Reinforcement Learning

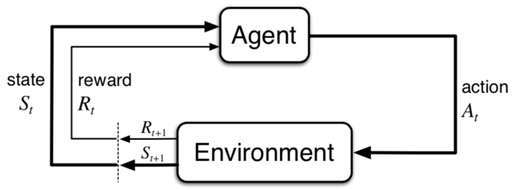
Traiko Dinev <traiko.dinev@gmail.com>

July 1, 2019

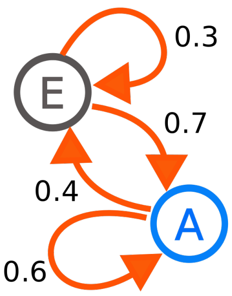
NOTE: This partially follows Reinforcement Learning, a masters level course at the University of Edinburgh. Credit also goes to `https://github.com/FrancescoSaverioZuppichini/Reinforcement-Learning-Cheat-Sheet` for some great ideas. The result graphs were taken from a courseswork exercise I did for Machine Learning Practical (MLP).

NOTE: Note this "summary" is NOT a reproduction of the course materials nor is it copied from the corresponding courses. It was entirely written and typeset from scratch.

1 Markov Decision Processes



(a) Agent-Environment Interaction (from Sutton & Barto)



(b) An example Markov Decision Process (from wikipedia)

The standard reinforcement learning problem is defined as follows (Sutton and Barto 1998); An agent is placed in an environment. At each timestep t the environment is in a state $S_t = s \in \mathcal{S}$; the agent takes an action $A_t = a \in \mathcal{A}$ and receives a real-valued reward $R_{t+1} = r \in \mathbb{R}$. S_t, A_t and R_t are random variables. After taking an action, the state changes probabilistically, as defined in the transition function for the environment $P(S_{t+1} = s' \mid S_t = s, A_t = a)$. Our goal is to learn a policy (in most cases deterministic) $\pi \colon \mathcal{S} \to \mathcal{A}$ that for each state maps to the action that the agent would take. We define the return as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{1}$$

where γ is the discount parameter. The agent's goal is to maximize the expected return $\mathbb{E}[G_t \mid S_0 = s_0]$. We define the state-action value function for a policy π as the expected return for a state and action under that policy:

$$q_\pi(S_t, A_t) \;=\; \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a\right] \tag{2}$$

We can define an optimal policy π^* as the policy that for each state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$ satisfies $q_{\pi^*}(s, a) = q_*(s, a) > q_\pi(s, a)$ for every other policy π. The Q-function for each optimal policy is the same. We define the Bellman optimal equation (Sutton and Barto 1998):

$$q_*(s_t, a_t) \;=\; \mathbb{E}_{\pi^*} \left[R_{t+1} + \gamma \ \max_{a'} q_*(s_{t+1}, a') \mid S_t = s_t, A_t = a_t\right] \tag{3}$$

We can also define a value function $V(s_t) = \mathbb{E}\left[G_t \mid S_t = s_t\right]$ and an optimal value function in a corresponding way as the optimal Q function. The value functions also satisfy a similar Bellman equation (Sutton and Barto 1998)

In the classical reinforcement learning scenario, the optimal policy can be learned by instead learning an optimal Q function (or value function) and then computing a greedy policy:

$$\pi^*(a \mid s) = \text{argmax}_a \ Q^*(s, a) \tag{4}$$

1.1 Q-learning and SARSA

Q-learning is one of the simplest algorithms (Sutton 1988). The Q values are stored in a table and continuous states are discretized into buckets. After each step, the Q value for the state-action pair is updated according to the following rule:

$$\begin{aligned} q(S_t,\ A_t) = (1\ -\ \alpha)\ q(S_t,\ A_t) + \\ \alpha\ (R_{t+1} + \gamma\ \max_a\ q(S_{t+1},\ A_{t+1})) \end{aligned} \tag{5}$$

Here α is the learning rate, starting at 1.0, which is exponentially decrease (anneal) by some factor (e.g. 0.995) after each episode. The max operator approximates the Bellman optimalitiy equation.

