Vipuvoimaa EU:lta 2014–2020
Euroopan unioni

Reinforcement Learning for Engineering Applications

Mikhail Zolotukhin,
Faculty of Information Technology, University of Jyväskylä, Finland.

Abstract

Many automatic control and optimization tasks commonly found in modern manufacturing settings involve complex contact dynamics and friction. When operating in such environments applying traditional models may result in inaccurate controllers that have to be manually tuned for each particular task. Recently, reinforcement machine learning has been demonstrated to be capable of learning controllers by interacting with the environment and progressively improving their performance at a given task, without being explicitly programmed. However, running reinforcement learning algorithms in the real world poses sample efficiency and safety challenges. For these reasons, there is a need to simulate the real environments in such a way that the agents can be trained in simulations. The main purpose of this report to demonstrate how to create a gym based on real-time simulation tools to train artificial intelligence agents and test state-of-art reinforcement learning algorithms in the resulting environments to evaluate possibility of using those for real-life control tasks.

1 Introduction

Often industrial tasks may require a degree of adaptability that is difficult to achieve with conventional robotic automation techniques such as proportional–integral–derivative (PID) controllers. Moreover, standard control methods can struggle in the presence of complex dynamical phenomena that are hard to model analytically, such as complex contacts and collisions. Reinforcement machine learning (RL) offers a different solution. Whereas common machine learning applications are tasked with learning how to make predictions, for example for speech recognition or customer segmentation, RL for engineering applications relies on trial and error learning instead of accurate modeling to construct an effective controller in tasks pertaining to autonomous vehicles and robotics even when accurate models are unavailable [1]. Moreover, RL with expressive function approximation, usually referred as deep reinforcement learning, has been shown to automatically handle high dimensional inputs such as images [2]. In this case, deep neural networks fulfill the role of helping the model to reach an internal representation of observations that enables it to provide values for the states that have not been visited yet based on the proximity to already visited ones.

In theory, using deep RL can save programming time when building a control system by implementing the controller as an RL agent which learns how to act properly taking into account features extracted from its observations of the environment. However, deep RL has thus far not seen wide adoption in the automation community due to several practical obstacles. Probably, the biggest obstacle of applying reinforcement learning algorithms in real-life environments is its low sample efficiency. Even though, recent progress in developing better RL algorithms has led to significantly better sample efficiency, even in dynamically complicated tasks [3, 4], it still remains a challenge. Another major, often underappreciated, obstacle is reward function specification. Most of the time it is carefully shaped [5] which can be a significant challenge, as one must additionally build a perception system that allows computing dense rewards on state representations. Shaping a reward function so that an agent can learn from it is also a manual process that requires considerable manual effort. An ideal RL system would learn from rewards that are natural and easy to specify. These two obstacles can be partially eliminated by designing a realistic simulation software to emulate the real world environments such that the RL agents can be trained in simulations. Even if such software is real-time, we can solve the sample efficiency challenge by running several simulations in parallel. Most of the state-of-art reinforcement learning algorithms are capable to learn from many environments at once by design. The reward shaping challenge can also be solved in the simulated environment as we can construct any sort of dense or sparse reward function using data obtained from the real world model used in the simulator.

The main purpose of this report is to make a short introduction into deep reinforcement learning and its state-of-art algorithms and evaluate those algorithms in realistic engineering applications. The rest of the report is organized as follows.

In section 2, we provide some theoretical background needed to understand deep reinforcement learning algorithms. Section 3 presents performance results of several RL algorithms using an advanced robotics and big manufacturing machine simulator. Section 4 concludes the report and outlines future work.

2 Reinforcement Learning

Reinforcement learning is a machine learning paradigm in which software agents and machines automatically determine the ideal behavior within a specific context by continually making value judgments to select good actions over bad. A reinforcement learning problem can be modeled as Markov Decision Process (MDP) that includes three following components: a set of agent states and a set of its actions, a transition probability function which evaluates the probability of making a transition from an initial state to the next state taking a certain action, and an immediate reward function which represents the reward obtained by the agent for a particular state transition. If the transition probability function is known, the agent can compute the solution before executing any action in the environment. However, in real-world environment, the agent often knows neither how the environment will change in response to its action nor what immediate reward it will receive for executing the action. It is not enough to only account the immediate reward of the current state, the far-reaching rewards should also be taken into consideration. Most of the time RL algorithms focus on the optimization of infinite-horizon discounted model, implying that the rewards that come sooner are more probable to happen, since they are more predictable than the long term future reward.

There are three main approaches for the reinforcement learning: value-based, policy-based and model-based. In value-based RL, the goal is to maximize the value function which is essentially a function that evaluates the total amount of the reward an agent can expect to accumulate over the future, starting at a particular state. The agent then uses this function by picking an action at each step that is believed to maximize the value function. On the other hand, policy-based RL agent attempts to optimize the policy function directly without using the value function. The policy function in this case is the function that defines the action the agent selects at the given state. Finally, model-based approach focuses on sampling and learning the probabilistic model of the environment which is then used to determine the best actions the agent can take. Assuming the model of the environment has been properly learned, model-based algorithms are much more efficient than model-free ones, however, since the agent only learns the specific environment model, it becomes useless in a new environment and requires time to learn another model.

