Implementation and Experimentation of a DQN solver in Julia for POMDPs.jl

Carol Hsin
Stanford University

Abstract

The goal of this project is to implement and test a deep reinforcement learning solver for the POMDPs.jl [7] framework that returns the learned weights given a reinforcement learning problem using the Deep Q-Network training algorithm that was discussed in the Deep Mind publications regarding Atari games [17], [18]. The DQN code is contained in this project's repo, DQN.jl [10] and was implemented in Julia using Tensorflow [1] to construct the neural networks and tested using the CartPole-v0 and Pong-v0 environments in OpenAI Gym [2]. SISL's DeepRL.jl [4] and Gym.jl [5] libraries were used to ensure there would be compatibility with SISL's other reinforcement learning libraries, such as POMDPs.jl [7] and POMDPsModels.jl [6]. The implementation used the new Tensorflow.jl [16] library that wraps the Tensorflow API. Tensorboard functionality was also implemented and used for debugging the DQN. Memory and performance optimization were also implemented. The success metric was the observation of learning through the loss and score histories in addition to the performance improvements observed from OpenAI Gym's video clips over the training sessions.

1. Introduction

1.1. Description and significance

Interest in deep reinforcement learning has grown since DeepMind published [17] in which the researchers used deep reinforcement learning to train a computer to play Atari games with human level performance. However, SISL's main reinforcement learning library, POMDPs.jl [7] doesn't currently have an implemented solver using deep learning techniques and the lab members currently use Python for training DQNs (deep Q-networks), even though most of the rest of the lab's work, particularly libraries for autonomous vehicles, is in Julia because Julia is faster and more efficient than Python.

Therefore, this project addresses the need to implement a DQN solver in Julia that can be integrated into the POMDPs.jl framework and interact with SISL's DeepRL.jl [4] and Gym.jl [5] libraries that wraps the functionalities of OpenAI Gym, [2], which is the main platform containing models/environments to test DQNs. In addition, the Julia DQN needs to be compatible with DeepRL.jl because DeepRL.jl provides an interface to work with the models/environments in POMDPsModels.jl [6], which is the main library of POMDP models used in SISL's POMDPs.jl.

1.2. Background: Deep Q-Network algorithm

While deep learning approaches in reinforcement learning includes policy gradient methods in addition to deep Q-learning, we will focus only on deep Q-learning in this project. The algorithm this project implemented is the one in DeepMind's 2015 Nature paper [18]:

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

2. Approach

2.1. Design for types of problems

The most well-known uses for DQN is to train computers to play Atari games as seen in the DeepMind publication [17] in which Deep Q-learning was applied to video games, where the state is composed of the last few consecutive frames/images from the game. Therefore, from the outset of this project, the vision was to have the Julia DQN be functional for Atari games. But it is notable that the current POMDPs.jl [7] solvers cannot effectively handle images as inputs and seem to be designed to solve classical re-

inforcement learning MDP problems where the state-space is a vector as seen in POMDPsModels.jl [6]. Therefore, the DQN solver needs to be benchmarked with a classical reinforcement learning problem in which the state space is described by a vector, such as CartPole-v0 [3]. Since DeepRL.jl [4] provides an interface for the SISL libraries, we believe it is enough to make the DQN solver compatible with DeepRL.jl [4] for it to be able to be compatible with POMDPs.jl for vector based states while being able to solve problems using image states.

Thus, we will be benchmarking against two types of problems, ones in which the input is composed of a state vector and the other in which the input is composed of pixels. The former will be from the classic problem CartPole-v0 and the later will be the classic Atari game Pong-v0, both of which are available from OpenAI Gym [2].

