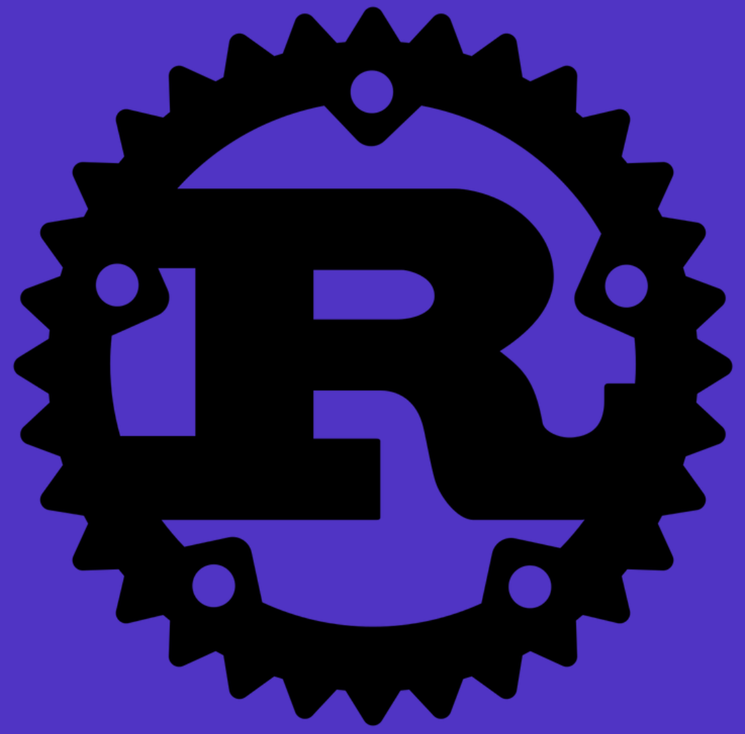
Sviluppo e Ottimizzazione di una Rete Neurale Generica tramite Configurazione e Esecuzione ottimizzata con Standard ONNX – Progetto 2.1

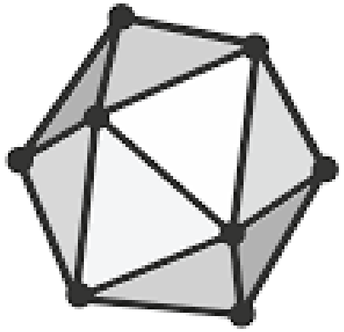


Gruppo 34

Sante Laera s310226

Dario Marchitelli s310030

Gioele Ramondetti s316257



ONNX

Indice

Introduzione

Implementazione di un interprete ONNX in Rust per la configurazione e l’ottimizzazione dell'esecuzione di reti neurali generiche.

1.1 Obiettivo

Lo scopo del progetto consiste nella configurazione ed esecuzione ottimizzata di una rete neurale generica mediante lo standard Open Neural Network eXchange (ONNX).

Nello specifico gli obiettivi sono stati:

-Creazione di un **parser** per l’estrazione di informazioni a partire da un file ONNX

-Implementazione di un sotto-insieme delle **funzionalità** e delle **operazioni** ONNX

-Dimostrazione dell'**inferenza** a partire da un modello già allenato

-Utilizzo estensivo della **parallelizzazione** nell'esecuzione della rete

-Fornire dei **binding** Rust verso altri linguaggi come Javascript e Python.

2. Implementazione interprete Onnx in rust

Serializzazione e deserializzazione dei modelli ONNX utilizzando la libreria protoc con protobuf.

2.1 Creazione del Parser ONNX

Per permettere la **lettura dei file** con estensione onnx è stata utilizzata la libreria **protoc**.
In particolare questa libreria utilizza **Protobuf** per leggere un file onnx.proto e generare un file rust (onnx.rs) con la definizione delle strutture di cui il file onnx è formato.
Il file **build.rs** presenta il codice necessario alla creazione del file onnx.rs durante la fase di build.

```rust
fn main() -> Result<(), Box<dyn std::error::Error>> {
    prost_build::Config::new() Config
        .out_dir(path: "src") &mut Config
        .compile_protos(protos: &["src/onnx.proto3"], includes: &["/src"])?;
    Ok(())
}
```

2.2 Estrazione delle informazioni dai file serializzati

La funzione **decode_message** consente di **leggere un file strutturato** (.onnx oppure .pb) decodificandolo in una struttura dati **prost::Message**.

```rust
fn decode_message<M: prost::Message + std::default::Default>(path: &str) -> M {
    let data: Vec<u8> = std::fs::read(path).expect(msg: "Failed to read ProtoBuf file");
    prost::Message::decode(buf: &data[..]).expect(msg: "Failed to decode ProtoBuf data")
}
```

Sono state quindi definite le funzioni specifiche per leggere il **modello** da file onnx e per leggere i **tensori** dai file protobuf

2.2 Estrazione delle informazioni dai file serializzati

La funzione ***read_model*** usa *decode_message* per leggere il **modello**.
In seguito vengono settati i **nomi dei nodi** (dato che sono opzionali) a partire dal nome dell'output del nodo stesso. Questo per far si che la run e la visualizzazione funzionino correttamente.

```rust
pub fn read_model(path: &str) -> ModelProto{
    let mut model: ModelProto = decode_message(path);
    //è necessario che tutti i nodi abbiano un nome univoco che è uguale all nome dell'output del nodo
    for node: &mut NodeProto in &mut model.graph.as_mut().unwrap().node {
        node.name = node.output.get(index: 0).unwrap().clone();
    };
    model
}
```

Similmente **read_tensor** usa *decode_message* per leggere un **tensore**

```rust
pub fn read_tensor(path: &str) -> TensorProto{
    decode_message(path)
}
```