2.1 Deep Learning

RL algorithms require both value and policy function to be approximated based on sample data collected during interaction with the environment [6]. This approximation problem can be solved by representing those functions in form of deep multi-layer neural networks. A deep neural network consists of multiple layers of nonlinear processing units. The main idea behind deep learning is using the first layers to find compact low-dimensional representations of high-dimensional data whereas later layers are responsible for achievement of the task given, e.g. regression or categorical classification. All the neurons of the layers are activated through weighted connections. In order the network being capable to approximate a nonlinear transformation, a non-linear activation function is applied to the neuron output. The learning is conducted by calculating error in the output layer and backpropagating gradients towards the input layer. In regular deep neural network layer, each neuron in a hidden or output layer is fully connected to all neurons of the previous layer with the output being calculated by applying the activation function to the weighted sum of the previous layer outputs. Such layers have few trainable parameters and therefore learn fast compared to more complicated architectures, however they may suffer when dealing with spatio-temporal data such as images and time-series.

Most of the time, convolutional neural networks (CNNs) are employed for automatic extraction of low-Level features such as edges, color, gradient orientation in image related problems [7]. The main building block of CNN is the convolutional layer which calculates an integral that expresses the amount of overlap of the layer's filter as it is shifted over the input data [1]. Similarly to the previous case, the integral value is passed through an activation function to account for non-linearity in data. As a rule, multiple convolutions are performed on the input, each using a different filter. Resulting feature maps are then stuck together and become the final output of the convolution layer. CNNs can be employed to handle data of any dimension, since the result of the convolution operation is always a scalar. After a convolution operation, pooling can be performed to reduce the dimensionality of the output. The most common type of pooling is max pooling which takes the max value in the pooling window. Contrary to the convolution operation, pooling has no parameters. It slides a window over its input, and simply takes the max value in the window. CNNs usually consist of several convolutional layers mixed with few pooling layers followed by standard fully-connected layers. Stacking multiple convolutional layers allows one to learn both basic features as well as higher level representations to recognize objects in different shapes and positions.

[1] Many machine learning libraries implement cross-correlation, but call it convolution.

Temporal dependencies in the data can be extracted with the help of recurrent neural networks. In distinction to a fully-connected layer, a recurrent layer assumes that input data samples are time-series. To accommodate this fact, each recurrent layer has its own internal state the value of which is calculated based on the state value of the previous sample. The output of the recurrent layer is essentially an activation of the weighted sum of the previous layer outputs added to the weighted sum of the previous state values. During the learning process, derivatives are backpropagated through time, all the way to the beginning or to a certain point. All the derivatives multiply the same weight matrix which may result in either infinite or vanishing update values. While gradient exploding can be fixed by straight-forward clipping [8], dealing with gradient vanishing requires an intelligent control over the state via forget gates [9, 10]. It is worth mentioning that convolutional layers can also be employed to extract features from temporal data. For example, in context of reinforcement learning, DeepMind's Q-network that teaches itself to play Atari games, stacks last four frames of the historical data to produce an input for the CNN [2].

2.2 Deep Reinforcement Learning

Deep Q-Network (DQN) proposed in [2] presents the first deep value-based RL model to learn control policies directly from high-dimensional sensory input. In particular, the original DQN algorithm used the images shown on the Atari emulator as input, using convolution neural network to process image data. The value function of each action at each time step, Q-function, is evaluated using Bellman equation [11] that is proven to converge to an optimal value [12]. DQN uses a deep neural network as the function approximation to estimate the value function. The network is trained by minimizing the loss function, which is essentially the difference between the value of Q-function predicted in that particular time step and the target value function that is evaluated using the real reward value obtained from the environment. There are two main issues with using deep Q-networks. Fist, deep learning assumes data samples to be independent, however, the training data for the Q-network are collected by the sequence correlated states which led out by actions chosen. Second, the collected data distributions are non-stationary, since the agent keeps learning new strategies at every iteration. To overcome these issues, two following mechanisms can be employed: experience replay and freezing the target. The later requires constructing two neural networks with the same structure, but different weight values. One of these networks is updated online at each training step, while weights of the another one are kept frozen for a fixed amount of iterations. To calculate value of the loss function during training, Q-function value is predicted with the first neural network, while the target is evaluated using the second one. Weights of the target network are periodically updated with weight values of the online network. Since the evaluation of the target value uses the old parameters whereas the predicted value uses the current parameters, this can break the data correlation efficiently. Moreover, this mechanism allows one to avoid policy oscillations caused by rapid changes of the Q-function. The second mechanism, experience replay, is the method which aims to to break correlations between data samples by accumulating a buffer of experiences from many previous episodes and training the agent by sampling mini-batches of experiences randomly from this buffer uniformly at random.

