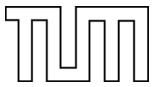


Technische Universität München

Fakultät für Mathematik

TUM Data Innovation Lab

End-to-End Learning Artificial Intelligence

Project Documentation

Egor Labintcev
Haris Jabbar
Adrian Sieler
Christian Holland

Supervision: M.Sc Matthias Wissel, Dr. Ricardo Acevedo
Examination: Prof. Dr. Massimo Fornasier

Contents

1 Introduction

1.1 General Setting of Reinforcement Learning

The task of reinforcement learning is to develop an algorithm, i.e. agent, which can solve a specific environment by learning from it. In this case, solving means that the agent is able to maximize some reward r_t in a long run. The learning process includes observation of the state s_t of the environment and reception of corresponding reward after performing an action a_t. The environment could be affected by the action of the agent or could not.

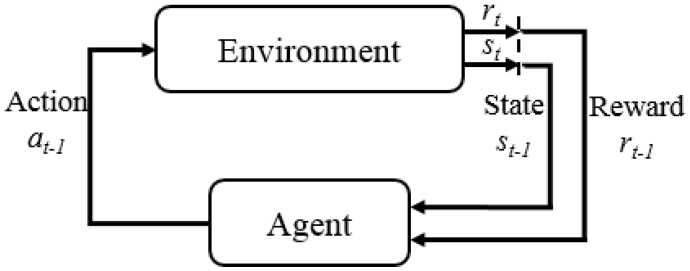


Figure 1: General reinforcement learning setting

This setting encourages the agent to select the best (in terms of future reward) action on each step. It also means that the agent may not have any prior information about the environment and it operates with given state and reward pairs only. The typical machine learning exploitation/exploration dilemma is present here and it is crucial for the agent to find balance between those activities.

Reinforcement learning can be formulated using Markov decision process (MDP) setting. MDP is defined by tuple

$$(S, A, P.(\cdot,\cdot), r.(\cdot,\cdot), p_0),$$

where S and A represent spaces of states and actions respectively, P is a transition function $P_k(j,i) = p(s_{t+1} = i \mid s_t = j, a_t = k)$, r stays for a reward function $r : S \times A \longrightarrow R$, and p_0 is an initial distribution of states. We assume that reinforcement learning process has the Markov property, i.e. future state of the process depends only on the current state.

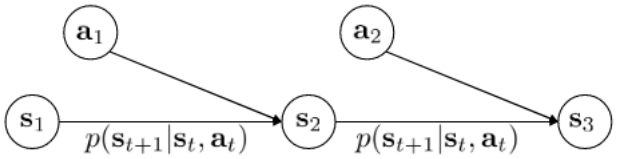


Figure 2: Markov Property

The core problem for MDP is to find a (parameterized) policy function π_θ, which returns the action a when the agent is in the state s. This optimization problem could be formulated as

$$\underbrace{p_\theta(s_1, a_1, ..., s_T, a_T)}_{p_\theta(\tau)} = p_0(s_1) \prod_{t=1}^{T} \pi_\theta(a_t \mid s_t) p(s_{t+1} \mid s_t, a_t),$$

where θ acts as the parameter of the policy, which maximizes the sum of the rewards

$$\theta^\star = argmax_\theta \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \Big[\sum_t r(s_t, a_t) \Big].$$

With that, one can define Q-function, which represents the total reward from taking a_t in s_t and continue with policy π_θ afterwards

$$Q^{\pi}(s_t, a_t) = \sum_{t\prime=t}^{T} \mathbb{E}_{\pi_\theta}[r(s'_t, a'_t) \mid s_t, a_t]$$

and value function, which defines the reward from s_t while following the policy π

$$V^{\pi}(s_t) = \mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)}[Q^{\pi}(s_t, a_t)].$$

Using Q-function and value function we can formulate advantage function, which states for the difference in taking the action a_t in the state s_t from average return in s_t

$$A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t).$$

1.2 Applications

Due to the generality of the formulation of the optimization problem, reinforcement learning finds its application in many areas, where the problem of learning process can be formulated as interaction between environment and agent. Self-driving cars, robotics and dialog systems can be represented via this setting. For instance, in dialog systems one could use reinforcement learning definitions to setup dialog [9] for two agents, where both of them try to receive reward, which is represented by semantic coherence of answers and information flow, i.e. existence of evolution of respected dialog. Furthermore, there are some classical problems with time dependency such as time series prediction, which can be formulated as reinforcement learning problem. Also optimal control problems can be described in terms of reinforcement learning, e.g. temperature control or energy consumption optimization. We consider one of the control problems in Section 6.

1.3 Objectives of the Project

The main idea of this project was to set up a distributed scalable framework for learning agents on various environments using state-of-the-art algorithms based on deep learning approach. Distributed way of computing can drastically speed up a learning process. Moreover, making the learning process scalable result in advantage of using multiple machines and environments. Same algorithms applied to the copies of one environment, but started with different initial states can enhance the exploration part of learning. Recent advances in developing algorithms (A3C [11], PCL [12]) for deep reinforcement learning rely on multiple agents, each of them can update master network and achieve overall accuracy faster.
Another advantage of a distributed framework is when the environment is itself so complex and computationally demanding, that it occupies all of a compute node. In such a case, having multiple copies of the environment in a scalable manner can linearly decrease the computation time.

1.4 Research

To find a suitable goal for our project we focused on the research at first. Here we presented the short overview of our findings. General task of achieving better and faster learning of the agent can be decomposed in such way:

- Optimizing on algorithms' level;
- Using additional information (transfer learning);
- Distributed way of computing.

1.4.1 Optimizing on Algorithms' Level

We started with exploring current state-of-the-art approaches and implementations. One can categorize the learning algorithms into three main classes: off-policy, on-policy and combined methods. Off-policy methods approximate the optimal Q* -function with learned Q-function independently

of the policy being followed, as they use state s_{t+1} and some greedy chosen action a_{t+1} as updates to the Q-values. Q-learning can be defined as

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

where α is the learning rate and γ is the discount reward factor ($0 \leq \gamma \leq 1$), which expresses the importance of sooner versus later rewards.
In contrast, on-policy methods use the action given by current policy, so they estimate state-action pairs assuming the current policy continues. The example to that class would be A3C (Asynchronous Advantage Actor-Critic algorithm), which we explain in detail in Section 3.1. Combined methods take advantages from both strategies and can learn either from policy given actions or as a value based function. We will consider Path Consistency Learning (PCL) algorithm, which can learn on- and off-policy.

1.4.2 Using additional information (transfer learning)

In general, transfer learning (also inductive learning) is the way to apply knowledge gained in solving one problem to another problem, which is related to solved one. It allows to reduce the scope of possible hypotheses to test and should help to find better or/and faster solution to the new problem.
In case of reinforcement learning we can transfer the model, which solved one environment to another. We expect that this transfer results in

- Higher start of learning, better quality at the start with some prior;
- Higher slope of learning, faster;
- Higher asymptote, better quality at the end of learning.

Reinforcement learning is one of the first areas of machine learning, where transfer learning was applied. However, in our chosen setting of OpenAI gym we didn't explore its capabilities.

1.4.3 Distributed Way of Computing

Reinforcement learning appears to be the one of the most difficult tasks of machine learning from computational point of view. Due to the large number of possible hyperparameters on each level of learning setting and temporary dependency of evolution of environments, it is hard to learn the given problem fast enough. But this problem can be solved by converting the algorithms to asynchronous ones, which can be run independently and send their updates to master network. Then the problem could be scaled and it results in faster learning. We adopted Distributed Tensorflow framework to our methods and we describe details in Section 4.

2 Frameworks and Tools

To enable the development of a reinforcement learning platform, several frameworks and tools are required. The *OpenAI Gym* as a simulation environment to various environments. *Distributed Tensorflow* as efficent framework to distribute the training of policy and value-function approximating neural networks. *ROS (Robot Operating System)* as foundation to implement a scalable representation of the environment and the *LRZ Cloud* to leverage a multi machine multi CPU architecture to enhance training speed.

2.1 OpenAI Gym

One of the core challenges of reinforcement learning is the handling of the environment. Typically, a large number of observations is needed to train an agent in a meaningful way, requiring large computational resources for simulation. Also the definition of "desired behaviour" and thus the reward generation may be unclear.

A good solution to both of these problems is provided by the open source library "OpenAI Gym"[17]. The Gym is meant to be a general toolkit for developing and comparing reinforcement algorithms. On the one hand, it provides simple to use and compute efficient environments, while on the other hand giving users the possibility to check their results by providing a common baseline for comparison.

One popular group of environments provided by the Gym are Atari games. It replaces the usual frame sequence of a game with a stepwise exchange of actions for observations (with given rewards), thus providing a functioning interface for the reinforcement learning loop as described in section 1.1. In particular, those observations are graphical, thus returning the exact picture of the TV-screen in every step.

Another useful group of environments that we used are the "classic control problems" which instead of a screen output, contain the state of the game in a different way, such as positions or angles allowing for tests with lower computational effort.

2.2 Distributed Tensorflow

The framework Tensorflow offers a native distributed functionality[10], allowing distributed computing without the use of a cluster computing framework like Apache Spark, saving possible overhead.

To use this, a cluster spec with a list of "servers" and their respective "tasks" has to be provided, assigning a client address and a communication port to each server. The servers itself can then be started on the respective machine with a Tensorflow command, handling all subsequent communication until it is terminated. The tasks are usually separated in worker tasks, which do computations and parameter tasks, which store the current model parameters.

There are two approaches for running the code:

1. **In-graph replication:** One client builds a Tensorflow dataflow graph and builds multiple copies of the computational part of the model which correspond to different workers.
2. **Between-graph replication:** Each client runs his own copy of the code building his own dataflow graph. Computational tasks are then done by each worker separately, but updating shared parameters. The parameter server usage logic is handled by Tensorflow's replica device setter.

