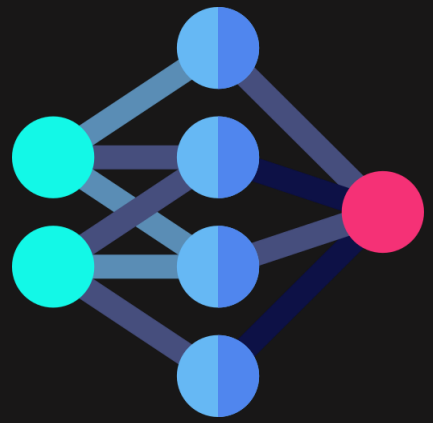
SPIKING NEURAL NETWORK E RESILIENZA



PROGRAMMAZIONE DI SISTEMA – POLITECNICO DI TORINO – A.A. 2022/2023

DAVIDE PALATRONI – matricola 314819

LARA MORESCO – matricola 320153

ANDREA SILLANO – matricola 314771

OVERVIEW

Obiettivi Progetto

SNN

Implementazione Rete

LIF Neuron

Parallelismo

Guasti

Studio Resilienza

Obiettivi progetto

1

LIF Neuron

Creare una rete neurale formata da generici neuroni ed implementare il modello LIF come modello interno, completamente configurabile

2

SNN Configurabile

Permettere all'utente di configurare completamente i parametri della rete, ovvero numero di strati, neuroni per ogni strato e valore dei pesi tra i neuroni

3

Guasti e Resilienza

Permettere l'inserimento nella rete di errori su bit che possano intaccare diversi componenti della rete, e studiarne l'effetto sul segnale di output della rete

4

Parallelizzazione

Utilizzare tecniche di parallelizzazione al fine di migliorare le prestazioni della rete

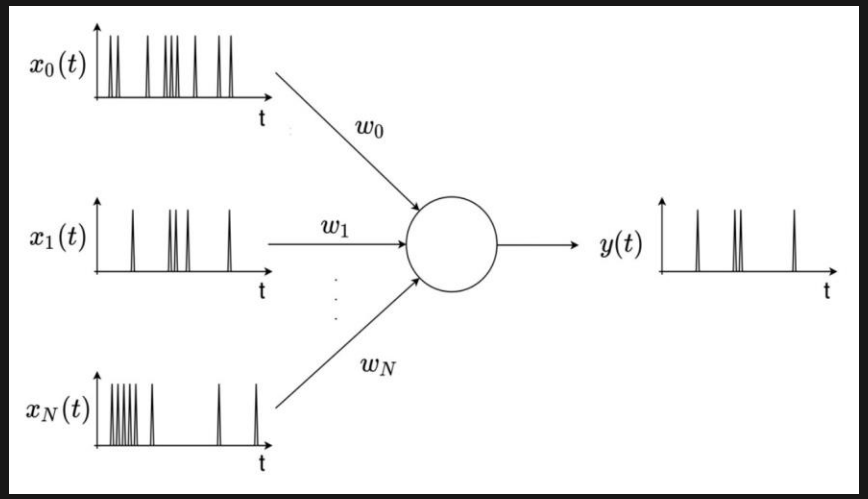
SPIKING NEURAL NETWORK *COS'E'*

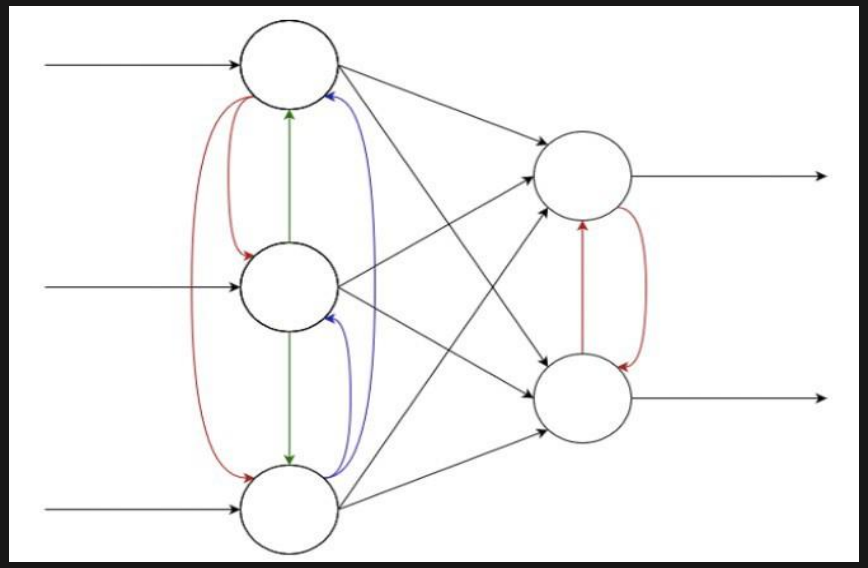
Una **Spiking Neural Network** (**SNN**) è una rete neurale che cerca di emulare in modo più fedele rispetto ai modelli classici un cervello biologico.

