ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

STUDIO E SPERIMENTAZIONE DI ALGORITMI DI REINFORCEMENT LEARNING APPLICATI A VIDEO GAME

Elaborato in
Programmazione Di Applicazioni Data Intensive

Relatore
Prof. Gianluca Moro

Presentata da
Lorenzo Gardini

Terza Sessione di Laurea
Anno Accademico 2019 – 2020

PAROLE CHIAVE

Reinforcement Learning

Deep Neural Networks

Python

Stable Baselines

Gym

A mio fratello.

Introduzione

L'intelligenza artificiale (AI) è recentemente diventata uno dei principali trend tecnologici strategici grazie ad avanzamenti importanti nella ricerca scientifica del settore e nell'enorme sviluppo delle capacità di calcolo dei sistemi hardware.

Negli ultimi dieci anni la ricerca scientifica in ambito AI ha fatto progressi tali da rendere possibile sviluppare sistemi software che hanno risultati eclatanti in quasi ogni ambito delle discipline scientifiche.

Una delle differenze più importanti tra i metodi di AI classici e quelli recenti è che questi ultimi sono fortemente basati sull'estrazione di conoscenza da masse di dati e maggiore è la sua disponibilità, più la conoscenza estratta è accurata.

In particolare nell'ultimo periodo, ha iniziato a guadagnare molta popolarità una macro area di tecniche nota con il nome di **machine learning**.

I recenti successi del machine learning sono trasversali a numerosi settori tra cui *speech recognition* e *computer vision*.

L'utilizzo del machine learning può però portare a risultati non soddisfacenti quando si ha la necessità di gestire immagini o testi in forma naturale poiché l'attività di preparazione dei dati e di selezione delle variabili decisive per l'obiettivo finale richiede l'intervento di esperti ed è molto dispendiosa. In questi casi, infatti, si ottengono risultati generalmente più accurati e precisi attraverso tecniche diverse, appartenenti alla macro area di ricerca del **deep learning** che ha tra le sue prerogative quella di riuscire ad estrarre autonomamente le variabili migliori direttamente dai dati grezzi e di elaborare in maniera ancora più *approfondita* i dati.

All'interno di questo elaborato verranno analizzati e sperimentati diversi approcci recenti in ambito di deep learning (DL) e più nello specifico di Reinforcement Learning (RL), con l'obbiettivo di addestrare una rete a giocare al gioco arcade Pong sviluppato dalla casa Atari.

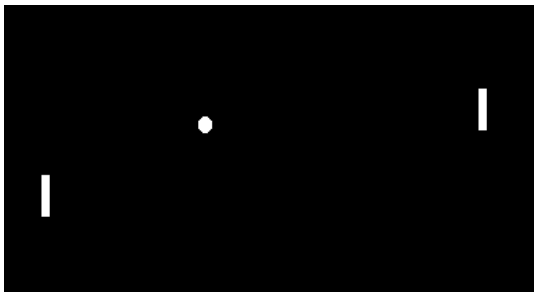

Figura 1: Gioco del Pong
Fonte: Andrej Karpathy blog,Deep Reinforcement Learning: Pong from Pixels

Il Reinforcement Learning è una tecnica di apprendimento automatico, ispirata dall'apprendimento animale, che punta a realizzare agenti autonomi (in questo caso la racchetta) in grado di scegliere azioni da compiere (andare su o giù) per il conseguimento di determinati obiettivi (fare goal nella porta avversaria) tramite interazione con l'ambiente in cui sono immersi (il campo di gioco).

Nella prima parte del mio lavoro viene presentato uno studio di una prima soluzione "from scratch" sviluppata da Andrew Karpathy. Seguono due miei miglioramenti: il primo modificando direttamente il codice della precedente soluzione e introducendo, come obbiettivo aggiuntivo per la rete nelle prime fasi di gioco, l'intercettazione della pallina da parte della racchetta, migliorando l'addestramento iniziale; il secondo é una mia personale implementazione utilizzando algoritmi più complessi, che sono allo stato dell'arte su giochi dell'Atari, e che portano un addestramento molto più veloce della rete.

Il lavoro di tesi è stato suddiviso nei seguenti capitoli:

- **Capitolo 1** - Introduzione al Machine Learning;
- **Capitolo 2** - Introduzione al Reinforcement Learning;
- **Capitolo 3** - Applicazione "from scratch" di Andrej Karpathy al gioco Pong;
- **Capitolo 4** - Versione di Pong con assegnazione di ricompensa al tocco della pallina sulla racchetta;
- **Capitolo 5** - Versioni di Pong in Stable Baselines.

Indice

Elenco delle figure

Capitolo 1

Introduzione al Machine Learning

1.1 Che cos'è il Machine Learning?

Il Machine Learning è la scienza (e l'arte) di programmare i computer in modo che possano imparare dai dati.

Ecco una definizione leggermente più generale:

> L'apprendimento automatico è il campo di studio che offre ai computer la capacità di apprendere senza essere programmati esplicitamente.
>
> - Arthur Samuel, 1959

E uno più orientato all'ingegneria:

> Si dice che un programma per computer impari dall'esperienza E rispetto a qualche compito T e qualche misura di prestazione P, se la sua prestazione su T, misurata da P, migliora con l'esperienza E.
>
> - Tom Mitchell, 1997

Il filtro antispam è un programma di Machine Learning che, dati esempi di email di spam (e.g. Segnalate dagli utenti) ed esempi di email regolari (non spam, chiamate anche "ham"), può imparare a segnalare lo spam. Gli esempi che il sistema utilizza per apprendere sono chiamati *set di addestramento*. Ogni esempio di addestramento è chiamato *istanza di addestramento (o campione)*. In questo caso, l'attività T è contrassegnare lo spam per i nuovi messaggi di posta elettronica, l'esperienza E è i dati di addestramento e la misura di prestazione P deve essere definita; ad esempio, si può utilizzare il rapporto tra le email classificate correttamente. Questa particolare misura delle prestazioni viene chiamata *accuratezza* ed è spesso utilizzata nelle attività di classificazione.

1.2 Perché utilizzare il Machine Learning?

Un filtro antispam scritto utilizzando la programmazione tradizionale segue le indicativamente i seguenti passi:

1. Innanzitutto considera l'aspetto tipico dello spam. Innanzitutto si considera l'aspetto tipico dello spam. Si possono prendere in considerazione alcune parole o frasi (come "x te", "carta di credito", "gratis" e "incredibile") che tendono a comparire spesso nella riga dell'oggetto; in aggiunta a schemi nel nome del mittente, nel corpo dell'email e in altre parti dell'email.

2. Si utilizzerebbe un algoritmo di rilevamento per ciascuno dei modelli identificati e il programma contrassegnerebbe le email come spam se venissero rilevati alcuni di questi modelli.

3. Testare il programma e ripetere i passaggi 1 e 2 finché non si ottengono i risultati necessari.

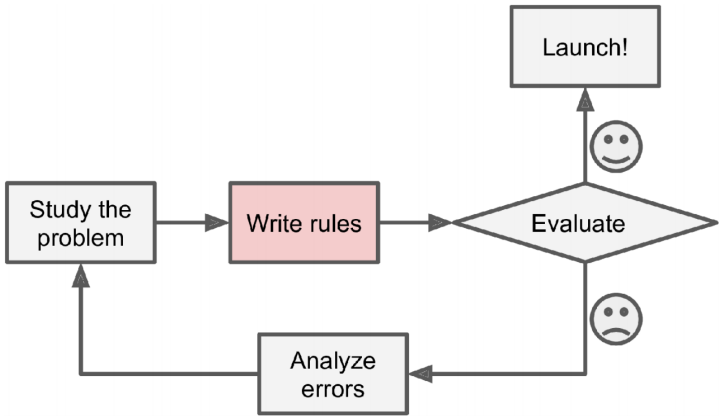

Figura 1.1: Approccio tradizionale
Fonte: Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems, Aurelien Geron [4]

Poiché il problema è difficile, il programma dovrà gestire un lungo elenco di regole complesse, difficile da mantenere. Al contrario, un filtro antispam basato su tecniche di Machine Learning apprende automaticamente quali parole e frasi sono buoni indicatori di spam rilevando schemi di parole frequenti negli esempi di spam rispetto agli esempi di ham. Il programma è molto più breve, più facile da mantenere e molto probabilmente più accurato.

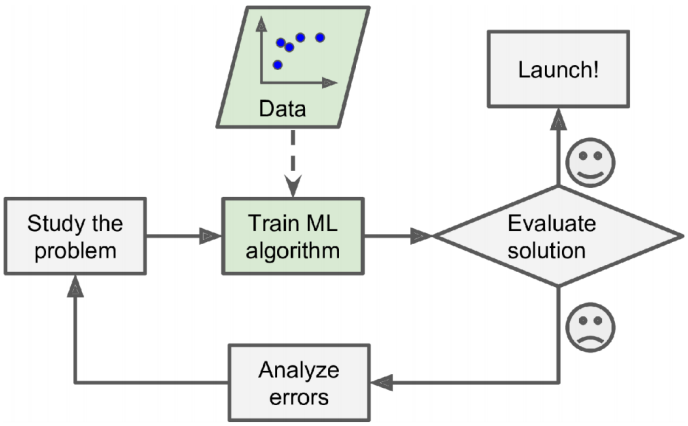

Figura 1.2: Approccio approccio Machine Learning
Fonte: Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems, Aurelien Geron [4]

Cosa succede se gli spammer notano che tutte le loro e-mail contenenti "x te" sono bloccate? Potrebbero iniziare a scrivere "x Te". Un filtro antispam che utilizza tecniche di programmazione tradizionali dovrebbe essere aggiornato per contrassegnare le e-mail "For U". Se gli spammer continuano a aggirare i filtri antispam, bisongna aggiornare o aggiungere nuove regole per sempre. Al contrario, un filtro antispam basato su tecniche di Machine Learning rileva automaticamente che "x Te" è diventato insolitamente frequente nello spam segnalato dagli utenti e inizia a contrassegnarli in automatico.

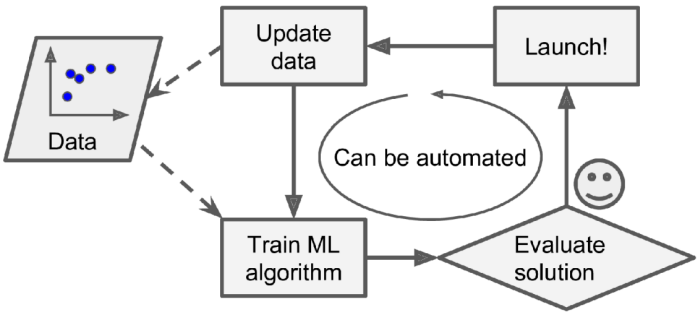

Figura 1.3: Adattamento automatico al cambiamento
Fonte: Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems, Aurelien Geron [4]

Capitolo 2

Introduzione al Reinforcement Learning

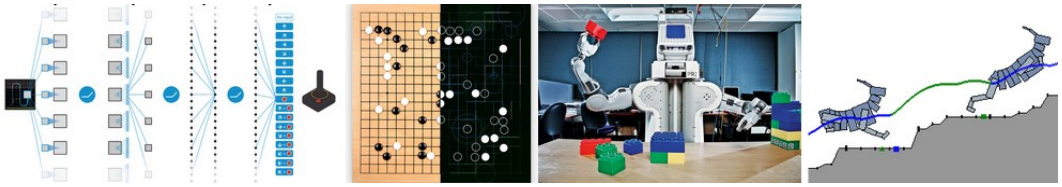

Figura 2.1: Esempi di RL. Da sinistra a destra : rete Deep Q Learning che gioca ad un gioco Atari, AlphaGo, robot che impilano Lego, la simulazione di un quadrupede che avanza saltando sul terreno.
Fonte: Andrej Karpathy blog,Deep Reinforcement Learning: Pong from Pixels

Il Reinforcement Learning (RL) è oggi uno dei campi più interessanti del Machine Learning e anche uno dei più antichi. È in circolazione dagli anni '50, producendo molte applicazioni interessanti nel corso degli anni, in particolare nei giochi (ad esempio, TD-Gammon, un programma per giocare a Backgammon) e controllo delle macchine. La rivoluzione ha avuto luogo nel 2013, quando i ricercatori di una startup britannica, chiamata DeepMind, hanno sviluppato un sistema che impara a giocare praticamente a qualsiasi gioco Atari da zero, superando infine gli umani nella maggior parte di essi, utilizzando solo pixel grezzi come input e senza alcuna precedente conoscenza delle regole. Altro loro risultato importante è avvenuto nel 2016 con la vittoria di AlphaGo contro Lee Sedol, un famoso giocatore professionista del gioco del Go, e nel maggio 2017 contro Ke Jie, il campione del mondo. Prima di allora, nessun programma si era mai avvicinato a battere un maestro di questo gioco. Oggi l'intero campo di RL ribolle di nuove idee, con un'ampia gamma di applicazioni. DeepMind è stata acquistata da Google per oltre $500 milioni nel 2014.

2.1 Che cos'è il Reinforcement Learning?

Il RL prende ispirazione e rispecchia l'apprendimento umano ed animale. Tutto si basa su problemi di assegnazione di credito e dilemmi esplorazione-sfruttamento, presenti anche nella vita quotidiana umana.

Il sistema di apprendimento, chiamato in questo contesto *agent (agente)*, può osservare l'*environment (ambiente)* in cui si trova, selezionare ed eseguire azioni e ottenere ricompense come loro conseguenza (o penalità sotto forma di ricompense negative). Deve quindi imparare da solo qual è la strategia migliore, chiamata *policy (politica)*, per ottenere la massima ricompensa nel tempo. Una policy definisce quale azione l'agente deve scegliere quando si trova in una determinata situazione.

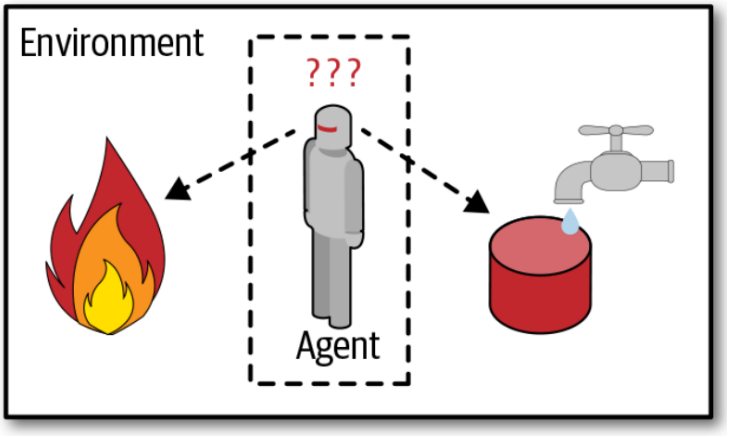

Figura 2.2: L'agent osserva l'environment e le azioni che può intraprendere.
Fonte: Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems, Aurelien Geron [4]

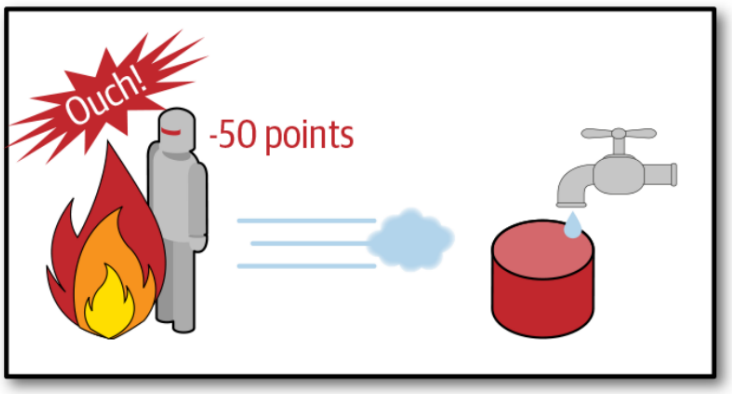

Figura 2.3: L'agent effettua una scelta sbagliata e riceve una ricompensa negativa
Fonte: Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems, Aurelien Geron [4]

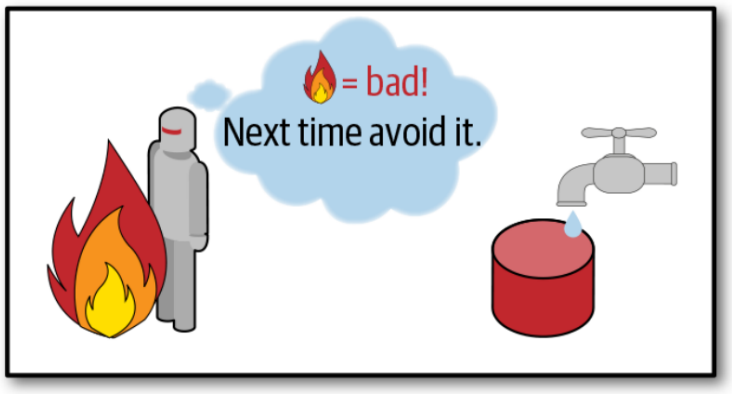

Figura 2.4: L'agent impara dall'azione appena effettuata per evitarla la prossima volta.
Fonte: Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems, Aurelien Geron [4]

Ad esempio, AlphaGo ha imparato la sua policy vincente analizzando milioni di giochi e poi giocando molte partite contro se stesso. L'apprendimento è stato disattivato durante le partite contro il campione; AlphaGo stava solo applicando la policy che aveva appreso [4].

2.2 Utilizzo di RL su Pong

Si supponga di voler insegnare a una rete neurale a giocare Pong. L'input per la tua rete sarebbero immagini dello schermo e l'output sarebbero tre azioni: su, giù o nessun movimento. Avrebbe senso trattarlo come un problema di

classificazione: per ogni schermata di gioco si deve decidere se effettuare uno spostamento in su, in giù o rimanere fermo. Per l'addestramento si ha bisogno solo di feedback occasionali sul fatto che la scelta effettuata è stata quella giusta e ricavare le altre conseguenze.

Questo è il compito che il RL cerca di risolvere. Si trova a metà strada tra l'apprendimento supervisionato e quello non supervisionato. Mentre nell'apprendimento supervisionato si ha un'etichetta target per ogni istanza del problema e nell'apprendimento non supervisionato non si ha alcuna etichetta, nell'apprendimento per rinforzo si hanno etichette sparse e che si ottengono con un certo ritardo. Le etichette equivalgono alle ricompense. Basandosi solo su quelle, l'agente deve imparare a comportarsi nell'ambiente.

Sebbene l'idea sia abbastanza intuitiva, in pratica ci sono diverse complicazioni. Ad esempio poco prima di subire un goal, le azioni che hanno seguito il superamento della racchetta da parte della pallina non hanno minimamente influenza sull'esito di quel round (si sa già di aver perso) oppure qual'è stata l'azione decisiva che ha portato a guadagnare un punto. Questo è chiamato *problema di assegnazione del credito*, ovvero quale delle azioni precedenti era responsabile dell'ottenimento della ricompensa (sia positiva che non) e in che misura.

Ora si pone un ulteriore problema: dopo aver individuato una strategia per raccogliere un certo numero di ricompense, conviene mantenerla e rinforzarla o sperimentare qualcosa che potrebbe portare a ricompense ancora più grandi? Questo è chiamato il dilemma *esplorazione-sfruttamento (explore-exploit)*.

Capitolo 3

Pong "from scratch": Sviluppo del videogame senza librerie di machine learning

Ora verrà illustrata la soluzione proposta da Andrej Karpathy, direttore dell'area di intelligenza artificiale e e di visione per pilotaggio automatico a Tesla, che ha presentato una soluzione in Python senza l'utilizzo di librerie di RL ma solo utilizzando la libreria matematica NumPy [1]. Per addestrare la rete ha utilizzato Policy Gradient (che verrà illustrato successivamente) invece di DQN in quanto, se impostato correttamente, permette di ottenere risultati migliori, come dimostrato da DeepMind [5] .

La piattaforma utilizzata è Gym che offre una raccolta di problemi di test, i cosiddetti *environments (gli ambienti)*, su cui testare algoritmi di RL. Nello specifico è stato utilizzato "Pong-v0". Per vincere una partita (chiamato *episode*) occorre che un giocare raggiunga 21 punti. Ogni volta che viene effettuato un punto il gioco viene resettato alla situazione iniziale con il punteggio di chi ha effettuato il punto incrementato di uno. L'ambiente acquisisce in ingresso l'azione che deve effettuare l'agente (in questo caso la racchetta di desta) e restituisce l'immagine con il nuovo stato, la ricompensa ottenuta dall'azione appena effettuata, un valore booleano che indica se l'episodio è concluso e infine un dizionario contenente informazioni utili per il debug. Le azioni possibili che può compiere l'agente sono *GIU* e *SU*.

