POLITECNICO DI TORINO

Facoltà di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Electronic Engineering

Tesi di Laurea Magistrale

Deep Information Network: complexity, analysis and hardware implementation

Relatori:
Prof. Guido Masera
Prof.ssa Monica Visintin

Candidato:
Nicola Rallo

Dicembre 2019

Sommario

Le reti neurali profonde (DNNs) sono degli algoritmi particolarmente avanzati ad elevata complessità computazionale operanti in diverse applicazioni di intelligenza artificiale, tra le quali robotica, riconoscimento di immagini e riconoscimenti vocali. L'applicazione dei DNNs si riscontra anche nelle auto a guida autonoma, in apparecchiature per la rilevazioni di malattie e nei videogiochi.

Il continuo investimento nel settore ha portato grandi sviluppi e miglioramenti con annesse facilità nell' uso quotidiano, merito della semplicità con le quali riescono ad assumere automaticamente delle decisioni sulla base di un apprendimento iniziale. Si tratta, infatti, di algoritmi sofisticati che permettono al sistema di imparare a comportarsi in un modo ben definito *(training phase)* qualora si presentassero delle situazioni da gestire autonomamente e non programmate precedentemente *(test phase)*.

Sono state realizzate diverse implementazioni, sia hardware che software, in grado di ottenere informazioni utili e necessarie al fine di stimare nel modo più efficiente possibile gli eventi circostanti partendo semplicemente da un insieme di dati in ingresso.

Questo lavoro di tesi ha avuto come obiettivo quello di analizzare nel dettaglio un modello "esatto" di rete neurale descritto a livello software, realizzando successivamente una possibile implementazione hardware con opportuni confronti.

La rete in questione prende il nome di DIN *(Deep Information Network)* ed è costituita da una serie di nodi disposti su diversi layers che hanno lo scopo di elaborare i dati in ingresso al fine di poter predire un determinato evento. L'intera struttura ad albero ha presenta sia elevate capacità di estrazione delle informazioni sia una buona precisione sulla stima finale.

Una volta analizzato il comportamento dell'algoritmo, è stato affrontato il concetto di parametrizzazione e quantizzazione dei dati realizzando un modello software in fixed point.

Un primo step di progetto verte sulla scelta ottimale del parallelismo dei dati elaborati dall'algoritmo.

La fase successiva dello studio si è incentrata sulla progettazione di un'architettura hardware che fosse in grado di seguire il flusso del modello esatto. Ulteriore

oggetto di studio sono state le scelte effettuate dal punto di vista della gestione dei dati, passando da una struttura completamente seriale ad una parallela, aumentando la velocità di esecuzione a discapito di un aumento dell'area occupata.

Non di minore importanza sono stati i confronti in termini di prestazioni e speed-up rispetto al modello software.

I risultati finali ottenuti mostrano che l'architettura parallela è in grado di completare l'elaborazione richiesta entro un tempo significativamente inferiore rispetto a quello richiesto nell'esecuzione del modello software. In particolare, i risultati ottenuti sono stati i seguenti:

$$T_{ciclo,software} = 31 \cdot 4367 = 135,377ms \tag{1}$$

$$T_{ciclo,hardware} = 31 \cdot 23 = 713\mu s \tag{2}$$

Infine sono state discusse le conclusioni del lavoro svolto, evidenziando anche altri accorgimenti implementativi, ed eventuali sviluppi futuri.

Indice

Capitolo 1

Introduzione

Le reti neurali profonde (DNNs) sono alla base di molteplici applicazioni di intelligenza artificiale che vanno dal riconoscimento vocale e video sino all'avvento della guida autonoma. Oggi, i DNNs sono anche in grado di superare addirittura l'accuratezza umana in virtù della loro capacità di estrarre funzionalità ad alto livello dai dati sensoriali. Ovviamente, la precisione di questi algoritmi deriva da un'elevata complessità computazionale seguita da un elevato numero di strumenti di calcolo estremamente affidabili.

1.1 L'evoluzione delle reti neurali profonde

Le reti neurali profonde corrispondono ad un campo principale dell'intelligenza artificiale (AI), ovvero la scienza che si occupa di progettare sistemi intelligenti che sono in grado di raggiungere dei veri obiettivi come nel caso umano. Nella figura seguente, è mostrata un'immagine che mette in relazione le DNNs con l'intelligenza artificiale:

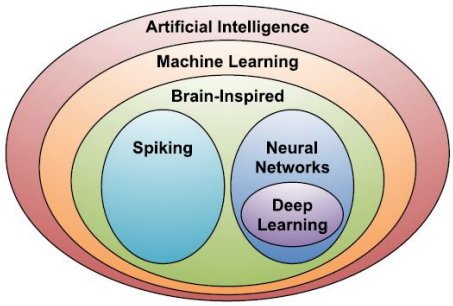

Figura 1.1. Reti neurali profonde nel contesto dell'intelligenza artificiale [1]

Come è possibile notare, all'interno dell'AI è presente un altro sottocampo che prende il nome di *Machine Learning*. Quest'ultimo rappresenta un campo di studio che offre ai computer la possibilità di apprendere autonomamente senza un'esplicita programmazione. Pertanto, si tratta di un meccanismo per addestrare un algoritmo affinché sia in grado di imparare da determinate situazioni. La fase di apprendimento comporta l'uso di una notevole quantità di dati nonchè un algoritmo efficiente. Si tratta, dunque, di una macchina ad apprendimento automatico che, attraverso una fase di *training* , saprà gestire ulteriori problemi in maniera del tutto autonoma. Inoltre, gli algoritmi di apprendimento sono basati su un modello matematico i quali elaborano i dati in ingresso utili per l'addestramento, al fine di poter prendere successivamente delle decisioni mediante stime del tutto autonome (fase di *testing*).

All'interno del campo del machine learning è presente un'area che spesso è indicata come *Brain-Inspired*, ovvero quell'area nella quale sono concentrati tutti i calcoli ispirati al funzionamento de cervello umano [1]. Dal momento che quest'ultimo è la "macchina" perfetta per l'apprendimento e risoluzione dei problemi, l'area indicata rappresenta quella zona dove cercare di aumentare l'efficienza degli algoritmi per avvicinarsi sempre più al comportamento umano.

L'unità cellulare che costituisce il sistema nervoso è il *neurone* che è in grado di elaborare, trasmettere e ricevere impulsi nervosi rappresentando la principale unità di calcolo dell'intelligenza dell'uomo [2]. La figura sottostante mostra una struttura anatomica di un neurone:

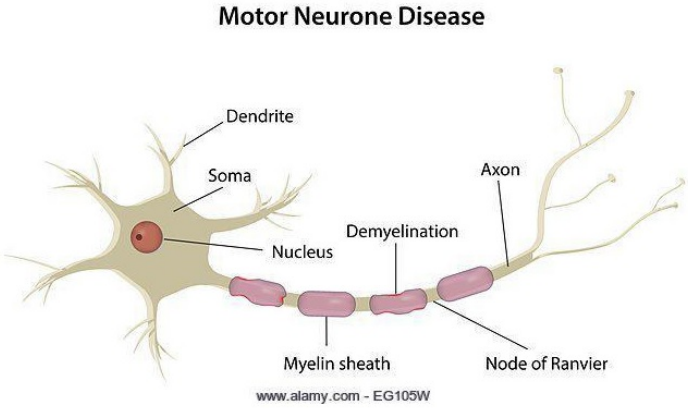

Figura 1.2. Struttura anatomica di un neurone
[3]

Il sistema nervoso umano è composto da circa 86 miliardi di neuroni i quali risultano connessi mediante una serie di *sinapsi*. Ogni neurone riceve i segnali mediante

i suoi *dentriti*, elabora i dati ricevuti e genera le informazioni in uscita lungo il suo *assone*. Nella figura seguente, invece, è stato schematizzato un modello matematico per un neurone utilizzate nelle reti neurali DNNs:

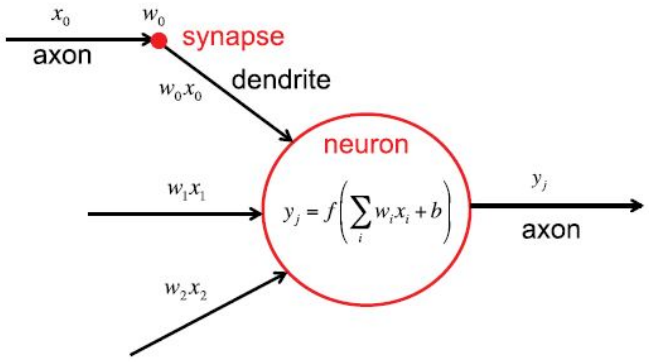

Figura 1.3. Modello matematico di un neurone usato nelle DNN [1]

Tutti i segnali che giungono in ingresso al neurone attraverso i dentriti, vengono elaborati mediante una determinata funzione non lineare generando un segnale di uscita lungo il suo assone. A sua volta, l'assone si ramifica e si connette ai dentriti di altri neuroni traportando le informazioni in cascata. Una caratteristica fondamentale della sinapsi è consiste nella capacità di ridimensionare il segnale di ingresso (Xi). Il ridimensionamento, indicato come punto di forza sinaptico (w_i), rappresenta una quantità indicante il modo con cui il cervello impara attraverso il cambiamento del peso al quale è legato l'apprendimento. Inoltre, i segnali di ingresso e di uscita sono rappresentati come delle vere e proprie attivazioni.

Nella figura 1.1 è presente anche una sottozona chiamata *Spiking*, ovvero quell'area finalizzata allo studio della comunicazione tra i vari neuroni. In particolare, i segnali che viaggiano tra gli assoni e i dentriti sono degli impulsi le cui informazioni non dipendono solo dalla loro ampiezza ma dipendono anche dall'istante in cui si presentano [1]. Dunque, l'elaborazione che viene fatta all'interno di un neurone dipende dall'ampiezza del picco e dall'istante temporale.

Un'ulteriore sottozona ispirata al cervello umano è invece il *Deep Learning* (figura 1.1) che è in completa opposizione con l'area *Spiking* in quanto il tempismo dei segnali non è rilevante. Il calcolo computazionale che svolge il neurone dipende solo dalle ampiezze degli ingressi e dai pesi w_i [1].

1.2 Teoria dell'informazione

La schematizzazione mediante il modello matematico del comportamento di un neurone ha aperto la strada a diversi studi e teorie su come poter implementare nel modo più efficiente possibile una rete neurale.

Diversi approcci sono stati implementati, tra cui le CNNs *(Convolutional Neural Network)*, molto usate per l'elaborazioni di immagini e video che sfruttano il prodotto di convoluzione dei dati. In questo lavoro di tesi, invece, è stato affrontato una rete neurale mediante la teoria dell'informazione, sfruttando calcoli statistici e probabilistici della sequenza dei dati in ingresso.[4]

Quest'ultime prendono il nome di DIN *(Deep Information Network)* e si basano sul concetto di probabilità e probabilità condizionata. Nel dettaglio, la probabilità rappresenta una quantità compresa tra 0 e 1 che quantifica il verificarsi di un evento piuttosto che un altro. Invece, la probabilità condizionata di un evento A rispetto ad un evento B indica la probabilità che si verifichi l'evento A noto a priori la verifica di B.

Conoscendo questi concetti di base sulla teoria dell'informazione, è stato possibile studiare nel dettaglio questo nuovo modello di rete neurale partendo dal modello software con *Matlab*, evidenziandone pregi e difetti circa la stima finale. L'algoritmo in questione, rappresenta un modello di riferimento e quindi una base per realizzare l'architettura hardware.

1.3 Obiettivi del lavoro svolto e contributi

L'obiettivo principale del lavoro di tesi è stato quello di implementare in hardware l'elaborazione dei dati di un singolo nodo della rete neurale calcolando le varie matrici di probabilità.

Come specificato precedentemente, l'algoritmo è basato sulla teoria dell'informazione, ossia su calcoli statistici e probabilistici di una sequenza di dati in ingresso elaborati da ciascun nodo. Nel caso specifico, i dati forniti in ingresso ai processing element sono dei numeri binari con parallelismo pari a due ottenendo fino a 2^2-1 valoro differenti.

Un esempio applicativo sviluppato in questo lavoro di tesi ha riguardato una serie di voti che i parlamentari americani hanno dato relativamente a diverse leggi che sono state proposte.

A tal proposito, è stato considerato il seguente caso puramente ternario:

- **'0'**, corrisponde al voto favorevole di una legge,
- **'1'**, corrisponde al voto contrario di una legge,

- **'2'**, corrisponde al caso in cui un parlamentare si sia astenuto per diverse ragioni al voto;

A questo punto, in maniera del tutto autonoma, l'algoritmo deve essere in grado, una volta note le votazioni dei parlamentari, di prendere una decisione se costoro fossero democratici o repubblicani. Quindi, a posteriori, noti gli ingressi, l'algoritmo deve fare una stima quanto più affidabile possibile sulle ideologie dei votanti. Tale decisione è binaria ed è stata associata ai seguenti casi:

- **'0'**, repubblicano,
- **'1'**, democratico;

L'elaborazione dei dati in ingresso è stata attuata attraverso calcoli di probabilità, i quali sono stati affrontati anche durante la progettazione hardware del singolo nodo. Tale architettura, dunque, deve essere in grado di riprodurre fedelmente il modello software della rete neurale e di produrre delle stime quanto più affidabili possibili.

Capitolo 2

Deep Information Network

A differenza dei classici algoritmi di deep learning, quali CNN *(Convolutional Neural Network)* che sfruttano principalmente l'operazione di convoluzione dei dati in ingresso al fine di definire una buona caratterizzazione sulla stima finale, il modello in esame si basa sulla teoria dell'informazione. Nello specifico, si tratta di un nuovo modello di classificatore avente una struttura ad albero, progettato e studiato mediante un particolare apprendimento statistico di una serie di dati di ingresso.

Trattandosi di un sistema autonomo sulla stima finale, è essenziale distinguere in primis le fasi salienti dell'algoritmo:

- **training**
- **test**

La prima fase si riferisce ad uno step di apprendimento in cui l'algoritmo conosce a priori sia i dati in ingresso (il voto di ciascun parlamentare per una data legge) che l'uscita (repubblicano o democratico). Ciascun process element elabora questi dati aggiornando in continuazione delle matrici di probailità condizionata usate per la fase successiva.

La fase di *test*, invece, non rende più note le uscite ovvero l'orientamento politico dei parlamentari, ma si è a conoscenza solamente del risultato dei loro voti. Questo step è molto delicato perchè, tramite le matrici di probabilità calcolate precedentemente, sono state stimate le uscite (democratico o repubblicano) sulla base degli ingressi.

2.1 Processing Element

L'elemento principale della rete DIN è dunque il "nodo informativo" che è in grado di comprimere i dati di ingresso mantenendo ovviamente inalterate le informazioni

elaborate. Quest'ultimo svolge il ruolo del neurone per il cervello umano ed è stato schematizzato con il seguente modello matematico:

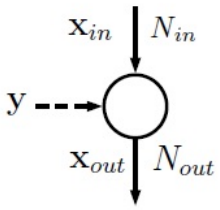

Figura 2.1. Modello matematico di un neurone per la rete DIN [4]

dove:

- $\mathbf{X}_{in}$ rappresentano i dati in ingresso al nodo;
- $\mathbf{N}_{in} = 3$, rappresenta la cardinalità dei dati in ingresso che nel nostro corrisponderanno agli eventi "favorevole" o "contrario" ad una legge o "astenuto" al voto;
- $\mathbf{X}_{out}$, rappresenta il dato in uscita dal nodo dopo l'opportuna elaborazione.
- $\mathbf{N}_{out} = 2$, rappresenta la cardinalità dei valori in uscita dal nodo che corrisponderanno alle ideologie dei parlamentari "repubblicano" o "democratico";
- $\mathbf{y}$, rappresenta il target, ovvero l'ideologia del parlamentare che è nota solo nella fase di training e che viene fornito come ingresso. Nella fase di test, l'obiettivo sarà quello di fare la stima e confrontare i valori ottenuti con quelli noti;

Dalla figura del nodo, è possibile notare anche una compressione dei dati di ingresso passando da una cardinalità N_{out} a N_{in} ($N_{out}<N_{in}$) mantenendo tuttavia le informazioni essenziali [4].

Bisogna sottolineare che durante la fase di training il nodo tiene conto sia dei dati in ingresso, X_{in}, sia delle dichiarazioni "democratico" o "repubblicano" che forniscono a priori ogni singolo individuo Y.

2.2 Struttura dataset

In seguito è mostrata la struttura tipica dei dati forniti ad ogni singolo nodo:

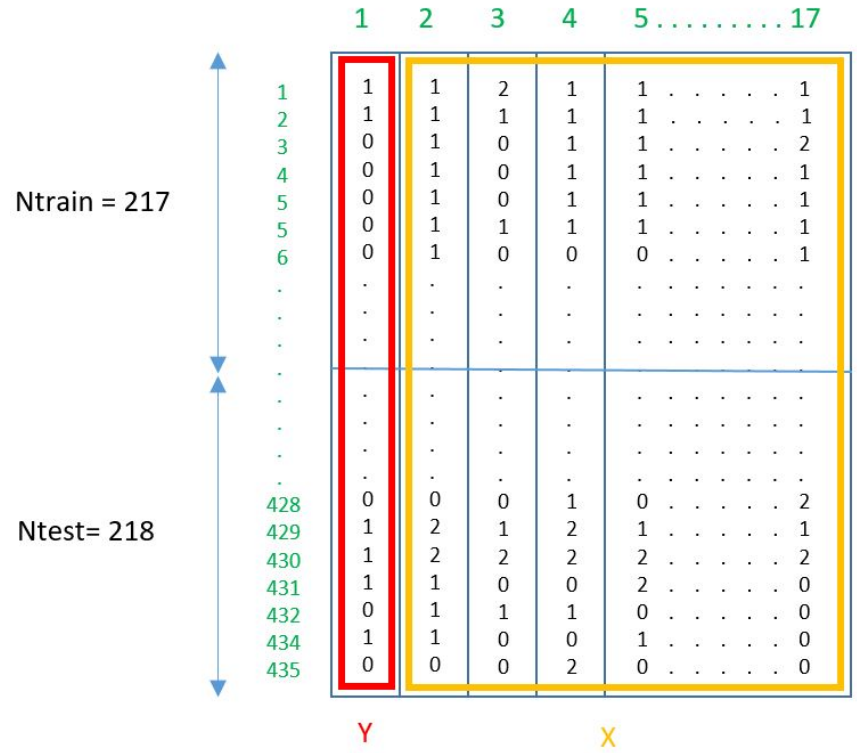

Figura 2.2. Dataset delle votazioni dei parlamentari americani

Il dataset è stato suddiviso in due parti fondamentali corrispondenti alle uscite $\mathbf{Y}$ (dati che l'algoritmo deve successivamente stimare) e agli ingressi $\mathbf{X}$ (dati forniti al nodo). Più in dettaglio, la matrice è costituita da $\mathbf{Y}_{train}$, che corrisponde al vettore delle dichiarazioni di ogni parlamentare e quindi ai dati noti, mentre $\mathbf{Y}_{test}$ corrisponde alle uscite finali note a priori che devono essere confrontate con i valori ottenuti in fase di test. Stessa cosa per la matrice $\mathbf{X}$ in cui sono presenti i dati utili per l'addestramento, $\mathbf{X}_{train}$, e quelli neccessari per il test $\mathbf{X}_{test}$.

Nella *fase di training*, ogni nodo ha in ingresso una feature cioè il voto $\mathbf{X}_{in}$ (0 o 1 o 2) per una determinata legge e riceve anche le dichiarazioni note dei parlamentari $\mathbf{Y}_{train}$. Ogni colonna di $\mathbf{X}$ corrisponde ad una singola legge per un totale di 16 per il dataset in esame. Da queste informazioni in ingresso, il nodo fornisce in uscita un vettore $\mathbf{X}_{out}$ con valori binari (0 o 1) mediante la probabilità condizionata di generare l'uscita X_{out} dato X_{in} e Y.

Questi dati in ingresso vengono compressi mediante tale matrice di probabilità condizionata (cioè la probabilità di avere una certa uscita del nodo, noto il valore di ingresso) mantenendo invariata l'informazione ottenuta. Così facendo, non si fa altro che aggiornare, tra un nodo e l'altro, queste matrici di probabilità condizionata

trattandosi di un processo iterativo.

Nella *fase di test*, invece, il nodo riceve in ingresso solo il voto $\mathbf{X}_{in}$ (corrispondente alla sezione *test* della matrice). Mediante le matrici di probabilità condizionate calcolate nella fase di training, il nodo deve essere in grado di fare una stima ottimale sulle ideologie dei parlamentari fornendo in uscita un vettore $\mathbf{Y}_{stima}$. Quest'ultimo, confrontato con il vettore $\mathbf{Y}_{test}$, permette di calcolare l'errore sulla stima. L'ottimo si avrebbe qualora i due vettori coincidessero ottendendo un errore nullo.

L'intero dataset è dunque una matrice 435 x 17 in cui:

- **435**, corrisponde al numero totale di parlamentari;
- **17**, rappresentano le colonne della matrice in cui la prima è riferita alle uscite da stimare e le restanti 16 rappresentano il numero totali di leggi considerate.

2.3 Principi fondamentali dell'algoritmo

Il punto cardine dell'algoritmo sta nel determinare la matrice di probabilità condizionata P(Xout = j | Xin = i) sfruttando l'algoritmo di Blahut-Arimoto [4] [5]. Tale algoritmo è iterativo e converge molto rapidamente:

$$P(Xout = j|Xin = i) = \frac{P(Xout)e^{\beta d(i,j)}}{Z(i;\beta)}, i = 0, ..., Nin-1; j = 0..., Nout-1 \quad (2.1)$$

dove:

- **P(Xout = j)**, rappresenta la probabilità che si verifichi una certa uscita Xout, ed è ottenibile dalla seguente espressione:

$$P(Xout = j) = \sum_{i=0}^{Nin-1} P(Xin = i)P(Xout = j|Xin = i), i = 0, ..., Nin-1; j = 0..., Nout-1 \quad (2.2)$$

- **d(i,j)**, rappresenta il coefficiente di Kullback-Leibler [4]:

$$d(i,j) = \sum_{m=0}^{Nclass-1} P(Y = m|Xin = i) log_2 \frac{P(Y = m|Xin = i)}{P(Y = m|Xout = j)}, Nclass = 2 \quad (2.3)$$

- **P(Xin = i)**, rappresenta la probabilità che si verifichi un certo evento in ingresso;
- **P(Y=m|Xin=i)**, rappresenta la probabilità di ottenere una determinata stima sull'uscita noto a priori il valore di ingresso Xin;

- β è un parametro intero fissato a 2 in quanto per tale valore non si hanno escursioni eccessive sulle uscite;

- **P(Y=m|Xout=j)** rappresenta la probabilità di ottenere una determinata stima in uscita noto a priori il valore Xout:

$$P(Y = m|Xout = j) = \sum_{i=0}^{Nin-1} P(Y = m|Xin = i)P(Xin = i|Xout = j), \qquad (2.4)$$

con: m=0,...,Nclass-1; j=0,...,Nout-1

- **Z(i,β)**, rappresenta il coefficiente di normalizzazione il quale indica che la somma di tutti gli elementi delle matrici di probabilità deve essere uguale a 1 qualunque sia il valore di Xin:

$$\sum_{j=1}^{Nout-1} P(Xout = j|Xin = i) = 1 \qquad (2.5)$$

Come è possibile notare, le elaborazioni dei valori in ingresso che un nodo informativo esegue in fase di training, sono tutte funzioni non linari. Le varie matrici di probabilità verranno sfruttate nella fase di test per ottenere le stime sulle uscite.

2.4 Rete neurale DIN

L'intera rete neurale è costituita da $\log_2$(D)+1 layers dove D rappresenta il numero di colonne della matrice X del dataset di ingresso.

In questo lavoro di tesi è stato considerato il caso D=16 e 5 layers. In ogni layer sono presenti i nodi informativi che ricevono in ingresso i dati e forniscono le uscite sulla base delle elaborazioni elencate precedentemente.

