

SPIKING NEURAL NETWORK RUST LIBRARY

PROGRAMMAZIONE DI SISTEMA – POLITECNICO DI TORINO – A.A. 2021/2022

- FRANCESCO ROSATI - S296247
- GIUSEPPE LAZZARA - S302064
- MARIO MASTRANDREA - S302219

SOMMARIO

Obiettivi del
progetto

Spiking Neural
Network

Modello LIF

Event-based
approach

Generalizzazione
del modello

Builder pattern

Parallelismo

Implementazione
della rete

Esempi di utilizzo

Test di
accuratezza

OBIETTIVI DEL PROGETTO

- Implementare una **libreria Rust** in grado di simulare il comportamento di una *Spiking Neural Network*, con l'utilizzo di iperparametri già allenati
- Sfruttare tecniche di **parallelizzazione** per ottimizzare le prestazioni della rete
- Fare in modo che il modello di neurone e i parametri della rete siano **completamente configurabili**
- Implementare il **modello LIF** come modello interno al **neurone**

SPIKING NEURAL NETWORK

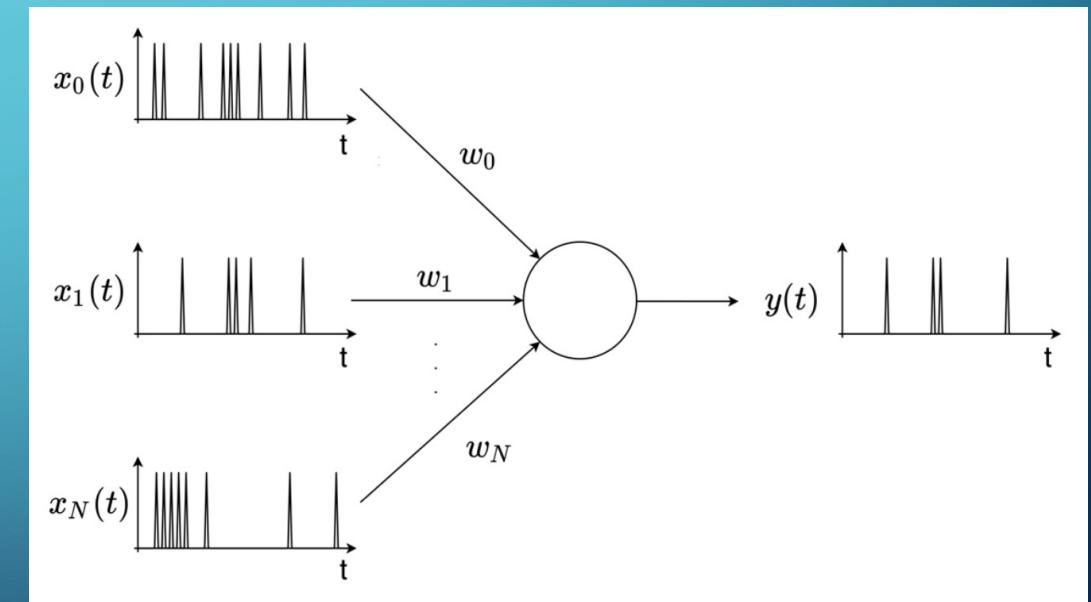
COS'È

- Una *Spiking Neural Network (SNN)* è una rete neurale in cui le informazioni si trasmettono tramite **impulsi binari (spikes)**
- Lo scopo è quello di emulare il comportamento di un **cervello biologico** in maniera più accurata rispetto alle classiche *Artificial Neural Networks*

SPIKING NEURAL NETWORK

COME FUNZIONA

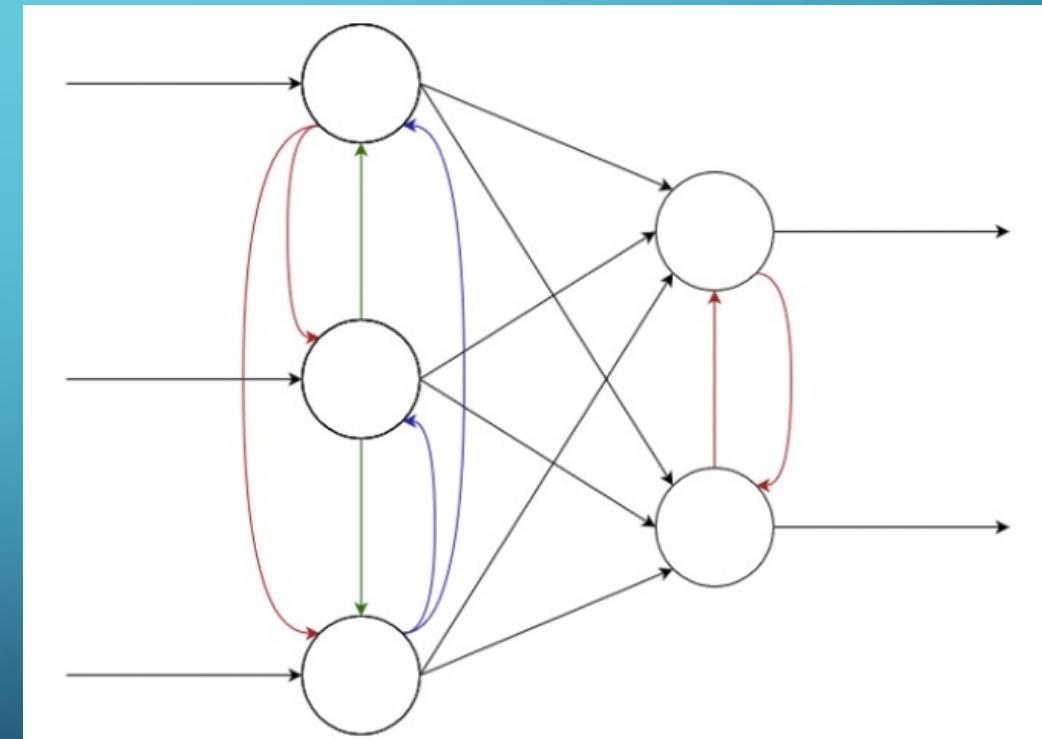
- Ciascuno strato (*layer*) è costituito da uno o più neuroni che modulano il proprio **potenziale di membrana** in funzione degli **spikes in input**
- Quando il potenziale **superà** un valore di soglia, il neurone emette uno *spike* in **output**
- L'informazione trasmessa, sia in input che in output, è strettamente correlata alla **frequenza** con cui vengono emessi gli *spikes*