2.3 Operazioni di serializzazione e di salvataggio del modello

La funzione **write_message** consente di salvare un file a partire da una struttura generica prost::Message.
Questa funzione è utilizzata per il **salvataggio del modello** su un file onnx a seguito di una modifica.

```rust
pub fn write_message<M: prost::Message>(message: &M, path: &str) -> std::io::Result<usize> {
    let mut buf: Vec<u8> = Vec::new();
    buf.reserve(additional: message.encoded_len());
    message.encode(&mut buf).unwrap();
    std::fs::write(path, contents: &*buf)?;
    Ok(buf.len())
}
```

Per esempio una possibile implementazione è:

```rust
write_message(message: &model_proto, path: "model.onnx").unwrap();
```

2.4 Operazioni implementate per l'inferenza

1. **ADD**: Somma due tensori.
2. **RELU**: Applica la funzione di attivazione ReLU ai tensori di input.
3. **EXP**: Calcola l'esponenziale dei tensori di input.
4. **CONCAT**: Concatena i tensori lungo una dimensione specificata.
5. **FLATTEN**: Appiattisce il tensore di input.
6. **RESHAPE**: Cambia la forma del tensore di input secondo una nuova forma specificata.
7. **CONV**: Esegue l'operazione di convoluzione tra i tensori di input e i kernel specificati.
8. **MAXPOOL**: Esegue un'operazione di max pooling sui tensori di input.
9. **BATCH NORMALIZATION**: Applica la normalizzazione batch ai tensori di input.
10. **DROPOUT**: Applica la tecnica di dropout ai tensori di input.
11. **SOFTMAX**: Applica la funzione di attivazione softmax ai tensori di input.

12. **GEMM**: Esegue una moltiplicazione di matrici generalizzata (General Matrix Multiply).
13. **MATMUL**: Esegue una moltiplicazione di matrici.
14. **REDUCESUM**: Calcola la somma ridotta di tutti gli elementi del tensore di input.
15. **GLOBALAVERAGEPOOL**: Esegue una pool media globale sui tensori di input.
16. **LRN**: Esegue la normalizzazione locale responsiva.

2.5 Modelli utilizzati

Per valutare l’efficienza nella parallelizzazione e l'inferenza del sistema, sono stati utilizzati **cinque modelli**:

- AlexNet
- CaffeNet
- MobileNetV2
- ResNet
- SqueezeNet

Questi modelli rappresentano architetture di reti neurali **eterogenee** ognuna con caratteristiche specifiche, ad esempio alcune sono caretterizzate dall’assenza di nodi in **parallelo** e altre dalla loro presenza.

3. Efficenza nell’esecuzione della rete

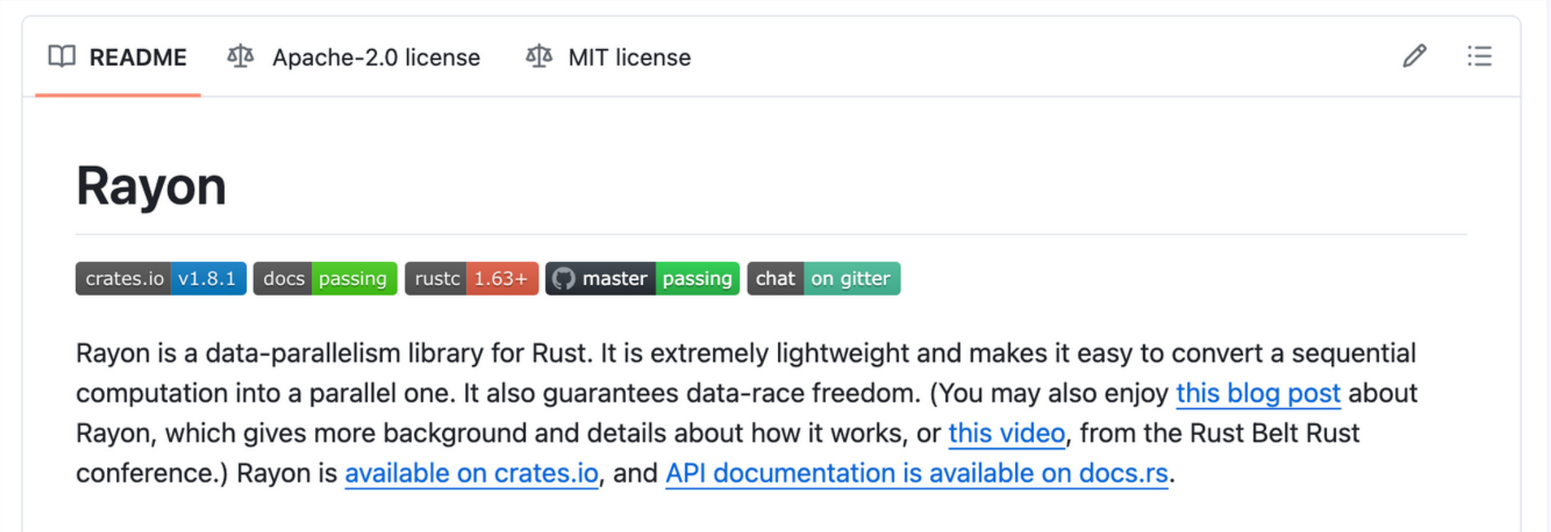
Viene offerta all’utente la possibilità di scegliere il tipo di esecuzione della rete, scegliendo tra **esecuzione parallela** ed **esecuzione sequenziale**.

La parallelizzazione è stata implementata su due livelli:
-Parallelizzazione **interna** ai nodi
-Parallelizzazione **esterna** dei nodi

3.1 Parallelizzazione interna ai nodi

La parallelizzazione **interna** ai nodi è stata eseguita cercando di parallelizzare le istruzioni eseguite internamente alle singole operazioni.

