UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

DEEP REINFORCEMENT LEARNING FOR COMPUTER GAMES

Diplomová práca

2016 **Ing. Matúš Tuna**

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

DEEP REINFORCEMENT LEARNING FOR COMPUTER GAMES

Diplomová práca

Študijný program: Kognitívna veda
Študijný odbor: 2503 Kognitívna veda
Školiace pracovisko: Katedra aplikovanej informatiky
Školiteľ: prof. Ing. Igor Farkaš, Dr.

Bratislava 2016 **Ing. Matúš Tuna**

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Ing. Matúš Tuna
Study programme:	Cognitive Science (Single degree study, master II. deg., full time form)
Field of Study:	Cognitive Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Deep reinforcement learning for computer games

Aim:
1. Survey the deep reinforcement learning literature.
2. Focus on deep Q-learning algorithm and apply it to a chosen computer game, evaluate its performance.
3. Analyse potential applications of deep reinforcement learning in robotics and other areas of artificial intelligence.

Literature:
Mnih V. et al. (2015). Human-level control through deep reinforcement learning, Nature, 518 (7540), pp. 529-533.
Mnih V. et al. (2013). Playing Atari with deep reinforcement learning, Arxiv.org, http://arxiv.org/abs/1312.5602.
Schmidhuber J. (2015). Deep learning in neural networks: An overview, Neural Networks, 61, pp. 85-117.

Annotation: Optimal action selection in complex and dynamical environments is one of the most significant problems in cognitive science and artificial intelligence. Deep networks, capable of learning complex mappings from inputs to outputs, combined with reinforcement learning based on ecologically feasible feedback, provide a promising approach to solving these difficult tasks.

Keywords: reinforcement learning, neural networks, deep learning, games

Supervisor:	prof. Ing. Igor Farkaš, Dr.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	prof. Ing. Igor Farkaš, Dr.
Assigned:	09.11.2014

Approved: 09.11.2014 — prof. Ing. Igor Farkaš, Dr.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Čestne prehlasujem, že som túto diplomovú prácu vypracoval samostatne s použitím citovaných zdrojov.

Ing. Matúš Tuna

Chcem sa poďakovať svojmu školiteľovi prof. Ing. Igorovi Farkašovi, Dr. za cennú pomoc, rady, konzultácie, čas a ochotu ktorú mi venoval počas písania diplomovej práce.

ABSTRAKT

Témou tejto diplomovej práce je najnovší výskum v oblastiach učenia posilňovaním a hlbokého učenia. Táto diplomová práca sa zameriava na algoritmus hlbokého Q-učenia, ktorý v literatúre preukázal schopnosť učenia sa efektívnych rozhodovacích stratégií priamo z vysokorozmernej reprezentácie prostredia v doméne jednoduchých počítačových hier. Hlavnými prínosmi tejto práce je prehľad a analýza súčasného výskumu v oblasti algoritmov učenia posilňovaním založených na neurónových sieťach a ich aplikácií, ako aj replikácia niektorých výsledkov z literatúry hlbokého Q-učenia. Na základe analýzy relevantného výskumu sme navrhli niekoľko metód založených na súčasnom výskume v oblasti hlbokého učenia, ktoré by mohli byť použité na rozšírenie a vylepšenie algoritmov hlbokého učenia posilňovaním, ako napríklad hlbokého Q-učenia, tak aby takýto algoritmus mohol "emulovať" niektoré kľúčové kognitívne kompetencie prítomné u ľudí a zvierat, ktoré sú potrebné na to, aby mohol umelý agent pôsobiť "kompetentne" vo svojom prostredí. Taktiež sme vykonali niekoľko experimentov hlbokého Q-učenia s využitím našej vlastnej implementácie. Počas týchto experimentov sa nám podarilo replikovať niektoré výsledky z kľúčovej literatúry hlbokého Q-učenia, ako aj zlepšiť niektoré výsledky pomocou alternatívnych algoritmov učenia a inicializácie váh.

Kľúčové slová: učenie posilňovaním, hlboké učenie, neurónové siete, hry.

ABSTRACT

The topic of this thesis are the recent advancements in the fields of reinforcement learning and deep learning. We focus on the Deep Q-learning algorithm that was previously shown to be capable of learning effective control policies from a high-dimensional representation of the environment in the domain of simple computer games. The main contributions of this thesis are the review and the analysis of the current research on the topic of the neural network based reinforcement learning algorithms and their potential applications, as well as the replication of some results from the Deep Q-learning literature. Based on the review and the analysis of the relevant research we proposed several methods based on the current deep learning research that could be used to extend and to improve deep reinforcement learning algorithms like Deep Q-learning so that this algorithm could "emulate" some of the core cognitive competences of human and animal minds that are necessary for artificial agents allowing them to act as competent agents in the environment. We also managed to perform a limited number of experiments with the Deep Q-learning algorithm based on our own implementation. We were able to replicate some results from the key Deep Q-learning literature and also improve these results by utilizing a different learning algorithm and weight initialization techniques compared to the original literature.

Key words: reinforcement learning, deep learning, neural networks, games.

Obsah

Introduction

Designing an artificial agent that can autonomously perform actions and learn with minimal amount of supervision is one of the biggest challenges for artificial intelligence. There are numerous practical applications of such agents, for example in self-driving cars, robotics, AI assistants and chatbots, or question answering computers.

One of the approaches to this challenge is the reinforcement learning (RL), which is a subfield of machine learning concerned with a question how should a system (an agent) select actions in some environment so as to maximize the cumulative value known as reward. An agent has to infer the optimal action from a reward that is determined by the environment and from observations of the state of the environment. Although there are many algorithms for learning optimal action selection function or policy, only the recent advancements in the field of deep learning made it possible to learn policies from the high-dimensional representations of complex environments. Our primary focus is on the Deep Q-learning algorithm, which uses deep neural networks to learn optimal policies based on the high-dimensional representation of the environment. So far, Deep Q-learning has been used for learning policies in various domains such as arcade games, robotic control tasks or board games. We think that deep reinforcement learning algorithms like Deep Q-learning could be used as a basis of a controller in more complex tasks, for example in self-driving cars, humanoid robots or AI assistants.

There are three main goals that we want to achieve in this work. In Chapter 1, we examine the theoretical basis of RL algorithms including the biological significance of this concept. In Chapter 2, we examine discrete and continuous versions of the Deep Q-learning algorithm. We also consider potential extensions of the Deep Q-learning algorithm based on the current development in the field of deep learning. More specifically, we will consider the possibility of endowing Deep Q-learning agent with the core cognitive competences or "ingredients" of the human intelligence that are necessary for artificial agents allowing them to act as competent agents in an environment. Among these competences are developmental start-up software, learning by rapid building of the models of environment and fast thinking. In Chapter 3, we will attempt to replicate and improve upon some of the results from Deep Q-learning paper by Mnih et al. (2013). For this purpose, we use our own Python based implementation of Deep Q-learning algorithm.

1. Theory of reinforcement learning

1.1 Reward, learning and control

The term reinforcement learning (RL) has two basic meanings [1]. One meaning refers to a general RL problem. The second meaning refers to the subfield of machine learning. The learning problem concerns the question of how should some system, otherwise known as an agent, select actions in some environment so as to maximize the cumulative value known as a reward. As a subfield of machine learning, reinforcement learning can be viewed as a collection of algorithms and techniques for solving learning problems.

Reinforcement learning differs in many ways from traditional machine learning algorithms. One major difference is that in RL there is no supervisor that tells the agent the correct answer to some problem. As a result, an agent has to infer correct answer from a scalar feedback signal known as a reward. Reward signal is determined by the environment and can be viewed as an indicator of how good (or bad in the case of negative reward) the agent is doing in the environment. The type and characteristics of a reward signal is determined by the environment and the task that our agent tries to accomplish. For example, for a humanoid robot in the environment of a typical household, positive reward would be awarded to such an agent if it accomplishes some household task, for example cleaning, and negative rewards would be awarded if the agent broke some property. Similarly, if the agent operated in a stock market environment, a positive reward would be awarded for making money on some investment and a negative reward for losing money on investment. The second major difference between reinforcement learning and supervised learning is that the feedback signal (reward) can in many environments be delayed and the decisions that an agent makes can have long term consequences. We can now formulate the basic structure of RL problems:

- at time t the agent receives an observation O_t and a scalar reward R_t,
- based on this observation and a policy that determines what action to take the agent selects an action A_t,
- environment receives an action A_t and emits an observation O_{t+1} and a reward R_{t+1},
- agent receives an observation O_{t+1} and a reward R_{t+1}, selects a new action A_t and, if necessary, modifies its policy.

Reinforcement learning agent can operate in two basic types of environments. The first type is fully observable environment. In this environment the state and the dynamics of the environment are fully known to the agent. Formally, we can say that an environment is fully observable if all states of the environment have a Markov property, which means that the current state is all that is needed to determine the future dynamics of the given environment, therefore the history can be omitted. We can define a Markov state by the following equation where p denotes probability, S denotes the state and t denotes time step.

$$p[S_{t+1}|S_t] = p[S_{t+1}|\, S_1, \dots, S_t]$$

This kind of environment, in which every state has a Markov property, can in RL be modeled using a Markov decision process, which is a memoryless random process defined as a tuple $\langle S, A, P, R, \gamma \rangle$ where S denotes states of an environment, A denotes a set of possible actions, P denotes probability of a transition from state S to so a successor state S' when taking some action a, R denotes expectation of a reward if agent performs an action a in state s and γ denotes a discount factor $\gamma \in [0,1]$ which is introduced to reduce importance of future rewards. Formally we can express the state transition probability P and the reward transition probability R by the following equations, where E denotes expectation.

$$P_{ss'}^{a} = p[S_{t+1} = s'|\, S_t = s, A_t = a]$$

$$R_s^a = E[R_{t+1}|\, S_t = s, A_t = a]$$

A large number of RL problems can be modeled using Markov decision process formalism, for example:

- decision making in board games such as backgammon, chess or Go,
- decision making in many computer games where current screen depicts the whole game space, such as Packman, Tetris or Breakout,
- control of autonomous vehicles,
- control of humanoid robots.

Considering the large number of real world problems that can be modeled using a Markov decision process and current state of RL research, in the rest of the work we will be considering only problems that are modeled using Markov decision process. However, there are many problems where the environment is not assumed to be fully observable that cannot be modeled by Markov decision process. This kind of problems can be modeled

using partially observable Markov decision process. We consider RL problems with partially observable environment to be an interesting and beneficial avenue for future research.

Besides the environment in which a RL agent operates, there are three major components of RL agents. These components are:

- policy, which defines agent's behavior
- value function which defines how good or bad each state and/or each action is,
- a model which is the representation of dynamics of agent's environment.

We will now describe each of these components from the perspective of Markov decision process formalism.

Policy is a function that defines the behavior of a RL agent in some environment. In the case of fully observable environments, policy is only dependent on the current state of environment. Thus policy can be defined as a function π that maps states of an environment to actions. Policy can be either deterministic in which agent always takes the same action if it is in particular state, or stochastic, in which agent takes some action in particular state with certain probability. Therefore we can define deterministic and stochastic policies by the following equations.

$$\pi(s) = a$$

$$\pi(a|s) = p[A_t = a|S_t = s]$$

Given some known Markov decision process and a stochastic policy π, we can now define state transition probability $P_{ss'}^{\pi}$ and reward transition probability R_s^{π} in the following way.

$$P_{ss'}^{\pi} = \sum_{a \in A} \pi(a|s) P_{ss'}^{a}$$

$$R_s^{\pi} = \sum_{a \in A} \pi(a|s) R_s^{a}$$

Value function "informs" our RL agent of "goodness" or "badness" of some particular state of an environment. There are two basic types of value functions. The first type is the so-called state-value function $V_{\pi}(s)$ which is defined as an expected cumulative discounted reward (return) from some state s and then following a policy π. State-value function can be expressed using following equation where G_t denotes a return or an expected cumulative discounted reward and T denotes final time step at the end of an episode.

$$V_\pi(s) = E_\pi[G_t | S_t = s]$$
$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-1} R_T$$

The second type of value function is the so-called action-value function $Q_\pi(s)$ which is defined as an expected cumulative discounted reward (return) from some state s, when our agent took an action a and then followed a policy π.

$$Q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a]$$
$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-1} R_T$$

Both value functions $V_\pi(s)$ and $Q_\pi(s)$ have a recursive property that can be used to derive the value for each state S_t. Specifically, return at time step *t* can be decomposed into an immediate reward R_{t+1} and a discounted reward received from the next time step until the end of episode which is just the discounted value of the next state S_{t+1}.

$$V_\pi(s) = E_\pi[R_{t+1} + \gamma V_\pi(S_{t+1}) | S_t = s]$$
$$Q_\pi(s, a) = E_\pi[R_{t+1} + \gamma Q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a]$$

These two equations are known as Bellman expectation equations for $V_\pi(s)$ and $Q_\pi(s)$ and can be used to find value functions of states of Markov decision processes using several techniques that we will discuss in next subchapter. By finding a value function for a particular state and a policy, we can also derive optimal value function that specifies best possible performance of an agent in a particular Markov decision process. Optimal state-value function $V^*(s)$ and action-value function $Q^*(s, a)$ is simply defined as a maximum value function over all policies.

$$V^*(s) = \max_\pi V_\pi(s)$$
$$Q^*(s, a) = \max_\pi Q_\pi(s, a)$$

If we find $Q^*(s, a)$, we can also determine an optimal policy for the given Markov decision process by selecting an action with the highest optimal action-value function.

$$a^* = \underset{a \in A}{\operatorname{argmax}} Q^*(s, a)$$

In the next subchapter, we will define basic RL methods that can be used to find optimal value functions and policies in Markov decision processes.

1.2 Basic reinforcement learning methods

There are several methods commonly used in RL for evaluating and finding optimal policies. Among these methods are dynamic programming, Monte-Carlo methods, TD-learning, SARSA and Q-learning.

Dynamic programming methods assume complete knowledge of the dynamics of a Markov decision process, that is to say, that we have to know state transition probabilities $P_{ss'}^{a}$ and reward transition probabilities R_s^a for every state in a Markov decision process. This assumption is unrealistic for most real-world problems due to the high number of possible states or even continuous states in real-world environments. Therefore, this method can only be used for solving smaller problems with manageable number of states and known dynamics. Considering that Deep RL aims to solve real world problems where dynamics of underlying Markov decision process in unknown, we will not consider methods based on dynamic programming. Instead we will move forward to algorithms that can achieve model-free prediction and control in environments with unknown underlying Markov decision processes.