SPIKING NEURAL NETWORK

STRUTTURA DELLA RETE

- La rete ha una struttura **fully-connected**, ciascun neurone è collegato:
 - a tutti i neuroni dello **stesso layer** (*intra-layer links*)
 - a tutti i neuroni del **layer successivo** e di quello **precedente** (*extra-layer links*)
- Ogni *link* è caratterizzato da un **peso**:
 - **negativo** per gli *intra-layer links*
 - **positivo** per gli *extra-layer links*



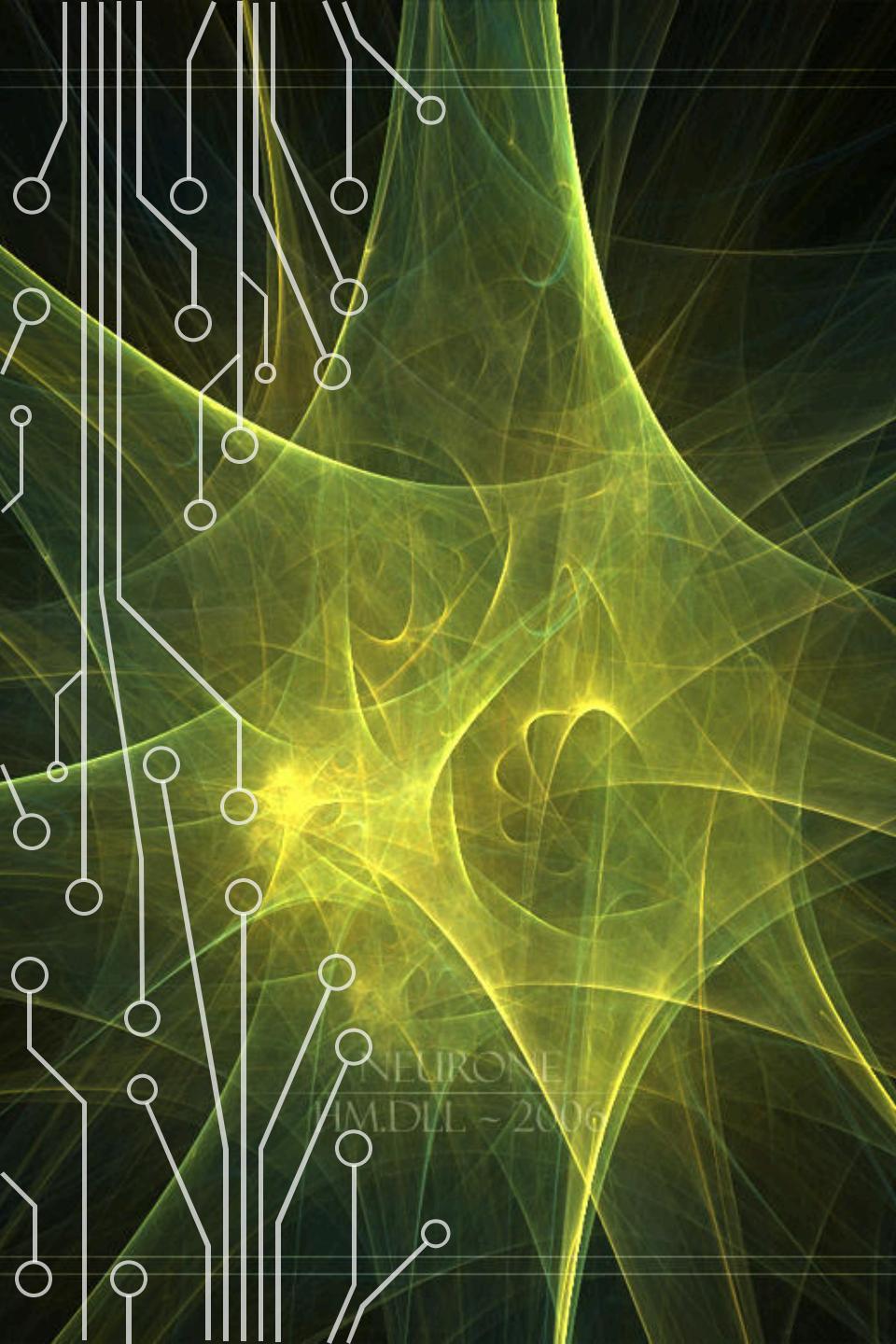
SPIKING NEURAL NETWORK

IMPLEMENTAZIONE

- Sono state fornite due diverse implementazioni per la *Spiking Neural Network*, entrambe costituite da un array (dinamico) di *Layers*:
 - **SNN** → controlla l'**input** a *compile-time* e fornisce in **output** un array statico di spikes
 - **DynSNN** → controlla l'**input** a *run-time* e fornisce in **output** un array dinamico di spikes
- Entrambe le implementazioni sfruttano un **tipo generico N** di neurone

```
pub struct SNN<N: Neuron + Clone + Send + 'static,  
           const NET_INPUT_DIM: usize, const NET_OUTPUT_DIM: usize>  
{  
    layers: Vec<Arc<Mutex<Layer<N>>>>,  
}
```

```
pub struct DynSNN<N: Neuron + Clone + 'static>  
{  
    layers: Vec<Arc<Mutex<Layer<N>>>>  
}
```



MODELLO LEAKY INTEGRATE FIRE

COS'È?

