Report

分析代码

主干部分

代码的主要部分是main.py中的for循环：

```
for step in progressive:
    if done:
        observations, _, _ = env.reset()
        for obs in observations:
            obs_queue.append(obs)

    training = len(memory) > WARM_STEPS
    state = env.make_state(obs_queue).to(device).float()
    action = agent.run(state, training)
    obs, reward, done = env.step(action)
    obs_queue.append(obs)
    memory.push(env.make_folded_state(obs_queue), action, reward, done)

    if step % POLICY_UPDATE == 0 and training:
        agent.learn(memory, BATCH_SIZE)

    if step % TARGET_UPDATE == 0:
        agent.sync()

    if step % EVALUATE_FREQ == 0:
        avg_reward, frames = env.evaluate(obs_queue, agent, render=RENDER)
        with open("rewards.txt", "a") as fp:
            fp.write(f"{step//EVALUATE_FREQ:3d} {step:8d} {avg_reward:.1f}\n")
        if RENDER:
            prefix = f"eval_{step//EVALUATE_FREQ:03d}"
            os.mkdir(prefix)
            for ind, frame in enumerate(frames):
```

```
28                 with open(os.path.join(prefix, f"
{ind:06d}.png"), "wb") as fp:
29                     frame.save(fp, format="png")
30         agent.save(os.path.join(
31             SAVE_PREFIX,
f"model_{step//EVALUATE_FREQ:03d}"))
32         done = True
```

其中 progressive 是一个tqdm类，tqdm提供了将可视化枚举的工具，所以从实际意义上来说，这里的第一行与

```
1 for step in range(MAX_STEPS):
```

并没有不同。

循环的主体部分是第 7-12 行，内容是首先判断是否要开始训练，然后从环境中根据观测队列读取一个状态，然后让agent根据这个状态采取一个动作，再让环境判定这个动作产生的下一步状态和奖励，把这个状态加进观测队列和记忆中。

此外还有几个if语句，我们一个个分析：

```
1     if step % POLICY_UPDATE == 0 and training:
2         agent.learn(memory, BATCH_SIZE)
```

这个语句是每隔 POLICY_UPDATE 次就进行一次学习，更新agent的 policy 网络的参数。

```
1     if step % TARGET_UPDATE == 0:
2         agent.sync()
```

这个语句是每隔 TARGET_UPDATE 就将 policy 网络赋值给 target 网络；

```
if step % EVALUATE_FREQ == 0:
        avg_reward, frames = env.evaluate(obs_queue, agent, render=RENDER)
        with open("rewards.txt", "a") as fp:
            fp.write(f"{step//EVALUATE_FREQ:3d} {step:8d} {avg_reward:.1f}\n")
        if RENDER:
            prefix = f"eval_{step//EVALUATE_FREQ:03d}"
            os.mkdir(prefix)
            for ind, frame in enumerate(frames):
                with open(os.path.join(prefix, f"{ind:06d}.png"), "wb") as fp:
                    frame.save(fp, format="png")
        agent.save(os.path.join(
            SAVE_PREFIX, f"model_{step//EVALUATE_FREQ:03d}"))
        done = True
```

这里是每隔 EVALUATE_FREQ 次就保存一次训练的结果，并测试效果如何，然后在下一次训练的时候将环境重置。

然后我们再详细地分析一下训练的过程：

```
    action = agent.run(state, training)
```

然后 agent.run() 是这样的：

```
    def run(self, state: TensorStack4, training: bool = False) -> int:
        """run suggests an action for the given state."""
        if training:
            self.__eps -= (self.__eps_start - self.__eps_final) / self.__eps_decay
            self.__eps = max(self.__eps, self.__eps_final)

        if self.__r.random() > self.__eps:
            with torch.no_grad():
                return self.__policy(state).max(1).indices.item()
        return self.__r.randint(0, self.__action_dim - 1)
```

这是一个 ϵ–贪婪法，不过当 training 为真时，ϵ 是不断衰减的，一开始时 $\epsilon = 1$，也就是说开始时相当于完全随机地选择动。当 len(memory) > WARM_STEPS 时开始训练。这是容易理解的，一开始先随机地选择action，容易获得更多样的信息，从而训练时能更容易地区分哪些action是好的，哪些是坏的，防止模型过拟合。

再来看看agent.learn()是啥样子：

```
    def learn(self, memory: ReplayMemory, batch_size: int)
-> float:
        """learn trains the value network via TD-
learning."""
        state_batch, action_batch, reward_batch,
next_batch, done_batch = \
            memory.sample(batch_size)

        values =
self.__policy(state_batch.float()).gather(1, action_batch)
        values_next =
self.__target(next_batch.float()).max(1).values.detach()
        expected = (self.__gamma *
values_next.unsqueeze(1)) * \
            (1. - done_batch) + reward_batch
        loss = F.smooth_l1_loss(values, expected)
        self.__optimizer.zero_grad()
        loss.backward()
        for param in self.__policy.parameters():
            param.grad.data.clamp_(-1, 1)
        self.__optimizer.step()
        return loss.item()
```

