

Università degli Studi di Padova

DIPARTIMENTO DI MATEMATICA “TULLIO LEVI-CIVITA”

CORSO DI LAUREA IN INFORMATICA



Utilizzo di Deep Learning per il rilevamento di
anomalie elettrocardiografiche

Tesi di Laurea

Relatore

Prof. Tullio Vardanega

Laureando

Oscar Konieczny

Matricola 2042335

ANNO ACCADEMICO 2023-2024

“I disagree strongly with whatever work this quote is attached to.”

— Randall Munroe

Ringraziamenti

/

Padova, Dicembre 2024

Oscar Konieczny

Sommario

/

Indice

1	Contesto aziendale	1
1.1	L'azienda	1
1.2	Tipologia di clientela	1
1.3	Organizzazione aziendale	2
1.4	Processi aziendali	3
1.5	Tecnologie e strumenti	4
1.5.1	Tecnologie utilizzate	4
1.5.2	Strumenti di supporto ai processi	5
1.6	Propensione all'innovazione	8
2	<i>Stage</i> proposto	9
2.1	Gestione aziendale degli <i>stage</i>	9
2.2	Descrizione progetto	10
2.3	Obiettivi	11
2.4	Vincoli	12
2.4.1	Vincoli tecnologici	12
2.4.2	Vincoli temporali	12
2.4.3	Vincoli organizzativi	12
2.5	Pianificazione delle attività	13
2.6	Motivazione della scelta	14
3	Svolgimento dello <i>stage</i>	16
3.1	Analisi dei requisiti	16
3.2	Progettazione	18
3.2.1	Visione d'insieme	18

3.2.2	Addestramento di modelli	19
3.2.3	Gestione del <i>dataset</i>	20
3.2.4	Valutazione dei modelli	23
3.3	Codifica	24
3.3.1	Prototipo	24
3.3.2	Gestione del <i>dataset</i>	26
3.3.3	Addestramento di reti neurali artificiali	28
3.3.4	Menzioni onorevoli	30
3.4	Verifica e validazione	31
3.5	Risultati raggiunti	31
3.5.1	Piano qualitativo	31
3.5.2	Piano quantitativo	33
4	Retrospettiva	i
4.1	Conseguimento degli obiettivi	i
4.1.1	Obiettivi aziendali	i
4.1.2	Obiettivi personali	i
4.2	Competenze acquisite	i
4.3	Divario tra università e lavoro	i
	Glossario	ii
	Bibliografia	v
	Sitografia	vi

Elenco delle figure

1.1	Rappresentazione del <i>framework</i> Scrum utilizzato in azienda . . .	4
1.2	Esempio di gestione delle attività con <i>Jira</i>	5
1.3	<i>Screenshot</i> di una <i>repository</i> in <i>Bitbucket</i>	6
1.4	Pagina principale del <i>Confluence</i> di M31 <i>Academy</i>	7
1.5	<i>Screenshot</i> di un canale di comunicazione di M31 <i>Academy</i> . . .	7
2.1	Rappresentazione dei punti principali del progetto	10
3.1	Architettura generale con le principali tecnologie utilizzate . . .	18
3.2	Confronto tra <i>Sequential API</i> e <i>Functional API</i>	19
3.3	Diagramma delle classi di addestramento di reti neurali artificiali	20
3.4	Diagramma delle classi di gestione del <i>dataset</i>	23
3.5	Cattura dello schermo rappresentante il <i>file</i> di <i>config</i>	32
3.6	Cattura dello schermo rappresentante i <i>file</i> di <i>output</i>	32

Elenco delle tabelle

2.1	Lista dei vari obiettivi aziendali.	11
3.1	Tabella contenente le varie <i>user story</i>	17

3.2	Tabella contenente le prestazioni del modello binario.	33
3.3	Tabella contenente le prestazioni del modello multi-classe. . . .	33

Elenco dei codici sorgenti

3.1	Codice della variabile del generatore	24
3.2	Codice della funzione che definisce il generatore	25
3.3	Parte di codice che sfrutta il comando <code>yield</code>	25
3.4	Codice delle funzioni che ottengono gli effettivi dati	26
3.5	Codice del generatore di dati che include informazioni aggiuntive	27
3.6	Codice del generatore di dati attraverso <i>file</i> DICOM	28
3.7	Esempio di definizione di un modello con la <i>Sequential</i> API . . .	29
3.8	Esempio di definizione di un modello con la <i>Functional</i> API . .	29
3.9	Esempio di funzione per la valutazione dei modelli	30

Capitolo 1

Contesto aziendale

1.1 L'azienda

M31 S.r.l. è un'azienda italiana nata nel 2007 con sede a Padova. Essa si specializza in ingegneria all'avanguardia su una variegata gamma di ambiti, come ad esempio: *mechanical & robotics*, *software engineering*, *artificial intelligence*, *IoT* e molto altro. L'azienda affianca i suoi clienti con cui svolge progetti, partendo dalla fase esplorativa fino alla fase di industrializzazione e certificazione. Nei diciassette anni di operazione, M31 ha realizzato più di cento unici progetti di innovazione che hanno avvantaggiato proprie startup, imprese del territorio e grandi gruppi.

1.2 Tipologia di clientela

La clientela di M31 è abbastanza variegata, ma i settori in cui opera e ha operato sono:

- **Biomedicina:** per la creazione di strumenti diagnostici, piattaforme di *liquid handling* e sistemi di monitoraggio remoto.
- **Monitoraggio e Sicurezza:** sistemi che variano da monitoraggio, antintrusione, videosorveglianza e sensori di allarme.
- **Applicazioni industriali:** per l'evoluzione di processi produttivi, in ambiti come quello delle lavorazioni meccaniche o tessili.

- ***Automation & smart cities:*** per l'automatizzazione di processi in vari ambienti, che siano domestico, di assistenza sanitaria oppure di processi fotografici.
- ***Cloud & digital twins:*** lo sviluppo di una controparte digitale di un prodotto o processo sul *cloud*.
- ***Racing & automotive:*** nell'ambito di vari sistemi e della telemetria per vari sport.

1.3 Organizzazione aziendale

L'organizzazione aziendale di M31 è suddivisa in diversi settori, questi sono: *governance*, *R&D team*, *marketing & sales* e *procurement & supply chain*.

Governance

Questo settore si occupa di amministrare l'azienda, questo avviene in vari modi: presa di decisioni per la direzione dell'azienda, gestione e manutenzione delle risorse finanziarie e la gestione e organizzazione del capitale umano.

I ruoli che possiamo trovare in questo ramo sono principalmente: il *CEO*, anche chiamato amministratore delegato, *accounting specialist* e *HR manager*.

Marketing & sales

Questo è il settore che si occupa di aumentare la visibilità e rapporti con possibili clienti, ma anche di ricercare quali sono le richieste attuali del mercato.

Questo *team* non è relativamente grande, nonostante ciò è possibile notare la presenza del *CBO*, anche detto direttore commerciale in lingua italiana.

Procurement & supply chain

Questa area aziendale si occupa di far sì che le operazioni aziendali possano procedere in modo adeguato, garantendo l'approvvigionamento delle risorse necessarie e la gestione della catena di fornitura.

R&D team

Infine, passiamo al settore aziendale dove sono stato inserito, ovvero il reparto di ricerca e sviluppo. Questo settore è quello più grande dell'azienda, ed è qui che i prodotti effettivi vengono sviluppati.

Ci sono varie aree di specializzazione in questo comparto, esse sono divise in una sorta di gruppi di lavoro, che possono o meno coordinarsi per lavorare insieme ai progetti aziendali attualmente in sviluppo.

Personalmente sono stato inserito sotto al *team* di *computer vision* e, come ogni altro stagista, anche sotto allo *stage coordinator*.

1.4 Processi aziendali

Durante il mio periodo di *stage* ho potuto assistere a come vengono gestiti i progetti all'interno dell'azienda. Viene utilizzata la metodologia *Agile*, ma in particolare il *framework* Scrum. Attraverso questo *framework*, i progetti vengono suddivisi in varie iterazioni denominati *sprint*, ad ognuno di esse, viene assegnata una certa durata di tempo, che può variare da una o più settimane.

Ogni *sprint* inizia con un *sprint planning*, nel quale vengono definite le attività da svolgere nello stesso *sprint*, queste attività vengono assegnate ai vari membri dello specifico *team* che si occupa di quel progetto.

All'inizio di ogni giornata lavorativa, viene eseguito il *daily stand-up*, nel quale viene discusso il procedimento dei lavori; in particolare viene discusso quello che è stato fatto la giornata precedente e cosa verrà fatto nella giornata attuale.

A fine di ogni *sprint* vengono fatti lo *sprint review* e lo *sprint retrospective*, il primo ha la finalità di valutare se tutte le attività sono state svolte e che siano state fatte correttamente, mentre il secondo considera le problematiche riscontrate durante lo *sprint*, definendo possibili miglioramenti al *way of working*, nel caso ne fosse necessario.