SARSA (A. Rummery and Niranjan 1994) is similar to Q-learning but the max operator is replaced by the action that would be taken by the current policy. The update rule is as follows:

$$\begin{aligned} q(S_t,\ A_t) = (1\ -\ \alpha)\ q(S_t, A_t) + \\ \alpha\ (R_{t+1} + \gamma q(S_{t+1}, A_{t+1})) \end{aligned} \tag{6}$$

where the learning rate α is similarly annealed. SARSA is an example of an on-policy method, where the Q updates are computed by sampling the next action from the current policy; thus S_{t+1}, A_{t+1} are computed before the update is made.

1.2 Deep Q Learning

Deep Q Networks (DQN) were introduced by DeepMind in their seminal paper (Mnih, Kavukcuoglu, Silver, Rusu, et al. 2015). Instead of using tables to store and update the Q values, neural networks are used. This eliminates the problem that tables grow exponentially large with the complexity of the state. At each step, the transitions $(s_t, a_t, r_{t+1}, s_{t+1})$ are stored in what is called the *experience replay.* The replay memory is also buffered with a first-in-first-out policy so as not to learn from earlier, less informative transitions. Periodically, the neural networks are trained via minibatch gradient descent. The inputs to the network are the states $\{s_i\}_{i=1}^{N}$ and the targets y^i are computed as follows:

$$y^i = \begin{cases} r_i & \text{for final } s_{i+1} \\ r_i + \gamma \ \max_a Q(s_{i+1}, a; \theta) & \text{otherwise} \end{cases} \tag{7}$$

where θ are the neural network parameters. Effectively we update the Q-value of the taken action to reflect the best possible value of Q under the current policy.

Because of the max operation in equation 6, the value for Q is often overestimated. Since our predictions for Q is inherently noisy, we pick the "most overestimated" value of Q. Hasselt, Guez, and Silver (2015a) proposed a simple solution. Instead of using one network, we now have two identical deep networks. The second network is updated to match the first every τ steps. We use the second network to compute the value updates instead of using the max operator, thus reducing the overestimation of Q values.

1.3 Double Deep Q Networks (DDQN)

:

$$y^i = \begin{cases} r_i & \text{for final } s_{i+1} \\ r_i + \gamma \ Q(s_{i+1}, \text{argmax}_{a'} Q(s_{i+1}, a'; \theta); \theta^-) & \text{otherwise} \end{cases} \tag{8}$$

Where θ^- are the target network's weights. In both cases the networks are trained not after every action, but each T steps taken, where the counter carries over episodes. The networks are not immediately trained, but we wait a number of iterations to let the agent gather some data by randomly exploring first.

For both algorithms, we usually suspnd training if an episode is solved, as defined in each environment. This way once we think we have reached an optimal policy π^*, we stop training until the policy fails. We do not want to "un-learn" π^* by continuing training afterwards.

1.4 ϵ - greedy

The ϵ-greedy exploration policy is what enables the agent to randomly *explore* rather than *exploit* the best found strategy so far. Instead of taking the action according to the greedy policy, as described above, with a small probability ϵ the agent takes a random action each step. ϵ is annealed similarly to α, the learning rate, in our case by a factor of 0.995 each episode.

See algorithm 1 for the DQN training algorithm.

Algorithm 1 DQN Training with ϵ-greedy

```
Initialize neural network with parameters $\theta$
counter ← 0
s ← initial environment state
repeat
    a ← argmax_a Q(s, a)
    a ← random action with probability $\epsilon$
    Perform action and observe tuple $(s', r, \text{done})$
    Store tuple $(s, a, s', r, \text{done})$ in replay memory
    if last environment was not solved then
        if counter ≥ start_training then
            if counter % train_frequency == 0 then
                Sample random minibatch of size $B$
                Train network according to equation 6
            end if
        end if
    end if
    s ← $s'$
    if done then
        s ← initial environment state
        Reset environment
    end if
    counter ← counter + 1
until maxEpisodes is reached
```

2 Policy Iteration

These are a family of dynamic programming algorithms. The general Policy Iteration algorithm first evaluates a policy (Policy Evaluation) after which it updates it, taking a maximum over the values calculated in the evaluation step.

