Backpropagation-Free Parallel Deep Reinforcement Learning

William H. Guss
Machine Learning at Berkeley
2650 Durant Ave, Berkeley CA, 94720
wguss@ml.berkeley.edu

Mike Zhong
Machine Learning at Berkeley
Berkeley CA, 94720
mlyzhong@berkeley.edu

James Bartlett
Machine Learning at Berkeley
Berkeley CA, 94720
bartlett@ml.berkeley.edu

Max Johansen
Machine Learning at Berkeley
Berkeley CA, 94720
max@ml.berkeley.edu

Abstract

In this paper we conjecture that an agent, envirionment pair (μ, E) trained using DDPG with an actor network μ and critic network Q^μ can be decomposed into a number of sub-agent, sub-environment pairs (μ^n, E^n) ranging over every neuron in μ; that is, we show empircially that treating each neuron n as an agent $\mu^n : \mathbb{R}^n \to \mathbb{R}$ of its inputs and optimizing a value function Q^{μ^n} with respect to the weights of μ^n is dual to optimizing Q^μ with respect to the weights of μ. Finally we propose a learning rule which simultaneously optimizes each μ^n without error backpropogation achieving state of the art performance and speed across a variety of OpenAI Gym environments.

Todo list

1 Introduction

Introduction to DDPG and recent advances in deep RL.

The problem statment is here. Biological diffusion of dopamine in the brain $\implies$ error backpropagation is not biologically feasible.

Recent work has attempted to address the issue of decoupling connections between layers in the network using decoupled neural interfaces [**?**]. The initial coupling is due to the reliance on the forwards and backwards computations to compute the backpropagated error gradient for a given layer. Synthetic gradient modules model the error gradient using only local information, allowing immediate feedback that is later corrected when the backpropogated error is finally computed. This method has had success in both feedforward and recurrent architectures, where the network is able to model much greater time horizons.

Although synthetic gradients decouple the network modules to some degree, this technique still relies on synchronous backpropagation to learn the local regressors. In addition, they model at a lower granularity, learning layer-level gradients instead of those of the individual neurons.

Coupling synthetic gradients with dopamine we get blah. Therefore it is feasible that each neuron is maximizing the expectation on his future dopamine intake, and so we propose the following theorem.

2 Agent-Environment Value Decomposition

In this section we will develop a theoretical basis for decomposing the agent and its environment in to neuromorphically local agent which act on the activations of their priors. We then will show that under some mild conditions, these agents act in environments which are so simple that it suffices to estimate the policy gradient using a linear approximation, much in the style of synthetic gradients. These results lead to a new decentralized, local, and asynchronous learning paradigm for deep reinforcement learning without the use of deep error-backpropagation.

2.1 Background

Recall the standard reinforcement learning setup. We say E is an *environment* if $E \overset{\text{def}}{=} (\mathcal{S}, \mathcal{A}, \mathcal{R}, T, r)$ where T describes transition probability measure $T\left(s_{t+1} \mid s_t, a_t\right)$ and $r : \mathcal{S} \times \mathcal{A} \to \mathcal{R}$ is a reward function. Furthermore $\mathcal{S}$, $\mathcal{A}$, $\mathcal{R}$ are the *state space, action space, and reward space* respectively. We restrict $\mathcal{R}$ to a compact subset of $\mathbb{R}$ and action space and state space to finite dimensional real vector spaces. As in DDPG we assume that the environment E is *fully observed*; that is, at any time step the state s_t is fully described by the observation presented, x_t, and not by the history $(x_1, a_1, \ldots, a_{t-1})$.

We define the policy for an agent to be $\mu : \mathcal{P}(\mathcal{A}) \times \mathcal{S} \to [0,1]$. In general the policy is a probability measure on some σ-algebra $\mathcal{M} \subset \mathcal{P}(\mathcal{A})$ conditioned on $\mathcal{S}$ so that $\mu\left(\mathcal{A} \mid s \in \mathcal{S}\right) = 1$. However, we will deal only with *deterministic* policies where for every s_t there is unique a_t so that $\mu\left(\{a_t\} \mid s = s_t\right) = 1$ and the measure is 0 otherwise. Thus we will abuse notation and define a *deterministic agent* by a policy function $\mu : \mathcal{S} \to \mathcal{A}$. Additionally we denote the state-space trajectories of μ by

$$\Gamma_\mu(\mathcal{S}) = \left\{((s_1, a_1), (s_2, a_2) \ldots) \mid s_1 \sim T(s_0), s_{t+1} \sim T\left(s \mid s_t, \mu(s_t)\right)\right\}. \tag{2.1.1}$$

