

DOCUMENTAZIONE CV

Introduzione

Al giorno d'oggi l'utilizzo del cibo rimasto in dispensa può essere un problema relativamente piccolo che si potrebbe purtroppo tradurre in uno spreco dello stesso, trasformandosi in un problema complesso che su larga scala può causare conseguenze nel dominio economico, sociale o ambientale.

Una buona soluzione al riutilizzo del cibo è fornita in questo contesto, in cui viene costruito un modello che elabora dei file ".jpg" raffiguranti un alimento appartenente ad una delle venti categorie elaborate con l'obiettivo di predire la categoria di appartenenza da cui poi ricavare tramite una web query una ricetta che utilizza gli alimenti raffigurati nelle immagini.

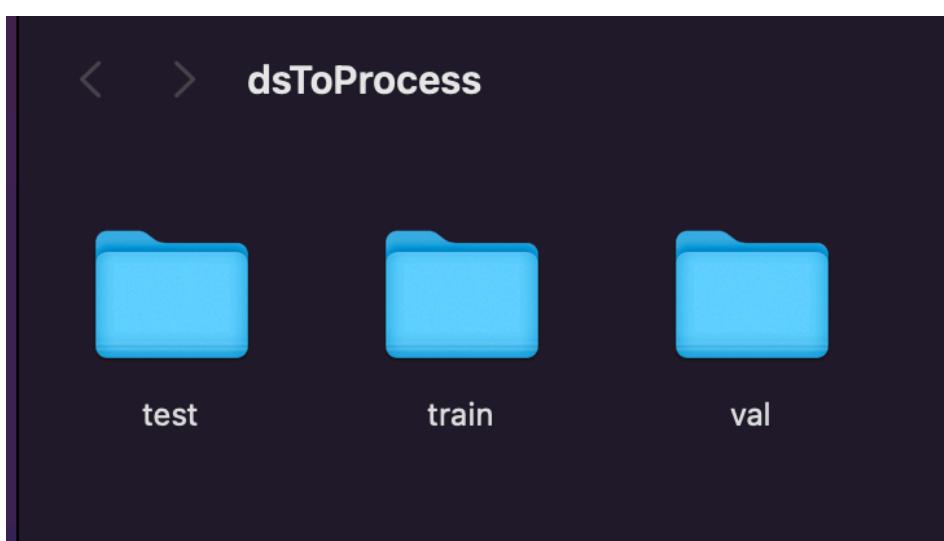
Creazione e modifica del dataset

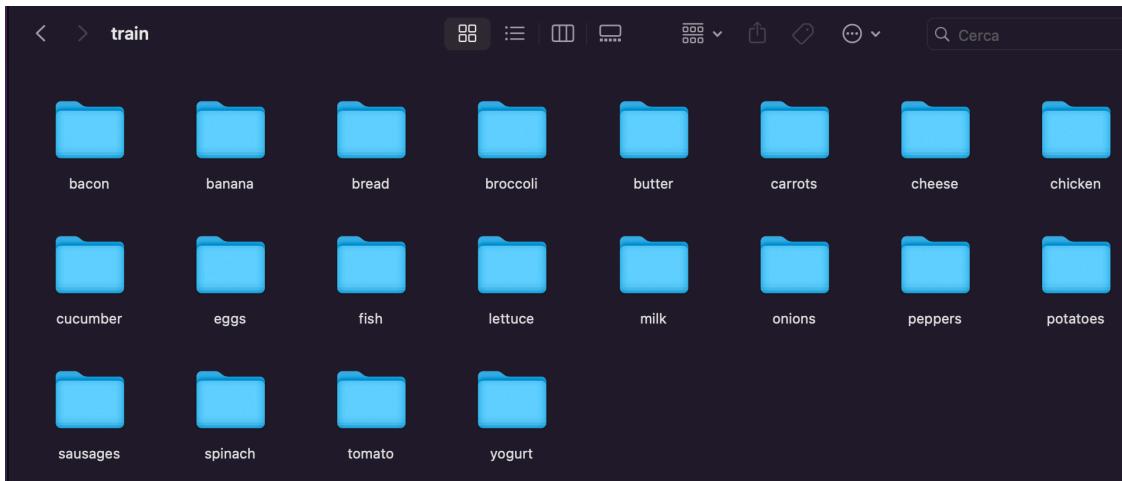
In questo progetto è stato utilizzato un dataset ricavato dalla piattaforma kaggle [1] costituito da 40.000 immagini divise in venti categorie:

