Rainbow实验

1.深度Q网络（Deep Q-Networks,DQNs）

< 因为参考的资料中使用了英文，表达比较准确，故不另外翻译为中文 >

Reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value (also known as Q) function. This instability has several causes: the correlations present in the sequence of observations, the fact that small updates to Q may significantly change the policy and therefore change the data distribution, and the correlations between the action-values (Q) and the target values $r + \gamma \max_{a'} Q(s', a')$.

The authors suggest two key ideas to address these instabilities with a novel variant of Q-learning: Replay buffer and Fixed Q-target.

Uniformly random sampling from Experience Replay Memory

Reinforcement learning agent stores the experiences consecutively in the buffer, so adjacent (s, a, r, s') transitions stored are highly likely to have correlation. To remove this, the agent samples experiences uniformly at random from the pool of stored samples $\big((s, a, r, s') \sim U(D)\big)$. See sample_batch method of Replay Buffer class for more details.

Replay buffer

Typically, people implement replay buffers with one of the following three data structures:

- collections.deque
- list
- numpy.ndarray

deque is very easy to handle once you initialize its maximum length (e.g. deque(maxlen=buffer_size)). However, the indexing operation of deque gets terribly slow as it grows up because it is internally doubly linked list. On the other hands, **list** is an array, so it is relatively faster than deque when you sample batches at every step. Its amortized cost of **Get item** is O(1). Last but not least, let's see **numpy.ndarray**. numpy.ndarray is even faster than list due to the fact that it is a homogeneous array of fixed-size items, so you can get the benefits of locality of reference. Whereas list is an array of pointers to objects, even when all of them are of the same type.

Here, we are going to implement a replay buffer using **numpy.ndarray**.

```python
class ReplayBuffer:

    def __init__(self, obs_dim: int, size: int, batch_size: int = 32):
        self.obs_buf = np.zeros([size, obs_dim], dtype=np.float32)
        self.next_obs_buf = np.zeros([size, obs_dim], dtype=np.float32)
        self.acts_buf = np.zeros([size], dtype=np.float32)
        self.rews_buf = np.zeros([size], dtype=np.float32)
        self.done_buf = np.zeros(size, dtype=np.float32)
        self.max_size, self.batch_size = size, batch_size
        self.ptr, self.size, = 0, 0

    def store(
        self,
        obs: np.ndarray,
        act: np.ndarray,
        rew: float,
        next_obs: np.ndarray,
        done: bool,
    ):
        self.obs_buf[self.ptr] = obs
        self.next_obs_buf[self.ptr] = next_obs
        self.acts_buf[self.ptr] = act
        self.rews_buf[self.ptr] = rew
        self.done_buf[self.ptr] = done
        self.ptr = (self.ptr + 1) % self.max_size
        self.size = min(self.size + 1, self.max_size)

    # 随机取样
    def sample_batch(self) -> Dict[str, np.ndarray]:
        idxs = np.random.choice(self.size, size=self.batch_size,
replace=False)
        return dict(obs=self.obs_buf[idxs],
                    next_obs=self.next_obs_buf[idxs],
                    acts=self.acts_buf[idxs],
                    rews=self.rews_buf[idxs],
                    done=self.done_buf[idxs])

    def __len__(self) -> int:
        return self.size
```

经验回放

对比网络训练的速度，训练网络数据采集总是太慢。所以训练的瓶颈一般在智能体跟环境互动的过程中。如果我们能把互动过程中的数据，都存起来，当数据最够多的时候，再训练网络，那么就快很多了。

经验回放(Experience replay)就是实现这样的过程:

我们把每一步的 s，选择的 a，进入新的状态 s'，获得的奖励 r，新状态是否为终止状态，全部存在一个回放缓存池(replay buffer)。

- 当智能体与环境互动期间，就会不断产生这样一条一条数据。 数据1： 数据2： 数据3： ....
- 当数据量足够，例如达到我们设定一个batch的大小，我们便从中抽出一个batch大小的数据，把这笔数据一起放入网络进行训练。
- 训练之后我们继续进行游戏，继续把新产生的数据添加到回放缓存里。

就这样，我们每次都随机抽出一个batch大小的数据训练智能体。这样，以前产生的数据同样也能用来训练数据了, 效率自然更高。使用经验回放除了使训练更高效，同时也减少了训练产生的过度拟合，过度依赖局部经验会导致模型不够健壮。

Fixed Q-target

DQN uses an iterative update that adjusts the action-values (Q) towards target values that are only periodically updated, thereby reducing correlations with the target; if not, it is easily divergy because the target continuously moves. The Q-learning update at iteration i uses the following loss function:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)\right)^2\right]$$

in which γ is the discount factor determining the agent's horizon, θ_i are the parameters of the Q-network at iteration i and θ_i^- are the network parameters used to compute the target at iteration i. The target network parameters θ_i^- are only updated with the Q-network parameters (θ_i) every C steps and are held fixed between individual updates.

固定Q目标(Fixed Q-targets)

DQN的目标：$\gamma * maxQ(s') + r$

目标本身就包含一个Q网络，会造成Q网络的学习效率比较低，而且不稳定。

如果把训练神经网络比喻成射击游戏，在target中有Q网络的话，就相当于在射击一个移动靶，因为每次射击一次，靶就会挪动一次。相比起固定的靶，无疑加上了训练的难度。那怎么解决这个问题呢？既然现在是移动靶，那么我们就把它弄成是固定的靶，先停止10秒。10后挪动靶再打新的靶。这就是Fixed Q-targets的思路。

Epsilon-greedy

Qlearning使用了noisy-greedy的方式，保持探索和开发之间的平衡。智能体会根据Q值表选择Q值最大的动作。但在选择动作之前，会先给Q值加上一个随机的噪音，使得最终的最大值带有一定的随机性；随着游戏进行，噪音会逐渐减少，最终噪音将会小得不再影响智能体的决定。

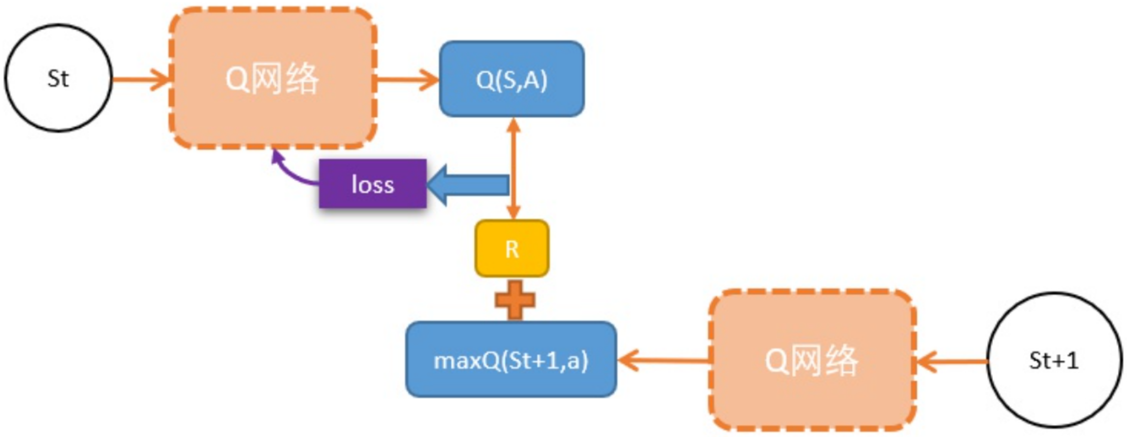
在DQN，同样可以使用增加噪音的方式，不过更常用的Epsilon-greedy，为了保证大部分的state都能被探索到的基础上，最终也能够按照智能体学习到的方式取进行。Agent 在选择动作的时候，会先随机一个[0,1]之间的值， 如果随机出来的值小于epsilon，就随机探索，否则就用最大Q值的动作。随着迭代次数增加，智能体逐渐从**探索**变为**开发**。

