Classificazione delle immagini con Intelligenza Artificiale

di **Massimo .Vailati**
06/11/2025

MACHINE LEARNING RETI NEURALI CLASSIFICAZIONE IMMAGINI RESNET
SCIENTIFICO S.A., TT INFORMATICA

Le **reti neurali convoluzionali (CNN)** sono oggi lo strumento principale per la **classificazione delle immagini**, ispirate al funzionamento del cervello umano. Attraverso strati di neuroni artificiali e filtri che apprendono automaticamente, la rete individua bordi, forme e oggetti sempre più complessi. L'architettura **ResNet**, introdotta da Microsoft, ha permesso di superare i limiti delle reti profonde grazie ai *residual blocks*, migliorando precisione e stabilità. Con strumenti come **ML.NET Model Builder**, anche studenti e sviluppatori possono creare facilmente classificatori di immagini, addestrandoli su dataset reali e integrandoli in applicazioni pratiche.

Come funziona una rete neurale per la classificazione delle immagini

Negli ultimi anni, le r**eti neurali artificiali** si sono affermate come la tecnologia di riferimento per il riconoscimento automatico delle immagini. Applicazioni come il riconoscimento facciale, la diagnosi medica da immagini radiologiche o i sistemi di guida autonoma si basano su questo tipo di modelli. Ma come funziona, esattamente, una rete neurale per la **classificazione delle immagini**?

Una rete neurale prende ispirazione, in maniera semplificata, dal funzionamento del cervello umano: è composta da **neuroni artificiali** disposti in strati (layer) che elaborano informazioni.
Per la classificazione di immagini, l'obiettivo è assegnare un'etichetta a un input visivo: ad esempio, dire se una foto contiene un **gatto** o un **cane**.

Un'immagine digitale è rappresentata da una matrice di numeri (i valori dei pixel). Questi valori vengono forniti alla rete neurale come **input**.
Ogni neurone esegue un'operazione molto semplice: combina i valori in ingresso con **pesi** (numeri che rappresentano l'importanza di ciascun input), aggiunge un **bias** e applica una funzione non lineare detta **funzione di attivazione**.

Architettura tipica: le reti convoluzionali (CNN)

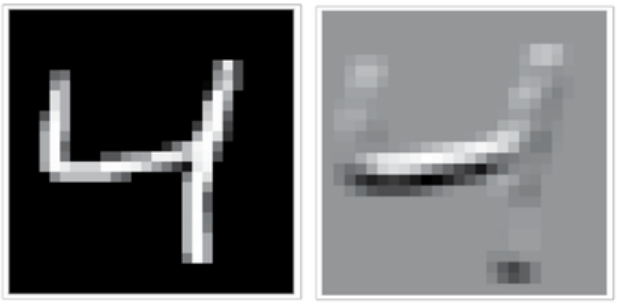
Per la visione artificiale, le **Convolutional Neural Networks (CNN)** sono lo standard. A differenza delle reti neurali tradizionali, le CNN sfruttano **filtri convoluzionali** in grado di rilevare automaticamente caratteristiche visive.

Un **filtro (o kernel)** è una piccola matrice (es. 3×3 o 5×5) che scorre sull'immagine compiendo una convoluzione: i valori dei pixel e del filtro vengono moltiplicati elemento per elemento e poi sommati, producendo un numero che rappresenta una nuova caratteristica dell'immagine in quel punto.

Ad esempio, un filtro come:

$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

è utile per rilevare **bordi orizzontali**: in una zona uniforme restituirà 0, mentre in corrispondenza di una transizione netta tra chiaro e scuro produrrà un valore alto (vedi esempio).



