UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Zurabi Isakadze

Towards More Human Like Reinforcement Learning

Master's Thesis (30 ECTS)

Supervisor:	Jaan Aru, PhD
Supervisor:	Raul Vicente, PhD

Tartu 2017

Acknowledgments

I would first like to thank my thesis advisors Dr. Jaan Aru and Dr. Raul Vicente not only for their valuable contributions but also for the enthusiasm and inspiration they carry with them.

I would also like to thank PhD students at Computational Neuroscience Lab - Tambet Matiisen and Ardi Tampuu who first introduced me to the joy of training game agents.

Finally I would like to express my gratitude towards Estonian Foreign Ministry's Development Cooperation and Humanitarian Aid funds for supporting my study at University of Tartu.

Towards More Human Like Reinforcement Learning

Abstract:
Making machines more intelligent can potentially make human life easier. A lot of research has gone into the field of artificial intelligence (AI) since the creation of first computers. However, today's systems still lag behind humans' general ability to think and learn. Reinforcement Learning (RL) is a framework where software agents learn by interaction with an environment. We investigate possibilities to use observations about human intelligence to make RL agents smarter. In particular, we tried several methods: 1) To use "Tagger" - an unsupervised deep learning framework for perceptual grouping, to learn more usable abstract relationships between objects. 2) Make one RL algorithm (A3C) more data efficient to learn faster. 3) To conduct these experiments, we built a web based RL dashboard based on visualization tool - visdom. Finally, we provide some concrete challenges to work on in the future.

Keywords:
Reinforcement learning, deep learning, neural networks

CERCS: P176 Artificial intelligence

Inimesele sarnasema innustusõppe suunas

Lühikokkuvõte:
Masinate targemaks muutmine võib muuta inimeste elu lihtsamaks. Alates esimeste arvutite loomisest on palju teadustööd pühendatud tehisintellekti uurimisse. Sellest uurimistööst hoolimata jäävad tänapäeva tehissüsteemid alla inimaju üldisele võimele mõelda ja õppida. Innustusõppe raames õpivad tehisagendid keskkonna abil. Antud töös uurime, kuidas kasutada vaatlusi inimese nutikusest, et teha ka tehislikke innustusõppe agente targemaks. Me rakendasime mitut viisi: 1) kasutasime ”Taggeri” algoritmi - juhendamiseta sügavõppe viisi tajulise grupeerimise jaoks, et õppida kasulikumaid seoseid objektide vahel, 2) proovisime ühte innustusõppe meetodit (A3C) teha tõhusamaks, et selle abil kiiremini õppida, 3) Nende eksperimentide läbiviimiseks arendasime välja veebipõhise keskkonna innustusõppe katsete visualiseerimiseks. Lõpuks pakume välja ka suundi edasise töö jaoks.

Võtmesõnad:
Võtmesõnad: Stiimulõpe, sügavõpe, närvivõrgud

CERCS: P176 Tehisintellekt

Contents

1 Introduction

The long term goal of artificial general intelligence (AGI) is to create machines that will exhibit human-like problem solving skills in arbitrary tasks. The recent progress in Machine Learning, primary attributed to techniques called deep learning has produced remarkable results in many areas like classification, pattern recognition and control. The field of reinforcement learning has been no exception. In 2013 it was demonstrated that one algorithm which used neural networks as a function approximator could learn a variety of Atari video games solely from pixel inputs and even get human-comparable results in some of them [MKS$^+$13]. Since then, the interest in deep reinforcement learning has increased.

The excitement is easily understandable as the basic idea of reinforcement learning is simple, yet powerful - instead of training a model in a supervised way the agent itself interacts and learns from the environment. After all, this is how a baby learns. Fascinated by watching agents play, one can think that they operate much the same way a human does, but in reality today's systems are far away from challenging powerful human intellectual tools like thinking, planning, reasoning and transfer learning.

The goal of the thesis is to review and seek for possible ways to enrich the current reinforcement learning algorithms based on the observations of human intelligence.

Authors of ”Building Machines That Learn and Think Like People” [LUTG16], note that:

> Just as scientists seek to explain nature, not simply predict it, we see human thought as fundamentally a model-building activity. Children come with the ability and the desire to uncover the underlying causes of sparsely observed events and to use that knowledge to go far beyond the paucity of the data.

We hypothesize that some kind of intermediate representation of input frames, where relationship between objects and their visuals are separated from each other, should be beneficial for learning algorithms that acquire this knowledge. In this work we try to apply the tagger framework on one Atari game.

Also, one observation pointed by [LUTG16] is that computers needs far more training data to learn than a human does. While this may have many explanations, recent work like [PUS$^+$17] has tried to solve this issue by using the idea of episodic memory. The proposed memory module helps to alleviate a problem that neural networks are slow in integrating new experiences. We try to apply the similar idea to on-policy reinforcement algorithm A3C.

Emulated environments especially games are great platforms for studying AI, as they give us a simulated world which can generate unlimited amount of data. Notion of time and reward is inherently built into such systems and they were originally built for humans. More than that, it is possible to watch the recorded game and evaluate agents' behavior or compare them to human plays. Because of these reasons we use also built a live customizable web based dashboard for monitoring how the agents learn and evolve.

The work is structured as follows - in Chapter 2 we introduce basics of reinforcement learning, in Chapter 3 we discuss some ideas worth exploring to incorporate in reinforcement learning, in Chapter 4 our experiments are described and finally we end by discussion of possible future work in Chapter 5.

2 Reinforcement Learning

In this chapter we will review the basics of reinforcement learning and discuss several recent algorithms, as well as their important practical or theoretical improvements which led to the increased benchmark scores on Atari games. Although reinforcement learning has a long history, overlapping different disciplines and uniting many definitions or methodologies, here we will only concentrate on computational study in a context of Artificial Intelligence.

2.1 Basics

Reinforcement Learning (RL) is an area of Machine Learning which studies sequential decision making process of software agents. The goal of a RL agent is to maximize the expected total reward the agent can get over time. It can do so by interacting with an environment - taking some action and observing the rewards (scalar values). By trial and error an agent is supposed to improve its strategy (in reinforcement learning terms a policy) over and over, discovering more about its environment and exploiting the knowledge it did not have in the beginning. An important aspect to notice is that by taking each action an agent can potentially modify the environment in which it navigates, thus changing the outcome of its future observations and reward signals.

Reinforcement learning has some ties with Behaviorism - an old school view at how biological organisms learn. Imagine a hungry chimpanzee - Mike sneaks into an abandoned lab where crazy scientists used to do experiments. While searching for some food, Mike explores the lab by looking around and touching things (getting observations from the environment via senses). Eventually, he finds a vending machine with several buttons on it and accidentally pushes one. Suddenly a nut pops down (positive reinforcement). Excited Mike quickly swallows the nut and continues pushing buttons arbitrarily. This time Mike gets a small electrical buzz (negative reinforcement). After a while he notices that pushing red buttons gives him a nut, yellow and blue ones a buzz, so he adjusts his policy only choosing red buttons. Soon Mike might get bored of his peanut diet and tries pushing other buttons again, even though he expects a small buzz. After some time Mike can discover that if he pushes the red button right after pushing the yellow button in addition to an electric shock he gets a fine banana which of course is a good bargain!

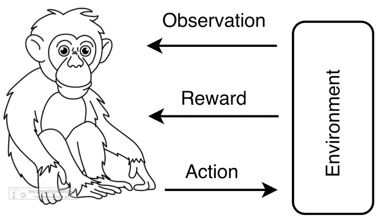


Figure 1. (clipart from [Cli])

Of course this explanation of learning is way oversimplified, especially in the case of human beings. Nevertheless, several brain imaging experiments have shown a correlation between responses emitted by dopamine neurons in a brain and signals predicted by reinforcement learning ([Niv09], [DT13]).

There are several main differences between RL and standard supervised learning. First of all, in RL one does not have true labels (actions) to train a model. Also the future reward and observation distributions very much depend on decisions made by the agent. Moreover, the only feedback one obtains from the environment about how good taken actions were, might be delayed in time so it is usually not easy to know which decisions were beneficial. Because of these reasons RL usually falls into a semi supervised learning category.

One important problem in RL is the "exploration exploitation dilemma" [RN95]. Even though an agent might have some idea about future rewards, sometimes it has to take actions with unknown outcomes or even decisions leading to negative rewards in the short term (like our Mike), hoping to learn more about the environment to maximize the total expected reward.

2.2 Definitions

Here we will define the RL environment as a Markov decision Process (MDP) with finite state and action space. States in MDP are expected to satisfy the Markov property that is:

$$P[S_{t+1} \mid S_t] = P[S_{t+1} \mid S_1, S_2, \dots S_n)]$$

In other words if an agent is in a particular state, outcome of the next state does not depend on the previous history of visited states. This property implies that when talking about RL in MDP, we should assume that either the environment is fully observable (in which case agents internal state is the same as that of environment's) or the chosen state representation satisfies this property. In real world almost all problems are partially observable, but these approaches can be generalized in Partially Observable Markov Decision Process Process (POMDP).

At each time step t, the agent observes a state - s_t from some state space S and acts by making some action a_t from $A(s_t)$. Doing so, it gets a reward - r_t and moves to the next state - s_{t+1}. In episodic tasks an agent continues to do so until it encounters a state which is defined as terminal, at which point the episode is declared to be finished.

A policy - π defines a mapping between states and actions chosen by the agent, in other words its behavior. Policy can be stochastic as well - $\pi(a \mid S)$, in which case it defines a probability distribution over actions.

As we already noted the agent's goal is to maximize the total discounted return:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

where $\gamma \in [0, 1]$ is a discount rate, denoting how important rewards in time are. We assume that the terminal state infinitely generates 0 rewards. Many RL algorithms only deal with discounted returns because this sum is required be finite. It also makes sense intuitively as humans and animals often seek for immediate rewards.

