Master in Computer Science
École Normale Supérieure de Lyon

Internship Report
ROMAIN LIAUTAUD

Optimizing neural network inference on embedded devices

And discovering the wonderful world of Rust in the process

MAY-JULY 2018 – AT SNIPS PARIS
UNDER THE SUPERVISION OF MATHIEU POUMEYROL

ABSTRACT

During my three-month internship in the Device Engineering team at Snips, I was tasked with improving the performance, efficiency and overall robustness of TFDeploy [18], their in-house port of TensorFlow inference to embedded devices. This report is a summary of this work, which involved topics ranging from static analysis and compiler optimizations to assembly programming.

TABLE OF CONTENTS

Introduction

Snips is a french artificial intelligence startup which makes voice assistant solutions. Their voice assistants are not unlike Alexa, Siri or Google Home; except for two key differences:

- First of all, they are entirely customize, meaning that the user can choose the hotword which will trigger the assistant – e.g. "Hey Snips!"; the intents that the assistant will be able to understand; and the way the assistant will respond to these intents.
- And, most importantly, they are *private by design*, meaning that they never send audio recordings of the user to a remote server for processing. This implies that the entire processing – which includes hotword detection, speech recognition, natural language understanding and speech synthesis – must happen directly on the device that recorded the audio, which can be anything from a PC or a smartphone to a Raspberry Pi.

To achieve this, Snips makes heavy use of machine learning and deep neural networks in particular: once the user has specified how he wants its assistant to behave, Snips trains several neural networks – for hotword recognition, speech recognition, natural language understanding, etc. – to match this behavior, and lets the user download these trained models on their device so that they can be run there.

One of the tasks of the Device Engineering team, in which I was doing my internship, is to make sure that these pre-trained models run as well as possible on embedded devices such as smartphones or single-board computers like the Raspberry Pi. Indeed, these devices are usually limited in computing power, memory and disk space; and some of them run even run on batteries, making energy efficiency an important factor.

However, most of the many open source machine learning libraries available online were not designed with embedded devices in mind, but rather with data centers and giant clusters of high-powered GPU machines. For instance, when compiled, the TensorFlow [7] library embarks code to both train and infer – i.e. evaluate – neural networks on CPUs and GPUs; and so the library takes way too much time to compile and too much disk space for what is actually needed: inference on tiny CPUs.

There have been several initiatives to improve this situation:

- There is now a separate, trimmed-down build of TensorFlow [2] for smartphones;
- There are attempts at using ahead-of-time compilation [8] and just-in-time compilation [9] to improve the inference speed of neural network models expressed with TensorFlow;
- More recently, Google has released a preview of TensorFlow Lite [1], an inference-only subset of TensorFlow optimized for embedded devices.

One of these initiatives is TFDeploy [18], a TensorFlow-compatible inference library for embedded devices written by Snips' Device Engineering team in Rust [10], a new systems programming language which promises to be as fast as C or C++ while preventing segfaults; guaranteeing thread safety; providing powerful abstractions – like a ML-based type system with pattern matching, or iterators – without runtime costs; and much more.

TFDeploy was designed with several goals in mind: first, the compiled inference runtime should take up as little disk space as possible; and the library should also help reduce the disk space used by pre-trained models. On a given model, TFDeploy inference should run as fast as TensorFlow inference; and ideally faster – by optimizing the inference code for architectures like ARM. My task, during this internship, was to help TFDeploy meet these goals.

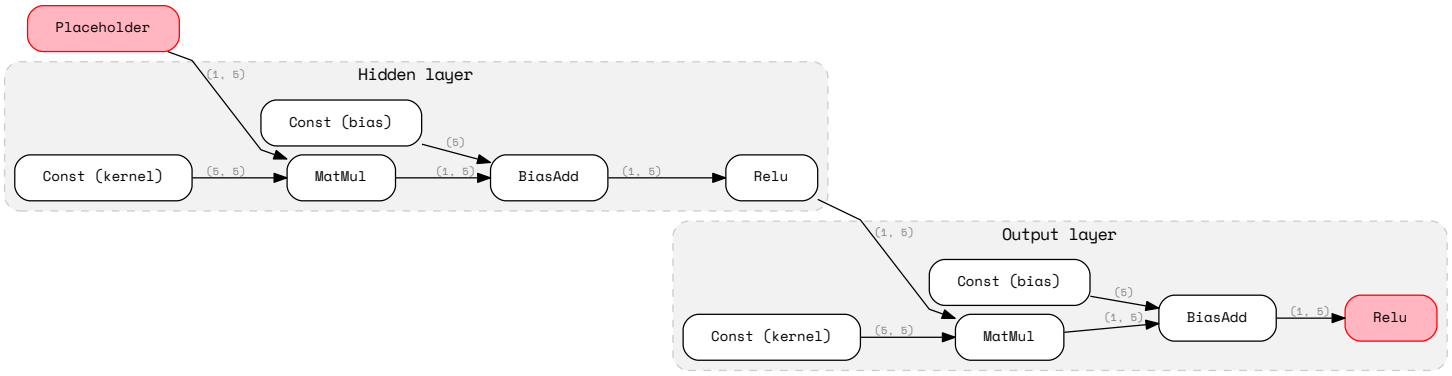
In this report, I will present a static analyser for dataflow graphs which helped reduce the disk space and inference time of pre-trained models; as well as new inference semantics which helped avoid redundant computations when dealing with streamed – e.g. audio – input data.

TensorFlow and its dataflow graph.

Among the many open source deep learning libraries available online, TensorFlow [7] is by far the most popular – with more than 100,000 stars on GitHub at the time of writing. Under active development by Google since 2015, it lets developers define, train and evaluate most types of deep neural networks – including convolutional and recurrent networks – but can also be used more generally as a framework for high performance numerical computation.

But, more than the library itself, we're actually mainly interested TensorFlow's internal representation of neural networks and numerical computations in general: the **dataflow graph**. As explained in the TensorFlow documentation [5], this graph is used to represent any computation in terms of the dependencies between individual operations – e.g. element-wise matrix operations, matrix products or 2-D convolutions. As an example, Fig. 1 shows the dataflow graph of a single-layer perceptron.

Figure 1: Dataflow graph of a single-layer perceptron with bias and ReLU activation.



Let us start by formalizing this internal representation. In TensorFlow, the basic unit of computation is the **tensor**, which is essentially a generalization of vectors and matrices to any dimension and datatype.

Definition 1. A **tensor** $t \in \mathcal{T}$ is an n-dimensional array of scalars, with $n \in \mathbb{N}$.

Every tensor has the following properties:

- A **datatype**, e.g. `float32` or `uint8`, which is shared by all the scalars inside the tensor;
- A **rank**, which is the dimensionality of its underlying array (here the integer n);
- A **shape**, which is an n-tuple of integers that represents the size of each dimension.

Note that a 0-ranked tensor, or equivalently a tensor with shape `()`, is simply a scalar.

Definition 2. A **dataflow graph** $G = (V_G,\ E_G,\ op,\ n_i,\ n_o,\ a)$ is a directed acyclic multigraph.

Note that, in the general case, dataflow graphs are not necessarily acyclic; but since the scope of this report – and this internship – is limited to convolutional neural networks, we will make the simplifying assumption that they don't contain any cycles.

Every node in $v \in V$ has:

- An operation $op(v)$, e.g. `Const`, `Add` or `Conv2D`, which defines the behavior of the node;

- A map $a(v)$ of attributes with keys from the set $\mathcal{A}_{op(v)}$ of possible keys for the operation;
- A set of $n_i(v)$ input ports and $n_o(v)$ output ports;

Some nodes have the special `Placeholder` operation, which is not an actual operation but instead a way to mark the nodes which will be replaced by the input data before inference.

Every edge $e \in E$ is a tuple $(v_s,\ p_s,\ v_d,\ p_d)$ with v_s and v_d the source and destination nodes of the edge respectively, $p_s \in [\![0,\ n_o(v_s)[\![$ the output port of the source node and $p_d \in [\![0,\ n_i(v_d)[\![$ the input port of the destination node used by the edge.

For convenience, let us define the incoming and outgoing edges of a node as:

$$e_i(v) = \bigcup_{p=0}^{n_i(v)} e_i(v,p) \qquad\qquad e_o(v) = \bigcup_{p=0}^{n_o(v)} e_o(v,p)$$

with the incoming *(resp. outgoing)* edges to *(resp. from)* an input *(resp. output)* port of a node as:

$$e_i(v,p) = \{(v_s,\ p_s,\ v,\ p) \in E \mid v_s \in V,\ p_s \in [\![0,\ n_o(v_s)[\![\}$$
$$e_o(v,p) = \{(v,\ p,\ v_d,\ p_d) \in E \mid v_d \in V,\ p_d \in [\![0,\ n_i(v_d)[\![\}$$

Let us now model the semantics of TensorFlow inference. Essentially, the semantics of every operation is given by its evaluation function, which takes a set of attribute values and a set of input tensors, and returns a set of output tensors. Running inference on a dataflow graph is then just a matter of finding an execution plan, replacing the `Placeholder` nodes with the input data, and, for every node in the plan, applying the evaluation function of its operation on the tensors of its input ports and putting the result on its output ports.

Definition 3. Every operation op is associated with an **evaluation function**:

$$eval_{op} \ :\ \mathcal{P}(\mathcal{A}_{op}) \ \times\ \mathcal{T}^{n_i} \ \rightarrow \mathcal{T}^{n_o}$$

with n_i and n_o respectively the number of input and output ports that op expects.

Here are several examples of evaluation functions:

- $eval_{\texttt{Const}}(attrs,\ ()) = attrs[\texttt{value}]$ (i.e. the value associated with the key `value` in *attrs*);
- $eval_{\texttt{Add}}(attrs,\ (a,b)) = (a+b)$ (with $a+b$ the element-wise sum of tensors a and b);
- $eval_{\texttt{AddN}}(attrs,\ inputs) = \left(\sum_{i=0}^{attrs[\texttt{n}]} inputs[i]\right)$
  Note how the operation can handle a variable number of inputs via the `n` *attribute.*

Definition 4. An **execution plan** for a dataflow graph G with output node $v^* \in V_G$, if it exists, is a topological ordering ord of $(V_G, \overline{E_G})$ such that:

$$\forall v \in V,\ op(v)\ =\ \texttt{Placeholder} \Rightarrow ord(v) <= ord(v^*)$$

With $\overline{E}$ the set of edges without the input and output port information:

$$\overline{E} = \{(v_s,\ v_d)\ \in V \times V \mid \exists p_s \in [\![0,\ n_o(v_s)[\![,\ p_d \in [\![0,\ n_i(v_d)[\![,\ (v_s,\ p_s,\ v_d,\ p_d) \in E\}$$

Definition 5. The **inference function** $infer_G : V_G \times \mathcal{T} \to \mathcal{T}$ of a dataflow graph G is defined as the return value of the following algorithm, which takes an output node $v^* \in V_G$ and an input tensor $t^* \in \mathcal{T}$, and returns the result at v^* of the computation described by G in $O(V_G + E_G)$ *(ignoring the complexity of computing the different evaluation functions)*.

```
1  function InferGraph(G, v*, t*)
2      ord ← ComputePlan(G, v*)
3
4      ▷ We tag each edge of G with a tensor (which doesn't yet have a value).
5      tensors ← [null for e ∈ E]
6
7      for v ∈ ord do
8          if op(v) = Placeholder then
9              for e ∈ e_o(v) do
10                 tensors[e] ← t*
11         else
12             inputs ← [tensors[e] for e ∈ e_i(v)]
13             outputs ← eval_{op(v)}(a(v), inputs)
14
15             ▷ The eval function returns a tuple of tensors, one for each output port.
16             for output ∈ outputs do
17                 p ← index(output, outputs)
18                 for e ∈ e_o(v, p) do
19                     tensors[e] ← output
20
21         ▷ When we reach the output node, return its outgoing tensors.
22         if v = v* then
23             return [tensors[e] for e ∈ e_o(v)]
```

1 Profiling and visualization tools

Although I already clarified what we meant by "*optimizing* inference" – we wanted to make it faster, more energy- and space-efficient and more robust – I didn't specify how we were planning on doing that. The reason is simple: we didn't actually know, in the beginning, which part of the inference process could be optimized. We had hunches, of course, but we didn't want to act on them without proof to avoid premature optimization.

