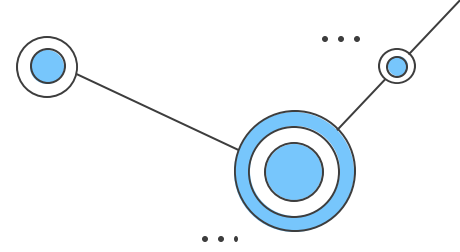


Spiking Neural Networks e Resilienza



Professore Alessandro Savino

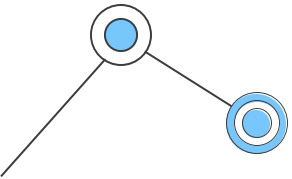
PhD Alessio Carpegna

Studenti:

Vittorio Tabaré s303480

Giorgio Ferraro s296395

Piergiuseppe Siragusa s295491



Indice



Introduzione

Obiettivi, Modello LIF,
Spiking Neural Network



Implementazione rete

LifNeuron, Spikes, Layer,
Network, Builder,



Tecniche di parallelismo

Programmazione concorrente
a livello di Layer



Studio resilienza

Modello per fault injection e
simulazioni parallele



Conclusioni

Descrizione grafici

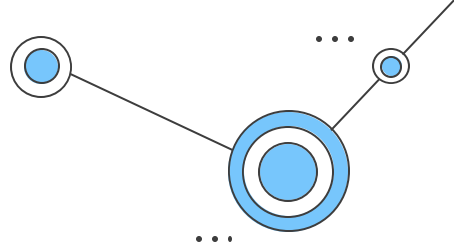


1

Introduzione

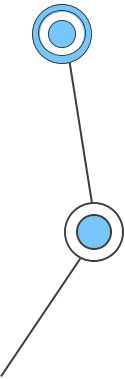
Obiettivi, Modello LIF,
Spiking Neural Network

Obiettivi



Obiettivi del progetto:

1. Sviluppare in Rust una [Spiking Neural Network](#) con un'interfaccia del neurone generica, indipendente dal modello scelto, con parametri interni configurabili oltre che a rendere configurabile il numero di layer e di neuroni in ognuno di essi
2. Implementare il modello [LIF](#) come modello interno al neurone
3. Sfruttare [tecniche di parallelizzazione](#) per ottimizzare le prestazioni della rete
4. Generalizzare il modello per permettere l'iniezione di errore a livello di singolo bit tra [StuckAt0](#), [StuckAt1](#) o [TransientBitFlip](#) su uno specifico componente nella rete



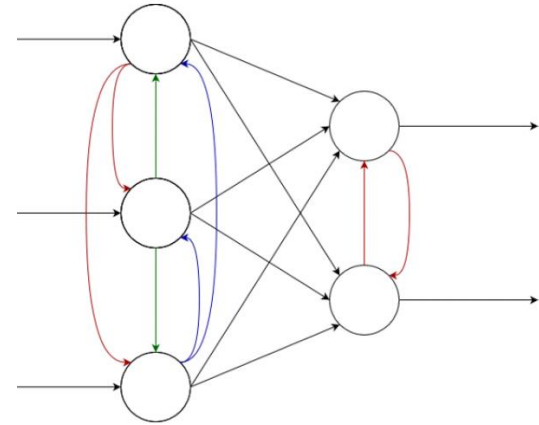
Spiking Neural Network

Una rete neurale spiking o rete neurale a impulso, in sigla SNN, è una rete neurale artificiale che tenta di mimare più precisamente le reti dei **neuroni biologici**. L'informazione è codificata attraverso impulsi elettrici chiamati "**spike**". I neuroni in una SNN inviano impulsi solo quando la somma dei loro input supera una certa soglia.

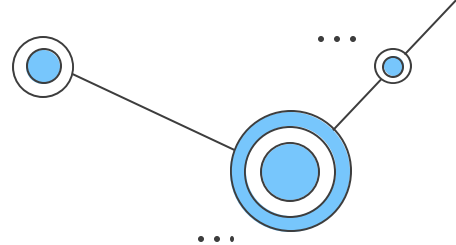
Gli impulsi viaggiano attraverso i collegamenti sinaptici verso altri neuroni, contribuendo così alla computazione complessiva della rete. Questo approccio si basa sulla **rappresentazione temporale** dell'informazione, poiché gli impulsi sono tracce temporali di eventi.

Ogni strato della rete è detto **layer** ed è composto da uno o più neuroni, ciascuno dei quali regola la propria attività in base agli impulsi che riceve in input. La struttura della rete è completamente connessa (**fully-connected**), il che significa che ogni neurone è collegato a tutti gli altri neuroni nello stesso strato (connessioni **intra-layer**) e a tutti i neuroni nei layer adiacenti, sia superiori che inferiori (connessioni **extra-layer**).

Ogni connessione è caratterizzata da un peso, con valori **negativi** per le connessioni **intra-layer** per limitare l'attività complessiva dello strato perché quando un neurone genera un impulso questo va a decrementare il potenziale di membrana di tutti i neuroni nello stesso strato, riducendo la probabilità che questi generino a loro volta un impulso e valori **positivi** per le connessioni **extra-layer**.



Modello LIF



Il modello LIF (Leaky Integrate-and-Fire) è un modello matematico utilizzato per descrivere il comportamento di un neurone in una rete neurale. Questo modello simula il comportamento dei neuroni biologici composto da fasi di integrazione e scarica.

$$V_{mem}(t_s) = V_{rest} + [V_{mem}(t_{s-1}) - V_{rest}] \cdot e^{-\frac{t_s - t_{s-1}}{\tau}} + \sum_{i=0}^N s_i \cdot w_i$$

$$Se V_{mem} > V_{th} \Rightarrow V_{mem} = V_{reset}$$

Dove:

- $V_{mem}(t_s)$ è il potenziale di membrana al tempo t_s
- V_{rest} è il potenziale di membrana a riposo
- $t(s-1)$ rappresenta l'intervallo di tempo relativo all'ultimo spike
- τ è una costante di tempo
- $s(i)$ è lo spike all'istante $t(i)$
- $w(i)$ è il peso associato a uno spike
- V_{th} è il potenziale di soglia
- V_{reset} è il potenziale di reset

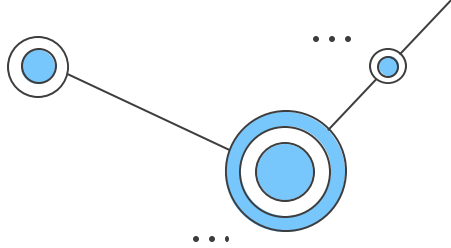


2


Implementazione rete

LifNeuron, Spikes, Layer, Network,
Builder

LifNeuron



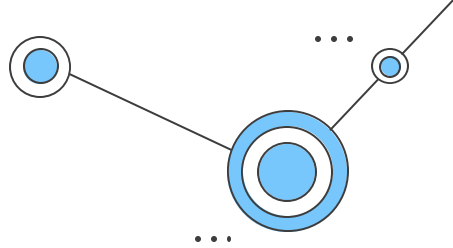
- Il [LifNeuron](#) è stato implementato tramite una struct
- Tutti i [parametri](#) completamente [configurabili](#)
- Al fine di generalizzare l'interfaccia del neurone è stato creato il [tratto Neuron](#), il quale contiene le funzioni generiche necessarie al neurone e la funzione di attivazione



```
pub struct LifNeuron {  
    v_th: f64,  
    v_rest: f64,  
    v_reset: f64,  
    tau: f64,  
    v_mem: f64,  
    ts: u64,  
    dt: f64,  
}
```

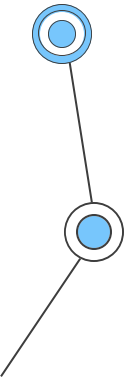
```
pub trait Neuron: Send {  
    fn calculate_v_mem(&mut self, t: u64, extra_sum: f64) -> u8;  
    fn init(&mut self);  
    fn get_tau(&self) -> f64;  
    fn get_v_reset(&self) -> f64;  
    fn get_v_rest(&self) -> f64;  
    fn get_ts(&self) -> u64;  
    fn get_v_mem(&self) -> f64;  
    fn set_v_mem(&mut self, val: f64);  
    fn set_tau(&mut self, val: f64);  
    fn set_v_reset(&mut self, val: f64);  
    fn set_v_rest(&mut self, val: f64);  
    fn set_ts(&mut self, val: u64);  
    fn get_dt(&self) -> f64;  
    fn set_dt(&mut self, val: f64);  
}
```


