Relazione progetto Intelligenza Artificiale

Predizione della percentuale di costruzione di un edificio

Mirco Chiarini
Emanuele Corsi
Antonio Sisinni

07/02/2024

Laurea Magistrale Informatica 2023/2024

Indice

1.a. Descrizione del problema

Il problema emerso è il risultato di un'approfondita analisi e ricerca in letteratura e online, mirata a individuare alternative per determinare la percentuale di completamento di un edificio. Durante questa ricerca, è emersa una carenza significativa di risorse, quali dataset, tempistiche affidabili e informazioni sul progresso percentuale delle costruzioni. Questa lacuna potrebbe generare inefficienze nella gestione del budget destinato al cantiere e influire sulle tempistiche previste.

L'importanza di risolvere questo problema diventa evidente quando si considera il potenziale impatto positivo su diverse parti interessate. L'introduzione di uno strumento di supporto basato su dati potrebbe migliorare notevolmente la gestione delle tempistiche di costruzione, contribuendo a evitare ritardi e consentendo una pianificazione più precisa.

In prospettiva a lungo termine, la precisione delle stime temporali potrebbe aumentare, fornendo ai lavoratori un valido strumento di previsione. Gli eventuali beneficiari di una soluzione a questo problema includono le aziende costruttrici, che potrebbero godere di un calcolo più accurato delle tempistiche di costruzione, con conseguenti vantaggi economici. Inoltre, la soluzione potrebbe essere presentata come uno strumento utile anche per i clienti finali, consentendo loro di valutare con maggiore precisione lo stato di avanzamento del progetto ed eventualmente pianificandolo in modo più efficace.

In termini pratici, questa soluzione potrebbe essere integrata con previsioni temporali sulla data di completamento del cantiere, portando a un beneficio concreto sia per i costruttori che per i clienti finali, stabilendo una solida base per una gestione più efficiente e trasparente del processo di costruzione.

1.b. Soluzione proposta

Visto l'obiettivo del nostro progetto di voler predire una percentuale a partire da immagini fornite al modello, ci siamo inizialmente concentrati sulla costruzione del dataset. Dopo aver esaminato diverse raccolte di dataset online, come Kaggle, abbiamo notato che non esisteva un dataset strutturato in base alle nostre esigenze. Di conseguenza, gran parte del nostro lavoro si è concentrato inizialmente sulla costruzione e sull'etichettatura del dataset, dato che abbiamo optato per l'apprendimento supervisionato.

La nostra idea è stata quella di utilizzare una serie di timelapse di costruzione di edifici ed estrapolare dei frame, assegnandogli successivamente una classe che identifica un range di percentuale per evidenziare a che punto della costruzione ci troviamo in base alla durata totale del video. Il compito del modello è quindi analizzare la struttura delle immagini in ogni classe per predire correttamente a quale classe appartengono. Per il training e il testing, abbiamo utilizzato modelli già esistenti che abbiamo adattato alle nostre esigenze (fine-tuning).

Nella realizzazione del dataset, abbiamo affrontato un problema riguardante la qualità dei video da utilizzare. Molti dei timelapse trovati non fornivano una vista fissa sull'andamento della costruzione, ma integravano anche panoramiche sugli operai al lavoro, ad esempio. Ciò poteva portare all'estrazione di frame non coerenti con il nostro obiettivo e influire negativamente sulla predizione della percentuale. Abbiamo cercato quindi di scegliere video che non presentassero queste caratteristiche o, quando possibile, li abbiamo tagliati in base alle nostre esigenze.

Per l'implementazione del progetto abbiamo scelto Python come linguaggio di programmazione e la libreria PyQt per la progettazione di un'interfaccia per la costruzione automatica o manuale del dataset. La comprensione del funzionamento di questa libreria e la sua implementazione hanno richiesto un po' di tempo.

Infine, anche la scelta della tipologia dei modelli ha richiesto una fase di ricerca e comprensione. Abbiamo considerato il nostro problema come un problema di Image Classification, ovvero predire la classe di appartenenza a partire da un'immagine. Dopo aver esaminato diverse tipologie di algoritmi, come edge detection, support vector machine, K-Nearest Neighbors, ecc., la nostra scelta finale è ricaduta sulle Convolutional Neural Network. Abbiamo notato che sono ampiamente utilizzate per il riconoscimento delle immagini, in quanto sono in grado di apprendere le loro caratteristiche e sono adatte per compiti come la rilevazione di oggetti, la segmentazione e la classificazione.

Rassegna della letteratura:

La creazione di un programma che punta ad analizzare delle immagini in modo algoritmico costituisce una sfida per l'implementazione. Un'immagine racchiude una grandissima quantità di dati non strutturati, che possono variare a seconda del caso che stiamo analizzando (anche se di poco).

Per questa motivazione la tecnica solitamente utilizzata in questo contesto è quella di utilizzare modelli di machine learning (ML), i quali riescono ad estrarre pattern e conoscenza per risolvere un determinato problema dai dati che li riguardano. La conoscenza di tali problemi viene costruita attorno ai valori che assumono le "Feature"(caratteristiche) degli input forniti al modello.

Basandosi sul modo in cui tali feature vengono definite abbiamo una biforcazione nelle tipologie di modelli ML:

- **Modelli classici**: in cui è necessaria una prima fase di feature engineering ossia si definiscono e modellano i dati in modo da poter essere successivamente analizzati dal modello in modo efficiente.

- **Deep Neural Networks (DNN)**: in cui è il modello stesso a mappare le feature all'interno di una "rete di neuroni", capendo in modo automatico quali sono i dati su cui porre l'attenzione.

L'insieme dei dati che vengono forniti al modello per addestrarsi prende il nome di dataset, a seconda del modo in cui gli elementi del dataset sono organizzati è possibile fare un'altra distinzione:

- **Apprendimento Supervisionato**: il modello viene addestrato su un dataset etichettato, dove ogni esempio di input è associato a un'etichetta corrispondente. L'obiettivo è far apprendere al modello la relazione tra le features e le etichette in modo che possa fare previsioni su nuovi dati. Questo tipo di apprendimento è spesso utilizzato in problemi di classificazione e regressione.

- **Apprendimento Non Supervisionato**: al contrario, nell'apprendimento non supervisionato, il modello viene addestrato su dati non etichettati senza informazioni sulla corrispondenza tra input e output. L'obiettivo è scoprire pattern intrinseci, strutture o raggruppamenti nei dati. Questo tipo di apprendimento è spesso utilizzato in algoritmi di clustering o riduzione della dimensionalità.

Per l'analisi di immagini vengono utilizzate le DNN, evitando così la fase di feature engineering che richiederebbe altrimenti di prestare attenzione a diversi domini

molto vasti (i quali includono campi come la biomedicina, il tracciamento di veicoli e molto altro…) per riuscire ad estrarre feature utili dalle immagini.

Per i problemi di classificazione di immagini, viene spesso adottato un approccio di apprendimento supervisionato. Il dataset utilizzato per addestrare il modello è composto da una serie di immagini, ciascuna delle quali è associata a un'etichetta che indica la classe di appartenenza dell'immagine. Queste etichette vengono utilizzate dal modello durante il processo di addestramento per imparare a riconoscere le caratteristiche distintive di un'immagine appartenente a una classe specifica.

Durante l'addestramento, il modello analizza le immagini del dataset e, modificando il peso delle connessioni tra neuroni nei livelli nascosti (intermedi), impara a mappare le caratteristiche dell'immagine alle rispettive classi. Le funzioni di attivazione, come la ReLU o altre varianti, introducono non linearità nella rete, permettendo al modello di catturare relazioni complesse tra le caratteristiche delle immagini.

Il processo di addestramento coinvolge l'utilizzo dell'algoritmo di retropropagazione dell'errore per aggiornare i pesi della rete in modo che l'errore complessivo tra le previsioni del modello e le etichette reali diminuisca.

In pratica, il modello apprende a riconoscere automaticamente le caratteristiche discriminanti nelle immagini durante l'addestramento, in modo che possa generalizzare e fare previsioni accurate su nuove immagini non appartenenti al dataset di addestramento.

I layer nascosti che compongono una CNN sono di 3 tipologie:

- **Convolution Layer**: layer che vengono utilizzati per effettuare l’estrazione delle feature dall'input, tramite questi livelli si andranno a costituire le cosiddette feature map che saranno utilizzate nei livelli più alti come soggetto di “high level reasoning”.

- **Pooling Layer**: questi layer sono utilizzati per tenere sotto controllo il fenomeno dell’over fitting, vanno infatti a diminuire il numero di parametri utilizzati per l’analisi. È possibile applicare diverse discipline di pooling andando a trattare così in modo diverso i parametri che vengono “uniti” per arrivare ad un più giusto numero di informazioni trattate.

- **Fully Connected Layer**: layer utilizzati per “high level reasoning”, per ogni neurone presente in uno di questi layer esiste una connessione verso ognuno di quelli del livello successivo.

I modelli di deep learning si dimostrano, al contrario di altri algoritmi di machine learning, in grado di ottenere un grado di precisione direttamente proporzionale alla dimensione del dataset con il quale vengono addestrati. Essi, infatti, sembrano non saturare le informazioni che possono estrarre. Al contrario, per ottenere buone performance, necessitano infatti (in alcuni casi) di quantità di dati minime esorbitanti.

Per questa motivazione se il materiale a disposizione per l'addestramento non risulta essere sufficiente è possibile utilizzare alcune reti che risultano già precedentemente addestrate con grandi dataset di immagini (come ad esempio ImageNet, CIFAR10, o CIFAR100). Successivamente sarà possibile andare ad effettuare un processo di fine tuning utilizzando il dataset a nostra disposizione, così da ottenere un modello specializzato nella mansione di nostro interesse.

Anche la creazione di un dataset di immagini implica l'osservazione di diversi passaggi, tra cui la raccolta delle immagini, la loro annotazione e la loro organizzazione in un formato utilizzabile:

1. **Raccolta delle immagini**: il primo passo nella creazione di un dataset di immagini è la raccolta delle immagini stesse. Questo può essere fatto in vari modi, ad esempio scaricando immagini da Internet, acquisendo immagini tramite fotocamere o scanner, o utilizzando immagini esistenti da altri dataset. È importante assicurarsi che le immagini siano di alta qualità e rappresentino adeguatamente la varietà di scenari che si desidera che il modello di apprendimento automatico sia in grado di gestire.

2. **Annotazione delle immagini**: una volta raccolte le immagini, il passo successivo è l'annotazione. Questo processo implica l'aggiunta di etichette o metadati alle immagini che descrivono ciò che contengono. Questo può essere un processo molto laborioso e richiede spesso l'intervento umano, anche se esistono strumenti che possono aiutare a semplificare il processo, come il tool di annotazione.

3. **Organizzazione del dataset**: infine, le immagini e le loro annotazioni devono essere organizzate in un formato che può essere facilmente utilizzato per l'addestramento del modello di apprendimento automatico. Questo potrebbe implicare la divisione del dataset in set di addestramento, validazione e test, e la memorizzazione delle immagini e delle annotazioni in un formato facilmente accessibile.

È importante notare che la creazione di un dataset di immagini può presentare sfide legali ed etiche. Ad esempio, se le immagini sono raccolte da Internet, potrebbe essere necessario ottenere il permesso dei detentori dei diritti d'autore.

I paper utilizzati per la rassegna della letteratura si trovano nella sezione della bibliografia [1] [2] [3] [4] [5].

Per quanto riguarda la metodologia di lavoro e la suddivisione dei compiti abbiamo scelto di lavorare insieme inizialmente nella costruzione del dataset, al fine di avere una visione comune sulla sua struttura e composizione. In seguito, abbiamo deciso di suddividere così il lavoro:

- Emanuele e Mirco: Implementazione interfaccia grafica per la costruzione del dataset
- Antonio: Revisione della letteratura e scelta dei modelli da poter implementare
- Antonio, Emanuele, Mirco: Effettiva implementazione e adattamento dei modelli scelti

I risultati ottenuti dall'implementazione sono abbastanza soddisfacenti dal punto di vista dell'accuracy ottenuta durante la fase di training, avvicinandosi molto al 100%. Meno positivo è stato il testing, nel quale abbiamo raggiunto un valore massimo di accuracy di circa il 50%, considerando 20 classi. Abbiamo notato che il numero delle classi in cui abbiamo deciso di dividere il dataset incide molto sulla precisione, difatti effettuando una prova con 4 classi il modello ha fornito nel testing una precisione dell' 80-90%. Avere molte classi significa avere immagini con caratteristiche molto simili in classi adiacenti, e il modello fatica ad assegnare un range di percentuale di costruzione preciso.

In generale, abbiamo notato che durante le prime esecuzioni, i modelli migliorano gradualmente sia nei valori di loss che di accuracy. Di conseguenza, abbiamo deciso di effettuare delle esecuzioni più prolungate, estendendole fino a 30 epoche, per valutare fino a che punto il modello è in grado di mostrare miglioramenti. Tuttavia, è emerso che a un certo punto i miglioramenti non si verificano più; anzi, in alcuni casi si sono riscontrati peggioramenti nelle prestazioni.

2.a. Scelta della soluzione

Il primo passo nella scelta della soluzione ha coinvolto la valutazione degli strumenti e delle tecniche più adatte per implementare il nostro dataset, considerando la mancanza di dati etichettati. Abbiamo optato per Python come linguaggio di programmazione principale, sfruttando le numerose librerie disponibili, come cv2 e PyQt. Queste librerie ci hanno permesso di estrarre e selezionare singoli frame da video timelapse ed etichettarli. Inoltre, abbiamo implementato un'interfaccia grafica che consente di interagire con i video, scegliendo il numero di frame e classi da selezionare per ogni video. La nostra interfaccia grafica ha anche fornito la possibilità di apportare modifiche manuali ai frame, eliminando manualmente quelli non significativi o in maniera automatica.

