Implementations of synchronous and asychronous DQN

Christopher Beckham
Montreal Institute of Learning Algorithms
University of Montreal
`christopher.j.beckham@gmail.com`

Abstract

My project for COMP767 was to try implement DQN and its asynchronous variant. Modest results were only achieved for DQN.

1 Introduction

Recently, deep learning has become very popular in the machine learning community, achieving state-of-the-art performance in many domains, such as computer vision, audio recognition, generative modelling, and many others. The huge success of deep learning can be attributed to the fact that it learns features in a hierarchical fashion, where the features become more abstract (and powerful) as one progresses deeper into the network. Deep learning has also found its place in reinforcement learning, where linear function approximators have been replaced with their deep counterparts, also achieving state-of-the-art performance. An example of this is in the work by Mnih et al. [1, 2], where a convolutional neural network is used in conjunction with Q-learning to train an agent to play Atari games at human-level performance, solely from the raw pixels. In this report, I try my hand at implementing this work, and also the asynchronous (faster) Q-learning counterpart presented in [3].

1.1 Q-learning

In the case of Q-learning, we wish to estimate the function $Q^*(s, a)$, where Q^* denotes the optimal Q-function, such that we can choose the optimal action at each state using our policy $\pi(a|s) = max_{a'} Q^*(s, a)$. Traditionally this is done through an iterative process using the Bellman equation:

$$Q_{t+1}(s, a) = \mathbb{E}_{s'} \Big[r + \gamma max_{a'} Q_t(s', a') | s, a \Big], \tag{1}$$

where $Q_t \to Q^*$ as $t \to \infty$. In the case of large state and/or action spaces however, it is impractial to keep a Q-value for every state-action combination. In the case of (differentiable) function approximation, we can instead approximate $Q(s, a)$ with $Q(s, a; \theta)$, where θ parameterises the function. Then, we can find parameters θ by minimising the following with gradient descent at each iteration i:

$$L(\theta_t) = \mathbb{E}_{(s,a,r,s') \sim D} \Big[(y - Q(s, a; \theta_t))^2 \Big], \tag{2}$$

where $y = r + \gamma \ \max_{a'} Q(s', a'; \theta_{t-1})$. In the case of deep Q-learning, we let θ parameterise some form of deep network, and in our case, this is a deep convolutional network.

Note that in practice, for the sake of stability [2], the deep Q-function used to compute y is not parameterised by θ_{t-1} but by some θ', where θ' is a much older version of θ_t, e.g, $N = 10,000$ time steps in the past. This means that every N time steps we set $\theta' = \theta_t$.[1]

[1]This puts the optimisation more on par with what is seen in supervised learning, where y is completely fixed and does not change.

2 Experiments and Results

The architecture of the network employed is very similar to the network described in [1], and is described in Table 1. The main difference is that we also utilise batch normalisation [4], since it has been empirically shown to stabilise training of deep nets. Also note that the input to the network takes four channels: these are the last four frames observed in the emulator environment, which will be elaborated on shortly.

Table 1: Description of the convolutional architecture used in experiments. For ConvLayer, WxHsS@F = filter size of dimension W x H, with stride S, and F output feature maps. Dense (N) is a fully-connected layer with N output units (denoting the size of the action space). This network contains a total of 538,566 learnable parameters.

Layer	**Output size**
Input (4x80x80)	4 x 80 x 80
ConvLayer (8x8s4@16)	16 x 19 x 19
BatchNormLayer + ReLU	16 x 19 x 19
ConvLayer (4x4s2@32)	32 x 8 x 8
BatchNormLayer + ReLU	32 x 8 x 8
Dense (256) + ReLU	256
Dense (8) + Softmax	8

We use the OpenAI Gym [5] to provide the Atari environment used to train the network on Pong. This framework provides a simple and easy-to-use API with a wide variety of games. As is typical in reinforcement learning, in this framework an episode consists of a trajectory $\{a_0, r_0, x_1, a_1, r_1, x_2, \ldots, a_{T-1}, r_{T-1}, x_T\}$, where x_i denotes the raw pixels of the game screen, and a_i and r_i the corresponding action and reward. In order to give the network a sense of time (which is useful to determine the velocity of the ball), the input to the network is a 'pre-processed' state $\phi_t = \phi(x_t, x_{t-1}, x_{t-2}, x_{t-3})$, where $\phi(\cdot)$ is also a pre-processing function that converts the 210 x 160px RGB frame into a greyscale and extracts an 80 x 80px center crop of the frame (essentially removing extraneous details from the game such as the numbers at the top depicting the scores).

An important detail to note is that OpenAI Gym by default uses the 'frame-skipping' technique. In other words, the x_i's seen by the agent are actually k frames apart, and actions are repeated in the skipped in-between frames. This design decision was made in part for computational speed reasons, so that a frame-skip of k would mean the agent can play roughly k times more episodes compared to not having a frame-skip at all.[2]

The complete algorithm for training the deep-Q network (DQN) is depicted in Algorithm 1 of [1]. Just like their experimental configuration, we use RMSProp with learning rate 0.0002, minibatches of size 32, and linearly anneal the ϵ factor from 1.0 to 0.1 over the course of a million frames. The target network update frequency was set to every 10,000 time steps. The maximum size of the experience replay buffer was set to 350k.[3] The results are shown in Figure 1.

