Curiosity-driven algorithm for reinforcement learning

Master’s thesis in Web Intelligence and Service Engineering

May 16, 2019

University of Jyväskylä
Faculty of Information Technology
Author: Vitalii Tsybulko

Contact information: tseybulko.vitalii@gmail.com

Supervisors: Prof. Vagan Terziyan

Title: Curiosity-driven algorithm for reinforcement learning

Project: Master’s thesis

Study line: Web Intelligence and Service Engineering

Page count: 58+5

Abstract: One problem of current Reinforcement Learning algorithms is finding a balance between exploitation of existing knowledge and exploration for a new experience. Curiosity exploration bonus has been proposed to address this problem, but current implementations are vulnerable to stochastic noise inside the environment. The new approach presented in this thesis utilises exploration bonus based on the predicted novelty of the next state. That protects exploration from noise issues during training. This work also introduces a new way of combining extrinsic and intrinsic rewards. Both improvements help to overcome a number of problems that Reinforcement Learning had until now.

Keywords: Reinforcement Learning, Proximal Policy Optimisation, Curiosity-driven exploration bonus
Glossary

A3C  Asynchronous Advantage Actor Critic Algorithm. Improved Actor Critic model, that uses advantage function instead of the value function and is also able to support multiple independent agents. The Actor Critic model is a reinforcement learning algorithm, which uses both main methods: value-based, like Q-learning, and policy-based, like policy gradients (Volodymyr Mnih et al., 2016).

CNN  Convolutional neural network, class of deep neural networks. Being a fully-connected network, CNN assembles sophisticated pattern from smaller and simpler patterns, which allows to control complexity. Hidden layers typically consist of several convolutional layers, pooling layers, normalisation and activation function. Commonly used in visual information classification and analysis.

CUDA Platform and programming interface between developed software and CUDA-enabled GPU. Allows developers to have direct access to the virtual instruction set of GPU and its parallel computational elements.

DQN  Deep Q-network algorithm. In classic Q-learning agent builds a table of Q-values, trying to learn action-value function Q. But with big tasks it is impossible to keep in memory table for combination of all states and actions. Using deep network agent generalise the approximation of Q-value function instead of remembering all solutions (Volodymyr Mnih et al., 2013).
GRU  Gated recurrent unit, gating mechanism for recurrent neural networks. Works like long short-term memory with a flush button. Similar to LSTM, but has fewer parameters and doesn't have an output gate. Aimed to solve vanishing gradient problem, common issue of recurrent neural networks (Kyunghyun Cho et al., 2014).

Hierarchical

Reinforcement Learning  Group of reinforcement learning algorithms which uses temporal abstraction to decompose complicated tasks into smaller and simpler routines. They provide hierarchical control architecture and corresponding learning algorithms. Relies on the theory of Markov decision process (Andrew G Barto and Sridhar Mahadevan, 2003, Thomas G. Dietterich, 1999).

KERAS  Open-source python neural network library. It offers high-level tools to create and experiment with deep neural networks on top of TensorFlow, Theano or Microsoft Cognitive Toolkit. It contains implementations of many building blocks for neural networks like activation functions, layers, optimisers and others. It also supports convolutional and recurrent neural networks.

MCTS  Monte Carlo tree search, heuristic search algorithm which is often applied in gameplay, also for games with incomplete information. The algorithm uses random sampling to choose the next move and plays many playouts to the very end. Backpropagation of playout result updates weights in the tree of decisions, so better moves are more likely to be performed in the next playouts (Cameron B. Browne et al., 2012).
| **MuJoCo** | Multi-Joint dynamics with Contact. Physics engine, simulator used by many developers and researches as a sandbox environment for testing robots, algorithms, biomechanics or graphics and animation developments. |
| **PPO** | Proximal Policy Optimisation, family of policy gradient methods for reinforcement learning. Relatively simple and efficient way, comparing to other methods, to choose suitable step size for policy updates. Basic idea is to keep a balance between sampling data through interaction with the environment and optimising an artificially created objective function (John Schulman et al., 2017). |
| **Q-learning** | Model-free reinforcement learning algorithm. Uses table of state-action combination filled with Q values. After that Bellman Equation can be used as an update rule for this table. For any finite Markov decision process Q-learning finds an optimal policy that maximises expected value of the total reward over all steps in the chain from the current state. |
| **RNN** | Recurrent neural networks, class of neural networks, which use mechanisms of internal memory to develop temporal dynamic behaviour. In such a way they can treat sequential impulses, which makes them useful for speech analysis and connected handwriting recognition, for example (Danilo P Mandic and Jonathon Chambers, 2001). |
| **TRPO** | Trust Region Policy Optimisation, another algorithm for policy optimisation. While it is more complicated than PPO and requires more tuning of hyperparameters, it is able to provide monotonic improvements to the policy function. Key benefits are applicability to model-free algorithms with large and non- |
linear policies, as well as scalability potential (John Schulman et al., 2015, John Schulman et al., 2017).

**UCT**

Upper Confidence Bounds for Trees method, an algorithm that uses the Monte-Carlo tree search, but extends it. Every layout is played using an existing tree, but after the known path is over it is played randomly. It gives a good balance between searching the best policy and also exploring alternative solutions (Levente Kocsis and Csaba Szepesv'ari, 2006).
List of Figures

Figure 2.1 Agent architecture ........................................................................................................22
Figure 2.2 Predictor and target networks ......................................................................................23
Figure 2.3 Policy network .................................................................................................................25
Figure 3.1 CNN and RNN policy networks performance comparison (Venture) ..............31
Figure 3.2 Episodic and non-episodic intrinsic reward performance in Venture ................32
Figure 3.3 suggested solution, next-state predictor and agent without curiosity bonus performance in Solaris ........................................................................................................33
Figure 3.4 suggested solution, next-state predictor and agent without curiosity bonus performance in Venture ........................................................................................................34
Figure 3.5 Mean internal and external return for the shorter experiment in Solaris ..........35
Figure 3.6 Episodic and total reward for the shorter experiment in Solaris .......................36
Figure 3.7 Mean internal and external return for the shorter experiment in Venture ..........36
Figure 3.8 Episodic and total reward for the shorter experiment in Venture .....................37
Figure 3.9 Mean internal and external return for the longer experiment in Venture ..........38
Figure 3.10 Episodic and total reward for the longer experiment in Venture ...................38
Figure 4.1 Median human-normalised performance across 57 Atari games. Comparison of rainbow DQN with other algorithms ........................................................................................................44
Figure 4.2 Goal function for lego brick stacking agent ................................................................49
Figure 4.3 Results of TRPO algorithm in HalfCheetah ................................................................50
Figure A.1 Venture. Agent visits sub level for the first time, intrinsic reward spike ...........59
Figure A.2 Venture. Agent kills both enemies, extrinsic reward spike ............................59
Figure A.3 Solaris. Agent kills enemy, extrinsic reward spike. One of the latest episodes, intrinsic reward is low .................................................................60
Figure A.4 Solaris. Agent leaves planet for the first time, intrinsic reward spike. Intrinsic reward level is high due to the novelty of everything in the beginning ......60
Figure A.5 Venture. Agent blocks itself in the corner, can’t move — the gap at intrinsic reward chart .................................................................................................................61
Figure A.6 Venture. Agent leaves room for a moment and sees familiar hall — drop at intrinsic reward chart ..............................................................................................................61
Figure A.7 Venture. Agent leaves room and sees familiar hall — intrinsic reward drops — but then moves around the hall to an unusual location, gaining new experience ........................................................................................................62
Figure A.8 Venture. Agent travels around the hall and enters the unfamiliar room, even kills one enemy — intrinsic reward spike ..................................................................................62
Figure A.9 Venture. Agent travels around the hall and enters room for the first time, intrinsic reward spike .................................................................................................................63
List of Tables

Table 3.1 Hardware components.................................................................29
Table 4.1 Performance (game scores) of the four real-time game playing agents,
compared to the off-line UCT game playing agent.........................................45
Contents

INTRODUCTION ..................................................................................................................... 10

CHAPTER 1: BRIEF THEORY .............................................................................................. 13
  1.1 Reinforcement learning ......................................................................................... 13
  1.2 Neural networks ..................................................................................................... 16

CHAPTER 2: IDEA ............................................................................................................... 20
  2.1 Noisy TV problem .................................................................................................. 21
  2.2 Suggested solution ................................................................................................. 22