Successivamente, nell'ambito dell'Image Classification, abbiamo esaminato diverse possibili soluzioni, come l'edge detection, la Support Vector Machine (SVM), il K-Nearest Neighbors (K-NN), eccetera. In particolare, è stata scelta la possibilità di lavorare con le Convolutional Neural Network (CNN) per diverse motivazioni, tra cui:

- Sono progettate per apprendere gerarchie di caratteristiche dalle immagini in modo automatico. Ciò significa che sono in grado di identificare pattern e feature complesse a diversi livelli di astrazione, consentendo una rappresentazione più ricca e dettagliata delle immagini.
- Sono particolarmente adatte per gestire dati complessi come immagini, dove la relazione tra pixel adiacenti è importante. Edge detection, SVM e K-NN possono presentare limitazioni nella gestione di questa complessità.
- Apprendono automaticamente invarianti rispetto a traslazioni e scalature, conferendo loro robustezza alle variazioni di posizione e dimensioni degli oggetti nell'immagine.
- Sono efficienti nell'apprendere in modo automatico le features rilevanti durante il processo di addestramento. A differenza di edge detection e metodi basati su features manualmente progettate, le CNN riducono la necessità di una conoscenza approfondita del dominio e di un lavoro manuale intensivo.
- Sono progettate per gestire grandi quantità di dati, una caratteristica che può essere una sfida per SVM e K-NN, soprattutto con dataset ad alta dimensionalità come le immagini.
- Dimostrano una buona capacità di generalizzazione su nuovi dati, contribuendo ad evitare il sovradattamento rispetto a SVM e K-NN, che possono essere più sensibili a dati di addestramento rumorosi.

- Possono gestire immagini di dimensioni diverse grazie all'uso di layer di pooling e convoluzioni, offrendo flessibilità nell'applicazione a vari scenari.
- Per compiti di Image Classification complessi, come il riconoscimento di oggetti in scene intricate, le CNN sono diventate lo standard de facto. Sono in grado di modellare relazioni spaziali complesse tra pixel, aspetto che edge detection e K-NN possono avere difficoltà a catturare.
- Sono in grado di automatizzare il processo di feature engineering, eliminando la necessità di progettare manualmente le features, come richiesto in alcune implementazioni di edge detection e SVM.

Dopo la selezione delle CNN, siamo andati sulla pagina ufficiale di Keras Application, libreria open-source per la costruzione e l'addestramento di reti neurali in python, e abbiamo visionato la disponibilità e le performance dei modelli di deep learning inclusi (vedi Fig 2.1). Abbiamo deciso di scegliere due modelli diversi per poter effettuare un eventuale confronto dei risultati, per cui la nostra scelta finale è ricaduta su due algoritmi ben consolidati: ResNet50 e VGG19. Nonostante la figura mostri che non siano i più efficienti, in realtà offrono una serie di vantaggi nei termini del nostro problema. Prima di tutto, sono modelli pre-addestrati su grandi dataset (es. ImageNet) e il trasferimento di conoscenze consente di utilizzare queste reti come estrattori di features per problemi specifici. Inoltre, hanno dimostrato ottime performance in competizioni di Visual Recognition e sono ampiamente adottati in progetti di ricerca e applicazioni pratiche, visto anche il notevole supporto online esistente. Per cui, in conclusione, abbiamo utilizzato in python la libreria Tensorflow che ci consente di importare i modelli di Keras.

Model	Size (MB)	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth	Time (ms) per inference step (CPU)	Time (ms) per inference step (GPU)
Xception	88	79.0%	94.5%	22.9M	81	109.4	8.1
VGG16	528	71.3%	90.1%	138.4M	16	69.5	4.2
VGG19	549	71.3%	90.0%	143.7M	19	84.8	4.4
ResNet50	98	74.9%	92.1%	25.6M	107	58.2	4.6
ResNet50V2	98	76.0%	93.0%	25.6M	103	45.6	4.4
ResNet101	171	76.4%	92.8%	44.7M	209	89.6	5.2
ResNet101V2	171	77.2%	93.8%	44.7M	205	72.7	5.4
ResNet152	232	76.6%	93.1%	60.4M	311	127.4	6.5
ResNet152V2	232	78.0%	94.2%	60.4M	307	107.5	6.6
InceptionV3	92	77.9%	93.7%	23.9M	189	42.2	6.9
InceptionResNetV2	215	80.3%	95.3%	55.9M	449	130.2	10.0
MobileNet	16	70.4%	89.5%	4.3M	55	22.6	3.4
MobileNetV2	14	71.3%	90.1%	3.5M	105	25.9	3.8
DenseNet121	33	75.0%	92.3%	8.1M	242	77.1	5.4
DenseNet169	57	76.2%	93.2%	14.3M	338	96.4	6.3
DenseNet201	80	77.3%	93.6%	20.2M	402	127.2	6.7
NASNetMobile	23	74.4%	91.9%	5.3M	389	27.0	6.7
NASNetLarge	343	82.5%	96.0%	88.9M	533	344.5	20.0
EfficientNetB0	29	77.1%	93.3%	5.3M	132	46.0	4.9
EfficientNetB1	31	79.1%	94.4%	7.9M	186	60.2	5.6
EfficientNetB2	36	80.1%	94.9%	9.2M	186	80.8	6.5
EfficientNetB3	48	81.6%	95.7%	12.3M	210	140.0	8.8
EfficientNetB4	75	82.9%	96.4%	19.5M	258	308.3	15.1
EfficientNetB5	118	83.6%	96.7%	30.6M	312	579.2	25.3
EfficientNetB6	166	84.0%	96.8%	43.3M	360	958.1	40.4
EfficientNetB7	256	84.3%	97.0%	66.7M	438	1578.9	61.6
EfficientNetV2B0	29	78.7%	94.3%	7.2M	-	-	-
EfficientNetV2B1	34	79.8%	95.0%	8.2M	-	-	-
EfficientNetV2B2	42	80.5%	95.1%	10.2M	-	-	-
EfficientNetV2B3	59	82.0%	95.8%	14.5M	-	-	-
EfficientNetV2S	88	83.9%	96.7%	21.6M	-	-	-
EfficientNetV2M	220	85.3%	97.4%	54.4M	-	-	-
EfficientNetV2L	479	85.7%	97.5%	119.0M	-	-	-
ConvNeXtTiny	109.42	81.3%	-	28.6M	-	-	-
ConvNeXtSmall	192.29	82.3%	-	50.2M	-	-	-
ConvNeXtBase	338.58	85.3%	-	88.5M	-	-	-
ConvNeXtLarge	755.07	86.3%	-	197.7M	-	-	-
ConvNeXtXLarge	1310	86.7%	-	350.1M	-	-	-

Fig 2.1. Modelli di deep learning in Keras.

L'addestramento dei modelli è stato eseguito utilizzando l'80% dei dati per il training e il restante 20% per il testing. Abbiamo configurato le dimensioni delle immagini e il numero di passaggi per ogni epoca, successivamente avviando i modelli con gli stessi parametri. L'impiego è avvenuto in maniera sequenziale, senza effettuare il retraining dei pesi, ma utilizzando un grande database (ImageNet) pre-addestrato.

Abbiamo configurato i modelli con diverse funzioni di attivazione, strutturando uno strato con 512 neuroni con funzione di attivazione relu e l'ultimo strato con il numero di neuroni corrispondente alle classi nel nostro dataset, utilizzando una funzione di attivazione softmax per convertire l'output in probabilità distribuite tra le varie classi.

Per quanto riguarda la valutazione dei modelli abbiamo scelto di usare come funzione di loss la categorical_crossentropy che è comunemente utilizzata nei problemi di classificazione multiclasse in cui ogni campione può appartenere a una sola classe tra diverse classi possibili. Questa loss function è particolarmente adatta quando si lavora con output softmax (che abbiamo usato), dove ogni classe ha una probabilità associata e la somma di tutte le probabilità è 1. Il suo ruolo è quello di calcolare la discrepanza tra le probabilità predette dal modello e le probabilità reali delle classi.

Invece, per quanto concerne le metriche di valutazione abbiamo scelto l'accuracy che effettua il rapporto tra il numero di previsioni corrette ed il numero totale di previsioni.

Infine, abbiamo scelto come funzione di ottimizzazione "Adam", in Fig 2.2. Adam, acronimo di "Adaptive Moment Estimation," è un algoritmo di ottimizzazione ampiamente utilizzato in machine learning per aggiornare iterativamente i pesi di un modello durante la fase di addestramento. Abbiamo visto che esistono varie alternative a quest'ultima come la Stochastic Gradient Descent, RMSprop, AdaGrad, Adadelta e Nadam. Rifacendoci, però, agli indici di performance riportati in letteratura abbiamo potuto notare che Adam spesso risulta essere la soluzione migliore anche se di poco, si veda in Fig 2.3. Ovviamente poi la scelta può dipendere dal tipo del problema e dalle sue caratteristiche, però ci sembrava comunque opportuno usare Adam vista la sua buona gestione di grandi dataset, robustezza e la necessità minima di fine-tuning.

```
resnet_model.compile(optimizer='adam',loss='categorical_crossentropy',metrics=['accuracy'])
```

Fig 2.2. Configurazione modello.

Table 4. Results of the models used in the study

Neuron Number of	Activation Function	Model No	Optimization Method	Accuracy	Error	Test Accuracy	Test Error
32	ReLU	1	Sgd	0.98	0.02	0.70	0.30
		2	Adagrad	0.99	0.01	0.75	0.25
		3	Nadam	0.99	0.004	0.86	0.14
		4	Adam	0.99	0.003	0.86	0.14
		5	Rmsprop	0.99	0.004	0.84	0.16
	Tanh	6	Sgd	0.98	0.02	0.74	0.26
		7	Adagrad	0.99	0.009	0.73	0.27
		8	Nadam	0.99	0.003	0.87	0.13
		9	Adam	0.99	0.004	0.83	0.17
		10	Rmsprop	0.99	0.006	0.78	0.22
64	ReLU	11	Sgd	0.98	0.02	0.71	0.29
		12	Adagrad	0.99	0.005	0.80	0.20
		13	Nadam	0.99	0.002	0.87	0.13
		14	Adam	0.99	0.003	0.89	0.11
		15	Rmsprop	0.99	0.002	0.92	0.08
	Tanh	16	Sgd	0.98	0.02	0.75	0.25
		17	Adagrad	0.99	0.006	0.78	0.22
		18	Nadam	0.99	0.003	0.84	0.16
		19	Adam	0.99	0.004	0.82	0.18
		20	Rmsprop	0.99	0.007	0.78	0.22

Fig 2.3. Risultati dei metodi di ottimizzazione in relazione alle funzione di attivazione.

3.a. Dimostrazioni e tecnologie

Istruzioni per l'esecuzione:

1. Scaricare la repository in questione.
2. Per la creazione del dataset avviare il file "main.py" nella directory src/scripts (assicurarsi di sostituire nel codice i percorsi per la creazione della cartella provvisoria nella quale saranno generati i frame per la costruzione "manuale" del dataset).
3. Una volta aperta l'interfaccia grafica selezionare la cartella dal quale prendere i video, la cartella nel quale riporre i frame successivamente estratti, il numero di frame che si vogliono estrarre da ogni video e il numero delle classi nel quale dividere il dataset. Infine, selezionare se effettuare un'esecuzione automatica o manuale (scegliere i frame uno ad uno).
4. Una volta terminato si possono avviare i modelli, sempre facendo attenzione a mettere il giusto percorso della cartella in cui è presente il dataset composto precedentemente. I file da eseguire sono nella directory src/notebooks e hanno estensione .ipyb.
5. Come ultimo passo dell'esecuzione di questi file verranno create le cartelle relative ai modelli appena addestrati con lo scopo di salvare lo stato delle reti.
6. Per poter testare i modelli c'è un template al percorso tests/main.ipynb. Per utilizzarlo bisogna fare attenzione nel definire l'array delle classi "class_names", a inserire i percorsi corretti di estrazione delle immagini e a richiamare col nome corretto il modello da utilizzare.

Il progetto è stato completamente integrato utilizzando Python, sfruttando diverse versioni come Python 3.9 e 3.10. Le librerie principali impiegate includono TensorFlow 2.15.0 e Keras 2.15.0 per la creazione della rete neurale e del modello, OpenCV 4.9.0.80 per la manipolazione del dataset attraverso la selezione dei video, PyQt5 5.15.10 per l'implementazione dell'interfaccia grafica del dataset, Matplotlib 3.8.2 per la visualizzazione dei grafici delle metriche dei modelli, e NumPy 1.26.3 per la gestione matriciale e vettoriale dei modelli.

Il dataset è stato sviluppato mediante Visual Studio Code, mentre il modello è stato implementato attraverso un Jupyter Notebook, entrambi con l'uso di Python.

Il modello è stato eseguito su una macchina con sistema operativo Windows 10, 16 GB di RAM, processore i7-10710 e GPU NVIDIA GeForce GTX 1650.

Il codice è stato gestito in locale sui vari computer del team, utilizzando GitLab come repository per effettuare push e pull e mantenere il codice aggiornato. Per gestire il trasferimento di contenuti pesanti come dataset e modelli, è stato implementato Git LFS.

Le dipendenze necessarie sono state elencate nel file requirements, che indica le librerie principali da installare, tra cui TensorFlow, Matplotlib, OpenCV, PyQt5 e NumPy.

Sono stati implementati due modelli, VGG19 e ResNet50, con le stesse configurazioni per evidenziare eventuali vantaggi e svantaggi. Prima di avviarli, è stato necessario configurare l'utilizzo dei dati provenienti dal dataset, determinando la loro destinazione come parte del training o del testing, si veda in Fig 3.1 e 3.2. La configurazione prevedeva la suddivisione percentuale del contenuto all'interno del dataset, specificando il percorso di origine, le dimensioni da rispettare, il seed di partenza per garantire la riproducibilità della suddivisione dei dati. Inoltre, le etichette sono state rappresentate utilizzando l'encoding one-hot, particolarmente utile in situazioni di classificazione con più classi. Infine, è stato selezionato il numero di batch, rappresentante il numero di campioni o esempi utilizzati per calcolare un singolo passaggio di ottimizzazione.