We also define a value function of a state, which is just an expected value of the return from this state following a policy π:

$$V_\pi(s) = \mathbb{E}_\pi[R_t \mid s_{t,i} = s]$$

where $\mathbb{E}[\cdot]$ defines an expectation when t is any time step and i is any episode. Will see later the value function is an important concept in RL framework. From now on, we will drop i from state notation to make it cleaner.

Similarly an action value function $Q_\pi(s, a)$ is defined as expected return when taking an action a, in state s and then following the policy π:

$$Q_\pi(s, a) = \mathbb{E}_\pi[R_t \mid s_t = s, a_t = a]$$

The optimal action-value function is simply a maximum action-value function over all possible policies:

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

2.3 Looking for optimal policy

Finding a solution for RL problem means to find an optimal policy. On a higher level there are two classes of algorithms for achieving this goal - In **model-based** learning there is a model for predicting rewards and state transition probabilities for each state action pair. Here we briefly review basic ideas of some of the **model-free** algorithms, where such model is not needed.

In order to find an optimal behavior one can try to start with a random policy, evaluate its value function, improve the policy (for example acting greedily on evaluated function), evaluate it again and so on. This general algorithm is known as policy iteration. For example, if the MDP is known, one can use dynamic programming for policy evaluation. Bellman equation expresses the recursive relationships for a value of s and its possible successor states.

$$V_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(s_{t+1}) \mid s_t = s]$$

Instead of solving this equation one can modify it to the iterative update rule for the state values in a following way:

$$V_\pi^{k+1}(s) = \sum_{a \in A} \pi(a|s) \left(r_s^a + \gamma \sum_{s' \in S} P_{ss}^a V_\pi^k(s') \right)$$

It can be shown that greedy policy improvement according to the calculated state values can stop if and only if Bellman optimality equation holds : $V_\pi(s) = \max_{a \in A} Q_\pi(s, a)$ and it is guaranteed to converge to a deterministic optimal policy.

An alternative approach for solving RL is to directly update action-value function to drive it towards optimal action-value function. In this case intermediate updates might not correspond to any policies (off policy learning). However if the optimal action-value function is known the agent can choose an action with highest value at each step - hence the optimal policy is also known.

Temporal difference learning is one such method. Suppose one has the way to improve the policy based on the current action-value estimates. (In practice popular choice is ϵ greedy, where with probability ϵ agent explores, and with $1-\epsilon$ it acts greedily). Then update rule in *Sarsa* which is an **on-policy** algorithm can be written as.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

Intuitively action-value is updated a towards better estimate, under the current policy. Notice that transition probability is not needed here, therefore the method is model free. There are many modifications of this update rule including Sarsa(λ) which uses n step Q-returns and other tricks to increase the efficiency and reduce the variance while learning.

In **off policy** learning, behavior policy is separated from the target policy. In other words, while the agent follows an actual strategy - $\mu(a|s)$ it tries to learn or evaluate the different policy - $\pi(a \mid s)$. One important motivation for off policy learning is that target policy π can be chosen to be an optimal policy. Thus, it can be learned while an agent tries to explore the environment with behavior policy μ. One way to learn the target policy by learning its action-value function is the following update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1}, a_\pi) - Q(s_t, a_t)) \tag{1}$$

where $a_{t+1} \sim \mu(\cdot \mid s_t)$ and $a_\pi \sim \pi(\cdot \mid s_t)$.

Q-learning (first introduced by Watkins in 1989 [WD92]) is an efficient and well-known algorithm which uses this idea. Here the target policy π is an optimal policy which is obviously greedy with respect to $Q(s, a)$, while behavior policy (exploration) implements ϵ-greedy policy mentioned above. Thus, both policy gets improved over time. Following formula is the result of (1) when a_π is replaced according to such policy.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

For more thorough introduction to reinforcement learning please refer to the book - "Reinforcement Learning: An Introduction" (Sutton and Barto).

2.4 Deep Reinforcement Learning

So far we have not discussed how policies or value functions are represented in computers. A straightforward way to keep action-values is to keep them in a table. However this is inefficient in most cases since the state space is high dimensional or action space is continuous. Deep reinforcement learning uses deep neural networks (DNN) for approximating value functions, policies or both.

TD-gammon - one of the first system that could beat professional backgammon players, successfully used neural networks in RL as early as in 1994 [Tes94]. Surprisingly, until the introduction of Deep Q-network (DQN) in 2013, such approaches remained rarely used in practice, for several reasons.

From theoretical view, many RL algorithms loose convergence properties when used in conjunction with NNs. In practice the optimization process is slow and unstable. Also, because the reward signal is sparse and delayed, it causes problems associated with class imbalance for neural networks in standard supervised learning. Furthermore, states encountered during online training are highly correlated.

Next, we will review some algorithms from recent history. Contribution of the following methods are that they came up with some ways to stabilize the process and make training data and compute efficient.

2.4.1 Deep Q-network

DQN [MKS$^+$13] was able to successfully learn a variety of (Atari 2600) games based only on pixel input and reward scores. Most surprising part for general audience was that it did so without changing program or model parameters from game to game and no feature engineering was used. In addition, these games are diverse, both by gameplay and graphics, for example some of them are 3 dimensional. In many of them DQN was able to play equally or better than professional human players.

The DQN algorithm is based on Q-learning and it introduced two key ideas for training to alleviate the problems associated to neural networks. First one is using Experience Replay [Lin93] to decorrelate samples collected during training. The second insight is adjusting parameters for action-value function towards a slowly moving target.

During the training DQN performs an update on a batch uniformly sampled from the experience replay which stores a buffer of recent experiences (triples of state, action, rewards). Essentially a network is trained in a supervised manner with stochastic gradient descent to minimize the following loss function at iteration i:

$$L_i(\theta_t) = \mathbb{E}_{(s,a,r,s`)\sim U(D)}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\right)^2\right]$$

Here D is an experience replay and θ_i^- is a target networks parameter at i th iteration.

Usage of experience replay also increases the data efficiency compared to standard Q-learning as each sample can be used many times. However, it also has a restriction to off-policy learning algorithms since these samples are not generated by a current policy.

DQN used a convolutional neural network (CNN) as a Q-value function approximator. In general the success of deep neural networks in the last years could be attributed to

CNNs. This type of network is well suited for vision, because layer by layer, it is able to extract increasingly complex features from images. In DQN, the input to the neural network at each time point are the last four images from the game emulator. This allows to incorporate tiny history of time into the state.

It is interesting to know what kind of representations this network learns in Atari games. Authors of "Graying the black box: Understanding DQNs" [ZBZM16] tried to analyze it by visualizing activations of last hidden layer via t-Distributed Stochastic Neighbor Embedding (t-SNE). They show that DQN maps the space to a lower dimension not only based on pixels and value estimates, but according to temporal structure as well. For example, in the game Breakout, clusters are distinguished by ball's relative position and direction. However in some special cases, like a bug in a game, there is a separate cluster which suggests that network also captures game dynamics.

2.4.2 Asynchronous methods

Until now we were implicitly learning policies by using value functions. There is a class of methods called **policy gradient** which explicitly tries to learn the desired policy function by parametrising it $\pi(a \mid s; \theta)$ and following the gradient. One definition of an objective function is the expected value of all states - $\mathbb{E}_{\pi_\theta}[r]$. It is possible to transform the gradient of this expectation using the likelihood ratio trick to the following expression:

$$\nabla_\theta J(\mathbb{E}_{\pi_\theta}[r]) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) Q_{\pi_\theta}(s,a)\right]$$

To increase the value of an objective function, one can now follow the gradient ascent of this expectation. An intuitive way to understand why it works is the following - via the gradient of our policy functions one can increase or decrease the probability of a particular action in a given state, based on the outcome of how good this action was $Q_{\pi_\theta}(s,a)$. Calculating the gradient of the log policy function is easy (assuming the it is differentiable), the problem is that the true action-value function under the policy is not known. One way to estimate it is by sampling unbiased Monte-Carlo returns instead. However, this approach yields to a very high variance. Instead, in actor-critic methods, separate parametrized value function is used to estimate the action-values (critic), which leads to approximate policy gradient. Another way to further reduce the variance is by subtracting the baseline function from value estimate. If we choose a baseline to be a state value function, the difference is called an advantage function:

$$A_{\pi_\theta}(s,a) = Q_\pi(s,a) - V_\pi(s)$$

Notice that this idea does not introduce any bias, but greatly reduces the variance. Finally, the approximated gradient becomes:

$$\nabla_\theta J(\mathbb{E}_{\pi_\theta}[r]) \approx \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) A_{\pi_\theta}(s,a)\right]$$

In general policy based methods have better convergence properties. They are particularly useful for partially observable and stochastic environments, as there simply might not exist a deterministic optimal policy.

In 2016 Authors of [MBM+16] introduced a simple and lightweight framework for deep RL by using asynchronous gradient descent for optimization. As we already saw, experience replay was successfully able to stabilize the training for Atari games. But experience replay has some drawbacks, namely these algorithms need more memory and they are limited to off-policy learning.

Instead, the authors proposed to asynchronously run multiple agents in parallel, each using separate instance of the environment. Because each agent will experience different sequences of states, their parallel updates will have a decorrelating effect on the training. This trick additionally enables on-policy methods like Sarsa or actor-critic to be used with deep neural networks.

Asynchronous methods also bring great practical benefits as training time scales linearly with respect to number of learners. While previous algorithms needed to be run for days on GPUs [MKS+13] or distributed clusters [NSB+15] to solve Atari games, these models can be trained on a single multi-core CPU machines with far reduced time.

Weight in the optimization process are updated using HOGWILD style methods [RRWN11] which do not require locking mechanisms allowing the gradients to be overwritten by each other when using parallel processes. This trick leads to an increased performance. In addition authors used n-step Q-estimates, where for estimating the value function in forward view, n true episode rewards are used as well -

$$r_t + \gamma r_{t+1} + \ldots + \gamma^2 r_{t+n-1} + \gamma^{n-1} Q(s_{t+n}, a)$$

This way rewards can propagate back faster.

In experiments reported by the paper actor-critic performed the best out of several methods used in asynchronous manner and they named the algorithm as A3C.

