



Università degli Studi di Ferrara

UNIVERSITÀ DEGLI STUDI DI FERRARA
CORSO DI LAUREA IN INFORMATICA

*Reti Neurali Convolutionali per la
classificazione di dati genetici*

Relatore:

Prof. Fabrizio RIGUZZI

Correlatori:

Dott. Arnaud N. FADJA

Dott. Emiliano TRUCCHI

Prof. Giorgio BERTORELLE

Laureando:

Matteo DONATO

ANNO ACCADEMICO 2018 – 2019

Indice

	Page
1 Introduzione	5
2 Descrizione del problema	7
3 Tecnologie Utilizzate	9
3.1 TensorFlow	9
3.2 Anaconda	10
3.3 Reti Neurali	10
3.3.1 Introduzione	10
3.3.2 Differenza tra IA Classica e IA Subsimbolica	11
3.3.3 I Sistemi Neurali Biologici	13
3.3.4 Le componenti base di una rete neurale	14
3.3.5 La funzione di trasferimento	16
3.3.6 Architettura di una rete neurale	18
3.3.7 Reti che apprendono	19
3.3.8 Reti Neurali Convoluzionali (CNN)	23
4 Dataset	27
4.1 Lo script dataset_creator.py	29
4.2 I dataset utilizzati	31

5	Modello e analisi del codice	33
5.1	Architettura utilizzata	34
5.2	Allenamento e Test della rete	38
5.2.1	Caricamento del dataset e suddivisione in batch	38
5.2.2	Iperparametri e funzioni necessarie	40
5.2.3	Allenamento della rete	42
5.2.4	Interfaccia	44
5.2.5	Salvataggio e caricamento del modello	45
5.2.6	Test della rete	45
6	Prove e risultati	47
6.1	Prove effettuate	47
6.2	Risultati dei test	50
7	Conclusioni	51

Introduzione

Con l'aumento delle dimensioni dei dataset di dati genetici relativi alla variabilità nelle popolazioni, per i ricercatori, estrarre informazioni diventa un compito sempre più complesso. Per rimanere al passo con questo aumento di dati si stanno rapidamente sviluppando metodologie computazionali per l'inferenza genetica sulle popolazioni in modo da poter utilizzare al meglio la grande mole di dati a disposizione. Più in particolare, una delle tecniche che ha prodotto risultati migliori è l'apprendimento automatico supervisionato. Questa tecnologia, applicata a dati genomici di popolazione, ha dimostrato non solo di essere in grado di sostituire i classici metodi di analisi dei dati, ma ha dimostrato anche che in molti casi risulta essere più efficiente. Il Machine Learning supervisionato, quindi, si è rivelato essere uno strumento importante e poco utilizzato, e ha un enorme potenziale per il mondo della genomica evolutiva[1].

L'obiettivo di questo progetto di tesi è quello di testare l'usabilità di una di queste metodologie per la classificazione di dati genetici di popolazione. Più precisamente è stato scelto di usare una Rete Neurale Convoluzionale come metodo di Machine Learning.

Descrizione del problema

Lo scopo di questo progetto è quello di testare l'applicabilità di algoritmi dedicati alla classificazione di immagini nella risoluzione di uno dei problemi fondamentali nella genetica di popolazioni, ovvero la discriminazione tra processi casuali (neutrali) e di selezione naturale responsabili del pattern di variabilità osservato in un campione di individui per una certa porzione del genoma.

In genetica di popolazioni, la selezione positiva corrisponde ad uno specifico tipo di selezione naturale secondo il quale è favorito un fenotipo¹ estremo, rispetto ad altri fenotipi, facendo sì che la frequenza dell'allele² che lo determina si sposti nel tempo aumentando in frequenza. Sotto selezione positiva, l'allele vantaggioso (indipendentemente se dominante o recessivo) aumenta quindi in frequenza come conseguenza delle differenze di sopravvivenza e riproduzione tra i diversi fenotipi.

¹**Fenotipo:** In genetica, il complesso delle caratteristiche morfologiche e funzionali di un organismo, prodotto dall'interazione dei geni tra loro e con l'ambiente.

²**Allele:** In genetica si definiscono alleli, due o più varianti dello stesso gene che si trovano nella medesima posizione su ciascun cromosoma omologo. Gli alleli controllano lo stesso carattere ma possono portare a fenotipi diversi. Esistono alleli dominanti e recessivi: l'allele dominante determina il fenotipo degli individui eterozigoti e omozigoti; l'allele recessivo determina il fenotipo solo in presenza di due alleli recessivi.

Questo problema viene tipicamente affrontato ricorrendo all'uso di modelli matematici in grado di descrivere le attese in caso di evoluzione principalmente neutrale o in caso di presenza di selezione mediante statistiche riassuntive della variabilità genetica.

In questo lavoro, abbiamo invece utilizzato direttamente il pattern di variabilità genetica senza ricorrere a statistiche riassuntive o a modelli matematici propri della genetica di popolazione: il dato genetico è stato prima convertito in immagini (cap. 4) e poi analizzato mediante Reti Neurali Convoluzionali (CNN), una tecnologia computazionale innovativa per la classificazioni di immagini.

Il lavoro è stato diviso in due fasi: nella prima fase mi sono occupato della creazione di dati genetici simulati in un campione di individui sulla base di un modello di evoluzione della loro composizione genetica sia neutrale (NEUTRAL) che dovuto a selezione naturale (SELECTION) per l'allenamento, la valutazione e il test della CNN; mentre nella seconda fase mi sono occupato della creazione della CNN con successivi test ed opportuni miglioramenti.

Tecnologie Utilizzate

Tutti i codici sono stati scritti in Python e il framework usato per la creazione della rete neurale è TensorFlow. La tecnologia più importante che viene usata è quella relativa alle reti neurali, più in particolare parliamo di Reti Neurali Convoluzionali (CNN) che verranno spiegate in maniera approfondita in questo capitolo. Un altro strumento che è stato ampiamente utilizzato è Anaconda per la creazione di ambienti di sviluppo su cui testare i codici.

3.1 TensorFlow

TensorFlow è una libreria software open source usata per il machine learning. Il framework è stato sviluppato per il progetto Google Brain di Google che, nel 2015, ha rilasciato il codice come licenza open source Apache 2.0 (licenza libera). Oggi TensorFlow è utilizzato in molti ambiti scientifici e aziendali per svolgere funzioni di deep learning che è, appunto, il nostro caso.

3.2 Anaconda

Anaconda è un software in grado di creare ambienti di sviluppo virtuali "separati" dal resto del sistema operativo in modo tale da poter avere diversi ambienti installati contemporaneamente sullo stesso computer. L'utilità è quella di avere un ambiente contenente solo ed esclusivamente le risorse necessarie all'esecuzione di uno o più programmi. Ad esempio, con le nuove versioni di TensorFlow si potrebbero verificare incompatibilità con il codice perché alcune funzioni sono state sostituite, altre sono state eliminate ed altre ancora sono state rimosse quindi il codice potrebbe necessitare di piccoli aggiustamenti ogni volta che il framework viene aggiornato, per non dover fare continuamente queste modifiche ho usato un ambiente di sviluppo su Anaconda che mantiene la versione di TensorFlow con cui ho iniziato, ovvero la 1.10.

3.3 Reti Neurali

3.3.1 Introduzione

La computazione neurale prende ispirazione dai sistemi neurali biologici, dei quali cerca di modellarne la struttura e di simularne le funzioni di base. Solitamente i sistemi basati sulle reti neurali vengono analizzati in contrapposizione ai sistemi digitali classici di tipo Von Neumann ed il motivo è che si tratta di due approcci caratterizzati da filosofie distinte. I calcolatori digitali di tipo Von Neumann, infatti, sono caratterizzati da una CPU che racchiude tutta la capacità computazionale del sistema ed esegue le operazioni che sono state programmate per essere eseguite in maniera sequenziale. Possiamo quindi dire che il concetto di algoritmo come insieme di operazioni organizzate in una sequenza opportuna sta alla base di questo approccio.

La filosofia delle reti neurali, dall'altra parte, ispirandosi ai sistemi biologici, considera un numero elevato di processori che hanno una capacità computazionale elementare ovvero i neuroni artificiali o nodi che vengono connessi ad altre unità dello stesso tipo (**connessionismo**). Come le sinapsi che collegano i neuroni biologici anche i collegamenti tra neuroni artificiali non sono tutti uguali infatti ad ogni connessione viene assegnato un peso in modo tale che un neurone possa

influenzarne un altro in funzione della "forza" della connessione tra i due, proprio come nei sistemi neurali biologici in cui un neurone (pre-sinaptico) influenza un altro neurone (post-sinaptico) in funzione del potenziale post-sinaptico della sinapsi fra l'assone del neurone pre-sinaptico e il dendente del neurone post-sinaptico.

3.3.2 Differenza tra IA Classica e IA Subsimbolica

Più che sulla contrapposizione tra computer seriale e connessionismo è bene soffermarsi sulla contrapposizione tra intelligenza artificiale (IA) classica, legata storicamente alla logica dei calcolatori seriali e alla matematica computazionale e l'IA subsimbolica legata all'approccio connessionista ed alla computazione neurale [11].

L'IA classica si occupa della simulazione dei processi cognitivi superiori adottando un approccio di elaborazione simbolica. In base all'ipotesi del sistema simbolico fisico di Newell e Simon (1976) [10] un sistema di occorrenze di simboli collegate in modo fisico e processi che operano su tale struttura simbolica è necessario e sufficiente per il comportamento intelligente. Il comportamento intelligente, quindi, può essere tradotto in un algoritmo opportuno e riprodotto attraverso un calcolatore. Tale approccio metodologico ha portato allo sviluppo di sistemi in grado di giocare a scacchi e risolvere teoremi in quanto si pensava che questa fosse l'essenza del comportamento intelligente, mentre si ritenevano meno degni di attenzione processi quali per esempio il processamento di immagini.

In realtà un bambino di un anno è in grado di riconoscere oggetti o facce in modo più veloce ed accurato del più evoluto programma di IA implementato nel più veloce supercomputer esistente. Questa evidenza sottolinea da una parte che i compiti di elaborazione delle immagini visive non sono banali, dall'altra rinforza la convinzione che il cervello umano sia di gran lunga superiore ai sistemi intelligenti artificiali e che debba tale superiorità alle caratteristiche della sua struttura interna. Tale struttura, permette al sistema biologico di avere diverse proprietà significativamente diverse da quelle dei sistemi artificiali [8]:

- robustezza e tolleranza agli errori: le cellule nervose nel cervello muoiono ogni giorno senza avere effetti significativi sulla performance totale del sistema;

- flessibilità: la struttura si può facilmente adeguare ad un nuovo ambiente attraverso l'apprendimento;
- ha a che fare con informazioni confuse, probabilistiche, indefinite o inconsistenti;
- parallelismo;
- struttura piccola, compatta e che dissipa pochissima energia.

La simulazione della struttura dei sistemi neurali biologici potrebbe portare allo sviluppo di sistemi artificiali che conservino le caratteristiche del comportamento intelligente dei sistemi biologici. Ciò rappresenta un cambiamento di prospettiva piuttosto radicale rispetto alla computazione convenzionale. Il processo di calcolo nella computazione neurale è inteso come “metafora del cervello” in quanto tenta di modellarne la struttura e non più come “metafora del computer” dei sistemi artificiali classici che utilizzano una logica sequenziale e predeterminata [5].