L'informazione viene trasmessa tra i neuroni per mezzo di impulsi binari (*spikes*) e il tempo acquisisce un ruolo fondamentale: informazioni complesse vengono codificate tramite sequenze temporali di impulsi.

La rete presenta una struttura **fully-connected,** ciascun neurone è collegato:

- A tutti i neuroni del layer precedente e di quello successivo (*extra-layer link*), collegamenti caratterizzati da pesi **positivi**
- A tutti i neuroni dello stesso layer (*intra-layer link*), collegamenti caratterizzati da pesi **negativi**





LIF NEURON

LEAKY INTEGRATE AND FIRE

Leaky Integrate and Fire (**LIF**) è un modello di neurone descritto come il parallelo di una capacità e una resistenza.
L'evoluzione temporale del potenziale di membrana può essere descritto come segue:

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

```rust
pub struct LIFNeuron{
    /* campi costanti */
    /// potenziale di soglia
    v_th: f64,
    /// potenziale di riposo
    v_rest: f64,
    ///potenziale di reset
    v_reset: f64,
    /// costante di tempo *tau=C·R*
    tau: f64,
    /// intervallo di tempo tra due istanti successivi
    d_t: f64,
    /*campi mutabili*/
    /// potenziale di membrana
    v_mem: f64,
    /// ultimo istante di tempo in cui si è ricevuto un impulso
    t_s: u64,
    /// *eventuale* errore su un bit del potenziale di membrana
    membrane_error:Option<ErrorBit>
}
```

Implementazione del modello LIF nella nostra rete

V_{mem}: potenziale di membrana

t_s: istante di tempo in cui vengono ricevuti impulsi

$V_{mem}(t_{s-1})$: il valore del potenziale di membrana nell'istante t_{s-1}

V_{rest}: *potenziale di riposo*

$t_s - t_{s-1}$: *distanza tra due impulsi*

τ: costante di tempo della membrana, capacità per resistenza

V_{th} : potenziale di soglia

V_{reset} : potenziale di reset

LIF NEURON

EVENT-BASED

Il potenziale di membrana così calcolato viene confrontato con il potenziale di soglia.
Il neurone emette un impulso in uscita solo nel caso il cui questa soglia venga superata, e il potenziale di membrana viene reimpostato al potenziale di reset.

In questo modo, l'elaborazione può essere di tipo **event-based**: il neurone svolge i calcoli solo ad intervalli discreti e nel caso in cui vi sia almeno un impulso in ingresso.

```rust
pub struct Evento {
    ts: u64, /* istante di tempo in cui viene generato l'output */
    spikes: Vec<u8>, /* vettore che contiene tutti gli output */
}
```

GENERALIZZAZIONE DEL MODELLO

TRATTO NEURON

Al fine di **generalizzare** l'interfaccia del neurone è stato creato il tratto *Neuron*, che definisce un metodo per il calcolo del ***potenziale di membrana*** *(update_v_mem)* e un vasta gamma di metodi per simulare un determinato errore all'interno del neurone

```rust
pub trait Neuron: Send{
/// Funzione per calcolare il nuovo potenziale di membrana del neurone;
/// Ritorna un segnale binario 0/1
/// # Argomenti
/// * `t` - Instante di tempo corrente
/// * `intra_weight` - Somma pesata dei segnali provenienti dal layer stesso
/// * `extra_weight` - Somma pesata dei segnali provenienti dal layer precedente
    fn update_v_mem(&mut self, t: u64, intra_weight: f64, extra_weight: f64, adder:  Adder, mult: Multiplier)->u8;
/// Funzione per resettare i parametri del neurone a quelli iniziali
    fn init_neuron(&mut self);
/// Funzione per per settare un errore stuck-at-X sul potenziale di membrana
/// # Argomenti
/// * `error_type` - valore a cui il bit è bloccato *(0/1)*
/// * `position` - posizione del bit bloccato
    fn set_membrane_error(&mut self, error_type:u8, position:u8);
/// Ritorna il valore del potenziale di soglia
    fn get_th(&self) -> f64;
/// Setta il valore del potenziale di soglia
    fn set_th(&mut self, new_th: f64);
/// Ritorna il valore del potenziale di membrana
    fn get_mem(&self) -> f64;
/// Setta il valore del potenziale di membrana
    fn set_mem(&mut self, new_mem: f64);
```

GENERALIZZAZIONE DEL MODELLO

LIF NEURON

In particolare è stato implementato il modello di neurone **LIF**, attraverso la struct LIFNeuron, avente tutti i parametri configurabili

```rust
pub struct LIFNeuron{
    /* campi costanti */
    /// potenziale di soglia
    v_th: f64,
    /// potenziale di riposo
    v_rest: f64,
    ///potenziale di reset
    v_reset: f64,
    /// costante di tempo *tau=C·R*
    tau: f64,
    /// intervallo di tempo tra due istanti successivi
    d_t: f64,
    /*campi mutabili*/
    /// potenziale di membrana
    v_mem: f64,
    /// ultimo istante di tempo in cui si è ricevuto un impulso
    t_s: u64,
    /// *eventuale* errore su un bit del potenziale di membrana
    membrane_error:Option<ErrorBit>
}
```

BUILDER PATTERN

COS'E'

Il **Builder Pattern è** un *design pattern* utilizzato nella programmazione ad oggetti che permette di separare la costruzione di un oggetto complesso dalla sua rappresentazione.

