Report

Describe the Deep Q-Network (7%)

Deep Q-Network 是 Reinforcement learnin 領域中一個演算法，旨在通過深度神經網絡來學習從觀察中直接生成動作的值函數，使代理可以在複雜的環境中學習最優策略，

DQN 主要特點包括以下幾點：

1.**狀態表示**：DQN 接受環境的狀態作為輸入，狀態可以是圖像、原始像素值、遊戲中的像素等。

2.**動作值函數**：DQN 使用深度神經網絡來近似動作值函數（Q 函數），通常是卷積神經網絡（CNN），該函數將狀態作為輸入，輸出每個動作的估計值。

2.**經驗回放**：為了解決樣本相關性和非靜態分佈的問題，DQN 引入了經驗回放機制，將過去的經驗存儲起來，包含狀態、動作、獎勵和下一個狀態元組，並隨機抽樣這些經驗進行訓練。

3.Q-Learning 更新：從回放記憶中隨機抽取一些經驗的小批量來更新神經網絡權重，只在最小化預測值和目標動作值之間的差異。

4.探索和利用：通過基於當前策略貪婪地選擇動作，或者隨機地選擇動作來平衡探索和利用。

5.目標網絡：使用與主網絡相同架構的單獨目標網絡來穩定學習過程。定期從主網絡中複製權重以更新目標網絡，以防止學習過程中的不穩定。。

重複步驟 1 到 6：與環境交互，收集經驗，更新網絡，並迭代地優化策略直到收斂。

Some of the information is based on https://medium.com/@shruti.dhumne/deep-q-network-dqn-90e1a8799871

Describe the architecture of your PacmanActionCNN (7%)

self.conv2 = nn.Conv2d(16, 32, kernel_size=4, stride=2): :

將 out_channels 設為 32 有以下考慮：

模型能夠具有足夠的表達能力來捕捉圖像中的各種特徵，同時還能保持相對較低的模型複雜度和計算量。如果輸出通道數過小，可能會限制模型對特徵的表達能力；而如果太大，則可能會增加模型的複雜度和計算成本。

另外我做了多個實驗來決定 kernel_size：

較小的卷積核能夠更好地捕捉局部細節特徵，而較大的卷積核則能夠捕捉到更大範圍的特徵。

我將 stride 固定為 2，然後更改 kernel_size 來看在 10000 步中誰的效果最佳

```
self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=2)
```

```
                    | 9999/100000 [10:29<3:18:11,  7.57it/s]Step: 10000, AvgScore: 210.0, ValueLos
s: 9947.4853515625
```

```
self.conv2 = nn.Conv2d(16, 32, kernel_size=4, stride=2)
```

```
  10%||  | 9999/100000 [09:38<2:49:08,  8.87it/s]
Step: 10000, AvgScore: 130.0, ValueLoss: 20.171
852111816406
```

```
self.conv2 = nn.Conv2d(16, 32, kernel_size=5, stride=2)
```

```
9999/100000 [11:55<3:13:10,  7.76it/s]Step: 10000, AvgScore: 70.0, ValueLoss: 16.01264762878418
```

由上對比，我選擇 kernel_size 大小為 4。雖然其平均分數與卷積核大小為 3 低，但值函數損失更低，這意味著模型的預測更準確。

Stride：

當 stride 設置較大時，模型在進行特徵提取時將對輸入圖像進行更粗糙的掃描，可能會丟失一些細節信息，但計算量較小，因為每個位置處理的次數減少了，反之當 stride 設置較小時，儘管保留了更多的細節信息，但計算量較大，因為每個位置處理的次數增加了。

```
self.conv2 = nn.Conv2d(16, 32, kernel_size=5, stride=1)
```

```
  4%|                                                  | 9999/250000 [13:45<10:20:10,  6.45it/s]
Step: 10000, AvgScore: 70.0, ValueLoss: 23.38239288330078
```

```
self.conv2 = nn.Conv2d(16, 32, kernel_size=5, stride=2)
```

```
9999/100000 [11:55<3:13:10,  7.76it/s]Step: 10000, AvgScore: 70.0, ValueLoss: 16.01264762878418
```

```
self.conv2 = nn.Conv2d(16, 32, kernel_size=5, stride=3)
```

```
  4%|                                                  | 9999/250000 [11:02<8:16:44,  8.05it/s]
Step: 10000, AvgScore: 70.0, ValueLoss: 28.02405548095703
```

由上對比，我選擇 stride 大小為 2，三者的 avgScore 相同但是 ValueLoss 以 stride = 2 最低，模型的預測更準確。

基於以上數據我認為 self.conv2 = nn.Conv2d(16, 32, kernel_size = 4, stride =2) 的 training model 應該有最好的效果。