La computazione neurale è detta anche **subsimbolica**: ciò vuol dire che il simbolo non è presente in modo esplicito in tali sistemi. Non esiste un algoritmo che traduca la realtà percepita, per esempio attraverso una telecamera, in un insieme di simboli che denotino gli oggetti del percetto e nemmeno un insieme di manipolazioni possibili di tali simboli, come per esempio una serie di regole di produzione.

La conoscenza nei sistemi neurali è **distribuita**: non è presente da qualche parte in modo esplicito come simbolo o insieme di simboli, ma è il risultato dell'interazione di un numero elevato di neuroni attraverso le connessioni fra essi. Tali concetti verranno ripresi più avanti e risulteranno più chiari dopo che sarà stata introdotta la struttura di una rete neurale. Per ora basti ricordare due aspetti della computazione neurale:

1. si contrappone alla computazione digitale classica dei calcolatori di tipo Von Neumann dal punto di vista strutturale. I sistemi neurali sono caratterizzati da un parallelismo massiccio e la conoscenza è distribuita all'interno di tali sistemi. I computer digitali, invece, centralizzano le risorse computazionali deputandole ad un singolo processore (CPU) e le operazioni devono essere eseguite in modo sequenziale;

2. l'IA si situa in una dimensione parallela a quella della progettazione di sistemi simbolici. Non è necessariamente una posizione alternativa, ma un modo diverso di affrontare il problema della costruzione di sistemi intelligenti: simulare la struttura (il cervello) diviene funzionale alla produzione di macchine intelligenti, la dove l'IA simbolica focalizzava l'attenzione più sulla simulazione del processo.

Infine è opportuno ricordare che la computazione neurale si serve, per la costruzione di modelli dei processi di percezione, memorizzazione e apprendimento, di un nuovo paradigma: il paradigma *connessionista*.

3.3.3 I Sistemi Neurali Biologici

Come già detto molte volte, la computazione neurale si ispira ai sistemi neurali biologici, per fare ciò è necessario comprendere qual è il funzionamento di un neurone biologico, quali sono le sue parti e come questi sono interconnessi tra loro. Un neurone può essere considerata l'unità computazionale elementare del cervello.

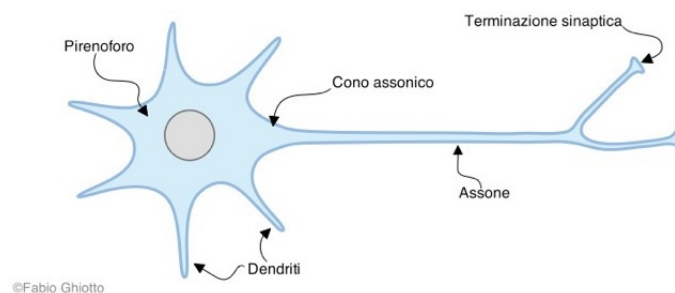


Figura 3.1: Rappresentazione schematica di un neurone biologico

Per analizzarne le caratteristiche anatomiche e funzionali si sceglierà come esempio un **neurone bipolare** (figura 3.1). La caratteristica principale del neurone è quella di generare un **potenziale elettrico** che si propaga lungo l'**assone** (l'output del neurone), fino a quando l'attività elettrica al livello del **soma** (corpo del neurone) supera una determinata **soglia**. L'input del neurone è un insieme di fibre chiamate **dendriti**: esse sono in contatto con gli assoni di altri neuroni dai quali ricevono i potenziali elettrici. Il punto di connessione fra un assone di un neurone e il dendrite di un altro neurone è chiamato **sinapsi**.

La sinapsi, fra le tante, ha anche la proprietà di modulare l'impulso elettrico proveniente dall'assone. Il potenziale elettrico generato da un neurone infatti è di tipo tutto-o-nulla: se l'attività elettrica del neurone supera una certa soglia si innesca l'impulso, altrimenti no; e la scarica non differisce per intensità da un neurone all'altro. Il potenziale si propaga lungo l'assone e giunge alla sinapsi con il dendrite di un altro neurone. Il potenziale post-sinaptico sul dendrite dipende dalle caratteristiche biochimiche della sinapsi. In presenza dello stesso potenziale pre-sinaptico, due sinapsi diverse generano potenziali post-sinaptici differenti. In altre parole *la sinapsi pesa il potenziale in ingresso modulandolo*. I potenziali post-sinaptici si propagano attraverso i dendriti del neurone; a livello del soma si sommano: solo se il risultato di tale somma è superiore ad una certa soglia il neurone innesca il potenziale che si propagherà attraverso il suo assone.

3.3.4 Le componenti base di una rete neurale

Fino ad ora si è parlato di reti neurali in termini di generici sistemi computazionali ispirati a sistemi neurali biologici. Si comincerà ora a descrivere che cos'è una rete neurale partendo dai suoi elementi costruttivi elementari.

I neuroni artificiali

Un neurone artificiale, che all'interno del connessionismo prende il nome di processing element (PE o nodo), è l'unità computazionale atomica di una rete neurale. Esso simula diverse funzioni di base del neurone biologico: valuta l'intensità di ogni input, somma i diversi input e confronta il risultato con una soglia opportuna.

Ingressi e uscita

Un neurone biologico, come si è detto, riceve diversi ingressi attraverso i dendriti. Così anche il neurone artificiale (PE) riceve diversi valori in ingresso (i_1, i_2, \dots, i_n) . Tutti gli ingressi vengono sommati ed il risultato costituisce il valore computato dal PE. Se tale valore supera una certa soglia, il neurone produce un segnale di output, o potenziale. Comparando l'uscita U con un opportuno valore di soglia q , il potenziale P sarà:

$$(1) \quad P = U - q$$

I pesi

Come detto nel capitolo precedente relativo ai Sistemi Neurali Biologici, le sinapsi **pesano** il potenziale in ingresso modulandolo, per simulare questo processo ad ogni ingresso del PE deve essere assegnato un **peso**, cioè un valore numerico che modula l'impatto che tale ingresso ha sulla somma totale per determinare il potenziale del PE. In altre parole ogni ingresso contribuisce in modo maggiore o minore a determinare il superamento del valore di soglia e l'innescò del potenziale. Alcuni di essi avranno un effetto maggiore sulla somma totale, altri addirittura avranno carattere inibitorio, cioè avranno l'effetto di abbassare il valore della somma totale degli ingressi e, di conseguenza, di diminuire la probabilità che essa superi il valore di soglia e si innesci un potenziale. Il peso di una connessione è un valore numerico

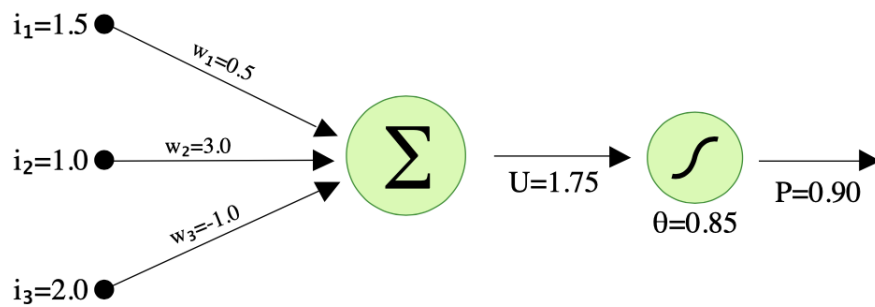


Figura 3.2: Il peso w_i ($i = 1, \dots, n$) determina quanto il relativo ingresso I_i contribuisca alla somma totale degli ingressi U . Più è elevato il peso, più l'ingresso corrispondente contribuirà ad elevare tale somma e, di conseguenza, più alta sarà la possibilità di superare la soglia q e di innescare il potenziale P .

per il quale moltiplicare il valore dell'ingresso. In questo modo l'ingresso avrà un effetto maggiore o minore sulla somma totale degli ingressi in funzione dell'entità del peso. La somma degli ingressi diventerà ora la **somma pesata** degli ingressi. Formalmente possiamo dire che:

$$(2) \quad U = i_1 \cdot w_1 + i_2 \cdot w_2 + \dots + i_n \cdot w_n$$

Matematicamente si può pensare all'ingresso e al peso corrispondente come a vettori del tipo $I = (i_1, i_2, \dots, i_n)$ e $W = (w_1, w_2, \dots, w_n)$. La somma pesata degli

ingressi sarà quindi il prodotto scalare di questi due vettori ($I \cdot W$)

3.3.5 La funzione di trasferimento

Si introdurrà ora un'altra proprietà del neurone artificiale ispirata ancora una volta ad una proprietà del neurone biologico. Si è detto che il neurone biologico, a livello del soma, somma tutti i potenziali post-sinaptici dei dendriti. In realtà tale somma non è proprio la somma algebrica dei valori di tali potenziali; diversi fattori, fra cui la resistenza passiva della membrana del neurone, fanno sì che il risultato della somma di tali valori non sia l'esatta somma algebrica, ma una funzione, di solito non lineare, di tale somma. In altre parole il neurone artificiale somma gli ingressi pesati e poi modifica il risultato in base ad una determinata funzione f . Tale funzione è detta **funzione di trasferimento** e, applicata all'uscita del PE determina il suo potenziale effettivo.

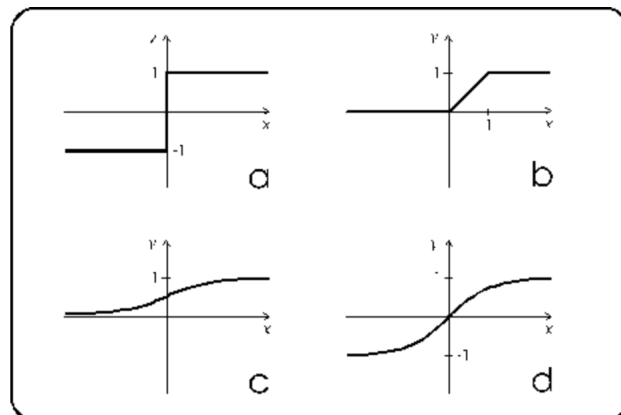


Figura 3.3: Le quattro funzioni di trasferimento più utilizzate

Le quattro principali funzioni di trasferimento sono:

- a) **funzione a gradino**: se il valore x in ingresso alla funzione è maggiore di 0 allora il valore restituito da $f(x)$ è 1, altrimenti restituisce -1 . Formalmente:

$$f(x) = \begin{cases} 1, & \text{se } x \geq 0 \\ -1, & \text{se } x < 0 \end{cases}$$

La funzione a gradino può assumere valori $(-1, 1)$, come quella descritta sopra, oppure valori binari $(0, 1)$. In questo caso $f(x) = 1$ se $x \geq 0$ mentre se $x < 0$ allora $f(x) = 0$

- b. **funzione lineare con saturazione**: se $0 \leq x \leq 1$ allora $f(x)$ diventa una funzione lineare e restituisce proprio il valore di x . Se invece $x > 1$ o $x < 0$ la funzione "appiattisce" i valori rispettivamente a 1 e 0. Formalmente:

$$f(x) = \begin{cases} 0, & \text{se } x < 0 \\ x, & \text{se } 0 \leq x \leq 1 \\ 1, & \text{se } x > 1 \end{cases}$$

- c. **funzione sigmoide (0,1)**: Se $x = 0$ $f(x) = 0.5$. Con l'aumentare del valore di x , $f(x)$ tende asintoticamente a 1, mentre con il diminuire di x , $f(x)$ tende asintoticamente a 0. Formalmente:

$$f(x) = \frac{1}{1 + e^{-x}}$$

La funzione sigmoide è continua e derivabile e per questo viene usata nei modelli di rete neurale nei quali l'algoritmo di apprendimento richiede l'intervento di formule in cui appaiono derivate.