2.4.3 Neural Episodic control

As discussed by authors of [PUS+17], even though previous algorithms were effective in learning Atari games, they still have one common drawback as they need far more game interactions than a human does. For example DQN needs 200 hours of video frames for 47 Atari games to get more or less the same results obtained by human player in 2 hours [LUTG16]. While there are many possible explanations as reviewed in a previous chapter, there are several issues : 1) Gradient descent optimization requires use of small learning rates, also common for other deep learning methods 2) There is an imbalance between low and high rewards, thus neural network performs worse while predicting larger rewards 3) reward signal propagates back slowly, even when using n-step Q-estimates.

Lengyel and Dayan [LD08] argued that the episodic memory has an important role in biological decision making process, but it is rather neglected in software control systems. Idea of episodic memory is that compared to semantic memory which can be looked as a general accumulated statistics about the environment from multiple events (for example we know that when it is raining, we will get wet), episodic memory is about concrete past experiences (it was raining on yesterday when I went outside). Authors also strengthened their argument by showing experiments on a simple task.

Blundell et all, [BUP$^+$16] extended the idea and applied it on more challenging tasks like the Atari environment and Labyrinth (3D test environment by DeepMind). As expected [BUP$^+$16] significantly increased data utilization, outperforming DQN and A3C as well as other algorithms on limited number of game frames.

Neural Episodic Control continues on the same direction, but instead of using Q-table for "episodic memory", authors propose to use module called differentiable neural dictionary (DND). DND is placed on top of convolutional network and tries to map state embeddings to its value estimates. This architecture looks more like table based Q-learning, and learning rate can be higher. Convolutional network serves as stable representation of states.

For review of other recent ideas in reinforcement learning see [Li17].

3 Possible directions for improving AI

Humans have been imagining "thinking" machines for a long time. The idea became somewhat less fictional after the introduction of programmable digital computers in 1940s [Buc05]. In the last years, Great progress has been made towards using neural networks to solve various problems, partly because of increased computational power and renewed interest in AI research. However, we agree with those who feel that important parts are missing from human like intelligence [LUTG16]. Since there is still no clear directions to follow, it should be vital to think about what these crucial parts are and how they can be implemented in machines. In this chapter we discuss some of the issues related to this topic.

3.1 What is the AI trying to solve?

Before we move to exploring ideas, it is first interesting to discuss what an AI is and what goals researchers are trying to achieve. Unlike systems that mimic specific capabilities of humans - like playing chess, we will mostly focus on something that is defined as a long term goal of AI: Artificial General Intelligence.

For categorizing different definitions of AI throughout the history, authors of [RN95] suggest to take a look at the following table:

Thinking Humanly	Thinking Rationally
Acting Humanly	Acting Rationally

Table 1. Categorizing different definitions of AI according to [RN95].

On top there are goals concerned with thought process and reasoning, while on the bottom we have ones dealing with behavior. In the left column success is defined in terms of human performance and on the right it is rationality - always doing "the right" thing.

Out of these four, probably acting humanly is the easiest to comprehend: if a person cannot distinguish machine from a human being in a well designed test, the problem would be considered as solved. On the other hand thinking like a human implies understanding the thought process in our minds. According to the same authors - "there are three ways to do this: through introspection - trying to catch our own thoughts as they go by; through psychological experiments - observing a person in action; and through brain imaging - observing the brain in action." These are some of the problems cognitive science tries to solve, therefore results obtained from this field should be valuable.

Thinking rationally can be referred to using mathematical logic and formal proofs to solve some of the common world problems. We will briefly review this approach in the next section. Agent acting rationally tries to achieve the best expected outcome, based on defined utilities. This approach is more plausible from mathematical viewpoint as the expectation maximization is a well defined term and it does not strictly require any human like reasoning. Many computer science researchers choose to work on this definition (for example, RL framework discussed in 2.4). There can be thousands of interesting philosophical discussions about each points, for some see the chapter "Philosophical Foundations" [RN95]. Finally it should be noted that these definitions do not necessarily exclude each other and one can try to build systems that satisfy each condition to certain degrees.

In general we think, that cognitive modeling approach is worth exploring more. It might even be surprising that despite the abundant experiment results accumulated in the field, it is hard to come up with a feasible computational frameworks.

Of course one could work on a completely different path, where this theoretical modeling is not needed and directly try to simulate the brain on a molecular or lower level. There has been some work toward this direction, for example Blue Brain project [Wik] tried to simulate a part of the rat neocortex. But current computational power and resolution of brain imaging technologies does not look very promising to accurately map human brain on a machine.

3.2 Symbolic reasoning

When AI research started in 1950s, the main paradigm was using mathematical logic to express and manipulate knowledge with symbols. John McCarthy created the LISP programming language which became a popular tool to develop these programs. For example one such formal system is First-order logic, where objects (Apple, Tree), relations (HangsOn, Red) and functions (root) are represented with symbols. There is also a syntax defined and some natural sentences can be represented in this language. For example:

There are some red apples that hang on a tree.

Then, humans can enter facts into a knowledge base - *there are some apple trees*, *an apple can have a color*, *color can be red and so on.* And a program could answer questions like - is there a red apple? Of course such kind of formal systems were much more useful. For example the General Problem Solver [NSS59] developed in 1959 was theoretically able to solve any kind of problem where the input was expressed as well-formed formulas, but in practice it was limited to simple problems such as Hanoi Tower, because A) There are only handful amount of problems which can have such formulation B) while solving, many of them will lead to a combinatorical explosion C) Generalizing or learning function in such kind of system seems unclear.

Over the years interest in symbolic AI has declined, primarily because the hand-crafted representations could not capture the rich statistics humans get while observing the continuous world around us. On the other hand neural networks excel at learning patterns from data, but how to convert this knowledge to more abstract level stays a hard problem.

3.3 Probabilistic Machine Learning

On a very basic level learning in Machine Learning can be seen as making better predictions via the model as more and more data becomes available. Anything is rarely certain during this process. Probability theory in mathematics provides a framework to express and manipulate this uncertainty in a consistent way. This approach in machine learning is known as probabilistic modeling and it has been one of the major area of research in the past years. It is important to note a distinction from deep neural network models, where usually the uncertainty is not explicitly tracked. For example Bayesian learning depends on the Bayes rule to update the prior distributions into posterior distributions according to the following formula:

$$P(\theta|D,m) = \frac{P(D|\theta,m)P(\theta|m)}{P(D|m)}$$

where m is a model, θ model parameters and D - observed data. Most recent Bayesian approaches can be seen as using this transformation to build more complex models (such as graphical models) where probability distributions of random variables and their conditional dependencies are expressed. Such examples include Bayesian networks or Markov networks. One practical challenge in this type of systems is calculating marginal probabilities as they involve integrals for which no polynomial algorithms exist. Sometimes methods like Markov chain Monte Carlo can be used to approximate them, but they are usually computationally expensive. Another issue is that the model needs to be flexible enough so that it continues learning. There are two ways to achieve the latter - number of parameters needs to be large enough for the problem or non-parametric models should be used. There are some experimental observations that the neural circuit in a brain is implementing something similar to Bayesian learning [TKGG11] but as of now it is not known exactly how.

Recent promising extension to graphical modeling is probabilistic programming, where instead of graph structure, computer programs (code) represent probabilistic models.

For thorough review of probabilistic methods in machine learning please refer to [Gha15].

3.4 Language

It is not hard to notice an inner voice going in our mind. It looks like that we think in a natural language, but it may only be a conscious experience - according to the language of thought hypothesis thinking takes place in an underline mental language with its own syntax and semantics [Fod75]. One can wonder how important language is for human intelligence. The evidence says it is critical and understanding how it is processed in the brain can be a huge leap towards general AI.

Experiments show that brain activity in language related centers can be detected by functional MRI scans in infants as young as five days old. In the first weeks babies can distinguish a melody of the native language from other language [DLDHP02]. Contrary to the intuition, it turns out that deafness has far greater consequences on development of a child than blindness. This is not due to the any special property of the sound signal, but rather because language development is crucial in early ages. Deaf children who have no signing parents might develop a language deficiency, even with cochlear implants which partially recovers speech perception. Also, if a hearing impaired baby is exposed to the signing language earlier (in addition to the spoken one) they even outperform hearing controls in theory of mind and lexical comprehension tests [TVDR+12].

Natural language has a central role in a paper - "Roadmap towards Machine Intelligence" [MJB15]. The authors suggest an interactive way of training agents, where the primary means of communication is language. After all, if we have intelligent machines, reasonable ways for humans to give commands to or ask questions from them is to use a spoken language. Besides, large part of humanity's knowledge about the world is represented as a text. If machines can learn from a text they can potentially use all this knowledge.

3.5 Model building and start-up software

Now we will discuss few insights from a paper - Building Machines That Learn and Think Like People [LUTG16]. Authors note the limitations of current research trends in AI, but instead of criticizing the deep learning methods, they suggest to use them in a smarter way.

The central idea of the work is to look learning as the process of model building, In other words one should seek to explain the observed data through a construction of causal model. To achieve this goal, according to authors we need a developmental "start-up software" - cognitive capabilities that are early present in human development. There is an argument that if they are presented in humans even earlier than a language, there should be something special about them that makes us able to learn. The two proposed components of such software are:

1. Intuitive physics - even infants have understanding of some object properties and very basic physics. For example they expect the objects to follow some path and not suddenly disappear. Toddlers might look very surprised when if they see a simple magic trick. (It even seems to be true for animals, there are anecdotal videos of dogs being scared by a levitating sausage.)

2. Intuitive psychology - infants understand beliefs and goals of other people, for example children watching games, can guess the objectives of player and what the enemies are trying to achieve.

