Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Ingegneria e Scienze Informatiche

Behavioural Cloning in Ambiente Simulato con Reti Neurali

Candidato:
Alberto Serluca
Matricola 0000731577

Relatore:
Dott.ssa
Damiana Lazzaro

Sessione III

Anno Accademico 2016 - 2017

Dedico questo lavoro alla mia famiglia e ai miei amici più cari,
le persone che più mi hanno aiutato a farmi strada
in questo difficile ma gratificante percorso.

Introduzione

Questa tesi descrive lo studio e l'implementazione di un sistema basilare di autopilota in cui vengano coniugati aspetti di *Computer Graphics* e tecnologie innovative del *Machine Learning*, nello specifico il *"Behavioural Cloning"* [1], una metodologia di apprendimento supervisionato in cui le capacità cognitive possono essere carpite e riprodotte da un programma, attraverso l'imitazione. Perché ciò avvenga è necessario possedere un record di una serie di azioni logiche che un soggetto ha precedentemente registrato e memorizzato. Il programma di apprendimento utilizza queste informazioni per costruire una funzione che mappi l'input di un certo stato, al suo corrispondente output.

Il progetto è suddiviso in tre parti:

- **Simulatore**: Un software di grafica 3D necessario per emulare un ambiente di guida reale. Grazie a questa applicazione è possibile registrare le azioni dell'utente e salvarle su disco, in modo che l'intelligenza artificiale abbia a disposizione i dati su cui essere allenata. Il simulatore è inoltre utilizzato per mettere in atto l'autopilota del veicolo, una volta che esso è stato addestrato, verificandone così le capacità di clonazione comportamentale.

- **Script Python - Training**: Questo script contiene il modello di rete neurale che viene usato per il training; svolge inoltre la funzione di generazione del dataset a partire dalle immagini salvate tramite il simulatore ed effettua l'addestramento della rete.

- **Script Python - Prediction**: Questo codice è utilizzato dal simulatore per ottenere le predizioni di sterzo ed acceleratore. Esso carica il modello neurale precedentemente salvato e predispone un sistema di comunicazione col simulatore in modo da ricevere immagini in input e trasmettere i risultati predetti in output.

Per la creazione del simulatore è utilizzato Unity, un game-engine che permette una veloce iterazione tramite assets e funzionalità avanzate; ciò unito all'uso del linguaggio C-Sharp rende estremamente rapida la creazione di applicazioni anche complesse in grafica 3D. L'aspetto di intelligenza artificiale è invece affidato alla libreria Keras, questa permette di realizzare modelli neurali in modo semplice, ed è ritenuta uno dei migliori punti di partenza per l'apprendimento del machine learning. La struttura di questa tesi è suddivisa nelle seguenti sezioni:

- **Le Reti Neurali Artificiali**: Descrive il sistema nervoso umano e come esso abbia portato all'invenzione delle *Artificial Neural Networks* (ANN). Vengono poi visti i primi modelli computazionali ed i paradigmi di apprendimento artificiale.

- **CNNs**: Questo capitolo sulle CNNs (Convolutional Neural Networks) descrive le reti neurali convoluzionali, elemento chiave nella computer vision odierna.

- **LSTMs**: Descrizione dei concetti e del funzionamento delle reti ricorrenti di tipo LSTM (Long Short-Term Memory).

- **Keras e Python**: Caratteristiche del linguaggio e funzionalità del framework Keras.

- **Game-engine Unity**: Vengono illustrati gli aspetti più importanti del motore di gioco con particolare attenzione agli elementi utilizzati di più nella tesi.

- **Design del progetto**: Questa parte è riservata al design del sistema ed alla sperimentazione dei vari modelli neurali. Viene esposto il codice utilizzato e vengono presi in considerazione studi sul Behavioural Cloning, valutando nuovi metodi per migliorare le prestazioni del simulatore come l'aggiunta di una rete LSTM al modello base.

Indice

Elenco delle figure

Capitolo 1

Le Reti Neurali Artificiali

Nonostante i progressi tecnologici e l'aumento esponenziale di potenza computazionale degli ultimi anni, persistono problemi non banali nella loro risoluzione da parte di una macchina.
Una serie di compiti quali possono essere la localizzazione di un oggetto in una scena o l'output di una certa risposta basato su un ragionamento percettivo sono esempi di come un' azione risulti estremamente semplice per una persona, ma infinitamente complicata da eseguire per vie convenzionali all'interno di un computer.

In questa sezione verrà mostrato il funzionamento del sistema nervoso umano, descrivendo poi come esso abbia ispirato le concezioni di rete neurale artificiale e dei primi modelli computazionali; verranno inoltre illustrati i paradigmi di apprendimento e gli algoritmi che rendono possibili queste tecnologie.

1.1 Il sistema nervoso

L'uomo è una macchina eccellente per risolvere questo tipo di problemi. Il cervello umano è in grado di elaborare informazioni attraverso un complesso sistema neurobiologico, il cui elemento fondamentale è detto *Neurone*.
I processi elettrochimici che hanno sede all'interno di questa cellula, processano gli input degli altri neuroni e li modulano per poi inviare questi nuovi segnali alle altre parti del cervello.

I neuroni sono composti da un corpo principale detto *soma* e da due diramazioni: i *dendriti* e l'*assone* (anche detto *cilindrasse*).

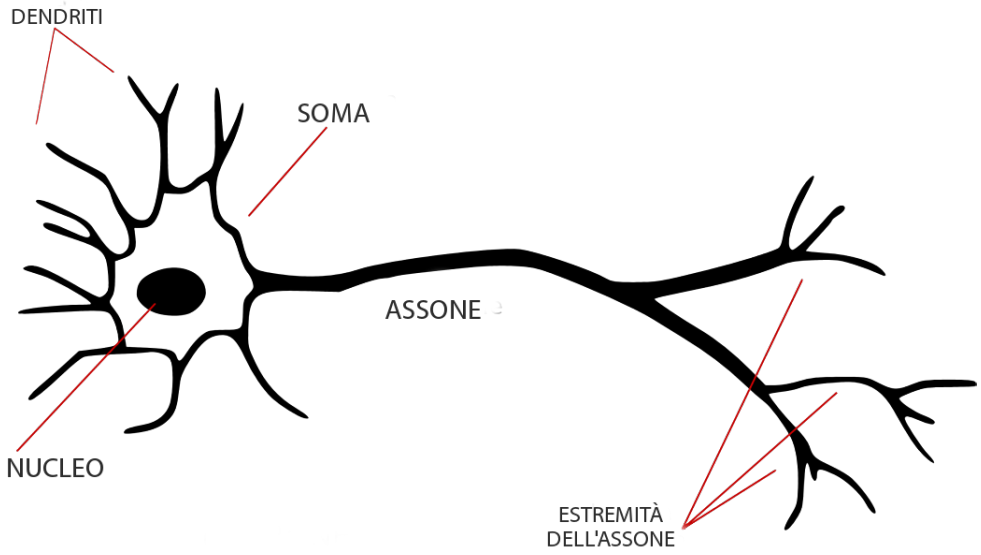


Figura 1.1: Struttura cellulare di un neurone con soma, dendriti e assone

Nel cervello umano sono presenti un numero di neuroni che si aggira intorno ai 100 miliardi, ciascuno dei quali è interconnesso ad un totale di circa 10^4 neuroni. Nelle interconnessioni avviene la *sinapsi*, un processo che serve per rinforzare o indebolire l'interazione neurale.

Gli impulsi elettrici all'interno del cervello viaggiano in sequenze molto brevi (tipicamente millisecondi) e con una frequenza che si aggira intorno ai 100Hz.
Le capacità cognitive di un individuo sono attribuite alla relazione e all' elaborazione dei segnali all'interno dei neuroni, ciò che definisce la funzionalità di una precisa parte del cervello è la struttura di interconnessione cerebrale presente in quella specifica regione.
Tutto ciò unito al fatto che nell'intero cervello sono presenti un numero approssimativo di 10^{15} interconnessioni, dona al cervello umano un'incredibile capacità di elaborazione dei segnali.

Il progresso nello studio del cervello ha spinto alcuni ricercatori a chiedersi se si potesse riprodurre lo stesso comportamento all'interno di un calcolatore.
Il primo passo verso una rete neurale artificiale è stato compiuto da *McCulloch & Pitts* con il loro primo importante studio redatto nel 1943 [7].

1.2 Il Percettrone

Un primo modello computazionale di neurone artificiale nasce quanto *F. Rosenblatt* fornisce il concetto di *Perceptron* o *Percettrone* [8].

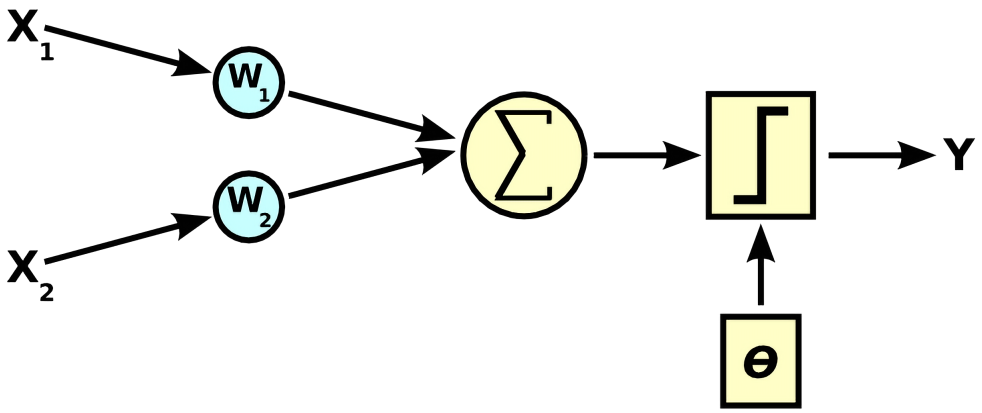


Figura 1.2: Schema computazionale di un Percettrone, come descritto da F. Rosenblatt [8]

Il Percettrone è un tipo di classificatore binario che correla un insieme di input, ad un output scalare y (di tipo reale) che viene calcolato come:

$$y = \theta \left(\sum_{j=1}^{n} w_j x_j - b \right)$$

Dove $\sum_{j=1}^{n} w_j x_j - b$ è la somma pesata delle corrispondenti x_j del vettore degli input, ciascuno pesato col il peso w_j e dove θ rappresenta una funzione chiamata *Funzione di Attivazione*.
Il valore b, detto *bias*, permette di 'spostare' la funzione di attivazione a destra o sinistra, il che può risultare critico per l'apprendimento (a seconda di che funzione di attivazione si usi per i neuroni).

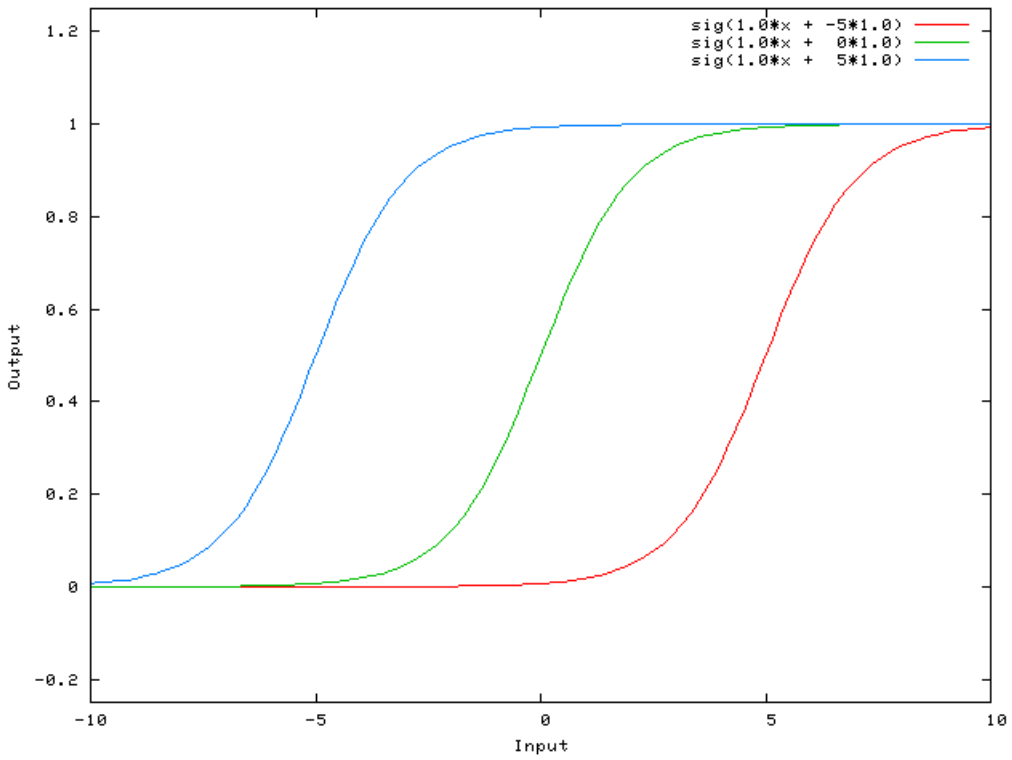


Figura 1.3: Funzionalità del valore bias, visualizzazione dello spostamento della funzione di attivazione

1.3 Concetto di ANN

L'architettura di una rete neurale artificiale basilare è derivata da quella del Percettrone. Può essere vista, infatti, come un grafo direzionale pesato in cui i nodi sono rappresentati dai neuroni, gli archi direzionali pesati sono le connessioni tra i neuroni.
Ci sono due categorie principali per suddividere le ANN, esse sono:

- Reti *Feed-Forward*: è la più classica rete neurale, il grafo che la rappresenta non ha cicli.
- Reti *Ricorrenti (RNN)*: sono reti neurali in cui il grafo che le rappresenta contiene dei cicli a causa di collegamenti tra i neuroni detti *feedback loops*, sono spesso usate per predire sequenze di dati temporali. Una variante di queste, chiamate reti *LSTM*, è utilizzata in questa tesi per l'Autopilot del veicolo.

Possiamo rappresentare i duei tipi di reti in questo modo:

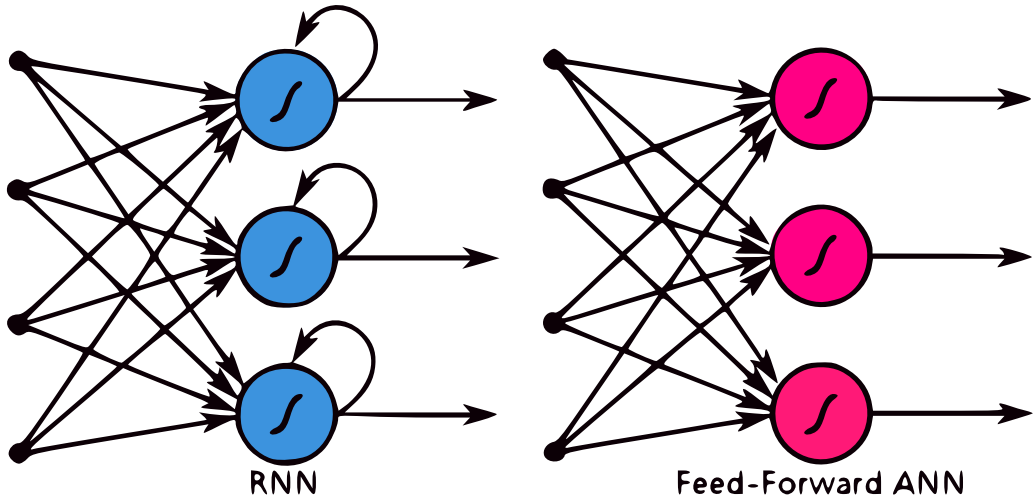


Figura 1.4: Schema semplificato di una RNN e Feed-Forward NN a confronto

1.4 Paradigmi di Apprendimento

1.4.1 Supervised

La gran parte dei problemi risolvibili con il Machine Learning sono di tipo supervisionato. Questi sono problemi in cui si dispone di un *Dataset* i cui output sono noti e quindi si dispone delle necessarie coppie di dati X e Y.
L'obiettivo quindi è quello di approssimare la funzione che mappa gli input X agli output Y abbastanza bene in modo che alla vista di un nuovo *Sample*, la rete sia in grado di predire un output Y significativo.
I problemi di tipo supervisionato sono divisibili in due categorie:

- **Classification**: Predizione di un output di tipo categorico, per esempio classificazione di un immagine come contenente *'gatto'* o *'cane'*.
- **Regression**: Un problema è detto di regressione quando il suo output è un valore di tipo reale.

