Università di Pisa

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

Localizzazione Indoor Basata su Beacon Bluetooth a Bassa Potenza Attraverso Tecniche di Deep Learning

Relatore:
Prof. GianLuigi Ferrari

Presentata da:
Marco Pampaloni

Anno Accademico 2019/2020

Sommario

I sistemi di localizzazione indoor, ovvero i sistemi che permettono la localizzazione di dispositivi all'interno di un ambiente chiuso, dove non è possibile sfruttare la copertura del sistema GPS, sono stati oggetto di notevole interesse. Questo elaborato illustra una soluzione tecnologica al problema della localizzazione indoor basata sull'uso di strumenti di machine learning. La soluzione proposta sfrutta una rete neurale convoluzionale (CNN) profonda. I dati di input del modello costituiscono una serie temporale di segnali broadcast *Bluetooth Low Energy* (BLE) emessi da un insieme di Beacon disposti all'interno dell'edificio oggetto della localizzazione indoor, mentre l'output è una coppia di coordinate relative alla posizione all'interno dell'edificio stesso. La soluzione inoltre utilizza alcune tecniche di *data augmentation* per generare un dataset di grandi dimensioni sulla base dei campionamenti dei segnali effettuati in loco.

A seguito dell'addestramento, il modello utilizzato ha mostrato un errore medio assoluto (MAE) sul dataset di test pari a *30cm*, esibendo una buona affidabilità anche rispetto a variazioni significative dei segnali dovute al rumore ambientale. Per ridurre ulteriormente l'errore medio è stato costruito un insieme di modelli, ognuno addestrato con diversi iperparametri. Questa tecnica ha permesso di ridurre l'errore medio fino a circa *26cm*.

Il modello prodotto risulta eseguibile in tempo reale su dispositivi mobile con ridotte capacità computazionali, rendendolo particolarmente adatto alla cosiddetta navigazione "blue-dot" all'interno di contesti Indoor. Tuttavia le varie fluttuazioni dell'output del modello tendono ad originare una navigazione poco fluida. Per arginare questo problema è stato applicato un filtro di Kalman al modello e viene sfruttato il sensore inerziale dello smartphone per produrre un'euristica utile a individuare i movimenti dell'utente.

A Snoopy

Indice

Capitolo 1

Introduzione

1.1 Localizzazione Indoor

Il problema della Localizzazione Indoor consiste nell'individuazione di un utente all'interno di uno spazio chiuso, in riferimento a un sistema di coordinate predefinito. Tale sistema di coordinate, relativo ad un determinato edificio, può essere poi espresso in termini georeferenziali conoscendo la precisa dislocazione geografica del locale in questione.

La localizzazione indoor apre le porte a diverse possibilità nel campo dell'esperienza utente all'interno di edifici pubblici, nel settore della gestione dei flussi di persone, della sicurezza e della contingentazione. Tali problematiche si fanno ancora più rilevanti a fronte della recente epidemia di Covid-19 che ha colpito il pianeta. Attraverso l'impiego di tale tecnologia è possibile coadiuvare la navigazione degli utenti all'interno di edifici complessi, assicurare il rispetto delle norme di distanziamento sociale interpersonale e migliorare l'esperienza individuale di persone affette da disabilità. Per ottenere questi risultati è però richiesto un certo grado di precisione, di affidabilità, di efficienza e di sicurezza nella gestione della privacy dei dati di localizzazione degli utenti. Inoltre la tecnologia scelta per risolvere il problema, per essere fruibile, deve avere come ulteriore requisito il basso impatto economico.

1.2 Soluzioni Tecnologiche

Nel corso degli anni sono state implementati diversi sistemi di localizzazione indoor, che possiamo dividere in due macrocategorie: soluzioni ad-hoc e soluzioni che sfruttano tecnologie esistenti. Nel primo caso si fornisce all'utente l'attrezzatura necessaria ad essere localizzato, mentre nel secondo si utilizza un dispositivo mobile di proprietà dell'utilizzatore. Spesso tale dispositivo è uno smartphone.

I sistemi che implementano tecnologie sviluppate ad-hoc, sono spesso più efficienti, più precisi e flessibili. Tuttavia, il loro impiego rimane limitato a causa dell'alto costo di progettazione, installazione e di gestione. È poi richiesto che ad ogni utente che intenda essere localizzato sia assegnato un dispositivo che si interfacci col sistema impiegato.

Per l'impiego su larga scala, un sistema di localizzazione indoor deve essere facilmente utilizzabile dalle masse e non deve richiedere particolari requisiti tecnologici.

1.3 Bluetooth Low Energy

La tecnologia *Bluetooth* è talmente pervasiva che ogni smartphone in circolazione ne implementa il protocollo, mostrandosi particolarmente adeguata alla risoluzione del problema in esame. Nello specifico, *Bluetooth Low Energy* (BLE) è un protocollo che riduce notevolmente il consumo energetico dei dispositivi che lo utilizzano. La tecnologia BLE viene utilizzata dalle moderne API di *contact tracing* sviluppate da Apple e Google, nonchè dalla applicazione Immuni[1] per il tracciamento dei contagi di Covid-19[2].

La soluzione riportata in questo elaborato prevede l'utilizzo di una serie di beacon BLE programmabili, ciascuno installato in un punto significativo dell'edificio e configurato per emettere un segnale broadcast con una frequenza di circa 50Hz. La potenza dei segnali viene quindi utilizzata per produrre, attraverso l'utilizzo di una rete neurale artificiale, una coppia di coordinate rappresentative della posizione dell'utente all'interno dell'edificio. Ciò viene reso possibile da una fase preliminare in cui viene mappata la superficie del locale raccogliendo i segnali ricevuti dai beacon in vari punti. Per ogni

punto della superficie mappato si registra una serie temporale di segnali, dei quali si considera solo il valore *RSSI*, ovvero la potenza del segnale nel punto in cui questo viene ricevuto.

Il modello utilizzato è di fatto completamente agnostico rispetto all'ubicazione dei beacon installati, supponendo che questa sia unica e non mutabile nel tempo.

L'utilizzo di tale sistema assicura il completo anonimato dell'utente, il quale non necessita di condividere la propria posizione, essendo quest'ultima calcolata direttamente sul suo smartphone in funzione dei segnali che riceve.

Questo elaborato si pone l'obiettivo di descrivere nello specifico la rete neurale progettata per risolvere il problema, le tecniche utilizzate per alzare il grado di precisione del modello e le principali differenze rispetto a modelli già esistenti.

1.4 Sperimentazione e Collaborazioni

Il progetto esposto in questo elaborato è stato realizzato per conto del Consorzio Metis e la sperimentazione è stata eseguita presso i locali dell'ASL Toscana Nord Ovest di Pisa. Il lavoro è stato svolto in autonomia, ma sia il Consorzio Metis che l'ASL hanno predisposto i propri ambienti per l'installazione dei Beacon Bluetooth e la raccolta dei dati.

Capitolo 2

Deep Learning

In questo capitolo saranno introdotti i concetti fondamentali alla base delle moderne tecniche di Deep Learning e le strutture matematiche necessarie alla loro comprensione.

2.1 Machine Learning

Il *Machine Learning*, o apprendimento automatico, è un insieme di tecniche e algoritmi che consente a dei programmi di "imparare" a svolgere un determinato compito sulla base di esperienze pregresse, senza bisogno da parte del programmatore di specificare come eseguire tali mansioni.

2.1.1 Regressione Lineare

Un classico esempio di algoritmo di machine learning è quello della regressione lineare. Scopo dell'algoritmo è predire l'output di una determinata funzione. Si consideri quindi un vettore $\boldsymbol{x} \in \mathbb{R}^n$, e un valore scalare $\hat{y} = \boldsymbol{\theta}^\intercal \boldsymbol{x}$. Il vettore $\boldsymbol{\theta}$ introduce i parametri del modello, mentre $\hat{y}$ ne rappresenta l'output, che è una funzione lineare di $\boldsymbol{x}$. Siano quindi $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m$ dei vettori in $\mathbb{R}^n$ e $y_1, y_2, \ldots, y_m$ i corrispettivi valori della funzione $f(\boldsymbol{x}_i)$ che stiamo cercando di approssimare.

Perchè la funzione $\hat{y}(\boldsymbol{x})$ approssimi $f(\boldsymbol{x})$ è necessario che i parametri $\boldsymbol{\theta}$ del modello si adattino in modo da minimizzare la differenza tra l'output prodotto dal modello e la cosiddetta *ground truth*: $y = f(\boldsymbol{x})$. A questo scopo si definisce una metrica di errore propria del processo di apprendimento: l'errore quadratico medio (MSE dall'inglese)

$$MSE = \frac{1}{m} \sum_{i=1}^{m} (\hat{y} - y_i)^2 \tag{2.1}$$

Per minimizzare l'errore sul nostro dataset di test è sufficiente porre a zero la derivata, rispetto a $\boldsymbol{\theta}$, della nostra funzione di costo. Nel caso della regressione lineare è possibile risolvere l'equazione risultante ottenendo un sistema di equazioni che prende il nome di *normal equations*. Esistono tuttavia metodi numerici iterativi che si basano sulle informazioni fornite dal gradiente della funzione di costo che permettono di aggiornare i parametri del modello cercando di ridurne l'errore, anche nel caso di modelli non lineari. L'algoritmo su cui si basano molti dei moderni metodi di apprendimento del Deep Learning è il *gradient descent* o metodo del gradiente.

Il metodo del gradiente aggiorna i parametri $\boldsymbol{\theta}$ del modello secondo la seguente regola:

$$\boldsymbol{\theta}' = \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{2.2}$$

dove $J(\boldsymbol{\theta})$ rappresenta la funzione di costo associata al modello, mentre η è un coefficiente chiamato *learning rate*. Un'interpretazione dell'algoritmo è data dalle informazioni sulla monotonia ottenute dal gradiente della funzione di costo: per η abbastanza piccolo risulta $J(\boldsymbol{\theta}') \leq J(\boldsymbol{\theta})$ poichè il gradiente negativo di J determina la direzione di massima decrescita della funzione[3]. Ne consegue che applicando ricorsivamente la regola di aggiornamento del metodo del gradiente, la nostra funzione $\hat{y}$ tenderà ad avvicinarsi alla funzione originale y, minimizzando la funzione di costo. Possiamo intepretare il coefficiente η come la velocità con cui seguire la pendenza della funzione di errore.

Nel caso di un modello di regressione lineare che usa l'MSE come funzione di costo,

risulta:

$$\begin{aligned}
&\forall j \in 1, \dots, n : \\
\frac{\partial}{\partial \theta_j} MSE &= \frac{\partial}{\partial \theta_j} \frac{1}{m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2 \\
&= \frac{\partial}{\partial \theta_j} \frac{1}{m} \sum_{i=1}^{m} (\boldsymbol{\theta}^\intercal \boldsymbol{x}_i - y_i)^2 \\
&= \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial \theta_j} (\boldsymbol{\theta}^\intercal \boldsymbol{x}_i - y_i)^2 \\
&= \frac{2}{m} \sum_{i=1}^{m} (\boldsymbol{\theta}^\intercal \boldsymbol{x}_i - y_i) x_i^{(j)}
\end{aligned}$$

Ovvero abbiamo che la regola di aggiornamento è:

$$\theta_j' = \theta_j - \eta \frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}) = \theta_j - \eta (\frac{2}{m} \sum_{i=1}^{m} (\boldsymbol{\theta}^\intercal \boldsymbol{x}_i - y_i) x_i^{(j)}), \quad \forall j \in \{1, \dots, n\}$$

2.1.2 Perceptron

Il *Perceptron* è il modello che ha posto le basi per le moderne reti neurali artificiali e il deep learning. Esso prende spunto dalla neurologia, cercando di imitare il comportamento dei neuroni del cervello umano, con ovvie limitazioni e senza presunzione di volerne fornire una simulazione accurata del funzionamento. Una schematizzazione del modello è descritta in Figura 2.1

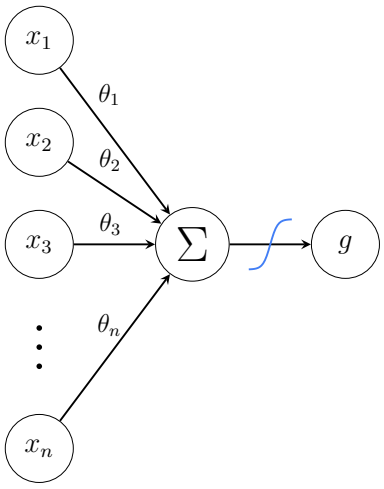


Figura 2.1: Schematizzazione del Perceptron

Il perceptron è molto simile al modello di regressione lineare descritto precedentemente, ma si distingue per un fattore fondamentale: la non linearità. L'output del perceptron è infatti definito come:

$$\hat{y}(\boldsymbol{x}) = g(\boldsymbol{\theta}^\intercal \boldsymbol{x})$$

dove g è una funzione *sigmoidea*, cioè una funzione che ha un andamento a "S", con due asintoti orizzontali come in Figura 2.2: di solito è utilizzata la funzione logistica $g(x) = \dfrac{1}{1+e^{-x}}$.

