Università degli Studi di Firenze
Scuola di Ingegneria - Dipartimento di Ingegneria dell'Informazione

Tesi di Laurea Magistrale in Ingegneria Informatica

DeepCrawl: Deep Reinforcement Learning per lo Sviluppo di Videogiochi Strategici Turn-based

Candidato
Alessandro Sestini

Relatori
Prof. Andrew D. Bagdanov
Prof. Marco Bertini

Correlatore
Dott. Alexander Kuhnle

Anno Accademico 2017-2018

Indice

Elenco delle figure **iv**

Elenco delle tabelle **v**

1 Introduzione **1**

2 Lavori Correlati **5**

2.1 Machine Learning e Deep Learning 6

2.2 Reinforcement Learning 11

2.2.1 Processo Decisionale di Markov 12

2.2.2 Funzione di Reward 14

2.2.3 Policy 15

2.2.4 Metodi Value Function 16

2.2.5 Metodi Policy Search 18

2.3 Deep Reinforcement Learning 20

2.3.1 Deep Q-Learning 21

2.3.2 Proximal Policy Optimization 23

2.4 AI nei Videogiochi 26

2.4.1 Finite State Machine 26

2.4.2 Behavior Tree 27

3 Game Design e Desiderata **32**
3.1 Tipologia di Gioco 32
3.2 Meccaniche di Gameplay 35
3.2.1 Attori 36
3.2.2 Ambiente 36
3.2.3 Spazio delle Azioni 38
3.3 Desiderata 41

4 Modello Proposto **45**
4.1 Rete Neurale 46
4.2 PPO e iperparametri 52
4.3 Funzione di Reward 53
4.4 Set-Up di Allenamento 54
4.5 Parametri di Allenamento 57
4.6 Complessità: Rete e Addestramento 58

5 DeepCrawl: Prototipo **60**
5.1 Tecnologie Usate 60
5.2 Progresso e Modalità di Gioco 64
5.3 Creazione del Dungeon 66
5.4 Interfaccia 69

6 Risultati **71**
6.1 Risultati Quantitativi 71
6.2 Risultati Qualitativi 78
6.2.1 Comportamento degli Agenti 80
6.2.2 Interfaccia e Implementazione 83

7 Conclusioni **86**

Bibliografia **91**

Elenco delle figure

2.1 Esempio di un Multi Layer Perceptron 7
2.2 Grafico della funzione ReLu 8
2.3 Immagine di un layer convoluzionale 10
2.4 Schema di una rete neurale ricorrente 11
2.5 Schema dell'interazione tra agente ed ambiente in un problema di RL 12
2.6 Schema della rete convoluzionale usata da DeepMind 22
2.7 Grafici della funzione di loss del PPO 25
2.8 Schema di esempio di una FSM 28
2.9 Schema di esempio di un BT 30

3.1 Differenze tra generi di videogiochi con Doom e Final Fantasy Tactics 34
3.2 Uno screenshot ASCII-based rappresentativo di Rogue 36
3.3 Mock-up iniziale e screenshot della versione finale del gioco . . 37
3.4 Esempio schematico di turno di un giocatore 40
3.5 Esempio di due esperimenti fallimentari delle prime fasi in relazione agli obiettivi preposti 43

4.1 Rete Neurale proposta 47
4.2 Rappresentazione delle osservazioni in input alla rete 50

4.3 Esempio di curriculum learning per un problema generale di DRL 55

5.1 Esempio di ECS relativo a DeepCrawl 63
5.2 Screenshot del gioco che mostra il funzionamento dell'altare . 65
5.3 Esempio di creazione del dungeon 67
5.4 Screenshot di una partita avanzata 68
5.5 Dettaglio degli elementi dell'interfaccia di gioco 69

6.1 Risultati dell'addestramento della classe guerriero 74
6.2 Risultati dell'addestramento della classe arciere 75
6.3 Risultati dell'addestramento della classe ranger 76
6.4 Differenza tra addestramento con e senza curriculum 77
6.5 Esempio di comportamento della classe guerriero 81
6.6 Esempio di comportamento della classe arciere 82
6.7 Esempio di oggetto non propriamente visualizzato 84

Elenco delle tabelle

6.1 Tabella riassuntiva dei risultati del test SeQ di DeepCrawl . . 79

Capitolo 1

Introduzione

Negli ultimi decenni l'industria videoludica ha subito una costante crescita: attualmente rappresenta uno dei settori economici in più rapida evoluzione, in grado di competere con le tradizionali industrie di entertainment e software. Ad oggi i giochi per PC e console hanno raggiunto lo status di mercato di massa grazie ai miglioramenti e alla diffusione di tecniche utili allo sviluppo; i prodotti ludici sono ormai disponibili in varie tipologie di formati, utilizzabili su piattaforme diverse accessibili a qualunque utente, da console casalinghe a diversi dispositivi mobile. Questo settore ha contribuito anche al progresso tecnologico di molti aspetti che possono essere applicati anche in altri ambiti: basti pensare al recente utilizzo delle GPU nel mondo scientifico, oppure all'uso della Virtual/Augmented Reality e ad innovativi metodi di controllo oggi disponibili in molti contesti diversi.

La crescente importanza dell'industria videoludica ha mosso gli sviluppatori a migliorare sempre più i componenti di un videogioco, dall'aspetto visivo continuamente più realistico a meccaniche di gioco in stabile evoluzione, passando a controlli sempre più innovativi e rivoluzionari. Tuttavia la definizione del comportamento dei personaggi non giocanti rappresenta

ancora oggi una grande sfida. L'intelligenza artificiale che muove gli attori è un elemento fondamentale per la buona riuscita di un prodotto di questo genere, e tutt'oggi la ricerca è molto attenta nel trovare delle soluzioni che rendano migliore questo aspetto: è infatti possibile trovare molteplici esempi di sistemi che non agiscono bene all'interno del proprio contesto e che portano inevitabilmente ad un'esperienza ludica non ottimale. Il problema è causato soprattutto dalle tecniche classiche che ancora oggi vengono utilizzate, come i Behavior Trees o le Finite State Machines: queste permettono di definire delle strategie che sono fortemente influenzate dal programmatore, altamente prevedibili, scarsamente dinamiche e nella maggior parte dei casi poco sfidanti.

I recenti sviluppi in ambito Deep Learning hanno contribuito alla crescita di una branca che si presta molto bene a problemi di questo tipo, il Deep Reinforcement Learning: tramite questi algoritmi è possibile allenare delle reti neurali che, attraverso l'interazione con un ambiente, riescono a comandare degli agenti con delle strategie atte a risolvere il contesto in cui sono immersi. Queste tecniche permettono agli attori di estrapolare autonomamente la giusta tattica di comportamento tramite un meccanismo di *trial-and-error*. Il primo esempio di applicazione di questi metodi in ambito videoludico è stato effettuato da DeepMind nel 2015: tramite l'addestramento di una rete neurale i ricercatori sono riusciti a creare un agente perfettamente funzionante in grado di giocare ad una grande quantità di prodotti per la console Atari 2600, in alcuni casi superando le capacità umane. In questo caso l'ambiente è rappresentato dai vari titoli disponibili, le osservazioni sono descritte dai frame di gioco mentre lo spazio delle azioni è delineato semplicemente da tutte le possibili combinazioni di tasti del game controller. Ad ogni passo, l'agente esegue una particolare mossa che viene ricompensata a seconda dello score

ottenuto ad ogni frame, e l'obiettivo dell'agente è quello di massimizzare il possibile punteggio che può ottenere.

I risultati ottenuti da DeepMind sono stati notevoli: attraverso la definizione di una rete neurale sono riusciti a creare un sistema artificiale in grado di estrarre le giuste informazioni dai frame di gioco e completare i vari ambienti con capacità che superano quelle umane. Questo successo ha portato i ricercatori a porre grande interesse nel Deep Reinforcement Learning, creando degli algoritmi sempre più complessi anche in ambiti diversi da quello videoludico; la continua ricerca dimostra quanto margine di miglioramento esista ancora in questo contesto. La quasi totalità degli esperimenti condotti finora però prevede la creazione di agenti intelligenti che sostituiscano gli utenti in giochi già esistenti, per lo più datati e dalle semplici meccaniche, cercando di definire dei sistemi con capacità "super-umane" che spesso hanno un comportamento facilmente percepibile come innaturale e macchinoso: non è questo il compito di un'intelligenza artificiale che deve comandare i personaggi non giocanti all'interno di un videogioco. Essi devono avere un comportamento credibile, intelligente ma al di sotto delle abilità umane: il giocatore deve avere sempre la possibilità di vincere. La scarsità di modelli in questo senso dunque non ha ancora avviato l'utilizzo del Deep Reinforcement Learning in ambito produttivo.

Lo scopo di questa tesi è stato quindi capire se queste tecniche possono essere utilizzate per creare delle intelligenze artificiali adatte allo sviluppo ludico: di conseguenza l'obiettivo è stato ideare un videogioco in cui i nemici costruiti dovevano saper giocare contro degli utenti umani e avere un comportamento credibile, vario e percepibile come intelligente. È stato fondamentale però limitare sufficientemente gli agenti in modo da risultare battibili, altrimenti il gioco sarebbe risultato troppo frustrante da portare a

termine.

La tesi è strutturata nel seguente modo:

- nel capitolo 2 verrà fornita una panoramica del Machine Learning e del Deep Reinforcement Learning, descrivendo nel dettaglio i due algoritmi principali per questo lavoro, il Deep Q-Learning e il Proximal Policy Optimization. Verranno inoltre analizzate brevemente le tecniche classiche di sviluppo di intelligenze artificiali nei videogiochi;

- nel capitolo 3 verrà descritto il contesto di gioco e le meccaniche di gameplay progettate appositamente per questo problema. Verranno inoltre definiti dei requisiti che il sistema ha dovuto soddisfare in modo da poter essere considerato un prodotto ludico;

- nel capitolo 4 verrà analizzato nel dettaglio il modello di Deep Reinforcement Learning definito da questo lavoro, con particolare attenzione alla struttura della rete neurale, agli iperparametri dell'algoritmo e al set-up di training;

- nel capitolo 5 verrà descritta l'implementazione di DeepCrawl, un prototipo perfettamente funzionante di videogioco con degli agenti comandati completamente dagli algoritmi precedentemente definiti;

- nel capitolo 6 verranno riassunti i risultati ottenuti attraverso un test di giocabilità svolto, per misurare la qualità finale del prototipo e dell'intelligenza artificiale;

- nel capitolo 7 verrà interpretato il successo del lavoro proposto e verranno consigliati degli eventuali sviluppi futuri che possono completare e migliorare il lavoro qui iniziato.

Capitolo 2

Lavori Correlati

In questo capitolo verranno introdotti i concetti principali del *machine learning* e del *deep reinforcement learning*. La sezione ha l'obiettivo di fornire al lettore le conoscenze fondamentali per poter capire appieno le tecniche usate in questo lavoro. Verrà brevemente descritto il contesto generale del machine learning e come il deep learning ha profondamente cambiato l'approccio ad esso. In seguito, verrà posta attenzione su un particolare ramo del machine learning, il reinforcement learning e la sua evoluzione deep: verranno esaminati tutti gli elementi che definiscono questa branca e verranno quindi approfondite le diverse tipologie di algoritmi usati nel Deep Reinforcement Learning. In questa sezione verranno presentati i due utili per il continuo della tesi, il Deep Q-Learning (DQN) e Proximal Policy Optimization (PPO). Infine, verranno descritte brevemente le due tecniche classiche principali utilizzate ancora oggi per definire delle intelligenze artificiali all'interno di videogiochi.

2.1 Machine Learning e Deep Learning

Con *machine learning* si intende un insieme di metodi che permettono a un sistema di apprendere da esempi o interazioni con un ambiente senza seguire uno specifico insieme di regole. Il programmatore imposta il calcolatore in modo che sia esso stesso a determinare una funzione f che per ogni input di ingresso fornisca l'output desiderato.

Possiamo distinguere tre tipi di approccio all'apprendimento automatico:

- *Supervised learning*: in questa tipologia esiste una forte distinzione tra input e output; la macchina viene addestrata a risolvere un problema di classificazione o regressione su un insieme di dati etichettati con il rispettivo output, che quindi può essere usata successivamente su nuovi input;

- *Unsupervised learning*: non troviamo nessuna informazione sugli elementi di input, ma il sistema dovrà trovare autonomamente uno schema sui dati;

- *Reinforcement learning*: in questo caso la supervisione è fatta in un secondo momento, in quanto viene data informazione sulle conseguenze di azioni che sono già state prese. Tratta di come gli agenti devono prendere delle decisioni all'interno di un ambiente per massimizzare le ricompense cumulative (e ritardate nel tempo).

Grazie ai recenti sviluppi nell'ambito del *deep learning*, i precedenti approcci hanno subito un drastico miglioramento soprattutto quando si ha a che fare con dati ad alta dimensionalità come immagini, video o sequenze temporali. Con l'espressione deep learning si fa riferimento a delle tecniche e algoritmi afferenti alla branca dell'intelligenza artificiale che cercano di replicare il funzionamento delle reti neurali biologiche; i modello matematici

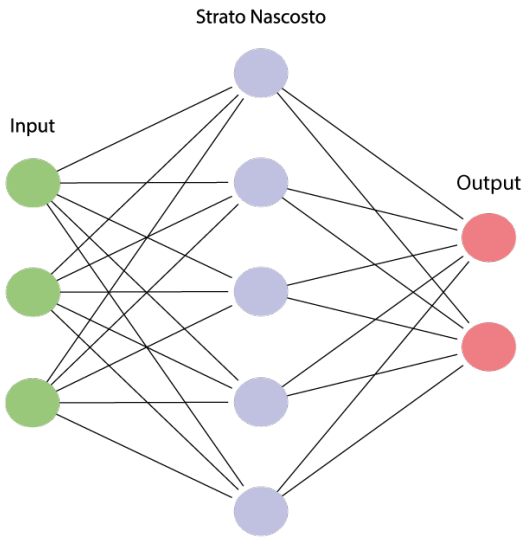


Figura 2.1: Esempio di un Multi Layer Perceptron.

utilizzati sono appunto chiamati *Deep Neural Network*. La particolarità di questi sistemi si trova nella loro struttura composta da più strati, o *layer*, ognuno di essi formati da molte trasformazioni non lineari, i neuroni; all'interno di una rete sono possibili un numero arbitrariamente grande di strati (per questo vengono dette profonde), e la tendenza degli ultimi anni è quella di crearne sempre più deep (con un numero di layer superiore a 100).

La tipologia più semplice di rete neurale è il Multi Layer Perceptron (MLP), formato da strati completamente connessi (o *fully connected*) (fig. 2.1), in cui tutti i neuroni di un livello sono relazionati a tutti quelli del successivo. Il primo layer rappresenta il vettore di input x, anche detto vettore di feature; lo strato che segue, ed eventualmente i successivi, vengono usualmente chiamati strati nascosti (o *hidden layer*) e rappresentano delle trasformazioni dei valori di ingresso attraverso una funziona parametrica non lineare, che è possibile esprimere come una moltiplicazione con la matrice dei pesi W_1, più un termine di bias b_1 seguito da una funzione non lineare come

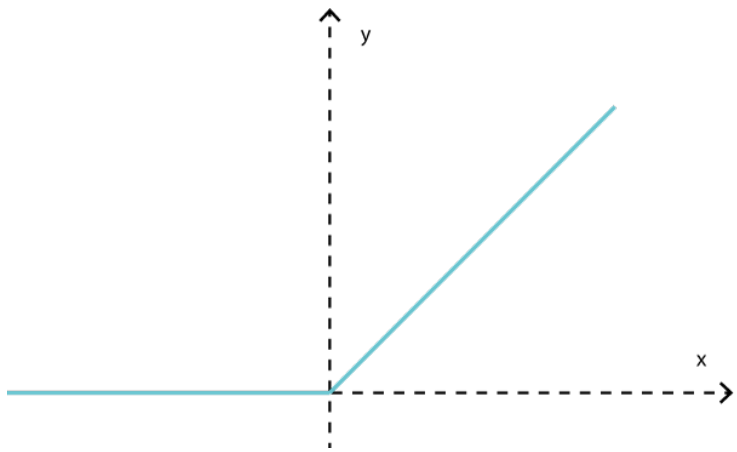


Figura 2.2: Grafico della funzione ReLu.

la ReLu (fig. 2.2):

$$h = ReLu(W_1 * x + b_1). \tag{2.1}$$

Lo strato nascosto h può essere a sua volta trasformato in altri insiemi di valori seguendo lo stesso procedimento, fino ad arrivare all'ultimo livello che fornisce gli elementi y di output; nel caso di esempio:

$$y = (W_2 * h + b_2). \tag{2.2}$$

La ReLu viene detta funzione di attivazione e modifica l'output di un particolare layer; esistono moltissime tipologie di funzioni di attivazione, lineari e non, che possono essere utilizzate a seconda della varietà di problema o della forma delle feature.

L'addestramento di una rete viene comunemente effettuato tramite un algoritmo di *backpropagation* [30]: in particolare nell'apprendimento supervisionato si ha a disposizione un insieme di dati (*training set*), costituito da coppie ingresso-uscita, che vengono utilizzate per far imparare alla rete le relazioni che esistono tra i diversi strati; l'algoritmo usa gli elementi appena citati per andare a modificare i pesi W con lo scopo di minimizzare una

loss function, che rappresenta l'errore delle predizioni ottenute sul training set. Se l'addestramento è effettuato in modo corretto, la rete acquisisce la capacità di prevedere l'output relativo ad un input mai visto prima.

Nelle applicazioni recenti sono state create moltissime tipologie di strati oltre a quelli fully connected, che possono portare vantaggi e svantaggi a seconda del contesto e del problema che abbiamo di fronte. Citiamo ora tre di queste particolarmente interessanti per la continuazione del lavoro:

- *Strati Convoluzionali:* gli strati di questo tipo [24] (fig. 2.3) considerano gli ingressi come dei dati disposti su una griglia 2D; ogni input (o pixel della griglia) è connesso ai neuroni del successivo strato, ma non a tutti: in particolare ogni unità avrà come ingresso una piccola parte dei dati. L'idea di questa procedura è di cercare di trovare la stessa feature su ogni possibile frazione dell'immagine applicando a ogni passo la stessa operazione, in modo simile a come un segnale viene filtrato per convoluzione. Per fare questo, ogni neurone deve imparare a calcolare la stessa funzione, quindi deve essere vincolato ad avere gli stessi pesi degli altri: si parla infatti di pesi condivisi. Gli output a questi strati avranno a loro volta un'organizzazione spaziale, quindi si può iterare il procedimento anche nei layer successivi per estrarre altre caratteristiche di più alto livello. All'interno dello stesso strato convoluzionale è possibile tentare di ricavare svariate tipologie di feature (dello stesso livello) utilizzando più filtri convoluzionali con diversi pesi;

- *Strato di Embedding*: questo livello [37] agisce come traduttore da oggetti discreti, come parole o numeri interi, a vettori di valori reali. La dimensione degli output tipicamente non ha alcun significato, bensì la rete cerca di sfruttare i pattern che riesce a riconoscere e la distanza tra questi. Solitamente vengono usati per elaborare gli input discreti

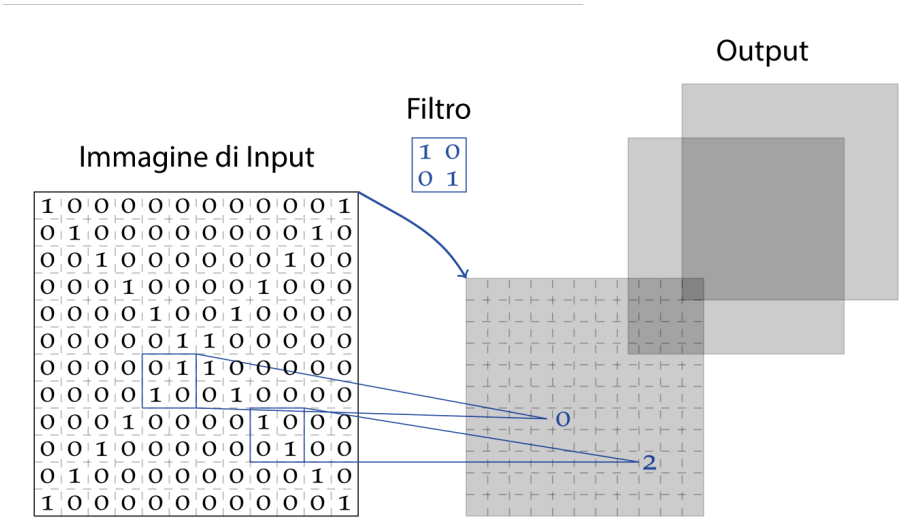


Figura 2.3: Immagine di un layer convoluzionale con un'immagine di input che viene convoluta attraverso diversi filtri per restituire un'immagine di output. In questo caso i pesi che vengono appresi rappresentano i coefficienti dei filtri.

