Report progetto Fondamenti di Data Science & Machine Learning: Yum or Yuck Butterfly Mimics challenge 2022

Andrea Terlizzi

13 novembre 2022

Indice

Abstract

Lo scopo del seguente elaborato è documentare lo sviluppo di un sistema di computer vision basato su reti neurali, volto alla classificazione di sei diverse specie di farfalle[1] diffuse in diverse zone del pianeta. Lo sviluppo del sistema è finalizzato alla partecipazione ad una challenge Kaggle, nota come Yum or Yuck Butterfly Mimics 2022 [55]. L'obbiettivo finale di quest'ultima è lo sviluppo di sistemi affidabili per il riconoscimento automatico delle farfalle, al fine di poter condurre studi dettagliati sulla distribuzione e sulle popolazioni di queste specie di insetti, alcune delle quali considerate a rischio. I risultati ottenuti con alcuni sistemi convoluzionali allo stato dell'arte sono incoraggianti nonostante le limitazioni dell'hardware a disposizione, raggiungendo circa il 94%/95% di accuratezza sul test set.

[1] In particolare, le specie in questione sono: Black Swallowtail (Papilio polyxenes), Farfalla Monarca (Danaus plexippus), Pipevine Swallowtail (Battus philenor), Spicebush Swallowtail (Papilio troilus), Eastern Tiger Swallowtail (Papilio glaucus) e Viceroy (Limenitis archippus).

Introduzione

Il documento è organizzato come segue: nel capitolo 1 verrà presentata la challenge, il dataset, le fasi di preprocessing e le tecniche di data augmentation applicate su di esso.

Il capitolo 2 sarà dedicato alle tenologie e tecniche di deep learning utilizzate per lo sviluppo del sistema, dando particolare rilievo ai modelli utilizzati.

Per concludere, nel capitolo 3 verranno descritti gli esperimenti effettuati con i sovramenzionati modelli, presentando i relativi risultati ottenuti.

Capitolo 1

Challenge, Dataset e Preprocessing

La challenge alla cui partecipazione il progetto è finalizzato è la "Yum or Yuck Butterfly Mimics 2022 Challenge" [55]. L'obiettivo di quest'ultima è la creazione di un modello di deep learning che riesca ad associare ad un immagine la corrispondente specie di farfalla, scelta tra le sei: Black Swallowtail (Papilio polyxenes), Farfalla Monarca (Danaus plexippus), Pipevine Swallowtail (Battus philenor), Spicebush Swallowtail (Papilio troilus), Eastern Tiger Swallowtail (Papilio glaucus) e Viceroy (Limenitis archippus).

1.1 Introduzione e obiettivi

Il nome della challenge fa riferimento al contrasto tra alcune specie di farfalle velenose (ad esempio la Farfalla Monarca), che risultando disgustose per gli uccelli predatori vengono da questi evitate ("yuck"), ed altre specie che al contrario non risultano velenose ("yum") ma imitano con la loro colorazione

Figura 1.1: Immagini esemplificative di (da sinistra): Pipevine Swallowtail (Battus philenor), Spicebush Swallowtail (Papilio troilus), Eastern Tiger Swallowtail (Papilio glaucus) e Black Swallowtail (Papilio polyxenes).

Figura 1.2: Immagini esemplificative di (da sinistra): Farfalla Monarca (Danaus plexippus) e Viceroy (Limenitis archippus).

altre specie velenose per scoraggiare i predatori (ad esempio la Spicebush Swallowtail che imita la colorazione della Pipevine).

In altri casi, alcune specie velenose assumono evolutivamente colorazioni simili a quelle di altre specie velenose (ad esempio la Viceroy che imita la farfalla Monarca), portando ad un vantaggio reciproco, in quanto i predatori necessitano di imparare un solo "pattern" di avvertimento anziché uno per ogni specie.

I comportamenti di queste specie, noti rispettivamente in letteratura come *mimetismo batesiano* [25] e *mülleriano* [39], rendono la loro classificazione visiva un compito particolarmente complesso, adatto a modelli di apprendimento automatico profondo.

L'importanza di questo compito risiede tuttavia nell'agevolazione che un modello performante potrebbe fornire a studi sulla distribuzione delle popolazioni di queste specie, alcune delle quali (ad esempio la Farfalla Monarca) recentemente dichiarate a rischio dalla International Union for Conservation of Nature (IUCN), così come sottolineato dai creatori della challenge stessa [55].

La partecipazione alla challenge avviene attraverso la sottomissione di un file contenente le predizioni effettuate dal modello sul test set fornito, mentre la valutazione delle performance verrà effettuata utilizzando una metrica molto nota nel machine learning, l'$F1$-score[1].

[1] Questa metrica viene calcolata a partire dalla Precision e dalla Recall, come F1-Score := $(2 \cdot \text{Precision} \cdot \text{Recall})/(\text{Precision} + \text{Recall})$. Queste ultime sono altre due metriche che in un problema di classificazione binaria indicano rispettivamente il tasso di istanze positive classificate correttamente in rapporto alla totalità delle istanze classificate come positive e alla totalità di effettive istanze positive, rispettivamente. Nei problemi di classificazione non binari, queste metriche vengono calcolate su ogni classe e poi mediate per ottenere un valore globale.

Tabella 1.1: Tabella esemplificativa di 5 righe estratte dal file images.csv.

Image	Name	Stage	Side
ggc1e08cbc	monarch	adult	ventral
gh6adf74a4	pipevine	adult	dorsal
ghac992a45	tiger	adult	dorsal
gi53e890fc	pipevine	adult	both

1.2 Struttura e Contenuto del Dataset

Al momento del download, il dataset fornito dai creatori della challenge è organizzato in quattro sezioni principali:

- una directory `images`, contenente 853 immagini RGB 225x225px in formato JPG, ritraenti le varie specie di farfalle, etichettate e da utilizzare per il training e la validazione del modello da sviluppare;
- una directory `images_holdouts`, contenente 356 immagini RGB 224x224px in formato JPG, non etichettate e da utilizzare per il testing finale del modello e la sottomissione delle predizioni effettuate da quest'ultimo;
- un file `images.csv` (tabella 1.1), i cui campi rappresentano metadati relativi alle immagini della directory `images`, in particolare:
  - `image`: nome alfanumerico del file;
  - `name`: label della specie ritratta dall'imagine (monarch, tiger, black, pipevine, viceroy, spicebush);
  - `stage`: label aggiuntiva rappresentante la fase della vita dell'esemplare ritratto (larva, pupa, adulto);
  - `side`: label aggiuntiva indicante l'ala dell'esemplare ritratto nell'immagine (sinistra, destra o entrambe);
- un file `images_holdouts.csv` contenente un unico campo `image` che indica i filename delle immagini presenti nella cartella `images_holdouts`.

Oltre a queste quattro sezioni principali, con il dataset sono inclusi anche:

- due file descrittivi `2022-Dataset-of-Butterfly-Mimics.pdf` e `YOYMimics-2022-dataset.pdf`;

Tabella 1.2: Tabella esemplificativa di 5 righe estratte dal file images_holdouts.csv.

Image
gi0c3c7e10
gl063e0456
gmc0c53c37
goabc6e644

- una tabella descrittiva delle sei specie di farfalle nel file `butterfly_info.csv` (mostrate nella tabella 1.3 nella pagina successiva);
- un file esemplificativo `sample_submission.csv` (1.4 nella pagina seguente) che mostra il formato di un file per la sottomissione, avente campi:
  - `image`: filename dell'immagine;
  - `name`: label predetta dal modello sviluppato.

1.3 Preprocessing ed analisi dei dati

Nelle sezioni di seguito verrà analizzata la composizione del dataset riportando le operazioni di preprocessing effettuate ai fini di adattarlo al nostro caso d'uso.

1.3.1 Composizione del dataset

Da una prima analisi del dataset fornito dai creatori della challenge emergono due principali problematiche:

- le sue dimensioni sono relativamente ridotte per poter ottenere performance soddisfacenti durante l'allenamento di un modello di apprendimento profondo;
- le classi presenti risultano alquanto sbilanciate, come è possibile osservare dal barplot che mostra la distribuzione delle label (figura 1.3 a pagina 7).

Entrambe queste problematiche possono essere indirizzate utilizzando tecniche di data augmentation che, come vedremo nella sezione 1.3.2 a pagina 8, sono state lo step principale della fase di preprocessing.

Tabella 1.3: Tabella informativa sulle specie delle farfalle.

Name	Common Name	Species	Yum?	Real?	Note
black	Black Swallowtail	Papilio polyxenes	yum	mimic	mimics pipevine
monarch	Monarch	Danaus plexippus	yuck	real	cardiac glycoside toxins
pipevine	Pipevine Swallowtail	Battus philenor	yuck	real	sequesters aristolochic acid
spicebush	Spicebush Swallowtail	Papilio troilus	yum	mimic	mimics pipevine
tiger	Eastern Tiger Swallowtail	Papilio glaucus	yum	mimic	females may mimic pipevine
viceroy	Viceroy	Limenitis archippus	yuck	mimic	sequesters salicylic acid

Tabella 1.4: Tabella esemplificativa che mostra il formato di sottomissione delle predizioni.

Image	Name
gi0c3c7e10	monarch
gl063e0456	pipevine
gmc0c53c37	black
goabc6e644	black

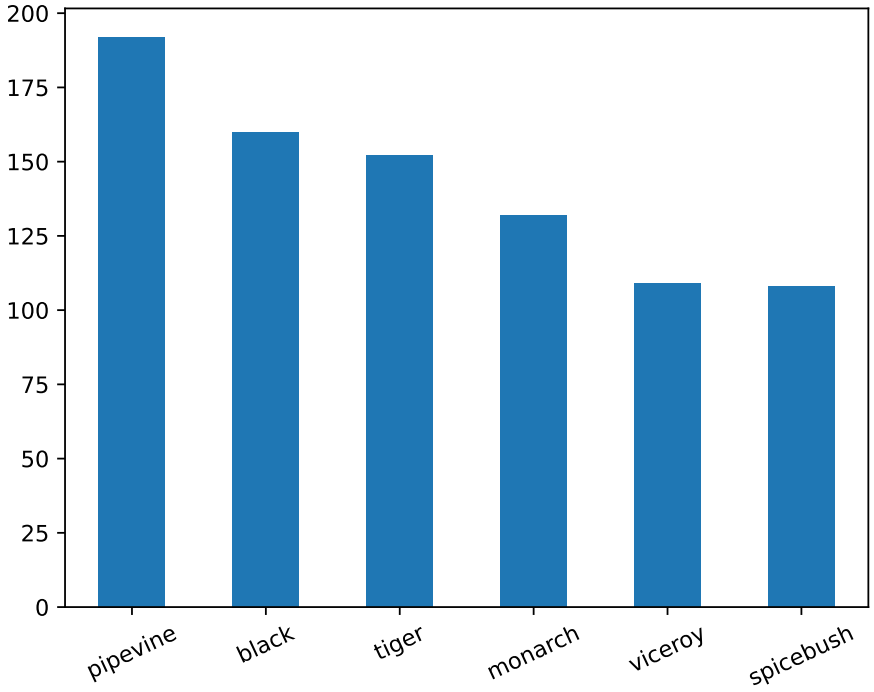

Figura 1.3: Grafico a barre rappresentante le occorrenze delle specie di farfalle nel dataset.

1.3.2 Preprocessing e data augmentation

Essendo il dataset già ben pulito e formattato, ed essendo la challenge finalizzata allo sviluppo di un modello di deep learning, in cui l'estrazione delle caratteristiche avviene in modo automatico, il preprocessing effettuato sui dati è stato relativamente semplice.

Come approfondiremo in seguito nella sezione 2.3 a pagina 29 dedicata alle tecnologie, tutte le operazioni che hanno caratterizzato questa fase sono state effettuate mediante le API messe a disposizione da:

- la nota libreria di data analysis Pandas [33];
- la famosa libreria per il machine learning Scikit-Learn [40];
- la sezione `tensorflow.data` della libreria open-source TensorFlow [9] per lo sviluppo di modelli di deep learning.

