Monkey Cleaner

24/01/2021

—

312049 Nicolò “Brüno” Posta
310814 Tommaso “Ladonna” Romani
301838 Nicolò “Eugenio” Vescera

[Typing-Monkeys/MonkeyCleaner]

Obiettivi

L'obiettivo del progetto è quello di costruire un programma composto da 2 parti:

- **`Classificatore`** : servirà per estrapolare le lettere da una matrice scritta a mano che verrà fornita tramite webcam e di inserirle in una struttura dati utilizzabile dell'Algoritmo di Ricerca.
- **`Algoritmo di ricerca`** : dopo aver reso utilizzabili i dati di input l'algoritmo cercherà una soluzione che porti l'agente a pulire tutte le stanze e a raggiungere la posizione finale nel minor numero di azioni possibili.

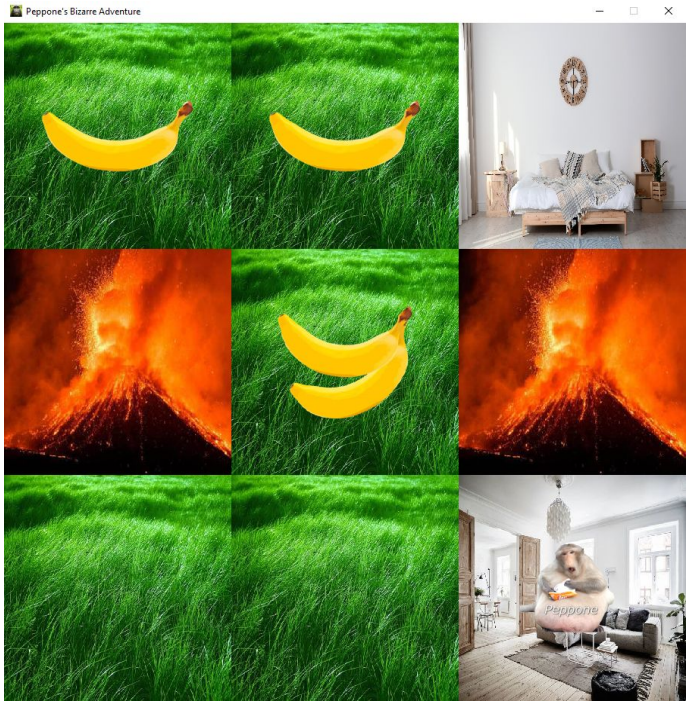
Entrambe le parti sono state implementate in 2 modi diversi:

- Per quanto riguarda il classificatore abbiamo utilizzato:
  - un approccio con **`Knn (K-Nearest Neighbours)`**
  - uno con **`Ann (Artificial Neural Network)`**
- Per l'Algoritmo abbiamo usato:
  - uno di ricerca informata (**A***).
  - uno di ricerca non informata (**BFS**).

Descrizione Formale

Nel nostro progetto abbiamo sostituito l'idea dell'aspirapolvere con quella di una scimmia che mangia banane mantenendo la struttura del problema identica.

Rappresentazione del problema



L'ambiente in cui si muove l'agente è composto da una matrice quadrata (n x n) le cui celle possono assumere uno dei seguenti valori :

- **`S`** : "`Start`" è la cella da cui l'agente partirà.
  - Rappresentata graficamente da un `Salotto Scandinavo`.
- **`F`** : "`Finish`" è la cella in cui l'agente si dovrà trovare al termine dell'esecuzione.
  - Rappresentata graficamente da una `Camera da letto Scandinava`.
- **`X`** : "`Inaccessibile`" una cella attraverso cui l'agente non potrà mai transitare.
  - Rappresentata graficamente da un `Vulcano`.
- **`D`** : "`Dirty`" una cella che richiede una sola azione di clean.
  - Rappresentata graficamente dall'aggiunta di una `Banana` sulla cella.
- **`V`** : "`Very Dirty`" una cella che richiede due azioni clean.
  - Rappresentata graficamente dall'aggiunta di `due Banane` sulla cella.
- **`C`** : "`Clean`" celle che non richiedono alcun tipo di azione da parte dell'agente, ma liberamente attraversabili.
  - Rappresentata graficamente da un `Manto Erboso`.