Questo pattern si basa su un *oggetto Builder* a cui l'utente fornisce uno alla volta gli elementi che costituiscono l'oggetto finale, e questo si occupa della creazione di una nuova istanza dell'oggetto senza che l'utente ne debba conoscere l'implementazione interna.

```rust
pub struct SnnBuilder<N: Neuron+Clone+Debug+'static>{
    params: SnnParams<N>,
    adder: Adder,
    mult: Multiplier
}
```

IMPLEMENTAZIONE RETE

La struttura SnnBuilder raccoglie le informazioni relative alla rete neurale, attraverso tre metodi:

- *.add_neurons()*: accetta il vettore di neuroni di un *layer*
- *.add_weights()*: accetta i vettori di pesi esterni con il *layer* precedente
- *.add_intra_weights()*: accetta i vettori di pesi interni al *layer*

Il metodo *.add_layer()* ha il solo compito di separare logicamente i livelli.

Il metodo *.build()* crea la rete SNN vera e propria in base ai parametri, e inietta un eventuale errore su uno dei componenti.

In base al tipo di errore e ai possibili componenti selezionati dall'utente, il metodo privato *.handle_errors()* sceglie casualmente una zona della rete in cui inserire tale errore.

SNNBUILDER

```rust
pub fn build<const INPUT_DIM: usize, const OUTPUT_DIM:usize>(&mut self, components: &Vec<i32>, error_type: i32,
                                                             info_table: &mut InfoTable) -> SNN<N, { INPUT_DIM }, { OUTPUT_DIM }>{
    if self.params.extra_weights.len() != self.params.neurons.len() || self.params.intra_weights.len() != self.params.neurons.len(){
        panic!("Wrong number bewteen layers!")
    }

    let mut layers: Vec<Arc<Mutex<Layer<N>>>> = Vec::new();
    let mut transient: Option<(usize, usize,i32, u8, (i32,i32))>=None;
    self.handle_errors(components,error_type,info_table, transient_error: &mut transient);

    let mut n_iter : IntoIter<Vec<N>> = self.params.neurons.clone().into_iter();
    let mut extra_iter : IntoIter<Vec<Vec<...>>> = self.params.extra_weights.clone().into_iter();
    let mut intra_iter : IntoIter<Vec<Vec<...>>> = self.params.intra_weights.clone().into_iter();

    while let Some(layer : Vec<N> ) = n_iter.next() {
        let new_extra_iter : Vec<Vec<f64>> = extra_iter.next().unwrap();
        let new_intra_iter : Vec<Vec<f64>> = intra_iter.next().unwrap();

        /* creazione di un nuovo layer */
        let new_layer : Layer<N> = Layer::new( neurons: layer, weights: new_extra_iter, intra_weights: new_intra_iter);
        layers.push( value: Arc::new( data: Mutex::new( t: new_layer)));
    }
    SNN::<N, {INPUT_DIM }, { OUTPUT_DIM }>::new(layers, transient_error: transient, adder: self.adder.clone(), multiplier: self.mult.clone())
}
```

```rust
/// Funzione per gestire l'iniezione di errori all'interno della rete
/// # Argomenti
/// * `components` - lista di valori per indicare i possibili componenti in cui iniettare l'errore.
///   Possibili valori:
///     - `0` -> Potenziale di soglia
///     - `1` -> Potenziale di membrana
///     - `2` -> Pesi esterni
///     - `3` -> Pesi interni
///     - `4` -> Adder output
///     - `5` -> Adder input
///     - `6` -> Multiplier output
///     - `7` -> Multiplier input
/// * `error_type` - tipo di errore da iniettare nella rete:
///     - `0` -> Stuck-at-0
///     - `1` -> Stuck-at-1
///     - `2` -> Transient bit-flip
/// * `info_table` - struttura per salvare le informazioni di tutti gli errori inseriti
/// * `transient_error` - variabile opzionale per salvare le informazioni temporanee di un errore transitorio
fn handle_errors(&mut self, components: &Vec<i32>, error_type: i32, info_table: &mut InfoTable, transient_error: &mut Option<(usize, usize, i32, u8,(i32,i32))>){
```

IMPLEMENTAZIONE RETE

SNN OBJECT

La struttura SNN rappresenta la rete vera e propria, caratterizzata da un vettore di *Layer*. La rete possiede inoltre due componenti, *Adder* e *Multiplier*, che rappresentano gli omonimi blocchi elaborativi hardware suscettibili a errori.