CHAPTER 3: IMPLEMENTATION AND TESTS ................................................................. 26
  3.1 Environment ........................................................................................................... 26
  3.2 TensorFlow ............................................................................................................ 28
  3.3 Hardware ............................................................................................................... 29
  3.4 Experiments ........................................................................................................... 30
    3.4.1 Episodes filtering ............................................................................................ 30
    3.4.2 Network architectures .................................................................................... 31
    3.4.3 Non-episodic rewards ..................................................................................... 32
    3.4.4 Comparison to existing solutions ................................................................... 33
  3.5 Results evaluation ................................................................................................. 35

CHAPTER 4: PROBLEMS OF REINFORCEMENT LEARNING ........................................ 39
  4.1 Reward assignment ............................................................................................... 39
  4.2 Exploration and exploitation ................................................................................ 40
  4.3 Effectiveness ......................................................................................................... 43
  4.4 Alternatives to Reinforcement Learning ............................................................. 45
  4.5 Overfit .................................................................................................................. 46
  4.6 Reward hacks ...................................................................................................... 47
  4.7 Stability ............................................................................................................... 50

CONCLUSION .................................................................................................................... 52

BIBLIOGRAPHY .................................................................................................................. 54

APPENDICES ..................................................................................................................... 59
  A. Selected milestones during agent training ............................................................ 59
Introduction

Reinforcement learning (RL) allows agent to do random actions in environment and to learn from discovered changes of it. Basically, this sounds like a primitive way of making ship in a bottle: let’s put inside some matches, glue and trash, shake it and see if it turns out to be a three masts Spanish galleon. This method works fine when by doing stochastic actions agent always receives a reward. In this case, however, it is way harder to force agent to follow some direct goal, or to solve particular problem acting by specific scenario. It also requires a lot of work from experiment creator, since each reward providing rules must be defined. When the agent switches to another environment, all the work must be done again.

Classical reinforcement learning fails to explore environments with sparse rewards. The most successful implementations overcome challenging tasks by processing multiple experiences achieved from many executions that run in parallel environments (Silver et al., 2016, Espeholt et al., 2018, Zoph and Le, 2016, Horgan et al., 2018). Active use of exploration methods based on information gain, prediction gain or pseudo-counts makes them barely scalable. Apart from scalability, those solutions require huge computation power as well. And progress of reinforcement learning in this direction was slower, comparing to other machine learning approaches, also due to hardware capabilities of nowadays machines.

This thesis suggests to solve the existing difficulties by an approach, which utilises natural and clear concept of one’s desire to discover and investigate. In this case, agent is not just trying to maximise a reward, but trying to find new ways to achieve it. The main difference from existing solutions is that agent is not using prediction to choose the most valuable next step, but looks for the most unknown next step.

Similar algorithms that use prediction error to lead agent’s exploration are prone to stochastic changes in the environment, so-called Noisy TV problem (Achiam and Sastry,
One of the goals of this thesis is to make an algorithm that is not attracted by those sporadic changes.

All these, first of all, force researcher to figure out the hidden obstacles relevant to the field. To overcome them, this thesis first has to lighten the topic by answering the following research question: **what are the main weaknesses of current reinforcement learning algorithms and what are the possible solutions to them?**

As was mentioned, curiosity might be a solution to some of those weaknesses. That’s why the second research question reads: **how curiosity can help to overcome known problems in reinforcement learning?**

Answers to those questions logically lead to the main research question of this thesis: **how to implement an exploration bonus which encourages agent to discover more, while protecting agent from undesired environment stochasticity?**

This thesis wouldn’t be complete without practical experiments with created algorithm. Evaluation of achieved results is done in Atari games environment sandbox, which became a standard benchmark for reinforcement learning due of many reasons (Mnih et al., 2013, Bellemare et al., 2016). Several games such as Gravitar, Pitfall!, Solaris, Venture, Montezuma's revenge, Private Eye have sparse rewards and are difficult for exploration. This makes them perfect candidates for experimental environments, especially because classical reinforcement learning algorithms fail to operate there due to the mentioned above challenges.

While the main goal is to achieve decent results with exploration bonus only, to increase the agent’s performance developed algorithm combines external reward applying policy optimiser (John Schulman et al., 2017).

All these together motivate to organise the rest of this thesis as follows. Chapter 1 contains a theoretical background which helps to understand the terms and concepts described in the rest of the thesis. Chapter 2 lists in more detail the challenges to be solved and reveals
specifics of the developed algorithm. Chapter 3 starts from describing the used tools, environments, frameworks and testing hardware, continuing with justification of performed tests. Rest of Chapter 3 provides results of performed experiments. Chapter 4, analysing other papers about RL, gives an overview of existing problems in reinforcement learning, also sharing some thoughts about possible ways of resolving them. Chapters 2, 3 and 4 are the core of this work and represent three layers of solving the problem with methods of reinforcement learning: good idea, bad results, ugly problems behind them. Finally, the last chapter provides conclusions of this work.
Chapter 1: Brief theory

This chapter introduces main terms and concepts to make it easier to understand consecutive description of the algorithm. First, we start with general theory of Reinforcement Learning. After that, some background of neural networks and their hyperparameters will be provided in the context of deep learning.

1.1 Reinforcement learning

Reinforcement learning is a family of goal-oriented algorithms, directed to maximise some score or to finish some task. They start from scratch being absolutely unaware about how environment works, like a newborn. Doing random actions, following reinforcement learning actors — agents — receive feedback from environment and they are punished or rewarded according to this feedback. Trying to avoid punishments and getting more rewards, algorithm evolves over iterations. Reinforcement learning also tries to solve problem when there are big delays between action and its result, when it is hard to recognise which outcome was caused by particular action. Agents can act in environments which are changeable by performed actions — then it must learn to imagine future gains of performed actions, and credit assignment problem takes place. If the agent is acting inside non-changeable environment, it needs to learn only immediate expected reward to its actions, and it is solved through Multi-arm Bandits (Sutton and Barto, 2018).

Reinforcement learning differs from familiar Machine Learning methods. It doesn’t depend on preprocessed data, but develops knowledge from gained experience (Kober et al., 2013). The main goal is to achieve certain performance, which means that it is significantly more important to keep balance between exploration and exploitation. Last, it is much more based on real-world communication with environment, than other fields of machine learning (like, for example, genetic algorithms).
Reinforcement learning is defined in terms of optimal control of Markov Decision Process (MDP). MDP is defined by:

1. Set of states $s \in S$, where $S$ denotes the state space

2. Set of actions $a \in A$, where $A$ denotes the action space

3. Transition probability

$$p(s' | s, a) = P_r(s_{t+1} = s' | s_t = s, a_t = a)$$

4. Scalar reward $r$

5. Reward function $R : S \times S \times A \rightarrow \mathbb{R}$

$$R(s, a, s') = \mathbb{E}(r_{t+1} | s_t = s, a_t = a, s_{t+1} = s')$$

6. Discount factor, $\gamma$

Agent is an actor, taking some actions and making decisions. Algorithm is an agent. In Super Mario game, for example, agent is Mario or Luigi. Mario can interact with the environment, move, jump etc. Collecting coins Mario get’s positive reward, being hit by enemy he receives negative reward.

Environment is real world, or, more frequently, simulated and simplified part of it, in which agent operates. It takes agent’s action and agent’s state as an input and returns back new state and reward or punishment. It operates according to some rules, like physics or social rules in real life or game rules in game.

Policy is a mapping between states and actions. Leaving in environment, agent needs to develop policy to achieve goal and get intermediate rewards. Policy can be deterministic and depend only on the state, or stochastic, defining a probability distribution over the actions from given state.
Aim of agent is to maximise the sum of rewards, called return, $G_t$:

$$G_t = r_t + r_{t+1} + r_{t+2} + \ldots + r_{N-1},$$

where $t$ is time index and $N$ means the end of an episode.

Reward can be immediate or delayed. An example of such thing might be score changes in a game. Reward may have a negative impact too: death in a game might be considered as an anti-reward, punishment. Talking about long-term rewards and immediate rewards, of course, there is a set of problems dedicated to matching long-playing rewards and their causations. One should distinguish delayed and future rewards, where the last one is the same as instant reward, but inaccessible in current state and available after few known actions. Delayed reward might require few action in combination, or just behave stochastically, or happens much later after action was performed. To make future rewards less desirable than the closest ones, discount factor may be applied:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k},$$

where $\gamma \in [0, 1)$ is the discount factor.

If discount factor $\gamma$ is 1, all known rewards worth the same for the agent. In this case coin at the right end of the level considered by agent as valuable as coin located a few steps to the left from it. But if $\gamma$ is 0.5, every extra step required to achieve some reward reduces reward’s value.

