



SAPIENZA
UNIVERSITÀ DI ROMA

Implementazione di un algoritmo di few-shot learning in ambiente Duckietown

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Corso di Laurea in Ingegneria Informatica e Automatica

Candidata
Michela Proietti
Matricola 1739846

Relatore
Prof. Daniele Nardi

Anno Accademico 2019/2020

Implementazione di un algoritmo di few-shot learning in ambiente Duckietown
Tesi di Laurea. Sapienza – Università di Roma

© 2020 Michela Proietti. Tutti i diritti riservati

Questa tesi è stata composta con L^AT_EX e la classe Sapthesis.

Email dell'autore: proietti.1739846@studenti.uniroma1.it

Sommario

Si vogliono presentare i risultati ottenuti nella classificazione di immagini mediante un algoritmo di few-shot learning e le differenze nelle prestazioni di tale algoritmo se applicato a due diversi datasets riguardanti Duckietown, dei quali uno simulato e uno reale. In particolare, si mostrano i tempi necessari per la classificazione nei due casi e come variano i risultati in funzione del diverso numero di immagini di training usate. Viene inoltre presentato un possibile modo per rendere l'algoritmo utilizzabile nel caso in cui si abbiano immagini contenenti più oggetti di interesse.

Indice

Introduzione	iv
1 Strumenti utilizzati	1
1.1 Duckietown	1
1.2 Descrizione dei datasets	1
1.2.1 Dataset simulato	2
1.2.2 Dataset reale	4
1.2.3 Sottoinsieme del dataset reale	6
1.3 Preprocessing	7
2 Metodologia	8
2.1 Classificazione	8
2.1.1 Multiclass classification	8
2.2 Reti neurali	8
2.2.1 Reti neurali convoluzionali	9
2.2.2 Funzioni di attivazione	10
2.2.3 Max Pooling	11
2.2.4 Funzione di costo	11
2.2.5 Metodi di regolarizzazione	12
2.2.6 Adam	12
2.3 Few-shot learning	13
2.4 Descrizione della rete utilizzata	13
2.4.1 Adattamento del modello al dataset reale	14
3 Risultati	17
3.1 Esperimenti sul dataset simulato	17
3.1.1 Esempio di applicazione in Duckietown	20
3.1.2 Aggiunta delle classi ground e sky	21
3.2 Esperimenti sul dataset reale	25
3.2.1 Utilizzo della rete addestrata sul dataset simulato	25
3.2.2 Addestramento della rete sul dataset reale con 7 classi	26
3.2.3 Rete addestrata sia sul dataset simulato che reale	28
3.2.4 Aggiunta delle classi wall e road	31
4 Conclusioni	33
Bibliografia	35

Introduzione

In questa relazione si affronta il problema della classificazione di immagini attraverso un algoritmo di few-shot learning da utilizzare nell'ambito del progetto Duckietown.

Nel few-shot learning si utilizza un numero molto ridotto di immagini di training, perciò risulta essere una tecnica molto efficace quando si hanno a disposizione pochi dati. Duckietown è una piattaforma per l'apprendimento della robotica e dei principi delle auto a guida autonoma nata nel 2016 al Massachusetts Institute of Technology (MIT) di Boston. Trattandosi di un progetto relativamente recente, è stato possibile trovare in rete un solo dataset relativo a una città in scala reale e realizzare datasets tramite il simulatore Gym-Duckietown richiede molto tempo, dal momento che si devono creare di volta in volta città contenenti oggetti diversi, ricavare degli screenshots e ritagliarli in modo da avere immagini con un solo oggetto di interesse. Il few-shot learning risulta dunque essere la tecnica più adatta per affrontare in questo caso il problema della classificazione di immagini. Quando si hanno a disposizione pochi dati, c'è un elevato rischio che l'algoritmo di apprendimento faccia overfitting sui dati di training, non riuscendo più a generalizzare, cioè non essendo più in grado di riconoscere nuove immagini che gli vengano sottoposte. Per evitare questo inconveniente, l'approccio che viene utilizzato in questo progetto è quello che viene definito come parameter-level approach, nel quale si cerca di limitare lo spazio dei parametri attraverso tecniche di regolarizzazione e funzioni di costo.

Nei vari addestramenti fatti nell'ambito di questo progetto, si è cercato di migliorare di volta in volta l'accuratezza dell'algoritmo, arrivando ad ottenere sui due datasets usati, dei quali uno simulato e uno reale, risultati ottimi, che indicano che l'overfitting è stato evitato con successo. Inoltre, pensando a un eventuale futuro utilizzo dell'algoritmo nell'ambito di Duckietown, si è posta una particolare attenzione alle prestazioni raggiunte. Più specificatamente, tenendo conto dei tempi impiegati dall'algoritmo per effettuare la classificazione sull'intero test set e su una singola immagine, si è lavorato sulla struttura stessa della rete, nel tentativo di renderla più leggera possibile, pur non causando un eccessivo peggioramento dei risultati, riducendo il numero di livelli e il numero di nodi in ciascun livello. Così facendo, utilizzando un massimo di 6 immagini di training per classe, è stato possibile ottenere livelli di accuratezza molto elevati e tempi di classificazione vicini al real-time. Tali risultati sono sorprendenti, se si pensa che sia nel caso del dataset simulato che nel caso del dataset reale, la qualità e la varietà delle immagini risultano essere molto ridotte.

Per finire, dal momento che sui Duckiebot è presente una telecamera che fornisce la visuale completa della strada antistante e non immagini con oggetti singoli, come possibile modalità di applicazione dell'algoritmo sviluppato, sono state prese

delle immagini contenenti più oggetti, è stata effettuata una divisione in più parti e una successiva classificazione di ciascuna patch tenendo conto, anche in questo caso, dei tempi necessari per effettuare tali operazioni. Di seguito vengono riportati a grandi linee i contenuti dei singoli capitoli che costituiscono la relazione.

Nel [capitolo 1](#) si spiega brevemente cos'è Duckietown e si espongono i motivi per cui risulta interessante sviluppare un algoritmo di few-shot learning per la classificazione nell'ambito di tale progetto. Vengono quindi presentati i due datasets utilizzati, mostrando la distribuzione delle immagini tra le varie classi e un esempio di immagine per ciascuna di esse. Infine, sono illustrate le operazioni di preprocessing eseguite su ciascun dataset.

Nel [capitolo 2](#) si fanno dei cenni alla teoria che è alla base di questo progetto. In particolare, si riportano le definizioni di classificazione e multiclass classification. Viene poi spiegato il funzionamento delle reti neurali e, nello specifico, delle reti neurali convoluzionali, parlando nel dettaglio delle funzioni di attivazione, delle funzioni di costo e dei metodi di regolarizzazione e ottimizzazione utilizzati. Si illustra quindi l'idea di base del few-shot learning e si passa alla descrizione del ragionamento che ha portato allo sviluppo della rete che fornisce i migliori risultati in fase di test e di come essa sia stata successivamente adattata al dataset reale.

Nel [capitolo 3](#) vengono riportati i risultati dei vari esperimenti. In particolare, si mostrano i valori di accuracy e loss ottenuti sul test set dopo aver effettuato gli addestramenti e i tempi necessari per effettuare la classificazione sull'intero test set e su un singolo sample. Sia per il dataset simulato che per quello reale, inoltre, viene illustrata una possibile tecnica che potrebbe permettere di utilizzare l'algoritmo creato all'interno di Duckietown. Per quanto riguarda il dataset reale, viene fatto uno studio approfondito del numero di immagini di training da utilizzare per ogni classe, al fine di ottenere i risultati migliori possibili.

Nel [capitolo 4](#) è possibile trovare le conclusioni a cui si è giunti attraverso questo progetto. Si riassumono brevemente i risultati più significativi ottenuti, giudicando se sia opportuno o meno utilizzare il few-shot learning nell'ambito di Duckietown, quali sono gli aspetti positivi e quali quelli negativi, che vanno migliorati.

Capitolo 1

Strumenti utilizzati

1.1 Duckietown

Duckietown è una piattaforma per l'apprendimento della robotica e dei principi delle auto a guida autonoma nata nel 2016 al Massachusetts Institute of Technology (MIT) di Boston. Essa è formata da piccoli veicoli a due ruote, definiti Duckiebots, con a bordo una paperella di gomma, o duckie, che si muovono all'interno di una città in miniatura, detta Duckietown, rispettando il codice della strada. Ciascun Duckiebot ha un solo sensore, rappresentato da una telecamera frontale, due motori elettrici per muoversi e cinque LED per comunicare con gli altri robot. Tutte le decisioni vengono prese a bordo di ciascun Duckiebot grazie alla presenza di un Raspberry-PI. Duckietown include inoltre un simulatore, Gym-Duckietown, progettato per essere fisicamente realistico e facilmente compatibile con il mondo reale. All'interno di tale simulatore è possibile collocare Duckiebots in città simulate la cui struttura può essere modificata, introducendo incroci, curve, duckies, ostacoli, semafori o segnali stradali. Essendo Duckietown un progetto relativamente recente, ancora non si hanno a disposizione molte immagini e quindi grandi datasets per addestrare algoritmi di machine learning. Allo stesso tempo, creare un nuovo grande dataset richiederebbe molto tempo, dal momento che si dovrebbero raccogliere molte immagini ed effettuare il labeling. Nel caso di un dataset reale, inoltre, si dovrebbe già disporre di una città in scala che contenga al suo interno molti oggetti appartenenti alle varie classi, in modo da poter ottenere immagini variegate. Un possibile approccio per affrontare tale problema è dunque quello di utilizzare un algoritmo di few-shot learning, che permetta di effettuare la classificazione delle immagini avendo a disposizione pochi samples di training.

1.2 Descrizione dei datasets

Nell'ambito di questo progetto sono stati utilizzati due diversi datasets realizzati manualmente. In particolare, uno dei datasets è stato costruito mediante il simulatore Gym-Duckietown, creando mappe diverse, facendo degli screenshots e ritagliandoli di volta in volta, in modo da ottenere immagini contenenti un unico oggetto di interesse, visto da diverse angolazioni. Il secondo dataset è stato costruito a partire da un dataset preesistente trovato in rete¹, ritagliando le immagini in modo tale da renderle adatte per la classificazione. Entrambi i datasets sono piuttosto piccoli, ma permettono di effettuare una valutazione efficace della rete.

¹È possibile trovare il dataset completo all'indirizzo <https://github.com/marquez0/darknet/tree/master/duckiestuff>

1.2.1 Dataset simulato

Il dataset simulato è costituito da 283 immagini di dimensioni diverse. Esso è diviso in 10 classi, i cui nomi corrispondono alle label da associare alle immagini che le costituiscono. Il training set contiene 5 immagini per ogni classe, per un totale di 50 immagini, mentre le restanti 233 immagini costituiscono il test set. La distribuzione dei dati è riportata nella tabella 1.1, in figura 1.1 e 1.2. La figura 1.3, invece, mostra un esempio per classe delle immagini del dataset, con la relativa classe di appartenenza.

Tabella 1.1. Numero istanze di training e test per classe nel dataset simulato.

Classe	Numero istanze di training	Numero istanze di test
barrier	5	32
building	5	26
bus	5	24
cone	5	20
duckie	5	20
duckiebot	5	25
stop	5	20
trafficlight	5	21
tree	5	23
truck	5	22

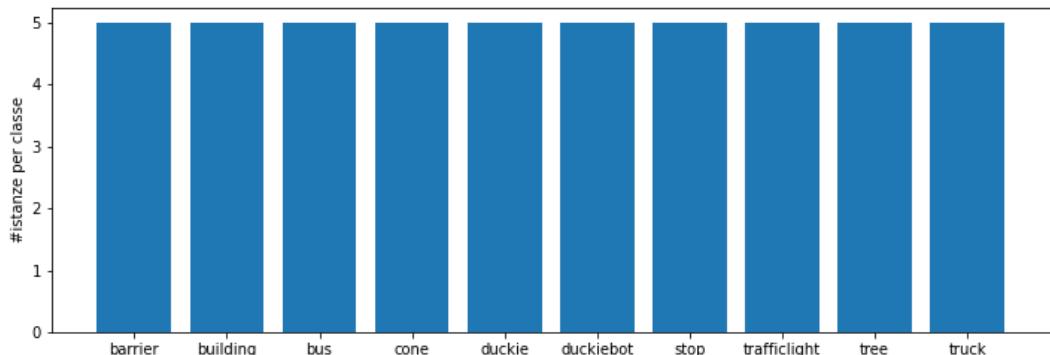


Figura 1.1. Grafico a barre della distribuzione delle classi del training set del dataset simulato.

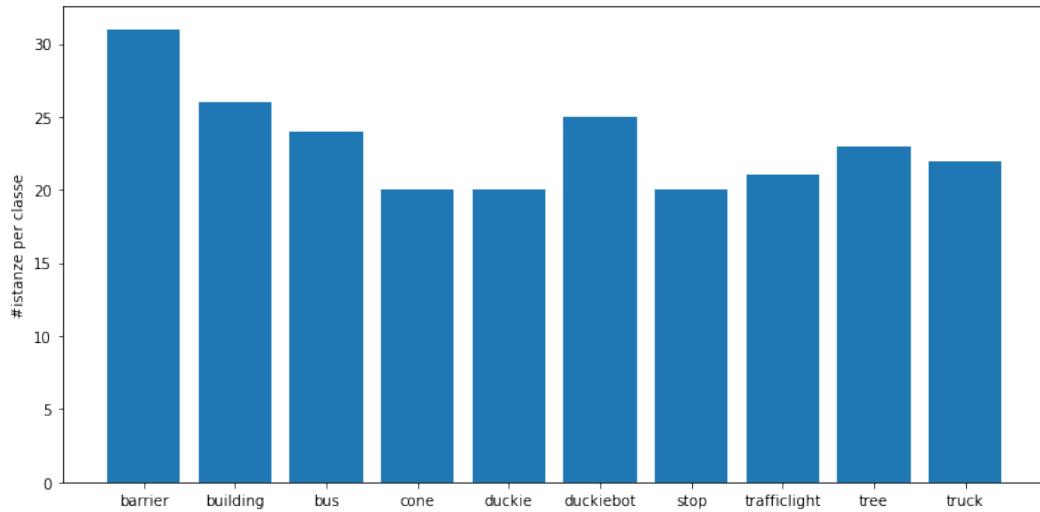


Figura 1.2. Grafico a barre della distribuzione delle classi del validation set del dataset simulato.

