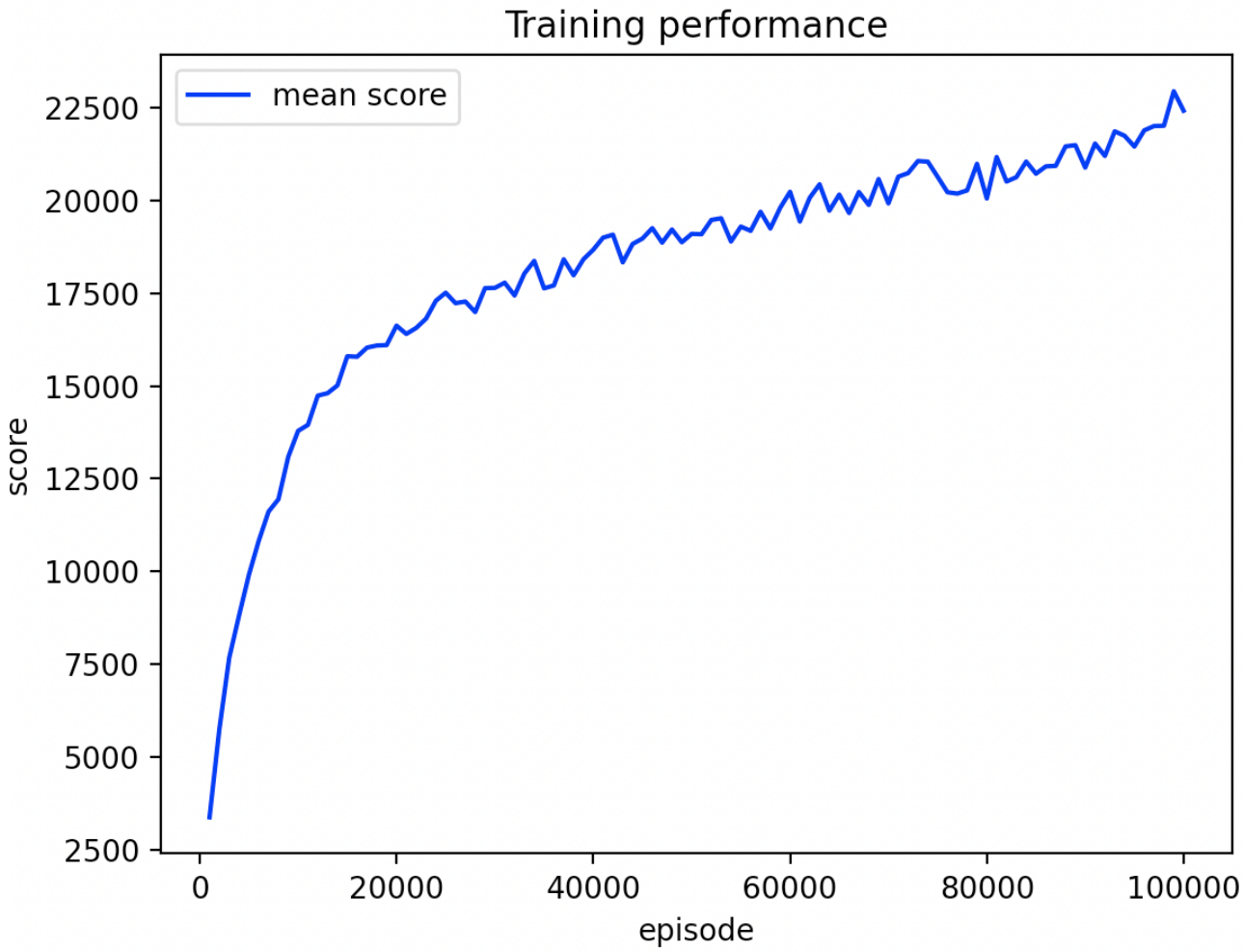
Plot



Bonus

1. Describe the implementation and the usage of n-tuple network. 2048共16個格子，若將所有格子的排列組合當作一個表學習，state太多記憶體肯定無法負荷，n-tuple network 的想法是不去考慮整個 16 格棋盤，而是選定其中n個格子，把這些格子組合起來形成一個「pattern」。每一個 pattern 就對應一個「lookup table (LUT)」，裡面存學到的權重值。棋盤的value為多個 pattern 的估值加總。 實作時 pattern 還會考慮到旋轉跟鏡射，同樣形狀的 pattern 不管轉幾次或對稱過，都會共用權重，這樣可以等於一次學到很多盤面，效果比較好也比較快收斂。比如如果我們挑一個 6 格的 pattern，那就會有 16^6個可能狀態，每個都對應到一個權重值。當遊戲跑的時候，只要去查對應 LUT 的 entry 就可以拿到估值。
2. Explain the mechanism of TD(0). 公式：

$$V(s) \leftarrow V(s) + \alpha(r + V(s'') - V(s))$$

   會將下一次折扣後的state value及當前的reward列入考慮，然後減掉目前的state value後拿到誤差來修正目前state value
3. Describe your implementation in detail including action selection and TD-backup diagram.

- Action selection 在 `select_best_move()` 中，會依序測試四個方向（上、下、左、右），對於能夠成功移動的動作，計算

$$Q(s,a) = R(s,a) + V(\text{after-state})$$

  最後挑選出`Q(s,a)` 最大的action作為最佳決策。

- TD(0) Backup 在 `update_episode()` 中，我從遊戲結束的最後一個 state 開始往前回溯。對每個 before-state `s`，先計算目前的`V(s)`。 接著用 bootstrap value（下一個 state 的目標值）計算 TD error： `δ=bootstrap−V(s)`。 然後用`α·δ` 來更新這個 state 的權重。 最後將 bootstrap 更新成：`bootstrap=R(s)+V_new(s)` 。這樣能確保了每個 state 的 value 會往「實際 reward + next state expectation」的方向靠近。