# Optimizing NN Inference on Embedded Devices

Master's Internship – May-July 2018
Snips Paris – Under the supervision of Mathieu Poumeyrol

Romain Liautaud

*École Normale Supérieure de Lyon*

# Overview

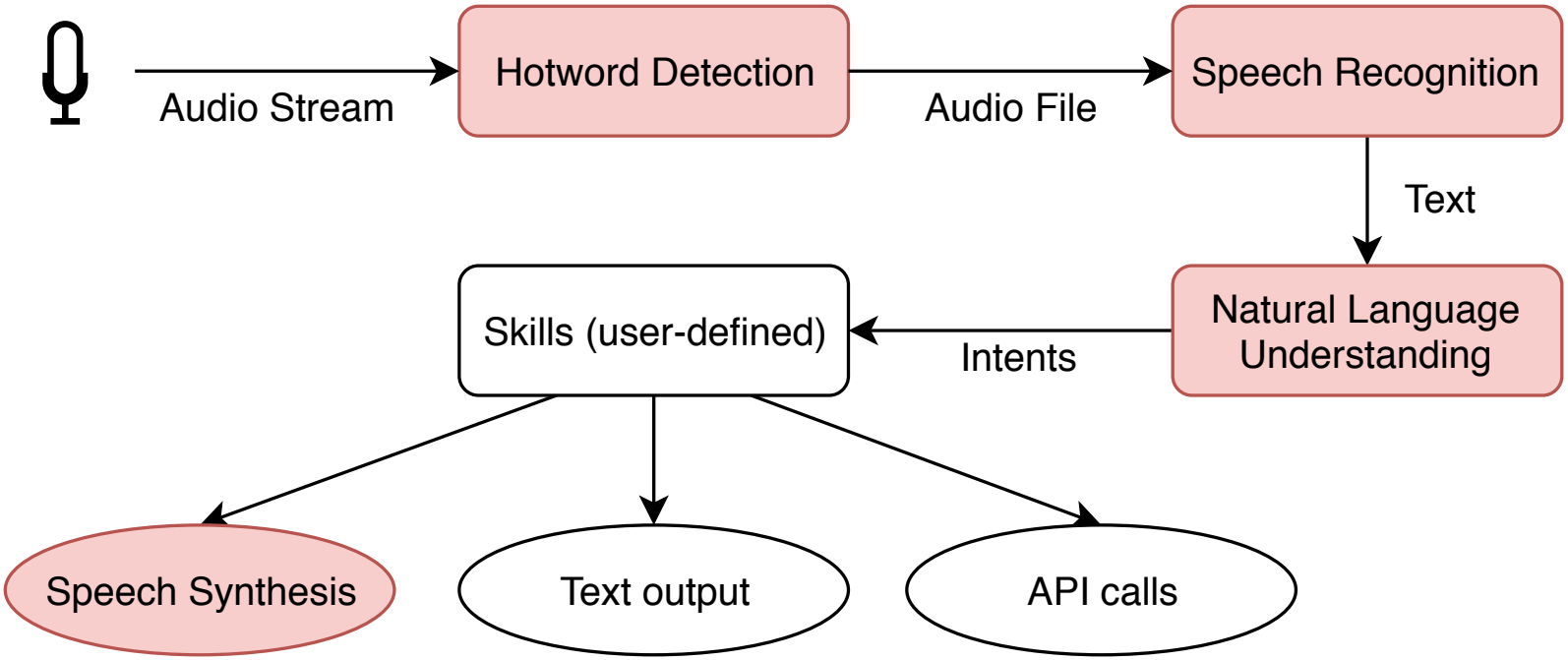
# About Snips

snips

Snips is a french artificial intelligence startup which makes voice assistants (like Alexa or Siri), but with two specific goals:

- Every assistant is **fully customizable**, from the choice of hotword to the intents that it recognizes.
- Every assistant is **private by design**, i.e. all queries that you ask to the assistant are never sent to a remote server (so all the computation must be done on device).

# Deep Neural Networks at Snips




Usage of Deep Neural Networks

# The problem

*Neural networks are used everywhere in voice assistants, and they must run on device (e.g. Raspberry Pi or smartphones).*

These devices have several limitations:

- Size of the inference runtime;
- Size of the stored trained models;
- Limited computational resources;
- Limited battery life;
- Special instruction set (ARMv7, ARMv8).

# About my internship

So traditional neural network libraries (e.g. TensorFlow, Theano, PyTorch, etc.) don't work well on these environments.

Several attempts at building specialized libraries:

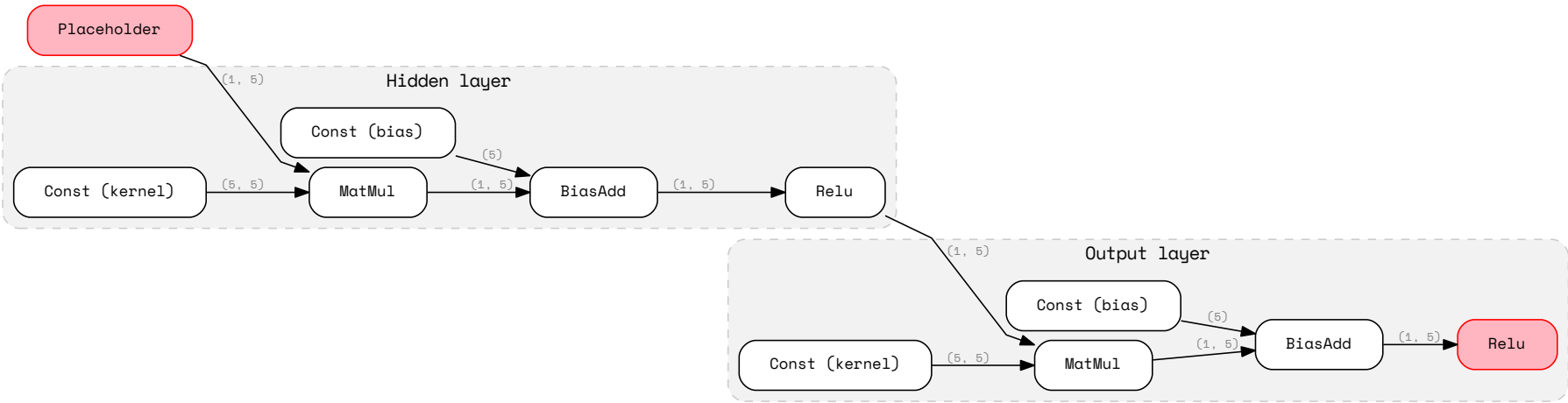
- TensorFlow Lite;
- XLA (ahead-of-time or just-in-time);
- TFDeploy, by the Snips Device Engineering team.
  Written in Rust, a novel systems programming language.

During this internship, I had to find ways to improve TFDeploy w.r.t. speed, robustness, and space- and energy-efficiency.

# Representing a neural network: dataflow graphs

TFDeploy is compatible with a subset of TensorFlow, so it borrows its internal representation of neural networks: **dataflow graphs**.

Here on a basic perceptron:




Each node has an *operation*, a set of input and output *ports* and a set of *attributes*. Edges go from output port of a node to input port of another.

# Profiling and visualization tools

Command-line profiler:

```
cargo run -- ../../models/classic.pb -s 41x40xf32 profile
```

Graph visualizer:

```
cargo run -- ../../models/classic.pb -s 41x40xf32 analyse --web
```

*Without hints about the input shape:*

```
cargo run -- ../../models/classic.pb analyse --web
```

# Problem statement

I first had to formalize and implement a **static analyser** for dataflow graphs (using ideas from abstract interpretation). Why?