- d. **funzione sigmoide (-1,1)**: Questo tipo di funzione ha una curva uguale alla precedente ma passa per l'origine quindi $f(x)$ può assumere anche valori negativi infatti restituisce valori compresi tra -1 e 1: con l'aumentare del valore di x la $f(x)$ aumenta in modo asintotico e tende ad 1 mentre con il diminuire del valore di x abbiamo che la $f(x)$ tende in modo asintotico a -1. Formalmente:

$$f(x) = \begin{cases} \frac{1}{(1+e^{-x})}, & \text{se } x \geq 0 \\ \frac{1}{(1+e^x)}, & \text{se } x < 0 \end{cases}$$

3.3.6 Architettura di una rete neurale

Dopo aver descritto i singoli elementi che vanno a comporre una rete neurale, si passerà ora a descrivere in che modo tutti questi elementi interagiscono tra loro e qual è il loro ruolo all'interno della struttura di una rete neurale.

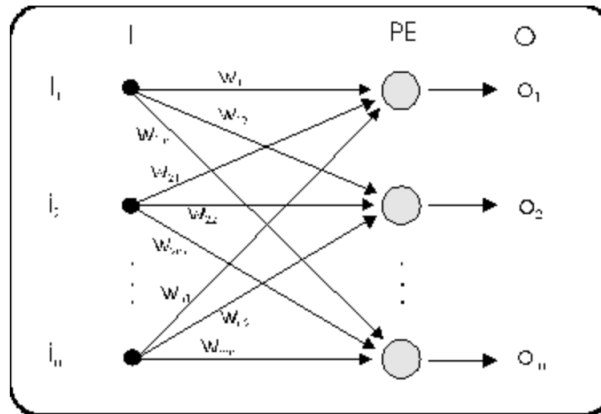


Figura 3.4: Struttura fondamentale di una rete neurale. Ogni ingresso i_k ($k = 1, \dots, n$) è connesso con tutti i nodi di uscita o_h ($h = 1, \dots, m$); i pesi w_{nm} determinano quanto ogni ingresso influisce sull'uscita di ogni nodo.

Si supponga di avere diversi ingressi e diversi nodi in modo tale che ciascun ingresso sia connesso a tutti i nodi. La figura 3.4 illustra questa struttura. Ogni ingresso i_k ($k = 1, \dots, n$) è connesso con tutti i nodi di uscita o_h ($h = 1, \dots, m$); ogni nodo o_h possiede tutte le proprietà descritte sopra nel paragrafo 3.4.4 (*I neuroni artificiali*) e svolge la sua attività parallelamente a quella degli altri nodi. Presentando un pattern in ingresso¹ si avranno dei valori per le uscite che dipendono sia dai valori in ingresso, sia dai pesi della rete. In generale un peso determinerà quanto l'ingresso relativo è in grado di influenzare un particolare nodo. L'insieme dei nodi della struttura di figura 3.4 prende il nome di **layer** (strato). Le reti neurali possono essere anche a più strati (figura 3.5), ogni strato aggiunto alla rete ne aumenta la capacità computazionale. Gli ingressi I sono valori numerici che

¹Presentare un pattern in ingresso significa sostanzialmente assegnare un valore numerico a ciascun ingresso. Si parla di pattern perché tali valori hanno senso tutti insieme, come pattern appunto.

vengono valutati dai pesi delle connessioni con il primo strato di nodi H (detto anche strato hide, o nascosto). Qui ogni nodo esegue la computazione così come descritta nel paragrafo 3.4.4 (*I neuroni artificiali*) ed eventualmente produce un potenziale che, a sua volta, si propaga verso i nodi dello strato di uscita O . Il potenziale prodotto dai nodi di uscita O costituisce l'uscita computata dalla rete neurale. Cambiando il modo in cui sono connessi i nodi l'uno all'altro, si cambia l'architettura della rete. Ciò non ha solo conseguenze di ordine pratico in quanto muta la capacità computazionale della rete, ma ha anche importanti conseguenze di ordine teorico che coinvolge il concetto di apprendimento, argomento della prossima sezione.

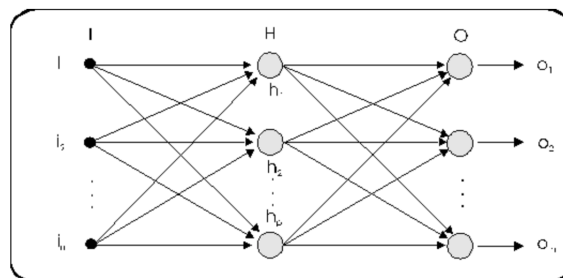


Figura 3.5: Rete a due strati. I : nodi d'ingresso. H : strato nascosto (hide). O : strato d'uscita

3.3.7 Reti che apprendono

Ora è necessario vedere in che modo una rete neurale apprende ad emettere risposte appropriate in relazione a determinati pattern d'ingresso, incominciando a definire il concetto stesso di apprendimento in ordine alla computazione neurale. L'apprendimento potrebbe essere definito come la modifica del comportamento in conseguenza dell'interazione con l'ambiente e il risultato dell'esperienza che determina l'emergere di nuovi pattern di risposta agli stimoli esterni (McCord, Nelson e Illingworth, 1991)[5]. Volendo adattare tale definizione ad una rete neurale, è necessario definire innanzi tutto cosa si intende per interazione con l'ambiente, stimoli esterni e pattern di risposta di una rete neurale. Poi bisogna descrivere in che modo la rete neurale accumula esperienza. Saranno questi gli argomenti dei paragrafi che seguono.

L'apprendimento

La capacità sorprendente di apprendere propria dei sistemi neurali biologici risiede nella loro grande plasticità a livello microstrutturale (Hebb, 1949)[3]. Le sinapsi modificano le loro proprietà in relazione a particolari stimoli esterni per far sì che il sistema neurale nel suo complesso produca risposte adeguate a determinati stimoli ambientali. Questa è la caratteristica a cui si ispirano i sistemi neurali artificiali che apprendono. Dire che una rete interagisce con l'ambiente significa sostanzialmente affermare che dall'ambiente riceve gli stimoli attraverso i suoi ingressi (figura 3.5 - *I*) e risponde attraverso la produzione di un pattern di uscita (figura 3.5 - *O*). La risposta della rete neurale deve essere modificata in modo da diventare il più appropriata possibile al pattern d'ingresso in quel momento. Questo viene ottenuto variando il valore dei pesi delle connessioni in modo che in presenza di un determinato pattern in ingresso alla rete si abbia una risposta corrispondente. La forma della risposta della rete, il pattern di uscita, può essere decisa a priori. In questo caso si parla di **apprendimento supervisionato**: l'operatore umano decide che in corrispondenza di un determinato pattern vi debba essere una determinata risposta. Altrimenti, se la forma della risposta non è nota a priori, si parla di **apprendimento non supervisionato**. La differenza fra i due tipi di apprendimento sta nell'algoritmo che si adotta per far apprendere la rete.

Algoritmi di apprendimento

L'algoritmo di apprendimento specifica in che modo vengano modificati i pesi in modo tale da far sì che la rete apprenda. Gli algoritmi di apprendimento si dividono in due classi: gli **algoritmi con supervisione** e quelli **senza supervisione**.

Algoritmi di apprendimento supervisionato

La logica su cui si fonda questo tipo di algoritmo è quella di modificare i pesi delle connessioni in modo tale che, dato un pattern d'ingresso alla rete, il pattern d'uscita corrisponda ad una forma già definita a priori. Il più celebre di questi algoritmi è senz'altro quello di Rumelhart e McClelland (1986)[7] chiamato **Backpropagation**.

Backpropagation e Stochastic Gradient Descent

L'algoritmo di Backpropagation è un algoritmo che aggiorna i pesi di una rete neurale in base all'errore della rete ovvero in base alla differenza tra output ottenuto ed output desiderato. Se consideriamo un output layer con un solo neurone l'**errore** del neurone di output all' n -esima iterazione (presentazione dell' n -esimo pattern del training set) è dato da:

$$e(n) = d(n) - y(n)$$

dove $d(n)$ è l'output atteso del neurone e $y(n)$ è l'output effettivamente ottenuto. Scegliamo come **loss function** l'*errore quadratico medio*, quindi se N è il numero totale di pattern del training set allora la loss function è data da:

$$E(n) = \frac{1}{2} \cdot \sum_{i=1}^N e_i(n) \quad \text{errore quadratico medio}$$

L'obiettivo dell'addestramento è quello di minimizzare $E(n)$ modificando opportunamente i parametri liberi della rete neurale ovvero i pesi e i bias che vengono aggiornati di pattern in pattern fino al raggiungimento di un'epoca². L'aggiustamento dei pesi viene fatto in base all'errore calcolato per ogni pattern presentato alla rete. Per minimizzare la funzione di costo E si adotta il metodo della discesa stocastica³ del gradiente. Il gradiente è la direzione di maggior crescita di una funzione (a più variabili) e dirigendoci nella parte opposta al gradiente tendiamo a far diminuire l'errore. Nel nostro caso il gradiente è dato da $\frac{\partial E(n)}{\partial w_j(n)}$ e gli aggiornamenti vengono fatti in senso opposto al gradiente:

$$\Delta w_j = -\eta \cdot \frac{\partial E(n)}{\partial w_j(n)}$$

dove η è il coefficiente di apprendimento, un valore compreso tra 0 e 1 che serve per modulare gli aggiornamenti dei pesi a valori più bassi in modo da aggiornare i pesi poco per volta e non rischiare di rimanere bloccati in un intervallo e non raggiungere mai il minimo. Calcolando ciascuna derivata del gradiente otteniamo che

$$\frac{\partial E(n)}{\partial w_j(n)} = -\delta(n) \cdot y_j(n)$$

²**Epoca:** presentazione completa del training set alla rete (Capitolo 4).

³**Stochastic:** i pesi vengono aggiornati dopo aver calcolato il gradiente su un singolo esempio o batch di esempi.

Il parametro $\delta(n)$ viene definito **gradiente locale** ed equivale a $\delta(n) = e(n) \cdot \varphi'[v(n)]$ dove φ è la funzione di attivazione e $v(n)$ è la somma pesata dell'output del layer precedente a quello di output.