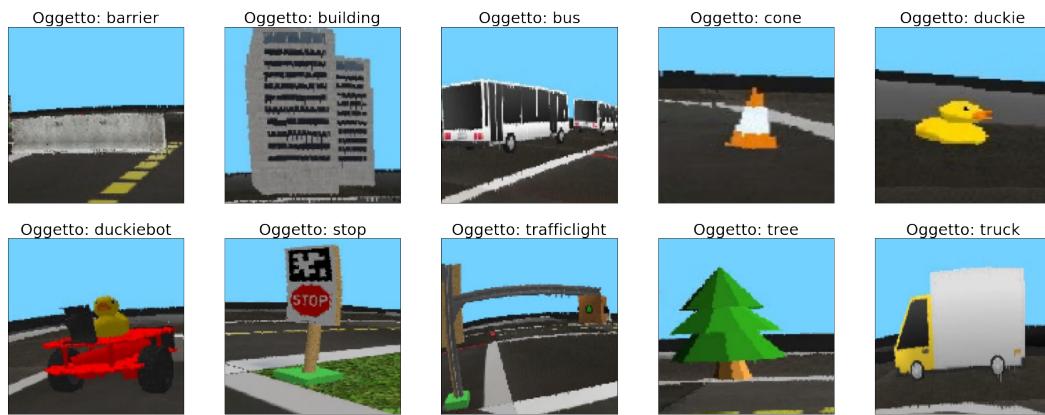


Figura 1.3. Esempi di immagini del dataset simulato.

Per effettuare alcuni test, che verranno presentati nei prossimi capitoli, al dataset appena visto sono state aggiunte due ulteriori classi, grass e ground. In tabella 1.2 è possibile vedere che sono bastati pochissimi samples di training per tali classi e in figura 1.5 viene mostrato un esempio per ciascuna di esse.

Tabella 1.2. Numero istanze di training e test per le classi ground e sky.

Classe	Numero istanze di training	Numero istanze di test
ground	2	5
sky	1	3

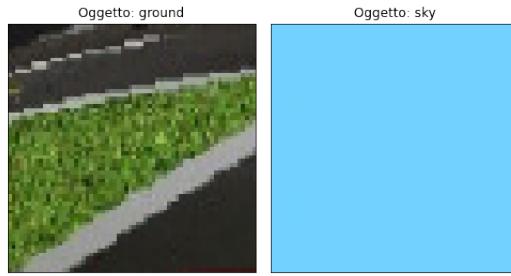


Figura 1.4. Esempi di immagini per le classi ground e sky del dataset simulato.

1.2.2 Dataset reale

Il dataset reale è costituito da 205 immagini, appartenenti a 7 classi, alcune delle quali non presenti nel dataset simulato. La tabella 1.3 mostra come le immagini siano ripartite tra training e test set, mentre le figure 1.5 e 1.6 mostrano come esse risultino distribuite tra le diverse classi. In figura 1.7, invece, è possibile vedere un esempio di immagine per ciascuna classe.

Tabella 1.3. Numero di istanze di training e test per classe nel dataset reale.

Classe	Numero istanze di training	Numero istanze di test
building	5	27
crossing	5	37
duckie	5	18
duckiebot	5	36
parking	3	9
stop	5	39
trafficlight	4	27

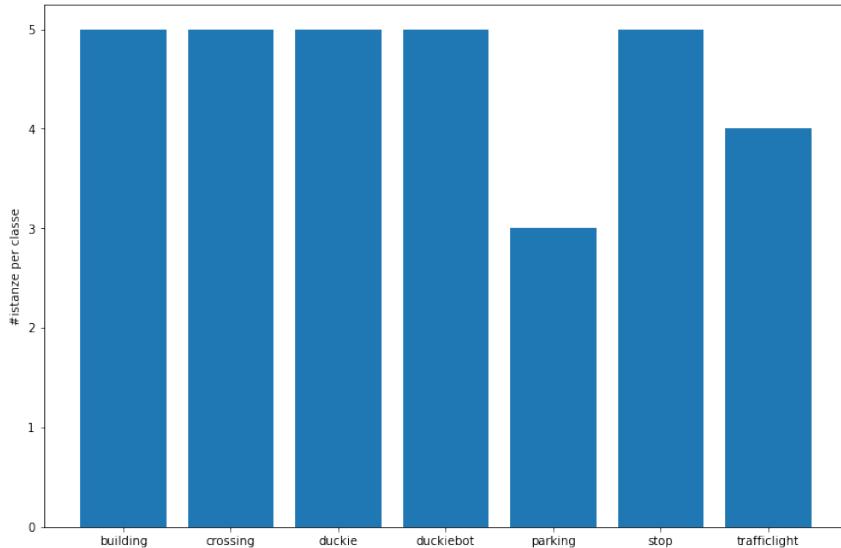


Figura 1.5. Grafico a barre della distribuzione delle classi del training set del dataset reale.

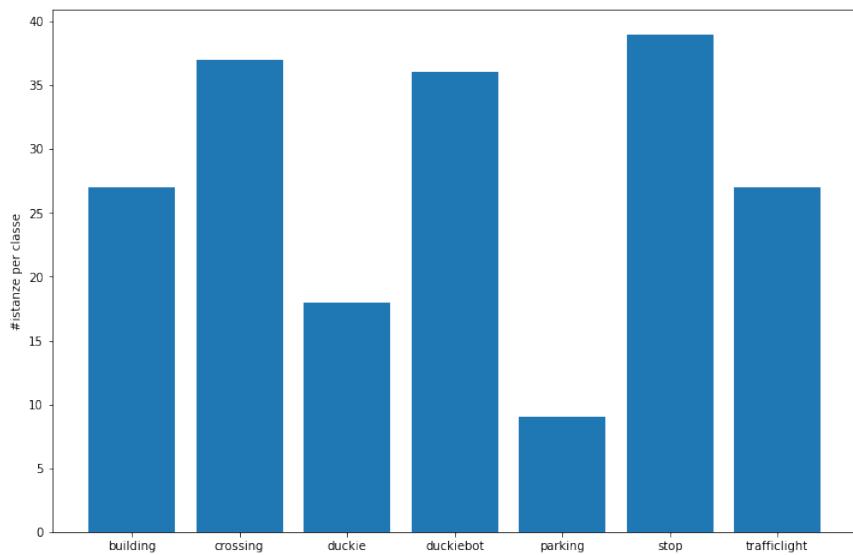


Figura 1.6. Grafico a barre della distribuzione delle classi del validation set del dataset reale.

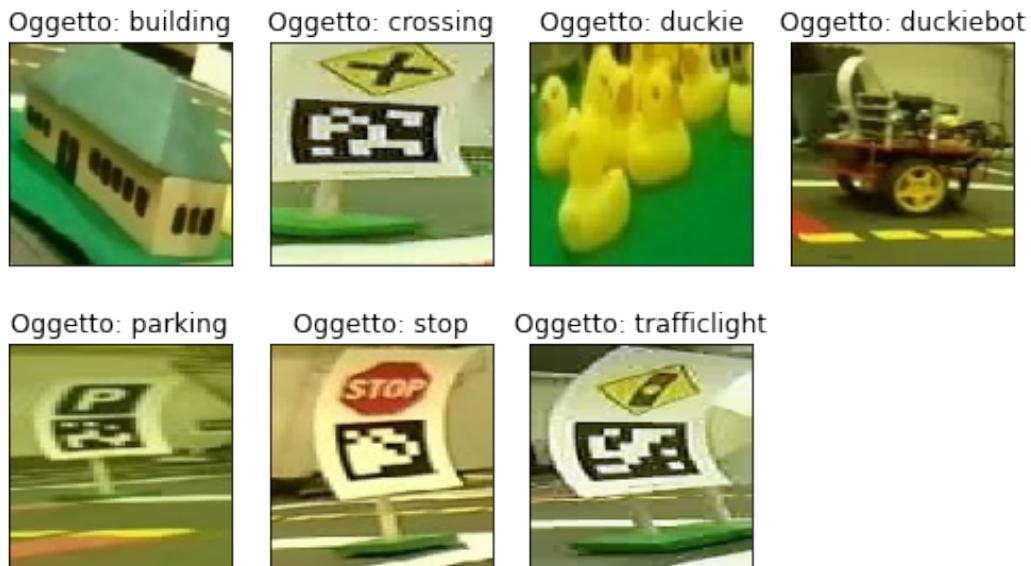


Figura 1.7. Esempi di immagini del dataset reale.

Come si può notare da questi pochi esempi riportati, le immagini del dataset reale sono molto sgranate e tutte tendenti verso il giallo, il che può aver influito anche sui risultati ottenuti.

1.2.3 Sottoinsieme del dataset reale

Dal dataset reale è stato ricavato un ulteriore dataset, costituito da 143 immagini totali di dimensioni diverse, appartenenti a un sottoinsieme delle classi del dataset simulato. La distribuzione dei dati tra le varie classi è mostrata nella tabella 1.4 e nella figura 1.8, mentre in figura 1.9 è riportato un sample per ogni classe.

Tabella 1.4. Numero di istanze per classe nel dataset reale.

Classe	Numero di istanze
building	33
duckie	23
duckiebot	42
stop	45

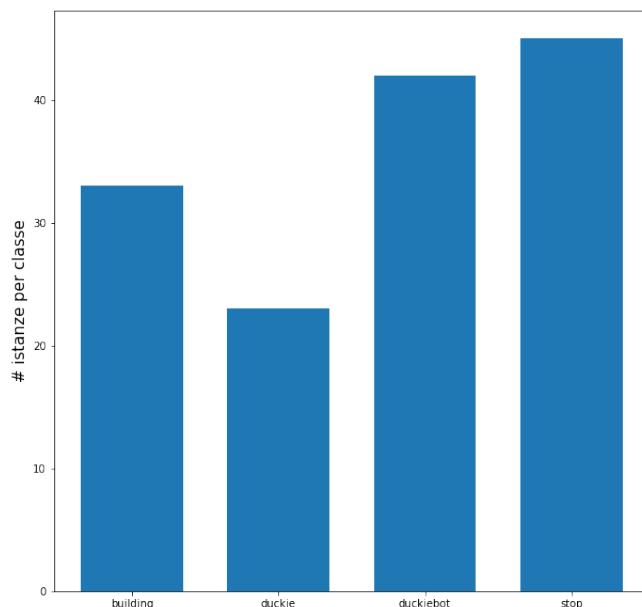


Figura 1.8. Grafico a barre della distribuzione delle classi nel dataset reale.

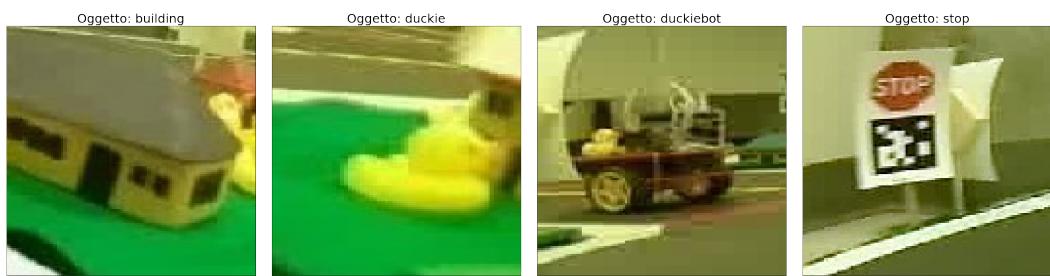


Figura 1.9. Esempi di immagini del sottoinsieme del dataset reale.

Anche in questo caso, sono stati fatti dei test nei quali sono state aggiunte 2 ulteriori classi: road e wall. La tabella 1.5 mostra il numero di samples utilizzati per ciascuna di esse, mentre in figura 1.10 vengono riportati esempi di immagini.

Tabella 1.5. Numero di istanze per le classi road e wall del dataset reale.

Classe	Numero istanze di training	Numero istanze di test
road	3	12
wall	3	10

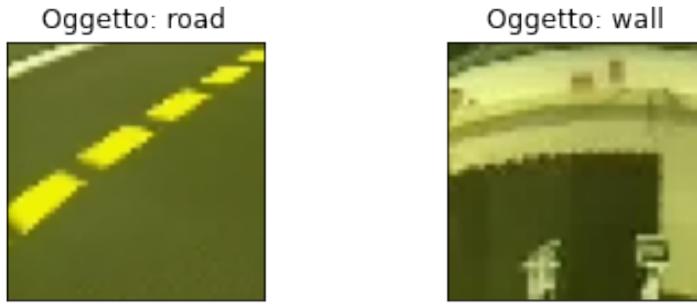


Figura 1.10. Esempi di immagini delle classi road e wall del dataset reale.

1.3 Preprocessing

Sia per il dataset simulato che per quello reale è stato effettuato il rescaling delle immagini, per velocizzare l'addestramento e migliorare le performance del modello. Per quanto riguarda il dataset simulato, è stato effettuato il ridimensionamento a 32*32, per ridurre al minimo il tempo impiegato per la classificazione. Tuttavia, con l'aggiunta delle classi ground e sky è necessario aumentare la risoluzione delle immagini a 64*64, altrimenti si ha un'eccessiva riduzione dell'accuratezza dell'algoritmo. Per quanto riguarda le immagini del dataset reale, la massima risoluzione utilizzabile risulta essere 48*48, poiché altrimenti la dimensione finale di alcune immagini risulterebbe essere maggiore di quella di partenza, causando dei problemi durante l'addestramento. Allo stesso tempo, non è possibile ridurre la risoluzione a 32*32, nel tentativo di velocizzare la classificazione, in quanto le immagini diventano ancor più sgranate, portando a pessimi valori di accuracy. Sono state fatte delle prove in cui si è utilizzata la tecnica del data augmentation, con l'introduzione del flipping orizzontale e della rotazione delle immagini entro un range di 30 gradi, ma questo non ha portato a un miglioramento significativo dei risultati.

Capitolo 2

Metodologia

2.1 Classificazione

Nel machine learning, si parla di apprendimento supervisionato quando ai dati che vengono utilizzati per addestrare l'algoritmo, detti dati di training, sono associate le relative label. La classificazione è uno dei problemi dell'apprendimento supervisionato: avendo in input un certo dato x_i , il modello deve restituire in output un valore intero, rappresentante la classe a cui esso ritiene che x_i appartenga.

2.1.1 Multiclass classification

Il problema che viene affrontato in questo progetto è di multiclass classification, in quanto vogliamo assegnare ciascuna istanza a una sola tra più di due classi. In questo caso si ha in output un vettore i cui elementi rappresentano la probabilità che l'istanza in input appartenga a ciascuna delle classi considerate e l'elemento maggiore rappresenta la classe predetta dal classificatore.

