Deep Reinforcement Learning Notes

Nicholas Denis

February 21, 2019

Contents

1 Policy Gradient Algorithms

1.1 Reinforce

- On-policy
- discrete or continuous actions

This typically estimates the gradient using:

$$\nabla_\theta J(\theta) = \mathbb{E}[R_t \nabla_\theta \log \pi_\theta(a_t|s_t)]$$

where R_t is the sum of (discounted) rewards from time t until the end of an episode. Hence this is typically done using pure Monte Carlo estimates of the Q-values.

1.2 Deterministic Policy Gradient

- Off-policy
- continuous actions
- actor-critic

Silver wants to answer the question if a policy gradient theorem exists when action selection is deterministic (e.g. policies are not stochastic). This theorem requires that the policy integrates only over the state space, not the action space. Since policy gradient methods need exploration, typically stochastic policies satisfy this. However, with deterministic policies, the agent must do off-policy learning. The idea is to choose actions accoring to a stochastic behaviour policy, but to learn about a deterministic target policy. This is used for an off-policy AC approach. Defining the objective function,

$$\begin{aligned} J(\pi_\theta) &= \int_{\mathcal{S}} \rho^\pi r(s, \pi_\theta(s)) ds \\ &= \mathbb{E}_{s\sim\rho^\pi}[r(s, \pi_\theta(s))] \end{aligned}$$

Where ρ^π is the steady state distribution when following π. The the deterministic gradient theorem provides the following:

$$\begin{aligned} \nabla J(\pi_\theta) &= \int_{\mathcal{S}} \rho^\pi \nabla_\theta \pi_\theta \nabla_a Q^\pi(s,a)|_{a=\pi_\theta(s)} ds \\ &= \mathbb{E}_{s\sim\rho^\pi}[\nabla_\theta \pi_\theta \nabla_a Q^\pi(s,a)|_{a=\pi_\theta(s)}] \end{aligned}$$

The authors also show that this is the limiting case for a stochastic policy where the variance goes to zero. The updates to the actor-critic are as follows:

$$\begin{aligned} \delta_t &= r_t + \gamma Q^w(s_{t+1}, a_{t+1}) - Q^w(s_t, a_t) \\ w_{t+1} &= w_t + \alpha_w \delta_t \nabla_w Q^w(s_t, a_t) \\ \theta_{t+1} &= \theta_t + \alpha_\theta \nabla_\theta \pi_\theta(s_t) \nabla_a Q^w(s_t, a_t)|_{a=\pi_\theta(s)} \end{aligned}$$

1.3 DDPG: Deep Deterministic Policy Gradients

- Off-policy
- continuous actions
- actor-critic

Off-policy actor critic. They also do something unique for the target actor and target critic networks. They update the parameters slowly, but continuosuly, using $\theta^- = \tau\theta + (1-\tau)\theta^-$, with $\tau << 1$. The exploration policy μ adds noise from an Ornstein-Uhlenbeck process to the actor policy. This noise process generates temporally correlated exploration. Note that the critic (Q-net) can make use of double Q learning as well. The general algorithm is:

Select action from actor policy plus noise
execute action, receive next state and reward
Store transition into replay buffer
Sample random batch of transitions
Compute target$y_i$according to DQN, double Q, etc
Update critic using MSE
Update actor using average of batch size DPG gradients
Update the target networks as described above

During test time there was no noise added to the policies.

1.4 Reactor

- Off-policy (uses Retrace)
- discrete actions
- actor-critic

This is an interesting algorithm that uses Retrace. However, there are much too many hand engineering, heuristics and hyper-parameters in the implementation. Reactor (Retrace-Actor) uses an off-policy multi-step learning algorithm based on an actor-critic architecture. The critic implements Retrace, whle the actor is trained by a new policy gradient alogirthm called β-leave-one-out, which makes use of both the off-policy Retrace-corrected return and the estimated Q-function. Reactor is sample efficient due to the use of memory replay, and numerically efficient as it uses multi-step returns, improving the speed of propagation of rewards. AC is used to evaluate behavioural policies (different than policies learned from Q-values). Since Reactor uses multi-step returns, sequences of transitions are sampled from the memory replay, rather than just single transitions. For this reason, they use a recurrent neural net