H = ((84 - 8) // 4 +1) H=((H - 4) // 2 +1)：計算第二個卷積層的輸出特徵圖的高度 H。

self.in_features = 32 *H*H:：計算經過兩個卷積層後，展平後的特徵數。

self.fc1 = nn.Linear(self.in_features, 256)：定義了第一個全連接層，將展平後的特徵映射到一個 256 維的特徵空間。

self.fc2 = nn.Linear(256, action_dim)：定義了第二個全連接層，將 256 維的特徵映射到動作空間的維度，以輸出每個動作的 Q 值。

def forward(self, x):

x = F.relu(self.conv2(x))：將第一個卷積層的輸出通過第二個卷積層，然後應用 ReLU 激活函數。

x = x.view((-1, self.in_features))：將卷積層的輸出展平成一維向量，以便輸入全連接層。

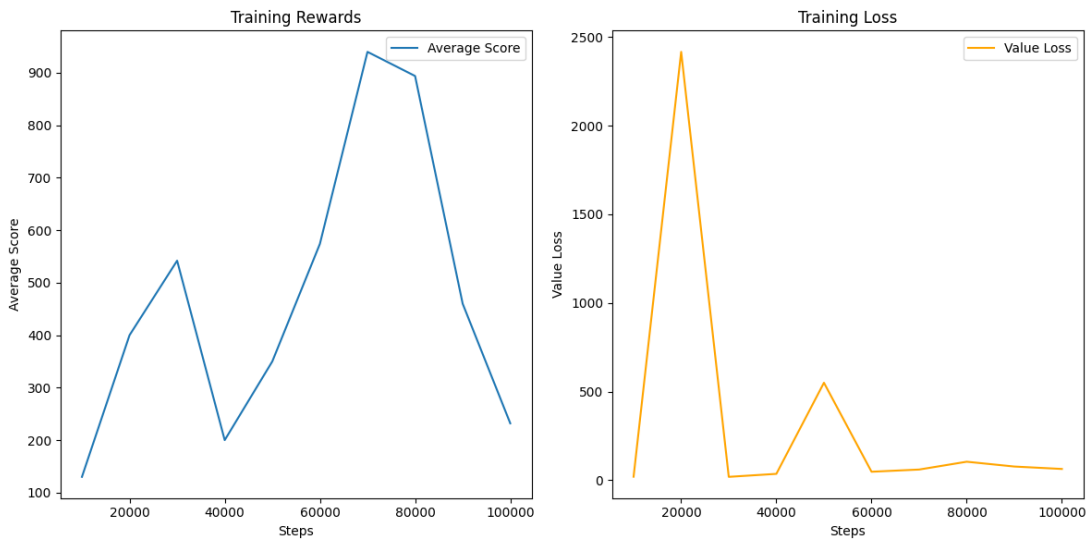
x = F.relu(self.fc1(x))：將展平後的特徵通過第一個全連接層，然後應用 ReLU 激活函數。

x = self.fc2(x)：將第一個全連接層的輸出通過第二個全連接層，以獲得每個動作的 Q 值。

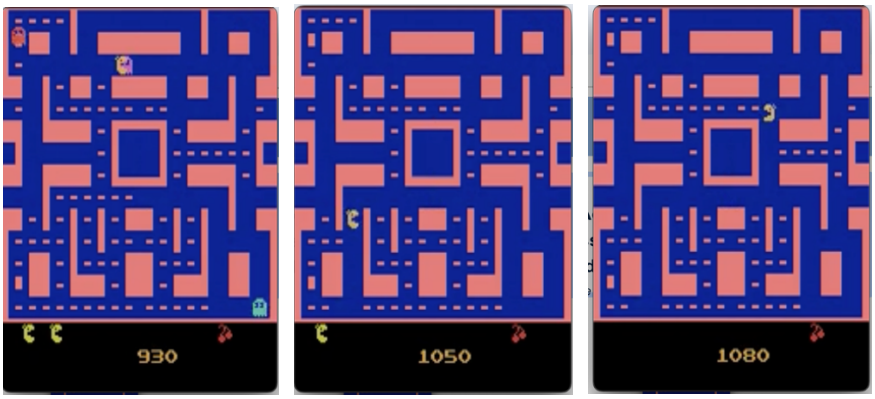
return x：返回網絡的輸出。

Plot your training curve, including both loss and rewards. (3%)

因為我意外刪除了第一次製作的結果因此，重新跑一次我將 max_step 設定為 100000。



Show screenshots from your evaluation video (3%)


930
1050
1080