DQN is proven to be powerful tool that could deal with problems involving low-dimensional, discrete observations and actions. However, in real world engineering applications, not only the dimensionality of both states and actions can be high, both the states and actions are often continuous. When there are a finite number of discrete actions, Q-function maximization poses no problem, because we can just compute the Q-values for each action separately and directly compare them, whereas when the action space is continuous, solving the resulting optimization problem can be non-trivial. In principle, DQN can be used to solve such problems using a set of applicable tools ranging from adaptive discretization and function approximation approaches to macro-actions or options [6]. However, this approach by design can lead to non-optimal solutions. Deep Deterministic Policy Gradient (DDPG) has been developed specifically for dealing with environments that operate in continuous action spaces [13]. Similarly to DQN, DDPG uses the Bellman equation to learn the Q-function which is in turn used to derive and learn the optimal policy. In addition to the value-function in DDPG, the second neural network that represents the agent's policy is employed to learn a deterministic policy which for every given state of the environment returns the action that maximizes the Q-function. Assuming the Q-function is differentiable with respect to action, gradient ascent is performed with respect to policy parameters only to find the action that maximizes the Q-value. Both DQN tricks experience replay and freezing the target can be also used with DDPG. The only difference is that in DQN the target network is just copied over from the main network every some-fixed-number of steps, whereas in DDPG-style algorithms, the target network is updated once per main network update by Polyak averaging. There is also difference in the exploration process. DQN often relies on ϵ-greedy policy meaning that ϵ percent of the time action is chosen completely at random. Value ϵ is usually set to decay over time to allow the algorithm to concentrate more on exploiting the best strategies that it has found. On the other hand, DDPG explores by adding a mean-zero Gaussian noise to the actions or policy parameters at the training stage.

Concerning deep policy-based RL, a neural network is used to estimate the agent's policy. The loss function is simply the negative logarithm of the probability of the action taken multiplied by the cumulative reward obtained from the environment. Updating the policy network parameters by taking random samples may introduce high variability in log probabilities and cumulative reward values, because trajectories during training can deviate from each other at great degrees. This results in unstable learning and the policy distribution skewing to a non-optimal direction. One way to

reduce variance and increase stability is subtracting the value function from the cumulative reward. This allows one to estimate how much better the action taken is compared to the return of an average action. The value function can be estimated by constructing the second neural network, which estimates the environment's state value in the manner similar to DQN. The resulting architecture is called advantageous actor-critic (A2C), where the critic estimates the value function, while the actor updates the policy distribution in the direction suggested by the critic [14]. To improve stability of the learning even further, trust region policy optimization (TRPO) relies on minimizing a certain surrogate objective function that guarantees policy improvement with non-trivial step sizes [15]. TRPO uses average KL divergence between the old policy and updated policy as a measurement for a region around the current policy parameters within which they trust the model to be an adequate representation of the objective function, and then chooses the step to be the approximate minimizer of the model in this region. Although TRPO has achieved great and consistent high performance, the computation and implementation of it is extremely complicated. The current state-of-art algorithm policy optimization (PPO) attempts to reduce the complexity of TRPO implementation and computation by tracing the impact of the actions with a ratio between the probability of action under current policy divided by the probability of the action under previous policy and artificially clipping this value in order to avoid having too large policy update [16]. Clipped surrogate objective allows the algorithm to restrict the range that the new policy can vary from the old one. In this report, we evaluate DDPG, vanilla A2C and PPO algorithms employing different neural network architectures for policy and value functions approximation in order to train an intelligent agent to control a machine in a simulated environment where both states and actions are continuous.

3 Performance Evaluation

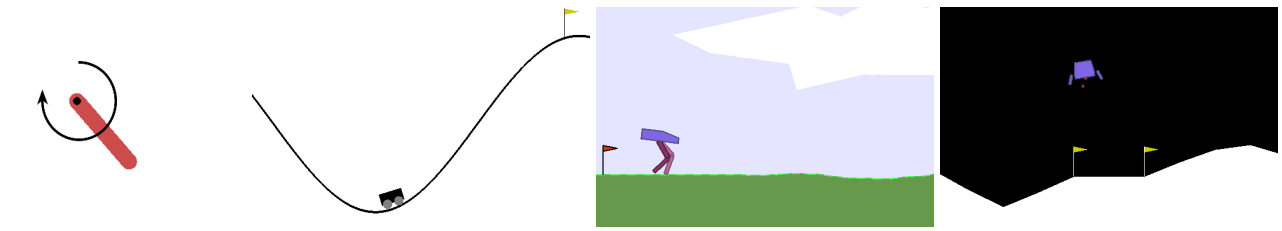
This section presents performance results of the state-of-art RL algorithms highlighted in the previous section evaluated in several virtualized environments imitating different control tasks varying from balancing a pendulum to digging a trench with an excavator. We first perform tests in simple Python-based environments to derive best combinations of a RL algorithms and a neural network model employed as policy and value function. After that, we show how can the algorithms tested be applied to solve three realistic problems using more sophisticated simulators including ROS2 [17, 18], Unity [19] and Mevea [20].

3.1 Algorithms and policies evaluation in OpenAI gym

To evaluate performance of different RL algorithms, we use OpenAI gym that has emerged recently as a standardization effort [21]. We run multiple copies of the environment in parallel. The training process is divided into episodes. Each episode lasts a certain fixed amount of time steps, during which one of the control tasks is performed by an agent implemented using OpenAI baselines [22]. Control tasks include swinging an inverted pendulum up from a random position, driving a car up to a mountain on a one-dimensional track, moving a bipedal walker through a one-dimensional track, and landing a space ship in a two-dimensional environment (see Figure 1). Table 1 shows more detailed information about the OpenAI gym environments used for the algorithms' evaluation.

Table 1: OpenAI gym environments.

