神經網路訓練手冊

坦克競技場 · 神經網路與應用期中專題｜從資料到上場：模仿學習、網路設定、看懂結果、做實驗｜實驗數字產生於 2026-10-20 22:23

這本手冊講「錄好資料之後」的事：怎麼訓練神經網路、怎麼看結果、怎麼做實驗讓它變強。規則策略怎麼寫，請看《規則策略函數說明書》。手冊裡的程式範例都實際執行過，第 7 節的數字是真的訓練、真的上場打出來的。

1　這一步在做什麼

這是**模仿學習**：規則策略看得到完整狀態，做出的決定當成「答案」；神經網路只拿得到 91 維觀測，要學會「看到這些，就做規則策略會做的事」。

	機器學習的說法	在這個專題
輸入	特徵 x	91 維觀測（射線、看得到的敵人、子彈、自己的狀態）
答案	標籤 y	規則策略在那一步選的 5 個動作
模型	分類器	一個網路、**5 個分類頭**：移動 3 選 1、車身 3 選 1、砲塔 3 選 1、開火 2 選 1、技能 2 選 1
損失	交叉熵	5 個頭的交叉熵加起來
上場	預測	每一步（1/20 秒）輸入觀測，5 個頭各選一個動作

2　資料

繳交網站「② 策略與錄製」下載的 `dataset_….zip`，裡面每一批錄製是一個 `.npz` 檔：

欄位	形狀	內容
`obs`	(筆數, 91)	每一步的觀測（模型的輸入）
`action`	(筆數, 5)	規則策略那一步的動作（答案）
`episode_id`	(筆數,)	第幾局
`t`	(筆數,)	那一局的第幾步
`rank`	(筆數,)	那一局最後的名次（1 最好）

範例：看看資料集裡有什麼

```
import io, zipfile, numpy as np

z = zipfile.ZipFile("dataset.zip")                      # 網站下載的資料集
for name in z.namelist():
    if not name.endswith(".npz"): continue
    d = np.load(io.BytesIO(z.read(name)))
    obs, act, ep, rank = d["obs"], d["action"], d["episode_id"], d["rank"]
    print(f"{name}：{len(np.unique(ep))} 局、{len(act):,} 筆（步）")
    print("  觀測", obs.shape, "動作", act.shape)              # (筆數, 91)、(筆數, 5)
    print("  開火的比例", round(float(act[:, 3].mean()), 3))   # 通常只有 3–5%
    print("  移動：前進 / 後退 / 停", np.bincount(act[:, 0], minlength=3) / len(act))
    print("  每局的名次", np.bincount(rank[np.unique(ep, return_index=True)[1]], minlength=6)
[1:])
```

一局大約 400–600 步，所以 1,000 局就有 50 萬筆左右。手冊實驗用的資料：1,200 局、623,931 筆，規則策略自己的平均名次 2.99、場均擊殺 0.73。

> **注意兩件事**
> ① **開火只佔 3–5% 的步數**（類別不平衡）：模型只要永遠「不開火」，開火準確率就有 95%，但完全沒用。所以 train.py 會給開火、技能的少數類別比較大的權重，看結果時也要看「召回率」（見第 5 節）。
> ② **驗證集要按局切**：同一局相鄰兩步幾乎一樣，如果隨機按筆切，驗證集等於偷看答案，準確率會虛高。train.py 已經按局切好了。

