ADL x MLDS Fall 2017

Homework 3 Report

Student: CSIE R06922068 Yu-Jing Lin 林裕景

Basic Model Description and Performance

Policy Gradient

Model

HW3 的第一部分：用 Policy Gradient RL 玩 Pong，我用以下的 model 實作的。

```
model = Sequential()
model.add(Conv2D(32, (8, 8), strides=(4, 4), padding='same',
                 activation='relu', kernel_initializer='he_normal', input_shape=self.state_size))
model.add(Conv2D(64, (4, 4), strides=(2, 2), padding='same',
                 activation='relu', kernel_initializer='he_normal'))
model.add(Flatten())
model.add(Dense(128, activation='relu', kernel_initializer='he_normal'))
model.add(Dense(self.action_size, activation='softmax', kernel_initializer='he_normal'))
opt = OPTIMIZER(lr=self.learning_rate)
model.compile(loss=LOSS, optimizer=opt)
```

首先是兩層的 CNN，參數分別是「32 個 kernels、filter size 為 8、strides 為 4」與「64 個 kernels、filter size 為 4、strides 為 2」，然後 Flatten 後接 hidden size 為 128 的 fully-connected layer，最後再接 output size 為 3 的 fully-connected layer 作為輸出。

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 20, 20, 32)        2080
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 10, 10, 64)        32832
_________________________________________________________________
flatten_1 (Flatten)          (None, 6400)              0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               819328
_________________________________________________________________
dense_2 (Dense)              (None, 3)                 387
=================================================================
Total params: 854,627
Trainable params: 854,627
Non-trainable params: 0
_________________________________________________________________
```

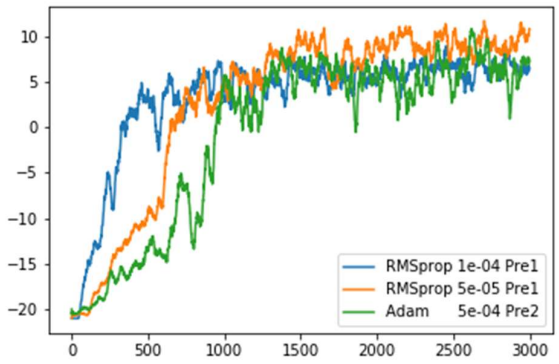
Model 的 input 是由 OpenAI Gym 的 Environment 經過 step 後回傳的 state，也就是遊戲的畫面。首先我對 state 做了兩種不同的 preprocess 方法，第一種是截去畫面的上下部分，只留下遊戲場地，然後用 rgb2gray 方法把 RGB 轉灰階，然後 resize 成 80x80 的大小；第二種（normalize）也是先截去上下部分，只取出紅色的 channel 並 resize 成 80x80 後，再將球和板子設為 1，其餘部分設為 0。經過上述的 preprocess 後，我再將這個畫面與前一個畫面相減形成真正為給 model 的 state，如此一來 model 才能識別遊戲的變化（球往哪裡飛、板子往哪邊移等等）。

在這裡，每一層的參數初始化方法我都是設定為 he_normal，是一種以零為中心的 truncated normal distribution，且為了加快訓練的速度，我將 agent 可以採取的 action 從原本的動作 0~5 縮減一半，限制為 1, 2, 3 三個動作，分別代表「不動、向上和向下」。

Performance

我做了 RMSprop 和 Adam 兩種 optimizer、不同 lr、不同 preprocess 的實驗。

左圖為 training 紀錄（經過 30 的 moving average），右表為 testing 分數（rnd seed 設 0）。



Parameters	Score
RMSprop 1e-04 Pre1	6.36666666667
RMSprop 5e-05 Pre1	11.0666666667
Adam 5e-04 Pre2	10.7666666667

雖然較大的 learning rate 可以 train 比較快，但是到後期是較小 learning rate 的 performance 比較好。而 Adam 搭配 normalized 的 preprocessing 會比較不穩。

我還是過不同的 model 架構，較少數量的 hidden size，較多層的結構，還有不一樣的 learning rate（太大太小都不行），但是都沒有成過 train 起來，有的是一直維持在-21~-19 分，也有經過長時間上升到-10~0 之間後卻怎麼樣也上不去的情況，才發現到即使是在 reinforcement learning 中，選擇適合的 model 架構依然是相當重要的一環。

DQN

Model

HW3 的第二部分：用 Deep-Q-Network RL 玩 Breakout，我用以下的 model 實作的。