In order to choose the most desired and beneficial transition between states, agent must measure it with value function (Kober et al., 2013). Value function is defined as expected sum of rewards, achieved by following certain policy from given state:

$$V_\pi(s) = \mathbb{E}_\pi(G_t | s_t = s) = \mathbb{E}_\pi(\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s)$$
One of the most important terms in reinforcement learning, Q-function or action value function is defined as expected sum of rewards while taking some action in some state and following some policy $\pi$:

$$Q_\pi(s, a) = \mathbb{E}_\pi(G_t \mid s_t = s, a_t = a) = \mathbb{E}_\pi(\sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t = s, a_t = a)$$

Theoretically, Q-learning, which tends to define value function, requires knowledge about all states, together with reinforced values of all actions from those states, which is problematic in high-dimensional environments. It is not possible to store this information in practice.

Policy search methods, or policy gradient methods calculate difference in policy parameters for current and next policies. It is easier to compute, but such methods have another downside: they can get stuck in local optima and never reach global optimal.

Combination of both methods was suggested as the Actor Critic model (Barto et al., 1983). Actor, control policy, chooses the action. Critic, value function, transmits values to the Actor and defines policy updates by that. A2C algorithm is an improvement of the Actor Critic model which uses advantage function instead of value function. And A3C, the next improvement, can update policy asynchronously, which gives an opportunity to run several agents in parallel environments.

1.2 Neural networks

Deep learning relies on deep neural networks. They allow to find compressed representation of such multi-dimensional data structures like images or sounds. Deep learning is widely applied in supervised learning, reinforcement learning and other fields of machine learning (Lin, 1992).

Each layer of neural network uses outputs of previous layer as an input. Input layer interprets environment signals, generalising sensors data while passing it through neurones.
Hidden layers of deep neural network perform feature extraction and transformation. Output layer prepares data in the desired format, for example, matching it with possible actions.

When computations flow forward from input to output, in the output layer and in each hidden layer we can compute error derivatives. Then we can propagate gradients back towards the input layer so that weights can be updated to optimise some loss function (Li, 2017). This is the core of the learning part, generic goal is to find neuron's weights and biases that minimise the error.

Convolutional neural networks (CNN) treat input as images, explicitly (Stanford CS class, 2018). Hidden layers are made that way so they are more sensitive to some particular features. Deeper layers react to more and more specific features. Architecture of CNN contains, mostly, of three types of layers: pooling layer, fully connected layer and convolutional layer.

Convolutional layer uses small filter to scan the whole image, performing convolution of filter’s inputs.

Pooling layer performs subsampling, reducing input dimension. It allows us to control overfitting (Stanford CS class, 2018) and reduce computation and number of parameters. There are different pooling methods, like max pooling or average pooling.

Fully connected layer acts as a regular deep neural network layer. All neurons are fully connected to all activations of previous layer, like in regular non-convolutional neural networks.

Recurrent neural networks (RNN) deal with sequential data. When regular neural networks treat two following inputs independently, RNN keeps the context of the previous data. It also affects how learning is done. We can’t propagate just error, but we need to propagate derivative through time — to the beginning or to a certain point in the past. All derivatives will multiply with the same weight. Since it can cause gradient exploding for the weight
value, gradient clipping can be applied (Pascanu et al., 2012). To solve the opposite problem, gradient vanishing, method called Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997) is used. LSTM is very similar to RNN, but has a different function to compute the hidden state. In general, choosing another activation function often helps to solve gradient problems.

In order to build a deep neural network which is capable to solve posed problem, first of all, researcher must think about hyperparameters, such as:

• Selection of activation functions
• Initial weights
• Amount of layers
• Amount of neurons in hidden layers
• Selection of loss functions
• Splits and the optional use of k-folds cross-validation
• Regularisation of functions
• Learning rate
• Kinds of numeric inputs and their domains

and others.

Apart from that, some hyperparameters are specific for CNN:

• Amounts of convolutional / fully connected / pooling layers
• Padding types
• Pooling types
• Filter size
• Types of connections between layers

And also for RNN:

• Initialisation of hidden layer

• Design of LSTM

• Structure in regards to problem which must be solved
Chapter 2: Idea

Reinforcement learning sets a few challenges in front of researcher or developer. It is tricky to create such a reward policy which provides frequent, meaningful rewards, so that agent can easily find reinforcement doing random actions. It might be also complicated to define those policies for every environment where the algorithm is to be evaluated. Vast majority of algorithms run in parallel environments to reduce learning time, so good approach also requires scalability.

The method explained in this chapter allows to overcome those difficulties, solving them in relatively simple way. When it is challenging to achieve extrinsic reward, algorithm replaces it with intrinsic feedback. It is a universal approach so it can be scaled to a new environment with fewer adjustments. It is quite simple to implement too. This can be achieved by using the natural idea of curiosity when developing agent behaviour.

Every algorithm has to keep balance between exploration and exploitation. Curiosity is a craving for discovery and exploration which is inherent to any sophisticated organism. It can be applied also to the reinforcement learning by forcing model to predict the next step and push agent to the most unknown states.

As was mentioned, one of the biggest challenges in RL is operating inside environments that have sparse rewards. Sometimes it might require hundreds of steps for agent to finally get an achievement for the right move towards its task. Two common approaches for developing policies that bring to the agent a lot of external return are:

- Simple. Keep track of all visited states and give a reward for those which are the newest. With wider variety of actions and steps, complexity of this approach grows in the non-linear manner.

- A bit more sophisticated. Model predicts consequences of all possible actions and then chooses the most attractive action in terms of gained reward. Difference between prediction and actual feedback from environment is propagated to the predictor, so it learns to
operate better in familiar states and reduces prediction error. The more agent learns, the less becomes prediction error, and only states with the highest prediction error receive intrinsic reward, which stimulates agent to explore more.

Both these approaches beat algorithms with extrinsic reward only, like DQN and A3C, but can’t beat human, if we are talking about such environments where it is possible to compare the performance of real player and agent.

There are several possible reasons why such algorithm can’t predict consequences of the next action well enough:

1. State is just new. This is a good error because it is part of discovery. Learning on such mistakes, agent recognises new states better and better, reducing this kind of errors.

2. Model has poor architecture. It just can’t find good generalisation function to predict the next state accurately enough. In such case, the architecture must be reconsidered.

3. Environment response is non-deterministic. When environment produces random noise, any new state is unpredictable, and every action receives high intrinsic reward unrelated to the agent choice. So-called Noisy TV problem causes reward hacking, which is an especially unpleasant weakness of any RL algorithm.

2.1 Noisy TV problem

When we compare agent’s irresistible urge to discovery with man’s curiosity, reward for agent is the same as dopamine for human. And, likely to humans, agents like sticking in front of a TV thoughtlessly switching channels. Such a behaviour quickly becomes dangerously addictive, and agent is no more interested in the end goal.

There is a solution to this problem, suggested in (Machado et al., 2017). Main concept is that agent has some kind of sticky actions, which with certain probability repeat again even if the policy suggest to act differently. Authors claim that it is the best existing approach, but it also fails in cases when such sticky action happens between natural transitions (for
example, when opening players inventory which occupies the whole screen) and agent’s input changes dramatically.

Imagine a robot moving around the room. When it finds a door, it can open it, and sees totally new picture comparing to the picture of closed door. Repeating interaction with door, input will be always different, which makes sticky action sticking policy. Agent became addicted to the door opening action, receives constant reward and don’t want to move any further.

### 2.2 Suggested solution

To overcome, first of all, the Noisy TV issue, and also to utilise curiosity metric to stimulate agent to explore more, the following architecture is suggested (Figure 2.1):

Next-state prediction, which is usually used for curiosity metrics, is vulnerable to Noisy TV problem. Using one additional neural network (target network) in the architecture allows us to solve this issue. So, apart from predictor and policy networks, solution offers target network which purpose is following.
Target network has random, static weights. It has never been trained. It only gives a constant output for any given state, and this output is the same for same state at any moment of time, but is always different for different state inputs:

\[ f_i(x) = f_j(x), f_i(x) \neq f_i(y) \]

Next, predictor is trying to predict target network output. Its purpose is to minimise the difference between their outputs. Over time, such prediction becomes more and more accurate for known states but stays bad for new states. This affects intrinsic reward: higher errors give higher rewards. Since revisiting known states gives low intrinsic reward, agent is motivated to search for unknown cases. This makes difference with existing approaches: algorithm is not trying to predict next state and calculate its error for this prediction. Instead, algorithm predicts novelty of next state, calculating how much it doesn’t know about it yet. Both networks are shown on Figure 2.2.