Definizione del Problema

L'agente a seconda delle celle in cui si trova può eseguire una delle seguenti azioni:

- `Clean` : se la stanza contiene una o più banane (corrispondente a sporca o molto sporca) allora può mangiarne una alla volta.
- `Move` : l'agente si sposta da una cella ad un altra adiacente.

Definizione dei Vincoli

L'agente per operare nell'ambiente deve rispettare i seguenti vincoli:

- ***Vincoli di Movimento*** :
  - Può muoversi al massimo di una cella alla volta.
  - Il movimento deve avvenire tra celle adiacenti, non sono consentiti spostamenti in diagonale.
  - Le celle `Vulcano` (celle inaccessibili) non possono essere raggiunte dall'agente.
  - I bordi della griglia sono dei limiti invalicabili (niente effetto pacman).
  - L'agente parte sempre da una cella iniziale (`Start`) prestabilita.
  - L'agente termina sempre in una cella finale (`Finish`) prestabilita.
  - Le singole azioni di movimento hanno costo unitario ed equivalente.
- ***Vincoli di Interazione*** :
  - L'agente può eseguire una sola azione `Clean` per volta.
  - Le azioni `Clean` hanno costo equivalente (rilevante per la ricerca informata).
  - Le azioni `Clean` possono essere eseguite solo all'interno di celle con una o più banane (di tipo `Dirty` o `Very Dirty`).
- ***Vincoli di costruzione*** :
  - Deve esistere una sola cella inizio per problema.
  - Deve esistere una sola cella fine per problema.
  - Non possono essere costruiti dei "muri" che impediscono l'accesso a celle importanti (`Dirty`, `Very Dirty`, `Finish`).

Ricerca della Soluzione

Rappresentazione dello Stato

A causa di problemi avuti nell'utilizzo dell'implementazione di Aima-Python relativa ad A* è stato necessario rappresentare lo stato nel seguente modo:

- Lo stato corrente viene visto come una lista (`list()`) avente 2 posizioni
    1. Una matrice contenente lo stato attuale del problema.
    2. Una tupla contenente le coordinate attuali (x,y) dell'agente scimmia.
- Il tutto convertito successivamente in una stringa `JSON` (!!!).

Ricerca Non Informata

L'algoritmo di ricerca non informata che abbiamo scelto è la BFS, che è stata presa dalla libreria `Aima-Python`.

Ricerca Informata

Come algoritmo di ricerca informata abbiamo utilizzato `A*` implementando un `Euristica` che si basa sul numero di celle che hanno lo stato attuale diverso da quello di goal (`MissPlacedBananasHeuristica`):

```python
# MissPlacedBananasHeuristica per A*
def h(self, node) -> int:
    # trasforma gli stati iniziali e finali da JSON (!!!)
    # in array utilizzabili
    jstate = json.loads(node.state)
    jgoal = json.loads(self.goal)

    # converte le matrici degli stati in np.array
    # per sfruttare le sue funzioni belle di comparazione :^)
    matrice = np.array(jstate[0])
    goal = np.array(jgoal[0])

    # controlla quanti elementi sono uguali
    cmp = matrice == goal

    # ritorna il numero di elementi differenti
    return np.count_nonzero(cmp == False)
```

Perchè JSON ?