1.4.2 Unsupervised

L'apprendimento si dice non supervisionato se si dispone dei soli input X senza corrispondenti variabili di output. L'obiettivo in questi casi è quello di capire la

struttura dei dati, indipendentemente dalla conoscenza delle variabili di output.
I problemi di tipo non supervisionato possono essere classificati in due aree:

- **Clustering**: Problemi in cui si vuole scoprire come formare dei gruppi a seconda dei dati di input.
- **Association**: Un problema di associazione si ha quando si vuole scoprire una relazione che descrive grandi porzioni di dati, come per esempio: *'Le persone che guardano questi programmi TV spesso guardano anche questi altri programmi'.*

1.4.3 Semi-Supervised

Questi problemi sono un subset dei due precedenti, possono essere risolti utilizzando le tecniche per *Supervised* e *Unsupervised* learning in congiunzione, in modo da ottenere più informazioni sui dati.

1.5 Apprendimento

L'apprendimento ha l'obiettivo di calibrare i pesi della rete in modo da minimizzare la funzione di perdita (*Loss Function*).
Questa funzione determina di quanto i valori predetti dal sistema si discostano dai valori effettivi spesso chiamati anche *Targets* o *Ground Truth*. La funzione di perdita è un buon indicatore della qualità di una ANN.

Una tipica funzione di perdita è l'errore quadratico medio *MSE* la cui formula è così rappresentata:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

In cui Y_i rappresenta un vettore di n targets e $\hat{Y}_i$ un vettore di valori predetti. Prendiamo ora come loss-function la precedente funzione, chiamandola $L(\omega)$:

$$L(\omega) = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

Come visto in precedenza per il percettrone (figura [1.2]), i valori predetti si ottengono tramite la seguente formula:

$$y = \theta \left(\sum_{j=1}^{n} w_j x_j - b \right)$$

Il processo di inferenza dei risultati è detto *Forwardpropagation* e produce in output i valori stimati, sostituendo nella loss-function al posto di $\hat{Y}_i$ il passo di forwardpropagation, otteniamo:

$$L(\omega) = \frac{1}{n} \sum_{i=1}^{n} \left(Y_i - \theta \left(\sum_{j=1}^{n} w_j x_j - b \right) \right)^2$$

L'obiettivo è minimizzare la funzione di perdita, cioè nell'individuare i valori di ω che la rendono minima, cioè quali pesi associare agli input affinché ciò avvenga. Uno dei metodi di ottimizzazione più rilevanti è il metodo *Gradient Descent*.

1.5.1 Gradient Descent

Grandient Descent è un algoritmo di ottimizzazione elaborato dal matematico francese Augustin-Louis Cauchy nel 1847. Il metodo della discesa del gradiente è una tecnica che serve per determinare i punti di massimo e minimo di una funzione in più variabili, quindi è ideale per l'ottimizzazione di reti neurali complesse.
Il suo funzionamento richiede di calcolare un vettore di derivate parziali, chiamato *Gradiente*; esso indica sempre la direzione in cui la funzione cresce maggiormente e perciò, permette di sapere in che direzione 'muoversi' sulla funzione per minimizzarla.

Per visualizzare questo concetto applicato alle reti neurali si può immaginare un sistema con soli due pesi, il cui valore formi gli assi x ed y su un piano in un sistema di riferimento tridimensionale. Data questa visualizzazione, il valore della funzione di perdita rappresenta l'elevazione nell'asse z del piano.
Da questo esempio si può capire che con il termine *'discesa'* si indica il movimento in direzione opposta al gradiente, effettivamente spostandosi verso un valore z della funzione di perdita sempre più basso.

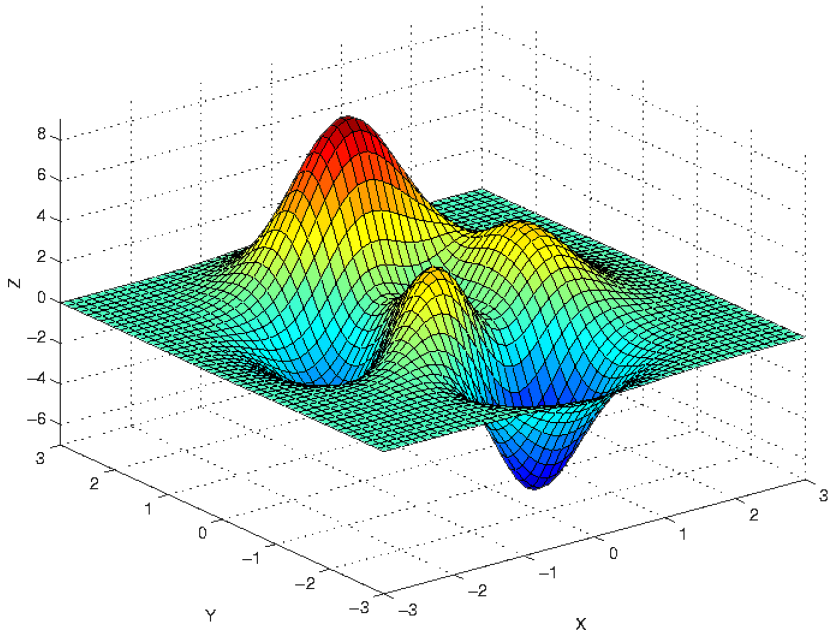


Figura 1.5: Rappresentazione grafica della relazione tra il valore di due pesi (Assi x ed y) e della funzione di perdita (Asse z)

1.5.2 Backpropagation

L'algoritmo di *Backpropagation* è un sistema di ottimizzazione delle reti neurali introdotto negli anni '80 e basato sulla tecnica di *Grandient Descent*.
Questo algoritmo è fondamentalmente suddiviso in due parti, la prima è la *Forwardpropagation*, ovvero il processo con cui la rete genera un risultato mentre la seconda è l'aggiornamento dei pesi in modo da portare la rete ad approssimare sempre più accuratamente la funzione desiderata.
L'aggiornamento dei pesi può essere visualizzato matematicamente come segue:

$$W^j = W^{j-1} - \alpha \frac{d}{dw} L\left(w\right)$$

Cioè i pesi al passo j si ottengono aggiornando i pesi al passo precedente nella direzione dell'antigradiente della funzione $L(\omega)$ da minimizzare.

Nella formula, il valore α è detto *Learning Rate*, esso regola la velocità di training dell'algoritmo per limitare il fenomeno dell'*Overshooting*, ovvero una situazione in cui un apprendimento troppo rapido, risulta in un 'salto' del punto di minimo locale ottimo. Questo valore va accuratamente scelto in modo tale da garantire un apprendimento veloce ma preciso.

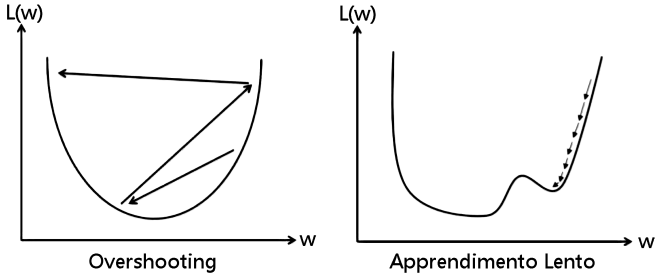


Figura 1.6: Overshooting con LR troppo alto, apprendimento troppo lento con LR basso

Il passo iterativo che compone l'algoritmo di backpropagation, permette di calcolare il gradiente rispetto ai pesi della rete mediante la regola delle derivate a catena, e rappresenta uno degli algoritmi più importanti nell'ambito del machine learning.

1.5.3 Funzioni di Attivazione e Vanishing Gradient

L'output del neurone artificiale è modulato da una specifica funzione di attivazione (θ in Figura *[1.2]*) che simula il funzionamento del *soma* in un neurone biologico. Contrariamente al passato, le tecniche moderne per l'apprendimento non fanno più uso di soglie fisse per l'attivazione del neurone (*Binary Step*) e si usano attualmente funzioni matematiche come la $ReLU$ che è attualmente una delle più usate.

Alcune delle funzioni di attivazione principali sono:

- *Sigmoid Function (Logistic o SoftStep):*

$$f(x) = \frac{1}{1 + e^{-x}} \tag{1.1}$$

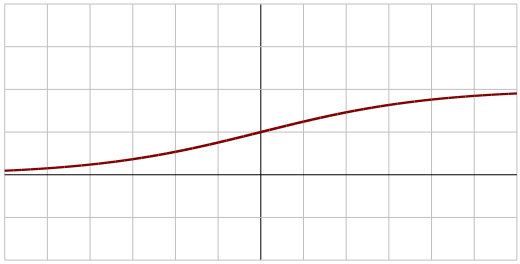


Figura 1.7: Grafico della funzione Sigmoide

- *Hyperbolic Tangent Function (TanH):*

$$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \tag{1.2}$$

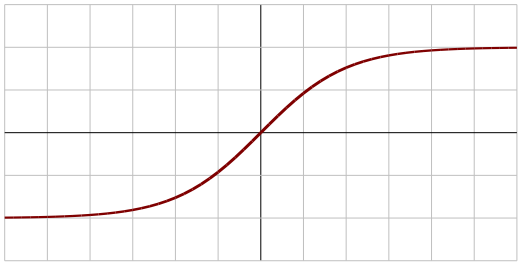


Figura 1.8: Grafico della funzione TanH

Uno dei problemi di queste funzioni di attivazione è il problema del *Vanishing Gradient* ovvero la diminuzione del 'potere di apprendimento' quando ci si avvicina agli estremi della funzione di attivazione.
Quello che avviene è che, a causa della forma della funzione di attivazione, quando il valore cresce (o diminuisce), il gradiente corrispondente si avvicina a zero e quindi si può riscontrare un'effettiva vanificazione dell'apprendimento della rete neurale.

Un ulteriore problema che si può riscontrare utilizzando queste attivazioni è quello della velocità di apprendimento (In parte legato al problema del *Vanishing Gradient*).
Per risolvere in parte questo problema sono state introdotte nuove funzioni di attivazione che tengono conto di questo fattore, tra cui la funzione *ELU* che viene usata in questa tesi per la creazione del sistema di Autopilot all'interno del simulatore.

- *Rectified Linear Unit (ReLU):*

$$f(x) = max(0, x) \tag{1.3}$$

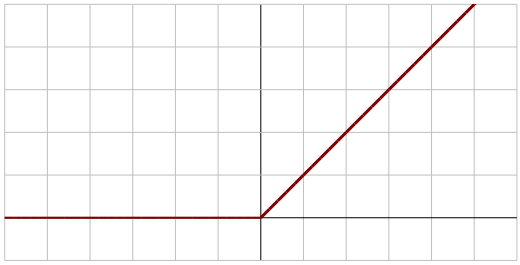


Figura 1.9: Grafico della funzione ReLU

- *Exponential Linear Unit (ELU):*

$$f(x) = \begin{cases} x & \text{if x} \geq 0 \\ \alpha\left(e^x - 1\right) & \text{if x} < 0 \end{cases} \tag{1.4}$$

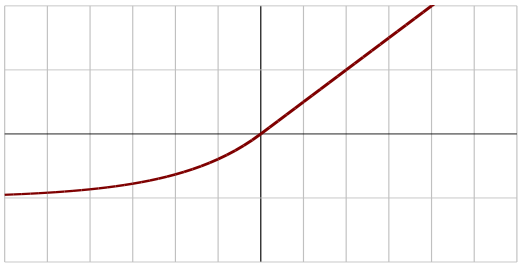


Figura 1.10: Grafico della funzione ELU

1.5.3.1 Vantaggi della funzione ELU

Il problema della funzione $ReLU$ è che il suo valore medio non è zero.
Se il valore medio di una funzione di attivazione è zero, si ottiene un apprendimento più rapido, ma quindi una delle domande che sorge spontanea è perché non si utilizzi una funzione di attivazione *Lineare*, la cui espressione banalmente è $f(x) = x$ ed il cui valore medio è appunto zero.
Il motivo è che se si usasse una funzione di questo tipo, l'intera rete neurale diventerebbe un approssimatore di funzione lineare, il che sarebbe come avere una rete contenente un singolo layer di neuroni.
Questo avviene poiché una funzione lineare non permette alla rete di imparare output non lineari e quindi di fatto vanifica il senso di avere più strati di neuroni.

Quello che la funzione ELU cerca di fare è quello di portare il valore medio vicino a zero e quindi effettivamente velocizzarne e favorirne la convergenza.

Capitolo 2

CNNs

Questo capitolo descrive la storia delle prime reti convoluzionali ed il come esse riescano a carpire in modo così eccelso i dettagli visivi di un'immagine. Sarà approfondito l'aspetto di input dei dati all'interno di una rete convoluzionale ed il come le immagini sono rappresentate all'interno di un calcolatore, per poi scendere nel dettaglio di come le CNN possano essere strutturate ed individuare caratteristiche nelle immagini.

2.1 Storia delle Convolutional Neural Networks

Le reti neurali convoluzionali sono state per prime introdotte da LeCun nel 1998 [6] e rappresentano una delle tecnologie più influenti nella *computer-vision*.
A partire dal 2012 le CNN hanno subito forte attenzione, in gran parte generata dalla vincita di *Alex Krizhevsky* all'annuale competizione ImageNet, nella quale l'uso di queste reti è riuscito per la prima volta ad abbattere l'errore di classificazione dal 26% al 15%, il che ha avuto un impatto decisivo sull'evoluzione e l'uso di questa tecnologia.

2.1.1 Input di una CNN

Un calcolatore non visualizza le immagini come un essere vivente, per questo è necessario specificare il tipo di input adatto per la rete.
Una foto è memorizzata come una serie di matrici, ognuna avente una dimensione M x N rappresentante larghezza ed altezza; il numero di matrici necessarie a rappresentare una determinata immagine è definita come la profondità o *depth* della

stessa e corrisponde al numero di canali colore che essa possiede. Per esempio, un'immagine *RGB* contiene *3* canali, uno per rosso, verde e blu, mentre se si ha una foto in scala di grigi si necessita di un solo canale per la rappresentazione del file.
L'insieme di matrici è anche chiamato *Tensore*, quello corrispondente ad un'immagine RGB si presenta in questo modo:

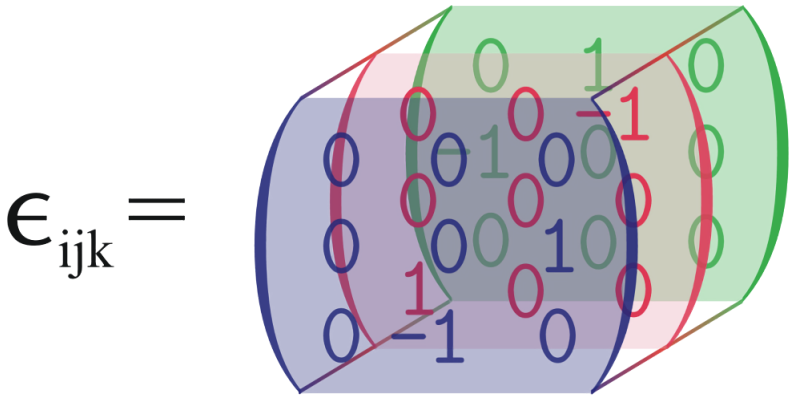


Figura 2.1: Rappresentazione di un Tensore

2.1.1.1 Pixel Values

Il contenuto delle matrici è invece un insieme di valori (Tipicamente da 0 a 255) che rappresentano per ogni pixel la predominanza cromatica di quel particolare canale.
All'interno di una rete neurale è preferibile mantenere l'intervallo di valori ridotto, quindi piuttosto che utilizzare l'immagine in forma originale si può applicare un procedimento detto *Normalizzazione* che consiste nel dividere ogni pixel dell'immagine per un valore, riducendone così il range. Per ottenere un intervallo di valori che vada da 0 ad 1 si può dividere il tensore per 255.