Ora viene analizzato passo per passo il codice da lui usato.

3.1 Struttura della rete

Nel modello sono presenti due strati. Il primo, detto *hidden layer* è composto da 200 neuroni, mentre il secondo, detto di *output* è un singolo neurone.

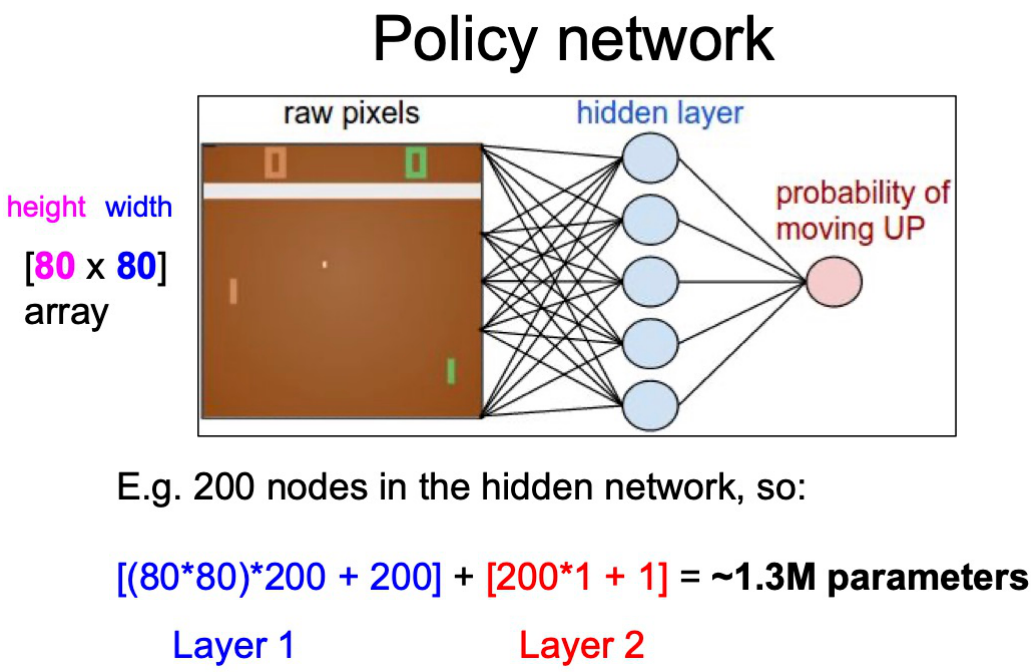

Figura 3.1: La semplice architettura della rete. Ogni cerchio colorato rappresenta un neurone. A fianco viene specificata la dimensione della matrice dell'immagine che viene passata alla rete; sotto colori delle scritte rappresentano il numero di parametri totali in ogni strato. Fonte: blog di Karpathy [1]

Fonte: Andrej Karpathy blog,Deep Reinforcement Learning: Pong from Pixels

Ogni strato si definisce *dense* perché i valori delle istanze x vengono utilizzate in ciascun neurone che effettua una regressione logistica.

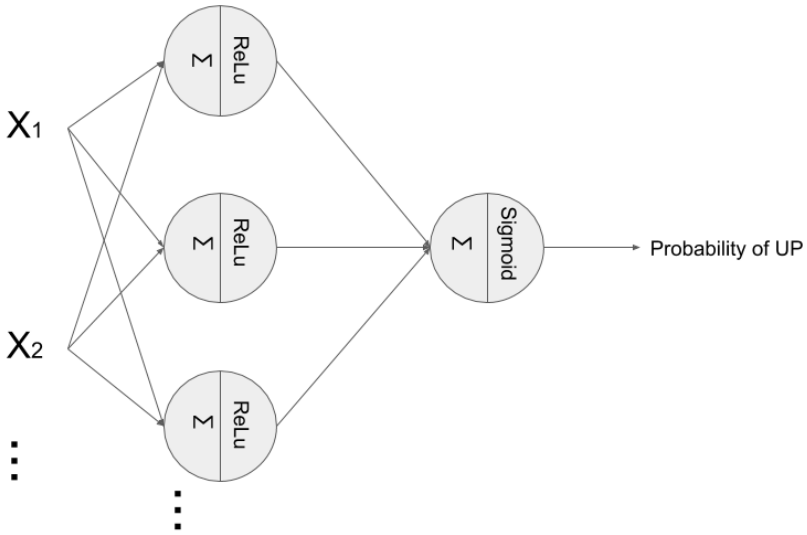

Figura 3.2: Ogni strato di neuroni è denso. In ogni neurone viene rappresentata la sequenza di operazioni che vengono svolte (per esempio nell'hidden layer prodotto scalare e poi applicazione della funzione di attivazione ReLu). I puntini di sospensione stanno ad indicare che l'input e il numero di neuroni della primo strato continua.

La regressione ha lo scopo di trovare i coefficienti dell'iperpiano separatore che massimizzi la separazione tra le classi, in questo caso SU e GIU. L'equazione sarà quindi nella forma

$$w^T x + b$$

dove w è il vettore contenente i coefficienti del singolo neurone, x sono i valori istanza del vettore immagine differenza e b è l'intercetta dell'iperpiano (detto bias). Nella prima riga viene effettuato il prodotto scalare dei 6400 coefficienti di ciascun neurone dello strato di input con il vettore immagine differenza a cui viene aggiunto il bias. Ogni coefficiente presente in ciascuno dei 200 neuroni è indipendente dagli altri.

Utilizzando come funzione di attivazione ReLu definito come

$$max(0, w^T x + b)$$

introducendo una non linearità. Infine l'ultimo strato, lo stato di output, riceve dallo strato superiore 200 valori che verranno moltiplicati per i 200 pesi a cui viene aggiunto il bias. Infine, si applica sigmoide che riduce la probabilità di uscita nell'intervallo $[0, 1]$.

Il primo e il secondo strato di neuroni vengono modellati rispettivamente con una matrice W1 grande $200 \times 6400 + 200$ biases e una matrice W2 grande $1 \times 200 + 1$ bias.

3.2 Inizializzazione dei pesi della rete

```
import numpy as np
import os.path, json_tricks

# neural model initialization with random values or with a previously saved neural model
# please mount your google drive with
#
# from google.colab import drive
# drive.mount('/content/drive')
#
# check whether exists a previously saved neural model to load (please create at least the following dir path)

def load_or_create_model(model_name):
  H = 200 # number of hidden layer neurons
  D = 80 * 80 # input dimensionality: 80x80 grid
  B = 1 # bias
  model = {} # rete neurale vuota, tabula rasa
  if os.path.exists(model_name):
    with open(model_name, "r") as infile:
      dictValues = json_tricks.load(infile)
      # assegna il modello neurale caricato da file a model
      model = dictValues
      print('retraining from a saved neural model')
  else : # non c'è un modello neurale su file quindi il modello è inizializzato con valori casuali
    model['W1'] = np.random.randn(H,D + B) / np.sqrt(D + B) # "Xavier" initialization
    model['W2'] = np.random.randn(H+B) / np.sqrt(H+B)
    print("model created")

  return model
```

Funzione per caricare un modello precedentemente creato da file o per crearne uno nuovo impostando le dimensioni della rete: il primo strato (hidden layer) da **H = 200** neuroni e **D = 80 * 80** come numero di pesi, il secondo strato con **H** pesi.

Si controlla se esiste un modello, precedentemente salvato, da caricare. Altrimenti inizializza i pesi dei due strati con valori utilizzando l'iniziazione Glorot(o Xavier).

Ma non basta inizializzare i pesi con valori random o zero?

Lo scopo dell'inizializzazione dei pesi è impedire che i valori di uscita dei neuroni diventino esageratamente grandi o si azzerino nel corso dei passaggi in avanti attraverso la rete [6]. Ne consegue che i gradienti saranno troppo grandi o troppo piccoli per la backpropagation e la rete impiegherà più tempo a convergere o non convergerà.

La moltiplicazione delle matrici è l'operazione matematica essenziale di una rete neurale. Nelle reti neurali con più strati, un passaggio in avanti comporta

l'esecuzione di moltiplicazioni di matrici consecutive su ogni strato, tra gli input di quel livello e la matrice dei pesi. Il prodotto di questa moltiplicazione su un livello diventa l'input del livello successivo e così via.

Per un esempio si ipotizzi di avere un vettore x che contiene alcuni input di rete. È prassi normale addestrare le reti neurali per garantire che i valori degli input siano ridimensionati in modo tale che rientrino in una distribuzione normale con una media di 0 e una deviazione standard di 1. Si consideri una semplice rete a 100 livelli con 100 neuroni l'uno,senza attivazioni, e che ogni livello è composto da una matrice A che contiene i pesi del livello. Per completare un singolo passaggio in avanti si esegue una moltiplicazione di matrici tra input e pesi di strato in ciascuno dei cento strati, il che comporterà un totale complessivo di 100 moltiplicazioni di matrici consecutive. L'inizializzazione dei valori dei pesi dei layer con distribuzione normale standard non è mai una buona idea. Da qualche parte durante quelle 100 moltiplicazioni, gli output dei layer sono diventati enormi.

```
x = np.random.randn(100)
x -= x.mean() #scalo i valori
x /= x.std()
check_val = 10000
flag = True #per la stampa

for i in range(100):
  A = np.random.randn(100,100)
  x = np.dot(A,x)
  if(flag and x.mean() > check_val):
    iterations = i
    flag = False
print("media: {} e deviazione standard: {} \nUn valore superiore a {} raggiunto in {} iterazioni".format(x.mean(),x.std(),check_val,iterations))
```

```
media: 2.0768525818029788e+98 e deviazione standard: 8.22073249601449e+99
Un valore superiore a 10000 raggiunto in 6 iterazioni
```

Analogamente bisogna considerare anche che gli output dei layer potrebbero azzerarsi. Per vedere cosa succede quando si inizializzano i pesi di rete in modo che siano troppo piccoli - con un ridimensionamento tale che i valori dei pesi, pur rientrando in una distribuzione normale con una media di 0, abbiano una deviazione standard di 0,01.

```
x = np.random.randn(100)
x -= x.mean() #scalo i valori
x /= x.std()

for i in range(100):
  A = np.random.randn(100,100) * 0.01
  x = np.dot(A,x)
print("media: {} e deviazione standard: {}".format(x.mean(),x.std()))
```

```
media: -1.3966795135600914e-101 e deviazione standard: 1.378085567137059e-100
```

Nei passaggi in avanti le uscite di attivazione sono diventate estremamente piccole. Per riassumere, se i pesi sono inizializzati troppo grandi, la rete non imparerà bene. Lo stesso accade quando i pesi sono inizializzati troppo piccoli.

Si può dimostrare che a un dato livello, il prodotto della matrice dei nostri input x e la matrice inizializzata con una distribuzione normale standard avranno, in media, una deviazione standard molto vicina alla radice quadrata del numero di connessioni di input, che in questo caso vale $\sqrt{100}$.

Per calcolare y si sommano 100 prodotti della moltiplicazione elemento per elemento degli input x per una colonna dei pesi A. Ne consegue quindi che la somma di questi 100 prodotti ha una media di 0, varianza di 100 e quindi una deviazione standard di $\sqrt{100}$.

```
std = 0.0
for i in range(1000):
  x = np.random.randn(100)
  A = np.random.randn(100,100)
  y = np.dot(A,x)
  std += y.std()
print("Varianza media: {} sqrt(100): {}".format(std/1000,np.sqrt(100)))
```

```
Varianza media: 9.90435772498651 sqrt(100): 10.0
```

Se si scala la matrice di peso A dividendo tutti i suoi valori scelti casualmente per $\sqrt{100}$, ogni moltiplicazione elemento per elemento in media ha una varianza di $\sqrt{100}$. Ora si può verificare che i valori di media e deviazione standard a seguito di moltiplicazioni successive rimangono stabili

```
x = np.random.randn(100)
x -= x.mean() #scalo i valori
x /= x.std()

for i in range(100):
  A = np.random.randn(100,100)/np.sqrt(100)
  x = np.dot(A,x)
print("media: {} e deviazione standard: {}".format(x.mean(),x.std()))
```

```
media: -0.026627981453673826 e deviazione standard: 0.4602710624112039
```

Vengono inoltre inizializzate a 0 due matrici con le stesse dimensioni di W1 e W2 che verranno utilizzate successivamente nella discesa del gradiente nell'algoritmo RMSProp.

```
def initialize_rsmprop(model):
  grad_buffer = { k : np.zeros_like(v) for k,v in model.items() } # update buffers that add up gradients over a batch
  rmsprop_cache = { k : np.zeros_like(v) for k,v in model.items() } # rmsprop memory
  return grad_buffer, rmsprop_cache
```

Le immagini che restituisce l'ambiente, che rappresentano il suo stato attuale, devono essere processate. La policy forward definisce quale azione deve scegliere la rete quando si trova in una determinata situazione, in questo caso il vettore della differenza di pixel pre-processati tra l'immagine precedente e quella attuale che per comodità si indicherà con x.

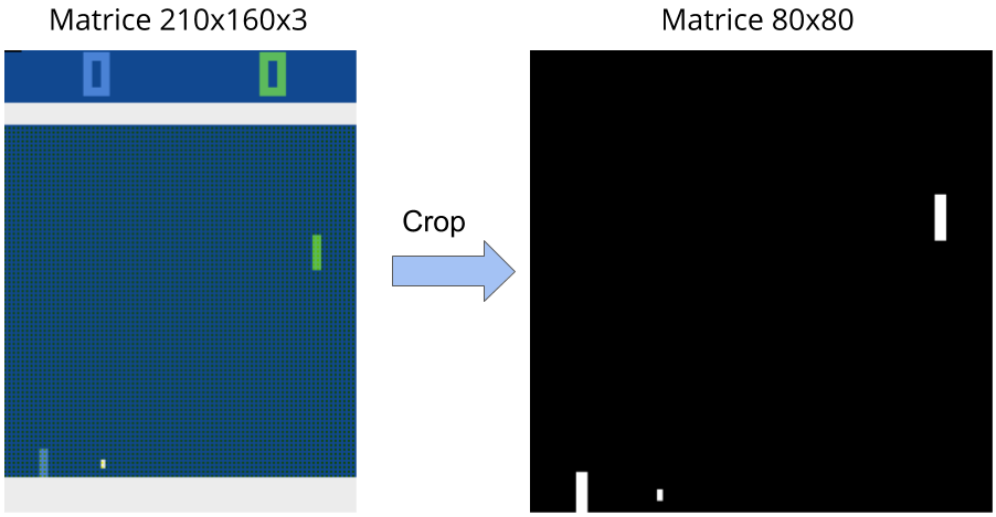

Figura 3.3: Per ogni immagine viene tolto lo sfondo, tagliati i bordi superiore e inferiore e viene mantenuto solamente un canale in cui i pixel di sfondo valgono 0 e quelli delle racchette e della pallina 1.

```python
# riduce la dimensione delle immagini e rimuove il background
def prepro(I):
  """ prepro 210x160x3 uint8 frame into 6400 (80x80) 1D float vector """
  I = I[35:195] # crop
  I = I[::2,::2,0] # downsample by factor of 2
  I[I == 144] = 0 # erase background (background type 1)
  I[I == 109] = 0 # erase background (background type 2)
  I[I != 0] = 1 # everything else (paddles, ball) just set to 1
  return I.astype(np.float).ravel()

def new_state(observation,prev_x = None):
  cur_x = prepro(observation)
  x = cur_x - prev_x if prev_x is not None else np.zeros(cur_x.size)
  prev_x = cur_x
  return x,prev_x
```

Questa funzione serve per pre-processare le immagini di ogni frame di gioco restituite dal modello grazie ai comandi **env.reset()** o **env.step()**. Da queste immagini, viste come tensori **altezza x larghezza x canali BGR** di dimensione **210x160x3**, viene mantenuto solo il primo canale (B), vengono rimosse la parte superiore contenente i punteggi di entrambi i giocatori e le barre divisiore del campo di gioco ottenendo una matrice **80x80**. Vengono poi azzerati i valori dei pixel corrispondendi allo sfondo mentre vengono posti a 1 i valori dei pixel di pallina e racchette. La matrice infine viene appiattita in un vettore contiguo lungo 6800 di tipo *float*.

La funzione **new_state** calcola la matrice da passare alla rete per la policy foreward come differenza tra il frame precedente e quello attuale già processati.

3.3 Policy Foreward

```
# funzione che a partire dal modello neurale appreso fino ad ora, restituisce, data un'immagine in input, l'azione
# da eseguire sulla racchetta, ossia di muoverla verso l'alto o verso il basso.
def policy_forward(x):
  # W1 è il layer di input della rete neurale con 80x80 neuroni. Attenzione in questo tutorial non sono utilizzate
  # librerie per reti neurali, infatti la rete neurale, la funzione di attivazione è tutto riprodotto matematicamente da zero
  # e.g. l'output del primo layer è dato semplicemente dal prodotti scalare del vettore x di input con il vettore
  # dei pesi corrispondenti W1 del primo layer.
  # Quindi h è l'output del layer di input a fronte dell'immagine differenza x fornita in input
  h = np.dot(model['W1'], x)
  # la ReLu è ottenuta banalmente con lo statement seguente, ossia restituisce 0 per valori negativi, quindi
  # lo statement seguente pone a zero i valori negativi prodotti dal primo layer
  h[h<0] = 0 # ReLU nonlinearity
  # l'output del primo layer viene passato in input al secondo layer
  # Bias nell'imput dell' output layer
  h_bias = np.append(h,1)
  logit = np.dot(model['W2'], h_bias)
  # il secondo layer impiega come funzione di attivazione la sigmoide che ha definito prima.
  p = sigmoid(logit)
  return p, h_bias # return probability of taking action 2, and hidden state
```

Intuitivamente, i neuroni nello strato nascosto (che hanno i loro pesi disposti lungo le righe di W1) possono rilevare vari scenari di gioco (ad esempio la palla è in alto e la paletta è nel mezzo) e i pesi in W2 possono quindi decidere se andare SU o GIU. Ora, i W1 e i W2 casuali iniziali provocheranno dei movimenti spasmodici sul posto della paletta. L'obbiettivo è trovare W1 e W2 in modo che la rete possa diventare un giocatore esperto.

La probabilità è ottenuta con la funzione sigmoidea, così definita

$$p(x) = \frac{1}{1 + e^{-x}}$$

,

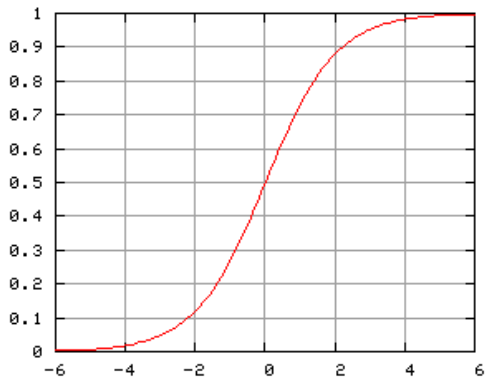

Figura 3.4: Il grafico della funzione sigmoide. Viene utilizzata dall'ultimo strato della rete per determinare la probabilità effettuare l'azione SU. Il punto di separazione delle classi è il valore 0, che porta ad avere una probabilità del 50%

che è la funzione di attivazione neuronale. Viene utilizzata sul risultato del neurone di output restituendo un numero p nell'intervallo $(0, 1)$ che corrisponde alla probabilità p di muovere la racchetta verso SU (o conseguentemente verso GIU con probabilità $1 - p$)

Inseme alla probabilità p di muovere SU, la funzione ritorna anche l'output del primo strato di neuroni perché poi verrà utilizzata successivamente per la backpropagation, il momento di apprendimento vero e proprio.

