



UNIVERSITÀ DEGLI STUDI DI CATANIA  
DIPARTIMENTO DI MATEMATICA E INFORMATICA  
CORSO DI LAUREA IN INFORMATICA MAGISTRALE

---

## RELAZIONE FINALE

---

MACHINE LEARNING

---

**STUDENTI**

*Giuseppe Giliberto W82000115*  
*Giuseppe Puglisi W82000112*  
*Giuseppe Sgroi W82000131*

**DOCENTE**

*Prof. Giovanni M. Farinella*

---

ANNO ACCADEMICO 2017/2018

---

# Indice

<b>1</b>	<b>Introduzione</b>	<b>2</b>
<b>2</b>	<b>Eperimenti</b>	<b>4</b>
2.1	Architettura MLP . . . . .	4
2.1.1	Classificazione . . . . .	4
2.1.1.1	Accuracy . . . . .	5
2.1.1.2	Matrice di confusione . . . . .	6
2.1.1.3	Score F1 e Score mF1 . . . . .	7
2.1.2	Regressione . . . . .	7
2.1.2.1	Loss functions . . . . .	9
2.1.2.2	MSE errors . . . . .	9
2.1.2.3	REC curves . . . . .	9
2.1.2.4	RMS error . . . . .	10
2.2	Architettura MLP Deep . . . . .	11
2.2.1	Classificazione . . . . .	11
2.2.1.1	Accuracy . . . . .	12
2.2.1.2	Matrice di confusione . . . . .	12
2.2.1.3	Score F1 e Score mF1 . . . . .	13
2.2.2	Regressione . . . . .	13
2.2.2.1	Loss functions . . . . .	15
2.2.2.2	MSE errors . . . . .	15
2.2.2.3	REC curves . . . . .	16
2.2.2.4	RMS error . . . . .	16
2.3	Architettura VGG16 pre-addestrata . . . . .	17

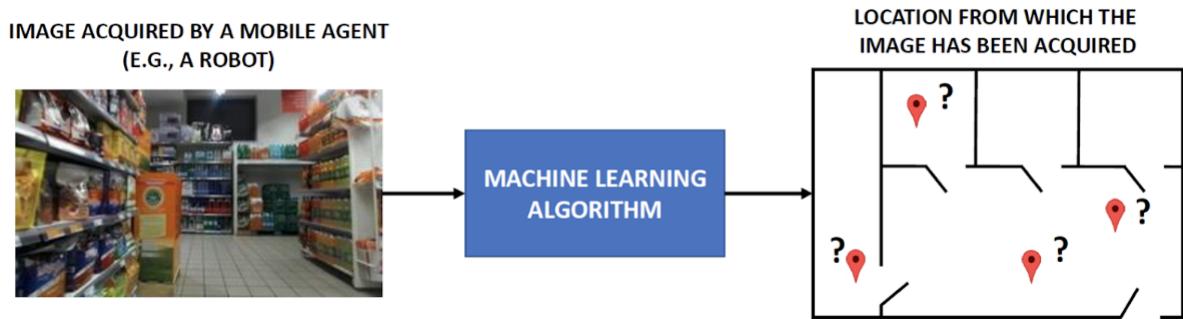
2.4 VGG16 pre-addestrata: 1° approccio . . . . .	18
2.4.1 Classificazione . . . . .	18
2.4.1.1 Accuracy . . . . .	19
2.4.1.2 Matrice di confusione . . . . .	20
2.4.1.3 Score F1 e Score mF1 . . . . .	21
2.4.2 Regressione . . . . .	22
2.4.2.1 Loss functions . . . . .	23
2.4.2.2 MSE errors . . . . .	23
2.4.2.3 REC curves . . . . .	24
2.4.2.4 RMS error . . . . .	24
2.5 VGG16 pre-addestrata: 2° approccio . . . . .	24
2.5.1 Classificazione . . . . .	25
2.5.1.1 Accuracy . . . . .	26
2.5.1.2 Matrice di confusione . . . . .	26
2.5.1.3 Score F1 e Score mF1 . . . . .	27
2.5.2 Regressione . . . . .	28
2.5.2.1 Loss functions . . . . .	29
2.5.2.2 MSE errors . . . . .	29
2.5.2.3 REC curves . . . . .	30
2.5.2.4 RMS error . . . . .	30
<b>3 Metodo Proposto</b>	<b>31</b>
3.1 Classificazione . . . . .	32
3.1.1 Accuracy . . . . .	33
3.1.2 Matrice di confusione . . . . .	33
3.1.3 Score F1 e Score mF1 . . . . .	34
3.2 Regressione . . . . .	35
3.2.1 Loss functions . . . . .	36
3.2.2 MSE errors . . . . .	36

3.2.3 REC curves . . . . .	36
3.2.3.1 RMS error . . . . .	37
<b>4 Conclusioni</b>	<b>38</b>

# 1

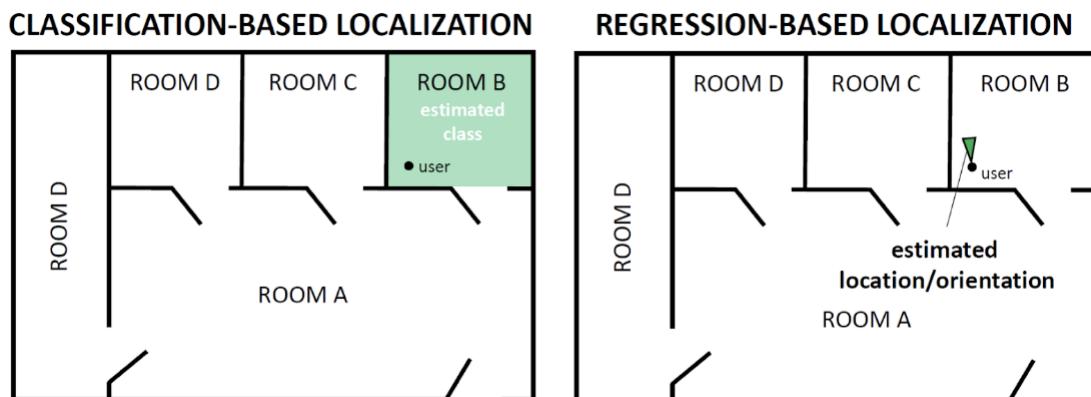
## Introduzione

Il challenge di Machine Learning consiste nell'affrontare un problema di localizzazione basato su immagini, ovvero costruire un algoritmo che, data un'immagine acquisita in uno spazio noto, permetta di inferire la posizione dalla quale l'immagine è stata scattata.



Esso può essere trattato come un problema di machine learning, e affrontato fondamentalmente in due modi:

- Localizzazione basata su classificazione
- Localizzazione basata su regressione



La sorgente dei dati in esame è una camera incorporata in un robot che effettua un determinato cammino in un supermercato. Il robot acquisisce un video che successivamente viene suddiviso nei suoi frames. Essi faranno parte del dataset.

Il **problema di classificazione** consiste nella localizzazione del robot in uno dei 16 reparti del supermercato.

Il **problema di regressione**, invece, consiste nel prevedere la posizione del robot nei momenti dell'acquisizione del video.

Il dataset quindi è formato dalle immagini e da un insieme di relative etichette che rappresentano rispettivamente:

- Posizione rispetto all'asse X
- Posizione rispetto all'asse Y
- Orientamento etichettato da U e V
- Classe di appartenenza

Per risolvere i problemi di classificazione e regressione abbiamo implementato e confrontato i seguenti algoritmi: **MLP**, **MLP Deep**, **VGG16 pre-addestrata** con opportuno fine tuning.

Per l'esecuzione dei test è stata utilizzata la seguente configurazione hardware/software:

- **O.S:** Ubuntu 18.04
- **CUDA:** v9.1.85
- **PyTorch:** v0.4
- **CPU:** Intel core i7 di 2° generazione
- **RAM:** 8 GB ddr3
- **GPU:** Nvidia geForce GTX 1080ti

---

# 2

## Esperimenti

I primi esperimenti effettuati vertono sulla risoluzione del **problema di classificazione**.

Abbiamo implementato le seguenti architetture di rete:

- **MLP**: semplice MLP con un livello nascosto
- **MLP Deep**: MLP con due livelli nascosti
- **VGG16 pre-addestrata**: viene ripresa l'architettura di rete VGG16 e inizializzata con i pesi della stessa rete pre-addestrata su un task differente. Vengono poi effettuate diverse strategie di fine-tuning della rete

Per la risoluzione del **problema di regressione** abbiamo effettuato gli esperimenti essenzialmente con le medesime architetture, andando però a modellare le architetture per la semantica del task.

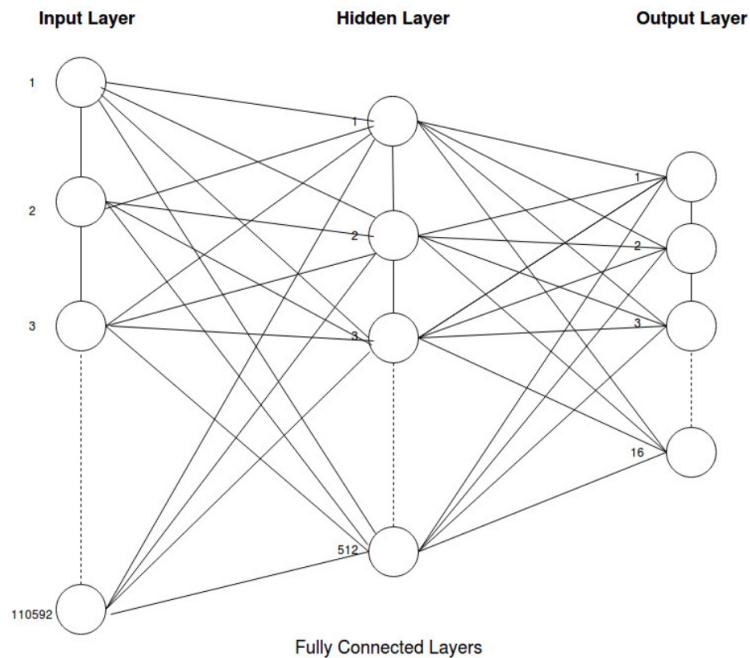
Per le reti MLP viene effettuata la normalizzazione delle immagini per media e deviazione standard del training set, mentre per quanto riguarda la rete VGG16 pre-addestrata viene effettuata la normalizzazione delle immagini per media e deviazione standard del dataset “ImageNet” su cui è stata allenata la rete VGG16.

