



UNIVERSITA' POLITECNICA DELLE MARCHE

FACOLTA' DI INGEGNERIA

Corso di Laurea triennale in ingegneria elettronica

**Implementazione di una rete neurale convoluzionale per la flood detection su
piattaforma embedded**

**Implementation of a convolutional neural network for flood detection on an
embedded system**

Relatore: Chiar.mo

Prof. Turchetti Claudio

Correlatore:

Dott.ssa Falaschetti Laura

Tesi di Laurea di:

Corso Gianluca

Indice

1	Introduzione	4
1.1	Topic	4
1.2	Obiettivo	5
2	Teoria	6
2.1	Machine learning	6
2.2	Reti neurali	7
2.3	CNN	8
2.4	Layers	8
2.4.1	Layer di convoluzione	8
2.4.2	Pooling layer	9
2.4.3	Dense layer	10
2.4.4	Funzioni di attivazione	11
2.4.5	Cross-entropy loss	11
3	Implementazione di una CNN per la flood detection	13
3.1	FloodNet dataset	13
3.2	Flood detection CNN in PyTorch	14
3.2.1	La libreria PyTorch	14
3.2.2	Codice sviluppato	14
3.3	Porting della rete in Keras	19
3.3.1	La libreria Keras	19
3.3.2	La libreria TensorFlow	19
3.3.3	Codice sviluppato	20

4	Risultati sperimentali	23
4.1	Testing con diverse backbones	23
4.1.1	ResNet	23
4.1.2	Xception	25
4.1.3	EfficientNet	26
4.1.4	Confronto backbones	27
4.2	Testing su board	29
4.2.1	Conversione del modello	29
4.2.2	OpenMV Cam H7 Plus	30
5	Conclusione e sviluppi futuri	31
	Bibliografia	34

Capitolo 1

Introduzione

1.1 Topic

La frequenza e la gravità delle catastrofi naturali minacciano salute umana, infrastrutture e sistemi naturali. E' estremamente fondamentale avere informazioni accurate, tempestive e comprensibili per migliorare i sistemi di gestione dei disastri. La raccolta rapida dei dati da aree remote può essere facilitata utilizzando piccoli sistemi aerei senza equipaggio (UAV) che forniscono immagini ad alta risoluzione. Gli UAV dotati di sensori fotografici, come il DJI Mavic Pro quadcopter (Fig. 1.1), possono operare in aree remote e di difficile accesso, analizzando l'immagine e avvisando l'eventuale presenza di calamità come edifici crollati, inondazioni o incendi al fine di mitigare più rapidamente i loro effetti sull'ambiente e sulla popolazione umana. La comprensione della scena visiva di queste immagini raccolte è vitale per una risposta rapida e per il recupero su larga scala post-disastro naturale. Gli algoritmi di deep learning, come le reti neurali convoluzionali (CNNs), sono stati ampiamente riconosciuti come un approccio di primo piano per molte applicazioni di computer vision (riconoscimento, rilevamento e classificazione di immagini/video) e hanno mostrato notevoli risultati in molte applicazioni. Tra i vantaggi derivanti dall'utilizzo di tecniche di deep learning in applicazioni di risposta alle emergenze e di gestione dei disastri troviamo la migliore preparazione e reazione nelle situazioni di criticità temporale e il sostegno ai processi decisionali. In particolare, la classificazione, in grado di fornire etichette alle immagini in tempo reale, si adatta perfettamente a questa situazione fornendo informazioni sulla scena per aiutare la task force a prendere decisioni. Anche se le CNNs hanno avuto sempre più successo in vari compiti di classificazione grazie alla loro velocità di elaborazione dei dati e generazione dei risultati su sistemi embedded, come quelli installati sugli UAV, queste sono ostacolate dall'elevato costo computazionale e dalla memoria di cui tali dispositivi dispongono. Infatti, la dimensione del modello di tali reti è proibitiva dal punto di vista della memoria a disposizione per l'inserimento in essi e ciò porta all'utilizzo di meccanismi di conversione e com-

pressione del modello che ne garantiscono l'inserimento a discapito, però, di leggere perdite di accuratezza. Tuttavia, per molte applicazioni, l'elaborazione locale, vicino al sensore, è preferita rispetto al cloud a causa dei concetti di privacy e sicurezza o del funzionamento in aree remote in cui la connettività è limitata o addirittura assente. [1]



Fig. 1.1: DJI Mavic Pro quadcopter [2]

1.2 Obiettivo

L'obiettivo della tesi è fornire uno studio delle caratteristiche principali di una rete neurale convoluzionale tramite nozioni di teoria (Capitolo 2) e tramite un esempio implementativo per la classificazione di immagini alluvionate e non alluvionate (flood detection). L'esempio è il codice sviluppato da Sahil Khose, Abhiraj Tiwari e Ankita Ghosh, ricercatori dell'Istituto della tecnologia di Manipal, per la FloodNet Challenge EARTHVISION 2021. Il primo task della challenge, prevedeva, appunto, la classificazione in 'Flooded' e 'Non flooded' delle immagini contenute nel FloodNet dataset, fornito appositamente per la sfida. Siccome per l'implementazione del codice è stata usata la libreria PyTorch di PyThon, la tesi ne propone la traduzione utilizzando la libreria Keras, più semplice ed intuitiva (Capitolo 3). Nel capitolo 4, l'elaborato fornisce, inoltre, l'analisi di diverse architetture di rete in termini di accuratezza, perdite di dati e memoria occupata al fine di trovare un modello idoneo per l'inserimento in un sistema embedded.

Capitolo 2

Teoria

2.1 Machine learning

Gli algoritmi di computer vision hanno avuto un rapido sviluppo negli ultimi anni, in particolare, la cooperazione tra computer vision e machine learning ha contribuito allo sviluppo di nuovi algoritmi e al miglioramento delle prestazioni dei sistemi di visione. Il machine learning è un tipo di intelligenza artificiale (AI) che consente ai computer di imparare dai dati senza essere esplicitamente programmati. In altre parole, l'obiettivo del machine learning è progettare metodi che eseguono automaticamente l'apprendimento utilizzando osservazioni del mondo reale ("dati di formazione"), senza una definizione esplicita di regole o logica da parte degli esseri umani ("formatore"/"supervisore"). Questi metodi di apprendimento possono essere suddivisi in tre principali approcci: supervisionati (supervised), semi-supervisionati (semisupervised) e non supervisionati (unsupervised). Nei metodi di apprendimento supervisionati, i dati di allenamento vengono presi a coppie (dato: x e etichetta: y) e l'obiettivo è produrre una previsione y in risposta a un campione x . L'input x può essere un vettore o dati più complessi come immagini, documenti o grafici. Nell'apprendimento non supervisionato, invece, si hanno solo dati di input x senza etichette corrispondenti. L'obiettivo di questo tipo di apprendimento è quello di modellare la distribuzione dei dati al fine di scoprire una struttura interessante in essi. I metodi di apprendimento semi-supervisionati si collocano a metà tra i due precedentemente descritti e vengono utilizzati quando è disponibile una grande quantità di dati di input ma solo alcuni di essi sono etichettati.

Mentre questi algoritmi di apprendimento automatico vengono utilizzati da molto tempo, la capacità di applicare automaticamente calcoli matematici complessi ai dati su larga scala è uno sviluppo recente. Questo perché l'aumento della potenza dei computer di oggi, in termini di velocità e memoria, ha aiutato le tecniche di machine learning ad evolvere per imparare da una grande quantità di dati di formazione. Per esempio, con

più potenza di calcolo e una memoria abbastanza grande, si possono creare reti neurali di molti strati (layers), chiamate deep neural networks.

