Monument info

Componenti del progetto:

- Letizia Grazia **Di Munno** - Matricola: **775552** l.dimunno@studenti.uniba.it
- Marco **Inverardi** - Matricola: **710324** m.inverardi@studenti.uniba.it

Il modello addestrato, i codici sorgente in Python (.py), la cartella con immagini per il test del programma e il dataset utilizzato per il training sono consultabili nel repository GitHub:
🔗GitHub Repository

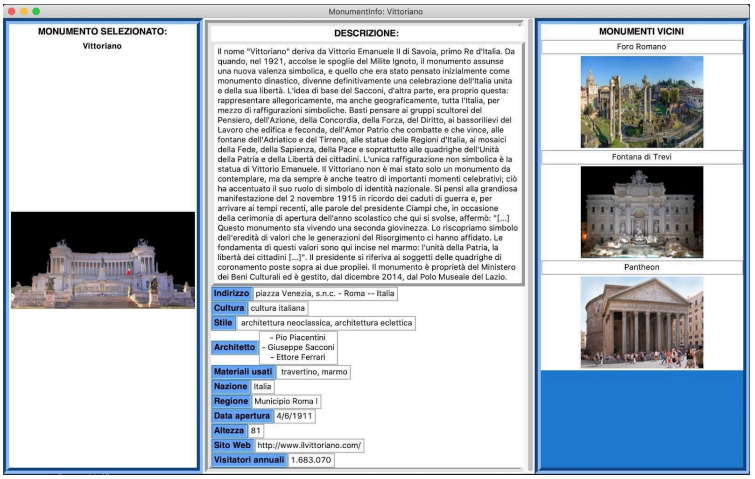
Introduzione

Negli ultimi anni, il deep learning ha rivoluzionato il riconoscimento delle immagini, permettendo di sviluppare sistemi sempre più accurati. In questo progetto, abbiamo creato un riconoscitore di monumenti basato su una **rete neurale convoluzionale (CNN)**, capace di analizzare le immagini ed effettuare una classificazione precisa.

Utilizzando **TensorFlow e Keras**, il modello è stato addestrato su un dataset di immagini di diversi monumenti, imparando a riconoscerne le caratteristiche distintive, come forme e dettagli architettonici. La rete è composta da *livelli convoluzionali* per l'estrazione delle caratteristiche, *livelli di pooling* per ridurre la dimensionalità e strati completamente connessi per la classificazione finale. Grazie a questa struttura, il sistema è in grado di riconoscere i monumenti con alta precisione, anche in condizioni di luce variabili o con parziali ostruzioni.

Interfaccia Grafica (GUI)

All'avvio del programma, l'utente può selezionare un monumento da riconoscere. Le immagini utilizzabili si trovano nella cartella test del repository.



L'interfaccia è suddivisa in tre colonne:

➔ **Colonna sinistra** – Visualizza l'immagine del monumento riconosciuto.

➔ **Colonna centrale** – Contiene una breve descrizione e altre informazioni correlate.