3　網路：agent.py

網路寫在 `agent.py`（課堂 notebook 6-3 會寫出一份 `my_tank/agent.py`，從它開始改）。開頭有兩區設定：**網路設定**決定網路長什麼樣子，**訓練設定**決定 train.py 怎麼訓練它（見第 4 節）。所有旋鈕都在這裡，train.py 本身不用改：

```
# ================= 網路設定（改這裡） ==================
MODEL = "mlp"       # "mlp"：前饋網路（可疊加最近幾步）；"lstm"：有記憶的遞迴網路
STACK = 4           # 只有 mlp 用：一次看最近幾步（1＝只看當下）
HIDDEN = 128        # 每層幾個神經元（寬度，不是層數）：越大越能學細節，也越慢
LAYERS = 1          # 只有 lstm 用：LSTM 疊幾層（1～4），每層各有自己的記憶
SAMPLE = False      # 比賽時怎麼選動作：False＝選分數最高的；True＝依機率抽

# ================= 訓練設定（改這裡，train.py 會讀） ==================
EPOCHS = 8          # 訓練幾輪（整份資料看幾遍）
LR = 0.001          # 學習率：太大會不穩，太小學很慢
BATCH = 1024        # 每次拿幾筆算 loss、更新一次權重（MLP 才用；LSTM 固定每次 64 段）
SEED = None         # 亂數種子：None＝每次隨機（建議）；要重現某次結果，填訓練時印出的數字
FIRE_WEIGHT = None # 開火類加重幾倍（開火很少見，加重才學得會）；None＝依資料比例自動決定
```

設定	意思	建議
`MODEL`	`"mlp"` 每一步獨立判斷（可疊加最近幾步）；`"lstm"` 有記憶，記得前面發生什麼	先用 MLP，跑順了再試 LSTM

`STACK`	把最近幾步的觀測接在一起餵給 MLP（輸入變成 91×STACK 個數字），讓它看得出移動方向	4（見第 7 節實驗）
`HIDDEN`	每一層有幾個神經元，越大越能學複雜的規則，但更容易過擬合、推論變慢	128–256
`SAMPLE`	上場時，True＝照機率抽一個動作，False＝選機率最高的	兩種都試試

網路本體有兩種：`MLP`（前饋網路）和 `LSTMNet`（有記憶），`build_model()` 依 `MODEL` 選一個。兩者每一行的意思、參數、資料形狀，見書末**附錄 A（MLP）**和**附錄 B（LSTM）**。

```
def build_model():
    return LSTMNet() if RECURRENT else MLP()
```

LSTM 每一步會更新記憶。敵人躲到牆後、從觀測裡消失時，MLP 就「忘了」他，LSTM 還記得大概在哪。代價是訓練比較慢、比較難調。

範例：自己改網路結構

把 agent.py 裡的 `class MLP` 換成下面這段，就變成 3 層、加 Dropout 的網路。只要 `forward` 的輸入輸出格式不變（輸入觀測、回傳 5 組 logits 和記憶），train.py 和 export_model.py 都不用改：

```
class MLP(nn.Module):
    """改成 3 層、加 Dropout（訓練時隨機關掉一些神經元，減少過擬合）"""
    def __init__(self):
        super().__init__()
        self.shared = nn.Sequential(
            nn.Linear(91 * STACK, HIDDEN), nn.ReLU(), nn.Dropout(0.1),
            nn.Linear(HIDDEN, HIDDEN), nn.ReLU(), nn.Dropout(0.1),
            nn.Linear(HIDDEN, HIDDEN), nn.ReLU())
        self.heads = nn.ModuleList([nn.Linear(HIDDEN, n) for n in HEADS])

    def forward(self, x, memory=None):
        z = self.shared(x)
        return [head(z) for head in self.heads], None
```

4　訓練：train.py

```
python train.py dataset_….zip my_tank/agent.py
python train.py 第一批.zip,第二批.zip my_tank/agent.py      # 多個檔案用逗號接
python train.py dataset.zip my_tank/agent.py --epochs 3     # 臨時覆蓋 agent.py 的設定（快速試跑）
```

訓練設定寫在 agent.py 的「訓練設定」區；沒寫的用預設值。優先順序：指令列參數 > agent.py > 預設值。這裡的 `my_tank/` 是課堂 notebook 的資料夾；自己的電腦上，就在放 agent.py、train.py 的那個資料夾執行 `python train.py dataset.zip agent.py`。每一輪會印出驗證集的移動準確率與開火召回率。

agent.py	指令列	預設	意思
`EPOCHS`	`--epochs`	15	整份訓練資料看幾遍（輪）；網站的 agent.py 範本設 8
`LR`	`--lr`	0.001	學習率（Adam），太大會不穩、太小學很慢
`BATCH`	`--batch`	1024	MLP 每次拿幾筆算 loss、更新一次權重（LSTM 固定每次 64 段）

