UNIVERSITÀ DEGLI STUDI DI ROMA
TOR VERGATA

FACOLTÀ DI INGEGNERIA
Corso di Laurea Magistrale in Ingegneria Informatica

TESI DI LAUREA IN
Ingegneria Informatica

MIGLIORAMENTO DELL'ACCURATEZZA E DELL'EFFICIENZA ENERGETICA DELLE RETI NEURALI AD IMPULSO TRAMITE TEMPISTICHE ACCURATE: UN CASO DI STUDIO

Relatore:
Prof. **A. Pellegrini**

Laureando:
Adrian Petru Baba
Matr. **0320578**

Correlatori:
A. Pimpini, A. Piccione

ANNO ACCADEMICO 2022/2023

Abstract

Le reti neurali ad impulsi sono una classe specifica di reti neurali che stanno guadagnando sempre più popolarità grazie alla loro elevata accuratezza ed efficienza energetica rispetto alle reti neurali artificiali standard. In questo lavoro si vuole studiare quanto, per queste reti, le due metriche menzionate in precedenza possano beneficiare dall'impiego di tecniche di simulazione temporali accurate, sfruttando il paradigma della Discrete Event Simulation (DES). Analizzando l'approccio tradizionale, ovvero la simulazione timestepped, e le sue limitazioni in termini di approssimazione temporale e costo computazionale, si evidenzia come queste influenzano negativamente sia l'accuratezza che l'efficienza energetica delle reti. Attraverso l'utilizzo di DES, e in particolare del simulatore ROOT-Sim, si propone una metodologia di simulazione che promette di superare tali limitazioni, attraverso una gestione più precisa e flessibile del tempo di simulazione e degli eventi neurali. Per dimostrare l'applicabilità e i vantaggi dell'approcio precedentemente proposto viene esplorato un caso di studio specifico, ovvero la classificazione delle lettere in linguaggio braille. I risultati sperimentali mostrano miglioramenti significativi le metriche considerate, confermando le ipotesi iniziali e offrendo spunti promettenti per ricerche future in quest'ambito.

Indice

Abstract **i**

1 Introduzione **1**

2 Contesto e Stato dell'Arte **3**

2.1 Spiking Neural Networks (SNNs) 3

2.1.1 Codifica dell'input 5

2.1.2 Decodifica dell'output 6

2.1.3 Dinamica dei neuroni 6

2.2 Simulazione time stepped 7

2.2.1 Modelli di neuroni 9

2.3 Training delle reti neurali timestepped 11

2.3.1 Reti neurali senza dinamiche temporali 12

2.3.2 Applicazione alle SNN 13

2.3.3 Problematiche dell'SNN 16

2.3.4 Plausibilità a livello biologico 18

3 Simulazione di SNN Precisa **19**

3.1 DES 19

3.2 PDES 21

3.2.1 Parallelizzazione 22

3.2.2 Sincronizzazione 23

3.2.3 Sincronizzazione ottimistica (Time Warp) 24

3.2.4 Rollback 25

3.3 The ROme OpTimistic Simulator (ROOT-Sim) 27

3.4 Training con questo tipo di simulazione 29
3.4.1 Spike-Timing Dependent Plasticity (STDP) 30

4 Caso di Studio **33**
4.1 Letter Recognition 33
4.2 Replicazione del modello 37
4.3 Il modello con ROOT-Sim 44
4.3.1 NeuronInit 45
4.3.2 NeuronHandleSpike e SynapseHandleSpike 47
4.3.3 NeuronWake 48
4.3.4 NeuronEndAlign 48
4.4 Training rete precisa 48
4.4.1 Implementazione 49
4.4.2 Utilizzo 50

5 Risultati Sperimentali **52**
5.1 Accuratezza della rete 52
5.1.1 Recurrent SNN (RSNN) 52
5.1.2 Feed-Forward SNN (FFSNN) 65
5.2 Training 68
5.3 Efficienza Energetica 77

6 Conclusioni e Lavori Futuri **80**

Elenco delle figure

2.1 Una semplice rappresentazione del modello 5
2.2 Confronto tra i 3 tipi di neuroni 11
2.3 rappresentazione di una rete srotolata. I parametri sono condivisi. x[t] rappresenta l'input al momento t 14
2.4 rappresentazione degli stati di una rete SNN srotolata 15
2.5 Confronto fra varie funzioni per il gradiente 17

3.1 Problema sincronizzazione 23
3.2 Variazione peso sinaptico in base alla correlazione dei tempi di spike 30

4.1 Accuracy simulazione time stepped 37
4.2 Corrente e potenziale dei primi 150 time steps 40
4.3 Risultati della replicazione degli esperimenti del paper 42

5.1 I risultati sono stati ottenuti dall'esecuzione della rete, su input di threshold $\theta = 2$, sul primo elemento del validation set e con il miglior set di pesi ottenuti dalla replicazione dello script del paper di riferimento 54
5.2 Simulazioni senza delay artificiale e periodo refrattario variabile 56
5.3 Simulazioni senza periodo refrattario e con delay artificiale variabile 57
5.4 Simulazioni con valori di periodo refrattario e delay artificiale misti 58
5.5 Simulazioni coi migliori parametri su tutti i set di pesi 59

5.6 Differenza cumulativa in accuracy, raggruppando per encoding threshold, sulle 5 diverse simulazioni 60
5.7 Simulazioni con pesi ricorrenti scalati di vari fattori e periodo refrattario variabile 62
5.8 Simulazioni con i migliori parametri 63
5.9 Differenza cumulativa in accuracy, raggruppando per encoding threshold, sulle 5 diverse simulazioni con i pesi ricorrenti scalati 64
5.10 Simulazione FFSNN con diversi periodi refrattari 65
5.11 Differenza cumulativa in accuracy delle esecuzioni con differenti delay sinaptici, per i diversi periodi refrattari 66
5.12 Risultati migliori FFSNN su ROOT-Sim 67
5.13 Consumi energetici stimati dalla simulazione del test set su un eventuale hardware per le due reti 79

List of listings

4.1 Definizione del tempo massimo . . . 38
4.2 Utilizzo dei potenziali . . . 38
4.3 Seed per la generazione dei valori . . . 41
4.4 Codice splitting RSNN . . . 43
4.5 Codice splitting FSNN . . . 44
4.6 struttura neuron_params_t e parametri di configurazione . . . 44
4.7 struct per lo stato dei neuroni . . . 46
4.8 parametri per l'applicazione di stdp . . . 49
4.9 array dinamico per memorizzare gli spike . . . 50
5.1 aggiunta dei parametri su finestra temporale e di scala . . . 69
5.2 parametri per gestire l'aggiunta o rimozione artificiale di spike in output . . . 72
5.3 parametri per l'apprendimento senza utilizzare i tempi di spike . 74

1. Introduzione

Le reti neurali ad impulsi (Spiking Neural Networks) hanno ottenuto un'interesse sempre crescente negli ultimi anni: date le loro caratteristiche che le accomunano, più di altri tipi di reti neurali, al funzionamento delle reti biologiche, permettono di ottenere soluzioni più accurate e con un notevole risparmio energetico. Questo avviene grazie al fatto che incapsulano anche il concetto di tempo, e quindi per il loro funzionamento non sono importanti solo i valori in input ed i pesi delle sinapsi, ma anche quando qualcosa avviene nella rete. Questo meccanismo si può sintetizzare in questo modo: l'input della rete è un treno di spike, ovvero una serie di inpulsi che avvengono in momenti specifici. Questi si propagano e hanno degli effetti sui neuroni adiacenti, in particolare sulla loro corrente e di conseguenza sul loro potenziale. Quando il potenziale di un certo neurone raggiunge un certo valore, questo genererà un'impulso (spike) a sua volta, così fino al termine della simulazione. Questo significa che l'input va converito in spike, e gli spike in output vanno interpretati per ottenere una predizione. Queste caratteristiche hanno però un prezzo: la rete non si può più valutare analiticamente, ma richiede un processo di simulazione. In letteratura viene utilizzato in maniera preponderante il paradigma timestepped: L'intero arco di durata della simulazione viene diviso in intervalli di dimensione fissa, e si simula ciò che avviene ad ogni intervallo. Questa soluzione è semplice a livello di algoritmo, relativamente efficiente e permette l'applicazione della Backpropagation Through Time, ma ovviamente ha il suo aspetto negativo: l'approssimazione di tutte le dinamiche neurali che sono ancorate ai timestep. Questo implica che la rete così simulata potrebbe discostarsi anche in maniera significativa dal comportamento reale di un modello biologico in funzione del

carico di lavoro, il che verosimilmente potrebbe portare a un calo dell'accuratezza. In questa tesi, concentreremo le nostre attenzioni su un paradigma di simulazione differente, detto Discrete Event Simulation (DES). Analizzeremo il suo funzionamento, e come questo è stato implementato nel simulatore reale utilizzato per gli scopi di questa tesi, ovvero ROOT-Sim. Successivamente verrà presentato il caso di studio preso in esame: un problema di classificazione delle lettere in linguaggio braille. Dopo aver descritto il problema—facendo riferimento ad un paper che verrà mostrato successivamente—verranno replicati gli esperimenti già presenti per ottenere le informazioni necessarie per l'implementazione della rete con il simulatore appena menzionato e dell'infrastruttura per il training. In seguito verranno presentati i risultati sperimentali ottenuti nelle simulazioni delle differenti tipologie di reti presentate nel paper e i dati riguardanti le accuratezze ottenute. A questo seguirà un'analisi del problema del training, che presenta notevoli differenze e problematiche da affrontare nel cambio di paradigma di simulazione, ed in ultima istanza verrà analizzato il problema dell'efficienza energetica, andando a fare una stima utilizzando dei dati e un modello matematico già presenti in letteratura, per effettuare un confronto fra i consumi previsti di un'eventuale futura implementazione hardware, che al momento del lavoro su questa tesi non esiste, delle migliori reti neurali dei due diversi paradigmi di simulazione.

2. Contesto e Stato dell'Arte

In questo capitolo verranno esposti i dettagli sulle SNN, sullo stato dell'arte riguardo le metodologie di simulazione correntemente utilizzate in letteratura (ovvero la simulazione timestepped) e su come avviene il training di una rete in questo contesto di simulazione.

2.1 Spiking Neural Networks (SNNs)

Una SNN è una tipologia particolare di rete neurale che cerca di simulare in maniera più accurata rispetto ad altre (come il *multilayer perceptron*) il comportamento delle reti neurali biologiche [23]. Per farlo, lo stato dei neuroni e delle sinapsi, nella loro evoluzione, incapsulano il concetto di tempo. Questa caratteristica porta il modello SNN ad essere potenzialmente molto più efficiente di altre reti neurali, rendendo in grado un singolo neurone di avere la stessa potenza espressiva di centinaia di neuroni, di un differente modello [24].

Lo stato della rete è definito dal **potenziale di membrana** di ciascun neurone e dal **peso delle sinapsi**. La comunicazione tra i neuroni avviene attraverso eventi di impulsi (**spike**): una volta che il potenziale di membrana di un neurone supera un suo valore ben definito di soglia (**threshold**), che nei modelli biologicamenti accurati si aggira intorno ai -50mV [5], uno spike verrà propagato a tutti i neuroni a cui quest'ultimo è connesso tramite una sinapsi. L'effetto dello spike dipenderà quindi dal momento nel tempo in cui questo è avvenuto e, dato che la propagazione avviene attraverso la sinapsi, dal peso associato a quest'ultima. In Figura 2.1 si può vedere una rappresentazione grafica del comportamento appena menzionato. Questo peso potrebbe variare nel

tempo in base all'attività sinaptica, per esempio per effetto degli stessi meccanismi di plasticità sinaptica discussi nei capitoli successivi; nelle reti simulate si tende tuttavia a mantenerlo costante. La propagazione dello spike ha ovviamente una durata: nei modelli neurali biologici potrebbe essere variabile [22], mentre in quelli simulati anche questo parametro solitamente viene mantenuto costante. Una volta che un neurone ha generato uno spike, questo entra in uno stato di riposo, abbassando il suo potenziale ad una certa soglia di reset, che nei modelli biologicamente accurati si aggira intorno ai -55mV [5], e per un certo periodo di tempo, definito **periodo refrattario**, questo neurone sarà incapace di generare ulteriori spike anche in presenza di nuovi stimoli perché il suo potenziale rimarrà ancorato al valore di reset. Questo comportamento è fondamentale per prevenire comportamenti di eccitazione incontrollata, avere un controllo sulla frequenza di spike che altrimenti potrebbe essere illimitata ed evitare situazioni di loop-feedback incontrollati. L'effetto di uno spike è quello di far variare la corrente elettrica attraverso il neurone destinatario, ed è questa corrente a definire di fatto la variazione di potenziale. Uno dei più grandi punti di debolezza della SNN è dato da quello che è anche il suo punto di forza: la dipendenza dal tempo. La rete così definita, a livello matematico, è intrattabile analiticamente: l'unico modo possibile per analizzare il suo comportamento è tramile la simulazione. A livello di topologia, in letteratura le SNN sono esattamente come le reti di altre tipologie: strati di neuroni (**layer**) interconnessi tra di loro. Si parla di rete ricorrente quando i neuroni di un certo layer presentano sinapsi con neuroni dello stesso layer.

L'utilizzo di una rete di questa tipologia implica che, per simularla, l'input, indipendentmente dalla sua natura, sia codificato come un **treno di impulsi** (**spike train**), ovvero una sequenza di spike posizionati in determinati istanti temporali precisi, in ingresso a neuroni di input. Questo significa che è richiesta un'operazione di codifica dell'input. Al termine della simulazione, l'output sarà anch'esso un treno di impulsi: per valutare il risultato è necessario effettuare un'operazione speculare di decodifica per valutare il risultato. Di seguito verranno esposte alcune possibili strategie di codifica e di decodifica [4, 16].

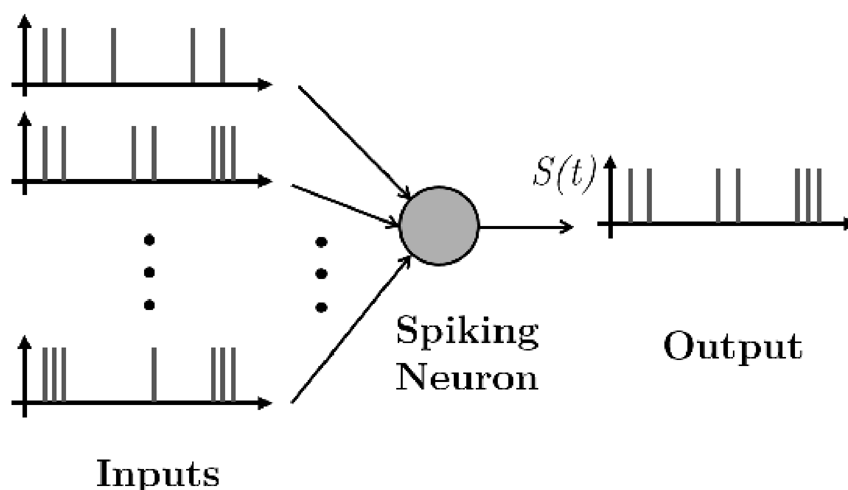


Figura 2.1: Una semplice rappresentazione del modello

2.1.1 Codifica dell'input

Prendiamo in esame un esempio semplice come quello della codifica delle immagini di un video. Per codificare le informazioni che contiene, possiamo utilizzare l'intensità dei singoli pixel in diversi modi, per esempio:

- *Time To First Spike Coding (TTFSC)*: In un contesto dove la rapidità degli spike è rilevante, la rete può essere realizzata affinché i neuroni in input effettuino uno spike iniziale con una velocità direttamente proporzionale all'intensità del pixel.
- *Rate Coding (RC)*: I pixel vengono visti come frequenze di spike, interpretate come parametri di una distribuzione di probabilità (poissoniana). Gli spike in input verranno generati in funzione di questa distribuzione.
- *Time Burst Coding (TBC)*: L'informazione viene codificata in un burst di spike, inviati in maniera contemporanea. L'informazione, dunque, viene contenuta nel numero di spike avvenuti, il cui numero può essere direttamente proporzionale all'intensità del pixel.
- *Temporal Contrast Coding (TCC)*: In questo approccio, si stabilisce un threshold per l'intensità del pixel. Uno spike varrà emesso quando l'intensità dello spike supera questo valore.

Esistono anche approcci che non convertono l'input in spike ma, per esempio, in una sorgente di corrente in ingresso ai neuroni dell'input. Tuttavia i primi approcci, detti *spike-based*, sono preferibili, siccome sfruttano al meglio le modalità con cui una SNN trae informazioni [12].

2.1.2 Decodifica dell'output

Solitamente, il layer di output presenterà un neurone per ciascuna classe da predire. Per decodificare lo spike train risultante di questi neuroni bisogna utilizzare strategie come:

- *Temporal Coding (TC)*: con questa decodifica, la predizione della rete sarà basata sul neurone che avrà generato uno spike per primo.
- *Rate Coding (RC)*: con questa decodifica, invece, la predizione si baserà sulla frequenza di spike di ciascun neurone, e quindi sul numero totale di spike di ciascun neurone.

Ovviamente l'interesse potrebbe essere quello di interpretare l'output non come un'assegnazione secca di una classe di appartenenza ma come la probabilità di appartenere all'una o all'altra classe.

Esistono modi per ottenere ciò a partire dalle strategie sopracitate; i dettagli esulano dagli scopi di questa tesi. Per una panoramica più approfondita su queste tecniche e sul loro confronto, si faccia riferimento a [12].

2.1.3 Dinamica dei neuroni

Per le reti SNN esistono molti modelli di neuroni possibili. Questi vengono solitamente modellati con equazioni differenziali più o meno complesse che descrivono il valore del potenziale e della corrente all'istante di tempo t.

Partendo dalle basi, la dinamica biologica di un neurone di tipo spiking viene descritto dalle seguenti equazioni differenziali [39]:

$$\frac{dV_i^l(t)}{dt} = \frac{-V_i^l(t) + V_r}{\tau_{mem}} + (I_i^l(t) + I_{i,ext}^l) \cdot R \tag{2.1}$$

$$\frac{dI_i^l(t)}{dt} = \frac{-I_i^l(t)}{\tau_{\text{syn}}} + \sum_j (W_{ji}^{fw} \cdot S_j^{l-1}(t) + W_{ji}^{rec} \cdot S_j^l(t)) \tag{2.2}$$

I vari elementi rappresentano:

- V_r: potenziale di reset
- $I_{i,ext}^l$: eventuale fonte di corrente esterna al neurone i del layer l
- R: resistenza della membrana del neurone
- τ_{mem} e τ_{sym}: costante di tempo della membrana e costante di tempo sinaptica
- S_j^l: spike train del neurone j appartenente al layer l. Questo è rappresentato come $\Theta(V^l(t) - V_{th})$, dove Θ è la funzione heaviside e V_{th} è il potenziale di threshold, cosicchè $S_j^l(t)$ valga 0 se il potenziale del neurone j al tempo t non ha superato il potenziale di threshold, 1 altrimenti
- W_{ji}^{fw}(rispettivamente W_{ji}^{rec}): peso assegnato alla sinapsi che collega il neurone j al neurone corrente i per le sinapsi feed-forward (rispettivamente sinapsi ricorrenti)

Per una panoramica più approfondita sul significato di queste variabili si faccia riferimento a [12].

2.2 Simulazione time stepped

Le equazioni differenziali presentate nel paragrafo precedente hanno una soluzione in forma chiusa (vedasi capitolo 3); in questa sezione ci concentreremo sulla soluzione più spesso adottata in letteratura per la simulazione di SNN, dove si tende ad utilizzare procedure numeriche che approssimano queste equazioni differenziali trattando il tempo come una quantità non continua ma discretizzata. La soluzione più adottata è l'integrazione di Eulero. I motivi per utilizzare questa tipologia di simulazione: una simulazione time stepped è

semplice da trattare e abbastanza prevedibile in termini di complessità computazionale. Inoltre permette di adottare in maniera relativamente semplice strategie di training che altrimenti sarebbero numericamente intrattabili. Il prezzo da pagare, come si può intuire, è sulla precisione con cui viene simulata la dinamica neuronale, che diventa solo un'approssimazione della dinamica reale. Senza perdita di generalità assumiamo $R = 1$, $V_r = 0$, $V_{th} = 1$, $I_{ext} = 0$. Di seguito sono presentate le soluzioni applicando appunto il metodo di Eulero:

$$V_i^l(t+1) = (\beta V_i^l(t) + I_{i,in}^l(t+1))(1 - S_j^{l-1}(t)) \tag{2.3}$$

$$\beta = e^{-\frac{\Delta t}{\tau_{mem}}}$$

$$I_i^t(t+1) = \alpha I_i^l(t) + \sum_j W_{ji}^{fw} \cdot S_j^{l-1}(t) + \sum_j W_{ji}^{rec} \cdot S_j^l(t) \tag{2.4}$$

$$\alpha = e^{-\frac{\Delta t}{\tau_{syn}}}$$

L'equazione del potenziale presentata utilizza il *reset-to-zero*, ma altri testi che trattano l'argomento potrebbero utilizzare un *reset-by-subtraction*, con l'equazione che potrebbe presentare, al posto del termine $(1 - S_j^{l-1}(t))$ il termine $-S_j^{l-1}(t)V_{th}$. Questo significa che dopo lo spike il potenziale del neurone verrà ridotto a quello che era il residuo oltre il threshold prima dello spike stesso, soluzione che in determinate situaizioni potrebbe migliorare l'accuratezza della rete [19, 18].

La simulazione timestepped, a livello algoritmico, è abbastanza semplice: fissato un intervallo di tempo Δt per la simulazione, lo stato dei neuroni viene aggiornato ad ogni passo. Questo comprende il processamento di ogni spike in ingresso at tempo t_1 per ciascun neurone, che determina i nuovi valori aggiornati di corrente e potenziale che questo assumerà a $t_1 + \Delta t$, e quindi se genererà uno spike a o meno. La gestione dei delay sinaptici solitamente avviene utilizzando un array circolare che tiene traccia di neuroni destinatari e tempi di arrivo degli spike futuri. Il compromesso tra accuratezza biologica e performance è già evidente: più si vuole rendere accurato il modello, più basso deve essere il timestep utilizzato, e il costo dell'algoritmo è direttamente pro-

porzionale al numero di step totali da simulare—e, di conseguenza, al rapporto tra tempo totale di simulazione e dimensione del timestep [37]; il discorso verrà ripreso nel prossimo capitolo.

2.2.1 Modelli di neuroni

Esistono diversi modi per utilizzare il modello presentato in precedenza, portandolo a diversi livelli di astrazione in base alle necessità. In questo paragrafo l'attenzione verrà concentrata sulla famiglia di modelli più utilizzati in letteratura, ovvero CUrrent BAsed Leaky Integrate-And-Fire (CUBA-LIF) e le sue diverse semplificazioni [6]. Per dettagli su altri modelli di neuroni si faccia riferimento a [39].

CUrrent BAsed Leaky Integrate-And-Fire (CUBA-LIF) Questo modello è il più realistico a livello biologico. Prende in considerazione per intero le dinamiche temporali legate al potenziale e alla corrente, facendo sì che i loro valori abbiano un decadimento esponenziale nel tempo. Le equazioni che governano il comportamento di un neurone di questo tipo sono le equazioni risolutive (2.3) e (2.4) presentate in sezione 2.2. Come si può notare, i due termini di decadimento sono αI_i e βV_i. Per le due costanti abbiamo che $0<\alpha<1$ e $0<\beta<1$, per valori positivi di τ_{mem} e τ_{sym}. Senza perdita di generalità possiamo considerare il time step adottato come unitario, così da avere $\Delta t = 1$. Il grado del decadimento esponenziale, quindi, sarà inversamente proporzionale al valore dei due τ_{mem} e τ_{syn}.