I modelli sono stati avviati in modo sequenziale, computando un passo alla volta. Successivamente, il modello è stato configurato con diverse impostazioni, tra cui l'esclusione dei livelli completamente connessi (top layer) della rete. Questa scelta è stata effettuata poiché si stava creando un nuovo modello con un numero diverso di classi. È stata impostata la forma dell'input del modello derivante dai dati del dataset. Inoltre, è stata specificata l'operazione di pooling finale utilizzando la media globale, contribuendo a ridurre il numero di parametri e computazioni nelle fasi successive della rete. Questa strategia aiuta a mitigare il rischio di overfitting e migliorare la capacità di generalizzazione del modello. Il numero di classi è stato definito in base al dataset, e sono stati utilizzati pesi pre-addestrati su ImageNet.

Successivamente, è stato necessario iterare su ogni livello pre-addestrato per impostare a falso il valore "trainable". Questo implica che i pesi di tali livelli non verranno aggiornati durante il processo di addestramento del nuovo modello. In pratica, si stanno congelando i pesi del modello pre-addestrato per preservare le caratteristiche apprese su ImageNet. È stato aggiunto uno strato di flattening per convertire l'output tridimensionale del modello in un vettore monodimensionale, spesso necessario prima di collegare strati densi (fully connected) alla fine del modello. Infine, sono stati aggiunti uno strato completamente connesso (dense) con 512 neuroni e funzione di attivazione ReLU, introducendo non linearità nel modello. Un ulteriore strato completamente connesso con 20 neuroni, uno per ogni classe nel task di classificazione, è stato aggiunto utilizzando la funzione di attivazione softmax. Quest'ultima è comunemente utilizzata nell'ultimo strato di modelli di classificazione

multiclasse per convertire gli output in probabilità, semplificando l'interpretazione come distribuzione di probabilità su diverse classi, configurazione del modello alla Fig 3.3.

È stato selezionato l'algoritmo di ottimizzazione "Adam", ampiamente utilizzato per addestrare reti neurali, in grado di regolare automaticamente il tasso di apprendimento durante l'addestramento, garantendo un equilibrio tra efficienza e prestazioni. La funzione di perdita specificata è 'categorical_crossentropy', comunemente utilizzata in problemi di classificazione multiclasse con etichette di destinazione codificate in modo one-hot, in quanto permette di misurare la discrepanza tra le previsioni del modello e le etichette reali. Infine, è stata inserita la metrica 'accuracy', rappresentante la frazione delle previsioni corrette rispetto al totale.

Il dataset è fornito insieme agli script del modello, consentendo la creazione immediata del dataset o l'utilizzo di frame già presenti nel progetto. Il modello è stato eseguito su diverse epoche per osservare il comportamento nei vari passaggi, rispettando i parametri impostati precedentemente per i batch.

```
#set delle immagini che verranno utilizzate come training
img_height,img_width=180,180
batch_size=32  #numero passi di ottimizzazione in un epoca (ogni epoca addestra su tutto il dataset che viene suddiviso in vari batch)
train_ds = tf.keras.preprocessing.image_dataset_from_directory(
  data_dir,
  validation_split=0.2, # 20% dataset per il test
  subset="training",
  seed=123, #garantire che la suddivisione dei dati sia riproducibile
  label_mode='categorical', #le etichette vengono rappresentate in one-hot encoding (casistica di classificazione con più classi)
  image_size=(img_height, img_width),
  batch_size=batch_size)
```

Fig 3.1. Configurazione del codice per le immagini di training.

```
#set delle immagini che verranno utilizzate come testing
val_ds = tf.keras.preprocessing.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="validation",
  seed=123,
  label_mode='categorical',
  image_size=(img_height, img_width),
  batch_size=batch_size)
```

Fig 3.2. Configurazione del codice per le immagini di testing.

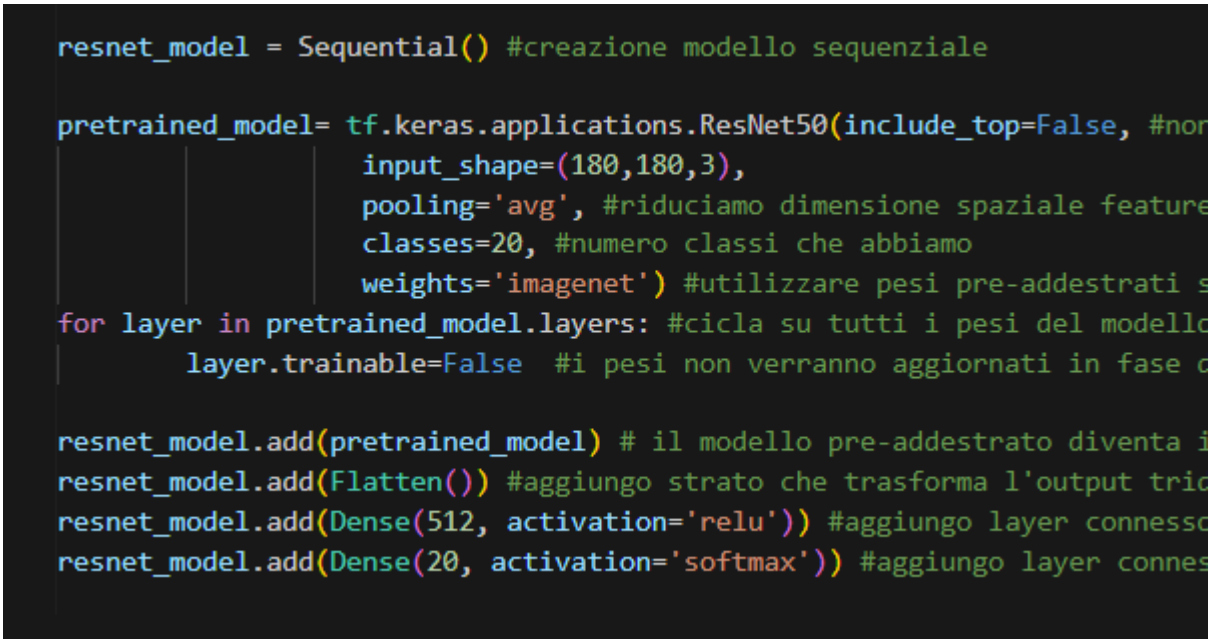

Fig 3.3. Configurazione codice del modello.

Infine è stato avviato l'addestramento con un numero abbastanza elevato di epoche per verificare l'andamento delle performance e valutare i risultati. Per cui, in seguito abbiamo implementato una parte di codice che valuta l'andamento del modello (parametro "monitor" in Fig. 3.4) in un certo numero di iterazioni (parametro "patience" in Fig 3.4) e se non avviene un miglioramento l'esecuzione si interrompe e vengono salvati i pesi registrati nell'epoca migliore. Così poi possiamo procedere a salvare il modello nella sua "condizione migliore".

Fig 3.4. Interruzione esecuzione modello.

3.b. Risultati

Nel confronto tra modelli ResNet50 è risultato essere il migliore. Come già accennato precedentemente abbiamo effettuato dei test considerando dataset con un numero di classi differente e ResNet50 ha restituito valori di accuracy e di loss migliori in tutte le casistiche. Abbiamo impostato la valutazione dando maggiore importanza al testing ed ai valori di loss in particolare, visto anche che la funzione di callback per il salvataggio dei pesi si basa nel valutare lo stato della rete in cui la perdita è minore.

Le Fig 3.5 e Fig 3.6 mostrano i risultati dell'esecuzione del modello considerando 20 classi. All'epoca 13 è possibile vedere che sono stati ottenuti i risultati migliori per quanto riguarda il testing: val_loss -> 1.4933 val_accuracy -> 0.5192.

```
Epoch 1/30
33/33 [==============================] - 220s 6s/step - loss: 3.3101 - accuracy: 0.1019 - val_loss: 2.7997 - val_accuracy: 0.0962
Epoch 2/30
33/33 [==============================] - 249s 7s/step - loss: 2.1837 - accuracy: 0.2990 - val_loss: 2.5261 - val_accuracy: 0.1885
Epoch 3/30
33/33 [==============================] - 210s 6s/step - loss: 1.6991 - accuracy: 0.4596 - val_loss: 2.1614 - val_accuracy: 0.2808
Epoch 4/30
33/33 [==============================] - 251s 7s/step - loss: 1.2762 - accuracy: 0.6212 - val_loss: 1.9581 - val_accuracy: 0.3192
Epoch 5/30
33/33 [==============================] - 259s 8s/step - loss: 1.0433 - accuracy: 0.6779 - val_loss: 2.0022 - val_accuracy: 0.3231
Epoch 6/30
33/33 [==============================] - 253s 8s/step - loss: 0.8007 - accuracy: 0.7769 - val_loss: 1.8029 - val_accuracy: 0.3731
Epoch 7/30
33/33 [==============================] - 221s 7s/step - loss: 0.6297 - accuracy: 0.8413 - val_loss: 1.7669 - val_accuracy: 0.3923
Epoch 8/30
33/33 [==============================] - 249s 7s/step - loss: 0.5171 - accuracy: 0.8683 - val_loss: 1.6080 - val_accuracy: 0.4269
Epoch 9/30
33/33 [==============================] - 244s 7s/step - loss: 0.3967 - accuracy: 0.9115 - val_loss: 1.5352 - val_accuracy: 0.4154
Epoch 10/30
33/33 [==============================] - 258s 8s/step - loss: 0.3173 - accuracy: 0.9375 - val_loss: 1.5852 - val_accuracy: 0.4615
Epoch 11/30
33/33 [==============================] - 209s 6s/step - loss: 0.2521 - accuracy: 0.9606 - val_loss: 1.6135 - val_accuracy: 0.5077
Epoch 12/30
33/33 [==============================] - 219s 6s/step - loss: 0.2096 - accuracy: 0.9644 - val_loss: 1.6049 - val_accuracy: 0.4769
Epoch 13/30
33/33 [==============================] - 250s 7s/step - loss: 0.1691 - accuracy: 0.9779 - val_loss: 1.4933 - val_accuracy: 0.5192
Epoch 14/30
33/33 [==============================] - 218s 6s/step - loss: 0.1573 - accuracy: 0.9740 - val_loss: 1.5611 - val_accuracy: 0.4808
Epoch 15/30
33/33 [==============================] - 248s 7s/step - loss: 0.1463 - accuracy: 0.9817 - val_loss: 1.7028 - val_accuracy: 0.4462
```

Fig 3.5 Risultati ResNet50.

```
Epoch 16/30
33/33 [==============================] - 263s 8s/step - loss: 0.1367 - accuracy: 0.9808 - val_loss: 1.5756 - val_accuracy: 0.4846
Epoch 17/30
33/33 [==============================] - 222s 7s/step - loss: 0.0968 - accuracy: 0.9875 - val_loss: 1.6436 - val_accuracy: 0.4731
Epoch 18/30
33/33 [==============================] - 268s 8s/step - loss: 0.0897 - accuracy: 0.9894 - val_loss: 1.6810 - val_accuracy: 0.4538
Epoch 19/30
33/33 [==============================] - 273s 8s/step - loss: 0.0931 - accuracy: 0.9846 - val_loss: 1.6377 - val_accuracy: 0.4808
Epoch 20/30
33/33 [==============================] - 269s 8s/step - loss: 0.0754 - accuracy: 0.9885 - val_loss: 1.6673 - val_accuracy: 0.5000
Epoch 21/30
33/33 [==============================] - 229s 7s/step - loss: 0.0593 - accuracy: 0.9942 - val_loss: 1.6106 - val_accuracy: 0.5077
Epoch 22/30
33/33 [==============================] - 256s 8s/step - loss: 0.0701 - accuracy: 0.9885 - val_loss: 1.7217 - val_accuracy: 0.4692
Epoch 23/30
33/33 [==============================] - 284s 8s/step - loss: 0.0577 - accuracy: 0.9952 - val_loss: 1.6974 - val_accuracy: 0.5000
Epoch 24/30
33/33 [==============================] - 266s 8s/step - loss: 0.0552 - accuracy: 0.9913 - val_loss: 1.7070 - val_accuracy: 0.4731
Epoch 25/30
33/33 [==============================] - 268s 8s/step - loss: 0.0650 - accuracy: 0.9894 - val_loss: 1.7126 - val_accuracy: 0.4846
Epoch 26/30
33/33 [==============================] - 254s 7s/step - loss: 0.0680 - accuracy: 0.9904 - val_loss: 1.7280 - val_accuracy: 0.4923
Epoch 27/30
33/33 [==============================] - 251s 7s/step - loss: 0.0684 - accuracy: 0.9875 - val_loss: 1.7226 - val_accuracy: 0.4923
Epoch 28/30
33/33 [==============================] - 214s 6s/step - loss: 0.0722 - accuracy: 0.9817 - val_loss: 1.8400 - val_accuracy: 0.4615
Epoch 29/30
33/33 [==============================] - 205s 6s/step - loss: 0.0821 - accuracy: 0.9856 - val_loss: 1.9948 - val_accuracy: 0.4808
Epoch 30/30
33/33 [==============================] - 208s 6s/step - loss: 0.0618 - accuracy: 0.9885 - val_loss: 1.8229 - val_accuracy: 0.4846
```

Fig 3.6 Risultati ResNet50.