Policy Evaluation. Repeat until the chanages in $V(s)$, $\Delta V(s)$ are still large.

$$\begin{aligned} &\forall s \in S \\ &V(s) = \sum_a \pi(a \mid s) \sum_{s',r} P(s', r \mid s, a)\big[r + \gamma V(s')\big] \end{aligned}$$

Policy Improvement. Repeat until the policy is unstable.

$$\begin{aligned} &\forall s \in S \\ &\quad \pi(s) = \arg\max_a \sum_{s',r} P(s', r \mid s, a)[r + \gamma V(s')] \end{aligned}$$

3 Value Iteration

This combines the above two in the same update cycle:

$$\begin{aligned} &\forall s \in S \\ &\quad V(s) = \underbrace{\max_a \sum_{s',r}}_{\text{Note max}} P(s', r \mid s, a)\big[r + \gamma V(s')\big] \end{aligned}$$

Note the *max* operator. This is used instead of summing over the policy $\pi(a \mid s)$. Thus we don't just evaluate the policy, but also improve it (by using the max operator). At the end we return a policy:

$$\pi(s) = \arg\max_a \sum_{s',r} P(s', r \mid s, a)[r + \gamma V(s)]$$

4 Monte Carlo

Monte Carlo methods average returns over random exploratory episodes.

```
Generate an episode at $s_0$ following$\pi$
$\forall s, a$
    $G \leftarrow$ return following first occurrence of $s, a$
    append G to Returns(s, a)
    $Q(s, a) \leftarrow$ average(Returns(s, a))
```

5 Eligibility Traces

TODO

6 Function Approximation

This is the general case of approximating Q-functions (or V-functions) with a function, such as a deep network above. The general update rule is just a gradient descent rule. Using an MSE error, we get:

$$\begin{aligned}\theta &\leftarrow \theta + \alpha(V^{\pi}(s_t) - V_t(s_t))\ \nabla_\theta V_t(s_t)\\ &= \theta + \alpha(V^{\pi}(s_t) - V_t(s_t))\ \nabla_\theta V_t(s_t)\end{aligned}$$

where:

$$V_t(s_t) = f(\theta)$$

and $V^{\pi}(s_t)$ is the state value as computed in the current episode-.

6.1 Policy Gradients Methods

For an excellent overview, see the tutorial by L. Weng (Weng 2018).

While SARSA tries to maximize the Q-function by using TD-errors, it still does not handle continuous actions or states very well. Indeed there are extensions of the method that better represent continuous states. Most notably, Deep Q-Networks (DQN, Mnih, Kavukcuoglu, Silver, Graves, et al. 2013) store the Q-function as a neural network. The inputs are the states $s \in \mathcal{S}$ and the outputs – the Q-value of the states for each action a. This representation, however, still relies on the environment having a discrete set of actions $\mathcal{A}$. Deep Q-Networks are similarly updated using a TD-error and the policy is chosen again using an arg max over actions.

For continuous action spaces, such as in the swing-up problem, Deep Q-Networks suffer from the same problem as SARSA – we need to discretize the action space, which leads to inaccuracies if there are too few buckets or high computational times if there are too many.

We can instead look at the value of the start state s_0 – $v_\pi(s_0)$ – and try to maximize it. The policy gradient theorem states that:

$$\nabla v_\pi(s) = \mathbb{E}_\pi\big[G_t \nabla_\theta \ln \pi(a \mid s; \theta)\big] \tag{9}$$

This assumes we have a parametric policy $\pi(a \mid s; \theta)$. We can now maximize $v_\pi(s_0)$ by performing a gradient **ascent** on the parameters of our policy θ.

We give a proof of the policy gradient theorem in the appendix. We combine the proof of Sutton 1988, Chapter 13 with the proof of Weng 2018 with some additional clarifications.