Leaky Integrate-And-Fire (LIF) È una semplificazione del modello precedente, il più utilizzato in letteratura nelle implementazioni di SNN. L'integrazione dell'input non prevede più un decadimento esponenziale per la corrente ma solo per il potenziale. Questo implica che, per avere una corrente che attraversa il neurone, sono richiesti spike in input continui. Di seguito sono

mostrate le equazioni che governano il comportamento dei neuroni LIF:

$$V_i^l(t+1) = (\beta V_i^l(t) + I_{i,in}^l(t+1))(1 - S_j^{l-1}(t)) \tag{2.5}$$

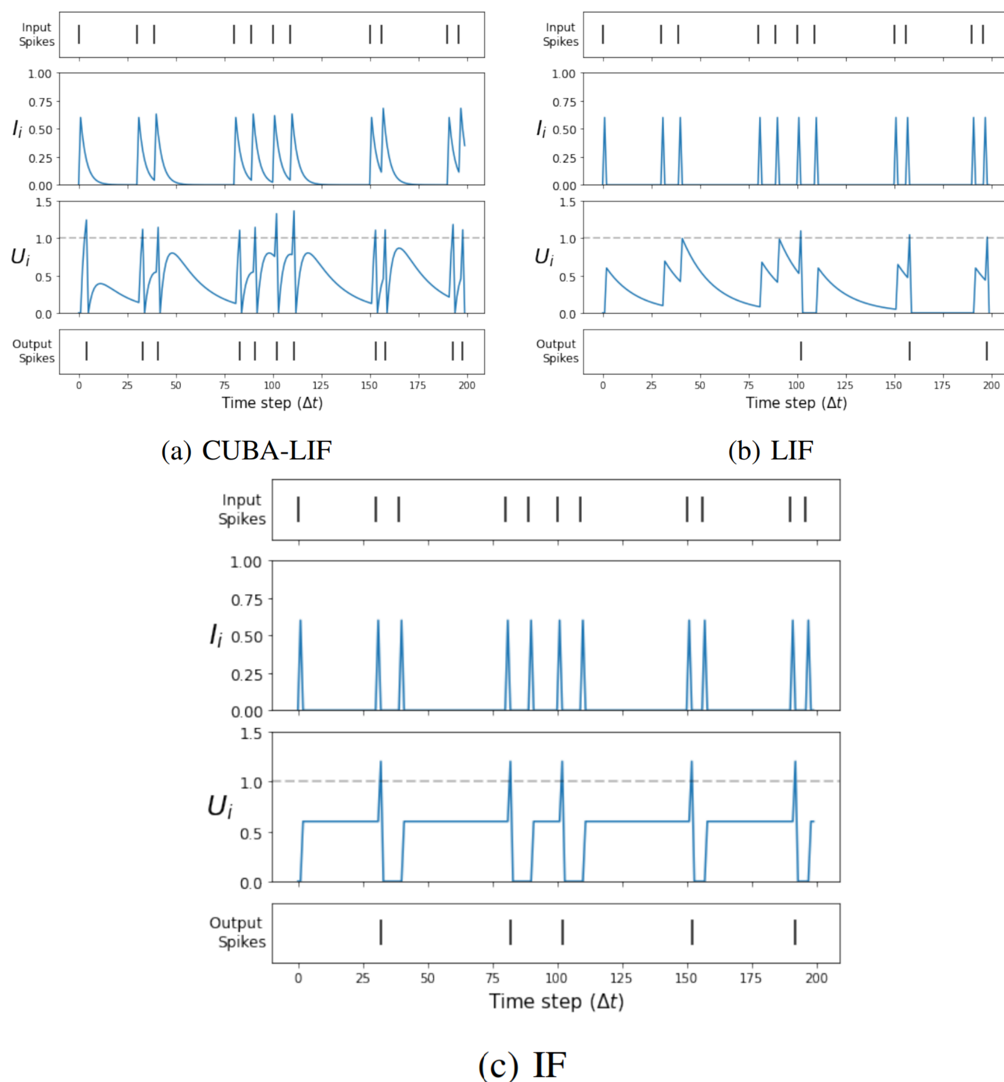
$$I_i^t(t+1) = \sum_j (W_{ji}^{fw} \cdot S_j^{l-1}(t) + \sum_j W_{ji}^{rec} \cdot S_j^l(t)) \tag{2.6}$$

Integrate-And-Fire (IF) È la semplificazione più grande del modello CUBA, nonché la meno realistica in confronto ai neuroni biologici. Come si può immaginare, questo tipo di neurone è simile ai precedenti solo che privo di termini di decadimento. In questo modo, tutte le dinamiche temporali sono perse, e non resta altro che un integratore che somma gli input. Questo è ottenuto impostando il parametro $\alpha = 0$ affinché ci sia un leakage infinito della sinapsi—il che si traduce in una corrente neuronale diversa da zero solo ed esclusivamente quando riceve degli spike—e il parametro $\beta = 1$ per avere un potenziale che rimane costante tra timestep successivi, in situazioni senza corrente esterna. Di conseguenza le equazioni che governano questo tipo di neurone sono:

$$V_i^l(t+1) = (V_i^l(t) + I_{i,in}^l(t+1))(1 - S_j^{l-1}(t)) \tag{2.7}$$

$$I_i^t(t+1) = \sum_j (W_{ji}^{fw} \cdot S_j^{l-1}(t) + \sum_j W_{ji}^{rec} \cdot S_j^l(t)) \tag{2.8}$$

Negli esperimenti presenti in letteratura, i parametri temporali legati al decadimento sono manetnuti costanti, a volte utilizzati come iperparametri, altre volte parte del processo di apprendimento. Anche questa è una semplificazione, come il modello di neurone in sé: a livello biologico è chiaramente osservabile che le dinamiche sono più complesse. Variazioni più o meno rapide nel potenziale, infatti, portano a variazioni di temperatura nella cellula, cambiamenti che influenzano a cascata la resistenza della membrana e dunque il valore delle costanti di tempo [12]. Per comprendere meglio i tre tipi di neuroni e le loro differenze si faccia riferimento alla Figura 2.2.



(a) CUBA-LIF (b) LIF

(c) IF

Figura 2.2: Confronto tra i 3 tipi di neuroni, fonte: [6]. L'U_i in queste immagini è ciò che prima è stato indicato con V_i

2.3 Training delle reti neurali timestepped

Esistono diversi approcci per allenare una rete SNN. Per gli scopi di questa tesi ne verranno approfonditi solamente due: uno che adatta la **backpropagation** al contesto SNN, e uno che utilizza una visione localizzata della rete per adattare le sinapsi in base al comportamento dei neuroni adiacenti. Per quest'ultimo si faccia affidamento al prossimo capitolo.

2.3.1 Reti neurali senza dinamiche temporali

Per trattare la prima stretegia iniziamo spiegando, in maniera non eccessivamente approfondita, quello che avviene in una rete neurale come il *multilayer perceptron*, in cui l'evoluzione della rete dipende solo da dinamiche spaziali, ovvero la posizione dei neuroni, la loro connessione e il peso delle loro connessioni, e non da dinamiche temporali, ovvero dall'istante in cui avviene qualcosa nella rete. Per altri approfondimenti si veda [28, 12]. Il problema dell'apprendimento, nella backpropagation, è modellato nel seguente modo: si cerca di minimizzare il valore di una certa funzione di costo generica $\mathcal{L}$, che rappresenta di quanto si discostano l'output reale $\mathbf{\hat{y}_i}$ dall'output desiderato $\mathbf{y_i}$, su tutto il dataset D. Per minimizzarlo, nella maggior parte dei casi si utilizza la regola della discesa del gradiente. Prendiamo una generica rete con soli due layer, uno di input e uno di output, con dei parametri $\mathbf{W}$ dove il peso della connessione tra un neurone j e un neurone i verrà indicato con $\mathbf{W_{ij}}$. Avremo allora che, ad ogni passo:

$$W_{ij} \leftarrow W_{ij} - \eta \Delta W_{ij}, \quad \text{con} \quad \Delta W_{ij} = \frac{d\mathcal{L}}{dW_{ij}} = \frac{d\mathcal{L}}{dy_i} \frac{dy_i}{da_i} \frac{da_i}{dW_{ij}} \tag{2.9}$$

In questa equazione y_i rappresenta l'output del neurone i, a reppresenta l'input del neurone, comunemente indicata come *funzione di attivazione*, η rappresenta un parametro di apprendimento che regola la sensibilità della variazione di quel parametro: un valore troppo grande potrebbe far oscillare troppo i parametri impedendo la convergenza verso un minimo, un valore troppo piccolo potrebbe rallentare eccessivamente l'addestramento o addirittura forzare il valore finale del peso in un punto subottimale. Senza scendere in dettagli matematici, nel caso di una rete con uno o più layer nascosti avremo:

$$\Delta W_{ij}^l = \delta_i^l y_j^{l-1} \quad \text{con} \quad \delta_i^l = \sigma' \left(a_i^l \right) \sum_k \delta_k^{l+1} W_{ik}^{T,l} \tag{2.10}$$

dove $\sigma'(a_i)$ è la derivata della funzione d'attivazione e T indica la trasposta. Questa operazione mostra come, partendo da uno stato finale conosciuto (e

avendo a disposizione l'input e conoscendo lo stato iniziale, possiamo ricavare lo stato finale) possiamo ripercorrere al contrario la rete, con una catena di derivate, per calcolare la variazione al peso da applicare ad ogni connessione di ogni layer. Per quanto riguarda la funzione di attivazione, in letteratura esistono diverse opzioni. Per i fini di questa tesi non verrà analizzata a fondo la questione, ma ci si limiterà a discutere di un problema noto che, in forma diversa, si presenterà successivamente in ambito SNN: il cosiddetto **vanishing/exploding gradient**. Ricordiamo infatti che la variazione dei pesi dipende da una catena di derivate: non ci sono problemi su reti con pochi layer, ma viene da chiedersi cosa succede su reti più ampie. Effettivamente, in maniera dipendente dalla funzione di attivazione, potrebbero esserci casi in cui il gradiente, procedendo a ritroso, diventa sempre più piccolo fino a scomprarire (cosa che porterà ad una totale assenza di apprendimento da quel punto in poi) opure diventa sempre più grande, esplodendo (portando a un'oscillazione eccessiva e una divergenza dall'ottimo). Diverse funzioni utilizzate, come la sigmoide o la tangente iperbolica, possono presentare questo problema. La ReLU invece, definita come

$$R(x) = \begin{cases} x & \text{se } x > 0 \\ 0 & \text{altrimenti} \end{cases},$$

non presenta il problema dell'exploding gradient dato che la sua derivata può valere solamente 1 oppure 0. Si potrebbe pensare che, nei casi in cui valga 0, si ripresenti il problema del vanishing gradient, ma in letteratura questa funzione si è dimostrata molto solida nei confronti di questo caso: anche se favorisce modelli sparsi (ovvero modelli in cui molti pesi vanno a 0), l'efficienza non sembra esserne intaccata [15].

2.3.2 Applicazione alle SNN

Prima di poter applicare i concetti visti in precedenza manca un passaggio. Il modo in cui una rete SNN, anche feed-forward (ovvero senza connessioni

ricorrenti) viene trattata per l'applicazione della backpropagation è analogo a ciò che avviene in una rete tradizionale con connessioni ricorrenti. Quest'ultima infatti viene ricondotta a una rete feed-forward con un procedimento detto srotolamento (**unrolling**) che, come indica il nome, prende la rete originale e rimuove le connessioni ricorrenti, trasformandole invece in connessioni verso una copia della rete che ne condivide i parametri come i pesi e che rappresenta il suo stato in un momento successivo, come in Figura 2.3.

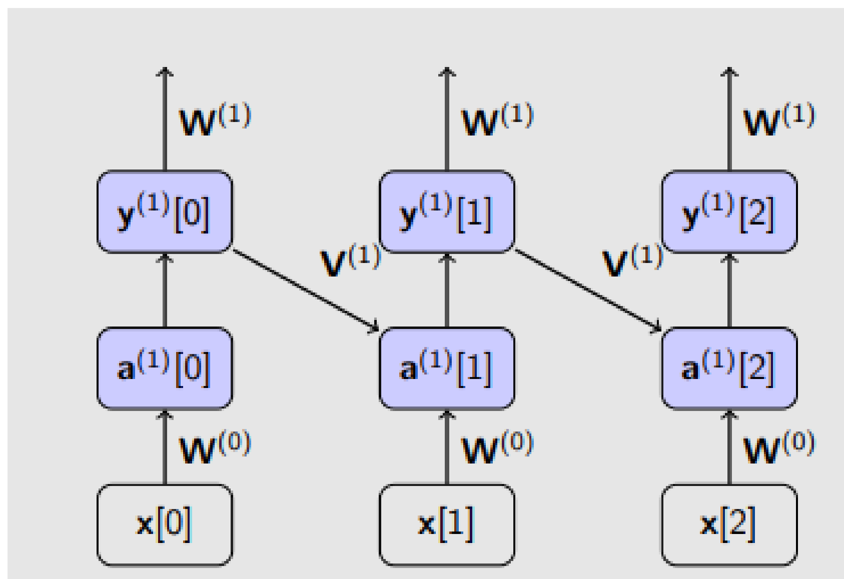


Figura 2.3: rappresentazione di una rete srotolata. I parametri sono condivisi. x[t] rappresenta l'input al momento t

Uuna rete SNN subirà lo stesso trattamento: verrà srotolata, e lo strato i-esimo rappresenterà lo stato della rete al momento $i \cdot \Delta t$, come in Figura 2.4. Ovviamente l'ampiezza dello srotolamento è direttamente proporzionale al numero di stati da rappresentare, e quindi dipenderà dalla durata della simulazione e dalla dimensione dell'unità Δt

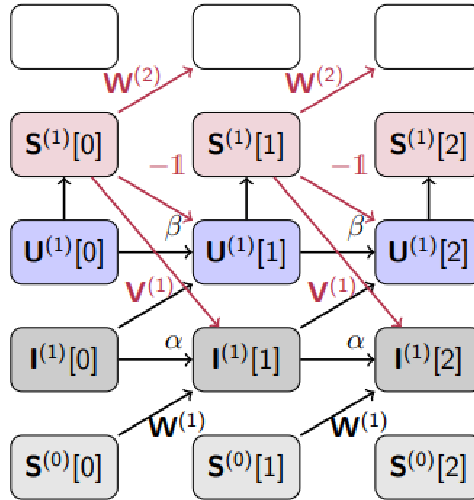


Figura 2.4: rappresentazione degli stati di una rete SNN srotolata

La funzione da ottimizzare, in questo caso, deve dipendere in qualche modo dagli spike. Nel caso di funzioni basate sullo *spike rate*, possiamo utilizzare funzioni basate sul numero di spike come in [40] o sul valore del potenziale, come in [17]. Per una panoramica generale: tra le strade percorribili scegliendo la prima soluzione abbiamo la possibilità di utilizzare una loss basata sulla cross-entropia, nel caso in cui abbiamo distribuzioni di probabilità per le varie classi che la rete deve riconoscere, oppure l'errore quadratico medio. Per la cross-entropia, il numero di spike viene utilizzato per calcolare la funzione softmax ed ottenere quindi dei valori di probabilità, da confrontare con i valori target. Nel caso dell'errore quadratico medio, vengono stabiliti dei valori target per il numero di spike delle varie classi (solitamente sotto forma di % sul numero di timesteps), da utilizzare per misurare la discrepanza fra output reale e previsto. In simulazioni che prevedono pochi timesteps, variazioni sui pesi potrebbero produrre con difficoltà variazioni sul numero di spikes. In quel caso si tende a preferire una funzione basata sul potenziale di membrana dei neuroni, anche se computazionalmente più onerosa. In questo caso, nell'approccio con cross-entropia, il valore massimo del potenziale viene utilizzato nella softmax, e l'obiettivo è incoraggiare il neurone della classe corretta a spikare di più portandolo ad avere potenziali più alti. Nell'approccio con l'errore quadratico

medio, un valore target di potenziale viene assegnato a ogni neurone ad ogni timestep. Esistono anche funzioni obiettivo basate sullo *spike time*, che tentano di ottimizzare per esempio il tempo del primo spike affinché appartenga alla classe corretta, tuttavia in letteratura i primi metodi sono più studiati e generalmente considerati migliori. Nel caso di SNN le modifiche ai pesi devono riflettere sia il cosiddetto *spacial credit assignment*, ovvero la quantità di "colpa" che ha ciascun peso in funzione di quanto ha contribuito al' errore della rete, sia il *temporal credit assignment*, ovvero in quale momento, durante l'evoluzione temporale di una rete, un certo peso ha influito sull'errore. La backpropagation è una soluzione solida per risolvere entrambi questi problemi. Per approfondimenti su quanto detto in precedenza si faccia riferimento a [12].

2.3.3 Problematiche dell'SNN

Per applicare quanto detto in precedenza bisogna risolvere dei problemi intrinseci all'architettura delle SNN. Ricordiamo che, per come l' abbiamo definita nel primo capitolo, la funzione di attivazione non è differenzialbile.

$$S(U(t)) = \Theta(V(t) - V_{th}) \quad con \quad \Theta = f.Heaviside \tag{2.11}$$

La derivata vale ovunque 0, tranne nel punto in cui $V(t) = V_{th}$ in cui esplode a $+\infty$.

Ci sono diversi modi per arginare questo problema. Alcune soluzioni si basano su modelli di neuroni particolari che non presentano questo problema, altre soluzioni trattano gli spiketime come fenomeni stocastici, sui cui valori attesi può essere definito un gradiente [27]. Questi metodi non sono di interesse per la tesi.

Una delle soluzioni più popolari è l'utilizzo di un *gradiente surrogato*: per ogni occorrenza del gradiente problematico $\frac{dS(t)}{dV(t)}$, questo viene approssimato utilizzando una funzione differente $\hat{S(t)}$, appiattita e il cui gradiente non

esplode. Funzioni comunemente adoperate sono:

$$Sigmoide \quad S(\cdot) = \frac{1}{1 + e^{(V_{th} - V(t))}} \tag{2.12}$$

$$Tangente \quad S(\cdot) = \frac{1}{\pi} arctan(\pi(V_{th} - V(t))) \tag{2.13}$$

Ovviamente queste sono solo alcune tra le possibili opzioni per il gradiente surrogato; per una panoramica su altre soluzioni possibili e sul loro confronto si faccia riferimento alla Figura 2.5.

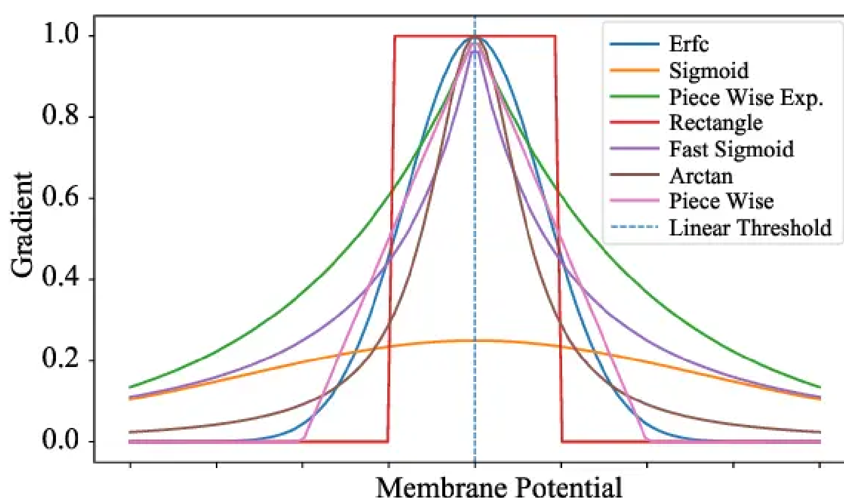


Figura 2.5: Confronto fra varie funzioni per il gradiente

Con questa soluzione abbiamo dato al gradiente una forma che permette il learning, ma resta un ultimo problema, quello che in letteratura viene denominato *problema del neurone morto.* Un neurone che non effettua uno spike, infatti, non contribuirà al learning: il gradiente della funzione di spike sarà zero, il peso non subirà variazioni e quindi il neurone non ha più alcun modo di cambiare la sua situazione. Per cercare di arginare questo problema, si potrebbe utilizzare una funzione surrogata non solo per i punti problematici in cui il neurone effettua uno spike (soluzione che approssima bene il problema e che introduce poco bias), ma si potrebbe rimpiazzare completamente la funzione di spike con la surrogata: in questo modo si introduce un bias molto significativo data dalla grande variazione tra valore del gradiente reale e surrogato, ma permette anche a neuroni con bassissima attivazione di imparare [12].

2.3.4 Plausibilità a livello biologico

La backpropagation mostrata nei paragrafi precedenti, si è dimostrata efficace in termini di capacità di miglioramento dell'accuratezza di una rete. La cosa non sorprende: la modellazione del problema tramite una funzione di loss, che ha portato alla tecnica della backpropagation, è un approccio studiato e utilizzato da tempo. La Back Propagation Trough Time (BPTT) (il nome del tipo particolare di backpropagation visto per le SNN, quello che srotola la rete neurale per trattare le dinamiche temporali come se fossero una semplice estensione della rete stessa) è efficace per gli stessi principi. Tuttavia, a livello biologico, le cose sono più complicate. Anche supponendo che i vari layer di neuroni nel cervello siano in grado di calcolare gradienti e mantenerli, applicare la fase di propagazione all'indietro implicherebbe la presenza di connessioni bidirezionali tra i vari neuroni del cervello, tra ogni strato. Non c'è alcuna evidenza scientifica sul fatto che questa sia una possibilità, e gli assoni del cervello propagano il segnale in maniera unidirezionale. I meccanismi reali del cervello sono ancora un mistero. Un'altra complicazione è data dal fatto che l'applicazione del BPTT richiede una definizione di un tempo Δt su cui operare, cosa chiaramente in contrasto con la realtà dove il tempo scorre in maniera continua. Esistono in letteratura alcune alternative, che in certi casi di test si sono rivelate efficaci, e che rilassano i vincoli della BPTT legati alla propagazione all'indietro rendendole più biologicamente plausibili, ma non è ancora chiaro quanto queste siano realistiche. [12]

3. Simulazione di SNN Precisa

Come è stato discusso nella sezione 2.2, la simulazione timestepped presenta delle limitazioni significative. Una di queste è legata al forte accoppiamento tra dimensione del timestep e accuratezza della rete simulata: un timestep maggiore porta a una rete facile e rapida da simulare al prezzo di un basso valore di accuratezza, data dalla grandissima approssimazione fatta sui tempi degli spike, e viceversa. Per reti con basso timestep, un altro limite è dato dalla necessità di valutare lo stato dei neuroni ad ogni timestep. In realtà, come è facile immaginare, questi ultimi potrebbero essere quiescenti per diverso tempo—e in alcuni casi anche per sempre—prima di essere sollecitati in qualche modo e di generare spike. Per questa ragione, un paradigma di esecuzione diverso potrebbe essere più adatto: in questo capitolo verrà presentato brevemente il metodo **Discrete Event Simulation (DES)** [11] (e **Parallel-DES (PDES)** [14]), il simulatore che implementa questa metodologia su cui il resto della tesi si concentrerà e infine come i modelli neurali SNN vengono implementati su questo simulatore. Infine, ci sarà una panoramica su come effettuare il training dei pesi nel contesto di una simulazione di questo tipo.

3.1 DES

La metodologia DES permette di simulare il comportamento di un sistema generico. Questo avviene assumendo che ciascun elemento che compone il sistema da simulare abbia uno stato che varia solo in specifici momenti nel tempo; proprio per questo lo scorrere del tempo non è visto come continuo, ma come un salto da un evento all'altro, e ciò che avviene nel mezzo è considerato

irrilevante. Quelli che abbiamo definito prima come eventi sono accadimenti generici che modificano lo stato del sistema. Quest'ultimo viene partizionato in sotto-unità, dette **Logical Processes (LPs)**. A questi concetti si aggiunge quello del **Virtual Time (VT)**, il tempo logico dell'esecuzione, in contrasto al **Wall Clock Time (WCT)**, il tempo reale. La simulazione dunque è composta da tre elementi: gli LPs con il loro stato, gli eventi che possono accadere nell'evoluzione dello stato del sistema, e gli effetti che ciascuna tipologia di evento può avere sul sistema e sullo stato stesso. Per essere più schematici, dunque, un simulatore che implementa questo paradigma deve avere una serie di componenti:

- Stato della simulazione: come detto prima, lo stato della simulazione che è composto dallo stato dei singoli LPs di cui è formato. Questo stato evolve nel tempo in base agli eventi che si verificano

- Eventi: questi vengono definiti a priori e indicano cosa può accadere durante la simulazione, e di conseguenza quali cambiamenti di stato—ed eventualmente altri eventi generati a cascata—si verificano in risposta ad un evento, che viene gestito attraverso un *event handler*. Oltre a quelli previsti dal sistema che si sta simulando, si tende ad aggiungerne due fittizi: un evento di *INIT* per inizializzare lo stato del sistema, delle strutture dati, e per schedulare i primi eventi, e uno di *DEINIT* al termine della simulazione per eventuali azioni da fare prima di concludere la simulazione

- Clock: elemento che tiene traccia del tempo virtuale della simulazione, che regola l'avanzamento della simulazione e che avanza di evento in evento, con l'unica accortezza di rispettare l'ordine causale degli eventi(concetto che verrà ripreso successivamente)

- Coda degli eventi: una coda in cui vengono memorizzati gli eventi futuri in ordine temporale (basato sul tempo virtuale). Difficilmente si avrà un evento per volta da eseguire in futuro, dunque questi necessitano di

una coda di priorità in cui essere inseriti. Siccome verranno eseguiti in ordine, devono essere memorizzati in ordine di timestamp

- Condizione di terminazione: Dato che lo stato del sistema evolve in continuazione e, potenzialmente, all'infinito partendo dalle condizioni iniziali, è necessario avere una condizione che possa permettere alla simulazione di terminare. Tipicamente si tratta di un vincolo sul tempo di simulazione, o sullo stato del sistema

In pratica, un simulatore che implementa DES può essere scomposto in due macroelementi: un kernel, che effettua le operazioni necessarie per far funzionare la simulazione seguendo correttamente il paradigma di cui abbiamo parlato e il modello, che definisce la parte specifica al sistema che si vuole correntemente simulare, ovvero lo stato del sistema e gli eventi, e dunque i loro handler, che il kernel deve chiamare durante la sua esecuzione (eccezion fatta per INIT/DEINIT). In maniera molto semplificata, il kernel dunque può essere visto come l'insieme di due operazioni, una fase di **INIT** e il **SIMULATION LOOP**. La prima imposta il VT al suo valore iniziale (solitamente 0), il flag della condizione di uscita a falso, inizializza eventuali strutture dati ed altri elementi necessari all'esecuzione e chiama l'handler della *INIT* (o schedula l'evento con timestamp 0). Una volta terminato questo, il *SIMULATION LOOP* ha inizio: il prossimo evento dalla coda viene pescato, il VT viene aggiornato con il valore del suo timestamp e il corretto handler per quell'evento viene invocato. Da qui l'handler, e quindi il modello, prende il controllo: l'evento viene eseguito, eventualmente altri vengono aggiunti in coda e lo stato viene aggiornato. Prima di tornare all'inizio del ciclo, la condizione di uscita viene verificata: se falsa, si continua.

