Neural K-Nearest Neibhorhood Training

Guannan Hu

April 2, 2018

1 Introduction

Until the AlphaGo defeated the 18-time world champion Lee Sedol, the reinforcement learning have a rapid development with deep neural network. such as play atari game without human knowledge [1], and AlphaZero [2] who is the extension version have been trained without human knowledge and self-play. Although the deep reinforcement learning have a great performance, there exist a lot of problems in deep reinforcement learning algorithms.

- The algorithms must able to learn from a scalar reward signal that is frequently sparse, noisy and delayed. The deplay between actions and resulting rewards, which can be thousands of time-steps long, seems particularly daunting when compared to the direct association between inputs and targets found in supervised learning;
- The most deep learning algorithms assume the data samples to be independent, while in reinforcement learning one typically encounters sequences of highly correlated states;
- In RL, the data distribution changes as the algorithm learns new behaviours, which can be problematic for deep learning methods that assume a fixed underlying distribution;
- Stochastic gradient optimisation requires the use of small learning rates. Due to the global approximation nature of neural networks, high learning rates cause catastrophic interference. Low learning rates mean that experience can only be incorporated into a neural network slowly;
- Environments with sparse reward signal can be difficult for a neural network to model as there may be very few instances where the reward is non-zero. This can be viewed as a form of class imbalance where low-reward samples outnumber high-reward samples by a unknown number. Consequently, the neural network disproportionally underperforms at predicting larger rewards, making it difficult for an agent to take the most rewarding actions;
- Reward signal propagation by value-bootstrapping Deep Q-Learning Networking techniques, such as Q-learning, results in reward information being propagated one step at a time through the history of previous interactions with the environment. This can be fairly efficient if updates happen in reverse order in which the transitions occur. However, in order to train on randomly selected transitions, and, in order to further stabilise training, required the use of slow updating *target network* further slowing down reward propagation.

Deep Q-learning Network Mnih et.al present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning.[3] The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. In DQN, the standard Q-learning update for the parameters after taking action A_t in state S_t and observing the immediate reward R_{t+1} and resulting state S_{t+1} is then

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha(Y_t^Q - Q(S_t, A_t; \boldsymbol{\theta}_t))\nabla_{\boldsymbol{\theta}_t} Q(S_t, A_t; \boldsymbol{\theta}_t) \tag{1}$$

where α is a scalar step size and the target Y_t^Q is defined as

$$Y_t^Q \equiv R_{t+1} + \gamma max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t) \tag{2}$$

This update resembles stochastic gradient descent, updating the current value $Q(S_t, A_t; \boldsymbol{\theta}_t)$ towards a target value Y_t^Q.

Double Deep Q-learning Network The idea of Double Q-learning[4] is to reduce overestimations by decomposing the max operation in the target into action selection and action evaluation. Although not fully decoupled, the target network in the DQN architecture provides a natural candidate for the second value function, without having to introduce additional networks. In DDQN, It proposed to evaluate the greedy policy according to the online network, but using the target network to estimate its value. DDQN update is the same as for DQN, but replacing the target Y_t^{DQN} with

$$Y_t^{DoubleDQN} \equiv R_{t+1} + \gamma Q(S_{t+1}, argmax_a Q(S_{t+1}, a; \boldsymbol{\theta}_t), \boldsymbol{\theta}_t^-) \tag{3}$$

Dueling Deep Q-learning Network The Dueling Deep Q-Learning Network [5] explicitly separates the representation of state values and (state-dependent) action advantages. The dueling architecture consists of two streams that represent the value and advantage functions, which sharing a common convolutional feature learning module. The two streams are combined via a special aggregating layer to produce an estimate of the state-action function Q. This dueling network should be understood as a single Q network with two streams that replaces the popular single-stream Q network in existing algorithms such as DQN. The dueling network automatically produces separate estimates of the state value function and advantage function, without any extra supervision.