For more stability: Gradient clipping

The authors also found it helpful to clip the error term from the update $r + \gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a,;\theta_i)$ to be between -1 and 1. Because the absolute value loss function $|x|$ has a derivative of -1 for all negative values of x and a derivative of 1 for all positive values of x, clipping the squared error to be between -1 and 1 corresponds to using an absolute value loss function for errors outside of the (-1,1) interval. This form of error clipping further improved the stability of the algorithm.

总结

DQN其实就是Q-learning的一种实现框架，在最早的Q-learning中，是通过Q-table来找到Q (s , a) ，这是查找是一种映射关系。但是当状态很多或者动作连续时Q-table变得很难处理，既然是映射关系就可以使用神经网络。



算法流程：

1. 初始化一个网络，用于计算Q值。
2. 开始一场游戏，随机从一个状态s开始。
3. 我们把s输入到Q，计算s状态下，a的Q值 Q（s）。
4. 我们选择能得到最大Q值的动作a。
5. 我们把a输入到环境，获得新状态s',r,done。
6. 计算目标 y = r + gamma * maxQ(s') 。
7. 训练Q网络，缩小Q（s，a）和 y 的差距。
8. 开始新一步，不断更新。

使用梯度下降法更新的loss函数为：

$$\left(R_{t+1}+\gamma_{t+1} \max_{a'} q_{\bar{\theta}}\left(S_{t+1}, a'\right)-q_{\theta}\left(S_{t}, A_{t}\right)\right)^{2}$$

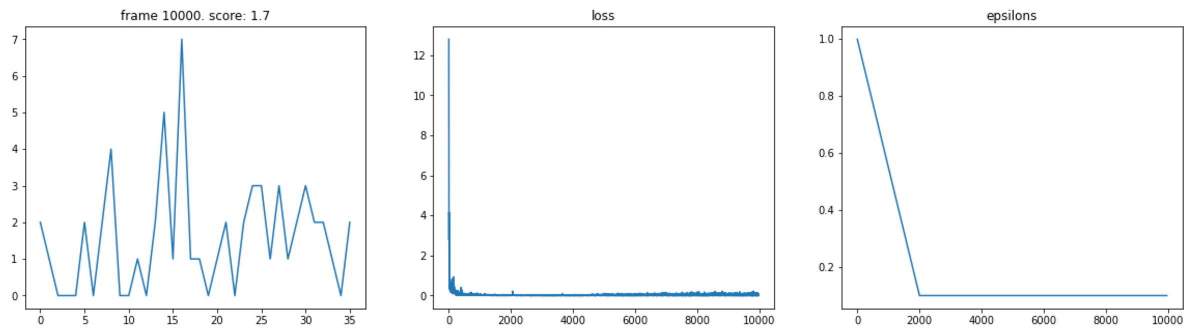
t是从经验回放池中随机选取的时间步长。

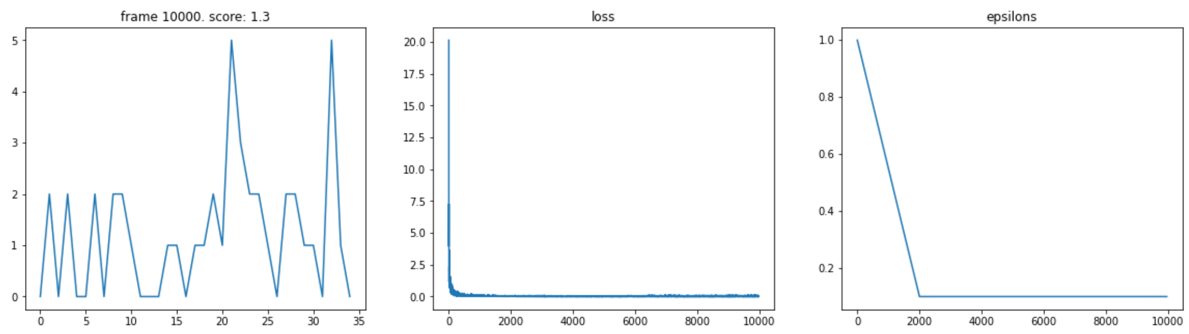
结果

采用的环境是Atari游戏中的 Breakout-ram-v0，受硬件设备的限制，程序的训练次数少，只能展示训练初期的效果，总体效果不太好。

游戏开始时，画面今显示8排砖块，每隔两排，砖块的颜色就不同。由下至上的颜色排序为黄色、绿色、橙色和红色。游戏开始后，玩家必须控制一块平台左右移动以反弹一个球。当那个球碰到砖块时，砖块就会消失，而球就会反弹。当玩家未能用平台反弹球的话，那么玩家就输掉了那个回合。当玩家连续输掉3次后，玩家就会输掉整个游戏。玩家在游戏中的目的就是清除所有砖块。玩家破坏黄色砖块能得1分、绿色能得3分、橙色能得5分、而红色则能得7分。当球碰到画面顶部时，玩家所控制的平台长度就会减半。另外，球的移动速度会在接触砖块4次、接触砖块12次、接触橙色砖块和接触红色砖块后加速。玩家在此游戏中最高能获896分，即完成两个448分的关卡。当玩家获得896分后，游戏不会自动结束，只会待玩家输掉3次后才会结束。

训练结果如下：





2.双深度Q网络（Double DQN）

van Hasselt et al., "Deep Reinforcement Learning with Double Q-learning." arXiv preprint arXiv:1509.06461, 2015.

Let's take a close look at the difference between DQN and Double-DQN. The max operator in standard Q-learning and DQN uses the same values both to select and to evaluate an action. This makes it more likely to select overestimated values, resulting in overoptimistic value estimates.