Consequently, we needed a proper set of tools to visualize dataflow graphs and profile their inference on both TFDeploy [18] and the reference TensorFlow implementation [7]; and that is what I did during the first few weeks of my internship. Although it was not really scientifically challenging work, it helped me get familiar with the TFDeploy codebase and with Rust [10] – the programming language it's written in – whose memory ownership model is a bit unsettling at first. In hindsight, spending time writing these tools first also saved me a lot of time in the long run, as I was able to work much faster using them.

1.1 Command-line profiler.

I started by writing a command-line interface [14] that allowed me to, among other things, display general information about a dataflow graph, profile the execution of a graph using randomly generated data, or compare the values returned by TFDeploy and the reference implementation to check for correctness. Fig. 2 shows an excerpt of the command-line output when profiling the execution of a graph. Notice how we distinguish the *real* time – which is the actual time elapsed, as measured by a clock on the wall – from the *user* and *sys* times – which correspond to the amount of CPU time spent in user-mode code and kernel calls respectively. This distinction was useful to pinpoint what exactly was taking up time, especially in multi-threaded contexts.

Figure 2: Partial output of the `./cli conv2d.pb -s 61x60xf32 profile` command.

```
Summary for ../tests/models/conv2d.pb:
======================================

Most time consuming nodes:
+-------+-----------------------------------------------------------------------------------------+
|   6   | Operation: Conv2D          | Name: Conv2D                                               |
+-------+-----------------------------------------------------------------------------------------+
        |  Real: 29.316 ms/i (99.9%) | User: 29.311 ms/i (99.9%) | Sys: 0.007 ms/i (100.0%)      |
        +-----------------------------------------------------------------------------------------+
        | Attribute strides: list {i: 1 i: 1 i: 1 i: 1}                                           |
        | Attribute T: type: DT_FLOAT                                                             |
        | Attribute data_format: s: "NHWC"                                                        |
        | Attribute use_cudnn_on_gpu: b: true                                                     |
        | Attribute padding: s: "VALID"                                                           |
        +-----------------------------------------------------------------------------------------+

Total execution time (for 6 nodes):
- Real: 29.335 ms/i.
- User: 29.330 ms/i.
- Sys: 0.007 ms/i.

Most time consuming operations:
- Conv2D (for 1 nodes):          - ExpandDims (for 2 nodes):     - Const (for 3 nodes):
    - Real: 29.316 ms/i (99.94%).    - Real: 0.017 ms/i (0.06%).     - Real: 0.001 ms/i (0.00%).
    - User: 29.311 ms/i (99.94%).    - User: 0.017 ms/i (0.06%).     - User: 0.001 ms/i (0.00%).
    - Sys: 0.007 ms/i (99.98%).      - Sys: 0.000 ms/i (0.00%).      - Sys: 0.000 ms/i (0.02%).
```

Getting accurate measurements was actually harder than I expected: as most of the events that I had to profile were taking less than 20 ms (sometimes down to 0.01 ms), slight variations in CPU clock rate – e.g. because of throttling – or even the measurement code itself could have a noticeable impact on the results. This was solved by adding a number of "warm-up iterations" that weren't measured but forced the CPU out of throttling, as well as measuring enough iterations to detect and remove outliers.

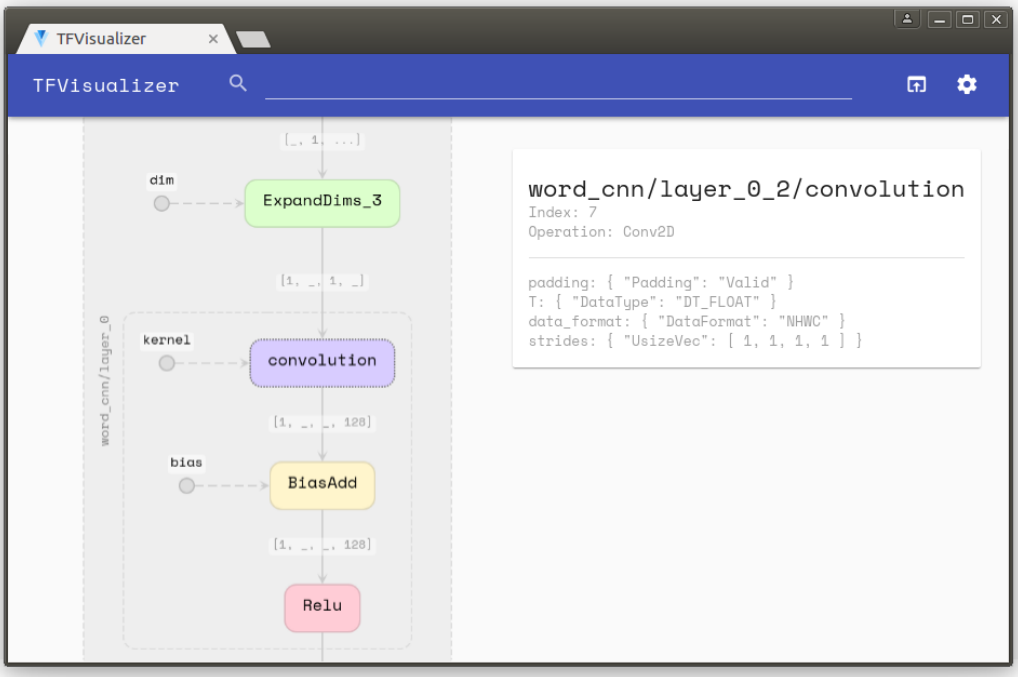
1.2 Dataflow graph visualizer.

After playing around with a few real-world CNN – convolutional neural network – models that the Snips hotword recognition team provided me – with more than 3000 nodes for some of them – I realized that I needed a better way to visualize dataflow graphs than a simple Graphviz output. Drawing inspiration from TensorBoard [3] – which was unfortunately incompatible with the graph format used by TFDeploy – I wrote a visualization tool [15] which:

- allows you to group related parts of the graph and collapse them for better readability;
- shows the properties of nodes and edges;
- displays the hints provided by the static analyser *(see Sec. 2)*.

Although there are many existing graph layout algorithms to choose from, such as [13] which is famously used by Graphviz or [12] which simulates a set of repulsive and attractive forces and tries to find a stable equilibrium, none of them gave me satisfying results – as Appendix A demonstrates. So I decided to design one specifically for displaying dataflow graphs – which are essentially directed acyclic graphs with a few subtleties depending on the type of each node. To this end, I translated my graph layout problem into a set of linear constraints on the x and y coordinates of each node in the graph, which I then solved using a Javascript implementation [19] of the Cassowary constraint solving algorithm [11]. Fig. 3 shows the output of the visualizer on the same model as in Appendix A.

Figure 3: Output of the visualizer on a hotword detection model used at Snips.



2 Static analysis of dataflow graphs

My first *actual* task was formalizing and implementing a **static analyser** for dataflow graphs, drawing inspiration from the abstract interpretation formalism *(see [17] and [16])*. This came as the natural answer to several problems:

- First, the error messages displayed by TFDeploy were not explicit enough, for instance when the shape of the input data was incompatible with the shapes that the operations in the graph expected;
- What's more, all these errors were triggered at runtime, and we wanted a way to catch as many of them as possible ahead of time to avoid unpredictable runtime behavior;
- We also wanted to improve the graph visualization tool [15] that I mentioned earlier by annotating edges with information about the tensors flowing through them – e.g. their datatype, shape and value;
- Finally, we needed a framework on top of which we could build ahead-of-time optimizations like constant folding and propagation.

Simply put, the role of this analyser is to tag every edge in the graph with invariants about the tensors that flow through it – specifically their datatype, their shape and possibly their value. To do so, we will supplement each TensorFlow operation with a set of rules that express constraints on the datatype, shape and value of its input and output tensors; and we will then propagate those constraints – along with constraints about the input of the model – to the entire graph until we find those invariants.

In the following sections, I will give a formal overview of the analyser; describe the propagation algorithm used to find the edge invariants; present the declarative solver that I designed to help express constraints about the input and output tensors of operations; and quickly present some of the optimizations that I was able to implement using this static analyser.

2.1 Formal overview.

As we want to tag the edges of the dataflow graph with invariants about tensors, we must first formalize what these invariants can be. Here are a few of the things we might want to express:

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

To do so, we first define datatype, shape, value and ultimately tensor **facts**. They are analogous to the abstract values that are used in abstract interpretation.

Definition 6. 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 *(e.g.* $Only_{\mathcal{TF}}(\texttt{float32})$*)*;
- $\bot_{\mathcal{TF}}$, which signifies an error.

In the code, $\top_{\mathcal{TF}}$ is called `TypeFact::Any` and $Only_{\mathcal{TF}}(T)$ is called `TypeFact::Only`.

Definition 7. 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.

In the code, $\top_{\mathcal{VF}}$ is called `ValueFact::Any` and $Only_{\mathcal{VF}}(t)$ is called `ValueFact::Only`.

Definition 8. 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.

For the sake of readability, we will write closed shape facts like regular sequences, and open shape facts with "`..`" in the end. While closed shape facts match an entire shape, an open shape fact only matches the beginning of a shape. So, for instance, the closed shape fact `[1, _]` only matches the shapes $(1,\ k)$ with $k \in \mathbb{N}$, whereas the open shape fact `[1, _, ..]` matches $(1,\ k)$ with $k \in \mathbb{N}$ as well as $(1,\ k,\ k')$ with $k, k' \in \mathbb{N}$, and so on.