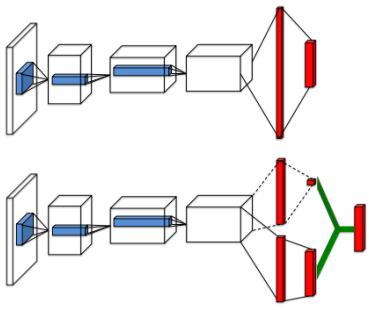

Figure 1: A popular single stream Q-network (**top**) and the dueling Q-network (**bottom**). The dueling network has two streams to separately estimate (scalar) state-value and the advantages for each action; the green output module implements equation (4) to combine them. Both networks output Q-values for each action.

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \left(A(s, a; \theta, \alpha) - \frac{1}{|\mathcal{A}| \sum_{a'} A(s, a'; \theta, \alpha)} \right) \tag{4}$$

Experience Replay and Prioritized Experience Replay Experience Replay [6] has gained popularity in DQN, where it it often motivated as a technique for reducing sample correlation. Experience Replay addresses both of these issue: with experience stored in a replay memory, it becomes possible to break the temporal correlations by mixing more and less recent experience for the updates, and rare experience will be used for more than just a single update. In general, experience replay can reduce the amount of experience required to learn, and replace it with more computation and more memory - which are often cheaper resources that the RL's agent's interactions with its environment. In Experience Replay, the experience transitions were uniformly sampled from a replay memory. the transition are replayed at the same frequency that they were original experienced, regardless of their significance. and in Prioritized Experience Replay [7], so as to replay important transitions more frequently, and therefore learn more efficiently. and the distributed prioritized experience relpay [8] the algorithm decouples acting from learning: the actors interact with their own instance of the environment by selecting actions according to a shared neural network, and accumulate the resulting experience in a shared experience replay memory; the learner replays samples of experience and updates the neural network.

Distributed Important Sampling A complementary family of techniques for speeding up training is based on variance reduction by means of *importance sampling*. This has been shown be useful in the context of neural networks. Sampling non-uniformly from a dataset and weighting updates according to the sampling probability in order to counteract the bias thereby introduced can increase the speed of convergence by reducing the variance of the gradients. One way of doing this is to select samples with probability proportional to the L_2 norm of the corresponding gradients. In supervised learning, this approach has been successfully extended to the distributed setting. An alternative is to rank samples according to their latest known loss value and make the sampling probability a function of the rank rather than of the loss itself.

Trust Region Policy Optimization, TRPO In Trust Region Policy Optimization [9, 10], an objective function(the "surrogate" objective) is maximized subject to a constraint on the size of the policy update. Specifically,

$$\begin{aligned} \max_{\theta} \ & \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{old}(a_t|s_t)} \hat{A}_t \right] \\ \text{subject to } & \hat{\mathbb{E}}_t[\mathrm{KL}[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]] \leq \delta \end{aligned} \tag{5}$$

Here,θ_{old} is the vector of policy parameters before the update. This problem can efficiently approximately solved using the conjugate algorithm, after making a linear approximation to the objective and a quadratic approximation to the constraint.

The theory justifying TRPO actually suggests using a penalty instead of a constraint, i.e., solving the unconstrained optimization problem

$$\max_{\theta} \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t - \beta \mathrm{KL}[\pi_{old}(\cdot|s_t), \pi_\theta(\cdot|s_t)] \right] \tag{6}$$

for some coefficient β. This follows from the fact that a certain surrogate objective (which computes the max KL over states instead of the mean) forms a lower bound (i.e., a pessimistic bound) on the performance of the policy π. TRPO uses a hard constraint rather than a penalty

because it is hard to choose a single value of β that performs well across different problem – or even within a single problem, where the characteristics change over the course of learning. Hence, to achieve our goal of the first-order algorithm that emulates the monotonic improvement of TRPO.

Clipped Surrogate Objective Let $r_t(\theta)$ denote the probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$, so $r(\theta_{old}) = 1$. TRPO maximizes a "surrogate" objective

$$L^{CPI}(\theta) = \hat{\mathbb{E}}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\right] = \hat{\mathbb{E}}_t[r_t(\theta)\hat{A}_t] \tag{7}$$

The superscript CPI refers to conservative policy iteration,where this objective was proposed. Without a constraint, maximization of L^{CPI} would lead an excessively large policy update; hence, we now consider how to modify the objective, to penalize changes to the policy that move $r_t(\theta)$ away from 1.