DQN存在高估问题，因为更新公式中总是选取最大的Q值，Double DQN（DDQN；van Hasselt、Guez&Silver；2016）通过解耦选择和引导行动评估解决了 Q 学习过度估计偏差的问题。

$$\theta_{t+1} = \theta_t + \alpha\big(Y_t^Q - Q(S_t, A_t; \theta_t)\big) \cdot \nabla_{\theta_t} Q(S_t, A_t; \theta_t),$$
$$\text{where } \alpha \text{ is a scalar step size and the target } Y_t^Q \text{is defined as}$$
$$Y_t^Q = R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_t).$$

In Double Q-learning (van Hasselt 2010), two value functions are learned by assigning experiences randomly to update one of the two value functions, resulting in two sets of weights, θ and θ'. For each update, one set of weights is used to determine the greedy policy and the other to determine its value. For a clear comparison, we can untangle the selection and evaluation in Q-learning and rewrite DQN's target as

$$Y_t^Q = R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \theta_t); \theta_t).$$

The Double Q-learning error can then be written as

$$Y_t^{DoubleQ} = R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \theta_t); \theta_t').$$

这个方法改变的是神经网络的结构，把Q值函数分解为价值函数V和优势函数A的和，即Q= V+A。其中V代表了这种状态的好坏程度，有时函数表明在这个状态下某一个动作相对于其他动作的好坏程度，而Q表明了这个状态下确定的某个动作的价值。为了限制A和V的值使得其在一个合理的范围内，我们可以用一种方法将A的均值限制为0以达到我们的要求。

The idea of Double Q-learning is to reduce overestimations by decomposing the max operation in the target into action selection and action evaluation. Although not fully decoupled, the target network in the DQN architecture provides a natural candidate for the second value function, without having to introduce additional networks. In conclusion, the weights of the second network θ_t' are replaced with the weights of the target network for the evaluation of the current greedy policy. This makes just a small change in calculating the target value of DQN loss.

DQN:

```
target = reward + gamma * dqn_target(next_state).max(dim=1, keepdim=True)[0]
```

DoubleDQN:

```
selected_action = dqn(next_state).argmax(dim=1, keepdim=True)

target = reward + gamma * dqn_target(next_state).gather(1, selected_action)
```

DoubleDQN 使用loss函数:

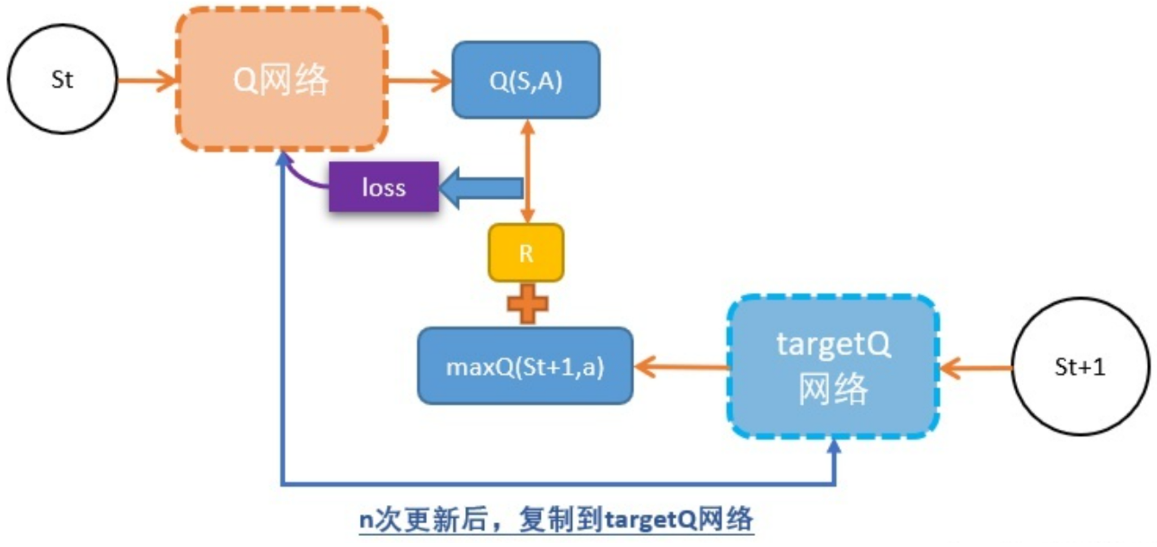
$$\left(R_{t+1}+\gamma_{t+1} q_{\bar{\theta}}\left(S_{t+1}, \underset{a^{\prime}}{\operatorname{argmax}} q_{\theta}\left(S_{t+1}, a^{\prime}\right)\right)-q_{\theta}\left(S_{t}, A_{t}\right)\right)^{2}$$

总结

DQN有一个显著的问题，就是DQN估计的Q值往往会偏大。这是由于我们Q值是以下一个s'的Q值的最大值来估算的，但下一个state的Q值也是一个估算值，也依赖它的下一个state的Q值...，这就导致了Q值往往会有偏大的的情况出现。我们在同一个s'进行试探性出发，计算某个动作的Q值。然后和DQN的记过进行比较就可以得出上述结论。

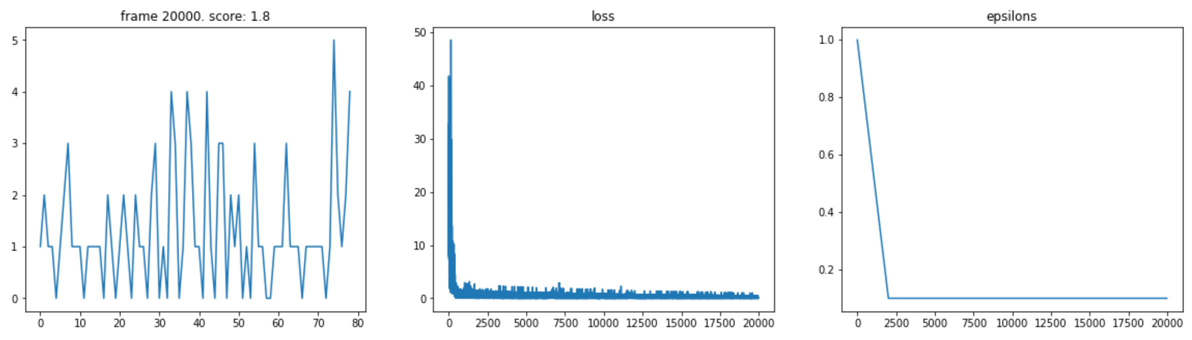
解决方法可以是

- 用两个Q网络，因为两个Q网络的参数有差别，所以对于同一个动作的评估也会有不同。我们选取评估出来较小的值来计算目标。
- Q1网络推荐能够获得最大Q值的动作；Q2网络计算这个动作在Q2网络中的Q值。



一个是原来的Q网络，用于估算Q(s)；另外一个是 targetQ 网络， targetQ网络自己并不会更新，也就是它在更新的过程中是固定的，用于计算更新目标 y = r + gamma * max(targetQ(s'))。我们进行N次更新后，就把新Q的参数赋值给旧Q。

结果



3.经验优先回放（Prioritized Experience Replay ）

T. Schaul et al., "Prioritized Experience Replay." arXiv preprint arXiv:1511.05952, 2015.

Using a replay memory leads to design choices at two levels:

- which experiences to store
- which experiences to replay (and how to do so).

This paper addresses only the latter: making the most effective use of the replay memory for learning, assuming that its contents are outside of our control.