6.1.1 REINFORCE

Equation 9 can be approximated by a single sample of G_t over one episode. This is the idea behind REINFORCE:

$$\nabla v_\pi(s) = \mathbb{E}_\pi\big[G_t \nabla_\theta \ln \pi(a \mid s; \theta)\big] \approx G_t \nabla_\theta \ln \pi(a \mid s; \theta)$$

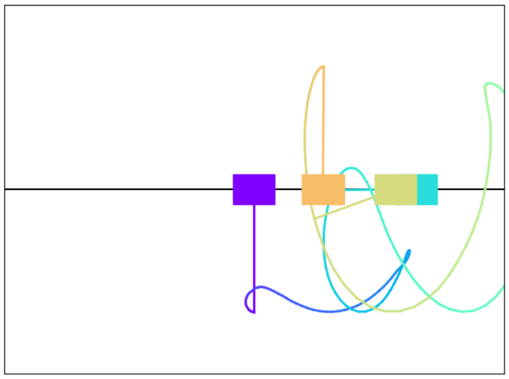


Figure 2: Trajectory using DDPG.

We can thus perform a simple gradient **ascent** that maximizes $v_\pi(s_0)$:

$$\theta \leftarrow \theta + \alpha G_t \nabla_\theta \ln \pi(A_t \mid S_t)$$

In order to apply this in practice, we average returns over the entire episode – $\frac{1}{N}\sum_i G_i$ and additionally normalize them. We also average the gradient of the policy function in the same way. This is akin to batch gradient descent, where averaging over many samples provides a more stable gradient.

The last key ingredient is the policy. Instead of using an ϵ-greedy exploration, we use a ”normal” policy:

$$\pi(a \mid s) = \mathcal{N}(f(s;\theta), \sigma)$$

where $f(s;\theta)$ is a parametric predictor of the mean of the normal distribution.

6.1.2 Deep Deterministic Policy Gradient (DDPG)

In REINFORCE, we used the episodic return G_t in the policy gradient. We could instead take the DQN approach (Mnih, Kavukcuoglu, Silver, Graves, et al. 2013) and use a **critic** neural network (with parameters θ^Q) to predict the $q_\pi(s,a)$ value. Additionally, we could use a deterministic policy, where $a = \pi(s;\theta^\pi)$. Again this is a neural network, called the **actor**. This slightly changes the policy gradient theorem, as proven in (Silver et al. 2014). Additionally, DDPG uses the double deep-Q-learning trick (Hasselt, Guez, and Silver 2015b), where the TD targets for the neural network use a second predictor network in order to reduce the variance of updates. Combining these steps, we arrive at Algorithm **??**. We next break down the steps of the algorithm.

We utilize an experience replay with stored transitions
(s_t, a_t, s_{t+1}, r_t). At each step of the algorithm, we train over a minibatch of transitions samples from the replay. **Firstly**, we update the Q-network using a one-step TD target, similar to SARSA's update rule:

$$y_i = r_i + q(s_{i+1}, \pi(s_{i+1};\theta^{\pi'});\theta^{Q'}) \tag{10}$$

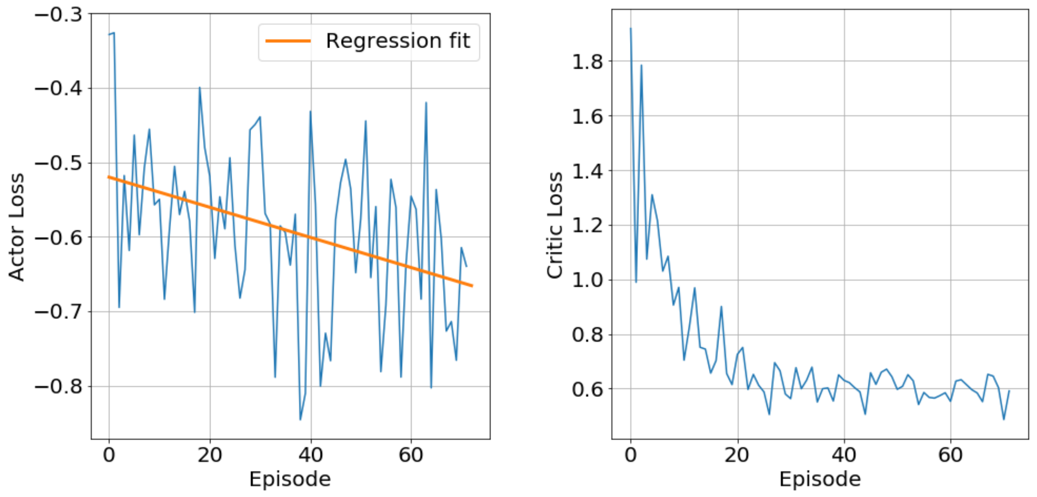


Figure 3: Actor and critic losses when using DDPG. The critic learns quickly, while the critic loss oscillates. We found that often the critic did not converge well and got stuck in local minima, such as immediately accelerating to the left or right. We show a fitted linear regression line for the actor loss.

We use the target Q-value network $\theta^{Q'}$ and a one-step on-policy TD error using the target policy network $\theta^{\pi'}$. We then update the neural network using mean-squared error to match the target in Equation 10. Thus the Q-network learns to predict the Q-values (returns) of the deterministic policy π. The policy is again a neural network, parameterized by $\theta^{\pi'}$.

Next, we update the actor network using the deterministic policy gradient theorem (Silver et al. 2014):

$$\nabla\theta^{\pi} = \frac{1}{N}\sum_i \nabla_a Q(s,a;\theta^Q)\big|_{s=s_i,a=\pi(s_i)} \nabla_{\theta^{\pi}}\pi(s;\theta^{\pi})\big|_{s_i}$$

This is the gradient of the Q-function:

$$\nabla\theta^{\pi} = \frac{1}{N}\sum_i \nabla_{\theta^{\pi}} Q(s_i,\pi(s_i;\theta^{\pi});\theta^Q)$$

Indeed this is an analog to the policy gradient theorem for deterministic policies, again using monte-carlo gradient estimation. The deterministic policy gradient theorem was introduces in Silver et al. 2014 and the proof is similar to that of the policy gradient theorem.

Finally, we update the target networks parameterized by $\theta^{Q'}$ and $\theta^{\pi'}$ using an exponential moving average. This ensures that our target network is more stable:

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$
$$\theta^{\pi'} \leftarrow \tau\theta^{\pi} + (1-\tau)\theta^{\pi'}$$

We applied DDPG to the inverted cart-pole system.

We used two neural networks with one layer with 100 neurons. Both the actor and the critic networks were trained using the Adam rule and the same σ and γ decay as previously. See Figure 2 for a trajectory learned by DDPG. See Figure 3 for an example plot of the loss functions for the actor and critic.

We found that DDPG was by far the most difficult to converge. While the critic learned to predict Q-values well (see an example in Figure 3), the actor oscillated a lot, sometimes getting stuck. Fitting a linear regression for the actor loss reveals that the overall trend is for the loss to decrease, albeit at a lower rate.

References

[1] Richard S. Sutton. “Learning to predict by the methods of temporal differences”. In: *Machine Learning* 3.1 (Aug. 1, 1988), pp. 9–44. ISSN: 0885-6125, 1573-0565. DOI: `10.1007/BF00115009`. URL: `https://link.springer.com/article/10.1007/BF00115009` (visited on 02/10/2018).

[2] G A. Rummery and Mahesan Niranjan. “On-Line Q-Learning Using Connectionist Systems”. In: *Technical Report CUED/F-INFENG/TR 166* (Nov. 4, 1994).

[3] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Google-Books-ID: CAFR6IBF4xYC. MIT Press, 1998. 356 pp. ISBN: 978-0-262-19398-6.

[4] Stanislaw H Zak. *Systems and control*. Vol. 198. 2003.

[5] Sarah Koskie. *Using the Lagrangian to obtain Equations of Motion*. 2008.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, et al. “Playing Atari with Deep Reinforcement Learning”. In: *arXiv:1312.5602 [cs]* (Dec. 19, 2013). arXiv: `1312.5602`. URL: `http://arxiv.org/abs/1312.5602` (visited on 09/22/2017).