### 2.1 Architettura MLP

#### 2.1.1 Classificazione

Utilizziamo una semplice architettura MLP con tre livelli:

- Livello di input: 110592 neuroni. In input abbiamo immagini di dimensione 144x256x3
- Livello nascosto: 512 neuroni
- Livello di output: 16 neuroni (Softmax)



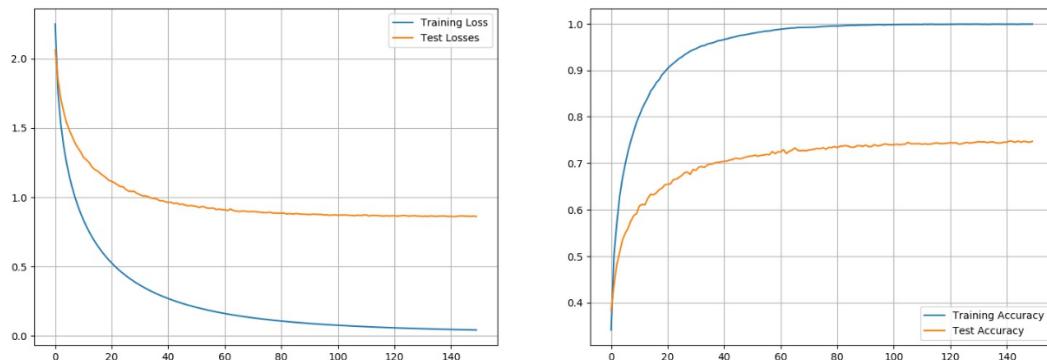
Per la procedura di training utilizziamo:

- Learning rate: 0.000001
- Momentum: 0.9
- Weight decay: 0.000001

Come loss function utilizziamo cross entropy loss, come funzione di attivazione utilizziamo ReLU, mentre come metodo di learning utilizziamo SGD.

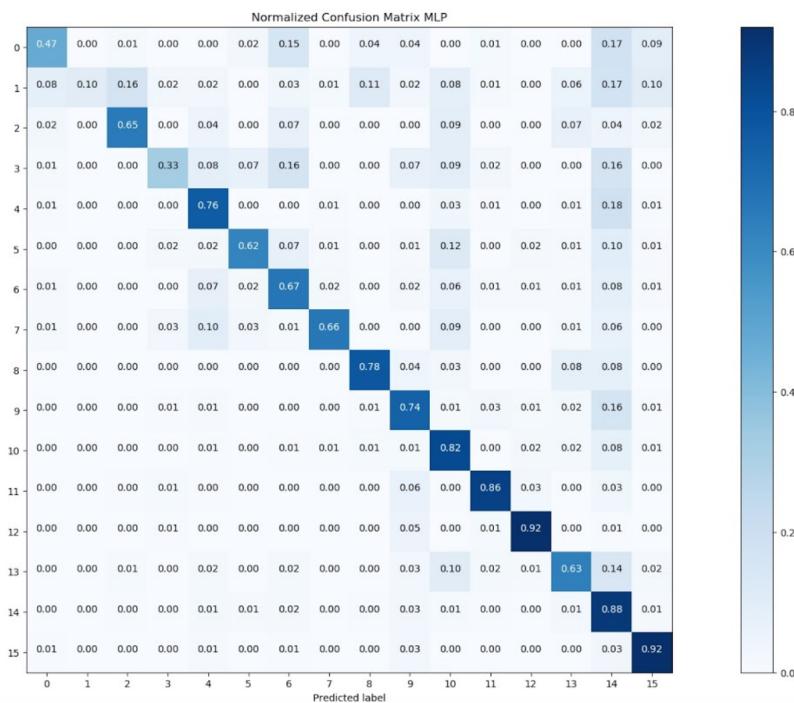
Abbiamo allenato il modello per 150 epoche, ed abbiamo ottenuto i seguenti risultati:

#### 2.1.1.1 Accuracy



Dai due plot si osserva come sul validation la loss rimane relativamente alta, e l'accuracy relativamente bassa. Da ciò si intuisce che questo approccio non dà i risultati sperati.

### 2.1.1.2 Matrice di confusione



Nella matrice di confusione, l'elemento di indici  $i,j$  indica quanti elementi **appartenenti alla classe  $i$  sono stati classificati come appartenenti alla classe  $j$** . In pratica un buon classificatore presenterebbe una matrice di confusione con i numeri sulla diagonale principale alti e i numeri fuori dalla diagonale principale bassi. Attraverso essa è possibile confrontare visivamente i valori reali con quelli stimati dal modello e quindi controllare quali elementi vengono classificati correttamente e quali no. Gli score F1 ne sono una conferma.

### 2.1.1.3 Score F1 e Score mF1

Classe	Score F1
0	0.57
1	0.19
2	0.61
3	0.44
4	0.74
5	0.70
6	0.58
7	0.75
8	0.75
9	0.78
10	0.77
11	0.87
12	0.88
13	0.72
14	0.75
15	0.87
<b>Score mF1</b>	0.68

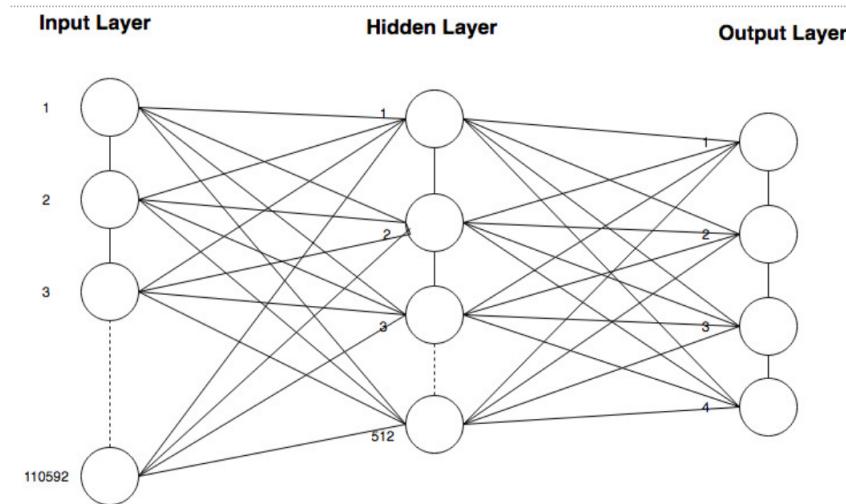
Gli score F1 indicano le performance del classificatore MLP per ogni classe. Per ottenere un indicatore generale di performance viene calcolata la media dei punteggi relativi alle singole classi, che è relativamente basso e pari a 0.68

Con questa architettura otteniamo una **accuracy** di **0.75** sul validation.

### 2.1.2 Regressione

Utilizziamo la semplice architettura MLP con tre livelli vista sopra, con la sola differenza che il livello di output avrà 4 neuroni e non più il classificatore softmax:

- Livello di input: 110592 neuroni. In input abbiamo immagini di dimensione 144x256x3
- Livello nascosto: 512 neuroni
- Livello di output: 4 neuroni



A differenza di quanto fatto per la classificazione, l'ultimo livello non sarà più softmax, in quanto non abbiamo più bisogno di una distribuzione di probabilità sulle classi, ma un vettore di 4 parametri corrispondenti ai valori x, y, u, v che localizzano un'immagine. Per la procedura di training utilizziamo:

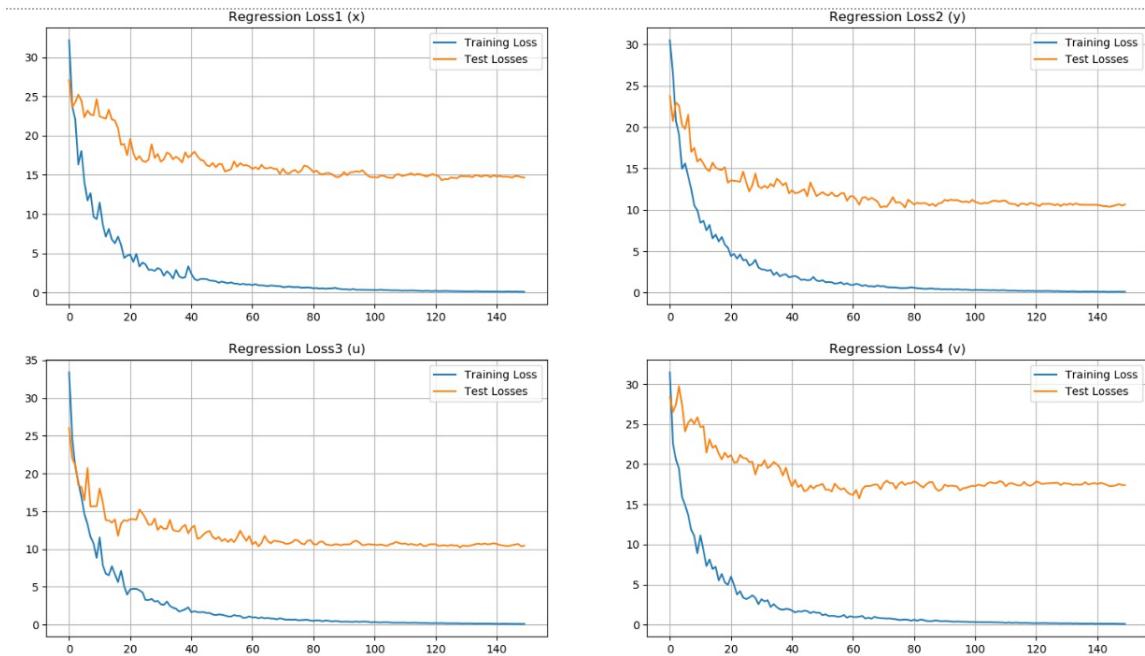
- Learning rate: 0.00001
- Momentum: 0.9
- Weight decay: 0.000001

Come loss function da ottimizzare utilizziamo MSE loss, come funzione di attivazione utilizziamo Tanh, mentre come metodo di learning utilizziamo SGD.

Poiché abbiamo da stimare quattro numeri reali x, y, u, v, utilizziamo a tale scopo quattro loss functions, una per ogni valore, le quali verranno ottimizzate assieme durante la procedura di training.