Lo script di preprocessing opera approssimativamente come segue, da un punto di vista ad alto livello:

1. legge i file `images.csv` ed `images_holdouts.csv` trasformandoli in istanze di `pandas.DataFrame`;
2. effettua, usando le API di Scikit-Learn, lo split casuale del `DataFrame` relativo ad `images.csv` in:

   train set : le immagini corrispondenti verranno utilizzate per allenare i modelli sviluppati (64% del dataset);

   validation set : le immagini corrispondenti saranno utilizzate per valutare l'andamento del training epoca dopo epoca, e per effettuare il tuning degli iper-parametri (16% del dataset);

   evaluation set : le immagini corrispondenti verranno utilizzate come test set etichettato per valutare le performance finali del modello realizzato su dati mai visti e sui quali gli iper-parametri non sono stati adattati (20% del dataset);

3. sfruttando le API di `tensorflow.data`, trasforma i `DataFrame` in istanze di `tensorflow.data.Dataset`, aggiungendo ad ogni istanza due colonne:

   - una contenente il tensore con shape $(225, 225, 3)$ dell'immagine RGB corrispondente a tale istanza (letto usando le API di `tensorflow.io`), con valori normalizzati tra 0 ed 1;

- una contenente la corrispettiva label (eccetto nel caso delle immagini del **test set** per la sottomissione, ove le label non sono note);

4. performa la data augmentation sul **train set**, replicando le immagini presenti in esso un numero di volte definito dalla costante `AUGMENTATION_RATIO`, e applicando a ciascuna di esse una serie di trasformazioni casuali con le API di `tensorflow.image`:

   - flip random dell'immagine rispetto all'asse y;
   - flip random dell'immagine rispetto all'asse x;
   - modifica random della luminosità di ogni pixel per un Δ che varia in un range definito dalla costante `MAX_DELTA_BRIGHTNESS`;
   - modifica random della saturazione di ogni pixel per un Δ che varia in un range definito dalle costanti `RANDOM_SATURATION_LOWER` e `RANDOM_SATURATION_UPPER`;

5. salva le istanze di `tensorflow.data.Dataset` risultanti nelle apposite directory.

Capitolo 2

Tecniche e tecnologie utilizzate

Prima di iniziare la descrizione del modello e degli esperimenti realizzati, verrà presentata una panoramica teorica delle tecniche applicate e una breve descrizione delle tecnologie utilizzate per implementarle (già accennate nel capitolo 1 a pagina 2).

2.1 Reti neurali

Le reti neurali sono un modello di calcolo ispirato all'architettura del cervello umano. Sono composte da una serie di nodi (o neuroni) interconnessi tra loro che elaborano e propagano le informazioni in modo simile al cervello. Le reti neurali possono essere utilizzate per risolvere una vasta gamma di problemi, tra cui il riconoscimento dei pattern, la previsione delle serie temporali e la classificazione delle immagini.

2.1.1 Percettrone e neuroni artificiali

L'unità base di calcolo nelle reti neurali è il *neurone artificiale* (2.1 nella pagina successiva), originariamente ideato da McCulloc e Pitts [29], e successivamente perfezionato da Rosenblatt [37], Minsky e Papert [31] con il cosiddetto *percettrone*.

Esso prende in input un vettore $x = [x_1, x_2, \ldots, x_n]$, ne effettua la somma pesata aggiungendo un *bias* b:

$$z := \sum_{i=1}^{n} w_i x_i + b,$$

ed applicandovi successivamente una funzione d'attivazione φ, producendo l'output:

$$o(x) := \varphi(z) = \varphi\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

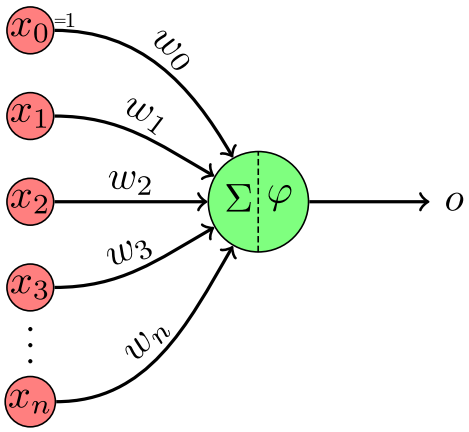

Figura 2.1: Schema illustrativo del neurone computazionale [7].

Nel modello originale di McCulloc e Pitts, la funzione d'attivazione utilizzata era una step-function binaria come la seguente:

$$\varphi(z) := \begin{cases} 1, & z \geq 0, \\ 0, & z < 0 \end{cases},$$

ma successivamente sono state esplorate altre funzioni d'attivazione più flessibili, e che riflettono maggiormente il comportamento dei neuroni biologici, come la sigmoide σ o la tangente iperbolica tanh:

$$\sigma(z) := \frac{1}{1 + e^{-x}},$$

$$\tanh(z) := \frac{e^x - e^{-x}}{e^x + e^{-x}},$$

oppure altre che lasciano quasi inalterato l'output della somma, come la ReLU o la LeakyReLU [35]:

$$\mathrm{ReLU}(z) := \begin{cases} z, & z > 0 \\ 0, & z \leq 0 \end{cases}$$

$$\mathrm{LeakyReLU}(z) := \begin{cases} z, & z > 0 \\ \alpha z, & z \leq 0 \end{cases} \text{ (con } \alpha \text{ molto piccolo).}$$

Supponiamo di avere un training set:

$$D = \left\{ \left(x^{(1)}, y^{(1)}\right), \left(x^{(2)}, y^{(2)}\right), \ldots, \left(x^{(m)}, y^{(m)}\right) \right\},$$

dove ciascun $y^{(i)}$ è un'etichetta che indica il valore $f(x^{(i)})$ di una certa funzione target f da approssimare (ad esempio in un problema di classificazione binaria, $y^{(i)} \in \{0, 1\}$ indica la classe a cui appartiene l'istanza). Il modello originale di McCulloc e Pitts, ma anche in altri modelli successivi simili, come ADALINE [54], utilizzano una tecnica di addestramento basata su tre step molto semplici:

1. inizializzare i pesi $w_1, w_2, ..., w_n$ e il bias b in modo casuale;
2. effettuare la predizione $\hat{y}^{(i)} = \varphi(z^{(i)})$ su ciascuna istanza di training $x^{(i)}$;
3. correggere i pesi $w_1, w_2, ..., w_n$ ed il bias b in base all'errore effettuato, seguendo una *delta-rule* (quest'operazione può essere effettuata sia dopo il calcolo dell'errore su una singola istanza che su tutte le istanze).

In particolare, nel percettrone di Rosenblutt l'aggiornamento avviene dopo ciascuna predizione su un'istanza $x^{(j)}$, secondo la seguente *delta-rule*:

$$w_i \leftarrow w_i + \eta \cdot \left(y^{(j)} - \hat{y}^{(j)}\right) \cdot x_i^{(j)},$$

dove $\hat{y}^{(j)} = o(x^{(j)})$ è l'output del percettrone sull'istanza $x^{(j)}$ e η è un iper-parametro noto come *learning rate*, che definisce quanto l'errore effettuato su ogni istanza influenza la successiva modifica dei pesi. Situazione identica nell'update del bias b.

Come dimostrato dal *Teorema di Convergenza del Percettrone* [31], questo modello computazionale può approssimare qualsiasi funzione di classificazione in uno spazio linearmente separabile (ad esempio le funzioni logiche AND, OR e NOT), ma non può in alcun modo apprendere una funzione di classificazione in spazi non linearmente separabili (come la funzione XOR 2.2 nella pagina seguente).

Allo scopo di approssimare questo tipo di funzioni più complesso, sono state sviluppate le cosiddette Reti Neurali Artificiali (ANN[1]), composte da uno o più strati posti a cascata l'uno dopo l'altro, ciascuno composto da più neuroni che lavorano parallelamente. L'esempio più semplice è il Multi-Layer Perceptron (MLP), costituito da 3 strati di percettroni disposti a cascata [36].

[1] Acronimo della traduzione inglese *Artificial Neural Network*.

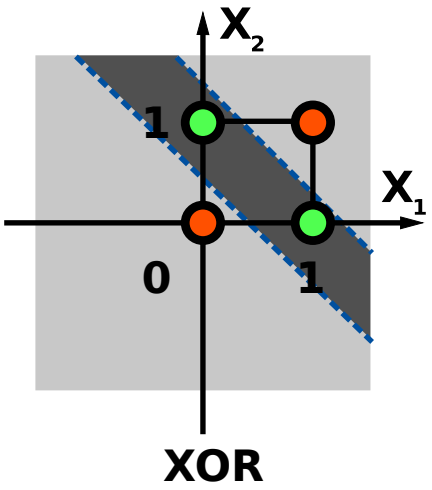

Figura 2.2: Prova visiva dell'impossibilità di trovare un singolo separatore lineare che approssimi lo XOR. [6].

2.1.2 Reti neurali profonde (DNN)

Una Rete Neurale Profonda (DNN[2]) è una rete neurale artificiale che presenta più di uno strato nascosto tra gli strati di input e di output per classificare un pattern di input o, più in generale, imparare ad approssimare una funzione target f. In presenza di un problema di classificazione, come quello affrontato in questo elaborato, si cerca di discriminare un pattern x in base a una caratteristica y, che può assumere valore tra Q categorie (dette appunto classi), ed il compito della DNN stima le probabilità di ciascuna classe $p_j, j \in \{1, ..., Q\}$ data l'istanza x. Le caratteristiche in ingresso a una DNN non richiedono un'attenzione particolare al preprocessing, ed anzi in letteratura è stato più volte osservato come il dare in input ad una rete dati quanto più grezzi possibile risulti in performance migliori e maggiore robustezza del modello, piuttosto che dare in input alla rete dati più raffinati e/o già puliti dal rumore. La figura 2.3 nella pagina successiva mostra un esempio di rete neurale profonda con 5 neuroni di output.

Più formalmente, una DNN feed-forward con H layer, matrici dei pesi $W_1, ..., W_H$, vettori dei bias $b_1, ..., b_H$ e funzioni d'attivazione $f_1, ..., f_H$ calcola una funzione non lineare [21]:

$$g_{W,b}(x) := a_H(x) \ ,$$

[2]Acronimo della traduzione inglese *Deep Neural Network*.

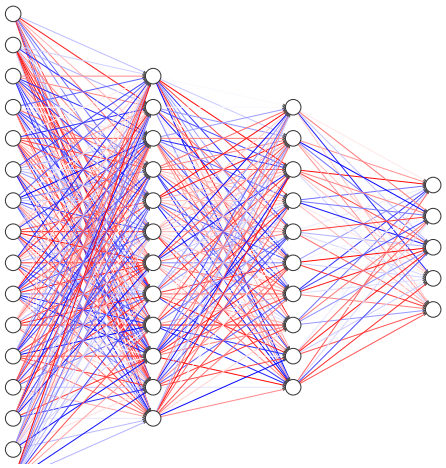

Figura 2.3: Rete neurale profonda con 5 neuroni di output.

dove, per ogni $h = 1, 2, ..., H$:

$$a_h(x) = f_h((a_{h-1}(x))^T \cdot W_h + b_h), \text{ e:}$$

$$a_0(x) = x$$

Nei task di classificazione, la funzione di attivazione dell'ultimo layer è spesso una funzione *softmax* σ, che genera una probabilità $\hat{p}_j$ per ogni classe j. La classe con la probabilità più alta viene poi considerata quella predetta:

$$q = \arg\max_j \hat{p}_j, \text{ dove:}$$

$$\hat{p}_j = f_H(x)_j = \sigma(x)_j := \frac{e^{x_j}}{\sum_{i=1}^{Q} e^{x_i}}$$

Training di una DNN In generale, le DNN vengono addestrate a minimizzare una *loss function* $L(W, b)$. In problemi di classificazione multi-classe come quello in oggetto, la loss function più utilizzata è la *categorical cross-entropy*:

$$L = \sum_{i=1}^{Q} p_j \log \hat{p}_j,$$

dove p_j è la probabilità target per la classe j (di norma 1 per la classe di appartenenza dell'istanza, 0 altrimenti) e $\hat{p}_j$ è la probabilità prevista dalla rete per la classe j.

Ad ogni ciclo di addestramento (chiamati *epoche*), i pesi W e i bias b vengono aggiornati in base al loro gradiente:

$$\nabla_{W,b}(L) = \left[\frac{\partial L}{\partial b}, \frac{\partial L}{\partial W}\right]^T$$

Mentre i gradienti vengono calcolati con l'algoritmo di backpropagation [38], la regola di aggiornamento del peso basato sul loro valore viene definita da un algoritmo di ottimizzazione, che in generale è considerato un iperparametro da regolare. Due notevoli e popolari esempi di algoritmi di ottimizzazione per aggiornare i parametri della rete sono Adam [23] e Adadelta [56], utilizzati anche negli esperimenti a cui questo elaborato fa riferimento.

Adam L'idea alla base dell'algoritmo Adam è quella di effettuare l'update dei pesi normalizzando il gradiente delle derivate prime (primo livello) per il gradiente delle derivate seconde (secondo livello), cercando di ridurre anche l'overfitting riducendo di epoca in epoca la modifica ai parametri.