3.4 Iperparametri

Un iperparametro è un parametro di un algoritmo di apprendimento (non del modello) che deve essere impostato prima dell'addestramento e quindi non è influenzato dall'algoritmo di apprendimento stesso.

```
# hyperparameters
# batch_size è il numero di partite utilizze ad ogni backpropagation (i.e. ad ogni discesa del gradiente per aggiornare i pesi della rete neurale)
batch_size = 10 # every how many episodes to do a param update?
learning_rate = 1*10**-2.75

# importanza della reward futura rispetto alla reward futura, 0.99 massima importanza per la reward futura.
gamma = 0.99 # discount factor for reward
decay_rate = 0.99 # decay factor for RMSProp leaky sum of grad^2
```

- **Batch Size**: È il numero di istanze da utilizzare nella discesa del gradiente per l'addestramento. È un sottoinsieme casuale delle istanze del dominio applicativo. Usare un sottoinsieme dei dati molto ristretto (qualche decina di istanze solitamente) accelera di molto l'apprendimento perché l'algoritmo manipola pochi dati alla volta ma potrebbe avere problemi di convergenza a causa di minimi locali. Inoltre la stabilità della convergenza dipende dalla grandezza del batch. In questo caso il batch size indica dopo quanti episodi effettuare l'addestramento (impostato a 10). Con episodio si intente la partita. Ogni episodio è composto da diversi round.
- **Learning Rate** η: Costituisce la lunghezza del passo di discesa (step size), il rapporto tra gradiente e spostamento ad ogni passo.
  - Se troppo basso l'algoritmo impiega molte iterazioni per convergere.
  - Se troppo alto l'algoritmo potrebbe rallentare nel trovare il minimo. Per via di avanzamenti troppo ampi da un punto all'altro della curva.
  - In alcuni metodi η non è costante e viene fatto variare da un'iterazione all'altra (in questo caso impostato a $10^{-2,75}$).
- **Gamma** γ: È un valore utilizzato nella funzione **discount_rewards** fattore di sconto esponenziale delle vincite (che può essere anche negativo)

di ogni singola reward in ogni episodio. Il suo utilizzo viene approfondito successivamente. Solitamente si utilizzano valori tra 0.9 a 0.99 se gli effetti delle azioni nel dominio siano rispettivamente a breve e a lungo termine (in questo caso impostato a 0,99).

- **Decay rate** β: È utilizzato nella discesa del gradiente effettuata con RMSProp. L'algoritmo accumula in una cache s solo i gradienti delle iterazioni più recenti. Viene approfondito successivamente. Può essere impostato tra 0,8 e 0,99 in base a quanto peso si vuole dare ai gradienti passati rispetto a quello nuovo (in questo caso impostato a 0,99).

3.5 Assegnazione delle ricompense

```
# la funzione restituisce l'array delle reward di una partita pesandole in modo decrescente dalla fine
# della partita verso l'inizio della partita, in sostanza le reward finali nel game pesano di più di quelle iniziali
def discount_rewards(r):
  """ take 1D float array of rewards and compute discounted reward """
  WIN = 1
  LOSE = -1
  discounted_r = np.zeros_like(r, dtype=np.float32)
  running_add = 0
  for t in reversed(range(0, r.size)):
    if r[t] == WIN or r[t] == LOSE: running_add = 0 # reset the sum, since this was a game boundary (pong specific!)
    running_add = running_add * gamma + r[t]
    discounted_r[t] = running_add
  return discounted_r
```

Il problema che si pone ora capire cosa ha portato all'ottenimento della ricompensa alla fine di ogni match. L'insieme delle azini più recenti prima che un Agente riceva una ricompensa è la più pertinente e pertanto dovrebbe essere incoraggiata in caso di ricompensa positiva e scoraggiata per una ricompensa negativa. Qualsiasi azione o frame più indietro nel tempo dal momento in cui il premio è stato ricevuto viene attribuito con meno credito di un fattore γ (iperparametri, section 3.4).

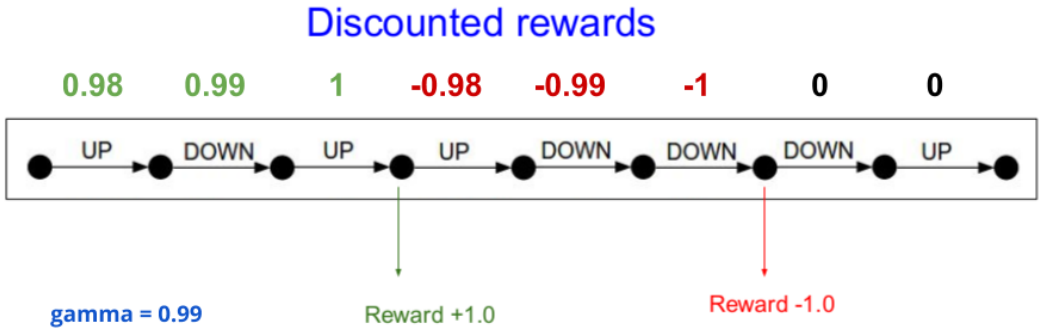

Figura 3.5: L'effetto dello sconto applicato ad ogni singolo match. Le azioni più recenti hanno il peso maggiore rispetto a quelle nel passato. Da notare che lo sconto viene propagato dalla ricompensa in tutte e sole le ricompense uguali a 0.

Fonte: Andrej Karpathy blog,Deep Reinforcement Learning: Pong from Pixels

Questo viene effettuato su ogni round in modo indipendente dagli altri ponendo la variabile **sum_of_rewards** (che contiene il valore della precedente ricompensa scontata) a 0 quando il premio per una data azione non è zero, evento che accade solo alla fine di ogni round. A è il vettore di tutte le ricompense scontate nel tempo.

Lo sconto ha l'effetto di attribuire in modo più accurato la ricompensa all'azione probabilmente ha contribuito maggiormente e che quindi è più responsabile della ricompensa e della vittoria.

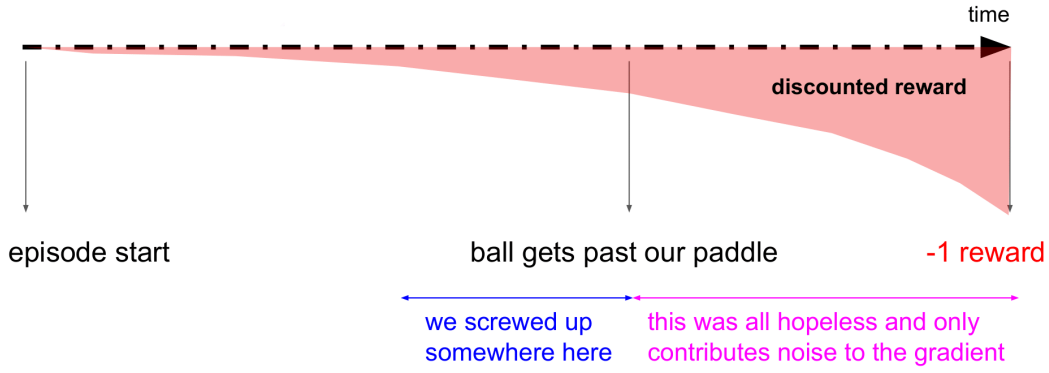

Figura 3.6: Il grafico mostra nel tempo il valore, che decade in modo esponenziale, delle ricompense. Inoltre viene sottolineato il fatto che le azioni successive all'attraversamento della racchetta da parte della pallina causino solo rumore nell'addestramento.

Fonte: Andrej Karpathy blog,Deep Reinforcement Learning: Pong from Pixels

Per stimare correttamente questi valori (chiamati vantaggio d'azione) si devono eseguire diversi episodi, scontarli con il metodo illustrato sopra e normalizzare tutti i valori (sottraendo la media e dividendo per la deviazione standard). Ora si possono fare supposizioni più accurate. Se un valore è negativo allora corrisponde ad un'azione negativa, viceversa con i valori positivi. In questo modo vengono incoraggiate e scoraggiate circa la metà delle azioni eseguite. La normalizzazione viene effettuata successivamente.

3.5.1 Ricompense più in generale

Pong ha una configurazione di premi molto semplice, poiché la ricompensa viene elargita solo al termine di ogni match e le azioni intraprese sono pesate sulla base di quest'ultima a ritroso. In un problema di reinforcement learning più generale si riceve una ricompensa ogni momento e le funzioni di sconto possono essere usate per valutare lo stato in base ai premi futuri previsti.

$$R_t = \sum_{n=1}^{H} \gamma^n r_{t+n}$$

Funzione di attualizzazione della ricompensa, γ è il fattore di sconto, n è il numero di passi temporali, r è la ricompensa per un dato passo $t+n$ esimo e H è il numero di eventi futuri che si vogliono considerare, può valere anche infinito.

L'espressione afferma che la forza con la quale incoraggiamo un'azione campionata è la somma ponderata di tutte le ricompense in seguito, ma le ricompense successive sono esponenzialmente meno importanti.

3.6 Policy Backward

```
# backpropagation: epx l'array degli stati
#   eph array dell'output dell'unico hidden layer (quello dopo il primo di input)
#   epdlogp la modulazione delle differenze tra il valore atteso e quello predetto rispetto alla reward
def policy_backward(epx, eph, epdlogp):
  """ backward pass. (eph is array of intermediate hidden states) """
  dW2 = np.dot(eph.T, epdlogp).ravel()
  # prodotto elemento per elemento del vettore modulato con reward col vettore dei pesi W2
  dh = np.outer(epdlogp, model['W2'][:-1])
  # applica la ReLu portando a zero i valori negativi
  dh[eph[:,:-1] <= 0] = 0 # backpro prelu
  # calcola il prodotto scalare tra l'uscita della ReLu con tutti gli stati (ogni stato è un vettore)
  dW1 = np.dot(dh.T, epx)
  return {'W1':dW1, 'W2':dW2}
```

L'algoritmo, mediante il quale si aggiornano i coefficienti della rete, si definisce backpropagation in virtù del fatto che l'errore che registrato in corrispondenza di un certo dato viene fatto propagare all'indietro nella rete per ottenere un aggiornamento dei coefficienti.

Si immagini ridurre la rete ad un neurone. Vengono indicati con i il valore di input, w il valore dell'unico coefficiente e p il valore di output espressione della probabilità di andare in alto calcolato con la funzione sigmoidea. Il valore da incoraggiare è SU.

Dato che i valori di input sono immodificabili occorre agire su w. Come deve essere aggiornato il valore di w affinché la rete dia l'output desiderato? Occorre calcolare l'errore della stima rispetto al valore corretto.

3.6.1 Misura dell'errore del modello

Un classificatore di regressione logistica binaria ha solo due classi (0,1) e calcola la probabilità della classe 1 come:

$$P(y = 1|x; w, b) = \frac{1}{1 + e^{-w^T X + b}} = \sigma(w^T X + b)$$

Poiché le probabilità di classe 1 e 0 sommate danno 1, la probabilità per classe 0 è $P(y = 0|x; w, b) = 1 - P(y = 1|x; w, b)$. Quindi, un esempio è classificato come esempio positivo (y = 1) se $\sigma(w^T X + b) > 0,5$ o equivalentemente se il punteggio $w^T X + b > 0$. La funzione di perdita massimizza quindi questa probabilità. Svolgendo i calcoli si ottiene:

Cross-entropy loss function

$$L = y\, log(p) + (1 - y) log(1 - p)$$

3.6.2 Regola della catena

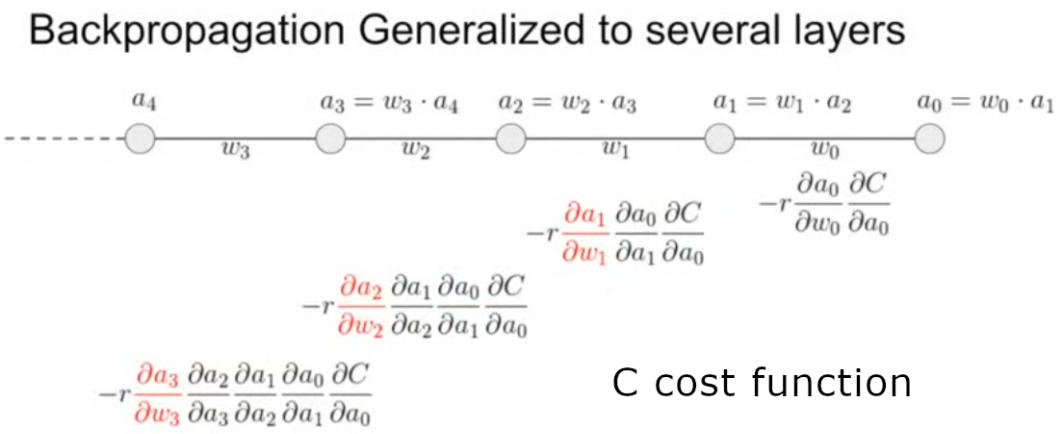

Figura 3.7: L'immagine mostra come viene applicata la regola della catena in cascata per derivare i vari livelli della rete e poter così effettuare back propagation.

Fonte: Youtube: Neural Network Backpropagation Basics For Dummies

La regola della catena è una regola di derivazione che permette di calcolare la derivata della funzione composta di due funzioni derivabili.

$$\frac{\partial}{\partial x} f(g(x)) = g'(x) \cdot f'(x)$$

In una rete neurale le funzioni sono funzioni composte di tutti gli strati. Con la regola della catena è possibile derivare le funzioni degli strati superiori.

La funzione di costo è una funzione in p che è una funzione in x $(w^T X + b)$. Si può usare la regola della catena per calcolare il gradiente dell'errore.

$$\frac{\partial L}{\partial x} = \frac{\partial p}{\partial x}\frac{\partial L}{\partial p} = y - p$$

Gradiente determina la direzione in cui la curva sale o scende più ripidamente da un punto determinato. Permette di identificare quindi in che direzione andare per massimizzare o minimizzare una funzione. In questo caso l'obiettivo è massimizzare le ricompense (che hanno un massimo di 21 per ogni episodio), quindi il gradiente viene sommato a ciascun peso della rete in modo proporzionale al learning rate aggiustandone i valori. Questo aggiornamento viene propagato all'indietro della rete attraverso la backpropagation.

Moltiplicando il valore di errore dell'azione scelta con il valore corrispondente della ricompensa si ottiene un valore dell'errore ponderato. La funzione di errore completa diventa quindi

$$E = A \cdot (y - p)$$

Con A il vettore di ricompense scontate, y il vettore delle azioni effettuata nel gioco e p il vettore di probilità di SU ricavate dalla rete.

3.6.3 Gradienti dei livelli della rete

Utilizzando la regola della catena si ricava come aggiornare i due strati della rete originale. Considerando W_1 e W_2 i pesi rispettivamente del primo e del secondo strato, O_1 l'output del primo strato della rete, A_i il valore scontato delle ricompense nel tempo ed E la funzine di errore:

$$\nabla W_2 = O_1 \cdot E$$

$$\nabla W_1 = X \cdot W_2 \cdot E$$

3.6.4 Apprendimento supervisionato vs reinforcement Learning

Come apprenderebbe una rete con apprendimento supervisionato?

Viene data un'introduzione al concetto di apprendimento supervisionato perché, come si osserverà successivamente, reinforcement learing è molto simile. Fare riferimento allo schema seguente. Nell'apprendimento ordinario supervisionato viene fornita un'immagine alla rete e otteniamo alcune probabilità, ad esempio per le due classi SU e GIU. Viene utilizzato il logaritmo delle probabilità (-1.2 -> 30%, -0.36 -> 60%) perché rende la matematica più semplice ed l'ottimizzazione risulta equivalente, dato che il logaritmo è monotonico. Ora, nell'apprendimento supervisionato si ha accesso a un'etichetta, cioè al valore corretto. Ad esempio, la mossa giusta da fare in questo momento è andare SU (etichetta 0). In un'implementazione si utilizza un gradiente di 1.0 sul logaritmo della probabilità di andare SU, si effettua backpropagation per calcolare il vettore gradiente $\nabla_w log\, p(y = UP|x)$. Questo gradiente indica come dovremmo modificare ognuno dei parametri della rete per renderla leggermente più propensa a prevedere SU. Ad esempio, uno dei milioni di parametri nella rete potrebbe avere un gradiente di -2,1, il che significa che se dovessimo aumentare quel parametro di un piccolo valore positivo (ad es. 0,001), il logaritmo della probabilità di SU diminuirebbe di 2,1 * 0,001 (diminuzione dovuta al segno negativo). Se si facesse un aggiornamento dei parametri la rete ora sarebbe leggermente più propensa a prevedere SU quando riceverà un'immagine molto simile in futuro.

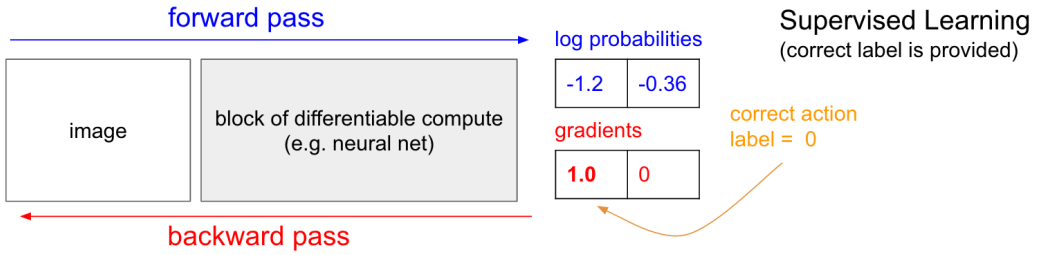

Figura 3.8: Nel supervised learning insieme ai dati viene fornita l'etichetta corrispondente. Per l'addestramento occorre solamente calcolare l'errore effettuato rispetto all'etichetta e utilizzarlo per aggiornare i pesi della rete.
Fonte: Andrej Karpathy blog,Deep Reinforcement Learning: Pong from Pixels

Più nello specifico, l'obbiettivo è massimizzare

$$\sum_i log\, p(y_i|x_i)$$

dove x_i e y_i sono esempi per l'addestramento (come immagini e rispettive etichette)

Cosa cambia nel reinforcement learning?

Nel reinforcement learning che non si possiedono etichette, come fare? La politica della rete rete ha calcolato la probabilità di SU del 30% (logprob -1.2) e GIU del 70% (logprob -0.36). Utilizzando quelle probabilità si suppone di campionare GIU e di eseguirlo nel gioco. Non avendo l'etichetta si sostituisce l'azione GIU con "un'etichetta falsa" e così per ogni altro input. Non si può procedere come prima in quanto non si conosce a priori al correttezza della previsione. Si può pero aspettare che la palla tocchi un bordo e la ricompensa che otteniamo (uno scalare, +1 se abbiamo vinto o -1 se abbiamo perso) come gradiente per l'azione intrapresa (GIU in questo caso). Nell'esempio che segue, andare verso il basso ha fatto perdere la partita (-1 come ricompensa). Quindi, se utilizziamo -1 per il logaritmo della probabilità di GIU e facciamo backpropagation si ottiene un gradiente che scoraggia la rete a intraprendere l'azione GIU per quell'input in futuro, dal momento che intraprendere quell'azione ci ha portato a perdere il gioco.

Si utilizza una politica che comporta una certa casualità (viene definita stocastica) che campiona le azioni, quelle che possono eventualmente portare a buoni risultati nel futuro vengono incoraggiate mentre quelle che portano a perdita vengono scoraggiate. Inoltre, la ricompensa non deve nemmeno essere +1 o -1 se alla fine vinciamo la partita. Le reward possono essere diverse e modellate con una misura arbitraria. Ad esempio, si può aggiungere una piccola ricompensa iniziale quando la paletta tocca la palla (in modo da accelerare

l'apprendimento sulla fase iniziale del gioco) e decrementarla mano a mano portando più valore alla vittoria dell'episodio.

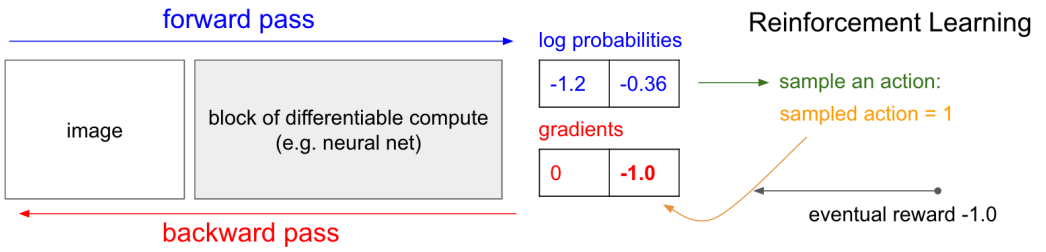

Figura 3.9: Nel reinforcement learning non si possiede a priori un'etichetta. Si utilizza quindi una finta etichetta utilizzano l'azione che in modo stocastico è stata campionata. In questo modo non è detto che l'addestramento avvenga in modo corretto.