Although, the latter might be the less intuitive approach, it has better performance and is the approach that is most commonly used and also what we used in our implementation.

2.3 LRZ Cloud

LRZ provides cloud services where the users can build their own templates and machine images for particular VMs. The cloud is built on OpenNebula framework and provides the users with a web based dashboard for all management functions. Additionally it also provides Amazon-EC API with which the users can manage the VMs, templates and images using linux command line or bash scripts. This feature is crucial for automatic provisioning of VMs. Another pertinent feature is cloud-init, with which the users can specify a script that can be run just after the VM is available. This can be used for installing and configuring software and setup of environment. Both these features were used in our deployment.

2.4 Robot Operating System (ROS)

Robot Operating System (ROS) [19] is an open source software infrastructure for developing heterogeneous robot systems and subsystems in a way that is both scalable and (to a large extent) agnostic to choice of programming language. This is achieved by encapsulation through programming snippets called 'nodes' and mechanisms for intra-node communication.

2.4.1 ROS Nodes

The nodes can represent the sensors, the compute logic or the actuators. They can be implemented in a wide array of programming languages and since the intra-node communication is achieved through ROS serialization services, the nodes can be of different programming languages and still work seamlessly with each other. ROS also provides handling of node failure and launching of nodes programmatically.

2.4.2 Intra Node Communication

There are two different types of intranode communication mechanisms. One implements the subscriber-publisher modality in which the publisher node sends out messages (called topics) at fixed intervals and any node can subscribe to these topics. This mechanism is good for sensors. The second modality involves a client-server architecture; where a server node offers a 'service'. A client can then query this service and get back a response from the server. This follows a synchronous communication paradigm.
In our implementation, we used only client-server architecture as it results in reduced communication overhead, which is vital to achieve a compute-vs-communicate ratio favorable for use on a CPU/GPU. However there can be scenarios when the publisher-subscriber model is more suitable. (See 8.3)

3 Mathematical Details of the Implemented Algorithms

In our research, as mentioned in Section 1, we focused on scalable reinforcement learning methods in order to tackle a given RL challenge with any available amount of computation power. So we agreed to implement the well-known *A3C* algorithm [11] and the relatively recent *Path-Consistency-Learning (PCL)* algorithm [12]. Both algorithms can be run asynchronously and are ready to be scaled to an arbitrary amount of workers.
The training of a reinforcement learning agent is always a trade-off between exploration and exploitation. The agent should visit states which yield a high reward frequently but should also explore the environment in order to find even more fruitful states. Therefore two different exploration methods are introduced. The first extends the A3C loss function with an *entropy* [6][11] term to encourage the policy to maximize the reward while keeping some uncertainty when choosing the actions. Another approach is to add parametric noise to some layers of the neural network representing the policy and value function, this method is called *Noisy-Nets* [3].
In the end of this section we present a tweak, called *Generalized Advantage Estimation* [20], to the loss function of the A3C algorithm to reduce the variance of the policy gradient and a heuristic approach to adaptively change the learning rate of the optimizer based on the *Kullback–Leibler divergence (KL-Divergence)* [22][21] of the current policy and the policy before the last update step.

3.1 Asynchronous Advantage Actor-Aritic - A3C

The A3C algorithm is based on the REINFORCE [11] and the actor-critic-algorithm, two policy-gradient model-free methods. In model-free reinforcement learning one does not assume anything about the underlying transition function (see Section 1.1). The goal is to find a good representation of the policy function $\pi_\theta(a \mid s)$ by interacting with the environment.

3.1.1 REINFORCE-Algorithm

The goal of the REINFORCE algorithm is to find a parameterized policy π_θ which maximizes the expected sum of the rewards for all possible rollouts τ, see Section 1.1 for details,

$$\theta^\star = argmax_\theta \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\Big[\sum_{t=1}^{T} r(s_t, a_t)\Big].$$

This can be achieved by gradient ascent of $\mathcal{J}(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\Big[\sum_{t=1}^{T} r(s_t, a_t)\Big]$. The gradient can be written as (see [8])

$$\nabla_\theta \mathcal{J}(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\Big[\Big(\sum_{t=1}^{T} \nabla \log(\pi_\theta(a_t \mid s_t))\Big)\Big(\sum_{t=1}^{T} r(s_t, a_t)\Big)\Big].$$

Based on this result the REINFORCE algorithm looks like:

Algorithm 1 REINFORCE algorithm

1: Initialize random policy $\pi_\theta(a \mid s)$, learning rate α
2: **while** θ not converged **do**
3: Sample $\{\tau^i\}$ from $\pi_\theta(a \mid s)$ ▷ run the policy
4: Estimate $\nabla_\theta \mathcal{J}(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\Big(\sum_{t=1}^{T}\nabla\log(\pi_\theta(a_t^i \mid s_t^i))\Big)\Big(\sum_{t=1}^{T} r(s_t^i, a_t^i)\Big)$
5: Update $\theta \leftarrow \theta + \alpha\nabla_\theta \mathcal{J}(\theta)$
6: **end while**

3.1.2 Actor-Critic-Algorithm

Since rollouts τ_i of the environment are considered, the sum of our rewards $\sum_{t=1}^{T} r(s_t^i, a_t^i)$ can yield a large variance.

- This variance can be reduced by subtracting a learned function of the state $b_t(s_t)$ [11], known as the baseline, from the return $r(s_t^i, a_t^i)$. The resulting gradient of one sampled time-step is $\nabla log(\pi_\theta(a_t \mid s_t)(r(s_t, a_t) - b_t(s_t))$.
- A learned estimate of the value function is commonly used as the baseline $b_t(s_t) \approx V_t(s_t)$ leading to a much lower variance estimate of the policy gradient.
- In this case the quantity $r(s_t, a_t) - V_t(s_t)$ used to scale the policy gradient can be seen as an estimate of the advantage of action a_t at s_t, $A(a_t, s_t) = Q(s_t, a_t) - V(s_t)$, because $r(s_t, a_t)$ is an estimate of $Q^\pi(a_t, s_t)$ and b_t is an estimate of V^π.

Those model assumptions lead to a actor-critic architecture where the policy π_θ is the actor and the baseline V_t is the critic.

One can estimate the advantage function A even better by [8]

$$A^\pi(s_t, a_t) \approx r(s_t, a_t) + \gamma V^\pi(s_{t+1}) - V^\pi(s_t).$$

So we need to fit a representation parameterized by ϕ to the value function $V^\pi(s)$. This can be formulated as a regression problem:

- Training data: $\left\{ \left(s_t, \underbrace{r(s_t, a_t) + \gamma V_\phi^\pi(s_{t+1})}_{y_t} \right) \right\}$
- Loss function: $\mathcal{L}(\phi) = \frac{1}{2} \sum_t (V_\phi^\pi(s_t) - y_t)^2$

So we can formulate the actor-critic algorithm, pseudocode can be found in Appendix 9.1.

3.1.3 A3C-Algorithm

This algorithm can now be used to run multiple actor-learners in parallel on multiple or the same machine [11], where all update asynchronous the same global model. This allows us to use Hogwild! [15] training. There are several advantages in applying this multi-threaded asynchronous actor-critic version:

- One can explicitly use different exploration policies in each actor-learner to maximize diversity.
- By running different exploration policies in different threads, the overall changes being made to the parameters by multiple actor-learners applying online updates in parallel are likely to be less correlated in time than a single agent applying online updates.
- Also, a reduction in training time roughly linear in the number of parallel actor-learners is obtained.

Typically convolutional neural networks in combination with LSTM layers with one softmax output for the policy $\pi_\theta(a_t \mid s_t)$ and one linear output for the value function $V_{\theta_v}(s_t)$ with all non-output layers shared are applied, see Section 4.
A visual representation of the architecture is depicted in Figure 3 and the pseudocode of A3C can be found in Appendix 9.2.

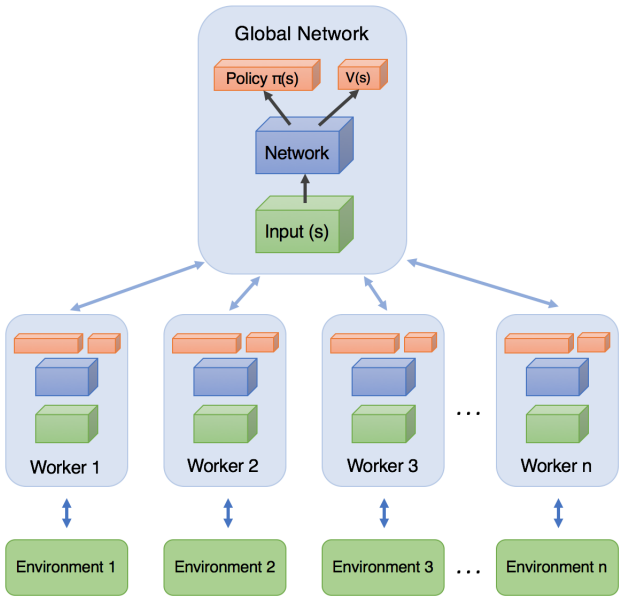


Figure 3: High level architecture of A3C

3.2 Path Consistency Learning - PCL

The biggest drawback of policy gradient methods like A3C is sample inefficiency, since policy gradients are estimated using online rollouts of the current policy, see Section 1.4.1. By contrast, value based methods, such as Q-learning, can learn from any trajectory sampled from the same environment. Such "off-policy" methods are able to exploit data from other sources, like experts, making them inherently more sample efficient than on-policy methods. Their key drawback is that off-policy learning does not stably interact with function approximation, due to changing nature of the training data [12].
Path Consistency Learning [12] aims to combine the unbiasedness and stability of on-policy training with the data efficiency of off-policy approaches. In the following we derive the objective and state an asynchronous version of this algorithm. Deterministic state dynamics and a discrete action are assumed for simplicity of derivation, in general those assumptions can be dropped, see [12].
One can state the expected discounted reward objective $\mathbb{E}_{\tau\sim\pi(\tau)}\left[\sum_t \gamma^t r(s_t, a_t)\right]$ recursively as

$$O_{ER}(s_t, \pi) = \sum_a \pi(a_t \mid s_t)[r(s_t, a_t) + \gamma O_{ER}(s_{t+1}, \pi)].$$