Abbiamo allenato il modello per 150 epoche, ed abbiamo ottenuto i seguenti risultati:

### 2.1.2.1 Loss functions



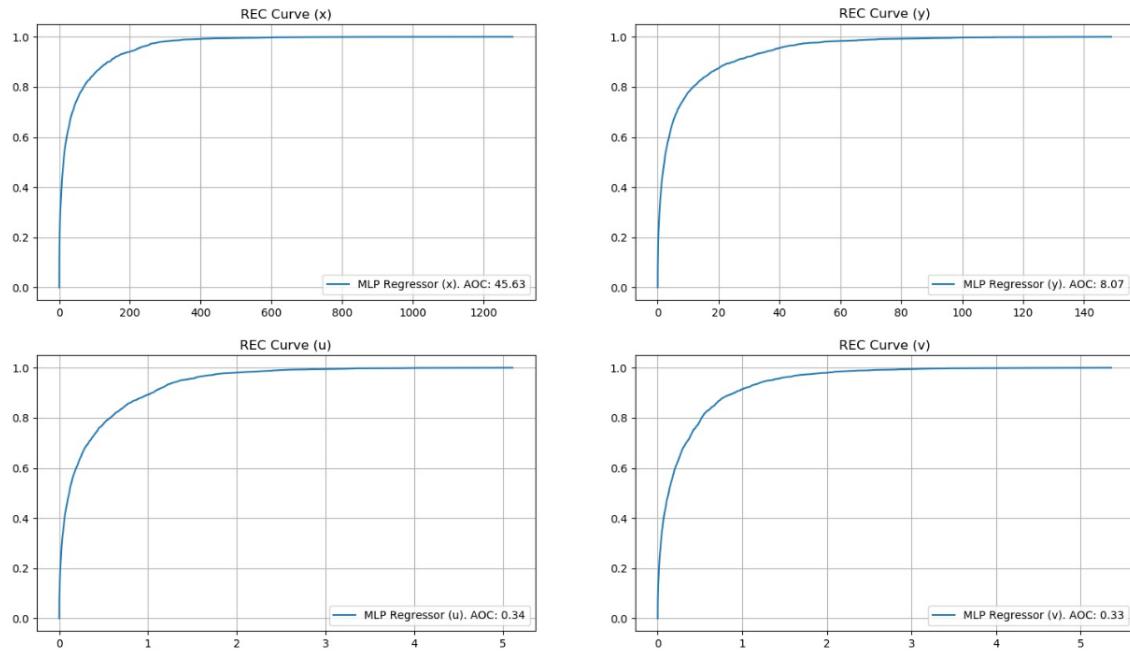
Come è possibile osservare dai 4 plot le loss di validation non raggiungono un valore ottimale, come, invece, accade sul training.

### 2.1.2.2 MSE errors

MSE sul parametro X	45.84
MSE sul parametro Y	8.10
MSE sul parametro U	0.34
MSE sul parametro V	0.33

### 2.1.2.3 REC curves

Le REC curve rappresentano un metodo grafico per valutare la bontà di un metodo di regressione. Inoltre, alla curva è spesso associata l'area sopra la curva (AOC) per offrire una misura dell'errore del metodo.



#### 2.1.2.4 RMS error

errore RMS (Root Mean Square) medio e mediano relativo a posizione (in metri) e orientamento (in gradi):

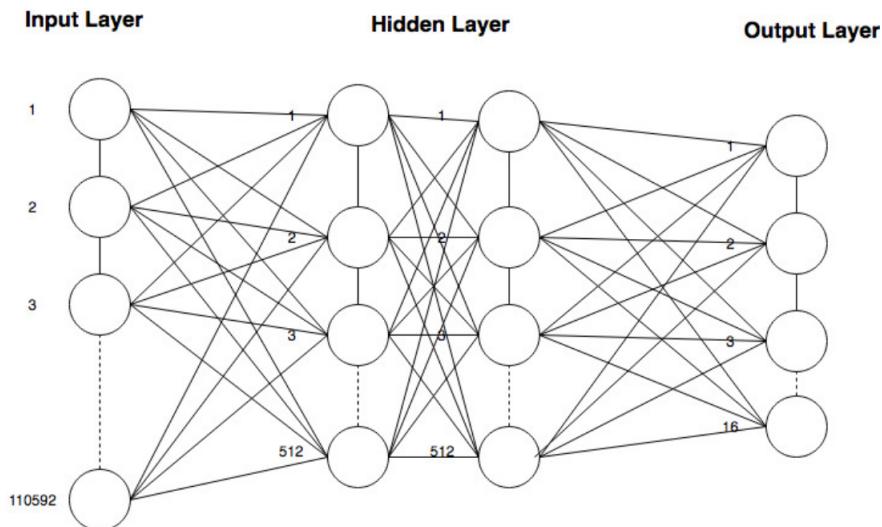
Mean location error	5.7011
Median location error	4.3916
Mean orientation error	46.4265
Median orientation error	29.7348

## 2.2 Architettura MLP Deep

### 2.2.1 Classificazione

Utilizziamo un'architettura di rete molto simile a quella appena vista, con la differenza dell'aggiunta di un secondo livello nascosto:

- Livello di input: 110592 neuroni. In input abbiamo immagini di dimensione 144x256x3
- 2 livelli nascosti: 512 neuroni ciascuno
- Livello di output: 16 neuroni (Softmax)



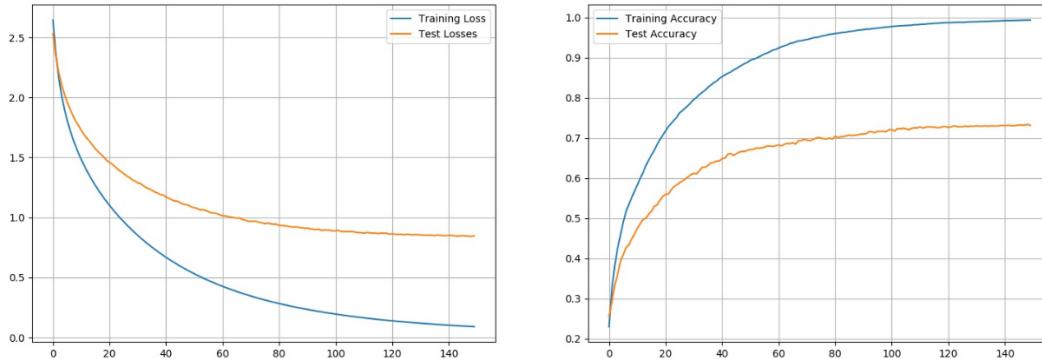
Per la procedura di training utilizziamo:

- Learning rate: 0.00001
- Momentum: 0.9
- Weight decay: 0.000001

Come loss function utilizziamo cross entropy loss, come funzione di attivazione utilizziamo ReLU, mentre come metodo di learning utilizziamo SGD.

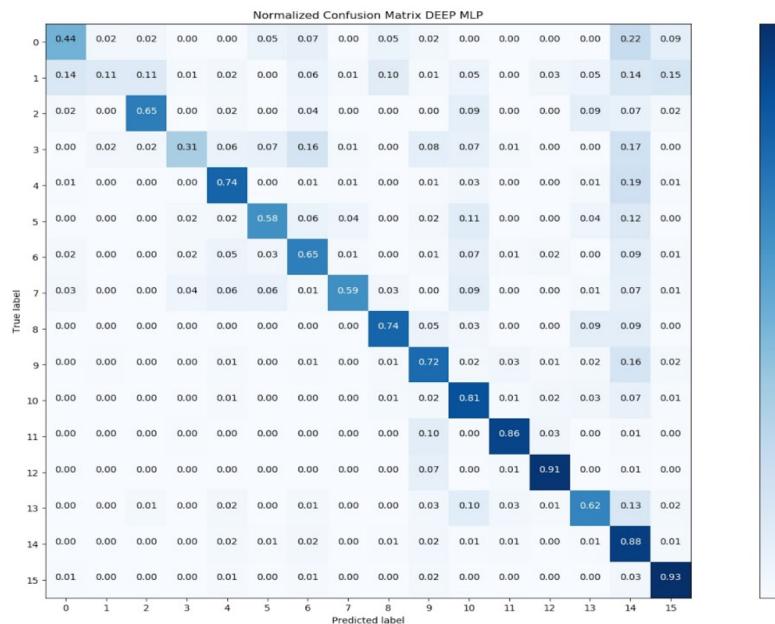
Abbiamo allenato il modello per 150 epoche, ed abbiamo ottenuto i seguenti risultati:

### 2.2.1.1 Accuracy



Anche in questo caso, dai due plot, si osserva come sul validation la loss rimane relativamente alta, e l'accuracy relativamente bassa. Da ciò si intuisce che anche questo approccio non dà i risultati sperati.

### 2.2.1.2 Matrice di confusione



Dalla matrice di confusione possiamo vedere sulla diagonale principale i rates in percentuale che sussistono tra le etichette predette e quelle reali.

A conferma di questi valori si possono osservare gli score F1.

### 2.2.1.3 Score F1 e Score mF1

Classe	Score F1
0	0.52
1	0.19
2	0.61
3	0.43
4	0.74
5	0.65
6	0.57
7	0.68
8	0.71
9	0.75
10	0.76
11	0.86
12	0.87
13	0.71
14	0.74
15	0.85
<b>Score mF1</b>	0.67

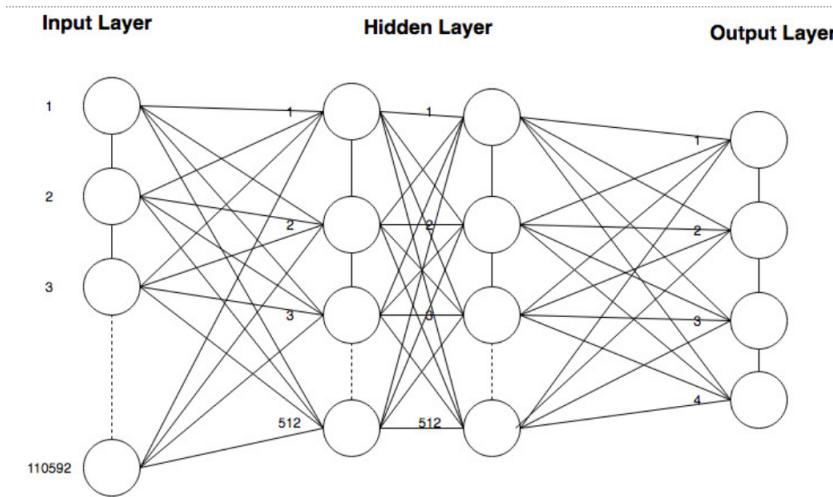
Come è possibile vedere, anche in questo caso, otteniamo un score mF1 relativamente basso e pari a 0.67.

Con questa architettura otteniamo una **accuracy** di **0.73** sul validation.