在传统DQN中，经验池中的数据采样都是随机的，但其实不同样本的价值是不同的，需要给样本一个优先级，并根据样本的优先级进行采样。

The central component of prioritized replay is the criterion by which **the importance of each transition** is measured. A reasonable approach is to use the magnitude of a transition's TD error δ, which indicates how 'surprising' or unexpected the transition is. This algorithm stores the last encountered TD error along with each transition in the replay memory. The transition with the largest absolute TD error is replayed from the memory. A Q-learning update is applied to this transition, which updates the weights in proportion to the TD error. One thing to note that new transitions arrive without a known TD-error, so it puts them at maximal priority in order to guarantee that all experience is seen at least once. (see **store** method)

We might use 2 ideas to deal with TD-error:

1. greedy TD-error prioritization
2. stochastic prioritization.

However, greedy TD-error prioritization has a severe drawback. Greedy prioritization focuses on a small subset of the experience: errors shrink slowly, especially when using function approximation, meaning that the initially high error transitions get replayed frequently. This lack of diversity that makes the system prone to over-fitting.

作者首先给出了一个衡量指标：“TD-error”，随后贪心地选”信息量“最大的transition，然而这种方法存在以下缺陷：

- 由于考虑到算法效率，不会每次critic更新后都更新所有transition的TD-error，我们只会更新当次取到的transition的TD-error。因此transition的TD-error对应的critic是是上次取到该transition时的critic，而不是当前的critic。也就是说某一个transition的TD-error较低，只能够说明它对之前的critic“信息量”不大，而不能说明它对当前的critic“信息量”不大，因此根据TD-error进行贪心有可能会错过对当前critic“信息量”大的transition。
- 容易overfitting：因为从原理上来说，被选中的transition的TD-error在critic更新后会下降，然后排到后面去，下一次就不会选中这些transition，导致overfitting。

To overcome this issue, we will use a **stochastic sampling method** that interpolates between pure greedy prioritization and uniform random sampling.

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

where $p_i > 0$ is the priority of transition i. The exponent α determines how much prioritization is used, with $\alpha = 0$ corresponding to the uniform case. In practice, we use additional term ϵ in order to guarantee all transactions can be possibly sampled: $p_i = |\delta_i| + \epsilon$, where ϵ is a small positive constant.

greedy TD-error prioritization通过更加频繁的更新去衡量“TD-error”，然而也会带来的多样性损失问题，则利用随机优先级采样、偏置和重要性采样来避免该问题，在随机化的采样过程，“信息量”越大，被抽中的概率越大，但即使是“信息量”最大的transition，也不一定会被抽中，仅仅只是被抽中的概率较大。

One more. Let's recall one of the main ideas of DQN. To remove correlation of observations, it uses uniformly random sampling from the replay buffer. Prioritized replay introduces bias because it doesn't sample experiences uniformly at random due to the sampling proportion correspoding to TD-error. We can correct this bias by using importance-sampling (IS) weights

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta$$

that fully compensates for the non-uniform probabilities $P(i)$ if $\beta = 1$. These weights can be folded into the Q-learning update by using $w_i\delta_i$ instead of δ_i. In typical reinforcement learning scenarios, the unbiased nature of the updates is most important near convergence at the end of training, We therefore exploit the flexibility of annealing the amount of importance-sampling correction over time, by defining a schedule on the exponent β that reaches 1 only at the end of learning.

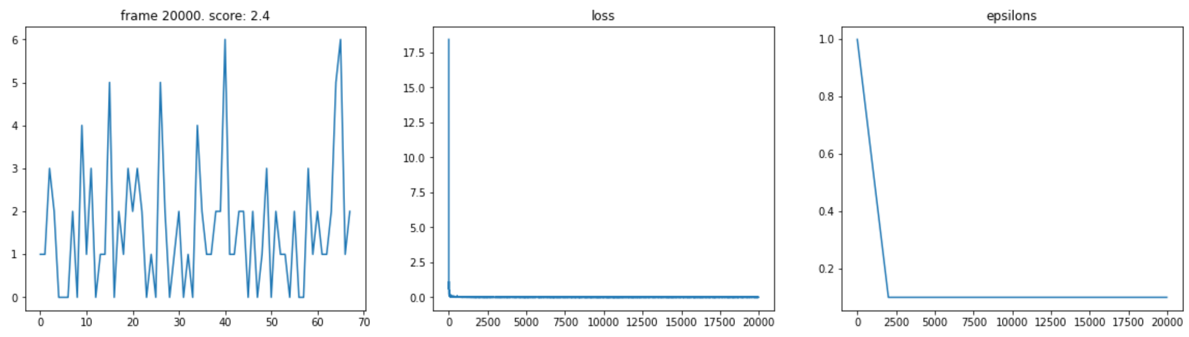
那么如何有效进行抽样？如果我们每次抽样都对p进行排序选取其中的最大值，会浪费大量的计算资源，从而使得训练时间变长。常用的解决方法是使用树形结构来储存数据。这种树状结构本质上也是采用了概率式的选择方式，优先级p的大小跟选中的概率成正比，和线性的结构相比，这种方法不用逐一检查数据，大大提高了运行效率。为了方便优先回放存储与及采样，我们采用Segment Tree树来存储。

Segment Tree

The key concept of PER's implementation is **Segment Tree**. It efficiently stores and samples transitions while managing the priorities of them.

```python
class PrioritizedReplayBuffer(ReplayBuffer):
    """Prioritized Replay buffer.

    Attributes:
        max_priority (float): max priority
        tree_ptr (int): next index of tree
        alpha (float): alpha parameter for prioritized replay buffer
        sum_tree (SumSegmentTree): sum tree for prior
        min_tree (MinSegmentTree): min tree for min prior to get max weight

    """

    def __init__(
        self,
        obs_dim: int,
        size: int,
        batch_size: int = 32,
        alpha: float = 0.6
    ):
        """Initialization."""
        assert alpha >= 0

        super(PrioritizedReplayBuffer, self).__init__(obs_dim, size, batch_size)
        self.max_priority, self.tree_ptr = 1.0, 0
        self.alpha = alpha

        # capacity must be positive and a power of 2.
        tree_capacity = 1
        while tree_capacity < self.max_size:
            tree_capacity *= 2

        self.sum_tree = SumSegmentTree(tree_capacity)
        self.min_tree = MinSegmentTree(tree_capacity)

    def store(
        self,
        obs: np.ndarray,
        act: int,
        rew: float,
        next_obs: np.ndarray,
        done: bool
    ):
        """Store experience and priority."""
        super().store(obs, act, rew, next_obs, done)

        self.sum_tree[self.tree_ptr] = self.max_priority ** self.alpha
        self.min_tree[self.tree_ptr] = self.max_priority ** self.alpha
        self.tree_ptr = (self.tree_ptr + 1) % self.max_size

    def sample_batch(self, beta: float = 0.4) -> Dict[str, np.ndarray]:
        """Sample a batch of experiences."""
        assert len(self) >= self.batch_size
        assert beta > 0

        indices = self._sample_proportional()

        obs = self.obs_buf[indices]
        next_obs = self.next_obs_buf[indices]
        acts = self.acts_buf[indices]
        rews = self.rews_buf[indices]
        done = self.done_buf[indices]
        weights = np.array([self._calculate_weight(i, beta) for i in indices])

        return dict(
            obs=obs,
            next_obs=next_obs,
            acts=acts,
            rews=rews,
            done=done,
            weights=weights,
            indices=indices,
        )

    def update_priorities(self, indices: List[int], priorities: np.ndarray):
        """Update priorities of sampled transitions."""
        assert len(indices) == len(priorities)

        for idx, priority in zip(indices, priorities):
            assert priority > 0
            assert 0 <= idx < len(self)

            self.sum_tree[idx] = priority ** self.alpha
            self.min_tree[idx] = priority ** self.alpha

            self.max_priority = max(self.max_priority, priority)

    def _sample_proportional(self) -> List[int]:
        """Sample indices based on proportions."""
        indices = []
        p_total = self.sum_tree.sum(0, len(self) - 1)
        segment = p_total / self.batch_size

        for i in range(self.batch_size):
            a = segment * i
            b = segment * (i + 1)
            upperbound = random.uniform(a, b)
            idx = self.sum_tree.retrieve(upperbound)
            indices.append(idx)

        return indices

    def _calculate_weight(self, idx: int, beta: float):
        """Calculate the weight of the experience at idx."""
        # get max weight
        p_min = self.min_tree.min() / self.sum_tree.sum()
        max_weight = (p_min * len(self)) ** (-beta)

        # calculate weights
        p_sample = self.sum_tree[idx] / self.sum_tree.sum()
        weight = (p_sample * len(self)) ** (-beta)
        weight = weight / max_weight

        return weight
```

