INTRODUZIONE INTELLIGENZA ARTIFICIALE

A.A. 2020/2021

PROGETTO SMART VACUUM

STUDENTE:

Cristian Cosci

310898

Indice

1. Introduzione

1.1 Descrizione e introduzione al progetto

Vacuum Cleaner è un tipico dominio dell'intelligenza artificiale in cui si immagina di avere una stanza con un certo numero di stanze pulite o sporche e un robot aspirapolvere che si deve occupare della pulizia muovendosi, sotto certi vincoli, fra le stanze. In questo progetto proponiamo una versione estesa di tale dominio in cui le stanze possono essere più o meno sporche, c'è un punto di partenza del robot e un punto di arrivo. Alcune stanze non sono accessibili. La topologia delle stanze si assume predefinita secondo una matrice quadrata passata come input attraverso un'immagine. Si avrà quindi una prima parte in cui viene predisposta una rete neurale in grado di riconoscere dall'immagine di input le dimensioni della matrice e le caratteristiche delle varie stanze.

Si passa poi a tradurre i dati generati dal classificatore in una configurazione dello spazio degli stati, sulla quale dovranno operare degli algoritmi di ricerca, sia informata che non, che si presteranno a risolvere il problema.

Nella terza fase si ha quindi la risoluzione vera e propria, con i diversi tipi di algoritmi, e si noteranno le differenze tra i diversi tipi di ricerca, in ambito di ottimalità e prestazioni in base al problema di ricerca posto in essere.

Nell'ultima fase del progetto è stato implementato un simulatore grafico che mostra la soluzione trovata dagli algoritmi con una simpatica animazione.

Nello svolgere il progetto sono state utilizzate diverse librerie del linguaggio di programmazione python. Il tutto è stato sviluppato mediante la fruizione del servizio Google Colab. Google Colab è uno strumento gratuito presente nella suite Google che consente di scrivere codice python direttamente dal proprio browser. Abbiamo optato per tale soluzione in quanto la consecutiva riproduzione del nostro progetto diventa più semplice nel momento in cui non sono necessarie configurazioni di alcun tipo del proprio sistema. Sarà cosi più immediata una riproduzione in proprio del nostro progetto in qualsiasi dispositivo a prescindere dal sistema operativo e ambiente di programmazione. Google Colab permette anche di allenare la rete neurale mettendo a disposizione un ambiente virtuale fornito da Google stesso. Anche con poca potenza di calcolo del nostro PC personale sarà così possibile eseguire tale progetto, infatti basterà semplicemente avere un browser a disposizione.

Le principali librerie utilizzate sono state :

- **OpenCV** per la lavorazione delle immaggini
- **Pygame** per il simulatore grafico
- **Keras** e **sklearn** per la costruzione della rete neurale
- **Mnist**, per il dataset di immagini per il training

1.2 Obiettivi

Le stanze possono essere più o meno sporche, c'è un punto di partenza del robot e un punto di arrivo. Alcune stanze non sono accessibili. La topologia delle stanze si assume predefinita secondo una matrice quadrata. Partendo da una configurazione iniziale, si chiede di trovare la sequenza di azioni del robot aspirapolvere per avere tutte le stanze pulite e il robot nella posizione codificata come finale. Gli stati iniziale e finale del robot, insieme allo stato iniziale delle stanze, sono passati al sistema tramite un'immagine.

La condizione delle stanze e le posizioni iniziale e finale del robot sono codificate attraverso una lettera maiuscola secondo la legenda:

- S (Start): posizione iniziale del robot

- F (Finish) : posizione finale del robot

- X : stanza non accessibile

- C (Clean) : stanza pulita

- D (Dirty) : stanza sporca

- V (Very Dirty) : stanza molto sporca

1.3 Descrizione formale dei vincoli

- Il robot può compiere **solo un'azione alla volta**, che sia un movimento verso un'altra stanza o la pulizia della stanza corrente.
- il robot **si può muovere solo tra stanze adiacenti**, se accessibili.
- Le stanze Very Dirty (V) saranno trasformate in Dirty (D) dopo un'operazione di pulizia.
- Le stanze Dirty (D) saranno trasformate in Clean (C) dopo un'operazione di pulizia.
- Le stanze Closed (X) sono inaccessibili, quindi non permettono l'accesso al robot.
- La posizione iniziale (S) e la posizione finale (F) non hanno caratteristiche sul loro stato, riguardo alla pulizia della stanza.
- Nelle stanze pulite non potrà essere eseguita l'azione di pulizia, ma potranno essere soltanto attraversate.
- Se il robot è adiacente ad uno o più bordi, non potrà effettuare movimenti verso la direzione del bordo.

Estrazione dati di input per la configurazione iniziale della stanza.

- La configurazione della stanza verrà mappata sui dati di un'immagine presa in input da una matrice quadrata (tramite foto).
- Lo **stato iniziale** è composto da una lista che contiene:
  - Una matrice quadrata con gli stati delle varie stanze.
  - Una coppia di valori che rappresentano la posizione del robot nella mappa (in questo caso i valori corrispondono agli indici della posizione della lettera S).
- Lo **stato finale** è ottenuto mediante la trasformazione dello stato iniziale, impostando tutte le stanze sporche (V o D) come pulite (C) e posizionando il robot nella posizione finale (F).

Azioni possibili:

- **Movimento in alto (+1 per la X)**
- **Movimento in basso (-1 per la X)**
- **Movimento a destra (+1 per la Y)**
- **Movimento a sinistra (-1 per la Y)**
- **Pulizia (da V a D o da D a C)**

2. Acquisizione, classificazione e risoluzione

2.1 Interpretazione delle immagini passate in input

2.1.1 Acquisizione delle immagini

L'immagine in questione è rappresentata da una matrice quadrata con lettere all'interno. Le lettere e la matrice possono essere di 2 tipi: scritte a mano o digitali.

Per l'acquisizione e la classificazione degli input sono stati usati i seguenti materiali:

- **Opencv** --> Manipolazione delle immagini.
- **Emnist** --> Dataset di allenamento.
- **Keras** --> Costruzione rete neurale.
- **Numpy** --> libreria python necessaria per l'installazione di Opencv

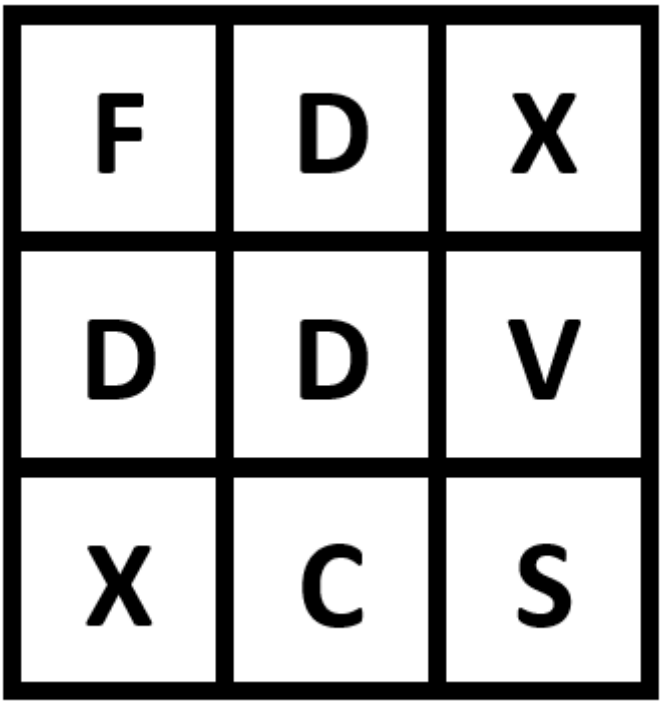
Durante l'acquisizione delle immagini sono state usate diverse funzioni, le più importante sono:

- **trovaMatrice( )** : si occupa della manipolazione dell'immagine per ottenere la matrice principale.
- **trovaQuadrati( ) :** si occupa della rilevazione delle singole celle nella matrice. Queste celle verranno poi singolarmente memorizzate in file appositi, che verranno poi letti dalle funzioni per il riconoscimento.

In "combo" vengono utilizzate le seguenti funzioni come:

- **BlobGriglia( )** : *scorre immagine fino che trova l'oggetto più grande in essa (Matrice*)
- **Disegnalinee( )** : *si occupa di disegnare le linee della matrice*
- **InteresezioneLinee( )** : *calcola gli angoli della matrice*
- **MediaLinee( )**: *approssima le linee trovate in precedenze*
- **CoppieAdicenti( )** : *restituisce coppie di linee adiacenti*
- **DistanzaPunti( )** : *trova distanza tra 2 punti dell'intersezione*
- **DistanzaBordo( )**: *trova distanza tra un punto e il bordo più vicino dell'immagine*
- **BlobCella( ):** *trovava il blob corrispondente alla singola cella*
- **BlobGriglia( )**: *trova il blob della "griglia" principale*
- **PunteggioBlob( )**: *usato da blob cella per decidere qual è il blob migliore da assegnare alla cella.*
- **Ritagliacella( ):** *si occupa di ritagliare il centro di ogni cella della matrice*
- **Alpha_beta_correction( ):** *modifica i valori di luminosità in base a dei valori alpha beta.*
- **Gamma_correction( ):** *usata per codificare i valori di luminosità e rgb dell'immagine.*

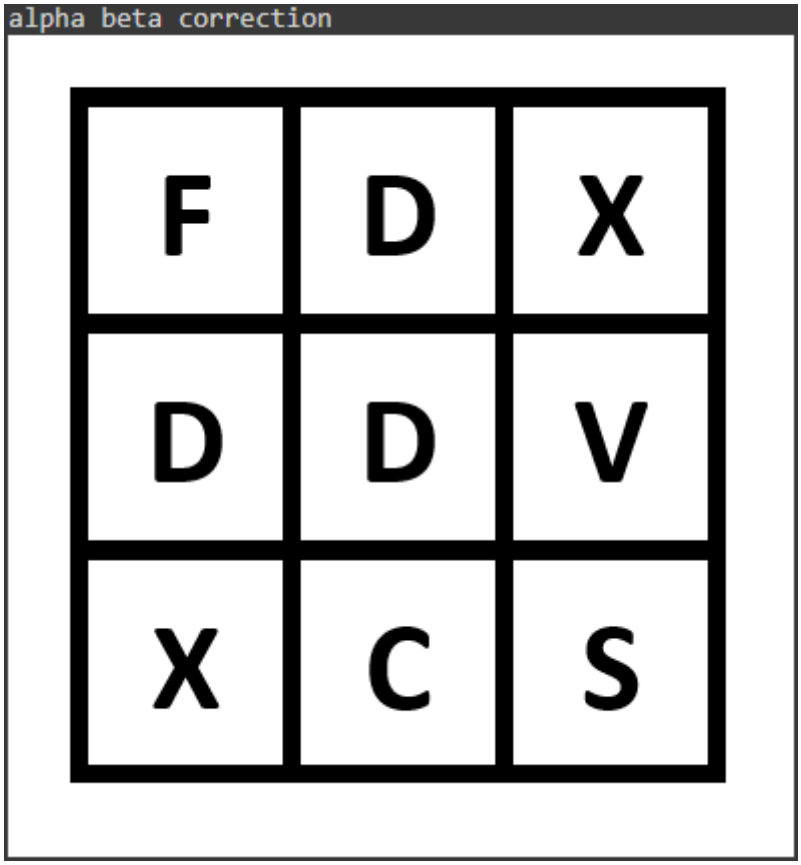
1. ***<u>Per prima cosa quindi verrà passata l'immagine originale in input.</u>***



2. ***<u>Eseguiremo una serie di modifiche grafiche affinchè l'immagine risulti il più utile possibile da analizzare.</u>***

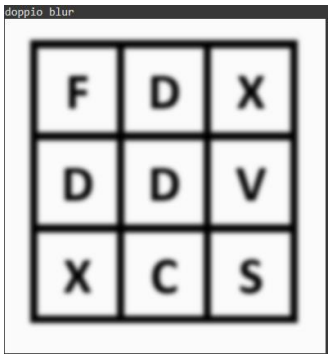
Le modifiche grafiche effettuate sono state:

- **Alpha-beta correction**



La prima modifica grafica effettuata è l'alpha beta correction, che facciamo invocando la medesima funzione, la quale si occupa di modificare i valori di luminosità in base a dei parametri assegnati in fase di pre-elaborazione.

