Reinforcement Learning

Trong

November 1, 2018—December 19, 2018

Contents

1 Cumulative reward

Cumulative reward is sometimes called return. The return from time t is the expected reward starting from time t:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + r_{t+4} + \ldots = \sum_{k=0}^{\infty} r_{t+k+1}.$$

We can also add the discount factor:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1},$$

where $0 < \gamma < 1$.

2 Temporal difference update

$$\begin{aligned} V(s) &= V(s) + \alpha(V(s') - V(s)) \\ &= (1-\alpha)V(s) + \alpha V(s'), \end{aligned}$$

essentially a weighted average between the old and the new values.

3 Law of Effect

Of several responses made to the same situation, those which are accompanied or closely followed by satisfaction to the animal will, other things being equal, be more firmly connected with the situation, so that, when it recurs, they will be more likely to recur; those which are accompanied or closely followed by discomfort to the animal will, other things being equal, have their connections with that situation weakened, so that, when it recurs, they will be less likely to occur. The greater the satisfaction or discomfort, the greater the strengthening or weakening of the bond.—Thorndike, 1911, p. 244.

4 Policy Function

Definition 1. *A policy π is a probability distribution over actions given states:*

$$\pi(a|s) = P[A_t = a | S_t = s].$$

I.e. Given the current state, what is the most likely action to take. A policy fully defines the behavior of an agent.

Given an MDP $M = (S, A, P, R, \gamma)$:

- The state sequence $S_1, S_2, \ldots$ is a Markov process S, P^π. IOW, the policy defines the particular Markov process, given the set of all possible states S and actions A.
- The state reward sequence $S_1, R_2, S_2, \ldots$ is a Markov reward process S, P^π, R^π, γ, where

$$\mathcal{P}^\pi_{ss'} = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{P}^a_{ss'}$$
$$\mathcal{R}^\pi_s = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{R}^a_s.$$

5 State value function and action value function

State value function: $V(s)$ is the expected return when starting from state s acting according to policy π :

$$V^\pi(s) = \mathbb{E}_\pi \left[G_t | s_t = s \right].$$

Action value function: $Q(s, a)$ is the value of an action in a given state and acting under a policy π :

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[G_t | s_t = s, a_t = a \right].$$

Note that Q is more specific than V, in the sense that V takes into account all possible actions.

6 Markov Decision Process

Definition 2. *A Markov Decision Process is a tuple* (S, A, P, R, γ)*, where* S *is a finite set of states,* A *is a finite set of actions,* P *is a state transition probability matrix,* R *is a reward function, and* γ *is a discount factor.*

Given a state $s \in S$ and an action $a \in A$, the probability of transitioning into a new state s' is given by $P(s, a)$, the reward of the transition is given by R_s^a. This assumes that the environment is responding deterministically, not stochastically, i.e. taking action a in state s will always bring you to a particular and the same s' every time.

7 Discrete Markov Chain

Definition 3. *Given a finite set of states* $\Omega = \{X_0, \ldots, X_N\}$*, a discrete Markov chain is a stochastic process that satisfies*

$$P\left(X_{n+1} = x_{n+1} | X_n = x_n, \ldots, X_0 = x_0\right) = P\left(X_{n+1} = x_{n+1} | X_n = x_n\right),$$

i.e. the present is enough to determine the future.

Question. Are we working with finitely many states? (It's always finite in the real world.)

Notation: the probability of moving from state i to state j at time n is written

$$p_{ij}(n) = P\left(X_{n+1} = j | X_n = i\right).$$

Definition 4. *Transition matrix:*

$$P(n) = \begin{pmatrix} p_{00}(n) & p_{01}(n) & p_{02}(n) & \dots & p_{0j}(n) & \dots \\ p_{10}(n) & p_{11}(n) & p_{12}(n) & \dots & p_{1j}(n) & \dots \\ p_{20}(n) & p_{21}(n) & p_{22}(n) & \dots & p_{2j}(n) & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ p_{i0}(n) & p_{i1}(n) & p_{i2}(n) & \dots & p_{ij}(n) & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}.$$

Properties:

- $P(n)$ is a square matrix.
- $\Sigma_j \, p_{ij}(n) = 1$ for all $j \in \Omega$. Row i gives the probabilities of moving from state i to all states, so they should all add up to 1.
- $p_{ij}(n) \geq 0$.

Note that in this example, the transition matrix is the same for all time steps $n : P = P(n)$.

8 State vector

We can describe the state of a Markov chain with a state vector $\pi^{(n)}$ where $\pi_j^{(n)}$ represents the probability of being in state j at time n.