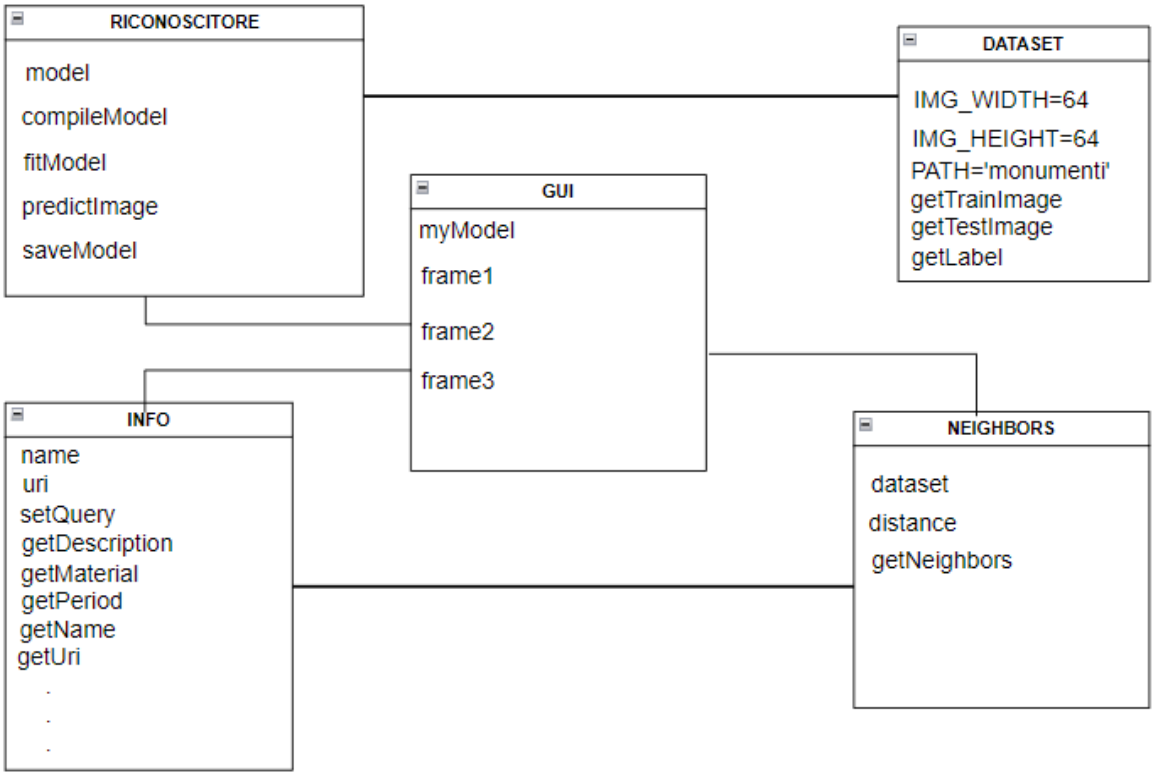
➔ **Colonna destra** – Presenta tre monumenti consigliati situati nelle vicinanze.

Descrizione del Progetto

Struttura del Repository

- **modello/** – Contiene il file *mymodel.h5* con il modello del riconoscitore.
- **monumenti/** – Include immagini di monumenti classificati per il training, oltre ai file:
  - *label.csv*: nomi dei monumenti per le query.
  - *monuments.csv*: coordinate geografiche (LAT, LON) dei monumenti italiani con la proprietà *attrazione turistica* su WikiData (usato nel k-NN).
- **test/** – Contiene immagini di test.

Classi Principali



- **GUI** (gui.py) – Gestisce l'interfaccia grafica e richiama le altre classi.
- **RICONOSCITORE** (myModel.py) – Implementa una rete neurale convoluzionale per il riconoscimento delle immagini.
- **INFO** (infoMonumento.py) – Esegue query per estrarre informazioni sul monumento riconosciuto.
- **DATASET** (dataset.py) – Gestisce la creazione dei dataset di training e validation.
- **NEIGHBORS** (neighbors.py) – Implementa il sistema di raccomandazione basato su k-NN.

Dettagli implementativi:

Il programma è stato sviluppato utilizzando **Python 3.7** e implementa una r**ete neurale convoluzionale (CNN)** per la classificazione delle immagini. Inoltre, grazie all'impiego di **ontologie online** e **Linked Open Data**, è possibile recuperare informazioni strutturate dal **Semantic Web**, fornendo all'utente una descrizione dettagliata del monumento riconosciuto. Il sistema suggerisce altri monumenti nelle vicinanze utilizzando **l'algoritmo di apprendimento supervisionato case-based K-NN (k=3)**, sfruttando un dataset creato appositamente con dati estratti da **WikiData**.