### 2.2.2 Regressione

L'architettura di rete è la seguente:

- Livello di input: 110592 neuroni. In input abbiamo immagini di dimensione 144x256x3
- 2 livelli nascosti: 512 neuroni ciascuno
- Livello di output: 4 neuroni

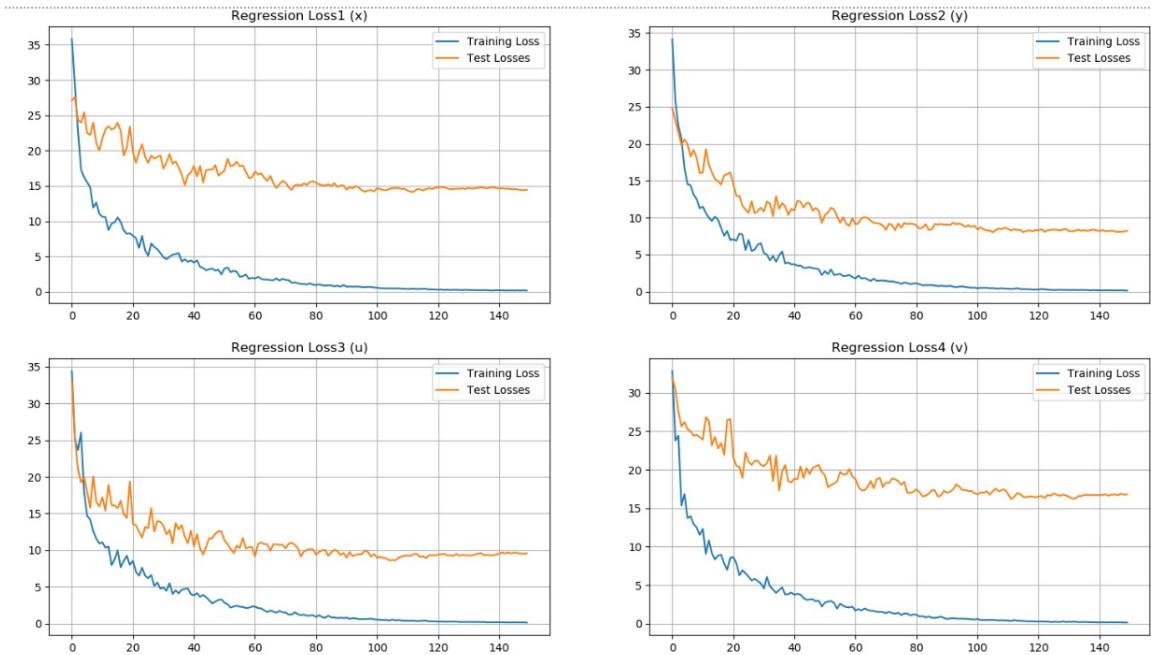


Essendo un problema di regressione, il livello di output sarà costituito da 4 neuroni corrispondenti ai valori  $x$ ,  $y$ ,  $u$ ,  $v$  che localizzano un'immagine. Per la procedura di training utilizziamo:

- Learning rate: 0.00001
- Momentum: 0.9
- Weight decay: 0.000001

Come loss function da ottimizzare utilizziamo MSE loss, come funzione di attivazione utilizziamo Tanh, mentre come metodo di learning utilizziamo SGD. Poiché abbiamo da stimare quattro numeri reali  $x$ ,  $y$ ,  $u$ ,  $v$ , utilizziamo a tale scopo quattro loss functions, una per ogni valore, le quali verranno ottimizzate assieme durante la procedura di training. Abbiamo allenato il modello per 150 epoch, ed abbiamo ottenuto i seguenti risultati:

### 2.2.2.1 Loss functions

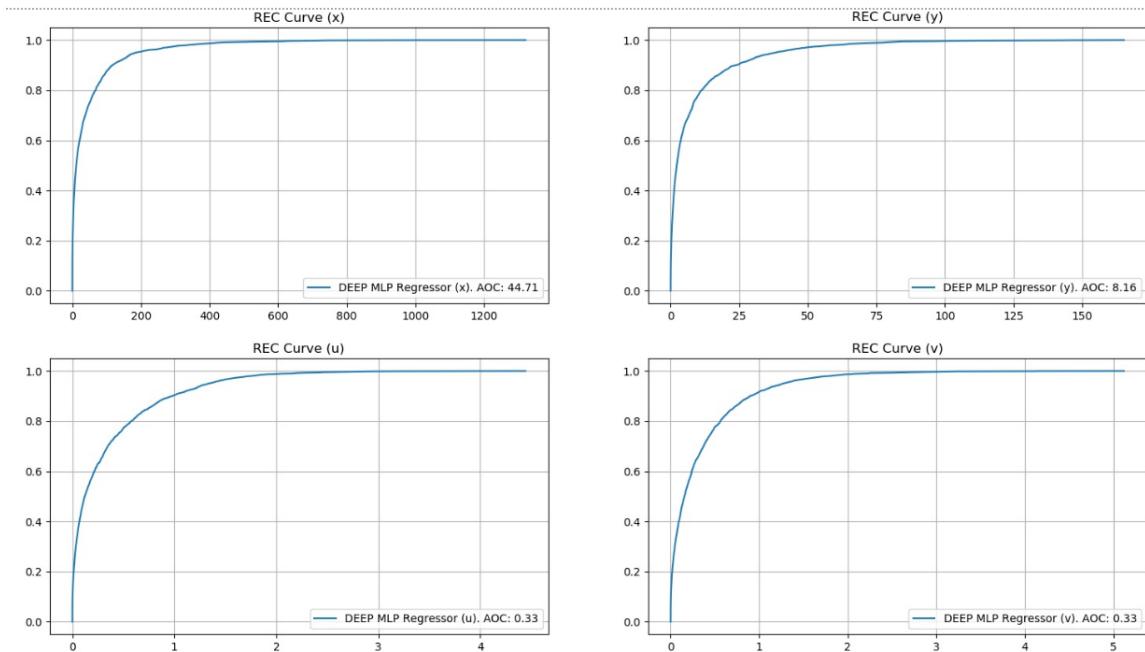


Anche in questo caso i plot mostrano dei risultati relativamente alti per le loss functions sul validation.

### 2.2.2.2 MSE errors

MSE sul parametro X	44.92
MSE sul parametro Y	8.19
MSE sul parametro U	0.33
MSE sul parametro V	0.34

### 2.2.2.3 REC curves



### 2.2.2.4 RMS error

errore RMS (Root Mean Square) medio e mediano relativo a posizione (in metri) e orientamento (in gradi):

Mean location error	5.6398
Median location error	4.4736
Mean orientation error	48.5022
Median orientation error	33.9735

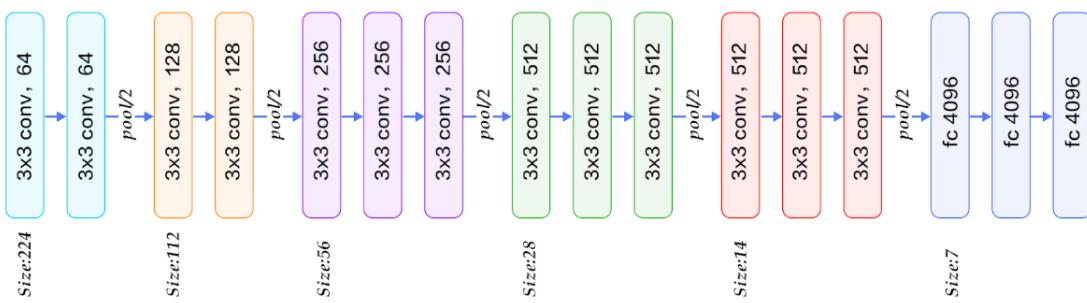
## 2.3 Architettura VGG16 pre-addestrata

Come si è osservato negli esperimenti sovrastanti, i risultati raggiunti non sono stati soddisfacenti, in particolare i plot mostravano che la loss di validation non decresceva così come quella di training. Questo è un chiaro esempio di modelli che non generalizzano a dovere.

Risulta quindi necessario affrontare un'altra tecnica chiamata transfer learning che ci aiuta ad allenare modelli più accurati, con la speranza di ottenere risultati ottimali.

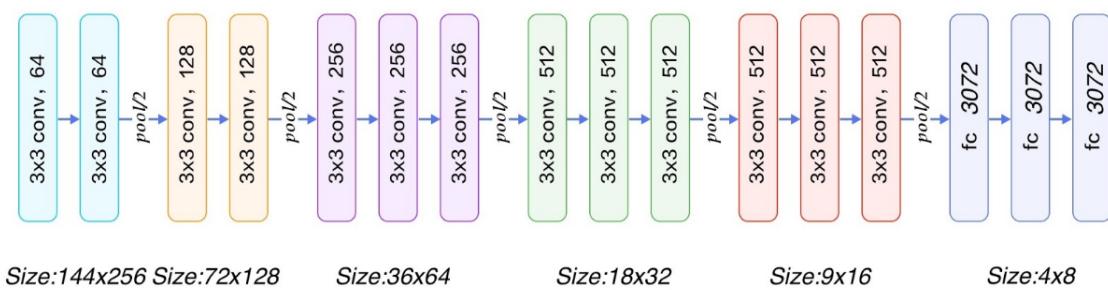
Abbiamo sfruttato l'architettura di rete **VGG-16** pre-addestrata sul dataset “ImageNet” per affrontare i nostri task di classificazione e regressione.

Diamo uno sguardo all'architettura generale:



L'architettura contiene cinque blocchi convoluzionali ed un blocco fully connected finale, e tutti i parametri sono stati allenati per la classificazione su 1000 classi. La rete prende in input immagini 244x244x3 sotto forma di batches, che vengono normalizzate con media e deviazione standard del dataset “ImageNet”.

Abbiamo quindi la necessità, come primo passo, di adattare l'architettura alla dimensione dei nostri input, in modo tale da poter accettare in input immagini 144x256x3:



Successivamente abbiamo seguito due differenti approcci di fine-tuning sulla rete:

1. “Freeze” dei parametri di tutti i livelli convoluzionali, a differenza di quelli dell'ultimo blocco fully connected

2. “Freeze” dei parametri di tutti i livelli convoluzionali, a differenza di quelli dell’ultimo blocco convoluzionale e del blocco fully connected
3. “Freeze” dei parametri di tutti i livelli convoluzionali, a differenza di quelli degli ultimi due blocchi convoluzionali, del blocco fully connected e con l’aggiunta di data augmentation. Questo approccio rappresenterà il nostro metodo proposto

## 2.4 VGG16 pre-addestrata: 1° approccio

Sono stati freezati tutti i livelli del modulo “features” che contiene i blocchi convoluzionali. In questo modo verranno aggiornati solamente i parametri dei livelli del modulo “classifier”, lasciano invariati quelli di tutti i blocchi convoluzionali.