2.2 Reti neurali

Le reti neurali costituiscono una sotto-branca del machine learning che nasce dall'idea di riprodurre il funzionamento del cervello umano, sfruttando le potenzialità dei processori e delle schede grafiche dei computer. La struttura distribuita e sequenziale del cervello umano, nel quale le varie aree che lo costituiscono processano le informazioni in ingresso svolgendo funzioni diverse, viene riprodotta nelle reti neurali tramite una struttura multilayer.

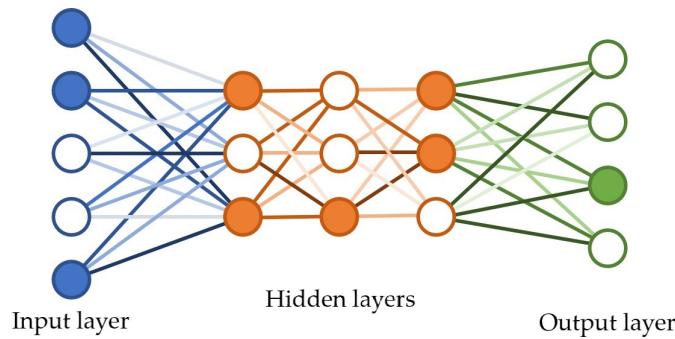


Figura 2.1. Struttura di una rete neurale.

Come è mostrato in figura 2.1, si ha un input layer, costituito dai nodi ai quali giunge l'informazione di input, un output layer, che serve a computare la loss e quindi il risultato finale, e una serie di hidden layers intermedi, definiti “nascosti” perché nel corso dell’addestramento non è possibile conoscerne il valore. L’unità di base di una rete neurale viene definita percettrone ed ha una struttura simile a quella del neurone. Il percettrone riceve una serie di input $x_1, x_2 \dots x_n$, dei quali viene fatta una somma pesata tramite il vettore dei pesi $w_1, w_2 \dots w_n$, alla quale viene aggiunta una componente fissa, detta bias. Dopodiché, viene applicata al valore così ottenuto una funzione, chiamata funzione di attivazione, che fornisce l’output del percettrone, che costituirà a sua volta l’input dei percettroni del livello successivo oppure, nel caso dell’output layer, servirà per il calcolo della loss.

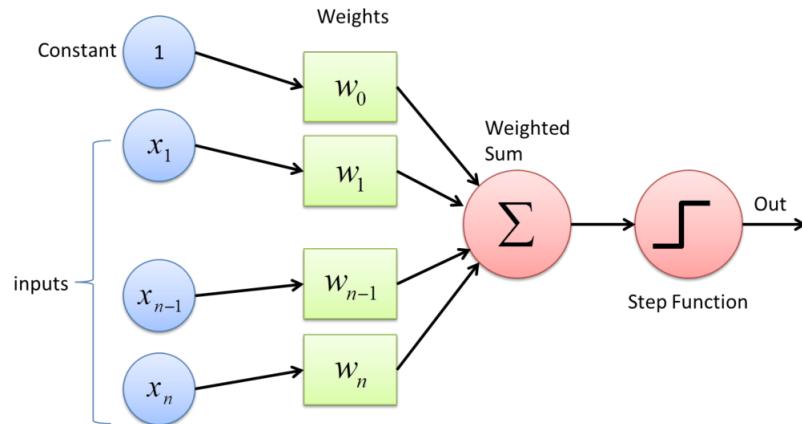


Figura 2.2. Struttura di un percettrone.

Chiaramente, l’errore che calcoliamo alla fine non dipenderà solamente dai nodi dell’ultimo livello della rete, in quanto a tale errore avranno contribuito anche tutti i nodi dei livelli precedenti, o comunque buona parte di essi nel caso in cui la rete non sia fully connected. Per poter aggiornare i pesi in modo da ridurre l’errore commesso dalla rete e continuare l’addestramento, è necessario capire quanto ciascun nodo abbia contribuito all’errore stesso. Si ha quindi una fase di retropropagazione dell’errore, durante la quale si procede dall’output layer verso l’input layer, calcolando di volta in volta la derivata parziale della loss e l’aggiornamento per ciascun peso.

2.2.1 Reti neurali convoluzionali

Le reti convoluzionali sono molto utilizzate nell’ambito del riconoscimento delle immagini. A differenza di quanto avviene nelle reti neurali non convoluzionali, ogni nodo applica al dato in input un filtro, perciò l’output di ciascun layer sarà un blocco tridimensionale. Nel caso delle immagini, ciascun nodo fornirà in output una nuova immagine, di dimensione più piccola rispetto a quella di partenza, perciò quello che si fa è aggiungere del padding all’immagine originale, in modo da poter applicare il filtro anche sui pixel sul bordo. Si utilizza la convoluzione perché attraverso di essa è possibile analizzare l’immagine considerando degli intorni, permettendo così di individuare delle caratteristiche dell’immagine non intuitibili effettuando un’analisi pixel a pixel. Di solito, la convoluzione risulta essere seguita da una funzione di attivazione e da un pooling.

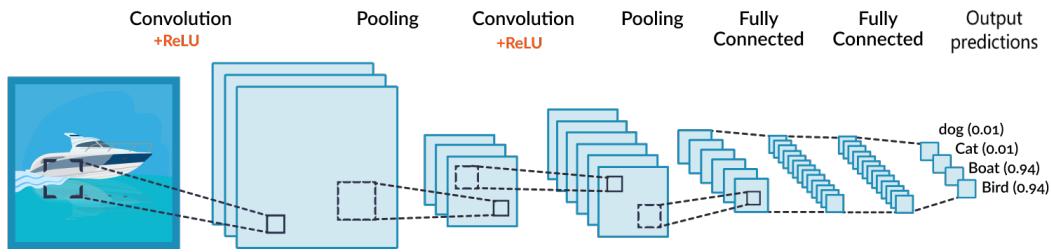


Figura 2.3. Struttura di una rete convoluzionale.

2.2.2 Funzioni di attivazione

Come detto in precedenza, alla fine di ogni percepitrone si ha una funzione di attivazione che viene applicata sul suo output. Essa ha un ruolo fondamentale nel modo in cui la rete apprende, perché permette di filtrare l'informazione, mantenendo le parti importanti e trascurando il rumore presente all'interno di ogni tipo di dato. La funzione di attivazione permette dunque di aumentare la potenza del nostro algoritmo, ma allo stesso tempo è necessario individuare la tipologia più adatta al problema affrontato. Nell'ambito di questo progetto sono stati utilizzati due tipi di funzione di attivazione:

- La ReLU (Rectified Linear Unit), che effettua una partizione sull'asse x , in quanto se l'output è minore di 0, l'informazione viene completamente ignorata, mentre se è maggiore di 0, viene restituito il valore così com'è. Il vantaggio della ReLU è che, avendo una derivata molto semplice, permette di effettuare i calcoli in maniera molto rapida.

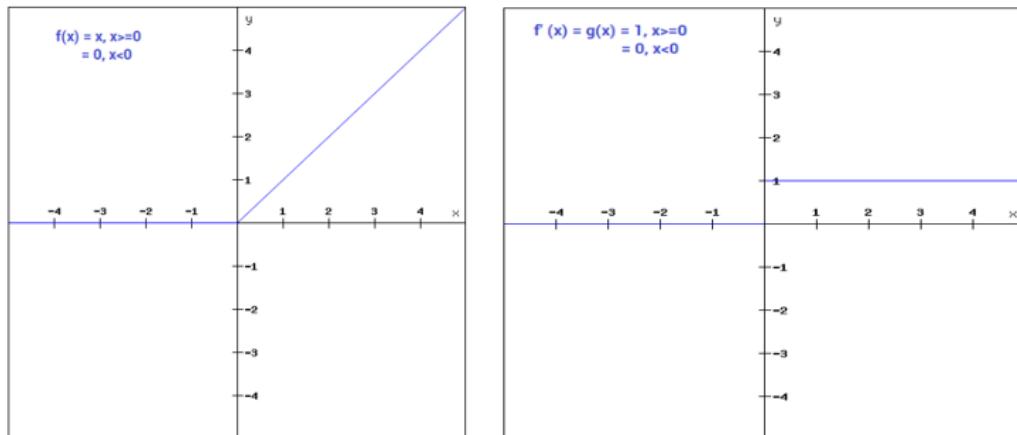


Figura 2.4. Funzione di attivazione ReLU.

- La funzione Softmax, utilizzata nell'output layer, che fa in modo che tutti gli elementi del vettore di output siano normalizzati tra 0 e 1 e che la loro somma faccia 1. L'algoritmo predirà la classe a cui è associata la probabilità più alta. Essa assume la seguente forma:

$$f(\hat{y}_i) = \frac{e^{\hat{y}_i}}{\sum_{j=0}^m e^{\hat{y}_j}}$$

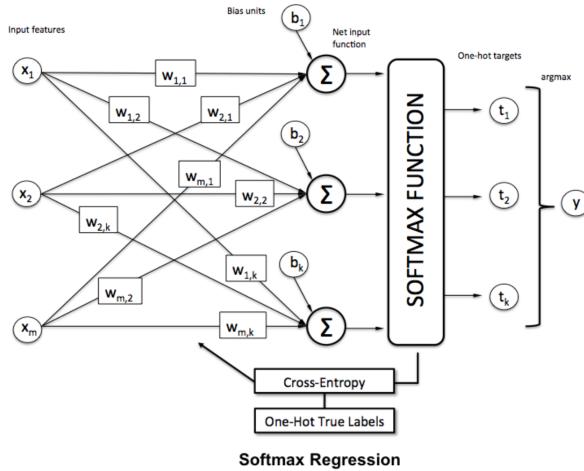


Figura 2.5. Utilizzo della softmax sull'output layer di una rete convoluzionale.

2.2.3 Max Pooling

La funzione del pooling è di ridurre progressivamente la dimensione dello spazio delle features, in modo tale da diminuire il numero di parametri da apprendere e quindi la quantità di calcoli che la rete deve effettuare. La più comune forma di pooling è il max pooling, nel quale a ciascuna immagine viene applicato un filtro che mantiene solo l'elemento maggiore di ogni area considerata.

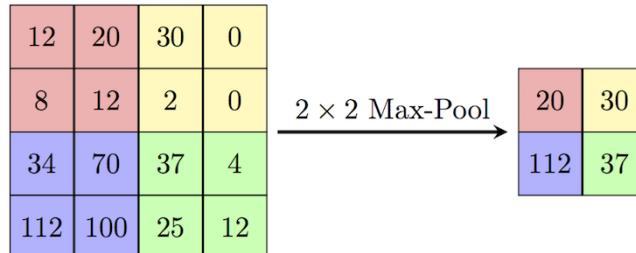


Figura 2.6. Esempio di applicazione del max pooling.

2.2.4 Funzione di costo

La funzione di costo utilizzata è la categorical cross-entropy. La cross-entropy loss misura le performance di un modello di classificazione, il cui output è una probabilità, il cui valore sarà quindi compreso tra 0 e 1. Essa è definita come segue:

$$CE = - \sum_{i=0}^m y_i \log \hat{y}_i$$

dove y_i è il groundtruth, \hat{y}_i è la predizione del modello e m è il numero di classi. Come detto in precedenza, prima di dare il risultato del modello in input alla loss function, a tale valore viene applicata un'activation function, perciò potremmo scrivere $f(\hat{y}_i)$. Se la funzione di attivazione utilizzata è la Softmax, parleremo di categorical cross-entropy. Nel caso di multiclass classification, che è quello qui considerato, solamente un elemento nel vettore y , corrispondente alla classe a cui

l'istanza considerata appartiene, sarà diverso da 0, perciò solo tale classe manterrà il proprio termine nella funzione di costo. Dunque, partendo dalla formula della softmax, definiamo la categorical cross-entropy come segue:

$$f(\hat{y}_i) = \frac{e^{\hat{y}_i}}{\sum_{j=0}^m e^{\hat{y}_j}} \Rightarrow CE = -\log \frac{e^{\hat{y}_i}}{\sum_{j=0}^m e^{\hat{y}_j}}$$

2.2.5 Metodi di regolarizzazione

La regolarizzazione è fondamentale nel machine learning, perché permette di evitare che la rete vada in overfitting. Molto spesso, infatti, se la rete che si sta utilizzando è troppo complessa o il dataset è troppo piccolo, accade che la rete si fossilizzi sui dati di training e non sia più in grado di riconoscere nuove immagini che gli vengano sottoposte, cioè non è più in grado di generalizzare. La regolarizzazione permette di modificare la matrice dei pesi, portando a una riduzione della funzione di costo. In questo progetto sono state utilizzate prevalentemente le seguenti tecniche di regolarizzazione:

- Il dropout, che consiste nell'ignorare un sottoinsieme dei nodi della rete scelti randomicamente durante la fase di training. Più precisamente, si definisce un valore p , che indica la probabilità che ad ogni iterazione il singolo nodo venga mantenuto, e di conseguenza $1-p$ sarà la probabilità che esso non venga considerato.
- L'early stopping, che consiste nel monitorare le performance del modello sul validation set ad ogni epoca e nel terminare l'addestramento in funzione delle prestazioni sul validation set. Infatti, procedendo con l'addestramento, la loss per il training set decresce, in quanto la rete sta apprendendo, mentre la loss per il validation set inizialmente decresce, ma poi torna ad aumentare quando la rete inizia a fare overfitting sui dati di training.

2.2.6 Adam

L'Adam (adaptive momentum estimation) è un metodo di ottimizzazione che sfrutta i vantaggi sia del momento che del RMSProp, permettendo di arrivare velocemente all'ottimo smorzando contemporaneamente le oscillazioni.

$$\begin{aligned} v_t &= \beta_1 * v_{t-1} - (1 - \beta_1) * g_t \\ s_t &= \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2 \\ \Delta w_t &= -\eta * \frac{v_t}{\sqrt{s_t}} * g_t \\ w_{t+1} &= w_t + \Delta w_t \end{aligned}$$

La prima equazione fa riferimento al metodo più legato al momento, mentre la seconda fa più riferimento al RMSProp, perciò mediante gli iperparametri β_1 e β_2 , possiamo stabilire quale dei due metodi usare maggiormente. Come vediamo, nell'aggiornare i pesi, il learning rate è moltiplicato per un termine legato a quanto appena detto: il termine del RMSProp tende a smorzare il learning rate, mentre il momento tende ad aumentare il learning rate e questo ci permette di arrivare più velocemente verso l'ottimo, ma di muoverci più lentamente una volta arrivati vicino a esso.