`SEED`	`--seed`	隨機	亂數種子；None 就隨機並印出來。想重現某次結果再填那個數字（全班用同一個數字，初始權重會一樣，抄襲比對可能誤判）
`FIRE_WEIGHT`	`--fire-weight`	自動	開火類別的 loss 加重幾倍；None＝依資料比例自動（開火約 4% 時約 5 倍），會印出實際用的權重
	`--val`	0.15	拿幾成的局當驗證集
	`--seq-len`、`--burn`	64、16	LSTM 才用：每局切成 64 步一段，前 16 步只用來暖身記憶、不算損失

train.py 依序做這些事：按局切出驗證集 → 算訓練資料每一維的平均和標準差、做**正規化**（讓 91 個數字的大小差不多）→ 疊加觀測（STACK）→ 給開火、技能的少數類別加權重 → 用 Adam 每批 1,024 筆訓練，印出每一輪的 loss → 在驗證集上算準確率。輸出兩個檔案到 agent.py 同一個資料夾：`model.pt`（權重＋正規化參數）和 `metrics.json`（驗證結果）。

5　看懂訓練結果

指標	意思	怎麼看
`acc_move`、`acc_body`、`acc_turret`	移動、車身、砲塔：模型和規則策略選一樣動作的比例	越高越像規則策略；車身、砲塔通常比移動難
`acc_fire`、`acc_skill`	開火、技能的準確率	**不要只看這個**：類別不平衡，永遠不開火也有 95%
`recall_fire`	召回率：規則策略開火的時候，模型也開火的比例	太低＝該開火時不開
`precision_fire`	精確率：模型開火的時候，規則策略也開火的比例	太低＝亂開火（浪費裝填時間）
`recall_skill`、`precision_skill`	技能的召回率、精確率	被動模組沒有意義
`acc_turret_no_enemy_visible`	看不到敵人時，砲塔轉向猜對的比例	MLP 通常較低，LSTM 有記憶會比較好

```
import json
m = json.load(open("my_tank/metrics.json"))
for k in ("acc_move", "acc_body", "acc_turret", "acc_fire", "acc_skill"):
    print(f"{k:12s} {m[k]:.1%}")
print("開火：召回率", f"{m['recall_fire']:.1%}", "精確率", f"{m['precision_fire']:.1%}")
```

loss 應該一輪比一輪低；如果驗證準確率不升反降，可能是過擬合（網路太大、資料太少），試著多錄資料或減少輪數。

6　匯出與上傳

```
python export_model.py my_tank/agent.py          # → my_tank/submit/model.pt（上傳這個）
```

- 匯出會把網路、正規化參數、STACK、SAMPLE 等設定包成一個檔案，並自動比對：同樣的觀測，匯出前後選的動作必須完全相同。
- 同一個資料夾的 `strategy.py` 和 `agent.py` 會一起打包進去，老師看程式評分用，比賽時不會執行。

- 比賽時由老師的程式載入 model.pt，自動處理正規化、疊加觀測、LSTM 記憶與抽樣，每一步推論要在 10 ms 以內（網路別太大）。
- 上傳的是 `submit/model.pt`，不是 train.py 存的 `model.pt`（那是權重檔，網站會擋下）。

7　做實驗：實測結果

改進的方法是**一次改一件事**、其他條件不變，比較前後的上場成績。下表是同一份資料（1,200 局，配裝 標準＋衝刺，規則策略用範例 C 改過參數的版本）， 每個設定用同一個亂數種子訓練 **10 輪**（「只訓練 3 輪」那列除外），上場和 4 台範例 bot 打 200 場，用天梯的計分方式算分：