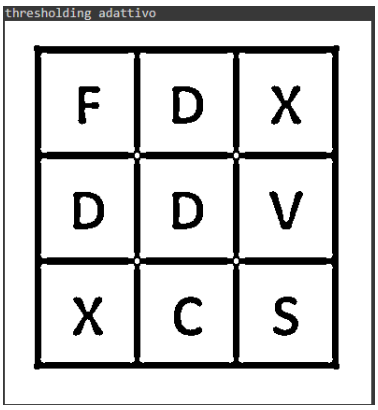
- **Doppio blur**



Nell'operazione di Blur, l'immagine è elaborata con un filtro gaussiano invece del filtro box. Il filtro gaussiano è un filtro low-pass che si occupa di rimuovere le componenti ad alta frequenza dell'immagine.

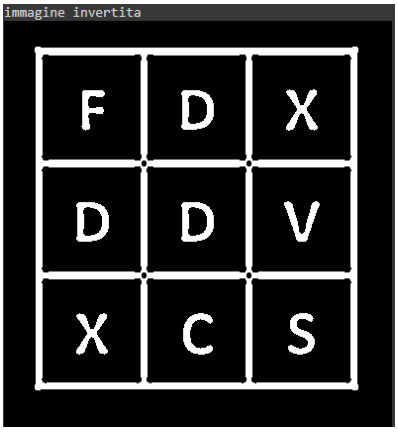
Viene utilizzato 2 volte per appunto ridurre il più possibile le componenti dell'immagine con un'alta frequenza, da qui il nome doppio blur.

- **Threshold**



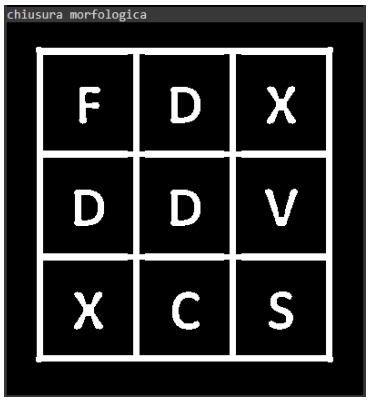
Per l'analisi del immagine abbiamo adottato la tecnica del thresholding (o binarizzazione) dell'immagine, cioè l'applicazione di una soglia lungo una particolare scala di valori, per filtrare in qualche modo un'immagine. In questo caso la soglia è adattiva, ovvero va ad adattarsi all'immagine, senza stabilire un valore di soglia in fase di pre-elaborazione.

- **Inversione colori**



Un’altra tecnica utilizzata è quella di convertire una qualsiasi immagine in scala di grigi (o a colori) in una totalmente in bianco e nero. Questo è molto utile per riconoscere delle forme regolari, dei contorni o delle figure all’interno di un’immagine, o anche per delimitarne e suddividere zone all’interno, per poi essere utilizzate in maniera differente nelle elaborazioni successive.

- **Chiusura morfologica**



La chiusura morfologica è la combinazione di 2 processi.

1. Erosione

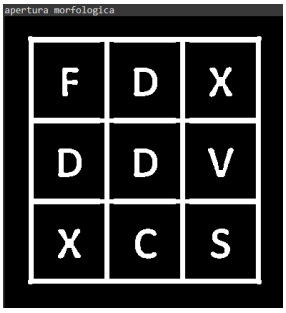
L'idea di base dell'erosione è proprio come l'erosione del suolo, erode i confini dell'oggetto in primo piano (cerca sempre di mantenere il primo piano in bianco). Quindi quello che succede è che tutti i pixel vicino al confine verranno scartati a seconda della dimensione del kernel. Quindi lo spessore o la dimensione dell'oggetto in primo piano diminuisce o semplicemente la regione bianca diminuisce nell'immagine.

2. Dilatazione

È esattamente l'opposto dell'erosione. Qui, un elemento pixel è "1" se almeno un pixel nel kernel è "1". Quindi aumenta la regione bianca nell'immagine o la dimensione dell'oggetto in primo piano aumenta. Normalmente, in casi come la rimozione del rumore, l'erosione è seguita dalla dilatazione. Perché l'erosione rimuove i rumori bianchi, ma restringe anche il nostro oggetto. Quindi lo dilatiamo. Poiché il rumore è sparito, non torneranno, ma la nostra area dell'oggetto aumenta.

All'interno della chiusura morfologica andremo ad applicare **dilatazione seguita da erosione.**

- ## Apertura morfologica



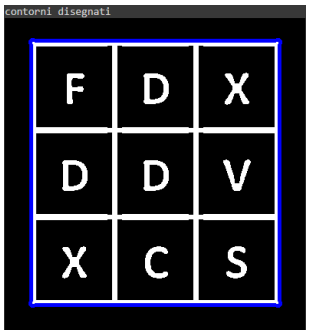
Anche l'apertura morfologica è la combinazione dei processi di **erosione e dilatazione.** In questo caso però viene prima applicato il processo di **erosione seguita da dilatazione.**

3. ***Dopo aver analizzato l'immagine rendendola passiamo alla rilevazione dei bordi, prima della matrice e poi delle singole celle.***

Dopo aver analizzato l'immagine possiamo dunque alterarla a nostro piacimento per arrivare al nostro obbiettivo, ovvero identificare una singola lettera.

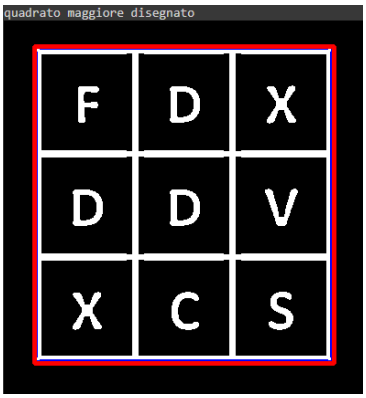
Trasformazioni effettuate:

- **Draw contours**



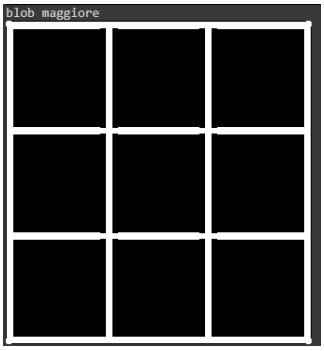
Andiamo quindi a disegnare i contorni della matrice passata in input. Per fare ciò' invochiamo il metodo di openCV drawContours.

- **Quadrato maggiore**



Siccome come nel caso precedente i contorni non sono rilevati perfettamente si passa ad un'approssimazione dei contorni del quadrato della matrice in input.

- **Blob griglia**

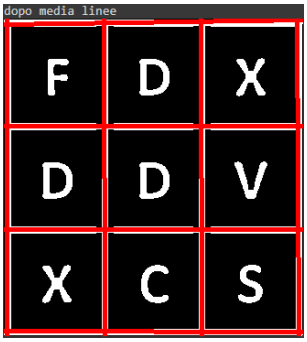


In questo caso viene richiamata l'omonima funzione blob griglia che si occupa di andare ad evidenziare la griglia principale senza le lettere all'interno.

- **Houghlines**

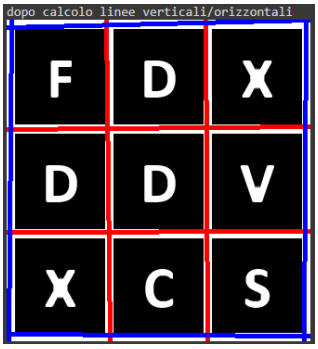
La trasformata di Hough è una tecnica di estrazione utilizzata nel campo dell'elaborazione digitale delle immagini. Utilizziamo questa funzione nella sua forma classica ovvero riconoscere le linee della nostra matrice.

- **Media linee**



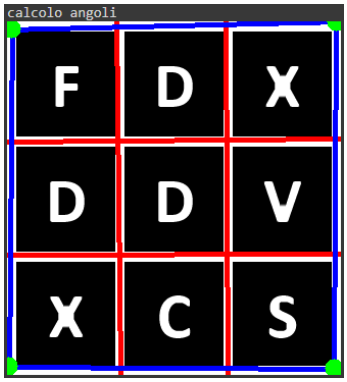
Siccome anche la trasformata di Hough forniva una stima approssimativa delle celle si è deciso anche qui di fare un'approssimazione mediante la medesima funzione denominata mediaLinee che come specificato in precedenza va a fornire una stima più precisa delle linee trovate.

- **Linee orizzontali e verticali**



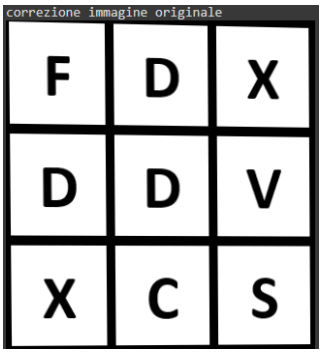
Successivamente si è deciso di andare a differenziare le linee verticali da quelle orizzontali per cercare di ricostruire in modo più preciso possibile la struttura della matrice, ma anche per evidenziare le differenze.

- **Calcolo angoli**

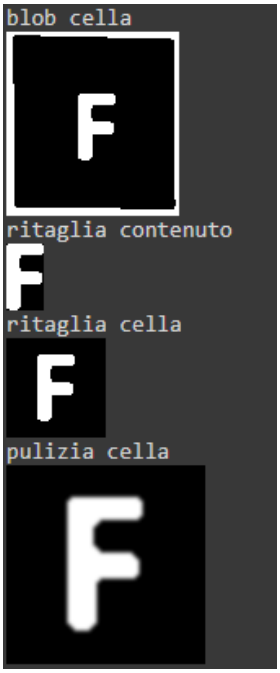


Infine vengono evidenziate le intersezioni degli angoli della matrice.

- **Output finale**



Siccome le matrice principale è solo una rappresentazione delle celle più piccole, si sono adottate le stesse tecniche di analisi anche su ogni singola cella più piccola andando a fornire i seguenti output



Andando così a raggiungere il nostro obbiettivo ovvero quello di leggere ogni lettera contenuta nelle singole celle.

2.1.2 Classificazione

Per la classificazione è stato implementato una rete neurale, nello specifico un cnn descritta qui in seguito. Le funzioni principali sono:

- **creaModello( ):** si occupa del caricamento del dataset, del pre-processing dei dati, creazione e allenamento del modello
- **previsioni( ):** si occupa del riconoscimento delle lettere nelle celle, in base al modello precedentemente creato. Inoltre, questa funzione richiama appositi metodi per la creazione di stato iniziale e stato goal.

Il modello prevede la creazione di una **rete neurale convoluzionale,** basata sui dati del dataset EMNIST.

CNN (convolutional neural networks)

La convolutional neural network rappresenta un'architettura di rete neurale artificiale di grande successo nelle applicazioni di visione artificiale e ampiamente utilizzate in applicazioni che processano media come audio e video. L'applicazione più popolare di rete neurale convoluzionale resta comunque quella di identificare cosa un'immagine rappresenta.

La classica architettura di una cnn è così formata:

```
Input -> Conv -> ReLU -> Conv -> ReLU -> Pool -> ReLU -> Conv -> ReLU -> Pool ->Fully Connected
```

dove **Input, Conv, Relu, Pool e Fully Connected** identificano ciascuno un livello della rete neurale convoluzionale.

In altre parole, abbiamo il:

- **Livello di input**: rappresenta l'insieme di numeri che rappresenta, per il computer, l'immagine da analizzare. Essa è rappresentata come un insieme di pixel. Ad esempio, 28 x 28 x 1 indica la larghezza (28), altezza (28) e profondità (1 o 3, i tre colori Red, Green e Blue nel formato RGB) dell'immagine.
- **Livello convoluzionale (Conv)**: è il livello principale della rete. Il suo obiettivo è quello di individuare schemi, come ad esempio curve, angoli, circonferenze o quadrati raffigurati in un'immagine con elevata precisione. Sono più di uno, e ognuno di essi si concentra nella ricerca di queste caratteristiche nell'immagine iniziale. Maggiore è il loro numero e maggiore è la complessità della caratteristica che riescono ad individuare.
- **Livello ReLU (Rectified Linear Units)**: si pone l'obiettivo di annullare valori negativi ottenuti nei livelli precedenti e solitamente è posto dopo i livelli convoluzionali.