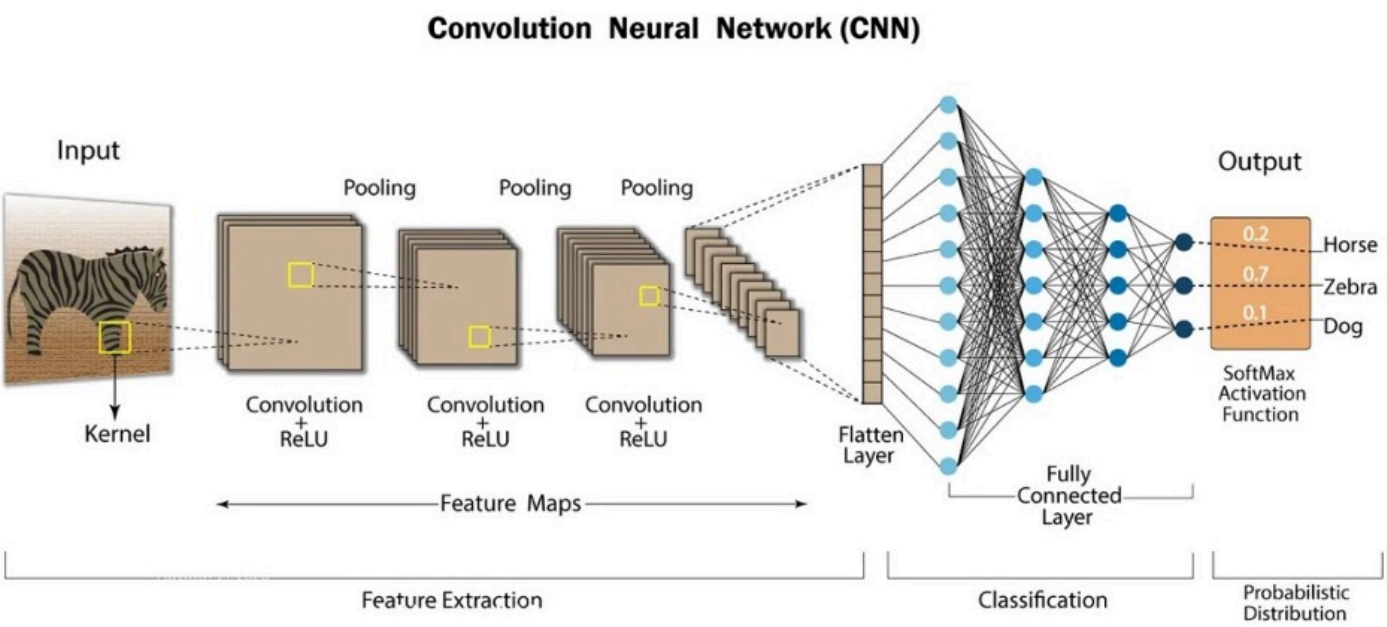
Consulta anche: **https://it.wikipedia.org/wiki/Matrice_di_convoluzione**

Nelle CNN, però, questi filtri **non vengono scelti a mano**, ma sono **imparati automaticamente** durante l'addestramento. Nei primi strati rilevano forme semplici (linee, bordi), negli strati intermedi pattern più complessi (occhi, ruote, petali), e negli strati finali concetti ad alto livello (un volto, un animale, un oggetto).

Questo approccio ha tre grandi vantaggi:

- lo stesso filtro si applica ovunque nell'immagine (invarianza alla posizione),
- riduce il numero di parametri rispetto a una rete completamente connessa,
- consente alla rete di costruire una rappresentazione **gerarchica** dell'immagine.

In pratica, una CNN combina più livelli di convoluzione e pooling (operazioni che riducono le dimensioni mantenendo l'informazione più importante). Dopo vari passaggi, l'output passa a strati **fully connected** che producono la classificazione finale.



Il problema della profondità e la ResNet

Una delle scoperte più importanti nel deep learning è che **reti più profonde** (con molti strati) tendono ad avere prestazioni migliori, perché possono imparare rappresentazioni più complesse. Tuttavia, aumentando la profondità, si presenta un problema: **il gradiente tende a sparire o esplodere**, rendendo difficile l'addestramento.

Qui entra in gioco **ResNet (Residual Network)**, introdotta da Microsoft nel 2015. L'idea chiave è l'uso dei **residual blocks**, ovvero connessioni che permettono al segnale di "saltare" alcuni strati. In questo modo, la rete impara una funzione residua F(x) rispetto all'input x, facilitando la propagazione dell'informazione anche in architetture molto profonde. Grazie a questa innovazione, modelli come **ResNet-50**, **ResNet-101** o **ResNet-152** hanno raggiunto performance eccezionali in competizioni come ImageNet, con migliaia di categorie da classificare.

Oggi, ResNet è uno **standard di riferimento**: molte reti moderne (DenseNet, EfficientNet, Vision Transformers) si ispirano alla sua struttura.