設定	訓練	移動準確	砲塔準確	開火召回	每場得分	10 分制	名次	擊殺
MLP、只看當下、HIDDEN 64（範例 C 的設定）	11 秒	90.9%	88.7%	99.0%	**0.52**	**2.6**	3.61	0.34
MLP、疊加 4 步	13 秒	92.2%	87.5%	97.3%	**0.71**	**3.6**	3.1	0.47
MLP、疊加 4 步、HIDDEN 256	18 秒	92.3%	88.0%	97.2%	**0.80**	**4.0**	3.33	0.56
LSTM、HIDDEN 128	41 秒	92.1%	87.5%	98.1%	**0.39**	**1.9**	3.62	0.22
同「疊加 4 步、HIDDEN 256」，資料只有 200 局	4 秒	87.6%	83.5%	78.2%	**0.49**	**2.4**	3.08	0.29
同「疊加 4 步、HIDDEN 256」，只用名次 1–2 的局	10 秒	92.2%	86.2%	96.4%	**0.52**	**2.6**	3.35	0.32
同「疊加 4 步、HIDDEN 256」，只訓練 3 輪	7 秒	91.7%	86.3%	97.5%	**0.50**	**2.5**	3.15	0.3
同「疊加 4 步、HIDDEN 256」，SAMPLE=True（依機率抽樣）	18 秒	92.3%	88.0%	97.2%	**0.85**	**4.2**	3.09	0.59

怎麼讀這張表

- 每個設定上場打 200 場。每場得分本身有運氣成分，兩個設定的差距要大於約 ±0.18 才算真的不一樣，下面的「明顯」都是照這個標準判斷。
- **疊加 4 步（vs 只看當下）**：0.52 → 0.71 分，明顯比較好。單一步看不出自己和敵人往哪裡移動，疊幾步就看得出來。
- **網路加大 HIDDEN 64 → 256**：0.71 → 0.80 分，差不多（差距在誤差內）。
- **資料量 200 → 1,200 局**：0.49 → 0.80 分，明顯比較好。資料少時連離線準確率都比較低（移動 87.6% vs 92.3%）。
- **訓練輪數 3 → 10 輪**：0.50 → 0.80 分，明顯比較好。
- **只用名次 1–2 的局（剩 479 局）**：0.80 → 0.52 分，明顯比較差。篩選後資料品質變好，但數量也變少，這次是數量的影響比較大；篩選要和資料量一起考慮。
- **SAMPLE=True（vs 取最大）**：0.80 → 0.85 分，差不多（差距在誤差內）。抽樣讓動作有隨機性，有時能避免卡在同一個動作。
- **LSTM（vs 疊加 4 步的 MLP）**：離線準確率幾乎一樣（移動 92.1% vs 92.3%、砲塔 87.5% vs 88.0%），上場卻 0.39 vs 0.80 分，明顯比較差。LSTM 訓練也比較久（41 秒 vs 18 秒），需要更多資料和調整才會發揮。
- **最重要的一點：離線準確率高，不代表上場強**。上面的 LSTM 就是例子。模型自己開時，一點小偏差會把坦克帶到規則策略沒去過的狀況，越錯越多（分布偏移）。所以比較設定時，最後一定要看上場成績。

報告的「訓練結果」可以照這個方式寫：改了什麼、打幾局、前後數字多少、結論是什麼。上場成績用天梯（每 30 分鐘更新）最準；想自己先比較，可以用網站錄製時「規則策略的平均名次」比較策略版本，用離線指標粗略比較網路設定。

8　常見問題

現象	可能原因與改法
loss 不下降	學習率太大（agent.py 試 `LR = 0.0003`）；或資料太少
驗證準確率高，上場很差	分布偏移（見第 7 節最後一點）：多錄不同情境的資料、試疊加觀測或 LSTM、試 SAMPLE=True
模型從來不開火	看 recall_fire；資料裡規則策略是不是本來就很少開火（看第 2 節的範例輸出）
上傳說推論太慢	HIDDEN 或 STACK 太大，縮小網路
上傳說不是匯出的檔案	上傳了 train.py 存的 model.pt，要上傳 submit/model.pt
改了 agent.py 設定，匯出失敗	改網路設定後要重新訓練，舊的 model.pt 和新網路形狀對不上