1. Bacon
2. Banana
3. Bread
4. Broccoli
5. Butter
6. Carrots
7. Cheese
8. Chicken
9. Cucumber
10. Eggs
11. Fish
12. Lettuce
13. Milk
14. Onions
15. Peppers
16. Potatoes
17. Sausages
18. Spinach
19. Tomato
20. Yogurt

La prima elaborazione del dataset consiste nell'utilizzare un subset del dataset originale, ovvero 20.000 immagini divise in 10.000 per il training, 5.000 per il validation e 5.000 per il testing, il che comporta 500 immagini per categoria per il training e 250 immagini per categoria per il validation e il testing.

È stato usato un semplice script in python che tiene una variabile di conteggio in base alla quale effettua una copia delle immagini nelle nuove 20 cartelle che sono le categorie, a loro volta per ogni cartella che rappresenta training, validation e testing:





Data Preprocessing e Augmentation

Avendo scelto come rete neurale, la rete ResNet50 pre-addestrata su ImageNet che è costituito da 1.000.000 di immagini divise su 1.000 categorie, questo comporta l'applicazione di un preprocessing sulle immagini del dataset utilizzato nel progetto, in base alle immagini appartenenti ad ImageNet affinché ci sia una coerenza tra gli input su cui è stata addestrata la rete e gli input che verranno invece utilizzati.

La trasformazione delle immagini [2] è un processo per modificare i valori originali dei pixel dell'immagine in un set di nuovi valori.

Un tipo di trasformazione che eseguiamo sulle immagini è trasformare un'immagine in un tensore PyTorch, e quando un'immagine viene trasformata in un tensore PyTorch, i valori dei pixel vengono ridimensionati tra 0,0 e 1,0.

In PyTorch, questa trasformazione può essere eseguita utilizzando `torchvision.transforms.ToTensor()` che converte l'immagine con un intervallo di pixel di [0, 255] in un PyTorch FloatTensor di forma (C, H, W) con un intervallo [0,0, 1,0].

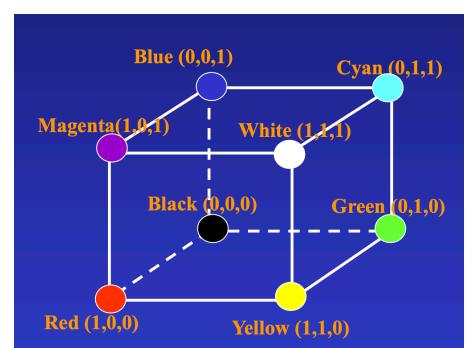
La normalizzazione delle immagini è un'ottima pratica quando lavoriamo con reti neurali profonde perché le reti neurali lavorano meglio quando i dati seguono una distribuzione normale (gaussiana) e sono centrati attorno a zero.

Normalizzare le immagini, infatti, significa trasformare le immagini in valori tali che la media e la deviazione standard dell'immagine diventino rispettivamente 0,0 e 1,0 e per fare ciò, prima la media del canale viene sottratta da ogni canale di input e poi il risultato viene diviso per la deviazione standard del canale.

In questo caso è stata utilizzata la funzione di PyTorch `torchvision.transforms.Normalize()` che accetta come argomento la media e la deviazione standard, a cui per l'appunto sono stati passati due vettori di dimensione 3 che corrispondono agli standard di ImageNet essendo il dataset con cui la rete è stata pre-addestrata.

Il modello RGB è un sistema di codifica dei colori che utilizza tre componenti, rosso, verde e blu, i quali vengono mescolati in diverse intensità per produrre un'ampia gamma di colori.

