HW 4-3

Double DQN for random mode(Pytorch lightning)

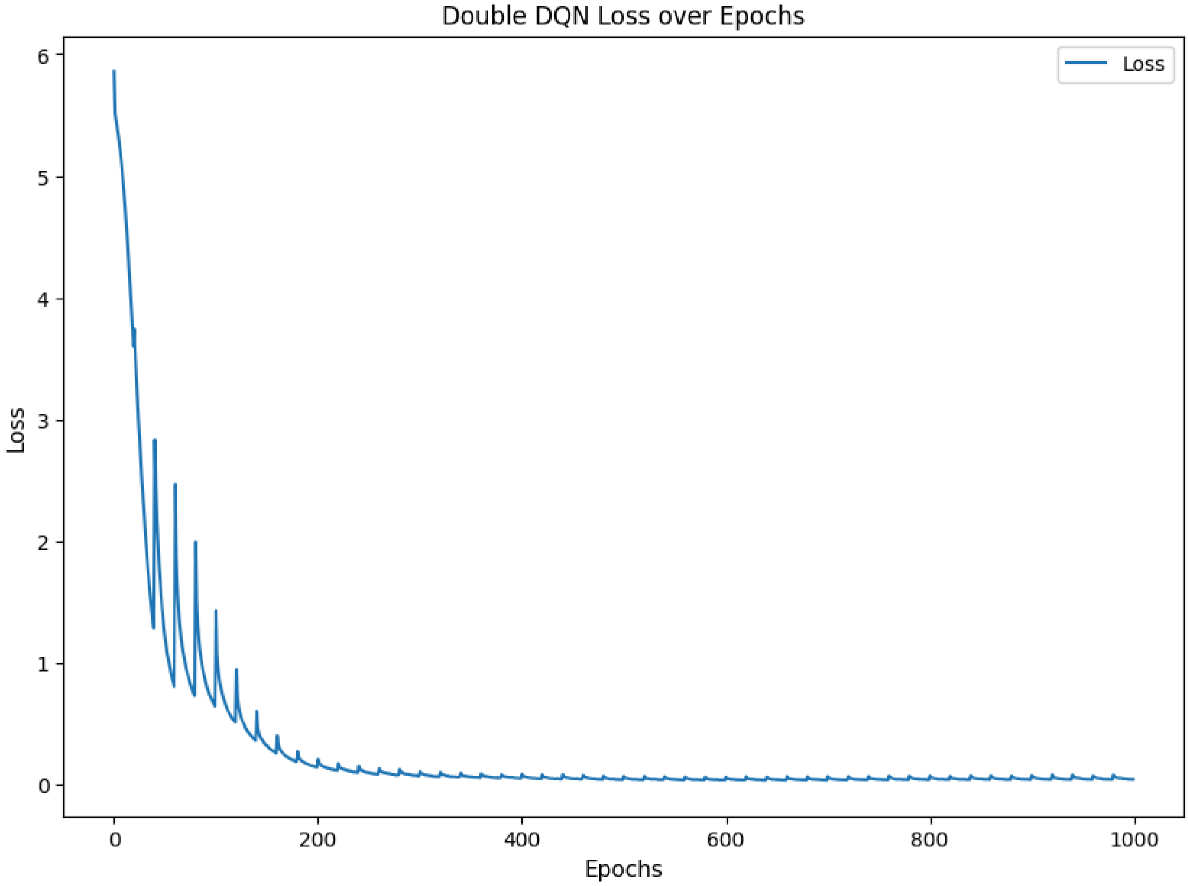
```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import random
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
import pytorch_lightning as pl
from pytorch_lightning.callbacks import Callback

# ==== 模擬 ReplayBuffer 和 Dataset ====
class ReplayBuffer:
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = []

    def push(self, *transition):
        if len(self.buffer) >= self.capacity:
            self.buffer.pop(0)
        self.buffer.append(transition)

    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        return zip(*batch)

    def __len__(self):
        return len(self.buffer)

class ReplayDataset(Dataset):
    def __init__(self, buffer):
        self.buffer = buffer.buffer

    def __len__(self):
```

```
        return len(self.buffer)

    def __getitem__(self, idx):
        state, action, reward, next_state, done = self.buffer[idx]
        return (
            torch.tensor(state, dtype=torch.float32),
            torch.tensor(action, dtype=torch.int64),
            torch.tensor(reward, dtype=torch.float32),
            torch.tensor(next_state, dtype=torch.float32),
            torch.tensor(done, dtype=torch.float32),
        )

# ==== Q-Network ====
class QNet(nn.Module):
    def __init__(self):
        super(QNet, self).__init__()
        self.fc1 = nn.Linear(64, 128)
        self.fc2 = nn.Linear(128, 4)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        return self.fc2(x)

# ==== Double DQN Lightning 模型 ====
class DoubleDQNLightning(pl.LightningModule):
    def __init__(self, gamma=0.9, lr=1e-3):
        super().__init__()
        self.q_net = QNet()
        self.model = QNet()
        self.target_model = QNet()
        self.target_model.load_state_dict(self.model.state_dict())
        self.loss_fn = nn.MSELoss()
        self.gamma = gamma
        self.lr = lr
        self.sync_interval = 20
        self.automatic_optimization = True

    def forward(self, x):
```

```
        return self.q_net(x)

    def training_step(self, batch, batch_idx):
        state, action, reward, next_state, done = batch

        q_values = self.model(state)
        q_value = q_values.gather(1, action.unsqueeze(1)).squeeze(1)

        # Double DQN 核心邏輯
        with torch.no_grad():
            next_actions = self.model(next_state).argmax(dim=1)
            next_q_values = self.target_model(next_state)
            next_q_value = next_q_values.gather(1, next_actions.unsqueeze(1)).squ
            expected_q = reward + self.gamma * next_q_value * (1 - done)

        loss = self.loss_fn(q_value, expected_q)

        # 每 sync_interval 次更新 target model
        if self.current_epoch % self.sync_interval == 0 and batch_idx == 0:
            self.target_model.load_state_dict(self.model.state_dict())

        self.log("loss", loss, on_epoch=True, prog_bar=True)
        return loss
        # return {"loss": loss}

    def configure_optimizers(self):
        return optim.Adam(self.model.parameters(), lr=self.lr)

# ==== 自定義 Callback 來紀錄 loss ====
class LossHistoryCallback(Callback):
    def __init__(self):
        self.losses = []

    def on_train_epoch_end(self, trainer, pl_module):
        loss = trainer.callback_metrics.get("loss")
        if loss is not None:
            self.losses.append(loss.item())
            print(f"Epoch {trainer.current_epoch}, Loss: {loss.item():.4f}")
```

```
# ==== 模擬 buffer 並創建 DataLoader ====
buffer = ReplayBuffer(capacity=10000)


for _ in range(2000):  # 探索階段
    game = Gridworld(size=4, mode='random')
    state = game.board.render_np().reshape(64,) + np.random.rand(64)/100.0
    state1 = state.astype(np.float32)
    done = False
    while not done:
        action = np.random.randint(0, 4)
        game.makeMove(action_set[action])
        next_state = game.board.render_np().reshape(64,) + np.random.rand(64)
        reward = game.reward()
        done = reward != -1
        buffer.push(state1, action, reward, next_state.astype(np.float32), float(do
        state1 = next_state

dataset = ReplayDataset(buffer)
dataloader = DataLoader(dataset, batch_size=128, shuffle=True)

# ==== 訓練 ====
model = DoubleDQNLightning()
loss_callback = LossHistoryCallback()

trainer = pl.Trainer(
    max_epochs=1000,
    accelerator="gpu",  # 或 "auto"
    devices=1,
    callbacks=[loss_callback],
    logger=False  # 不用 tensorboard logger
)

trainer.fit(model, dataloader)

# ==== 繪製 Loss 圖 ====
plt.figure(figsize=(10, 7))
```

```
plt.plot(loss_callback.losses, label="Loss")
plt.xlabel("Epochs", fontsize=11)
plt.ylabel("Loss", fontsize=11)
plt.title("Double DQN Loss over Epochs")
plt.legend()
plt.show()
```