Dataset:

esempio di alcune immagini presenti nel dataset:

TRAINING SET

Il dataset è stato realizzato utilizzando immagini prelevate dal web, elaborate per migliorare la qualità e aumentare il numero di esempi disponibili. Ogni monumento dispone di circa 15 immagini, suddivise in:

- 90% per il training

- 10% per la validazione

Le immagini sono ridimensionate a **64x64 pixel** e convertite in array con valori normalizzati tra 0 e 1. Per il **preprocessing**, si utilizza la libreria **Keras.preprocessing.image**, che permette di generare i batch di training e validation tramite **ImageDataGenerator**.

Riconoscitore:

Il cuore del sistema di riconoscimento dei monumenti è una **rete neurale convoluzionale (CNN)**, progettata per analizzare e classificare le immagini in modo accurato. La scelta di una **CNN** è motivata dal fatto che questo tipo di architettura è particolarmente efficace nel riconoscere pattern visivi, come forme, texture e dettagli distintivi, fondamentali per distinguere tra diversi monumenti.

L'obiettivo principale del riconoscitore è quello di esaminare un'immagine in ingresso e determinare con un'elevata probabilità a quale monumento essa appartenga. Per raggiungere questo scopo, il modello è stato sviluppato utilizzando **Keras**, una delle librerie più diffuse per la costruzione di reti neurali, con il backend di **TensorFlow**.

I principali livelli della rete neurale sono stati importati dal modulo **tensorflow.keras.layers,** che fornisce gli strumenti necessari per costruire la CNN:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten, Dropout, MaxPooling2D
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

- **Conv2D**: livello convoluzionale per l'estrazione delle caratteristiche.
- **MaxPooling2D**: livello di pooling per ridurre la dimensionalità dei dati.
- **Flatten**: conversione dell'output convoluzionale in un vettore.
- **Dense**: strato completamente connesso per la classificazione finale.
- **Dropout**: livello per ridurre l'overfitting (utilizzato in fase di tuning).
- **Adam**: ottimizzatore per migliorare l'apprendimento della rete.

L’utilizzo di questi livelli consente di costruire un modello robusto, capace di estrarre informazioni rilevanti dalle immagini e di classificarle con precisione.

Struttura della rete neurale

Il modello è stato costruito seguendo **un'architettura sequenziale**, cioè una successione di livelli (**layers**) che elaborano progressivamente l’immagine. Ogni livello svolge un ruolo specifico nel processo di riconoscimento, trasformando l’immagine grezza in una rappresentazione numerica che può essere interpretata dalla rete.

1. Livello di input: Elaborazione dell’immagine

Quando un’immagine viene fornita al modello, il primo passaggio consiste nel preprocessarla per adattarla al formato richiesto dalla rete. Poiché la **CNN** è stata progettata per lavorare con immagini di dimensioni **64x64 pixel** e con tre canali di colore (**RGB**), ogni immagine in ingresso viene ridimensionata e convertita in un array numerico normalizzato con **valori** compresi tra **0 e 1**.

Questa normalizzazione è fondamentale, perché permette alla rete di apprendere più velocemente, evitando che i valori troppo grandi o troppo piccoli influenzino negativamente l’addestramento.

2. Livelli convoluzionali: Estrazione delle caratteristiche

Una volta elaborata l’immagine, il modello passa attraverso una **serie di strati convoluzionali** (**Conv2D**). Questi livelli agiscono come filtri che analizzano l’immagine alla ricerca di *pattern visivi* specifici.

- Il primo strato convoluzionale applica **16 filtri di dimensione 3x3** all’immagine. Questi filtri lavorano come "lenti" che identificano dettagli di base, come i contorni e le linee orizzontali o verticali.

- Dopo questa prima operazione, viene utilizzata una funzione di attivazione chiamata **ReLU (Rectified Linear Unit)**, che introduce una componente di non linearità, permettendo alla rete di apprendere strutture più complesse.