For a policy μ the action-value function is the expected future reward under μ by performing a_t at state s_t. A temporally local definition thereof can be obtained using the Bellman equation

$$Q^\mu(s_t, a_t) = \mathop{\mathbb{E}}_{s_{t+1} \sim E}\left[r(s_t, a_t) + \gamma Q^\mu(s_{t+1}, \mu(s_{t+1}))\right] \tag{2.1.2}$$

with $\gamma \in (0,1)$ a discount factor, and the second expectation removed because μ is deterministic. [Some survey] provides an extensive exposition into a justification of this equation and choice for the action-value of μ, so we will assume such a choice is a valid measure of performance.

Insert reference and make this a footnote

Among a variety of methods, the deep reinforcement learning approach to solving environments (MDPs) has been predominately been separated into deterministic policy gradient methods and direct, optimal Q-Learning methods. In deterministic policy gradient (DPG) methods, we define an actor $\mu : \mathcal{S} \to \mathcal{A}$ and a critic $Q^\mu : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ and optimize $Q^\mu(s_t, \mu(s_t|\theta^\mu))$ with respect to the paramers θ^μ of μ. Deep determinsiitc policy gradient (DDPG) learning directly learns to approximate Q_a by creating two different deep neural networks for the actor and the critic and then back-propagating the Q gradient from the critic to the actor. Specifically, DDPG maximizes $Q^\mu(s_t, \mu(s_t|\theta^\mu))$ with respect to the paramers θ^μ of μ. This method is provably the true policy gradient of μ if Q^μ is known. In order to decompose the action-value function we will make heavy use of this methodology at a scale local to each neuron in the flavor of (Synthetic gradients.)

Cite deepmind

2.2 Towards Neurocomputational Decomposition of Q^μ

In order to decompose the Q^μ algorithm we will abstractly define a neurocomputational agent in terms of an operator on voltages with no restrictions on the topology of the network, and then relate the action-value function of the whole agent to those which are defined for each individual neuron in the network.

Definition 2.2.1. *If $\mathcal{V}$ is an N-dimensional vector space then a* ***neurocomputational agent*** *is a tuple $\mathcal{N} = (\mu, \epsilon, \delta, K, \Theta, \sigma, D)$ such that:*

- *$\epsilon : \mathcal{S} \to \mathcal{V}$ encodes the state into the voltages. Realistically, only a subset of all neurons are input neurons, denoted as $N_I \subset \mathcal{V}$, so $\epsilon(s_t) = \pi_{N_I}(\epsilon(s_t))$, where $\pi_K(x)$ is the cannonical projection of x on dimension(s) K,*

- $\delta : \mathcal{V} \to \mathcal{A}$ *decodes the voltages of the* output neurons $N_O \subset V$ *into an action, so that* $\delta(v_t) = \delta(\pi_{N_O}(v_t))$.
- $K : \mathcal{V} \to \mathcal{V}$ *is the linear voltage graph transition function of the graph representing the topolopy of* $\mathcal{N}$*, parameterized by* θ.
- $\Theta : \mathcal{V} \to \mathcal{V}$ *is a nonlinear inhibition function.*
- $\sigma : \mathcal{V} \to \mathcal{V}$ *is the elementwise application of some activation function to the voltage vector.*
- $D : \mathcal{V} \times \mathcal{V} \to \mathcal{V}$ *is called voltage dynamic of* $\mathcal{N}$ *such that*

$$v_{t+1} \overset{def}{=} D(v_t, v_{in}) \overset{def}{=} \sigma\left(\Theta K[v_t]\right) + v_{in} \tag{2.2.1}$$

where v_t *is the internal voltage vector of* $\mathcal{N}$ *at time* t *and* v_{in} *is an input voltage to the network. We will occasionally abuse notation an say that* $D(v_t) = D(v_t, 0)$ *when* v_{in} *is 0.*