The algorithm often used for model free prediction and control in RL problems is Monte-Carlo method. Using Monte-Carlo methods it is possible to approximate or learn a value function or an action-value function directly from episodes of experience without knowing the underlying dynamics of a given Markov decision process. First we will consider Monte-Carlo policy evaluation where the goal is to learn a value function $V_\pi(s)$. Monte-Carlo agent explores the state space of an underlying Markov decision process by sampling episodes of experience using a policy π, which results in the stream of state-action pairs and rewards $S_{1,}A_{1,}R_{2,}S_{2,}A_{2,}R_{3,},\dots,S_k$. Based on this stream of experience, the agent will approximate the value function $V_\pi(s)$, which is defined as an expected cumulative discounted reward given some state s, by calculating an empirical mean return. We can calculate an empirical mean return either by first-visit method or by every-visit method. First-visit method can be summarized as follows:

- every time some particular state s is visited by the agent for the first time during an episode, increment the counter C for the given state $C(s) \leftarrow C(s) + 1$, and
- increment the total return for that state $T(s) \leftarrow T(s) + G_t$,
- estimate the value of that state $V(s)$ by calculating the empirical mean return for that state $V(s) = T(s)/C(s)$,

- repeat previous three steps for every new episode,
- by the law of large numbers, when $C(s) \rightarrow \infty$, empirical mean return for every state converges to the value function for that state, given a policy $V(s) \rightarrow V_\pi(s)$

In every-visit method incremental mean is calculated every time step when that state is visited during an episode. This method also converges to true value function for a given state and a policy $V_\pi(s)$. This method can be augmented by using incremental Monte-Carlo updates so we can compute an empirical mean return incrementally without keeping track of an incremental total return $T(s)$. This algorithm is the same as in every-visit method, the difference being the value update in which the value of state S_t is updated in the direction of difference between a true return and an expected return scaled by the total number of visits of that particular state.

$$V(S_t) = V(S_t) + (G_t - V(S_t))/C(S_t)$$

Using incremental Monte-Carlo updates, we can not only evaluate a policy π, but also improve our policy, by incrementally updating a action-value function $Q(S_t, A_t)$ instead of the value function. Incremental Monte-Carlo policy updating can be summarized in the following way:

- every time some particular state S is visited and an action A is performed by the agent during the episode, increment counter C for the given state and action $C(S_t, A_t) \leftarrow C(S_t, A_t) + 1$, and
- estimate the action-value function $Q(S_t, A_t)$ by updating the action-value function in the direction of difference between a true return and an expected return scaled by the total number of visits of that particular state when action A_t was taken

$$Q(S_t, A_t) = Q(S_t, A_t) + (G_t - Q(S_t, A_t))/C(S_t, A_t)$$

- update our policy by acting ϵ-greedily with respect to updated action-value function $\pi \leftarrow \epsilon$-greedy(Q),
- repeat previous three steps for every new episode,
- by the law of large numbers when $C(S_t, A_t) \rightarrow \infty$, action-value function converges to an optimal action-value function $Q(S, A) \rightarrow Q^*(S, A)$

Although Monte-Carlo methods are in practice quite effective in solving RL problems, these methods depend on calculating true return G_t which means, that agent has to sample from a Markov decision process until it reaches the end of episode, which in

practice can dramatically slow down the convergence process. Relying on true return G_t also means that this method only works for episodic environments that terminate after finite amount of steps. In the next section we will focus on Temporal-Difference learning that addresses many of the shortcomings of Monte-Carlo methods.

Temporal-Difference (TD) learning [2] is one of the most popular algorithms in reinforcement learning. The main advantage of this algorithm over Monte-Carlo methods is that TD learning can learn online from incomplete sequences of experience. Thus it can potentially speed up learning, because it does not rely on true return G_t in value function update and can be used in continuing non-terminating enviroments. Instead of true return G_t, TD learning uses TD error updates δ_t that are computed incrementally every time some state S_t is visited.

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

We can see that this update fully exploits recursive property of the value function $V(S_t)$. TD update also replaces true return G_t with estimated return $R_{t+1} + \gamma V(S_{t+1})$ which makes incremental on-line learning possible. TD value function update rule can thus be expressed in the following way, where α is the learning rate.

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$$

This version of TD-learning is known as TD(0) and like Monte-Carlo methods, it converges to the true value function for a given policy $V_\pi(s)$. TD(0) approximates true value function by "looking" one step in the future unlike Monte-Carlo methods that has to sample complete trajectories of experiences from current time step until the end of episode in order to compute true return G_t. Besides one-step TD learning there is a class of algorithms that combine TD-learning and Monte-Carlo method. N-step return TD learning combines Monte-Carlo and TD-learning by calculating true return for n future step and then approximating true return from step n. This method uses the so-called *n*-step return $G_t^{(n)}$ instead of true return when updating value function.

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n})$$

We can see that if we select $n = 1$, our TD-target is equivalent to TD(0) target. On the other hand, if we select $n = \infty$, TD-target is equivalent to the true return G_t in Monte-Carlo update.

$$G_t^{(1)} = R_{t+1} + \gamma V(S_{t+1})$$

$$G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-1} R_T$$

By selecting n we can effectively combine Monte-Carlo method and TD-learning and in theory combine best of the both worlds depending on a RL problem we want to solve. Value function update for *n*-step return thus has the following form

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t^{(n)} - V(S_t))$$

Another variant of n-step return algorithm is TD(λ). This algorithm is based on the n-step return, one difference being that the TD(λ) algorithm averages all returns in N-step return $G_t^{(n)}$ by geometrically weighting the return $G_t^{(n)}$ by factor $\lambda \in (0,1)$. N-step return and value function update in TD(λ) have the following form

$$G_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$$

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t^{\lambda} - V(S_t))$$

We can see that by selecting λ = 1 we get an update that is equivalent to Monte-Carlo update and by selecting λ = 0 we get an update that is equivalent to TD(0) update.

Although by using TD(λ) algorithm we can combine Monte-Carlo learning with TD(0) algorithm, necessity of calculating n-step return $G_t^{(n)}$ by sampling real experience from underlying Markov decision process means that we lose on-line updating capability of TD(0) algorithm. By introducing the quantity called eligibility traces $E_t(s)$ into the TD(0) value update, we can effectively combine TD(0) algorithm with Monte-Carlo algorithm and use on-line updates as in TD(0). Eligibility trace is an accumulator that is incremented every time some state s is visited in the environment. In each visit the total accumulated eligibility trace for some state s is also discounted by factor γ and weighted by factor λ. Thus we can define the eligibility trace for state s at time step t in the following way

$$E_0(s) = 0$$

$$E_t(s) = \gamma\lambda E_{t-1}(s) + 1(S_t = s)$$

Another advantage of using eligibility traces is that by using them we can better address the credit assignment problem. Credit assignment problem describes a situation when a reward that is caused by agent's action or is associated with certain state is delivered by the environment with a high temporal delay. This can lead to a situation where reward that is associated with distant state or action only very weakly affects this state or action, which can have a negative impact on the speed of convergence. By incrementing an accumulator every time some state S_t is visited we take into account frequency heuristic that assigns credit to more frequently visited states. On the other hand, we also take into account

recency heuristic that assigns more credit to more recent states, by temporally discounting states by weight factor λ.

Value function update for TD(λ) algorithm with eligibility traces thus has the following form

$$V(S_t) \leftarrow V(S_t) + \alpha E_t(s)(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$$

Similarly to TD(λ) algorithm without eligibility traces, when λ = 0 TD(λ) with eligibility traces is equivalent to TD(0) update and when λ = 1, it is equivalent to every visit Monte-Carlo update.

So far we only discussed Temporal-Difference methods for value function updating. TD learning can also be used for improving policy of RL agents. Policy improvement technique based on a TD-update rule is called SARSA algorithm (State-Action-Reward-State-Action) [3]. The update rule for SARSA algorithm is identical to update rule used in TD-learning, but instead of the value function $V(S_t)$ we are updating action-value function $Q(S_t, A_t)$. The basic SARSA algorithm that uses updates similar to TD(0) updates, can be summarized in the following way:

- at the start of training, arbitrarily initialize Q values for every state-action pair,
- at each time step t in the episode select action A_t based on state S_t by acting ϵ-greedly with respect to action-value function Q,
- perform action A_t, observe reward from environment R_{t+1} and next state S_{t+1},
- select next action A_{t+1} based on state S_{t+1} by acting ϵ-greedly with respect to action-value function Q,
- update the action-value function in the direction of TD-error
  $Q(S_t A_t) \leftarrow Q(S_{t,} A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1} A_{t+1}) - Q(S_{t,} A_t))$,
- $S_t \leftarrow S_{t+1}$, $A_t \leftarrow A_{t+1}$.

As in Temporal-Difference learning, there are methods combining SARSA updates with Monte-Carlo updates known as n-step SARSA and SARSA(λ). N-step SARSA updates uses the same principle as n-step TD-updates but replaces n-step return $G_t^{(n)}$ with n-step Q-return $q_t^{(n)}$.

$$q_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n})$$

$$Q(S_t A_t) \leftarrow Q(S_{t,} A_t) + \alpha(q_t^{(n)} - Q(S_{t,} A_t))$$

In SARSA(λ) we replace geometrically the weighted return G_t^λ by geometrically weighted Q-return q_t^λ.

$$q_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}q_t^{(n)}$$

$$Q(S_tA_t) \leftarrow Q\big(S_{t,}A_t\big) + \alpha(q_t^\lambda - Q\big(S_{t,}A_t\big))$$

In SARSA(λ) with eligibility traces, $E(S,A)$ are calculated for each state and action pair as

$$E_t(s,a) = \gamma\lambda E_{t-1}(s,a) + 1(S_t = s, A_t = a)$$

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha E_t(s,a)(R_{t+1} + \gamma Q(S_{t+1},A_{t+1}) - Q(S_t,A_t))$$

So far we have only examined RL algorithms that learn from the same policy being used to generate actions in an environment. This is called on-policy learning and has several disadvantages. One of the main disadvantages is inability to learn from past experience generated from previous policies which is very important for Deep Q-Learning algorithm discussed in the next chapter. Other disadvantages include inability to learn the optimal policy while following some exploratory policy, or inability to learn by observing other agents in the same environment. We will now consider algorithms that deal with these disadvantages, namely algorithms that can perform off-policy learning.

Both TD-learning and Monte-Carlo methods can be adapted to off-policy learning by the method called Importance Sampling**.** We will not consider these methods, instead we will focus on one of the most popular RL algorithm that is the basis of Deep Q-learning, namely Q-learning [4]. Q-learning enables learning the action-value function $Q(S_t,A_t)$ off-policy by selecting action A_t using some behavior policy $\mu(\cdot\,|S_t)$ but using an alternative action A' generated by different (target) policy $\pi(\cdot\,|S_t)$ in the computation of TD-error. This setup makes it possible to learn both behavior and target policies simultaneously. This is advantageous in situations where behavior and target policy are different, for example when we use ϵ-greedy policy as behavior policy and greedy policy as target policy. We can define ϵ-greedy policy as a policy in which our agent performs random action with probability ϵ and greedy action selected by target policy with probability $1-\epsilon$. This policy is useful in situations where we know that our target policy is not yet close to an optimal policy and we want to make sure that our agent sufficiently explores the state-action space. Another situation where Q-learning is advantageous is the situation where we want to simultaneously learn behavior and target policies with different parameters for

example in Deep Q-learning. We can now define Q-learning update procedure assuming that we use greedy target policy $\pi(S_{t+1}) = \underset{a'}{\operatorname{argmax}}\, Q(S_{t+1}, a')$ and a ϵ-greedy behaviour policy $\mu(S_{t+1})$, where a' is a set of alternative actions.

$$\delta_t = R_{t+1} + \gamma Q\left(S_{t+1}, \underset{a'}{\operatorname{argmax}}\, Q(S_{t+1}, a')\right) - Q(S_t, A_t) =$$

$$= R_{t+1} + \max_{a'} \gamma\, Q(S_{t+1}, a') - Q(S_t, A_t)$$

$$Q(S_t A_t) \leftarrow Q\left(S_{t,} A_t\right) + \alpha\delta_t$$

In subchapter 1.3 we will discuss significance of reinforcement learning in animal and human cognition and evidence of RL mechanisms in the brain.

1.3 Reinforcement learning in the brain

Rewards are one of the most important concepts in both machine and human/animal decision making. It is possible that the main reason why brain evolved is that it allows much more effective acquisition of food, water and mating partners which can be represented as rewards. In other words, brains make animals and humans learn how to select, approach and consume the best rewards for survival and reproduction by identifying reward value of objects and situations that are needed to acquire these rewards [5].

There are two basic types of rewards, namely primary homeostatic and reproductive rewards and nonprimary rewards [5]. Primary rewards are basic rewards like food, liquids and activities necessary for mating and caring for offspring that are necessary for survival of individual and gene propagation. Nonprimary rewards can be defined as rewards that support and enhance primary rewards. These rewards often have no homeostatic or reproductive value by themselves but pursuing them may support acquisition of primary rewards in indirect ways. For example, money is not rewarding in itself but acquisition of money enhances the chance of acquiring better food or mating partners making money rewarding by association with primary rewards. Both types of rewards have three distinct components, namely sensory components, salience components and value component [5]. Sensory components such as color, smell, texture of rewarding objects or combination of these components enable identification of the rewarding objects or situations in the environment. Salience components include physical salience, novelty and surprise salience and motivational salience and their main function is to direct attention towards rewarding objects or situations. Value component determines the “goodness” of specific object or

situation and mediates behavioral reinforcing, approach generation and emotional effects of rewards. These components combined together ensure maximal reward acquisition.

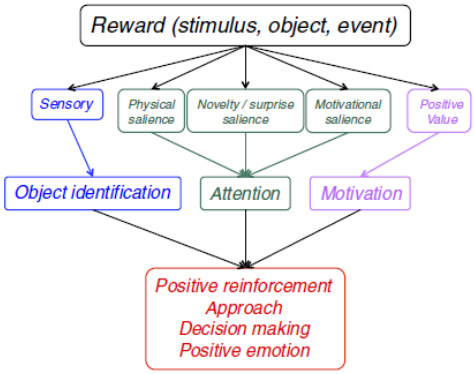

Figure 1. Basic reward components [5].