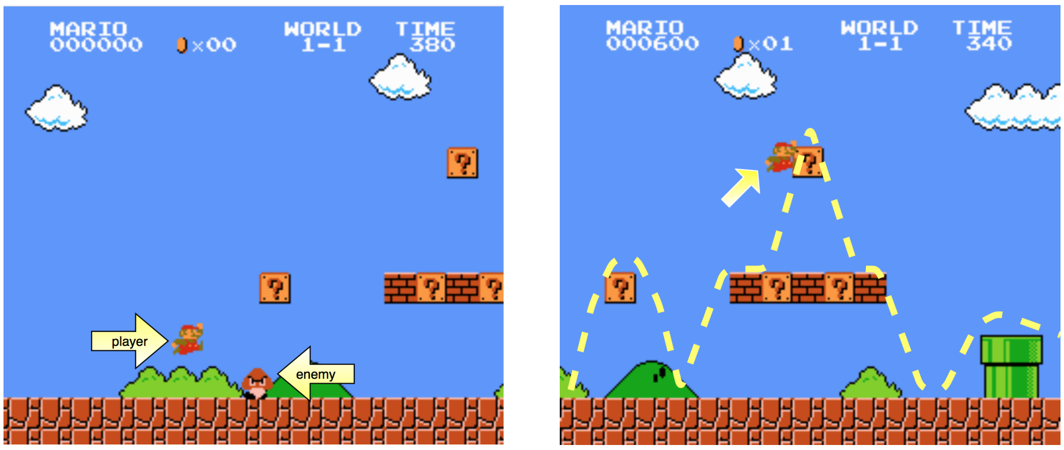
If end-to-end trained deep learning models can extract mentioned causal dependencies it is not clear how to reuse, represent or generalize them. One question is - should one try to perform object segmentation, represent extracted objects as symbols along with their spacial positions and then process this information for example with relational reinforcement learning [DDRD01]? This approach seems to have some considerable problems. In the real world object projections are entangled in many ways. It is not clear in what level they should be separated or what counts as a background. On the other hand, humans clearly perceive objects separately. For example consider a woman enjoying a nice view in nature: grass, stones, trees. After some time, something black moves in the grass. The person directs her focus and also sees a chasing cat behind. Now the scene for her becomes live, she sees that there is a hole under the tree, and predicts that mouse might be running towards that place. Suddenly these three objects are in the center of attention and other things move to the background. Thus attentional mechanism, behavioral context and curiosity all had the effect in which way the objects were separated and tracked. It might look like a trivial example, but actual number of objects humans can track at a time is not much higher [AF07].

3.6 Curiosity and motivation

When introducing RL in section 2, we looked at reward as an external signal, computed by the environment. In a biological system however this computation is a part of an agent itself - brain releases neurotransmitters for a rewarding stimulus. Also, in many cases humans do not act for getting any "external" rewards. For example unlike eating food, we play games for enjoyment. This kind of behavior is also known as intrinsic motivation, because it is intrinsically rewarding [Sch10]. Another interesting thing to is that, many of our actions are guided by curiosity. Hoping to get some insights, we did a small experiment and recorded how a very young child would explore and navigate in one game.

The classic NES game - Super Mario Bros (shown in 2a) was chosen, because of its simplicity and gameplay. Mario moves to the right and encounters moving enemies which either he should jump or squash. On top, there is a score shown, but goal of the game is to travel and go through different levels (Mario worlds).

The subject was a typical 39 month old boy. He had previously no experience with video games, not even on a smartphone or on a tablet. Hence, this was the very first time he ever tried a game. Because the subject had never held a gamepad, to make it easier for him, we restricted controls to two buttons - *jump* and *move forward*. Prior to the game the boy was not explained how the game works, he was only told "here is a game, you can



(a) Super Mario Bros. Mario meets an enemy (b) Episode 9 strategy

Figure 2. Two visually different but conceptually similar game scenarios.

push these two buttons, just try pushing them". It was not explained which character the gamepad controls or what is the goal of the game. During the game he looked engaged, but it was unclear whether he understood what was going on. However, after playing for 2 minutes, he gave a spontaneous explanation: "I made the boy jump and there were ghosts". When asked about why he needed to jump the boy said: "To get over the barriers ... to get higher". After 3 minutes, he was able to play fairly well. In the later episodes, it was interesting to see that the boy was always trying to jump on a higher ground. For example on 2b movement is denoted by dashed yellow line. There is no enemy on the ground and the shortest path would be to just walk and jump over a green barrier. The player even spent considerable time trying to get on the highest brick.

The game score was not a guide for the player, as he could not even read these numbers. Also, even the goal of the character was not known. This is also true for many adult persons playing action games. One factor of motivation in adventure games like this is probably curiosity to see new scenes and game characters. Another factor can be the pleasure of defeating enemies. For these reasons, we think that future generation of RL algorithms will not even use explicit reward score from the game emulator, and instead it will be based on intrinsic motivation.

4 Experiments

In this chapter we review the work done for building a dashboard and some of our experiments along with the results.

4.1 Testing environment

In the Introduction, we already reviewed the benefits of using games as test environments for RL algorithms. Initially, we were planning to use Nintendo Entertainment System (NES) [Nin] as a test platform for several reasons. Compared to the most common emulator used in RL benchmarks (Atari 2600 [BNVB13]), in NES gameplay is richer and there are many objects to interact with (Figure 5). An agent also has time to explore the environment without dying soon. Furthermore, in most of the games graphics still stays 2D, removing extra work for model to handle 3D scenes. Above all, games have a storyline, visual appearance of the world changes from level to level (but physics stays the same) and sometimes agent even has a sidekick (hence there is a possibility for intuitive psychology experiments) subsection 3.5.

Unfortunately, even Atari games take many hours to train on a modern computer. So using NES would need powerful computing resources for current RL algorithms. Besides, there is no known open source RL wrapper of a NES emulator, and making one where rewards are provided would take some time. So we decided to go back and experiment with a well tested Atari environment for this work.

4.1.1 OpenAI Gym and Atari

OpenAI Gym [BCP$^+$16] is an open source toolkit for developing and comparing RL algorithms. In essence, it provides a simple and standardized API for different environments. For example, the following snippet creates a new *FrozenLake* environment. gym then processes an action chosen by the agent and returns a new observation along with a reward and a boolean flag weather the episode is finished or not - *done*.

```
env = gym.make("FrozenLakev0")
observation = env.reset() # get an initial observation
action = my_agents_policy(observation)
observation, reward, done, _ = env.step(action)
```

Internally gym uses Arcade Learning Environment platform to emulate Atari games [BNVB13]. Observation is an RGB image (210 X 160 X 3) of a game frame. To make a game stochastic, random number of frames (2 - 5) are skipped. There are also deterministic versions of some environments, but their solutions are rather uninteresting as an agent sometimes learns to repeat the same sequence of winning moves for every episode.

In addition gym provides a way to wrap some functions. For example to change the observations or rewards which agent gets. This way it is possible to normalize or stack the game frames conveniently.

4.1.2 A3C implementation review

We decided to use PyTorch [Pyt] as a deep learning framework mainly because its dynamic nature of computation. Numerical gradients makes debugging easier than other deep learning frameworks that use static computational graphs.

Baseline implementation for A3C algorithm in PyTorch (subsubsection 2.4.2) was taken from github [iko]. Repository for our modifications and additions can be found publicly on github as well [sci]. The implementation uses processes to run agents in separate instances of an environment. Agents keep their own networks and after each episode, gradients are applied to the shared network via the Adam [KB14] optimizer. Neural network architectures are defined separately in *models/* directory. Training scripts can be found in *algorithms/* folder. *envs.py* hosts the gym wrapper functions. For efficiency, we convert the frames to 42X42 greyscale images and use the running normalization. Also LSTM network [HS97] is used after the convolution layers.

4.2 Building a live dashboard for monitoring software agents

Because training a deep neural network is slow and sparse rewards in RL problems make it even slower, it is useful to have a good live visualization of the process and keep logs in an effective way. In this way one can detect whether there is something wrong with the run and save both their own and computing time. Next we will describe how we built such live dashboard in python using Visdom [Vis] and SQLite [SQL].

In the past few years several great deep learning frameworks such as Theano, Keras, Caffe, Torch have been released, however the task of visualization and logging is usually left to users. One of the exceptions was Tensorflow, which came with a tool for training visualization called *Tensorboard* [Ten], but at the time of writing one can find following problems with it:

- There is no way to see all parameters corresponding to the specific run.
- Tensorboard supports scalar and histogram summaries, but in practice many different plots might be needed for better visualization.
- For different projects different layout of dashboard can be useful. For example sometimes it is helpful to see two heatmaps next to each other.

4.2.1 Review of a visualization tool - visdom

As we already mentioned, one of the main advantage of working on RL from the gaming perspective is that one can observe what problem an agent has or how it evolves over time. So a powerful visualization tool is even more important than in typical deep learning scenarios and we decided to try an software by Facebook research.

Visdom is an open source live visualization tool powered by Plotly [Inc15] windows are composed of independent panes, which can be easily dragged, resized or closed. Each pane can host an interactive graph, for example, it is possible to zoom in or change the perspective of 3D scatter points. There can be many windows and it is easy to switch between them.

4.2.2 Integrating visdom with the a Deep Learning framework

Unlike Tensorboard, Visdom does not yet come with live file log. So directly calling visdom from the evaluation function during training is probably a bad idea. When visdom server shuts down, all evaluation data is lost. There was a need for some intermediate live storage where the logs would be safe. We decided to use SQLite [SQL] because of its server-less and simple design.

In the beginning of each experiment, the database file corresponding to that run is initialized, say - *run.sqlite3* and then each time there is something to log, one just serializes data in the code, turn it to a byte string and save it as a BLOB (Binary large object) in a database. If the database ID is incremental, it is possible to follow the same order when other script reads the log file later.

