the final experiments.

For reinforcement learning methods, the topic of sparsity versus density of rewards is important. If the rewards are too sparse (meaning that the agent is rarely given a reward), training can be very slow since it takes time for the agent to "figure out" the goal of the game. More dense rewards can thus lead to faster learning. Manhattan distance and negative rewards as mentioned above gives more dense rewards than only giving rewards upon eating fruits. Our choice of using the Manhattan distance is grounded in the fact that we have a limited time and limited computing resources, which means that fast training is of the essence in our case.

However, it is possible that another reward function (e.g. negative reward or rewards only given when eating fruits) might lead to a higher total reward if given much more training.

4.2.3 Stable Baselines

We used an implementation of PPO from a collection of reinforcement learning algorithms called Stable Baselines [22]. This implementation is an improved version of OpenAI's implementation, called OpenAI Baselines [23]. We chose this implementation since it was well-documented and appropriate to extend with our own CNN. Also, this implementation is built on top of the machine learning library TensorFlow [24]. This library provides methods of saving and loading models, which was necessary for us.

4.2.4 OpenAI Gym

The experiments were conducted using the OpenAI Gym toolkit, which is a way to create environments suitable for autonomous agents [25]. We chose this toolkit since it was a convenient way to structure our experiments, and also because it is the default way to create environments for Stable Baselines [22].

4.3 Experiments

All training was conducted on randomly generated maps (for the empty map, this means that the placement of the snake and fruits are random between games).

4.3.1 Main experiments

We trained agents on three different settings. The first setting was called *Empty*, where an agent was trained for 10 million timesteps on the empty map, i.e. maps with complexity 0.00.

The second setting was called *Increase*, where an agent was trained for 1 million timesteps on maps of increasing complexity. In this setting, the first 1 million timesteps of training were performed on maps with the complexity 0.00. The second 1 million timesteps were performed on maps with the complexity 0.01, and so on up to and including complexity 0.09, resulting in 10 million total timesteps.

The third setting was called *Complex*, where an agent was trained for 10 million timesteps on maps with complexity 0.09.

The reason for conducting these experiments was to compare how the different agents behave in different environments.

4.3.2 Transfer learning experiments

We also conducted experiments in transfer learning, where we transferred models from the main experiments to other environments and resumed training.

We took the fully trained agent from the *Empty* setting and transferred it to the *Complex* setting and trained for another 1 million timesteps.

Furthermore, we also took both of the fully trained *Increase* and the *Complex* models and transferred them to the *Empty* setting and trained for another 1 million timesteps.

The reason for conducting these experiments was to see if there are any benefits in starting training with a pre-trained model (which has been trained on a different environment) compared to using a totally untrained model.

4.3.3 Summary

A summary of the models trained are shown in Table 4.1.

Model name	**Timesteps**	**Description**
Empty	10,000,000	Trained on maps with complexity 0.00.
Increase	10,000,000	Trained on maps with complexities in the range $[0.00, 0.09]$, 1,000,000 timesteps each.
Complex	10,000,000	Trained on maps with complexity 0.09.
Empty $\rightarrow$ *Complex*	1,000,000	Trained on maps with complexity 0.09, used the model *Empty* as initial model.
Increase $\rightarrow$ *Empty*	1,000,000	Trained on maps with complexity 0.00, used the model *Increase* as initial model.
Complex $\rightarrow$ *Empty*	1,000,000	Trained on maps with complexity 0.00, used the model *Complex* as initial model.

Table 4.1: The models with descriptions of how they were trained.

Chapter 5

Results

In this chapter, the results are presented, with the aim of giving an understanding of the outcome of our experiments.

5.1 Performance

In this section, the performances of the agents in terms of eaten fruits are presented.

5.1.1 Complexity 0.00

As seen in Figure 5.1, the model which performs best on maps with complexity 0.00, i.e. the empty map, is the model which trained only on the empty map.

The model *Increase* $\rightarrow$ *Empty* almost has the same level of performance as the model *Empty*. Also, the model *Complex* $\rightarrow$ *Empty* has a reasonably high performance, however it is not as high as the two aforementioned models.

Furthermore, the model *Increase* outperforms the model *Complex* on this complexity.