- **Livello Pool:** permette di identificare se la caratteristica di studio è presente nel livello precedente. Semplifica e rende più grezza l'immagine, mantenendo la caratteristica utilizzata dal livello convoluzionale.
- **Livello FC (o Fully connected, completamente connesso)**: connette tutti i neuroni del livello precedente al fine di stabilire le varie classi identificative visualizzate nei precedenti livelli secondo una determinata probabilità. Ogni classe rappresenta una possibile risposta finale che il computer ti darà.

Nel mio caso le immagini da trattare sono relativamente semplici in quanto l'obiettivo è riconoscere delle lettere. Per raggiungere una buona accuratezza ma comunque un semplicità del modello sono bastati 2 livelli convoluzionali, ciascuno seguito da un layer di pool (nel mio caso di maxPooling), seguiti da un livello flatten e 2 dense prima del livello di output. Qui di seguito viene mostrato la struttura della rete seguita da una breve spiegazione.

```
[133] def create_model(num_classes): #creazione modello
        input_layer = layers.Input(shape=(784,), name="input_layer")
        reshaped = layers.Reshape((28, 28, 1), input_shape=(784,))(input_layer)

        cnn_layer = Conv2D(32, (3, 3), activation="relu", name="conv_layer_1")(reshaped)
        cnn_layer = MaxPooling2D(pool_size=(2, 2), name="maxpool_layer_1")(cnn_layer)
        cnn_layer = Conv2D(32, (5, 5), activation="relu", name="conv_layer_2")(cnn_layer)
        cnn_layer = MaxPooling2D(pool_size=(2, 2), name="maxpool_layer_2")(cnn_layer)

        inner_layer = Dropout(rate=0.2, name="drop_layer")(cnn_layer)
        inner_layer = Flatten(name="flatten_layer")(inner_layer)

        dense_layer = Dense(128, activation="relu", name="dense_layer_1")(inner_layer)
        dense_layer = Dropout(rate=0.2, name="drop_layer2")(dense_layer)
        dense_layer = Dense(64, activation="relu", name="dense_layer_2")(dense_layer)

        output_layer = Dense(num_classes, activation="softmax", name="output_layer")(dense_layer)


        model = Model(inputs=[input_layer], outputs=[output_layer])
        model.summary()
        model.compile(loss="sparse_categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
        return model
```

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	[(None, 784)]	0
reshape_8 (Reshape)	(None, 28, 28, 1)	0
conv_layer_1 (Conv2D)	(None, 26, 26, 32)	320
maxpool_layer_1 (MaxPooling2	(None, 13, 13, 32)	0
conv_layer_2 (Conv2D)	(None, 9, 9, 32)	25632
maxpool_layer_2 (MaxPooling2	(None, 4, 4, 32)	0
drop_layer (Dropout)	(None, 4, 4, 32)	0
flatten_layer (Flatten)	(None, 512)	0
dense_layer_1 (Dense)	(None, 128)	65664
drop_layer2 (Dropout)	(None, 128)	0
dense_layer_2 (Dense)	(None, 64)	8256
output_layer (Dense)	(None, 26)	1690

Total params: 101,562
Trainable params: 101,562
Non-trainable params: 0

Come possiamo vedere la struttura della rete è così composta:

- **LIVELLO DI INPUT:** in questo livello si prendono in input i parametri ricevuti, ovvero un array di dimensione 784, ovvero 28 x 28 che corrisponde alla dimensione di ogni singola immagine da analizzare. Viene successivamente aggiunto un livello di reshape che riporta l'array ad un formato matriciale di 28 x 28 x 1. Questo viene fatto per permettere ai successivi livelli di tipo convoluzionale di poter applicare i rispettivi filtri.
- **LIVELLO CONVOLUZIONALE 1 --> parametri:**
  - 32 filtri
  - Dimensione filtro 3 x 3
  - No padding
  - Stride o passo = 1

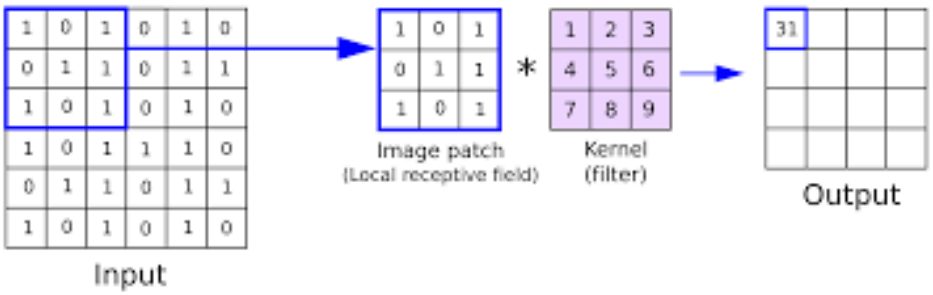
  Per filtro generalmente, si intende una piccola matrice di poche righe e colonne che rappresenta una caratteristica (feature) che il livello convoluzionale vuole identificare, ad esempio le curve o una linea retta.

  Inizialmente per i primi livelli si dice che il filtro rappresenta una caratteristica di basso livello perché identifica semplici oggetti come appunto curve o linee.

  Per un livello convoluzionale il filtro identificherà le curve, per un altro linee orizzontali, per un altro ancora circonferenze, e così via negli ultimi livelli, fino a formare figure complesse che rappresenteranno oggetti più complicati.

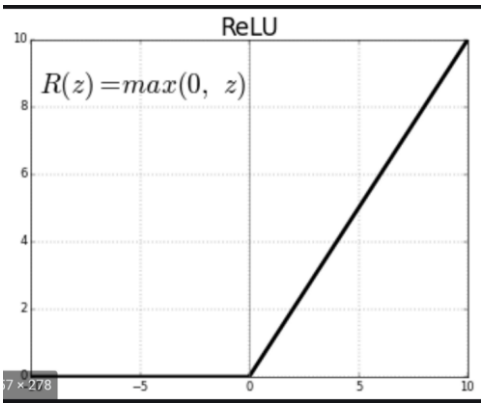
  Essendo i filtri di dimensione 3 x 3 e il passo di uno, l'output (o mappa di attivazione) di questo livello sarà di 26 x 26 x 32.

Un esempio di come lavorano i filtri è rappresentato dall'immagine di seguito.

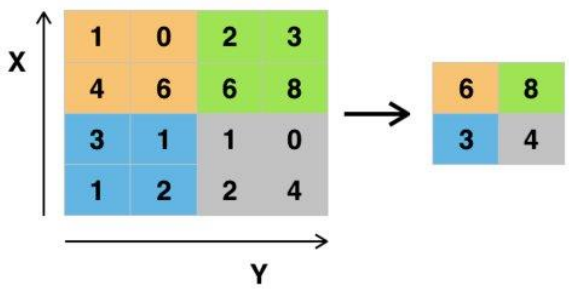


La rete neurale è quindi composta da 2 livelli convoluzionali con funzione di attivazione ReLu.Un neurone con una funzione di attivazione **ReLU** accetta qualsiasi valore reale come suo input, ma si attiva solo quando questi input sono maggiori di 0. Di seguito è possibile trovare un grafico della funzione di attivazione ReLU.

I ricercatori hanno scoperto che con questi livelli le reti neurali convoluzionali funzionano molto meglio perché la rete è in grado di allenarsi molto più velocemente (a causa dell'efficienza computazionale) senza impattare significativamente sull'accuratezza dei risultati.



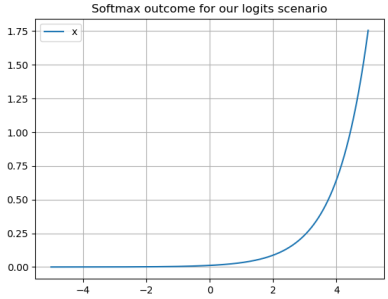
- **LIVELLO DI MAXPOOLING 1(per ridurre l'overfitting):** esso è applicato ad entrambi i livelli convoluzionali nel medesimo modo. Permette di dimezzare la dimensione spaziale(in quanto il filtro utilizzato è un 2 x 2) per ridurre la complessità e migliorare i requisiti computazionali per i livelli successivi.



- **LIVELLO CONVOLUZIONALE 2 --> parametri:**
    - 32 filtri
    - Dimensione filtro 5 x 5
    - No padding
    - Stride o passo = 1

  Questo livello è simile al precedente. Sono stati utilizzati ancora 32 soli filtri in quanto le immagini da trattare sono semplici e non complesse, non si aggiunge così complessità al sistema. La dimensione dei filtri in questo caso è invece di 5 x 5 per generalizzare maggiormente le caratteristiche. Per quanto riguarda la funzione di attivazione si ha sempre la ReLu e non vi è presenza di padding in quando non risulta esserci perdita di informazione che potrebbe andare a degradare l'accuratezza del sistema.

- **LIVELLO DI MAXPOOLING 2** (uguale al precedente)
- **DROPOUT 1 :** si utilizza un dropout del 20% per ridurre l'overfitting e alleggerire il numero di parametri prima del livello flatten
- **LIVELLO FLATTEN :** si utilizza questo livello per riportare i dati elaborati nei livelli convoluzionali (in forma matriciale) a dati processabili per i successivi livelli Dense (array)
- **LIVELLI DENSE :** si succedono infine due livelli dense con 128 e 64 neuroni rispettivamente, con funzione di attivazione ReLu tra i quali vi è un dropout sempre del 20% per non caricare troppo la rete.
- **LIVELLO DI OUTPUT:** anche in questo caso parliamo di un livello di tipo dense, ma con solo 26 neuroni ovvero pari al numero delle classi con cui avremo a che fare. L'altra differenza dai livelli precedenti è che la funzione di attivazione è in questo caso Softmax. Softmax assegna probabilità decimali a ciascuna classe in un problema multi-classe. Quelle probabilità decimali devono aggiungere fino a 1.0. Questo vincolo aggiuntivo aiuta la formazione a convergere più rapidamente di quanto non sarebbe altrimenti. Di seguito è possibile trovare un grafico della funzione di attivazione Softmax. Il modo in cui questo livello completamente connesso funziona è che guarda l'output del livello precedente (che dovrebbe rappresentare le mappe di attivazione di caratteristiche di alto livello) e determina quali caratteristiche sono maggiormente correlate a una particolare classe.



FASE DI TRAINING

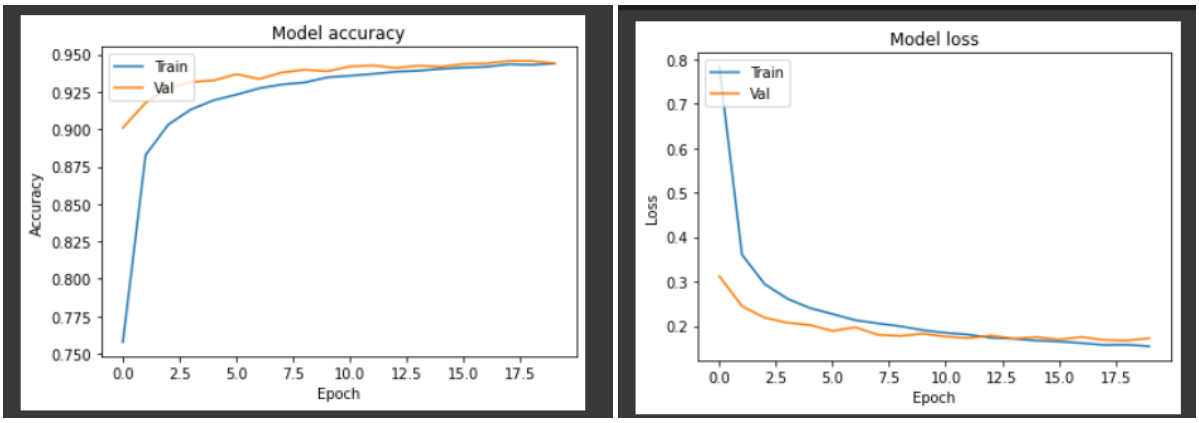
Dopo la creazione del modello si passa all'allenamento, per il quale è stato utilizzato un dataset pre-implementato chiamato EMNIST.