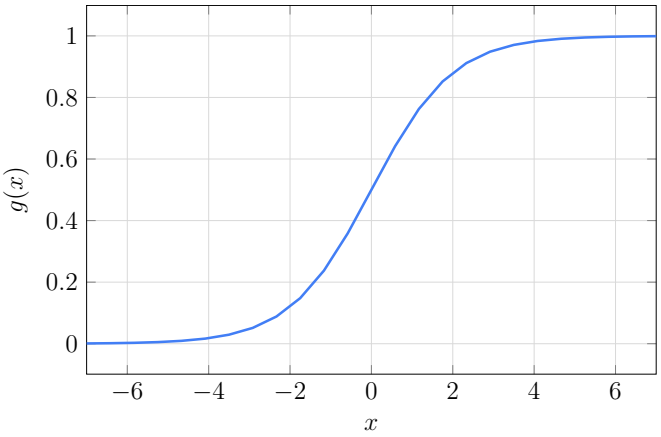


Figura 2.2: Funzione sigmoidea $g(x) = \dfrac{1}{1+e^{-x}}$

Malgrado la nonlinearità del modello, il perceptron non è in grado di approssimare molte classi di funzioni. È famoso l'esempio della funzione XOR, la quale non può essere imparata dal perceptron. Per questo motivo è stata sviluppata un'estensione del modello che prende il nome di *Multi Layer Perceptron*.

2.2 Multi Layer Perceptron

Il Multi Layer Percepton (MLP) è uno dei modelli più emblematici del Deep Learning. Esso si basa sui concetti descritti finora ed è la naturale estensione del perceptron. L'MLP risolve infatti il principale problema del modello su cui è fondato, essendo in grado di approssimare qualsiasi tipo di funzione continua con precisione arbitraria, sotto l'assunzione che la funzione di attivazione sia non polinomiale[4].

L'MLP è un modello di machine learning che fa parte della categoria delle reti neurali, in quanto è composto da un insieme di neuroni artificiali (perceptron) disposti su più livelli e interconnessi tra di loro. Una schematizzazione del modello è fornita in Figura 2.3.

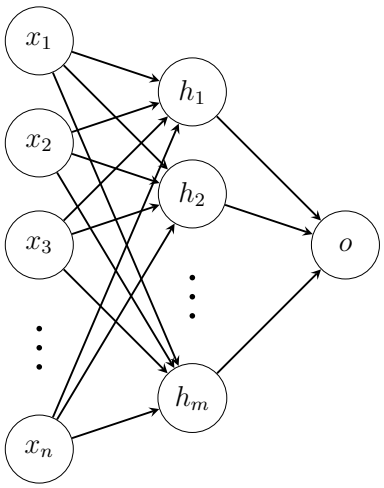


Figura 2.3: Schematizzazione del Multi Layer Perceptron: in Figura è mostrata una rete neurale con n valori di input, un singolo hidden layer con m neuroni e un solo output. Si noti che non si è limitati ad usare un solo neurone di output. Sono omessi per chiarezza i parametri del modello e le funzioni di attivazione, riassunte in questo caso all'interno di ogni neurone.

Nello specifico, l'MLP presenta un primo livello N-dimensionale che coincide con il suo input, un livello di output M-dimensionale e una serie arbitraria di livelli intermedi (*hidden layers*) di ampiezza variabile. Ogni nodo del livello precedente è connesso con ogni neurone del livello successivo e ogni neurone del modello si comporta come un singolo perceptron, ovvero esegue una somma pesata degli input ricevuti dal livello precedente e produce in output il risultato attraverso una funzione di attivazione non-lineare. Per questo motivo i parametri del modello, cioè i coefficienti con cui vengono sommati gli input di ogni nodo, sono uno per ogni connessione. Questo tipo di modelli vengono anche chiamati *Feed Forward Networks*, in quanto le connessioni tra nodi sono dirette soltanto verso i livelli direttamente successivi, e mai il contrario.

Sia $\boldsymbol{\Theta}^{\ell}$ la matrice dei pesi delle connessioni tra il livello $\ell - 1$ e quello successivo, definita in modo che Θ_{ij}^{ℓ} rappresenti il coefficiente relativo alla connessione tra il neurone i-esimo del livello ℓ e il neurone j-esimo del livello $\ell - 1$. Questa configurazione della matrice dei coefficienti, nonostante sia poco intuitiva, permette di esprimere l'insieme di valori uscenti da un generico livello ℓ sotto forma di prodotto: $\boldsymbol{h}^{\ell} = \boldsymbol{\Theta}^{\ell}\boldsymbol{h}^{\ell-1}$. Se L è il numero di livelli intermedi della rete neurale, risulta:

$$\begin{aligned} o(\boldsymbol{x}) &= \boldsymbol{\Theta}^{L+1}\boldsymbol{h}^{L} \\ &= \boldsymbol{\Theta}^{L+1} g(\boldsymbol{\Theta}^{L}\boldsymbol{h}^{L-1}) \\ &= \boldsymbol{\Theta}^{L+1} g(\boldsymbol{\Theta}^{L} g(\dots g(\boldsymbol{\Theta}^{1}\boldsymbol{x}))) \end{aligned}$$

2.3 BackPropagation

Per utilizzare il metodo del gradiente con una rete neurale Feed Forward, è necessario calcolare la derivata della funzione di costo del modello. Tuttavia nell'MLP essa dipende sia dai parametri del livello immediatamente precedente che da tutti i parametri dei livelli più bassi, per ricorsione. Si può pensare al BackPropagation come un modo per propagare all'indietro le informazioni relative all'errore commesso dai nodi di output rispetto ai target del dataset di addestramento.

L'algoritmo calcola il gradiente della funzione di costo in modo automatico applicando ricorsivamente lungo il grafo di computazione di $J(\boldsymbol{\Theta})$ la "regola della catena", ovvero la regola di derivazione per le funzioni composte: $(f \circ g)' = (f' \circ g) \cdot g'$. Esso sfrutta la tecnica della programmazione dinamica per evitare di ricalcolare più volte la derivata di branch comuni dell'albero di computazione, rendendolo di fatto un metodo molto efficiente.

2.4 Attivazione: ReLU

Come descritto, la funzione di attivazione conferisce al modello la nonlinearità. Ciò è fondamentale perchè l'MLP riesca ad approssimare funzioni continue arbitrarie. Se non ci fosse alcuna funzione di attivazione o se questa fosse lineare rispetto ai parametri del modello, l'output della rete si potrebbe esprimere come una semplice applicazione lineare, rendendolo di fatto equivalente a un semplice modello di regressione lineare.

Con l'avvento delle reti neurali profonde e di nuove architetture più complesse, l'utilizzo delle funzioni di attivazioni logistiche è andato calando. Una delle cause è il cosiddetto "vanishing gradient", un problema che insorge nel calcolare numericamente il gradiente della funzione di costo di una rete neurale con molti livelli. In questo caso l'uso della funzione sigmoidea è in generale sconsigliato poichè il suo valore "satura" a causa dei suoi asintoti e questo rende la sua derivata prossima a zero. Il metodo del gradiente risulta quindi poco efficace nell'aggiornare i parametri del modello. Per questo motivo è stata introdotta la funzione ReLU (dall'inglese *rectified linear unit*) definita nel modo seguente: $g(x) = \max\{0, x\}$. L'operazione di rettificazione è mostrata in Figura 2.4.

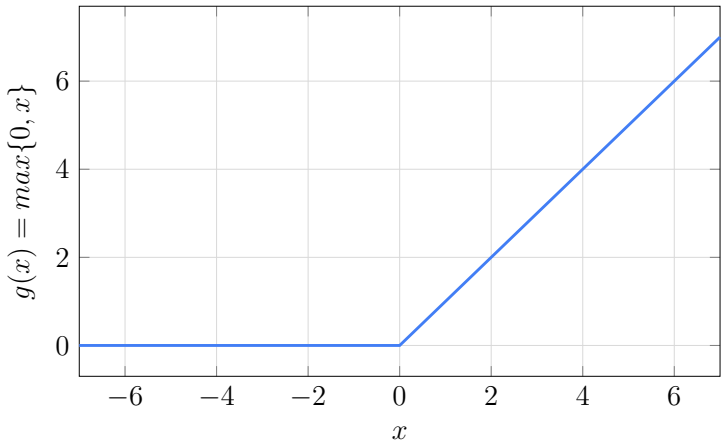


Figura 2.4: La funzione di attivazione ReLU

L'output del rettificatore, essendo esso quasi una funzione lineare, mantiene le informazioni del gradiente in modo migliore rispetto al sigmoide, rendendo la convergenza del metodo del gradiente più veloce.

2.5 Reti Neurali Convoluzionali

Una particolare categoria di reti feed forward è quella delle reti neurali convoluzionali, o *convolutional neural networks* (CNN) in inglese. Una CNN non è altro che una rete neurale che adotta l'operazione di *convoluzione* in almeno uno dei suoi layer, al posto della più comune moltiplicazione matriciale.

L'operazione di convoluzione, usualmente indicata con il simbolo $*$, è definita per due funzioni continue f e g come:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(x)k(t-x)dx$$

mentre, nel caso discreto, è invece definita nel seguente modo:

$$(f * g)(t) = \sum_{x=-\infty}^{\infty} f(x)k(t-x)$$

Poichè in una rete neurale le funzioni f e g sono rappresentate generalmente da tensori, l'operazione può essere limitata ai soli elementi di tali insiemi di valori, con l'assunzione che le due funzioni siano nulle ovunque tranne nei punti in cui sono definiti i tensori. Nel caso di input bidimensionali, come è quello delle immagini, si ha quindi:

$$\begin{aligned}(X * K)(i,j) &= \sum_m \sum_n X(m,n)K(i-m, j-n) \\ &= \sum_m \sum_n X(i-m, j-n)K(m,n)\end{aligned}$$

in cui X rappresenta l'input dell'operazione, mentre K è chiamato *kernel*, o filtro e costituisce i parametri adattivi della CNN. Nella pratica è tuttavia più comune utilizzare l'operazione di *cross-correlation*, definita come

$$(K * X)(i,j) \sum_m \sum_n X(i+m, j+n)K(m,n)$$

Tale operazione equivale a spostare il filtro K lungo le due dimensioni dell'input X, eseguendo una moltiplicazione elemento per elemento tra i due tensori e sommandone i risultati. Un esempio è illustrato in Figura 2.5.

2.5.1 Pooling

L'output della convoluzione è chiamato *feature map* e generalmente ne vengono calcolate svariate decine per ogni livello, ognuna attraverso l'applicazione di un filtro diverso sullo stesso input. Il numero di feature map aumenta generalmente con il crescere della profondità della rete. Questo accade per far fronte alla riduzione dimensionale causata dai livelli di *pooling*. Questi ultimi hanno lo scopo di rendere il modello parzialmente invariante rispetto a piccole traslazioni o disturbi nell'input. Ciò avviene applicando un'operazione di sottocampionamento (*subsampling*) a sottoinsiemi, disgiunti o no, dell'input del livello. Tale operazione può prevedere la selezione del massimo degli elementi del sottoinsieme, oppure della loro media aritmetica. L'operazione di pooling è

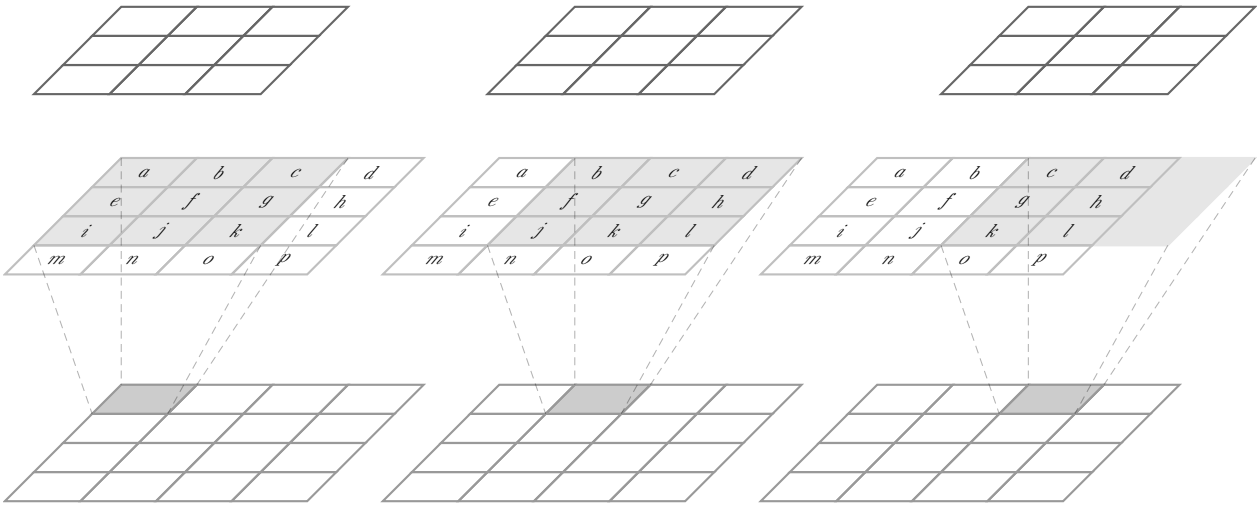


Figura 2.5: Spostamento di un filtro 3×3 lungo una matrice di dimensioni 4×4. Nel primo e secondo riquadro il filtro rientra completamente nelle dimensioni della matrice, mentre nel terzo caso una colonna risulta al di fuori. Gli elementi del filtro che vengono proiettati esternamente alla matrice vengono moltiplicati con valori nulli e non contribuiscono al valore finale della convoluzione.

illustrata in Figura 2.6.

