UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INFORMATICA

Tesi di Laurea Magistrale in Informatica (LM-18)

Autoencoder quantistico per rilevare attacchi alla rete elettrica

Relatore

Prof. Esposito Christiancarmine

Candidato

Gerardo Sessa

Matr. 05225 01583

ANNO ACCADEMICO 2024-2025

Abstract

Con l'introduzione di dispositivi intelligenti e tecnologie di controllo automatizzato nelle reti elettriche, cresce la superficie di attacco esposta a potenziali minacce informatiche. In particolare, in ambienti interconnessi basati su *Internet of Things (IoT)*, diventa fondamentale distinguere eventi di guasto reali da attacchi malevoli che potrebbero compromettere la stabilità dell'intero sistema. In questo contesto, l'impiego di tecniche di **intelligenza artificiale** e **machine learning** rappresenta una soluzione efficace per il rilevamento automatico di anomalie e tentativi di intrusione.

Questa tesi si propone di trasportare una pipeline classica di classificazione per il riconoscimento di attacchi informatici con una rete neurale, già presente in letteratura, in un ambiente quantistico, progettando un'architettura di **Quantum Autoencoder (QAE)** in pila in maniera che risulti come il diretto modello equivalente per il quantum computing.

Dopo aver studiato le basi della computazione quantistica e delle reti neurali, è stata implementata un'architettura in grado di comprimere l'input e apprendere una rappresentazione significativa, sfruttando un numero limitato di qubit. Il modello è stato testato su dati rappresentativi di eventi critici nella rete elettrica e addestrato con diversi set di iperparametri ottimizzati.

Il lavoro apre interessanti prospettive future: tra queste, la sperimentazione di nuovi circuiti per encoder e decoder, finalizzati a migliorare l'efficienza e la scalabilità del Quantum Autoencoder in scenari reali.

Table of Contents

Introduzione

Le principali tecnologie di automazione utilizzate da sistemi di distribuzione di elettricità, acqua e gas, vanno a sfruttare per sistemi di controllo e monitoraggio *infrastrutture informatiche interconnesse*, fornendo capacità avanzate di controllo dell'automazione e interoperabilità. L'implementazione di questo ulteriore livello nella logica di queste strutture però ha un aspetto negativo, ovvero l'apertura di una potenziale superfice d'attacco esposta, aspetto problematico in ambienti che vanno ad usufruire dei servizi di **Internet of Things (IoT)** di nodi e sensori interconnessi che permettono la lettura e l'attraversamento di dati sensibili cruciali, diventando bersaglio di attacchi informatici.

Per le *reti elettriche* sono presenti dispositivi di controllo e misurazione intelligenti, come i registratori di guasti digitali, che forniscono dati sullo stato e sulle operazioni del sistema complessivo, supportando le decisioni di automazione e il monitoraggio dei guasti.

Eventi come guasti delle linee di trasmissione, possono essere si spontanei, ma c'è la possibilità che dispositivi di controllo siano compromessi per inviare comandi di controllo fraudolenti e impersonare disturbi, con cui il malintenzionato avvierà una catena di reazioni che porta a blackout o guasti critici, se le reazioni appropriate non vengono eseguite in modo tempestivo.

Distinguere attacchi da guasti reali non è un compito facile, per questo viene introdotto per il riconoscimento tecnologie di **intelligenza artificiale (IA)** e **apprendimento automatico (ML)** per costituire un sistema che permetta di correlare in maniera corretta eventi e osservazioni, per attivare automaticamente le azioni appropriate quando viene scoperto un attacco.

Il tema della tesi è rilevante anche in vista di recenti avvenimenti: durante la stesura di questo elabortato, nello specifico a fine aprile 2025, un'improvvisa instabilità nella rete elettrica ha coinvolto diverse aree di **Spagna** e **Portogallo**, causando blackout diffusi e quindi il blocco temporaneo di servizi essenziali, tra cui i trasporti pubblici e i sistemi digitali di comunicazione. L'evento ha portato le indagini delle autorità a verificare se si sia trattato di un possibile attacco informatico.[1]

Questo episodio, di cui ad oggi non sono state date conferme di un'azione dolosa, ha messo in luce un punto cruciale: la difficoltà nel discriminare rapidamente e con precisione un guasto tecnico da un attacco mirato può compromettere l'efficacia della risposta del sistema, ritardando l'attivazione di contromisure appropriate.

Di fronte a reti sempre più automatizzate e interconnesse, la capacità di riconoscere tempestivamente le anomalie diventa un'esigenza prioritaria. In questo scenario, approcci basati su intelligenza artificiale offrono prospettive innovative per affrontare il problema, permettendo di costruire modelli capaci di rilevare pattern complessi e supportare il processo decisionale anche in situazioni ad alto rischio e incertezza.

L'obiettivo della tesi è quello di osservare tipologie di autoencoder già esistenti che permettono il riconoscimento di un tentativo di attacco nella rete elettrica e realizzare un'implementazione quantistica, in modo da poterne poi confrontare i risultati e stabilire quale delle due soluzioni sia la più efficace.

Il documento è strutturato in quattro capitoli: nel primo capitolo vengono descritte

le basi fondamentali per comprendere gli aspetti di **Deep Neural Network** che riguarderanno l'architettura da studiare e successivamente sviluppare; il secondo capitolo riassume i fondamentali del lavoro presente in letteratura per l'**autoencoder** per rivelare gli attacchi alla rete elettrica, che verrà utilizzato come punto di partenza per il lavoro di questa tesi; il terzo capitolo è stato dedicato alla spiegazione degli essenziali della **computazione quantistica**; il quinto capitolo rappresenta l'analisi dettagliata dello sviluppo del **modello quantistico** utilizzato per rilevare attacchi alla rete elettrica; nel quinto capitolo sarannò descritte le conclusioni e gli sviluppi futuri.

1 Deep Neural Network

1.1 Introduzione alla Deep Neural Network

Le **Deep Neural Networks** (*DNN*, reti neurali profonde) sono modelli che hanno capacità di apprendere in modo autonomo. Le DNN sono ispirate alla struttura del *sistema visivo* dei mammiferi: elabora le informazioni dalla retina al centro visivo strato dopo strato, estraendo in sequenza i bordi, poi le parti, le forme e infine formando concetti astratti. Le DNN sono formate da diversi **livelli** di nodi (neuroni artificiali):

- *Livello di input*: ogni nodo del livello di entrata rappresenta una caratteristica univoca dei dati che vanno ad entrare nella rete;
- *Livelli nascosti*: si occupano dell'elaborazione dei dati tramite una struttura basata su **pesi** e **funzioni**. Le DNN possono avere un numero diverso di livelli nascosti a seconda della complessità dei compiti da assolvere;
- *Livello di output*: è il livello che restituisce il risultato del lavoro svolto dalla DNN, una *previsione*.

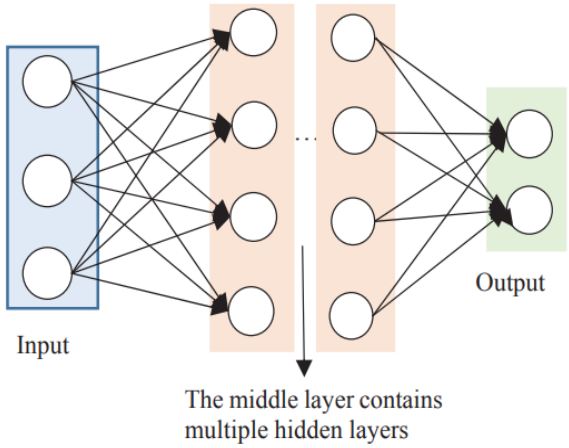

Figura 1.1: Struttura del DNN

Una *rete neurale* tradizionale prende decisioni basate sui dati in ingresso, ma è limitata a ciò che il programmatore ha fornito esplicitamente come caratteristiche. Una **rete neurale profonda** supera questo concetto, imparando dalle esperienze precedenti, adattando le sue decisioni in base agli schemi che riconosce in grandi insiemi di dati.

L'utilizzo di **pesi** per i neuroni permette il riconoscimento di una classe tramite associazione per valore. I pesi vengono scelti inizialmente casualmente, e solo durante l'allenamento, man mano che la rete sbaglia, corregge i valori per fare in modo da migliorare le previsioni al tentativo successivo (questo processo è definito *backpropagation*).