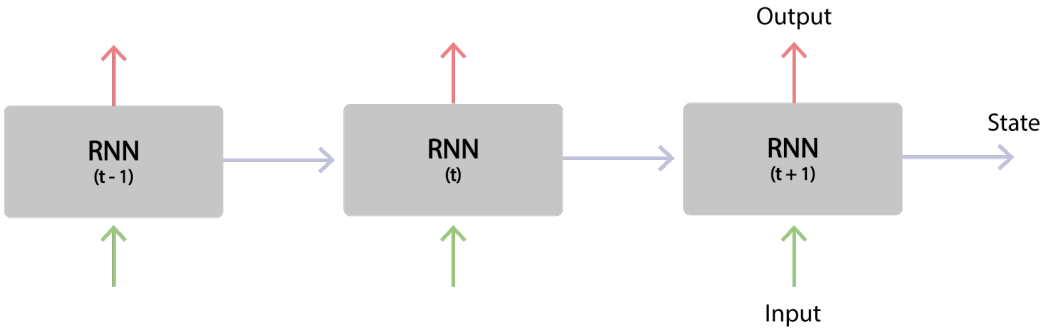


Figura 2.4: Schema di una rete neurale ricorrente che processa sequenze di dati temporali; parte dell'output al tempo t è input al tempo $t + 1$.

in quanto i classificatori, e in generale i sistemi di machine learning, sfruttano meglio i numeri reali e continui, piuttosto che valori interi;

- *Strati Ricorrenti*: gli strati ricorrenti sono molto utili quando si parla di sequenze di dati. I sistemi che sono composti da questi livelli sono una classe particolare di reti neurali che vengono chiamate *recurrent neural network* (RNN) in cui le connessioni tra le unità formano un ciclo. Questo crea uno stato interno alla rete che consente di determinare un comportamento che cambia dinamicamente nel tempo (fig. 2.4); l'apprendimento per questa tipologia di rete è standard e non è necessario aggiungere nessuna tecnica particolare. Esistono diverse tipologie di strati ricorrenti, quello più usato negli ultimi tempi è il Long Short-Term Memory, o LSTM [17].

2.2 Reinforcement Learning

Introduciamo brevemente la teoria fondamentale del Reinforcement Learning (RL) [2, 36] che servirà a capire meglio la sua versione deep. L'essenza

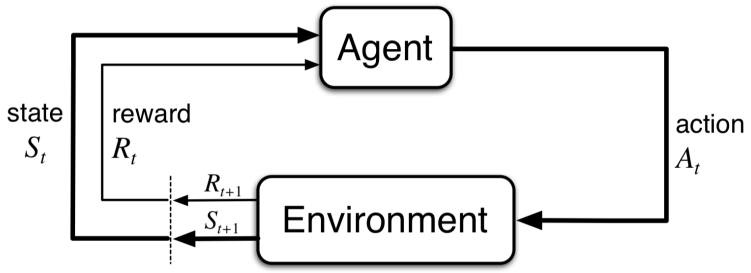


Figura 2.5: Schema dell'interazione tra agente ed ambiente in un problema di RL [36].

del RL è imparare attraverso le interazioni: un agente esegue delle azioni all'interno del suo ambiente e, dopo averne osservato le conseguenze, può imparare ad alterare il suo comportamento per massimizzare le ricompense che riceve ad ogni passo. Alla base del RL c'è quindi un meccanismo di *trial-and-error* che l'agente deve ripetere con l'obiettivo di ottenere una strategia che risolva l'ambiente in cui è immerso. L'attore ha la possibilità di intuire quali sono le mosse di maggior successo valutando il reward ottenuto e adattando di conseguenza la sua policy, cioè la metodologia di scelta delle azioni che definisce il suo comportamento, in modo da ottenere la massima ricompensa cumulativa nel tempo. La struttura del problema viene raffigurata in fig. 2.5.

2.2.1 Processo Decisionale di Markov

Formalmente un problema di RL può essere modellato come un Processo Decisionale di Markov (MDP), tramite cui è possibile descrivere l'evoluzione nel tempo dell'ambiente facendo l'assunzione che quest'ultimo sia completamente osservabile, cioè un contesto in cui lo stato precedente caratterizza in modo esaustivo tutto il processo passato. La condizione è conosciuta come Proprietà di Markov: in poche parole, questa significa che qualunque deci-

sione presa allo stato s_t può essere basata solamente dallo stato precedente s_{t-1}, piuttosto che dall'insieme $\{s_0, s_1, ..., s_{t-1}\}$.

Un MDP rappresenta un ambiente di RL in cui tutti gli stati hanno la proprietà di Markov. In particolare, considerando il caso discreto, esso può essere descritto da una tupla $< S, A, T, R, \gamma >$, in cui troviamo:

- un insieme di stati S;
- un insieme di azioni A;
- una funzione di transizione $T : S$ x A x $S \rightarrow [0, 1]$ che restituisce la probabilità che, dato una stato s e un'azione a al tempo t, si raggiunga un possibile stato s':

$$T_a(s, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a); \tag{2.3}$$

- una *reward function* $R : S$ x A x $S \rightarrow \mathbb{R}$ che restituisce la ricompensa immediata/istantanea. Dato uno stato s e un'azione a, se si passa allo stato s' si ottiene:

$$R_a(s, s') = R(s'|s, a) = \mathbb{E}[r_{t+1} | s_t = s, a_t = a, s_{st+1} = s'], \tag{2.4}$$

  dove E rappresenta il valore atteso;

- un fattore di sconto $\gamma \in [0, 1]$ che rappresenta le differenze tra ricompense future e quelle immediate.

Un episodio di questo processo forma una sequenza finita di stati, azioni e ricompense: $s_0, a_0, r_1, s_1, a_1, ..., s_{T-1}, a_{T-1}, r_T, s_T$; questo termina dopo T passi con lo stato finale s_T (ad esempio, la schermata di game over di un gioco).

2.2.2 Funzione di Reward

Il risultato di un'azione al tempo t dell'agente viene espresso sotto forma numerica e chiamato reward $r \in \mathbb{R}$, con $|r_k| < \infty, \forall k < t$; questo numero fa parte dell'input al tempo $t + 1$ dell'agente assieme allo stato s_{t+1} risultante dall'azione eseguita. Con *reward function* si intende la funzione all'interno dell'ambiente che restituisce un certo numero r_t dati lo stato s_t, l'azione a_t e lo stato successivo s_{t+1} (2.4). Le ricompense risultano essere una caratteristica essenziale negli algoritmi di RL in quanto rappresentano una parte della quantità cumulativa che deve essere massimizzata e fornisce una valutazione qualitativa del comportamento dell'agente.

Per avere una buona strategia a lungo termine che possa far ottenere il reward cumulativo più alto all'interno di un singolo episodio, l'agente deve tener conto non solo delle ricompense istantanee, ma anche di quelle future. Si definisce quindi *discounted future reward* al tempo t la quantità:

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}, \tag{2.5}$$

dove T rappresenta la fine dell'episodio; spesso, G_t viene anche indicato come *ritorno* (o *return*). Una buona strategia per l'agente è cercare di scegliere al passo t quella azione che massimizzi G_t per avere una ricompensa totale cumulativa più alta. Il parametro $\gamma \in [0, 1]$ rappresenta il fattore di sconto e definisce quanta importanza diamo ai reward immediati piuttosto che a quelli lontani nel tempo: se $\gamma = 0$, allora la strategia farà affidamento solo ai reward istantanei senza tenere conto di quello che potrebbe succedere nel futuro; se il nostro ambiente fosse deterministico, cioè se ad ogni azione corrispondesse sempre lo stesso reward, allora potremmo usare un $\gamma = 1$.

2.2.3 Policy

Per interagire in modo adeguato con l'environment, l'agente deve avere un metodo per prendere le decisioni opportune nel momento in cui riceve una nuova osservazione: questo è il compito della policy. In generale viene indicata con π e rappresenta una mappatura dagli stati alla distribuzione di probabilità sulle azioni:

$$\pi : S \rightarrow p(A = a|S). \tag{2.6}$$

Se l'ambiente è episodico, allora la sequenza di stati, azioni e ricompense all'interno di un gioco costituisce una traiettoria o *rollout* della policy. Ogni rollout accumula i rewards dall'ambiente con i quali viene costruito G_t; l'obiettivo di un problema di RL è dunque quello di trovare una policy ottima π^* che riesca a raggiungere la massima quantità prevista di G_t a partire da tutti gli stati:

$$\pi^* = \underset{\pi}{\operatorname{argmax}}\, \mathbb{E}[G_t|\pi] \tag{2.7}$$

La policy definita in (2.6) è di tipo stocastica: è possibile definire anche policy deterministiche dove, invece che restituire una probabilità, essa associa ad uno stato direttamente una azione univoca. È anche possibile considerare un MDP non episodico, dove $T = \infty$; in situazioni di questo genere, per prevenire una somma infinita di ricompense dobbiamo usare un $\gamma < 1$.

Esistono diversi algoritmi per risolvere un problema di RL ed ottenere quindi la π^* desiderata: i due approcci principali sono i metodi basati sulle *value function* e metodi chiamati *policy search*. Esiste anche una terza tipologia chiamata *actor-critic* che utilizza entrambe le tecniche precedenti.

2.2.4 Metodi Value Function

I metodi value function sono basati sulla stima del valore G_t atteso di trovarsi in un determinato stato. La funzione *state-value* $V^\pi(s)$ rappresenta quindi il ritorno previsto partendo da s ed eseguendo la policy π:

$$V^\pi(s) = \mathbb{E}[G_t|s, \pi]. \tag{2.8}$$

La policy ottima π^* ha una funzione di state-value corrispondente e viceversa; la funzione di state value ottima può essere definita come:

$$V^*(s) = \max_\pi V^\pi(s) \ \ \forall s \in S. \tag{2.9}$$

Se abbiamo $V^*(s)$ disponibile, la policy ottima può essere ottenuta considerando tutte le azioni disponibili nello stato s_t e scegliendo quella che massimizza

$$\pi^* = \mathbb{E}_{s_{t+1} \sim T(s_{t+1}|s_t, a)}[V^*(s_{t+1})]. \tag{2.10}$$

In un setting RL generico però la funzione di transizione T non è conosciuta; per questo viene costruita un'altra funzione, la *state-action value function* o *quality function*:

$$Q^\pi(s, a) = \mathbb{E}[G_t|s, a, \pi]. \tag{2.11}$$

Simile a V^π, la Q-function rappresenta il valore della scelta di un'azione a in uno stato s seguendo la policy π, che è uguale al ritorno aspettato partendo dallo stato s, eseguendo l'azione a secondo la policy π. La strategia ottima data $Q^\pi(s, a)$ può essere trovata scegliendo l'azione a in modo *greedy* ad ogni stato:

$$\pi^* = \operatorname*{argmax}_a Q^\pi(s, a); \tag{2.12}$$

sotto questa π, possiamo definire anche V^π massimizzando $Q^\pi(s, a)$:

$$V^\pi(s) = \max_a Q^\pi(s, a) \tag{2.13}$$

In un generico problema MDP in cui il numero di stati e azioni è finito e limitato, è possibile usare una lookup table per tenere in memoria la state-action value function: essa consiste semplicemente in una matrice, che viene chiama *Q-table*, in cui gli stati e le azioni corrispondono alle righe o alle colonne. In particolare, mentre l'agente prende delle decisioni sulla base della policy π, si aggiornano i valori della tabella mediando le esperienze passate in base ai reward ottenuti, agli stati esplorati e alle azioni compiute.

Per imparare Q^π, è possibile sfruttare la proprietà di Markov e definire la funzione conosciuta come *equazione di Bellman* [6] che ha la seguente forma ricorsiva:

$$Q^\pi(s_t, a_t) = \mathbb{E}_{s_{t+1}}[r_{t+1} + \gamma Q^\pi(s_{t+1}, \pi(s_{t+1}))]; \tag{2.14}$$

ciò significa che la valutazione di Q^π può essere migliorata usando il valore stimato corrente. Questo concetto è alla base degli algoritmi *Q-Learning* [41] e *state-action-reward-state-action* (SARSA) [31]:

$$Q^\pi(s_t, a_t) \leftarrow Q^\pi(s_t, a_t) + \alpha\delta, \tag{2.15}$$

dove α è il learning rate, $\delta = Y - Q^\pi(s_t, a_t)$ è l'errore di differenza temporale (TD) e Y è la funzione target di un normale problema di regressione. L'algoritmo SARSA, che rientra nella tipologia degli *on-policy*, viene usato per migliorare la stima di Q^π usando le transizioni generate dalla policy derivata da essa, che prevede $Y = r_t + \gamma Q^\pi(s_{t+1}, a_{t+1})$. Il Q-learning invece viene definito *off-policy* in quanto Q^π viene aggiornato a partire dalle transizioni che non necessariamente vengono generate dalla policy derivata: infatti, questo algoritmo usa $Y = r_t + \gamma \max_a Q^\pi(s_{t+1}, a)$, che cerca di approssimare direttamente Q^π.

L'aggiornamento descritto precedentemente, applicato in maniera iterativa a tutti gli stati successivi di tutti gli episodi, permette di trovare la funzione Q^*, e di conseguenza la policy di selezione stato-azione ottimale.

2.2.5 Metodi Policy Search

I metodi policy search non hanno bisogno di mantenere in memoria una funzione che valuti il modello, ma cercano di stimare direttamente una policy ottima π^*. Tipicamente, viene scelta una strategia parametrizzata π_θ, i cui parametri vengono aggiornati per massimizzare il ritorno atteso:

$$J(\theta) = \mathbb{E}_{\pi_\theta}[\sum_t G_t|\theta]. \tag{2.16}$$

La (2.16) rappresenta la *score function* di un problema di ottimizzazione, che è possibile risolvere attraverso i metodi basati sul gradiente. Dato un rollout di dimensione T, dobbiamo trovare quindi i parametri θ migliori che massimizzino la funzione:

$$\theta^* = \underset{\theta}{\operatorname{argmax}}\, \mathbb{E}_{\pi_\theta}[\sum_t G_t]; \tag{2.17}$$

per fare questo calcoliamo il gradiente della score function con i parametri θ attuali, ed aggiorniamo quest'ultimi con la regola:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta). \tag{2.18}$$

Quando si costruisce direttamente la policy, usualmente si fanno restituire dei parametri per una distribuzione di probabilità: per degli ambienti che necessitano di azioni continue, questi potrebbero essere la media e la deviazione standard di distribuzioni Gaussiane, mentre per problemi discreti potrebbero essere le probabilità individuali di una distribuzione multimodale. Il risultato è quindi una policy stocastica da cui possiamo direttamente campionare le azioni.

Come abbiamo accennato prima, nei metodi *gradient based* viene calcolato il gradiente per migliorare la stima della policy ottima; per questo obiettivo viene usata la tecnica del REINFORCE [42], conosciuta anche come stima

della *score-function* [12] o stima della *likelihood-ratio* [13]. Formalmente, la regola del REINFORCE può essere usata per calcolare il gradiente del valore atteso su una funzione f di una variabile aleatoria X con rispetto ai parametri θ:

$$\nabla_\theta \mathbb{E}_X[f(X;\theta)] = \mathbb{E}_X[f(X;\theta)\nabla_\theta \log p(x)]. \tag{2.19}$$

Nel nostro caso quindi, dobbiamo calcolare:

$$\nabla_\theta \mathbb{E}_X[f(X;\theta)] = \nabla_\theta \mathbb{E}_{\pi_\theta}[\sum_t G_t] = \mathbb{E}_{\pi_\theta}[(\sum_t G_t \nabla_\theta \log \pi_\theta(a_t|s_t))]. \tag{2.20}$$

Tuttavia in un contesto come questo, in cui i campioni dei dati sono ad alta varianza, cercare di stabilizzare i parametri del modello potrebbe essere molto difficile. Si cerca quindi di introdurre una nuova variabile chiamata *baseline* b che si va a sottrarre al reward cumulativo:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[(G_t - b)\nabla_\theta \log \pi_\theta(a_t|s_t)]. \tag{2.21}$$

L'idea della baseline è quella di aumentare la probabilità di un'azione che è risultata migliore di quanto ci si aspettasse; può essere vista come una stima del reward che si può ottenere a partire solo dallo stato. La definizione rende la funzione b molto simile ad una state-value function, ed è proprio su questo concetto che si basano i metodi ibridi Actor-Critic [22, 33]. Qui abbiamo due soggetti: un "attore", ovvero la policy, impara usando i feedback dal "critico", cioè la value function. Gli algoritmi ibridi usano quindi una state-value function al posto della baseline, ma a differenza degli altri metodi basati su questa, gli actor-critic utilizzano una funzione b che viene imparata; dobbiamo quindi calcolare sia il gradiente della policy che quello della baseline.

Con i metodi actor-critic si è conclusa l'analisi di tutti gli aspetti e i concetti fondamentali del RL che permettono al lettore di proseguire il corso

della tesi: verrà ora trattato come il Deep Learning può essere associato alle tecniche viste fin'ora creando il Deep Reinforcement Learning, verranno esaminati quali aspetti cambiano rispetto alla modalità classica e verranno approfonditi i due algoritmi principali per questo lavoro, il Deep Q-Learning (della famiglia dei metodi value function) e il Proximal Policy Optimization (della tipologia dei policy gradient).

2.3 Deep Reinforcement Learning

Il Deep Reinforcement Learning (DRL) è nato per trattare i problemi di RL che hanno a che fare con dati ad alta dimensionalità che le tecniche classiche faticano a risolvere a causa della *curse of dimensionality*. Questo passo avanti è stato possibile grazie alla capacità delle reti neurali di imparare a rappresentare delle feature di basso livello su input molto complessi e dalla loro proprietà di *function approssimator*. Intuitivamente, il DRL si basa sull'addestrare delle deep neural networks per approssimare la policy ottimale π^* e/o le value function V^* e Q^*. Lo sviluppo del DRL si deve soprattutto alla creazione dell'algoritmo Deep Q-Learning (DQN) [26] sviluppato nel 2015 da DeepMind di Google [8]: da quel momento molti ricercatori hanno posto interesse in questo ambito e ancora oggi si stanno scoprendo nuovi algoritmi e tecniche che migliorano i risultati ottenuti precedentemente, dimostrando che ci sono margini di miglioramento e problematiche da risolvere. Nelle sezioni successive analizzeremo il DQN e il Proximal Policy Optimization, che è l'algoritmo usato per questo lavoro, ma è possibile trovarne altri esempi tra cui Double Deep Q-Learning [15], Deep Q-Learning from Demonstrations [16], A3C [25], Normalised Advantage Function [14], Generalized Advantage Estimation [33], etc.