[7] David Silver et al. “Deterministic Policy Gradient Algorithms”. In: *Proceedings of the 31st International Conference on Machine Learning*. Ed. by Eric P. Xing and Tony Jebara. Vol. 32. Proceedings of Machine Learning Research 1. Bejing, China: PMLR, 22–24 Jun 2014, pp. 387–395. URL: `http://proceedings.mlr.press/v32/silver14.html`.

[8] Hado van Hasselt, Arthur Guez, and David Silver. “Deep Reinforcement Learning with Double Q-learning”. In: *CoRR* abs/1509.06461 (2015). arXiv: `1509.06461`. URL: `http://arxiv.org/abs/1509.06461`.

[9] Hado van Hasselt, Arthur Guez, and David Silver. “Deep Reinforcement Learning with Double Q-learning”. In: (Sept. 22, 2015). arXiv: `1509.06461`. URL: `http://arxiv.org/abs/1509.06461` (visited on 02/13/2018).

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (Feb. 2015), p. 529. ISSN: 1476-4687. DOI: `10.1038/nature14236`. URL: `https://www.nature.com/articles/nature14236` (visited on 02/01/2018).

[11] Lilian Weng. *Policy Gradient Algorithms*. [Online; accessed ¡today¿]. 2018. URL: `https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html#dpg`.

7 Appendix

Here we provide short proofs of Lagrange's equations and the policy gradient theorem.

7.1 Lagrangian Dynamics

Here we give a short proof of the following theorem, known as Lagrange's equations:

$$\frac{\mathrm{d}}{\mathrm{dt}}\left(\frac{\partial L}{\partial \dot{\mathbf{q}}_i}\right) - \frac{\partial L}{\partial \mathbf{q}_i} = Q_i$$

The proof follows that of Zak 2003, Section 1.5 and Koskie 2008.

By Newton, work is defined as the force times the distance ($W = Fd$). In two (and similarly in three) dimensions, we define a generic parametric curve C as:

$$\begin{aligned}
&x = x(t), \quad y = y(t) \\
&\mathbf{q} = \langle x, y \rangle, \quad \mathbf{v} = \langle \dot{x}, \dot{y} \rangle \\
&0 < t < 1
\end{aligned}$$

If placed in a vector field $\mathbf{F} = \langle F_x, F_y \rangle$, the work exerted can be expressed as:

$$\lim_{\Delta\mathbf{q}\to 0} \sum_i \mathbf{F} \cdot \Delta\mathbf{q} = \int_C \mathbf{F} \cdot \mathrm{d}\mathbf{q} = \int_C F_x \mathrm{dx} + \int_C F_y \mathrm{dy} \tag{11}$$

From Newton's second law, we know that:

$$\mathbf{F} = m\mathbf{a} = m\frac{\mathrm{d}\mathbf{v}}{\mathrm{dt}}$$
$$F_x = m\ddot{x} = m\frac{\dot{x}}{\mathrm{dt}} \qquad F_y = m\ddot{y} = m\frac{\dot{y}}{\mathrm{dt}}$$

We also know that:

$$\ddot{x}dx = \frac{\mathrm{dx}}{\mathrm{dt}}dx = \dot{x}\mathrm{d\dot{x}} \tag{12}$$

Substituting in Equation 11, we obtain:

$$W_{AB} = \int_A^B m\dot{x}\mathrm{d\dot{x}} + \int_A^B m\dot{y}\mathrm{d\dot{y}} = m\left[\frac{\dot{x}^2}{2} + \frac{\dot{y}^2}{2}\right]\Bigg|_A^B = \frac{m}{2}(\hat{\mathbf{n}}\dot{\mathbf{q}}_B{}^2 - \hat{\mathbf{n}}\dot{\mathbf{q}}_A{}^2) \tag{13}$$

which is the definition of kinetic energy. Hence the work done is the difference in kinetic energies K_A and K_B at A and B. This is the work-energy theorem:

$$W_{AB} = K_B - K_A \tag{14}$$

By the law of conservation of energy, the sum of potential and kinetic energies must remain constant. Equivalently, if the kinetic energy changes, the potential energy changes in the opposite direction.