```
model = Sequential()
model.add(Conv2D(32, (8, 8), strides=(4, 4), border_mode='same', activation='relu', input_shape=self.model_input_shape))
model.add(Conv2D(64, (4, 4), strides=(2, 2), border_mode='same', activation='relu'))
model.add(Conv2D(64, (3, 3), strides=(1, 1), border_mode='same', activation='relu'))
model.add(Flatten())
model.add(Dense(512, activation='relu'))
# model.add(Dense(512))
# model.add(LeakyReLU())
if DUEL == 'none':
    model.add(Dense(self.action_size, activation='linear'))
else:
    model.add(Dense(self.action_size + 1, activation='linear'))
    if DUEL == 'avg':
        model.add(Lambda(lambda a: K.expand_dims(a[:, 0], axis=-1) + a[:, 1:] - K.mean(
            a[:, 1:], keepdims=True), output_shape=(self.action_size, )))
    elif DUEL == 'max':
        model.add(Lambda(lambda a: K.expand_dims(a[:, 0], axis=-1) + a[:, 1:] - K.max(
            a[:, 1:], keepdims=True), output_shape=(self.action_size, )))
    elif DUEL == 'naive':
        model.add(Lambda(lambda a: K.expand_dims(a[:, 0], axis=-1) + a[:, 1:], output_shape=(self.action_size, )))
opt = OPTIMIZER(lr=self.learning_rate)
model.compile(loss='mse', optimizer=opt)
```

這裡的 model 包含了 DQN 的 improvements，關於 improvements 會在下一段提到。

我使用架構類似第一部分的 model，先是三層 CNN，依序為「32 個 kernels、filter size 為 8、strides 為 4」、「64 個 kernels、filter size 為 4、strides 為 2」與「64 個 kernels、filter size 為 3、strides 為 1」，經過 Flatten 後接 hidden size 為 512 的 fully-connected layer，然後接出去到 output size 為 4 的 fully-connected layer。

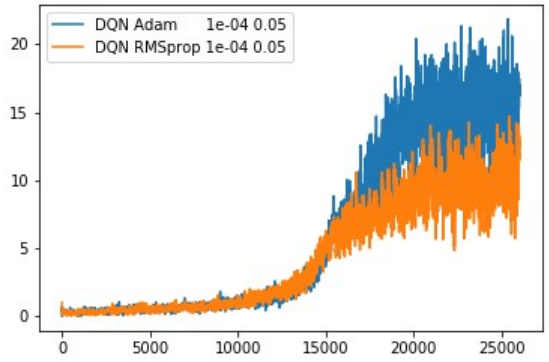
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 21, 21, 32)        8224
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 64)        32832
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 11, 11, 64)        36928
_________________________________________________________________
flatten_1 (Flatten)          (None, 7744)              0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               3965440
_________________________________________________________________
dense_2 (Dense)              (None, 4)                 2052
=================================================================
Total params: 4,045,476
Trainable params: 4,045,476
Non-trainable params: 0
_________________________________________________________________
```

我的 DQN 使用了兩個 networks，分別是 Q network 和 target network，前者很頻繁的更新而後者久久更新一次，看似很像 DDQN 但其實不然。在這裡是使用 target network predict 出最大的 Q value（max target Q value）去 train Q network 的 weights。會這樣更新是希望能提升 training 的穩定性，我認為要經過數次 training 的過程才能將 Q network 的 max Q 更新到符合 target network 中 max Q 的數值。