Parameter	Environment			
	Pendulum	Mountain car	Bipedal walker	Lunar lander
Number of environment copies	16			
Time steps per episode / per environment	125 / 2500			
State space	[-1, 1] x [-1, 1] x [-8, 8]	[-1.2, 0.6] x [-0.07, 0.07]	24 * (-∞, ∞)	8 * (-∞, ∞)
Action space	[-2, 2]	[-1, 1]	4 * [-1, 1]	2 * [-1, 1]
Reward function	(-∞, 0)	(-∞, 100)	[-100, 300]	[-100, 100]



(a) Pendulum (b) Mountain car (c) Bipedal walker (d) Lunar lander racing

Figure 1: OpenAI gym environments.

We evaluate three following RL algorithms: DDPG, A2C, and PPO. Configurations of neural network models used for policy and value function approximation are presented in Table 2. In case of CNN architectures, we employ the same approach as in [2], i.e. we stack four latest environment observations and send it to the input layer of the neural network. For on-policy algorithms, A2C and PPO, we also test LSTM networks. In this case, we track the hidden state of the LSTM cells, i.e. information available to the agent includes not only the current state of the environment, but also information about other all other states encoded in particular way. It is worth noticing that we use the same architecture to evaluate both the policy and value function, i.e. all the trainable parameters except for the ones in the output layers are shared among the actor and the critic.

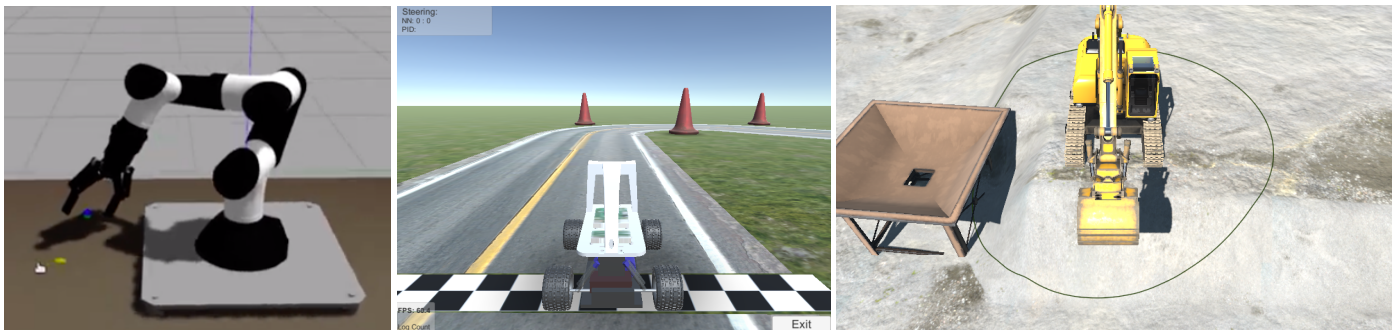
Table 2: Neural network architectures.

Name	Layers	Trainable variables
MLP2x64	2 fully-connected layers, 64 neurons per layer	4547
MLP2x256	2 fully-connected layers, 256 neurons per layer	67331
MLP2x1024	2 fully-connected layers, 1024 neurons per layer	1055747
MLP3x64	3 fully-connected layers, 64 neurons per layer	8707
MLP3x256	3 fully-connected layers, 256 neurons per layer	133123
MLP3x1024	3 fully-connected layers, 1024 neurons per layer	2105347
CNN2x64	1 convolutional layer and 1 fully-connected layer, 64 neurons per layer	5123
CNN2x256	1 convolutional layer and 1 fully-connected layer, 256 neurons per layer	69635
CNN2x1024	1 convolutional layer and 1 fully-connected layer, 1024 neurons per layer	1064963
LSTMx64	64 long-short term memory cells	17539
LSTMx256	256 long-short term memory cells	266755

Evaluation results are presented in Table 3, while the training progress is shown in Figures 4 and 5 one can find in the Appendix. In our experiments, we aim to find the methods that not only perform consistently good, but also converge in reasonable amount of time in order to be able to train in real-time simulators or even real-world environments. For this reason, we first run all the algorithms with different policies for a particular environment, then we find the minimum and the maximum value of the reward function averaged per all steps in all the environment copies in the current epoch. Then we calculate the threshold value as the sum of the minimal reward value and 95 % of the difference between the minimal and the maximal value, and count the number of steps required for the current algorithm to reach this threshold. The first observation we can make is that RL agents are capable of learning to act in all the environments tested. Moreover, PPO consistently provides good results in terms of both average reward and convergence speed. Using either convolutional or LSTM layers does not seem to make significant difference compared to traditional MLP models. In case of LSTM, this can be caused by the increased number of the trainable parameters and therefore the model requires more iterations to converge. Speaking of CNNs, convolutional networks do not seem to be a great choice for dealing with temporal pattern recognition, however they are perfect for extracting spatial features as we will show in the next subsection by employing convolutional layers to learn from raw images.

3.2 RL algorithm application in realistic simulators

Based on the results obtained in the previous subsection, we select PPO algorithm with MLP and LSTM policies for testing applicability of reinforcement machine learning to the realistic control tasks. In this subsection, we evaluate this in three following real world simulators of a robotic hand, a donkey car and an excavator (see Figure 2). It is worth noticing that all the simulators used in this study employ OpenAI gym as a front-end that communicates with the more complicated software via TCP protocol. In case of the robotic hand and donkey car, such implementations have been found in open access [23, 19]. In case of the excavator, we implemented the environment by ourselves.



(a) ROS2 robotic hand (b) Unity donkey car (c) Mevea excavator

Figure 2: Realistic simulator environments.