Un'altra osservazione da fare per poter lavorare con una rete neurale è che il valore associato al peso non deve essere troppo grande, altrimenti rischia di andare in **overfitting**, ovvero di riuscire ad associare abilmente i valori all'input della rete sui dati utilizzati per l'addestramento, ma così facendo potrebbe rischiare di fallire su dati inediti.

L'*overfitting* si verifica quando la rete diventa troppo complessa e inizia a memo-

rizzare i dati di addestramento, anziché imparare a generalizzare nuovi dati. Per evitare ciò si usa la regolarizzazione, una tecnica che penalizza i pesi troppo grandi, aiutando la rete a generalizzare in maniera migliore.[2]

1.2 Tipologie di DNN

Esistono diverse architetture di deep neural networks. Tra i modelli di DNN più utilizzati si hanno le Reti neurali convoluzionali, le reti ricorrenti e gli autoencoder.

1.2.1 CNN

Le **reti neurali convoluzionali** (*CNN*) vengono utilizzate particolarmente nell'ambito della **computer vision**, quindi per elaborazione di immagini e video. Sono progettate per elaborare i dati presenti nell'immagine mediante **struttura a griglia**. Grazie agli strati convulazionali, sono in grado di estrarre caratteristiche locali e svolgere attività come il *rilevamento di oggetti*, il *riconoscimento di immagini* e il *riconoscimento facciale*. Almeno tre livelli costituiscono una CNN:

1. **Livello convolutivo**: Scansiona l'immagine con dei filtri per trovare caratteristiche locali, come *bordi*, *curve*, *colori*.
2. **Livello di *pooling***: Riduce la dimensione dei dati per semplificare l'elaborazione, mantenendo solo le informazioni più importanti. Quando ci si riferisce al *max-pooling*, si tratta del valore più alto in una zona.
3. **Livello completamente connesso**: collega le caratteristiche collegate precedentemente ad un classificatore.

Ad ogni livello la complessità della CNN aumenta. I primi livelli si concentrano su funzioni semplici, come i colori e le forme. Mentre i dati dell'immagine avanzano attraverso i livelli della CNN, vengono riconosciuti elementi o forme più grandi fino a quando non si riesce ad identificare l'oggetto.

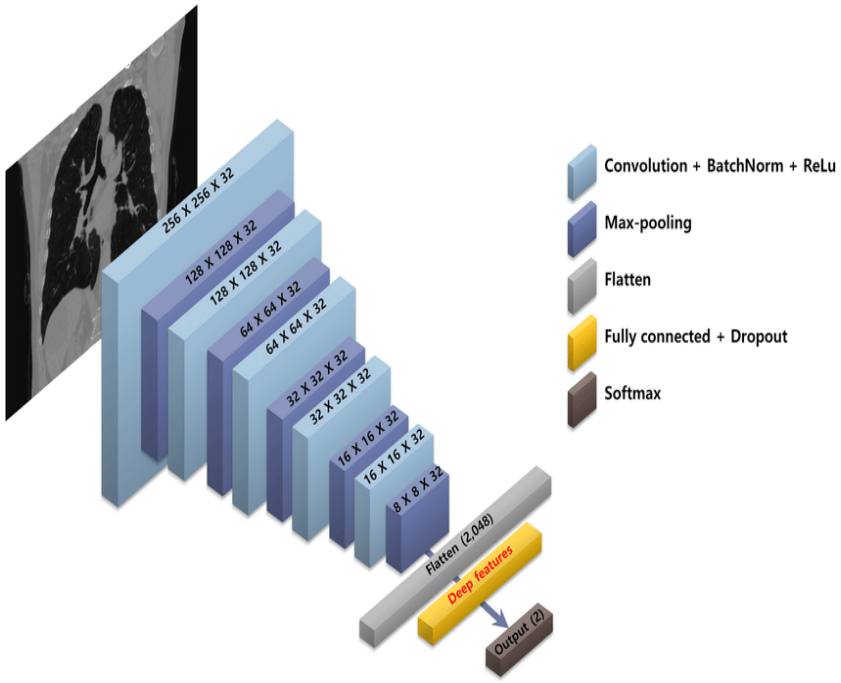

Figura 1.2: Struttura CNN

1.2.2 RNN

Le **reti neurali ricorrenti** (*RNN*) vengono utilizzate per elaborare dati sequenziali come testo e audio, data la capacità di memorizzare informazioni nel tempo.
Le reti neurali ricorrenti sono caratterizzate dai *cicli di feedback*, che consentono loro di mantenere una memoria interna degli input passati per migliorare le elaborazioni successive, attraverso un contesto influenzato dai dati di serie temporali, cioè si ha dipendenza con gli input e output precedenti, laddove in un'altra tipologia di rete

neurale sarebbero state indipendenti l'une dall'altre.

RNN utilizza una variante della backpropagation chiamata retropropagazione nel tempo (BPTT), che è diversa dato che somma gli errori in ogni fase temporale, mentre la retropropagazione standard non ha bisogno di sommare gli errori, non condividendo i parametri in ogni livello.

Viene utilizzato per la traduzione del linguaggio e il riconoscimento vocale, come con **Siri** e **Google Translate**.

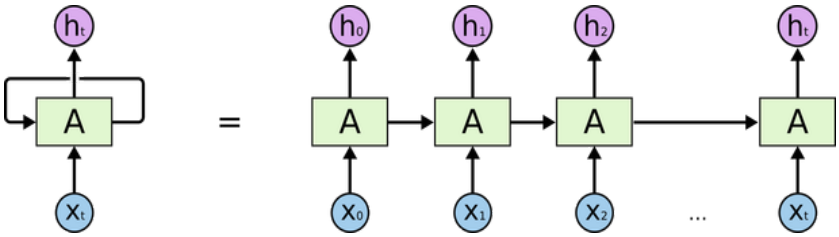

Figura 1.3: Struttura RNN

1.2.3 Autoencoder

Gli **autoencoder** sono un tipo di rete neurale non supervisionata, termine con cui si intende che durante l'allenamento della rete neurale non vengono forniti dei dati etichettati come risposte corrette, utilizzata per l'apprendimento di rappresentazioni compresse dei dati di input.

Sono storicamente stati tra i primi modelli di deep learning utilizzati con successo per quella che consideriamo ad oggi l'**AI generativa**, quindi per la generazione di immagini e suoni, questo per la loro abilità di «comprendere» da sè la struttura interna di un insieme di dati e poi crearli da zero. Gli autoencoder lavorano codificando i dati non etichettati in una rappresentazione compressa e di seguito decodificando i dati nella loro forma originale.

L'architettura è composta da blocchi di **encoder**, che con una codifica mappano i dati di input in una rappresentazione di dimensioni ridotte, e in blocchi di **decoder**, che tramite decodifica ricostruiscono i dati originali dalla rappresentazione compressa. L'abilità principale degli autoencoder è la capacità di gestire grandi quantità di dati comprimendoli, e dall'input in forma compressa riuscire a rilevare quelli che sono le caratteristiche salienti. [3]

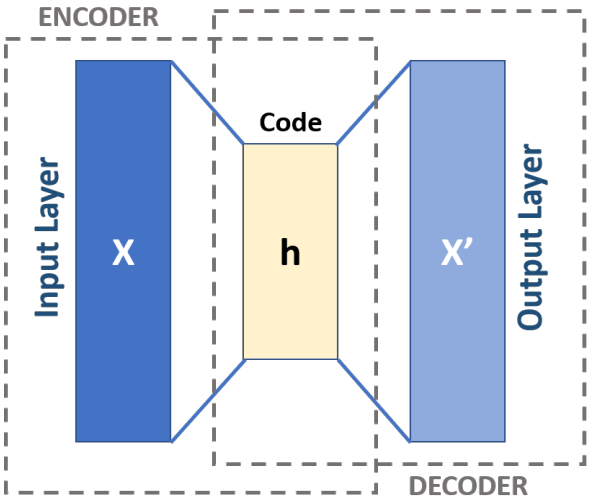

Figura 1.4: Struttura di un autoencoder

Esistono diverse applicazioni per utilizzare un autoencoder, come il *Denoising Autoencoder (DAE)*, che aggiunge rumore all'input e impara a ricostruire quello originale; *Sparse Autoencoder (SAE)*, che sfrutta un malus per forzare attivazioni più "sparse" di un minor numero di neuroni per favorire l'apprendimento di caratteristiche rilevanti; *Variational Autoencoder (VAE)*, con una struttura probabilistica per generare nuovi dati simili all'input per favorire l'utilizzo in ambito generativo; e Stacked Autoencoder, un tipo di deep autoencoder in cui più autoencoder sono posizionati in pila per formare una rete profonda.