2.3 Few-shot learning

Come si evince dal nome, per few-shot learning si intende la pratica di fornire a un modello di apprendimento una piccola quantità di dati di training. Questa tecnica è molto utile quando si hanno a disposizione pochi dati con label, in quanto anche nel caso in cui si abbiano molti dati a disposizione, il labeling potrebbe risultare molto costoso. Esistono due principali approcci ai problemi di few-shot learning:

- Data-level approach: si tratta di un approccio basato sull'idea che ogni volta che non si hanno dati a sufficienza è necessario aggiungerne altri. Una tecnica molto comune è quella di utilizzare sorgenti di dati esterne, ma come detto precedentemente non si ha nessun altro dataset riguardante Duckietown a disposizione. Un'altra tecnica consiste nel produrre nuovi dati, ad esempio facendo data augmentation, ma lo scopo di questo progetto è proprio quello di limitarsi all'utilizzo di poche immagini riuscendo però ad ottenere risultati ottimi.
- Parameter-level approach: si tratta di un approccio che tenta di evitare problemi di overfitting limitando lo spazio dei parametri. Per fare questo, si utilizzano le tecniche di regolarizzazione e le funzioni di costo. Tale metodo è quello utilizzato nell'ambito di questo progetto.

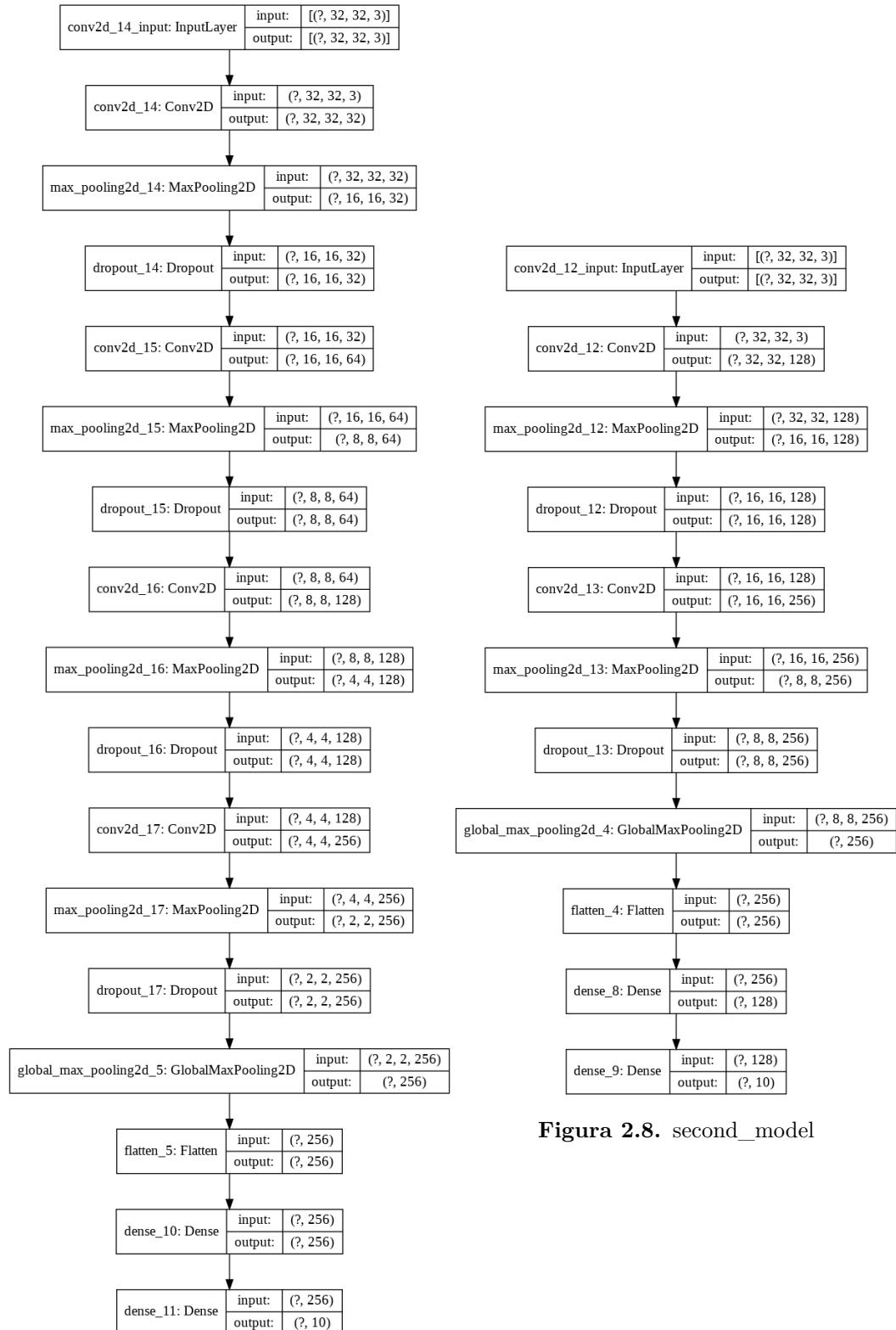
2.4 Descrizione della rete utilizzata

I primi esperimenti sono stati effettuati sul dataset simulato con l'obiettivo di trovare un modello che riuscisse a classificare correttamente la maggior parte delle istanze di test, ma che fosse contemporaneamente il più leggero possibile e quindi più veloce nell'effettuare la classificazione. Si è partiti da un modello che chiameremo initial_model, avente la struttura in figura 2.7. Esso risulta costituito da 4 livelli convoluzionali formati rispettivamente da 32, 64, 128 e 256 nodi, che applicano alle immagini dei filtri 3*3 e utilizzano come funzione di attivazione la ReLU. Ciascuno di essi è seguito da un livello che effettua il max pooling e si utilizza inoltre il dropout come metodo di regolarizzazione. Infine, si ha un Dense 256 con funzione di attivazione ReLU e un Dense 10 con funzione di attivazione Softmax, dal momento che è stata scelta come funzione di costo la categorical cross-entropy loss, poiché indicata per problemi di multiclass classification. Di esperimento in esperimento, si è tentato di ridurre il numero di livelli e il numero di nodi all'interno di ciascun livello. Si è passati dunque alla rete avente la struttura mostrata in figura 2.8, che chiameremo second_model, nella quale si hanno 2 soli livelli convoluzionali di 128 e 256 nodi. Modificando la rete in vari modi e vedendo come variano di volta in volta i risultati, si è giunti infine alla rete avente la struttura in figura 2.9, che chiameremo final_model, nella quale i 2 livelli convoluzionali hanno 64 e 128 nodi e il penultimo Dense ha 128 nodi e non più 256. Inoltre, il numero di epoch è stato fissato a 150 e il batch_size a 5, è stato impostato il parametro β_1 dell'Adam a 0.5, per fare in modo che il learning rate non crescesse troppo rapidamente, ed è stato implementato l'early stopping con patience pari a 10, per evitare che il modello andasse in overfitting, monitorando la validation loss e controllando che questa continuasse a diminuire nel corso del training. Dopo essere riusciti a ottenere risultati molto buoni sul test set, sia per quanto riguarda l'accuracy che i tempi necessari per la classificazione, si è pensato a un modo per rendere questo algoritmo utilizzabile nell'ambito di Duckietown. Infatti, in Duckietown non si hanno immagini contenenti un solo oggetto di interesse, ma le immagini riprese dalla telecamera

presente sul Duckiebot, che inquadra la strada di fronte a sé. Sono state quindi utilizzate immagini contenenti più oggetti di interesse, effettuando su di esse una divisione in più parti e successivamente classificando ciascuna patch attraverso la rete precedentemente addestrata. Dato che in alcune parti di immagini non è presente alcun oggetto, per migliorare i risultati in quest'ultimo tipo di esperimento, sono state aggiunte le due ulteriori classi, grass e sky, presentate nel capitolo precedente.

2.4.1 Adattamento del modello al dataset reale

Come mostrato nel capitolo precedente, il dataset reale presenta classi diverse rispetto a quelle del dataset simulato, perciò sono stati fatti diversi tentativi. Inizialmente, è stata effettuata direttamente la classificazione sul dataset reale ottenuto considerando solo le 4 classi presenti anche nel dataset simulato, utilizzando la rete precedentemente addestrata sul dataset simulato. Come secondo tentativo, è stata addestrata ulteriormente la rete utilizzando un piccolo training set ricavato dal sottoinsieme del dataset reale, arrivando ad avere risultati sul test set molto simili a quelli avuti sul dataset simulato. In questo caso, è stato fatto uno studio sul numero di immagini di training del dataset reale da utilizzare per ottenere le migliori performance. Come ultimo tentativo, è stata addestrata la rete utilizzando il dataset reale, considerando tutte e 7 le classi, ma i risultati non sono stati entusiasmanti. Infine, utilizzando il secondo metodo, che prevede due addestramenti della rete, prima attraverso il dataset simulato e poi attraverso il dataset reale, è stata effettuata la classificazione delle patches di immagini contenenti più oggetti, precedentemente suddivise in più parti. Anche in questo caso, per rendere tali esperimenti più consistenti, sono state aggiunte le due classi, road e wall.

**Figura 2.7.** initial_model**Figura 2.8.** second_model

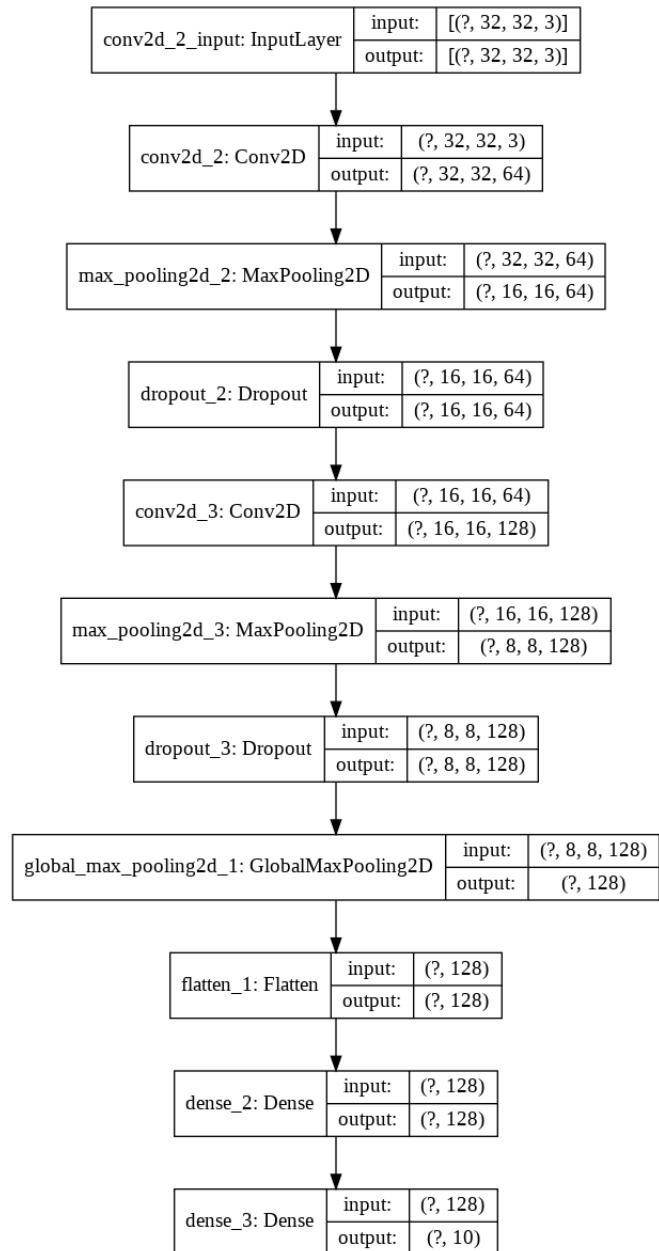


Figura 2.9. Struttura della rete utilizzata (final_model).

Capitolo 3

Risultati

3.1 Esperimenti sul dataset simulato

La tabella 3.1 mostra i risultati ottenuti sul test set negli esperimenti più significativi, effettuati utilizzando i diversi modelli presentati nel capitolo precedente e apportandovi delle modifiche.

Tabella 3.1. Risultati degli esperimenti con modelli diversi sul dataset simulato.

Modello	Test loss	Test accuracy	Class. corrette	Class. errate
Test 1	0.880	0.751	175	58
Test 2	0.848	0.816	190	43
Test 3	0.436	0.897	209	24
Test 4	0.309	0.919	214	19
Test 5	0.336	0.919	214	19
Test 6	0.381	0.931	217	16
Test 7	0.183	0.944	220	13
Test 8	0.193	0.949	221	12

- Test 1: initial_model.
- Test 2: initial_model con dropout impostato a 0.2.
- Test 3: initial_model con dropout pari a 0.2 e parametro β_1 dell'Adam pari a 0.5.
- Test 4: second_model con penultimo livello Dense di 256 nodi, dropout pari a 0.2 e parametro β_1 dell'Adam pari a 0.5.
- Test 5: second_model con dropout pari a 0.2 e parametro β_1 dell'Adam pari a 0.5.
- Test 6: second_model con parametro β_1 dell'Adam pari a 0.5.
- Test 7: second_model con parametro β_1 dell'Adam pari a 0.5, early stopping, aumento del numero di epoche a 150 e diminuzione del batch_size a 5.
- Test 8: final_model con parametro β_1 dell'Adam pari a 0.5, early stopping, numero di epoche pari a 150 e batch_size pari a 5.

L'initial_model è costituito da ben 4 livelli convoluzionali, più un Dense di 256 nodi e un ultimo Dense 10, perciò pur riuscendo a classificare correttamente circa il 75% delle immagini di test, risulta essere piuttosto pesante e lento nella classificazione. Già con il second_model si è ottenuto un grande miglioramento non solo nell'accuratezza, ma anche nei tempi, mentre il final_model ha permesso di raggiungere circa il 95% di accuracy e di ottenere tempi brevissimi nella classificazione sul test set. La tabella 3.2 mostra i tempi impiegati dal final_model per effettuare la classificazione sull'intero test set e su una singola immagine, utilizzando la CPU e la GPU.

Tabella 3.2. Tempi impiegati dal final_model per la classificazione.

	Test set	Sample
CPU	0.447	0.054
GPU	0.045	0.045

In figura 3.1 sono mostrati gli andamenti dell'accuracy e della loss sul training e sul validation set in funzione delle epoche.