E.g. Suppose the starting state is $\pi^{(0)} = (1, 0, 0, 0)$, then the next state vector is just the first row of the transition matrix, which is $(0.90, 0.07, 0.02, 0.01)$. In general,

$$\pi^{(n)} = \pi^{(0)} P^n.$$

9 Transient and steady state distribution

$$\frac{d\pi(t)}{dt} = \pi(t)Q$$

$$\pi(t) = \pi(0)e^{Qt} = \pi(0)\left(\mathbb{I} + \sum_{k=1}^{\infty} \frac{Q^k t^k}{k!}\right)$$

The steady state distribution (if it exists) can be found by solving

$$\pi Q = 0.$$

10 N-armed bandit

Simple case: only action affects reward. Compared with Contextual bandit where state and action both together affect reward. Full RL problem: action affects state and reward, and state affects reward.

E.g. k slot machines are an example of a k-armed bandit.

Question. What is the diff between having multiple bandits and a k-armed bandit?

11 ϵ-greedy algorithm

Pick the best action with probability $1-\epsilon$; OTW pick a random action with probability ϵ.

12 Bellman Equation

State-value function can be decomposed into immediate reward plus discounted value of successor state:

$$V_\pi(s) = \mathbb{E}_\pi \left[R_{t+1} + \gamma V_\pi \left(S_{t+1} \right) | S_t = s \right]$$

Similarly for the action-value function:

$$Q_\pi(s,a) = \mathbb{E}_\pi \left[R_{t+1} + \gamma Q_\pi \left(S_{t+1}, A_{t+1} \right) | S_t = s, A_t = a \right].$$

13 Notation

NOTE. The environment can be stochastic as well: taking action a at state s might not always get you to the same state s'. E.g.

$$\mathcal{P}^a_{ss'} = \Pr \left(s_{t+1} = s' | s_t = s, a_t = a \right)$$

is the probability of ending up in state s' by taking action a at state s.

Similarly,

$$\mathcal{R}^a_{ss'} = \mathbb{E}\left[r_{t+1} | s_t = s, s_{t+1} = s', a_t = a\right].$$

14 Deriving the Bellman Equation

$$\begin{aligned}
V^\pi(s) &= \mathbb{E}_\pi\left[R_t | s_t = s\right] \\
&= \mathbb{E}_\pi\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots | s_t = s\right] \\
&= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s\right] \\
&= \mathbb{E}_\pi\left[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \middle| s_t = s\right] \\
&= \mathbb{E}_\pi\left[r_{t+1} | s_t = s\right] + \mathbb{E}_\pi\left[\gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \middle| s_t = s\right] \quad (*)
\end{aligned}$$

The first term is

$$\mathbb{E}_\pi\left[r_{t+1} | s_t = s\right] = \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \mathcal{R}^a_{ss'}.$$

And the second term:

$$\mathbb{E}_\pi\left[\gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \middle| s_t = s\right] = \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \gamma \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \middle| s_{t+1} = s'\right].$$

Putting these back in $(*)$ and refactor, we get

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} \mathcal{P}^{a}_{ss'} \left(\mathcal{R}^{a}_{ss'} + \gamma \mathbb{E}_{\pi} \left[\sum_{k=0}^{\infty} \gamma^{k} r_{t+k+2} \middle| s_{t+1} = s' \right] \right)$$
$$= \sum_{a} \pi(s,a) \sum_{s'} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma V^{\pi}\left(s'\right) \right].$$

Similarly, the Bellman equation for the action value function is

$$Q^{\pi}(s,a) = \sum_{s'} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma \sum_{a'} \pi\left(s',a'\right) Q^{\pi}\left(s',a'\right) \right].$$

These equations give us an iterative way of calculating value functions: knowing the next state gives us the value of the current state. (So you can calculate backwards.)

15 Alternative forms for the Bellman equations for V and Q

$$v_{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}^{a}_{s} + \gamma \sum_{s' \in S} \mathcal{P}^{a}_{ss'} v_{\pi}\left(s'\right) \right)$$
$$q_{\pi}(s,a) = \mathcal{R}^{a}_{s} + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}^{a}_{ss'} \sum_{a' \in \mathcal{A}} \pi\left(a'|s'\right) q_{\pi}\left(s',a'\right).$$

Can write the Bellman equation as an induced Markov Reward Process:

$$v_{\pi} = \mathcal{R}^{\pi} + \gamma \mathcal{P}^{\pi} v_{\pi},$$

with direct solution

$$v_\pi = (I - \gamma \mathcal{P}^\pi)^{-1} \mathcal{R}^\pi.$$

16 Q-Learning Algorithm

The Q-table is initialized to a matrix where the columns correspond to actions and the rows to states. In this table, higher values are better. Given state s and action a, the new Q-value for s, a is

$$\begin{aligned} Q_{sa} &\leftarrow Q_{sa} + \alpha(r_{sa} + \gamma \max_{a'} Q_{s'a'} - Q_{sa}) \\ &= (1 - \alpha) Q_{sa} + \alpha(r_{sa} + \gamma \max_{a'} Q_{s'a'}) \\ &= (1 - \alpha) Q_{sa} + \alpha Q_{\text{target}}, \end{aligned}$$

where α is the learning rate, γ is the discount factor.

