ATTIVITÀ PROGETTUALE FONDAMENTI DI INTELLIGENZA ARTIFICIALE M

SVILUPPO DI UNA RETE NEURALE PER IL GIOCO FORZA 4 E CONFRONTO CON L'ALGORITMO MINMAX

Indice

Introduzione **2**
Capitolo 1: introduzione ed implementazione di forza 4 ..... **3**
1.1 implementazione base del gioco 3
Capitolo 2: tipi di giocatore **5**
2.1 Human 5
2.2 AI_RANDOM 5
2.3 AI_MINMAX 6
2.4 AI_NN 6
2.5 AI_NN_NON_DET 7
Capitolo 3: Training della rete neurale **8**
3.1 Dataset 8
3.2 Training 9
Capitolo 4: Risultati delle partite **10**
4.1 AI_RANDOM vs AI_RANDOM 10
4.2 AI_RANDOM vs AI_MINMAX 10
4.3 AI_RANDOM vs AI_NN 11
4.4 AI_RANDOM vs AI_NN_NON_DET 13
Capitolo 5: Conclusioni **15**
Bibliografia **16**

Introduzione

L'obiettivo di questa attività progettuale è stato quello di realizzare una rete neurale in grado di giocare a Forza 4 e confrontarne i risultati ottenuti con un'altra tecnica di intelligenza artificiale: l'algoritmo Minmax.

La scelta di questo gioco è basata sul fatto che esso possiede perfettamente le proprietà necessarie per essere catalogato come tale per l'intelligenza artificiale:

- Gioco per due giocatori
- Le mosse si alternano
- Un giocatore vince e uno perde (o al limite si conclude in pareggio)
- Gioco con conoscenza perfetta poiché entrambi i giocatori possiedono la stessa informazione (non ci sono elementi nascosti come avviene, ad esempio, nei giochi classici con delle carte in cui un giocatore non può sapere cosa ha in mano l'avversario).

La relazione verrà strutturata nel seguente modo:

- Nel primo capitolo verrà fatta una breve introduzione sul gioco e l'implementazione della struttura base in Python
- Nel secondo capitolo verranno spiegate le varie tipologie di giocatore e la loro realizzazione
- Nel terzo capitolo si discuterà dei dataset creati e del training della rete neurale
- Nel quarto ed ultimo capitolo verranno mostrati e commentati i risultati ottenuti nella sperimentazione tra le varie tecniche.

Capitolo 1: introduzione ed implementazione di Forza 4

Forza 4 (Connected 4 o Four in a Row in inglese) è un gioco da tavolo prodotto dalla Milton Bradley nel 1974 per 2 giocatori i quali, a turno, fanno cadere una pedina (rispettivamente di colore rosso e giallo nella versione originale) in una griglia verticale formata da 6 righe e 7 colonne: la vittoria si ottiene nel momento in cui si forma una linea di 4 pezzi dello stesso colore in orizzontale, verticale o diagonale.
La peculiarità di forza 4 è la posizione verticale, e non orizzontale, della tavola: inserendo la pedina in una delle 7 colonne, la gravità farà cadere il pezzo nella cella più in basso disponibile. Questa caratteristica impone che, nonostante il numero di posizioni possibili sia uguale a 42, il numero massimo di mosse valide sia pari a 7 (il numero delle colonne).
Nella modalità classica, sono possibili 4,531,985,219,092 configurazioni della griglia raggiungibili da quella iniziale[1], il che rende la complessità dello spazio di ricerca pari a 10^{14}[2].

Nel corso degli anni sono state create alcune varianti che differiscono per il numero di pedine consecutive necessarie per vincere la partita, per la possibilità di cambiare la configurazione della tavola di gioco durante l'incontro o per la possibilità di effettuare due turni consecutivi senza la possibilità però di effettuare una mossa vincente.

Un altro aspetto fondamentale di Forza 4 è la sua appartenenza alla categoria dei giochi risolti[2], giochi per cui si può predire l'esito della partita da una qualsiasi configurazione della tavola nel caso in cui i due giocatori giochino perfettamente, ovvero quando un giocatore agisce con la miglior mossa possibile, indipendentemente dalla risposta dell'avversario. Questa caratteristica implica che il primo giocatore, giocando perfettamente, potrà sempre vincere al massimo alla 41-esima mossa, indipendentemente dalla posizione in cui inserirà le pedine il secondo giocatore. La prima persona a risolvere il gioco è stata James D. Allen il primo Ottobre 1988. [3]

1.1 Implementazione base del gioco

La struttura base del gioco è stata divisa in due classi distinte: game.py e controller.py:
- in game.py sono presenti tute le funzioni necessarie alla creazione e modifica della tavola, alla verifica correttezza della mossa scelta e della vittoria
- in controller.py ci sono le funzioni che permettono di effettuare una o più partite (indicando l'esito), una che crea i due giocatori (che possono essere di diverso tipo, come verrà spiegato in maniera più approfondita nel secondo capitolo) e quelle che effettuano un turno di gioco in base alla tipologia del giocatore.