(RNN) architecture. The others note the following methods to estimate the gradient, G:

$$G = \sum_a Q^\pi(a) \nabla \pi(a)$$

Denoting A as the action selected, the importance sampling likelihood ratio estimate is:

$$\hat{G}_{ISLR} = \frac{\pi(A)}{\mu(A)}(R(A) - V)\nabla\pi(A)$$

This is high-variance as the IS terms can be unbounded. The *leave one out* policy-gradient estimate is:

$$\hat{G}_{LOO} = R(A)\nabla\pi(A) + \sum_{a \neq A} Q(a)\nabla\pi(a)$$

This is low variance but may be biased if the estimated Q values are different than Q^π. For this reason $\beta - LOO$ is introduced:

$$\hat{G}_{\beta-LOO} = \beta(R(A) - Q(A))\nabla\pi(A) + \sum_a Q(a)\nabla\pi(a)$$

When $\beta = 1$this reduces to LOO, and when $\beta = \frac{1}{\mu(A)}$ the estmate is unbiased. One modification the authors take is rather than storing the prior behaviour policy probabilities $\mu(a_s|x_s)$, they use a NN to predict these values (which uses an LSTM). The policy head of the network uses a softmax over the actions mixed with the uniform distribution, where the mixing parameter is a hyper-parameter. The results were good/decent, but not overly impressive.

1.5 Proximal Policy Optimization

- On-policy
- continuous actions
- actor-critic (uses advantage functio)

Tries to do what TRPO does, but with more simplicity, and only 1st order approximation techniques. TRPO uses an objective function:

$$\max \hat{\mathbb{E}}\Big[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\Big]$$
$$\text{subject to} \hat{\mathbb{E}}[KL[\pi_{\theta_{old}}(.|s_t), \pi_\theta(.|s_t)]] \leq \delta$$

Which is solved using conjugate gradient method after making a linear approximation of the objective and a quadratic approximation of the constraint. This motivates using the following objective (without constraint):

$$\max \hat{\mathbb{E}}\Big[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t - \beta KL[\pi_{\theta_{old}}(.|s_t), \pi_\theta(.|s_t)]\Big]$$

However, no single value of β may work, so this is why the hard constraint defined for TRPO is used. The authors introduce two modifications, one using the clippped surrogate objective, and an adaptive KL penalty (β). I won't explain the adaptive KL penalty, but essentially β scales up or down by a fixed factor (hyper-parameter) given the how large the KL estimate currently is. For the clipped surrogate objective:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}\Big[\min(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t, \text{clip}(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon)\hat{A}_t)\Big]$$

The clipping objective is pessimistic, in the sense that when the policy would be changed in the direction of improvement, we restrict how far it will move, and when the policy would become worse, we prefer moving in that direction. The objective also adds an entropy regularization term for the policy. The algorithm uses N parallel actors, each acting for T time steps, and the T-step TD error"

$$\hat{A}_t = -V(s_t) + r_t + \gamma r_{t+1} + ... + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T)$$

is used. The loss function is applied to these NT timesteps of data, and optimized using gradient ascent for K epochs, with minibatch size of $M \leq NT$.

1.6 Trust Region Policy Optimization

1.7 Soft Actor-Critic

- Off-policy
- continuous actions (and discrete?)
- actor-critic
- uses maximum entropy regularizer

SAC uses an off-policy maximum entropy deep RL with a stochastic actor. The objective function is to maximize reward and the entropy of the policy. This incentivizes exploration, reduces converging on a deterministic policy. Assuming the policy class is related to a class of distributions, their Bellman operator converges to the optimal policy in that policy class. Their approach requires a policy evaluation and policy improvement. The new policy at each improvement step minimizes the KL divergence between the exponential of of the Q value of the previous policy. Their results are positive.

1.8 TD3: Twin Delayed Deep Deterministic Policy Gradients

- Off-policy (built off of DDPG)
- continuous actions
- actor-critic using Double Q networks

This paper seeks to address the over-estimation bias in DQN (critic networks) as they are used in actor-critic architectures. They draw on the intuition and results of using Double DQN to reduce this over-estimation bias, and translate this to the AC setting. Specifically, they look at the DDPG algorithm. Additionally, noting that the quality of the critic will induce quality issues in the actor, they suggest delaying the actor after several iterations of critic updates, to reduce variance and error propagation. Thus, Twin Delayed DDPG is introduced. Code is available on github.