Let $V^*(s_t)$ denote the optimal state value at a state s_t given by the maximum value of $O_{ER}(s_t, \pi)$ over the space of policies, in particular $V^*(s_t) = max_\pi O_{ER}(s, \pi)$ and $\pi^* = argmax_\pi O_{ER}(s, \pi)$. Such an policy is a one-hot distribution that assigns a probability of 1 to an action with maximal return and 0 elsewhere. Thus we have

$$V^*(s_t) = O_{ER}(s, \pi^*) = max_a(r(s_t, a_t) + \gamma V^*(s_{t+1})).$$

This is the well-known hard-max Bellman temporal consistency. Instead of state values, one can equivalently and more commonly express this consistency in terms of optimal action values Q^*:

$$Q^*(s_t, a_t) = r(s_t, a_t) + \gamma max_{a_{t+1}} Q*(s_{t+1}, a_{t+1}).$$

Q-learning relies on a value iteration algorithm based on this consistency, where $Q(s_t, a_t)$ is bootstrapped based on successor action values $Q(s_{t+1}, a_{t+1})$.

3.2.1 Softmax Temporal Consistency

To drop the assumption of a hard-max Bellman temporal consistency for optimal state and action values, a softmax form of temporal consistency is introduced [12]. Which arises by augmenting the expected reward objective with a discounted entropy regularizer. This kind of entropy regularization greedily maximize entropy at the current time step, but does not explicitly optimize for policies that aim to reach states where they will have high entropy in the future as in Section 3.3, for details see [6].

$$O_{ENT}(s_t, \pi) = O_{ER}(s_t, \pi) + \tau \mathcal{H}(s_t, \pi)$$

Where $\tau \geq 0$ is a temperature parameter that controls the degree of entropy regularization. The discounted entropy $\mathcal{H}(s_t, \pi)$ is recursively defined as

$$\mathcal{H}(s_t, \pi) = \sum_a \pi(a_t \mid s_t)[-\log \pi(a_t \mid s_t) + \gamma \mathcal{H}(s_{t+1}, \pi)].$$

Therefore our objective $O_{ENT}(s_t, \pi)$ can be expressed recursively as,

$$O_{ENT}(s_t, \pi) = \sum_a \pi(a_t \mid s_t)[r(s_t, a_t) - \tau \log \pi(a_t \mid s_t) + \gamma O_{ENT}(s_{t+1}, \pi)].$$

Let $V^*(s_t) = max_\pi O_{ENT}(s_t, \pi)$ and $\pi^*(a_t, s_t) = argmax_\pi O_{ER}(s_t, \pi)$ denote the optimal policy. When $\tau > 0$, the optimal policy is no longer a one-hot distribution, since the entropy term prefers the use of policies with more uncertainty.
So the optimal policy $\pi^*(a_t, s_t)$ in terms of the O_{ENT} optimal state values of successor states $V^*(s_{t+1})$ can be characterized as a Boltzman distribution of the form

$$\pi^*(a_t, s_t) \sim \exp\{(r(s_t, a_t) + \gamma V^*(s_{t+1}))/\tau\}.$$

To derive $V^*(s_t)$ in terms of $V^*(s_{t+1})$, the policy $\pi^*(a_t, s_t)$ can be substituted into the expected reward objective with a discounted entropy regularizer, which after some manipulation yields the definition of optimal state value in terms of a softmax (i.e. log-sum-exp) backup,

$$V^*(s_t) = O_{ENT}(s_t, \pi^*) = \tau \log \sum_a \exp\{(r(s_t, a_t) + \gamma V^*(s_{t+1}))/\tau\}.$$

3.2.2 Consistency Between Optimal Value & Policy

The quantity $\exp\{V^*(s_t)/\tau\}$ also serves as the normalization factor of the optimal policy $\pi^*(a_t, s_t)$, that is

$$\pi^*(a_t, s_t) = \frac{\exp\{(r(s_t, a_t) + \gamma V^*(s_{t+1}))/\tau\}}{\exp\{V^*(s_t)/\tau\}}.$$

With this definition of $\pi^*(a_t, s_t)$ in mind one can derive the following theorem:

Theorem 1 *For $\tau > 0$, the policy π^* that maximizes O_{ENT} and state values $V^*(s_t) = max_\pi O_{ENT}(s_t, \pi)$ satisfy the following temporal consistency property for any state s_t, action a_t and corresponding s_{t+1}*

$$V^*(s_t) - \gamma V^*(s_{t+1}) = r(s_t, a_t) - \tau \log \pi^*(a_t \mid s_t).$$

The proof can be found in [12]

An important property of this one-step softmax consistency is that it can be extended to multi-step consistency defined on any action sequence from any given state. That is, the softmax optimal state values at the beginning and end of any action sequence can be related to the rewards and optimal log-probabilities observed along the trajectory.

Theorem 2 *For $\tau > 0$, the optimal policy π^* and optimal state values $V^*(s_t)$ satisfy the extended temporal consistency property for any state s_1 and any action $a_1, \ldots, a_{t-1}$ and corresponding $s_2, \ldots, s_t$*

$$V^*(s_1) - \gamma^{t-1} V^*(s_t) = \sum_{i=1}^{t-1} \gamma^{i-1}[r(s_i, a_i) - \tau \log \pi^*(a_i \mid s_i)].$$

The proof can be found in [12].

3.2.3 Path Consistency Learning (PCL) Algorithm

The temporal consistency properties introduced above lead to a natural path-wise objective for training a parameterized policy π_θ and a parameterized value function V_ϕ via the minimization of a soft consistency error.
So we define a notion of soft consistency for a d-length sub-trajectory
$s_{t:t+d} \equiv (s_t, a_t, \ldots, s_{t+d-1}, a_{t+d-1}, s_{t+d})$ as a function of θ and ϕ:

$$C(s_{t:t+d}, \theta, \phi) = -V_\phi(s_t) + \gamma^d V_\phi(s_{t+d}) + \sum_{i=1}^{d-1} \gamma^i [r(s_{i+t}, a_{i+t}) - \tau \log \pi_\theta(a_{i+t} \mid s_{i+t})].$$

The goal of a learning algorithm can then be to find V_ϕ and π_θ such that $C(s_{t:t+d}, \theta, \phi)$ is as close to 0 as possible for all sub-trajectories $s_{t:t+d}$. Therefore the objective of the PCL algorithm can be stated as follows:

$$min_{\theta,\phi} O_{PCL}(\theta, \phi) = \sum_{s_{t:t+d} \in E} \frac{1}{2} C(s_{t:t+d}, \theta, \phi)^2.$$

The PCL update rules for θ and ϕ are derived by calculating the gradient of the objective above. For a given trajectory $s_{t:t+d}$ the gradients look like:

$$\Delta\theta = \nu_\pi C(s_{t:t+d}, \theta, \phi) \sum_{i=1}^{d-1} \nabla_\theta \log \pi_\theta(a_{i+t} \mid s_{i+t})$$
$$\Delta\phi = \nu_v C(s_{t:t+d}, \theta, \phi)(\nabla_\phi V_\phi(s_t) - \gamma^d \nabla_\phi V_\phi(s_{t+d})).$$

With ν_π and ν_π the value and policy learning rates. Since the consistence property needs to hold on any path, the PCL algorithm applies updates to trajectories sampled on-policy from $_\theta$ as well as trajectories sampled from a replay buffer. The union of these trajectories comprise the set E.
PCL can also leverage the asynchronous framework defined in Section 3.1, the pseudocode for asynchronous PCL is provided in Appendix 9.3.

3.3 Entropy for Exploration

The entropy of the policy π_θ is commonly added to the objective function of the A3C algorithm (Section 3.1) in order to improve exploration by discouraging premature convergence to suboptimal deterministic policies. This technique was originally proposed by [11]. Entropy is defined as the average amount of information produced by a stochastic source of data.

$$H(\pi_\theta(s_t)) = -\sum_a \pi_\theta(a \mid s_t) \log(\pi_\theta(a \mid s_t)).$$

The gradient of the full objective function including the entropy regularization term with respect to the policy parameters takes the form:

$$\nabla_\theta \log \pi_\theta(a_t \mid s_t) A_\phi(a_t, s_t) + \beta \nabla_\theta H(\pi_\theta(s_t)).$$

Where β is a hyperparameter controlling the strength of the entropy regularization term.
This maximum entropy objective differs from the one introduced in Section 3.2.1, for details revisit Section 3.2.1 or see [6].

