Università degli Studi Milano-Bicocca

FACOLTA' DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Triennale in Fisica

Classificazione con Reti Neurali Quanto-Classiche

Relatore:
Dr. Andrea Giachero

Correlatori:
Prof. Alberto Leporati
Prof. Claudio Ferretti

Candidato:
Gaia Stella Bolognini
Matricola 838719

Anno accademico 2020-2021

Ai miei genitori
e alla mia famiglia

Sommario

Il *Machine Learning*, ovvero l'insieme di meccanismi che permettono a una macchina di migliorare le proprie capacità e prestazioni nel tempo agendo su dati, ha permesso di ottenere importanti risultati e di risolvere notevoli problemi in diversi settori, si pensi ad esempio al riconoscimento vocale o alle pubblicità traccianti. Le *reti neurali* sono state sviluppate costruendo macchine con architettura ispirata al sistema nervoso umano per compiti di Machine Learning; nello specifico caso del *Deep learning* l'apprendimento avviene tramite modelli matematici e algoritmi di calcolo statistico in maniera automatica. *Teoricamente* le reti neurali sono capaci di simulare ogni algoritmo di apprendimento se possiedono un sufficiente numero di dati, soddisfando in alcuni casi la condizione di *Turing Completeness*. Può però sorgere un problema legato proprio al set di dati che, se di grandi dimensioni, può fortemente incidere sul tempo impiegato per l'apprendimento. É in questo quadro che si inserisce la possibilità dell'utilizzo dei *computer quantistici*.

I computer quantistici fanno uso delle proprietà della meccanica quantistica, come *l'interferenza* tra funzioni d'onda, *l'entanglement* e la *sovrapposizione* degli stati, per eseguire calcoli e computazione. Come i computer classici codificano l'informazione e i dati in bit (0 o 1), i computer quantistici utilizzano i bit quantistici, *qubits*, che si possono trovare in sovrapposizone di stati, assumendo un *continuo* di valori. Grazie a questa proprietà, queste macchine potrebbero risolvere problemi con un notevole risparmio di tempo rispetto ai computer classici, condizione nota come *Quantum supremacy*. In maniera analoga alle porte logiche che si trovano in computazione classica, diverse operazioni possono essere eseguite sui qubit: si parla in questo caso di *gate* quantistici. Attualmente diverse aziende, tra cui Google, IBM, Intel, Rigetti e D-Wave, stanno sviluppando i loro componenti quantistici e la ricerca in questi settori si concentra sia sugli algoritmi che possono essere utilizzati su questo tipo di computer sia sulle sfide tecnologiche, entrando nella fisica della materia per sfruttarne le proprietà quantistiche.

La possibilità di utilizzare i computer quantistici al fine di ridurre in maniera significativa il tempo di computazione e le dimensioni del set di dati utili all'apprendimento vede la nascita del *Quantum Machine Learning*. La ricerca in questo ambito è ancora allo stato dell'arte ma sono diverse le idee sviluppate che potrebbero portare a benefici. Per esempio, alcune operazioni di ottimizzazione potrebbero essere eseguite più velocemente da processori basati su *Quantum Annealing*, dove viene impiegata la fisica quantistica per trovare stati a energia più bassa per un sistema. Nel breve termine gli algoritmi ibridi, ovvero funzionanti in parte su un computer classico e in parte su uno quantistico, sono i più promettenti. In questo contesto si inseriscono le reti neurali ibride, circuiti

quantistici dipendenti da alcuni parametri da ottimizzare per operazioni di classificazione.

Nel lavoro di tesi si studiano le caratteristiche di una particolare implementazione di Quantum Machine Learning utilizzando delle reti neurali ibride per il riconoscimento di immagini, digit MNIST, in un contesto di *Supervised Machine Learning*. All'interno delle *layer* nascoste di una rete neurale viene inserita una layer costruita con un circuito quantistico dipendente da alcuni parametri che sono il risultato delle layer precedenti e vengono utilizzati come angoli di rotazione per i gate interni al circuito quantistico. Possono essere implementate diverse operazioni sui qubit del circuito, per esempio l'entanglement o gate dedicati, o possono essere aggiunti qubit. La misura sul circuito viene utilizzata per continuare la computazione classica ottenendo un output dalla rete neurale che possa essere confrontata numericamente con il risultato atteso, a partire dalla *loss function*. I parametri inseriti nel circuito vengono poi ottimizzati insieme ai parametri delle layer classiche con l'utilizzo dell'algoritmo di *backpropagation*, dove il metodo di ottimizzazione viene implementato per essere compatibile con il circuito quantistico, attraverso il meccanismo di *parameter shift rule*. L'obiettivo è minimizzare la loss function e creare un modello che sia in grado di fare previsioni corrette sui dati in ingresso. I programmi sono stati implementati con le librerie *Qiskit* e *Pytorch* fornite per il linguaggio Python e i vari programmi sono stati testati su simulatori e su computer quantistici, utilizzando la piattaforma online *IBM Quantum Experience*, che mette a disposizione via cloud i computer quantistici costruiti da IBM.

Sono stati effettuati diversi esperimenti a partire dal modello sopra descritto, modificando le caratteristiche del circuito quantistico e il numero di qubit. Sono stati inoltre aggiunti errori gaussiani al variare della deviazione standard nel set di dati di validazione, per testare il modello, e nel set di dati di apprendimento, per rafforzarlo. In particolare sono stati implementati i seguenti circuiti: un circuito con una rotazione in tre dimensioni, gate U3, per un singolo qubit, un circuito a più qubit, un circuito a qubit entangled e un circuito ispirato a un algortimo ibrido di ottimizzazione, QAOA. In tutti questi casi si è ottenuto un riconoscimento delle immagini con un'accuratezza intorno al 99% sia sul simulatore quantistico che sul computer reale. Il primo dei circuiti si è inoltre rivelato promettente per il riconoscimento di dati rumorosi, con un'accuratezza superiore al 99% fino a un valore di deviazione standard pari a 0.7. Anche il circuito a più qubit ha dato buoni riusltati sul set di dati rumorosi quando l'apprendimento avveniva a sua volta con dati rumorosi (deviazione standard pari a 0.2).

La tesi è suddivisa in tre capitoli:

1. Nel primo capitolo viene riportato un approfondimento riguardo alla teoria sottostante al Machine Learning e ai Computer Quantistici, con particolare attenzione alle caratteristiche utilizzate per implementare il lavoro di tesi.

2. Nel secondo capitolo la prima parte è dedicata ad una visione di insieme riguardo al Quantum Machine Learning. Vengono poi evidenziate le caretteristiche tecniche dei programmi implementati e vengono descritti i vari esperimenti effettuati.

3. Il terzo capitolo è dedicato ad un'analisi critica sui risultati ottenuti, alle possibili future implementazioni del modello e alle conclusioni.

Indice

Capitolo 1

Il Machine Learning e i Computer Quantistici

1.1 Machine Learning

La complessità del nostro organismo è davvero stupefacente. Diversi sono gli esempi che si possono portare a testimonianza di questa tesi, dal ciclo metabolico alla reazione davanti ad una situazione pericolosa. Tutti questi meccanismi avvengono in maniera a noi completamente inconscia, il processo è completamente automatizzato. Questi esempi sono comuni a quasi tutti gli esseri viventi; sono frutto di un lento e complesso processo di evoluzione. Ancora più meraviglioso è quello che siamo in grado di fare con il nostro cervello. La capacità di apprendere, di fare errori e di corregerci è stata la caratteristica che ha permesso agli esseri umani di prevalere sulle altre specie. Il processo di apprendimento a noi sembra così scontato che neanche ci soffermiamo a pensarci; riuscire però a spiegare il suo funzionamento è tutt'altro che banale.
La difficoltà diventa evidente quando proviamo a far riconoscere un'immagine ad un computer. Trovare tutti i particolari e i dettagli da inserire all'interno di un programma per effettuare un corretto riconoscimento sembra davvero una missione complicata. É qui che entra in gioco il *Machine Learning* e in particolare le *reti neurali* le quali, ispirandosi al sistema nervoso umano e applicando modelli matematici e statistici, cercano attraverso l'esperienza di migliorare le prestazioni di una macchina relativamente ad uno specifico compito.
Esistono tre metodi fondamentali attraverso cui una macchina è in grado di apprendere [22]:

- *Supervised Learning*: la macchina ricava a partire da un set di dati iniziali, *training data*, una funzione che sia in grado di fare previsioni corrette su di essi. In questo caso sono note le informazioni sui dati a disposizione, ovvero sappiamo a quale classe appartengono, e viene implementato un algoritmo che in maniera iterativa

fa previsioni ed è in grado di corregersi, apprendendo il nesso che unisce i dati in ingresso alla loro classe.

- *Unsupervised Learning*: in questo caso vengono usati dati senza informazione riguardante la classe di appartenenza. Viene implementato un algoritmo che sia in grado di fare distinzioni o riconoscere strutture interessanti nei dati a disposizione senza nessuna informazione a priori.

- *Reinforcement Learning*: anche in questo caso i dati a disposizione non contengono informazioni a priori sulla loro classe di appartenenza. La macchina prova diverse operazioni sui dati in ingresso e impara quali di queste siano le migliori in base ad un feedback e il suo lavoro viene quantificato attraverso una ricompensa.

Il lavoro di tesi qui presentato si sviluppa all'interno di un contesto di Supervised Machine Learning basato sul riconoscimento di cifre scritte a mano, note anche come *digit*. Trateremo dunque in maniera più approfondita questo caso.

1.1.1 Introduzione alle Reti Neurali

Nel 1949 D. O. Hebb ipotizza la possibilità di istruire le macchine con un apprendimento simile a quello su cui si basa l'apprendimento umano. Il prototipo per le reti neurali artificiali sono infatti quelle biologiche, nonostante molte caratteristiche siano differenti. Gli elementi costituenti sono i *neuroni*, ovvero funzioni matematiche, che vengono raccolti all'interno di strutture note come *layer*: la input layer è quella che graficamente si trova a sinistra della rete neurale, l'output layer è quella finale e ciascuna delle layer intermedie è nota come *hidden layer* (figura 1.1). Non c'è un modello certo per saper di quante layer deve essere costituita la rete e neanche il numero di neuroni per layer, ma uno degli obiettivi è cercare di trovare il numero ottimale al fine di massimizzare l'apprendimento [6].

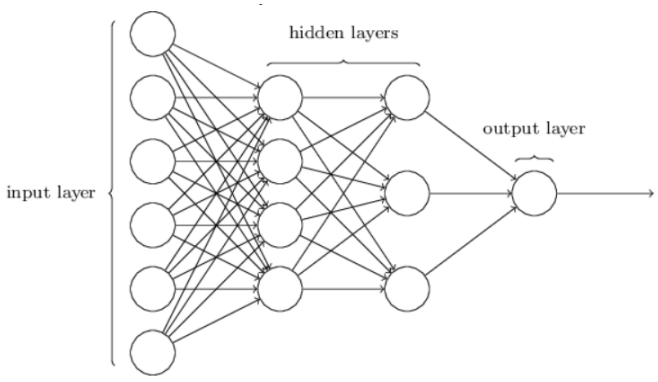


Figura 1.1: Rappresentazione delle layer che costituiscono una rete neurale

Ogni neurone riceve dei dati in ingresso dalla layer precedente e restituisce dei valori in uscita, utilizzati dalla layer successiva. Il collegamento di ciascun neurone con quelli precedenti o successivi viene garantito attraverso dei parametri che pesano l'importanza di ciascun input nel collegamento con quello specifico neurone. Questo collegamento può essere interpretato come una funzione lineare e i parametri sono noti come *weigths* e *bias*. I neuroni infine agiscono sui dati che ricevono attraverso una funzione non lineare nota come *activation function*. L'insieme di queste operazioni deve essere interpretata in forma vettoriale: il numero di valori in ingresso per il singolo neurone è tendenzialmente maggiore di uno e quindi questi valori vengono codificati in un vettore e i *weigths* rappresentati come una matrice. Si può dunque rappresentare l'insieme di queste operazioni nella forma matriciale:

$$\begin{bmatrix} a_0 \\ a_1 \\ .. \\ .. \\ a_n \end{bmatrix} = \begin{bmatrix} w_{00} & w_{01} & ... & ... & w_{0m} \\ w_{10} & w_{11} & ... & ... & w_{1m} \\ ... & ... & ... & ... & ... \\ w_{n0} & w_{n1} & ... & ... & w_{nm} \end{bmatrix} \begin{bmatrix} in_0 \\ in_1 \\ .. \\ .. \\ in_m \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ .. \\ .. \\ b_n \end{bmatrix}$$

$$out_j^i = \sigma(\bar{\bar{w}}\bar{i} + \bar{b}) \tag{1.1}$$

Dove out_j^i è il valore in uscita del neurone j-esimo nella layer i-esima e σ rappresenta la funzione di attivazione. Quest'ultima ha due ruoli fondamentali: dare alla funzione di output del neurone diverse pendenze a diversi valori nelle layer interne della rete e concentrare gli output in un range specifico nell'ultima layer [25]. Funzioni di attivazione ottimali hanno di tendenza una regione centrale per cui gli output lineari sono sensibili, mentre gli altri valori si accumuleranno ai bordi. La funzione sigmoide e la tangente iperbolica possono rappresentare due buoni esempi. É da citare anche la funzione ReLU (*Rectified Linear Unit*) che viene considerata in alcuni casi come una delle funzioni di attivazione più performanti e che nel corso degli esperimenti di questa tesi è stata ampiamente utilizzata. Si riportano in figura 1.2 due esempi delle funzioni di attivazione di cui abbiamo parlato.

L'apprendimento si basa quindi sulla stima dei migliori parametri (*weigths* e *bias*) del modello che possano fare previsioni corrette sulla classe di appartenenza dei dati in ingresso.

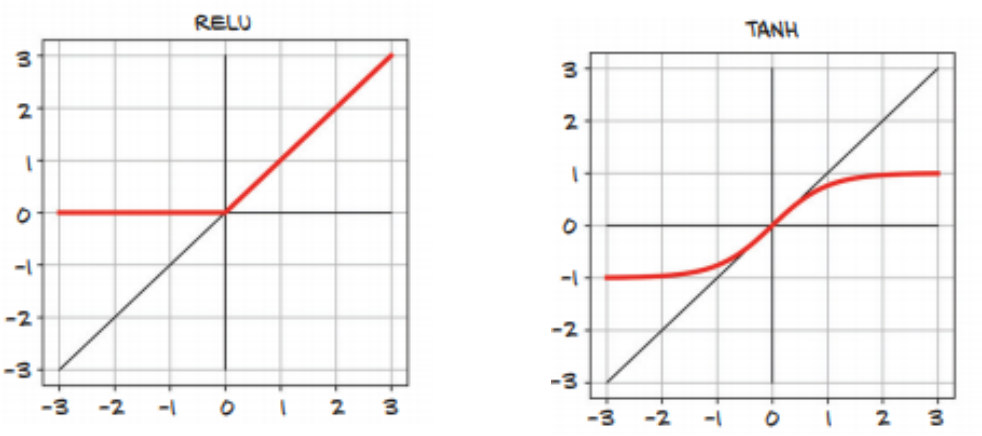


Figura 1.2: Esempi di due funzioni di attivazione, ReLU e la tangente iperbolica, [25].