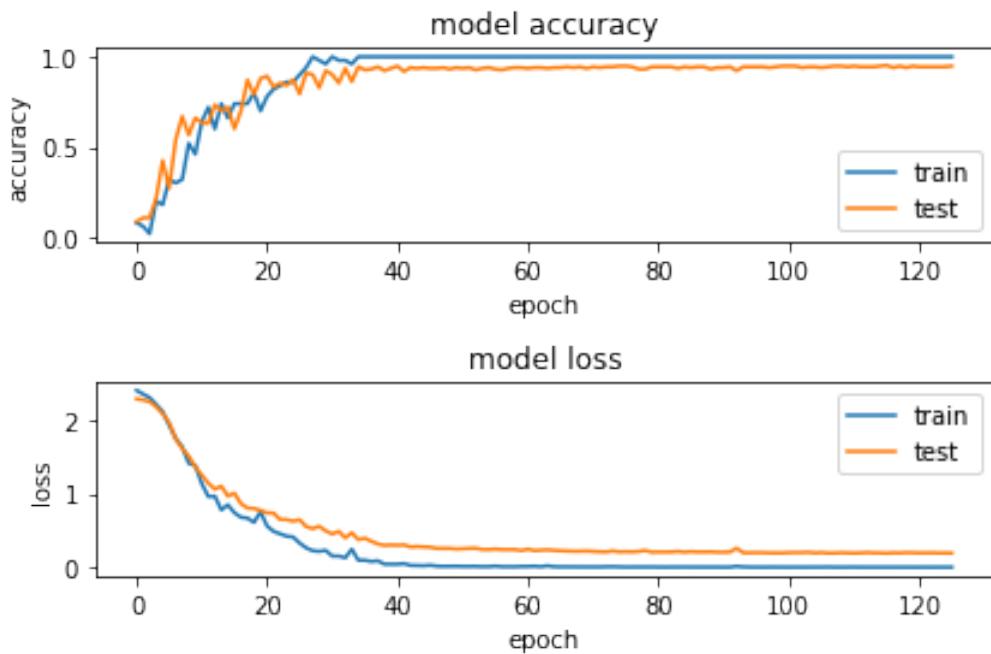


Figura 3.1. Andamento dell'accuracy e della loss per il training e il validation set in funzione delle epoche.

In figura 3.2, sono riportate le immagini classificate erroneamente dalla rete. Esse risultano appartenere alle classi barrier(0), bus(2), cone(3), stop(6) e tree(8). Per quanto riguarda la classe barrier, 3 samples vengono classificati come truck, probabilmente perché le features relative alle linee gialle della strada vengono considerate come rilevanti nell'immagine e non come parte del background, mentre 2 samples vengono classificati come building, perché il rumore del simulatore fa sì che si abbiano delle macchie scure sulle barriere, che vengono scambiate per le finestre di un edificio. Gli errori riguardo la classe tree vengono fatti su immagini in cui

l'albero risulta essere più in ombra e quindi i pixels sono più scuri e si confondono con lo sfondo, portando l'algoritmo a classificarli come barriere. Nelle immagini della classe trafficlight, i semafori risultano essere più distanti e si ha una grande porzione di sfondo, che viene però presa in considerazione nella classificazione, portando l'algoritmo a predire la classe barrier. Per quanto riguarda gli errori relativi al cono, allo stop e al bus, di nuovo si ha che essi sono probabilmente dovuti al fatto che gli oggetti risultano essere più lontani e la presenza del rumore tende a renderli molto sgranati, portando l'algoritmo a sbagliare.

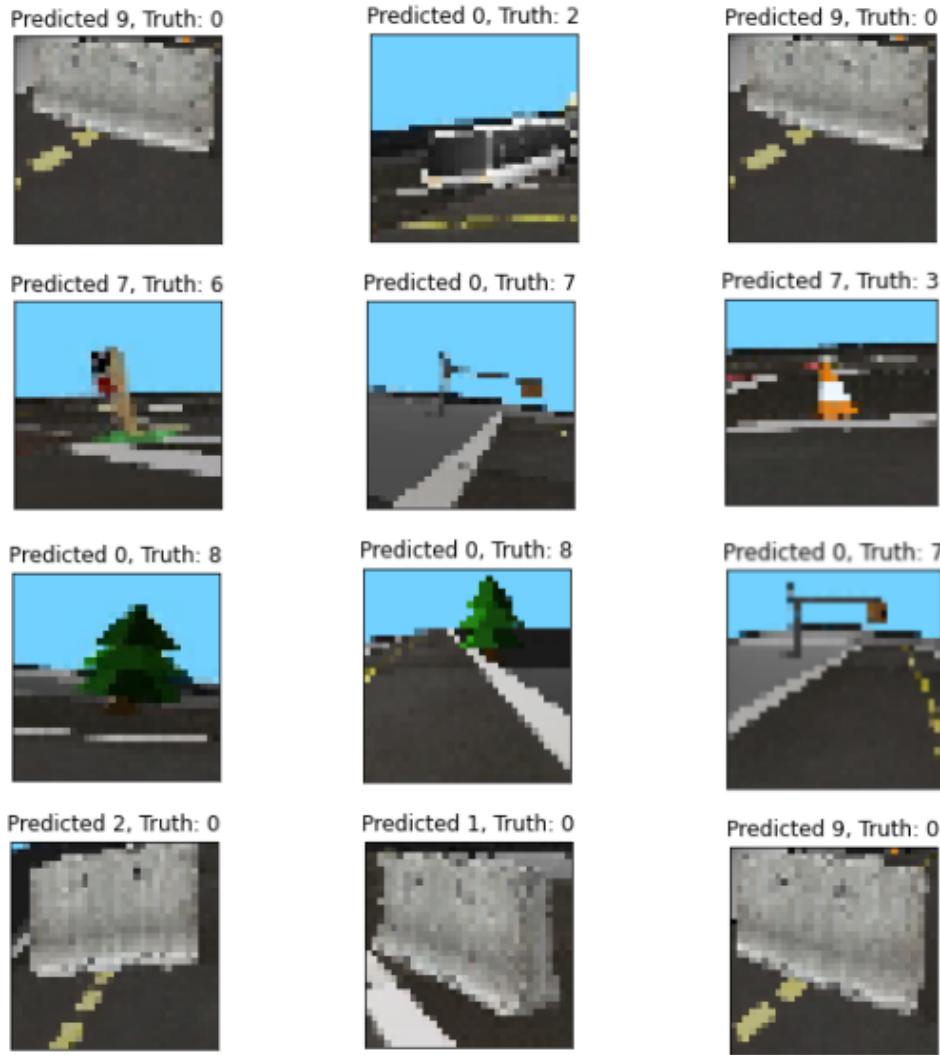


Figura 3.2. Classificazioni errate.

Per tentare di ridurre ulteriormente il numero di errori commessi, sono stati fatti ulteriori test aumentando il numero di immagini di training per le varie classi, ma questo non ha portato ad alcun miglioramento nei risultati.

3.1.1 Esempio di applicazione in Duckietown

Dal momento che in Duckietown non si hanno immagini contenenti singoli oggetti, ma si ha la visione complessiva della strada di fronte a sé, sono stati fatti dei test finali, prendendo delle immagini così come erano nel simulatore, dividendole in 6 parti 64×64 e classificando ciascuna di esse. Negli esempi riportati in figura 3.3, l'algoritmo riesce a classificare correttamente quasi tutte le parti delle varie immagini, facendo pochi errori. In particolare, l'aspetto positivo è che se in una parte d'immagine sono presenti più oggetti di interesse, come ad esempio nel caso delle sezioni centrali o quella in basso a destra dell'ultimo esempio, l'algoritmo predice l'oggetto che si trova più vicino tra quelli presenti, ovvero stop e trafficlight.

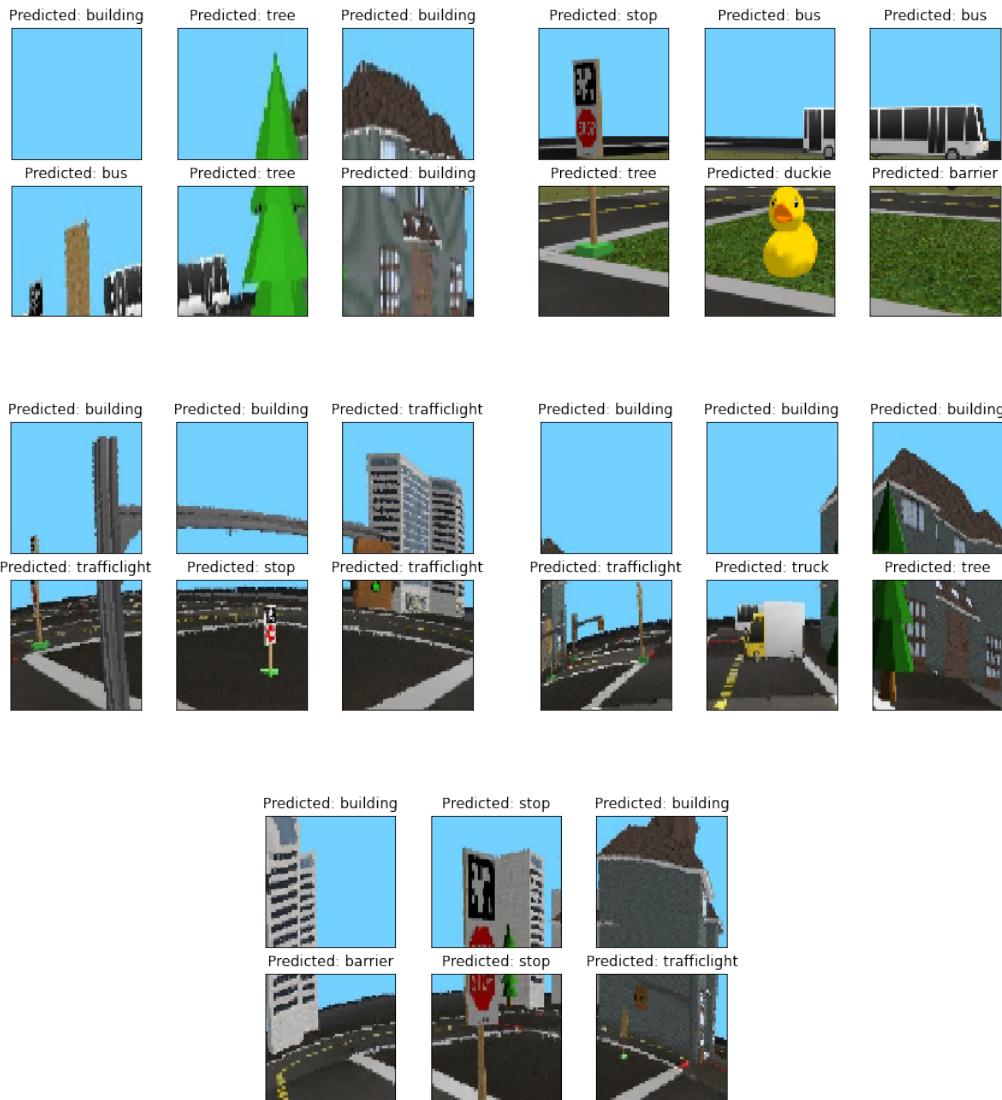


Figura 3.3. Esempi di applicazione della rete su immagini contenenti più oggetti.

Il tempo impiegato per la classificazione di tutte e sei le parti delle singole immagini, utilizzando la CPU, è di 11.68 secondi. Supponendo di voler classificare solo gli oggetti che si trovano davanti a noi, cioè nel riquadro in basso al centro, per

evitare di andarvi a sbattere, o nel riquadro in basso a destra, nel caso di un cartello o un semaforo, si riducono ulteriormente i tempi, in quanto l'algoritmo impiega soli 5.584 secondi per la classificazione delle 2 patches. Abbassando la risoluzione delle immagini a 32*32, l'algoritmo necessita di 3.40 secondi per classificare tutte e 6 le parti e 1.885 per classificare solo le due in basso al centro e a destra. Tuttavia, in questo caso l'algoritmo tende a classificare in maniera errata le parti in alto dell'immagine perché, abbassando la risoluzione, gli oggetti più lontani risultano ancora più sgranati. Se si intende effettuare esclusivamente la classificazione delle patches in basso, però, impostare la risoluzione a 32*32 risulta essere un metodo efficace per ridurre ulteriormente i tempi.

Tabella 3.3. Tempi impiegati per la classificazione nel test finale.

Risoluzione	6 parti	2 parti
64*64	11.68	5.584
32*32	3.40	1.885

3.1.2 Aggiunta delle classi ground e sky

Come appena visto, dal momento che la nostra rete neurale fornisce in output un vettore di probabilità e a ciascun dato viene associata la classe più probabile, anche se alcune patches delle immagini in figura 3.3 non contengono alcun oggetto di interesse viene loro associata una delle 10 classi considerate. In questo modo, il nostro algoritmo predrà building dove invece c'è solamente il cielo, o barrier dove c'è semplicemente un prato. Per poter risolvere questo problema, sono state aggiunte due ulteriori classi, ground e sky, ed è stata addestrata nuovamente la rete. I risultati ottenuti sono mostrati in tabella 3.4.

Tabella 3.4. Risultati sul dataset simulato con l'aggiunta delle classi ground e sky.

Risoluzione	Test loss	Test accuracy	Class. corrette	Class. errate
32*32	0.6759	0.8155	198	43
64*64	0.1869	0.9270	224	17

Impostando la risoluzione delle immagini a 64*64, i livelli di accuratezza sono simili a quelli ottenuti precedentemente e i tempi necessari per la classificazione, riportati in tabella 3.5, pur essendo peggiori di quelli riportati nelle tabelle 3.2 e 3.3, sono comunque accettabili.

Tabella 3.5. Tempi necessari per la classificazione con l'aggiunta delle classi ground e sky.

	Test set	Sample	Applicazione finale
CPU	1.5369	0.0578	12.1994
GPU	0.0495	0.0607	32.3596

In figura 3.4 vengono riportate le immagini classificate in maniera errata dall'algoritmo. Molte volte l'algoritmo predice erroneamente la classe ground (6), quindi sembrerebbe che l'introduzione di tale classe non porti alcun vantaggio.