Successful acquisition of rewards requires not only their identification, but also prediction of their occurrence and behavior that leads to reward acquisition. There is a large body of evidence that neurons in ventral tagmental area (VTA) and substantia nigra (SN) are responsible for the processing of rewarding stimuli [6]. These mid-brain structures send their axons to structures responsible for goal-directed behavior, motivation and executive processes, for example frontal cortex, striatum and nucleus accumbens.

The evidence supporting the role of VTA and SN in reward processing comes from single cell recordings from dopamine neurons in these areas. In these experiments, an animal is presented with rewarding stimuli, called unconditioned stimuli, in the form of small quantities of sweet juice or other kinds of food, which results in short phasic activation of dopamine neurons in these areas. If we modify this experiment by presenting some cue in the form of visual or auditory stimuli without intrinsic rewarding properties before actual rewarding stimuli, dopamine neurons in these areas start to change their behavior to these stimuli. After multiple pairings of cues and rewards dopamine neurons change the time of their phasic activation from the time of actual reward delivery to the time of cue delivery as depicted in Figure 2. So previously unrewarding stimuli or cues become the predictors of reward called the conditioned stimuli. After this training when the rewarding stimuli is not presented after the conditioned stimuli, the activity of dopamine neurons is substantially decreased at the time of expected reward stimuli delivery. The activation of dopamine neurons also increases with the expected magnitude of reward [7]. In Figure 3 we can see that the activity of dopamine neurons at the onset of conditioned stimuli scales with an expectation of reward, defined as the probability of reward times the

amount of reward represented by sweet fluid given to subjects (Macaque monkey). This behavior of dopamine neurons in VTA and SN indicates that activity of dopamine neurons in these areas encodes the difference between the predicted reward and the time of its delivery and the actual reward and its time of delivery [6].

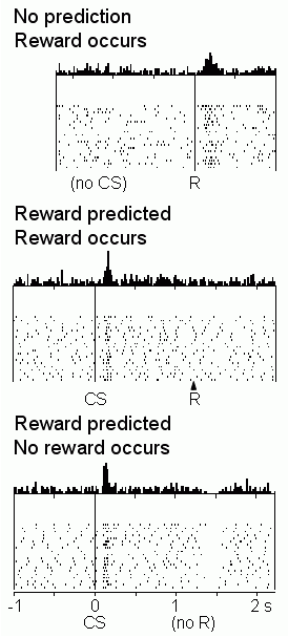

Figure 2. Dopamine neurons change the time of their phasic activation from the time of actual reward delivery (top) to the time of cue delivery (middle). If no reward occurs, phasic activation at the expected time of reward delivery is lowered (bottom) [6].

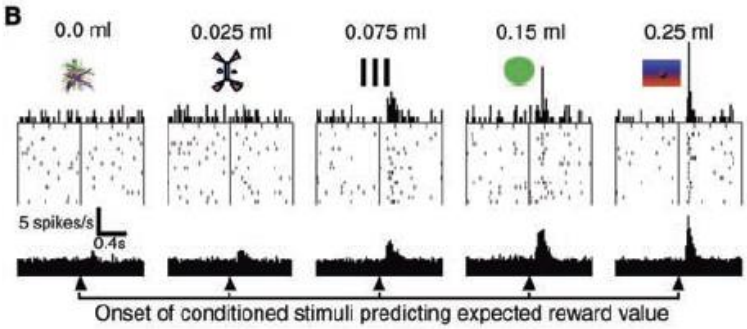

Figure 3. Activity of dopamine neurons at the onset of conditioned stimuli scales with expectation of reward. Volume of a sweet fluid administered to a Macaque monkey increases from left to right. Dopamine neuron responses also increase with higher dosages of sweet fluid [7].

Behavior of dopamine neurons in VTA and SN is consistent with Temporal-Difference RL algorithm [6]. If we look at the activity of dopamine neurons in VTA and SN through the prism of TD algorithm, we could say that the output of these neurons at the time of reward stimuli delivery represents TD-error updates δ_t and the activity at the time of cue

delivery represents the prediction of this cue, in other words, the discounted sum of rewards that animal would get by being in a particular state $V(S_t)$).

Besides the phasic dopamine response explained above, there exists another smaller dopamine response that precedes the reward prediction error signal. This response occurs before dopamine neurons have identified reward value of stimulus and appears to reflect physical, motivational and surprise salience and may reflect early assumptions about novel potentially rewarding events [5]. These initial and secondary dopamine responses are two basic reward components coded by dopamine neurons in VTA and SN, as depicted in Figure 4.

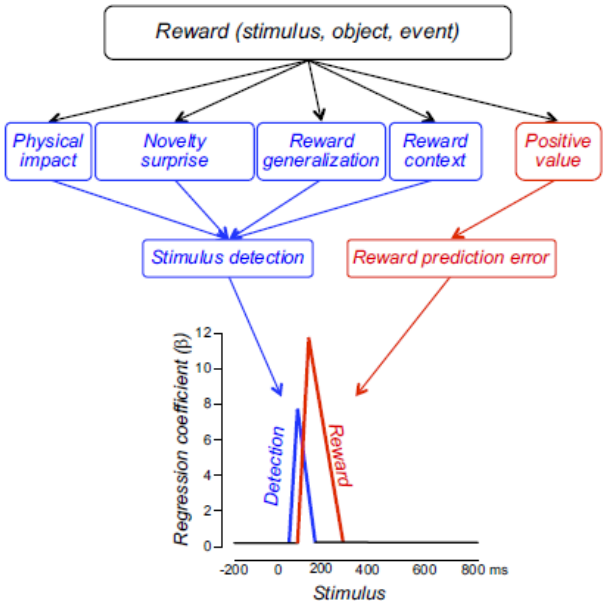

Figure 4.Stimulus detection and reward prediction components of overall dopamine response [5].

Phasic dopamine reward signal is only one of many functions of dopamine in the brain. We will not address these other functions in this work. We think that the fact that one of the functions of dopamine neurons, namely reward prediction error signaling, can be explained by RL theory, illustrates the significance of RL in the context of both animal and machine decision making.

In the next chapter, we will discuss RL methods that can be applied do broad spectra of real-world decision making problems, namely Deep Q-learning methods and their possible extensions.

2. Deep Q-learning

2.1 Value function approximation

In previous sections we discussed basic RL algorithms. We assumed that there exists some state-value function $V(s)$ or action-value function $Q(s,a)$ for every possible state or state-action pair in which case we can represent these value functions by a lookup table. If we want to solve complex real-world tasks using reinforcement learning, we have to address the fact that most real-world tasks have environments that have large amount of states or even infinite amount of states in the case of continuous environments. Storing these states in a lookup table and learning the value of each state or state-action pair is simply not practical. For example, in robotics we deal with continuous environments and continuous action spaces. Even classic board games with discrete state spaces have far more states than are practical or even possible to store in lookup table or evaluate. For example the game of Go with a large 19x19 board has 10^{170} possible states, which is more than there are atoms in the observable universe. The solution to this problem is to use a function approximator with some parameters w to approximate the value function $V_\pi(s)$ or the action-value function $Q_\pi(s,a)$.

$$v(s,w) \approx V_\pi(s)$$
$$q(s,a,w) \approx Q_\pi(s,a)$$

Using a function approximator allows us to learn the parameter vector w from observations of the environment and then generalize to unseen states of the environment.

There are many potential function approximation algorithms, for example neural networks, linear combination of features, support vector machines, nearest neighbor, genetic algorithms or decision trees. Considering the recent advancements in the neural network research and the state of the art performance of neural networks on challenging tasks such as image or voice recognition, we will only consider neural network based value function approximation methods in this work, although we think that using other function approximation methods is a viable direction for future research.

Neural networks are currently among the most popular machine learning methods. We think that neural networks are a good choice for approximating action-value functions due to the fact, that neural networks with at least one hidden layer are universal function approximators [8]. Also neural networks are currently the state-of-the-art models in many

challenging machine learning tasks such as large-scale image classification [9], large-scale video classification [10] and sentence classification [11]. We think that the state-of-the-art performance of neural networks on these tasks makes modern neural networks an attractive choice for value-function and action-value function approximators, particularly for RL tasks where the state of the environment is represented by a sequence of pictures or words.

We are not aware of single universally accepted definition of neural networks but all neural network models share some characteristics such as presence of interconnected basic computational units (neurons), which are connected to each other via a set of adaptable parameters called weights. Each computational unit in a neural network evaluates function of its input. The whole network of interconnected computational units then represents a composite function f, otherwise known as the network function, that transforms an input space X to an output space Y.

$$f : X \to Y$$

By observing examples of X, the neural network has to "learn" an optimal set of weights so that the network function f is able to correctly transform the input to the output even for examples of X that the network did not encounter before. The basic computation that every neuron in neural network represents is the sum of inputs x to that particular neuron weighted by weights w that corresponds to particular input. Usually there is also some nonlinearity σ, otherwise called the activation function, applied to this weighted sum. The output of particular neuron in the network y can thus be expressed the following way, where b denotes bias of a particular neuron:

$$y = \sigma\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

There are several types of activation functions, but among the most widely used activation functions are the sigmoid, hyperbolic tangent, softmax or rectified linear unit activation functions.

Learning in the context of neural networks is performed by a learning algorithm, which has to find some set of weights so that the output of a neural network is as close to the correct output as possible. More formally, the learning algorithm has to find a set of weights that minimize some measure of correctness of the output of the neural network called the cost function. The cost function is often task-specific, but for many classification-based tasks mean squared error or cross-entropy is used. Given a training set of p training examples and corresponding vectors of correct labels t and network outputs

o, we can define the mean squared error (MSE) cost function and the cross-entropy (CE) cost function in the following way:

$$MSE = \frac{1}{2p}\sum_{i=1}^{p}(o(i) - t(i))^2$$

$$CE = -\sum_{i=1}^{p}[\, t(i)\ln o(i) + (1 - t(i))\ln(1 - o(i))]$$

Several learning algorithms can be used for training neural networks, for example evolutionary algorithms, simulated annealing or expectation-maximization algorithms. In this work we will consider only the most commonly used algorithm used for training neural networks, namely error backpropagation (BP). The BP algorithm, discovered independently by Paul Werbos [12] and David Rumelhart [13], makes efficient training of neural networks with multiple hidden layers possible. The core principle of BP is to use the gradient descent optimization method for iteratively finding a combination of weight parameters that minimize the cost function, i.e. maximize the performance of the neural network. The BP algorithm can be divided into following steps:

- forward pass, in which the output of network is calculated from the training data,
- evaluation of the cost function E,
- calculation of the gradient of the cost function with respect to every weight parameter of neural network $\nabla E = (\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \dots, \frac{\partial E}{\partial w_l})$,
- each weight and a bias in neural network is updated by subtracting gradient scaled by learning constant α from the value of weight $w_l^{t+1} = w_l^t - \alpha \frac{\partial E}{\partial w_l^t}$.

There are several popular variants of the gradient descent algorithm that can be used in the conjunction with the BP algorithm for calculating gradients. Most of these algorithms are substantially faster than the standard gradient descent. Among the most widely used variants of the gradient descent algorithm are AdaGrad [14], AdaDelta [15], ADAM optimizer [16] or stochastic gradient descent.

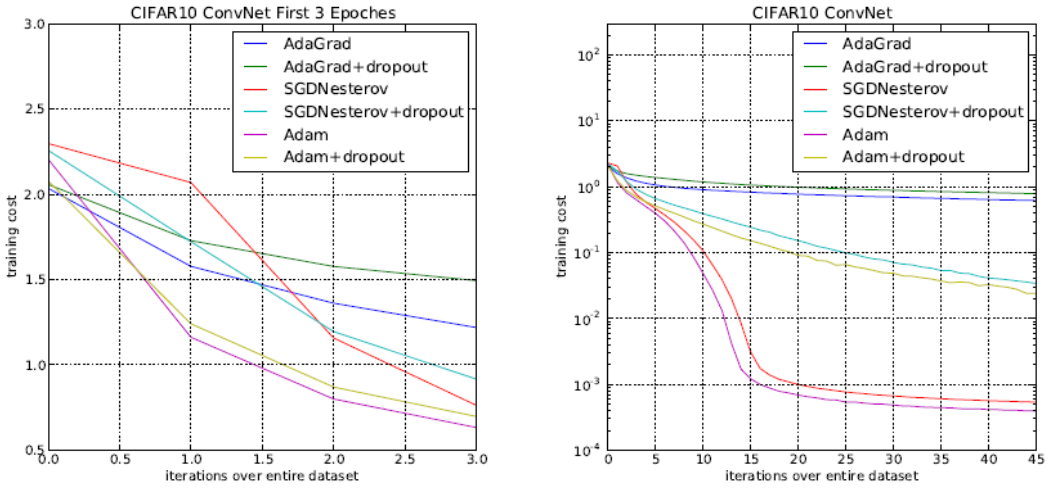

Figure 5. Comparison of various gradient descent optimizers on CIFAR10 datasets, left: first three epochs, right: 45 epochs. [16]

There are several neural network architectures that can be used for approximating the value and the action-value function, for example the multilayer perceptron, convolutional networks or recurrent neural networks. One of the most commonly used neural network architectures are the convolutional neural networks developed by Yann Le Cun for the use in automatic handwritten digit recognition system [17]. Architecture of the convolutional layers is inspired by the receptive field arrangement of neurons in the primary visual cortex which was first discovered in seminal work by Hubel and Wiesel on the topic of functional architecture of cat's visual cortex [18]. Each neuron in a convolutional layer is connected with a small region of the input. This small region can be seen as analogous to receptive field of the neurons in the primary visual cortex. When presented with an input image, the weight matrix of each neuron in a convolutional layer is convolved with overlapping regions of the input to form feature maps that contain the response of each neuron in that layer. This arrangement of neurons is able to tolerate various translations in the original input and can make better use of the local characteristics of the input. The max pooling layers are often inserted between two convolutional layers. These layers pool together the outputs of neighboring neurons by selecting only maximum activation of neurons in predefined locations in the feature map. Final processing in convolutional neural networks is usually done by multilayer perceptrons. Similarly to the multilayer perceptrons, convolutional networks are trained using standard BP algorithm. Architectures based on convolutional neural networks are currently the state-of-the-art models in the natural image

recognition with a performance rivaling that of the human labelers [19], which makes them a good candidate for the value function and the action-value function approximation in the cases where the state of the environment is represented by a picture or a sequence of pictures.