1.1.2 L'apprendimento delle reti neurali

Per prima cosa è necessario definire una funzione dipendente dai parametri del modello, nota come *loss function* o anche *cost function*, che possa valutare numericamente la differenza tra i risultati corretti e quelli in uscita dal modello stesso. L'obiettivo sarà ottimizzare i parametri al fine di minimizzare la loss function e l'algoritmo che permette di eseguire questa operazione è noto come *backpropagation*, in cui l'ottimizzazione avviene attraverso l'algoritmo di *gradient descent* [25]. L'idea generale consiste nell'aggiornare i parametri verso la direzione di decrescita della funzione, dove la direzione può essere ricavata a partire dal gradiente, il vettore delle derivate parziali della funzione rispetto a ciascuno dei parametri. I parametri tuttavia sono definiti all'interno di un calcolo complesso (ciascun input per uno specifico neurone, che è a sua volta una funzione, viene calcolato come risultato delle operazioni sulle layer precedenti); è necessario quindi utilizzare la *chain rule* per ciascuna delle derivate effettuate. L'equazione 1.2 rappresenta questa concatenazione di operazioni nel caso di layer costituite da un solo neurone, effettuando la derivata rispetto ai parametri del modello (weigths w e bias b). C è la loss function, definita come la differenza in quadratura (mean squared error) tra l'output dell'ultimo neurone L-esimo ($out^{(L)}$) e y che è il valore atteso. $out^{(L)}$ è definito a sua volta come l'applicazione della funzione di attivazione (σ) su $a^{(L)}$ che è il risultato della funzione lineare dipendente dai parametri della layer L-esima sui risultati del neurone precedente. Questi ultimi dipendono a loro volta dalle funzioni e dai parametri delle layer precedenti della rete neurale.

$$
\begin{aligned}
C(...) &= (out^{(L)} - y)^2 \\
out^{(L)} &= \sigma(a^{(L)}) \\
a^{(L)} &= w^{(L)} a^{(L-1)} + b^{(L)} \\
\frac{\partial C}{\partial w^{(L)}} &= \frac{\partial a^{(L)}}{\partial w^{(L)}} \frac{\partial out^{(L)}}{\partial a^{(L)}} \frac{\partial C}{\partial out^{(L)}}
\end{aligned}
\tag{1.2}
$$

Dove $\frac{\partial C}{\partial w^{(L)}}$ definisce uno degli elementi all'interno del gradiente della funzione:

$$
\nabla C(x) = \begin{bmatrix} \partial C/\partial w_1 \\ \partial C/\partial b_1 \\ \partial C/\partial w_2 \\ .. \\ .. \\ \partial C/\partial w_n \\ \partial C/\partial b_n \end{bmatrix}
$$

Questa operazione viene poi estesa per le derivate rispetto ai parametri dei neuroni precedenti. Infine, viene estesa al caso di più neuroni per ogni layer. Essenzialmente questo si riduce a passare ad un calcolo in forma matriciale, dove sono presenti più indici per i diversi componenti che abbiamo definito nelle equazioni precedenti (per esempio w^L_{jk} rappresenta il parametro per la layer L-esima che connette il neurone j-esimo della layer (L-1)-esima e il neurone k-esimo della layer L-esima).

Una volta calcolato il gradiente della funzione rispetto a ciascuno dei parametri da cui dipende il modello, questi vengono aggiornati scalando i parametri di un fattore proporzionale al gradiente della funzione, dove il fattore di proporzionalità è detto *learning rate*. La scelta di questo parametro è cruciale negli esperimenti; non deve essere troppo grande perchè in questo modo non verrebbe trovato il minimo ma neanche troppo piccolo, in quanto non garantirebbe un aggiornamento sufficiente dei parametri. In figura 1.3 viene riportata una rappresentazione grafica di questo aspetto.

L'insieme di queste operazioni per l'aggiornamento dei parametri avviene in maniera iterativa all'interno di un *training loop* dove ciascuna iterazione è nota come epoca (*epoch*). In alternativa il processo può essere ripetuto fino a quando non viene incontrato un certo criterio per la loss function.

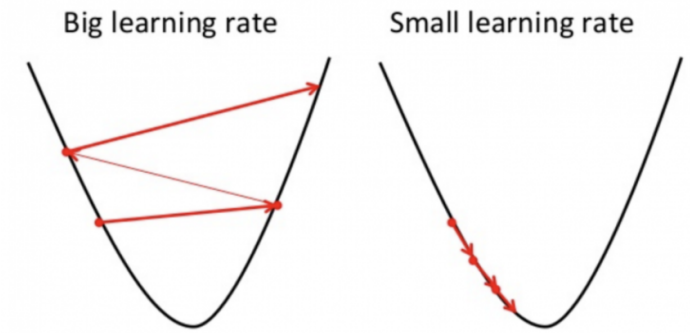


Figura 1.3: Rappresentazione grafica per evidenziare come la corretta scelta del learning rate sia cruciale negli esperimenti di machine learning: se questo valore è troppo elevato si potrebbe non trovare il minimo della funzione convessa ma se troppo piccolo l'aggiornamento dei parametri sarebbe molto lento.

Per riassumere, definendo con *model* il modello dipendente dai nostri parametri (w, b, in cui viene sottoistesa la forma vettoriale e matriciale), con $loss_{fn}$ la loss function e con $grad_{fn}$ la funzione che permette di calcolare il gradiente, per ciascuna epoca si effettuano le operazioni riportate nelle equazioni 1.3, 1.4 e 1.5:

$$w, b = params \tag{1.3}$$

$$\begin{aligned} out &= model(in, w, b) \\ loss &= loss_{fn}(out, correct) \\ grad &= grad_{fn}(out, w, b) \end{aligned} \tag{1.4}$$

$$params = params - LR \cdot grad \tag{1.5}$$

Dove *in*, *out* e *correct* sono rispettivamente i dati in ingresso, il valore ricavato dal modello e il risultato che vogliamo ottenere e LR rappresenta il learning rate. Il passaggio in cui viene valutato il modello in dipendenza da certi parametri e il calcolo del gradiente della funzione sono detti nel linguaggio del Machine Learning *Forward pass* e *Backward pass*.
Le funzioni impiegate possono essere definite in maniera autonoma oppure possono essere utilizzate alcune librerie già implementate, utili soprattutto quando si ha un grande numero di parametri per cui calcolare il gradiente non sarebbe scontato.

Una volta trovato il modello che si pensa essere quello corretto, è necessario poterlo testare a partire da un nuovo set di dati. Questo set di dati viene tipicamente scelto in maniera da essere indipendente rispetto a quello usato per l'apprendimento, per evitare fenomeni di *overfitting*, ovvero il caso in cui il nostro modello impari molto bene le

caratteristiche del set di dati ma non sia in grado di estrapolare delle informazioni generali sulla loro classe di appartenenza. Dividiamo dunque il set di dati di ingresso in *training set*, su cui effettuare l'apprendimento, e in un *validation set*, per validare il modello. Viene stimata in questo modo l'accuratezza del nostro modello, ovvero quanti previsioni sono giuste rispetto al numero totale di confronti che sono stati effettuati.

1.1.3 La libreria Pytorch

Pytorch [7] è un framework di deep learning per programmi sviluppato in python dal *Facebook AI Research* group.
La struttura base di questa libreria è il *Tensore* [25], ovvero un vettore multidimensionale, simile ai vettori di NumPy [17], che contiene una collezione di numeri accessibili individualmente con degli indici. Una caratteristica fondamentale dei tensori di pytorch è la loro possibilità di essere raccolti all'interno della GPU (graphics processing unit) in modo da eseguire computazioni paralllele e più rapide rispetto ai tensori raccolti nella CPU.
Il componente di Pytorch *autograd* permette di effettuare l'algoritmo di backpropagation in maniera autonoma. I tensori di PyTorch possiedono una caratteristica interessante: la possibilità di tenere memoria delle operazioni e dei tensori che hanno originato loro stessi e quindi di fornire in maniera automatica la catena di derivate di queste operazioni. É possibile impostare questa proprietà nel momento in cui si definisce il tensore attraverso l'attributo requires_grad = True:

$$\text{Params} = \text{torch.tensor ([1, 1, ..., 1], requires\_grad = True)}$$

Il valore delle derivate sarà contenuto all'interno del'attributo grad del tensore Params. Una volta definito questo tensore, è sufficiente calcolare il modello e la loss function in maniera analoga a quanto fatto sopra e chiamare l'attributo della loss function:

$$\text{loss.backward()}.$$

A questo punto l'attributo grad del tensore params conterrà le derivate della loss function rispetto a ognuno dei parametri. Un punto delicato che è bene notare riguardo all'attributo grad è la necessità di azzerarlo dopo ogni operazione: infatti il valore delle derivate in seguito alla chiamata della funzione backward viene accumulato nel grad del tensore, quindi sommato, e non semplicemente raccolto.
Un'altro interessante modulo di PyTorch è toch.optim, che implementa diversi algoritmi di ottimizzazione per aggiornare i parametri sotto forma di funzioni, che dipendono dal tensore da ottimizzare e dal leanring rate selezionato dal programmatore. In questo modo si può effettuare l'aggiornamento dei parametri in maniera automatica. La scelta dell'optimizer da usare è flessibile: alcuni esempi sono 'SGD' [23] e 'Adam' [19] che sono stati utilizzati per la maggiore nella tesi.

PyTorch ha inoltre un modulo dedicato alla definizione delle reti neurali, *torch.nn* che contiene gli elementi per costruire ogni possibile struttura di neural network, ovvero le diverse *layer*. Questo modulo permette inoltre di utilizzare delle loss function già implementate in precedenza, senza bisogno di ridefinirle, per esempio la MSELoss (ovvero Mean Square Error) e la NNLoss (Negative Log Likelihood). Si accennerà alle diverse caratteristiche del modulo in seguito.

1.1.4 Il riconoscimento di immagini e il pacchetto MNIST

La classificazione di immagini, ovvero riconoscere a quale categoria certe immagini appartengono, è una importante applicazione delle reti neurali. Questo compito è incluso all'interno di un più vasto campo dell'intelligenza artificiale noto come *Computer Vision*, in cui la macchina cerca di derivare informazioni rilevanti da immagini digitali, video e altri input visivi ed eseguire operazioni in base a queste informazioni. Questo tipo di tecnologia ha già avuto numerose applicazioni, per esempio nei veicoli a guida autonoma e nell'industria per l'identificazione di possibili difetti nei prodotti [1].
Il modulo di pytorch *Dataset* permette di poter scaricare immagini di diverso tipo, dal pacchetto MNIST per il riconosciento di numeri scritti a mano (figura 1.4) [5], a CIFAR10 che contiene oggetti reali come aerei e uccelli. É necessario però comprendere come sia possbile trasformare un'immagine in un tensore che possa essere utilizzato effettivamente nel programma python. Ogni immagine può essere vista come un insieme di *pixel*, ovvero una matrice contenente o singoli valori che rappresentano la scala del grigio per immagini in bianco-nero oppure più valori (tipicamente tre) per immagini colorate. Sfruttando questa proprietà, le immagini possono essere caricate e trasformate in maniera automatica in tensori.

Figura 1.4: Esempio immagini conteute nel pacchetto MNIST

Questo tensore viene inserito come input nella rete neurale. Si ha la possibilità di definire la rete come una classe di python e un esempio di una sua implementazione è riportato in figura 1.5. Nel lavoro di tesi, sono stati usati due tipi di layer presenti nel modulo torch.nn: reti lineari, anche dette *fully connected*, e convoluzionali. La rete lineare si applica agli input in maniera analoga a quanto abbiamo definto in precedenza, attraverso i weight e i bias. Le reti convoluzionali vengono invece utilizzate per il rico-

noscimento di immagini al fine di rendere l'apprendimento "invariante sotto traslazioni" delle immagini, non fissandosi dunque sulle caratteristiche specifiche di un'immagine ma estrapolando le caratteristiche generali di essa, in particolare le relazioni tra pixel vicini tra di loro. Gli argomenti per definire ciascuna delle layer sono il numero dei suoi elementi in ingresso e in uscita, oltre ad altri più specifici.
Inoltre, all'interno della nostra rete possono esser usati altri strumenti per rendere l'apprendimento più efficace: ne sono un esempio il Maxpooling, che sottocampiona un'immagine per estrapolarne le caratteristiche generali, oppure il Dropout che permette di deattivare alcuni dei neuroni, scelti in maniera casuale, per ridurre l'overfitting.
Al fine di rendere il modello più chiaro è inoltre possibile dividere "l'architettura" della rete neurale dalla sua parte funzionale, identificata come F nella figura 1.5.

```python
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.dropout = nn.Dropout2d()
        self.fc1 = nn.Linear(256, 64)
        self.fc2 = nn.Linear(64,2)

    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)),2)
        x = F.max_pool2d(F.relu(self.conv2(x)),2)
        x = self.dropout(x)
        x = x.view(1,-1) #reshaping tensor
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        x = nn.Softmax(dim=1)

        return x
```

Figura 1.5: Esempio dell'implementazione di una Rete Neurale come una classe di python

Nel calcolo dell'output della rete neurale, si vuole ottenere una risultato che rappresenti la probabilità che l'immagine data in input appartenga ad una certa classe (per esempio un particolare numero o oggetto). Vogliamo quindi che ogni valore sia compreso tra 0 e 1 e che la somma totale di tutti gli output della rete dia 1. Viene aggiunta alla fine della rete neurale una funzione, *softmax*, che permetta di ottenere queste caratteristiche per l'output. Si stima ora l'accuratezza del riconoscimento, confrontando numericamente questo valore di probabilità con quello corretto (0 o 1) per ciascuna delle classi a partire dalla loss function.

1.2 I Computer Quantistici

Technologies based on the laws of quantum mechanics, which govern physics on an atomic scale, will lead to a wave of new technologies that will create many new businesses and help solve many of today's global challenges.

Nel 2018 la Commissione europea ha avviato il programma decennale di ricerca e sviluppo *Flagship in Quantum Technologies* con il coinvolgimento di tutti gli stati membri e con ingenti investimenti. Nel documento alla base del programma, il *Quantum Manifesto* [9], da cui è presa la frase riportata sopra, si parla di un'imminente seconda rivoluzione quantistica in quanto la comprensione e l'applicazione della meccanica quantistica ai sistemi microscopici hanno già prodotto una prima rivoluzione tecnologica con l'invenzione, per esempio, del transistor, dell'illuminazione Led e del laser. Viene richiesto di guardare al mondo e alla natura che ci circonda in un modo fondamentalmente nuovo, sfruttando proprio le regole del mondo quantistico: la *sovrapposizione* degli stati, ovvero la possibilità di essere in combinazione di stati diversi, e *l'entanglement*, la possibilità degli stati di essere correlati pur senza un'interazione fisica. Gli obiettivi tecnologici principali sono quattro:

- La comunicazione quantistica
- I simulatori quantistici
- I sensori quantistici
- I computer quantistici

I computer quantistici sfruttano le proprietà quantistiche della materia al fine di effettuare una computazione. Avere un'idea chiara riguardo ai vantaggi, sia in termini di tempo, di memoria e di numero di dati da utilizzare, che si potrebbero ottenere con questo tipo di computer è complesso, sia a livello matematico, in quanto è difficile da dimostrare al momento se la computazione quantistica sia più potente o veloce, e anche praticamente, a causa delle difficoltà incontrate nella costruzione delle macchine, limitate in dimensioni e soggette a rumore e errori. Si stima tuttavia che diversi siano i vantaggi che si potranno ottenere con un computer di questo tipo. Si parla di *Quantum Supremacy* quando un computer quantistico è in grado di offrire un grosso guadagno in termini di tempo su un calcolo rispetto a un normale computer classico. Nel 2019 Google ha annunciato di aver impiegato minuiti a risolvere un problema su un computer quantistico che sul più performante computer classico avrebbe impiegato migliaia di anni.