![Figure 2.2 Predictor and target networks](image-url)
With such approach, using two networks instead of one doesn’t help so solve the Noisy TV problem directly. However, if agent is trapped in front of any Noisy TV, the target network generates a lot of different outputs, and predictor learns to predict them. At some point, predictor is good enough to be quite precise in those predictions, which gives less reward and agent is no longer attracted to this TV.

Nice benefit is that ending an episode is also an unwanted scenario for the algorithm. Start of each episode is the most familiar, and also the most boring state for it. Death screen or start screen gives less reward, and agent is trying to stay alive as long as it can.

Since predictor always predicts just the state of another network, which is environment independent, there is no more need to find good predictor architecture for each task. This makes suggested approach more flexible comparing to next-step predictors.

On the other hand, intrinsic reward will decrease as more states become familiar. It might be that certain cases require more curiosity or, opposite to this, require agent to visit some place many times somehow receiving relatively high reward every time. This demands sensitive tuning of hyperparameters when switching to the new environment. But this disadvantage can be partially solved by normalising intrinsic rewards in each episode.

The last, policy network (Figure 2.3), decides about the next step based on current state input and internal trained model. It uses CNN to consume environment state, and recurrent layers to capture more context of the previous states than bare CNN. It also has a policy optimiser, which uses the Actor Critic model. Main purpose of policy optimiser is to reduce the size of changes made to the policy strategy, making them smooth.

Policy optimiser treats intrinsic and extrinsic reward separately, providing again more flexibility. Extrinsic rewards are handled only after episode is over. This makes sure that agent can’t develop any reward hack (more detailed description of reward hacking problem is provided in Chapter 4). Intrinsic rewards are calculated regardless of episode length, but after fixed amount of steps. Using non-episodic rewards encourages agent to take risky approach in exploration and trying some complicated mechanics, exactly like human do.
Sometimes episode ending is not a reason to give up with a certain path, and by this approach, agent keep trying.

![Figure 2.3 Policy network](image)

It is interesting that applying absolutely the same approach, but minimising intrinsic reward in policy optimiser instead of maximising it, we can have the opposite of curiosity agent. Such an algorithm will be using only the most known path, exploiting its knowledge instead of exploring environment. Since previous architecture has similarities with human’s curiosity, this one can be called a conservative agent.

To make a short conclusion for this chapter, the suggested approach is unique, first of all, because of target-predictor networks cooperation. It is simple to implement, but allows us to solve the Noisy TV problem. Also, the same architecture can be used for many environments with minimal changes. Policy optimiser, which manipulates with both non-episodic and episodic rewards, gives certain flexibility.
Chapter 3: Implementation and tests

This chapter describes some details about algorithm implementation, also providing reason- ning for certain tools.

The most generic choice was made related to programming language. Python is not the fastest language and has its downsides, but it is the most popular way to create AI-related projects. First of all, it has simple and clear syntax, so it is easy to learn or read. It also has several useful libraries, which makes researcher’s life easier. Among them Matplotlib, NumPy, OpenCV, TensorFlow and many others. Aside from this, as the shortest road is the most familiar one and Python became my primary programming language a few years ago, it is convenient choice for me personally.

3.1 Environment

Second general choice is environment where the implementation of suggested algorithm will be operating. In RL community nowadays it is widely popular to test algorithms in Gym sandbox (Brockman et al., 2016). In this work Gym is used as an environment as well.

Gym is a toolkit, designed to be used for RL. It provides useful tools for performance analysis and comparison evaluation. It is also compatible with TensorFlow, importance of this tool will be described later. By design, Gym is a collection of different environments, among which:

1. algorithmic environments;

2. Atari 2600 games environments;

3. Box2D 2D physics environments;

4. classic RL control tasks environments;
5. MuJoCo physics environments (not open source);

6. robotics environments based on MuJoCo;

7. text game environments.

Also, it is possible to design custom environments, that’s why Gym has several third-party extensions like Gazebo extension, some 2D transportation puzzles, 3D simulators, etc.

Several environments from Atari games were chosen. There are 118 different environments, each representing either image-based or RAM-based input of one of 59 Atari 2600 games. Some of the games are rather simple, others are difficult even for humans. As any game, each of those environments has straightforward goals: to gain more points, to receive more bonuses, to stay alive as long as possible. It is easy to evaluate progress because of straightforward goals and also it is easy to compare with average human progress. All environments have unified controls with just a few options, providing also unified feedback to agent, which makes it a perfect benchmark for algorithm evaluation. This is why Gym became de-facto a standard in the field, like ImageNet dataset in supervised learning.

For this thesis, Solaris and Venture environments were selected. Both games require from player to discover and explore levels, which perfectly matches with goals of this evaluation. Venture is an example of a game which is slightly complex but has a stochastic noise which might make suggested agent vulnerable. Solaris is a bit more simple game, but it has colourful details of interface which alter contextual meaning of input. It was chosen to test the capabilities of algorithm to handle such details because input to neural networks is converted to grayscale.

Solaris is a game about space travels. Player’s ship can travel to different sectors and quadrants of the galaxy to visit different planets. When navigating to the hostile battle group, space battle happens. During the travels, fuel must be managed as losing it costs life to the player. Fuel can be refilled by visiting friendly planets but may be lost during space battle.
Player has an option to visit planets in some quadrants. Friendly planets must be defended; player must rescue all cadets from enemy planets; and enemy corridor planets force player to traverse through fast-paced corridor. Main goal is to visit planet Solaris and rescue colonists from there.

Venture is about moving in the maze with 4 rooms. Player has bow and arrows and can shoot some monsters. Hall monsters can’t be defeated, but when player enters the room he can shoot monsters there. Monsters remain lethal if player touches them. The more time player spends inside room, the bigger the chance that Hall monster will come, and, being immortal, most probably kill player. The faster the room monsters are killed, the more points player gets. Ultimate goal is to collect treasures from all rooms and get as many points as possible.

Few other environments were tested too, PrivateEye and Pitfall!, but due to very slow progress tests in those environments were discarded. When for chosen games it is possible to see some progress after weeks of training, complex games like PrivateEye and Pitfall! would require even more resources.

### 3.2 TensorFlow

TensorFlow is a machine learning system based on deep learning networks. It can run on multiple platforms, including mobile, in clusters or standalone, on CPU or GPU. CUDA support makes it great for parallel GPU computations. It also gives a high-level API, which makes it easy to learn and use. But at the same time, it fulfils the need of bigger flexibility by Keras Functional API and Model Subclassing API for creation of complex topologies, as well as support of eager execution.

Because of at least those reasons, TensorFlow became so widely popular, that nowadays there are more than 57 000 public repositories on GitHub mentioning it.
Being extremely useful for researchers, TensorFlow is not that well supported at Mac OS. Macs have very raw support of external GPU, but some devices are powerful enough for such computations even with onboard GPU. TensorFlow doesn’t support GPU computations on Mac OS at all, providing CPU extension only. That’s why on my development machine it wasn’t possible to run software stack with necessary performance. Fortunately, I have access to powerful GPU machine running Ubuntu at work.

3.3 Hardware

I am lucky to have access to one of the most powerful machines in our city. Usually, it is used to train CNN models for production use and does it pretty well. Unfortunately, the machine wasn’t completely available all the time, so it wasn’t possible to use its full potential. But training algorithm with such great capabilities for parallel processing is hard to underestimate. Some details about hardware setup, which was used for testing suggested approach are listed in Table 3.1

<table>
<thead>
<tr>
<th>Component</th>
<th>Model</th>
<th>Characteristics</th>
</tr>
</thead>
</table>
| CPU       | Intel® Xeon® Platinum 8180 | • 28 cores  
            • 2.50 GHz (3.80 GHz on turbo frequency)  
            • 38.5 MB L3 cache |
| RAM       | 4 x Micron 36ASF4G72PZ-2G6D1 | • DIMM Synchronous 2666 MHz  
            • 0.4 ns  
            • 4 x 32 GB |
| GPU       | Quadro GV100 | • 640 NVIDIA Tensor cores  
            • 5120 CUDA cores  
            • 32 GB HBM2  
            • 870 GB/s |

Table 3.1 Hardware components
3.4 Experiments

Training model to the acceptable extent requires a lot of time, even on such powerful machine. Since computer used for training is not available at any moment, long-lasting experiments happened in two phases.