換成pytorch lightning版本的改動

- replay buffer

```
class ReplayBuffer:
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = []

    def push(self, *transition):
        if len(self.buffer) >= self.capacity:
```

```
            self.buffer.pop(0)
        self.buffer.append(transition)

    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        return zip(*batch)

    def __len__(self):
        return len(self.buffer)
```

在 DQN 中，**agent 不會直接用最新的一筆經驗學習**，而是會將經驗存進 buffer，再**隨機抽樣一批資料訓練**，這樣能夠：

- 打破資料的時間相關性（temporal correlation）
- 提升訓練穩定性（減少模型震盪）
- 更有效率地利用歷史資料（資料重複使用）

- replay dataset

```
class ReplayDataset(Dataset):
    def __init__(self, buffer):
        self.buffer = buffer.buffer

    def __len__(self):
        return len(self.buffer)

    def __getitem__(self, idx):
        state, action, reward, next_state, done = self.buffer[idx]
        return (
            torch.tensor(state, dtype=torch.float32),
            torch.tensor(action, dtype=torch.int64),
            torch.tensor(reward, dtype=torch.float32),
            torch.tensor(next_state, dtype=torch.float32),
            torch.tensor(done, dtype=torch.float32),
        )
```

為了將
`ReplayBuffer` 中儲存的經驗轉換成 **PyTorch 能用的 Dataset 格式**，這樣你就能搭配
`DataLoader` 來做 mini-batch 訓練

- test in static mode

```
Initial State:
[['+' '-' ' ' 'P']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 0; Taking action: d
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' 'P']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 1; Taking action: d
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' 'P']
 [' ' ' ' ' ' ' ']]
Move #: 2; Taking action: d
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' 'P']]
Move #: 3; Taking action: d
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' 'P']]
...
Game lost; too many moves.
False
Games played: 1000, # of wins: 0
Win percentage: 0.0%
```

- test in player mode

```
Initial State:
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' 'P']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 0; Taking action: d
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' 'P']
 [' ' ' ' ' ' ' ']]
Move #: 1; Taking action: d
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' 'P']]
Move #: 2; Taking action: d
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' 'P']]
Move #: 3; Taking action: d
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' 'P']]
...
Game lost; too many moves.
False
Games played: 1000, # of wins: 0
Win percentage: 0.0%
```

- test in random mode

```
Initial State:
[[' ' 'W' ' ' ' ']
 ['P' ' ' ' ' '+']
 [' ' ' ' ' ' ' ']
 [' ' '-' ' ' ' ']]
Move #: 0; Taking action: u
[['P' 'W' ' ' ' ']
 [' ' ' ' ' ' '+']
 [' ' ' ' ' ' ' ']
 [' ' '-' ' ' ' ']]
Move #: 1; Taking action: u
[['P' 'W' ' ' ' ']
 [' ' ' ' ' ' '+']
 [' ' ' ' ' ' ' ']
 [' ' '-' ' ' ' ']]
Move #: 2; Taking action: u
[['P' 'W' ' ' ' ']
 [' ' ' ' ' ' '+']
 [' ' ' ' ' ' ' ']
 [' ' '-' ' ' ' ']]
Move #: 3; Taking action: u
[['P' 'W' ' ' ' ']
 [' ' ' ' ' ' '+']
 [' ' ' ' ' ' ' ']
 [' ' '-' ' ' ' ']]
...
Game lost; too many moves.
False
Games played: 1000, # of wins: 127
Win percentage: 12.7%
```