First, they use two separate pairs of actors and critics $(\pi_{\phi_1}, \pi_{\phi_2})$, and $(Q_{\theta_1}, Q_{\theta_2})$, and targets are used as follows:

$$y_1 = r + \gamma Q_{\theta_2'}(s', \pi_{\phi_1}(s'))$$
$$y_2 = r + \gamma Q_{\theta_1'}(s', \pi_{\phi_2}(s'))$$

While using this approach is more effective at reducing over-estimation bias in DDPG, it is not sufficient. Rather, they introduce a "clipped" Double Q-learning algorithm:

$$y_1 = r + \gamma \min_{i=1,2} Q_{\theta_i'}(s', \pi_{\phi_1}(s'))$$

TD3 uses two critic networks, a single actor network, each of the three networks with an associated target network. The algorithm samples noise for exploration, takes actions from the critic and adds the transitions into memory. After each step, a sample of N transitions are made, their *clipped* targets are performed, and the critics are both updated. Ever d time steps the actor is updated using DDPG, and the target networks are updated using $\theta' = \tau\theta + (1-\tau)\theta'$, for $\tau = 0.005$ (where θ is meant to represent the parameters for each of the three networks used).

1.9 DDDPG: Distributed Distributional Deterministic Policy Gradients

- Off-policy (built off of DDPG)
- discrete actions (and continuous?)
- actor-critic
- distributional

This paper uses the DDPG algorithm as a baseline and augments it to use the distributional perspective on RL, as well as the distributed off-policy approach. Since DDPG uses the Q values as the baseline to update the actor, this can be done with any off-policy approach. The authors implement the ApeX framework which improves wall-clock time (original ApeX paper uses 360 actors

each on their own core). The authors use the DDPG gradient theorem for the distributional setting:

$$\nabla_\theta J(\theta) \approx \mathbb{E}_\rho[\nabla_\theta \pi_\theta(x) \mathbb{E}[\nabla_a Z_w(x,a)]|_{a=\pi_\theta(x)}].$$

Recalling the distributional perspective on RL paper, the distributional function Z is a function of (x,a) and maps this state-action pair to a distribution. The authors favor a categorical distirbution termed $C51$, wherein they divide the interval (Vmax, Vmin) into 51 atoms. Since the Bellman operator on this Z_w may not be a closed operation, a projection onto one of the 51 atoms must always occur after each Bellman update. The authors note through ablation studies that first and foremost, the distributional approach improves over D3PG, as well, using $n = 5$ step size over $n = 1$ also further improves the results, while use of prioritized experience replay vs regular replay buffer was mostly similar. Cross-entropy between the $\mathcal{T}Z$ and Z, $\mathcal{T}$ being the Bellman operator, was used to compute the Bellman residual error (which takes into account the projection).

1.10 ACER: Actor-Critic with Experience Replay

- Off-policy (truncated importance sampling)
- discrete and continuous actions
- actor-critic

To use AC with ER the authors introduce truncated importance sampling with bias correction, stochastic dueling network architectures and a new trust region policy optimization method. The authors are motivated by the fact that DQN learns deterministic policies, which may not be optimal in adversarial domains, and that finding the greedy action for large action spaces is costly. At this time, continuous and discrete online policy algorithms (e.g. A3C) are sample inefficient. This algorithm uses the off-policy Retrace algorithm.

The authors note that estimating the gradient,
$g = \mathbb{E}\Big[\sum_{t\geq 0} A^\pi(x_t,a_t)\nabla_\theta \log \pi_\theta(a_t|x_t)\Big]$
can be done by replacing the advantage function with either $Q^\pi(x,a), R_t$ (the empirical discounted sum of rewards) or the temporal difference residual $r_t + \gamma V^\pi(x_{t+1}) - V^\pi(x_t)$, each without introducing bias, though each with varying amounts of variance. Of course, exact values for Q, V, A are not known, and so estimates are used, which introduce bias. The authors note that A3C uses the $n - step$ temporal difference error gradient approximation:

$$\hat{g}_{a3c} = \sum_{t\geq 0} \Big(\Big(\sum_{i=0}^{k-1} \gamma^i r_{t+i} \Big) + \gamma^k V^\pi_{\theta_v}(x_{t+k}) - V^\pi_{\theta_v}(x_t) \Big) \nabla_\theta \log \pi_\theta(a_t|x_t)$$

The authors introduce both the discrete and continuous action version of ACER. Noting that it is an off-policy version of A3C, the parellelization of agents is used with ACER. They use a single network with policy and value function head to represent the AC. For discrete actions, network outputs the policy and the Q values. The retrace critic, Q^{ret}. It is used for both the critic, and for estimating the actors gradients. The multi-step nature of the retrace critic reduces bias in the policy gradient and enables faster learning of the critic.