In particolare, in game.py le funzioni create sono le seguenti:

- `creat_board(self)`: crea la griglia formata da 6 righe e 7 colonne
- `create_board_history(self)`: crea la lista in cui verrà inserita l'evoluzione della tavola nel corso della partita
- `get_board_history(self)`: restituisce la lista precedente
- `print_board(self, board)`: permette di stampare la tavola di gioco con le colonne numerate e "simula la gravità" poiché altrimenti la colonna 0 sarà quella graficamente in alto, andando contro la logica del gioco
- `drop(self, board, col, piece)`: inserimento di una pedina nella colonna col
- `add_board_history(self, board)`: aggiorna la board history con la tavola passata come argomento
- `is_valid(self, board, col)`: verifica la validità di una mossa, ovvero controlla se la colonna passata come argomento è interamente occupata o meno
- `get_next_open_row(self, board, col)`: restituisce la prima riga vuota della colonna passata come argomento, necessario per inserire la pedina nella posizione corretta
- `winning_move(self, board, piece)`: verifica se è presente una sequenza di 4 pedine uguali consecutive in orizzontale, verticale o diagonale
- `get_valid_locations(self, board)`: restituisce l'insieme delle mosse valide, ovvero l'insieme di colonne non piene.

Per quanto riguarda controller.py, le funzioni inizialmente create per il corretto svolgimento del gioco sono le seguenti:

- `choose_players(self)`: permette di creare i due giocatori scegliendo tra umano, random, Minmax, rete neurale, rete neurale non deterministica
- `initialize_games(self)`: inizializza tutto il programma, permettendo di scegliere tra varie opzioni, tra cui il numero di partite che i due giocatori effettueranno
- `play_single_game(self, player1, player2, showPrint)`: viene svolta una singola partita, alternando il turno fra i giocatori e restituendo il risultato della partita (0 in caso di pareggio, 1 in caso di vittoria del primo giocatore e 2 in caso di vittoria del secondo giocatore)
- `play_multiple_games(self, player1, player2, n_games, showPrint)`: simula il numero di partite passato come argomento restituendo i risultati totali, ovvero numero e percentuale di vittorie dei singoli giocatori o degli incontri finiti in pareggio.

Capitolo 2: Tipi di giocatore

Definita la struttura e le funzioni principali di Forza 4, passiamo ora ai diversi tipi di giocatore creati:

- `HUMAN`: una persona sceglie la colonna in cui inserire la pedina da linea di comando. Creato per testare personalmente i risultati ottenuti con le altre tipologie di giocatore
- `AI_RANDOM`: scelta casuale tra le mosse valide, senza nessun tipo di indicazioni
- `AI_MINMAX`: sfrutta la potenza dell'algoritmo Minmax per decidere la mossa migliore da compiere
- `AI_NN`: la scelta della mossa viene fatta in base alle previsioni fatte da una rete neurale. Nei paragrafi successivi verrà spiegata più nel dettaglio
- `AI_NN_NON_DET`: a differenza del tipo precedente, in questo caso la scelta non è deterministica ma viene effettuata dando un peso ad ogni possibile colonna. Anche questa tipologia di giocatore verrà spiegata dettagliatamente nei paragrafi successivi.

2.1 HUMAN

L'implementazione è definita tramite la funzione `human_turn(self, board, n_playerm showPrint)` in controller.py: viene chiesto al giocatore, tramite terminale, di inserire il numero della colonna in cui desidera posizionare la pedina. A questo punto vengono fatti due controlli:

- se la colonna scelta è nel range 0-`N_COLS` con `N_COLS` numero di colonne della tavola di gioco
- se la colonna scelta è una mossa valida (ovvero se la colonna non è piena)

In caso di esito positivo ad entrambi i controlli, la pedina viene inserita nella tavola.