结果



4.竞争双深度 （Dueling Network）

[Z. Wang et al., "Dueling Network Architectures for Deep Reinforcement Learning." arXiv preprint arXiv:1511.06581, 2015.]

Dueling 网络架构（Wang 等人；2016）可以通过分别表示状态值和动作奖励来概括各种动作。从多步骤引导程序目标中学习（Sutton；1988；Sutton & Barto 1998）如 A3C（Mnih 等人；2016）中使用偏差-方差权衡而帮助将最新观察到的奖励快速传播到旧状态中。

The proposed network architecture, which is named **dueling architecture**, explicitly separates the representation of state values and (state-dependent) action advantages.

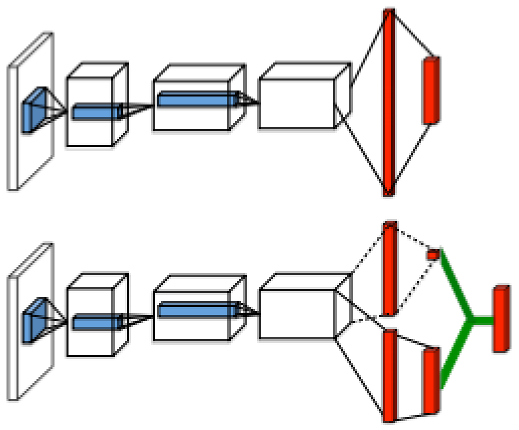


Figure 1. A popular single stream Q-network (**top**) and the dueling Q-network (**bottom**). The dueling network has two streams to separately estimate (scalar) state-value and the advantages for each action; the green output module implements equation (9) to combine them. Both networks output Q-values for each action.

The dueling network automatically produces separate estimates of the state value function and advantage function, without any extra supervision. Intuitively, the dueling architecture can learn which states are (or are not) valuable, without having to learn the effect of each action for each state. This is particularly useful in states where its actions do not affect the environment in any relevant way.

一般DQN在提升某个状态下的S值时，只会提升某个动作，Dueling DQN在网络更新的时候，由于有**A值之和必须为0**的限制，所以网络会**优先更新S值**。S值是Q值的平均数，平均数的调整相当于一次性S下的所有Q值都更新一遍。所以网络在更新的时候，不但更新某个动作的Q值，而是把这个状态下，所有动作的Q值都调整一次。这样，我们就可以在更少的次数让更多的值进行更新。

The dueling architecture represents both the value $V(s)$ and advantage $A(s,a)$ functions with a single deep model whose output combines the two to produce a state-action value $Q(s,a)$. Unlike in advantage updating, the representation and algorithm are decoupled by construction.

$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s).$

The value function V measures the how good it is to be in a particular state s. The Q function, however, measures the the value of choosing a particular action when in this state. Now, using the definition of advantage, we might be tempted to construct the aggregating module as follows:

$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha),$

where θ denotes the parameters of the convolutional layers, while α and β are the parameters of the two streams of fully-connected layers.

Unfortunately, the above equation is unidentifiable in the sense that given Q we cannot recover V and A uniquely; for example, there are uncountable pairs of V and A that make Q values to zero. To address this issue of identifiability, we can force the advantage function estimator to have zero advantage at the chosen action. That is, we let the last module of the network implement the forward mapping.

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \big(A(s,a;\theta,\alpha) - \max_{a'\in|\mathcal{A}|} A(s,a';\theta,\alpha)\big).$$

This formula guarantees that we can recover the unique V and A, but the optimization is not so stable because the advantages have to compensate any change to the optimal action's advantage. Due to the reason, an alternative module that replaces the max operator with an average is proposed:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \big(A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\theta,\alpha)\big).$$

Unlike the max advantage form, in this formula, the advantages only need to change as fast as the mean, so it increases the stability of optimization.

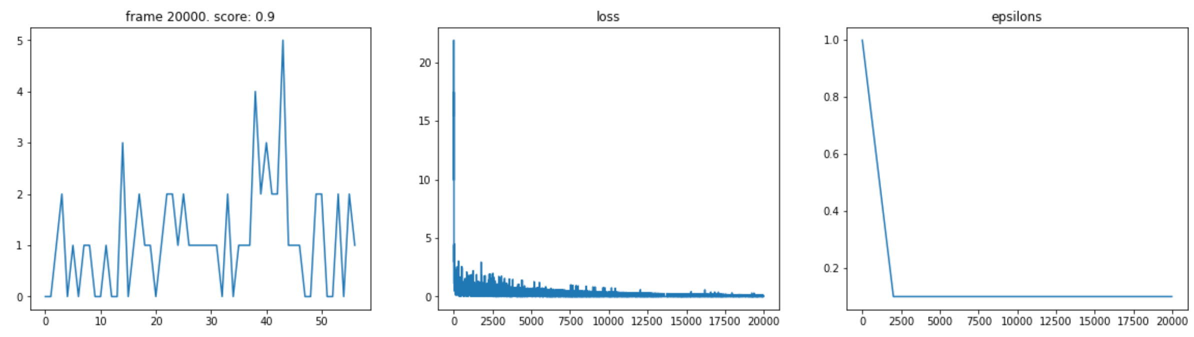
DeulingDQN的实现很简单，只需要修改Q网络的网络架构就可以了。而且可以和其他DQN的技巧，例如经验回放，固定网络，双网络计算目标等共用。