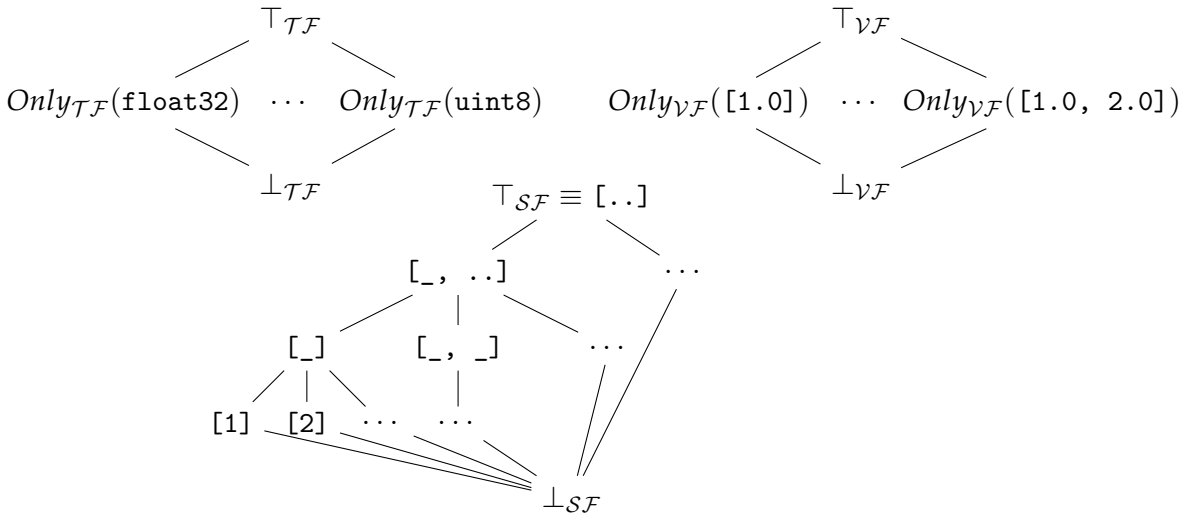
This definition is a bit trickier than the previous ones, but is necessary to represent partial rank and shape information with enough precision. Looking back at our examples:

- "Every tensor has shape $(2, 2)$" can be represented by `[2, 2]`;
- "Every tensor has rank 2" can be represented by `[_, _]`;
- "Every tensor has rank ≥ 2 and first dimension 5" can be represented by `[5, _, ..]`.

Definition 9. A **tensor fact** is an element of $\mathcal{F} \equiv \mathcal{TF} \times \mathcal{SF} \times \mathcal{VF}$.

Now that we have a way to represent partial information (i.e. invariants) about tensors, we need to define a few well-chosen orders $\sqsubseteq_{\mathcal{TF}}$, $\sqsubseteq_{\mathcal{SF}}$, $\sqsubseteq_{\mathcal{VF}}$ and ultimately $\sqsubseteq_{\mathcal{F}}$ on our facts to give them a lattice structure. For the sake of clarity, we will not do that formally, but rather give excerpts of the Hasse diagrams for $\mathcal{TF}$, $\mathcal{SF}$ and $\mathcal{VF}$ on Fig. 4, and define that $(t_1,\ s_1,\ v_1) \sqsubseteq_{\mathcal{F}} (t_2,\ s_2,\ v_2)$ iff. $t_1 \sqsubseteq_{\mathcal{TF}} t_2$, $s_1 \sqsubseteq_{\mathcal{SF}} s_2$ and $v_1 \sqsubseteq_{\mathcal{VF}} v_2$.

Figure 4: Hasse diagrams for $\mathcal{TF}$, $\mathcal{SF}$ and $\mathcal{VF}$.



Definition 10. For a lattice L, we finally define the **unification** of $a, b \in L$, written as $a \sqcap_L b$, as the largest $c \in L$ such that $c \sqsubseteq_L a$ and $c \sqsubseteq_L b$ *(which always exists in a lattice)*.

In other words, $a \sqcap_L b$ is the most general fact that combines the information from both a and b – it can be $\bot_L$ if a and b are incompatible facts. For instance:

$$\begin{array}{rcll} \top_{\mathcal{TF}} & \sqcap_{\mathcal{TF}} & \mathit{Only}_{\mathcal{TF}}(\texttt{float32}) & = \mathit{Only}_{\mathcal{TF}}(\texttt{float32}) \\ \mathit{Only}_{\mathcal{VF}}(\texttt{[1]}) & \sqcap_{\mathcal{VF}} & \mathit{Only}_{\mathcal{VF}}(\texttt{[2]}) & = \bot_{\mathcal{TF}} \\ \texttt{[1, \_, ..]} & \sqcap_{\mathcal{SF}} & \texttt{[\_, \_, 3]} & = \texttt{[1, \_, 3]} \end{array}$$

In practise, we don't actually implement the lattices themselves, but directly the unification functions. These functions, along with the Rust type definitions for `TypeFacts`, `ShapeFacts`, `ValueFacts` and `TensorFacts`, can be found on Appendix B. Note that we never directly encounter $\bot$ in the code, but raise an error as soon as the $\bot$ value would be produced.

2.2 At the graph level: propagation algorithm.

Using the above formalism, we can now describe the propagation algorithm used to find the edge invariants. Essentially, we start by tagging every edge of the graph with $\top_{\mathcal{TF}}$ – the most general tensor fact possible. Then, in turn, each node in the graph adds more constraints to the tensor facts of its incoming and outgoing edges – we call this operation *enrichment* – until reaching a fixed point.

Definition 11. We implement, for each operation op, an **enrichment function**:

$$\mathit{enrich}_{op} \ : \ \mathcal{P}(\mathcal{A}_{op}) \times \mathcal{F}^{n_i} \times \mathcal{F}^{n_o} \to \mathcal{F}^{n_i} \times \mathcal{F}^{n_o}$$

with n_i and n_o respectively the number of input and output ports that op expects. This function takes the attributes of a node as well as the current facts about the node's input and output tensors, and returns an enriched – i.e. more precise – set of facts.

That gives us the following algorithm, with $\mathit{hints} \in \mathcal{P}(V_G \times \mathcal{F})$ a set of hints to give to the analyser, which is useful if we already know the shape of the input data for instance.

```
1  function AnalyseGraph(G, hints)
2      ord ← ComputePlan(G, v*)
3
4      ▷ We tag each edge of G with a tensor fact (which is the most general for now).
5      facts ← [⊤ for e ∈ E]
6
7      ▷ We use the given hints to add more information to the edges.
8      for (v, fact) ∈ hints do
9          for e ∈ e_o(v) do
10             facts[e] ← facts[e] ⊓ fact
11
12     ▷ We propagate the facts until reaching a fixed point.
13     loop
14         changed ← false
15
16         for v ∈ ord do
17             inputs ← [facts[e] for e ∈ e_i(v)]
```

```
18          outputs ← [facts[e] for e ∈ e_o(v)]
19
20          (inputs, outputs) ← enrich_op(v)(a(v), inputs, outputs)
21
22          for fact ∈ inputs do
23              p ← index(fact, inputs)
24              for e ∈ e_i(v, p) do
25                  facts[e] ← facts[e] ⊓ fact
26
27          for fact ∈ outputs do
28              p ← index(fact, outputs)
29              for e ∈ e_o(v, p) do
30                  facts[e] ← facts[e] ⊓ fact
31
32      ord ← reverse(ord)
33
34      if ¬changed then
35          return facts
```

This algorithm roughly corresponds to the chaotic iterations algorithm from traditional static analysis. Let $n \equiv |V_G|$, $e \equiv |E_G|$, h the height of the lattice $\mathcal{F}$, and c the cost *(which we assume is the same for every operation)* of calling $enrich_{op(v)}$ and $\sqcap$. In the worst case, in each iteration of the outer loop, only one edge of one node would get a tiny bit more information *(but each of the n nodes would make a call to* $enrich_{op(v)}$ *and* $\sqcap$*)*, giving us a complexity of $O(n * c * e * h)$.

This worst-case complexity, however, is far from realistic. One way to improve the practical performance of the algorithm is to use a well-chosen heuristic for the order in which nodes are picked in the inner loop. Here, as you can see on lines `17` and `33`, we start by propagating the information forward – w.r.t. the execution plan – then backward, then forward, etc.

The intuition behind this is that we'll start by propagating information about the `Placeholder` nodes *(that we get from the hints)* to the rest of the graph, and along the way the nodes will add more constraints about the inputs and outputs, which we'll use on our way back up, etc. Using this heuristic, I was able to get down the number of outer loop iterations to at most 4 on all the real-world graphs I tested.

▷ *The Rust implementation of the propagation algorithm can be found in Appendix C.*

2.3 At the operation level: declarative constraint solver.

There is still a problem, though: we must now write the $enrich_{op}$ function for every operation, and the task is much more cumbersome than it seems – especially considering that TFDeploy supports more than 20 different operations.

My first attempt at tackling that problem was to implement two simpler functions for every operation: $enrich^{\rightarrow}{}_{op}$ and $enrich^{\leftarrow}{}_{op}$. The first one, called *forward enrichment*, would only return a new set of output facts given a node's current input facts; and the second one, called *backward enrichment*, would do the opposite. Even though that doubled the number of functions to implement, they were simpler to write (`https://github.com/kali/tensorflow-deploy-rust/`

`tree/33b48389aecd20262cc16056af9b1e3f7c8a53f0` shows what the code looked like using this approach). But, after spending more than a week implementing all these functions, I was still not satisfied.

First of all, the separation into forward and backward enrichment reduced the accuracy of the static analyser. Take the `Pad` operation [6] for instance: it has two input ports called `input` and `paddings`, one output port called `output`, and it pads the beginning and end of each dimension of `input` with a number of zeros specified in `paddings`. So, if we have information about the value of `paddings` and the shape of `output`, we might be able to deduce the shape of `input`; but this is impossible to express when forward and backward enrichment are separated.

An even bigger problem was that $enrich^{\rightarrow}{}_{op}$ and $enrich^{\leftarrow}{}_{op}$ were still cumbersome to write – no matter how many helper functions and macros I tried to throw at the problem – as can be seen in Appendix D for the `Pad` operation; and were hard to reason about.

I eventually realized that this could only be solved if I stopped thinking procedurally, and instead expressed the relationships between the input and output facts of every operation using declarative rules. So I decided to design a declarative constraint solver which would allow me to write something like Fig. 5 instead of Appendix D.

Figure 5: Solver rules for `Pad` *(pseudocode)*.

1. `output.datatype = input.datatype`
2. `paddings.datatype = int32`
3. `paddings.rank = 2`
4. `paddings.shape[1] = 2`
5. `output.rank = input.rank = output.shape[0]`
6. $\forall$`i` $\in [\![0,$ `input.rank`$[\![$:
   `output.shape[i] = input.shape[i]`
   `+ paddings.value[i][0] + paddings.value[i][1]`

Here is an overview of the formalism that I used, as well as the actual solver algorithm.

Definition 12. A solver expression is either:

- A **variable expression**, which acts as a reference to any fact that can be compared using the solver. During the execution of the solver, it will resolve to the actual fact that the variable points to. *For instance,* `output.datatype` *is a reference to the datatype of the first output port of the node, and it will later resolve to one of* $\top_{TF}$, $Only_{TF}($`float32`$)$, *etc.*
- A **constant expression**, which holds any fact that can be compared using the solver. *For instance,* `int32` *actually corresponds to* `ConstantExpression`$(Only_{TF}($`int32`$))$.
- A **product expression**, which represents the product of another expression by a scalar.

Definition 13. A solver rule is either:

- An **equals rule**, which states that two expressions should resolve to the same value;
- An **all equals rule**, which generalizes of the *equals rule* to multiple expressions;
- An **equals zero rule**, which states that the sum of all its expressions resolve to 0;
- A **given rule**, which lets you wait until an expression resolves to a value to add new rules – which depend on this value – to the solver.

Figure 6: Solver rules for `Pad` *(internal representation)*.