### 2.4.1 Classificazione

```
VGG(
    (features): Sequential(
        (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): ReLU(inplace)
        (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): ReLU(inplace)
        (4): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
        (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (6): ReLU(inplace)
        (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (8): ReLU(inplace)
        (9): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
        (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (11): ReLU(inplace)
        (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (13): ReLU(inplace)
        (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (15): ReLU(inplace)
        (16): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
        (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (18): ReLU(inplace)
        (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (20): ReLU(inplace)
        (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (22): ReLU(inplace)
        (23): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
        (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (25): ReLU(inplace)
        (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (27): ReLU(inplace)
        (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (29): ReLU(inplace)
        (30): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
    )
    (classifier): Sequential(
        (0): Linear(in_features=16384, out_features=3072, bias=True)
        (1): ReLU(inplace)
        (2): Dropout(p=0.5)
        (3): Linear(in_features=3072, out_features=3072, bias=True)
        (4): ReLU(inplace)
        (5): Dropout(p=0.5)
        (6): Linear(in_features=3072, out_features=16, bias=True)
    )
)
```

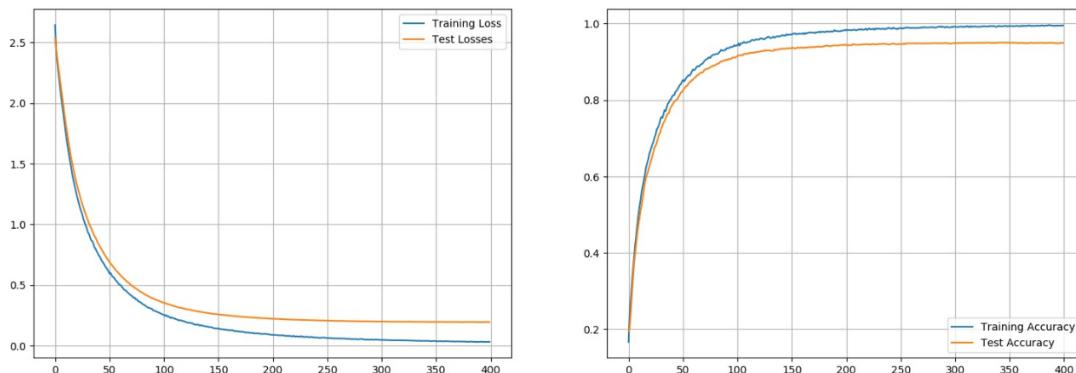
Poichè il task di classificazione richiede in output 16 classi, sono stati modificati gli ultimi livelli lineari impostando a 16 le features di output dell'ultimo livello con classificatore softmax. Per la procedura di training utilizziamo:

- Learning rate: 0.00001
- Momentum: 0.9
- Weight decay: 0.000001

Come loss function utilizziamo cross entropy loss, come funzione di attivazione utilizziamo ReLU, mentre come metodo di learning utilizziamo SGD.

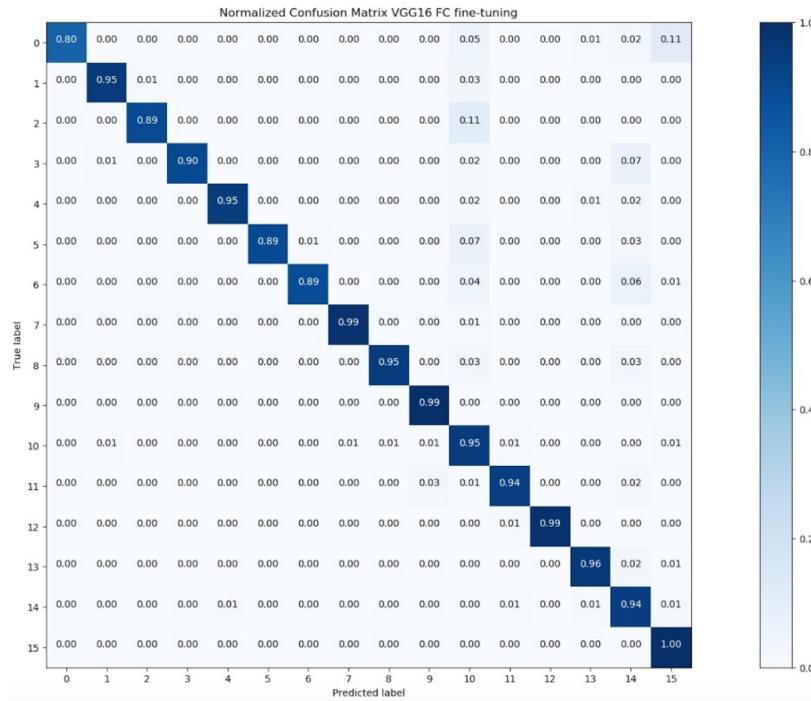
Abbiamo allenato il modello per 400 epoche, ed abbiamo ottenuto i seguenti risultati:

#### 2.4.1.1 Accuracy



Dai plot, di cui sopra, possiamo osservare che le CNN danno risultati ampiamente migliori. Infatti siamo riusciti ad avvicinare quanto più possibile la loss di validation a quella di training.

### 2.4.1.2 Matrice di confusione



Anche dalla matrice di confusione vengono evidenziati miglioramenti avvenuti con le CNN.

#### 2.4.1.3 Score F1 e Score mF1

Classe	Score F1
0	0.89
1	0.95
2	0.93
3	0.94
4	0.96
5	0.93
6	0.93
7	0.98
8	0.95
9	0.99
10	0.92
11	0.95
12	0.99
13	0.97
14	0.93
15	0.95
<b>Score mF1</b>	0.95

Come è possibile vedere si ha uno score mF1 più alto e pari a 0.95.

Con questa architettura otteniamo una **accuracy** di **0.95** sul validation.

### 2.4.2 Regressione

```

VGG(
    (features): Sequential(
        (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): ReLU(inplace)
        (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): ReLU(inplace)
        (4): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
        (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (6): ReLU(inplace)
        (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (8): ReLU(inplace)
        (9): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
        (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (11): ReLU(inplace)
        (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (13): ReLU(inplace)
        (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (15): ReLU(inplace)
        (16): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
        (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (18): ReLU(inplace)
        (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (20): ReLU(inplace)
        (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (22): ReLU(inplace)
        (23): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
        (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (25): ReLU(inplace)
        (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (27): ReLU(inplace)
        (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (29): ReLU(inplace)
        (30): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
    )
    (classifier): Sequential(
        (0): Linear(in_features=16384, out_features=3072, bias=True)
        (1): ReLU(inplace)
        (2): Dropout(p=0.5)
        (3): Linear(in_features=3072, out_features=3072, bias=True)
        (4): ReLU(inplace)
        (5): Dropout(p=0.5)
        (6): Linear(in_features=3072, out_features=4, bias=True)
    )
)

```

Poiché il task di regressione richiede in output 4 numeri reali  $x, y, u, v$ , sono stati modificati gli ultimi livelli lineari impostando a 4 le features di output dell'ultimo livello. Per la procedura di training utilizziamo:

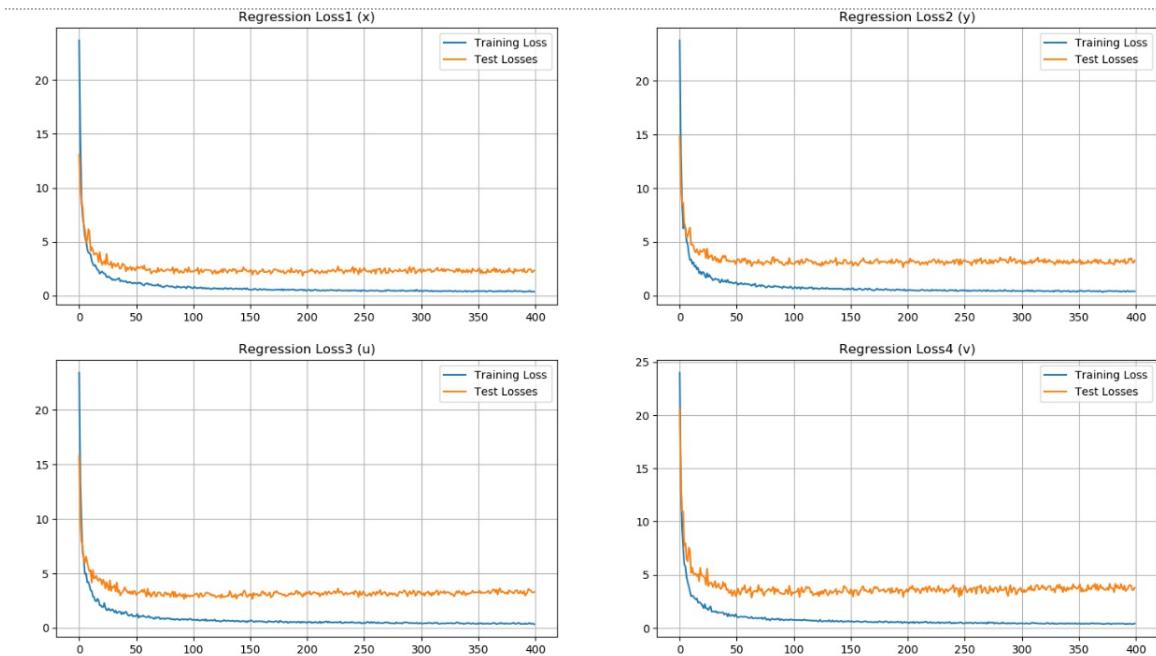
- Learning rate: 0.000001
- Momentum: 0.9
- Weight decay: 0.000001

Come loss function da ottimizzare utilizziamo MSE loss, come funzione di attivazione utilizziamo ReLU, mentre come metodo di learning utilizziamo SGD.

Poiché abbiamo da stimare quattro numeri reali  $x, y, u, v$ , utilizziamo a tale scopo quattro loss functions, una per ogni valore, le quali verranno ottimizzate assieme durante la procedura di training.

Abbiamo allenato il modello per 400 epoche, ed abbiamo ottenuto i seguenti risultati:

### 2.4.2.1 Loss functions

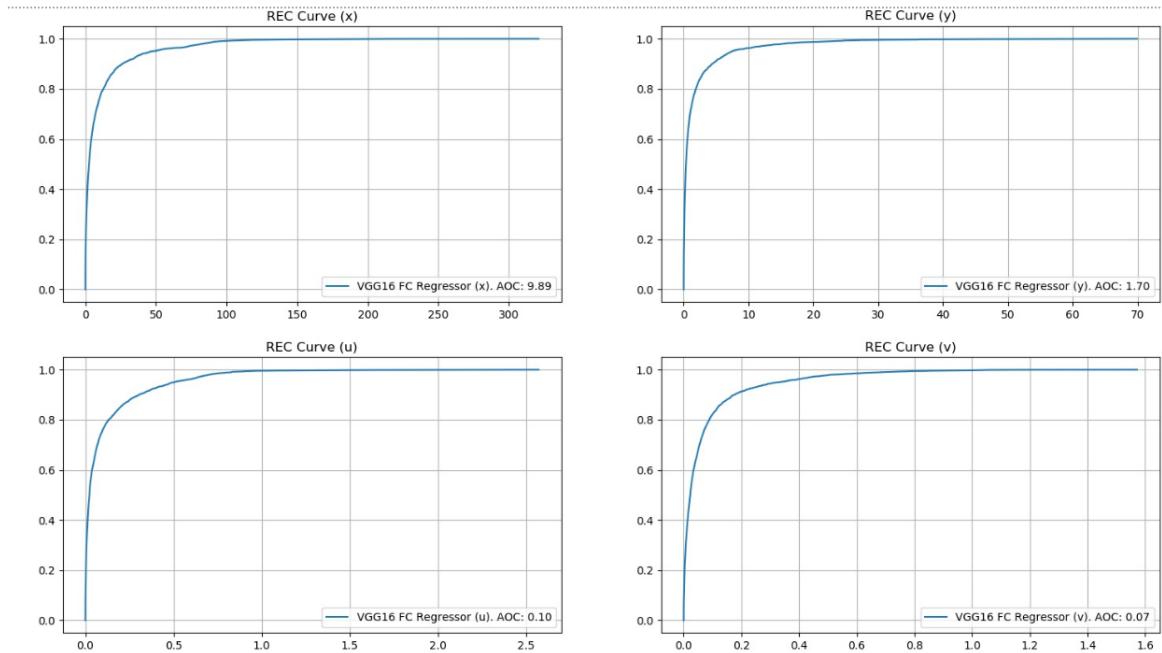


Come è possibile osservare le loss sul validation si avvicinano a quelle del training. Tuttavia ancora siamo lontani dai risultati ottimali avuti col metodo proposto.