- To clarify TFDeploy error messages;
- To catch as many errors as possible before runtime;
- To get hints about the edge shapes for the visualizer;
- To build optimizations, e.g. constant propagation and folding.

Essentially, we want to tag each edge of the graph with invariants about the tensors that flow through it. For instance:

- Every tensor flowing through this edge has datatype `float32`;

# Problem statement

I first had to formalize and implement a **static analyser** for dataflow graphs (using ideas from abstract interpretation). Why?

- To clarify TFDeploy error messages;
- To catch as many errors as possible before runtime;
- To get hints about the edge shapes for the visualizer;
- To build optimizations, e.g. constant propagation and folding.

Essentially, we want to tag each edge of the graph with invariants about the tensors that flow through it. For instance:

- Every tensor flowing through this edge has datatype `float32`;
- Every tensor flowing through this edge has the same value of $t$;

# Problem statement

I first had to formalize and implement a **static analyser** for dataflow graphs (using ideas from abstract interpretation). Why?

- To clarify TFDeploy error messages;
- To catch as many errors as possible before runtime;
- To get hints about the edge shapes for the visualizer;
- To build optimizations, e.g. constant propagation and folding.

Essentially, we want to tag each edge of the graph with invariants about the tensors that flow through it. For instance:

- Every tensor flowing through this edge has datatype `float32`;
- Every tensor flowing through this edge has the same value of $t$;
- Every tensor flowing through this edge has rank 2 or more, and its first dimension is 5.

# Formalism: type, value and shape facts

## Definition

A **datatype fact** $f \in \mathcal{TF}$ is either:

- $\top_{\mathcal{TF}}$, which matches any possible datatype;
- $Only_{\mathcal{TF}}(T)$ for $T$ a datatype, which only matches a specific datatype;
- $\bot_{\mathcal{TF}}$, which signifies an error.

## Definition

A **value fact** $f \in \mathcal{VF}$ is either:

- $\top_{\mathcal{VF}}$, which matches any possible tensor;
- $Only_{\mathcal{VF}}(t)$ for $t$ a tensor, which only matches a specific tensor;
- $\bot_{\mathcal{VF}}$, which signifies an error.

# Formalism: type, value and shape facts

**Definition**

A **shape fact** is an element of $\mathcal{SF} \equiv \{0,\ 1\} \times (\mathbb{N} \cup \{\_\})^*$. The first element defines whether the shape is *closed* or *open*, and the second element is a sequence of either an integer – which matches a specific dimension – or "_" – which matches any dimension.

A few examples:

- "Every tensor has shape $(2, 2)$": `[2, 2]`;

# Formalism: type, value and shape facts

## Definition

A **shape fact** is an element of $\mathcal{SF} \equiv \{0,\ 1\} \times (\mathbb{N} \cup \{\_\})^*$. The first element defines whether the shape is *closed* or *open*, and the second element is a sequence of either an integer – which matches a specific dimension – or "_" – which matches any dimension.

A few examples:

- "Every tensor has shape $(2, 2)$": `[2, 2]`;
- "Every tensor has rank 2": `[_, _]`;

# Formalism: type, value and shape facts

## Definition

A **shape fact** is an element of $\mathcal{SF} \equiv \{0,\ 1\} \times (\mathbb{N} \cup \{\_\})^*$. The first element defines whether the shape is *closed* or *open*, and the second element is a sequence of either an integer – which matches a specific dimension – or "_" – which matches any dimension.

A few examples:

- "Every tensor has shape $(2, 2)$": `[2, 2]`;
- "Every tensor has rank 2": `[_, _]`;
- "Every tensor has rank $\geq 2$ and first dimension 5": `[5, _, ..]`.

# Formalism: type, value and shape facts

**Definition**

A **shape fact** is an element of $\mathcal{SF} \equiv \{0,\ 1\} \times (\mathbb{N} \cup \{\_\})^*$. The first element defines whether the shape is *closed* or *open*, and the second element is a sequence of either an integer – which matches a specific dimension – or "_" – which matches any dimension.

A few examples:

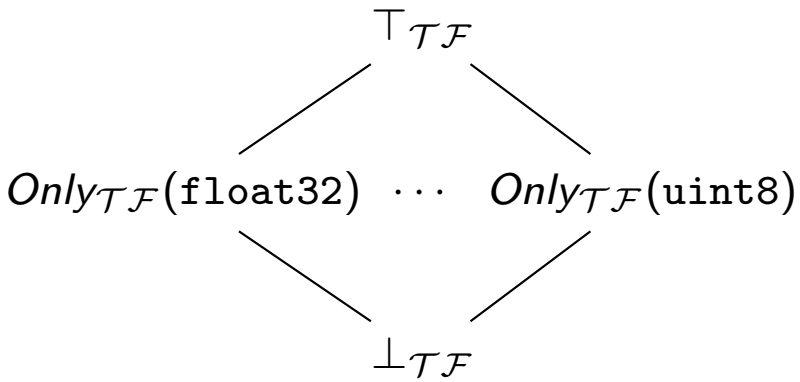
- "Every tensor has shape $(2, 2)$": `[2, 2]`;
- "Every tensor has rank 2": `[_, _]`;
- "Every tensor has rank $\geq 2$ and first dimension 5": `[5, _, ..]`.

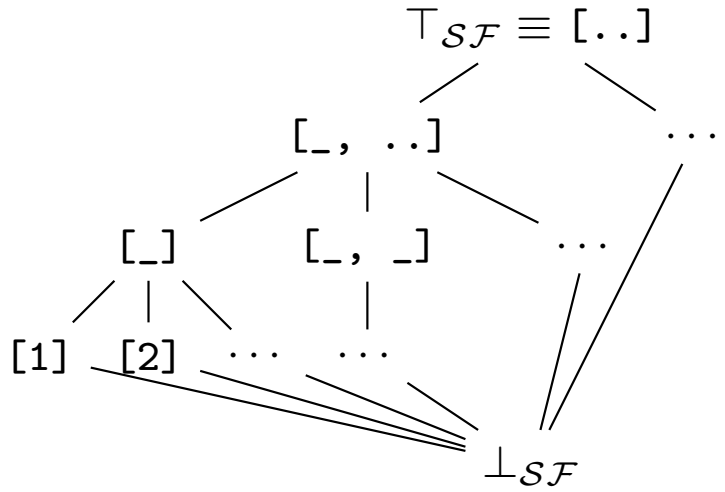
**Definition**

A **tensor fact** is an element of $\mathcal{F} \equiv \mathcal{TF} \times \mathcal{SF} \times \mathcal{VF}$.

# Formalism: Lattice structure and unification

From top to bottom: largest (least general) to smallest (most general).