- $\mu : \mathcal{S} \to \mathcal{A}$ *is the deterministic policy for* $\mathcal{N}$ *such that*

$$\mu(s_t) = \delta(D(v_t, \epsilon(s_t)) \tag{2.2.2}$$

It is not hard to see that this definition encompasses any DQN or DDPG network with either reccurent or non recurrent layers. Additionally other paradigms such as the leaky integrattor are neurocomputational agents. In this paper we will mainly discuss standard feed forward neural networks, in which case Θ is not defined and the neural dynamics are repeatedly applied until an output of N_O is produced (See Appendix A.)

Show equivalence in the appendix.

Definition 2.2.2. *If* n *is some neuron in* $\mathcal{N}$*, we say* $E^n = (\mathcal{S}, \mathcal{A}, \mathcal{R}, T^n, r^n)$ *is a deterministic* ***sub-environment*** *of* E *with respect to* $\mathcal{N}$ *if*

- *The state space is* $\mathcal{S} = \mathcal{V}$*; that is, the environment that neuron* n *observes is the voltages of all other neurons. Although this definition permits a fully connected graph for* $\mathcal{N}$*, realistically, each neuron only sees the voltages of a subset of all neurons. In the case of standard feed forward networks,* n *would observe only the activations of the previous layer.*
- *The action space is the set of voltages* $\mathcal{A} = \mathbb{R}$ *which the neuron* n *can output.*
- *The reward space* $\mathcal{R}$ *is the same as that of the base environment* E*. Furthermore the subenvironment emitts the same same reward as does the base environment. If the agent* μ *of* $\mathcal{N}$ *acts on a state* s_t *and receives a reward* r_t*, then every sub-environment emits the same reward* r_t*.*
- *The transition function* $T : \mathcal{V} \times \mathbb{R} \to \mathcal{S}^n$ *is such that*

$$T^n(v_t, \alpha_t) = (I - \chi_{n,n})D(v_t)) + e_n\alpha + \epsilon(s_t) \tag{2.2.3}$$

where e_n *is the* n^{th} *unit basis vector,* I *is the identity, and* $\chi_{n,n} = 1$ *if* $n = n$ *and* 0 *otherwise. Intuitively, the transition in* E^n *is the normal*[1] *neural dynamics on* $\mathcal{N}$ *except for at the neuron* n*, itself; in* E^n *we set the voltage of* n *in* v_{t+1} *to be the voltage chosen,* α_t*, plus the newly encoded voltage* $\epsilon(s_t)$.

Lastly an agent $\mu^n : \mathcal{V} \to \mathbb{R}$ *is called* ***neuromorphically local*** *to* $\mathcal{N}$ *if* $\mu^n : v_t \mapsto \langle D(v_t), e_n\rangle$*; that is,* μ^n *acts according to the normal dynamics.*

With the basic definitions given, we will now show performing reinforcement learning on $\mathcal{N}$ can be decomposed into the dual problem of perfoming reinforcement learning on agents and environments of type given in Definition 2.2.2.

[1] Since the state space of E^n is only $\mathcal{V}$, s_t is a hidden variable of the markov decision process, and T should be a function on $\mathcal{V} \times \mathcal{S} \times \mathbb{R}$, but this is ommitted for simplicity.

Theorem 2.2.3 (Neurocomputational Decompostion)**.** *Let E be an environment and $\mathcal{N}$ be a neurocomputational agent. Then there exists a set of agent environment pairs $\mathfrak{D}_{\mathcal{N}} \overset{def}{=} \{(E^n, \mu^n)\}_{n\in\mathcal{N}}$ such that for every $n \in \mathcal{N}$, the following diagram commutes*

$$\begin{tikzcd}
\mathcal{V} \times \mathcal{S} \arrow[r, "\mu\circ\pi_2"] \arrow[d, "\pi_1\times\epsilon\circ\pi_2"] & \mathcal{A} \\
\overbrace{\mathcal{V}\times\mathcal{V}}^{(v_t,\epsilon(s_t))} \arrow[r, "D"] \arrow[dr, "\mu^n\circ\pi_1+\pi_n\circ\pi_2"'] & \overbrace{\mathcal{V}}^{v_{t+1}} \arrow[u, "\delta"'] \arrow[d, "\pi_n"] \\
 & \mathbb{R}
\end{tikzcd} \tag{2.2.4}$$

Proof. If E and $\mathcal{N}$ are given, then for every $n \in \mathcal{N}$ let E^n be a sub-environment of E with respect to $\mathcal{N}$. Next let μ^n be a neuromorphically local agent in E^n with respect to $\mathcal{N}$.