First, two rather shorter experiments happened in parallel. Each experiment was running in 5 parallel environments of the same game, either Solaris or Venture. They were running for about 14 days or 330 hours non-stop. All results for parallel environments for the same game look very similar, that’s why the average result was chosen to represent result per game.

Then for another experiment is was chosen to run only Venture in single environment. This experiment was running for roughly 49 days or 1191 hours.

During the training there was a recording of screen of the game, which shows agent actions and environment changes. Also, all metrics were recorded too. But not every episode was recorded, because not each of them shows changes that are valuable for this research. Details about episode selection will be covered in the next section.

One of the reasons why Solaris wasn’t chosen for the longer experiment is the amount of logs and data generated during training. Because agent is able to stay alive much longer in this game, difference in logs size is approximately 500 times comparing to the same time agent spent training in Venture.

3.4.1 Episodes filtering

When logs were stored for every episode and every step, it was impossible to store all playback data, which seemed to be beneficial for future analysis. To be honest, the latest experiment was ended only because agent learned how to play Solaris well and episodes became longer, which resulted in logs folder to exceed expected 100 GB limit. It blocked other work for a while, and experiments had to be stopped.
That’s why not all episodes are stored for playback. When agent reaches any new state or level it is recorded. But if agent revisits the same state, it is recorded only if the attempt number is a square. In such case, only initial visits are getting recorded frequently, rarely recording later visits of the same state or level.

After filtering, video of the short session for Venture environment contains 190 games, 34 episodes each. For Solaris this number is 850 games. The long experiment in Venture environment recorded video of 3715 games.

### 3.4.2 Network architectures

Both predictor and target networks follow usual CNN architecture with convolutional encoders followed by dense layers as it is described in (Mnih et al., 2015). But on the contrary, policy network architecture is inspired by (Cho et al., 2014) and uses RNN with GRU cells. RNN seems to help with capturing longer episode context and adjusting policy smoothly. This idea was confirmed with a short experiment, involving both CNN and RNN architectures for policy network of the suggested algorithm (Figure 3.1):

![Figure 3.1 CNN (orange) and RNN (purple) policy networks performance comparison (Venture environment)](image-url)
It is clear from this figure that RNN performs much better as architecture for policy network, so the final version of agent uses recurrent neural network.

3.4.3 Non-episodic rewards

As was explained above, extrinsic and intrinsic rewards are treated differently. To avoid reward hacking, extrinsic rewards are always episodic. But when human plays game, they transfer some knowledge and reward through several episodes. Some complex behaviour is developed after multiple failures. To mimic this mechanism, intrinsic rewards are calculated non-episodically, but during configurable time period. To prove that this theory actually works, again relatively short test was performed in Venture environment (Figure 3.2). In both cases policy network was using only intrinsic reward, so agent was capable to explore only.

Figure 3.2 Episodic (pink) and non-episodic (green) intrinsic reward performance in Venture environment
From the chart we can see that episodic reward increases slowly, but non-episodic intrinsic reward instantly give a significant boost to the episodic return. Also, for non-episodic return value decreases slightly with time because agent is visiting familiar places, and those states became even more familiar. In that way exploration bonus decreases and it became harder and harder to get it.

3.4.4 Comparison to existing solutions

The next test involves three agents: suggested approach; same agent, but without exploration bonus; and agent using curiosity metric, but with the next-state predictor. Test demonstrates different performance in different environments. Next-state predictor shows similar result for both Venture and Solaris, but exploration bonus helps a lot in case of Venture. It is not that noticeable for Solaris, because it doesn’t have those cases where agent can easily stick with some noisy behaviour. And, naturally, Venture is that kind of game where exploration plays much bigger role. Performances of all three solutions in Solaris are compared at Figure 3.3. Comparison for Venture environment is provided at Figure 3.4.

Figure 3.3 Suggested solution (blue), next-state predictor (light blue) and agent without curiosity bonus (green) performance in Solaris environment
As we can see here, agent without curiosity bonus performs with the same success, stable. Agent with next-state predictor exploration bonus improves slower and has more unstable result. Finally, agent with curiosity-driven exploration bonus performs similarly. But we can also see that in the end it performs slightly better. Unfortunately, this experiment was too short to show a big difference between exploration bonuses here. Solaris is not so much of an exploration game, but it has a lot of changing interface which distracts curious agent, and agent with extrinsic reward only works well too. Systematic drops of return show that agent learned the level to a certain extent and wasn’t able to receive enough reward. But after some time agent discovered a way to go to another level and learn new surrounding, which takes exploration bonus back to normal.

Figure 3.4 Suggested solution (blue), next-state predictor (light blue) and agent without curiosity bonus (green) performance in Venture environment

In Venture agent without exploration bonus doesn’t work that well anymore, because this game requires more discovery. We can clearly see from this chart, that exploration bonus
makes the difference here. Also, it is a bit more clear here that curiosity bonus works more stable and gives slightly better results than just next-state predictor.

3.5 Results evaluation

All experiments showed natural growth of reward both in total and for each episode. It is much more clear for the longer experiment, but trend is also visible with shorter experiments.

For 330 hour long experiment in Solaris internal return has noticeable peak at the beginning (Figure 3.5). It can be explained by all new information and blinking interface, which quickly became usual. Since it is quite simple game, and interface elements repeat predictably, internal return stays low. But in general, total reward increases almost linearly (Figure 3.6).

![Figure 3.5 Mean internal (left) and external (right) return for the shorter experiment in Solaris environment](image)
Figure 3.6 Episodic (left) and total (right) reward for the shorter experiment in Solaris environment

Situation with Venture is slightly different. As this game is more difficult, charts for both internal and external returns are noisier (Figure 3.7):

Figure 3.7 Mean internal (left) and external (right) return for the shorter experiment in Venture environment

First of all, we don’t see a spike in the beginning, because agent needs to learn how to move and it is much harder to change picture since in this game agent must move by itself. Doing random actions without any policy or even doing nothing in Solaris still give all possible screens in the very first episode. Second, when agent learns how to move and discovers first sub-level, it fails to learn that enemies are bad. Enemy is a random moving blinking object, which attracts curious agent. But, fortunately, with experience agent learns how to defeat enemies. However, agent often fails to run away from Hall monster, which
appears if player spends too much time on sub-level. It is blinking object which agent can’t kill, and it moves too fast so it almost always kills agent.

The reward chart doesn’t look as good as the one for Solaris environment, but still, agent managed to develop some policy and increase the received reward quite a lot. It is interesting that the last part of the chart shows more rapid growth of generated reward, which means that agent figured out how to operate in the environment and started to act smarter (Figure 3.8):

![Figure 3.8 Episodic (left) and total (right) reward for the shorter experiment in Venture environment](image)

The last, 1191 hour long experiment with Venture showed that, as expected, with time agent learns more. All numbers are just on different scale, but other things look almost the same at least if we compare with experiments in the same environment. Total reward chart looks very smooth and linear because all small noises and slowdowns are covered by the scale of changes (Figure 3.9). Internal and external returns are not stable and charts look noisy, but still show noticeable improvement of exploration over time (Figure 3.10).
Analysing all results together, it can be said that agent successfully proved applicability and benefits of the suggested approach, and produced a decent result. Agent is stimulated to discover more, and it is what it does the best. But still, progress is very slow, and even for the longest experiment it is good, but not great. That’s why the next chapter provides comprehensive analysis of reinforcement learning problems nowadays, suggests some solutions, improvements and even covers a few generic problems, peculiar to machine learning.
Chapter 4: Problems of reinforcement learning

This chapter provides an overview of the weaknesses which reinforcement learning has now. Statements are made from analysis of recent papers, published in this field.

Reinforcement learning nowadays has two main problems (and always had them before). First, it is not easy to assign a reward to some actions when their complexity increases, which called the long term reward assignment. Second, agent always needs to keep balance between exploring the world around and using accumulated experience. These problems are special for reinforcement learning exclusively (Sutton and Barto, 2018). Of course, there are problems of learning from humans, learning in multi-agent environments, learning in conditions of partial observability, learning together with humans and many other problems, but they are mostly consequences of these two main problems.

4.1 Reward assignment

How does the agent acquire rewards in the world of reinforcement learning? They do random things and see what happens next. And it is quite straightforward when every action leads to clear result: was it wrong or bad choice; also, when agent receives feedback on every step. Unfortunately, it is not usually the case.