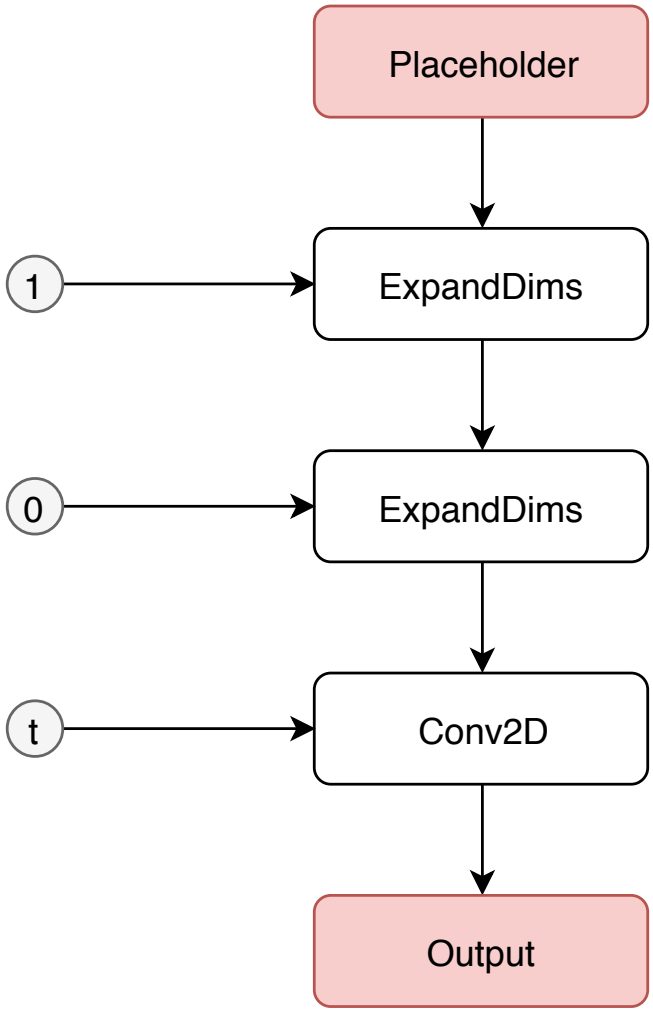




**Definition**

For a lattice $L$, the **unification** of $a, b \in L$, written as $a \sqcap_L b$, is the largest $c \in L$ such that $c \sqsubseteq_L a$ and $c \sqsubseteq_L b$, i.e. the *most general fact that combines the information from a and b*. It can be $\bot_L$ if $a$ and $b$ are incompatible facts.