In other words, the new Q-value is an average between the old Q-value Q_{sa} and the reward plus the discounted expected reward in the new state. The value of the action is based on a combination of its previous value and (*) how much reward the action gave us, plus a little bit of future reward.

This latter value (*) can also be interpreted as the target Q-value:

$$Q_{\text{target}} = r_{sa} + \gamma \max_{a'} Q_{s'a'}.$$

A higher α means we're putting more stock on the present and future, and less on the previously learned values; a higher γ means we're putting more stock in the future.

Inefficient for large state spaces.

This looks like a special case of the temporal difference update rule we had before. It's also derived from the Bellman Equation. We should do that.

17 Deep Q-Learning: Q-Learning + Deep Neural Network

Replace the Q-Table / agent with a neural network.

18 Experience replay / replay buffer: separating experience from learning

This solves two things: (1) Reduces forgetting and (2) reduces correlating experience.

(1) because you're constantly learning from old experiences, not just the most recent ones.

(2). Correlating experiences means that if you learn things sequentially, consecutive experiences are highly correlated. This causes you to rely on your correlated bias to react to an experience, instead of treating each experience as independent and explore other more optimal actions.

The general strategy is to separate learning from interaction.

First you interact with the environment to gather experience, but you only start learning after you've filled the experience / replay buffer. "This helps avoid being fixated on one region of the state space. This prevents reinforcing the same action over and over." Variety is key!

(Learning V.S. Performing is often called Explore V.S. Exploit. This one is about separating Seeing/Experience from Believing/Learning LOL.)

19 Temporal difference update for DQN

$$\Delta w = \alpha \left[\left(R + \gamma \max_a Q\left(s', a, w\right) \right) - Q(s, a, w) \right] \nabla_w Q(s, a, w).$$

Recall how this is similar to the update rule for the Q-Table:

$$\begin{aligned} Q_{sa} &\leftarrow Q_{sa} + \alpha(Q_{\text{target}} - Q_{sa}) \\ &= (1 - \alpha) Q_{sa} + \alpha Q_{\text{target}}, \end{aligned}$$

which can be written as

$$\begin{aligned} Q_{sa}^{\text{new}} - Q_{sa}^{\text{old}} &= \alpha(Q_{\text{target}} - Q_{sa}^{\text{old}}) \\ \Delta Q_{sa} &= \alpha(Q_{\text{target}} - Q_{sa}). \end{aligned}$$

This multiply gradient $\nabla_w Q(s, a, w)$ doesn't make sense though. Shouldn't it be a quotient, like this?

$$\Delta w = \frac{\alpha \left[\left(R + \gamma \max_a Q\left(s', a, w\right) \right) - Q(s, a, w) \right]}{\nabla_w Q(s, a, w)},$$

because $\nabla_w Q(s,a,w)$ should look like $\frac{\Delta Q}{\Delta w}$.

20 Optimal state and action-value function

$v_*(s)$ is the maximum value function over all policies

$$v_*(s) = \max_\pi v_\pi(s).$$

Similarly for the action-value function

$$q_*(s,a) = \max_\pi q_\pi(s,a).$$

21 Partial ordering of policies

Define $\pi \geq \pi'$ if $v_\pi(s) \geq v_{\pi'}(s), \forall s$. Define similar ordering on action value functions?

Theorem 5 (An optimal policy exists)**.** *For any MDP,*

- *There exists an optimal policy π_* s.t. $\pi_* \geq \pi$ for all π.*
- *All optimal policies achieve the optimal value function: $v_{\pi_*}(s) = v_*(s)$.*
- *All optimal policies achieve the optimal action value function: $q_{\pi_*}(s,a) = q_*(s,a)$.*

22 Finding an optimal policy

Knowing the optimal action value function gives us a deterministic optimal policy:

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \operatorname{argmax}_{a \in A} q_*(s,a) \\ 0 & \text{otherwise.} \end{cases}$$

Well this is pretty obvious: the best policy is to take the action with the most value.

23 Bellman optimality equations

$$\begin{aligned} v_*(s) &= \max_a q_*(s,a) \\ q_*(s,a) &= \mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a v_*(s') \end{aligned}$$

Putting those together we get the Bellman optimality equation for v_* :

$$v_*(s) = \max_a \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \,.$$

Similarly for the optimal action:

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s',a') \,.$$

24 Negative log-likelihood and cross-entropy loss function

TODO. This is slightly wrong. The last formula below is for binary classification.

$$\text{Loss} = -A \log(\pi),$$

where A represents "advantage", a measure of how much better an action was compared to a base line. π is the policy, in this case corresponds to the chosen action's weight. Intuitively, a better action decreases loss and vice versa.