Quindi la variazione di peso sinaptico tra il neurone j -esimo e il neurone di output relativo all' n -esimo pattern del training set è la seguente

$$\Delta w_j = \eta \cdot \delta(n) \cdot y_j(n)$$

L'algoritmo di backpropagation, inoltre, è chiamato così perché la procedura di aggiornamento dei pesi viene eseguita aggiornando prima i neuroni degli strati più vicini all'output layer procedendo verso l'input layer.

Adam Optimizer

Uno dei problemi che si possono riscontrare mentre si cerca di minimizzare la funzione di costo è quello di rimanere bloccati in un minimo locale e quindi di non raggiungere mai il minimo globale. Il metodo più utilizzato negli algoritmi di backpropagation è lo Stochastic Gradient Descent che è stato spiegato precedentemente ma esistono molte alternative e tra queste una delle migliori (se non la migliore) sembra essere l' **Adam Optimizer** in cui il parametro η varia ad ogni iterazione modulato da altri due parametri chiamati *momenti*. In questo modo si aumentano le probabilità di riuscire a raggiungere il minimo globale della funzione di costo.

Algoritmi di apprendimento non supervisionato

La forma dell'output non è conosciuta a priori e l'aggiustamento dei pesi della rete neurale avviene in base ad alcune valutazioni della qualità del pattern d'ingresso da parte delle funzioni dell'algoritmo stesso. Per esempio un pattern può avere più elementi attivi, diversi da zero, di un altro: i pesi della rete verranno modificati in modo tale che presentando in ingresso il primo dei due pattern si avrà una certa risposta, mentre presentando il secondo se ne avrà un'altra in base al criterio della numerosità degli elementi attivi. Per farlo l'algoritmo modifica i pesi in modo da enfatizzare le differenze tra i pattern in ingresso. La differenza fondamentale rispetto all'algoritmo di apprendimento supervisionato è che non è necessario sapere a priori la forma dell'output e quindi non è necessario un confronto uscita effettiva-uscita corretta. La conseguenza è che non è possibile stabilire a priori quale dei nodi di uscita corrisponderà a ciascuno dei pattern d'ingresso.

3.3.8 Reti Neurali Convoluzionali (CNN)

Le reti neurali convoluzionali (CNN) vengono considerate lo stato dell'arte per la risoluzione di problemi di classificazione, localizzazione ed identificazione di oggetti all'interno di immagini. Le CNN basano la propria efficacia sulla capacità di riconoscere elementi rilevanti attraverso l'applicazione di filtri (convoluzione). Inizialmente avremo filtri in grado di riconoscere elementi ricorrenti semplici come linee e segmenti per poi avere filtri in grado di riconoscere elementi sempre più complessi come ad esempio automobili o persone all'interno di un paesaggio. Vediamo ora quali sono gli elementi fondamentali di una CNN facendo riferimento all'architettura descritta in figura 3.6.

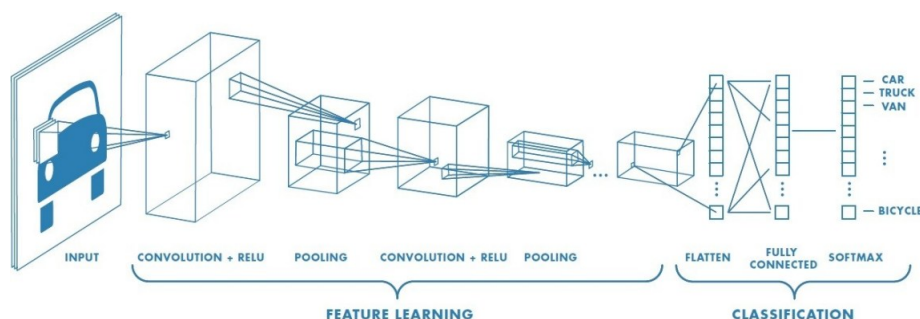


Figura 3.6: Architettura di una CNN classica

Convoluzioni e filtri

Una CNN è strutturata in maniera molto simile ad una normale rete neurale, abbiamo quindi in layer di input, dei layer nascosti che effettuano calcoli e un layer di output che restituisce un risultato. La differenza nelle CNN sta proprio nelle **convoluzioni** ovvero delle tecniche che consistono nell'applicazione di filtri (**kernel**) all'immagine da classificare. Lo scopo di questi filtri è quello di estrarre delle **feature** (caratteristiche) ricorrenti dalle immagini. Più preci-

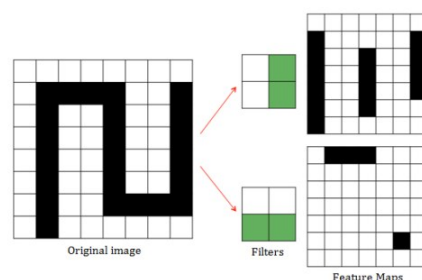


Figura 3.7: Filtri e feature map

samente la convoluzione consiste nell'applicare dei filtri facendoli scorrere sull'immagine, quindi il kernel sarà definito da una larghezza ed una altezza ed inoltre dovremo anche definire di quanti pixel per volta far scorrere il kernel una volta applicato in una regione dell'immagine. Ad esempio nella figura 3.7 abbiamo un'immagine 8x8 pixel a cui applichiamo dei filtri 2x2 che viene fatto scorrere con un passo (**stride**) di 1 sia in orizzontale che in verticale.

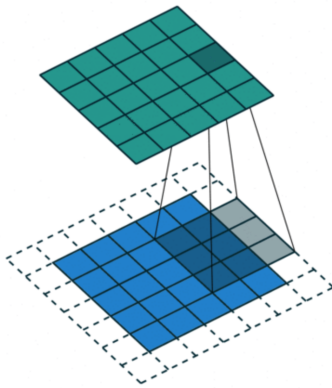


Figura 3.8: Esempio di applicazione di un kernel 3x3 con stride=1 su un immagine 7x7.

L'applicazione del kernel all'immagine consiste nel fare il prodotto scalare tra il kernel e la regione dell'immagine a cui fa riferimento, questo significa che ogni volta che viene applicato il kernel si ottiene un solo pixel in output indipendentemente dalla grandezza del kernel, infatti come possiamo vedere le feature estratte hanno dimensione 7x7. In realtà è possibile mantenere le dimensioni utilizzando lo **zero padding** che mette degli zeri attorno alla feature per mantenere le dimensioni originali. L'insieme di tutte le feature estratte è dato dalla **feature map**.

Funzione di attivazione

Come possiamo vedere dalla figura 3.6 la convoluzione è il primo layer di una CNN che comprende anche una funzione di attivazione, solitamente viene usata la **ReLU** (Rectified Lineaur Unit) come funzione di attivazione perché il calcolo della sua derivata è veloce.

Pooling Layer

Solitamente dopo la convoluzione viene applicato un layer di pooling che non costituisce nessun processo di apprendimento ma ha come funzione quella di ridurre le dimensioni della feature map senza però perdere informazioni riguardo le feature estratte. Il metodo con cui viene ridotta la feature map è molto simile all'applicazione del kernel durante la convoluzione: anche in questo caso andiamo a lavorare su porzioni dell'immagini e scorriamo l'immagine con uno stride ma in questo caso andiamo anche a decidere quale algoritmo usare per la riduzione della feature map. Nel nostro caso abbiamo usato l'algoritmo **max_pooling** che ritorna il valore massimo della regione presa in considerazione come descritto in figura 3.9.

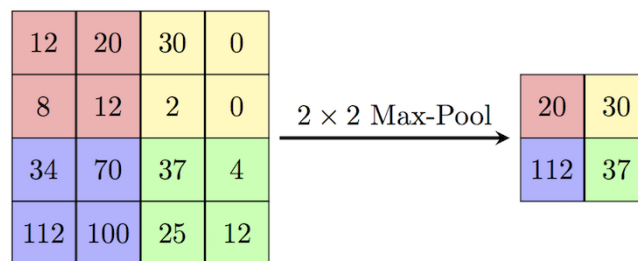


Figura 3.9: Semplice esempio di max_pooling

Architettura di una CNN

Abbiamo descritto gli elementi principali di una CNN ma la convoluzione e il pooling servono solamente ad estrarre le feature che dovranno poi essere classificate con le tecniche già spiegate nei capitoli precedenti. Inoltre va precisato che una CNN può avere un'architettura con più convoluzioni diverse tra loro susseguite sempre da un layer di pooling.

Dataset

Per l'allenamento, la valutazione e il test finale della rete neurale sono necessari tre appositi dataset distinti e siccome la rete neurale in questione è una CNN abbiamo bisogno necessariamente di dataset di immagini. I tre dataset di cui abbiamo bisogno sono quindi:

- il **training dataset** contenente le immagini per l'allenamento della rete;
- il **validation dataset** contenente le immagini per per valutare lo stato dell'allenamento alla fine di ogni epoca;
- il **test dataset** contenente le immagini per il test finale della rete.

In questo esperimento vengono usati tre dataset distinti per assicurare che l'alta precisione sia dovuta ad un corretto allenamento e non ad un overfitting della rete. La rete neurale dovrà classificare immagini di tipo SELECTION e NEUTRAL quindi ogni dataset conterrà metà immagini di un tipo e metà dell'altro. Per la creazione dei dataset necessari ho utilizzato il software `ms` (Hudson, 2002)[\[4\]](#) che consente la simulazione di dati genetici sulla base di un modello demografico senza selezione (i.e., NEUTRAL) impostato dall'utente e della sua versione modificata `mssel` (messo a disposizione da R. Hudson) che consente di aggiungere al modello

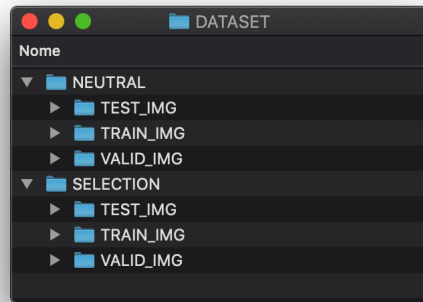


Figura 4.1: Struttura delle cartelle dei dataset generati da *dataset_creator.py*

demografico l'effetto della selezione naturale (SELECTION). Mediante questo software abbiamo simulato sequenze di DNA di lunghezza definita in un campione di individui presi da una popolazione con una dimensione definita di individui. In ciascuna simulazione, questo campione di sequenze di DNA possono essere collegate tra loro mediante un albero di coalescenza che risale nelle generazioni precedenti fino ad un antenato comune (Kingman, 1982)[6]. Lungo questo albero possono comparire delle mutazioni casuali che modificano una base della sequenza del DNA con una probabilità impostata dall'utente. Il pattern di variabilità finale di questo campione simulato di sequenze di DNA dipenderà quindi dalla dimensione della popolazione, dalla probabilità di mutazione e dalla storia demografica impostata nel modello. Nel caso di *mssel*, il pattern di variabilità delle sequenze di DNA nel campione dipenderà anche dall'intensità di selezione, altro parametro impostato nel modello. In entrambi i casi, il software produce matrici di sequenze di DNA, dove una dimensione corrisponde alla lunghezza della sequenza mentre l'altra al numero di individui simulati. Trattandosi di una simulazione, è facile tenere traccia dello stato ancestrale¹ e di quello mutato per ogni singola base della sequenza di DNA e quindi riportare le sequenze come stringhe di "0" (stato ancestrale) e "1" (stato derivato, ovvero in quella base è avvenuta una mutazione). La matrice di

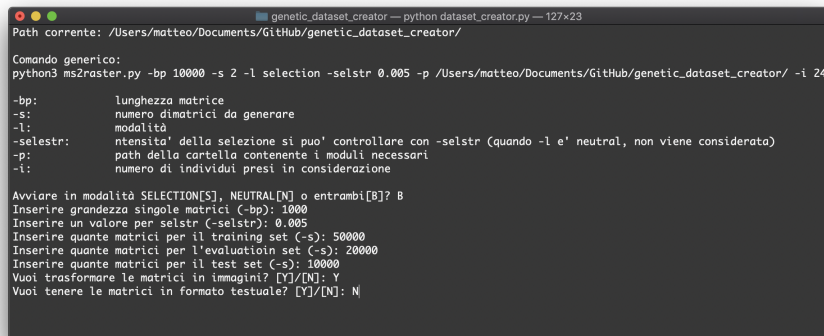
¹**Ancestrale:** in biologia, usato con riferimento alla serie degli antenati di un dato individuo, si dice di caratteri, somatici o psichici, che si suppone fossero presenti nei remoti antenati. Si considerano caratteri molti organi rudimentali, come il coccige dell'uomo che rappresenta un residuo della coda. (Treccani)

sequenze può quindi essere trascritta come una matrice binaria (stringhe di 0 e 1) che può essere facilmente convertita in un'immagine in bianco e nero.