### 2.4.2.2 MSE errors

MSE sul parametro X	9.94
MSE sul parametro Y	1.71
MSE sul parametro U	0.10
MSE sul parametro V	0.07

### 2.4.2.3 REC curves



### 2.4.2.4 RMS error

errore RMS (Root Mean Square) medio e mediano relativo a posizione (in metri) e orientamento (in gradi):

Mean location error	2.5976
Median location error	1.9156
Mean orientation error	12.9392
Median orientation error	7.6587

## 2.5 VGG16 pre-addestrata: 2° approccio

Sono stati freezati tutti i livelli del modulo “features” che contiene i blocchi convoluzionali, ad eccezione dell’ultimo blocco. In questo modo verranno aggiornati solamente i parametri dei livelli del modulo “classifier”, ed i parametri dei livelli dell’ultimo blocco convoluzionale.

### 2.5.1 Classificazione

```

VGG(
    (features): Sequential(
        (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): ReLU(inplace)
        (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): ReLU(inplace)
        (4): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
        (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (6): ReLU(inplace)
        (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (8): ReLU(inplace)
        (9): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
        (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (11): ReLU(inplace)
        (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (13): ReLU(inplace)
        (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (15): ReLU(inplace)
        (16): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
        (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (18): ReLU(inplace)
        (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (20): ReLU(inplace)
        (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (22): ReLU(inplace)
        (23): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
        (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (25): ReLU(inplace)
        (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (27): ReLU(inplace)
        (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (29): ReLU(inplace)
        (30): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
    )
    (classifier): Sequential(
        (0): Linear(in_features=16384, out_features=3072, bias=True)
        (1): ReLU(inplace)
        (2): Dropout(p=0.5)
        (3): Linear(in_features=3072, out_features=3072, bias=True)
        (4): ReLU(inplace)
        (5): Dropout(p=0.5)
        (6): Linear(in_features=3072, out_features=16, bias=True)
    )
)

```

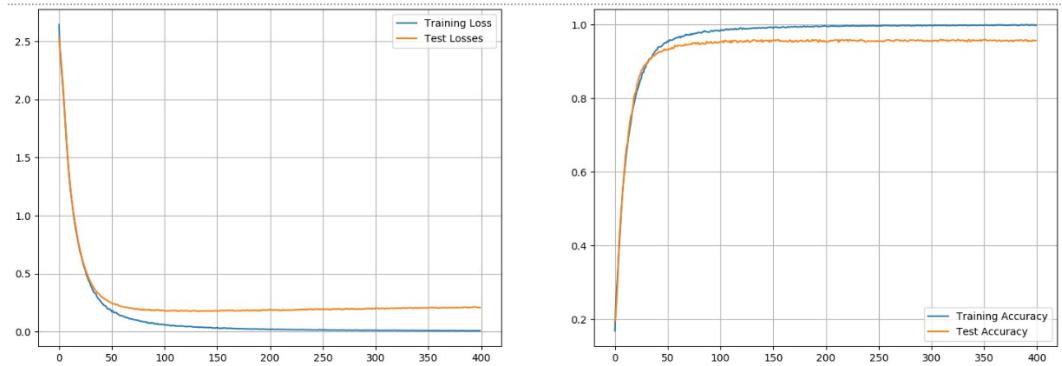
Poichè il task di classificazione richiede in output 16 classi, sono stati modificati gli ultimi livelli lineari impostando a 16 le features di output dell'ultimo livello con classificatore softmax. Per la procedura di training utilizziamo:

- Learning rate: 0.00001
- Momentum: 0.9
- Weight decay: 0.000001

Come loss function utilizziamo cross entropy loss, come funzione di attivazione utilizziamo ReLU, mentre come metodo di learning utilizziamo SGD.

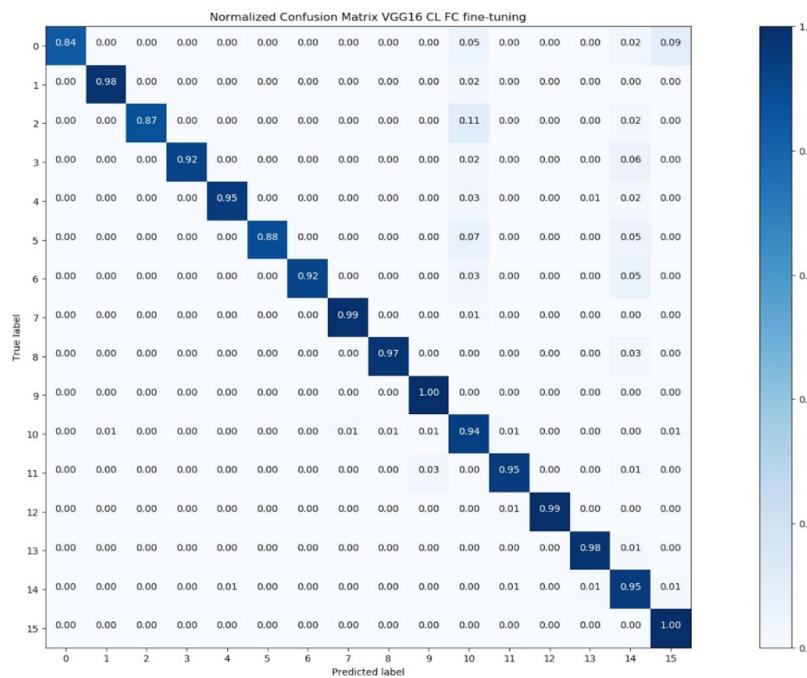
Abbiamo allenato il modello per 400 epoche, ed abbiamo ottenuto i seguenti risultati:

### 2.5.1.1 Accuracy



Dai plot, di cui sopra, è possibile osservare che i ottiene una accuracy relativamente alta: 0.96.

### 2.5.1.2 Matrice di confusione



Abbiamo ottimi risultati sulla diagonale principale, infatti avremo degli score F1 maggiori.

### 2.5.1.3 Score F1 e Score mF1

Classe	Score F1
0	0.91
1	0.97
2	0.93
3	0.95
4	0.96
5	0.93
6	0.96
7	0.96
8	0.96
9	0.99
10	0.92
11	0.96
12	0.99
13	0.98
14	0.95
15	0.96
<b>Score mF1</b>	0.95

Con questa architettura abbiamo raggiunto una **accuracy** di **0.96** sul validation.

### 2.5.2 Regressione

```

VGG(
    (features): Sequential(
        (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): ReLU(inplace)
        (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): ReLU(inplace)
        (4): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
        (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (6): ReLU(inplace)
        (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (8): ReLU(inplace)
        (9): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
        (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (11): ReLU(inplace)
        (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (13): ReLU(inplace)
        (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (15): ReLU(inplace)
        (16): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
        (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (18): ReLU(inplace)
        (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (20): ReLU(inplace)
        (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (22): ReLU(inplace)
        (23): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
        (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (25): ReLU(inplace)
        (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (27): ReLU(inplace)
        (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (29): ReLU(inplace)
        (30): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
    )
    (classifier): Sequential(
        (0): Linear(in_features=16384, out_features=3072, bias=True)
        (1): ReLU(inplace)
        (2): Dropout(p=0.5)
        (3): Linear(in_features=3072, out_features=3072, bias=True)
        (4): ReLU(inplace)
        (5): Dropout(p=0.5)
        (6): Linear(in_features=3072, out_features= 4 , bias=True)
    )
)

```

Poiché il task di regressione richiede in output 4 numeri reali x, y, u, v, sono stati modificati gli ultimi livelli lineari impostando a 4 le features di output dell'ultimo livello.

Per la procedura di training utilizziamo:

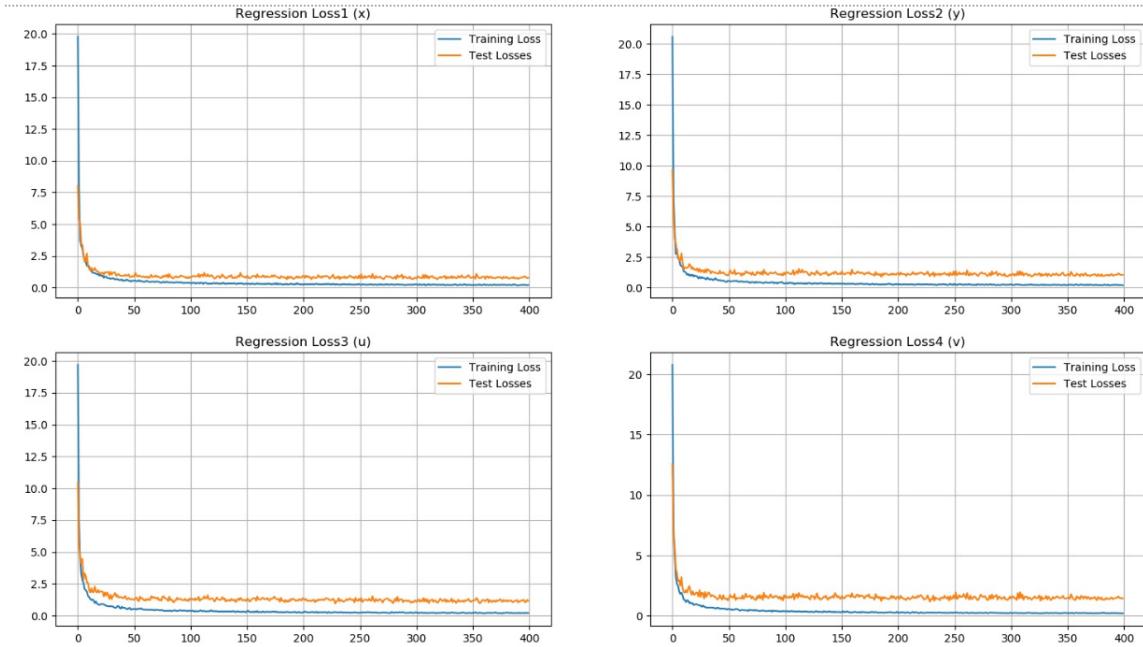
- Learning rate: 0.000001
- Momentum: 0.9
- Weight decay: 0.000001

Come loss function da ottimizzare utilizziamo MSE loss, come funzione di attivazione utilizziamo ReLU, mentre come metodo di learning utilizziamo SGD.