Il set di dati EMNIST è un insieme di cifre di caratteri scritte a mano derivato dal database speciale NIST 19 e convertito in un formato immagine 28x28 pixel e una struttura del set di dati che corrisponde direttamente al set di dati MNIST.

Nello spazio sottostante è riportato la **fase di training** della rete neurale precedentemente creata.

```
Epoch 1/20
975/975 [==============================] - 5s 5ms/step - loss: 1.3151 - accuracy: 0.6066 - val_loss: 0.3122 - val_accuracy: 0.9010
Epoch 2/20
975/975 [==============================] - 4s 4ms/step - loss: 0.3833 - accuracy: 0.8762 - val_loss: 0.2447 - val_accuracy: 0.9175
Epoch 3/20
975/975 [==============================] - 4s 4ms/step - loss: 0.3015 - accuracy: 0.9003 - val_loss: 0.2189 - val_accuracy: 0.9273
Epoch 4/20
975/975 [==============================] - 4s 4ms/step - loss: 0.2642 - accuracy: 0.9116 - val_loss: 0.2075 - val_accuracy: 0.9315
Epoch 5/20
975/975 [==============================] - 4s 4ms/step - loss: 0.2431 - accuracy: 0.9187 - val_loss: 0.2024 - val_accuracy: 0.9325
Epoch 6/20
975/975 [==============================] - 4s 4ms/step - loss: 0.2260 - accuracy: 0.9230 - val_loss: 0.1890 - val_accuracy: 0.9367
Epoch 7/20
975/975 [==============================] - 4s 4ms/step - loss: 0.2061 - accuracy: 0.9293 - val_loss: 0.1972 - val_accuracy: 0.9335
Epoch 8/20
975/975 [==============================] - 4s 4ms/step - loss: 0.2062 - accuracy: 0.9297 - val_loss: 0.1805 - val_accuracy: 0.9378
Epoch 9/20
975/975 [==============================] - 4s 4ms/step - loss: 0.1990 - accuracy: 0.9309 - val_loss: 0.1778 - val_accuracy: 0.9397
Epoch 10/20
975/975 [==============================] - 4s 4ms/step - loss: 0.1896 - accuracy: 0.9350 - val_loss: 0.1829 - val_accuracy: 0.9387
Epoch 11/20
975/975 [==============================] - 4s 4ms/step - loss: 0.1855 - accuracy: 0.9350 - val_loss: 0.1766 - val_accuracy: 0.9419
Epoch 12/20
975/975 [==============================] - 4s 4ms/step - loss: 0.1775 - accuracy: 0.9382 - val_loss: 0.1729 - val_accuracy: 0.9426
Epoch 13/20
975/975 [==============================] - 4s 4ms/step - loss: 0.1704 - accuracy: 0.9399 - val_loss: 0.1787 - val_accuracy: 0.9409
Epoch 14/20
975/975 [==============================] - 4s 4ms/step - loss: 0.1673 - accuracy: 0.9401 - val_loss: 0.1720 - val_accuracy: 0.9424
Epoch 15/20
975/975 [==============================] - 4s 4ms/step - loss: 0.1646 - accuracy: 0.9413 - val_loss: 0.1753 - val_accuracy: 0.9418
Epoch 16/20
975/975 [==============================] - 4s 4ms/step - loss: 0.1595 - accuracy: 0.9425 - val_loss: 0.1696 - val_accuracy: 0.9435
Epoch 17/20
975/975 [==============================] - 4s 4ms/step - loss: 0.1584 - accuracy: 0.9428 - val_loss: 0.1753 - val_accuracy: 0.9439
Epoch 18/20
975/975 [==============================] - 4s 4ms/step - loss: 0.1547 - accuracy: 0.9449 - val_loss: 0.1687 - val_accuracy: 0.9456
Epoch 19/20
975/975 [==============================] - 4s 4ms/step - loss: 0.1543 - accuracy: 0.9438 - val_loss: 0.1674 - val_accuracy: 0.9455
Epoch 20/20
975/975 [==============================] - 4s 4ms/step - loss: 0.1539 - accuracy: 0.9447 - val_loss: 0.1726 - val_accuracy: 0.9441
```

Per pura dimostrazione sono state impostate 20 epoche, in questo modo andando a controllare il grafico di accuracy e loss del modello può essere scelto un parametro opportuno.



Come si può notare dai grafici è possibile scegliere anche un parametro per l'epoch di circa 16 ed ottenere comunque dei buoni livelli sia per accuratezza che tasso di errore. I dati in media infatti si attestano sui valori di:

- **94.5% di accuratezza sui dati di test**
- **16% di tasso d'errore sui dati di test**

Dopo aver creato il modello ed averlo allenato, la rete è in grado di effettuare le previsioni sui dati relativi alle immagini passate in input. Le previsioni vengono effettuate in seguito alle tecniche di elaborazione dell'immagine viste precedentemente. Questo processo avviene dentro la funzione previsioni().

2.2 Traduzione dei dati risultanti dall'analisi delle immagini nella configurazione per il robot

Una volta riconosciute le lettere, e averle classificate in maniera corretta, verrà richiamata la funzione creaStanza(), dove verranno settate le posizioni (x,y) del percorso. Si andrà poi a verificare se sia presente lo stato iniziale "S" che rappresenta il punto di partenza del nostro percorso.

Verrà riportato un errore scritto a video se la matrice non è quadrata o se lo stato iniziale o finale non sono presenti.

Infine viene richiamata la funzione creaStatoGol() la quale va a definire quello che rappresenta lo stato obiettivo, in cui tutte le stanze sono puliti e il robot si trova nella posizione finale.

2.3 Invocazione del risolutore tramite una tecnica di ricerca informata e una non informata e produzione della soluzione

Dopo aver creato lo stato iniziale e lo stato goal con successo, andremo dunque a richiamare il risolutore tramite una tecnica di ricerca informata e una non informata, della classe SmartVacuum, che produce, se esiste, la soluzione del problema, ovvero la sequenza azioni da eseguire per raggiungere lo stato goal.

In seguito verranno poi mostrati una serie di test con relativi risultati a seconda degli algoritmi utilizzati.

3. Implementazione delle classi per la ricerca nello spazio degli stati

Per la risoluzione dei problemi di ricerca sono stati utilizzati i seguenti strumenti:

- **AIMA-python** --> Libreria python
- **Dominio SmartVacuum** --> come classe derivata da Problem
- **Importazioni di altri moduli da aima-python** --> collections, search, heapq

Dopo aver acquisito lo stato iniziale e lo stato goal, e creato la stanza con le varie caratteristiche descritte precedentemente, siamo andati a definire le possibili azioni ed il loro risultato mediante 2 funzioni:

- **Action():** ritorna le azioni possibili in un determinato stato.
- **Result():** torna il risultato di una possibile azione, andando così a creare un nuovo stato dopo aver eseguito ogni azione.
- **Goal_test():** esegue il test gol su ogni nodo selezionato.
- **Path_node():** ritorna il percorso dal nodo start al nodo finish della soluzione

3.1 Ricerca nello spazio degli stati NON INFORMATA

L'obiettivo principale della ricerca non informata è di distinguere tra lo stato target e non target e ignora totalmente la destinazione verso cui si sta dirigendo nel percorso fino a quando non scopre l'obiettivo e segnala il successore. Questa strategia è anche conosciuta come ricerca cieca. Significa che **le strategie non dispongono di informazione aggiuntiva sugli stati, oltre a quella fornita nella definizione del problema.**

3.1.1 Breadth First Search

Come ricerca non informata è stato deciso di utilizzare la bfs *(breadth-first search)* o **ricerca in ampiezza** il suo obiettivo è quello di espandere il raggio d'azione al fine di esaminare tutti i nodi del grafo sistematicamente, fino a trovare il nodo cercato. In altre parole, se il nodo cercato non viene trovato, la ricerca procede in maniera esaustiva su tutti i nodi del grafo.

Il suo funzionamento è così spiegato:

- Si espande prima il nodo radice, poi tutti i suoi successori, poi i loro successori
- Tutti i nodi ad una determinata profondità dell'albero devono essere stati espansi prima che si possa espandere uno dei nodi al livello successivo
- Vi è l'utilizzo di una coda **FIFO** (first-in-first-out) per la frontiera
- L'algoritmo scarta qualsiasi nuovo cammino che porta a uno stato già presente nella frontiera o nell'insieme esplorato
- La ricerca in ampiezza utilizza sempre il cammino meno profondo per ciascun nodo sulla frontiera

La ricerca in ampiezza è:

- **Completa** (in uno spazio degli stati finito)
- **Ottima** perché espande sempre il nodo più vicino alla radice

3.2 Ricerca nello spazio degli stati INFORMATA

La tecnica di ricerca informata utilizza la conoscenza specifica del problema per dare un indizio alla soluzione del problema. Questo tipo di strategia di ricerca impedisce effettivamente agli algoritmi di inciampare sull'obiettivo e sulla direzione della soluzione. La ricerca informata può essere vantaggiosa in termini di costo in cui l'ottimalità viene raggiunta a costi di ricerca inferiori.

Per cercare un costo di percorso ottimale in un grafico implementando la strategia di ricerca informata, i nodi più promettenti **n** vengono inseriti nella funzione euristica ***h(n)***. Quindi la funzione restituisce un numero reale non negativo che è un costo approssimativo del percorso calcolato dal nodo n al nodo di destinazione.

Qui la parte più importante della tecnica informata è la funzione euristica che facilita l'apprendimento della conoscenza aggiuntiva del problema all'algoritmo. Di conseguenza, aiuta a trovare la strada verso l'obiettivo attraverso i vari nodi vicini.

3.2.1 Astar search Algorithm

Come ricerca informata è stato deciso di utilizzare **A*** è un algoritmo di ricerca su grafi che individua un percorso da un dato nodo iniziale verso un dato nodo goal (o che passi un test di goal dato). Utilizza una "**stima euristica**" che classifica ogni nodo attraverso una stima della strada migliore che passa attraverso tale nodo. La valutazione dei nodi viene eseguita combinando ***g(n)*** (il costo per raggiungere il nodo n) e ***h(n)*** (il costo stimato per andare da lì all'obiettivo), si ha quindi la seguente formula:

$$f(n) = g(n) + h(n)$$

L'algoritmo prova per primo il nodo col valore più basso di ***f(n)***

CONDIZIONI PER L'OTTIMALITA': AMMISSIBILITA' E CONSISTENZA

- La prima condizione richiesta per l'ottimalità è che ***h(n)*** **sia un'euristica ammissibile,** cioè non sbagli mai per eccesso la stima del costo per arrivare all'obiettivo. Questo perché le euristiche ammissibili sono "**ottimiste**" per natura, perché pensano che il costo della soluzione sia inferiore a quello reale
- Una seconda condizione denominata **consistenza o monotonicità,** è richiesta soltanto per applicazioni di A* alla ricerca su grafo. Un'euristica h(n) è consistente se, per ogni nodo n e ogni successore n' di n generato da un'azione *a* , il costo stimato di raggiungere l'obiettivo partendo da n non è superiore al costo di passo per arrivare a n' sommato al costo stimato per andare da lì all'obiettivo:

  $$h(n) <= c(n, a, n') + h(n)$$ ---> **disuguaglianza triangolare**

  Ovviamente un'euristica consistente è anche ammissibile.

OTTIMALITA' DI A*

Se ***h(n)*** **è ammissibile** (per ricerca su albero) e se ***h(n)*** **è consistente** (per ricerca su grafo) , A* è ottimamente efficiente. Non c'è alcun algoritmo ottimo che garantisca di espandere meno nodi di A* .

3.2.2 Studio delle euristiche

L'algoritmo A* fa uso della dell'euristica definita nella funzione **heuristic().** Tale funzione ritorna un valore corrispondente ad ***h(n)***

Questo valore è una combinazione (somma i due valori restituiti) di due funzioni qui sotto riportate e costituisce la stima del robot per completare la pulizia e andare verso la posizione finale F.

- **Max_dist_from_F():**

Questa funzione ritorna la distanza di Manhattan dalla posizione attuale del robot a F non tenendo conto di eventuali cammini inaccessibili.

- **Cleaning_operations():**

Questa funzione ritorna il numero di operazioni di pulizia rimaste che devono essere eseguite dal robot per completare il suo lavoro.