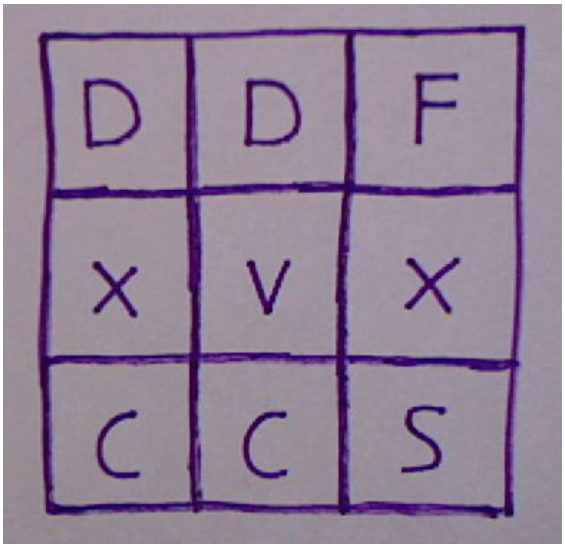
A causa di importanti problemi avuti con l'implementazione di `Aima-Python` relativa ad `A*` abbiamo dovuto trasformare il nostro stato attuale in JSON (!!!) dato che qualsiasi tipo di dato "non base", in python, non può essere inserito all'interno della struttura dati `set()` che Aima utilizzata per costruire la frontiera.

A* Vs BFS

Come ci si aspetterebbe nel confrontare un tipo di ricerca **Informata** e uno **Non Informato**, `A*` è risultato nettamente superiore a BFS sia in termini di memoria che, soprattutto, in termini di tempo.

Per mostrare ancora di più questa differenza basta passare in input una matrice più grande di una 3 x 3 per vedere che la BFS ha un'alta probabilità di impiegare tempi molto lunghi per risolvere il problema.

Di seguito alcuni esempi:

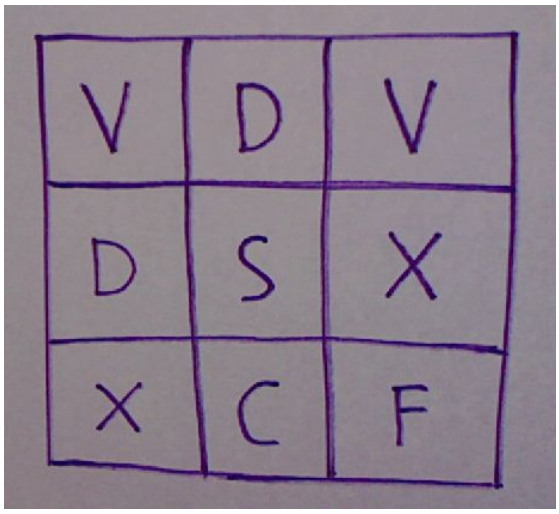


A*

Execution Time: 0.005 s

BFS

Execution Time: 0.215 s

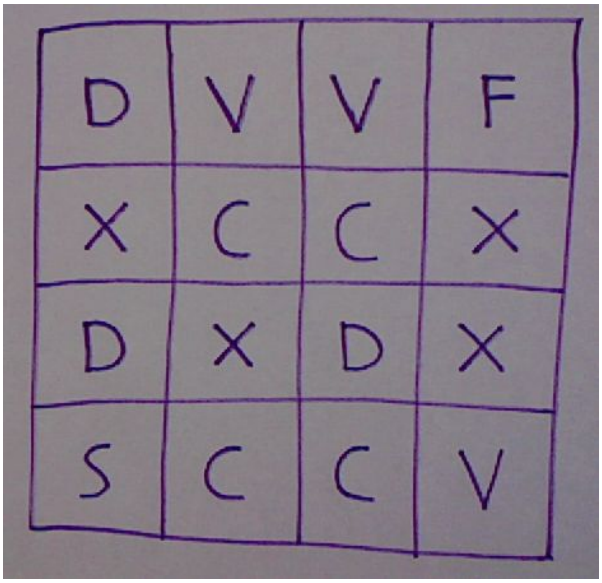


A*

Execution Time: 0.027 s

BFS

Execution Time: 67.684 s



A*

Execution Time: 0.5 s

BFS

Execution Time: > 30 m

Classificazione

Per poter acquisire lo stato iniziale abbiamo utilizzato la libreria openCV che tramite la webcam riconosce in tempo reale i contorni della matrice, delle celle, e delle lettere.