9　進階方向

- **資料篩選**：只用打得好的局訓練，或把好局複製多份（樣本加權）。範例：

```
import io, zipfile, numpy as np

z = zipfile.ZipFile("dataset.zip")
parts = [np.load(io.BytesIO(z.read(n))) for n in z.namelist() if n.endswith(".npz")]
for k, d in enumerate(parts):
    good = d["rank"] <= 2                                  # 只留名次 1、2 的局（rank 是那一局最後的名次）
    np.savez_compressed(f"good_{k}.npz", **{key: d[key][good] for key in d.files})
    print(f"第 {k + 1} 批：{len(np.unique(d['episode_id']))} 局 → 留下 {len(np.unique(d['episode_id'][good]))} 局")
# 訓練時用逗號接起來：python train.py good_0.npz,good_1.npz my_tank/agent.py
```

- **資料多樣性**：錄製時換地圖、換對手配裝、10 台模式，讓模型看過更多狀況。
- **改規則策略**：模型最多學到和規則策略一樣好；策略本身變強，模型的上限才會提高。
- **DAgger**：讓模型自己開，記下它遇到的狀況，再請規則策略標答案、加進資料重新訓練，專治分布偏移（需要自己寫，屬於挑戰題）。
- **輔助任務**：多加一個回歸頭預測敵人下一步的位置，幫網路學到更好的特徵（要改 agent.py 和 train.py）。

附錄 A　MLP 逐行說明

`import torch.nn as nn`：載入 PyTorch 的神經網路模組，`nn` 是簡稱。網路的「零件」（全連接層、激活函數、LSTM…）都在裡面。下面是 agent.py 的 `MLP`：

```
class MLP(nn.Module):
    """前饋網路：輸入是最近 STACK 步接起來的 91×STACK 個數字"""
    def __init__(self):
        super().__init__()
        self.shared = nn.Sequential(nn.Linear(91 * STACK, HIDDEN), nn.ReLU(),
                                    nn.Linear(HIDDEN, HIDDEN), nn.ReLU())
        self.heads = nn.ModuleList([nn.Linear(HIDDEN, n) for n in HEADS])   # 5 個分類頭

    def forward(self, x, memory=None):      # x：(筆數, 91×STACK)
        z = self.shared(x)
        return [head(z) for head in self.heads], None       # MLP 沒有記憶，回傳 None
```

寫法	參數	意思
`class MLP(nn.Module)`		所有網路都繼承 `nn.Module`：`__init__` 建零件，`forward` 定義資料怎麼流過這些零件。`super().__init__()` 一定要寫，PyTorch 才能登記零件裡的權重
`nn.Linear(in, out)`	in：輸入幾個數字 out：輸出幾個數字	全連接層：**y = Wx + b**。W 是 out×in 的權重矩陣、b 是 out 個偏差，都是訓練要學的參數，共 in×out＋out 個
`nn.ReLU()`	沒有參數	激活函數 max(0, x)：負的變 0。沒有它，兩層 Linear 疊起來仍然只是一個 Linear，學不了複雜的規則
`nn.Sequential(…)`	依序放零件	把零件串成一條：資料從第一個流到最後一個。`self.shared` 是 5 個頭共用的身體
`nn.ModuleList([…])`	一串零件	放 5 個分類頭。不能用一般的 list：PyTorch 不會登記 list 裡的權重，訓練時就不會更新
`[nn.Linear(HIDDEN, n) for n in HEADS]`	n＝3、3、3、2、2	每個頭輸出 n 個**分數**（logits）。分數經過 softmax 變成機率，最高的就是網路的選擇；訓練時 `cross_entropy` 直接吃 logits
`forward(self, x, memory=None)`	x：觀測 memory：MLP 用不到	回傳（5 組 logits, None）。MLP 沒有記憶所以回傳 None；這個格式 train.py、export_model.py 都要用，不能改

資料形狀與參數數量（STACK＝4、HIDDEN＝128，一批 1,024 筆）

經過	輸出形狀	參數數量
輸入：最近 4 步的觀測接起來	(1024, 364)	
`nn.Linear(364, 128)` ＋ ReLU	(1024, 128)	46,720
`nn.Linear(128, 128)` ＋ ReLU	(1024, 128)	16,512