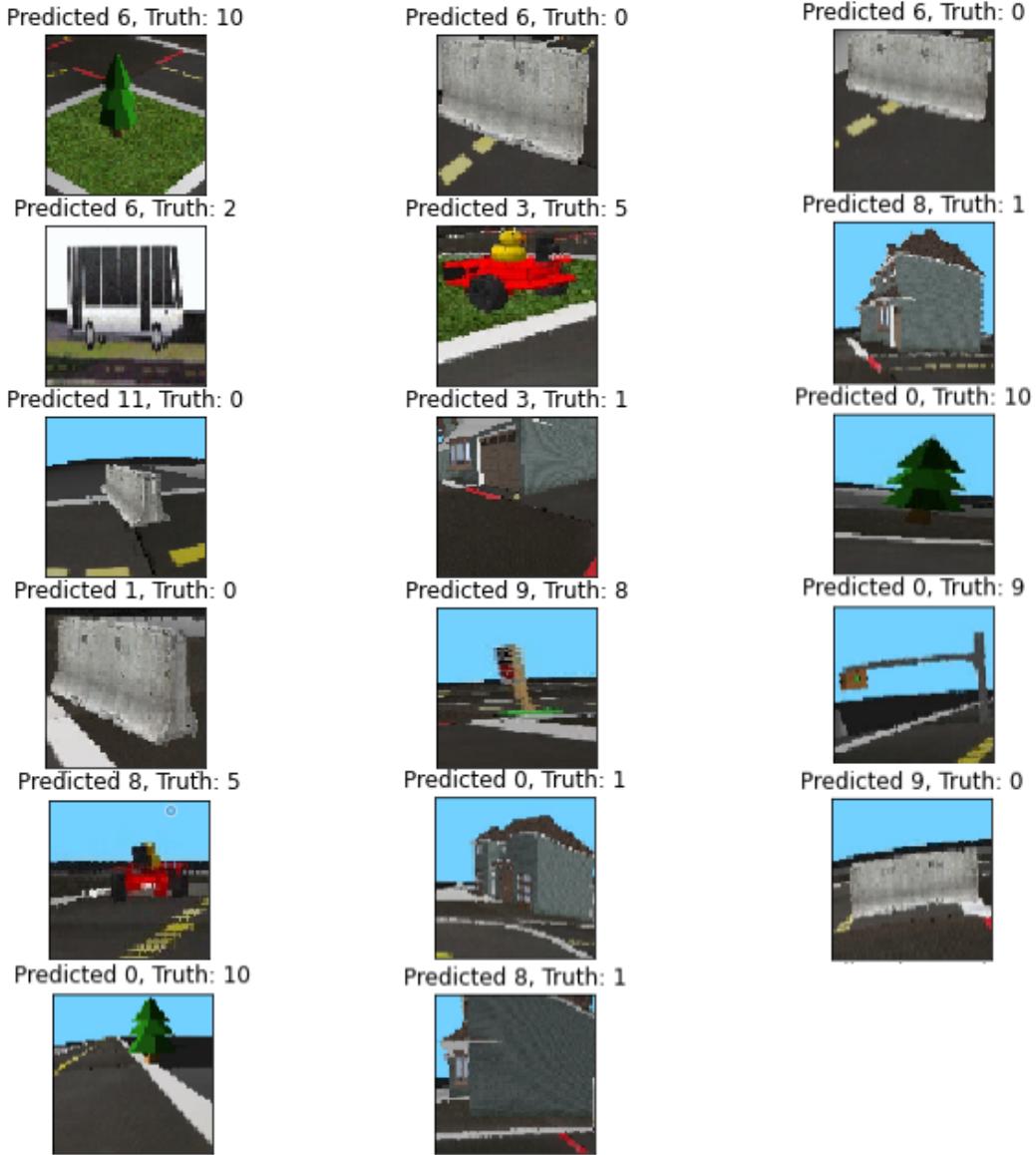
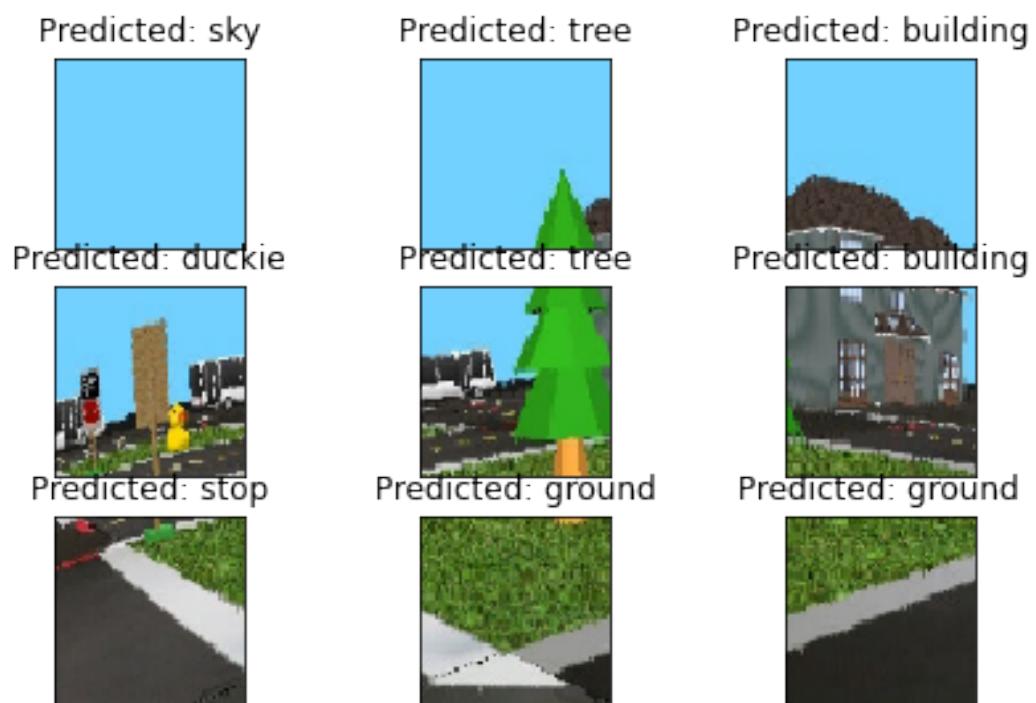
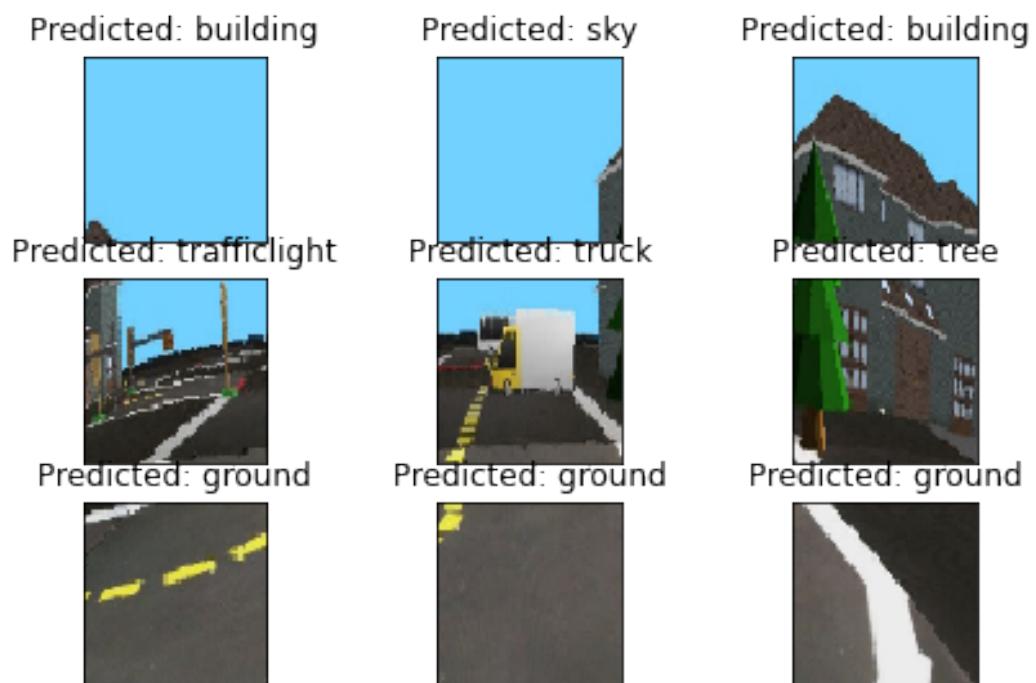


Figura 3.4. Classificazioni errate in caso di aggiunta delle classi ground e sky.

Tuttavia, pensando all'applicazione dell'algoritmo in Duckietown, prendendo immagini contenenti più oggetti è possibile in questo modo effettuare una divisione in 9 parti, piuttosto che in 6, e ottenere che l'algoritmo classifichi ground o sky lì dove non è presente alcun oggetto di interesse, come è possibile vedere negli esempi di seguito riportati. Nel terzo esempio riportato in figura 3.5, è possibile vedere come, se si ha una patch grigio scuro, l'algoritmo predica building e non ground. Aggiungendo un'immagine della strada completamente grigia tra i samples di training, l'algoritmo tende a fare più errori nella classificazione di edifici e barriere, perciò i risultati visti sono i migliori ottenibili.



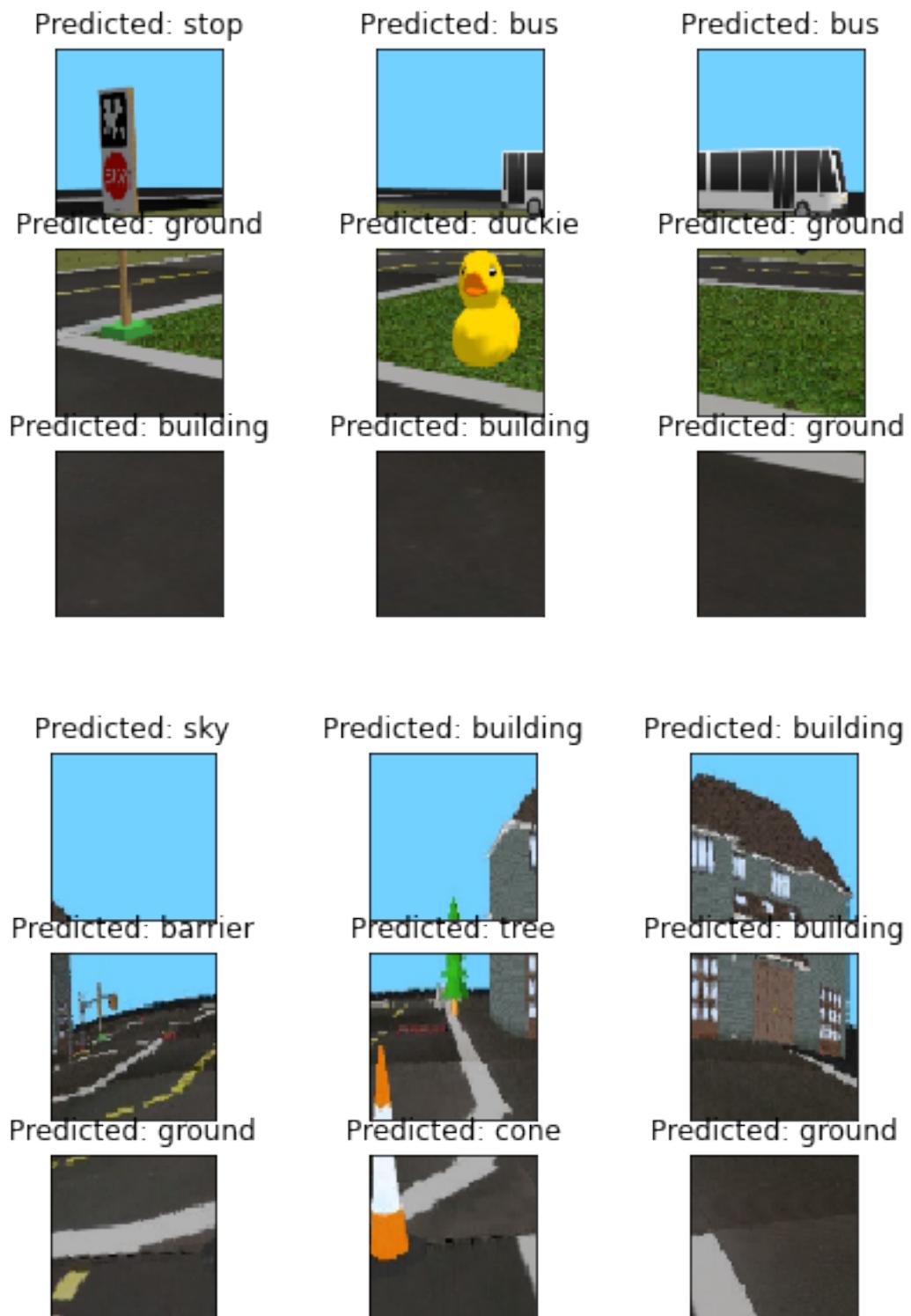


Figura 3.5. Esempi di applicazione della rete su immagini contenenti più oggetti (12 classi)

3.2 Esperimenti sul dataset reale

Come detto in precedenza, per quanto riguarda il dataset reale sono stati fatti esperimenti di vario tipo. Nei prossimi paragrafi sono riportati i risultati ottenuti per ciascuno di essi.

3.2.1 Utilizzo della rete addestrata sul dataset simulato

Il primo esperimento consiste nell'effettuare la classificazione sul dataset reale, del quale vengono considerate solamente le classi presenti anche nel dataset simulato, attraverso la rete addestrata proprio sul dataset simulato. La tabella 3.6 riporta i risultati ottenuti.

Tabella 3.6. Risultati ottenuti sul dataset reale con la rete addestrata sul dataset simulato.

Test loss	6.4637
Test accuracy	0.2213
Classificazioni corrette	27
Classificazioni errate	95
CPU time - test set	0.2032
CPU time - sample	0.0594

La maggior parte delle istanze di test vengono classificate in maniera errata dalla rete. Per capire meglio cosa accade, in figura 3.6 sono riportati 9 esempi di classificazioni corrette e in figura 3.7 9 esempi di classificazioni errate. Molti errori sono dovuti alla qualità delle immagini: infatti, accade molto spesso che in un'immagine si abbiano più oggetti di interesse, ad esempio un edificio in primo piano e una paperella dietro di esso.

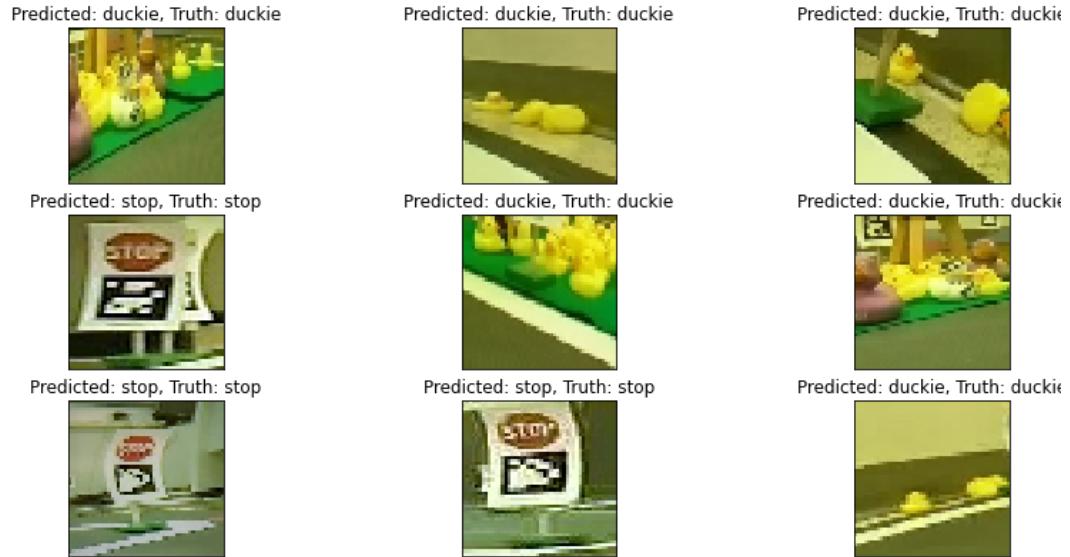


Figura 3.6. Esempi di classificazioni corrette.