3.4 Noisy-Nets for Exploration

Noisy-Nets [3] are a neural networks architecture whose weights and biases are perturbed by a parametric function of noise. This induced stochasticity of the agent's policy can be used to aid efficient exploration. Those noisy parameters are adapted with gradient descent while the agent learns.
So lets define the corresponding setting. Let $y = f(x)$ be a neural network parameterised by the

vector of noisy parameters θ which takes the input x and outputs y. The noisy parameters θ are represented as

$$\theta \stackrel{def}{=} \mu + \Sigma \odot \epsilon, \text{ where } \eta \stackrel{def}{=} (\mu, \Sigma) \text{ is a set of learnable parameters.}$$

ϵ is a vector of zero-mean noise with fixed statistics (e.g. mean, variance) and $\odot$ represents element-wise multiplication. The usual loss of the neural network is wrapped by expectation over the noise $\epsilon : \overline{L}(\eta) = \mathbb{E}[L(\theta)]$. Optimisation now occurs with respect to the set of paramters η.
Consider a linear layer of a neural network with p inputs and q outputs, represented by

$$y = wx + b,$$

where $x \in \mathbb{R}^p$ is the layer input,$w \in \mathbb{R}^{qp}$ the weight matrix, and $b \in \mathbb{R}^q$the bias. The corresponding noisy linear layer is defined as:

$$y \stackrel{def}{=} (\mu^w + \sigma^w \odot \epsilon^w)x + \mu^w + \sigma^b \odot \epsilon^b,$$

where $\mu^w + \sigma^w \odot \epsilon^w$ and $\mu^w + \sigma^b \odot \epsilon^b$ replace w and b. The parameters $\mu^w \in \mathbb{R}^{qp}, \mu^b \in \mathbb{R}^q, \sigma^w \in \mathbb{R}^{qp}$ and $\sigma^b \in \mathbb{R}^q$ are learnable whereas $\epsilon^w \in \mathbb{R}^{qp}$ and $\epsilon^b \in \mathbb{R}^q$ are noisy random variables. Two options of a specific noise distribution for the noisy random variables are introduced in [3].

3.4.1 Independent Gaussian Noise

The noise applied to each weight and bias of the neural network feedforward layer is independent, where each entry $\epsilon^w_{i,j}$ (respectively each entry ϵ^b_j) of the random matrix ϵ^w (respectively of the random vector ϵ^b) is drawn from a unit Gaussian distribution. This means that for each noisy feedforward layer, there are $pq + q$ noise variables (for p inputs to the layer and q outputs).

3.4.2 Factorised Gaussian Noise

This noise assumption reduces the number of random variables in the network from one per weight, to one per input and one per output of each noisy linear layer. By factorising $\epsilon^w_{i,j}$, we can use p unit Gaussian variables ϵ_i for noise of the inputs and q unit Gaussian variables ϵ_j for noise of the outputs (thus $p + q$ unit Gaussian variables in total). Each $\epsilon^w_{i,j}$ and ϵ^b_j can then be written as:

$$\epsilon^w_{i,j} = f(\epsilon_i)f(\epsilon_j)$$
$$\epsilon^b_j = f(\epsilon_j)$$

where f is a function. [3] proposes to use $f(x) = sgn(x)\sqrt{\mid x \mid}$.

3.4.3 Application in A3C and PCL

We assume a shared network for the policy $\pi_{\theta\pi}$ and value function $V_{\theta v}$. The policy-head $\pi_{\theta\pi}$ consists of fully-connected layers with a softmax output whereas the value-function-head $V_{\theta v}$ consists of fully-connected layers with a linear output. One can now choose if both layers corresponding to the two heads are adopted with noisy layers or if one only wants to perturb the policy or value function part.
When adding noisy weights to the network, sampling these parameters corresponds to choosing a different current policy which naturally favours exploration. In an algorithmic setting one would sample new noisy weights whenever the environment gets reset, to slightly disturb the current policy when performing the next episodic rollout of the environment at hand [3].

3.5 Generalized Advantage Estimation

As introduced in Section 3.1 and Section 3.2 the typical problem formulation in reinforcement learning is to maximize the expected total reward of a policy. A key challenge is the long time delay between actions and their positive or negative effect on reward and in the context of policy gradient

methods the high variance of the policy estimate. Applying value functions represent one solution to the problem described. They allow us to estimate the goodness of an action before the delayed reward arrives. By leveraging value functions in the actor-critic framework described in Section 3.1 we lower the variance but at the cost of introducing bias.
Even with an unlimited number of samples, bias can cause the algorithm to fail to converge, or to converge to a poor solution that is not even a local optimum. Generalized Advantage Estimation [20] aims to significantly reduce variance while maintaining a tolerable level of bias.

The choice of $\Psi_t = A^\pi(s_t, a_t)$ as defined in Section 1.1 yields almost the lowest possible variance though in practice, the advantage function is not known and must be estimated. A step in the policy gradient direction should increase the probability of better-than-average actions and decrease the probability of worse-than-average actions. The advantage function, by it's definition $A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$ measures whether or not the action is better or worse than the policy's default behavior. Hence we should choose Ψ_t to be the advantage function $A^\pi(s_t, a_t)$, so that the gradient term $\Psi_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)$ points in the direction of increased $\pi_\theta(a_t \mid s_t)$ if and only if $A^\pi(s_t, a_t) > 0$ [20].
The parameter γ introduced in Section 1.4 allows us to reduce variance by downweighting rewards corresponding to delayed effects, at the cost of introducing bias. We are going to derive an estimate $\tilde{A}_t$ of $A^\pi(s_t, a_t)$, which we are going to use to construct a policy gradient estimator of the form:

$$\tilde{g} = \frac{1}{N} \sum_{n=1}^{N} \sum_{t=0}^{\infty} \tilde{A}_t^n \nabla_\theta \log \pi_\theta(a_t^n \mid s_t^n),$$

where n index over batch of episodes.

Let V^π be an approximate value function and define $\delta_t^{V^\pi} = r_t + \gamma V^\pi(s_{t+1}) - V\pi(s_t)$. As introduced in Section 3.1 $\delta_t^{V^\pi}$ can be considered as an estimate of the advantage of the action a_t at s_t. But only under certain conditions is V^π an unbiased estimator of A^π, see [20].
Let us consider taking the sum of k of these δ terms, which we will denote by $\tilde{A}_t^{(k)}$.

$$\begin{aligned}
\tilde{A}_t^{(1)} &:= \delta_t^V = -V(s_t) + r_t + \gamma V(s_{t+1}) \\
\tilde{A}_t^{(2)} &:= \delta_t^V + \delta_{t+1}^V = -V(s_t) + r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) \\
\tilde{A}_t^{(3)} &:= \delta_t^V + \delta_{t+1}^V +^2 \delta_{t+2}^V = -V(s_t) + r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 V(s_{t+3}) \\
&\dots \\
\tilde{A}_t^{(k)} &:= \sum_{l=0}^{k-1} \gamma^l \delta_{t+l}^V = -V(s_t) + r_t + \gamma r_{t+1} + \dots + \gamma^{k-1} r_{t+k-1} + \gamma^k V(s_{t+k}).
\end{aligned}$$

The bias generally becomes smaller as $k \to \infty$, since the term $\gamma^k V(s_{t+k})$ becomes more heavily discounted, and the term $-V(s_t)$ does not affect the bias. Taking $k \to \infty$, we get

$$\tilde{A}_t^{(\infty)} = \sum_{l=0}^{\infty} \gamma^l \delta_{t+l}^V = -V(s_t) + \sum_{t=0}^{\infty} \gamma^l r_{t+l}.$$

The generalized advantage estimator $GAE(\gamma, \lambda)$ is defined as the exponentially-weighted average of these k -step estimators:

$$\begin{aligned}
\tilde{A}_t^{GAE(\gamma,\lambda)} &= (1-\lambda)(\tilde{A}_t^{(1)} + \lambda \tilde{A}_t^{(2)} + \lambda^2 \tilde{A}_t^{(3)} + \dots) \\
&= (1-\lambda)\Big(\delta_t^V (\frac{1}{1-\lambda}) + \gamma \delta_{t+1}^V \frac{\lambda}{1-\lambda} + \gamma^2 \delta_{t+2}^V \frac{\lambda^2}{1-\lambda} + \dots\Big) \\
&= \sum_{t=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V.
\end{aligned}$$

There are two notable special cases of this formula, obtained by setting $\lambda = 0$ and $\lambda = 1$ [20]

$$GAE(\gamma, 0) : \tilde{A}_t := \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$
$$GAE(\gamma, 1) : \tilde{A}_t := \sum_{t=0}^{\infty} (\gamma)^l \delta_{t+l}^V = \sum_{l=0}^{\infty} (\gamma\lambda)^l r_{t+l} - V(s_t).$$

$GAE(\gamma, 1)$ is an unbiased estimator but it has high variance due to the sum of terms. $GAE(\gamma, 0)$ induces more bias but it typically has much lower variance. The generalized advantage estimator for $0 < \lambda < 1$ makes a compromise between bias and variance, controlled by parameter λ.

3.6 Adaptive Learning Rate via KL-Divergence

When a paramterized policy function $\pi_\theta(a \mid s)$ is learned by a reinforcement learning agent, one separates two different changes after a gradient update is obtained based on a batch of a sequence of states s_t and actions a_t. A change in parameter space θ and a change in the policy space $\pi_\theta(\cdot \mid \cdot)$. One defines a learning rate ν that the optimizer at hand (e.g. ADAM) uses to update the parameters θ. This learning rate ν should be chosen in a way that the parameters θ do not change to heavily but at the same time get altered sufficiently such that learning is obtained. But a small step in the parameter space is not equivalent to a change in the policy space, small updates to θ can alter the policy significantly and vis versa. But when learning a policy $\pi_\theta(a \mid s)$ one wants to make sure to perform steps that do not alter the policy in an uncontrolled manner. One way to measure the change of the policy is the KL-Divergence of the most recent policy and the the policy before the last update step $\pi_{\theta_{old}}(a \mid s)$.

$$KL(\pi_{\theta_{old}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)) = \sum_a \pi_{\theta_{old}}(a \mid s_t) \log \frac{\pi_{\theta_{old}}(a \mid s_t)}{\pi_\theta(a \mid s_t)}.$$

The definition above applies for a particular s_t but we want to control the KL-divergence with respect to all possible states s our environment can reach. So a value δ is defined, that our KL-divergence should not exceed:

$$KL(\pi_{\theta old}, \pi_\theta) = \mathbb{E}_s[KL(\pi_{\theta_{old}}(\cdot \mid s), \pi_\theta(\cdot \mid s))] < \delta.$$

Therefore heuristics can be implemented to increase the learning rate ν if $KL(\pi_{\theta old}, \pi_\theta) < \delta$ and decrease the learning rate if $KL(\pi_{\theta old}, \pi_\theta) > \delta$. One possible heuristic is depicted below:
Compute an estimate $d = \tilde{KL}(\pi_{\theta old}, \pi_\theta)$ based on an action, state sequence and adjust the learning rate ν in the following way.

$$\text{If } d < \frac{\delta}{1.5}, \nu \leftarrow \nu \cdot 2$$
$$\text{If } d > \delta \cdot 1.5, \nu \leftarrow \frac{\nu}{2}.$$

More sophisticated approaches to control the learning of an agent based on the KL-divergence have been focus topics in the literature, therefore see [21][22].