Nella figura successiva è stato mostrato la rete neurale completa durante la fase di testing:

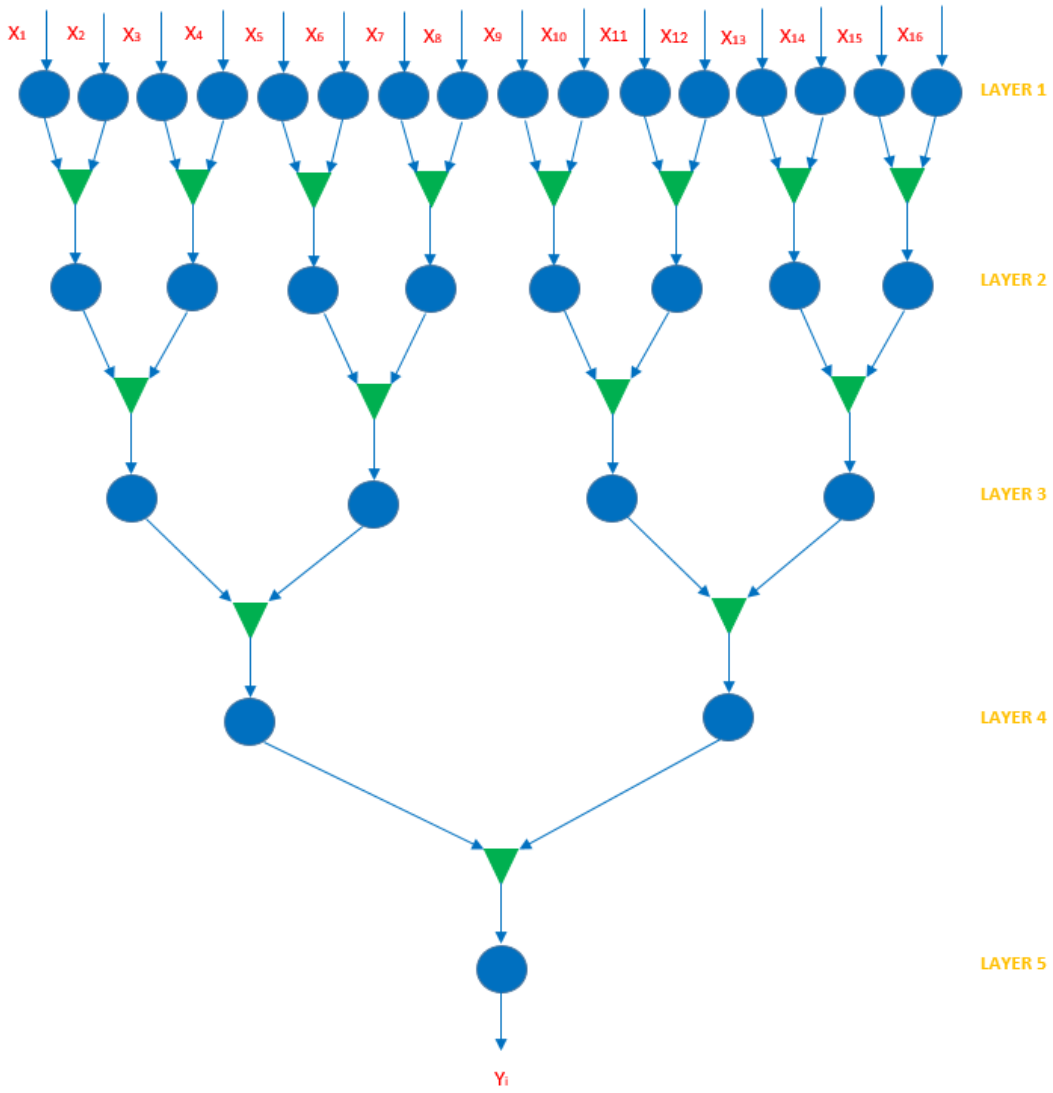

Figura 2.3. Struttura della rete neurale DIN in fase di testing

Come è possibile notare dalla figura, sono presenti 32 nodi (cerchi blu) e 5 layers. Ciascun processing element fornisce il dato elaborato nel layer successivo passando per un unità di concatenazione (triangolo verde). Quest'ultimo ha il compito di unire le informazioni provenienti da 2 nodi. Si arriva all'ultimo layer (layer 5) in cui il processing element finale, sulla base delle informazioni elaborate in tutti gli step precedenti, fornisce una stima binaria finale (**Yi**).

Nella fase di training, invece, ciascun nodo oltre a ricevere gli ingressi Xin riceve anche l'uscita nota Y.

Nella figura successiva è mostrato uno schema a blocchi inerente al flusso dei dati:

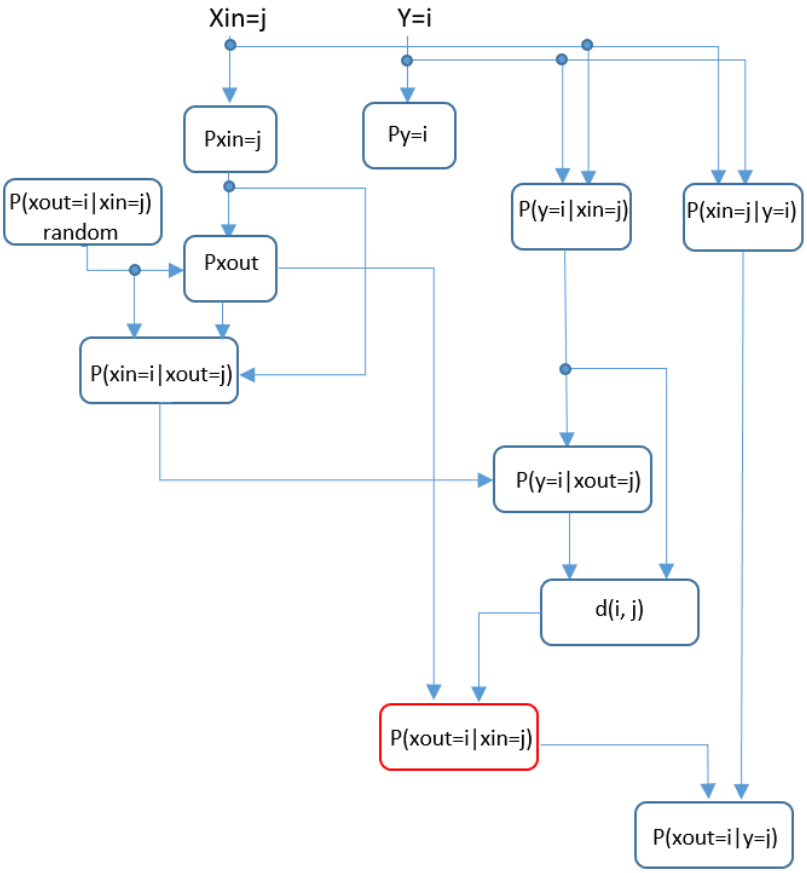

Figura 2.4. Dataflow: flusso dei dati

L'algoritmo di Blahut-Arimoto [4] [5] ha un comportamento di tipo dataflow e tramite la matrice di probabilità condizionata P(xout|xin) è possibile stimare le uscite Y (riquadro rosso).

2.5 Risultati del modello esatto

Nella figura 2.4, è mostrata la simulazione finale del modello esatto in *fase di training* per una singola estrazione dei dati di ingresso:

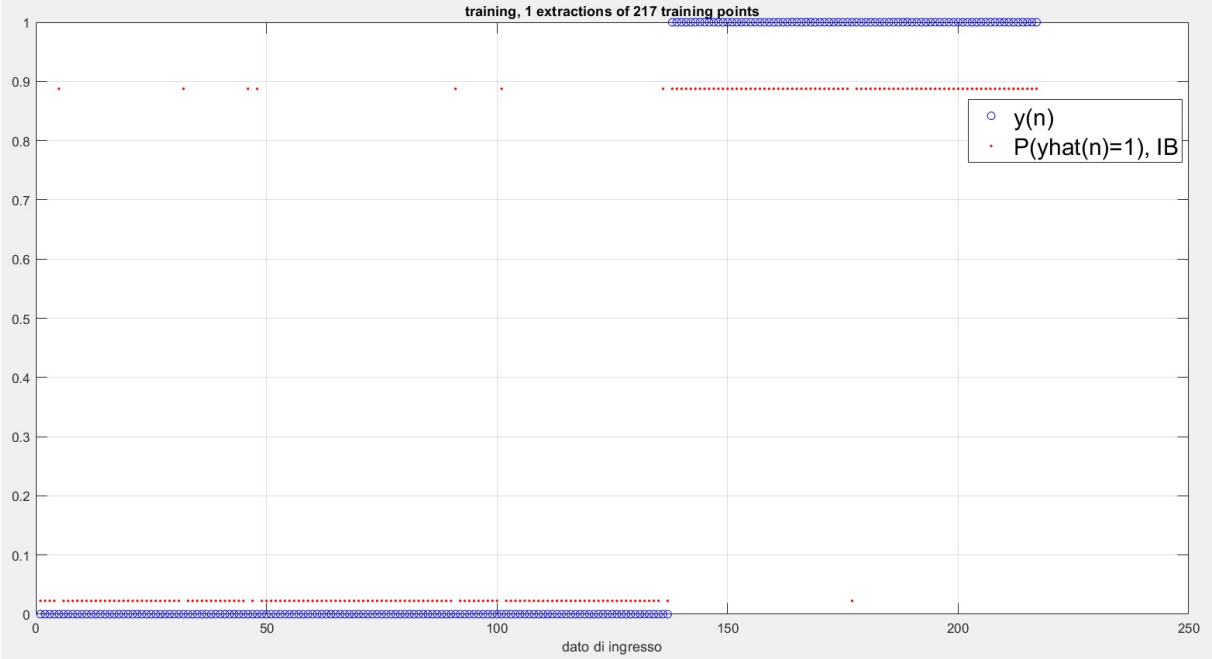

Figura 2.5. Training: singola estrazione dei dati di ingresso

Nella simulazione, sono stati rappresentati in blu i valori noti a priori dell'uscita Y (Y = 0, democratici e Y = 1, repubblicani). I punti in rosso, invece, rappresentano la probabilità di ottenere una stima sull'uscita Y = 1 (P(Yhat)=1), ovvero la probabilità che il parlamentare che ha votato sia repubblicano. Più tale valore è prossimo all'unità e più è probabile il verificarsi dell'evento Y=1. Viceversa, più è prossimo allo zero e più risulterà improbabile l'uscita di tale evento.

La simulazione è stata eseguita attraverso un'unica estrazione dei dati del dataset. Inoltre, dall'analisi degli andamenti si evince che, in fase di training, l'algoritmo ha "imparato" molto bene riuscendo a sbagliare solamente in 8 circostanze su 217 valori (figura 2.4). In particolare, nelle prime 7 ha fatto una stima sbagliata sull'uscita Y=0 (decidendo per Yhat=1) mentre nel'ultimo caso ha commesso un solo errore sulla stima di Y=1 (decidento per Yhat=0).

La figura successiva (figura 2.5), invece, mostra come si è comportato l'algoritmo nella *fase di testing*:

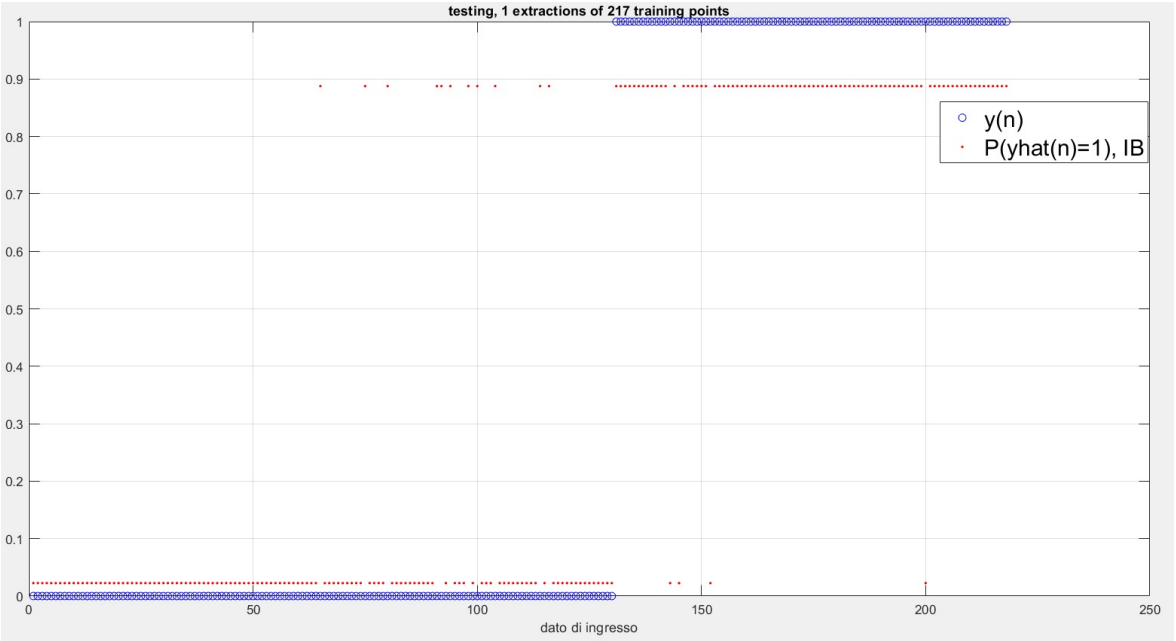

Figura 2.6. Testing: singola estrazione dei dati di ingresso

In quest'ultima simulazione, l'algoritmo ha reagito molto bene al test ricevendo in ingresso solo i dati delle votazioni (X) per una singola estrazione. Mediante le matrici di probabilità calcolate nella fase di training, il modello è riuscito a fare una stima piuttosto affidabile delle uscite sbagliando 12 volte in totale.

In seguito, è stata determinata la probabilità di errore sia in fase di training che di testing del modello:

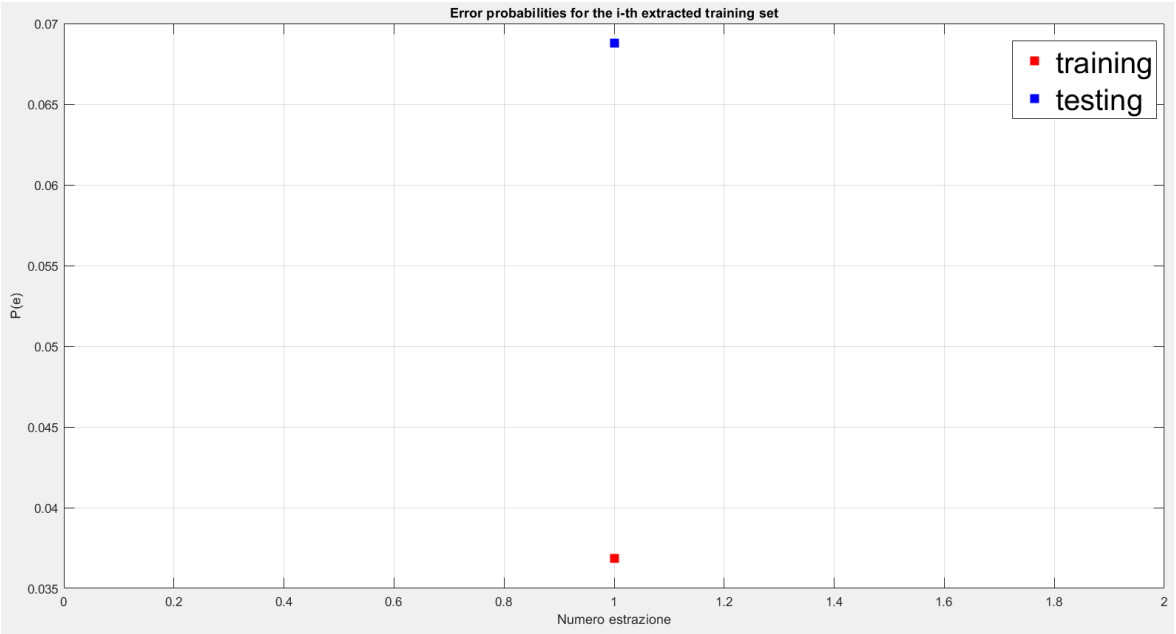

Figura 2.7. Probabilità di errore durante la fase di testing e training

Nella figura 2.6, è stata quantificata la probabilità di errore su una sola estrazione dei dati di ingresso. Il risultato finale è riportato in seguito:

- **testing phase**, P(e) = 5.85 %;
- **training phase**, P(e) = 3.2 %.

Capitolo 3

Modelli floating point e fixed point

Il modello esatto dell'algoritmo della rete neurale è stato realizzato sfruttando la doppia precisione di Matlab al fine di ottenere la massima precisione possibile.

3.1 Analisi dell'affidabilità dell'algoritmo

Uno step fondamentale ai fini delle prestazioni finali verte sull'analisi dell'affidabilità del modello in esame. Nello specifico, più la probabilità di ottenere una previsione corretta si avvicina a 1, più risulta essere efficiente. Al contrario, più la probabilità tende al valore 0, minore risulterà essere la sua affidabilità, aumentando di gran lunga gli errori finali. La probabilità di ottenere l'uscita esatta è stata indicata con **P(exact)**, la quale potrà assumere valori compresi tra 0 e 1.

L'intera architettura, in generale, effettua una stima dell'uscita in base ai dati che giungono in ingresso alla rete. Come è stato descritto precedentemente, l'intero dataset iniziale è stato definito da una matrice **X** con N righe e D colonne (features) con il corrispondente vettore delle classi **Y**. A sua volta, esso è stato suddiviso in 2 parti al fine di differenziare i dati necessari per il training è per il testing:

- $\mathbf{x}_{train}$, $\mathbf{y}_{train}$
- $\mathbf{x}_{test}$, $\mathbf{y}_{test}$

La figura sottostante, mostra nel dettaglio la divisione del dataset:

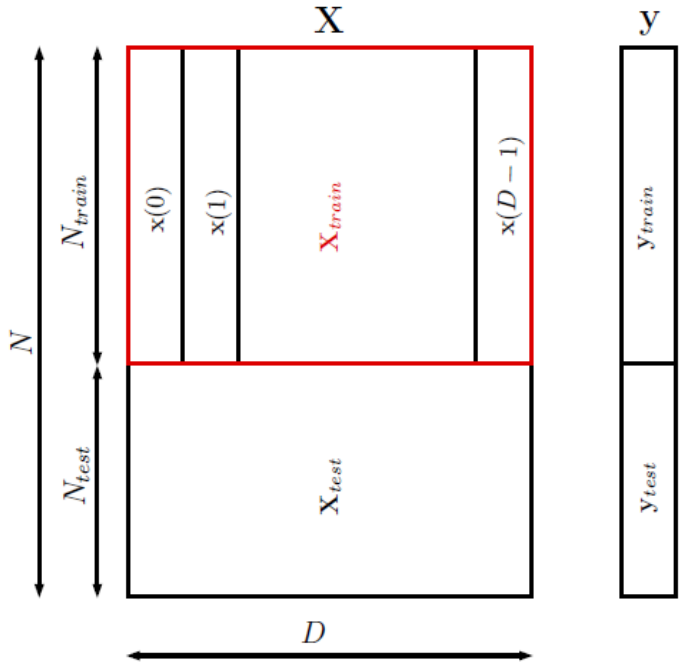

Figura 3.1. Dataset con relativa suddivisione [4]

I vettori $\mathbf{y}_{train}$ e $\mathbf{y}_{test}$ rappresentano rispettivamente le uscite esatte note a priori, mentre, le stime generate ed elaborate dal modello sono state definite dai seguenti vettori:

- $\mathbf{y}_{train\_hat\_av}$, per la fase di training
- $\mathbf{y}_{test\_hat\_av}$, per la fase di testing

La probabilità di commettere un errore in uscita, P(e), è stata calcolata confrontando il valore stimato con quello esatto, in entrambe le fasi. In questo modo, è stato possibile definire l'efficienza dell'algoritmo, la quale dipende dalla probabilità che la previsione sia corretta:

$$P(exact) = 1 - P(e) \tag{3.1}$$

L'obiettivo principale, dunque, è stato quello di cercare di ottenere tale valore quanto più prossimo all'unità per avere migliori prestazioni.

3.2 Modello fixed-point e relativa suddivisione in classi

Lo step successivo è stato quello di passare dal modello in floating point dell'algoritmo a quello in fixed point, suddividendolo opportunamente in sezioni aventi grado di parallelismo differente a seconda dei parametri in esame. La scelta migliore è stata quella di creare tre classi, ottenute in base ai valori attesi dal modello in floating point. L'obiettivo è stato quello di adottare un livello di parallelismo accettabile per ogni classe ed avere un comportamento quanto più vicino al modello esatto.

3.2.1 Classe 1

La prima classe è stata utilizzata per i *contatori di occorrenze* sia per gli ingressi X_i che Y. In generale, tale classe è stata estesa per tutte le variabili intere positive utilizzate all'interno dell'algoritmo. I simboli utilizzati sono stati i seguenti:

- n_i, parallelismo totale
- f_i, parte frazionaria degli n_i bits (f_i=0).

In questo modo, una generica variabile intera è stata rappresentata in fixed point senza segno con $n_i - f_i$ bits di parte intera e f_i di parte frazionaria.

3.2.2 Classe 2

Vista la presenza di tante operazioni tra matrici di *probabilità*, è stato necessario realizzare una seconda classe per gestire opportunamente queste variabili.
Per definizione, la probabilità che un certo evento si verifichi, è rappresentato da un numero compreso tra 0 e 1. Proprio per questo motivo, è stata scelta una classe appropriata al fine di evitare una loro rappresentazione in floating point.

In particolare, i simboli utilizzati per tale classe sono stati i seguenti:

- n_p parallelismo totale;
- f_p parte frazionaria degli n_p bits.

Così facendo, una variabile che rappresenta una certa probabilità che si verifichi un evento piuttosto che un'altro, è stata rappresentata in fixed point senza segno con $n_p - f_p$ bit di parte intera e f_p di parte frazionaria. In questo caso particolare, tali variabili necessitano di un solo bit per la parte intera e molti più bit per la parte frazionaria.

3.2.3 Classe 3

L'ultima classe dell'algoritmo è stata creata per gestire il fattore *d(i,j)* chiamato Kullback-Leibler divergence. Poichè tale parametro dipende dalla seguente quantità, $\log_2 \frac{P(Y=m|Xin=i)}{P(Y=m|Xout=j)}$, occorre gestire il suo parallelismo in modo diverso rispetto alle classi passate, in quanto il risultato potrebbe essere un numero decimale positivo o negativo.

I simboli utilizzati per tale classe sono stati i seguenti:

- nd_p parallelismo totale;
- fd_p parte frazionaria degli nd_p bit.

In questo caso, tale coefficiente è stato rappresento in fixed point con segno con $\mathrm{nd}_p - \mathrm{fd}_p$ bit di parte intera e fd_p di parte frazionaria.

3.3 Scelta del parallelismo per ciascuna classe

L'obiettivo principale è determinato dalla scelta ottimale del parallelismo interno dell'architettura valutando i margini di errore rispetto al modello esatto. È stato scelto come modello di riferimento l'algoritmo in floating point , il quale è stato confrontato con l'architettura in fixed point variando di volta in volta la dinamica dei bit di ogni classe.

3.3.1 Parallelismo classe 1

Al fine di determinare il parallelismo corretto per la prima classe, ovvero per le variabili intere che incidono nell'elaborazione del risultato finale, è stata effettuata una simulazione confrontando l'uscita dei due modelli di partenza. Il risultato ottenuto è stato riassunto in un grafico che tiene conto della probabilità di uscita corretta P(exact) del modello esatto rispetto a quella del modello fixed point. Ovviamente, è stata variata la dinamica del parametro n_i fissando al valore massimo quella delle altre classi al fine di analizzare la sola dipendenza dell'algoritmo in fixed point da n_i.