A more general version for binary classification:

$$C = -\frac{1}{n} \sum_x \left(\ln a_y^L \right)$$

Even more general is the cross-entropy loss for multi-class classification problems:

$$C = -\frac{1}{n} \sum_x (y \ln a + (1-y) \ln(1-a))$$

25 Fixed Q-targets

Clone the DQNetwork every once in a while and use it to generate targets, so that targets don't move while we're training. Will this really help?

26 Double DQN

Decouple action selection from target Q generation. One network for each task.

27 Dueling DQN (DDQN)

Recall that the Q-value $Q(s, a)$ represents the value of taking an action a at state s. We can decompose this into the value of being in state s, plus the value (also called advantage) of taking action a at s :

$$Q(s, a) = V(s) + A(s, a).$$

These 3 techniques break up / refactor a monolith network into smaller components. Good programming strategy in general. Nice to see how it's applied in Machine Learning.

28 Prioritized Experience Replay

Learn rare but important experiences, by prioritizing ones with a big difference between predicted and target Q:

$$p_t = |\delta_t| + e,$$

where δ_t is the magnitude of the temporal difference error, and e is a small number to ensure that all experiences have positive probability

of being chosen, i.e. none has zero probability of being chosen. The experience replay buffer then looks like:

$$\begin{bmatrix} S_t & A_t & R_{t+1} & S_{t+1} & p_t \\ S_{t+1} & A_{t+1} & R_{t+2} & S_{t+2} & p_{t+1} \\ & & \vdots & & \end{bmatrix}$$

OTOH, we also don't want to prioritize them all the time, which leads to overfitting. Therefore we introduce stochastic prioritization, where we define the probability of being chosen for replay:

$$P(t) = \frac{p_t^a}{\Sigma_k p_k^a},$$

where the sum is over all experiences in the replay buffer, and $a \in [0, 1]$ is a hyperparameter we choose to adjust how random we want the selection to be. E.g. $a = 0$ means that all experiences in the buffer are equally likely; $a = 1$ means we always choose the experience with the highest priority.

Even though there's some randomness in the prioritization, it is still biased towards the higher priority experiences. We can try and solve that with Importance Sampling.

29 Importance sampling

TODO. Need to read more about this.

30 Policy Gradients

Learning the policy (the best action to take in any given state) directly from the states, as opposed to learning the value of each state and each action to take in that state like we did in Q-Learning.

Question 6. *Can we combine both approaches: learn both the best states and the best actions to take in each state?*

30.1 Deterministic V.S. Stochastic Policies

Definition 7. *A stochastic policy takes in the state and outputs a distribution over actions to take:*

$$\pi_\theta(a|s) = \mathbb{P}[a = a_t|s = s_t].$$

We use a stochastic policy when the environment is uncertain, e.g. taking the same action won't necessarily bring you to the same state every time. Such a process is called a Partially Observable Markov Decision Process (POMDP).

30.2 Policy Gradient Objective Function

Finding the optimal policy is an optimization problem where we maximize the expected gain from each state:

$$\text{argmax}_\theta \, J(\theta)$$
$$J(\theta) = \mathbb{E}_{\pi_\theta}[V(S)] = \sum_{s \in S} \mathbb{P}[s] V(s)$$

where

$$\mathbb{P}(s) = \frac{N(s)}{\Sigma_{\sigma \in S} N(\sigma)}$$

is the probability of each state occurring and

$$V(s) = \sum_a \pi_\theta(a|s) q_\pi(s,a),$$

where $q_\pi(s,a)$ is the reward of taking action a in state s, and recall that $\pi_\theta(a|s)$ is the probability of taking a in s. IOW, the value of state s is just the sum of the values of taking all the actions a available at s weighted by their probability of occurring.

Putting it all together, we have the objective function

$$J(\theta) = \sum_{s \in S} \mathbb{P}[s] \sum_a \pi_\theta(a|s) q_\pi(s,a).$$

30.3 Gradient Ascent

Now that we have the objective function, we can perform gradient ascent on it, using the update rule

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta).$$

But first, we'll need to calculate $\nabla_\theta J(\theta)$.

31 Policy Gradient Theorem

Theorem 8. *The gradient of the objective function does not depend on the gradient of the state distribution $\mathbb{P}$, only on that*

of the action distribution π_θ:

$$\nabla J(\theta) \propto \sum_{s \in S} \mathbb{P}[s] \sum_a q_\pi(s,a) \nabla \pi_\theta(a|s). \qquad (*)$$

Note 9. Proportionality $\propto$ in the theorem is essentially equality, because any constant of proportionality will be absorbed by the learning rate α, which is arbitrary and decays as we train.