Il metodo *.process()* ha il compito di processare gli impulsi in ingresso e ritornare gli impulsi in uscita alla rete. Questi impulsi, da vettori di bit, devono prima essere convertiti a *Eventi*, ovvero strutture che contengono il vettore di impulsi e l'istante di tempo associato.

In caso di presenza di errori transitori (*flip-bit*), essendo dipendenti dalla durata totale dell'input, tale istante viene selezionato casualmente solo dopo aver ricevuto l'ingresso, prima che questo venga processato.

```rust
pub struct SNN<N: Neuron + Clone+'static, const SNN_INPUT_DIM: usize, const SNN_OUTPUT_DIM: usize>{
    layers: Vec<Arc<Mutex<Layer<N>>>>,
    transient_error: Option<(usize, usize,i32, u8,(i32,i32))>,//layer, neuron, component, position, (input1, input2)
    adder: Adder,
    multiplier: Multiplier
}
```

```rust
pub fn process<const SPIKES_DURATION: usize>(&mut self, input_spikes: &[[u8; SNN_INPUT_DIM]; SPIKES_DURATION])
                                             -> [[u8; SNN_OUTPUT_DIM]; SPIKES_DURATION] {
    /* trasformiamo l'input in Eventi */
    let input_events :Vec<Evento> = SNN::<N, SNN_INPUT_DIM, SNN_OUTPUT_DIM>::spikes_to_events( spikes_matrix: input_spikes);
    if self.transient_error.is_some(){
        let (layer :usize , neuron :usize , component :i32 , position :u8 , input_errors :(i32, i32) )=self.transient_error.unwrap();
        let mut rng :ThreadRng =rand::thread_rng();
        let random_instant:u64=rng.gen_range( range: 0..SPIKES_DURATION) as u64;
        /* settiamo l'errore transitorio sul layer corrispontente */
        self.layers[layer].lock().unwrap().set_transient_error(neuron,component,position,random_instant, input_errors);
    }
    let processor = Processor {};
    let mut adder :Adder = self.adder.clone();
    let mut  mult :Multiplier = self.multiplier.clone();
    let output_events :Vec<Evento> = processor.process_events( snn: self,  spikes: input_events, adder, mult);

    /* trasformiamo gli Eventi di output in vettori di segnali, in modo tale che
        il valore di ritorno sia coerente con l'argomento in ingresso della funzione */
    let output_spikes:[[u8; SNN_OUTPUT_DIM]; SPIKES_DURATION]
        = SNN::<N, SNN_INPUT_DIM, SNN_OUTPUT_DIM>::spikes_from_events( eventi: output_events);

    output_spikes
}
```

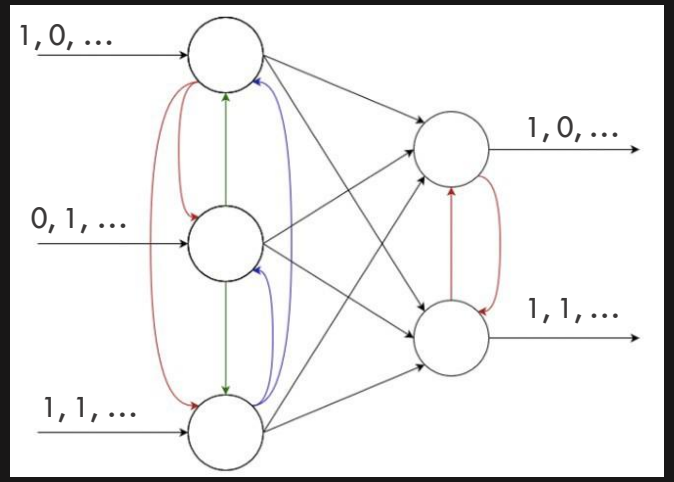
IMPLEMENTAZIONE RETE

INPUT E OUTPUT

La rete riceve in ingresso e ritorna in uscita un treno di vettori di impulsi, ovvero per ogni istante di tempo, l'i-esimo elemento vale:

- **1** se l'i-esimo neurone riceve/emette uno *spike* in quell'istante
- **0** se l'i-esimo neurone non riceve/emette uno *spike* in quell'istante

[[1, 0, 1],
[0, 1, 1],
…]

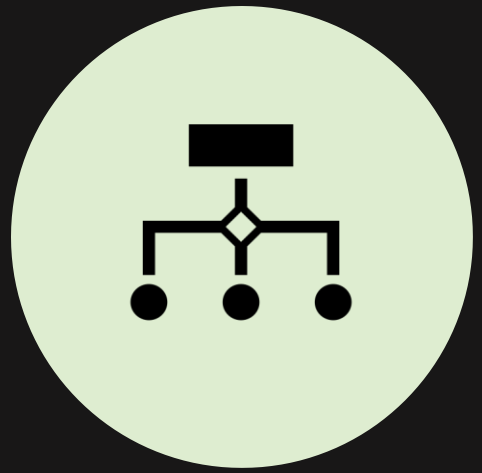


[[1, 1],
[0, 1],
…]