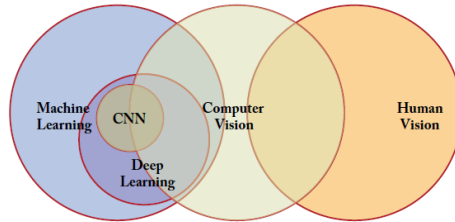


Fig. 2.1: Relazione tra visione umana, computer vision, machine learning, deep learning e CNNs. [3]

2.2 Reti neurali

Le reti neurali sono ispirate dal funzionamento della corteccia cerebrale nei mammiferi. E' importante sottolineare, tuttavia, che questi modelli non riproducono perfettamente il lavoro, la scala e la complessità del cervello umano. I modelli di rete neurale artificiale possono essere intesi come un insieme di unità di elaborazione strettamente interconnesse che operano in base agli input forniti per elaborare le informazioni e generare gli output desiderati. Le reti neurali comprendono una gerarchia di livelli di elaborazione chiamati "network layers" costituiti da un numero di "nodi di elaborazione" (chiamati anche "neuroni" o "unità"). In genere, l'ingresso viene alimentato attraverso un input layer e l'uscita attraverso un output layer che fa le previsioni mentre gli strati intermedi, nascosti, eseguono l'elaborazione. Le singole unità di elaborazione in ogni livello sono i nodi di una rete neurale, questi implementano una "funzione di attivazione" che, dato un input, ha il compito di decidere se il nodo si attiva o meno. I nodi in una rete neurale sono interconnessi e possono comunicare tra loro, ogni connessione ha un peso che ne specifica la forza.

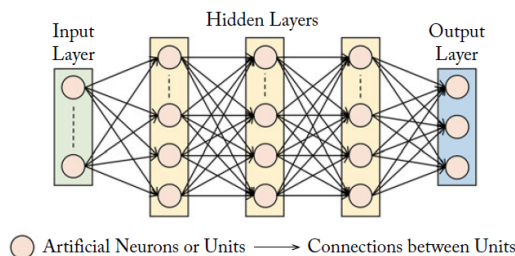


Fig. 2.2: Esempio di una semplice rete neurale con connessione Dense. [3]

2.3 CNN

Le CNNs (Reti neurali convoluzionali) sono una delle categorie più popolari di reti neurali e vengono utilizzate specialmente per i dati di grandi dimensioni (ad esempio: immagini e video). Queste reti operano in modo molto simile a quelle standard. Una differenza fondamentale, tuttavia, è che ogni unità di un layer della CNN è un filtro bidimensionale (o N dimensionale) che è convoluto con l'ingresso. La CNN impara ad associare a un'immagine la sua categoria corrispondente rilevando una serie di rappresentazioni astratte, partendo da quelle più semplici fino ad arrivare a quelle più complesse. Queste rappresentazioni sono poi utilizzate all'interno della rete per prevedere la corretta categoria dell'immagine in input. Come le altre reti neurali, la CNN è composta da diversi elementi di base, chiamati strati (layers), dei quali è fornita una descrizione dettagliata nel paragrafo successivo.

2.4 Layers

2.4.1 Layer di convoluzione

Il layer di convoluzione è il componente più importante di una CNN. Esso comprende una serie di filtri (chiamati anche nuclei convoluzionali) che sono convoluti con un dato input per generare una mappa delle caratteristiche di output. Ogni filtro, in uno strato convoluzionale, è una griglia di numeri discreti, chiamati pesi, che vengono appresi durante la formazione della CNN. Questa procedura di apprendimento comporta un'inizializzazione casuale dei valori del filtro all'inizio dell'allenamento. In seguito, date coppie input-output, i pesi sono sintonizzati durante l'apprendimento. Data una mappa 2D delle caratteristiche di input e un filtro convoluzionale con matrice di dimensioni 4×4 e 2×2 , rispettivamente, il layer di convoluzione moltiplica il filtro 2×2 con una patch evidenziata (anch'essa di dimensione 2×2) della mappa di input e somma tutti i valori per generarne uno di output. Si noti che il filtro scorre lungo la larghezza e l'altezza della mappa di input e questo processo continua fino a quando non può più scorrere ulteriormente.

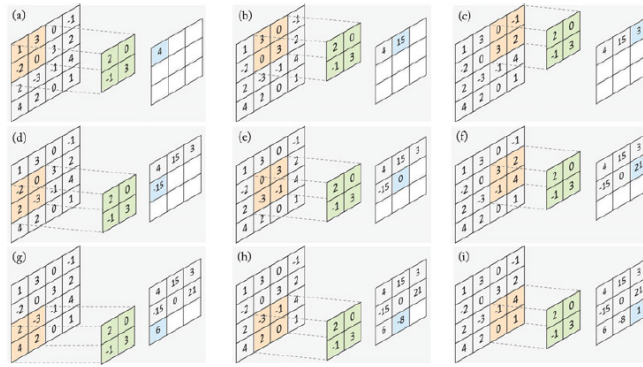


Fig. 2.3: Funzionamento di uno strato di convoluzione. (a)-(i) mostrano i calcoli eseguiti in ogni fase, come il filtro è "scivolato" sulla mappa delle caratteristiche di input per calcolare il valore corrispondente nella mappa delle caratteristiche di output. Il filtro 2x2 (mostrato in verde) è moltiplicato con la regione di dimensioni uguali (mostrata in arancione) all'interno di una mappa caratteristica 4x4 di ingresso e i valori risultanti sono sommati per ottenere un'entrata corrispondente (mostrata in blu) nella mappa caratteristica di output ad ogni fase di convoluzione. [3]

In Fig. 2.3 possiamo vedere che la dimensione spaziale della mappa di uscita è ridotta rispetto a quella della mappa d'ingresso. Precisamente, per un filtro con dimensioni $f \times f$ e una mappa di input con dimensioni $h \times w$ e passo s , le dimensioni della caratteristica d'uscita sono date da:

$$h' = \left\lfloor \frac{h - f + s}{s} \right\rfloor, \quad w' = \left\lfloor \frac{w - f + s}{s} \right\rfloor,$$

dove $\lfloor \cdot \rfloor$ si chiama *funzione floor* e prende la parte intera del valore contenuto al suo interno.

2.4.2 Pooling layer

Il Pooling layer opera su blocchi della mappa di input e ne combina le caratteristiche mediante una *funzione di pooling* come la media o il massimo. Come con il layer di convoluzione, anche in questo caso, abbiamo bisogno di specificare la dimensione della regione raggruppata (pooled) e il passo. La figura 2.4 mostra l'operazione di *max pooling*, dove viene scelta l'attivazione massima dal blocco di valori selezionato. Questa finestra viene fatta scorrere attraverso la mappa delle caratteristiche di input con un determinato passo (1 nell'esempio).

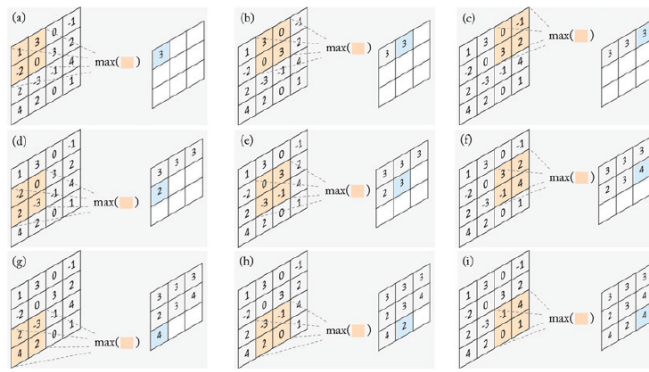


Fig. 2.4: Funzionamento del max-pooling layer quando la dimensione della regione di pooling è 2×2 e il passo è 1. (a)-(i) mostra come la regione raggruppata nella mappa di input (mostrata in arancione) viene fatta scorrere ad ogni passaggio per calcolare il valore corrispondente nella mappa delle caratteristiche di output (mostrata in blu). [3]

Se la dimensione della regione raggruppata è definita come $f \times f$, con un passo s , la dimensione della mappa delle caratteristiche di uscita è data da:

$$h' = \left\lfloor \frac{h - f + s}{s} \right\rfloor, w' = \left\lfloor \frac{w - f + s}{s} \right\rfloor.$$

L'operazione di pooling effettivamente sotto campiona la mappa delle caratteristiche di input. Tale processo è utile per ottenere una rappresentazione delle caratteristiche compatta e invariante per cambiamenti sulla scala degli oggetti e sulla traduzione dell'immagine.