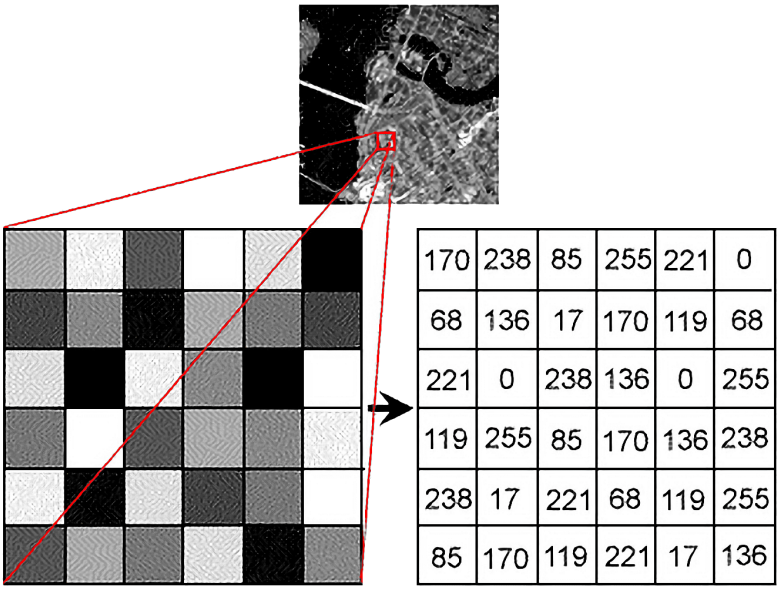


Figura 2.2: Immagine a scala di grigi rappresentata da una matrice con i corrispondenti pixel values

2.1.2 Struttura di una CNN

Il primo strato di una rete CNN è costituito da un *Layer Convoluzionale* il cui input è l'immagine da processare.
Per attuare il processo di *Convoluzione* si fa scorrere un certo *Filtro* (Chiamato anche *Kernel*) sul tensore. Un filtro è un insieme di pesi rappresentati sotto forma di un tensore di dimensione m x n e la cui profondità è la medesima dell'immagine di input. Quando questo filtro viene fatto scorrere sull'immagine, la regione su cui si trova in quel momento è detta *Campo Recettivo*.
Durante questo processo, i valori del filtro vengono moltiplicati ai pixel values originali, ed infine sommati in un singolo valore scalare che rappresenta l'output di quel passo di convoluzione.

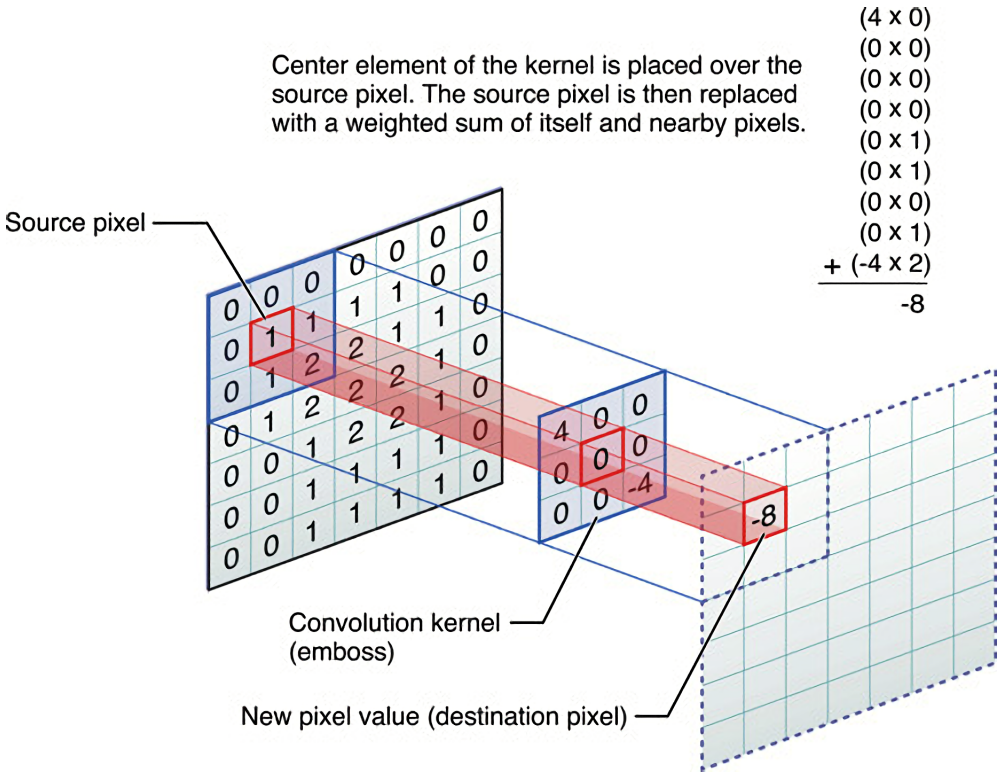


Figura 2.3: Processo di convoluzione su una matrice

Questo processo viene ripetuto fino ad arrivare alla fine dell'immagine. Il tensore risultate è la rappresentazione di tutti gli output del processo di convoluzione e si chiama *Activation Map* (Mappa di attivazione) o *Feature Map* (Mappa delle caratteristiche).

Figura 2.4: Rappresentazione di un filtro per il riconoscimento di una feature

La convoluzione su un immagine f rappresentata da un tensore m x n con filtro k è rappresentabile matematicamente come:

$$(f * k)[m \times n] = \sum_i \sum_j f(m-i, n-j) \cdot K(i,j)$$

2.1.3 Individuazione delle Features

All'interno di una rete neurale, il processo di convoluzione svolge il ruolo di identificare una certa caratteristica di ciò che si sta osservando, come per esempio: curve, linee o angoli. Strati convoluzionali preliminari tendono a codificare features molto semplici, mentre in profondità la rete carpisce caratteristiche più complesse come volti interi ed oggetti.

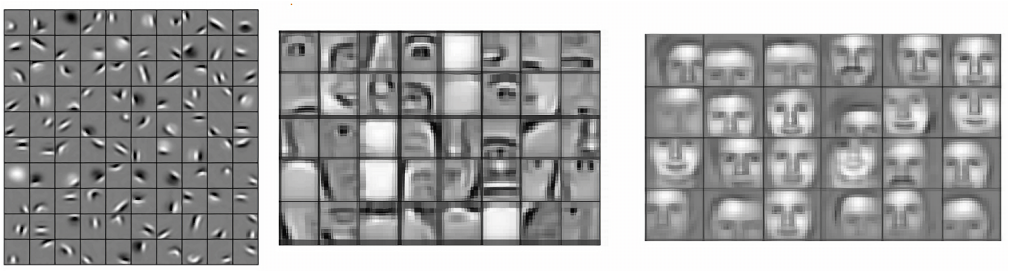


Figura 2.5: Gerarchia di apprendimento delle caratteristiche nei filtri

2.1.4 Pooling Layers

Dopo ogni layer convoluzionale si può applicare un'operazione chiamata *Pooling*. Ciò consiste nel creare una versione ridimensionata dell'output dello strato precedente in modo da ridurre le dimensioni del risultato ed ottenere una rappresentazione più compatta delle feature iniziali.
Questa operazione, anche detta *sub-sampling* o *down-sampling* (Sotto campionamento) si può effettuare secondo varie regole, alcune di queste sono:

- **Max Pooling**: Si applica un filtro *Max* alla rappresentazione iniziale. Ciò condensa le feature in un subset rappresentante le caratteristiche predominanti dell'input.
- **Average Pooling**: Viene applicato un filtro *Avg* che restituisce in output una rappresentazione media delle feature iniziali.

2.1.5 Flatten e Dense Layer

Alla fine di una rete convoluzionale è necessario ridurre la dimensionalità dell'output degli strati convoluzionali in modo da avere in uscita un vettore. Ciò si ottiene applicando in uscita un'operazione chiamata *Flatten* che prende il tensore uscente da uno strato convoluzionale e lo ri-distribuisce come un vettore che può essere utilizzato come input all'ultima parte della CNN.
Il risultato finale viene usato come input per una rete neurale 'classica' in cui spesso vengono utilizzati strati completamente interconnessi di neuroni (chiamati *Dense Layers*) che creano la rappresentazione finale dell'input.

All'ultimo strato, a seconda del problema specifico, viene applicata una funzione di attivazione, che può essere anche lineare in caso di output reali, o probabilistica se si utilizzano funzioni di attivazione come la *SoftMax* nel caso di classificatori.

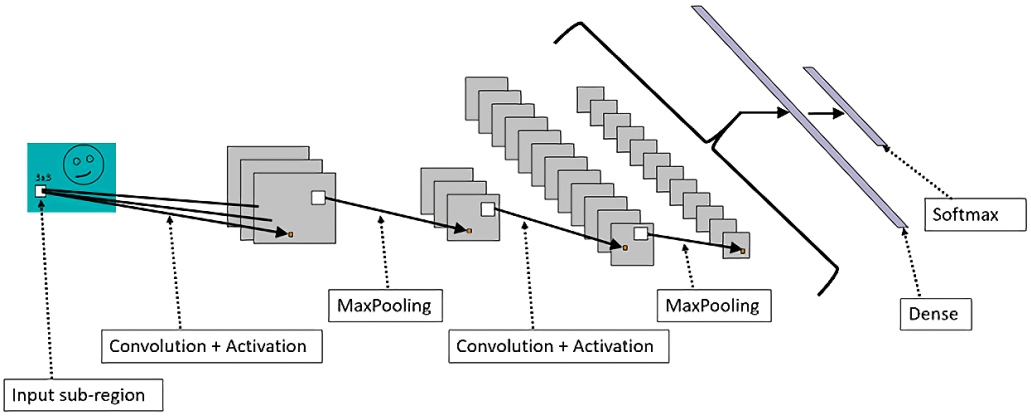


Figura 2.6: Intera struttura di una CNN

Capitolo 3

LSTMs

Questo capitolo descrive i concetti alla base delle reti ricorrenti di tipo LSTM. Vengono evidenziate le capacità che questo tipo di rete presenta nell'apprendimento di sequenze temporali e dipendenze a lungo termine e ne viene poi mostrata la struttura interna.

3.1 Reti neurali ricorrenti

Le Reti Neurali Ricorrenti (RNN) nascono con l'idea di mantenere una memoria degli eventi passati; questo comportamento non è possibile con le reti normali ed è per questo che le RNN sono utilizzate in ambiti in cui le reti classiche falliscono, come per esempio la predizione di serie temporali (meteo, quotazioni in borsa...) che fanno riferimento a dati precedenti.
L'elemento fondamentale di queste reti è il neurone ricorrente; la sua caratteristica è quella di essere connesso a se stesso tramite un collegamento detto *feedback loop*. Una RNN può essere visualizzata come una serie di reti identiche fra di loro ma interconnesse, questa rappresentazione è detta *srotolamento* della rete, e da vita ad una struttura a catena:

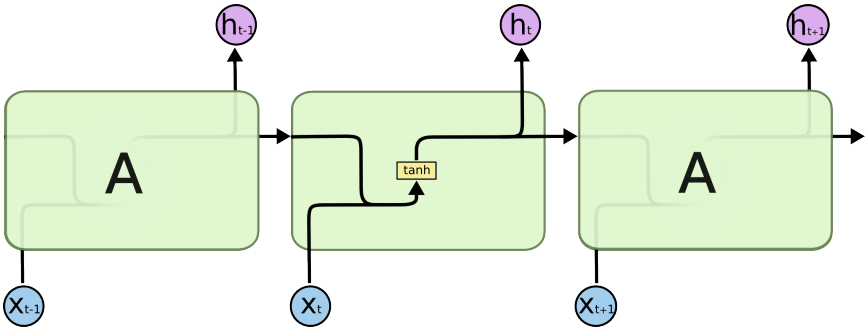


Figura 3.1: Struttura a catena di una RNN

3.2 Dipendenze a lungo termine

Le reti ricorrenti permettono di attuare predizioni che sono influenzate dai dati precedenti. Se il problema che stiamo analizzando dipende però da eventi molto distanti nel passato, le reti ricorrenti classiche non sempre hanno buone prestazioni. Un problema di questo tipo è l'utilizzo di una RNN per la generazione di una lunga sequenza di testo (come per esempio un intero articolo o storia); in questo caso le capacità di apprendimento della rete non sono sufficienti.
Nasce quindi la necessità di imparare dipendenze a lungo termine.

3.2.1 Reti Long Short-Term Memory

Le reti ricorrenti di tipo LSTM sono state introdotte per la prima volta da Hochreiter & Schmidhuber [4] nel 1997 per risolvere problemi di questo tipo. Le LSTM funzionano sorprendentemente bene su una grandissima quantità di problemi che non sono limitati alle sole dipendenze a lungo termine; esse infatti riducono anche drasticamente il problema del *Vanishing Gradient* e forniscono prestazioni superiori nell'apprendimento di sequenze temporali.

La differenza delle LSTM risiede nella struttura degli elementi che si ripetono al suo interno, esse sono costituite infatti da quattro moduli differenti che interagiscono fra di loro:

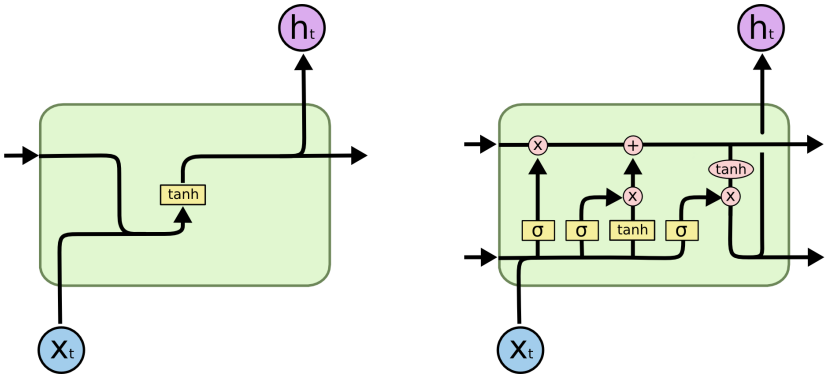


Figura 3.2: Confronto modulo RNN standard e modulo LSTM

I moduli LSTM sono in grado di regolare ciò che viene memorizzato e cancellato, questo è possibile grazie alla presenza di vari elementi detti gates composti da uno strato neurale a sigmoide ed un operatore di moltiplicazione pointwise.
L'output di ogni gate è nel range (0,1), rappresentante di fatto la percentuale di informazioni che fluiscono al suo interno.

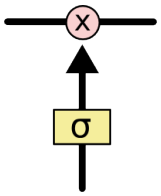


Figura 3.3: Gate generico di una LSTM

3.2.1.1 Forget Gate Layer

La prima parte del modulo LSTM decide quali informazioni sono eliminate dalla cella.
Come citato in precedenza, il gate prende gli input h_{t-1} ed x_t e restituisce in uscita un valore compreso tra 0 ed 1 per ogni stato della cella C_{t-1}. Chiamato f_t l'output del gate, possiamo visualizzare il suo significato come segue:

- $f_t = 0 \rightarrow$ Azzeramento completo dello stato della cella.
- $f_t = 1 \rightarrow$ Mantenimento totale del valore della cella.

La funzione che esprime l'output f_t del forget gate layer è rappresentata dalla seguente espressione matematica:

$$f_t = \sigma\left(W_5 \cdot [h_{t-1}, x_t] + b_f\right)$$

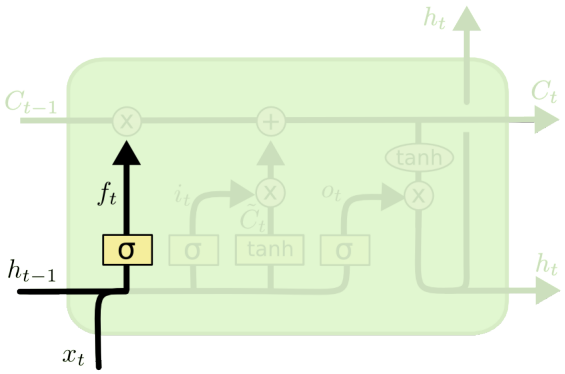


Figura 3.4: Forget Gate Layer

3.2.1.2 Memorizzazione dei dati

La memorizzazione dei dati è suddivisa in due parti. La prima è affidata ad uno strato sigmoide chiamato *Input Gate Layer*; esso effettua un'operazione che stabilisce quali valori dovranno essere aggiornati.
La seconda fase invece è affidata ad uno strato *tanh* che crea un vettore di valori *'candidati'* ad essere aggiornati, dato che il vettore dei valori delle celle è detto C_t, quello dei valori candidati è definito come $\tilde{C}_t$. Per creare un insieme aggiornato dei valori si combinano gli output dei due layer.