Sparse Autoencoder

Gli **autoencoder sparsi** (*SAE*) sono una tipologia di reti neurali che impongono un **vincolo di scarsità**, riducendo il numero di nodi che possono essere attivati contemporaneamente andando a dare una penalità superata una certa soglia.
Questa forzatura, oltre a permettere di evitare overfitting, da la possibilità di attivare nodi nella rete solo se sono essenziali per rilevare una caratteristica fondamentale, in maniera da riuscire ad apprendere efficientemente specifiche immagini entro i vincoli della scarsità.
Inoltre, dare una soglia permette di rendere la rete più scalabile con un maggior numero di nodi, rimanendo sempre nella logica che l'attivazione di questi si ha sempre per una certa taglia ammessa.

Stacked Autoencoder

Uno **Stacked Autoencoder**, costruisce rappresentazioni gerarchiche dei dati, combinando più livelli di autoencoder, in modo che ogni livello dell'autoencoder lavori sull'output del livello precedente, costruendo così una catena di trasformazioni che consente di estrarre progressivamente caratteristiche sempre più astratte, facilitando il riconoscimento di pattern complessi nei dati. L'addestramento di uno stacked autoencoder[4] avviene in due fasi principali: **pre-training** e **fine-tuning**.

Pre-Training

1. **Suddivisione in autoencoder singoli**: il SAE viene scomposto in più autoencoder, ciascuno costituito da due livelli adiacenti (input e codifica).
2. **Addestramento del primo AE**: si parte dal primo autoencoder in basso. Questo viene addestrato per minimizzare l'errore di ricostruzione, cioè la differenza tra l'input originale e quello ricostruito dalla rete.
3. **Catena di addestramento**: l'output del primo AE diventa l'input per il secondo AE, che viene a sua volta addestrato. Il procedimento continua verso l'alto, strato dopo strato, considerando in questa prima parte l'addestramento come individuale per ogni AE creata.
4. **Salvataggio dei pesi**: una volta addestrati tutti gli autoencoder, si salvano i pesi appresi in una matrice denominata $\boldsymbol{W}$, che verrà usata nella fase successiva.

Fine-Tuning

1. **Costruzione del SAE completo:** si ricompone l'intera rete utilizzando i pesi W ottenuti nella fase precedente.
2. **Ottimizzazione supervisionata:** tramite l'algoritmo di backpropagation, si correggono i pesi per ridurre ulteriormente l'errore di ricostruzione. Il risultato è una nuova matrice di pesi ottimizzata, indicata come $W + \varepsilon$.

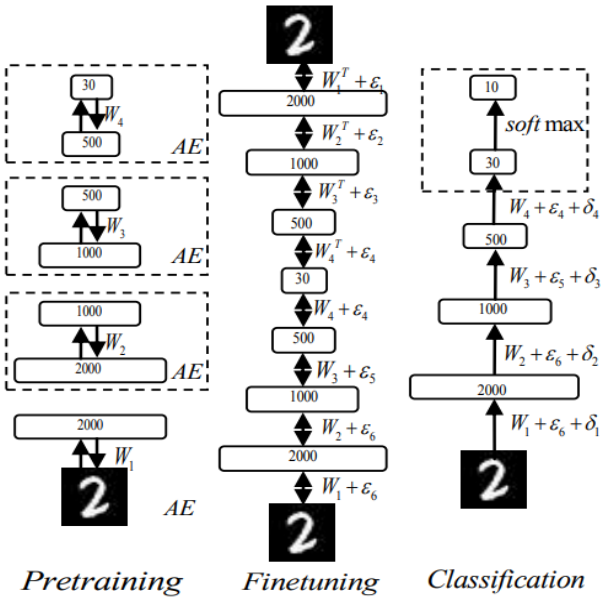

Figura 1.5: Addestramento SAE

1.3 Conclusioni

In questo capitolo sono state descritte le principali tipologie di reti neurali profonde per avere una conoscenza dell'ambito dell'elaborato, con maggiore attenzione e approfondimento dedicato agli autoencoders, che saranno l'argomento principale del nostro caso di studio.

Nel prossimo capitolo verrà descritto ed analizzato uno stacked autoencoder per rilevare attacchi informatici nei sistemi interconnessi di controllo dell'alimentazione.

2 Autoencoder per rilevare attacchi alla rete elettrica

In questo capitolo verrà descritto uno stacked autoencoder per rilevare attacchi informatici sulla rete elettrica, in grado di distinguere problematiche spontanee rispetto a quelle indotte da un malintenzionato.
Il testo di riferimento è «**A stacked autoencoder-based convolutional and recurrent deep neural network for detecting cyberattacks in interconnected power control systems**»[5], in cui viene presentato un framework di deep learning per rilevare attacchi informatici nei sistemi di controllo dell'energia elettrica.

2.1 Framework

Il modello sfrutta una combinazione di due **autoencoder sparsi** (SAE), potenziati rispettivamente con una **rete neurale convoluzionale** (CNN) e una **rete neurale ricorrente** (RNN), in grado di sfruttare sia **dipendenze spaziali** (tra valori di diversi sensori e misurazioni) sia **dipendenze temporali** (eventi che si verificano nel tempo delle osservazioni).

L'architettura consente di estrarre automaticamente caratteristiche significative dai dati di monitoraggio, senza aver bisogno di una conoscenza preliminare del dominio o dei dispositivi specifici. Questo lo rende riutilizzabile in ogni **sistema di controllo industriale**, purché si abbia a disposizione un numero sufficiente di osservazioni nel tempo.

2.2 Rilevamento delle anomalie

L'architettura si basa su una **RNN** in grado di estrarre le principali caratteristiche dai dati di misurazione dei **sistemi di controllo industriale (ICS)** per distinguere comportamenti normali da quelli anomali, e trovare in questi dati raccolti correlazioni spaziali e temporali.
Lo scopo delle dipendenze spaziali è scoprire informazioni che mettano in relazione le variabili di stato da analizzare. Lo scopo delle dipendenze temporali invece è di apprendere dipendenze che possono esistere nel tempo.
La scelta di combinarlo con un modello di tipo **CNN** è data dall'utilizzo di diversi livelli convoluzionali per la rappresentazione astratta dei dati di input per l'analisi delle dipendenze spaziali e temporali.
I due *autoencoder sparsi* che vengono utilizzati sono:

- **CNN-SAE**: La funzione di codifica e di decodifica del primo SAE racchiude una CNN addestrata sui valori delle variabili di stato disposti in una matrice, replicando il formato che avrebbe un'immagine da analizzare.
- **RNN-SAE**: Le funzioni di codifica e decodifica utilizzano una RNN addestrata sui dati provenienti dallo spazio latente del primo SAE.

La coppia di SAE viene inserita in un'achitettura a pila per formare un'unica rete

neurale profonda. L'output ottenuto viene inviato a una **rete di classificazione** *Softmax* che eseguirà il rilevamento dell'attacco.