Performance

我做了 Adam 和 RMSprop 兩種 optimizer 的實驗。

左圖為 training 紀錄（經過 30 的 moving average），右表為 testing 分數（rnd seed 設 0）。

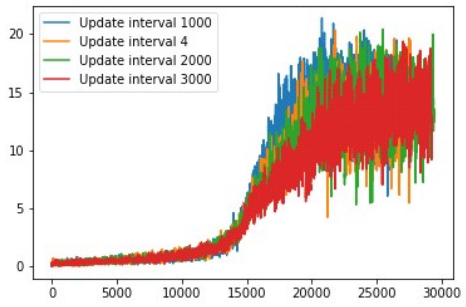


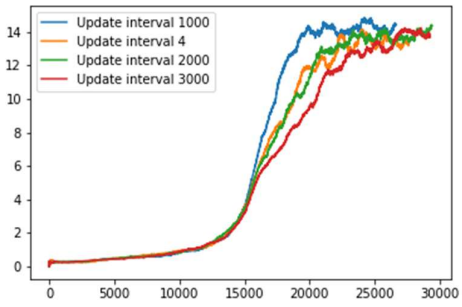
Parameters	Score
Adam 1e-04 0.05	62.92
RMSprop 1e-04 0.05	34.64

在相同的 learning rate 下，Adam 比 RMSprop 的 performance 高。由於在前面都在 exploring，到了約 14000 episodes 時，才降低到剩下 5%的機會採取 random action，因此直到 exploiting 時分數才開始飆升。

Experimenting with DQN Hyperparameters

以下圖（下一頁）為我選擇 update interval 作為 hyperparameter，調整 4, 1000, 2000, 3000 四種數值實驗的結果，發現在我的 model setting 下，update interval 取 1000 最好。





Improving DQN

我使用了以下兩種方法增強 DQN：Double DQN 與 Dueling DQN。

Double DQN

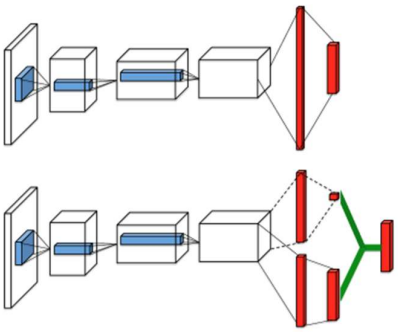
傳統的 DQN 有個問題是 Q network 會有預測上的誤差，容易高估 max Q value，經過層層遞進後會形成很大的誤差，因而出現 DDQN。

與原本 DQN 不同的是 Q network 的更新是先由 Q network 預測出最適合的 action，在取其對應的 target Q value 去更新 Q network。