Ciò è stato fatto mediante l'utilizzo del metodo **into_par_iter** fornito dal crate **Rayon** in Rust che consente di convertire una collezione in un iteratore le cui operazioni vengono eseguite in parallelo



3.1 Parallelizzazione interna ai nodi

```
if is_par_enabled {
    results = (0..batch_size) Range<usize>
        .into_par_iter() Iter<usize>
        .map(map_op: |i: usize| {
            let batch_data: ArrayBase<ViewRepr<&f32>, …> = input_nd_array.index_axis(Axis(0), index: i);
            batch_data.mapv(|el: f32| el.exp())
        }) Map<Iter<usize>, impl Fn(…) -> …>
        .collect();
} else {
```

In questo esempio, **into_par_iter** converte il Range<usize> in un iteratore parallelo.
Il **vantaggio** è che Rayon gestisce automaticamente la parallelizzazione dell'esecuzione su più core, rendendo il codice più efficiente in termini di prestazioni in situazioni in cui ci sono risorse di calcolo disponibili su più core.

3.2 Parallelizzazione esterna dei nodi

La parallelizzazione **esterna** dei nodi è stata eseguita grazie all’utilizzo dei canali di Rust, questo tipo di parallelizzazione è molto utile, poiché in molte reti il risultato di un nodo viene utilizzato come input da diversi nodi, consentendo loro di essere eseguiti in parallelo e accelerare l'esecuzione generale della rete.

3.2 Parallelizzazione esterna dei nodi

L’esecuzione della rete viene gestita mediante la struttura **OnnxRunningEnviroment** che contiene:

-**input_tensor**: è il tensore di input alla rete

-**input_senders**: vettore di senders (solitamente di dimensione 1) che consentono l’invio di input_tensor ai nodi iniziali della rete

-**output_receiver**: receiver tramite cui verrà ricevuto il risultato finale dell’esecuzione della rete

-**model**: il modello di cui avverrà l’esecuzioen

-**node_io_vec**: vettore di **NodeIO**, struttura ausiliara per l’esecuzione del singolo nodo

```rust
pub struct OnnxRunningEnvironment {
    input_tensor: TensorProto,
    input_senders: Vec<Sender<TensorProto>>,
    output_receiver: Receiver<TensorProto>,
    model: ModelProto,
    node_io_vec: Vec<NodeIO>,
}
```

3.2 Parallelizzazione esterna dei nodi

La struttura **NodeIO** è una struttura ausiliaria per l'esecuzione del singolo nodo, essa contiene:

-**senders**: contiene l'insieme di senders a cui inviare il risulta dell'esecuzione del nodo

-**optional_receiver**: contiene il receiver da cui ricevere i dati in input su cui eseguire le operazioni

-**node**: contiene le informazioni del nodo, prese direttamente dal modello

-**initializers**: contiene la lista di initializers richiesti per l'esecuzione del nodo

```
struct NodeIO {
    senders: Vec<Sender<TensorProto>>,
    optional_receiver: Option<Receiver<TensorProto>>,
    node: NodeProto,
    initializers: Vec<TensorProto>,
}
```

3.2 Parallelizzazione esterna dei nodi

Il processo di preparazione all'esecuzione inizia con la **lettura** dei nodi dal modello.
Per ogni nodo, viene creato un elemento nel vettore di NodeIO.
Esso comprende un nuovo **receiver** e un **vettore di sender** vuoto. Successivamente, si esamina la struttura e si selezionano i nodi che hanno come output i nodi di input del nodo corrente. Per ciascun elemento, si aggiunge il sender del canale al vettore di sender.

```
let (sender, receiver) = channel();
let mut senders = Vec::new();
optional_receiver = Some(receiver);
let mut current_node_clone = current_node.clone();
let new_node_io = NodeIO {
    senders,
    optional_receiver,
    node: current_node_clone.clone(),
    initializers: Self::get_initializers(
        model.clone().graph.unwrap(),
        current_node.clone(),
    ),
};
```

```
for node_io in &mut node_io_vec {
    if node_io
        .node
        .output
        .iter()
        .any(|x| current_node.input.contains(x))
    {
        node_io.senders.push(sender.clone());
    };
}
```

3.2 Parallelizzazione esterna dei nodi

Durante l'esecuzione, si scorre la struttura, e per ogni nodo viene creato un **thread**. Il processo include il **riconoscimento degli input** di cui necessita il nodo, l'**attesa dell'arrivo di tutti gli input** dal receiver, la lettura delle costanti e degli attributi, l'esecuzione dell'operazione e l'invio del risultato a tutti i sender nel vettore. Questo approccio consente una gestione efficiente della concorrenza e una parallelizzazione delle operazioni della rete neurale, contribuendo così a migliorare le prestazioni complessive del sistema.

```
thread::scope(|s: &Scope<'_, '_>| {
    for current_node: &NodeIO in self.node_io_vec.iter() {
        s.spawn(move || {
            let NodeIO {
                senders: &Vec<Sender<TensorProto>>,
                optional_receiver: &Option<Receiver<TensorProto>>,
                node: &NodeProto,
                initializers: &Vec<TensorProto>,
            } = current_node;
            if let Some(receiver: &Receiver<TensorProto>) = optional_receiver {
                let inputs: Vec<TensorProto> = Self::get_inputs(receiver, node, initializ
                let output_result: Result<TensorProto, OnnxError> = find_and_do_operation
                    node_for_op: node,
                    initializers.clone(),
                    inputs.clone(),
                    is_par_enabled: flag_par,
                );
```

4. Esportazione funzionalità

All’interno del sistema sono stati implementati dei binding verso diversi linguaggi di programmazione, più nello specifico per Python e Javascript.