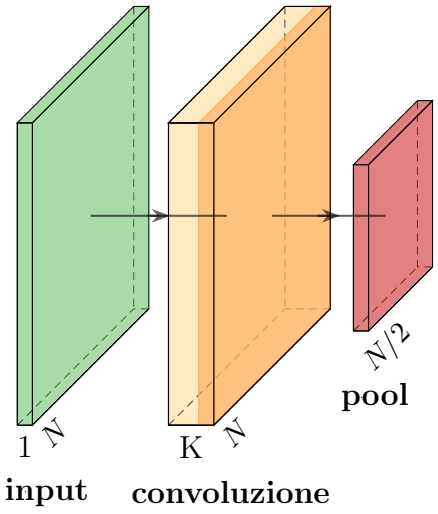


Figura 2.6: Schematizzazione di un semplice blocco convoluzionale a cui è applicata l'operazione di pooling. L'input è bidimensionale e con un solo canale (per esempio l'itensità dei pixel di un'immagine in bianco e nero) e vengono prodotte K feature map. Il livello di pooling dimezza le dimensioni dell'input.

Capitolo 3

Architettura Software

In questo capitolo è descritta nel dettaglio l'architettura software sviluppata per il progetto di locallizazione indoor, inclusa la rete neurale, le librerie utilizzate, gli ambienti di sviluppo e gli strumenti che hanno coadiuvato il testing e la sperimentazione del prototipo realizzato.

3.1 TensorFlow

TensorFlow è una libreria open source Python che fornisce metodi e costrutti per definire grafi computazionali e calcolarne automaticamente la derivata. Ciò consente di costruire reti neurali arbitrariamente complesse ed addestrarle tramite le implementazioni degli algoritmi di apprendimento fornite dalla libreria. La versione della libreria utilizzata è TensorFlow 2.2.1, la quale consente di definire funzionalmente modelli di machine learning, interfacciandosi con Keras, una API ad alto livello progettata per essere comprensibile e di facile utilizzo.

3.2 Google Colab

L'ambiente di sviluppo principalmente utilizzato durante il processo di prototipazione è stato Google Colab: una piattaforma cloud hosted progettata da Google per l'esecuzione

di Notebook Python e pensata per lo sviluppo di modelli di machine learning. Google Colab mette a disposizione gratuitamente sistemi di elaborazione con acceleratori computazionali quali GPU e TPU.

3.3 Weights & Biases

Per la gestione sistematica dei test, dei risultati sperimentali e per l'ottimizzazione degli iperparametri del modello, è stato utilizzato Weights & Biases. Esso fornisce una suite di strumenti per il tracking degli esperimenti di machine learning grazie alla registrazione automatica delle metriche di errore durante la fase di addestramento, del salvataggio dei modelli e ai vari tool grafici accessibili dalla piattaforma web di Weights & Biases.

3.4 Architettura della Rete Neurale

La rete neurale sviluppata per il problema di localizzazione indoor è illustrata schematicamente in Figura 3.1. Essa consiste in una serie di blocchi convoluzionali seguiti da alcuni livelli di neuroni completamente connessi. Il modello sfrutta, oltre ai segnali RSSI emessi dai beacon, anche due input ausiliari che non sono processati dalla sezione convoluzionale della rete.

3.4.1 Input del Modello

L'input principale del modello è composto da una serie temporale di valori RSSI relativi ai segnali emessi da 15 beacon disposti lungo il perimetro dell'edificio nel quale si sono svolte le sperimentazioni del prototipo. La dimensione temporale dell'input può essere arbitrariamente lunga, poichè una sua variazione determina solamente una differente dimensione dell'asse temporale dell'output della CNN. La sezione convoluzionale della rete si conclude infatti con un livello di pooling globale che consiste nell'estrazione, per ogni feature map prodotta, della media aritmetica dei valori di input lungo la

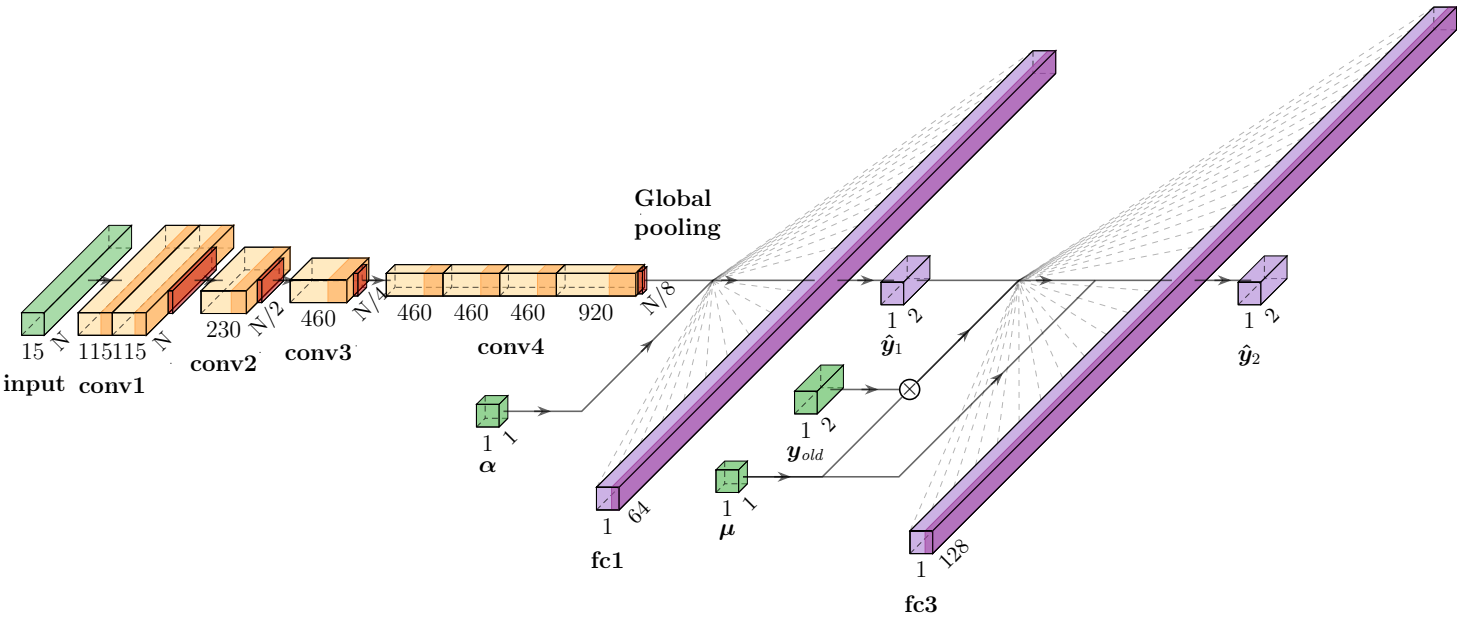


Figura 3.1: Architettura della rete neurale

dimensione del tempo. Come si vede in Figura 3.1, infatti, la dimensione dell'output di tale livello è sempre costante e risulta essere 920×1.

A completare l'input del modello sono il valore emesso dal sensore magnetico dello smartphone e l'ultima posizione nota dell'utente all'interno dell'edificio, indicati rispettivamente con α e $\boldsymbol{y}_{old}$. Il valore della bussola è utile per determinare l'orientamento della persona nello spazio, rendendo la rete neurale capace di considerare le variazioni dei segnali BLE dovuti all'assorbimento da parte del corpo dell'utilizzatore dello smartphone. Il secondo input ausiliario è invece utilizzato per correggere eventuali scostamenti rilevanti dell'output della CNN rispetto alla posizione precedente dell'utente. Ci si aspetterebbe infatti che tale posizione non variasse di molto in un lasso di tempo breve.

L'ultima posizione nota dell'utente viene pesata da un coefficiente, anche esso input del modello, che in Figura 3.1 è indicato con la lettera greca μ. Tale valore, cui possiamo riferirci con il termine *coefficiente di memoria residua*, è compreso tra zero e uno, e determina il peso che si vuole dare all'ipotesi di continuità della posizione dell'utente nel tempo. $\mu = 0$ indica l'assenza di tale assunzione, con la conseguente massima riduzione della correzione dell'output della CNN da parte dei livelli successivi, mentre $\mu = 1$ associa il massimo peso su tale ipotesi. Il coefficiente di memoria residua è esso stesso input del livello successivo della rete, il quale riceve anche i valori dell'output ausiliario e di $\mu \cdot \boldsymbol{y}_{old}$.

Il valore della bussola è input del primo livello completamente connesso, insieme all'output della CNN.

3.4.2 Blocco Convoluzionale

La prima parte del modello è una semplice rete neurale convoluzionale unidimensionale. Sebbene sia composto da due dimensioni, quella temporale e quella dei beacon, quest'ultima può essere interpretata come l'insieme dei canali della prima, come nel caso dei canali r, g, b di un'immagine a colori. Ciò permette di applicare l'operazione

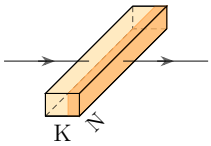


Figura 3.2: Singolo blocco convoluzionale: la parte chiara indica l'operazione di convoluzione, mentre l'ombreggiatura sulla destra illustra la funzione di attivazione ReLU. K è il numero di filtri utilizzati e di conseguenza equivale al numero di feature map prodotte, mentre N è la lunghezza della serie temporale. Il processo di Batch normalization è omesso dalla schematizzazione.

di convoluzione soltanto lungo l'asse temporale. La CNN proposta è composta da otto blocchi convoluzionali consecutivi così strutturati:

- Operazione di convoluzione sull'output del livello precedente
- Funzione di attivazione ReLU sull'output della convoluzione
- Livello di batch normalization

Un esempio di blocco convoluzionale è illustrato in Figura 3.2, mentre in Figura 3.1 sono mostrati anche i livelli di pooling, rappresentati da una superficie rossa apposta accanto i blocchi convoluzionali.

L'output del secondo, terzo e quarto blocco convoluzionale sono sottoposti ciascuno all'applicazione di un particolare metodo di dropout chiamato *dropout gaussiano*. Esso consiste nell'applicare del rumore moltiplicativo, con distribuzione gaussiana di media unitaria, all'output del blocco convoluzionale. Lo scopo è quello di simulare una corruzione casuale dei dati di addestramento del modello, con l'obiettivo di regolarizzare quest'ultimo. Applicare il rumore all'output dei livelli intermedi della rete, piuttosto che al dataset iniziale, consente di manipolare più profondamente la rappresentazione dei dati imparata dal modello, rendendolo conseguentemente più robusto rispetto alle variazioni dei segnali di input[5]. Gli effetti dell'utilizzo del dropout gaussiano sono illustrati in Tabella 3.1.

L'output dell'ultimo blocco convoluzionale è infine seguito da un livello di pooling globale e dall'applicazione del dropout.

Modello	Loss	MAE	RMSE	MaxAE
Baseline	0.7796	0.3070	0.6716	3.001
No Dropout Gaussiano	0.8911	0.3171	0.7138	3.256

Tabella 3.1: Modello *baseline* messo a confronto con una versione dello stesso che non utilizza il dropout gaussiano. Le metriche fanno riferimento al dataset di *test*.

Modello	Loss	MAE	RMSE	MaxAE
Baseline	0.7796	0.3070	0.6716	3.001
No Bussola	1.619	0.4470	0.9784	4.500

Tabella 3.2: Varie metriche a confronto per i due modelli in esame: *Baseline* è il modello finale, mentre il secondo differisce dal primo soltanto dall'uso dei valori della bussola, che vengono semplicemente scartati. Le metriche si riferiscono al dataset di *test*, ovvero al processo di valutazione successivo alla fase di addestramento.