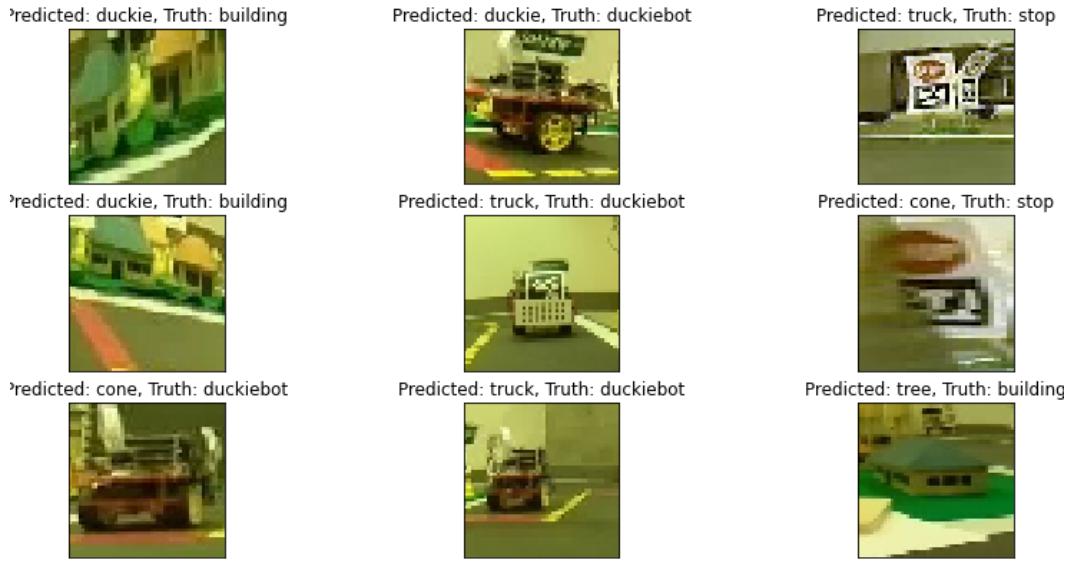


Figura 3.7. Esempi di classificazioni errate.

Purtroppo non è stato possibile migliorare ulteriormente le immagini, perché il dataset da cui sono state ricavate non era destinato ad essere utilizzato per problemi di classificazione e gli oggetti tendono ad essere spesso sovrapposti. Inoltre, in molte immagini le paperelle e gli edifici si trovano su un prato di un verde molto acceso, il che porta l'algoritmo a classificarle come tree, anche se in questo dataset la classe tree non è presente.

3.2.2 Addestramento della rete sul dataset reale con 7 classi

Il fatto che il dataset reale comprenda solo 4 delle 10 classi che troviamo nel dataset simulato fa sì che, nell'esperimento precedente, l'algoritmo effettui la classificazione su più classi di quelle che effettivamente si hanno, perciò ha una maggiore probabilità di sbagliare. Inoltre, il dataset reale contiene anche 3 classi che non sono presenti in quello simulato e che non sono state finora considerate, dal momento che il modello addestrato sul dataset simulato non sarebbe stato in grado di riconoscerle. A questo punto, perciò, è stata addestrata la rete direttamente sul dataset reale con 7 classi.

Tabella 3.7. Risultati ottenuti sul dataset reale addestrando la rete sul training set reale.

Test loss	0.9156
Test accuracy	0.6563
Classificazioni corrette	126
Classificazioni errate	66
CPU time - test set	0.8318
CPU time - sample	0.5152

I risultati sono nettamente migliori di quelli ottenuti nell'esperimento precedente, ma i tempi necessari per la classificazione, sia che questa venga effettuata sull'intero test set che su una singola immagine, risultano essere maggiori. Anche in questo caso, si riportano esempi di immagini classificate correttamente e in modo errato dall'algoritmo.



Figura 3.8. Esempi di classificazioni corrette.

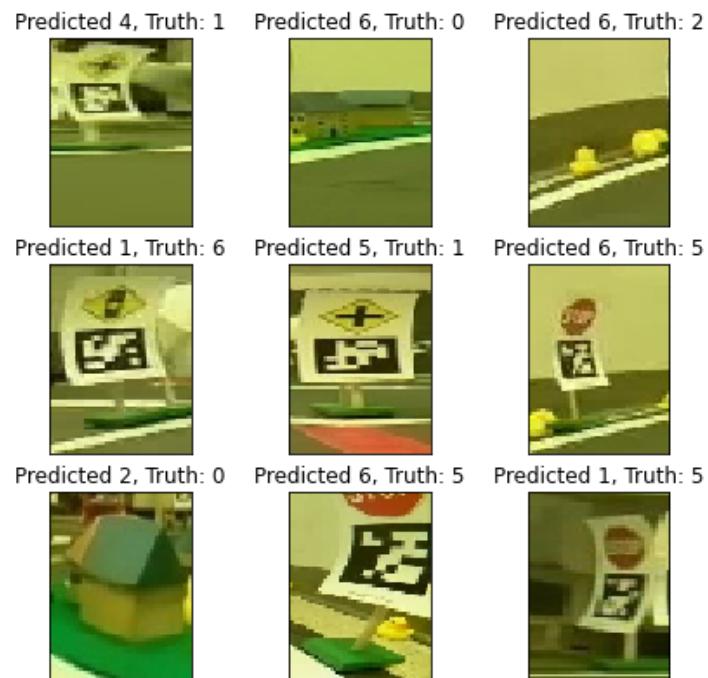


Figura 3.9. Esempi di classificazioni errate.

3.2.3 Rete addestrata sia sul dataset simulato che reale

In quest'ultimo esperimento, è stata utilizzata la rete pre-addestrata sul dataset simulato, ulteriormente addestrata su un piccolo training set ricavato dal dataset reale, del quale sono state considerate solo 4 classi. In particolare, è stato fatto uno studio sul numero di immagini di training del dataset reale da utilizzare per rendere il modello più accurato nella classificazione delle istanze di test.

Tabella 3.8. Risultati degli esperimenti sul dataset reale con la rete pre-addestrata.

Training samples	Loss	Accuracy	Class. corrette	Class. errate
1 - 0 - 2 - 1	1.3616	0.5410	66	56
2 - 1 - 3 - 2	0.5866	0.7623	93	29
2 - 1 - 3 - 3	0.5034	0.8279	101	21
3 - 2 - 4 - 4	0.4417	0.8361	102	20
4 - 2 - 4 - 5	0.2977	0.8771	107	15
4 - 3 - 4 - 5	0.2891	0.8853	108	14
5 - 3 - 5 - 5	0.3027	0.8935	109	13
5 - 3 - 6 - 5	0.3025	0.9016	110	12
5 - 3 - 6 - 6	0.2827	0.9262	113	9
6 - 3 - 6 - 6	0.2550	0.9344	114	8

La prima colonna della tabella 3.8 indica il numero di samples di training delle classi building, duckie, duckiebot e stop del dataset reale utilizzate. In ogni tentativo fatto, si sono aggiunte immagini di training per le classi su cui il modello compiva più errori e, in corrispondenza dell'ultima riga, si trova la combinazione che permette di raggiungere i risultati migliori ottenibili dalla rete. Infatti, pur aggiungendo altre immagini per le diverse classi, i risultati non migliorano e, addirittura, peggiorano. In figura 3.10 sono riportate le immagini classificate in maniera errata dalla rete.

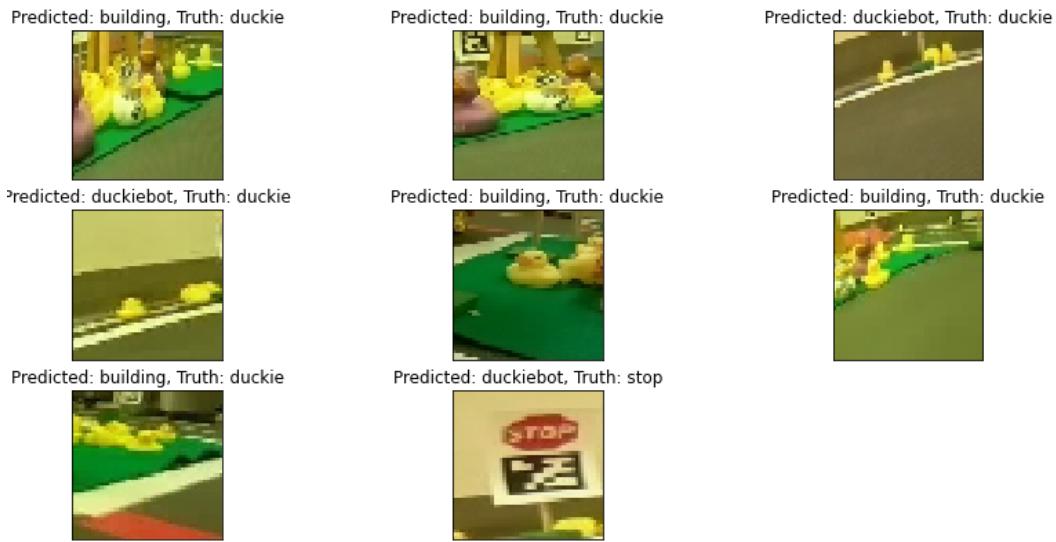


Figura 3.10. Esempi di classificazioni errate.

Come si può facilmente notare, la maggior parte degli errori consistono nella classificazione di immagini di paperelle come edifici. Questo tipo di errore è molto

probabilmente dovuto al fatto che all'interno delle immagini del dataset originale le paperelle e gli edifici, che sono quasi della stessa grandezza, si trovano entrambi sul telo verde che sta a rappresentare il prato, e sono molto spesso sovrapposte, perciò è possibile trovare parti di paperelle nelle immagini della classe building e viceversa, sia per quanto riguarda il training set che il test set. Anche in questo caso, come possibile modalità di applicazione della rete in Duckietown sono state considerate immagini contenenti più oggetti di interesse e sono state suddivise in 6 parti sulle quali è stata effettuata la classificazione.

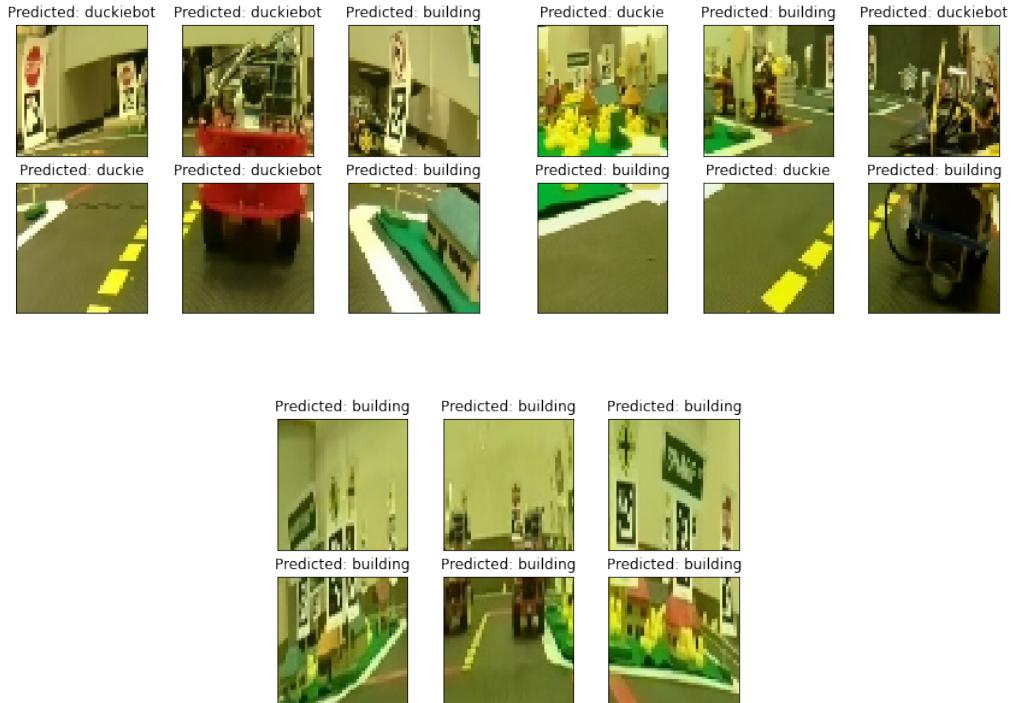


Figura 3.11. Esempi di applicazione della rete su immagini contenenti più oggetti.

Come risulta evidente in figura 3.11, in questo caso i risultati sono peggiori rispetto a quelli che si avevano sul dataset simulato. Sono state fatte delle prove anche suddividendo l'immagine in 9 parti, nel tentativo di isolare il più possibile i singoli oggetti, ma come è possibile vedere in figura 3.12, questo non ha portato ad alcun miglioramento dei risultati. La tabella 3.9 mostra i tempi impiegati dalla rete con l'utilizzo della combinazione ottima di istanze di training per effettuare la classificazione sull'intero test set, su una singola immagine, per la prova finale con divisione in 6 parti e la prova finale con divisione in 9 parti. Per quanto riguarda gli ultimi due, si tratta di una media dei tempi impiegati nei 3 esempi riportati.

Tabella 3.9. Tempi impiegati per la classificazione con l'utilizzo della combinazione ottima di istanze di training.