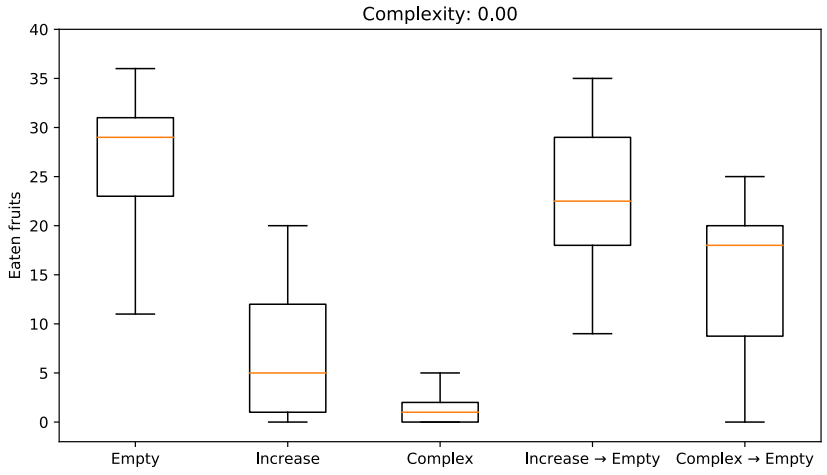

Figure 5.1: Comparison between the models on 100 maps with complexity 0.00, playing 10 games on each map.

5.1.2 Complexity 0.03

On this complexity level, the model *Increase* outperforms both *Empty* and *Complex*. The model *Complex* has a higher performance than *Empty*. This is illustrated in Figure 5.2.

5.1.3 Complexity 0.06

On this complexity level, the models *Increase* and *Complex* are on about the same level of performance as on complexity 0.03. *Empty*, however, has a lower performance on this complexity level. This is illustrated in Figure 5.3.

5.1.4 Complexity 0.09

On this complexity level, the model *Increase* has the best performance. The model *Complex* is almost on the same level of performance as *Increase*.

The model *Empty* $\rightarrow$ *Complex* is better than *Empty*, but not on the same level as *Increase* or *Complex*. The model *Empty* is not able to eat any fruits on this

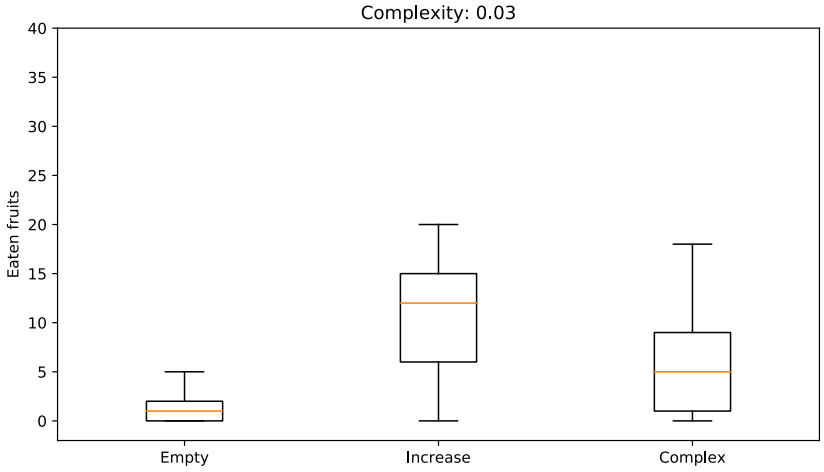

Figure 5.2: Comparison between the models on 100 maps with complexity 0.03, playing 10 games on each map.

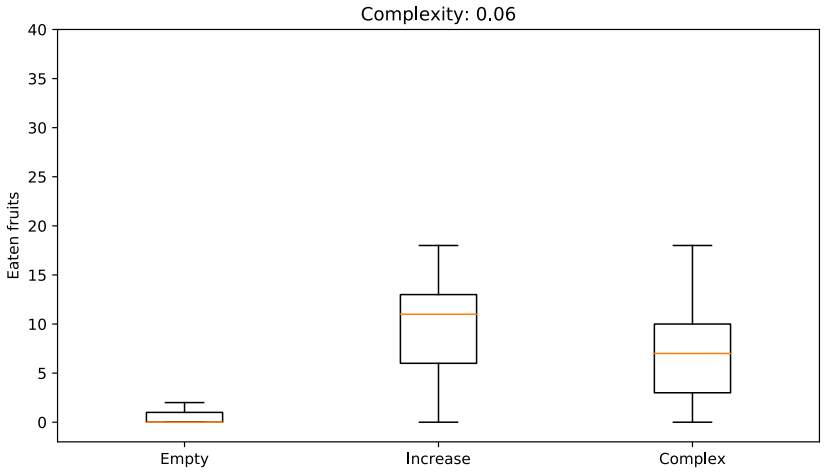

Figure 5.3: Comparison between the models on 100 maps with complexity 0.06, playing 10 games on each map.

complexity level. This is illustrated in Figure 5.4.