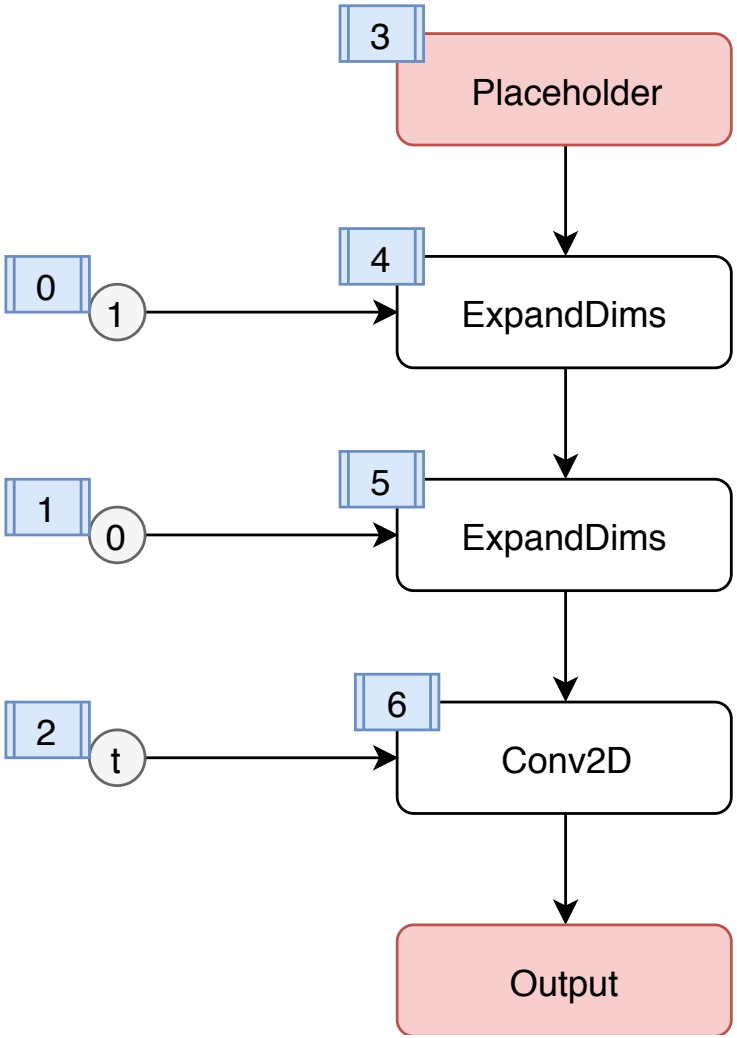
# Graph level: propagation algorithm

*Basic example of execution:*



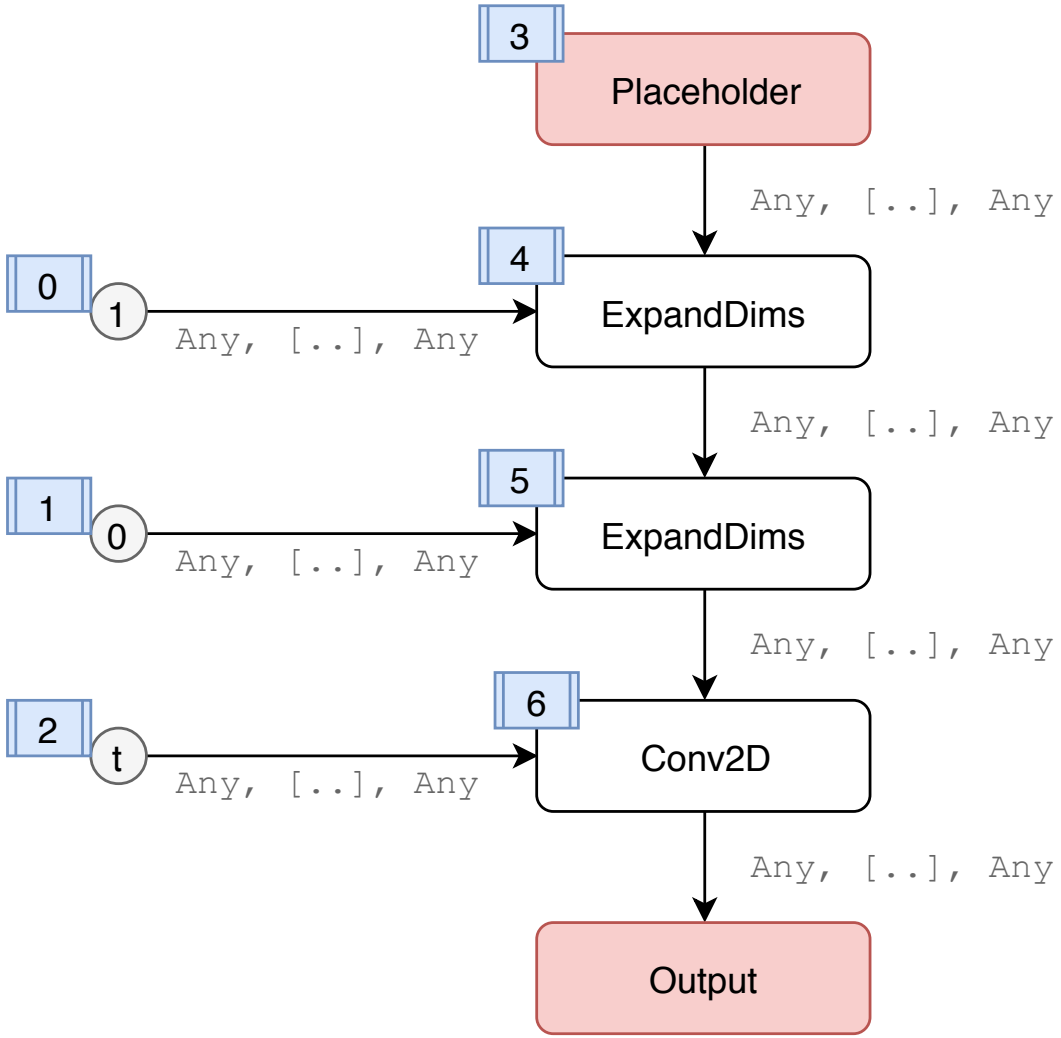
# Graph level: propagation algorithm

*Basic example of execution:*



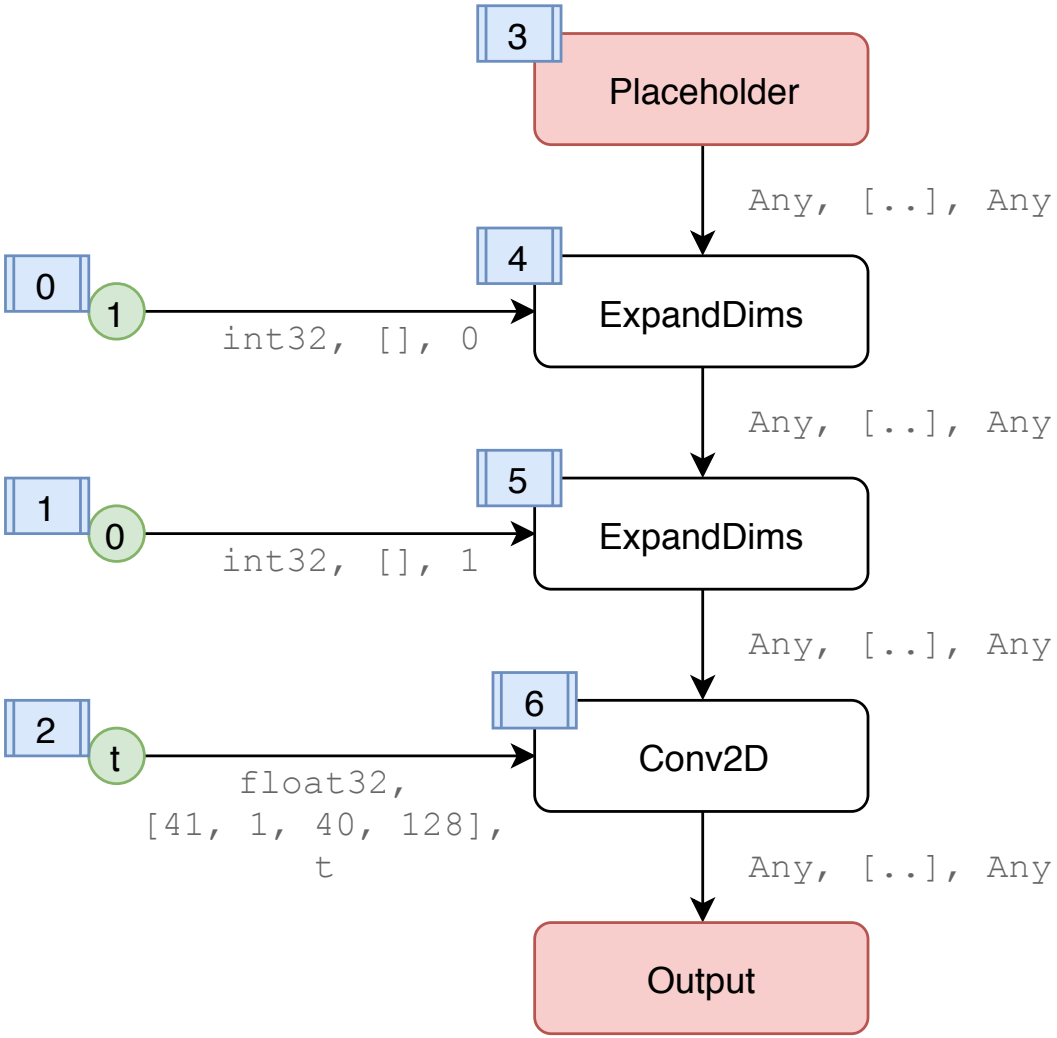
# Graph level: propagation algorithm

*Basic example of execution:*



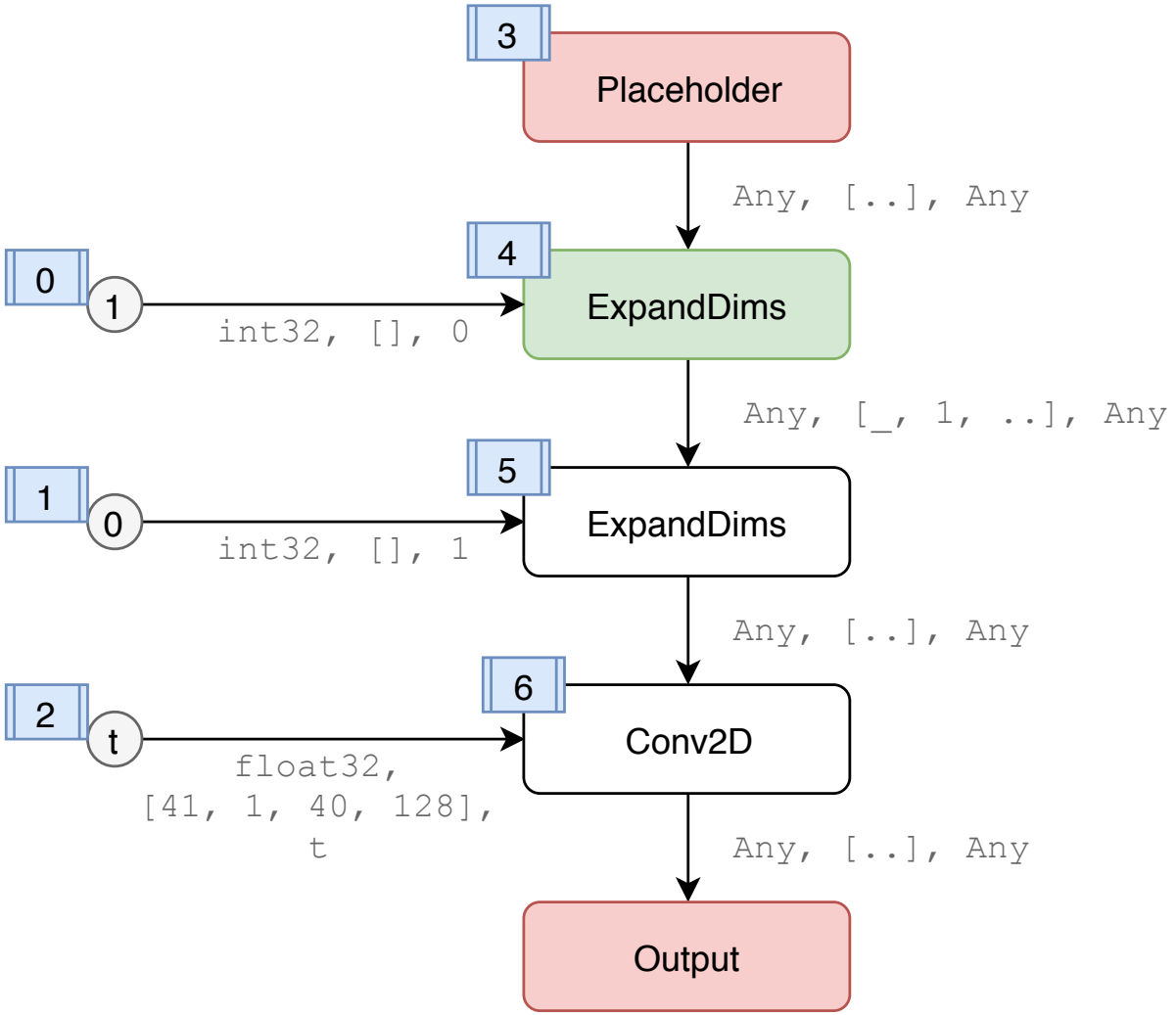
# Graph level: propagation algorithm

*Basic example of execution:*



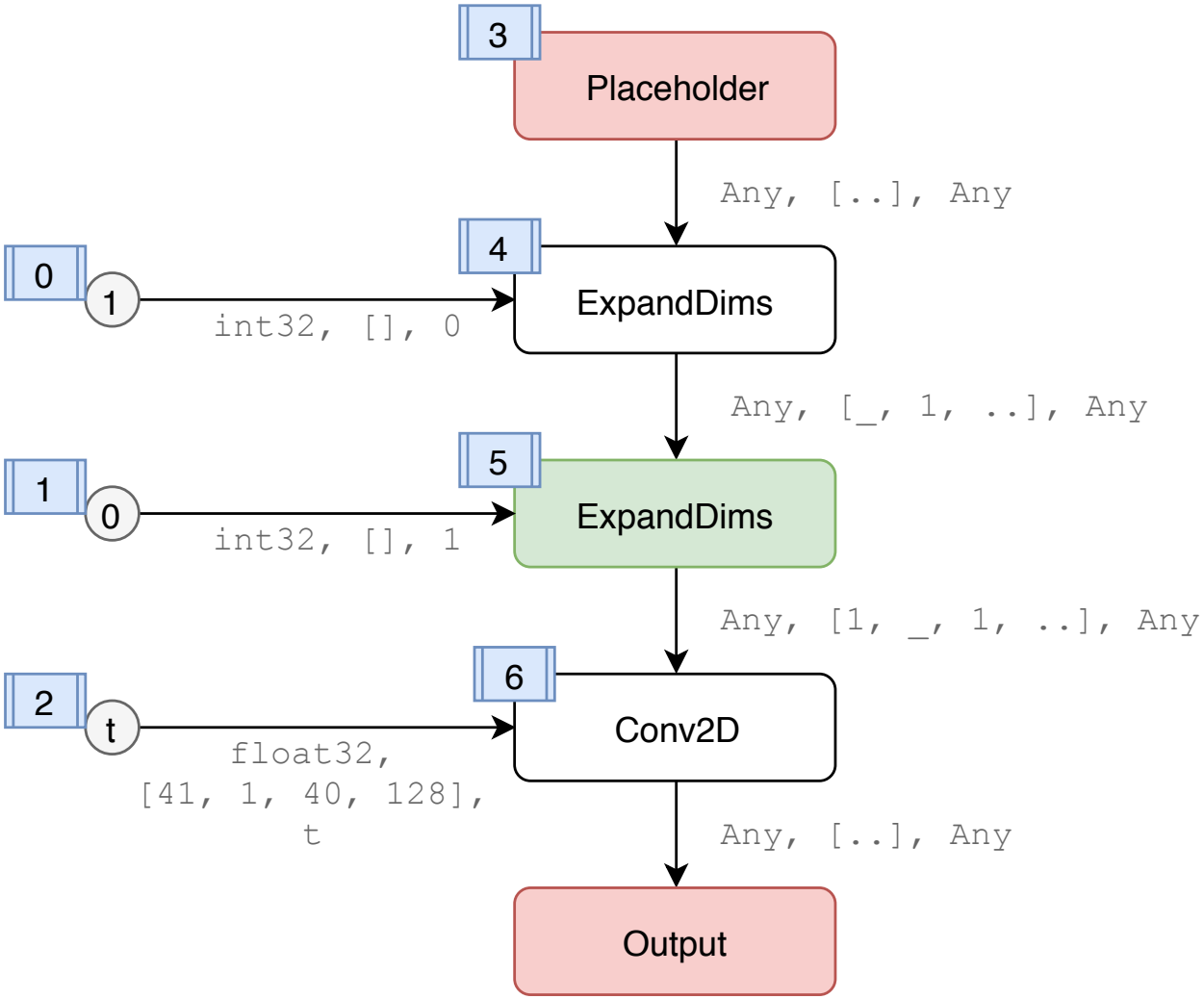
# Graph level: propagation algorithm

*Basic example of execution:*



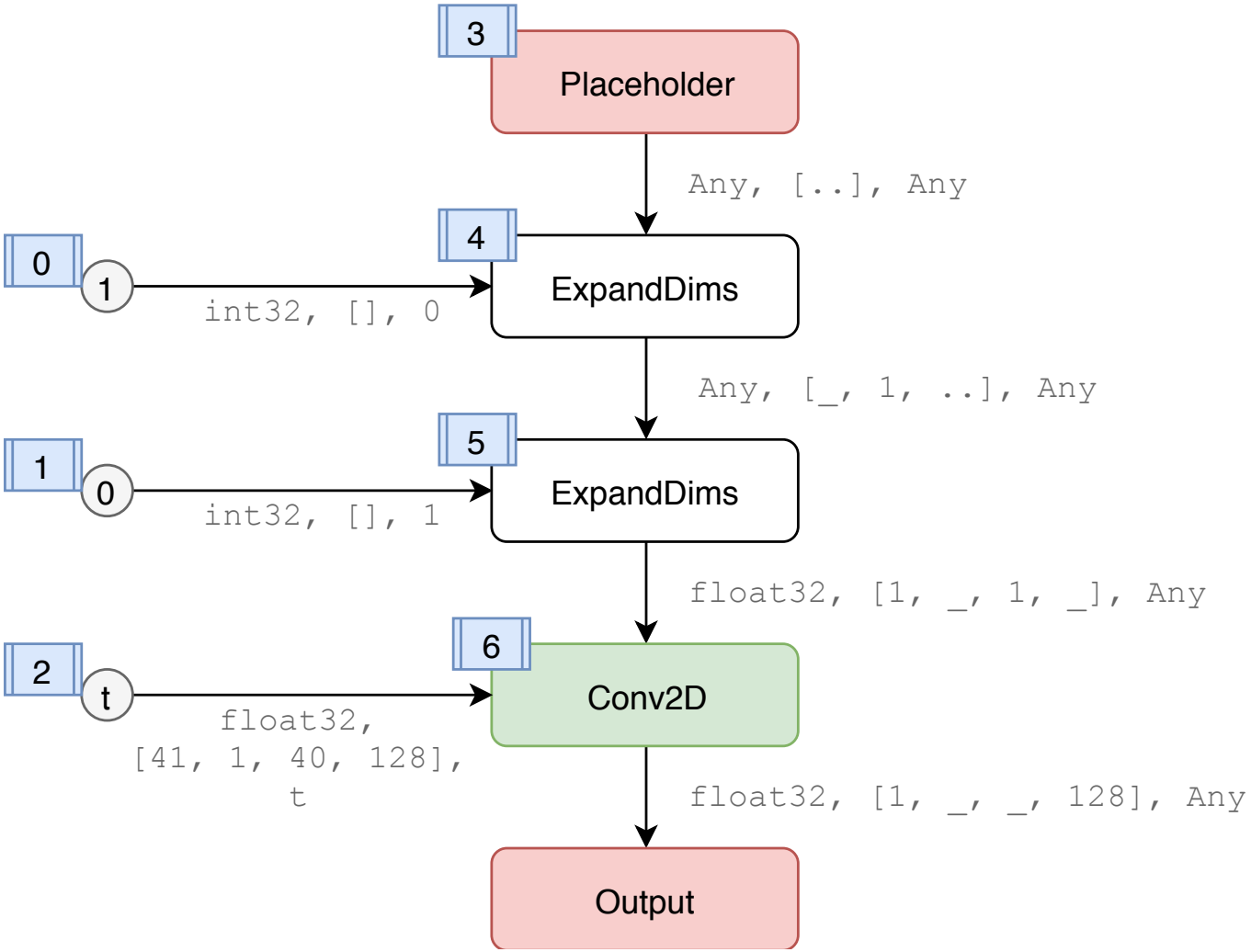
# Graph level: propagation algorithm

*Basic example of execution:*



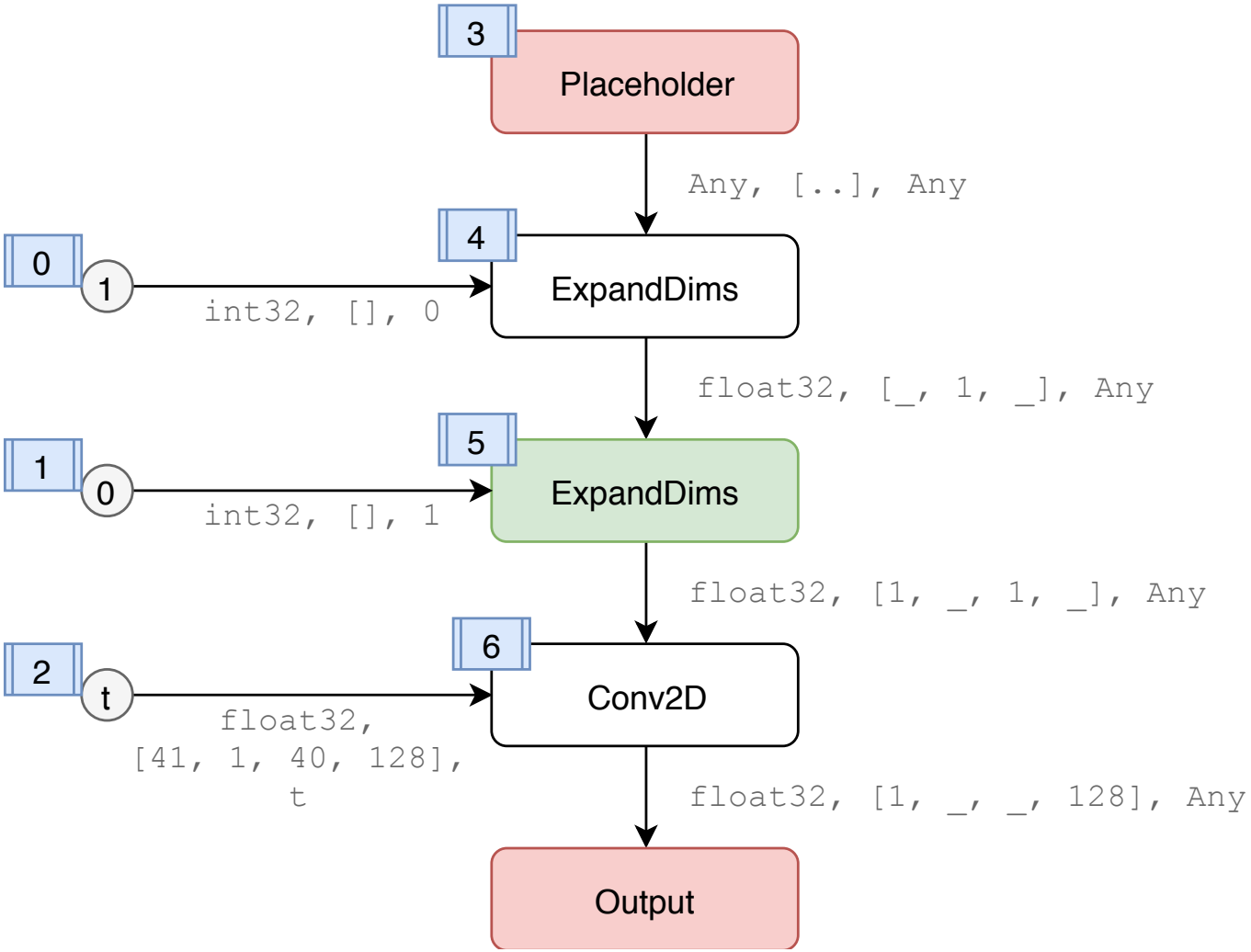
# Graph level: propagation algorithm

*Basic example of execution:*



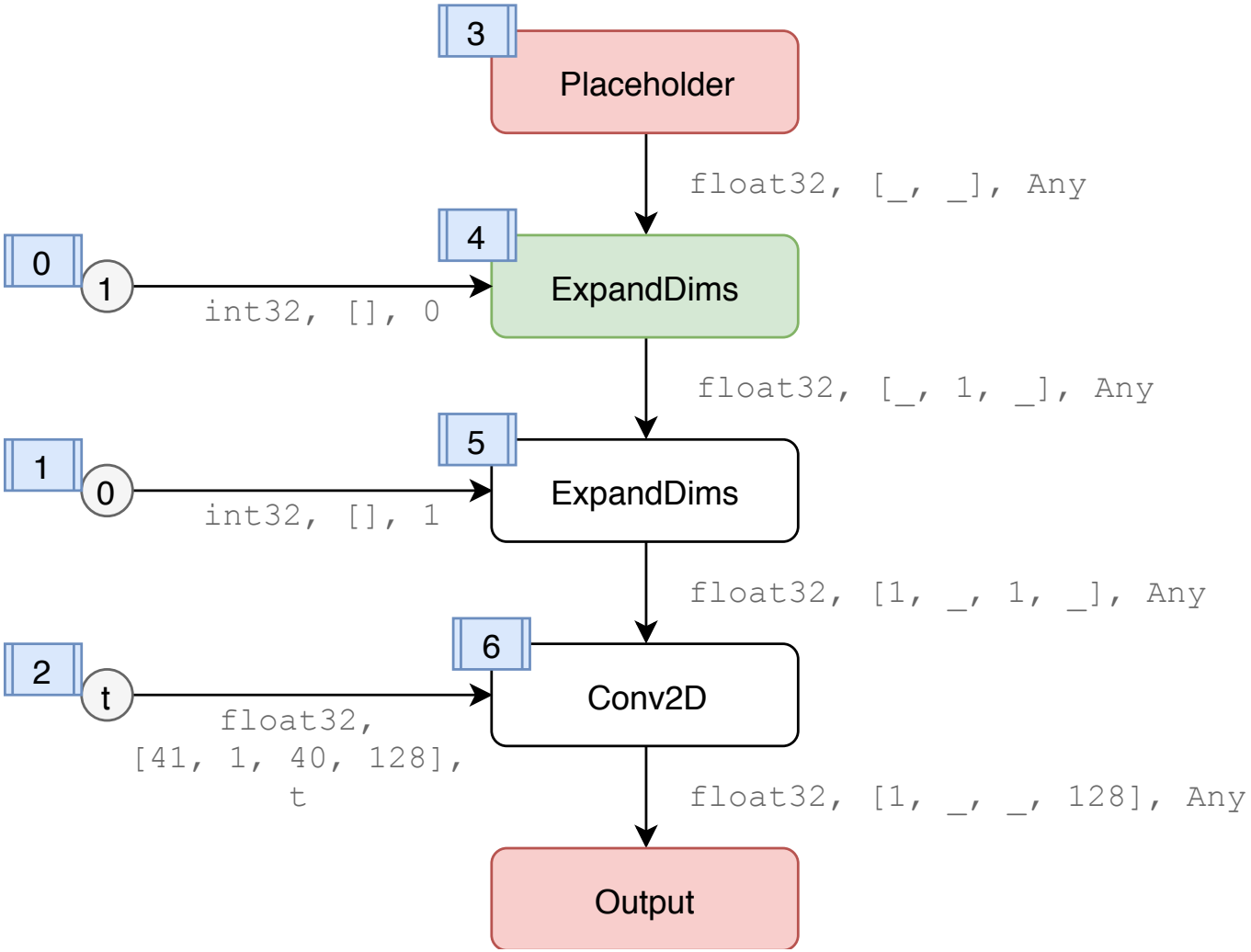