Esempio di spikes in input e output dalla rete e matrici corrispondenti

PARALLELISMO

UN THREAD PER LAYER



LA SOLUZIONE ADOTTATA UTILIZZA **UN SOLO PROCESSO** E **UN THREAD PER OGNI LAYER** (OLTRE AL *MAIN THREAD*)

TRADE-OFF TRA RISORSE UTILIZZATE, SEMPLICITÀ DI IMPLEMENTAZIONE E PARALLELISMO

IMPLEMENTAZIONE RETE

PARALLELIZZAZIONE

Il compito di creare i thread viene affidato al ***Processor***, una struct *unit-like*. Il suo unico metodo *.process_events()* infatti crea un *thread* per ogni *layer*, ed un **canale asincrono** *sync::mpsc::channel()* tra ogni layer e quello successivo, insieme ad uno verso il primo *layer* e uno dall'ultimo *layer*.

In seguito, ciascun *Evento* in input alla rete viene mandato al primo *layer*, e grazie ai *channel* precedentemente creati, questi vengono propagati in essa.

```rust
pub fn process_events<'a, N: Neuron+Clone+'static, S: IntoIterator<Item=&'a mut Arc<Mutex<Layer<N>>>>>
    (self, snn: S, spikes: Vec<Evento>, adder: Adder , mult:  Multiplier) -> Vec<Evento>{
     let mut threads :Vec<JoinHandle<()>>  = Vec::<JoinHandle<()>>::new();
    let (net_input_tx :Sender<Evento> , mut layer_rc :Receiver<Evento> ) = channel::<Evento>();
    for layer_ref :&mut Arc<Mutex<Layer<…>>> in snn {
        let layer_ref :Arc<Mutex<Layer<…>>> = layer_ref.clone();
        let (layer_tx :Sender<Evento> , next_layer_rc :Receiver<Evento> ) = channel::<Evento>();
        let thread :JoinHandle<()>  = thread::spawn(move|| {
            let mut layer :MutexGuard<Layer<N>> = layer_ref.lock().unwrap();
            layer.process(adder, multiplier: mult, layer_input_rc: layer_rc, layer_output_tx: layer_tx);
        });
        threads.push( value: thread);
        layer_rc = next_layer_rc;
    }
    let net_output_rc :Receiver<Evento> = layer_rc;
    for evento :Evento in spikes {
        let instant :u64 = evento.ts;
        net_input_tx.send( t evento) :Result<(), SendError<Evento>>
            .expect( msg: &format!("ERROR: sending spikes event at t={}", instant));
    }
    drop( _x net_input_tx);
    let mut spikes_output :Vec<Evento> = Vec::<Evento>::new();
    while let Ok(spike_event :Evento ) = net_output_rc.recv() {
        spikes_output.push( value: spike_event);
    }
    for thread :JoinHandle<()> in threads {
        thread.join().unwrap();
    }
    spikes_output
```

PROBLEMI DI PARALLELIZZAZIONE

Il problema principale per la parallelizzazione è il passaggio di un *layer* ad un *thread*:
Infatti, seppur ogni *thread* lavori su un singolo *layer*, questi fanno parte del medesimo oggetto SNN, per il quale bisogna garantire ownership e sincronizzazione.
Una prima soluzione semplice è di clonare ciascun *layer* e passare al *thread* un clone, di cui ottiene il possesso.
Questo comporta però un alto consumo di memoria, soprattutto per reti di grandi dimensioni.
Un'altra soluzione è passare al thread un &'static Layer, impossibile senza consumare il *layer* o l'utilizzo di codice *unsafe*.

La soluzione per noi risultata migliore è l'utilizzo di due *smart pointers*:

- Arc<T> permette il possesso condiviso del *layer* tra *main thread* (dove si trova SNN) e *thread* secondario
- Mutex<T> per garantire che un *layer* possa essere modificato da un solo *thread*

Per questo motivo i layer si presentano come *Arc<Mutex<Layer<N>>>*

IMPLEMENTAZIONE RETE

LAYER

Un Layer è caratterizzato da un vettore dei Neuroni che lo costituiscono (*neurons*), i pesi dei collegamenti con il layer precedente (*weights*) e i pesi dei collegamenti interni tra neuroni (*intra_weights*).
Contiene inoltre il vettore degli ultimi impulsi inviati dal layer (*prev_output*), necessari a calcolare l'influenza che ogni neurone ha sugli altri;
È anche presente una opzionale struttura *error* che rappresenta un errore transitorio che si presenterà solamente in un istante specifico in uno dei componenti di quel layer.

```rust
pub struct Layer<N: Neuron+Clone+'static>{
    /// Vettore di neuroni nel layer
    neurons: Vec<N>,
    /// Vettore di vettori di pesi tra ciascun neurone del layer e i neuroni del layer precedente
    weights: Vec<Vec<f64>>,
    /// Vettore di vettori di pesi tra ciascun neurone del layer
    /// e gli altri neuroni del layer stesso
    intra_weights: Vec<Vec<f64>>,
    /// Impulsi di output del layer nell'istante precendete
    prev_output: Vec<u8>,
    /// Eventuale errore transitorio su uno dei componenti del layer
    error: Option<TransientError>
}
```

IMPLEMENTAZIONE RETE

PROCESS

Ogni *layer* è affidato a un *thread*, il quale esegue il suo metodo *.process()*

Il metodo *.process()* attende la ricezione da parte del **Receiver** di un Evento dal layer (*thread*) precedente, fa processare questi impulsi da ogni singolo neurone del layer, salva l'output ottenuto come *prev_output* e lo invia attraverso un **Sender** al layer (*thread*) successivo.