4. Simulatore mediante l'utilizzo di pygame

Per l'implementazione di un simulatore per l'esecuzione del piano di azioni calcolato, si è deciso di utilizzare i le seguenti risorse:

- **Libreria pygame**

Inoltre sono state create le seguenti funzioni:

- **ResizeSprite():** *si occupa di ridimensionare gli sprite utilizzati per Luigi e per lo stato delle stanze, in base alle dimensioni della matrice.*
- **SpawnSprite():** si occupa di assegnare ad uno sprite una determinata immagine.
- **Spawn():** *fa comparire lo sprite nella mappa, nella direzione corretta.*
- **AnimazioneIniziale():** *genera l'animazione iniziale del personaggio.*
- **MappaSfondo():** *ridimensiona la matrice della stanza utilizzata, con dimensioni fisse (600x600).*
- **SpawnSporco():** *si occupa di "disegnare" sulla matrice a schermo lo stato di ogni stanza, per poi "disegnare" lo sprite di Luigi.*
- **AnimazioneClean():** *genera l'animazione della pulizia.*
- **AnimazioneFinale():** *genera l'animazione finale del personaggio.*
- **Movimento():** *si occupa di far "muovere" il personaggio in base ai vari stati del cammino ottimale.*
- **Stampa():** *si occupa di stampare l'intero "gioco", quindi mappa, stanze e personaggio*.

Per la simulazione del percorso all'interno della matrice, si è deciso di lavorare con degli sprite presi dal videogame "Super Mario".

La nostra SMART VACUMM è rappresentata da Luigi (personaggio del video gioco Super Mario) che si muove sulla matrice in base ad un percorso calcolato con le operazioni precedenti.

Il percorso ovviamente sarà differente in base alla soluzione trovata dall'algoritmo utilizzato.

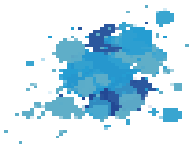
La prima operazione che viene effettuata è un resize degli sprite in base alla dimensione della matrice. Il resize viene effettuato mediante l'omonima funzione. Dopo aver ridimensionato gli sprite a dovere viene fatto comparire il protagonista(Luigi) sulla posizione iniziale data da "S".

Luigi dunque dopo essere "spawnato" nella posizione iniziale compie il suo percorso per arrivare alla posizione finale. Per far muovere il personaggio abbiamo implementato la funzione movimento(), che va a modificare le coordinate (x,y) del personaggio sulla matrice in base ai risultati precedentemente ottenuti nella risoluzione degli algoritmi. Durante il percorso Luigi può incorrere in diverse difficoltà.

Qui sotto è riportata una piccola legenda:

- **Stanza pulita**

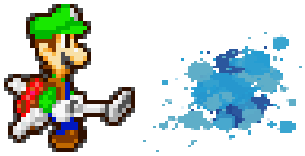
- **Stanza sporca**



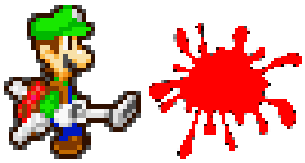
- **Stanza molto sporca**

- **Stanza non accessibile**

- **Pulire una stanza sporca**



- **Pulire una stanza molto sporca**



- **Inizio**

- **Fine**

Quando il nostro compagno Luigi incontrerà una delle seguenti difficoltà si adopererà per risolverla. Per esempio se dovrà pulire una stanza sporca, tirerà fuori l'aspirapolvere e aspirerà lo sporco, tramite la chiamata della funzione AnimazioneClean().

Se la stanza risultasse molto sporca, il nostro personaggio impiegherà 2 "turni" a pulire la stessa stanza. La direzione dello sprite verrà cambiata ogni volta in base a dove si trova lo sporco mediante l'invocazione della funzione spawn(). Quando il robot arriverà all'obiettivo, verrà invocata la funzione AnimazioneFinale().

5. Test

In questa sezione vengono mostrati dei test per mostrare l'efficacia, l'ottimalità e la completezza del programma. I test forniti sono sia con l'utilizzo di matrici artificiali(lettere e matrice fatte digitalmente) e con l'utilizzo di matrici e lettere scritte a mano.

5.1 Matrici e lettere artificiali

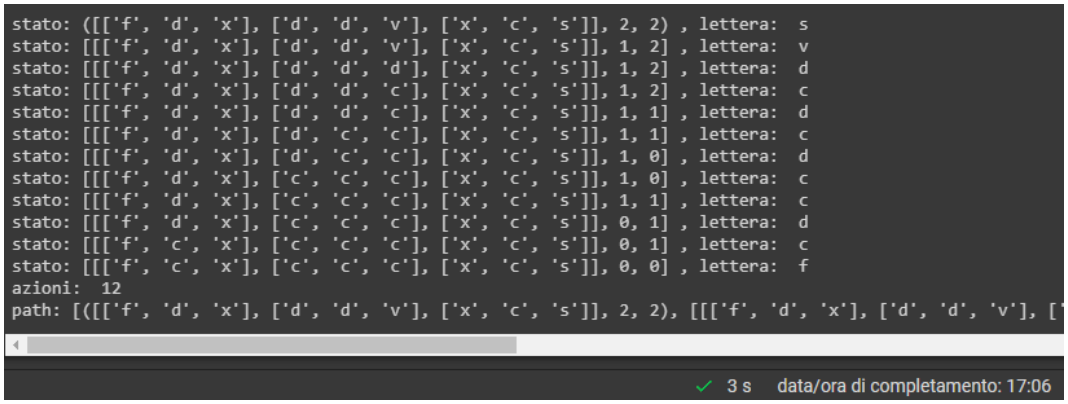
Un primo test è effettuato su una semplice matrice di dimensione 3 x 3

- ***Matrice 3x3***

F	D	X
D	D	V
X	C	S

- **Ricerca non informata.**

In questo caso la **BFS** è riuscita correttamente **in 3 secondi**, trovando una soluzione ottima con **12 azioni.**



- **Ricerca informata**

A* invece è riuscita correttamente a risolvere il problema in un tempo notevolmente minore, trovando una soluzione ottima con **12 azioni.**

```
stato: ([['f', 'd', 'x'], ['d', 'd', 'v'], ['x', 'c', 's']], 2, 2) , lettera:  s
stato: [[['f', 'd', 'x'], ['d', 'd', 'v'], ['x', 'c', 's']], 1, 2] , lettera:  v
stato: [[['f', 'd', 'x'], ['d', 'd', 'd'], ['x', 'c', 's']], 1, 2] , lettera:  d
stato: [[['f', 'd', 'x'], ['d', 'd', 'c'], ['x', 'c', 's']], 1, 2] , lettera:  c
stato: [[['f', 'd', 'x'], ['d', 'd', 'c'], ['x', 'c', 's']], 1, 1] , lettera:  d
stato: [[['f', 'd', 'x'], ['d', 'c', 'c'], ['x', 'c', 's']], 1, 1] , lettera:  c
stato: [[['f', 'd', 'x'], ['d', 'c', 'c'], ['x', 'c', 's']], 1, 0] , lettera:  d
stato: [[['f', 'd', 'x'], ['c', 'c', 'c'], ['x', 'c', 's']], 1, 0] , lettera:  c
stato: [[['f', 'd', 'x'], ['c', 'c', 'c'], ['x', 'c', 's']], 0, 0] , lettera:  f
stato: [[['f', 'd', 'x'], ['c', 'c', 'c'], ['x', 'c', 's']], 0, 1] , lettera:  d
stato: [[['f', 'c', 'x'], ['c', 'c', 'c'], ['x', 'c', 's']], 0, 1] , lettera:  c
stato: [[['f', 'c', 'x'], ['c', 'c', 'c'], ['x', 'c', 's']], 0, 0] , lettera:  f
azioni:  12
✓ 0 s   data/ora di completamento: 17:05
```

Come possiamo notare per matrici di questa dimensione, entrambi i tipi di ricerca riescono a risolvere il problema, ovviamente la bfs riporta una soluzione ottima e come possiamo notare è la stessa della ricerca informata. Questo dimostra come per problemi di piccola dimensione, anche la ricerca informata mediante le euristiche porta ad una soluzione ottima.

Qui di seguito possiamo vedere invece un test con una matrice di dimensione 4 x 4

- ***Matrice 4x4***

F	D	C	X
X	X	C	X
D	D	C	V
X	C	C	S

- **Ricerca non informata**

Qui abbiamo trovato subito i limiti della ricerca non informata, ovvero la **BFS non ha trovato una soluzione** perché le dimensioni del problema risultano eccessive per limiti di memoria disponibile fornita alla macchina di esecuzione di google colab.

- **Ricerca Informata**

A* è riuscita correttamente in un tempo impercettibile a risolvere il problema, trovando una soluzione con **16 azioni** che in questo caso risulta ottima.

Dopo aver provato le soluzione descritte antecedentemente, abbiamo voluto spingere al “limite” la possibilità di calcolo ed elaborazione computazionale del sistema fornito da google colab con i nostri algoritmi.

Siamo così andati a provare:

- ***Matrice 7x7***

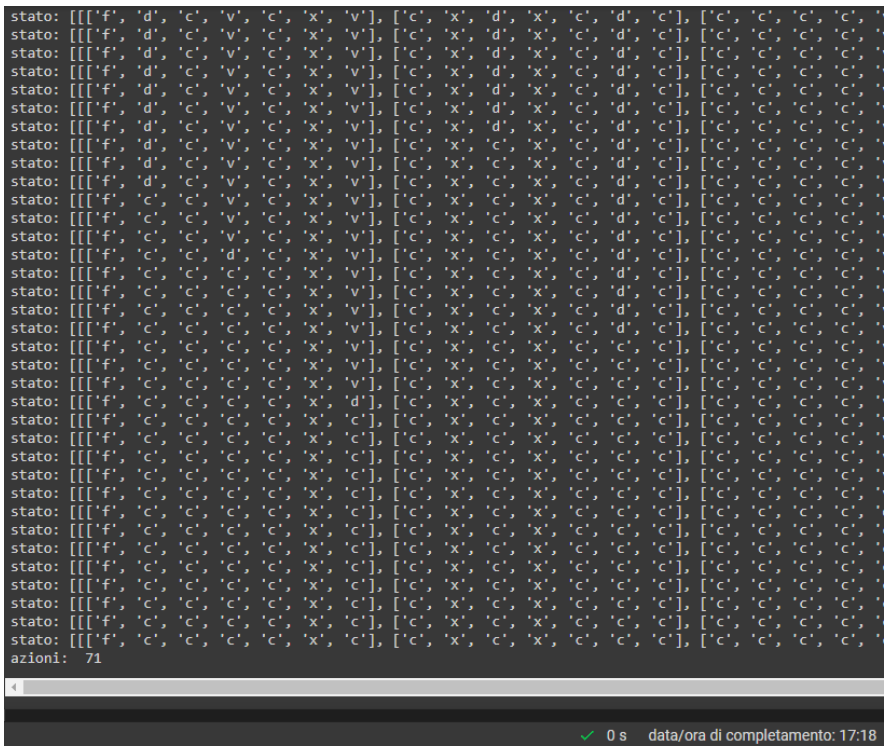
F	D	C	V	C	X
C	X	D	X	C	D
V	D	V	C	X	C
D	X	C	D	C	D
C	X	C	X	C	V
V	D	C	C	D	S

- **Ricerca non informata**

Ovviamente anche in questo caso la **BFS non è terminata con successo** poiché le dimensioni della matrice sono eccessive.