2.3.1 Deep Q-Learning

Il DQN [26], sviluppato da DeepMind, è stato pensato per creare degli agenti che potessero giocare ai videogiochi della console Atari 2600 [5] in modo comparabile a dei videogiocatori umani. Nel loro set-up, lo spazio delle azioni è discreto e rappresenta tutte le 18 combinazioni di tasti disponibili nel *game controller*, ma il numero di possibili stati è troppo grande per poter memorizzare la $Q(s, a)$ semplicemente attraverso una matrice. Questo sistema infatti ha come input i frame greyscale dei giochi scalati alle dimensioni 84×84 pixel, con valori tra $[0, 255]$: di conseguenza la grandezza della tabella sarebbe $|S| \times |A| = 18 \times 256^{86 \times 86}$; anche se fosse possibile creare e gestire una matrice così grande, essa sarebbe molto sparsa. La forza del DQN risiede dunque nella sua abilità di rappresentare in modo compatto sia le osservazioni ad alta dimensionalità che la *Q-function* usando delle reti neurali.

L'obiettivo è quindi trovare la funzione parametrizzata

$$Q(s_t, a_t; \theta), \tag{2.22}$$

attraverso il sistema raffigurato in fig. 2.6 , dove θ rappresenta i pesi della rete. Un problema nell'approssimare la state-action value function attraverso una funzione non lineare come una rete neurale è la forte instabilità dei parametri. Questa è causata da vari fattori: le correlazioni presenti nella sequenza di osservazioni, il fatto che piccoli aggiornamenti nella funzione Q possono portare a grandi cambiamenti della policy e le correlazioni che esistono tra i valori di $Q(s, a)$ e i valori target.

DeepMind usa due tecniche particolari per migliorare il risultato ed evitare di rendere instabile la funzione di approssimazione:

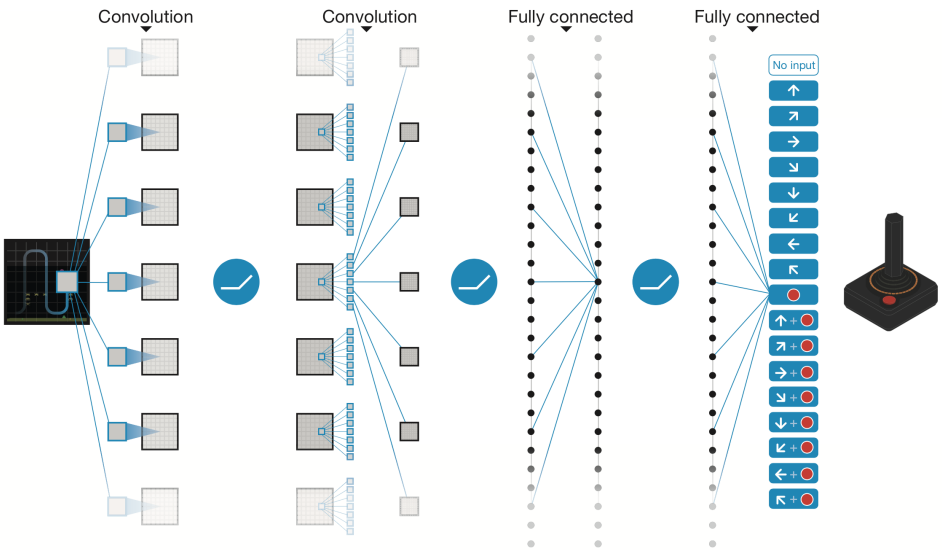


Figura 2.6: Schema della rete convoluzionale usata da DeepMind [26]. L'input è formato da immagini 84×84 mentre il primo hidden layer è uno strato convoluzionale composto da 32 filtri 8×8 con stride 4 e seguito da una ReLu; il secondo è anch'esso convoluzionale con 64 filtri 4×4 e stride 2 e sempre seguito da ReLu; l'ultimo layer nascosto è composto da uno strato fully connected con 256 neuroni. Il layer di uscita è un fully connected con un singolo output per ogni azioni disponibile nel gioco.

- *Experience replay*: questa tecnica tiene in memoria le esperienze passate del tipo:

$$e_t = (s_t, a_t, r_t, s_{t+1}), \tag{2.23}$$

  per ogni timestep t all'interno di un dataset:

$$D_t = \{e_1, ..., e_t\}. \tag{2.24}$$

  Durante la fase di training, viene effettuato l'aggiornamento della rete solamente su un campione casuale delle esperienze passate $U(D)$, utilizzando quindi un addestramento *offline*;

- *Target model*: durante l'aggiornamento si utilizzano contemporaneamente due funzioni di approssimazione, ovvero due reti strutturalmente identiche ma con pesi differenti: la Q-network e la rete target, con la differenza che i parametri dell'ultima vengono tenuti fissi per un certo periodo di tempo. L'errore della nuova rete viene calcolato come differenza tra il valore della rete target e il valore stimato di quella attuale.

L'aggiornamento risultante utilizza dunque una loss function del tipo:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\sim U(D)}[(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta))^2], \tag{2.25}$$

dove γ è il discount factor, θ sono i paramteri della Q-network e θ^- sono i parametri della rete target. Quest'ultimi sono aggiornati solamente dopo un certo numero di passi e vengono tenuti fissi tra gli aggiornamenti della Q-function; inizialmente $\theta = \theta^-$.

2.3.2 Proximal Policy Optimization

Negli algoritmi Deep Policy Gradient semplicemente si cerca di approssimare π_θ attraverso una rete neurale, aggiornando i pesi tramite backpropa-

gation definendo la Policy Gradient Loss per timestep da (2.21):

$$L^{PG}(\theta) = \mathbb{E}_t[\log \pi_\theta(a_t|s_t)A_t], \tag{2.26}$$

dove $A_t = G_t - b$. In questo caso non è necessario tenere in memoria le esperienze passate, in quanto i policy gradient lavorano direttamente sui rollout: alla fine di un'iterazione dell'aggiornamento, questo viene eliminato. Una soluzione al problema delle instabilità discusso precedentemente è fornita dall'algoritmo Trust Region Policy Optimization (TRPO) [32] che, ad ogni aggiornamento cerca di non modificare troppo la nuova policy rispetto alla precedente. Il meccanismo dell'algoritmo però porta con sè dei calcoli molto complessi che rendono impossibile la risoluzione di problemi ad alta dimensionalità. La soluzione proposta da OpenAI è il Proximal Policy Optimization (PPO) [34] che si basa sempre sull'idea di limitare le differenze tra la nuova e la vecchia policy. Definiamo

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, \tag{2.27}$$

con $r(\theta_{old}) = 1$. Senza nessun tipo di vincolo, la massimizzazione della funzione obiettivo potrebbe condurre ad un aggiornamento eccessivamente grande della policy, per questo il PPO cerca di penalizzare i cambiamenti che muovono $r_t(\theta)$ troppo lontano da 1 tramite la definizione di una nuova loss function:

$$L^{CLIP}(\theta) = \mathbb{E}_t[\min(r_t(\theta)A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)], \tag{2.28}$$

dove ϵ rappresenta un iperparametro dell'algoritmo. In questo modo si obbliga r_t a stare nell'intervallo $[1-\epsilon, 1+\epsilon]$ limitando i cambiamenti eccessivi di π. In fig. 2.7 è rappresentato il grafico della loss nei casi in cui A_t sia positivo o negativo.

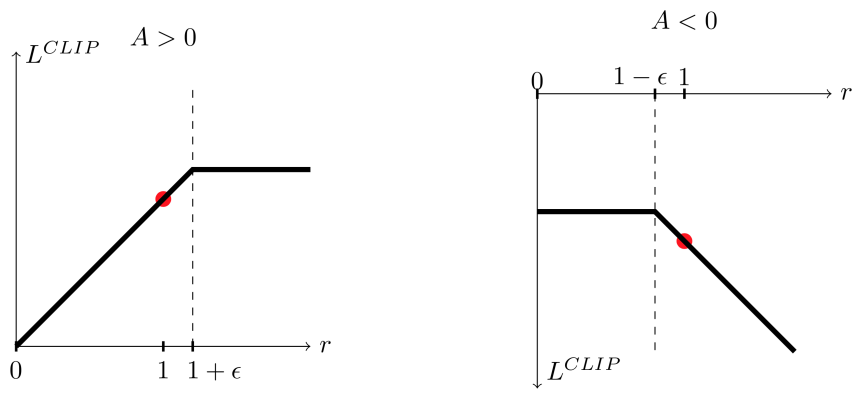


Figura 2.7: Grafici [34] della funzione di loss del PPO: nella sinistra abbiamo il caso in cui A_t è positivo, mentre a destra il caso in cui A_t è negativo.

Con queso capitolo si è conclusa la breve analisi degli algoritmi di DRL. Il PPO rappresenta lo stato dell'arte di queste tecniche applicate ai videogiochi: basti pensare all'esperimento OpenAI Five [27], in cui i ricercatori hanno creato 5 reti neurali per giocare contro 5 giocatori professionisti al videogioco competitivo DOTA 2 [10]. Le reti sono state allenate tramite l'algoritmo PPO a cooperare ed essere più forti e intelligenti dei giocatori umani: il risultato è stato sorprendente, con le reti che hanno vinto più partite contro i professionisti. La maggior parte dei lavori in questo ambito [3, 4, 20, 40] trattano di come creare delle intelligenze artificiali che sorpassino le capacità umane con dei videogiochi già esistenti. Lo scopo di questa tesi è però diverso, in quanto volevamo capire se il DRL può essere utile allo *sviluppo* di un videogioco commerciale: le nostre intelligenze artificiali non hanno quindi delle capacità *"super-umane"*, ma devono divertire l'utente e quindi risultano essere battibili e non troppo punitive e frustranti, ma comunque impegnative. Nella prossima sezione illustreremo le due tecniche classiche per creare dei nemici intelligenti all'interno di un prodotto ludico per permettere al lettore

di fare dei paragoni con la nuova tecnica sviluppata in questo lavoro.

2.4 AI nei Videogiochi

In questo capitolo vedremo le tecniche classiche ad oggi più popolari e usate di AI all'intero videogiochi. Sebbene ci siano tanti settori nello sviluppo di un videogioco in cui l'AI può essere utile, noi ci concentriamo solamente sulle meccaniche strategiche degli antagonisti del giocatore. I due algoritmi più utilizzati per comandare dei personaggi non giocanti sono le Finite State Machine e i Behavior Tree. Il loro forte utilizzo è evidenziato dal fatto che tutt'oggi, quando si parla di *game AI*, viene fatto riferimento ad uno di questi metodi.

2.4.1 Finite State Machine

Una Macchina a Stati Finiti (FSM) [43], e la sua variante FSM gerarchica, è sicuramente la tecnica più utilizzata per comandare le AI nei giochi. Questo algoritmo fa uso di grafi: sono una rappresentazione astratta di un insieme interconnesso di oggetti, simboli, eventi, azioni o proprietà del fenomeno che deve essere rappresentato. In particolare un grafo contiene dei nodi, una rappresentazione matematica di una qualche astrazione, e gli archi, le relazioni condizionali che esistono tra i nodi. Una FSM può stare in uno stato alla volta: il nodo corrente può cambiare solo se una delle condizioni rappresentate dagli archi che partono da esso viene soddisfatta. In breve, una FSM in un gioco è composta da tre elementi fondamentali:

- un insieme di *stati* che memorizzano le informazioni riguardanti un compito: ad esempio, un agente si può trovare nello stato *esplora*;

- un insieme di *transizioni* tra nodi che indicano come uno stato cambia e sono descritte da condizioni che si devono verificare: ad esempio, se un agente sente il rumore di uno sparo, si deve muovere allo stato *allerta*;
- un insieme di *azioni* che deve essere eseguito all'interno di ogni stato: ad esempio, se un nemico si trova nel nodo *esplora*, deve *muoversi casualmente* e *cercare il giocatore*.

Le FSM sono molto semplici da progettare e implementare; tuttavia possono essere estremamente complesse da creare su larga scala e sono quindi limitate a certe semplici tipologie di compiti. Inoltre non sono molto flessibili, dinamiche e riusabili e con questa tecnica vengono creati dei nemici che sono altamente prevedibili e di conseguenza poco divertenti. In fig. 2.8 è possibile vedere un semplice esempio di FSM.

2.4.2 Behavior Tree

Gli alberi dei comportamenti (BT) [43] sono un sistema che, similmente alle FSM, modellano delle transizioni tra un insieme finito di *task*. Il vantaggio rispetto alle FSM è la loro modularità: se progettati a dovere, i BT possono gestire delle strategie complesse composte da semplici compiti. La più grande differenza tra la tecnica precedente è data dal fatto che l'albero non è formato da stati, ma da *comportamenti*. I BT usano una struttura ad albero con un nodo radice e un insieme di nodi padri e figli che rappresentano i comportamenti; in fig. 2.9 ne è mostrato un esempio. Un agente naviga l'albero a partire dalla radice; in seguito, attiva le coppie parent-child come indicato nello schema. Un nodo figlio può restituire tre valori al suo parent: *run* se quel comportamento è ancora attivo, *success* se è concluso o *failure* se è fallito. I BT sono composti da tre diversi tipi di nodi:

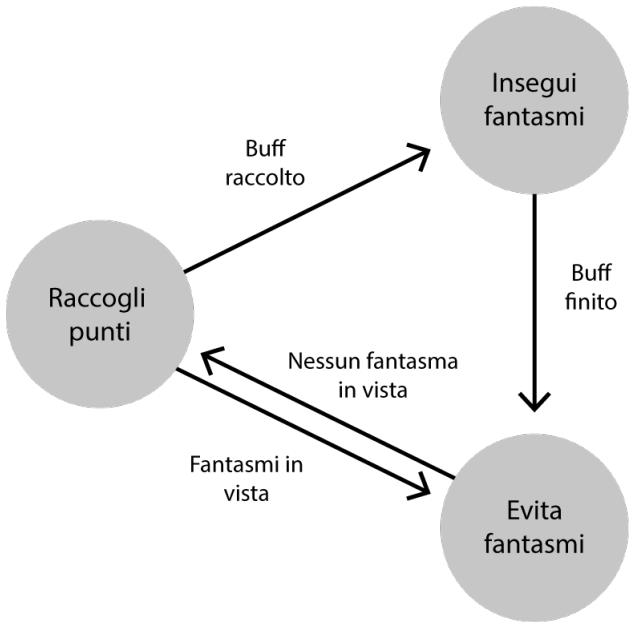


Figura 2.8: Schema di esempio di una FSM: i nodi rappresentano i compiti dell'agente, mentre gli archi descrivono le condizioni per passare da uno stato all'altro.

- *Sequence*: i nodi figli rappresentano una sequenza di azioni. Se il nodo figlio ha successo, la catena continua ed eventualmente, se tutti i figli di essa restituiscono *success*, allora anche il nodo padre ha successo; altrimenti la sequenza fallisce;

- *Selector*: esistono due sotto-tipologie di nodi selector: *priority* e *probability*. Quando viene usato un *probability selector* i comportamenti dei figli sono selezionati sulla base di una probabilità scelta a priori; se invece viene usato un *priority selector* i task dei figli vengono ordinati in una lista e si eseguono uno alla volta. In qualunque caso, se i figli hanno successo, allora lo avrà anche il padre; se usiamo un *probability selcetor* e i figli falliscono, allora fallisce anche il padre; altrimenti, con un *priority*, viene preso il nodo successivo nella lista. Se tutti i nodi conseguenti falliscono, fallisce anche il padre;

- *Decorator*: aggiunge della complessità e affina le capacità di un comportamento di un nodo figlio; ad esempio, specifica il numero di volte che deve eseguire le sue istruzioni per completare l'obiettivo.

Paragonati alle FSM, i BT sono più facili da gestire e da testare ma soffrono sempre degli stessi problemi, anche se in maniera meno evidente.

Come è possibile capire, gli algoritmi classici hanno delle incertezze causate dal fatto che dipendono fortemente da un modello di strategia ben preciso e stabilito a priori. Inoltre, quando il gioco diventa molto complesso ed è necessario creare diverse tipologie di nemici, queste tecniche non scalano sufficientemente bene e c'è bisogno di molto lavoro in più per gestire tutti i possibili stati e/o task. Attraverso il nostro esperimento abbiamo dimostrato che è possibile creare tramite il DRL delle AI per agenti che siano completamente *model-free* e che siano comparabili, se non migliori, alle entità create

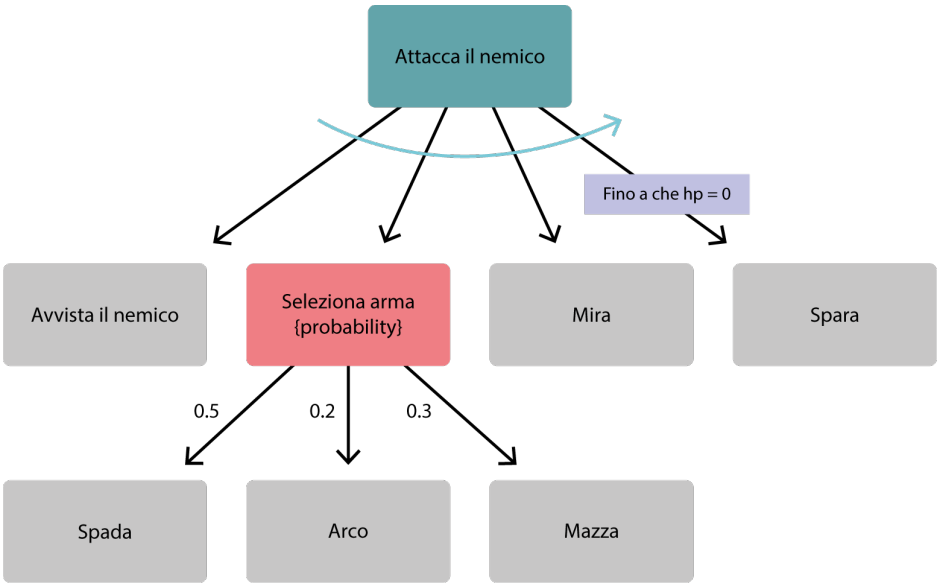


Figura 2.9: Schema di esempio di un BT: rettangoli blu rappresentano i nodi *sequence*, quelli rossi descrivono i *selector* mentre i rettangoli viola i *decorator*.

con i BT o le FSM: basta trovare il giusto set-up di allenamento con la corretta reward function e rete neurale, e a quel punto sarà l'agente ad imparare autonomamente quali sono i comportamenti migliori, quali sono le strategie da seguire e quali sono gli elementi dell'ambiente da tenere conto. In più, è possibile creare infinite tipologie di nemici cambiando semplicemente dei parametri nel set-up: sarà l'algoritmo stesso ad inferire la giusta tattica a seconda delle caratteristiche dell'agente. Dopo aver appreso quindi la teoria fondamentale del DRL e delle AI generali nei videogiochi, nel prossimo capitolo verrà presentato il contesto sul quale abbiamo applicato queste tecniche creando un prototipo di videogioco perfettamente funzionante e testabile guidato dal DRL.

Capitolo 3

Game Design e Desiderata

Come già descritto, lo scopo di questo lavoro era creare un videogioco commerciale, un prodotto interamente ludico, avendo come idea principale quella di utilizzare il DRL per costruire le intelligenze artificiali che avrebbero guidato gli antagonisti del giocatore. Come passo iniziale dunque è stato necessario definire e progettare la tipologia di gioco e le meccaniche principali che potessero essere accostate ai vari algoritmi. In questo capitolo verranno descritti i motivi della scelta di determinate caratteristiche di gioco e verranno definite le meccaniche principali di gameplay utili alla definizione di un problema di DRL.