L'operazione è stata impiegata sia per la fase di training che per la fase di testing dell'intero sistema.
La figura sottostante mostra l'andamento appena descritto:

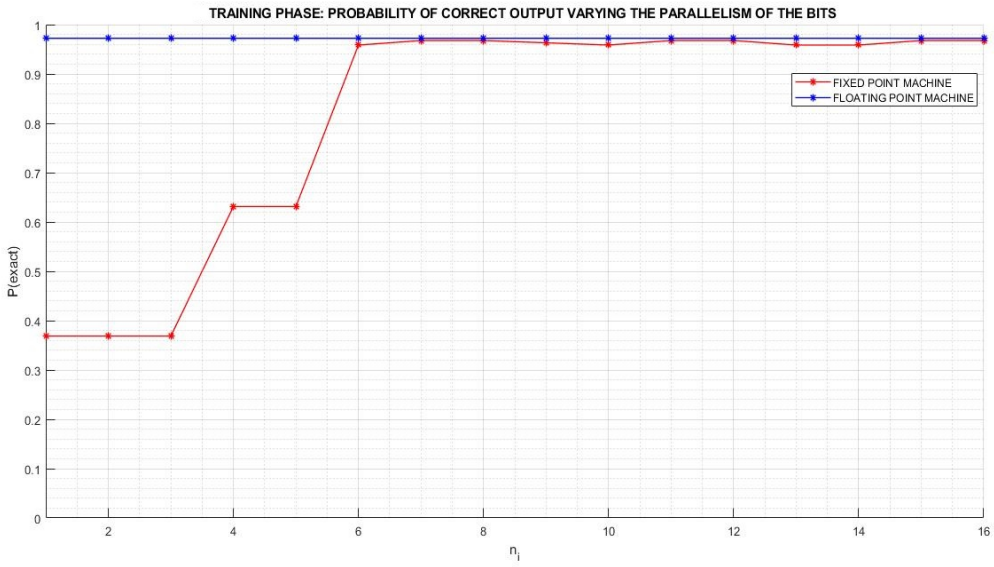

Figura 3.2. TRAINING PHASE: Confronto dei due modelli al variare del parametro n_i

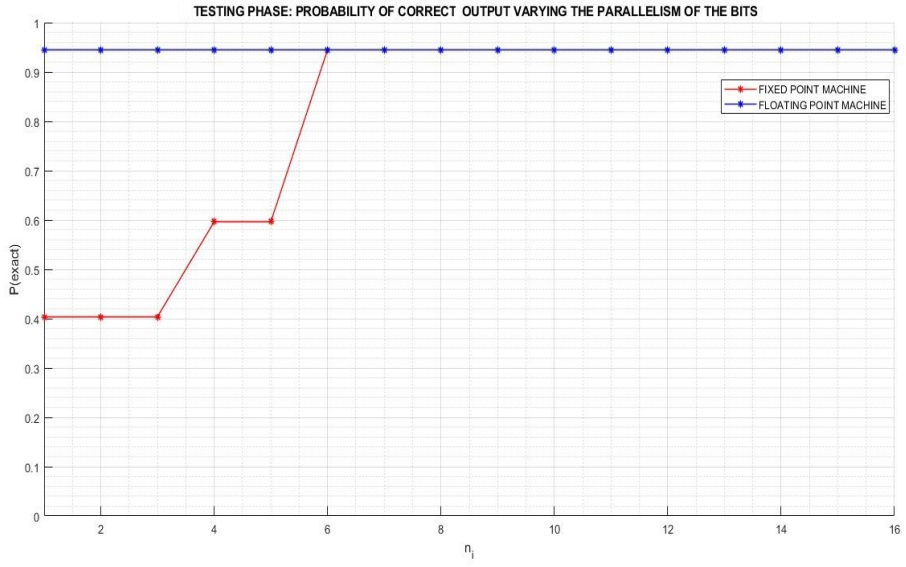

Figura 3.3. TESTING PHASE: Confronto dei due modelli al variare del parametro n_i

Si nota che il modello esatto è del tutto indipendente dalla variazione del numero di bit del parametro n_i, dando una probabilità di uscita corretta prossima ad 1. Invece, la macchina in fixed point dipende fortemente dalla dinamica di tale parametro per cui per un numero molto basso di bit di n_i, l'algoritmo non è in grado di prendere la giusta decisione sull'uscita finale (Pexact $\simeq$0.4). All'aumentare della dinamica aumenta nettamente la probabilità di ottenere l'uscita corretta convergendo pressochè al valore esatto in cui Pexact = 1.

In base ai grafici in esame, le scelte del parallelismo sono le seguenti:

- $n_i = 8$
- $f_i = 0$ (avendo considerato variabili intere)

Con questa scelta, l'algoritmo ha una probabilità molto alta di prendere la corretta decisione sull'uscita finale.

3.3.2 Parallelismo classe 2

Il parallelismo della seconda classe, nonchè per tutte quelle variabili probabilistiche, è stata utitlizzata la medesima tecnica precedente con la consapevolezza che tali variabili hanno bisogno di un solo bit per la parte intera e i restanti per la parte frazionaria. Anche in questo caso, la dinamica dei parametri delle altre classi sono stati fissati al massimo range possibile tranne il termine n_i, il quale è stato mantenuto al valore ottimale scelto precedentemente (n_i=8).

Il confronto sia in fase di training che in fase di testing è mostrato nelle seguenti figure:

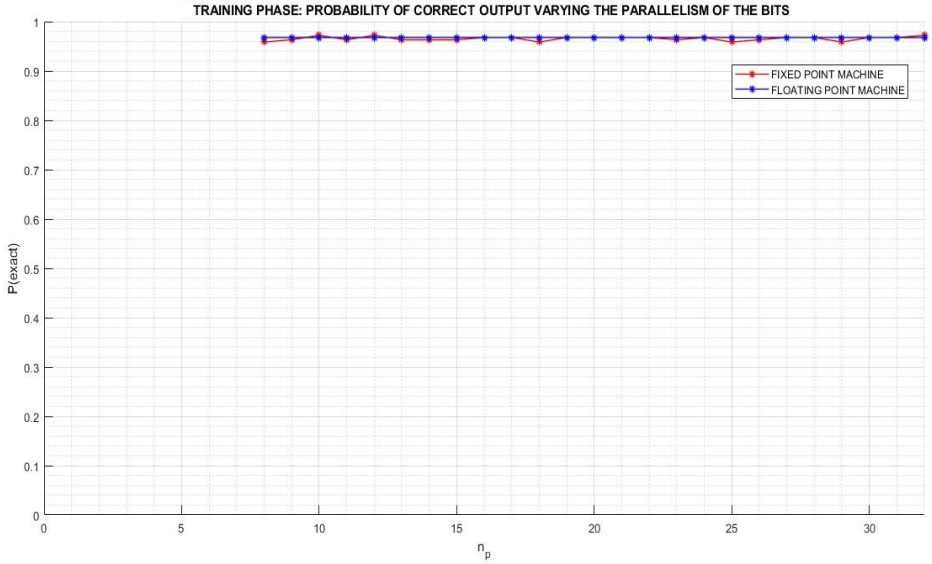

Figura 3.4. TRAINING PHASE: Confronto dei due modelli al variare del parametro n_p con f_p=n_p−1

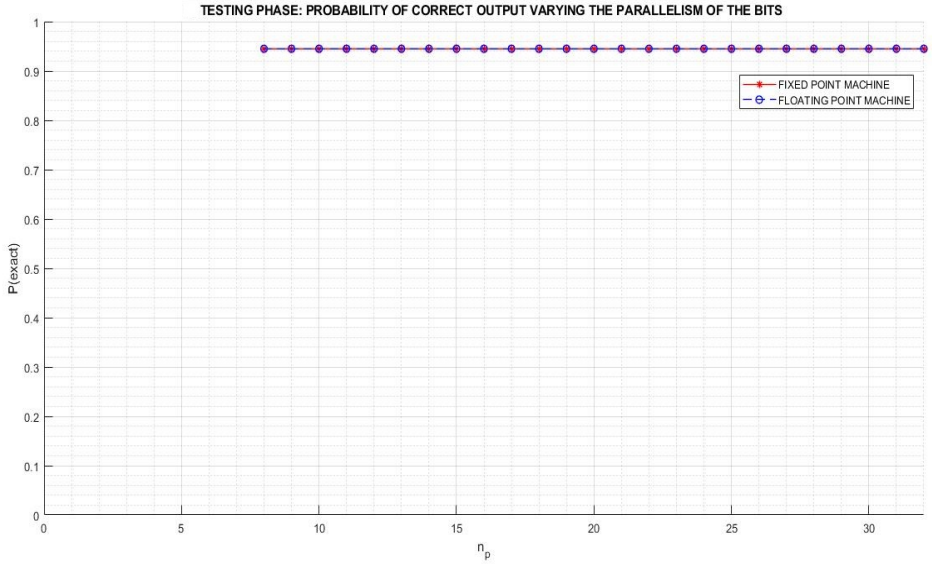

Figura 3.5. TESTING PHASE: Confronto dei due modelli al variare del parametro n_p con f_p=n_p−1

In questo caso specifico, il minimo numero di bit utilizzato per rappresentare la probabilità è n_p=8 (con f_p=n_p−1) poichè per dinamiche inferiori l'algoritmo prende decisioni completamente errate e non rappresentabili analiticamente. Aumentando la dinamica di n_p, è possibile notare come la probabilità di commettere un errore diminuisce notevolmente già con soli 8 bit (Pexact ≃1).

La decisione ottimale sul parallelismo per la seconda classe è stata la seguente:

- n_p = 16 bit
- f_p = 15 bit
- n_i = 8 bit
- f_i = 0 bit

3.3.3 Parallelismo classe 3

Riguardo il parallelismo per la classe relativa al parametro di Kullback-Leibler divergence, gli andamenti sottostanti mostrano il criterio di scelta opportuno. In particolare, le figure 3.6 e 3.7 mostrano le fasi di training e di testing al variare del numero di bit della parte frazionaria mantenendo fissa la dinamica complessiva (nd_p) al massimo valore possibile. I parametri delle classi precedenti sono stati mantenuti fissi al loro valore ottimale scelto precedentemente.

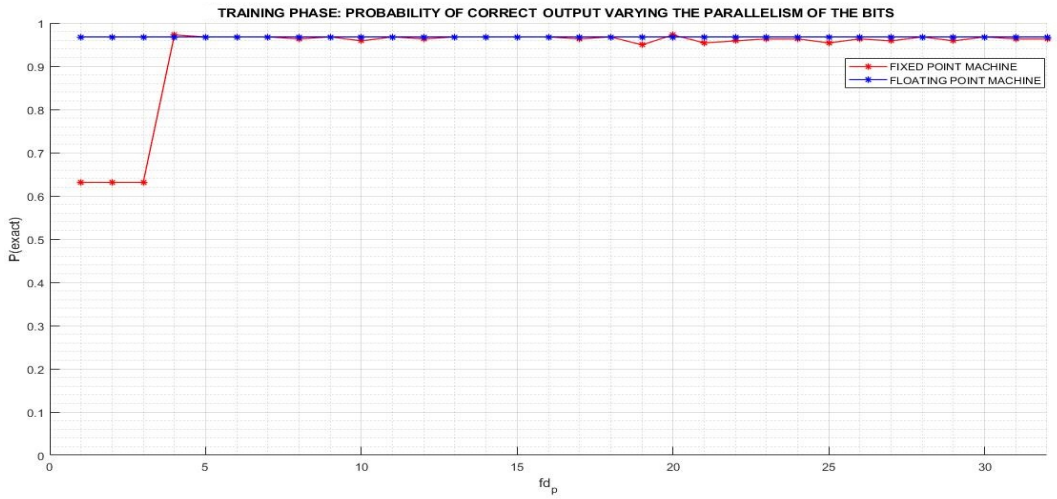

Figura 3.6. TRAINING PHASE: Confronto dei due modelli al variare del parametro fd_p con nd_p fissato

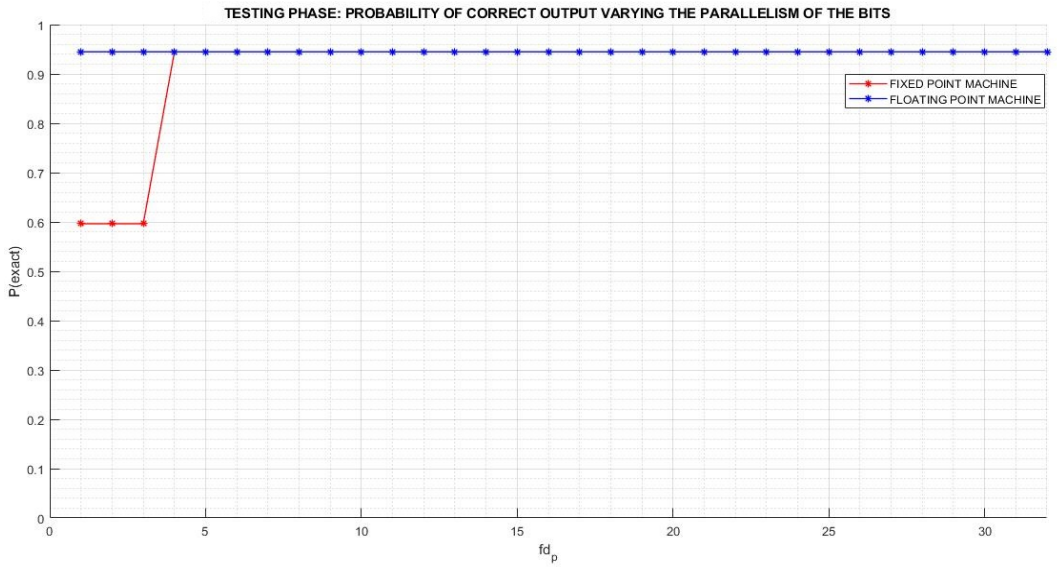

Figura 3.7. TESTING PHASE: Confronto dei due modelli al variare del parametro fd_p con nd_p fissato

Nel caso specifico, si ha un'ottima risposta per fd_p>4 bit ed è più opportuno scegliere fd_p=10 bit. Si riportano i range di parallelismo scelti fino a questo punto:

- $fd_p = 10$ bit
- $n_p = 16$ bit
- $f_p = 15$ bit
- $n_i = 8$ bit
- $f_i = 0$ bit

Una volta noto il valore ottimale della parte frazionaria, si è deciso di variare solo la parte intera, fissando il valore ottimale di fd_p=10 bit e analizzato il comportamento dell'algoritmo al variare della parte intera (ottenuta aumentando l'intera dinamica nd_p). I restanti parametri delle altre classi sono stati mantenuti costanti al loro valore ottimale calcolato precedentemente.
Le figure 3.8 e 3.9 mostrano i risultati delle simulazioni sia nella fase di training che in quella di testing:

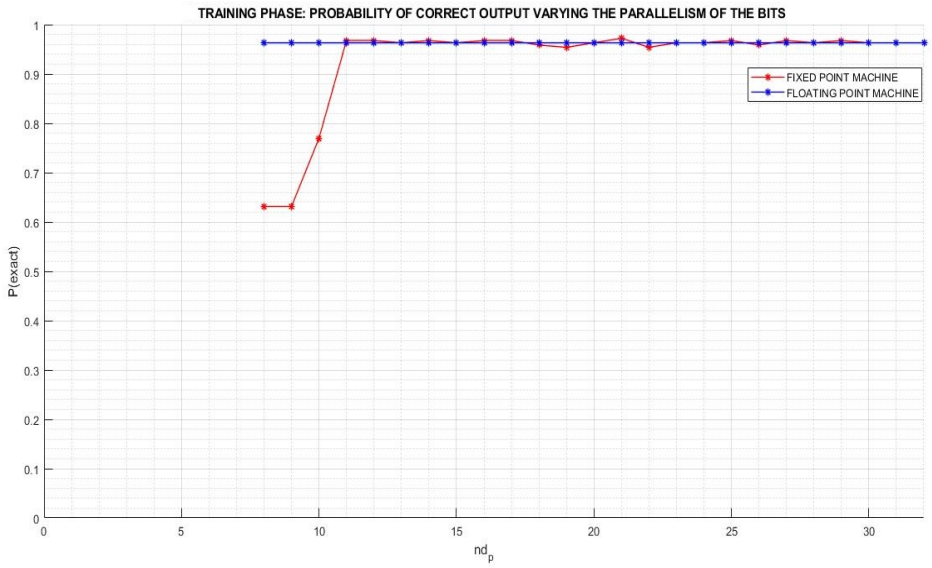

Figura 3.8. TRAINING PHASE: Confronto dei due modelli al variare del parametro nd_p con fd_p ottimale

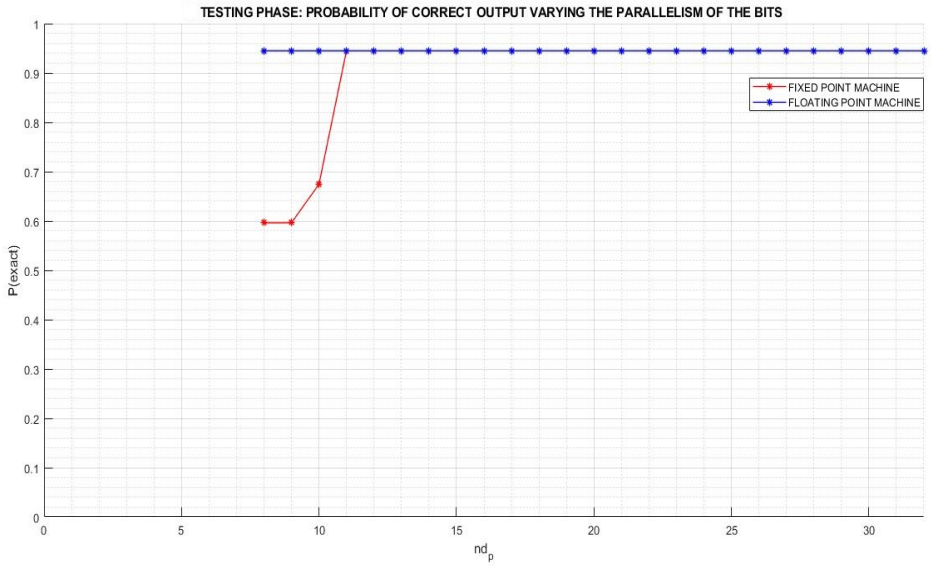

Figura 3.9. TESTING PHASE: Confronto dei due modelli al variare del parametro nd_p con fd_p ottimale

In questo caso, i valori accettabili per la dinamica complessiva del parametro 'd' devono essere $nd_p \geq fd_p$ al fine di ottenere una corretta stima. Considerando che si tratta di un parametro con segno, la scelta ottimale è ricaduta sui 16 bit totali (nd_p=16 bit). In questo modo, la matrice 'd' è stata dimensionata con nd_p - fd_p = 6 bit di parte intera e fd_p= 10 di parte frazionaria.

In conclusione, la scelta finale del parallelismo interno dell'architettura in fixed point per ciascuna classe è stata la seguente:

	CLASSE 1
n_i [bit]	8
f_i [bit]	0

	CLASSE 2
n_p [bit]	16
f_p [bit]	15

	CLASSE 3
nd_p [bit]	16
fd_p [bit]	10

Lo step finale consiste nell'analizzare il comportamento dei due modelli adottando un meccanismo di shuffling dei dati in ingresso. La simulazione seguente mostra l'andamento di P(exact) per ogni estrazione casuale del dataset (iextract) e per il modello in fixed point, ovviamente, è stato fissato il parallelismo interno, calcolato precedentemente:

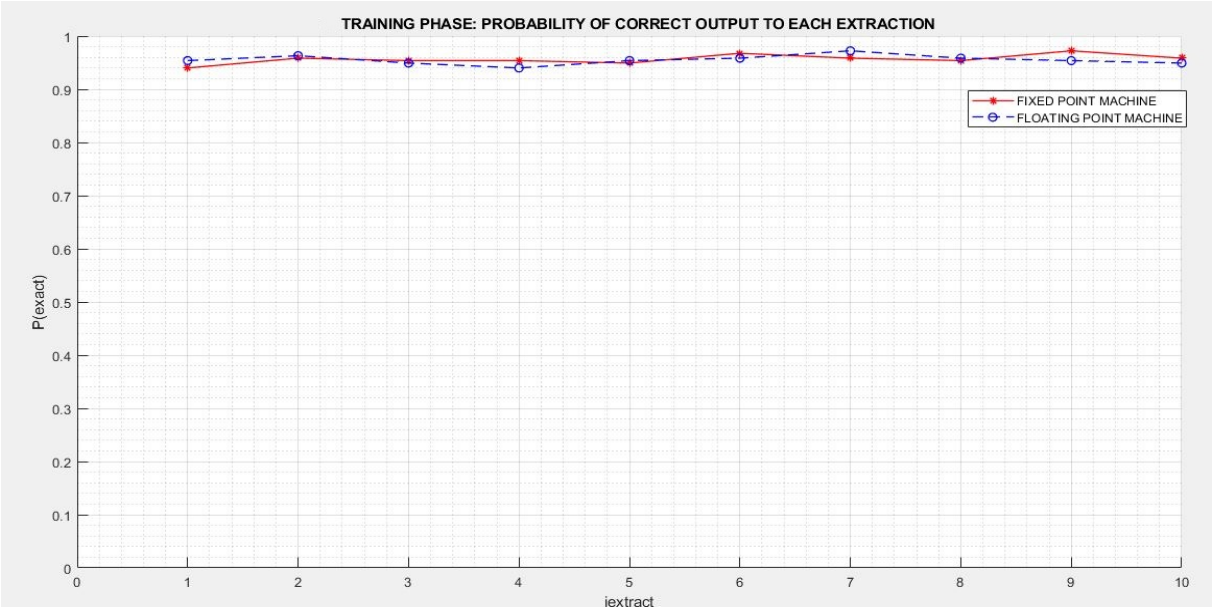

Figura 3.10. TRAINING PHASE: Confronto dei due modelli ad ogni estrazione del dataset

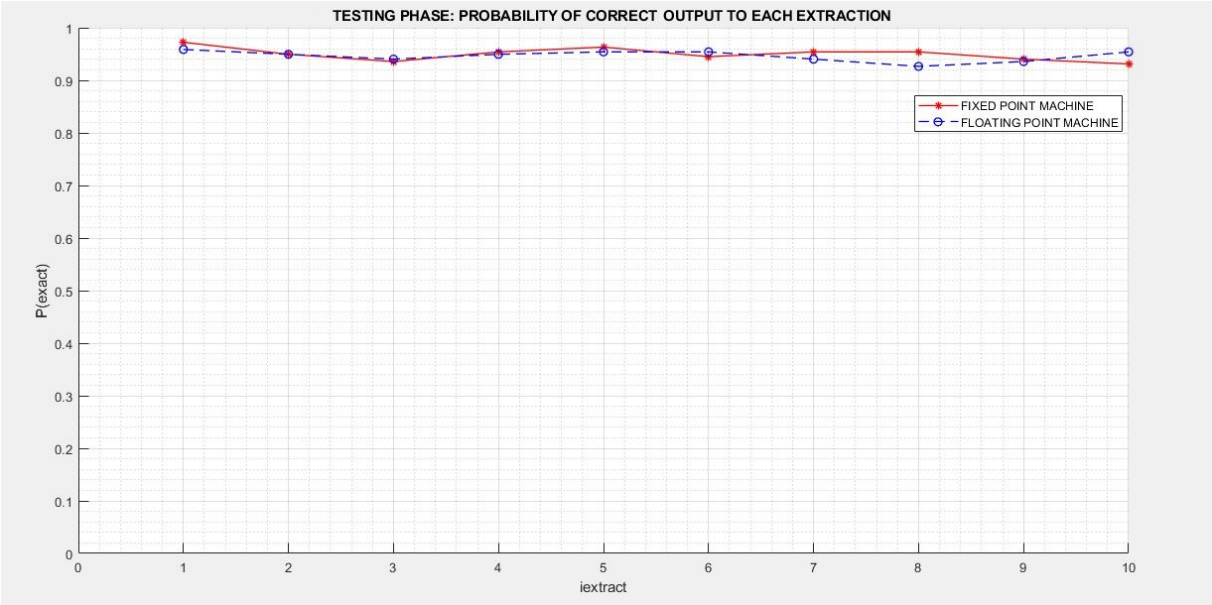

Figura 3.11. TESTING PHASE: Confronto dei due modelli ad ogni estrazione del dataset

Nelle precedenti figure, la probabilità di ottenere l'uscita corretta, sia in fase di training che di testing, è prossima a 1 per entrambi i modelli validandola scelta sul parallelismo interno dei dati.