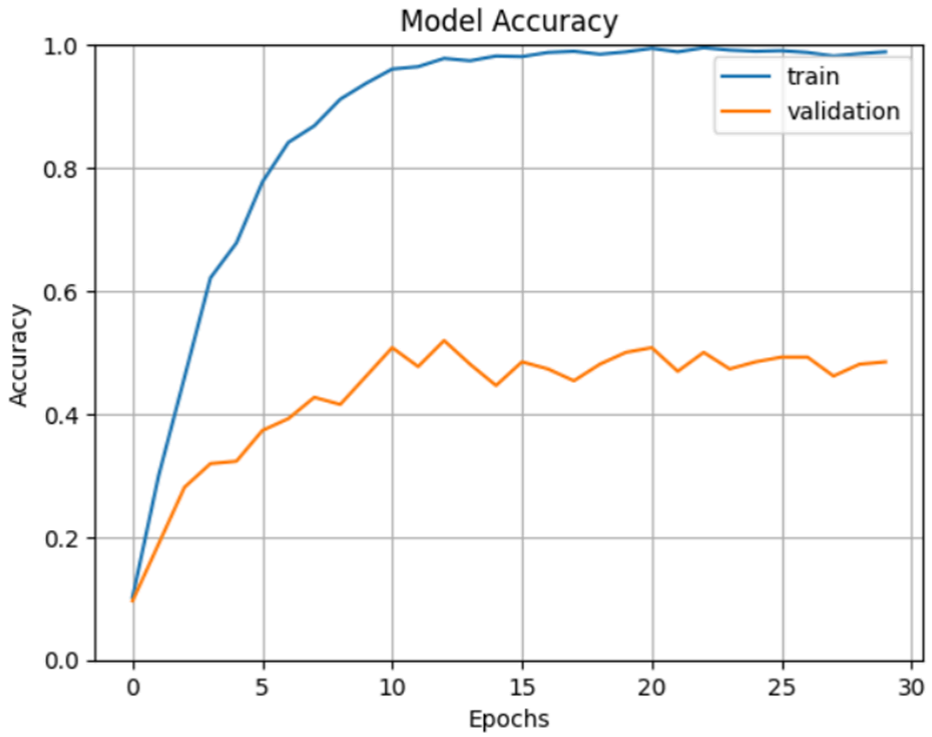

Fig 3.7 Grafico Accuracy ResNet50 20 con classi.

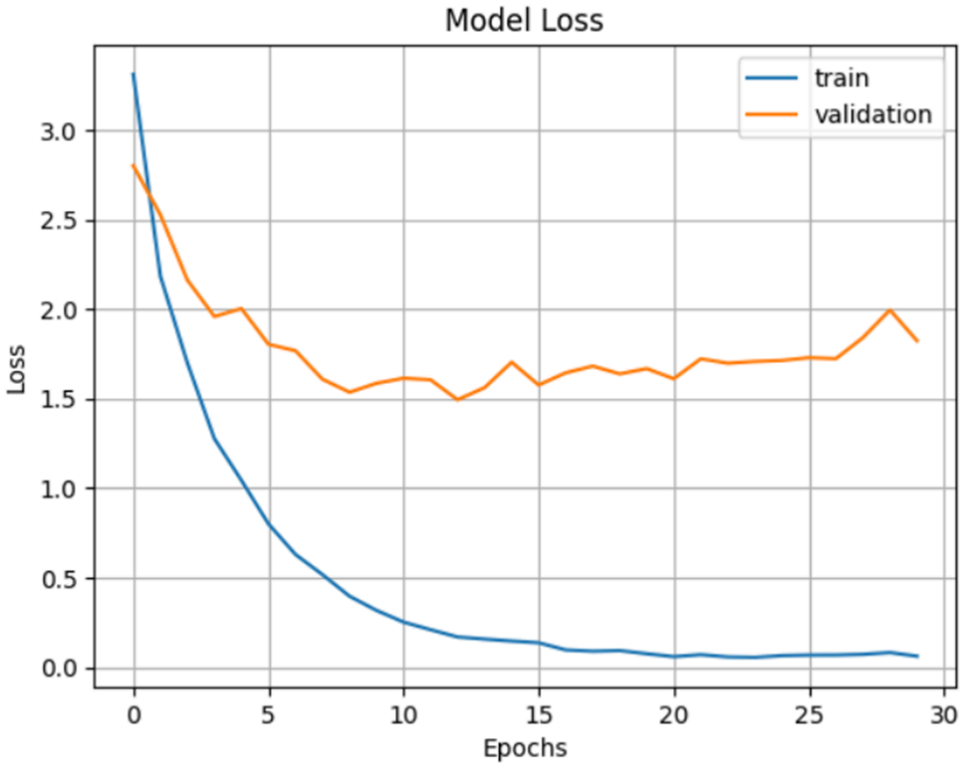

Fig 3.8 Grafico Loss ResNet50 20 con classi.

In generale è possibile osservare che dopo un tot di esecuzioni il modello tende a peggiorare sia per quel che riguarda il training che il testing.
Abbiamo inoltre provato ad effettuare un secondo addestramento della stessa rete con 13 epoche per vedere se saremmo riusciti ad ottenere ulteriori miglioramenti. Come è possibile vedere in Fig. 3.9 abbiamo migliorato leggermente la val_accuracy all'epoca 5, ottenendo 0.5423 come valore, peggiorando però il valore di perdita minimo.

```
Epoch 1/13
33/33 [==============================] - 252s 7s/step - loss: 0.0420 - accuracy: 0.9913 - val_loss: 1.6918 - val_accuracy: 0.5077
Epoch 2/13
33/33 [==============================] - 253s 7s/step - loss: 0.0385 - accuracy: 0.9952 - val_loss: 1.7359 - val_accuracy: 0.5000
Epoch 3/13
33/33 [==============================] - 258s 8s/step - loss: 0.0347 - accuracy: 0.9933 - val_loss: 1.6784 - val_accuracy: 0.5154
Epoch 4/13
33/33 [==============================] - 251s 7s/step - loss: 0.0284 - accuracy: 0.9933 - val_loss: 1.7823 - val_accuracy: 0.5154
Epoch 5/13
33/33 [==============================] - 258s 8s/step - loss: 0.0170 - accuracy: 0.9981 - val_loss: 1.6270 - val_accuracy: 0.5423
Epoch 6/13
33/33 [==============================] - 208s 6s/step - loss: 0.0261 - accuracy: 0.9952 - val_loss: 1.8216 - val_accuracy: 0.5038
Epoch 7/13
33/33 [==============================] - 206s 6s/step - loss: 0.0299 - accuracy: 0.9952 - val_loss: 1.7266 - val_accuracy: 0.5192
Epoch 8/13
33/33 [==============================] - 255s 8s/step - loss: 0.0307 - accuracy: 0.9933 - val_loss: 1.7479 - val_accuracy: 0.5077
Epoch 9/13
33/33 [==============================] - 251s 7s/step - loss: 0.0589 - accuracy: 0.9923 - val_loss: 1.8982 - val_accuracy: 0.4731
Epoch 10/13
33/33 [==============================] - 252s 7s/step - loss: 0.0377 - accuracy: 0.9962 - val_loss: 1.8975 - val_accuracy: 0.4923
Epoch 11/13
33/33 [==============================] - 250s 7s/step - loss: 0.0490 - accuracy: 0.9913 - val_loss: 1.9122 - val_accuracy: 0.4885
Epoch 12/13
33/33 [==============================] - 209s 6s/step - loss: 0.0424 - accuracy: 0.9894 - val_loss: 1.9704 - val_accuracy: 0.4731
Epoch 13/13
33/33 [==============================] - 255s 8s/step - loss: 0.0641 - accuracy: 0.9885 - val_loss: 2.1076 - val_accuracy: 0.4538
```

Fig 3.9 Seconda esecuzione ResNet50.

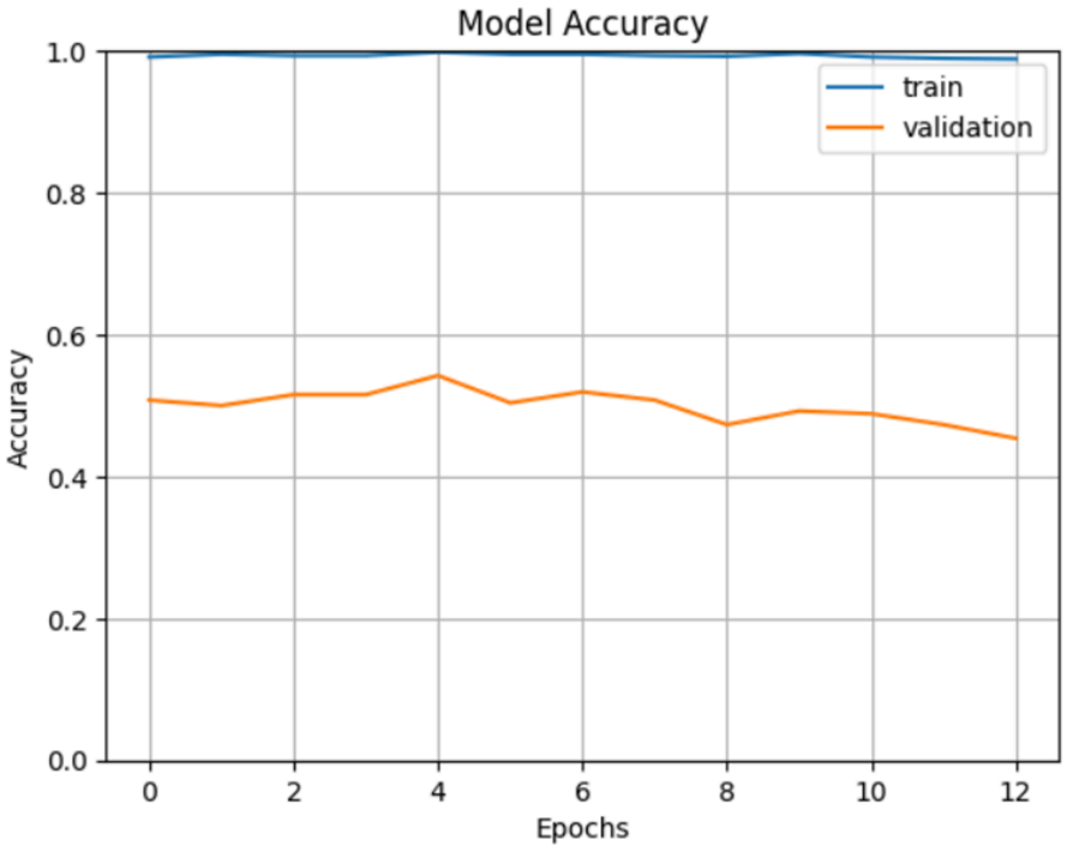

Fig. 4.0 Grafico Accuracy del secondo training di ResNet50 con 20 classi.

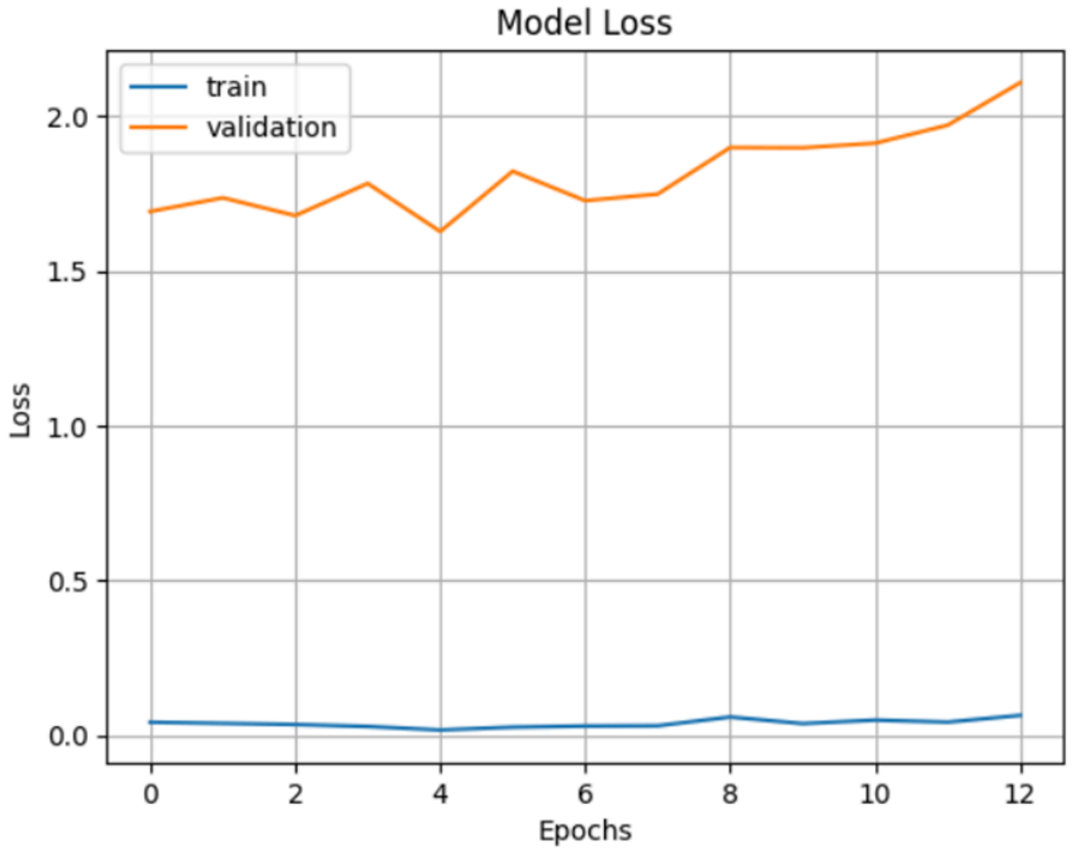

Fig. 4.1 Grafico Loss del secondo training di ResNet50 con 20 classi.