4 Neural Networks as Function Approximators in RL

For complex approximation tasks as in reinforcement learning functions we need to have both expressive enough and easy-to-distribute function approximators. We focused on applying deep neural networks [4] as well-defined standard in such tasks.

4.1 Preprocessing of Images

In contrast to image recognition, where the details or the picture are important and even a high resolution may be beneficial, in learning of Atari games the required information size is typically a lot smaller than the visible picture. This means, when using the visual screen input, preprocessing should be done to speed up the computations.
In the Atari environment that we use, OpenAI-Gym, the images come as a (210, 160, 3) shaped tensor representing rgb colored pixels on a TV screen[17]. The different preprocessing methods are then selected with a set of parameters, which can be changed conveniently.

Some important preprocessing methods are the following:

4.1.1 Frame skipping

Skipping of Frames has proven itself to be generally a good idea, as for most games the speedup of the episodes has had more impact on the learning rate than the information loss from skipped frames.
However, a static frame skip can lead to problems occasionally , for example a static four frame skip in Space Invaders leads to the blinking lasers of the monsters being invisible. To minimize the loss of information one should thus use a random frame skip e.g. from a range of 2-4 frames[11]. The environment OpenAI-gym does this automatically when producing new states.

4.1.2 Cropping

Atari games often have "Dead Space" at the borders or, even worse objects that may confuse the learner. Ideally , everything which should have no impact on the agent's policy should be cropped out. See figure 4 where in the game "Pong" borders, score as well as enemy player have been cropped out as they should not influence the learner's decisions.

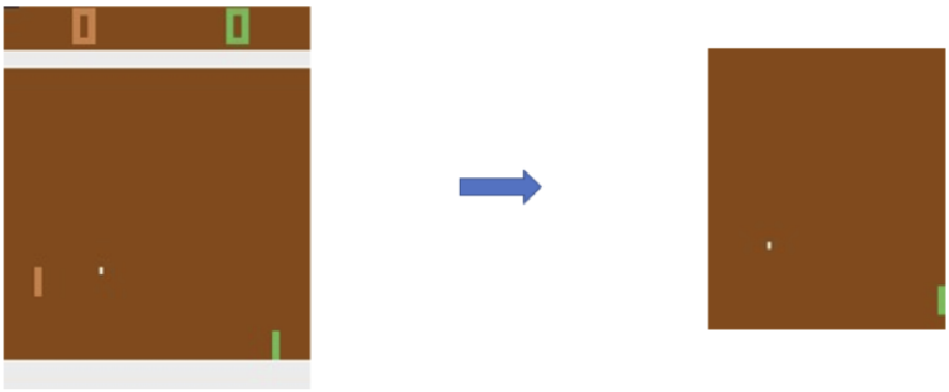


Figure 4: Cropped Pong image

4.1.3 Downsizing

The reduction of the picture resolution by downscaling will often maintain the important information while reducing the size of the input greatly. Some sort of downsizing is almost always used in implementations that learn games from visual input.

4.1.4 Grayscale

Conversion to grayscale allows to drop one dimension of the input tensor further reducing the complexity of the observations.
Going even further on that note, the range of the color can also be reduced, in the most extreme case to a binary spectrum. For this, we provide a preprocessing class which chops up the color spectrum in chunks of desired size. In the game of "Pong" for example a ternary color range differentiating between background, ball and racket maintains all of the useful information (5).

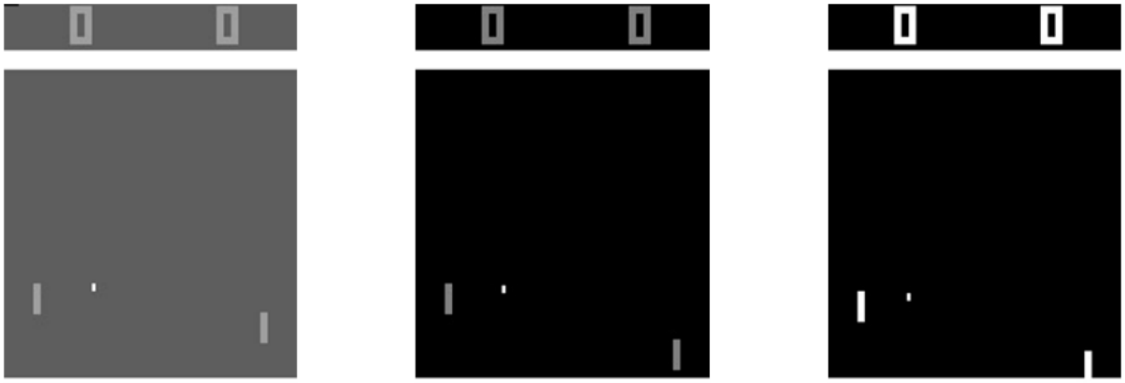


Figure 5: left: grayscale image, middle: 3 color values, right: 2 color values

4.2 Sequencing

The next state in an Atari game sometimes not only depend on the current picture (and the action), thus not fulfilling the Markov property, which is desired for a memoryless agent. For example in the game of "Pong" the ball has a trajectory which can not be known from a single frame, but will influence where the ball will go in the next frame.
To circumvent this problem the input can, instead of a single frame, be a sequence of some number of past frames, thus acquiring the Markov property in the "Pong" example.

4.3 Convolutional Neural Networks to capture Image Dynamics

After applying a combination of preprocessing steps to the image we normalize the resulting image matrix and pass it to the neural network afterwards.
For processing the state vector we applied several architectures and their combinations. In case of the image as the state, it is useful to apply convolutional neural networks (CNN) to recognize space patterns. For each of the environments, the exact architecture of the network can differ, but general structure stays the same as the cascade of consecutively applied filters to initial vector and (optionally) maximum pooling between layers. Kernel sizes usually are chosen in descending way to regularize the information on first layers more aggressively. We use ReLU as activation function. In Figure 6 you can see an example of applying CNN to the state.

4.4 LSTM Networks to capture Time Dependencies

We also apply long short-term memory network (LSTM) on top of the CNN to catch temporal dependencies, which can be crucially in some environments. LSTM network can be used independently in the environments, where the state is represented as coordinate vector, for example. In short, LSTM, the extension of classical recurrent neural network (RNN), has an ability to add and remove information to the cell state based on longer term dependencies, than RNN.
In Figure 7 you can see the general schema of LSTM network in case of Gated Recurrent Unit. Here first equation to the right of the schema defines "forget gate layer", where under the sigmoid

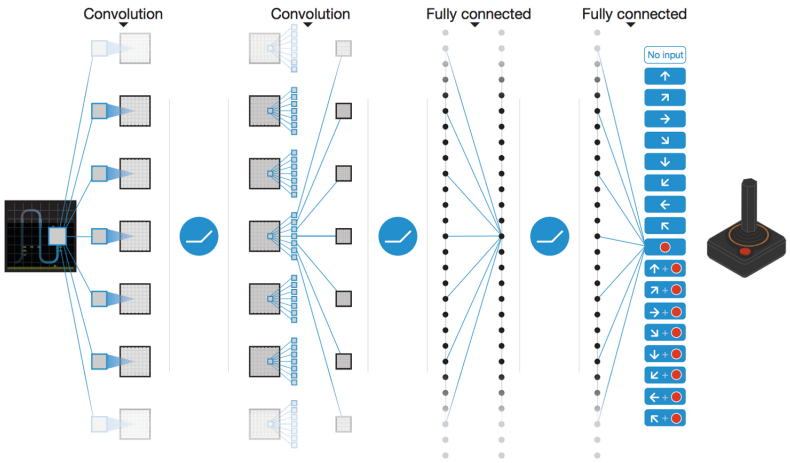
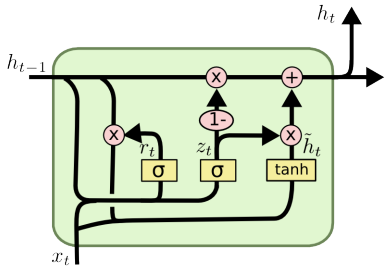


Figure 6: Example of using CNN in reinforcement learning task [7]

function h_{t-1} is the hidden state of the previous cell, x is the current state. The next two expressions define the rule to update the cell with new values, in this case we have merged hidden and cell state. The last formula defines the output of the cell.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 7: Gated Recurrent Unit schema [16]

After applying these processing steps to the raw state we need to pass the resulting vector to the policy and value networks. In case of A3C we have the shared output, which is passed to fully-connected layers of policy and value functions. We describe modification of applying Noisy-Nets for better exploration to value and policy networks in Section 3.4.