1. EqualsRule(VariableExpr(`output.datatype`), VariableExpr(`input.datatype`))
2. EqualsRule(VariableExpr(`paddings.datatype`), ConstantExpr($Only_{TF}$(`int32`)))
3. EqualsRule(VariableExpr(`paddings.rank`), ConstantExpr($Only_{VF}$(2)))
4. EqualsRule(VariableExpr(`paddings.shape[1]`), ConstantExpr($Only_{DF}$(2)))
5. EqualsAllRule(
   VariableExpr(`output.rank`),
   VariableExpr(`input.rank`),
   VariableExpr(`output.shape[0]`))
6. GivenRule(
   VariableExpr(`input.rank`),
   AddRules(
   EqualsZeroRule(
   ProductExpr(-1, VariableExpr(`output.shape[0]`)),
   VariableExpr(`input.shape[0]`),
   VariableExpr(`paddings.value[0][0]`),
   VariableExpr(`paddings.value[0][1]`)),
   $\ldots$,
   EqualsZeroRule(
   ProductExpr(-1, VariableExpr(`output.shape[`*rank*`]`)),
   VariableExpr(`input.shape[`*rank*`]`),
   VariableExpr(`paddings.value[`*rank*`][0]`),
   VariableExpr(`paddings.value[`*rank*`][1]`)),
   ))

Fig. 6 shows how the rules for `Pad` from Fig. 5 translate into this formalism. Let us assume that, for every TensorFlow operation *op*, we have a set $\mathcal{R}_{op}$ of such rules. We can then deduce a generic implementation of $enrich_{op}$, which works as follows:

```
1  function enrich_op(attrs, input_facts, output_facts)
2      ▷ A context is essentially a map from the name of the port (e.g. input,
3      ▷ paddings or output) to the corresponding tensor fact.
4      context ← ContextFrom(input_facts, output_facts)
5      changed ← true
6      rules ← R_op
7
8      while changed do
9          changed ← false
10
11         for rule ∈ rules do
12             (used, added, context) ← ApplyRule(rule, context)
13             if used then
14                 rules ← rules \ {rule}
15             changed ← changed ∨ used ∨ (length(added) > 0)
16             rules ← rules ∪ added
17
18     return (context.inputs, context.outputs)
```

Where APPLYRULE is a function which takes a rule and a context and returns:

- whether the rule was used *(and can now be ignored)*;
- a list of new rules to be considered by the solver *(which is useful in the case of GivenRule)*;
- a new context in which the facts are modified according to the rule.

For the sake of readability, we will not give the precise implementation of APPLYRULE *(which can be found in Appendix E)* but instead a high-level overview of how it works:

- If APPLYRULE encounters an EqualsRule or an EqualsAllRule with expressions $e_1, \cdots, e_n$, it tries to resolve them – using *context* passed as an argument to fetch the current value of VariableExpressions – into the facts $f_1, \cdots, f_n$. Then, it computes the unification $f = \bigsqcap_{i=1}^{n} f_i$ of these facts, and finally returns $context'$, which is *context* modified in such a way that all the e_i now resolve into f.
  It raises an error if $f = \bot$ or if there is no way to build $context'$.

- If APPLYRULE encounters an EqualsZeroRule, it resolves all the $e_1, \cdots, e_n$ into $f_1, \cdots, f_n$. Let $\Delta = \{f_i,\ 1 \leq i \leq n,\ f_i = \top\}$. If $|\Delta| > 1$, the rule can't be used yet, and so it returns $(\texttt{false}, \emptyset, context)$. Otherwise, it computes $f = \sum_{f \in (f_i)_{1 \leq i \leq n} \setminus \Delta} f_i$, and returns $context'$ which is *context* modified in such a way that the only element of Δ now equals f.
  It raises an error if $f = \bot$ or if there is no way to build $context'$.

- If APPLYRULE encounters a GivenRule, it tries to resolve e into f. If $f = \top$, the rule can't be used yet, and so it returns $(\texttt{false}, \emptyset, context)$. Otherwise, it executes its inner closure – passing it the value of f – and return $(\texttt{true}, R, context)$ where R are the rules that were declared in the closure.

There were several interesting implementation details regarding the solver. First of all, it was quite challenging to coerce Rust's type system into letting me write something like Appendix. F, especially since Rust doesn't support runtime reflection and looses most of its type information before compilation. I also wanted the rules declaration to be type-safe, so that we wouldn't be able to write `solver.equals(&inputs.len, DataType::DT_INT32)` for instance. The solution – which involved traits, associated types, implicit casting and a bit of unsafe code – is available at `https://github.com/kali/tensorflow-deploy-rust/tree/master/src/analyser/interface` for the curious reader.

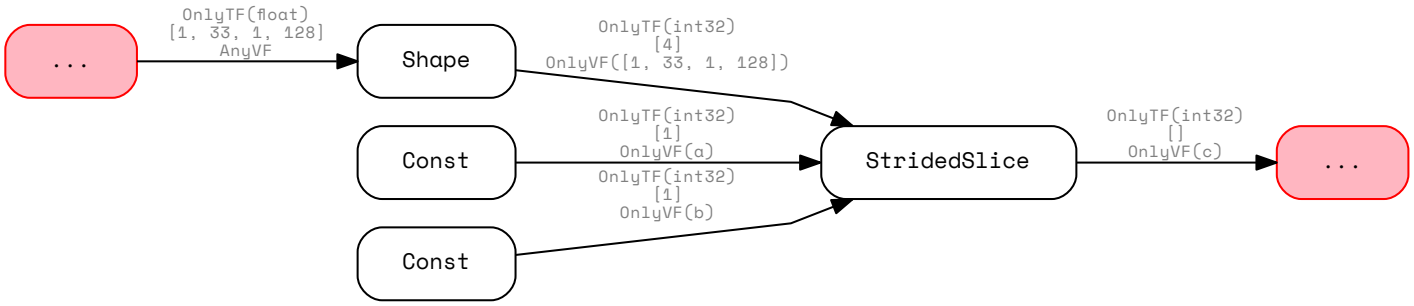
2.4 Ahead-of-time optimizations.

Now that we have the analyser working both at the graph and operation level, we can start reaping its benefits. As explained in the beginning of the section, one of these benefits is that we can use the analyser as a framework to build ahead-of-time optimizations.

One such optimization is **constant propagation** and **folding**, whose purpose is to both reduce the space of the graph file and make inference faster by pre-computing the value of constant parts of the graph before inference. Let us look at a real-world example on Fig. 7, which is annotated using information from the static analyser, to understand why this is necessary.

As you can see, even though the edge going to the `Shape` node is tagged with $\top_{\mathcal{VF}}$ – meaning that the static analyser doesn't have enough information to deduce its value – it managed to find an invariant on its shape. But since the `Shape` node only uses the shape of its input to compute its output, the analyser was able to deduce the value of that output ahead of time. Because we also know the value of `Const` nodes ahead of time, the analyser was finally able to deduce the output value of `StridedSlice` – which we've called c here.

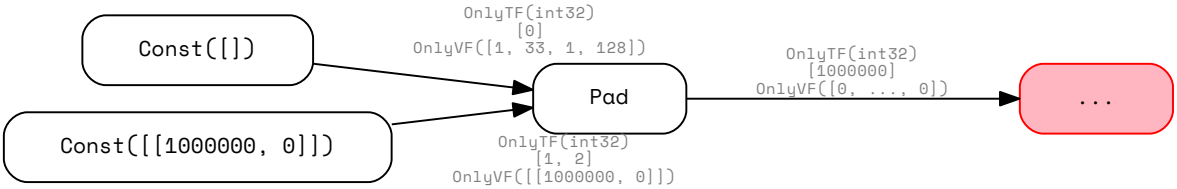
Figure 7: Excerpt of a dataflow graph where constant folding could be used.



In this scenario, we could save both space and execution time by replacing this entire part of the graph with a `Const` node of value c.

In other scenarios, however, we might not want to: take a look at Fig. 8 for instance. As an approximation, let α be the number of bytes necessary to store the name and operation of each node in the graph; β the number of bytes needed to store each edge; and assume that a tensor containing k values of type `int32` can be stored using $4k$ bytes only. Then, storing the graph from Fig. 8 as is would take up $3\alpha + 3\beta + 8$ bytes. But if we used constant folding on that graph, we would end up with a single `Const` node holding a tensor of 1,000,000 zeroes, which would take up $\alpha + \beta + 8000000$ bytes. This shows the importance of choosing a good heuristic when applying constant folding.

Figure 8: Excerpt of a dataflow graph where constant folding should not be used.



Definition 14. Let G be a dataflow graph. We define the **constant underlying graph** of G, written G_{const}, as the graph constructed using the following rules:

- If an edge of G has a constant value according to the analyser, it is in G_{const};
- If all the outgoing edges of a node in G are in G_{const}, that node is also in G_{const};
- If an edge in G_{const} has no target, it is called a **sink**.

We can now describe the constant folding algorithm in the general case:

1. Compute the constant underlying graph G_{const} of G;
2. Compute the undirected connected components C of G_{const};
3. Choose a pruning strategy and apply it to each component of C.

There are several pruning strategies to choose from, depending on an estimate of the size of the component after applying the strategy – which we compute in the same way as for Fig. 8:

- The simplest is to remove all nodes but the sinks of each component, and to replace the latter with `Const` nodes. This might however increase the size of the model dramatically in cases like the one described above.
- We could also search for the lowest common ancestor of all the sinks in each connected component, and prune every node and edge that isn't part of a path between that ancestor and a sink. If no such ancestor exists, we don't do anything. This helps in some cases, but still doesn't work in the one described above.
- Or we simply don't do anything.

▷ *The Rust implementation of the constant folding algorithm can be found in Appendix G.*

3 New streaming semantics for dataflow graphs

The ahead-of-time optimizations that I was able to write using the static analyser – constant folding in particular – helped me make CNN inference more space-efficient; but I was now looking for ways to make it more energy-efficient – and, ideally, faster.

Following a lead from my tutor, I realized that "traditionnal" inference on dataflow graphs – which is described in the introduction – is ill-suited to handling real-time data, such as an audio feed from a microphone, and leads to a waste of computing resources. To solve this problem, I decided to formalize and implement new semantics for inference on dataflow graphs which would be designed around real-time – or *streaming* – data.

In the following sections, I will clarify the mismatch between "traditional" inference and real-time data; formalize new streaming semantics for inference; use these semantics to propose a new inference algorithm that works well with real-time data; and give a few details about implementation and performance.

3.1 Problem statement.

One of the main uses of neural networks at Snips is hotword detection, which consists in listening to an audio feed – e.g. from a microphone – continuously to detect when a series of words is said by the user – for instance "Ok Google!" or, in that case, "Hey Snips!".

In practise, "listening to an audio feed continuously" is done by recording an audio sample from the feed every ΔT milliseconds; turning that sample into a row vector of $\mathbb{R}^K$ by applying a Fourier – or similar – transform and keeping the intensities at K different frequencies; and finally feeding the last N vectors into a NN which expects a $[N,\ K]$-shaped input tensor.

Let us start a clock at time $t = 0$, and call $s_0,\ s_1,\ s_2,\ \cdots$ the row vectors of $\mathbb{R}^K$ recorded at times $0,\ \Delta T,\ 2\Delta T$, etc. Let us then define, for $i >= 0$:

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

We won't run inference while $t < (N-1) * \Delta T$, but at $t = (N-1) * \Delta T$ we'll build input tensor I_0 and feed it into the neural network, and then at $t = N * \Delta T$ we'll build I_1 and feed it into the neural network again, etc.

The issue with this is that, each time we feed a new I_i into the neural network, $N-1$ samples "overlap", and so we potentially waste a lot of computing power re-computing values that were already computed ΔT milliseconds before.