- 计算svalue，就是让网络预估的平均值。
- 计算avalue，对avalue进行归一化处理，也就是增加“A值的平均值为0”的限制。归一化的处理很简单，我们求A值的平均值，然后用A值减去平均值即可。

Advantage and value layers separated from feature layer：

```python
class Network(nn.Module):
    def __init__(self, in_dim: int, out_dim: int):
        """Initialization."""
        super(Network, self).__init__()

        # set common feature layer
        self.feature_layer = nn.Sequential(
            nn.Linear(in_dim, 128),
            nn.ReLU(),
        )

        # set advantage layer
        self.advantage_layer = nn.Sequential(
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, out_dim),
        )

        # set value layer
        self.value_layer = nn.Sequential(
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, 1),
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """Forward method implementation."""
        feature = self.feature_layer(x)

        value = self.value_layer(feature)
        advantage = self.advantage_layer(feature)

        q = value + advantage - advantage.mean(dim=-1, keepdim=True)

        return q

```

结果



5.分布式深度Q网络（Distributional Q-learning）

M. G. Bellemare et al., "A Distributional Perspective on Reinforcement Learning," arXiv preprint arXiv:1707.06887, 2017.

分布式 Q 学习（Bellemare、Dabney & Munos；2017）学习了折扣回报（discounted returns）的分类分布（代替了估计平均值）。

The authors argued the importance of learning the distribution of returns instead of the expected return, and they proposed to model such distributions with probability masses placed on a discrete support z, where z is a vector with $N_{atoms} \in \mathbb{N}^+$ atoms, defined by $z_i = V_{min} + (i-1)\frac{V_{max}-V_{min}}{N-1}$ for $i \in \{1, \ldots, N_{atoms}\}$.

在传统DQN中，网络输出的是动作价值Q的估计，但是其实还是忽略了很多信息。假如两个动作能够获得的价值期望相同都是20，第一个动作有90%的情况是10，10%的情况下是110，第二个动作的50%的情况下是25，50%的情况下是15，那么虽然期望一样，但是要想减少风险，就应该选择后一种动作，只输出期望值看不到背后隐含的风险。如果采用分布视角（deistributional perspective）来建模，可以的得到更好更稳定的结果。我们可以把价值限定在$[V_{min}, V_{max}]$之间，选择N个等距的价值采样点，通过神经网络输出这N个采样点的概率，通过Q和TargetQ网络得到估计的价值分布和目标的价值分布，计算两个分布之间的差距。

The key insight is that return distributions satisfy a variant of Bellman's equation. For a given state S_t and action A_t, the distribution of the returns under the optimal policy π^* should match a target distribution defined by taking the distribution for the next state S_{t+1} and action $a^*_{t+1} = \pi^*(S_{t+1})$, contracting

it towards zero according to the discount, and shifting it by the reward (or distribution of rewards, in the stochastic case). A distributional variant of Q-learning is then derived by first constructing a new support for the target distribution, and then minimizing the Kullbeck-Leibler divergence between the distribution d_t and the target distribution

$$d'_t = (R_{t+1} + \gamma_{t+1} z, p_{\hat{\theta}}(S_{t+1}, \hat{a}^*_{t+1})),$$
$$D_{KL}(\phi_z d'_t \| d_t).$$

Here ϕ_z is a L2-projection of the target distribution onto the fixed support z, and $\hat{a}^*_{t+1} = \arg\max_a q_{\hat{\theta}}(S_{t+1}, a)$ is the greedy action with respect to the mean action values $q_{\hat{\theta}}(S_{t+1}, a) = z^T p_\theta(S_{t+1}, a)$ in state S_{t+1}.

现在的问题是如何表示这个分布，Distributional DQN的论文作者提出了一个叫做C51的算法，用51个等间距的atoms来描述一个分布，类似直方图。

这时候网络的架构需要做一定的调整，原DQN输入的是状态s，输出的是一个包含各个动作价值的向量$(Q(s,a1), Q(s,a2), \ldots, Q(s,a_m))(Q(s,a1), Q(s,a2), \ldots, Q(s,am))$。在Distributional DQN中，输入不变，输出变成一个矩阵，每一行代表一个动作，而列则代表了直方图中的每一条柱子，这个分布的范围是人为确定的，例如atoms有51个，范围为(0 , 50) (0,50)(0,50)，则atoms是

$0, 1, 2, \ldots, 49, 50$。而神经网络输出的每一个动作对应每一个atoms的概率，有N个atoms则有N个概率，相加为1，形式是 $p_0^a, p_1^a, \ldots, p_{N-1}^a$，每个概率和对应的atoms的值相乘可以得到动作价值的期望，即原本的Q值。确定了分布的形式之后，接下来就是如何衡量两个分布的距离，以此来决定Loss函数。这里作者采用的是KL散度。

KL散度，也称为相对熵，是衡量两个分布差距的计算公式。公式如下：

$$\mathrm{D_{KL}(p\|q) = \sum_{i=1}^{N} p\left(x_i\right) * \left(\log p\left(x_i\right) - \log q\left(x_i\right)\right) = \sum_{i=1}^{N} p\left(x_i\right) * \log \frac{p\left(x_i\right)}{q\left(x_i\right)}}$$

可以证明这个值大于等于0。当两者分布接近时，这个值会趋近于0，这个值越大，分布差距越大。

从式子中可以看出，KL散度计算的是数据的原分布与近似分布的概率的对数差的期望值。散度不具有交换性，可以理解为一个分布相比另一个分布的信息损失。

he parametrized distribution can be represented by a neural network, as in DQN, but with atom_size x out_dim outputs. A softmax is applied independently for each action dimension of the output to ensure that the distribution for each action is appropriately normalized.

To estimate q-values, we use inner product of each action's softmax distribution and support which is the set of atoms $\{z_i = V_{min} + i\Delta z : 0 \le i < N\}, \Delta z = \frac{V_{max} - V_{min}}{N-1}$.

$$Q(s_t, a_t) = \sum_i z_i p_i(s_t, a_t),$$

$$\text{where } p_i \text{ is the probability of } z_i \text{ (the output of softmax).}$$

在原DQN中，动作价值的更新目标是

$$\mathrm{r} + \gamma \max_{\mathrm{a}} \mathrm{Q}\left(\mathrm{s}_{t+1}, \mathrm{a}\right)$$

在分布式DQN中：

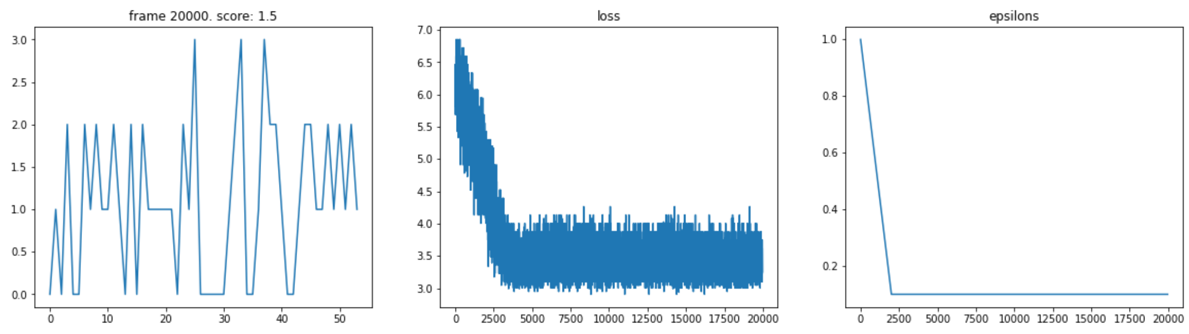
$$\mathrm{Q}\left(\mathrm{s}_{t+1}, \mathrm{a}\right) = \sum_{\mathrm{i}} \mathrm{z_i p_i}\left(\mathrm{s}_{t+1}, \mathrm{a}\right)$$