- Il modello *Leaky Integrate and Fire* (LIF) è un particolare modello che considera ogni neurone della rete neurale come il parallelo di una capacità e una resistenza
- Con questo modello, è possibile descrivere l'evoluzione temporale del potenziale della membrana neuronale come segue:

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

EVENT-BASED APPROACH

La libreria sfrutta una strategia *event-based*

In una strategia *event-based* l'elaborazione di un dato (quindi l'utilizzo di risorse) avviene solo quando necessario, ad **istanti di tempo discreti**

Nel modello LIF, seguendo la sua evoluzione temporale, il neurone effettua solo i calcoli essenziali quando un impulso di ingresso lo costringe ad aggiornarsi

EVENT-BASED APPROACH

SPIKE EVENT

- La libreria considera **istanti di tempo discreti** separati da un parametro dt , in ciascuno dei quali vengono emessi (o meno) degli *spikes*
- L'informazione sugli *spikes* si propaga da un *layer* all'altro per mezzo di una *struct SpikeEvent* che contiene un array di *spikes* e l'istante di tempo discreto in cui avviene la computazione

```
pub struct SpikeEvent {  
    ts: u64,           /* discrete time instant */  
    spikes: Vec<u8>,  /* vector of spikes in that instant (a 1/0 for each input neuron) */  
}
```

GENERALIZZAZIONE DEL MODELLO

TRATTO NEURON

- Al fine di **generalizzare** l'interfaccia del neurone, è stato creato il tratto **Neuron**, che definisce una generica funzione per il calcolo del **potenziale di membrana** e, conseguentemente, l'emissione (o meno) di uno **spike**

```
/**  
 * Trait for the implementation of all the Neuron models.  
 * It represents a general Neuron of a Layer  
 */  
pub trait Neuron: Send {  
    /** The neuron function is invoked only when some input spikes arrive from the previous layer;  
     * therefore, extra_weighted_sum is always > 0  
     * - t: time instant when the input spikes arrive  
     * - extra_weighted_sum: dot product between *input spikes* and incoming *weights*  
     * - intra_weighted_sum: dot product between the *input spikes of the previous instant in which at least one neuron (of the previous layer) fired* and the *intra-layer weights*  
     */  
    fn compute_v_mem(&mut self, t: u64, extra_weighted_sum: f64, intra_weighted_sum: f64) -> u8;  
  
    /**  
     * Bring the Neuron to the initial state: initialize all data structures  
     */  
    fn initialize(&mut self);  
}
```

GENERALIZZAZIONE DEL MODELLO

LIF NEURON

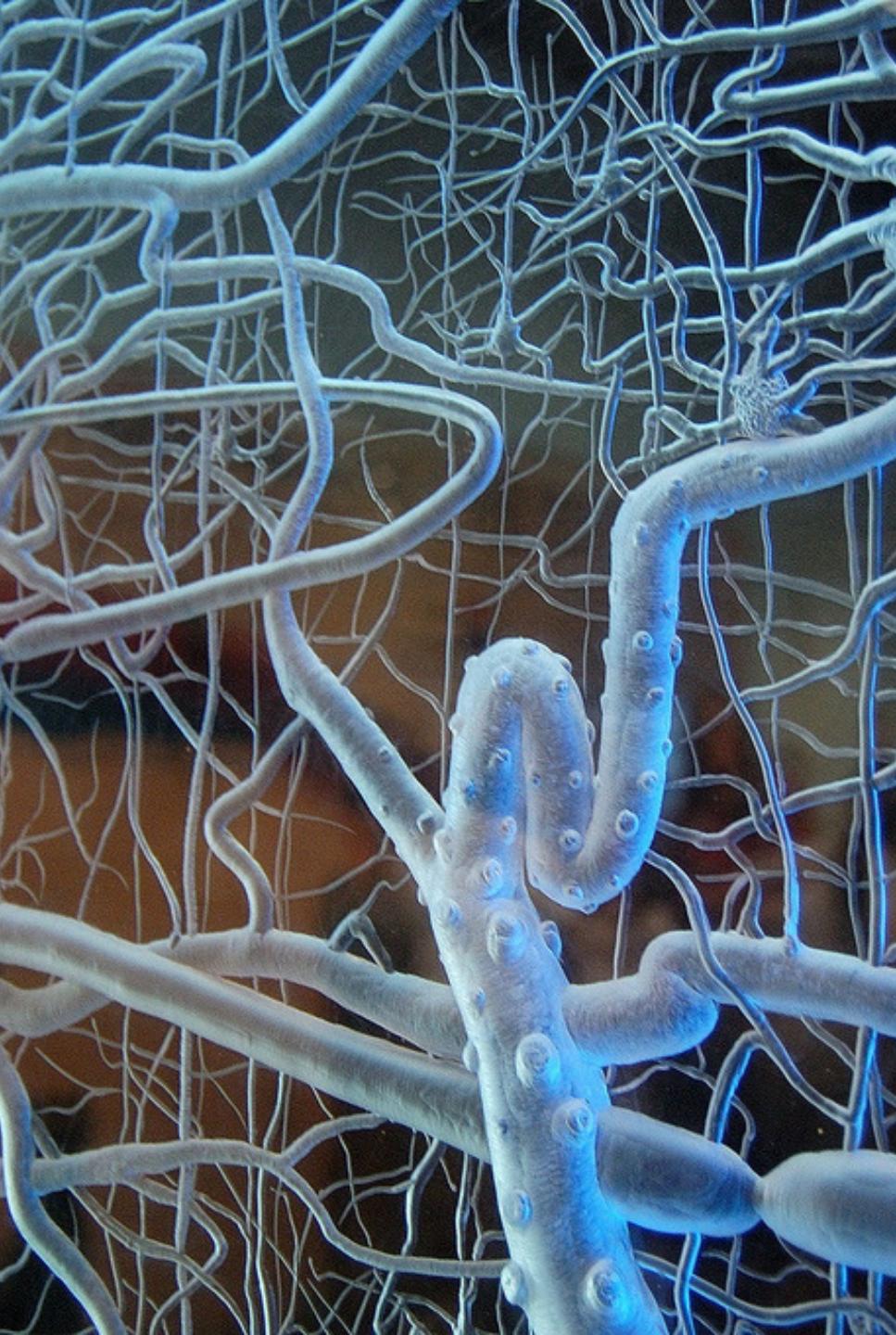
- In particolare, è stato implementato il modello di neurone **LIF**, attraverso la struct **LifNeuron**, avente tutti i parametri completamente configurabili

```
pub struct LifNeuron {  
    /* const fields */  
    v_th: f64,          /* threshold potential */  
    v_rest: f64,        /* resting potential */  
    v_reset: f64,       /* reset potential */  
    tau: f64,  
    dt: f64,           /* time interval between two consecutive instants */  
    /* mutable fields */  
    v_mem: f64,         /* membrane potential */  
    ts: u64,            /* last instant in which has been received at least one spike */  
}
```