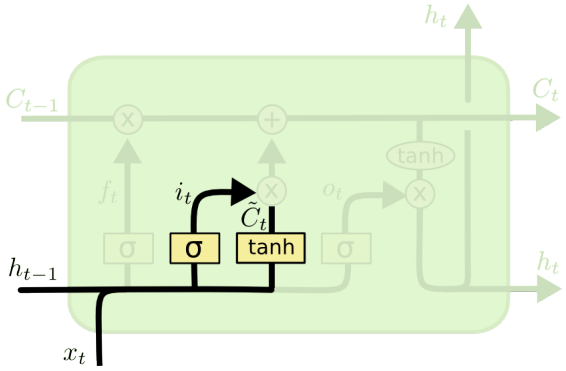


Figura 3.5: Input e Tanh layers

Le funzioni che regolano le uscite dei sui layer sono:

$$i_t = \sigma\left(w_i \cdot [h_{t-1}, x_t] + b_i\right)$$
$$\tilde{C}_t = \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right)$$

3.2.1.3 Aggiornamento dei dati

L'aggiornamento dello stato della cella da C_{t-1} a C_t viene effettuato moltiplicando il vecchio stato per f_t e sommando $i_t * \tilde{C}_t$, ottenendo così i nuovi valori della cella.

$$\tilde{C}_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

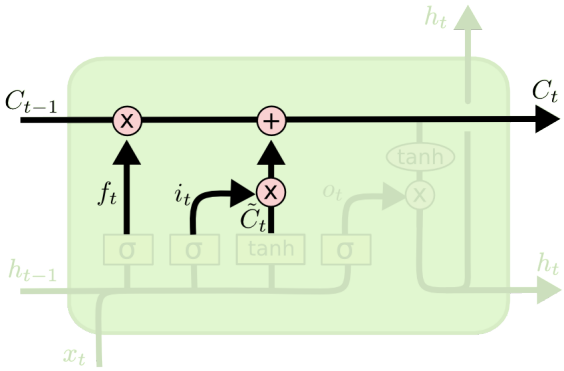


Figura 3.6: Parte del gate che svolge la funzione di aggiornamento

3.2.1.4 Output della cella

Rimane ora da stabilire quali informazioni vengono inviate in output. Il risultato sarà dato da uno strato sigmoide che decide quali parti della cella contribuiranno all'output e dallo stato corrente della cella, filtrato tramite una funzione *tanh* per ottenere un range da -1 a 1.
Il risultato di questa operazione è moltiplicato per il valore del layer sigmoide in modo da ottenere in uscita solo gli output desiderati:

$$o_t = \sigma\left(W_0\left[h_{t-1}, x_t\right] + b_0\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

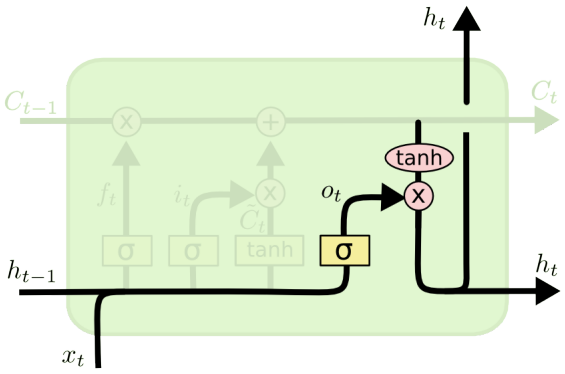


Figura 3.7: Parte del modulo LSTM che svolge l'output

Capitolo 4

Keras e Python

La scelta delle tecnologie con cui si sviluppa un sistema è fondamentale per determinarne una buona riuscita; in questo capitolo vengono esposte le funzionalità del framework Keras in congiunzione al linguaggio Python per la creazione di modelli di Machine Learning.
Vengono poi mostrate le caratteristiche principali di Keras come i vari layer disponibili, la creazione di modelli tramite metodologia ad oggetti e le funzioni matematiche che il framework mette a disposizione. Viene poi esposta la metodologia con cui le reti neurali possono essere allenate tramite questa libreria, per poi ottenere una predizione finale.

4.1 Il framework Keras e Python

Keras rappresenta uno dei più pratici framework di ricerca sul machine learning, ed è basato su due back-end: *Tensorflow* e *Theano*. Entrambe le librerie base, come anche Keras stesso, sono disponibili in Python, linguaggio che offre una ampissima scelta di pacchetti per lo sviluppo di applicazioni ed è il più diffuso nell'ambito

della ricerca sull'intelligenza artificiale e sulla data science.
Il back-end Theano sta lentamente cadendo in disuso data la cessazione del suo sviluppo anche a causa dell'espansione della libreria Tensorflow (Creata da Google) che è in costante sviluppo ed ormai definisce lo standard de facto dell'industria.
Ciò che Keras permette di ottenere, è un utilizzo semplificato del suo back-end e fornisce uno strumento essenziale nella prototipazione e nello studio del machine learning, dando la possibilità di ottenere veloci iterazioni del codice e dei modelli neurali con uno sforzo minimizzato rispetto all'utilizzo diretto delle librerie base (come Tensorflow).

4.1.1 Tipologie base di layers disponibili in Keras

Il framework mette a disposizione un insieme di layer utilizzabili per prototipare reti neurali che spaziano da comuni strati interconnessi fino a quelli ricorrenti e convoluzionali.

Alcuni dei principali sono riportati qui di seguito (nel caso di layer ad input multidimensionale, si inserisce la variante 2D, dato l'utilizzo di immagini in questa tesi):

- **Sequential**: Permette di definire la struttura di un modello come sequenza di layer.
- **Dense**: La base delle reti neurali, uno strato di neuroni interconnessi fra di loro.
- **Activation**: Una parte fondamentale delle ANN, permette di applicare una funzione di attivazione allo strato precedente.
- **Flatten**: Cambia la dimensionalità dell'input passando per esempio da un dato multidimensionale ad uno monodimensionale.
- **Dropout**: Il *Dropout* è un'operazione molto importante per evitare il fenomeno dell'*Overfitting*, ovvero la tendenza della rete a ripetere il training set piuttosto che generalizzare. Funziona disattivando casualmente un numero di neuroni nell'input. Ciò avviene ad ogni ciclo di training e la quantità di dropout è regolata da una percentuale chiamata *Rate*.
- **Conv2D**: La base di una rete convoluzionale 2D.
- **MaxPooling2D**: Strato che effettua un'operazione di pooling con filtro *Max*.
- **AveragePooling2D**: Pooling applicato con filtro *Avg* (Media).

- **TimeDistributed**: Suddivide un input non-sequenziale in step temporali. Utile per adattare un certo strato ad essere utilizzato come input ad una successiva rete ricorrente.
- **LSTM**: Strato che rappresenta un insieme di neuroni ricorrenti di tipo Long Short-Term Memory. Riceve in input una sequenza temporale di dati.

4.1.2 Creazione di un modello in Keras

Il framework è strutturato secondo la metodologia di programmazione ad oggetti, grazie a ciò la creazione di un modello è molto semplice: si seleziona l'architettura di base e poi si aggiungono i layer necessari a creare il modello desiderato.

4.1.2.1 Esempio di modello neurale in Keras

Si riporta un esempio di modello creato utilizzando il framework Keras:

```
1  from keras.models import Sequential # Basic class for sequential models
2  from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPool2D
3
4  SIZE_Y = 32
5  SIZE_X = 32
6
7  model = Sequential()
8  model.add(Conv2D(64, kernel_size=(2, 2), activation='relu',
    ↪  input_shape=(SIZE_Y, SIZE_X, 1)))
9  model.add(MaxPool2D(2,2))
10 model.add(Dropout(0.2))
11 model.add(Flatten())
12 model.add(Dense(32, activation='relu'))
13 model.add(Dropout(0.1))
14 model.add(Dense(2, activation='softmax'))
15 model.summary()
```

4.1.2.2 Descrizione del modello e dell'output

Il codice precedente crea la struttura di un semplice classificatore. È composto principalmente da un layer convoluzionale di 64 neuroni e da una serie di strati interconnessi finali, in cui l'attivazione dell'ultimo strato è di tipo *SoftMax* e quindi restituisce un vettore di probabilità con codifica *One-Hot* (1 in corrispondenza di una certa classe e 0 altrimenti).
In questo caso lo strato di output ha 2 neuroni, ciò significa che il vettore risultante è composto da due elementi, i quali avranno un valore indicante la probabilità che l'input appartenga ad una delle due classi. Ciò significa che se l'output è per esempio [0.9, 0.1], allora l'input è al 90% appartenente alla prima classe.

Un modo efficace per visualizzare la struttura di un modello è la funzione *summary* utilizzata nell'ultima parte dell'esempio.Questo codice stampa nella console la composizione del modello in tutti i suoi strati, specificandone anche il numero di parametri:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 31, 31, 64)        320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2)(None, 15, 15, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 15, 15, 64)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 14400)             0
_________________________________________________________________
dense_1 (Dense)              (None, 32)                460832
_________________________________________________________________
dropout_2 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 66
=================================================================
Total params: 461,218
Trainable params: 461,218
Non-trainable params: 0
_________________________________________________________________
```

4.1.3 Hyper-parametri

Gli *Hyper-parametri* sono tutti quei valori che costituiscono le caratteristiche del nostro modello, come: numero di neuroni, percentuali di dropout o dimensioni di pooling.
Avere *Hyper-parametri* giusti può fare un'enorme differenza nelle performance di un modello e ciò, unito al fatto che non ci sono modi prestabiliti per la loro decisione, fa si che la scelta di questi dati sia una delle parti più difficoltose nella creazione di un buon modello neurale.
Inoltre, la scelta di funzioni di loss e di ottimizzatori efficaci è fondamentale per il training ottimale di una rete.

4.1.4 Funzioni di loss, ottimizzatori e compilazione

Keras mette a disposizione molte funzioni già predisposte (come per il caso delle funzioni di attivazione) ed inoltre fornisce la possibilità di estenderne il funzionamento o definirne di proprie.
Nel caso delle *loss-functions*, Keras predispone le più utilizzate, tra le quali troviamo:

- `mean_squared_error`
- `mean_absolute_error`
- `mean_absolute_logarithmic_error`
- `categorical_crossentropy`
- `binary_crossentropy`

Per quanto riguarda gli ottimizzatori è sempre disponibile la possibilità di estenderne il funzionamento o di implementarne di nuovi. Quelli predefiniti sono:

- `SGD` : *Stochastic Gradient Descent*, il più classico ottimizzatore basato su gradient descent.
- `RMSprop` : Consigliato per reti di tipo ricorrente.
- `Adam` : Uno dei migliori ottimizzatori, è seguito da molte sue varianti.
- `Adagrad`
- `Adadelta`

- `Adamax`
- `NAdam`

Queste opzioni devono essere specificate durante la fase di compilazione:

```
# Build a custom RMSprop optimizer with lower Learning Rate
optimizerRMSPROP = RMSprop(lr=0.0005, rho=0.9, epsilon=None, decay=0.0)

# Use LOSS,OPTIMIZER and keep track of VALIDATION LOSS during training
model.compile(loss='mean_squared_error', optimizer=optimizerRMSPROP,
 ↪  metrics=['val_loss'])
```

4.1.5 Parametri di training

Una volta ultimata la compilazione del modello, è necessario procedere al suo training. Per fare ciò bisogna disporre di un array Numpy contenente i valori X *(Input)* ed Y *(Target)*.
La funzione che si occupa del training è *fit()* e deve essere chiamata sul modello interessato, specificando poi i parametri di addestramento appropriati.
Alcuni di questi sono:

- `epochs`: Il numero di 'epoche' in cui il training sarà suddiviso.
- `batch_size`: Imposta la dimensione del 'blocco' di training, ad ogni iterazione verranno utilizzati n elementi per addestrare la rete.
- `validation_split`: Questo parametro è molto utile in quanto permette di definire una percentuale del dataset di training su cui, invece dell'addestramento, verrà effettuato un checkup (detto validazione) della rete. Ciò permette di misurarne le capacità di generalizzazione.
- `shuffle`: Parametro che permette di randomizzare gli elementi del dataset.
- `verbose`: Stampa a video i progressi con maggior dettaglio.

4.1.6 Callback Functions

I callbacks sono utili funzioni messe a disposizione da Keras che servono per attuare delle operazioni durante la fase di training.
Le più importanti sono:

- `EarlyStopping`: L' early stop monitor è una funzione che permette di fermare il training dopo n epochs senza miglioramenti. Per specificare il valore n si può utilizzare un parametro chiamato *patience*.
- `ModelCheckpoint`: La funzione checkpoint è molto utile per salvare i progressi della rete. Attraverso vari parametri è possibile specificare secondo quale valore determinare il salvataggio, se salvare solo il modello più performante ed il *Path* in cui effettuare il salvataggio.

4.1.7 Allenare la rete

Una volta specificati i parametri di training e create le appropriate funzioni di callback, si può procedere all'allenamento come segue:

```
from keras.callbacks import EarlyStopping, ModelCheckpoint

# Setup the early stop monitor
earlyStopMonitor = EarlyStopping(patience=5)

# Setup the checkpoint saver
checkpointSaver = ModelCheckpoint(os.path.join(SAVE_PATH, MODEL_NAME),
 ↪  monitor='val_loss', verbose=1, save_best_only=True,
 ↪  save_weights_only=False, mode='auto', period=1)

# Start the training
# Validation Split : 10%

modelTraining = model.fit(train_data, train_targets, batch_size=32, epochs=100,
 ↪  verbose=1, validation_split=0.1, callbacks=[earlyStopMonitor,
 ↪  checkpointSaver])
```

4.1.7.1 Plotting dei risultati di training

Per visualizzare l'andamento del training si può fare uso della libreria Python *matplotlib*, questa permette la realizzazione di grafici con cui è possibile rappresentare l'andamento di un certo valore durante l'allenamento.

```
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

# Take the validation loss from the model training history
plt.plot(modelTraining.history["val_loss"], 'tab:blue')
# Create the label for the graph
plt.xlabel('Epochs')
patch = mpatches.Patch(color='tab:blue', label='Loss')
plt.legend(handles=[patch])
# Show the plot
plt.show()
```

4.1.8 Caricamento del modello e predizione di un risultato

Una modello Keras può essere caricato da disco tramite la funzione *load_ model()*.

```
from keras.models import load_model
# Load the model from disk
model = load_model(os.path.join(SAVE_PATH, MODEL_NAME))
```

Caricato il modello da disco è possibile fare una predizione usando la rete neurale. Per fare ciò si utilizza la funzione *predict(Input)* che restituisce un vettore di risultati, la dimensionalità del risultato rispetta quella dei target utilizzati per il training.

```
# Predict some data using an image as input
# Specify verbose=0 to not print excessive console info
prediction = model.predict(image_to_predict, verbose=0)
print("Predicted results are: " + prediction)
```

Capitolo 5

Game-engine Unity

Unity è un game-engine sviluppato da Unity Technologies per la produzione di applicazioni e giochi 2D-3D. Dispone di un potente motore grafico e di un'interfaccia che permetto una veloce programmazione e prototipazione di software.
In questa parte verranno mostrate le caratteristiche salienti dell'editor, come lo store di assets, i componenti essenziali del motore di gioco e l'integrazione con C-Sharp per la programmazione delle applicazioni.

5.1 Interfaccia Utente

Il motore di gioco dispone di un editor con un'interfaccia intuitiva suddivisa in finestre liberamente spostabili. Le principali componenti dell'interfaccia sono:

- `Hierarchy`: Mostra la lista di tutti gli elementi nella scena di gioco; è possibile inoltre rinominare gli oggetti o creare gerarchie di GameObjects che agiranno come un singolo elemento per quanto riguarda le trasformazioni tridimensionali.

- `Inspector`: È una delle parti più importanti di Unity, l'inspector è una sezione dell'interfaccia che permette la visualizzazione di tutti i componenti presenti all'interno di un GameObject, permettendone un controllo completo dell'oggetto. Questa finestra inoltre espone le variabili pubbliche di tutti i componenti, abilitando lo sviluppatore al collegamento visuale (Drag & Drop) dei dati, sarà possibile perciò riempire qualsiasi campo tramite il trascinamento oppure semplicemente scrivendo il dato nell'inspector nel caso di variabili numeriche o stringhe.

- `Scene`: Questa finestra mostra la vista di debug della scena corrente, è possibile manipolare la posizione dei GameObjects in modo da ottenere la composizione voluta.

- `Game`: Serve per visualizzare effettivamente il gioco in funzione, questa mostra normalmente la visuale della telecamera principale di gioco e, se viene premuto il tasto play nell'editor, avvia l'esecuzione in-loco dell'applicazione. Durante l'esecuzione tutte le variabili dell'inspector sono aggiornate dinamicamente per riflettere quelle a runtime.

- `Project`: Mostra la gerarchia dei file di progetto come modelli, suoni ed immagini.

- `Console`: Espone una semplice console, molto utile per le funzionalità di debug.

5.2 Assetstore

Un elemento vincente di Unity è il suo store di assets, ovvero un negozio in cui è possibile acquistare strumenti ed oggetti di gioco fatti da altre persone, così come suoni, modelli 3D e tanto altro.
Tutto ciò permette uno sviluppo molto rapido, soprattutto nelle fasi iniziali in cui si costruisce un prototipo di quello che sarà poi il gioco finale.

5.3 Linguaggio di scripting

Il linguaggio di programmazione principale in Unity è *C-Sharp*, è possibile utilizzare l'IDE che più si preferisce per lo sviluppo, quello di default è *MonoDevelop* ma senza problemi l'engine è compatibile con *VisualStudio* di Microsoft, comprendendo debugging ed IntelliSense.

La programmazione in Unity è strutturata secondo file detti *Script*. Uno script, appena creato, viene predisposto con uno scheletro di base, che comprende l'importazione delle librerie del motore di gioco ed i suoi metodi standard.
Uno script vuoto si presenta così:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class NewScript : MonoBehaviour {
  void Start () {
    // This method is called at the creation of this game object
  }
  void Update () {
      // Update is called once per frame
  }
}
```

5.4 GameObjects

L'uso di Unity si basa sui GameObject, strutture fondamentali che rappresentano le entità presenti nella scena di gioco. Ogni GameObject ha delle proprietà intrinseche come per esempio il nome, ed altre che possono essere espanse a piacimento.