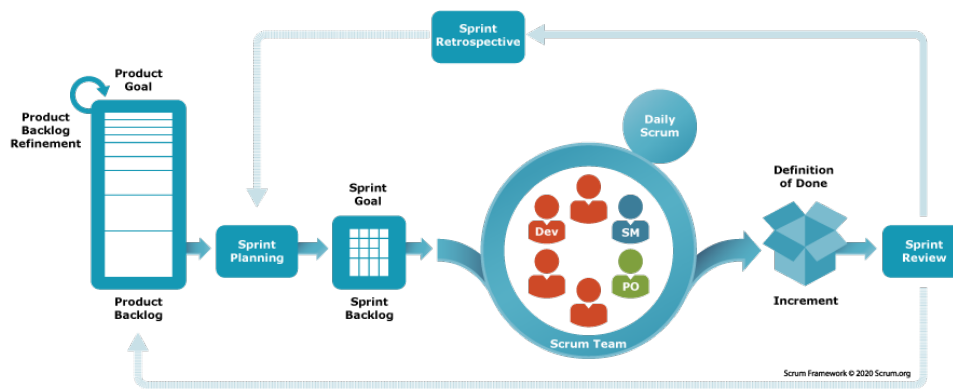


Figura 1.1: Rappresentazione del *framework* Scrum utilizzato in azienda
Fonte: [scrum.org](https://www.scrum.org)

1.5 Tecnologie e strumenti

1.5.1 Tecnologie utilizzate

In azienda vengono usate svariate tecnologie, questo è dovuto principalmente dal fatto che vengono svolti più progetti allo stesso tempo. Non avendo familiarità sugli altri progetti, esporrò le tecnologie che furono a me più vicine.

Python

Python è un linguaggio di programmazione ad alto livello, interpretato e di uso generale. È attualmente il linguaggio più popolare¹ al mondo, questo è dovuto dalla semplicità di utilizzo e dal buon supporto di librerie di terze parti, che semplificano ambiti come ad esempio: *web development*, *data science*, *machine learning*_G e *scripting* di operazioni ripetitive. Perciò non stupisce l'utilizzo di questo linguaggio per l'ambito di *machine learning* e per manipolare i *dataset*.

TensorFlow

TensorFlow è una libreria *open-source* sviluppata da *Google* per il *machine learning* e il *deep learning*_G. È una delle librerie più popolari per la creazione e adde-

¹Fonte: <https://www.tiobe.com/tiobe-index/>

stramento di modelli basati su tecniche di *machine learning*. *TensorFlow* integra al suo interno librerie per sfruttare a pieno processori specifici, come ad esempio le *GPU* e le *TPU*, permettendo così di utilizzare risorse computazionali più adeguate, per questo ambito, in modo semplice.

Keras

Keras è un'altra libreria *open-source* per il *deep learning*, sviluppata per essere semplice e modulare. Di recente è stata riscritta basandosi su *TensorFlow* e, come quest'ultima, permette di creare e addestrare modelli di intelligenza artificiale, ma rende il processo più rapido e intuitivo. Keras supporta una variegata tipologia di reti neurali pronte all'uso, come ad esempio reti neurali convoluzionali e ricorrenti, lasciando comunque la possibilità di modificare qualsiasi cosa si voglia adattare meglio alle proprie esigenze.

1.5.2 Strumenti di supporto ai processi

Jira

Jira è un software proprietario sviluppato da *Atlassian*, è utilizzato come Issue Tracking System (ITS), viene utilizzato per gestire le attività dei vari progetti con la metodologia *Agile*. All'interno di questo software si possono dedicare delle *board* per i vari *sprint* che vengono svolti.

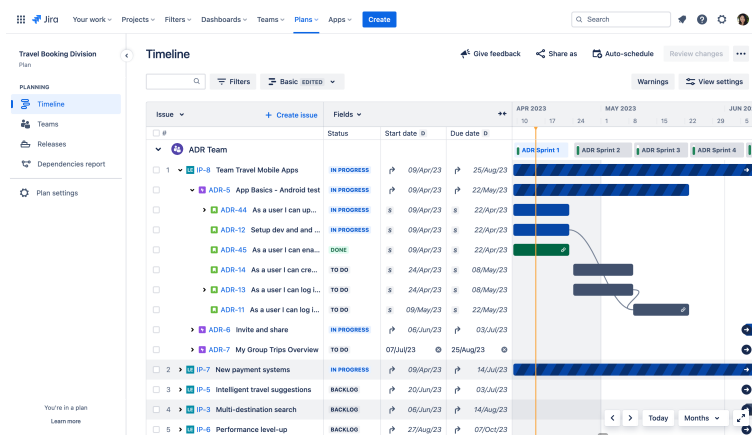



Figura 1.2: Esempio di gestione delle attività con *Jira*

Fonte: atlassian.com

Bitbucket

Un'altro software nella suite di *Atlassian* è *Bitbucket*, che viene utilizzato per il controllo di versione e per la gestione del codice sorgente. Con questo software è possibile creare *repository*, questo spazio permette ai *team* di collaborare in un modo efficiente, avendo anche la disponibilità di utilizzare funzionalità come le *pull request* .

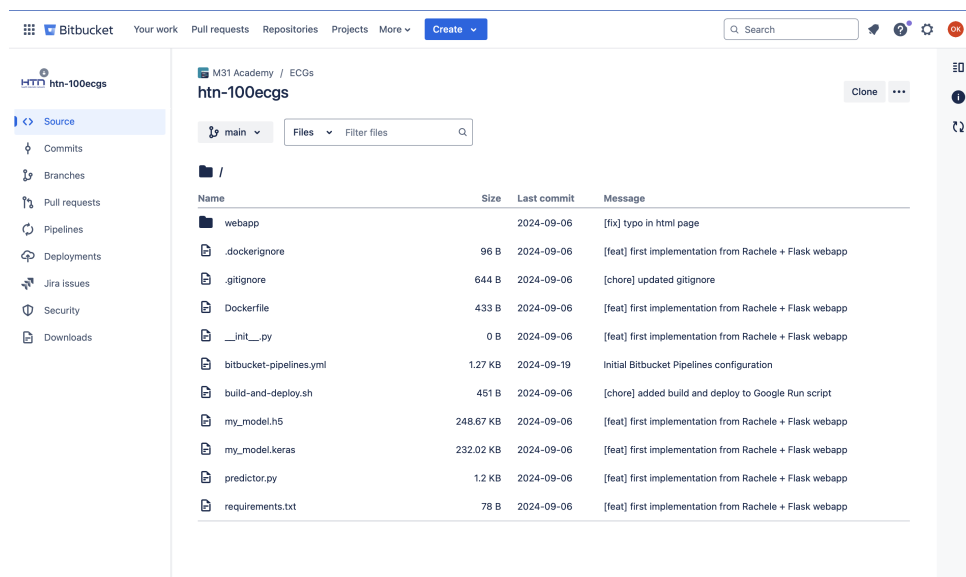


Figura 1.3: Screenshot di una *repository* in *Bitbucket*

Confluence

Confluence è un'altro software sviluppato da *Atlassian* e viene utilizzato per la gestione della documentazione. Facendo parte della *suite* di *Atlassian* è ben integrato con le tecnologie precedentemente espone. All'interno di questo spazio vengono create documentazioni di prodotti e vari documenti contenenti risorse utili anche internamente.

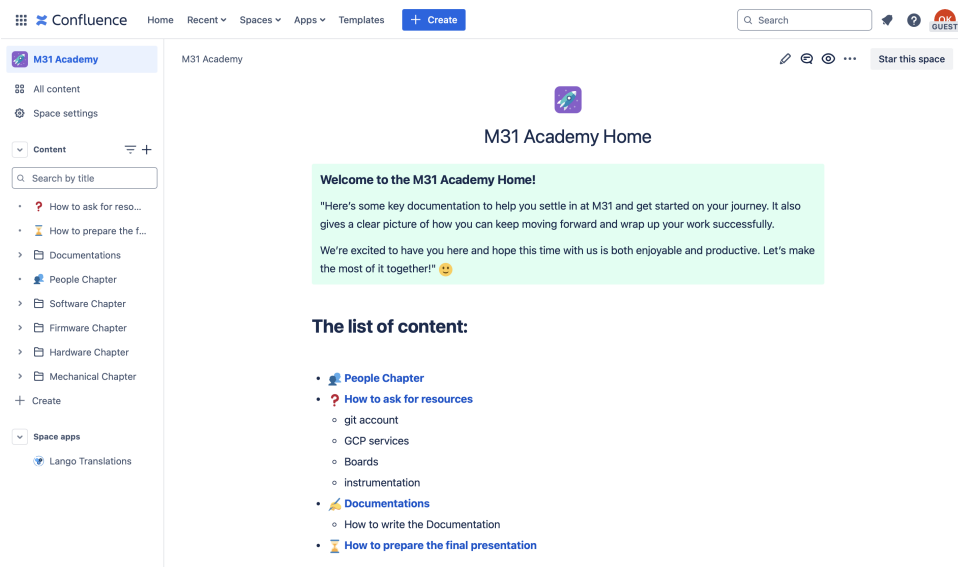


Figura 1.4: Pagina principale del *Confluence* di *M31 Academy*

Microsoft Teams

L'applicativo che viene utilizzato per comunicare è *Microsoft Teams*. Questo software sviluppato da *Microsoft* permette di comunicare, con singoli individui o con gruppi di persone, sia tramite messaggi testuali che tramite videochiamate. Può essere usato anche per trasferimenti di piccoli file, ed è la prima opzione per comunicare con i membri dei team che lavorano da remoto.