32 REINFORCE: Monte Carlo Policy Gradient

Note that the RHS of $(*)$ is a sum weighted by how often the states occur under policy π; therefore if π is used to sample actions, the states will occur with the same distribution π:

$$\nabla J(\theta) \propto \sum_{s \in S} \mathbb{P}[s] \sum_a q_\pi(s,a) \nabla \pi_\theta(a|s) = \mathbb{E}_\pi\Big[\sum_a q_\pi(s_t,a) \nabla \pi_\theta(a|s_t)\Big].$$

We can do the same thing for the actions:

$$\begin{aligned}
\mathbb{E}_\pi\Big[\sum_a q_\pi(s_t,a) \nabla \pi(a|s_t)\Big] &= \mathbb{E}_\pi\Big[\sum_a \pi(a|s_t) q_\pi(s_t,a) \frac{\nabla \pi(a|s_t)}{\pi(a|s_t)}\Big] \\
&= \mathbb{E}_\pi\Big[q_\pi(s_t,a_t) \frac{\nabla \pi(a_t|s_t)}{\pi(a_t|s_t)}\Big] \\
&= \mathbb{E}_\pi[q_\pi(s_t,a_t) \nabla \ln \pi(a_t|s_t)] \qquad (*) \\
&= \mathbb{E}_\pi[G_t \nabla \ln \pi(a_t|s_t)], \qquad (**)
\end{aligned}$$

where $G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$ is the return starting from time t. $(*)$ follows from the chain rule: $\nabla \ln f = \frac{\nabla f}{f}$; and $(**)$ follows from $\mathbb{E}_\pi[G_t|s_t, a_t] = q_\pi(s_t, a_t)$.

We can't calculate the real gradient directly, but in batches we can use $G_t \nabla \ln \pi(a_t|s_t)$ instead, whose expected value is equal to the gradient. Therefore the update rule becomes

$$\theta_{t+1} = \theta_t + \alpha G_t \nabla \ln \pi(a_t|s_t).$$

Question 10. *I wonder if using the random variable instead of the expected value works in other situations in Machine Learning. It should as long as we're sampling randomly, e.g. as in mini-batch updates.*

33 Intuition for the REINFORCE update rule

Consider the update rule

$$\theta_{t+1} = \theta_t + \alpha G_t \nabla \ln \pi(a_t|s_t) = \theta_t + \alpha G_t \frac{\nabla \pi(a_t|s_t)}{\pi(a_t|s_t)}.$$

This rule says that θ changes in the direction that most increases the probability of taking action a_t (according to $\nabla \pi(a_t|s_t)$) proportional to G_t (because we want actions that maximize return) and inversely proportional to $\pi(a_t|s_t)$ (because if an action occurs often, we want to dampen its effect so that rare events have a fighting chance).

34 Alan Turing

> The idea of artificial evolution was suggested by one of the founders of computer science, Alan Turing, in 1948. Turing wrote an essay while working on the construction of an electronic computer called the Automatic Computing Engine (ACE) at the National Physical Laboratory in the UK. His employer happened to be Sir Charles Darwin, the grandson of Charles Darwin, the author of 'On the Origin of Species'. Sir Charles dismissed the article as a "schoolboy essay"! It has since been recognized that in the article Turing not only proposed artificial neural networks but the field of artificial intelligence itself.—Julian F. Miller, *Cartesian Genetic Programming.*

35 Intuition for the Cross Entropy Loss Function

Definition 11. *Given two discrete distributions p and q, the Cross Entropy of p and q is defined as*

$$H(p,q) = \mathbf{E}_p[\log q] = -\sum_x p(x) \log q(x).$$

We'd like to know what this formula means.

Recall that the inner product is a measure of how close two vectors in $\mathbf{R}^n$ are; e.g. if v and w are unit vectors, then their dot product is -1 if they point in opposite directions (i.e. they are

very far apart), 0 if they are perpendicular (moderately far), and 1 if they point in the same direction (very close to each other). In addition, the inner product also encodes information about their relative lengths, according to

$$\langle v, w \rangle = |v||w| \cos \varphi,$$

where φ is the angle between v and w. In short, the inner product of two vectors tells us their relative "difference" with respect to length and angle, i.e. it tells us how they're situated relative to each other in n-dimensional space, in the same way that the difference between two real numbers tells us how they're situated relative to each other on the real line.