5.4.1 Transform e Components

Il principio base su cui si fonda il comportamento di un GameObject è la presenza di certi componenti che lo caratterizzano. La posizione nello spazio tridimensionale è gestita dal *Transform* che tiene memoria del vettore posizionale e rotazionale del GameObject.
Altri componenti essenziali sono:

- `RigidBody`: Gestisce la fisica di un oggetto in modo realistico.
- `Camera`: Rappresenta una telecamera tridimensionale che permette la visualizzazione del mondo di gioco, è possibile impostare il campo visivo (FOV) ed altri parametri come la modalità di proiezione *Prospettica* o *Ortogonale*.

- `Collider`: Permette di specificare dei bounding-box di collisione per un certo oggetto. Spesso questi coincidono con la forma del modello 3D assegnato al GameObject.
- `MeshRenderer`: Renderizza un modello 3D all'interno di un GameObject.
- `MonoBehaviour`: *MonoBehaviour* è la classe a cui appartengono gli script che possono essere connessi ad un oggetto. Ogni codice che estende questa classe può essere usato come componente di un GameObject per fare in modo che esso segua determinati comportamenti.
- `AudioSource`: Permette di abilitare un oggetto all'emissione di suoni. Come per gli altri componenti le sue proprietà sono gestibili via codice e sono supportati effetti avanzanti come la spazializzazione tridimensionale dell'audio.

Innumerevoli altri componenti sono disponibili per arricchire il comportamento degli elementi di gioco, insieme a strumenti avanzati per l'animazione, il mixing audio e la creazione di UI interattive per le applicazioni.

5.5 RenderTextures

All'interno del mondo di gioco è possibile far renderizzare una telecamera su una superficie diversa dallo schermo. Questa è una soluzione molto utile nei casi in cui si debbano acquisire immagini di gioco provenienti da telecamere diverse da quella principale o per salvare screenshots.
Per fare ciò si utilizzano le *RenderTexture*, ovvero elementi che rappresentano una texture renderizzata da una telecamera.
Per poter catturare un'immagine si crea un oggetto di tipo RenderTexture e lo si assegna come target di rendering alla telecamera interessata, successivamente basta forzarne l'aggiornamento tramite la funzione *Render()* ed ora si potranno estrarre i dati grezzi dall'oggetto RenderTexture. Questi dati rappresentano le informazioni dei pixel di un'immagine e possono essere manipolati a piacimento.

Capitolo 6

Design del progetto

La seguente sezione descrive in tutte le sue parti la creazione del progetto; in particolare viene prima illustrata la costruzione del Simulatore in Unity con tutte le sue parti costituenti: il tracciato di test, il veicolo con sistema di guida ed anche gli aspetti di comunicazione col back-end Python, come la registrazione delle immagini e la ricezione di predizioni.
In secondo luogo è mostrata la sperimentazione riguardante i modelli di rete neurale, vengono confrontate caratteristiche e prestazioni dei vari modelli, analizzandone la capacità di clonazione comportamentale.

6.1 Prerequisiti Progetto

Questo progetto è stato costruito in modo che sia facilmente esportabile in modalità standalone, ovvero senza la necessità di installare programmi o estensioni di terze parti per la sua esecuzione; tuttavia per la modifica e creazione del codice sorgente sono necessari alcuni tool e programmi.
Oltre al game-engine Unity per la creazione del simulatore, il back-end necessita di un ambiente Python e di alcune librerie e pacchetti utili alla gestione della rete neurale e del dataset.
In particolare in questa tesi sono state utilizzate le librerie: *Keras* con back-end *Tensorflow*, oltre a *Numpy Scipy* per la gestione delle operazioni matematiche e dei vettori.
Altri pacchetti di supporto sono *h5py* utile per il salvataggio di modelli su disco, *Pyinstaller* per la creazione di eseguibili da codice Python, *Matplotlib* per la visualizzazione di grafici insieme a *tqdm* per la realizzazione di progressbar interattive

durante il processing del dataset.
Una parte fondamentale dell'applicazione in Python è la libreria *OpenCV 2* che permette l'elaborazione e il caricamento di immagini in modo facile e rapido.

6.2 Simulatore

L'ambiente base che costituisce il simulatore è una scena contenente un veicolo ed un circuito di prova su cui la macchina può essere allenata e testata.

6.2.1 Tracciato di prova

La creazione del percorso rappresenta un punto fondamentale per l'apprendimento dell'intelligenza artificiale.
Dato che essa percepisce il mondo tramite input visivo, è necessario che la strada sia ben visibile da parte della telecamera di gioco; l'apprendimento infatti è condizionato dalla qualità con cui la rete neurale carpisce la forma del percorso e come una persona, migliore è la sua visibilità, maggiori saranno le prestazioni. Non è da sottovalutare inoltre la differenza di 'risoluzione' tra l'occhio umano e la telecamera virtuale.

La topologia del tracciato inoltre è un punto chiave per l'apprendimento, se il percorso fosse troppo semplice, l'Autopilot non disporrebbe di un numero sufficientemente differenziato di campioni e quindi risulterebbe difficile ottenere una capacità cognitiva soddisfacente.

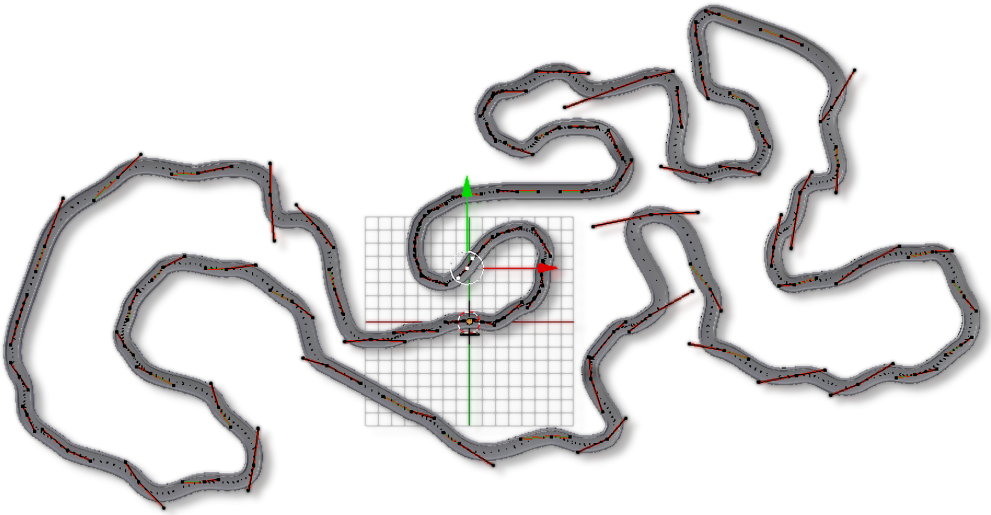


Figura 6.1: Vista progettuale del tracciato di test

In seguito a queste considerazioni, il percorso è stato realizzato in modo da risultare in maggior contrasto possibile con l'ambiente circostante ed in maniera tale che la sua forma presenti un numero equo e abbastanza vario di svolte a destra e sinistra; se così non fosse, infatti, potrebbe verificarsi una condizione tale per cui l'intelligenza artificiale tenderebbe a non essere equa nelle capacità di curvare.
Il modello 3D del tracciato è stato realizzato tramite il software di grafica *Blender*, utilizzando una curva *Bézier* per modificarne la conformazione.

6.2.2 Veicolo

Per una veloce sperimentazione la scelta del veicolo è ricaduta sull'asset *MS Vehicle System* [9] che predispone un semplice veicolo 3D con sterzo ed acceleratore.

Figura 6.2: Vista del veicolo

Il sistema è stato poi modificato per rendere i comandi intercambiabili tra input utente e intelligenza artificiale. Tramite la pressione di un tasto è possibile infatti arrestare la guida controllata dall'utente ed avviare l'autopilota. Sul veicolo è presente un componente chiamato *Autopilot* che svolge diversi compiti: Avvio del processo Python per la gestione dell'AI, streaming di immagini provenienti dal veicolo ed ovviamente output dei comandi predetti alla macchina.

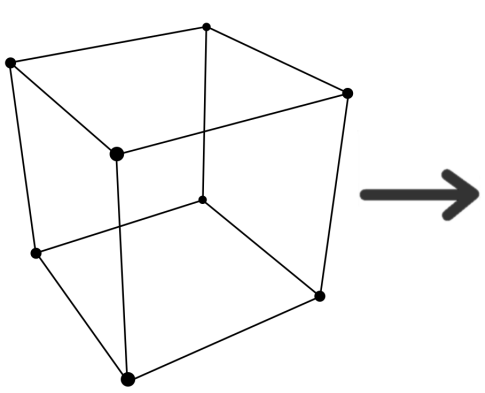
6.2.3 Acquisizione delle immagini e Base64

La registrazione di immagini dal mondo di gioco è un passo fondamentale, sia per la predizione di guida, sia per la creazione del Dataset su cui la rete neurale può essere allenata.
Per fare ciò ci serviamo di un sistema basato su *RenderTextures*:

```
1  private RenderTexture renderTexture;
2  private Texture2D renderedTexture2D;
3  private Rect renderedTextureRect;
4
5  public Camera cameraInput;
6
7  void Start () {
8      // Create a render texture with resolution 100x50 16bit
9      renderTexture = new RenderTexture(100, 50, 16);
```

```
10      // Create a 2D Image and a Rectangle of the same size
11      renderedTexture2D = new Texture2D(renderTexture.width,
         ↪  renderTexture.height);
12      renderedTextureRect = new Rect(0, 0, renderTexture.width,
         ↪  renderTexture.height);
13      // Set the camera to render into our texture
14      cameraInput.targetTexture = renderTexture;
15      // Force the rendering
16      cameraInput.Render();
17  }
18  // This method is used to fill a 2D image with PNG-encoded data from the render
     ↪  texture
19  public byte[] GetRenderTexturePixels()
20  {
21      RenderTexture.active = renderTexture;
22      renderedTexture2D.ReadPixels(renderedTextureRect, 0, 0);
23      renderedTexture2D.Apply();
24      return renderedTexture2D.EncodeToPNG();
25  }
```

Il metodo *GetRenderTexturePixels* viene utilizzato per convertire il contenuto della render texture in una immagine PNG codificata in *Base64*. Ciò permette di trasformare un'immagine (rappresentata da una serie di pixel) in una stringa testuale; questo, sebbene non il più efficiente, fornisce un modo semplice di trasferire dati tra il Simulatore ed il processo in Python poiché la comunicazione avviene tramite *STDIN/STDOUT* che sono canali di comunicazione di tipo testuale.



iVBORw0KGgoAAAANSUhEUgAAABIAAAARCAYAAADQWvz5A
AAAAXNSR0IArs4c6QAAAARnQU1BAACxjwv8YQUAAAAJcEhZ
cwAADsQAAA7EAZUrDhsAAAAGYktHRAD/
AP8A/6C9p5MAAAHRSURBVDhPhZPJrkFBElbLkAgxLcXaEHsr7
+gVLLwLOyKxMYt5nmMOfat+5xztXq4vke4q5/yn6u8um2LoC/
f7HT+n02lk/
vJHaLvd0vV6pfV6TQ6Hg263G3m9XioUCpRIJCgUCpHb7Tae1h
ChbDarRqORajQaarlcKhaT9AvtdhvrcDhU1WpVLRYLxCYQajab
qlKpIPEJeVlnt9upWq2m8vk8YrtUZbfbKR6PU7lcpsvlgkp/
Y7az2WyIK6fZbEaBQAA2AFETZRPZr1YrI1JqMpmoer2uisWi6na
7aF2n0+lghVCv11N8KkgI4lU6nUb+dDpBeL/fG/+
+YhYBocPhALN1RMSE21Dn89mInshHxuMx9vBIsNlsxu6B2Tu3
QvxVYrMR63Cl5Pf7sYeQGHk8HpEQuBoaDAbER04ul4tSqRRFIh
EqlUrGE0+sS4q6mEwmo/r9PowVxNh38MVES4J
+SJaQXMLpdGpED8M/IR+T09N9tDzy+XwYDzYVMf
+H9R3S5nw
+x9hYPPSesA9Y9bJ1Wq0WpkButs7b6ZcTkgrD4TBivh7EM4Yqo
9Eocr95KyQv5nI5SiaTGAU5Ypn6/7A80vF4PMSlUzAYpFgs9lWEi
OgHTDKjEilGxoAAAAASUVORK5CYII=

Figura 6.3: Immagine convertita in Base64

Si noti che nella precedente immagine la stringa illustrata fa riferimento ad una versione a bassa risoluzione dell'originale; questo poiché la lunghezza del testo risultante dalla codifica è proporzionale alle dimensioni dell'immagine convertita.

6.2.4 Registrazione del dataset

L'input visivo del veicolo è essenziale per il sistema; per fare in modo che la rete ottenga delle buone immagini su cui essere allenata, si colloca una telecamera virtuale sul tettuccio del veicolo, questa è differente da quella con cui l'utente visualizza il mondo 3D del simulatore ed è unicamente utilizzata per l'acquisizione di immagini. Indichiamo con *cameraInput* la variabile che fa riferimento a questa telecamera negli script di questa sezione.

Per assicurarsi di avere le migliori immagini possibili, si devono trovare i parametri migliori per la telecamera; dopo alcune sperimentazioni è risultato che un campo visivo (FOV) di 90° ed un rapporto di aspetto 2:1 permettono una visione ottima del tracciato, mantenendo il percorso sempre ben inquadrato anche durante curve strette.