4.1 Binding Javascript con Neon

Per utilizzare funzioni Rust tramite Javascript abbiamo sfruttato ‘Neon’.

```
[dependencies.neon]
version = "0.10"
default-features = false
features = ["napi-6"]
```

Nel file lib.rs o in un modulo separato, vengono definite le funzioni Rust da esportare a JavaScript, dentro un apposito modulo ‘js_binding’.
Per un corretto funzionamento è necessario avere Node.js installato e aver inserito Neon come dipendenza del file ‘package.json’ seguito dai comandi:

- “npm install” per installare le dipendenze
- “npm build” per compilare il progetto rust in un modulo di Node.js

```rust
#[neon::main]
fn main_js(mut cx: ModuleContext) -> NeonResult<()> {
    cx.export_function(key: "start", f: start)?;
    cx.export_function(key: "select_model", f: select_model)?;
    Ok(())
}
```

```rust
fn delete_node(mut cx: FunctionContext) -> JsResult<JsString>{
    let node_name: String = cx.argument::<JsString>(0)?.value(&mut cx);
    let graph: GraphProto = stateful_backend_environment::remove_node(node_name).graph.unwrap();
    // Convert GraphProto to JSON structure
    let json_result: JsonResult = (&graph).into();

    // Convert the JsonResult to JSON string
    let json_string: String = serde_json::to_string(&json_result).expect(msg: "Failed to convert to JSON string");
    Ok(cx.string::<String>(json_string))
}

#[neon::main]
fn main_js(mut cx: ModuleContext) -> NeonResult<()> {
    cx.export_function("hello", hello)?;
    cx.export_function("start", start)?;
    cx.export_function("select_model", select_model)?;
    cx.export_function("run", run)?;
    cx.export_function("get_node_js", get_node_js)?;
    cx.export_function("create_node", create_node)?;
    cx.export_function("modify_node", modify_node)?;
    cx.export_function("delete_node", delete_node)?;
    Ok(())
}
```

4.2 Binding python con PyO3

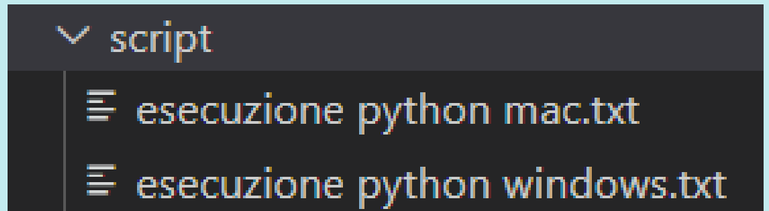
Per utilizzare funzioni Rust all’interno di Python è stata utilizzata PyO3 come dipendenza nel file Cargo.toml, includendo la feature "extension-module" per abilitare le funzionalità di modulo estensione.

```
pyo3 = { version = "0.19.2", features = ["extension-module"], optional = false }
```

Nel file src/lib.rs abbiamo inserito PyO3 per definire le funzioni Rust da esportare.

```rust
#[pymodule]
fn group_34(_py: Python<'_>, m: &PyModule) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(main_python, m)?)?;
    Ok(())
}
```

Eseguendo da terminale lo script contenuto nel file di testo è possibile eseguire l’ambiente virtuale Python e utilizzare la funzione “main_python” che permette l’esecuzione del programma.