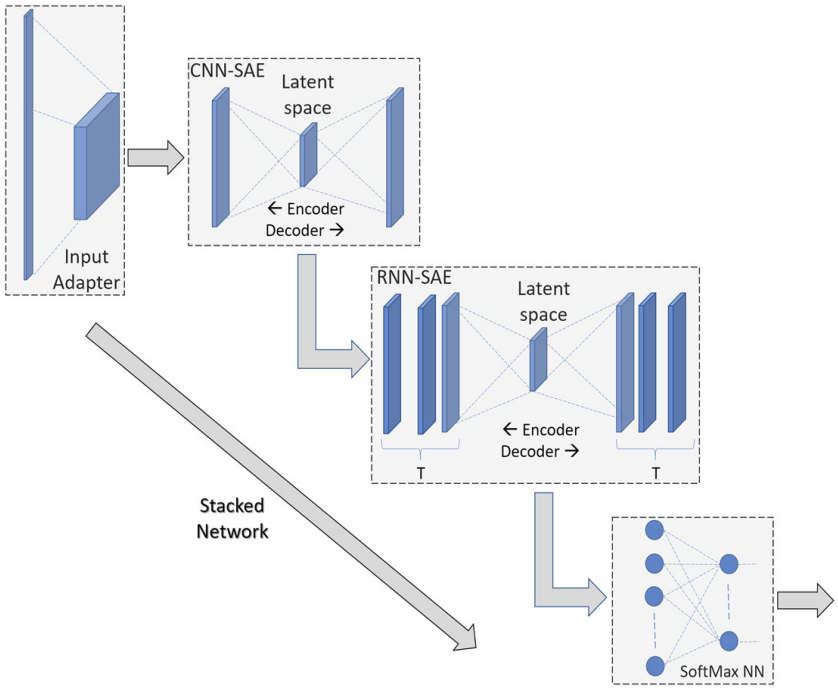

Figura 2.1: Struttura Stacked Autoencoder

2.3 Conclusioni

Presentata la struttura principale del autoencoder di questo caso di studio, sarà nelle fasi successive presentato il quantum computing, con l'obiettivo di comprenderne la tecnologia per poi traslare in questo ambiente la struttura e le funzionalità dello stacked autoencoder di riferimento.

3 Computazione quantistica

In questo capitolo verranno descritte le basi del **quantum computing**, tecnologia basata sulla meccanica quantistica che verrà utilizzata per lo sviluppo dell'autoencoder che verrà presentato per come verrà sviluppato nel prossimo capitolo.

3.1 Quantum computing

Il **calcolo quantistico** è un tipo di computazione che utilizza i principi della **meccanica quantistica** per elaborare informazioni. È una tecnologia caratterizzata da un'elevata potenza di calcolo, col particolare utilizzo dei **qubit**, ovvero i *bit quantistici*: I computer tradizionali utilizzano i bit per la rappresentazione delle informazioni, che possono assumere valore di zero o uno, nel tipico sistema binario. I qubit invece possono assumere valore zero, uno, e sopratutto *zero e uno contemporaneamente*, peculiarità che rende le elaborazioni in quantistico più rapide, tramite l'utilizzo di algoritmi quantistici multidimensionali. Questa combinazione che il qubit può adoperare si chiama **sovrapposizione**, e il concetto si basa sul paradosso in fisica quantistica del *gatto di Schrödinger*: come nel celebre esperimento non sappiamo se il gatto è vivo o morto finché non apriamo la scatola, allo stesso modo finché non misurato il qubit assume in maniera sovrapposta lo stato di 0 e di 1.

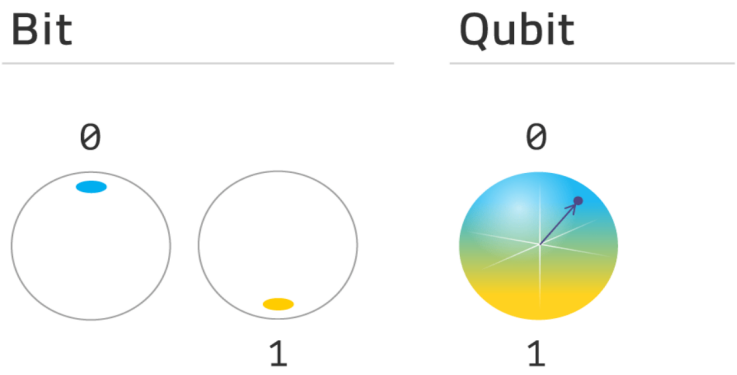

Figura 3.1: Rappresentazione grafica di un Qubit

Un altro fenomeno di fisica quantistica alla base del funzionamento dei qubit è l'**entanglement**: come nel legame tra due elementi come le particelle (mantenuto anche a distanza), anche i qubit avranno una dipendenza diretta di stati l'uno con l'altro. E' l'entanglement quantistico a cuore della potenza e la velocità d'esecuzione peculiare del quantum computing, dato che questo permette di avere da un risultato anche tutti gli altri valori di qubit «collegati» in maniera istantanea.

3.1.1 Circuiti quantistici

In un tipico *circuito booleano*, si ha una composizione di **fili** e **porte logiche** (gates), con i fili che vanno a trasportare informazioni (valori binari) e le porte per le operazioni booleane, rappresentate in forme grafiche convenzionali per l'elettronica. Il *circuito aciclico* da sinistra verso destra ha di partenza fili di input per i bit 0/1.

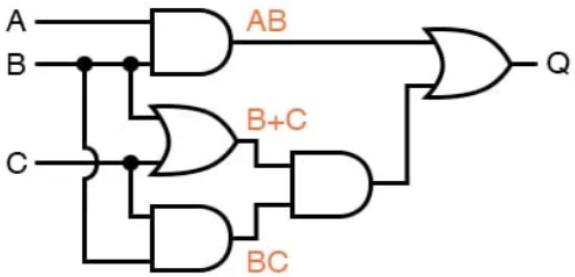

Figura 3.2: Esempio di circuito booleano

In relazione a questa tipologia, in un **circuito quantistico**, il ruolo che hanno i fili nel trasporto delle informazioni lo hanno i qubit, mentre i gates sono l'equivalente degli operatori unitari e delle misurazioni. La rappresentazione aciclica da sinistra a destra viene qui intesa come la scansione temporale di operazioni in sequenza, ognuna descritta da una matrice di riferimento differente. Alla fine del circuito si otterrà come risultato una matrice finale. [6]

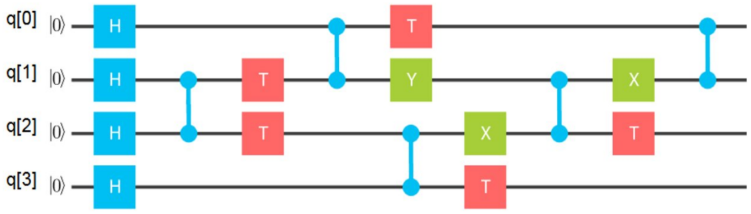

Figura 3.3: Esempio di circuito quantistico 4x1

Le **matrici** hanno un ruolo fondamentale nel calcolo quantistico: lo stato di un qubit è rappresentato da un *vettore colonna* (matrice 2×1), mentre le operazioni dei gates sono *matrici* 2x2, quindi sul circuito ogni porta logica rappresenterà una moltiplicazione tra il vettore e la matrice dell'operatore.
Le matrici più utilizzate per gli operatori dei gates sono:

- **I**, *identità*: non cambia il qubit;
- **X**, *Pauli-X*: inverte;
- **H**, *Hardman*: stabilisce la sovrapposizione;
- **Z**, *Pauli-Z*: cambia il segno.

Nella figura 3.3 sono presenti anche i **controlled gate**, utilizzati per introdurre una logica condizionale tra i qubit. Applica un'operazione a un qubit solo se un altro qubit ha verificato quella condizione, e questo permette di introdurre la funzionalità dell'entanglement all'interno del circuito. Delle tipologie di controlled gate un esempio può essere il *cnot* (Controlled-NOT), che permette l'inversione del target.

3.1.2 Reti neurali quantistiche

Come definito nei capitoli precedenti, le **reti neurali** classiche sono ispirati alla struttura del cervello umano, con i **nodi** a rappresentare i neuroni, strutturati a strati.
Quello che si va a fare con il *machine learning quantistico* è integrare le proprietà dell'informatica quantistica alle reti neurali classiche. Le **QNN** (*Quantistic Neural Networks*) combinano quindi i circuiti quantistici alle reti neurali tipiche.
Con l'apprendimento automatico, le QNN si pongono ancora l'obiettivo di trovare pattern nascosti nei dati somministrati, caricati dall'input per essere elaborati in

stato quantistico dai *gates*, che per questo utilizzo vengono paratremizzati da *pesi* addestrabili.

L'utilizzo del QNN per il machine learning quindi è per caricamento, elaborazione e misutazione, il cui output viene utilizzato per una *funzione di perdita* che addestra i pesi tramite *backpropagation*.