2.4.3 Dense layer

Il dense layer è uno strato profondamente collegato con quello che lo precede ed è il più comunemente usato nelle reti neurali artificiali. Il neurone del dense layer, in un modello, riceve l'uscita da ogni neurone del suo strato precedente e ne esegue la moltiplicazione matrice-vettore. La regola generale di questa moltiplicazione è che il vettore riga e il vettore colonna devono avere lo stesso numero di colonne:

$$\begin{aligned}
 \mathbf{Ax} &= \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \\
 &= \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \end{bmatrix}.
 \end{aligned}$$

Dove A è una matrice ($M \times N$) e x è un vettore ($1 \times N$). I valori dentro la matrice sono i parametri addestrati degli strati precedenti e il risultato è un vettore N -dimensionale. Quindi, fondamentalmente, il dense layer è usato per cambiare la dimensione dei vettori per mezzo di ogni neurone. [4]

2.4.4 Funzioni di attivazione

I layers di una CNN sono spesso seguiti da una funzione di attivazione non lineare (o lineare a tratti). Questa funzione prende un input e lo comprime in un piccolo intervallo, ad esempio $[0; 1]$ e $[-1; 1]$. L'applicazione di una funzione non lineare dopo i vari strati è molto importante, in quanto consente alla rete neurale di imparare le mappature non lineari. In assenza di non linearità, una rete sovrapposta di soli layers è equivalente a una mappatura lineare dal dominio di input al dominio di output. In Fig. 2.5, una lista delle più comuni funzioni di attivazione usate nelle reti neurali profonde:

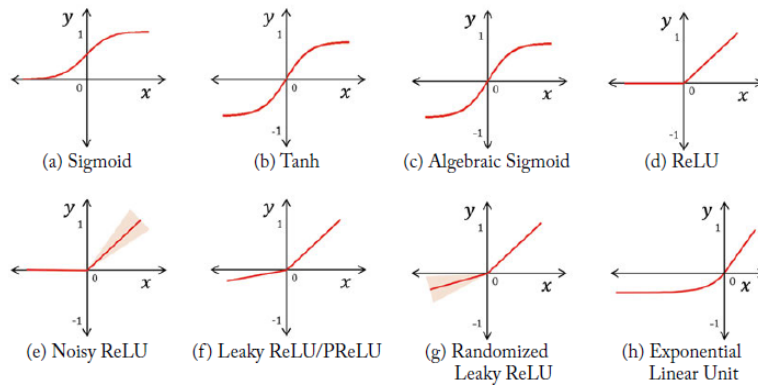


Fig. 2.5: Alcune tra le più comuni funzioni di attivazione utilizzate nelle reti neurali profonde. [3]

2.4.5 Cross-entropy loss

Il layer finale di una CNN, il quale viene utilizzato soltanto durante l'allenamento della rete, utilizza una "funzione di perdita" per stimare la qualità delle previsioni fatte sui dati etichettati. Questa funzione quantifica la differenza tra l'output stimato dal modello (predizione) e l'output corretto. Un esempio di funzione di perdita è la *cross-entropy loss*, definita come:

$$L(p, y) = - \sum_n y_n \log(p_n), \quad n \in [1, N],$$

con N che rappresenta il numero di neuroni nel layer di output, y l'output desiderato e p la probabilità per ogni categoria di output. La probabilità di ogni classe può essere calcolata usando la funzione *soft-max*:

$$p_n = \frac{\exp(\hat{p}_n)}{\sum_k \exp(\hat{p}_k)},$$

dove la \hat{p} è il punteggio finale non normalizzato del layer precedente.

Capitolo 3

Implementazione di una CNN per la flood detection

3.1 FloodNet dataset

FloodNet è un dataset costituito da immagini ad alta definizione acquisite da un sistema UAV dopo l'uragano Harvey che ha colpito gli Stati Uniti nel 2017. Il set è costituito da 2343 immagini di dimensione 3000x4000, divise in training (1445), validation (450) e test (448). I dati di training e validation sono quelli utilizzati per l'addestramento della rete. In particolare, i primi sono quelli da cui vengono estratte le features, mentre gli altri sono quelli utilizzati per validare le caratteristiche apprese. I dati di test, invece, sono quelli utilizzati per calcolare l'evaluation, cioè l'affidabilità delle previsioni fatte dalla rete rispetto alla realtà. Generalmente, tutti questi dati sono etichettati, in questo dataset, invece, le uniche immagini etichettate sono parte di quelle di training (398 su 1445) e le etichette utilizzate sono "Flooded"(alluvionato) e "Non-Flooded"(non alluvionato), in particolare 51 sono Flooded e 347 Non-Flooded.

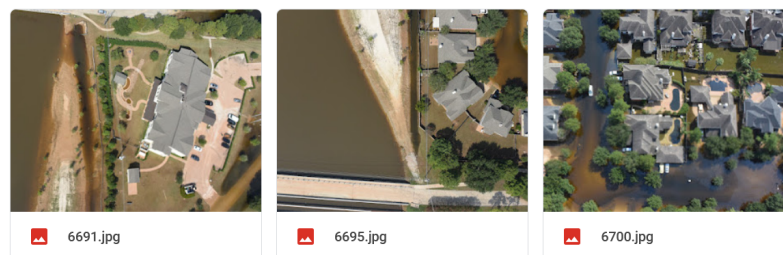


Fig. 3.1: Alcune immagini prese dal FloodNet dataset. [5]

3.2 Flood detection CNN in PyTorch

3.2.1 La libreria PyTorch

PyTorch è una libreria di apprendimento automatico open source per Python. È stata sviluppata dal gruppo di ricerca sull'intelligenza artificiale di Facebook. A differenza di Torch che è scritto in Lua (un linguaggio di programmazione relativamente impopolare), PyTorch sfrutta la crescente popolarità di Python. Dalla sua introduzione, questa libreria è diventata rapidamente la preferita tra i ricercatori di apprendimento automatico perché consente di costruire facilmente alcune architetture complesse. [6]

3.2.2 Codice sviluppato

Di seguito sono riportati, commentati, i blocchi principali del codice sviluppato in PyTorch dai ricercatori dell'Istituto della tecnologia di Manipal, per la FloodNet Challenge EARTHVISION 2021. [7]

Per prima cosa viene creato il collegamento con Google Drive, in cui è presente il dataset, e vengono importate le varie librerie utili per l'implementazione della rete.

```
1  from google.colab import drive
2  drive.mount('/content/drive')
3
4  # TensorFlow and tf.keras
5  import tensorflow as tf
6  # Helper libraries
7  import numpy as np
8  import matplotlib.pyplot as plt
9  from tensorflow import keras
10 print(tf.__version__)
11 %load_ext tensorboard
12 import torchvision
13 from torchvision import transforms as T
14 import torch
15 import torch.nn as nn
16 import numpy as np
17 import pandas as pd
18 from tqdm.auto import tqdm
19 from IPython.display import Image
20 import os
21 from skimage import io
22 import time
23 from sklearn.model_selection import train_test_split
24 from collections import Counter
25 import copy
26 from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, f1_score
27 from torch.utils.tensorboard import SummaryWriter
28 from skimage.transform import resize as io_resize
29 import cv2
30 import json
31 import datetime
```

Successivamente viene creata la classe TIDataset che prenderà in ingresso immagini ed etichette e creerà i dataset corrispondenti.

```
34 class TIDataset(torch.utils.data.Dataset):
35     """
36     This is to use with array of files and array of labels as an input
37     """
38     def __init__(self, X, y, transform=None):
39         self.X = X
40         self.y = y
41         self.transform = transform
42
43     def __len__(self):
44         return len(self.X)
45
46     def __getitem__(self, idx):
47         image = io.imread(self.X[idx])
48         if self.transform is not None:
49             image = self.transform(image)
50         if self.y is None:
51             return (image, idx)
52         return (image, self.y[idx])
```

Nel blocco seguente, viene definita la data augmentation (aumento dei dati), cioè le immagini vengono trasformate e sommate a quelle originali in modo da arricchire il dataset e renderlo più vasto per allenare la rete. In questo caso, l'aumento dei dati viene fatto distintamente per i dati di training e di validation. In particolare, per i primi viene fatto un crop, cioè l'immagine viene tagliata della dimensione specificata all'interno delle parentesi, poi capovolta orizzontalmente e verticalmente, infine ridimensionata e normalizzata, per i secondi, invece, viene soltanto fatto il crop, il resize e la normalizzazione.

```
54 # Data augmentation
55 transforms = {
56     'train' : T.Compose([
57         T.ToPILImage(),
58         T.RandomResizedCrop(3000),
59         T.RandomHorizontalFlip(),
60         T.RandomVerticalFlip(),
61         T.Resize((300, 400)),
62         T.ToTensor(),
63         T.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),
64     ]),
65     'valid' : T.Compose([
66         T.ToPILImage(),
67         T.CenterCrop(3000),
68         T.Resize(500),
69         T.ToTensor(),
70         T.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
71     ])
72 }
```