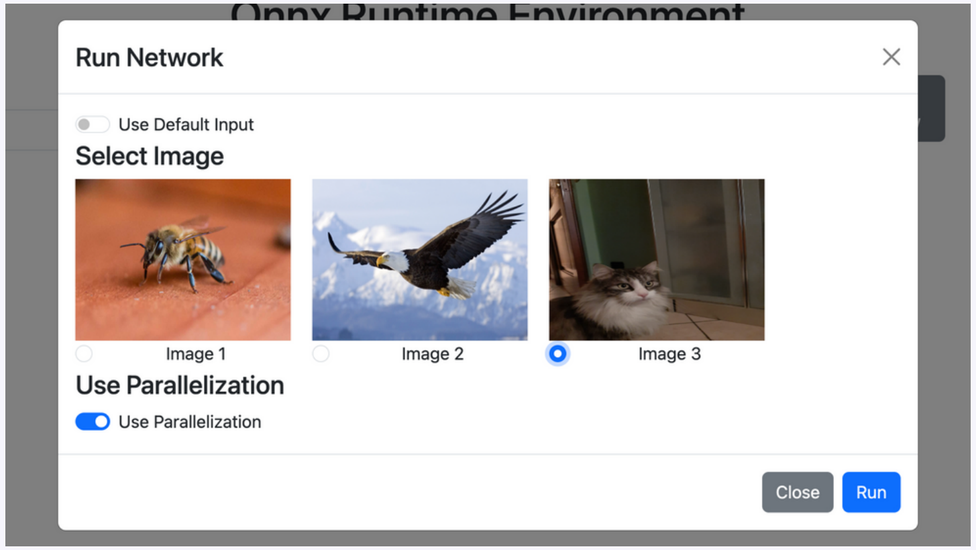


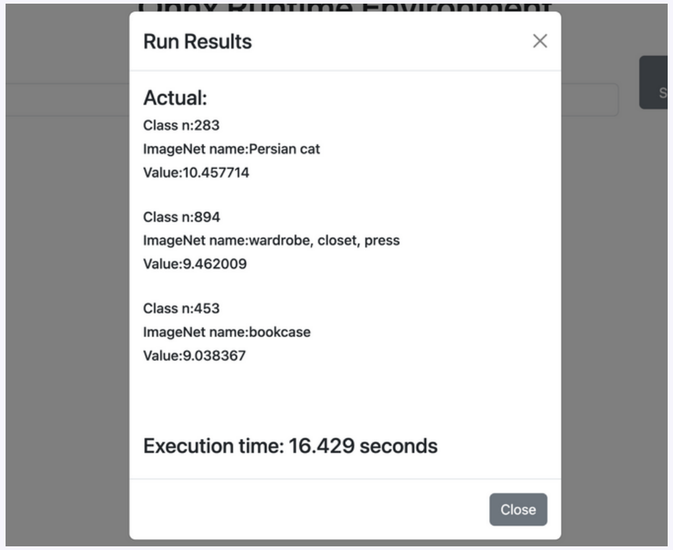
5. Nuove features

Sono state sviluppate alcune features per la visualizzazione e la modifica del modello in maniera più intuitiva e un metodo per utilizzare una rete come classificatore di immagini.

5.1 Image classification con Mobilenetv2

Abbiamo utilizzato la rete neurale MobileNetV2, precedentemente addestrata su ImageNet, per la classificazione di immagini fornite dall'utente.
La rete è in grado di identificare automaticamente gli oggetti nelle immagini, fornendo i rispettivi nomi di classe reperiti direttamente dalle classi di Imagenet.
L'utente può scegliere un'immagine che viene automaticamente adattata alle dimensioni richieste dalla rete. Questo consente una classificazione del contenuto dell'immagine che viene fatto tramite esecuzione parallela o sequenziale in base alla scelta effettuata dall'utente.





Fornendo un'immagine di dimensioni variabili, come quella visualizzata a lato
Il risultato della classificazione restituirà le seguenti informazioni:

- Classe Riconosciuta 1: Persian cat
- Classe Riconosciuta 2: Wardrobe

MobileNetV2, addestrato su ImageNet, è in grado di identificare simultaneamente gli oggetti presenti nell'immagine, fornendo una descrizione accurata delle classi rilevate.

5.2 CRUD dei nodi del modello

Viene fornita la possibilità di inserire, modificare ed eliminare un nodo, con la conseguente possibilità di salvare il modello modificato e di eseguirlo.

INSERIMENTO

La possibilità di inserire un nodo prevede che l'utente sia esperto in materia e vada ad inserire correttamente i parametri e le inizializzazioni necessarie per creare un nodo correttamente.

MODIFICA

La modifica relativa ad un nodo riceve il nome relativo al nodo da modificare con tutti i parametri che il nodo richiede
Anche in questo caso un utente con esperienza.

ELIMINAZIONE

L'eliminazione di un nodo riceve il nome del nodo da eliminare e provvede alla sua rimozione dal modello.

5.3 Salvataggio delle modifiche

Per gestire lo stato in modo che successivamente le modifiche apportate dal frontend siano persistenti si sono realizzati dei metodi per il salvataggio dello stato.

```rust
pub fn start() -> std::io::Result<()>
```

All'avvio del frontend viene creata una cartella *backend_state* dove viene salvato una copia del modello selezionato nel file *model.onnx*, il quale verrà modificato ad ogni operazione di CRUD e salvato nuovamente. In aggiunta viene creato un json con il nome del modello e i valori di default degli input

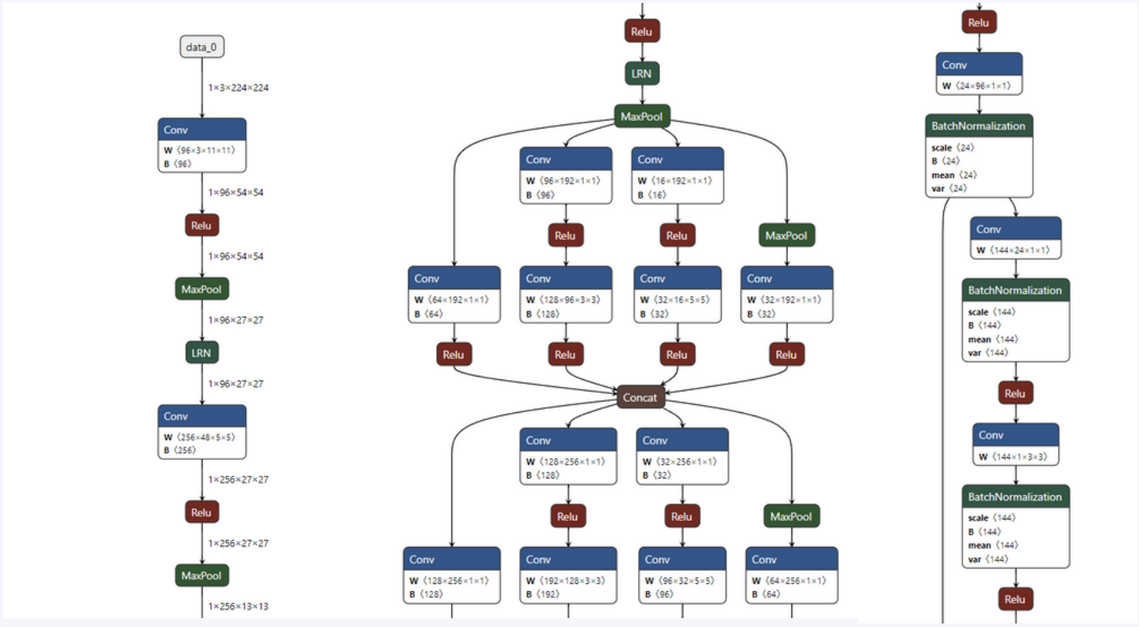
```rust
pub struct ServerState {
    pub model_name: String,
    pub default_input_path: String,
    pub default_output_pat: String,
}
```

Ogni funzione di CRUD viene quindi incapsulata in una funzione che effettua il salvataggio dello stato e che converte il DTO fornito dal frontend in variabili utilizzabili dalla funzione di CRUD.