Dopo ogni livello convoluzionale, si applica un’operazione chiamata **MaxPooling2D**, che riduce la dimensionalità dell’immagine selezionando solo le caratteristiche più importanti. In pratica, questa operazione riduce la risoluzione, ma conserva le informazioni più significative, riducendo il rischio di sovraccaricare il modello con dettagli irrilevanti.

Dopo il primo blocco di **convoluzione e pooling**, il modello applica due ulteriori livelli convoluzionali:

- Il secondo livello aumenta il **numero di filtri a 32**, migliorando la capacità della rete di rilevare caratteristiche più specifiche.

- Il terzo livello porta il **numero di filtri a 64**, permettendo al modello di riconoscere dettagli distintivi come decorazioni architettoniche o forme specifiche dei monumenti.

Questa progressione nell'aumento dei filtri è importante perché consente alla rete di passare da una rappresentazione generica dell'immagine a una sempre più dettagliata, migliorando la capacità di classificazione.

3. Flattening: Conversione in vettore

Dopo l'ultimo strato convoluzionale, l'output della rete è ancora in forma *bidimensionale* (una matrice di valori). Tuttavia, per poter effettuare la classificazione, questi dati devono essere trasformati in un **vettore unidimensionale.**

Questa operazione, chiamata **Flattening**, prende tutte le caratteristiche estratte e le appiattisce in un'unica sequenza di valori numerici. Questo passaggio è cruciale perché permette di collegare l'output convoluzionale ai livelli successivi, che si occupano della classificazione finale.

4. Strati completamente connessi (Dense Layers): Decisione finale

A questo punto, i dati vengono elaborati da una serie di livelli completamente connessi (**Dense layers**), che combinano le informazioni estratte nei passaggi precedenti per effettuare la classificazione finale.

- Il primo livello **fully connected** è composto da **128 neuroni**, con funzione di attivazione **ReLU**, che continua a modellare le informazioni estratte in modo non lineare.

- Il secondo livello ha **64 neuroni** e utilizza ancora **ReLU** per ridurre progressivamente la dimensionalità dei dati mantenendo le informazioni più rilevanti.

- Infine, l'ultimo strato contiene un **numero di neuroni pari al numero di classi del dataset,** ognuno dei quali rappresenta un monumento diverso. La funzione di attivazione utilizzata è **Softmax**, che assegna una probabilità a

ciascuna classe, indicando con quale confidenza il modello ha riconosciuto il monumento in esame.

Compilazione e addestramento del modello

Per rendere la rete operativa, è necessario definirne i parametri di apprendimento. La **CNN** è stata compilata con:

- **Ottimizzatore Adam**, che regola dinamicamente il tasso di apprendimento per migliorare l'addestramento.
- **Funzione di perdita Binary Crossentropy**, ideale per problemi di classificazione multi-classe.
- **Metriche di valutazione**: Accuratezza, per monitorare le prestazioni del modello durante l'addestramento.

L'addestramento è stato eseguito utilizzando il **metodo fit**, con i seguenti parametri:

- **Numero di epoche: 20**, scelto in base a test con **K-fold Cross Validation (k=5).**
- **Batch size: 3**, selezionato per ottimizzare l'aggiornamento dei pesi senza appesantire la memoria.
- **Dataset** suddiviso in **90% training e 10% validation**, per valutare le prestazioni del modello.