The main object we propose is the following:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{8}$$

The motivation for this objective is as follows. The first term inside the min is L^{CPI}. the second term, $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t$, modifies the surrogate objective by clipping the probability ratio, which removes the incentive for moving r_t outside of the interval $[1-\epsilon, 1+\epsilon]$. Finally, we take the minimum of the clipped and unclipped objective, so the final objective is lower bound on the unclipped objective.With this scheme, we only ignore the change in probability ratio when it would make the objective improve and include it when it makes the objective worse. Note that $L^{CLIP}(\theta) = L^{CPI}(\theta)$ to first order around θ_{old}, however, they become different as θ moves away from θ_{old}.

Adaptive KL Penalty Coefficient Another approach, which can be used as an alternative to the clipped surrogate objective, or in addition to it, is to use a penalty on KL divergence, and to adapt the penalty coefficient so that we achieve some target value of the KL divergence d_{targ} each policy update. In the experiments, it found that the KL penalty performed worse than the clipped surrogate objective, however, we've included it here because it's an important baseline.

In the simplest instantiation of this algorithm, we perform the following steps in each policy update:

- Using several epochs of minibatch SGD, optimize the KL-penalized objective

$$L^{KLPEN}(\theta) = \hat{\mathbb{E}}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t - \beta\text{KL}[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]\right] \tag{9}$$

- Compute $d = \hat{\mathbb{E}}_t[\text{KL}[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]]$
  - If $d < d_{targ}/1.5, \beta \leftarrow \beta/2$
  - If $d > d_{targ} \times 1.5, \beta \leftarrow \beta \times 2$

The updated β is used for the next policy update. With this scheme, we occasionally see policy updates where the KL divergence is significantly different from d_{targ}, however, there are rare, and β quickly adjusts. The parameters 1.5 and 2 above chosen heuristically.

Proximal Policy Optimization[11]

Neural Episodic Control Neural Episodic Control [12] Deep reinforcement learning methods attain super-human performance in a wide range of environments. Such methods are grossly inefficient, often taking orders of magnitudes more data than human to achieve reasonable performance. Neural Episodic Control : a deep reinforcement learning agent that is able to rapidly assimilate new experiences and act upon them. Our agent uses a semi-tabular representation of the value function: a buffer of past experience containing slowly changing state representations and rapidly updated estimates of the value function.

The agent of NEC consists of three components: a convolutional neural network that processes pixel images s, a set of memory modules (one per action), and a final network that converts read-outs from the actions memories into $Q(s, a)$ values.

Differentiable Neural Dictionary (DND) For each action $a \in \mathcal{A}$, NEC has a simple memory module $M_a = (K_a, V_a)$, where K_a and V_a are dynamically sized arrays of vectors, each containing the same number of vectors. The memory module acts as an arbitrary association from keys to corresponding values, much like the dictionary data type found in programs. Thus we refer to this kind of memory module as a *differentiable neural dictionary* (DND). There are two operations possible on a DND: *lookup* and *write*, Performing a lookup on a DND maps a key h to an output value o:

$$o = \sum_i w_i v_i \tag{10}$$

where v_i is the ith element of the array V_a and

$$w_i = k(h, h_i) / \sum_j k(h, h_j) \tag{11}$$

where h_i is the ith element of the array K_a and $k(x, y)$ is a kernel between vectors x and y, e.g., Gaussian or inverse kernels, such as:

$$k(h, h_i) = \frac{1}{||h - h_i||_2^2 + \delta} \tag{12}$$

Thus the output of a lookup in a DND is a weighted sum of the values in the memory, whose weights are given by normalised kernels between the lookup key and the corresponding key in memory. To make queries into very large memories scalable we shall make two approximations in practice: firstly, we shall limit (10) to the top p-nearest neighbours. Secondly, we use an approximate nearest neighbours algorithm to perform the lookups, based up kd-trees.

keys and values are written to the memory by appending them onto the end of the arrays K_a and V_a respectively. if a key already exists in the memory, then its corresponding value is updated, rather than being duplicated.

