Noname manuscript No.
(will be inserted by the editor)

Deep Deterministic Policy Gradients: Components and Extensions

Yannik Frisch · Tabea Wilke ·
Maximilian Gehrke

The introduction should contain
- a brief motivation for RL
- a brief(!) description of DPG, DQN and DDPG and their connections (no math, just the main intuitions)
- The purpose of this paper
Basically extend section 1 substantially to give more context and move section 1.1-1.3 to a seperate section (e.g. Preliminaries)

Received: date / Accepted: date

Abstract TODO

Keywords DDPG · DQN · DPG

1 Introduction

Deep Deterministic Policy Gradients (DDPG) arises from Deterministic Policy Gradients (DPG) and Deep Q-Learning (DQN). In the following we describe the underlying algorithms DPG and DQN and which aspects DDPG uses of both of them.

dont use brackets and parantheses in your cites, try \citep{}

1.1 Deep Q-Learning

~~Does not belong here~~

~~cite Q-Learning and briefly introduce it. Show the Q-Learning update rule before you say that DQN combines it with NNs.~~

The Deep Q-Network approach (DQN) [Mnih et al. (2013)] combines the approximation power of Neural Networks with traditional Q-learning.It enables solving the classic Reinforcement Learning problem ~~of achieving the maximum expected reward over time~~, even for large state spaces (e.g. image frames). The algorithm is an off-policy, model-free approach and is able to find a close to optimal action-value function for many cases and from this a close to optimal deterministic policy by greedily selecting the action: $\pi(s) = \max_a Q^*(s, a)$. ~~In~~

cites?

F. Author
first address
Tel.: +123-45-678910
Fax: +123-45-678910
E-mail: fauthor@example.com

S. Author
second address

use \boldsymbol{} and \mathbf{} for vectors and scalars

~~what is s, a, T? define MDP in preliminaries. Discounting is also not DQN-specific.~~

~~term of a formula~~ The optimal action-value function is represented by

~~use p' or $p^{\prime}$ but not $p^{'}$~~

$$Q^*(s_t, a_t) = \max_\pi E\left[\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} | s_t = s, a_t = a, \pi\right]$$

where $\gamma \in [0, 1)$ is called the *Discount Factor*, controlling the agents preference for rewards closer or further away in time. Rewards later on in an episode will still have impact on the result but their influence decreases by the amount of time-steps required to reach them in the future. By definition this optimal value function yields the *Bellman Equation* [Sutton and Barto (2018)] and can be reinterpreted as maximizing the current reward and the discounted action-value of the resulting state. In formula this gives:

Only capitalize names

$$Q^*(s, a) = E_{s' \sim \epsilon}\left[r + \gamma \max_{a'} Q^*(s', a') | s, a\right]$$

~~what is epsilon?~~

For approximating the action-value function $Q(s, a|\theta) \approx Q^*(s, a)$ the approach uses a deep neural network, called the Q-Network. The Q-Network can be trained by minimizing a sequence of loss functions $L_i(\theta_i)$, depending on it's weights:

~~What is theta?~~

~~Call them parameters~~

~~minimize sequences or rather sequentially minimize?~~

$$L_i(\theta_i) = E_{s,a \sim \rho(.), s' \sim \epsilon}\left[\left(r + \gamma \max_{a'} Q(s', a'|\theta_{i-1}) - Q(s, a|\theta_i)\right)^2\right]$$

~~maybe write ^{(i)} for denoting learning iterations. \theta_i would usually relate to the i-th parameter.~~

This loss function is similar to the classical temporal-difference loss used in Q-Learning, but with approximated action-value functions instead of lookup-tables. Derivating this loss w.r.t. the approximation's weights gives:

$$\nabla_{\theta_i} L_i(\theta_i) = E_{s,a \sim \rho(.), s' \sim \epsilon}\left[\left(r + \gamma \max_{a'} Q(s', a'|\theta_{i-1}) - Q(s, a|\theta_i)\right) \nabla_{\theta_i} Q(s, a|\theta_i)\right]$$

~~Also: L_i is a function of \theta, not \theta_i~~

~~approximated~~

The expectation can be obtained by sampling from an environment and this gradient can be used to optimize the loss function by using stochastic gradient descent.

