APML-Exercise 4 - Part II

Daniel Segal 205951270
Talia Nir 318338001
Guy Lutsker 207029448

December 27, 2020

Preface

We have chosen to implement the following algorithm: Q-Actor-Critic. This is a reinforcement-learning algorithm which uses an actor-critic architecture. The actor is responsible for choosing the action, the critic is responsible for evaluating the quality of the action via optimization of the expected advantage: $\mathbb{E}\left[r_t + \gamma \cdot Q\left(s_{t+1}, a_{t+1}\right) - Q\left(s_t, a_t\right)\right]$. The pseudo-code for the algorithm:

Algorithm 1 Q-Actor-Critic

Init $\gamma \in (0,1)$, opt_time$\in [1, steps-1]$
Init parameters of the NN of π_θ - the policy/actor network
Init parameters of the NN of Q_α - the Critic/Q network
for t in [0, ..., steps-1]:

sample: $a_t \sim \pi_\theta\left(s_t\right)$
Make the step with the action a_t, get the next state and reward
store the tuple of action, state, next state and reward in memory
if t % opt_time == opt_time - 1:

For a batch $(\vec{s}, \vec{r}, \vec{a}, \vec{s}_{next})$ of recorded transition in memory:

Train the actor:
Let $T = \#$ of steps in the batch
make a gradient step for the following
optimization problem:
$argmin_\theta \ \frac{1}{T}\sum_{t=1}^{T} \log\left(\pi_\theta\left(a_t|s_t\right)\right) \cdot Q_\alpha\left(s_t, a_t\right)$
Train the critic:
Let $A_t = r_t + \gamma \cdot$ no_grad$[Q_\alpha\left(s_{t+1}, a_{t+1}\right)] - Q_\alpha\left(s_t, a_t\right)$
make a gradient step for the following optimization
problem(w.r.t to all samples in the batch):
$argmin_\alpha \ \frac{1}{T}\sum_{i=1}^{T} A_i^2$

We have chosen to implement QAC with two neural networks:

- Policy network: $\pi : S \to [0,1]^3$
- Q network: $Q : S \times A \to \mathbb{R}^3$

The networks have a very similar architecuture:

- 2 convolutional layers
- Linear layer

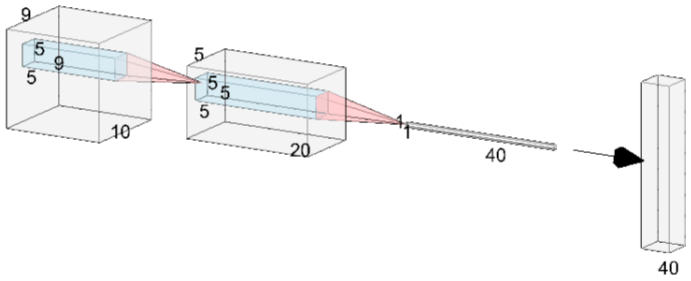
The two networks differ at the output after the linear layer - the policy network has a softmax layer to output probabilites, the Q network doesn't, as it outputs the value of a state and action.

Optimization:

- Policy network - RMSProp , lr=5e-5.
- Q network- Adam, lr=0.001.

- Normalization of the rewards by 5, s.t we $\forall t : r_t \in [-1, 1]$, this is for stabilization of the training process.

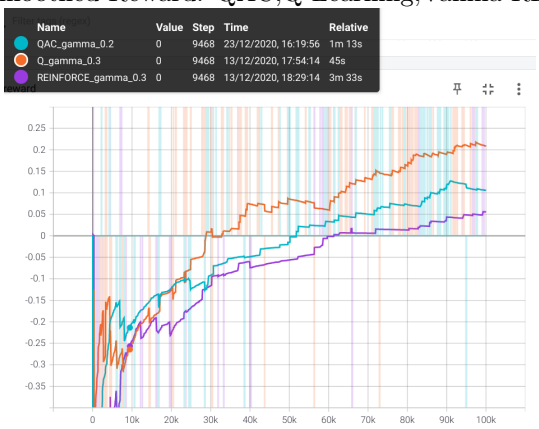
Figure 1: The neural network architecture