Poiché abbiamo da stimare quattro numeri reali x, y, u, v, utilizziamo a tale scopo quattro loss functions, una per ogni valore, le quali verranno ottimizzate assieme durante la procedura di training.

Abbiamo allenato il modello per 400 epoche, ed abbiamo ottenuto i seguenti risultati:

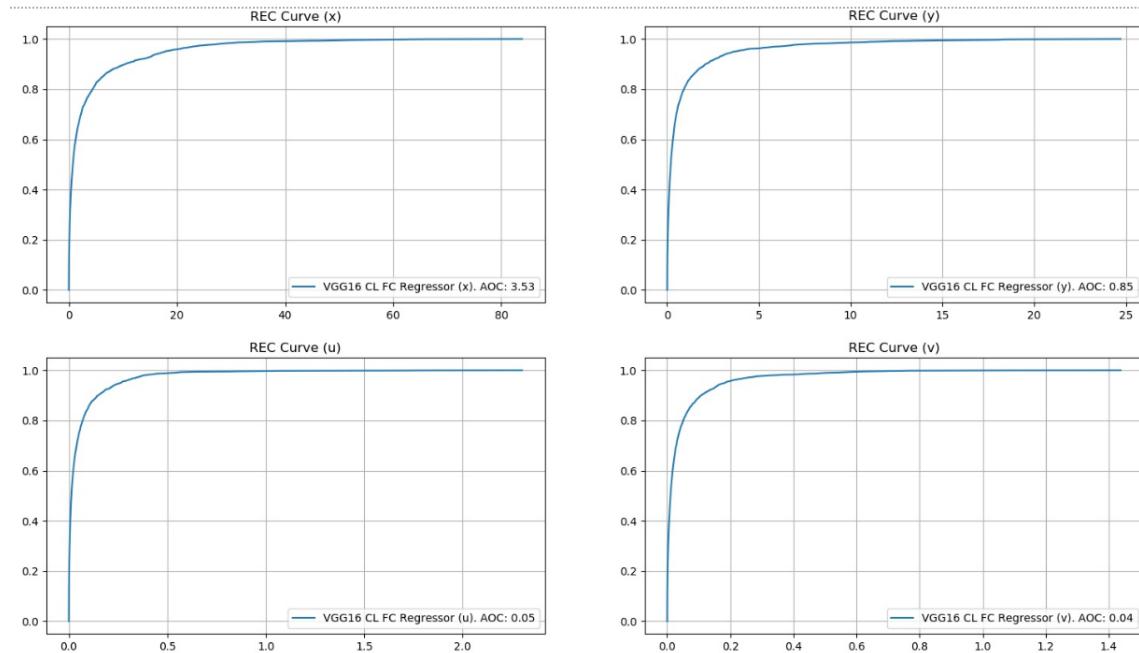
### 2.5.2.1 Loss functions



### 2.5.2.2 MSE errors

MSE sul parametro X	3.55
MSE sul parametro Y	0.85
MSE sul parametro U	0.06
MSE sul parametro V	0.04

### 2.5.2.3 REC curves



### 2.5.2.4 RMS error

errore RMS (Root Mean Square) medio e mediano relativo a posizione (in metri) e orientamento (in gradi):

Mean location error	1.5754
Median location error	1.1219
Mean orientation error	8.7991
Median orientation error	5.3730

---

# 3

## Metodo Proposto

Il metodo che ha fornito i migliori risultati, sia per il task di classificazione che per quello di regressione, è stato ottenuto affinando le diverse strategie sperimentate sopra elencate. Esso sfrutta ancora una volta l'architettura di rete VGG16 preaddestrata sul dataset “ImageNet” e si basa essenzialmente sull'aggiunta di “data augmentation”, ovvero una tecnica che permette di aumentare i dati in maniera sintetica trasformando “al volo” i dati in input in maniera casuale e facendo in modo che, dopo la trasformazione, l'etichetta sia ancora valida, e su un opportuno “fine-tuning” della rete.

Ciò ci ha permesso di ottenere un modello più robusto e capace di generalizzare, riducendo l'overfitting. Le trasformazioni, applicate solo in fase di training, sono le seguenti:

- Random Horizontal Flip: flip orizzontale random con probabilità 0.5
- Color Jitter: il valore di ogni pixel viene perturbato leggermente in maniera casuale
- Random Rotation(20): rotazione random dell'immagine tra  $-20^\circ$  e  $+20^\circ$
- Random Crop(128): crop 128x128 viene estratto casualmente dall'immagine

Per compatibilità, in fase di test vengono estratti crop 128x128 dalla parte centrale dell'immagine. A seguito delle trasformazioni, siamo costretti ad adattare la struttura dell'architettura per accettare in input immagini da 128x128, andando essenzialmente a modificare i livelli lineari del blocco fully connected.

E' stato effettuato poi il fine-tuning del modello, freezando tutti i livelli del modulo “features” che contiene i blocchi convoluzionali, ad eccezione degli ultimi due blocchi, e del blocco fully connected del modulo “classifier”.

In questo modo verranno aggiornati solamente i parametri dei livelli del blocco fully connected, ed i parametri dei livelli degli ultimi due blocchi convoluzionali.

Sono stati poi cambiati i dropout nei livelli fully connected, impostando come probabilità 0.25.

### 3.1 Classificazione

```
VGG(
    (features): Sequential(
        (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): ReLU(inplace)
        (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): ReLU(inplace)
        (4): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
        (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (6): ReLU(inplace)
        (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (8): ReLU(inplace)
        (9): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
        (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (11): ReLU(inplace)
        (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (13): ReLU(inplace)
        (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (15): ReLU(inplace)
        (16): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
        (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (18): ReLU(inplace)
        (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (20): ReLU(inplace)
        (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (22): ReLU(inplace)
        (23): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
        (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (25): ReLU(inplace)
        (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (27): ReLU(inplace)
        (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (29): ReLU(inplace)
        (30): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
    )
    (classifier): Sequential(
        (0): Linear(in_features=8192, out_features=2048, bias=True)
        (1): ReLU(inplace)
        (2): Dropout(p=0.25)
        (3): Linear(in_features=2048, out_features=2048, bias=True)
        (4): ReLU(inplace)
        (5): Dropout(p=0.25)
        (6): Linear(in_features=2048, out_features=16, bias=True)
    )
)
```

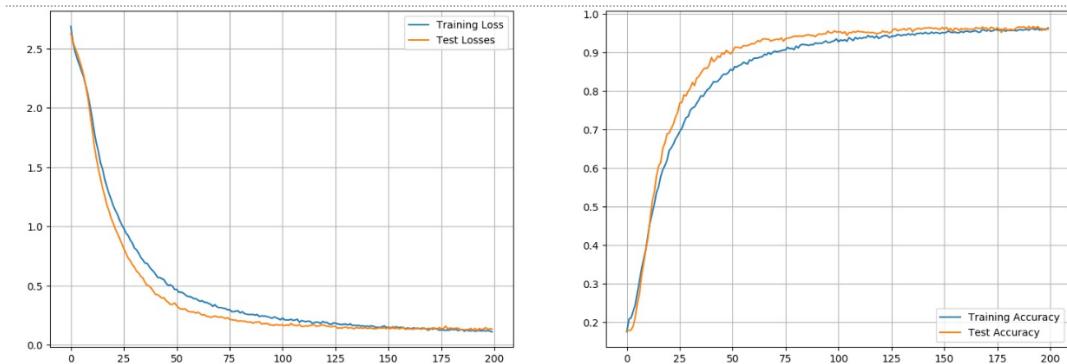
Poichè il task di classificazione richiede in output 16 classi, sono stati modificati gli ultimi livelli lineari impostando a 16 le features di output dell'ultimo livello con classificatore softmax. Per la procedura di training utilizziamo:

- Learning rate: 0.000001
- Momentum: 0.9
- Weight decay: 0.000001

Come loss function utilizziamo cross entropy loss, come funzione di attivazione utilizziamo ReLU, mentre come metodo di learning utilizziamo SGD.

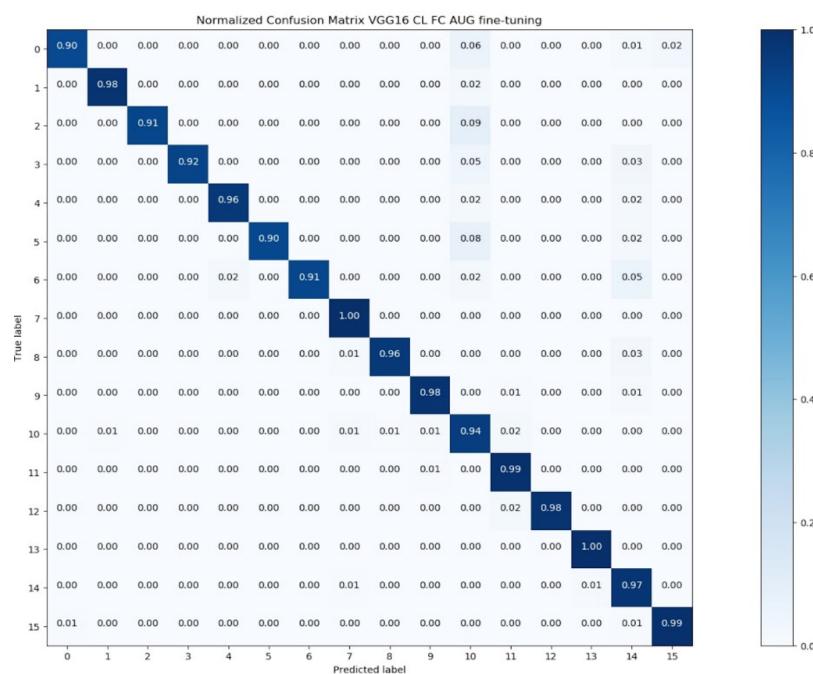
Abbiamo allenato il modello per 400 epoche, ed abbiamo ottenuto i seguenti risultati:

### 3.1.1 Accuracy



Dai due plot, di cui sopra, possiamo osservare la convergenza della validation/training loss e della validation/training accuracy.

### 3.1.2 Matrice di confusione



### 3.1.3 Score F1 e Score mF1

Classe	Score F1
0	0.94
1	0.97
2	0.95
3	0.95
4	0.97
5	0.94
6	0.95
7	0.95
8	0.96
9	0.98
10	0.92
11	0.97
12	0.99
13	0.99
14	0.97
15	0.99
<b>Score mF1</b>	0.96

Con questa architettura abbiamo raggiunto una **accuracy** di **0.97** sul validation.