Furthermore, a replay buffer is used which stores samples of the environment. This allows random mini-batch sampling, which decorrelates the samples and is proven to improve the ~~performance by greater~~ data efficiency [x]. The mini-batch sampling also enables the use of improved derivatives of vanilla stochastic gradient descent, e.g. *RPROP* as in the *Neural Fitted Q-Learning* approach [Riedmiller (2005)] or *ADAM Update* [Kingma and Ba (2014)].

is this really proven? Better motivate the decorrelation.

"The minibatch sampling" -> "Sampling mini-batches" Also: are mini-batches really necessary for RPROP/ADAM?

Reformulate, motivate why we need to fix the weights

There are different ways of estimating the expected Q-values. Either with a target network with the same structure as the network for the action-value function or the normal network. If a target network is used, the target weights need to be updated after some training steps [Mnih et al. (2015)].

A pseudo-code for the DQN approach can be found in 1. The DQN approach was able to significantly outperform earlier learning methods despite incorporating almost no prior knowledge about the inputs [Mnih et al. (2013)]. ~~but~~ is limited by the disability to cope with continuous and high-dimensional action spaces due to the max operator in the action selection [Lillicrap et al. (2015)]. This limitations can be adressed by combining the approach with the Deterministic Policy Gradient, which is described in the following section.

~~However, it~~

~~[...] algorithm 1~~

Explain: Why can't we maximize a continuous function?

Algorithm 1 Deep Q-Learning (DQN)

Initialize: Replay buffer D with high capacity
Initialize: Neural network for action-value function Q with random weights θ
Initialize: Neural network for target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
  for episode 1 **to** M **do**
    reset environment to state s_1
    for $t = 1$ **to** T **do**
      if random $i \leq \epsilon$ **then**
        random action a_t
      else
        $a_t = \text{argmin}_a Q(s_t, a|\theta)$ argmin?
      end if
      execute $a_t \rightarrow$ reward r_t and next state s_{t+1}
      save (s_t, a_t, r_t, s_{t+1}) in D
      sample mini-batch (s_i, a_i, r_i, s_{i+1}) from D mini-batch should be a set of tuples, not a single tuple
      $q_i = \begin{cases} r_i & \textit{if episode terminates at step i+1} \\ r_i + \gamma \max_{a'} \hat{Q}(s_{i+1}, a'|\theta^-) & \textit{else} \end{cases}$
      perform gradient descent on $(q_i - Q(s_i, a_i|\theta))^2_\theta$
      every C steps update $\hat{Q} = Q$
    end for
  end for

1.2 Deterministic Policy Gradient

why?

~~As [Lillicrap et al. (2015)] stated correctly,~~ most problems in reinforcement learning consist of a continuous action space which makes it very difficult to greedily choose the best action given a policy, due to the max operator. From a stochastic point of view, the policy is a probability distribution $a \sim \pi(a|s)$ over all actions. In order to calculate the gradient of a parameterized policy $\pi(a, s|\theta)$ over the total reward w.r.t. the weights, one needs to solve an integral over all actions and states, which becomes intractable for large state-action spaces. From a deterministic view the policy is a discrete mapping from states to actions $a = \pi(s)$ and thus only one integration over the state space is sufficient.

It's not just a matter of view, whether a policy ist stochastic or deterministic - it's a design choice.

why? refer to specific equation

It's not clear, what computation you are refering to.

The *policy gradient theorem* [Silver et al. (2014)] gives the update rule for a parameterized policy, optimizing the loss function:

badly formatted are you using $\log$?

$$\nabla_\theta J(\theta) = E_{s\sim\rho^\pi, a\sim\pi_\theta}\left[\nabla_\theta log \pi_\theta(a|s) Q^\pi(s, a)\right]$$

Use primary sources! The policy gradient theorem is not from Silver et al. (2014).