La rete non nasce "sapendo" riconoscere immagini. Impara attraverso un processo detto **addestramento**, che consiste in:

1. Input: si passa un'immagine etichettata (es. "questo è un gatto").
2. Forward pass: la rete produce una predizione (es. 70% gatto, 30% cane).
3. Errore: si confronta la predizione con l'etichetta corretta usando una funzione di perdita (**loss function**).
4. Backpropagation: si calcolano gli errori strato per strato.
5. Ottimizzazione: i pesi vengono aggiornati tramite un algoritmo (es. **stochastic gradient descent**, Adam) per ridurre l'errore.

Con architetture come **ResNet**, si riesce ad addestrare reti con anche **centinaia di strati** senza i problemi di instabilità tipici delle reti tradizionali.

Alla fine dell'addestramento, la rete è in grado di ricevere un'immagine mai vista e produrre una **probabilità per ogni classe**. Ad esempio:

Gatto: 0.92
Cane: 0.06
Altro: 0.02

La classe con la probabilità più alta diventa l'etichetta assegnata.

ResNet ha rappresentato una svolta storica perché ha reso possibile:

l'addestramento di reti **estremamente profonde**;
il raggiungimento di prestazioni da record in classificazione di immagini;

la creazione di un'**architettura modulare** facilmente estendibile, oggi usata non solo per la classificazione, ma anche per **object detection**, **segmentazione semantica** e persino in modelli generativi.

Attività pratica n.1

Classificatore di immagini con ML.NET e Model Builder

ML.NET è un **framework open source di machine learning sviluppato da Microsoft** che permette agli sviluppatori .NET di creare, addestrare e utilizzare modelli di intelligenza artificiale direttamente nelle applicazioni scritte in C#.

Con ML.NET si possono realizzare diversi tipi di soluzioni, ad esempio:

Classificazione (es. riconoscere se un'email è spam o no).
Regressione (es. prevedere un prezzo).
Clustering (es. raggruppare clienti in segmenti).
Elaborazione del linguaggio naturale (es. analisi del sentiment di un testo).
Riconoscimento immagini (usando anche modelli pre-addestrati)

Il **Model Builder** è uno strumento grafico integrato in **Visual Studio** che semplifica ulteriormente l'uso di ML.NET.
In pratica, permette di **creare un modello di machine learning senza scrivere codice complesso**: basta seguire una procedura guidata in pochi passi.

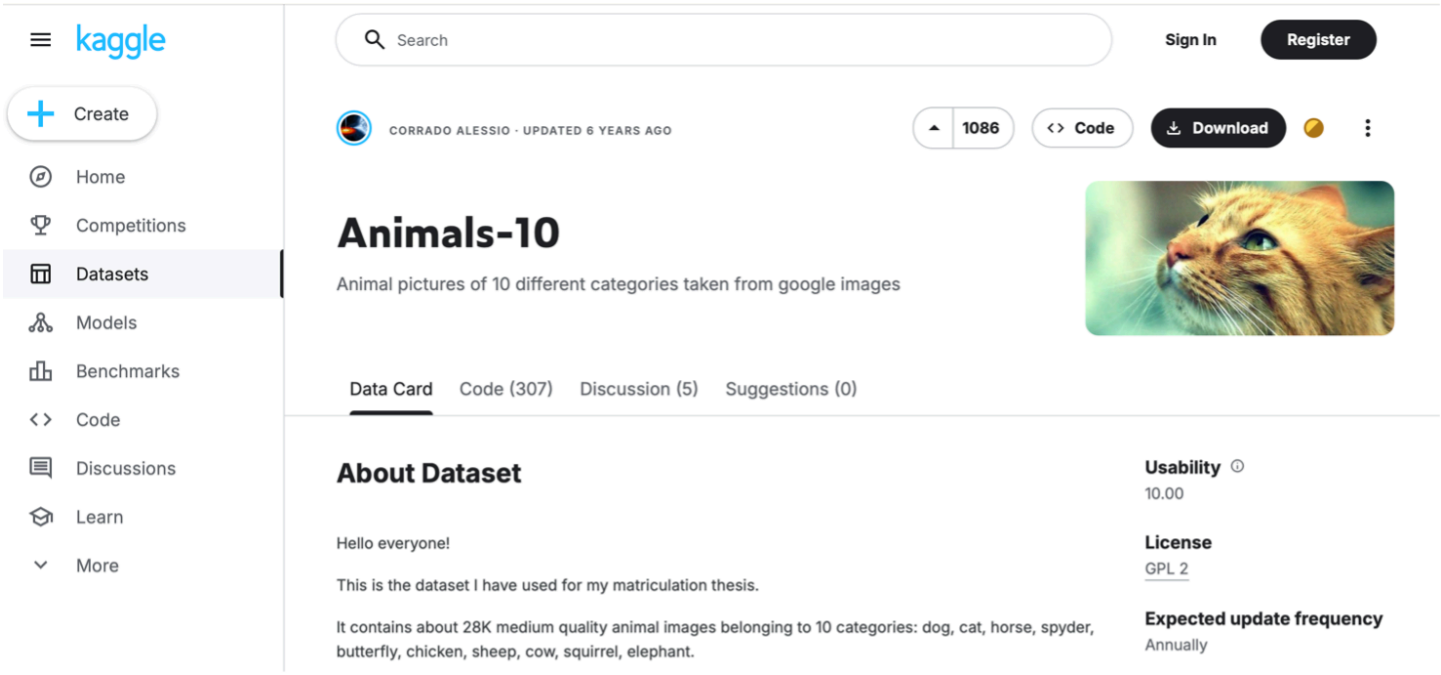
Il flusso tipico con Model Builder è:

1. Selezionare lo scenario → ad esempio classificazione, previsione, rilevamento anomalie.
2. Caricare i dati → un file CSV, un database SQL, o altre sorgenti.
3. Addestrare il modello → Model Builder prova diversi algoritmi e trova quello migliore.
4. Valutare i risultati → mostra metriche come accuratezza, precisione, errore medio, ecc.
5. Generare il codice → viene prodotto automaticamente il codice C# e il modello .zip da usare nell'applicazione.

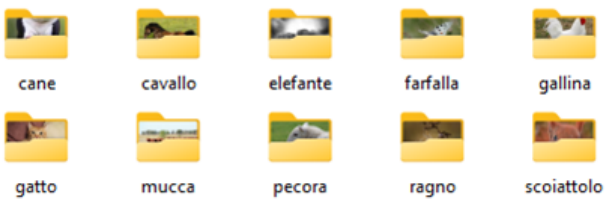
In questo modo, anche chi non ha esperienza di machine learning può **integrare l'AI in un'app .NET** con relativa facilità.

In questa attività realizziamo un classificatore di immagini. Per farlo dobbiamo installare **Visual Studio 2022 Community Edition** con il supporto per lo **Sviluppo .NET Desktop** e quindi l'estensione **ML.NET Model Builder**.

Ci servirà anche un dataset di immagini per addestrare la rete. Abbiamo scelto il dataset Animals-10 dal sito Kaggle (**https://www.kaggle.com/datasets/alessiocorrado99/animals10?resource=download**) che contiene circa 28.000 immagini di animali appartenenti a 10 diverse categorie.

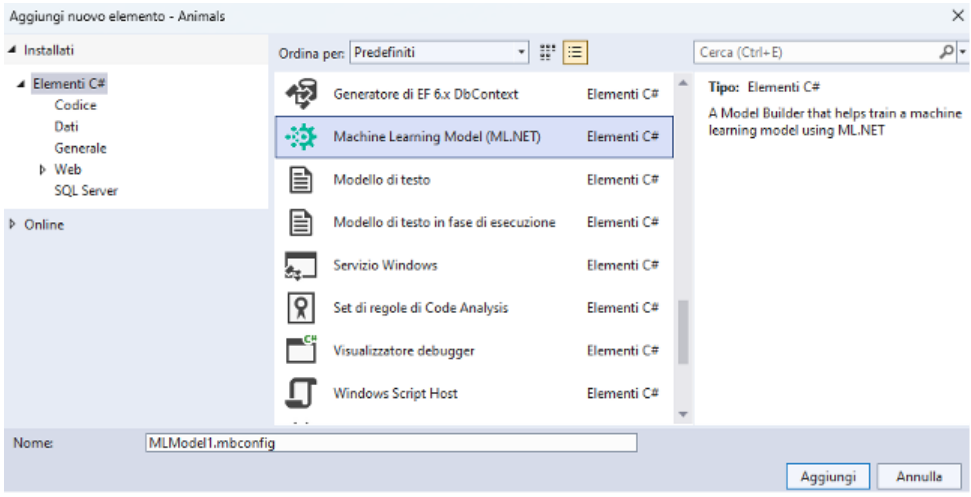


Una volta scaricato il dataset e scompattato il file .zip otteniamo una directory row-img contenente all'interno il dataset di immagini suddiviso in 10 cartelle: il nome della cartella corrisponde all'etichetta dei dati.