2.2. Usability design

As observed from the algorithm in Section 1.2, there are over a dozen hyper-parameters (including learning rate, batchsize, etc) and variables (such as the environment) that will change depending on the problem. Therefore, the first ideas regarding how the user will interact with the library centered around what essential changes had to be made for each problem for the DQN to be able to run, so the hyper-parameters were ignored at first since their changes are only necessary for the DQN to perform well through tuning, as opposed to being able to perform at all. Therefore, three main required inputs are the functions below (which needs to be based on the environment that is passed in), the details of which are in this project's Gihub repo [10].

```
s, readout, wgts = createNetwork(ACTIONS)
s_t, r_t, is_terminal, s_0 = frame_step(action, prev_state)
s_t = preprocess(x, prev_state)
```

The user may also need to implement the `runDQN` function, but since the current `runDQN` function makes calls to the SISL library's DeepRL.jl and Gym.jl library, there should not much of a need for changes if the user's environment is from POMDPsModels.jl.

2.3. DQN implementation approach

This project uses Google's Tensorflow [1], since unlike the other open source deep learning libraries [12] [19], Tensorflow is produced by professional engineers, which makes the library a bit more stable, efficient and includes extra debugging functionality such as Tensorboard. But Google has not released a Julia Tensorflow API, in fact, the main API is in Python. To use Tensorflow in conjunction with Julia, this project uses Tensorflow.jl [16], which is an open-source and recently developed wrapper around TensorFlow's Python API that uses PyCall.jl [13] to make calls to Tensorflow's Python API directly, resulting in a Julia library that has syntax that is very close to the Python API. At the moment, even though Tensorflow.jl is relatively new and probably not quite stable, it is the cleaner way of implementing the DQN solver as opposed to having a library that is a mix of Python and Julia through using PyCall.jl [13] directly.

The first step was to implement a functioning DQN in Julia using Tensorflow.jl and SISL's OpenAI Gym Julia wrapper [5] (for compatibility with SISL's other libraries). OpenAI Gym's CartPole-v0 [2] problem environment was used as a benchmark for testing the performance of and to debug the DQN code since CartPole-v0 is a simple and classic RL problem. This step also involved adding functionality so the user can use Tensorboard to debug their DQN by observing the loss history along with graphs of how the network weights changes with each training step. This was also essential in the debugging of the DQN.

Cart-Pole-v0 has a simple state-space where the main variables are given in a 4-dimensional vector. But since the goal of this project was to implement a DQN library that would also work with image states like that from Atari games as was used in the DeepMind paper [18], the next step was to generalize the DQN so it would work with the OpenAI Gym environment for the Atari game Pong-v0 [2]. At this step, efficiency and performance optimizations were necessary as well as a rethinking of the success metric since Pong-v0 typically takes days to train [14].

2.4. Success metrics

One metric a proof of correct implementation was based on the DQN performance on CartPole-V0, which is achieving an episode length of 200, which was the same metric used by OpenAI Gym for measuring success.

A more general metric for both types of games was using Tensorboard to see that the loss was decreasing and the score was increasing over the training session. The loss and score history was the main metric for a functional and well-implemented DQN because the training can take days, so there was not enough time to iteratively tune the hyper-parameters for each training session. In addition, if a high score was desired, that should be the result of neural network architecture tuning and hyper-parameter experiments done by the user for their specific problem since the main objective of this project is not the tuning of hyper-parameters to optimize the scores for CartPole-v0 and Pong-v0. To have the loss history decrease and score increase was enough to prove that the DQN library can be generalized to solve reinforcement learning problems for simple state-spaces (like CartPole-v0's 4-D vector) and image state-spaces (like Pong-v0 RGB image states).

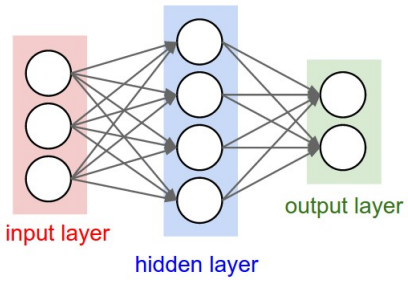
In addition, having the DQN generalize to both problems served as proof of the functionalities of Q-network weight saving/loading and Tensorboard debugging, which is essential to the usability of the DQN code.