3.4.3 Uso della Bussola e Output Ausiliario

L'utilizzo dei valori forniti dal sensore magnetico dello smartphone sono giustificati dal voler mitigare il cosiddetto effetto del *body shadowing*. Tale fenomeno si verifica quando un segnale wireless si propaga in un ambiente e collide contro un corpo umano. Tale collisione provoca un decadimento del segnale, il quale arriva disturbato al punto di ricezione. Ciò influisce sulla precisione dei sistemi di localizzazione indoor basati sui valori RSSI dei segnali wireless in modo considerevole, poichè è sufficiente che l'utente volti le spalle a un sottoinsieme dei beacon attivi per introdurre rumore all'interno del sistema. Utilizzando il valore emesso dalla bussola dello smartphone, è possibile rendere il modello consapevole dell'orientamento dell'utente e attenuare il rumore introdotto dal *body shadowing*. I risultati ottenuti dall'utilizzo di questo input sono illustrati in Tabella 3.2, la quale mette in relazione il modello finale con uno che non considera l'input della bussola.

L'input corrispondente al sensore magnetico è un valore scalare $\alpha \in \mathbb{R}$, con $0 \leq \alpha \leq 360$, in cui il valore 0 indica il nord. Tale dato è fornito, come mostrato in Figura 3.1, al primo livello completamente connesso della rete, insieme all'output della CNN. Questo livello produce un vettore di 64 elementi, il quale diventa input del secondo livello

completamente connesso del modello. L'output di quest'ultimo livello, indicato in figura come $\hat{\boldsymbol{y}}_1$, è una coppia di coordinate reali che indica la posizione dell'utente all'interno dell'edificio. Tale previsione è soltanto parziale, in quanto non tiene conto dell'ultima posizione nota dell'utente, ma risulta utile per guidare l'addestramento del modello verso una soluzione meno dipendente da tale input. A questo scopo $\hat{\boldsymbol{y}}_1$ è interpretrato dal modello come output ausiliario.

A ogni output ausiliario è associata una funzione di costo, il cui valore va poi integrato additivamente con quello della funzione di costo dell'output principale. Nel caso della rete neurale in esame si ha:

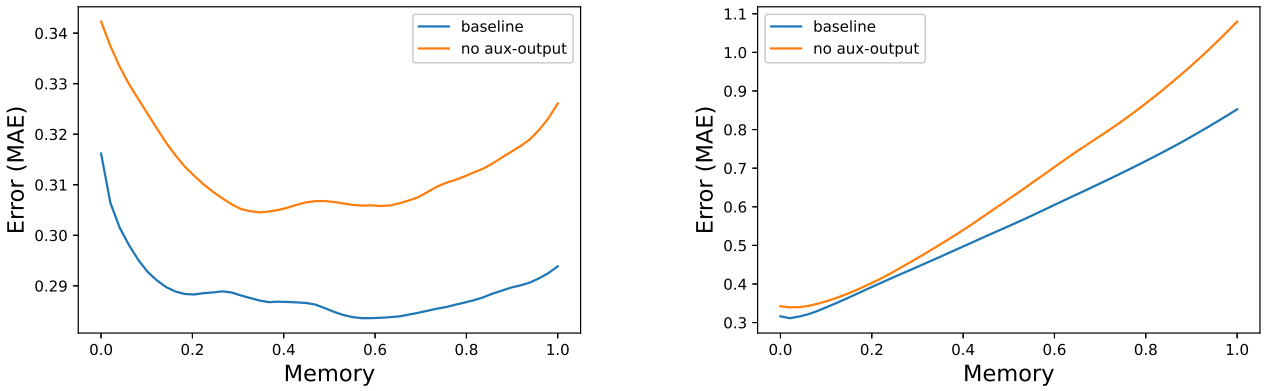
$$J_1(\boldsymbol{\Theta}) = \frac{1}{m}\sum_{i=1}^{m} \|\hat{\boldsymbol{y}}_{1i} - \boldsymbol{y}_i\|_2^2$$
$$J_2(\boldsymbol{\Theta}) = \frac{1}{m}\sum_{i=1}^{m} \|\hat{\boldsymbol{y}}_{2i} - \boldsymbol{y}_i\|_2^2$$
$$J(\boldsymbol{\Theta}) = c_1 J_1(\boldsymbol{\Theta}) + c_2 J_2(\boldsymbol{\Theta})$$

in cui la scelta dei parametri c_1 e c_2 determina il peso di ciascuna funzione di costo nel bilancio dell'errore totale: il modello implementato assegna ai coefficienti i valori $c_1 = \frac{1}{2}, c_2 = 1$. Si noti che tali valori sono completamente arbitrari e rappresentano due iperparametri del modello.

Se l'output $\hat{\boldsymbol{y}}_1$ non pesasse direttamente nella funzione di costo del modello (cioè se fosse $c_1 = 0$), l'output della rete sarebbe troppo condizionato dal valore dell'input ausiliario $\boldsymbol{y}_{old}$. Poichè in fase di raccolta dei dati, la posizione precedente dell'utente non è nota, si assume che tale variabile aleatoria sia distribuita secondo una distribuzione normale centrata nella posizione del campionamento e con deviazione standard σ pari a una piccola costante, indicativa della variabilità del moto di una persona mentre cammina (per esempio $\sigma = 1$). Ciò implica che, qualora fosse $\sum_i \|\boldsymbol{y}_{old\,i} - \boldsymbol{y}_i\|_2^2 < \sum_i \|\hat{\boldsymbol{y}}_{2i} - \boldsymbol{y}_i\|_2^2$, cioè se la distanza media tra l'ultima posizione nota e l'effettiva posizione dell'utente durante il campionamento dei segnali, fosse minore della precisione media ottenibile dal modello, i parametri della rete convergerebbero verso dei valori che tenderebbe-

ro a ignorare l'input dei beacon. L'utilizzo della sola posizione precedente dell'utente per esprimere l'output del modello, garantirebbe infatti un errore inferiore rispetto al considerare anche i valori RSSI dei segnali.

Utilizzando l'output ausiliario descritto, pesato con un coefficiente non nullo, viene garantito che lo scenario appena descritto non si verifichi, a patto che il coefficiente selezionato sia sufficientemente grande. In Figura 3.3 il modello descritto è messo a confronto con uno in cui l'output ausiliario non ha peso all'interno della funzione di costo ($c_1 = 0$).



(a) Deviazione standard $\sigma = 1$

(b) Deviazione standard $\sigma = 5$

Figura 3.3: Analisi dell'utilizzo dell'output ausiliario: i grafici mostrano l'andamento dell'errore (MAE) commesso dai due modelli sul dataset di test al variare del coefficiente di memoria residua. La curva blu rappresenta il modello finale, mentre la curva arancione indica il modello senza output ausiliario (cioè in cui $c_1 = 0$). Nel grafico a sinistra, l'input della posizione precedente ($\boldsymbol{y}_{old}$) è perturbato con del rumore gaussiano la cui deviazione standard è pari a $\sigma = 1$. A destra invece $\sigma = 5$. L'input perturbato determina una stima meno precisa dell'ultima posizione nota dell'utente e, con il crescere del coefficiente di memoria residua, entrambi i modelli tendono a sovrastimare l'importanza di tale input. Tuttavia, come si evince dai grafici, l'errore commesso dal modello *baseline* cresce più lentamente ed è sempre minore di quello commesso dal modello senza output ausiliario.

3.4.4 Output del Modello

L'output del modello, indicato in Figura 3.1 come $\hat{\boldsymbol{y}}_2$, consiste in una coppia di coordinate reali che indica la posizione prevista dell'utente all'interno dell'edificio, dopo essere

Modello	Loss	MAE	RMSE	MaxAE
No Augmentation	1.2090	0.4072	0.7947	3.481
Replicazione Dataset	1.076	0.3168	0.8344	3.958
Baseline	0.7796	0.3070	0.6716	3.001

Tabella 3.3: Metriche di errore ottenute dal dataset di *test* per tre modelli diversi: il primo non processa in alcun modo i dati originali, il secondo replica il dataset più volte e l'ultimo è il modello finale comprensivo di data augmentation. Come si può notare, l'ultimo modello è il migliore per tutte le metriche esaminate, ma è molto vicino al secondo per quanto riguarda il MAE. Tuttavia la differenza è notevole in termini di RMSE, indicando una varianza più alta nel modello privo di data augmentation.

stata opportunamente corretta in base alle conoscenze relative alla precedente locazione dell'utilizzatore.

3.5 Dataset Augmentation e Preprocessing

Poichè il processo di raccolta dei dati è particolarmente dispendioso, sono state impiegate tecniche di *data augmentation* al fine di arricchire il dataset di addestramento. Un insieme di dati più grande corrisponde spesso a una maggiore precisione del modello e consente di utilizzare architetture più profonde. Il problema dell'overfitting descresce infatti di intensità con il tendere della cardinalità del dataset di addestramento a infinito.

Le tecniche di data augmentation possono essere interpretate come dei metodi per regolarizzare il modello: esse si basano infatti sull'idea di introdurre rumore all'interno dei dati e rendere quindi la rete neurale più robusta alla corruzione delle informazioni di input. Modellare in qualche modo questo tipo di rumore casuale all'interno del modello permette di limitare la dipendenza dell'output da eventuali variabili latenti non osservate.

In Tabella 3.3 sono mostrati i risultati sperimentali ottenuti dall'utilizzo delle tecniche di data augmentation descritte nel seguito.

3.5.1 Jittering

Si può assumere che l'errore intrinseco della raccolta dei dati, dovuto a limiti tecnologici o a fenomeni non osservati, segua una distribuzione gaussiana a media nulla. Risulta quindi possibile alterare l'input del modello sommandolo con un cosiddetto rumore bianco, ottenendo così un dataset di dimensione doppia rispetto all'originale.

Sia $D = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}$ il dataset di addestramento, con $\boldsymbol{x}_i \in \mathbb{R}^n$. Siano poi $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_m$ variabili aleatorie con distribuzione normale multidimensionale tali che:

$$\boldsymbol{z}_i \in \mathbb{R}^n,$$
$$\boldsymbol{z}_i \sim \mathcal{N}(0, \Sigma).$$

La matrice Σ, detta di covarianza, determina la varianza della distribuzione lungo ciascuna dimensione dello spazio. In questo contesto si assume $\Sigma = \sigma^2 I$. Applicando il rumore gaussiano ad ogni input iniziale si ottiene $D' = \{\boldsymbol{x}_1 + \boldsymbol{z}_1, \ldots, \boldsymbol{x}_m + \boldsymbol{z}_m\}$. Il dataset finale consiste nell'unione dei due insiemi $D \cup D'$.

Un esempio illustrativo del jittering è mostrato in Figura 3.4.

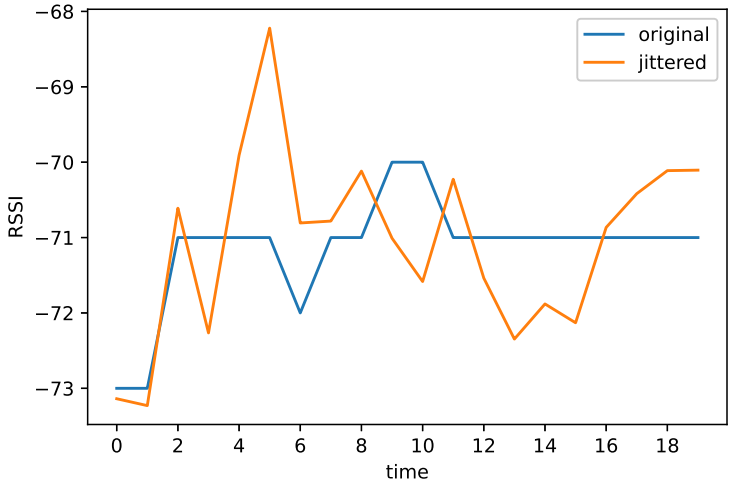


Figura 3.4: Jittering applicato ai segnali emessi da un beacon in un intervallo di tempo limitato. L'asse x descrive il tempo, mentre l'asse y indica la potenza del segnale ricevuto, in dB, al momento t.

3.5.2 Ridimensionamento (Scaling)

La tecnica del ridimensionamento consiste nell'applicare del rumore moltiplicativo costante al segnale di ingresso, mantenendone quindi intatta la struttura, ma modificandone l'ampiezza. L'utilizzo dello *scaling* è giustificato dall'assunzione che, per lo scopo della localizzazione indoor, l'informazione contenuta in una serie temporale di segnali è invariante rispetto a piccole variazioni dell'ampiezza dei segnali, poichè queste potrebbero essere causate da fenomeni esterni non osservati.