This is especially true with convolutional neural networks – which is the case for most hotword detection networks. To illustrate this, let us consider the first layer of a CNN, which is usually the convolution of the input tensor – say I_0 for example – with a filter in $M_S(\mathbb{R})$. For the sake of simplicity, we'll take $K = N = 3$ and $S = 2$; which means we're computing the following convolution:

$$\begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} \\ s_{1,0} & s_{1,1} & s_{1,2} \\ s_{2,0} & s_{2,1} & s_{2,2} \end{pmatrix} \star \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} s_{0,0}\,a + s_{0,1}\,b + s_{1,0}\,c + s_{1,1}\,d & s_{0,1}\,a + s_{0,2}\,b + s_{1,1}\,c + s_{1,2}\,d \\ s_{1,0}\,a + s_{1,1}\,b + s_{2,0}\,c + s_{2,1}\,d & s_{1,1}\,a + s_{1,2}\,b + s_{2,1}\,c + s_{2,2}\,d \end{pmatrix}$$

Now, if we were to compute the same convolution with input I_1 instead:

$$\begin{pmatrix} s_{1,0} & s_{1,1} & s_{1,2} \\ s_{2,0} & s_{2,1} & s_{2,2} \\ s_{3,0} & s_{3,1} & s_{3,2} \end{pmatrix} \star \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} s_{1,0}\,a + s_{1,1}\,b + s_{2,0}\,c + s_{2,1}\,d & s_{1,1}\,a + s_{1,2}\,b + s_{2,1}\,c + s_{2,2}\,d \\ s_{2,0}\,a + s_{2,1}\,b + s_{3,0}\,c + s_{3,1}\,d & s_{2,1}\,a + s_{2,2}\,b + s_{3,1}\,c + s_{3,2}\,d \end{pmatrix}$$

Note that what the machine learning community calls "convolution" is actually discrete cross-correlation. See `https://www.tensorflow.org/api_guides/python/nn#Convolution` for a detailed explanation.

As we can see, we compute the quantities $s_{1,0}\,a + s_{1,1}\,b + s_{2,0}\,c + s_{2,1}\,d$ and $s_{1,1}\,a + s_{1,2}\,b + s_{2,1}\,c + s_{2,2}\,d$ twice, and this only gets worse as we increase K and N. The problem also gets worse as we increase the number of successive convolutional layers in the network, which seems to be the current trend in audio processing – following a 2016 paper [20] from Google DeepMind which introduced "dilated causal convolutions", a way to replicate the behavior of recurrent neural networks using only successive layers of dilated convolutions.

One way to mitigate the problem is to use batching: instead of feeding I_i to the neural network every ΔT milliseconds, we feed it every $N * \Delta T$ milliseconds to avoid "overlapping" samples. But this introduces latency – which we're trying to avoid – and tends to decrease the accuracy of the network's predictions *(on the hotword detection models used at Snips, at least)*.

Ideally, we would like a way to only feed the latest sample to the network every ΔT milliseconds, and let the network "remember" the previous samples to only re-compute values when necessary. Since this is not feasible using the "traditional" inference mechanism for neural networks described in the introduction, I had to come up with a new one.

3.2 Streaming semantics: formal overview.

This formalism essentially revolves around the idea of **streaming dimension**: we want a way to declare which – if any – of the dimensions of our input tensors corresponds to streamed – i.e. real-time – data. For instance, in our previous example, we would replace our `[`N`, `K`]`-shaped input tensor with a `[S, `K`]`-shaped one, where the `S` indicates a streaming dimension.

We must then adapt the static analyser to handle these special dimensions, and to propagate this information to the rest of the graph so that every node knows which of the dimensions of its input tensors correspond to streamed data.

That gives us the following updated definition:

Definition 15. A **shape fact** is an element of $\mathcal{SF} \equiv \{0,\ 1\} \times (\mathbb{N} \cup \{\_,\ \texttt{S}\})^*$. 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; or "S" – which matches a special, streaming dimension.

We also introduce a concept of **chunk**, which is a simply a tensor with size 1 along the streaming dimension. In our previous example, every s_i would have been a chunk.

We finally introduce a concept of **buffer**, which is a way for nodes to store intermediary results between evaluations of the network. In our previous example, the `Conv2D` operator would have stored the quantities $s_{1,0}\ a + s_{1,1}\ b + s_{2,0}\ c + s_{2,1}\ d$ and $s_{1,1}\ a + s_{1,2}\ b + s_{2,1}\ c + s_{2,2}\ d$ in its buffer to avoid re-computing them. We let each operation specify what it wants to store in a buffer, so for an operation op we just say the the corresponding buffers are elements of $\mathcal{B}_{op}$ without giving more precision about what this set contains.

This allows us to define the streaming variant of the evaluation function:

Definition 16. Every operation op is associated with an **streaming evaluation function**:

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

with n_i and n_o respectively the number of input and output ports that op expects.

This function takes:

- The set of attributes of the node;
- The buffer for this node from the previous execution;
- The port of the input for which we're currently receiving a chunk;
- The dimension which is being streamed on that port *(or $\bot$ if there is none)*;
- The chunk that we're currently receiving.

And it returns a new buffer, which will be stored for the next execution, and either a new chunk for each output port of the node or $\bot$ if there is nothing to return yet.

To illustrate this definition, we will use the example of the `Concat` operation [4]. This operation takes an argument `axis` along which to concatenate; and an arbitrary number of input tensors whose datatype and rank must match, and whose sizes must match in all dimensions but `axis`. It returns a single output tensor which is the concatenation of all the input tensors along `axis`. As all the input tensors must have the same rank and the same size along every dimension but `axis`, they must in particular all share the same streaming dimension. That leaves us with two possibilities:

- Either all the tensors are streamed along `axis`, in which case we don't store anything in the buffer, and return every chunk we receive as is;
- Or they are streamed along another dimension, so we buffer the chunks we receive and return $\bot$ until we have received a chunk for each port, at which point we return their concatenation.

▷ *The Rust implementation of $step_{\texttt{Conv2D}}$ is given in Appendix H. It is worth taking a look at, as it gives a good example of how to use buffers to avoid computing the same values twice.*

3.3 Streaming inference algorithm.

We can now glue the pieces together and write a new inference algorithm,which allows us to push chunks of data to the model multiple times. For the sake of readability, we will only give a high-level description of the algorithm.

1. Start by asking the value of all the inputs of the graph – i.e. `Placeholder` nodes – which don't have a streaming dimension.
2. Replace those inputs with `Const` nodes with the value that was given.
3. Use the static analyser on the graph, and apply the constant folding algorithm. This will yield a graph where all nodes are either streaming nodes (i.e. nodes in the path of a streaming dimension) or constant nodes.
4. Compute an execution plan (e.g. topological ordering) for streaming nodes of the graph.
5. Allocate a new buffer for each of these nodes.
6. When a new chunk is available for one of the inputs:
    - Run $step_{op}$ for the first node, replace its previous buffer with the one it just returned, and stop if it returns $\bot$.
    - If it returns a set of chunks instead, run $step_{op}$ for every successor of that node.
    - More generally, perform a breadth-first traversal of the graph **without marking already seen nodes**, and stopping only when a node returns $\bot$.

▷ *The Rust implementation of the algorithm is given in Appendix I.*

Conclusion

During the three months of this internship, I successfully formalized and implemented a static analyser and new inference semantics for dataflow graphs, which helped me optimize the energy- and space-efficiency of neural network inference on embedded devices.

I also started looking at ways to improve the speed of inference in the special case of convolutional neural networks, but I didn't have time to come up with any significant contributions on the subject beyond bibliographical work, so I chose to elide this part from the report.

Incidentally, I was able to learn the Rust programming language hands-on; and, by the end of the internship, I had written about 8,000 lines of Rust code which is now a part of TFDeploy and can be seen at `https://github.com/kali/tensorflow-deploy-rust/`.

I really enjoyed the experience, as working on a project which was already used in production meant that I was able to do everything from spotting issues in "real-world" situations; to coming up with abstractions to solve these issues; to implementing them directly in the production codebase and seeing their impact on the voice assistants shipped by Snips.

But that also came at a cost. Because I had to work my way around an existing codebase, I spent a lot of time trying to figure out how things worked and where they were before being able to implement any improvements.

Working in production also made it more difficult to iterate over new ideas quickly, as every idea had to be implemented throughout the entire codebase before being able to realize whether or not it was good. As an example, choosing to implement $\mathit{restrict}^{\rightarrow}$ and $\mathit{restrict}^{\leftarrow}$ for every operation – as I described in section 2.3 – turned out to be a really bad design decision, but since TFDeploy supported more than 20 complex operations, it took me more than a week to realize my mistake and start over.

Finally, working with a systems programming language like Rust made implementing my ideas a bit more tedious, as I constantly had to think about low-level details – even when prototyping.

Acknowledgments

As a closing note, I would like to thank everyone at Snips for their warm welcome, and in particular Mathieu – for all his help and his Rust magic tricks; the rest of the Device Engineering team; and Isabelle – for bearing with my last minute plans in May.

References

[1] Introduction to TensorFlow Lite. https://www.tensorflow.org/mobile/tflite/.

[2] Introduction to TensorFlow Mobile. https://www.tensorflow.org/mobile/mobile_intro.

[3] TensorBoard. *(GitHub repository)*. https://github.com/tensorflow/tensorboard.

[4] Tensorflow documentation: Concat. https://www.tensorflow.org/api_docs/cc/class/tensorflow/ops/concat.

[5] TensorFlow documentation: Graphs and Sessions. https://www.tensorflow.org/guide/graphs.

[6] Tensorflow documentation: Pad. https://www.tensorflow.org/versions/r1.6/api_docs/cc/class/tensorflow/ops/pad.

[7] TensorFlow. *(GitHub repository)*. https://github.com/tensorflow/tensorflow.

[8] TensorFlow: Using AOT compilation. https://www.tensorflow.org/performance/xla/tfcompile.

[9] TensorFlow: Using JIT compilation. https://www.tensorflow.org/performance/xla/jit.

[10] The Rust Programming Language. https://www.rust-lang.org/.

[11] G. J. Badros, A. Borning, and P. J. Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Trans. Comput.-Hum. Interact.*, 8(4):267–306, Dec. 2001.

[12] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164.

[13] E. R. Gansner, E. Koutsofios, S. C. North, and K. . Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, March 1993.

[14] R. Liautaud. TFDeploy command-line interface. *(GitHub repository)*. https://github.com/kali/tensorflow-deploy-rust/tree/master/cli.

[15] R. Liautaud. TFDeploy Visualizer. *(GitHub repository)*. https://github.com/kali/tensorflow-deploy-rust/tree/master/visualizer.

[16] Michael I. Schwartzbach. Lecture Notes on Static Analysis. http://lara.epfl.ch/w/_media/sav08:schwartzbach.pdf.

[17] Pierre Roux and Pierre-Loïc Garoche. Dessine moi un domaine abstrait fini. https://www.onera.fr/sites/default/files/447/jfla2011.pdf, 2011.

[18] M. Poumeyrol and R. Liautaud. TFDeploy. *(GitHub repository)*. https://github.com/kali/tensorflow-deploy-rust.

[19] H. Rutjes. Kiwi.js. *(GitHub repository)*. https://github.com/IjzerenHein/kiwi.js/.

[20] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. Wavenet: A generative model for raw audio. In *Arxiv*, 2016.

Appendices

Appendix A: Output of Graphviz's layout on a hotword detection model used at Snips.