We first show that (2.2.4) commutes. Let $v_t \in \mathcal{V}$ and $s_t \in \mathcal{S}$. Observe that

$$[\mu \circ \pi_2]\,(v_t, s_t) = \mu(s_t) = \delta\left(D(v_t, \epsilon(s_t))\right) = [\delta \circ D \circ (\pi_1 \times \epsilon \circ \pi_2)]\,(v_t, s_t),$$

and therefore the top half of the diagram commutes. Given some $(v_t, \epsilon(s_t)) \in \mathcal{V} \times \mathcal{V}$ we have that

$$\begin{aligned}
[\mu^n \circ \pi_1 + \pi_2]\,(v_t, \epsilon(s_t)) &= \mu^n(v_t) + \pi_n(\epsilon(s_t)), \\
&= \pi_n\left(D(v_t) + \epsilon(s_t)\right), \\
&= [\pi_n \circ D]\,(v_t, \epsilon(s_t)).
\end{aligned}$$

Thus the diagram in (2.2.4) commutes. □

Make a sub environment tikz diagram to help people understand waht this decomposition looks like.

Theorem 2.2.3 gives a natural connection between the state space trajectories of μ and μ^n because in $\mathcal{N}$, the voltage v_t is a hidden variable which governs action of $\mathcal{N}$ at any timestep and dually the state s_t is a hidden variable of the Markov Decision Process formed by E^n which governs the state given by T^n at any timestep.

Naturally, the following question arises: does DPG learning on $\mathcal{N}$, specifically μ on E, commute with performing DPG learning on on every $(E^n, \mu^n) \in \mathfrak{D}_{\mathcal{N}}$? Supposing that we have the true Q^μ function and μ is optimal with respect to Q^μ, then it is intuitive, but not obvious, that every μ^n should behave optimally with respect to an Q^{μ^n} – but will the reverse hold? To answer these questions we give the following result.

Theorem 2.2.4. *If $\mathcal{N}$ is a nuerocomputational agent in E, then policy gradient for μ agrees with the simultaneous policy gradients of its decomposition; that is for every $(E^n, \mu^n) \in \mathfrak{D}_{\mathcal{N}}$*

$$\nabla_{K^n} Q^{\mu^n}(v, \alpha)\Big|_{v=v_t,\alpha=\mu^n(v_t)} = \nabla_{K^n} Q^{\mu}(s, a)\Big|_{s=s_t,a=\mu(s_t)} \tag{2.2.5}$$

for every time step t, where K^n is the nth column of the linear voltage graph transition matrix, i.e. the weights of the connections from all neurons to neuron n.

Proof. Let $\mathcal{N}, E$ be given and fix $(E^n, \mu^n) \in \mathfrak{D}_{\mathcal{N}}$. For any initial state s_0, Theorem 2.2.2 gives that the state-action trajectory $\kappa \in \Gamma_\mu(\mathcal{S})$ generated by μ from s_0 is dual to the state-action trajectory $\kappa^* \in \Gamma_{\mu^n}(\mathcal{V})$ generated by μ^n from $v_0 = 0$ when the hidden state of T^n is s_t. Because E^n is a sub-environment of E, $r^n \equiv r$ the sequence of rewards on κ and κ^* are the same. Using the definition of the action-value function assuming that s_0 fixed,

$$Q^\mu(s_t, a_t) = \sum_{\tau=t}^{\infty} r(s_t, \mu(s_t))\gamma^{\tau-t} = \sum_{\tau=t}^{\infty} r^n(v_t, \mu^n(v_t))\gamma^{\tau-t} = Q^{\mu^n}(v_t, a_t) \tag{2.2.6}$$

where $\kappa = ((s_t, \mu(s_t)))_{t\in\mathbb{N}}$ and $\kappa^* = ((v_t, \mu^n(v_t)))_{t\in\mathbb{N}}$.