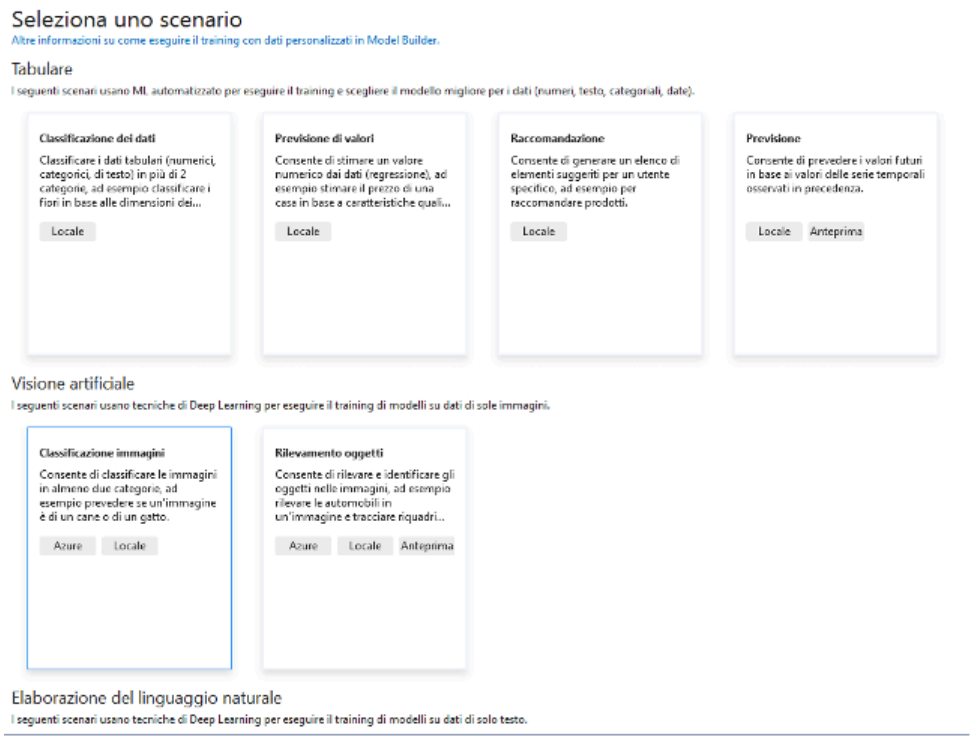


Ora siamo pronti per creare l'applicazione. Apriamo Visual Studio e creiamo un nuovo progetto di tipo **App Console**, quindi diamo un nome al progetto ad esempio *Animals* e scegliamo infine il framework **.NET 8.0**.

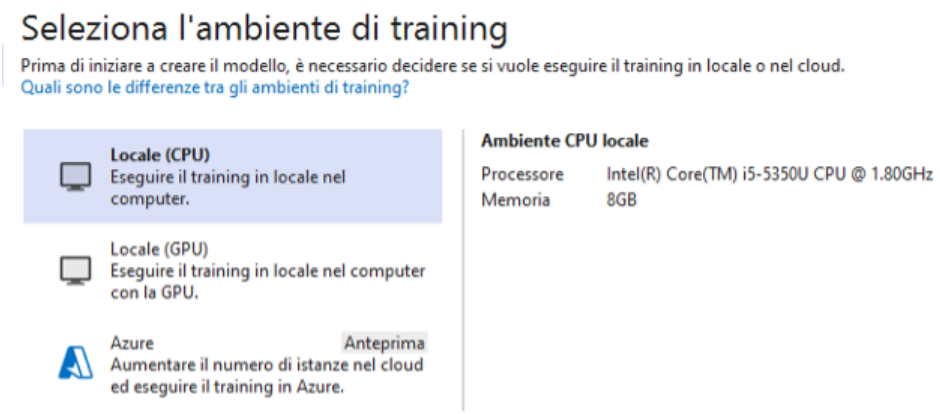
Nel riquadro **Esplora soluzioni** facciamo clic con il tasto destro del mouse sul nome del progetto *Animals*, scegliamo dal menu la voce **Aggiungi** > **Modello di Machine Learning** e aggiungiamo il file al progetto.