3. CartPole-v0 Experiments

3.1. Neural network architecture

Since CartPole-v0 is a simple problem, we figured a simple one hidden layer network will do. In the project repo [10], we implemented a `createNetwork` function in the file `net1.jl` that creates the CartPole-v0 network based on the input dimension (the size of the state vector), the number of actions (`ACTIONS`), how large the user wants the hidden dimension to be `hidden\_dim` and also a string prefix for Tensorboard debugging purposes. The architecture is variable based on inputs to the function, but matches the one-hidden layer neural network diagram in Figure 1 very closely.

Figure 1: Neural network with one hidden layer.[15]



3.2. First DQN implementation

The replay memory was implemented as a type that was designed to function like a circular array. Two functions were implemented to allow for pushing new experience onto the replay memory and to be able to sample randomly from the replay memory.

The states for CartPole-v0 did not need processing so the preprocessing function did nothing and the framestep function just returns (with minor changes) variables from the OpenAI environment's step function. These functions can be viewed in the project repo [10] in the CartPole-v0 example.

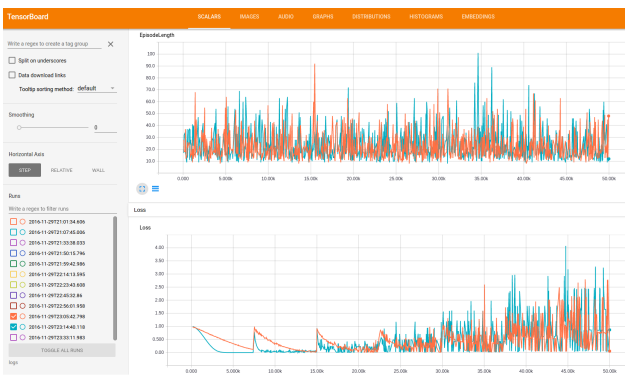
The environment and video monitoring is initialized in `runDQN` along with the main Q-network and the target Q-network. The DQN algorithm in Section 1.2 is implemented in the `trainNetwork` function, which is heavily commented as a form of documentation. Like in [18], the minibatch sizes were 32 and the behavior policy during training was ϵ-greedy with ϵ annealed linearly based on the exploration hyper-parameter from 1.0 to a final ϵ that ranged from 0.05 to 0.1, depending on the experiments run.

3.3. DQN tuning and debugging with Tensorboard

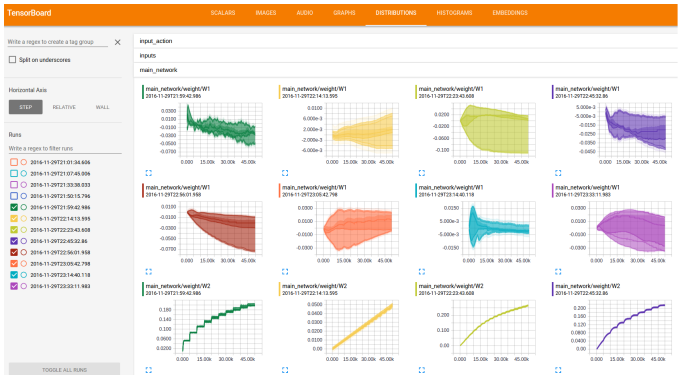
After the DQN was able to run with no errors, it was allowed to run for a few test sessions, but since there was no improvement, Tensorboard was used to debug the network. The DQN was run for multiple sessions and we also kept track of the score, which is the total number of frames from an initial state to a terminal state (200 frames or failure) as a indicator of how well the DQN was performing. The score and loss plots for two typical training session are shown in Figure 2a. As you can see in the sidebars, multiple independent training sessions were run to tune the hyper-parameters (learning rate, discount factor, target update frequency, etc.), but none were successful. The two sessions in Figure 2a shows how the target update frequency parameter changed the loss since the loss spikes with each target network update. The weights of multiple sessions were also examined and shown in Figure 2b and some of the plots seem to indicate the weights may be getting too small, but a TA advised that it does not seem like the gradients are vanishing and to focus only on the loss history for debugging.