可以看到在random mode的勝率已經不高，且也缺乏泛化性，在static, player兩個mode的勝率都是0。

需要加上一點traininig tips來改進

Double DQN for random mode(PyTorch Lightning) with training techniques

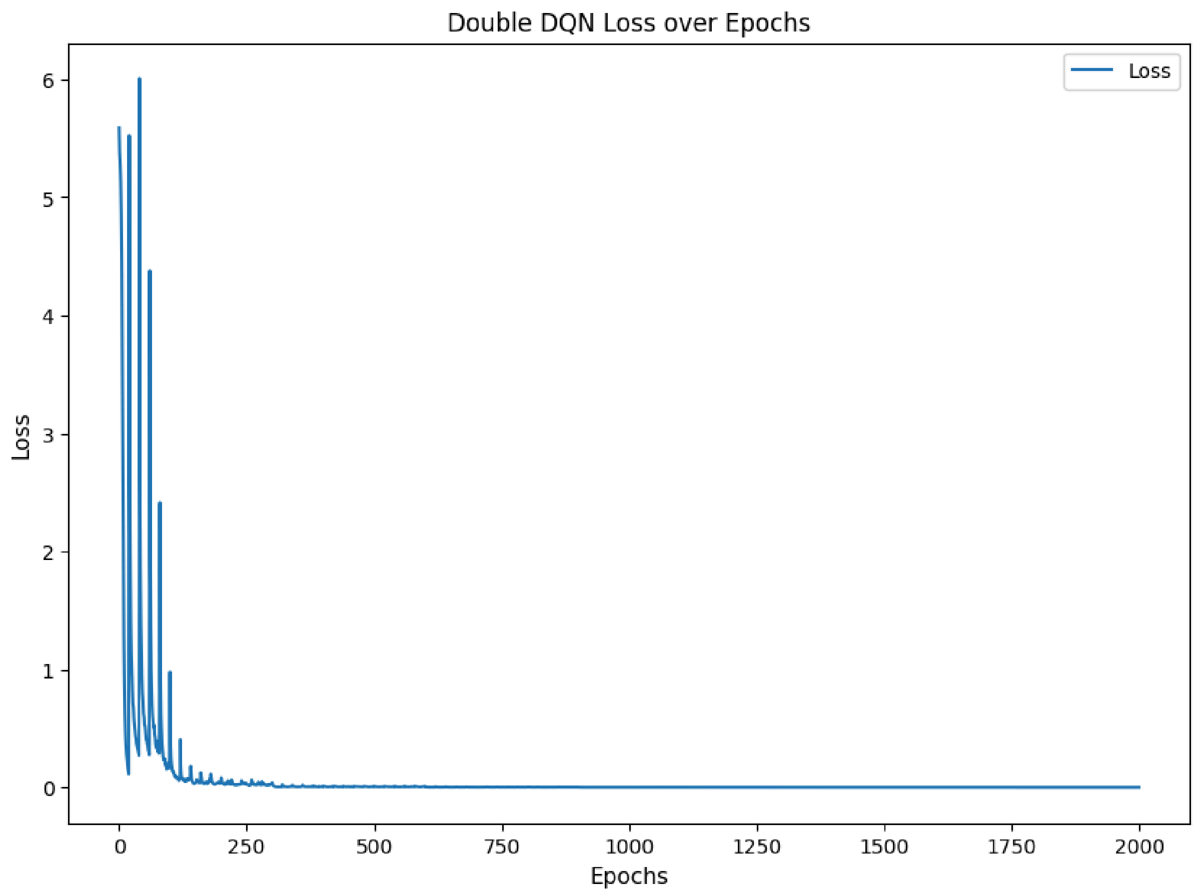
```
# ==== 模擬 ReplayBuffer 和 Dataset ====
class ReplayBuffer:
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = []
```

```
    def push(self, *transition):
        if len(self.buffer) >= self.capacity:
            self.buffer.pop(0)
        self.buffer.append(transition)

    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        return zip(*batch)

    def __len__(self):
        return len(self.buffer)

class ReplayDataset(Dataset):
    def __init__(self, buffer):
        self.buffer = buffer.buffer

    def __len__(self):
        return len(self.buffer)

    def __getitem__(self, idx):
        state, action, reward, next_state, done = self.buffer[idx]
        return (
            torch.tensor(state, dtype=torch.float32),
            torch.tensor(action, dtype=torch.int64),
            torch.tensor(reward, dtype=torch.float32),
            torch.tensor(next_state, dtype=torch.float32),
            torch.tensor(done, dtype=torch.float32),
        )

# ==== Q-Network ====
class QNet(nn.Module):
    def __init__(self):
        super(QNet, self).__init__()
        self.fc1 = nn.Linear(64, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 4)

    def forward(self, x):
```

```
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)

# ==== Double DQN Lightning 模型 ====
class DoubleDQNLightning(pl.LightningModule):
    def __init__(self, gamma=0.9, lr=1e-3):
        super().__init__()
        self.q_net = QNet()
        self.target_q_net = QNet()
        self.target_q_net.load_state_dict(self.q_net.state_dict())
        self.loss_fn = nn.MSELoss()
        self.gamma = gamma
        self.lr = lr
        self.sync_interval = 20
        self.epsilon = 1.0
        self.automatic_optimization = True

    def forward(self, x):
        return self.q_net(x)

    def training_step(self, batch, batch_idx):
        state, action, reward, next_state, done = batch

        q_values = self.q_net(state)
        q_value = q_values.gather(1, action.unsqueeze(1)).squeeze(1)

        with torch.no_grad():
            next_actions = self.q_net(next_state).argmax(dim=1)
            next_q_values = self.target_q_net(next_state)
            next_q_value = next_q_values.gather(1, next_actions.unsqueeze(1)).squ
            expected_q = reward + self.gamma * next_q_value * (1 - done)

        loss = self.loss_fn(q_value, expected_q)

        if self.current_epoch % self.sync_interval == 0 and batch_idx == 0:
            self.target_q_net.load_state_dict(self.q_net.state_dict())
```

```
        # Gradient clipping
        torch.nn.utils.clip_grad_norm_(self.q_net.parameters(), max_norm=1.0)

        self.log("loss", loss, on_epoch=True, prog_bar=True)
        return loss

    def configure_optimizers(self):
        optimizer = optim.Adam(self.q_net.parameters(), lr=self.lr)
        scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=300, gamma
        return [optimizer], [scheduler]

# ==== 自定義 Callback 來紀錄 loss ====
class LossHistoryCallback(Callback):
    def __init__(self):
        self.losses = []

    def on_train_epoch_end(self, trainer, pl_module):
        loss = trainer.callback_metrics.get("loss")
        if loss is not None:
            self.losses.append(loss.item())
            print(f"Epoch {trainer.current_epoch}, Loss: {loss.item():.4f}")

# ==== 模擬資料產生（用 random 模式） ====
buffer = ReplayBuffer(capacity=10000)
for _ in range(2000):
    game = Gridworld(size=4, mode='random')
    state = game.board.render_np().reshape(64,) + np.random.rand(64)/100.0
    state1 = state.astype(np.float32)
    done = False
    while not done:
        action = np.random.randint(0, 4)
        game.makeMove(action_set[action])
        next_state = game.board.render_np().reshape(64,) + np.random.rand(64
        reward = game.reward()
        done = reward != -1
        buffer.push(state1, action, reward, next_state.astype(np.float32), float(do
        state1 = next_state
```

```python
# ==== 建立資料集與 DataLoader ====
dataset = ReplayDataset(buffer)
dataloader = DataLoader(dataset, batch_size=128, shuffle=True)

# ==== 訓練 ====
model = DoubleDQNLightning()
loss_callback = LossHistoryCallback()

trainer = pl.Trainer(
    max_epochs=2000,
    accelerator="gpu",
    devices=1,
    callbacks=[loss_callback],
    logger=False
)

trainer.fit(model, dataloader)

# ==== 畫 Loss 曲線 ====
plt.figure(figsize=(10, 7))
plt.plot(loss_callback.losses, label="Loss")
plt.xlabel("Epochs", fontsize=11)
plt.ylabel("Loss", fontsize=11)
plt.title("Double DQN Loss over Epochs")
plt.legend()
plt.show()
```