Table 3: Evaluation of RL algorithms with OpenAI gym. The first number is the average reward of the agent after the training, the number in the brackets corresponds to the amount of episodes required to reach 95 % of the score which is maximum among all agents for the environment. None in the brackets means that the reward desired has never been reached during the training.

Algorithm	Network	Maximal reward (episodes required to reach 95 % of the maximal reward)			
		Pendulum	Mountain car	Bipedal walker	Lunar lander
DDPG	MLP2x64	-1.10 (**1**)	-0.07 (None)	0.10 (None)	0.96 (45)
	MLP2x256	**-1.08** (4)	-0.08 (None)	0.05 (None)	0.89 (None)
	MLP2x1024	-1.09 (2)	-0.09 (None)	-0.04 (None)	0.62 (None)
	MLP3x64	-1.17 (4)	-0.05 (None)	-0.05 (None)	1.02 (28)
	MLP3x256	-1.20 (2)	-0.07 (None)	-0.08 (None)	0.77 (None)
	MLP3x1024	-1.25 (4)	-0.08 (None)	-0.08 (None)	0.37 (None)
	CNN2x64	-1.17 (4)	-0.01 (8)	0.14 (None)	1.03 (30)
	CNN2x256	-1.13 (4)	0.00 (17)	0.04 (None)	1.01 (22)
	CNN2x1024	-1.13 (10)	0.00 (37)	-0.07 (None)	-1.61 (None)
A2C	MLP2x64	-1.96 (None)	-0.01 (None)	-0.43 (None)	0.23 (None)
	MLP2x256	-1.33 (46)	-0.01 (None)	-0.29 (None)	0.27 (None)
	MLP2x1024	-1.27 (39)	-0.01 (None)	-0.13 (None)	0.34 (None)
	MLP3x64	-1.24 (28)	-0.01 (None)	-0.21 (None)	0.22 (None)
	MLP3x256	-1.27 (38)	-0.01 (None)	-0.41 (None)	0.25 (None)
	MLP3x1024	-3.77 (None)	-0.01 (None)	-0.17 (None)	0.51 (None)
	CNN2x64	-1.29 (23)	-0.01 (None)	-0.21 (None)	0.24 (None)
	CNN2x256	-1.28 (19)	-0.01 (None)	-0.28 (None)	0.31 (None)
	CNN2x1024	-1.68 (None)	-0.01 (None)	-0.48 (None)	0.28 (None)
	LSTMx64	-1.25 (24)	-0.01 (None)	-0.50 (None)	0.27 (None)
	LSTMx256	-1.28 (20)	0.00 (38)	-0.61 (None)	0.22 (None)
PPO	MLP2x64	-1.17 (32)	**0.00** (**4**)	0.21 (None)	0.97 (15)
	MLP2x256	-1.31 (47)	**0.00** (**4**)	0.32 (13)	**1.34** (17)
	MLP2x1024	-1.13 (9)	**0.00** (**4**)	0.25 (32)	1.10 (11)
	MLP3x64	-1.12 (24)	**0.00** (**4**)	**0.35** (16)	1.03 (33)
	MLP3x256	-1.12 (12)	**0.00** (**4**)	0.30 (11)	1.23 (10)
	MLP3x1024	-1.17 (8)	**0.00** (**4**)	0.24 (29)	1.23 (4)
	CNN2x64	-1.18 (25)	**0.00** (**4**)	0.33 (11)	1.20 (10)
	CNN2x256	-1.16 (16)	**0.00** (**4**)	0.31 (**9**)	1.11 (14)
	CNN2x1024	-1.17 (9)	**0.00** (**4**)	0.18 (None)	1.28 (**3**)
	LSTMx64	-1.25 (44)	**0.00** (**4**)	**0.35** (16)	1.07 (39)
	LSTMx256	-1.22 (31)	**0.00** (**4**)	0.15 (None)	0.97 (18)

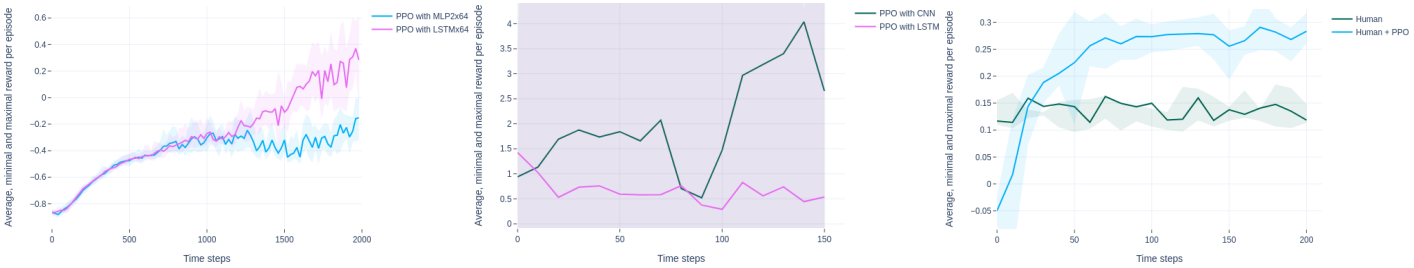
In case of the robotic hand, we set the initial position of the robot to zero for all joints and reset the robot to this initial position when the number of steps exceeds the maximum time steps for an episode. The aim of the robot is to reach a particular fixed target point. An RL algorithm generates actions that are translated into the corresponding ROS2 messages and are executed in simulation. The simulation then returns the observations and gives them to the algorithm. The reward is calculated as a negative sum of the distance of the end-effector to the target point and the difference between the end-effector orientation and the target orientation. Donkey car simulator in Unity is high fidelity simulator of a small scale self-driving car. The aim of the agent in this environment is to maximize the car's velocity while having it stay within the track region. Unlike previous simulations, in this environment the RL agent learns to take actions based on information extracted from raw pixel images taken by the front camera of the car. Those images are resized, gray-scaled and stacked together represent the current state of the environment. The car then takes continuous steering and throttle values as actions. The reward is calculated based on the cross track error which measures the distance between the center of the track and car.