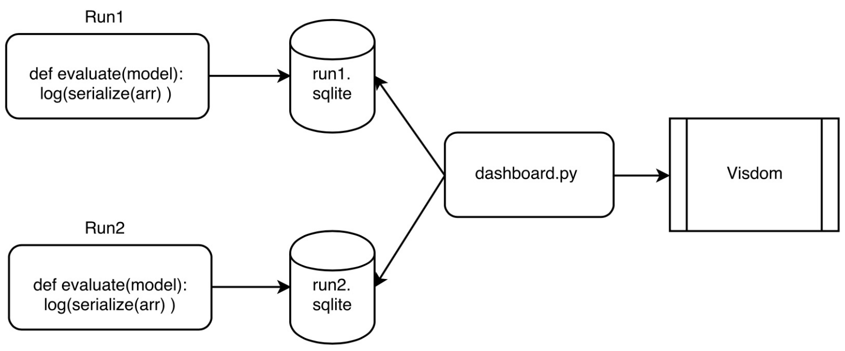


Figure 3. flowchart of our simple logging architecture [Dra]

We then have a separate script (*dashboard.py* 3) which connects to these databases, waits for an update and if there is something new, deserializes it and immediately calls visdom API to update plots in a real time. Other advantages of this approach are the following:

- The project does not depend on a visualization framework.
- Instead of running on a same server to render plots, visdom can be run on a local machine which has access to the database.
- There is no need to browse in folders, all of training history - model checkpoints, run parameters, videos, will be contained as a single *.sqlite* file.
- SQLite is easy to install on most systems.

4.2.3 Serializing the log data

We give our logs an event name, for example it can be a string like *QuickEval* (for quick evaluation test) or SlowEval (For slow evaluation which might have images or videos as a data). When logging the data is needed, we construct a Python dictionary with desired keys and values in addition to an event name. In the code it might look something like this:

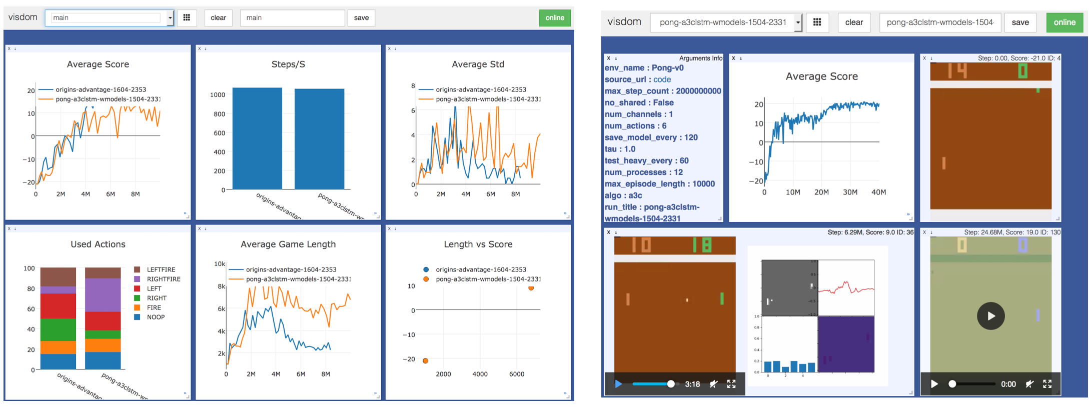
```
data = {'evtname' : 'QuickEval', 'std' : std, 'result' : np.random.rand(2, 3)}
dblogger.log(data)
```

dblogger instance of our class will internally use Pythons in-built serialization library pickle to convert this dictionary into a bytestring or further compresses it. The main requirement is that dictionary values are recognized by pickle, which is true for at least Python standard object types and numpy [Num] arrays. *dblogger* will then commit to database *(eventname, objectstr)* pairs. An advantage of using standard dictionary instead of creating our own class is that a reader class will not depend on any schema changes and always will be able to deserialize the object. It is better to still validate the data before logging, for example with python package voluptuous [ale], where schema for the dictionary above will look like this:

```
Schema({
  'evtname': 'QuickEval' # we force it to be a correct name
  'std': float,
  'result': np.ndarray,
}, required=True) # all fields are required
```

4.2.4 Dashboard review

Experiments can be run on a cluster and logs are written to live files. *dashboard.py* can be run locally if it has a way to connect to database files. On Linux it can be done by


visdom
main
clear
save
online
Average Score
Steps/S
Average Std
Used Actions
Average Game Length
Length vs Score
origins-advantage-1604-2353
pong-a3clstm-wmodels-1504-2331
LEFTFIRE
RIGHTFIRE
LEFT
RIGHT
FIRE
NOOP
pong-a3clstm-wmodels-1504-2331
Arguments Info
Step: 0.00, Score: -21.0 ID: 4
Step: 6.29M, Score: 9.0 ID: 36
Step: 24.68M, Score: 19.0 ID: 130

```
env_name : Pong-v0
source_url : code
max_step_count : 2000000000
no_shared : False
num_channels : 1
num_actions : 6
save_model_every : 120
tau : 1.0
test_heavy_every : 60
num_processes : 12
max_episode_length : 10000
algo : a3c
run_title : pong-a3clstm-wmodels-1504-2331
```

(a) main window (b) specific run window

Figure 4. Screenshots of a dashboard opened in Firefox

mounting a remote log directory to local folder. We can then run visualization script with the arguments shown in Table 2.

After that, navigating to visdom address (*default - localhost:8097*) will bring up a web dashboard. We use main window (4b) to compare data from different runs like - average data, computation speed, used actions, game length, entropy and so on. If we are interested to see more detailed view of a certain run, we can switch to it from the top left corner - for each experiment there is a separate window named after a log file. On the left side of this window (4b) there is a pane listing all of the experiment parameters, including a link to source code which generated the results. There are also gradually recorded videos of agent's play.

Some logged data, needs to be rendered as video files. Openai Gym provides a way to record a game played by the agent, however the agent usually sees preprocessed frames. It is sometimes very useful to see this video itself, because maybe some important details in the original input frames are lost during preprocessing. We log array of exact states agent encounters and then in visualization script they are rendered as a video file. We also render the value estimate from the network, action distributions and one convolutional filter layer outputs (middle video on a bottom row of figure 4b). They are useful because if a particular action of an agent is strange, we can pause the video and see why exactly agent choose that action, or what other options it was considering.

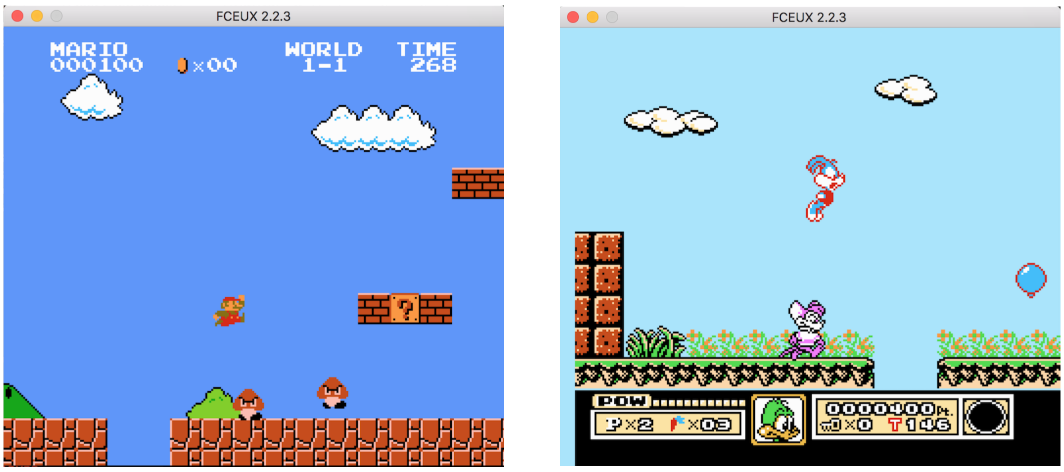
4.3 Trying to decouple the game scene using Tagger

In section 3 we reviewed some observations about how humans perceive objects and how understanding a relationship between them might be very beneficial for future AI algorithms. Although, as we saw in Chapter 2 DQNs can learn such relations to some extent, it can be argued that this knowledge only exists on pixel level and it is not

Argument	**Description**
–env	Gym environment name to visualize e.g ’Pong-v0’.
–dbdir	Location where environment database logs are located (mounted).
–heavy-ids	List of log ids needed to visualize which take a long time to render. (i.e converting to mpeg).
–env-count	Number of last database logs to read from folder and simultaneously visualize.
–max-steps	Maximum number of steps for each logs to read.

Table 2. Argument list for dashboard

clear how to generalize or use it in other tasks. Consider an example on Figure 5. In both games agents have to jump over, or jump on the enemies. There are also pits and obstacles. It is safe to say that if a child knows how to play the first game, he or she will not have much trouble understanding the other one. But in our experiments while training with A3C algorithm (subsubsection 2.4.2) the network did not benefit much from learning the first game before. (Note: because NES game episodes last long, we only trained on a small part of the game).



(a) NES game - Super Mario Bros. (b) NES game - Tiny Toon Adventures

Figure 5. Two visually different but conceptually similar game scenarios.

In addition there is a general problem in Deep Learning known as catastrophic forgetting ([KPR+17]). After the agent learns a second game, the first one is totally forgotten. Even recent network architecture modifications struggle to have significant progress in using experiences from previous games ([FBB+17]).

As we see, objects and background on these pictures visually look different, but semantically they are very similar. It is worth to look for approaches where there is a separate network which first tries to decouple the scene, and separates object representations from their structure. Then there is a separate network which tries to learn dynamics of this structure. If done so there might be a chance to incorporating progress in symbolic or analogical reasoning (SME framework [FFLG16]). Also learning "intuitive physics" (subsection 3.5) should probably become easier.

However this is easier said than done. Should flowers or clouds be a separate object in 5b? Probably no, but balloons on the other hand can be picked up, so it should be! Also what to do about the ground?

Next, we will try to investigate the possibility of using one unsupervised perceptual inference method for this task on one Atari game.

4.3.1 Brief review of Tagger framework

Authors of Tagger [GRB$^+$16] recently proposed an iterative inference for perceptual grouping, called *iTerative Amortized Grouping (TAG).* The goal is to make a neural network separate its input into K different groups. The model also needs to learn representation of each individual group. Thus network needs to make an inference on two sets of variables - first, the discrete random variables for each element of input to denote in which group it belongs and, second, the reconstructed representation for each group.