If (v_t, s_t) are give, Theorem 2.2.3 states that both μ^n and μ commute with the dynamics on $(v_t, \epsilon(s_t))$ (see the middle path in (2.2.4)). Thus if the dynamics are parameterized by $K^n \in \mathcal{K}_n$, application of the equality in (2.2.6), gives the following commutitive diagram, $\mathfrak{k}_n$:

$$\begin{tikzcd}
 & \mathcal{A} \arrow[dr, "Q^{\mu}(s_t,\cdot)"] & \\
\mathcal{K}_n \arrow[ur, "\mu(s_t)"] \arrow[r, "D(v_t,\epsilon(s_t))"] \arrow[dr, "\mu^n(v_t)"'] & \mathcal{V} \arrow[u, "\delta"] \arrow[d, "\pi_n"] & \mathbb{R} \\
 & \mathbb{R} \arrow[ur, "Q^{\mu^n}(v_t,\cdot)"'] &
\end{tikzcd} \tag{2.2.7}$$

Recall from category theory that differentiation is a functor on the category of C^1 manifolds $\mathbf{Man}^1$ because for any two morphisms $f, g \in \mathrm{Hom}(\mathbf{Man}^1)$, $\nabla f \circ g = \nabla f \circ \nabla g$ by chain rule. It then follows that the diagram $\nabla(\mathfrak{k}_n)$ commutes and so

cite manifold functor paper

$$\begin{aligned}
\nabla_{K^n} Q^{\mu^n}(v_t, \alpha)\Big|_{\alpha=\mu^n(v_t)} &= \nabla_\alpha Q^{\mu^n}(v_t, \alpha) \nabla_{K^n} \mu^n(v_t) \\
&= \nabla_a Q^{\mu}(s_t, a) \nabla_{K^n} \mu(s_t) \\
&= \nabla_{K^n} Q^{\mu}(s_t, a)\Big|_{a=\mu(s_t)}
\end{aligned}$$

Because this equality holds for any (s_t, v_t) and therefore any state-action trajectory with arbitrary initial hidden variables, the theorem holds. $\square$

Remarkably, the prevous theorem shows that optimizing every neuromorphically local agent μ^n with respect to a critic is exactly equivalent to optimizing the entire $\mathcal{N}$ with respect to a global critic Q^μ. Additionally optimization of the each μ^n does not require any backpropagation of $\nabla_a Q^{\mu^n}$ through the network since every Q^{μ^n} is diagramatically independent of eachother! As a result the neurocomputational decomposition $\mathfrak{D}_{\mathcal{N}}$ of $\mathcal{N}$ can be trained asynchronously and simultaneously.

Discuss Recurrent Neural Networks and infinite time horizon learning

2.2.1 Simplicity of Sub-Environments

An algorithm which learns $\mathcal{N}$ by performing DPG learning on every $(E^n, \mu^n) \in \mathfrak{D}_{\mathcal{N}}$ is only beneficial if each Q^{μ^n} can be approximated simply. For example if Q^{μ^n} are so complex that an approximation parameterized by K requires more than $|K^n| \notin O(1)$ parameters, then standard error-backpropagation is sureley superior.

However the results established in (Synthetic Gradients) show that the approximation of the error backpropagated signal can be empircally linear in the activations. In the framework of DDPG, this intuitively says that the infintesmal effect of the weights K^n on the dynamics and thereby the true Q^μ of a neurocomputational agent $\mathcal{N}$ might be linear in the local neural neightborhood of n. In the spirit of this hypothesis, we present a theorem on the simplicity of E^n and thereby Q^n.

Therefore we propose the following learning rule in aims to evidence the reverse, training μ using simultaneous optimization on all Q_n w.r.t π_n's weights.

3 Decentralized Deep Determinstic Policy Gradient Learning

Proposal of the rule. Linear approximation of the Q function for every neuron is good enough, (experimentally).

Implications of the rule to DDPG

Implications of the rule to entirely recurrent networks (infinite time horizon and NO unrolling since the environment the local actions of the neuron which globally recur to that neuron again are *encoded* into Q_n; large time horizion probably implies that better regresser needed for Q_n.)