Importance Weight Truncaion with Bias Correction The authors introduce an estimate of the policy gradient that uses clipping, that is unbiased, and bias correction occurs for large vaues of c (they use $c = 10$), meaning when the variance of the off-policy estimator is high.

Efficient TRPO Rather than constraining the updated policy to be close to the prior policy in KL divergence, they propose an *average policy network* that represents the running average of past policies, and constrains updates to not differ in KL too much from this average policy network. The average policy network gets updated "softly" by updating parameters $\theta_a = \alpha\theta_a + (1-\alpha)\theta$, where parameters θ are for the policy (actor) network. The constrained optimization is solved using a linearized KL divergence constraint, and results in a quadratic programming problem, solved using the KKT conditions.

The authors use 16 threads (agents), each with their own replay buffer, updating every 20 steps, using entropy regularization, and found that replay ratio of 4 (4 off-policy updates for every step in the environment) was best.

For Continuous ACER, the authors use a Stochastic Dueling Network approach. Due to the continuous nature of things, Q is estimated (n=5):

$$\tilde{Q}_{\theta_v}(x_t, a_t) \sim V_{\theta_v}(x_t + A_{\theta_v}(x_t, a_t) - \frac{1}{n}\sum_{i=1}^{n} A_{\theta_v}(x_t, u_i), \quad u_i \sim \pi_\theta(.|x_t)$$

1.11 GAE: Generalized Advantage Estimator

- Off-policy (uses TRPO)
- discrete actions
- actor-critic

Uses TRPO. Stochastic policy optimization using PG is hard. They propose methods to reduce variance while maintaining a tolerable level of bias. These methods are parameterized by $\gamma \in [0, 1]$, and $\lambda \in [0, 1]$. The algorithm assumes undiscounted setting. The authors note that for policy gradient methods, estimating the gradient using the advantage function, A, yields almost the lowest possible variance, but must be estimated in practice and thus introduces bias.

Advantage function

$$\begin{aligned} A^\pi(s,a) &= Q^\pi(s,a) - V^\pi(s) \\ \mathbb{E}[A^\pi(s,a)] &= \mathbb{E}[Q^\pi(s,a) - V^\pi(s)] \\ &= \sum_a \pi(a|s) Q^\pi(s,a) - V^\pi(s) \\ &= V^\pi(s) - V^\pi(s) \\ &= 0 \end{aligned}$$

Hence, the advantage of (s,a) is a measure of how, on average, an action is better or worse than the policy behaviour. Hence the gradient term $\nabla_\theta \log \pi_\theta(a|s) A(s,a)$ points in the direction of increased $\pi_\theta(a|s) \iff A(s,a) > 0$.

The authors introduce γ as a parameter to reduce variance, at the cost of introducing bias, though it is exactly the same as a γ-discounting rewards. In the same way that λ-returns generalize the exponentially weighted sum of n-step returns $\forall n \geq 1$, their Generalized Advantage Estimator has the same form, where $\lambda = 0$ is simply the one step TD error, and $\lambda = 1$ is pure bootstrapping, with $\lambda \in (0,1)$ some intermediate mixture of the two extremes.

The authors use trust region methods to learn and update both the value network (critic) and the policy network (actor). For the critic, the new parameters are constrained to not change too much from the old parameters, and for the actor, for g the gradient of the objective:

$$\begin{aligned} &\text{minimize } g^T(\phi - \phi_{old}) \\ &\text{subject to } \frac{1}{N} \sum_{n=1}^{N} (\phi - \phi_{old})^T H (\phi - \phi_{old}) \leq \epsilon \end{aligned}$$

Where H is the Gauss-Newton approximation of the Hessian of the objective, and it is up to a scalar factor equivalent to the Fisher information matrix.