We want from agent to be able to solve complex tasks. We don’t want to control every movement. If a two-legged agent must learn how to walk forward, it needs to find a strategy of controlling several joints at the same time, and do some actions in particular sequence. Only after few steps are done human can see that this attempt must be rewarded. But for agent there were thousands of decisions made. Figuring out which series of actions are actually responsible for the high reward is the problem of credit assignment.

And with more computation power, with hardware improvements, this problem only gets worse. Algorithms can perform more operations per second. The scale of tasks complexity
increases. And if there is too big delay between choices made and reward assignment, any algorithm nowadays will just fail — even if the choice was right.

How to overcome it? There are a few solutions. First, we could assign rewards more frequently, i.e. reduce the scale at which rewards are generated. Which leads us to another problem, so-called reward hacking (Jack Clark, 2016). This happens because agent is not optimising immersive rewards, but it is optimising value landscape. And value landscape might contain a lot of such exploits for agent, if wasn’t considered carefully.

All this leads to a natural question — why rewards are the way to define goals? The answer is: because in this case optimisation can be applied and it will produce better policy. Reward is the most convenient way to embed field-specific knowledge to agent from operator.

There are other, better ways to specify goals — like imitation learning, where it is possible to take labels from the target distribution, meaning a priori optimal rule. Hierarchical reinforcement learning is also a way to go. In this case, we are trying to separate the problem into several subtasks with different, own subgoals (Andrew G Barto and Sridhar Mahadevan, 2003). Decompositions allow us to set wider timeframes for decision making, and still get an effective result. Sometimes policies, achieved during subtasks, can be even helpful in other subtasks.

Hierarchical RL can give us a scalar reward, back propagated through the Markov chain over any amount of levels of hierarchy (in theory). However, nowadays goal transfer is still a hard task, and current solutions use only one level of depth.

4.2 Exploration and exploitation

On every step agent learns (hopefully), how to answer the following question: should it take a new path, containing relatively semi-optimal actions, which may lead it to a bigger
reward, or is it better to follow the same policy if it works? And there is no correct answer because it is always a compromise between these two.

Bellman’s equations ensure that optimal function value can be achieved if and only if every state was visited an infinite amount of times, and every action was tested also an infinite amount of times on each of them. Which is, obviously, not possible in real life, since we don’t have infinity in our hands.

\[
Q_v(s, a) = \mathbb{E}(r_t + \gamma \max_a Q_v(s_{t+1}, a') | s_t = s, a_t = a) = \sum_{s'} p(s' | s, a) [R(s, a, s') + \gamma \max_a Q_v(s', a')]
\]

Of course, there is no real need in optimal strategy, usually. It is sufficient to find good enough policy. Basically, if policy can be found relatively fast and don’t break everything — it is a good policy, even if it takes millions of steps to find it. Millions of steps are still less than any infinity. And this number grows exponentially if actions space or states space increase (Sutton and Barto, 2018).

The best way is to choose random exploration. Agent never knows, and never can be sure that some action will give lower reward unless agent has already tried this action. Moreover: generic ways of developing policy don’t provide enough information about optimal strategy and environment changes the picture. Moving from generic solutions to more specific also has its own, very clear, problems. Among them complexity, necessity to configure algorithm for every new task, dependency on environment and many others.

Even when random action gives to agent huge reward, it doesn’t mean that the same action will give same reward also next time. Wise agent just slightly increases value in its policy for transition to this state if reward stays constant. We end up teaching agent how to make small, conservative updates to functions that are trying to approximate expectations of arbitrarily complex probability distributions over an arbitrarily large number of states and actions.
The necessity to make an infinite amount of action in infinity states infinity times requires from us some kind of generalisation, generalisation of existing knowledge to hidden yet knowledge. Fortunately, solution already exists. Unfortunately, it is supervised learning, and by nature, it can’t be applied for solving reinforcement learning problems, not yet.

De-facto, we could approximate goal function, which means that we don’t need a giant dictionary of states and actions, but can resolve any state-action pair at any moment by passing them to our function. Which is really close to supervised learning, it seems.

And neural networks are pretty much ineffective in this task because of the slow pace of gradient descent.

During exploration face and data gathering agent can’t have ground truth to be fixed — this is what makes it different from supervised learning. Agent is constantly changing its attitude to the same things. And the only possible way to overcome this and find an optimal solution is to continue random exploration. Policy obtained is never optimal, but here is what makes RL so useful and unique: agent still learns and improves, even with non-optimal policy or function.

So, we can’t dramatically increase exploration because it will lead agent to unexpected changes in the target landscape. And, as described above, noticeable changes in exploitation value also would ruin everything. It makes training set more complex, comparing to supervised learning, as well as it makes RL sensitive to initial values. And there is no stable training data, because training data is dependent on algorithm output and stochastic nature of environment. This is why different runs of RL algorithms may give quite different results.

RL is made to act in continuous action spaces, and the most popular way to do it is to utilise one of the on-policy methods. They are really great when you have a chance to learn the optimal policy by observing another policy being executed. But then such methods are consistent with current policy, so whenever you update current policy — previous experience has no use anymore.
That all conclude not only the main problem of reinforcement learning, but AI in general: exploration problem. It takes so much time at the beginning for any algorithm because agent has zero knowledge about everything at all. All priors, all accumulated knowledge allow us, humans, learn things faster, flexibly apply achieved experience to new tasks. RL needs to explore randomly, and cannot limit its exploration to a small set of promising states like we do.

Apart from these main problems, I would like to demonstrate a few others, more specific RL problems.

### 4.3 Effectiveness

Long story short, before applying RL in your solutions, it worth to check other methods.

MuJoCo, physical engine is very popular as a benchmark. Some researches even use it as a sandbox for their models which must operate in real world later.

DeepMind uses it also in their white paper about learning complex locomotion behaviours (Heess et al., 2017). They don’t specify in the article exactly how their algorithm was trained, but from the code it looks like it required more than 6400 hours of training all together. Results are impressive, especially for RL, but it still fails quite often (and does it quite funny, to be honest). For real tasks, it would be better to choose another ML approach and get these results much faster (Lin, 1992).

Atari games are classical benchmark for reinforcement learning because they are quite simple, play on constant 60 FPS and have a straightforward goal: get as many points as possible (however, some games also used to evaluate other abilities like exploration). And, of course, results can be compared to average human results.

Rainbow Deep Q-Network (Hessel et al., 2017) shows impressive results and beats human in 40 games out of 57. It uses Q-Learning with neural networks of acceptable size and also
applies some optimisation tricks. Comparison of results to other architectures can be seen on this chart (Figure 4.1):

![Median human-normalised performance across 57 Atari games.](image)

**Figure 4.1 Median human-normalised performance across 57 Atari games.**

Comparison of rainbow DQN with other algorithms

Results by vertical axis are calculated as average median result of 57 different DQN networks, one for each game, and human result is taken as 100%. Horizontal axis represents millions of frames required to learn for this result. Rainbow DQN needs about 18 millions of frames to beat human, which is ~83 hours of playing when human needs only a few minutes for most of these simple games.

Let’s take a look at other candidates too. Distributional DQN (Bellemare et al., 2017) needs 70 millions of frames, which is almost 4 times slower than Rainbow DQN. And on a third place there is Nature DQN (Mnih et al., 2015), but even with 200 millions of frames it can’t beat human — and never win against humans at all.
4.4 Alternatives to Reinforcement Learning

Reinforcement learning is a beautiful and idealistic idea, but usually there is no requirement to solve a problem with RL technics. RL is universal and can solve any task, that’s why it is not the best for particular requirements. Specific algorithms usually work better and faster. And in real life, unless you are developing reinforcement learning for reinforcement learning, you just need it working reliably and fast.