3.2 PDES

Utilizzare PDES significa, come intuibile, parallelizzare l'esecuzione di una DES. Questo permette potenzialmente un notevole guadagno di prestazio-

ni, ma al costo di alcuni accorgimenti da prendere che verranno analizzati di seguito.

3.2.1 Parallelizzazione

In primo luogo, la nozione di Logical Process (LP) assume un'importanza maggiore, dato che è l'elemento che permette la parallelizzazione. Paralelizzando l'esecuzione di DES ciascun LP varrà mappato su una specifica istanza di kernel in esecuzione, e quindi processo su un determinato processore. Di conseguenza, avremo diversi LP che descrivono, nella loro interezza, il sistema. Ciascuno di questi, nell'evoluzione del proprio stato e quindi degli eventi che lo definiscono, si troverà in un punto del tempo (virtuale) differente: per questo motivo ognuno di essi avrà un VT privato detto **Local Virtual Time (LVT)**. Oltre a questo, ci sarà anche il **Global Virtual Time (GVT)**, che indica il punto in cui la simulazione è effettivamente *committed* (concetto che verrà ripreso in seguito). Questa separazione tra gli LP implica anche che i loro stati devono essere separati, e quindi il modello non deve utilizzare variabili condivise, per evitare problemi di causalità. Supponiamo infatti di avere due LP LP_1 e LP_2, con $LVT_1 > LVT_2$ e in due momenti WCT_1 e WCT_2. Questi condividono una variabile x. Se LP_1 scrive in x al tempo LVT_1, nel caso in cui $WCT_1 < WCT_2$ avremo una violazione data dal fatto che LP_2 leggerà questo valore, già aggiornato, al tempo LVT_2: una situazione in cui il futuro ha avuto una ripercussione sul passato. Il problema della causalità, come vedremo di seguito, non è esaurito qui.

3.2.2 Sincronizzazione

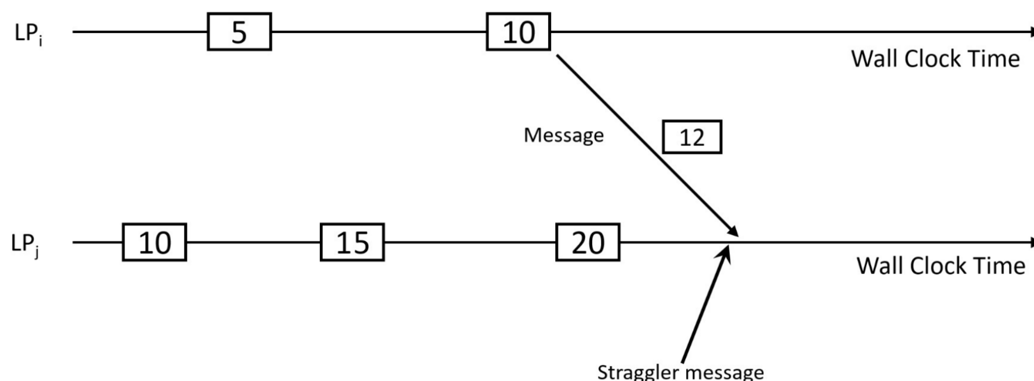


Figura 3.1: Problema di sincronizzazione, fonte: [30]

Per capire questo problema, facciamo riferimento alla Figura 3.1. Immaginiamo una situazione in cui due LP LP_j e LP_i ad un determinato WCT abbiano un $LVT_i = 10$ e $LVT_j = 20$. Entrambi devono eseguire il prossimo evento che si trova sulla propria coda (ricordiamo che che LP differenti sono mappati su kernel differenti e hanno code diverse), e quindi il valore dei rispettivi LVT corrispondono al timestamp dell'evento appena pescato dalla coda. L'evento e_j^{20} viene eseguito senza problemi, mentre l'evento e_i^{10}, nella sua esecuzione, genera un nuovo evento per l'LP_j con timestamp 12. Quest'ultimo si troverà allora nella situazione in cui, anche se il valore corrente del LVT è 20, ha nella coda un evento la cui esecuzione sarebbe dovuta avvenire nel passato. L'evento che causa la desincronizzazione è detto *straggler message*. Questa desincronizzazione, che chiaramente compromette la correttezza della simulazione, è conseguenza naturale della parallelizzazione della simulazione.

In letteratura esistono due approcci differenti per risolvere il problema: quello **conservativo** [8] e quello **ottimistico** [21]. Il primo evita del tutto l'occorrenza di problemi di sincronizzazione, eseguendo gli eventi solo ed esclusivamente quando sono ritenuti sicuri. Il secondo prevede l'esecuzione degli eventi senza curarsi di nulla, nella speranza che non avvengano problemi di sincronizzazione: questo implica che siano previsti meccanismi per individuare

le violazioni, qualora accadano, ed attuare azioni correttive. Le due strategie hanno pro e contro che dipendono dalla specifica simulazione, e in alcuni casi un approccio ibrido potrebbe essere il migliore. Ai fini di questa tesi ci concentreremo sull'approccio ottimistico.

3.2.3 Sincronizzazione ottimistica (Time Warp)

Come abbiamo detto, in questo caso la simulazione procede in maniera speculativa, con la speranza che non si verifichino problemi di causalità. Questo significa che in termini di tempo, in confronto all'altra strategia, si paga una penalità quando bisogna buttare i progressi di una simulazione a causa di uno *straggler message*. Se questo non avviene, si è guadagnato tempo proseguendo con la simulazione anche quando non si avevano garanzie. Cosa avviene nel caso in cui si verifica una violazione? C'è bisogno di effettuare un **rollback**, ovvero un ripristino di uno stato con un timestamp precedente a quello della violazione, in cui questo è ancora consistente. Oltre allo stato dell'LP, si ripristina anche il valore del LVT e il contenuto della coda degli eventi. Questa soluazione non è banale come sembra: ricordiamo infatti che durante l'esecuzione degli eventi spesso ne vengono generati e propagati altri. Questo significa che, durante un'esecuzione speculativa, se avviene un rollback, anche i messaggi inviati durante questo lasso di tempo devono essere rimossi. In pratica questo avviene con l'utilizzo degli *antimessaggi*, che non sono altro che copie dei messaggi di cui effettuare il rollback ma con segno opposto. Alla ricezione di un antimessaggio generico A_i^t, l'LVT_i dell'LP_i può essere precedente al timestamp dell'antimessaggio, ed in quel caso significa che l'evento in coda generato dal messaggio da annullare è ancora lì, e l'unico effetto dell'antimessaggio è la sua rimozione dalla coda. Al contrario, potrebbe essere l'antimessaggio ad essere precedente, il che implica che l'evento da annullare è già stato eseguito. Questo, ovviamente, porta ad un'azione di rollback anche da parte dell'LP_i [37].

3.2.4 Rollback

Come avviene in pratica il rollback? Anche qui esistono due approcci profondamente diversi. Uno è la **computazione inversa** [7] che, come dice il nome, prevede che dallo stato corrente si torni ad uno stato precedente invertendo gli effetti degli eventi avvenuti nel tempo ripercorrendoli all'indietro. Questo implica una complessità aggiuntiva a livello computazionale, e non è scontato che si possa sempre fare. Un secondo approccio è quello della **copia dello stato e ripristino** [13] dove gli stati degli LP, a determinati istanti temporali vengono memorizzati e mantenuti in memoria (**snapshot**) e, all'occorrenza, ripristinati in seguito a una violazione. Questo approccio, se non gestito con attenzione, può introdurre notevole overhead in termini di uso memoria [33, 38]. Per gli scopi di questa tesi, verrà analizzato nello specifico quest'ultimo approccio.

L'efficacia dell'approccio dipende interamente dal compromesso sulla memoria. Risulta evidente infatti che, in una situazione ideale con memoria infinita e latenze trascurabili, potremmo effettuare uno snapshot ad ogni evento e tornare indietro a grana molto fine. Ovviamente questa è una situazione ideale, nella pratica bisogna effettuare dei compromessi.

Una prima soluzione che ci permette l'implementazione di questo approccio riguarda il già precedentemente citato GVT, definito come il timestamp minimo di un qualsiasi messaggio o antimessaggio ad un certo WCT=t. Da questa definizione e dall'osservazione che nessun evento e_1 con $LVT_1 = t_1$ può generare un messaggio—e quindi un altro evento e_2—rivolto nel passato, ma solo ed esclusivamente futuro e quindi con $LVT_2 \geq LVT_1$, possiamo dedurre che il GVT rappresenta il tempo di *commit*: gli eventi con timestamp inferiore sono già eseguiti e non dovranno essere ritoccati, siccome nessun evento con timestamp $\geq$ GVT potrà causare uno *straggler message* nel passato. Tutti gli snapshot con timestamp $<$ GVT, dunque, si possono scartare (operazione chiamata **fossil collection**). Il calcolo del GVT avviene ogni intervallo di tempo detto **GVT period**. La seconda soluzione adottata nelle implementazioni di

questi simulatori riguarda l'intervallo di checkpointing. Simulazioni complesse prevedono una quantità enorme di eventi, il che rende impensabile avere uno snapshot ad ognuno di essi. Bisogna quindi stabilire un intervallo di eventi minimo che deve intercorrere tra uno snapshot e l'altro. Questo può essere un valore fissato a priori oppure calcolato a runtime ad intervalli di tempo variabili in base all'andamento della simulazione. Teniamo presente che un valore troppo basso di questo intervallo significa un grande utilizzo di memoria, mentre un valore troppo alto significa un'elevata sparsità di snapshot, il che implica dover rieseguire, ad ogni rollback, un gran numero di eventi (e quindi un doppio utilizzo di risorse per loro). Una trattazione approfondita su questo argomento esula dagli scopi di questa tesi. Per ulteriori dettagli su modelli di calcolo del checkpoint interval si faccia riferimento a [13].

Per conlcudere il paragrafo, c'è da menzionare un'ultima cosa: le equazioni (2.1) e (2.2) sono state risolte in forma chiusa, in modo che siano adatte al contesto della simulazione precisa. Le nuove soluzioni sono [36]:

$$V(t) = I_0 A_1 \alpha + (V_0 - A_2 - I_0)\beta + A2 \tag{3.1}$$

$$I(t) = \alpha I_0 \tag{3.2}$$

con

$$A1 = \frac{1}{\left(\frac{1}{\tau_{mem}} - \frac{1}{\tau_{sym}}\right) C_m}$$

$$A2 = V_r + \tau_{mem} \frac{I_{ext}}{C_m}$$

$$\tau_{mem} = RC \tag{3.3}$$

Ricordiamo inoltre che:

$$\alpha = e^{-\frac{\Delta t}{\tau_{syn}}} \quad \beta = e^{-\frac{\Delta t}{\tau_{mem}}}$$

3.3 The ROme OpTimistic Simulator (ROOT-Sim)

Per la simulazione di SNN, è stato utilizzato ROOT-Sim [32]. ROOT-Sim è il kernel di un simulatore PDES che utilizza un paradigma di sincronizzazione ottimistico, scritto interamente in linguaggio *c*. Come di consueto quando si implementa questo paradigma, prevede un forte accoppiamento tra messaggi ed eventi: nello specifico, ciascun messaggio incorpora un evento per l'LP destinatario. Il simulatore prevede un evento di INIT in cui lo stato di ciascun LP viene inizializzato; inoltre fornisce una libreria numerica per la generazione di numeri casuali secondo varie distribuzioni, libreria su cui fare riferimento dato che permette di rispettare le esigenze nate dall'applicazione dei rollback, e quindi di ripristinare, oltre allo stato degli LP, anche la sequenza di valori generati al punto corretto nel tempo. I dettagli implementativi di più basso livello non sono di interesse per questa tesi, per approfondire si faccia riferimento a [31]. Il simulatore prevede il calcolo dinamico del checkpointing interval seguendo un modello apposito i cui dettagli si possono vedere qui: [33]. Su ROOT-Sim è stato realizzato un modulo che fa da interfaccia tra il simulatore e il modellista, ottimizzato in ottica di simulazione SNN, anch'esso scritto in linguaggio c [35]. Questo modulo maschera la complessità di ROOT-Sim mettendo a disposizione delle API che gestiscono la comunicazione con il kernel e delle interfacce lasciate a carico del modellista, che semplificano il suo lavoro. Tra queste abbiamo:

- Spike/SendSpike: queste API permettono di schedulare un evento di spike ad un tempo specifico, utile per utilizzare lo spiketrain nel modello. La prima schedula l'evento sull'LP corrente, la seconda permette di specificare l'LP target. Nella pratica, questo si concretizza con una serie di messaggi inviati a ciascun neurone postsinaptico, contenente il peso della sinapsi, che per il momento è considerato costante.

- MaybeSpike/MaybeSpikeAndWake: queste API inseriscono nella coda

un evento di tipo MAYBESPIKE/MAYBESPIKEWAKE, uno spike che avverrà, ipoteticamente, se il neurone non ne riceverà altri in input nel frattempo. Il neurone tiene traccia di un contatore che incrementa progressivamente ad ogni messaggio ricevuto. Il valore di questo contatore viene copiato in uno dei campi dell'evento quando questo viene schedulato: quando l'evento deve essere eseguito, basterà controllare che il valore salvato sia uguale a quello corrente. Se i valori corrispondono, lo spike deve essere davvero materializzato, e l'effetto sarà lo stesso di SendSpike. La seconda API, inoltre, dopo lo spike del neurone invocherà l'interfaccia NeuronWake.

- NeuronWake: quest'interfaccia, a carico del modellista, viene invocata dopo l'esecuzione di un evento MAYBESPIKEWAKE. Il suo compito è aggiornare lo stato del neurone e schedulare il prossimo eventuale spike in base allo stato corrente.

- NeuronHandleSpike: interfaccia invocata quando un neurone riceve uno spike da parte di un altro neurone, con il compito di aggiornare il suo stato ed eventualmente schedulare un prossimo spike.

- SynapseHandleSpike: interfaccia invocata all'invio di uno spike, che permette di modificare dinamicamente il valore delle sinapsi e quindi il valore ricevuto dal neurone postsinaptico, questo per ogni sinapsi a cui il messaggio viene inviato.

- NeuronInit: implementata dal modellista, deve inizializzare lo stato del neurone corrente (il cui identificativo è ricevuto come parametro). Per fare ciò si utilizzano le funzioni rs_alloc, messe a disposizione da ROOT-Sim, per allocare memoria rollbackabile in maniera gestita autonomamente dal simulatore.

- Connect: API che permette di creare una sinapsi tra il neurone corrente e un neurone postsinaptico generico, che restituisce un puntatore alla struttura dati della sinapsi.

Per ulteriori dettagli implementativi, si faccia riferimento all'analisi del caso di studio specifico nel prossimo capitolo.

3.4 Training con questo tipo di simulazione

Come effettuare il training di una SNN così simulata, in letteratura, è un problema poco affrontato. Di seguito vedremo una possibile soluzione, frutto di idee e studi effettuati in nell'arco degli ultimi decenni. Uno dei punti di partenza è la supposizione che il timing relativo degli spike dei neuroni pre e postsinaptici fosse importante per determinare come il valore del peso delle sinapsi varia nel tempo. L'ipotesi è stata messa alla prova, scoprendo che, presi due neuroni connessi da una sinapsi, se il neurone presinaptico dovesse effettuare uno spike poco prima del neurone postsinaptico (diciamo, per esempio, 10ms prima) per un'elevato numero di volte, allora c'è una correlazione tra gli spike dei due neuroni e la connessione ne esce rafforzata. Al contrario, se lo spike presinaptico avviene dopo quello postsinaptico, la connessione ne esce indebolita. Questo è un meccanismo di **plasticità sinaptica**, ovvero un modo di potenziare o deprimere la forza delle sinapsi dei neuroni [25].

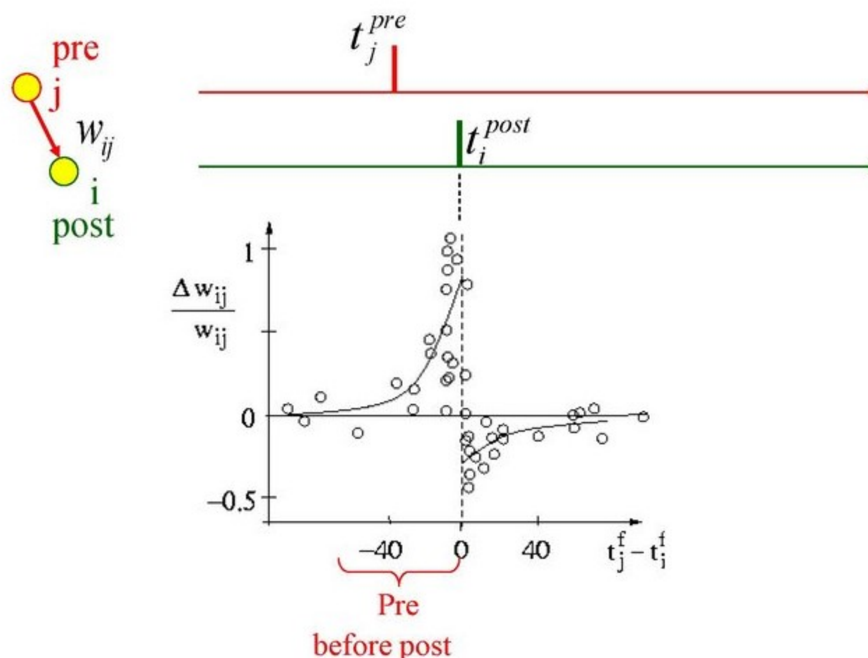


Figura 3.2: Variazione peso sinaptico in base alla correlazione dei tempi di spike, fonte: [41].

3.4.1 Spike-Timing Dependent Plasticity (STDP)

Un approccio basilare per valutare quanto il peso di una sinapsi w_{ij} varia nel tempo, e quindi il Δw_{ij}, è il seguente. Teniamo in conto, per ciascuna sinapsi, i tempi di arrivo a destinazione di uno spike da parte del neurone presinaptico i, che ovviamente sarà funzione dei tempi di spike di i e del delay sinaptico tra i due neuroni, e i tempi di spike del neurone postsinaptico j. Indichiamo con t_i^n i tempi di spike del neurone presinaptico i, dove n = 1,2,3... rappresenta il conteggio degli spike. Allo stesso modo facciamo con t_j^m per il neurone postsinaptico. La variazione del peso della sinapsi, allora, si può calcolare come:

$$\Delta w_{ij} = \sum_{n=1}^{N} \sum_{m=1}^{M} W(t_j^m - t_i^n) \tag{3.4}$$

$W(x)$ denota una funzione STDP, detta anche *learning window*. Tra le diverse opzioni la più popolare in letteratura è [41]:

$$W(x) = A_+ exp(-x/\tau_+) \quad per \quad x >= 0$$

$$W(x) = -A_- exp(x/\tau_-) \quad per \quad x < 0 \tag{3.5}$$

Ma ne esistono anche altre, meno popolari, come i cosiddetti *mirrored STDP* o *probabilistic STDP* [34]. In particolare, per il secondo, abbiamo che la learning window dipende più dal peso sinaptico corrente che dalla relazione tra i tempi di spike:

$$W(x) = A_+ exp(-w_{ij}) \quad per \quad x >= 0$$

$$W(x) = -A_- \quad per \quad x < 0 \tag{3.6}$$

La relazione tra i tempi di spike, in questo caso, è rilevante solo per determinare se e quando la sinapsi viene rafforzata o depotenziata.

Il parametro moltiplicativo A viene fissato a priori, mentre il valore di τ, che informalmente si può dire che rappresenta l'intervallo di tempo significativo in cui la presenza di spike consecutivi tra i due neuroni ha un'influenza sulla variazione del peso della sinapsi, è tendenzialmente di 10ms [41]. Se questo modo di applicare STDP prevede una preventiva esecuzione in cui si raccolgono i dati (dunque offline), esistono strategie per applicarlo anche online durante la simulazione, basate sul fatto che quando un neurone presinaptico i effettua uno spike ad un tempo t_i^n, questo lascia una traccia in questo neurone, traccia ultilizzata dal neurone postsinaptico j per aggiornare il valore della sinapsi che li collega quando effettuerà uno spike al tempo t_j^n [41].

Per ragioni di plausibilità biologica potrebbe essere rilevante che i pesi sinaptici w abbiano dei limiti (bound), e quindi che $w_{min} < w_i < w_{max}$, questo per ogni sinapsi. Per far ciò, si possono utilizzare funzioni appostite A(w_i) che quindi assegnano un valore al parametro moltiplicativo in base ai pesi sinaptici, per esempio

- Soft Bounds:

$$A_+(w_i) = (w_{max} - w_j)\eta_+ \quad A_-(w_i) = (w_j - w_{min})\eta_- \tag{3.7}$$

- Hard Bounds:

$$A_+(w_i) = \Theta(w_{max} - w_j)\eta_+ \quad A_-(w_i) = \Theta(w_j - w_{min})\eta_- \tag{3.8}$$

La funzione Θ è la stessa funzione heaviside utilizzata nei capitoli precedenti. Nel caso del soft bound, una sinapsi dal peso più vicino al limite superiore subirà maggiormente l'effetto delle depressioni rispetto a quello dei potenziamenti, e una volta superato il limite subirà l'effetto inverso (quindi verrà depotenziata anche nel caso in cui la relazione tra gli spiketime prevedesse un potenziamento, e viceversa). Nel caso dell'hard bound, il peso della sinapsi subirà un potenziamento (o una depressione) proporzionale a η_+ (o η_-), solo finchè si troverà entro i confini prestabiliti. I valori di η_+ ed η_- si possono scegliere a piacere, finché per ambedue $\eta > 0$ e $\eta << 1$.

4. Caso di Studio

Parte del lavoro di questa tesi consiste in una valutazione dell'efficienza della simulazione SNN precisa, su un modello specifico descritto in [26]. Il presente capitolo è così strutturato: la prima parte spiegherà il lavoro effettuato nel paper di riferimento che è di interesse per questa tesi, i risultati che sono stati ottenuti, e la rete utilizzata, in modo da aver chiari il modello adoperato ed i termini utilizzati. Dopo verrà presentato il lavoro della tesi, nell'ordine: la raccolta di dati e informazioni per replicare il modello con ROOT-Sim, l'implementazione con ROOT-Sim di questo modello e di un'infrastruttura per effettuare il training adatta al caso.

4.1 Letter Recognition

Il paper si propone di analizzare il problema del riconoscimento di pattern spazio-temporali. Questo avviene, nello specifico, sul problema della lettura di lettere in braille. Lo scopo del paper è quello di analizzare i risultati non solo in termini di accuratezza ma anche di efficienza energetica, per quanto riguarda diverse tipologie di reti neurali e diverse simulazioni, anche su hardware specifico per SNN (Intel LOIHI [9] per esmpio). Ai fini di questa tesi ci si concentrerà sulla raccolta dei dati e sulla costruzione delle reti SNN.

La lettura del braille, a differenza della lettura normale, è un'attività sequenziale: non si può avere una visione di insieme di un'intera parola o di un gruppo di parole—infatti, leggendo normalmente, l'occhio e la mente sono in grado di effettuare operazioni di comprensione e predizione sorprendentemente rapide—, ma bisogna procedere lettera per lettera per comporre prima ciascu-

na parola, e poi la frase nella sua interezza. La natura sequenziale e dipendente dal tempo di questo problema lo rende un caso di studio interessante in ambito SNN, che ricordiamo essere delle reti con dinamiche temporali incorporate. In questo caso di studio specifico, a differenza di molti altri che affrontano il problema, non si utilizza una classificazione ottica ma una classificazione tattile. Per farlo, l'idea è di utilizzare sensori *event-driven*, ovvero sensori che sono in grado di trasmettere segnali solo nel momento in cui avviene una variazione significativa del segnale, nel dominio che sono in grado di valutare. Anche questo comportamento è particolarmente adatto per la riproduzione del tatto, che biologicamente funziona in maniera simile. Per la raccolta del dataset è stato utilizzato un robot che facesse scorrere un polpastrello dotato di 12 sensori in posizioni diverse. La velocità e la distanza con cui lo scorrimento è stato effettuato, è stata fissata a priori, e ovviamente la dimensione delle stampe delle lettere è stata realizzata appositamente per combaciare con la dimensione del finto polpastrello, affinché questo sia in grado di percepire la lettera per intero. Per ottenere il risultato di un sensore tattile event-based, che al momento della realizzazione dello studio di questo paper non esisteva, sono stati interpretati i valori del sensore reale come segue: ad ogni aumento/diminuzione di pressione di un certo valore θ, un evento ON/OFF viene generato. Per questo lavoro sono stati utilizzati quattro differenti soglie (**threshold**) θ con valori 1, 2, 5 e 10, dove 1 rappresenta la precisione massima e quindi nessuna perdita di informazioni. Risulta evidente quindi che con il primo threshold avremo il maggior numero di eventi, mentre con gli altri gli eventi sono più sparsi nel tempo, cosa che offre una minore precisione ma anche un minore overhead.