$$\begin{aligned}\Delta K + \Delta U &= 0 \\ W = \Delta K &= -\Delta U\end{aligned}$$

The fundamental theorem of vector calculus states that if $\mathbf{U}$ is a conservative (solenoid) field, i.e. $\mathbf{U} = \nabla f$, then:

$$\int_a^b \mathbf{U} \cdot \mathrm{d}\mathbf{q} = f(b) - f(a) \tag{15}$$

We know that:

$$\Delta U = -\int_A^B \mathbf{F} \cdot \mathrm{d}\mathbf{q}$$

We can thus conclude that:

$$\mathbf{F}(\mathbf{q}) = -\nabla U(\mathbf{q}) \tag{16}$$

We can define the kinetic energy in multiple dimensions as :

$$K = \frac{m\dot{\mathbf{q}}^{\mathrm{T}}\dot{\mathbf{q}}}{2} \quad \frac{\partial K}{\partial \dot{\mathbf{q}}_i} = m\dot{\mathbf{q}}_i \tag{17}$$

Combining Equation 17 with Equation 16, we obtain:

$$\mathbf{F}_i = m\mathbf{a}_i = \frac{\mathrm{d}}{\mathrm{dt}}\left(\frac{\partial K}{\partial \dot{\mathbf{q}}_i}\right) = -\frac{\partial U}{\partial \mathbf{q}_i} \tag{18}$$

Define the **Lagrangian** as $L = K - U$. Rearranging, we obtain:

$$\frac{\mathrm{d}}{\mathrm{dt}}\left(\frac{\partial L}{\partial \dot{\mathbf{q}}_i}\right) - \frac{\partial L}{\partial \mathbf{q}_i} = 0$$

where we note that:

$$\frac{\partial L}{\partial \dot{\mathbf{q}}_i} = \frac{\partial K}{\partial \dot{\mathbf{q}}_i}, \qquad \frac{\partial L}{\partial \mathbf{q}_i} = \frac{\partial U}{\partial \mathbf{q}_i}$$

An important case is when we add some energy to the system. As that adds overall energy, we can write:

$$\frac{\mathrm{d}}{\mathrm{dt}}\left(\frac{\partial L}{\partial \dot{\mathbf{q}}_i}\right) - \frac{\partial L}{\partial \mathbf{q}_i} = Q_i$$

where Q_i is the generalized force contribution in the direction of $\mathbf{q}_i$.

7.2 Policy Gradient Theorem

Here we prove the policy gradient theorem. This proof combines that of Sutton 1988 with Weng 2018 with some additional clarifications I felt were needed. We have a policy $\pi(a \mid s; \theta)$, which is parameterized by some vector θ. For instance, we could use a neural network or a linear regression. Then:

$$\nabla v_\pi(s) = \mathbb{E}_\pi\big[G_t \nabla_\theta \ln \pi(a \mid s; \theta)\big]$$

where $\mu(s)$ is the stationary distribution of the markov chain. To define this, first we define the transition probability as a matrix. Let:

$$M_\pi[s_t, s_{t-1}] = P(S_t = s_t \mid S_{t-1} = s_t, \pi)$$

The transition probability depends both on the transition function and the current policy π. Then assume we start at the state s_0. This makes the distribution of states a vector with 0 entries everywhere and 1 at s_0. The distribution over next states would be:

$$\begin{aligned}
p(S_0) &= [0, 0, \ldots, \underbrace{0}_{\mathrm{at} s_0}, \ldots, 0]^T \\
p(S_1) &= M p(S_0) \\
p(S_t) &= M^t p(S_0)
\end{aligned}$$