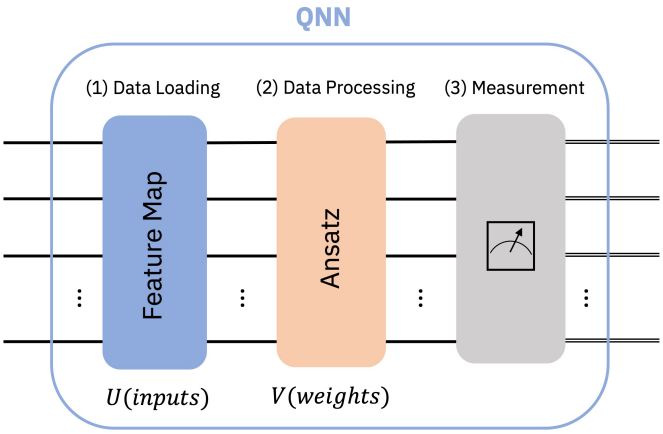

Figura 3.4: Struttura QNN

L'*output quantistico* che viene restituito è un valore classico (dato che alla misurazione non si ha più sovrapposizione), quindi 0, 1 o una probabilità. La **funzione di perdita** confronta l'output ottenuto rispetto a quello atteso e dalla diffferenza dei due si aggiornano i parametri del circuito per retropropagazione.

3.2 Conclusioni

Sono stati descritti tutti gli elementi della computazione quantistica necessari a sviluppare il sistema proposto da questo elaborato.

Nel prossimo capitolo si andrà a descrivere l'adattamento dell'autoencoder soggetto di studio in una rete neurale quantistica.

4 Sviluppo dell'autoencoder quantistico

In questo capitolo sarà descritto il sistema sviluppato, partendo dalla presentazione di quali dataset, software e librerie sono stati utilizzati.
A seguire, lato software andremo ad esaminare il sistema adottato per la raccolta e la visualizzazione dei dati. Infine, nel sottocapitolo del sistema proposto verrà presentato il sistema nel suo insieme, presentando il codice sviluppato per la trasposizione della pipeline studiata in precedenza nell'ambito delle reti neurali quantistiche.

4.1 Dataset

Il dataset di cui si è andato ad usufruire è *Power System*[7], reso disponibile in rete da *Bachir Barika* sul sito di *Kraggle*. Questo dataset contiene misurazioni di sistemi elettrici da utilizzare per analisi di **guasti** ed **anomalie**. Gli scenari presentati sono 37, suddivisi in 8 **eventi naturali**, 1 **evento nullo** e 28 **eventi d'attacco**.
L'ambiente del dataset è una rete elettrica[8] composta da **generatori** (G1 e G2), **interruttori** (da BR1 a BR4) e **dispositivi elettronici intelligenti** (IED, da R1 a R4). Le linee elettriche sono due e collegano gli interruttori BR1-BR2 e BR3-BR4.

Ogni IED controlla un interruttore e utilizza un sistema di protezione che attiva l'interruttore in caso di guasto, senza distinguere tra guasti reali o falsi, poiché non sono in grado di riconoscerne la differenza. Gli operatori possono azionare manualmente gli interruttori tramite gli IED per scopi di manutenzione.

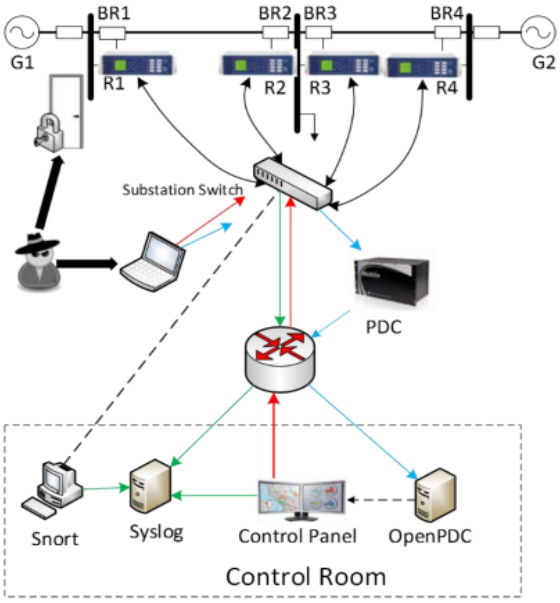

Figura 4.1: Rete elettrica degli scenari del dataset

4.2 Pennylane AI

PennyLane AI è un framework open-source sviluppato da *Xanadu* per la programmazione, simulazione e addestramento di algoritmi di machine learning quantistico. Permette di integrare **circuiti quantistici** direttamente nel workflow dei modelli di machine learning classico.

PennyLane consente di creare circuiti quantistici con parametri automaticamente calcolati usando la backpropagation.

La libreria Pennylane è resa disponibile per l'installazione tramite il terminale di **Python**.

4.3 NumPy

Libreria Python per il calcolo scientifico, offre array multidimensionali ad alte prestazioni e un vasto insieme di funzioni matematiche per l'elaborazione numerica. Può essere utilizzata come base per *PyTorch*, creando gli array che poi verranno resi tensori.

4.4 PyTorch

Torch è una libreria di machine learning sviluppata da *Facebook*, utilizzata per costruire e addestrare reti neurali. Fornisce strutture dati chiamate tensori, simili agli array di **NumPy** ma con la possibilità di sfruttare la GPU per l'elaborazione, e supporta il calcolo automatico del **gradiente**, fondamentale per il training dei modelli.

4.5 Matplotlib

Libreria per la creazione di grafici e visualizzazioni in Python, utilizzata per rappresentare dati in forma di grafici a linee, istogrammi, ecc., ed è particolarmente utile per visualizzare l'andamento dell'addestramento dei modelli (ad esempio, accuracy e loss nel tempo).

4.6 Sistema proposto

Il sistema sviluppato si articola in una pipeline strutturata in più fasi, ognuna con un ruolo specifico nel processo di apprendimento e classificazione, con l'obiettivo di rispettare in ambito quantistico la struttura per come proposta nella documentazione di riferimento presentata nei capitoli precedenti e confrontarne i risultati. La pipeline adattata sussiste, nell'ordine, in:

- **Preprocessing**;
- Un **QAE** (quantum autoencoder) con un encoder all'entrata e un decoder in uscita per i dati sottoposti al *QCNN* (Rete neurale convoluzionale quantistica) interno;
- Un successivo **QAE** con un encoder all'entrata e un decoder in uscita per i dati sottoposti al *QRNN* (Rete neurale ricorrente quantistica) interno;
- **Classificatore**, per l'elaborazione finale e l'output dei valori confrontati, completando il processo di apprendimento supervisionato.

In fase sperimentale, è stato condotto un **fine tuning** dei principali iperparametri, prendendo in considerazione:

- il **learning rate** (`lr`),
- la **dimensione del batch** (`batch_size`),
- il **numero di qubit** (`n_qubits`) impiegati nella codifica,
- il **numero di layer** quantistici (`n_layers`),

  e vengono tracciate le metriche finali di training e validazione:

- **accuratezza** (`final_train_acc`, `final_val_acc`),

- **perdita** (`final_train_loss`, `final_val_loss`).

Questi valori sono stati salvati o registrati per valutare il comportamento dei modelli e confrontare le prestazioni del classificatore quantistico rispetto a quello classico in condizioni ottimizzate.

4.6.1 Preprocessing

Stabilito quale dataset che verrà utilizzato per l'addestramento e il testing della rete neurale quantistica da sviluppare, viene fatto il **preprocessing** dei dati raccolti.

Il *preprocessing* è una fase fondamentale nell'analisi dei dati per il **machine learning**. Si tratta di una serie di operazioni che permettono la trasformazione di dati grezzi in modo da poterli renderli utilizzabili per l'addestramento dei modelli.
E' stato realizzato a questo scopo uno script in grado di processare i file **CVS** (tipologia più comune per la raccolta di dati in formato tabellare, visualizzabile anche tramite *Microsoft Excel*) resi disponibili.
Prima di iniziare, viene indicato in un percorso segnalato dalla variabile *SAVEPATH* dove andranno salvati gli elementi processati, nel nostro esempio dati di training ed etichette costruite.
Vengono caricati i dati dalla cartella in cui sono contenuti i file *.csv* di riferimento del dataset utilizzato che è stato preparato nelle fasi precedenti. Ogni file rappresenta una porzione del dataset complessivo, e l'unione permette di ottenere un unico **DataFrame** Pandas verticale su cui poter operare.