Spikes

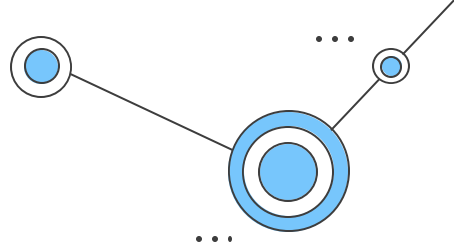


- La rete prende in input una matrice di impulsi binari codificati sotto forma di [SpikeEvent](#)
- Lo [SpikeEvent](#) è una struttura composta da un vettore di u8 e una variabile temporale
- L'istante di tempo definisce l'esatto momento in cui un impulso raggiunge il Layer successivo, mentre il vettore è l'output binario del Layer precedente, il quale sarà processato dal Layer successivo

```
pub struct SpikeEvent {  
    ts: u64,  
    spikes: Vec<u8>,  
}
```

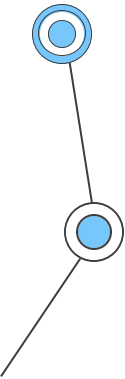


Layer

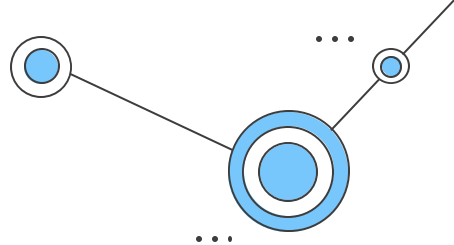


- Il **Layer** rappresenta la struttura che contiene i neuroni di uno stesso livello
- Ogni Layer è composto da uno o più neuroni che regolano il proprio **potenziale** di **membrana** in risposta agli **spikes** in input
- La comunicazione tra neuroni dello stesso Layer avviene attraverso pesi interni (**intra-layer**) , mentre la comunicazione con i neuroni del Layer precedente è garantita dai pesi esterni (**extra-layer**).
- Ogni Layer può manifestare una tipologia di Failure di tipo generico

```
pub struct Layer<N: Neuron + Clone + Send + 'static,  
    R: Configuration + Clone + Send + 'static> {  
    neurons: Vec<N>,  
    weights: Vec<Vec<f64>>,  
    intra_weights: Vec<Vec<f64>>,  
    prev_spikes: Vec<u8>,  
    configuration: R,  
}
```

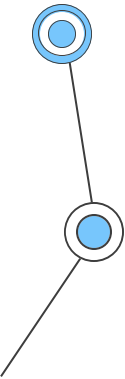


Network



- La struttura Network rappresenta l'intera rete ed è composta da un vettore di [Layer<N, R>](#)
- Al suo interno è presente il fulcro della rete, la funzione [Process](#), la quale riceve una matrice di bit e manda in output una matrice di bit.
- I bit in ingresso e in uscita, prima di essere processati, vengono codificati in [SpikeEvent](#).
- All'interno della Process viene chiamata una funzione interna, la [ProcessEvent](#), la quale crea un thread per ogni layer e li mette in comunicazione tramite dei canali asincroni

```
pub struct SNN<N: Neuron + Clone + 'static,  
  R: Configuration + Clone + Send + 'static> {  
  layers: Vec<Arc<Mutex<Layer<N, R>>>>,  
}
```



Builder

- Permette di creare un'istanza di un oggetto composto tramite una classe separata, che fornisce un'interfaccia per specificare i vari elementi che lo compongono
- È possibile personalizzare: Layer, neuroni, pesi esterni, pesi interni e tipologia di fault
- Implementa i metodi `new`, `add_layer` e `build` che costruiscono tutta la rete dopo aver definito i suoi componenti tramite il pattern builder.
- Il `Builder` è adatto a reti di grandi dimensioni (allocazione in `heap`) di conseguenza i controlli vengono fatti a run-time

```
// Implementation  
pub struct SnnParams<N: Neuron, R: Configuration> {  
    pub input_dimensions: usize,  
    pub neurons: Vec<Vec<N>>,  
    pub extra_weights: Vec<Vec<Vec<f64>>>,  
    pub intra_weights: Vec<Vec<Vec<f64>>>,  
    pub num_layers: usize,  
    pub configuration: Vec<R>,  
}
```

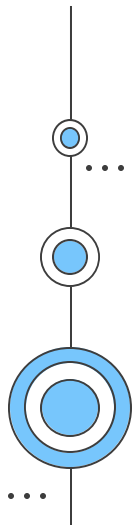
```
pub struct SnnBuilder<N: Neuron,  
    R: Configuration> {  
    params: SnnParams<N, R>,  
}
```



3

Tecniche di parallelismo

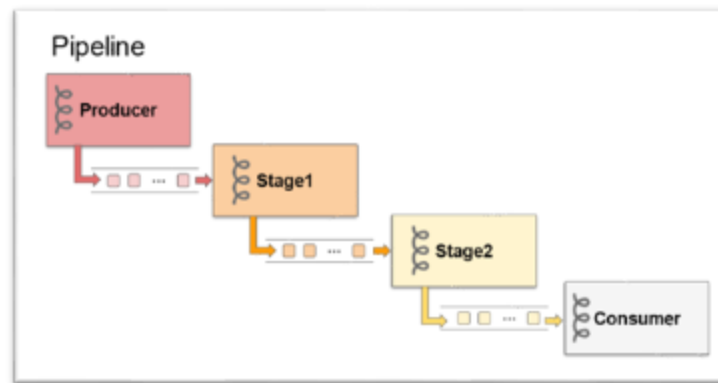
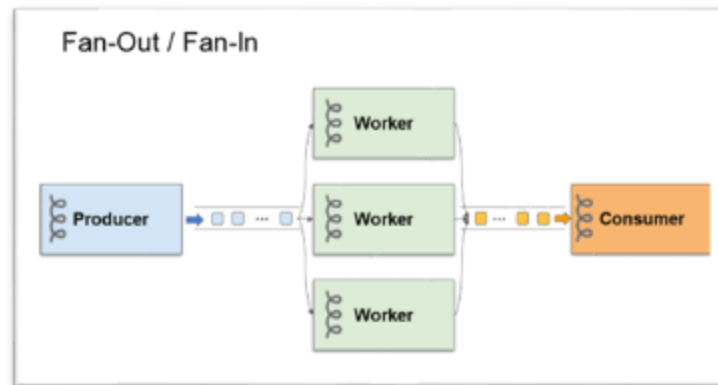
Programmazione concorrente a livello
di Layer



Tecniche di parallelismo

Le soluzioni valutate sono state seguendo i pattern:

- **Fan-out/Fan-in**: distribuire attività a più thread, ciascuno rappresentante un Neurone, e di raccogliere i risultati in un punto specifico. Questo avviene mediante l'utilizzo di una coppia di canali per la distribuzione e la raccolta dei dati tra un Layer e il successivo
 - **Pro**: utile se ogni neurone richiede un elevato carico computazionale
 - **Contro**: utilizzo eccessivo di risorse per implementare un meccanismo di comunicazione tra tutti i thread che rappresentano i neuroni
- ✓ **Pipeline**: crea una serie di fasi di lavorazione, ciascuna delle quali è eseguita da un singolo thread, ciascuno rappresentante un Layer che utilizza un canale per inoltrare gli Spikes al Layer successivo
 - **Pro**: semplicità di implementazione
 - **Contro**: in presenza di Layer con un elevato numero di neuroni, l'utilizzo di canali per inoltrare gli Spikes tra i diversi Layer può generare una latenza di comunicazione significativa