其中z就是直方图的横坐标的各个值，p就是对应的各个概率。那么更新目标$Z(s, a)$有p0概率为$r + \gamma z_0$，有p1概率为$r + \gamma z_1$，以此类推。当分布更新后，对应直方图的横坐标会发生偏移，因此我们需要调整分布。把$r + \gamma z_i$的分布均摊到z_i上，例如$r + \gamma z_0$ 位于z0和z1之间，我们可以计算它到两个点距离的比值决定均摊概率的比值。

总结

原来 DQN 输出的是一个 Q 值，但是实际上，一个状态的 Q 值应该是一个分布，某些值得概率比较大，某些值得概率比较小，所以分布式 RL 的做法就是不直接输出 Q 了，而是输出一个 Q 值得分布，如输出 100 单元表示将输出范围分为 100 份，每个值得大小表示该 Q 值得概率。

结果



6.噪声深度Q网络（Noisy Nets）

M. Fortunato et al., "Noisy Networks for Exploration." arXiv preprint arXiv:1706.10295, 2017.

Noisy DQN（Fortunato 等人；2017）使用随机网络层进行探索（exploration）。

NoisyNet is an exploration method that learns perturbations of the network weights to drive exploration. The key insight is that a single change to the weight vector can induce a consistent, and potentially very complex, state-dependent change in policy over multiple time steps.

Firstly, let's take a look into a linear layer of a neural network with p inputs and q outputs, represented by

$$y = wx + b,$$

where $x \in \mathbb{R}^p$ is the layer input, $w \in \mathbb{R}^{q\times p}$, and $b \in \mathbb{R}$ the bias.

RL过程中总会想办法增加agent的探索能力，传统的DQN通常采用ε-greedy的策略，即以ε的概率采取随机策略，通过在训练初期采取较大的ε来增加系统的探索性，训练后期减小ε来实现系统的稳定。另外一种办法就是噪声网络，即通过对参数增加噪声来增加模型的探索能力。一般噪声会添加在全连接层。

假如两层神经元的个数分别是p和q，那么w是q*p维的，b是q维的，如果b和w满足均值为μ，方差为σ的正态分布，同时存在一定的随机噪声N。假设噪声满足标准正态分布，那么前项计算公式变为：

The corresponding noisy linear layer is defined as:

$$y = (\mu^w + \sigma^w \odot \epsilon^w)x + \mu^b + \sigma^b \odot \epsilon^b,$$

where $\mu^w + \sigma^w \odot \epsilon^w$ and $\mu^b + \sigma^b \odot \epsilon^b$ replace w and b in the first linear layer equation. The parameters $\mu^w \in \mathbb{R}^{q\times p}, \mu^b \in \mathbb{R}^q, \sigma^w \in \mathbb{R}^{q\times p}$ and $\sigma^b \in \mathbb{R}^q$ are learnable, whereas $\epsilon^w \in \mathbb{R}^{q\times p}$ and $\epsilon^b \in \mathbb{R}^q$ are noise random variables which can be generated by one of the following two ways:

1. **Independent Gaussian noise**: the noise applied to each weight and bias is independent, where each random noise entry is drawn from a unit Gaussian distribution. This means that for each noisy linear layer, there are $pq + q$ noise variables (for p inputs to the layer and q outputs).
2. **Factorised Gaussian noise:** This is a more computationally efficient way. It produces 2 random Gaussian noise vectors (p, q) and makes $pq + q$ noise entries by outer product as follows:

$$\begin{aligned} \epsilon^w_{i,j} &= f(\epsilon_i)f(\epsilon_j), \\ \epsilon^b_j &= f(\epsilon_i), \\ \text{where } f(x) &= sgn(x)\sqrt{|x|}. \end{aligned}$$

In all experiements of the paper, the authors used Factorised Gaussian noise, so we will go for it as well.

产生噪声的方法主要有两种，一种是Independent Gaussian noise，一种是Factorised Gaussian noise。

- 对于DQN这种很稠密的算法来说我们往往使用Factorised Gaussian noise。

  Factorised Gaussian noise 相对节省计算量，它给每一个神经元一个噪声，图中只需要p + q个随机噪声。
  具体到每一个神经元上，w_{ij} 的噪声为：

$$\mathrm{N}^{\mathrm{w}}_{\mathrm{i,j}} = \mathrm{f}\left(\mathrm{N_i}\right)\mathrm{f}\left(\mathrm{N_j}\right)$$

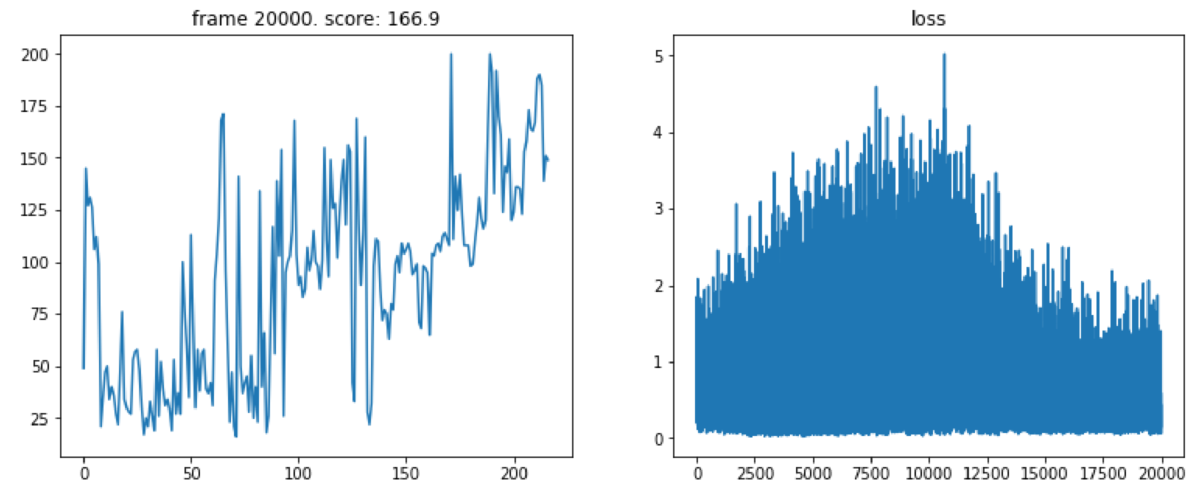
- 对应A3C这种并行更依赖采样的方法我们可以使用Independent Gaussian noise。是给每一个权重都添加一个独立的随机噪声，一共需要 p(q+1)个随机噪声。