Nel caso dello script per la generazione del dataset, un metodo simile a *GetRenderTexturePixels* è utilizzato insieme alla funzione *File.WriteAllBytes(path, image)* per memorizzare permanentemente ogni frame catturato su disco.

```
1  // Async method to record images
2  public IEnumerator RecordCameraDataToDisk()
3  {
4      while(shouldRecordImages) {
5          // Activate current rendertexture and read all pixels into a Texture2D
6          cameraInput.targetTexture = renderTexture;
7          RenderTexture.active = renderTexture;
8          renderedTexture2D.ReadPixels(renderedTextureRect, 0, 0);
9          renderedTexture2D.Apply();
10
11         // Create a filename [DATE_MILLISECONDS]_[IMG_NUM]_STEERING_ACCELERATOR
12         string finalFileName =
13             "[" + (DateTime.Now - new DateTime(1970, 1, 1)).TotalMilliseconds +
                ↪  "]_[" + currentImageSavedNumber + "]_" +
14             Map(vehicleController.GetCurrentSteeringAngle(),-1, 1, 0, 1) + "_"
                ↪  +
```

```
15              // Increase speed by datasetSpeedMultiplier so you can drive slower
                ↪  in Record mode
16              Map(Mathf.Clamp(vehicleController.GetCurrentAcceleration() *
                ↪  datasetSpeedMultiplier, -1, 1) ,-1, 1, 0, 1);
17
18          string path = Path.Combine(savePath,finalFileName + ".png");
19          currentImageSavedNumber++;
20          // Start async method to save image to disk
21          StartCoroutine(SaveImageToDisk(path, renderedTexture2D.EncodeToPNG()));
22          // Records an image as fast as 1 every 0.3s (But it depends on the
            ↪  current FPS)
23          yield return new WaitForSeconds(0.3f);
24      }
25  }
26  IEnumerator SaveImageToDisk(string path, byte [] image){
27      try{
28          File.WriteAllBytes(path, image);
29      }
30      catch (Exception e){
31          Debug.Log(e);
32      }
33      yield return null;
34  }
35  // Function that maps a value onto a different range of values
36  public float Map(float currentValue, float in_min, float in_max, float out_min,
    ↪  float out_max)
37  {
38      return (currentValue - in_min) * (out_max - out_min) / (in_max - in_min) +
        ↪  out_min;
39  }
```

Il nome del file è molto importante; il primo parametro è rappresentato dalla data di registrazione espressa in millisecondi e permette di mantenere i vari frame registrati in ordine, ciò torna utile nel caso di utilizzo sequenziale del dataset e permette di non mescolare le immagini nell'eventualità in cui si effettuino più registrazioni.
Nel filename inoltre sono codificate le informazioni essenziali per il training dell'intelligenza artificiale; per le sperimentazioni effettuate in questa tesi sono stati memorizzati due dati che rappresentano la 'label' di ogni immagine: L'angolo di sterzo e il valore di accelerazione (Range -1 a 1).
Come visto in precedenza nel caso dell'input visivo, è bene normalizzare il range

dei dati che transitano nella rete neurale, per questo, sia per output del database che successivo input delle predizioni, è utilizzata una funzione chiamata *Map*; questa è in grado di ri-mappare un dato numerico in un differente intervallo di valori. Nel nostro caso portiamo l'angolo di sterzo e acceleratore dal range (-1,1) a quello (0,1) e viceversa quando poi invece otteniamo gli input predetti dalla rete.

6.2.5 Comunicazione interprocesso tra C-Sharp e Python

L'anello di congiunzione fra il simulatore ed il cervello che gestisce l'Autopilot è la comunicazione fra i due processi. Un modo pulito e veloce per fare ciò è ottenere un eseguibile dallo script Python che gestisce le predizioni di guida, ed avviarlo come processo nascosto dal simulatore; le due applicazioni comunicano per via testuale tramite i canali *Standard Input STDIN* e *Standard Output STDOUT* dell'applicativo Python.
Normalmente queste modalità di comunicazione sono accessibili dalla console con cui si avvia lo script Python, ma è possibile reindirizzarli in modo che comunichino direttamente in modo trasparente con il programma che ha avviato il processo (nel nostro caso il simulatore).

Il codice che gestisce questa comunicazione da parte del simulatore è contenuto nello script *Autopilot* presente sul veicolo:

```
1  private ProcessStartInfo startInfo;
2  private Process pythonProcess;
3  public bool useAutoPilot = false;
4  public bool brainReady = false;
5  // Awake is called before Start
6  private void Awake() {
7      startInfo = new ProcessStartInfo {
8          FileName = @"AutopilotBrain_Drive.exe",
9          UseShellExecute = false,
10         CreateNoWindow = true,  // Hide process console
11         RedirectStandardOutput = true, // Redirect STDOUT
12         RedirectStandardInput = true // Redirect STDIN
13     };
14     // Start the hidden autopilot brain
15     pythonProcess = Process.Start(startInfo);
16 }
17 void Start () {
18     // Start the async method that manages the brain I/O
```

```
19      StartCoroutine(StartBrain());
20  }
21  IEnumerator StartBrain() {
22      if (!brainReady) {
23          startResponse = pythonProcess.StandardOutput.ReadLine();
24          if (startResponse != null)
25          {   // Check if the Python brain has loaded the neural network
26              if (startResponse == "MODEL_LOADED")
27              {   // Set the autopilot status as READY
28                  brainReady = true;
29              }
30          }
31      }
32      yield return null;
33  }
34  private void OnApplicationQuit() {
35      if (pythonProcess != null && !pythonProcess.HasExited)
36          pythonProcess.Kill();
37  }
```

Quando il sistema è pronto, si può procedere alla comunicazione da C-Sharp a Python, per la quale è sufficiente utilizzare i metodi:

- **pythonProcess.StandardInput.WriteLine(base64img)**:
  Per scrivere un'immagine sullo standard input del cervello dell'autopilota.
- **pythonProcess.StandardOutput.ReadLine().Split('_')**:
  Per ottenere un vettore i cui elementi rappresentano le predizioni della rete.

Questo 'protocollo di comunicazione' deve ovviamente essere supportato anche dall'applicativo in Python. Per fare ciò è stato implementato un semplice script di supporto che agevola la comunicazione col processo C-Sharp:

```
# Script 'CSharpImageStream.py'
import sys
import base64
# Wait for an image from the C# simulator
def ReceiveImage():
    userInput = sys.stdin.readline()
    userInput = userInput.replace('\n','')
    # Once received, decode the Base64 image
    return base64.b64decode(bytearray(userInput, 'utf-8'))
# Send an output to the simulator
def SendOutput(output):
    sys.stdout.write(output)
    # FLUSH IS NEEDED TO OUTPUT THE STDOUT TO THE STOUTPUT STREAM WITHOUT
     ↪  WAITING FOR PROGRAM EXIT
    sys.stdout.flush()
```

Nel software che svolge effettivamente le predizioni basta importare il precedente script ed utilizzarlo per le operazioni I/O:

```
# Script the streaming utility
import CSharpImageStream as CStream
def main():
    if not os.path.exists(os.path.join(SAVE_PATH,MODEL_NAME)):
        CStream.SendOutput("ERROR_MODEL_NOT_FOUND" + '\n')
    else: # Load the neural network model from disk
        model = load_model(os.path.join(SAVE_PATH,MODEL_NAME))
        CStream.SendOutput("MODEL_LOADED" + '\n')
        while(True): # This loops does the predictions
            receivedImage = CStream.ReceiveImage()
            # Predict (And then output the prediction to the simulator)
            predict_from_image(model, receivedImage)
# CALL THE MAIN FUNCTION WHEN EXECUTING THE SCRIPT
if __name__ == "__main__":
    main()
```

6.2.6 Ricezione predizioni Autopilot ed output di guida

Se il processo Python è stato avviato correttamente dal simulatore, il flag booleano *brainReady* è impostato su *True*; ciò indica che si può procedere con l'avvio del sistema di autopilota.

```
void Update () {
    // If the key 'K' is pressed, start the autopilot mode
    if (Input.GetKeyDown(KeyCode.K) && brainReady) {
        useAutoPilot = !useAutoPilot;
        vehicleController.SetAutopilotActive(useAutoPilot);
        if (useAutoPilot) {
            StartCoroutine(AutoPilot());
        } else {
            StopCoroutine(AutoPilot());
        }
    }
}
public IEnumerator AutoPilot() {
    // Predict as fast as you can (This is a non-blocking method)
    while (useAutoPilot) {
        StreamImageToBrain();
        yield return null;
    }
}
void StreamImageToBrain() {
    // Force re-render of camera into its rendertexture
    cameraInput.Render();
    // Send Base64 camera image onto STDIN of the Python brain
    string base64img = Convert.ToBase64String(GetRenderTexturePixels());
    pythonProcess.StandardInput.WriteLine(base64img);
    // Wait till we receive steering and acceleration predictions
    string[] results = pythonProcess.StandardOutput.ReadLine().Split('_');
    // Output the predictions to the vehicle drive system (Remapping the
     ↪  results to the correct range)
    vehicleController.SetSteering(Map(float.Parse(results[0]),0,1,-1,1));
    vehicleController.SetAcceleration(Map(float.Parse(results[1]),0,1,-1,1));
}
```

6.3 UI - Interfaccia Utente

La *User Interface* è ciò che permette all'utente di ottenere un feedback visivo su quello che sta accadendo nel simulatore.
Ci sono diverse tipologie di dato che dal simulatore possono essere mostrate all'utente, tra le quali informazioni di debug, feedback sul funzionamento dei sistemi di autopilota e registrazione del dataset ed anche dati numerici come il tempo di predizione, utile per misurare l'impatto prestazionale che vari tipi di rete neurale impongono sul calcolatore.
Unity mette a disposizione un sistema veloce per l'implementazione di interfacce utente; all'interno del simulatore si crea un oggetto *Canvas* al cui interno si inseriscono altri GameObjects rappresentanti testo, immagini o effetti grafici.

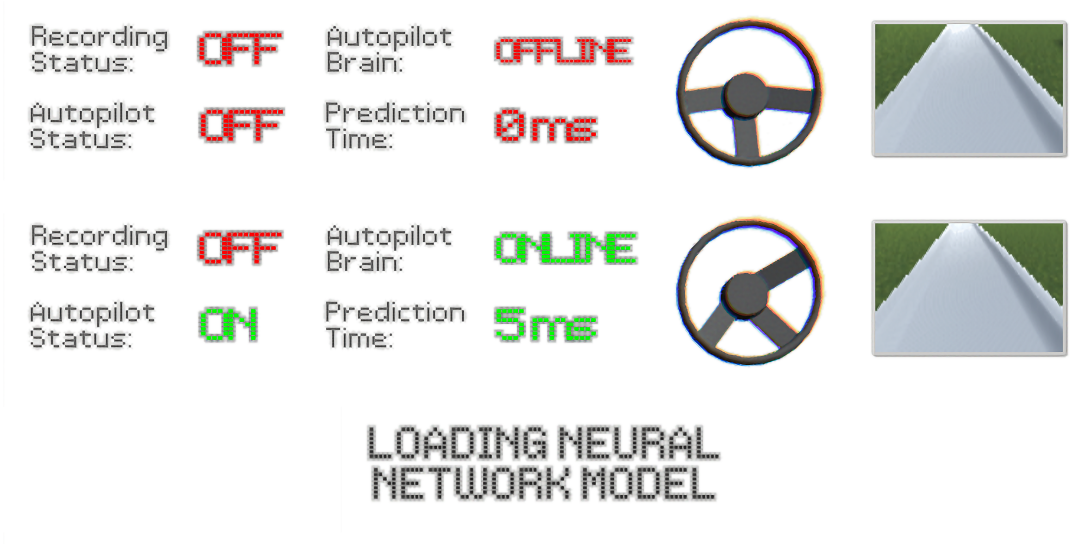


Figura 6.4: Esempi di interfaccia del simulatore

In figura è rappresentato un esempio di come l'interfaccia esponga i dati del simulatore; la prima sezione è relativa ad uno stato stazionario della simulazione, mentre la seconda mostra come l'interfaccia cambia quando si attiva l'Autopilot.
Nella parte destra sono mostrati l'angolo di sterzo correntemente predetto e un monitor che mostra l'immagine su cui viene effettuata la predizione attuale; infine si mostra la scritta che indica il caricamento del modello neurale, questa scompare quando il flag *brainReady* diventa True.

La gestione dell'UI è affidata ad uno script chiamato *UI_ Controller* situato nel-

l'oggetto Canvas; esso prende in input tutti gli elementi dinamici dell'interfaccia ed implementa i metodi utili per modificarli.
Questo codice gestisce anche l'emissione di suoni al verificarsi di eventi chiave come l'attivazione dell'autopilota o l'avvenuto caricamento della rete:

```
public Text AutopilotStatus;
public AudioSource source;
public AudioClip autopilotOn;
// Use a simil-singleton pattern to access this class
public static UI_Controller Instance = null;
private void Awake() {
    if (Instance == null) {
        Instance = this;
    }
}
/*
 * [OTHER UI METHODS]
 */
public void SetAutopilotStatus(bool status){
    if (status) {
        AutopilotStatus.text = "ON";
        // Play a sound when the autopilot is activated
        source.PlayOneShot(autopilotOn, 0.6f);
        AutopilotStatus.color = Color.green;
    } else {
        AutopilotStatus.text = "OFF";
        AutopilotStatus.color = Color.red;
    }
}
```

I metodi della classe *UI_ Controller* sono facilmente accessibili tramite l'istanza statica *Instance*; possiamo vedere un esempio dal codice dell'Autopilot:

```
void StreamImageToBrain() {
    int tick = DateTime.Now.Millisecond;
    /*
     * [IMAGE STREAMING CODE]
     */
    int tock = DateTime.Now.Millisecond;
    // Show the prediction time (In milliseconds) on the UI
    UI_Controller.Instance.SetPredictionTime(tock - tick);
}
```

6.4 Sistema di Autopilot - Training Script

Come detto in precedenza, il software in Python è suddiviso in due parti: Script di training e script di predizione.
Il primo, oltre a gestire il modello di rete neurale, si occupa di elaborare tutte le immagini salvate durante la fase di registrazione, memorizzandole poi su disco in un singolo file che rappresenta effettivamente il dataset; una volta che è stato generato, non è più necessario rielaborare le immagini al successivo avvio del codice.
A differenza dello script di predizione, quello di training non viene compilato in un eseguibile ed è utilizzato solo qualvolta si voglia modificare il modello di rete neurale, oppure rieffettuare il training.

6.4.1 Creazione file dataset e Data Augmentation

Data la directory in cui si trovano i frame della telecamera è possibile caricarli in memoria tramite un ciclo che scorre tutti i file della cartella. Nell'operazione torna utile la libreria *tqdm* che genera una barra di progresso nella console permettendo di visualizzare l'andamento del processo.

Figura 6.5: Esempio di barra di progresso generata dal pacchetto tqdm

Il codice per la creazione del dataset è contenuto in *preprocess_train_data()* ed appare come segue:

```
def preprocess_train_data():
    training_data = []
    training_targets = []
    count = 0
    for driving_frame in tqdm(os.listdir(TRAIN_DS_DIR)):
        # Remove the extension from the filename
        noExtensionFilename = driving_frame.replace('.png', '')
        # The parameter [2] in the image name is the steering angle we want to predict and [3] is the accelerator value
        img_label = [float(noExtensionFilename.split('_')[2]), float(noExtensionFilename.split('_')[3])]
        img_filepath = os.path.join(TRAIN_DS_DIR, driving_frame)
        # Once we have the labelled part of the data and the filepath, load the image and resize it using OpenCV to the correct input dimension for the neural network
        img_file = cv2.imread(img_filepath)
        img_file = cv2.resize(img_file, (IMG_RESIZE_SIZE_X, IMG_RESIZE_SIZE_Y))
        # We use grayscale
        img_file = cv2.cvtColor(img_file, cv2.COLOR_BGR2GRAY)

        # Add image and label to the training data array
        training_data.append(img_file)
        training_targets.append(img_label)

        # DATA AUGMENTATION
        imgAug = cv2.flip(img_file, 0) # Horizontal flip
        steerTmp = -1 * map(img_label[0], 0, 1, -1, 1) # Also flip the steering
        steerTmp = map(steerTmp, -1, 1, 0, 1)
        labelAug = [steerTmp, img_label[1]]

        training_data.append(imgAug)
        training_targets.append(labelAug)

```

```
30      training_data = np.array(training_data)
31      training_data = training_data.astype('float32')
32      training_data /= 255 # Normalise data to [0, 1] range
33      # Save the result so we don't need to do this again the next time
34      np.save(os.path.join(SAVE_PATH, TRAIN_IMG_SAVE_NAME), training_data)
35      np.save(os.path.join(SAVE_PATH, TRAIN_TAR_SAVE_NAME), training_targets)
36
37      return [training_data, training_targets]
```

Il processo di *Data Augmentation* visto nel codice è un modo comune per aumentare la quantità di campioni nel dataset. Si possono effettuare una serie di modifiche ai dati in modo da rendere più robusta la rete a diverse situazioni; per esempio si possono scurire o sfocare le immagini, oppure come in questo caso invertirle orizzontalmente.

All'avvio dello script, esso controlla la disponibilità di salvataggi precedenti; se ne trova, carica tutto in memoria, scompattando i file del dataset in array numpy che verranno poi utilizzati per il training. Nel caso sia disponibile anche un precedente modello salvato (chiamato *Checkpoint*), anch'esso viene caricato ed il prossimo apprendimento ripartirà da quel punto.

6.5 Sistema di Autopilot - Prediction Script

Lo script che si occupa di effettuare le predizioni di guida è una versione modificata di quello di training, ciò che cambia è l'assenza di tutta la parte che riguarda l'addestramento, la creazione del dataset e del modello di rete neurale.
Viene semplicemente caricato un checkpoint del modello da disco, e successivamente l'applicazione si mette in attesa di input da predire.