Figura 1.6: IBM Q System One istallato in un centro IBM vicino a Stoccarda

La computazione su un computer quantistico è basata su tre elementi:

- Dati = Qubits
- Operazioni = Quantum gates
- Risultati = Misurazioni

1.2.1 Il Qubit e la misura dello stato

Uno dei principali vantaggi di un computer quantistico è la possibilità dei costituenti che codificano l'informazione di trovarsi in una sovrapposizione di stati [15].
Come nella computazione classica il bit si trova in uno stato, o 0 o 1, anche il *qubit*, il bit quantistico, si trova su uno stato che può essere $|0>$ o $|1>$. Questo tipo di notazione viene definita come *notazione di Dirac*, dove $|$ corrisponde al *bra* e $>$ corrisponde al *ket*. La peculiarità del qubit è la possibilità di trovarsi su uno stato costituito dalla combinazione lineare di questi, proprietà nota in meccanica quantistica come *sovrapposizione* o *superposition*:

$$|\psi>= \alpha|0> +\beta|1> \tag{1.6}$$

dove α e β sono due numeri complessi. Si può dunque interpretare il qubit come un vettore in uno spazio vettoriale complesso (*Spazio di Hilbert*) con base di due elementi,

ovvero ($|0>, |1>$) nota come *base computazionale*. I due parametri α e β devono soddisfare la condizione di normalizzazione:

$$|\alpha|^2 + |\beta|^2 = 1. \tag{1.7}$$

Lo stato quantistico si può trovare in un continuo insieme di valori, non in un numero discreto come nel caso del bit classico. Ma è bene esaminare la situazione con maggiore attenzione: questo non significa che effettuando la misura sul qubit finale si può ottenere un qualunque valore compreso tra 0 e 1. Il risultato di una misurazione sarà o 0 o 1 come nel caso del bit classico. La differenza sta nel fatto che non si può predire con certezza quale stato otteremo nonostante le condizioni iniziali siano note. Entra dunque in gioco un aspetto statistico che è *intrinseco* nella teoria: non si possono migliorare le condizioni sperimentali al fine di eliminare questa incertezza. L'output della misura sarà 0 o 1 con una certa probabilità e se si ripete la misura con le stesse condizioni sperimentali si potrebbe non ottenere lo stesso risultato. Si può copiare il procedimento che ha permesso di ottenere quel particolare qubit ma non il qubit stesso (condizione imposta dal *no-cloning theorem*). La probabilità di ottenere un risultato piuttosto che un altro è data dai coefficienti posti davanti ai singoli stati, ovvero α e β, il quale modulo quadro rappresenta appunto la probabilità di trovare il qubit in quel specifico stato in seguito alla misurazione. In meccanica quantistica si parla di *collasso della funzione d'onda* per riferirsi alla misurazione, *immediatamente* dopo la quale il nostro qubit o stato quantistico non si troverà più in sovrapposizione ma sarà su uno stato definito con il 100% di probabilità [16].
In particolare, effettuare una misurazione è l'unico modo attraverso il quale possiamo conoscere il valore di un qubit. L'effetto della misura sullo stato quantistico è quello di creare una *perturbazione* che lo modifichi. Ogni osservabile fisica viene definita a partire da un operatore Hermitiano e i possibili risultati che si possono ottenere da una misurazione su uno stato sono dati dagli *autovalori* dell'operatore associato a quell'osservabile fisica.

Un qubit può essere rappresentato in maniera visiva utilizzando la *sfera di Bloch* (figura 1.7), ovvero una superficie sferica in tre dimensioni di raggio unitario. Si può infatti passare in coordinate sferiche per i coefficienti degli stati ottenendo:

$$|\psi> = cos\left(\frac{\theta}{2}\right)|0> + e^{i\phi} sen\left(\frac{\theta}{2}\right)|1> . \tag{1.8}$$

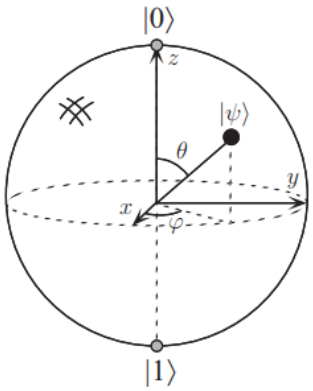


Figura 1.7: Rappresentazione di un qubit su una sfera di Bloch in dipendenza da due parametri θ e ϕ

La situazione può essere estesa al caso di più qubit. Nel caso di due qubit ogni stato si può trovare in $|0>$ o $|1>$ e quindi si hanno complessivamente quattro combinazioni ($|00>, |01>, |10>$ e $|11>$) in cui il sistema complessivo si può trovare e, considerando la sovrapposizione, il generico stato si può scrivere come:

$$|\psi >= \alpha_{00}|00 > +\alpha_{01}|01 > +\alpha_{10}|10 > +\alpha_{11}|11 > . \tag{1.9}$$

dove i coefficienti sono tra loro in relazione secondo la normalizzazione dello stato. Misurando entrambi i qubit si ottiene lo stato $|ab>$ con probabilità $|\alpha_{ab}|^2$ in analogia a quanto visto per un solo qubit ma con quattro possibilità. Se si misura invece un solo qubit, si ottiene lo stato $|a>$ con una probabilità data dalla somma delle probabilità $|\alpha_{a0}|^2$ + $|\alpha_{a1}|^2$. É interessante notare la possibilità di codificare i coefficienti dei singoli stati all'interno di un vettore, notazione che sarà utile per definire le operazioni quantistiche:

$$\begin{bmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{bmatrix}$$

Nel caso di n-qubit la situazione è semplicemente una generalizzazione di quanto fatto per due qubit, dove in questo caso le possibilità sono 2^n. Ogni possibile combinazione viene rappresentata per semplicità da un singolo numero ($|0>$, $|1>$, $|2>$ ecc.). Lo stato generico sarà dunque:

$$|\psi >= \sum_{x=0}^{2^n-1} \alpha_x |x > \tag{1.10}$$

1.2.2 Le operazioni quantistiche: i Quantum Gates

In maniera analoga a quanto si verifica classicamente con le porte logiche, anche nei computer quantistici possono essere introdotte delle operazioni che agiscono e trasformano gli stati: sono note come porte quantistiche o *quantum gates*. Queste operazioni sono matrici unitarie, ovvero tali per cui $UU^+ = U^+U = 1$, che agiscono sullo stato trasformandolo. Una conseguenza importante dell'unitarietà delle matrici è che esiste un'inversa per qualunque operazione che può essere definita, ovvero si ha la condizione di *reversible computing*; ogni gate deve avere lo stesso numero di entrate e di uscite e alcune operazioni classiche, come or, and, xor, non possono essere direttamente implementate nel caso quantistico [15].
L'applicazione di un gate su uno stato, per esempio ad un qubit, sarà dunque il prodotto tra una matrice 2×2 e il vettore colonna rappresentante lo stato, ovvero il vettore contenente i coefficienti α e β riportati nell'equazione 1.6, ottenendo:

$$\begin{bmatrix} \alpha' \\ \beta' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

Per azione su stati a più qubit, la situazione è analoga solo con matrici e vettori di dimensione superiore, ovvero matrici $N \times N$ dove N è il numero di combinazioni possibili degli stati quantistici singoli (ovvero 2^n, con n numero di qubit).

Un modo interessante di interpretare i gate quantistici è quello di utilizzare la sfera di Bloch introdotta nella sezione precedente (figura 1.7). Possiamo infatti definire i gate come delle rotazioni dello stato quantistico nella sfera di Bloch tenendo fisso un asse (x, y o z) e ruotando il sistema attorno ad esso di un angolo θ:

$$R_x(\theta) = \begin{bmatrix} cos\Big(\theta/2\Big) & -isin\Big(\theta/2\Big) \\ -isin\Big(\theta/2\Big) & cos\Big(\theta/2\Big) \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} cos\Big(\theta/2\Big) & -sin\Big(\theta/2\Big) \\ sin\Big(\theta/2\Big) & cos\Big(\theta/2\Big) \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix}$$

La combinazione di queste rotazioni è la forma più generale di un gate su un singolo qubit, da cui è possibile ricavare gli altri gate. Per esempio il gate Z:

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

che agisce su uno stato nella seguente maniera:

$$\begin{aligned} Z|0>&= |0> \\ Z|1>&= -|1> \end{aligned} \tag{1.11}$$

può essere interpretato come una rotazione $R_z(\pi)$. Z appartiene a una particolare categoria di gate noti come *Pauli gates* insieme al gate X (che è il corrispondente quantistico del NOT classico, ovvero che scambia i due stati) e Y, che sono sempre rappresentabili con singole rotazioni sulla sfera di Bloch per particolari angoli. Può essere provato che esistono angoli α, β e γ per cui un'operazione unitaria possa sempre essere fattorizzata come il prodotto dei tre gate di rotazione:

$$U = R_x(\alpha)R_y(\beta)R_z(\gamma). \tag{1.12}$$

Un altro gate interessante per qubit singoli è l'Hadamard gate (H) che permette di ottenere una sovrapposizione equa degli stati quantistici. Può essere interpretato come una rotazione di 180 gradi attorno all'asse x seguita da una rotazione di 90 gradi attorno all'asse y nel sistema definito nella figura 1.7. La sua azione e forma matriciale sono le seguenti:

$$\begin{aligned} H|0>&= \frac{|0>+|1>}{\sqrt{2}} \\ H|1>&= \frac{|0>-|1>}{\sqrt{2}} \\ H &= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \end{aligned} \tag{1.13}$$

Per quanto riguarda i gate che agiscono su un numero maggiore di qubit, il più interessante di tutti è il CNOT (*Controlled-not gate*) in quanto permette di ottenere il fenomeno dell'*entanglement*. Dato un sistema quantistico ($|\psi>$) costruito a partire da due stati $|\psi_1>$ e $|\psi_2>$, essi si definiscono entangled se non è possibile riscrivere lo stato complessivo come il prodotto dei due stati singoli, ovvero:

$$|\psi>= |\psi_1>|\psi_2> . \tag{1.14}$$

Esempi di stati entangled sono i gli stati di *Bell*:

$$\begin{gathered} \frac{|00> \pm |11>}{\sqrt{2}} \\ \frac{|01> \pm |10>}{\sqrt{2}} \end{gathered} \tag{1.15}$$

Si parla in questo caso di stati *correlati*. Effettuando una misura su un singolo qubit anche il secondo qubit sarà definito: se viene effettuata una misura sul primo qubit e si ottiene per esempio $|1>$ la misura sul secondo qubit darà necessariamente risultato $|1>$ nel caso del primo stato di Bell o $|0>$ nel secondo caso.
La matrice corrispondente al CNOT gate è:

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

e agisce in maniera tale che se il primo qubit è $|0>$ non cambia nulla nel sistema ma se è $|1>$ il secondo qubit viene invertito:

$$\begin{aligned} CNOT|00> &= |00> \\ CNOT|01> &= |01> \\ CNOT|10> &= |11> \\ CNOT|11> &= |10> . \end{aligned} \tag{1.16}$$

1.2.3 IBM e Qiskit

Qiskit [8] è un framework open source per la computazione quantistica, che usa il linguaggio di programmazione Python. Questa libreria fornisce metodi per creare e manipolare programmi e algoritmi quantistici e utilizzarli sui devices di *IBM Quantum Experience* [4]; questa è una piattaforma online, sviluppata nel 2016, che permette di collegarsi via cloud ai simulatori e ai computer quantistici messi a disposizione dall'azienda IBM. Per il loro utilizzo sul cloud è sufficiente collegarsi al *backend* e si viene messi in coda attendendo il proprio turno.
Per ciascuno dei sistemi quantistici viene fornito il numero di qubit a disposizione e una proprietà detta *Quantum Volume* [21] che permette di definire le capacità e il *rate* di errore di quel particolare computer quantistico; è un numero che si riferisce alla performance del device in generale prendendo in considerazione diversi aspetti come la connettività tra i diversi qubit, la frazione di errori compiuti e le interazioni incontrollate con il sistema. Questo parametro permette di comparare tra di loro i diversi devices e più è grande il suo valore più sono complessi i problemi che si stima possano essere risolti.

La capacità di un computer di eseguire un algoritmo, che viene presa come metrica per valutare la potenza del computer stesso, dipende dal numero di qubit (N) e dal numero di passaggi necessari, *quantum depth* (d), per risolvere il problema. Nella formula 1.17 viene data una prima definizione di quantum volume, $\tilde{V_Q}$, dove d(N) dipende da ϵ_{eff}, il rate di errore medio per un gate a due qubit.

$$\begin{aligned} \tilde{V_Q} &= min\left[N, d(N)\right]^2 \\ d &\simeq \frac{1}{N\epsilon_{eff}} \end{aligned} \tag{1.17}$$

Bisogna tuttavia prestare attenzione rigurado a quest'ultima definizione di d: dato che dipende anche dal numero di qubit N, il suo valore decresce all'aumentare del numero di qubit con lo stesso rate di errore; non è però detto che tutti gli N qubit siano necessari per eseguire un certo algoritmo. Si può dunque ridefinire in maniera completa il Quantum Volume, V_Q, nel seguente modo:

$$V_Q = max_{n<N}\left\{min\left[n, \frac{1}{n\epsilon_{eff}(n)}\right]^2\right\} \tag{1.18}$$

dove il massimo viene preso su un arbitraria scelta dei qubit tale da massimizzare il V_Q che può essere ottenuto.

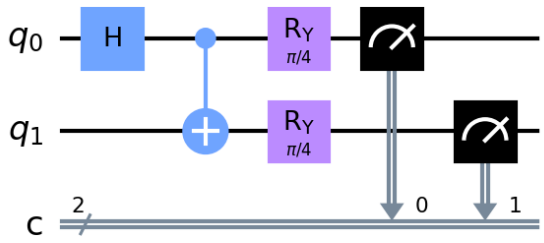
É possibile avere ulteriori informazioni sul backend nel momento in cui si fa eseguire un programma, come riportato in figura 1.8. Oltre ai *Pending jobs* che indicano il numero di programmi in coda davanti al proprio, sono interessanti le proprietà *Avg. T1*, *Relaxation time*, e *Avg. T2*, *Dephasing time*, che indicano rispettivamente il tempo impiegato affinchè un qubit nello stato $|1>$ ritorni nello stato $|0>$ e il tempo in cui uno stato permane in sovrapposizione. Entambi i valori sono legati al concetto di decoerenza e più sono alti questi valori migliori sono le caretteristiche del device, legate principalmente al numero di operazioni utili che possono essere eseguite su di esso.

Oltre ai computer quantistici veri e propri, IBM mette a disposizione una serie di simulatori tramite il componente qiskit.Aer. Nel corso di questa tesi è stato ampiamente utilizzato il simulatore *qasm simulator*.

```
ibmq_manila               ibmq_quito                ibmq_belem
-----------               ----------                ----------
Num. Qubits:  5           Num. Qubits:  5           Num. Qubits:  5
Pending Jobs: 20          Pending Jobs: 7           Pending Jobs: 0
Least busy:   False       Least busy:   False       Least busy:   True
Operational:  True        Operational:  True        Operational:  True
Avg. T1:      127.8       Avg. T1:      93.6        Avg. T1:      82.9
Avg. T2:      63.1        Avg. T2:      60.1        Avg. T2:      98.7
```

Figura 1.8: Informazioni ottenibili riguardo a un computer quantistico con il comando backend_overiew() di Qiskit

Uno o più qubit e tutte le possibili operazioni che vengono eseguite su di essi, anche le eventuali misurazioni, costruiscono un *circuito quantistico*. IBM Quantum Experience mette a disposizione un'interfaccia grafica *IBM Quantum Composer* per costruire i circuiti quantistici e effettuare operazioni su di essi. Di default lo stato iniziale in qiskit è scelto come $|0>$ e la misurazione consiste nel misurare la proiezione del qubit lungo l'asse z, nel sistema di riferimento rappresentato in figura 1.7. In figura 1.9 viene riportato un esempio di un circuito quantistico e il relativo codice implementato su IBM con alcune operazioni (Hadamard gate e Rotazioni attorno all'asse y) e le misurazioni finali.




```qiskit
circuit = QuantumCircuit(2,2)

#creating a Bell pair
circuit.h(0)
circuit.cx(0,1)

#applying rotations
circuit.ry(np.pi/4, 0)
circuit.ry(np.pi/4, 1)

#measuring
circuit.measure(range(2), range(2))

circuit.draw(output='mpl')
```

Figura 1.9: Esempio di un circuito quantistico implementato su IBM Quantum Experience con relativo programma

Capitolo 2

Quantum Machine Learning

Un'importante applicazione dei computer quantistici potrebbe essere proprio il Machine Learning. Infatti, nonostante il Machine Learning abbia risolto numerosi problemi e aiutato a trovare soluzioni approssimate per altri, ha delle limitazioni legate al grande numero di dati in ingresso e il tempo necessari a ottenere una risposta corretta dal modello.
In base alla computazione e ai dati utilizzati, se classici o quantistici, si possono avere diversi sviluppi del Quantum Machine Learning, riportate in figura 2.1.