Per quanto riguarda nello specifico la rete SNN, il paper prende in considerazione una rete neurale composta da un solo layer intermedio (hidden). Questa rete prevede due versioni. La prima è la Feed-Forward SNN (FFSNN), in cui abbiamo una connessione completa (sinapsi) tra tutti i neuroni del layer di input e quelli dell'hidden layer, e poi tra tutti quelli dell'hidden layer e quelli dell'output layer, mentre la seconda è la Recurrent SNN (RSNN), in cui, oltre alle sinapsi della FFSNN abbiamo anche uno strato completamente

ricorrente di sinapsi tra i neuroni dell'hidden layer: ciascuno di questi neuroni è connesso a tutti gli altri dello stesso layer ed anche a se stesso. Le sinapsi delle varie reti possono essere sia eccitatorie che inibitorie. Questo significa che, alla ricezione di uno spike, il neurone postsinaptico subirà an aumento della corrente nel primo caso, mentre una diminuzione nel secondo. Questo tipo di rete dovrebbe favorire un meccanismo di "memorizzazione", per cui le sinapsi ricorrenti rappresentano il modo in cui i neuroni tengono memoria di certi pattern per migliorare la predizione. Il treno di impulsi nel layer di input è basato sugli eventi ON e OFF: nella rete neurale ogni sensore è rappresentato da due neuroni, uno per ciascun tipo di evento, per un totale di 24. Ogni istante in cui uno degli eventi è stato registrato da un certo sensore x corrisponde a un istante della simulazione in cui il rispettivo neurone di input effettuerà uno spike. L'apprendimento è stato effettuato mediante la backpropagation. Per i motivi discussi nel capitolo 2, è stata utilizzata una *fast sigmoid* come funzione per il gradiente surrogato. La dimensione migliore del timestep insieme ai valori di diversi altri iperparametri è stato uno dei problemi affrontati, ma ai fini di questa tesi questa parte non è di interesse. Per la rete finale, i migliori valori ottenuti per questi parametri, in funzione del threshold θ, verranno tenuti fissi. L'introduzione del timestep ci porta ad un primo problema che riguarda questa simulazione: l'approssimazione dei tempi di spike. Nella simulazione time stepped e per la backpropagation, come abbiamo detto nel capitolo 2, il tempo è discretizzato e si salta da un momento all'altro di un certo Δt. Questo implica che, per adattare il dataset alle esigenze della rete, bisogna modificare il treno di impulsi in ingresso. La logica è la seguente: il neurone in input registra uno spike al tempo $t + \Delta t$ se nel dataset è presente almeno uno spike nel periodo fra t e $t + \Delta t$, altrimenti no. Questo ovviamente per ciascuno dei chunk di tempo dei segnali in input, ovvero $\frac{T_{rec}}{\Delta t}$, dove T_{rec} è l'arco di tempo impiegato dal sensore per raccogliere l'input per ciascuna lettera (nel caso specifico 1.25s). Effettuare questo lavoro significa non solo ritoccare il dataset, ma anche potenzialmente perdere degli spike a causa del fatto che, ad ogni istante temporale, un neurone può effettuare un solo spike.

I neuroni scelti per queste reti sono di tipo CUBA-LIF (per ulteriori dettagli si faccia riferimento al capitolo 2.2.1). La topologia della rete è formata da:

- layer di input: come già specificato, questo layer è formato da 2 neuroni per ogni evento, questo per ogni sensore. Ciascun threshold θ inoltre presenta un numero variabile di copie di questo gruppo di 24 neuroni (e quindi copie del segnale in ingresso alla rete), con sinapsi dai pesi differenti.

- hydden layer: questo è formato da 450 neuroni.

- output layer: inizialmente formato da 28 neuroni: uno per ogni lettera da A a Z, uno per rappresentare il carattere 'spazio' e uno per rappresentare un caso limite per esempio di errata lettura della lettera da parte del sensore. Quest'ultima classe non si è rivelata significativa per l'accuratezza del modello, dunque l'implementazione finale prevede solo i primi 27 neuroni.

Per concludere, in base a quanto specificato nel capitolo 2 ed in particolare nel paragrafo 2.1, possiamo dire che la codifica dell'input di questo modello utilizza il Temporal Contrast Coding (TCC). La predizione della rete è basata sul neurone che avrà generato più spike, quindi la decodifica dell'output è basata su Rate Coding (RC). La simulazione timestepped di riferimento è stata implementata utilizzando le librerie apposite in python, in particolare *torch*. Di seguito sono presentati i risultati migliori per ciascun threshold e rete. Per ulteriori dettagli sulla configurazione con cui la simulazione è stata eseguita si faccia affidamento al paper.

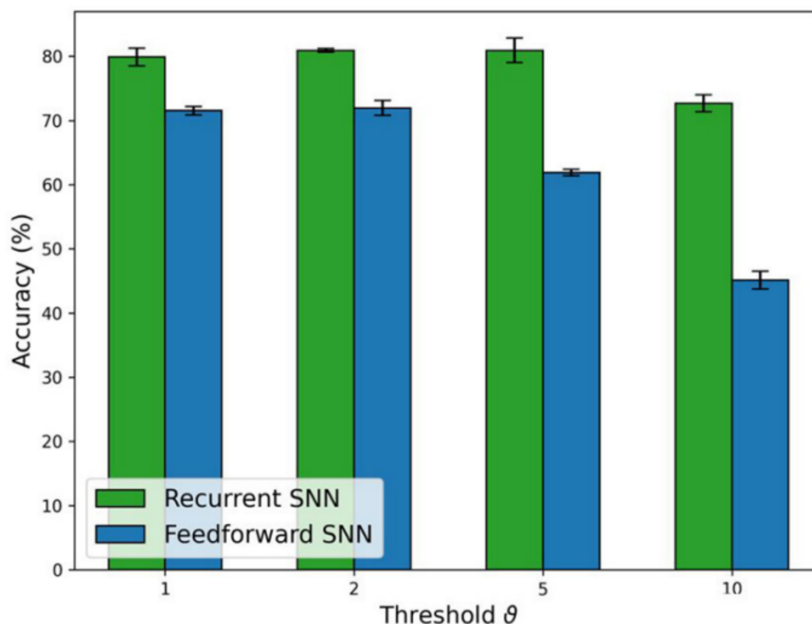


Figura 4.1: Accuracy simulazione time stepped, [26] Figura 7

L'accuratezza cresce di poco tra i primi due threshold, e da quel punto in poi diminuisce con l'aumento della sparsità dell'input.

4.2 Replicazione del modello

Per essere in grado di replicare il modello utilizzato dagli autori del paper è stato necessario un lavoro preliminare di raccolta informazioni. Il paper, infatti, non presentava alcuni parametri fondamentali per comprendere la rete. Quello che abbiamo è (tabella 3, [26]):

- Il numero di copie dei neuroni che trasmettono l'input dei sensori, che permette di definire quanti sono i neuroni dell'input layer per ciascun threshold. Questi nello specifico sono $24 \cdot 4(48)$ per $\theta = 1$ e $\theta = 10$, $24 \cdot 8(192)$ per $\theta = 2$ e $24 \cdot 4(96)$ per $\theta = 5$

- La dimensione del time step, di 3, 5, 5 e 3 millisecondi rispettivamente per i 4 threshold in ordine crescente

- τ_{syn} e τ_{mem}, di 6 e 60, 5 e 50, 7 e 70, 7 e 70 millisecondi rispettivamente per i 4 threshold in ordine crescente

Ciò che manca è il potenziale di reset V_r, il potenziale di threshold V_{th}, il periodo refrattario, il delay sinaptico, i pesi delle singole sinapsi w_{ij}, la durata totale della simulazione. Inoltre, per essere in grado di confrontare gli eventuali risultati di ROOT-Sim con quelli pubblicati nel paper, sarebbe opportuno avere a disposizione le informazioni sul dataset e la sua ripartizione in training set e test set. Per ricavarli, si è fatto riferimento al materiale pubblico pubblicato su github, con i codici sorgenti dello script python per la RSNN e la FFSNN [1]. Per la durata totale della simulazione, analizzando il codice si deduce che vale 1275 ms:

```
1  def load_and_extract(params, file_name, taxels=None,
     letter_written=letters):
2    max_time = int(51*25)  # ms
3    /* ... */
4    time = range(0, max_time, time_bin_size)
5    /* ... */
6    events_array = np.zeros([nchan, round((max_time/time_bin_size)+0.5), 2])
```

Listing 4.1: Definizione del tempo massimo

Infatti dala parte evidenziata in 4.1, si può notare l'utilizzo della variabile *max_time* per la definizione dei timestep che caratterizzano l'intera durata della simulazione, e della struttura che tiene traccia del mapping tra gli spike reali nel dataset e quelli utilizzati nella simulazione. Per quanto riguarda invece i potenziali di threshold e di reset si veda 4.2.

```
1  class feedforward_layer:
2    /* ... */
3    def compute_activity(nb_input, nb_neurons, input_activity, nb_steps):
4        /* ... */
5        for t in range(nb_steps):
6            mthr = mem-1.0
7            out = spike_fn(mthr)
8            rst_out = out.detach()
9            /* ... */
10           new_mem = (beta*mem + syn)*(1.0-rst_out)
11           /* ... */
12
13   /* ... */
14   class SurrGradSpike(torch.autograd.Function):
15   /* ... */
16   def forward(ctx, input):
```

```
17            /* ... */
18            out = torch.zeros_like(input)
19            out[input > 0] = 1.0
20            return out
21        /* ... */
22    spike_fn = SurrGradSpike.apply
```

Listing 4.2: Utilizzo dei potenziali

Stando a quel codice, senza entrare in dettagli troppo specifici sull'implementazione, possiamo dire che nelle righe 6 e 7 si stabilisce se un neurone ha emesso uno spike o meno, e questo in base alla condizione che il potenziale abbia superato il valore 1.0 (che quindi è il valore del threshold). Nella riga 9 viene calcolato il valore del nuovo potenziale che, oltre alle formule presentate nel Capitolo 2, viene moltiplicato per 0 o 1, se il neurone ha oppure non ha effettuato uno spike. Questo implica che 0 è il lower bound per il valore del potenziale, e quindi anche il potenziale di reset, ed inoltre lo spike produce un *reset-to-zero*.

Per quanto riguarda il periodo refrattario, dal codice non è stata ricavata un'indizaione certa. L'ultimo pezzo nel codice 4.2 tuttavia, nel calcolo del potenziale ad ogni timestep, non tiene conto di eventuali spike appena avvenuti ma indipendentmente dalla storia del valore del potenziale, il suo valore cambia in base alla stessa formula. Questo farebbe ipotizzare che non ci sia un vero e proprio periodo refrattario ben codificato, a parte quello artificiale indotto dalla presenza di timestep. Ricordiamo infatti che se in una simulazione precisa un neurone genera uno spike non appena il potenziale supera il threshold, in una timestepped il valore del potenziale viene valutato solo ad un certo timestep. Questo implica che, su timestep di (per esempio) 5ms, per un valore di potenziale che supera il threshold al tempo 1ms (e quindi dopo il timestep a 0ms e prima di quello a 5ms) il suo spike verrà registrato e generato al secondo timestep, al tempo 5ms, momento in cui il potenziale tornerà al valore di reset. Questo, replicato su una simulazione precisa, può essere visto come un caso in cui il periodo refrattario vale 4ms. Chiaramente è un modo molto approssimativo di modellare la situazione. Per testare l'ipotesi del periodo

refrattario nullo, è stato realizzato un semplice script python per simulare il comportamento di un solo neurone replicando ciò che avviene nello script degli autori del paper.

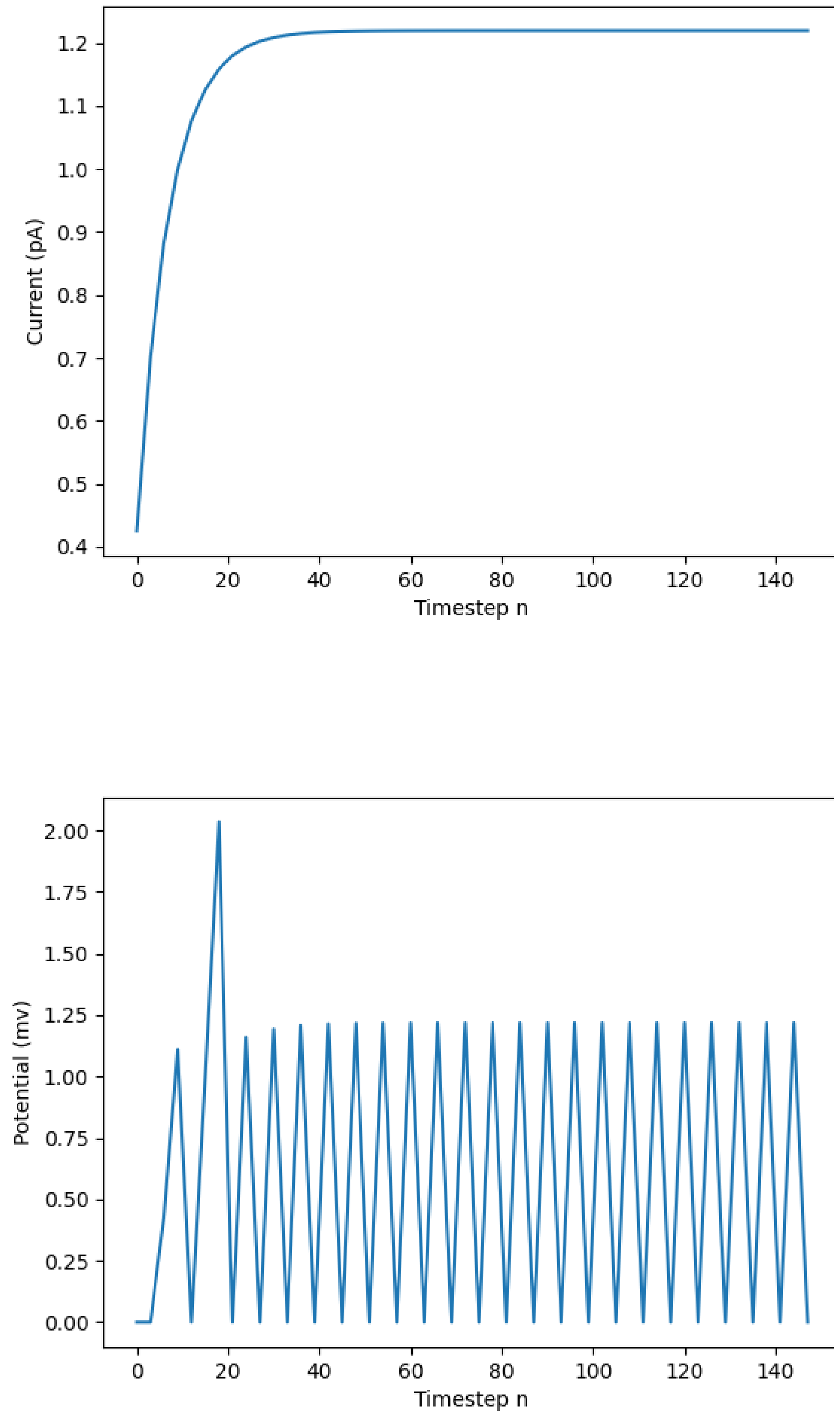


Figura 4.2: Corrente e potenziale dei primi 150 time steps

Come si può vedere nei grafici in Figura 4.2, nella quasi totalità dei casi non esistono valori consecutivi di potenziale al valore di reset, indice del fatto che il periodo refrattario, in questa simulazione, vale 0. Per quanto riguarda il delay sinaptico, valgono le stesse conclusioni tratte per il periodo refrattario: può essere considerato nullo, anche se l'esistenza stessa dei timestep induce un delay artificiale. Il prossimo passo è ricavare gli elementi che compongono training set e testing set. All'interno dello script a disposizione lo splitting è ovviamente casuale, basato sul seed di partenza. Il seed del generatore pseudo-casuale utilizzato è uno dei problemi riscontrati in questo tentativo di replicare i risultati del paper, ma su questo argomento si tornerà in dettaglio al termine di questo paragrafo.

```
1    if use_seed:
2      seed = 42  # "Answer to the Ultimate Question of Life, the Universe, and
       Everything"
3      os.environ['PYTHONHASHSEED'] = str(seed)
4      np.random.seed(seed)
5      torch.manual_seed(seed)
6      print("Seed set to {}".format(seed))
7  else:
8      print("Shuffle data randomly")
```

Listing 4.3: Seed per la generazione dei valori

Le righe di codice in 4.3 [2, 3] lasciano intuire il valore del seed all'interno del codice pari a 42 (se questo viene utilizzato). Per ricavare gli elementi degli splitting si è ricorso a una leggera modifica del loro script che vada a scrivere in un file esterno l'indice di ciascun elemento appartenente ai due set, file da utilizzare poi dal modello implementato con ROOT-Sim. Per quanto riguarda i pesi delle sinapsi, lo stesso stratagemma è stato utilizzato. Questi, nello script, vengono inizializzati a un valore campionato da una distribuzione normale (quindi anche qui il valore del seme utilizzato è rilevante) con media $\mu = 0$ e una deviazione standard σ molto bassa, dipendente dalla dimensione del layer e dal tipo di connessione (se ricorrente o feedforward). Questi pesi vengono allenati per 300 epoche. Il set di pesi migliori sarà quello che si è comportato meglio sul testing set indipendentemente dal training set, questo

ovviamente per prevenire situazioni di overfitting. Lo stesso procedimento è ripetuto cinque volte per ottenere cinque set di pesi differenti ciascuno con la propria accuratezza migliore. Per gli scopi del lavoro, sono stati esportati in un file esterno solo i pesi migliori di ciascuna delle cinque iterazioni.

Ottenute queste informazioni, prima della conclusione del capitolo c'è da fare una doverosa precisazione. Ci sono diversi punti poco chiari e discrepanze tra quanto presentato nel paper e tra i risultati ottenuti dalla replicazione degli script. In primo luogo, il grafico dei valori delle accuratezze ottenute dalla simulazione effettuata per questa tesi non corrisponde a quello presente nel paper (Figura 4.1)

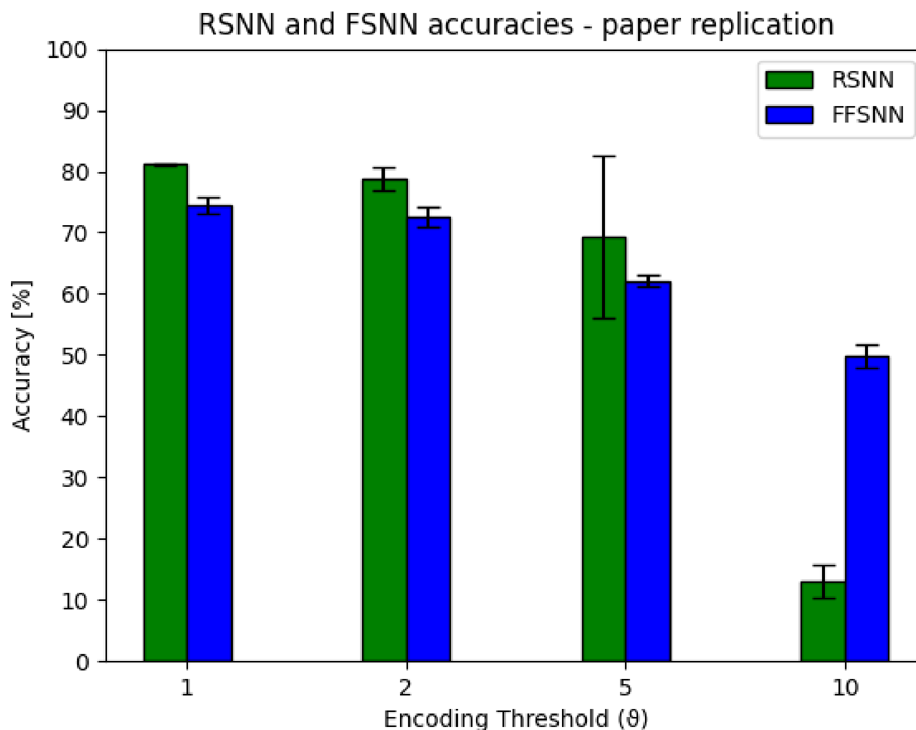


Figura 4.3: Risultati della replicazione degli esperimenti del paper

Ci sono diverse possibilità sul perché ciò avviene. Una potrebbe essere legata alle differenze tra le macchine su cui la simulazione è stata eseguita. A livello di libreria python utilizzata non è stato trovato nulla che giustificasse tale comportamento, tuttavia potrebbero esserci delle differenze più sottili che non sono state identificate. Una seconda possibilità riguarda il seed utilizzato

per la generazione di numeri randomici. Questo influenza, in maniera evidente, sia l'evoluzione della rete (che partirà da valori dei pesi differenti), sia la misura di accuratezza dato il diverso splitting che verrebbe eseguito. Come accennato in precedenza, sul paper non c'è alcun riferimento al seed utilizzato, quindi tutte le informazioni a disposizione si trovano all'interno del codice. Le esecuzioni lì presenti ([2, 3]) non sembrano utilizzare il seed, e se questo non è stato utilizzato per produrre i risultati del paper, allora la rete non sarebbe perfettamente riproducibile. Questo tuttavia non spiega il perché, come si può notare, nel caso di RSNN con $\theta = 10$ la rete ha un comportamento anomalo. Per tutte e 5 le iterazioni, infatti, la sua accuracy dopo poche epoche si stabilizza intorno al 15%. Per quanto sembri poco probabile, potrebbe comunque essere causato dai pesi di partenza, e quindi dal seed utilizzato, che portano a problemi di convergenza. Altri problemi riguardano gli script in sè, che presentano bug di lieve entità che hanno inizialmente impedito la loro esecuzione ma che sono stati ovviamente risolti. Un possibile bug, su cui si è sorvolato, riguarda i training con le opzioni di allenamento sui parametri temporali. In quel caso, si arrivava già in poche epoche in situazioni in cui la rete non poteva essere più allenata perché i pesi assumevano valori NaN. Questa, probabilmente, è una situazione generata da qualche operazione aritmetica problematica, tuttavia non è stato possibile stabilire quale fosse l'origine. L'ultimo significativo problema riguarda la suddivisione del dataset. Se nel paper viene menzionato uno splitting 80/20, all'interno degli script non solo lo splitting ha proporzioni differenti, ma varia tra l'FFSNN e l'RSNN. Per aggiungere ulteriori problemi, non solo varia di proporzioni, ma anche di identità. Lo script dell'RSNN [3], infatti, presenta anche un validation set, che non viene in alcun modo menzionato nel paper. Nel caso della FFSNN [2] abbiamo uno split 90/10, nel caso della RSNN abbiamo uno split 70/20/10, come evidenziato nei codici 4.4 e 4.5.

```
1  /* ... */
2      # create 70/20/10 train/test/validation split
3      # first create 70/30 train/(test + validation)
4      x_train, x_test, y_train, y_test = train_test_split(
5          data, labels, test_size=0.30, shuffle=True, stratify=labels)
6      # split test and validation 2/1
```

```
7       x_test , x_validation , y_test , y_validation = train_test_split(
8           x_test , y_test , test_size=0.33 , shuffle=True , stratify=y_test)
9   /* ... */
```

Listing 4.4: Codice splitting RSNN

```
1   /* ... */
2       x_train , x_test , y_train , y_test = train_test_split(data , labels ,
        test_size=0.10 , shuffle=True , stratify=labels)
3   /* ... */
```

Listing 4.5: Codice splitting FSNN

Il paper dice esplicitamente che i valori di accuracy di riferimento riguardano il testing set; perciò, nel seguito del lavoro, si farà riferimento solo a training set e testing set, tralasciando il validation set e ignorando il fatto che i set abbiano dimensioni diverse.