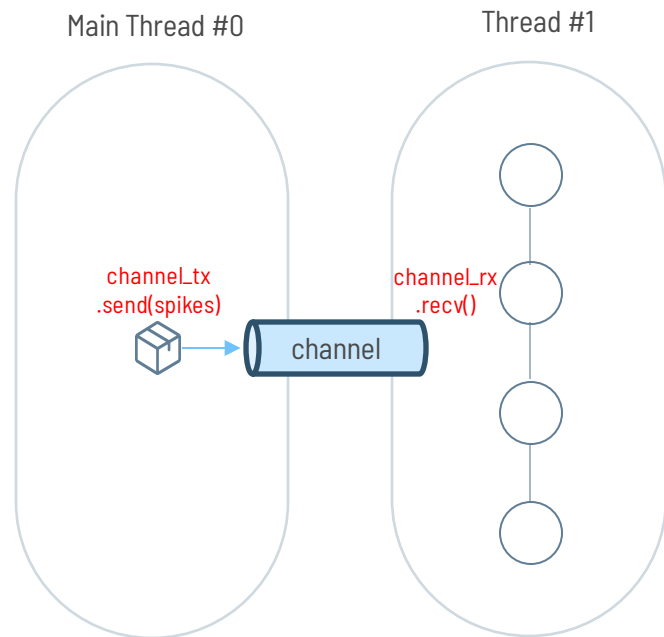


Tecniche di parallelismo

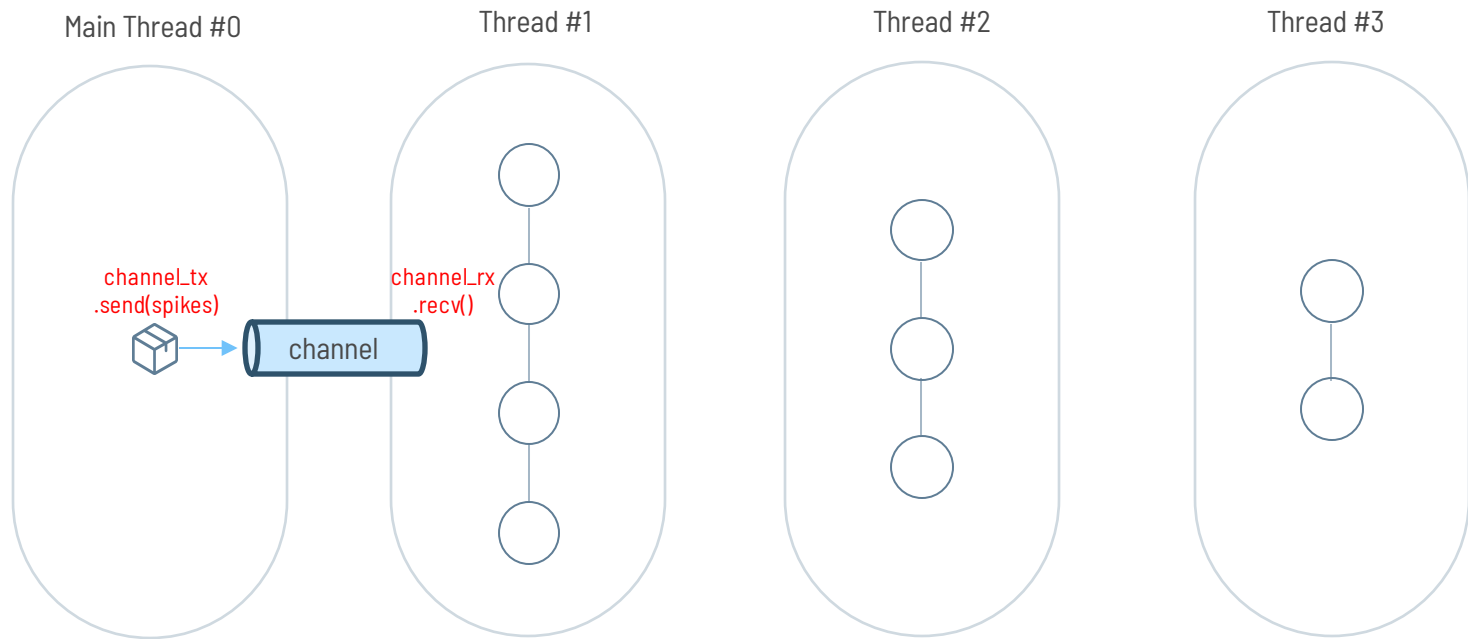
La comunicazione tra thread consecutivi avviene per mezzo di canali asincroni della funzione `sync::mpsc::channel()`. Il main thread si occupa di istanziare per tutti i thread i canali coi quali comunicheranno grazie ai riferimenti al `Sender<SpikeEvent>` e al `Receiver<SpikeEvent>`

La soluzione adottata è il giusto compromesso tra risorse utilizzate e parallelismo, infatti istanziare un thread per neurone comporta un sovraccarico di comunicazione tra thread, poiché ogni neurone richiede una comunicazione frequente con gli altri neuroni. Perciò parallelizzare a **livello di layer** limita questo sovraccarico e rende **l'implementazione più semplice**.

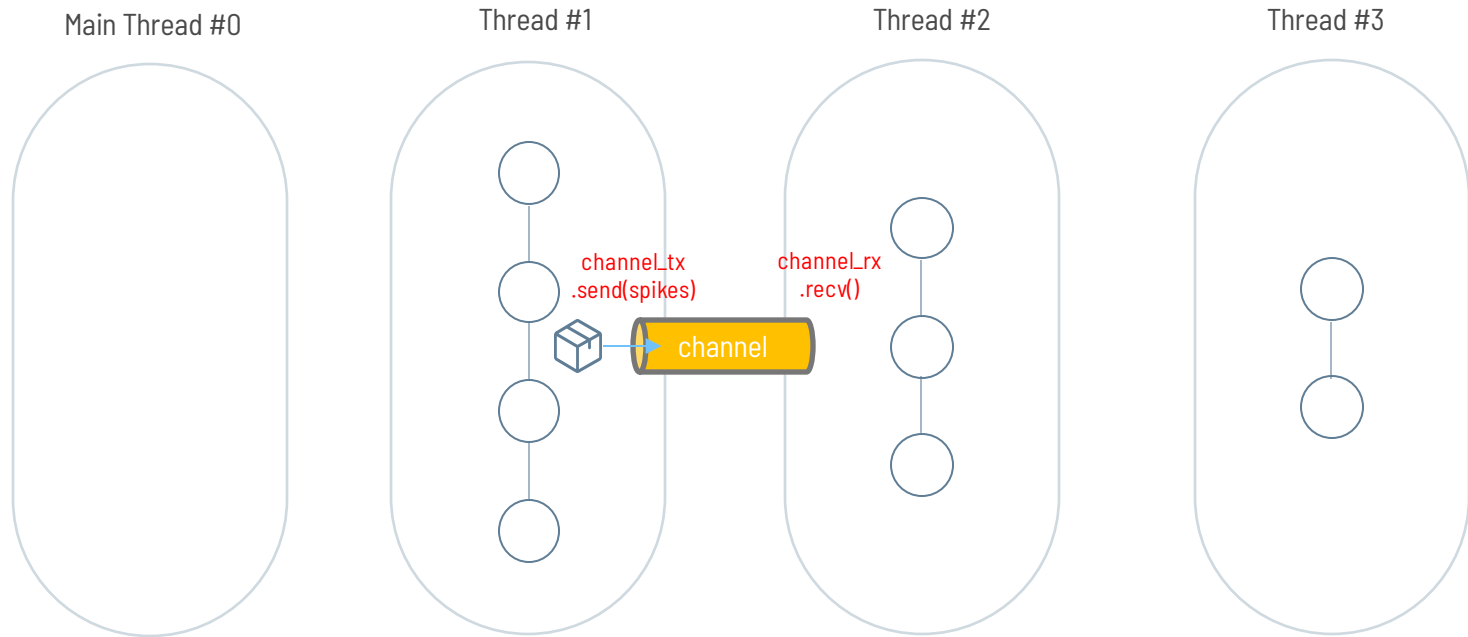
Per condividere dati in maniera sicura il main thread necessita di incapsulare la struct `Layer<N, R>` in un `Arc<Mutex<Layer<N, R>>>` per ottenere un riferimento condivisibile in modo atomico (Arc) per garantire l'accesso esclusivo ai dati (Mutex).