BUILDER PATTERN

COS'È

- Il **builder pattern** è un *design pattern* che permette di creare oggetti composti in modo complesso
- Consente di creare un'istanza di un oggetto composto attraverso una classe di costruzione separata, che fornisce un'interfaccia per specificare i vari elementi che compongono l'oggetto
- In questa libreria consente all'utente di avere massima personalizzazione della rete specificando:
 - **Layer**
 - **Neuroni** (con i relativi parametri) per ciascun *layer*
 - **Pesi intra-layer ed extra-layer**



BUILDER PATTERN

TIPI DI BUILDER

All'interno della libreria sono state implementate due tipologie di **builder**:

- **SnnBuilder** (builder statico)

- ✓ L'utente ha disposizione, a **compile-time**, i controlli sulla coerenza dei parametri forniti dall'utente (es. matrice dei pesi in accordo con il numero di neuroni di ciascun layer, ecc.)

- ✗ Non adatto a reti di grandi dimensioni (le strutture dati fornite dall'utente sono allocate sullo **stack**, rischio di **stack overflow**)

- **DynSnnBuilder** (builder dinamico)

- ✓ Adatto a reti di grandi dimensioni (è tutto allocato nell'**heap**)

- ✗ I controlli sulla coerenza delle strutture dati fornite dall'utente vengono effettuati solo a **run-time**

SNNBUILDER

BUILDER STATICO

- Il *SnnBuilder* raccoglie le informazioni sulla struttura della rete specificate dall'utente, quali layers, neuroni, pesi extra-layer e pesi intra-layer, controllandone la coerenza a **compile-time**
- Il metodo *.build()* ha il compito di costruire la rete vera e propria (*SNN*) utilizzando i parametri inseriti

```
pub struct SnnBuilder<N: Neuron + Clone + Send + 'static> {  
    params: SnnParams<N>  
}
```

```
pub struct SnnParams<N: Neuron + Clone + Send + 'static> {  
    pub neurons: Vec<Vec<N>>, /* neurons per each layer */  
    pub extra_weights: Vec<Vec<Vec<f64>>>, /* (positive) weights between layers */  
    pub intra_weights: Vec<Vec<Vec<f64>>>, /* (negative) weights inside the same layer */  
}
```

SNNBUILDER

FLUENT BUILDER PATTERN

- Il *SnnBuilder*, è stato implementato seguendo il *Fluent Builder Pattern*: ciascun metodo ritorna una *struct* che offre solo i metodi consentiti, in base ai parametri inseriti fino a quel momento
- In questo modo, l'utente viene integralmente guidato nel flusso di costruzione della rete

```
impl<N: Neuron + Clone + Send + 'static, const INPUT_DIM: usize, const NET_INPUT_DIM: usize>
    WeightsBuilder<N, INPUT_DIM, NET_INPUT_DIM> {
    /* ... */
    pub fn weights<const NUM_NEURONS: usize>(mut self, weights: [[f64; INPUT_DIM]; NUM_NEURONS])
        -> NeuronsBuilder<N, NUM_NEURONS, NET_INPUT_DIM> {...}
}
```

```
impl<N: Neuron + Clone + Send + 'static, const NUM_NEURONS: usize, const NET_INPUT_DIM: usize>
    NeuronsBuilder<N, NUM_NEURONS, NET_INPUT_DIM> {
    /* ... */
    pub fn neurons(mut self, neurons: [N; NUM_NEURONS])
        -> IntraWeightsBuilder<N, NUM_NEURONS, NET_INPUT_DIM> {...}
}
```

```
impl<N: Neuron + Clone + Send + 'static, const NUM_NEURONS: usize, const NET_INPUT_DIM: usize>
    IntraWeightsBuilder<N, NUM_NEURONS, NET_INPUT_DIM> {
    /* ... */
    pub fn intra_weights(mut self, intra_weights: [[f64; NUM_NEURONS]; NUM_NEURONS])
        -> LayerBuilder<N, NUM_NEURONS, NET_INPUT_DIM> {...}
}
```

```
impl<N: Neuron + Clone + Send + 'static, const OUTPUT_DIM: usize, const NET_INPUT_DIM: usize>
    LayerBuilder<N, OUTPUT_DIM, NET_INPUT_DIM> {
    /* ... */
    pub fn add_layer(self) -> WeightsBuilder<N, OUTPUT_DIM, NET_INPUT_DIM> {...}
    pub fn build(self) -> SNN<N, { NET_INPUT_DIM }, { OUTPUT_DIM }> {...}
}
```