Utilizzando la stessa notazione della sezione precedente, definiamo una variabile aleatoria $c \sim \mathcal{N}(1, \sigma^2)$ con distribuzione gaussiana a media unitaria e varianza σ^2. Il singolo elemento del dataset a cui è applicato il ridimensionamento è quindi definito come:

$$\boldsymbol{x}' = c\boldsymbol{x}$$

Un esempio di scaling applicato ai segnali emessi da un beacon è illustrato in Figura 3.5.

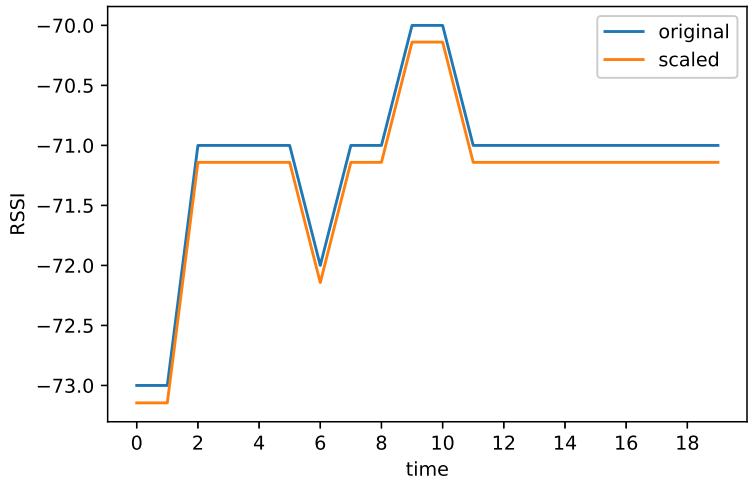


Figura 3.5: Illustrazione grafica del ridimensionamento di un segnale.

3.5.3 Magnitude Warping

Data la natura delle perturbazioni a cui sono soggetti i segnali dei beacon nel contesto della localizzazione indoor, è lecito pensare che tali distorsioni possano verificarsi anche in periodi particolarmente limitati del tempo e con una intensità relativamente elevata. Tale assunzione permette di sviluppare una tecnica di data augmentation a cui possiamo dare il nome di *magnitude warping*, la quale prevede di deformare l'intensità del segnale in punti limitati della serie temporale, applicando del rumore moltiplicativo a una porzione di sottosequenze, non necessariamente disgiunte, del segnale di input. In Figura 3.6 è mostrato graficamente l'utilizzo di tale tecnica. Il Listato 3.1 descrive invece i dettagli implementativi del magnitude warping.

Algorithm 3.1 Descrizione algoritmica del funzionamento del magnitude warping

function MAGNITUDEWARPING($\boldsymbol{v}$, σ^2, *MaxLength*, *MaxPeaks*)
  $\boldsymbol{v}' \leftarrow \boldsymbol{v}$
  sample $n \sim$ *Uniform*(1, *MaxPeaks*)
  for $peak = 0, \ldots, n$ **do**
    sample $\ell \sim$ *Uniform*(1, *MaxLength*)
    sample $p \sim$ *Uniform*(0, $length(\boldsymbol{v}) - \ell$)
    sample $m \sim \mathcal{N}(1, \sigma^2)$
    for $i = p, \ldots, p + \ell$ **do**
      $\boldsymbol{v}'[i] \leftarrow m \cdot \boldsymbol{v}'[i]$
  return $\boldsymbol{v}'$

3.5.4 Permutazione di Sottoinsiemi (Subset Shuffling)

Un'altra assunzione che è possibile fare riguardo l'input del problema è che l'ordine con cui si verificano variazioni all'interno della serie temporale non dovrebbe avere peso nell'estrazione di informazioni o di pattern dai segnali. Per questo motivo è stata implementata una tecnica di *shuffling* in cui sottoinsiemi dell'input vengono scambiati tra di loro secondo permutazioni casuali. Tale metodo è illustrato in Figura 3.7

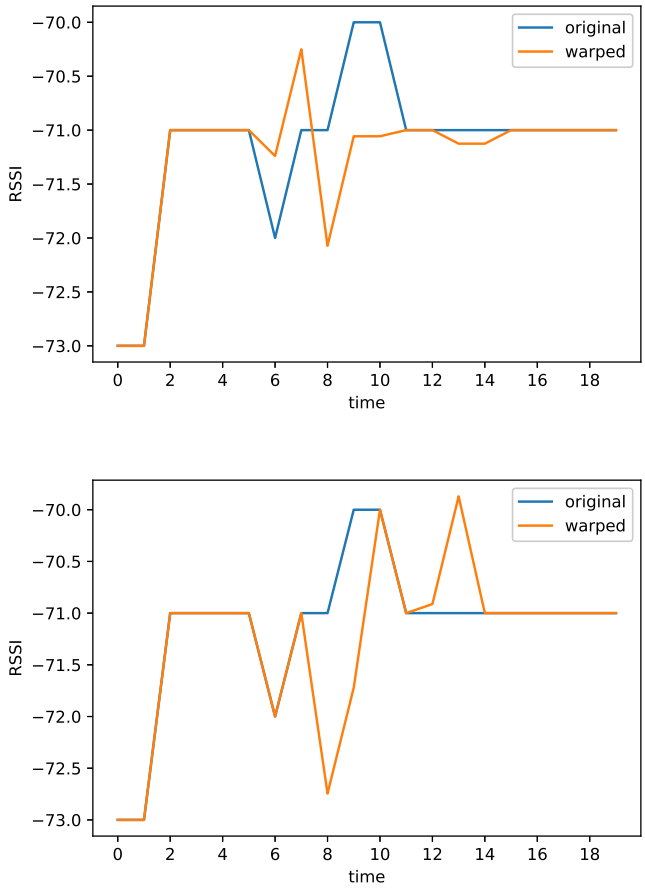


Figura 3.6: Esempi grafici di magnitude warping applicato a un segnale.

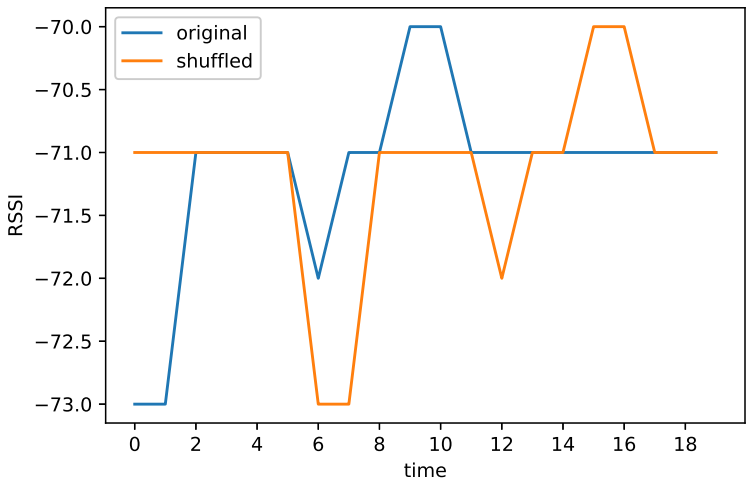


Figura 3.7: Shuffling di sottoinsiemi applicato a una serie temporale.

3.5.5 Deattivazione Selettiva

Per rendere il modello più robusto alla eventuale perdita di segnali di alcuni beacon, è stata introdotta una tecnica di data augmentation che prevede la conversione dei valori della serie temporale di un beacon in valori soglia indicano la mancata ricezione di segnale. A rappresentare il placeholder con il quale rimpiazzare gli effettivi valori RSSI dei beacon, nell'ambito di questo progetto, è stato scelto il valore -200db.

Questa strategia è simile a quella del dropout, ma, al posto del valore nullo, l'input viene rimpiazzato casualmente con il placeholder scelto. A gestire la frequenza con cui un beacon viene disattivato all'interno del dataset è un iperparametro di tipo percentuale.

3.6 Addestramento del Modello

Il modello descritto è stato addestrato su un dataset di 279457 elementi, ottenuti applicando le tecniche di data augmentation introdotte precedentemente. L'algoritmo utilizzato per l'addestramento è una variante del metodo del gradiente stocastico con momento, che prende il nome di *Adam*. Questo ottimizzatore gestisce autonomamente l'iperparametro della rete che determina l'entità dell'aggiornamento dei parametri del modello: il cosiddetto *learning rate*. A coadiuvare l'algoritmo è stato inserito un metodo, guidato dal dataset di validazione, che riduce esponenzialmente il learning rate ogni volta che la procedura di addestramento raggiunge uno stallo.

Un ulteriore metodo di regolarizzazione introdotto è quello dell'*Early Stopping*, il quale termina precocemente l'addestramento quando una determinata metrica sul dataset di validazione non migliora per un periodo di tempo predefinito. Quando ciò accade, la configurazione del modello che ha ottenuto i migliori risultati viene ripristinata. Empiricamente si è visto che il metodo dell'early stopping riduce l'errore commesso dal modello sul dataset di validazione, ma è stato anche mostrato come tale algoritmo agisca da regolarizzatore per la rete in un modo simile a quello della regolarizzazione

Modello	MAE	RMSE	MaxAE
Baseline	0.3070	0.6716	3.001
Ensemble	0.2592	0.5536	2.4693

Tabella 3.4: Confronto tra il modello *baseline* e un ensemble di cinque modelli addestrati individualmente. Le metriche di errore si riferiscono al dataset di test.

L2[6].

Infine, al termine della procedura di addestramento, il modello ottenuto viene valutato sul dataset di test e i risultati vengono salvati per essere in futuro analizzati e confrontati con altri modelli.

3.7 Ensembling

Al fine di ottenere un errore di generalizzazione minore e migliorare ulteriormente l'affidabilità del sistema di localizzazione indoor, sono stati addestrati indipendentemente, e con differenti configurazioni di iperperparamentri, vari modelli, i quali sono stati successivamente aggregati tramite il metodo dell'*ensembling*. Tale tecnica consiste nell'unire le previsioni di modelli diversi per formarne una il cui errore sia statisticamente più basso, a patto che gli errori commessi dai singoli modelli siano sufficientemente non correlati[3]. In un problema di regressione, ad esempio, per aggregare l'output di un insieme di modelli è possibile utilizzare la media aritmetica delle varie previsioni.

Un ulteriore vantaggio dell'utilizzo di un ensemble di modelli, è la possibilità di ottenere un'indicazione sulla variabilità della previsione, calcolando la deviazione standard dell'output dei modelli. Ciò fornisce un'informazione sulla confidenza dell'ensemble relativamente alla previsione di un determinato output. Tale informazione può essere poi sfruttata al fine di stabilizzare il modello con un filtro di Kalman, come descritto nel Paragrafo 4.4.2. Utilizzando un ensemble di cinque modelli è stato possibile ottenere i risultati illustrati in Tabella 3.4.

Capitolo 4

Applicazione Mobile

Questo capitolo introduce una descrizione dell'applicazione mobile sviluppata come prototipo per il progetto di navigazione indoor realizzato per conto del Consorzio Metis e ASL Toscana. I seguenti paragrafi analizzano le tecnologie software usate, l'architettura dell'applicazione e i metodi utilizzati per produrre una navigazione indoor fluida sulla base del modello di machine learning prodotto precedentemente.

4.1 Flutter

L'applicazione mobile è stata sviluppata utilizzando Flutter[7], un framework per lo sviluppo cross-platform di applicazioni. Tramite Flutter è possibile utilizzare una singola codebase per produrre applicativi eseguibili nativamente su architetture desktop, smartphone Android e iOS, e sul web. Il linguaggio di programmazione utilizzato da Flutter è Dart, un linguaggio funzionale general purpose recentemente sviluppato da Google.

4.2 Planimetrie e Poligoni

Per implementare la navigazione indoor, l'applicazione è stata dotata di un visualizzatore di planimetrie. Queste ultime, prima di essere processate dal componente grafico,

vengono convertite in un insieme di poligoni ciasuno indicante un vano o un elemento del locale da visualizzare. I poligoni vengono poi disegnati a schermo in modo da mostrare la planimetria all'utente, insieme alla sua presunta posizione all'interno dell'edificio. In Appendice B è mostrata l'interfaccia delle varie schede dell'applicazione, incluso il visualizzatore di planimetrie.

4.3 Backend TensorFlow

L'applicazione proposta sfrutta le librerie di TensorFlow per caricare in memoria il modello di machine learning descritto nel capitolo 3. Le stesse librerie permettono di utilizzare tale modello per eseguire inferenze sui dati.

Per ottimizzare le perfomance del modello su dispositivi mobile dotati di ridotte capacità computazionali, il modello è stato prima compresso tramite gli strumenti forniti da TensorFlow Lite[8]. Tensorflow Lite permette di minimizzare lo spazio occupato in memoria del modello e contemporaneamente di ottimizzare le operazioni in virgola mobile quantizzandone gli operandi. Ciò comporta un'esecuzione a runtime più veloce e tempi di latenza minori durante la navigazione, al costo di una precisione inferiore che però non compromette significiativamente il risultato della computazione.