1.12 ACKTR: Actor-Critic using Kronecker-Factor Trust Region

- On-policy (uses trust region in some sense)
- discrete and continuous actions
- actor-critic

Is the first scalable trust region natural gradient method for actor-critic methods. Achieved 2-3-fold improvement in sample efficiency on average, compared to other state-of-the-art AC methods. Claim that SGD and other gradient methods explore parameter space inefficienctly. Natural policy gradient uses

natural gradient descent, which follow the steepest descent direction that uses the Fisher metric as the underlying metric, which is independent of choice of coordiantes but rather moves on the manifold. The natural gradient requires inverting the Fisher information matrix, which is intractable for large neural networks. TRPO avoids this by using Fisher-vector products, which requires many steps of conjugate gradient descent to perform a parameter update, and accurate curvature estimation requires lots of data in each batch, hence is inefficient and impractical.

Kroneker-factored approximated curvature (K-FAC) is a scalable approximation to natural gradient that has been used to speed up training state-of-the-art modern neural nets in SL using large batch sizes. Unlike TRPO, each update is comparable in cost to an SGD update, and it keeps a running average of the curvature information, allowing it to use small batches. In ACKTR, K-FAC allows the covarance matrix of the gradient to be inverted efficiently. Per update, the computation costs of ACKTR are roughly 10 -25 percent higher than SGD-based methods.

ACKTR follows A3C using the advantage function of the form:

$$A^{\pi}(s_t, a_t) = \sum_{i=0}^{k-1} \left(\gamma^i r(s_{t+i}, a_{t+i}) + \gamma^k V_{\phi}^{\pi}(s_{t+k})\right) - V_{\phi}^{\pi}(s_t)$$

where $V_{\phi}^{\pi}(s_t)$ is the value network. To train the value network, the bootstrapped k-step returns R_t are used as targets to train MSE loss for $V_{\phi}^{\pi}(s_t)$.

Natural gradient using K-FAC To minimize some non-convex function with steepest descent, subject to $||\Delta\theta||_B < 1$, where $||x||_B = (x^T B x)^{\frac{1}{2}}$, B a positive semidefinite matrix. When the norm is Euclidean, $B = I$. The solution has the form $\Delta\theta \propto -B^{-1}\nabla_{\theta} J$, for the objective function J, and $\nabla_{\theta} J$ is the standard gradient. The Euclidean norm depends on the parameterization of θ, which is arbitrary. The natural gradient uses B as the Fisher information matrix F, which is a local quadratic approximation to the KL divergence, and is independent of the model parameterization θ of the class of probability distributions. Let $p(y|x)$ denote the output distribution of a NN. Let $L = \log p(y|x)$. Let $W \in \mathbb{R}^{C_{out} x C_{in}}$ denote the weight matrix at the ℓ^{th} layer, and $a \in \mathbb{R}^{C_{in}}$ the input activation with $s = Wa$ the pre-activation for the next layer. The weight gradient is given as $\nabla_W L = (\nabla_s L)a^T$. Then K-FAC uses this to approximate the block F_{ℓ}:

$$\begin{aligned} F_{\ell} &= \mathbb{E}[vec\{\nabla_W L\} vec\{\nabla_W L\}^T] = \mathbb{E}[aa^T \otimes \nabla_s L (\nabla_s L)^T] \\ &\approx \mathbb{E}[aa^t] \otimes \mathbb{E}[\nabla_s L (\nabla_s L)^T] := A \otimes S := \hat{F}_{\ell} \end{aligned}$$

With $A = \mathbb{E}[aa^T]$, $S = \mathbb{E}[\nabla_s L (\nabla_s L)^T]$. This approximation is equivalent to the assumption that the second-order statistics of the activations and the backpropagated derivatives are uncorrelated. With the identities: $(P \otimes Q)^{-1} = P^{-1} \otimes Q^{-1}$

and $(P \otimes Q)vec(T) = PTQ^T$, then,

$$vec(\Delta W) = \hat{F}_\ell^{-1} vec\{\nabla_W J\} = vec(A^{-1}\nabla_W J(S^{-1})^T).$$

Hence, K-FAC only requires computations on matrices comparable in size to W. Distributed approaches also occur which have 2-3x speedup.

Natural gradient in AC Noting that $F = \mathbb{E}_{p(\tau)}[\nabla_\theta \log\pi(a_t|s_t)(\nabla_\theta \log\pi(a_t|s_t))^T]$, where expectation is taken over distribution of trajectories, thisis approximated during rollouts from training. Learning the critic can involve least-squares, using the Gauss-Newton algorithm that approximates the curvature of the Gauss-Newton matrix $G = \mathbb{E}[J^T J]$, where J is the Jacobian of the mapping from parameters to outputs, which is equivalent to Fisher matrix for a Gaussian observation model. Assume the output of the critic v is defined to be a Gaussian distribution $p(v|s_t) \sim \mathcal{N}(v; V(s_t), \sigma^2)$. The Fisher matrix of the critic is defined with respect to this Gaussian output distribution. With $\sigma = 1$ we have vanilla Gauss-Newton.