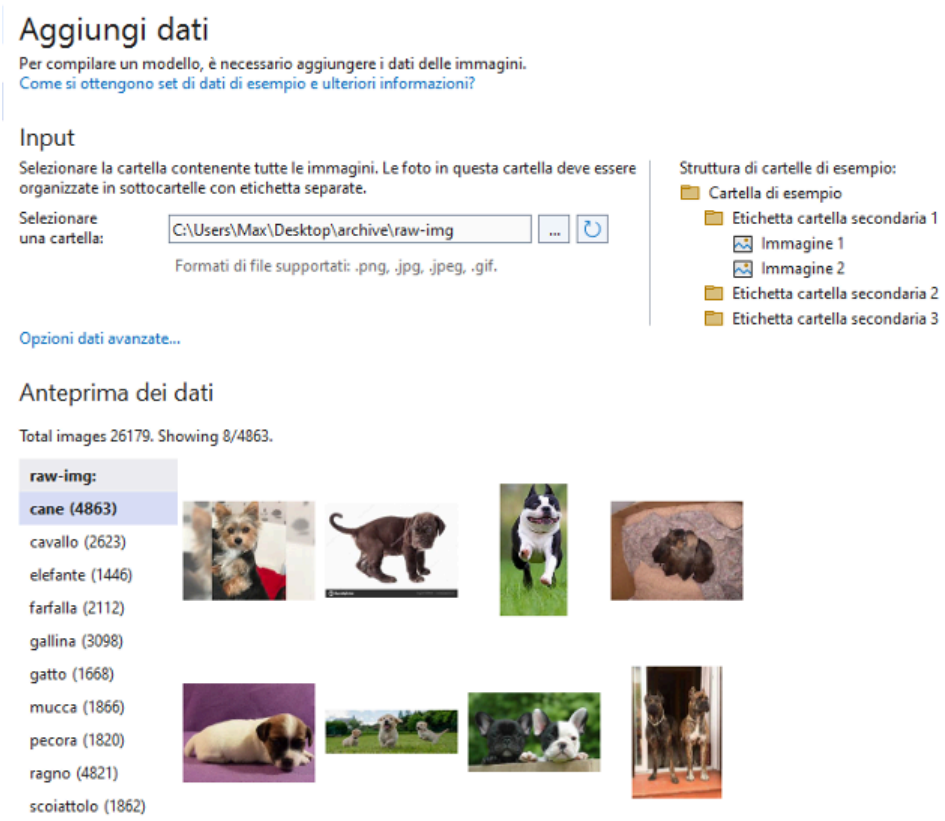
Inizia la procedura guidata per la creazione del modello di machine learning desiderato. È possibile scegliere tra diversi scenari: tabulare, visione artificiale, elaborazione del linguaggio naturale. Scegliamo **Classificazione Immagini** nella sezione Visione artificiale.



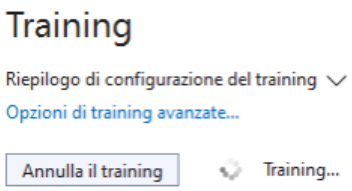
Nel passaggio successivo è possibile scegliere l'ambiente di **training** tra cui: **locale** usando la **CPU** o la **GPU** oppure un server **Azure**. La scelta del training locale su CPU è quasi sempre possibile mentre l'uso della GPU (più veloce) richiede la presenza di GPU compatibili con CUDA. Selezioniamo la prima opzione.



Nel prossimo passaggio dobbiamo fornire i dati per il training. In questo modello di machine learning occorre fornire una **cartella di immagini** organizzate per sottocartelle separate con la relativa etichetta. Il dataset Animals10 che abbiamo scaricato è impostato in questo modo.

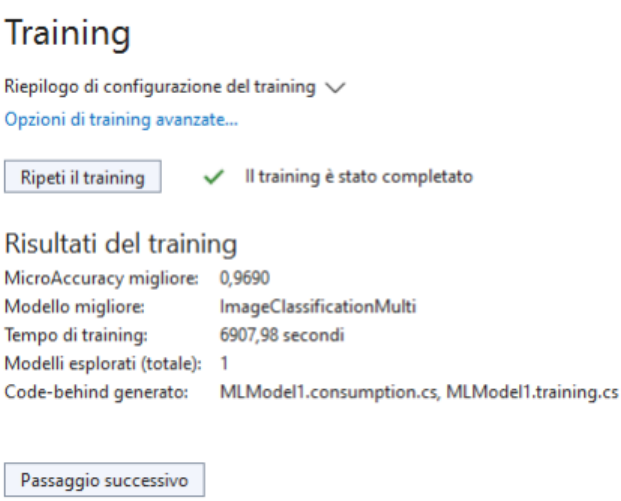


Possiamo esplorare il dataset e siamo pronti per **avviare il training** al passaggio successivo.