Per ogni neurone, il layer calcola la somma degli impulsi ricevuti pesati in base ai *weights* e la somma degli impulsi dell'istante precedente pesati in base agli *intra_weights*, e fa processare queste somme al neurone, che ritornerà o meno un impulso.

```rust
pub fn process(&mut self, adder: Adder, multiplier:  Multiplier, layer_input_rc: Receiver<Evento>, layer_output_tx: Sender<Evento>){

    /* Prendiamo l'output del layer precedente */
    while let Ok(input_spike : Evento ) = layer_input_rc.recv() {
        let mut local_adder : Adder =  adder;
        let mut local_mult : Multiplier = multiplier;
        let mut at_least_one_spike : bool = false;

        let instant : u64 = input_spike.ts;
        let mut output_spikes : Vec<u8> = Vec::<u8>::with_capacity( capacity: self.neurons.len());
        /* controlliamo che non vi sia un transient bit-flip in questo determinato istante */
        let check_res : Option<(...)> =self.check_transient_error( current_instant: instant,  adder: &mut adder.clone(),  mult: &mut multiplier.clone());
        match check_res{...}

        /* Processiamo l'input per ogni neurone nel layer */
        for (n_index : usize , neuron : &mut N ) in self.neurons.iter_mut().enumerate(){...}
        /* Salvataggio dell'output per il prossimo istante */
        self.prev_output=output_spikes.clone();

        if !at_least_one_spike {
            continue;
        }

        /* Creazione dell'Evento contenente l'output da inviare al prossimo layer */
        let output_spike : Evento = Evento::new( ts: instant, output_spikes);


        /* Mandiamo l'output al prossimo layer */
        layer_output_tx.send( t: output_spike) : Result<(), SendError<Evento>>
            .expect( msg: &format!("ERROR: sending spike event at t={}", instant));
    }
```

GUASTI

I guasti possibili sono a singolo bit, ovvero prevedono che solo un bit tra tutti quelli passibili di guasto si "rompa".

I componenti affetti da tali guasti possono essere selezionati dall'utente tra i registri di memoria che contengono i dati riferiti a membrana, soglia, pesi intra ed extra layer oppure tra i componenti elaborativi come adder e multiplier.
Nel caso di tali componenti è possibile inserire tale errore sia negli ingressi del componente che nell'uscita.

1

Stuck-at-0

Il bit rimane fisso a 0, anche se richiesto il contrario

2

Stuck-at-1

Il bit rimane fisso a 0, anche se richiesto il contrario

3

Bit-Flip

Il valore del bit viene invertito

GUASTI

La funzione *embed_error() permette di gestire le 3 tipologie di errore, trasformando il* valore su cui effettuare il guasto in bit e successivamente al computo dell'errore ritornando il nuovo valore in *f64*

```rust
fn embed_error(variable:f64, error:ErrorType, position: u8)->f64{
    let mut bit_value:u64=variable.to_bits();
    bit_value=match error {
        ErrorType::None => bit_value,
        Stuck0 => unset_bit(bit_value, position),
        Stuck1 => set_bit(bit_value, position),
        Flip => flip_bit(bit_value, position),
    };

    f64::from_bits( v: bit_value)
}
```

Implementazione di embed_error()

GUASTI

Le funzioni che permettono l'iniezione di un guasto sono le seguenti, ognuna riceve due parametri: il valore in bits sul quale applicare il guasto e la posizione del bit che deve essere influenzato.

```rust
/**
Setta a 0 il bit in posizione *position* di *value*
 */
fn unset_bit(value:u64, position:u8)->u64{
    /* maschera del tipo 1110111 */
    let bit_mask :u64 =!(1u64<<position);
    value & bit_mask
}
```

Stuck-at-0

```rust
/**
Setta a 1 il bit in posizione *position* di *value*
 */
fn set_bit(value:u64, position:u8)->u64{
    /* maschera del tipo 0001000 */
    let bit_mask :u64 =1u64<<position;
    value | bit_mask
}
```

Stuck-at-1

```rust
/**
Inverte il bit in posizione *position* di *value*
 */
fn flip_bit(value:u64, position:u8)->u64{
    /* maschera del tipo 0001000 */
    let bit_mask :u64 =1u64<<position;
    value ^ bit_mask
}
```

Flip-bit

Guasti

COMPONENTI

I componenti sui quali è possibile inserire un guasto sono i seguenti:

Soglia

Il potenziale di soglia del neurone

Membrana

Il potenziale di membrana del neurone

Intra-Pesi

I pesi tra neuroni appartenenti allo stesso layer

Extra-Pesi

I pesi tra neuroni appartenenti a layer differenti

Adder Output

L'output del sommatore

Adder Input

L'input del sommatore

Multiplier Output

L'output del moltiplicatore

Multiplier Input

L'input del moltiplicatore

GUASTI

GESTIONE DEI CASI DI ERRORE

In base al tipo di errore e al componente affetto da tale, si possono distinguere tre casi generali con distinte gestioni:

- ***Stuck-at-X su registri read-only:*** essendo questi parametri costanti e mai sovrascritti (*e.g. potenziali di soglia, pesi...*), è sufficiente che l'errore venga imposto una volta solamente, all'inizio, modificando il valore del parametro affetto.

- ***Stuck-at-X su valori variabili***: ogni volta che questi valori (*e.g. potenziale di membrana, ...*) vengono sovrascritti bisogna forzare ad X il bit 'rotto'. Per questo motivo questi componenti contengono parametri aggiuntivi per il 'salvataggio' dell'errore, che sarà imposto ogni volta che la variabile deve essere utilizzata.

- ***Transient bit-flip***: indipendentemente dal componente, questo errore deve essere imposto una volta solamente. L'istante in cui si scatena, per essere rilevante, può essere definito solo conoscendo la durata complessiva dell'utilizzo della rete, quindi solo al momento della ricezione dell'input e viene 'salvato' nel *layer* che ne verrà affetto. Successivamente è il *layer* stesso a imporre l'errore nell'**istante opportuno**: nel caso di pesi e neuroni, semplicemente sovrascrivendo i valori, nel caso di sommatore e moltiplicatore creando copie fallate utilizzate solamente in quell'istante.

STUDIO RESILIENZA

CONFIGURAZIONE RETE

Lo studio della resilienza parte con la configurazione della SNN.

Si configurano per ogni layer: Pesi Extra-layer, Neuroni, Pesi Intra-Layer.

E' possibile per l'utente scegliere in numero arbitrario il numero di layer, il numero di neuroni per layer ed il valore dei pesi.

Successivamente si imposta il treno di spikes che verrà inviato come input ai neuroni del primo layer della rete:

```
let input : [[u8; 3]; 6] = [[0,1,1], [0,0,1], [1,1,1], [1,0,0], [0,0,1], [0,1,0]];
```

```
let mut binding : SnnBuilder<LIFNeuron> = SnnBuilder::new();
let builder : &mut SnnBuilder<LIFNeuron> = binding.add_layer().add_weight( weights: [
    [0.1, 0.2],
    [0.3, 0.4],
    [0.5, 0.6]
]).add_neurons( neurons: [
                    LIFNeuron::new( v_th: 0.03,  v_rest: 0.05,  v_reset: 0.1,  tau: 1.0,  d_t: 1.0),
                    LIFNeuron::new( v_th: 0.05,  v_rest: 0.05,  v_reset: 0.1,  tau: 1.0,  d_t: 1.0),
                    LIFNeuron::new( v_th: 0.09,  v_rest: 0.05,  v_reset: 0.1,  tau: 1.0,  d_t: 1.0),
]).add_intra_weights( intra_weights: [
    [0.0, -0.25, -0.3],
    [-0.10, 0.0, -0.3],
    [-0.1, -0.3,  0.0]
]) : &mut SnnBuilder<LIFNeuron>
.add_layer() : &mut SnnBuilder<LIFNeuron>
    .add_weight( weights: [
        [0.1, 0.2, 0.3],
        [0.4, 0.5, 0.6]
    ]).add_neurons( neurons: [
    LIFNeuron::new( v_th: 0.07,  v_rest: 0.04,  v_reset: 0.4,  tau: 1.0,  d_t: 1.0),
    LIFNeuron::new( v_th: 0.3,  v_rest: 0.01,  v_reset: 0.4,  tau: 1.0,  d_t: 1.0),
]).add_intra_weights( intra_weights: [
    [0.0, -0.25],
    [-0.10, 0.0]
]).add_layer().add_weight( weights: [
    [0.1, 0.2],
    [0.3, 0.4],
    [0.5, 0.6]
]).add_neurons( neurons: [
    LIFNeuron::new( v_th: 0.03,  v_rest: 0.01,  v_reset: 0.1,  tau: 1.0,  d_t: 1.0),
    LIFNeuron::new( v_th: 0.05,  v_rest: 0.03,  v_reset: 0.2,  tau: 1.0,  d_t: 1.0),
    LIFNeuron::new( v_th: 0.09,  v_rest: 0.06,  v_reset: 0.4,  tau: 1.0,  d_t: 1.0),
]).add_intra_weights( intra_weights: [
    [0.0, -0.25, -0.3],
    [-0.10, 0.0, -0.3],
    [-0.1, -0.3,  0.0]
]).add_layer() : &mut SnnBuilder<LIFNeuron>
    .add_weight( weights: [
        [0.1, 0.2, 0.3],
        [0.4, 0.5, 0.6]
    ]).add_neurons( neurons: [
    LIFNeuron::new( v_th: 0.07,  v_rest: 0.01,  v_reset: 0.2,  tau: 1.0,  d_t: 1.0),
    LIFNeuron::new( v_th: 0.03,  v_rest: 0.08,  v_reset: 0.3,  tau: 1.0,  d_t: 1.0),
]).add_intra_weights( intra_weights: [
    [0.0, -0.25],
    [-0.10, 0.0]
]);
```

Esempio di Configurazione

STUDIO RESILIENZA

MENU

Una volta eseguita la configurazione è possibile eseguire il programma.