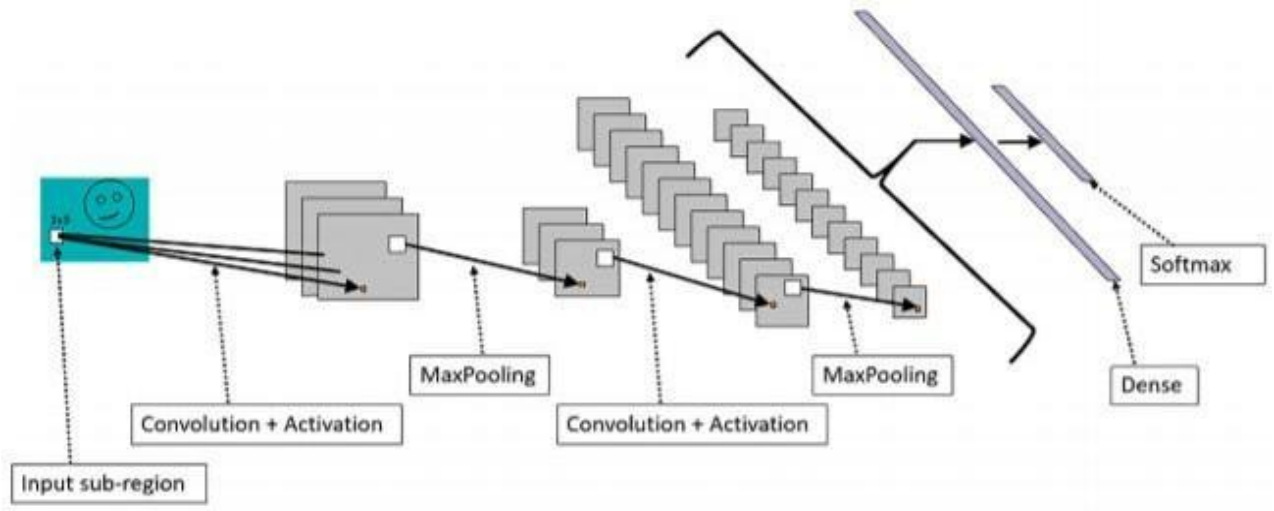
Dopo l'addestramento, il modello ha raggiunto **un'accuratezza media** del **98%,** dimostrando un'elevata capacità di riconoscere correttamente i monumenti nel dataset.

Monitoraggio delle prestazioni

Durante il training, è stato generato un grafico della **funzione di perdita**, che mostra l'andamento dell'errore nel tempo. Questo è un indicatore fondamentale per capire se il modello sta migliorando o se si sta verificando **overfitting** (quando la rete si adatta troppo ai dati di training e perde capacità di generalizzazione).

L'andamento della curva ha confermato che il modello sta apprendendo in modo efficace, **senza segni** evidenti **di overfitting**.

esempio **rappresentazione grafica CNN**:



MyModel:

Il nostro modello di rete neurale convoluzionale (**CNN**) è stato progettato con un'architettura ben definita, composta da **9 livelli**, ognuno con una funzione specifica nel processo di riconoscimento delle immagini.

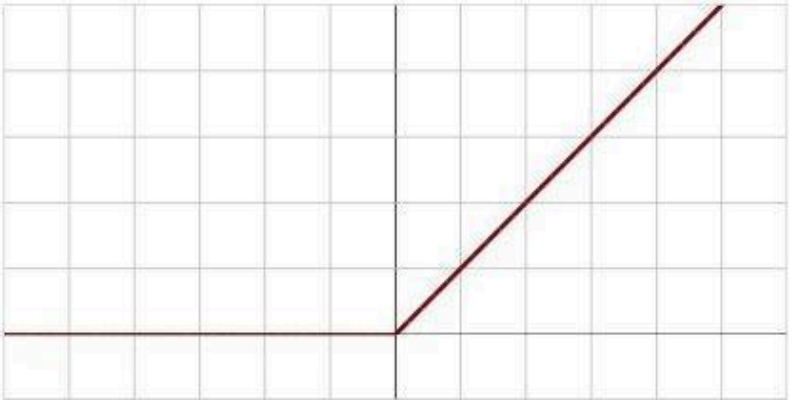
```
self.model = Sequential([
    Conv2D(16, 3, padding='same', activation='relu', input_shape=(64, 64, 3)),
    MaxPooling2D(),
    Conv2D(32, 3, padding='same', activation='relu'),
    MaxPooling2D(),
    Conv2D(64, 3, padding='same', activation='relu'),
    MaxPooling2D(),
    Flatten(),
    Dense(512, activation='relu'),
    Dense(train.num_classes, activation='softmax')
])
```