Analysis

We compare QAC with other algorithms.
First, we compare QAC($\gamma = 0.2$) with Q-Learning($\gamma = 0.3, \epsilon = 0.3$), and Vanilla-REINFORCE($\gamma = 0.3$):

Figure 2: Smoothed Reward: QAC,Q-Learning,Vanilla-REINFORCE

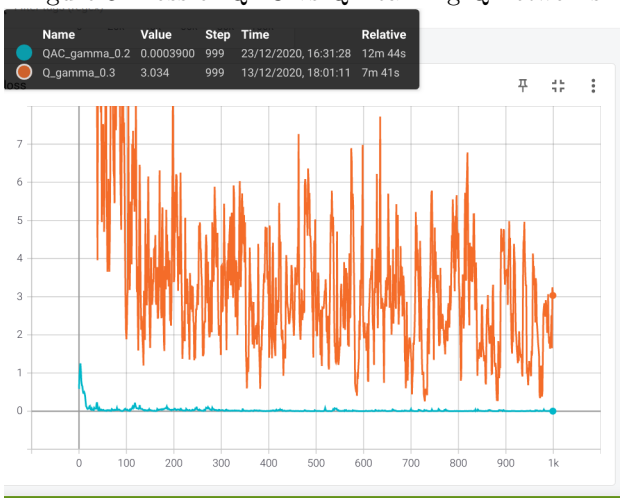


Notice that Q-Learning has the best smoothed reward plot, then QAC, and then REINFORCE. This result can be attributed to a couple of reasons:

- Q-Learning might perform the best because it approximates Q^*- the Q function of the optimal policy, while the Q network in QAC tries to maximize the advantage, and vanilla-REINFORCE tries to optimize by maximizing the likelihood of good actions from sampled episodes.
- Vanilla-REINFORCE is harder to optimize, and required careful hyperparamter tuning in order to converge to positive rewards, while Q-Learning is easier to optimize. Vanilla-REINFORCE only converged with RMSProp with a low learning rate(5e-5).

Another metric we used for comparison is the loss of the Q-network in QAC compared to the loss of the Q-network in Q-Learning:

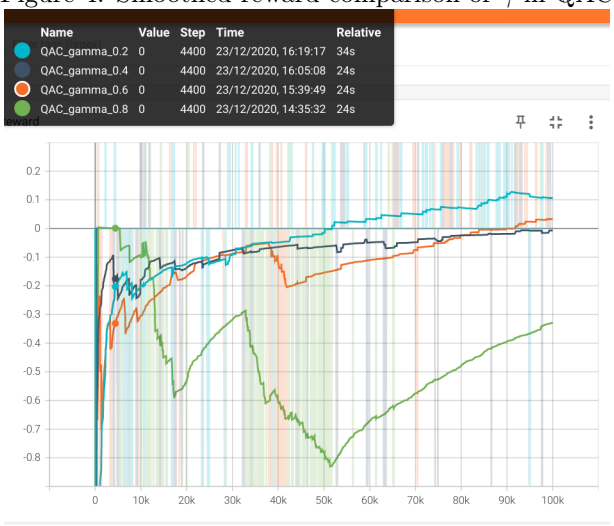
Figure 3: Loss of QAC vs Q-Learning Q-networks



Notice that the loss of Q is much noisier and higher, this might be because that it lacks the actor in the actor-critic architecture

Next, we compare different γ for QAC:

Figure 4: Smoothed reward comparison of γ in QAC



Notice that the smoothed reward is higher for smaller γ, this can be due to the reasoning that higher γ puts more weight for future reward, and perhaps this model doesn't handle that correctly, our hypothesis is that the critic network isn't a good estimator of the expected reward for high γ.