The method is completely unsupervised as amortizing posterior inference happens via the task to denoise the corrupted input. For an intuition, training starts with some probabilities of group assignments and reconstructions of each group. Over the iterations this estimates are refined by parameter mapping - a neural network (ladder [RBH$^+$15]) which in addition to these values takes a corrupted input and tries to improve these estimates so that denoising the input becomes easier. It is interesting that the network does not know anything about image segmentation, it learns to do so because it is beneficial for the to model to learn representations separately. Although K is fixed, Tagger significantly outperformed convolutional network in constructed 2 digit MNIST test. These digits had cluttered textures and were overlapping each other, but the network achieved surprisingly good results [GRB$^+$16].

4.3.2 Applying Tagger on image data generated by a game

It was interesting to see how tagger would group objects in Atari games. We used original source code of Tagger for training and evaluation published with the paper. We took 40 000 (42 X 42 greyscale) frames recorded by A3C agent on playing Atari game *KungFuMaster-v0*. This game was chosen for specific purposes, because of computation limits we could not afford number of groups in Tagger -K to be big, In this game the character fights with enemies from right and left side, so natural number of grouping would be 4 - background, character, left enemies, right enemies. After trying out several parameters for the network, we chose the ladder encoder projection to be - $(2000, 1000, 500)$, Gaussian noise of 0.1, batch size - 100, learning rate - 0.0004, number of epochs - 250. Training wall time for this parameters was around 11 hours on NVIDIA Tesla K20 GPU.

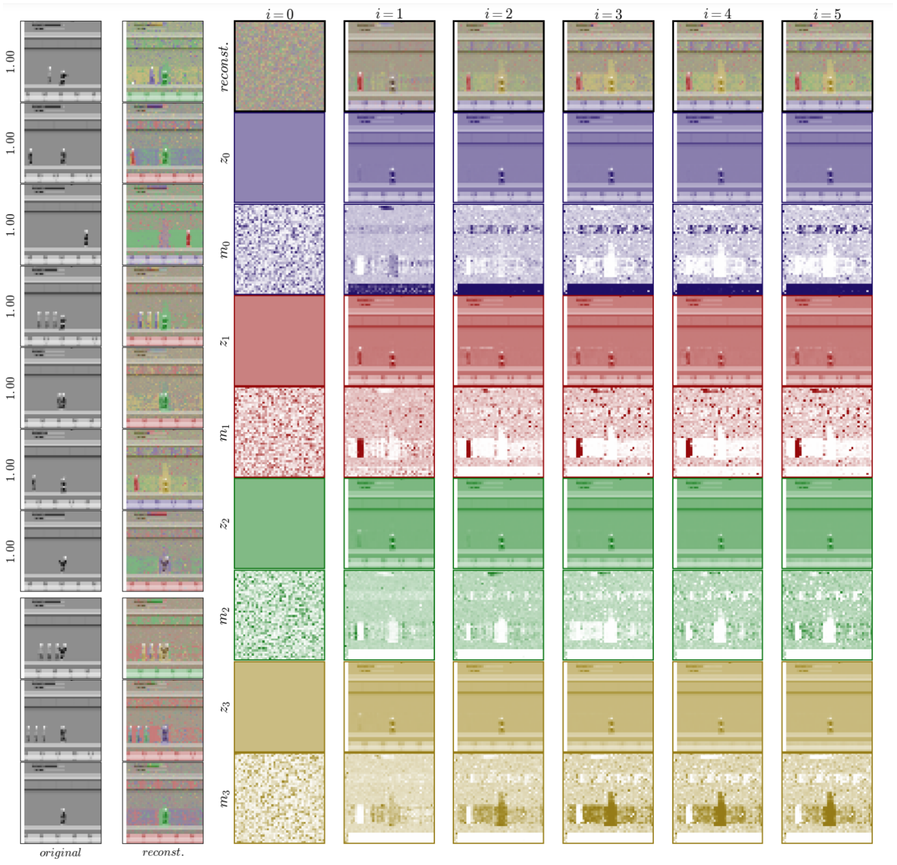


Figure 6. Tagger framework applied on images generated by Atari KungFuMaster game

In the left two column of Figure 6 there are original frames (collected by the agent) and their grouping visualized by different colors. In the following columns there is an iteration process $(1 - 5)$ shown, which is done by the network for one concrete video frame. First row shows the full reconstruction of the image by grouping. Next rows show, mask -m_i (probabilities that the pixel belongs to that group) and visual reconstruction of that specific group z_i. We see that the first grouping - m_0 is a floor and ceiling representation on the image. m_1 separates the left character, m_2 probably takes responsibility for some part of the background. And the last group represents the character itself.

In general the network does a pretty good job, but sometimes there seems to be unwanted groupings. For example, in the image where a character and enemy are close to each other, they are processed as a whole. The next steps we planned for our experiments, was to train an another agent where states for it would be these generated masks. Unfortunately, processing one frame takes fair computational time (10s of seconds) and RL algorithms needs to be fast at processing them. One option would be to think about tagger modification, to make it more effective for RL, but we chose to temporarily abandon this idea and try something else.

4.4 Experiments with A3C

We first tried to run several Atari games for testing A3C algorithm and a dashboard. Most of the experiments were run on high frequency Intel Xeon E5 (16-core) cpus.

One observation based on recorded videos was that, in some games the agent can get relatively high score within an hour, but might need 3-4 hours more to improve it by a little. From intermediate video recordings of *Pong-v0*, it seems that when the agent looses the ball it does everything right according to its strategy. After hitting the ball the agent goes somewhere in the side, where it **waits** to kick the ball from sideways to increase the speed and unpredictability. But sometimes it misses the last moment and is just a little bit late. We think that it can be related to stochastic nature of an environment which can make a speed of a ball non uniform.

4.5 Trying to increase data efficiency

The main idea of NEC [PUS$^+$17] discussed in 2.4 was to effectively use past episodic experiences. We were interested to try the same thing in A3C. Unfortunately A3C, unlike Q-learning used in NEC is an on-policy algorithm, meaning that if one is not careful while updating the policy, an agent might learn a bad behavior from which it can not recover.

Nevertheless, we thought whether one can increase the data efficiency and learn effective policies faster. To take the Pong's example, the agent does not get any explicit reward when it hits the ball. It only gets one if an enemy cannot bounce it back. But since in-built opponent plays rather well, the agent has to wait long time before sparsely won points change the policy parameters little by little, so it learns that hitting a ball is a beneficial action.

We decided to speed up this process by generating artificial intermediate rewards. As we saw in a Chapter 2, A3C uses an n-step value estimates, so an episode reward in advantage function is already biased. Thus, our generated rewards during training might further increase this bias. To make it less damaging, we do not generate the artificial rewards at every steps, but we do it randomly (in our experiments 2% times).

This reward is generated by the following logic:

First, We keep the experiences an agent encounters - triplets of (h, a, r_n) in p different tables ($T_{a_1}, T_{a_2} \cdots , T_{a_p}$), where p is the number of possible actions. h is some embedding of a state - we chose it to be the concatenation of LSTM hidden and cell units. r_n is an n step value estimate from the critic (it was set to 20). We keep the maximum size of a table and when the table is full, we delete it and start rebuilding a new one. This has some advantages, agent's episodic memory is refreshed by more recent and better experiences after it learned for some time. But, on the other hand, this increases computation and as during the time this table is not filled sufficiently, new artificial rewards can not be generated.

As we already saw, in A3C advantage calculated as:

$$A(s, a) = R_t - V(s)$$

Now, we add an additional reward terms x while calculating R_t:

$$R_t = r_t + x_t + \gamma(r_{t+1} + x_{t+1}) + \ldots + \gamma^2(r_{t+n-1} + x_{t+n-1}) + \gamma^{n-1}Q(s_{t+n}, a)$$

These x s are generated by weighted (distance) sum of k nearest neighbors of (h_t) and their corresponding rewards.(we used $k = 30$). h_t is an embedding of a state before getting reward r_t.

Because these experience tables are large, in practice we used library faiss [Fac] by Facebook research. This library provides a way for an efficient similarity queries on high dimensional vectors. Because it is better if the dimensionality of a vector is low, we chose to reduce the number of hidden units in a recurrent network. We used L2 distance for similarity metric.

Effect between choosing size of 256 and 64 can be seen on figure 7. On the x axis there is a number of steps, and on the y axis an average score (over 3 runs) achieved by the agent. Maximum score in Pong is 20, minimum: -20. As we see there is a decreased performance.

We then took these two runs as a baseline solution and compared them with our modified A3C algorithm with experience tables. We set a hidden layer size to 64 and limit of records in a table 100 000 and 10 000. These runs are visualized in 8. Both of these modification (ep-100K and ep-10K) used 64 hidden units. As we see our modification of A3C *ep-100k-mem* in this game outperforms a baseline solution with 64 hidden units and performs similarly with a baseline solution which uses 256 hidden units. Figure 9 shows entropy comparison of action distributions during training. However, it should be investigated more where this improvement comes from and what happens on other games.