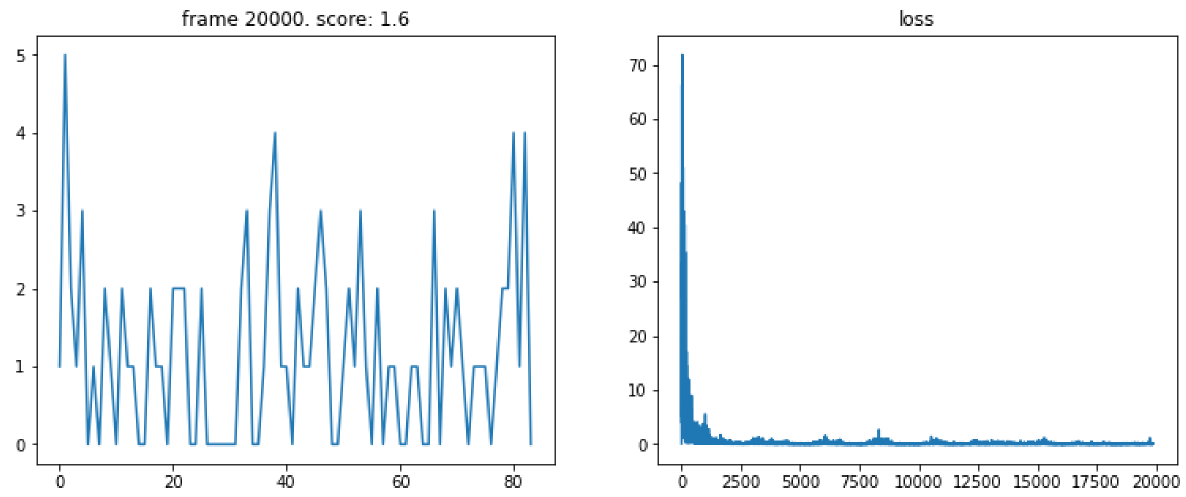
总结

核心思想：改正的是 $\epsilon - greedy$ 的方法，$\epsilon - greedy$ 的方法在很多环境中需要进行很多的探索才能找到第一个 reward，该方法就是在权值上直接乘一个噪声，使得网络输出本来就含有一定的噪声。

结果

上图为CartPole-v0训练结果，下图为Breakout-ram-v0训练结果。





7.算法融合

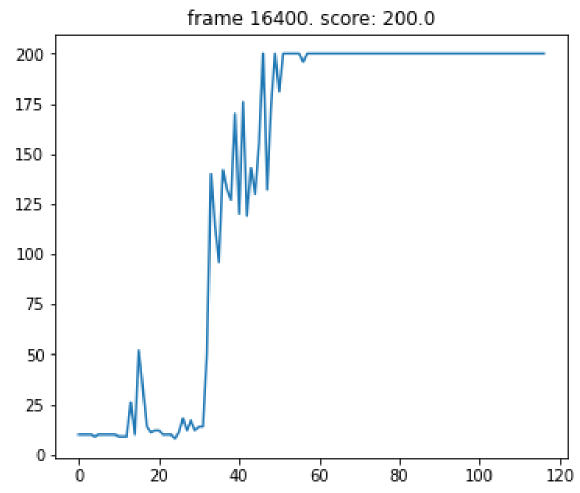
由于这些方法并不冲突，且共享一个架构，可以整合在一个DQN网络中，可行的融合方法如下：

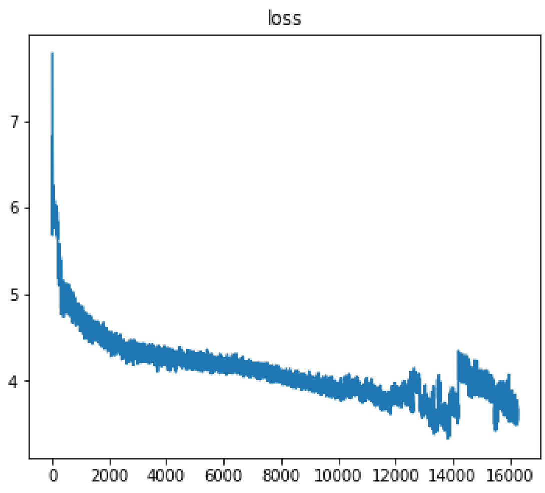
1. 构建优先经验回放池
2. 构建噪声层
3. **NoisyNet + DuelingNet + Categorical DQN**
4. 构建Agent，在其中实现Double DQN

Method	Note
select_action	select an action from the input state.
step	take an action and return the response of the env.
compute_dqn_loss	return dqn loss.
update_model	update the model by gradient descent.
target_hard_update	hard update from the local model to the target model.
train	train the agent during num_frames.
test	test the agent (1 episode).
plot	plot the training progresses.

```
Attribute:

•    env (gym.Env): openAI Gym environment

•    memory (PrioritizedReplayBuffer): replay memory to store transitions

•    batch_size (int): batch size for sampling

•    target_update (int): period for target model's hard update

•    gamma (float): discount factor

•    dqn (Network): model to train and select actions

•    dqn_target (Network): target model to update

•    optimizer (torch.optim): optimizer for training dqn

•    transition (list): transition information including

•            state, action, reward, next_state, done

•    v_min (float): min value of support

•    v_max (float): max value of support

•    atom_size (int): the unit number of support

•    support (torch.Tensor): support for categorical dqn

•    use_n_step (bool): whether to use n_step memory

•    n_step (int): step number to calculate n-step td error

•    memory_n (ReplayBuffer): n-step replay buffer
```

由于训练次数少Breakout-ram-v0训练结果运行效果不好，这里展示一下运行CartPole-v0的结果：





参考资料

[OpenAI spinning-up]

Segment Tree | Sum of given range - GeeksforGeeks

[用可视化直观理解DQNDQN实战篇] - 知乎 (zhihu.com)

Double DQN原理是什么，怎样实现？（附代码） - 知乎 (zhihu.com)

强化学习基础篇（十）OpenAI Gym环境汇总 - 简书 (jianshu.com)

(31条消息) 强化学习之DQN超级进化版Rainbow微笑小星的博客-*CSDN*博客多步dqn

BY571/DQN-Atari-Agents: DQN-Atari-Agents: Modularized & Parallel PyTorch implementation of several DQN Agents, i.a. DDQN, Dueling DQN, Noisy DQN, C51, Rainbow, and DRQN (github.com)

mohith-sakthivel/rainbow_dqn: Implementation of the Rainbow DQN algorithm published by DeepMind (github.com)

Curt-Park/rainbow-is-all-you-need: Rainbow is all you need! A step-by-step tutorial from DQN to Rainbow (github.com)

DQN系列(3): 优先级经验回放(Prioritized Experience Replay)论文阅读、原理及实现 - 腾讯云开发者社区-腾讯云 (tencent.com)