5.4 Frontend

Per lo sviluppo del frontend, ci siamo ispirati alla UI offerta da Netron un'applicazione web che consente di visualizzare una rete neurale a partire dalla sua specifica in formato ONNX.
Il frontend da noi sviluppato tenta di superare quanto offerto da Netron aggiungendo: la possibilità di eseguire la rete selezionando gli input e la configurazione e la possibilità di modificarne la struttura e i nodi consentendo in seguito l'esecuzione del modello creato e il suo salvataggio in formato ONNX.

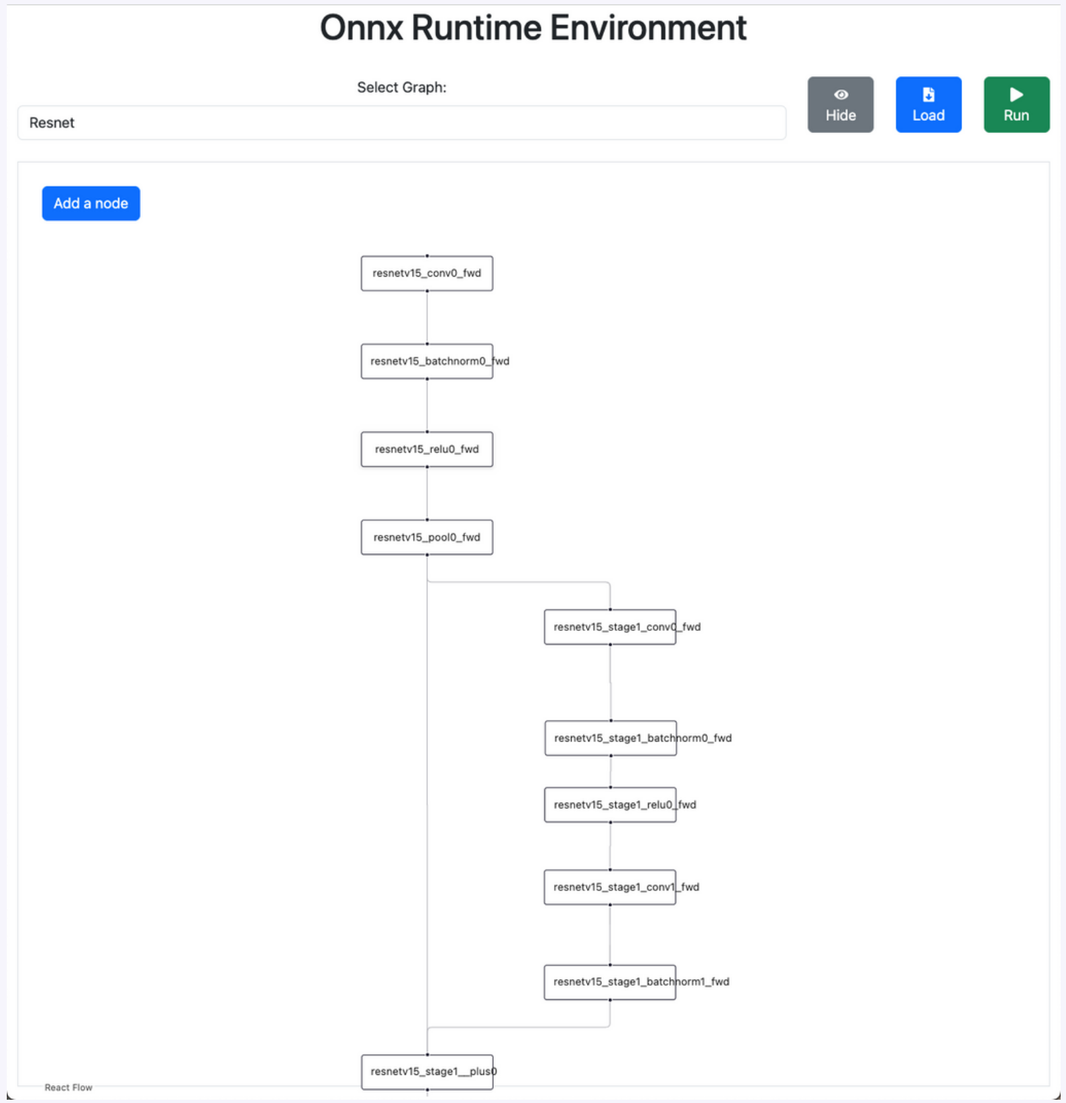
NETRON



Selezione e visualizzazione del modello

Tramite la visualizzazione web è possibile **selezionare** una rete tra quelle rese disponibili e **visualizzarne il grafo** tramite il pulsante “Show”/”Hide”.