To use shared layers for AC, the joint distribution $p(a, v|s) = \pi(a|s)p(v|s)$ to be independent. The Fisher metric is constructed using K-FAC as normally, except the networks ouputs need to be sampled independantly, and then apply K-FAC to approximate $\mathbb{E}_{p(\tau)}[\nabla \log\ p(a, v|s)\nabla \log\ p(a, v|s_t)^T]$. They also use "Tikhonov damping approach" and perform asynchronous computation of second-order statistics and inverses required by K-FAC.

For trust region optimization, since $\theta = \theta - \eta F^{-1}\nabla_\theta L$, $\eta := \min\Big(\eta_{max}, \sqrt{\frac{2\delta}{\Delta\theta^T \hat{F} \Delta\theta}}\Big)$.

Results ACKTR is on-policy, and learned 10x faster than A2C on atari games. It outperforms both A2C and TRPO on continuous control domains. It can also use larger batch sizes than A2C.

2 DQN and Variants

2.1 Prioritized Experience Replay

Authors look at the TD-error, δ and use sampling methods based on this error to determine how to sample from experience replay buffer. They consider: $\frac{p_s^\alpha}{\sum_{i=1}^D p_{s_i}^\alpha}$, where α is a temperature parameters, with $\alpha = 0$ being uniform distribution. They also considered ordering the transitions in the replay buffer by largest to smallest TD-error, then sampling $p_s = \frac{1}{rank(s)}$.

2.2 Double DQN

This is an improvement on the DQN paper. Where the DQN uses the following training target:

$$Y_t^{DQN} = r_t + \gamma * maxQ(s_{t+1}, a; \theta^-)$$

where $theta^-$ are the parameters of the target DQN, the update for the DoubleQ learning algorithm is,

$$Y_t^{DoubleQ} = r_t + \gamma Q(s_{t+1}, argmax\ Q(s_{t+1}, a; \theta); \theta^-).$$

Hence, the action considered for $Q(s_{t+1})$ is the one maximized by the DQN network currently updated and driving the policy. However, the Q-values target are still those from the target network. Parameter updates:

$$\theta_{t+1} = \theta_t + \alpha(Y_t^Q - Q(s_t, a_t; \theta_t))\nabla_{\theta_t} Q(s_t, a_t; \theta_t)$$

The intuition behind the Double-Q network is to address the overestimation problems of DQN. The results were positive, and in all papers that followed, double Q learning always out performed DQN.

2.3 Dueling DQN

This paper separates the Value estimate and the Advantage estimates. Given CNN layers, the linear layers are separated into two streams, one to estimate the value of the state, the other stream to evaluate the advantage. These are later combined to construct the Q-value estimates. The other use Double-Q learning updates with this approach. The Q value estimate is given as:

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

The authors produce results better than Double DQN (DDQN) alone.

2.4 A3C

- On-policy
- discrete (and continuous?) actions
- actor-critic

A3C sidesteps the need for an exerience replay buffer, which requires more memory and computation per environment interaction, and requires off-policy learning algorithms. Rather, they use multiple agents in parallel, each collecting their own data independently, which decorrelates at given time-step experienced within the dataset. This came out after Gorilla, which has each actor with its own copy of the environment, and with its own replay buffer. The gradients are asynchronously sent to a central parameter server which updates the central copy of the model. Gorilla used 100 separate actor-learners and 30 parameter servers, and learned 20x faster than DQN.

Uses multi-threading, asynchronous version of Sarsa, DQN, n-step DQN and advantage actor critic, which uses:

$$\nabla_\theta log\pi_\theta(a_t|s_t;\theta)(R_t - b(s_t)) =$$
$$\nabla_\theta log\pi_\theta(a_t|s_t;\theta)A(s_t, a_t;\theta,\theta_{value})$$

Each actor explores differently in the environment, and maximizes diversity. With no experience replay, they focus on on-policy algorithms (Sarsa and actor-critic).