From this, a deterministic approach is derived in [EDIT: by?] [Silver et al. (2014)], which gives the update rule for a parameterized deterministic policy function $\pi(s|\theta)$. Rather than trying to maximize the action-value function $Q(s, a)$ globally by greedy improvements of the policy, the authors move the policy in the direction of the gradient of $Q(s, a)$:

Where does this come from?

What is p^\beta?

$$\nabla_{\theta^\pi} J \approx E_{s_t\sim\rho^\beta}\left[\nabla_{\theta^\pi} Q(s, a|\theta^Q)\right]$$

Applying the chain rule to this equation gives the *deterministic policy gradient (DPG) theorem*:

$$\nabla_{\theta^\pi} J \approx E_{s_t\sim\rho^\beta}\left[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\pi(st)} \nabla_{\theta^\pi} \pi(s|\theta^\pi)|_{s=s_t}\right]$$

[...] approximated from samples.

where the expectation can again be ~~obtained from sampling from an environment~~. Only using deterministic action outputs will vanish the algorithms exploration, so ~~one~~ needs to make sure there still is exploration [EDIT: FORMULATION]. This is realized by using an off-policy approach which follows a stochastic ~~behavior~~ policy while learning ~~about~~ a deterministic ~~target~~ policy. The authors also introduce the notion of *compatible function approximation*. Using these to estimate the gradient, an unbiased approximation is guaranteed.

too vague

The following section describes a typical structure of how to use deep neural networks for function approximation in reinforcement learning. Together with this section this led to the algorithm described in chapter 2.

1.3 Actor-Critic Methods

A lot of recent success in reinforcement learning is based on *Actor-Critic* methods [Konda and Tsitsiklis (2000)]. In contrast to value-function or policy-gradient methods, they parameterize both, the value function $Q(s,a) \approx \hat{Q}(s,a|\theta^Q)$, also known as the *Critic*, and the policy $\pi(s|a) \approx \hat{\pi}(s|a,\theta^\pi)$. To get an intuition about these methods figure 1 illustrates the update-cycle:

In the figure, the critic is V(s), in the text it is Q(s,a). Explain!

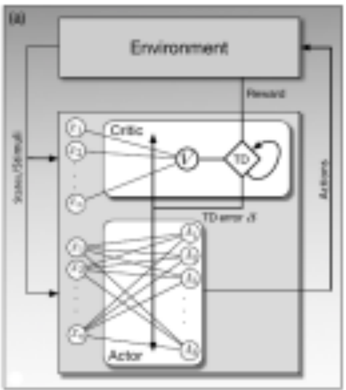

Fig. 1 Intuition about actor-critic methods (figure from Sutton and Barto (2018))

While the actor learns how to choose the right action and is responsible to update his policy, the critic has to learn and update the parameters of the state-value function. The actors' and the critics' parameters can be updated following the TD-error of the critic which is calculated from the observed reward and the current error of the estimated state values in every time-step. As Fig. 1 illustrates, the actor has no information about the current reward and the critic has no direct influence on the actions. A pseudo-code for an actor-critic method using TD-errors is shown in algorithm 2.

computed

Cite algorithm

Algorithm 2 Episodic One-step Actor-Critic for Estimating $\pi(s|a, \theta^\pi) \approx \pi^*(s|a)$

Initialize: Differentiable policy parameterization $\pi(a|s, \theta^\pi)$
Initialize: Differentiable action-value function parameterization $Q(s, a|\theta^Q)$
Initialize: Random initial weights θ^π and θ^Q
Initialize: Step size parameters $\alpha^Q > 0$ and $\alpha^\pi > 0$
Initialize: Discount factor γ
  for episode 1 **to** M **do**
    Get initial state s
    $i \leftarrow 1$
    for time-step 1 to T **do**
      Draw action from actor: $a \sim \pi(s|a, \theta^\pi)$
      Do action a, observe reward r and successor state s'
      Calculate the TD-error:
        $\delta \leftarrow r + \gamma \max'_a Q(s', a'|\theta^Q) - Q(s, a|\theta^Q)$
      Update the weights:
        $\theta^Q \leftarrow \theta^Q + \alpha^Q \delta \nabla_{\theta^Q} Q(s, a|\theta^Q)$
        $\theta^\pi \leftarrow \theta^\pi + \alpha^\pi i \delta \nabla_{\theta^\pi} \log \pi(a|s, \theta^\pi)$
      Update:
        $i \leftarrow \gamma i$
        $s \leftarrow s'$
    end for
  end for