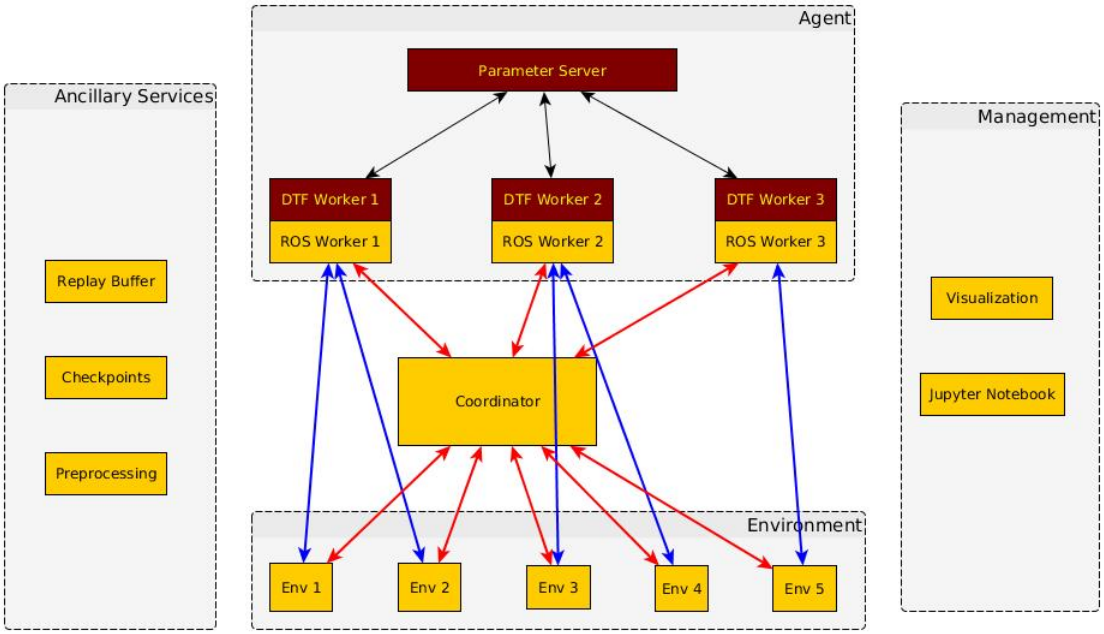


Figure 8: Distributed Reinforcement Learning in ROS

5 ROS as a distributed Reinforcement Learning Infrastructure

Reinforcement Learning (RL) essentially comprises of an AI 'agent' that communicates with a particular 'environment' and learns useful behavior (1). This naturally leads to a client server architecture that is available in ROS (Section 2.4.2). A second characteristic of RL training is that the agent can utilize parallel querying of the environment, which leads to faster exploration and hence quicker convergence. Thus we endeavored to use ROS for this distributed learning setup that can be used by any RL agent, regardless of the RL algorithm used.
Such an environment already exists at Google Inc. and is aptly named as GORILA (Google Reinforcement Learning Architecture) [14]; however it is not open source. Thus our infrastructure is basically an open source version of GORILA.

5.1 Framework

The proposed network is shown in Figure 8. Essentially there are two major sections of the framework that communicate with each other at specific points. One section comprises of the ROS nodes and provisions the environment in a scalable manner (yellow boxes). The other section comprises of a distributed deep learning framework that can access multiple copies of environments in a transparent manner (brown boxes). This modularity lends to easy swapping of deep learning framework.
Both the 'agent' and the 'environment' are implemented as ROS nodes and they communicate using the client-server architecture of ROS. Some ancillary functions that help in RL training are also implemented as ROS nodes and they use either client-server or publisher-subscriber model depending on the function. Our framework consists of four types of nodes:

1. **Environment Nodes**
2. **Agent (Worker) Nodes**
3. **Coordinator**
4. **Ancillary Services**

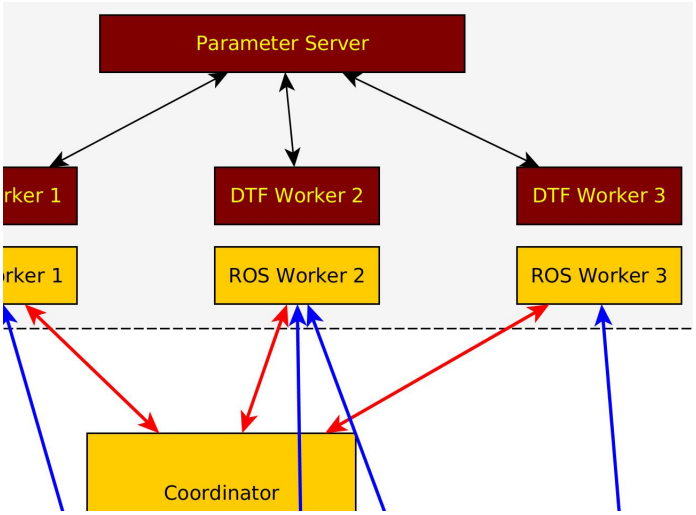


Figure 9: ROS-DTF communication

5.1.1 Coordinator

This is a central node that registers and coordinates communication between all other type of nodes. Basically it provides registration services for worker and environment nodes and evenly distributes the environments between the available workers; thus achieving load balancing. It also registers ancillary nodes that can be used by worker/environment nodes.

5.1.2 Environment Nodes

These nodes represent the 'environment' in an RL architecture and house the simulation environments. One environment node can have multiple copies of environments thus providing a second level of parallelism. Another advantage of using multiple copies is to achieve higher compute vs communication.

5.1.3 Worker Nodes

The worker nodes represent the RL 'agent' and are the hubs where all training takes place. These nodes are connected to distributed worker component of deep learning environment e.g. Tensorflow or PyTorch. Hence the Tensorflow/PyTorch will also reside in the same process and data is transmitted to/from the worker nodes

5.1.4 Ancillary Services

These nodes/functions provide ancillary services; for example 'replay buffers', tensorboard/visualization nodes, checkpointing etc.

5.2 ROS-DTF Interface

As stated earlier, the process running worker node comprises of both ROS and distributed Deep Learning framework. (From here on we will use Distributed Tensorflow (DTF) as the framework but it can be swapped by any other). Looking at the basic architecture of RL (Figure 1) we see that the agent (implemented as the DTF worker) requires state and reward as input and outputs an action. Similarly, the ROS Worker needs an action as input which is run through by the Gym Environment (or any other simulation) and outputs rewards and next states. This naturally leads to following pseudo-code for ROS-DL interface loop:

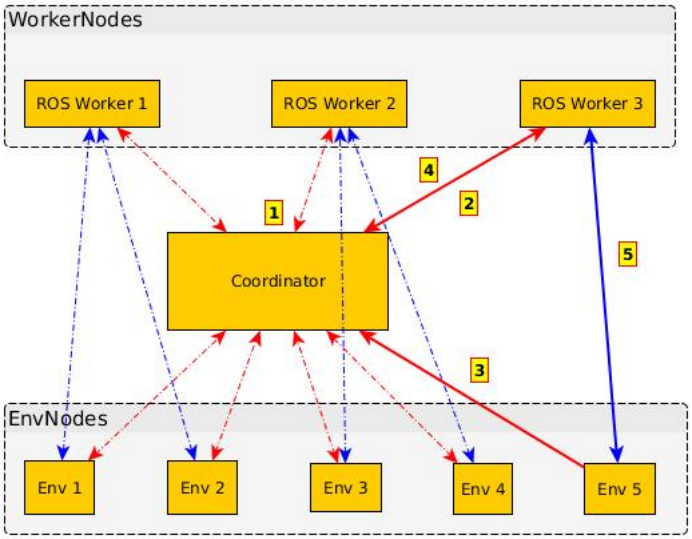


Figure 10: Distributed Reinforcement Learning in ROS

$$action = DistributedTF(state, reward)$$
$$[state, reward] = ROSworker(action)$$

The three main nodes are the EnvNodes, WorkerNodes and the Coordinator Node. In the following we are providing a typical sequence with which the whole system is brought up. The enumerated steps are depicted in Figure 10.

1. There is only one Coordinator Node in a system and it is brought up first.
2. Then a WorkerNode is brought up and it registers itself with the Coordinator Node.
3. Then an EnvNode is brought up. It also registers itself with the Coordinator Node and conveys information like number and type of Gym environments it is running and their initial state.
4. Here, the Coordinator Node decides to which WorkerNode this EnvNode should be allocated. The corresponding WorkerNode is then passed the information about this EnvNode.
5. Subsequently all communication happens directly between the two nodes

5.3 Deployment in LRZ Cloud

We used LRZ Cloud platform (section 2.3) for the deployment of the system. Since the bulk of communication is between a Worker Node and its assigned Environment Nodes, there is no singular central communication hub. Thus the system can scale up arbitrarily. For automatic deployment of this system, we note that the LRZ Cloud's Amazon-EC interface enables a user to launch VMs from a bash script. Moreover, the cloud-init functionality enables the user to install software and execute it on the VM at the time of provisioning. We utilized both these features and thus developed a script that does the following on the fly:

1. Launch VMs and designate them as EnvNodes or Worker Nodes.
2. Specify the software to be installed when the VM is running.

3. Pass the IP addresses of all machines as environment variables
4. Setup ROS and Distributed Tensorflow environments
5. Start Controller, Workers and Environments in proper sequence.
6. Setup Tensorboard for monitoring the progress of training.

6 Tests and Experiments

For computation intensive environments an exhaustive (hyper-)parameter search without heuristics is practically impossible, as one successful training run takes several hours even with high end hardware. We have instead concluded a parameter study on one of the classic control environments, "Cartpole".

6.1 Cartpole Study

The game involves a pole, that is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum starts upright and the goal is to prevent it from falling over by increasing and reducing the cart's velocity. Each timestep of not falling over grants a reward up until a maximum reward threshold is reached.
The starting point for the tests is a stable parameter combination for the A3C algorithm that we used, can be found in 9.4. Then the parameters were slightly disturbed and a series of 15 runs was started until the environment was either solved or exceeded 2500 steps. An environment was considered solved when maximum reward was achieved 5 times in a row.