Nel modello RGB, ogni colore è rappresentato da un insieme di valori che indicano l'intensità dei tre componenti e ogni componente può assumere valori che vanno da 0 a 255 in una gamma a 8 bit, il che significa che ci sono 256 possibili intensità per ciascun colore. Combinando questi valori, si possono ottenere oltre 16 milioni di colori diversi [3].



Avendo optato per la ResNet50 come rete neurale, l'input che accetta deve avere dimensioni 224x224, quindi questo porta ad effettuare, prima della trasformazione in tensore e della normalizzazione, dell'ulteriore trasformazione di resize dell'immagine a 256 attraverso

`torchvision.transforms.Resize(256)`, in modo che tutte abbiano la stessa dimensione, e di ritaglio della porzione centrale attraverso la funzione `torchvision.transforms.CenterCrop(224)`.

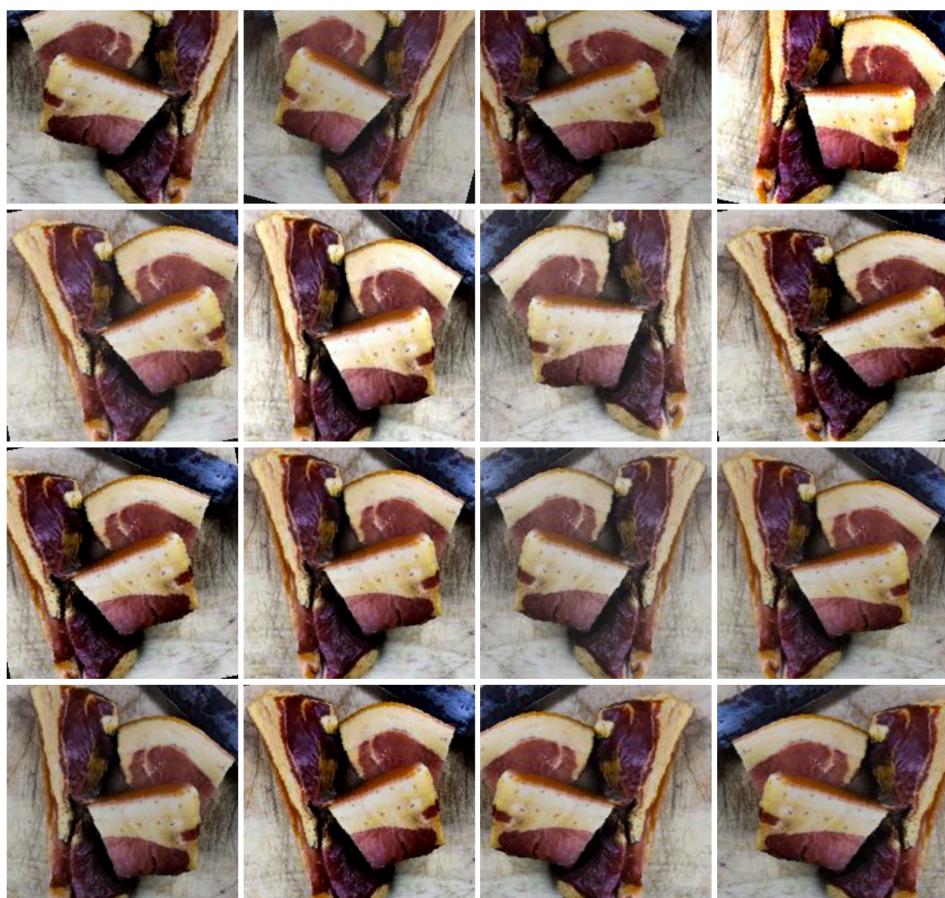
Le tre trasformazioni di cui sopra, sono state applicate ai dati di training, di validation e di test.

Una buona pratica, quando si usano reti neurali, è quella di ultimare la tecnica di data augmentation. Di solito non disponiamo di quante immagini vorremmo all'interno del dataset e quello che si fa è aumentarlo prendendo le immagini che già possediamo e modificarle, ovvero invertirle, ruotarle, ritagliarle e ridimensionarle, oppure è possibile anche applicare dei filtri, senza alterare troppo l'immagine di partenza.