2.2 AI_RANDOM

Questo tipo di giocatore è stato creato inizialmente per testare varie funzioni (principalmente la simulazione di più partite), poi per creare i dataset per effettuare il training della rete neurale ed infine per verificare il livello l'intelligenza dei vari giocatori, considerando quello della `AI_RANDOM` il peggiore.
Tramite la funzione `randomAI_turn(self, board, n_player, showPrint)`, la scelta viene fatta in modo completamente casuale tra le mosse disponibili: questa decisione è dipesa dal fatto che, in caso di partite con quasi tutte le colonne occupate, la ricerca di una posizione valida impiegava diverso tempo.

2.3 AI_MINMAX

L'algoritmo Minmax[4] è un algoritmo ricorsivo di intelligenza artificiale usato per la ricerca della mossa migliore in un gioco che si svolge tra due giocatori (min e max), il quale analizza l'albero di gioco a partire dai nodi terminali e risalendo progressivamente fino alla posizione corrente dei giocatori.

L'implementazione del giocatore `AI_MINIMAX` è stata divisa in due parti:
- in minimax.py è stato creato l'algoritmo e le funzioni necessarie per il corretto funzionamento di esso
- In controller.py è stata definita la funzione `minmaxAI_turn(self, board, n_player, showPrint)` che effettua la mossa per il giocatore, scegliendo la colonna in base al risultato dell'algoritmo.

Nella classe minimax sono state definite quattro funzioni:
- `evaluate_window(self, window, piece)`: valuta un intervallo di 4 celle della tavola ed assegna un punteggio ai casi in cui ci siano 4 pedine consecutive (caso di vittoria), 3 pedine e uno spazio vuoto, 2 pedine e 2 spazi vuoti e il caso in cui l'avversario abbia 3 pedine e uno spazio vuoto (situazione di possibile vittoria dell'avversario, da evitare)
- `score_position(self, board, piece)`: calcolo complessivo del punteggio, analizzando la tavola passata come argomento
- `is_terminal_node(self, board)`: verifica i nodi terminali della tavola passata come argomento
- `minmax(self, board, depth, maxPlayer)`: implementazione dell'algoritmo il quale restituisce la colonna selezionata ed il relativo punteggio assegnato. La colonna verrà poi restituita da `mimnaxAI_turn` in controller.py

Tramite la variabile `DEPTH_MINMAX` è possibile indicare di quanto guarda avanti l'algoritmo per cercare la soluzione migliore: per lo scopo di questa attività progettuale, il limite massimo testato è stato 3 poiché, già con questa impostazione, è difficile competere con l'algoritmo mentre, con una profondità pari a 4, i tempi per calcolare la mossa migliore crescono esponenzialmente.

2.4 AI_NN

Le reti neurali[5] sono ispirate alla biologia ed utilizzano una metodologia di elaborazione dei dati simili a quelle che i neuroni effettuano quotidianamente: un modello di rete neurale impara dalle esperienze e, dato un input, è capace di prendere decisioni.
Ora è necessario identificare alcune caratteristiche e applicarle allo scopo dell'attività progettuale: la struttura di una rete neurale si divide in strati (layer) dove obbligatoriamente

sono presenti quello di input, quello di output e solitamente un numero di strati nascosti. Nell'attività progettuale svolta, la rete neurale è stata sviluppata con Keras[6] e l'input scelto consiste in 42 valori (le celle della tavola di gioco) mentre l'output consiste in 3 valori, ovvero la probabilità che, dato l'ingresso, la partita si concluda in pareggio, con una vittoria del primo giocatore o con una vittoria del secondo giocatore. Oltre a definire gli input e gli output, è stato necessario definire i vari livelli nascosti, le funzioni di attivazione di ogni livello e l'optimizer della rete e, dopo vari tentativi, è risultato avere i risultati migliori (valutando l'accuracy) la seguente configurazione:

- 1 Input layer: 42 valori in input, funzione di attivazione relu
- 5 Hidden layer: 42 valori in input, funzione di attivazione relu
- 1 output layer: 3 valori di output, funzione di attivazione softmax
- Optimizer: Adam

La scelta della mossa da parte della rete neurale è stata implementata tramite la funzione `neuralNetworkAI_turn(self, board, n_player, showPrint)` in controller.py nel seguente modo: la rete predice la probabilità di vittoria (in base al giocatore che deve effettuare la mossa) per ogni colonna libera e sceglie, in modo deterministico, quella che restituisce la probabilità maggiore.

Nella sezione 3 verrà spiegato nel dettaglio come è stato effettuato il training della rete per fare in modo che essa possa scegliere correttamente la mossa migliore.