3.1 Tipologia di Gioco

Per il nostro scopo è stato necessario bilanciare due aspetti: il bisogno di avere un ambiente relativamente limitato, in modo da poter testare gli algoritmi e i concetti di DRL, con lo sviluppo di un gioco che fosse divertente e stimolante. Per scegliere le giuste meccaniche di gameplay abbiamo fatto riferimento alle principali caratteristiche rintracciabili nei generi videoludici

(in fig. 3.1 è possibile vedere due esempi di tipologie completamente diverse tra loro):

- *third person vs first person game*: nei videogiochi, la prima persona si riferisce ad una prospettiva grafica resa dal punto di vista del personaggio giocante. Tipicamente i prodotti di questo tipo fanno parte della categoria dei giochi di azione, la maggior parte definiti come First Person Shooter (FPS). Al contrario, i giochi in terza persona rientrano in un genere in cui la telecamera inquadra una porzione più grande dello scenario, tipicamente a volo di uccello; tra i maggiori esponenti di questa tipologia possiamo includere i Giochi di Ruolo (RPG) e i Giochi di Strategia (RTS). Per mantenere la proprietà di Markov descritta nel capitolo 2 ed avere un ambiente completamente osservabile, la visuale in terza persona si è rilevata la più adatta: in questo modo l'agente ha le stesse informazioni che vede il giocatore, e permette facilmente di offrire delle osservazioni codificate *"perfette"* piuttosto che le immagini dirette del gioco;

- *turn based vs real time*: i giochi *real time* offrono all'utente la possibilità di inserire un input ogni frame, rendendolo più frenetico e meno strategico. Al contrario, i *turn based* permettono la scelta di un'azione solamente in determinati momenti, nella maggior parte dei casi rendendo necessario l'utilizzo di una tattica. Tipicamente in ambienti *real time*, la codifica delle azioni in un problema di RL non è banale e spesso si fa uso di valori continui; diversamente, in contesti *turn based* lo spazio delle azioni risulta essere fortemente quantizzato. In più, se vogliamo che l'agente riesca ad estrapolare una strategia dagli elementi dell'ambiente, la scelta ottimale ricade sull'ultimo genere. In generale,

(a) Doom.

(b) Final Fantasy Tactics.

Figura 3.1: Differenze tra generi di giochi con Doom 3.1a [9] e Final Fantasy Tactics 3.1b [11]: il primo è un FPS *real time* e tipicamente 3D, la manovra è molto frenetica e con poca strategia e lo spazio di movimento e di azioni è molto ampio; il secondo invece è un RPG strategico a turni, in cui ogni personaggio ha il suo set di azioni discreto e la tattica risulta essere fondamentale.

i giochi *turn based* hanno dei vincoli di efficienza più rilassati rispetto ai *real time*;

- *3D vs 2D*: tendenzialmente i giochi *first person* e/o *real time* sono immersi in un ambiente in cui anche la terza dimensione assume un ruolo fondamentale: i personaggi infatti possono muoversi e guardare anche sull'asse verticale e non solo sul piano XZ. Questo renderebbe la codifica delle osservazioni e delle azioni di un agente ancora più complessa, e dunque, per lasciarla più semplice possibile, abbiamo preferito un contesto in cui gli attori si possono muovere solamente nel piano orizzontale e all'interno di una griglia.

3.2 Meccaniche di Gameplay

Date le considerazioni precedenti si è deciso di sviluppare un *RougeLike*: questo termine fa riferimento ad un particolare genere di giochi di ruolo *turn based*, tipicamente 2D con visuale in terza persona, caratterizzato da elementi comuni come mappa casuale e morte permanente; il nome si riferisce a *Rogue* [29] (fig. 3.2), videogioco del 1980 capostipite di questo genere. Questa tipologia è tutt'oggi molto apprezzata, soprattutto la sua variante *real time RougeLite* rintracciabile in esempi moderni come i giochi *The Binding of Isaac* [19], *Hoplite* [18], *Pixel Dungeon* [28], etc.

In fig. 3.3 è mostrato uno dei mock-up iniziali ed uno screenshot di una delle ultime versioni del gioco; vediamo ora tutti gli elementi principali del gameplay.

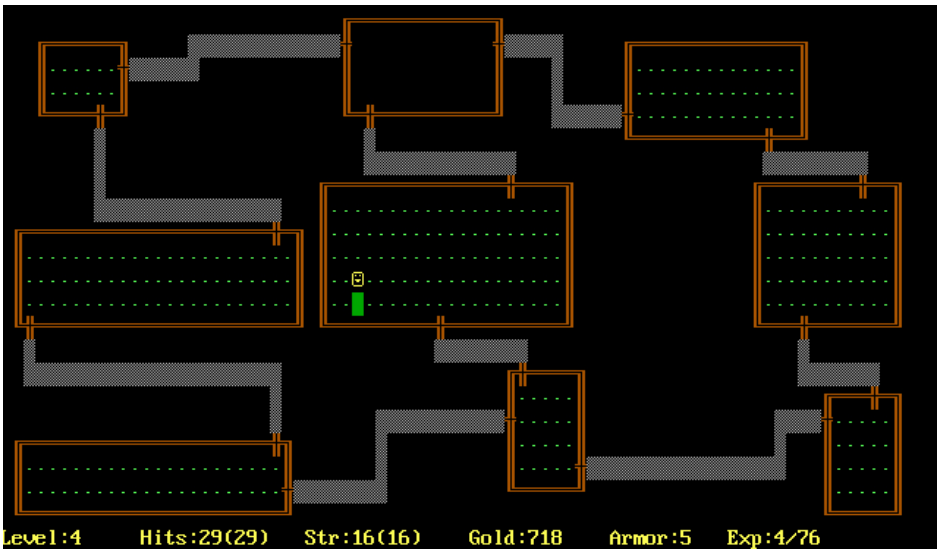


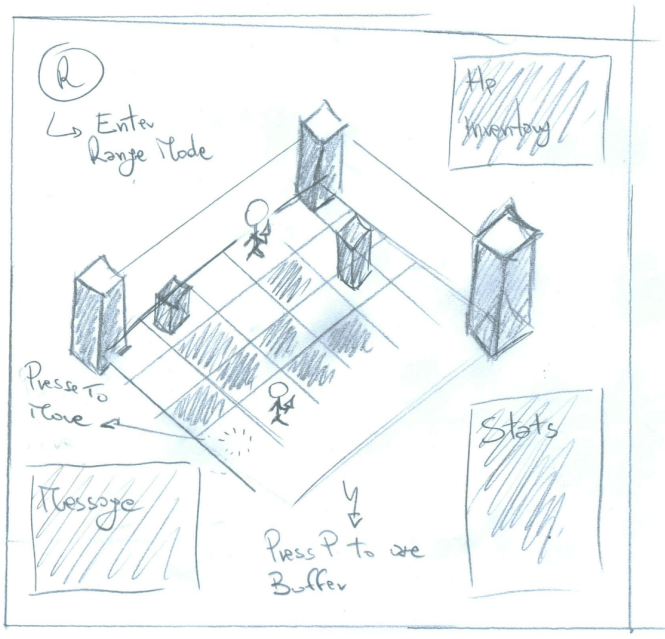
Figura 3.2: Uno screenshot ASCII-based rappresentativo di Rogue [29].

3.2.1 Attori

Il gioco prevede lo scontro diretto tra il giocatore e uno o più agenti. L'utente impersonerà il proprio personaggio e dovrà vincere la sfida contro dei nemici guidati completamente da un algoritmo di DRL. Lo scopo di ciascun attore è portare i punti vita del proprio obiettivo a 0.

3.2.2 Ambiente

L'Ambiente è formato da una griglia di forma casuale di grandezza massima 10×10. In ogni casella è possibile trovare un personaggio, una colonna/impedimento che blocca il passaggio oppure un oggetto da raccogliere. Gli oggetti sono di tre tipi: armi da mischia, armi a distanza o pozioni; queste ultime sono a loro volta suddivise in pozioni curative o buff. Ogni personaggio ha delle caratteristiche particolari:



(a) Mock-up.

(b) Screenshot.

Figura 3.3: Mock-up (3.3a) iniziale e screenshot (3.3b) della versione finale del gioco.

- *Punti Vita* (HP): determinano il numero di danni che un personaggio può subire. Quando i punti vita scendono a 0, l'attore muore;
- *Attacco* (ATK): determina il modificatore dei danni quando si esegue un attacco in mischia;
- *Destrezza* (DEX): determina sia il modificatore dei danni quando si esegue un attacco a distanza sia il range di tiro del personaggio;
- *Difesa* (DEF): determina il modificatore dei danni subiti per entrambe le tipologie di attacco;

Ogni personaggio ha a disposizione un inventario in cui può raccogliere gli oggetti che incontra sulla mappa: questo è composto da un'arma da mischia, un'arma a distanza e una pozione. Quando un attore raccoglie un oggetto, il precedente salvato nell'inventario andrà perso; per raccoglierlo basta andare nella sua stessa casella. Il numero e la disposizione dei collezionabili e degli oggetti invalicabili, così come la forma della mappa, è procedurale: ogni stanza ha la sua particolare e unica configurazione.

3.2.3 Spazio delle Azioni

È importante evidenziare che gli agenti e l'utente possono eseguire le stesse azioni, motivo per cui entrambi gli attori gareggiano allo stesso gioco: questo fattore fa sì che i nemici e il giocatore siano sullo stesso livello, rendendo fondamentale l'uso di una strategia per poter risolvere lo scenario. Questa meccanica di gameplay non è scontata in quanto molti videogiochi del genere, per aumentare il tasso di sfida e ovviare al problema della creazione di agenti intelligenti, utilizzano degli avversari con maggiori abilità rispetto a quelle dell'utente, che di conseguenza vengono percepiti come più scaltri. Ogni personaggio può svolgere 17 azioni discrete:

- *8 azioni di movimento*: ogni soggetto si può muovere di una casella a turno in tutte le 8 direzioni orizzontali, verticali e diagonali. Non è possibile finire il movimento in una casella con un impedimento o una colonna; se nella destinazione è presente un oggetto, verrà raccolto e posto automaticamente nell'inventario; se nella casella di destinazione è presente un personaggio nemico, allora chi ha effettuato l'azione eseguirà un attacco in mischia invece di muoversi. Questa tipologia infligge all'avversario un numero di danni pari a:

$$D = melee_damage + ATK_{character} - DEF_{enemy}, \tag{3.1}$$

  dove $melee_damage = rand(atk_min_{melee}, atk_max_{melee})$: ogni arma è infatti caratterizzata da un attacco massimo e minimo;

- *Usare la pozione*: ogni soggetto può utilizzare una pozione che è presente nell'inventario. Questa è l'unica azione che non fa perdere il turno, pertanto dopo averla usata, è possibile effettuare una nuova mossa. Gli effetti sono di due tipi:
  - *Pozione Curativa*: dona a chiunque la usi 5 HP. Il numero massimo iniziale di HP non può mai essere superato;
  - *Buff*: aumenta a chiunque la usi ATK e DEF per 3 turni;

- *8 azioni a distanza*: ogni soggetto può sparare con la propria arma a distanza invece che muoversi. È possibile lanciare in una delle 8 direzioni: se in una di queste è presente un nemico all'interno del range del personaggio, verrà inflitto un danno pari a:

$$D = range_damage + DEX_{character} - DEF_{enemy}, \tag{3.2}$$

  con $range_damage = rand(atk_min_{range}, atk_max_{range})$. Se non è presente nessun nemico a tiro, queste azioni non possono essere ef-

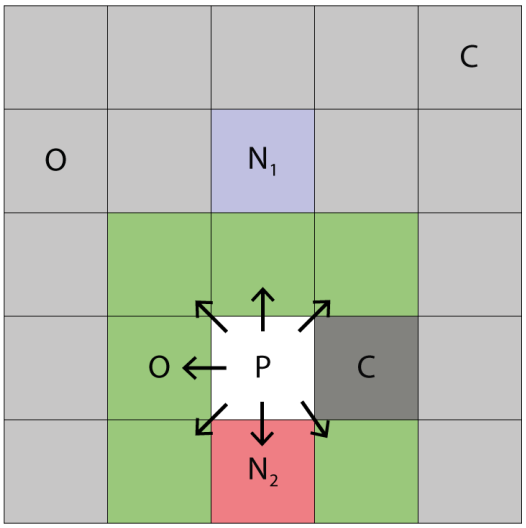


Figura 3.4: Esempio schematico di turno di un giocatore: P indica il personaggio giocante, N_1 e N_2 sono i nemici, le O indicano gli oggetti raccoglibili e le C indicano le colonne/oggetti non oltrepassabili. Nel suo turno, P può muoversi in una delle caselle verdi: se la destinazione coincide con la O, raccoglierà l'oggetto in essa; se si muove nella posizione di N_2, eseguirà un attacco in mischia contro quel nemico rimanendo però dove si trova in questo turno; non può muoversi nella casella con C; in alternativa, può effettuare un attacco a distanza contro N_1 in quanto il suo range è superiore a 2. Una volta effettuata una di queste azioni, il turno passerà ai nemici.

fettuate. In fig. 3.4 è possibile vedere un esempio schematico di turno.

Con questa sezione si è conclusa l'analisi generale del contesto e delle meccaniche di gameplay pensate per il nostro gioco: abbiamo capito quali erano le scelte più opportune per la tesi e abbiamo definito degli agenti che agiscono all'interno di un particolare ambiente con 17 azioni disponibili, cercando di vincere contro l'utente. Il passo successivo è stato riflettere

sugli obiettivi adatti al nostro prodotto: dai primi esperimenti e dagli studi effettuati su altri lavori infatti è emerso che gli agenti allenati con il machine learning possono avere dei comportamenti che non si prestano bene ad un videogioco ludico.

3.3 Desiderata

Nelle sezioni precedenti abbiamo definito un contesto di gioco e degli algoritmi che permettono a degli agenti di risolvere nel miglior modo possibile l'ambiente in esso. Ricordiamo però che stiamo parlando di un prodotto ludico: è fondamentale che il gioco sia divertente e stimolante dal punto di vista dell'utente. Verranno ora presentati al lettore gli obiettivi che ci siamo posti, e che abbiamo raggiunto, per creare un vero videogioco:

- l'agente deve essere *convincente*, deve agire in modi che siano in un certo senso prevedibili e che possono essere interpretati come intelligenti. Deve essere una minaccia per il giocatore, altrimenti risulterebbe troppo facile da battere e, senza una vera sfida, poco divertente. Il nemico deve quindi avere una strategia che lo porti a vincere e non si deve comportare in maniera controintuitiva: l'utente non si deve accorgere di star giocando contro qualcosa di finto e macchinoso, ma l'algoritmo deve essere comparabile alle intelligenze artificiali classiche di altri prodotti del genere;

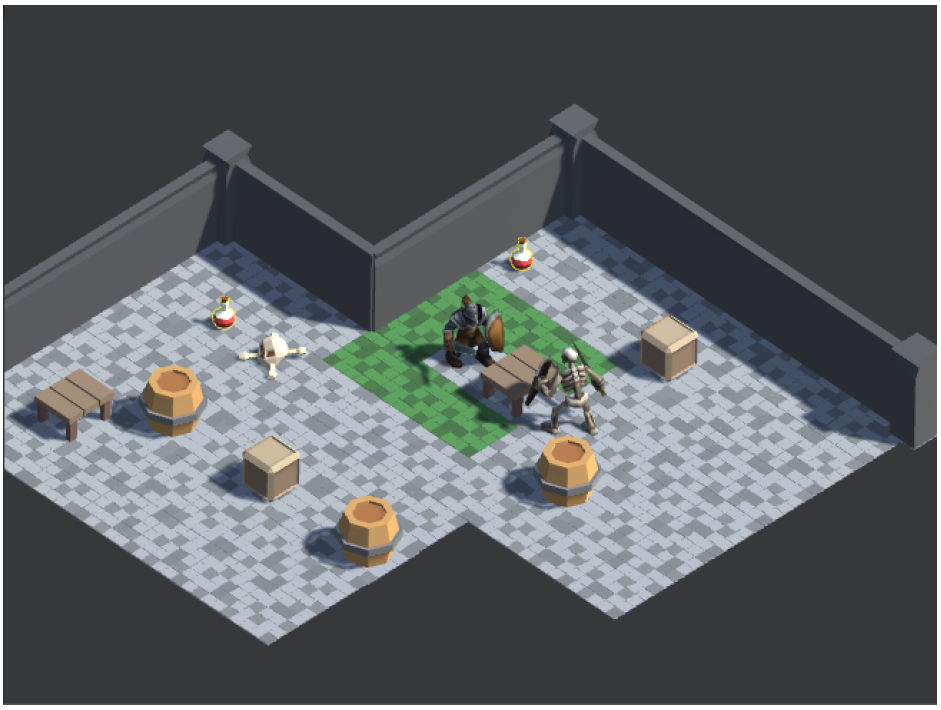
- al contempo l'agente deve essere *imperfetto*: un nemico "troppo intelligente" che riesce a risolvere il gioco già dalla prima mossa non è adatto per un prodotto ludico. È importante che l'utente abbia sempre la possibilità di vincere per evitare che il gioco venga considerato troppo

punitivo e frustrante, in quanto combattere contro un nemico con capacità "super umane" risulterebbe scoraggiante; è quindi fondamentale che l'agente sia sempre battibile;

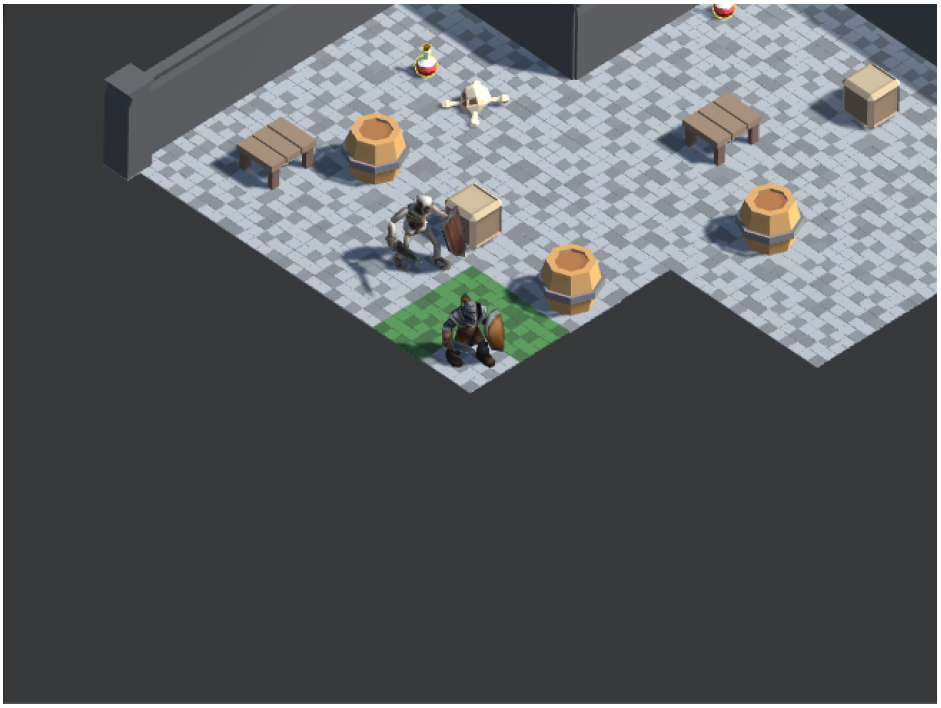
- il nemico deve essere *model-free*: riprendendo il concetto generale di machine learning (2.1), il programmatore non deve impostare l'agente in modo che segua una particolare strategia ma, grazie al processo *trial-and-error* tipico del DRL, deve imparare in maniera autonoma quali sono le mosse giuste in ogni condizione. Inoltre un agente *model-free* deve rendere le tecniche sviluppate nella tesi generalizzabili ad un qualsiasi gioco che mantenga le stesse caratteristiche di gameplay complessive appena definite;

- allo stesso tempo è necessario un certo grado di varietà nelle dinamiche di gioco per aumentare il divertimento: combattere sempre con lo stesso nemico alla lunga potrebbe diventare monotono e tedioso, con il rischio che l'utente abbandoni prematuramente il gioco. Il sistema deve quindi avere dei parametri tali che, una volta modificati, permettano di far imparare all'agente nuove strategie per creare diverse tipologie di nemici.