Per quanto riguarda l'esecuzione del modello con 4 classi i risultati migliori nel testing sono stati all'epoca 11: val_loss -> 0.2525 val_accuracy -> 0.9038.

```
Epoch 1/30
WARNING:tensorflow:From c:\Users\Utente PC\AppData\Local\Programs\Python\Python310\lib\site-packages\keras\src\utils\tf_utils.py:4

WARNING:tensorflow:From c:\Users\Utente PC\AppData\Local\Programs\Python\Python310\lib\site-packages\keras\src\engine\base_layer_u

33/33 [==============================] - 53s 1s/step - loss: 1.6910 - accuracy: 0.4808 - val_loss: 0.8105 - val_accuracy: 0.6385
Epoch 2/30
33/33 [==============================] - 50s 1s/step - loss: 0.6140 - accuracy: 0.7462 - val_loss: 0.5803 - val_accuracy: 0.7885
Epoch 3/30
33/33 [==============================] - 46s 1s/step - loss: 0.3777 - accuracy: 0.8769 - val_loss: 0.5001 - val_accuracy: 0.8000
Epoch 4/30
33/33 [==============================] - 50s 2s/step - loss: 0.2800 - accuracy: 0.9144 - val_loss: 0.4849 - val_accuracy: 0.8115
Epoch 5/30
33/33 [==============================] - 48s 1s/step - loss: 0.2263 - accuracy: 0.9240 - val_loss: 0.4592 - val_accuracy: 0.8192
Epoch 6/30
33/33 [==============================] - 46s 1s/step - loss: 0.1649 - accuracy: 0.9510 - val_loss: 0.3415 - val_accuracy: 0.8577
Epoch 7/30
33/33 [==============================] - 44s 1s/step - loss: 0.0948 - accuracy: 0.9808 - val_loss: 0.3251 - val_accuracy: 0.8923
Epoch 8/30
33/33 [==============================] - 45s 1s/step - loss: 0.0803 - accuracy: 0.9856 - val_loss: 0.3094 - val_accuracy: 0.8923
Epoch 9/30
33/33 [==============================] - 45s 1s/step - loss: 0.0567 - accuracy: 0.9933 - val_loss: 0.3011 - val_accuracy: 0.8731
Epoch 10/30
33/33 [==============================] - 45s 1s/step - loss: 0.0498 - accuracy: 0.9933 - val_loss: 0.2837 - val_accuracy: 0.8962
```

Fig. 4.2 Risultati ResNet50 con 4 classi.

```
Epoch 11/30
33/33 [==============================] - 45s 1s/step - loss: 0.0366 - accuracy: 0.9962 - val_loss: 0.2525 - val_accuracy: 0.9038
Epoch 12/30
33/33 [==============================] - 45s 1s/step - loss: 0.0244 - accuracy: 0.9990 - val_loss: 0.3528 - val_accuracy: 0.8692
Epoch 13/30
33/33 [==============================] - 45s 1s/step - loss: 0.0350 - accuracy: 0.9933 - val_loss: 0.4821 - val_accuracy: 0.8538
Epoch 14/30
33/33 [==============================] - 48s 1s/step - loss: 0.0432 - accuracy: 0.9894 - val_loss: 0.3171 - val_accuracy: 0.8923
Epoch 15/30
33/33 [==============================] - 47s 1s/step - loss: 0.0189 - accuracy: 0.9990 - val_loss: 0.3632 - val_accuracy: 0.8654
Epoch 16/30
33/33 [==============================] - 45s 1s/step - loss: 0.0128 - accuracy: 1.0000 - val_loss: 0.2716 - val_accuracy: 0.8962
Epoch 17/30
33/33 [==============================] - 45s 1s/step - loss: 0.0096 - accuracy: 1.0000 - val_loss: 0.2896 - val_accuracy: 0.8962
Epoch 18/30
33/33 [==============================] - 45s 1s/step - loss: 0.0077 - accuracy: 1.0000 - val_loss: 0.2701 - val_accuracy: 0.8885
Epoch 19/30
33/33 [==============================] - 45s 1s/step - loss: 0.0068 - accuracy: 1.0000 - val_loss: 0.3122 - val_accuracy: 0.8769
Epoch 20/30
33/33 [==============================] - 45s 1s/step - loss: 0.0060 - accuracy: 1.0000 - val_loss: 0.3143 - val_accuracy: 0.8846
Epoch 21/30
33/33 [==============================] - 45s 1s/step - loss: 0.0055 - accuracy: 1.0000 - val_loss: 0.2865 - val_accuracy: 0.8923
Epoch 22/30
33/33 [==============================] - 45s 1s/step - loss: 0.0047 - accuracy: 1.0000 - val_loss: 0.3091 - val_accuracy: 0.8885
```

Fig. 4.3 Risultati ResNet50 con 4 classi.

```
Epoch 23/30
33/33 [==============================] - 45s 1s/step - loss: 0.0049 - accuracy: 1.0000 - val_loss: 0.3224 - val_accuracy: 0.8885
Epoch 24/30
33/33 [==============================] - 44s 1s/step - loss: 0.0063 - accuracy: 0.9981 - val_loss: 0.2847 - val_accuracy: 0.8885
Epoch 25/30
33/33 [==============================] - 44s 1s/step - loss: 0.0045 - accuracy: 1.0000 - val_loss: 0.3390 - val_accuracy: 0.8846
Epoch 26/30
33/33 [==============================] - 44s 1s/step - loss: 0.0058 - accuracy: 0.9990 - val_loss: 0.3228 - val_accuracy: 0.8962
Epoch 27/30
33/33 [==============================] - 44s 1s/step - loss: 0.0116 - accuracy: 0.9990 - val_loss: 0.2683 - val_accuracy: 0.8846
Epoch 28/30
33/33 [==============================] - 44s 1s/step - loss: 0.0049 - accuracy: 0.9990 - val_loss: 0.3290 - val_accuracy: 0.8923
Epoch 29/30
33/33 [==============================] - 44s 1s/step - loss: 0.0060 - accuracy: 0.9990 - val_loss: 0.2589 - val_accuracy: 0.8885
Epoch 30/30
33/33 [==============================] - 44s 1s/step - loss: 0.0025 - accuracy: 1.0000 - val_loss: 0.3149 - val_accuracy: 0.8885
```

Fig. 4.4 Risultati ResNet50 con 4 classi.

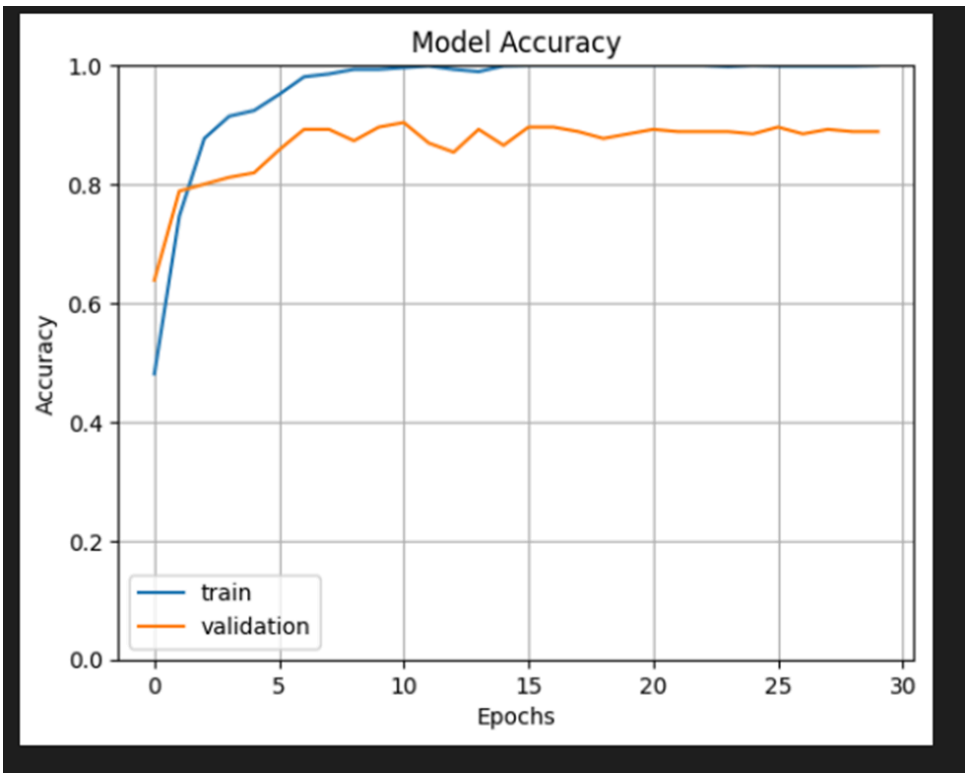

Fig. 4.5 Grafico Accuracy ResNet50 con 4 classi.

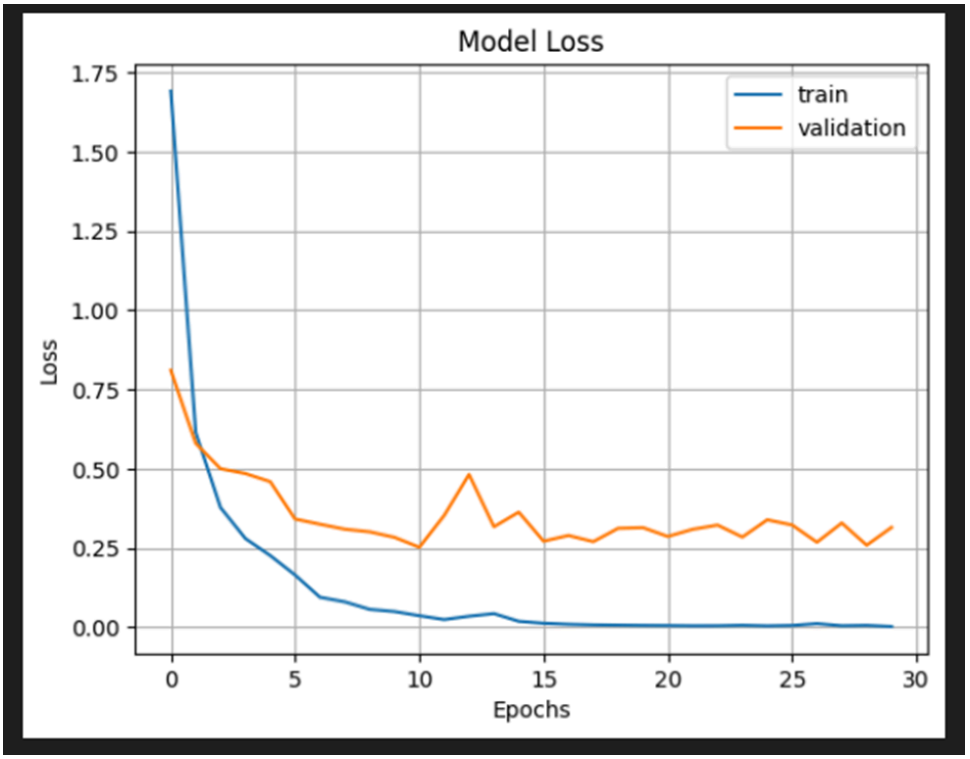

Fig. 4.6 Grafico Loss ResNet50 con 4 classi.

Invece, VGG19 con un dataset da 20 classi restituisce nelle prime iterazioni dei valori di accuracy migliori rispetto a ResNet50 ma dei valori di loss peggiori. In generale il modello si dimostra inferiore in quasi tutte le epoche successive alle prime (vedere Fig. 4.7, Fig. 4.8, Fig. 4.9). Il risultato migliore in fase di testing è stato: val_loss -> 1.9584 val_accuracy-> 0.4154.

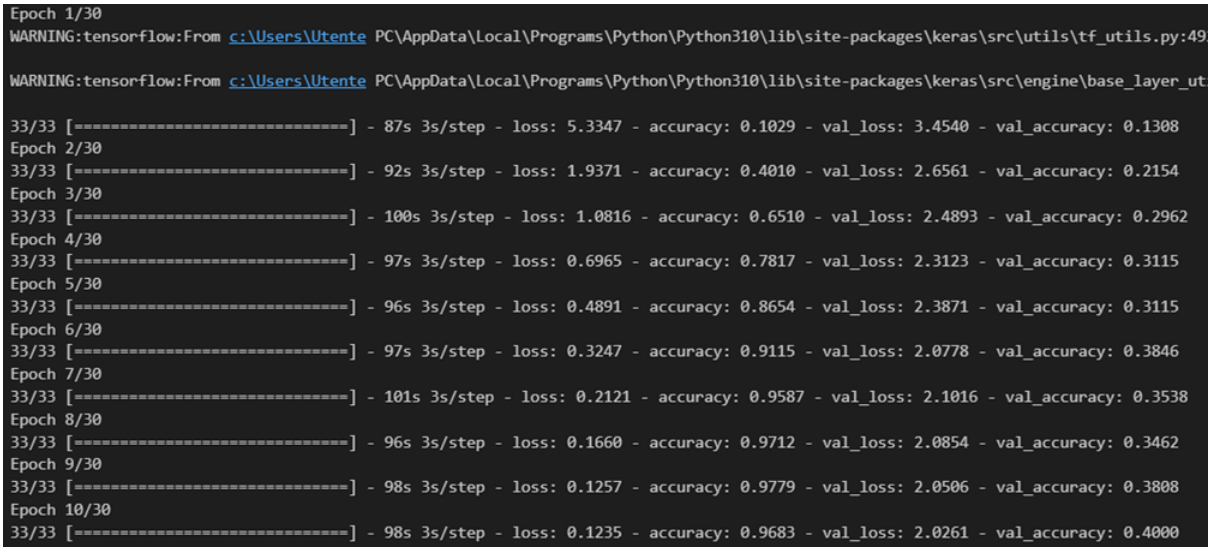

Fig. 4.7 Risultati VGG19 con 20 classi.