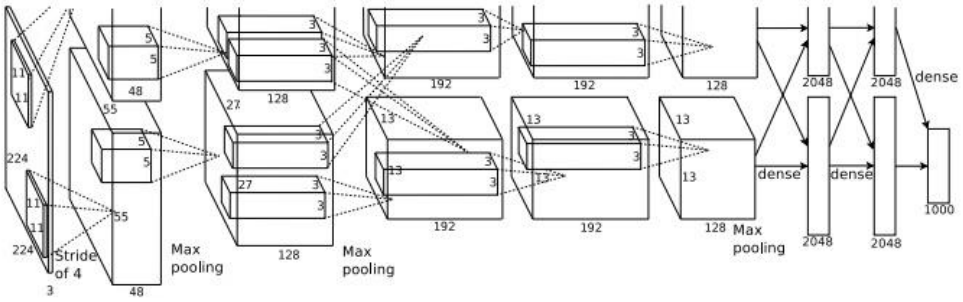

Figure 6. Example of a convolutional neural network architecture [20].

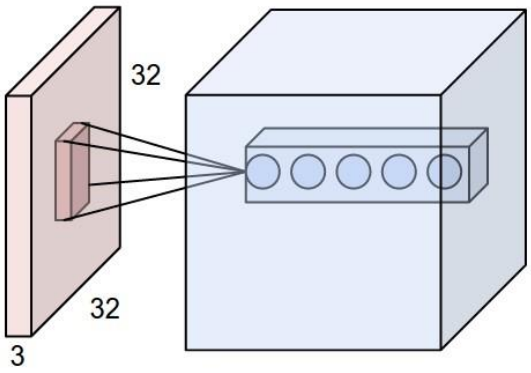

Figure 7. Example of convolutional layer architecture, each neuron (circles in the volume on right side) is connected to a small receptive field in input space (right) (from http://cs231n.github.io/convolutional-networks).

Another commonly used neural network architecture are recurrent neural networks. These networks can be used for modeling sequential data where inputs at the different time steps are dependent upon each other, which can be helpful in tasks such as time-series prediction, prediction of the next word in the sentence, machine translation, speech recognition or generating image description. The most common variant of recurrent neural networks used today is Long Short-Term Memory [21], which makes training recurrent neural networks that process extended time intervals possible.

In subchapter 2.2 we will discuss recent algorithm called Deep Q-learning [22], [23], which uses deep convolutional neural networks for approximation of action-value function. This algorithm was successfully used for playing a wide variety of simple computer games

directly from the picture representation of game states without any feature engineering or an access to the underlying game logic which is similar to the way humans learn to play games and perform actions in real-world environments.

2.2 Deep Q-learning for discrete action spaces

Deep Q-learning algorithm, developed by Mnih et al. (2013), aims to provide a practical way to apply RL principles in complex environments. This algorithm uses deep convolutional neural network Q with parameters θ as an action-value function approximator and uses Q-learning updates for updating the parameters of neural network. The authors of Deep Q-learning algorithm used the collection of 50 classic Atari 2600 games to test the performance of this algorithm. The goal of Deep Q-learning algorithm was to learn the optimal policy for each of these games directly from the visual representation of the game state using the same neural network architecture across all games.

Nonlinear function approximators are known to lead to instability when used in an action-value function approximation due to the correlations in the sequences of observations of the environment states, sensitivity of the policy to the small changes in Q-value updates and the correlations between action-values and target values [23]. Deep Q-learning addresses these instabilities using two key ideas. First, Deep Q-learning uses experience replay which stores transitions of states s_t, actions a_t, rewards r_t and states at the next time step s_{t+1} in replay memory D and at the time of learning samples random batches of transitions from this memory. This sampling strategy removes correlations in sequences of experience and improves the stability of the algorithm.

$$e_t = (s_t, a_t, r_t, s_{t+1})$$
$$D = \{e_1, e_{2,} \dots, e_t\}$$

Second, Deep Q-learning uses secondary deep convolutional network $\hat{Q}$ with parameters θ^- to generate Q-value target. This second network is updated periodically by copying parameters of the action-value function approximator. This results in quasi-stable Q-value target, which furthermore improves the stability of the algorithm.

The input to the action-value function approximation network Q is the representation of the state of the environment in the form of pictures generated by the computer game. Each state consists of four most recent frames from the game. This is because we want our

action-value function approximator to learn to react to a changing environment. For example, many classic computer games such as Pong, Breakout or Space Invaders are physics based games in which the agent has to infer the trajectory of objects in the game. Because a trajectory can only be approximated and predicted using more than one frame from the game, Deep Q-learning uses multiple frames from the game to approximate the action-value function. The output of action-value function approximator Q is a discrete set of valid actions for a particular game. Actions are selected according to ε-greedy strategy, with gradually decreasing value of ε. The figure below represents the convolutional neural network used as an action-value function approximator Q.

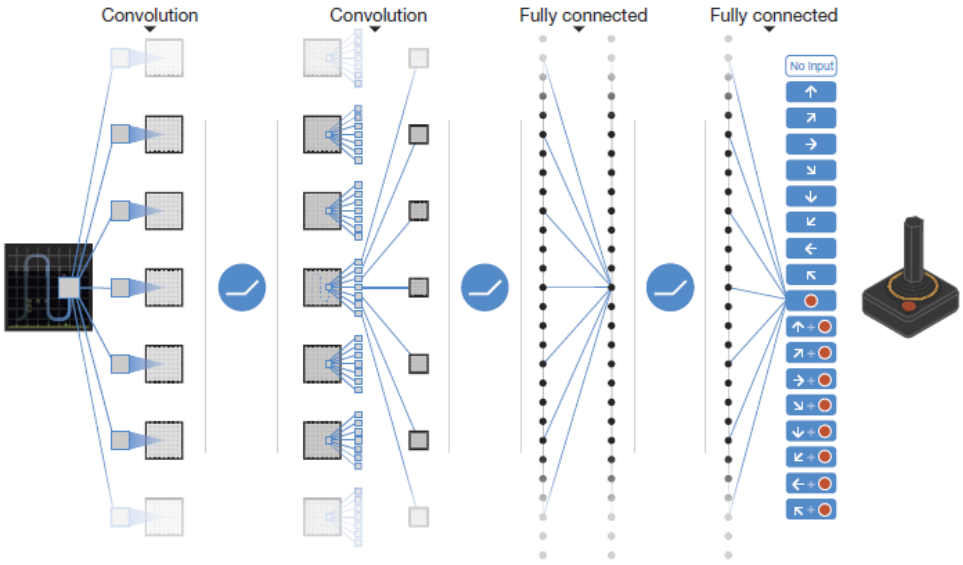

Figure 8. Neural network architecture used by Deep Q-learning (Mnih et al. 2015). Input to the network consists of 84x84x4 pixel array that contains 4 most recent game screens. The network consists of 3 convolutional layers, 2 fully connected layers and output layer with one output for each valid action [23].

We can now define Deep Q-learning algorithm [23]:

- Initialize replay memory D to capacity N,
- Initialize the action-value function approximator Q with random weights θ
- initialize target action-value function approximator $\hat{Q}$ by copying weights from Q, $\theta^{-} = \theta$
    - for episode = 1 to M do
        - get initial state s_1 from the game
        - for $t = 1$ until terminal state do

- with probability ε select random action a_t, otherwise select $a_t = \underset{a}{\text{argmax}}\, Q(s_t, a; \theta)$
- execute action a_t in the game and observe reward r_t and state s_{t+1}
- store transition (s_t, a_t, r_t, s_{t+1}) in replay memory D
- after every U actions do
    - sample random batch of transitions (s_j, a_j, r_j, s_{j+1}) from replay memory D
    - for every transition in batch set $Y_j = r_j$ if episode $j + 1$ is terminal episode, otherwise $Y_j = r_j + \gamma \max_{a'} \hat{Q}(s_j, a'; \theta^-)$
    - compute cost function $E = (Y_j - Q(s_t, a; \theta))^2$
    - update every parameter θ_l in Q, $\theta_l = \theta_l - \alpha \frac{\partial E}{\partial \theta_l}$
    - every C updates do $\theta^- = \theta$
- end for

▪ end for

Deep Q-learning was able to achieve a performance that rivaled and in many cases surpassed human performance in most Atari 2600 games it was tested on. It also surpassed the performance of other state-of-the-art RL algorithms in this domain. Comparison of the performance of Deep Q-learning algorithm on different games can be seen in Figure 9.

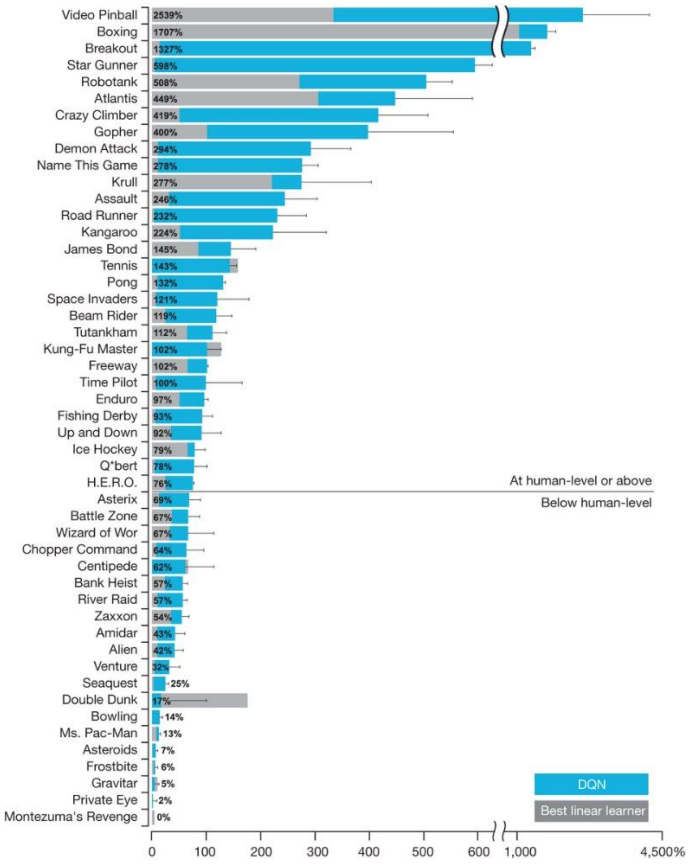

Figure 9. Comparison of performance of Deep Q-learning algorithm on various Atari 2600 games in percentage of human performance [23].

There are several variants of Deep Q-learning algorithm that try to address some of the shortcomings of the original algorithm. One of the variants is Double Deep Q-learning [24], which addresses the tendency to overestimate Q-values by original Deep Q-learning algorithm. According to Hasselt et al., this overestimation is due to the max operator in standard Deep Q-learning algorithm, which uses the same parameters both to select and evaluate an action [24]. To prevent this overestimation, Double Deep Q-learning algorithm decouples the action selection and the action evaluation by evaluating greedy policy using the policy network Q and estimating the value of this policy using target network $\hat{Q}$. Double Deep Q-learning algorithm is therefore the same as Deep Q-learning, with the difference being only in the computation of the target value Y_j.

$$Y_j^{DQN} = r_j + \gamma \max_{a'} \hat{Q}(s_j, a'; \theta^-)$$

$$Y_j^{DoubleDQN} = r_j + \gamma \hat{Q}(s_j, \operatorname*{argmax}_a Q(s_j, a; \theta); \theta^-)$$

Another variant is the Deep Q-learning with prioritized experience replay [25]. This algorithm is designed so that more "important" transitions from replay memory are

sampled more frequently as opposed to standard Deep Q-learning where transitions are sampled randomly. Combined with Double Deep Q-learning, this algorithm outperformed standard Deep Q-learning on 41 out of 49 Atari 2600 games [25].

In subchapter 2.3 we focus on Deep Q-learning-based algorithm for continuous action spaces.

2.3 Deep Q-learning for continuous action spaces

Deep Q-learning algorithm from the previous subchapter was demonstrated to be effective in learning powerful policies in many domains directly from high-dimensional pixel representations of an environment. However, it can only learn policies for discrete low-dimensional action spaces. Therefore it is not directly applicable for RL problems with continuous action spaces such as autonomous driving, robotic limb control or controlling a character in the first-person shooter game. Although it is theoretically possible to discretize continuous action spaces, this approach is not practical even for relatively low-dimensional action spaces. For example if we coarsely discretized action space of a humanoid robot such as the iCub with 53 degrees of freedom so that each degree of freedom would have only 2 possible values, we would need 2^{53} output neurons.

For RL problems with continuous action spaces we need to separately approximate the value of every action and state pair and the agent's policy. Reinforcement learning algorithm that is designed to cope with this kind of task is called the actor-critic. Actor-critic based algorithms generally contain two sets of function approximators. The actor selects an appropriate action based on the current state of the environment. The critic evaluates the value of the current state of the environment and an action selected by the actor. The parameters of the actor are then updated in the direction of policy gradient.

Recent work by Lillicrap et al. shows how to combine actor-critic approach with Deep Q-learning [26]. Using their algorithm called Deep Deterministic Policy Gradient it is possible to use deep neural networks as a function approximator in RL problems with continuous action spaces. Instead of one deep neural network that approximates the action-value function based on the state of the environment, actor-critic uses two networks, one as the actor network μ with parameters θ^{μ}, the second as the critic network Q with parameters θ^{Q} . Similarly to the Deep Q-learning, there are periodically updated target networks for both the actor network $\hat{Q}$ and the critic network $\hat{\mu}$ that stabilize the learning. Target

networks slowly track actor and critic networks, with parameter τ determining what fraction of the parameters of the actor and the critic networks is updated. The critic network parameters are updated using Q-learning update as in Deep Q-learning. The actor network is updated using Deterministic Policy Gradient method [27].

Deep Deterministic Policy Gradient algorithm can be defined following way [26].