與原版相比，做了哪些改動

- 更改Q-net架構

```
class QNet(nn.Module):
    def __init__(self):
        super(QNet, self).__init__()
        self.fc1 = nn.Linear(64, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 4)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)
```

多加了一層，目的為了增加泛化性，學習真正重要的特徵

- gradient clipping

```
    # Gradient clipping
    torch.nn.utils.clip_grad_norm_(self.q_net.parameters(), max_norm=1.0)
```

目的：防止梯度爆炸（尤其是在深層網路或強變化 reward 的情況下）。

作用：將梯度限制在範圍內，訓練更穩定。

這對 DQN 尤其重要，因為 TD error 在 early stage 通常很大。

- Target Network 同步機制

```
if self.current_epoch % self.sync_interval == 0 and batch_idx == 0:
    self.target_q_net.load_state_dict(self.q_net.state_dict())
```

目的：穩定 TD target（減少 `moving target` 問題）。

作用：定期將 main Q-network 的參數同步到 target Q-network。

- Learning Rate Scheduler

```
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=300, gamma=0.5)
```

目的：隨訓練逐步降低學習率，幫助模型收斂。

作用：每 300 個 epoch 把 lr 減半。

通常可以提升後期穩定性與精度。

- 隨機初始化資料（加 noise）

```
for _ in range(2000):
    game = Gridworld(size=4, mode='random')
    state = game.board.render_np().reshape(64,) + np.random.rand(64)/100.0
    state1 = state.astype(np.float32)
    done = False
    while not done:
        action = np.random.randint(0, 4)
        game.makeMove(action_set[action])
        next_state = game.board.render_np().reshape(64,) + np.random.rand(64)
        reward = game.reward()
        done = reward != -1
```

```
        buffer.push(state1, action, reward, next_state.astype(np.float32), float(do
        state1 = next_state
```

目的：讓影像狀態略微變動，減少 overfitting。

這是個簡單但有效的 data augmentation 技巧，類似 epsilon-greedy 的概念。

- test in static mode

```
Initial State:
[['+' '-' ' ' 'P']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 0; Taking action: u
[['+' '-' ' ' 'P']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 1; Taking action: u
[['+' '-' ' ' 'P']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 2; Taking action: u
[['+' '-' ' ' 'P']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 3; Taking action: l
[['+' '-' 'P' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
...
Game lost; too many moves.
False
Games played: 1000, # of wins: 2
Win percentage: 0.2%
```

- test in player mode

```
Initial State:
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' 'P' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 0; Taking action: l
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 ['P' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 1; Taking action: u
[['+' '-' ' ' ' ']
 ['P' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 2; Taking action: u
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Game won! Reward: 10
True
Initial State:
[['+' '-' 'P' ' ']
 [' ' 'W' ' ' ' ']
...
Game lost; too many moves.
False
Games played: 1000, # of wins: 254
Win percentage: 25.4%
```

- test in random mode

```
Initial State:
[[' ' ' ' '-' '+']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']
 ['W' 'P' ' ' ' ']]
Move #: 0; Taking action: u
[[' ' ' ' '-' '+']
 [' ' ' ' ' ' ' ']
 [' ' 'P' ' ' ' ']
 ['W' ' ' ' ' ' ']]
Move #: 1; Taking action: d
[[' ' ' ' '-' '+']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']
 ['W' 'P' ' ' ' ']]
Move #: 2; Taking action: u
[[' ' ' ' '-' '+']
 [' ' ' ' ' ' ' ']
 [' ' 'P' ' ' ' ']
 ['W' ' ' ' ' ' ']]
Move #: 3; Taking action: d
[[' ' ' ' '-' '+']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']
 ['W' 'P' ' ' ' ']]
...
Game won! Reward: 10
True
Games played: 1000, # of wins: 847
Win percentage: 84.7%
```

Dueling DQN for random mode(PyTorch Lightning)