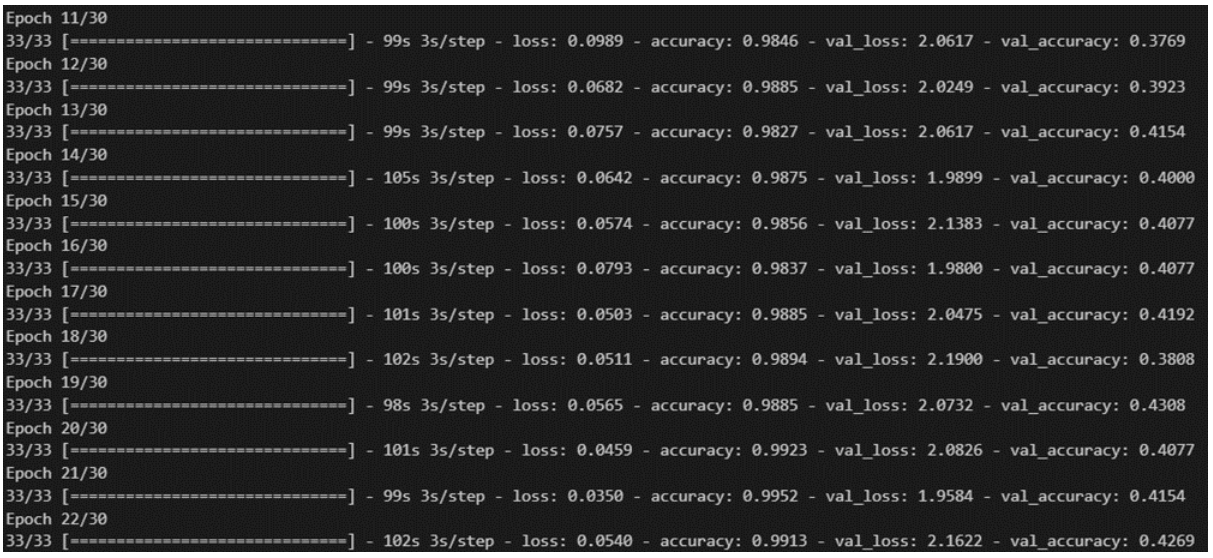

Fig. 4.8 Risultati VGG19 con 20 classi.

```
Epoch 23/30
33/33 [==============================] - 99s 3s/step - loss: 0.0550 - accuracy: 0.9856 - val_loss: 2.2128 - val_accuracy: 0.4038
Epoch 24/30
33/33 [==============================] - 102s 3s/step - loss: 0.0613 - accuracy: 0.9875 - val_loss: 2.1265 - val_accuracy: 0.4192
Epoch 25/30
33/33 [==============================] - 99s 3s/step - loss: 0.0636 - accuracy: 0.9875 - val_loss: 2.1724 - val_accuracy: 0.4038
Epoch 26/30
33/33 [==============================] - 99s 3s/step - loss: 0.0845 - accuracy: 0.9798 - val_loss: 2.2857 - val_accuracy: 0.4000
Epoch 27/30
33/33 [==============================] - 99s 3s/step - loss: 0.0628 - accuracy: 0.9827 - val_loss: 2.1290 - val_accuracy: 0.4154
Epoch 28/30
33/33 [==============================] - 99s 3s/step - loss: 0.0568 - accuracy: 0.9894 - val_loss: 2.1559 - val_accuracy: 0.4192
Epoch 29/30
33/33 [==============================] - 99s 3s/step - loss: 0.1285 - accuracy: 0.9731 - val_loss: 2.5152 - val_accuracy: 0.3769
Epoch 30/30
33/33 [==============================] - 99s 3s/step - loss: 0.0932 - accuracy: 0.9779 - val_loss: 2.4523 - val_accuracy: 0.3962
```

Fig. 4.9 Risultati VGG19 con 20 classi.

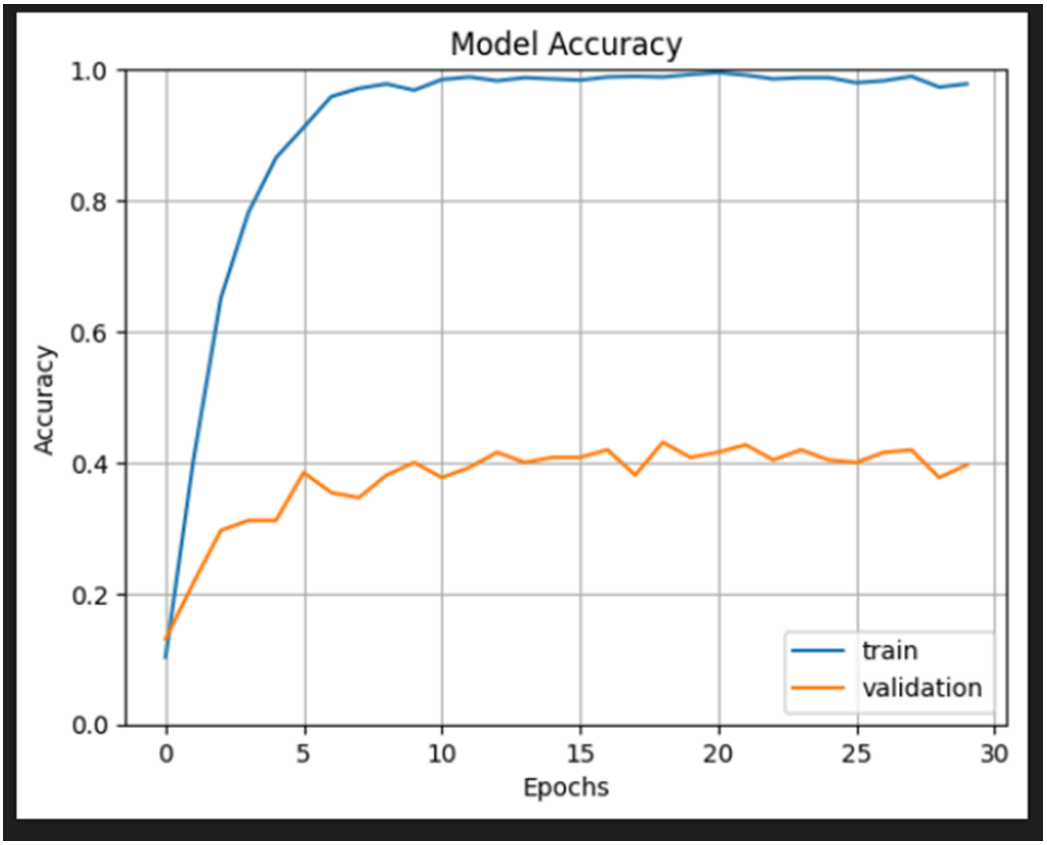

Fig. 5.0 Grafico Accuracy VGG19 con 20 classi.

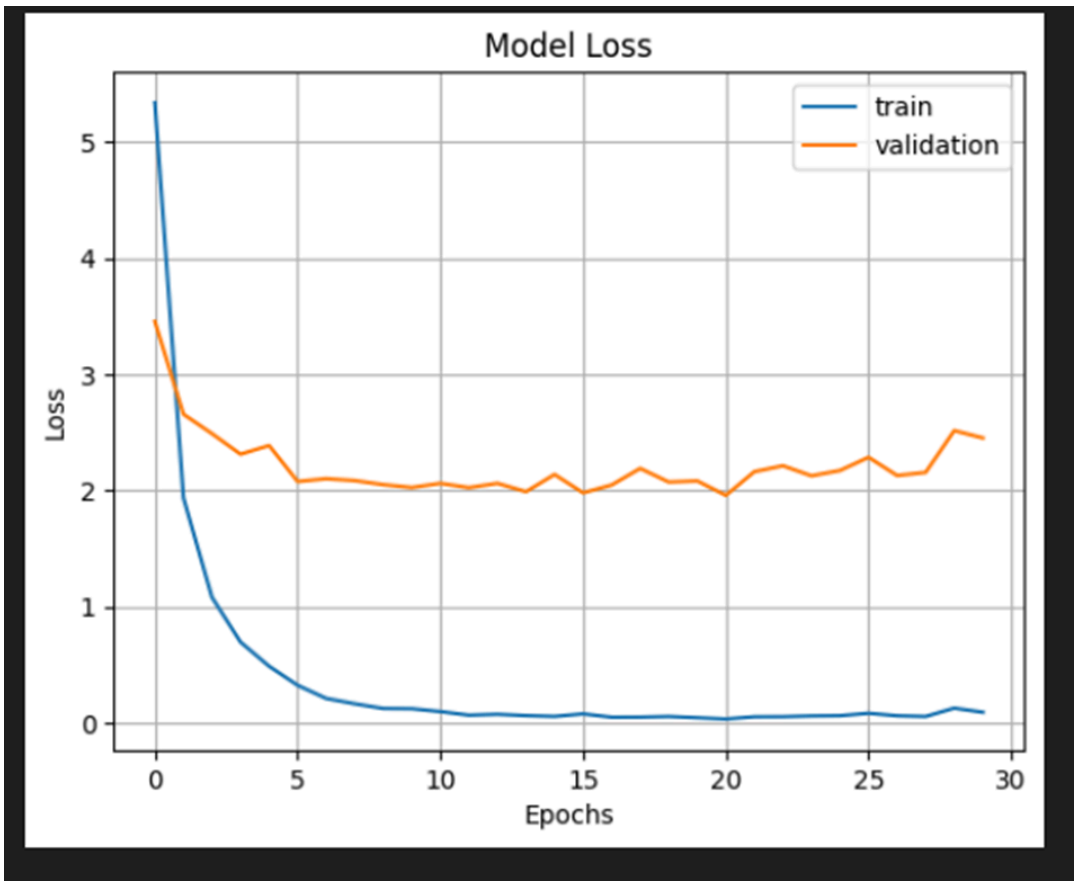

Fig. 5.1 Grafico Loss VGG19 con 20 classi.

Anche considerando 4 classi il confronto tra VGG19 e ResNet50 restituisce risultati simili alla casistica precedente nella quale sono state considerate 20 classi. I migliori risultati di VGG19 sono: val_loss -> 0.4851 val_accuracy -> 0.8500 (vedi Fig. 5.2, Fig. 5.3, Fig. 5.4).

```
Epoch 1/30
WARNING:tensorflow:From c:\Users\Utente PC\AppData\Local\Programs\Python\Python310\lib\site-packages\keras\src\utils\tf_utils.py:4

WARNING:tensorflow:From c:\Users\Utente PC\AppData\Local\Programs\Python\Python310\lib\site-packages\keras\src\engine\base_layer_u

33/33 [==============================] - 85s 3s/step - loss: 2.6405 - accuracy: 0.4933 - val_loss: 1.1739 - val_accuracy: 0.6500
Epoch 2/30
33/33 [==============================] - 96s 3s/step - loss: 0.5226 - accuracy: 0.8173 - val_loss: 0.6965 - val_accuracy: 0.7538
Epoch 3/30
33/33 [==============================] - 100s 3s/step - loss: 0.1985 - accuracy: 0.9212 - val_loss: 0.5685 - val_accuracy: 0.8115
Epoch 4/30
33/33 [==============================] - 98s 3s/step - loss: 0.1565 - accuracy: 0.9442 - val_loss: 0.5446 - val_accuracy: 0.8192
Epoch 5/30
33/33 [==============================] - 99s 3s/step - loss: 0.0965 - accuracy: 0.9692 - val_loss: 0.5523 - val_accuracy: 0.8038
Epoch 6/30
33/33 [==============================] - 103s 3s/step - loss: 0.0668 - accuracy: 0.9865 - val_loss: 0.5466 - val_accuracy: 0.8115
Epoch 7/30
33/33 [==============================] - 98s 3s/step - loss: 0.0466 - accuracy: 0.9894 - val_loss: 0.6037 - val_accuracy: 0.7962
Epoch 8/30
33/33 [==============================] - 98s 3s/step - loss: 0.0312 - accuracy: 0.9923 - val_loss: 0.4924 - val_accuracy: 0.8385
Epoch 9/30
33/33 [==============================] - 98s 3s/step - loss: 0.0232 - accuracy: 0.9962 - val_loss: 0.4971 - val_accuracy: 0.8231
Epoch 10/30
33/33 [==============================] - 101s 3s/step - loss: 0.0234 - accuracy: 0.9952 - val_loss: 0.5446 - val_accuracy: 0.8154
Epoch 11/30
33/33 [==============================] - 98s 3s/step - loss: 0.0153 - accuracy: 0.9990 - val_loss: 0.5222 - val_accuracy: 0.8308
Epoch 12/30
33/33 [==============================] - 98s 3s/step - loss: 0.0115 - accuracy: 0.9981 - val_loss: 0.5122 - val_accuracy: 0.8346
```

Fig. 5.2 Risultati VGG19 con 4 classi.

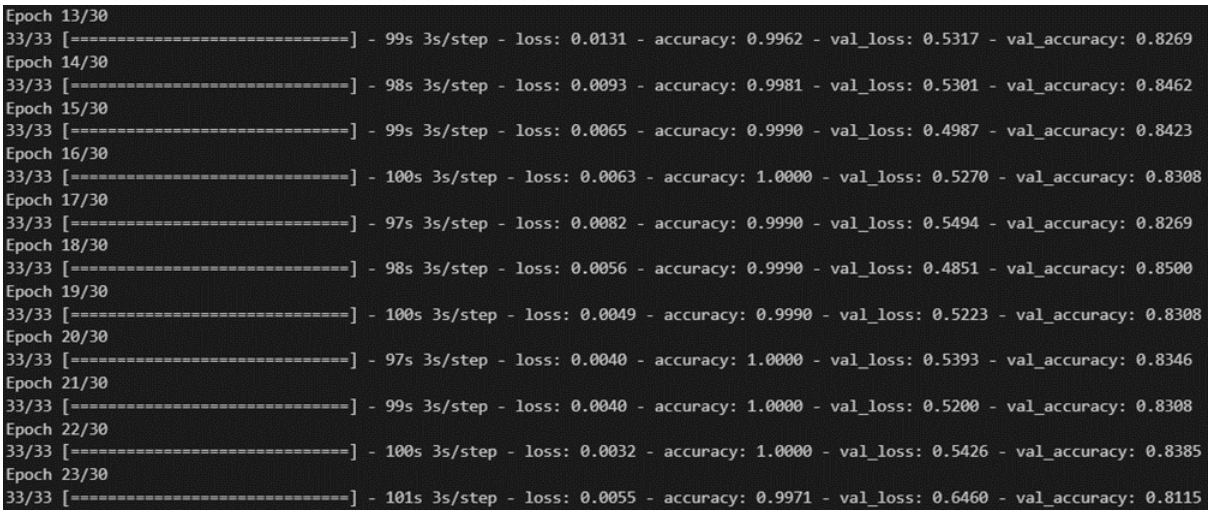

Fig. 5.3 Risultati VGG19 con 4 classi.