3.4 Risultati finali fixed point

Con le scelte adottate precedentemente, si conclude che il modello in fixed point ha avuto pressochè lo stesso comportamento del modello floating con un errore trascurabile. Lo dimostra anche l'immagine seguente in cui la stima finale dell'intera rete neurale del modello in fixed point coincide con il modello esatto in floating point:

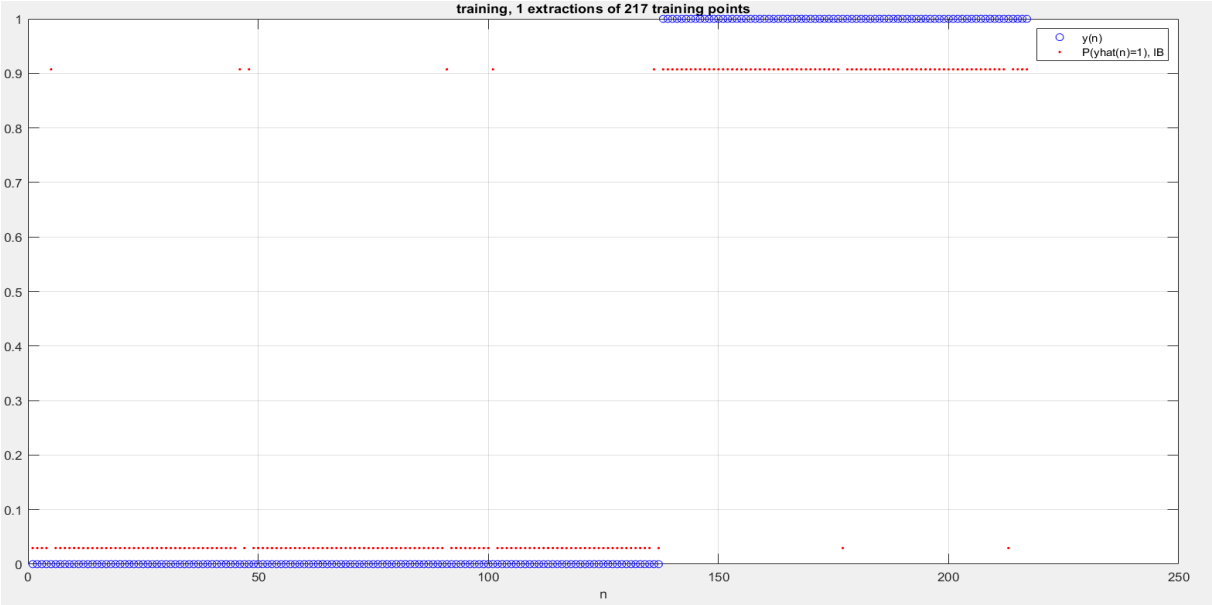

Figura 3.12. Fase di training per il modello in fixed point

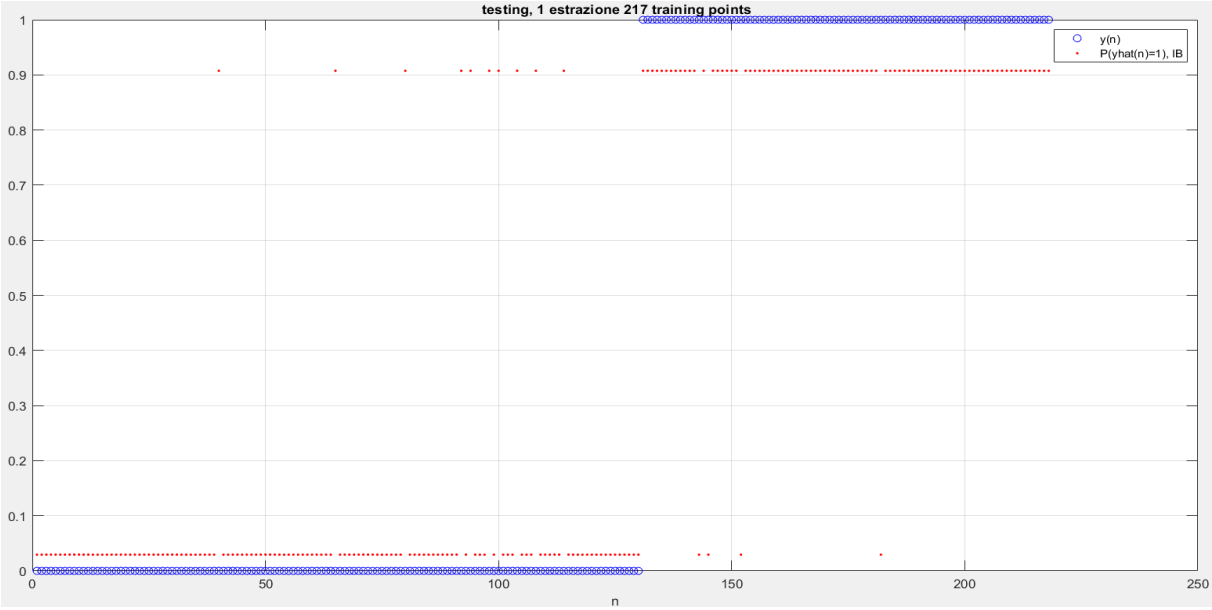

Figura 3.13. Fase di testing per il modello in fixed point

Capitolo 4

Implementazione hardware - Datapath

La fase principale di questo lavoro di tesi si focalizza su una possibile soluzione hardware per l'implementazione del nodo. Nello specifico, l'implementazione è stata incentrata solo sulla parte iniziale dell'elaborazione dei dati di ingresso. Dunque, ci si è limitati ad implementare l'algoritmo di generazione delle matrici di probabilità e probabilità condizionata lasciando la restante parte per un altro lavoro futuro.

Nel senso di avanzamento dei dati all'interno di un singolo nodo, l'algoritmo DIN presenta un comportamento data flow. Ragion per cui, è stato deciso di suddividerlo in tre step ciascuno dei quali ha generato dei risultati parziali utili per gli step successivi. Dunque, l'elaborazione del nodo è stata scomposta in più sottosezioni ciscuna con una diversa complessità percentuale sul totale (100%):

- **STEP 1**, generazione matrici di probabilità e probabilità condizionate, 40%;
- **STEP 2**, elaborazioni dei dati in funzione della matrici di probabilità calcolate nello step 1, 40%;
- **STEP 3**, parte finale delle elaborazioni in cui sono state generate le stime finali sulla base del dataset in ingresso, 20%.

Tali percentuali sono state ottenute dal modello Matlab in base ai tempi di esecuzione dei tre step. Si è ritenuto che un tempo di esecuzione maggiore implicasse una maggiore complessità nell'elaborazione.

Il lavoro di tesi si è focalizzato sullo studio di un'architettura hardware efficiente per lo STEP 1 durante la fase di training.

4.1 Quantizzazione dati di ingresso

Prima di procedere all'implementazione hardware della parte specifica del singolo nodo, è stato affrontata la quantizzazione dei dati in ingresso.

Poichè, come anticipato precedentemente, è stato trattato un caso puramente ternario, è stato deciso di quantizzare con 2 bit i dati presenti nel dataset. In seguito è stata schematizzata la quantizzazione adottata:

- **00**, corrisponde al voto favorevole di una legge,
- **01**, corrisponde al voto contrario di una legge,
- **10**, corrisponde al caso in cui un parlamentare si sia astenuto per diverse ragioni al voto,
- **11**, caso non presente in quanto non corrisponde a nessuna azione.

Dunque il dataset considerato è stato riportato nella figura seguente:

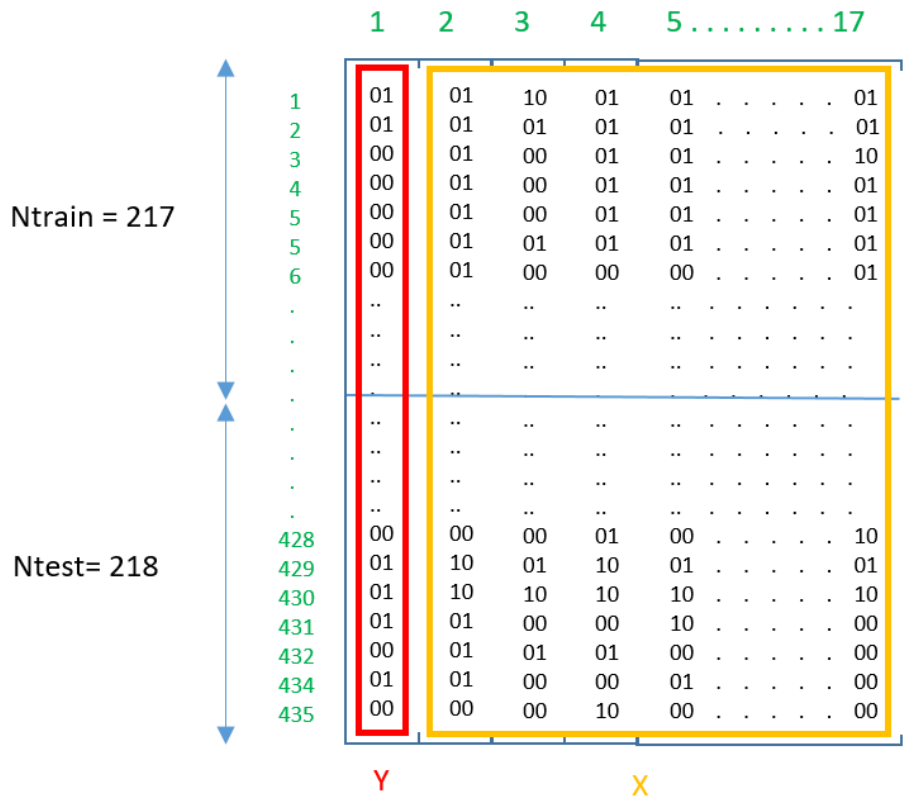

Figura 4.1. Dataset quantizzato

Inoltre, poichè lo studio è stato incentrato su un singolo processing element, i dati che sono stati presi in considerazione come input corrispondono a quelli di una singola colonna sia per Y che per X. Per semplicità di gestione, è stata analizzata solo la prima colonna della matrice **X** e l'unica colonna del vettore **Y** facendo riferimento esclusivamente alla fase di training. Così facendo, si sono ottenuti i vettori colonna X_1 e Y_1 costituiti da 217 elementi ciascuno (consideranto la fase di training). Questi vettori corrispondono rispettivamente a:

- $\mathbf{X}_1$, serie di voti dei 217 parlamentari americani per una determinata legge,
- $\mathbf{Y}_1$, valori noti delle ideologie dei 217 parlamentari (democratici o repubblicani).

Nella figura seguente sono stati riportati, sulla base delle valutazioni fatte precedentemente, i vettori di ingresso al singolo nodo durante la fase di apprendimento:

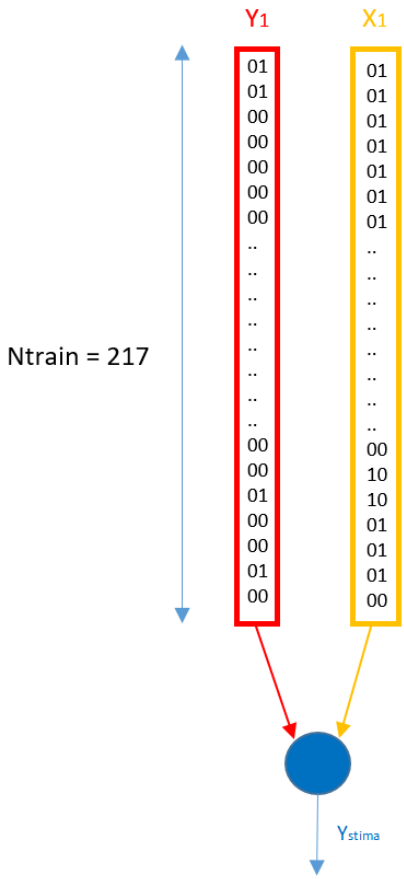

Figura 4.2. Fase di training: Vettori di ingresso al processing element

4.1.1 Gestione parallelismo interno

Come è stato affermato precedentemente, l'implementazione hardware è stata realizzata partendo dalle informazioni note riguardo il parallelismo interno calcolato nel modello fixed point.

La probabilità di un evento, per definizione, è sempre un numero compreso tra 0 e 1 e poichè l'algoritmo presenta delle non linearità nel processing, si è deciso di convertire tale quantità in un numero intero introducendo un fattore di scala (pari a 2^7) seguito poi da un arrotondamento opportuno al fine di passare a numeri interi facilmente gestibili nel processing.

In seguito è stato mostrato un esempio di gestione del parallelismo interno:

Sia data una probabilità che si verifichi un evento A pari a P(A)=0,4134567. Tale quantità, per i motivi spiegati nel capitolo 3, è stata gestita con $n_p = 16$ e $f_p = 15$:

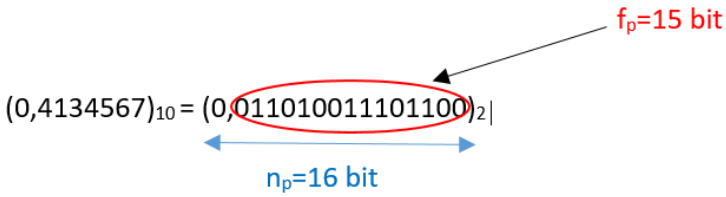

Figura 4.3. Esempio di conversione binaria di un valore probabilistico

I valori binari delle probabilità rappresentano quantità molto importanti per l'architettura hardware e per questo motivo si è scelta una strada più efficiente per la loro gestione. A tale scopo, come detto precedentemente, si è pensato di gestirle come numeri interi introducendo un fattore di scala di 2^7 seguito da un arrotondamento:

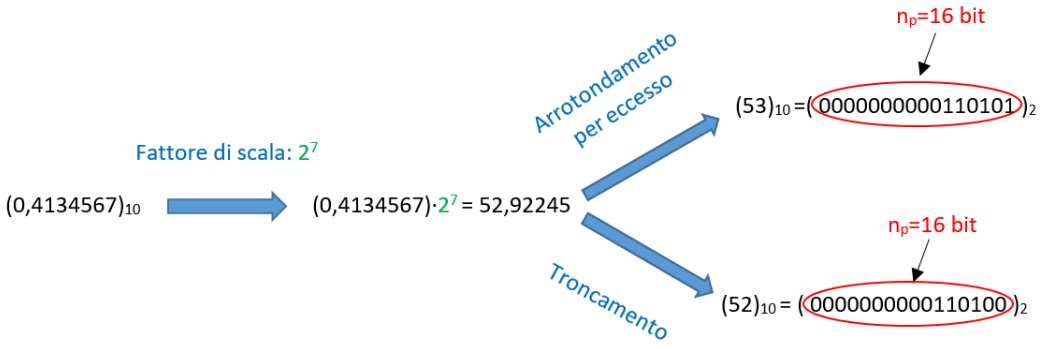

Figura 4.4. Esempio di conversione binaria di un valore probabilistico

Come è possibile vedere in figura 4.4, introducendo il fattore di scala e procedendo con un arrotondamento, si è passato dalla rappresentazione frazionaria ad una intera.

La scelta del parametro $\mathbf{2^7}$ è stata dettata dall'evitare un eventuale overrange su calcoli successivi basandoci sempre sul parallelismo ottimale calcolato precedentemente (16 bit).

4.1.2 Tecniche di arrotondamento e relativo confronto

Non trascurabili sono le tecniche di arrotondamento con l'obiettivo di eliminare la parte frazionaria del numero e trattare solamente la parte intera:

- **Arrotondamento per eccesso**
- **Troncamento**

Queste operazioni sono state necessarie a valle di unità aritmetiche. La scelta sulla tecnica più efficiente è dettata dalla minore influenza sui calcoli eseguiti dall'algoritmo per la decisione sulla stima finale. In particolare, è stato eseguito un confrontro tra l'algoritmo di riferimento, il cosiddetto modello esatto, e lo stesso algoritmo in cui sono state applicate le 2 tecniche di arrotondamento.

La figura sottostante mostra le differenze:

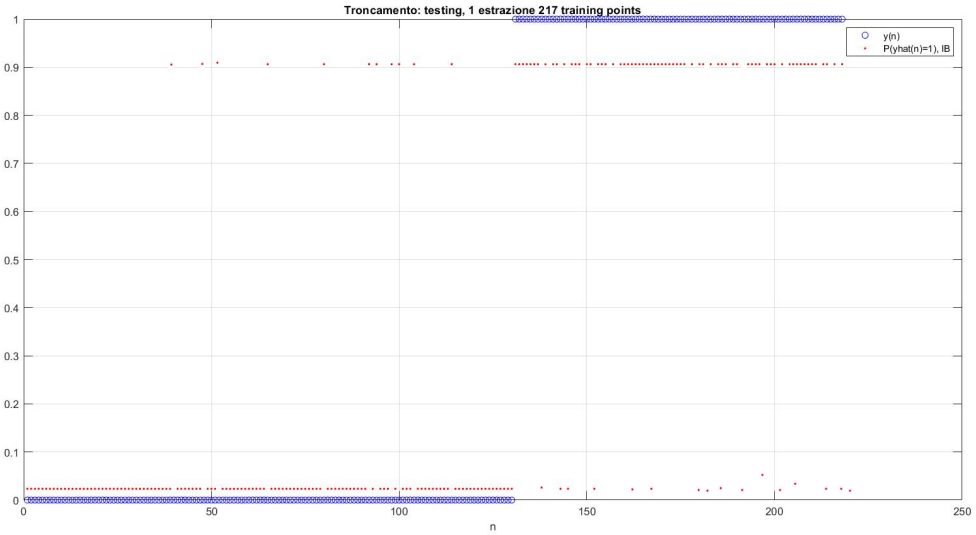

Figura 4.5. Risultato finale in fase di training mediante il troncamento dei dati

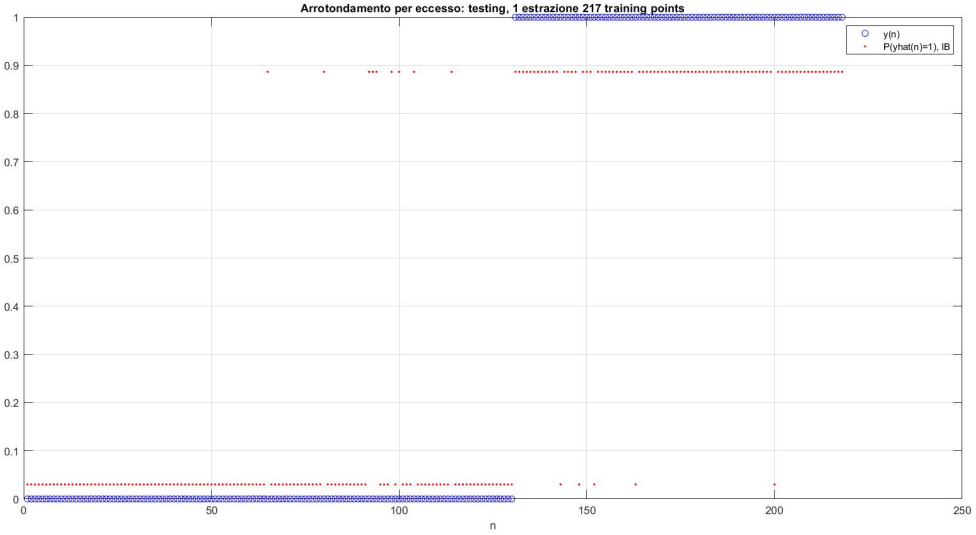

Figura 4.6. Risultato finale in fase di training mediante arrotondamento per eccesso dei dati

Tra questi due metodi, la scelta più adatta ricade nella tecnica di arrotondamento per eccesso in quanto l'errore sulla stima finale risulta essere minore rispetto all'altra

tecnica. Ciò è anche dovuto al fatto che i valori probabilistici di partenza sono approssimati dal momento che il modello è in fixed point, e dunque non potrebbero concidere esattamente con i valori reali del modello Matlab in floating point.

4.2 Memoria Off-chip

La memoria off-chip è l'elemento principale per iniziare l'implementazione. Tale unità, ha il compito di leggere il contenuto di un *file.txt* contenente l'intero dataset di dati, ed effettuare una scrittura nella memoria stessa per essere pronta alla lettura. Infatti, è stata eseguita una scrittura iniziale in memoria attraverso i dati contenuti nel file di testo, e successivamente sono state effettute continue letture delle sue locazioni. Poichè è stato considerato un singolo nodo, il dataset di dati che è stato preso in considerazione è costituito solamente dalle prime 2 colonne della fase di training. La figura sottostante mostra l'interazione della memoria off chip con il data set:

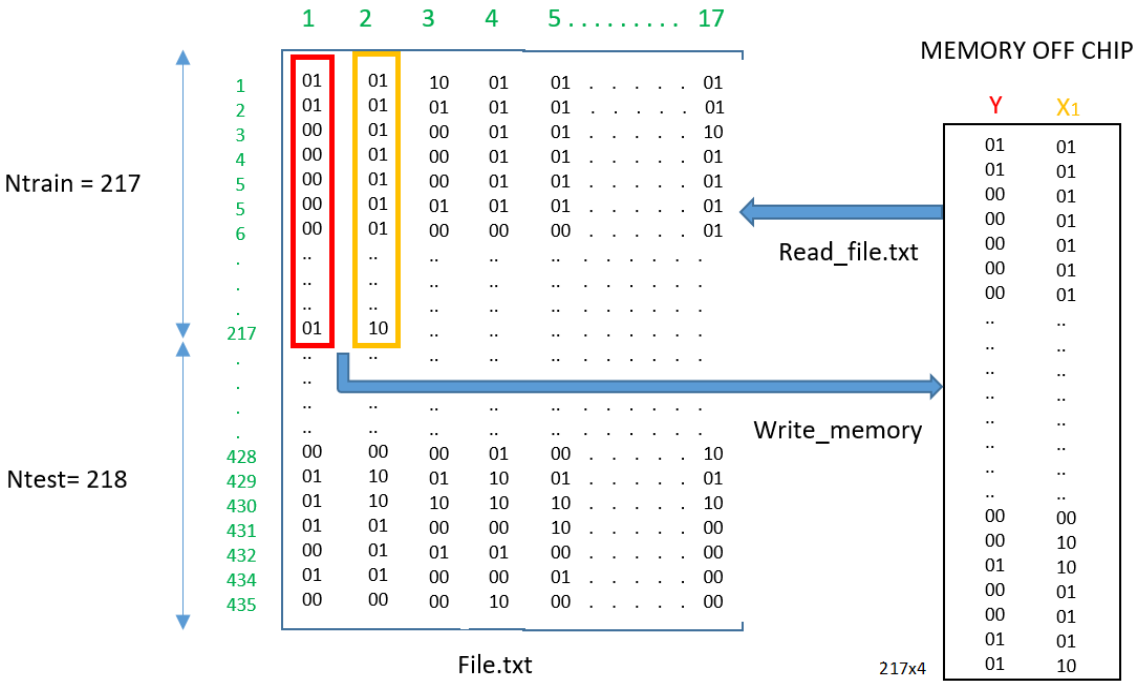

Figura 4.7. Comunicazione tra la memoria off-chip e il dataset .txt

L'intero dataset contenuto in un file di testo (file.txt) è costituito da 17 colonne (ciascuna di 2 bit), di cui la prima fa riferimento alla stima sui parlamentari (colonna rossa), mentre le altre corrispondono alle votazioni in merito a 16 leggi proposte. Nel nostro caso, tuttavia, sono state considerate solamente le prime 2 colonne trattandosi di uno studio di un singolo nodo:

- **Y**, vettore colonna contenente a priori le informazioni binarie sulle ideologie dei parlamentari (00 = repubblicano, 01 = democratico). Queste informazioni sono di fondamentale importanza durante la fase di apprendimento al fine di migliorare l'efficienza sulla stima finale durante la fase di test;

- **X1**, vettore colonna contenente le informazioni binarie circa l'esito delle votazioni dei parlamentari su una determinata legge (00 = favorevole, 01 = contrario, 10 = astenuto).