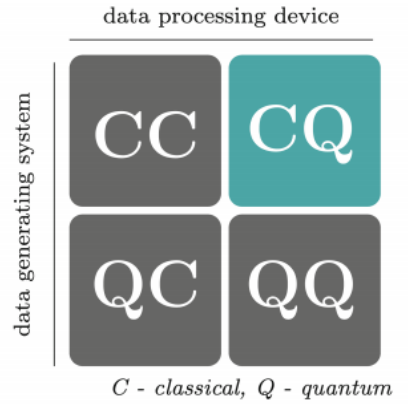


Figura 2.1: Combinazioni tra classico e quantistico in Quantum Machine Learning. Image credits: Supervised Learning with Quantum Computers, Schuld, Petruccione (2018)

Nella maggior parte dei lavori di ricerca nell'ambito del Quantum Machine Learning, si sfrutta il caso di dati in input classici e computazione quantistica (riquadro azzurro nella figura 2.1). Questo è stato utilizzato anche in questo lavoro di tesi.

Si stima che l'utilizzo dei computer quantistici per compiti di Machine Learning possa portare a diversi benefici [22], in particolare:

- Ridurre il tempo impiegato per la computazione, ottenendo più velocemente i risultati.
- Migliorare la capacità di apprendimento, ovvero le capacità associative della macchina.
- Migliorare l'efficienza di apprendimento, richiedendo una minore quantità di informazioni in input o modelli più semplici.

Diversi sono i metodi che possono essere implementati al fine di ottenere i benefici citati sopra: per esempio l'azienda *D-wave* sfrutta i *Quantum Annealers* [11], ovvero processori che naturalmente trovano lo stato a energia più bassa per un sistema fisico, caratteristica che potrebbe essere sfruttata per problemi di ottimizzazione. Numerosi approcci utilizzano algoritmi ibridi, ovvero funzionanti in parte su un computer classico e in parte su un computer quantistico, di cui una delle applicazioni è quella delle *reti neurali ibride* [12], ovvero dei circuiti quantistici parametrizzati soggetti a ottimizzazione per processi di classificazione. Uno degli approcci in questo caso è quello di codificare i dati all'interno di un circuito quantistico come stati iniziali, processo noto come *Quantum embedding* [20], e in eseguito effettuare un processo di ottimizzazione su delle operazioni parametrizzate che agiscono su questi stati.
É bene precisare che attualmente la ricerca nell'ambito del Quantum Machine Learning è ancora alle sue basi; si hanno poche risposte riguardo a quali problemi possano essere risolti in maniera più efficiente su un computer quantistico o ibrido [15]. É comunque interessante proseguire la ricerca in questo ambito viste le prospettive.

2.1 Le Reti Neurali Ibride

In questo progetto di tesi sono state utilizzate delle reti neurali ibride in cui, all'interno di un modello classico di Deep Learning, viene inserita una layer costituita da un circuito quantistico parametrizzato [3].
Le librerie utilizzate all'interno di questo programma sono Qiskit per la computazione quantistica e Pytorch per la parte relativa al Machine Learning e l'apprendimento della macchina è basato sul riconoscimento di digit del dataset MNIST (figura 1.4).
Si spiega ora come è stato possibile inserire un circuito quantistico all'interno di una rete neurale classica. Nelle diverse layer nascoste che costituiscono la rete, dove i dati in input vengono codificati in maniera analoga a quanto visto nel primo capitolo, viene inserito un circuito quantistico parametrizzato, ovvero i cui gate dipendono da alcuni parametri, che saranno gli output della layer classica precedente. Nel sezione 1.2.2 si è evidenziato come ogni gate quantistico che agisce su un qubit possa essere fattorizzato da rotazioni

dello stesso nella sfera di Bloch. I parametri a cui si fa riferimento definiscono gli angoli di rotazione per i gate. In seguito la misura statistica sul nostro circuito viene utilizzata o come input per la layer successiva oppure come risultato della rete da confrontare con i risultati attesi. Con *misurazione statistica* si intende che la misurazione sul qubit viene effettuata diverse volte e poi mediata, al fine di ottenere il risultato probabilistico del circuito, ovvero quante volte si è ottenuto il risultato $|0>$ o $|1>$ nel caso di un qubit. Il procedimento è rappresentato schematicamente nella figura 2.2.

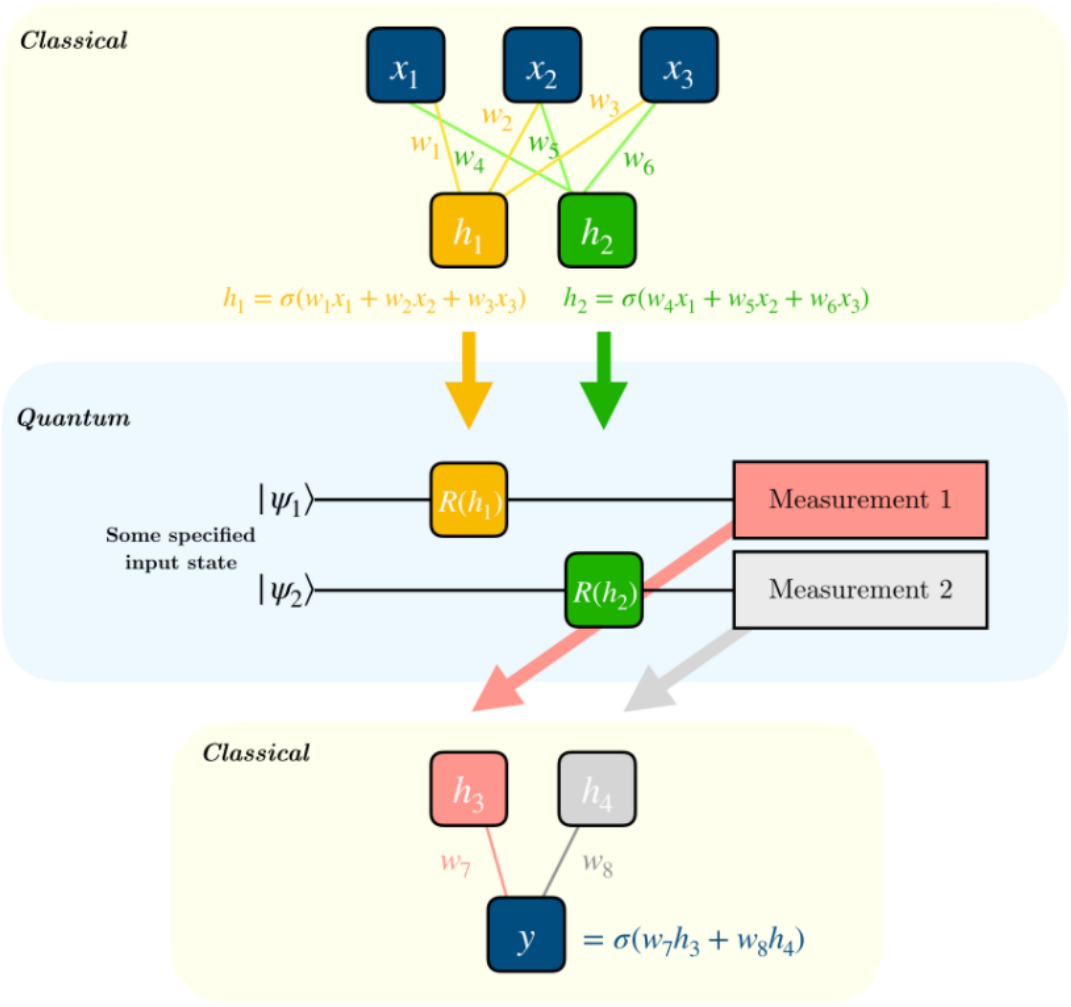


Figura 2.2: Rappresentazione schematica del funzionamento di una rete neurale ibrida [3]

I parametri inseriti all'interno del circuito diventano a questo punto dei parametri da ottimizzare al fine di ottenere il risultato voluto, e quindi saranno anch'essi soggetti a un processo di *backpropagation* il cui calcolo avviene tramite un procedimento noto come *parameter shift rule*. Si analizzano in maggior dettaglio i metodi utilizzati nel programma per includere un circuito quantistico di questo tipo nelle sezioni successive. Inoltre è riportato un esempio del codice utilizzato per un particolare esperimento in appendice A.

2.1.1 Inserimento di un circuito quantistico come classe di Python

Il circuito quantistico viene implementato come una classe di Python sfruttando dunque il paradigma di *Programmazione Orientata ad Oggetti* per questo linguaggio [18]. In particolare all'interno del costruttore della classe vengono inserite le caratteristiche di architettura che costituiscono in circuito, ovvero la definizione dei parametri e le varie operazioni che devono essere eseguite sul qubit. Il circuito può essere implementato in diversi modi, consentendo ad esempio la libertà di definire il numero di qubit o i vari gate, che eventualmente consentano anche l'interazione tra più qubit, come il CNOT gate. Tra queste operazioni vengono incluse delle rotazioni (R_x, R_y o R_z) come le abbiamo definite nella sezione 1.2.2 con angoli di rotazione parametrizzati, in base alla definizione precedente.
In seguito è stato implementato in questa classe un metodo, *run*, al fine di sostituire i parametri con dei valori numerici, che nel caso della rete neurale vengono passati come output dalla layer precedente, e effettuare le misurazioni sul circuito. Si è già visto il significato di effettuare una misura di un qubit nella sezione 1.2.1: si ottiene con una certa probabilità o il risultato $|0>$ o $|1>$, nel caso di uno spazio di Hilbert in due dimensioni. Questa operazione viene ripetuta automaticamente da qiskit un numero fissato di volte (*shots*) al fine di dare validità statistica al risultato, ottenendo la frequenza con cui si è ottenuto ciascuno degli stati. Si riporta un esempio dell'implementazione della classe QuantumCircuit in figura 2.3.

Si può inoltre scegliere il valore da far ritornare dal nostro circuito; per esempio il valore di aspettazione della misura per circuiti ad un qubit, ovvero:

$$Exp = n_0 0 + n_1 1 \tag{2.1}$$

dove n_0 e n_1 rappresentano il numero delle volte in cui abbiamo ottenuto come risultato $|0>$ o $|1>$. Oppure si possono ottenere le probabilità relative a ogni combinazione possibile dei risultati (per esempio $|00>, |01>, |10>$ e $|11>$ nel caso di 2 qubit).

```python
class QuantumCircuit:
    """
    This class provides an interface to interact with our Quantum Circuit
    """

    def __init__(self, n_qubits, backend, shots):

        #-----Circuit definition
        self._circuit = qiskit.QuantumCircuit(n_qubits)
        self.parameters = qiskit.circuit.ParameterVector('parameters', 3)

        all_qubits = [i for i in range (n_qubits)]

        self._circuit.h(all_qubits)
        self._circuit.barrier()
        self._circuit.u(self.parameters[0], self.parameters[1], self.parameters[2], all_qubits)
        self._circuit.barrier()
        self._circuit.measure_all()
        #-----

        self.backend = backend
        self.shots = shots
        self.n_qubits = n_qubits

    def expectation_Z(self,counts, shots, n_qubits):
        expects = np.zeros(n_qubits)
        for key in counts.keys():
            percentage = counts[key]/shots
            check = np.array([(float(key[i]))*percentage for i in range(n_qubits)])
            expects += check
        return expects


    def run(self, thetas):
        #acting on a simulator
        thetas = thetas.squeeze()
        p_circuit = self._circuit.bind_parameters({self.parameters[k]: thetas[k].item() for k in range(len(thetas))})
        job_sim = qiskit.execute(p_circuit,
                         self.backend,
                         shots=self.shots)

        result = job_sim.result()
        counts = result.get_counts(p_circuit)

        return self.expectation_Z(counts, self.shots, self.n_qubits)
```

Figura 2.3: Esempio del programma implementato per la definizione della classe Quantum Circuit

2.1.2 Implementazione delle funzioni forward e backward

Le funzioni necessarie per l'algoritmo di backpropagation vengono implementate usando Pytorch ed estendendo *torch.autograd* [2]. Questo significa implementare una nuova sottoclasse di *Function*, che autograd utilizza al fine di calcolare i risultati e i gradienti e tenere traccia delle varie operazioni eseguite, per i metodi forward(), che effettua l'operazione di calcolo, e backward(), che calcola il gradiente.
La backward nel caso di un circuito quantistico viene implementata tramite un procedimento noto come *parameter shift rule* [13], con cui si calcola il gradiente delle operazioni quantistiche rispetto ai parametri. Si sceglie un fattore fissato (*shift*, $\vec{s}$), delle stesse dimensioni del vettore dei nostri parametri ($\vec{\theta}$), e si calcola:

$$\vec{\theta_{\pm}} = \vec{\theta} \pm \vec{s}. \tag{2.2}$$

Con questo nuovo set di parametri ottenuti si rieseguono le operazioni quantistiche, l'insieme delle quali verrà denominata con QC, e si definisce il gradiente del circuito nel seguente modo:

$$\nabla_{\vec{\theta}} \left[QC(\vec{\theta}) \right] = QC\left(\vec{\theta_{+}}\right) - QC\left(\vec{\theta_{-}}\right). \tag{2.3}$$

Si definisce a questo punto una layer quantistica come estensione del modulo nn di Pytorch di cui abbiamo parlato nella sezione 1.1.3, riportando le due funzioni definite sopra e il circuito; la layer viene poi inserita in una rete neurale, prestando attenzione alla compatibilità tra la rete e il circuito in termini di dimensione dei parametri passati in quest'ultimo.

In figura 2.4 viene riportato un esempio dell'implementazione della rete risultante, che non è altro che l'analogo della figura 1.5 a cui viene aggiunta la layer quantistica finale. Nella figura, Hybrid corrisponde alla layer quantistica definita precedentemente, che prende in argomento il backend su cui far agire il computer quantistico (in questo caso un simulatore), il numero di volte in cui dobbiamo far avvenire la misurazione (n_shots) e il fattore fissato (shift) necessario per il calcolo dell'algoritmo parameter shift rule. Il circuito che era stato definito in questo caso era costituito da un singolo qubit e dava come risultato il valore di aspettazione della misura denotato con x, come definito nell'equazione 2.1. Il risultato della rete neurale, che viene poi confrontato con quello atteso, dava come risultato un tensore costituito da (x, 1-x) quindi due valori per un riconoscimento binario.