Successivamente vengono presi i file flooded e non flooded dalla cartella di training del dataset e vengono assegnate manualmente le etichette 1 e 0, rispettivamente. Siccome nel FloodNet dataset i validation data non sono etichettati ma, per validare i risultati, è essenziale che essi lo siano, questi vengono presi dividendo in due le 398 immagini di allenamento di cui si conoscono le etichette. La divisione viene fatta dalla `train_test_split()` ottenendo, quindi, 199 immagini per il training e 199 per la validation.

```

74 flooded_root = '/content/drive/MyDrive/FloodNet Challenge @ EARTHVISION 2021 - Track 1/Train/Labeled/Flooded/image/'
75 flooded_files = [flooded_root+name for name in os.listdir(flooded_root)]
76 flooded_y = [1]*len(flooded_files)
77
78 non_flooded_root = '/content/drive/MyDrive/FloodNet Challenge @ EARTHVISION 2021 - Track 1/Train/Labeled/Non-Flooded/image/'
79 non_flooded_files = [non_flooded_root+name for name in os.listdir(non_flooded_root)]
80 non_flooded_y = [0]*len(non_flooded_files)
81
82 X = flooded_files+non_flooded_files
83 y = flooded_y+non_flooded_y
84 print(len(X), len(y))
85
86 # metà dei dati di train (labeled) vengono utilizzati come validation
87 X_train, X_valid, y_train, y_valid = train_test_split(X, y, train_size=0.5, stratify=y)
88 print(len(X_train), Counter(y_train), len(X_valid), Counter(y_valid))

```

Una volta definiti `X_train`, `Y_train`, `X_valid` e `Y_valid`, vengono passati all'interno della classe `T1Dataset` per creare il `train_set` e il `valid_set` con un `BATCH_SIZE = 64`, che rappresenta il numero di campioni che vengono prelevati per ogni epoca, cioè per ogni ciclo di apprendimento della rete.

```

90 BATCH_SIZE = 64
91
92 unique_labels, counts = np.unique(y_train, return_counts=True)
93 class_weights = [1/c for c in counts]
94 # class_weights[1]*=1.5 # SKEWING THE SAMPLER MORE
95 sample_weights = [0] * len(y_train)
96 for idx, lbl in enumerate(y_train):
97     sample_weights[idx] = class_weights[lbl]
98 print(sample_weights)
99 sampler_train = torch.utils.data.WeightedRandomSampler(weights=sample_weights, num_samples=len(sample_weights), replacement=True)
100
101
102 train_set = T1Dataset(X_train, y_train, transform=transforms['train'])
103 valid_set = T1Dataset(X_valid, y_valid, transform=transforms['valid'])
104
105 dataloaders = {
106     'train': torch.utils.data.DataLoader(train_set, batch_size=BATCH_SIZE, sampler=sampler_train),
107     'valid': torch.utils.data.DataLoader(valid_set, batch_size=BATCH_SIZE, shuffle=True)
108 }
109

```


Per effettuare il training semi supervisionato, ai due dataset precedenti viene aggiunto quello che contiene i file non etichettati (unlabeled), ai quali vengono assegnate le stesse trasformazioni fatte sui dati di validation.

```

110 #SEMISUPERVISED TRAINING
111 unlabeled_root = "/content/FloodNet Challenge @ EARTHVISION 2021 - Track 1/Train/Unlabeled/image/"
112 unlabeled_file_names = os.listdir(unlabeled_root)
113 unlabeled_file_names = list(map(lambda x: unlabeled_root+x, unlabeled_file_names))
114
115 # Adding validation set to unlabeled data
116 for ds in ['Validation', 'Test']:
117     data_root_ = f"/content/FloodNet Challenge @ EARTHVISION 2021 - Track 1/{ds}/image/"
118     data_file_names_ = os.listdir(data_root_)
119     unlabeled_file_names += list(map(lambda x: data_root_+x, data_file_names_))
120
121 unlabeled_file_names.sort()
122 pseudo_labels = np.array([-1]*len(unlabeled_file_names))
123
124 unlabeled_ds = TlDataset(unlabeled_file_names, None, transform=transforms['valid'])
125 unlabeled_loader = torch.utils.data.DataLoader(unlabeled_ds, batch_size=BATCH_SIZE, shuffle=True, num_workers=0)
126
127 dataloaders['unlabeled'] = unlabeled_loader

```

Una volta preparati tutti i dati da dare in pasto alla rete, viene definita la classe *sexy_tuner()*, nella quale viene creato il modello tramite la rete convoluzionale preimplementata ResNet 18, approfondimenti sull'architettura ResNet al paragrafo 4.1.1. All'output della rete viene aggiunto il layer *torch.nn.Linear* che applica una trasformazione lineare all'ingresso che riceve per ridurne le dimensioni. Viene, poi, utilizzata la funzione *CrossEntropyLoss()*, descritta nel paragrafo 2.4.5, per il calcolo delle perdite e viene scelto come ottimizzatore l'SGD con *learning_rate = 0.01*, il quale ha la funzione di rendere ottimi i valori di accuratezza dell'allenamento. Il modello, così definito, viene, poi, passato in ingresso alla classe *ss_train*, dove *ss* sta per *semisupervised*, che si occupa di implementare l'apprendimento della rete semi supervisionato. Tra gli altri parametri passati alla classe troviamo *num_epochs* che rappresenta il numero di epoche di allenamento, in questo caso pari a 200.

```

244 def sexy_tuner():
245     for lr in [0.01]:
246         # model = torch.hub.load('facebookresearch/dino:main', 'dino_resnet50')
247         model = torchvision.models.resnet18(pretrained=True)
248         final_layer_in = model.fc.in_features
249         model.fc = torch.nn.Linear(final_layer_in, 2)
250         model = model.to(device)
251         criterion = torch.nn.CrossEntropyLoss()
252         optimizer = torch.optim.SGD(model.parameters(), lr=lr)
253         # optimizer = torch.optim.Adam(model.parameters())
254
255         model, best_ep = ss_train(model, criterion, optimizer, num_epochs, start_alpha_from=10, reach_max_alpha_in=150)
256         current_time = datetime.datetime.now()
257         torch.save({
258             'epoch': best_ep,
259             'model_state_dict': model.state_dict(),
260             'optimizer_state_dict': optimizer.state_dict(),
261         }, f"/content/drive/MyDrive/checkpoints/in_{lr}_{best_ep}_{current_time.day}_{current_time.hour}_{current_time.minute}_best.pt")
262
263     return (model, best_ep)

```

L'allenamento semi supervisionato viene, infine, perfezionato da quello supervisionato, il quale lavora solo sui dati definiti all'inizio, cioè quelli etichettati e splittati. Il modello è identico a quello utilizzato per l'allenamento semi supervisionato, l'unica differenza è che, in questo caso, alla classe *train*, che si occupa di implementare il supervised training, vengono passate soltanto 5 epoche, perchè la maggiorparte dell'apprendimento viene svolto nella fase semi supervised.

```
294 #SUPERVISED TRAINING
295 model = torchvision.models.resnet18(pretrained=True)
296 final_layer_in = model.fc.in_features
297 model.fc = torch.nn.Linear(final_layer_in, 2)
298
299 checkpoint = torch.load('/content/drive/MyDrive/checkpoints/fn_0.01_6_21_20_4_best.pt', map_location=device)
300 model.load_state_dict(checkpoint['model_state_dict'])
301 model = model.to(device)
302 optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
303 optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
304 criterion = torch.nn.CrossEntropyLoss()
305
306 train(model, criterion, optimizer, 5)
```

3.3 Porting della rete in Keras

Il codice descritto nel paragrafo precedente è stato tradotto utilizzando la libreria Keras, la quale sta acquisendo sempre più popolarità negli ultimi anni grazie alla velocità di implementazione e alla semplicità degli strumenti di cui dispone (Fig. 3.2).