# Graph level: propagation algorithm

*Basic example of execution:*



# Graph level: propagation algorithm

*Basic example of execution:*



# Operation level: declarative constraint solver

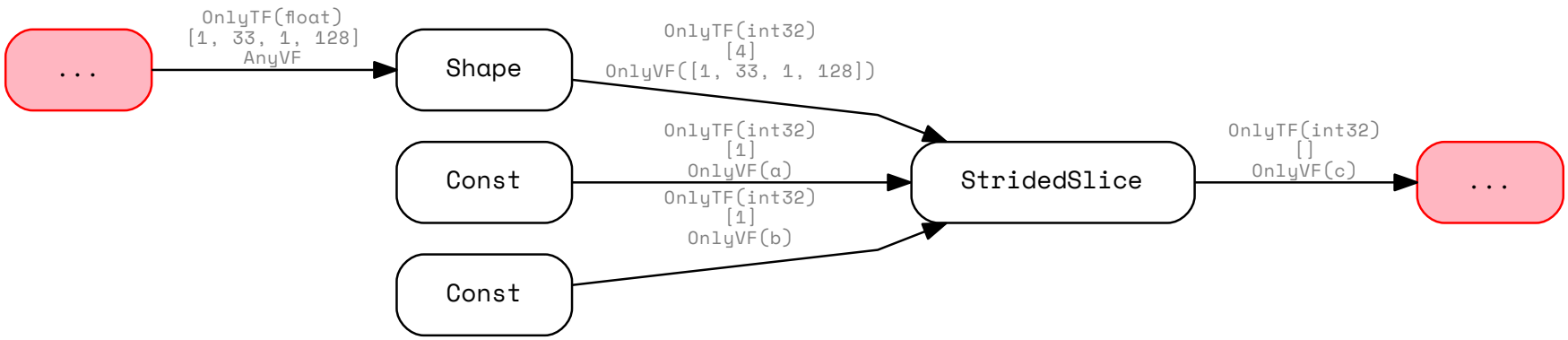
How do we easily (and generically) express the $enrich_{op}$ function?

```
fn rules(&self, inputs, outputs) { // For the Pad operation.
    let (input, paddings, output) = (&inputs[0], &inputs[1], &outputs[0]);
    solver
        .equals(&inputs.len, 2)
        .equals(&outputs.len, 1)
        .equals(&output.datatype, &input.datatype)
        .equals(&paddings.datatype, DataType::DT_INT32)
        .equals(&input.rank, &output.rank)
        .equals(&paddings.rank, 2)
        .equals(&paddings.shape[0], &input.rank)
        .equals(&paddings.shape[1], 2)
        .given(&input.rank, move |solver, rank: usize| {