When we start, RL doesn’t have any understanding about environment. It doesn’t know about gravity, about enemies, about health or traps or anything else. It doesn’t use models, because it is a universal approach. And usage of models makes an algorithm-specific, but also gives it an advantage in speed and precision. Using search method Monte-Carlo, MCTS and it’s implementation in agent UCT (Bellemare et al., 2013, Guo et al., 2014) it is possible to get much better results (Table 4.1):

<table>
<thead>
<tr>
<th>Agent</th>
<th>B.Rider</th>
<th>Breakout</th>
<th>Enduro</th>
<th>Pong</th>
<th>Q*bert</th>
<th>Seaquest</th>
<th>S.Invaders</th>
</tr>
</thead>
<tbody>
<tr>
<td>DQN</td>
<td>5184</td>
<td>225</td>
<td>661</td>
<td>21</td>
<td>4500</td>
<td>1740</td>
<td>1075</td>
</tr>
<tr>
<td>-best</td>
<td>4092</td>
<td>168</td>
<td>470</td>
<td>20</td>
<td>1952</td>
<td>1705</td>
<td>581</td>
</tr>
<tr>
<td>UCC</td>
<td>10514</td>
<td>351</td>
<td>942</td>
<td>21</td>
<td>29725</td>
<td>5100</td>
<td>1200</td>
</tr>
<tr>
<td>-best</td>
<td>5342(20)</td>
<td>175(5.63)</td>
<td>558(14)</td>
<td>19(0.3)</td>
<td>11574(44)</td>
<td>2273(23)</td>
<td>672(5.3)</td>
</tr>
<tr>
<td>-greedy</td>
<td>5676</td>
<td>269</td>
<td>692</td>
<td>21</td>
<td>19890</td>
<td>2760</td>
<td>680</td>
</tr>
<tr>
<td>UCC-I</td>
<td>10732</td>
<td>413</td>
<td>1026</td>
<td>21</td>
<td>29900</td>
<td>6100</td>
<td>910</td>
</tr>
<tr>
<td>-best</td>
<td>10388(4.6)</td>
<td>215(6.69)</td>
<td>601(11)</td>
<td>19(0.14)</td>
<td>13189(35.3)</td>
<td>2701(6.09)</td>
<td>670(4.24)</td>
</tr>
<tr>
<td>-greedy</td>
<td>5702</td>
<td>380</td>
<td>741</td>
<td>21</td>
<td>20025</td>
<td>2995</td>
<td>692</td>
</tr>
<tr>
<td>UCR</td>
<td>2405(12)</td>
<td>143(6.7)</td>
<td>566(10.2)</td>
<td>19(0.3)</td>
<td>12755(40.7)</td>
<td>1024(13.8)</td>
<td>441(8.1)</td>
</tr>
</tbody>
</table>

Table 4.1 Performance (game scores) of the four real-time game playing agents, compared to the off-line UCT game playing agent

In this table UCR is short for UCTtoRegression, UCC is short for UCTtoClassification, and UCC-I is short for UCTtoClassification-Interleaved
The same can be said about complex moving behaviours in MuJoCo. Online trajectory optimisation (Tassa et al., 2012) shows how correct results can be obtained in almost real time, without thousands of hours of training. And this research was released 7 years ago, using that time hardware.

Comparing to any other non-RL algorithm, reinforcement learning almost always gives worse results because of its non-specific idea. Still, RL has chances to become the most important branch of ML in the closest future, just not yet. And, probably, the best evidence of RL potential is AlphaGo, when RL actually beats all other techniques. And human.

4.5 Overfit

Overfit is something that every researcher tries to avoid, but it is an integral part of the work of almost any reinforcement learning algorithm. When you are solving one particular problem in some environment, it isn’t a huge deal. But overfitted model doesn’t work in another environment. That’s why the article (Hessel et al., 2017) uses 57 different models instead of one. When there is work around tuning one model to play in another environment (Rusu et al., 2016), usually result is not guaranteed and is rather a surprise than expected outcome.

A very interesting example of this problem can be found in (Lanctot et al., 2017). Two agents were playing laser tag game, and several pairs of agent were playing separately. When authors took one agent from one pair and forced it to play against another agent from another pair, it turns out that they can’t do anything at all.

It isn’t really clear why we see such results in these cases. The key might be in agent’s speed of learning. Those which started together compete with almost equal forces. And, vice versa, different agents might (and, most probably, will) have different learning speed, so the stronger will be exploiting errors of the weaker agent, which lead to overfitting.
Multiagent learning is a really trending topic now, and the idea is brilliant: when the “student” network learns how to cheat and, for example, make generated human face look like real, “teacher” learns to detect fakes better and better. It gives quite a few advantages, but, unfortunately, reinforcement learning doesn’t seem to be the field where this approach can be applied. Symmetrical game is the closest available method for RL. And it is a crucial part of such successful algorithms like AlphaGo, AlphaZero and Dota 2 Shadow Fiend.

### 4.6 Reward hacks

Reward hacks is an interesting topic when it comes to RL problems. It is often quite funny to find some new behaviour which algorithm developed trying to achieve goal. These new behaviours are not what authors wanted from their code, so results aren’t useful at all, but in my opinion, it is also something which helps to promote RL — even when it shows how stupid RL can be. With sophisticated behaviours, it always looks like algorithm has its own consciousness. Those creative ways of hacking experiment entertain, but also let us take RL like something more human-like.

Creation of an algorithm for RL requires careful selection and shaping of the reward function, as was mentioned earlier. When some hidden options are not taken into account, algorithm may develop unexpected ways to gain a reward. From program’s point of view there is no problem at all: in reinforcement learning algorithms must be field- and model-agnostic. It just tries its best to get as much reward as possible. When for experimenter this behaviour may look like ridiculous or just unacceptable way of solving the task, because human knows already what they expect as an end result.

Here is an example of reward hack. Agent has to walk around the room, policy defines reward when goal point inside the room is reached. The episode is over when agent escapes from room boundaries. However, there is no punishment in this case. After training agent was always killing itself first thing in every episode. It was too hard to achieve any positive
reward and too easy to receive negative reward. This suicidal behaviour gave him a fast neutral result, which considered better than the long episode with a lot of negative scores.

There are several ways to overcome such issues:

• Disperse reward. As mentioned earlier, there are many more decisions than we imagine between simple actions. We must give smaller rewards more frequently, and reward at the end of the episode is not enough for these cases.
• Shape rewards until they start to work. It is a long and unpleasant path.
• Use transfer learning to train in different environments and get “common sense” of those environments. It is similar to how humans act, so this is the most preferable way but at the same time the most challenging.
• Learning from demonstrations. Basically, it is learning how human would solve the problem. It can give some initial direction to the policy which can be further developed.
• Human feedback after each episode. Similar interactive approach is applied in Boston Dynamics algorithms for their robots, while their algorithms are not RL by any means. This small feedback could prevent unusual behaviour and fix those problems on early stage.

Local optimums may look similar to reward hacks, but they are actually problems of balance between exploration and exploitation. Like in (Gu et al., 2016), where agent in Half-Cheetah environment supposed to move forward using normalised advantage function. Cheetah-agent makes a flip at the beginning of episode and continues moving forward swinging its legs. Doing definitely not what authors designed it for, agent gets a reward and don’t want to change its behaviour. Some more productive ways of moving could give more reward, but it is hard to find them when you are already in local optimum.

It happened because model finds out during training, that flip is beneficial comparing to freezing at the start. After this behaviour was repeated a certain amount of times, it became a positively reinforced action. Later, model learned that after applying small force it is pos-
sible to get a bit more reward by shaking legs and moving slightly further. Repeating this action, model reinforced this new part of behaviour. It became much harder to learn “proper way” of walking; it is much easier to get small, but well-known reward, then take a risk and try something new. This is caused by too high exploitation, and makes agent act definitely not as we expected from it.

When it is funny and harmless in the context of virtual environment, some people are afraid that if similar situation will happen in human world, consequences can be much more dangerous. And while some of those discussions happen only as imaginary experiments (Paperclip maximiser, 2019), there are already some works and publishings about the potential negative impact caused by lack of safety regulations in AI (Amodei et al., 2016).