Since the Tensorflow.jl library is rather new, to ensure that the problem was not in the library, the DQN was im-

Figure 2: Debugging DQN with Tensorboard episode length (score) and loss plots on the left and weights plots on the right for multiple independent training sessions



(a) Score and loss plots show DQN not learning

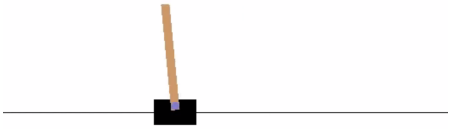


(b) Graph of the weights of each training session

plemented in Python, but still without success. After some forum searching, it seems that the rewards were not normalized to be in the range $[-1, 1]$ as was done in the DeepMind paper [18], which effects the stability of the DQN, so after the rewards were divided by 200, the DQN was able to learn.

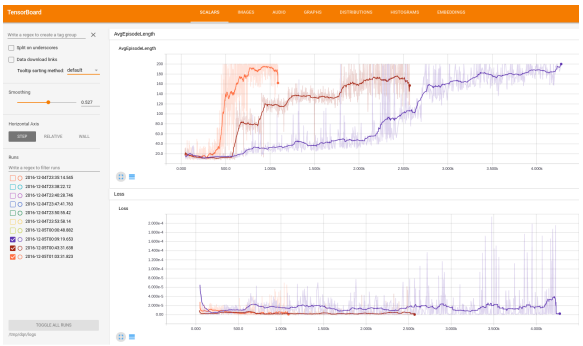
3.4. Success metric for DQN for vector states

Figure 3: A video of clips from a successful CartPole-v0 training session from this project is available in [9]



As shown in Figure 4, the DQN was able to learn and successfully keeps the pole balanced for 200 frames. The learning rate was decreased and the target update frequency was increased for the sessions shown in Figure 4, changes which improved the speed at which the DQN was able to successfully learn.

Figure 4: CartPole-v0 score and loss plots shows learning



4. Pong-v0 Experiments

4.1. Convolutional neural network

We used a convolutional neural network architecture for pixel-based problems that was basically the same as the version of the network DeepMind used in [18] except we made it smaller to increase learning speed. The details are shown in Table 1.

4.2. Implementing/Debugging DQN for Pong-v0

Unlike CartPole-v0, Pong-v0 is a video game and the states are four consecutive images that have been scaled down, gray scaled and stacked together for an input size of 80×80. Therefore, the preprocessing function is more complicated, but the framestep function remains the same. The rest of the DQN code was updated to use a 2D state shape instead of a 1D state dimension as seen in [10]. The rewards were still divided by 200 as a way to keep them in the range $[-1, 1]$.

Table 1: Layer specifications for convnet1.

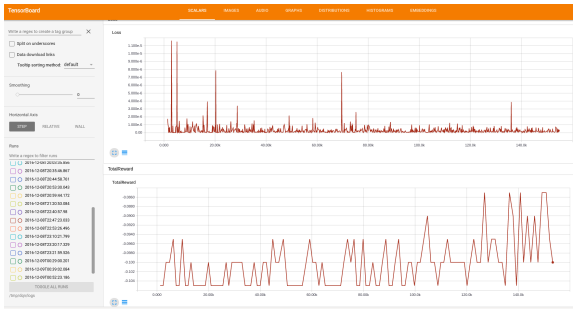
Layer	Dim	Stride	# filters
Input	80x80x4		
Conv1	8x8	4	32
Pool1	2x2	2	
Conv2	4x4	2	64
Conv3	3x3	1	64
Dense	512		
Output	# ACTIONS		

The DQN was able to run with very minor changes and was set to run overnight, but it was discovered the next morning that the program had crashed very quickly since the size of the replay memory was set at $590,000$, which is much smaller than the 1 million used by the DeepMind paper [18], but still too big. To do the computations, assuming the replay memory were to be full and considering that most of the space is taken by the two $80 \times 80 \times 4$ tensors for the current and next states, then the math shows that the space required would be over $80 \times 80 \times 590000 \times 2 \times 4\ bytes \simeq 120\ gigabytes$. The computer the project was using to train the DQN could not support this much space usage and the program crashed. Thus, the replay memory size was scaled down to 20000 for Pong-v0 and the program was allowed to run uninterrupted for the rest of the day.