Fonte: Andrej Karpathy blog,Deep Reinforcement Learning: Pong from Pixels

In sintesi ora bisogna massimare

$$\sum_i A_i \, log \, p(y_i|x_i)$$

dove y_i è l'azione campionata y_i e A_i è un numero che si chiama vantaggio. Nel codice il prodotto viene effettuato in **epdlogp *= discounted_epr**. Nel caso di Pong, ad esempio, A_i potrebbe essere 1.0 se alla fine avessimo vinto nell'episodio che conteneva x_i e -1.0 se abbiamo perso. Ciò assicurerà di massimizzare il logaritmo della probabilità delle azioni che hanno portato a buoni risultati e di ridurre al minimo quello delle azioni che non lo hanno fatto. Quindi l'apprendimento con reinforcement learning è esattamente come l'apprendimento supervisionato, ma su un set di dati in continua evoluzione (gli episodi), scalato dal vantaggio e per ciascun set di dati campionati vengono effettuati solo uno (o pochissimi) aggiornamenti.

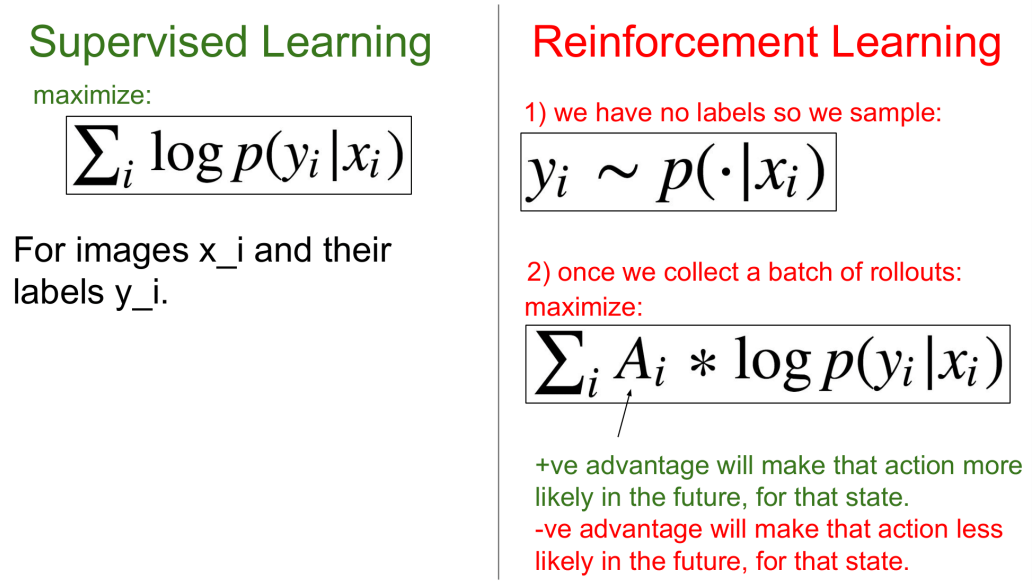

Figura 3.10: A sinitra cosa massimizzare nel supervised learning. A destra cosa massimizzare nel reinforcement learning, le azioni $(log\, p(y_i, x_i))$ vengono pesate in base al risultato, scontato rispetto al momento della loro esecuzione, che hanno portato all'interno di un singolo match (A_i)

Fonte: Andrej Karpathy blog,Deep Reinforcement Learning: Pong from Pixels

```python
# addestramento di default su 100 partite
def train_model(env, model,rewards_path,model_name, total_episodes = 100,save_freq=None,
                racket_reward_on = False,racket_reward = 0.05,decay_racket_reward=0.99):
  hist = []
  observation = env.reset()
  prev_img = None

  prev_x = None # used in computing the difference frame
  xs,hs,dlogps,drs = [],[],[],[]
  running_reward = None
  direction = RIGHT
  reward_sum = 0
  episode_number = 0
  current_racket_reward = racket_reward
  total_racket_reward = 0
  past_rewards = load_previos_data(rewards_path)
  current_rewards = []
  grad_buffer, rmsprop_cache = initialize_rsmprop(model)
  decay_racket_reward = pow(decay_racket_reward,len(past_rewards))


  while True:
    if racket_reward_on:
      res, direction = hit(observation,prev_img,direction)
      if res:
        drs[-1] = current_racket_reward
        total_racket_reward += current_racket_reward
      prev_img = observation
    # preprocess the observation, set input to network to be difference image
    x,prev_x = new_state(observation,prev_x)
    x_bias = np.append(x,1)
    # forward the policy network and sample an action from the returned probability
    # verificare e sistemare in modo che riutilizzi le funzioni già definite
    aprob, h = policy_forward(x_bias)
    # l'azione da fare, che gli ha restituito la rete neurale, è condizionata all'estrazione di un numero casuale tra 0 ed 1 per avere la possibilità di esplorare e fare nuove esperienze
    action = 2 if np.random.uniform() < aprob else 3 # roll the dice!
```

```
# record various intermediates (needed later for backprop)
# aggiunge lo stato x all'array
xs.append(x_bias) # observation
# aggiunge l'ouput del primo hidden layer all'array
hs.append(h) # hidden state
# etichettura delle instanze su cui avviene il learning
y = 1 if action == 2 else 0 # a "fake label"
dlogps.append(y - aprob) # grad that encourages the action that was taken to be taken (see http://cs231n.github.io/neural-networks-2/#losses if confused)

# step the environment and get new measurements
# nella reward abbiamo l'esito, ossia se l'azione ha prodotto o meno un risultato positivo
observation, reward, done, info = env.step(action)
reward_sum += reward

drs.append(reward) # record reward (has to be done after we call step() to get reward for previous action)
```

```
if done: # an episode finished ossia se il singolo game è finito
  episode_number += 1

  # stack together all inputs, hidden states, action gradients, and rewards for this episode
  epx = np.vstack(xs)
  eph = np.vstack(hs)
  epdlogp = np.vstack(dlogps)
  epr = np.vstack(drs)
  xs,hs,dlogps,drs = [],[],[],[] # reset array memory

  # compute the discounted reward backwards through time
  discounted_epr = discount_rewards(epr)
  # standardize the rewards to be unit normal (helps control the gradient estimator variance)
  discounted_epr -= np.mean(discounted_epr)
  discounted_epr /= np.std(discounted_epr)

  # se non facesse questo cosa accadrebbe ? non ci sarebbe nessuna relazione tra le reward e la differenza tra ciò che la rete ha
  # indicato di fare e ciò che invece doveva essere fatto
  epdlogp *= discounted_epr # modulate the gradient with advantage (Policy Gradiient magic happens right here.)
  grad = policy_backward(epx, eph, epdlogp)
  for k in model: grad_buffer[k] += grad[k] # accumulate grad over batch

  # perform rmsprop parameter update every batch_size episodes
  if episode_number % batch_size == 0:
    for k,v in model.items():
      g = grad_buffer[k] # gradient
      rmsprop_cache[k] = decay_rate * rmsprop_cache[k] + (1 - decay_rate) * g**2
      # non essendoci il meno, verificare se sta massimizzando o minimizzando una loss e quale loss.
      model[k] += learning_rate * g / (np.sqrt(rmsprop_cache[k]) + 1e-5)  # cos'è 1e-5 ?
      grad_buffer[k] = np.zeros_like(v) # reset batch gradient buffer
```

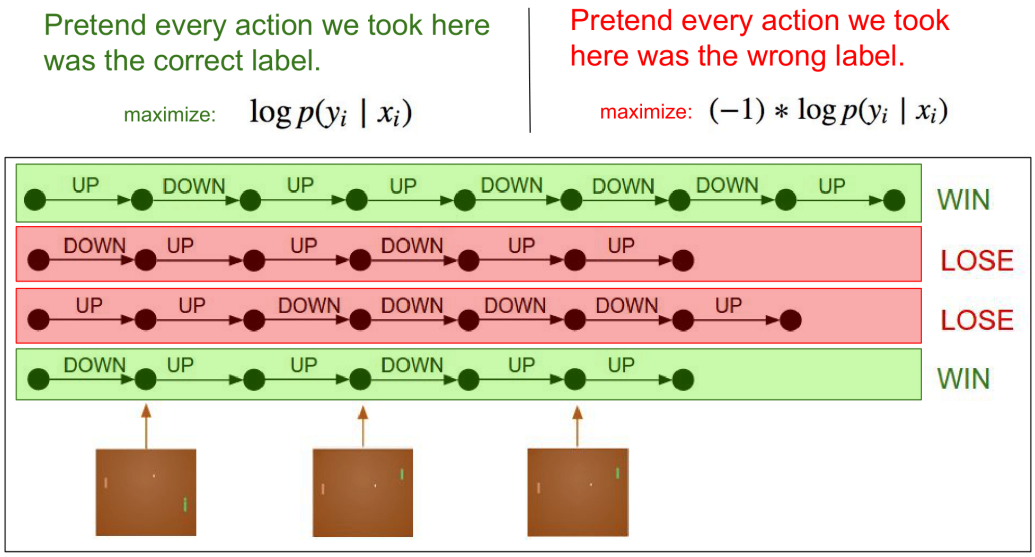

Figura 3.11: Come massimizzare le azioni: la probabilità è data dalla funzione sigmoidea che restituisce valore prossimi a 0 per x negativi o valore vicino a 1 per valori positivi. Nel caso di azione positiva, la rete aumenta i pesi per ottenere così un numero più grande positivo e di conseguenza una probabilità più alta. Al contrario se l'azione è negativa la rete abbassa il valore dei pesi in modo da ottenere un numero più piccolo e quindi una probabilità minore.
Fonte: Andrej Karpathy blog,Deep Reinforcement Learning: Pong from Pixels

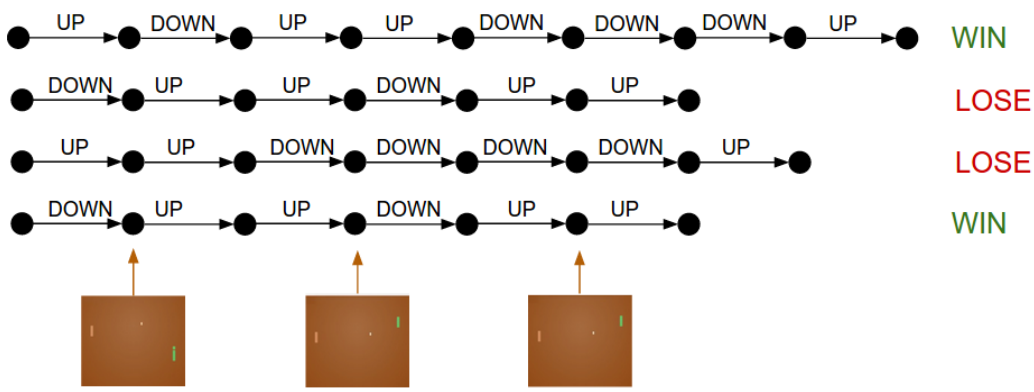

Figura 3.12: Ogni cerchio nero è uno stato di gioco (tre stati di esempio sono visualizzati in basso) e ogni freccia è una transizione, annotata con l'azione che è stata campionata. In questo caso sono stati vinti 2 partite e perse 2. Vengono incoraggiate leggermente le azioni effettuate nelle due partite che vinte e scoraggiate quelle dei due giochi persi.
Fonte: Andrej Karpathy blog,Deep Reinforcement Learning: Pong from Pixels

Si inizializza la rete con W1 e W2 e si ipotizzano 100 partite. Supponiamo che ogni gioco sia composto da 200 frame, quindi in totale sono state prese 20.000 decisioni per andare SU o GIU e per ognuno di questi si calcola il gradiente dei parametri che indica che aggiornamenti apportare per incoraggiare quella decisione in quello stato in futuro. Non resta che etichettare ogni decisione come positiva o negativa. Ad esempio, si ipotizza lo scenario in cui si vincono 12 partite e se ne perdono 88. Per tutte le 200 * 12 = 2400 decisioni vincenti si effettua un aggiornamento positivo (inserendo un valore positivo nel gradiente per l'azione campionata, e aggiornando i parametri che incoraggiano le azioni scelte in tutti quegli stati). Le altre 200 * 88 = 17600 decisioni prese nei giochi perdenti comporteranno invece un aggiornamento negativo in modo da scoraggiare quelle azioni in futuro. La rete diventerà ora leggermente più propensa a ripetere azioni che hanno funzionato e leggermente meno a ripetere azioni che non hanno funzionato.

3.6.5 Eploration vs Exploitation

L'obiettivo di generare di numeri casuali con **np.random.uniform()** è introdurre l' esplorazione di nuovi stati, che è fondamentale all'inizio quando la rete non può davvero differenziare le azioni positive da quelle negative, poiché i pesi sono inizializzati in modo casuale e l'output di probabilità per l'immagine di input è vicina a 0,5 a causa della sigmoide. Ciò si traduce in una serie casuale di azioni da parte della rete senza alcun senso di coordinamento all'inizio. Il processo di selezione di un'azione in base alla probabilità di azione calcolata modulata da un fattore casuale viene anche definito *campionamento*.

Man mano che la rete viene sempre più addestrata e supponendo che la ricompensa media stia aumentando, il fattore di casualità gioca un ruolo minore, poiché nel tempo la probabilità ricavata dalla rete che una determinata azione debba essere intrapresa è più vicina agli estremi (0 e 1) e quindi determina il risultato in modo decisivo. Un altro fattore che contribuisce alla rete che esplora lo spazio degli stati è l'uso di una politica stocastica. Infatti se la politica fosse deterministica si farebbe affidamento esclusivamente sul generatore di numeri casuali per l'esplorazione, che avrebbe un comportamento come quello che ha la rete inizialmente; tuttavia significherebbe che qualsiasi nuova azione è puramente casuale e meno guidato dalle precedenti probabilità acquisite dalla rete. La riduzione dell'esplorazione e un maggiore sfruttamento di buone azioni note possono anche intrappolare la politica in un ottimo locale, poiché la rete smette di esplorare lo spazio degli stati.

3.6.6 Variabili utilizzate nell'apprendimento

```
def load_previous_data(path_reward):
  try:
    with open(path_reward, "r") as parameters:
      return load(parameters)
  except:
      return []

def write_data(path_reward,path_model,model,past_rewards,current_rewards):
  with open(path_reward, 'w+') as outfile:
    dump(past_rewards+current_rewards, outfile)

  with open(path_model, 'w+') as outfile:
    dump(model, outfile)

  print('scritture effettuate')
```

Sono funzioni che permettono di caricare e scrivere su file i valori di rewards.

```
hist = []
observation = env.reset()
prev_x = None # used in computing the difference frame
xs,hs,dlogps,drs = [],[],[],[]
running_reward = None
reward_sum = 0
episode_number = 0
past_rewards = load_previous_data(rewards_path)
current_rewards = []
grad_buffer, rmsprop_cache = initialize_rsmprop(model)
```

- **hist = [ ]** Conterrà i dati di **episode_number, reward_sum, running_reward** che sono rispettivamente numero di episodi, la somma delle ricompense e la ricompensa pesata in base a tutte quelle precenti.
- **observation = env.reset()** ritorna l'immagine stato iniziale del gioco.
- **prev_x = None** l'immagine precedente.

- **xs,hs,dlogps,drs = [ ], [ ], [ ], [ ]** sono i vettori che accumulano tutti i dati del **batch_size** che poi verranno utilizzati nella backpropagation. Sono rispettivamente l'array degli stati del gioco,l'output del primo strato, il vettore degli errori delle azioni ponderato rispetto al tempo e l'array con tutte le reward acquisite.

- **running_reward** contiene il valore medio delle rewards ponderato alla quantità

- **reward_sum** contiene la somma di tutte le rewards ottenute in ogni match di un singolo episodio.

- **episode_number** rappresenta il numero dell'episodio corrente.

- **total_racket_reward** è l'array che contiene eventuali rewards precedenti e viene letto da file utilizzando la funzione **load_previous_data**.

- **current_rewards** è l'array delle rewards acquisite durante questo addestramento.

3.6.7 Salita del gradiente

Per aggiornare i pesi della rete W1 e W2 occorre calcolare il gradiente ricavato dalla backpropagation e sommarlo ai pesi stessi [7].

$$W_1 \leftarrow W_1 + \eta \nabla W_1$$

$$W_2 \leftarrow W_2 + \eta \nabla W_2$$

Con η il learning rate (iperparametri, section 3.4).

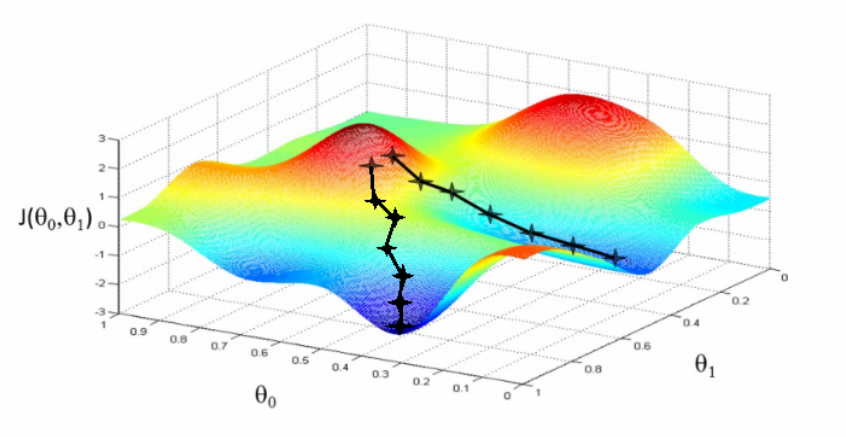

Figura 3.13: In questo caso si cerca di trovare il valore massimo delle ricompense che si possono ottenere applicando la policy foreward J alla rete composta di due pesi θ_0 e θ_1. Il gradiente viene utilizzato per "salire" nello spazio
Fonte: Lecture 2 | Machine Learning (Stanford)

Questo metodo potrebbe risultare eccessivamente lento e/o rischiare rimanere bloccato in un massimo locale. Vengono introdotti metodi per ottimizzare di molto la salita del gradiente.

3.6.8 Accelerazione con momentum

L'idea di base è calcolare la media ponderata in modo esponenziale dei gradienti e quindi utilizzare quel nuovo gradiente per aggiornare i pesi [8].

Si consideri di dover trovare il minimo (ragionamento per trovare il massimo è equivalente) di una funzione di errore che le curve di livello (lungo la stessa linea l'errore è equivalente e diminuisce spostandosi verso quelle più centrali) come quelle in figura e il punto rosso indica la posizione di minimo.

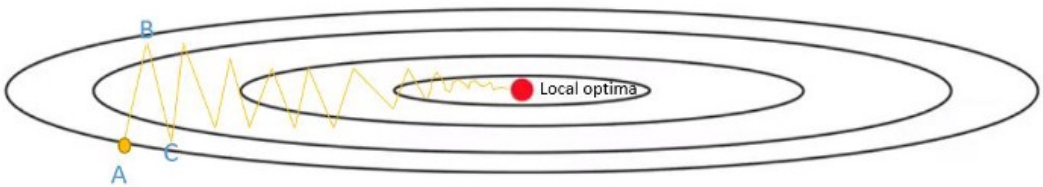

Figura 3.14: La figura mostra una discesa del gradiente standard, in cui sono presenti oscillazioni di varia dimensione che portano un rallentamento (a volte interrompono completamente) della convergenza al valore ottimo.
Fonte: Youtube, Gradient Descent With Momentum (C2W2L06)

La discesa del gradiente inizia dal punto 'A' e finisce nel punto 'B' dopo un'iterazione, l'altro lato dell'ellisse. Quindi un'altra iterazione può terminare nel punto 'C'. Con le iterazione della discesa del gradiente si converge verso gli ottimi locali (i punti 'A', 'B' e 'C') causando forti oscillazioni verticali. Questa oscillazione verticale quindi rallenta la discesa del gradiente e impedisce di utilizzare un tasso di apprendimento molto più elevato dato che l'oscillazione risulterebbe maggiore.

Usando una media esponenzialmente pesata dei valori del gradiente si riducono le oscillazioni nella direzione verticale rendendole più vicine allo zero (media di valori di salita positivi e di discesa negativi). Ora si più utilizzare un learning rate più elevato dato che le oscillazioni verticali sono state ridotte e quindi si può accelerare la direzione orizzontale.

Ciò consente all'algoritmo di intraprendere un percorso più diretto verso il minimo in alcune iterazioni.