Siano m e v i due vettori di primo e secondo momento (rispettivamente riferiti al gradiente di primo grado e secondo grado), entrambi inizializzati a $\underline{0}$, e con pesi β_1 e β_2; sia η il learning rate. Allora la t-esima iterazione (epoca) dell'algoritmo Adam per l'ottimizzazione di un parametro θ opera come segue:

1. computa il gradiente di primo e secondo livello rispetto a θ:

$$g_\theta = \nabla_\theta f(\theta);$$

2. aggiorna i vettori di primo e secondo momento:

$$m \leftarrow \beta_1 \cdot m + (1 - \beta_1) \cdot g_\theta$$

$$v \leftarrow \beta_2 \cdot v + (1 - \beta_2) \cdot g_\theta^2$$

3. normalizza i vettori di primo e secondo momento in base al numero di iterazione:

$$\hat{m} \leftarrow m/(1 - \beta_1^t)$$

$$\hat{v} \leftarrow v/(1 - \beta_2^t)$$

4. aggiorna i parametri θ sulla base dei momenti normalizzati (più una costante ϵ molto piccola che garantisce di non avere divisioni per 0):

$$\theta \leftarrow \theta - \eta \cdot \hat{m}/(\sqrt{\hat{v}} + \epsilon)$$

Adadelta L'idea alla base di Adadelta è che il modello possa imparare ad adattarsi alla variazione del learning rate nel tempo. Esso è un'estensione di Adagrad [5] che tenta di correggere il problema di ritardo nell'apprendimento che quest'algoritmo presenta, utilizzando la media mobile del quadrato dei gradi di variazione dei pesi per calcolare l'aggiornamento degli stessi e modificare il learning rate adattivamente. Rispetto ad Adagrad infatti, Adadelta risulta meno sensibile alle variazioni nella direzione del gradiente, non richiedendo inoltre un'impostazione iniziale del learning rate.

Indichiamo con $E[x^2]_t$ la *media mobile* della variabile x^2 all'iterazione t, calcolata come:

$$E\left[x^2\right]_t = \rho \cdot E\left[x^2\right]_{t-1} + (1-\rho) \cdot x^2,$$

dove ρ è un *tasso di decadimento* dell'update definito a priori (nel paper originale [56], viene consigliato di porre $\rho = 0.95$), simile al *momento* presente in Adam [23] ed Adagrad [5]. Denotiamo poi con $\text{RMS}[x]_t$ il *root mean square* della variabile x all'iterazione t, calcolato come:

$$RMS\left[x\right]_t = \sqrt{E\left[x^2\right]_t + \epsilon},$$

dove ϵ è un valore costante molto piccolo (e.g. epsilon macchina) che evita valori dell'RMS pari a 0.

L'algoritmo Adadelta allora, nell'ottimizzazione di un parametro θ, dopo aver inizializzato le medie mobili di gradiente e quadrato della variazione a 0:

$$E\left[g^2\right]_0, E\left[\Delta\theta^2\right]_0 = 0,$$

opera come segue:

1. computa il gradiente rispetto a θ_{t-1}:

$$g_t = \nabla_{\theta_{t-1}} f(\theta_{t-1});$$

2. accumula il valore del gradiente (per adattare l'aggiornamento dei pesi all'andamento del valore del gradiente) computando la media mobile di quello di secondo livello:

$$E\left[g^2\right]_t = \rho \cdot E\left[g^2\right]_{t-1} + (1-\rho) \cdot g^2;$$

3. computa l'update del peso θ_{t-1} usando i root mean square di gradiente e update:

$$\Delta\theta_t = -\frac{\text{RMS}\left[\Delta\theta\right]_{t-1}}{\text{RMS}\left[g\right]_t} \cdot g_t;$$

4. accumula l'update (per adattare l'aggiornamento dei pesi all'andamento degli update precedenti) computando la media mobile del quadrato della variazione:

$$E\left[\Delta\theta^2\right]_t = \rho \cdot E\left[\Delta\theta^2\right]_{t-1} + (1-\rho) \cdot \Delta\theta^2;$$

5. aggiorna il valore del parametro:

$$\theta_t = \theta_{t-1} + \Delta\theta_t.$$

2.1.3 Regolarizzazione

La regolarizzazione è un processo che consente di migliorare l'apprendimento e la capacità di generalizzazione dei modelli di apprendimento automatico, spesso applicando una penalità alla funzione di costo che cresce con la complessità del modello, al fine di prevenire l'*overfitting*. Informalmente, questa condizione indica un modello che si è adattato eccessivamente ai dati di addestramento, rendendolo incapace di generalizzare ciò che ha appreso a dati reali. Più formalmente, sia $h : A \mapsto B$ un modello che approssima una funzione $f : A \mapsto B$, allenandosi su un training set X; indicato con $E_X(h)$ l'errore di h sul training set e con $E(h)$ il suo errore di generalizzazione, h viene detto in *overfitting* $\iff$:

$$\exists \text{ un altro modello } h' : A \mapsto B \text{ t.c. } E_X(h') \geq E_X(h) \text{ ma } E(h') < E(h)$$

Nel machine learning, ogni tipologia di modello ha le sue forme di regolarizzazione; nell'ambito delle reti neurali artificiali essa consiste nell'applicare una penalità alla loss function in base ai pesi o ai valori delle unità nascoste. A differenza dei modelli classici, le reti neurali hanno una struttura iperparametrica, cioè il numero di parametri è superiore al numero di osservazioni; pertanto, se non si applicasse regolarizzazione, una sovrapposizione del modello alle osservazioni risulterebbe estremamente probabile, generando overfitting. Allo scopo di evitare questa condizione, sono state sviluppate molte differenti tecniche di regolarizzazione nelle reti neurali, come il dropout [14], la regolarizzazione L_1, L_2 [49] [15] [48] o ElasticNet [59].

Regolarizzazione L_1 e L_2 La regolarizzazione L_1 e L_2 sono tecniche che consistono nel penalizzare la loss function con un termine che è proporzionale alla norma L_1 o L_2 dei pesi. Queste tecniche sono state introdotte da Tikhonov nel 1963 [49], da Hoerl e Kennard nel 1970 [15] e da Tibshirani nel 1996 [48].

In un modello con vettore dei parametri w, la regolarizzazione L_1 ed L_2 vengono applicate aggiungendo un termine di regolarizzazione alla loss function $L(w)$ che dipende dalla norma di w, rispettivamente:

$$L(w) + \lambda \|w\|_1 ,$$

$$L(w) + \lambda \|w\|_2^2 ,$$

dove λ è un parametro che controlla la pesantezza della penalizzazione.

In questo lavoro le regolarizzazione L_1 ed L_2 sono state applicate per ridurre l'overfitting sugli ultimi layer dell'architettura proposta.

ElasticNet L'ElasticNet è una tecnica di regolarizzazione che consiste nel penalizzare la loss function con un termine che è proporzionale alla somma della norma L_1 e L_2 dei pesi. Questa tecnica è stata introdotta da Zou e Hastie nel 2005 [59]. Il termine di penalizzazione è il seguente:

$$\lambda \left(\rho \|w\|_1 + \frac{1-\rho}{2} \|w\|_2^2 \right) ,$$

dove λ è un parametro che controlla la pesantezza della penalizzazione e ρ è un iperparametro che determina il peso dei termini L_1 ed L_2 l'uno rispetto all'altro.

Dropout Il dropout è una tecnica di regolarizzazione introdotta da Hinton nel 2012 [14]. Essa consiste nel disattivare casualmente (con una probabilità ρ fissata) alcune unità nascoste durante il training, in modo che il modello non si sovrapponga alle osservazioni, ed i neuroni non si attivino eccessivamente. I valori d'attivazione delle unità disattivate vengono inoltre distribuiti equamente tra le unità attive rimanenti, lasciando così inalterata la norma del vettore d'attivazione durante il training, ma non la sua distribuzione.

Come nel caso della regolarizzazione L_1 ed L_2, in questo il dropout è stato utilizzato per ridurre la possibilità di overfitting e permettere ai modelli addestrati di acquisire maggiori capacità di generalizzazione.

2.1.4 Reti neurali convoluzionali

Le reti neurali convoluzionali (CNN) sono una classe di reti neurali artificiali diventata dominante in diversi compiti di computer vision ed elaborazione di segnali, suscitando tutt'ora interesse in un ampia gamma di domini. Questo tipo di rete è in grado di apprendere automaticamente e in modo adattivo gerarchie di caratteristiche di alto livello attraverso strati di filtri dinamici di piccolo peso che le individuano o evidenziano efficacemente. Solitamente, le architetture delle CNN sono composte da più blocchi costruttivi, come gli strati di convoluzione, normalizzazione e pooling (figura 2.4).

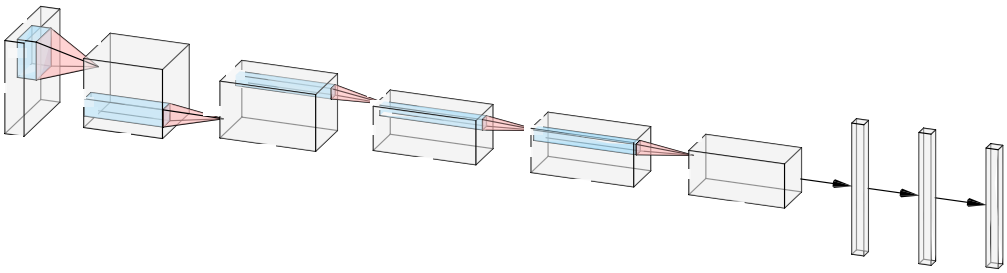

Figura 2.4: Diagramma rappresentativo di una rete convoluzionale che estrae gerarchicamente feature da un'immagine di input.

Layer Convoluzionale Un layer convoluzionale l_m riceve in input un segnale $\boldsymbol{X}^{(m-1)}$ con K_m canali (ad esempio un'immagine a 3 canali RGB grezza o l'output dell'$(m-1)$-esimo layer), e computa in output un nuovo segnale $\boldsymbol{X}^{(m)}$ composto da O_m canali. L'output di ogni canale è noto come **feature map**, e viene calcolato come [3]:

$$\boldsymbol{X}_o^{(m)} = g_m \left(\sum_k \boldsymbol{W}_{o,k}^{(m)} * \boldsymbol{X}_k^{(m-1)} + b_o^{(m)} \right)$$

dove $*$ denota l'operazione di *convoluzione* 2D:

$$\boldsymbol{W}_{o,k} * \boldsymbol{X}_k[s,t] = \sum_{p,q} \boldsymbol{X}_k[s+p, t+q] \boldsymbol{W}_{ok}[P_m - 1 - p, Q_m - 1 - q]$$

dove $\boldsymbol{W}_{o,k}^{(m)} \in \mathbb{R}^{P_m \times Q_m}$ è una matrice nota come *kernel convoluzionale*, $b_o^{(m)} \in \mathbb{R}$ è un valore di bias g_m è la funzione d'attivazione applicata al risultato finale. Il kernel convoluzionale $\boldsymbol{W}_{o,k}^{(m)}$ agisce come contenitore di parametri addestrabili di un filtro che il layer può utilizzare per rilevare o evidenziare

alcune caratteristiche presenti in gruppi di pixel dell'immagine in ingresso. I pesi e la conseguente feature rilevata/evidenziata dal filtro vengono appresi durante il processo di addestramento, proprio come le matrici dei pesi delle reti neurali fully-connected che abbiamo analizzato fin'ora.

Layer di Batch Normalization La *batch normalization* è una tecnica che mira ad accelerare l'addestramento delle reti neurali profonde riducendo lo shifting delle covarianze interne ai dati. Ciò avviene attraverso un'operazione di normalizzazione che fissa le medie e le varianze degli input dei layer a livello di batch[3]. Quest'operazione ha anche una serie di effetti "collaterali" estremamente vantaggiosi:

- la forte riduzione della dipendenza dei gradienti dalla scala dei parametri o dai loro valori iniziali, che consente di utilizzare learning rate molto più elevati senza il rischio di divergenza o di esplosione del gradiente [34];
- la regolarizzazione dei valori di attivazione delle unità del modello, che riduce la necessità di dropout [14] o di altre tecniche di regolarizzazione pesanti.

La batch normalization su un batch $\mathcal{B} = \{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$ con m instanze [20] è applicata come segue:

1. computa media e varianza del batch $\mathcal{B}$:

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$
$$\sigma^2_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)} - \mu_{\mathcal{B}}\right)^2$$

2. calcola il valore dell'istanza normalizzata $\hat{x}^{(i)}$ per ogni $i \in \{1, 2, ..., m\}$:

$$\hat{x}^{(i)} = \frac{x^{(i)} - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}},$$

   dove ϵ è un valore molto piccolo (e.g. epsilon macchina) aggiunto per prevenire le divisioni per zero.