DYNSNNBUILDER

BUILDER DINAMICO

- Il **DynSnnBuilder** raccoglie le informazioni sulla struttura della rete specificate dall'utente e accetta strutture dati solamente allocabili nell'heap
- Il metodo **add_layer()** riceve tutti i dati per la costruzione di un layer (neuroni e pesi *intra* ed *extra* layer) e ne controlla quindi la coerenza solo a ***run-time***

```
pub struct DynSnnBuilder<N: Neuron> {
    params: DynSnnParams<N>
}
```

```
impl<N: Neuron + Clone> DynSnnBuilder<N> {
    /* ... */
    pub fn add_layer(self, neurons: Vec<N>, extra_weights: Vec<Vec<f64>>,
                    intra_weights: Vec<Vec<f64>>) -> Self {...}
    pub fn build(self) -> DynSNN<N> {...}
}
```

```
pub struct DynSnnParams<N: Neuron> {
    pub input_dimensions: usize,           /* dimension of the network input layer */
    pub neurons: Vec<Vec<N>>,           /* neurons per each layer */
    pub extra_weights: Vec<Vec<Vec<f64>>>, /* (positive) weights between layers */
    pub intra_weights: Vec<Vec<Vec<f64>>>, /* (negative) weights inside the same layer */
    pub num_layers: usize,                 /* number of layers */
}
```

PARALLELISMO

TECNICHE DI PARALLELISMO

Per consentire una computazione parallela, sono state analizzate diverse alternative:

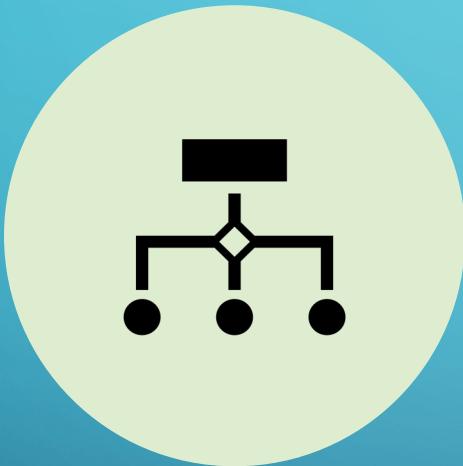
- un **processo** per ciascun *layer*
 - ✓ maggiore indipendenza tra i layer, possibile parallelizzazione su più CPU
 - ✗ maggiore complessità e maggiori risorse impiegate
- un **thread** per ciascun **neurone**
 - ✓ maggiore livello di parallelismo
 - ✗ grande complessità di sincronizzazione tra *layers* e maggiori risorse impiegate
- un **thread** per ciascun *layer*
 - ✓ maggiore semplicità di implementazione
 - ✗ possibile parallelismo su una sola CPU



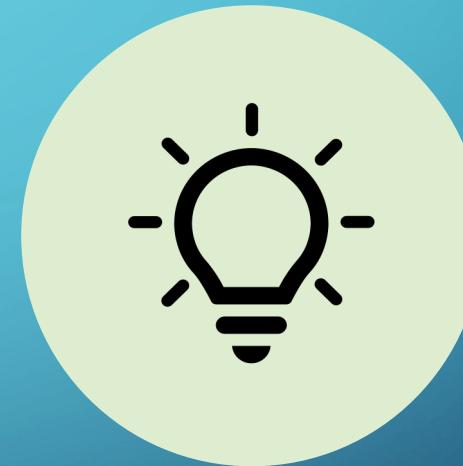
soluzione adottata

PARALLELISMO

SOLUZIONE ADOTTATA: 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 DELLA RETE

INTERFACCIA

- Sia la *SNN* che la *DynSNN* espongono il metodo `.process()` che processa gli *spikes* ricevuti in input, restituendo in output gli *spikes* prodotti dalla rete:
 - *SNN* → controlla a **compile-time** che l'input ricevuto sia coerente con i parametri della rete
 - *DynSNN* → controlla a **run-time** che l'input ricevuto sia coerente con i parametri della rete