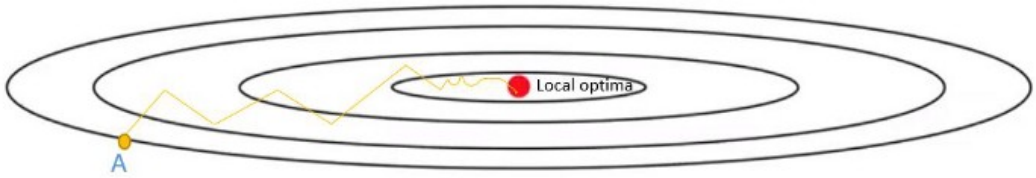

Figura 3.15: La figura rappresenta una discesa del gradiente utilizzando il momento come acceleratore della convergenza. Le oscillazioni sono ridotte in verticale mentre viene aumentato il passo in orizzontale.
Fonte: Youtube, Gradient Descent With Momentum (C2W2L06)

Come scegliere il fattore di decadimento (momentum) β ?

Deve essere alto in modo da dare molto peso ai gradienti passati. Solitamente si utilizza il valore predefinito di $\beta = 0,9$ ma, se necessario, può essere regolato tra 0,8 e 0,999.

3.6.9 Algoritmo RMSProp

La sigla sta per "Root Mean Square Propagation". Accumula nel buffer solo i gradienti di ogni aggiornamento (quindi di quelli calcolati in **batch_size** epoche) per poi azzerarli con **grad_buffer[k] = np.zeros_like(v)**.

Il vettore dei gradienti (uno per ogni peso) è scalato di un fattore di un fattore $\sqrt{s + \epsilon}$ con ϵ un termine di arrotondamento per evitare divisioni per zero, tipicamente impostato a 10^{-10} (in questo caso 10^{-6}).

In breve, questo algoritmo riduce il learning rate, ma lo fa più velocemente per le dimensioni ripide che per quelle con pendenze più dolci. Si chiama *adaptive learning rate*. Aiuta a indirizzare gli aggiornamenti in modo più diretto verso l'ottimale globale. Un ulteriore vantaggio è che richiede una regolazione molto minore di η.

$$s \leftarrow \beta s + (1 - \beta)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$$

$$w \leftarrow w + \eta \nabla_\theta J(\theta) \oslash \sqrt{s + \epsilon}$$

Con il prodotto $\otimes$ si intende moltiplicazione elemento per elemento e viene usata per calcolare il gradiente al quadrato. Con la divisone $\oslash$ si intente la divisone elemento per elemento.

β è l'iperparametro del momentum.

3.6.10 Perché si accumula il gradiente?

```
for k in model: grad_buffer[k] += grad[k] # accumulate grad over batch
```

Si accumulano i gradienti dei pesi della rete per ogni specifico episodio. Eseguire alcuni passaggi (in questo caso ogni **batch_size** iterazioni) senza aggiornare nessuno dei pesi del modello è il modo in cui, logicamente, si suddivide il set completo di dati in mini-batch. Il batch di campioni utilizzato in ogni passaggio è effettivamente un mini-batch e tutti i campioni di quei passaggi combinati sono effettivamente il batch globale.

Non aggiornando le variabili in tutti questi passaggi facciamo in modo che tutti i mini-batch utilizzino le stesse variabili del modello per il calcolo dei gradienti. Questo è obbligatorio per garantire che gli stessi gradienti e gli stessi aggiornamenti vengano calcolati come se stessimo utilizzando la dimensione globale del batch. L'accumulo dei gradienti in tutti questi passaggi produce la stessa somma dei gradienti come se stessimo usando la dimensione globale del batch, risparmiando di dover effettuare la discesa del gradiente per ogni episodio ma ogni **batch_size** [9].

3.6.11 Addestramento

```
env = wrap_env(gym.make("Pong-v0"))
model_name = name+"_model.txt"
model = load_or_create_model(model_name)

train_model(env,model,name+"_rewards.txt",model_name=model_name,total_episodes = 3000,save_freq=100)
```

L'addestramento avviene richiamando la funzione **train_model** con i rispettivi valori di addestramento (nell'immagine indicativa viene il modello viene addestrato per 3000 episodi e i dati vengono salvati ogni 100 episodi).

Capitolo 4

Sviluppo di Pong con Ricompensa su un'Azione di Gioco

Ora si propone una nuova modifica all'esempio aggiungendo una retribuzione, che decade in modo esponenziale, ogni volta che la racchetta riesce a colpire la pallina.

```
racket_rewards = [1, 0.5, 0.2, 0.1, 0.05]
decay_racket_rewards = [0.99,0.999]
```

In questo modo la rete impara inizialmente, quando ancora effettua ancora azioni in modo casuale, a colpire la pallina e successivamente, grazie alla riduzione esponenziale della ricompensa della racchetta, apprende come fare punto come nella versione normale.

Il seguente codice serve per pre-processare le immagini in modo da poter identificare una situazione di tocco dal parte della pallina sulla racchetta. La funzione cuore è **hit** che ritorna il booleano che indica se è avvenuto il tocco e la direzione attuale della pallina (utilizzata nella successiva identificazione).

```python
#ball direciton values
RIGHT = 1
LEFT = 0

def split_image(img):
  left_cut = 110
  right_cut = 150
  start_racket = 140
  end_racket = 143
  I = img.copy()
  I = img[35:194,:].copy()
  left = I[:,left_cut:start_racket]
  right = I[:,end_racket+1:right_cut]
  return left,right


def cut_and_simplify(img,value = 1):
  I = img[...,0].copy()
  I[I == 144] = 0 # erase background (background type 1)
  I[I == 109] = 0 # erase background (background type 2)
  I[ I != 0] = value
  return I.astype(int)


def hit(current,prev = None, direction=RIGHT):
  limit = 20
  if prev is None : return False, RIGHT
  left,right = split_image(cut_and_simplify(current) - cut_and_simplify(prev)*2)

  neg_coordinates = np.argwhere(left<0)
  pos_coordinates = np.argwhere(left>0)

  #check hit
  if pos_coordinates.shape[0] != 0 and pos_coordinates[0][1] > limit and neg_coordinates.shape[0] == 0 and direction == RIGHT:
    return True, LEFT

  if neg_coordinates.shape[0] == 0 or pos_coordinates.shape[0] == 0 :
    return False,direction


  if neg_coordinates[0][1] > pos_coordinates[0][1] and direction == RIGHT:
    return True,LEFT

  #change direction
  if pos_coordinates.shape[0] != 0 and neg_coordinates.shape[0] != 0 and neg_coordinates[0][1] < pos_coordinates[0][1]:
    direction = RIGHT

  if pos_coordinates.shape[0] != 0 and neg_coordinates.shape[0] != 0 and neg_coordinates[0][1] > pos_coordinates[0][1]:
    direction = LEFT

  return False, direction
```

Sono stati provate diverse configurazioni di sconto e di rewards utilizzando una **grid search CV**, per alcune di esse l'addestramento è stato prolungato per verificare in modo più accurato l'andamento dell'addestramento.

```
#grid search cv
racket_rewards = [1,0.5,0.2,0.1,0.05]
decay_racket_reward = [0.99,0.999]

for r in racket_rewards:
  for d in decay_racket_reward:
    env = wrap_env(gym.make("Pong-v0"))
    model_name = f"{r}_{d}_model.txt"
    model = load_or_create_model(model_name)
    train_model(env,model,f"{r}_{d}_rewards.txt",model_name=model_name,
                total_episodes =1000,save_freq=100,
                racket_reward_on = True,racket_reward = r,decay_racket_reward=d)
```

Nella funzione **train_model** vengono aggiunte una serie di variabili per l'utilizzo della racchetta.

```
#used with racket
prev_img = None
direction = RIGHT
total_racket_reward = 0
current_racket_reward = racket_reward
decay_racket_reward = pow(decay_racket_reward,len(past_rewards))
```

- **prev_img** contiene l'immagine presente utilizzata nella funzione **hit** per verificare la collisione.
- **direction** indica la direzione della pallina, viene aggiornata dalla funzione **hit** che la utilizza per identificare la collisione.
- **total_racket_reward** contiene la somma delle rewards totali per ogni episodio date dalla racchetta.
- **current_racket_reward** corrisponde al valore corrente di reward che assegna ogni hit della racchetta.
- **decay_racket_reward** è il valore di sconto esponenziale da applicare ad ogni nuova reward ed è scalato in base a eventuali precedenti rewards.

Se la pallina colpisce la racchetta viene sommato alla penultima posizione dell'array (perché l'hit viene identificato nel frame successivo il tocco) la reward corrente della racchetta.

```
if racket_reward_on:
  res, direction = hit(observation,prev_img,direction)
  if res:
    drs[-1] = current_racket_reward
    total_racket_reward += current_racket_reward
  prev_img = observation
```

Qui sotto sono riportati i grafici in confronto con la versione standard senza l'utilizzo della reward applicata alla racchetta.

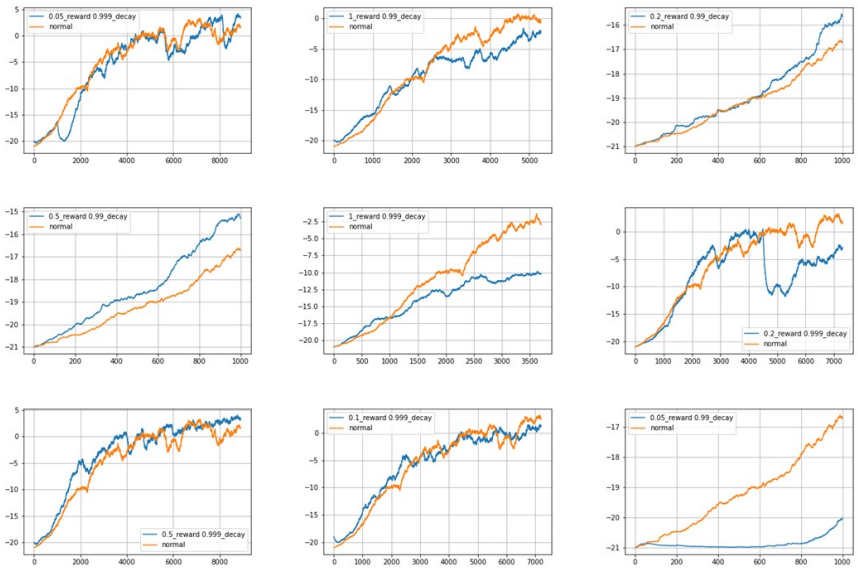

Figura 4.1: Grid search CV su rewards e decay

Utilizzando questa versione rispetto a quella normale di Karpathy si ottiene un netto miglioramento nell'apprendimento.

Capitolo 5

Sviluppo di Pong in Stable Baselines

Per migliorare l'apprendimento è stato deciso di utilizzare la librearia Stable Baseline, che mette a disposizione, in modo molto semplice e open source, un insieme di implementazioni migliorate di algoritmi di reinforcement learning.

Dato che si utilizzano immagini, si è deciso di effettuare l'addestramento della rete utilizzando come policy per i vari algoritmi, data la sua efficienza e il suo largo impiego nella classificazione di immagini, una rete convoluzionale al posto di quella multi layer perceptron (Mpl) utilizzata da Karpathy.

Gli algoritmi utilizzati sono: Double Q Learning, Proximal Policy Optimization (PPO2), Sample Efficient Actor-Critic with Experience Replay (ACER).

Come viene anche dimostrato da Deep Mind sul documento di PPO2 [10] e di ACER [11] *"Su Atari, PPO2 si comporta (in termini di complessità del campione) in modo simile ad ACER sebbene sia molto più semplice. ACER raggiunge le performance allo stato dell'arte su reti con funzione Q e prioritize replay nei giochi dell'ATARI"*.

Infatti eseguendo l'addestramento ho ottenuto il seguente grafico che evidenzia minime differenze. In ogni caso si può osservare che i due algoritmi convergono utilizzando all'incirca 500 episodi.

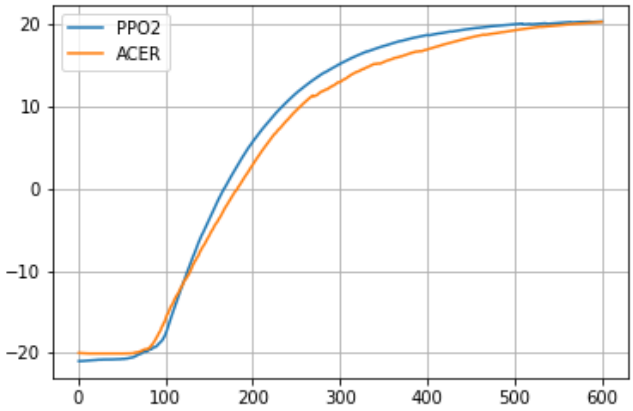

Figura 5.1: Grafici a confronto di PPO2 e ACER.

La convergenza di DDQN é rallentata dall'utilizzo del double-q learning e dal fatto che la funzione Q necessita di diverse iterazioni per impostarsi correttamente nell'addestramento. Inoltre l'algoritmo è molto suscettibile agli iperparametri, che possono causare forti oscillazioni nelle performance durante l'addestramento [12].

La convergenza viene raggiunta all'incirca in 3500 episodi.

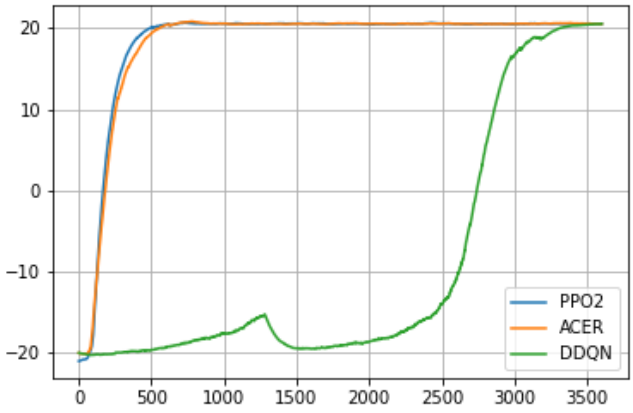

Figura 5.2: Andatura dell'addestramento di DDQN rispetto a PPO2 e ACER.

Tutti gli algoritmi utilizzati, grazie all'introduzione di tecniche più sofisticate e di una rete specifica per il riconoscimento di immagini, permettono di

migliorare in modo molto importante le velocità di convergenza ottenendo un miglioramento netto.

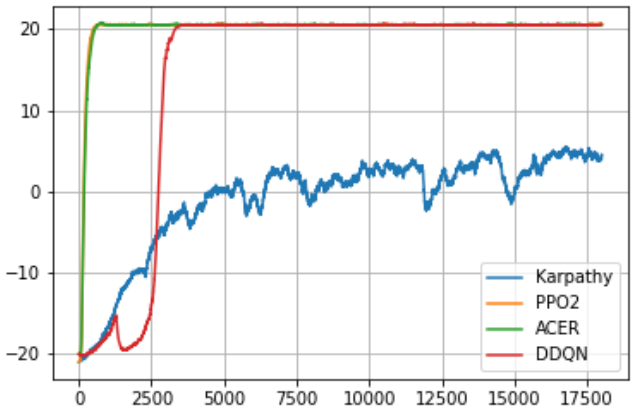

Figura 5.3: La soluzione di Karpathy rispetto agli altri algoritmi.

Per, ogni algoritmo, viene data una spiegazione dei suoi aspetti chiave. Ora viene data un'introduzione alle reti convoluzionali.

5.1 Codice in Python

Import delle librerie utilizzate:

```
from stable_baselines import ACER
from stable_baselines import PPO2
from stable_baselines import DQN

import tensorflow as tf
import gym
import numpy as np
from json_tricks import dump, dumps, load, loads, strip_comments # per salvare i pesi della rete neurale su disco
from stable_baselines.common.cmd_util import make_atari_env
from stable_baselines.common.callbacks import BaseCallback
from stable_baselines.common.vec_env import VecFrameStack
```

Funzioni di supporto per il caricamento e la scrittura di dati su file:

```
path = "savesStableBaseline/"

def load_previos_data(path_reward):
  try:
    with open(path+path_reward, "r") as parameters:
      return load(parameters)
  except:
      return []

def write_data(path_reward,path_model,model,env,past_rewards):
  with open(path+path_reward, 'w+') as outfile:
    current_rewards = env.env_method('get_episode_rewards')[0]
    dump(past_rewards+current_rewards, outfile)

  model.save(path+path_model)

  print('data saved')


def load_model(model_type,env,path_model,tensorboard_path):
  try:
    model = model_type.load(path+path_model,env,tensorboard_log=tensorboard_path)
    print("model loaded")
  except:
    model = None
  finally:
    return model
```

I diversi environments per ogni algoritmo:

```
def get_env():
  return VecFrameStack(make_atari_env('PongNoFrameskip-v4', num_env=16, seed=0), n_stack=4) #for ACER
  return VecFrameStack(make_atari_env('PongNoFrameskip-v4', num_env=8, seed=0), n_stack=4) #for PPO2
  return make_atari_env('PongNoFrameskip-v4', num_env=1, seed=0) #for DDQN
```

Inizializzazione degli algoritmi:

```
#ACER
acer = "acer/"
model_name = acer+'acer'
reward_file = acer+'acer_rewards.txt'
tensorboard_path = path+acer
#PPO2
ppo2 = "ppo2/"
model_name = ppo2+'ppo2'
reward_file = ppo2+'ppo2_rewards.txt'
tensorboard_path = path+ppo2
#DDQN
ddqn = "ddqn/"
tensorboard_path = path+ddqn
model_name = ddqn+'ddqn'
reward_file = ddqn+'ddqn_rewards.txt'

past_rewards = load_previos_data(reward_file) #load previous data

# creation of the callback for data saving, every check_freq episodes
callback = SaveCallback(reward_file,model_name,past_rewards,check_freq=10000)
env = get_env()

#ACER
model = load_model(ACER,env,model_name,tensorboard_path)
#PPO2
model = load_model(PPO2,env,model_name,tensorboard_path)
#DDQN
model = load_model(DQN,env,model_name,tensorboard_path)


if model is None: #if no previous model was trained, a new one is created
  model = ACER('CnnPolicy', env, verbose=1,buffer_size=5000,ent_coef=0.01,lr_schedule='constant')

  model = PPO2('CnnPolicy', env, verbose=1,n_steps=128,noptepochs=4,nminibatches=4,
               learning_rate=2.5e-4,cliprange=0.1,vf_coef=0.5,ent_coef=0.01,cliprange_vf=-1)

  model = DQN('CnnPolicy',env,buffer_size=10000,exploration_final_eps=0.01,
              learning_rate=1e-4,learning_starts=10000,prioritized_replay=True,
              target_network_update_freq=1000,train_freq=4,
              verbose=1,tensorboard_log=tensorboard_path)
  print('model created')
```

Addestramento della rete:

```
#ACER
acer = "acer/"
model_name = acer+'acer'
reward_file = acer+'acer_rewards.txt'
tensorboard_path = path+acer
#PPO2
ppo2 = "ppo2/"
model_name = ppo2+'ppo2'
reward_file = ppo2+'ppo2_rewards.txt'
tensorboard_path = path+ppo2
#DDQN
ddqn = "ddqn/"
tensorboard_path = path+ddqn
model_name = ddqn+'ddqn'
reward_file = ddqn+'ddqn_rewards.txt'

past_rewards = load_previos_data(reward_file) #load previous data

# creation of the callback for data saving, every check_freq episodes
callback = SaveCallback(reward_file,model_name,past_rewards,check_freq=10000)
env = get_env()

#ACER
model = load_model(ACER,env,model_name,tensorboard_path)
#PPO2
model = load_model(PPO2,env,model_name,tensorboard_path)
#DDQN
model = load_model(DQN,env,model_name,tensorboard_path)


if model is None: #if no previous model was trained, a new one is created
  model = ACER('CnnPolicy', env, verbose=1,buffer_size=5000,ent_coef=0.01,lr_schedule='constant')

  model = PPO2('CnnPolicy', env, verbose=1,n_steps=128,noptepochs=4,nminibatches=4,
               learning_rate=2.5e-4,cliprange=0.1,vf_coef=0.5,ent_coef=0.01,cliprange_vf=-1)

  model = DQN('CnnPolicy',env,buffer_size=10000,exploration_final_eps=0.01,
              learning_rate=1e-4,learning_starts=10000,prioritized_replay=True,
              target_network_update_freq=1000,train_freq=4,
              verbose=1,tensorboard_log=tensorboard_path)
  print('model created')
```

5.2 Rete convoluzionale (CNN)

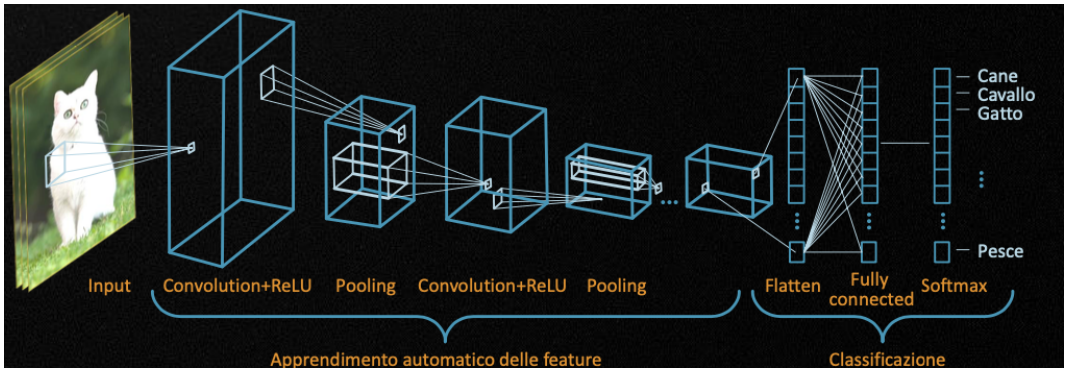

Figura 5.4: Tipica architettura di una rete neurale convoluzionale.
Fonte: Corso di Visone Artificiale Prof. Cappelli

Tramite una rete neurale convoluzionale è possibile classificare cosa un'immagine mostra e identificare con una certa probabilità il suo contenuto.