Per motivi computazionali, viene eseguito un campionamento casuale di 2000 esempi, selezionati in modo da mantenere comunque la variabilità statistica del dataset nonostante sia un minor numero di campioni rispetto al totale del dataset.

```
folder_path = "C:/Users/ilnov/Desktop/Gerardo/Uni/Power"
csv_files = [f for f in os.listdir(folder_path) if f.endswith(".csv")]
df_list = [pd.read_csv(os.path.join(folder_path, file)) for file in csv_files]
df = pd.concat(df_list, ignore_index=True)
```

Figura 4.2: Caricamento ed unione dei file csv

Successivamente, il dataset viene suddiviso in variabili indipendenti (*feature*) e dipendenti (*target*). In particolare, la colonna dei **marker**, contenente le etichette delle classi, viene separata dal resto delle feature. Alcune colonne specifiche, se si sono rivelate con una distribuzione altamente sbilanciata (o con valori molto grandi), vengono trasformate attraverso una funzione logaritmica sicura (log1p) che stabilizza la distribuzione mantenendo i valori non negativi.

Inoltre, viene applicato un clipping superiore per limitare eventuali outlier estremi che potrebbero compromettere le elaborazioni successive. Si utilizzano i valori medi di ciascuna colonna per imputare i dati mancanti (*NaN*). Quindi il DataFrame viene riformato per mantenere i nomi delle colonne.

```
X = df.drop("marker", axis=1)
log_columns = ['R1-PM1:V', 'R2-PM1:V', 'R3-PM1:V', 'R4-PM1:V']
X[log_columns] = np.log1p(X[log_columns].clip(lower=0))
X = X.clip(upper=1e5)
imputer = SimpleImputer(strategy='mean')
X = pd.DataFrame(imputer.fit_transform(X), columns=X.columns)
```

Figura 4.3: Campionamento

Il dataset viene poi suddiviso in set di addestramento e test, con una proporzione rispettivamente dell'80% e del 20% (rapporto 80/20). Viene applicato uno *StandardScaler* per centrare i dati (media 0) e normalizzarne la varianza (deviazione standard 1).

Questa trasformazione è particolarmente utile perché rende tutte le feature compa-

rabili tra loro, anche se originariamente avevano scale diverse.
Questo viene fatto per stabilizzare l'addestramento e rendere i dati più adatti a PCA e ai modelli successivi.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Figura 4.4: Normalizzazione

Dopo la normalizzazione dei dati, si applica la **PCA** (Principal Component Analysis), una tecnica di riduzione dimensionale. Essa trasforma il dataset in un nuovo insieme di variabili, ordinate per importanza in termini di varianza spiegata. Viene scelto di mantenere solo le componenti che insieme rappresentano almeno il 95% della varianza totale. Questo permette di ridurre la complessità del dataset, mantenendo le informazioni essenziali per questa fase implementativa del codice.

```
pca = PCA(n_components=0.95)
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)
```

Figura 4.5: PCA

Dopo la PCA, le nuove feature ottenute vengono normalizzate nell'intervallo tra 0 e 1 utilizzando *MinMaxScaler*. Questa trasformazione è necessaria per poter andare a fornire input siano compresi entro un range definito e stabile.

Il dataset viene suddiviso in blocchi di dimensione fissa (4 feature per blocco). Poiché il numero totale di feature dopo la PCA può non essere un multiplo esatto di 4, si usa

il padding con zeri per completare ogni vettore. Ogni campione viene così ristrutturato in sotto-sequenze di 4 valori. Questa trasformazione viene attuata perché il successivo circuito quantistico (il *QCNN*) è progettato per lavorare su 4 qubit, quindi su blocchi di 4 input.

4.6.2 QAE con QCNN

Dopo aver completato il preprocessing, viene introdotto il primo quantum autoencoder dei due previsti dalla struttura in pila. La codifica è affidata a un *Quantum Convolutional Neural Network* (**QCNN**), un circuito quantistico parametrico progettato per trasformare ogni blocco di input in un insieme di feature quantistiche. Viene creato un simulatore quantistico con 4 **qubit** (uno per ogni feature del blocco). Sono 6 invece gli **strati** (*layers*) di parametri casuali per il circuito QCNN. Ogni layer contiene 4 parametri, uno per ogni qubit. Questi parametri controllano rotazioni e operazioni entangling che simulano l'effetto delle convoluzioni.

```
dev = qml.device("default.qubit", wires=4)
n_layers = 6
rand_params = pnp.random.uniform(high=2 * np.pi, size=(n_layers, 4))
```

Figura 4.6: Inizializzazione QCNN

Il circuito, definito come una **QNode**, inizia codificando i quattro valori numerici in rotazioni su ogni qubit, seguendo le rotazioni Y. Successivamente il circuito applica una serie di operazioni entangling denominate RandomLayers, che combinano rotazioni e porte *CNOT* in una struttura simile a quella di una rete convoluzionale

classica, ma ampliata nella complessità quantistica.

Successivamente il circuito applica una serie di operazioni entangling denominate *RandomLayers*, che combinano rotazioni e porte CNOT in una struttura simile a quella di una rete convoluzionale classica, ma ampliata nella complessità quantistica. Infine, ciascun qubit passa per l'operatore *Pauli-Z*, restituendo quattro valori numerici che rappresentano la trasformazione quantistica del blocco originale.

```
@qml.qnode(dev)
def circuit(phi):
    for j in range(4):
        qml.RY(np.pi * phi[j], wires=j)
    RandomLayers(rand_params, wires=list(range(4)))
    return [qml.expval(qml.PauliZ(j)) for j in range(4)]
```

Figura 4.7: Circuito QCNN

La funzione denominata **quanv_tabular** applica sistematicamente il circuito a ogni blocco di ciascun esempio del dataset. L'intero dataset di training e di test viene trasformato in due matrici di feature quantistiche, pronte per essere impiegate nella fase di decoding e classificazione. Durante questa fase, per ogni decina di esempi processati, viene stampato un messaggio di avanzamento, così da monitorare l'avanzamento del dataset sul circuito quantistico.

```
def quanv_tabular(blocks):
    output = []
    for i, sample_blocks in enumerate(blocks):
        sample_out = []
        for block in sample_blocks:
            result = circuit(block)
            sample_out.extend(result)
        output.append(sample_out)
        if i % 10 == 0:
            print(f"Processati {i+1}/{len(blocks)}")
    return np.array(output)
```

Figura 4.8: Quav tabular

Dopo la codifica via QCNN, il codice introduce una funzione di decodifica molto semplificata che opera invertendo i valori quantistici ottenuti (1 - valore misurato), completando simbolicamente il flusso della pipeline QAE per com'era prevista nel lavoro di partenza. I risultati vengono salvati insieme alle feature originali per i confronti che verranno fatti per validare la differenza con i nuovi risultati del modello quantistico nell'ultima parte del codice.

4.6.3 QAE con QRNN

Seguendo la stessa impostazione generale per il secondo **QAE** della pila, il quantum autoencoder va a includere una rete neurale ricorrente quantistica. Viene utilizzato un **QRNN** (*Quantum Recurrent Neural Network*) con l'obiettivo di introdurre una correlazione tra i qubit vicini, similmente a come le RNN trattano dipendenze sequenziali.

Per cominciare questa sezione, dopo aver effettuato una selezione manuale delle prime sei feature (quanti i qubit che verranno impiegati per il prossimo circuito quantistico) dei dataset di training e test. I dati selezionati vengono convertiti poi

in tensori compatibili con *PyTorch*, perché il training verrà eseguito attraverso un modello costruito con PyTorch e con componenti quantistici integrati tramite *PennyLane*.

Per la definizione del circuito quantistico utilizzato per QRNN, la codifica dei qubit avviene tramite *AngleEmbedding*, che li trasforma in rotazioni lungo l'asse Y, che apre al QAE.