La maggior parte dei lavori svolti sul DRL [3,4,27,35] prevede la creazione di agenti "perfetti" che riescono a risolvere l'ambiente al massimo delle loro capacità contro l'intelligenza artificiale già definita all'interno del gioco. Il nostro algoritmo invece deve avere a che fare con utenti umani che devono essere messi alla prova, ma devono sempre essere in grado di vincere anche con un certo margine di errore. Tutti questi motivi inducono una certa difficoltà di valutazione del sistema, poiché il risultato *quantitativo* è tanto importante quanto quello *qualitativo*. A supporto di questo concetto, in fig. 3.5a è



(a) Esempio di agente stupido.



(b) Esempio di agente intelligente.

Figura 3.5: Esempio di due esperimenti fallimentari in relazione agli obiettivi preposti.

mostrato come il sistema, senza il giusto modello, sia riuscito creare con facilità un agente stupido: in questo caso, il nemico è riuscito a riconoscere la posizione dell'utente nella mappa, ma non è stato in grado di capire ciò che aveva intorno; questo ha portato l'agente a rimanere bloccato contro degli oggetti invalicabili e ad avere inevitabilmente un basso score quantitativo. Mentre in fig. 3.5b invece è mostrato l'esempio di un agente così intelligente da risultare poco divertente: in una prima fase di sviluppo del gioco, in un ambiente limitato cui erano presenti danni deterministici e solamente attacchi ravvicinati, questo agente sapeva che chi attaccava per primo era praticamente sicuro di vincere, per questo costringeva l'utente ad eseguire le sue mosse offensive dopo quelle del nemico. Questo comportamento portava l'algoritmo ad avere un reward altissimo a discapito del risultato qualitativo. In definitiva l'agente deve avere il giusto modello che gli permetta di migliorare il suo reward, verificabile tramite la massimizzazione della sua score function ma al contempo è necessario che abbia un comportamento credibile e paragonabile a quello degli umani, e questo fattore è testabile solamente tramite dei *test di giocabilità* del prodotto stesso. La definizione dei requisiti è risultata un'importante punto di riferimento in prospettiva della risoluzione del problema: tramite il DRL infatti si possono creare tantissime tipologie di agenti e comportamenti, che possono essere altamente imprevedibili e innaturali; attraverso la specifica degli obiettivi abbiamo quindi definito in maniera precisa cosa volevamo ottenere da questa tesi. Nel prossimo capitolo verrà presentato nel dettaglio il modello proposto dal nostro lavoro che soddisfa tutti i requisiti di questa sezione.

Capitolo 4

Modello Proposto

In questo capitolo verrà presentato nel dettaglio il sistema di DRL proposto; ogni scelta è stata fatta tenendo conto degli obiettivi desiderati descritti nella sezione 3.3. Verranno ora approfonditi i quattro principali elementi che ne determinano il risultato: la rete neurale, i parametri del PPO, la reward function e il set-up di training. La definizione finale di tale modello è avvenuta dopo molteplici esperimenti: per risolvere un problema di DRL è necessario procedere per piccoli passi per capire qual è la miglior combinazione di parametri possibile. Nelle prime fasi dello sviluppo abbiamo quindi deciso di ridurre ai minimi termini la complessità del sistema: siamo partiti con un semplice agente che doveva raggiungere un determinato punto della mappa, e abbiamo aggiunto sempre più elementi fino ad arrivare alle meccaniche di gioco complete. Abbiamo provato molte strutture di reti differenti, con vari strati e operazioni, diverse reward function e svariati set-up di allenamento, e il modello descritto in seguito è quello che ha contribuito nel miglior modo al soddisfacimento dei requisiti specificati precedentemente.

Per questa parte di lavoro è stato usato *TensorForce* [23] sviluppato da Alexander Kuhnle et al., un framework open-source per il DRL con un enfasi

particolare sulla modularità e sulla semplicità di utilizzo in applicazioni sia per la ricerca che nella pratica. TensorForce è costruito sulle basi di TensorFlow [1] ed è compatibile con Python 3. Questa libreria segue un insieme di scelte di design di alto livello che la differenziano da altri framework simili:

- *design modulare component-based*: le implementazioni delle varie funzionalità cercano di essere più generali possibile in modo da essere altamente configurabili;
- *separazione tra gli algoritmi RL e l'applicazione*: gli algoritmi sono totalmente indipendenti dalla particolare struttura degli input e degli output, così come dall'interazione con l'ambiente dell'applicazione;
- *modelli basati completamente su TensorFlow*: l'intera logica di RL è implementata in TensorFlow, con il quale è possibile, in un secondo momento, estrapolare il grafo della rete in modo da poter essere utilizzato in altre applicazioni.

Uno dei più grandi vantaggi di TensorForce rispetto alle altre librerie è la possibilità di usare delle reti neurali completamente personalizzate, in modo da creare dei sistemi anche molto complessi. Il framework implementa molti algoritmi di DRL, tra cui il DQN e il PPO visti nel capitolo 2, permettendo ampia libertà di scelta sugli iperparametri e sulla struttura di essi.

4.1 Rete Neurale

In fig. 4.1 è rappresentato lo schema della rete neurale utilizzata come *policy approximator* nell'ultima versione del lavoro. Essa è composta da quattro rami di input:

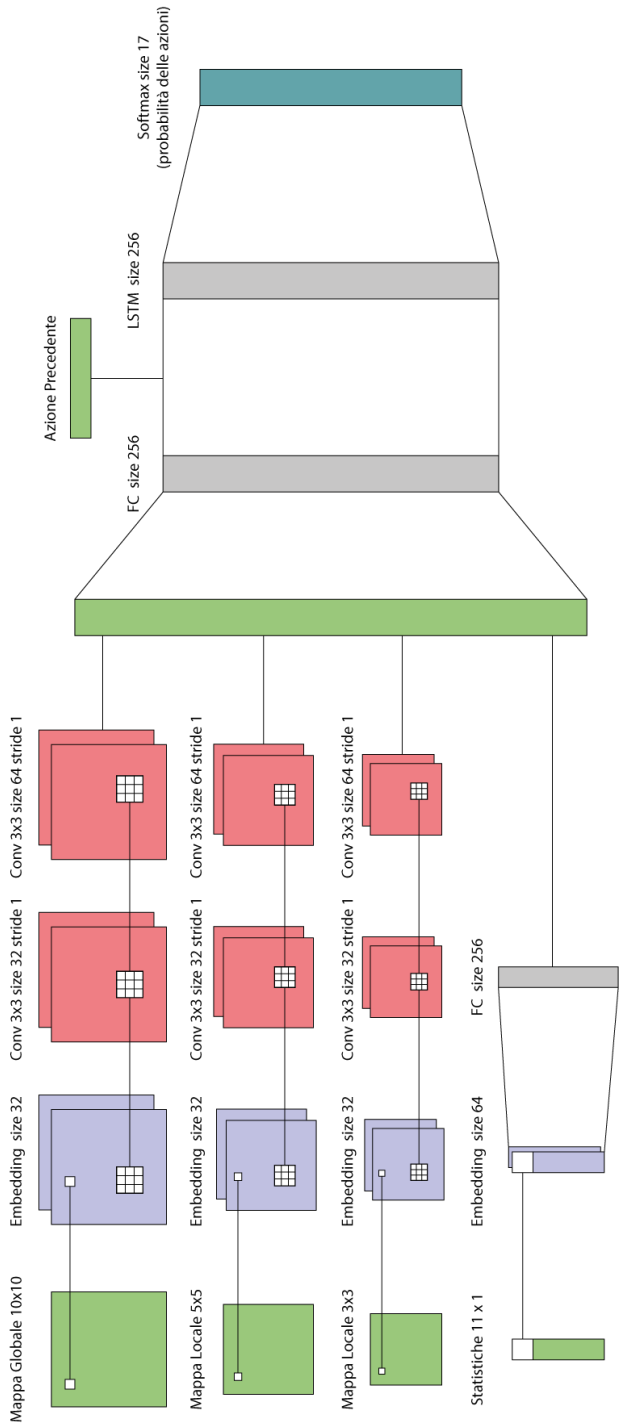


Figura 4.1: Rete Neurale proposta. I layer viola corrispondono a embeddings, i layer rossi a delle convoluzioni, i layer grigi a degli strati FC e l'ultimo layer blu illustra un FC con softmax, che rappresenta l'uscita della rete; i layer verdi indicano gli input alla rete.

- il primo ramo prende in ingresso la matrice raffigurante la mappa statica e globale della stanza di grandezza 10×10, ed è composta dalla seguente codifica:
  - 0 = casella invalicabile/altro agente;
  - 1 = casella libera;
  - 2 = agente;
  - 3 = giocatore;
  - 4+ = oggetti raccoglibili.

  Lo strato di input è seguito da un embedding di size 32, da una convoluzione con 32 filtri di dimensioni 3×3 e stride 1, e da un'altra convoluzione con 64 filtri 3×3 e stride 1;

- il secondo ramo prende in ingresso la mappa locale 5×5 relativa alla posizione dell'agente: a differenza del primo questo input è dinamico ed è sempre centrato nella posizione dell'osservatore; la codifica è fatta esattamente allo stesso modo del primo ingresso. La mappa è seguita da un embedding di grandezza 32, una convoluzione a 32 filtri 3×3 e stride 1 e da un'altra a 64 filtri con le stesse caratteristiche della precedente;

- il terzo ramo è simile al secondo, con la differenza che la mappa locale è di grandezza 3×3. Ha la stessa codifica e sequenza di operazioni dei primi due;

- l'ultimo ramo ha come ingresso un vettore di 11 elementi contenente l'insieme di valori descritto in seguito:
  - HP dell'agente nell'intervallo $[0, 20]$;

- la pozione salvata nell'inventario dell'agente;
- l'arma da mischia salvata nell'inventario dell'agente;
- l'arma a distanza salvata nell'inventario dell'agente;
- un valore indicante se l'agente ha un buff attivo oppure no;
- un valore indicante se l'agente può attaccare a distanza e in quale direzione;
- HP del giocatore nell'intervallo $[0, 20]$;
- la pozione salvata nell'inventario del giocatore;
- l'arma da mischia salvata dell'inventario del giocatore;
- l'arma a distanza salvata dell'inventario del giocatore;
- un valore indicante se il giocatore ha un buff attivo oppure no.

Questo strato è seguito da un embedding di dimensioni 64 e da uno strato FC di size 256.

Un esempio di codifica delle osservazioni è mostrato in fig. 4.2. Le uscite di tutti i rami vengono concatenate per formare un unico vettore che viene fatto passare attraverso uno strato FC a 256 neuroni nascosti; al risultato di quest'operazione viene concatenata la codifica *one-hot* dell'azione eseguita al passo precedente e il nuovo vettore rappresenta l'ingresso di uno strato LSTM.

Come è stato già accennato nel capitolo 2, l'uscita di un algoritmo Policy Gradient è una distribuzione di probabilità sulle possibili mosse permesse dall'environment: in questo caso l'ultimo layer della rete è rappresentato da una softmax a 17 uscite, tante quanto è grande lo spazio delle azioni; ogni valore di questo output rappresenta la probabilità che una determinata mossa, eseguita in quello stato, porti a massimizzare il discounted future reward.

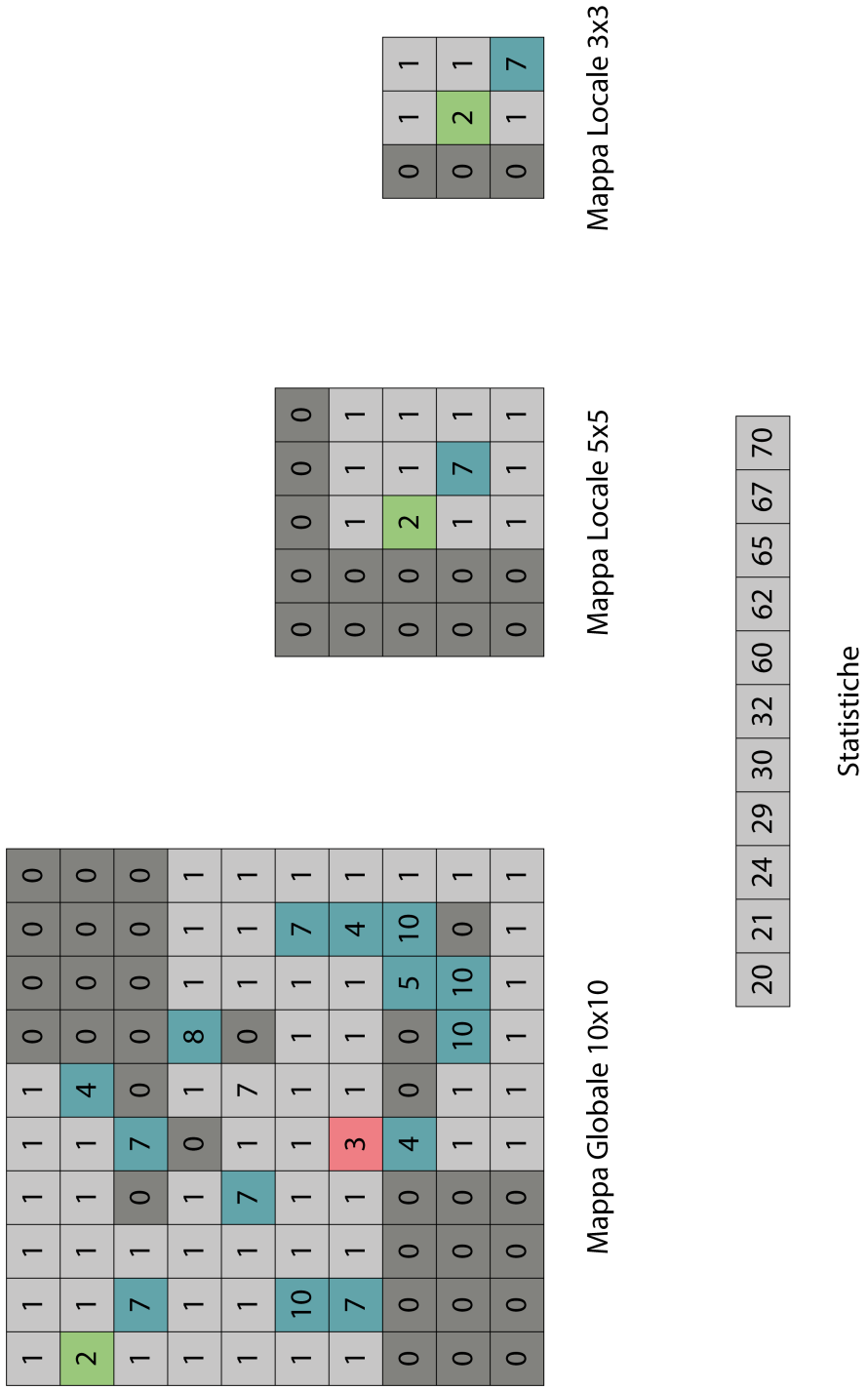


Figura 4.2: Rappresentazione delle osservazioni in input alla rete. Nella figura è mostrato il punto di vista dell'agente (in verde) sulla mappa globale e locale, più il suo vettore delle statistiche. I numeri sono elencati nella sezione 4.1.

La policy ha poi bisogno di un metodo per scegliere l'azione da eseguire data l'uscita della rete. Nel modello proposto non viene selezionata la mossa con il valore più alto, ma viene semplicemente campionato il vettore a seconda delle probabilità calcolate: dunque non si sceglie la mossa *più probabile*, ma *una delle più probabili*. Eventualmente questo fattore porta l'agente a non fare sempre ciò che considera più giusto, e dunque si permette al sistema di commettere degli errori: questo comportamento lo pone maggiormente sullo stesso livello di un utente umano. In più, evita che l'agente si arresti in un ciclo di mosse ripetitive e sempre uguali, aumentando la varietà di situazioni.

La configurazione è frutto di molti esperimenti: come già evidenziato in fig. 3.5, nelle prime fasi era presente solo l'input globale del primo ramo; come risultato avevamo un sistema che riusciva a capire in che porzione della mappa si trovava il nemico, ma non riusciva a riconoscere ciò che aveva intorno. Si è resa necessaria l'aggiunta di un input relativo e locale all'agente a due scale differenti (5×5 e 3×3), in questo modo il sistema riesce a distinguere correttamente cosa si trova attorno a lui agendo di conseguenza.

Molta importanza ha l'uso degli embedding: questi permettono al sistema di distinguere nel modo migliore le differenze di significato tra i numeri interi in input. Particolare attenzione va posta nell'input dell'ultimo ramo; seguendo l'idea di DeepMind per OpenAi Five [27], che usa un embedding distinto per ciascuna statistica, per far si che l'agente non confonda il contenuto di due valori uguali ma semanticamente diversi ogni valore ha il suo range disgiunto dagli altri, ad esempio:

- HP dell'agente $\in [0, 20]$,
- Pozione $\in [21, 23]$,
- Arma in mischia $\in [24, 26]$,

- e così via...

L'utilizzo di un embedding con un vocabolario espanso è uguale all'uso di uno strato distinto di questo tipo per ciascun valore. Per quanto riguarda la baseline è stata usata la stessa struttura con la differenza che lo strato LSTM è sostituito da un FC con size 256 senza usare l'azione al passo precedente.

4.2 PPO e iperparametri

Come ampiamente specificato, per questo lavoro abbiamo utilizzato l'algoritmo PPO (2.3.2). Anche per questa sezione la scelta degli iperparametri è conseguenza di molteplici esperimenti; di seguito verranno elencati i valori più importanti:

- per quanto riguarda il metodo di aggiornamento, un rollout dell'agente è composto da 10 episodi, ed ognuno di questi è costituito al massimo da 100 step/azioni; ogni episodio può terminare in un successo (vittoria dell'agente), in un fallimento (sconfitta dell'agente) o nel raggiungimento del numero massimo di passi. Al termine di 10 episodi, il sistema procede all'aggiornamento dei pesi attraverso l'esperienza appena memorizzata;

- ricordiamo che in un algoritmo actor-critic come il PPO sono presenti due funzioni che devono essere imparate, la policy e la baseline. Quest'ultima ha il compito di una normale state-value function, ovvero quello di valutare lo stato corrente e dare una stima del possibile reward che l'agente può ottenere, con l'obiettivo di aumentare la probabilità di un'azione che è risultata migliore di quanto ci si aspettasse; il valore calcolato dal critico viene poi usato nella loss function della

rete principale, utilizzando $A_t = G_t - b$ invece che semplicemente G_t. Come già accennato nella sezione 4.1, in questo modello la baseline ha una struttura simile a quella della policy. L'utilizzo di due reti neurali prevede dunque l'apprendimento dei pesi di due sistemi diversi, che hanno bisogno quindi di due learning rate separati; mentre la policy utilizza la funzione definita in 2.28, per aggiornare i suoi parametri la baseline impiega una distanza L2 tra il discounted future reward reale e quello predetto:

  - learning rate della policy $lr_p = 10^{-6}$;
  - learning rate della baseline $lr_b = 10^{-4}$;

- $\epsilon = 0.2$, parametro della loss function 2.28 del PPO;
- $\gamma = 0.99$, il fattore di sconto che determina la differenza tra ricompense immediate e future 2.2.2;

Il PPO, e in generale gli algoritmi di DRL, sono molto sensibili al cambiamento degli iperparametri: una combinazione sbagliata potrebbe portare l'agente a non imparare correttamente, oppure a raggiungere un massimo locale anziché assoluto. I valori elencati sono quelli che empiricamente ci hanno portato il giusto compromesso tra risultato quantitativo e qualitativo.