3.3.1 La libreria Keras

Keras è un'API di alto livello costruita sulla base di un motore di backend, come TensorFlow, di cui beneficia dei vantaggi. Keras fornisce un insieme di astrazioni di livello superiore, che rendono facile e veloce la configurazione e la sperimentazione di reti neurali. La libreria è costituita da numerosi strumenti che facilitano il lavoro con i dati di immagine e testo. [8]

3.3.2 La libreria TensorFlow

Tensorflow è stata originariamente sviluppata dal team di Google Brain ed è scritta con una API di Python su un motore C/C++ per il calcolo numerico. Il controllo completo della programmazione è fornito dall'API di livello più basso, chiamato TensorFlow Core. Quest'ultimo è consigliato ai ricercatori di machine learning che hanno bisogno di livelli sottili di controllo sui loro modelli. Le API di livello superiore sono costruite sulla base di TensorFlow Core e sono più facili da imparare e da usare, rispetto ad esso. Tensorflow offre funzionalità di differenziazione automatica, che semplificano il processo di definizione di nuove operazioni nella rete utilizzando i grafici di flusso dei dati per eseguire calcoli numerici. Tensorflow supporta multiple backend, CPU o GPU su desktop, server o piattaforme mobili. Ha buoni legami con Python e C++ ed è dotato di strumenti per supportare il rinforzo dell'apprendimento. [9]

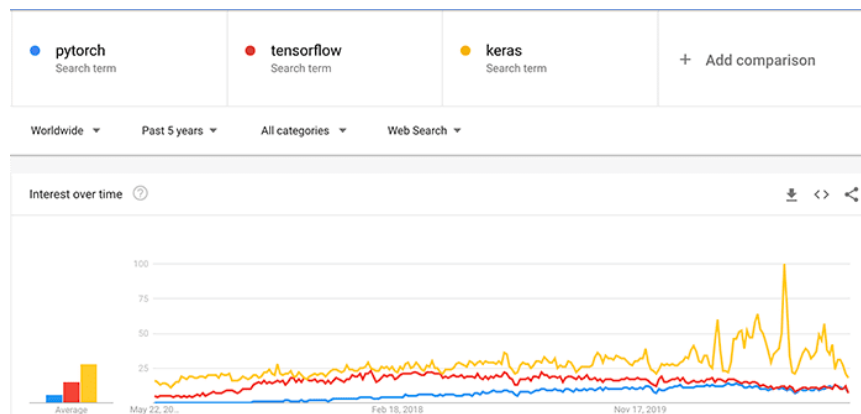


Fig. 3.2: Trend di popolarità delle librerie Keras, TensorFlow e PyTorch tra il 2018 e il 2019. [10]

3.3.3 Codice sviluppato

Di seguito è riportato il codice, commentato, della CNN sviluppata tramite le librerie Keras e TensorFlow. Lo studio è focalizzato sulla parte supervisionata, quella non supervisionata è lasciata per eventuali sviluppi futuri. Il codice è stato implementato sfruttando Google Colab, un tool fornito da Google che consente di scrivere codice in Python direttamente dal proprio browser. A causa dei limiti di RAM imposti dalla versione gratuita del software, il batch size è stato abbassato a 32 e il numero di epoche a 100.

Come fatto in PyTorch, quindi, vengono importate le librerie utili e creato il collegamento con Google Drive.

```
1 #Definizione librerie
2 import numpy as np
3 import tensorflow as tf
4 import tensorflow_datasets as tfds
5 from tensorflow import keras
6 import matplotlib.pyplot as plt
7 from os import path
8 from skimage import io
9 from tensorflow.keras import layers
10 from keras.layers import Dense, Flatten, Activation, Dropout, GlobalAveragePooling2D
11 from PIL import Image
12 from IPython.display import Image
13 from keras.preprocessing.image import ImageDataGenerator
14 import cv2
15 from pathlib import Path
16 from keras import Model, layers
17 from keras.models import load_model, model_from_json
18 print(tf.__version__)
19 from google.colab import drive
20 drive.mount('/content/drive')
21 import os
```

Nel blocco seguente viene definita la rotta di salvataggio del modello creato e vengono definiti i path delle immagini flooded e non flooded. Siccome all'interno delle cartelle etichettate del dataset ci sono anche le maschere, tipi di dati non utilizzati ai fini della classificazione, vengono estratte soltanto le immagini utili e viene creata una nuova cartella *train*, che a sua volta conterrà le sottocartelle *flooded* e *non_flooded*, nelle quali inseriamo i dati estratti. La creazione delle nuove cartelle è necessaria per utilizzare la funzione *generator_flow_from_directory()* di TensorFlow, definita in successivamente.

```

23 #Definizione rotta di salvataggio del modello
24 MODELS_PATH = '/content/drive/MyDrive/Colab Notebooks'
25 name_model = os.path.join(MODELS_PATH, 'ResNet50.h5')
26 RESIZE=(400,300)
27 BATCH_SIZE=32
28
29 #Definizione rotte immagini etichettate
30 flooded_root = '/content/drive/MyDrive/FloodNet Challenge @ EARTHVISION 2021 - Track 1/Train/Labeled/Flooded/image'
31 non_flooded_root = '/content/drive/MyDrive/FloodNet Challenge @ EARTHVISION 2021 - Track 1/Train/Labeled/Non-Flooded/image'
32 #Siccome all'interno delle cartelle Flooded e Non-Flooded sono presenti anche le maschere
33 #(che non servono ai fini della classificazione)prelevo solo le immagini di interesse e le salvo in delle nuove cartelle
34 !mkdir -p train/flooded
35 !mkdir -p train/non_flooded
36 #Inserimento immagini nelle cartelle create
37 !cp '$flooded_root'/*.jpg train/flooded
38 !cp '$non_flooded_root'/*.jpg train/non_flooded

```

Di seguito viene definita la data augmentation. Questa, in Keras, non è stata differenziata tra train e validation in quanto è stato usato un solo generatore diviso. Tuttavia, l'aumento dei dati è significativo specialmente per i dati di allenamento.

```

41 #Data augmentation
42 augmentation = tf.keras.Sequential([
43     layers.Normalization(mean=[0.485, 0.456, 0.406],
44     variance=[np.square(0.299),
45     np.square(0.224),
46     np.square(0.225)]),
47     layers.RandomCrop(height=3000, width=3000, seed=None),
48     layers.RandomTranslation(height_factor=0.5, width_factor=0.5,
49     fill_mode='reflect', interpolation='bilinear', seed=None)
50 ])

```

Come accennato sopra, quindi, viene creata la classe *train_datagen* che contiene la data augmentation e il valore dello splitting. Da questa, tramite *train_datagen.flow_from_directory()*, vengono creati i due generatori di train e validation tramite il path *train_data_dir* della cartella *train*, che contiene le immagini etichettate divise in flooded e non flooded. All'interno delle due funzioni viene esplicitato il tipo di classi (*class_mode = binary*), le etichette vengono, poi, automaticamente attribuite dai generatori (*1 : flooded; 0: non flooded*).

```

52 train_data_dir = '/content/drive/MyDrive/train'
53 #Creazione dei generatori di training e validation con le immagini aumentate e splittate
54 train_datagen = ImageDataGenerator(
55     augmentation,
56     horizontal_flip=True,
57     vertical_flip=True,
58     validation_split=0.5)
59
60 train_generator = train_datagen.flow_from_directory(
61     train_data_dir,
62     target_size=RESIZE,
63     batch_size=BATCH_SIZE,
64     class_mode='binary',
65     subset='training')
66
67 validation_generator = train_datagen.flow_from_directory(
68     train_data_dir,
69     target_size=RESIZE,
70     batch_size=BATCH_SIZE,
71     class_mode='binary',
72     subset='validation')