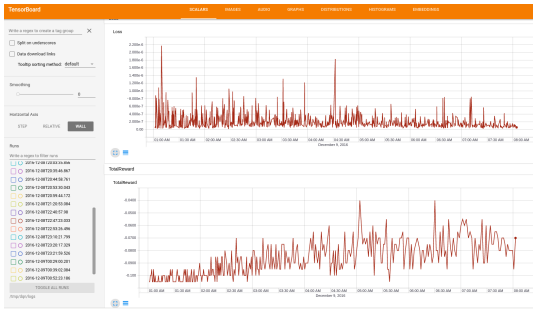
Weight saving and loading was also added at this stage since Pong-v0 actually takes a long time to run, so if the program were to crash, much time would have been wasted. This was not a problem with CartPole-v0 since that was such a simple problem it only took about an hour or less to train. In addition, the main goal of this project was to train and return the weights, so it might as well be done at this step.

When the DQN was checked after about 8 hours, the DQN performance was very poor as seen from the video clips and also from the loss and score plots in Figure 5a. Advice was sought and the insight was that each Pong-v0 episode ends when one of the players win 20 games, which can take a very long time. Therefore, each episode would be very long, but since we are only training per episode, very little training is actually completed in several hours of runtime. Therefore, we need to train by frame instead of by episode if we want faster improvement, as a comparison, 38 days was used in the DeepMind paper [18]. The

Figure 5: Tensorboard loss and score plots for Pong-v0, each session was run for about 8 hours



(a) DQN not learning and shows poor performance



(b) DQN learning is successful

DQN was run again with these changes, but we stopped it after a few episodes because it was so slow and efficiency optimizations were needed.

4.3. Efficiency and performance optimizations

The DQN was training too slowly and the advice was to use the Julia type system since even though Julia can infer types like Python does, if the types were actually given, there will be speed improvements.

Another issue that an expert noticed was that the code was reallocating the batch memory at each training step. At that time, the `sample` function from ReplayMemory.jl was used to return a copy of a batch of random memories and then another function `unpack_memory(minibatch, BATCHSIZE)` was used to turn that array of arrays into the four arrays `s_j_batch, a_batch, r_batch, s_j1_batch` required in the algorithm in Section 1.2.

Thus, another custom type `ExperienceBatch` was implemented to reduce the need to infer types and so that minibatch memory allocation would happen once. The custom type `ReplayMemory` also needed to be changed so the field were all typed instead of using Julia's `Any` type, which required the changing of all functions that used `ReplayMemory`. To prevent double copying, `sample!` was changed to take as input the batch arrays `s_j_batch, a_batch, r_batch, s_j1_batch` and write directly to them. Then the `trainNetwork` function was changed to reflect these optimization changes. The difference between the old and new implementations are shown in Table 2.

Table 2: Optimization results

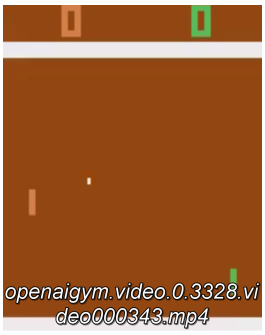
	Mem allocs per frame	Minutes for 3 episodes
old	13 MB	1:57
new	6.7 MB	1:38

After these changes, the DQN was allowed to run overnight and was able to successfully learn as shown in Figure 5b.

4.4. Success metric for DQN for image states

As shown in Figure 5b, the DQN was able to learn and while the score is not at the level seen in the DeepMind paper [18] or in other trained DQNs such as [14], the loss and score plots show that learning is occurring and the DQN would likely improve if given more time, however, the performance of [18] required over 38 days and that of [14] was over 3 days. It would be highly unreasonable to hope for better performance in several hours with very little tuning of hyper-parameters. A video of a collection of clips from the Pong-v0 training are also available in [11], which shows that learning is happening since the DQN is hitting the ball more often over time.