Lo script `dataset_creator.py` genera le matrici di DNA usando `ms` o `mssel` per poi convertirle in immagini ed organizzarle nelle cartelle come in figura 4.1. Vedremo l'utilità di strutturare le cartelle del dataset in questo modo quando parleremo del modello della rete neurale. Vediamo ora nel dettaglio il funzionamento di `dataset_creator.py`.

4.1 Lo script `dataset_creator.py`

All'avvio dello script viene subito creata la struttura di cartelle come indicato in figura 4.1. Se all'avvio è già presente una struttura dati relativa ad un'esecuzione precedente viene sovrascritta con la struttura relativa all'esecuzione corrente. Una volta creata la struttura di cartelle vengono visualizzate a schermo una serie di informazioni relative al funzionamento di `ms`, in particolare viene mostrato il path corrente, un esempio di comando di `ms` e un elenco in cui vengono spiegati tutti i suoi parametri.



```
genetic_dataset_creator — python dataset_creator.py — 127x23
Path corrente: /Users/matteo/Documents/GitHub/genetic_dataset_creator/

Comando generico:
python3 ms2raster.py -bp 10000 -s 2 -l selection -selstr 0.005 -p /Users/matteo/Documents/GitHub/genetic_dataset_creator/ -i 24

-bp:      lunghezza matrice
-s:      numero di matrici da generare
-l:      modalità
-selestr: intensità della selezione si può controllare con -selstr (quando -l è 'neutral', non viene considerato)
-p:      path della cartella contenente i moduli necessari
-i:      numero di individui presi in considerazione

Avviare in modalità SELECTION[S], NEUTRAL[N] o entrambi[B]? B
Inserire grandezza singole matrici (-bp): 1000
Inserire un valore per selstr (-selstr): 0.005
Inserire quante matrici per il training set (-s): 50000
Inserire quante matrici per l'evaluation set (-s): 20000
Inserire quante matrici per il test set (-s): 10000
Vuoi trasformare le matrici in immagini? [Y]/[N]: Y
Vuoi tenere le matrici in formato testuale? [Y]/[N]: N
```

Figura 4.2: Interfaccia di input dei parametri per la creazione di un dataset con `dataset_creator.py`

Dopo aver stampato a schermo queste informazioni vengono subito chiesti tutti i parametri necessari per la creazione del dataset (Figura 4.2), in particolare viene chiesto:

1. se si vogliono creare immagini di un solo tipo o di entrambi i tipi;
2. il parametro *bp* che indica la lunghezza in basi delle sequenze di DNA da simulare e che corrisponde al numero di righe delle matrici generate;
3. il parametro *selstr* (selection strength) che è il parametro che controlla l'intensità della selezione a favore di un certo allele, è stato impostato ad un valore tale da rappresentare una selezione molto intensa per l'allele vantaggioso. In particolare, l'intensità della selezione viene sempre condizionata sulla dimensione effettiva N_e della popolazione in cui viene studiata in base alla relazione $S = 4N_e \cdot s$. Le nostre simulazioni sono state effettuate impostando una popolazione effettiva di 1000 individui e una *selstr* di 0.005 che corrisponde ad un valore S di 200, considerato molto alto (Tamuri, 2012)[9], in modo da testare inizialmente la rete neurale in condizioni semplificate²;
4. il numero totale di matrici per il training set: lo script ne genererà metà di un tipo e metà dell'altro;
5. il numero totale di matrici per la valutazione del modello: anche in questo caso metà di un tipo e metà dell'altro;
6. il numero di matrici per il test finale del modello: come sopra;
7. se si vogliono trasformare i dati generati in immagini;
8. se tenere o eliminare le matrici in formato testuale.

Una volta inseriti tutti i dati relativi alla creazione del dataset lo script creerà tutte le immagini richieste generando le matrici in formato testuale con *ms* per poi convertire tutte le matrici in immagini .PNG a due canali quindi in bianco e nero. Durante l'esecuzione viene anche creato un file *log.txt* contenente tutte le

²Future implementazioni di questo studio riguarderanno esperimenti con valori di selezione decrescenti per individuare il limite minimo in cui i due modelli (SELECTION e NEUTRAL) non riescono più ad essere discriminati con accuratezza.

caratteristiche del dataset. Convertendo i dati in immagini, inoltre, riduciamo le dimensioni del dataset del 90%.

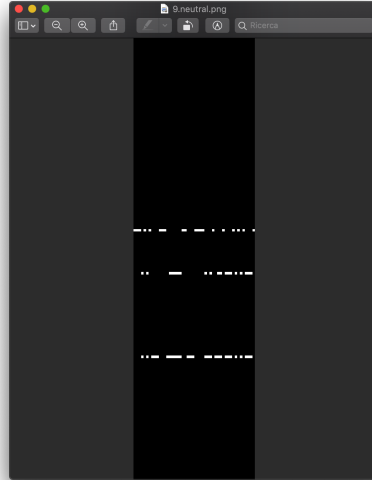


Figura 4.3: Una parte di un'immagine di tipo NEUTRAL generata da *dataset_creator.py*

4.2 I dataset utilizzati

Durante lo sviluppo della rete neurale sono stati creati decine di dataset diversi per testarne il corretto funzionamento anche nei casi più particolari. Tutti i dataset utilizzati per i test differiscono solamente nel numero di elementi che contengono ma le immagini sono state sempre create con le stesse caratteristiche:

- un *bp* di 1000;
- un *selstr* di 0.005;
- un numero di individui pari a 24.

I dataset saranno dunque composti da immagini B/N in formato .PNG con 48 colonne e 1000 righe.

Modello e analisi del codice

Dopo aver descritto l'architettura e il funzionamento di una rete neurale generica e descritto i dataset utilizzati, vediamo ora nel dettaglio la rete neurale che è stata creata appositamente per la classificazione dei nostri dati genetici. L'intero progetto si può suddividere in tre codici principali:

- *cnn_genetic.py*: codice principale contenente il *main* del progetto, gli altri moduli vengono utilizzati al suo interno e contiene anche le funzioni principali per il corretto utilizzo del modello della rete neurale più tutti gli iperparametri;
- *cnn_model_fn.py*: modulo contenente tutta l'architettura della rete neurale, al suo interno vengono quindi gestite tutte le componenti fondamentali di una CNN (capitolo 3.4.8);
- *load_dataset.py* : modulo che si occupa di reperire le immagini create con *dataset_creator.py* e di organizzarle in strutture dati opportune. Inoltre contiene anche una funzione per scorrere i dataset suddividendoli in batch di dimensione arbitraria.

Vediamo ora nel dettaglio i vari codici analizzando prima la parte più importante della rete neurale ovvero il modulo contenente l'architettura della rete neurale.

5.1 Architettura utilizzata

Come già detto l'architettura della rete neurale è definita all'interno del modulo *cnn_model_fn.py* che prende in input il il tensore¹ contenente le immagini da classificare e la modalità di esecuzione che può essere TRAIN o TEST. L'architettura utilizzata è decritta schematicamente in figura 5.1, vediamo ora ogni layer nel dettaglio.

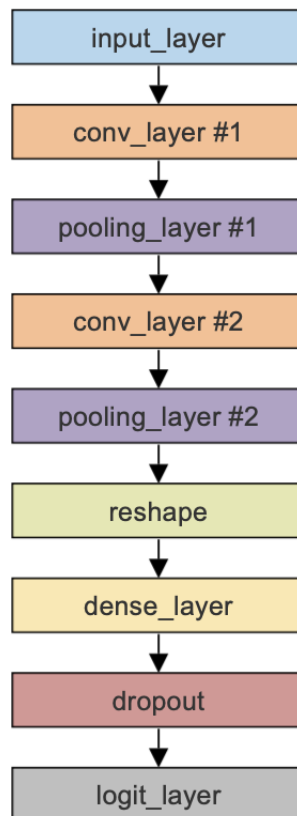


Figura 5.1: Tutti i layer dell'architettura utilizzata

¹In matematica, la nozione di tensore generalizza tutte le strutture definite usualmente in algebra lineare a partire da un singolo spazio vettoriale.

input_layer

Essendo il codice scritto con TensorFlow bisogna trasformare i dati da processare in tensori. Per fare ciò è stato usato `tf.reshape` che oltre a trasformare le immagini in un tensore permette anche di ridimensionarle, nel nostro caso però sono state mantenute le dimensioni originali per non perdere informazioni. Passiamo quindi alla funzione la variabile `X` contenente le immagini da trasformare e una lista di valori che indicano in ordine: il numero di immagini da elaborare (-1 indica che il numero può essere variabile e che quindi verrà calcolato volta per volta), la lunghezza e larghezza dell'immagine, il numero di canali dell'immagini che essendo in bianco e nero avranno un solo canale.

```
1 input_layer = tf.reshape(X, [-1, 1000, 48, 1])
```

conv_layer #1

Dopo aver strutturato i dati in maniera corretta è possibile procedere all'applicazione della prima convoluzione utilizzando `tf.layers.conv2d` che richiede i seguenti parametri:

- **inputs:** il tensore da classificare;
- **filters:** il numero di filtri che si vuole applicare, nel nostro caso quattro;
- **kernel_size:** la dimensione dei filtri da applicare, nel nostro caso è stato scelto un kernel 10x10;
- **strides:** indica il passo con cui si deve spostare il kernel, nel nostro caso si sposta di due pixel sia durante lo scorrimento orizzontale che durante lo scorrimento verticale;
- **padding:** come già visto l'applicazione dei filtri riduce le dimensioni dell'immagine, con questo parametro si può decidere di mantenere le dimensioni originali aggiungendo zeri agli estremi della matrice che rappresenta l'immagine (**same**) oppure tenere l'immagine ridimensionata (**valid**) che è il nostro caso.