2.5 AI_NN_NON_DET

La rete neurale non deterministica è stata realizzata in quanto è possibile che alcune mosse abbiamo una percentuale simile di vittoria ma potrebbe accadere che, quella con probabilità inferiore, risulti migliore in termini assoluti. La logica è stata implementata tramite la funzione `neuralNetworkAI_nonDeterministic_turn(self, board, n_player, showPrint)` in controller.py la quale, similmente alla precedente, predice la probabilità di vittoria di un singolo giocatore ma, a differenza della precedente che selezionava la colonna con probabilità maggiore, effettua una scelta casuale tra le mosse disponibili moltiplicate per un peso proporzionale alla probabilità di vittoria.

Per chiarire meglio la scelta che la rete neurale non deterministica effettuerà, ipotizziamo che siano disponibili le colonne 1, 3, 4, e 6 con probabilità di vittoria rispettivamente di 45%, 40%, 35% e 35%. Per decidere la mossa da effettuare, verrà scelto casualmente un valore tra 1 e 155 in cui:

- nel range 0-44 verrà scelta la colonna 1
- nel range 45-84 verrà scelta la colonna 2
- nel range 85-120 verrà scelta la colonna 3
- nel range 121-155 verrà scelta la colonna 4

Capitolo 3: Training della rete neurale

Dopo aver definito la struttura della rete neurale, è stato necessario effettuare il training della stessa in modo tale che possa predire correttamente la mossa migliore. Come paradigma di apprendimento è stato deciso di adottare quello supervisionato[7], ovvero l'apprendimento è basato su diversi dataset contenenti simulazioni di partite tra due giocatori, variando sia il numero di partite che le tipologie di giocatore.

3.1 Dataset

I dataset sono file .csv e sono strutturati in modo tale che, in ogni riga, ci sia il risultato della partita e un elemento della board history: in questo modo sono presenti tutte le mosse di tutte le partite. I dataset creati sono i seguenti:

- 100k.csv: dataset con 100.000 partite giocate tra due giocatori di tipo `AI_RANDOM`, le quali hanno coperto 2.106.701 posizioni, di cui 1.449.706 posizioni univoche (percentuale posizioni totali coperte pari al 0,00003%)
- 500k.csv: dataset con 500.000 partite giocate tra due giocatori di tipo `AI_RANDOM`, le quali hanno coperto 10.566.958 posizioni, di cui 6.547.815 posizioni univoche (percentuale posizioni totali coperte pari al 0,00014%)
- 2000k.csv: dataset con 2.000.000 partite giocate tra due giocatori di tipo `AI_RANDOM`, le quali hanno coperto 42.260.660 posizioni, di cui 23.544.744 posizioni univoche (percentuale posizioni totali coperte pari al 0,00052%)
- 20k_minmax.csv: dataset con 20.000 partite (tra una rete neurale addestrata e un giocatore di tipo `AI_MINMAX` con `DEPTH_MINMAX = 1`). Sono state coperte 970.000 posizioni, di cui 86 univoche.

Considerando le posizioni totali e quelle univoche, è necessario analizzare la percentuale di esse: nei dataset generati con `AI_RANDOM`, il numero di configurazioni della tavola non ripetute diminuisce con l'aumentare delle partite simulate (è più probabile incontrare stati già "visti" se il numero di partite aumenta), ma si ottengono percentuali superiori al 50-60% del totale. Nel caso del dataset creato con `AI_MINMAX` e `AI_NN` le posizioni vengono quasi sempre ripetute: questo è dato dal fatto che l'algoritmo Minmax tende a ripetere le stesse mosse con la conseguenza di avere una probabilità molto alta di creare stati della tavola identici. Tutti i dataset creati sono stati salvati nell'omonima cartella.

3.2 Training

Il dataset utilizzato è stato diviso nei due sottoinsiemi train e test per valutare le perormances della rete neurale tramite la procedura train-test split con la divisione 80/20 (80% dei dati faranno parte del sottoinsieme train e il 20% del sottoinsieme test).
Per effettuare l'addestramento, è stato necessario decidere sia la batch size (numero di esempi che verrano propagati nella rete) e il numero di epochs (numero di volte l'intero dataset vene passato attraverso la rete neurale): per quanto riguarda la prima, sono stati fatti diversi tentativi cercando di minimizzare la loss e massimizzare la accuracy mentre il numero di epochs è stato deciso in maniera automatica, impostando la patience a 5 (numero di epochs senza miglioramenti dopo di cui il training viene fermato), per evitare di continuare un addestramento che non portava a significativi progressi.