Parallelism, no error backprop, and only 2x operations, but no locking on GPU, so all can be run sumultaneously if we cache!

4 Experimentation

4.1 Experiment 1: Learning Q functions on components of the Actor network.

Motivation: Our first batch of experiments aimed to establish empirically that the Q-functions of each layer, denoted $Q_1...Q_L$ are similar to the Q-function of the overall network, denoted Q_μ. To that end, we extended the actor-critic methodology used in Lillicrap et al (2016) as follows. In addition to training Q^μ, to estimate the Q functions of the actor network μ, sub-critic networks Q^n were initialized for each individual layer. We then compare the Q values estimated by the subcritics to the Q estimation provided by the main critic, Q^μ.

The inputs used in the calculation of the Q-functions for each layer follow the conceptual framework laid out in Section 2.2. The state and action parameters are therefore voltages represented by the outputs of $\epsilon : \mathcal{S} \to \mathcal{V}$ and $\delta : \mathcal{V} \to \mathcal{A}$ respectively, where the changes in voltage are described by a voltage transition function $K : \mathcal{V} \to \mathcal{V}$.

To determine and compare the rate at which the subcritics and the main critic learn, the experiment was run in two phases. First, each of the subcritics and the main critic were trained using the standard DDPG algorithm on some actor μ. In the second phase, a new actor μ' was initialized and its Q-function set to Q_μ as determined above in phase 1. The subcritics were trained, and the values of $Q_1 \dots Q_L$ were plotted as training occurred.

Introduce the notion of similary used to compare the q-functions.

After [number of iterations], each $Q^n(s, a)$ networks effectively learned the same cost function as $Q^\mu(s, a)$.

Incorporate metrics regarding the performance Q_n w.r.t Q^μ's weights

. Furthermore, the Q-value plots for each Q^ncorrelate directly with the Q^μ's Q plot. This confirms that we are able teach the Q^n model on the level of subcomponents of μ supporting the argument that we can use Q^n nets to approximate Q^μ. We can now use each Q^n to calculate the gradient of the specific neuron component that the Q^n is critiquing. We proceed with this in Experiment 2.

4.2 Experiment 2: Treating each neuron as its Actor-Critic network using linear approximators

Now that we have established that the Critic networks for each of the individual neuron learns the same Q-function as does the entire agent, we now treat each neuron as its own actor. Each neuron changes the weights of its presynaptic neurons' connections, as parameters for its actor – its voltage on the next timestep – to optimize its learned approximated Q function. We use the linear approximation:

$$Q^n(v, a) \approx \theta_{n,v}^T v + \theta_{n,a} a = \theta_n^T (v, a)^T$$
$$\mu^n(v) = \sigma(K^n v)$$

GOT RID OF Θ in that it's not very necessary ...

The algorithm for learning (very much INSPIRED by [CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING]):

For each neuron n:

```
        Initialize random weights $\theta_{n,v}$, $\theta_{n,a}$, and $K^n$
        Initialize target network $Q^{n\prime}$ and $\mu^{n\prime}$ with same weights, $\theta'_{n,v}$, $\theta'_{n,a}$, and $K'^n$
        Initialize replay buffer R
        For each episode:
            Receive initial observation voltages $v_1$
            for t=1, T, do:
                follow dynamics, $a^n_{t+1} = \sigma(K^n v_t)$
                Observe reward $r_t$, observe new voltages $v_{t+1}$
                Store transition $(v_t, a_t, r_t, v_{t+1})$ in R.
                Sample a random minibatch of $N$ transitions $(v_i, a_i, r_i, v_{i+1})$ from R
                Set $y_i = r_i + \gamma Q'(v_{i+1}, \mu'(v_{i+1}))$
                Update critic weights $\theta_n$ by minimizing loss $L := \frac{1}{N}\sum_i (y_i - Q(v_i, a_i))^2$
                Update actor weights (connections) using the sample policy gradient:
                    $\nabla_{K^n} J \approx \frac{1}{N}\sum_i \nabla_a Q(v,a)|_{v=v_i, a=\mu^n(v_i)} \nabla_{K^n}\mu^n(v)|_{v_i}$
                update the target networks:
                    $\theta'_n \leftarrow \tau\theta_n + (1-\tau)\theta'_n$
                    $K^{n\prime} \leftarrow \tau K^n + (1-\tau)K^{n\prime}$
            end for
        end for
```