[3]Sottoinsieme del dataset sulla base delle cui istanze viene computato il gradiente prima di aggiornare i pesi. Solitamente, per ridurre il costo computazionale del calcolo del gradiente e dell'algoritmo di backpropagation, il training set viene suddiviso in diversi batch, ed i pesi vengono aggiornati dopo aver computato l'output della rete su ciascun istanza presente in un singolo batch.

3. calcola il valore di output del layer per l'istanza $\hat{x}^{(i)}$:

$$y^{(i)} = \gamma\, \hat{x}^{(i)} + \beta = \mathrm{BN}_{\gamma,\beta}\left(x^{(i)}\right),$$

dove γ e β sono parametri addestrabili appresi durante la backpropagation.

Layer di Pooling I layer di pooling di una CNN implementano una trasformazione di riduzione della dimensionalità progettata per ridurre il numero di parametri addestrabili per gli strati successivi, consentendo loro di concentrarsi su aree più ampie dei modelli in ingresso, conservando allo stesso tempo la maggior parte delle informazioni in essi contenute.

Dato in input un segnale $\boldsymbol{X}^{(m-1)}$ (immagine grezza o output di un layer precedente) con K_m canali, un layer di pooling 2D con pool size $(P_m, Q_m) \in \mathbb{N}$ e stride $\alpha_m, \beta_m \in \mathbb{N}$ è un'operazione channel-wise (che opera singolarmente su ogni canale o) come quella che segue [3]:

$$\boldsymbol{X}_o^{(m)}[s,t] = \kappa \cdot \left(\sum_{p,q} \left(\boldsymbol{X}_o^{(m-1)}\left[s + \alpha_m \cdot p, t + \beta_m \cdot q\right] \right)^{\rho} \right)^{1/\rho}$$

dove $\kappa, \rho \in \mathbb{N}$ sono parametri fissati dipendentemente dal tipo di layer di pooling che viene applicato.

Si noti che utilizzare $P_m = Q_m = \alpha_m = \beta_m$ corrisponde a suddividere ogni canale del segnale di input in $P_m \times Q_m$ patches (regioni) senza intersezioni e rimpiazzare i valori in ciascuna regione con un singolo valore calcolato sulla base di ρ e κ.

Nei layer di *max pooling*, dove $(\rho = \infty, \kappa = 1)$, il valore di output della formula precedente è il massimo dei valori presenti nella patch:

$$\boldsymbol{X}_o^{(m)}[s,t] = \lim_{\rho \to \infty} \left(\sum_{p,q} \left(\boldsymbol{X}_o^{(m-1)}\left[s + \alpha_m \cdot p, t + \beta_m \cdot q\right] \right)^{\rho} \right)^{1/\rho} =$$

$$\max \left\{ \boldsymbol{X}_o^{(m-1)}\left[s + \alpha_m \cdot p, t + \beta_m \cdot q\right] : p \in \{1, 2, ..., P_m\}, q \in \{1, 2, ..., Q_m\} \right\}$$

Nei layer di *average pooling* invece, utilizzati anche in questo lavoro, dove $(\rho = 1, \kappa = 1/P_m\, Q_m)$, l'output è la media dei valori della patch:

$$\boldsymbol{X}_o^{(m)}[t] = \frac{1}{P_m\, Q_m} \cdot \sum_{p,q} \boldsymbol{X}_o^{(m-1)}\left[\alpha_m \cdot s + p, \beta_m \cdot t + q\right]$$

2.1.5 VGG

Le reti VGG sono una classe di reti neurali convoluzionali create da Simonyan e Zisserman nel 2014 [27]. L'idea alla base di queste architetture è quella di utilizzare filtri di piccola dimensione 3×3 per estrarre caratteristiche di basso livello, dalle cui feature map sono poi estratte caratteristiche di alto livello da ulteriori filtri di piccola dimensione, in un ottica gerarchica. Quest'architettura ha ottenuto il primo posto nella ImageNet Large Scale Visual Recognition Challenge (ILSVRC) del 2014 [19], ottenendo un errore del 6.8%, il miglior risultato di sempre all'epoca.

Architettura e blocchi costruttivi L'architettura della rete descritta dagli autori [27] ha due varianti: la VGG16 (figura 2.5 nella pagina successiva) e la VGG19 (figura 2.6 a pagina 24), entrambe addestrate su ImageNet [19] ottenendo performance simili, che presentano fondamentalmente gli stessi blocchi costruttivi di base:

- layer convoluzionali 2D con kernel 3×3 e funzione d'attivazione ReLU;
- layer di max-pooling che dimezzano la dimensione del segnale in input (facendo riferimento alla notazione usata in precedenza 2.1.4 nella pagina precedente, $P_m = Q_m = \alpha_m = \beta_m = 2$).

Entrambe le architetture sono suddivise in 6 sezioni consecutive distinte:

1. 2 layer di convoluzione con 64 kernel 3×3 ed un layer di max pooling che dimezza la dimensione dell'input;
2. 2 layer di convoluzione con 128 kernel 3×3 ed un layer di max pooling che dimezza la dimensione delle feature map ottenute dai layer precedenti;
3. 3 layer di convoluzione con 256 kernel 3×3 nella VGG16, 4 nella VGG19, ed un layer di max pooling che dimezza la dimensione delle feature map ottenute dai layer precedenti;
4. 3 layer di convoluzione con 512 kernel 3×3 nella VGG16, 4 nella VGG19, ed un layer di max pooling che dimezza la dimensione delle feature map ottenute dai layer precedenti;
5. 3 layer di convoluzione con 512 kernel 3×3 nella VGG16, 4 nella VGG19, ed un layer di max pooling che dimezza la dimensione delle feature map ottenute dai layer precedenti;

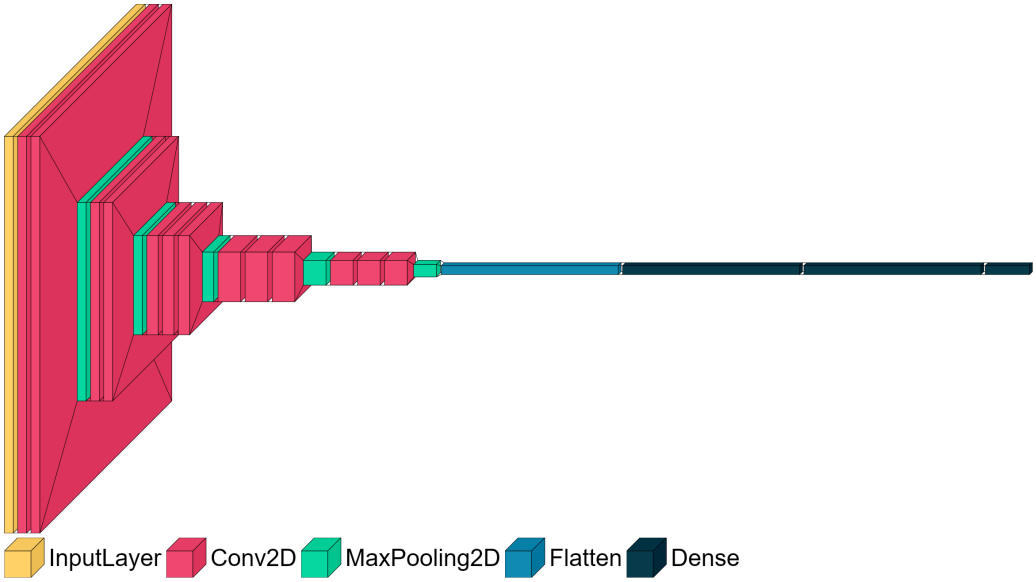

Figura 2.5: Architettura della rete VGG16 [8].

6. un layer di flattening (che appiattisce le feature map bidimensionali ottenute trasformandole in vettori monodimensionali) e 3 layer fully-connected da 4096, 4096 e 1000 neuroni l'uno (questa quantità nell'ultimo layer corrisponde al numero di classi di ImageNet [19]).

2.1.6 EfficientNetV2

Le EfficientNetV2 [44] sono una famiglia di reti neurali profonde progettate per migliorare la prestazione di apprendimento automatico profondo rispetto alle reti convoluzionali, sfruttando i potenti meccanismi di skip-connection e attention recentemente popolarizzati dalle architetture ResNet [11], GoogLeNet [43] e transformer [51].

Scaling neural architecture Le EfficientNetV2 sono state progettate utilizzando una tecnica di progettazione di reti profonde chiamata *scaling neural architecture*, basata sull'idea che il funzionamento di una rete neurale può essere migliorato aumentando o diminuendo la dimensione di alcuni dei suoi componenti. In taluni casi infatti, è possibile ottenere un miglioramento delle prestazioni di una rete neurale aumentando la dimensione di tutti i suoi componenti, mentre in altri casi, è possibile ottenere un miglioramento maggiore incrementando la scala di alcuni componenti della rete e diminuendo quella di altri.

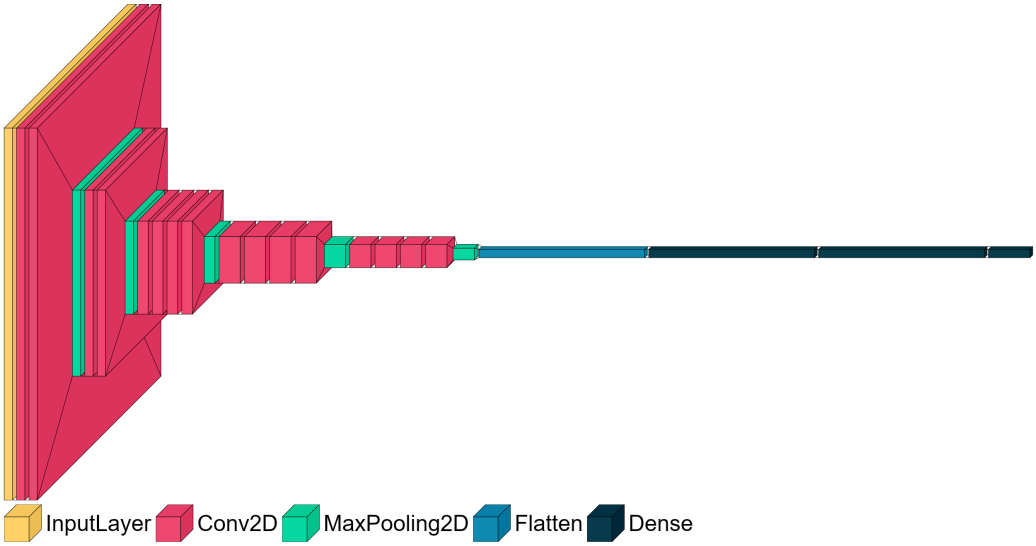

Figura 2.6: Architettura della rete VGG19 [8].

Le dimensioni manipolate durante lo *scaling* del modello sono principalmente tre:

- **width**: il numero di canali di input e output di ogni layer della rete.
- **depth**: il numero di layer della rete.
- **resolution**: la dimensione delle immagini di input.

Questa tecnica è stata utilizzata con successo in diversi campi, come mostrato nell'ambito computer vision da Tan e Le con l'architettura EfficientNet [45] (predecessore di EfficientNetV2).

La rete EfficientNetV2 in particolare è stata allenata su ImageNet [19], un dataset di immagini di 1.000 classi utilizzato in passato per competizioni di machine learning, ed è pubblicamente disponibile su TensorFlow Hub [46], dove sono anche presenti diverse versioni dell'architettura, tutte con scale differenti di ampiezza, profondità e risoluzione, come evidenziato dalla tabella 2.1 nella pagina seguente.

Architettura e blocchi costruttivi Pur avendo un numero di parametri differente, ciascuna versione della EfficientNetV2 presenta, a grandi linee, gli stessi blocchi costruttivi [44]:

Convoluzioni : comuni layer convoluzionali composti da un numero variabile di kernel 3×3;

Tabella 2.1: Tabella comparativa delle varie versioni della EfficientNetV2.