Authors of the article (Houthooft et al., 2016) about stacking lego bricks also faced several funny reward hacks. To overcome them, they had to find quite complex goal function. Their function for the last subtask looks like this (Figure 4.2):

\[
\begin{align*}
    r(b_i^{(1)}, s^P, s^{B1}, s^{B2}) &= \begin{cases} 
        1 & \text{if stack}(b_i^{(1)}, s^P, s^{B1}, s^{B2}) \\
        0 & \text{otherwise} 
    \end{cases} \\
    r(b_i^{(1)}, s^P, s^{B1}, s^{B2}) &= \begin{cases} 
        1 & \text{if stack}(b_i^{(1)}, s^P, s^{B1}, s^{B2}) \\
        0.25 & \text{if } \neg \text{stack}(b_i^{(1)}, s^P, s^{B1}, s^{B2}) \land \text{grasp}(b_i^{(1)}, s^P, s^{B1}, s^{B2}) \\
        0 & \text{otherwise} 
    \end{cases} \\
    r(b_i^{(1)}, s^P, s^{B1}, s^{B2}) &= \begin{cases} 
        1 & \text{if stack}(b_i^{(1)}, s^P, s^{B1}, s^{B2}) \\
        0.25 & \text{if } \neg \text{stack}(b_i^{(1)}, s^P, s^{B1}, s^{B2}) \land \text{grasp}(b_i^{(1)}, s^P, s^{B1}, s^{B2}) \\
        0.125 & \text{if } \neg (\text{stack}(b_i^{(1)}, s^P, s^{B1}, s^{B2}) \lor \text{grasp}(b_i^{(1)}, s^P, s^{B1}, s^{B2})) \land \text{reach}(b_i^{(1)}, s^P, s^{B1}, s^{B2}) \\
        0 & \text{otherwise} 
    \end{cases} \\
    r(b_i^{(1)}, s^P, s^{B1}, s^{B2}) &= \begin{cases} 
        1 & \text{if stack}(b_i^{(1)}, s^P, s^{B1}, s^{B2}) \\
        0.25 + 0.25r_{S2}(s^{B1}, s^P) & \text{if } \neg \text{stack}(b_i^{(1)}, s^P, s^{B1}, s^{B2}) \land \text{grasp}(b_i^{(1)}, s^P, s^{B1}, s^{B2}) \\
        0.125 & \text{if } \neg (\text{stack}(b_i^{(1)}, s^P, s^{B1}, s^{B2}) \lor \text{grasp}(b_i^{(1)}, s^P, s^{B1}, s^{B2})) \land \text{reach}(b_i^{(1)}, s^P, s^{B1}, s^{B2}) \\
        0 + 0.125r_{S1}(s^{B1}, s^P) & \text{otherwise} 
    \end{cases}
\end{align*}
\]

Figure 4.2 Goal function for lego brick stacking agent

And it seems like a huge barrier in finding working solution with RL methods. Which leads us to the next problem.
4.7 Stability

In machine learning almost always algorithm is tweaked with hyperparameters, either randomly or manually chosen. Reinforcement learning uses them too, of course. And being in general very sensitive to any changes in implementation, changing hyperparams makes RL dramatically unstable.

Good example can be found in (Popov et al., 2017). Authors use sparse reward in MuJoCo’s HalfCheetah testing TRPO algorithm (Figure 4.3). This chart represents the average return (vertical axis) at each iteration (horizontal axis). Dark line shows median result for 10 random seeds, and coloured area is the result between 25th and 75th percentile. And it is quite clear that the line of the 25th percentile is very close to 0. If it is just because of random seed, how much it is better than random?

![Figure 4.3 Results of TRPO algorithm in HalfCheetah environment](image)
Supervised learning or any other machine learning approach would be considered as failing if it can’t provide a result better than random in 25% of cases. With RL you can’t be sure if it is because of wrong hyperparameters, bug in implementation or just unfortunate location of Uranus towards Mercury Retrograde.

“SL wants to work. Even if you screw something up you’ll usually get something non-random back. RL must be forced to work. If you screw something up or don’t tune something well enough you’re exceedingly likely to get a policy that is even worse than random. And even if it's all well tuned you'll get a bad policy 30% of the time, just because.

Long story short your failure is more due to the difficulty of deep RL, and much less due to the difficulty of "designing neural networks” (Andrej Karpathy, 2017).

And solving this issue again pushes us back to the complexity problem. It is hard to prove any algorithm when it is so unstable. Even if it works well, it might fail because of just something random, or hyperparam which is not set correctly. And it obviously takes time to check the hypothesis with all variety of hyperparams in all necessary random inputs.

Authors of the article (Henderson et al., 2017) calculated, how dramatic can be the difference when we are talking not only about random fluctuations in results of different runs, but when algorithm really changes. There are some conclusions:

- When it is common in the field to use 5 random seeds for reporting algorithm results, it is possible that they form non-overlapping intervals.
- Such hyperparameters as reward scaling may cause a huge difference in results. Hyperparameters must be algorithm agnostic.
- Different implementations of the same algorithm may show a totally different performance. Researchers must document every smallest step and even insignificant parameter to make their work reproducible.

RL is also very sensitive to data, because good examples do their job, but bad samples can make model inapplicable at all. Unlike other areas of ML, in such case reinforcement learning learns that any actions are pointless.
Conclusion

First of all, this work had to show what must be fixed in the field, and which downsides reinforcement learning has in comparison to other machine learning techniques. That’s why the thesis describes several weaknesses of reinforcement learning, valid at least for the situation now. In that way, Chapter 4 answers the first research question raised in the beginning: what are the main weaknesses of current reinforcement learning algorithms and what are the possible solutions to them?

It is possible to overcome some of the weaknesses, and part of them can be solved with the solution provided in this thesis. Main issue, balance between exploration and exploitation is resolved by combining the Actor Critic policy optimiser with curiosity bonus, so the agent always wants to explore, but also exploits existing knowledge about explored areas to find new ways to explore. Another problem, reward assignment, is also partially resolved by curiosity bonus. Together with a combination of episodic and non-episodic reward assignment, curiosity bonus fixes a sparse reward landscape. It is also easier to move existing implementation to the new environment because reward assignment is not model dependent. Which also makes it possible to run agent in parallel environments and addresses — just slightly and only in comparison to other reinforcement learning methods — training time issue. Reward hacking and in particular Noisy TV problem is fixed by suggested architecture or curiosity agent. All of the above answers second research question: how curiosity can help to overcome known problems in reinforcement learning?

After this, it is easy to answer the last question which this thesis tends to resolve: how to implement an exploration bonus which encourages agent to discover more, while protecting agent from undesired environment stochasticity? Chapter 2 explains that prediction of not just reward gain of next state, but prediction of its novelty helps to overcome the issue with unpredictable stochastic noise of the environment. And Chapter 3 proves that suggested idea works, also explaining some deeper technical details of the implementation.
Even though research questions were successfully answered, the algorithm can’t solve all issues with reinforcement learning. It is slow, as can be slow RL. If it operates in an environment which doesn’t require a lot of exploration, agent is not that useful. It also seems that stability could be improved, probably by tuning hyperparameters more accurately.

As an improvement for this work, it would be beneficial to perform more comprehensive, longer tests, also in multiple parallel environments. It seems like better managing of parallel computations would improve results. Also, it would be an interesting experiment if agent could reuse knowledge generated in one environment for faster development in another environment if it is possible. Maybe, combining some other metrics apart from curiosity and balancing intrinsic reward between those metrics — like it is done with intrinsic and extrinsic rewards in policy optimiser — would give some interesting outcome too.

There is definitely room for further research, and since results are promising, curiosity metric looks like a good direction for some specific applications of reinforcement learning. However, being just an engineering trick, it can solve some problems separately from RL. For example, by utilising curiosity exploration metric scanning software can look for unchecked locations, chat-bot can deviate from the programmed narrative and text analyser can diversificate knowledge bank.

There is also a particular case, for which I hope to use achieved knowledge at some point. Project I am currently working on is related to the autonomous robot, which navigates in the supermarket. One of the related routines is collecting a 3D point cloud of the area to generate a map. Now it is done by operator driving the robot around the store trying to cover all spots. The idea is to reuse the method for calculation of intrinsic reward. It is not obligatory to use a reinforcement learning approach, of course. But with implemented curiosity robot can be navigate around when it doesn’t have a map of the environment ready yet. Looking for any undiscovered spot within a certain area, robot would create map without blind spots. As a result, map can be created with the operator being remote, and robot becomes autonomous even earlier, simplifying initial setup.
Bibliography


Browne, Cameron B., Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and


Appendices

A. Selected milestones during agent training

Figure A.1 Venture. Agent visits sub level for the first time, intrinsic reward spike

Figure A.2 Venture. Agent kills both enemies, extrinsic reward spike
Figure A.3 Solaris. Agent kills enemy, extrinsic reward spike. One of the latest episodes, intrinsic reward is low.

Figure A.4 Solaris. Agent leaves planet for the first time, intrinsic reward spike. Intrinsic reward level is high due to the novelty of everything in the beginning.
Figure A.5 Venture. Agent blocks itself in the corner, can’t move — the gap at intrinsic reward chart

Figure A.6 Venture. Agent leaves room for a moment and sees familiar hall — drop at intrinsic reward chart
Figure A.7 Venture. Agent leaves room and sees familiar hall — drop at intrinsic reward chart — but then moves around the hall to an unusual location, gaining new experiences.

Figure A.8 Venture. Agent travels around the hall and enters the unfamiliar room, even kills one enemy — intrinsic reward spike.
Figure A.9 Venture. Agent travels around the hall and enters room for the first time, intrinsic reward spike