6.1.1 Learning Rate

Selection of the right learning rate is crucial since the algorithm will either overshoot a global optimum and be instable when the learning rate is too big, or get stuck in smaller local optima and have slower learning, if the learning rate is too small. Indeed, starting at a learning rate of about 0.02 tests began to show that even when maximum reward was reached once, it would not settle there but jump back to low values.

learning rate	% solved	average ep to solved
0.003	93	1389
0.005	80	1047
0.01	100	585
0.02	86	1635

6.1.2 Environment Number

A bigger number of environments will produce more uncorrelated samples and thus making the algorithm a lot more stable, but will also increase the computing requirement for one gradient update. For Cartpole two environments (on each of the six workers) had the most time efficient results but higher number of environments converged more likley and in less gradient updates.

num envs	% solved	average ep to solved
1	93	690
2	100	220
4	100	189
12	100	162

6.1.3 Batch Size

Generally a smaller batch size will produce more gradient updates which can be advantageous if the total number of samples is not high. Nevertheless there is some evidence that for large samples, relatively large batch sizes don't increase the loss significantly[5].
In the case of reinforcement learning batch size is also connected to the reward structure as usually the reward discounts are calculated over one batch. The optimal value here is highly dependant on the environment but there will almost always occur some amount of information loss if the batch size is too small.

batch size	% solved	average steps to solved
episode	100	11452
40	100	8380
20	80	17732
10	93	14112

6.1.4 Discount Factor

Regulates the impact of rewards on previous states and is again dependent on the environment and also the time scale of the episodes.

gamma	% solved	average ep to solved
0.9	100	1006
0.98	86	740
0.99	100	492
0.995	100	438

6.1.5 Noisy Nets

Noise pertubations obviously have to be handled carefully, as they will harm the learning progress if they are to severe. However, our tests have shown good convergence properties if they are applied to both value and policy network.

noise	% solved	average ep to solved
none	93	779
policy	60	1069
value	80	833
both	100	1205

7 Conclusion

Neural networks (4) seem the right function approximator choice to solve reinforcement learning problems without manual handcrafting of features End-To-End (besides some preprocessing of images to enhance training speed). But training those is really hard, even reproducing already achieved results is not a trivial challenge. Even various low-level engineering tweaks of the algorithms such as gradient clipping can dramatically affect overall performance. Due to the several levels of hyperparameters (preprocessing directives, neural network specification, learning process parameters) it is hard to find the right balance between them. Moreover, the learning process doesn't begin instantly after running the algorithm, at least it is hard to notice that from evolution of losses, so it requires the huge amount of computational time to actually see the results of particular hyperparameters combination.
Given those challenges, we implemented a **scalable reinforcement learning workbench** (5):

- Including current state-of-the-art algorithms (A3C, PCL) and optimization tweaks (GAE, Noisy-Nets, KL-Divergence learning rate adaption (3)) as well as a broad preprocessing spectrum.
- The implementation is structured in a way that there is a central place to change all hyperparameters and environment settings needed in the learning process.
- A comprehensive and flexible network architecture based on a configuration dictionary has been implemented to exchange the network structure rapidly from purely feed forward to CNN networks combined with LSTM layers.
- We managed to implement A3C [11] and PCL [12] in a scalable manner to leverage multiple machines like GORILA [14] as well as multiple threads on the same machine like the proposed implementation in [11], see Section 5.
- Tensorflow allowed us to produce a clean workflow, based on the repository location folders holding tensorboard logs and model checkpoints are created. While learning progresses one can follow several metrics (e.g. learning rate, gradient norm, advantage and value function estimates) in the tensorboard. Based on user defined settings, model checkpoints are saved and can be restored any time on any machine to pick up the parameters and continue learning.

Still challenges remain, given a neural network architecture, an optimizer with corresponding learning rate and a fixed batch size that solved or produced high scores on one environment might not solve the same environment when started with a different random seed of the environment. We observed training one agent on multiple environments with different random seeds stabilized learning (see Section 6.1), but slowed down learning since larger batches need to be processed and produced. So, more research needs to be done to develop more stable and data efficient algorithms, the theory behind the softmax consistency introduced in Section 3.2.1[12] might be a first step into this direction. More research in the way reinforcement learning agents approximated by neural networks are optimized would also be a possible way to go, some remarks are proposed in [11].
Additional effort needs to be done to enhance training speed, GA3C might be one possibility, see Section 8.2 [1].

During the course of project, we came across many use-cases where a distributed RL framework (Section 5) developed in ROS proved advantageous. They are enumerated below:

- This framework scales up arbitrarily without creating communication bottlenecks.
- This scalability is vital for more computationally demanding environments as it affords multiple environments to be explored in parallel.
- ROS provides a robust framework through its built-in client server architecture which takes care of communication and provides mechanism for node failure.

- Another advantage of having multiple environment nodes running in parallel, is the potential utilization of GPU based RL algorithm implementations (e.g. GA3C [1]) which require large amounts of data to be computationally efficient.
- Finally, ROS is used in a lot of Robotics related projects and hence these projects can easily incorporate RL algorithms through this framework.

8 Outlook

Reinforcement Learning is a fast evolving field and all of us can be sure to see rapid progress in theory and applications in the upcoming years. In the following, two current research results in the domain of underlying mathematical theory (PCL [12][13]) and efficient implementation (GA3C [1]) are presented, as well as a an outlook regarding ROS in the context of reinforcement learning.

8.1 Comprehensive PCL Evaluation and Trust-PCL

Due to the short amount of time, we have not been able to evaluate PCL, Section 3.2 [12], on image based reinforcement learning problems like Atari games. Since the algorithm is quite recent, no comprehensive study in this domain has been conducted so far.
Interesting features to test would be:

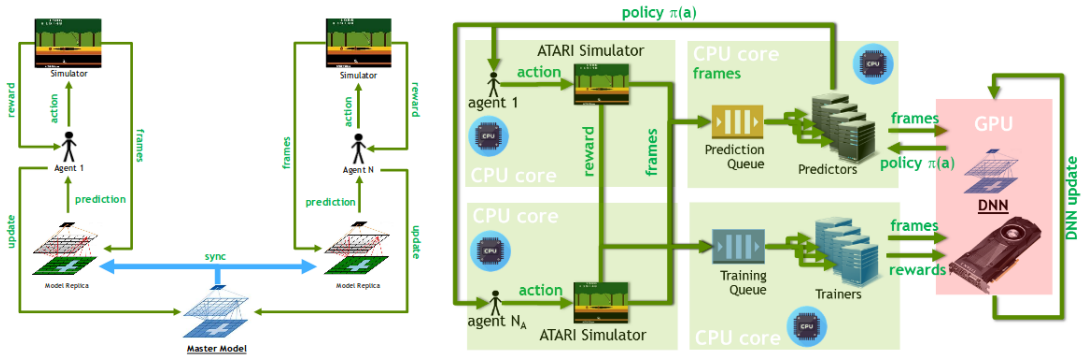
- Combinations of offline and online updates, to leverage promising episodes via offline updates but at the same time explore the environment via online updates.
- The influence of the entropy regularization parameter τ and different adaption techniques like continuously decay while training.
- Various replay buffer prioritization could be tested as well, like reward based, time based, random or mixtures of the mentioned [12].
- The rollout length used in calculating the objective is a important hyperparameter in the PCL framework. Testing different values and averaging over various lengths could stabilize learning.

In the end of last year a paper called *"Trust-PCL: An Off-Policy Trust Region Method for Continuous Control"*[13] has been published. This method build on top of the PCL [12] theory is an off-policy trust region method. Trust-PCL is able to maintain optimization stability while exploiting off-policy data to improve sample efficiency. Adoptions required for Trust-PCL could be incorporated in our current implementation of PCL.

8.2 GA3C - Boost computation speed of you learner via GPUs

The general A3C implementation introduced by [11] uses multiple worker threads, that all contain a copy of the global model and update this asynchronously. The framework is developed to use CPUs. General benchmarks [11] and our experience show that using around 8 worker threads to learn Atari-Games like Pong, MsPacman or SpaceInvaders takes multiple days to see learning progress. The main reason for using CPU other than GPU, is the inherently sequential nature of RL in general, and A3C in particular. In RL, the training data are generated while learning, which means the training and inference batches are small and GPU is mostly idle during the training, waiting for new data to arrive. Therefore a CPU implementation is as fast as a naive GPU implementation.
Nvidia published a paper [1] which introduces a hybrid CPU/GPU version of the Asynchronous Advantage Actor-Critic (A3C) algorithm, that leverages a system of queues and a dynamic scheduling strategy, achieving a significant speed up compared to a pure CPU implementation.
The architecture depicted in Figure 11 consists of a Deep Neural Network (DNN) with training and prediction on a GPU and a multi-process, multi-thread CPU architecture talking to the GPU instance of the model.

- **Agent** is a process interacting with the simulation environment, that chooses actions according to the learned policy and gathers experience used for training the parameterized policy π_θ. Multiple concurrent agents run independent instances of the environment, where one Agent occupies one thread/CPU. Each agent does not have its own copy of the model, it queues policy requests in a Prediction Queue before each action, and periodically submits a batch of input/reward experiences to a Training Queue.



(a) A3C (b) GA3C

Figure 11: Comparison of A3C and GA3C architectures. Agents act concurrently both in A3C and GA3C. In A3C, however, each agent has a replica of the model, whereas in GA3C there is only one GPU instance of the model. In GA3C, agents utilize predictors to query the network for policies while trainers gather experiences for network updates, source [1].

- **Predictor** is a thread which dequeues as many prediction requests as are immediately available from the Prediction Queue and batches them into a single inference query to the DNN model on the GPU. When predictions are completed, the predictor returns the requested policy to each respective waiting agent.
- **Trainer** is a thread which dequeues training batches from the Training Queue submitted by agents and submits them to the GPU for model updates.