A stationary distribution is then the probability distribution over states as $t \to \infty$. With a little abuse of notation:

$$\mu(S_\infty) = \lim_{t\to\infty} M^t p(S_0) \tag{19}$$

Firstly, we need to prove the following **Lemma 1**:

$$\begin{aligned}
q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] &= \sum_{s', r} \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a, R_{t+1} = r, S_{t+1} = s'] P(s', r \mid s, a) \\
&= \sum_{s', r} P(s', r \mid s, a)\ \mathbb{E}_\pi\big[R_{t+1} + G_{t+1} \mid s, a, r, s'\big] = \sum_{s', r} P(s', r \mid s, a)\big[r + v_\pi(s')\big]
\end{aligned}$$

Then we expand the definition of the value function:

$$
\begin{aligned}
\nabla v_\pi(s) = \nabla \mathbb{E}_\pi[G_t \mid S_t = s] = \nabla \Bigg[\sum_a \mathbb{E}[G_t \mid S_t = s, A_t = s]\ \pi(a \mid s) \Bigg] &= \nabla \Bigg[\sum_a q_\pi(s,a)\ \pi(a \mid s) \Bigg] \\
&= \sum_a \Bigg[\nabla \pi(a \mid s) q_\pi(s,a) + \pi(a \mid s) \nabla q_\pi(s,a) \Bigg] \\
&= \sum_a \Bigg[\nabla \pi(a \mid s) q_\pi(s,a) + \pi(a \mid s) \nabla \sum_{s',r} P(s', r \mid s, a)(r + v_\pi(s')) \Bigg] \\
&= \sum_a \Bigg[\nabla \pi(a \mid s) q_\pi(s,a) + \pi(a \mid s) \sum_{s'} P(s' \mid s, a)\ \nabla v_\pi(s')) \Bigg] \\
&= \phi(s) + \sum_{s'} \sum_a \pi(a \mid s) P(s' \mid s, a) \nabla v_\pi(s') = \phi(s) + \sum_{s'} P(s \to s', 1) \nabla v_\pi(s')
\end{aligned}
$$

where $P(s \to s', 1)$ is the transition probability of getting to state s' from state s in one step. $\phi(s) = \sum_a \Bigg[\nabla \pi(a \mid s) q_\pi(s,a) \Bigg]$. We continue to unroll the recursion, eventually arriving at the stationary distribution:

$$
\begin{aligned}
\nabla v_\pi(s) &= \phi(s) + \sum_{s'} P(s \to s', 1) \nabla v_\pi(s') \\
&= \phi(s) + \sum_{s'} P(s \to s', 1) \Bigg[\phi(s') + \sum_{s''} \big[P(s' \to s'', 1) \nabla v_\pi(s'') \big] \Bigg] \\
&= \phi(s) + \sum_{s'} P(s \to s', 1) \phi(s') + \sum_{s''} P(s \to s'', 2) \nabla v_\pi(s'') \\
&= \dots \\
&= \sum_x \sum_{k=0}^{\infty} P(s \to x, k) \phi(x) \\
&= \sum_s \eta(x) \sum_a q_\pi(s,a) \nabla_\theta \pi(a \mid s) \\
&\propto \sum_s \frac{\eta(x)}{\sum_{\bar{x}} \eta(\bar{x})} \sum_a q_\pi(s,a) \nabla_\theta \pi(a \mid s) \\
&\propto \sum_s \mu(x) \sum_a q_\pi(s,a) \nabla_\theta \pi(a \mid s) \\
&\propto \sum_s \mu(x) \sum_a \pi(a \mid s) q_\pi(s,a) \frac{\nabla_\theta \pi(a \mid s)}{\pi(a \mid s)} \\
&= \mathbb{E}_\pi \big[q_\pi(s,a) \nabla_\theta \ln \pi(a \mid s; \theta) \big] \\
&= \mathbb{E}_\pi \big[G_t \nabla_\theta \ln \pi(a \mid s; \theta) \big]
\end{aligned}
$$

where on the last line we used the chain rule and the fact that $\nabla \ln x = \frac{1}{x} \nabla x$. The last step uses the following property of conditional expectations:

$$
\mathbb{E}\big[\mathbb{E}[X \mid Y]\big] = \sum_y \mathbb{E}[X \mid Y = y] P(Y = y) = \mathbb{E}[X]
$$