Da qui si ha la possibilità di scattare una foto che verrà elaborata per estrarre gli elementi di input in base al loro livello di gerarchia, che verranno poi passate al classificatore scelto all'avvio del programma.

Abbiamo implementato 2 classificatori:

- **KNN** (**k-Nearest Neighbours**)
- **ANN** (**Artificial Neural Network**)

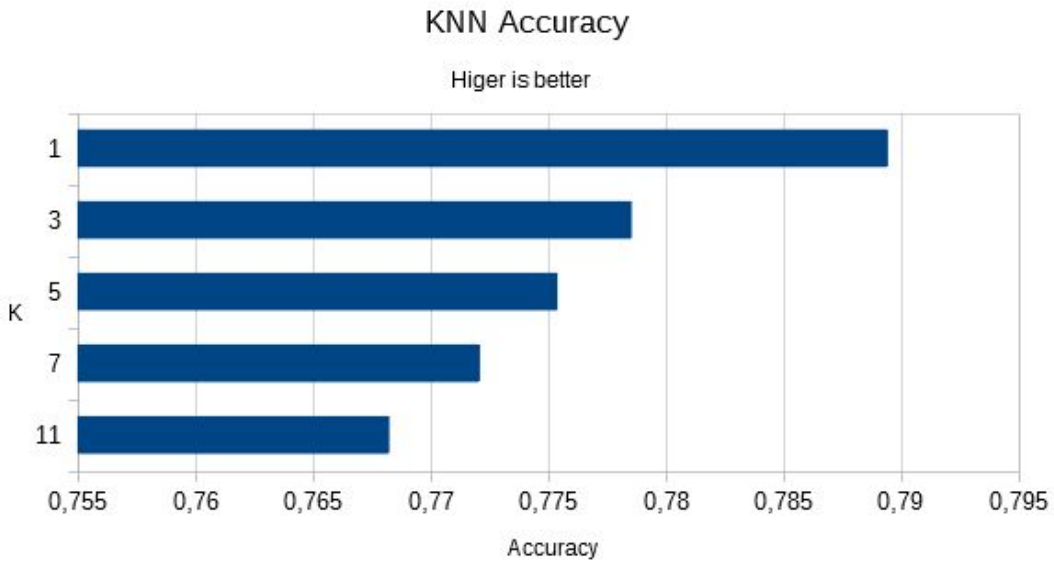
Entrambi hanno utilizzato 2 diversi dataset nei quali sono state rimosse le lettere non necessarie, lasciando quindi solamente "C", "D", "F", "S", "V", "X".

Facendo ciò abbiamo ottenuto più di 93mila record per uno e 30mila per l'altro.

1. Il primo (`A_Z Handwritten Data`, preso da kaggle) è stato utilizzato per allenare i 2 modelli
2. Il secondo (`eMNIST`) è stato utilizzato per stimare la precisione della classificazione.

KNN

Per questo metodo di classificazione abbiamo usato una funzione già presente nella libreria openCV ed abbiamo eseguito vari test per vedere, al variare di K, la variazione della precisione (`Accuracy`) del classificatore.