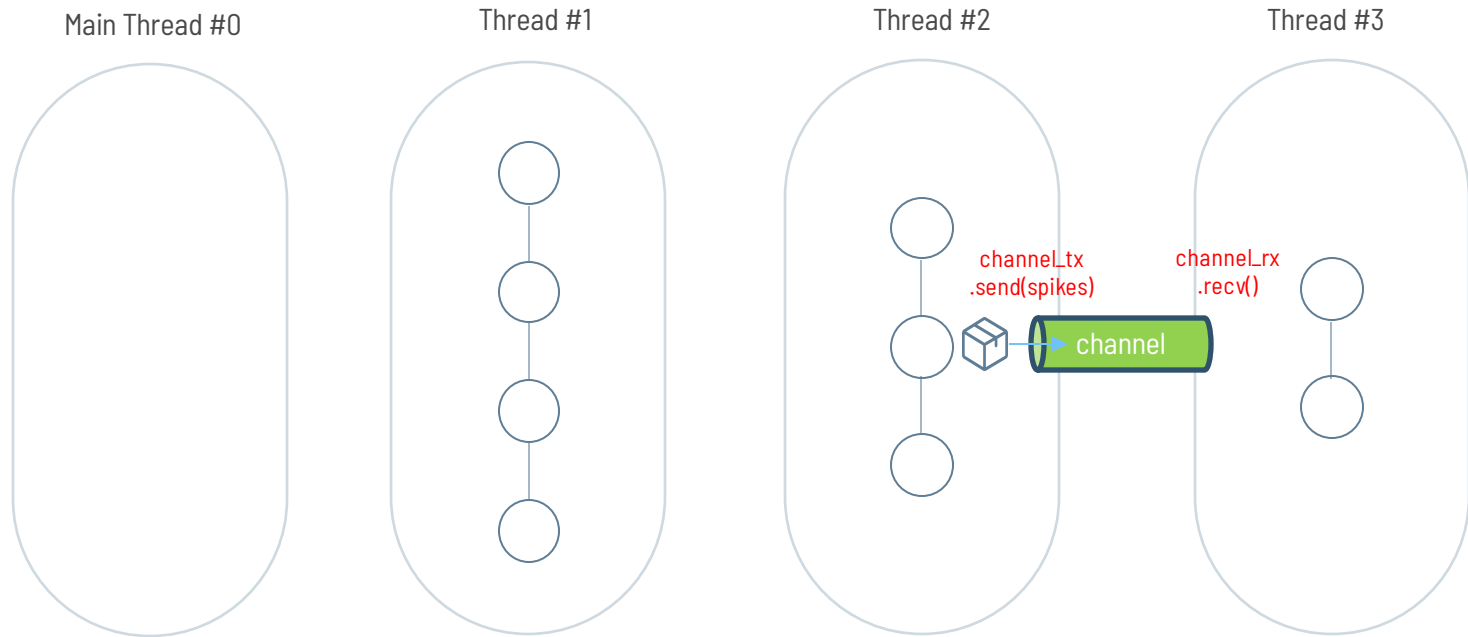
Tecniche di parallelismo



Tecniche di parallelismo



Tecniche di parallelismo

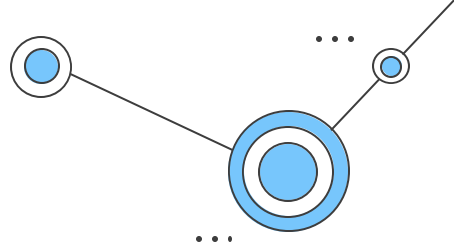


4

Studio resilienza

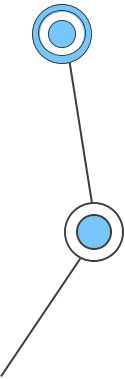
Modello per fault injection
e simulazioni parallele

Implementazione del failure



- Rappresenta un tipo generico di errore da iniettare nella rete
- Completamente configurabile
- È possibile scegliere il componente, il tipo di `Fault`, la posizione del bit del componente scelto, l'indice del neurone e un booleano che utilizziamo per ottimizzare le chiamate alla modifica del bit

```
pub struct Conf {  
    components: Vec<Components>,  
    failure: Failure,  
    index_neuron: usize,  
    done: bool,  
}
```



Failure Design

- L'aggiunta di un guasto alla rete può essere influenzata dalla scelta di collegare il guasto a livello globale (Builder/Network), a un Layer o a un neurone specifico.
- **Neurone:**
 - **Pro:** Accesso istantaneo ai componenti del neurone e possibilità di differenziare il tipo di guasto per singolo neurone
 - **Contro:** difficoltà nel raggiungere i componenti esterni al neurone di livello superiore, complessità superiore della struct LifNeuron e programmazione verbosa per ogni neurone di ogni Layer
- ✓ **Layer:**
 - **Pro:** Ottima visibilità dei componenti da guastare per singolo Layer e possibilità di gestire guasti diversi per Layer
 - **Contro:** Necessità di scrivere il guasto per ogni Layer
- **Builder:**
 - **Pro:** inserito una sola volta per tutta la rete
 - **Contro:** Difficoltà nel trasmettere l'informazione all'interno del singolo Layer e nel raggiungere il singolo neurone. Maggiore complessità del codice con strutture dati aggiuntive e impossibilità di definire guasti diversi per Layer

Generalizzazione e tipologie di fault

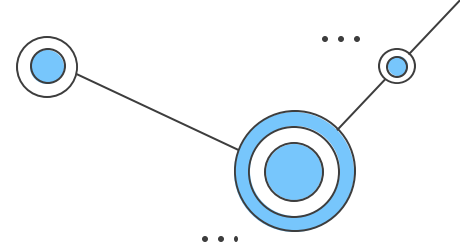
- Per generalizzare l'interfaccia del guasto è stato creato il tratto `Configuration`, che contiene le funzioni generiche necessarie per creare un guasto qualsiasi

```
pub trait Configuration: Send {  
    fn init(&mut self);  
    fn get_vec_components(&self) -> Vec<Components> ;  
    fn get_len_vec_components(&self) -> usize;  
    fn get_failure(&self) -> Failure;  
    fn get_index_neuron(&self) -> usize;  
    fn set_done(&mut self, val : bool);  
    fn get_done(&self) -> bool;  
}
```

- Tipologia di guasti:
 - `StuckAt0`: Forza il bit a 0 per tutta la durata della rete
 - `StuckAt1`: Forza il bit a 1 per tutta la durata della rete
 - `TransientBitFlip`: Inverte il valore di un bit per una singola istanza di tempo

```
pub enum Failure {  
    StuckAt0(StuckAt0),  
    StuckAt1(StuckAt1),  
    TransientBitFlip(TransientBitFlip),  
    None,  
}
```

Componenti guastabili



- **Interni al neurone:** Ogni componente interno al neurone che potenzialmente verrà memorizzato in un registro fisico
- **Pesi esterni:** Pesi di collegamento tra il Layer precedente e il Layer corrispondente
- **Pesi interni:** Pesi di collegamento interni al Layer che collega
- **Spike in ingresso:** Vettore di valori binari che rappresentano lo stato di uscita del Layer precedente

```
pub enum Components {  
    /* List of possible fault components of LifNeuron */  
    VTh, VRest, VReset,  
    Tau, VMem, Ts, Dt,  
  
    /* List of possible fault components of Layers*/  
    Weights, IntraWeights,  
    PrevSpikes,  
  
    None,  
}
```



Generate fault

- La **Generate fault** è un metodo di Layer, il quale permette di applicare la **modifica** al **bit** in base : ...
 - alla **tipologia** di guasto
 - Al **componente** da guastare
 - L'**indice** del neurone
 - L'indice del bit
- Prima di chiamare questo metodo controlliamo che il booleano "**done**" sia falso
- Dopo l'esecuzione del metodo verifichiamo che il componente non sia VMem, Ts o PrevSpikes. Se la condizione è vera viene settato il booleano a true
- Questo impedisce di chiamare il metodo anche quando non cambierebbe il valore del componente, **ottimizzandone** l'esecuzione