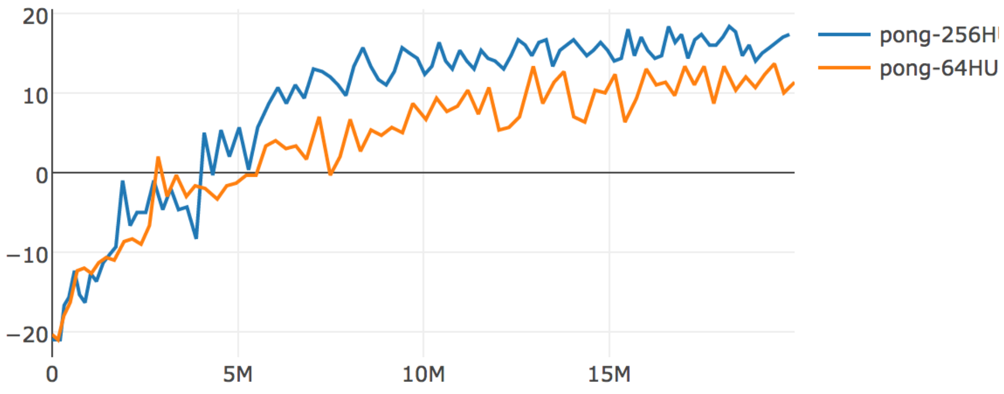


Figure 7. Comparison between 256 and 64 hidden units, on a Pong-v0 game

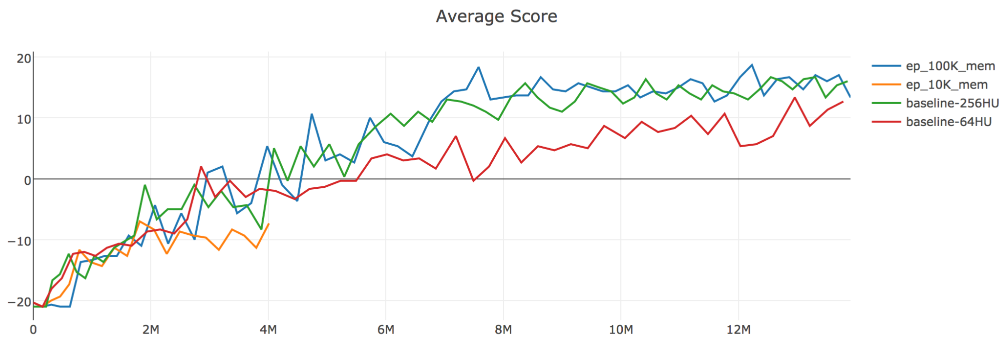


Figure 8. Score comparison

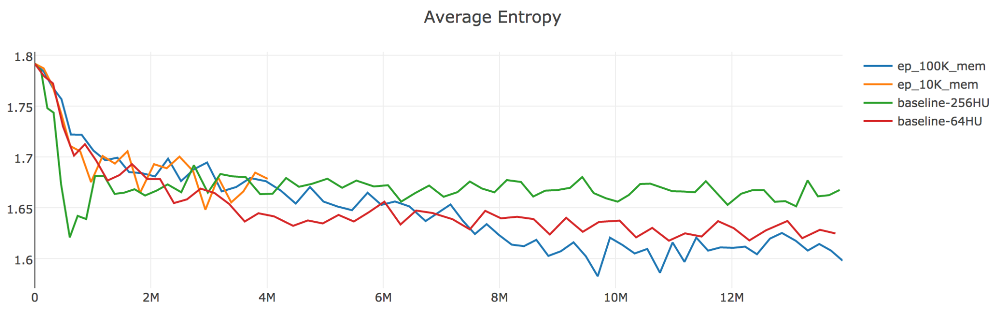


Figure 9. Entropy comparison

5 Discussions

Unlike other science fields, where at least some initial directions and formal methodologies are known, the correct way for approaching AGI is still a mystery even for researchers on the field. Thus, before committing to the specific branch of research, it should be beneficial to get a good overview of what has been done and what is known about the inner workings of human mind so far. The interdisciplinary field of cognitive science provides a promising way to do so. Since, there have been many books titled like "How to create a mind" [Kur12], maybe there already even exists a published cognitive architecture, unknown to researchers which would lead to successful AGI. The work done during writing of this thesis has been only a small step towards this review.

It might be argued that the cognitive modeling approach of AI can be misleading, just like trying to invent an airplane by observing birds. But ultimately, if there is an AGI, we will need to communicate with it and it will be easier if we work the similar way. Somewhat surprisingly it turns out that the current convolutional neural networks perform visual object recognition quite similarly to the human visual system [KVP+17]. Hence, there are anyhow similarities between the modern AI systems and the biological brains.

We reviewed RL algorithms because of their fundamental idea to learn from the interaction with the environment. So far, gaming platforms still seem to be the good ways to experiment with new ideas. Thus no matter which direction we choose, our dashboard will be useful for later research [sci].

Our primary motivation to improve the current RL algorithms was by a model building view of the world [LUTG16]. In particular, this approach suggests to explain the observed data by physical and psychological relationships between objects. We decided to try tagger [GRB+16] for games, because we think that some intermediate representation of visual perception is needed, because higher level knowledge of game dynamic gets blended with pixel patterns and it becomes hard to extract later. Unfortunately we encountered several problems when experimenting with tagger 1) Training time does not scale well with the input dimension and number of grouping - this makes it hard to experiment with games that need higher resolution. 2) Tagger has a very good convergence speed on test samples, but it is still slow for live RL training. If an agent takes 10 seconds for grouping the scene before making an action it cannot process millions of frames. 3) Tagger grouping seems to be good for many images, however for some, we had unwanted inference. It seems instead of only static images, tagger needs more context, like motion. We plan to investigate the possibility of fixing these 3 issues in the near future.

We also analyzed a game recording of a child to see how he would explore the environment and learn to navigate. Based on observations we think rewards from game emulators will soon be an obsolete part and agents in such games should exhibit some level of curiosity and intelligent navigation. This time we only had 9 episode game play of one player, but later we intend to do similar study on many subjects, where we will systematically analyze location trajectories and actions tried by children unaware of game rules. In general, we also think that the study of cognitive development is important for getting machines to learn like humans.

Finally, we experimented policy gradient method to make it data efficient and use an episodic memory. We used the k-nearest neighbors search to extract similar states from past experience and based on them generate artificial rewards for the agent, Initial results look promising but need further evaluation on more challenging environments. Also, in this experiment we completely deleted tables and started building new ones, when they reached their maximum allowed memory, it is interesting to try what happens when the least used records are replaced by the new one, as done by NEC [PUS$^+$17].

References

[AF07] George A Alvarez and Steven L Franconeri. How many objects can you track?: Evidence for a resource-limited attentive tracking mechanism. *Journal of vision*, 7(13):14–14, 2007.

[ale] alecthomas. voluptuous - voluptuous, despite the name, is a python data validation library. `https://github.com/alecthomas/voluptuous`. Accessed: 2017-05-01.

[BCP$^+$16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[BNVB13] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, jun 2013.

[Buc05] Bruce G Buchanan. A (very) brief history of artificial intelligence. *Ai Magazine*, 26(4):53, 2005.

[BUP$^+$16] Charles Blundell, Benigno Uria, Alexander Pritzel, Yazhe Li, Avraham Ruderman, Joel Z Leibo, Jack Rae, Daan Wierstra, and Demis Hassabis. Model-free episodic control, 2016.

[Cli] Clipaart. A source for free clipart, clip art pictures and illustrations. `http://classroomclipart.com`. Accessed: 2017-05-01.

[DDRD01] Sašo Džeroski, Luc De Raedt, and Kurt Driessens. Relational reinforcement learning. *Machine learning*, 43(1-2):7–52, 2001.

[DLDHP02] Ghislaine Dehaene-Lambertz, Stanislas Dehaene, and Lucie Hertz-Pannier. Functional neuroimaging of speech perception in infants. *science*, 298(5600):2013–2015, 2002.

[Dra] Drawio. Flowchart maker and online diagram software. `http://draw.io`. Accessed: 2017-05-01.

[DT13] Nathaniel D Daw and Philippe N Tobler. Value learning through reinforcement: the basics of dopamine and reinforcement learning. *Neuroeconomics,*, pages 283–298, 2013.

[Fac] Facebook. A library for efficient similarity search and clustering of dense vectors. `https://github.com/facebookresearch/faiss`. Accessed: 2017-05-01.

[FBB+17] Chrisantha Fernando, Dylan Banarse, Charles Blundell, Yori Zwols, David Ha, Andrei A Rusu, Alexander Pritzel, and Daan Wierstra. Pathnet: Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv:1701.08734*, 2017.

[FFLG16] Kenneth D Forbus, Ronald W Ferguson, Andrew Lovett, and Dedre Gentner. Extending sme to handle large-scale cognitive modeling. *Cognitive Science*, 2016.

[Fod75] Jerry A Fodor. *The language of thought*, volume 5. Harvard University Press, 1975.

[Gha15] Zoubin Ghahramani. Probabilistic machine learning and artificial intelligence. *Nature*, 521(7553):452–459, 2015.

[GRB+16] Klaus Greff, Antti Rasmus, Mathias Berglund, Tele Hao, Harri Valpola, and Juergen Schmidhuber. Tagger: Deep unsupervised perceptual grouping. In *Advances in Neural Information Processing Systems*, pages 4484–4492, 2016.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[iko] ikostrikov. Baseline implementation of a3c in pytorch. `https://github.com/ikostrikov/pytorch-a3c`. Accessed: 2017-05-01.

[Inc15] Plotly Technologies Inc. Collaborative data science, 2015.

[KB14] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[KPR+17] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, page 201611835, 2017.

[Kur12] Ray Kurzweil. *How to create a mind: The secret of human thought revealed.* Penguin, 2012.

[KVP+17] Ilya Kuzovkin, Raul Vicente, Mathilde Petton, Jean-Philippe Lachaux, Monica Baciu, Philippe Kahane, Sylvain Rheims, Juan R Vidal, and Jaan Aru. Frequency-resolved correlates of visual object recognition in human brain revealed by deep convolutional neural networks. *bioRxiv*, page 133694, 2017.

[LD08] Máté Lengyel and Peter Dayan. Hippocampal contributions to control: The third way. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 889–896. Curran Associates, Inc., 2008.

[Li17] Yuxi Li. Deep reinforcement learning: An overview, 2017.

[Lin93] Long-Ji Lin. *Reinforcement learning for robots using neural networks*. PhD thesis, Fujitsu Laboratories Ltd, 1993.

[LUTG16] Brenden M. Lake, Tomer D. Ullman, Joshua B. Tenenbaum, and Samuel J. Gershman. Building machines that learn and think like people, 2016.

[MBM$^+$16] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. 2016.

[MJB15] Tomas Mikolov, Armand Joulin, and Marco Baroni. A roadmap towards machine intelligence. *arXiv preprint arXiv:1511.08130*, 2015.