也比较好理解，先进行采样，然后计算 policy network 输出的值 和 target network 输出的值，用smooth_l1_loss函数计算损失(这个损失函数与普通的L1损失函数的差别也有待了解)，然后反向传播。

memory是如何 push 和 sample 的呢?

```
    def push(
            self,
            folded_state: TensorStack5,
            action: int,
            reward: int,
            done: bool,
```

```
    ) -> None:
        self.__m_states[self.__pos] = folded_state
        self.__m_actions[self.__pos, 0] = action
        self.__m_rewards[self.__pos, 0] = reward
        self.__m_dones[self.__pos, 0] = done

        self.__pos = (self.__pos + 1) % self.__capacity
        self.__size = max(self.__size, self.__pos)

    def sample(self, batch_size: int) -> Tuple[
            BatchState,
            BatchAction,
            BatchReward,
            BatchNext,
            BatchDone,
    ]:
        indices = torch.randint(0, high=self.__size, size=
(batch_size,))
        b_state = self.__m_states[indices,
:4].to(self.__device).float()
        b_next = self.__m_states[indices,
1:].to(self.__device).float()
        b_action =
self.__m_actions[indices].to(self.__device)
        b_reward =
self.__m_rewards[indices].to(self.__device).float()
        b_done =
self.__m_dones[indices].to(self.__device).float()
        return b_state, b_action, b_reward, b_next, b_done
```

push操作无非就是在一个循环数组中找下一个值，然后把它覆盖掉，非常简单；sample是随机产生batch_size个下表，把这些下标对应的经验返回。

最后再来看看它的神经网络：

```
class DQN(nn.Module):

    def __init__(self, action_dim, device):
        super(DQN, self).__init__()
        self.__conv1 = nn.Conv2d(4, 32, kernel_size=8,
stride=4, bias=False)
        self.__conv2 = nn.Conv2d(32, 64, kernel_size=4,
stride=2, bias=False)
```

```
        self.__conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1, bias=False)
        self.__fc1 = nn.Linear(64*7*7, 512)
        self.__fc2 = nn.Linear(512, action_dim)
        self.__device = device

    def forward(self, x):
        x = x / 255.
        x = F.relu(self.__conv1(x))
        x = F.relu(self.__conv2(x))
        x = F.relu(self.__conv3(x))
        x = F.relu(self.__fc1(x.view(x.size(0), -1)))
        return self.__fc2(x)
```

由三个卷积层和两个全连接层构成，卷积核的大小分别为 8、4、3，stride为 4、2、1，通道数为 32,64,64；激活函数为 relu，最后输出action_dim个值表示不同action的得分。

其它部分：

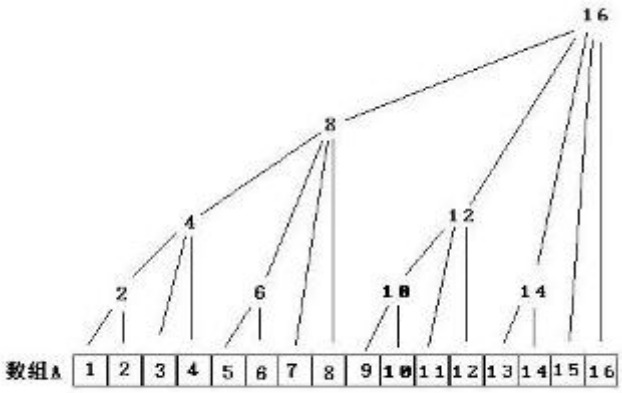
- utils_types.py中定义了很多变量类型，但都赋为了typing中的Any类。Any类型可以执行任何操作或方法调用，并将其赋值给任何变量。这样做仅是对程序员的一种提醒，方便程序员在其它文件中审慎地使用类型，而不会对静态检查有影响。
- utils_memory.py中初始化即固定记忆容量，分配好空间。
- utils_model.py的init_weights中使用了kaiming初始化方法，不知道具体怎么实现，可能是一种优化手段。
- utils_drl.py中
  - 优化器使用的是Adam优化算法，学习率为0.0000625，较小。
  - save只是个调用torch.save的接口。
- utils_env.py
  - 初始化使用了atari中的方法。
  - reset调用底层gym方法，并预先往前走底层step方法参数为0的5步（按注释这样的动作应该没有任何操作），这一步使初始观测队列非空，使后续main中的make_state可以顺利进行。
  - step同样调用底层gym方法。
  - get_frame是进行渲染的，只在RENDER为真时被调用，而main中RENDER为False（有很多相关代码，但都没有被调用）。

make_folded_state创建HTML文件并利用IPython来播放MP4视频，并没有被调用。

- to_tensor，get_action_dim，get_action_meanings，get_eval_lives是一些辅助小函数。
- make_state，make_folded_state都是根据观测队列返回状态，前者返回5个，后者返回4个。
- evaluate函数利用现有模型进行多次试运行（默认5*3=15次），返回平均效果