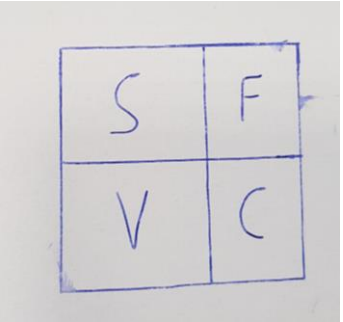
- **Ricerca informata**

A* invece è riuscita correttamente in un tempo minimo a risolvere il problema, trovando comunque una soluzione **con 71 azioni. (in questo caso la soluzione non è ottima ma comunque buona)**



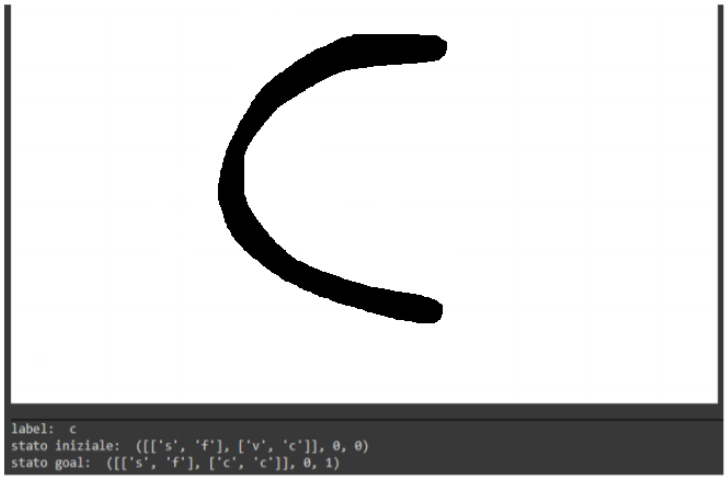
5.2 Matrici e lettere non artificiali

- ***Matrice 2x2***



Con lettere o matrici scritte a mano, il riconoscimento e la classificazione funziona in modo analogo. Bisogna però tenere in considerazione la qualità di queste immagini. Matrici disegnate in maniera poco chiara (per esempio a matita), oppure fotografate a distanza troppo ravvicinata, potrebbero non essere riconosciute in maniera corretta.

In questo caso, le lettere sono chiare, quindi non c'è bisogno di ulteriori accorgimenti come quelli utilizzati nelle prossime matrici.



La ricerca funziona anch'essa in modo analogo a quella utilizzata con le matrici "artificiali", quindi in questo caso BFS e A* offrono ottimi risultati in brevissimo tempo, quasi nullo.

- **Ricerca non informata**

In questo caso la BFS riesce ad essere eseguita correttamente, date le minime dimensioni della matrice, e la ricerca viene eseguita in tempo praticamente nullo, con una soluzione ottima (6 azioni).

```
stato: ([['s', 'f'], ['v', 'c']], 0, 0) , lettera:  s
stato: [[['s', 'f'], ['v', 'c']], 1, 0] , lettera:  v
stato: [[['s', 'f'], ['d', 'c']], 1, 0] , lettera:  d
stato: [[['s', 'f'], ['c', 'c']], 1, 0] , lettera:  c
stato: [[['s', 'f'], ['c', 'c']], 1, 1] , lettera:  c
stato: [[['s', 'f'], ['c', 'c']], 0, 1] , lettera:  f
azioni:  6
path: [([['s', 'f'], ['v', 'c']], 0, 0), [[['s', 'f'], ['v', 'c']], 1, 0], [
```

✓ 0 s

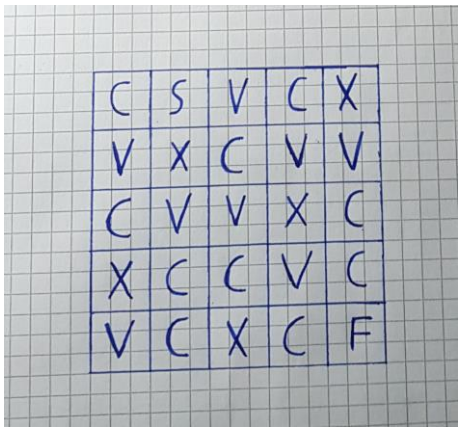
- **Ricerca informata**

Di conseguenza, anche la A* viene eseguita correttamente, sia in termini di tempo, che in termini di soluzione (anche in questo caso 6 azioni).

```
stato: ([['s', 'f'], ['v', 'c']], 0, 0) , lettera:  s
stato: [[['s', 'f'], ['v', 'c']], 1, 0] , lettera:  v
stato: [[['s', 'f'], ['d', 'c']], 1, 0] , lettera:  d
stato: [[['s', 'f'], ['c', 'c']], 1, 0] , lettera:  c
stato: [[['s', 'f'], ['c', 'c']], 0, 0] , lettera:  s
stato: [[['s', 'f'], ['c', 'c']], 0, 1] , lettera:  f
azioni:  6
```

✓ 0 s

- ***Matrice 5x5***

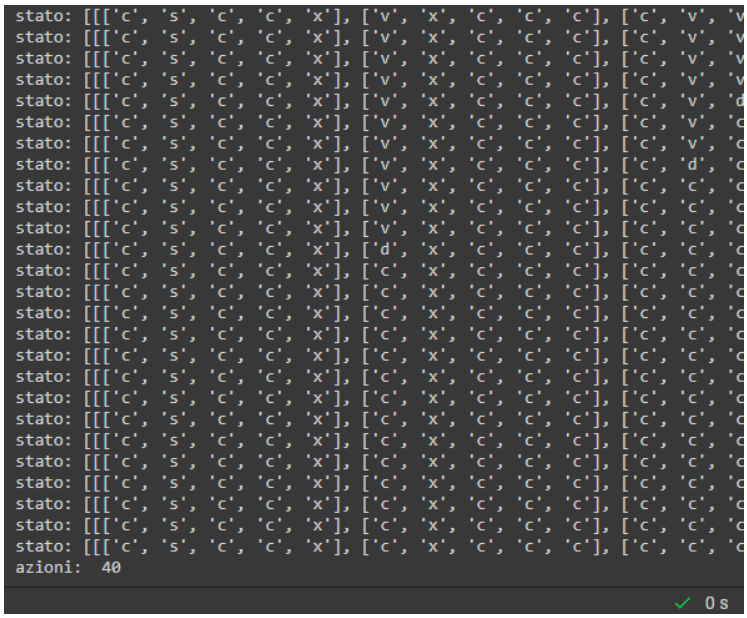


- **Ricerca non informata**

La BFS, date le dimensioni della matrice, non è applicabile.

- **Ricerca informata**

L'A*, a differenza della BFS, riesce ad essere eseguita, in tempi brevissimi, e a trovare una soluzione in 40 azioni.



5.3 Considerazioni finali

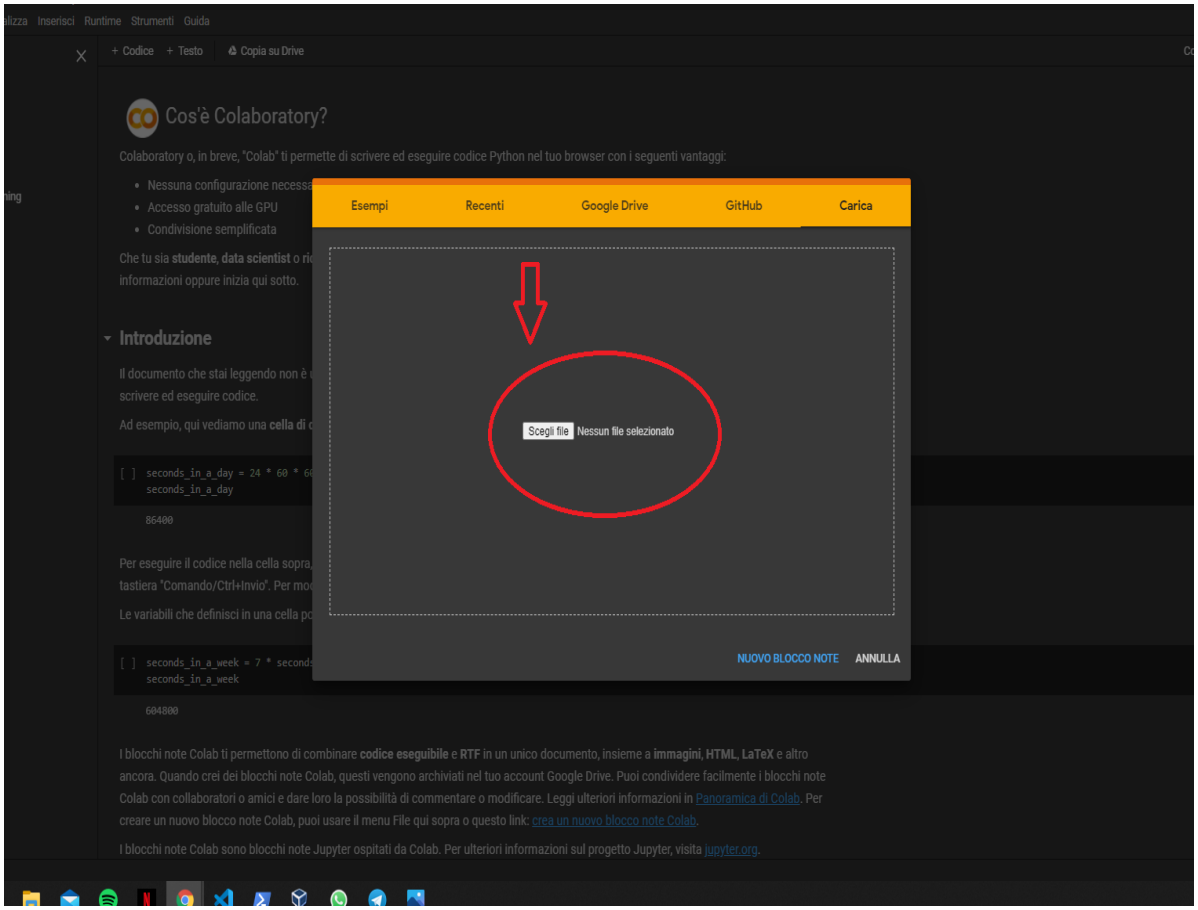
Com'è possibile notare dai risultati ottenuti risultati sovrastanti, A* termina sempre il suo percorso andando a trovare la soluzione ottima ove possibile e una soluzione accettabile per problemi di maggior dimensione. Per quanto riguarda matrici di dimensione superiore alla 7 x 7 la capacità di calcolo e memoria fornita dalla piattaforma non ci rende possibile passare alla risoluzione di questo tipo di problemi. Con ulteriori studi potrebbero comunque essere implementate delle euristiche migliori (necessariamente ammissibili) che potrebbero permettere la risoluzione di problemi di dimensione maggiore anche all'interno della piattaforma di Google Colab.

La BFS invece è ottima e porta ad una soluzione del problema solo quando lo spazio degli stati è notevolmente ridotto (in questo caso non riesce ad andare oltre una 3 x 3)

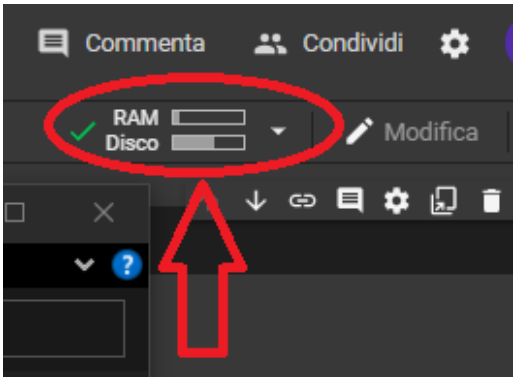
6. Guida per l'esecuzione del codice

In questa sezione viene riportata una guida per permettere a chiunque di replicare ed eseguire il software appena presentato. Insieme alla suddetta relazione sono forniti anche una cartella con un set di matrici di prova, il modello di classificazione e tutte le immagini necessarie per il simulatore grafico.

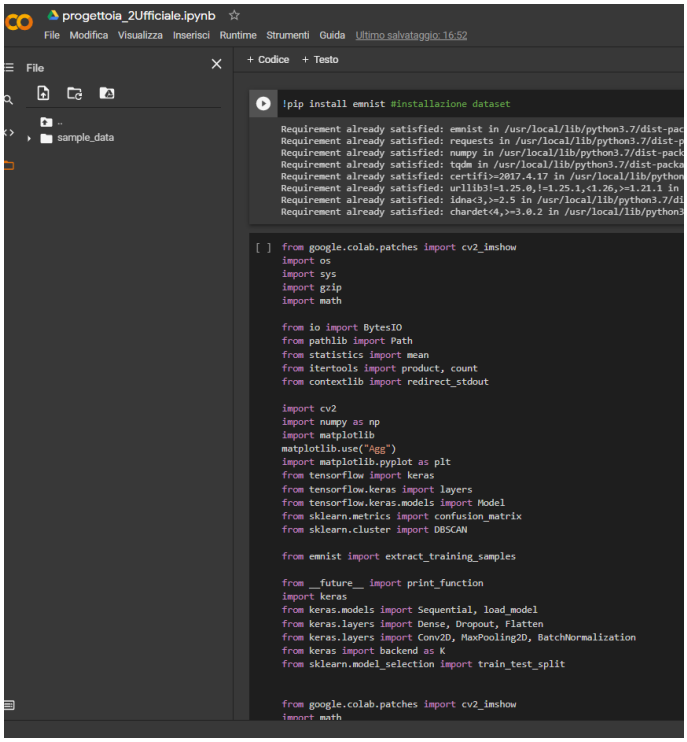
(a) Per prima cosa collegarsi e registrarsi, se necessario, al sito di Google Colab qui riportato. Ci ritroveremo poi di fronte ad una schermata simile, andare su "*carica*" e caricare il file .ipynb fornito.