[MKS$^+$13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[Nin] Nintendo. Nintendo entertainment system. `https://en.wikipedia.org/wiki/Nintendo_Entertainment_System`. Accessed: 2017-05-01.

[Niv09] Yael Niv. Reinforcement learning in the brain. *Journal of Mathematical Psychology*, 53(3):139–154, 2009.

[NSB$^+$15] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015.

[NSS59] Allen Newell, John C Shaw, and Herbert A Simon. Report on a general problem solving program. In *IFIP congress*, volume 256, page 64. Pittsburgh, PA, 1959.

[Num] Numpy. Numpy is the fundamental package needed for scientific computing with python. `https://github.com/numpy/numpy`. Accessed: 2017-05-01.

[PUS$^+$17] Alexander Pritzel, Benigno Uria, Sriram Srinivasan, Adrià Puigdomènech, Oriol Vinyals, Demis Hassabis, Daan Wierstra, and Charles Blundell. Neural episodic control, 2017.

[Pyt] Pytorch. Pytorch, tensors and dynamic neural networks in python with strong gpu acceleration. `http://pytorch.org`. Accessed: 2017-05-01.

[RBH$^+$15] Antti Rasmus, Mathias Berglund, Mikko Honkala, Harri Valpola, and Tapani Raiko. Semi-supervised learning with ladder networks. In *Advances in Neural Information Processing Systems*, pages 3546–3554, 2015.

[RN95] Stuart Russell and Peter Norvig. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25:27, 1995.

[RRWN11] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[Sch10] Jürgen Schmidhuber. Formal theory of creativity, fun, and intrinsic motivation (1990–2010). *IEEE Transactions on Autonomous Mental Development*, 2(3):230–247, 2010.

[sci] scientist1642. Main repositroy for our experiments. `https://github.com/scientist1642/bombora`. Accessed: 2017-05-01.

[SQL] SQLite. Sqlite is a self-contained, high-reliability, embedded, full-featured, public-domain, sql database engine. `https://www.sqlite.org`. Accessed: 2017-05-01.

[Ten] Tensorboard. Tensorboard: Visualizing learning. `https://www.tensorflow.org/get_started/summaries_and_tensorboard`. Accessed: 2017-05-01.

[Tes94] Gerald Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, 6(2):215–219, 1994.

[TKGG11] Joshua B Tenenbaum, Charles Kemp, Thomas L Griffiths, and Noah D Goodman. How to grow a mind: Statistics, structure, and abstraction. *science*, 331(6022):1279–1285, 2011.

[TVDR$^+$12] Elena Tomasuolo, Giovanni Valeri, Alessio Di Renzo, Patrizio Pasqualetti, and Virginia Volterra. Deaf children attending different school environments: Sign language abilities and theory of mind. *Journal of deaf studies and deaf education*, page ens035, 2012.

[Vis] Visdom. A flexible tool for creating, organizing, and sharing visualizations of live, rich data. supports torch and numpy. `https://github.com/facebookresearch/visdom`. Accessed: 2017-05-01.

[WD92] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[Wik] Wikipedia. Blue brain project. `https://en.wikipedia.org/w/index.php?title=Blue_Brain_Project&oldid=779397472`. Accessed: 2017-05-01.

[ZBZM16] Tom Zahavy, Nir Ben-Zrihem, and Shie Mannor. Graying the black box: Understanding dqns. *arXiv preprint arXiv:1602.02658*, 2016.

Appendix

I. Dashboard

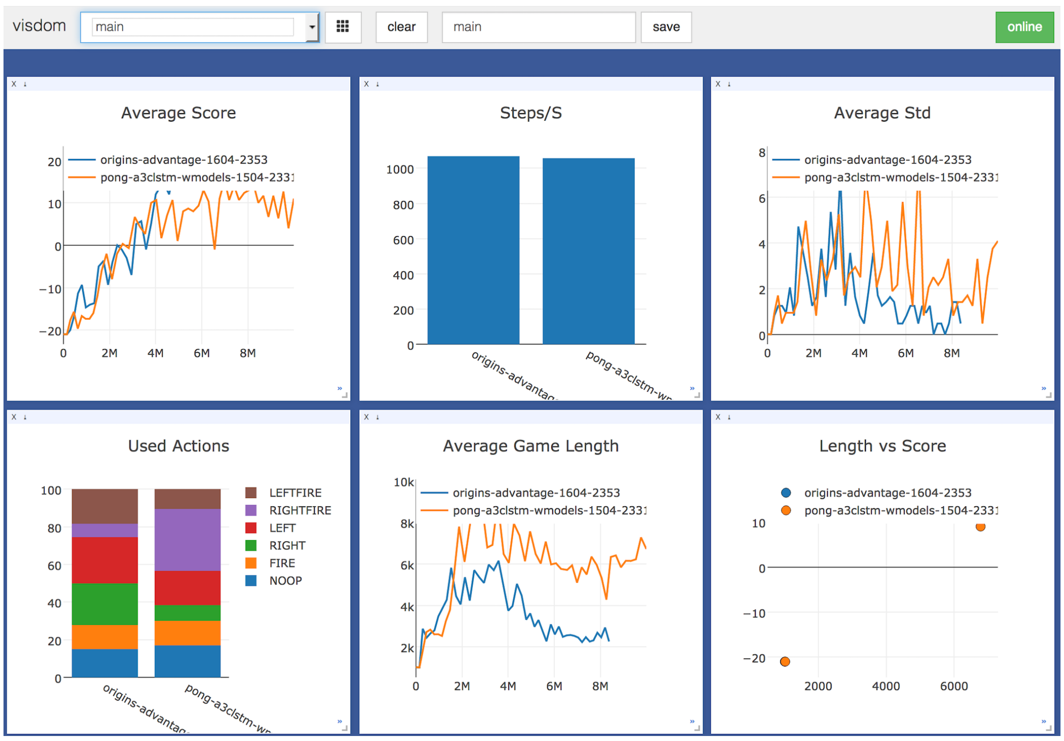
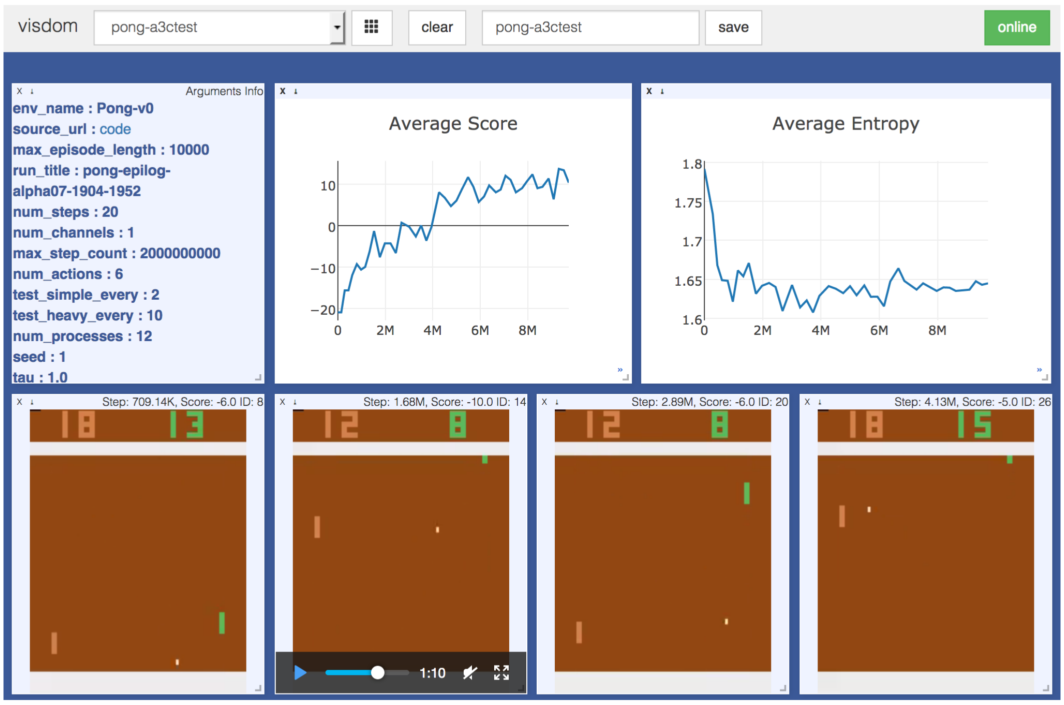


Figure 10. Main view of a dashboard for two environments



```
env_name : Pong-v0
source_url : code
max_episode_length : 10000
run_title : pong-epilog-alpha07-1904-1952
num_steps : 20
num_channels : 1
max_step_count : 2000000000
num_actions : 6
test_simple_every : 2
test_heavy_every : 10
num_processes : 12
seed : 1
tau : 1.0
```

Step: 709.14K, Score: -6.0 ID: 8

Step: 1.68M, Score: -10.0 ID: 14

Step: 2.89M, Score: -6.0 ID: 20

Step: 4.13M, Score: -5.0 ID: 26

Figure 11. Specific window of the environment

II. Source code

All our software implementations can be found on github at `http://github.com/scientist1642/bombora`. Particularly, to reproduce the A3C experiment discussed in subsection 4.5, one can checkout the *epilog* branch and run the program with following parameters:

```
python main.py --lr 0.0001 --gamma 0.99 --tau 1.0 --seed 1 --num-processes 15 --num-steps 20 --max-
    episode-length 10000 --env-name Pong-v0 --no-shared false --debug False --algo epilog --arch
    lstm_universe --num-test-episodes 3 --test-simple-every 1 ----test-heavy-every 20 --hidden-size 64 -
    -episodic-every 50
```

After running the following command, dashboard can be seen by navigating to `http://localhost:8097`. Recommended browsers are *Firefox* and *Chrome*:

```
python dashboard.py --dbdir dblogs --env 'Pong-v0'
```

III. Licence

Non-exclusive licence to reproduce thesis and make thesis public