Scala	# Parametri	Accuratezza ImageNet
B0	7.4M	78.7%
B1	8.1M	79.8%
B2	9.2M	$\approx$ 80%
B3	14M	82.1%
S	24M	84.9%
M	55M	86.2%
L	121M	86.8 %
XL	208M	87.3%

MBConvolution : particolare tipologia di blocco convoluzionale (figura 2.7 nella pagina successiva) che opera come segue [44]:

1. applica una convoluzione iniziale con un kernel 1×1 sul segnale X in input con dimensioni $H \times W \times C$;
2. computa una convoluzione *depthwise* 3×3 (che opera separatamente su ciascun canale con kernel differenti), dando in output un segnale $\hat{X}$ con dimensioni $H \times W \times 4C$;
3. applica ad $\hat{X}$ un blocco di *Squeeze-Excitation* [16], che individua le relazioni tra i canali e fungendo da meccanismo di *attenzione* ed enanchment di queste ultime:
   (a) esegue un'operazione di *squeezing* effettuando un average pooling su ogni canale $\hat{X}_1, ..., \hat{X}_{4C}$ in input, ottenendo un vettore $\vec{z} = z_1, ..., z_{4C}$ con un singolo valore per canale;
   (b) esegue un'operazione di *excitement*, dando in input $\vec{z}$ una sequenza di due layer fully-connected con attivazione rispettivamente sigmoide (σ) e ReLU e matrici dei pesi/bias W_1, W_2, b_1, b_2, computando la funzione:
   $$\vec{s} = \sigma\left(\text{ReLU}\left(\vec{z} \cdot W_1 + b_1\right) \cdot W_2 + b_2\right)$$
   (c) moltiplica ciascun canale di input $\hat{X}_c$ per il corrispettivo valore del vettore $\vec{s}$, ottenendo il segnale squeezed-excited:
   $$U_c = \hat{X}_c \cdot \vec{s}_c, \text{ con } c = 1, 2, ..., 4C;$$
4. riduce la dimensione del segnale sequeezed-excited U attraverso un'apposita convoluzione 1×1 che riporta l'input alla dimensione originale $H \times W \times C$, ottenendo $\hat{U}$;

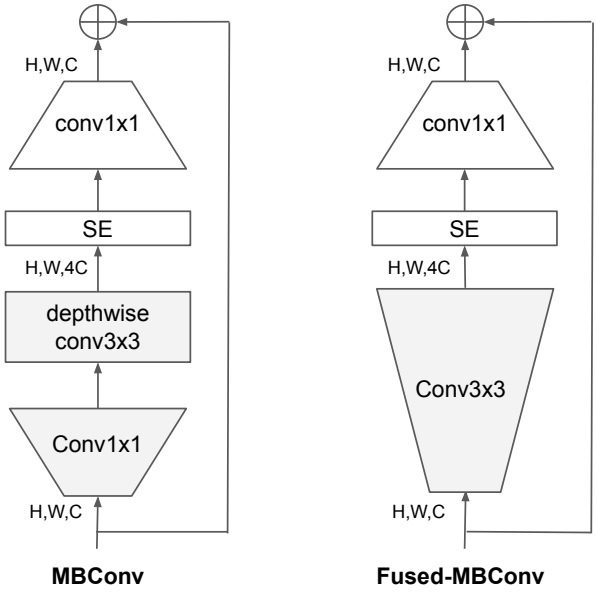

Figura 2.7: Diagramma rappresentativo della struttura di un blocco MBConvolutional e Fused-MBConvolutional [44].

5. somma l'input del layer X ad $\hat{U}$ in un'ottica simil-ResNet [11], tipica anche delle architetture basate sull'attention come i trasformer [51].

Fused-MBConvolution : variazione del blocco MBConvolution appena presentato ove i due step iniziali (la convoluzione 1×1 e quella depth-wise 3×3) sono sostituite da una semplice convoluzione 2D con kernel 3×3 (figura 2.7). Questo tipo di blocco è utilizzato generalmente nei primi layer delle architetture EfficientNetV2 in quanto, come osservato dagli autori [44], le convoluzioni depth-wise sono molto costose se poste tra i primi layer della rete.

Convoluzione e Pooling finale : coppia di layer posta solitamente alla fine dell'architettura, che presenta una convoluzione 1×1 ed un layer di max o average pooling 2D.

La tabella 2.2 nella pagina successiva mostra i parametri e l'architettura del modello in cui sono disposti i vari blocchi presentati nella versione EfficientNetV2S.

Tabella 2.2: Architettura di EfficientNetV2S.

Layer	Stride	# Canali
Conv 3x3	2	24
Fused-MBConv 3x3	1	24
Fused-MBConv 3x3	2	48
Fused-MBConv, 3x3	2	64
MBConv, 3x3, SE0.25	2	128
MBConv, 3x3, SE0.25	1	160
MBConv, 3x3, SE0.25	2	256
Conv1x1, Pooling, FC	-	1280

2.1.7 Transfer-learning

Il *transfer learning* è una tecnica di apprendimento automatico che consente di trasferire le conoscenze acquisite da un modello pre-allenato all'interno di uno nuovo. L'idea alla base di questa tecnica è che spesso i modelli di deep learning apprendono rappresentazioni dei dati generali e riutilizzabili in contesti differenti, consentendo a nuovi modelli di apprendere più rapidamente e con meno dati. Questa tecnica risulta particolarmente utile quando si dispone di una rete già allenata su un grande dataset e si desidera utilizzare queste conoscenze per allenare un nuovo modello su un dataset più piccolo, risparmiando tempo e risorse. Ideata da Geoffrey Hinton, Oriol Vinyals e Jeff Dean nel 2006 [13], il transfer learning è da allora stato applicato in molte modalità e contesti differenti, come il miglioramento delle performance in task di speech recognition [52], face-recognition [28] e riconoscimento di immagini [17]. Alcune delle più diffuse tecniche di transfer learning sono:

fine-tuning : tecnica di transfer learning che consiste nell'utilizzare i pesi dei layer di una rete pre-addestrata su un task simile a quello d'interesse in un nuovo modello comprendente nuovi layer, che viene allenato su un nuovo dataset (congelando o meno i pesi appresi in precedenza [53], [28], [17]).

feature extraction pre-training : tecnica di transfer learning che consiste nell'allenare un modello con architettura encoder-decoder (e.g. autoencoder [24] o trasformer [4]) ad estrarre feature significative alla ricostruzione dai dati del dataset di interesse (o uno simile, magari di dimensioni maggiori); successivamente, viene allenato un nuovo modello (ad esempio con architettura classificativa) su queste feature a svolgere il task d'interesse.

In questo lavoro, come vedremo in seguito (3.1 a pagina 32), la tecnica applicata è stata quella del fine-tuning: sono stati difatti utilizzati pesi delle varie reti EfficientNetV2 [44] e VGG [27] addestrate su ImageNet [19], allo scopo di allenare nuove reti sul dataset precedentemente descritto (1 a pagina 2).

2.2 Stato dell'arte

Come affermato da Almryad e Kutucu [1], l'identificazione delle specie di farfalle è un compito difficile a causa dell'enorme quantità di specie diverse e delle elevate somiglianze tra di esse. Sebbene esistano molte pubblicazioni sulla classificazione delle immagini, non molti sono i lavori scientifici che trattano direttamente l'applicazione della classificazione delle immagini alle specie di farfalle, in quanto la maggior parte delle pubblicazioni sulla classificazione delle immagini in questo campo risulta troppo generica o specifica. I lavori generici non forniscono spesso informazioni sufficienti su come applicare la classificazione delle immagini alle specie di farfalle, mentre quelle specifiche sono rare e talvolta troppo focalizzate su una singola specie di farfalla. Dunque, per analizzare lo stato dell'arte, esamineremo sia alcuni lavori generici sulla classificazione delle specie di insetti sia altri specifici inerenti a quella delle specie di farfalle.

Zhu e Zhang [57] hanno classificato le immagini di insetti lepidotteri utilizzando una regione integrativa abbinata e presentando un approccio wavelet discreto a doppio complesso. Testando il loro metodo su una collezione di oltre 100 insetti lepidotteri di 18 generi e l'accuratezza del riconoscimento è risultata dell'84.47%.

Silva, Grassi, Francoy e Costa [42] hanno effettuato esperimenti atti a comprendere quali fossero le strategie di selezione delle caratteristiche e i classificatori migliori per identificare le sottospecie di api tra i sette selezionatori di caratteristiche e i metodi di classificazione disponibili. Dai risultati è emerso come la migliore combinazione sia il classificatore naive Bayes e l'estrazione di caratteristiche basata su mutua informazione [50].

Hernandez, Jimenez-Segura [12] hanno utilizzato un approccio basato su reti neurali per categorizzare 740 specie ed un dataset di 11198 istanze. L'accuratezza è stata del 91.65% per i pesci, del 92.87% per la flora e del 91.25% per le farfalle.

Iamsaata, Horataa, Sunata e Thipayanga [18] hanno utilizzato le macchine di apprendimento estremo (ELM) per categorizzare le farfalle e hanno confrontato i risultati con l'algoritmo SVM. L'approccio ELM si è rivelato avere un'accuratezza del 90.37%, leggermente superiore a quella ottenuta dall'SVM.

Kang, Cho e Lee [22] hanno prodotto una rete neurale per classificare le farfalle in base alla forma delle ali, con un'accuratezza dell'80.3%.

La CNN realizzata da Arzar, Sabri, Mohd Johari, Amilah, Noordin e Ibrahim [2] ha ottenuto ottime performance nella classificazione delle specie di farfalle, ma su un dataset di dimensioni limitate.

Zhu e Spachos [58] hanno anche sfruttato con successo un modello VGG19 [27] per identificare le specie di farfalle per un'applicazione mobile. Nonostante il dataset utilizzato non fosse molto grande (< 1000 immagini), hanno riscontrato come il loro metodo superasse i metodi tradizionali con un margine significativo.

Un ulteriore passo in avanti è stato fatto da Lin, Jia, Gao e Huang [26] hanno applicato una rete neurale convoluzionale con skip-connections (SCCNN) su un dataset contenente svariate specie di farfalle, raggiungendo elevati livelli di precisione ed accuratezza. Tuttavia, il loro dataset prendeva in considerazione solo immagini prodotte laboratorio ad alta fedeltà acquisite in condizioni di illuminazione stabile di una lampada a LED vuota senza ombre, rendendo i loro modelli presumibilmente poco robusti.

Infine, anche se non ancora sperimentate nell'ambito della classificazione di specie di insetti e farfalle, sono degne di nota le architetture EfficientNet [45] ed EfficientNetV2 [44] sono state con successo utilizzate nella classificazione di immagini, superando le più moderne tecniche di stato dell'arte su popolari dataset di benchmark come ImageNet [19], suggerendo l'ipotesi di poter ottenere risultati eccellenti anche nei task di classificazione di specie di farfalle.

Nel complesso, gli ottimi risultati ottenuti in letteratura sembrano suggerire che le tecniche di deep learning siano significativamente migliori dei metodi tradizionali nella classificazione delle specie di insetti, comprese quelle di farfalle. Ciononostante, non tutti i modelli allo stato dell'arte sono stati messi alla prova su questo task, creando un razionale significativo per gli esperimenti svolti e presentati in questo elaborato.

2.3 Tecnologie

2.3.1 Numpy

NumPy [32] è un pacchetto di Python che offre una potenti ed efficiente struttura dati ad array multidimensionali implementata a basso livello in C. Vengono fornite poi funzioni matematiche avanzate che operano su tali array e una vasta selezione di tool di algebra lineare, trasformate di Fourier e funzioni di randomizzazione.

In questo lavoro, le API offerte da NumPy sono state utilizzate per implementare molte delle funzioni di trasformazione del dataset applicate durante il preprocessing. Ad esempio, alcune funzioni NumPy sono state usate per effettuare l'operazione di normalizzazione dei valori dei pixel tra 0 ed 1 (originariamente compresi tra 0 e 255 come da standard nelle immagini RGB).

2.3.2 Pandas

Pandas [33] è un pacchetto Python open-source che fornisce una struttura dati efficiente e versatile per l'analisi dei dati, chiamata *Data Frame*. Questa libreria può essere utilizzata per effettuare una molteplici operazioni sui dati, tra cui filtraggio, pulizia, trasformazione e varie analisi statistiche.

In questo lavoro, come già detto in precedenza (1.3.2 a pagina 8), le API di Pandas sono state usate per la lettura iniziale dei file CSV previa alle operazioni di preprocessing, e nella formattazione ed il salvataggio del file di sottomissione finale della challenge (1.3.2 a pagina 8).

2.3.3 Scikit-Learn

Scikit-Learn [40] è una popolare libreria per l'apprendimento automatico gratuita e open-source per Python, con gran parte della sua implementazione di basso livello scritta in C. È progettata per essere completamente interoperabile con le librerie numeriche e scientifiche come NumPy [32] e SciPy [41] di Python.