Now notice that

$$H(p, q) = -\sum_x p(x) \log q(x) = -\langle p, \log q \rangle.$$

In other words, $H(p, q)$ is a measure of how close the two distributions p and $\log q$ are: the closer they are (hence the closer p and q are), the larger $\langle p, \log q \rangle$ is, and the smaller $H(p, q) = -\langle p, \log q \rangle$ is.

Finally, in Machine Learning we often want to make predictions about the world, and these predictions will come out to follow a certain distribution p, and we want to know how well these predictions compare to ground truths, which obey another distribution q. In that case, we want to train the network to gradually push p towards q, i.e. we want to make p as close to q as possible, which means we want to minimize the Cross Entropy $H(p, q)$ e.g. using Gradient Descent.

36 Policy Gradient with Cross Entropy on Cartpole

Consider the following implementation of Cartpole using Policy Gradient with Cross Entropy:

```
class Agent:
  def __init__(self, sess, STATE_SIZE, ACTION_SIZE, name='Agent'):
    self.sess = sess
    with tf.variable_scope(name):
      self.inputs = inputs = tf.placeholder(
        tf.float32, [None, STATE_SIZE], name="inputs")
      self.actions = actions = tf.placeholder(
        tf.int32, [None, ACTION_SIZE], name="actions")
      self.discountedRewards = discountedRewards = tf.placeholder(
        tf.float32, [None, ], name="discountedRewards")

      fc1 = tf.contrib.layers.fully_connected(
        inputs=inputs,
        num_outputs=10,
        activation_fn=tf.nn.relu,
        weights_initializer=tf.contrib.layers.xavier_initializer())

      fc2 = tf.contrib.layers.fully_connected(
        inputs=fc1,
        num_outputs=10,
        activation_fn=tf.nn.relu,
        weights_initializer=tf.contrib.layers.xavier_initializer())

      fc3 = tf.contrib.layers.fully_connected(
        inputs=fc2,
        num_outputs=ACTION_SIZE,
        activation_fn=None,
        weights_initializer=tf.contrib.layers.xavier_initializer())

      self.actionDistr = tf.nn.softmax(fc3)

      negLogProb = tf.nn.softmax_cross_entropy_with_logits_v2(
        logits=fc3, labels=actions)
      self.loss = loss = tf.reduce_mean(negLogProb * discountedRewards)

      optimizer = tf.train.AdamOptimizer(LEARNING_RATE)
      self.minimize = optimizer.minimize(loss)
```

```
  @staticmethod
  def discountNormalizeRewards(EpisodeRewards):
    discountedRewards = np.zeros_like(EpisodeRewards)
    cumulative = 0.0
    for i in reversed(range(len(EpisodeRewards))):
      cumulative = cumulative * GAMMA + EpisodeRewards[i]
      discountedRewards[i] = cumulative

    mean = np.mean(discountedRewards)
    std = np.std(discountedRewards)
    discountedRewards = (discountedRewards - mean) / std

    return discountedRewards
```

To summarize, the network takes in state inputs, maps them to action predictions **fc3**, and compares them to "ground truth" actions using the Cross Entropy:

```
negLogProb = tf.nn.softmax_cross_entropy_with_logits_v2(logits=fc3, labels=actions)
```

Then it takes into account the discounted rewards to form the loss:

```
loss = tf.reduce_mean(negLogProb * discountedRewards)
```

We want to make sense of the loss function and how it causes the agent to learn via Gradient Descent.

36.1 Cartpole Background

In Cartpole, the agent is given one point reward for every extra step taken. To make things concrete, in a single episode, action predictions **fc3**, actions, raw episode rewards, discounted rewards, and the cross entropy **negLogProb** look like:

```
fc3 = [
  [ 0.02386748 -0.04044494]
  [ 0.00657506 -0.01413923]
```

```
  [ 0.01565559 -0.03301657]
  [ 0.0252676  -0.05284684]
  [ 0.0191054  -0.038988  ]
  [ 0.01111583 -0.02329292]
  [ 0.01875739 -0.0180726 ]
  [ 0.01143455 -0.02348156]
  [ 0.02049933 -0.04157549]
  [ 0.0294909  -0.06044051]
  [ 0.03897711 -0.08020715]
  [ 0.03263206 -0.06620266]
  [ 0.02537303 -0.05065311]
  [ 0.03361636 -0.06837433]
  [ 0.04239355 -0.08706529]
  [ 0.03523387 -0.07183101]
  [ 0.03210401 -0.05775316]
  [ 0.03719867 -0.07324304]
  [ 0.04349663 -0.0900949 ]
]

actions = [
  [1., 0.]
  [1., 0.]
  [1., 0.]
  [0., 1.]
  [0., 1.]
  [0., 1.]
  [1., 0.]
  [1., 0.]
  [1., 0.]
  [1., 0.]
  [0., 1.]
  [0., 1.]
  [1., 0.]
  [1., 0.]
  [0., 1.]
  [0., 1.]
  [1., 0.]
  [1., 0.]
  [1., 0.]
]

EpisodeRewards = [1.0 1.0 . . . 1.0]

discountedRewards = discountNormalizeRewards(EpisodeRewards) == [
  1.41340597  1.29896618  1.17850325  1.05170017  0.91822324
  0.77772121  0.62982433  0.47414341  0.31026875  0.13776911
  -0.04380945 -0.23494478 -0.43613987 -0.64792417 -0.87085502
```

```
  -1.10551906 -1.35253385 -1.61254941 -1.88625
]

negLogProb = [
  0.79828846 0.37360975 0.59066236 0.5557426  0.6059689  0.87303066
  0.45209357 0.9313073  0.44665492 0.46960846 0.7693604  0.99207836
  0.4362965  1.007341   1.0459526  1.0709308  1.0972168  1.125012
  0.37752238
]
```