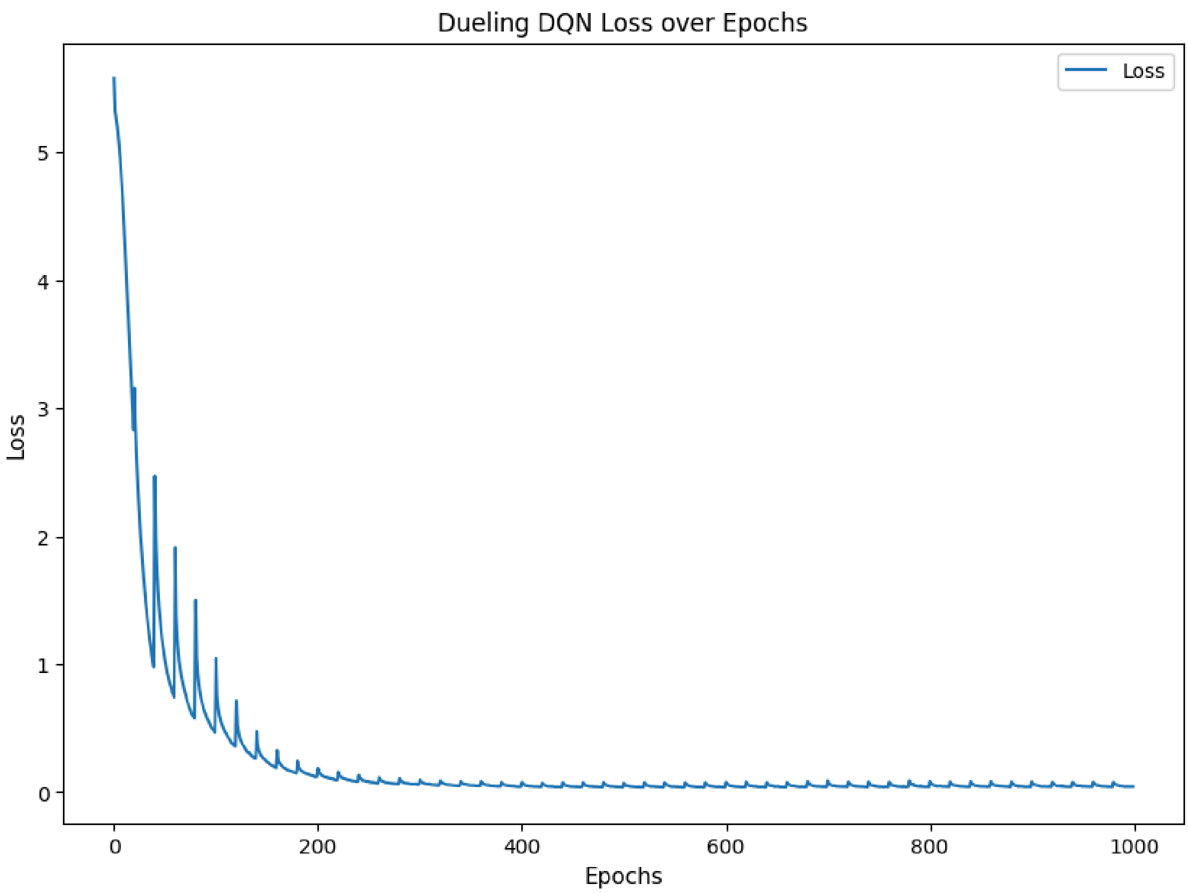
```
from pytorch_lightning.loggers import CSVLogger
buffer.buffer.clear()  # 清空 ReplayBuffer 中的所有內容

# ==== 模擬 ReplayBuffer 和 Dataset ====
class ReplayBuffer:
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = []

    def push(self, *transition):
        if len(self.buffer) >= self.capacity:
```

```
            self.buffer.pop(0)
        self.buffer.append(transition)

    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        return zip(*batch)

    def __len__(self):
        return len(self.buffer)

class ReplayDataset(Dataset):
    def __init__(self, buffer):
        self.buffer = buffer.buffer

    def __len__(self):
        return len(self.buffer)

    def __getitem__(self, idx):
        state, action, reward, next_state, done = self.buffer[idx]
        return (
            torch.tensor(state, dtype=torch.float32),
            torch.tensor(action, dtype=torch.int64),
            torch.tensor(reward, dtype=torch.float32),
            torch.tensor(next_state, dtype=torch.float32),
            torch.tensor(done, dtype=torch.float32),
        )

# ==== Dueling Q-Network (Dueling DQN) ====
class DuelingQNet(nn.Module):
    def __init__(self):
        super(DuelingQNet, self).__init__()
        self.fc1 = nn.Linear(64, 128)

        # 狀態價值（Value）分支
        self.value_fc = nn.Linear(128, 1)

        # 優勢（Advantage）分支
        self.advantage_fc = nn.Linear(128, 4)
```

```
    def forward(self, x):
        x = torch.relu(self.fc1(x))

        # 計算狀態價值
        value = self.value_fc(x)

        # 計算優勢函數
        advantage = self.advantage_fc(x)

        # Q值是價值加上優勢（進行標準化處理）
        q_values = value + (advantage - advantage.mean(dim=1, keepdim=True))

        return q_values

# ==== Dueling DQN Lightning 模型 ====
class DuelingDQNLightning(pl.LightningModule):
    def __init__(self, gamma=0.9, lr=1e-3):
        super().__init__()
        self.q_net = DuelingQNet()
        self.model = DuelingQNet()
        self.target_model = DuelingQNet()
        self.target_model.load_state_dict(self.model.state_dict())
        self.loss_fn = nn.MSELoss()
        self.gamma = gamma
        self.lr = lr
        self.sync_interval = 20
        self.automatic_optimization = True

    def forward(self, x):
        return self.q_net(x)

    def training_step(self, batch, batch_idx):
        state, action, reward, next_state, done = batch

        q_values = self.model(state)
        q_value = q_values.gather(1, action.unsqueeze(1)).squeeze(1)
```

```
        # Dueling DQN 核心邏輯
        with torch.no_grad():
            next_actions = self.model(next_state).argmax(dim=1)
            next_q_values = self.target_model(next_state)
            next_q_value = next_q_values.gather(1, next_actions.unsqueeze(1)).squ
            expected_q = reward + self.gamma * next_q_value * (1 - done)

        loss = self.loss_fn(q_value, expected_q)

        # 每 sync_interval 次更新 target model
        if self.current_epoch % self.sync_interval == 0 and batch_idx == 0:
            self.target_model.load_state_dict(self.model.state_dict())

        self.log("loss", loss, on_epoch=True, prog_bar=True)
        return loss

    def configure_optimizers(self):
        return optim.Adam(self.model.parameters(), lr=self.lr)

# ==== 自定義 Callback 來紀錄 loss ====
class LossHistoryCallback(Callback):
    def __init__(self):
        self.losses = []

    def on_train_epoch_end(self, trainer, pl_module):
        loss = trainer.callback_metrics.get("loss")
        if loss is not None:
            self.losses.append(loss.item())
            print(f"Epoch {trainer.current_epoch}, Loss: {loss.item():.4f}")

# ==== 模擬 buffer 並創建 DataLoader ====
buffer = ReplayBuffer(capacity=10000)

for _ in range(2000):  # 探索階段
    game = Gridworld(size=4, mode='random')
    state = game.board.render_np().reshape(64,) + np.random.rand(64)/100.0
    state1 = state.astype(np.float32)
    done = False
```

```
    while not done:
        action = np.random.randint(0, 4)
        game.makeMove(action_set[action])
        next_state = game.board.render_np().reshape(64,) + np.random.rand(64)
        reward = game.reward()
        done = reward != -1
        buffer.push(state1, action, reward, next_state.astype(np.float32), float(do
        state1 = next_state

dataset = ReplayDataset(buffer)
dataloader = DataLoader(dataset, batch_size=128, shuffle=True)

# ==== 訓練 ====
model = DuelingDQNLightning()
loss_callback = LossHistoryCallback()

trainer = pl.Trainer(
    max_epochs=1000,
    accelerator="gpu",
    devices=1,
    callbacks=[loss_callback],
    logger=CSVLogger("logs")   # 不用 tensorboard logger
)

trainer.fit(model, dataloader)

# ==== 繪製 Loss 圖 ====
plt.figure(figsize=(10, 7))
plt.plot(loss_callback.losses, label="Loss")
plt.xlabel("Epochs", fontsize=11)
plt.ylabel("Loss", fontsize=11)
plt.title("Dueling DQN Loss over Epochs")
plt.legend()
plt.show()
```