```

Preparati i dati, viene definito il modello. La ResNet 18, non disponibile in Keras, viene sostituita con la ResNet 50 V2, approfondimenti sulle ResNet al paragrafo 4.1.1. A causa di incompatibilità dimensionali, all'output della ResNet viene aggiunto un GlobalAveragePooling 2D layer, che sottocampiona l'ingresso tramite il suo valore medio (Paragrafo 2.4.2). Infine, il Linear layer di PyTorch viene sostituito con un Dense layer, che va ugualmente a ridurre le dimensione dell'ingresso, in questo caso, però, tramite moltiplicazioni matrice vettore (Paragrafo 2.4.3). Per quanto riguarda, invece, ottimizzatore e funzione di perdite sono le stesse utilizzate in PyTorch. Il modello, così definito, viene addestrato solo con i dati etichettati (supervised training) tramite la funzione *model.fit()*, la quale prende in ingresso il *train_generator*, il numero di epoche e il *validation_datagen* e che permette l'allenamento della rete senza la definizione esplicita della classe di training, come veniva fatto in PyTorch. Il modello addestrato viene, infine, salvato in formato h5, con il quale lavora Keras, all'interno della variabile *name_model*, definita all'inizio.

```

75 #Creazione modello
76 conv_base = tf.keras.applications.ResNet50V2(include_top=False,
77     weights="imagenet")
78 x = conv_base.output
79 x = layers.GlobalAveragePooling2D()(x) #Sottocampiona l'ingresso
80 x = layers.Dense(128, activation='relu')(x)
81 output = layers.Dense(2, activation='softmax')(x) #Riduce la dimensione del vettore
82 model = Model(conv_base.input, output)
83 optimizer = keras.optimizers.SGD(learning_rate=0.01)
84 model.compile(loss='sparse_categorical_crossentropy',
85     optimizer=optimizer,
86     metrics=['accuracy'])
87 #Training con 100 epoche
88 history = model.fit(
89     train_generator, epochs=100, validation_data=validation_generator, steps_per_epoch=(200/BATCH_SIZE))
90 model.save(name_model)

```

Capitolo 4

Risultati sperimentali

4.1 Testing con diverse backbones

Completato il porting del codice, la CNN è stata testata con tre diverse reti pre addestrate (backbones), scelte per le prestazioni di accuratezza e memoria che hanno fornito con l'ImageNet validation dataset, un'ampia base di dati costituita da più di 14 milioni di immagini. Le caratteristiche principali di queste reti sono riportate sotto. [11]

4.1.1 ResNet

ResNet sta per Residual Network ed è una rete neurale innovativa introdotta nel 2015, della quale esistono diverse varianti in base al numero di layers da cui sono costituite. Per esempio, la ResNet 50 denota la variante che lavora con 50 layers. Quando si lavora con reti neurali convoluzionali profonde per risolvere un problema relativo alla visione artificiale, gli esperti di apprendimento automatico si impegnano a impilare più strati con lo scopo di ridurre al minimo l'errore. L'aumento del numero di layers della rete neurale, tuttavia, può portare alla fuga o all'esplosione del gradiente, facendo aumentare il tasso di errore di allenamento e test. La ResNet è stata creata con l'obiettivo di affrontare questo problema facendo uso di blocchi residui che, tramite le connessioni skip, migliorano la precisione dei modelli (Fig. 4.1). La connessione skip collega le attivazioni tra due layers saltandone alcuni intermedi, formando un blocco residuo. Il vantaggio di aggiungere questi tipi di collegamenti skip è che, se uno strato danneggia le prestazioni dell'architettura, allora verrà saltato dalla regolarizzazione portando alla formazione di una rete neurale profonda senza i problemi causati dalla fuga o dall'esplosione del gradiente.

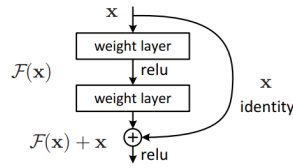


Fig. 4.1: Connessione skip. [12]

La prima architettura di questo tipo è stata la ResNet 34, che ha portato all’inserimento di connessioni skip per trasformare una rete semplice nella sua controparte di rete residua. L’architettura di ResNet 34 è stata ispirata dalle reti neurali VGG (VGG-16, VGG-19), costituite da filtri 3x3. Tuttavia, rispetto a queste, le ResNets hanno meno filtri e minore complessità. L’architettura ResNet 50 si basa su quella della 34, con un’unica differenza, ciascuno dei blocchi a 2 strati in Resnet 34 è stato sostituito con un blocco a collo di bottiglia a 3 strati garantendo una precisione molto più elevata. La ResNet 50 è stata migliorata, poi, con la versione 2 che presenta migliori disposizioni degli strati nel blocco residuo. Di seguito le architetture delle principali ResNets:

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Fig. 4.2: Architetture dei modelli ResNet a 18,34,50,101 e 152 layers. [13]

4.1.2 Xception

Xception è una architettura di rete neurale convoluzionale conosciuta come versione "estrema" del modulo Inception, il quale effettua molteplici convoluzioni sullo stesso input e combina tutte le uscite lasciando decidere al modello quali e quante caratteristiche prendere. A causa delle convoluzioni che avvengono non solo spazialmente, ma anche attraverso la profondità, quest'ultima assume dimensioni troppo grandi rendendo il modello computazionalmente inefficiente. Tuttavia, la profondità può essere ridotta facendo delle convoluzioni 1x1 che ne comprimono la dimensione. La Inception, quindi, effettua le convoluzioni per comprimere l'input originale, e, da ciascuno di quegli spazi d'ingresso, usa diversi tipi di filtri su tutta la profondità. La Xception, invece, inverte questo passaggio, prima applica i filtri e poi comprime lo spazio di input usando la convoluzione 1x1, applicandola su tutta la profondità. L'altra differenza tra Inception ed Xception è la presenza nell'una e l'assenza nell'altra di una non-linearità dopo la prima operazione. Nel modello Inception, entrambe le operazioni sono seguite da una non-linearità Relu, mentre l'Xception non introduce alcuna non-linearità. Di seguito l'architettura di rete Xception:

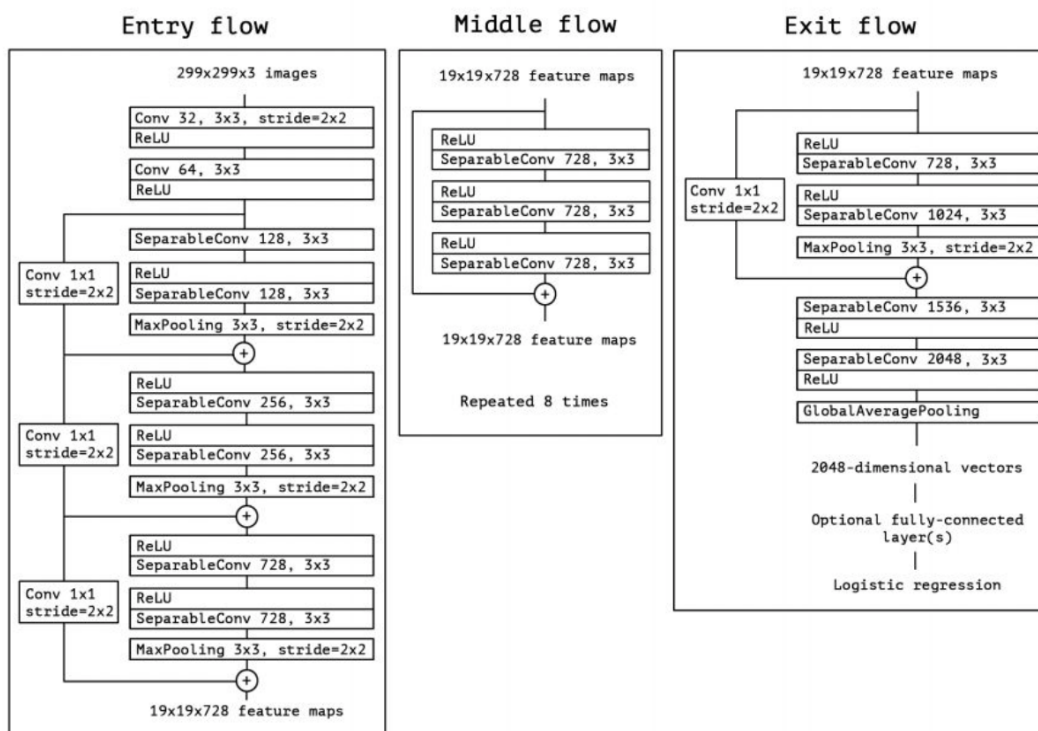


Fig. 4.3: Architettura di rete Xception. I dati prima passano attraverso il flusso di entrata, poi attraverso quello centrale (ripetendosi 8 volte) e, infine, attraverso il flusso di uscita. [14]

4.1.3 EfficientNet

La EfficientNet è costituita principalmente dagli MBConv (blocchi residui invertiti), tipi di blocchi residui che hanno una struttura invertita per motivi di ottimizzazione. In un blocco residuo tradizionale l'ingresso ha un elevato numero di canali, che sono compressi con una convoluzione 1x1 e, poi, di nuovo aumentati con un'altra convoluzione 1x1 in modo da poter aggiungere ingresso e uscita. Al contrario, un blocco residuo invertito, prima allarga l'ingresso con una convoluzione 1x1, poi usa una convoluzione 3x3 depthwise (che riduce notevolmente il numero di parametri), quindi usa un'altra convoluzione 1x1 per ridurre il numero di canali in modo da poter aggiungere ingresso e uscita. Questa struttura aiuta a ridurre il numero complessivo di operazioni richieste e la dimensione del modello. Ulteriori miglioramenti sono stati introdotti con la EfficientNet v2 che garantisce una maggiore velocità di formazione e una migliore efficienza dei parametri rispetto ai modelli precedenti.