Durante l'addestramento alla rete vengono passate una serie di immagini che, attraversando diversi livelli specifici della rete, subiranno trasformazioni che daranno origine alle probabilità di ciascuna classe.

Ora verrà data una breve spiegazione di ogni livello.

5.2.1 Livello convoluzionale

È il livello principale della rete. Il suo obiettivo è quello di individuare schemi, come ad esempio curve, angoli, circonferenze o quadrati raffigurati in un'immagine con elevata precisione. Sono più di uno, e ognuno di essi si concentra nella ricerca di queste caratteristiche nell'immagine iniziale. Maggiore è il loro numero e maggiore è la complessità della caratteristica che riescono ad individuare.

Si introduce il concetto di **filtro**, una piccola matrice di poche righe e colonne. Ha lo scopo di estrarre certe features dall'immagine data come input o di applicare diverse operazioni, come sfocatura. F è chiamato filtro, maschera, o kernel. Di norma è quadrato, con dimensioni $m \times m$, m dispari. I suoi elementi sono chiamati coefficienti del filtro o pesi. Di solito si fa riferimento ai coefficienti del filtro considerando due assi cartesiani (con l'asse verticale diretto vero il basso) con origine in corrispondenza del centro.

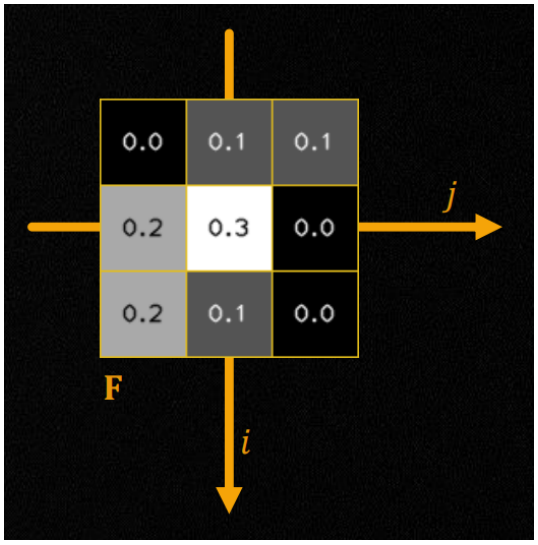

Figura 5.5: Esempio di filtro 3x3.
Fonte: Corso di Visone Artificiale Prof. Cappelli

L'applicazione si ottiene facendo la convoluzione tra il filtro e la matrice. L'operazione si può descrivere come il prodotto scalare tra il filtro e la sotto matrice della matrice cui si vuole applicare il filtro chiamato campo ricettivo. La formula per il calcolo del valore di un pixel nell'immagine destinazione può essere quindi scritta come:

$$I'[y,x] = \sum_{i=-d}^{d} \sum_{j=-d}^{d} F[i,j] \cdot I[y+i, x+j], \text{con d} = \left\lfloor \frac{m}{2} \right\rfloor$$

Si crea un'immagine risultato i cui valori dei diversi pixel sono il risultato dell'operazione. In realtà per di ottimizzazione, si effettua la correlazione, che si ottiene utilizzando il filtro negli assi.

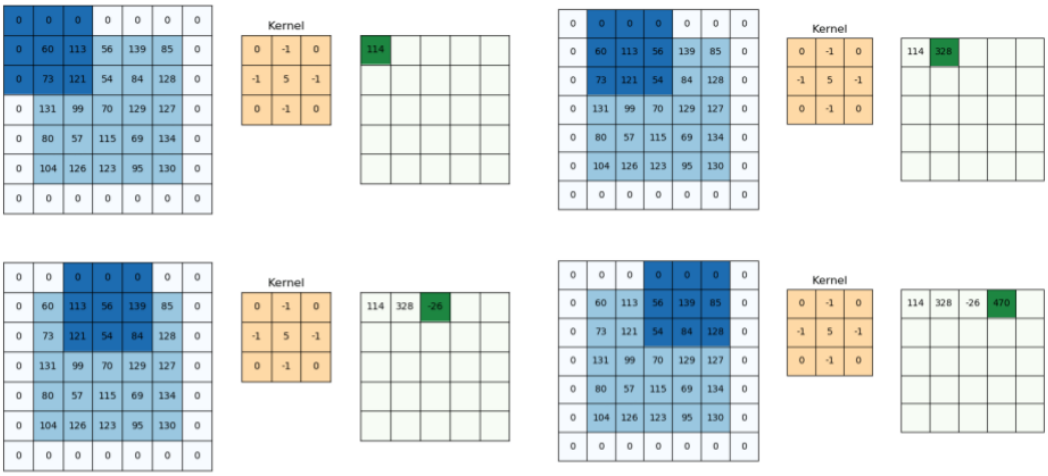

Figura 5.6: Qualche passo dell'applicazione del filtro nella matrice dell'immagine
Fonte: Neural Networks: Concepts and Details

Nei primi livelli viene utilizzato per identificare le **caratteristiche di basso livello**, come linee o curve. Man mano che si procede nei livelli successivi le features diventano sempre più complesse fino ad arrivare all'identificazione, per esempio, di mani o volti, **caratteristiche di alto livello**.

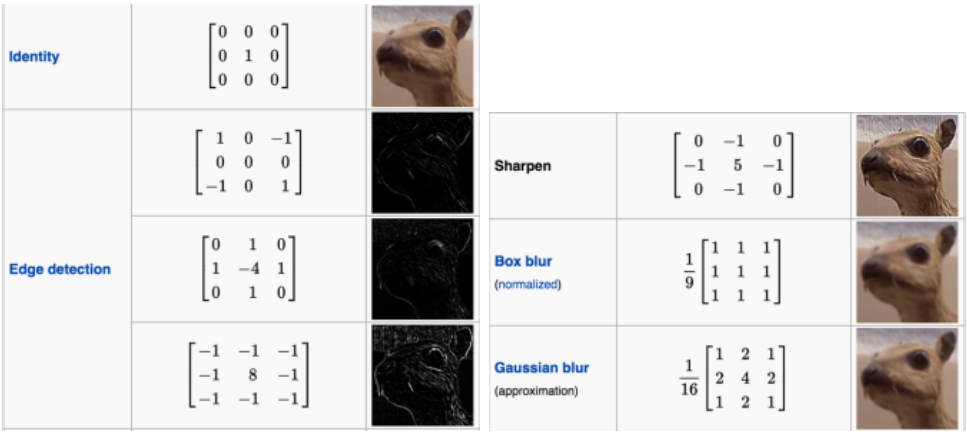

Figura 5.7: Esempi di risultato dell'applicazione di filtri comuni
Fonte: Wikipedia: Kernel (image processing)

Variando il **passo** (o **strike**) si ottengono matrici risultato diverse. Più è grande il passo più la matrice risultato è piccola.

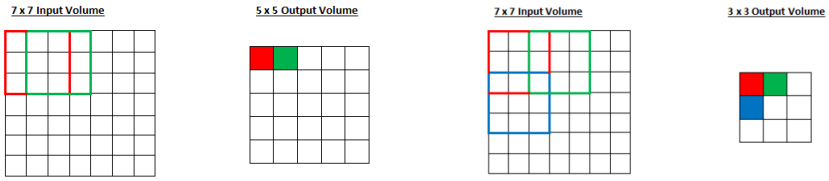

Figura 5.8: Diversi risultati dell'applicazione del filtro con passo 1 a sinistra e 2 a destra.
Fonte: A Beginner's Guide To Understanding Convolutional Neural Networks Part 2

Altro fattore importante è l'applicazione o meno del **padding**, cioè un bordo aggiuntivo da applicare alla matrice di input al fine di non perdere informazioni nell'immagine. Si ricorda che il risultato dell'operazione di convoluzione viene memorizzato nel valore centrale rispetto al filtro, quindi per i bordi l'unico modo per applicare l'operazione è aggiungendo un padding di pixel con un valore, solitamente 0 (**zero-padding**).

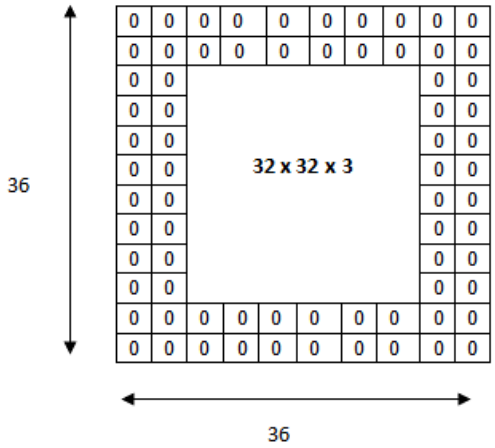

Figura 5.9: Esempio di zero-padding spesso 2 pixels perche il passo utilizzato è di 2.
Fonte: A Beginner's Guide To Understanding Convolutional Neural Networks Part 2

5.2.2 Livello ReLu (Rectified Linear Units)

È un livello posto solitamente successivamente e insieme al livello convoluzionale. Rappresenta un livello non lineare, il cui scopo è quello di introdurre la non linearità in un sistema che effettuando operazioni lineari durante i livelli convoluzionali (tramite il prodotto scalare tra il filtro e la porzione della matrice, il campo ricettivo).

Con questi livelli le reti neurali convoluzionali si addestrano molto più velocemente (grazie all'efficienza computazionale) senza impattare significativamente sull'accuratezza dei risultati.

Il livello ReLU applica la funzione

$$f(x) = max(0, x)$$

a tutti i valori nel volume di input.

In parole semplici, questo livello annulla tutti i valori negativi, aumentando le proprietà non lineari del modello e della rete globale senza influenzare i campi ricettivi del livello convoluzionale.

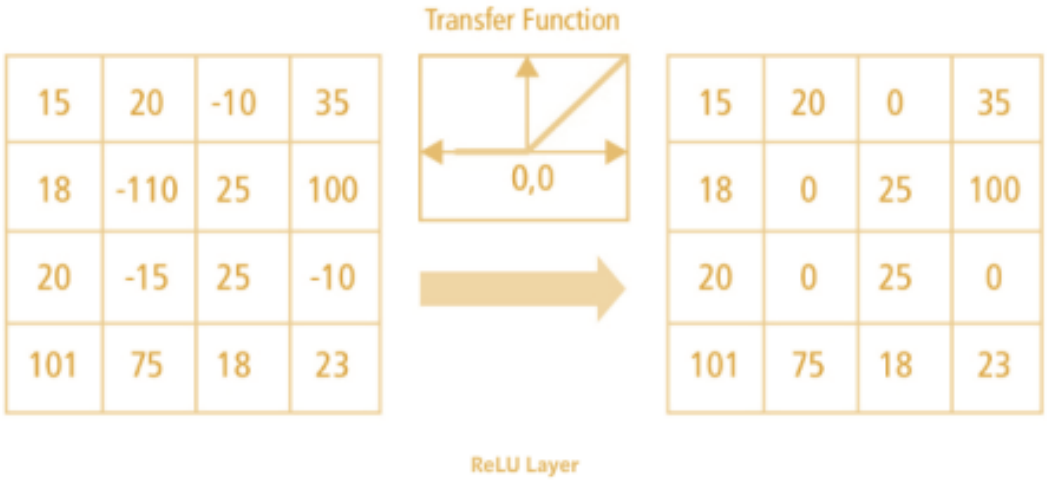

Figura 5.10: L'applicazione di ReLu, tutti i valori negativi vengono azzerati
Fonte: Understanding of Convolutional Neural Network (CNN) - Deep Learning

Ci sono altre funzioni non lineari come *tanh* o *sigmoid* ma hanno prestazioni minori.

5.2.3 Livello Pool

Dopo alcuni livelli ReLU si può applicare un livello pool che ha lo scopo di raggruppare e ridurre il numero di parametri quando le immagini sono troppo grandi e, di conseguenza, i requisiti computazionali ai livelli successivi. Il pooling può essere di diversi tipi:

- Il massimo
- La media
- La somma

Il massimo è il più popolare, e per questo si considererà quello.

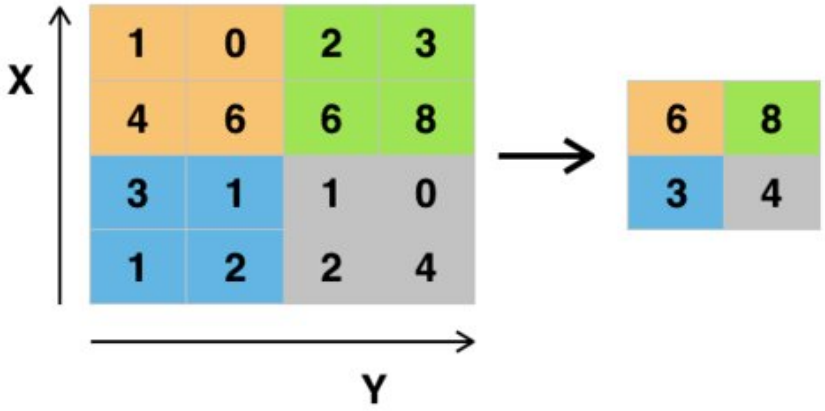

Figura 5.11: Pool di massimo con un filtro 2x2 e con passo 2. Per ogni gruppo viene estratto il valore massimo.
Fonte: Wikipedia: Convolutional neural network

Il pooling, chiamato anche **sotto-campionamento**, riduce la dimensionalità di ciascuna matrice ma conserva le informazioni importanti. Il ragionamento è che una volta che si sa che una caratteristica specifica è nell'input originale (ci sarà un alto valore di attivazione), è sufficiente sapere la sua posizione relativa rispetto alle altre caratteristiche.

5.2.4 Livello fully connected

È l'ultimo livello e effettua la classificazione vera e propria.

Questo livello prende l'input e genera un vettore N-dimensionale in cui N è il numero di classi nella classificazione. Ogni numero in questo vettore di dimensione N rappresenta la probabilità di una certa classe.

Il livello ottiene l'output della rete precedente e lo appiattisce rendendolo un vettore monodimensionale; successivamente calcola calcola i prodotti tra i pesi e il livello precedente per ottenere le probabilità corrette per le diverse classi in base a quali caratteristiche di alto livello sono maggiormente correlate ad una particolare classe.

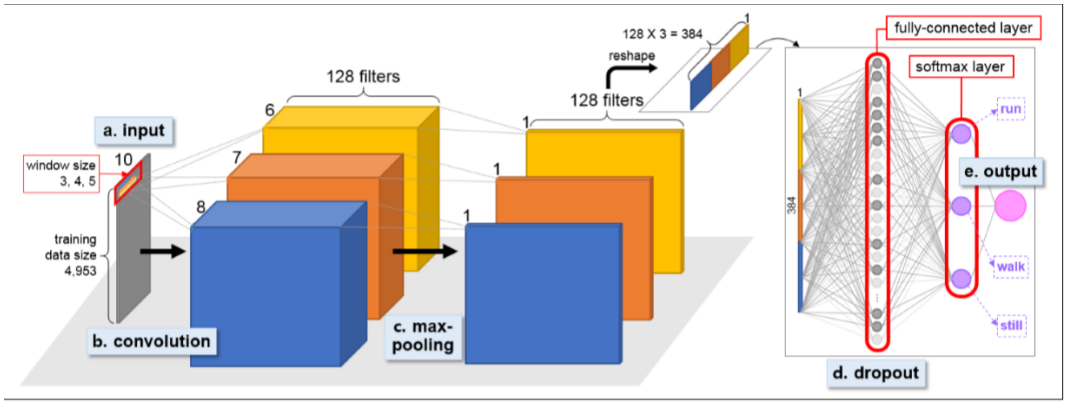

Figura 5.12: Si noti la parte finale, in cui l'output dell'ultimo strato viene appiattito e passato ad uno stato denso

Fonte: Fonte: Pubblicato nel 2017 IEEE International Conference on Big Data and Smart Computing (BigComp). Human activity recognition from accelerometer data using Convolutional Neural Network S. Lee, S. Yoon, H. Cho

Ogni neurone della rete effettua prodotto scalare tra i suoi pesi e i valori che riceve in input. Dopodiché viene utilizzata la funzione $softmax$ cosi definita:

$$softmax(x) = log(1 + e^x)$$

per determinare, per ogni classe, la probabilità della classificazione.

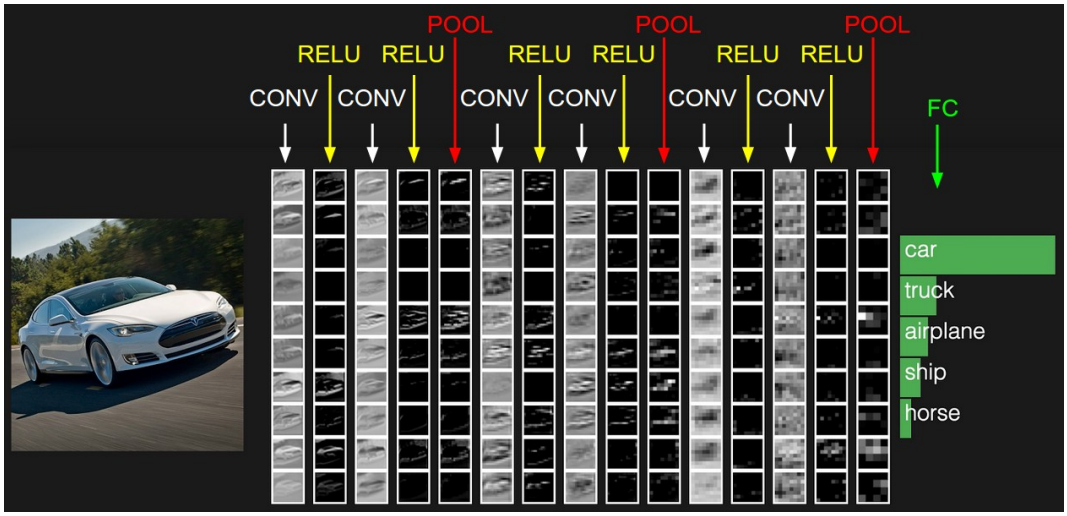

Figura 5.13: Nell'immagine vengono mostrati gli output di ogni layer della rete e infine come viene effettuata la classificazione

Fonte: CS231n Convolutional Neural Networks for Visual Recognition

5.3 Introduzione a DQN, processo decisionale di Markov

Come esempio, si consideri un agente situato in un ambiente. L'ambiente è in un certo stato. L'agente può eseguire determinate azioni nell'ambiente. Queste azioni a volte si traducono in una ricompensa (ad es. Aumento del punteggio che può essere sia positivo sia negativo). Le azioni trasformano l'ambiente e portano a un nuovo stato, in cui l'agente può eseguire un'altra azione e così via. Le regole su come scegliere queste azioni sono chiamate policy. L'ambiente in generale è stocastico, il che significa che lo stato successivo può essere alquanto casuale (ad esempio, la perdita di una palla e il lancio una nuova in direzione casuale).