- test in static mode

```
Initial State:
[['+' '-' ' ' 'P']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 0; Taking action: d
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' 'P']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 1; Taking action: d
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' 'P']
 [' ' ' ' ' ' ' ']]
Move #: 2; Taking action: d
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' 'P']]
Move #: 3; Taking action: d
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' 'P']]
...
Game lost; too many moves.
False
Games played: 1000, # of wins: 0
Win percentage: 0.0%
```

- test in player mode

```
Initial State:
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 ['P' ' ' ' ' ' ']]
Move #: 0; Taking action: d
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 ['P' ' ' ' ' ' ']]
Move #: 1; Taking action: d
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 ['P' ' ' ' ' ' ']]
Move #: 2; Taking action: d
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 ['P' ' ' ' ' ' ']]
Move #: 3; Taking action: d
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 ['P' ' ' ' ' ' ']]
...
Game lost; too many moves.
False
Games played: 1000, # of wins: 0
Win percentage: 0.0%
```

- test in random mode

```
Initial State:
[[' ' ' ' '+' ' ']
 [' ' ' ' '-' ' ']
 ['P' ' ' ' ' 'W']
 [' ' ' ' ' ' ' ']]
Move #: 0; Taking action: d
[[' ' ' ' '+' ' ']
 [' ' ' ' '-' ' ']
 [' ' ' ' ' ' 'W']
 ['P' ' ' ' ' ' ']]
Move #: 1; Taking action: d
[[' ' ' ' '+' ' ']
 [' ' ' ' '-' ' ']
 [' ' ' ' ' ' 'W']
 ['P' ' ' ' ' ' ']]
Move #: 2; Taking action: d
[[' ' ' ' '+' ' ']
 [' ' ' ' '-' ' ']
 [' ' ' ' ' ' 'W']
 ['P' ' ' ' ' ' ']]
Move #: 3; Taking action: d
[[' ' ' ' '+' ' ']
 [' ' ' ' '-' ' ']
 [' ' ' ' ' ' 'W']
 ['P' ' ' ' ' ' ']]
...
Game lost; too many moves.
False
Games played: 1000, # of wins: 107
Win percentage: 10.7%
```

跟剛剛的double DQN有一樣的問題，所以接下來要加上一點training tips。

Dueling DQN for random mode(PyTorch Lightning) with training techniques

```
buffer.buffer.clear()  # 清空 ReplayBuffer 中的所有內容
# ==== 模擬 ReplayBuffer 和 Dataset ====
class ReplayBuffer:
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = []

    def push(self, *transition):
```