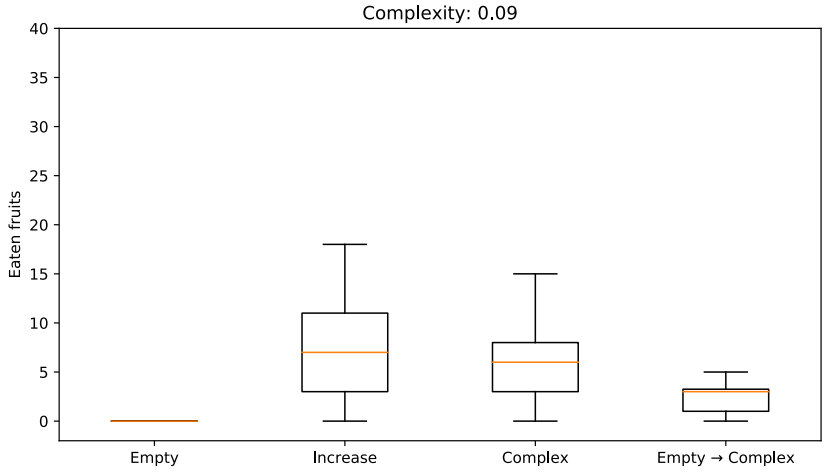

Figure 5.4: Comparison between the models on 100 maps with complexity 0.09, playing 10 games on each map.

5.2 Transfer learning

In this section, the results from the transfer learning experiments are presented.

5.2.1 Transferring to Complex

In Figure 5.5 the performance for the first 1,000,000 timesteps for the model *Empty* is shown. We can see that in the beginning, the agent is not able to eat any fruits, and during training this steadily increases.

The performance throughout training for *Increase* $\rightarrow$ *Empty* is displayed in Figure 5.6. Here, we can see that the agent is able to get fruits from the beginning, and after training the performance has gone up.

A similar trend as the one for *Increase* can be identified in Figure 5.7, but both the initial and final performance is lower in this case.

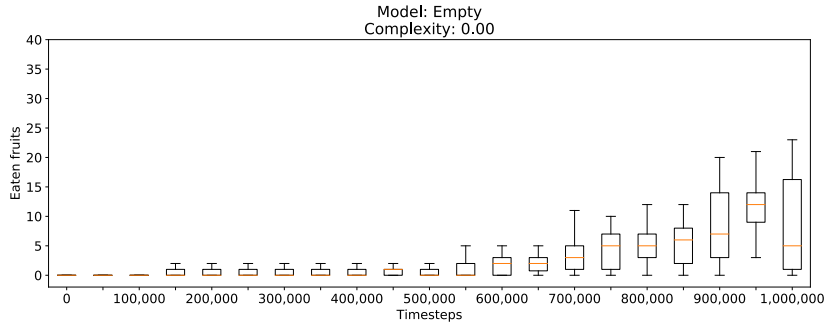

Figure 5.5: The performance throughout training for the model *Empty*, measured on 10 different maps playing 10 games on each map.

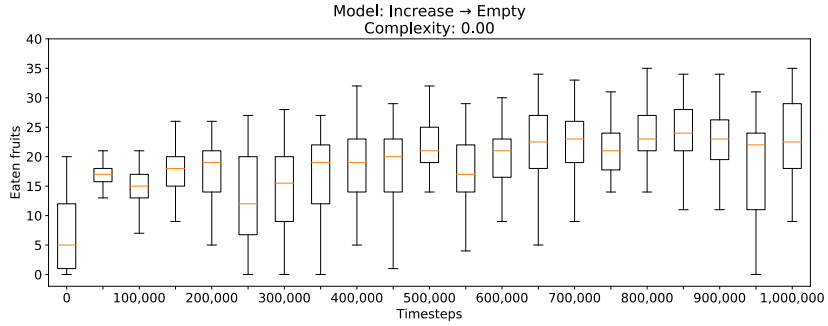

Figure 5.6: The performance throughout training for the model *Increase* → *Empty*, measured on 10 different maps playing 10 games on each map.