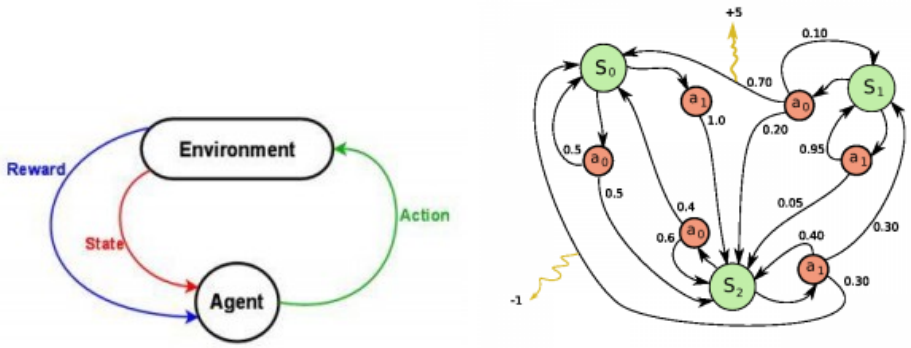

Figura 5.14: A sinistra: architettura apprendimento RL. A destra: processo decisionale di Markov.
Fonte: Sinistra: [13],Destra Wikipedia: Markov decision process

L'insieme degli stati e delle azioni, insieme alle regole per passare da uno stato all'altro e per ottenere ricompense, costituiscono un processo decisionale markoviano. Un episodio di questo processo (ad esempio un gioco) forma una sequenza finita di stati, azioni e ricompense:

$$s_0, a_0, r_1, ..., s_t, a_t, r_{t+1}$$

Qui s_i rappresenta lo stato, a_i è l'azione e r_{i+1} è la ricompensa dopo aver eseguito l'azione. L'episodio termina con uno stato terminale (es. schermata "game over"). Un processo decisionale markoviano si basa sul presupposto markoviano, ovvero la probabilità del prossimo stato s_{i+1} dipende solo dallo stato attuale s_i e ha eseguito l'azione a_i, ma non su stati o azioni precedenti.

5.3.1 Concetto di Discounted Future Reward

Per ottenere buoni risultati a lungo termine, bisogna tenere in considerazione non solo i premi immediati, ma anche i premi futuri che si ricevono.

Data una serie di processi decisionali di Markov, possiamo facilmente calcolare la ricompensa totale per un episodio, da un istante t:

$$R_t = r_t + r_{t+1} + ... + r_T$$

Ma poiché l'ambiente è stocastico, non si può essere sicuri se si otterranno le stesse ricompense la prossima volta che si eseguono le stesse azioni. Più andiamo nel futuro, più potrebbe divergere. Per questo motivo è comune utilizzare ricompense future scontate :

$$R_t = r_t + \gamma\, r_{t+1} + ... + \gamma^{T-1} r_T = r_t + \gamma(r_{t+1} + \gamma(r_{t+2} + ...)) = r_t + \gamma R_{t+1}$$

Se impostiamo il fattore di sconto $\gamma = 0$, si considerano solo le ricompense immediate. Un equilibrio tra ricompense immediate e future si può ottenere impostando $\gamma = 0,9$. Se l'ambiente è deterministico e le stesse azioni producono sempre gli stessi premi, allora si può impostare $\gamma = 1$.

La strategia per un agente è quella di scegliere l'azione che massimizza la ricompensa futura scontata.

5.3.2 Q-learning

In Q-learning si definisce una funzione che rappresenta la ricompensa futura scontata quando eseguiamo un'azione a nello stato s e continua in modo ottimale da quel punto in poi.

$$Q(a_t, s_t) = max R_{t+1}$$

Il modo di pensare $Q(s, a)$ è che è "il miglior punteggio possibile alla fine del gioco dopo aver eseguito l'azione a nello stato s". Si chiama funzione Q, perché rappresenta la "qualità" di una certa azione in un dato stato.

Supponi di essere nello stato s e riflettere sul effettuare l'azione a o b. L'obbiettivo è quello di selezionare l'azione che si traduce nel punteggio più alto alla fine del gioco. Ipotizzando di possedere la funzione Q, basta scegliere l'azione che permette di ottenere il valore Q più alto.

$$\pi(a) = argmax_a Q(s, a)$$

dove π è la policy, la regola con la quale si sceglie l'azione da effettuare [14].

Proprio come con i future discounted rewards si può esprimere il valore Q di stato s e azione a in termini di valore Q del prossimo stato s'.

$$Q^*(s,a) = E_{s' \sim \epsilon}\left[r + \gamma max_{a'} Q^*(s', a')\right] \text{ [12]}$$

Questa è chiamata **equazione di Bellman**, che modella la massima ricompensa futura per questo stato e l'azione è la ricompensa immediata più la massima ricompensa futura per lo stato successivo.

L'idea principale in Q-learning è che si può approssimare iterativamente la funzione Q usando l'equazione di Bellman. Nel caso più semplice la funzione Q è implementata come una tabella, con gli stati come righe e le azioni come colonne.

Si rappresenta quindi la funzione Q con una rete neurale che prende lo stato (per esempio schermate di gioco) e l'azione come input e restituisce il valore Q corrispondente. In alternativa, si utilizzano solo le schermate di gioco come input per produrre il valore Q per ogni possibile azione. Questo approccio ha il vantaggio che se vogliamo eseguire un aggiornamento del valore Q o scegliere l'azione con il valore Q più alto, basta effettuare solo un passo nella rete ed avere tutti i valori Q per tutte le azioni immediatamente disponibili.

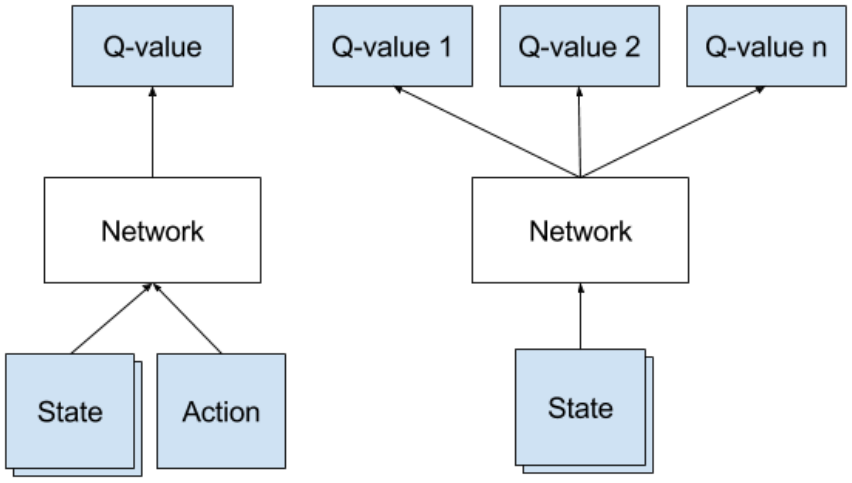

Figura 5.15: A sinistra: formulazione semplificata della rete Q profonda. A destra: architettura più ottimale della rete Q profonda, utilizzata nel documento DeepMind.

Fonte: DEMYSTIFYING DEEP REINFORCEMENT LEARNING

Data una transizione $< s, a, r, s' >$, la regola di aggiornamento della tabella Q diventa:

1. Eseguire un passaggio feedforward per lo stato corrente s per ottenere i valori Q previsti per tutte le azioni.

2. Effettuare un ulteriore passaggio feedforward per lo stato successivo s' e calcolare il massimo su tutte le uscite di rete $max_{a'}Q(s', a')$.

3. Si imposta il valore Q per l'azione a con $r + \gamma max_{a'}Q(s', a')$ (si utilizza il massimo calcolato nel passaggio 2). Per tutte le altre azioni, si imposta il valore Q uguale a quello restituito originariamente dal passaggio 1, rendendo l'errore 0 per quelle uscite.

4. Si aggiornano i pesi usando backpropagation

5.3.3 Exploration vs Exploitation

Il Q-learning tenta di risolvere il **problema dell'assegnazione del credito**: propaga i premi indietro nel tempo, fino a raggiungere il punto decisionale cruciale che era la causa effettiva del premio ottenuto.

In primo luogo, una rete Q viene inizializzata in modo casuale, con la conseguenza che anche le sue previsioni sono inizialmente casuali. Se si sceglie un'azione con il valore Q più alto, l'azione sarà casuale e l'agente esegue una rudimentale "esplorazione". Quando una funzione Q converge, restituisce valori Q più coerenti e la quantità di esplorazione diminuisce. Quindi si potrebbe dire che il Q-learning incorpora l'esplorazione come parte dell'algoritmo. Ma questa esplorazione termina con la prima strategia efficace che trova.

Una soluzione semplice è l'utilizzo di un fattore ϵ. Con probabilità ϵ si scegle un'azione casuale, altrimenti scegli l'azione con il valore Q più alto. Il valore di ϵ può essere diminuito nel tempo da 1 a 0,1. Così facendo all'inizio il sistema fa mosse completamente casuali per esplorare lo spazio degli stati, e poi diminuisce gradualmente l'esplorazione rinforzando le strategie che ha già trovato.

5.4 Double Q Network

Nella funzione Q,

$$Q^*(s, a) = E_{s' \sim \epsilon}\left[r + \gamma max_{a'}Q^*(s', a')\right] \text{ [12]}$$

ed in particolare il termine $max_{a'}Q^*(s', a')$, porta alla sovrastima del valore Q sugli stati. Dato che il Q-learning apprende utilizzando delle stime, questo porta diversi problemi.

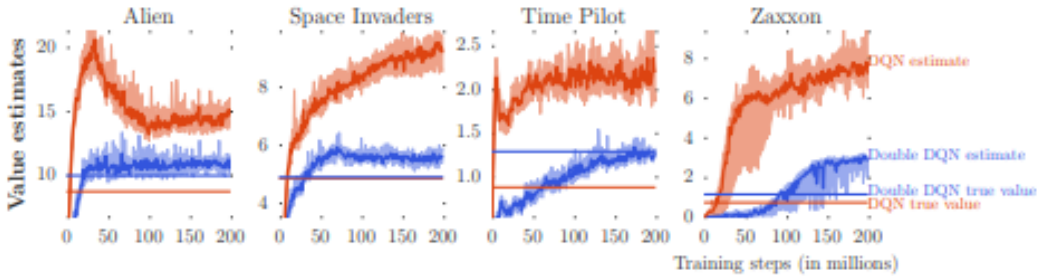

Figura 5.16: Il grafico mostra come i valori stimati con DQN si discostano anche molto dal valore reale, mentre quelli stimati utilizzando DDQN rimangono molto più stabili.Fonte [2]

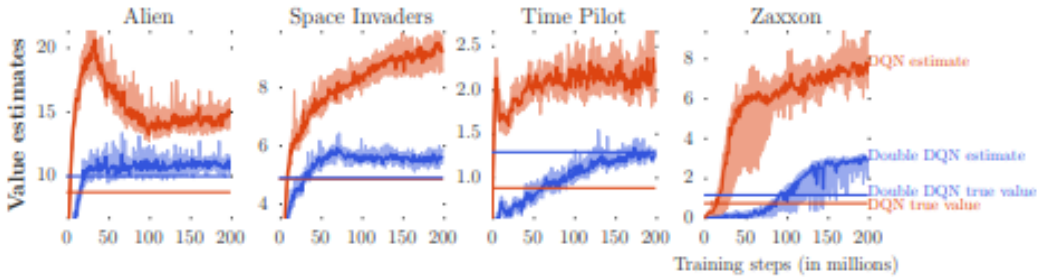

Figura 5.17: Una sovrastima dei valori portano ad un apprendimento molto instabile e di bassa qualità. Fonte [2]

La soluzione prevede l'utilizzo di due stimatori del valore Q separati, ciascuno dei quali viene utilizzato per aggiornare l'altro. Usando questi stimatori indipendenti, si stima il valore Q delle azioni selezionate usando lo stimatore opposto [15].

5.4.1 Prima versione, Hasselt, 2010

Nella prima versione dell'algoritmo (*"Double Q-learning" (Hasselt, 2010 [16])* l'algoritmo di apprendimento utilizza due stime indipendenti Q^A e Q^B. Con una probabilità di 0.5 si utilizza la stima Q^A per determinare l'azione di massimizzazione aggiornando Q^B. Al contrario, si usa Q^B per determinare l'azione di massimizzazione aggiornando Q^A. In questo modo si ottiene uno stimatore imparziale per il valore Q atteso inibendo il bias.

5.4.2 Seconda versione, Hasselt, 2015

Nella versione aggiornata dello stesso autore (*"Deep Reinforcement Learning with Double Q-learning" (Hasselt et al., 2015) [17]* si ha un modello Q un modello target Q', invece di due modelli indipendenti. Si usa Q' per la selezione dell'azione e Q per la valutazione dell'azione. Questo è:

$$Q^*(s_t, a_t) \approx r_t + \gamma Q(s_{t+1}, argmax_{a'} Q'(s_t, a_t))$$

Si riduce al minimo l'errore quadratico medio tra Q e Q*, ma si ha che Q' lentamente copia i parametri di Q. Questo si può realizzare copiando periodicamente i parametri o tramite la media Polyak:

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$$

dove θ' è il parametro di rete di destinazione, θ è il parametro di rete principale e τ (tasso di media) è solitamente impostato su 0.01.

5.4.3 Clipped Double Q-learning, Fujimoto, 2018

Nell'ultima versione, chiamata **Clipped Double Q-learning** (*"Addressing Function Approximation Error in Actor-Critic Methods" (Fujimoto et al., 2018 [3])*, si segue la formulazione originale di Hasselt 2015. Si hanno due stime indipendenti del valore Q reale. Per calcolare il valore di aggiornamento si considera il minimo dei due valori di azione dello stato successivo prodotti dalle due reti Q; Quando la stima Q di uno è maggiore dell'altra la si minimizza, evitando la sovrastima.

Algorithm 1 : Clipped Double Q-learning (Fujimoto et al., 2018)

Initialize networks Q_{θ_1}, Q_{θ_1}, replay buffer $\mathcal{D}$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
  for each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma min_{i=1,2} Q_{\theta_i}(s_{t+1}, argmax_{a'} Q_{\theta_i}(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters:
      $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$

Figura 5.18: Algoritmo Clipped Double Q-learning [3]

Fujimoto presenta un altro vantaggio di questa impostazione: l'operatore minimo dovrebbe fornire un valore maggiore agli stati con un errore di stima della varianza inferiore. Ciò significa che la minimizzazione porterà a una preferenza per gli stati con stime di valore a bassa varianza, portando ad aggiornamenti delle policy più sicuri con obiettivi di apprendimento stabili.

5.5 Proximal Policy Optimization (PPO)

Dietro l'idea di PPO si cerca un equilibrio tra facilità di implementazione, efficienza del campione e facilità di sintonizzazione [10]. PPO utilizza **policy optimization**, non utilizza il replay buffer per memorizzare esperienza ma effettua un apprendimento online, cioè apprende con tutto ciò che l'agente incontra nell'ambiente e una volta che il batch di esperienza viene utillizzata per aggiornare i pesi della rete viene scartata. Questo implica che i metodi policy gradient sono in genere meno efficiente rispetto a Q learning perché usano solo l'esperienza raccolta una volta per fare un aggiornamento e la nostra politica generale. La funzione di loss viene solitamente cosi definita:

$$L^{PG}(\theta) = E\left[log\, p(a_t, s_t) A_t\right]$$

dove $log\, p(a_t, s_t$ è il logaritmo della policy valutata sull'azione a; è una rete neruale che prende lo stato dall'ambiente come inpunt e ricava l'azione da intraprendere come output. A_t che rappresenta una stima del valore relativo all'azione scelta.

Per calcolare il vantaggio occorre la somma scontata delle ricompense (**discounted sum of rewards o return**) e una stima della baseline (o **value function**).

La somma scontata delle ricompense è il prodotto tra la somma pesata di tutte le ricompense che l'agente ha ottenuto durante ogni passo nell'episodio corrente e un valore di sconto $\gamma \in [0, 1]$, solitamente tra 0.9 e 0.99 dato che all'agente interessano più le ricompense che può tenere nell'immediato piuttosto che dopo molte azioni.

Il valore di vantaggio è calcolato dopo che l'intero episodio è stato memorizzato, quindi si conoscono tutte le scelte dell'agente e tutte le ricompense che queste hanno portato.

Nella seconda parte, la funzione di valore, rappresentata con una rete neurale, cerca di dare una stima del ritorno da quel punto in avanti partendo dallo stato corrente. Durante l'addestramento, la rete che rappresenta la funzione di valore viene frequentemente aggiornata utilizzando l'esperienza accumulata dall'agente. L'aggiornamento corrisponde ad un problema di supervised learning, in quanto per l'addestramento si utilizza il valore del ritorno. La rete acquisisce in input lo stato e cerca di predire la discounted sum of rewards.

Dato che questo valoreè un output di una rete avrà una certa variazione e non il valore predetto non sarà esatto ma avrà del rumore.

Sottraendo questi due valori si ottiene la stima del vantaggio A_t che rappresenta è stata l'azione intrapresa dall'agente basata sull'aspettativa di ciò che accade normalmente nello stato in cui si trovava; in altre parole, l'azione intrapresa dall'agente è migliore di quanto atteso o no?

Se la stima del vantaggio è positiva allora significa Intrapresa dalla gente è migliore di quella media, aumentando quindi la probabilità di effettuarla di nuovo nel caso si trovasse nuovamente in quella situazione; l'opposto succede se il valore è negativo.

Il problema è che se si aggiorna la rete solo su un batch di dati accumulati c'è il rischio di aggiornare i parametri della rete molto fuori il range dove sono stati raccolti questi dati, per esempio la funzione di vantaggio, che è in linea di principio una stima rumorosa del vero vantaggio, sarà completamente sbagliata, andando a compromettere di molto la policy.

La soluzione consiste nell'assicurarsi che il valore dell'aggiornamento della della politica non si discosti molto da quella vecchia. Questa idea è stata introdotta nel **Trust Region Policy Optimization or TRPO**, che è la base di PPO.

La funzione obbiettivo è:

$$\underset{\theta}{\text{maximize}} \quad E\left[\frac{p(a_t, s_t)}{p_{old}(a_t, s_t)}\right]$$

Confrontando la funzione obiettivo definita prima si può notare che la cosa che cambia è il logaritmo viene sostituito con una divisione. Ottimizzare questa funzione equivale è equivalente alla versione base di policy gradient.

Per assicurarti che anche la politica aggiornata non si sposti lontano dall'attuale politica, TRPO aggiunge un vincolo KL all'ottimizzazione obiettivo e ciò che questo KL assicura che la nuova politica aggiornata non si allontani troppo dal quella vecchia.

$$E\left[KL\left[p_{old}(\cdot|s_t), p(\cdot|s_t)\right]\right] \leq \delta$$

Il problema è che questo vincolo aggiunge ulteriore overhead al processo di ottimizzazione e a volte può portare a molto indesiderabile comportamento nella fase di allenamento.

PPO risolve il problema includendo direttamente il vincolo all'interno nella funzione obbiettivo.

Per prima cosa si definisce la frazione indicata prima $\frac{p(a_t,s_t)}{p_{old}(a_t,s_t)}$ con $r_t(\theta)$ che esprime il rapporto di probabilità tra gli outputs della nuova policy e gli outputs della precedente. Così, data una sequenza di azioni campionate, il valore di $r_t(\theta)$ sarà maggiore di 1 se l'azione è più probabile ora rispetto alla vecchia versione della policy, e sarà un valore compreso tra 0 e 1 se l'azione è meno probabile ora rispetto all'ultimo aggiornamento con il gradiente.

Se moltiplichiamo questo rapporto $r_t(\theta)$ con

la funzione di vantaggio otteniamo funzione obbiettivo di TRPO in una forma più leggibile.

$$L^{CPI}(\theta) = E\left[r_t(\theta)A_t\right]$$

Con questa notazione si può scrivere la funzione obiettivo che viene usata in PPO:

$$L^{CPIP}(\theta) = E\left[min(r_t(\theta)A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)\right]$$

Prima di tutto puoi vedere la funzione obiettivo che PPO ottimizzata è un operatore di aspettativa. Ciò significa che il calcolo viene effettuato su batches e che prende il minimo tra due termini. Il primo di questi termini è $r_t(\theta)A_t$, qyuindi è la l'obbiettivo di default della normale policy gradient la quale spinge la policy verso azioni che producono un vantaggio positivo rispetto alla baseline. Il secondo termine è molto simile al primo eccetto che contiene la versione troncata di $r_t(\theta)$ applicando un'operazione di taglio tra $1-\epsilon$ e $1+\epsilon$, dove ϵ è un valore come 0.2. Infine l'operatore di minimo è applicato ai due termini per otternere il risultato finale.

In primo luogo, è importante notare che la stima del vantaggio A_t può essere sia positiva e negativa e questo cambia l'effetto dell'operatore principale.