L'idea è che quell'immagine deve ritornare sempre la stessa label anche se modificata.

In questo caso sono state applicate delle trasformazioni solo sui dati di training attraverso PyTorch, del tipo:

1. Random Resize Crop: genera un ritaglio casuale dell'immagine dopo averla ridimensionata
2. Random Rotation: ruota l'immagine di un angolo specificato
3. Color Jitter: modifica la luminosità, il contrasto e la saturazione di un'immagine
4. Random Horizontal Flip: capovolge l'immagine orizzontalmente con una probabilità, di solito 0.5



Modello

Il nostro obiettivo è adattare un modello pre-addestrato di ResNet50 per un nuovo compito di classificazione. Questo processo viene chiamato transfer learning, perché partiamo da un modello già addestrato su un dataset molto grande (come ImageNet) e lo riutilizziamo per un altro compito, adattandolo al nostro dataset specifico.

Per farlo, utilizziamo una tecnica chiamata fine-tuning, che permette di modificare alcuni layer del modello per renderlo più adatto al nostro problema.

	Block type	Stem	Stage 1		Stage 2		Stage 3		Stage 4		FC layers	ImageNet GFLOP	top-5 error	
			layers	Blocks	Layers	Blocks	Layers	Blocks	Layers	Blocks	Layers			
ResNet-18	Basic		1	2	4	2	4	2	4	2	4	1	1.8	10.92
ResNet-34	Basic		1	3	6	4	8	6	12	3	6	1	3.6	8.58
ResNet-50	Bottle		1	3	9	4	12	6	18	3	9	1	3.8	7.13

Cosa abbiamo fatto

1. Caricamento pesi pre-addestrati

- Prima di tutto, abbiamo indicato il percorso del file con i pesi pre-addestrati di ResNet50. Si tratta di un file “.pth” che contiene i parametri del modello già addestrato su un altro dataset.
- Poi abbiamo creato un’istanza del modello ResNet50 senza pesi di default (model = resnet50(weights=None)) e caricato i pesi salvati usando torch.load().

2. Modifica del Layer Finale:

- Dato che il numero di classi nel nostro dataset è diverso da quello usato originariamente per addestrare ResNet50, abbiamo dovuto modificare il **layer finale** del modello.
 - Abbiamo aggiunto un **dropout** per evitare l’overfitting e sostituito il layer finale con uno nuovo che ha un numero di output pari al numero di classi del nostro dataset.
- Questa modifica permette al modello di adattarsi meglio al nuovo compito.

3. Congelamento e Sblocco dei Layer:

- Inizialmente, abbiamo **congelato tutti i pesi** del modello. Congelare significa che questi pesi non vengono aggiornati durante l’addestramento, e questo ci aiuta a preservare le conoscenze generali che il modello ha già appreso dal dataset precedente (ad esempio, ImageNet).
- Poi, per il processo di **fine-tuning**, abbiamo **sbloccato solo l’ultimo blocco** del modello (layer 4) e il layer completamente connesso che abbiamo modificato. Così facendo, il modello può adattare solo queste parti ai dati specifici, mentre le altre rimangono fisse.

4. Preparazione per l’Addestramento:

- Abbiamo definito la funzione di perdita CrossEntropyLoss, che è adatta per problemi di classificazione.
- Per aggiornare i pesi del modello, abbiamo scelto l’ottimizzatore **Adam**, ma con un **learning rate differenziato**:
 - Un learning rate più piccolo per layer4, perché non vogliamo che cambi troppo.
 - Un learning rate più alto per il layer finale, perché questo deve essere più flessibile per adattarsi al nuovo compito.

5. Scheduler per il Learning Rate:

- Abbiamo usato uno scheduler per regolare dinamicamente il **learning rate** durante l’addestramento, basandoci sulla **validation loss**. Se la validation loss non migliora dopo un po’, lo scheduler riduce il learning rate, permettendo un fine-tuning più delicato nelle fasi finali dell’addestramento.