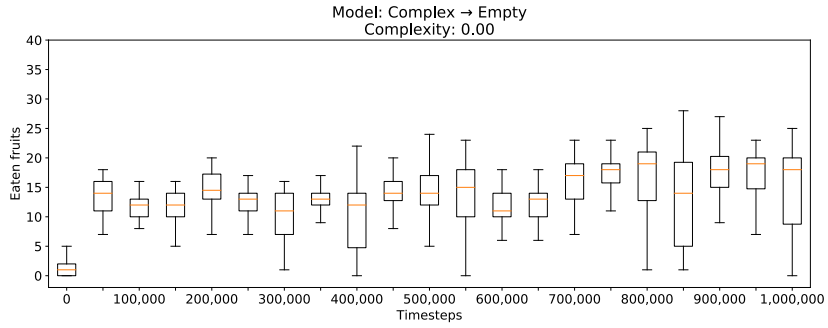

Figure 5.7: The performance throughout training for the model *Complex* → *Empty*, measured on 10 different maps playing 10 games on each map.

It is worth noting that for both the *Increase* and *Complex* models, the most gain in performance comes after the first 50,000 timesteps of transfer learning.

To sum up, transferring the agents trained in more complex environments to a less complex environment gives an increase in both final achieved performances, and also how quickly a high performance can be achieved. Furthermore, the model *Increase* outperforms the model *Complex*.

5.2.2 Transferring to Empty

In Figure 5.8, the performance throughout training for the model *Complex* is shown. We can see that the agent is not able to get any fruits during these first 1,000,000 timesteps.

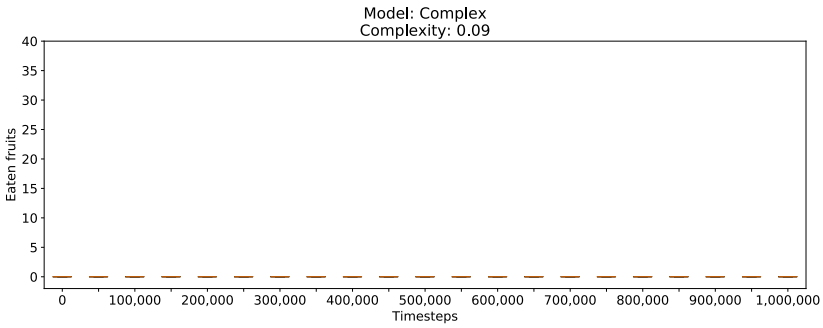

Figure 5.8: The performance throughout training for the model *Complex*, measured on 10 different maps playing 10 games on each map.

However, the model depicted in Figure 5.9, which has previously been trained in a simpler environment, is able to get fruits after 50,000 timesteps and onward.

To sum up, using a pre-trained agent, which has been trained on a simpler environment, gives an increased performance compared to an untrained agent.

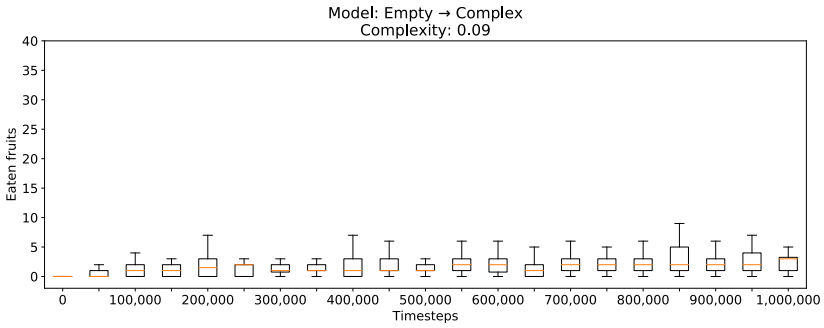

Figure 5.9: The performance throughout training for the model *Empty* $\rightarrow$ *Complex*, measured on 10 different maps playing 10 games on each map.