Appendix B: Type definitions and unification functions for the analyser facts.

```rust
use tfpb::types::DataType;
use Tensor;
use Result;

/// Partial information about any value.
pub trait Fact: Clone + PartialEq {
    type Concrete;

    /// Tries to transform the fact into a concrete value.
    fn concretize(&self) -> Option<Self::Concrete>;

    /// Returns whether the value is fully determined.
    fn is_concrete(&self) -> bool {
        self.concretize().is_some()
    }

    /// Tries to unify the fact with another fact of the same type.
    fn unify(&self, other: &Self) -> Result<Self>;
}

/// Partial information about a tensor.
///
/// The task of the analyser is to tag every edge in the graph with information
/// about the tensors that flow through it - specifically their datatype, their
/// shape and possibly their value. During the analysis, however, we might only
/// know some of that information (say, for instance, that an edge only carries
/// tensors of rank 4, but without knowing their precise dimension).
///
/// This is where tensor facts come in: they hold partial information about the
/// datatype, shape and value of tensors that might flow through an edge of the
/// graph. The analyser will first tag each edge with a fact, starting with the
/// most general one and specializing it at each iteration. Eventually, it will
/// reach a fixed point that - hopefully - holds enough information.
#[cfg_attr(feature = "serialize", derive(Serialize))]
#[derive(Clone, PartialEq)]
pub struct TensorFact {
    pub datatype: TypeFact,
    pub shape: ShapeFact,
    pub value: ValueFact,
}

impl Fact for TensorFact {
    type Concrete = Tensor;

    /// Tries to transform the fact into a concrete value.
    fn concretize(&self) -> Option<Self::Concrete> {
        self.value.concretize()
    }

    /// Tries to unify the fact with another fact of the same type.
    fn unify(&self, other: &Self) -> Result<Self> {
        let tensor = TensorFact {
            datatype: self.datatype.unify(&other.datatype)?,
            shape:    self.shape.unify(&other.shape)?,
            value:    self.value.unify(&other.value)?,
        };

        trace!("Unifying {:?} with {:?} into {:?}.", self, other, tensor);
```

```
        Ok(tensor)
    }
}

/// Partial information about a value of type T.
#[cfg_attr(feature = "serialize", derive(Serialize))]
#[derive(Clone, PartialEq)]
pub enum GenericFact<Clone + PartialEq> {
    Any,
    Only(T)
}

impl<T: Clone + PartialEq> Fact for GenericFact<T> {
    type Concrete = T;

    /// Tries to transform the fact into a concrete value.
    fn concretize(&self) -> Option<T> {
        match self {
            GenericFact::Any => None,
            GenericFact::Only(m) => Some(m.clone()),
        }
    }

    /// Tries to unify the fact with another fact of the same type.
    fn unify(&self, other: &Self) -> Result<Self> {
        let fact = match (self, other) {
            (_, GenericFact::Any) => self.clone(),
            (GenericFact::Any, _) => other.clone(),
            _ if self == other    => self.clone(),
            _ => bail!("Impossible to unify {:?} with {:?}.", self, other),
        };

        Ok(fact)
    }
}

/// Partial information about a type.
pub type TypeFact = GenericFact<DataType>;

/// Partial information about a shape.
///
/// A basic example of a shape fact is `shapefact![1, 2]`, which corresponds to
/// the shape `[1, 2]` in Tensorflow. We can use `_` in facts to denote unknown
/// dimensions (e.g. `shapefact![1, 2, _]` corresponds to any shape `[1, 2, k]`
/// with `k` a non-negative integer). We can also use `..` at the end of a fact
/// to only specify its first dimensions, so `shapefact![1, 2; ..]` matches any
/// shape that starts with `[1, 2]` (e.g. `[1, 2, i]` or `[1, 2, i, j]`), while
/// `shapefact![..]` matches any shape.
#[cfg_attr(feature = "serialize", derive(Serialize))]
#[derive(Clone, PartialEq)]
pub struct ShapeFact {
    pub open: bool,
    pub dims: Vec<DimFact>,
}

impl ShapeFact {
    /// Constructs an open shape fact.
    pub fn open(dims: Vec<DimFact>) -> ShapeFact {
        ShapeFact { open: true, dims }
    }

    /// Constructs a closed shape fact.
```

```
    pub fn closed(dims: Vec<DimFact>) -> ShapeFact {
        ShapeFact { open: false, dims }
    }
}

impl Fact for ShapeFact {
    type Concrete = Vec<usize>;

    /// Tries to transform the fact into a `Vec<usize>`, or returns `None`.
    fn concretize(self: &ShapeFact) -> Option<Vec<usize>> {
        if self.open {
            debug!("Impossible to concretize an open shape.");
            return None;
        }

        let dims: Vec<_> = self.dims.iter().filter_map(|d| d.concretize()).collect();

        if dims.len() < self.dims.len() {
            debug!("Impossible to concretize a shape with unknown dimensions.");
            None
        } else {
            Some(dims)
        }
    }

    /// Tries to unify the fact with another fact of the same type.
    fn unify(&self, other: &Self) -> Result<Self> {
        let (x, y) = (self, other);

        use itertools::EitherOrBoth::{Both, Left, Right};
        use itertools::Itertools;

        let xi = x.dims.iter();
        let yi = y.dims.iter();

        let dimensions: Vec<_> = xi.zip_longest(yi)
            .map(|r| match r {
                Both(a, b) => a.unify(b),
                Left(d) if y.open => Ok(*d),
                Right(d) if x.open => Ok(*d),

                Left(_) | Right(_) => bail!(
                    "Impossible to unify closed shapes of different rank (found {:?} and {:?}).",
                    x,
                    y
                ),
            })
            .collect::<Result<_>>()?;

        if x.open && y.open {
            Ok(ShapeFact::open(dimensions))
        } else {
            Ok(ShapeFact::closed(dimensions))
        }
    }
}

/// Partial information about a dimension.
#[cfg_attr(feature = "serialize", derive(Serialize))]
#[derive(Clone, Copy, PartialEq)]
pub enum DimFact {
    Any,
    Streamed,
```

```
    Only(usize),
}

impl DimFact {
    /// Returns whether the dimension is streamed.
    pub fn is_streamed(&self) -> bool {
        self == &DimFact::Streamed
    }
}

impl Fact for DimFact {
    type Concrete = usize;

    /// Tries to transform the dimension fact into an `usize`, or returns `None`.
    fn concretize(&self) -> Option<usize> {
        match self {
            DimFact::Any => None,
            DimFact::Streamed => None,
            DimFact::Only(i) => Some(*i),
        }
    }

    /// Returns whether the dimension is concrete.
    fn is_concrete(&self) -> bool {
        match self {
            DimFact::Any => false,
            DimFact::Streamed => true,
            DimFact::Only(_) => true,
        }
    }

    /// Tries to unify the fact with another `DimFact`.
    fn unify(&self, other: &Self) -> Result<Self> {
        let fact = match (self, other) {
            (_, DimFact::Any)  => self.clone(),
            (DimFact::Any, _)  => other.clone(),
            _ if self == other => self.clone(),
            _ => bail!("Impossible to unify {:?} with {:?}.", self, other),
        };

        Ok(fact)
    }
}

/// Partial information about a value.
pub type ValueFact = GenericFact<Tensor>;
```

▷ *Edited for clarity, see the full version at* `https://github.com/kali/tensorflow-deploy-rust/blob/master/src/analyser/types.rs`.