- Initialize replay memory D to capacity N,
- initialize critic network Q with random weights θ^Q and actor network μ with random weights θ^μ,
- initialize target networks $\hat{Q}$ and $\hat{\mu}$ with weights $\theta^{Q^-} = \theta^Q$, $\theta^{\mu^-} = \theta^\mu$
    - for episode = 1 to M do
        - initialize random process ρ for action exploration
        - get initial state s_1 from the game
        - for $t = 1$ until terminal state do
            - select action $a_t = \mu(s_t; \theta^\mu) + \rho_t$
            - execute action a_t in the game and observe reward r_t and state s_{t+1}
            - store transition (s_t, a_t, r_t, s_{t+1}) in replay memory D
            - after every U actions do
                - sample random batch of transitions (s_j, a_j, r_j, s_{j+1}) from replay memory D
                - for every transition in batch set $Y_j = r_j$ if episode $j + 1$ is terminal episode, otherwise $Y_j = r_j + \gamma\, \hat{Q}(s_j, \hat{\mu}(s_{j+1}; \theta^{\mu^-}); \theta^{Q^-})$
                - evaluate the cost function $E = (Y_j - Q(s_j, a_j; \theta^Q))^2$
                - update every parameter θ_l^Q in Q, $\theta_l^Q = \theta_l^Q - \alpha \frac{\partial E}{\partial \theta_l^Q}$
                - update every weight parameter θ_l^μ in μ, $\theta_l^\mu = \theta_l^\mu - \nabla_a Q(s_j, \mu(s_j; \theta^\mu); \theta_l^Q) \nabla_{\theta^\mu} \mu(s_j; \theta^\mu)$
                - every C updates do $\theta^{Q^-} = \tau\theta^Q + (1-\tau)\theta^{Q^-}$, $\theta^{\mu^-} = \tau\theta^\mu + (1-\tau)\theta^{\mu^-}$
            - end for
        - end for

Deep Deterministic Policy Gradient method was tested on multiple RL tasks such as cart-pole swing-up task, reaching task, monopoded balancing tasks, two locomotion task and driving tasks in Torcs driving simulator. Both the low- dimensional environments (for example in the form of joint angles and positions) and the high-dimensional environments (pixel representation of the environment) were used in the training of this algorithm.

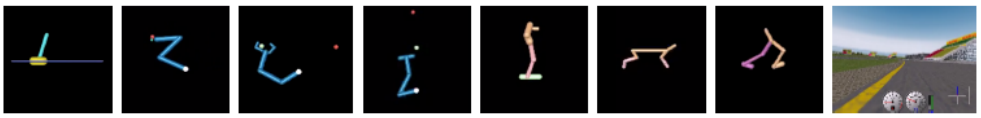

Figure 10. Examples of tasks used for testing of the Deep Deterministic Policy Gradient. From left to right: the cartpole swing-up task, a reaching task, a gasp and move task, a puck-hitting task, a monoped balancing task, two locomotion tasks and Torcs driving simulator [26].

For comparison, two baselines were used, the first baseline being uniform random policy and the second baseline being planning algorithm with the access to underlying physical model of the particular task. On every task Deep Deterministic Policy Gradient outperformed uniform random policy. On most tasks Deep Deterministic Policy Gradient managed to approach the performance of benchmark planning algorithm with access to underlying physical model, even surpassing it on some tasks.

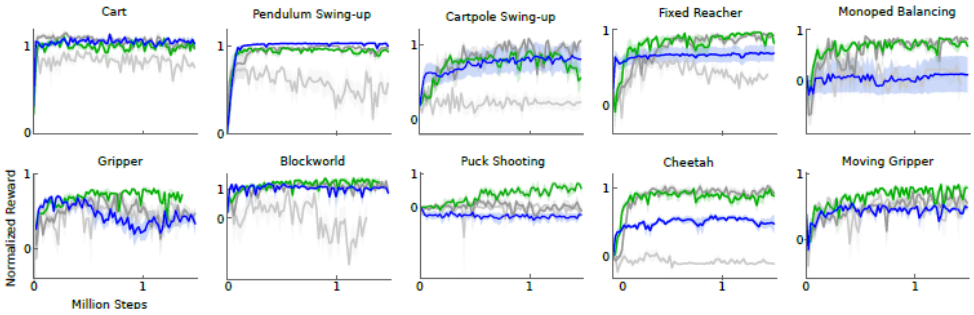

Figure 11. Performance curves for several RL problems using different variants of Deep Deterministic Policy Gradient during the training: Deep Deterministic Policy Gradient (light gray), Deep Deterministic Policy Gradient with target network (dark gray), Deep Deterministic Policy Gradient with target network and batch normalization (green), Deep Deterministic Policy Gradient with target network from pixel input (blue). Scores are normalized so that 0 is the performance of uniform random policy and 1 is the performance of benchmark planning algorithm [26].

In this subchapter and previous subchapter 2.2, we have discussed the Deep Q- learning algorithm, which was successfully applied to complex RL problems that require learning policies from high-dimensional inputs in both discrete and continuous action spaces. In

subchapter 2.4, we will discuss possible extensions of neural network based RL algorithms like Deep Q-learning so that these algorithms can “think” more like animals and humans.

2.4 Possible extensions of Deep Q-learning

We consider Deep Q-learning to be a big advancement in the field of RL and the artificial intelligence in general. Learning behavior policies directly from high-dimensional representation of the environment with only minimal prior knowledge is an important first step towards artificial agents that act and learn with human-like flexibility. Despite the human-like, or in some cases even superhuman, performance of Deep Q-learning on certain RL tasks, there are many differences between the performance of people and deep RL algorithms like Deep Q-learning. These differences are illustrated in Figure 12.

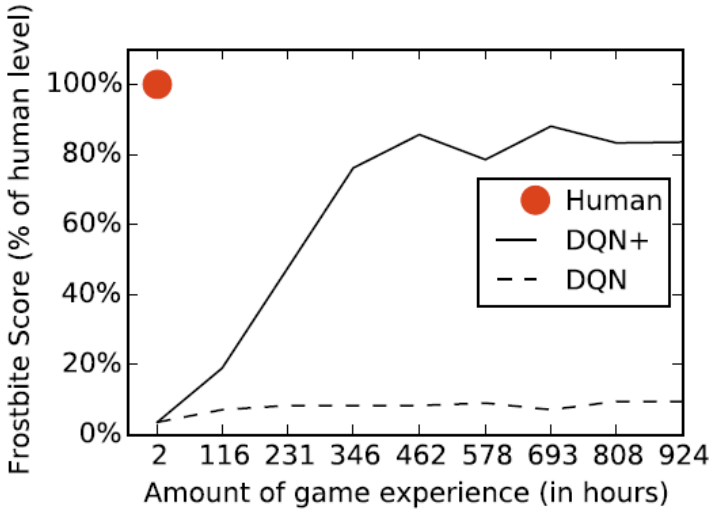

Figure 12. Relationship between the amount of training time and the performance on Atari 2600 game Frostbite. DQN+ denotes Deep Q-learning with prioritized experience replay [28].

We can see that although Deep Q-learning with prioritized experience replay can achieve nearly human-level performance in relatively complex Atari 2600 game called Frostbite, the amount of time it takes for the Deep Q-learning to approach human-level performance is much higher. An average person can learn the rules of the game in the order of minutes. Deep Q-learning needs more than 300 hours of game experience to reach its peak performance. The amount of time or training data needed for the Deep Q-learning to achieve peak performance significantly limits the flexibility of the agent. This is a common trait of all current neural network models, including the Deep Q-learning. Although this may not be a problem in situations where the algorithm has an access to the

large amount of training data, for example in simulated environments or in supervised learning problems like image recognition, it may be a problem in a real-world scenarios where the environment does not provide the agent with a large amount of experience. For example, it may not be practical or even economical to allocate hundreds of hours to teach a hypothetical household robot to perform a single simple task, such as washing dishes or cleaning the table, not to mention more complex tasks we perform every day that people can learn effortlessly. Another problem may be the generalization abilities of neural network based algorithms. People can perform the same task even if the details of the task change. For example, people could play the game of Pong even if the visual representation of background, ball or paddles changed, without learning the game anew. Neural network based algorithms could potentially do the same, but we would have to provide a large number of examples for each different task in order to make the algorithm generalize well across the same task with a different visual representation. Considering the enormous amount of different tasks a competent agent has to master in real-world environment, this approach would be highly impractical. Instead, we would need to endow our agent with some core competences that enable humans and animals to successfully perform tasks in a changing environment and rapidly learn new tasks when necessary.

Lake et al. identified some of the core cognitive competences or "ingredients" of the human intelligence that are necessary for artificial agents in order to act as competent agents in the environment [28]. Among these core competences are:

- developmental start-up software,
- learning by rapid building of the models of environment and
- fast thinking.

The term developmental start-up software refers to the intuitive understanding of several domains that is necessary for further cognitive development of a child. Among the most important parts of the developmental start-up software are intuitive physics and intuitive psychology.

Intuitive physics can be characterized as an ability to understand the basic properties and behavior of objects. Two-month old infants already have certain expectations about the objects in the environment, for example that objects move along the smooth path do not randomly appear and disappear out of existence, do not penetrate each other and do not act on distance [28]. Six-month old infants have basic understanding of physical properties of different objects and hold different expectations for solid, soft or liquid objects [28]. One-year old infants have developed basic understanding of concepts such as inertia, support,

containment and collision [28]. All of these basic abilities make predictions about physical objects possible. Deep learning system that possesses these capabilities would have to be able to approximate physical properties of objects. Based on these properties, it would have to be able to predict the behavior of these objects and behavior of systems that these objects are part of and generalize to objects that the system has never seen before based on the similarities between seen and unseen objects. In Figure 13 we provide an example of a task that such a system would have to be able to solve.

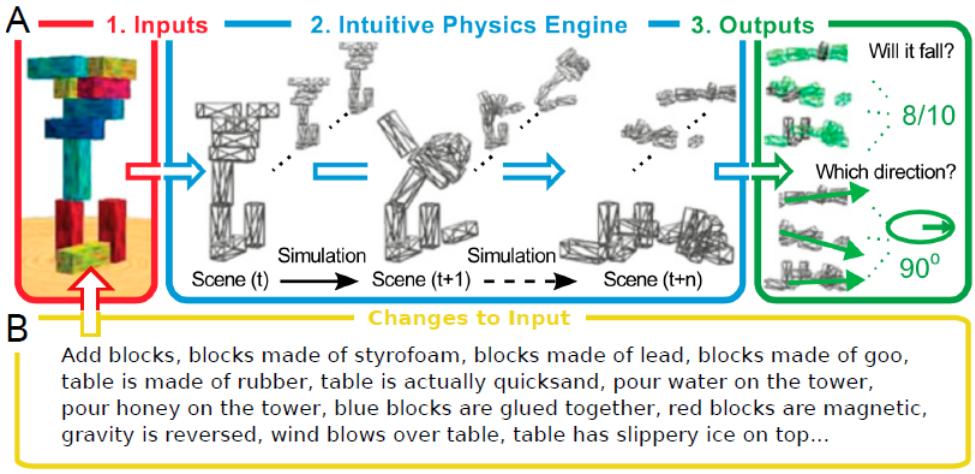

Figure 13. Example of a task that the deep learning system with an understanding of intuitive physics (intuitive physics engine) would have to solve. Based on the visual representation of the scene on the left, it would have to determine the physical properties of the objects in the structure, determine the stability of such a structure and predict what will be the final configuration of objects after the structure had collapsed [28].

The term intuitive psychology refers to the ability of infants to distinguish agents from inanimate objects, identify goals and general behavioral tendencies in other agents and the ability to distinguish between anti-social, neutral and pro-social agents [28]. Endowing deep learning system with this set of abilities is potentially an even more challenging task than endowing such an agent with intuitive physics. Similarly to intuitive physics, the artificial agent would have to learn to identify properties of other agents such as long- and short-term goals, habits, emotions, personality traits and relationship between the agent and other agents. Artificial agent would also have to predict behavior of other agents and generalize to agents it has never met before.

Learning by rapid building of the models of environment is a key competence of agents such as human beings that makes fast learning of new concepts possible. Unlike modern machine learning algorithms, children and adults are able to learn new concepts even from one example. This ability is called *one-shot learning* and so far has been impossible to

achieve in modern neural network based machine learning algorithms. People are also capable of generating new examples of novel categories from one example. Lake et al. identified three key competences that are necessary for rapid building of models of environment: compositionality, causality and learning to learn [28].

The term compositionality refers to the ability to learn or to construct a large number of representations from a finite set of primitives. Real-world objects and environments can be parsed into the set of primitives and spatiotemporal relations between these primitives. An artificial agent endowed with the notion of compositionality has to be able to parse an environment into primitives that can then be used to classify novel objects and situations into categories, only after one exposure to a novel object or situation. Such an agent also has to be able to produce novel examples of some category by combining primitives into novel instances of a particular object.

Causality refers to the ability to model processes that produce perceptual observations, in other words, to create generative models of an environment. This concept is also strongly related to the concept of compositionality in the sense that generative models can use primitives to construct generative models of the environment.

The term *learning to learn* refers to the ability to accelerate the learning of some concept or tasks by using priors (parameters) learned on a similar or a related concept or task. This term is also called the *transfer learning* or the multitask learning.

The last core competence that we will consider is *fast thinking*, which simply refers to the ability of the human mind to rapidly understand the state of the surrounding environment and to select appropriate actions to achieve some set of goals.

Recent advances in the field of deep learning made it possible for researchers to start to emulate some of the core competences we talked about. In the area of intuitive physics, Lerer et al. used deep convolutional neural networks to learn physical behavior of both virtual and simulated objects [29]. They managed to train an artificial neural network to both predict the stability of stacked boxes (if they are likely to fall or not) and to visualize the final position of boxes after they fell. Prediction performance was comparable to human participants in the case of physical boxes, and higher than human participants in the case of virtual boxes. The trained network was also able to generalize to the number of boxes it had not been trained on. As with other neural network approaches, the number of training examples needed to achieve a high level of accuracy was in the thousands, and the model was not able to generalize well to the situations where the number of boxes deviated significantly from the number of boxes in training examples.

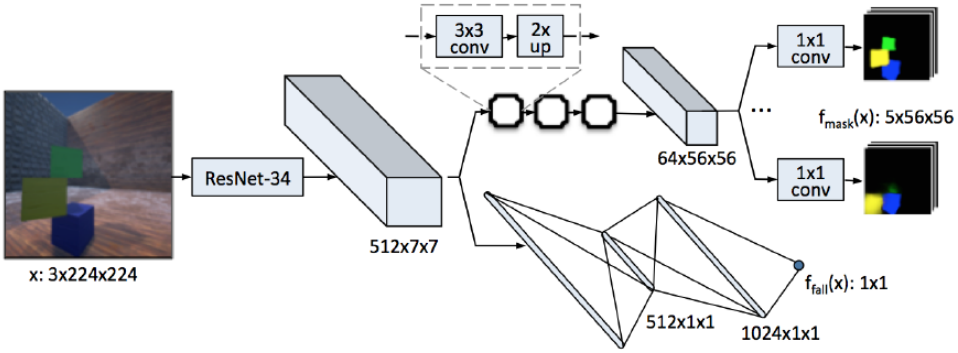

Figure 14. Architecture of the deep learning model used in Lerer et al. The input to the neural network model was a picture of boxes configuration (left). The network was trained to simultaneously output the probability of the box structure falling, the mask with the position of the boxes and the position of the boxes after the structure had collapsed (right) [29].