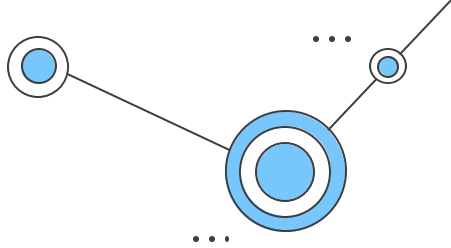
```
if self.configuration.get_done() == false {  
    self.generate_faults();  
    if !self.configuration.get_vec_components().contains(&Components::VMem)  
        && !self.configuration.get_vec_components().contains(&Components::Ts)  
        && !self.configuration.get_vec_components().contains(&Components::PrevSpikes) {  
        self.configuration.set_done(val: true);  
    }  
}
```


Modify bit

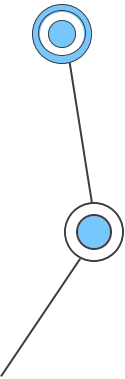
- La modifica del bit è stata implementata con una funzione pubblica esterna alla rete ma visibile a livello globale chiamata `Generate fault`.
- La funzione riceve in ingresso il tipo generico `Failure` e il valore in u64 da modificare, e ritorna il valore col bit modificato.
- Per settare il bit a 0/1 nella posizione richiesta facciamo uso del `Crate "bit"`, il quale implementa il metodo `set_bit` che cambia il valore del parametro modificando il bit, data la posizione e il valore booleano true/false.

```
match failure {
  Failure::StuckAt0(_s: StuckAt0) => {
    val.set_bit(pos: position, val: false);
    val
  }
  Failure::StuckAt1(_s: StuckAt1) => {
    val.set_bit(pos: position, val: true);
    val
  }
  Failure::TransientBitFlip(mut t: TransientBitFlip) => {
    if !t.get_bit_changed() {
      let old_bit: bool = val.bit(pos: position);
      val.set_bit(pos: position, val: !old_bit);
      t.set_bit_changed(val: true);
    }
    val
  }
}
_ => { val }
```

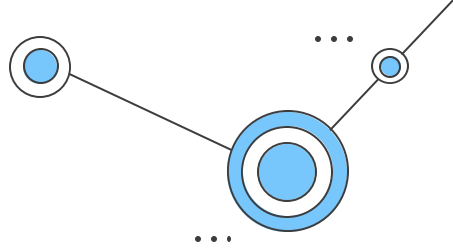
Simulation main.rs



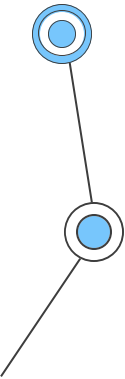
1. La simulazione viene eseguita su 50 immagini di test estratte dal dataset MNIST.
2. Ogni input spike di un'immagine corrisponde a un matrice di 3500 righe e 784 colonne.
3. La **process** riceve l'i-esimo input spike ricavato dall'i-esima immagine e calcola gli output spikes (matrice di 400 righe e 3500 colonne, 400 = numero neuroni)
4. Successivamente si calcola la somma dei 3500 valori del neurone i-esimo, in modo da avere un solo valore per neurone. Si ottiene un vettore di 400 valori per ogni immagine. I vettori così costruiti sono salvati in un file.
5. Alla fine si esegue uno script python per calcolare l'accuratezza dovuta alla classificazione delle 50 immagini.



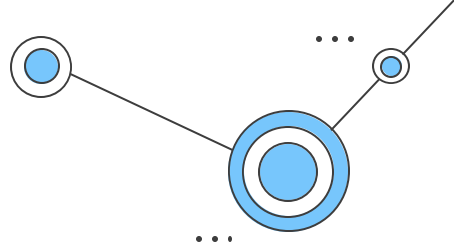
Script python



1. Per ogni label da 0 a 9, si eseguono questi passi.
2. Si calcola un vettore che rappresenta il numero di spike per una specifica label
3. Si aggiorna il vettore di classificazione con la label i -esima, in specifiche posizioni solo se il numero di spike per quella label supera una certa soglia.
4. Quindi dopo 10 iterazioni si ottiene il vettore di classificazione finale che verrà confrontato con il vettore di classificazione desiderato, per capire quanta è alta l'accuratezza.
5. Questa è ottenuta dal rapporto tra il numero di valori uguali tra i 2 vettori e il numero totale di valori.



Parallelizzazione simulazioni

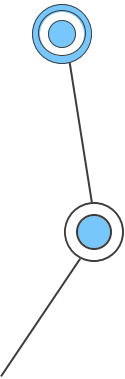


Struttura simulazioni:

- Vengono generati due indici casuali: uno per selezionare un bit tra i 64 bit della variabile che descrive il componente, e uno per selezionare un neurone tra tutti quelli del Layer
- Per eseguire più simulazioni in parallelo, vengono creati due thread per ciascun componente guastabile
- Ogni thread svolge il processamento di una rete e ne calcola l'accuratezza

Questo approccio consente di testare gli **effetti** per tipologie di guasto **differenti** nello **stesso** bit dello **stesso** componente. Per ottimizzazione, come riportato in figura, nel caso in cui il bit sia già a 0, la simulazione per lo StuckAt0 non viene effettuata, analogamente lo stesso avviene per lo StuckAt1

```
let bit : bool = val.bit(position);
match vec_type_fail[f] {
  Failure::StuckAt0(_) if bit == false => {
    println!("Useless simulation StuckAt0");
    continue;
  }
  Failure::StuckAt1(_) if bit == true => {
    println!("Useless simulation StuckAt1");
    continue;
  }
  _ => {}
}
```



5

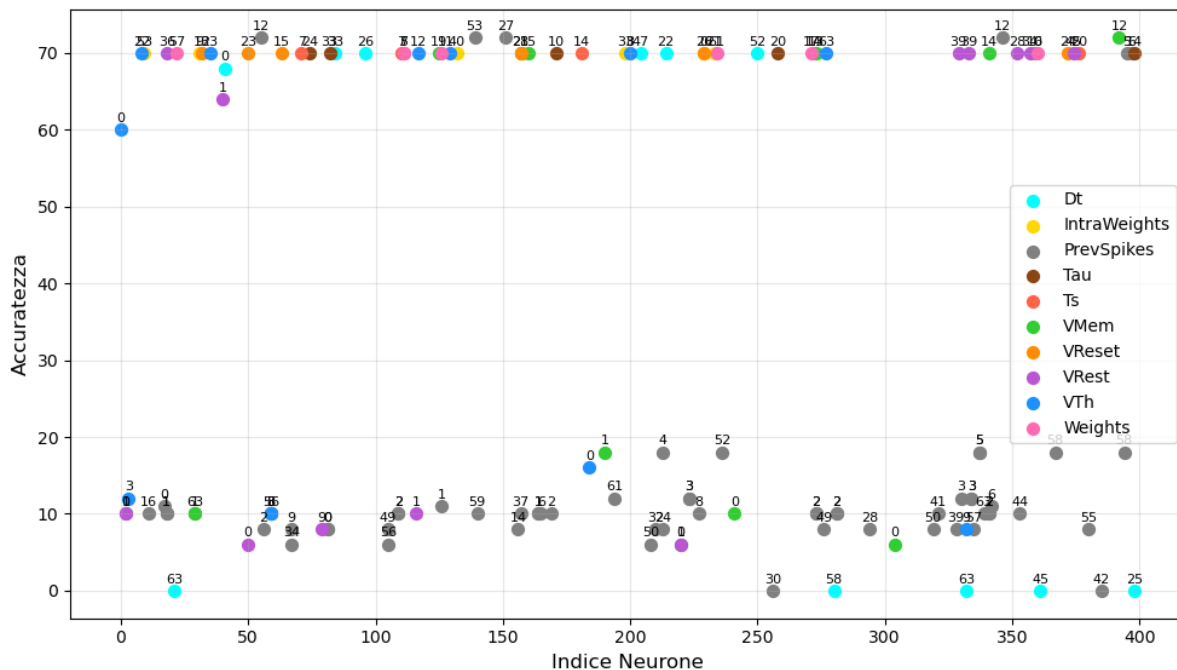
Conclusioni

Descrizione grafici

Risultati simulazioni

A fronte di 650 simulazioni, si è analizzata l'accuratezza in relazione ai seguenti fattori:

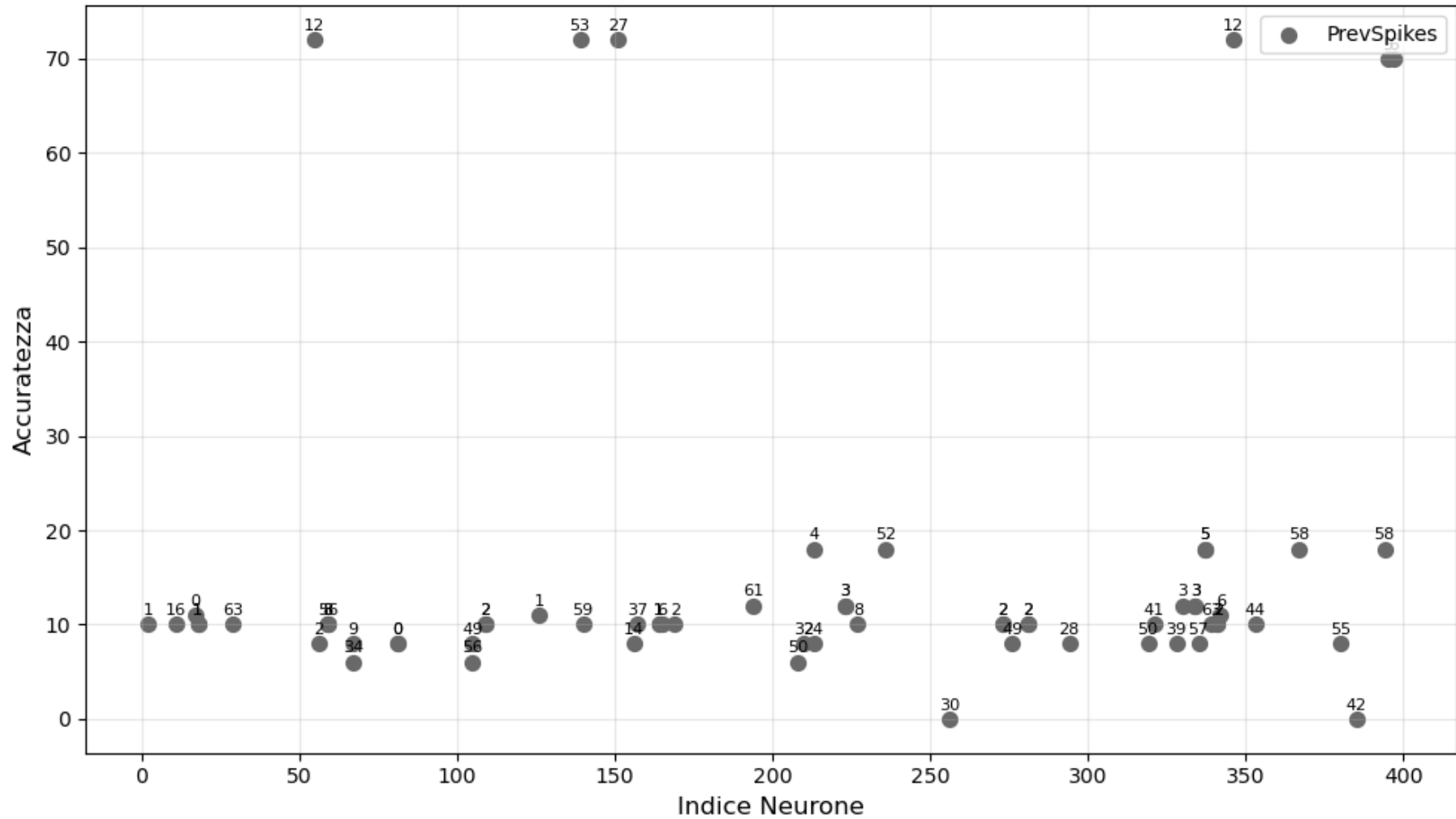
- **Tipo di componente** (VTh, VMem, VReset, VRest, Tau, Ts, Dt, Weights, IntraWeights, PrevSpikes)
- **Tipo di guasto** (StuckAt0, StuckAt1, TransientBitFlip)
- **Indice di bit** (0-64)
- **Indice di neurone** (0-400)



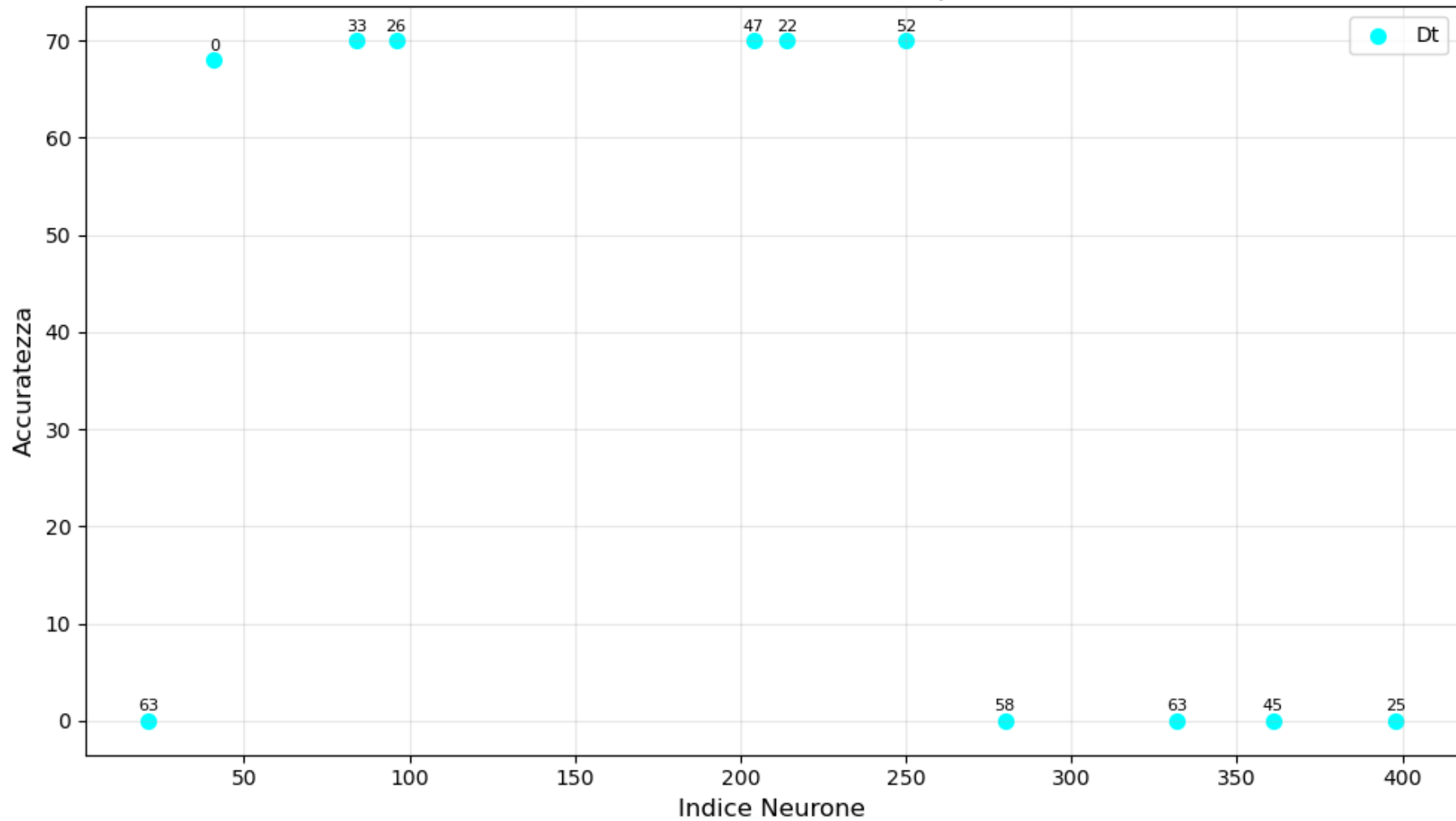
Dalle simulazioni effettuate si può dedurre che:

- La **maggior parte delle reti con guasti** hanno valori di accuratezza pari a quello di una rete senza guasti intorno al 70%
- I **componenti più fragili** sono PrevSpikes, Dt, VRest, Vmem, VTh
- I **bit più fragili** sono i MSB perchè relativi ai bit di segno ed esponente per i floating point f64

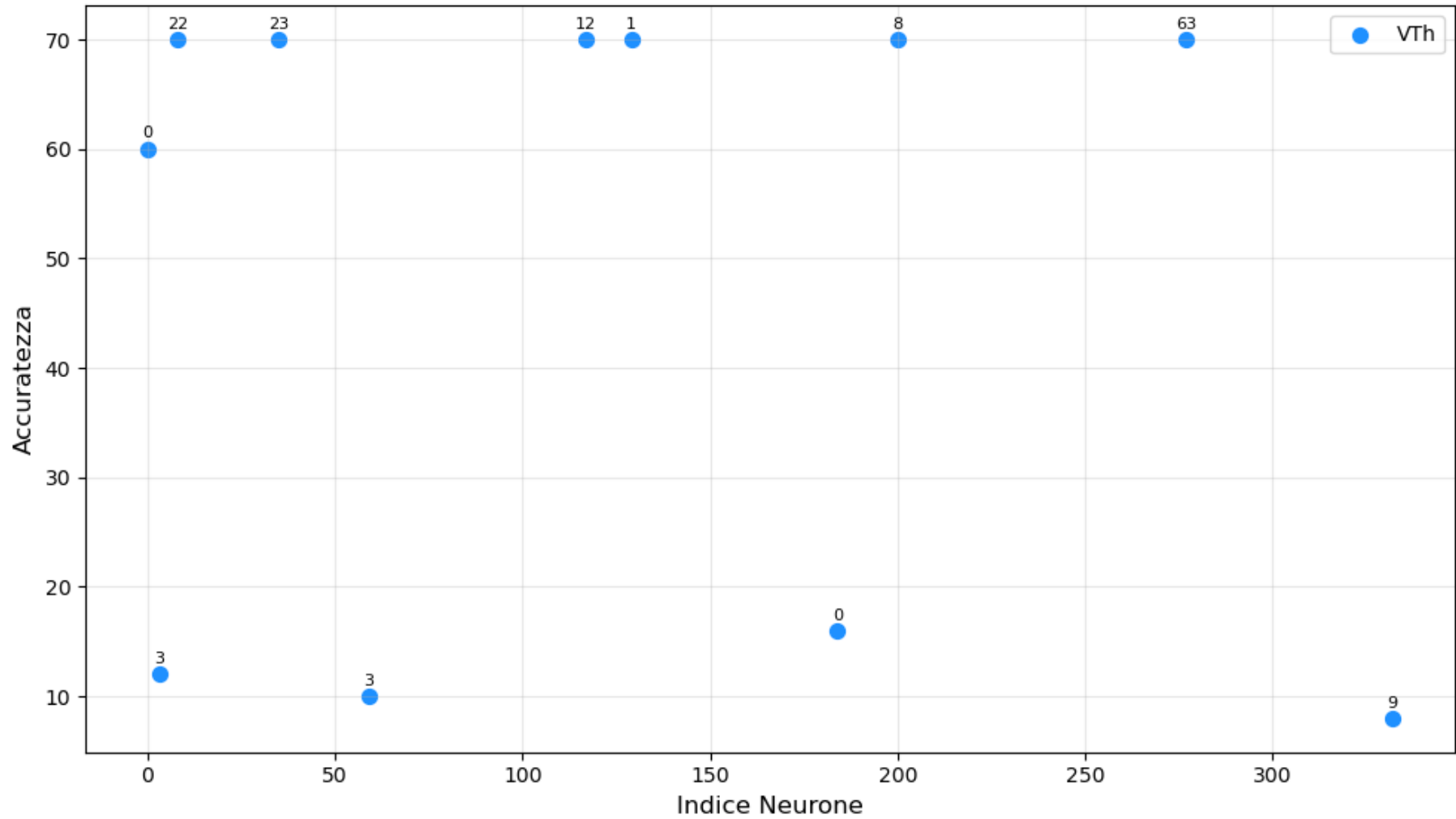
Andamento dell'Accuratezza per PrevSpikes