2 Deep Deterministic Policy Gradient

The combination of above approaches led to the *Deep Deterministic Policy Gradient (DDPG)* approach [Lillicrap et al. (2015)], which is a model-free and off-policy algorithm. It can be grouped into the class actor-critic methods and uses a deterministic target policy and deep Q-Learning. Both, the actor and the critic, are realized by deep neural networks. The pseudo-code for DDPG can be found in 3.
It consists of a parameterized deterministic policy, the actor, $\pi(s|\theta^\pi)$ and a parameterized action-value function $Q(s, a|\theta^Q)$, the critic. The critic is updated using the *Bellman Equation* with a TD-error similar in Q-Learning [Watkins and Dayan (1992)] [EDIT: EQUATION?] and the actor is updated using the DPG theorem [EDIT: LINK? EQUATIONS WITH NUMBERS?].
The use of neural networks to parameterize the above functions means that the convergence guarantees do not hold anymore. Therefore the Actor-Critic DPG approach is combined with recent successes from DQN. too vague
To ensure independently and identically distributed data, the authors use a replay buffer and sample random mini-batches from it. This again decorrelates the samples and allows the efficient use of hardware optimization, e.g. the ADAM update [Kingma and Ba (2014)]. How does ADAM relate to hardware optimizations?
To adress instability issues from applying deep neural network approximation to Q-Learning they also use *target networks* which are copies of the actor $\pi'(s|\theta^{\pi'})$ and the critic $Q'(s, a|\theta^{Q'})$. These target-networks track the learned networks and are constrained to slow changes by using soft updates: $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$ with $\tau << 1$. These consistent targets might slow down

the learning process but greatly improve the stability of it.
Using low dimensional feature input might give very different scales for the single states. This can lead to problematic learning for the neural networks and is adresses by using *batch normalization* which normalizes each dimension across the samples in a mini-batch.

How does the scale depend on the dimensionality of the input?

To ensure exploration while using a deterministic policy, a noise process N is added to the action output of the actor network. This noise process can be chosen to suit the environment. The algorithm was evaluated on more than

Algorithm 3 Deep Deterministic Policy Gradient (DDPG)

Initialize: Replay buffer D with high capacity
Initialize: Critic network $Q(s,a|\theta^Q)$ and actor network $\pi(s|\theta^\pi)$ with random weights θ^Q and θ^π
Initialize: Initialize target networks Q' and π' with weights $\theta^{Q'} \leftarrow \theta^Q$ and $\theta^{\pi'} \leftarrow \theta^\pi$
  for episode 1 **to** M **do**
    Initialize random process N for action exploration
    Reset environment to state s_1
    for $t = 1$ **to** T **do**
      Select action $a_t = \pi(s_t|\theta^\pi) + N_t$ from local actor
      Execute action a_t and observe reward r_t and next state s_{t+1}
      Save (s_t, a_t, r_t, s_{t+1}) in replay buffer D
      Sample mini-batch (s_i, a_i, r_i, s_{i+1}) from D
      Set TD-target from target networks:
        $y_i = r_i + \gamma Q'(s_{i+1}, \pi'(s_{i+1}|\theta^{\pi'})|\theta^{Q'})$
      Update the critic by minimizing the loss:
        $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
      Update the actor using the sampled policy gradient:
        $\nabla_{\theta^\pi} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\pi(s_i)} \nabla_{\theta^\pi}\pi(s|\theta^\pi)|_{s=s_i}$
      Update the target networks:
        $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
        $\theta^{\pi'} \leftarrow \tau\theta^\pi + (1-\tau)\theta^{\pi'}$
    end for
  end for

20 simulated physical tasks using the same algorithm, network structures and hyper-parameters, including classic control problems like the cart-pole environment. Using low-dimensional feature input, it was able to find policies performing really well on most of the tasks. Their performance is competitive with these found by a controller with full access to the environment. The algorithm is even able to find good policies using high dimensional pixel input. For simple tasks this is ~~just~~ as fast as using low dimensional state features.
The most challenging issues of the approach are his poor sample efficiency and some instabilities. We present some possible extensions to DDPG in the next chapter which might improve on these issues.