Inizialmente è stato effettuato il training con i dataset originali ma successivamente, dopo alcune considerazioni riguardo gli stati, sono stati ridotti considerando solamente le configurazioni univoche della tavola. Il ragionamento è stato il seguente: da una configurazione, sono possibili diverse sequenze di mosse per vincere la partita ma, essendo forza 4 un gioco risolto, esisterà una successione migliore delle altre per cui, anche per evitare confusione nell'apprendimento della rete, è necessario eliminare dal dataset le configurazioni doppie.
I risultati ottenuti con i dataset ridotti (e mostrati al termine del capitolo) confermano quanto scritto sopra dal momento che, effettuando nuovamente il training della rete, i valori della loss e dell'accuracy sono leggermente migliorati.

Una volta addestrata la rete, per evitare di effettuare ogni volta il training di essa, i modelli ottenuti sono stato salvati tramite `model.save(MODEL_NAME)`: ognuno di essi è memorizzato in un file con estensione .h5 nella directory 'modelSaved' ed il nome è espresso come: model-(nome dataset)_(batch size)bs_[noDupl in caso di configurazioni doppie].h5. Per recuperare il modello desiderato, si utilizza la funzione `load_load(MODEL_NAME)`. I risultati sono riassunti nella tabella sottostate:

Model name	Epochs	Loss	Accuracy	Val_loss	Val_accuracy
model-100k_256bs.h5	83	0,6550	0,6170	0,6616	0,6116
model-100k_256bs_noDupl.h5	78	0,6520	0,6211	0,6642	0,6107
model-500k_512bs.h5	51	0,6619	0,6120	0,6626	0,6117
model-500k_512bs_noDupl.h5	120	0,6642	0,6234	0,6559	0,6223
model-2000k_512bs.h5	35	0,6600	0,6135	0,6616	0,6125
model-2000k_512bs_noDupl.h5	43	0,6521	0,6247	0,6504	0,6260
model-20k_minmax_64bs.h5	17	0,0144	0,9895	0,0140	0,9897
model-20k_minmax_64bs_noDupl.h5	10	0,7998	0,6801	0,9146	0,4444

Capitolo 4: Risultati delle partite

Dopo aver descritto l'implementazione di Forza 4, le varie tipologie di giocatore create e la struttura della rete neurale, sono state simulate migliaia di partite tra le varie IA e sono stati raccolti ed analizzati i risultati.

4.1 `AI_RANDOM` vs `AI_RANDOM`

La prima analisi è stata fatta considerando due giocatori che scelgono le mosse in modo casuale tra quelle disponibili, ottenendo comunque degli esiti che vale la pena commentare. Sono state simulate 5000 partite con i seguenti risultati:

Numero partite	Vittorie primo giocatore	Vittorie secondo giocatore	Pareggi	Numero medio di mosse per partita
5000	2760 (55,2%)	2232 (44,64%)	8 (0,16%)	20,98

Come si può notare, il primo giocatore ha vinto un numero considerevole (528, pari al 10,56%) di partite in più: questo dato può sembrare anomalo inizialmente poiché, utilizzando la stessa strategia, i giocatori dovrebbero vincere un numero di partite molto simile. Questa differenza è data dal vantaggio che il primo giocatore possiede potendo inserire prima la pedina nella tavola: in questo caso, è leggermente più probabile che il primo giocatore completi una sequenza vincente prima che lo faccia il suo avversario. Questa caratteristica influirà anche i risultati successivi.

Inoltre risalta la bassissima percentuale di pareggi: questo dato è conseguente al fatto delle migliaia di combinazioni possibili per vincere una partita, rendono abbastanza improbabile occupare tutte le posizioni della tavola senza mai ottenere una sequenza vincente giocando in modo casuale. La media delle mosse necessarie per terminare una partita è di circa 21 pedine inserite.

4.2 `AI_RANDOM` vs `AI_MINMAX`

Ora analizziamo la potenza dell'algoritmo Minmax: verranno analizzati i risultati sia nel caso in la `AI_MINMAX` giochi come primo e secondo giocatore sfidando un avversario di tipo `AI_RANDOM`. Inoltre verrà modificato il parametro `DEPTH_MINMAX` analizzando come esso influisca sul risultato delle partite.