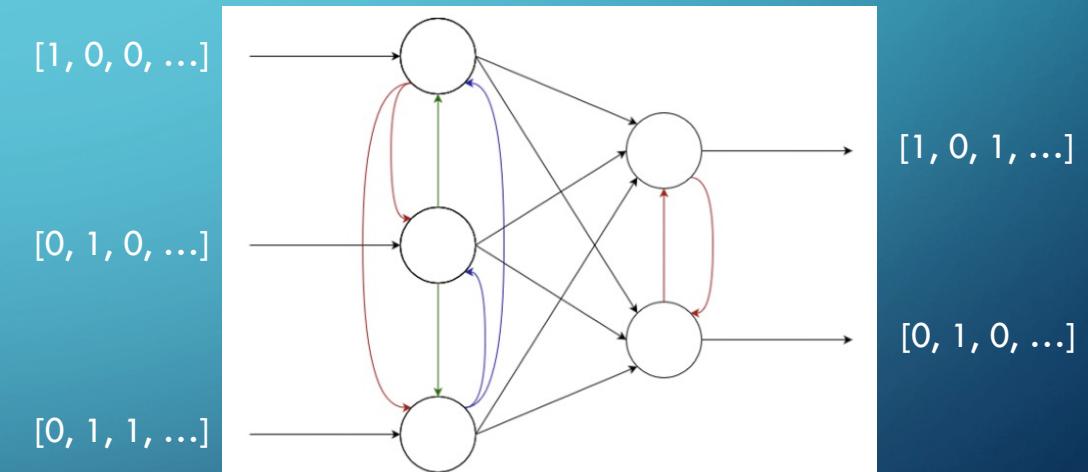
```
impl<N: Neuron + Clone> DynSNN<N> {
    /* ... */
    pub fn process(&mut self, spikes: &Vec<Vec<u8>>) -> Vec<Vec<u8>> {...}
}

impl<N: Neuron + Clone + Send + 'static, const NET_INPUT_DIM: usize, const NET_OUTPUT_DIM: usize>
    SNN<N, NET_INPUT_DIM, NET_OUTPUT_DIM> {
    /* ... */
    pub fn process<const SPIKES_DURATION: usize>(&mut self, spikes: &[[u8; SPIKES_DURATION]; NET_INPUT_DIM])
        -> [[u8; SPIKES_DURATION]; NET_OUTPUT_DIM] {...}
}
```

IMPLEMENTAZIONE DELLA RETE

INPUT E OUTPUT

- In entrambe le implementazioni, sia l'**input** che l'**output** sono un array contenente un *treno di spikes* per ciascun neurone
- Un *treno di spikes* è un array contenente tanti bytes quanti sono gli istanti di tempo considerati. L'i-esimo byte vale:
 - 1 se il neurone riceve/emette uno spike all'istante i-esimo
 - 0 se il neurone **non** riceve/emette spikes all'istante i-esimo



Esempio di treni di spikes in input ed output dalla rete

IMPLEMENTAZIONE DELLA RETE

CODIFICA E DECODIFICA DI SPIKE EVENTS

- Il metodo `.process()` si occupa di **codificare** gli *spikes* in input in *SpikeEvents*, processarli mediante il componente *Processor* e **decodificare** gli *SpikeEvents* ottenuti in *treni di spikes*

```
pub fn process<const SPIKES_DURATION: usize>(&mut self, spikes: &[u8; SPIKES_DURATION]; NET_INPUT_DIM)
    -> [[u8; SPIKES_DURATION]; NET_OUTPUT_DIM] {
    /* encode spikes into SpikeEvent(s) */
    let input_spike_events :Vec<SpikeEvent> = SNN::<N, NET_INPUT_DIM, NET_OUTPUT_DIM>::encode_spikes(spikes);

    /* process input and produce SNN output spikes */
    // let output_spike_events = self.process_events(input_spike_events);
    let processor = Processor {};
    let output_spike_events :Vec<SpikeEvent> = processor.process_events( snn: self, spikes: input_spike_events);

    /* decode output into array shape */
    let decoded_output: [[u8; SPIKES_DURATION]; NET_OUTPUT_DIM] =
        SNN::<N, NET_INPUT_DIM, NET_OUTPUT_DIM>::decode_spikes( spikes: output_spike_events);

    decoded_output
}
```

IMPLEMENTAZIONE DELLA RETE

PROCESSOR

- La *struct Processor* rappresenta il **cuore** di entrambe le *Spiking Neural Networks*: si occupa di processare gli *SpikeEvents* in input lanciando un *thread* per ciascun *layer*, e di restituire il risultato alla (*Dyn*)SNN
- La comunicazione tra thread consecutivi avviene per mezzo di **canali asincroni** (*sync::mpsc::channel()*)

```
pub struct Processor { }

impl Processor {
    /* ... */
    pub fn process_events<'a, N: Neuron + Clone + Send + 'static, S: IntoIterator<Item=&'a mut Arc<Mutex<Layer<N>>>>
        (&self, snn: S, spikes: Vec<SpikeEvent>) -> Vec<SpikeEvent> {...}
}
```

IMPLEMENTAZIONE DELLA RETE

LAYER

- Ciascun *layer* (*thread*) riceve uno *SpikeEvent* dal *layer* (*thread*) precedente, lo processa e, qualora il *layer* emetta almeno uno *spike*, trasmette lo *SpikeEvent* risultante al *layer* (*thread*) successivo
- All'interno del *layer*, ciascun neurone effettua una **somma pesata** degli *spikes* rispetto ai pesi *extra-layer* e aggiorna il suo potenziale di membrana, **emettendo o meno uno *spike***
- Gli **spikes dell'istante precedente** decrementano il potenziale di membrana secondo i pesi *intra-layer*

```
pub struct Layer<N: Neuron + Clone + Send + 'static> {  
    neurons: Vec<N>, /* neurons of the layer */  
    weights: Vec<Vec<f64>>, /* weights between the neurons of this layer and the previous one */  
    intra_weights: Vec<Vec<f64>>, /* weights between the neurons of this layer */  
    prev_output_spikes: Vec<u8> /* output spikes of the previous instant */  
}
```

```
impl<N: Neuron + Clone + Send + 'static> Layer<N> {  
    /* ... */  
    pub fn process(&mut self, layer_input_rc: Receiver<SpikeEvent>,  
                  layer_output_tx: Sender<SpikeEvent>) {...}  
}
```

IMPLEMENTAZIONE DELLA RETE

PROBLEMA: PASSAGGIO DEL LAYER AL THREAD

- Un problema cruciale è il **passaggio di un oggetto Layer al thread** corrispondente: esso **non** può essere direttamente mosso nel thread, poiché la (Dyn)SNN ne resterebbe priva e **non** sarebbe riutilizzabile. Le **soluzioni individuate** erano:

- Clonare il *layer* e passare al thread il suo **clone**
 - ✓ Soluzione più semplice
 - ✗ **Spreco di memoria (duplicazione dei layers)**
- Passare una `&‘static` del *layer* al thread, utilizzando la funzione **Box::leak()**
 - ✓ Semplice
 - ✗ **Box::leak() consuma l’oggetto ricevuto (necessario duplicare i layers)**
- Incapsulare il *layer* in un **Arc<T>** per passarlo al thread
 - ✓ No spreco di memoria
 - ✗ Necessità di incapsulare *Layer* in un **Mutex<T>** per renderlo **mutabile** (spreco di risorse)
- Utilizzare la funzione **std::mem::transmute()** per ottenere una `&‘static` dei *layers*
 - ✓ Non introduce alcun overhead
 - ✗ Uso di **codice unsafe**

soluzione adottata

IMPLEMENTAZIONE DELLA RETE

SOLUZIONE: PASSAGGIO DEL LAYER AL THREAD

- La soluzione adottata sfrutta le proprietà degli *smart pointers*:
 - ***Arc<T>*** → al fine di condividere lo **stesso layer** tra *main thread* e *thread secondario*
 - ***Mutex<T>*** → al fine di rendere **mutabile** il *layer* condiviso
- Pertanto ciascun *layer* si presenta come ***Arc<Mutex<Layer<N>>>***

```
/* create input TX and output RC for each layer and spawn layers' threads */
for layer_ref :&mut Arc<Mutex<Layer<...>>> in snn {
    /* create channel to feed the next layer */
    let (layer_tx :Sender<SpikeEvent> , next_layer_rc :Receiver<SpikeEvent> ) = channel::<SpikeEvent>();

    let layer_ref :Arc<Mutex<Layer<...>>> = layer_ref.clone();

    let thread :JoinHandle<()> = thread::spawn(move || {
        /* retrieve layer */
        let mut layer :MutexGuard<Layer<N>> = layer_ref.lock().unwrap();
        /* execute layer task */
        layer.process( layer_input_rc: layer_rc, layer_output_tx: layer_tx);
    });

    threads.push( value: thread); /* push the new thread into threads' pool */
    layer_rc = next_layer_rc; /* update external rc, to pass it to the next layer */
}
```

Dettaglio del Processor: per ogni *layer* viene creato un *thread* che riceve un *clone* di ***Arc<Mutex<Layer<N>>>*** e mette il *layer* in esecuzione

ESEMPI DI UTILIZZO

BUILDING DINAMICO

- In questo esempio, viene creata una *DynSNN* utilizzando il **builder dinamico**
- La rete ha le seguenti caratteristiche:
 - 5 input per i neuroni del primo *layer*
 - 2 *layers*
 - 4 neuroni di tipo *LifNeuron* nel primo *layer*
 - 2 neuroni di tipo *LifNeuron* nel secondo *layer*

```
let mut snn : DynSNN<LifNeuron> = DynSnnBuilder::<LifNeuron>::new(input_dimension: 5)
    .add_layer(neurons: vec![ /* add 1st layer */
        /* 4 LIF neurons */
        LifNeuron::new(v_th: 0.1, v_rest: 0.1, v_reset: 0.23, tau: 0.45, dt: 1.0),
        LifNeuron::new(v_th: 0.3, v_rest: 0.12, v_reset: 0.54, tau: 0.23, dt: 1.0),
        LifNeuron::new(v_th: 0.2, v_rest: 0.23, v_reset: 0.23, tau: 0.65, dt: 1.0),
        LifNeuron::new(v_th: 0.4, v_rest: 0.34, v_reset: 0.12, tau: 0.45, dt: 1.0)
    ], extra_weights: vec![
        vec![0.9, 0.42, 0.1, 0.31, 0.3],
        vec![0.2, 0.56, 0.1, 0.9, 0.76],
        vec![0.2, 0.23, 0.3, 0.95, 0.5],
        vec![0.23, 0.1, 0.2, 0.4, 0.8]
    ], intra_weights: vec![
        vec![0.0, -0.34, -0.12, -0.23],
        vec![-0.23, 0.0, -0.56, -0.23],
        vec![-0.05, -0.01, 0.0, -0.23],
        vec![-0.23, -0.23, -0.23, 0.0]
    ]).add_layer(neurons: vec![ /* add 2nd layer */
        /* 2 LIF neurons */
        LifNeuron::new(v_th: 0.17, v_rest: 0.12, v_reset: 0.78, tau: 0.67, dt: 1.0),
        LifNeuron::new(v_th: 0.25, v_rest: 0.36, v_reset: 0.71, tau: 0.84, dt: 1.0)
    ], extra_weights: vec![
        vec![0.1, 0.3, 0.4, 0.2],
        vec![0.7, 0.3, 0.1, 0.3]
    ], intra_weights: vec![
        vec![0.0, -0.62],
        vec![-0.12, 0.0]
    ]).build(); /* create DynSNN */
```

ESEMPI DI UTILIZZO

PROCESSING DINAMICO

- In questo esempio, la rete DynSNN precedentemente creata, viene utilizzata per processare gli input (definiti all'interno della variabile ‘`input_spikes`’)
- Sono stati considerati 10 istanti di tempo
- Gli spikes sono memorizzati all'interno di **array dinamici** (`Vec`)

```
let input_spikes : Vec<Vec<u8>> = vec![ /* (10 time instants) */  
    vec![1, 0, 1, 1, 0, 0, 1, 0, 0, 1], /* 1st neuron input spikes */  
    vec![0, 0, 1, 1, 1, 0, 1, 1, 0, 1], /* 2nd neuron input spikes */  
    vec![0, 1, 0, 1, 0, 0, 1, 0, 0, 0], /* 3rd neuron input spikes */  
    vec![0, 1, 0, 1, 1, 0, 1, 0, 0, 0], /* 4th neuron input spikes */  
    vec![1, 1, 1, 0, 0, 0, 1, 0, 0, 1] /* 5th neuron input spikes */  
];  
  
let output_spikes : Vec<Vec<u8>> = dynamic_snn.process(&input_spikes);
```