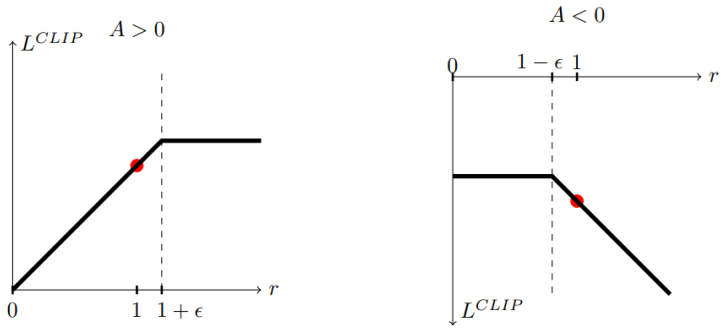

Figura 5.19: Grafici che mostrano un termine (cioè un singolo passo temporale) della funzione surrogata L^{CLIP} come una funzione del rapporto di probabilità r, per vantaggi positivi (sinistra) e vantaggi negativi (destra). Il cerchio rosso su ciascuno grafico mostra il punto di partenza per l'ottimizzazione, i.e. r = 1. Notare che L^{CLIP} somma molti di questi termini

Fonte: [10]

Un grafico con la funzione obbiettivo con i valori negativo e positivo del vantaggio stimato. Nella metà sinistra il valore del vantaggio è positivo o in tutti i casi in cui l'azione ha un valore migliore di quanto atteso; da notare come la funzione di loss viene appiattita quando il valore di $r_t(\theta)$ diventa troppo alto e questo succede quando l'azione è molto più probabile sotto la policy corrente rispetto a quella vecchia. In questo caso si limita la dimensione dell'aggiornamento tagliando il valore.

Nella metà desta viene rappresentata la situazione dove l'azione ha un valore stimato negativo sul risultato e la funzione obbiettivo viene schiacciata vicino allo zero. Questo corrisponde ad azioni che sono molto meno più probabili adesso rispetto alla vecchia policy e avrà lo stesso effetto eseguire un aggiornamento che porti queste azioni ad avere probabilità a zero.

Nelle parti non tagliate il prodotto tra i due termini è minore dei due vincoli e indica se e quanto rendere più o meno probabili le azioni svolte dall'agente.

TRPO e PPO impongono un vincolo ai valori per non esagerare con la dimensione degli aggiornamenti, con la differenza che la funzione obbiettivo di PPO è molto più semplice e non richiede di calcolare vincoli addizionali o divergenze KL e spesso supera la controparte più complessa.

Nell'algoritmo vengono utilizzati due threads alternati. Nel primo la policy corretne sta interagendo con l'ambiente generando sequenze di episodi per i quali viene subito calcolata la funzione di vantaggio utilizzando la la stima di base adeguata per il valore degli stati.

Dopo un certo numero di episodi un secondo thread utilizza tutta l'esperienza accumulata per effettuare la discesa del gradiente, utilizzando la funzione obbiettivo con i vincoli, aggiornando la rete.

La funzione di loss che viene utilizzata realmente per addestrare una rete utilizzando PPO è la combinazione di L^{CLIP} con altri due termini:

$$L_t^{CLIP} = E_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[p](s_t)]$$

Il primo termine $c_1 L_t^{VF}(\theta)$ è incaricato di aggiornare la baseline stimando quale il valore medio delle ricompense che ci si aspetta di avere da quel punto specifico.

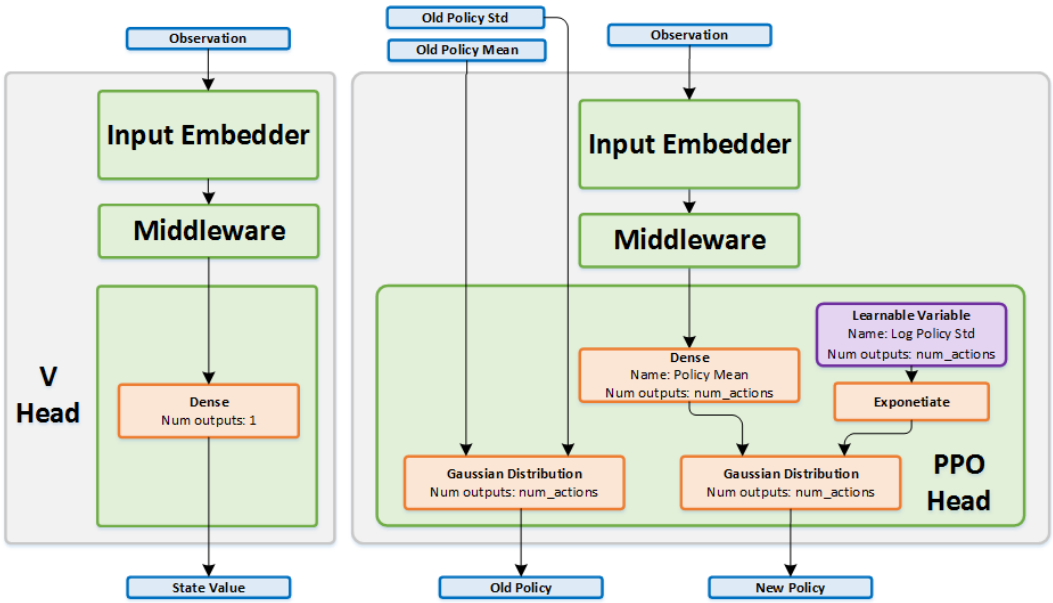

Figura 5.20: Struttura della rete PPO, con le due "teste" che la compongono, policy e valore

Fonte: Proximal Policy Optimization

Anche se i valori di ouput della policy e della funzione di valore formano due parti separate della stessa rete fanno parte dello stesso calcolo, quindi si può combinare tutto in una singola funzione di loss. Inoltre condividono una porzione grande dei parametri. Quando si prova a stimare il valore dello stato corrente oppure si vuole si vuole scegliere l'azione migliore da compiere probabilmente si avrà bisogno di una pipeline di estrazione di caratteristiche molto simili dall'osservazione dello stato attuale, quindi queste parti della rete sono semplicemente condivise.

L'ultimo termine nella funzione obiettivo $c_2 S[p](s_t)$ è chiamato **entropy** (entropia) e questo termine ha il compito di assicurarsi che l'agente esplori sufficientemente l'ambiente durante l'addestramento. In contrasto con le policy di azione discrete che producono come output le probabilità di scelta di quelle azioni, la parte della policy di PPO restituisce in uscita i parametri in una

distribuzione gaussiana per ogni tipo di azione disponibile e, quando si addestra l'agente, la policy campionerà quindi da queste distribuzioni per ottenere un valore di output continuo per ogni azione.

L'entropia di una variabile stocastica che è guidata da una certa distribuzione di probabilità è la quantità media di bit che è necessaria per rappresentare il suo risultato.

$$H(p) = -\sum_i p_i log_2(p_i)$$

È una misura di quanto sia davvero imprevedibile un risultato di questa variabile, e quindi massimizzare la sua entropia forzerà avere un'ampia diffusione su tutte le opzioni possibili, ottendendo un risultato più imprevedibile.

I due iperparametri c_1 e c_2 che variano i contributi di queste diverse parti nella funzione.

5.6 Actor Critic

Come specificato prima nell'esempio di Karpathy il *Policy Gradient* è:

$$\nabla_\theta J(\theta) = E\left[\sum_i A_i \, \nabla_\theta \, log \, p(y_i|x_i)\right]$$

D'ora in avanti verranno fatti i seguenti cambiamenti di notazione:

$$i \to t, A_i \to R_t, y_i \to a_t, x_i \to s_t$$

Dato che vengono effettuati aggiornamenti utilizzando campioni casuali si viene a creare un'elevata variabilità intrinseca nelle probabilità logaritmiche (logaritmo della distribuzione delle policy) e nei valori cumulativi di ricompensa.

Ciò produrrà gradienti rumorosi e causerà apprendimento instabile e/o la distribuzione delle policy distorta verso una direzione non ottimale.

Oltre all'elevata varianza dei gradienti anche il caso di ricompensa 0 è un problema, in quento non apporta nessun tipo di aggiornamento della rete, né favorendo azioni positive né sfavorire azioni negative.

Questi problemi contribuiscono all'instabilità e alla lenta convergenza dei metodi policy gradient base.

5.6.1 Come migliorare policy gradient?

Un modo per ridurre la varianza e aumentare la stabilità è sottrarre alla ricompensa cumulativa una baseline

$$\nabla_\theta J(\theta) = E\left[\sum_t (R_t - b(s_t)) \, \nabla_\theta \, log \, p(a_t|s_t)\right]$$

Ciò produrrà gradienti più piccoli e quindi aggiornamenti più piccoli e più stabili. Il valore della baseline dipende dal tipo di funzione utilizzata, come la prestazione attesa o in altri termini la prestazione media.

Prima di tornare alla baseline, si introduce l'architettura Actor Critic.

Si può scomporre l'aspettativa del policy gradient base.

$$\nabla_\theta J(\theta) = E_{s_0,a_0,\ldots,s_t,a_t}[\sum_t \nabla_\theta \, log \, p(a_t|s_t)] E_{r_{t+1},s_{t+1},\ldots,r_T,s_T}[R_t]$$

Il secondo termine è il valore Q

$$E_{r_{t+1},s_{t+1},\ldots,r_I,s_I}[R_t] = Q(s_t, a_t)$$

e quindi l'equazione diventa:

$$\nabla\theta J(\theta) = E\left[\sum_t Q_w(s_t, a_t) \, \nabla_\theta \, log \, p(a_t|s_t)\right]$$

Come sappiamo, il valore Q può essere appreso parametrizzando la funzione Q con una rete neurale (indicata con pedice w sopra). Questo ci porta al metodo **Actor Critic** [11], dove:

1. Il "Critic", o critico, stima la funzione valore. Questo potrebbe essere il valore dell'azione (il valore Q) o il valore dello stato (il valore V).

2. L'"Actor", o attore, aggiorna la distribuzione delle politiche nella direzione suggerita dal Critic (come con i policy gradient).

ed entrambe le funzioni Critic e Actor sono parametrizzate con reti neurali. Nella derivazione sopra, la rete neurale Critic parametrizza il valore Q, quindi si chiama **Q Actor Critic**.

La base $b(s)$ viene sostituita dalla funzione V, che è il valore dello stato, o la media di tutte le ricompense (causate da tutte le azioni intraprese) nello stato s.

Si sottrae al valore di Q il valore V. Intuitivamente, questo significa quanto è meglio intraprendere un'azione specifica rispetto all'azione media e generale in un dato stato. Questo valore viene chiamato **advantage** (vantaggio):

$$A(s_t, a_t) = Q_w(s_t, a_t) - V_v(s_t)$$

Bisogna creare due reti neurali w e v? No. Sarebbe molto inefficiente. Invece, si può usare la relazione tra la Q e la V dall'equazione di ottimalità di Bellman:

$$Q(s_t, a_t) = E\left[r_{t+1} + \gamma V(s_{t+1})\right]$$

Quindi il vantaggio si puó scrivere come:

$$A(s_t, a_t) = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_{t+1})$$

Quindi, si usa solo una rete neurale per la funzione V (parametrizzata v sopra). Quindi possiamo riscrivere l'equazione di aggiornamento come:

$$\nabla_\theta J(\theta) \sim \sum_t \nabla_\theta \, log \, p(a_t|s_t)(r_{t+1}+\gamma V_v(s_{t+1})-V_v(s_{t+1})) = \sum_t \nabla_\theta \, log \, p(a_t|s_t) A(s_t, a_t)$$

Questo è **Advantage Actor Critic** dove $\nabla_\theta \, log \, p(a_t|s_t)$ è l'*Actor* e $A(s_t, a_t)$ è il *Critic*

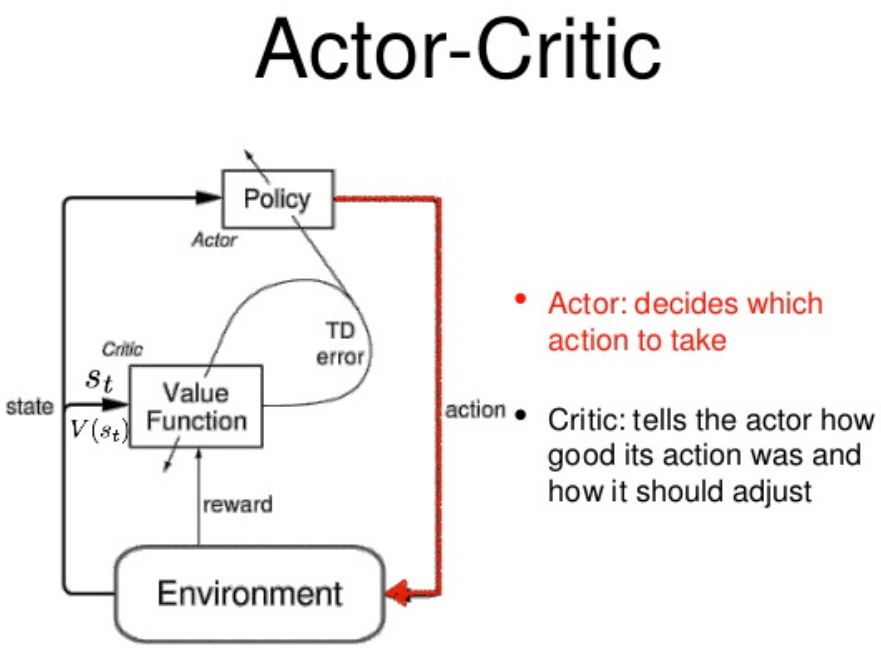

Figura 5.21: Architettura Actor Critic
Fonte: Sutton and Barto, 1998.

5.7 Experience replay

Con le reti profonde, si utilizza spesso la tecnica chiamata *experience replay* durante l'addestramento. Si salvano le esperienze dell'agente in ogni fase temporale in una memoria che rappresenta un set di dati chiamato *replay buffer*,

tipicamente una tabella. Nella pratica la memoria ha una dimensione finita, e quindi memorizzerà solo le ultime N esperienze.

L'esperienza dell'agente si può rappresentare come:

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

Questa tupla contiene lo stato dell'ambiente s_t, l'azione a_t presa dallo stato s_t, la ricompensa dato all'agente al momento $t-1$ come risultato della precedente coppia stato-azione (s_{t-1}, a_{t-1}) e il successivo stato dell'ambiente s_t.

5.7.1 Perché utilizzare *experience replay*

Perché si sceglie di addestrare la rete su campioni casuali presi dalla memoria di replay, piuttosto che fornire alla rete le esperienze sequenziali che si verificano nell'ambiente?

I campioni che si verificano in sequenza nell'ambiente sono altamente correlati e ciò causerebbe un addestramento della rete inefficiente, facendola convergere in un minimo locale. Il prelievo di campioni casuali dalla memoria di riproduzione interrompe questa correlazione.

Inizialmente si accumula nella memoria esperienza (le azioni possono essere selezionate in base ad una certa *policy*,e.g. soft-max per uno spazio ad azioni discrete, Gaussian per uno continuo etc.) finché non raggiunge una certa dimensione. A questo punto

5.7.2 Prioritized replay

Nel *replay con priorità* insieme all'esperienza ottenuta si salva anche un valore di priorità che rappresenta il *learning value* di ogni tupla.

Come si ricava questo valore ?

$$error = Q(s, a) - q\_target$$

dove q_target è calcolato utilizzando l'equazione di Bellman

$$Q(s, a) = r_{t+1} + \gamma \cdot max(Q(s_{t+1}))$$

Il valore di priorità si calcola come

$$p = |error| + offset$$

L'*offset* ha il compito di evitare che il valore di priorità valga 0 (nel caso l'errore fosse nullo) perché l'esperienza potrebbe essere utile in futuro.

Si converte valore di priorità in una probabilità che quella tupla venga scelta.

$$Pr(i) = \frac{p_i}{\sum p}$$

Ma poiché le priorità provengono da un errore che dipende dall'approssimazione del valore Q_target approssimato la priorità potrebbe avere errori di approssimazione che potrebbero portare alcune probabilità ad essere troppo alte o troppo basse.

Per evitare questo si eleva ciascuna probabilità di un fattore α nell'intervallo $[0, 1]$ dove $\alpha = 1$ corrisponde a campionamento con priorità completo e $\alpha = 0$ diventa il puro campionamento casuale dell'esperienza.

$$Pr(i) = \frac{{p^\alpha}_i}{\sum p^\alpha}$$

La selezione di esperienze in modo non uniforme rende la rete più propensa a tendere troppo verso esperienze con priorità più alta (con la *replay buffer* si sta approssimando l'ambiente, che però restituisce stati con distribuzione uniforme), causando *overfitting*. L'aggiornamento dei pesi della rete viene scalato di un fattore w che dipende dalla probabilità che quell'esperienza venga scelta.

$$\theta \leftarrow \theta - \gamma \cdot \nabla\theta J(\theta) \cdot w$$

con γ *learning rate* e

$$w_i = \frac{1}{N} \cdot \frac{1}{Pr(i)}$$

con N la grandezza totale del replay buffer.

Conclusioni e sviluppi futuri

Il mio approccio al Reinforcement Learning è avvenuto seguendo il lavoro svolto da Andrej Karpathy, analizzando e migliorando l'addestramento di una rete neurale profonda costruita senza l'ausilio di librerie se non quelle matematiche offerte da Python. Utilizzando come base il suo lavoro ho implementato una versione che aggiunge il concetto di rewards assegnata tocco della pallina con la racchetta nelle prime fasi di gioco, grazie all'implementazione di un algoritmo che verifica il tocco.

Successivamente mi sono cimentato nella realizzazione di una soluzione utilizzando diversi algoritmi di diversa complessità ottenendo un drastico miglioramento nella convergenza.

Per concludere penso che utilizzerò quello che ho appreso nel mio futuro ambito lavorativo in ambito aziendale per sviluppare sistemi software basati su Reinforcement Learning basati per esempio sulla robotica o sulla approssimazione di problemi considerati np-hard, o anche in ambito videludico.

Ringraziamenti

Il primo ringraziamento va ai miei compagni di università con i quali ho condiviso momenti per me molto importanti nel corso di questi 3 anni di studi.

Il mio secondo ringraziamento va al mio professore e relatore, il Prof. Gianluca Moro, che ha reso possibile tutto ciò e mi ha fornito le basi e l'entusiasmo per questo incredibile argomento che è il Reinforcement Learning.

Il mio terzo ringraziamento va alla mia famiglia, al supporto e l'affetto che mi ha dato.

Bibliografia

[1] Andrej Karpathy. Deep reinforcement learning: Pong from pixels. `http://karpathy.github.io/2016/05/31/rl`, May 2016.

[2] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 2094–2100. AAAI Press, 2016.

[3] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *CoRR*, abs/1802.09477, 2018.

[4] A. Géron and an O'Reilly Media Company Safari. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition*. O'Reilly Media, Incorporated, 2019.

[5] 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.

[6] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and D. Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*, volume 9 of *JMLR Proceedings*, pages 249–256. JMLR.org, 2010.

[7] John Schulman, Nicolas Heess, Theophane Weber, and Pieter Abbeel. Gradient estimation using stochastic computation graphs. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 3528–3536, 2015.

[8] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1):145–151, 1999.

[9] Samrat Mukhopadhyay. Stochastic gradient descent for linear systems with sequential matrix entry accumulation. *Signal Process.*, 171:107494, 2020.

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

[11] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Rémi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[13] Zhiwen Tang and Grace Hui Yang. A reinforcement learning approach for dynamic search. In Ellen M. Voorhees and Angela Ellis, editors, *Proceedings of The Twenty-Sixth Text REtrieval Conference, TREC 2017, Gaithersburg, Maryland, USA, November 15-17, 2017*, volume 500-324 of *NIST Special Publication*. National Institute of Standards and Technology (NIST), 2017.

[14] Yunhao Tang and Shipra Agrawal. Discretizing continuous action space for on-policy optimization. *CoRR*, abs/1901.10500, 2019.

[15] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 2094–2100. AAAI Press, 2016.

[16] Hado van Hasselt. Double q-learning. In John D. Lafferty, Christopher K. I. Williams, John Shawe-Taylor, Richard S. Zemel, and Aron Culotta, editors, *Advances in Neural Information Processing Systems 23: 24th Annual Conference on Neural Information Processing Systems 2010. Proceedings of a meeting held 6-9 December 2010, Vancouver, British Columbia, Canada*, pages 2613–2621. Curran Associates, Inc., 2010.

[17] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.