I due vettori colonna hanno dimensione 217 x 2 poichè è stata analizzata solo la fase di training su un dataset di 217 votanti per una singola legge.

Dal punto di vista hardware la memoria presenta un'abilitazione per la fase di training ed è sincrona al clock, ciò significa che ad ogni fronte di salita del clock, è stato ottenuto in uscita un vettore riga costituito da 4 bit, 2 per la Y e 2 per la X. Nella figura seguente si mostra la sequenza di output sincrono al segnale di sincronismo:

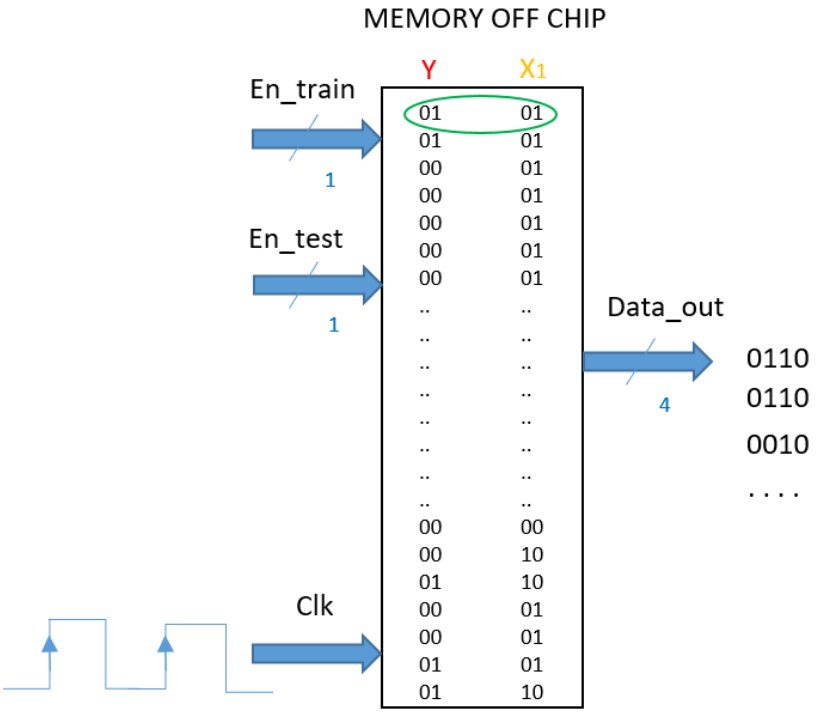

Figura 4.8. Memoria off-chip

Per leggere tutti gli elementi della memoria sono serviti 217 colpi di clock.

4.3 Memoria On-chip

Le memorie on-chip sono state pensate per memorizzare i dati in uscita dalla memoria off-chip in quanto risultano fondamentali per le fasi successive dell'algoritmo. Infatti, poichè è risultato necessario leggere più volte i valori dei dati in ingresso, è stato deciso di implementare due memorie on-chip che contenessero tali valori. In particolare:

- $\mathbf{Y}_{train}$, è una memoria on-chip che memorizza l'intero vettore colonna Y fornito dalla memoria off-chip. La scrittura di tale memoria avviene in parallelo con la lettura della memoria off-chip (considerando un colpo di clock come delay);
- $\mathbf{X}_{train}$,è una memoria on-chip che memorizza l'intero vettore colonna X1 fornito dalla memoria off-chip. La scrittura di tale memoria avviene in parallelo con la lettura della memoria off-chip (considerando un colpo di clock come delay).

Nella figura sottostante, è stata mostrata la scelta adottata:

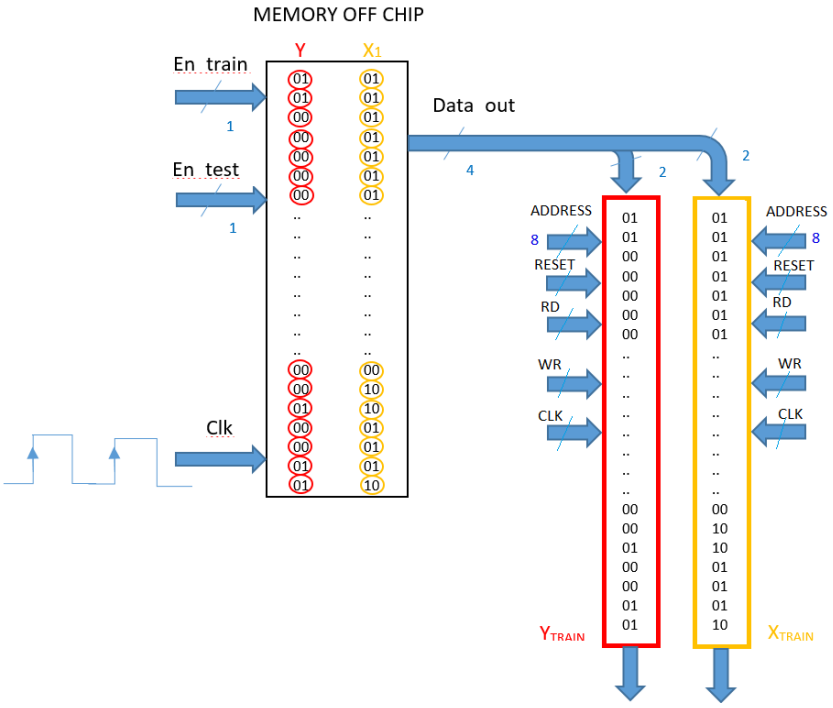

Figura 4.9. Memorie on chip: Ytrain e Xtrain

Ad ogni colpo di clock la memoria off-chip fornisce in uscita i 4 bit di dato (fase di lettura memoria principale) che andranno sia alle unità successive che alle due memorie on-chip. Al colpo di clock successivo, questi 4 bit saranno scritti all'interno delle due memorie Y_{train} e X_{train} (fase di scrittura) , oltre la lettura dei successivi 4 bit di dato della memoria off-chip. Questo ciclo si ripeterà fino a quando non si è letta tutta la memoria off-chip ed è avvenuta la corretta scrittura dei dati nelle memorie on-chip.

In termini di tempi di ciclo, bastano solamente 217 colpi di clock per scrivere entrambe le memorie.

4.4 Gestione occorrenze dei dati

La gestione sul numero di occorrenze dei dati, proveniente dalla memoria off-chip è stata analizzata sia mediante un'architettura seriale che con una di tipo parallela.

4.4.1 Implementazione seriale

A tal proposito, è stata implementata un'unità base chiamata *cnt_occorrenze* al fine di contare tutte le occorrenze di 0, 1, 2 dei vettori colonna Y e X1. In altre parole, quest'unità ha il compito di contare quanti parlamentari sono stati favorevoli o contrari ad una proposta di legge o si sono astenuti alla votazione riferiti al vettore X1. Riguardo Y, ha il compito di contare quanti parlamentari sono democratici o repubblicani. Il conteggio è stato gestito serialmente, cioè contando le occorrenze dei dati prima del vettore Y e poi successivamente del vettore X1.

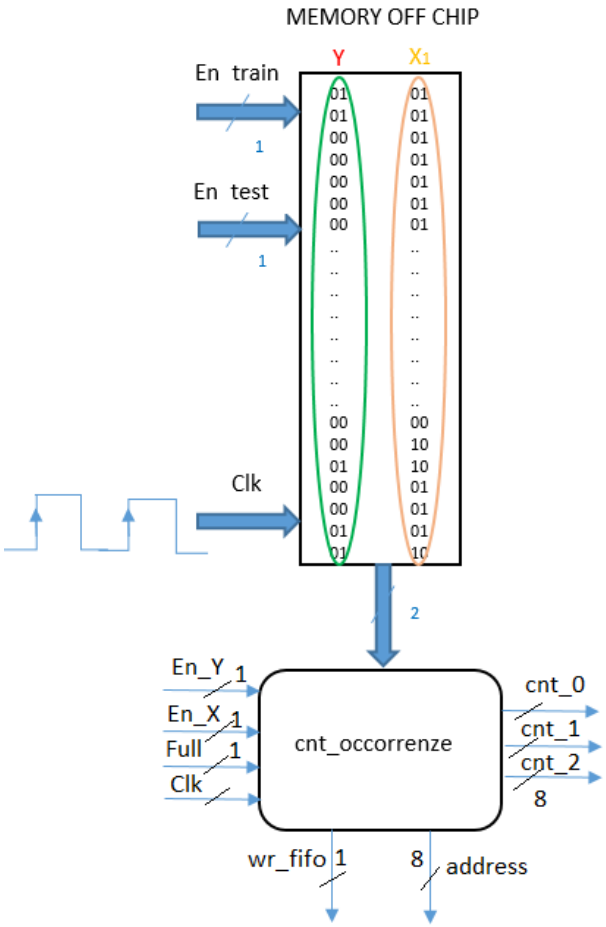

Figura 4.10. Gestione seriale dei dati

Tale unità presenta due enable, uno per abilitare la lettura del vettore Y e l'altro per abilitare X. Inoltre, un ingresso, *full*, segnala il completamento dei dati letti (*full* = 1 quando ha letto tutti i due vettori).

Ad ogni ciclo di clock, sono stati presi in considerazione 2 bit alla volta di tutto

il vettore colonna Y e successivamente di tutto il vettore colonna X. Dunque, il conteggio delle occorrenze è stato gestito serialmente prima per Y e poi per X1.
Il *cnt_occorrenze*, oltre a tenere traccie delle occorrenze, è stato progettato al fine di memorizzare anche le posizioni in cui si presentano i dati.
In figura è mostrato un esempio del flusso di conteggio considerando il vettore X costituito da 14 elementi:

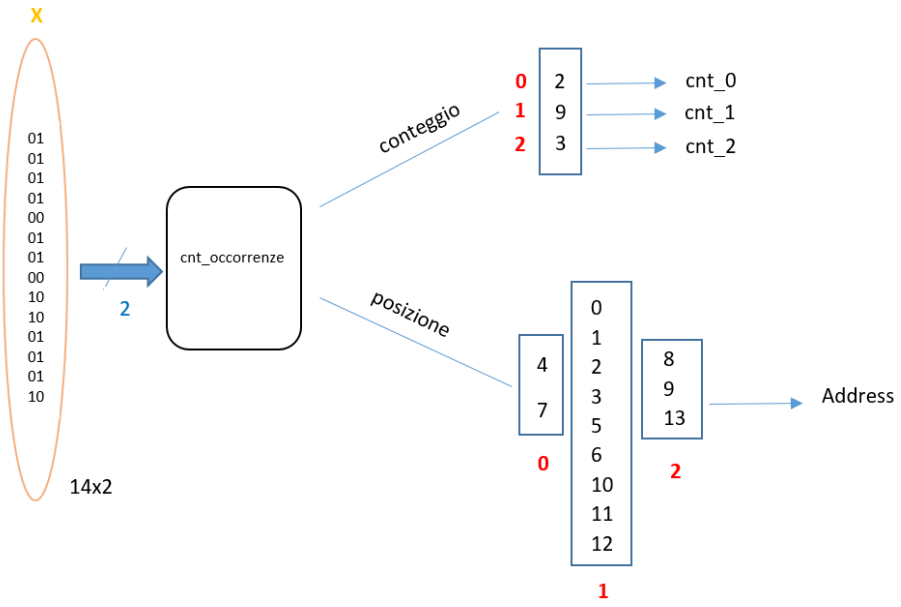

Figura 4.11. Esempio di elaborazione del contatore occorrenze

Nel caso reale di 217 dati, trattandosi di una gestione seriale di quest'ultimi, il numero di cicli di clock per terminare l'intero flusso di processing è stato:

$$n_{clock,serial} = n\_righe \cdot n\_vettori\_colonna = 217 \times 2 = 434 \tag{4.1}$$

Il processo di Y e X1 termina dopo 434 cicli di clock ottenendo in uscita i conteggi e le posizioni. Inoltre, considerando una frequenza di clock pari a 50 MHz, si è ottenuto un tempo di ciclo pari a:

$$T_{ciclo,serial} = n_{clock,serial} \cdot T_{clock} = n_{clock,serial} \cdot \frac{1}{f_{clock}} = 434 \cdot 20 \cdot 10^{-9} = 8.68\mu s \tag{4.2}$$

La scelta di considerare un valore piuttosto basso sulla frequeza (50MHz), deriva dal fatto di aver preso in considerazione il caso peggiore per il calcolo del tempo ciclo per poi salire in frequenza verificando le differenze.

L'idea successiva è stata quella di migliorare il tempo di ciclo per la sola elaborazione dei dati di ingresso, passando ad un'elaborazione parallela.

4.4.2 Implementazione parallela

Per ridurre il tempo di ciclo, si è passati ad un implementazione parallela raddoppiando il numero di risorse, in modo tale da gestire separatamente il numero di occorrenze del vettore Y e di X1. Con questa scelta, sono state definite 2 unità *cnt_occorrenze_Y* e *cnt_occorrenze_X1* in cui ciascuno elabora i dati per il rispettivo vettore (figura 4.12):

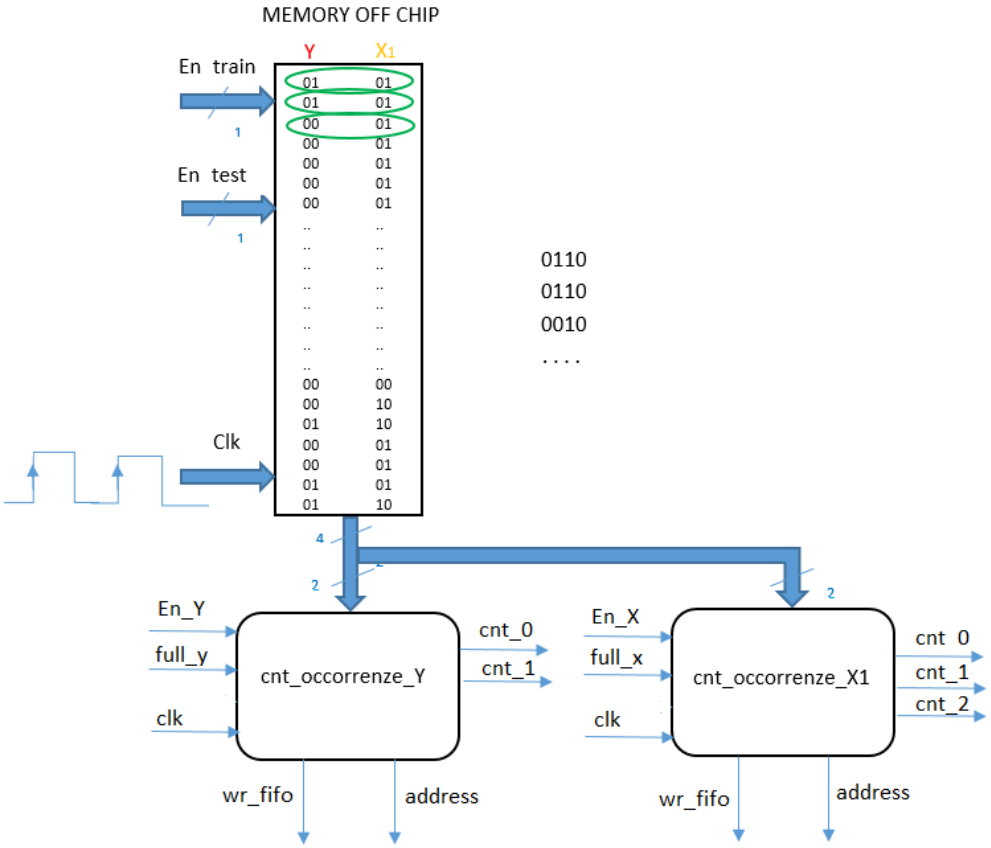

Figura 4.12. Gestione parallela dei dati

dove:

- **cnt_occorrenze_Y**, conta il numero di occorrenze di 0 e 1 del vettore colonna Y e determina anche le posizioni in cui si sono presentate;
- **cnt_occorrenze_X1**, conta il numero di occorrenze di 0, 1 e 2 del vettore colonna X e determina anche le posizioni in cui si sono presentate.

Quest'elaborazione è stata gestita completamente in parallelo.
La figura sottostante mostra un flusso di esempio per 2 vettori di input (Y e X1) ciascuno di 14 elementi:

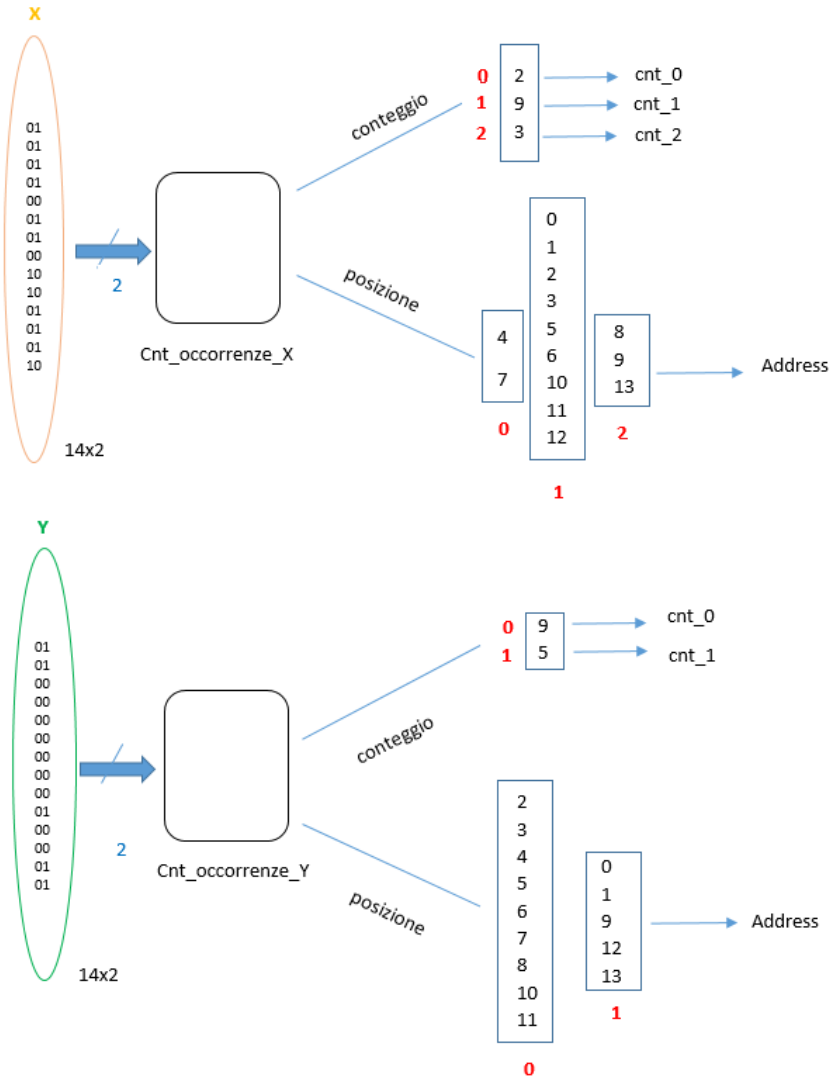

Figura 4.13. Esempio di elaborazione in parallelo dei contatori di occorrenze

Dunque, l'uscita della memoria off chip (4 bit) è stata splittata in due parti dove i primi 2 bit (rappresentanti la Y) sono stati forniti al *cnt_occorrenze_Y*, mentre i restanti 2 bit (rappresentanti la X1) sono stati forniti al *cnt_occorrenze_X1*. Così facendo, i dati sono processati in parallelo e la gestione è sempre sincrona al clock. Anche in questo caso, è stato calcolato il numero di colpi di clock necessari per

terminare l'intero flusso di processing considerando il caso completo (217 elementi):

$$n_{clock,parallel} = \frac{(n\_righe \cdot n\_vettori\_colonna)}{2} = 217 \tag{4.3}$$

Dopo 217 colpi di clock, termina il processo dei dati di Y e X1 e considerando una frequenza di clock pari a 50 MHz, si è ottenuto un tempo di ciclo pari a:

$$T_{ciclo,parallel} = n_{clock,parallel} \cdot T_{clock} = n_{clock,parallel} \cdot \frac{1}{f_{clock}} = 217 \cdot 20^{-9} = 4.34\mu s \tag{4.4}$$

Si è stato ottenuto miglioramento del di tempo ciclo passando da un'implementazione seriale ad una parallela con grado di parallelismo pari a 2.

4.5 FIFO

Nell'architettura hardware è risultato necessario l'utilizzo di alcune fifo per la gestione delle posizioni fornite dalle unità *cnt_occorrenze*. Per quest'ultime è stato scelto un parallelismo pari a 8 bit in quanto sono stati usati come indirizzi per accedere alle 2^8 locazioni delle due memorie on-chip (Y_{train} e X_{train}). Le fifo sono state dimensionate considerando come caso peggiore il caso in cui si presentassero 217 valori uguali con dimensione pari a 256×8. In questo caso, la fifo sarebbe riempita da 217 valori, da 0 a 216, corrispondenti alle posizioni in cui si presenta il dato. La figura successiva mostra il caso specifico:

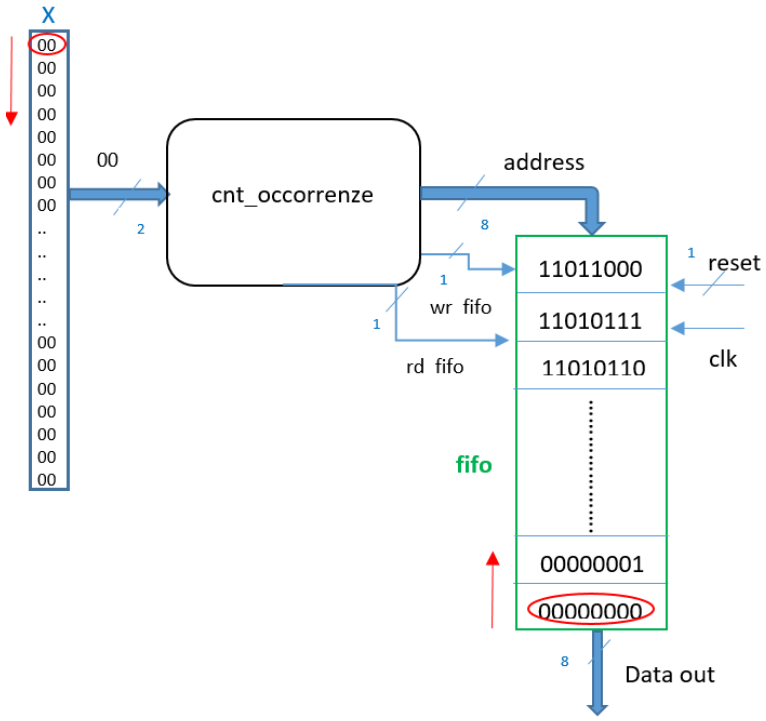

Figura 4.14. Dimensionamento massimo della fifo

Ovviamente, se i dati che entrano nella fifo sono in alternanza, quest'ultima si riempirà con meno posizioni. Infatti, la figura 4.15 mostra il caso in cui il dato da considerare è '00':

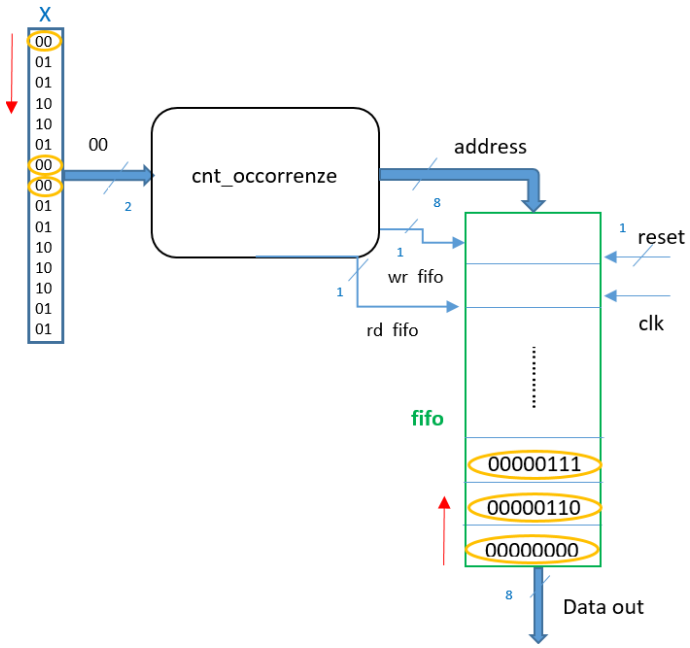

Figura 4.15. Riempimento posizioni per il valore '00'

4.5.1 Gestione seriale e parallela

Anche in questo caso, per la gestione delle posizioni del numero di occorrenze, si è cercato di implementare una soluzione seriale ed una parallela partendo con la serializzazione dell'elaborazione implementando una sola fifo per ogni *cnt_occorrenze*. La situazione descritta è quella mostrata in seguito:

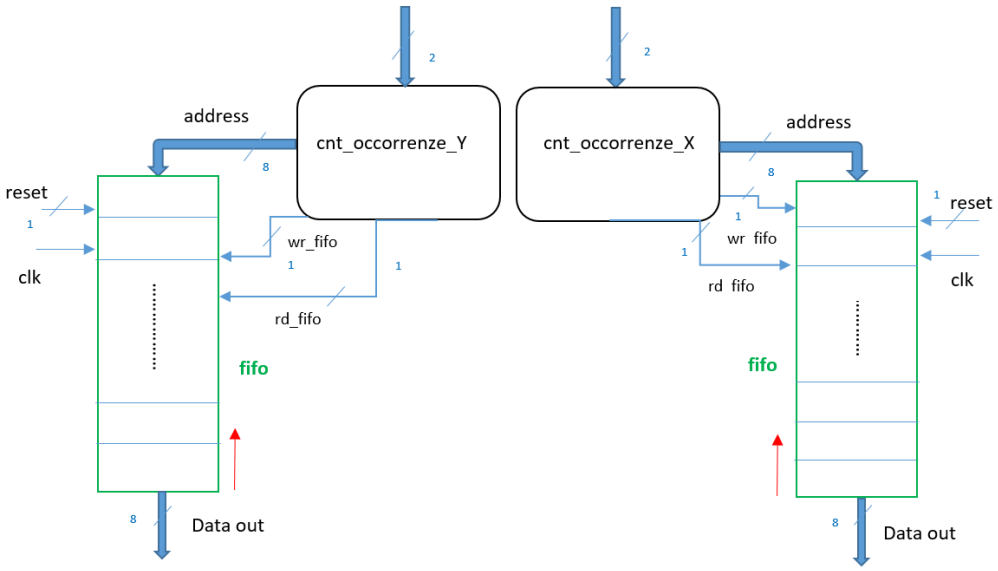

Figura 4.16. Gestione seriale fifo

Analizzando il lato sinistro della figura 4.16, si nota come il *cnt_occorrenze_Y* che fornisce le uscite ad un'unica fifo (FIFO 1). Quest'ultime corrispondono alle posizioni per le quali si è presentato uno 0 piuttosto che un 1 in uscita dalla memoria off-chip e viceversa.