Table 4: Realistic simulator environments.

Parameter	Environment		
	Robotic hand	Donkey car	Excavator
Number of environment copies	4	2	2
Time steps per episode / per environment	125 / 10000	128 / 1000	128 / 200
State space	24 * [-∞, ∞]	[0, 255] * 120x160x3	13 * [0, 1]
Action space	6 * [-π, π]	[-1, 1] x [0, 5]	4 * [0, 1]
Reward function	(-∞, ∞)	(-∞, ∞)	(-∞, ∞)
Policy	MLP2x64 / LSTM2x64	CNN2d + MLP2x64 / LSTM2x64	MLP2x64 / MLP2x1024 + MLP2x64

Mevea software allows one to achieve real-time simulation of a big manufacturing machine even in case of an extremely complicated model. In our study, we use an excavator model and aim to teach the RL agent to dig a trench in the target location. The current environment state consists of the excavator joint positions as well as the mass of the soil in the bucket. The reward is calculated as either the negative distance to the target if there is no soil in the excavator's bucket taken from a proximity to the target or the sum of the negative distance to the dumper and the mass of the soil in the bucket otherwise. In other words, we aim to maximize the mass of the ground taken from the place of the target trench and minimize the amount of the soil lost on the way to the dumper. Instead of randomly exploring strategies to perform such complicated task from scratch, we can inject prior information into an RL algorithm in order to speed

up the training process, as well as to minimize unsafe exploration behavior. For this purpose, we use data derived from several sessions of a professional excavator operator that have been recorded in advance. We then augment the data by randomly selecting the value of the target angle and generating a series of the data points that correspond to the resulting target. The resulting dataset is used to train a neural network that predicts the trajectory of the excavator joints depending on the target given. The RL agent selects actions by combining the supervised policy trained with the demonstration data with its own parametric policy.



(a) ROS2 robotic hand (b) Unity donkey car (c) Mevea excavator

Figure 3: Performance of PPO with MLP and LSTM policies in realistic simulators.

Table 4 shows the dimensions of both the observation and action spaces for each environment as well as some parameters of our experiments. As one can notice, the more complicated the environment the less the number of time steps and environment copies we are able to cope with as we have to accommodate for the increased demand of the computational resources as well as the reduced speed of the simulations. Since the donkey car learns from the raw images, we add several convolutional layers to our neural network policies to extract features. In case of the excavator, we train an additional MLP to predict moves of the excavator joints based on the demonstration data provided by a human operator. The results of the experiments are presented in Figure 3. As one can notice, RL algorithms are able to learn good policies even in complicated realistic environments. In case of the excavator, one can notice that the learning accelerates when the agents uses demonstration data to outperform the baseline policy. This however does not imply that the RL agent does better job than human operator, as it it maximizes the reward function, but fails to dig the trench properly. This result shows another challenge of applying reinforcement learning for real-world control systems, namely reward function engineering. It is worth noticing that we can define the reward simply as the amount of soil in the dumper, but this leads to sparse reward function, that require much more iterations to learn.

4 Conclusion

This report briefly highlights several state-of-art reinforcement machine learning algorithms and evaluates their performance using realistic simulation software. Despite promising results, the RL algorithms require detailed analysis in order to be applied to a concrete task since the performance depends not only on the RL model selected but also architecture of the neural network used for policy and value function approximation. In our study, we came to the conclusion that PPO is the best algorithm for the reinforcement learning in terms of both convergence speed as well as achieving consistently high reward. PPO however may struggle from the drop in the performance after completing a certain amount of iterations, therefore the model checkpoints are supposed to be saved periodically during the training in order to be able to restore the model from the best checkpoint stored if such drop takes place. We also studied how demonstration data could be used in order to accelerate the learning process and showed that such approach allowed one to outperform the baseline policy provided by the human operator.

References

[1] Y. Li. Deep reinforcement learning: An overview. arXiv preprint arXiv:1701.07274, 2017.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with Deep Reinforcement Learning. Proc. ofNIPS Workshop on Deep Learning, 2013, pp. 1–9.

[3] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor, Proc. of ICML, 2018.

[4] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. Proc. of AAAI, 2018.

[5] I. Popov, N.M. Heess, T.P. Lillicrap, R. Hafner, G. Barth-Maron, M. Vecerík, T. Lampe, Y. Tassa, T. Erez, and M.A. Riedmiller. Data-efficient Deep Reinforcement Learning for Dexterous Manipulation. ArXiv, abs/1704.03073, 2018.