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.dropout = nn.Dropout2d()
        self.fc1 = nn.Linear(256, 64)
        self.fc2 = nn.Linear(64,1)
        self.hybrid = Hybrid(qiskit.Aer.get_backend('qasm_simulator'), n_shots, shift )

    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)),2)
        x = F.max_pool2d(F.relu(self.conv2(x)),2)
        x = self.dropout(x)
        x = x.view(1,-1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        x = self.hybrid(x)
        return torch.cat((x, 1-x),-1)
```

Figura 2.4: Implementazione classe Net contenente le varie caratteristiche della rete neurale con introduzione di una layer quantistica (self.hybrid)

2.2 Esperimenti

Nel lavoro di tesi sono stati condotti diversi esperimenti sul circuito modificandolo, introducendo un numero maggiore di qubit e l'entanglement. Inoltre i programmi sono stati eseguiti sia sui simulatori quantistici sia su diversi backend quantistici, forniti da IBM Quantum Experience.
Si è inoltre modificato il set di dati a disposizione sia per l'apprendimento che per la validazione, aggiungendo delle sorgenti di errore gaussiano, per testare il modello ottenuto dall'apprendimento.
Si descrive nella sezione successiva il processo con cui si è ottenuto il rumore gaussiano e in seguito i vari esperimenti che sono stati effettuati modificando le condizioni del circuito quantistico.

2.2.1 Errore gaussiano

Come descritto nella sezione 1.1.4, un'immagine viene codificata in un tensore di Pytorch al fine di effettuare una classificazione e di utilizzare le informazioni relative all'immagine. Aggiugere dell'errore ad un'immagine significa aggiungere al nostro tensore una specifica funzione di distribuzione che nel caso gaussiano è rappresentata da una distribuzione normale [24]. Questo tipo di errore può sorgere durante l'acquisizione dell'immagine per diversi motivi, come poca illuminazione, alte temperature o trasmissioni in circuiti elettronici. Nel nostro specifico caso è stata implementata una funzione del tipo:

$$\text{tensor} + \text{torch.randn}(\text{tensor.size}()) \cdot \text{std} + \text{mean}$$

dove tensor rappresenta il tensore iniziale che codifica l'immagine, torch.randn è una funzione di torch che crea un tensore, di dimensione definita, riempito di numeri casuali secondo una distribuzione gaussiana; questo viene moltiplicato per la deviazione standard impostata precedentemente. Si riporta un esempio dei risultati ottenuti a partire da questo tipo di trasformazione sui digit in figura 2.5.

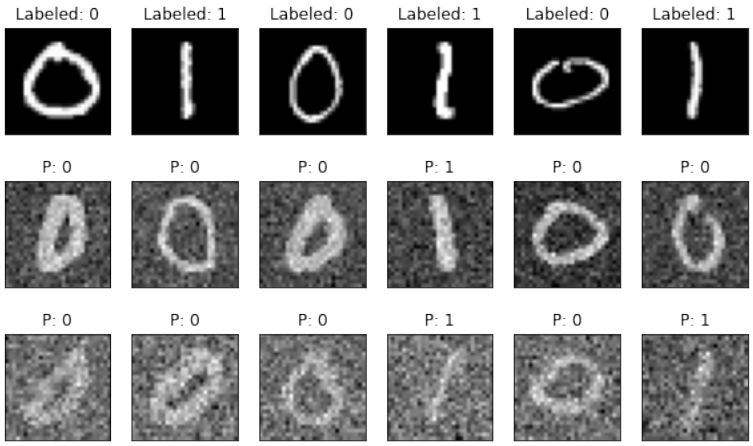


Figura 2.5: Rappresentazione grafica dei digit trasformati aggiungendo una sorgente di rumore gaussiano, rispettivamente con deviazione standard pari a 0, 0.2 e 0.5

2.2.2 Circuito con rotazione 3d

In questo caso si è implementato il circuito introducendo il gate U3 dipendente da tre angoli di Eulero, θ, ϕ e λ. Questa scelta è stata dettata dal fatto che ogni gate quantistico che agisce su un singolo qubit può essere rappresentato da questa operazione, in base a quanto riportato nella sezione 1.2.2. La forma matriciale del gate U3 è la seguente:

$$U3(\theta, \phi, \lambda) = \begin{bmatrix} cos(\theta/2) & -e^{i\lambda}sin(\theta/2) \\ e^{i\theta} & e^{i\phi+i\lambda}cos(\theta/2) \end{bmatrix}$$

In figura 2.6 è riportato lo schema del circuito utilizzato in questo caso, costituito da un Hadamard gate per creare la sovrapposizione degli stati $|0>$ e $|1>$, il gate U3 e infine la misurazione.

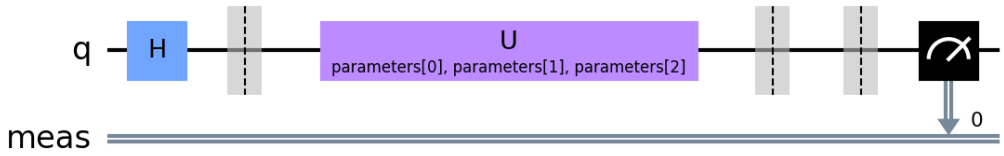


Figura 2.6: Circuito implementato con gate U3

Si riportano in tabella 2.2.2 i parametri scelti per questo esperimento: il learning rate, lo shift usato sul circuito quantistico, l'optimizer e la Loss function forniti dalla libreria Pytorch, il numero di misure che ha permesso di ottenere una statistica sul risultato quantistico (Shots) e le dimensioni del set di dati di training e di validation:

Learning rate	Shift	Optimizer	Loss function	Shots	N training	N validation
0.01	$\pi/2$	SGD	NLLLoss	1000	100	2000

I backend che sono stati utilizzati sono qasm_simulator per il simulatore quantistico, ibmq_athens e ibmq_belem per i devices quantistici.

In questo caso l'output del circuito quantistico è stato scelto come il valore di aspettazione della misura sul qubit, come riportato nell'equazione 2.1, denotato con x. Il circuito quantistico rappresenta l'ultima layer della rete e per avere un valore binario da confrontare con i risultati attesi si è scelto di avere come output della rete neurale (x, 1-x).
Questo tipo di modello ha permesso di ottenere buoni risultati sia sul simulatore che sul circuito quantistico. L'accuratezza dei risultati sul set di validation è risultata del 99.9% in tutti e tre i casi. Si riporta in figura 2.7 la rappresentazione della convergenza dell'apprendimento sul simulatore quantistico e su ibmq_athens, ovvero il valore della

loss function al variare delle epoche. In figura 2.8 è riportata la sovrapposizione dei grafici. Si può notare come non ci sia una rilevante differenza tra l'esperimento eseguito sul simulatore quantistico e sul computer: le due curve si sovrappongono molto bene.

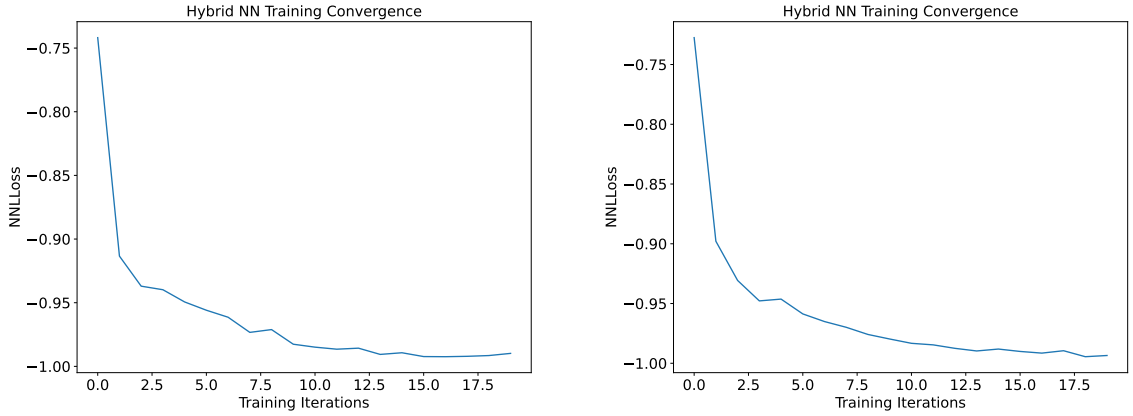


Figura 2.7: Convergenza del training loop per circuito con gate U3 rispettivamente sul qasm_simulator e sul backend ibmq_athens

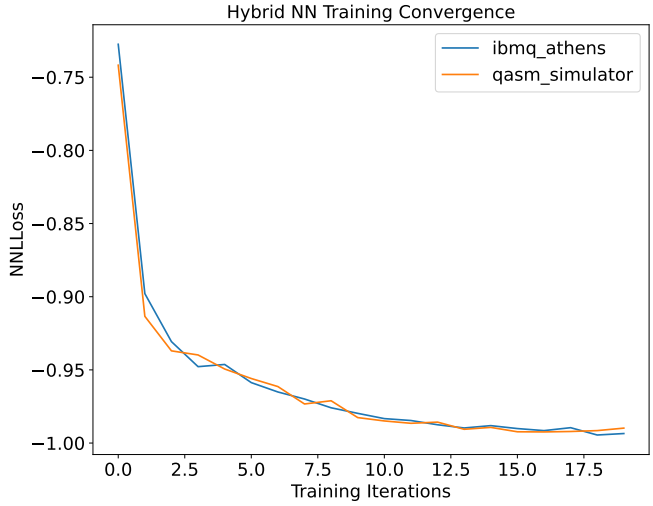


Figura 2.8: Sovrapposizione delle due curve di convergenza per il circuito U3

Si riporta in figura 2.9 la rappresentazione di un set di digit riconosciuti dalla macchina, dove "Predicted" sono le previsione effettuate dalla macchina sui digit in esame.

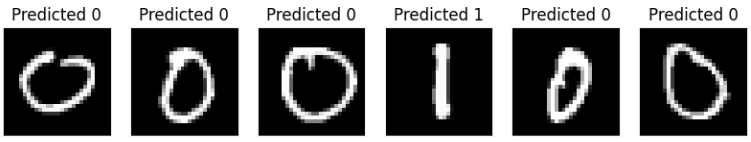


Figura 2.9: Previsioni effettuate dalla macchina con circuito quantistico con gate U3

Si è in seguito testato il modello anche per dati di validazione rumorosi, aumentando di volta in volta la deviazione standard associata, per testare l'efficacia dell'apprendimento della macchina e per constatare che non si fossero verificati fenomeni di overfitting. Si ottengono i risultati riportati in tabella 2.1 per il modello ottenuto sul backend quantistico athens:

Standard deviation	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Accuracy	99.9%	99.9%	99.9%	99.6%	99.5%	99.1%	98.9%	97.7%

Tabella 2.1

Il modello è in grado di riconoscere immagini rumorose in maniera corretta al 99% fino a un valore di 0.7 come deviazione standard.

Si è in seguito utilizzato un training set rumoroso con standard deviation pari a 0.3. Il modello in questo caso riesce a riconoscere i dati rumorosi nel validation set con un'accuratezza superiore al 99% fino a deviazione standard pari a 0.9, come mostrato in tabella 2.2.

Standard deviation	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
Accuracy	99.8%	99.8%	99.7%	99.7%	99.7%	99.4%	99.1%	98.9%

Tabella 2.2

Si riportano in figura 2.10 i grafici relativi all'accuratezza per il riconoscimento dei dati al variare della deviazione standard, sia con dati di apprendimento non rumorosi che con dati di apprendimento rumorosi.

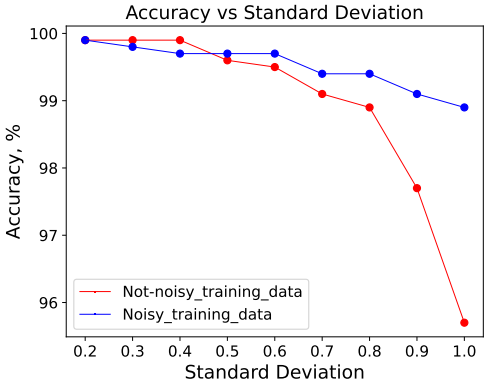


Figura 2.10: Accuratezza al variare della deviazione standard del set di dati di riconoscimento con dati di apprendimento non rumorosi e rumorosi (0.3)

2.2.3 Circuito con più qubit

Si è in seguito provato ad aumentare il numero di qubit a disposizione: il numero di qubit poteva essere scelto in maniera arbitraria e a ciascuno di essi veniva imposta una rotazione parametrizzata R_y in seguito ad un Hadamard gate. Il numero di parametri in questo caso era quindi uguale al numero di qubit. Il circuito è rappresentato in figura 2.11. L'output del circuito è la frequenza con cui si ottengono specifiche combinazioni (per esempio il numero delle volte che si ottiene |00>, |01>, |10> e |11> per due qubit). Al contrario del caso precedente, il circuito quantistico non era l'ultima layer della rete ma è stata aggiunta un'ulteriore layer classica (nn.Linear) per ridimensionare l'output al numero di digit da riconoscere.

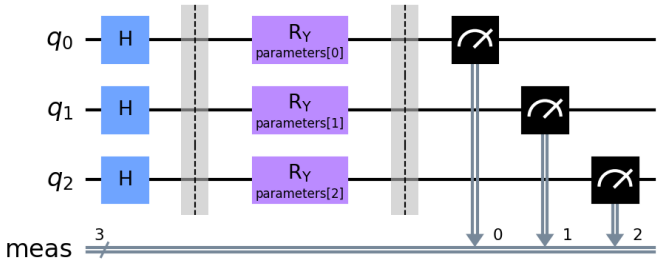


Figura 2.11: Circuito con più qubits e rotazioni R_y

Si riportano in tabella 2.3 i parametri scelti per far verificare l'apprendimento.

I backend uilizzati sono stati qasm_simulator come simulatore e ibmq_athens come computer quantistico.

Learning rate	Shift	Optimizer	Loss function	Shots	N training	N validation
0.01	$\pi/2$	Adam	CrossEntropyLoss	1000	100	2000

Tabella 2.3

Si è ottenuta un'accuratezza sul validation set pari al 99.7% sul simulatore e pari a 99.6% sul circuito quantistico e si riporta in figura 2.12 la convergenza della loss function sui backend. In figura 2.13 invece è riportata la sovrapposizione delle due e come nel caso precedente non si sono riscontrate grosse differenze tra l'utilizzo dei due backend.

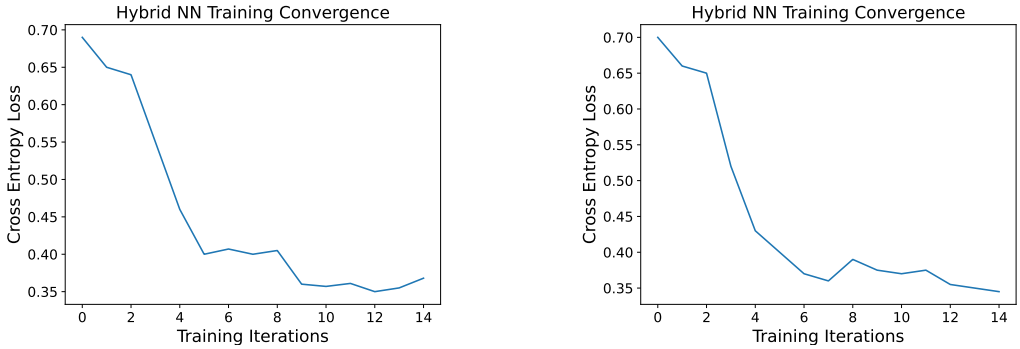


Figura 2.12: Convergenza del training loop per circuito con più qubit rispettivamente sul qasm_simulator e sul backend ibmq_athens

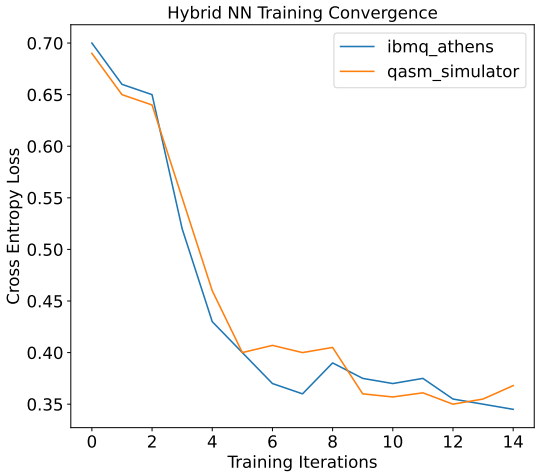


Figura 2.13: Sovrapposizione delle due curve di convergenza per il circuito a più qubit

Anche in questo caso si è testato il modello ottenuto con set di dati di validazione rumorosi e i risultati sono riportati in tabella 2.4.

Standard deviation	0.1	0.2	0.3
Accuracy	99.4%	93.0%	73.5%

Tabella 2.4

Si nota come il modello sia ancora in grado di riconoscere i dati in maniera efficace solo fino a un valore limitato di deviazione standard, 0.2 per avere un riconoscimento sopra al 90%.

Si è provato a rafforzare il modello con un set di dati per il training rumoroso con deviazione standard pari a 0.2, i cui risultati sono riportati in tabella 2.5. In questo caso è stato necessario diminuire il valore del learning rate a 0.008 al fine di avere una corretta convergenza. I risultati indicano un miglioramento nel riconoscimento che risulta sopra al 90% fino a un valore di deviazione standard pari a 0.6.

Standard deviation	0.2	0.3	0.4	0.5	0.6	0.7
Accuracy	99.1%	98.6%	98.0%	96.8%	94.2%	89.7%

Tabella 2.5

L'andamento dell'accuratezza per il riconoscimento dei dati rumorosi al variare della deviazione standard è riportata in figura 2.14.

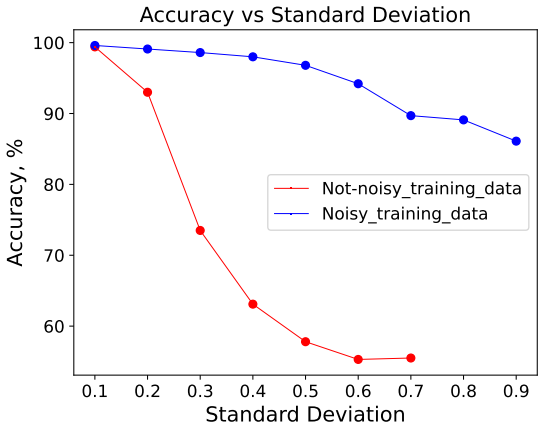


Figura 2.14: Accuratezza al variare della deviazione standard del set di dati di riconoscimento con dati di apprendimento non rumorosi e rumorosi (0.2)