TODO:

- discretizing the action space often suffers from the course of dimensionality

– naive extension of DPG with nns turns out to be unstable for challenging problems

3 Improvements for DDPG

It is sometimes not clear, whether the improvements are suggested by you, or from related work

Despite it's good performance on many simulated tasks there is still ~~some~~ room to improve the DDPG algorithm. We show some possible extensions for it in ~~this~~ section.

Specify the problems more clearly

3.1 Using importance sampling to sample from the replay-buffer

In practice the algorithm is limited by the maximum storage size N of the replay-buffer D. Overwriting older samples by current ones does nowhere differentiate between more or less important experiences, because uniform random samples does weight all experiences equally. One could use a technique similar to *prioritized sweeping* [Moore and Atkeson (1993)] which uses *importance sampling* [Glynn and Iglehart (1989)] to prefer transitions which are more important over ones that have less value for the training process.

Too vague

3.2 Using Action Noise in Parameter Space

Instead of adding noise to the action space to ensure exploration, one could add adaptive noise directly to the parameters of the neural network [Plappert et al. (2017)]. This would add some randomness into the parameters of the agent and therefore into the decision it makes, while still always fully depending on it's current observation about it's environment. This parameter noise makes an agent's exploration more consistent and results in a more effective exploration, increased performance and smoother behavior.

Why?

3.3 Evolutionary Approaches

One can consider an even more extreme case of the above mentioned extension, which would be the use of *Evolutionary Strategies* to approximate the gradient of our objective function [Salimans et al. (2017)]. This does not require backpropagation ~~at all~~ and is competitive with sate of the art RL.

Not clear what you are suggesting here. Replacing DDPG completely by ES?

debatable

3.4 Improvements of the Deep Neural Network Architectures

TODO

4 Conclusion

References

Glynn PW, Iglehart DL (1989) Importance sampling for stochastic simulations. Management Science 35(11):1367–1392

Kingma DP, Ba J (2014) Adam: A method for stochastic optimization. arXiv preprint arXiv:14126980

Konda VR, Tsitsiklis JN (2000) Actor-critic algorithms. In: Advances in neural information processing systems, pp 1008–1014

Lillicrap TP, Hunt JJ, Pritzel A, Heess N, Erez T, Tassa Y, Silver D, Wierstra D (2015) Continuous control with deep reinforcement learning. arXiv preprint arXiv:150902971

Mnih V, Kavukcuoglu K, Silver D, Graves A, Antonoglou I, Wierstra D, Riedmiller M (2013) Playing atari with deep reinforcement learning. arXiv preprint arXiv:13125602

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

Moore AW, Atkeson CG (1993) Prioritized sweeping: Reinforcement learning with less data and less time. Machine learning 13(1):103–130

Plappert M, Houthooft R, Dhariwal P, Sidor S, Chen RY, Chen X, Asfour T, Abbeel P, Andrychowicz M (2017) Parameter space noise for exploration. arXiv preprint arXiv:170601905

Riedmiller M (2005) Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In: European Conference on Machine Learning, Springer, pp 317–328

Salimans T, Ho J, Chen X, Sidor S, Sutskever I (2017) Evolution strategies as a scalable alternative to reinforcement learning. arXiv preprint arXiv:170303864

Silver D, Lever G, Heess N, Degris T, Wierstra D, Riedmiller M (2014) Deterministic policy gradient algorithms. In: ICML

Sutton RS, Barto AG (2018) Reinforcement learning: An introduction. MIT press

Watkins CJ, Dayan P (1992) Q-learning. Machine learning 8(3-4):279–292