For A-DQN, they allow multiple time steps to occur, accumulating the gradients, before performing an update. For A3C, they also added an entropy regularization term, to help with exploration:

$$\nabla_\theta log\pi_\theta(a_t|s_t;\theta)A(s_t, a_t;\theta,\theta_{value}) + \beta\nabla_\theta H(\pi(s_t;\theta))$$

β is a hyper-parameter. They use RMSprop. The algorithm, generally stated for A3C is below:

Take action sampled from policy$\pi(a_t|s_t;\theta)$
Get reward and next state
Until terminal or reached max amount of time
$R = 0$if terminal, or$V(s_t;\theta_{value})$ for$i \in \{t-1, ..., t_{start}\}$
$R = r_i + \gamma R$
accumulate gradients wrt$\theta : d\theta = d\theta + \nabla_\theta log\pi(a_i|s_i;\theta)(R - V(s_i;\theta_{value}))$
accumulate gradients wrt$\theta_{value} : d\theta_{value} = d\theta_{value} + \dfrac{\partial(R - V(s_i;\theta_{value}))}{\partial\theta_{value}}$
Perform asynchronous update of parameters using accumulated gradients

2.5 Retrace(λ)

- Off-policy; NOT actor-critic, it is just a critic (DQN augmented)
- discrete actions
- multi-step returns with truncation/importance sampling

Claim to have low variance, *safely* uses samples collected off-policy, and is efficient as it makes the best use of samples collected from "near on-policy" behaviour policies. It is the first off-policy learning algorithm that converges to Q^* without the GLIE assumption. Can properly and safely make use of all data from past policies in the experience replay buffer.

Return Based Operators: The λ-return extension of the Bellman operator This considers the exponentially weighted sums of n-step returns:

$$\mathcal{T}_\lambda^\pi Q := (1-\lambda)\sum_{n\geq 0}\lambda^n[(\mathcal{T}^\pi)^{n+1}Q] = Q + (I-\lambda\gamma P^\pi)^{-1}(\mathcal{T}^\pi Q - Q)$$

where $\mathcal{T}^\pi Q - Q$ is the Bellman residual of Q for policy π. It can be shown that Q^π is also the fixed point of $\mathcal{T}_\lambda^\pi$. At one extreme ($\lambda = 0$), this is simply the bellman operator, and at the other ($\lambda = 0$), we simply have the policy evaluation operator. Intermediate values of λ trade off bias and variance.

A general opterator to compare several return-based off-policy algorithms:

$$\mathcal{RQ}(x,a) := Q(x,a) + \mathbb{E}_\mu\big[\sum_{t\geq 0}\gamma^t\big(\Pi_{s=1}^t c_s\big)\big(r_t + \gamma\mathbb{E}_\pi(Q(x_{t+1},.) - Q(x_t,a_t))\big)\big]$$

where $\mathbb{E}_\pi(Q(x_{t+1},.) - \sum_a \pi(a|x)Q(x|a)$, and $\Pi_{s=1}^t c_s = 1$ when $t = 0$. Importance sampling uses $c_s = \frac{\pi(a_s|x_s)}{\mu(a_s|x_s)}$, with μ the behaviour policy. $Q(\lambda$ uses $c_s = \lambda$. This algorith, Retrace(λ) uses $c_s = \lambda\ \min\big(\frac{\pi(a_s|x_s)}{\mu(a_s|x_s)}\big)$.

Intuition Importance sampling suffers from possibly high (even infinite) variance. Retrace doesn't. $Q(\lambda)$ assumes the two policies are similar, but retrace doesn't require this. Retrace doesn't cut traces as much as the Tree-backup algorithm. Retrace is also a γ-contraction around Q^π for arbitrary μ, π, and converges to Q^*.

Implementation The agent adapts the DQN to replay short sequences from the memory (k-steps):

$$\Delta Q(x_t,a_t) = \sum_{s=t}^{t+k-1}\gamma^{s-t}\big(\Pi_{t+1}^s c_i\big)[r(x_s,a_s) + \gamma\mathbb{E}_\pi Q(x_{s+1},.) - Q(x_s,a_s)]$$

2.6 Rainbow

- Off-policy; NOT actor-critic, it is just a critic (DQN augmented)
- discrete actions

Rainbow seeks to combine several orthogonal improvements over the original DQN paper. Specifically, the authors cook up ways to combine: Double Q learning, prioritized experience replays, dueling networks, mult-step learning and distributional RL. The results improve over each independently and ablations.