36.2 How the Loss Function causes the agent to learn

Recall from before that the Cross Entropy

```
negLogProb = tf.nn.softmax_cross_entropy_with_logits_v2(logits=fc3, labels=actions)
```

measures the difference between the action predictions **`fc3`** and the "ground truth" actions. "Ground truth," because these are not the optimal actions we're feeding the agent; rather they're actions that caused the discounted rewards in a previous episode. By formulating the loss as

```
loss = tf.reduce_mean(negLogProb * discountedRewards)
```

we're telling the agent to produce actions that compromise between two tasks:

1. Come as close as possible to the ground truth actions, in order to receive the discounted rewards.
2. Increase the length of the episode.

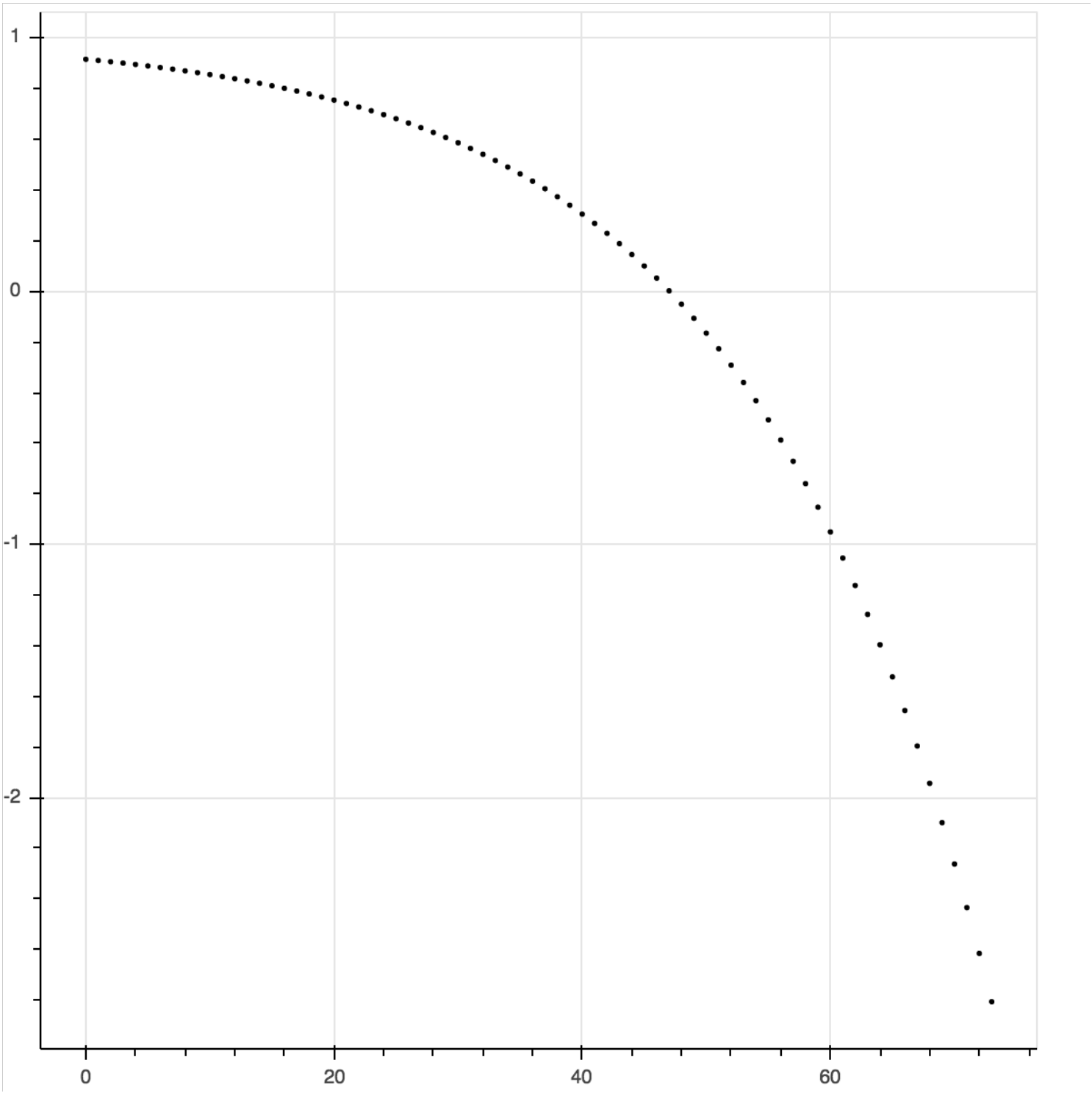
It's clear how (1) is achieved: the closer **`fc3`** is to **`actions`**, the smaller **`negLogProb`** is, and the smaller is the loss, roughly speaking.