We train the critic weights $(\theta_{n,v}, \theta_{n,a})$ by minimizing the loss:

$$L = \frac{1}{2}((r_i + \gamma Q^{n\prime}(v_{i+1}, a_{i+1})) - Q(v_i, a_i)) = \frac{1}{2}D_i^2$$

where $D_i := (r_i + \gamma\theta_n^{T\prime}(v_{i+1}, a_{i+1})) - \theta_n(v_i, a_i)$

We thus have:

$$\nabla_{\theta_n} L = -D_i(v_i, a_i)$$

or, using gradient descent with learning rate η_Q, we have:

$$\theta_n -= \eta_Q \nabla_{\theta_n} L = -\eta_Q D_i(v_i, a_i)$$

or:

$$\theta_{n,v} += \eta_Q D_i v_i$$

and

$$\theta_{n,a} += \eta_Q D_i a_i$$

As for updating the actor policy, with learning rate eta_A, we have:

$$K^n += \eta_A \nabla_a Q(v,a)|_{v=v_i, a=\mu^n(v_i)} \nabla_{K^n}\mu^n(v)|_{v_i} = \eta_A \theta_{n,a}\sigma'(K^n v_i)v_i$$

IN CONCLUSION:

$$\theta_{n,v} += \frac{1}{N}\sum_i \eta_Q D_i v_i$$

$$\theta_{n,a} \mathrel{+}= \frac{1}{N}\sum_i \eta_Q D_i a_i$$

$$K^n \mathrel{+}= \frac{1}{N}\sum_i \eta_A \theta_{n,a}\sigma'(K^n v_i)v_i$$

where $D_i := (r_i + \gamma\theta_n^{T'}(v_{i+1}, a_{i+1})) - \theta_n(v_i, a_i)$

Emperical justification of the iff using the following experiment (s).

Intuitively, it should be the case that by treating each neuron in the neural network as its own Q-learner, the policy gradient of an individual agent should be the same as for the policy gradient entire agent, if each neuron sees the same reward r_t as the entire agent. In other words, the optimal way of updating the entire connection matrix K should be the optimal way for updating the weights connected to each neuron w.r.t. each neuron as its own Q-learner, given the same rewards.

3. Plot the output of both Q^μ and Qn using TENSORFLOW summaries.

From Experiment 1, we see that there is a correlation between the Q-gradient for neuron n, and the nth column of the gradient of the agent's Q function:

from strong to weak:

$$\nabla_{K^n} Q^{\mu^n}(v,a) = \nabla_{K^n} Q^\mu(s,a)$$

$$\nabla_{K^n} Q^{\mu^n}(v,a) \propto \nabla_{K^n} Q^\mu(s,a)$$

$$corr[\nabla_{K^n} Q^{\mu^n}(v,a), \nabla_{K^n} Q^\mu(s,a)] \approx EMPERICALVALUE$$

although (assuming we get a weak experimental result), we propose that it is due to our neurons not being the ideal Q-learner (convergence issues, etc.).