Verranno presentati tre menù in sequenza dove l'utente potrà definire la configurazione per lo studio della resilienza.

Terminata la procedura di configurazione verrà avviata l'esecuzione della rete.

```
#########################################################
#                                                       #
#        Components:                                    #
#         0 => Threshold                                #
#         1 => Membrane                                 #
#         2 => Extra Weights                            #
#         3 => Intra Weights                            #
#         4 => Adder Output                             #
#         5 => Adder Input                              #
#         6 => Multiplier Output                        #
#         7 => Multiplier Input                         #
#         8 => All Components                           #
#                                                       #
#########################################################
```

Menù Componenti

```
#########################################################
#                                                       #
#       Error Type:                                     #
#         0 => Stuck-at-0                               #
#         1 => Stuck-at-1                               #
#         2 => Flip-bit                                 #
#                                                       #
#########################################################
```

Menù Errori

```
Insert number to select the number of faults!
>
```

Selezione Numero Errori

STUDIO RESILIENZA

OUTPUT

Al termine dell'esecuzione sarà possibile visionare i risultati della configurazione, comparati con una rete priva di guasti, in tre tabelle stampate sia sul terminale sia nel file *report.txt*

```
#########################################################
#                                                       #
#       Configuration:                                  #
#                                                       #
#         Components:                                   #
#             -Threshold                                #
#             -Membrane                                 #
#             -Adder Output                             #
#             -Adder Input                              #
#                                                       #
#         Error Type:                                   #
#             Stuck-At-1                                #
#                                                       #
#         Number of Faults:                             #
#             10                                        #
#                                                       #
#########################################################
```

Riassunto Configurazione

Layer	Neuron	Component	Bit	Error	Impact On Accuracy
/	/	Adder Input - (1,0)	41	Stack-At-1	0%
3	0	Threshold	6	Stack-At-1	0%
1	1	Threshold	20	Stack-At-1	0%
/	/	Adder Input - (0,1)	22	Stack-At-1	0%
/	/	Adder Output	51	Stack-At-1	0%
/	/	Adder Output	35	Stack-At-1	0%
/	/	Adder Output	29	Stack-At-1	0%
/	/	Adder Output	45	Stack-At-1	0%
1	1	Threshold	63	Stack-At-1	16.66%
2	1	Threshold	13	Stack-At-1	0%

Tabella di output

STUDIO RESILIENZA

OUTPUT

Al termine dell'esecuzione sarà possibile visionare i risultati della configurazione, comparati con una rete priva di guasti, in tre tabelle stampate sia sul terminale sia nel file *report.txt*

```
################################################################################
#                                MAX IMPACT INFO                               #
################################################################################
+-------+--------+-----------+-----+------------+--------------------+
| Layer | Neuron | Component | Bit | Error      | Impact On Accuracy |
+-------+--------+-----------+-----+------------+--------------------+
|     1 |      1 | Threshold |  63 | Stack-At-1 |             16.66% |
+-------+--------+-----------+-----+------------+--------------------+
```

Tabella con i componenti ad impatto massimo

```
################################################################################
#                                   SUMMARY                                    #
################################################################################
+--------------------------+-----------------------------+----------------------+----------------------------+
| Total Affected Inferences | Total Affected Inferences % | Max Impact On Accuracy | Average Impact On Accuracy |
+--------------------------+-----------------------------+----------------------+----------------------------+
|                        1 |                        10% |               16.66% |                     16.66% |
+--------------------------+-----------------------------+----------------------+----------------------------+
```

Tabella Riassuntiva

CONCLUSIONI

- Il linguaggio Rust permette in maniera efficiente la creazione di reti multi-thread con una discreta velocità di esecuzione.
- Grazie alle caratteristiche del linguaggio, è stato possibile simulare il comportamento di un guasto su un componente hardware reale con discreta accuratezza.
- Attraverso l'utilizzo delle tecniche di parallelizzazione e sincronizzazione (*fearless concurrency*), è stato possibile realizzare uno strumento adatto per lo studio della resilienza di una SNN in grado di rappresentare in maniera ottimale il comportamento nella realtà.

RIFERIMENTI

- Carpegna, A. (2021, Ottobre). *Design of an hardware accelerator for a Spiking Neural Network*

Grazie per l'attenzione