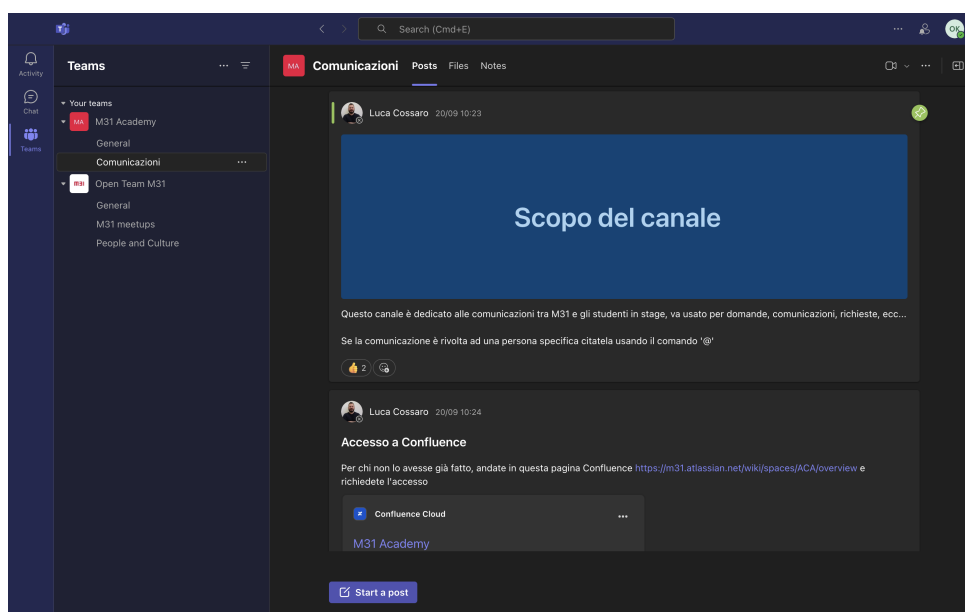


Figura 1.5: *Screenshot* di un canale di comunicazione di *M31 Academy*

1.6 Propensione all'innovazione

M31 si interessa parecchio all'innovazione e questo lo si può vedere dalla filosofia con cui intraprende nuovi progetti di lavoro, come già menzionato nella sezione introduttiva [1.1](#), i progetti che svolge sono tutti con lo scopo di portare progresso nei settori in cui si specializza. Inoltre, durante lo svolgimento del mio tirocinio, ho potuto assistere ad un *meeting* plenario che viene svolto periodicamente, nel quale ho potuto vedere in quali settori l'azienda è intenzionata ad esplorare in futuro, sia dal lato della dirigenza che dal lato dei dipendenti.

Capitolo 2

Stage proposto

2.1 Gestione aziendale degli *stage*

M31 mostra molto interesse verso la creazione di rapporti con le nuove generazioni, questo avviene in diversi modi: tramite la *M31 Academy* e tramite la partecipazione ai progetti di “Ingegneria del *Software*” della laurea di informatica, come azienda proponente.

M31 Academy è una sorta di settore aziendale dedicato alla realizzazione di progetti con studenti e studentesse universitarie o neo-laureati e neo-laureate. Ora come ora, l'azienda sta svolgendo progetti solo con studenti e studentesse universitarie, che prendono il ruolo di stagisti e stagiste, perciò passo a descrivere meglio come essi vengono gestiti.

L'influsso di stagisti e stagiste è principalmente dovuto alla partecipazione dell'azienda allo STAGE-IT, un evento promosso da Confindustria Veneto Est e l'Università di Padova, dove studenti e studentesse vengono introdotti a tre progetti che possono svolgere in azienda. M31 però offre una lista più numerosa di progetti, che vengono esposti quando gli studenti e studentesse contattano direttamente l'azienda. Questi progetti, essendo numerosi, possono essere di vario tipo. Alcuni pongono gli stagisti e stagiste in un progetto su cui l'azienda sta già lavorando. Mentre altri, come quello che ho scelto io, servono all'azienda per esplorare nuove tecnologie o per prepararsi per un progetto aziendale che non ha ancora avuto modo di iniziare.

2.2 Descrizione progetto

Il progetto di *stage* consiste nella realizzazione di un'applicazione per l'analisi di dati tramite tecniche di *machine learning_G* e di *deep learning_G*; in particolare, per l'analisi di segnali elettrocardiografici, tramite elettrocardiogrammi.

Questo progetto di *stage* serve all'azienda per prepararsi ad un progetto che si aspettano di affrontare in futuro, collaborando insieme ad un'azienda con cui sono attualmente in comunicazione.

Questo progetto vede come *focus* principale la gestione di grandi quantità di dati, sotto forma di migliaia di *file* contenenti segnali elettrocardiografici e altre informazioni sui pazienti. Questi dati, facenti parte di un *dataset_G*, vengono utilizzati all'interno di un programma per addestrare *reti neurali artificiali_G*, ovvero di modelli che cercano di replicare il funzionamento del cervello umano. Questi dati non possono subire modifiche inaspettate quando vengono inseriti nel processo di addestramento. Inoltre, questi dati devono essere gestiti in modo efficiente, in modo da non caricare tutto all'interno della *RAM_G*, come viene generalmente fatto nel caso di *dataset* di piccole dimensioni, così da non subire limitazioni di *hardware*.

In secondo luogo, il progetto vede anche l'effettivo addestramento e valutazione di vari tipi di reti neurali artificiali, sia con lo scopo di testare, che di esplorare quali sono le tecniche migliori per l'analisi di elettrocardiogrammi.

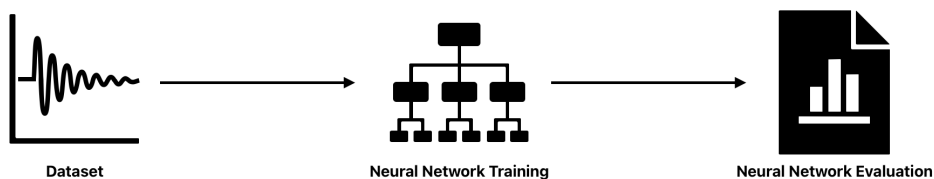


Figura 2.1: Rappresentazione dei punti principali del progetto

2.3 Obiettivi

Il progetto di *stage* era definito da alcuni obiettivi richiesti dall'azienda, ognuno di essi era contrassegnato con una certa importanza; questa può rientrare in una dei tre seguenti tipi:

- **Obbligatorio:** indica un obiettivo primario richiesto dal committente, il suo completamento non è negoziabile
- **Desiderabile:** indica un obiettivo non vincolante o strettamente necessario, ma dal riconoscibile valore aggiunto
- **Facoltativo:** indica un obiettivo facoltativo, che rappresenta valore aggiunto e non strettamente competitivo

Nella tabella 2.1, definita successivamente, vengono esposti i vari obiettivi aziendali che sono stati imposti per il completamento del progetto:

Obiettivo	Importanza
Studio delle tecnologie e del contesto specifico	Obbligatorio
Creazione di un dataset appropriato e di tecniche per la sua gestione	Obbligatorio
Implementazione in <i>Python</i> degli algoritmi basati su tecniche di <i>deep learning</i>	Obbligatorio
Valutazione dell'esecuzione degli algoritmi	Obbligatorio
Realizzazione di <i>unit test</i>	Desiderabile
Idee o suggerimenti su come migliorare in futuro le prestazioni dell'applicativo	Facoltativo

Tabella 2.1: Lista dei vari obiettivi aziendali.

2.4 Vincoli

2.4.1 Vincoli tecnologici

Alcune delle tecnologie mi sono state imposte per la realizzazione di questo progetto. Queste tecnologie sono relative all'addestramento dei modelli di intelligenza artificiale, nella quale Keras era la tecnologia che dovevo usare. Utilizzando questa tecnologia, però, si è inoltre obbligati a fare uso di *Python* e, per aumentare la semplicità di installazione dei pacchetti di questo linguaggio di programmazione, mi è stato imposto anche l'uso di *Conda*, un sistema per la gestione di pacchetti e *environment*_G, che ha permesso l'installazione di ciò che era necessario senza troppi intoppi.

2.4.2 Vincoli temporali

Per la sua natura, il tirocinio curricolare ha un limite di tempo: ovvero deve essere dalle 300 alle 320 ore di lavoro, e non è possibile sforare 40 ore settimanalmente. Per lavorare al progetto, ho distribuito lo *stage* in giornate lavorative da 8 ore ciascuno, per un totale di all'incirca 38 giorni.