## 3.2 Regressione

```
VGG(
    (features): Sequential(
        (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): ReLU(inplace)
        (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): ReLU(inplace)
        (4): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
        (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (6): ReLU(inplace)
        (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (8): ReLU(inplace)
        (9): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
        (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (11): ReLU(inplace)
        (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (13): ReLU(inplace)
        (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (15): ReLU(inplace)
        (16): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
        (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (18): ReLU(inplace)
        (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (20): ReLU(inplace)
        (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (22): ReLU(inplace)
        (23): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
        (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (25): ReLU(inplace)
        (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (27): ReLU(inplace)
        (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (29): ReLU(inplace)
        (30): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1), ceil_mode=False)
    )
    (classifier): Sequential(
        (0): Linear(in_features=8192, out_features=2048, bias=True)
        (1): ReLU(inplace)
        (2): Dropout(p=0.25)
        (3): Linear(in_features=2048, out_features=2048, bias=True)
        (4): ReLU(inplace)
        (5): Dropout(p=0.25)
        (6): Linear(in_features=2048, out_features= 4, bias=True)
    )
)
```

The code shows a VGG architecture definition. It consists of two main parts: 'features' and 'classifier'. The 'features' part is a Sequential block containing 30 layers, each either a Conv2d or MaxPool2d layer followed by a ReLU activation. The 'classifier' part is another Sequential block containing 6 layers, each a Linear layer with a ReLU activation and a Dropout layer in between. A red bracket groups the first 30 layers under the label 'frozen', indicating they are not trainable. The last 6 layers are grouped under the label 'fine-tuning', indicating they are trainable.

Poiché il task di regressione richiede in output 4 numeri reali x, y, u, v, sono stati modificati gli ultimi livelli lineari impostando a 4 le features di output dell'ultimo livello.

Per la procedura di training utilizziamo:

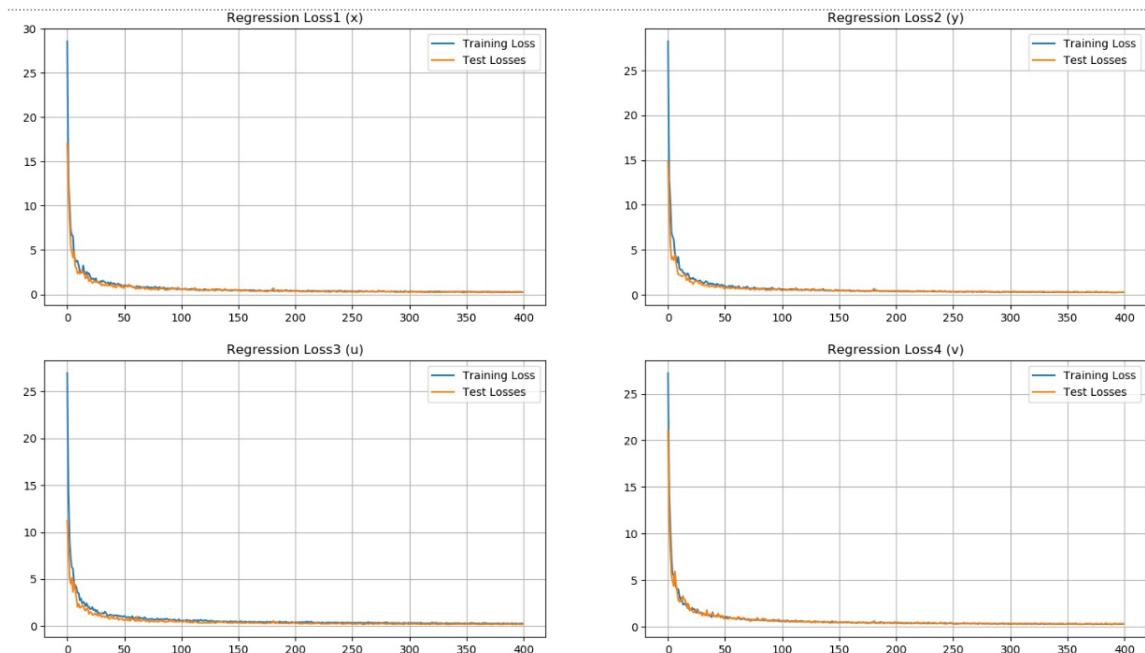
- Learning rate: 0.00001
- Momentum: 0.9
- Weight decay: 0.000001

Come loss function da ottimizzare utilizziamo MSE loss, come funzione di attivazione utilizziamo ReLU, mentre come metodo di learning utilizziamo SGD.

Poiché abbiamo da stimare quattro numeri reali  $x$ ,  $y$ ,  $u$ ,  $v$ , utilizziamo a tale scopo quattro loss functions, una per ogni valore, le quali verranno ottimizzate assieme durante la procedura di training.

Abbiamo allenato il modello per 400 epoche, ed abbiamo ottenuto i seguenti risultati:

### 3.2.1 Loss functions



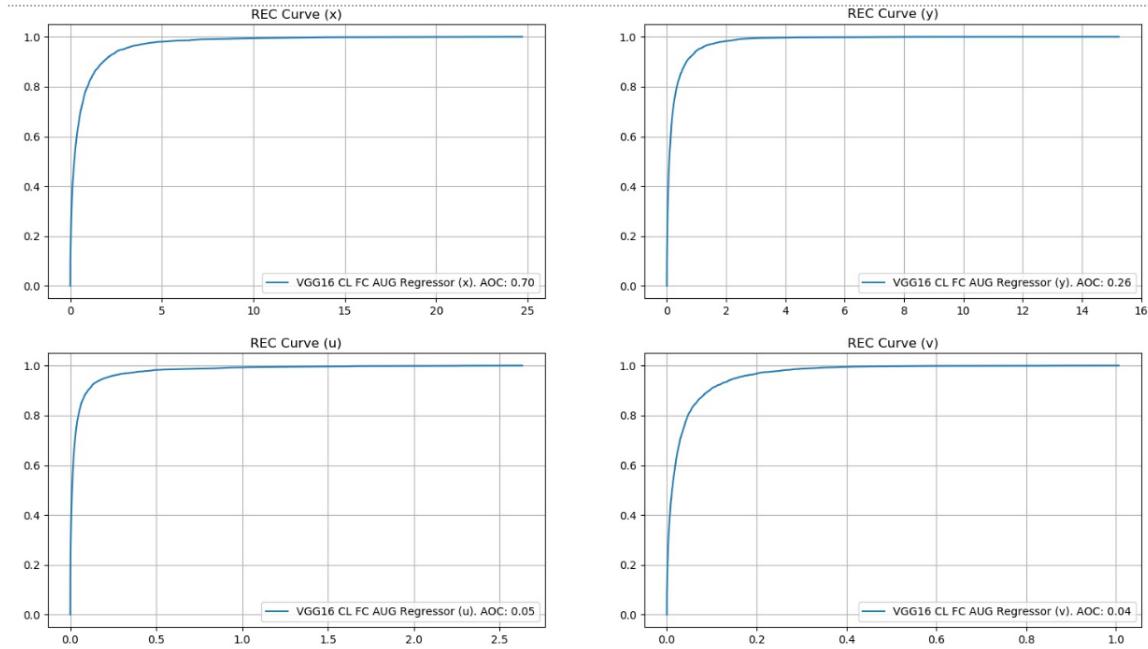
Dai 4 plot che mostrano le loss function, si può osservare come esse tendono a convergenza. A prova di ciò otteniamo valori ottimali con questa strategia.

### 3.2.2 MSE errors

MSE sul parametro X	0.71
MSE sul parametro Y	0.26
MSE sul parametro U	0.05
MSE sul parametro V	0.04

### 3.2.3 REC curves

Le REC curve rappresentano un metodo grafico per valutare la bontà di un metodo di regressione. Inoltre, alla curva è spesso associata l'area sopra la curva (AOC) per offrire una misura dell'errore del metodo.



### 3.2.3.1 RMS error

errore RMS (Root Mean Square) medio e mediano relativo a posizione (in metri) e orientamento (in gradi):

Mean location error	0.79 m
Median location error	0.65 m
Mean orientation error	9.60°
Median orientation error	6.16°

---

# 4

## Conclusioni

In conclusione, ripercorriamo brevemente gli esperimenti condotti che ci hanno permesso di affrontare, nel migliore dei modi, i problemi di localizzazione basati su classificazione e regressione.

Abbiamo iniziato esplorando la neural network più conosciuta, la MLP, ed abbiamo utilizzato questa architettura su entrambi i task.

Dapprima una semplice MLP con un solo livello nascosto, poi con una MLP più profonda con due livelli nascosti. Con entrambe le architetture non abbiamo però ottenuto dei risultati soddisfacenti, ma addirittura ci siamo accorti di una peculiarità: rendendo più complessa l'architettura, aggiungendo più livelli, le performance del modello peggioravano. Questo comportamento indica come l'architettura MLP non riesce a trovare i parametri corretti e non si adatta bene a lavorare su problemi che hanno a che fare con le immagini.

Per questo motivo, ci siamo orientati sulle CNN, le quali permettono di applicare le reti neurali in maniera efficiente al processamento di immagini. Esse sostituiscono le trasformazioni lineari con le convoluzioni, che richiedono meno parametri e permettono di conservare le informazioni spaziali.

Come primo passo abbiamo ripreso l'architettura di rete VGG16 pre-addestrata su “ImageNet” ed eseguito tre diversi approcci di fine-tuning sulla rete: con i primi due abbiamo di volta in volta aumentato i blocchi, e quindi il numero di parametri, coinvolti nell'aggiornamento, mantenendo le low level features di VGG16, mentre con l'ultimo approccio abbiamo inoltre optato per l'aggiunta di data augmentation ed una serie di trasformazioni per provare a generalizzare meglio il modello.

Ci siamo accorti come i risultati dei primi due approcci sono simili tra loro, ma che, all'aumentare dei livelli “sfreezati”, occorreva meno tempo a raggiungere la convergenza.

L'ultimo approccio invece è risultato essere il migliore, in quanto, con “freeze” degli ultimi due blocchi convoluzionali e del blocco fully connected finale insieme alla data augmentation, ci hanno permesso di ottenere i seguenti risultati:

- In classificazione: **validation accuracy = 0.97**

- In regressione:

Mean Location Error	0.79 m
Median Location Error	0.65 m
Mean Orientation Error	9.60°
Median Orientation Error	6.16°

In conclusione, possiamo ritenerci soddisfatti dei risultati ottenuti per quanto riguarda il task di classificazione. Avremmo potuto effettuare altri esperimenti, provando magari ad impostare un learning rate adattivo che si riducesse all'aumentare del numero delle epoche secondo determinate condizioni, o magari provare altre combinazioni di parametri per affinare i risultati oppure un'altra architettura studiata ad hoc, ma il costo computazionale e il conseguente impatto in termini temporali ci avrebbe impedito di rispettare i termini di scadenza per la consegna del progetto. Per quanto riguarda invece il task di regressione, possiamo anche qui ritenerci abbastanza soddisfatti dei risultati ottenuti, anche se con un pò di rammarico nel non aver magari provato ad integrare, assieme alla CNN, una LSTM che avrebbe sfruttato le dipendenze e la natura sequenziale dei dati in input facilitando magari le performance dell'apprendimento e quindi la qualità dei risultati ottenuti.