In questo lavoro, questa libreria è stata utilizzata in due fasi differenti:

- splitting del dataset etichettato in train, validation ed evaluation set, come già detto in precedenza (1.3.2 a pagina 8);
- valutazione finale delle performance dei modelli sull'evaluation set, dove alcune delle API di Scikit-Learn sono state richiamate per valutare le performance sulle predizioni del modello con specifiche metriche e visualizzare i risultati nelle matrici di confusione.

2.3.4 TensorFlow

TensorFlow [9] è una libreria software open-source per il calcolo numerico che utilizza grafi di flusso di dati. I nodi del grafo rappresentano le operazioni matematiche, mentre i bordi del grafo rappresentano gli array di dati multidimensionali (tensori) che fluiscono tra esse nell'esecuzione. Quest'architettura

flessibile consente di distribuire il calcolo su una o più CPU/GPU in un desktop, server o dispositivo mobile con una singola API. Il framework è stato originariamente sviluppato dai ricercatori e dagli ingegneri del Google Brain Team [10], all'interno dell'organizzazione di ricerca sulla Machine Intelligence di Google, per condurre ricerche sull'apprendimento automatico e sulle reti neurali profonde, ma il sistema è abbastanza generale da poter essere applicato anche in un'ampia varietà di altri settori.

Nel lavoro correlato a questo elaborato, TensorFlow è stato utilizzato in diverse fasi:

- creazione e formattazione dei dataset di training, testing, evaluation e validation;
- esecuzione della data augmentation;
- sviluppo, training, testing e salvataggio dei modelli realizzati (2.3.5);
- interrogazione dei modelli salvati allo scopo di realizzare i file di sottomissione.

2.3.5 Keras

Keras è un'API open-source ad alto livello per lo sviluppo di reti neurali, scritta in Python ed in grado di funzionare su TensorFlow [9], CNTK [30] e Theano [47]. È stata sviluppata con l'obiettivo di consentire una sperimentazione rapida e rendere il passaggio dall'idea al risultato il meno complesso possibile, caratteristica considerata dagli autori fondamentale per una buona ricerca.

In questo lavoro, Keras (in particolare la versione inclusa in TensorFlow) è stata utilizzata per sviluppare, allenare e salvare i modelli utilizzati nelle sperimentazioni effettuate.

Capitolo 3

Esperimenti effettuati

Di seguito viene presentata una panoramica delle architetture adoperate negli esperimenti ed i relativi risultati ottenuti.

3.1 Architetture proposte

3.1.1 ButterflyNet

La prima architettura proposta è stata denominata ButterflyNet, con riferimento al task della challenge in oggetto, ovvero la classificazione di specie di farfalle. Come si può osservare nella figura 3.1 nella pagina seguente, essa è composta da:

1. un estrattore di feature basato sull'architettura VGG, pre-allenato sul dataset ImageNet [19];
2. un MLP che agisce da classificatore finale sulle feature estratte dalla VGG, che presenta 6 neuroni di output, uno per ciascuna classe.

La versione della VGG utilizzata, così come la struttura del classificatore MLP è stata fatta variare sperimentando diverse configurazioni, con risultati variabili che verranno mostrati nella sezione apposita 3.2 a pagina 35. In particolare, sono state sperimentate:

- le versioni VGG16 e VGG19 presentate dagli autori [27];
- una versione del classificatore MLP con 1 layer interno, con un numero di neuroni variabile tra 128 a 512.

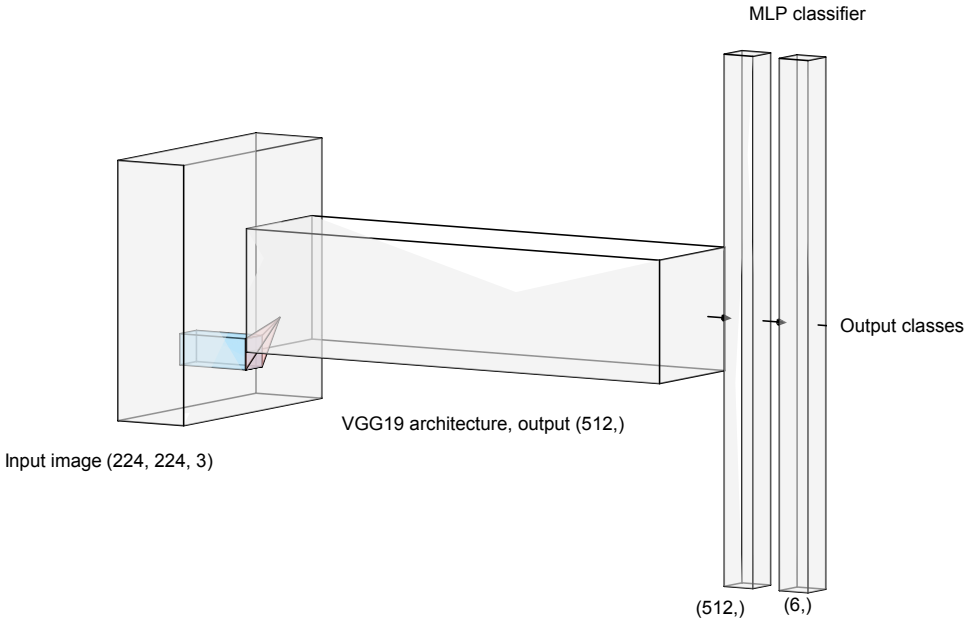

Figura 3.1: Diagramma rappresentativo dell'architettura di ButterflyNet.

3.1.2 YoYNet

La seconda architettura proposta è stata denominata YoYNet, come abbreviazione di Yum or Yuck Network. Come si può osservare nella figura 3.2 nella pagina seguente, essa è composta da:

1. un estrattore di feature basato sull'architettura EfficientNetV2, preallenato sul dataset ImageNet [19];
2. un layer di average o max pooling globale che ha lo scopo di comprimere le feature map estratte dalla EfficientNetV2;
3. un MLP che agisce da classificatore finale sulle feature compresse dal layer di pooling, che presenta 6 neuroni di output, uno per ciascuna classe.

La versione di EfficientNetV2 utilizzata, così come la struttura del classificatore MLP è stata fatta variare sperimentando diverse configurazioni, con risultati variabili che verranno mostrati nella sezione apposita 3.2 a pagina 35. In particolare, sono state sperimentate:

- le versioni EfficientNetV2 B0, B1, B2 e B3;

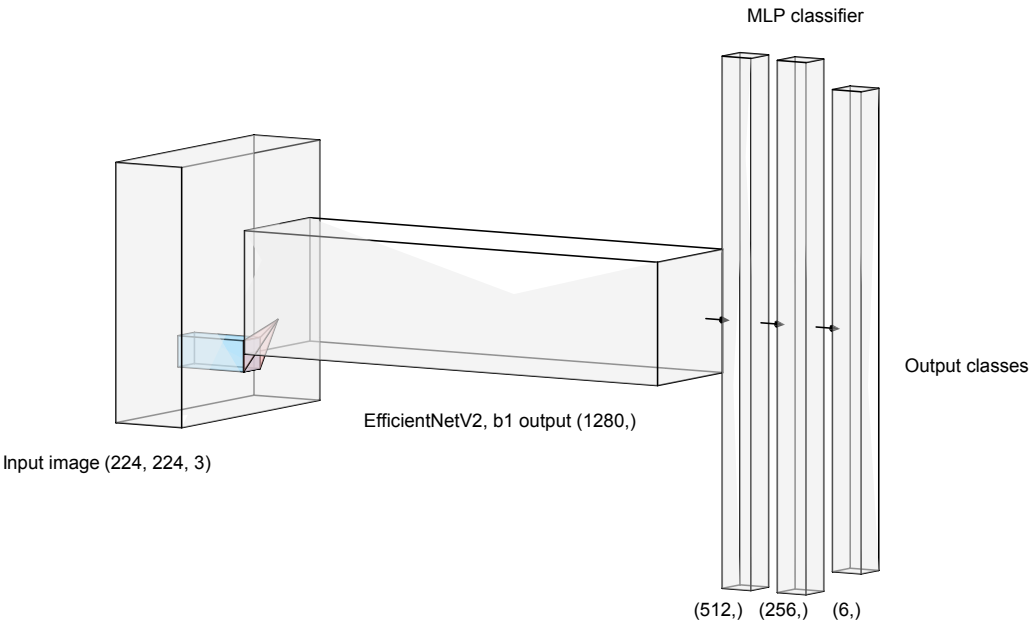

Figura 3.2: Diagramma rappresentativo dell'architettura di YoYNet.

- una versione del classificatore MLP a 1 o 2 layer interni, con un numero di neuroni variabile tra 128 a 512 (decrescente).

3.1.3 Training e regolarizzazione

Regolarizzazione Essendo il modello particolarmente profondo, sono state applicati diverse tecniche di regolarizzazione pesante (oltre ai layer di batch normalization presenti all'interno dell'architettura EfficientNetV2, ove utilizzata):

- dropout rate pari a 0.5 [14] su tutti i layer fully-connected interni;
- regolarizzazione L_1 e/o L_2 applicata a pesi, bias e attivazione di uno o più layer densi e a con λ variabile 10^{-6} e 10^{-5}.

Training Il training dei modelli realizzati è stato effettuato sfruttando le API di Keras, utilizzando due diversi tipi di ottimizzatori per minimizzare la loss function categorical cross-entropy (2.1.2 a pagina 14): Adadelta [56] e Adam [23]. Dopo aver caricato il dataset ed averne effettuato un pre-batch con una batch size pari a 32, i modelli sono stati allenati per un massimo di 400 epoche, fermando l'allenamento e ripristinando i pesi dell'epoca migliore

(in un'ottica early-stopping) qualora la loss non fosse migliorata per 30 epoche. La figura 3.3 nella pagina successiva mostra la loss e l'accuracy durante il training del miglior modello ottenuto nelle prime 200 epoche.

3.2 Risultati

Le tabelle 3.1 a pagina 37, 3.2 a pagina 38 e 3.3 a pagina 39 3.4 a pagina 39 mostrano i risultati finali più rilevanti ottenuti dai modelli al variare degli iperparametri, valutati secondo l'accuratezza di classificazione e l'$F1$-score.

Nella figura 3.4 a pagina 41 vengono mostrate le matrici di confusione per la classificazione binaria "Yum or Yuck" e per quella tra le sei specie delle farfalle.

3.2.1 Discussione dei risultati

Parametri dei modelli Dai risultati dei vari modelli mostrati nelle tabelle tabelle 3.1 a pagina 37, 3.2 a pagina 38 e 3.3 a pagina 39 3.4 a pagina 39 è possibile trarre alcune conclusioni:

- l'algoritmo di ottimizzazione Adadelta appare avere performance migliori di Adam nel dare ai modelli capacità di generalizzazione;
- le architetture VGG19 e VGG16 non sembrano avere variazioni significative di performance tra di loro;
- come d'aspettativa, i modelli che fanno uso di architettura VGG non hanno ottenuto performance paragonabili a quelli che fanno uso di EfficientNetV2, essendo tale architettura più recente e tendenzialmente migliore;
- la versione B1 della EfficientNetV2 sembra avere il range ottimale di complessità, in quanto sia le architetture B0 che B2 sembrano avere performance peggiori;
- la regolarizzazione si rivela essere, non sorprendentemente, di fondamentale importanza in un modello complesso come quello costruito, e la configurazione ottimale sembra essere il porre un termine di regolarizzazione sia sui bias e sui pesi dei layer densi interni, che sui valori di attivazione di tutti i layer densi, con un $\lambda = 10^{-6}$.