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 21, 21, 32)        8224
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 11, 11, 64)        32832
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 11, 11, 64)        36928
_________________________________________________________________
flatten_2 (Flatten)          (None, 7744)              0
_________________________________________________________________
dense_4 (Dense)              (None, 512)               3965440
_________________________________________________________________
dense_5 (Dense)              (None, 5)                 2565
_________________________________________________________________
lambda_1 (Lambda)            (None, 4)                 0
=================================================================
Total params: 4,045,989
Trainable params: 4,045,989
Non-trainable params: 0
_________________________________________________________________
```

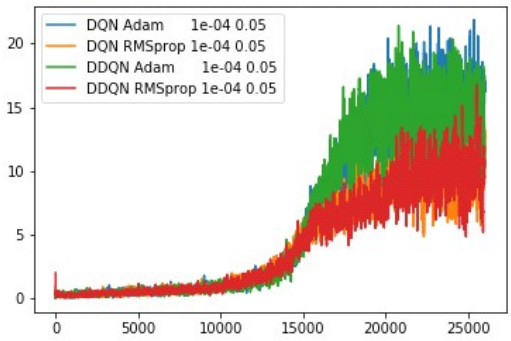
Dueling DQN

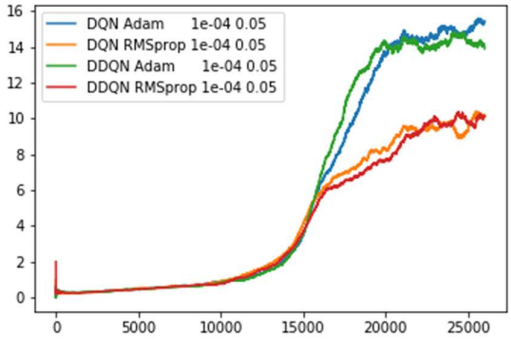
一般預測的結果是從 non-linear fully-connected layer 直接接出去 Q values，然而有人提出 advantage 的概念，不同的狀態和動作組合的重要性都不一樣，對於重要的動作我們希望能多更新到 network 上，而沒有影響的動作則希望對 network 少點影響，因此由 network predict advantage 的概念因而出現。Dueling DQN 的示意圖如右圖所示，上面是一般的 DQN network；下面則是 dueling DQN 的架構。藉由在最後 output 前多一層不一樣的 layer，分別預測 target Q value 和 advantage。



Performance

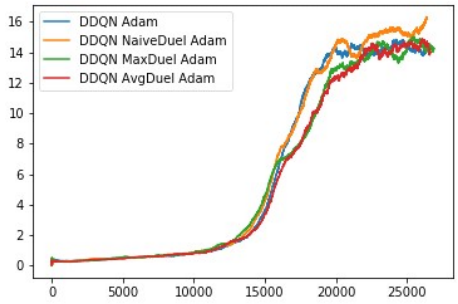
以下是我的 Dueling DDQN training 結果，跟 DQN 比較圖，以及 testing 的分數：

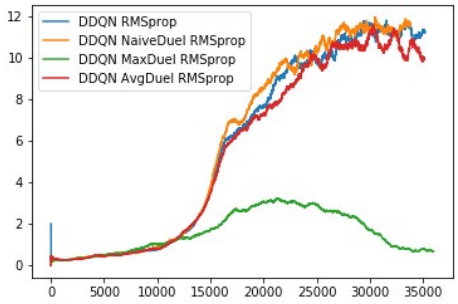




Parameters	Score
DQN Adam 1e-04 0.05	62.92
DQN RMSprop 1e-04 0.05	34.64
DDQN Adam 1e-04 0.05	66.89
DDQN RMSprop 1e-04 0.05	39.34

使用 Adam 和 RMSprop 之 DDQN 的 performance 有分別比 DQN 同樣 optimizer 的好。而從經過大小 1000 的 moving average（上右圖）也可以發現，DDQN 的 training 過程的確比傳統 DQN 好一些。

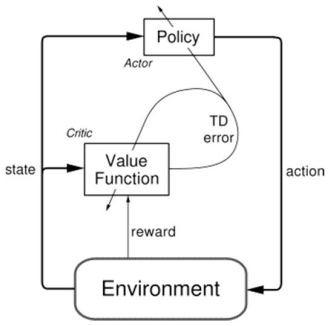




在同樣參數下，雖然有些 dueling 在 training 時比較差，但在後期會追過沒有 dueling 的 DDQN。值得注意的是，max dueling 方法在 RMSprop 上失效，不但沒 train 上去，從 20000 episodes 時還逐漸掉回 0。

Improving Policy with episodic A2C

Model



Advantage Actor Critic 是一個結合 policy gradient 和 function approximation 的方法，主要思想為透過 actor 基於機率選擇 action（PG），然後由 critic 預測這個狀態與動作的可能 reward（FA）。

而由於是要玩 pong 遊戲，聽說用 off-line training 的方法會 train 不起來，因此我實作的 A2C 是採取 episodic 的更新方式，也就是 n-step training，在遊戲結束後才一次更新 memorized history。

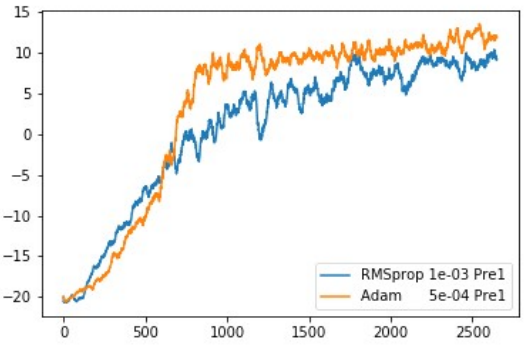
Pseudo code 如下（去掉 asynchronous 的部分即為 A2C）：

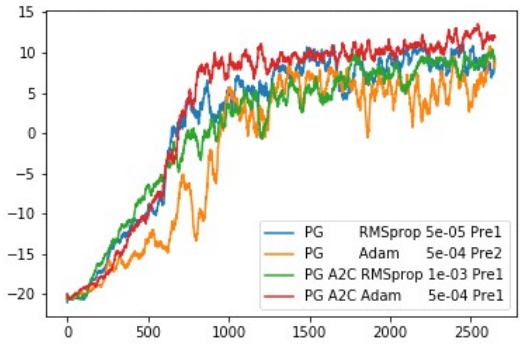
Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ and θ_v and global shared counter $T = 0$
// Assume thread-specific parameter vectors θ' and θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t+1$
    $T \leftarrow T+1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial (R - V(s_i;\theta'_v))^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

Performance

以下是我的 PG A2C training 結果，跟 PG 比較圖，以及 PG A2C 的 testing 分數：





Parameters	Score
RMSprop 1e-03 Pre1	10.4333333333
Adam 5e-04 Pre1	12.3666666667

PG A2C

Parameters	Score
RMSprop 5e-05 Pre1	11.0666666667
Adam 5e-04 Pre2	10.7666666667

PG

在實驗中我發現到使用了 A2C 的 PG，需要比較大的 learning rate，太小的話會 train 不起來。然而在 training 的時候 value loss 和 policy loss 都一直呈現亂跳的狀態，就像是 critic 和 actor 在不斷的互相抗衡，互相修正。