Agent Architecture Figure (2) shows a DND as part of the NEC agent for asingle action, whilst Algorithm (1) describes the general outline of the NEC algorithm.

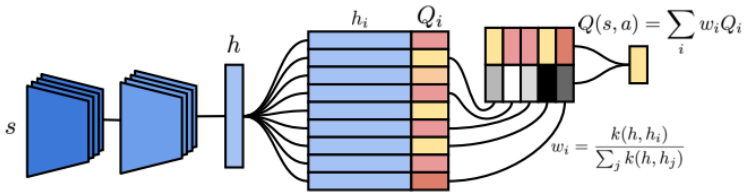

Figure 2: Architecture of episodic memory module for a single action a. Pixels representing the current state enter through a convolutional neural network on the bottom left and an estimate of $Q(s,a)$ exists top right. Gradients flow through the entire architecture.

The pixel state s is processed by a convolutional neural network to produce a key h. The key h is then used to lookup a value from the DND, yielding weights w_i in the process for each element of the memory arrays. Finally, the output is a weighted sum of the values in the DND. The values in the DND, in the case of an NEC agent, are the Q values corresponding to the state that originally resulted in the corresponding key-value pair to be written to hte memory. Thus this architecture produces an estimate of $Q(s,a)$ for a single given action a/ The architecture is replicated once for each action a the agent can take, with the convolutional part of the network shared among each separate DND M_a. The NEX agent acts by taking the action with the highest Q-value estimate at each time step. In practice, we use ϵ-greedy policy during training with a low ϵ.

Algorithm 1 Neural Episodic Control

$\mathcal{D}$: replay memory
M_a: a DND for each action a.
N: horizon for N-step Q estimate.
for each episode **do**
  for $t = 1, 2, ..., T$ **do**
    Receive observation s_t from environment with embedding h.
    Estimate $Q(s_t, a)$ for each action a via (10) from M_a
    $a_t \leftarrow \epsilon-$greedy policy based on $Q(s_t, a)$
    Take action a_t, receive reward r_{t+1}.
    Append $(h, Q^{(N)}(s_t, a_t))$ to M_{a_t}.
    Append $(s_t, a_t, Q^{(N)}(s_t, a_t))$ to $\mathcal{D}$.
  end for
end for

The N-step Q-value estimate is

$$Q^{(N)}(s_t, a) = \sum_{j=0}^{N-1} \gamma^j r_{t+j} + \gamma^N \max_{a'} Q(s_{t+N}, a'). \tag{13}$$

The bootstrap term of (13), $\max_{a'} Q(s_{t+N}, a')$ is found by querying all memories M_a for each action a and taking the highest estimated Q-value returned. Note that the earliest such values can be added to memory is N steps after a particular (s, a) pair occurs.

When a state-action value is already present in a DND (i.e the exact same key h is already in K_a), the corresponding value present in V_a, Q_i, is updated in the same way as the classic

tabular Q-learning algorithm:

$$Q_i \leftarrow Q_i + \alpha(Q^{(N)}(s,a) - Q_i) \tag{14}$$

where α is the learning rate of the Q update.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. Computer Science, 2013.

[2] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. Nature, 550(7676):354, 2017.

[3] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. nature, 529(7587):484–489, 2016.

[4] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In AAAI, volume 16, pages 2094–2100, 2016.

[5] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. arXiv preprint arXiv:1511.06581, 2015.

[6] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay. arXiv preprint arXiv:1611.01224, 2016.

[7] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. arXiv preprint arXiv:1511.05952, 2015.

[8] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay. arXiv preprint arXiv:1803.00933, 2018.

[9] John Schulman, Sergey Levine, Philipp Moritz, Michael I Jordan, and Pieter Abbeel. Trust region policy optimization. Computer Science, pages 1889–1897, 2015.

[10] Yuhuai Wu, Elman Mansimov, Shun Liao, Roger Grosse, and Jimmy Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. 2017.

[11] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. 2017.

[12] Alexander Pritzel, Benigno Uria, Sriram Srinivasan, Adria Puigdomenech, Oriol Vinyals, Demis Hassabis, Daan Wierstra, and Charles Blundell. Neural episodic control. arXiv preprint arXiv:1703.01988, 2017.