4.3 Funzione di Reward

Per quanto riguarda i reward l'obiettivo era quello di creare una funzione che fosse più sparsa possibile per far sì che l'agente riuscisse ad estrapolare una strategia in modo autonomo, senza essere influenzato da ricompense date per delle singole azioni. Alla luce di questo, la relazione proposta in funzione di ogni passo è:

$$R(t) = -0.01 + \begin{cases} -0.1 \text{ per un movimento contro un oggetto invalicabile} \\ +10.0 * hp_factor \text{ per la vittoria} \end{cases}, \tag{4.1}$$

dove hp_factor è l'ammontare normalizzato di HP rimasti all'agente. Il valore -0.01 viene dato per ogni mossa indipendentemente dal risultato: l'obiettivo è quello di accelerare l'agente perché concluda il proprio task senza perdere tempo in mosse inutili. A questo viene aggiunto -0.1 per ogni movimento contro oggetti invalicabili, per aiutare il sistema ad apprendere dove è possibile spostarsi: non è necessario una strategia per questo comportamento, deve solamente essere imparato rapidamente per evitare che l'agente si blocchi contro qualche oggetto. L'ultimo valore $10.0 * hp_factor$ viene dato in caso di vittoria; il termine hp_factor aiuta l'agente ad imparare l'importanza dei punti vita: non deve solamente vincere, ma lo deve fare con il maggior numero di HP possibile. Questo aiuto non influenza la strategia di combattimento, ma velocizza la capacità riconoscere quanti HP rimangono prima di perdere.

4.4 Set-Up di Allenamento

Per avere un agente che riesca a gestire tutte le possibili situazioni che possono occorrere in un combattimento contro un giocatore umano, è necessario un certo grado di casualità nelle caratteristiche dell'ambiente:

- la mappa della stanza, così come il numero degli oggetti invalicabili e raccoglibili e la loro posizione, è casuale;
- la posizione iniziale sia dell'agente che del nemico è casuale;

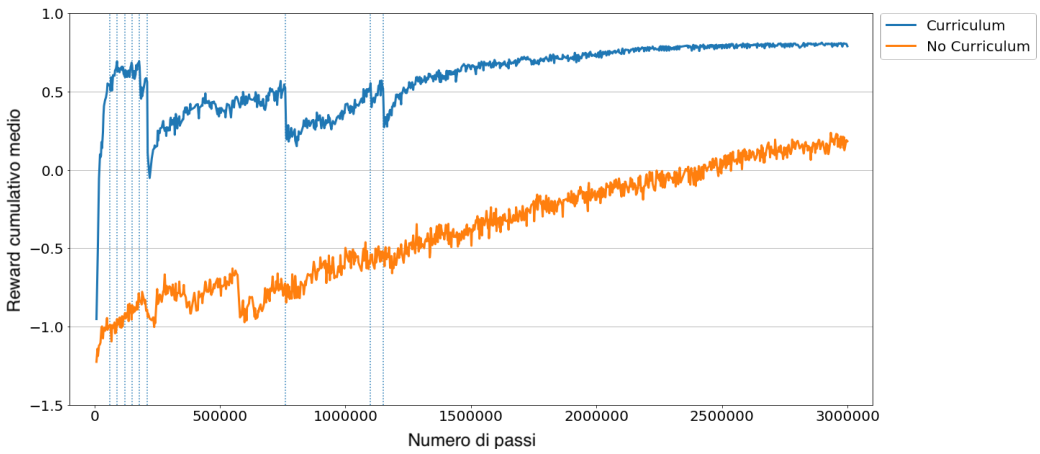


Figura 4.3: Esempio di curriculum learning per un problema generale di DRL [21]. È possibile vedere che in determinati problemi l'allenamento senza curriculum impiega molto più tempo a raggiungere un certo score, e non è detto che riesca ad arrivare allo stesso valore massimo di una sessione con curriculum.

- l'equipaggiamento iniziale di entrambi gli attori è casuale.

Per facilitare l'allenamento di un compito così difficile, aggravato da una reward function così sparsa, è stata usata la tecnica del Curriculum Learning [7] (fig. 4.3). Questa procedura di apprendimento prevede l'introduzione graduale di aspetti che aumentino sempre di più la complessità dell'ambiente. In questo modo, nella fase iniziale di allenamento l'agente avrà a che fare con un contesto molto semplificato rispetto a quello finale, dove raggiungere l'obiettivo risulterà essere molto facile. Da qui possiamo lentamente complicare il raggiungimento del traguardo di modo che l'agente riesca ad ottenere il massimo in tutte le situazioni. Per definire un curriculum, dobbiamo prima specificare quante fasi utilizzare e quali parametri del gioco dovranno variare all'aumentare della difficoltà; la scelta è ricaduta su:

- i punti vita dell'agente: nella prima fase sono 20, dalla seconda fase in poi variano casualmente tra $[5, 20]$;
- i punti vita del nemico: nella prima fase ha solamente 1 HP, nella seconda aumentano a 10 e dalla terza in poi variano casualmente tra $[10, 20]$;
- il numero degli oggetti raccoglibili: nelle prime tre fasi è uguale al 20% delle caselle libere sulla stanza, nella quarta varia casualmente tra $[10\%, 20\%]$ e dalla quinta varia tra $[5\%, 20\%]$.

Nel nostro caso il cambiamento tra le varie fasi avviene dopo aver superato un certo numero di passi:

- la prima fase è quella iniziale;
- la seconda fase avviene dopo 2.5×10^6 passi;
- la terza avviene dopo 2×10^6 passi dalla seconda;
- la quarta avviene dopo 1.5×10^6 passi dalla terza;
- la quinta avviene dopo 1×10^6 passi dalla quarta.

Le statistiche degli attori rimangono fisse per tutta la durata dell'allenamento. Questi parametri possono eventualmente essere cambiati dal programmatore come viene specificato nella sezione successiva. Grande importanza ha il comportamento del nemico: per soddisfare tutti i requisiti definiti nel capitolo 3.3 è stato scelto un attore che esegue mosse del tutto casuali; questo permette all'agente di vedere tutte le possibili azioni che un utente può effettuare, e nel contempo permette di allenarsi contro un nemico limitato rispetto alle capacità umane rendendo il sistema battibile con la giusta strategia. È stato tuttavia provato anche un approccio diverso degno di nota,

come già presentato nella fig. 3.5: in una particolare fase del lavoro abbiamo provato ad addestrare l'agente in più fasi contro delle vecchie versioni di sè stesso; questo procedimento ha portato la creazione di un attore troppo attento, molto difficile da battere ma poco divertente, andando contro gli obiettivi e i principi di gioco imprescindibili per questo lavoro.

4.5 Parametri di Allenamento

I valori delle statistiche degli attori possono essere modificati prima di iniziare l'allenamento. Questi parametri non vengono osservati dall'agente, ma permettono un facile modo per creare diversi comportamenti: cambiando i valori di ATK, DEX e DEF del sistema, questo cercherà di effettuare le giuste azioni che meglio si adattano alle sue caratteristiche. In particolare, nel nostro caso, abbiamo creato tre tipologie di nemici:

- *Arciere*: ATK = 0, DEX = 4 e DEF = 3;
- *Guerriero*: ATK = 4, DEX = 0 e DEF = 3;
- *Ranger*: ATK = 3, DEX = 3 e DEF = 3;

Per semplicità, abbiamo usato un nemico con sempre le stesse caratteristiche: ATK = 3, DEX = 3 e DEF = 3. Eventualmente è possibile cambiare la strategia dell'agente tramite la modifica della reward function, ad esempio è ragionevole ricompensare specifici comportamenti in modo da cambiare la tattica del sistema.

4.6 Complessità: Rete e Addestramento

L'addestramento degli agenti è stato eseguito tramite una GPU Nvidia GeForce 1050 ti, con 4GB di RAM; con questa configurazione, allenare un agente fino alla saturazione del suo reward ha richiesto mediamente due giorni, ma tutte e tre le tipologie di agente sono state addestrate nello stesso momento: la dimensione ridotta delle reti permette infatti di allenare più sistemi in parallelo, anche con una GPU di medio profilo. Il numero totale dei pesi che devono essere imparati dalla policy è:

- *primo ramo*:
  - embedding = 384 parametri
  - convoluzione = 9248 parametri
  - convoluzione = 18496 parametri
- *secondo ramo*:
  - embedding = 384 parametri
  - convoluzione = 9248 parametri
  - convoluzione = 18496 parametri
- *terzo ramo*:
  - embedding = 384 parametri
  - convoluzione = 9248 parametri
  - convoluzione = 18496 parametri
- *quarto ramo*:
  - embedding = 4608 parametri

  - FC = 180480 parametri

- FC = 2261248 parametri
- LSTM = 542720 parametri
- Softmax = 4369 parametri
- ***totale*** = 3077809 parametri

Mentre la baseline ha lo stesso numero di variabili eccetto per gli ultimi due strati, per un totale di 2596769. La somma complessiva dei parametri di entrambe le reti è dunque:

$$total_parameters = 5674578. \tag{4.2}$$

Infine, il modello finale addestrato richiede circa 12MB di memoria; ricordiamo però che più agenti del medesimo tipo possono usare lo stesso modello: il sistema dunque non scala con il numero di nemici, ma solamente con le diverse classi che vogliamo creare e/o utilizzare.

In questo capitolo abbiamo dunque descritto tutte le caratteristiche del modello di DRL con cui sono stati creati tre agenti differenti, soddisfacendo tutti i requisiti della tesi. Come più volte descritto, la scelta di questi parametri è frutto di numerosi esperimenti che alla fine hanno portato ad un buon compromesso tra risultato quantitativo e qualitativo, ma per poter fare dei ragionamenti a riguardo è stato necessario confezionare un prodotto che avesse tutte le caratteristiche di un vero videogioco; nel prossimo capitolo verrà presentato dunque il primo prototipo di DeepCrawl.

Capitolo 5

DeepCrawl: Prototipo

In questo capitolo verrà presentata l'implementazione di *DeepCrawl*: il prototipo finale del videogioco per sistemi iOS e Android che mette insieme tutti i concetti visti fino a questo momento. Fin'ora sono state descritte delle semplici meccaniche di gameplay limitate ad una mappa 10×10 e sono stati definiti agenti con un modello che permette loro di risolvere il relativo ambiente e giocare contro un utente umano. Tutto questo però non basta a definire un prototipo di videogioco: è stato infatti necessario aggiungere delle componenti indipendenti dal DRL ma che contribuiscono alla definizione di un prodotto ludico. Il capitolo inizia con la breve descrizione delle tecnologie utilizzate per l'implementazione e procede con l'analisi degli elementi e delle meccaniche che definiscono il prodotto finale.

5.1 Tecnologie Usate

DeepCrawl è stato sviluppato attraverso Unity [39], un motore grafico cross-platform basato su C# per la creazione di videogiochi 3D/2D o altri contenuti multimediali. In questo particolare caso, DeepCrawl è stato svi-

luppato per i sistemi operativi iOS e Android. La scelta di Unity è stata fatta principalmente per la disponibilità del framework Unity Machine Learning Agents Toolkit (ML-Agents) [21], un plugin open-source che consente ai giochi e alle simulazioni create con il motore di servire come ambienti di addestramento per agenti di DRL. Il framework permette ad altre librerie Python, ad esempio TensorForce, di interagire direttamente con Unity ed utilizzare gli algoritmi implementati utilizzando un videogioco creato indipendentemente in C#. Inoltre, il toolkit ML-Agents offre la possibilità di utilizzare dei grafi pre-addestrati direttamente all'interno del gioco, grazie a TensorFlowSharp [38]. Il plugin consente di effettuare l'addestramento tramite TensorForce, di esportare il grafo creato con TensorFlow e così di importare quest'ultimo all'interno del progetto Unity in modo da poterlo distribuire sulle piattaforme mobile.

Tutti gli asset grafici di gioco, tra cui i modelli 3D, i materiali, gli shader e altro, sono stati presi dallo Unity Asset Store, a cui poi sono state fatte leggere modifiche.

DeepCrawl è stato implementato per la maggior parte seguendo il pattern architetturale Entity-Component-System (ECS), tipico dello sviluppo dei videogiochi. Questa struttura segue il principio della *composizione a favore dell'ereditarietà* e fornisce grande flessibilità e riuso delle componenti software. L'ECS è composto fondamentalmente da tre oggetti:

- *Component*: un componente è semplicemente una struttura dati; non ha nessun metodo e la sua unica responsabilità è quella di memorizzare delle informazioni. Ogni componente descrive qualche aspetto di un'entità;

- *Entity*: un'entità è un oggetto che esiste nel mondo di gioco (personaggio, casella, muro, etc...). Essa è composta semplicemente da una lista

di componenti: l'insieme di questi costituisce le caratteristiche di una particolare entità. I componenti possono essere aggiunti e/o rimossi per definire delle responsabilità dinamiche;

- *System*: i sistemi sono gli unici oggetti che si preoccupano della logica del gioco. Ciascuno di essi opera su determinati gruppi di componenti che appartengono alla stessa entità; i sistemi vengono aggiornati ogni frame in parallelo.

In fig. 5.1 è mostrato un esempio di ECS applicato in DeepCrawl. Il prototipo utilizza un metodo strutturale ibrido, in cui la parte di gameplay è interamente amministrata dall'ECS, mentre la creazione e gestione della mappa di gioco utilizza un approccio Object Oriented.

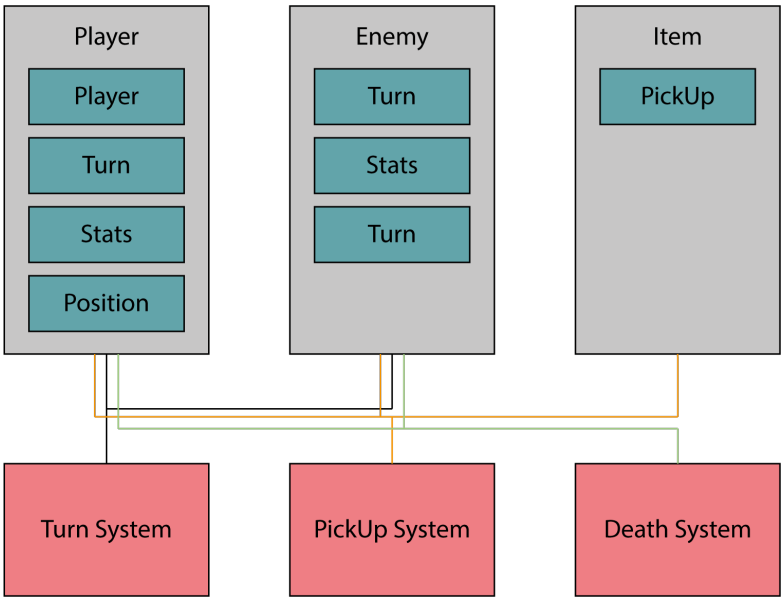


Figura 5.1: Esempio di ECS relativo a DeepCrawl: in grigio è possibile vedere le entità, che sono formate da uno o più elementi blu, che rappresentano i componenti. I sistemi, in rosso, agiscono solamente sui componenti di specifiche entità.

5.2 Progresso e Modalità di Gioco

Tendenzialmente i videogiochi di tipo RogueLike sono caratterizzati da un progresso che identifica una difficoltà sempre crescente e dall'esplorazione della mappa di gioco. Abbiamo deciso di mantenere queste meccaniche classiche e di creare un dungeon procedurale e progressivo con una struttura a livelli: ogni piano sarà composto da più stanze e ad ogni profondità ne verrà aggiunta una nuova da esplorare, partendo dal livello 1 con solamente una camera. In ognuna di esse sarà presente uno o più agenti da affrontare, e una volta sconfitti tutti i nemici in ogni stanza il giocatore passerà al livello successivo. All'inizio di ogni piano l'utente inizierà la sua esplorazione in un hub centrale in cui è presente un altare (fig. 5.2). Questo oggetto, a seconda della modalità di gioco scelta prima di intraprendere la discesa, fornisce un certo numero di punti da attribuire a libera scelta a tutte le caratteristiche del personaggio (HP, ATK, DEX e DEF). L'incremento di queste permette al giocatore di fronteggiare l'aumento delle statistiche degli agenti: ad ogni livello infatti tutte le caratteristiche dei nemici aumentano di 1 punto, a differenza degli HP che crescono di 2 unità. Questo fattore non influenza il comportamento dei nemici: come è stato presentato nel capitolo 4 infatti le caratteristiche non cambiano le osservazioni degli agenti ed essi mantengono il loro comportamento indipendentemente. Il progresso ha anche l'obiettivo di personalizzare l'esperienza utente e renderla in qualche modo unica ad ogni sessione: il giocatore infatti può scegliere la sua strategia di gioco e aumentare una determinata caratteristica piuttosto che un'altra per personalizzare le capacità del suo personaggio. Le modalità accennate prima sono tre, ognuna definisce il numero di punti che vengono offerti al giocatore per livello:

- *Modalità Facile*: ad ogni livello vengono assegnati 5 punti;

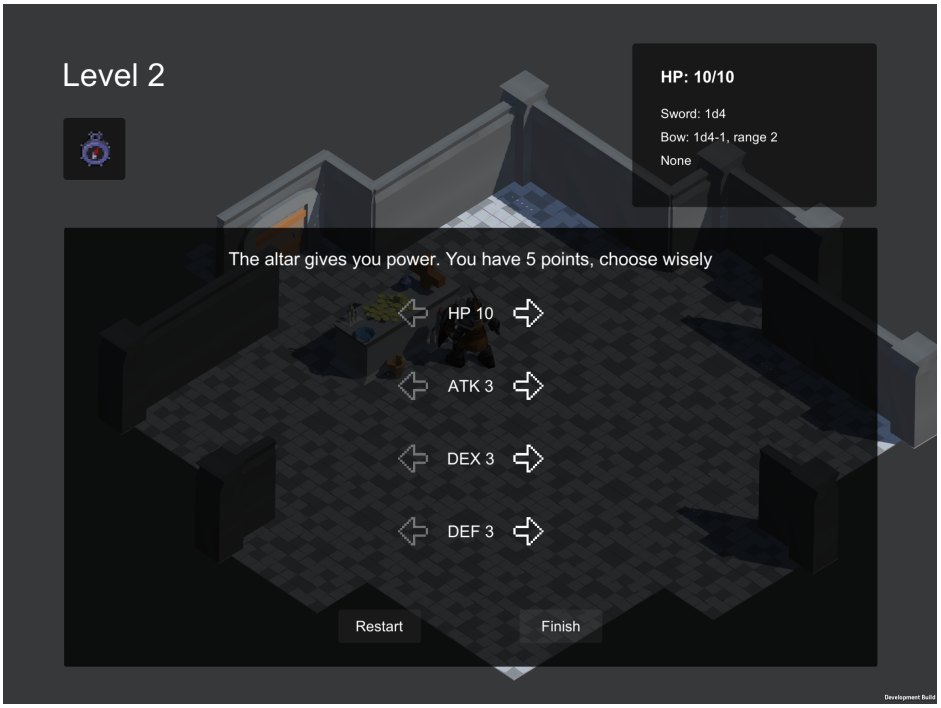


Figura 5.2: Screenshot del gioco che mostra il funzionamento dell'altare. In figura è mostrata la modalità facile, che offre all'utente 5 punti da distribuire alle caratteristiche. Questo passaggio è una parte fondamentale del progresso di gioco.

- *Modalità Media*: ad ogni livello vengono assegnati 4 punti;
- *Modalità Difficile*: ad ogni livello vengono assegnati 3 punti;

Indipendentemente dalla difficoltà scelta, il giocatore inizia con 3 punti già assegnati ad ogni caratteristica e con 10 HP.