4.4 Stabilizzazione del Modello

Come descritto precedentemente, le fluttuazioni dell'output prodotto dal modello potrebbero inficiare l'esperienza d'uso dell'applicazione da parte dell'utente. Seppure la precisione media del modello sia molto bassa, errori isolati e rapidi scostamenti dal valore atteso sono possibili e non eliminabili. Di conseguenza si è reso necessario applicare alcune tecniche di *denoising* in modo da rendere la navigazione più fluida e costante. Tra queste vi è l'utilizzo di un filtro di Kalman e dei sensori inerziali dello smartphone.

4.4.1 Utilizzo di Sensori Inerziali

Tutti i moderni smartphone contengono al loro interno alcuni sensori inerziali, i quali possono essere sfruttati per determinare la posizione dell'utente nello spazio e il loro stato di moto. In particolare l'accelerometro è un sensore che misura le variazioni dell'accelerazione subite dal dispositivo lungo i tre assi dello spazio. Tali valori vengono collezionati periodicamente e possono essere utilizzati per individuare la velocità dell'utente che si muove all'interno dell'edificio. Poichè l'accelerazione è la derivata della velocità, le componenti di quest'ultima possono essere ricavate dalla relazione:

$$v(t) = v_0 + \int_{t_0}^{t} a(t)dt.$$

Per calcolare numericamente il valore di tale integrale è stata utilizzata la regola del trapezio, la quale permette di approssimare il valore di un integrale definito dividendone l'intervallo in n sottonsiemi disgiunti e calcolando per ciascuno di essi l'area del trapezio contenuto tra l'asse delle ascisse e la funzione in quell'intervallo. La somma di queste aree parziali rappresenta una approssimazione dell'integrale richiesto, la quale migliora al crescere di n. Denotando con $\tilde{v}(t)$ l'approssimazione dell'area dell'integrale dell'accelerazione ricavata dal sensore al tempo t, si ha:

$$\tilde{v}(t) = v_0 + \frac{t - t_0}{n} \sum_{i=0}^{n-1} \frac{a(t_i) - a(t_{i+1})}{2}.$$

La precisione dei sensori inerziali degli smartphone attuali è tuttavia piuttosto bassa ed è difficile ottenere una stima accurata della velocità relativa del dispositivo tramite metodi numerici, i quali approssimano solamente la soluzione corretta. Per questi motivi si è scelto di non utilizzare direttamente il valore calcolato della velocità dell'utente o il suo orientamento, ma di limitare la valutazione del suo stato di moto al calcolo di un coefficiente indicativo della confidenza con cui si asserisce che l'utente si stia muovendo.

Tale coefficiente è definito come

$$\phi = \tanh(|\tilde{v}|).$$

L'uso della funzione tanh e del valore assoluto permette di proiettare il codominio della funzione nell'intervallo $(0, 1)$. In questo modo si può considerare $\phi = 0$ come assenza di moto, mentre al tendere di ϕ al valore asintotico 1 cresce la confidenza con cui si attesta che l'utente sia in movimento.

4.4.2 Filtro di Kalman

Un filtro di Kalman è un particolare filtro ricorsivo che corregge lo stato di un sistema dinamico attraverso l'analisi di misurazioni soggette a qualche forma di rumore. Nel caso di questo elaborato, il sistema dinamico è rappresentato dall'utente che si muove all'interno dell'edificio e le misurazioni corrispondono all'output del modello di localizzazione indoor descritto nel capitolo 3.

Il filtro di Kalman permette di sfruttare le conoscenze del sistema dinamico e dei parametri di errore del sistema di misurazione in modo da ottenere stime più precise. A tale scopo si può utilizzare l'euristica prodotta a partire dal sensore inzeriale e definita precedentemente come $\phi = \tanh(|\tilde{v}|)$. Tale coefficiente viene moltiplicato con la stima indiretta della velocità dell'utente ottenuta attraverso output successivi del modello. Si definisce quindi la velocità lungo un asse del piano in cui si muove l'utente come:

$$v = \phi \frac{y_{t+1} - y_t}{\Delta t}$$

dove con y_t e y_{t+1} si indica (una componente del) l'output del modello in due istanti di tempo consecutivi e con Δt l'intervallo di tempo trascorso tra i due.

Il filtro di Kalman necessita poi di conoscere la varianza del sistema che emette le misurazioni, la quale si può ottenere dall'ensemble di modelli come descritto nel paragrafo 3.7. Utilizzando queste informazioni e, sebbene l'implementazione realizzata

sia più un'euristica che un vero filtro di Kalman, è stato possibile riddure buona parte delle oscillazioni durante la navigazione.

Capitolo 5

Implementazione e Criteri di Sperimentazione

Questo capitolo descrive il metodo di sperimentazione adottato per il progetto di localizzazione indoor ed espone l'implementazione delle principali componenti software sviluppate. Nonostante l'interfaccia grafica dell'applicazione mobile sia anche essa stata sviluppata in modo originale, per non appesantire la trattazione si è scelto di non mostrarne l'implementazione, ma si rimanda al Capitolo 4 e all'Appendice B per maggiori dettagli riguardanti le funzionalità e l'effettivo aspetto grafico.

5.1 Dettagli Sperimentali

Il prototipo del software di navigazione è stato sviluppato per la sede dell'ASL Toscana Nord Ovest di Pisa. Nell'ambito di questa prima sperimentazione, il locale è stato allestito per gestire la localizzazione lungo un corridoio di circa 50 metri situato al primo piano dell'edificio. Il processo sperimentale utilizzato si può suddividere in quattro fasi:

- Dislocamento dei beacon
- Raccolta dei dati
- Addestramento del modello

- Test di navigazione

Ciascuno di questi punti è descritto nel dettaglio nei paragrafi successivi.

5.1.1 Dislocamento dei Beacon

Per la sperimentazione presso l'ASL sono stati predisposti 15 beacon Bluetooth programmabili. In particolare, sono stati adottati degli ESP32[1] e configurati per emettere un segnale BLE broadcast alla massima frequenza ammissibile dall'hardware e dallo standard (50Hz). Il codice sorgente, in linguaggio C, con cui sono stati programmati i microcontrollori è illustrato nel Listato 5.1. A ciascun beacon è stato associato un identificatore univoco corrispondente agli ultimi tre ottetti del suo indirizzo MAC, scartando di fatto l'identificativo del produttore, insieme a un prefisso condiviso.

I beacon, alimentati via presa USB da 5V, sono stati installati in 15 stanze diverse perimetrali al corridoio adibito alla sperimentazione, in modo da coprirne tutta la lunghezza. La disposizione dei beacon è stata pensata in modo da poter ricevere, in ciasun punto del corridoio, il segnale di almeno 8 beacon. Ad eccezione di uno dei microcontrollori, il quale è stato collegato a una fotocopiatrice, i restanti beacon sono stati collegati ai PC presenti nei vari uffici. Per lo scopo di questa sperimentazione, si è reso sufficiente che i Computer fossero accesi soltanto durante la fase di campionamento e di testing del prototipo.

5.1.2 Raccolta dei Dati

Prima di procedere all'acquisizione dei segnali è stato misurato preventivamente il corridoio e identificato mentro per metro i punti in cui effettuare il campionamento. È stata poi installata l'applicazione mobile su uno smartphone Android e si è proceduto a effettuare la raccolta dei dati muovendosi lungo il corridoio e soffermandosi su ogni locazione individuata precedentemente durante la fase di misurazione. Per ogni

[1]Gli ESP32 sono una famiglia di microcontrollori a basso consumo energetico con modulo Wi-Fi e Bluetooth integrato. Il loro costo esiguo li rende particolarmente adatti ad essere impiegati nel contesto della localizzazione indoor.

```
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEServer.h>
#include <esp_system.h>

void setup() {
  uint8_t chipid[6];
  char name[256];
  /* Read ESP's MAC address */
  esp_efuse_read_mac(chipid);
  /* Take last 3 octects and discard organisation's identifier */
  snprintf(name, 256, "BEACON_ASL-%02X%02X%02X", chipid[3], chipid[4], chipid[5]);

  BLEDevice::init(name);
  BLEDevice::setPower(ESP_PWR_LVL_P9); /* Maximum power level */
  BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
  /* Minimum interval for BLE is 20ms:
   * (0x20 = 32d --> interval = 0.625*32 = 20ms --> 50Hz) */
  pAdvertising->setMinInterval(0x20);
  pAdvertising->setMaxInterval(0x20);
  pAdvertising->setScanResponse(false);
  pAdvertising->setMinPreferred(0x0);
  BLEDevice::startAdvertising();
}

void loop() { /* Empty loop */ }
```

Listato 5.1: Codice sorgente ESP32

punto selezionato, sono stati rilevati i segnali dei beacon secondo diversi orientamenti dell'utente, registrando le variazioni della bussola dello smartphone. L'interfaccia dell'applicazione per la raccolta dei dati è mostrata in Appendice B, mentre la componente dell'applicazione sviluppata per la raccolta dei dati e il loro processamento è illustrata nel Listato 5.2, in codice Dart.

Durante la fase di acquisizione, i segnali intercettati vengono convertiti in uno stream continuo, il quale viene poi trasformato raggruppando i dati, secondo la frequenza impostata, in porzioni di uguale dimensione. Tale trasformazione è mostrata nel Listato 5.3 ed è necessaria per assicurare che il sistema raccolga i segnali ad intervalli regolari e che, per ogni intervallo, il numero di segnali raccolti sia lo stesso. Qualora la dimensione del campione raccolto sia inferiore al valore scelto, il trasformatore di stream si occupa anche di aggiungere o rimuovere dati da ciascuna porzione di segnali.

Il componente DataCollector avvia e processa lo stream trasformato, aggiundendovi l'ubicazione del punto del campionamento (x, y) (in coordinate relative rispetto alla planimetria dell'edificio), i dati del sensore magnetico dello smartphone registrati nel momento dell'acquisizione dati, e una coppia di coordinate casuali x_{old}, y_{old} tali che $x_{old} \sim \mathcal{N}(\sigma^2,\, \mu_1 = x)$ e $y_{old} \sim \mathcal{N}(\sigma^2,\, \mu_2 = y)$. Tali coordinate rappresentano la presunta posizione precedente dell'utente, utilizzata dalla rete neurale come input ausiliario, come descritto nel Paragrafo 3.4.1. L'utilizzo di variabili aleatorie gaussiane è giustificato dal fatto che, non conoscendo a priori il percorso che potrebbe seguire l'utente all'interno dell'edificio, la posizione di quest'ultimo prima che raggiunga la coordinata (x, y) è probabile che sia nelle immediate vicinanze di tale ubicazione.

5.1.3 Addestramento del Modello

Una volta composto il dataset, il modello è stato addestrato secondo i metodi descritti dal Paragrago 3.6 sfruttando inizialmente le risorse gratuite di calcolo, ma limitate, offerte da Google Colab. Per rendere la procedura di addestramento più rapida e flessibile, si è scelto poi di affittare GPU attraverso servizi di Cloud Computing a

```
class DataCollector {
  /* class fields */
  ...

  DataCollector(this._selectedDevices,
      {this.target = const Point(0.0, 0.0),
      this.interval = const Duration(milliseconds: 1000),
      this.chunkSize = 30}) {
    ...
  }

  Stream<List<num>> startCollecting({Duration timeout}) {
    _compass.listen();
    Stream<List<num>> signalsStream = _bluetooth
        .scan(timeout: timeout, scanMode: ScanMode.lowLatency)
        .transform(ChunkCollector(_selectedDevices, interval, chunkSize));

    // Add other parameters to the data
    return signalsStream.map((signals) {
      // Sample the previous location from N(p; [x,y]^T, sigma*I)
      final s = 0.5;
      final previousX = Normal.generate(1, mean: target.x, variance: s).first;
      final previousY = Normal.generate(1, mean: target.y, variance: s).first;
      return [
        previousX,
        previousY,
        target.x,
        target.y,
        _compassValue,
        ...signals
      ];
    });
  }

  ...
}
```

Listato 5.2: DataCollector (Dart): avvia lo stream di acquisizione dei segnali.

```dart
class ChunkCollector extends StreamTransformerBase<ScanResult, List<num>> {
  /* class fields */
  ...

  ChunkCollector(this.devices, this.interval, this.size);

  // add {avg(list)}^(list.length - size) elements to list.
  Iterable<int> _normalizeList(Iterable<int> list, int size) {
    if (list.length > size) {
      return list.take(size);
    }
    if (list.length < size) {
      final int mean = (list.reduce((a, b) => a + b) / list.length).round();
      return List.from(list)..addAll(List.filled((size - list.length), mean));
    }
    return list;
  }

  List<int> processChunk(Map<String, List<int>> signals) {
    // Make sure we sampled from all of the selected devices.
    // If not, add a placeholder (-200) for that device
    for (String deviceName in devices) {
      signals.putIfAbsent(deviceName, () => [-200]);
    }
    // Normalize and flatten the measurements into a row vector
    final flattened =
        signals.values.expand((xs) => _normalizeList(xs.toList(), size)).toList();
    return flattened;
  }

  Stream<List<num>> bind(Stream<ScanResult> stream) async* {
    // Connect to the stream and transform it in chunks of size this.size
    ...
  }
}
```

Listato 5.3: ChunkCollector (Dart): si occupa di processare lo stream di segnali in ingresso.

pagamento e di implementare una serie di script per gestire il deploy delle procedure di addestramento sull'hardware remote.