Chapter 6

Discussion

In this chapter, the results are discussed, along with a discussion on the problems with our experiments and also suggestions for future research.

6.1 Increasing complexity

From our results, we can conclude that there are several benefits of using environments with increasing complexity, compared to a fixed high complexity. Using an increasing complexity results in a more versatile agent which performs better across all maps of different complexities, which can be seen in Figure 5.1, Figure 5.2 and Figure 5.3. It also performs slightly better on the most complex environment, as seen in Figure 5.4.

It is worth noting that the agent trained in the *Complex* setting is unable to eat any fruits in the beginning of its training process, as seen in Figure 5.8. It is possible that environments with high complexity results in the agent having trouble to figure out how to acquire rewards, since it is so easy to collide with an obstacle early on. We manually inspected the agent trained in the *Complex* environment during training, and we saw that it had a hard time using all of the different actions, i.e. avoiding certain movement actions in an attempt to stay alive. This was a sign that learning basic movement was problematic, and until basic movement had been figured out the agent was unable to collect fruits.

This could be related to human learning as well. If the task is too complex, it is hard to learn anything at all. In the game of Snake, a single mistake leads to failure, which also might be a contributing factor to why it is harder to learn

in a more complex environment. Starting with a simple problem and slowly increasing the complexity would in many cases be a much more natural way to learn a task for a human.

It might be natural for teams working with iterative development to use increasing complexity environments as well, given that the problem can be simplified. In these scenarios, it makes sense to train an agent on this simplified version in the beginning and increasing the complexity as development progresses. This gives continuous feedback on the development process, since there will always be an agent trained on the current state of the environment which can be evaluated.

The Snake environment is a pretty generic environment and is closely related to the fundamentals of reinforcement learning, i.e. positive and negative rewards, a limited observation space giving perfect information, an action space of four different actions. Similar results, i.e. that there are benefits of using increasing complexity environments, were also discovered by Örnberg and Nylund [18] and Wang et al. [19]. We therefore conclude that an agent trained on increasingly complex environments displays benefits in both versatility and final performance.

However, if versatility is not needed and the environment is somewhat simple, it might be a better idea to only train on the simple environment. This is what we saw when evaluating the model trained in the *Empty* setting on the 0.00 complexity map, where it outperformed all other models, as seen in Figure 5.1.

6.2 Transfer learning

In all of our trials, we saw that it is better to use a pre-trained model which has been trained in a different complexity environment, than using a randomly initialized model. This was noted when transferring to a simpler environment, as seen when comparing Figure 5.5 to Figure 5.6 and Figure 5.7. It was also noted when transferring to a more complex environment, as seen when comparing Figure 5.8 to Figure 5.9. Therefore, the pre-trained models should be used in real-world situations where a pre-trained model exists.

Furthermore, we also saw that the largest benefit was given in the beginning of the transfer learning, as seen in Figure 5.6, Figure 5.7 and Figure 5.9. It might be a good idea to monitor the learning and only train the agent until it has an acceptable performance to save time and computing resources.

6.3 Problems

We only trained a single agent for each setup, giving us a single starting point. If more agents had been trained in parallel with different starting points, other results might have been achieved. It would also be interesting to see the variance between the agents.

Furthermore, we used 10 million timesteps for the original training and 1 million timesteps for the transfer learning. We conducted initial experiments on 1 million and 5 million timesteps, and we saw that the performance was still increasing, so 10 million was chosen because of constraints on both time and computing resources. It would be interesting to see the performance and behaviour of the agents given even more training.

Even though PPO is considered a somewhat stable method of performing deep reinforcement learning, we still experienced fairly unstable training with it, which can be seen in e.g. Figure 5.7. This instability could explain the variance in our transfer learning results. We also have a lot of random factors in our environments, such as random spawning locations for the snake and the fruits. Training on a single map would perhaps give a more stable training process and a higher overall performance but would perhaps come at the cost of lower versatility.

6.4 Future research

We do not know how any of these agents behave asymptotically. A suggestion for future research would be to see if *Increase* $\rightarrow$ *Empty* would eventually beat *Empty* on the 0.00 complexity map, and in that case how much training this would take (given that both of the models receive additional training). This type of phenomena was seen by Wang et al. [19], i.e. training on a more complex environment, and then going back to a simpler environment gave a better result than only training on the simpler environment.