5.3 Creazione del Dungeon

Il labirinto rappresenta una parte fondamentale dell'esperienza di gioco di DeepCrawl. L'algoritmo con cui viene creato è piuttosto semplice e prende spunto dal titolo già citato The Binding of Isaac [19] (fig. 5.3):

1. Sia $R = level \times level$ una matrice di stanze, dove $level$ = numero del livello;
2. Sia $R[level/2, level/2]$ = la stanza iniziale e $count = 1$;
3. Fintanto che $count <= level$
   (a) Sia $current_room$ = una stanza casuale già creata;
   (b) Se $current_room$ non ha vicini liberi, torna ad (a);
   (c) Sia $next_room$ un vicino libero casuale nelle 4 direzioni Nord, Sud, Est e Ovest di $current_room$;
   (d) Sia $R[next_room]$ = nuova stanza;
   (e) $count++$;

La forma e l'orientamento delle singole stanze è determinato casualmente da una serie di template predefinito, mentre gli oggetti invalicabili e raccoglibili sulla mappa sono generati in posizioni casuali; il numero di questi varia

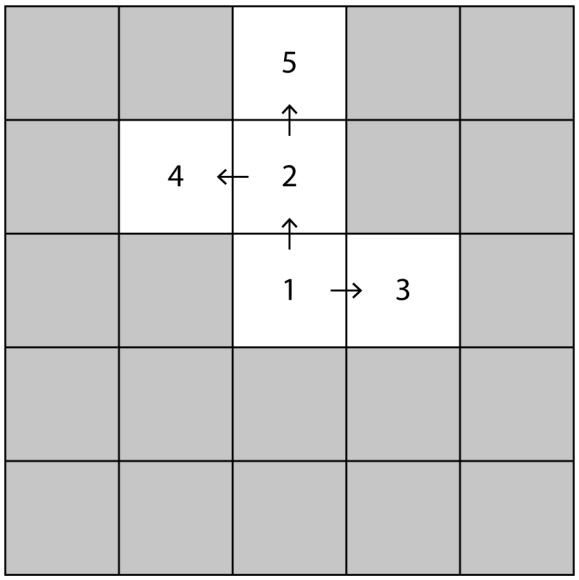


Figura 5.3: Esempio di creazione del dungeon relativo al livello 5.

sempre tra il 5% e l'8% delle caselle libere. In ogni stanza è presente uno o più nemici, numero definito secondo la seguente formula:

- Se livello < 3, è presente sempre 1 nemico;
- Se $3 <=$ livello < 6 e la stanza ha il 70% delle caselle totali libere, allora sono presenti 2 nemici, altrimenti viene generato 1 nemico;
- Se livello $>= 6$ e la stanza ha il 80% delle caselle totali libere, allora sono presenti 3 nemici; altrimenti, se la stanza ha il 60% delle caselle totali libere, allora sono presenti 2 nemici; altrimenti viene generato 1 nemico.

I nemici vengono creati non appena il giocatore varca la soglia della stanza e vengono generati nel lato opposto a quello di entrata dell'utente. Gli agenti rimangono attivi solo all'interno della propria stanza: non appena il giocatore esce da questa, il nemico cessa le sue attività per riprenderle non appena

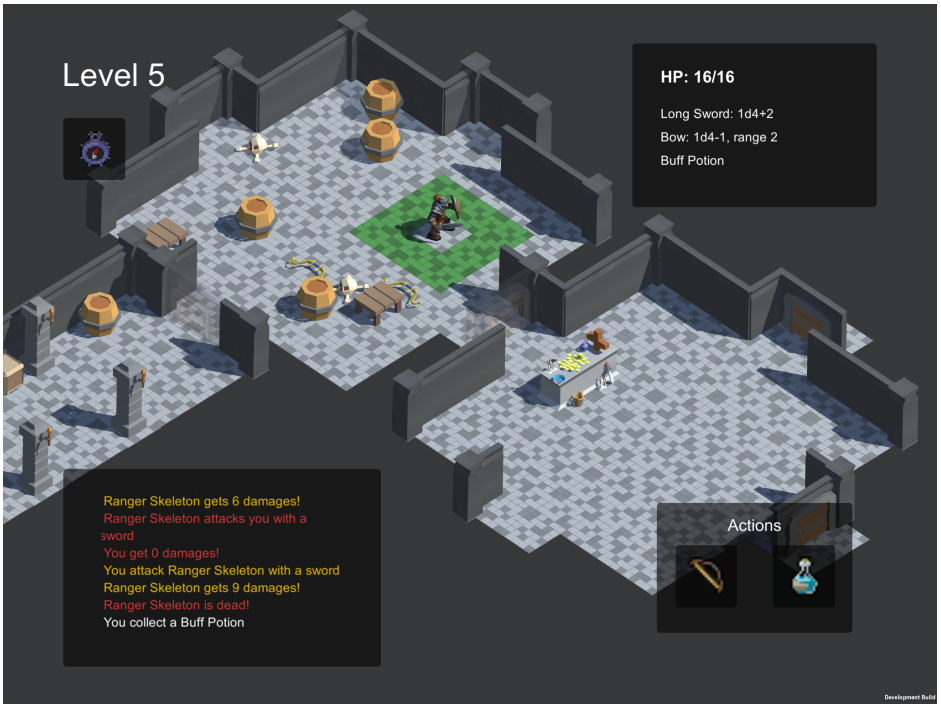


Figura 5.4: Screenshot di una partita avanzata. Come è possibile vedere, il giocatore è arrivato al livello 5 e mancano ancora delle stanze da esplorare: finchè l'utente non prova ad entrare in una di esse, non può vedere cosa è presente all'interno.

l'utente rientra in quel luogo. Fintanto che una particolare stanza non è stata esplorata la prima volta, rimarrà oscura al giocatore. In fig. 5.4 è mostrato un esempio di partita avanzata.

5.4 Interfaccia

In fig. 5.5 è mostrata nel dettaglio l'interfaccia utente, composta dai seguenti elementi:

Figura 5.5: Dettaglio degli elementi dell'interfaccia di gioco.

1. in alto a sinistra è specificato il livello corrente. Sotto ad esso, è presente un tasto che centra la telecamera nella posizione del personaggio;

2. in alto a destra è presente l'inventario del personaggio: i suoi HP e l'equipaggiamento salvato;

3. in basso a sinistra è presente una piccola sezione in cui vengono descritte tutte le mosse fatte sia dai nemici che dal giocatore. Se l'utente esegue un longpress su un qualsiasi oggetto e/o nemico, in questa sezione appariranno le caratteristiche dell'entità selezionata;

4. in basso a destra sono presenti due tasti che permettono di utilizzare l'attacco a distanza o la pozione nell'inventario;

L'applicazione permette tutti i comandi tipici di un ambiente mobile, ad esempio lo swipe per muovere la telecamera e il pinch-to-zoom.

Con l'analisi dell'interfaccia si è conclusa anche la descrizione del prototipo di DeepCrawl: siamo riusciti a creare un videogioco a tutti gli effetti partendo da delle semplici meccaniche di gameplay e da degli agenti già allenati per giocare contro l'utente. Con il prototipo è stato quindi possibile eseguire dei test, per valutare sia la qualità dell'intelligenza artificiale creata sia l'apprezzamento generale di DeepCrawl e dei suoi componenti.

Capitolo 6

Risultati

In questo capitolo verranno presentati al lettore tutti i risultati ottenuti sia dall'allenamento che dai test di giocabilità effettuati su DeepCrawl. In generale, abbiamo ottenuto degli esiti molto soddisfacenti sia dal punto di vista quantitativo che della qualità del videogioco; verranno quindi presentati e ne verrà offerta un'interpretazione, in particolar modo per quanto riguarda l'aspetto qualitativo del prototipo.

6.1 Risultati Quantitativi

Come abbiamo ampiamente descritto nel corso della tesi, in un lavoro di questo genere la valutazione qualitativa è in generale più importante di quella quantitativa: è necessario che l'agente riesca a massimizzare il suo score, ma in definitiva deve avere un comportamento che sia credibile e divertente dal punto di vista dell'utente. Durante la fase di allenamento degli agenti tuttavia è stato molto importante osservare l'evolversi del reward, in modo da essere sicuri che il sistema stesse effettivamente imparando dalla sua interazione con l'ambiente.

Nelle fig. 6.1a, 6.2a e 6.3a sono rappresentati i grafici della somma dei reward ottenuti dall'agente all'interno di un episodio:

$$R_e = \sum_t R(t), \tag{6.1}$$

dove t rappresenta i passi eseguiti nel corso di un singolo gioco ed R è la reward function definita in 4.1; le ricompense cumulative vengono poi mediate per 1000 episodi. Le linee tratteggiate fanno riferimento alle varie fasi del curriculum learning spiegato nella sezione 4.4. Come è possibile vedere per tutte le tipologie, nel primo stadio gli agenti imparano molto velocemente a interagire con l'ambiente e diventano consapevoli di quello che possono fare e non possono fare, poiché è possibile notare un aumento sostanziale del reward in un numero relativamente basso di passi. Ad ogni passaggio di curriculum invece il task cresce di complessità e il miglioramento degli agenti diventa sempre meno evidente, fino ad arrivare all'ultima fase in cui il sistema si allena fino ad una saturazione delle ricompense. Alla fine tutte e tre le categorie arrivano ad uno score molto alto, ma è interessante approfondire le varie differenze:

- la classe guerriero arriva più velocemente delle altre al suo massimo. Questo significa che la sua tipologia è la più facile da padroneggiare, probabilmente perché ha bisogno di poca strategia per risolvere lo scenario;

- l'arciere invece è la classe che arriva più tardi alla massimizzazione del reward ed è quella che ottiene il maggior valore. Questo fa intendere come sia la classe più difficile da conoscere appieno, ma è anche quella a cui il giocatore deve fare più attenzione;

- il ranger invece rappresenta una via di mezzo; il suo score diventa più grande rispetto a quello del guerriero impiegando però molti più passi

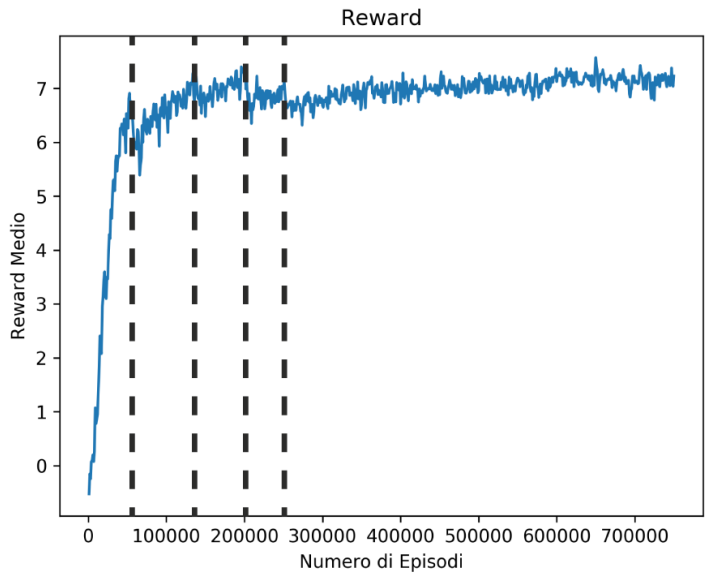
per arrivare ad una convergenza, mentre rimane comunque al di sotto del valore massimo dell'arciere.

Il successo dell'allenamento è evidenziato anche dai grafici 6.1b, 6.2b e 6.3b. Questi illustrano la variazione dell'entropia media della distribuzione di probabilità della scelta delle azioni all'interno di un episodio, mediate a loro volta su 1000 episodi. L'entropia di una distribuzione è definita come:

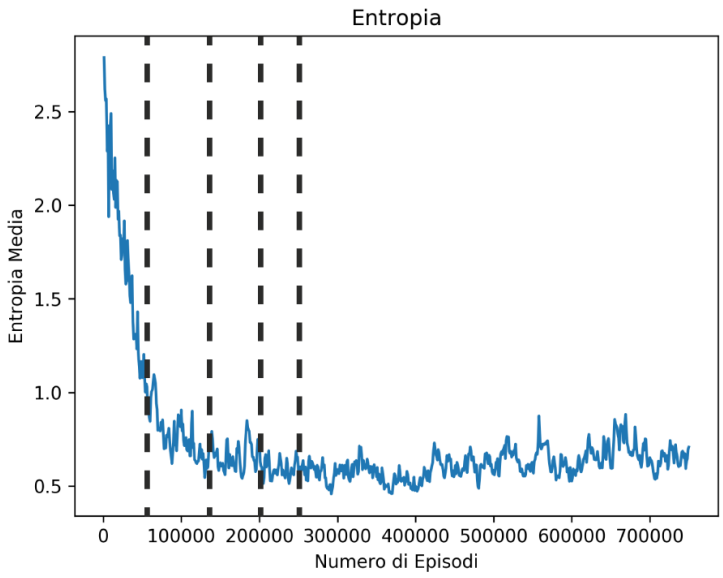
$$H = -\sum_{x} p(x) \log(p(x)), \tag{6.2}$$

dove, in questo caso, $p(x)$ rappresenta la probabilità di eseguire di una particolare azione. Valori alti di entropia corrispondono a delle incertezze degli agenti, in quanto significa che molte azioni hanno la stessa probabilità di essere effettuate; man mano che questa si abbassa, il sistema diventa sempre più sicuro e consapevole di quali sono le mosse giuste da eseguire. I grafici indicano come, all'aumentare dello score degli agenti, l'entropia diminuisce il suo valore dimostrando la buona riuscita dell'addestramento.

È interessante notare in fig. 6.4 la differenza tra un allenamento con e senza curriculum. Sebbene si tratti della classe più semplice da padroneggiare, anche nell'addestramento del guerriero è visibile un evidente miglioramento nell'utilizzo di questa tecnica: senza l'uso di un curriculum infatti l'agente accresce il suo reward molto più lentamente e senza mai arrivare al valore massimo raggiunto nell'altra modalità.

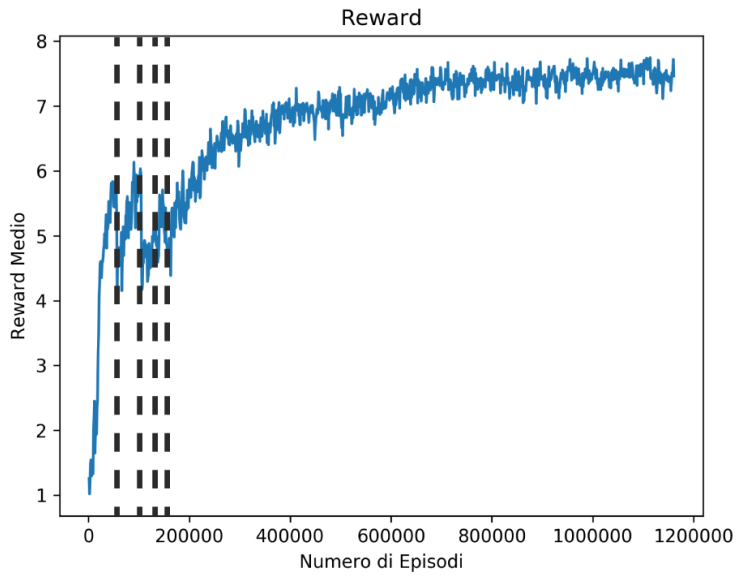


(a) Reward medio.

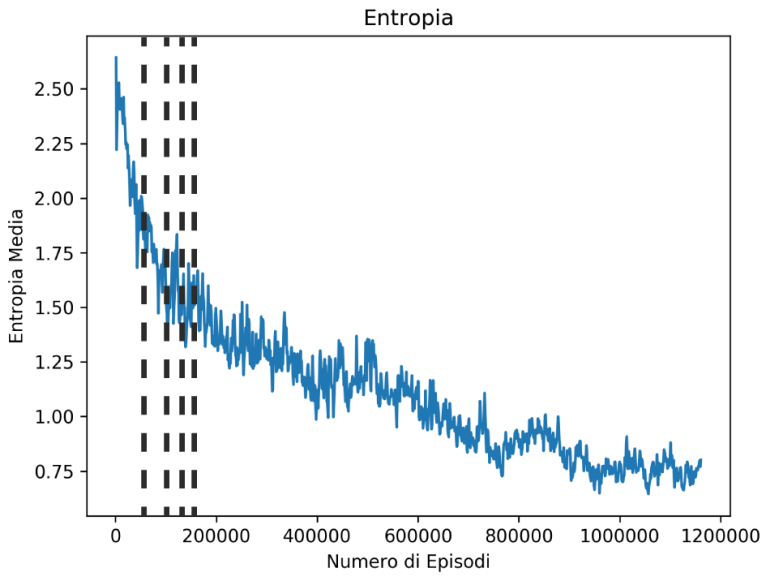


(b) Entropia media.

Figura 6.1: Risultati dell'addestramento della classe guerriero.

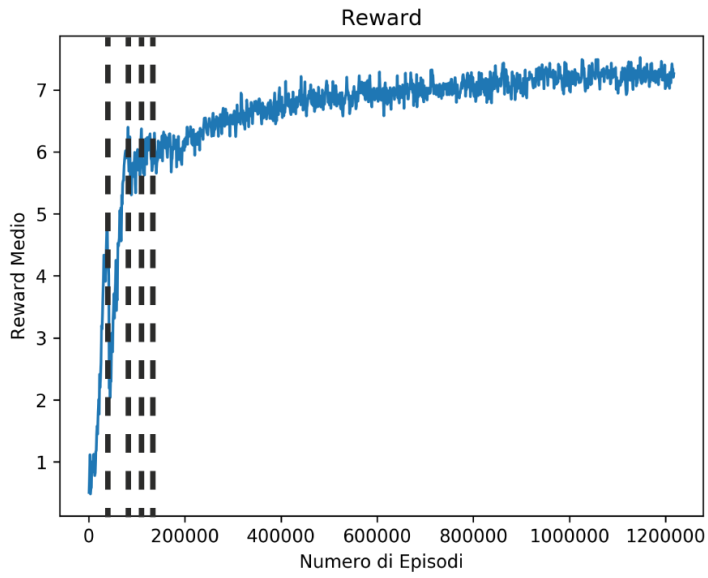


(a) Reward medio.

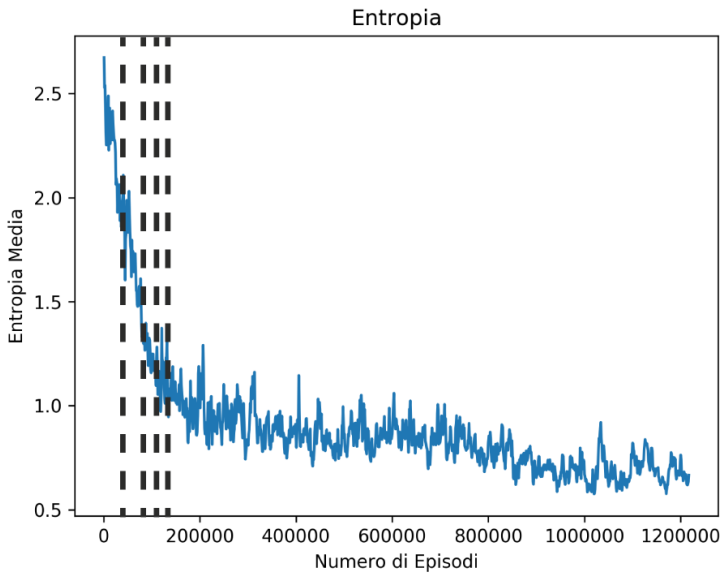


(b) Entropia media.

Figura 6.2: Risultati dell'addestramento della classe arciere.



(a) Reward medio.



(b) Entropia media.

Figura 6.3: Risultati dell'addestramento della classe ranger.

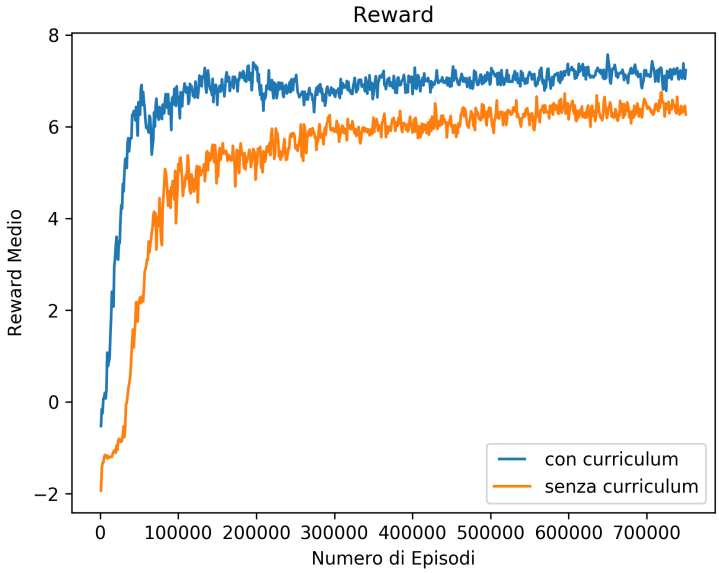


Figura 6.4: Differenza tra addestramento con e senza curriculum per la classe guerriero. È possibile notare come l'uso della tecnica aiuti a velocizzare l'addestramento e permetta all'agente di arrivare ad un valore di reward che altrimenti sarebbe difficile da raggiungere.