Stage i	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels \hat{C}_i	#Layers \hat{L}_i
1	Conv3x3	224×224	32	1
2	MBConv1, k3x3	112×112	16	1
3	MBConv6, k3x3	112×112	24	2
4	MBConv6, k5x5	56×56	40	2
5	MBConv6, k3x3	28×28	80	3
6	MBConv6, k5x5	14×14	112	3
7	MBConv6, k5x5	14×14	192	4
8	MBConv6, k3x3	7×7	320	1
9	Conv1x1 & Pooling & FC	7×7	1280	1

Fig. 4.4: Architettura di rete EfficientNet-B0. [15]

Confrontando le EfficientNets con altre CNNs, in generale, le prime raggiungono una maggiore precisione e una migliore efficienza rispetto alle altre, riducendo le dimensioni dei parametri (Fig. 4.5).

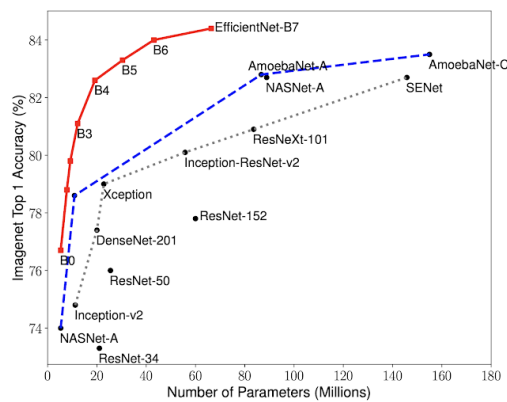


Fig. 4.5: Confronto tra dimensione e precisione dei modelli. [16]

4.1.4 Confronto backbones

Le tre reti scelte sono state, quindi, testate con il FloodNet dataset e i risultati ottenuti in termini di accuratezza e perdite con i dati di training, validation e test e in termini di memoria occupata dal modello sono stati riportati nella tabella seguente.

	Train acc.	Train loss	Valid acc.	Valid loss	Test acc.	Test loss	Memory size (MB)
ResNet 50 v2	1.0000	0.0037	0.9394	0.3038	0.6607	0.3392	91.397
EfficientNet v2 B0	1.0000	0.0121	0.9192	0.2461	0.6294	0.3705	23.911
Xception	0.9950	0.0127	0.9242	0.2516	0.6808	0.3191	80.976

Tabella 4.1: Confronto backbones

Dalla tabella 4.1, si nota che le reti forniscono prestazioni simili in termini di accuratezza e perdite, presentano, invece, grosse differenze in termini di memoria occupata dal modello. Infatti, quello generato utilizzando la EfficientNet v2 B0 pesa circa 24 MB mentre i modelli generati con la ResNet 50 v2 e con la Xception pesano rispettivamente 91 e 81 MB.

I valori di train accuracy, train loss, validation accuracy e validation loss sono stati ottenuti dall'ultima epoca dell'output della funzione *model.fit()*, per quanto riguarda, invece, il calcolo di test accuracy, test loss e memory size sono state implementate delle funzioni apposite.

In particolare, per calcolare la memoria occupata, viene scaricato il modello, stimata la dimensione in bytes tramite la funzione *os.path.getsize()* e, poi, convertita in MB (Megabytes) e KB (Kilobytes).

```
93 # Download saved model
94 try:
95     from google.colab import files
96     files.download(name_model)
97 except ImportError:
98     pass
99
100 #Funzione per il calcolo della memory size
101 def get_file_size(file_path):
102     size = os.path.getsize(file_path)
103     return size
104
105 def convert_bytes(size, unit=None):
106     if unit == "KB":
107         return print('File size: ' + str(round(size / 1024, 3)) + ' KB')
108     elif unit == "MB":
109         return print('File size: ' + str(round(size / (1024 * 1024), 3)) + ' MB')
110     else:
111         return print('File size: ' + str(size) + ' bytes')
112
113 # Compute model size
114 convert_bytes(get_file_size(name_model), "MB")
```

Per calcolare le prestazioni con i dati di test, generalmente, si usa la funzione `model.evaluate()`, che confronta le etichette stimate dalla `model.predict()` con quelle reali presenti nel dataset. Siccome, però, il FloodNet dataset non presenta etichette per i dati di test, queste sono state stimate visivamente e inserite nel vettore `y_test` che viene, tramite un ciclo for, confrontato con quello contenente le etichette predette, ottenendo soltanto una stima dell'evaluation, motivo per cui i valori sono più bassi rispetto all'accuratezza per i dati di test e validation.

```
120 test = test_datagen.flow_from_directory(  
121     "/content/drive/MyDrive/FloodNet Challenge @ EARTHVISION 2021 - Track 1/Test",  
122     target_size=RESIZE,  
123     batch_size=BATCH_SIZE)  
124 #Caricamento modello salvato e calcolo delle previsioni  
125 model_loaded = load_model(name_model)  
126 pred_probs = model_loaded.predict(test)  
127 #Evaluation  
128 i = acc = loss = 0  
129 for i in range(448):  
130     if y_test[i] == np.argmax(pred_probs[i]):  
131         acc = acc+1  
132     else:  
133         loss = loss+1  
134     i = i+1  
135 test_acc = acc / 448  
136 test_loss = loss / 448  
137 print('test_acc:',test_acc )  
138 print('test_loss:',test_loss)
```

4.2 Testing su board

4.2.1 Conversione del modello

Come accennato nell'introduzione della tesi, le dimensioni di memoria a disposizione dei sistemi embedded su cui caricare i modelli sono molto limitate quindi, come evince dal confronto sulle backbones, il modello sviluppato con la EfficientNet v2 B0 è quello che "pesa" di meno e, di conseguenza, il più idoneo per l'utilizzo in una board. Tuttavia, 24 MB sono ancora troppi, quindi bisogna ricorrere alla conversione del modello dal formato h5 di Keras, in cui è stato salvato, in tflite, più piccolo ed efficiente. Sotto è riportato il codice utilizzato per la conversione tramite quantizzazione intera che converte i numeri a virgola mobile a 32 bit (come pesi e uscite di attivazione) nei numeri a virgola fissa a 8 bit più vicini.

Per prima cosa è stata definita la *representative_data_gen()*, funzione generatore che fornisce un insieme di dati di input sufficientemente grande da rappresentare valori tipici. Questo consente al convertitore di stimare un intervallo dinamico per tutti i dati variabili. La conversione viene eseguita dalla funzione *converter.convert()*, dove il convertitore è il TFLiteConverter, che prende in ingresso il modello in formato h5 e restituisce in uscita il corrispettivo in formato tflite.

```
140 # Convert using integer-only quantization
141 # Now you have an integer quantized model that uses integer data for
142 # the model's input and output tensors, so it's compatible with integer-only hardware
143
144 def representative_data_gen():
145     for _ in range(100):
146         img = next(train_generator)
147         yield [img[0]]
148
149 converter = tf.compat.v1.lite.TFLiteConverter.from_keras_model_file('/content/drive/MyDrive/Colab Notebooks/EfficientNet.h5')
150 converter.optimizations = [tf.lite.Optimize.DEFAULT] #converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
151 converter.representative_dataset = representative_data_gen
152 # Ensure that if any ops can't be quantized, the converter throws an error
153 converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
154 # Set the input and output tensors to uint8 (APIs added in r2.3)
155 converter.inference_input_type = tf.uint8
156 converter.inference_output_type = tf.uint8
157 tflite_model_quantInt = converter.convert()
158
159 interpreter = tf.lite.Interpreter(model_content=tflite_model_quantInt)
160 input_type = interpreter.get_input_details()[0]['dtype']
161 print('input: ', input_type)
162 output_type = interpreter.get_output_details()[0]['dtype']
163 print("output: ", output_type)
164
165 # Save the quantized int model:
166 import pathlib
167 tflite_models_dir = pathlib.Path("./")
168 tflite_models_dir.mkdir(exist_ok=True, parents=True)
169 tflite_model_quantInt_file = tflite_models_dir/"EfficientNet_model.tflite"
170 tflite_model_quantInt_file.write_bytes(tflite_model_quantInt)
171
172 # Estimate size
173 convert_bytes(get_file_size(tflite_model_quantInt_file), "MB")
```