2.4.3 Vincoli organizzativi

Per il corretto svolgimento del tirocinio curricolare, ho preso, insieme a alle figure che mi supervisionavano, alcuni accorgimenti.

In primis, periodicamente vengono svolti degli incontri di *sprint review* con il mio *tutor* aziendale, nei quali vengono esposte le attività svolte da me durante l'ultimo periodo, in modo da far sì che l'avanzamento del progetto si allineasse con gli obiettivi imposti in modo corretto.

In secundis, ogni cinque giorni lavorativi dovevo mandare una comunicazione al mio relatore di tesi, indicando quali attività erano previste da svolgere durante la settimana lavorativa passata, e come esse sono state svolte rispetto alle aspettative. Questo per monitorare il corretto proseguimento dello *stage*.

2.5 Pianificazione delle attività

Come già menzionato nella sezione 2.4.2, il progetto di *stage* è stato suddiviso in diverse giornate lavorative. Le giornate sono suddivise in otto settimane, ciascuna delle quali prevede delle attività da svolgere, decise all'inizio dello svolgimento del tirocinio, basandosi sul piano di lavoro redatto prima dell'inizio dello stesso. Di seguito vengono esposte le varie attività per settimana:

- **Prima settimana:**
 - Studio del problema specifico dell'analisi di elettrocardiogrammi
 - Definizione delle *user story*
- **Seconda settimana:**
 - Formazione e ricerca di tecniche di *deep learning* specifiche per il problema di analisi di serie temporali
 - Studio del *dataset* pubblico PTB-XL
- **Terza settimana:**
 - Ricerca e analisi delle tecniche di *deep learning* specifiche per il problema affrontato
 - Analisi e confronto del *dataset* pubblico con le caratteristiche del *dataset* privato
 - Realizzazione di un primo prototipo dell'applicazione di addestramento
- **Quarta settimana:**
 - Modifica e riduzione del *dataset* rispettando i requisiti imposti
 - Implementazione della *Sequential API*_G di Keras
 - Prima implementazione del caricamento ottimizzato del *dataset*

- **Quinta settimana:**
 - Implementazione della *Functional* API di Keras
 - Completamento dell'implementazione del caricamento ottimizzato del *dataset*
 - *Testing* dell'implementazione della *Sequential* API
- **Sesta settimana:**
 - Implementazioni di tecniche di *downsampling_G* e di *preprocessing_G*
 - *Testing* dell'implementazione della *Functional* API
- **Settima settimana:**
 - *Testing* dell'intero applicativo
 - Svolgimento di vari addestramenti di reti neurali artificiali
- **Ottava settimana:**
 - Valutazione dell'applicazione creata
 - Realizzazione della presentazione finale in azienda

2.6 Motivazione della scelta

Sono venuto a conoscenza di M31 attraverso l'evento STAGE-IT, un'iniziativa che mi ha dato la possibilità di incontrare, in un singolo posto, diverse aziende e le loro proposte di *stage*. Tra le aziende presenti, M31 era una di quelle che mi ha interessato di più, questo perchè avevo un maggiore interesse a svolgere un progetto che utilizzava, in qualche maniera, l'intelligenza artificiale.

M31 durante l'evento STAGE-IT aveva proposto un tema di *machine learning* in ambito di *computer vision_G*, che ero intenzionato nel svolgere quando stavo contattando l'azienda, ma vedendo la lista aggiornata dei progetti di *stage* offerti, ho deciso di scegliere un secondo progetto, sempre di intelligenza artificiale, ma in ambito biomedico, questo perchè le tecniche utilizzate per questo specifico ambito mi sarebbero state nuove.

Gli obiettivi personali che mi sono posto prima di svolgere questo tirocinio sono i seguenti:

- **Approfondimento tecnico:** avendo parecchio interesse verso l'intelligenza artificiale, con questo *stage*, desideravo approfondire questo argomento svolgendo un progetto con un'applicazione in questo ambito, per me nuova. Questo obiettivo riflette anche la mia intenzione di proseguire gli studi con la laurea magistrale con indirizzamento all'intelligenza artificiale.
- **Capire l'ambiente lavorativo:** essendo la mia prima esperienza lavorativa in questo settore, ero incuriosito da come, effettivamente, le aziende operano internamente. Non solo apprendere le metodologie aziendali, ma anche la cultura lavorativa.
- ***Problem solving*:** migliorare nell'abilità di risolvere problemi, in modo da trovare soluzioni efficienti a vari problemi a diverse sfide.

Capitolo 3

Svolgimento dello *stage*

In questo capitolo descriverò in dettaglio lo svolgimento dello *stage*, suddiviso nelle varie fasi che hanno caratterizzato il progetto.

Inizierò con la descrizione dell'analisi dei requisiti, dove verranno illustrate le esigenze e le aspettative degli *stakeholder* e come queste sono state raccolte e documentate tramite le *user story*.

Successivamente tratterò delle fasi di progettazione e codifica, spiegando le scelte architetture e le tecnologie utilizzate per sviluppare il progetto.

Concluderò con la verifica, validazione e illustrando i risultati che ho ottenuto, sia da un punto di vista qualitativo che quantitativo.

3.1 Analisi dei requisiti

Per garantire che il prodotto finale soddisfi le aspettative e gli obiettivi, è necessario analizzare a fondo le varie esigenze e aspettative degli *stakeholder* che esso dovrà soddisfare. Per fare ciò, ho utilizzato l'approccio delle *user story*, ovvero varie descrizioni semplici delle funzionalità dal punto di vista dell'utente.

Le *user story* vengono raccolte e identificate in una lista tramite un codice, esso è strutturato nel seguente modo:

US-[Numero incrementale]

Queste *user story* sono state approvate con il *tutor* e ognuna di essa è mandatoria.

Identificatore	<i>User story</i>
US-1	Come utente, voglio che il <i>dataset</i> pubblico PTB-XL venga usato per svolgere addestramenti di reti neurali artificiali, così che io possa provare diverse strutture nell'ambito dell'analisi di segnali elettrocardiografici.
US-2	Come utente, voglio che il <i>dataset</i> venga caricato nella fase di <i>training</i> senza sforare i limiti della RAM, in modo da permettere addestramenti di reti neurali artificiali con <i>dataset</i> di qualsiasi dimensioni
US-3	Come utente, voglio che i parametri per l'addestramento di reti siano in un singolo posto, in modo da facilitare il processo di addestramento di nuovi modelli basati su nuove strutture
US-4	Come utente, voglio che la rete neurale artificiale abbia come input aggiuntivo, oltre al segnale elettrocardiografico, anche altre informazioni come età e peso del paziente, in modo da aumentare i risultati dei modelli creati con i vari addestramenti
US-5	Come utente, voglio che ci sia la possibilità di fare <i>preprocessing</i> , in modo da applicare funzioni per estrarre <i>features</i>
...	...
US-19	Come utente, voglio che il programma crei un grafico di <i>confidenza_G</i> usando i dati del <i>dataset</i> privato, in modo da valutare oggettivamente la prestazione del modello addestrato
US-20	Come utente, voglio che l'applicazione salvi ogni metrica e modelli creati in una <i>directory</i> dedicata, in modo da poter salvare ogni addestramento svolto e dare la possibilità di confrontare i risultati se ce ne fosse bisogno

Tabella 3.1: Tabella contenente le varie *user story*.

3.2 Progettazione

3.2.1 Visione d'insieme

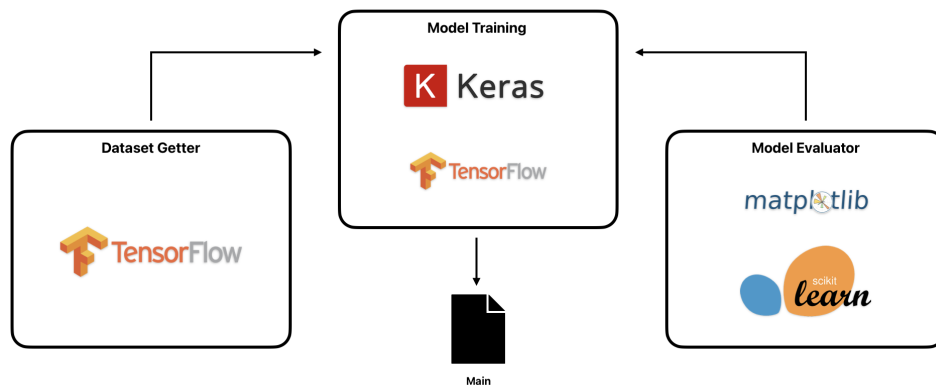


Figura 3.1: Architettura generale con le principali tecnologie utilizzate

L'applicazione progettata la si può vedere in tre macro sezioni, dove la principale di esse è quella che si occupa di effettuare l'addestramento delle reti neurali artificiali.

Per soddisfare i requisiti ho deciso di realizzare un semplice programma in Python utilizzabile dal terminale, quindi senza nessuna interfaccia grafica. Questa scelta è dovuta sia dal fatto che non era richiesto per i requisiti, che dal fatto che non è necessario semplificare troppo l'utilizzo del programma, siccome ci si aspetta che l'utente finale sia già esperto e che voglia una certa libertà nella modifica dei parametri che vuole cambiare.