Infine viene applicata la funzione di attivazione **ReLU**. Il codice per l'applicazione della prima convoluzione sarà quindi:

```
1 conv1 = tf.layers.conv2d(  
2     inputs=input_layer ,  
3     filters=4,  
4     kernel_size=[10, 10],  
5     strides=(2, 2),  
6     padding="valid",  
7 )  
8 conv1_relu = tf.nn.relu(conv1)
```

pooling_layer #1

L'algoritmo che è stato usato per il pooling è `max_pooling2d` con un filtro 2x2 e un passo di 2 che significa che le regioni raggruppate non si sovrappongono. Il codice per eseguire il pooling sarà quindi:

```
1 pool1 = tf.layers.max_pooling2d(  
2     inputs=conv1_relu ,  
3     pool_size=[2, 2],  
4     strides=2  
5 )
```

conv_layer #2 & pooling_layer #2

Viene applicata una seconda convoluzione allo stesso modo in cui viene applicata la prima ma questa volta vengono utilizzati 64 filtri con un kernel 5x5 mentre il secondo pooling viene applicato esattamente come il primo.

reshape

Questo layer serve a ridurre ulteriormente il carico computazionale trasformando il tensore multidimensionale usato finora in un tensore a due dimensioni dove tutte le feature relative ad una singola immagine vengono espresse da un singolo vettore invece che da diverse matrici. Avremo quindi un tensore a due dimensioni dove il numero di righe corrisponde al numero di immagini da classificare e ogni colonna è un vettore che rappresenta tutte le feature di quell'immagine. Per fare ciò viene prima ricavata la dimensione del tensore prodotto dal `pooling_layer #2` che viene salvata in una generica variabile `x` e poi viene applicata la funzione `tf.reshape`.

```
1 x = tf.TensorShape.as_list(pool2.shape)
2 pool2_flat = tf.reshape(pool2, [-1, x[1] * x[2] * x[3]])
```

Il reshape è stato fatto in questo modo per far sì che si adattasse automaticamente ad ogni modifica dei parametri in fase di calibrazione della rete.

dense_layer

Dopo aver estratto le feature e trasformato tutte le feature di ogni immagine in un singolo vettore si procede con la classificazione per mezzo del `dense_layer` che è stato creato con 1024 neuroni. La funzione di attivazione utilizzata è sempre la ReLU.

```
1 dense = tf.layers.dense(
2     inputs=pool2_flat,
3     units=1024,
4 )
5 dense_relu = tf.nn.relu(dense)
```

dropout

Per evitare l'overfitting è stata inserita una funzione di dropout che esclude in maniera random alcuni neuroni durante l'allenamento. Il modulo `cnn_model_fn.py` prevede anche l'inserimento del parametro `MODE` che viene usato solo per decidere se applicare o no il dropout che viene applicato solamente in modalità `TRAIN` e non in modalità `TEST`. È stato impostato una percentuale di dropout di 0.4 che significa che ogni neurone ha il 40% di probabilità di venire escluso durante l'allenamento.

```
1 if MODE == 'TRAIN':
2     dropout = tf.layers.dropout(
3         inputs=dense_relu,
4         rate=0.4,
5         training=True
6     )
7 else:
8     dropout = tf.layers.dropout(
9         inputs=dense_relu,
10        rate=0.4,
11        training=False
12    )
```

logits

L'ultimo layer è quello che restituisce i risultati grezzi prodotti dalla rete. Più precisamente è un dense layer ma di soli due neuroni (un neurone per le immagini SELECTION e uno per le immagini NEUTRAL) in cui ogni neurone calcola con quale probabilità ogni immagine potrebbe appartenere alla classe a cui fa riferimento.

```
1 logits = tf.layers.dense(  
2     inputs=dropout,  
3     units=2  
4 )  
5 return logits
```

Il modello si conclude restituendo la variabile `logits` che è una matrice $2 \cdot n$ dove n è il numero di immagini analizzate. Per ogni immagine viene quindi indicata la probabilità che quest'ultima possa appartenere alla classe SELECTION e la probabilità che possa appartenere alla classe NEUTRAL. Ora che sono stati prodotti e restituiti i risultati grezzi è necessario analizzarli per valutare ed ottimizzare il modello. Queste operazioni vengono eseguite all'interno del codice principale ovvero `cnn_genetic.py`.

5.2 Allenamento e Test della rete

Dopo aver analizzato nel dettaglio l'architettura della rete e i risultati da essa prodotti vediamo ora in che modo viene utilizzato ed ottimizzato il modello, vediamo quindi nel dettaglio il codice `cnn_genetic.py`. Il codice in questione permette di eseguire un nuovo allenamento da zero, continuare un allenamento iniziato e non terminato, perfezionare un vecchio allenamento o usare un modello già allenato per eseguire dei test. Per la suddivisione in batch ho usato un codice già scritto che mi è stato fornito dal tutor mentre per il caricamento delle immagini ho scritto la funzione da zero perché doveva essere coerente con il modo in cui ho creato il dataset.

5.2.1 Caricamento del dataset e suddivisione in batch

Una volta scelta la modalità di esecuzione è necessario recuperare dal dataset corretto tutte le immagini relative alla modalità selezionata. Come abbiamo visto

la quantità di immagini da processare è notevole (fino a 50.000 immagini per il training) quindi una volta caricato il dataset necessario abbiamo bisogno di suddividerlo in pacchetti più piccoli da passare alla rete chiamati **batch**.

Caricamento del dataset

Per fare un allenamento supervisionato non basta semplicemente caricare le immagini ma è necessario che ogni immagine sia associata alla sua classe di appartenenza, per questo motivo le immagini generate sono state raggruppate in cartelle in base alla classe di riferimento. Per il caricamento delle immagini del dataset si procede in questo modo:

1. si caricano tutte le immagini **SELECTION** che vengono aggiunte una alla volta in una lista. Contemporaneamente viene creata una lista con le etichette di ogni immagine ma essendo tutte immagini appartenenti alla stessa classe viene aggiunta sempre la stessa etichetta per ogni immagine aggiunta, in questo caso l'etichetta applicata è $[0, 1]$;
2. si ripetono le stesse identiche operazioni ma questa volta si carica la seconda metà del dataset con le immagini **NEUTRAL** applicando le etichette $[1, 0]$. Sia le immagini che le etichette vengono aggiunte alle stesse due liste;
3. alla fine del caricamento avremo una lista di immagini dove nella prima metà sono presenti tutte immagini **SELECTION** e nella seconda metà sono presenti tutte immagini **NEUTRAL**. Allo stesso modo la lista delle etichette conterrà per la prima metà tutte le etichette $[0, 1]$ corrispondenti alla classe **SELECTION** mentre per la seconda metà conterrà tutte le etichette $[1, 0]$ corrispondenti alla classe **NEUTRAL**;
4. avere le liste ordinate in due metà è un problema per l'allenamento della rete perché rischia di specializzarsi sulla seconda metà quindi dopo il caricamento vengono randomizzate entrambe le liste utilizzando gli stessi indici per non perdere la corrispondenza univoca immagine-etichetta;
5. una volta completata la randomizzazione vengono restituite le due liste alla funzione principale. L'intero processo può richiedere qualche minuto quin-

di verrà visualizzata una barra di avanzamento con il tempo rimanente necessario.

5.2.2 Iperparametri e funzioni necessarie

Il modulo `cnn_model_fn.py` contiene tutta l'architettura della rete ma non contiene le funzioni e i parametri necessari al suo addestramento e al suo utilizzo, queste funzioni saranno quindi definite all'interno del codice principale.

Iperparametri

Gli iperparametri sono i parametri indispensabili per il corretto funzionamento della rete neurale. Non c'è un metodo deterministico per definire quali iperparametri è meglio usare e per questo motivo vengono definiti come variabili in modo da poterli modificare facilmente durante i test. Dopo vari test gli iperparametri scelti sono:

- `learning_rate`: 0.001
- `batch_size`: 512
- `epoche`: 5

Oltre a questi sono stati impostati come iperparametri anche altezza e larghezza delle immagini in modo da poter adattare facilmente il codice ad ogni tipo di immagine.

Funzioni per l'allenamento e l'esecuzione

Come abbiamo visto l'ultimo strato della rete restituisce solamente dei dati grezzi ovvero delle percentuali di probabilità, questi dati vanno analizzati, valutati ed infine usati per migliorare l'efficienza della rete.

```
1 logits = cnn_model_fn(x, MODE)
```

La prima funzione implementata richiama `cnn_model_fn` e ne salva i dati in una variabile. La funzione va richiamata passando come argomenti la struttura dati contenente tutte le immagini da classificare, nel nostro caso è la variabile `x`, e la modalità con cui si intende eseguire il modello che può essere `TRAIN` oppure

TEST. La variabile `logits` conterrà dunque una sequenza di n coppie di valori dove n è il numero di matrici classificate e le coppie di valori fanno riferimento ai risultati prodotti dai due neuroni del `logit_layer`.

```
1 prediction = tf.nn.softmax(logits)
```

La funzione `prediction` mappa il vettore `logits` restituendo, per ogni classe, la probabilità che possa essere quella corretta in relazione all'altra classe. La somma di questi valori (in questo caso due per ogni immagine) sarà, quindi, uguale a 1.

```
1 loss = tf.reduce_mean(  
2     tf.nn.softmax_cross_entropy_with_logits_v2(  
3         logits=logits,  
4         labels=y  
5     )  
6 )
```

La funzione `loss` calcola l'errore di ogni esecuzione durante l'allenamento. Questo valore (diverso per ogni esecuzione) serve per saper se e quanto la rete sta sbagliando in modo da poter applicare l'ottimizzazione in maniera proporzionale all'errore. Questo significa che più l'errore è grande e più la modifica dei pesi sarà consistente.

```
1 optimizer = tf.train.AdamOptimizer(  
2     learning_rate=learning_rate  
3 )
```

La funzione `optimizer` è la funzione che si occupa dell'ottimizzazione e quindi della modifica dei pesi in relazione al parametro `loss`. L'algoritmo utilizzato è l' **Adam Optimizer**, un'alternativa al classico **Stochastic Gradient Descent** (discesa stocastica del gradiente) che è uno degli algoritmi di ottimizzazione più utilizzati. È stato usato l'Adam Optimizer perché è un algoritmo più efficiente in quanto non mantiene sempre lo stesso learning rate ma lo adatta di volta in volta.

```
1 train_op = optimizer.minimize(loss)
```

La funzione `train_op` è la funzione utilizzata per l'allenamento della rete infatti come possiamo vedere dal codice è costituita dalla funzione `optimizer` che modifica i pesi ad ogni iterazione basandosi sull'errore `loss`.

```
1 correct_predict = tf.equal(  
2     tf.argmax(prediction, 1),  
3     tf.argmax(y, 1)  
4 )
```

La funzione `correct_predict` compara le valutazioni della rete con le effettive etichette delle immagini classificate, se la previsione è corretta allora il risultato sarà un booleano `True`, altrimenti sarà `False`. Per ogni iterazione `correct_predict` conterrà i valori booleani relativi a tutte le immagini del batch dell'iterazione corrente.

```
1 accuracy = tf.reduce_mean(  
2     tf.cast(  
3         correct_predict,  
4         tf.float32  
5     )  
6 )
```