Furthermore, it would be interesting to investigate how the strategy of increasingly complex environments works on different, and perhaps more complex, games. Also, this could be evaluated in a real-world scenario, e.g. using robots.

It would also be interesting to have denser timesteps when plotting the training process, i.e. performing inference more often. This might lead to smoother transitions and thus explain what happens between the jumps seen in Figure 5.6 and Figure 5.7.

Chapter 7

Conclusions

In this chapter, our conclusions are presented, which aims to answer our research questions.

We conclude that an agent, playing our implementation of Snake, trained in increasingly complex environments has benefits in both overall performance and versatility over an agent trained in a fixed complexity environment, as seen in Figure 5.1, Figure 5.2, Figure 5.3 and Figure 5.4. As evident from Wang et al. [19] and Örnberg and Nylund [18] this conclusion seems to hold generally, but more research could be done on both simulated and real-world scenarios to strengthen this assertion.

Furthermore, we conclude that transferring a pre-trained agent to an environment with different complexity and resuming training gives a benefit in performance over starting with an untrained agent, as seen when comparing Figure 5.5 to Figure 5.6 and Figure 5.7, along with the comparison of Figure 5.8 to Figure 5.9.

Bibliography

[1] The Robot Report. *10 Biggest Challenges in Robotics*. https://www.therobotreport.com/10-biggest-challenges-in-robotics/ (accessed 2019-06-02).

[2] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), p. 529.

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

[4] OpenAI. *Key Concepts in RL*. https://spinningup.openai.com/en/latest/spinningup/rl_intro.html (accessed 2019-03-18).

[5] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. 2nd. Cambridge, MA, USA: MIT Press, 2018.

[6] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press. 2015.

[7] Balázs Csanád Csáji. "Approximation with Artificial Neural Networks". In: (2001). Eötvös Loránd University.

[8] Stanford University. *Convolutional Neural Networks for Visual Recognition*. http://cs231n.github.io/convolutional-networks/ (accessed 2019-05-09). 2019.

[9] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[10] OpenAI. *Kinds of RL Algorithms*. https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html (accessed 2019-03-19).

[11] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[12] OpenAI. *Proximal Policy Optimization*. https://spinningup.openai.com/en/latest/algorithms/ppo.html (accessed 2019-03-25).

[13] L Torrey and J Shavlik. “Transfer Learning”. In: *Handbook of Research on Machine Learning Applications* (2009).

[14] Matthew E Taylor and Peter Stone. “Transfer learning for reinforcement learning domains: A survey”. In: *Journal of Machine Learning Research* 10.Jul (2009), pp. 1633–1685.

[15] Bowei Ma, Meng Tang, and Jun Zhang. *Exploration of Reinforcement Learning to SNAKE*. Stanford University. 2016.

[16] Vlad Ovidiu Chelcea and Björn Ståhl. *Deep Reinforcement Learning for Snake*. 2018.

[17] Zhepei Wei et al. “Autonomous Agents in Snake Game via Deep Reinforcement Learning”. In: (July 2018), pp. 20–25. DOI: 10.1109/AGENTS.2018.8460004.

[18] Oscar Örnberg and Jonas Nylund. *Incrementally Expanding Environment in Deep Reinforcement Learning*. 2018.

[19] Rui Wang et al. “Paired Open-Ended Trailblazer (POET): Endlessly Generating Increasingly Complex and Diverse Learning Environments and Their Solutions”. In: *CoRR* abs/1901.01753 (2019). arXiv: 1901.01753. URL: http://arxiv.org/abs/1901.01753.

[20] Marlos C. Machado et al. “Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents”. In: *CoRR* abs/1709.06009 (2017).

[21] Maxim Egorov. *Multi-Agent Deep Reinforcement Learning*. Stanford University. 2016.

[22] Ashley Hill et al. *Stable Baselines*. https://github.com/hill-a/stable-baselines (accessed 2019-04-09). 2018.

[23] Prafulla Dhariwal et al. *OpenAI Baselines*. https://github.com/openai/baselines (accessed 2019-04-09). 2017.

[24] TensorFlow. *TensorFlow*. https://www.tensorflow.org/ (accessed 2019-04-18). 2019.

[25] Greg Brockman et al. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540.

TRITA-EECS-EX-2019:384