5 個頭	(1024, 3)、(1024, 3)、(1024, 3)、(1024, 2)、(1024, 2)	1,677
合計		**64,909**

第一層的參數最多：輸入有 364 個數字，每個都要和 128 個神經元連線。所以 STACK 加大，第一層會跟著變大；HIDDEN 加倍，第二層會變成約 4 倍。

附錄 B　LSTM 逐行說明

LSTM 是**有記憶的遞迴網路**：一局裡每一步都把「記憶」傳給下一步，所以不用疊加觀測（STACK 固定 1）。下面是 agent.py 的 `LSTMNet`：

```
class LSTMNet(nn.Module):
    """先把 91 個數字變成 HIDDEN 個特徵，再進 LSTM：每一步都帶著上一步留下的記憶"""
    def __init__(self):
        super().__init__()
        self.encoder = nn.Sequential(nn.Linear(91, HIDDEN), nn.ReLU())
        self.lstm = nn.LSTM(HIDDEN, HIDDEN, num_layers=LAYERS, batch_first=True)   # 疊 LAYERS
層：下一層吃上一層每一步的輸出
        self.heads = nn.ModuleList([nn.Linear(HIDDEN, n) for n in HEADS])

    def forward(self, x, memory=None):      # x：(筆數, 時間長度, 91)；memory：上一步留下的記憶
        z, memory = self.lstm(self.encoder(x), memory)
        return [head(z) for head in self.heads], memory
```

寫法	參數	意思
`self.encoder`	`nn.Linear(91, HIDDEN)` ＋ReLU	先把 91 個觀測轉成 HIDDEN 個特徵，再交給 LSTM
`nn.LSTM(input_size, hidden_size, batch_first=True)`	input_size：每一步輸入幾個數字（這裡＝HIDDEN） hidden_size：記憶有多大（這裡＝HIDDEN） batch_first：True 表示形狀是 (筆數, 時間, 特徵)	LSTM 本體，只有一層（`num_layers` 預設 1）。每一步讀進這一步的特徵和上一步的記憶，輸出這一步的結果和新的記憶
`memory`	(h, c) 兩個張量，形狀都是 (1, 筆數, HIDDEN)	**h**：短期狀態，也是每一步的輸出；**c**：長期的「細胞狀態」，負責記得比較久的事。傳 None 表示全部從 0 開始（每局開始時）
LSTM 裡的 3 個門	自動學，不用設定	**遺忘門**：舊記憶留多少；**輸入門**：這一步的新資訊寫進多少；**輸出門**：這一步要輸出多少。LSTM 的參數就是這些門（加上候選記憶）的權重，所以大約是同大小 Linear 的 4 倍
`forward(self, x, memory=None)`	x：(筆數, 時間長度, 91)	回傳（5 組 logits, 新的記憶）。比賽時一次餵一步，把回傳的記憶帶到下一步

資料形狀與參數數量（HIDDEN＝128，訓練時一批 64 段、每段 64 步）

經過	輸出形狀	參數數量
輸入：一段連續的觀測	(64, 64, 91)	
`encoder`：`nn.Linear(91, 128)` ＋ ReLU	(64, 64, 128)	11,776

`nn.LSTM(128, 128)`	每一步的輸出 (64, 64, 128)；記憶 h、c 各 (1, 64, 128)	132,096
5 個頭（每一步都輸出）	(64, 64, 3)、(64, 64, 3)、(64, 64, 3)、(64, 64, 2)、(64, 64, 2)	1,677
合計		**145,549**

怎麼訓練：train.py 把每一局切成 64 步一段（`--seq-len`），前 16 步（`--burn`）只用來讓記憶「暖身」、不算 loss，再一段一段訓練。**比賽時**：每局開始記憶清成 0，之後一次餵一步、帶著記憶往下走，這些由老師的程式自動處理。

和 MLP 比：LSTM 的參數多約 2.2 倍（和附錄 A 的 MLP 比），訓練也比較慢；第 7 節的實驗中它的離線準確率和 MLP 差不多，上場卻比較差，需要更多資料和調整。