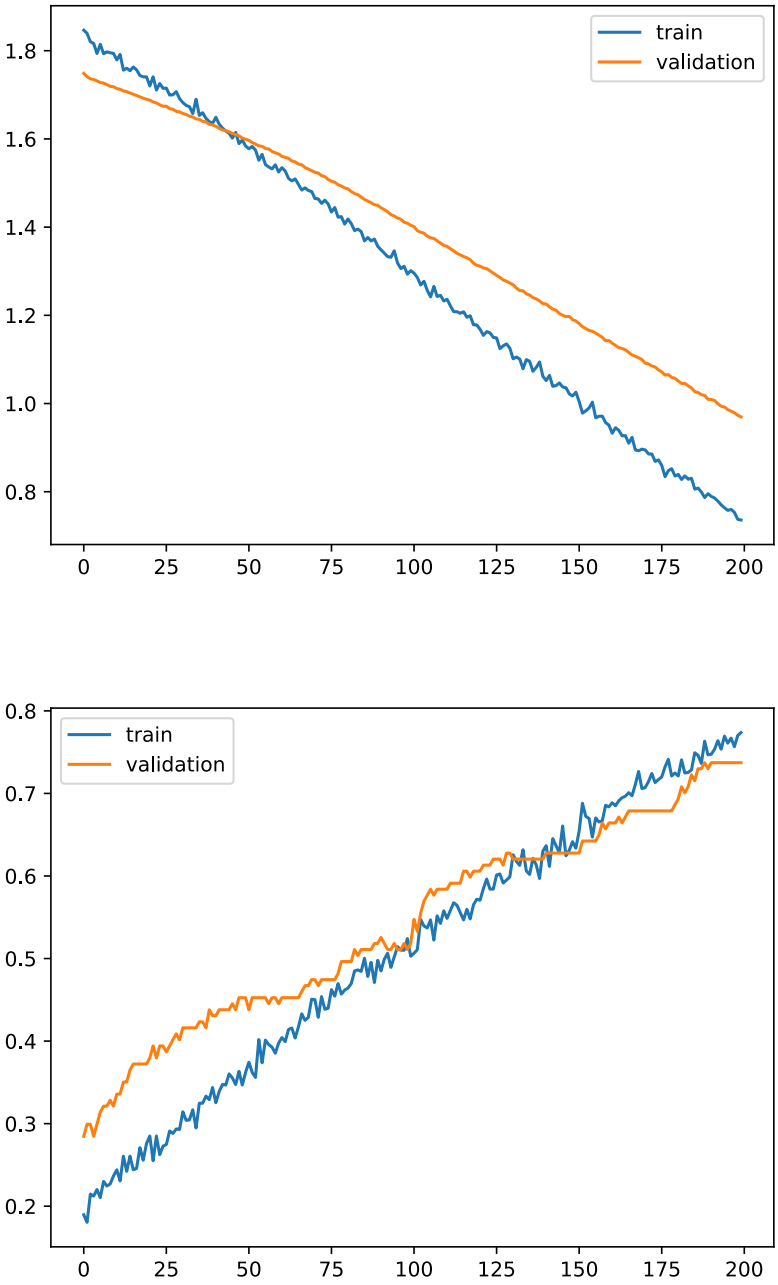

Figura 3.3: Grafici illustrativi dell'andamento di accuratezza e loss su train e validation set durante le prime 200 epoche di training del miglior modello ottenuto.

Tabella 3.1: Tabella che mostra scala della EfficientNetV2 o della VGG pre-trained utilizzata, se i relativi pesi sono stati freezati o meno, le dimensioni dei layer dell'MLP e i relativi termini di regolarizzazione sui bias nei più rilevanti modelli sperimentati.

#	**Sc./Ver.**	**Freeze pre-trained**	**Dense dim.**	**Regol. Bias**
1	B0	Sì	$[512, 128, 6]$	$\left[L_1 : 10^{-6}, L_1 : 10^{-6}, -\right]$
2	B0	No	$[512, 256, 6]$	$\left[L_1 : 10^{-6}, L_1 : 10^{-6}, -\right]$
3	B2	No	$[512, 6]$	$\left[L_1 : 10^{-6}, -\right]$
4	B1	No	$[512, 256, 6]$	$\left[L_2 : 10^{-5}, L_2 : 10^{-5}, -\right]$
5	B1	No	$[512, 6]$	$[-, -]$
6	B1	No	$[512, 512, 6]$	$\left[L_1 : 10^{-6}, L_1 : 10^{-6}, -\right]$
7	**B1**	**No**	$\mathbf{[512, 256, 6]}$	$\left[\boldsymbol{L_1 : 10^{-6}, L_1 : 10^{-6}, -}\right]$
8	B1	No	$[512, 512, 6]$	$\left[L_1 : 10^{-6}, L_1 : 10^{-6}, -\right]$
9	VGG19	No	$[512, 6]$	$[-, -]$
10	VGG19	No	$[512, 256, 6]$	$\left[L_1 : 10^{-6}, L_1 : 10^{-6}, -\right]$
11	VGG16	No	$[512, 256, 6]$	$\left[L_1 : 10^{-5}, L_1 : 10^{-5}, -\right]$
12	VGG16	No	$[512, 6]$	$\left[L_2 : 10^{-5}, -\right]$

Tabella 3.2: Tabella che mostra termini di regolarizzazione su pesi ed attivazione nei più rilevanti modelli sperimentati.

#	**Regol. Pesi**	**Regol. Attiv.**
1	$[L_1 : 10^{-6}, L_1 : 10^{-6}, -]$	$[-, L_1 : 10^{-6}, L_1 : 10^{-6}]$
2	$[L_1 : 10^{-6}, L_1 : 10^{-6}, -]$	$[-, L_1 : 10^{-6}, L_1 : 10^{-6}]$
3	$[L_1 : 10^{-6}, -]$	$[-, L_1 : 10^{-6}]$
4	$[L_2 : 10^{-5}, L_2 : 10^{-5}, -]$	$[L_1 : 10^{-6}, L_1 : 10^{-6}, L_1 : 10^{-6}]$
5	$[L_1 : 10^{-6}, -]$	$[-, -]$
6	$[L_1 : 10^{-6}, L_1 : 10^{-6}, -]$	$[L_1 : 10^{-6}, L_1 : 10^{-6}, L_1 : 10^{-6}]$
7	$\boldsymbol{[L_1 : 10^{-6}, L_1 : 10^{-6}, -]}$	$\boldsymbol{[L_1 : 10^{-6}, L_1 : 10^{-6}, L_1 : 10^{-6}]}$
8	$[L_1 : 10^{-5}, L_1 : 10^{-6}, -]$	$[L_1 : 10^{-6}, L_1 : 10^{-6}, L_1 : 10^{-6}]$
9	$[-, -]$	$[-, -]$
10	$[L_1 : 10^{-6}, L_1 : 10^{-6}, -]$	$[L_1 : 10^{-6}, L_1 : 10^{-6}, L_1 : 10^{-6}]$
11	$[L_1 : 10^{-5}, L_1 : 10^{-5}, -]$	$[L_1 : 10^{-5}, L_1 : 10^{-5}, L_1 : 10^{-5}]$
12	$[L_2 : 10^{-5}, -]$	$[-, L_1 : 10^{-6}]$

Tabella 3.3: Tabella che mostra ottimizzatore e learning rate utilizzati nel training dei più rilevanti modelli sperimentati.

#	**Ottimizzatore**	η
1	Adadelta	1
2	Adadelta	1
3	Adadelta	1
4	Adam	$3 \cdot 10^{-7}$
5	Adadelta	1
6	Adadelta	1
7	**Adadelta**	**1**
8	Adadelta, Adam	$1, 3 \cdot 10^{-7}$
9	Adam	$3 \cdot 10^{-7}$
10	Adam	$3 \cdot 10^{-7}$
11	Adam	$3 \cdot 10^{-7}$
12	Adam	$3 \cdot 10^{-6}$

Tabella 3.4: Tabella che mostra precision, recall, accuracy ed $F1$-score ottenuti dai più rilevanti modelli sperimentati.

#	**Precision**	**Recall**	**Accuratezza**	**F1-score**
1	88.30%	88.51%	89.78%	88.11%
2	91.18%	91.28%	91.25%	91.24%
3	92.58%	92.25%	92.70%	92.40%
4	86.74%	86.55%	86.55%	86.28%
5	92.10%	91.23%	91.23%	91.21%
6	94.88%	93.97%	94.15%	94.12%
7	**96.64%**	**96.49%**	**96.49%**	**96.48%**
8	94.30%	94.15%	94.74%	94.68%
9	83.58%	83.34%	83.63%	82.27%
10	90.69%	90.97%	90.64%	90.47%
11	88.82%	88.08%	88.89%	87.89%
12	88.30%	88.66%	88.89%	88.26%

Performance del miglior modello Dalle matrici di confusione 3.4 nella pagina successiva è possibile effettuare diverse osservazioni sul miglior modello ottenuto:

- benché le performance sembrino essere ottime nella classificazione delle specie, risultano ancora migliori nel task di distinzione tra farfalle "Yum" e "Yuck", dove è stato ottenuto circa il 98.84% di accuratezza;
- i principali errori commessi sembrano riguardare le classi "monarch", "viceroy" e "spicebush".

I due fenomeni possono avere spiegazioni e connessioni che vale la pena analizzare nel dettaglio.

In primo luogo, le tre classi coinvolte sono quelle con il numero più scarso di campioni tra tutte, com'è evidente dalla figura 1.3 a pagina 7; una possibile spiegazione dunque delle performance più basse potrebbe essere la "scarsa abitudine" del modello a immagini che ritraggono esemplari di queste specie. Quest'ipotesi è ancora più probabile se si osserva che la farfalla monarca viene spesso confusa dal modello proprio con le viceroy e la spicebush swallowtail, e che altri due errori riguardano proprio quest'ultima, che viene invece confusa con la black swallowtail. Un'altra possibile spiegazione agli errori effettuati dal modello nella distinzione tra viceroy e farfalla monarca come tra black e spicebush swallowtail è l'estrema somiglianza visiva che queste specie presentano fra loro (non a caso, si parla di *mimetismo*). Si noti inoltre che tali ipotesi non si escludono tra loro, ed è altresì probabile che siano entrambe vere.

La spiegazione delle alte performance nel task di distinzione tra farfalle "Yum" e "Yuck" è invece abbastanza immediata: un task di classificazione binaria è generalmente più semplice di uno multiclasse. Ciononostante, è interessante notare come la maggioranza degli errori (4/6) riguardino farfalle che appartengono alla stessa categoria di classificazione binaria. Ciò potrebbe persino portare a supporre che il modello abbia imparato dei pattern che distinguono le reali specie velenose da quelle che invece fanno uso di *mimetismo batesiano*, ma sarebbero necessari studi futuri su dataset più ampi per approfondire questo aspetto.

3.3 Conclusioni

I risultati ottenuti dai modelli realizzati si presentano come promettenti, soprattutto considerate le limitazioni dell'hardware a disposizione che hanno

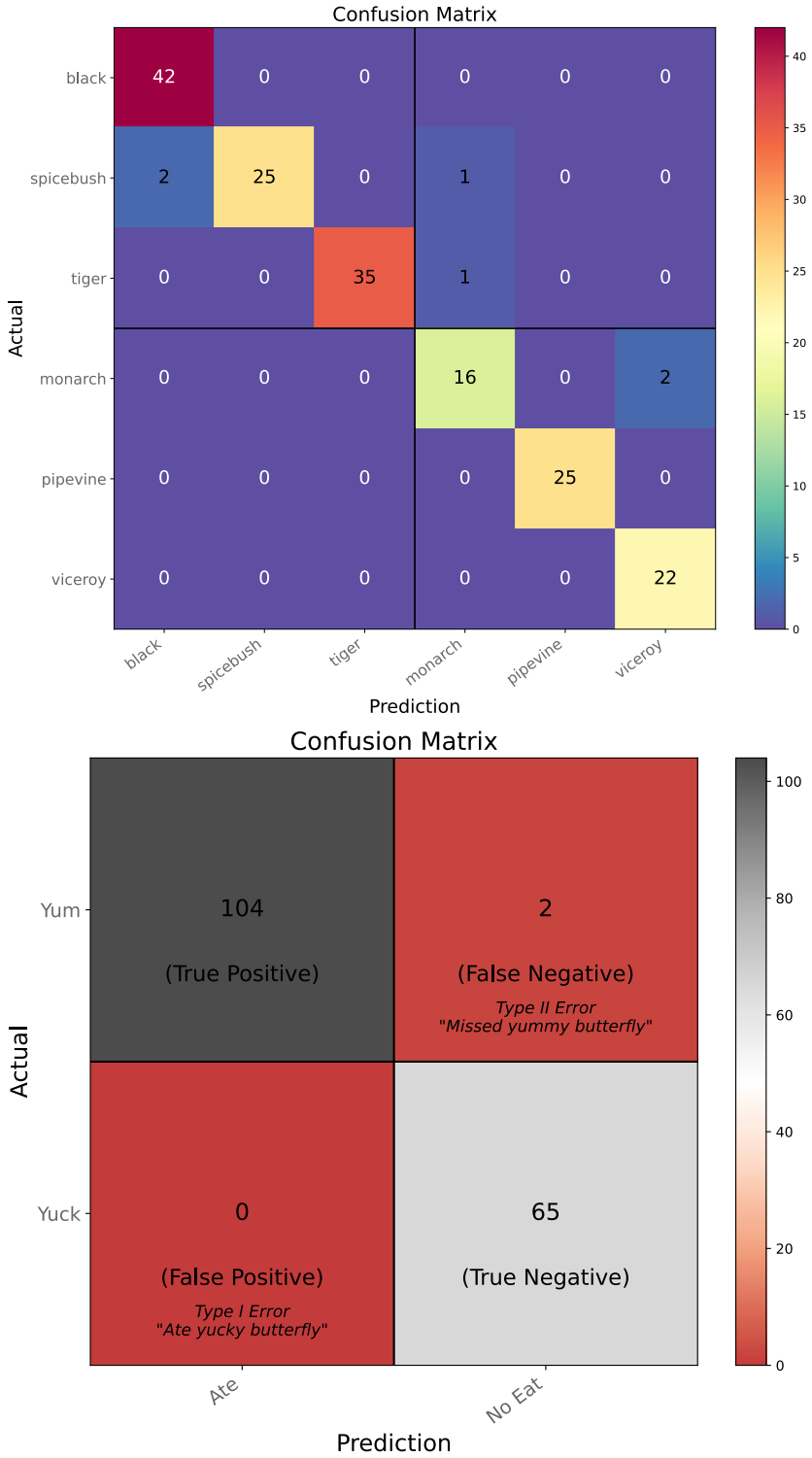

Figura 3.4: Matrici di confusione per le sei specie di farfalle e per la classificazione binaria "Yum or Yuck" per il miglior modello ottenuto.

limitato molto la gamma di possibili architetture e variazioni delle stesse da poter sperimentare.