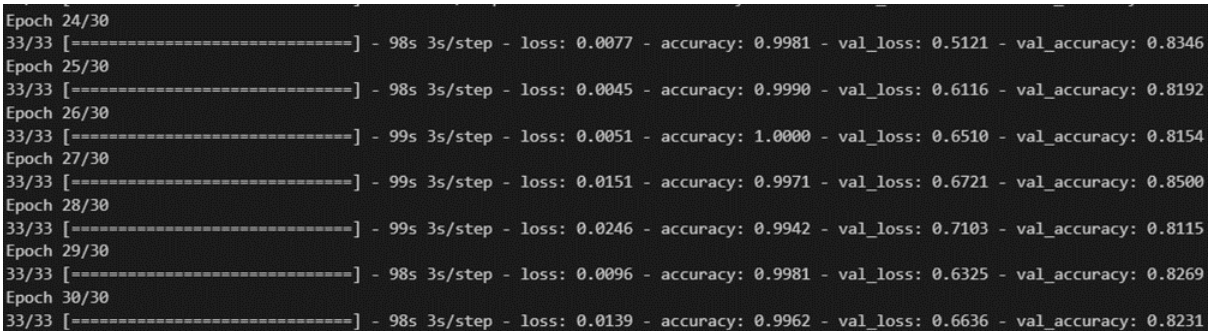

Fig. 5.4 Risultati VGG19 con 4 classi.

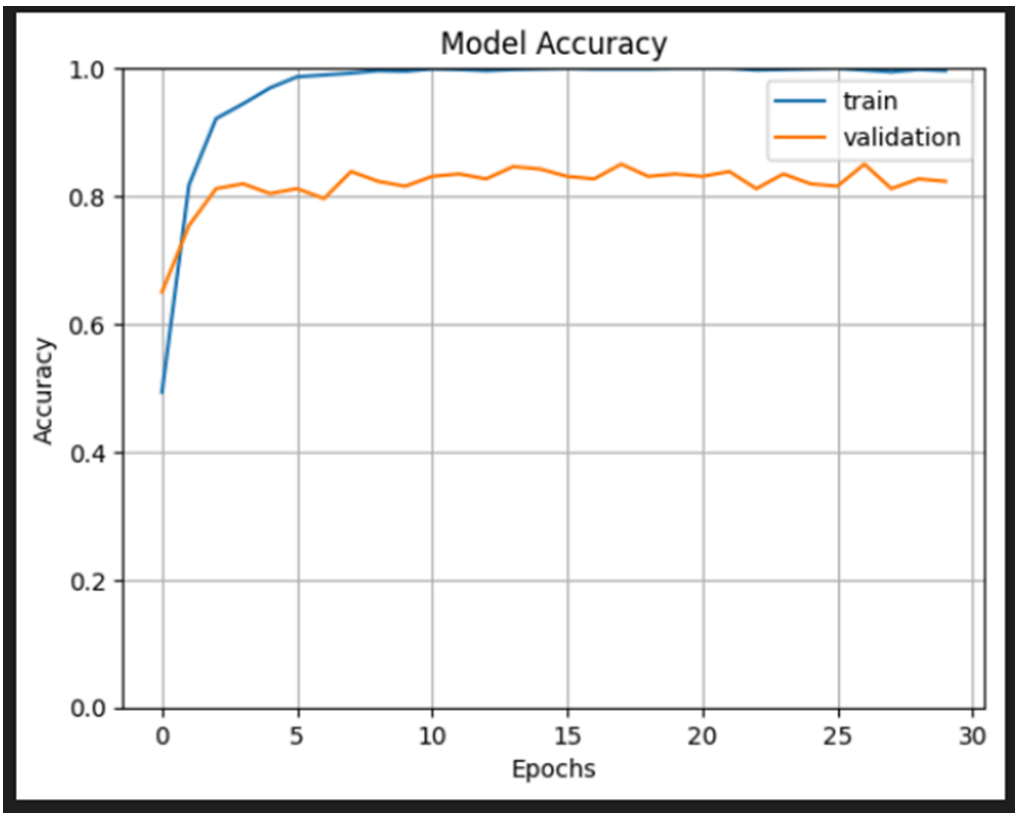

Fig. 5.5 Grafico Accuracy VGG19 con 4 classi.

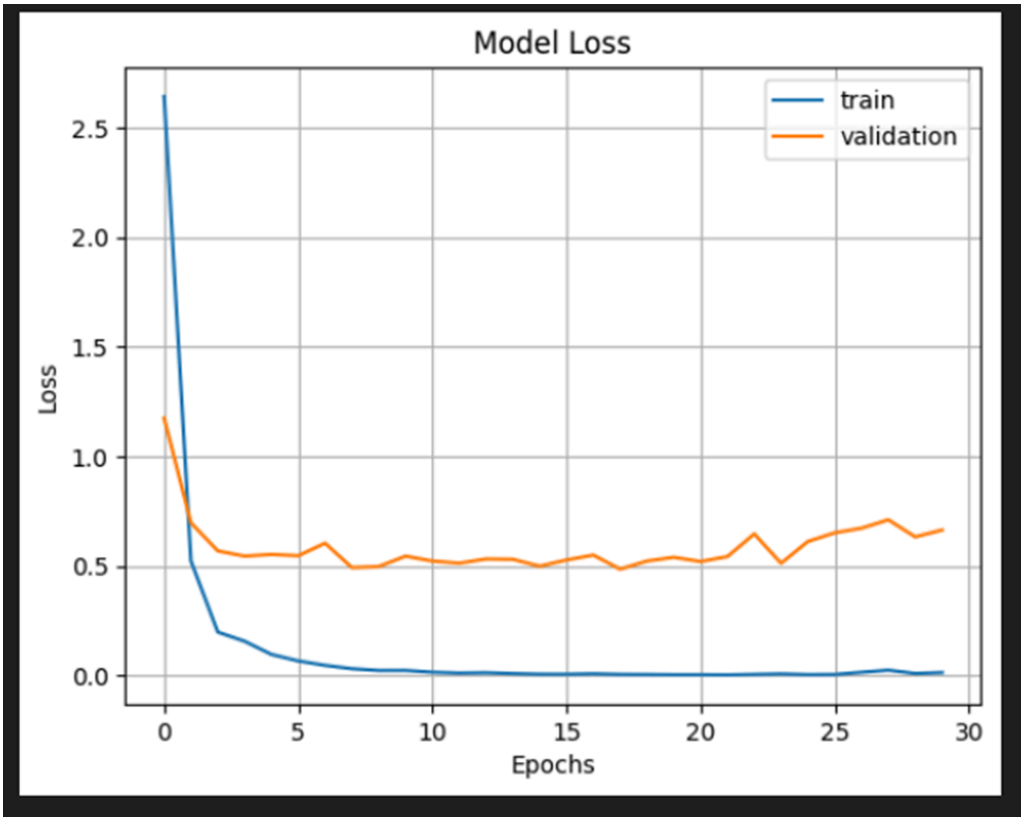

Fig. 5.6 Grafico Loss VGG19 con 4 classi.

Come passo ulteriore abbiamo implementato una parte di codice per poter analizzare il numero di errori di predizione del modello divisi per classe, al fine di comprendere dove si verificano le maggiori difficoltà. Qui di seguito sono riportati i grafici relativi agli errori dei due modelli considerando un numero di classi differente.

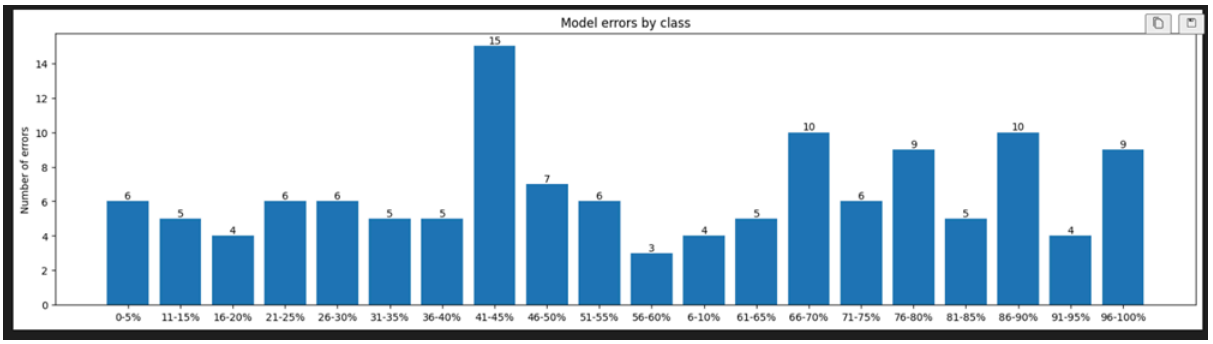

Fig. 5.7 ResNet50 (20 classi) con accuracy 50% (130 errori su 260 immagini testate).

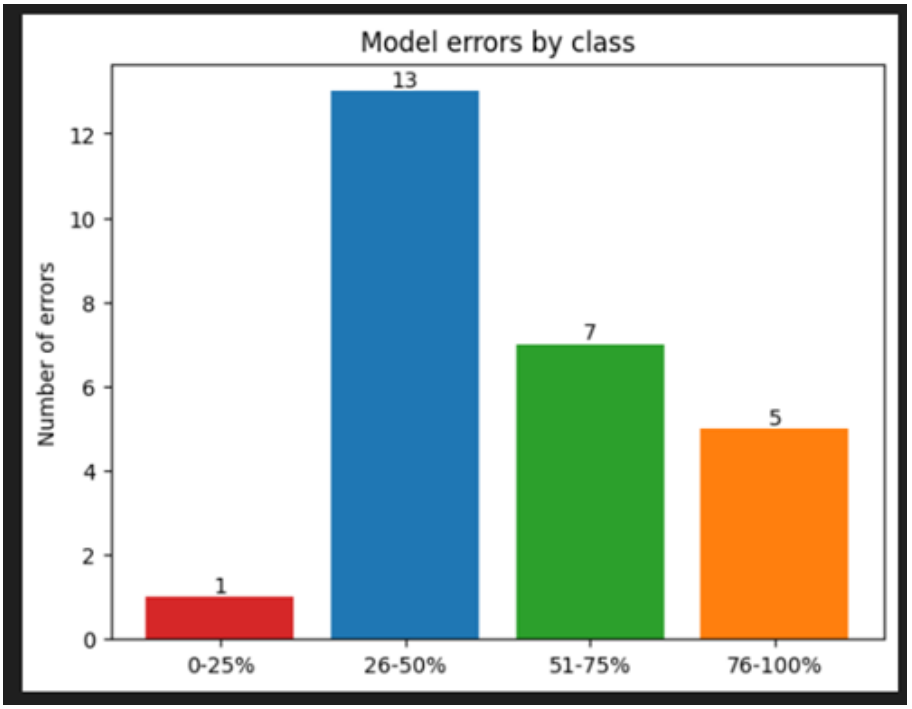

Fig. 5.8 ResNet50 (5 classi) con accuracy 90% (26 errori su 260 immagini testate).

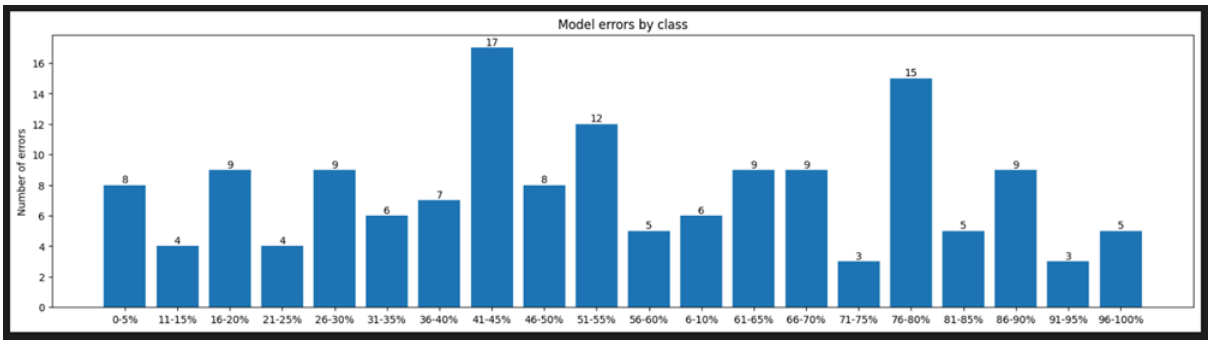

Fig. 5.9 VGG19 (20 classi) con accuracy 42% (153 errori su 260 immagini testate).