The task of learning compositionality can be subdivided into detecting object candidates in the environment and detecting essential part of objects in the environment. Both of these tasks can be tackled by deep learning approaches. Pinheiro et al. used a deep neural network model to detect objects in the natural scenes and produce segmentation mask for potential objects in the scene [30]. Their approach achieved the state-of-the-art results in the image segmentation benchmarks PASCAL VOC and COCO. Their approach also achieved impressive generalization capabilities where the network that was trained only on the subset of all object categories was able to segment images almost as well as the network that was trained on all categories.

Figure 15. Segmentation proposals from Pinheiro et al. Incorrect segmentation proposals are marked with red outline [30].

Another approach to generating a segmentation mask for potential objects in the scene by Chen et al. used a deep convolutional neural network and conditional random fields [31]. They managed to achieve the segmentation accuracy on real-world scenes similar to Pinheiro et al..

Tsogkas et al. used a similar approach to Chen at al. (deep convolutional neural networks and conditional random fields) for detecting parts of objects (semantic part segmentation) [32]. Although they managed to achieve high object part segmentation accuracy, they trained their model separately for each object category. A more general approach to semantic part segmentation would be to train a single deep learning model to segment previously unseen objects into parts. Such a model could make the categorization of new objects faster by reducing the dimensionality of input into object classifier.

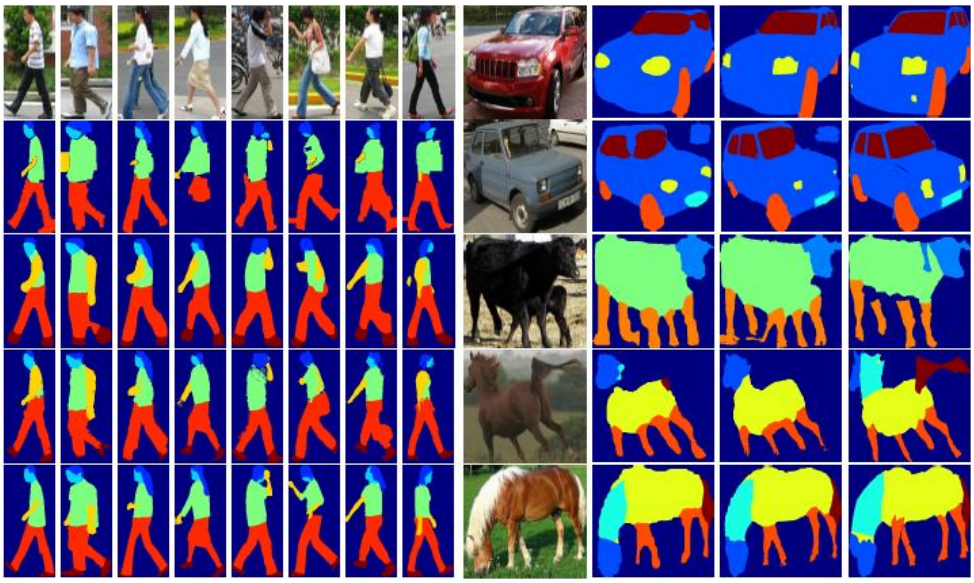

Figure 16. Left: Object segmentation results on pedestrian images dataset. From top to bottom: original images, results from benchmark model, convolutional neural network score, convolutional neural network with conditional random fields score, ground truth. Right: Object segmentation results from PASCAL-Parts dataset for different objects and animals. From left to right: original image, convolutional neural network score, convolutional neural network with conditional random fields score, ground truth [32].

Deep learning models that are able to learn generative models of environment are currently among the most researched topics in the field of deep learning. One example is the Deep Recurrent Attentive Writer [33]. This model uses a deep recurrent neural network with an attention mechanism to generate novel instances of various objects in pictures for example handwritten digits, street house numbers or natural objects.

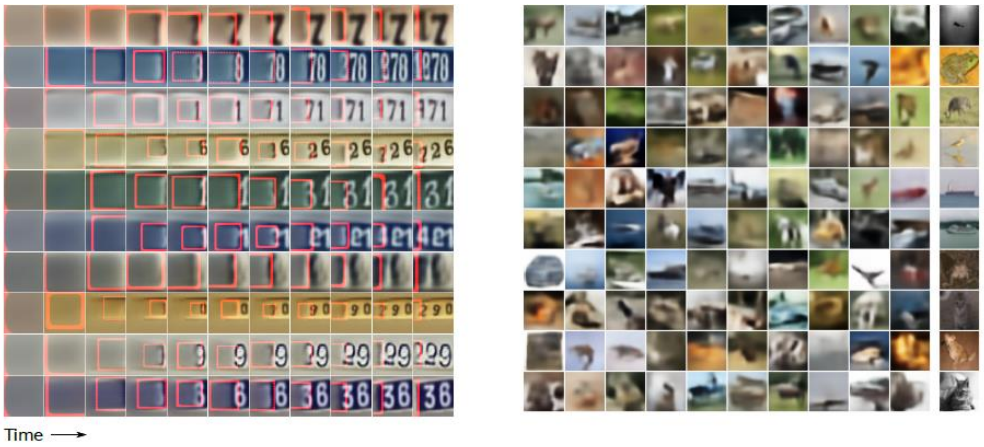

Figure 17. Left: The process of generation of street house numbers in which the red rectangle represents the attention window where the network is currently generating the image. Right: Examples of generated images from the CIFAR dataset. The rightmost column represents the nearest example from the dataset to the column beside it [33].

Another example of a deep generative model is Deep Convolutional Inverse Graphics Network [34]. This model uses convolutional encoder to learn the series of latent variables, such as pose, light or shape, that represent graphics code for images it was trained on. Convolutional decoder is then used to generate pictures of objects with some property changed. For example, incrementally changing one latent variable for input picture, for example orientation, results in the network generating pictures that depict the object in the picture from different viewpoints.

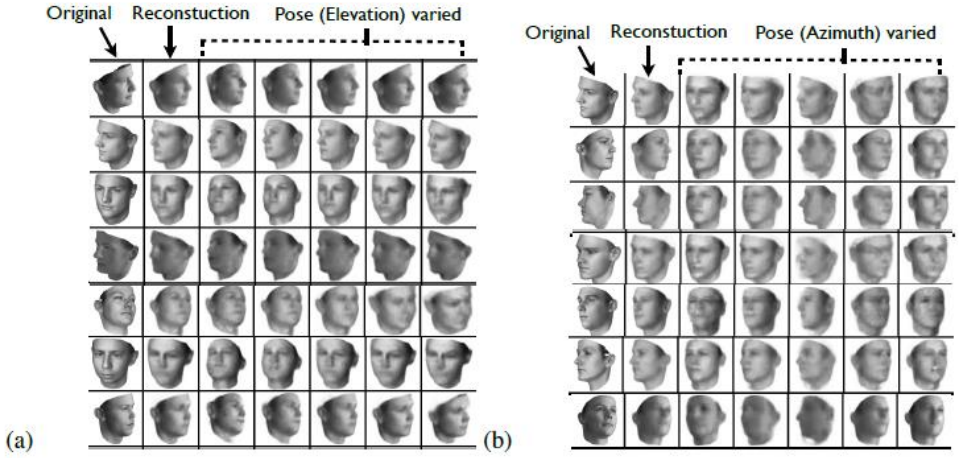

Figure 18. Reconstructed images that result from varying latent variable corresponding to elevation (left) and azimuth (right) [34].

Another interesting work by Oh et al. used an encoder-decoder architecture similar to Deep Convolutional Inverse Graphics Network to predict future frames in Atari 2600 games [35]. Subsequently they tested the quality of predictions of their model by using predicted frames as an input to Deep Q-learning network from Mnih et al. and compared the performance of this network when the true frames generated by simulator were used as an input versus when the predicted frames were used as an input. Although the performance of Deep Q-learning controller was degraded by using predicted frames, in many games the performance was still well above the random score and even approaching performance of the network with true frames from the simulator when the timescale of prediction was not too long. Despite the fact that this model can successfully predict the dynamics of subset of Atari 2600 games, it is limited when it comes to generalization, due to the necessary retraining for every new game.

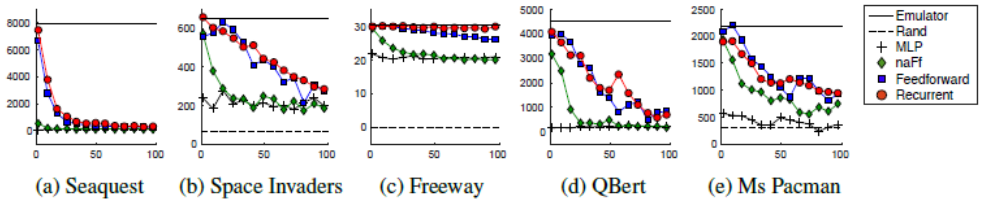

Figure 19. Performance of Deep Q-learning algorithm on 5 different Atari 2600 games. Emulator refers to the performance of Deep Q-learning network with true frames, Rand corresponds to the performance of random policy, MLP corresponds to multilayer perceptron predictor model with actions taken and last frame as an input, naFt corresponds to MLP with only the last frame as an input, Feedforward corresponds to encoder-decoder frame predictor architecture and Recurrent corresponds to recurrent encoder-decoder frame predictor architecture. The X axis denotes the number of time steps of prediction (timescale) and Y axis denotes the average score from 30 episodes of play [35].

Learning to learn or transfer learning can be achieved in multiple ways in deep learning models. One technique reuses weights from an already trained neural network model and then retrains this model on a similar task leaving most of the parameters fixed during training. The reasoning behind this technique is that parameters of neural network models closer to the input encode general features that can be applied on similar tasks. For example, a convolutional neural network model trained to recognize pictures of cars, can be used to recognize pictures of cats with retraining only the last fully connected layer of the network. This is because the parameters in the layers closer to the input may recognize general features useful for recognizing any image for example edges or textures. Another technique of transfer learning is similar to the previous technique, but instead of leaving

parameters closer to the input fixed, we would fine-tune the parameters trained on a similar task.

Parisotto et al. recently developed a Deep Q-learning approach that incorporates the learning-to-learn technique called Actor-Mimic [36]. Actor-Mimic uses two sets of Deep Q-learning networks. One is the so-called expert network that is trained only on one particular task (one Atari 2600 game). The other is the so-called multitask policy network that is used to learn multiple policies for a different task. This is achieved by two objective methods, the policy regression method and the feature regression method. Policy regression objective method uses cross-entropy cost between the outputs of the expert network and multitask policy network to learn the policy of the expert network. Feature regression objective method uses a separate regression network that predicts the activations of particular parameter layer (features) in the expert network from the activations of equivalent parameters in multitask policy network. Mean square error between these activations is then back propagated through the regression network and multitask policy network. This forces the parameters in the multitask network to incorporate information from the expert network beneficial for the particular task. The final objective that is used for training of multitask network is simply the combination of the feature regression objective and the policy regression objective. Intuitively we can think of the policy regression objective as a teacher who teaches multitask network the best way to act (policy), while we can think of the feature regression objective as a teacher who teaches the multitask network why it should act in a particular way. The multitask network was able to achieve performance similar to the expert network on many Atari 2600 games while having the same amount of parameters as the expert network. When the parameters of the multitask network were used as a starting point to learn a previously unseen task, in many tasks it managed to significantly reduce the training time and at the same time it managed to achieve the same score as the expert network. This was especially true for tasks that were conceptually similar, for example in tasks requiring the control of paddles that are used for interaction with a virtual ball (Breakout and Video Pinball).

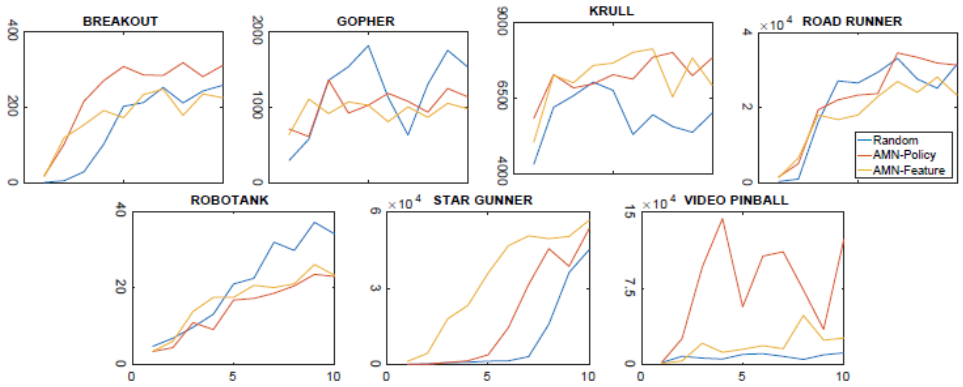

Figure 20. Comparison between the performance of Actor-Mimic algorithm and standard Deep Q-learning algorithm. The X axis corresponds to the number of training frames in millions. The Y axis corresponds to game scores. Random denotes the performance of randomly initialized network, AMN-Policy denotes the performance of network initialized with parameters from the multitask network with policy regression objective, AMN-Feature denotes the performance of a network initialized with parameters from the multitask network with policy regression objective and feature regression objective [36].

We think that all deep learning techniques mentioned above can potentially be used for enhancing the performance of a Deep Q-learning and other deep learning based RL algorithms. This can theoretically be done in several ways.

First, we could use methods such as those in [30], [31] and [32] to segment the representation of the environment into important elements and thus reduce the dimensionality of the representation of the environment. For example, in the domain of Atari 2600 games we could train a deep learning model to segment the game screens to elements that are common across different games such as background, enemies, projectiles and other common elements of the environment. In the domain of self-driving cars we could segment the video input into important elements such as pedestrians, roads, other vehicles, obstacles or road signs. Preprocessed segmented representation of the environment can subsequently be used as an input to a deep RL system such as Deep Q-learning. We suspect that by reducing the dimensionality of the representation of the environment we could increase the performance and decrease the training time of a deep RL system because such a system would not have to relearn important elements of the environment for every new task.