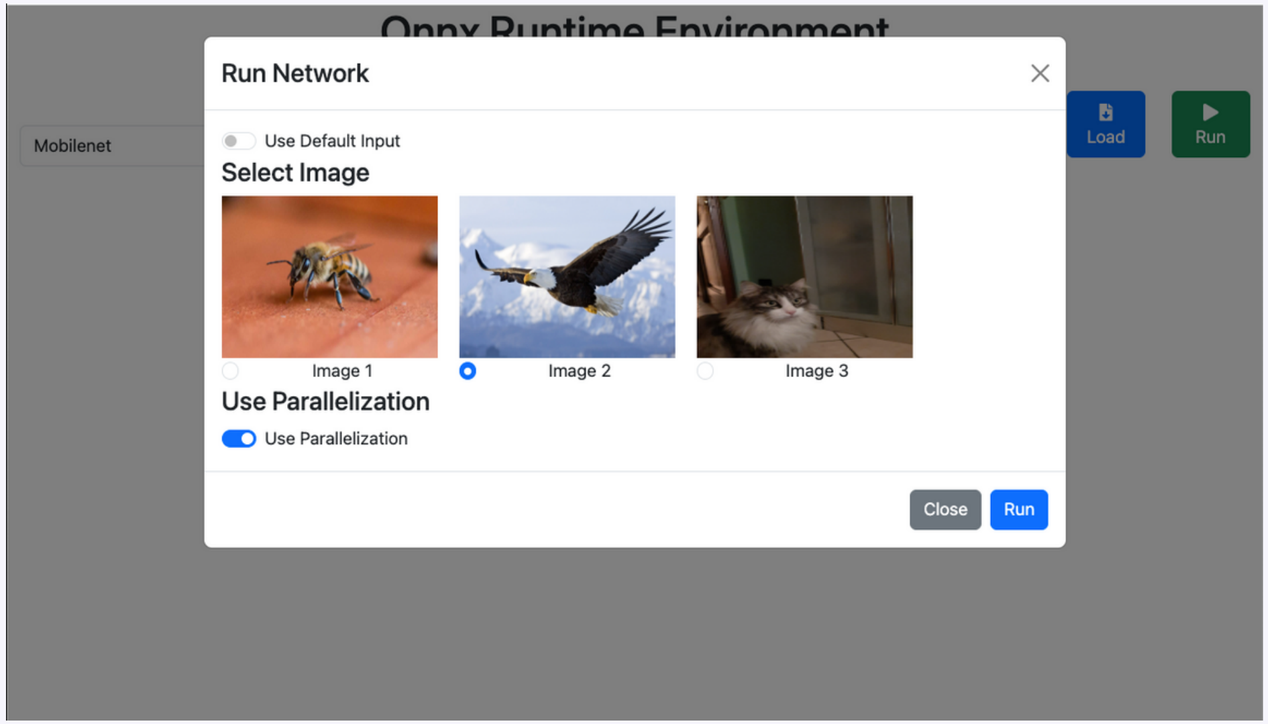
Disclaimer: la visualizzazione del grafo di reti molto grandi potrebbe non essere istantanea, ma non è necessaria se si vuole solo eseguire il run della rete.



Esecuzione del modello

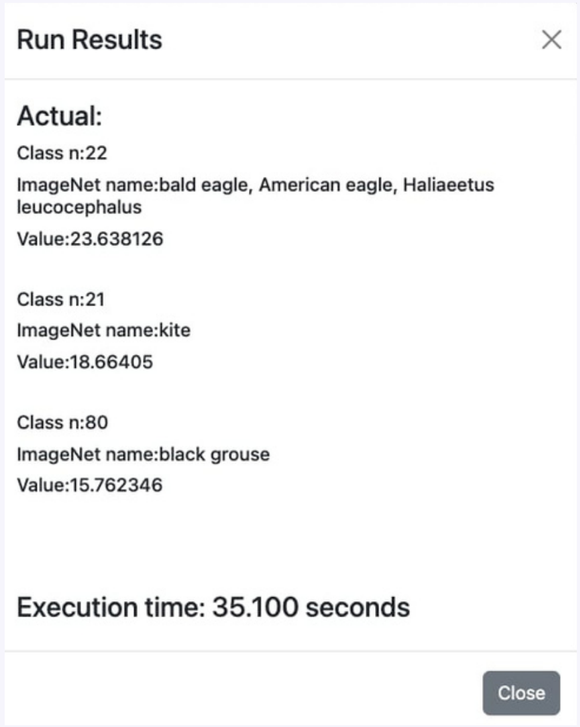
Dopo aver caricato un modello, mediante il tasto “Run” è possibile **eseguirlo** previa selezione della **configurazione** di esecuzione.

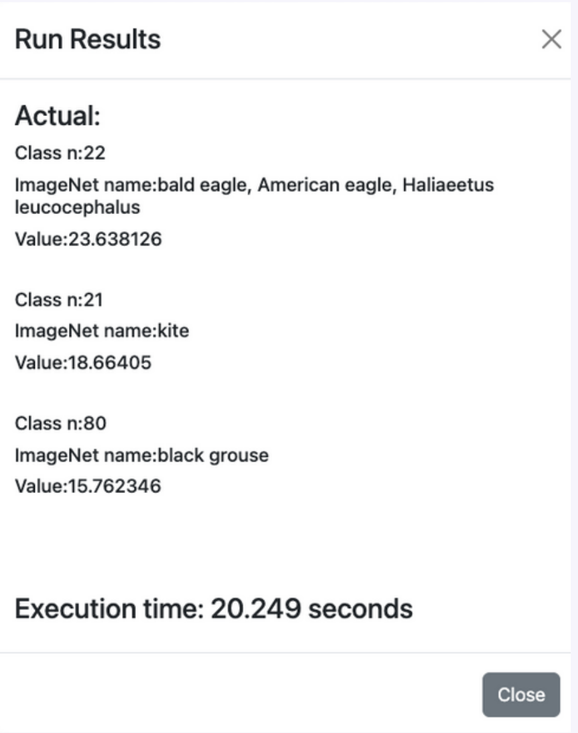
Nella modale di configurazione è possibile:
-Selezionare l’**input** (utilizzo dell’immagine attualmente solo per la Mobilenet)
-Attivare o meno l’utilizzo di **parallelizzazione**