1. Training a network on Atari using DDPG and plotting average critic functions for neurons using window.

EXPERIMENT 1 SPECIFICATION. 1. Set up a standard DDPG to play the set of atari games in OpenAi Gym using TENSORFLOW (this will be mac,linux, or windows bash only). If on Windows bash install Xming (its an X server) and run all OpenAI Gym commands with `DISPLAY=localhost:0.0 python3 src/experiment1/some__script_in_src.py`. This will pipe the visual output fo the OpenAI Gym simulators to the display. If you cannot get this to work on your screen, do not do env.render. We can also stop using the atari games, since this works for a fact on the basic Box2d versions. **We are going to write all of this in Python3, make sure to install gym in python3.**
2. DDPG has a Q^μ network which we use to optimize $\mu(s_t \mid \theta)$ with respect to θ. The goal of this experiment is to train a standard DDPG network to play one of these OpenAI Gym simulations, whislt concurrently estimating and viewing the Q functions for every single neuron. THEREFORE, we need to select a subset of neurons in the fully connected layers (for example) of the μ network (actor) and concurrently train a network $Q^n(s,a)$ to estimate the Q function of the neuron based on its inputs s and its SINGLE output voltage s. This can be a 3 layer fully connected network with $|s| + 1$ inputs (one for each input the neuron n and 1 for the voltage of the neuron after receiving that output.) Tensorflow is a dataflow language so the output of a layer looks like $O2 = \sigma(W * O1)$. Therefore you just need to make another "network" whose dataflow could be like $Qn = \sigma(W_3^n * \sigma(W_2^n * \sigma(W_1^n * \text{concat}(O1, O2[i]))))$ where n is the ith neuron on layer $O2$. And $O1, O2, O3$ are the outputs of the neurons on those layers in the network. Then as in standard DQN you tain Qn with a lag network $W(Qn') = W(Qn)(1-\tau) + \tau W(Qn)$ where W denotes the weights of Qn, say $W_3^n, W_2^n, W_1^n, \ldots$. And then actually do gradient decent on the weights of Qn not Qn' by minimizing the following bellman equation

$$L(s_t, a_t, r_t, s_{t+1}, a_{t+1}) = (Qn(O1(s_t), O2[i](s_t)) - r_t - Qn'(O1(s_{t+1}), O2[i](s_{t+1})))^2$$

with respect to the parameters $W(Qn)$. Note I didn't actually use a_t above since really the Qn function takes in the input of the previous layer (to n) as its input, say $O1(s_t)$ and the action for that same time step which is just the output of the neuron n, say $O2(s_t)[i]$. The same goes for Qn' but at the next time step.

5 Results

To validate the new learning rule we throw a fuck ton of experiments together on the following list (or better using OpenAI Gym).

```
blockworld1 1.156 1.511 0.466 1.299 -0.080 1.260
blockworld3da 0.340 0.705 0.889 2.225 -0.139 0.658
canada 0.303 1.735 0.176 0.688 0.125 1.157
canada2d 0.400 0.978 -0.285 0.119 -0.045 0.701
cart 0.938 1.336 1.096 1.258 0.343 1.216
cartpole 0.844 1.115 0.482 1.138 0.244 0.755
cartpoleBalance 0.951 1.000 0.335 0.996 -0.468 0.528
cartpoleParallelDouble 0.549 0.900 0.188 0.323 0.197 0.572
cartpoleSerialDouble 0.272 0.719 0.195 0.642 0.143 0.701
cartpoleSerialTriple 0.736 0.946 0.412 0.427 0.583 0.942
cheetah 0.903 1.206 0.457 0.792 -0.008 0.425
fixedReacher 0.849 1.021 0.693 0.981 0.259 0.927
fixedReacherDouble 0.924 0.996 0.872 0.943 0.290 0.995
fixedReacherSingle 0.954 1.000 0.827 0.995 0.620 0.999
gripper 0.655 0.972 0.406 0.790 0.461 0.816
gripperRandom 0.618 0.937 0.082 0.791 0.557 0.808
hardCheetah 1.311 1.990 1.204 1.431 -0.031 1.411
hopper 0.676 0.936 0.112 0.924 0.078 0.917
hyq 0.416 0.722 0.234 0.672 0.198 0.618
movingGripper 0.474 0.936 0.480 0.644 0.416 0.805
pendulum 0.946 1.021 0.663 1.055 0.099 0.951
reacher 0.720 0.987 0.194 0.878 0.231 0.953
reacher3daFixedTarget 0.585 0.943 0.453 0.922 0.204 0.631
reacher3daRandomTarget 0.467 0.739 0.374 0.735 -0.046 0.158
reacherSingle 0.981 1.102 1.000 1.083 1.010 1.083
walker2d 0.705 1.573 0.944 1.476 0.393 1.397
```

2. Show that training decentralized policy gradient $\implies$ total policy optimization

3. Show speed improvements on update step through parallelism (samples per second vs DDPG).

4. Show results are comparable with the state of the art.

6 Conclusion

We wrecked deep reinforcement learning using biological inspiration.

6.1 Future Work

Would like to try the method with full recurrent networks and purely asynchronous implementation of leaky integration networks.

Would like to prove the conjecture. List possible methods of proof.