6.2 Risultati Qualitativi

Per valutare il risultato qualitativo del prototipo e dell'intelligenza artificiale sono stati condotti dei test di "giocabilità". La ricerca è stata effettuata su 10 candidati, tutti videogiocatori più o meno appassionati e che hanno piena conoscenza del mondo videoludico. Il test si è composto da una singola fase in cui è stato richiesto agli utenti di arrivare al livello 10 dopo aver familiarizzato con DeepCrawl; ad ognuno dei candidati è stato brevemente descritto il sistema di intelligenza artificiale presente nel videogioco. Ad ogni utente è stato fornito un dispositivo mobile (Android o iOS) in cui era già pre-installata l'applicazione; ogni giocatore ha iniziato la partita nella modalità facile. Durante la fase di test è stato inoltre chiesto ai candidati di commentare riguardo il comportamento dell'intelligenza artificiale e della qualità generale di DeepCrawl; le sessioni sono durate tra i 30 e i 60 minuti. Alla fine della prova è stato richiesto ad ogni videogiocatore di rispondere a 15 domande di tipo Single Ease Question (SEQ), tipiche dei test di usabilità di software. Il questionario era costituito da domande che riguardavano sia il comportamento degli agenti, ma anche la funzionalità dell'interfaccia e ovviamente la qualità ludica dell'applicazione; tutte le domande sono state pensate con l'obiettivo di capire se i requisiti definiti nel capitolo 3 erano stati soddisfatti. Nella tab. 6.1 sono riassunti i risultati.

N°	Domande	Media	σ
1	Ti sentiresti in grado di arrivare fino al livello 10?	5.54	1.03
2	All'aumentare del livello, i nemici sono sembrati troppo forti?	4.63	0.67
3	Ti sembra che i nemici siano intelligenti?	5.72	0.78
4	Ti sembra che i nemici seguano una strategia?	6.18	0.40
5	Ti sembra che i nemici facciano delle mosse controintuitive?	2.00	0.63
6	Le tipologie di nemico hanno lo stesso comportamento?	1.27	0.46
7	Il significato delle icone e delle scritte è facilmente comprensibile?	5.72	1.67
8	Le informazioni offerte dall'interfaccia sono chiare e sufficienti?	5.54	1.21
9	I livelli sono grandi e dispersivi?	2.00	1.34
10	Gli oggetti posti sulla mappa sono tutti ben visibili o hai avuto difficoltà a riconoscerli?	5.81	1.66
11	È utile leggere le caratteristiche di un nemico e/o oggetto (longpress)?	6.90	0.30
12	Quanto è importante avere una strategia?	6.90	0.30
13	Valuta in generale le capacità dei nemici rispetto ad altri giochi RogueLike	6.00	0.77
14	Il gioco è piacevole e divertente?	5.80	0.87
15	L'applicazione presenta molti bug?	1.09	0.30

Tabella 6.1: Tabella riassuntiva dei risultati del test SeQ di DeepCrawl.

Verranno analizzati i due principali aspetti che è possibile trarre visualizzando gli esiti del test: il comportamento degli agenti e la qualità del gioco e dell'interfaccia.

6.2.1 Comportamento degli Agenti

Come dimostra la tabella l'applicazione ha riscosso in generale grande successo, il gioco è stato apprezzato e ha offerto un'interessante e divertente sfida agli utenti. Il comportamento degli agenti ha contribuito in modo sostanziale alla buona qualità del prodotto: i giocatori hanno percepito gli agenti come esseri intelligenti, che seguono una determinata strategia e che generalmente non efettuano delle mosse controintuitive (tranne in qualche rara eccezione). L'intelligenza dei nemici ha reso necessario l'utilizzo di una tattica per poter risolvere i vari piani, rendendo l'esperienza molto divertente ed impegnativa. Sebbene nessun tester sia riuscito ad arrivare a livello 10, data anche la scarsità di tempo a disposizione, la maggior parte di essi è sicura di riuscire a finire il gioco dedicandoci la giusta attenzione: questo a dimostrare che i nemici danno la sensazione di essere ostici pur essendo comunque battibili. Molto apprezzata è stata anche la varietà offerta dagli agenti, che è risultata molto evidente sopratutto all'aumentare del livello:

- il guerriero tende ad andare sempre corpo a corpo, obbligando il giocatore ad evitare di essere attaccato per primo. Interessanti sono le sue azioni di movimento (fig. 6.5): gran parte delle volte tende ad avvicinarsi all'utente tentando di non entrare nel range di tiro di quest'ultimo, cercando di evitare gli attacchi a distanza che per un guerriero potrebbero essere fatali;

Figura 6.5: Esempio di comportamento della classe guerriero; è interessante notare come entrambi i guerrieri si siano posizionati lontano dall'area di tiro del giocatore.

Figura 6.6: Esempio di comportamento della classe arciere; in questo screenshot è possibile vedere come il nemico si sia tenuto lontano dall'utente per poter effettuare un attacco a distanza.

- è stato notato come l'arciere (fig. 6.6) abbia un comportamento opposto a quello del guerriero; questa classe è specializzata negli attacchi a distanza e dunque tenta sempre di avere il giusto spazio tra sè e il giocatore per effettuare dei danni con gli archi. In più, quando l'utente si avvicina, cerca di allontanarsi poiché è consapevole che in uno scontro ravvicinato potrebbe avere la peggio;

- il ranger è stata la classe che offre più varietà; a livelli alti rappresenta una sfida molto impegnativa, in quanto padroneggia entrambi gli attacchi ed è la categoria più imprevedibile.

Il successo degli agenti sviluppati con queste tecniche è dimostrato anche dal fatto che, paragonandoli ad altri nemici di giochi RogueLike che spesso non seguono nessun tipo di strategia, quelli proposti da questa tecnica risultano essere generalmente migliori.

6.2.2 Interfaccia e Implementazione

Gli esiti del test di giocabilità hanno evidenziato l'apprezzamento generale delle meccaniche di gioco; Il *look and feel* è stato giudicato positivamente e ricorda molto le classiche avventure RogueLike. La struttura a livelli è stata gradita ai più, che hanno visto un interessante fattore strategico nell'organizzazione a stanze: ad alti livelli è necessario esplorarle a fondo ed eventualmente tornare indietro nei posti già visitati per raccogliere gli oggetti che aiutano a completare il livello. I tester hanno percepito bene il senso del progresso tipico dei RogueLike, grazie alla personalizzazione delle caratteristiche che è possibile effettuare al completamento di un piano del labirinto. Dai risultati però si evince qualche incertezza sull'interfaccia di gioco: non tutti gli utenti hanno trovato gradevoli gli elementi ed alcuni diventavano chiari e comprensibili solamente dopo un certo numero di tentativi; in più, non sempre gli elementi di gioco (come gli oggetti raccoglibili) erano ben visibili nella mappa (fig. 6.7). Al contrario, la possibilità di leggere le caratteristiche di un nemico o un oggetto si è rivelata intuitiva e molto utile nelle dinamiche, ed è stata considerata un elemento fondamentale per poter passare ai livelli successivo.

Dai risultati quantitativi e qualitativi emerge dunque che tutti i requisiti definiti nel corso del lavoro sono stati soddisfatti: abbiamo creato delle tipologie diverse di agenti che riescono ad offrire una giusta sfida all'utente, risultando comunque battibili ma senza rivelarsi frustranti. La struttura a

Figura 6.7: Esempio di oggetto non propriamente visualizzato; nel cerchio bianco è evidenziato un collezionabile non visibile nel modo corretto sulla mappa di gioco.

livelli e l'organizzazione in stanze è risultata un fattore molto gradevole, evidenziando quelle sensazioni tipiche del genere RogueLike. Ci sono comunque molti margini di miglioramento, non tanto dal lato del DRL ma soprattutto sull'aspetto visivo di DeepCrawl e sulla varietà di situazioni che è possibile trovare. Nel prossimo capitolo verranno dunque presentati tutti i possibili lavori futuri che possono migliorare il lavoro svolto in questa tesi.

Capitolo 7

Conclusioni

In questa tesi è stato sperimentato un innovativo metodo di sviluppo di intelligenze artificiali in ambito videoludico tramite concetti di machine learning, in particolare attraverso il deep reinforcement learning. È stato quindi ideato un modello in grado di sviluppare degli agenti che possono giocare contro degli utenti umani in un videogioco di tipo RogueLike, modello con cui successivamente è stato creato un prodotto per piattaforme iOS e Android.

Il lavoro è iniziato con una breve analisi degli elementi teorici che formano il machine learning, il deep learning e il reinforcement learning, e come questi tre concetti si fondono insieme per formare il deep reinforcement learning. È stato descritto l'algoritmo DQN, capostipite del DRL, e il PPO, usato in questo lavoro. Sono state analizzate brevemente anche le tecniche classiche di intelligenza artificiale per lo sviluppo di videogiochi, come i behavior trees e le finite state machines.

Successivamente sono state presentate le meccaniche principali del contesto creato per l'occasione, motivando ogni particolare scelta di gameplay, e nel contempo sono stati definiti dei requisiti relativi agli agenti e al DRL. Ab-

biamo così descritto un quadro di riferimento sul quale applicare gli algoritmi studiati.

In seguito sono state definite le caratteristiche principali del modello di DRL: la rete neurale, gli iperparametri del PPO, la reward function e il set-up di training. Tutte le scelte sono frutto di numerosi esperimenti e sono state fatte tenendo conto dei requisiti definiti; tramite questo modello sono state create tre tipologie di agenti che riescono a risolvere il contesto precedente.

È stato quindi sviluppato DeepCrawl, un prototipo perfettamente funzionante di videogioco in cui i nemici sono guidati completamente da reti neurali. Sono stati poi effettuati dei test di giocabilità che hanno dimostrato il soddisfacimento dei requisiti e la buona qualità del prototipo, evidenziando come gli agenti agiscano in maniera credibile, intelligente e divertente.

Gli obiettivi della tesi sono stati pienamente soddisfatti; abbiamo dimostrato come il machine learning possa essere utilizzato all'interno dello sviluppo di un videogioco, ed è facile vedere come questo metodo sia migliore sotto molti punti di vista rispetto alle tecniche classiche:

- gli agenti non utilizzano una strategia dettata dal programmatore ma riescono ad inferire una tattica in maniera autonoma; questo rende i nemici tipicamente più intelligenti e imprevedibili rispetto ai normali giochi, e nella maggior parte dei casi questo porta ad un maggiore divertimento;

- una volta trovato il modello adatto al gioco, è possibile creare molteplici varietà di nemici semplicemente cambiando dei piccoli aspetti: i loro parametri, la reward function, oppure direttamente le azioni che possono compiere; l'algoritmo sarà in grado di definire la strategia migliore per la particolare tipologia di nemico. I modelli classici di AI nei videogiochi sono poco dinamici e scarsamente riusabili, soprattutto

quando si parla di molta varietà, per questo il DRL rappresenta un grande vantaggio rispetto alle tecniche finora utilizzate;

- il modello proposto non è necessariamente limitato al videogioco sviluppato in questo lavoro, ma potenzialmente può andar bene con qualsiasi RogueLike che mantenga le stesse caratteristiche generali di DeepCrawl: con poche modifiche l'algoritmo può adattarsi semplicemente ad altri prodotti.

Lo svantaggio principale è rappresentato dal fatto che il modello non è così semplice da trovare: è necessario effettuare molti esperimenti e procedere per piccoli passi, ed è importante definire a priori degli obiettivi che determinano un punto di riferimento per lo sviluppo, per non incorrere in continui fallimenti. Il crescente interesse in questo ambito potrebbe però aumentare il numero di modelli già definiti per tipologie di videogioco, in modo da poter prendere spunto e velocizzare la definizione del sistema; come abbiamo detto prima, infatti, l'esempio proposto da questo lavoro può essere semplicemente adattato per un qualsiasi gioco RogueLike turn-based.

Il successo del lavoro descritto in questa tesi pone le basi per numerose idee ed eventuali sviluppi futuri, di cui ne proponiamo qualche esempio:

- è possibile creare nuove tipologie di nemici, come ampiamente descritto precedentemente modificando qualche parametro senza cambiare l'algoritmo di DRL. La creazione di nuovi nemici aumenterebbe la varietà di situazioni;

- è possibile migliorare l'interfaccia utente dell'applicazione, poiché è emerso dai test di giocabilità che non è sempre chiara ed immediata;

- è possibile modificare la struttura della rete neurale: sebbene quella definita funzioni molto bene per questo contesto, è sempre possibile

tentare di migliorare il risultato provando nuovi strati e nuove strutture; stesso discorso vale per gli iperparametri del PPO, è possibile che diverse combinazioni possano portare ad un risultato diverso, anche se non necessariamente migliore;

- è possibile migliorare il prototipo e renderlo un videogioco completo: aggiungere meccaniche di gameplay, incrementare il numero di oggetti che è possibile raccogliere, aumentare la varietà grafica del labirinto, aggiungere effetti sonori e così via.

Inoltre un ragionamento approfondito può essere fatto riguardo la scalabilità generale del progetto: la rete neurale che è stata creata funziona bene in un ambiente relativamente limitato e ben definito come questo, ma lo sviluppo di un videogioco più complesso potrebbe portare all'aggiunta di nuove meccaniche di gameplay in itinere. L'utilizzo del sistema descritto porta inevitabilmente al riaddestramento completo degli agenti per ogni nuovo elemento inserito, ma potrebbe essere interessante approfondire alcuni approcci più complessi che possono rendere il lavoro più completo. Ad esempio, potrebbe essere utile investigare il concetto di *Meta-Learning* al'interno del DRL: in breve, è uno strumento che si basa sull'apprendimento multi-task per capire come adattare rapidamente un modello a nuove attività usando esperienze di compiti passati, cercando di "imparare ad imparare" e provando a rendere molto più veloce e meno costoso l'addestramento con l'aggiunta di nuove meccaniche. Inoltre potrebbe essere utile approfondire l'idea del *fine-tuning* di reti già addestrate, cercando di capire cosa e come un agente già istruito può imparare di nuovo solamente cambiando i pesi di alcuni strati del sistema. Un altro approccio possibile potrebbe basarsi sulla creazione di una tecnica che sia più modulare possibile, provando perfino a utilizzare più reti per differenti task: ad esempio, una rete si potrebbe preoccupare del

movimento dell'agente, un'altra di cercare l'equipaggiamento giusto, e così via. In questo modo è possibile categorizzare i vari compiti e, all'aggiunta di un nuovo elemento, potrebbe essere necessario solamente riallenare una singola sotto-rete.

Sempre sul concetto di generalizzazione del sistema, un'altra osservazione può essere fatta riguardo le modalità di testing: qualora ci fosse qualche bug nel comportamento degli agenti, nelle metodologie classiche è necessario cambiare determinate righe di codice e testare nuovamente quel singolo modo di agire per determinare se il problema è ancora presente o meno; con il DRL invece è possibile che il riaddestramento dell'agente, con la modifica qualche parametro, cambi completamente il comportamento dell'attore. Sarebbe interessante dunque approfondire e studiare, se esistono, delle tecniche di testing più efficaci per sperimentare nel miglior modo possibile la condotta dell'agente già allenato.

Bibliografia

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Kailash Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil A. Bharath. A brief survey of deep reinforcement learning. *CoRR*, abs/1708.05866, 2017.

[3] Andrea Asperti. Rogueinabox : an environment for roguelike learning. 2017.

[4] Andrea Asperti, Daniele Cortesi, and Francesco Sovrano. Crawling in rogue's dungeons with (partitioned) A3C. *CoRR*, abs/1804.08685, 2018.

[5] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *CoRR*, abs/1207.4708, 2012.

[6] Richard Bellman. On the theory of dynamic programming. *PNAS*, 38(8):716–719, 1952.

[7] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 41–48, New York, NY, USA, 2009. ACM.

[8] Deepmind. `https://deepmind.com`. Consultato: 06/01/2019.

[9] Doom - wikipedia. `https://it.wikipedia.org/wiki/Doom`. Consultato: 09/01/2019.

[10] Dota 2. `http://blog.dota2.com`. Consultato: 06/01/2019.

[11] Final fantasy tactics - wikipedia. `https://en.wikipedia.org/wiki/Final_Fantasy_Tactics`. Consultato: 09/01/2019.

[12] Michael C Fu. Gradient estimation. *Handbooks in Operations Research and Management Science*, 13:575–616, 2006.

[13] Peter W Glynn. Likelihood ratio gradient estimation for stochastic systems. *Communications of the ACM*, 33(10):75–84, 1990.

[14] Shixiang Gu, Timothy P. Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. *CoRR*, abs/1603.00748, 2016.

[15] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 2094–2100. AAAI Press, 2016.

[16] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Learning from demonstrations for real world reinforcement learning. *CoRR*, abs/1704.03732, 2017.

[17] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997.

[18] Hoplite - apps on google play. `https://play.google.com/store/apps/details?id=com.magmafortress.hoplite&hl=en_US`. Consultato: 09/01/2019.

[19] The binding of isaac: Rebirth. `http://bindingofisaac.com`. Consultato: 09/01/2019.

[20] Max Jaderberg, Wojciech M. Czarnecki, Iain Dunning, Luke Marris, Guy Lever, Antonio García Castañeda, Charles Beattie, Neil C. Rabinowitz, Ari S. Morcos, Avraham Ruderman, Nicolas Sonnerat, Tim Green, Louise Deason, Joel Z. Leibo, David Silver, Demis Hassabis, Koray Kavukcuoglu, and Thore Graepel. Human-level performance in first-person multiplayer games with population-based deep reinforcement learning. *CoRR*, abs/1807.01281, 2018.

[21] Arthur Juliani, Vincent-Pierre Berges, Esh Vckay, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A general platform for intelligent agents. *CoRR*, abs/1809.02627, 2018.

[22] Vijay R. Konda and John N. Tsitsiklis. On actor-critic algorithms. *SIAM J. Control Optim.*, 42(4):1143–1166, 2003.

[23] Alexander Kuhnle, Michael Schaarschmidt, and Kai Fricke. Tensorforce: a tensorflow library for applied reinforcement learning. Web page, 2017.

[24] Yann Lecun and Yoshua Bengio. *Convolutional networks for images, speech, and time-series*, page 3361. MIT Press, 1995.

[25] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[26] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[27] Openai five. `https://blog.openai.com/openai-five/`. Consultato: 06/01/2019.

[28] Pixel dungeon. `http://pixeldungeon.watabou.ru`. Consultato: 09/01/2019.

[29] Rogue (video game) - wikipedia. `https://en.wikipedia.org/wiki/Rogue_(video_game)`. Consultato: 09/01/2019.

[30] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[31] G. A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical Report TR 166, Cambridge University Engineering Department, Cambridge, England, 1994.

[32] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[33] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[34] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[35] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[36] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[37] Embedding | tensorflow. `https://www.tensorflow.org/guide/embedding`. Consultato: 06/01/2019.

[38] Tensorflowsharp: Tensorflow api for .net languages. `https://github.com/migueldeicaza/TensorFlowSharp`. Consultato: 17/01/2019.

[39] Unity. `https://unity3d.com`. Consultato: 17/01/2019.

[40] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy P. Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. Starcraft II: A new challenge for reinforcement learning. *CoRR*, abs/1708.04782, 2017.

[41] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279:292, 1992.

[42] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, 1992.

[43] Georgios N. Yannakakis and Julian Togelius. *Artificial Intelligence and Games*. Springer, 2018. `http://gameaibook.org`.