Infine la funzione `accuracy` calcola l'accuratezza della rete all'iterazione corrente nel caso dell'utilizzo durante l'allenamento, mentre calcola l'accuratezza definitiva della rete nel caso del test. Per fare ciò per prima cosa viene convertito `correct_predict` da un vettore di booleani ad un vettore di float in cui tutti i valori `True` vengono convertiti in valori 1.0 e tutti i `False` vengono convertiti in valori 0.0. Dopodiché `reduce_mean` calcola la media di questo vettore. In realtà sommando tutti i valori del vettore, contano solo gli 1.0 (cioè gli esempi classificati concretamente) questa somma viene poi divisa per il numero totale di elementi, si ha quindi la frazione degli esempi classificati correttamente in rapporto al totale degli elementi classificati (cioè l'accuracy). Formalmente abbiamo che l'accuracy è data da:

$$Acc = \frac{N_{True}}{N_{True} + N_{False}} = \frac{N_{True}}{N_{Tot}}$$

5.2.3 Allenamento della rete

Vediamo ora in che modo vengono usate queste funzioni e queste strutture dati per allenare la rete neurale. Innanzitutto è bene specificare che lo stesso training dataset viene utilizzato più volte per allenare la rete, il numero di volte in cui la rete usa il training dataset è chiamato **epoche**. Come abbiamo già visto, però, l'intero dataset è troppo grande per essere usato interamente e quindi lo andremo

a dividere in parti più piccole chiamate **batch**. Il numero di batch in cui viene suddiviso l'intero dataset è calcolato in funzione dalla grandezza scelta per il batch, ovvero il **batch_size** e come abbiamo visto è un iperparametro fissato a 512 elementi. La rete andrà quindi ad allenarsi su un solo batch di 512 immagini per volta, l'operazione di allenamento sui singoli batch è chiamata **iterazione**. Il numero di iterazioni che vengono svolte per ogni epoca è quindi dato in funzione del **batch_size**, più grande è il **batch_size** e meno iterazioni saranno necessarie per scorrere l'intero dataset e viceversa. Banalmente il numero di iterazioni viene calcolato in questo modo:

$$iterazioni = \frac{dataset_length}{batch_size} + 1 = \frac{50000}{512} + 1 = 98$$

Una volta stabilito il numero di epoche e iterazioni si procede con l'allenamento della rete. Dopo aver estratto dal dataset il batch corrente viene creato un dizionario contenente tutte le immagini del batch corrente e le rispettive etichette.

```
1 train_feed = {x: X_batch, y: Y_batch}
2
3 sess.run(train_op, feed_dict=train_feed)
4 los, acc = sess.run(
5     [loss, accuracy],
6     feed_dict=train_feed
7 )
8
9 if acc >= best_acc:
10     best_acc = acc
11     saver.save(sess, save_path)
```

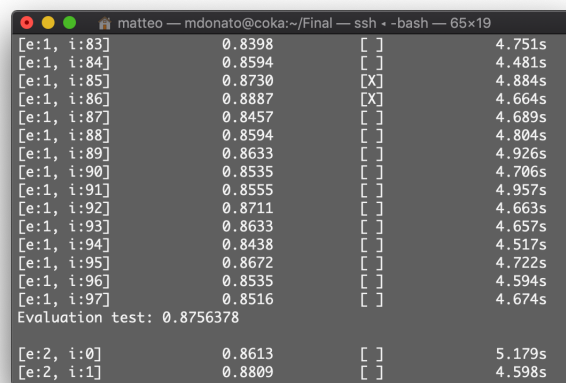
Successivamente il dizionario appena creato viene dato in input alla funzione **train_op**, una volta finita la classificazione vengono valutate le risposte della rete con le funzioni **loss** e **accuracy**. Essendo una rete a due classi è stato usato solamente il parametro **accuracy** per la valutazione, viene quindi confrontata l'accuracy ottenuta con la migliore ottenuta precedentemente (alla prima iterazione della prima epoca l'accuracy migliore è impostata a 0) e nel caso in cui sia migliore viene salvato il progresso fatto, in caso contrario si procede all'iterazione successiva.

Validation

Alla fine di ogni epoca viene eseguito un validation test (senza modifica dei pesi) sulla rete appena allenata utilizzando il validation dataset per controllare come si comporta la rete con immagini che non sono mai state usate per l'allenamento e che quindi la rete non ha mai avuto modo di elaborare. Se il validation test restituisce valori di accuracy in linea con quelli dell'allenamento significa che la rete sta generalizzando le risposte in maniera corretta. Se invece dovessimo riscontrare valori alti dell'accuracy durante l'allenamento ma poi il validation test restituisse valori inferiori significa che stiamo avendo problemi di overfitting.

5.2.4 Interfaccia

Durante l'allenamento vengono costantemente visualizzati a schermo i dati relativi all'andamento dell'allenamento, più precisamente viene visualizzato il numero dell'epoca corrente, il numero dell'iterazione corrente, l'accuracy relativa all'iterazione corrente, una checkbox che segnala quando si raggiunge un'accuracy migliore e quanto tempo è durata l'iterazione corrente. Inoltre alla fine di ogni epoca viene visualizzata l'accuracy del validation test.



```
matteo — mdonato@coka:~/Final — ssh — -bash — 65x19
[e:1, i:83]      0.8398      [ ]      4.751s
[e:1, i:84]      0.8594      [ ]      4.481s
[e:1, i:85]      0.8730      [X]      4.884s
[e:1, i:86]      0.8887      [X]      4.664s
[e:1, i:87]      0.8457      [ ]      4.689s
[e:1, i:88]      0.8594      [ ]      4.804s
[e:1, i:89]      0.8633      [ ]      4.926s
[e:1, i:90]      0.8535      [ ]      4.706s
[e:1, i:91]      0.8555      [ ]      4.957s
[e:1, i:92]      0.8711      [ ]      4.663s
[e:1, i:93]      0.8633      [ ]      4.657s
[e:1, i:94]      0.8438      [ ]      4.517s
[e:1, i:95]      0.8672      [ ]      4.722s
[e:1, i:96]      0.8535      [ ]      4.594s
[e:1, i:97]      0.8516      [ ]      4.674s
Evaluation test: 0.8756378

[e:2, i:0]      0.8613      [ ]      5.179s
[e:2, i:1]      0.8809      [ ]      4.598s
```

Figura 5.2: Un esempio di come viene visualizzato il progresso durante l'allenamento

File log

Durante l'esecuzione vengono anche creati due file log, uno in formato .txt che rispecchia l'output visualizzato a schermo e uno in formato .csv utile per la visualizzazione su fogli di calcolo. Ogni file log contiene nel nome la data e l'ora in cui è stato cominciato l'allenamento a cui fa riferimento.

5.2.5 Salvataggio e caricamento del modello

Come già detto il modello viene salvato automaticamente ogni volta che l'accuracy migliora. Questo è utile non solo per non dover fare l'allenamento da capo ogni volta che si vuole utilizzare la rete neurale ma è utile anche nel caso in cui si voglia interrompere l'allenamento per continuarlo in un secondo momento o nel caso in cui per qualsiasi motivo venga interrotto l'allenamento prima di essere concluso (crash del sistema, interruzione di corrente, ecc...). Quando si esegue il programma in modalità TRAIN è possibile scegliere se caricare un allenamento svolto in precedenza o se cominciare un nuovo allenamento da zero.

5.2.6 Test della rete

La modalità TEST esegue un ultimo ed ulteriore test sulla rete, questa volta però il test viene eseguito ad allenamento completato. Per il test finale viene usato il test dataset che contiene immagini mai utilizzate dalla rete durante l'allenamento sempre per scongiurare l'ipotesi di overfitting, questa volta però non viene diviso in batch ma viene utilizzato per intero.

```
1 saver.restore(sess, save_path)
2 test_feed = {x: X_test, y: Y_test}
3 test_acc = sess.run(accuracy, feed_dict=test_feed)
4 print('Testing Accuracy:' + str(test_acc))
```

Per eseguire il test della rete per prima cosa bisogna caricare il modello salvato durante l'allenamento, dopodiché si crea un dizionario con le immagini da classificare e le rispettive etichette e infine si usa la funzione `accuracy` per valutare le previsioni fatte dal modello. Il valore restituito dalla funzione `accuracy` è l'accuratezza raggiunta dal modello.

Prove e risultati

6.1 Prove effettuate

Per l'esecuzione del modello della Rete Neurale Convoluzionale è stato usato un ambiente di Anaconda installato su Coka, il cluster dell'Università di Ferrara ed è stata utilizzata la versione 1.10 di TensorFlow. Il modello è stato testato utilizzando sia le GPU che le CPU, i test fatti utilizzando le GPU non sono stati presi in considerazione perché sono stati ottenuti risultati nettamente migliori utilizzando solamente le CPU. Inizialmente si era pensato di effettuare il training con 10.000 immagini nel training set, così è stato creato il dataset ed è stato fatto un primo allenamento di 10 epoche. Essendo un problema di classificazione binaria è stata usata solamente l'accuracy come metrica di valutazione (p. 44).

Il grafico in Figura 6.1 indica i valori del validation test per ogni epoca, com'è possibile vedere l'accuracy raggiunge un valore massimo del 92,8% all'ultima epoca. Analizzando il grafico si può vedere come l'accuracy tenda a crescere ad ogni epoca quindi probabilmente se ci fossero state ulteriori epoche l'accuracy sarebbe migliorata ulteriormente. Per ottenere risultati migliori ho ripetuto l'allenamento quattro volte con quattro dataset diversi e tutti con un totale di 50 epoche così

da trovare il miglior compromesso tra dimensione del dataset e numero di epoche necessarie.

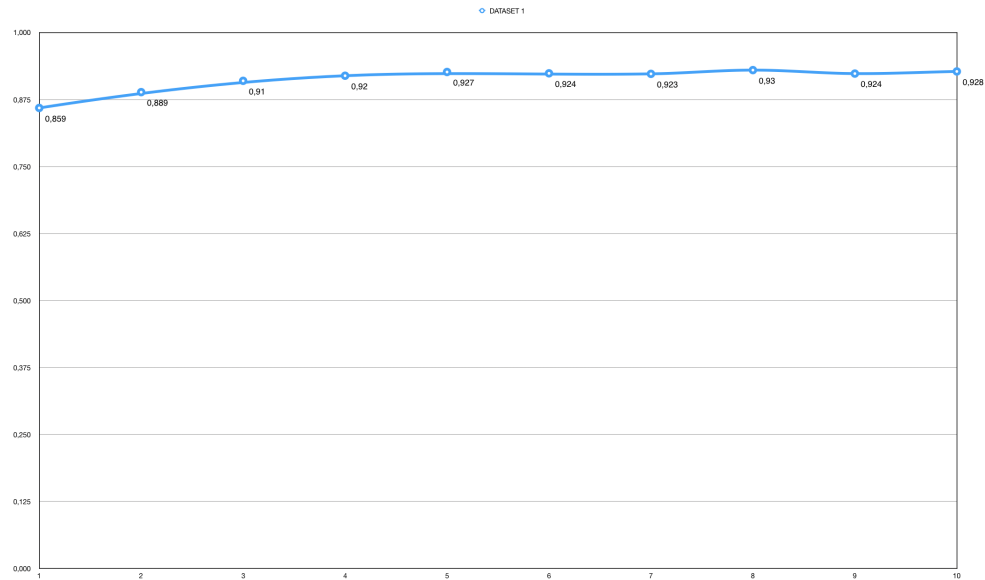


Figura 6.1: Andamento del training in base ai valori del validation test con un training set di 10.000 immagini.