4.2.2 OpenMV Cam H7 Plus

Il modello tflite, in uscita dalla conversione, ha una dimensione di circa 7 MB, siccome l'idea è quella di realizzare un sensore a basso costo per monitorare lo stato di emergenza delle alluvioni, questa è ancora troppo grande. Infatti, l'hardware da utilizzare sarebbe una board di tipo ST, cioè un microprocessore privo di sistema operativo che dispone di soli 2 MB di memoria. Tuttavia, è stato trovato un compromesso tra basso costo e sufficiente dimensione di memoria con la OpenMV Cam H7 Plus (Fig. 4.6), che è un ibrido tra le board ST e le Raspberry PC, le quali dispongono di sistema operativo Linux. L'H7 Plus, infatti, è costituita da un microprocessore di tipo ST e da un mini sistema operativo che permette di scrivere in MicroPython, ma rientra sempre nella fascia embedded low-cost in quanto non ha un sistema operativo vero e proprio. Essa dispone, come le altre board, di 2 MB di memoria interna ma ha anche la possibilità di inserire una micro SD fino ad un massimo di 32 MB permettendo, quindi, il caricamento di modelli più "pesanti". La cam è dotata, inoltre, di un sensore ottico OV5640 in grado di acquisire immagini 2592x1944 (5MP) con un obiettivo da 2,8 mm su un supporto per obiettivo M12 standard, che permette l'inserimento di lenti più specializzate.

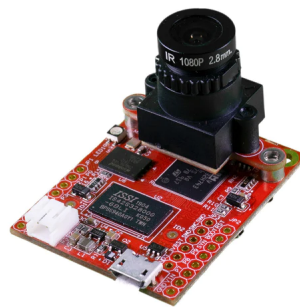


Fig. 4.6: OpenMV Cam H7 Plus. [17]

Trovato, quindi, l'hardware adatto su cui caricare il modello non resta che convertire il codice in MicroPython per renderlo eseguibile dalla OpenMV Cam ed effettuare i test finali su board. Quest'ultimi, purtroppo, non sono presenti in questo elaborato in quanto si ritiene che la versione di TensorFlow con cui è stato creato il modello (versione 2.8.2) non sia compatibile con quella dell'OpenMV IDE, tool utilizzato per programmare la Cam. Ciò ha reso impossibile l'utilizzo del modello sulla board e, di conseguenza, la possibilità di svolgere i vari test.

Capitolo 5

Conclusione e sviluppi futuri

La tesi si è proposta, quindi, di spiegare gli elementi fondamentali che costituiscono una rete neurale convoluzionale e fornirne un esempio applicativo di codice, sviluppato da ricercatori, per la classificazione di immagini alluvionate e non alluvionate. Viene proposta, poi, una possibile traduzione del codice per mezzo della libreria Keras, più semplice ed intuitiva, al fine di renderlo più gestibile e comprensibile. All'interno del capitolo Risultati sperimentali, è stata fornita una panoramica sulle caratteristiche principali di tre reti pre implementate scelte per le loro caratteristiche favorevoli in termini di accuratezza delle previsioni e memoria occupata dal modello e costruita una tabella riepilogativa delle prestazioni che hanno fornito con il FloodNet dataset. Dal confronto è risultato che il modello più idoneo per l'implementazione su un sistema embedded è quello ottenuto con la EfficientNet v2 B0 a causa della sua dimensione ridotta. Questo è stato, quindi, compreso al fine di caricarlo su board e, conseguentemente, scelto il miglior hardware da utilizzare.

Il test finale sarebbe stato quello sulla OpenMV Cam che, come spiegato, non è stato possibile effettuare. Si presume che il problema sia l'incompatibilità del converter introdotto nel modello tflite di TensorFlow 2.8.2 in uscita dalla conversione che non viene riconosciuto dall'OpenMV IDE. Non è stata trovata, però, alcuna documentazione ufficiale a riguardo, per cui, un possibile sviluppo futuro può essere l'approfondimento di questa incompatibilità con lo scopo di poter testare il modello direttamente dalla board. Ovviamente, non sarebbe comunque stato possibile effettuare le così dette "prove su campo", in quanto, queste prevedono la presenza di vere zone alluvionate ma, come test su board, si può pensare di valutare l'accuratezza del modello caricando nella micro SD della cam i dati di test. Un altro test su board potrebbe essere la valutazione degli FPS, frame per secondo, cioè il numero di fotogrammi per secondo che la cam riesce a catturare con il modello caricato. Per rendere la rete più precisa, un altro possibile sviluppo futuro potrebbe essere la traduzione in Keras della parte semi supervisionata, cioè quella che sfrutta anche i dati non etichettati. Essendo questi ultimi, presenti in

netta maggioranza all'interno del dataset, l'accuratezza della rete può migliorare ulteriormente nonostante le prestazioni ottenute soltanto con i dati etichettati siano comunque molto valide.

Bibliografia

- [1] C. Kyrkou and T. Theocharides, “Deep-learning-based aerial image classification for emergency response applications using unmanned aerial vehicles.” in *CVPR Workshops*, 2019, pp. 517–525.
- [2] “DJI Mavic Pro,” <https://cuneotrekking.com/recensione/recensione-del-dji-mavic-pro-il-drone-praticamente-perfetto/>, online; accessed 8 July 2022.
- [3] S. Khan, H. Rahmani, S. A. A. Shah, and M. Bennamoun, “A guide to convolutional neural networks for computer vision,” *Synthesis lectures on computer vision*, vol. 8, no. 1, pp. 1–207, 2018.
- [4] “Dense layer,” <https://analyticsindiamag.com/a-complete-understanding-of-dense-layers-in-neural-networks/>, online; accessed 5 July 2022.
- [5] S. Khose, A. Tiwari, and A. Ghosh, “Semi-supervised classification and segmentation on high resolution aerial images,” *arXiv preprint arXiv:2105.08655*, 2021.
- [6] “PyTorch,” <https://pytorch.org/>, online; accessed 1 July 2022.
- [7] “Repository GitHub Sahilkhose,” <https://github.com/sahilkhose/FloodNet>, online; accessed 1 July 2022.
- [8] “Keras,” <https://keras.io/>, online; accessed 30 June 2022.
- [9] “TensorFlow,” <https://www.tensorflow.org/>, online; accessed 1 July 2022.
- [10] “Pyimagesearch,” <https://pyimagesearch.com/2021/07/05/what-is-pytorch/>, online; accessed 1 July 2022.
- [11] “backbones,” <https://keras.io/api/applications/>, online; accessed 8 July 2022.
- [12] “ResNet,” <https://www.geeksforgeeks.org/residual-networks-resnet-deep-learning/>, online; accessed 7 July 2022.
- [13] “ResNet,” <https://www.analyticsvidhya.com/blog/2021/06/understanding-resnet-and-analyzing-various-models-on-the-cifar-10-dataset/>, online; accessed 7 July 2022.

- [14] “Xception,” <https://iq.opengenus.org/xception-model/>, online; accessed 5 July 2022.
- [15] “EfficientNet,” <https://towardsdatascience.com/efficientnet-scaling-of-convolutional-neural-networks-done-right-3fde32aef8ff>, online; accessed 5 July 2022.
- [16] “EfficientNet,” <https://ai.googleblog.com/2019/05/efficientnet-improving-accuracy-and.html>, online; accessed 5 July 2022.
- [17] “OpenMV Cam h7 plus,” <https://openmv.io/products/openmv-cam-h7-plus>, online; accessed 8 July 2022.