6. Funzioni di Addestramento e Validazione:

- Abbiamo scritto due funzioni: **train()** e **validate()**.
 - **train()**: Allena il modello sui dati di training, calcolando la loss e aggiornando i pesi.
 - **validate()**: Calcola le prestazioni del modello sui dati di validazione, ma senza aggiornare i pesi.
- In questo modo possiamo monitorare se il modello sta migliorando o se inizia a fare overfitting (cioè, ad adattarsi troppo ai dati di training e non generalizzare bene).

7. Loop di Addestramento:

- Per 50 epoche, abbiamo addestrato il modello usando la funzione **train()** e poi verificato le sue prestazioni sui dati di validazione con **validate()**.
- Dopo ogni epoca, abbiamo registrato i risultati e aggiornato il learning rate con lo scheduler. Così possiamo vedere come si evolve il modello e quando è meglio rallentare il processo di apprendimento.

8. Salvataggio del Modello Addestrato:

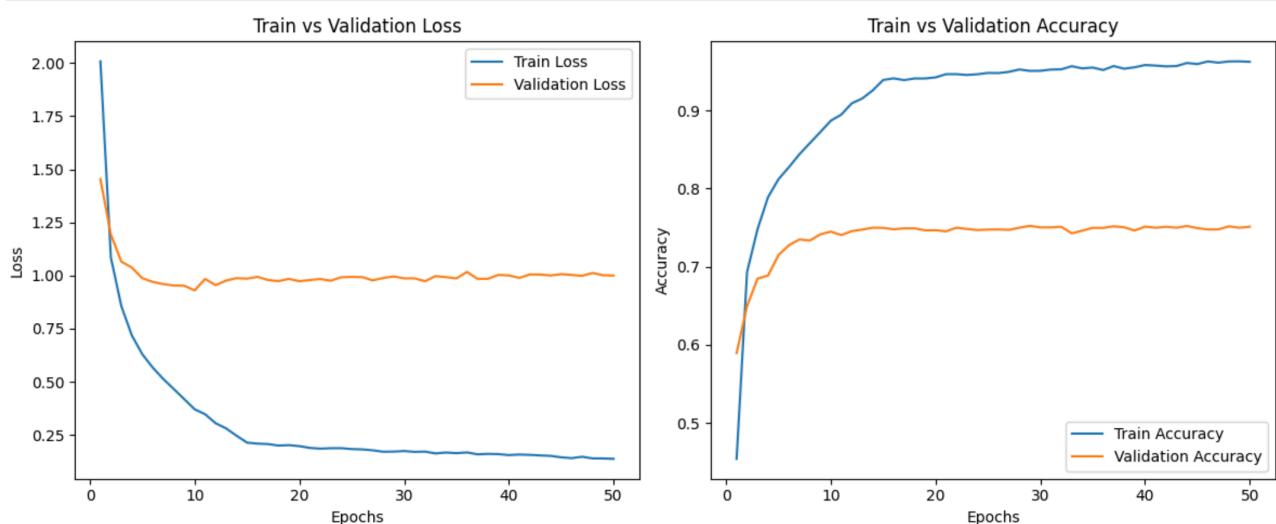
- Alla fine, abbiamo salvato i pesi del modello addestrato in un file “.pth”, così possiamo riutilizzarlo nell’app per predire l’alimento nell’immagine, senza doverlo addestrare da capo.

Analisi

Effettuato il training su train set e validation set, dopo 50 epoche sono stati ottenuti i seguenti risultati:

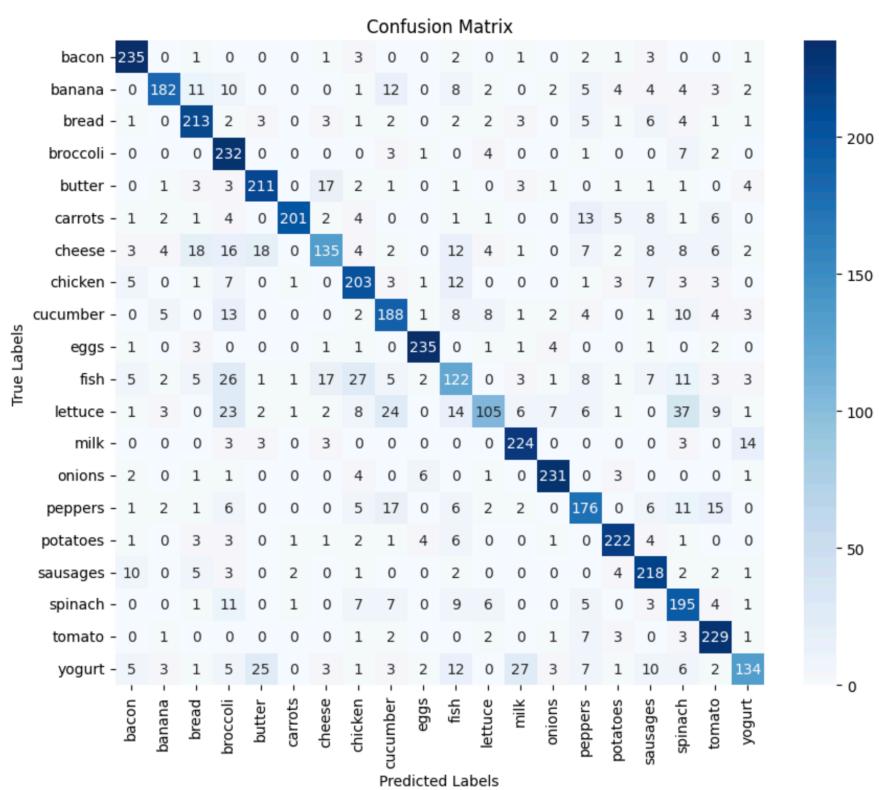
- train loss di 0.1376
- train accuracy di 0.9623
- validation loss di 1.0001
- validation accuracy di 0.7512.

Di seguito sono riportati i grafici relativi alla loss e all’accuracy durante il processo di training e validation.



Successivamente è stata effettuata la valutazione del modello ottenuto sui dati di test, dati che il modello non ha mai visto prima e non conosce.

Sono state ottenute per ogni immagine di test, le predizioni, si è preso il valore massimo lungo la dimensione delle classi e si è potuto così stampare la matrice di confusione.



La matrice di confusione consiste in righe e colonne rappresentate rispettivamente dalle label vere e dalle label predette.

Considerando i come indice della riga e j come indice della colonna, il valore m_{ij} mi indica il numero di immagini che hanno come label vera la classe i e che sono state predette con la classe j .

Di conseguenza l'obiettivo è ottenere dei valori più alti possibile lungo la diagonale della matrice, poiché la coincidenza dei due indici i e j dimostra la giusta predizione delle immagini e quindi il funzionamento del modello addestrato.

A partire dall'immagine infatti, si nota che la diagonale della matrice è messa in evidenza con colori più scuri, segno di una buona predizione.