Ho deciso di progettare il programma incentrando tutto sul componente dedicato a svolgere gli addestramenti. Questo componente possiede due dipendenze, che sono: il componente dedicato all'ottenimento dei dati e il componente che si occupa di valutare il risultato.

3.2.2 Addestramento di modelli

Il *focus* principale del programma è di creare modelli tramite l'addestramento di reti neurali artificiali. Per fare ciò ho utilizzato, come tecnologia principale, Keras.

Attraverso questa tecnologia è possibile creare modelli attraverso due API che sono disponibili all'utilizzo. Il loro funzionamento non è dissimile, ma possiedono delle caratteristiche fondamentali da capire:

- ***Sequential API***: attraverso questa API la definizione dei modelli è veramente molto semplice, e risulta molto semplice anche il loro utilizzo. Però questa semplicità avviene ad un costo: ovvero che non è possibile inserire più *input* di diverso tipo in contemporanea. Questo perchè i modelli che si possono definire, sono sequenziali e perciò sono una singola catena di processamento.
- ***Functional API***: attraverso questa API si possono definire modelli più complessi, ma anche l'utilizzo e definizione diventano relativamente più complessi. Questa intricatezza permette di creare modelli che supportano *multi-input*, *multi-output* oppure entrambi.

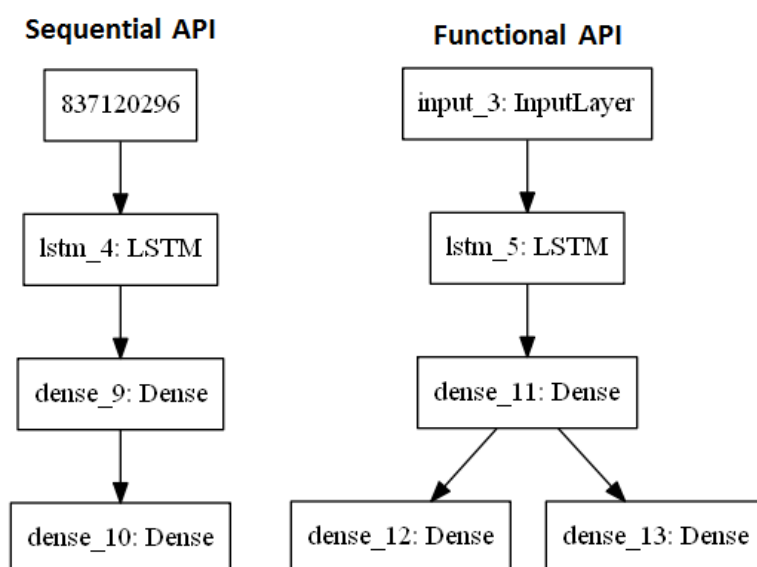


Figura 3.2: Confronto tra *Sequential API* e *Functional API*

Fonte: stackoverflow.org

Il programma da me progettato, include entrambe le API. L'utilizzo di una di esse viene deciso dall'utente nel *file* principale di esecuzione del programma.

La scelta di implementare entrambe deriva dal fatto che l'utente finale necessita di confrontare l'influenza delle informazioni aggiuntive che vengono inserite nel modello sulle prestazioni finali ottenute. In questo specifico ambito, non è detto che informazioni aggiuntive come l'età e il peso del paziente influiscano notevolmente sulle prestazioni del modello creato.

Da un punto di vista teorico, le informazioni aggiuntive influiscono in modo positivo sulle prestazioni, però questa aggiunta espande il modello, e facendo ciò, vengono incrementati anche i requisiti prestazionali della macchina che deve eseguirlo.

Perciò ho deciso di implementare le due API sotto forma di due classi che implementano la stessa classe astratta, in modo da avere in comune più codice possibile.

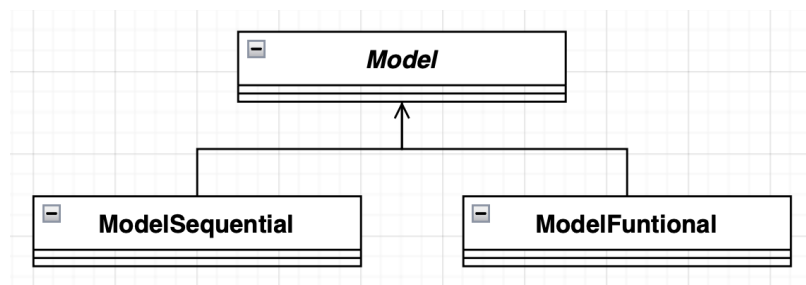


Figura 3.3: Diagramma delle classi di addestramento di reti neurali artificiali

3.2.3 Gestione del *dataset*

Una delle dipendenze della parte di addestramento dei modelli è l'ottenimento dei dati su cui eseguire l'addestramento. Questo è un'importante passo negli addestramenti di reti neurali artificiali, senza di esso non sarebbe possibile neanche realizzare un singolo modello.

Per far sì che un addestramento avvenga, è necessario che i dati richiesti dall'addestramento stesso, siano disponibili all'interno della RAM, ma per fare ciò, ci sono due metodi:

- **Caricare l'intero *dataset*:** questa è l'opzione più semplice da realizzare. Ha il vantaggio di predisporre tutti i dati necessari in una memoria veloce come la RAM, permettendo tempi di risposta tra le varie *epoche* veramente bassi. Tuttavia, questo metodo viene utilizzato solo quando il *dataset* utilizzato non è tanto grande, siccome più grande un *dataset* è, più RAM è richiesta.
- **Caricare in modo dinamico:** questa opzione risolve il problema del limite della RAM. I file richiesti durante l'addestramento vengono letti dalla memoria per l'archiviazione a lungo termine quando sono necessari, o se si utilizza il *prefetch*, momenti prima. Perciò i dati permangono per poco tempo all'interno della RAM prima di essere rimpiazzati dai dati successivi. Questo permette di utilizzare *dataset* di qualsiasi dimensione per addestrare reti neurali artificiali in qualsiasi macchina si voglia. Lo svantaggio di questa tecnica è che subisce un *bottleneck* nel caso si stia addestrando un modello computazionalmente semplice. Questo per via del limite fisico dovuto alla lettura dei dati dalla memoria di archiviazione a lungo termine.

Questo progetto, come spiegato precedentemente, tratta di *dataset* di grandi dimensioni, perciò posso considerare di sviluppare solo la seconda delle due opzioni qui sopra presentate.

Per implementare ciò, ho scelto di utilizzare i *generator* disponibili utilizzando TensorFlow, essendo completamente compatibile con l'addestramento effettuato con Keras. Ero però dubbioso in che formato era meglio mantenere il *dataset* per ottenere i migliori risultati di velocità nel caricamento dei dati.

Per capire ciò, ho sperimentato utilizzando diversi formati di file per cercare di ottenere il miglior compromesso tra dinamicità e velocità.

Prima di esporre quale soluzione ho deciso di utilizzare alla fine, vado ad esporre una delle soluzioni che ho sperimentato. Essa era stata ideata basandosi sulla riduzione dei tempi di caricamento dei vari file, riducendo il numero di file da caricare. Per fare ciò ho compresso i *file* originali del *dataset*, in vari file HDF5, un formato di *file* progettato per la memorizzazione di grandi quantità di dati,

in modo da avere un numero minore di *file* con una dimensione leggermente più grande. Questo ha permesso di avere tempi di risposta minori tra una epoca e l'altra nella fase di addestramento. Ma è anche vero che questa soluzione possedeva due problemi:

- **Gestione delle *label_G*:** questa soluzione complicava la gestione delle etichette associate ai dati, che erano ora in gruppi di *file*. Una possibile soluzione avrebbe portato a progettare un sistema più complicato del necessario, e avrebbe portato via molto tempo.
- **Troppa specificità:** decidere quanti dati inserire in un HDF5 *file* è di per sè un dilemma. La dimensione di certi *file* potrebbe andare bene per uno scenario d'installazione e d'uso, ma non è detto che sia la scelta più ottimale per altri scenari. Inoltre il vantaggio viene perso se si svolge addestramenti di modelli computazionalmente demandanti.

Perciò, considerando tutte le opzioni, ho deciso che era meglio mantenere i vari dati del *dataset* nel loro formato originale. Per il semplice fatto che questa soluzione avrebbe portato alla minore produzione di codice, che è importante da considerare, siccome richiede meno modifiche in futuro, nel caso ce ne fosse bisogno.

Ho progettato questo componente similmente al componente precedentemente esposto: ovvero tramite una classe astratta da dove vengono implementate diverse classi concrete. In questo caso, avere la classe astratta è di maggiore importanza. Questo perchè oltre a creare le due classi per dare l'*input* alle due API, che ne richiedono due diversi, dovevo implementare anche un generatore per i *file* *DICOM_G* che facevano parte del piccolo *dataset* privato. Questi *file* sono una piccolissima frazione del *dataset* con cui l'azienda si troverà a lavorare in futuro, perciò come da requisiti, dovevo implementarli per validare la possibilità di utilizzarli nel codice da me prodotto.