Più precisamente i quattro dataset utilizzati hanno le seguenti caratteristiche:

DATASET	TRAIN	VALIDATION	TEST
DATASET 1	10K	2K	2K
DATASET 2	20K	5K	5K
DATASET 3	50k	10k	10k
DATASET 4	100k	20k	20k

Analizziamo ora l'andamento del training a seconda dei vari dataset (Figura 6.2).

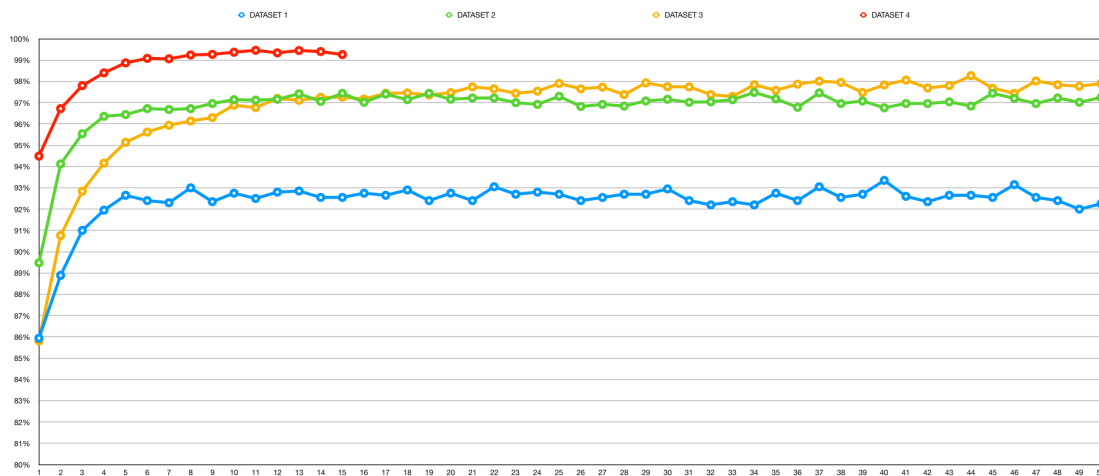


Figura 6.2: Comparazione dell'andamento dei quattro training in base ai valori dei validation test.

DATASET	MAX VALIDATION ACCURACY
DATASET 1	93,35%
DATASET 2	97,48%
DATASET 3	98,27%
DATASET 4	99,46%

La prima cosa che possiamo notare dal grafico in Figura 6.2 è che indipendentemente dal dataset, il modello tende a stabilizzarsi molto prima di arrivare a completare tutte e 50 le epoche richieste quindi possiamo dire che per allenare questo modello, indipendentemente dal dataset utilizzato, sono necessarie non più di 20 epoche. Solo nel caso del DATASET 3 abbiamo un aumento costante, anche se lieve, delle prestazioni, ma non è un aumento che può giustificare un allenamento di 50 epoche, infatti raggiunge il suo risultato migliore alla 44° epoca con un'accuracy del 98,27%. Nonostante gli ottimi risultati del DATASET 2 e del DATASET 3 appare evidente come il DATASET 4 raggiunga risultati nettamente migliori anche senza essere arrivato a concludere tutte e 50 le epoche richieste (essendo un dataset molto grande il tempo richiesto era eccessivo e avendo ottenuto

ottimi risultati anche solo in 15 epoche non era necessario arrivare a 50) infatti arriva ad avere un'accuracy del 99,45% alla 13° epoca.

6.2 Risultati dei test

Dopo aver analizzato l'andamento dell'allenamento del modello con vari dataset non ci resta che avviare il modello in modalità **TEST** per vedere gli effettivi risultati della rete allenata con un dataset che non è mai stato elaborato dalla rete. Nonostante il validation dataset e il test dataset siano completamente differenti

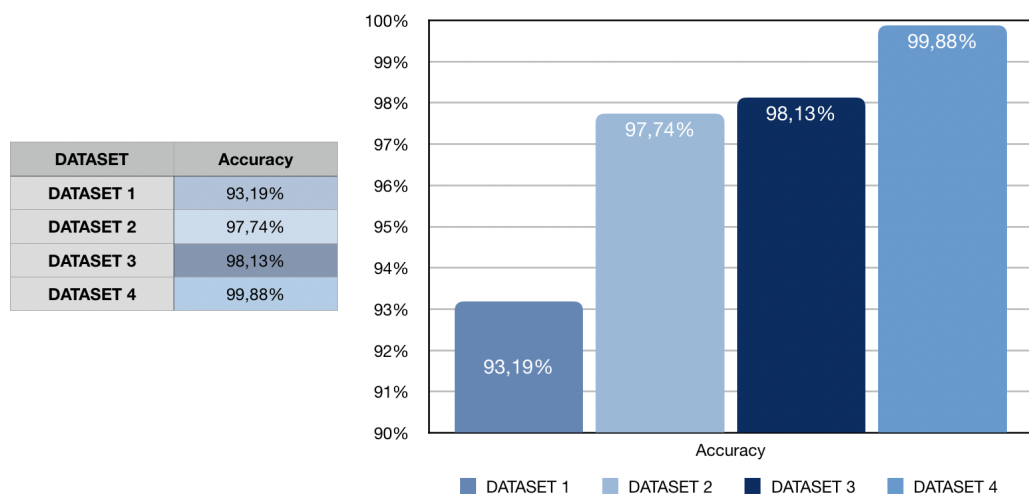


Figura 6.3: Risultati dei test fatti sul modello allenato con i quattro differenti dataset

tra loro, le modalità con cui vengono usati sono le medesime infatti come possiamo vedere il validation test fatto durante l'allenamento rispecchia appieno quelli che poi sono i risultati effettivi della rete. Possiamo quindi confermare che il risultato migliore è stato ottenuto con un dataset di training di 100.000 immagini con circa 15 epoche.

Conclusioni

Durante tutto il corso della tesi abbiamo visto cosa si intende per selezione positiva; quali erano i dati a nostra disposizione e come sono stati utilizzati per la creazione di dataset; abbiamo visto la teoria che sta alla base delle reti neurali e delle reti neurali convoluzionali per poi utilizzare questa tecnologia per creare un modello ad hoc in grado di classificare i dati in nostro possesso. Il modello si è da subito dimostrato in grado di riconoscere e distinguere i dati con una precisione iniziale del 92% circa, precisione che è poi aumentata grazie alle modifiche sul codice e sui dataset fino ad ottenere una precisione superiore al 99%.

L'utilizzo del Machine Learning per questo tipo di problema è già stato testato, ottimi risultati sono stati ottenuti anche con il software **lmaGene**^[2] che usa una Rete Neurale Convoluzionale con un'architettura molto simile a quella appena vista ma differisce di molto nella trasformazione dei dati in immagini, mentre noi trasformiamo i singoli dati in immagini di dimensione 48x1000 pixel, **lmaGene** trasforma i singoli dati in immagini di 128x128 pixel, questo comporta non solo una riduzione delle dimensioni del dataset (in termini di spazio occupato), ma riduce anche il carico computazionale con una conseguente riduzione dei tempi di allenamento ed elaborazione. Nonostante l'approccio diverso i risultati in termini di

precisione sono quasi identici visto che nel nostro caso raggiungiamo un'accuracy di 99,88% e lmaGene arriva a 99,9%.

Sviluppi futuri di questo progetto saranno volti a testare il modello con immagini create con parametri differenti esplorando le performance della Rete Neurale Convoluzionale rispetto a variazioni di alcuni parametri come ad esempio L'Intensità di selezione (selstr) oppure utilizzando modelli demografici diversi rispetto a quello a popolazione costante utilizzato in questo progetto. Successivamente il modello verrà testato con dati di popolazioni realmente esistenti così da poter confrontare le performance del sistema sia con dati simulati che con dati reali per poi poter confrontare l'efficienza di entrambi i metodi.

Ringraziamenti

Desidero ringraziare innanzitutto il relatore di questa tesi, il professor Fabrizio Riguzzi, per avermi dato l'opportunità di lavorare a questo progetto. Ringrazio, inoltre, il Dottor Arnaud N. Fadjia per l'enorme supporto datomi durante tutta la durata del progetto.

I miei più preziosi ringraziamenti vanno a mia mamma Sabrina, che grazie ai suoi sforzi e insegnamenti mi ha accompagnato fino al raggiungimento di questo traguardo, e alla mia ragazza Ionela per essermi sempre stata vicino durante questo percorso, soprattutto nei momenti più bui.

Un ringraziamento speciale va a tutta la mia famiglia che mi ha sempre sostenuto ma che soprattutto non ha mai perso occasione di chiedermi come andavano gli esami, soprattutto fuori sessione.

Infine vorrei ringraziare tutti i miei amici per essermi sempre stati vicino e per avermi sempre supportato.

Matteo Donato
17 dicembre 2019

Bibliografia

- [1] Daniel R.Schrider; Andrew D.Kern. “Supervised Machine Learning for Population Genetics: A New Paradigm”. In: *Trends in Genetics* 34.8 (2018), pp. 301–312.
- [2] Luis Torada; Lucrezia Lorenzon; Alice Beddis; Ulas Isildak; Linda Pattini; Sara Mathieson; Matteo Fumagalli. “ImaGene: a convolutional neural network to quantify natural selection from genomic data”. In: *BMC Bioinformatics* 20.9 (2018), p. 337.
- [3] D.O. Hebb. *The organization of behavior; a neuropsychological theory*. lawrence erlbaum associates, 1949.
- [4] Richard R. Hudson. “Generating samples under a Wright–Fisher neutral model of genetic variation”. In: *Bioinformatics* 18.2 (2002), pp. 337–338.
- [5] M. McCord Adams; Nelson; Illingworth. *A practical guide to neural nets*. Addison-Wesley, 1991.
- [6] J.F.C. Kingman. “The Coalescent”. In: *Stochastic Processes and their Applications* 13 (1982), pp. 235–248.
- [7] D.E. Rumelhart; J.L. McClelland. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations*. Cambridge: MIT Press, 1986.

- [8] J.A. Hertz; A.S. Krogh; R.G. Palmer. *Introduction to the Theory of Neural Computation*. Santa Fe Institute Studies on the Sciences of Complexity. CRC Press, 1991.
- [9] Asif U. Tamuri; Mario dos Reis; Richard A. Goldstein. “Estimating the Distribution of Selection Coefficients from Phylogenetic Data Using Sitewise Mutation-Selection Models”. In: *Genetics* 190.3 (2012), pp. 1101–1115.
- [10] A. Newell; H.A. Simon. “Computer science as empirical inquiry: symbols and search”. In: *Communications of the ACM* 19.13 (1976), pp. 113–126.
- [11] Gianluca Smeraldi. *Introduzione alle Reti Neurali*. URL: <http://www.emernet.it/TR9601.htm>.