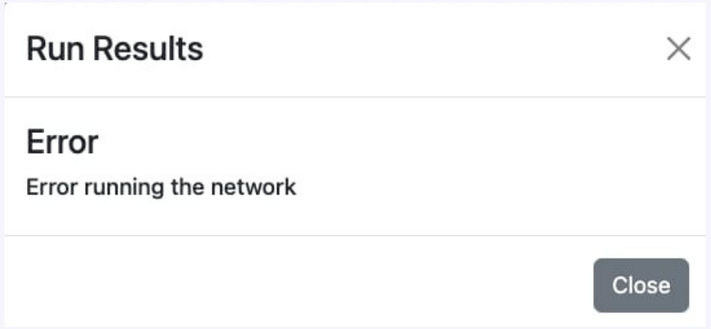
Risultato dell’esecuzione

Se l’esecuzione è andata a buon fine verranno visualizzati i **risultati** dell’inferenza che comprendono le **classi riconosciute** ed il **tempo di esecuzione**.
Al lato si può notare l’efficienza dell’utilizzo estensivo della **parallelizzazione** per l’esecuzione della rete **mobilenet**.





Se l’esecuzione non va a buon fine viene mostrato un messaggio di **errore**.



Creazione e modifica dei nodi

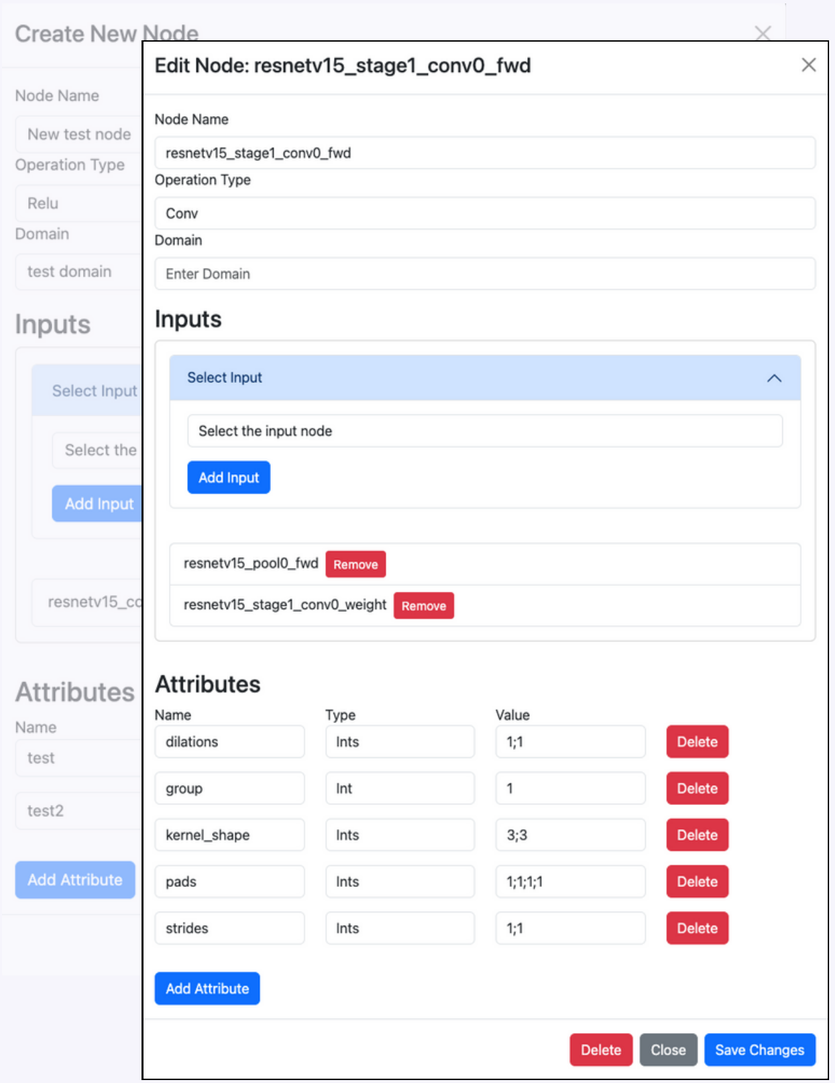
Tramite il pulsante “Add a node” è possibile inserire un nuovo nodo specificando:

-**Operazione**

-**Input** e **initializers**

-Eventuali **attributi**, con possibilità di selezionare nome, tipo (fra quelli supportati) e valore

Selezionando un nodo del grafo è possibile modificarlo o eliminarlo tramite una modale apposita.



6. Sviluppi futuri

-Download del modello ONNX

-Upload dell’immagine su cui effettuare l’inferenza

-Validazione delle operazioni di CRUD

-Migliorare e ottimizzare la visualizzazione del grafo

-Creazione di initializers

-Rendere il binding python più flessibile

Grazie per l'attenzione!

Gruppo 34

Sante Laera s310226

Dario Marchitelli s310030

Gioele Ramondetti s316257

Referenze

[1] «ONNX,» [Online]. Available: https://onnx.ai/onnx/index.html.
[2] «Interpreti,» [Online]. Available: https://github.com/microsoft/onnxruntime.
[3] «ONNXRUNTIME,» [Online]. Available: https://onnxruntime.ai
[4] «Operatori,» [Online]. Available: https://onnx.ai/onnx/operators/index.html.
[5] «Modelli,» [Online]. Available: https://github.com/onnx/models/tree/main
[6] «Js Biding,» [Online]. Available: https://neon-bindings.com/
[7] «Python BInding,» [Online]. Available: https://pyo3.rs/
[8] «Protoc,» [Online]. Available: https://grpc.io/docs/protoc-installation/
[9] «Frontend,» [Online]. Available:https://www.npmjs.com/package/react-flow-renderer