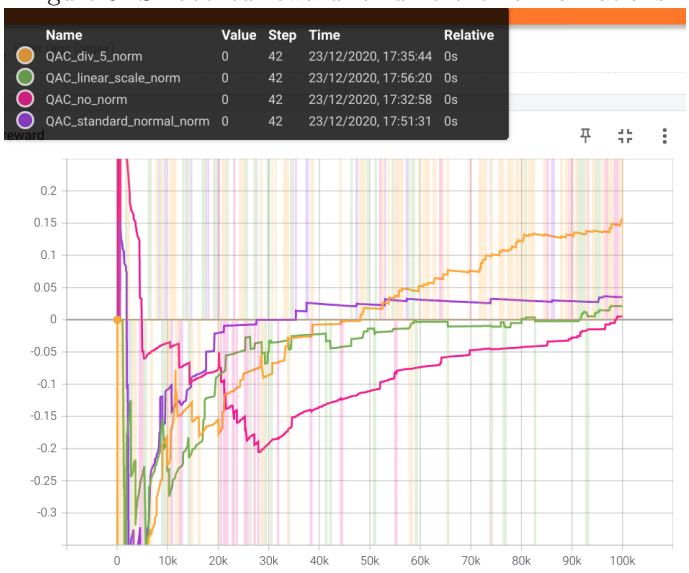
Comparison Between Normalizations

We now compare how different normalizations of the reward effect the performance of QAC.
We use the following normalizations(with the corresponding colors in the graph below):

1. No normalization at all (red)
2. $R_t =: R_t/5$ (yellow)
3. $R_t = \frac{R_t - \bar{R}}{\sqrt{Var(R)}}$ (purple)
4. $R_t = \frac{R_t - min(R)}{max(R) - min(R)}$ (green)

Let us now compare the smoothed rewards of QAC with the above normalizations:

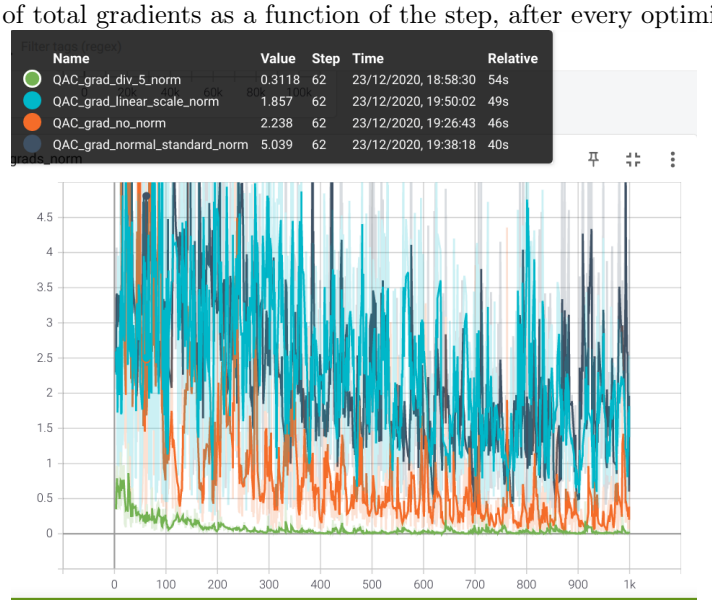
Figure 5: Smoothed reward for different normalizations



As can be seen in the graph above, we get the highest smoothed reward for division by 5, the other normalization methods aren't as good and have similar performance between them. This can be attributed to a couple of reasons:

- Normalization in general will improve performance because it stabilizes training, high magnitude rewards might cause instability because of exploding gradients. This explains why no-normalization achieves the worst result. In figure (6) we can see the total norm of the gradients of the model in each step, for each normalization method.
- In different batches we might get different rewards, and so normalization methods which depend on the particular batch might cause a distortion in the proportional rewards between batches. This may result in bad performance as can be seen above.
- Normalization by division by 5 probably performs the best because it is independent of the particular batch, and the rewards are between $[-1, 1]$.

Figure 6: Norm of total gradients as a function of the step, after every optimization iteration



As can be seen, indeed the the gradients for non-normalized rewards are much larger and noisier, and the gradients look the smoothest for a division by 5 normalization.

We also wanted to watch how our model plays, so we modified snaker_wrapper.py to use our trained model, the video: https://www.cs.huji.ac.il/~guy_lutsker/APML_EX4_GroupPart_SnakeVideo.mp4