In questo caso, le posizioni corrispondenti alle occorrenze 0 e 1 saranno tutte memorizzate in sequenza nella stessa fifo. Ciò renderebbe complessa la gestione hardware necessaria per discriminare queste posizioni. Occorre distinguere le posizioni relative all'occorrenza 0 e quelle relative all'occorrenza 1. Quest'informazione è estremamente importante perchè serve ad accedere nella memoria on-chip per leggere il suo contenuto in funzione della posizione corrente che sarà successivamente processato in un altro blocco.

L'utilizzo di un'unica risorsa ha causato una diminuzione del tempo di ciclo per il processing in quanto, al fine di distinguere a quali posizioni sono associate le occorrenze, è stato necessario memorizzare in primis le posizioni per gli 0. Quest'ultime sono state usate successivamente come indirizzi per accedere nella memoria on-chip.

Terminato tale processing, si è passato all'elaborazione per le occorrenze di 1 ripetendo l'accesso nella memoria on-chip. Questo procedimento è risultato necessario per determinare il valore ha assunto l'uscita X_{train} in funzione di Y=0 e X=1, con lo scopo di determinare successivamente P(Xin=i|Y=j).

È stata sviluppata una possibile gestione seriale considerando le prime metà locazioni della fifo (da 0 alla 127) associate alle posizioni in cui si presentano gli 0 e l'altra metà locazioni (dalla 128 alla 256) per le posizioni in cui si presentano gli 1. In figura è stato mostrato il caso specifico:

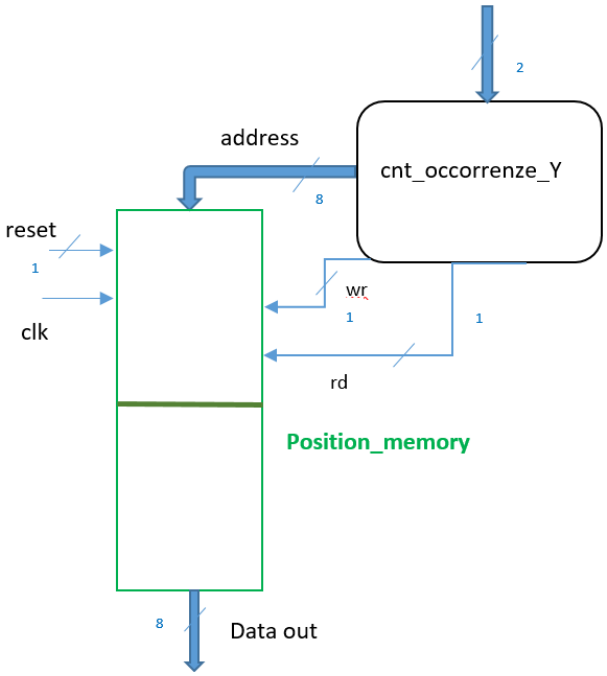

Figura 4.17. Divisione memoria per gestire la presenza di 2 occorrenze

Ovviamente, in questo caso, non si tratta più di una fifo ma di una memoria generica chiamata *position_memory*, in quanto non assume più il comportamento di *first input first output* ma memorizza le posizioni in locazioni differenti. In particolare, il margine di errore è molto elevato perchè se si presentassero in uscita dalla memoria off-chip 216 valori pari a 0, nella memoria si andrebbero a memorizzare 216 posizioni associate alla presenza dello 0. Ma con la suddivisione delle locazioni, metà per il valore 0 e metà per l'1, avremmo indovinato solamente le prime 108 posizioni mentre le restanti saranno errate perchè per la memoria corrisponderebbero alle posizioni dell'occorrenza 1. Ovviamente, questa gestione è stata pensata anche per la parte destra della figura 4.16 per il *cnt_occorrenze_X* la cui uscita è connessa

ad un'unica fifo. In questo caso, poichè le occorrenze provenienti dalla memoria off-chip sono 0,1 e 2, è stato pensato di suddividere le locazioni della fifo in tre parti ciascuno delle quali associato ad un valore specifico. La figura sottostante mostra la situazione descritta:

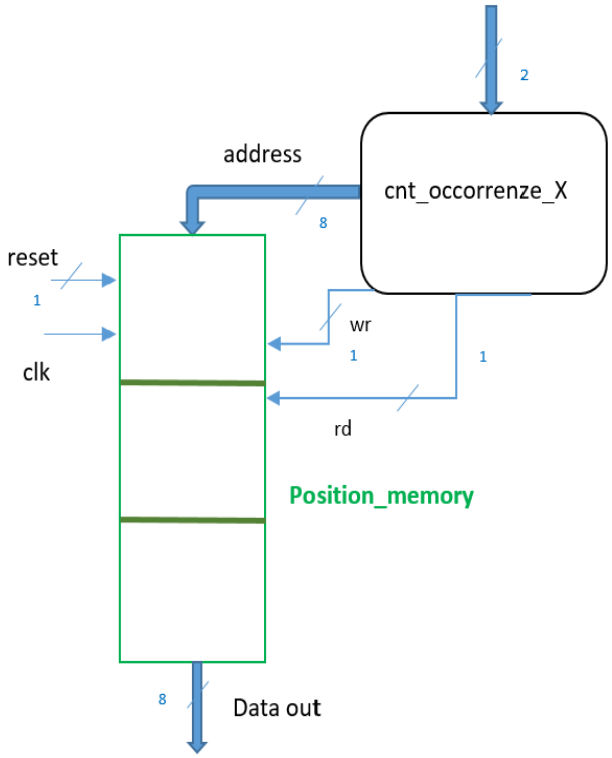

Figura 4.18. Divisione memoria per gestire la presenza di 3 occorrenze

Anche in questo caso, l'errore che si andrebbe a commettere non è trascurabile perchè a seconda del flusso dei dati iniziali, le posizioni associate cambierebbero ma per la fifo rimarrebbero fisse. Ragion per cui la gestione dinamica delle posizioni, utilizzando un'unica fifo, è stata scartata.

Si è optato per una gestione parallela aumentando il numero di risorse ma gestendo correttamente tutte le informazioni.

Nella gestione parallela, infatti, ogni fifo memorizza le posizioni associate ad un unico valore per cui sono state realizzate due fifo per l'uscita del *cnt_occorrenze_Y* in quanto i valori delle Y sono solo 0 e 1, e tre fifo per l'uscita del *cnt_occorrenze_X* in quanto i valori in questione sono 0, 1 e 2. La figura sottostante mostra la dinamica appena descritta:

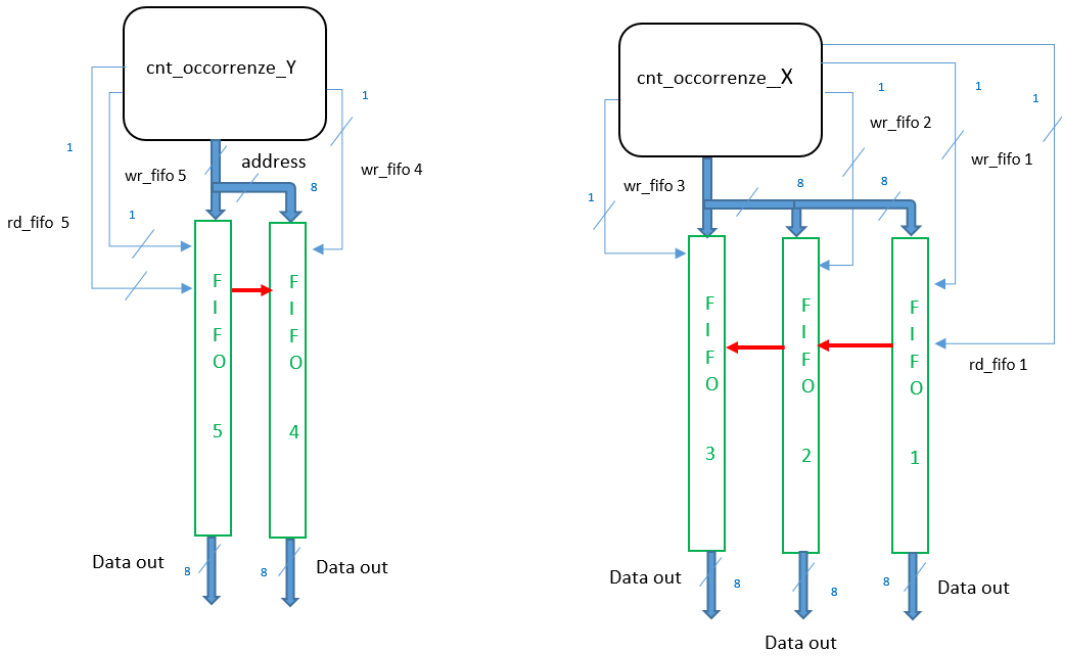

Figura 4.19. Gestione parallela delle posizioni

dove:

- **FIFO 1**, usata per memorizzare solo le posizioni in corrispondenza dell'occorrenza '00' (voto favorevole ad una legge);
- **FIFO 2**, usata per memorizzare solo le posizioni in corrispondenza dell'occorrenza '01' (voto contrario ad una legge);
- **FIFO 3**, usata per memorizzare solo le posizioni in corrispondenza dell'occorrenza '10' (astenuto al voto per una legge);
- **FIFO 4**, usata per memorizzare solo le posizioni in corrispondenza dell'occorrenza '01' (parlamentare votante democratico);
- **FIFO 5**, usata per memorizzare solo le posizioni in corrispondenza dell'occorrenza '00' (parlamentare votante repubblicano).

La parallelizzazione della gestione del flusso dei dati ha permesso di ridurre i tempi di ciclo in fase di scrittura delle fifo a differenza del caso seriale in cui con una fifo unica occorreva gestire tutte le differenti occorrenze.

La fase di scrittura è gestita in parallelo, mentre la lettura, poichè è stato necessario leggere le fifo in ordine, è stata eseguita in serie. Inoltre, il segnale indicato con una freccia rossa non è altro che un comando che permette di leggere la fifo successiva, una volta terminata quella attuale.

Ai fini del corretto calcolo della matrici di probabilità condizionata P(Y=m|Xin=i), è stato necessario leggere il valore di Y_{train} quando X=0 (FIFO 1), quando X=1 (FIFO 2) e quando X=2 (FIFO 3). Stessa cosa per il calcolo della matrice P(Xin=m|Y=i) in cui è stato letto il valore di X_{train} in funzione dei valori di Y noti.

4.6 Contatore Y e Contatore X

Come è stato detto precedentemente, è opportuno accedere alla memoria on-chip X_{train} e Y_{train} mediante gli indirizzi memorizzati nelle fifo.

Una volta effettuato l'accesso, i dati sono stati forniti a dei contatori chiamati *Contatore Y* e *Contatore X* aventi il compito di contare il numero di occorrenze. La figura 4.20 mostra il flusso appena descritto per il ramo di Y_{train} (il duale avviene anche per X_{train}):

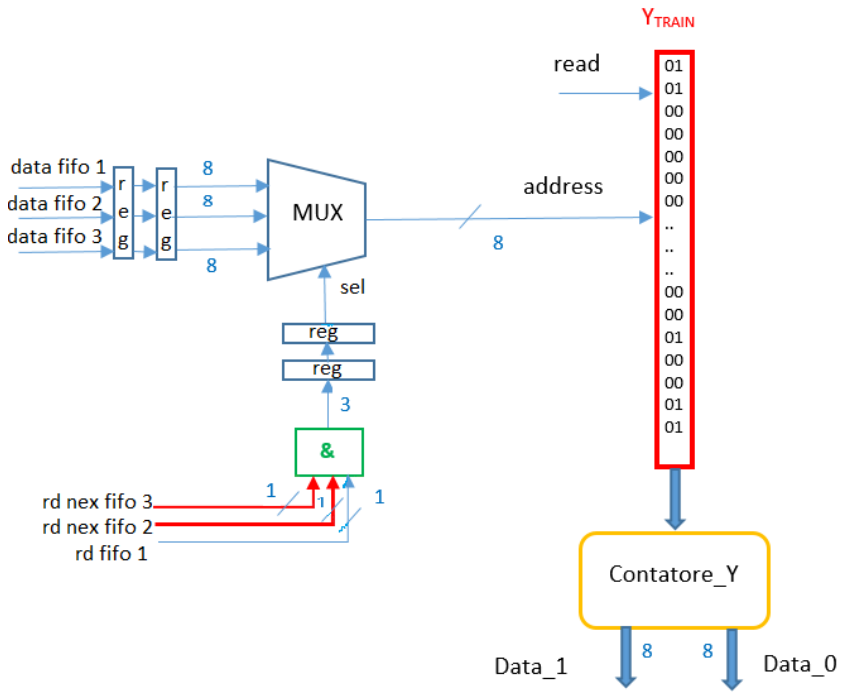

Figura 4.20. Accessi in memoria tramite indirizzamento posizionale

- **CONTATORE Y**, ha il compito di determinare quanti 00 e 01 sono presenti in Y_{train} per X = 00, X = 01 e X = 10. Detto in altri termini, calcola quanti democratici o repubblicani sono stati rispettivamente favorevoli, contrari o si sono astenuti ad una data legge. Sostanzialmente, questo procedimento rappresenta la fase del calcolo della probabilità P(Y=i|Xin=j);

- **CONTATORE X**, ha il compito di determinare quanti 00, 01 e 10 sono presenti in X_{train} per Y = 00 e Y = 01. Detto in altri termini, calcola quanti voti favorevoli, contrari e astenuti si sono ottenuti conoscendo a priori le ideologie dei parlamentari. Anche in questo caso, ci si sta proiettando alla determinazione della probabilità P(Xin=i|Y=j).

Nella figura precedente, è presente anche un multiplexer che ha il compito di selezionare le uscite provenienti dalle fifo che corrispondono alle posizioni utilizzate come indirizzi. Il selettore è stato gestito tramite un blocco concatenatore & che ha il compito di concatenare i bit di lettura delle fifo. In particolare:

- per la lettura della *fifo 1*, il segnale *rd_fifo_1* = 1 e in uscita dall'unità di concatenazione si ottiene "001" selezionando il primo ingresso corrispondente a *data_fifo_1*;

- una volta terminato di leggere la *fifo 1*, il segnale *rd_next_fifo_2* = 1 e si inizia a leggere il contenuto della *fifo 2*. In questo caso, in uscita del concatenatore si avrà "010" selezionando il secondo ingresso del *mux*, ovvero *data_fifo_2*;

- la stessa cosa avviene al termine della lettura della *fifo 2*, in cui il segnale *rd_fifo_3* andrà alto e in il segnale di selezione sarà "100" selezionando il terzo ingresso *data_fifo_2*.

Sono stati anche inseriti dei registri per di ritardare i segnali per due cicli di clock al fine di ottenere una corretta sincronizzazione.

4.7 Look Up Table

L'intera architettura è stata progettata con un parallelismo di $n_i = 8$ bit per i numeri interi uscenti dai vari contatori e fifo, mentre per la gestione dei numeri frazionari è stato scelto un parallelismo di $n_f = 16$ bit. La tecnica attuata per la gestione dei numeri frazionari è stata quella di utilizzare un fattore di scala e arrotondarli per eccesso al fine di processare solamente numeri interi.

Un blocco base di questa architettura hardware è la look up table (LUT) che è servita per effettuare una conversione totale dei dati processati fino ad ora in scala

logaritmica. Ciò dovuto al fatto che l'algoritmo DIN ha previsto molteplici divisioni e moltiplicazioni tra numeri frazionari andando a complicare l'implementazione hardware. Così facendo, è stato deciso di progettare una LUT contenente tutte le conversioni in scala logaritmica di 256 numeri interi (da 0 a 255) su cui è stato applicato il fattore di scala 2^7. Il risultato, a sua volta, è stato approssimato per eccesso e rappresentato in 16 bit. In seguito è mostrata l'espressione applicata per la conversione:

$$a = log(b)2^7 \tag{4.5}$$

dove:

- **a**, rappresenta il risultato in numero decimale della conversione;
- **b**, rappresenta il valore intero che giunge nella LUT su cui si vuole effettuare la conversione in scala logaritmica.

Il risultato, è stato poi convertito in binario con parallelismo 16 e memorizzato nella LUT.

Il vantaggio di aver utilizzato le LUT sta nell'aver salvato internamente tutte le conversioni logaritmiche e con un semplice accesso, ottenere il valore di uscita convertito a 16 bit. La conversione è instantanea e pronta per essere processata semplicemente, evitando operazioni molto onerose quali divisioni e moltiplicazioni. Di contro, si è incrementato di gran lunga l'area dell'intera architettura, poichè si tratta di unità piuttosto onerose dal punto di vista dell'area. Per questo motivo, si voleva implementare inizialmente un'unica LUT gestendo tutti i segnali che hanno bisogno di accedergli ma, essendo un numero molto elevato di segnali che indirizzano le locazioni della memoria, è risultato complicato gestirli e sincronizzarli tutti. Inoltre, utilizzando una sola unità è stato rilevante mettere delle memorie in uscita al fine di salvare il dato appena convertito e processarlo successivamente. Questo è dovuto al fatto che sono stati gestiti molteplici accessi in sequenza nelle locazioni della LUT, ragion per cui è necessario non perdere i dati appena forniti in uscita. Inoltre, per gestire l'indirizzamento della LUT mediante tutti questi segnali, è stata necessario implementare un multiplexer con tanti ingressi quanti sono i segnali. Il controllo di quest'ultimo diventa quanto più complesso quanto più risultano essere i suoi input, oltre al fatto che si va ad aumentare il parallelismo del selettore. Inoltre, non bisogna dimenticare le memorie in uscita da aggiungere per memorizzare i dati che andranno processati in momenti differenti.

Tutte queste osservazioni hanno spinto a genereare 8 LUT al fine di processare tutti dati in parallelo riducendo di gran lunga i tempi di ciclo (1/8 rispetto alla

singola LUT) a discapito di un aumento dell'area di 8 volte rispetta al caso iniziale. La figura sottostante mostra gli accorgimenti che sono stati fatti:

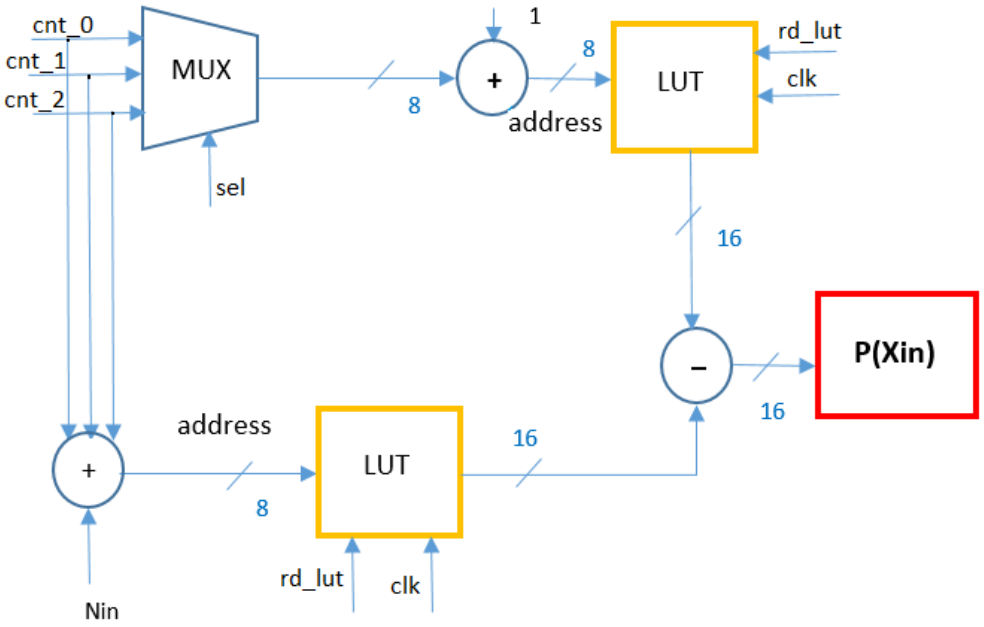

Figura 4.21. Gestione LUT per il calcolo di P(xin)

dove Nin rappresenta parametro che indica il caso ternario di riferimento in ingresso (Nin=3).

In figura si nota che il dato in uscita dalle LUT viene processato immediatamente riducendo la latenza riducendo anche la complessità delle operazioni più onerose quali divisioni e moltiplicazioni gestendole rispettivamente come somma e sottrazioni tra logaritmi.

In particolare, nella figura 4.21, si mostra lo step per il calcolo della probabilità Pxin data dalla differenza dei contenuti in uscita dalle due LUT.