A seguire, l'**encoder** quantistico applica tre tipi di rotazioni (RX, RY, RZ) su ciascun qubit, ciascuna parametrizzata da pesi ottimizzabili, ricordando che l'encoding ci permette di comprimere i dati considerati. Poi viene introdotto un entanglement controllato: solo coppie di qubit adiacenti vengono entangled con porte *CNOT*. Questa scelta mira a ridurre la complessità e imitare una struttura ricorrente. Successivamente, viene eseguito il **decoder**, che applica un secondo set di rotazioni RX, RY, RZ.

```python
# === QUANTUM CIRCUIT (Semplice rotazioni + entanglement limitato) ===
def qrnn_qae_circuit(inputs, weights_enc, weights_dec):
    qml.AngleEmbedding(inputs, wires=range(n_qubits))

    # Encoder: solo rotazioni RX, RY senza entanglement o con entanglement limitato
    for i in range(n_qubits):
        qml.RX(weights_enc[i, 0], wires=i)
        qml.RY(weights_enc[i, 1], wires=i)
        qml.RZ(weights_enc[i, 2], wires=i)

    # Entanglement limitato: solo CNOT fra qubit pari e dispari (esempio)
    for i in range(0, n_qubits - 1, 2):
        qml.CNOT(wires=[i, i + 1])

    # Decoder: rotazioni inverse
    for i in range(n_qubits):
        qml.RX(weights_dec[i, 0], wires=i)
        qml.RY(weights_dec[i, 1], wires=i)
        qml.RZ(weights_dec[i, 2], wires=i)

    return [qml.expval(qml.PauliZ(i)) for i in range(n_qubits)]
```

Figura 4.9: QRNN

4.6.4 Classificatore

Raggiunta l'ultima porzione del codice a cui si è lavorato, l'output ricostruito viene utilizzato come input per un **classificatore**. In questa sezione vengono introdotti due modelli distinti: uno **quantistico** ibrido che rispetta la descrizione della pipeline stabilita, con i due autoencoder quantistici in pila; e uno completamente **classico**, utilizzato come baseline di confronto.

L'obiettivo è quello di valutare in che misura la codifica quantistica dell'informazione possa contribuire ad ottenere un risultato finale migliore sui dati analizzati, validando la premessa iniziale del lavoro svolto per questo elaborato.

```
class FullQuantumModel(nn.Module):
    def __init__(self, n_classes):
        super().__init__()
        self.qae = q_layer
        self.classifier = nn.Linear(n_qubits, n_classes)

    def forward(self, x):
        x_encoded_decoded = self.qae(x)
        logits = self.classifier(x_encoded_decoded)
        return logits, x_encoded_decoded

class ClassicalModel(nn.Module):
    def __init__(self, input_dim, n_classes):
        super().__init__()
        self.classifier = nn.Sequential(
            nn.Linear(input_dim, 12),
            nn.ReLU(),
            nn.Linear(12, n_classes)
        )

    def forward(self, x):
        return self.classifier(x)
```

Figura 4.10: Definizione modelli

Per addestrare e valutare i due modelli (quantistico e classico), viene definita una funzione `train_model` che incapsula l'intero processo di training, validazione e monitoraggio delle prestazioni. Questa funzione riceve come input il modello da addestrare, i dati suddivisi in training e validation set, il numero di epoche e la dimensione del batch.

La funzione prevede l'inizializzazione di un ottimizzatore *Adam*, con **learning rate adattivo**: è fissato a `1 e-3` per i modelli quantistici (più delicati alla variazione dei parametri) e `1e-2` per i modelli classici.

Per la parte classificativa, si usa la `CrossEntropyLoss`, mentre per i modelli quantistici, che includono un **autoencoder**, si aggiunge anche una `MSELoss` che misura l'errore di ricostruzione tra input e output del circuito quantistico.

```
def train_model(model, X_train, y_train, X_val, y_val, epochs=15, batch_size=8, is_quantum=False):
    # Learning rate più basso per quantum
    lr = 1e-3 if is_quantum else 1e-2
    optimizer = optim.Adam(model.parameters(), lr=lr)
    loss_fn = nn.CrossEntropyLoss()
    recon_loss_fn = nn.MSELoss()

    train_acc, val_acc = [], []
    train_loss, val_loss = [], []
    quantum_gradients = []
```

Figura 4.11: Train Model

Durante ogni epoca, i dati di training vengono mischiati casualmente e suddivisi in mini-batch. Per ogni batch, il modello esegue una forward pass, calcola la loss, effettua il backpropagation e aggiorna i pesi.

```
for epoch in range(epochs):
    model.train()
    perm = torch.randperm(X_train.size(0))
    epoch_loss = 0
    correct = 0

    for i in range(0, X_train.size(0), batch_size):
        idx = perm[i:i + batch_size]
        batch_x, batch_y = X_train[idx], y_train[idx]

        optimizer.zero_grad()

        if is_quantum:
            outputs, recon = model(batch_x)
            loss = loss_fn(outputs, batch_y) + recon_loss_fn(recon, batch_x)
        else:
            outputs = model(batch_x)
            loss = loss_fn(outputs, batch_y)

        loss.backward()
```

Figura 4.12: Epoche e batch

Alla fine di ogni epoca, vengono salvati i valori che i due modelli hanno restituito per l'accuratezza e la loss (perdita).

```
    model.eval()
    with torch.no_grad():
        if is_quantum:
            val_outputs, _ = model(X_val)
        else:
            val_outputs = model(X_val)
        val_l = loss_fn(val_outputs, y_val).item()
        val_loss.append(val_l)
        val_a = (val_outputs.argmax(dim=1) == y_val).float().mean().item()
        val_acc.append(val_a)

    print(f"Epoch {epoch + 1}: Train Acc={train_acc[-1]:.3f}, Val Acc={val_acc[-1]:.3f}")
```

Figura 4.13: Valutazione

Per l'avvio del training, si istanziano i due modelli da confrontare: uno **quantistico**, che integra un autoencoder quantistico (QAE) seguito da un classificatore lineare, e uno **classico**, per come descritto precedentemente.

Il numero di classi `n_classes` viene calcolato dinamicamente sulla base del target `y`. I modelli vengono poi addestrati mediante la funzione `train_model`, e l'argomento `is_quantum` viene impostato opportunamente per attivare o disattivare il ramo quantistico del training.

Al termine dell'addestramento, per ciascun modello vengono raccolte le metriche di **accuratezza** e di **perdita**, sia sul training set che sul validation set.

```
n_classes = len(np.unique(y))
quantum_model = FullQuantumModel(n_classes)
classical_model = ClassicalModel(input_dim=6, n_classes=n_classes)

q_train_acc, q_val_acc, q_train_loss, q_val_loss = train_model(
    quantum_model, X_tensor, y_tensor, X_test_tensor, y_test_tensor, is_quantum=True)

c_train_acc, c_val_acc, c_train_loss, c_val_loss = train_model(
    classical_model, X_tensor, y_tensor, X_test_tensor, y_test_tensor, is_quantum=False)
```

Figura 4.14: Training

Infine, i risultati vengono visualizzati in due grafici affiancati, realizzati con *matplotlib*. Il primo mostra l'andamento dell'accuratezza di validazione per entrambi i modelli nel corso delle epoche, evidenziando le differenze di performance tra approccio quantistico e classico. Il secondo rappresenta invece l'andamento della loss di validazione, utile per analizzare il comportamento dell'ottimizzazione. Le curve sono codificate con colori e marker distinti (-ob per il modello quantistico in blu, -or per quello classico in rosso), ed è presente una legenda per ciascun grafico.

```
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.plot(q_val_acc, "-ob", label="Quantum Val Acc")
plt.plot(c_val_acc, "-or", label="Classical Val Acc")
plt.title("Validation Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(q_val_loss, "-ob", label="Quantum Val Loss")
plt.plot(c_val_loss, "-or", label="Classical Val Loss")
plt.title("Validation Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()

plt.tight_layout()
plt.show()
```

Figura 4.15: Plot accuracy e loss

4.6.5 Confronto

Per valutare le prestazioni dei due approcci sviluppati, sono stati registrati i risultati dei due modelli in termini di accuratezza e loss sul set di validazione, al variare delle epoche di addestramento.

Come si può denotare dai due esempi riportati in basso, nell'accuracy entrambi i modelli raggiungono una validazione simile, attorno al 72%, con un leggero vantaggio per il **modello quantistico**, che mostra maggiore stabilità e che soprattutto risulta crescente nella parte finale delle epoche.