Second, we could use generative and predictive models such as in [29], [33], [34] and [35] to learn to predict the behavior of the important elements of the environment. Model predictions could be used to generate artificial training data in cases where the environment does not provide the agent with enough training data due to the relative rarity

of important events such as collisions in driving environment or other rare events that could potentially damage or benefit the agent or the environment.

Third, we could use transfer learning techniques such as the Actor-Mimic to speed up the training of deep reinforcement learning, generative and segmentation models by leveraging model parameters acquired while training on similar tasks.

How closely can we emulate the flexibility of human and animal minds using deep learning techniques mentioned in this subchapter is an open scientific question that we plan to explore in future work. Although it is possible, that the current deep learning models are too data inefficient to be an effective approach to emulating key components of human cognition, we think that by improving upon these techniques by incorporating methods mentioned in this subchapter, deep RL methods can become a practical solution for some tasks such as computer controlled driving, controlling in-game characters or training artificial agents to perform repetitive manual tasks in real-world environments.

In Chapter 3, we will evaluate the performance of a Deep Q-learning agent on a small subset of RL tasks.

3. Computational experiments in deep reinforcement learning

We have three main objectives that we want to achieve in this chapter. First, we want to replicate the results from [22] on a small subset of Atari 2600 games. Second, we want to test whether a more sophisticated learning algorithm (in comparison to learning algorithm used in [22] and [23]) can improve the speed of convergence of the Deep Q-learning algorithm. Third, we want to test whether simple transfer learning techniques mentioned in Chapter 2 can be used to improve the speed of convergence of this algorithm. Because of the high computational demands of Deep Q-learning algorithm and a limited time, we only performed 7 experiments described further in this chapter.

We have used our own implementation of Deep Q-learning algorithm as described in [22] and [23]. Experiments were performed using the Python programming language and the Tensor Flow machine learning library on a Linux-based computer with an Intel 4770k CPU, 16 GB of RAM and a NVidia GTX 980Ti GPU. All neural network computations were performed on NVidia GPU which substantially reduced the runtime of experiments. Game interface was provided by the Python library called the Arcade Learning Environment [37].

We tried to keep all parameters of the experiment as close as possible to parameters from [22]. Below we list the key parameters that were kept constant across all experiments.

- Network architecture,
- number of neurons in the model,
- learning rate = 0.00025,
- replay memory size measured in game steps (transitions) = 1 000 000,
- number of game transitions used as an input to the network = 4,
- frame size in pixels = 84x84x1
- action repeat value = 4 (number of frames where previously selected action is repeated; these frames are not stored in replay memory or used as an input to the neural network)
- update frequency of the action-value function approximation network Q measured in the number of actions selected by the algorithm from the last update = 4
- C = 10 000 (update frequency of the target network $\hat{Q}$ measured in the number of updates to the action-value function approximation network Q since the last update)

- number of updates to the action-value function approximation network Q after which the performance of the Deep Q-learning algorithm is tested = 50 000
- number of test transitions used for calculating average score per game = 10 000
- ε = annealed from 1 at the start to 0,1 during the first 1 000 000 transitions,
- γ = 0.99,
- size of the mini batch = 32,
- initial number of random transitions used to populate replay memory = 50 000,
- total number of game steps (transitions) used in training = 8 000 000.

The network architecture is identical to the architecture used in [22] and [23]. In Figure 21 we provide a detailed specification of the network architecture. Both the action-value function approximation network Q and the target network $\hat{Q}$ share the same architecture.

Layer name	**Type**	**Number of neurons**	**Kernel size**	**Stride**	**Activation function**
h_conv1	Convolutional	32	8x8	4	ReLU
h_conv2	Convolutional	64	4x4	2	ReLU
h_conv3	Convolutional	64	3x3	1	ReLU
h_fc1	Fully connected	512	x	x	ReLU
h_fc2	Fully connected	Depends on the number of actions	x	x	Linear

Figure 21. Deep Q-network architecture specification.

In Figure 22 we provide parameter descriptions and the key results for each experiment. The third column refers to the learning algorithm, i.e. the variant of the gradient descent algorithm used for training. Both Deep Q-learning algorithms from [22] and [23] use the RMSprop algorithm. We tried to improve upon results from [22] and [23] by utilizing the ADAM learning algorithm that was shown to converge faster and achieve higher accuracy than RSMprop and other gradient based algorithms for training neural networks in a wide variety of classification tasks [16]. The fourth column refers to weight initialization method we used at the onset of training. RAND refers to the initialization technique used in [22] and [23] which initialized weights as a random number with a mean 0 and standard deviation 0.01 and biases as a constant with a value 0.1. XAVIER refers to the weight initialization by Glorot et al. (2010) which was shown to provide more accurate starting

point that can improve the speed of convergence of neural network models in many real world tasks [38]. This weight initialization technique initializes all weights as a random number with a mean 0 and a standard deviation $\sqrt{fanIn}$ where $fanIn$ refers to the number of inputs to the neuron that is being initialized. Biases are initialized as a constant with a value 0.1. PONG-1 and PONG-2 refer to initializations where all layers except the output layer are initialized by copying weights from the network, previously trained on a conceptually similar task. More specifically, we wanted to test our hypothesis that weights trained on a conceptually similar tasks (Pong) could be a better starting point for neural network learning algorithm and could improve the speed of convergence. Both Pong and Breakout are similar in that the player controls paddles that are used for the control of the virtual ball. However, these games differ in the goals. In Pong the player has to prevent the ball from going outside the game screen on his side and tries to strike the ball outside of the screen at the opponent's side. In Breakout the player tries to break as many tiles at the top of the screen as possible while trying to prevent the ball from striking the ground. In PONG-1 we copied the weights from the network trained on the game of Pong to the network that was supposed to learn the game of Breakout and fixed the parameters during learning, letting only the last fully connected layer learn. In PONG-2 we did the same but permitted the learning of the rest of parameters during training. The best scores for human players are taken from [23] and reflect the average score calculated from 20 games with a maximum length of 5 minutes per game. Human testers were also allowed to train on every game for 2 hours and were not permitted to pause the game during the evaluation. We chose the best scores from [22] instead of [23] as our baseline for performance comparison because in the former the algorithm was trained on 10 million frames instead of 50 million frames, which is closer to 8 million frames used in our Deep Q-learning implementation.

Experiment number	Game	Learning algorithm	Weight initialization	Best score (human)	Best score DQN from [22]	Best score (our implementation)	Runtime (in hours)
1	Breakout	RMSprop	RAND	31.8	168	150	20.42
2	Breakout	ADAM	XAVIER	31.8	168	245.5	20.7
3	Space Invaders	ADAM	XAVIER	1652	581	778.57	19.89
4	Pong	ADAM	XAVIER	9.3	20	24	21.09
5	Breakout	ADAM	PONG-1	31.8	168	11.6	17.43
6	Breakout	ADAM	PONG-2	31.8	168	59.88	20.57
7	Breakout	ADAM	XAVIER	31.8	168	11.4	20.53

Figure 22. Key parameters and results for the experiments we performed.

From Figure 22 we can see that our implementation of Deep Q-learning, which is identical to the original implementation from [22] (Experiment 1), achieves similar performance as the original implementation. When we substitute the learning algorithm and the initialization technique from the original implementation for XAVIER initialization and ADAM learning algorithm (Experiment 2) we can see that the best score improves substantially without a significant increase in runtime. As shown in Figure 23, the usage of ADAM learning algorithm and XAVIER initialization also significantly improves the speed of convergence. Both Experiment 1 and Experiment 2 show super-human performance of Deep Q-learning in the Breakout game.

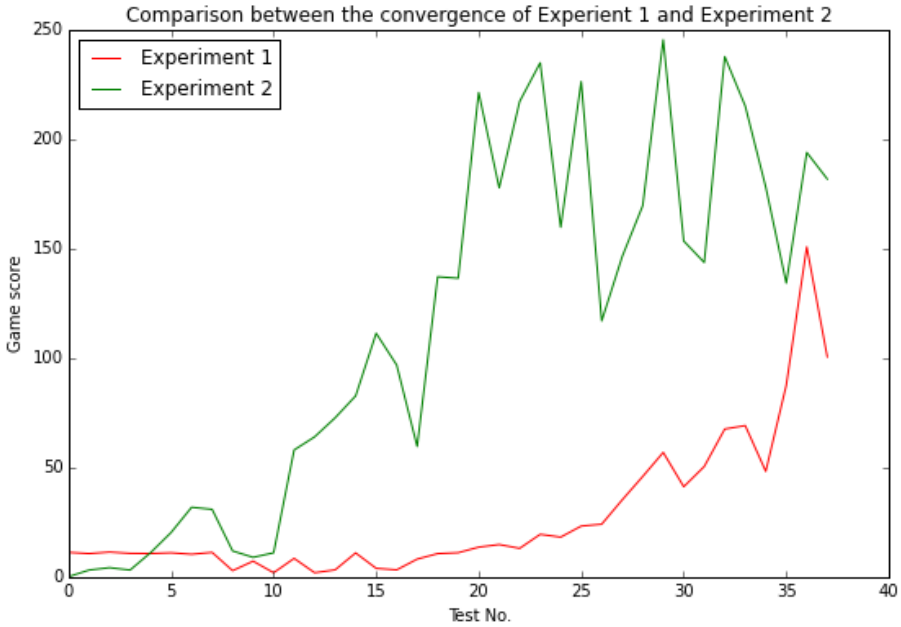

Figure 23. Comparison of the speeds of convergence in Experiment 1 and Experiment 2.

We can also see that the combination of ADAM and XAVIER algorithms improves the results for the games Space Invaders and Pong. In Pong game our implementation achieved super-human performance. In Space Invaders our implementation failed to achieve human-level performance although we think that with an additional training time it would also achieve human-level performance as shown in [23]. We can see the convergence graphs for these two games below in Figures 24. and 25..

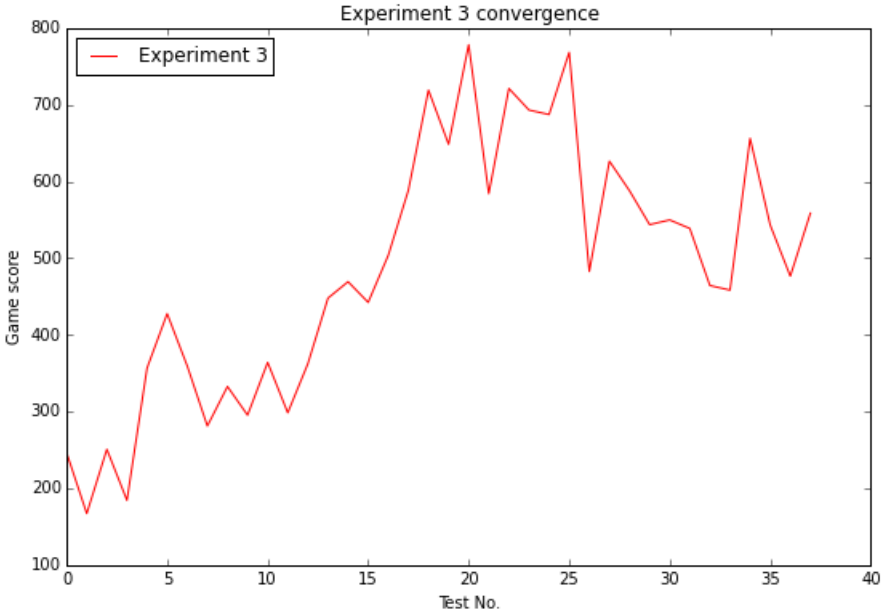

Figure 24. Speed of convergence in Experiment 3.

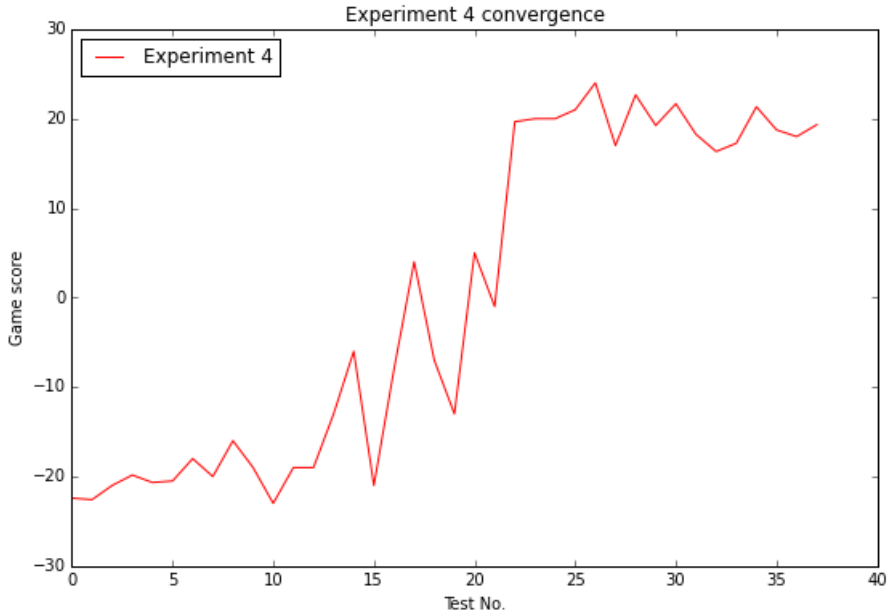

Figure 25. Speed of convergence in Experiment 4.

From the convergence graphs for Experiments 1 and 2 we can see that although the convergence speed is much higher in Experiment 2, the convergence is also quite unstable which can be seen as “dips” in performance in the right side of the graph. On the other hand, Experiment 1 converges much slower but is more stable than Experiment 2. If we trained our algorithm longer, this could result in better performance of the original algorithm. We plan to test this hypothesis in the future work. During our experiments we also discovered a curious case of instability that resulted in a poor performance of our implementation. This can be illustrated in Experiment 7 which is identical to Experiment 2. Despite the identical experimental setup, Experiment 7 did not converge at all. We do not know the exact cause or the frequency of occurrence of this “glitch” due to the small number of experiments we performed, but we think that this phenomena is worth further investigation.