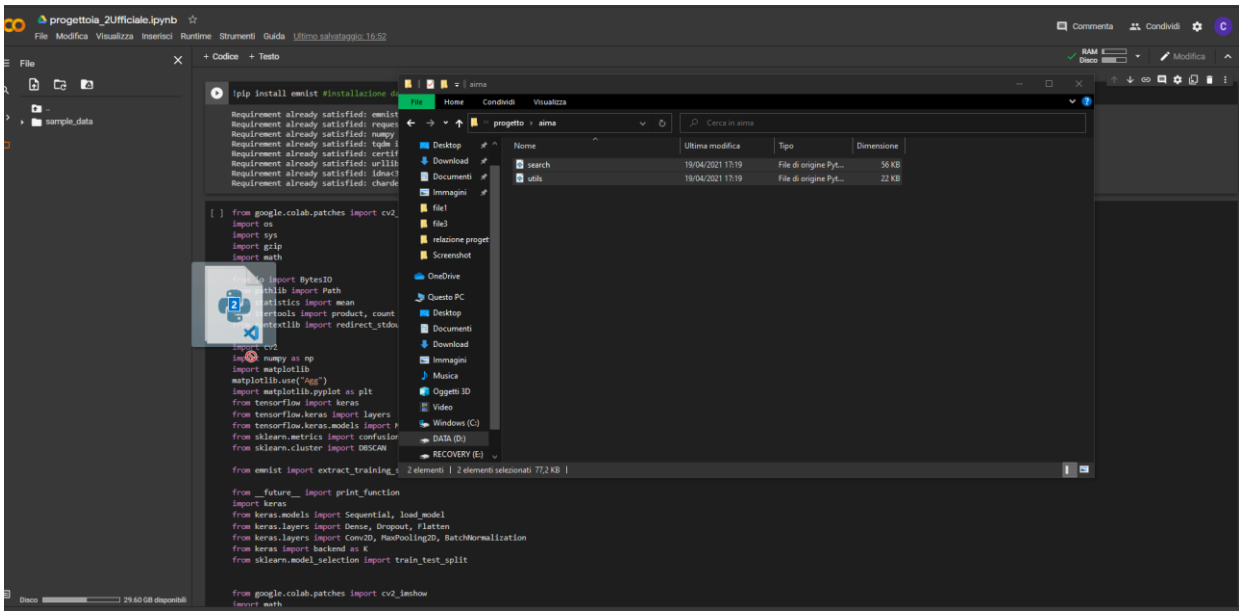
(b) Una volta caricato il notebook avremo di fronte questa schermata, cliccare su “*connetti*” in alto a destra ed attendere la connessione ad una macchina fornita dal servizio. Dovremo avere conferma dell’avvenuta connessione con questa simbologia.

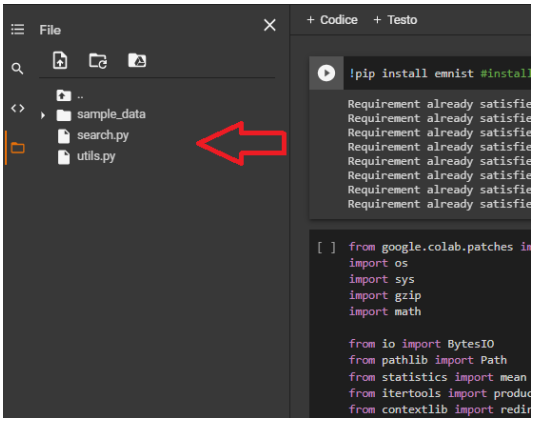


(c) Aprire poi il riquadro sulla colonna di sinistra con l’icona di una cartella. Sarà qui che poi dovremo caricare tutti i file forniti.

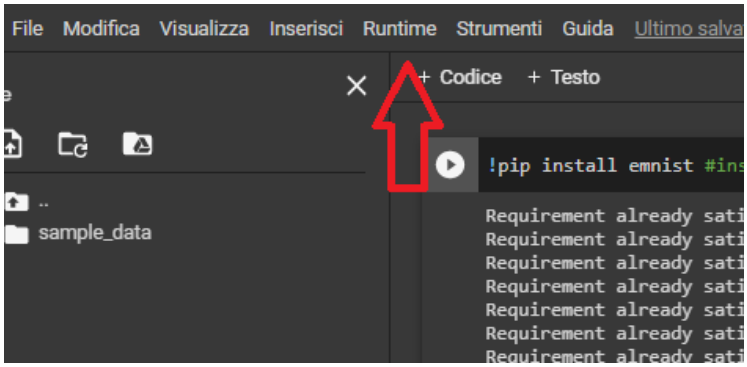


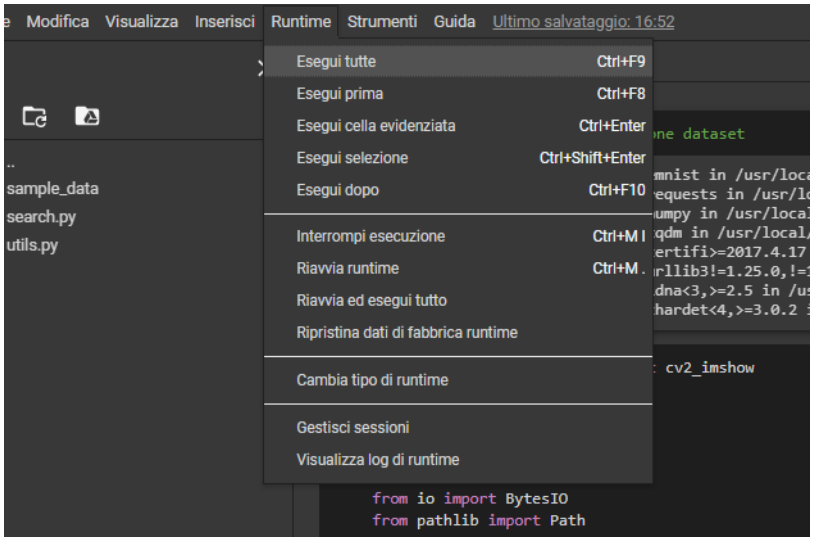
(d) Una volta aperta la prima schermata, trascinare dalla cartella “aima” fornita i due script python “*search” ed “utils”* . Li vedremo quindi caricati come da immagine.

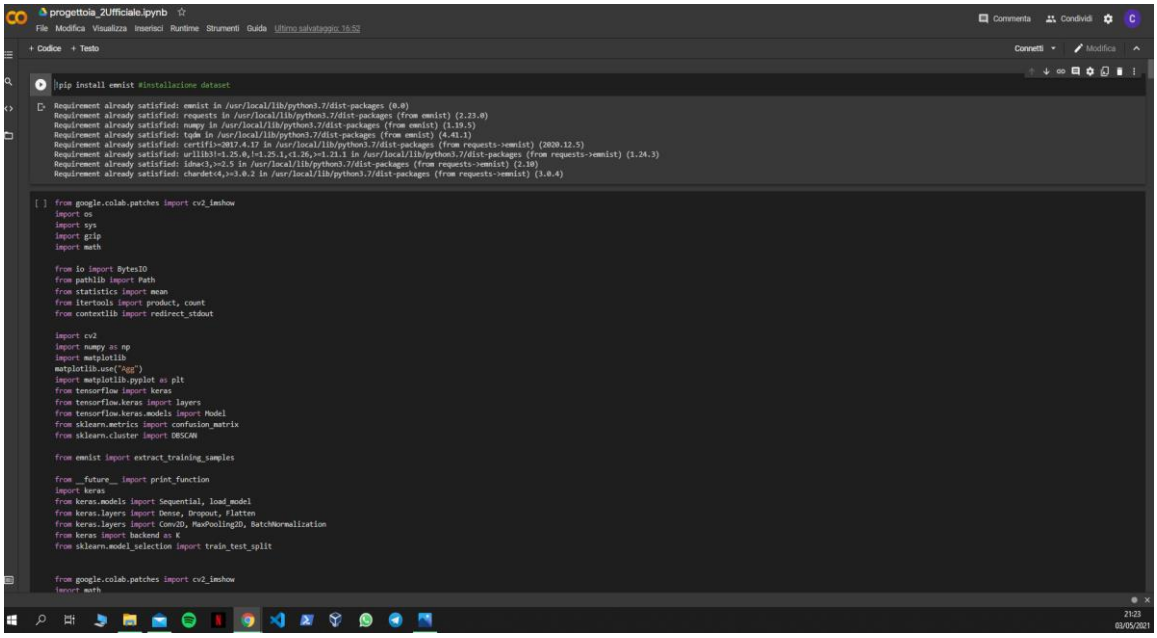




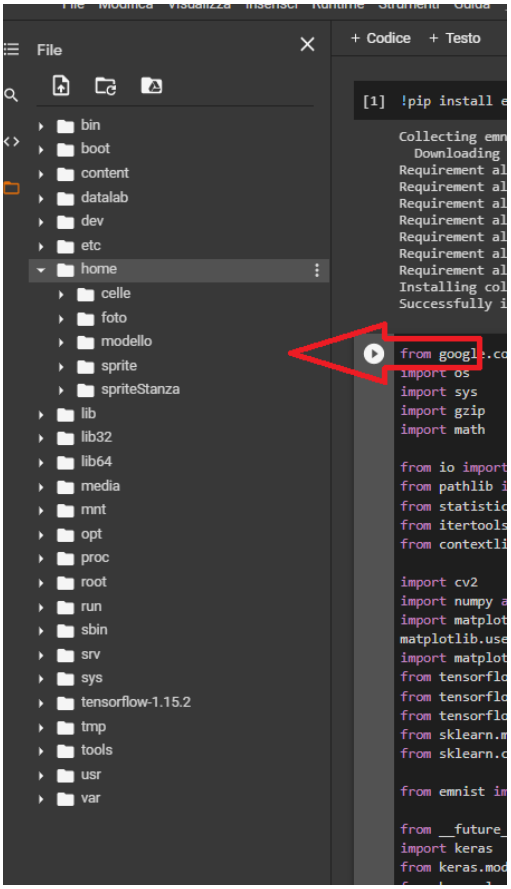
(e) Aprire poi la cartella con i due puntini e ci ritroveremo nel file system della macchina fornitaci dal servizio di google colab. Da questo punto cliccare su “*Runtime”* in alto e poi su “esegui tutte”. A questo punto partirà l’installazione delle principali librerie necessarie e dopo qualche secondo il programma si interromperà per errore. Quello che è appena successo è che lo script ha appena creato nella directory “home” le principali cartelle in cui dovremo inserire i file forniti insieme al programma.