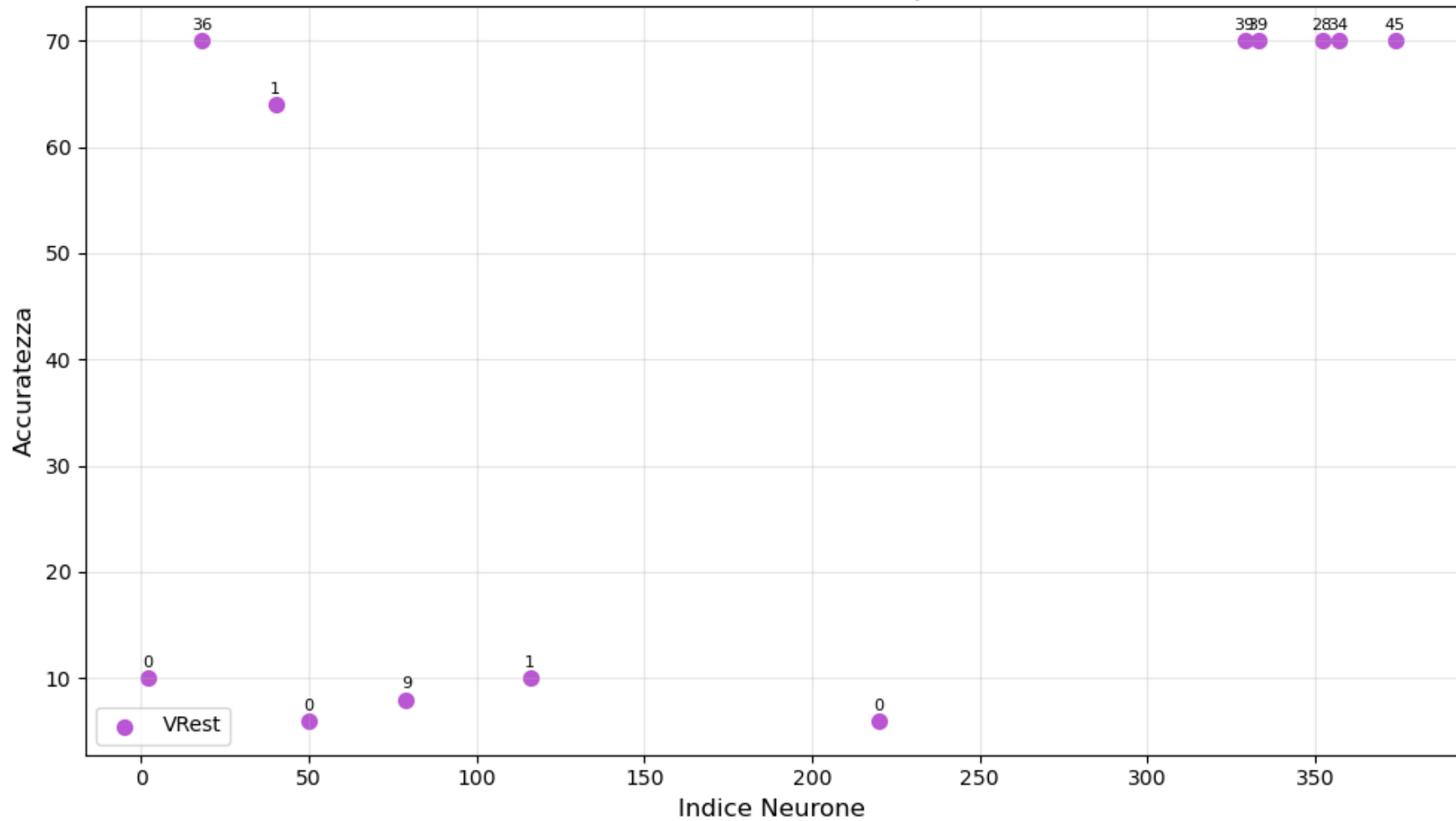
Andamento dell'Accuratezza per Dt



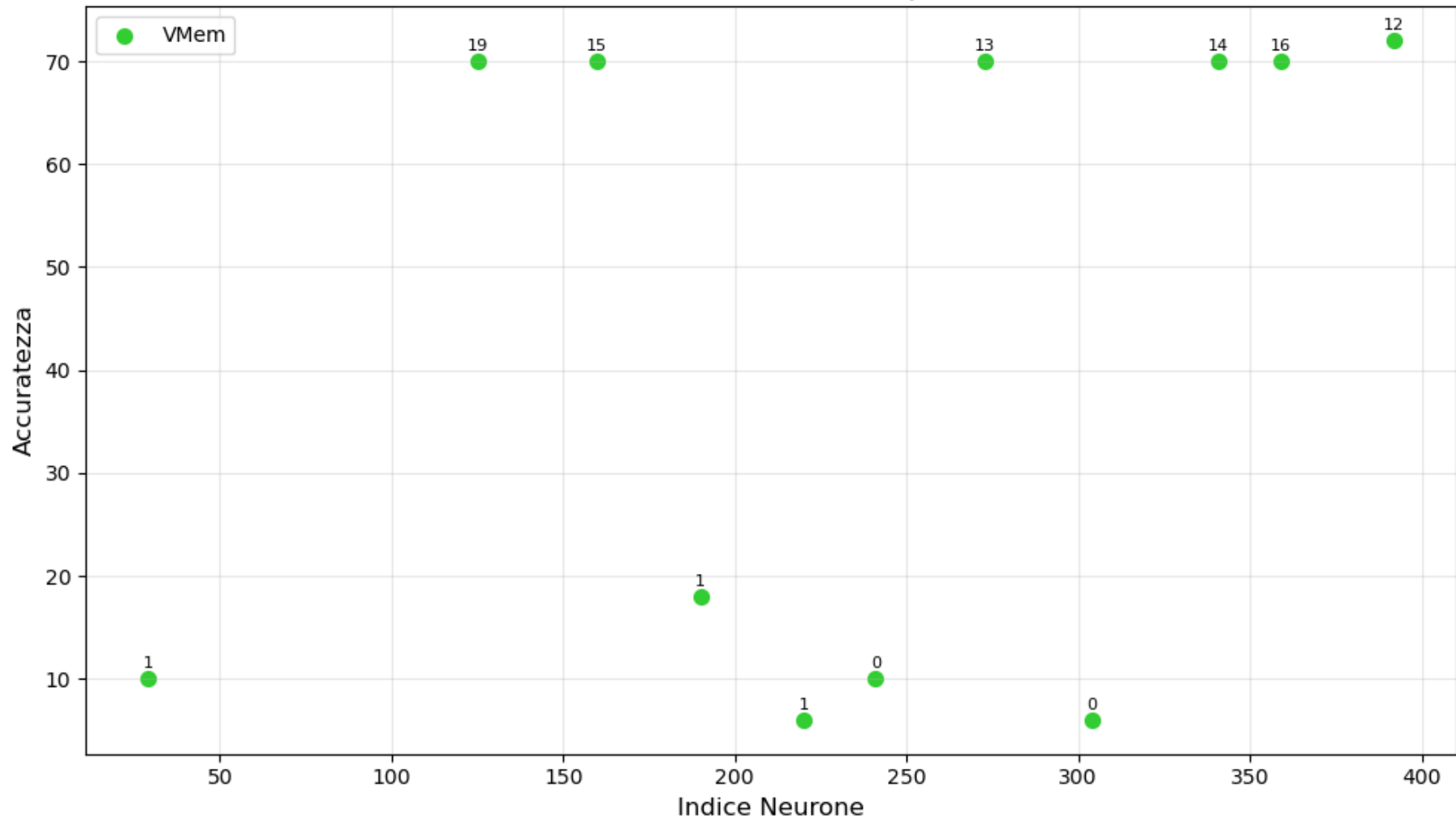
Andamento dell'Accuratezza per VTh



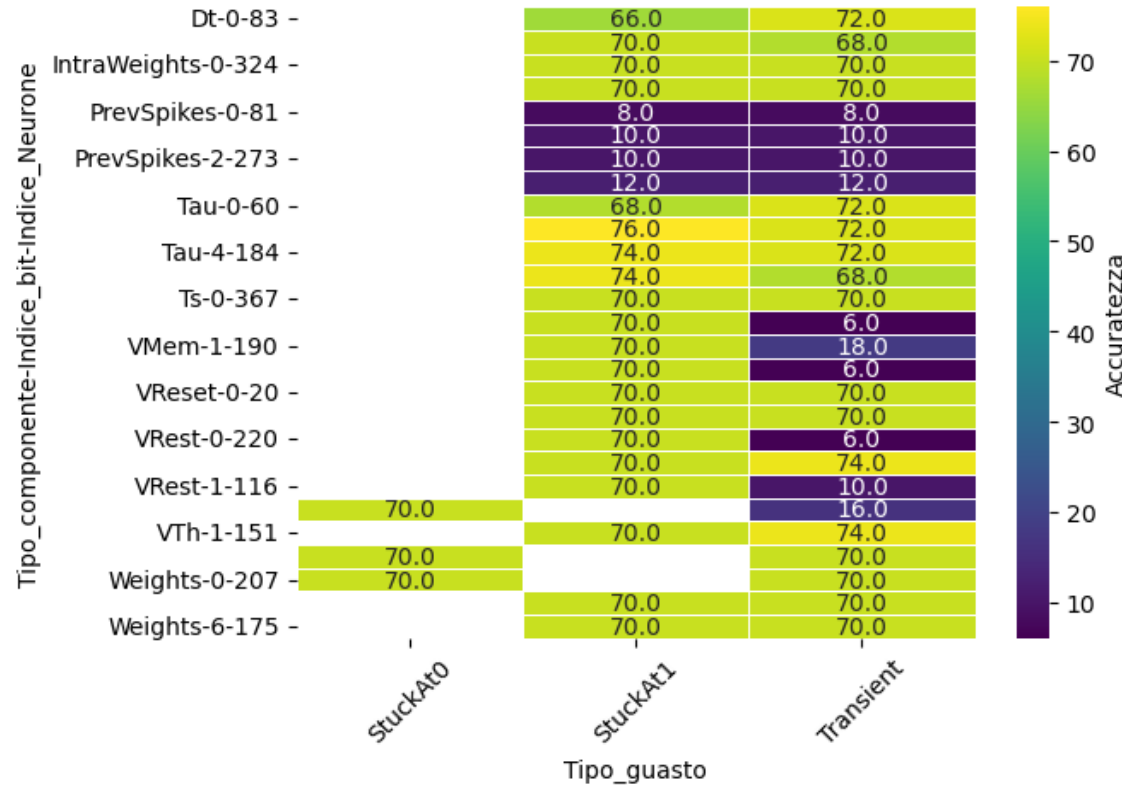
Andamento dell'Accuratezza per VRest



Andamento dell'Accuratezza per VMem



Risultati simulazioni



Dalla heatmap emerge che simulazioni effettuate **nello stesso punto** e **con lo stesso componente** della rete presentano **accuratezze diverse** a seconda del **tipo di guasto**.

Questo indica una **sensibilità differente** della rete a seconda del **tipo di guasto**, aggiungendo perciò un livello di complessità ulteriore alla comprensione del suo comportamento.



Grazie!

Vittorio Tabaré s303480

Giorgio Ferraro s296395

Piergiuseppe Siragusa s295491

...