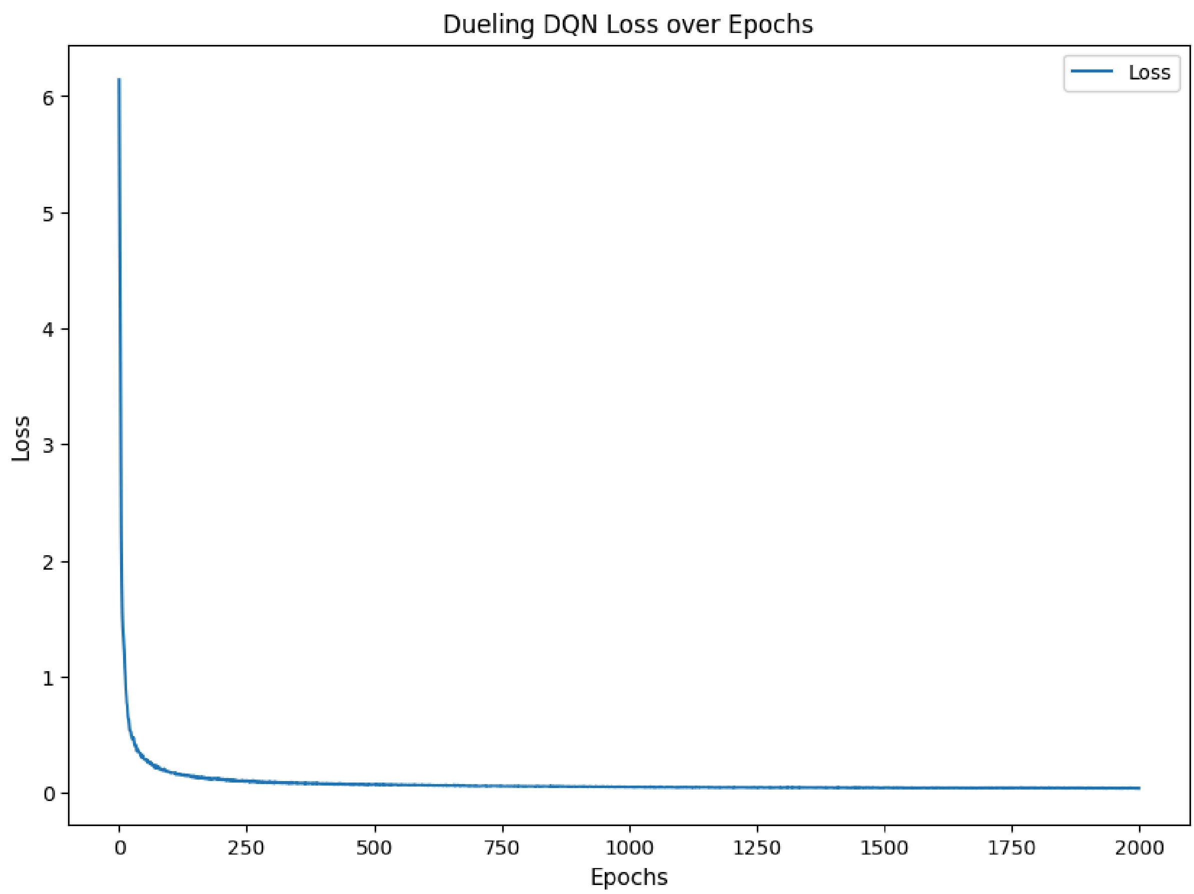
```
        if len(self.buffer) >= self.capacity:
            self.buffer.pop(0)
        self.buffer.append(transition)

    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        return zip(*batch)

    def __len__(self):
        return len(self.buffer)

class ReplayDataset(Dataset):
    def __init__(self, buffer):
        self.buffer = buffer.buffer

    def __len__(self):
        return len(self.buffer)

    def __getitem__(self, idx):
        state, action, reward, next_state, done = self.buffer[idx]
        return (
            torch.tensor(state, dtype=torch.float32),
            torch.tensor(action, dtype=torch.int64),
            torch.tensor(reward, dtype=torch.float32),
            torch.tensor(next_state, dtype=torch.float32),
            torch.tensor(done, dtype=torch.float32),
        )

# ==== Dueling Q-Network ====
class DuelingQNet(nn.Module):
    def __init__(self):
        super(DuelingQNet, self).__init__()
        self.feature = nn.Sequential(
            nn.Linear(64, 256),
            nn.ReLU(),
            nn.BatchNorm1d(256),
            nn.Linear(256, 128),
            nn.ReLU()
```

```
        )
        self.advantage = nn.Linear(128, 4)
        self.value = nn.Linear(128, 1)

    def forward(self, x):
        x = self.feature(x)
        adv = self.advantage(x)
        val = self.value(x).expand(x.size(0), 4)
        return val + adv - adv.mean(dim=1, keepdim=True)

# ==== Lightning 模型 ====
class DuelingDQNLightning(pl.LightningModule):
    def __init__(self, gamma=0.9, lr=1e-3, tau=0.01):
        super().__init__()
        self.q_net = DuelingQNet()
        self.target_net = DuelingQNet()
        self.target_net.load_state_dict(self.q_net.state_dict())
        self.loss_fn = nn.MSELoss()
        self.gamma = gamma
        self.lr = lr
        self.tau = tau

    def forward(self, x):
        return self.q_net(x)

    def training_step(self, batch, batch_idx):
        state, action, reward, next_state, done = batch

        q_values = self.q_net(state)
        q_value = q_values.gather(1, action.unsqueeze(1)).squeeze(1)

        with torch.no_grad():
            next_q_values = self.q_net(next_state)
            next_actions = next_q_values.argmax(dim=1)
            next_q_target = self.target_net(next_state)
            next_q_value = next_q_target.gather(1, next_actions.unsqueeze(1)).squ
            expected_q = reward + self.gamma * next_q_value * (1 - done)
```

```
        loss = self.loss_fn(q_value, expected_q)
        self.log("loss", loss, on_epoch=True, prog_bar=True)

        # Soft update target network
        for param, target_param in zip(self.q_net.parameters(), self.target_net.par
            target_param.data.copy_(self.tau * param.data + (1 - self.tau) * target_p

        return loss

    def configure_optimizers(self):
        return optim.Adam(self.q_net.parameters(), lr=self.lr)

# ==== 自定義 Callback 來紀錄 loss ====
class LossHistoryCallback(Callback):
    def __init__(self):
        self.losses = []

    def on_train_epoch_end(self, trainer, pl_module):
        loss = trainer.callback_metrics.get("loss")
        if loss is not None:
            self.losses.append(loss.item())
            print(f"Epoch {trainer.current_epoch}, Loss: {loss.item():.4f}")

# ==== 模擬 buffer 並創建 DataLoader ====

buffer = ReplayBuffer(capacity=10000)

for ep in range(2000):
    game = Gridworld(size=4, mode='random')
    state = game.board.render_np().reshape(64,) + np.random.rand(64)/100.0
    state1 = state.astype(np.float32)
    done = False
    steps = 0

    while not done:
        action = np.random.randint(0, 4)
        game.makeMove(action_set[action])
        next_state = game.board.render_np().reshape(64,) + np.random.rand(64)
```

```
        # reward shaping
        raw_reward = game.reward()
        player_pos = game.board.components['Player'].pos
        goal_pos = game.board.components['Goal'].pos
        distance = abs(player_pos[0] - goal_pos[0]) + abs(player_pos[1] - goal_p

        shaped_reward = raw_reward + (-0.1 * distance)
        done = (raw_reward == 10 or raw_reward == -10)

        buffer.push(state1, action, shaped_reward, next_state.astype(np.float32),
        state1 = next_state
        steps += 1

    if ep % 200 == 0:
        print(f"Episode {ep} finished with {steps} steps")

# Dataset
dataset = ReplayDataset(buffer)
dataloader = DataLoader(dataset, batch_size=128, shuffle=True)


# ==== 訓練 ====
model = DuelingDQNLightning()
loss_callback = LossHistoryCallback()

trainer = pl.Trainer(
    max_epochs=2000,
    accelerator="gpu",  # 或 "auto"
    devices=1,
    callbacks=[loss_callback],
    logger=False,
    gradient_clip_val=1.0  #  Gradient clipping
)

trainer.fit(model, dataloader)

# ==== 繪製 Loss 圖 ====
```

```
plt.figure(figsize=(10, 7))
plt.plot(loss_callback.losses, label="Loss")
plt.xlabel("Epochs", fontsize=11)
plt.ylabel("Loss", fontsize=11)
plt.title("Dueling DQN Loss over Epochs")
plt.legend()
plt.show()
```