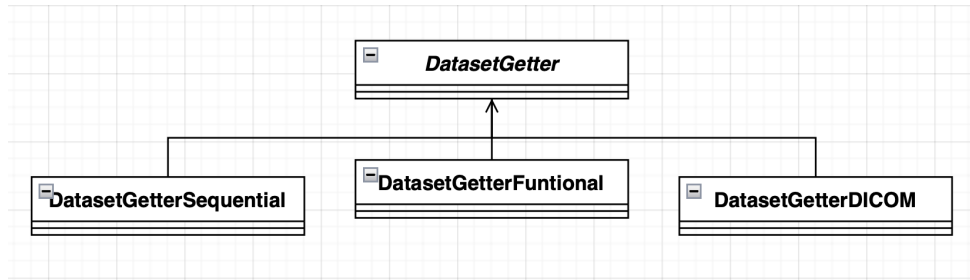


Figura 3.4: Diagramma delle classi di gestione del *dataset*

3.2.4 Valutazione dei modelli

L'ultima dipendenza per l'addestramento delle reti neurali artificiali è la realizzazione di vari *output* che danno la possibilità di valutare oggettivamente le prestazioni del modello creato.

A fine dell'addestramento, il componente a esso dedicato si occupa di chiamare questo componente, che ricevendo determinate informazioni dell'addestramento, si occupa di generare le varie metriche richieste dai requisiti. Di seguito vado ad esporre le metriche implementate:

- **Grafici di andamento del *training*:** durante l'addestramento, Keras, si occupa di valutare l'andamento. Prendendo le informazioni collezionate da Keras, è possibile creare dei grafici, utili per valutare varie cose.
- ***Matrice di confusione_G*:** "strumento" utilizzato per comparare le predizioni di *test* che un modello svolge, con gli effettivi risultati.
- ***Confidenza_G*:** generalmente un'istogramma dove viene visualizzata l'incertezza del modello nelle predizioni che svolge durante la fase di *test*.
- **Varie statistiche:** si tratta di una lista di metriche in forma testuale, che valuta numericamente le prestazioni del modello nel predire. Una delle metriche più notevoli è quella dell'accuratezza, che indica la proporzione di predizioni corrette rispetto al numero totale di predizioni effettuate.

Per la realizzazione dei grafici ho deciso di utilizzare la libreria Matplotlib, mentre per la matrice di confusione e le varie statistiche, ho scelto di usare la libreria: Scikit-learn.

Inoltre, questo componente si occupa di salvare i modelli che sono stati creati durante l'apprendimento in una specifica *directory*, insieme a tutte le metriche qui sopra esposte.

3.3 Codifica

In questa sezione descrivo come ho svolto la fase di codifica del progetto, mi soffermerò a descrivere alcune parti di codice che ritengo notevoli di attenzione.

3.3.1 Prototipo

Prima ancora di iniziare a realizzare quello che verrà considerato il prodotto finale, ho creato un prototipo che mi ha aiutato nella fase di progettazione.

Questo prototipo ha racchiuso, semplificando, i vari punti del progetto in un singolo *file* eseguibile, che quindi possedeva tutta la *pipeline* di: gestione del *dataset*, addestramento delle reti neurali artificiali e la valutazione del modello creato.

Grazie ad esso ho potuto sperimentare vari modi per caricare il *dataset* in RAM per effettuare l'addestramento in modo efficiente.

Di seguito espongo la versione finale che ho ottenuto dopo la fase di sperimentazione:

```
training_dataset = create_data_generator(sampling_rate=samp_rate)
training_dataset = training_dataset.batch(batch_size)
training_dataset = training_dataset.prefetch(buffer_size=2)
```

Codice 3.1: Codice della variabile del generatore

Il *dataset* viene visto all'interno del programma come una variabile. Essa è un `tf.data.Dataset` della libreria TensorFlow, che è un *input* supportato da Keras per l'addestramento di una rete. Da notare nel codice 3.1 è che in ultima riga viene applicato il *prefetch* (evidenziato in blu), in verde che in questo caso si occupa di ottenere, oltre al dato necessario al momento, anche i prossimi due dati richiesti. Questi dati sono stati raggruppati in *batch_G* (evidenziato in verde),

perciò non viene preso un singolo dato per processare, ma un piccolo gruppo. La definizione del generatore avviene tramite la funzione visibile nel codice 3.2, definito qui sotto. Essa si occupa di prelevare le informazioni che sono contenute in uno specifico file, che contiene i percorsi *file* dei vari dati, oltre che a informazioni aggiuntive associate a quest'ultimi, come le *label*. Successivamente si occupa di dichiarare il generatore dei dati, applicando una determinata struttura, e ritorna la variabile.

```
def create_data_generators(sampling_rate):
    y = pd.read_csv(path_to_dataset + 'new_ptb_xl.csv')

    train_df = y[(y.strat_fold == 'train')].to_numpy()

    np.random.shuffle(train_df)

    output_types = (tf.float32, tf.float32)
    output_shapes = (tf.TensorShape([sampling_rate * 10, 12]),
        ↪ tf.TensorShape([]))

    train_generator = tf.data.Dataset.from_generator(
        lambda: ecg_generator(train_df),
        output_types=output_types,
        output_shapes=output_shapes
    )

    return train_generator
```

Codice 3.2: Codice della funzione che definisce il generatore

La funzione che effettivamente “genera” i dati, raffigurata nel codice 3.3 definito di seguito.

```
def ecg_generator(elements):
    for element in elements:
        x, y = load_ecg_file(element)
        yield x, y
```

Codice 3.3: Parte di codice che sfrutta il comando `yield`

Questa funzione prende ogni singolo elemento della lista e ritorna una sorta di iteratore tramite il comando `yield` di Python. Gli elementi che vengono restituiti sono il segnale elettrocardiografico e la *label* ad esso associato, che a loro volta sono stati ricavati con le seguenti funzioni:

```
def binarize_label(label):
    if 'NORM' in label:
        return 0
    else:
        return 1

def load_ecg_file(element):
    pre_data = [wfdb.rdsamp('dataset/ptb-xl/' + str(element[4]))]
    return np.array([signal for signal, meta in pre_data])[0],
    ↪ binarize_label(element[2])
```

Codice 3.4: Codice delle funzioni che ottengono gli effettivi dati

Nel codice 3.4, possiamo vedere come vengono gestiti gli elettrocardiogrammi e le *label*.

Nel caso degli elettrocardiogrammi, in questo esempio stavo utilizzando il *dataset* pubblico PTB-XL, che salva ogni singolo segnale all’interno di un WFDB (*WaveForm DataBase*), ovvero una struttura specifica per memorizzare segnali di dati fisiologici. Essi vengono letti attraverso la funzione: `wfdb.rdsamp()`.

Per quanto riguarda le *label*, esse originano dalla lista che viene letta all’interno di 3.2. Nella lista sono segnate in varie superclassi, ma in questo esempio vengono trasformate in valore binario: ‘0’ nel caso di pazienti sani, ‘1’ nel caso di pazienti con patologie.

3.3.2 Gestione del *dataset*

Ho realizzato questo componente utilizzando del codice che non differenzia molto da quello che ho appena illustrato nella sotto-sezione precedente. Tuttavia, da requisiti, dovevo supportare altre “versioni” del *dataset*. Di seguito illustro due di queste versioni.

La prima versione che voglio esporre è quella che oltre al segnale elettrocardiografico, include anche altre informazioni del paziente:

```
def _load_single_ecg(self, element):
    pre_data = [wfdb.rdsamp(ConfigGetter().get_dataset_path() +
        ↪ str(element[4]))]
    data = np.array([signal for signal, meta in pre_data])[0]

    data = self._downsample_ecg(data)

    if self._preprocessing:
        data = self._preprocess_ecg(data)

    # ...

    # Adds the Age and the Sexuality
    additional_information = [element[0], element[1]]

    return data, additional_information, label

def _ecg_generator(self, elements):
    for element in elements:
        x, x_add, y = self._load_single_ecg(element)
        yield (x, x_add), y
```

Codice 3.5: Codice del generatore di dati che include informazioni aggiuntive

Con il codice 3.5 voglio notare che la maggior parte delle modifiche che vengono svolte sono nelle funzioni che utilizzano il `yield` di Python e la funzione che si occupa di ricavare gli effettivi dati.

La versione di codice che ho incluso qua fa parte della versione finale (in parte tagliata per questioni di spazio), ed è per questo che nella funzione `_load_single_ecg()` sono presenti anche i richiami per le funzioni di *downsampling* e di *preprocessing*, due procedimenti che erano richiesti dai requisiti.