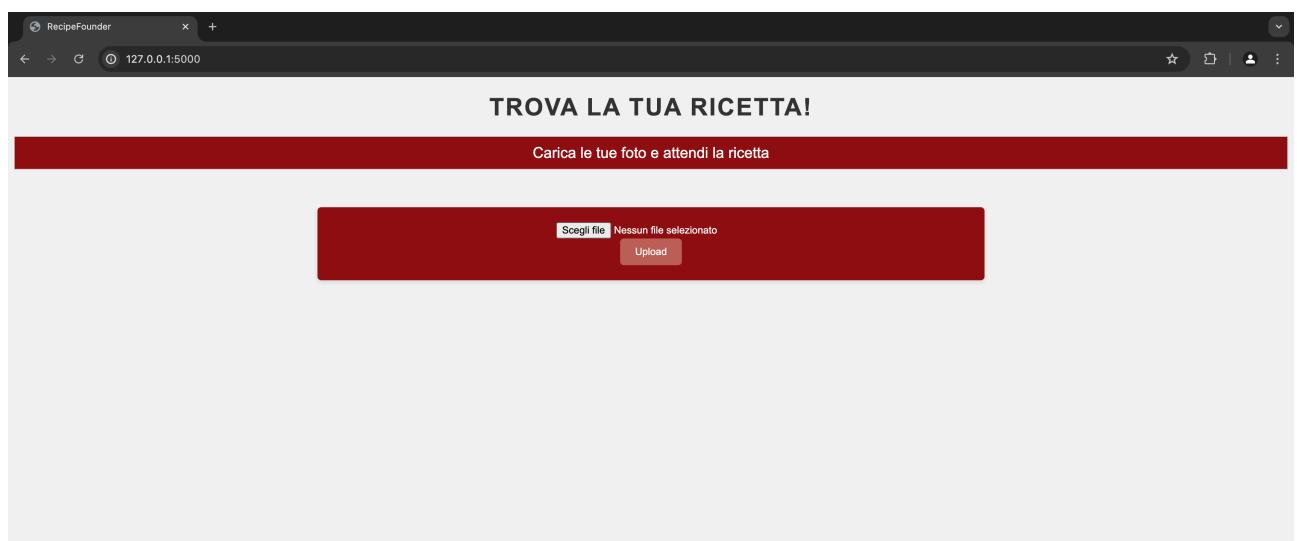
App

Descrizione delle Funzionalità dell'app Flask

1. Pagina Principale (/)
 - Visualizza la pagina principale dell'applicazione dove l'utente può caricare le immagini degli alimenti.
2. Gestione Mancato Caricamento del File (/NotFile)
 - Mostra una pagina di errore se l'utente tenta di caricare un'immagine senza selezionare un file.
3. Gestione Caricamento di Più File (/MoreFile)
 - Mostra una pagina che consente di inserire un'ulteriore immagine oltre a quella già caricata.
4. Caricamento Immagini (/upload)
 - Gestisce il caricamento dell'immagine. Se l'immagine viene caricata correttamente, viene salvata nella directory upload.
5. Fornitura Ricette (/give)
 - Analizza le immagini caricate, riconosce gli alimenti presenti utilizzando il modello pre-addestrato, e reindirizza l'utente a una pagina di ricerca con suggerimenti di ricette che contengono gli alimenti riconosciuti. In questa funzione avviene il caricamento del modello, la modifica dei livelli come effettuato durante l'addestramento, la modifica delle immagini caricate in precedenza in modo coerente con l'input accettato dal modello e la valutazione dello stesso sulle immagini modificate.

Come Utilizzare l'Applicazione

1. **Caricare un'immagine di un alimento:** L'utente può caricare un'immagine dalla pagina principale cliccando dapprima su "Scegli File" e poi su "Upload":



2. **Caricare più immagini o ottenere suggerimenti di ricette:** Dopo il caricamento, l'utente visualizzerà quanto segue:



3. **Ottener suggerimenti di ricette:** L'utente cliccherà "GetRecipe", e l'applicazione riconoscerà gli alimenti presenti nell'immagine(immagini se più di una) e reindirizzerà l'utente a una pagina con suggerimenti di ricette che contengono quegli alimenti:

Google Recipes with eggs bacon

Tutti Immagini Video Notizie Web Libri Finanza Strumenti

Ricette :



Scrambled eggs ricetta americana
Blog GialloZafferano.it
4.8 ★★★★★ (58)
20 min



Poached eggs with bacon and tomatoes
BBC
4.1 ★★★★★ (14)
40 min



Sunny-Side Up-Eggs and Bacon
Vermicular
Nessuna recensione

Mostra altre ricette ▾

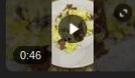
Video :



Scrambled eggs in bacon. A simple and delicious breakfast ...
YouTube - KAZAN
8 ott 2023
2:31



5-Ingredient Bacon & Eggs Bake | bacon, egg | The iconic ...
Facebook - Foodies of SA
17 mar 2023
1:00



Killer Scrambled Eggs with Bacon & Cheese #quickbites ...
YouTube - livelife365
12 feb 2024
0:46

Mostra tutto →

Bibliografia

- [1] <https://www.kaggle.com/datasets/liamboyd1/multi-class-food-image-dataset>
- [2] <https://www.geeksforgeeks.org/how-to-normalize-images-in-pytorch/>
- [3] slide lezioni