To summarise this approach, various threads (Trainer, Agent, Predictor) running on top of CPUs are started and requesting different computations (prediction, update) that are performed by the DNN located on the GPU. This architecture leverages the highly parallel nature of a GPU and utilizes the available resources sufficiently.
The implementation of NVIDIA can be found here [18]. Even multi GPU version have been implemented and can be found on Github [23].
PCL could be also incorperated into this framework, the online learning part would be the same as in GA3C. To also leverage a replay buffer of promising rollouts of the environment, one could introduce another Thread (e.g. **OfflineTrainer**) or add additional functionalities to the Trainer thread in order to sample episodes from a replay buffer.
A recently published paper called *"IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures"* refined the GA3C approach with more mathematical theory and showed IMPALA is able to solve multi-task reinforcement learning challenges [2].

8.3 Future outlook of ROS based scaling

The framework is very basic in terms of the flexibility it provides. There are quite a few features that need to be implemented for it to be deployed in realistic RL training settings. Some of these features are following :

- Presently, the framework does not handle node failures either in EnvNodes or in WorkerNodes. Since node failure can be handled in multiple ways in ROS framework, extending it to this framework is a good feature to have.
- Presently, the framework handles only one type of simulation environment. In settings of transfer learning, it will be nice to have multiple types of environments running and being trained simultaneously.
- The ancillary functions are not implemented in separate nodes. Doing so will enable scaling them up simultaneously. This is especially useful in case of replay buffers.

- ROS also provides a subsriber-publisher model of communication; which can be more efficient than a client-server model, when the system is scaled up beyond a certain number of nodes.
- Deploying this system in an already existing robotic environment with multiple different kind of nodes will be an interesting challenge.

9 Appendix

9.1 Pseudocode Actor-Critic-Algorithm

Algorithm 2 Actor-Critic algorithm

Initialize random policy $\pi_\theta(a \mid s)$, random value function $V_\phi^\pi(s)$, learning rate α
while θ not converged **do**
    Sample $\{s_t, a_t\}$ from $\pi_\theta(a \mid s)$ ▷ run the policy
    Fit $V_\phi^\pi(s)$ by performing gradient descent on $min_\phi \frac{1}{2} \sum_t (V_\phi^\pi(s_t) - y_t)^2$
    Evaluate $A^\pi(s_t, a_t) = r(s_t, a_t) + V_\phi^\pi(s_{t+1}) - V_\phi^\pi(s_1)$
    Estimate $\nabla_\theta \mathcal{J}(\theta) \approx \sum_{t=1}^{T} \nabla \log(\pi_\theta(a_t^i \mid s_t^i)) A^\pi(s_t, a_t)$
    Update $\theta \leftarrow \theta + \alpha \nabla_\theta \mathcal{J}(\theta)$
end while

9.2 Pseudocode A3C-Algorithm

Algorithm 3 A3C - pseudocode for each actor-learner thread

Assume global shared parameters vectors θ and θ_v and global shared count $T = 0$
Assume thread specific parameter θ' and θ'_v
repeat
    Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
    Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
    $t_{start} = t$
    Get state s_t
    repeat
        Perform a_t according to policy $\pi_{\theta'}(a_t \mid s_t)$
        Receive reward r_t and new state s_{t+1}
        $t \leftarrow t + 1$
        $T \leftarrow T + 1$
    until terminal s_t or $t - t_{start} == t_{max}$
    $R = \begin{cases} 0 & \text{for terminal } s_t \\ V_{\theta_v}(s_t) & \text{for non terminal} s_t \text{ bootstrap from last state} \end{cases}$
    for $i \in \{t-1, \ldots, t_{start}\}$ **do**
        $R \leftarrow r_i + \gamma R$
        Accumulate gradients wrt θ': $d\theta + \nabla_{\theta'} \log \pi_{\theta'}(a_i \mid s_i)(R - V_{\theta'_v}(s_i)$
        Accumulate gradients wrt θ'_v: $d\theta_v + \frac{\partial}{\partial \theta'_v}(R - V_{\theta'_v}(s_i))$
    end for
    Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$
until $T > T_{max}$

9.3 Pseudocode Asynchronous PCL-Algorithm

Algorithm 4 Asynchronous PCL - pseudocode for each learner thread

Assume global learning rates ν_π, ν_v, discount factor γ, rollout d, T_{max}, $T_{localmax}$
Assume local or global replay buffer, replay buffer capacity B, prioritized replay paramter α
Assume global shared parameters vectors θ and ϕ and global shared count $T = 0$
Assume thread specific parameter θ' and ϕ'
function GRADIENTS($s_{0:T}, \theta, \phi$)
    Compute $\Delta\theta = \sum_{t=0}^{T-d} C(s_{t:t+d}, \theta, \phi) \sum_{i=1}^{d-1} \nabla_\theta \log \pi_\theta(a_{i+t} \mid s_{i+t})$
    Compute $\Delta\phi = \sum_{t=0}^{T-d} C(s_{t:t+d}, \theta, \phi)(\nabla_\phi V_\phi(s_t) - \gamma^d \nabla_\phi V_\phi(s_{t+d}))$
    Return $\Delta\theta$, $\Delta\phi$
end function
repeat
    Reset gradients: $\Delta\theta \leftarrow 0$ and $\Delta\phi \leftarrow 0$
    $T_{local} = 0$
    Synchronize thread-specific parameters $\theta' = \theta$ and $\phi' = \phi$
    repeat
        Sample $s_{0:T} \sim \pi_{\theta'}$ from environment
        $\Delta\theta$, $\Delta\phi$ = Gradients($s_{0:T}, \theta', \phi'$) ▷ Online-Updates
        Update global model asynchronous $\theta \leftarrow \nu_\theta \Delta\theta$
        Update global model asynchronous $\phi \leftarrow \nu_\phi \Delta\phi$
        Add $s_{0:T}$ to replay buffer(RB) with priority Reward($s_{0:T}$)
        If $\mid RB \mid > B$, remove episodes uniformly at random
        Sample $s_{0:T}$ from RB.
        $\Delta\theta$, $\Delta\phi$ = Gradients($s_{0:T}, \theta', \phi'$) ▷ Offline-Updates
        Update global model asynchronous $\theta \leftarrow \nu_\theta \Delta\theta$
        Update global model asynchronous $\phi \leftarrow \nu_\phi \Delta\phi$
        $T = T + 1$
        $T_{local} = T_{local} + 1$
    until $T_{local} == T_{localmax}$
until $T > T_{max}$

9.4 Complete Hyperparameters used for Cartpole Study

name	value	comment
Algorithm	A3C	A3C or PCL Algorithm
learning rate	0.001	learning rate used by the optimizer
network pol	Dense	policy network used
network val	Dense	value network used
opt val	RMSProp	optimizer used for the value network
opt pol	Adam	optimizer used for the policy network
batch size	episode	number of minibatches per training step or training steps with whole episode
gamma	0.99	reward discount factor backwards in time
reward factor	0.005	rescaling factor for rewards and values
num envs	1	number of environments used by a worker per update
num workers	6	number of asynchronous worker threads
noise	none	noisy net specification (see 3.4)
preprocessing	none	preprocessing done
entropy	0	entropy regularization factor (see 3.3)

References

[1] Mohammad Babaeizadeh et al. "Reinforcement Learning through Asynchronous Advantage Actor-Critic on a GPU". In: (2016). URL: https://arxiv.org/abs/1611.06256.

[2] Lasse Espeholt et al. "IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures". In: (2018). URL: https://arxiv.org/abs/1802.01561.

[3] Meire Fortunato et al. "Noisy Networks for Exploration". In: (2017). URL: https://arxiv.org/abs/1706.10295.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[5] Priya Goyal et al. "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour". In: (2017). URL: https://research.fb.com/publications/accurate-large-minibatch-sgd-training-imagenet-in-1-hour/.

[6] Tuomas Haarnoja et al. "Reinforcement Learning with Deep Energy-Based Policies". In: (2017). URL: https://arxiv.org/abs/1702.08165.

[7] Arthur Juliani. *Simple reinforcement learning with tensorflow part 4*. 2016. URL: https://medium.com/@awjuliani/simple-reinforcement-learning-with-tensorflow-part-4-deep-q-networks-and-beyond-8438a3e2b8df.

[8] Sergey Levine. *CS 294: Deep Reinforcement Learning*. Fall 2017.

[9] Jiwei Li et al. "Deep Reinforcement Learning for Dialogue Generation". In: (2016). URL: http://arxiv.org/abs/1606.01541.

[10] Martın Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/deploy/distributed.

[11] Volodymyr Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: (2016). URL: https://arxiv.org/abs/1602.01783.

[12] Ofir Nachum et al. "Bridging the Gap Between Value and Policy Based Reinforcement Learning". In: (2017). URL: https://arxiv.org/abs/1702.08892.

[13] Ofir Nachum et al. "Trust-PCL: An Off-Policy Trust Region Method for Continuous Control". In: (2017). URL: https://arxiv.org/abs/1707.01891.

[14] Arun Nair et al. *Massively Parallel Methods for Deep Reinforcement Learning*. 2015. URL: https://arxiv.org/abs/1507.04296.

[15] Feng Niu et al. "HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent". In: (2011). URL: https://arxiv.org/abs/1106.5730.

[16] Chrisopher Olah. *Understanding LSTM Networks*. 2015. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[17] OpenAI. *Gym*. 2016. URL: https://github.com/openai/gym.

[18] NVIDIA Research Projects. *GA3C*. 2017. URL: https://github.com/NVlabs/GA3C.

[19] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*. Kobe, Japan, May 2009.

[20] John Schulman et al. "High-Dimensional Continuous Control Using Generalized Advantage Estimation". In: (2015). URL: https://arxiv.org/abs/1506.02438.

[21] John Schulman et al. "Proximal Policy Optimization Algorithms". In: (2017). URL: https://arxiv.org/abs/1707.06347.

[22] John Schulman et al. "Trust Region Policy Optimization". In: (2015). URL: https://arxiv.org/abs/1502.05477.

[23] Yuxin Wu. *tensorpack*. 2017. URL: https://github.com/ppwwyyxx/tensorpack.