La seconda versione che volevo mostrare era l'implementazione del supporto per i *file* DICOM:

```
def _load_single_ecg(self, element):
    # Loads from a DICOM file the ECG signals
    pre_data = dcmread(ConfigGetter().get_dicom_dataset_path() +
        ↪ str(element[4]))
    data = pre_data.waveform_array(0) / 1000

    data = self._downsample_ecg(data)

    if self._preprocessing:
        data = self._preprocess_ecg(data)

    if self._binary_labels:
        return data, binarize_label(element[2])
    else:
        return data, encode_label_for_multilabel(element[2])

def _ecg_generator(self, elements):
    for element in elements:
        x, y = self._load_single_ecg(element)
        yield x, y
```

Codice 3.6: Codice del generatore di dati attraverso *file* DICOM

Anche nel codice 3.6 è possibile notare che la maggior parte del codice rimane la stessa, con differenza di come viene letto il *file*. Essendo un formato diverso, il *file* DICOM non viene più letto attraverso `wfdb.rdsamp()`, ma attraverso `dcmread()`.

3.3.3 Addestramento di reti neurali artificiali

La componente per l'addestramento di reti neurali artificiali sarà la parte unisce le altre componenti per realizzare gli obiettivi del progetto, però è anche quella che è possiede meno codice rispetto alle altre componenti. Questo è dovuto principalmente dovuto alla semplicità di utilizzo delle API di Keras. A riguardo di questo, voglio mostrare la differenza delle due API utilizzabili, che ho menzionato nella sezione di progettazione.

Inizio illustrando con un'esempio di modello che utilizza la *Sequential* API:

```
self._model = keras.Sequential([
    layers.Input(shape=(self._sampling_rate * 10, 12)),
    layers.Bidirectional(layers.LSTM(32, return_sequences=True)),
    layers.AveragePooling1D(),
    layers.Dropout(rate=0.5),
    layers.BatchNormalization(),
    layers.Bidirectional(layers.LSTM(24, return_sequences=True)),
    layers.Dropout(rate=0.5),
    layers.Flatten(),
    layers.Dense(176, kernel_regularizer=regularizers.l2(0.02)),
    layers.Dense(44, kernel_regularizer=regularizers.l1(0.02)),
    layers.Dense(ConfigGetter().get_number_of_res_classes(),
        ↪ activation='sigmoid')
])
```

Codice 3.7: Esempio di definizione di un modello con la *Sequential* API

Nel codice 3.7 definito qui sopra, possiamo notare che la definizione è racchiusa in una singola variabile. I modelli sono realizzati a strati, dove ogni strato realizza ognuno una specifica operazione. L'*input* viene trattato uno strato alla volta, dall'alto verso il basso, dove viene restituito l'*output*.

```
input_layer = keras.Input(shape=(self._sampling_rate * 10, 12))

x1 = keras.layers.Conv1D(filters=24, kernel_size=40, strides=1,
    ↪ activation='relu')(input_layer)
x1 = keras.layers.AveragePooling1D()(x1)
x1 = keras.layers.Dropout(rate=0.4)(x1)
x1 = keras.layers.BatchNormalization()(x1)
x1 = keras.layers.Conv1D(filters=48, kernel_size=10, strides=1,
    ↪ activation='relu')(x1)
x1 = keras.layers.Dropout(rate=0.3)(x1)
x1 = keras.layers.Flatten()(x1)

additional_input = keras.Input(shape=(2,))
x2 = keras.layers.Dense(16)(additional_input)
x2 = keras.layers.Dense(16)(x2)
x2 = keras.layers.Dense(16)(x2)

x = concatenate([x1, x2])
x = keras.layers.Dense(125)(x)
x = keras.layers.Dense(25, kernel_regularizer=regularizers.l2(0.01))(x)

output_layer =
    ↪ tf.keras.layers.Dense(ConfigGetter().get_number_of_res_classes(),
    ↪ activation=activation)(x)

self._model = keras.Model(inputs=[input_layer, additional_input],
    ↪ outputs=output_layer)
```

Codice 3.8: Esempio di definizione di un modello con la *Functional* API

Mentre osservando l'esempio di *Functional API*, rappresentato nel codice 3.8, possiamo vedere che vengono dichiarate diverse variabili prima di dichiarare l'effettiva struttura finale del modello. Questo perchè vengono creati diverse strutture sequenziali che vengono unite alla fine per creare un singolo modello. Nell'ultimo esempio mostrato, ho creato due “*pipeline*” per processare i segnali elettrocardiografici e le informazioni aggiuntive dei pazienti, rispettivamente tramite `x1` e `x2`. Questi due elementi sequenziali uniscono il loro ultimo strato in un nuovo elemento sequenziale, `x`, che si occuperà di restituire l'*output* finale del modello.

3.3.4 Menzioni onorevoli

Per finire, desidero discutere di alcune altre parti di codice, non troppo rilevanti ma che meritano almeno una piccola menzione.

Una di esse è la componente che viene utilizzata quando il l'addestramento viene terminato, ovvero la componente di valutazione del modello creato. Ho codificato una semplice classe che raccoglie tutte le funzioni che generano le metriche necessarie per la valutazione, come ad esempio 3.9.

```
def generate_binary_histogram(model_id, acc_prevision,
    ↪ loss_prevision):
    fig2, histogram = plt.subplots()
    histogram.set_ylabel('Number of predictions')
    histogram.set_xlabel('prediction')
    histogram.hist([[x1[0] for x1 in acc_prevision.tolist()], [x2[0]
    ↪ for x2 in loss_prevision.tolist()]], edgecolor='#111',
    ↪ linewidth=0.5, label=['Accuracy_Model', 'Loss_Model'])
    histogram.legend()
    plt.savefig(ConfigGetter().get_models_path() + 'model_' + model_id
    ↪ + '/histogram_model_' + model_id + '.png')
```

Codice 3.9: Esempio di funzione per la valutazione dei modelli

Inoltre, questa componente utilizza varie funzioni che ho creato per il salvataggio delle metriche e modelli create durante esecuzione del programma.

Infine, per facilitare la modifica di variabili che si trovano in diverse parti del codice, ho creato una classe che si occupa di restituire i valori configurati in `config.json` dove è necessario, semplificando l'utilizzo del programma all'utente.

3.4 Verifica e validazione

Per questo processo di sviluppo del software ho svolto principalmente due cose. La prima è stata di realizzare alcuni *unit test*, soprattutto per controllare il corretto funzionamento delle funzioni che si occupano di gestire la trasformazione delle *label*. Fare ciò è abbastanza importante, questo perchè se le *label* vengono trasformate in un modo da non essere più veritiere quando vengono associate al proprio dato, allora quando esse vengono utilizzate per l'addestramento, il modello che ne risulterà fuori sarà inutilizzabile nel mondo reale. Il numero effettivo di *test* di unità realizzato è basso, essi sono una decina, con un *code coverage* difficilmente classificabile, tuttavia la parte di codice coperta, come spiegato precedentemente, è importante.

Mentre per il resto ho adottato un approccio manuale, controllando personalmente le varie parti del programma. Questo per via dell'ambito in cui il progetto si occupava, ovvero quello del *machine learning*. In questo campo la realizzazione dei vari *test* risulta parecchio complicata, dovuto principalmente alla natura probabilistica e dinamica dei modelli di predizione.

Per questo, mi è risultato più semplice e veloce nel controllare personalmente se le modifiche che svolgevo causavano problemi o meno.

3.5 Risultati raggiunti

3.5.1 Piano qualitativo

Il prodotto che ho realizzato lo si può definire come uno strumento utilizzabile per la creazione di modelli tramite l'addestramento di reti neurali artificiali.

In esso, l'utente può definire la struttura di modello su cui vuole svolgere l'addestramento, può definire se vuole o meno applicare funzionalità come il *down-sampling* oppure il *preprocessing* e infine può valutare le prestazioni di quello che ha ottenuto.

Questo strumento supporta diverse modalità di *training*, inoltre, il cambiamen-

to da una modalità è agevolato da un *file* di *config*, dove si possono modificare anche altre impostazioni. Raffiguro questo *file* attraverso 3.5.



```
1 {
2   "Dataset_Path": "./dataset/ptb-xl/",
3   "DICOM_Dataset_Path": "./dataset/DICOM/",
4   "Models_Path": "./models/",
5   "Evaluation_Path": "./evaluation/",
6   "Base_Sampling_Rate": 1000,
7   "Sampling_Rate": 250,
8   "Preprocessing": true,
9   "Binary": false,
10  "Number_of_Out_Classes": 5,
11  "Type": "multi-class"
12 }
```

Figura 3.5: Cattura dello schermo rappresentante il *file* di *config*

A fine dell'esecuzione del programma, l'utente può trovare le varie valutazioni all'interno di una cartella specifica contenuta nella *directory* del progetto. Di seguito mostro un'esempio dei contenuti prodotti:

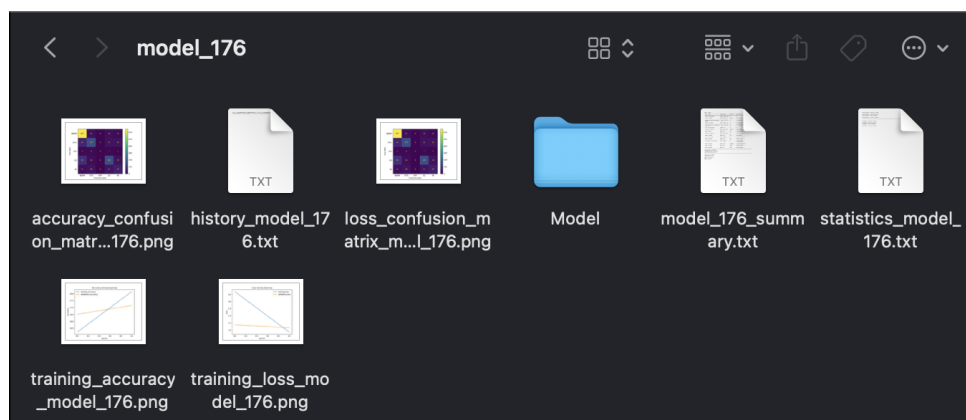


Figura 3.6: Cattura dello schermo rappresentante i *file* di *output*

Inoltre, durante allo *stage* avevo l'obiettivo di realizzare io stesso dei *training* di reti neurali artificiali. Perciò ho anche prodotto due modelli cercando di ottenere delle buone prestazioni. Essi sono due perchè ho salvato solo i risultati migliori che ho ottenuto per due tipi di classificazione: binaria, ovvero sano o

non sano, e multi-classe, dove il modello predice il risultato su cinque possibili classi.

3.5.2 Piano quantitativo

Sul piano quantitativo sono riuscito a soddisfare tutti i requisiti derivanti dall'analisi dei requisiti e rappresentati nella tabella 3.1.

Per quanto riguarda il *code coverage*, non avendo creato solo test automatici, ma svolto test manuali per via della natura del progetto: non posso quantificare in modo obiettivo questa metrica.

Tuttavia posso parlare dei risultati ottenuti con l'attività di addestramento.

<i>Binary model</i>	
<i>Accuracy</i>	86.25%
<i>Precision</i>	87.93%
<i>Recall</i>	81.07%
<i>Specificity</i>	90.63%
<i>F1 Score</i>	84.36%

Tabella 3.2: Tabella contenente le prestazioni del modello binario.

<i>Multi-class model</i>	
<i>Accuracy</i>	71.50%
<i>Precision</i>	51.11%
<i>Recall</i>	48.85%
<i>F1 Score</i>	49.65%

Tabella 3.3: Tabella contenente le prestazioni del modello multi-classe.

Come si può vedere nelle tabelle 3.2 e 3.3, i modelli creati da me non sono veramente ottimi, soprattutto se vengono seguiti gli *standard* richiesti in campo medico, dove si richiede un'accuratezza più alta possibile.

Tuttavia, questi risultati non sono causati da mie scelte errate di configurazione, ma al *dataset* utilizzato. Avendo utilizzato per la maggior parte il *dataset* pubblico PTB-XL di PhysioNet, ho potuto confrontare i miei risultati ottenuti con una ricerca scientifica che ha svolto i miei stessi tipi di addestramenti di reti neurali artificiali.

La pubblicazione scientifica in questione è «ECG Signal Classification Using Deep Learning Techniques Based on the PTB-XL Dataset», dove vengono mostrati dei risultati finali simili a quello che ho ottenuto, solo marginalmente migliori.

Capitolo 4

Retrospettiva

4.1 Conseguimento degli obiettivi

4.1.1 Obiettivi aziendali

Esposizione degli obiettivi aziendali raggiunti, facendo riferimento agli obiettivi descritti nelle sezioni precedenti.

4.1.2 Obiettivi personali

Esposizione degli obiettivi personali raggiunti, facendo riferimento agli obiettivi descritti nelle sezioni precedenti.

4.2 Competenze acquisite

Descrizione della maturazione professionale, in ambito di conoscenze e abilità.

4.3 Divario tra università e lavoro

Descrizione della distanza tra le competenze imparate in università e quelle richieste allo *stage*.

Glossario

API Un *Application Programming Interface* è un *set* di regole e protocolli che permettono a diverse applicazioni software di comunicare tra di loro. [13](#)

Batch Con il termine *batch*, si definisce una porzione di *dataset* che viene utilizzata per essere processata in un'insieme di dati in una singola iterazione di addestramento. È un punto chiave di un processo di ottimizzazione per fare in modo di processare in un modo più efficiente il *dataset*. [24](#)

Computer Vision La *computer vision* è un campo dell'intelligenza artificiale che si occupa di sviluppare sistemi in grado di interpretare e comprendere informazioni visive provenienti da immagini e video. [14](#)

Confidenza Un grafico della confidenza è una rappresentazione visuale che utilizza gli intervalli di confidenza per mostrare l'incertezza associata a stime statistiche. In questo ambito viene usato per capire quanto un modello di classificazione è sicuro delle predizioni che fa. [17](#), [23](#)

Dataset Un *dataset* è una collezione strutturata di dati, generalmente di grandi dimensioni, organizzata in forma relazionale. [10](#)

Deep Learning Con *deep learning* si intende una sotto-categoria del *machine learning*, che utilizza reti neurali artificiali che possiedono molteplici strati con lo scopo di analizzare e interpretare grandi moli di dati. [4](#), [10](#)

DICOM DICOM (*Digital Imaging and Communications in Medicine*) è uno standard usato in campo medico per l'immagazzinamento, trasmissione e gestione delle immagini medicali. [22](#)

Downsampling Il *downsampling* è una tecnica di processamento dei dati che va a ridurre il numero di dati in un “set”, ciò rende più semplice la gestione. Questo processo viene effettuato seguendo specifici metodi, che possono andare a causare la perdita di dati importanti se il livello di *downsampling* è troppo severo. [14](#)

Environment Gli *environment*, anche detti *virtual environment*, sono spazi dedicati per l’installazione di specifici pacchetti e le loro dipendenze, senza influenzare altri progetti o l’intero sistema. [12](#)

Epoca Un’epoca, nel contesto del *machine learning*, è una singola iterazione della fase di addestramento dove viene percorso l’intero *training dataset*. [21](#)

GPU L’acronimo GPU (*Graphics Processing Unit*) indica un componente *hardware* creato specificatamente per elaborare immagini e per il rendering grafico. Sono progettate per gestire moltissime operazioni contemporaneamente, tramite l’uso di specifiche librerie è possibile sfruttare questa caratteristica per l’ambito dell’intelligenza artificiale. [5](#)

Label Una *label*, anche detta etichetta, è un’annotazione che assegna una categoria ad uno specifico dato. Vengono utilizzate confrontare il risultato ottenuto da un modello con l’effettiva categoria. [22](#)

Machine Learning Con *machine learning* si intende una branca dell’intelligenza artificiale che si occupa di sviluppare algoritmi e modelli statistici per permettere ai computer di apprendere da dati e migliorare le proprie prestazioni, questo senza programmare istruzioni specifiche. [4](#), [10](#)

Matrice di Confusione La matrice di confusione è uno “strumento” utilizzato per valutare modelli di classificazione. Generalmente si tratta di una tabella quadrata dove vengono rappresentati i risultati delle predizioni del modello in confronto al vero valore. [23](#)

Prefetch Il *prefetch* è una tecnica utilizzata per caricare anticipatamente dei dati che verranno utilizzati a breve. [21](#)

Preprocessing Il *preprocessing* è uno step che spesso viene effettuato quando si effettua analisi di dati o applicazione di *machine learning*. In questo step si ha la trasformazione dei dati grezzi in una forma processata che è più adatta. [14](#)

Pull Request Sono una funzionalità generalmente trovata nei sistemi di controllo di versione, consente agli sviluppatori di proporre modifiche ad una sorgente di codice. Questa funzionalità facilita la collaborazione, permettendo la revisione e discussione di modifiche prima che esse vengano integrate. [6](#)

RAM La *Random Access Memory* è un componente presente nei *computer* che permette di immagazzinare dati che vengono letti e scritti, all'interno di essa, in modo rapido. Questa componente è necessaria per far funzionare programmi e per processare dati. [10](#)

Rete Neurale Artificiale Una rete neurale artificiale è un modello computazionale che si ispira alla struttura e al funzionamento del cervello umano. Questi modelli sono formati da unità di base che si connettono tra di loro per formare una rete. Le unità di base utilizzate si ispirano ad una versione parecchio semplificata dei neuroni biologici. [10](#)

TPU L'acronimo TPU (*Tensor Processing Unit*) indica un componente *hardware* progettato specificatamente per il calcolo di moltissime operazioni matematiche in contemporanea. Esse sono progettate e prodotte specificatamente per essere utilizzate nell'ambito del machine learning. [5](#)

Bibliografia

Articoli

Sandra, Śmigielski, Krzysztof Pałczyński e Damian Ledziński. «ECG Signal Classification Using Deep Learning Techniques Based on the PTB-XL Dataset». In: *Entropy* 23.9 (2021). DOI: [10.3390/e23091121](https://doi.org/10.3390/e23091121) (cit. a p. 34).

Sitografia