Figure 6: A video of clips from the successful Pong-v0 training session from this project is available in [11]



5. Discussions and Conclusions

The discussion of much of the experiments and results were elaborated upon in the previous sections. The project accomplished the task of implementing a DQN library in Julia that interfaces with SISL's DeepRL.jl [4] and Gym.jl [5]. The DQN satisfied the success metrics for CartPole-v0 and Pong-v0 from the OpenAI Gym environments [2], which meant this project satisfied the goal of implementing a DQN that would work with vector state inputs as well

as image state inputs. In addition, Tensorboard capabilities were added and the overall code was performance optimized. The weights are also returned as was the goal of this project.

Thus, the project succeeded in its main goal, which was to implement the core functionality of a DQN solver, which takes in the user's network and frame step functions and returns the learned weights. What is left to do is to implement the solver interface with POMDPs.jl [7]. But such a solver would need to export a set of required functionalities such as that in QMDPs.jl [8]. To integrate this project's repo, DQN.jl [10], into the POMDPs.jl framework, a format like the following would be required.

```
using POMDPs
POMDPs.add("DQN")
using DQN

# initializes the Solver type
solver = DQNSolver(create_network,
                   framestep,
                   preprocess,
                   env)
# saves and returns the trained weights
weights = solve(solver)

# get a state from the Tensorflow graph
state = create_state(solver)
# get action
a = action(weights, solver, state)
```

The main difficulty in doing this is that unlike the other solvers, a solved DQN policy would just be the weights and in order to get an action, we need to start a Tensorflow session to load the weights into the graph with which we can feed in a state, from which we can then get an action. It would also be troublesome to get the value function for most types of problems, particularly if the input are stacked images. From discussion with the main SISL researcher responsible for POMDPs.jl, Maxim Egorov, implementing the core DQN functionality that only solved for the weights would still be "extremely useful". Therefore, that became the goal of this project, which was accomplished.

6. Documentation

The documentation for this project follows the pattern of the other solvers in POMDPs.jl, so the installation and usage instructions will be in the README.md while examples/tutorials were implemented in Jupyter notebooks within the repo. The code itself is also heavily commented and with variables and functions appropriately named as a method of further documentation.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016.

[3] J. Cooper. Cartpole v0 overview. `https://github.com/openai/gym/wiki/CartPole-v0`, 2016.

[4] M. Egorov. Deeprl.jl. `https://github.com/sisl/DeepRL.jl`, 2016.

[5] M. Egorov. Gym.jl. `https://github.com/sisl/Gym.jl`, 2016.

[6] M. Egorov. Pomdpmodels.jl. `https://github.com/JuliaPOMDP/POMDPModels.jl`, 2016.

[7] M. Egorov. Pomdps.jl. `https://github.com/JuliaPOMDP/POMDPs.jl`, 2016.

[8] M. Egorov. Qmdp.jl. `https://github.com/JuliaPOMDP/QMDP.jl/blob/master/src/QMDP.jl`, 2016.

[9] C. Hsin. Cartpole-v0 dqn clips from one training session. `https://youtu.be/fDY96bwKw3M`, 2016.

[10] C. Hsin. Dqn.jl. `https://github.com/chsin/DQN.jl`, 2016.

[11] C. Hsin. Pong-v0 dqn clips from one training session. `https://youtu.be/_toBTIcEUpo`, 2016.

[12] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[13] S. Johnson. Pycall.jl. `https://github.com/JuliaPy/PyCall.jl`, 2016.

[14] A. Karpathy. Comment section of deep reinforcement learning: Pong from pixels. `http://karpathy.github.io/2016/05/31/rl/`, 2016.

[15] A. Karpathy. Neural networks part 1: Setting up the architecture. `http://cs231n.github.io/neural-networks-1/`, 2016.

[16] J. Malmaud. Tensorflow.jl. `https://github.com/malmaud/TensorFlow.jl`, 2016.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[18] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[19] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.