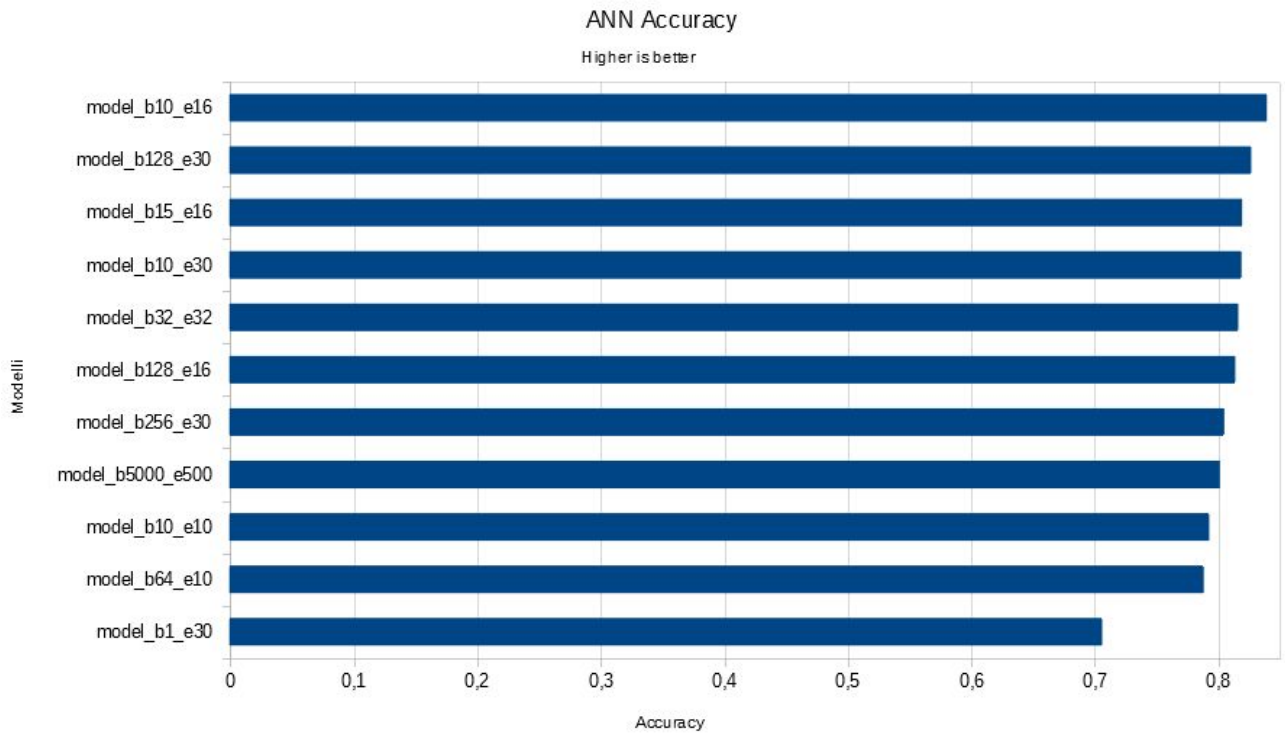


ANN

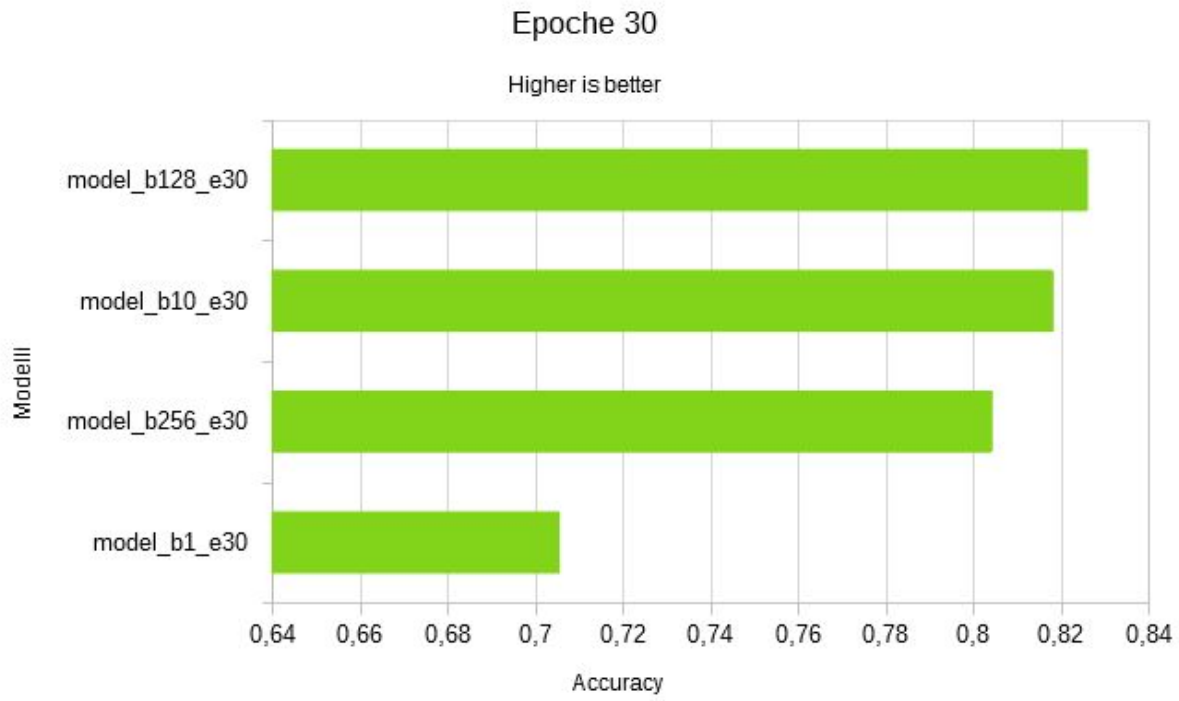
La rete neurale è stata realizzata con un solo hidden layer utilizzando TensorFlow e Keras.

```
def makeANNModel1():
    model  = keras.Sequential()

    model.add(layers.Dense(512, activation='relu', input_shape=(784,)))
    model.add(layers.Dense(256, activation='relu'))
    model.add(layers.Dense(6, activation='softmax'))