            (0..rank).for_each(|i| {
                solver.equals_zero(wrap!(
                    (-1, &output.shape[i]),       (1, &input.shape[i]),
                    (1,  &paddings.value[i][0]), (1, &paddings.value[i][1])));
            })
        });
}
```

# An example of optimization: constant folding

To save space and prevent useless runtime computations, we can pre-compute constant parts of the graph and store the result instead.

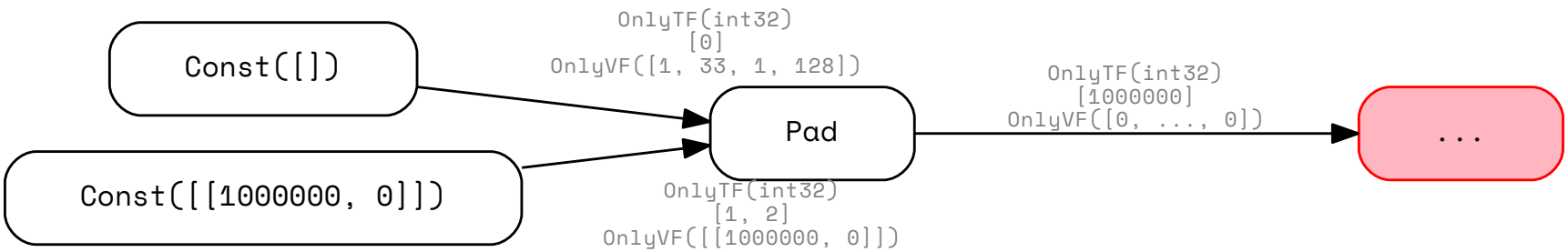
*Sometimes it makes sense:*



# An example of optimization: constant folding

To save space and prevent useless runtime computations, we can pre-compute constant parts of the graph and store the result instead.

*Sometimes it doesn't:*



# The issue with real-time data

How do you feed real-time data (e.g. samples from a microphone) to a neural network which expects a $[N,\ K]$-shaped tensor as an input?

$$I_i = \begin{pmatrix} s_i \\ s_{i+1} \\ \cdots \\ s_{i+N-1} \end{pmatrix} \in M_{N,K}(\mathbb{R})$$

- Naïve method: feed a new $I_i$ every time a new sample is recorded; *(But samples overlap, so useless computations).*
- Batched method: feed a new $I_i$ every $N$ new samples; *(But introduces latency and reduces accuracy).*
- Ideally, feed each new sample to the network separately. *(But not supported by traditional semantics of inference.)*

# New streaming semantics

So we define new semantics for dealing with real-time data.

- Every input of the network (i.e. `Placeholder` node) can have a **streaming dimension**. This information is propagated to the rest of the graph using the analyser.

# New streaming semantics

So we define new semantics for dealing with real-time data.