```
Episode: 4351 - Score: 11.0 - Step: 4108 - Value Loss: 311.7691650390625 - Policy Loss: -12.211324691772461 - Entropy: 0.29446929693222046.
Episode: 4352 - Score: 18.0 - Step: 2728 - Value Loss: 173.0712127685547 - Policy Loss: 353.8688049316406 - Entropy: 0.2986921966075897.
Episode: 4353 - Score: 15.0 - Step: 2976 - Value Loss: 283.4402770996094 - Policy Loss: 139.5428466796875 - Entropy: 0.2961297929286957.
Episode: 4354 - Score: 18.0 - Step: 2615 - Value Loss: 100.78397369384766 - Policy Loss: 159.230224609375 - Entropy: 0.29972565174102783.
Episode: 4355 - Score: 17.0 - Step: 2735 - Value Loss: 161.7696990966797 - Policy Loss: 72.55290985107422 - Entropy: 0.2981232702732086.
Episode: 4356 - Score: 16.0 - Step: 2852 - Value Loss: 184.0469207763672 - Policy Loss: -49.52626037597656 - Entropy: 0.29836973547935486.
Episode: 4357 - Score: 10.0 - Step: 3400 - Value Loss: 334.83319091796875 - Policy Loss: -438.7259521484375 - Entropy: 0.29396504163742065.
Episode: 4358 - Score: 15.0 - Step: 2925 - Value Loss: 251.1653289794922 - Policy Loss: -90.83651733398438 - Entropy: 0.30022451281547546.
Episode: 4359 - Score: 11.0 - Step: 3576 - Value Loss: 311.1526794433594 - Policy Loss: -212.16270446777344 - Entropy: 0.2965656209129333.
```

Fig. Training Adam 5e-04 Pre1 的最後 10 個 episodes

從 training 紀錄中也可以觀察到，使用 A2C 提升了 training 的穩定度，「Adam 5e-04 Pre1」在後期也達到比其他方法更高的 performance。