Si è infine provato ad utilizzare questo modello al fine di riconoscere più digit con l'obiettivo di riuscire a riconoscere i 10 digit MNIST. In questo caso il modello, provato solo sul simulatore quantistico qasm_simulator, non ha permesso un riconoscimento efficace nonostante ci sia un processo di apprendimento con la riduzione della loss function. Si ritiene comunque che non siano stati effettuati esperimenti in maniera esaustiva. Vengono riportati i risultati ottenuti per completezza con la discesa della loss_function in figura 2.15. Si è ottenuta un'accuratezza del 95.5% per digit 02 e di 87.8% per digit 04. Si pensa di poter riprovare questo tipo di riconoscimento, implementando le caratteristiche del circuito, provando a modificare i parametri per l'apprendimento e aumentando il set di dati di training.

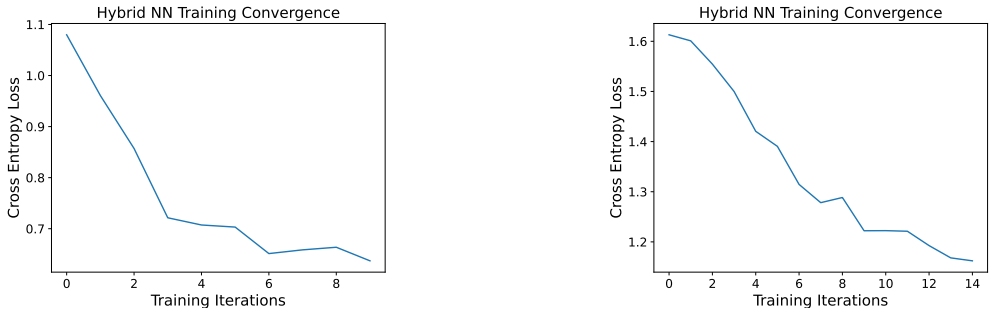


Figura 2.15: Decrescita della loss function per riconoscimento di più digit, rispettivamente riconoscimento di digit 0-2 e 0-4

2.2.4 Circuito con entanglement

Si è provato ad implementare dei circuiti che utilizzassero l'entanglement per vedere come si comportassero i qubit se in correlazione, mentre nei casi precedenti i qubit erano isolati e non interagivano tra di loro. In particolare è stato implementato il circuito riportato in figura 2.16, costituito da un Hadamard gate applicato al primo dei qubit e dei CNOT gate tra le coppie di qubit (01 e 12). In seguito vengono effettuate delle rotazioni R_z dipendenti dai parametri inseriti su ogni stato. L'output del circuito è il valore di aspettazione sul primo dei qubit, come definito nell'equazione 2.1.

Si riportano in tabella 2.6 i parametri utilizzati per l'apprendimento:

Learning rate	Shift	Optimizer	Loss function	Shots	N training	N validation
0.01	$\pi/2$	Adam	Cross Entropy Loss	1000	100	1000

Tabella 2.6

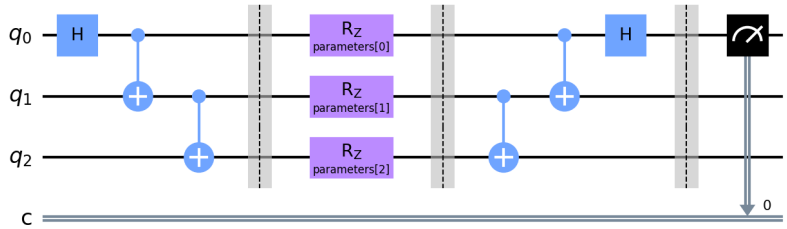


Figura 2.16: Circuito implementato con CNOT gate

I backend utilizzati sono stati: qasm_simulator e ibmq_santiago come device quantistico.

Si riporta in figura 2.17 la convergenza della Loss function per il programma eseguito sul computer quantistico e la curva sovrapposta a quella del simulatore e in figura 2.18 le previsioni fatte dal backend quantistico su un campione del set di dati di validation.

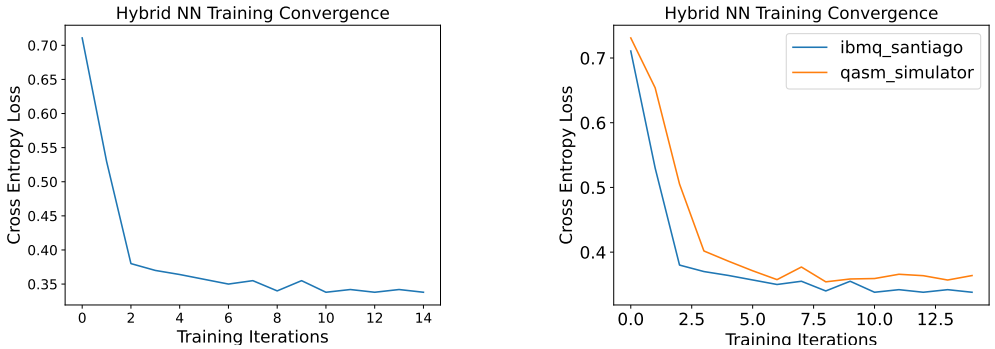


Figura 2.17: Convergenza del training loop per qubit entangled sul backend ibmq_santiago e sovrapposizione tra questa e la convergenza ottenuta sul simulatore quantistico

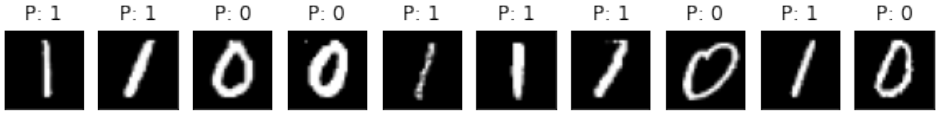


Figura 2.18: Previsioni del modello sul set di dati di validazione

Sul set di dati di validazione si è ottenuto in questo caso un'accuratezza del 99.8% e del 99.7% rispettivamete sul simulatore e sul backend quantistico.
Si riportano i risultati per la validazione del modello su set di dati rumorosi in tabella 2.7, con diversi valori di deviazione standard.

Standard deviation	0.1	0.2	0.3	0.4	0.5	0.6
Accuracy	99.6%	99.3%	98.3%	94.2%	87.0%	74.8%

Tabella 2.7

Il riconoscimento di dati rumorosi con un set di dati di training puliti ha permesso di ottenere risultati con accuratezza superiore al 90% fino a 0.5 come valore di deviazione standard. Si è provato a rafforzare l'apprendimento con un set di dati di training rumorosi con deviazione standard pari a 0.1. In questo caso è stato necessario diminuire il valore di learning rate a 0.008 al fine di ottenere una corretta convergenza. I risultati sul riconoscimento per il backend quantistico sono riportati nella tabella 2.8.

Std	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Accuracy	99.7%	99.7%	99.6%	99.6%	99.3%	98.8%	97.8%	97.2%	95.5%

Tabella 2.8

Anche in questo caso il riconoscimento è migliorato, andamento riportato in figura 2.19, arrivando sopra al 99% fino a 0.5 come valore di deviazione standard; si suppone dunque che il modello sia stato rafforzato e sia meno soggetto a possibili fenomeni di overfitting.

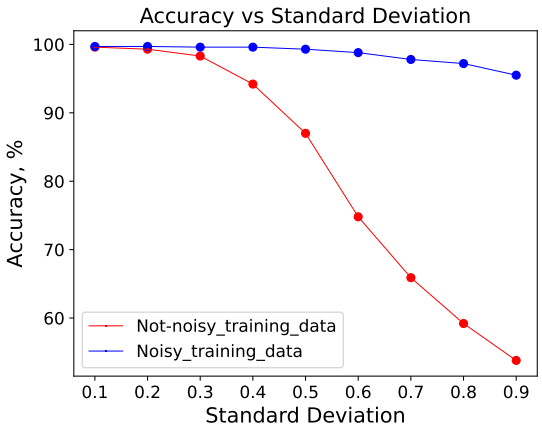


Figura 2.19: Accuratezza al variare della deviazione standard del set di dati di riconoscimento con dati di apprendimento non rumorosi e rumorosi (0.1)

2.2.5 Circuito ispirato al QAOA

Il *Quantum Approximate Optimization Algorithm* è un algoritmo ibrido proprosto da Fahri, Goldstone e Gutmann nel 2014 [14] per ottenere soluzioni approssimate per problemi di ottimizzazione. In questo caso la cost function da minimizzare può essere codificata in un'Hamiltoniana e trovare il minimo della funzione vuol dire trovare l'energia dello stato fondamentale del sistema. Si vuole infatti minimizzare:

$$C(x) = \sum_a w_a C_a(x) \tag{2.4}$$

dove x è una stringa di bit, w_a sono pesi reali e C_a è una funzione booleana. Per ogni C_a è possibile trovare un'Hamiltoniana H_a che possa essere riscritta a partire da una combinazione di operazioni R_z (Z) [15]:

$$H = a_0 I + \sum_i a_i Z_i + \sum_{i,j} a_{ij} Z_i Z_j + ... \tag{2.5}$$

É inoltre possibile mostrare che un'approssimazione dell'evoluzione del sistema possa essere ottenuta con uno stato parametrizzato nella forma:

$$|\beta, \gamma >= e^{-i\beta_p H_i} e^{-i\gamma_p H_f} ... e^{-i\beta_1 H_i} e^{-i\gamma_1 H_f} |s> . \tag{2.6}$$

dove $|s >= \sum_{i=0}^{2^n-1} |x>$ (che può essere riprodotto con un Hadamard gate) e $p >= 1$. Si cercano i migliori parametri β e γ per trovare la soluzione del problema, ovvero minimizzare l'Hamiltoniana.

L'operazione che può rappresentare questo algoritmo è anche nota come *Ising coupling gate* e la forma matriciale è riportata in seguito:

$$I(\alpha) = \begin{bmatrix} e^{i\beta} & 0 & 0 & 0 \\ 0 & e^{-i\beta} & 0 & 0 \\ 0 & 0 & e^{-i\beta} & 0 \\ 0 & 0 & 0 & e^{-i\beta} \end{bmatrix}$$

Il circuito corrispondente è costituito da due qubit messi in entanglement con un CNOT gate, seguito da un rotazione $R_z(\beta)$ dipendente da un parametro e un ulteriore CNOT gate. In seguito la dipendenza dal secondo parametro γ viene esplicitata da una rotazione $R_x(\gamma)$ applicata a ciascuno dei qubit. Il circuito è rappresentato in figura 2.20.

I parametri di questo circuito sono quindi due: uno per la rotazione R_z centrale ai due CNOT gate e uno per entrambe le rotazioni R_x sui due qubit che si effettuano prima della misurazione. Gli output per il circuito quantistico sono le frequenze per ciascuna delle combinazioni, in maniera analoga a quanto definito per il circuito a più qubit.

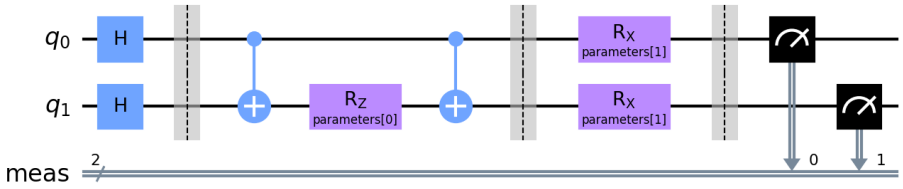


Figura 2.20: Circuito ispirato dall'algoritmo QAOA

In tabella 2.2.5 sono riportati i parametri utilizzati nell'esperimento:

Learning rate	Shift	Optimizer	Loss function	Shots	N training	N validation
0.01-0.008	0.9	Adam	CrossEntropyLoss	1000	100	1000

Il programma è stato eseguito sul simulatore qasm_simulator con un learning rate pari a 0.01 e sul device quantistico ibmq_santiago con un learning rate inferiore, 0.008. In figura 2.21 è riportata la convergenza della loss function nei due casi e in figura 2.22 la sovrapposizione delle due. Questo modello ha dato un'accuratezza del 99.4% sul simulatore quantistico e del 98.9% sul device reale.

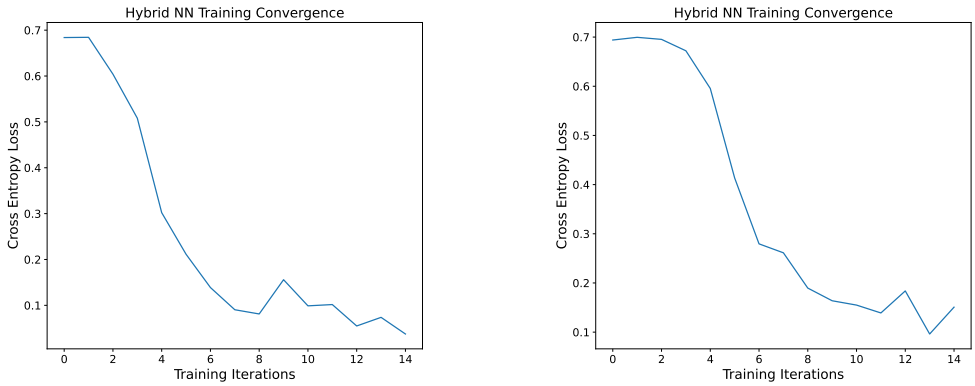


Figura 2.21: Convergenza della loss function nel caso dl circuito ispirato al QAOA, rispettivamente qasm_simulator e per ibmq_santiago

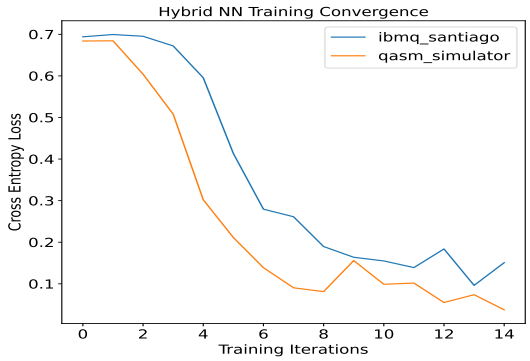


Figura 2.22: Sovrapposizione delle due curve di convergenza per il circuito ispirato al QAOA

2.2.6 Scelta del backend quantistico

Per gli ultimi due casi presi in analisi è necessario fare una precisazione riguardo alla scelta del backend quantistico. Si sono analizzate alcune delle proprietà dei computer quantistici messi a disposizione da IBM nella sezione 1.2.3. Una delle proprietà fondamentali è il valore del *Quantum Volume* che permette di poter stimare la *performance* generale del computer quantistico e quindi di poter comparare tra loro i diversi devices; maggiore è questo valore più complessi sono i problemi che il computer può risolvere.
Inizialmente, a causa dell'attesa che si veniva a creare utilizzando backend a maggior quantum volume che tendenzialmente hanno un maggior numero di *pending jobs* nel cloud, sono stati scelti dei computer a minor quantum volume come ibmq_lima e ibmq_belem che hanno un valore rispettivamente di 8 e 16. Alcuni dei risultati ottenuti in questo modo non erano accettabili; nonostante la rete neurale fosse in grado di riconoscere molto bene i dati in seguito all'apprendimento su simulatore quantistico lo stesso non poteva essere detto per il computer reale. Si sono verificati due casi: nel primo caso, circuito QAOA, non si aveva una decrescita efficace della loss function. Nel secondo caso, nonostante l'apprendimento avvenisse, la performance del modello sul set di dati di validazione non dava buoni valori di accuratezza, sotto al 90%, nel caso del circuito con qubit entangled. In particolare il fenomeno era maggiormente evidente utilizzando un set di dati per l'apprendimento rumoroso.
É stato quindi modificato il backend quantistico passando a un valore di quantum volume superiore, ovvero utilizzando ibmq_athens o ibmq_santiago che hanno entrambi valori di quantum volume pari a 32 con lo stesso numero di qubit a disposizione dei precedenti, 5. In questo caso l'apprendimento ha permesso di ottenere migliori risultati, riportati nelle sezioni precedenti. Si suppone che questo fenomeno sia stato evidente negli ultimi due circuiti e non nei primi perchè entravano in gioco delle complessità maggiori nel circuito quantistico, rappresentate soprattuto dall'utilizzo dell'entanglement.

Capitolo 3

Commento ai risultati e possibili future implementazioni

Riassumendo, nel progetto di tesi si sono studiate le caratteristiche di apprendimento di digit MNIST per reti neurali ibride con l'inserimento di un circuito quantistico come layer della rete. Sono state eseguite diverse operazioni, modificando il circuito quantistico secondo i seguenti modelli:

- Circuito con rotazione in tre dimensioni, dipendente da tre parametri, gate U3.
- Circuito con più qubit e un numero di parametri pari al numero di qubit.
- Circuito con qubit in stati entangled.
- Circuito ispirato a un algoritmo di ottimizzazione, QAOA.

3.1 Commento ai risultati

Nei casi di riconoscimento dati 01 tutti i modelli implementati hanno permesso di ottenere buoni risultati per quanto riguarda l'accuratezza del riconoscimento su un set di dati differente, con valore che era abbastanza stabile intorno al 99%. Sembrano promettenti i risultati ottenuti dal circuito quantistico U3 che complessivamente ha permesso di ottenere i migliori risultati su tutti gli esperimenti. Questo tipo di circuito ha inoltre permesso un riconoscimento efficace anche per dati rumorosi sia con apprendimento con dati "puliti" sia con dati anch'essi rumorosi. Un modello di questo tipo potrebbe dunque essere inserito con successo all'interno di una struttura più complessa dove possono entrare in gioco diverse componenti di errore. Si riporta invece in figura 3.1 la sovrapposizione dei grafici per gli ultimi tre esperimenti, dove è stata utilizzata la stessa loss function (Cross Entropy Loss).