It's less clear how the loss causes the length of the episode to increase. Increasing the length of the episodes is the thing we're trying to learn in Cartpole, i.e. allowing the agent to balance the pole longer.

To see how the loss function forces the network to increase the episode length, let's look at the discounted rewards:

```
discountedRewards = discountNormalizeRewards(EpisodeRewards) == [
  1.41340597  1.29896618  1.17850325  1.05170017  0.91822324  ]
  0.77772121  0.62982433  0.47414341  0.31026875  0.13776911  ] positive
  -0.04380945 -0.23494478 -0.43613987 -0.64792417 -0.87085502 }
  -1.10551906 -1.35253385 -1.61254941 -1.88625                } negative
]
```

Note that because of the way **`discountNormalizeRewards()`** operates on **`EpisodeRewards = [1.0 1.0 . . . 1.0]`**, by discounting each step with a **`GAMMA`** factor and normalization against the **`mean`** and **`std`**, **`discountedRewards`** has two parts: a positive head and a negative tail. In general, regardless of the length of the episode, **`discountedRewards`** will have the following shape:



Therefore, when we multiply **`discountedRewards`** with **`negLogProb`**, the positive head will force the corresponding head entries in **`negLogProb`** to become small, i.e. forcing the head of **`fc3`** to become more like the head of **`actions`**, while the negative tail

will force the corresponding tail entries in **negLogProb** to become large, i.e. forcing the tail of **fc3** to become *less* like the tail of **actions**.

36.3 Why train the network to be unlike the "Ground Truth"?

This latter behavior might at first sound like a bad idea: why would you want your network to behave differently than the "ground truth" actions?—until we remember that

> *the tail "ground truth" actions aren't that good to begin with: the tail end is when the cart fails to balance the pole and makes it fall over.*

Making the network behave differently than the "ground truth" is akin to trying new things. Sometimes the new behavior will cause a shorter episode, and sometimes a longer one. When it is longer, the tail end of **discountedRewards** will be deeper and more negative, thus causing the loss to decrease and the network to favor it.

To emphasize the point, this positive head and negative tail behavior is the reason why we normalize **EpisodeRewards** against the **mean** and **std** in

```
@staticmethod
def discountNormalizeRewards(EpisodeRewards):
```

```
  discountedRewards = np.zeros_like(EpisodeRewards)
  cumulative = 0.0
  for i in reversed(range(len(EpisodeRewards))):
    cumulative = cumulative * GAMMA + EpisodeRewards[i]
    discountedRewards[i] = cumulative

  mean = np.mean(discountedRewards)
  std = np.std(discountedRewards)
  discountedRewards = (discountedRewards - mean) / std

  return discountedRewards
```

so that we could have the positive head and negative tail. If we didn't, then **`discountedRewards`** would simply be a list of positive numbers, and when we multiply it with **`negLogProb`** in the **`loss`**, it would cause all entries of **`negLogProb`** to become small, causing **`fc3`** to become more like **`actions`** everywhere, including near the end of the episode when it should behave otherwise and be less like **`actions`**.

36.4 Summary

Finally, the loss function

```
negLogProb = tf.nn.softmax_cross_entropy_with_logits_v2(fc3, actions)
loss = tf.reduce_mean(negLogProb * discountedRewards)
```

or in shorthand:

$$L = -\frac{1}{n}\sum_i \langle A, \log s(\pi) \rangle \cdot G$$

$$G = \frac{[G_0 \ \cdots \ G_n] - \mu}{\sigma}$$

is good for games where the goal is to keep playing, where every step provides a reward.

Question 12. *Can we use it for other kinds of games?*

37 The big question

Question 13. *How to teach concepts to machines? What concepts should we teach first? What are the primitive concepts?*