Appendix C: Rust implementation of the propagation algorithm.

```rust
/// An edge of the analysed graph, annotated by a fact.
#[cfg_attr(feature = "serialize", derive(Serialize))]
#[derive(Debug, Clone, PartialEq)]
pub struct Edge {
    pub id: usize,
    pub from_node: Option<usize>,
    pub from_out: usize,
    pub to_node: Option<usize>,
    pub fact: TensorFact,
}

/// A graph analyser, along with its current state.
pub struct Analyser {
    // The original output.
    pub output: usize,

    // The graph being analysed.
    pub nodes: Vec<Node>,
    pub edges: Vec<Edge>,
    pub prev_edges: Vec<Vec<usize>>,
    pub next_edges: Vec<Vec<usize>>,

    // The execution plan and unused nodes.
    plan: Vec<usize>,

    // The current state of the algorithm.
    pub current_pass: usize,
    pub current_step: usize,
    pub current_direction: bool,
}

impl Analyser {
    /// Constructs an analyser for the given graph.
    ///
    /// The output argument is used to infer an execution plan for the graph.
    /// Changing it won't alter the correctness of the analysis, but it might
    /// take much longer to complete.
    pub fn new(model: Model, output: usize) -> Result<Analyser> {
        let nodes = model.nodes;
        let mut edges = vec![];
        let mut prev_edges = vec![Vec::new(); nodes.len() + 1];
        let mut next_edges = vec![Vec::new(); nodes.len() + 1];

        for node in &nodes {
            for input in &node.inputs {
                let id = edges.len();

                edges.push(Edge {
                    id,
                    from_node: Some(input.0),
                    from_out: input.1.unwrap_or(0),
                    to_node: Some(node.id),
                    fact: TensorFact::new(),
                });

                prev_edges[node.id].push(id);
                next_edges[input.0].push(id);
            }
```

```
        }

        // Add a special output edge.
        let special_edge_id = edges.len();
        edges.push(Edge {
            id: special_edge_id,
            from_node: Some(output),
            from_out: 0,
            to_node: None,
            fact: TensorFact::new(),
        });

        next_edges[output].push(special_edge_id);

        // Compute an execution plan for the graph.
        let plan = Plan::for_nodes(&nodes, &[output])?.order;
        let current_pass = 0;
        let current_step = 0;
        let current_direction = true;

        info!("Using execution plan {:?}.", plan);

        Ok(Analyser {
            output,
            nodes,
            edges,
            prev_edges,
            next_edges,
            plan,
            current_pass,
            current_step,
            current_direction,
        })
    }

    /// Adds an user-provided tensor fact to the analyser.
    pub fn hint(&mut self, node: usize, fact: &TensorFact) -> Result<()> {
        if node >= self.next_edges.len() {
            bail!("There is no node with index {:?}.", node);
        }

        for &j in &self.next_edges[node] {
            self.edges[j].fact = unify(fact, &self.edges[j].fact)?;
        }

        Ok(())
    }

    /// Returns a model from the analyser.
    pub fn into_model(self) -> Model {
        let mut nodes_by_name = HashMap::with_capacity(self.nodes.len());
        self.nodes.iter().for_each(|n| {
            nodes_by_name.insert(n.name.clone(), n.id);
        });

        Model {
            nodes: self.nodes,
            nodes_by_name,
        }
    }

    /// Computes a new execution plan for the graph.
    pub fn reset_plan(&mut self) -> Result<()> {
```

```
    self.plan = Plan::for_nodes(&self.nodes, &[self.output])?.order;
    Ok(())
}

/// Detaches the constant nodes and edges from the given graph.
pub fn propagate_constants(&mut self) -> Result<()> {
    constants::propagate_constants(self)
}

/// Removes the nodes and edges which are not part of the execution plan.
/// Returns the mapping between the old and new node indexes.
pub fn prune_unused(&mut self) -> Vec<Option<usize>> {
    let mut node_used = vec![false; self.nodes.len()];
    let mut edge_used = vec![false; self.edges.len()];
    for &i in &self.plan {
        node_used[i] = true;
    }

    // Remove the nodes while keeping track of the new indices.
    let mut deleted = 0;
    let mut node_mapping = vec![None; self.nodes.len()];

    for i in 0..self.nodes.len() {
        if !node_used[i] {
            self.nodes.remove(i - deleted);

            self.prev_edges.remove(i - deleted);
            self.next_edges.remove(i - deleted);
            deleted += 1;
        } else {
            node_mapping[i] = Some(i - deleted);

            self.prev_edges[i - deleted].iter().for_each(|&j| edge_used[j] = true);
            self.next_edges[i - deleted].iter().for_each(|&j| edge_used[j] = true);
        }
    }

    info!("Deleted {:?} unused nodes.", deleted);

    // Update the nodes and edges to use the new indices.
    for node in &mut self.nodes {
        node.id = node_mapping[node.id].unwrap();
        node.inputs.iter_mut().for_each(|i| i.0 = node_mapping[i.0].unwrap());
    }

    for edge in &mut self.edges {
        if let Some(i) = edge.from_node {
            edge.from_node = node_mapping[i];
        }

        if let Some(i) = edge.to_node {
            edge.to_node = node_mapping[i];
        }
    }

    // Remove the edges while keeping track of the new indices.
    let mut deleted = 0;
    let mut edge_mapping = vec![None; self.edges.len()];

    for i in 0..self.edges.len() {
        if !edge_used[i] {
            self.edges.remove(i - deleted);
            deleted += 1;
```

```
        } else {
            edge_mapping[i] = Some(i - deleted);
        }
    }

    info!("Deleted {:?} unused edges.", deleted);

    // Update the adjacency lists to use the new indices.
    for i in 0..self.nodes.len() {
        self.prev_edges[i].iter_mut().for_each(|j| *j = edge_mapping[*j].unwrap());
        self.next_edges[i].iter_mut().for_each(|j| *j = edge_mapping[*j].unwrap());
    }

    node_mapping
}

/// Runs the entire analysis at once.
pub fn run(&mut self) -> Result<()> {
    self.current_pass = 0;

    loop {
        if !self.run_two_passes()? {
            return Ok(());
        }
    }
}

/// Runs two passes of the analysis.
pub fn run_two_passes(&mut self) -> Result<bool> {
    let mut changed = false;

    info!(
        "Starting pass [pass={:?}, direction={:?}].",
        self.current_pass, self.current_direction,
    );

    // We first run a forward pass.
    self.current_step = 0;
    for _ in 0..self.plan.len() {
        if self.run_step()? {
            changed = true;
        }
    }

    info!(
        "Starting pass [pass={:?}, direction={:?}].",
        self.current_pass, self.current_direction,
    );

    // We then run a backward pass.
    self.current_step = 0;
    for _ in 0..self.plan.len() {
        if self.run_step()? {
            changed = true;
        }
    }

    Ok(changed)
}

/// Runs a single step of the analysis.
pub fn run_step(&mut self) -> Result<bool> {
    let changed = self.try_step()?;
```

```
        // Switch to the next step.
        self.current_step += 1;
        if self.current_step == self.plan.len() {
            self.current_pass += 1;
            self.current_direction = !self.current_direction;
            self.current_step = 0;
        }

        Ok(changed)
    }

    /// Tries to run a single step of the analysis, and returns whether
    /// there was any additional information gained during the step.
    fn try_step(&mut self) -> Result<bool> {
        let node = if self.current_direction {
            &self.nodes[self.plan[self.current_step]]
        } else {
            &self.nodes[self.plan[self.plan.len() - 1 - self.current_step]]
        };

        debug!(
            "Starting step for {} ({}) [pass={:?}, direction={:?}, step={:?}].",
            node.name, node.op_name, self.current_pass, self.current_direction, self.current_step,
        );

        let inputs: Vec<_> = self.prev_edges[node.id]
            .iter()
            .map(|&i| self.edges[i].fact.clone())
            .collect();

        let mut outputs = vec![TensorFact::new()];
        for &i in &self.next_edges[node.id] {
            outputs[0] = unify(&self.edges[i].fact, &outputs[0])?;
        }

        let enriched = node.op
            .enrich(inputs, outputs)
            .map_err(|e| format!("While enriching for {}: {}", node.name, e))?;

        let mut changed = false;

        for (i, &j) in self.prev_edges[node.id].iter().enumerate() {
            let fact = &enriched.0[i];
            let unified = unify(fact, &self.edges[j].fact)
                .map_err(|e| format!(
                    "While unifying inputs of node {:?}: {}",
                    node.name, e
                ))?;

            changed |= unified != self.edges[j].fact;
            self.edges[j].fact = unified;
        }

        for (_, &j) in self.next_edges[node.id].iter().enumerate() {
            if enriched.1.len() != 1 {
                panic!("Analyser only supports nodes with a single output port.");
            }

            let fact = &enriched.1[0];
            let unified = unify(fact, &self.edges[j].fact)
                .map_err(|e| format!(
                    "While unifying outputs of node {:?}: {}",
```

```
                    node.name, e
                ))?;

            changed |= unified != self.edges[j].fact;
            self.edges[j].fact = unified;
        }

        Ok(changed)
    }
}
```

Appendix D: Previous implementation of $enrich^{\rightarrow}{}_{\texttt{Pad}}$.

```rust
/// Deduces properties about the output tensors from the input tensors.
fn enrich_forward(&self, mut inputs: Vec<&TensorFact>) -> Result<Option<Vec<TensorFact>>> {
    use analyser::*;

    if inputs.len() != 2 {
        bail!("Pad operation needs exactly two inputs.");
    }

    // If we know everything about all the input ports.
    if let Some(output) = infer_forward_concrete(self, &inputs)? {
        return Ok(Some(output));
    }

    let (input_fact, paddings_fact) = args_2!(inputs);

    // If we know the type of the input, we can deduce the type of the output.
    let mut output_fact = TensorFact {
        datatype: input_fact.datatype,
        ..TensorFact::default()
    };

    // If we know the shape of the input and the value of paddings, we can
    // deduce the shape of the output.
    if let (Some(mut shape), Some(pad)) =
           (input_fact.shape.concretize(), paddings_fact.value.concretize()) {
        let pad = i32::tensor_to_view(pad)?;
        shape.iter_mut()
            .zip(pad.outer_iter())
            .for_each(|(s, p)| *s += p[0] as usize + p[1] as usize);

        output_fact.shape = shape.into();
    }

    Ok(Some(vec!(output_fact)))
}
```

▷ *Edited for clarity, see the full version at https://github.com/kali/tensorflow-deploy-rust/blob/5e73c94f60aef4ccf1b3c63af180a13cc4567a3b/src/ops/array/pad.rs.*