6.5.1 Preprocessing immagini e predizione

La parte di preprocessing delle immagini è identica allo script di training ma si noti che l'input proveniente dal simulatore è codificato in Base64, per cui è necessario prima decodificare il frame per poi processarlo ed infine effettuare una predizione. La parte di decodifica è già svolta dal metodo *CStream.ReceiveImage()* visto in precedenza, il suo output va tuttavia convertito in un'immagine caricabile da OpenCV:

```python
def base64_to_cv2_image(image_data):
    # Take the image data as uint8
    nparr = np.fromstring(image_data, np.uint8)
    # Decode the data as an image
    img = cv2.imdecode(nparr, cv2.IMREAD_COLOR)
    return img
```

6.5.2 Generazione eseguibile con Pyinstaller

Dato che il simulatore deve poter avviare un processo Python nascosto, l'ideale è possedere un eseguibile dello script che contenga tutto il necessario, da codice a librerie, in modo che un possibile utente terzo non debba installare per forza ogni singolo pacchetto.
Per fare ciò è utilizzata la libreria *Pyinstaller* che permette di ottenere file eseguibili a partire da codici sorgente in Python.
L'utilizzo del pacchetto è molto semplice ma per ridurre l'inclusione di librerie inutilizzate nell'eseguibile è opportuno segnalare a linea di comando tutti i pacchetti non necessari, o almeno quelli più pesanti. Il comando per la creazione del file .exe è quindi simile al seguente:

```
> Pyinstaller .\AutopilotBrain_Drive.py --exclude matplotlib --exclude Tornado
> --exclude scikit-learn --exclude time --exclude tqdm --onefile
```

L'opzione *–onefile* è molto comoda in quanto impartisce la creazione di un singolo file autocontenuto invece che di una cartella contenente librerie ed eseguibile.

6.6 Modello neurale di base

La parte finale del progetto consiste nella creazione di un modello neurale in Keras, che permetta alla macchina di guidarsi da sola. Il primo passo è identificare la tipologia di rete che sia adatta allo scopo:
dato che abbiamo degli input dotati di labels, e dobbiamo predire una serie di valori numerici a partire da immagini, la classe del nostro problema ricade in un sistema di regressione lineare nell'ambito del supervised learning.

Nella letteratura riguardante problemi di questo dominio è risultato rilevante uno studio elaborato da **NVIDIA Corporation**, che studia proprio la realizzazione di un modello convoluzionale per veicoli a guida autonoma. Nel paper ***End to End Learning for Self-Driving Cars*** [2], NVIDIA fa riferimento ad un modello CNN composto da 5 strati convoluzionali e 5 strati interamente connessi (Comprendendo il layer di output).

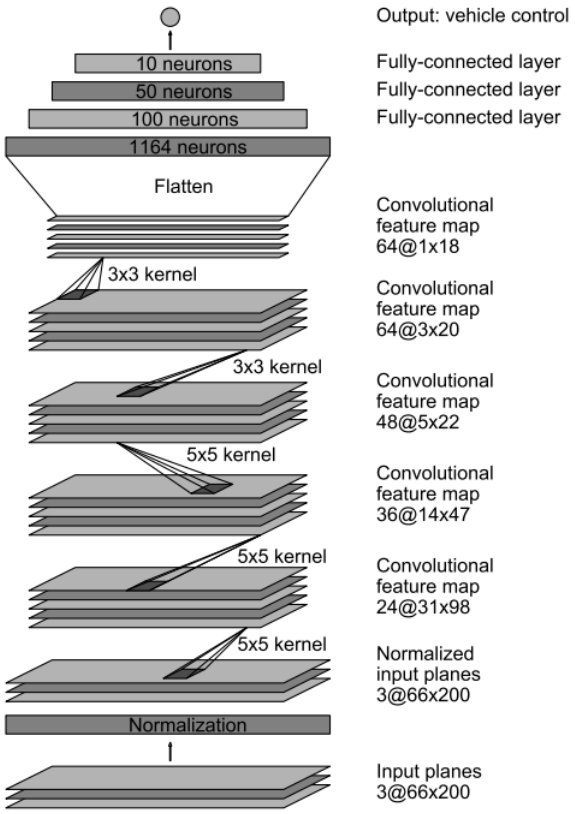


Figura 6.6: Modello CNN suggerito da NVIDIA [2]

Questo sistema tuttavia è progettato per l'input di 3 telecamere e per la guida di una vera automobile, il che lo rende un modello eccellente ma estremamente complesso e non adatto al nostro problema virtuale. Si può però prendere in considerazione un sistema simile in scala ridotta:

Dopo varie ottimizzazioni un primo modello funzionante della rete è ottenuto come segue:

```
model = Sequential()
model.add(Conv2D(4, kernel_size=(2, 2), kernel_regularizer=l2(0.002),
 ↪  activation="elu", input_shape=(IMG_RESIZE_SIZE_Y, IMG_RESIZE_SIZE_X, 1)))
model.add(MaxPool2D(2,2))
model.add(Conv2D(16, kernel_size=(2, 2), kernel_regularizer=l2(0.002),
 ↪  activation="elu"))
model.add(MaxPool2D(2,2))
model.add(Conv2D(88, kernel_size=(2, 2), kernel_regularizer=l2(0.002),
 ↪  activation="elu"))
model.add(MaxPool2D(2,2))
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(256, activation="elu"))
model.add(Dense(64, activation="elu"))
model.add(Dropout(0.1))
model.add(Dense(2))
model.summary()
```

Si tratta di una CNN composta da 3 layer convoluzionali e 3 interconnessi (output compreso). Uno degli obiettivi di questo progetto è quello di ottenere un modello performante ma che, allo stesso tempo, sia molto rapido nel training. L'allenamento di CNN è tuttavia spesso dispendioso in termini di tempo e di potenza di calcolo, per questo motivo sono stati utilizzati alcuni accorgimenti: degli strati di subsamplig con filtro Max sono stati interposti ai layer convoluzionali per ridurre il numero di campioni ed inoltre si è cercato di ottimizzare i parametri della rete in modo da ottenere buoni risultati con pochi neuroni negli strati interconnessi.
Il dataset di training è composto da alcune migliaia di frame (raddoppiate dopo la Data Augmentation); possono sembrare molti, ma nell'ambito del machine learning, i dataset sono composti anche da centinaia di migliaia di campioni, quindi, in questo caso, il numero non è troppo elevato ed è proporzionato alle dimensioni del modello; una rete molto più grande infatti avrebbe bisogno di numerosi campioni in più per non subire overfitting e performare al meglio.
Il parametro *kernel_regularizer* permette di specificare una tecnica di regolarizzazione del layer (in questo caso L2), questo serve, insieme a Dropout, per ridurre il rischio di overfitting ed è consigliabile quando il training set ha dimensioni limitate.

Questa rete presenta 495k parametri ed è stata allenata con l'ottimizzatore *'Adam'* ottenendo una *Validation Loss* pari a 0.046 dopo un tempo di training di solo 2:30 minuti, il checkpoint del modello finale pesa 5.8Mb:

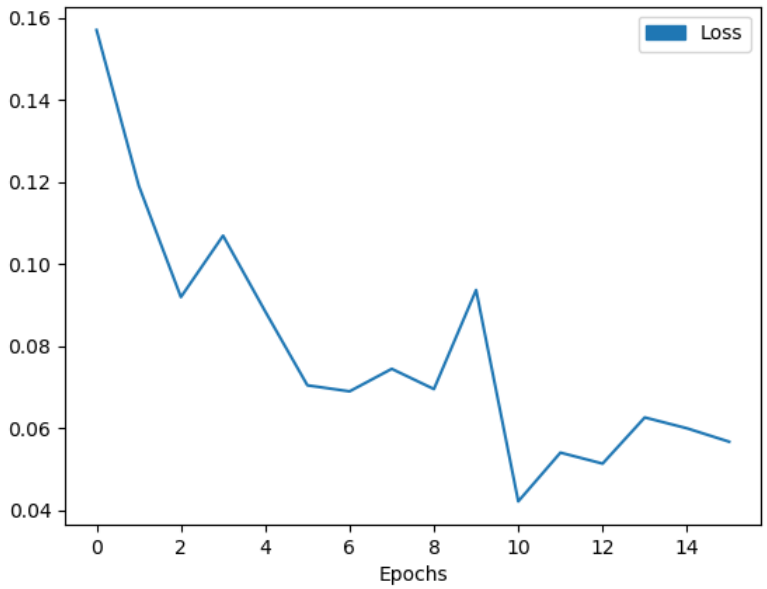


Figura 6.7: Grafico perdita di validazione del modello base

Dal grafico si può dedurre che un possibile miglioramento potrebbe essere la diminuzione del learning rate, ma data la breve durata dell'addestramento ed il punteggio di loss già relativamente basso, ciò può essere trascurabile.

6.6.1 Analisi prestazionale del modello

È ora di mettere alla prova il modello neurale creato avviando il simulatore, andremo di seguito ad analizzare prestazioni e performance di guida:

- **Tempo di predizione**: Una volta attivato il sistema di Autopilot, il tempo di predizione medio si aggira sui 5ms il che permette un'efficace simulazione senza latenze.
- **Comportamento in rettilineo**: La macchina riesce a mantenere una buona direzionalità.

- **Comportamento in curva**: Il veicolo riesce a seguire la curva ma si trova in difficoltà nei pressi delle sponde del percorso. Quest'ultimo comportamento, unito alla difficoltà di effettuare curve strette, fa pensare ad un problema di percezione spazio-temporale in cui il veicolo non riesce a rendersi conto di come modulare acceleratore e sterzo in base alla successione di curve.

- **Accelerazione**: Il veicolo accelera in modo deciso nei tratti rettilinei, ma minore rispetto alla fase di registrazione del dataset, si riscontra inoltre un'eccessiva fluttuazione nell'utilizzo dell'acceleratore, altro possibile indicatore di una carenza percettiva.

- **Traiettoria**: La traiettoria è prevalentemente regolare ma si denota una capacità di sterzo inferiore al necessario in uscita dalle curve ed eccessiva in entrata, il che alcune volte porta il veicolo ad uscire fuori pista.

Di seguito è riportato un time-lapse rappresentante il veicolo in modalità Autopilot visto dall'alto:

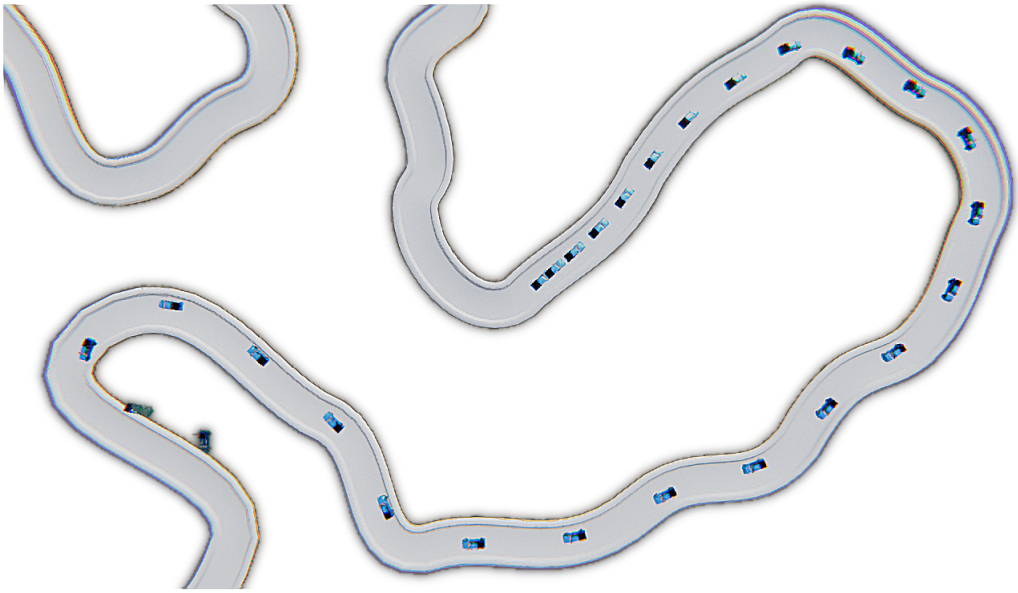


Figura 6.8: Modello di base alla guida

Come si evince dalla precedente immagine, il modello sembra performare discretamente bene, presenta tuttavia delle caratteristiche di guida che in certe situazioni, come si vede dal time-lapse, portano alla fuoriuscita del veicolo dal tracciato. Possiamo realizzare due grafici che rappresentino lo sterzo e l'acceleratore della macchina, ciò che ci si aspetta da essi è una corrispondenza tra le fluttuazioni nello sterzo e l'utilizzo dell'acceleratore; in particolare, se si verifica un picco nello sterzo (presenza di una curva) ci si aspetteranno due cose:

- Una conseguente diminuzione dell'acceleratore poco prima, il che rappresenta la capacità del veicolo di rallentare prima di una sterzata.
- Un aumento dell'accelerazione in corrispondenza dell'uscita di una curva.

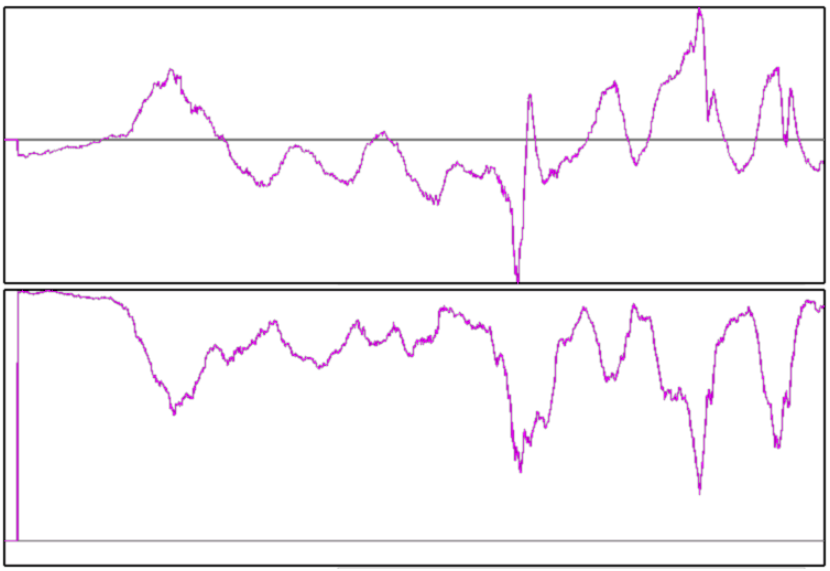


Figura 6.9: Grafico sterzo ed acceleratore CNN base

Come si può notare, è presente una corrispondenza evidente ma si vede anche che l'utilizzo dell'acceleratore è piuttosto brusco mentre quello dello sterzo è spesso moderato.

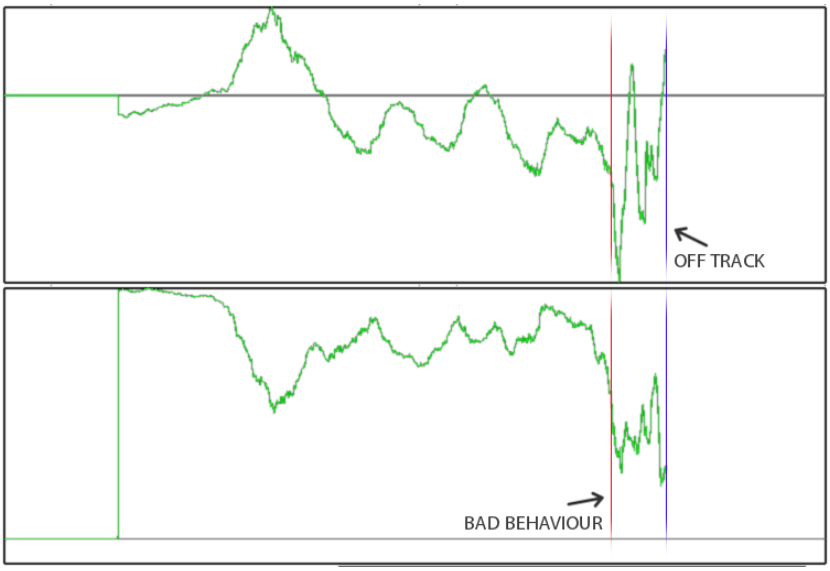


Figura 6.10: Grafico off-track CNN base

Molti di questi indizi mostrano che la rete riesce a copiare il comportamento visto nel dataset ma non è in grado di ricordare le relazioni temporali fra successioni di frame, i quali potrebbero rappresentare in modo decisivo il come l'utente affronta una determinata sezione del percorso.