Nel paragone in termini di loss, il **quantum model** mantiene un profilo di perdita decrescente e coerente, a differenza del modello classico, che inizia con valori più bassi ma poi tende a peggiorare, suggerendo sensibilità alla variabilità dei batch.

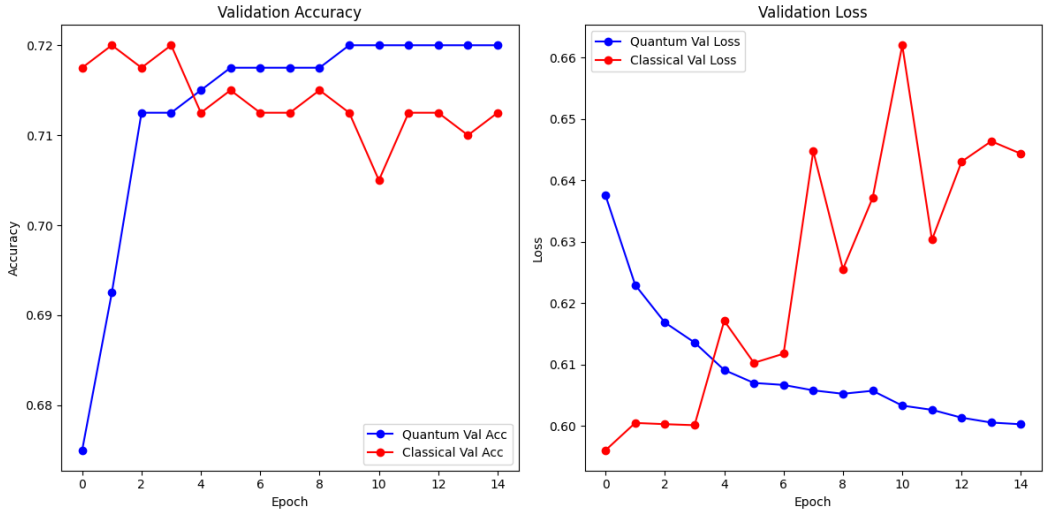

Figura 4.16: Confronto finale 1

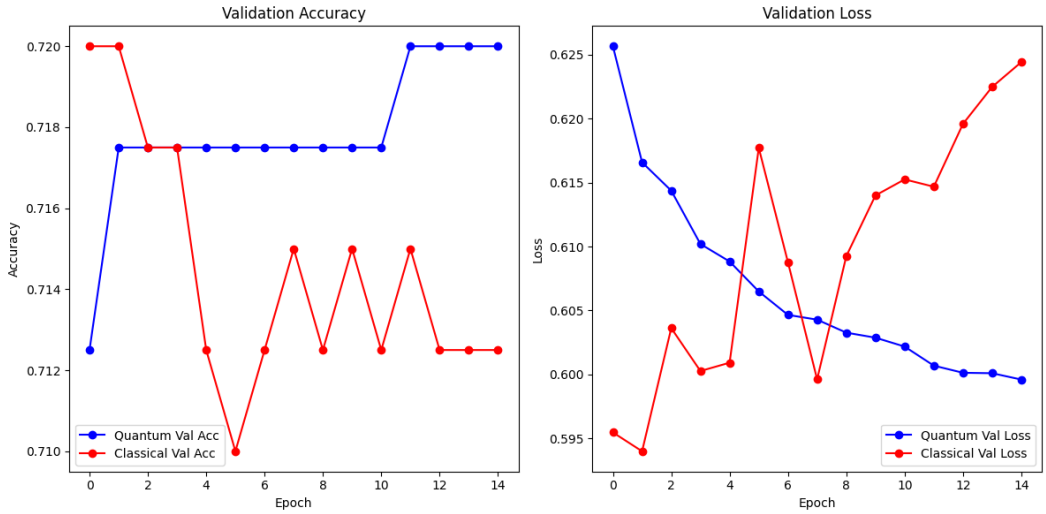

Figura 4.17: Confronto finale 2

4.6.6 Conclusioni

Il modello quantistico ha mostrato un comportamento di apprendimento più stabile e regolare rispetto al modello classico, con una perdita di validazione meno soggetta a fluttuazioni. Questo risultato indica una buona capacità di generalizzazione, favorita

dalla compressione efficace operata dal QAE.

5 Conclusioni

Dopo aver approfondito il funzionamento delle reti neurali classiche e delle basi del quantum computing, ci siamo concentrati sullo sviluppo dell'architettura quantistica per la pipeline presa in esame, per procedere poi col confronto tra il nuovo modello e il precedente.

L'esperimento ha mostrato che il modello quantistico è in grado di raggiungere prestazioni migliori rispetto al modello classico, sia in termini di accuratezza che di stabilità del processo di apprendimento.

L'analisi dell'andamento del **loss di validazione** ha evidenziato una **maggiore regolarità e robustezza** dell'addestramento nel caso quantistico, con una decrescita più stabile e meno soggetta a fluttuazioni. Ciò suggerisce che la presenza del QAE come struttura iniziale del modello possa aver contribuito a una **compressione efficace delle informazioni**, migliorando la generalizzazione del classificatore. La qualità dell'apprendimento e la stabilità delle prestazioni finali rendono il modello quantistico una promettente alternativa anche con risorse computazionali contenute.

Tra i possibili sviluppi futuri, si suggerisce l'esplorazione di **nuove modalità d'applicazione dei tipi di circuiti quantistici considerati**, come le **Quantum Convolutional Neural Networks (QCNN)** e le **Quantum Recurrent Neural Networks (QRNN)**, testando quelli che promettono una maggiore espressività del

modello; e lo studio di strategie alternative per l'implementazione degli encoder e decoder del QAE; se non valutare di testare e confrontare una pipeline differente rispetto a quella proposta per il trattamento dei dati presenti nel dataset per le reti neurali quantistiche.

Elenco delle figure

Bibliografia

[1] CNN. Spain rules out cyberattack in april blackout that left thousands without power, 2025. Accessed: 2025-06-20.

[2] Tomorrow Bio. Peso per esso: Come le reti neurali diventano più forti, 2023. Accessed: 2025-04-05.

[3] IBM. Cos'è il deep learning?, 2024. Accessed: 2025-04-05.

[4] Huang Yi, Sun Shiyu, Duan Xiusheng, and Chen Zhigang. A study on deep neural networks framework. In *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, pages 1519–1522, 2016.

[5] Gianni D'Angelo and Francesco Palmieri. A stacked autoencoder-based convolutional and recurrent deep neural network for detecting cyberattacks in interconnected power control systems. *International Journal of Intelligent Systems*, 36(12):7080–7102, 2021.

[6] IBM Quantum. Quantum circuits - basics of quantum information, 2024. Accessed: 2025-04-29.

[7] Bachir Barika. Power system. `https://www.kaggle.com/datasets/bachirbarika/power-system`, 2019. Accessed: April 30, 2025.

[8] University of Alabama in Huntsville. Power system dataset readme. `http://www.ece.uah.edu/~thm0009/icsdatasets/PowerSystem_Dataset_README.pdf`, 2023. Accessed: April 30, 2025.

Dediche e ringraziamenti

A conclusione di questo elaborato, desidero menzionare le persone senza le quali non avrei avuto il supporto necessario a compiere questi anni di studio.

Ringrazio i miei amici di sempre, Ivan, Gerardo, Antonio, Federico e Luigi, per essermi da anni vicino e per aver creduto in me.

Ringrazio Tonino, Alessandra, Rosa, Peppe, Lucia, Davide, Claudia, Cristian, Samuel, Giuseppe ed Amanda per la loro amicizia, per me mai scontata e sempre preziosa.

Ringrazio la mia famiglia, mia madre, mio padre e mio fratello per avermi permesso di arrivare fin qui. Ringrazio per l'affetto che mi hanno sempre dimostrato le mie zie Anna, Patrizia, Lucia, mio zio Raffaele e soprattutto mio nonno Gerardo.

Menzione d'onore per Alessio, grazie per avermi sopportato per tutta la durata della magistrale.

Dulcis in fundo, grazie al cherubino per essere quasi sempre stato presente alle 13.05 sul gruppo di Whatsapp.