ESEMPI DI UTILIZZO

BUILDING STATICO

- In questo esempio viene creata una SNN utilizzando il **builder statico**, specificando gli stessi parametri di quella precedente
- È evidente come il flusso di costruzione da seguire imponga all'utente la definizione (prima del building) dei seguenti parametri in ordine:
 - (extra-)weights
 - neurons
 - intra-weights

```
let mut snn :SNN<...> = SnnBuilder::new()
    .add_layer() :WeightsBuilder<...> /* add 1st layer */
    .weights([
        [0.9, 0.42, 0.1, 0.31, 0.3],
        [0.2, 0.56, 0.1, 0.9, 0.76],
        [0.2, 0.23, 0.3, 0.95, 0.5],
        [0.23, 0.1, 0.2, 0.4, 0.8]
    ]).neurons([ /* 4 LIF neurons */
        LifNeuron::new( v_th: 0.1, v_rest: 0.1, v_reset: 0.23, tau: 0.45, dt: 1.0),
        LifNeuron::new( v_th: 0.3, v_rest: 0.12, v_reset: 0.54, tau: 0.23, dt: 1.0),
        LifNeuron::new( v_th: 0.2, v_rest: 0.23, v_reset: 0.23, tau: 0.65, dt: 1.0),
        LifNeuron::new( v_th: 0.4, v_rest: 0.34, v_reset: 0.12, tau: 0.45, dt: 1.0)
    ]).intra_weights([
        [0.0, -0.34, -0.12, -0.23],
        [-0.23, 0.0, -0.56, -0.23],
        [-0.05, -0.01, 0.0, -0.23],
        [-0.23, -0.23, -0.23, 0.0]
    ]) :LayerBuilder<...>
    .add_layer() :WeightsBuilder<...> /* add 2nd layer */
    .weights([
        [0.1, 0.3, 0.4, 0.2],
        [0.7, 0.3, 0.1, 0.3]
    ]).neurons([ /* 2 LIF neurons */
        LifNeuron::new( v_th: 0.17, v_rest: 0.12, v_reset: 0.78, tau: 0.67, dt: 1.0),
        LifNeuron::new( v_th: 0.25, v_rest: 0.36, v_reset: 0.71, tau: 0.84, dt: 1.0)
    ]).intra_weights([
        [0.0, -0.62],
        [-0.12, 0.0]
    ]).build(); /* create SNN */
```

ESEMPI DI UTILIZZO

PROCESSING STATICO

- In questo esempio, la rete SNN precedentemente creata, viene utilizzata per processare gli input (definiti all'interno della variabile ‘`input_spikes`’)
- Sono stati considerati 10 istanti di tempo
- Gli spikes sono memorizzati all'interno di **array statici**

```
let input_spikes : [[u8; 10]; 5] = [ /* (10 time instants) */  
    [1, 0, 1, 1, 1, 0, 1, 0, 0, 0], /* 1st neuron input spikes */  
    [0, 0, 1, 1, 0, 0, 1, 0, 1, 1], /* 2nd neuron input spikes */  
    [0, 1, 1, 0, 1, 0, 1, 0, 0, 1], /* 3rd neuron input spikes */  
    [1, 0, 1, 1, 1, 0, 0, 0, 1, 0], /* 4th neuron input spikes */  
    [1, 0, 1, 1, 0, 0, 1, 0, 0, 1], /* 5th neuron input spikes */  
];  
  
let output_spikes : [[u8; 10]; 2] = snn.process(&input_spikes);
```

TEST DI ACCURATEZZA

- Il funzionamento della rete è stato testato mediante uno script in *Python* che valuta l'accuratezza della rete su un *dataset* di 10000 immagini
- Gli **iperparametri** della rete (come i pesi eccitatori) utilizzati erano già stati allenati precedentemente
- L'**accuratezza** raggiunta dalla rete è prossima a quella prevista (~70%)

```
Done!
[7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4 9 6 6 5 4 0 7 4 0 1 3 1 3 4 7 2 7
 1 2 1 1 7 4 2 3 5 1 2 4 4 6 3 5 5 6 0 4 1 9 5 7 8 9 3 7 4 6 4 3 0 7 0 2 9
 1 7 3 2 9 7 7 6 2 7 8 4 7 3 6 1 3 6 9 3 1 4 1 7 6 9]

Accuracy: ['70.00%']
```

Esempio di test di accuratezza: lo script è stato eseguito con 100 iterazioni valutando un'accuratezza media della rete del 70% sulle predizioni effettuate

CONCLUSIONI

- La realizzazione di una *Spiking Neural Network* in Rust ha mostrato che il linguaggio si presta bene anche alla creazione di sofisticati modelli neurali
- Il suo utilizzo ha permesso di ottenere un'elevata **efficienza e velocità di esecuzione**, nonché una maggiore **sicurezza e robustezza** del codice
- Sono stati applicati, oltre alle funzionalità di base in Rust, i concetti di **generalizzazione e parallelismo** predisponendo interfacce generiche (es. quella del neurone) ed utilizzando tecniche di parallelizzazione



GRAZIE PER L'ATTENZIONE

- Francesco Rosati
- Giuseppe Lazzara
- Mario Mastrandrea