跟原版相比，做了哪些更動

- Dueling Q-Network 架構

```
class DuelingQNet(nn.Module):
    def __init__(self):
        super(DuelingQNet, self).__init__()
        self.feature = nn.Sequential(
            nn.Linear(64, 256),
            nn.ReLU(),
```

```
            nn.BatchNorm1d(256),  #  [3] Batch Normalization
            nn.Linear(256, 128),
            nn.ReLU()
        )
        self.advantage = nn.Linear(128, 4)  #  Advantage branch
        self.value = nn.Linear(128, 1)      #  Value branch

    def forward(self, x):
        x = self.feature(x)
        adv = self.advantage(x)
        val = self.value(x).expand(x.size(0), 4)
        return val + adv - adv.mean(dim=1, keepdim=True)  # Combine V(s) + A(s
```

目的：分開學習「狀態價值」與「行動優勢」使模型學得更穩定，尤其在某些行動影響力不大時特別有效。

- Soft Target Update

```
# Soft update: 讓 target network 緩慢追隨 main network，提升穩定性
for param, target_param in zip(self.q_net.parameters(), self.target_net.parame
    target_param.data.copy_(self.tau * param.data + (1 - self.tau) * target_paran
```

目的：避免目標網路更新過快造成不穩定，讓學習曲線更平滑。

- Batch Normalization

```
# BatchNorm1d: 穩定特徵分佈，改善收斂速度
nn.BatchNorm1d(256),
```

目的：減少 internal covariate shift，提升訓練效率與泛化能力。

- Reward Shaping

```
#  Reward shaping: 額外懲罰與目標的距離，提供更連續的學習信號
raw_reward = game.reward()
player_pos = game.board.components['Player'].pos
goal_pos = game.board.components['Goal'].pos
distance = abs(player_pos[0] - goal_pos[0]) + abs(player_pos[1] - goal_pos[1]

shaped_reward = raw_reward + (-0.1 * distance)  # 額外懲罰距離
```

```
done = (raw_reward == 10 or raw_reward == -10)

buffer.push(state1, action, shaped_reward, next_state.astype(np.float32), float
```

目的：鼓勵智能體靠近目標，即使還沒達成也能獲得學習信號，加快學習速度。

- Gradient Clipping

```
trainer = pl.Trainer(
    max_epochs=2000,
    accelerator="gpu",
    devices=1,
    callbacks=[loss_callback],
    logger=False,
    gradient_clip_val=1.0  #  Clip 梯度避免爆炸
)
```

目的：當梯度過大時截斷，可防止不穩定訓練或數值爆炸。

- test in static mode

```
Initial State:
[['+' '-' ' ' 'P']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 0; Taking action: l
[['+' '-' 'P' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 1; Taking action: l
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 2; Taking action: l
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Game won! Reward: 10
True
Initial State:
[['+' '-' ' ' 'P']
 [' ' 'W' ' ' ' ']
...
Game won! Reward: 10
True
Games played: 1000, # of wins: 1000
Win percentage: 100.0%
```

- test in player mode

```
Initial State:
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 ['P' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 0; Taking action: u
[['+' '-' ' ' ' ']
 ['P' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 1; Taking action: u
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' ' ' ' ']
 [' ' ' ' ' ' ' ']]
Game won! Reward: 10
True
Initial State:
[['+' '-' ' ' ' ']
 [' ' 'W' ' ' ' ']
 [' ' ' ' 'P' ' ']
 [' ' ' ' ' ' ' ']]
Move #: 0; Taking action: u
[['+' '-' ' ' ' ']
 [' ' 'W' 'P' ' ']
...
Game won! Reward: 10
True
Games played: 1000, # of wins: 1000
Win percentage: 100.0%
```

- test in random mode

```
Initial State:
[[' ' ' ' ' ' ' ']
 [' ' ' ' 'P' '+']
 [' ' ' ' ' ' ' ']
 ['-' ' ' 'W' ' ']]
Move #: 0; Taking action: d
[[' ' ' ' ' ' ' ']
 [' ' ' ' ' ' '+']
 [' ' ' ' 'P' ' ']
 ['-' ' ' 'W' ' ']]
Move #: 1; Taking action: u
[[' ' ' ' ' ' ' ']
 [' ' ' ' 'P' '+']
 [' ' ' ' ' ' ' ']
 ['-' ' ' 'W' ' ']]
Move #: 2; Taking action: r
[[' ' ' ' ' ' ' ']
 [' ' ' ' ' ' '+']
 [' ' ' ' ' ' ' ']
 ['-' ' ' 'W' ' ']]
Game won! Reward: 10
True
Initial State:
[[' ' ' ' 'W' ' ']
 ['-' ' ' ' ' 'P']
...
Game lost; too many moves.
False
Games played: 1000, # of wins: 775
Win percentage: 77.5%
```