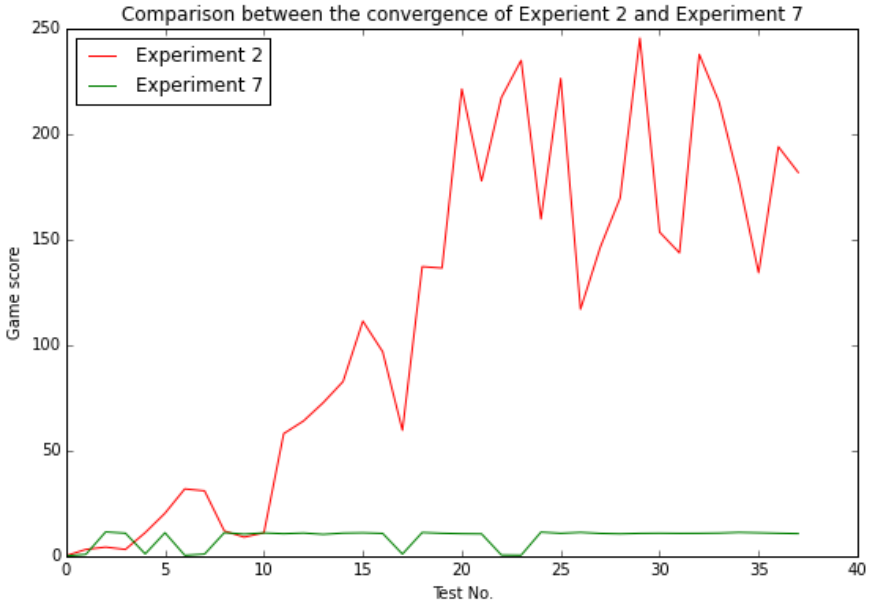

Figure 26. Comparison of the speeds of convergence in Experiment 2 and Experiment 7.

In Experiments 5 and 6 we tried to test the basic transfer learning techniques that rely on initializing weights by copying weights from an already trained network. As we can see from the Figure 27, neither experiment exhibited good convergence properties. Experiment 5 did not show any signs of convergence during learning. We think that this is because the simple retraining of the output layer could not utilize the features learned on a similar task. The features learned on a similar task are probably not general enough, i.e. these features are fine-tuned to the specific task they were originally trained on. In conclusion, the observation from the task of image recognition that it is often possible to retrain only the output layer in order for the neural network to learn to recognize novel objects does not apply to the control task performed by the Deep Q-learning algorithm. Experiment 6 where we trained all layers instead of only the output layer, showed some signs of convergence, but overall the initialization technique only slowed down the convergence, instead of speeding it up. We think that during training the network first had to "unlearn" the features from a similar task, which slowed down the convergence. Overall we think that more sophisticated techniques have to be deployed in order to utilize transfer learning in the context of learning control policies by deep neural networks such as the Actor-Mimic technique mentioned in Chapter 2.

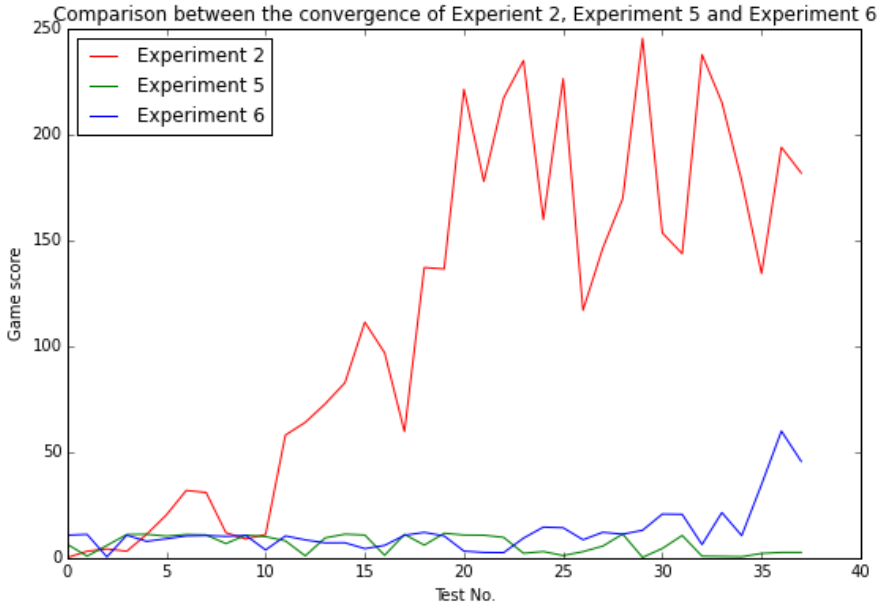

Figure 27. Comparison of the speeds of convergence in Experiment 2, Experiment 5 and Experiment 6.

There are some experiments that we wish we were able to perform in this work, but we were unable to do so, due to the lack of time and computational resources. For example, we would like to modify the network architecture used in Deep Q-learning by using more complex and deeper architectures. Judging from the research in the field of image recognition, where more complex and deeper architectures generally improve the performance of neural network systems, using more complex and deeper architectures might improve the results presented in this work. These architectures also come with a steep rise in a computational cost and considering the amount of training data generated by computer games, their utilization was simply not practical at the time of writing of this thesis. In future work we also wish to test the performance of Deep Q-learning algorithm on other, more sophisticated games, such as first person shooters, as well as in the domain of continuous control, for example in robotic control tasks. We would also like to test whether the techniques mentioned in Chapter 2 could improve the performance of this algorithm.

Conclusion

In Chapter 1 we examined the theoretical basis of reinforcement learning problems in general and basic RL algorithms with a focus on the model-free prediction and control. One of these algorithms is the Q-learning algorithm that is the basis of the update rule used in the Deep Q-learning algorithm. In this chapter we also examined the biological significance of RL and highlighted the evidence that the behavior of dopamine neurons in VTA and SN is consistent with Temporal-Difference RL algorithm.

In Chapter 2 we introduced the concept of value function approximation and provided an overview of how could modern deep learning methods make learning control policies from the high-dimensional representations of complex environments possible. In this chapter we also analyzed several Deep Q-learning algorithms both in the domain of discrete action spaces and in the domain of continuous action spaces. Finally, we introduced several methods from the realm of deep learning that could potentially be used to "emulate" some of the core cognitive competences of human and animal minds that are necessary for artificial agents in order to act as competent agents in the environment.

In Chapter 3 we provided the results of several experiments in Deep Q-learning. For these experiments we used our own implementation of Deep Q-learning algorithm. We managed to replicate the results from Mnih et al. (2013) for the Breakout game. We also managed to improve the original results from Mnih et al. (2013) in Breakout, Space Invaders and Pong games by using a different learning algorithm and the weight initialization technique. We also tested two different transfer learning techniques that are used in the domain of object recognition. We found out that these techniques are not suitable in the domain of Deep Q-learning and concluded that more sophisticated transfer learning techniques, such as those mentioned in Chapter 2, are necessary for this domain.

Designing competent artificial agents is one of the main goals of artificial intelligence. Historically this was proven to be an extremely challenging task due to the complexity of real world environments. We do not know if the current methods of deep learning or reinforcement learning are themselves capable of bringing us closer to this goal. It is possible that these techniques are just too inefficient. We think that inefficiency is the main disadvantage of current deep learning algorithms as demonstrated by the time and the amount of data needed by Deep Q-learning agents in order to perform well. The amount of training data could perhaps be reduced by endowing deep RL agent with core cognitive

competences by utilizing techniques from Chapter 2 of this work. It might be that even by utilizing these techniques the deep learning approach might not lead to competent artificial agents. We believe that answering this question is a viable avenue for future research. Even if deep learning methods eventually fail to bring us closer to competent artificial agents, we believe that this line of research might at least point us to the right direction.

References

[1] C. Szepesvári, *Algorithms for reinforcement learning*. Morgan & Claypool Publishers, 2010.

[2]R. Sutton, "Learning to predict by the methods of temporal differences", *Machine Learning*, vol. 3, no. 1, pp. 9-44, 1988.

[3] G. A. Rummery, M. Niranjan, "On-Line Q-Learning Using Connectionist Systems", Citeseer.ist.psu.edu, 1994. [Online]. Available: http://citeseer.ist.psu.edu/viewdoc/summary? doi=10.1.1.17.2539.

[4] C. Watkins, "Learning from delayed rewards", *Ph.D thesis*, King's College, 1989.

[5] W. Schultz, "Neuronal Reward and Decision Signals: From Theories to Data", *Physiology Review*, vol. 95, no. 3, pp. 853-951, 2015.

[6] W. Schultz, P. Dayan and P. Montague, "A Neural Substrate of Prediction and Reward", *Science*, vol. 275, no. 5306, pp. 1593-1599, 1997.

[7] P. Tobler, "Adaptive Coding of Reward Value by Dopamine Neurons", *Science*, vol. 307, no. 5715, pp. 1642-1645, 2005.

[8] K. Hornik, "Approximation capabilities of multilayer feedforward networks", *Neural Networks*, vol. 4, no. 2, pp. 251-257, 1991.

[9] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision", *Arxiv.org*, 2015. [Online]. Available: http://arxiv.org/abs/1512.00567.

[10] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar and L. Fei-Fei, "Large-scale Video Classification with Convolutional Neural Networks", *Cv-foundation.org*, 2014. [Online]. Available: http://www.cvfoundation.org/openaccess/content_cvpr_2014/html/Karpathy_Large-scale_Video_Classification_2014_CVPR_paper.html.

[11] Y. Kim, "Convolutional Neural Networks for Sentence Classification", *Arxiv.org*, 2014. [Online]. Available: http://arxiv.org/abs/1408.5882.

[12] P. Werbos, "Beyond regression: new tools for prediction and analysis in the behavioral sciences", *Ph.D thesis*, Harvard University, 1974.

[13] D. Rumelhart, G. Hinton and R. Williams, "Learning representations by back-propagating errors", *Nature*, vol. 323, no. 6088, pp. 533-536, 1986.

[14] J. Duchi, E. Hazan and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization", *Journal of Machine Learning Research*, vol. 12, pp. 2121-2159, 2011.

[15] M. Zeiler, "ADADELTA: An Adaptive Learning Rate Method", *Arxiv.org*, 2012. [Online]. Available: http://arxiv.org/abs/1212.5701.

[16] D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization", *Arxiv.org*, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980.

[17] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard and L. Jackel, "Backpropagation Applied to Handwritten Zip Code Recognition", *Neural Computation*, vol. 1, no. 4, pp. 541-551, 1989.

[18] D. Hubel and T. Wiesel, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex", *Journal of Physiology*, vol. 160, no. 1, pp. 106-154, 1962.

[19] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. Berg and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge", *Arxiv.org*, 2014. [Online]. Available: http://arxiv.org/abs/1409.0575.

[20] A. Krizhevsky, I. Sutskever and G. Hinton, "ImageNet classification with deep convolutional neural networks", *Advances in Neural Information Processing Systems*, 2012.

[21] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory", *Neural Computation*, vol. 9, no. 8, pp. 1735-1780, 1997.

[22] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning", *Arxiv.org*, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602 .

[23] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg and D. Hassabis, "Human-level control through deep reinforcement learning", *Nature*, vol. 518, no. 7540, pp. 529-533, 2015.

[24] H. van Hasselt, A. Guez and D. Silver, "Deep Reinforcement Learning with Double Q-learning", *Arxiv.org*, 2015. [Online]. Available: http://arxiv.org/abs/1509.06461.

[25] T. Schaul, J. Quan, I. Antonoglou and D. Silver, "Prioritized Experience Replay", *Arxiv.org*, 2015. [Online]. Available: http://arxiv.org/abs/1511.05952.

[26] T. Lillicrap, J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver and D. Wierstra, "Continuous control with deep reinforcement learning", *Arxiv.org*, 2015. [Online]. Available: http://arxiv.org/abs/1509.02971.

[27] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, et al., "Deterministic Policy Gradient Algorithms", ICML, Beijing, China. 2014.

[28] B. Lake, T. Ullman, J. Tenenbaum and S. Gershman, "Building Machines That Learn and Think Like People", *Arxiv.org*, 2016. [Online]. Available: https://arxiv.org/abs/1604.00289.

[29] A. Lerer, S. Gross and R. Fergus, "Learning Physical Intuition of Block Towers by Example", *Arxiv.org*, 2016. [Online]. Available: http://arxiv.org/abs/1603.01312.

[30] P. Pinheiro, R. Collobert and P. Dollar, "Learning to Segment Object Candidates", *Arxiv.org*, 2015. [Online]. Available: http://arxiv.org/abs/1506.06204.

[31] L. Chen, G. Papandreou, I. Kokkinos, K. Murphy and A. Yuille, "Semantic Image Segmentation with Deep Convolutional Nets and Fully Connected CRFs", *Arxiv.org*, 2014. [Online]. Available: https://arxiv.org/abs/1412.7062.

[32] S. Tsogkas, I. Kokkinos, G. Papandreou and A. Vedaldi, "Deep Learning for Semantic Part Segmentation with High-Level Guidance", *Arxiv.org*, 2015. [Online]. Available: http://arxiv.org/abs/1505.02438.

[33] K. Gregor, I. Danihelka, A. Graves, D. Rezende and D. Wierstra, "DRAW: A Recurrent Neural Network For Image Generation", *Arxiv.org*, 2015. [Online]. Available: https://arxiv.org/abs/1502.04623.

[34] T. Kulkarni, W. Whitney, P. Kohli and J. Tenenbaum, "Deep Convolutional Inverse Graphics Network", *Arxiv.org*, 2015. [Online]. Available: http://arxiv.org/abs/1503.03167.

[35] J. Oh, X. Guo, H. Lee, R. Lewis and S. Singh, "Action-Conditional Video Prediction using Deep Networks in Atari Games", *Arxiv.org*, 2015. [Online]. Available: http://arxiv.org/abs/1507.08750.

[36] E. Parisotto, J. Ba and R. Salakhutdinov, "Actor-Mimic: Deep Multitask and Transfer Reinforcement Learning", *Arxiv.org*, 2015. [Online]. Available: http://arxiv.org/abs/1511.06342

[37] M. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The Arcade Learning Environment: An Evaluation Platform for General Agents", *Journal of Artificial Intelligence Research*, vol. 47, pp. 253-279, 2013.

[38]X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks", In *Proceedings of the International Conference on Artificial Intelligence and Statistics* (AISTATS'10). Society for Artificial Intelligence and Statistics, 2010.

Other sources

D. Silver, "Advanced Topics: Reinforcement Learning (Lectures)", University College London, 2015, [Online]. Available: http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Teaching.html.