[6] J. Kober, J Andrew Bagnell, and J. Peters. Reinforcement learning in robotics: A survey. The International Journal of Robotics Research, 32(11):1238–1274, 2013.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. Proc. of the 25th International Conference on Neural Information Processing Systems (NIPS), Vol. 1, 2012.

[8] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training recurrent neural networks. Proc. of Int. Conference on Machine Learning, pp. 1310–1318, 2013.

[9] S. Hochreiter and J. Schmidhuber. Long short-term memory. Neural Computation, 1997.

[10] F. Gers, N. Schraudolph, and J. Schmidhuber. Learning precise timing with LSTM recurrent networks. Journal of Machine Learning Research, 3:115–143, 2002.

[11] R. E. Bellman. Dynamic Programming. Dover Publications, Inc., New York, NY, USA, 2003.

[12] C. Watkins and P. Dayan. Q-learning. Machine learning, 8(3-4):279–292, 1992.

[13] T. Lillicrap, J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971, 2015.

[14] V. Mnih, A. Badia, M. Mirza, A. Graves, T.Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. arXiv preprint arXiv:1602.01783, 2016.

[15] J. Schulman, S. Levine, P. Moritz, M. Jordan, and P. Abbeel. Trust Region Policy Optimization. arXiv preprint arXiv:1502.05477, 2015.

[16] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347, 2017.

[17] Y. Nuin, N. Lopez, E. Moral, L. Juan, A. Rueda, V. Vilches, and R. Kojcev. ROS2Learn: a reinforcement learning framework for ROS 2. arXiv preprint arXiv:1903.06282, 2019.

[18] N. Lopez, Y. Nuin, E. Moral, L. Juan, A. Rueda, V. Vilches and R. Kojcev. Gym-gazebo2, a toolkit for reinforcement learning using ROS 2 and Gazebo. arXiv preprint arXiv:1903.06278, 2019.

[19] OpenAI Gym Environments for Donkey Car. Available at https://github.com/tawnkramer/gym-donkeycar.

[20] S. Huttunen. Tutorials for MeVEA Simulation Software. Available at https://lutpub.lut.fi/handle/10024/67972, 2010.

[21] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang and Wojciech Zaremba. OpenAI Gym.arXiv:1606.01540, 2016.

[22] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov. OpenAI Baselines. GitHub, https://github.com/openai/baselines, 2017.

[23] Gym-gazebo2. Available at https://github.com/AcutronicRobotics/gym-gazebo2.