2.1 Asynchronous DQN

An asynchronous version of DQN was also attempted. In this version, each CPU thread (of which there are many) run Q-learning in their own copy of the environment but share the same network parameters. This means that each thread computes and accumulates $\frac{\partial L}{\partial \theta}$ for some number of iterations before performing a gradient descent update on the shared parameters θ. This is done in a 'hogwild' fashion [6], where each thread is free to update the parameters at any time without locks. Unlike the synchronous equivalent, no experience replay is required; this is because in the ideal situation, each thread at any particular point in time will contribute uncorrelated gradients since they are exploring different parts of the state space.

[2] I was not aware of this for a very long time, and ended up implementing my own frameskip in my code, meaning I was 'double frame-skipping'. It may have been the cause of a plethora of frustrations.

[3] At the time, this was due to memory reasons.

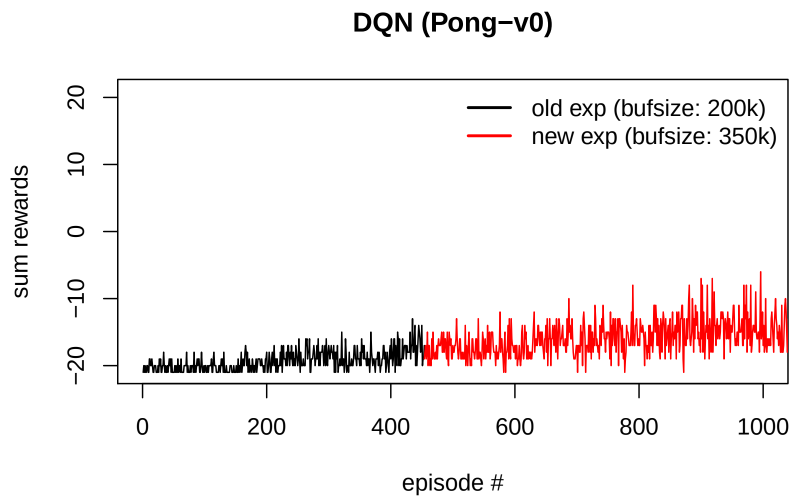

Figure 1: DQN experiment on Atari Pong. The black curve denotes an old experiment, where the experience replay size was of size 200k. The red experiment is the same experiment resumed with a 350k replay buffer. Better learning may result from using a much larger replay buffer from the start (in practice, this is usually 1M in size). The highest reward sum achieved was -3 (so in its best run, the agent almost performs just as well as the AI agent, but not better).

Pseudocode for the algorithm can be found in Algorithm 1 of [3]. The experimental setup is similar to that described in the paper, where each thread anneals ϵ from 1.0 to either $\{0.1, 0.01, 0.5\}$, where the choice is randomly sampled with corresponding probabilities $\{0.4, 0.3, 0.3\}$.

Unfortunately, while this implementation ran slightly faster than its synchronous (GPU) equivalent with 12 CPU threads, it does not appear to converge to a policy beyond seemingly random. It is not completely clear the reason for this, apart from speculating that the gradients contributed to by each thread are not decorrelated enough, resulting in the loss curve plateauing out. It does not seem likely that running the experiment for long will be worthwhile; the results in [3] indicate that beating pong with this implementation should take a mere couple of hours, and in our case, this experiment was run for a couple of days.

3 Conclusion

In conclusion, I was able to implement DQN and achieve modest results. Better results are likely to be achieved by re-running the experiment using a much larger experience replay buffer and more tweaking of optimisation hyperparameters.

It is currently not clear why the asynchronous version does not converge.

The code for this project can be found here [4]. The implementation was written with Theano [7] and Lasagne [8].

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[4]`https://github.com/christopher-beckham/comp767/tree/master/deep_q_learning`

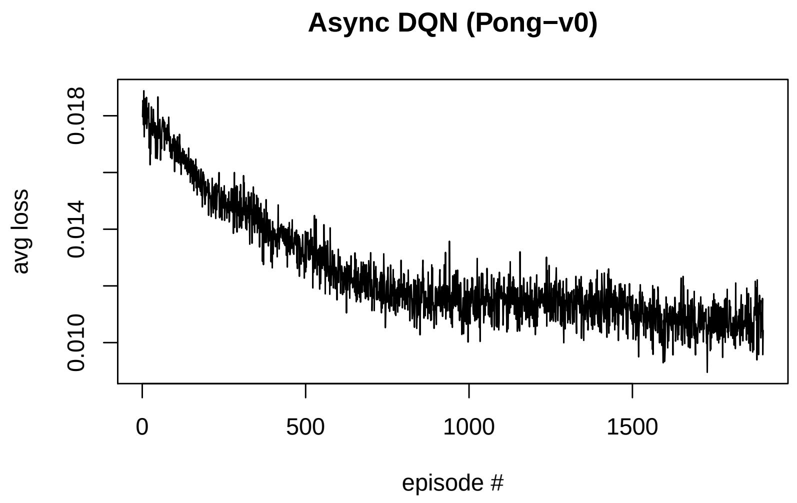

Figure 2: The loss averaged over each of the worker threads.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[3] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[4] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[5] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

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

[7] Theano development team. Theano: A python framework for fast computation of mathematical expressions. *CoRR*, abs/1605.02688, 2016.

[8] Lasagne development team. Lasagne: First release., August 2015.