Invece, la gestione della probabilità P(Y|Xin) è stata mostrata nella figura seguente:

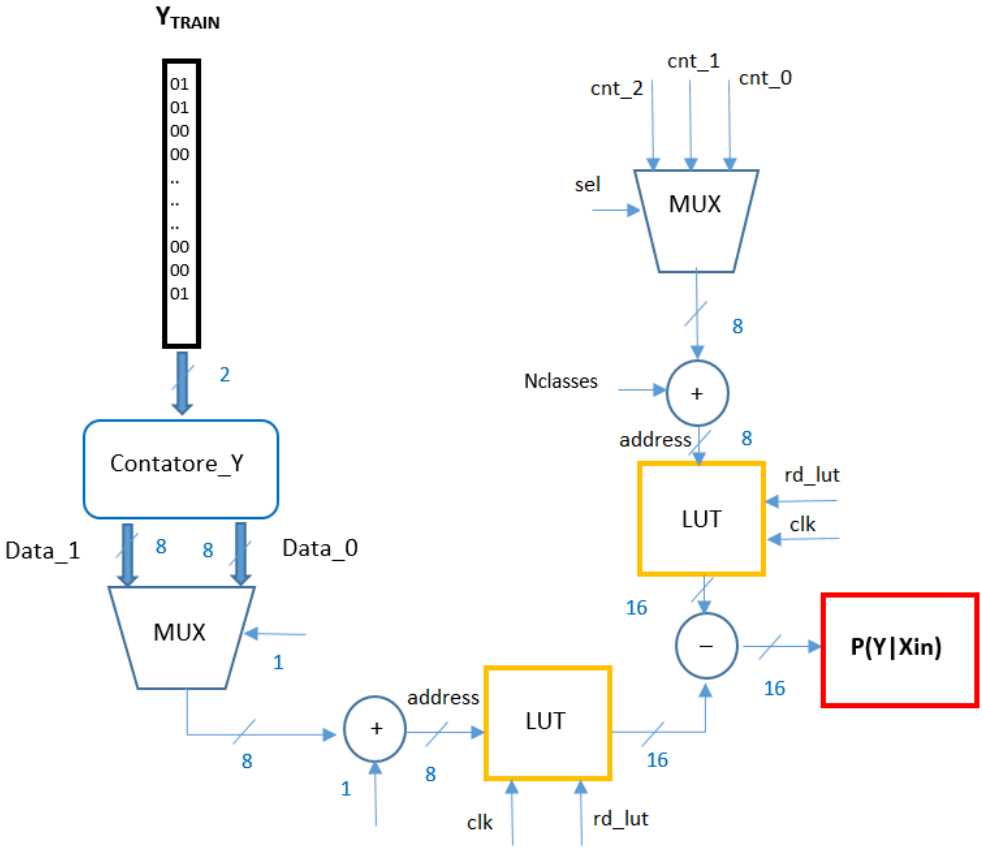

Figura 4.22. Gestione LUT per il calcolo di P(Y|Xin)

Anche in questo caso, Nclasses rappresenta la quantità che indica il caso binario di riferimento in uscita (Nclasses = 2).

Lo stesso procedimento è stato adottato per il calcolo degli altri parametri probabilistici Py e P(xin|y).

Capitolo 5

Unità di controllo

L'unità di controllo è un blocco di fondamentale importanza perchè ha permesso di fornire al datapath tutti i segnali di controllo dei singoli processing element al fine di eseguire le elaborazioni nell'istante esatto in cui serve farle.Tuttavia, è stato importante fornire all'unità di controllo anche dei segnali in ingresso che sono serviti per inizializzarla nel migliore dei modi.

5.1 Gestione dell'intera architettura

Prima di progettare l'unità di controllo, occorre definire i seguenti tre segnali esterni:

- **Clock**
- **Reset**
- **Start**

Il *clock* è il segnale di sincronismo per tutti i blocchi presenti nell'intera architettura. Il *reset* ha il compito di resettare l'intero sistema e portarlo nello stato iniziale detto IDLE. Il segnale di *start*, invece, è molto importante perchè ha permesso di settare l'istante iniziale del processing.

Questi tre segnali vengono forniti sia in ingresso alla Control Unit sia alla memoria off chip e sia al datapath. La figura mostra l'alto livello di organizzazione strutturale:

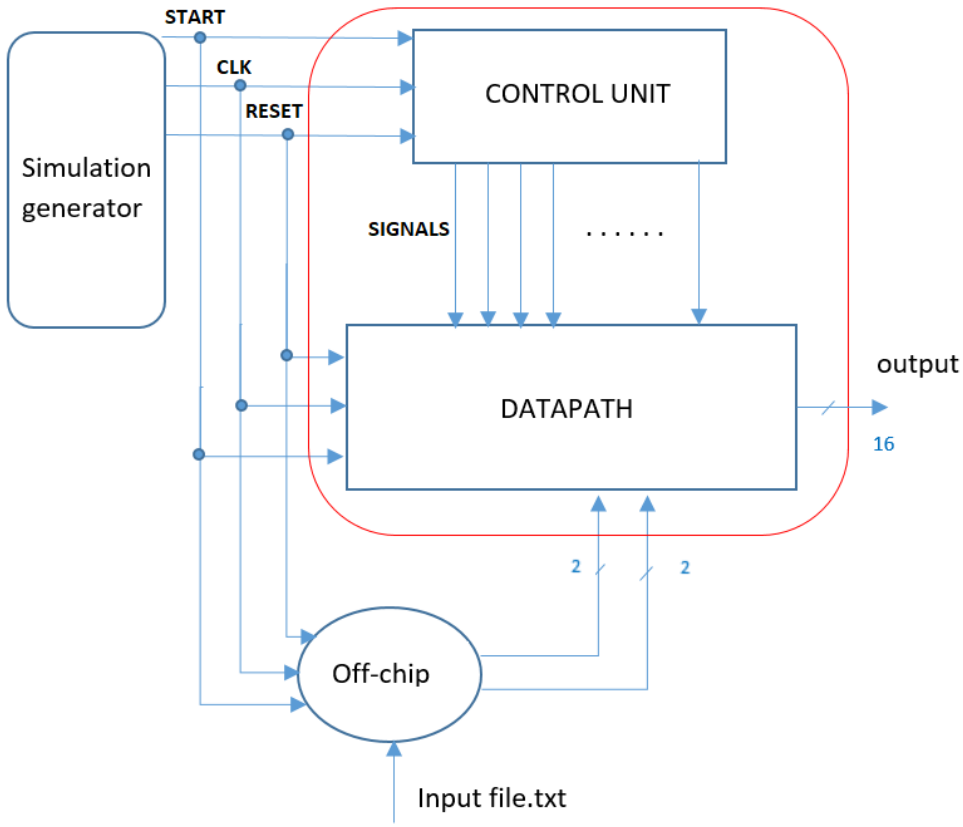

Figura 5.1. Gestione dell'intera architettura

Nello stato iniziale definito come *IDLE* in cui *RESET=1 e START=0*, il sistema non esegue nessun operazione e rimane in stanby in attesa di una commutazione di entrambi i segnali. Tuttavia, il RESET = 1 ha il compito di azzerare tutti i registri, memorie e contatori presenti nella struttura.

Con la commutazione di entrambi i segnali, *START=1 e RESET = 0*, la control unit inizia a fornire i segnali di controllo al datapath il quale inizia con il processing dei dati che giungono dalla memoria off chip (letti inizialmente da un file esterno). Il processo è sincrono al segnale di clock per cui le rispettive uscite saranno tutte sincronizzate. Dopo 450 cicli di clock, il datapath fornisce in uscita tutte le matrici di probabilità condizionata Pxin, Py, P(xin|y) e P(y|xin).

5.2 Macchina a stati

L'immagine sottostante indica l'intera struttura della macchina a stati:

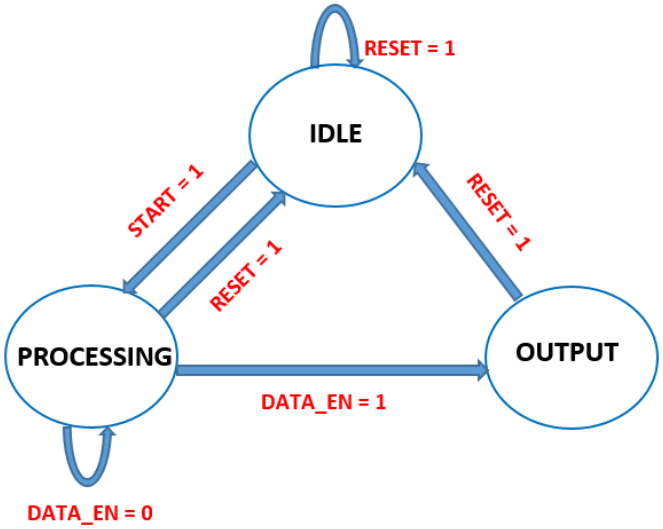

Figura 5.2. Macchina a stati

Nella figura precedente, è stato possibile definire tre stati principali in cui è può trovarsi l'intero sistema:

- **IDLE**
- **PROCESSING**
- **OUTPUT**

Lo stato di IDLE è lo stato di partenza della macchina in cui non viene compiuta alcuna operazione se non quella di inizializzazione azzerando tutti i registri, memorie e contatori. Dunque si tratta di una fase di stanby in cui non vengono elaborati i dati, ma si attende le commutazioni dei segnali per partire.

La macchina passa dallo stato di *IDLE* allo stato di *PROCESSING* quando START = 1. In questo caso, l'intera architettura parte a processare i dati provenienti dalla memoria off chip in cui la control unit fornisce gli opportuni segnali di controllo al datapath per una correta elaborazione. Dallo stato di *PROCESSING*, si può passare nuovamente allo stato di *IDLE*, se RESET = 0, oppure passare all'ultimo (con DATA_EN = 1) stato che prende il nome di *OUTPUT*.

In questo ultmo stato, la macchina ha terminato di fare tutte le opportune elaborazioni dei dati ed è pronta a fornire le uscite corrette. Da questo stato è possibile tornare nuovamente in *IDLE* con RESET = 1.

Capitolo 6

Simulazioni e validazione dell'hardware

La simulazione dell'intera architettura hardware viene eseguita mediante il simulatore Modelsim il quale ha permesso di verificare il corretto funzionamento del nodo e il corretto sincronismo del flusso dei dati.

6.1 Simulazioni Modelsim

L'obiettivo di sviluppo di questo lavoro di tesi si è incentrato prettamente sul 40% dell'elaborazione iniziale dell'intero nodo informativo, lasciando il restante 60% ad un eventuale lavoro futuro. In particolare, lo sviluppo in questione, ha permesso di determinare le seguenti matrici di probabilità le quali sono state ottenute mediante simulazione in *Modelsim*:

- **Pxin**, rappresenta la probabilità di ottenere un determinato ingresso Xin. Nel nostro caso ternario, Xin può rappresentare il voto favorevole ad una legge (Xin = 00), contrario (Xin = 01) o astenuto alla votazione. Dunque, tale paramentro tiene conto della proababilità che si verifichi un'evento Xin rispetto ad un'altro.

- **Py**, rappresenta la probabilità di ottenere una determinata uscita Y. Nel nostro caso, Y rappresenta l'ideologia del parlamentare che vota, dunque Y = 0 corrisponderà al caso democratico mentre Y = 1 al caso repubblicano.

- **P(Y=m|Xin=i)**, rappresenta la probabilità di ottenere una determinata uscita Y noto il valore di ingresso Xin. In altri termini, indica la probabilità che un parlamentare sia repubblicano o democratico conoscendo il risultato delle votazioni iniziali.

- **P(Xin=m|Y=i)**, rappresenta la probabilità di ottenere un determinato ingresso Xin noto a priori il valore di uscita Y. In altri termini, indica la probabilità che di ottenere un voto favorevole, contrario o astenuto conoscendo a priori l'ideologia dei parlamentari.

6.1.1 Pxin

La seguente sezione descrive in dettaglio la simulazione relativa alla determinazione della matrice di probabilità Pxin.

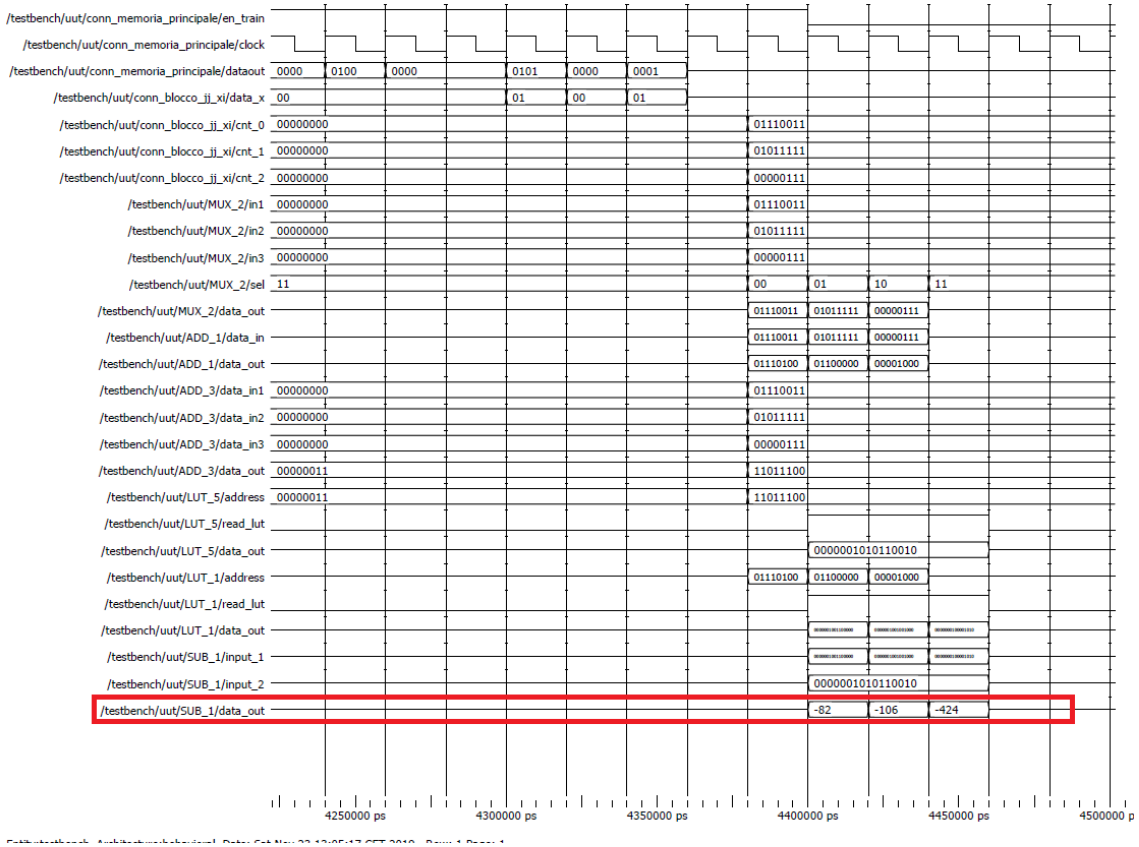

Figura 6.1. Simulazione Pxin

Si nota che il tratto evidenziato indica i valori delle probabilità al variare dei tre casi (0,1 e 2) ottenuti in uscita dal sottrattore il quale sostituisce l'operazione di divisione. Questi risultati sono stati disponibili esattamente dopo 4400000 ps corrispondenti a 220 cicli di clock con frequenza 50 MHz (T_{clk}=20 ns). Poichè l'implementazione è stata fatta nel dominio logaritmico, occorre ritornare nel dominio

reale mediante la seguente espressione:

$$y = e^{\frac{x}{2^7}} \tag{6.1}$$

dove x rappresenta il valore evidenziato nella simulazione.

Così facendo, si sono ottenuti i reali valori delle probabilità Pxin:

Xin	**Pxin**
0	0,5267
1	0,4349
2	0,0364

Per questo dataset, la probabilità di ottenere un evento 0 è leggermente superiore rispetto all'evento 1, mentre risulta essere improbabile l'occorrenza di un 2.

6.1.2 Py

La seguente sezione descrive in dettaglio la simulazione relativa alla determinazione della matrice di probabilità Py.

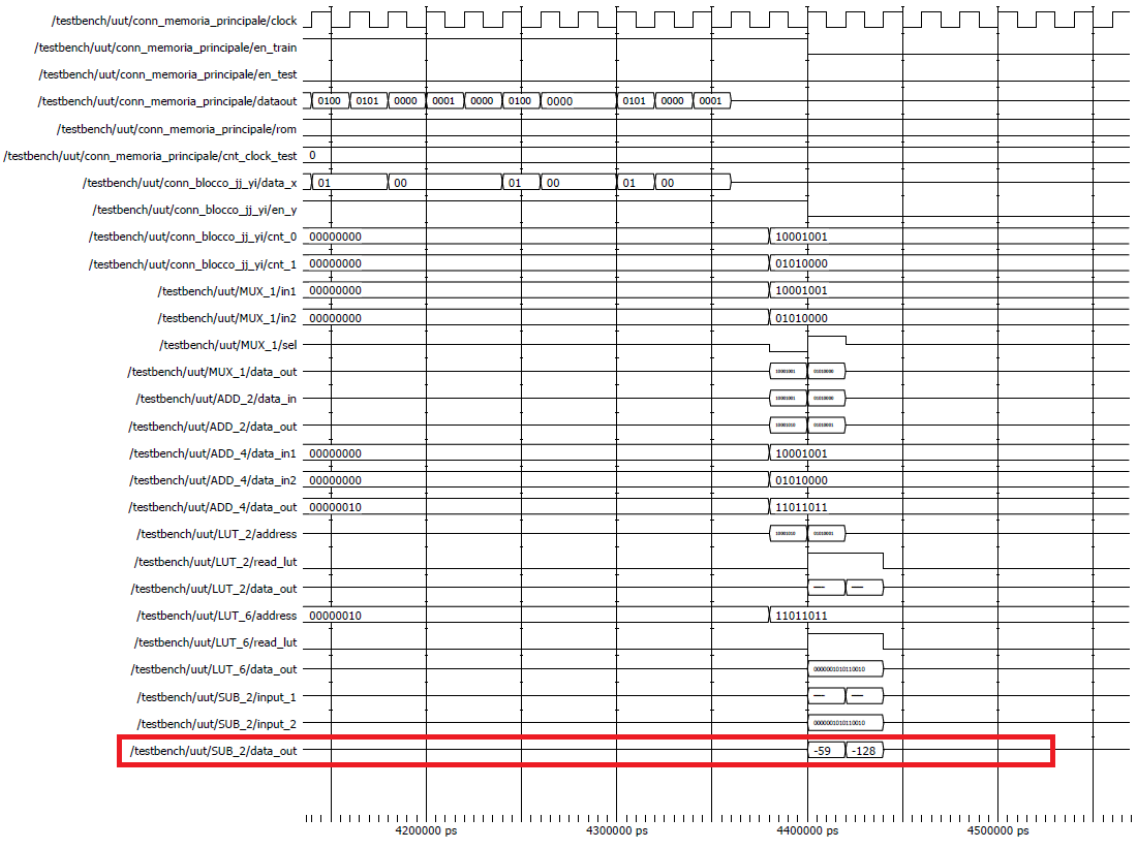

Figura 6.2. Simulazione Py

Anche in questo caso, l'uscita dal sottrattore corrispondente ai valori di Py è stata disponibile dopo 440000 ps (220 colpi di cicli con frequenza 50MHz). Applicando l'espressione 6.1, si è ottenuto:

Y	**PY**
0	0,6313
1	0,3679

Dunque, la probabilità di ottenere uno 0 in uscita è maggiore rispetto ad ottenere l'evento opposto.

6.1.3 Py|xin

La seguente sezione, è stato descritto in dettaglio la simulazione relativa alla determinazione della matrice di probabilità PY|Xin.

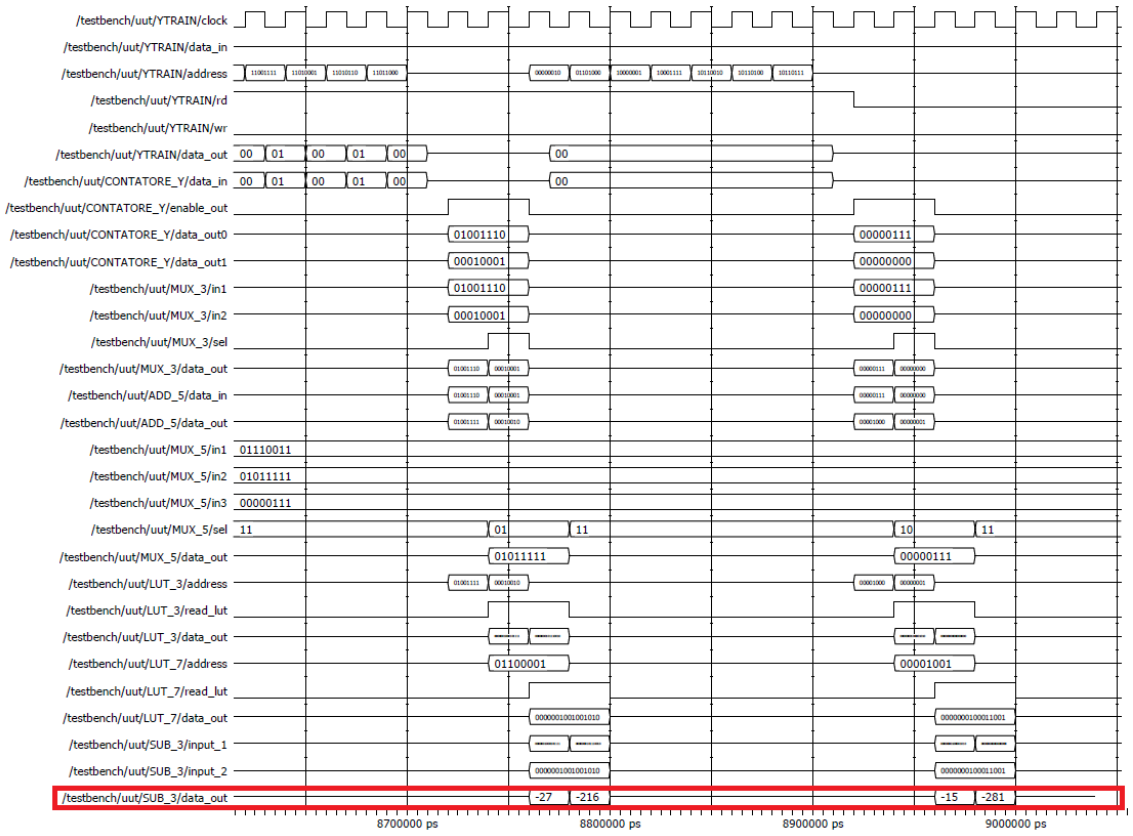

Figura 6.4. Simulazione Py|xin

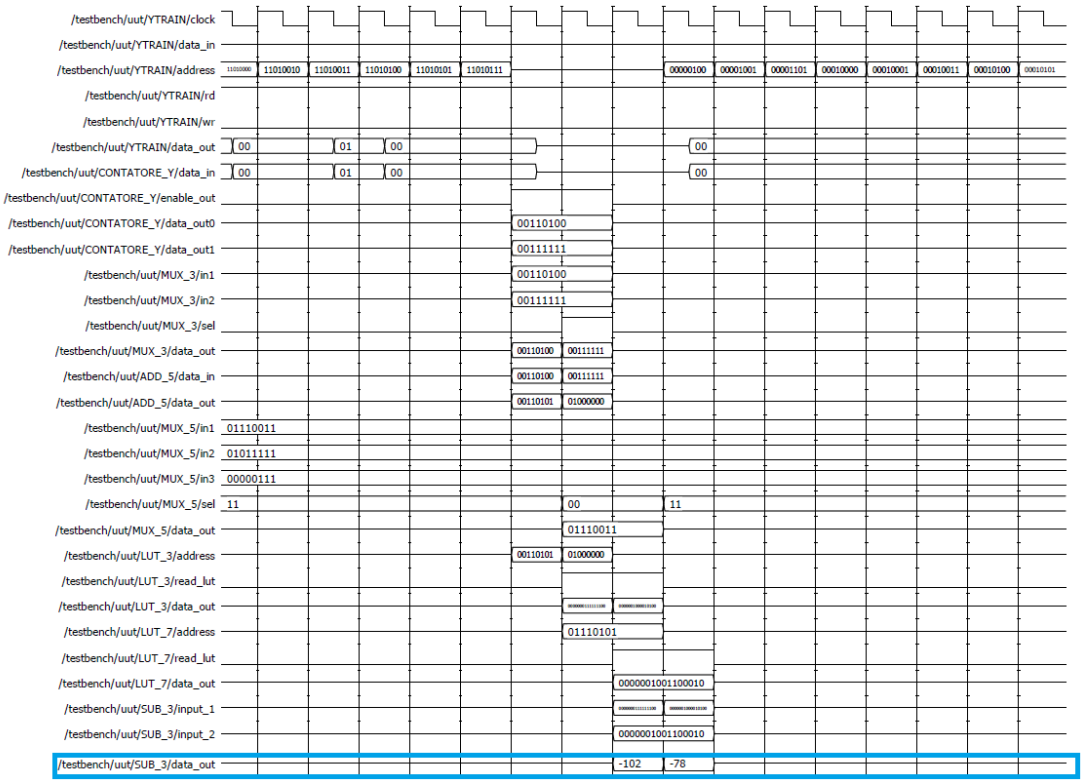

Figura 6.3. Simulazione Py|xin

Nelle figure 6.3 e 6.4 si evidenziano gli istanti differenti in cui si è ottenuta l'intera matrice di probabilità condizionata Py|xin. I primi dati sono stati disponibili dopo 6780000 ps (339 cicli di clock) mentre gli ultimi dopo altri 111 colpi di clock per un totale di 450, necessari per determinare tutti gli elementi della matrice di probabilità. Anche in questo caso, si è passati nel dominio reale ottenendo:

P(Y|Xin)	0	1	2
0	0,4507	0,8092	0,8891
1	0,5436	0,1846	0,1127

6.1.4 Pxin|y

La seguente sezione, infine, descrive in dettaglio la simulazione relativa alla determinazione della matrice di probabilità PXin|Y.