5 Appendix: Performance of RL algorithms with different policies in OpenAI gym

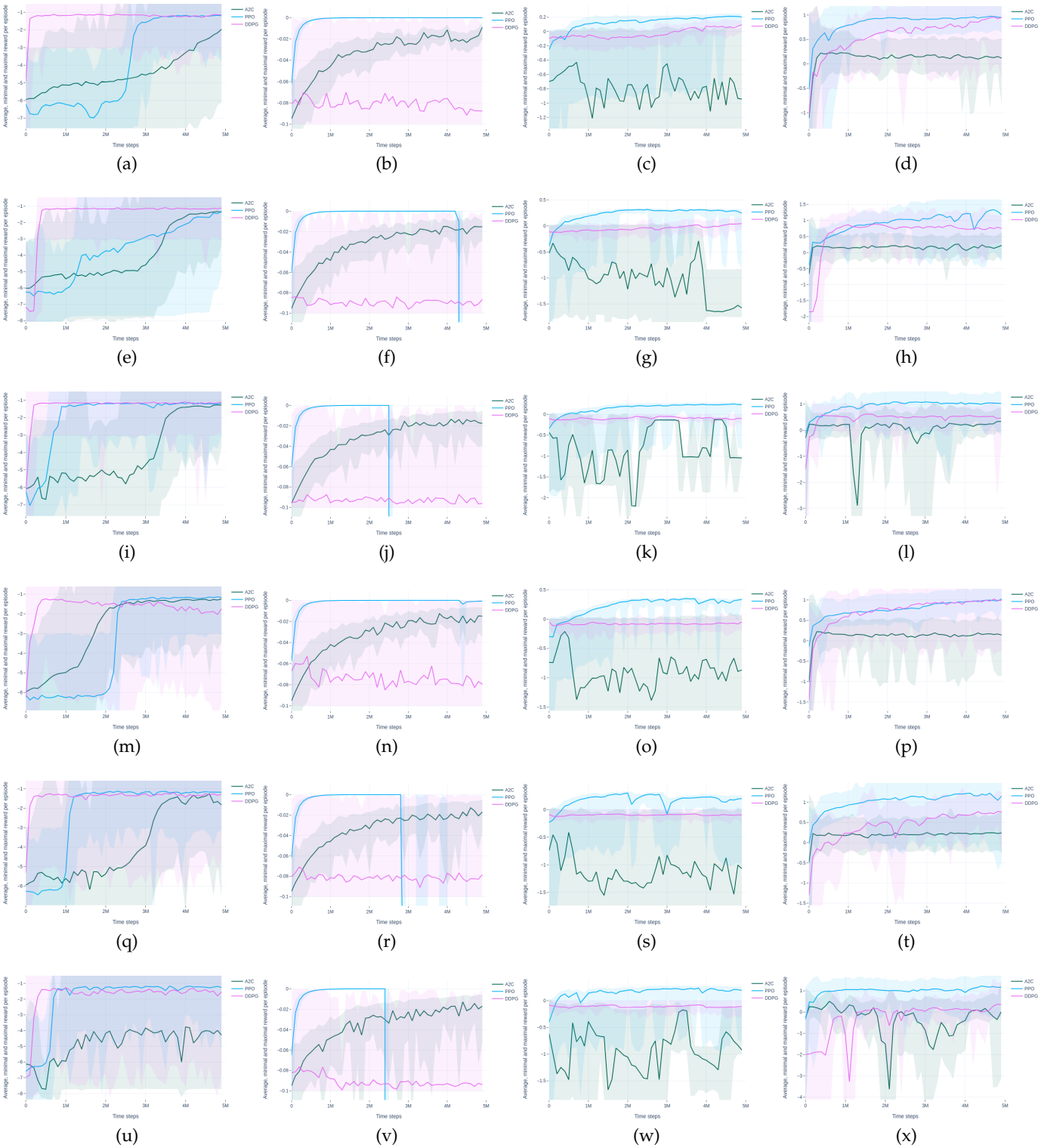


Figure 4: Performance of RL algorithms with MLP policies in OpenAI gym. PPO with either two or three layers of 64 neurons provides the most consistent results among all algorithm plus neural network policy tuples.

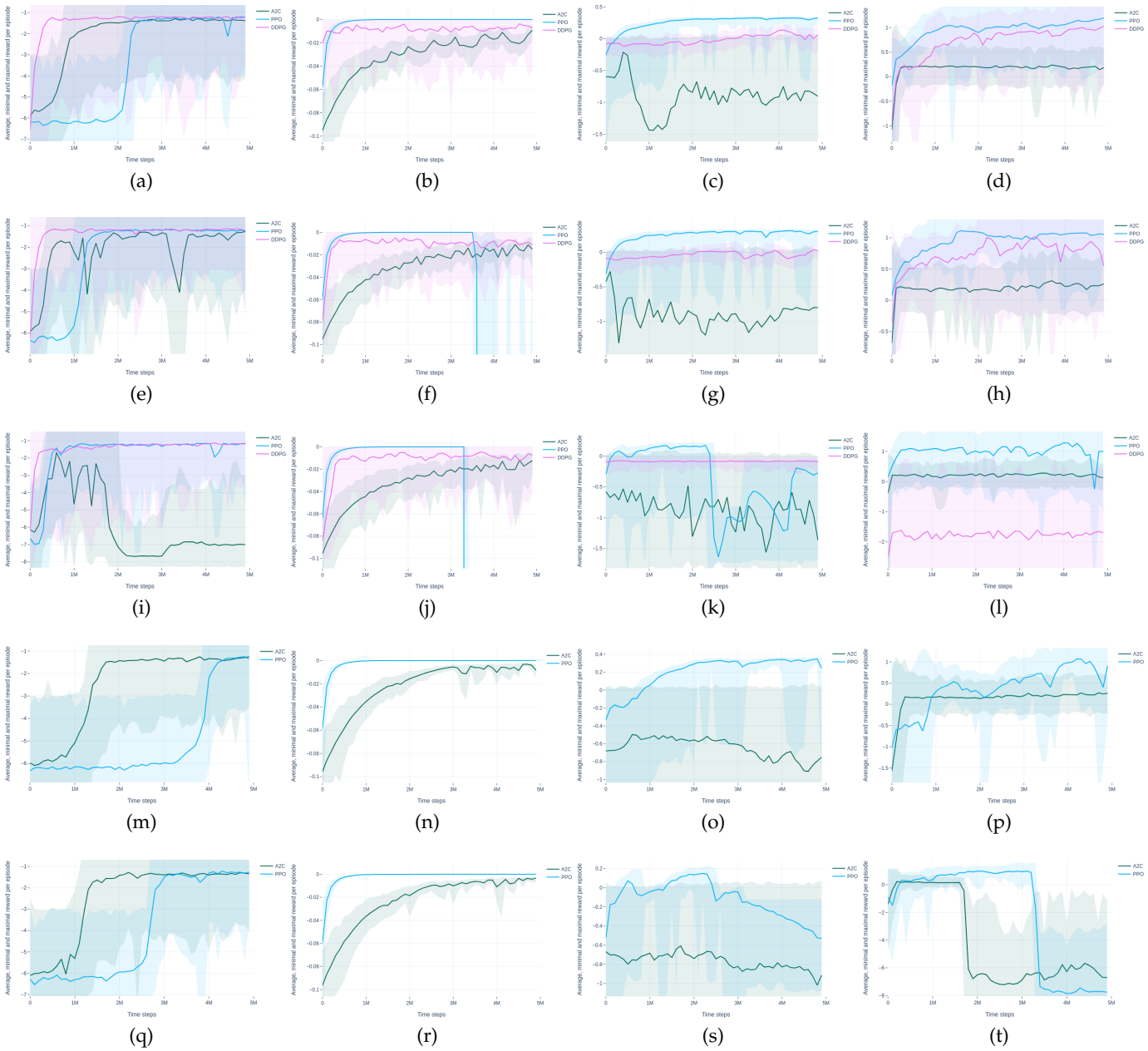


Figure 5: Performance of RL algorithms with CNN and LSTM policies in OpenAI gym. Using either convolutional or LSTM layers does not seem to provide significant improvement over straight-forward fully-connected layers.