CPU time - test set	0.1583
CPU time - sample	0.0314
Prova finale - 6 parti	5.0225
Prova finale - 9 parti	12.4207

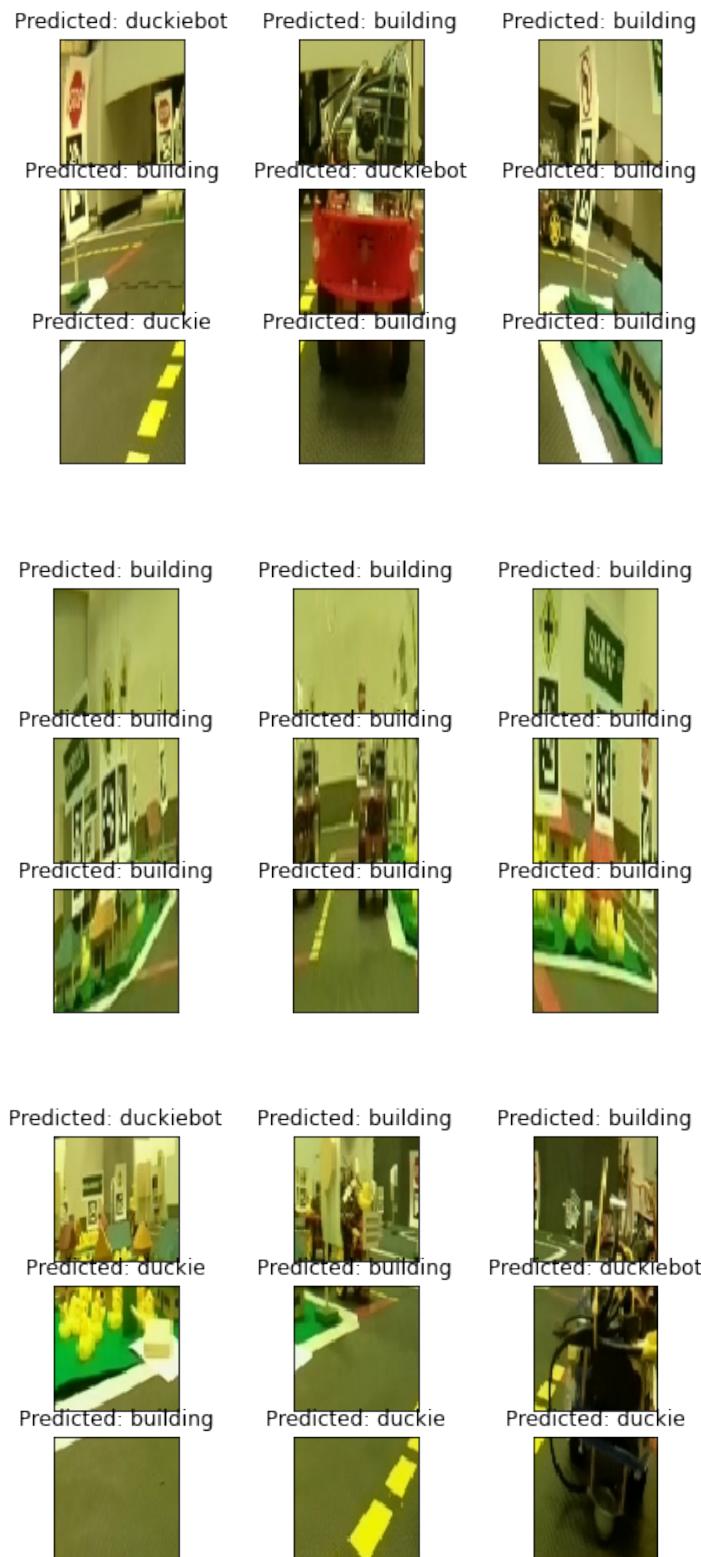


Figura 3.12. Esempi di applicazione della rete su immagini contenenti più oggetti con divisione in 9 parti.

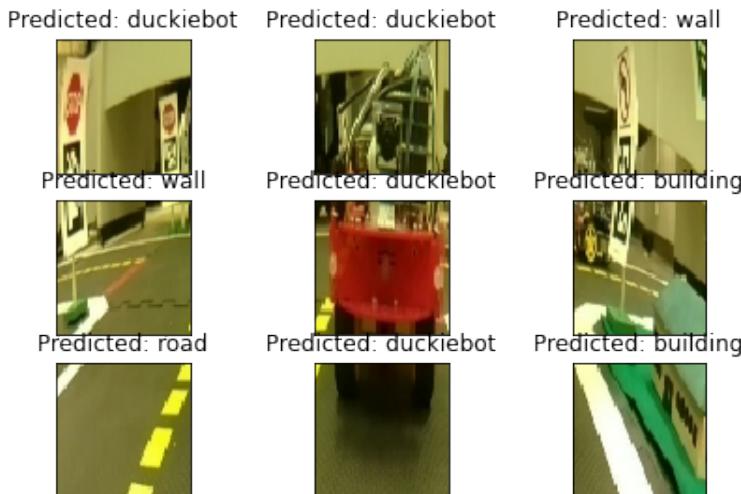
3.2.4 Aggiunta delle classi wall e road

Per tentare di migliorare le performance dell'algoritmo nella classificazione delle singoli parti di un'immagine contenente più oggetti di interesse, sono state aggiunte le classi wall e road, di cui si era già parlato nella sezione relativa al dataset reale. Il numero di samples di training e test utilizzati per ciascuna di tali classi riportati in tale sezione sono quelli che hanno permesso di raggiungere i migliori risultati possibili. Per poter addestrare la rete su questo nuovo dataset, è stato necessario aggiungere le stesse due classi, seppur vuote, cioè prive di istanze di training e test, al dataset simulato e sostituire il Dense 10 della rete con un Dense 12. Quindi, è stata addestrata la nuova rete sul dataset simulato così ottenuto, raggiungendo gli stessi livelli di accuracy avuti precedentemente. Infatti, dato che le due classi non dispongono di alcuna immagine all'interno del dataset simulato, non hanno in alcun modo influenzato l'addestramento. A quel punto, è stato possibile effettuare l'addestramento sul dataset reale, ottenendo i risultati riportati in tabella 3.10.

Tabella 3.10. Risultati ottenuti sul dataset reale con l'aggiunta delle classi road e wall.

Test loss	0.6413
Test accuracy	0.8197
Classificazioni corrette	119
Classificazioni errate	25
CPU time - test set	0.4309
CPU time - sample	0.0658

L'accuracy risulta essere molto più bassa di quella ottenuta precedentemente, tuttavia attraverso l'aggiunta di tali classi l'applicazione di tale algoritmo su immagini contenenti più oggetti di interesse permette di avere risultati più consistenti. In figura 3.13 vengono riportati alcuni esempi significativi.



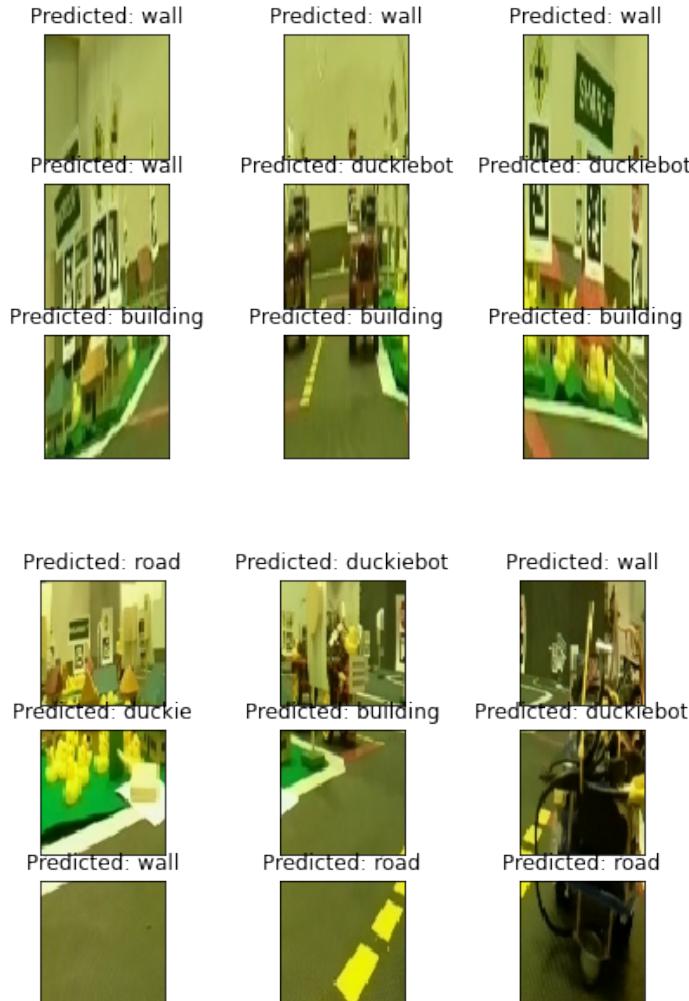


Figura 3.13. Esempi di applicazione della rete su immagini contenenti più oggetti con divisione in 9 parti dopo laggiunta delle classi road e wall.

In tabella 3.11 vengono riportati i tempi necessari per la suddivisione delle immagini in patches e la loro successiva classificazione.

Tabella 3.11. Tempi impiegati per la classificazione nel test finale sul dataset reale.

Numero di patches	CPU-time
6	6.8052
9	10.1160
4	5.5601

Relativamente all'ultima riga, viene riportato il tempo necessario per effettuare la suddivisione dell'immagine in 9 patches e la successiva classificazione delle sole 4 patches nell'angolo in basso a destra, in modo tale da ridurre i tempi. In questo modo vengono considerate solo le parti di immagine che potrebbero contenere eventuali oggetti contro cui rischiamo di andare a sbattere o che potrebbero contenere cartelli stradali o semafori che riguardano il nostro senso di marcia.

Capitolo 4

Conclusioni

I risultati di questo progetto mostrano le potenzialità del few-shot learning nell’ambito di Duckietown. Infatti, utilizzando pochi samples di training per ogni classe è stato possibile raggiungere valori di accuracy molto elevati, pur utilizzando una rete molto leggera.

In particolare, per quanto riguarda il dataset simulato, addestrando la rete con sole 50 immagini è stato possibile classificare correttamente circa il 95% delle immagini di test, ottenendo tempi di classificazione molto buoni, tanto da poter essere vicini al real-time su un buon PC, ma non abbastanza per far girare l’algoritmo real-time su un Raspberry insieme ad altri moduli. La suddivisione delle immagini in più parti e la successiva classificazione delle singole patches ha portato a risultati molto buoni sul dataset simulato. Quasi tutte le patches vengono classificate correttamente e gli errori vengono commessi, in genere, sulle parti in alto, cioè quelle relative agli oggetti più lontani, che proprio per questo motivo sono più sgranati. Volendo classificare solo le 2 parti in basso al centro e a destra, cioè quelle che si suppone contengano gli oggetti davanti a noi, contro cui potremmo andare a sbattere, ed eventuali cartelli stradali o semafori sulla destra, si riesce ad effettuare la suddivisione e la classificazione in pochi secondi. Utilizzando le 10 classi presenti nel dataset simulato, quando in una parte di immagine non è presente alcun oggetto, la rete predice comunque una di tali classi, perciò risulta esserci una barriera lì dove c’è una strada sgombra o un edificio lì dove c’è il cielo. Aggiungendo le classi ground e sky si è riusciti a rendere gli esperimenti sulle immagini contenenti più oggetti ancor più consistenti.

Per quanto riguarda il dataset reale, se si considerano tutte le 7 classi e si addestra la rete, si ottengono valori di accuracy molto bassi, pur aumentando il numero di immagini di training per ogni classe. Effettuando direttamente la classificazione attraverso la rete precedentemente addestrata sul dataset simulato, l’accuracy è ancora molto bassa, ma utilizzando un numero molto ridotto di immagini di training del dataset reale, si riescono a migliorare di molto i risultati ottenuti, raggiungendo valori pressoché uguali a quelli ottenuti sul dataset simulato. Il numero di immagini di training del dataset reale utilizzate per ciascuna classe influenza notevolmente i risultati ottenuti. Una volta arrivati alla combinazione ottima, se si continua ad aumentare il numero di istanze di training, i valori di accuracy tornano a diminuire. In questo caso, prendendo immagini contenenti più oggetti e classificando le singole parti, si ottengono risultati di gran lunga peggiori di quelli avuti sul dataset simulato. Questo è probabilmente dovuto al fatto che le immagini del dataset reale sono molto più sgranate e al loro interno gli oggetti risultano essere sovrapposti. Molto spesso

capita dunque che in un’immagine di un edificio ci sia parte di una paperella, perciò la rete tenderà a confondere i due oggetti. Lo stesso accade con le classi duckiebot e stop, poiché entrambi presentano degli april tags che portano la rete a confondere le due tipologie di oggetti. Dividendo le immagini in 9 parti piuttosto che in 6 parti, i risultati migliorano leggermente, perché si riesce a separare più efficacemente i vari oggetti all’interno delle singole immagini, ma gli errori sono comunque in numero più elevato rispetto al caso simulato. Ancora una volta, con l’aggiunta delle classi road e wall gli esperimenti sulle immagini contenenti più oggetti acquisiscono maggior senso, nonostante si abbia in questo caso una riduzione nell’accuracy del modello. I tempi sono approssimativamente gli stessi che si ottengono per il dataset simulato, perciò molto buoni, ma non abbastanza per un eventuale utilizzo in Duckietown.

Ad ogni modo, considerando solo le 10 classi che si hanno nel dataset simulato, di cui 4 sono in comune con il dataset reale, utilizzando solamente 71 immagini di training totali, si sono raggiunte performance molto elevate per un dataset reale con una rete leggera, che permette di effettuare la classificazione di un’immagine in soli 0.03 secondi. Se si disporrà in futuro di un dataset contenente immagini con una maggiore risoluzione, seppur di piccole dimensioni, e si riuscirà a trovare un metodo più veloce per suddividere un’immagine in più parti o comunque per applicare l’algoritmo di classificazione su immagini contenenti più oggetti, sarà possibile migliorare ulteriormente i risultati, rendendo il modello utilizzabile nell’ambito di Duckietown.

Bibliografia

- [1] Mohit Sewak, Md. Rezaul Karim, Pradeep Pujari, “*Practical Convolutional Neural Networks: Implement advanced deep learning models using Python*”, 2018
- [2] Florian Schroff, Dmitry Kalenichenko, James Philbin, “*FaceNet: A Unified Embedding for Face Recognition and Clustering*”, Giugno 2015, <https://arxiv.org/pdf/1503.03832.pdf>
- [3] Xiaomeng Li, Lequan Yu, Chi-Wing Fu, Meng Fang, and Pheng-Ann Heng, “*Revisiting Metric Learning for Few-Shot Image Classification*”, Luglio 2019, <https://arxiv.org/pdf/1907.03123v1.pdf>
- [4] Shruti Jadon, “*An Overview of Deep Learning Architectures in Few-Shots Learning Domain*”, Agosto 2020, <https://arxiv.org/pdf/2008.06365.pdf>
- [5] Aurélien Géron, “*Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*”, O'REILLY, 2019.
- [6] Duckietown, <https://duckietown.it/>