Nella tabella sottostante vengono riportati i risultati ottenuti dalle partite simulate nel caso in cui AI_MINMAX giochi per primo al variare di DEPTH_MINMAX:

DEPTH_MINMAX	Numero partite	Vittorie IA Minmax	Vittorie IA Random	Pareggi	Numero medio di mosse per partita
1	1000	995 (99,5%)	1 (0,1%)	4 (0,4%)	12,3
2	1000	999 (99,9%)	1 (0,1%)	0 (0%)	11,01
3	500*	500 (100%)	0 (0%)	0 (0%)	10,88

* sono state effettuate meno partite a causa del tempo necessario per effettuare una mossa con DEPTH_MINMAX = 3

Nel caso in cui AI_MINMAX giochi per secondo, i risultati ottenuti sono i seguenti:

DEPTH_MINMAX	Numero partite	Vittorie IA Minmax	Vittorie IA Random	Pareggi	Numero medio di mosse per partita
1	1000	940 (94%)	60 (6%)	0 (0%)	10,61
2	1000	995 (99,5%)	5 (0,5%)	0 (0%)	13,44
3	500*	999 (99,9%)	1 (0,1%)	0 (0%)	13,52

* sono state effettuate meno partite a causa del tempo necessario per effettuare una mossa con DEPTH_MINMAX = 3

I risultati indicano che il giocatore che sceglie le mosse in base all'algoritmo Minmax è in grado di vincere quasi tutte le partite (con un minimo del 94%) e ne vince un numero sempre maggiore con l'aumentare di DEPTH_MINMAX. Inoltre si può notare che, in caso giochi per primo, come avveniva nel paragrafo 4.1, vinca un numero maggiore di partite rispetto al caso in cui giochi per secondo.

Considerando infine il numero medio di pedine necessarie per terminare la partita, si possono notare due dati distinti: il primo è che la media è nettamente inferiore al caso del paragrafo 4.1, con un intervallo di confidenza ridotto e con un numero di partite terminate quasi immediatamente crescente al crescere della DEPTH_MINMAX (tra le 7 e le 8 mosse nel caso in cui AI_MINMAX giochi, rispettivamente, come primo o secondo giocatore). Il secondo dato è che, quando AI_MINMAX gioca per secondo, il numero di pedine inserite cresce al crescere della DEPTH_MINMAX: questo è dato dal fatto che ha più probabilità di dover interrompere una potenziale sequenza vincente del suo avversario e, con l'aumentare della DEPTH_MINMAX, è in grado di interrompere più sequenze a discapito però del numero di mosse necessarie per vincere.

4.3 AI_RANDOM vs AI_NN

Successivamente sono state simulate le partite tra la rete neurale addestrata con i diversi dataset e un giocatore di tipo AI_RANDOM. Come nella sezione 4.2, sono stati analizzati sia i

casi in cui `IA_NN` giochi come primo e secondo giocatore, analizzando anche la media di mosse per terminare una partita simulandone 500 per ogni modello.
Nel caso in cui la rete neurale giochi per primo, i risultati ottenuti sono i seguenti:

Model name	Numero partite	Vittorie Rete Neurale	Vittorie IA Random	Pareggi	Media mosse per partita
model-100k_128bs.h5	500	340 (68%)	158 (31,6%)	2 (0,4%)	21,9
model-100k_128bs_noDupl.h5	500	338 (67,6%)	160 (32%)	2 (0,4%)	22,4
model-500k_512bs.h5	500	285 (57%)	215 (43%)	0 (0%)	21,85
model-500k_512bs_noDupl.h5	500	317 (63,4%)	183 (36,6%)	0 (0%)	21,82
model-2000k_512bs.h5	500	293 (58,6%)	207 (41,4%)	0 (0%)	19,17
model-2000k_512bs_noDupl.h5	500	318 (63,6%)	182 (36,4%)	0 (0%)	21,29
model-20k_minmax_64bs.h5	500	268 (53,6%)	229 (45,8%)	3 (0,6%)	21,42
model-20k_minmax_64bs_noDupl.h5	500	287 (57,4%)	210 (42%)	3 (0,6%)	21,05

Come si può notare, la rete neurale vince sempre la maggior parte delle partite, con casi in cui supera anche abbondantemente il 60% delle vittorie. Confrontando i risultati con i casi delle sezioni 4.1 e 4.2, si può notare che le percentuali generalmente sono simili al caso in cui il giocatore sia di tipo randomico: questo deriva dal fatto che i dataset usati hanno coperto una percentuale bassissima rispetto alle possibili combinazioni della tavola, portando la rete ad avere una bassa confidenza ed a giocare in maniera più simile ad un giocatore randomico. Analizzando inoltre i valori della predict, solo in alcuni casi le colonne restituiscono un probabilità maggiore rispetto ad altre, il che significa che la rete ha imparato alcune mosse ma, nell'interezza della partita, queste mosse influiscono solo lievemente sul risultato finale.