6.7 Sperimentazione: Modello neurale con LSTM

Per risolvere questi problemi, l'idea è quella ti utilizzare uno strato ricorrente LSTM per aumentare le capacità cognitive della rete. L'utilizzo di una CNN abbinata a reti LSTM è diffuso soprattutto nella predizione di serie temporali di dati, come per esempio una sequenza di immagini all'interno di un video e per questo potrebbe risultare non immediato il loro utilizzo nella predizione di valori a partire da immagini singole, invece che da serie temporali.
Tuttavia, come citato nell'articolo **The Unreasonable Effectiveness of Recurrent Neural Networks** [5], è stato dimostrato che anche in assenza di input sequenziali, le reti LSTM hanno la capacità di processare i dati al loro interno in modo temporale, sono quindi in grado di fornire un output di tipo one-to-one che è comunque basato su un apprendimento interno sequenziale. Questa capacità permettono alle LSTM di mantenere una migliore rappresentazione spazio-temporale dei dati.

6.7.1 Ideazione di un modello con LSTM

Partendo da questi presupposti è stato realizzato un modello neurale di prova per verificare se il simulatore effettivamente ne potesse trarre beneficio.
Questo nuovo stack di layer contiene uno strato LSTM con 16 neuroni, il cui input è la precedente parte convoluzionale della rete. L'output risultante dalla prima parte della rete, deve essere adattato per l'input allo strato LSTM, ciò viene effettuato tramite l'applicazione di un layer *TimeDistributed()* che suddivide l'input in porzioni temporali.
Il modello Keras è composto come segue:

```
model = Sequential()
model.add(Conv2D(4, kernel_size=(2, 2), kernel_regularizer=l2(0.002),
 ↪  activation="elu", input_shape=(IMG_RESIZE_SIZE_Y, IMG_RESIZE_SIZE_X, 1)))
model.add(MaxPool2D(2,2))
model.add(Conv2D(16, kernel_size=(2, 2), kernel_regularizer=l2(0.002),
 ↪  activation="elu"))
model.add(MaxPool2D(2,2))
model.add(Conv2D(88, kernel_size=(2, 2), kernel_regularizer=l2(0.002),
 ↪  activation="elu"))
model.add(MaxPool2D(2,2))
model.add(Dropout(0.2))
model.add(TimeDistributed(Flatten()))
model.add(LSTM(16))
model.add(Dense(64, activation="elu"))
model.add(Dense(32, activation="elu"))
model.add(Dropout(0.1))
model.add(Dense(2))
model.summary()
```

6.7.2 Training e caratteristiche

L'ottimizzatore consigliato per le reti ricorrenti è RMSprop, per questo modello ne è stata implementata una versione customizzata che riduce il learning rate:

```
# Create a custom RMSprop optimizer with lower learning rate:

optimizerRMSPROP = RMSprop(lr=0.0005, rho=0.9, epsilon=None, decay=0.0)
```

Questa rete presenta un numero di parametri inferiore alla precedente, ovvero circa 72k ed il training è quindi di conseguenza molto più veloce; il tempo totale impiegato per l'allenamento è di 51s, il che è estremamente ridotto rispetto al precedente.
Il training si è concluso con una *Validation Loss* di 0.041 ovvero leggermente inferiore al precedente modello (fattore positivo) ed il risultante file di checkpoint pesa soltanto 600KB.

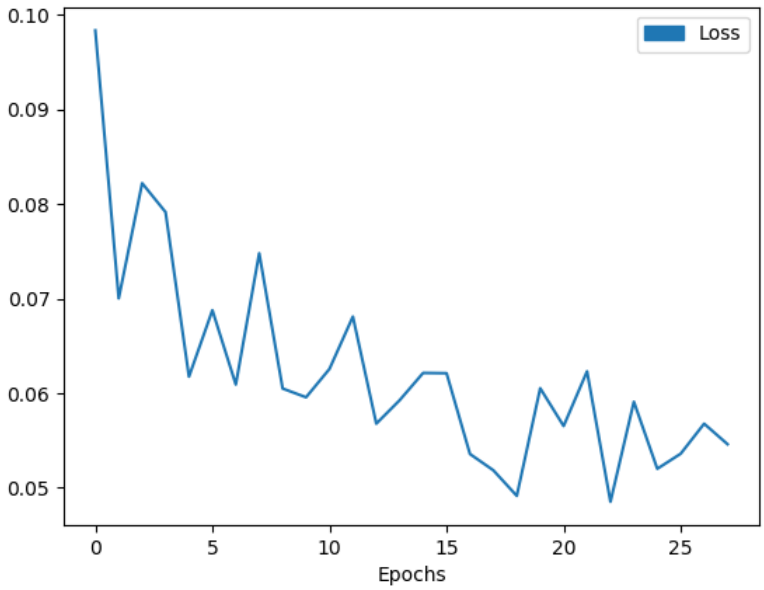


Figura 6.11: Grafico perdita di validazione del modello con LSTM

6.7.3 Analisi prestazionale del modello

Passando al testing di questo modello sperimentale, se ne analizza il comportamento e si riporta il time-lapse di guida di seguito:

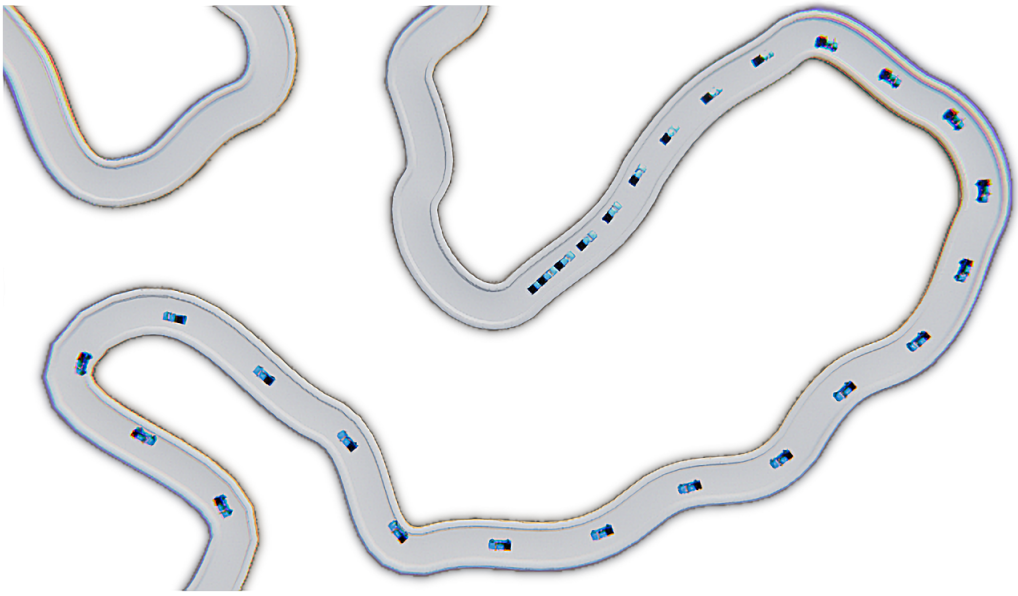


Figura 6.12: Modello con LSTM alla guida

Come si evidenzia dal time-lapse di questo nuovo modello, le sue performance sono superiori al precedente e riesce a completare il difficile tratto di pista senza problemi; segue l'analisi dettagliata del suo comportamento di guida:

- **Tempo di predizione**: In questo caso, dato il numero inferiore di parametri, il tempo di predizione si aggira sui 4ms.
- **Comportamento in rettilineo**: La macchina segue in modo molto preciso le fiancate del percorso.
- **Comportamento in curva**: Il veicolo affronta le curve con precisione mantenendosi sorprendentemente allineato col centro del percorso. Nel caso di curve strette la macchina raramente tocca poche volte il cordolo della strada e tende piuttosto a tagliare la curva (percorrendola quindi nel modo più rettilineo possibile).
- **Accelerazione**: L'accelerazione si presenta più moderata rispetto al modello precedente. Spesso l'utilizzo dell'acceleratore si rivela non sempre in

diretta corrispondenza alla quantità di sterzo, questo riflette in modo più preciso il comportamento dell'utente durante la fase di registrazione, poiché la potenza di frenata non è sempre la stessa all'interno del percorso, ma varia a seconda della difficoltà specifica della curva. Questa difficoltà nei vari punti del percorso è legata anche alla velocità del veicolo, dimensionalità che però non è disponibile alla rete neurale, in quanto i suoi unici input sono le immagini del percorso.

- **Traiettoria**: La traiettoria si presenta costante e precisa, sono rare le volte in cui il veicolo esce fuori pista; questo avviene soprattutto in presenza di velocità elevate (nonostante questo modello gestisca meglio velocità di percorrenza più alte rispetto al precedente).

Di seguito si riportano acceleratore e sterzo della rete sperimentale, compresa una sovrapposizione dei grafici di entrambi i modelli per evidenziarne le differenze:

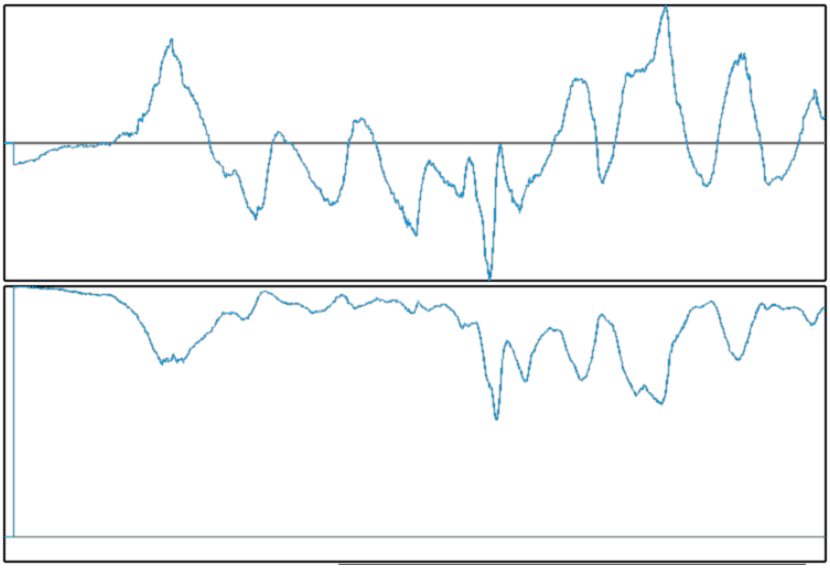


Figura 6.13: Grafico sterzo ed acceleratore CNN con LSTM

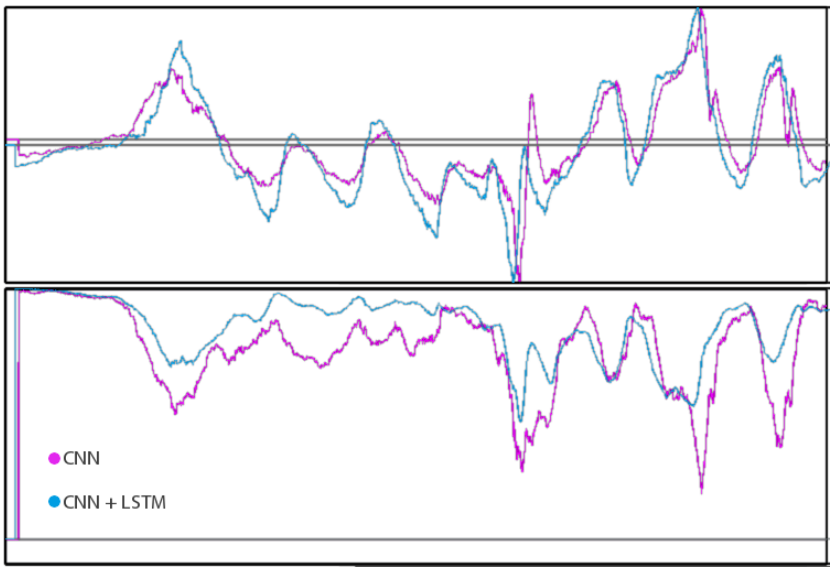


Figura 6.14: Grafico sovrapposto dei due modelli neurali

In quest'ultima immagine finale si mettono a confronto le due reti neurali, in magenta il modello CNN base, mentre in ciano quello sperimentale.
Come si può notare è evidente un uso più deciso dello sterzo da parte del sistema con LSTM, caratteristica che gli conferisce una risposta in curva più decisa e determinante nelle sezioni di percorso più tortuose. Per quanto concerne l'aspetto di accelerazione invece, si denota un utilizzo più ponderato dell'acceleratore, mettendo anche in evidenza il leggero anticipo di frenata in corrispondenza delle curve, caratteristica che non si presenta nel modello di base.

6.8 Conclusioni e Sviluppi

Il sistema realizzato è funzionante ed ottiene buone prestazioni sia sul percorso di prova che su altri tracciati non esposti nel set di training. La sperimentazione con le reti LSTM si è dimostrata valida per aumentare le capacità del modello ed ha fornito ottimi risultati.

Tra le migliorie apportabili al sistema c'è l'inserimento di ulteriori input alla rete; si può infatti modificare il modello per ricevere ulteriori parametri che descrivano lo stato fisico del veicolo, come per esempio: velocità attuale e distanze dai cordoli stradali. Si può anche pensare di aggiungere ulteriori telecamere per ampliare la percezione visiva del modello.
Una migliore comprensione spaziale è ottenibile eliminando dai modelli gli strati di MaxPooling; questo poiché essi, nonostante migliorino le velocità di training, distruggono parte delle informazioni, riducendo la robustezza della rete a cambiamenti nell'input visivo.

Questa tesi realizza una semplificazione di un modello di autopilota ma ulteriori approcci alla guida tramite machine-learning vengono continuamente sperimentati anche su veicoli reali.
Molti sono poi gli studi esposti nella letteratura (si veda ad esempio [3]), e l'ambito è in fervente crescita, soprattutto grazie alla recente fama che i veicoli a guida autonoma hanno acquisito dopo le implementazioni di aziende quali *Tesla Motors* e *Reanult*.

Bibliografia

[1] Michael Bain and Claude Sammut. ***A Framework For Behavioural Cloning***. Department of Artificial Intelligence, University of New South Wales.

[2] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. ***End To End Learning For Self-driving Cars***. NVIDIA Corporation.

[3] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. ***Deepdriving: Learning Affordance For Direct Perception In Autonomous Driving***. Princeton University.

[4] Sepp Hochreiter and Jürgen Schmidhuber. ***Long Short-term Memory***. 1997.

[5] Andrej Karpathy. ***The Unreasonable Effectiveness Of Recurrent Neural Networks***.

[6] Yann LeCun, Léon Bottou, Yoshua Bengio, and Partick Haffner. ***Gradient-based Learning Applied To Document Recognition***.

[7] Warren S. McCulloch and Walter H. Pitts. ***A Logical Calculus Of The Ideas Immanent In Nervous Activity***.

[8] F. Rosenblatt. ***The Perceptron: A Probabilistic Model For Information Storage And Organization***. Cornell Aeronautical Laboratory.

[9] Marcos Schultz. ***Ms Vehicle System***.

Ringraziamenti

Un enorme ringraziamento va alla professoressa e mia relatrice Damiana Lazzaro, per il suo sostegno, il suo sempre celere aiuto, ed il suo riguardo nei confronti di noi studenti in questo lungo percorso di studi.
Ringrazio anche il professor Davide Maltoni per i suoi consigli durante la stesura di questa tesi.

Un grazie a tutta la mia famiglia, per avermi sempre spronato e sostenuto nel raggiungimento di questo traguardo; ringrazio loro anche per avermi mantenuto in vita quando ero troppo impegnato a studiare per pensare a cose banali come mangiare o dormire.
Ringrazio anche tutti i miei parenti per aver creduto in me durante questi anni ed una dedica particolare va inoltre a mio nonno e mio zio, per avermi instillato sin da piccolo la voglia di imparare e *smanettare*, ricordandomi sempre:

" Tu devi diventare ingegnere! "

Non potrebbe poi mancare il ringraziamento finale a tutti i miei amici, quelli di sempre, che già sanno come sono fatto e quelli acquisiti durante il percorso, che lo hanno scoperto pian piano.
Grazie perciò a voi: Emilio, Mattia, Alice, Simone, Federico, Giulia... citarvi tutti sarebbe impossibile ma grazie per avermi supportato e sopportato fino alla fine.