La durata del training dipende dalla dimensione del dataset e dalla potenza di elaborazione del computer (potrebbe richiedere anche qualche decina di minuti).
Durante il training viene scaricato automaticamente il modello di rete più adatto al problema. In questo caso la rete RestNet50 sviluppata da Microsoft. In alcune situazioni il download fallisce con questo messaggio:

Non resta che scaricare manualmente il file (circa 100MB) all'indirizzo **https://aka.ms/mlnet-resources/meta/resnet_v2_50_299.meta** e copiarlo nella cartella AppData\Local\Temp\MLNET dell'utente e rieseguire il training.



Al termine del training il prossimo passaggio ci consente di valutare il modello addestrato caricando alcune immagini non incluse nel dataset per vedere i risultati della classificazione. Se riuscite provate a caricare delle foto di animali scattate da voi (io ho provato una foto del mio cane Tato).

Siamo giunti all'ultimo passaggio dove è possibile aggiungere il codice per utilizzare il modello nella nostra applicazione. Scegliamo il modello di progetto App Console.

Il codice prodotto è il seguente: viene caricata una immagine e quindi utilizzato il metodo `PredictAllLabels` dell'oggetto MLModel1 che abbiamo realizzato con il Model Builder. Il risultato è un array associativo chiave-valore che viene visualizzato mostrando l'etichetta e la relativa percentuale di riconoscimento.

```
// This file was auto-generated by ML.NET Model Builder.

using ConsoleApp3;
using System.IO;

// Create single instance of sample data from first line of dataset for model input
var imageBytes = File.ReadAllBytes(@"C:\Users\Max\Desktop\Tato.jpeg");
MLModel1.ModelInput sampleData = new MLModel1.ModelInput()
{
   ImageSource = imageBytes,
};

// Make a single prediction on the sample data and print results.
var sortedScoresWithLabel = MLModel1.PredictAllLabels(sampleData);
Console.WriteLine($"{"Class",-40}{"Score",-20}");
Console.WriteLine($"{"-----",-40}{"-----",-20}");

foreach (var score in sortedScoresWithLabel)
{
   Console.WriteLine($"{score.Key,-40}{score.Value,-20}");
}
```

Ecco l'output del programma in esecuzione.

```
2025-08-19 16:05:26.003554: I tensorflow/core/platform/cpu_feature_guard.cc:142] This TensorFlow
binary is optimized with oneAPI Deep Neural Network Library (oneDNN)to use the following CPU
instructions in performance-critical operations:  AVX2
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2025-08-19 16:05:26.023181: I tensorflow/compiler/xla/service/service.cc:168] XLA service
0x27d004219e0 initialized for platform Host (this does not guarantee that XLA will be used). Devices:
2025-08-19 16:05:26.023886: I tensorflow/compiler/xla/service/service.cc:176]   StreamExecutor device
(0): Host, Default Version
Class                                   Score
-----                                   -----
cane                                    0,99975544
ragno                                   7,79878E-05
scoiattolo                              6,1936014E-05
farfalla                                3,7139023E-05
elefante                                3,2415417E-05
gallina                                 1,6710392E-05
cavallo                                 1,0063341E-05
gatto                                   3,99407E-06
pecora                                  2,3680086E-06
mucca                                   1,8992658E-06

C:\Users\Max\source\repos\ConsoleApp3\ConsoleApp3\bin\Debug\net8.0\ConsoleApp3.exe (processo 12476)
terminato. Codice restituito: 0 (0x0).
Per chiudere automaticamente la console quando viene arrestato il debug, abilitare Strumenti->Opzioni-
>Debug->Chiudi automaticamente la console quando viene arrestato il debug.
Premere un tasto qualsiasi per chiudere questa finestra...
```

Il punteggio percentuale più elevato corrisponde alla prima posizione della collezione quindi la rete ha riconosciuto correttamente l'immagine di un cane.