Nel caso in cui la rete neurale giochi come secondo giocatore, i risultati ottenuto sono:

Model name	Numero partite	Vittorie Rete Neurale	Vittorie IA Random	Pareggi	Media mosse per partita
model-100k_128bs.h5	500	250 (50%)	248 (49,6%)	2 (0,4%)	20,03
model-100k_128bs_noDupl.h5	500	259 (51,8%)	240 (48%)	1 (0,2%)	21,57
model-500k_512bs.h5	500	283 (56,6%)	217 (43,4%)	0 (0%)	19,93
model-500k_512bs_noDupl.h5	500	259 (51,8%)	241 (48,2%)	0 (0%)	20,81
model-2000k_512bs.h5	500	316 (64,6%)	177 (35,4%)	0 (0%)	21,47
model-2000k_512bs_noDupl.h5	500	269 (53,8%)	231 (46,2%)	0 (0%)	19,78
model-20k_minmax_64bs.h5	500	269 (53,8%)	230 (46%)	1 (0,2%)	18,64
model-20k_minmax_64bs_noDupl.h5	500	242 (48,4%)	258 (51,6%)	0 (0%)	21,64

In questo caso otteniamo comunque la vittoria della rete neurale nella maggior parte delle partite giocate, ma il divario tra vittorie e sconfitte diminuisce rispetto al caso in cui giochi come primo giocatore: questo è conseguenza del vantaggio nel giocare per primo descritto precedentemente. Giocando per secondo, la rete neurale riesce a superare questo vantaggio ma senza eccellere nei risultati. Unico risultato anomalo è quello generato dal dataset 20k-minmax senza i duplicati: in questo caso il numero di dati è talmente piccolo che la rete non apprende quasi nulla e ottiene i risultati che otterrebbe un giocatore di tipo `AI_RANDOM`.

Analizzando infine il numero medio di mosse necessarie per terminare la partita, si può notare che se la rete neurale gioca come secondo giocatore, generalmente sono necessarie 1 o 2 pedine in meno.

4.4 `AI_RANDOM` vs `AI_NN_NON_DET`

L'ultima analisi effettuata è stata quella delle performances della rete neurale non deterministica nel caso in cui essa giochi contro un tipo di giocatore `AI_RANDOM`. Anche in questo caso sono state simulate 500 partite per ogni modello generato, analizzando sia il caso in cui il giocatore di tipo `AI_NN_NON_DET` giochi per primo o per secondo e la media di mosse necessarie per terminare la partita.

I risultati ottenuti se la rete neurale non deterministica gioca come primo giocatore sono i seguenti:

Model name	Numero partite	Vittorie Rete Neurale	Vittorie IA Random	Pareggi	Media mosse per partita
model-100k_128bs.h5	500	271 (54,4%)	229 (45,8%)	0 (0%)	20,8
model-100k_128bs_noDupl.h5	500	277 (55,4%)	223 (44,6%)	0 (0%)	21,25
model-500k_512bs.h5	500	273 (54,6%)	226 (45,2%)	1 (0,2%)	22,12
model-500k_512bs_noDupl.h5	500	274 (54,8%)	224 (44,8%)	2 (0,4%)	21,19
model-2000k_512bs.h5	500	265 (53%)	235 (47%)	0 (0%)	21,11
model-2000k_512bs_noDupl.h5	500	265 (53%)	233 (46,6%)	2 (0,4%)	20,88
model-20k_minmax_64bs.h5	500	273 (54,6%)	227 (45,4%)	0 (0%)	20,91
model-20k_minmax_64bs_noDupl.h5	500	286 (57,2%)	213 (42,6%)	1 (0,2%)	21,05

Invece, se la rete neurale gioca come secondo giocatore, i risultati ottenuti sono:

Model name	Numero partite	Vittorie Rete Neurale	Vittorie IA Random	Pareggi	Media mosse per partita
model-100k_128bs.h5	500	223 (44,6%)	277 (55,4%)	0 (0%)	20,92
model-100k_128bs_noDupl.h5	500	234 (46,8%)	265 (53%)	1 (0,2%)	21,15
model-500k_512bs.h5	500	213 (42,6%)	286 (57,2%)	1 (0,2%)	20,58
model-500k_512bs_noDupl.h5	500	209 (41,8%)	290 (58%)	1 (0,2%)	20,7
model-2000k_512bs.h5	500	240 (48%)	260 (52%)	0 (0%)	21,19
model-2000k_512bs_noDupl.h5	500	209 (41,8%)	290 (58%)	1 (0,2%)	20,4
model-20k_minmax_64bs.h5	500	230 (46%)	270 (54%)	0 (0%)	20,91
model-20k_minmax_64bs_noDupl.h5	500	218 (43,6%)	279 (55,8%)	3 (0,6%)	21,71