4.3 Il modello con ROOT-Sim

Il modello è stato implementato in c utilizzando ROOT-Sim. Il progetto, allo stato iniziale, consiste in due file in c e due header. I file **model_topology.c** e **model_topology.h** definiscono le strutture dati con le informazioni che riguardano le reti in base al threshold e la conseguente configurazione di ROOT-Sim come mostrato nel codice a 4.6.

```
1   struct neuron_params_t neuron_params[4] ={
2       //valid for spikes with encoding threshold =1
3       {
4           .threshold = 1.0,
5           .reset_potential = 0.0,
6           .inv_tau_mem = 1 / 60.0,
7           .inv_tau_syn = 1 / 6.0,
8           .inv_C_m = 1/1.0,
9           .refractory_period = 0.0,
10          .synaptic_delay = 5.0
11      },
12      /* ... */
13      #define TERMINATION_TIME 1275
14      struct neurome_configuration conf = {
15      .lps = 100,
16      .termination_time = TERMINATION_TIME,
17      .gvt_period = 1000,
18      /* ... */
19  };
```

Listing 4.6: struttura neuron_params_t e parametri di configurazione

Ovviamente il numero totale di neuroni, e quindi gli lps, dipendono dal threshold della rete simulata, quindi verranno aggiustati di conseguenza. Inoltre vengono esportate altre variabili che riguardano la dimensione dei vari layer (input, output, hidden). Per utilizzare ROOT-Sim, si può scrivere un programma in linguaggio c con il main come entrypoint, in cui vengono chiamati, in questo ordine, **NeuromeInit**, con il puntatore alla struttura neurome configuration utilizzata dalla simulazione, e **NeuromeRun** per avviare la simulazione. Per la simulazione il programma permette di definire il valore di determinate variabili, mediante opzioni in input o mediante un file di configurazione apposito. Tra le variabili ci sono la modalità (FFSNN o RSNN), threshold θ, set di pesi da utilizzare tra i 5 disponibili e set di dati da utilizzare (training set, testing set, validation set nel caso di RSNN) e quale elemento del rispettivo set simulare. Questi parametri sono obbligatori. In base alle impostazioni verranno ottenuti i corretti pesi sinaptici, lo spike train di quella specifica simulazione, i parametri di configurazione come il numero di lps. Per quanto riguarda l'output, è prevista la stampa (opzionale) del numero di spike di ciascun neurone di output, corrispondente a una lettera, e l'aggiornamento di uno specifico file che tiene traccia dell'accuratezza globale dopo ciascuna esecuzione. Per quanto riguarda le funzioni implementate:

4.3.1 NeuronInit

Per prima cosa, la funzione inizializzerà lo stato del neurone chiamando la funzione **InitLIFNeuron**, che non farà altro che restituire un puntatore a un'istanza privata di *neurone_state_t*, struttura dati che contiene i dati del neurone da utilizzare durante la simulazione come mostrato nel codice in 4.7.

La struttura *neuron_helper_t* è stata introdotta per mantenere i dati condivisi da un'intera popolazione di neuroni in un'unica area di memoria. In questo caso tutti i neuroni di tutte le reti condividono queste stesse informazioni, ovvero: i parametri moltiplicativi qui chiamati A0 e A2 (si faccia riferimento a 3.3), l'assenza di corrente esterna e Icond che rappresenta una condizione per valutare il tempo di spike futuro del neurone [36] (aspetto approfondito

```
typedef struct neuron_state_t{
  struct neuron_helper_t *helper;
  unsigned long int times_fired;
  double membrane_potential; // [mV]
  double I;     // [pA]
  simtime_t last_fired; // For refractory period
  simtime_t last_updated;
  spikes_array spikes_record;
} neuron_state_t;
struct neuron_helper_t {
  double Iext;
  double A0;
  double A2;
  double Icond;
  double self_spike_time; // This is the time at which the neuron self spikes
};
```

Listing 4.7: struct per lo stato dei neuroni

più tardi). Gli altri parametri, che sono specifici per ciascun neurone, sono il valore di corrente e di potenziale a un determinato istante di tempo t, questo tempo t in cui l'ultimo aggiornamento è avvenuto, l'ultimo istante in cui il neurone ha emesso uno spike e il numero di spikes effettuati fino al tempo t. La struttura *spikes_array* verrà approfondita in seguito.

Dopo aver inizializzato il suo stato con i valori azzerati, il neurone viene connesso agli eventuali neuroni postsinaptici. L'identificazione dei neuroni avviene con un valore *unsigned long* che va da 0 a numero di lps-1. Per la connessione, si è adottato un semplice schema in cui in neuroni con identificativo $0 <= x < input_layer_size$ appartengono all'input layer (la cui dimensione ricordiamo essere variabile in base al threshold θ). Questi creeranno la sinapsi verso tutti i neuroni dell'hidden layer tramite la funzione **Connect**. Questi neuroni inoltre devono anche chiamare la funzione di Spike affinché utilizzino lo spiketrain del dataset che si sta simulando. Quelli con l'identificativo $input_layer_size <= x < (input_layer_size + hidden_layer_size)$ appartengono all'hidden layer: questi creeranno la sinapsi verso i neuroni dell'output layer e, se la rete è ricorrente, creeranno anche le sinapsi ricorrenti verso i neuroni dell'hidden layer stessi. I restanti fanno parte dell'output layer, e non avendo connessioni con nessuno, nessun'altra azione è richiesta.

4.3.2 NeuronHandleSpike e SynapseHandleSpike

Per quanto riguarda la *SynapseHandleSpike*, le sinapsi di questo modello sono semplici e il loro valore rimane costante a runtime, dunque l'implementazione della funzione ritornerà il valore corrente del peso della sinapsi senza fare altro. La *NeuronHandleSpike*, invece, come prima cosa aggiornerà lo stato del neurone chiamando la funzione **bring_to_present**. Quest'ultima, per aggiornare i valori di potenziale e corrente, e quindi applicare le equazioni 3.1 e 3.2, controlla il timestamp dell'ultimo aggiornamento e dell'ultimo spike. Se l'ultimo aggiornamento è avvenuto durante il periodo refrattario e se questo non è ancora terminato, significa che il potenziale è bloccato a V_r, quindi si aggiorna solamente il valore della corrente e il parametro *last_updated.* In caso contrario, si calcola il valore di corrente su tutto l'arco temporale dall'ultimo aggiornamento al momento corrente di simulazione, mentre il valore del potenziale solamente dal termine del periodo refrattario. Terminata l'esecuzione di bring_to_present, si chiamerà la funzione **getNextFireTime** per determinare, stando al valore attuale dei parametri, qual è il prossimo istante temporale in cui, in assenza di altri input, il neurone effettuerà uno spike: in quel momento verrà schedulato un evento di MAYBESPIKEANDWAKE. La funzione getNextFireTime parte da un paio di assunzioni semplici: in primo luogo, un neurone (che possiamo chiamare self-spiking) può raggiungere la condizione di spike anche senza ricevere nessuno spike in input, se:

$$\lim_{t \to \infty} V(t) = A_2 > V_{th} \tag{4.1}$$

Se non è self-spiking, ci sono delle condizioni apposite per cui si può avere la certezza che, in un certo intervallo di tempo nel futuro, date le condizioni attuali, il neurone effettuerà uno spike. Per i dettagli si faccia riferimento a [36]. Questo permette di utilizzare il metodo della bisezione per trovare l'istante temporale, con una certa tolleranza (che per questo lavoro è stata scelta di 0.01ms), in cui avverrà lo spike.

4.3.3 NeuronWake

Questa funzione gestisce i neuroni nel momento del loro spike. Quello che deve fare è dunque aggiornare le variabili di stato del neurone, aggiungendo uno al contatore di spike *times_fired*, aggiornando il valore della corrente e resettando il valore del potenziale a V_r. Successivamente anche qui verrà chiamata la funzione *getNextFireTime* per schedulare, eventualmente, un evento MAYBESPIKEANDWAKE in funzione del valore della corrente e del fatto che il neurone potrebbe essere self-spiking.

4.3.4 NeuronEndAlign

L'ultimo problema da affrontare è il fatto che la memoria rollbackabile gestita autonomamente da ROOT-Sim viene deallocata al termine della simulazione, dunque per tenere traccia del numero di spike effettuati dai neuroni dell'output layer, necessari per la predizione, non è sufficiente mantenere un riferimento globale alle strutture dati utilizzate dal simulatore, a cui accedere al termine della simulazione. Per sopperire a questo problema si è aggiunto un nuovo tipo di evento, denominato **NEURONENDALIGN**, da schedulare al termine dell'esecuzione. Il suo scopo è quello di effettuare una copia di tutte le informazioni utilizzate nel contesto della simulazione a cui si vuole accedere successivamente, senza incorrere in problemi di memoria. Nel caso specifico, si tiene traccia del contatore di spike per ogni neurone dell'output layer in un vettore accessibile successivamente.

4.4 Training rete precisa

Per quanto riguarda il trainig, come già spiegato la backpropagation non è una strada percorribile per questo tipo di simulazione e quindi si è fatto ricorso a un'implementazione di STDP specifica per il problema in questione. Le simulazioni sono state gestite con un programma c apposito che, rispetto al precedente Makefile, ha permesso una più facile gestione dei parametri di

esecuzione e del batching delle prove. Utilizzare il batching, e la sua eventuale dimensione, è un fattore del tutto facoltativo; tuttavia, l'intuito farebbe propendere verso una soluzione che aggiorna i pesi ad ogni esecuzione.

4.4.1 Implementazione

Per implementarlo, si è aggiunto un file c e un header file al progetto. Per prima cosa, sono state implementate delle strutture dati che contenessero le informazioni della topologia di rete (puntatori ai pesi correnti delle sinapsi), poi le informazioni correnti della simulazione appena eseguita, ovvero le tracce di tutti i tempi di spike relativi a ciascun neurone della rete ed infine i parametri di configurazione per l'esecuzione di STDP. Anche per questi è stato previsto l'utilizzo di un file di configurazione apposito. I parametri principali sono

```
1  typedef struct stdp_params {
2    double A_plus;
3    double A_minus;
4    double tau_plus_inv;
5    double tau_minus_inv;
6    //eta = 0 to use prefixed A
7    double eta_plus;
8    double eta_minus;
9    bool soft_bounds; //if false, hard bound will be applied. if eta=0, will be
       used the pre-chosen A
10   double w_max;
11   double w_min;
12         #altri parametri
```

Listing 4.8: parametri per l'applicazione di stdp

mostrati nel codice in 4.8. Per comprendere il significato di questi parametri si faccia riferimento al paragrafo 3.4. Sono stati memorizzati gli inversi dei due τ_+ e τ_- per lo stesso motivo per cui sono stati memorizzati gli inversi dei τ_{syn} e τ_{mem} nella struttura con i dati del neurone: queste variabili appaiono solamente al denominatore dell'esponente della funzione STDP, quindi per ragioni di efficienza, anziché una divisione viene effettuata una moltiplicazione per l'inverso. Gli altri parametri verranno discussi in seguito. I tempi di spike vengono mantenuti in array di puntatori, uno per ogni layer. Questi contengono un puntatore, per ogni neurone, a strutture *spikes_array*, mostrate nel codice in (4.9).

```
1      typedef struct spikes_array{
2    simtime_t *times;
3    size_t size;
4    size_t capacity;
5      } spikes_array;
```

Listing 4.9: array dinamico per memorizzare gli spike

Di fatto questo spikes_array non è altro che un array dinamico, con le relative funzioni che permettono di inizializzarlo e inserire elementi. Per ottenere questo, in primo luogo un'istanza di questa struttura è stata inserita nello stato del neurone. Questo array verrà inizializzato nell'ambito della funzione *NeuronInit*, e verrà aggiornato ad ogni esecuzione della funzione *NeuronWake*. Ricordiamo però che gli spike non verranno necessariamente committati ma questo dipenderà dall'andamento dell'esecuzione della simulazione, e quindi l'unica accortezza è stata quella di inizializzare e aggiornare anche l'array **times* nella struttura utilizzando le funzioni *rs_alloc* che, come già detto in precedenza, è una soluzione offerta da ROOT-Sim per gestire la memoria dinamica soggetta a rollback senza che sia il modellista a doversene occupare. Per quanto riguarda l'utilizzo di questi dati per l'applicazione di STDP, ricordiamo che al termine della simulazione con ROOT-Sim la memoria utilizzata verrà deallocata, quindi è necessario anche aggiornare la funzione *NeuronEndAlign* e copiare il contenuto di questa struttura dati.

4.4.2 Utilizzo

Per ottenere l'aggiornamento dei pesi bisogna chiamare la funzione *weightUpdate*, dando in input una struttura *topology_params* che contiene appunto i parametri della topologia e un puntatore ai parametri della simulazione. Questa funzione restituirà una struttura *weights_delta*, che contiene i puntatori a matrici (tre o due, in base al tipo di rete utilizzata) contenenti i delta di peso da applicare a ciascuna sinapsi della rete. Il calcolo di STDP è stato gestito come segue: avviando il programma normalmente con il parametro *-l*, oltre alla simulazione, verrà chiamata la funzione weightUpdate. Per fare ciò la struttura *topology_params* è stata inizializzata in una prima fase, andando

a ricavare i dati dal file di configurazione, e successivamente aggiornata in base ai risultati della simulazione. Una volta ottenuti i delta, questi vengono scritti su file utilizzando la funzione *weightsFileUpdate* con in input il file destinazione, il puntatore alla matrice dei delta da applicare sul peso delle sinapsi e le sue dimensioni. Questa funzione non farà altro che aggiornare i valori presenti nel file, se questo già esiste (e quindi una precedente simulazione con learning è stata effettuata, simulazione che ha prodotto a sua volta dei delta memorizzati su questi file), altrimenti lo crea e inserisce i valori dei delta appena calcolati. Il mapping da utilizzare per interpretare questo file è lo stesso utilizzato per i file con i pesi originali: ciascuna riga n-esima, dove n è un determinato neurone presinaptico, contiene m pesi, uno per ciascun neurone postsinaptico. Ciascuno di questi valori è separato da un carattere di spazio. Per applicare le modifiche ai pesi, è previsto l'avvio del programma con parametro *-u*, opzione che porta ad aggiornare i pesi (basandosi sui parametri nel file di configurazione) e cancellare i file contenenti i delta applicati.

5. Risultati Sperimentali

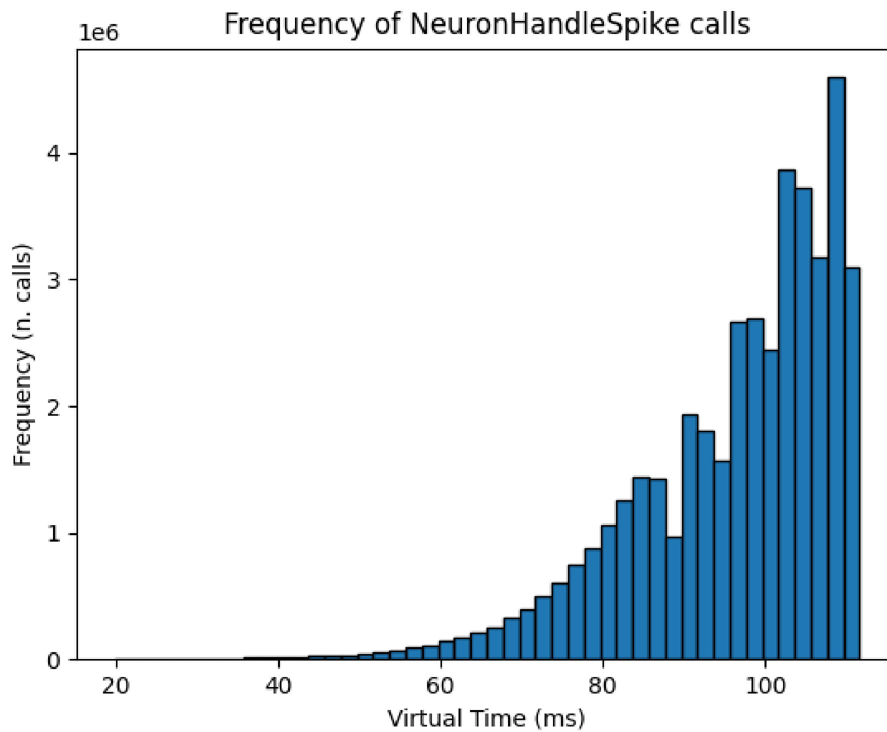
5.1 Accuratezza della rete

Per quanto riguarda la valutazione dell'accuratezza con ROOT-Sim, sono stati eseguiti diversi batch di esecuzioni, variando i parametri in maniera empirica, cercando la soluzione migliore sul set di pesi che, stando alla riproduzione dei risultati del paper, è il migliore. I parametri presi in esame non sono gli stessi per RSNN e FFSNN: a causa delle peculiarità delle due reti, che saranno più chiare in seguito, sono state percorse strade diverse. Una volta trovato il migliore insieme di parametri, la validazione è stata effettuata simulando la rete per tutti i threshold θ e per tutti i 5 set di pesi ottenuti dalla simulazione timestepped. Un'aspetto rilevante da tenere in considerazione prima di trarre le conclusioni è che i pesi utilizzati provengono da una simulazione e una tipologia di training diverse da quelle adottate su ROOTSim. Dato il problema riscontrato con la replicazione della rete con threshold $\theta = 10$, nel caso della rete RSNN questa non è stata simulata. Le simulazioni sono state gestite utilizzando un mekefile personalizzato.

5.1.1 RSNN

Per quanto riguarda questa rete, fin dalle prime prove ci sono stati dei problemi. Avere un periodo refrattario pari a 0 porta ad avere dei feedback loop a livello dell'hidden layer impossibili da gestire che portano al crash del simulatore (dopo una quantità notevole di tempo reale di simulazione, che però corrisponde a davvero poco in termini di tempo virtuale), probabilmente

a causa della saturazione della ram. A riprova di ciò, nei grafici in Figura 5.1 verranno mostrate le chiamate alle funzioni *NeuronHandleSpike* e Neuron-Handle. Chiaramente va ricordato che questi valori sono tutti speculativi: una percentuale molto significativa degli effetti degli eventi schedulati durante la chiamata di queste funzioni, nonché l'effetto delle funzioni stesse, non verrà committato.



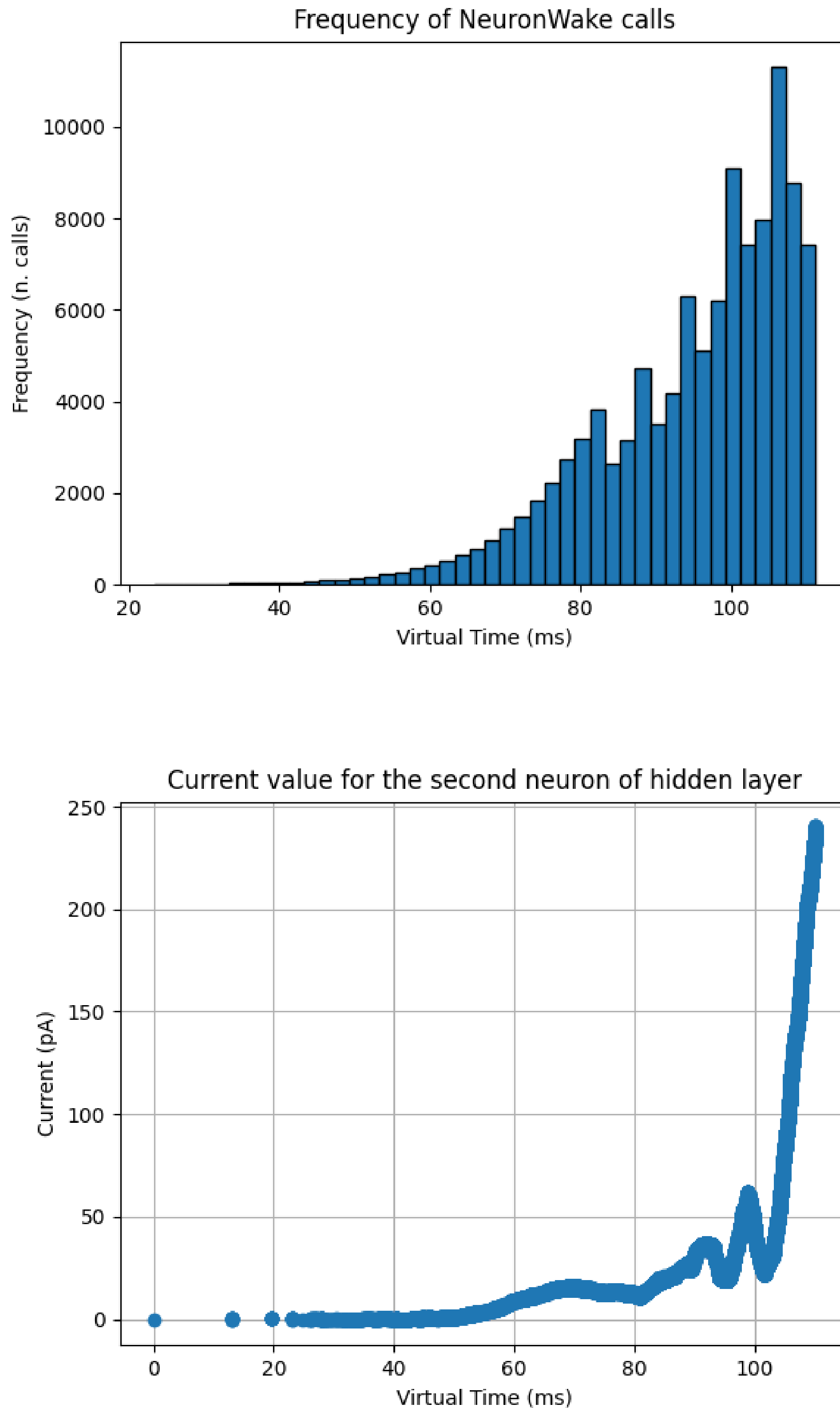


Figura 5.1: I risultati sono stati ottenuti dall'esecuzione della rete, su input di threshold $\theta = 2$, sul primo elemento del validation set e con il miglior set di pesi ottenuti dalla replicazione dello script del paper di riferimento

La frequenza delle chiamate alle due funzioni è crescente nel tempo, e questo, come detto prima, è dovuto al feedback loop formato per ciò che si può vedere nel terzo grafico: senza un periodo refrattario alcuni dei neuroni che iniziano a spikare presto producono una reazione a catena di spike, alimentata eventualmente ancora di più dalla presenza della connessione ricorrente tra il neurone e se stesso, portando la corrente a crescere in maniera esponenziale (o per altri neuroni a diminuire allo stesso modo). Con una corrente sempre più alta, il neurone spikerà sempre più velocemente dopo ogni spike, portando a una frequenza sempre maggiore di spike. Nonostante l'ipotesi iniziale che la connessione ricorrente verso se stessi fosse la causa principale di questo effetto, successive prove empiriche effettuate rimuovendo questa sinapsi hanno portato alle stesse conclusioni. Siccome avere o non avere questa sinapsi, come verrà mostrato successivamente, non sembra avere impatti significativi sull'accuratezza, per mantenere il più possibile la fedeltà alla rete originale si è deciso non non tagliarla. Per risolvere questo problema sono state intraprese due strade distinte. Da un lato, si è cercato di smussare le differenze tra questo metodo di simulazione e il metodo timestepped. Come abbiamo menzionato in precedenza, pur non avendo delay espliciti nella simulazione timestepped, il timestep stesso fa da limite. Questo introduce un delay artificiale, sia per il periodo refrattario che per la propagazione dello spike. Il periodo refrattario e il delay sinaptico di uno spike sono supportati dal modello fino a questo punto: per utilizzarli il codice è stato aggiornato per contenere questi due valori come parametri in input, o come opzioni nel file di configurazione. Variare il delay sinaptico non è stato ritenuto importante (e, come mostrato successivamente, variarlo non ha impatti significativi sulle performance delle reti), quindi dato che un delay di valore zero non sarebbe realistico a livello biologico, si è deciso di utilizzare un delay standard di lunghezza pari alla dimensione del timestep della stessa rete timestepped. Oltre a ciò, per aggiungere un delay artificiale anche in assenza di periodo refrattario è necessaria una modifica alle funzioni *NeuronWake* e *NeuronHandleSpike*: anziché schedulare a priori l'evento MAYBESPIKEANDWAKE all'istante in cui il neurone dovrebbe effettuare il

prossimo spike, si controlla prima che questo tempo rispetti l'intervallo minimo previsto tra uno spike e l'altro. Se questo intervallo viene rispettato l'evento viene schedulato di conseguenza, altrimenti viene schedulato a un istante posticipato, affinché l'intervallo venga rispettato. Questo stratagemma rappresenta comunque una forzatura per questo tipo di simulazione, che sarebbe opportuno evitare. Nelle Figure 5.2 e 5.3 sono riportati i grafici con i risultati di diverse esecuzioni al variare di uno tra periodo refrattario e delay artificale di spike, sul testing set:

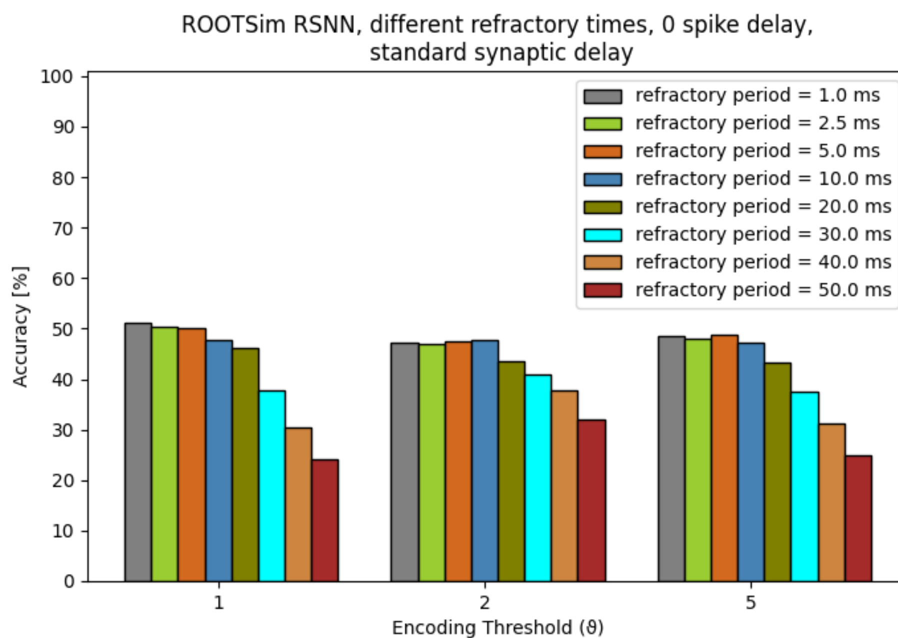


Figura 5.2: Simulazioni senza delay artificiale e periodo refrattario variabile

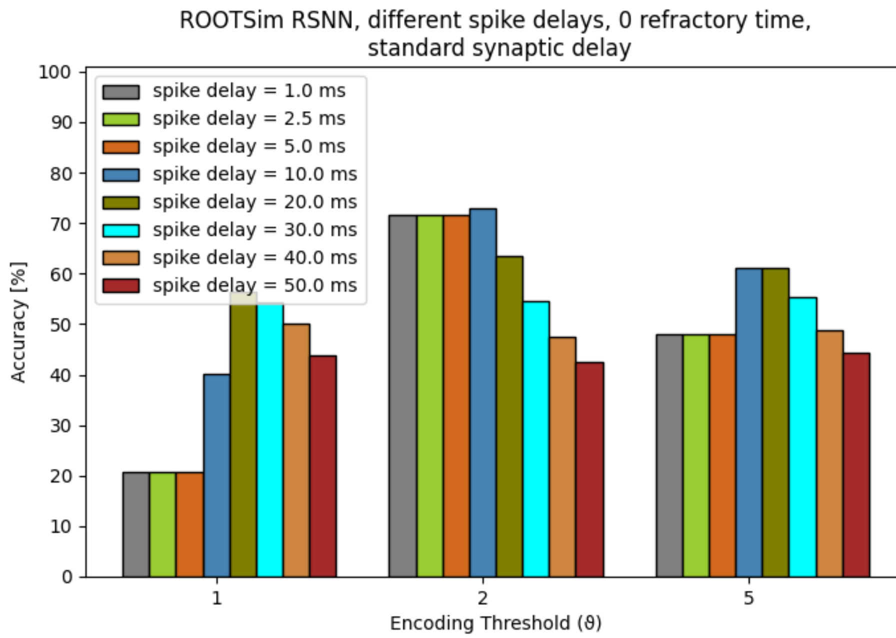
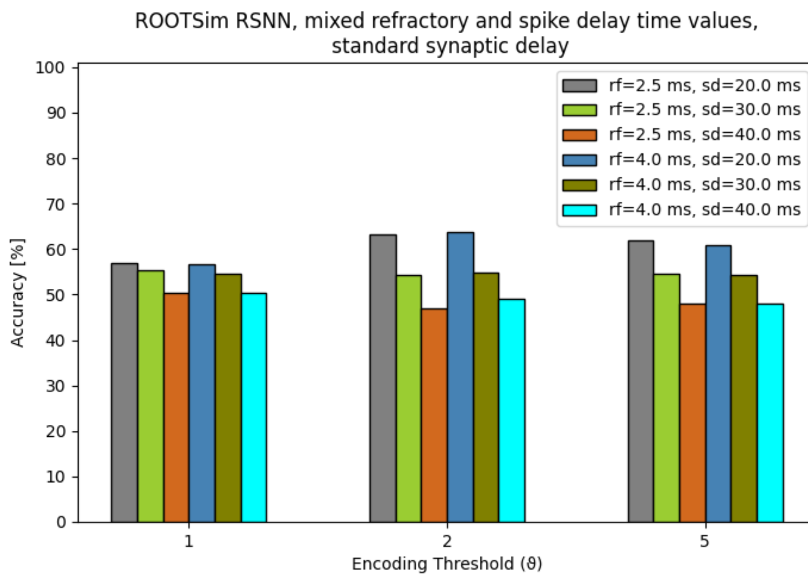


Figura 5.3: Simulazioni senza periodo refrattario e con delay artificiale variabile

I risultati mediamente migliori sembrano essere ottenuti con un delay artificiale dal valore di 20ms.

Le prossime simualzioni hanno preso in considerazione l'utilizzo congiunto di perido refrattario e delay:



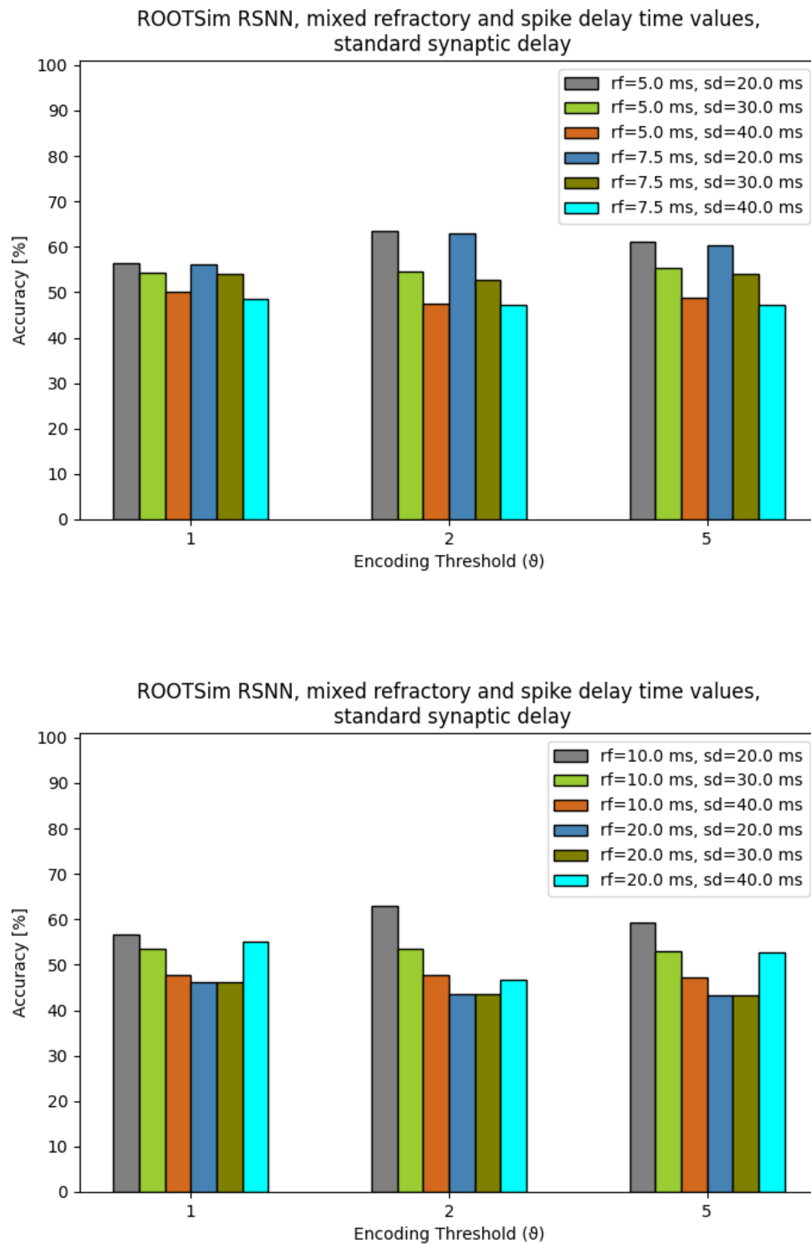


Figura 5.4: Simulazioni con valori di periodo refrattario e delay artificiale misti, rf= periodo refrattario, sd: spike delay artificiale

Stando ai grafici in Figura 5.4 i parametri mediamente migliori sembrano essere 10ms di periodo refrattario e 20ms di delay artificiale. In Figura 5.5 vengono mostrati i risultati della simulazione su tutti i 5 set di pesi a disposizione usando questi parametri:

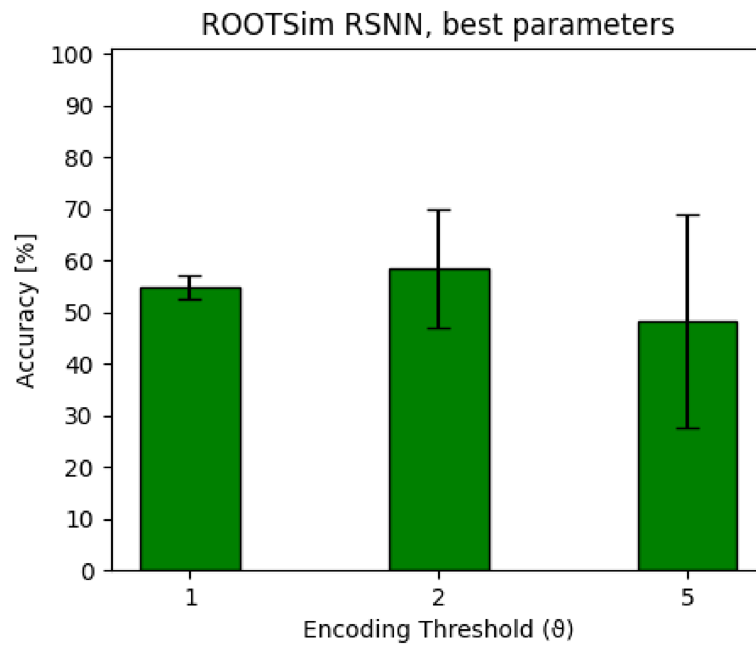


Figura 5.5: Simulazioni coi migliori parametri su tutti i set di pesi

Prendendo in considerazione questi stessi parametri si è valutato anche l'impatto di un valore di delay sinaptico diverso, in particolare di 0.1 ms, e della presenza di sinapsi ricorrenti verso lo stesso neurone.

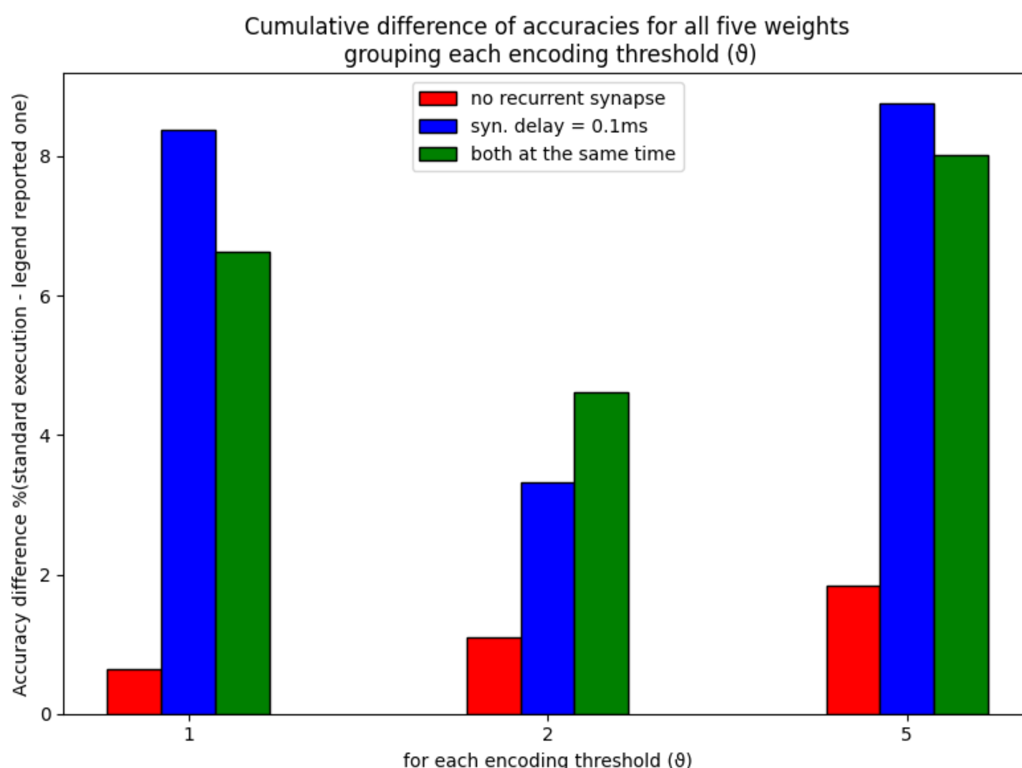
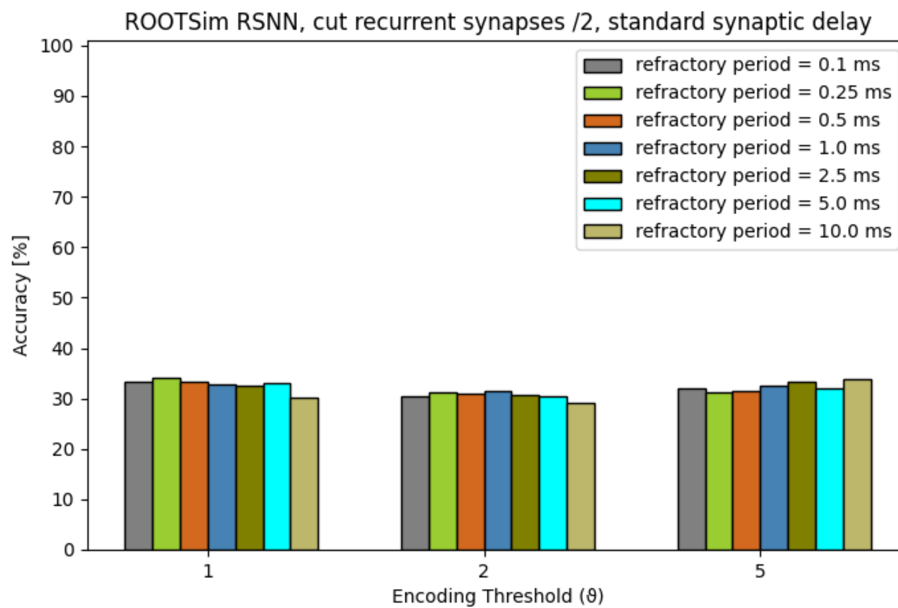


Figura 5.6: Differenza cumulativa in accuracy, raggruppando per encoding threshold, sulle 5 diverse simulazioni

Come si può vedere in Figura 5.6 l'impatto è basso, e non solo: le simulazioni effettuate in precedenza hanno sempre un'accuracy migliore, indipendentemente dal threshold e dalle tre differenti configurazioni con cui la confrontiamo. Anche se come abbiamo detto le valutazioni sono state effettuate sul testing set, sono state effettuate delle esecuzioni anche sul validation set. Le conclusioni sono le medesime, con valori di accuratezza e andamento al variare dei parametri confrontabili, anche se i valori di periodo refrattario e delay artificiale che massimizzano l'accuratezza sembrano essere differenti.

Per quanto riguarda questa strada per la simulazione della RSNN, i risultati non sono paragonabili a quelli del paper. Tuttavia non è un problema: questa soluzione non è preferibile dato che i risultati migliori sono ottenuti aumentando i tempi del periodo refrattario a un valore alto, oltre i valori medi utilizzati in letteratura, e soprattutto aggiungendo un delay ulteriore e artificiale che non vorremmo.

Un'altra possibile strada è frutto dell'osservazione che il problema riscontrato nasce a causa di tutte le sinapsi ricorrenti dell'hidden layer, e non solo di quella verso se stessi. Per arginare la crescita esponenziale della corrente, quindi, si è deciso di effettuare delle prove andando a scalare i pesi delle sinapsi in questione. Fare questo non ha richiesto modifiche al codice del modello, ma solamente la modifica dei pesi utilizzati. Nei grafici in Figura 5.7 sono riportati i risultati di varie prove con vari fattori di scala:



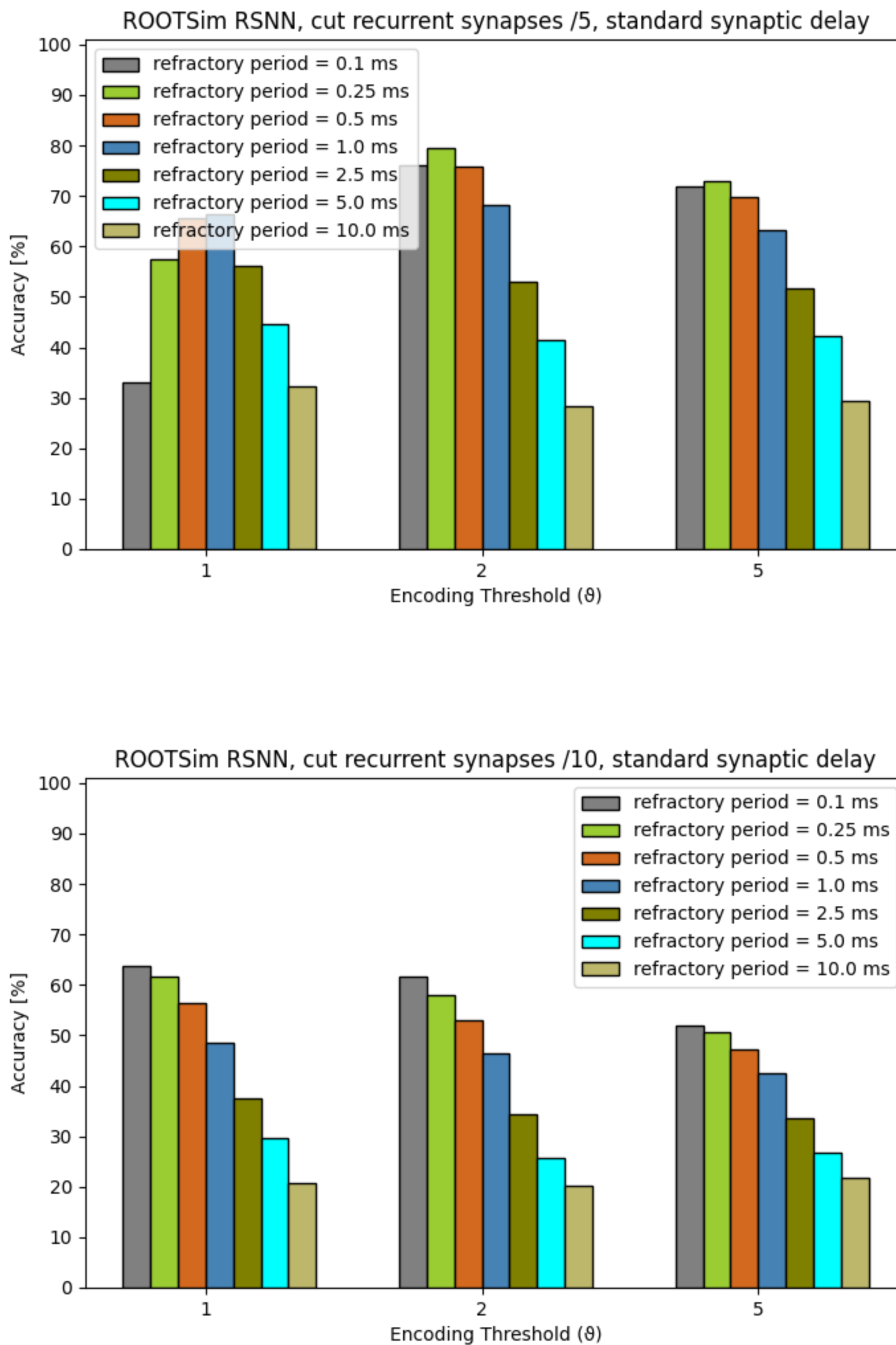


Figura 5.7: Simulazioni con pesi ricorrenti scalati di vari fattori e periodo refrattario variabile

La soluzione migliore sembra essere quella con i pesi delle sinapsi ricorrenti originali scalati di un fattore 5 e con periodo refrattario di 0.25ms. I risultati della validazione su tutti e cinque i set di pesi con questi parametri sono mostrati in Figura 5.8.

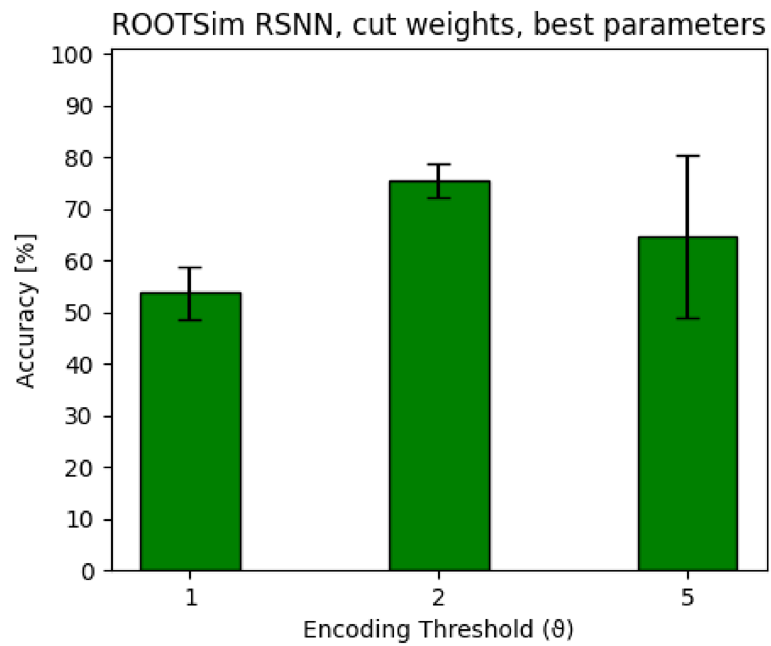


Figura 5.8: Simulazioni con i migliori parametri

Anche in questo caso è stato valutato l'impatto della presenza di sinapsi ricorrenti verso lo stesso neurone e di un delay sinpatico minimale, ed i risultati sono mostrati in Figura 5.9.

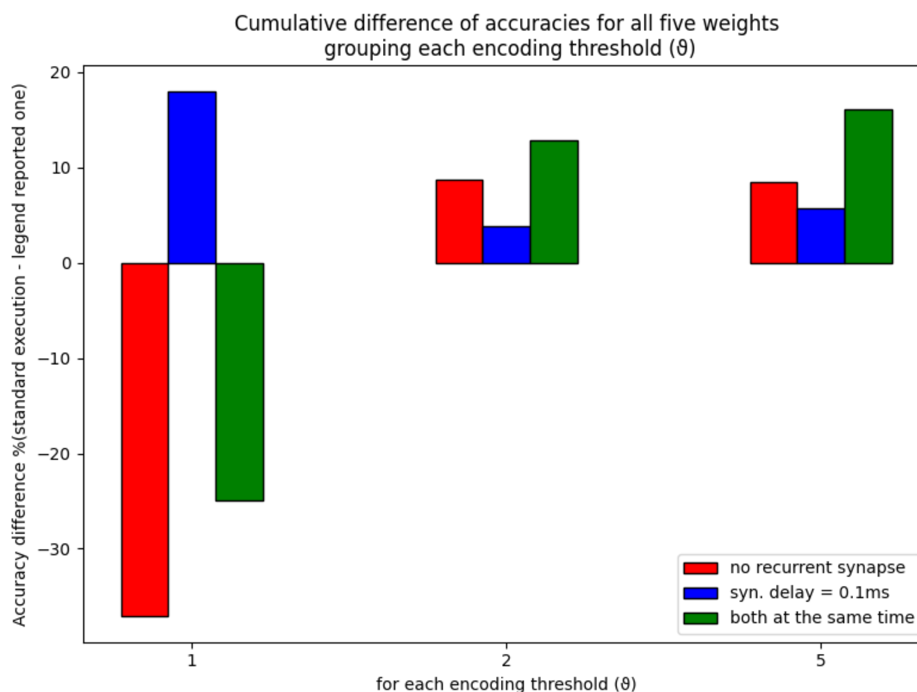


Figura 5.9: Differenza cumulativa in accuracy, raggruppando per encoding threshold, sulle 5 diverse simulazioni con i pesi ricorrenti scalati

In questo caso, delay sinaptici diversi e assenza di connessioni ricorrenti verso se stessi sembrano avere un impatto molto più significativo sull'accuracy della simulazione. In particolare si noti che l'assenza di connessione ricorrente, con l'encoding threshold 1, sembra produrre un miglioramento dell'accuracy medio di più del 5% per ciascun set di pesi. Questo porterebbe l'accuracy media di quella simulazione da circa 55% a circa 60%. tra le alternative invece, nulla impatta in maniera particolare, tantomeno positiva, gli altri threshold. I risultati sono decisamente migliori rispetto alla prima soluzione. Anche evitando del tutto l'utilizzo di un delay artificiale tra spike, si ottengono risultati discreti, che nel caso specifico dell'encoding threshold 2 sono comparabili con quelli del paper. L'encoding threshold 5, tenendo conto della grande deviazione standard, si comporta bene e in maniera simile ai risultati della rete replicata (che sono più bassi rispetto a quelli presenti nel paper). Tutte e due queste reti, quella replicata e la simulazione con ROOT-Sim, presentano un set di pesi nello specifico che quando utilizzato porta l'accuracy ad un valore molto inferiore (diverse decine di punti %) in confronto al risultato medio ottenuto

con gli altri quattro set.