File Modifica Visualizza Inserisci Runtime Strumenti Guida Ultimo salva
+ Codice + Testo
!pip install emnist #ins
sample_data
Requirement already sati
Requirement already sati
Requirement already sati
Requirement already sati
Requirement already sati
Requirement already sati

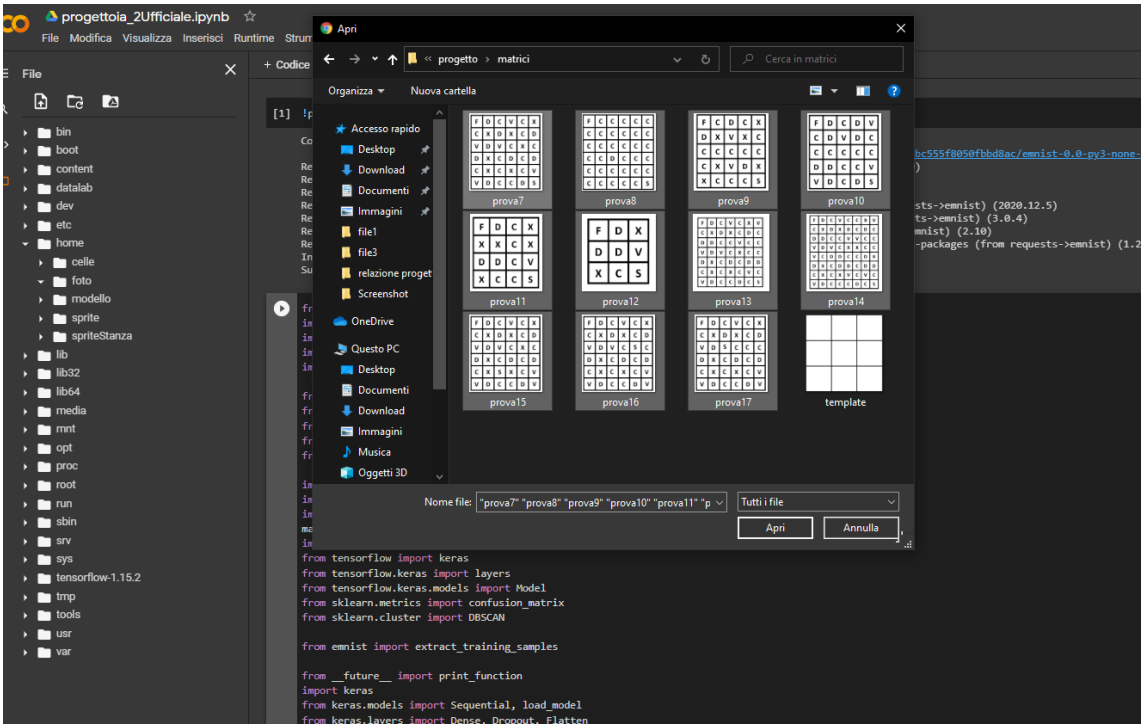

Modifica Visualizza Inserisci Runtime Strumenti Guida Ultimo salvataggio: 16:52
Esegui tutte Ctrl+F9
Esegui prima Ctrl+F8
Esegui cella evidenziata Ctrl+Enter
Esegui selezione Ctrl+Shift+Enter
Esegui dopo Ctrl+F10
Interrompi esecuzione Ctrl+M I
Riavvia runtime Ctrl+M .
Riavvia ed esegui tutto
Ripristina dati di fabbrica runtime
Cambia tipo di runtime
Gestisci sessioni
Visualizza log di runtime
sample_data
search.py
utils.py
from io import BytesIO
from pathlib import Path


progettoia_2Ufficiale.ipynb
+ Codice + Testo
!pip install emnist #installazione dataset
Requirement already satisfied: emnist in /usr/local/lib/python3.7/dist-packages (0.0)
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-packages (from emnist) (2.23.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from emnist) (1.19.5)
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages (from emnist) (4.41.1)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/dist-packages (from requests->emnist) (2020.12.5)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/lib/python3.7/dist-packages (from requests->emnist) (1.24.3)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-packages (from requests->emnist) (2.10)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-packages (from requests->emnist) (3.0.4)
from google.colab.patches import cv2_imshow
import os
import sys
import gzip
import math
from io import BytesIO
from pathlib import Path
from statistics import mean
from itertools import product, count
from contextlib import redirect_stdout
import cv2
import numpy as np
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import Model
from sklearn.metrics import confusion_matrix
from sklearn.cluster import DBSCAN
from emnist import extract_training_samples
from __future__ import print_function
import keras
from keras.models import Sequential, load_model
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D, BatchNormalization
from keras import backend as K
from sklearn.model_selection import train_test_split
from google.colab.patches import cv2_imshow
import math

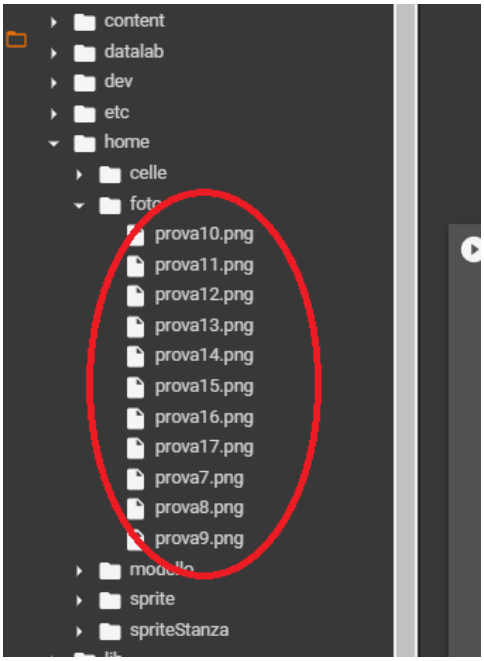
(f) Andare quindi sulla cartella “home” e aprirla, avremo così una schermata simile. La cartella “celle” sarà l’unica in cui non dovremo caricare nulla.



(g) Andare ora ad inserire i vari file partendo dalla cartella foto. Fare click sulla cartella e poi sui tre puntini sul lato destro e cliccare su “carica”. Andate poi a caricare tutte le matrici di prova fornite (si trovano nella cartella “matrici”, tranne template, la quale può essere utilizzata per creare matrici personalizzate di prova).



(h) Verranno così caricate tutte le varie immagini.

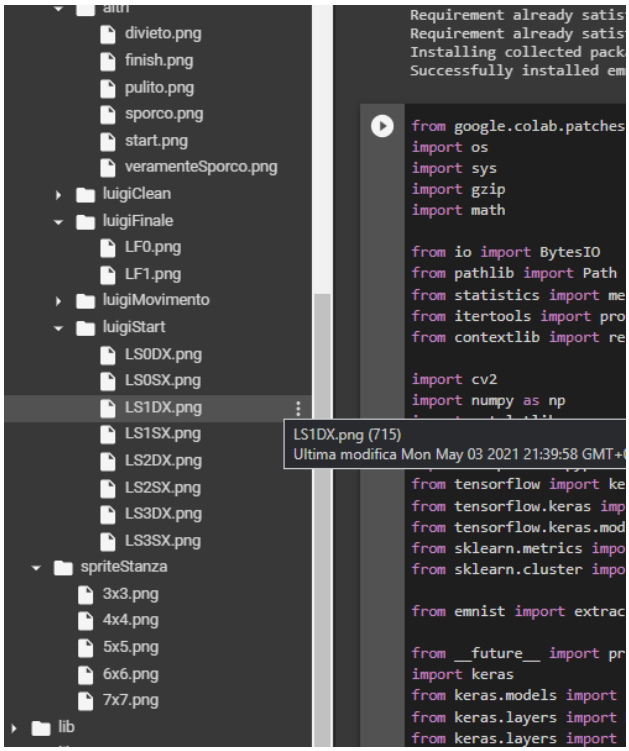


(i) Effettuare la medesima cosa per il modello. Il quale può anche non essere caricato, ma in questo caso dovremo attendere che venga allenato il classificatore alla prossima esecuzione del programma.

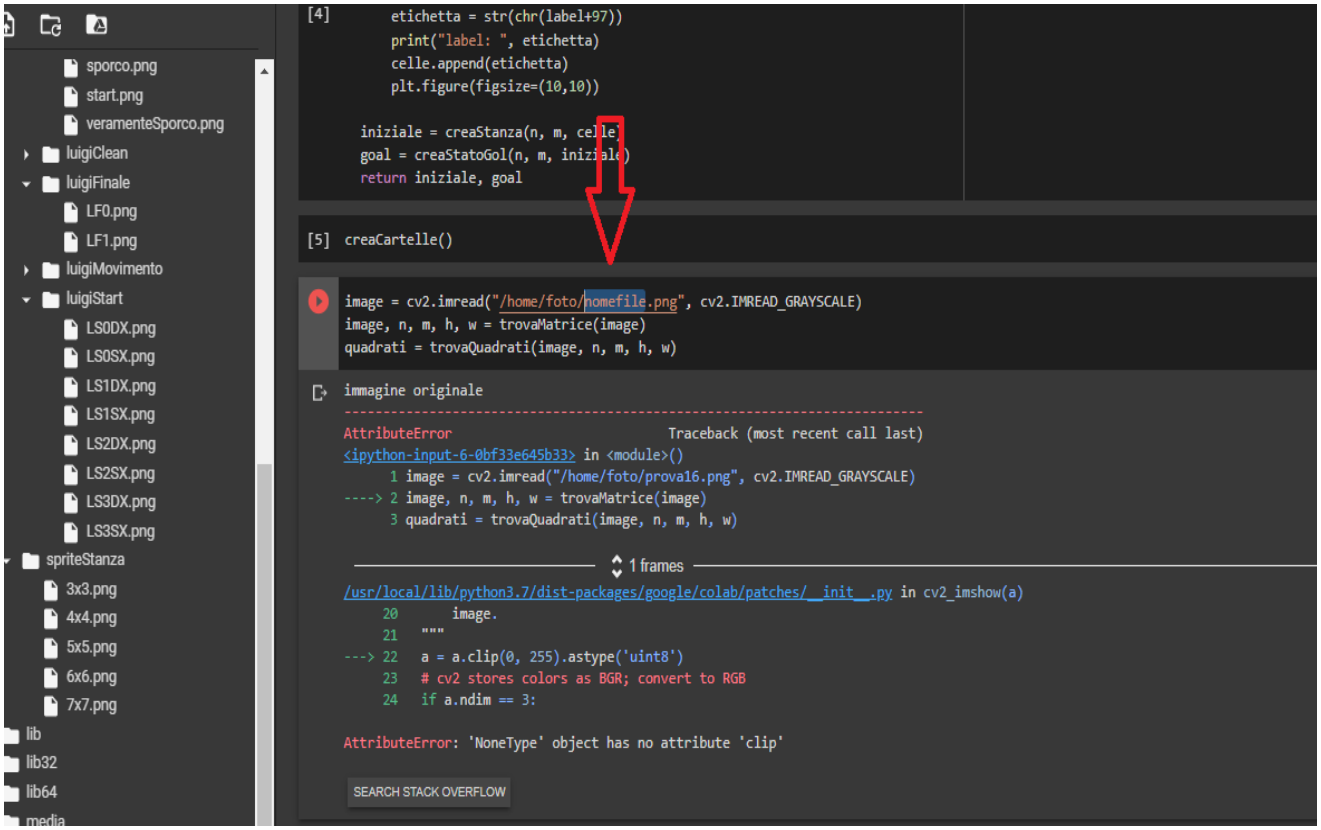
(j) Procedere con il caricamento degli sprite, aprendo la cartella “sprite” ci troveremo di fronte a delle sottocartelle che dovranno essere riempite con i file forniti nelle omonime cartelle, con la stessa e identica procedura vista in precedenza per le matrici. Si prega di prestare attenzione a non dimenticare nessuna immagine e a caricare ogni immagine nella giusta cartella di corrispondenza.

(k) Come ultima cartella, inserire gli sprite delle stanze.

(l) Ci ritroveremo alla fine con una situazione simile.



(m) Ora non resta che andare nella sezione di codice in cui viene presa l'immagine di input da elaborare e inserire quella che più si preferisce.



(n) Una volta fatto aprire nuovamente la finestra "Runtime" e cliccare su "esegui tutte". A questo punto partirà l'esecuzione completa e sarà possibile seguire passo passo tutto il processo. (In alcuni punti verranno installate le librerie non installate precedentemente)

(o) Seguendo tutta l'esecuzione arriveremo nella sezione finale in cui viene eseguito il simulatore grafico di pygame, in questa sezione non bisogna scorrere all'interno della finestra che viene creata alla fine, in quanto perderemo l'esecuzione del movimento, ma spostarsi tenendo il puntatore del mouse all'esterno del riquadro. (Questa piccola nota è dovuta al fatto che il servizio google colab, non mette a disposizione l'utilizzo di un'interfaccia monitor a cui collegarsi mediante la libreria pygame, questo porta all'utilizzo della soluzione proposta, in cui vengono mostrate a schermo le immagini in sequenza per simulare il movimento. Il tutto ovviamente è replicabile senza questo inconveniente in locale ma con degli aggiustamenti al codice relativo al solo sistema di esecuzione google colab.)