Struttura del modello

La rete inizia con un **primo livello convoluzionale** che prende in input immagini con dimensioni **64x64x3** (dove 3 rappresenta i tre canali RGB). Questo livello utilizza **16 filtri** con kernel di dimensione **3x3**, che permettono di estrarre caratteristiche fondamentali dalle immagini. La funzione di attivazione adottata è la **ReLU**, scelta per la sua capacità di introdurre non linearità e migliorare l'apprendimento del modello.

- *Rectified Linear Unit (ReLU):*

$$f(x) = max(0, x)$$



Dopo questa prima fase, viene applicato un livello di **MaxPooling2D**, che ha il compito di ridurre la dimensionalità delle immagini senza perdere informazioni rilevanti. Questo processo consente di ottimizzare il tempo di calcolo e prevenire il rischio di overfitting.

Successivamente, troviamo **due ulteriori coppie di livelli convoluzionali e di pooling**, che servono a raffinare progressivamente l'estrazione delle caratteristiche visive. In questa fase, il numero di filtri aumenta gradualmente, passando da **16 a 32 e poi a 64**, permettendo al modello di identificare dettagli sempre più complessi nelle immagini in input.

A questo punto entra in gioco il livello **Flatten**, il cui scopo è trasformare la rappresentazione multi-dimensionale generata dai livelli convoluzionali in un unico vettore piatto. Questo passaggio è essenziale per collegare la parte convoluzionale della rete con i livelli densi (**fully connected**).

Infine, troviamo i **livelli completamente connessi (Dense layers)**, che elaborano il vettore prodotto dallo strato Flatten e permettono al modello di classificare l'immagine in base alla probabilità di appartenenza a una determinata categoria. L'ultimo strato utilizza la funzione di attivazione **Softmax**, che restituisce un vettore di probabilità: ogni elemento rappresenta la probabilità che l'immagine appartenga a una delle classi previste dal modello.

Compilazione e addestramento

Dopo aver definito l'architettura del modello, è necessario **compilarlo** specificando i parametri di ottimizzazione.

```
def compileModel(self):
    self.model.compile(optimizer='adam',
                       loss='binary_crossentropy',
                       metrics=['accuracy'])
```

Abbiamo scelto **ADAM** come ottimizzatore, noto per la sua efficienza nell'aggiornamento dei pesi durante l'addestramento. Per la funzione di perdita (**loss function**), è stata adottata la **binary_crossentropy**, utile nei problemi di classificazione.

L'accuratezza del modello viene misurata attraverso la metrica **accuracy**, che confronta le previsioni con le etichette reali per valutare la qualità dell'apprendimento.

- L'OTTIMIZZATORE specifica l'algoritmo scelto per effettuare la discesa di gradiente
- La LOSS utilizzata è la log loss:

$$\text{log\_loss} = -(y\log(p)+(1-y)\log(1-p))$$

dove y è la classe reale, mentre p è quella predetta dalla CNN

Per effettuare il train richiamiamo il **metodo *fit()***:

```
def fitModel(self):
    total_train = self.train.samples
    batch_train = self.train.batch_size

    total_val = self.val.samples
    batch_val = self.val.batch_size

    history=self.model.fit_generator(
        self.train,
        steps_per_epoch=total_train // batch_train,
        epochs=self.epochs,
        validation_data=self.val,
        validation_steps=total_val // batch_val
    )
```

Per avviare l'addestramento, utilizziamo il metodo **fit()**, fornendo come input:

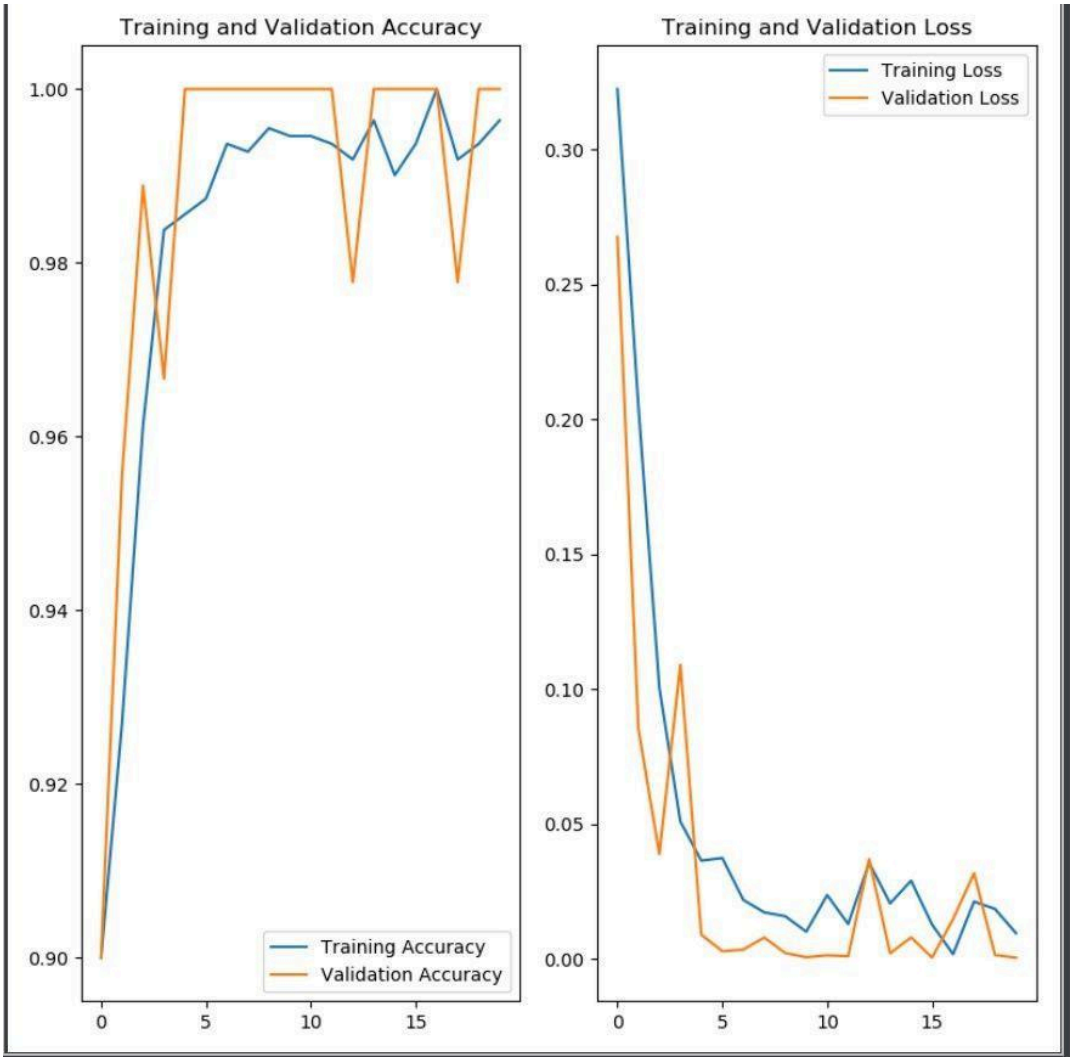
- Il set di **training**, utilizzato per istruire il modello.
- Il set di **validazione**, utile per monitorare l'andamento dell'apprendimento e individuare eventuali problemi di overfitting.
- Il parametro **epochs**, che indica quante volte il modello deve processare l'intero dataset.