使用优先经验回放

参考 这篇博客 的做法，给memory设置优先级，给TD误差大的经验回放更高的优先级。具体做法是使用树状数组:



树状数组是一种支持 $O(logn)$ 单点修改和区间查询的数据结构（当然这里这用单点查询），我们每次往memory中添加一个经验时，计算它对应的TD误差，把树状数组对应的下表修改一下；而当我们进行采样时，可以使用这种方法：

从根节点出发，每次以 $左子节点的值/当前节点的值$ 的概率往左子节点走，否则留在原地，并将当前节点的值减去左子节点的值，直到走到叶子节点或者没有左子节点了为止，此时叶子结点对应的下标就是我们的采样值。伪代码如下：

```
right=根节点下标
sub_val=1<<(树的高度-1)
right_val=treearray[right]
    while sub_val != 0:
        left = right - sub_val
        left_val = treearray[left]
        if np.random.rand() < left_val / right_val:
            right = left
            right_val = left_val
        else:
            right_val -= left_val
            sub_val //= 2
return right
```

这样就能用 $O(logn)$ 的时间复杂度完成一次 push 操作和 sample 操作。经实际测试，在使用 Tesla T4 GPU的情况下，训练的速度约是原来的十分之一。树状数组的代码如下所示：（C++20行能写完的东西，python竟然用了40行）

```
class treearray:
    def __init__(self, capacity):
        self.__capacity = capacity
        self.__bits = math.ceil(math.log(capacity, 2))
        self.__max_len = 1 << self.__bits
        self.__array = torch.zeros((self.__max_len + 1,
1), dtype=torch.float)

    def add(self, loc, val):
        # 单点加法
        while loc < self.__max_len:
            self.__array[loc] += val
            loc += loc & (-loc)

    def get_array(self):
        return self.__array

    def get_prefix_sum(self, loc):
        # 得到一个前loc个值的和
        val = 0
        while loc != 0:
            val += self.__array[loc]
            loc -= loc & (-loc)
        return val
```

```
24
25      def change(self, loc, val):
26          # 单点修改，不过要先查询之前的值才能加上去
27          nowval = self.get_prefix_sum(loc) -
    self.get_prefix_sum(loc - 1)
28          #print(val,nowval)
29          self.add(loc, val - nowval)
30
31      def search(self):
32          # 进行采样
33          sub_val = (1 << (self.__bits - 1))
34          right = self.__max_len
35          right_val = copy(self.__array[right])
36          while sub_val != 0:
37              left = right - sub_val
38              left_val = copy(self.__array[left])
39              if np.random.rand() < left_val / right_val:
40                  right = left
41                  right_val = left_val
42              else:
43                  right_val -= left_val
44              sub_val //= 2
45          return right
```

此外，ReplayMemory类也要进行相应的修改，由于篇幅所限，这里不再赘述。

经测试，使用了优先经验回放之后运行速度差了几倍，在训练次数相同的情况下，模型的表现并不比没有原始版本强多少。这是优先经验回放版本：

```
0       0 4.3
1  100000 2.3
2  200000 1.3
3  300000 9.3
4  400000 7.0
5  500000 12.3
6  600000 11.7
7  700000 15.7
8  800000 20.7
```

这是原始版本：

```
0       0 4.3
1  100000 1.7
2  200000 3.0
3  300000 5.7
4  400000 10.7
5  500000 6.0
6  600000 12.7
7  700000 14.3
8  800000 19.0
```

也就是说，在训练时间相同的情况下，优先经验回放的效果还不如原始版本，这让我开始怀疑人生。分析了一下，可能有如下几个原因：

1. Memory 数量太多，而 policy network 的训练时很频繁的，也就是说很多经验的优先级可能是在很久以前生成，导致结果不准确，而如果经常更新Memory的优先级又显得很浪费时间，所以可以把Memory调小一点。
2. agent 训练时，loss要乘一个权重 w_j ，而这个权重太小，可能导致训练效果不佳。于是考虑把每一批的权重都乘上一个值，使得它们的平均值为1。

修改之后发现效果变差了。

一些感想

- GPU资源比较稀缺，这里使用的是一位同学的本机的GPU和上学期某课程的GPU节点。
- 炼丹时间很久，每次修改都要一小时左右才能看到一定的结果。
- 尝试调了一些参数，如学习率，α，β，Memsize，但都效果不佳。
- 最后还想多进行一些尝试，可惜没有时间了，如果能多给一点时间也许会做的更好。