L'addestramento della rete neurale ha impiegato mediamente 50 minuti per terminare utilizzando una GPU Nvidia RTX 2080 Ti, mentre le varie procedure di ottimizzazione degli iperparametri sono durate ciascuna tra le 15 e le 20 ore su hardware similari.

5.1.4 Test di Navigazione Indoor

Terminata la procedura di addestramento e prodotto un modello funzionante, la fase di testing prevedeva la valutazione dell'efficacia del sistema di navigazione provandolo direttamente all'interno dell'edificio adibito alla sperimentazione. Tuttavia la recente emergenza Covid-19 ha reso impossibile concludere i test del prototipo in loco, per ragioni di sicurezza sanitaria, all'interno dei locali dell'ASL. Sono risultati quindi determinanti nella valutazione dell'applicazione i precedenti test effettuati presso la sede del Consorzio Metis, seppur meno significativi e rigorosi per via delle molto inferiori dimensioni dello stabile.

5.2 Data Augmentation

Di seguito sono mostrate le implementazioni Python delle tecniche di data augmentation presentate nel Paragrafo 3.5. Tali metodi sono stati incapsulati in una classe TimeseriesDataAugmenter, la quale è servita a rendere il codice più leggibile e funzionale. L'implementazione di tale classe è mostrata nel Listato 5.4, nel quale sono state omesse le implementazioni dei singoli metodi per semplicità di trattazione. Il costrutto `TimeseriesDataAugmenter` permette di trasformare il dataset di input applicando internamente i metodi di data augmentation descritti in precedenza e le cui implementazioni sono fornite in questo Paragrafo.

```
class TimeseriesDataAugmenter():
    def __init__(self, dataset, target,
            jittering=True,
            shuffling=True,
            scaling=True,
            magnitude_warp=True):
        self.dataset = np.copy(dataset)
        self.target = np.copy(target)

    def augment(self,
            iterations=10,
            seed=42,
            deactivation_rate=0.2,
            scale_sigma=0.008,
            warp_sigma=0.04):
        ...
    def jitter(self, v, sigma=0):
        ...
    def mag_warp(self, v, mu=1, sigma=0.1, max_length=3, max_peaks=4):
        ...
    def replace_dataset_entries(self, dataset, placeholder=-200, p=0.2):
        ...
    def scale_dataset_entries(self, dataset, sigma=0.005):
        ...
    def shuffle_dataset_entries(self, dataset, permutations):
        ...
```

Listato 5.4: Interfaccia della classe TimeseriesDataAugmenter. La classe si occupa di arricchire il dataset di input applicando le trasformazioni definite nel Paragrafo 3.5

```
def jitter(self, v, sigma=0):
    return np.add(v, np.random.normal(0, sigma, v.shape))
```

Listato 5.5: Implementazione Jittering: applica del rumore additivo gaussiano a media nulla a tutti i segnali del dataset

```
# Apply a costant multiplicative random noise to each time series
def scale_dataset_entries(self, dataset, sigma=0.005):
  scales = np.random.normal(1.0, sigma, dataset.shape[0])
    return np.array([scale * xs for (xs, scale) in zip(dataset, scales)])
```

Listato 5.6: Implementazione Scaling: permette di applicare del rumore moltiplicativo gaussiano di media unitaria a tutti gli input del dataset.

```python
# Shuffle each entry of the given 2D dataset according to an array of permutations
def shuffle_dataset_entries(self, dataset, permutations):
    return np.array([np.concatenate(np.array(np.array_split(xs, len(p)))[p])
        for (xs, p) in zip(dataset, permutations)])
```

Listato 5.7: Implementazione Shuffling: consente di riorganizzare i segnali del dataset in modo che la nuova configurazione segua la permutazione passata in input. Ciascuna serie temporale del dataset viene divisa in un numero di sottoinsiemi mutualmente esclusivi pari alla lunghezza dell'array di permutazione specificato. Tali sottogruppi vengono poi poi riorganizzati come specificato.

```python
def mag_warp(self, v, mu=1, sigma=0.1, max_length=3, max_peaks=4):
    new_v = np.copy(v)
    for row in range(v.shape[0]):
        peaks = round(np.random.uniform(1, max_peaks))
            for peak in range(peaks):
                length = round(np.random.uniform(1, max_length))
                    application_point = np.random.randint(0, v.shape[1]-length)
                    mass = np.random.normal(mu, sigma)
                    for point in range(application_point,
                    ↪  application_point+length):
                        new_v[row][point] *= mass
                            return new_v
```

Listato 5.8: Implementazione Magnitude Warping: si rimanda al Paragrafo 3.5.3 per maggiori dettagli.

```python
# Replace each dataset entry with a placeholder with probability `p`
def replace_dataset_entries(self, dataset, placeholder=-200, p=0.2):
    us = np.random.uniform(size=len(dataset))
    return np.array([np.where(u > p, xs, placeholder)
        for (u, xs) in zip(us, dataset)])
```

Listato 5.9: Implementazione Deattivazione selettiva: tutti gli elementi di ciascuna serie temporale vengono rimpiazzati, con propabilità p, dal placeholder fornito in input.

5.3 Rete Neurale

Il modello presentato nel Capitolo 3 è stato implementato e addestrato in Python utilizzando la libreria TensorFlow 2.0 e le API funzionali di Keras. Il Listato 5.11 mostra la definizione della rete neurale, la quale viene costruita attraverso composizioni successive di singoli blocchi convoluzionali. Questi ultimi sono definiti dal Listato 5.10 e rappresentano un grafo di computazione in cui viene applicata all'input l'operazione di convoluzione unidimensionale ed, eventualmente, la funzione di attivazione nonlineare, la normalizzazione tramite Batch Normalization, il Dropout gaussiano e l'operazione di pooling. Si riferisca alla Figura 3.2 per una sua rappresentazione grafica.

```
def conv_block(input, filters=16, window=5, activation='relu',
               normalize=True,
               gauss_dropout=None,
               pool=False,
               decay=1e-4):
  conv = layers.Conv1D(filters, window,
                       input_shape=(None, BEACONS), padding='same',
                       kernel_regularizer=l2(decay))(input)

  if activation is not None:
    conv = Activation(activation)(conv)
  if normalize:
    conv = layers.BatchNormalization()(conv)
  if gauss_dropout is not None:
    conv = GaussianDropout(gauss_dropout)(conv)
  if pool:
    conv = MaxPool1D(pool_size=2)(conv)

  return conv
```

Listato 5.10: Blocco convoluzionale usato per comporre la rete neurale.

```
def build_model(config):
  aux_input = Input(shape=(2, ), name='aux_input')
  main_input = Input(shape=(DOWNSAMPLED_LENGTH, BEACONS), name='main_input')
  compass_input = Input(shape=(1, ), name='compass_input') # only compass
  # Memory input to the network: it is a value between [0, 1]
  memory_input = Input(shape=(1,), name='memory_input')
  ...
  ########################## CONVOLUTIONAL LAYERS ###############################
  # First block
  conv = conv_block(main_input, kernels, window,
                    input_shape=(DOWNSAMPLED_LENGTH, BEACONS), decay=decay,
                    activation=activation)
  # Second block
  conv = conv_block(conv, kernels, window, activation, decay=decay,
  ↪  gauss_dropout=gauss/2, pool=True)
  # Third block
  kernels *= 2
  conv = conv_block(conv, kernels, window, activation, decay=decay,
  ↪  gauss_dropout=gauss, pool=True)
  # Fourth block
  kernels *= 2
  conv = conv_block(conv, kernels, window, activation, decay=decay,
  ↪  gauss_dropout=gauss, pool=True)
  # Series of non pooling convolutional blocks
  for i in range(CNN_layers):
    conv = conv_block(conv, kernels, window, activation, decay=decay)
  # Last CNN block
  kernels *= 2
  conv = conv_block(conv, kernels, window, activation, decay=decay)
  # Global pooling layer
  conv = layers.GlobalAvgPool1D()(conv)
  flattened = layers.Flatten()(conv)
  flattened = layers.Dropout(config['dropout_rate'])(flattened) # Apply dropout to
  ↪  approximate model set averaging
  flattened = layers.concatenate([flattened, compass_input])
  aux_output = Dense(64, activation='relu')(flattened)
  aux_output = Dense(2, activation='linear', name='aux_output')(aux_output)
  # Reshape the memory tensor (which is actually a scalar) to match aux_input shape
  coefficient = layers.RepeatVector(aux_input.shape[1])(memory_input)
  coefficient = layers.Flatten()(memory_input)
  weighted_aux_input = layers.multiply([coefficient, aux_input])
  merged = layers.concatenate([aux_output, weighted_aux_input, memory_input]) #
  ↪  merge beacons and other inputs
  ############################## DENSE LAYERS ###################################
  layer2 = Dense(128, activation='relu',
  ↪  kernel_regularizer=tf.keras.regularizers.l2())(merged)
  output = Dense(2, activation='linear', name='main_output')(layer2)
  return Model(inputs=[main_input, compass_input, aux_input, memory_input],
               outputs=[output, aux_output])
```

Listato 5.11: Costruzione del modello con le API di Keras

Capitolo 6

Conclusioni

La navigazione indoor si è rivelato un problema particolarmente adatto alla risoluzione tramite tecniche di deep learning, anche se queste ultime non si sono mostrate esenti da difetti. In particolare la difficoltà nella raccolta dati per l'addestramento del modello rmiane l'ostacolo più grande in questo contesto e nel machine learning in generale. Tuttavia l'uso di tecniche di data augmentation ha mostrato come anche con una quantità iniziale di dati relativamente scarsa sia possibile generare un modello che stimi la posizione dell'utente all'interno di un edificio con precisione di circa 30cm.

Le fluttuazioni dell'output del modello durante la navigazione sono state parzialmente risolte utilizzando un filtro di Kalman, congiuntamente ai dati del sensore inerziale dello smartphone, ma ulteriori approfondimenti sono necessari allo scopo di valutare concretamente l'entità di questi miglioramenti. Infine, ulteriori approcci al problema della stabilizzazione possono essere esplorati, come discusso nei successivi paragrafi.

6.1 Lavori futuri

Sulla base dei risultati sperimentali ottenuti e sulle difficoltà riscontrate durante la risoluzione del problema della locallizazione indoor, sono state individuate varie idee meritevoli di appronfondimento.

6.1.1 Particle Filter

I Particle Filter sono una categoria di algoritmi Monte Carlo per l'approssimazione di distribuzioni di probabilità non gaussiane. L'utilizzo di questi metodi si rende utile nel momento in cui il modello è soggetto a rumore non gaussiano. È verosimile pensare che l'errore commesso dal modello descritto non presenti una distribuzione gaussiana. In questo contesto, l'utilizzo di un Particle Filter sarebbe da preferire rispetto ad un filtro di Kalman. In particolare per il problema della localizzazione indoor, sono stati descritti vari metodi di tipo Particle Filter che hanno mostrato buoni risultati nella correzione delle previsioni del sistema di navigazione[9, 10].

6.1.2 Input a Lunghezza Variabile

Nella attuale implementazione della rete neurale, le serie temporali dei segnali emessi dai beacon hanno dimensione fissa. Ciò significa che la dimensione del campionamento effettuata durante la fase di navigazione deve essere sempre la stessa, eventualmente aggiungendo valori fittizi alla serie temporale. Tuttavia il modello, grazie al livello di Pooling globale, permette di associare all'input una dimensione qualsiasi lungo l'asse temporale. Sarebbe quindi possibile addestrare il modello con input di lunghezza variabile e a tale scopo non sarebbe difficile arricchire il dataset con variazioni dei segnali originali modificandone la lunghezza. Non è da escludere che un simile approccio migliori ancora la precisione del modello, oltre a renderlo più flessibile in fase di utilizzo, non essendo più legato ad una dimensione costante per l'input dei segnali.