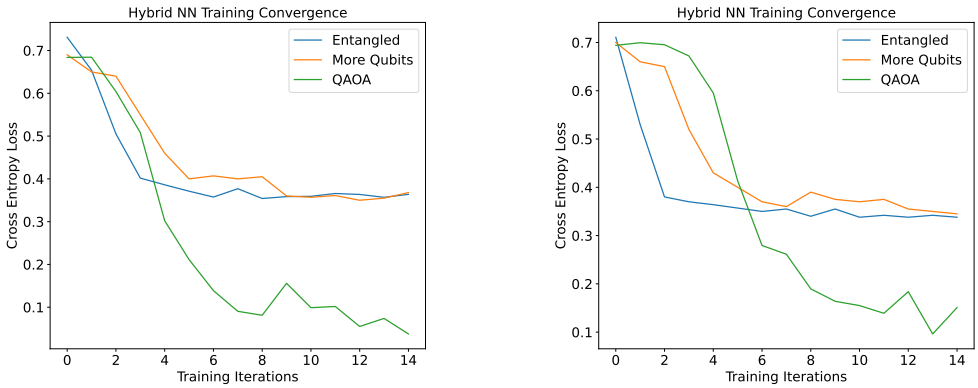


Figura 3.1: Sovrapposizione della discesa della loss function per gli ultimi tre esperimenti sul simulatore (sinistra) e sui backend quantistici (destra)

Riferendosi al caso dei backend quantistici, si può notare come l'esperimento con i qubit entangled abbia permesso di ottenere una convergenza in maniera molto più rapida e con un numero minore di epoche complessivamente rispetto agli altri due esperimenti. Nel caso dell'esperimento con il QAOA si è invece ottenuta una loss function minore, intorno al valore di 0.15, rispetto agli altri due esempi che invece si stabilizzavano entrambi intorno a 0.35, nonostante tutti gli esperimenti partissero da valori di loss function pari, 0.7; questa convergenza si è però verificata con un numero maggiore di epoche, raggiungendo una condizione quasi stabile dopo circa 8 loop.

Nel caso di riconoscimento di dati rumorosi sono stati ottenuti risultati meno promettenti rispetto al circuito con gate U3 nel caso del circuito a più qubit e nel caso del circuito con stati entangled, nonostante la dimensione dei parametri di apprendimento fosse la stessa; il problema si è però in parte risolto provando ad effettuare l'apprendimento con dati rumorosi, che ha infatti permesso di ottenere dei buoni risultati per riconoscimento di dati fino a un valore di 0.5 di deviazione standard. Tra i due in questo caso si sono ottenuti migliori risultati per il circuito con qubit entangled, che ha inoltre utilizzato un set di apprendimento a deviazioni standard più basse, 0.1 contro il 0.2 per l'apprendimento nel caso del circuito a più qubit. Si riporta l'andamento complessivo dell'accuratezza per diverse deviazioni standard in figura 3.2 per set di dati di apprendimento non rumorosi (destra) e per set di dati di apprendimento rumorosi (sinistra).

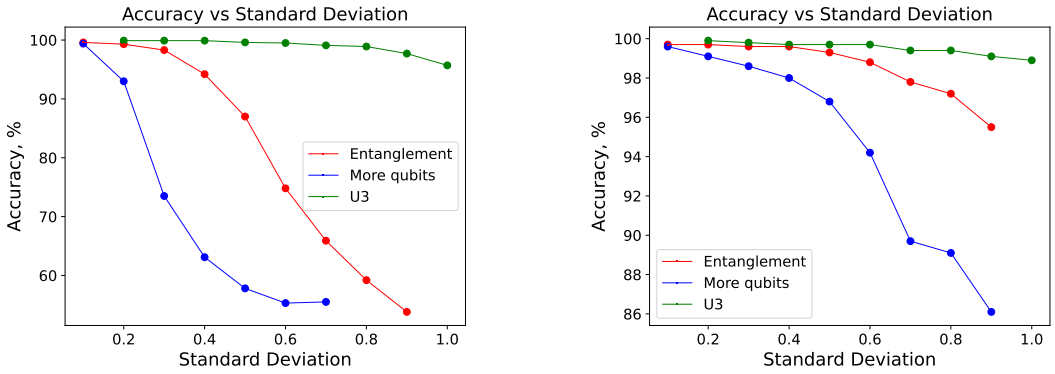


Figura 3.2: Accuratezza al variare della deviazione standard per set di dati di appendimento non rumorosi (destra) e rumorosi (sinistra) con deviazioni standard pari a 0.3, 0.2, 0.1 rispettivamente per gli esperimenti U3, Più Qubit e Entanglment

Per quanto riguarda i parametri che sono stati utilizzati per effettuare l'apprendimento sono stati rilevanti il valore del learning rate, legato alla parte classica della rete, e il valore dello shift scelto, quindi legati alla singola operazione di ottimizzazione dei parametri. Il valore del learning rate è stato simile, intorno a 0.01, per la maggior parte dei casi; è stato però necessario diminuire il suo valore in alcuni casi di riconoscimento di dati rumorosi e di apprendimento sui computer quantistici reali. Anche il valore dello shift dei parametri quantistici è stato simile nella maggior parte degli esperimenti, $\pi/2$, tranne nell'ultimo esperimento dove è stato utilizzato un valore di 0.9. Non sono state riscontrate sensibilità del modello al variare della scelta della loss function o dell'optimizer forniti da pytorch.

Non sono state riscontrate grosse differenze tra i risultati ottenuti con i backend quantistici e quelli ottenuti con i simulatori basati su backend classici. Sono state tuttavia evidenziate nella sezione 2.2.6 delle dipendenze del modello dalla scelta del backend quantistico, legate soprattuto al valore del quantum volume nel momento in cui i circuiti quantistici diventavano più complessi, aggiungendo l'entanglement.

Nel corso del progetto di tesi sono state sviluppate delle reti neurali classiche per riconoscimento di digit 01 e 09 al fine di poter confrontare i risultati ottenuti con il modello ibrido. Non sono stati effettuati confronti riguardanti il tempo impiegato dalle macchine ad apprendere, prendendo in considerazione il fatto che per poter utilizzare i computer quantistici è necessario collegarsi in cloud. Riguardo all'accuratezza che è stata ottenuta invece nel caso classico, dove il modello di rete neurale implementato è analogo a quello riportato in figura 1.5, si ottengono dei risultati molto buoni, sopra al

99.5%. Si suppone che questo sia legato sia alla grande dimensione della rete classica, che possiede molti parametri, sia al compito non particolarmente complesso. In ogni caso non è verificato che effettivamente l'aggiunta della parte quantistica alla rete neurale possa dare dei vantaggi da un punto di vista di tempo o di accuratezza per questo specifico problema, che sembra essere risolto molto bene anche nel caso classico. É comunque un buon punto di partenza per capire che tipo di implementazioni possono essere effettuate per reti neurali ibride, le quali potrebbero poi essere utilizzate per problemi non risolvibili in maniera efficace dai computer classici.

3.2 Possibili implementazioni del modello

Diversi sono i punti in cui il modello può essere migliorato e la sua analisi approfondita. Si riportano alcune idee:

- Estendere il riconoscimento a diversi dati: il riconoscimento ai valori 02 e 04 non ha dato dei buoni risultati sul simulatore quantistico. Si ritiene tuttavia che non siano stati effettuati sufficienti esperimenti, a causa del lungo tempo che la macchina impiegava ad apprendere e ai diversi parametri che dovevano essere modificati al fine di ottenere una corretta convergenza della loss function. In ogni caso, estendere il riconoscimento a più digit potrebbe essere un buon modo per stimare la validità del modello e si potrebbero ottenere interessanti risultati per quanto riguarda l'accuratezza, da confrontare con il caso classico. Per fare questo potrebbe essere utilizzato o il modello a più qubit con un'ulteriore layer classica per ridimensionare l'output, come nel caso del circuito a più qubit nella sezione 2.2.3, oppure si potrebbe implementare un circuito che abbia tanti valori in uscita tanti sono i digit da riconoscere, utilizzando quindi il circuito come ultima layer. Eventualemente il riconoscimento potrebbe essere esteso anche a dati più complessi, per applicazioni per esempio in High Energy Physics.

- Implementazione di algoritmi più specifici: potrebbero essere creati circuiti quantistici con modelli ispirati ad algoritmi di ottimizzazione già consolidati in quantum computing, in maniera analoga al caso del circuito ispirato al QAOA. In questo caso bisognerebbe capire come poter sfruttare questo tipo di algoritmi al meglio.

- Studiare i parametri utilizzati: comprendere in maggior dettaglio l'effetto di una specifica loss function o di un optimizer, per trovare quale sia il migliore per problemi di questo genere.

- Rete classica vs Rete ibrida: cercare di comprendere meglio quali potrebbero le differenze tra i due casi e se la parte quantistica del circuito dia effettivamente un

vantaggio in casi specifici. Si può pensare di ridurre le dimensioni della rete, semplificando il modello, e confrontare l'apprendimento nei due casi per reti dipendenti dallo stesso numero di parametri.

3.3 Conclusioni

Con questo lavoro di tesi si è voluto testare uno specifico caso di quantum machine learning che facesse uso di reti neurali ibride, introducendo degli elementi tipici della meccanica quantistica nel calcolo necessario all'apprenidmento della macchina. Questo metodo è solo introduttivo e si è verificato che il probelma poteva essere risolto in maniera efficace anche senza l'utilizzo della parte quantistica, ma è un interessante punto di partenza per diversi sviluppi che si possono avere con questo tipo di tecnologie. L'idea di utilizzare la meccanica quantistica al fine di riconoscere delle strutture nei dati a disposizione si basa proprio sulla diversità di questa teoria, sulla possibilità di riconoscere strutture che non potrebbero essere riconosciute classicamente. Inoltre l'algebra lineare è alla base di entrambe queste teorie, per codificare i dati nel caso del machine learning e per eseguire le operazioni sui diversi stati quantistici nel caso della meccanica quantistica; questa somiglianza potrebbe essere un'ulteriore punto a favore per l'unificazione dei due approcci [10]. Nello specifico caso delle reti neurali ibride, una delle maggiori sfide da affrontare attualmente in questo ambito è come poter includere dati classici all'interno di un computer quantistico. Nel caso in esame si è scelto di utlizzare i dati classici come parametri per i gate quantistici. Altri approcci vedrebbero di utilizzare i dati proprio come stati quantistici su cui viene in seguito effettuata una classificazione e quindi si dovrebbe cercare un modo per codificare i dati classici in uno spazio di Hilbert in maniera efficace.

Il Quantum Machine Learning è solo una delle possibili applicazioni dei computer quantistici: diversi sono gli ambiti di ricerca impegnati al fine di evidenziare problemi che possano essere risolti con successo con questo tipo di computer. Si pensi per esempio alla crittografia quantistica, allo studio delle molecole, che potrebbe portare a grossi progressi in medicina e nella modellizzazione di nuovi farmaci, alle comunicazioni, si sta pensando alla possibilità di una rete internet quantistica, e alla costruzione di nuovi sensori quantistici. Sarà però fondamentale superare le sfide tecnologiche poste dalla creazione di questo tipo di computer, provando ad ottenere meno errori, una maggior efficienza e maggiori dimensioni dei devices. Tutto questo è elettrizzante; siamo al principio di una nuova rivoluzione che coinvolgerà non solo i fisici ma anche molti altri scienziati e che modificherà profondamente la vita quotidiana di tutti.
L'aspetto che forse trovo più interessante è il tentativo di approcciarsi alla natura in maniera differente e originale. La meccanica quantistica offre alla nostra mente costruita dall'esperienza dei cinque sensi dei paradossi che però non contengono nessuna contraddizione logica intrinseca; il problema è nel nostro modo di ragionare. É quindi necessario un profondo cambio di punto di vista e utilizzare strumenti che funzionino nello stesso modo dei fenomeni che vogliamo analizzare è sicuramente un'idea geniale; se davvero Dio gioca a dadi con l'universo, dobbiamo cambiare la nostra prospettiva per comprendere le regole del gioco.

Appendice A

Codice degli esperimenti

Si riporta il codice del programma utilizzato per un particolare esperimento, ovvero il circuito con qubit entangled. Gli altri esperimenti sono stati effettuati in maniera analoga, cambiando alcune caratteristiche nella definizione del circuito, nei parametri scelti per l'apprendimento e nella rete neurale.