I risultati ottenuti ricalcano quelli del paragrafo 4.1: riprendendo quanto detto nella sezione 4.3, la confidenza tra le varie mosse è bassa e quindi, usando una rete non deterministica, la percentuale di scegliere una colonna invece che un'altra è quasi identica ed, aggiungendo quindi questo fattore randomico, si ottengono risultati simili al caso in cui si sfidino due giocatori ti tipo `AI_RANDOM`.

Capitolo 5: Conclusioni

Addestrare una rete neurale per il gioco Forza 4 richiede tempo e risorse notevoli dal momento che, seppur il gioco risulti semplice, sono necessarie centinaia di ore (se non settimane o mesi) per coprire e fare il training di tutte le possibili configurazioni. Nel caso specifico di questa attività progettuale, ogni ora si generavano circa 400.000 partite tra giocatori di tipo `AI_RANDOM` ed erano necessarie diverse ore per addestrare la rete neurale con il risultato di ottenere una rete in grado a malapena di vincere contro un giocatore che sceglie le mosse in modo casuale. Con le risorse e il tempo necessario disponibile, si potrebbe essere in grado di addestrare una rete perfetta in modo da soddisfare la condizione di giochi risolti, ovvero di riuscire a vincere sempre se essa gioca come primo giocatore: probabilmente questa il motivo che sono stati fatti pochi studi su Forza 4 applicando delle reti neurali. I progetti pubblicati in rete sono principalmente inerenti allo studio e all'applicazione delle reti neurali più che a trovare la soluzione applicando le reti neurali.[8] [9][10][11]

I possibili sviluppi futuri di questa attività progettuale possono essere vari: si possono usare dataset maggiormente completi e verificare il livello di apprendimento della rete, riducendo la confidenza delle mosse ed analizzando il comportamento della rete neurale non deterministica. Inoltre è possibile creare in modo alternativo la rete neurale utilizzata (cambiando il numero di hidden layers, le funzioni di attivazione dei vari strati o l'optimizer) oppure addestrarne diverse ed usarle contemporaneamente per decidere quale sia la mossa migliore tramite varie metodologie come, ad esempio, una votazione maggioritaria, una in cui a turno una rete diversa decide quale sia la scelta migliore oppure cambiare la rete incaricata a scegliere la mossa migliore in base ai risultati ottenuti dalle partite.

Bibliografia

[1] John Tromp (2012). “Number of legal 7 X 6 Connect-Four positions after n plies”: https://oeis.org/A212693

[2] M.J.H. Heule, L.J.M. Rothkrantz. “Solving games, Dependence of applicable solving procedures”: https://www.cs.utexas.edu/~marijn/publications/solving_games.pdf

[3] “John's Connect Four Playground”: https://tromp.github.io/c4/c4.html

[4] Mello, P. “Soluzione di giochi”, Materiale didattico del corso di Fondamenti di Intelligenza Artificiale M, Facoltà di Ingegneria, A.A. 2020-2021

[5] Michael Nielsen (2019). “Using neural nets to recognize handwritten digits”: http://neuralnetworksanddeeplearning.com/chap1.html

[6] Jason Brownlee (2020). “TensorFlow 2 Tutorial: Get Started in Deep Learning With tf.keras”: https://www.tensorflow.org/datasets/keras_example

[7] Jason Brownlee (2016). “Supervised and Unsupervised Machine Learning Algorithms”: https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/

[8] Anthony Young (2018). “AZFour: Connect Four Powered by the AlphaZero Algorithm”: https://medium.com/@sleepsonthefloor/azfour-a-connect-four-webapp-powered-by-the-alphazero-algorithm-d0c82d6f3ae9

[9] Rob Dawson (2020). “Learning to play Connect 4 with Deep Reinforcement Learning ”: https://codebox.net/pages/connect4

[10] Luke Kim, Hormazd Godrej, Chi Trung Nguyen. “Applying Machine Learning to Connect Four”: http://cs229.stanford.edu/proj2019aut/data/assignment_308832_raw/26646701.pdf

[11] Marvin Oliver Schneider, João Luís Garcia Rosa. “Neural Connect 4 – A Connectionist Approach to the Game” https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.11.5368&rep=rep1&type=pdf