5.1.2 FFSNN

La simulazione di questa rete ha presentato meno problemi rispetto alla RSNN. L'assenza delle connessioni ricorrenti, e quindi di "memoria" da parte della rete, oltre a semplificare la topologia, apparentemente rende la simulazione più semplice da gestire con il cambio di paradigma di simulazione. Non è stato necessario alterare i pesi delle connessioni e nemmeno utilizzare dei delay artificiali: è stato sufficiente testare diversi valori di periodo refrattario, mantenendo un delay sinaptico pari alla dimensione del timestep per quel determinato θ come fatto fin'ora, per ottenere quanto mostrato nei grafici in Figura 5.10.

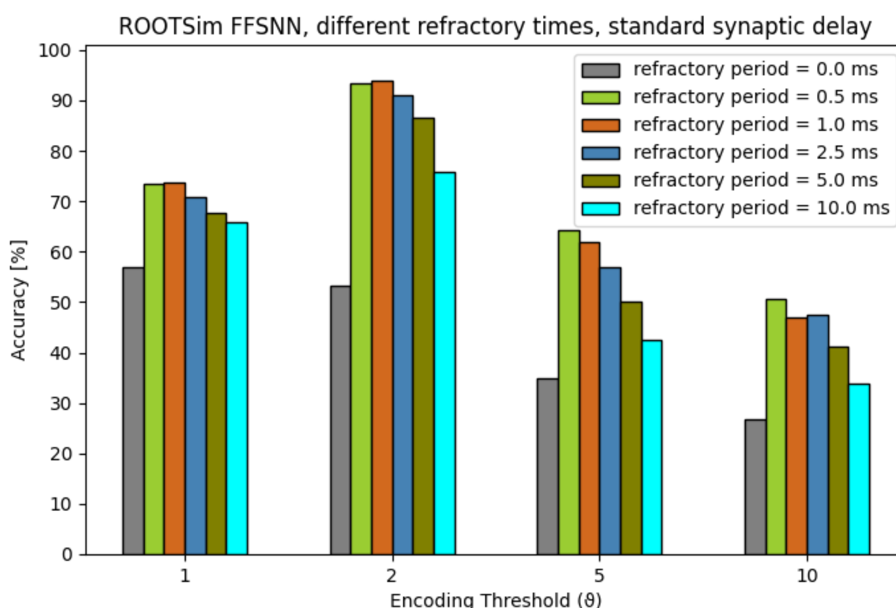


Figura 5.10: Simulazione FFSNN con diversi periodi refrattari

Anche in questo caso è stata effettuata una valutazione dell'impatto di un delay sinaptico minimale anziché standard. I risultati sono mostrati in Figura 5.11.

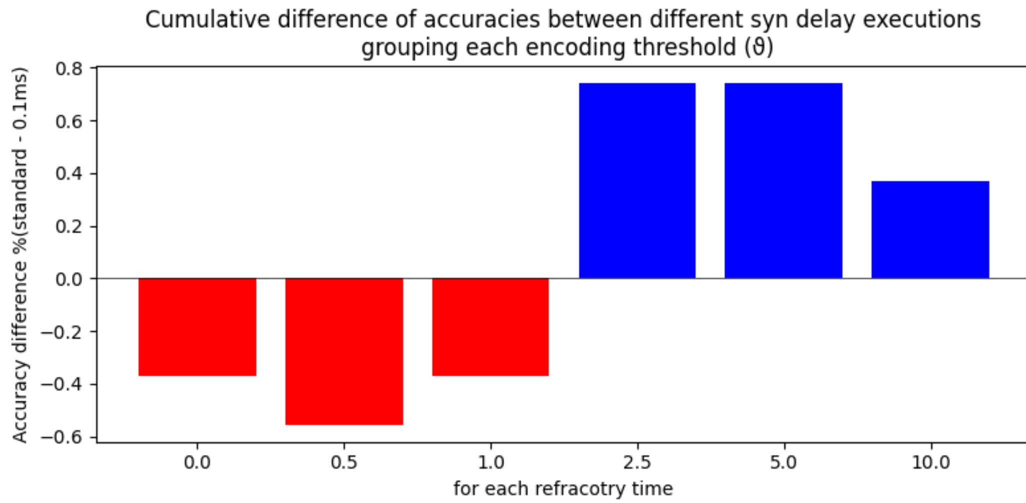


Figura 5.11: Differenza cumulativa in accuracy delle esecuzioni con differenti delay sinaptici, per i diversi periodi refrattari

In maniera ancora più marcata rispetto al caso dell'RSNN la differenza è totalmente trascurabile. La validazione è stata effettuata quindi eseguendo sui cinque set di pesi a disposizione con il periodo refrattario ritenuto migliore in base al grafico in Figura 5.10, ovvero 1.0ms, e con delay sinaptico standard. I risultati sono mostrati nel grafico in Figura 5.12.

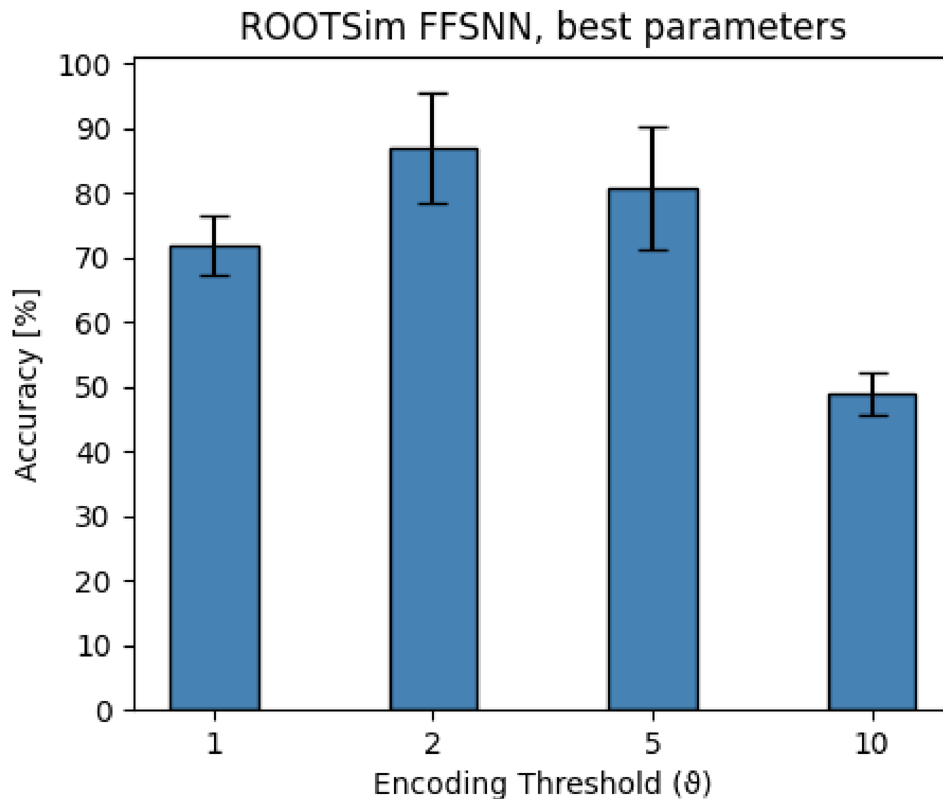


Figura 5.12: Risultati migliori FFSNN su ROOT-Sim

Per quanto riguarda questa tipologia di rete, i risultati sono al di sopra delle aspettative: per ciascun threshold l'accuratezza media è al di sopra sia dell'accuratezza ottenuta dalla replicazione del modello, che di quella presentata nel paper. La differenza maggiore è chiaramente ottenuta sull'encoding threshold $\theta = 2$ con una differenza di accuracy a volte maggiore del 20%. Questo threshold sembra essere il migliore, indipendentemente dalla piattaforma di simulazione utilizzata o dal tipo di rete. Si può concludere che l'esecuzione senza timestep ha un comportamento generale migliore rispetto all'esecuzione timestepped, e resta da chiedersi se con pesi allenati in maniera specifica per questo paradigma di simulazione non si possano ottenere risultati migliori.

5.2 Training

Per quanto riguarda l'applicazione di STDP, molti tentativi, prove e idee sono state messe alla prova. Dato il tempo richiesto per effettuare anche una sola epoca di training, soprattutto in una rete RSNN, la scelta iniziale è stata quella di concentrarsi su un set di pesi di partenza specifico (il migliore) e un threshold θ specifico. I primi tentativi hanno evidenziato l'importanza dei parametri η_+ e η_-: è molto facile infatti arrivare ad una situazione in cui la rete non effettua più alcuno spike, se il secondo è troppo grande e dominante sul primo (e quindi si favorisce molto la depressione rispetto al potenzialmento), e viceversa a spikare in maniera incontrollata quasi indipendente dall'input nel caso in cui è il primo ad essere dominante. Empiricamente, la scelta migliore sembra essere mantenere i due η di pari valore, opure differenziarli di poco quando sono particolarmente piccoli. In letteratura non ci sono indicazioni specifiche per il valore di questi parametri. Le prove sono state effettuate con valori che spaziano dall'ordine di 10^{-3}, che si sono rivelati decisamente grandi (e che, anche nel caso in cui i due η fossero uguali, portavano a una rete in cui l'output layer smetteva di spikare), a 10^{-13}, bassi con variazioni poco significative sui pesi delle sinapsi e quindi sull'output della rete. Anche il numero di batch tra le varie prove è variato, tra batch di dimensione 1, 10, 100, metà del dataset o il dataset intero. In questo primo ciclo di simulazioni nessun risultato significativo è stato ottenuto: nella stragrande maggioranza delle simulazioni l'accuratezza sul training set diminuiva fin da subito, a volte in maniera anche repentina, mentre in altre rimaneva a valori abbastanza stabili intorno a quello di partenza, senza miglioramenti significativi. Il primo cambiamento preso in considerazione include l'aggiunta di due variazioni per STDP come mostrato in 5.1. In riferimento a quel codice, il primo parametro aggiunge la possibilità di considerare per il calcolo una finestra temporale specifica di *window_size* · *2*: per una sinapsi x il calcolo considererà, dato il neurone postsinaptico che ha generato uno spike al tempo z, solamente gli spike arrivati dal neurone presinaptico al tempo *z'* con $z - window_size <= z' <= z + window_size$. In

```
/* ... */
    double window_size; //[ms]
    bool use_scaling_factor;
    bool correctly_predicted;
    double scaling_factor;
/* ... */
```

Listing 5.1: aggiunta dei parametri su finestra temporale e di scala

letteratura in generale questa considerazione non sembra essere prevista dato che la formula di base assegna un peso molto basso a spiketime particolarmente distianti. L'aggiunta è stata ispirata dal lavoro in [20], tuttavia lì l'idea dell'aggiunta di una finestra temporale per l'applicaizone di STDP è nata da considerazioni riguardo la particolare topologia della rete, che è diversa rispetto a quella esaminata in questa tesi. L'utilizzo della finestra di per sè non ha prodotto alcuna variazione significativa nei risultati degli esperimenti.

Gli altri tre parametri servono ad aggiungere un ulteriore fattore di scala, nel caso in cui le predizione siano corrette. L'idea di partenza è quella di premiare l'apprendimento ottenuto da esecuzioni che hanno portato alla corretta predizione, moltiplicando per un fattore di scala il delta sui pesi che queste esecuzioni calcolano. Di conseguenza, se si vuole utilizzare questa soluzione, bisogna configurare *use_scaling_factor* a true e assegnare un valore a *scaling_factor*. Le prove effettuate utilizzando questo fattore, indipendentemente dal suo valore, non hanno evidenziato risultati significativi. Aumentando il suo valore, la rete sembra ottenere una degradazione rispetto al caso in cui ha un valore contenuto o non viene utilizzato. con entrambe queste aggiunte, le prove sono state effettuate su batch di varia grandezza. Questo effetto, però, è probabilmente legato al valore del parametro moltiplicativo η (soprattutto quando abbiamo batch grandi o un unico batch): Nella situazione corrente, la percentuale di accuratezza sul training set con threshold $\theta = 2$ è molto alta ($> 97\%$), dunque si può dire che il training sulla quasi totalità dei casi utilizzerà il fattore di scala. Il suo effetto pratico è dunque che, anziche avere concettualmente un sistema che premia l'apprendimento nelle situazioni desiderate, abbiamo un sistema che, in maniera indiscriminata, utilizza un η scalato di quel fattore: se utilizzare un fattore di scala dovesse essere un'opzione utile

probabilmente lo sarebbe in casi in cui l'accuratezza è minore.

Sono state provate anche altre leggere variazioni. Verranno menzionate per completezza, ma senza soffermarsi sui dettagli, dato che non hanno prodotto risultati degni di nota:

- Apprendimento inverso: se la predizione è sbagliata, i delta avranno il valore opposto

- Parametri diversi tra layer diversi: il calcolo per il layer ricorrente utilizzerà un η diviso per un fattore 50. Questa idea è motivata dal fatto che, nel codice del paper, le sinapsi ricorrenti vengono generate con valori molto più piccoli rispetto alle altre, e questa differenza di scala è riscontrabile anche nel valore finale delle sinapsi

- Scalare ulteriormente di un certo valore i delta, ma solamente nei casi di potenziamento di una sinapsi verso il neurone di output giusto della simulazione in esame

A questo punto, si è ritenuto che da questo set di pesi non si potesse ottenere più nulla. È importante notare che la configurazione ricorrente, scelta dagli autori del paper per massimizzare l'accuratezza della rete quando simulata con paradigma timestepped, non è risultata adeguata quando applicata alla simulazione a tempistiche precise, e quindi la conclusione più probabile è che il training stesso sia inadeguato per questo caso di studio quando applicato a questo genere di rete.

L'idea successiva, per poter validare la procedura di STDP, è quella di tentare di applicarlo a pesi generati randomicamente per allenare da zero una rete. Per semplificare il lavoro, per questa fase si è scelta la rete con il threshold $\theta = 10$ che, data la sparsità degli spike presenti nel dataset, permette un'esecuzione molto più rapida. Inizialmente, inoltre, si è scelto di partire da una rete FFSNN per velocizzare ulteriormente l'esecuzione in maniera significativa. Così facendo è stato possibile valutare il percorso di apprendimento su più di un' epoca per ciascun tentativo in tempi relativamente brevi. Il primo problema che ci si è posti è come rappresentare l'output in questo caso. con

STDP infatti l'idea, semplificando, è quella di potenziare le sinapsi quando gli spike presinaptici hanno una forte correlazione con quelli postsinaptici, questo su tutto il percorso dall'input all'output. Lo stesso discorso vale per le sinapsi da deprimere, a causa dell' indipendenza fra spike pre e postsinaptici. Questo però implica che la rete abbia già una struttura ben definita: dato il funzionamento dell'algoritmo, questo ha senso solo su una rete con delle sinapsi con valori coerenti rispetto a quello che deve essere l'output della rete e quindi a quello che si vuole ottenere. Quando i pesi vengono generati a caso, il pattern di spike della rete è ovviamente casuale e, in una certa misura, poco dipendente dall'input: i neuroni di output che arriveranno a spikare, e quindi la predizione che ne verrà fuori, sarà anch'essa casuale. Inoltre, provando ad eseguire la rete con questi pesi random, si può notare che ci sono gruppi di neuroni di output che tendono a non spikare mai, e gruppi che tendono a spikare in maniera preponderante. Allenare con STDP una rete generata così quindi porterà a variare i pesi delle sinapsi in funzione del suo comportamento casuale, ottenendo un comportamento non predicibile e funzione della randomicità dei pesi e non di ciò che si vuole ottenere in output. La prima idea è stata quella di far vincere sempre il neurone di output corretto in fase di training. Il primo modo per implementarlo è stato modificando la corrente in ingresso ai neuroni: possiamo inserire una corrente positiva al neurone di output corretto per farlo spikare più spesso indipendentemente dall'input, una corrente negativa nei neuroni di output sbagliati per frenarli dallo spikare, o tutte e due. Tuttavia decidere il valore della corrente a priori è semplice se si effettua un training con un solo batch, e quindi si può valutare in anticipo il numero di spike medio del neurone vincente e quindi di quanto caricare la corrente per ottenere il risultato sperato. Allenare con batch di un solo elemento alla volta invece, provocherà dei cambiamenti alla rete ad ogni iterazione, e quindi sarebbe richiesta una valutazione della corrente da utilizzare passo dopo passo. Per semplificare questo aspetto, si è optato per un'implementazione leggermente diversa il cui effetto dovrebbe essere esattamente lo stesso: modificare il contenuto degli array dei tempi di spike prima di utilizzarli, in modo che per il neurone giusto figurino

più spike di quanti ce ne sono davvero stati e l'opposto per i neuroni sbagliati. Per fare questo sono stati aggiunti diversi parametri nella struttura *stdp_params* come mostrato in 5.2, da inizializzare nel file di configurazione apposito del learning. I primi due parametri (*add_spike* e *remove_spikes*) indicano se

```
1  struct stdp_params {
2      /* ... */
3      bool add_spikes;
4      bool remove_spikes;
5      bool random_remove;
6      int target;
7      int prediction;
8      int max_spikes;
9      int tg_spikes;
10     double min_output_time; //the starting point in time for added spikes
11     double max_output_time; //the ending point in time for added spikes
12     double spikes_to_add;
13     double spikes_to_remove;
14     /* ... */
```

Listing 5.2: parametri per gestire l'aggiunta o rimozione artificiale di spike in output

utilizzare questa soluzione, da un lato per aggiungere spike, dall'altro per toglierne. Per quanto riguarda la rimozione, sono state previste due possibilità: una rimozione randomica di un certo numero x di spike, oppure la rimozione degli ultimi x spike dalla fine, e il terzo parametro (*random_remove*) permette di decidere se utilizzare la prima opzione quando vale "true" o la seconda quando vale "false". I successivi quattro parametri (*target*, *prediction*, *max_spikes* e *tg_spikes*) tengono traccia delle informazioni specifiche della simulazione in corso riguardanti il neurone target che dovrebbe spikare di più, quello che ha realmente generato più spike, e il loro numero totale di spike. Chiaramente il loro valore viene inizializzato al termine della simulazione. I successivi due parametri (*min_output_time* e *max_output_time*) permettono di scegliere gli intervalli di tempo in cui aggiungere gli spike. L'idea è stata quella di aggiungere gli spike in maniera uniforme in questo intervallo di tempo, e per le prove selezionare un intervallo di tempo che sia più vicino possibile al termine dell'esecuzione, in modo da avere la maggior parte degli spike presinaptici antecedenti a questi falsi spike, e quindi amplificare il più possibile l'effetto di potenziamento che ne risulta. Gli ultimi due parametri (*spikes_to_add* e *spikes_to_remove*) permettono di specificare un valore percentuale di elemen-

ti su cui lavorare. Nel caso degli spike da aggiungere, il valore x indica che il neurone target corretto, dopo l'aggiunta dei falsi spike, arriverà ad averne un valore pari a quelli del neurone (sbagliato) che ha generato più spike aumentato di $x\%$. Questo permette di decidere quanto modificare l'output: un valore troppo piccolo potrebbe non apportare alcun miglioramento, un valore troppo grande potrebbe far divergere facilmente il valore delle sinapsi a valori che favoriscono lo spike di neuroni di output specifici.

Una delle ispirazioni per l'idea di questa strategia è il lavoro effettuato in [10]. Tuttavia, anche in questo caso, la topologia della rete è diversa, in particolare è ricorrente su tutti i layer (compreso quello di output) e la connessione sinaptica tra hidden layer e output layer avviene in tutti e due i versi. Questo significa in primo luogo che per la simulazione di questo paper l'unica possibilità è quella di aggiungere corrente, perché gli spike devono avvenire realmente durante la simulazione: a differenza del modello esaminato in questa tesi, uno spike in un neurone di output verrà propagato, e quindi avrà ripercussioni, sull'intera rete. In secondo luogo, questo effetto a cascata (e quindi questa topologia) potrebbe essere fondamentale per la riuscita del training con questa strategia. Le simulazioni effettuate infatti non hanno prodotto alcun risultato utile nemmeno in questo caso. La conclusione tratta è che non è applicabile per questa rete.

Il fatto che gli spike aggiunti artificialmente non abbiano alcuna reale correlazione con il comportamento del resto della rete porta a pensare che applicando le strategie viste in precedenza, se dividessimo la rete a metà, ciò che si impara nella prima metà (tra input layer e hidden layer) si potrebbe considerare totalmente scorrelato da ciò che si impara nella seconda metà (tra hidden layer e output layer). Questo per dire che, se questa osservazione è corretta, allora le idee precedenti non portano a un algoritmo che permetta davvero di correlare l'input al comportamento della rete necessario per ottenere l'output desiderato. Inoltre, l'aggiunta di spike arbitrari non garantisce nemmeno con certezza l'effetto di potenziamento desiderato: anche se piazzati al termine dell'esecuzione, la quantità di potenziamento della sinapsi data

dagli spike presinaptici che però sono lontani temporalmente è insignificante rispetto alla quantità di depressione data eventualmente da spike presinaptici scorrelati. Per fare un esempio, se uno spike artificiale viene aggiunto al timestamp 1000ms, il potenziamento della sinapsi prodotto da uno spike presinaptico avvenuto con timestamp 100ms è del tutto irrilevante rispetto alla depressione indotta da un altro spike presinaptico che però è avvenuto con timestamp 1001ms. Il successivo aggiornamento dell'algoritmo è frutto di questa osservazione: la predizione della rete si basa solo ed esclusivamente sul numero di spike, quindi il training si può limitare a rafforzare indistintamente le sinapsi giuste, senza tener alcun conto dei tempi relativi di spike, in base al comportamento casuale che ha la rete. Questo, nella pratica, si traduce in un algoritmo che applica l'STDP tradizionale sulle sinapsi tra layer di input e hidden layer (e sulle sinapsi ricorrenti). Sul resto della rete, invece, si potenziano le sinapsi dirette da un qualsiasi neurone dell'hidden layer al neurone di output che deve spikare di più nella simulazione corrente, mentre si depotenziano le altre sinapsi dirette ai neuroni sbagliati. Il potenziamento/depressione sarà di un valore pari a $x{\cdot}\eta$, dove x è il numero di spike in ingresso al neurone di output attraverso quella sinapsi (che quindi sarà pari al numero di spike effettuati dal neurone dell'hidden layer presinaptico) ed η è un nuovo parametro di learning il cui valore è da definire, implementato come mostrato in 5.3. In riferimento

```
1  struct stdp_params {
2      /* ... */
3      bool strenghten_output;
4      double strenghtening_learning_rate;
5      /* ... */
```

Listing 5.3: parametri per l'apprendimento senza utilizzare i tempi di spike

al codice appena menzionato, il primo parametro indica se utilizzare questa soluzione, il secondo quanto deve valere il parametro η. Basandosi sul fatto che il numero medio di spike di un neurone dell'hidden layer è nell'ordine di qualche decina e sul fatto che nel training set ci sono tra i 3000 e 5000 elementi in base al tipo di rete, il valore di questo parametro dovrebbe essere sufficientemente basso per evitare una situazione in cui i pesi divergono in una qualche

direzione, positiva o negativa. Inoltre, per il fatto che per ogni simualzione il potenziamento di una sinapsi è corrisposto da 26 depressioni, si è valutato anche di ridurre di qualche fattore solamente il valore della depressione. Infine, per evitare che il peso durante le varie iterazioni superi i valori di bound (cosa che con un learning rate sufficientemente basso non dovrebbe comunque avvenire) si è applicato lo stesso meccanismo di soft bound applicato con STDP base. La speranza è da un lato rafforzare la correlazione tra input layer e hidden layer in funzione dei pesi sinaptici generati casualmente, e dall'altra rafforzare la correlazione fra l'attività dell'hidden layer (che in senso globale sarà casuale dato che è frutto dei pesi randomici, ma poi sarà anche funzione dell'input dato che avrà subito un processo di apprendimento con STDP sulle sinapsi che collegano questi due layer) e output layer. Tuttavia, al momento, nemmeno questo approccio ha prodotto i risultati sperati. La conclusione più probabile è che l'algoritmo specifico di STDP utilizzato non sia adatto al caso di studio: questo, infatti, potrebbe essere adatto al training di reti con sinapsi esclusivamente eccitatorie. Come abbiamo già detto, invece, tutte le reti analizzate in questa tesi presentano sia sinapsi eccitatorie che inibitorie. Prima di concludere il capitolo, si vuole evidenziare infine che le varie prove hanno compreso l'utilizzo di ambedue le funzioni di learning window per STDP menzionate in 3.5 ed 3.6. Sebbene le prove, come detto in precedenza, non hanno portato ai risultati sperati, il probabilistic STDP sembra produrre risultati leggermente migliori. Per concludere il paragrafo, si faccia riferimento alla Tabella 5.1 per una panoramica su una ristretta selezione degli esperimenti effettuati con STDP. Si faccia riferimento all'elenco al termine del paragrafo per una panoramica dei parametri utilizzati per questi esperimenti.