Uno dei possibili sviluppi futuri del progetto potrebbe essere la sperimentazione dei modello su un dataset più ampio, in modo da verificare le congetture formulate ed effettuare eventuali cambiamenti all'architettura per migliorarla ulteriormente.

Un altro dei possibili sviluppi futuri dei modelli potrebbe essere l'applicazione di un meccanismo di self-attention o normale attention che coinvolga le feature globali estratte dagli ultimi layer dell'EfficientNetV2 o delle VGG con quelle di layer intermedi. Molteplici studi, tra cui quelli riguardanti le architetture EfficientNet [45] e [44], hanno suggerito come un approccio di questo tipo possa migliorare notevolmente le performance di un'architettura convoluzionale.

Bibliografia

[1] Ayad Saad Almryad e Hakan Kutucu. «Automatic identification for field butterflies by convolutional neural networks». In: *Engineering Science and Technology, an International Journal* (2020). DOI: 10.1016/j.jestch.2020.01.006.

[2] Nur Arzar et al. «Butterfly Species Identification Using Convolutional Neural Network (CNN)». In: 2019. DOI: 10.1109/I2CACIS.2019.8825031.

[3] Gustavo Carneiro, Jacinto Nascimento e Andrew P. Bradley. «Chapter 14 - Deep Learning Models for Classifying Mammogram Exams Containing Unregistered Multi-View Images and Segmentation Maps of Lesions». In: *Deep Learning for Medical Image Analysis*. A cura di S.Kevin Zhou, Hayit Greenspan e Dinggang Shen. Academic Press, 2017, pp. 321–339. ISBN: 978-0-12-810408-8. DOI: https://doi.org/10.1016/B978-0-12-810408-8.00019-5. URL: https://www.sciencedirect.com/science/article/pii/B9780128104088000195.

[4] Jacob Devlin et al. «BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding». In: *CoRR* (2018). DOI: 10.48550/arXiv.1810.04805.

[5] John Duchi, Elad Hazan e Yoram Singer. «Adaptive subgradient methods for online learning and stochastic optimization». In: *Journal of Machine Learning Research* (2011). URL: https://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf.

[6] *File:Computer.Science.AI.Neuron.XOR.svg*. URL: https://commons.wikimedia.org/wiki/File:Computer.Science.AI.Neuron.XOR.svg (visitato il 18/09/2022).

[7] *File:Perceptron-unit.svg*. URL: https://commons.wikimedia.org/wiki/File:Perceptron-unit.svg (visitato il 18/09/2022).

[8] Paul Gavrikov. *visualkeras*. https://github.com/paulgavrikov/visualkeras. 2020.

[9] Google. *TensorFlow*. URL: https://www.tensorflow.org/ (visitato il 17/09/2022).

[10] *Google Brain Team*. URL: https://research.google/teams/brain/ (visitato il 19/09/2022).

[11] Kaiming He et al. «Deep Residual Learning for Image Recognition». In: *arXiv* (2015). DOI: 10.48550/ARXIV.1512.03385.

[12] Andres Hernandez-Serna e Luz Jimenez-Segura. «Automatic identification of species with neural networks». In: *PeerJ* (2014). DOI: 10.7717/peerj.563.

[13] Geoffrey Hinton, Oriol Vinyals e Jeff Dean. «Distilling the knowledge in a neural network». In: *arXiv* (2015). DOI: https://doi.org/10.48550/arXiv.1503.02531.

[14] Geoffrey E. Hinton et al. «Improving neural networks by preventing co-adaptation of feature detectors». In: *ArXiv* abs/1207.0580 (2012).

[15] Arthur E Hoerl e Robert W Kennard. «Ridge regression: Biased estimation for nonorthogonal problems». In: *Technometrics* (1970). URL: http://homepages.math.uic.edu/~lreyzin/papers/ridge.pdf.

[16] Jie Hu et al. «Squeeze-and-Excitation Networks». In: *arXiv* (2017). DOI: 10.48550/ARXIV.1709.01507.

[17] Minyoung Huh, Pulkit Agrawal e Alexei Efros. «What makes ImageNet good for transfer learning?» In: (2016).

[18] Suthasinee Iamsa-at et al. «Improving butterfly family classification using past separating features extraction in extreme learning machine». In: *Proceedings of the 2nd International Conference on Intelligent Systems and Image Processing*. 2014. URL: https://scholar.archive.org/work/t7oqxsmmq5gz5ppojpskwl6axe/access/wayback/https://www2.ia-engineers.org/conference/index.php/icisip/icisip2014/paper/download/385/316.

[19] *ImageNet*. URL: https://www.image-net.org/index.php (visitato il 19/09/2022).

[20] Sergey Ioffe e Christian Szegedy. «Batch normalization: Accelerating deep network training by reducing internal covariate shift». In: (2015).

[21] Shahin Ismail et al. «Speaker identification and verification using Gaussian mixture speaker models». In: *Neural Computing and Applications* (2021). DOI: 10.1007/s00521-021-06226-w.

[22] Seung-Ho Kang, Jung-Hee Cho e Sang-Hee Lee. «Identification of butterfly based on their shapes when viewed from different angles using an artificial neural network». In: *Journal of Asia-Pacific Entomology* (2014). DOI: https://doi.org/10.1016/j.aspen.2013.12.004.

[23] Diederik Kingma e Jimmy Ba. «Adam: A Method for Stochastic Optimization». In: *International Conference on Learning Representations* (2014). DOI: 10.48550/arXiv.1412.6980.

[24] Maximilian Kohlbrenner et al. «Pre-Training CNNs Using Convolutional Autoencoders». In: 2017.

[25] Andrew Kraemer e Dean Adams. «Predator Perception of Batesian Mimicry and Conspicuousness in a Salamander». In: *Evolution; international journal of organic evolution* (2013). DOI: 10.1111/evo.12325.

[26] Zhongqi Lin et al. «Fine-Grained Visual Categorization of Butterfly Specimens at Sub-species Level Via a Convolutional Neural Network with skip-connections». In: *Neurocomputing* (2019). DOI: 10.1016/j.neucom.2019.11.033.

[27] Shuying Liu e Weihong Deng. «Very deep convolutional neural network based image classification using small training sample size». In: *2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR)*. 2015. DOI: 10.1109/ACPR.2015.7486599.

[28] Joseph Luttrell et al. «A deep transfer learning approach to fine-tuning facial recognition models». In: *2018 13th IEEE Conference on Industrial Electronics and Applications (ICIEA)*. 2018. DOI: 10.1109/ICIEA.2018.8398162.

[29] W.S. McCulloch e W. Pitts. «A logical calculus of the ideas immanent in nervous activity». In: *Bulletin of Mathematical Biophysics* (1943). DOI: 10.1007/BF02478259.

[30] *Microsoft Cognitive Toolkit*. URL: https://learn.microsoft.com/en-us/cognitive-toolkit/ (visitato il 19/09/2022).

[31] Marvin Minsky e Seymour Papert. *A Step toward the Understanding of Information Processes: Perceptrons. An Introduction to Computational Geometry*. 1969.

[32] *NumPy*. URL: https://numpy.org/ (visitato il 19/09/2022).

[33] *Pandas*. URL: https://pandas.pydata.org/ (visitato il 17/09/2022).

[34] George Philipp, Dawn Song e Jaime G. Carbonell. «Gradients explode - Deep Networks are shallow - ResNet explained». In: *CoRR* abs/1712.05577 (2017). DOI: 10.48550/arXiv.1712.05577.

[35] Joseph Redmon et al. «You Only Look Once: Unified, Real-Time Object Detection». In: (2016). DOI: 10.1109/CVPR.2016.91.

[36] F. Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, 1962.

[37] F. Rosenblatt. «The Perceptron: a probabilistic model for information storage and organization in the brain». In: *Psychological Review* (1958).

[38] David E. Rumelhart, Geoffrey E. Hinton e Ronald J. Williams. «Learning representations by back-propagating errors». In: *Nature* (1986). DOI: https://doi.org/10.1038/323533a0.

[39] Graeme D. Ruxton, Thomas N. Sherratt e Michael P. Speed. *115The evolution and maintenance of Müllerian mimicry*. Oxford University Press, 2004. DOI: 10.1093/acprof:oso/9780198528609.003.0010.

[40] *Scikit-Learn*. URL: https://scikit-learn.org/stable/ (visitato il 17/09/2022).

[41] *SciPy*. URL: https://scipy.org/ (visitato il 19/09/2022).

[42] Felipe Silva et al. «Evaluating classification and feature selection techniques for honeybee subspecies identification using wing images». In: *Computers and Electronics in Agriculture* (2015). DOI: 10.1016/j.compag.2015.03.012.

[43] Christian Szegedy et al. «Going deeper with convolutions». In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015. DOI: 10.1109/CVPR.2015.7298594.

[44] Mingxing Tan e Quoc Le. «EfficientNetV2: Smaller Models and Faster Training». In: *Proceedings of the 38th International Conference on Machine Learning*. 2021. DOI: 10.48550/arXiv.2104.00298.

[45] Mingxing Tan e Quoc V. Le. «EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks». In: *CoRR* (2019). DOI: 10.48550/arXiv.1905.11946.

[46] *TensorFlow Hub*. URL: https://www.tensorflow.org/hub (visitato il 19/09/2022).

[47] *Theano*. URL: https://pypi.org/project/Theano/ (visitato il 19/09/2022).

[48] R. Tibshirani. «Regression Shrinkage and Selection via the Lasso». In: *Journal of the Royal Statistical Society (Series B)* (1996). DOI: 10.1111/j.2517-6161.1996.tb02080.x.

[49] A. N. Tikhonov. «Solution of incorrectly formulated problems and the regularization method». In: *Soviet Mathematics* (1963).

[50] Kari Torkkola. «Feature extraction by non-parametric mutual information maximization». In: *Journal of machine learning research* Mar (2003). DOI: 10.5555/944919.944981.

[51] Ashish Vaswani et al. «Attention is All you Need». In: *Advances in Neural Information Processing Systems*. 2017. URL: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[52] Dong Wang e Thomas Fang Zheng. «Transfer learning for speech and language processing». In: *2015 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA)*. DOI: 10.1109/APSIPA.2015.7415532.

[53] K. Wang et al. «Pay Attention to Features, Transfer Learn Faster CNNs». In: *ICLR*. 2020.

[54] Bernard Widrow. «Adaptive "ADALINE" neuron using chemical" memistors». In: (1960). URL: https://www-isl.stanford.edu/~widrow/papers/t1960anadaptive.pdf.

[55] *Yum or Yuck Butterfly Mimics 2022*. URL: https://www.kaggle.com/competitions/yum-or-yuck-butterfly-mimics-2022/overview/the-butterflies (visitato il 15/09/2022).

[56] Matthew D. Zeiler. «ADADELTA: An Adaptive Learning Rate Method». In: (2012). DOI: https://doi.org/10.48550/arXiv.1212.5701.

[57] Leqing Zhu e Zhen Zhang. «Insect recognition based on integrated region matching and dual tree complex wavelet transform». In: *Journal of Zhejiang University - Science C* (2011). DOI: 10.1631/jzus.C0910740.

[58] Lili Zhu e P. Spachos. «Butterfly Classification with Machine Learning Methodologies for an Android Application». In: 2019. DOI: 10.1109/GlobalSIP45357.2019.8969441.

[59] Hui Zou e Trevor Hastie. «Regularization and Variable Selection via the Elastic Net». In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* (2005). DOI: 10.1111/j.1467-9868.2005.00503.x.