```python
#IMPORTING LIBRARIES
import numpy as np

from qiskit import *
from qiskit import IBMQ

import matplotlib.pyplot as plt

import torch
from torch.autograd import Function
from torchvision import datasets, transforms
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F

import itertools


# Loading your IBM Quantum account(s)
provider = IBMQ.load_account()
```

```python
############################
# GLOBAL VARIABLE DEFINITION
n_qubits = 3
n_shots = 1000
shift = np.pi / 2
learning_rate = 0.008

backend = qiskit.Aer.get_backend('qasm_simulator')
#backend = provider.get_backend('ibmq_santiago')
#print("We are executing on...",backend)
#print("It has", backend.status().pending_jobs, "pending jobs")

#CREATING QUANTUM CIRCUIT POSSIBLE OUTPUTS
def create_QC_OUTPUTS(n_qubits):
    measurements = list(itertools.product([1, 0], repeat=n_qubits))
    return [''.join([str(bit) for bit in measurement]) for measurement in measurements]
```

Figura A.1: Inclusione librerie a destra, definizione delle variabili globali, definizione del backend (quantistico o simulatore) e funzione per definire i possibili output del circuito (non usato in questo caso ma utile per misurazioni su più qubit) a sinistra.

```python
#QUANTUM CIRUCIT CLASS DEFINITION
class QuantumCircuit:
    """This class provides an interface to interact with our Quantum Circuit"""
    def __init__(self, n_qubits, backend, shots):

        # -----Circuit definition
        self._circuit = qiskit.QuantumCircuit(n_qubits, 1)
        self.n_qubits = n_qubits
        self.parameters = qiskit.circuit.ParameterVector('parameters', n_qubits)
        all_qubits = [i for i in range(n_qubits)]  # qubits vector

        self._circuit.h(0)
        self._circuit.cx(0, 1)
        self._circuit.cx(1,2)
        self._circuit.barrier()

        for k in range(n_qubits):
            self._circuit.rz(self.parameters[k], k)
        self._circuit.barrier()

        self._circuit.cx(1,2)
        self._circuit.cx(0,1)
        self._circuit.h(0)
        self._circuit.barrier()

        self._circuit.measure(0,0)
        # -----

        self.backend = backend
        self.shots = shots

    #EVALUATING EXPECTATION VALUE FOR FIRST QUBIT OUTPUT
    def expectation_Z(self,counts, shots, n_qubits):
        expects = np.zeros(1)
        for key in counts.keys():
            percentage = counts[key]/shots
            check = np.array([(float(key[i]))*percentage for i in range(1)])
            expects += check
        return expects

    def run(self, thetas):
        # acting on backend
        thetas = thetas.squeeze()
        p_circuit = self._circuit.bind_parameters({self.parameters[k]: thetas[k].item()
                                                   for k in range(self.n_qubits)})
        job_sim = qiskit.execute(p_circuit,
                                 self.backend,
                                 shots=self.shots)

        result = job_sim.result()
        counts = result.get_counts(p_circuit)

        expectation = self.expectation_Z(counts, self.shots, self.n_qubits)

        return expectation
```

Figura A.2: Definizione della classe per la definizione del circuito quantistico

```
####################################
#TESTING THE CIRCUIT
circuit = QuantumCircuit(n_qubits, backend, n_shots)
circuit._circuit.draw(output='mpl', filename = 'Bell.png')

rotation = torch.Tensor([np.pi / 4] * n_qubits)
exp = circuit.run(rotation)
print('Expected value for rotation pi/4: {}'.format(exp))

#EXTENDING AUTOGRAD FOR ML FUNCTION DEFINITION
class HybridFunction(Function):
    """Hybrid quantum-classical function definition"""

    @staticmethod
    def forward(ctx, input, quantum_circuit, shift):
        """Forward pass computation"""

        ctx.shift = shift
        ctx.quantum_circuit = quantum_circuit

        expectation_z = ctx.quantum_circuit.run(input)  #evaluating model with trainable parameter
        result = torch.tensor([expectation_z])
        ctx.save_for_backward(input, result)

        return result
```

Figura A.3: Test del circuito e prima parte dell'estensione della classe Function di autograd, ridefininendo la funzione forward

```
    @staticmethod
    def backward(ctx, grad_output):  # grad_output is previous gradient
        """Backward computation"""

        input, expectation = ctx.saved_tensors  # evaluated in forward
        input = torch.reshape(input, (-1,))
        gradients = torch.Tensor()

        # iterating to evaluate gradient
        for k in range(len(input)):
            # shifting parameters
            shift_right, shift_left = input.detach().clone(), input.detach().clone()
            shift_right[k] += ctx.shift
            shift_left[k] -= ctx.shift

            # evaluating model after shift
            expectation_right = ctx.quantum_circuit.run(shift_right)
            expectation_left = ctx.quantum_circuit.run(shift_left)

            # evaluating gradient with parameter shift rule
            gradient = torch.tensor([expectation_right]) - torch.tensor([expectation_left])

            gradients = torch.cat((gradients, gradient.float()))

        result = gradients.float() * grad_output.float()

        return (result).T, None, None

#CREATING QUANTUM LAYER
class Hybrid(nn.Module):
    """Hybrid quantum-cassical layer definition"""

    def __init__(self, n_qubits, backend, shots, shift):
        super(Hybrid, self).__init__()

        self.quantum_circuit = QuantumCircuit(n_qubits, backend, shots)
        self.shift = shift  # parameter shift

    def forward(self, input):
        return HybridFunction.apply(input, self.quantum_circuit, self.shift)
        # calling forward and backward
```

Figura A.4: Seconda parte dell'estensione della classe Function di autograd, ridefinendo la funzione backward. Creazione di una layer quantistica

```
# DATA LOADING
# training data
n_samples = 100
X_train = datasets.MNIST(root='./data', train=True, download=True,
                         transform=transforms.Compose([transforms.ToTensor()]))

#keeping only label 0 and 1
idx = np.append(np.where(X_train.targets == 0)[0][:n_samples],
                np.where(X_train.targets == 1)[0][:n_samples])
X_train.data = X_train.data[idx]
X_train.targets = X_train.targets[idx]

# making batches (dim = 1).
train_loader = torch.utils.data.DataLoader(X_train, batch_size=1, shuffle=True)
data_iter = iter(train_loader)

# showing samples
n_samples_show = 6
fig, axes = plt.subplots(nrows=1, ncols=int(n_samples_show), figsize=(10, 3))

while n_samples_show > 0:
    images, targets = data_iter.__next__()

    axes[int(n_samples_show) - 1].imshow(images[0].numpy().squeeze(),
                                         cmap='gray')
    axes[int(n_samples_show) - 1].set_xticks([])
    axes[int(n_samples_show) - 1].set_yticks([])
    axes[int(n_samples_show) - 1].set_title("Labeled: {}".format(targets.item()))

    n_samples_show -= 1
```

```
##################################
#CREATING THE NN
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
        self.dropout = nn.Dropout2d()
        self.fc1 = nn.Linear(256, 64)
        self.fc2 = nn.Linear(64,n_qubits)
        self.hybrid = Hybrid(n_qubits, qiskit.Aer.get_backend(backend), n_shots, shift )

    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)),2)
        x = F.max_pool2d(F.relu(self.conv2(x)),2)
        x = self.dropout(x)
        x = x.view(1,-1) #reshaping tensor
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        x = self.hybrid(x) #calling the forward method

        return torch.cat((x, 1-x), -1) #returning probabilities
```

Figura A.5: Loading dei dati di apprendimento e creazione della rete neurale

```
#####################################
#TRAINING LOOP DEFINITION AS A FUNCTION
def training_loop (n_epochs, optimizer, model, loss_fn, train_loader):
    loss_values = []
    for epoch in range(0, n_epochs, +1):
        total_loss = []

        for batch_idx, (data, target) in enumerate(train_loader):
            optimizer.zero_grad()#getting rid of previous gradients

            output = model(data)#forward pass
            loss = loss_fn(output, target)
            loss.backward()

            optimizer.step()#updating parameters
            total_loss.append(loss.item())

        loss_values.append(sum(total_loss)/len(total_loss))#obtainign the average loss
        print('Training [{:.0f}%]   Loss: {:.4f}'.format(100*(epoch+1)/n_epochs, loss_values[-1]))

    return loss_values
#TRAINING THE NETWORK
model = Net()
params = list(model.parameters())

optimizer = optim.Adam(params, learning_rate)
loss_func = nn.CrossEntropyLoss()

loss_list = []
model.train()

epochs = 15
loss_list = (training_loop(epochs, optimizer, model, loss_func, train_loader))

#plotting the training graph
plt.figure(num=2)
plt.plot(loss_list)
plt.title('Hybrid NN Training convergence')
plt.xlabel('Training Iterations')
plt.ylabel('Cross Entropy Loss')
```

Figura A.6: Training loop e apprendimento della rete con plot della convergenza della loss function

```python
#LOADING VALIDATION DATA
n_samples = 1000

X_test = datasets.MNIST(root='./data', train=False, download=True,
                        transform=transforms.Compose([transforms.ToTensor()]))

idx = np.append(np.where(X_test.targets == 0)[0][:n_samples],
                np.where(X_test.targets == 1)[0][:n_samples])

X_test.data = X_test.data[idx]
X_test.targets = X_test.targets[idx]

test_loader = torch.utils.data.DataLoader(X_test, batch_size=1, shuffle=True)

#DEFINING A FUNCTION TO VALIDATE THE MODEL
def validate(model, test_loader, loss_function, n_test, axes):
    correct, count = 0, 0
    total_loss = []
    with torch.no_grad():
        for batch_idx, (data, target) in enumerate(test_loader):
            output = model(data)  #evaluating the model on test data

            # evaluating the accuracy of our model
            pred = output.argmax(dim=1,keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

            #evaluating loss function
            loss = loss_function(output, target)
            total_loss.append(loss.item())

            #printing the resut as images
            if count >= n_test:
                continue
            else:
                axes[count].imshow(data[0].numpy().squeeze(), cmap='gray')

                axes[count].set_xticks([])
                axes[count].set_yticks([])
                axes[count].set_title('P: {}'.format(pred.item()))
            count += 1

        print('Performance on test data: \n\tLoss: {:.4f}\n\tAccuracy: {:.1f}%'
              .format(sum(total_loss)/len(total_loss),(correct / len(test_loader))*100))
```

Figura A.7: Loading dei dati di validazione e definizione della funzione per validare il modello

```
# TESTING THE MODEL
n_test_show = 10
fig, axes = plt.subplots(nrows=1, ncols=n_test_show, figsize=(10, 3))

model.eval()
validate(model, test_loader, loss_func, n_test_show, axes)

#################################
#ADDING GAUSSIAN NOISE TO VALIDATION DATA
class AddGaussianNoise(object):
    def __init__(self, mean=0., std=5):
        self.std = std
        self.mean = mean

    def __call__(self, tensor):
        return tensor + torch.randn(tensor.size()) * self.std + self.mean

    def __repr__(self):
        return self.__class__.__name__ + '(mean={0}, std={1})'.format(self.mean, self.std)


stop, mean, std_dv = 10, 0, 0.1
for i in range(1, stop):
    print('Gaussian noise with std deviation: ', std_dv)

    X_test_n = datasets.MNIST(root='./data', train=False, download=True,
                              transform=transforms.Compose([transforms.ToTensor(),
                                                            AddGaussianNoise(mean, std_dv)]))

    idx = np.append(np.where(X_test_n.targets == 0)[0][:n_samples],
                    np.where(X_test_n.targets == 1)[0][:n_samples])

    X_test_n.data = X_test_n.data[idx]
    X_test_n.targets = X_test_n.targets[idx]
    test_loader_n = torch.utils.data.DataLoader(X_test_n, batch_size=1, shuffle=True)
    test_iter_n = iter(test_loader_n)

    fig_1, axes_1 = plt.subplots(nrows=1, ncols=n_test_show, figsize=(10, 3))

    model.eval()
    validate(model, test_loader_n, loss_func, n_test_show, axes_1)
    std_dv = std_dv + 0.1
```

Figura A.8: Test sul modello, definizione della funzione che aggiunge rumore gaussiano ai dati, loading dei dati rumorosi e test del modello sui dati rumorosi in un ciclo, aggiornando il valore della deviazione standard

Bibliografia

[1] Computer vision. https://www.ibm.com/topics/computer-vision.

[2] Extending pytorch. https://pytorch.org/.

[3] Hybrid quantum-classical neural networks with pytorch and qiskit. https://qiskit.org/textbook/ch-machine-learning/machine-learning-qiskit-pytorch.html.

[4] Ibm quantum experience. https://quantum-computing.ibm.com/.

[5] The mnist dataset. http://yann.lecun.com/exdb/mnist/. LeCun Yann; Cortes Corinna; Burges Christopher.

[6] Neural networks and deep learning. http://neuralnetworksanddeeplearning.com/. Nielsen Micheal.

[7] Pytorch. https://pytorch.org/.

[8] Qiskit. https://qiskit.org/.

[9] Quantum manifesto. https://ec.europa.eu/futurium/en/content/quantum-manifesto-quantum-technologies.html.

[10] Quantum ml. www.youtube.com/channel/UCVROlDxzfRrlRexJvvCuQeg. Wittek Peter.

[11] What is quantum annealing. https://docs.dwavesys.com/docs/latest/c_gs_2.html.

[12] A. Abbas, D. Sutter, C. Zoufal, et al. The power of quantum neural networks. *Nat Comput Sci 1, 403–409*, 2021.

[13] E. Gavin Crooks. Gradients of parametrized quantum gates using the parameter-shift rule and gate decomposition. *arXiv:1905.13311*, 2019.

[14] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. Quantum approximate optimization algorithm. *arXiv:1411.4028*, 2014.

[15] Elias Fernandez-Combarro Alvarez. Online introductory lectures on quantum computing, indico.cern.ch/event/970903/. 2020.

[16] Stephen Gasiorowicz. *Quantum Physics, 3nd edition.* 2003.

[17] C.R. Harris, K.J. Millman, S.J. van der Walt, et al. Array programming with numpy. *Nature 585, 357-362*, 2020.

[18] Hunt. *A beginners Guide to Python 3 Programming.* 2019.

[19] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[20] S. Lloyd, M. Schuld, A.I. Ijaz, J. Izaac, and N. Kilora. Quantum embeddings for machine learning. *arXiv:2001.03622*, 2020.

[21] N. Moll et al. Quantum optimization using variational algorithms on near-term quantum devices. *Quantum Science and Technology 3 030503*, 2018.

[22] Frank Phillipson. Quantum machine learning: Benefits and practical examples. *International Workshop on QuAntum SoftWare Engineering & pRogramming (QANSWER)*, 2020.

[23] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.

[24] Al-salam A. Selami and Fadhil. A study of the effects of gaussian noise on image features. *Kirkuk University Journal*, 2016.

[25] Eli Stevens, Luca Antiga, and Thomas Viehmann. *Deep Learning with Pytorch.* 2020.

Ringraziamenti

Grazie ai miei relatori: Dr. Andrea Giachero, Prof. Alberto Leporati e Prof. Claudio Ferretti dell'Università degli studi Milano-Bicocca.
Grazie a Stefano Carrazza dell'Università degli studi di Milano e a Leonardo Banchi dell'Università degli studi Firenze per i preziosi consigli.

Grazie ai miei genitori, nonché allenatori, che mi hanno insegnato, letteralmente, a "passare" ogni ostacolo sul percorso con fiducia e determinazione.
Grazie alla mia famiglia, presenza costante nella mia crescita.

Grazie a Eleonora, con cui ho condiviso fin dal primo giorno questo impegnativo ma soddisfacente percorso.
Grazie alle mie amiche: Alessia, Giulia, Alessandra e Arianna.

Grazie all'Atletica, sport che mi ha insegnato a pormi degli obiettivi e lavorare per ottenerli, affrontando le sfide con consapevolezza, e che mi ha reso la persona che sono oggi.
Grazie agli amici incontrati in pista, per il supporto nei momenti difficili (e nelle ripetute) e per la felicità condivisa nei momenti speciali.