Per quanto riguarda i parametri, per le varie configurazioni:

- 1: RSNN con pesi ricorrenti scalati di 5, periodo refrattario 0.25ms, probabilistic STDP, no window, soft bounds, scaling factor x2, due batch di dimensione metà training set, $\eta_+ = \eta_- = 10^{-12}$, w_{min} e $w_{max} = 0.8/-0.8$, $\tau_+ = \tau_- = 10ms$

Conf.	ϑ	Iniziale	Numero epoca 1	2	3	4	5	6	7	8
1	2	79.54 %	79.82 %							
2	2	79.54 %	79.63 %	79.48 %						
3	10	5.56 %	5.0 %	5.37 %	5.74 %	5.74 %	5.74 %			
4	10	5.55 %	5.19 %	4.81 %	5.19 %	4.44 %	4.63 %	4.07 %	3.89 %	4.26 %
5	10	5.56 %	3.89 %	4.07 %	5.0 %	5.93 %	6.30 %	6.67 %	5.56 %	4.08 %
6	10	3.15 %	3.15 %	3.33 %	3.52 %	3.33 %	2.96 %	3.33 %	3.15 %	3.15 %

Tabella 5.1: Accuracy di alcuni degli esperimenti effettuati. La prima colonna di accuracy indica quella iniziale, le seguenti indicano le accuracy dopo ciascuna epoca di training.

- 2: RSNN con pesi ricorrenti scalati di 5, periodo refrattario 0.25ms, standard STDP, no window, soft bounds, scaling factor x1.5, x2 sul potenziamento delle sinapsi verso gli output neuron corretti, batch unitario, $\eta_+ = \eta_- = 8 \cdot 10^{-11}$, w_{min} e $w_{max} = 0.8/-0.8$, $\tau_+ = \tau_- = 10ms$
- 3: FFSNN con pesi random, periodo refrattario 5ms, standard STDP, no window, soft bounds, no scaling factor, batch unitario, $\eta_+ = \eta_- = 2.5 \cdot 10^{-8}$, w_{min} e $w_{max} = 0.7/-0.7$, $\tau_+ = \tau_- = 10ms$, $spikes_to_add = 20\%$, rimozione spikes random con $spikes_to_remove = 50\%$
- 4: FFSNN con pesi random, periodo refrattario 5ms, standard STDP, no window, soft bounds, no scaling factor, batch unitario, $\eta_+ = \eta_- = 2 \cdot 10^{-7}$, w_{min} e $w_{max} = 0.7/-0.7$, $\tau_+ = \tau_- = 10ms$, $spikes_to_add = 150\%$, rimozione spikes random con $spikes_to_remove = 25\%$
- 5: FFSNN con pesi random, periodo refrattario 5ms, probabilistic STDP, no window, soft bounds, no scaling factor, batch unitario, $\eta_+ = \eta_- = 10^{-5}$, w_{min} e $w_{max} = 1.0/-1.0$, $\tau_+ = \tau_- = 10ms$, $spikes_to_add = 50\%$, rimozione spikes random con $spikes_to_remove = 50\%$
- 6: FFSNN con pesi random, periodo refrattario 5ms, standard STDP, no window, soft bounds, no scaling factor, batch unitario, w_{min} e $w_{max} = 1.0/-1.0$, $\eta_+ = \eta_- = 4 \cdot 10^{-8}$, rinforzo dell'output in base agli spike dell'hidden layer—e non basato su STDP—con learning rate $= 8 \cdot 10^{-8}$

5.3 Efficienza Energetica

Per quanto riguarda l'efficienza energetica, si è valutato il consumo che avrebbe un'eventuale implementazione su hardware dedicato del modello migliore analizzato nel paper, ovvero l'RSNN con i suoi vari threshold, confrontato con quello del modello migliore ottenuto con ROOT-Sim, ovvero la FFSNN. In realtà, un hardware specifico per implementare un modello come quello simulato con ROOT-Sim correntemente non esiste: per l'analisi sono stati utilizzati i valori medi di consumo energetico per quanto riguarda hawrdware dedicato già eistente e di cui si hanno informazioni in [29]. Per fare ciò, per tutte le reti menzionate sono stati raccolti i dati riguardanti il numero di spike totale di ciascun layer e di ciascun neurone, per la simulazione del testing set. ricordiamo che il codice del paper di riferimento presenta un'incongruenza fra i set delle due diverse reti: siccome qui non siamo interessati a una valutazione dell'accuratezza, per questo caso specifico i dati dell'esecuzione con ROOT-Sim sono stati raccolti utilizzando comunque lo stesso set dell' RSNN. Stando ai dati raccolti, la RSNN ha un numero totale di spike nell'hidden layer di un ordine di grandezza inferiore a quello della FFSNN su ROOT-Sim dalle migliori prestazioni in termini di accuratezza. In particolare, la ricorrente effettua in media 10^6 spike totali, mentre l'altra rete 10^7: stando alle dimensioni dell'hidden layer (450) e alla dimensione del test set utilizzato (1085), questo significa in media poco più di due spike per neurone dell'hidden layer nel caso della rete ricorrente (rispettivamente 20 nel caso della rete non ricorrente) . L'input layer ha un numero di spike di molto inferiore rispetto a quello dell'hidden layer e un numero di sinapsi identico tra le due reti: anche se il suo contributo è stato considerato per il calcolo dei risultati, in realtà si può facilmente verificare che è trascurabile rispetto a quello dell'hidden layer, ancor di più nel caso RSNN.

Per quanto riguarda il modello utilizzato, ci basiamo sull'equazione 9 in [29]:

$$E_{syntot} = E_{syn} + \frac{E_{neu}}{r_a \cdot s_{neu}} \tag{5.1}$$

dove:

E_{syntot} = totale energia dissipata per un messaggio che attraversa una sinapsi

$$E_{syn} = \text{energia dissipata per sinapsi}$$

$$E_{neu} = \text{energia dissipata da un neurone}$$

$$s_{neu} = \text{sinpasi per neurone}$$

$$r_a = \text{percentuale di sinapsi attive} \tag{5.2}$$

Ora, per chiarificare l'equazione e semplificare il calcolo osserviamo quanto segue:

- il termine $r_a \cdot s_{neu}$ non è altro che il numero di sinapsi di ciascun neurone presinaptico che ha prodotto il messaggio che attraversa la sinapsi presa in esame. Per i neuroni dell'hidden layer queste saranno $450 + 27$ (connessione ricorrente e connessione verso ciascun neurone dell'output layer) nel caso dell'RSNN, 27 (solo connessione verso ciascun neurone dell'output layer) nel caso dell'FFSNN. Chiamiamo questo termine, per semplicità, S_n. Inoltre abbiamo stabilito che i neuroni in output avranno un'impatto come se avessero una sola sinapsi in uscita.

- L'equazione presentata calcola il contributo energetico per evento sinaptico, ovvero per un solo messaggio trasmesso lungo una specifica sinapsi quando un neurone effettua uno spike. Sappiamo, quindi, che lo spike di ciascun neurone contribuirà al totale S_n volte il consumo per evento sinaptico. chiamiamo questo contributo E_{spike}

Manipolando l'equazione precedente dunque otteniamo:

$$E_{spike} = S_n \cdot E_{syntot} = (S_n) \cdot (E_{syn} + \frac{E_{neu}}{S_n}) = S_n \cdot E_{syn} + E_{neu} \tag{5.3}$$

Utilizzando l'equazione appena ottenuta per calcolare il contributo di ciascuno spike, tenendo conto del valore di S_n strettamente dipendente dal layer

preso in considerazione—valore che però è costante tra tutti i neuroni dello stesso layer, dato che presentano esattamente le stesse connessioni sinaptiche, fatto che ci permette di considerare layer per volta anziché neurone per volta—e dei valori medi di consumo E_{syn} pari a 621.645 aJ e di E_{neu} pari a 2477.112 aJ, otteniamo i risultati presentati in Figura 5.13.

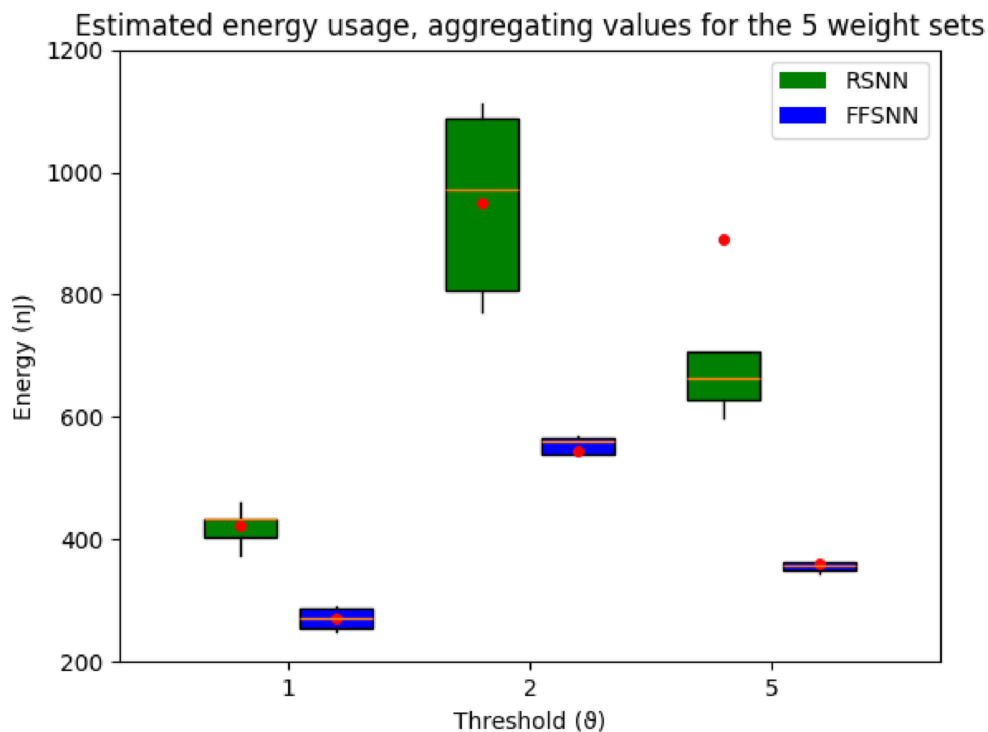


Figura 5.13: Consumi energetici stimati dalla simulazione del test set su un eventuale hardware per le due reti

Come si può vedere, l'FFSNN oltre alla migliore accuratezza sembra permettere anche un notevole risparmio energetico.

6. Conclusioni e Lavori Futuri

Il lavoro presentato ha evidenziato chiaramente i limiti dell'approccio timestepped adottato in letteratura: una simulazione a grana fine permette anche a una rete non ricorrente, strutturalmente più semplice, di essere competitiva e anzi, di avere prestazioni addirittura migliori in termini di accuratezza di predizione rispetto alla migliore rete timestepped, che per sopperire ai suoi limiti introduce uno strato addizionale di sinapsi. Questa rete con meno sinapsi è vantaggiosa per più di una ragione: non solo è computazionalmente più semplice da simulare, indipendentemente dal simulatore adottato, ma come abbiamo visto, anche a livello di consumi energetici previsti, quando un hardware adeguato verrà creato, sembra essere vincente. A livello di consumi energetici inoltre le potenzialità sono ancora alte: quello che ha evidenziato l'RSNN timestepped è che si può raggiungere un'accuratezza abbastanza alta anche con un numero molto esiguo di spike. Questo implica che verosimilmente una rete FFSNN con accuratezza comparabile ma numero di spike molto inferiore è possibile da ottenere, rete che avrebbe un'impatto energetico ancora minore rispetto alle stime già ottime. Per fare ciò, però, è necessario sviluppare un processo di training più sostenibile, che, come abbiamo visto in precedenza, è l'unico elemento limitante per la rete con tempistiche accurate in termini di accuratezze ottenute (e di tempi di esecuzione). Proprio per questo per il futuro ci sono diverse opzioni da valutare:

- In primo luogo, ottimizzare l'integrazione del processo di training nel simulatore per renderlo più sostenibile
- Valutare accuratamente qual è l'aproccio migliore da adottare per il trai-

ning. I risultati ottenuti in questo lavoro non sono esaustivi a rispondere a diverse domande, come: c'è un modo di codificare l'output qui ideato che possa andar bene? In caso di risposta affermativa, i risultati ottenuti potrebbero essere frutto di una scelta non ottimale dei parametri di learning. Altrimenti, il problema andrà analizzato più a fondo

- Verificare se le implementazioni di STDP offline di questa tesi e quella online [41] sono equivalenti, oppure se l'alternativa online è in realtà più accurata, e permette di ottimizzare meglio i pesi

- Verificare se è effettivamente possibile semplificare una rete FFSNN più di quanto visto in questo lavoro. Per semplificazione intendiamo sia nei confronti del livello di attivazione (e quindi numero di spike) di una rete dalla topologia costante, che permetterebbe un risparmio energetico ancora più marcato rispetto alla controparte mantenendo l'accuratezza originale, sia nei confronti della topologia stessa della rete, che potrebbe essere ridotta ad avere meno sinapsi e/o meno neuroni rispetto a quanti ne aveva in origine. In questo caso, sarebbe utile da confrontare il grado di semplificazione possibile della topologia tra le due differenti reti nel caso di paradigmi di simulazione diversa, a parità di accuratezza

Per il momento le prospettive sono più che incoraggianti. Ovviamente, quanto mostrato in questa tesi è un primo passo, mentre il successivo grande passo sarà la reale implementazione in hardware di queste nuove precise SNN.

Bibliografia

[1] 2022. "https://github.com/event-driven-robotics/tactile_braille_reading"

[2] 2022. "https://github.com/event-driven-robotics/tactile_braille_reading/blob/main/notebooks/braille_reading_ffsnn.ipynb"

[3] 2022. "https://github.com/event-driven-robotics/tactile_braille_reading/blob/main/notebooks/braille_reading_rsnn.ipynb"

[4] Daniel Auge, Julian Hille, Etienne Mueller, and Alois Knoll. 2021. A Survey of Encoding Techniques for Signal Processing in Spiking Neural Networks. *Neural Processing Letters* 53, 6 (01 Dec 2021), 4693–4710. https://doi.org/10.1007/s11063-021-10562-2

[5] H Bostock, K Cikurel, and D Burke. 1998. Threshold tracking techniques in the study of human peripheral nerve. *Muscle Nerve* 21, 2 (Feb. 1998), 137–158.

[6] Mohamed Sadek Bouanane, Dalila Cherifi, Elisabetta Chicca, and Lyes Khacef. 2023. Impact of spiking neurons leakages and network recurrences on event-based spatio-temporal pattern recognition. *Front. Neurosci.* 17 (Nov. 2023), 1244675. https://doi.org/10.3389/fnins.2023.1244675

[7] Christopher D Carothers, Kalyan S Perumalla, and Richard M Fujimoto. 1999. Efficient Optimistic Parallel Simulations Using Reverse Computation. *ACM Transactions on Modeling and Computer Simulation* 9, 3 (July 1999), 224–253. https://doi.org/10.1145/347823.347828

[8] K M Chandy and J Misra. 1981. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM* 24, 4 (April 1981), 198–206. https://doi.org/10.1145/358598.358613

[9] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, Yuyun Liao, Chit-Kwan Lin, Andrew Lines, Ruokun Liu, Deepak Mathaikutty, Steven McCoy, Arnab Paul, Jonathan Tse, Guruguhanathan Venkataramanan, Yi-Hsin Weng, Andreas Wild, Yoonseok Yang, and Hong Wang. 2018. Loihi: A Neuromorphic Manycore Processor with On-Chip Learning. *IEEE micro* 38, 1 (Jan. 2018), 82–99. https://doi.org/10.1109/mm.2018.112130359

[10] Vyacheslav Demin and Dmitry Nekhaev. 2018. Recurrent Spiking Neural Network Learning Based on a Competitive Maximization of Neuronal Activity. *Front Neuroinform* 12 (Nov. 2018), 79.

[11] Rafael Diaz and Joshua G. Behr. 2010. *Discrete-Event Simulation.* John Wiley & Sons, Ltd, Chapter 3, 57–98. https://doi.org/10.1002/9780470590621.ch3 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470590621.ch3

[12] Jason K. Eshraghian, Max Ward, Emre O. Neftci, Xinxin Wang, Gregor Lenz, Girish Dwivedi, Mohammed Bennamoun, Doo Seok Jeong, and Wei D. Lu. 2023. Training Spiking Neural Networks Using Lessons From Deep Learning. *Proc. IEEE* 111, 9 (2023), 1016–1054. https://doi.org/10.1109/JPROC.2023.3308088

[13] Josef Fleischmann and Philip A Wilsey. 1995. Comparative Analysis of Periodic State Saving Techniques in Time Warp Simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation (PADS '95).* IEEE Computer Society, Piscataway, NJ, USA, 50–58. https://doi.org/10.1145/214282.214298

[14] Richard M. Fujimoto. 1990. Parallel discrete event simulation. *Commun. ACM* 33, 10 (oct 1990), 30–53. https://doi.org/10.1145/84537.84545

[15] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011. Deep Sparse Rectifier Neural Networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 15)*, Geoffrey Gordon, David Dunson, and Miroslav Dudík (Eds.). PMLR, Fort Lauderdale, FL, USA, 315–323. https://proceedings.mlr.press/v15/glorot11a.html

[16] W. Guo, M.E. Fouda, A.M. Eltawil, and K.N. Salama. 2021. Neural Coding in Spiking Neural Networks: A Comparative Study for Robust Neuromorphic Systems. *Frontiers in Neuroscience* 15 (2021), 638474. https://doi.org/10.3389/fnins.2021.638474

[17] Yufei Guo, Xinyi Tong, Yuanpei Chen, Liwen Zhang, Xiaode Liu, Zhe Ma, and Xuhui Huang. 2022. RecDis-SNN: Rectifying Membrane Potential Distribution for Directly Training Spiking Neural Networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 326–335.

[18] Bing Han, Gopalakrishnan Srinivasan, and Kaushik Roy. 2020. RMP-SNN: Residual Membrane Potential Neuron for Enabling Deeper High-Accuracy and Low-Latency Spiking Neural Network. https://doi.org/10.48550/arXiv.2003.01811 arXiv:2003.01811 [cs.NE]

[19] Zecheng Hao, Tong Bu, Jianhao Ding, Tiejun Huang, and Zhaofei Yu. 2023. Reducing ANN-SNN Conversion Error through Residual Membrane Potential. https://doi.org/10.48550/arXiv.2302.02091 arXiv:2302.02091 [cs.NE]

[20] Zhanhao Hu, Tao Wang, and Xiaolin Hu. 2017. *An STDP-Based Supervised Learning Algorithm for Spiking Neural Networks.* Springer International Publishing, 92–100. https://doi.org/10.1007/978-3-319-70096-0_10

[21] David R. Jefferson. 1985. Virtual time. *ACM Trans. Program. Lang. Syst.* 7, 3 (jul 1985), 404–425. https://doi.org/10.1145/3916.3988

[22] Bernard Katz and Ricardo Miledi. 1965. The measurement of synaptic delay, and the time course of acetylcholine release at the neuromuscular junction.

Proceedings of the Royal Society of London. Series B, Biological Sciences 161 (1965), 483–495. https://doi.org/10.1098/rspb.1965.0016

[23] Wenshuo Li, Hanting Chen, Jianyuan Guo, Ziyang Zhang, and Yunhe Wang. 2022. Brain-inspired Multilayer Perceptron with Spiking Neurons. arXiv:2203.14679 [cs.CV]

[24] Wolfgang Maass. 1997. Networks of spiking neurons: The third generation of neural network models. *Neural Networks* 10, 9 (1997), 1659–1671. https://doi.org/10.1016/S0893-6080(97)00011-7

[25] H Markram, J Lübke, M Frotscher, and B Sakmann. 1997. Regulation of synaptic efficacy by coincidence of postsynaptic APs and EPSPs. *Science* 275, 5297 (Jan. 1997), 213–215.

[26] Simon F Müller-Cleve, Vittorio Fra, Lyes Khacef, Alejandro Pequeño-Zurro, Daniel Klepatsch, Evelina Forno, Diego G Ivanovich, Shavika Rastogi, Gianvito Urgese, Friedemann Zenke, and Chiara Bartolozzi. 2022. Braille letter reading: A benchmark for spatio-temporal pattern recognition on neuromorphic hardware. *Front Neurosci* 16 (Nov. 2022), 951164.

[27] Emre O. Neftci, Hesham Mostafa, and Friedemann Zenke. 2019. Surrogate Gradient Learning in Spiking Neural Networks: Bringing the Power of Gradient-Based Optimization to Spiking Neural Networks. *IEEE Signal Processing Magazine* 36, 6 (2019), 51–63. https://doi.org/10.1109/MSP.2019.2931595

[28] Michael A. Nielsen. 2018. Neural Networks and Deep Learning. http://neuralnetworksanddeeplearning.com/

[29] Dmitri E. Nikonov and Ian A. Young. 2019. Benchmarking Delay and Energy of Neural Inference Circuits. *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits* 5, 2 (2019), 75–84. https://doi.org/10.1109/JXCDC.2019.2956112

[30] Alessandro Pellegrini. 2015. *Parallelization of Discrete Event Simulation Models*. Sapienza Università Editrice.

[31] Alessandro Pellegrini and Francesco Quaglia. 2014. The ROme OpTimistic Simulator: A Tutorial. In *Euro-Par 2013: Parallel Processing Workshops*, Dieter an Mey, Michael Alexander, Paolo Bientinesi, Mario Cannataro, Carsten Clauss, Alexandru Costan, Gabor Kecskemeti, Christine Morin, Laura Ricci, Julio Sahuquillo, Martin Schulz, Vittorio Scarano, Stephen L. Scott, and Josef Weidendorfer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 501–512.

[32] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. 2012. The ROme OpTimistic Simulator: Core Internals and Programming Model. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques (SIMUTOOLS)*. ICST, Brussels, Belgium, 96–98. https://doi.org/10.4108/icst.simutools.2011.245551

[33] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. 2015. Autonomic State Management for Optimistic Simulation Platforms. *IEEE Transactions on Parallel and Distributed Systems* 26 (2015), 1560–1569. https://doi.org/10.1109/TPDS.2014.2323967

[34] Paweł Pietrzak, Szymon Szczęsny, Damian Huderek, and Łukasz Przyborowski. 2023. Overview of Spiking Neural Network Learning Approaches and Their Computational Complexities. *Sensors* 23, 6 (2023). https://doi.org/10.3390/s23063037

[35] Adriano Pimpini. 2020. *High Performance Simulation of Spiking Neural Networks*. mathesis. Sapienza, University of Rome.

[36] Adriano Pimpini, Andrea Piccione, Bruno Ciciani, and Alessandro Pellegrini. 2022. Speculative Distributed Simulation of Very Large Spiking Neural Networks. In *Proceedings of the 2022 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Atlanta, GA, USA) *(SIGSIM-PADS '22)*. Association for Computing Machinery, New York, NY, USA, 93–104. https://doi.org/10.1145/3518997.3531027

[37] Adriano Pimpini, Andrea Piccione, and Alessandro Pellegrini. 2022. On the Accuracy and Performance of Spiking Neural Network Simulations. In *2022 IEEE/ACM 26th International Symposium on Distributed Simulation*

and Real Time Applications (DS-RT). 96–103. https://doi.org/10.1109/DS-RT55542.2022.9932062

[38] Robert Rönngren and Rassul Ayani. 1994. Adaptive checkpointing in Time Warp. In *Proceedings of the Eighth Workshop on Parallel and Distributed Simulation* (Edinburgh, Scotland, United Kingdom) *(PADS '94)*. Association for Computing Machinery, New York, NY, USA, 110–117. https://doi.org/10.1145/182478.182577

[39] Sanaullah, Shamini Koravuna, Ulrich Rückert, and Thorsten Jungeblut. 2023. Exploring spiking neural networks: a comprehensive analysis of mathematical models and applications. *Front Comput Neurosci* 17 (Aug. 2023), 1215824. https://doi.org/10.3389/fncom.2023.1215824

[40] Sumit Bam Shrestha, Longwei Zhu, and Pengfei Sun. 2022. Spikemax: Spike-based Loss Methods for Classification. https://doi.org/10.48550/arXiv.2205.09845 arXiv:2205.09845 [cs.NE]

[41] J. Sjöström and W. Gerstner. 2010. Spike-timing dependent plasticity. *Scholarpedia* 5, 2 (2010), 1362. https://doi.org/10.4249/scholarpedia.1362 revision #184913.