- Every input of the network (i.e. `Placeholder` node) can have a **streaming dimension**. This information is propagated to the rest of the graph using the analyser.
- Instead of thinking about tensors, we think about **chunks** (which have size 1 along the streaming dimension);

# New streaming semantics

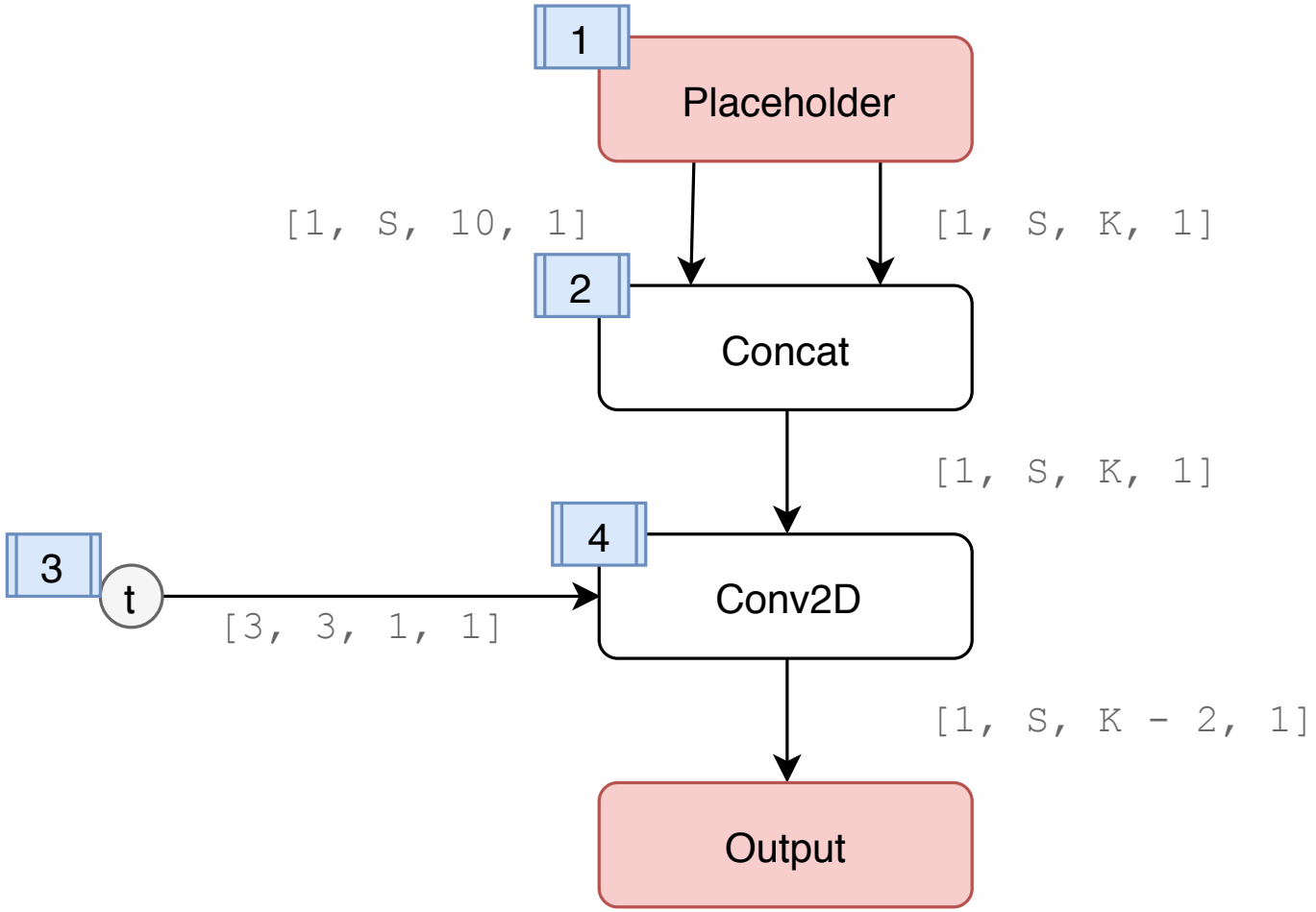
So we define new semantics for dealing with real-time data.

- Every input of the network (i.e. `Placeholder` node) can have a **streaming dimension**. This information is propagated to the rest of the graph using the analyser.
- Instead of thinking about tensors, we think about **chunks** (which have size 1 along the streaming dimension);
- Each operation is associated with a new **streaming evaluation function** which takes a chunk from one of the ports and might return a new chunk, and *remembers data from previous iterations* using **buffers**:

$$step_{op} \ : \ \mathcal{P}(\mathcal{A}_{op}) \ \times \ \mathcal{B}_{op} \times 0, \ n_i \times (\{\bot\} \cup \mathbb{N}) \times \mathcal{T} \ \rightarrow \mathcal{B}_{op} \times (\{\bot\} \cup \mathcal{T}^{n_o})$$

# Streaming inference algorithm

*Example of execution on the board:*

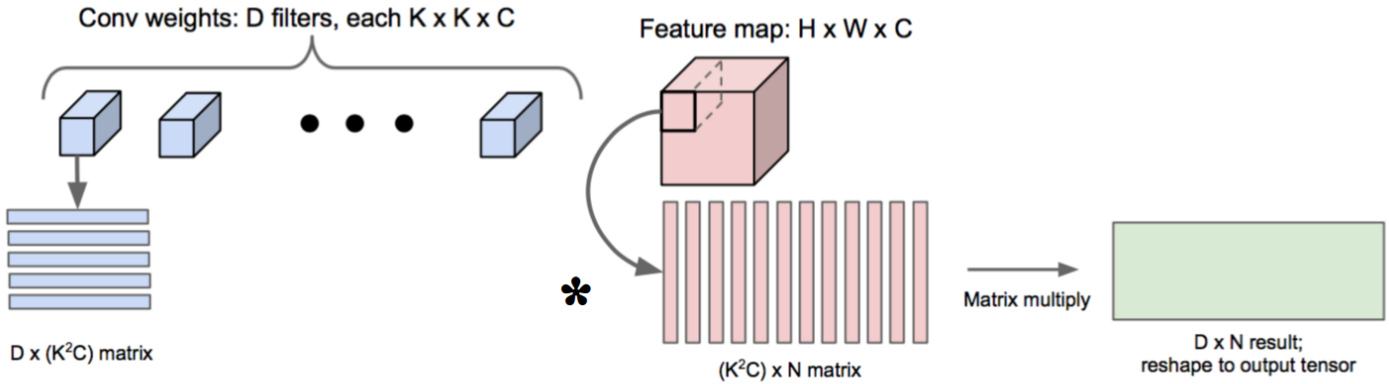


# How do we implement fast convolution?

2-D convolution *(actually cross-correlation)* is usually the most time consuming operation during CNN inference.

So the implementation matters:

- **Naïve approach**: slide the filter and perform every matrix product separately;
- **Im2col** approach: single matrix product of two well-built matrices:




*(Courtesy of Leonardo Araujo dos Santos.)*

- **Im2col** approach with optimized GEMM *(e.g. from GotoBLAS or BLIS)*;
- **FFT**-based approaches: $f \times g = \mathcal{F}^{-1}\left(\mathcal{F}(f) \cdot \mathcal{F}(g)\right)$

# Conclusion

During this internship, I improved the speed, space- and energy-efficiency and robustness of TFDeploy via optimizations built on top of a static analyser for dataflow graphs; via new inference semantics for streaming data; and via a careful choice of the convolution implementation.

I also learned to tame Rust, and got a sense of how it feels doing research and implementation in a production environment.

Possible future improvements:

- Make the declarative rule solver more expressive;
- Improve the performance of static analysis;
- Reduce the overhead on streaming evaluation;
- Custom implementation of im2col-based convolution to better control memory usage and speed.