    model.summary()
    model.compile(loss="sparse_categorical_crossentropy", optimizer="adam", metrics=["accuracy"])

    return model
```

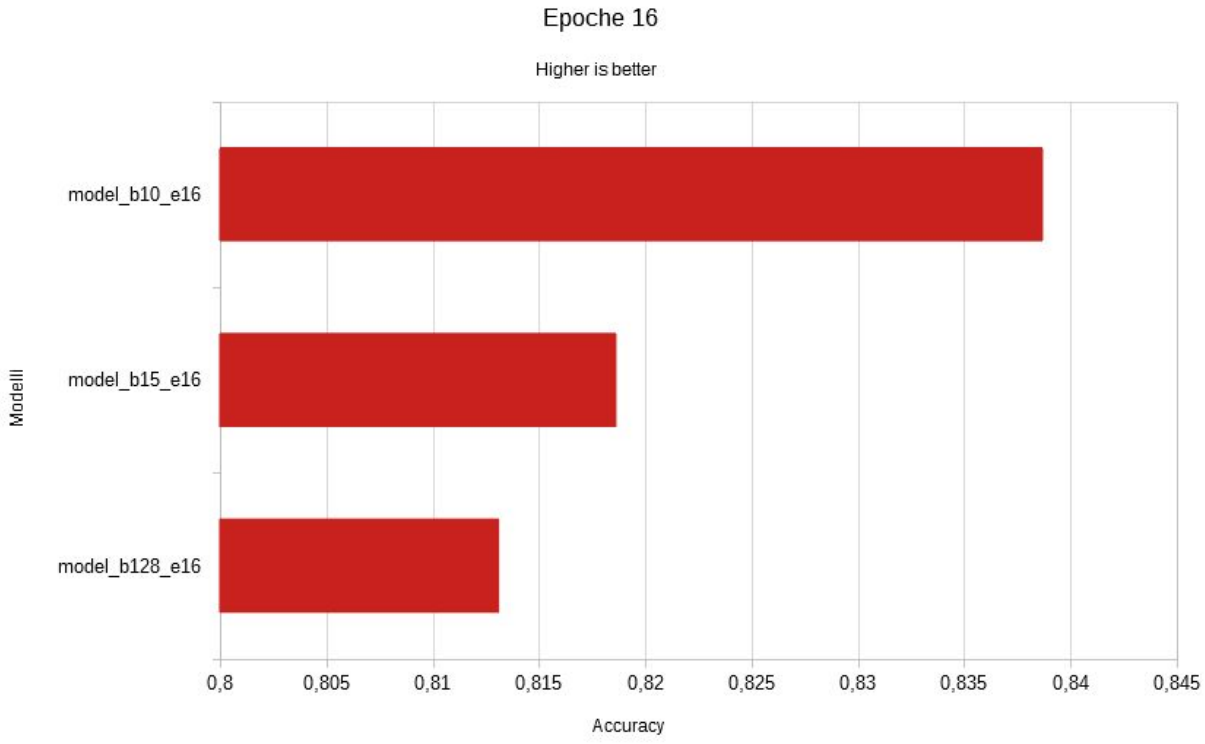
Per la fase di training abbiamo effettuato varie prove variando batch size e numero di epoche, ottenendo così vari modelli che abbiamo provato con il dataset di testing ottenendo i seguenti risultati:



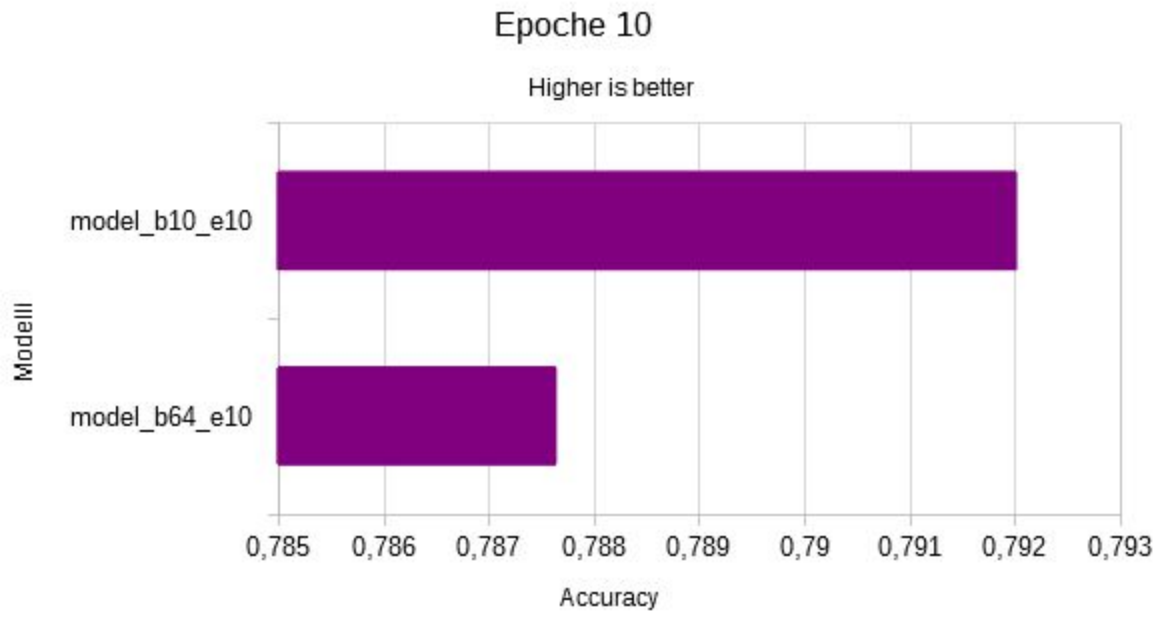
Tutti i modelli a confronto



Modelli con epoche uguali a 30 ma diversi batch size



Modelli con epoche uguali a 16 ma diversi batch size



Modelli con epoche uguali a 10 ma diversi batch size

Nel grafico seguente sull'asse delle y abbiamo i nomi dei modelli in cui **b** è il `batch size` ed **e** corrisponde al `numero di epoche`.

Considerazioni finali

Dai vari test che abbiamo effettuato è risultato che il metodo di classificazione migliore dato il nostro problema è l'ANN e l'algoritmo di ricerca più efficiente è di gran lunga A*.