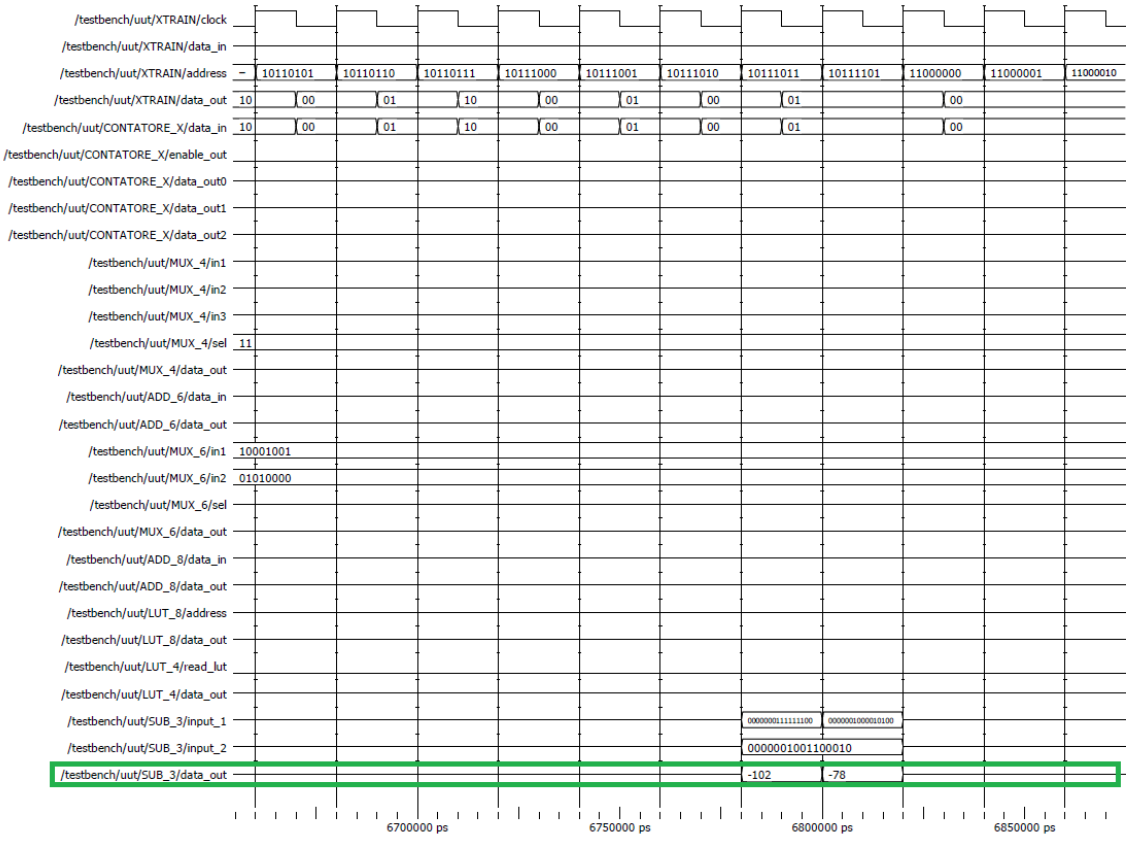

Figura 6.5. Simulazione Pxin|y

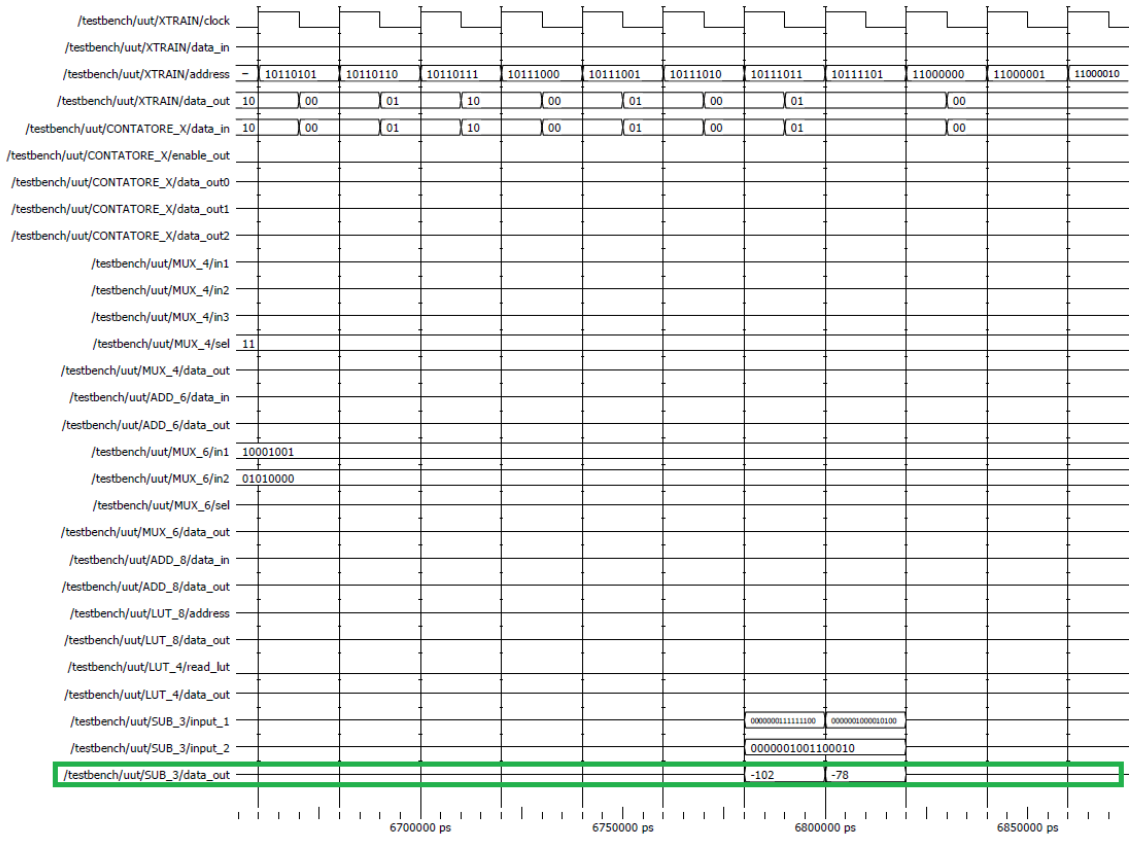

Figura 6.6. Simulazione Pxin|y

Anche in questo caso, sono stati necessari diversi istanti temporali al fine di determinare l'intera matrice di probabilità condizionata, utilizzando 450 colpi di clock per determinare tutti gli elementi di tale matrice.
Passando nel dominio reale si è ottenuto:

P(Xin|Y)	0	1
0	0,3766	0,7667
1	0,5609	0,216
2	0,0568	0,012

6.2 Analisi dei risultati

Le matrici di probabilità ottenute precedentemente sono state opportunamente confrontate con quelle del modello esatto di riferimento realizzato mediante Matlab.

Le figure successive riportano i risultati ottenuti con il modello esatto:

Xin	**Pxin (esatto)**
0	0,52727
1	0,43636
2	0,03636

Y	**PY (esatto)**
0	0,63013
1	0,36986

P(Y|Xin) (esatto)	0	1	2
0	0,45299	0,81443	0,8888
1	0,54700	0,18556	0,1111

P(Xin|Y) (esatto)	0	1
0	0,37857	0,77108
1	0,56428	0,2168
2	0,05714	0,01204

Gli errori ottenuti rispetto ai valori in uscita dall'architettura hardware sono piuttosto trascurabili e al di sotto dello 0,50%, come è mostrato nella figura sottostante:

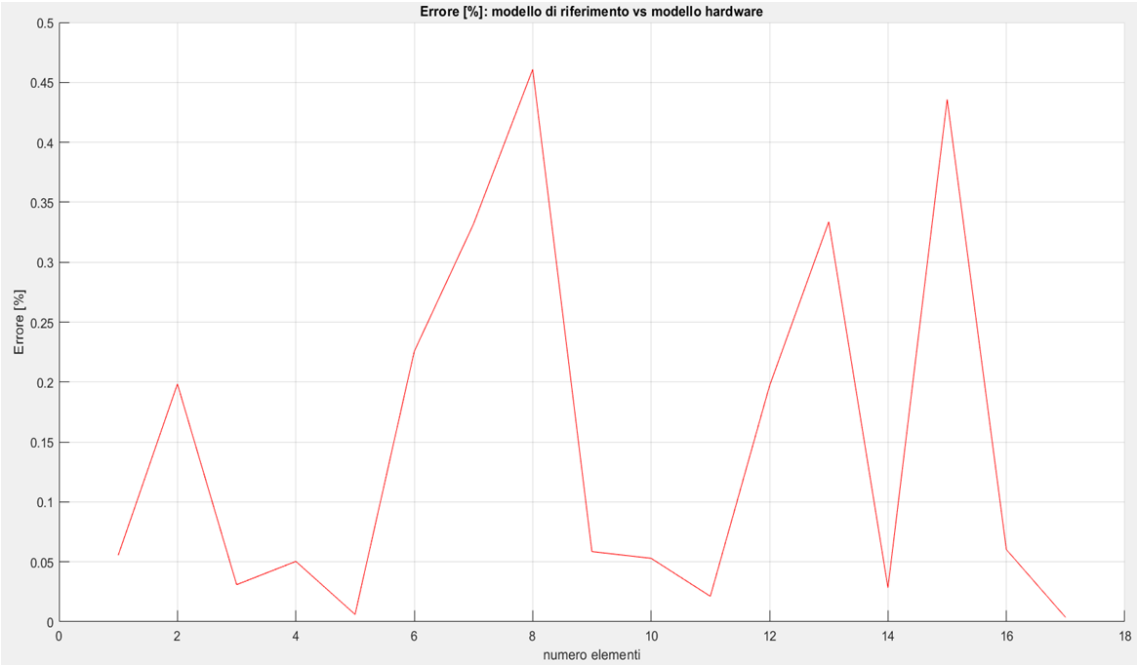

Figura 6.7. Errore % tra il modello di riferimento e il modello hardware

La figura successiva è relativa ad un grafico di errore per tutte le matrici di probabilità:

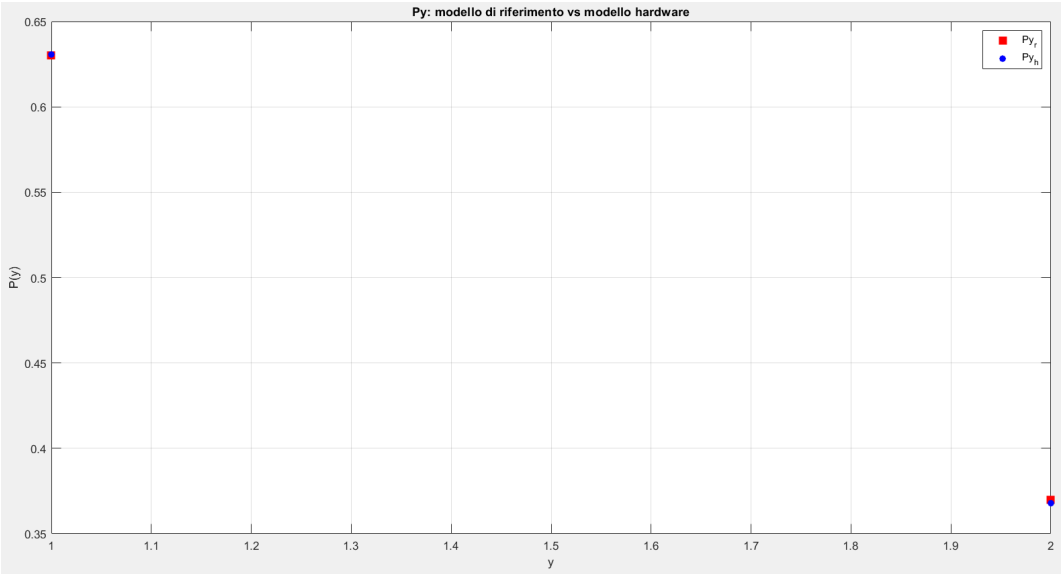

Figura 6.8. Confronto hardware e software per Py

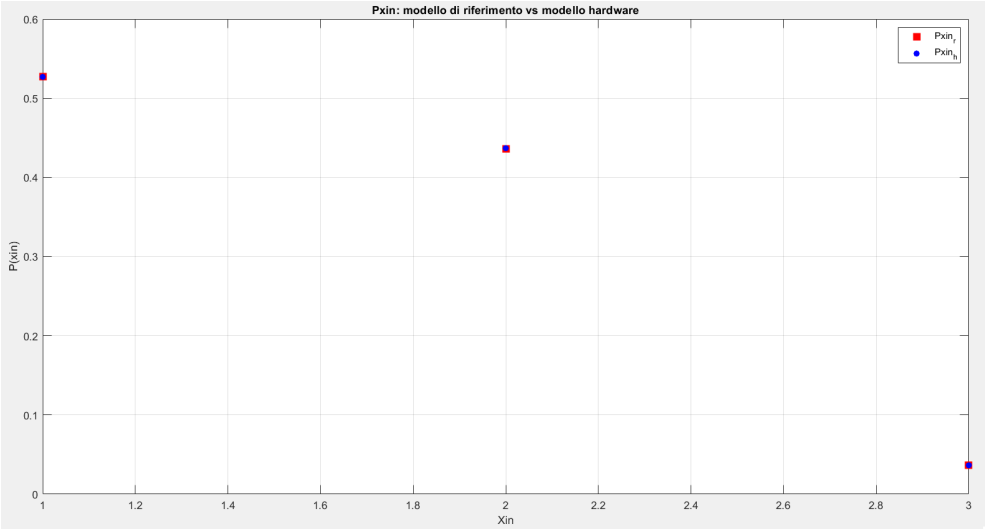

Figura 6.9. Confronto hardware e software per Pxin

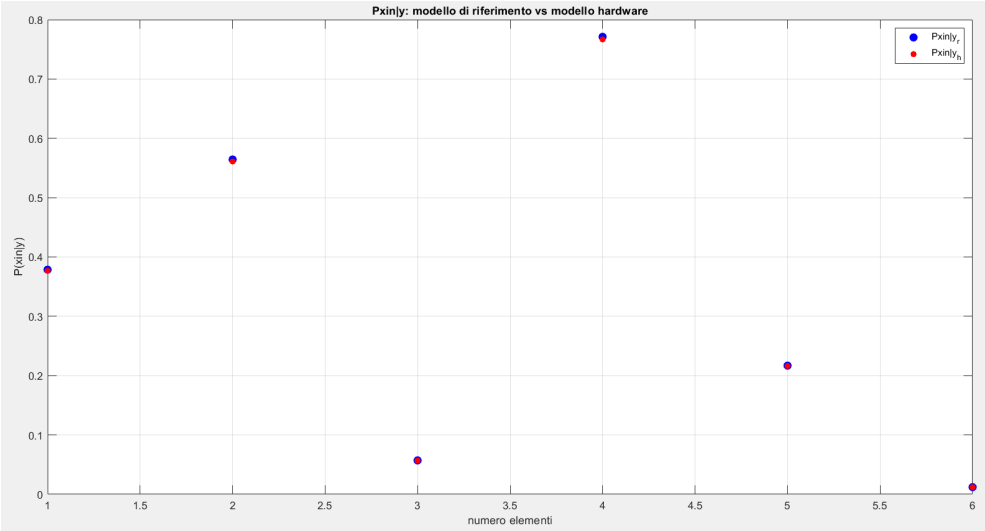

Figura 6.10. Confronto hardware e software per Pxin|y

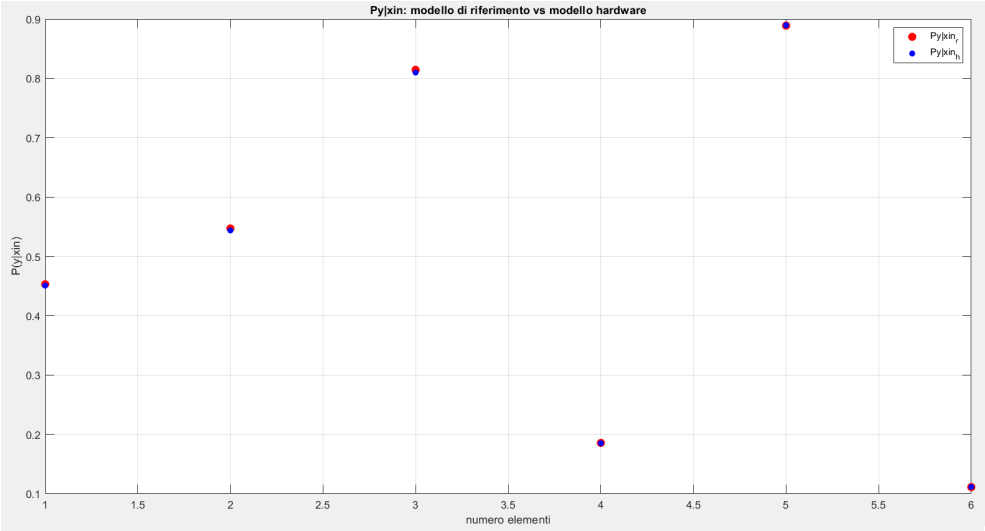

Figura 6.11. Confronto hardware e software per Py|xin

Le immagini precedenti presentano la quasi perfetta sovrapposizione dei punti rappresentati, segno di un errore alquanto trascurabile nel confronto tra i due modelli.

6.3 Confronto tempi di ciclo e speed up

Come accennato precedentemente, il tempo di ciclo e lo speed up sono parametri fondamentali ed utili per fare delle scelte corrette durante la progettazione. In particolare, in questo lavoro di tesi si è cercato di migliorare molto le prestazioni sacrificando dall'altra parte l'area occupata a causa della gestione parallela.

Un confronto in termini di prestazioni, tra il modello hardware e quello software, è stato eseguito considerando ovviamente il 40% del processing totale. Per tale quantità, si sono ottenuti i seguenti risultati:

- **MODELLO SOFTWARE:**

	fclk [GHz]	Tclk [ns]	n° cicli di clock	**Tcliclo** [μs]
esecuzione n° 1	4	0,25	6988	1747
esecuzione n° 2	4	0,25	11596	2899
esecuzione n° 3	4	0,25	20936	5234
esecuzione n° 4	4	0,25	9940	2485
esecuzione n° 5	4	0,25	32516	8129

L'algoritmo è stato eseguito cinque volte per analizzare quale fosse il caso migliore in termini di tempo ciclo. Il primo tentativo corrisponde al caso ottimale con un Tciclo = 1747 μs per il modello software tenuto conto della frequenza di un processore standard pari a 4 Ghz.

- **MODELLO HARDWARE:**

fclk [MHz]	Tclk [ns]	n° cicli di clock	**Tcliclo** [μs]
50	20	450	9

In questo caso, l'architettura hardware è stata eseguita una sola volta poichè per tentativi successivi i risultati non sarebbero cambiati. Questo perchè, a differenza del software che gira su un PC, nel caso della simulazione con Modelsim il tempo di ciclo è sempre lo stesso.

Tuttavia, l'aspetto di fondamentale importanza è che:

$$T_{ciclo,hardware} < T_{ciclo,software} \tag{6.2}$$

Questo nonostante la frequenza del processore di un pc sia notevolmente più alta.

Inoltre, questi tempi di ciclo sono stati calcolati per il 40% del processing di un nodo informativo. Per cui, in maniera approssimativa è stato possibile dedurre che un intero processing element impiega circa 4376 μs per il modello software e

circa 23 μs per il caso hardware. Poichè l'intera rete neurale, per il caso studiato in questo lavoro di tesi, ha bisogno di 31 nodi per ottenere la stima finale, si è ottenuto approssimativamente:

$$T_{ciclo,software} = 31 \cdot 4367 = 135,377ms \tag{6.3}$$

$$T_{ciclo,hardware} = 31 \cdot 23 = 713\mu s \tag{6.4}$$

Questi tempi rappresentano una stima per l'intera rete neurale. Preme sottolineare, tuttavia, la differenza tra i due modelli in termini di prestazioni in cui il modello hardware risulta essere più veloce rispetto a quello software.

Capitolo 7

Conclusioni e sviluppi futuri

Per concludere, le analisi svolte per questo lavoro di tesi hanno portato sviluppi particolarmente interessanti. In primis, partendo da un modello di riferimento prettamente software, è stata implementata un'architettura parallela che fosse in grado di avere lo stesso comportamento del modello reale. Tale scelta si è basata soprattutto dalle migliori prestazioni in termini di latenza e speed up oltre che per la semplicità di gestione. Di contro, queste scelte organizzative hanno portato ad un aumento drastico dell'area occupata.

I risultati, inoltre, dimostrano come l'implementazione hardware sia più veloce in termini di speed up rispetto al modello software e che gli errori probabilistici sono stati oltretutto trascurabili seguendo in modo efficiente la logica del modello esatto.

Sicuramente, un aspetto da migliorare nell'implementazione hardware è la riduzione dell'area attuando uno sharing delle risorse cercando di non perdere in termini di prestazioni. Un lavoro da fare in futuro sarà quello di completare l'intero nodo informativo per provare a realizzare tutta la rete neurale ottenendo le stime opportune confrontate con il modello esatto. Inoltre, sarà opportuno sintetizzare il singolo nodo per determinare i consumi in termini di potenza. Lo sharing delle risorse sarà un altro punto molto importante per un lavoro futuro al fine di ridurre l'area occupata.

In questa tesi, si è proposto dunque un'alternativa hardware dell'algoritmo in termini di complessità e prestazioni.

Bibliografia

[1] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, Joel S. Emer, "Efficient Processing of Deep Neutral Networks: A tutorial and Survey", December 2017.
[2] [Online]. Available: https://it.wikipedia.org/wiki/Neurone
[3] [Online]. Available: https://biologiawiki.it/wiki/neurone/
[4] Monica Visintin, Giulio Franzese, "Deep Informations Network", Dipartimento di Elettronica e Telecomunicazioni Politecnico di Torino, Italia, 6 marzo 2018.
[5] N. Tishby, N. Zaslavsky, 'Deep Learning and the Information Bottleneck Principle" March 2015.