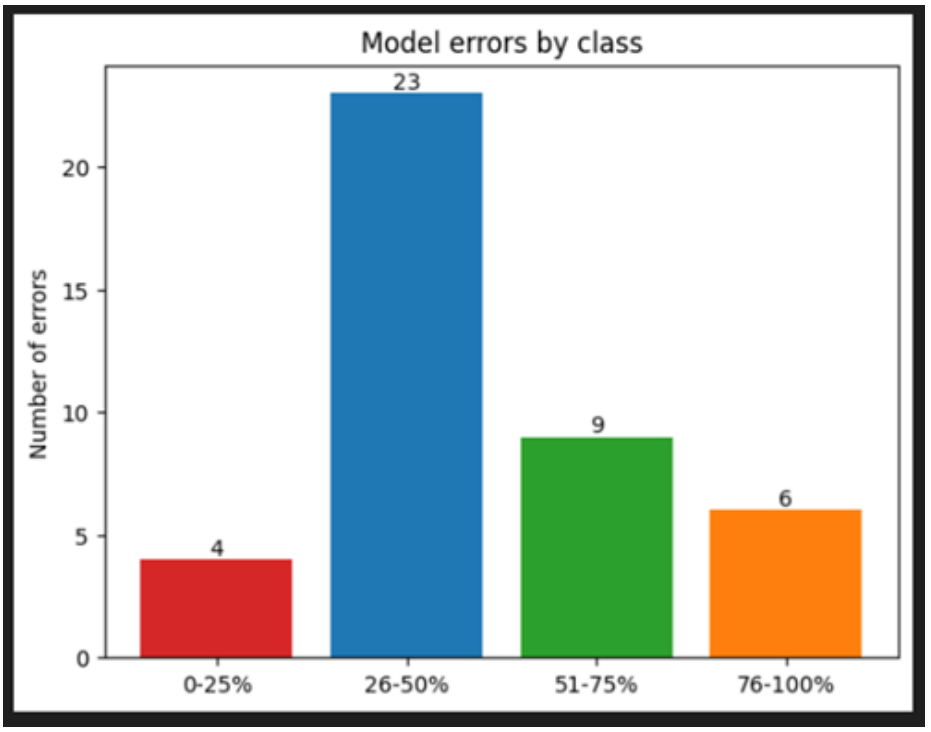

Fig. 6.0 VGG19 (5 classi) con accuracy 83% (42 errori su 260 immagini testate)

Infine, è stata utilizzata la libreria shap per mostrare cosa guarda il predittore per comprendere la classe di appartenenza delle immagini che valuta.

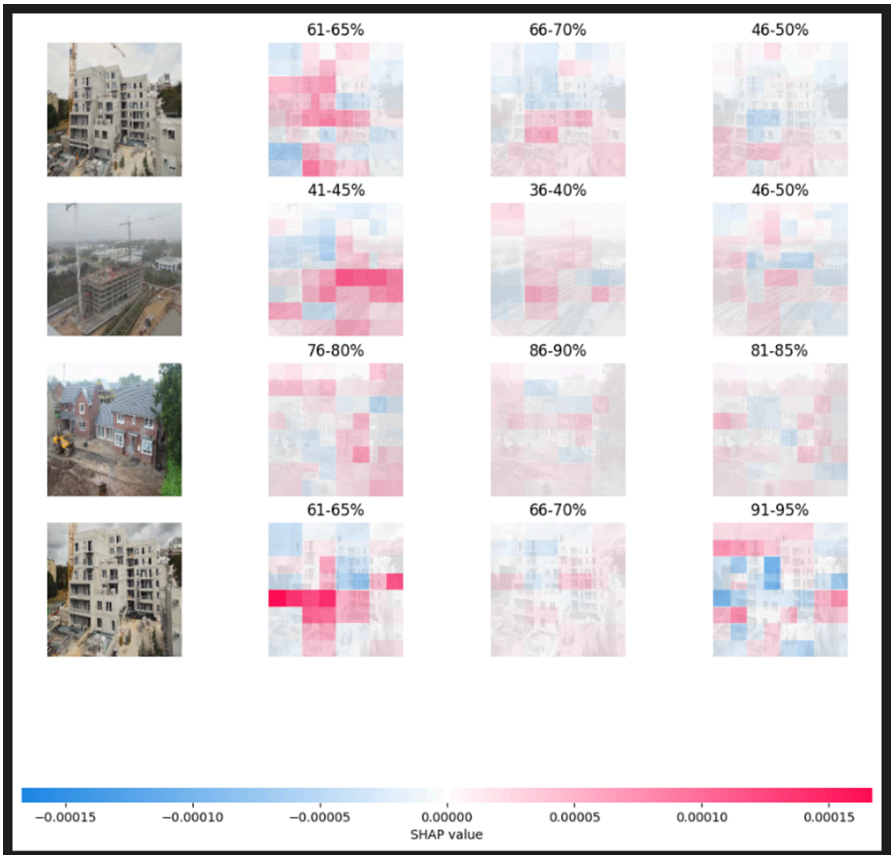

Fig. 6.1 Utilizzo di shap per l'image caption.

Studio di comparazione con letteratura:

I modelli utilizzati, come VGG19 e ResNet50, sono comunemente adoperati per la classificazione di immagini. Questo ci consente di confrontare i risultati ottenuti con altre ricerche. Nei nostri esperimenti, abbiamo notato un aumento dell'accuratezza all'aumentare delle epoche. Ad ogni epoca, il modello viene riaddestrato sull'intero dataset, basandosi su quanto appreso nelle epoche precedenti. È interessante osservare che oltre un certo punto si verifica un deterioramento delle prestazioni del modello, e il picco di accuratezza è generalmente raggiunto intorno alla ventesima epoca. Tale tendenza si riscontra anche nei test di confronto, sebbene modelli in altre ricerche possano ottenere risultati migliori, il che potrebbe essere attribuito all'uso di dataset più ampi con classi più nettamente distinte.

Altri modelli, come VGG16, mostrano prestazioni simili, seppur leggermente inferiori, quando sottoposti allo stesso livello di addestramento. Nei nostri test, abbiamo osservato che ResNet è leggermente superiore a VGG19.

Tra le tecniche utilizzate per migliorare l'assimilazione delle immagini dalla rete è stato applicato sia da noi che nel paper di confronto un ridimensionamento delle immagini per:

- **Uniformità delle dimensioni**: le reti neurali richiedono che tutte le immagini in input abbiano le stesse dimensioni. Ridimensionando le immagini, si assicura che tutte abbiano la stessa risoluzione, semplificando così il processo di elaborazione e consentendo alle reti neurali di lavorare in modo coerente.

- **Riduzione del carico computazionale**: immagini di grandi dimensioni richiedono più risorse computazionali per essere elaborate. Ridimensionarle a dimensioni più piccole può ridurre significativamente il carico computazionale, consentendo una maggiore efficienza nel processo di addestramento e inferenza della rete neurale.

- **Prevenzione dell'overfitting**: ridurre la dimensione delle immagini può aiutare a prevenire l'overfitting, specialmente in presenza di dataset limitati. L'overfitting si verifica quando il modello impara a memorizzare le caratteristiche specifiche del set di addestramento anziché generalizzare per riconoscere pattern più ampi. Riducendo la dimensione delle immagini, si riduce la complessità del modello e si limita la possibilità di overfitting.

- **Velocità di addestramento e inferenza**: le immagini ridimensionate richiedono meno tempo per essere elaborate durante il processo di addestramento e inferenza, consentendo una maggiore efficienza complessiva del sistema.

epoca	VGG19 (training accuracy)	RESNET (training accuracy)
1	0.1308	0.0962
3	0.2962	0.2808
4	0.3115	0.3192
5	0.3115	0.3231
10	0.4000	0.4615
15	0.4077	0.4462
20	0.4077	0.5000
30	0.3962	0.4846

Fig 6.2 Risultati ottenuti dal nostro modello.

epoch	VGG 16 (Training data Accuracy)	VGG 16 (Test data Accuracy)	VGG19 (Training data Accuracy)	VGG 19 (Test data Accuracy)	ResNet 50 (Training data Accuracy)	ResNet 50 (Test data Accuracy)
1	0.8516	0.9367	0.843	0.924	0.9176	0.9656
3	0.9552	0.9545	0.935	0.9489	0.978	0.9716
4	0.9621	0.9478	0.9574	0.9556	0.9762	0.9472
5	0.9664	0.9578	0.9707	0.9672	0.9814	0.9633
10	0.9852	0.9589	0.9859	0.965	0.9971	0.9722
15	0.9912	0.9639	0.9931	0.9744	0.9938	0.9683
20	**0.9919**	**0.9667**	**0.9936**	**0.9707**	**0.9976**	**0.9733**
100	1	0.9633	1	0.9689	1	0.9533

Fig 6.3 Risultati ottenuti dal modello nel paper di confronto [1].

4.a. Discussione dei risultati

Per quanto riguarda le aspettative di performance dei modelli impiegati, prevedevamo una maggiore accuratezza e precisione da ResNet50 rispetto a VGG19, basandoci sulle percentuali riportate in letteratura direttamente da Keras. E così è stato: siamo riusciti a confermare un risultato superiore con il modello ResNet50.

Tuttavia, le nostre previsioni consideravano la possibilità di un peggioramento o delle difficoltà nel raggiungere elevati livelli di precisione con un numero elevato di classi. Questa considerazione era legata al fatto che molte immagini sarebbero potute essere molto simili tra di loro, causando difficoltà nel differenziarle. Tale tendenza è stata successivamente confermata durante i test dei modelli con un numero notevolmente inferiore di classi, dove abbiamo ottenuto una precisione nettamente più elevata. Ciò ha confermato la nostra ipotesi che modelli con un numero inferiore di classi tendono a raggiungere precisioni più elevate rispetto a quelli con un numero maggiore di classi, evidenziando la possibilità di miglioramenti nella precisione se il dataset avesse incluso frame con differenze più marcate tra le classi.

4.b. Validità del metodo

Si osserva una coerenza dei risultati con gli obiettivi prefissati, mirati a sviluppare un efficace riconoscitore per la predizione della percentuale di un edificio in costruzione, e tali risultati confermano questa intenzione. Durante la fase di testing, il modello ha conseguito un'accuratezza del 54%, registrata dopo una nuova esecuzione.

È evidente una difficoltà iniziale nel raggiungere performance soddisfacenti con un numero limitato di iterazioni. Nei primi passaggi, la precisione è notevolmente bassa, richiedendo circa 10-15 passi per ottenere risultati considerevoli e utilizzabili. Questo suggerisce che il modello richiede un periodo di apprendimento più prolungato per acquisire efficacemente le caratteristiche del dataset.

In conclusione, il metodo impiegato ha permesso una valutazione accurata della scelta dei modelli e ha facilitato un'analisi delle caratteristiche. Tuttavia, per ottenere dati più efficienti, sarebbe necessario disporre di frame più distinti tra loro, in modo che le classi siano più facilmente distinguibili. Questa considerazione risulta particolarmente rilevante quando si lavora con un numero significativo di classi, poiché i video iniziali potrebbero presentare caratteristiche molto simili, generando difficoltà nel processo di distinzione. Possibili approcci per migliorare l'efficienza potrebbero includere l'implementazione di tecniche di aumento dei dati o la selezione di frame più rappresentativi per ciascuna classe.

4.c. Limitazioni e maturità

Il modello, durante la predizione della percentuale, potrebbe manifestare inefficienze particolarmente nelle fasi finali della costruzione, e ciò è attribuibile alla presenza di numerosi frame simili. Questo fenomeno si configura come un limite significativo durante la fase di predizione, poiché nelle fasi conclusive ci si attende una maggiore variabilità interna degli edifici rispetto agli elementi esterni.

Un aspetto rilevante da considerare riguarda la presenza di un bias principale, identificato nella natura delle immagini utilizzate. Le immagini estratte dai video possono risultare poco "pulite" in quanto composte da vari elementi di contesto e strumentazioni. Questa complessità rappresenta una sfida considerevole, potenzialmente inducendo il modello a effettuare previsioni erronee o a considerare elementi non rilevanti durante la fase di predizione. La gestione di questa complessità richiede un'attenzione particolare per assicurare che il modello sia in grado di discernere correttamente le caratteristiche rilevanti, evitando di essere influenzato in maniera distorta da elementi di disturbo nel contesto delle immagini.

La valutazione della tecnologia disponibile ci conduce a considerarla attualmente poco matura, principalmente a causa della sua capacità di predizione non sufficientemente elevata per un'applicazione pratica nella realtà. Tuttavia, si ritiene che un'implementazione strategica, focalizzata in modo significativo sulla mitigazione del bias associato ai modelli più accurati, potrebbe aprire nuove prospettive.

In particolare, concentrarsi sulla risoluzione del bias potrebbe garantire una selezione più precisa e affidabile delle predizioni, contribuendo a colmare il divario di

maturità attuale. Questa strategia potrebbe consentire una progressiva evoluzione della tecnologia, rendendola idonea per l'applicazione in contesti più ampi e diversificati.

4.d. Lavori Futuri

Un passo ulteriore per avanzare il progetto potrebbe essere quello di riuscire a predire le tempistiche del raggiungimento di una determinata percentuale di completamento o in generale poter indicare quanto tempo è ancora necessario per concludere il processo di costruzione. Ciò potrebbe essere realizzato integrando nel dataset la data di inizio e di fine dei lavori ai timelapse che già abbiamo.
Un'altra applicazione potrebbe consistere nell'integrare i dati temporali e la percentuale di completamento con i costi sostenuti fino a quel momento. Questo consentirebbe di tentare una previsione dei costi sostenuti e/o rimanenti e verificare se sia possibile rispettare il budget del progetto.

5. Bibliografia

[1] https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9687944

[2] https://www.sciencedirect.com/science/article/pii/S1877050918309335

[3] https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=967614

[4] https://arxiv.org/abs/1706.00712

[5]https://www.semanticscholar.org/paper/The-CREENDER-Tool-for-Creating-Multimodal-Datasets-Aprosio-Menini/0558b0a4d480bce184295e4974331d5159cf98b2