6.1.3 Reti Neurali Residuali

Le reti neurali residuali sono un particolare tipo di architettura, inizialmente ideato per le reti convoluzionali, che permette di ridurre notevolmente il problema della scomparsa del gradiente in modelli molto profondi[11]. L'adozione di reti neurali residuali con un numero più elevato di livelli potrebbe consentire di migliorare la precisione del sistema di navigazione. Tuttavia l'incremento di profondità del modello andrebbe a discapito

della performance in fase di addestramento e in fase di inferenza. Rimane quindi dubbia l'utilizzabilità di tali reti in contesti mobile.

6.1.4 Variational Autoencoder: Generazione di nuovi dati

I Variational Autoencoder (VAE) sono modelli di machine learning generativi. Essi vengono addestrati su un dataset per simulare la generazione di nuovi esempi, simili a quelli visti durante la procedura di addestramento[12]. Utilizzando un VAE è possibile arricchire ulteriormente il dataset di addestramento del modello proposto in questo elaborato, fornendo esempi nuovi e possibilmente migliori rispetto a quelli prodotti implementando manualmente tecniche di data augmentation.

6.1.5 Transfer Learning: Input Masking e Ricostruzione dei Segnali

Un possibile approccio alla risoluzione del problema della raccolta dei dati è quello che prende il nome di Transfer Learning. Addestrando prima un modello su una grande quantità di dati privi di target (il valore cioè che il modello dovrebbe inferire a partire dall'input) su un problema simile all'originale, è possibile sfruttare la configurazione dei parametri risultante per addestrare lo stesso modello su un nuovo problema, spesso più complesso, attraverso un nuovo dataset, stavolta comprendente l'output richiesto. Nel caso della localizzazione indoor, il problema di tipo non supervisionato potrebbe essere quello di ricostruire un segnale perturbato con del rumore, o alterato in qualche modo. Un approccio simile è utilizzato dai modelli di tipo Transformer[13], come BERT[14] e il più recente GPT-3[15], in cui l'input della rete viene mascherato nascondendone una parte e successivamente ricostruito. Se il modello così addestrato risulta essere in grado a ricostruire i segnali, è probabile che abbia individuato delle relazioni tra gli input dei diversi beacon. Ciò dovrebbe aiutare, nel contesto del problema originale, a costruire una rappresentazione significativa dell'input anche con una quantità relativamente piccola di dati e quindi risolvere il problema in modo più efficiente. La raccolta del primo

dataset è effettuabile in modo molto economico navigando l'edificio in cui sono disposti i beacon e raccogliendo i segnali ricevuti.

6.1.6 Simulatore BLE

L'utilizzo di un simulatore di propagazione dei segnali bluetooth all'interno di un edificio potrebbe eliminare completamente la necessità di raccogliere dati manualmente. L'implementazione di un tale simulatore dovrebbe prevedere la possibilità di configurare quest'ultimo impostando una planimetria di un locale e la disposizione dei beacon all'interno di esso. Poichè i segnali bluetooth emessi sono naturalmente soggetti a rumore ambientale, non dovrebbe essere necessario modellare la realtà con un elevato grado di precisione. L'introduzione nel simulatore di sorgenti di rumore casuale dovrebbe essere sufficiente a rappresentare con la giusta fedeltà il contesto del problema e, in questo senso, la resistenza al rumore e alle variazioni dell'input del modello dovrebbero garantire buoni risultati anche con dati sintetici.

6.1.7 Posizionamento Magnetico

La struttura di un edificio e i materiali in esso contenuti alterano il campo magnetico naturalmente emesso dalla Terra. Queste variazioni sono uniche e identificabili, rendendo possibile una tecnica di localizzazione indoor che prende il nome di posizionamento magnetico. Attraverso la mappatura di tutte le perturbazioni del campo magnetico causate dall'edificio stesso, è possibile implementare un sistema di navigazione con un discreto grado di precisione[16]. Il modello presentato in questo elaborato sfrutta già le informazioni ricavate dal sensore magnetico dello smartphone, come descritto nel Paragrafo 3.4.1, ma il loro utilizzo può essere approfondito.

Appendice A

Metriche di Errore

Le metriche utilizzate per valutare l'efficacia del modello sviluppato sono:

- RMSE (Root Mean Squared Error): indica l'errore quadratico medio commesso dal modello su un insiete di dati e sottomesso all'operazione di radice quadrata. Tale metrica pone particolare peso sulla entità dell'errore piuttosto che sul loro numero, per via dell'operazione di elevazione al quadrato. L'adozione della radice permette di valutare la metriche nell'unità di misura di riferimento; in questo caso il metro, poichè le coordinate dell'output del modello sono espresse in metri. Un basso valore di MSE può quindi comportare un alto numero di errori commessi a fronte di una alta precisione.

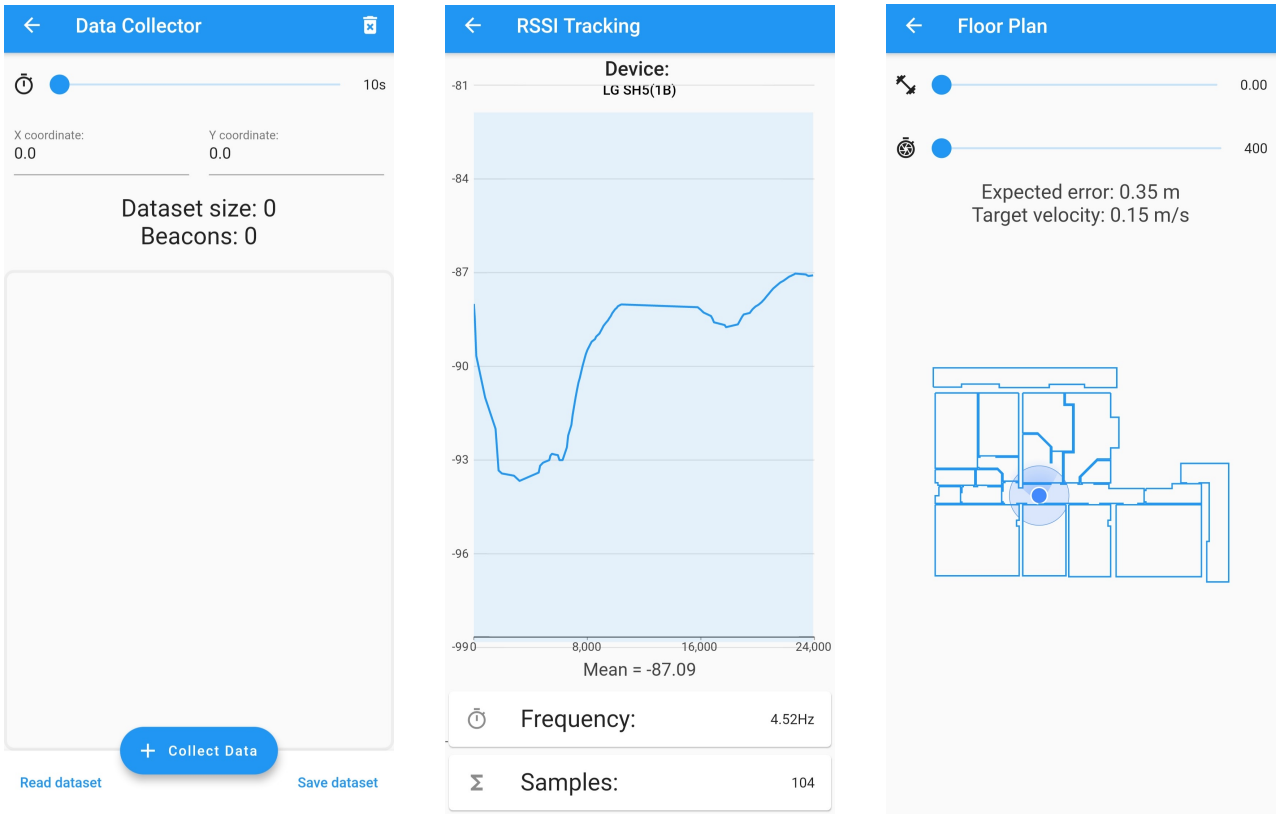
- MAE (Mean Absolute Error): indica l'errore assoluto medio commesso dal modello ed è utilizzato per valutare l'efficacia del modello ponendo meno attenzione sulla presenza di eventuali valori anomali. Come nel caso di RMSE, anche questa metrica esprime una valutazione della precisione del modello in metri.

- MaxAE (Max Absolute Error): determina il massimo errore assoluto commesso dal modello ed è utile per ottenere una stima superiore della precisione del modello. Un alto valore di MaxAE correlato con un basso valore di MAE indica una buona precisione media, ma con errori significativi e sporadici. Questa metrica,

poichè non standardizzata, non è presente tra quelle di TensorFlow ed è stato necessario implementarla.

Appendice B

Interfaccia Applicazione



(a) Acquisizione dei segnali **(b)** Tracking dei segnali RSSI **(c)** Visualizzatore Planimetrie

Figura B.1: Screenshot delle varie schede dell'applicazione mobile.

In Figura B.1a è mostrata la scheda relativa alla raccolta dei dati. Essa permette di indicare le coordinate del punto da cui si intende raccogliere i segnali Bluetooth e la lunghezza del periodo di tempo del campionamento. L'interfaccia permette poi di salvare i dati nella memoria interna del dispositivo in formato CSV, dopo essere stati processati. La frequenza di campionamento è fissa e pari alla massima frequenza disponibile per la ricerca di dispositivi Bluetooth in Android.

In Figura B.1b è illustrata la componente grafica dell'applicazione sviluppata per lo studio dei segnali Bluetooth; Il suo utilizzo è stato per lo più prototipale e inizialmente utile a valutare l'andamento dei segnali emessi dai dispositivi Bluetooth in ambienti chiusi. Il grafico mostra l'andamento in tempo reale della media dei valori RSSI raccolti dall'applicazione per uno specifico dispositivo Bluetooth. In basso sono indicati il numero di campionamento la frequenza con cui questi vengono effettuati.

La Figura B.1c mostra il visualizzatore di planimetrie: in alto sono presenti due slider che gestiscono rispettivamente il coefficiente di memoria residua del modello (Paragrafo 3.4.1) e la frequenza di campionamento dei segnali. Il visualizzatore è stato sviluppato senza utilizzare asset predefiniti e la grafica del cursore è di tipo vettoriale.

Bibliografia

[1] Ministero della Salute. *Immuni*. 2020. URL: https://github.com/immuni-app/immuni-documentation.

[2] Apple e Google. *Privacy-Preserving Contact Tracing*. 2020. URL: https://www.google.com/covid19/exposurenotifications/.

[3] Ian Goodfellow, Yoshua Bengio e Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[4] Allan Pinkus Moshe Leshno Vladimir Ya. Lin e Shimon Schocken. "Multilayer Feedforward Networks With a Nonpolynomial Activation Function Can Approximate Any Function". In: *Neural Networks* (1993).

[5] Ben Poole, Jascha Sohl-Dickstein e Surya Ganguli. "Analyzing noise in autoencoders and deep networks". In: *CoRR* abs/1406.1831 (2014). arXiv: 1406.1831. URL: http://arxiv.org/abs/1406.1831.

[6] Christopher Bishop. "Regularization and Complexity Control in Feed-forward Networks". In: *Proceedings International Conference on Artificial Neural Networks ICANN'95*. Vol. 1. EC2 et Cie, gen. 1995, pp. 141–148. URL: https://www.microsoft.com/en-us/research/publication/regularization-and-complexity-control-in-feed-forward-networks/.

[7] Google. *Flutter SDK*. URL: https://flutter.dev/.

[8] Google. *TensorFlow Lite*. URL: https://www.tensorflow.org/lite.

[9] J. M. Pak et al. “Improving Reliability of Particle Filter-Based Localization in Wireless Sensor Networks via Hybrid Particle/FIR Filtering”. In: *IEEE Transactions on Industrial Informatics* 11.5 (2015), pp. 1089–1098.

[10] P. Yang e W. Wu. “Efficient Particle Filter Localization Algorithm in Dense Passive RFID Tag Environment”. In: *IEEE Transactions on Industrial Electronics* 61.10 (2014), pp. 5641–5651.

[11] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[12] Diederik P. Kingma e Max Welling. “An Introduction to Variational Autoencoders”. In: *CoRR* abs/1906.02691 (2019). arXiv: 1906.02691. URL: http://arxiv.org/abs/1906.02691.

[13] Ashish Vaswani et al. “Attention Is All You Need”. In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[14] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *CoRR* abs/1810.04805 (2018). arXiv: 1810.04805. URL: http://arxiv.org/abs/1810.04805.

[15] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL].

[16] W. Storms, J. Shockley e J. Raquet. “Magnetic field navigation in an indoor environment”. In: *2010 Ubiquitous Positioning Indoor Navigation and Location Based Service*. 2010, pp. 1–10.