Dopo un'analisi con **K-fold Cross Validation (k=5)**, sono stati determinati i parametri ottimali:

- **Batch size:** 3
- **Epochs:** 20

Queste impostazioni ci hanno permesso di ottenere un'**accuratezza media del 98%** sul dataset di validazione.

Qui un grafico, ricavato direttamente dal programma, sull'andamento dell'errore del nostro modello sull'insieme di validazione e su quello di train:



Query e Recupero Informazioni

Una volta che il riconoscitore ha identificato il monumento presente nell'immagine fornita dall'utente, il passo successivo è quello di estrarre informazioni dettagliate su di esso. Per farlo, il programma sfrutta basi di conoscenza strutturate come **WikiData** e **DBpedia**, due database online che raccolgono e organizzano informazioni in modo strutturato, rendendole facilmente accessibili tramite query.

Uso di SPARQL per l'estrazione delle informazioni

WikiData e DBpedia funzionano principalmente attraverso un formato chiamato **triple RDF** (*soggetto-predicato-oggetto*), che consente di rappresentare le informazioni in modo relazionale.
Per interrogare questi database e ottenere le informazioni necessarie, il sistema utilizza **SPARQL**, un linguaggio di interrogazione progettato specificamente per database semantici.

```
def setQuery(cls, query, wrapper):

    user_agent = "MonumentInfo/%s.%s" % (sys.version_info[0], sys.version_info[1])
    sparql = SPARQLWrapper(wrapper,agent=user_agent)
    sparql.setQuery(query)
    sparql.setReturnFormat(JSON)
    while True:
        i = 1
        try:
            results = sparql.query().convert()
            return results
        except urllib.error.HTTPError as err:
            print(err)
            time.sleep(i)
            i += 1
```

Per implementare queste query, il programma sfrutta la libreria **SPARQLWrapper**, che consente di interfacciarsi con gli **endpoint di WikiData e DBpedia.** Il metodo principale utilizzato è `setQuery(query)`, che accetta in input la **query SPARQL** sotto forma di stringa e l'**endpoint** da cui prelevare le informazioni. Inoltre, per garantire un'elaborazione strutturata dei risultati, il formato di output viene impostato su **JSON (**`setReturnFormat(JSON)`**).** In questo modo, il programma può estrarre e organizzare facilmente i dati ottenuti.

Raccomandazione di monumenti vicini con k-NN

Oltre alle informazioni base sul monumento riconosciuto, il sistema è in grado di suggerire altri luoghi di interesse nelle vicinanze. Questo è possibile grazie all'uso dell'algoritmo **k-Nearest Neighbors (k-NN)**, un metodo di apprendimento supervisionato basato sulla similarità tra dati.

L'algoritmo lavora partendo da un **dataset pre-costruito** che contiene informazioni su vari monumenti e luoghi di interesse turistico in Italia. Questo **dataset** è stato creato tramite **una query su WikiData** che ha estratto tutti i monumenti etichettati come **"attrazione turistica"**, recuperando per ognuno la ***posizione geografica*** (latitudine e longitudine) e ***l'URI*** corrispondente, che permette di ottenere ulteriori dettagli attraverso **query SPARQL**.

Quando un utente riconosce un monumento, il sistema calcola la distanza geografica tra questo e gli altri luoghi presenti nel dataset, selezionando i **tre più vicini**. In questo modo, l'utente riceve consigli su altri siti da visitare nelle vicinanze, migliorando l'esperienza turistica e fornendo un valore aggiunto all'applicazione.

Informazioni Aggiuntive sul Codice

Il codice Python allegato è organizzato in modo modulare per facilitare la manutenzione e l'espansione del progetto:

- **Modularità**: Ogni componente è separato per rendere il codice leggibile e facilmente aggiornabile.
- **Efficienza**: L'uso di reti neurali convoluzionali consente un riconoscimento accurato e veloce.
- **Scalabilità**: È possibile espandere il dataset e migliorare il modello aggiungendo nuove immagini e informazioni estratte dal web.