Appendix E: Rust implementation of the declarative constraint solver.

```rust
use analyser::interface::expressions::Expression;
use analyser::interface::expressions::IntoExpression;
use analyser::interface::expressions::Output;
use analyser::interface::path::{get_path, set_path, Path};
use analyser::types::{Fact, IntFact, SpecialKind, TensorFact};
use Result;

use std::fmt;

/// A structure that holds the current sets of TensorFacts.
///
/// This is used during inference (see `Solver::infer`) to let rules compute
/// the value of expressions which involve tensor properties.
#[derive(Debug, new)]
pub struct Context {
    pub inputs: Vec<TensorFact>,
    pub outputs: Vec<TensorFact>,
}

impl Context {
    /// Returns the current value of the variable at the given path.
    pub fn get<T: Output>(&self, path: &Path) -> Result<T> {
        let value = get_path(self, &path[..])?;

        Ok(T::from_wrapped(value)?)
    }

    /// Tries to set the value of the variable at the given path.
    pub fn set<T: Output>(&mut self, path: &Path, value: T) -> Result<()> {
        set_path(self, &path[..], T::into_wrapped(value))?;

        Ok(())
    }
}

/// A rule that can be applied by the solver.
pub trait Rule<'rules>: fmt::Debug {
    /// Tries to apply the rule to a given context.
    ///""
    /// The method must return Ok(true) if the rule was applied successfully
    /// (meaning that the Context was mutated), or Ok(false) if the rule was
    /// not applied but didn't generate any errors.
    fn apply(&self, context: &mut Context) -> Result<(bool, Vec<Box<Rule<'rules> + 'rules>>)>;

    /// Returns the paths that the rule depends on.
    fn get_paths(&self) -> Vec<&Path>;
}

/// The `equals` rule.
/// It states that the given expressions must all be equal.
///
/// It can be added to the solver via the following two methods:
/// ```text
/// solver.equals(a, b);
/// solver.equals_all(vec![a, b, ...]);
/// ```
struct EqualsRule<T: Output + Fact> {
    items: Vec<Box<Expression<Output = T>>>,
```

```rust
}

impl<T: Output + Fact> EqualsRule<T> {
    /// Creates a new EqualsRule instance.
    pub fn new(items: Vec<Box<Expression<Output = T>>>) -> EqualsRule<T> {
        EqualsRule { items }
    }
}

impl<'rules, T: Output + Fact> Rule<'rules> for EqualsRule<T> {
    /// Tries to apply the rule to a given context.
    fn apply(&self, context: &mut Context) -> Result<(bool, Vec<Box<Rule<'rules> + 'rules>>)> {
        if self.items.len() < 1 {
            return Ok((false, vec![]));
        }

        // Unify the value of all the expressions into one.
        let mut value: T = Default::default();
        for item in &self.items {
            value = value.unify(&item.get(context)?)?;
        }

        if value != Default::default() {
            // Set all the values to this unified one.
            for item in &self.items {
                item.set(context, value.clone())?;
            }

            Ok((true, vec![]))
        } else {
            Ok((false, vec![]))
        }
    }

    /// Returns the paths that the rule depends on.
    fn get_paths(&self) -> Vec<&Path> {
        self.items.iter().flat_map(|e| e.get_paths()).collect()
    }
}

impl<'rules, T: Output + Fact> fmt::Debug for EqualsRule<T> {
    fn fmt(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
        write!(formatter, "{:?}", self.items[0])?;
        for item in &self.items[1..] {
            write!(formatter, " == {:?}", item)?;
        }
        Ok(())
    }
}

/// The `equals_zero` rule.
/// It states that the sum of the given expressions must equal zero.
///
/// It can be added to the solver via the following method:
/// ```text
/// solver.equals_zero(vec![a, b, ...]);
/// ```
struct EqualsZeroRule {
    items: Vec<Box<Expression<Output = IntFact>>>,
}

impl EqualsZeroRule {
    /// Creates a new EqualsZeroRule instance.
```

```
    pub fn new(items: Vec<Box<Expression<Output = IntFact>>>) -> EqualsZeroRule {
        EqualsZeroRule { items }
    }
}

impl<'rules> Rule<'rules> for EqualsZeroRule {
    /// Tries to apply the rule to a given context.
    fn apply(&self, context: &mut Context) -> Result<(bool, Vec<Box<Rule<'rules> + 'rules>>)> {
        // Find all the expressions which have a value in the context.
        let mut values = vec![];
        let mut sum: IntFact = 0usize.into();

        let mut misses = vec![];

        for item in &self.items {
            let value = item.get(context)?;

            if value.is_concrete() {
                values.push(value.clone());
                sum = sum + value;
            } else {
                misses.push(item);
            }
        }

        if misses.len() > 1 {
            Ok((false, vec![]))
        } else if misses.len() == 1 {
            match sum {
                IntFact::Only(sum) => {
                    misses[0].set(context, IntFact::Only(-sum))?;
                    Ok((true, vec![]))
                }
                IntFact::Special(SpecialKind::Streamed) => {
                    misses[0].set(context, IntFact::Special(SpecialKind::Streamed))?;
                    Ok((true, vec![]))
                }
                IntFact::Any => Ok((false, vec![])),
            }
        } else if sum == 0usize.into() || sum == IntFact::Special(SpecialKind::Streamed) {
            Ok((true, vec![]))
        } else {
            bail!(
                "The sum of these values doesn't equal zero: {:?}. ({:?})",
                values,
                sum
            );
        }
    }

    /// Returns the paths that the rule depends on.
    fn get_paths(&self) -> Vec<&Path> {
        self.items.iter().flat_map(|e| e.get_paths()).collect()
    }
}

impl fmt::Debug for EqualsZeroRule {
    fn fmt(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
        write!(formatter, "{:?}", self.items[0])?;
        for item in &self.items[1..] {
            write!(formatter, " + {:?}", item)?;
        }
        write!(formatter, " == 0")
```

```rust
    }
}

/// The `given` rule.
/// It allows you to add more rules to the solver once the value of a given
/// expression is known, using a closure that takes the value as parameter.
///
/// It can be added to the solver via the following method:
/// ```text
/// solver.given(input.rank, |solver, ir|
///     // Add more rules to `solver` here.
/// );
/// ```
pub struct GivenRule<'rules, T: Output + Fact, E: Expression<Output = T>, C: Output> {
    pub item: E,
    pub closure: Box<Fn(&mut Solver<'rules>, C) + 'rules>,
}

impl<'rules, T: Output + Fact, E: Expression<Output = T>, C: Output> GivenRule<'rules, T, E, C> {
    /// Creates a new GivenRule instance.
    pub fn new<F>(item: E, closure: F) -> GivenRule<'rules, T, E, C>
    where
        F: Fn(&mut Solver<'rules>, C) + 'rules,
    {
        let closure = Box::new(closure);

        GivenRule { item, closure }
    }
}

impl<'rules, T: Output + Fact, E: Expression<Output = T>, C: Output> Rule<'rules>
    for GivenRule<'rules, T, E, C>
{
    /// Tries to apply the rule to a given context.
    fn apply(&self, context: &mut Context) -> Result<(bool, Vec<Box<Rule<'rules> + 'rules>>)> {
        // When calling `self.item.get(context)?`, we get a value of type T.
        // However, the developer might have wanted to explicitely convert
        // this value into a value of type C (using type annotations on the
        // closure parameters), so we need to perform that conversion.
        //
        // Thankfully, because both T and C implement Output, the conversion
        // is as simple as wrapping and un-wrapping the value.
        let wrapped = self.item.get(context)?.wrap();

        if let Ok(value) = C::from_wrapped(wrapped) {
            trace!("    Given rule: {:?} is {:?}", self.item, value);
            // We create a new solver instance, which will be populated with
            // new rules by the code inside the closure.
            let mut solver = Solver::default();

            (self.closure)(&mut solver, value);

            Ok((true, solver.take_rules()))
        } else {
            trace!(
                "In {:?}, failed to convert {:?} to expected type",
                self,
                self.item.get(context)?.wrap()
            );
            Ok((false, vec![]))
        }
    }
```

```rust
    /// Returns the paths that the rule depends on.
    fn get_paths(&self) -> Vec<&Path> {
        self.item.get_paths()
    }
}

impl<'s, T: Output + Fact, E: Expression<Output = T>, C: Output> fmt::Debug
    for GivenRule<'s, T, E, C>
{
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "GivenRule {{ {:?} }}", self.item)
    }
}

/// A declarative constraint solver for tensors.
#[derive(Default)]
pub struct Solver<'rules> {
    // The rules used by the solver.
    pub rules: Vec<Box<Rule<'rules> + 'rules>>,
}

impl<'rules> Solver<'rules> {
    /// Consumes the solver and returns the rules that it uses.
    pub fn take_rules(self) -> Vec<Box<Rule<'rules> + 'rules>> {
        self.rules
    }

    /// Runs the solver on a set of TensorFacts.
    ///
    /// This method returns:
    /// - Err(_) if a constraint couldn't be satisfied.
    /// - Ok(None) if no more information about tensors could be deduced.
    /// - Ok(Some(facts)) otherwise, with `facts` the new TensorFacts.
    pub fn infer(
        self,
        facts: (Vec<TensorFact>, Vec<TensorFact>),
    ) -> Result<(Vec<TensorFact>, Vec<TensorFact>)> {
        let mut context = Context::new(facts.0, facts.1);

        // Apply the rules until reaching a fixed point.
        let mut changed = true;
        let mut added_rules = vec![];
        let mut rules: Vec<_> = self.rules.into_iter().map(|r| (false, r)).collect();

        while changed {
            changed = false;

            for (used, rule) in &mut rules {
                // Don't try to apply rules which have already been used.
                if *used {
                    continue;
                }

                trace!("  Applying rule {:?}", rule);
                let (step_used, mut step_added) = rule.apply(&mut context)?;
                *used |= step_used;

                // There is a change if the rule was used, or if it added new rules.
                changed |= step_used;
                changed |= step_added.len() > 0;

                added_rules.append(&mut step_added);
            }
```

```
        for rule in added_rules.drain(..) {
            rules.push((false, rule));
        }
    }

    Ok((context.inputs, context.outputs))
}

/// Ensures that two expressions are equal.
///
/// For instance, one could write:
/// ```text
/// solver.equals(outputs[0].rank, inputs[1].shape[0]);
/// solver.equals(outputs[1].rank, 3);
/// ```
pub fn equals<T, EA, EB, A, B>(&mut self, left: A, right: B) -> &mut Solver<'rules>
where
    T: Output + Fact + 'static,
    EA: Expression<Output = T> + 'static,
    EB: Expression<Output = T> + 'static,
    A: IntoExpression<EA>,
    B: IntoExpression<EB>,
{
    let items: Vec<Box<Expression<Output = T>>> = wrap![left, right];

    let rule = EqualsRule::new(items);
    self.rules.push(Box::new(rule));
    self
}

/// Ensures that an several expressions are equal.
///
/// For instance, one could write:
/// ```text
/// solver.equals_all(vec![
///     outputs[0].rank.into(),
///     inputs[1].shape[0].into(),
///     3.into(),
/// ]);
/// ```
pub fn equals_all<T>(&mut self, items: Vec<Box<Expression<Output = T>>>) -> &mut Solver<'rules>
where
    T: Output + Fact + 'static,
{
    let rule = EqualsRule::new(items);
    self.rules.push(Box::new(rule));
    self
}

/// Ensures that the sum of several expressions equals zero.
///
/// For instance, one could write:
/// ```text
/// solver.equals_zero(vec![
///     outputs[0].rank.into(),
///     outputs[1].rank.into(),
///     (-1, inputs[1].shape[0]).into(),
/// ]);
/// ```
pub fn equals_zero(
    &mut self,
    items: Vec<Box<Expression<Output = IntFact>>>,
```

```
    ) -> &mut Solver<'rules> {
        let rule = EqualsZeroRule::new(items);
        self.rules.push(Box::new(rule));
        self
    }

    /// Adds rules to the solver once the value of an expression is known.
    ///
    /// For instance, one could write:
    /// ```text
    /// solver.given(input.rank, |solver, ir|
    ///     (0..ir).map(|i| solver.equals(input.shape[ir], 0))
    /// );
    pub fn given<T, E, C, A, F>(&mut self, item: A, closure: F) -> &mut Solver<'rules>
    where
        T: Output + Fact + 'static,
        E: Expression<Output = T> + 'static,
        C: Output + 'static,
        A: IntoExpression<E>,
        F: Fn(&mut Solver<'rules>, C) + 'rules,
    {
        let rule = GivenRule::new(item.into_expr(), closure);
        self.rules.push(Box::new(rule));
        self
    }
}
```

Appendix F: Solver rules for `Pad` *(Rust code)*.

```rust
impl<T: Datum> InferenceRulesOp for Pad<T> {
    fn rules(&self, inputs, outputs) {
        let input = &inputs[0];
        let paddings = &inputs[1];
        let output = &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])
                    ));
                })
            })
        ;
    }
}
```

Appendix G: Rust implementation of the constant folding algorithm.

```rust
use super::prelude::*;
use super::Result;
use ops::OpBuilder;
use tfpb;
use tfpb::tensor::TensorProto;
use Node;

/// All constant tensors with an area lower than COPY_THRESHOLD will be
/// replaced with a constant node containing a copy of that tensor.
const COPY_THRESHOLD: usize = 100;

#[derive(Debug)]
pub enum Element {
    Node(usize),
    Edge(usize),
}

#[derive(Debug)]
pub struct Component {
    pub elements: Vec<Element>,
    pub outputs: Vec<usize>,
}

/// Computes all the connected components of the constant underlying graph.
///
/// The constant underlying graph G is constructed using these rules:
/// - If an edge has a constant value according to the analyser, it is in G.
/// - If all the outgoing edges of a node are in G, that node is also in G.
/// - If an edge in G has no target, it is called an "output".
pub fn connected_components(analyser: &Analyser) -> Result<Vec<Component>> {
    let is_edge_const: Vec<bool> = analyser
        .edges
        .iter()
        .map(|e| e.fact.value.is_concrete())
        .collect();

    let is_node_const: Vec<bool> = analyser
        .next_edges
        .iter()
        .map(|next| next.len() > 0 && next.iter().all(|i| is_edge_const[*i]))
        .collect();

    let mut components = vec![];
    let mut is_node_colored = vec![false; analyser.nodes.len()];
    let mut is_edge_colored = vec![false; analyser.edges.len()];
    let mut stack = vec![];

    macro_rules! process_edges {
        ($from:ident, $field:ident, $component:expr, $node:expr) => {{
            for &edge in &analyser.$from[$node] {
                if !is_edge_const[edge] || is_edge_colored[edge] {
                    continue;
                }

                is_edge_colored[edge] = true;
                $component.elements.push(Element::Edge(edge));

                let target = analyser.edges[edge].$field;
```

```
                if target.is_none() {
                    continue;
                }

                if !is_node_const[target.unwrap()] {
                    $component.outputs.push(edge);
                } else {
                    stack.push(target.unwrap());
                }
            }
        }};
    };

    for (node, &is_const) in is_node_const.iter().enumerate() {
        if is_const && !is_node_colored[node] {
            let mut component = Component {
                elements: vec![],
                outputs: vec![],
            };

            stack.push(node);

            while let Some(node) = stack.pop() {
                if !is_node_const[node] || is_node_colored[node] {
                    continue;
                }

                is_node_colored[node] = true;
                component.elements.push(Element::Node(node));

                process_edges!(prev_edges, from_node, component, node);
                process_edges!(next_edges, to_node, component, node);
            }

            components.push(component);
        }
    }

    Ok(components)
}

/// Creates a new Const node with the given Tensor value.
fn build_const_node(id: usize, name: String, tensor: TensorProto) -> Node {
    let node_def = tfpb::node()
        .name(name.clone())
        .op("Const")
        .attr("dtype", tensor.get_dtype())
        .attr("value", tensor);

    Node {
        id,
        name,
        op_name: "Const".to_string(),
        inputs: vec![],
        op: OpBuilder::new().build(&node_def).unwrap(),
    }
}

/// Detaches the constant nodes and edges from the given graph.
///
/// The following algorithm is used:
/// 1. Compute the constant underlying graph of the given graph.
/// 2. Compute the undirected connected components of that underlying graph.
```

```rust
/// 3. Choose a pruning strategy and apply it to each connected component.
///
/// There are several pruning strategies to choose from:
/// - The simplest is to prune all nodes but the sinks of each component, and
///   to replace the latter with Const nodes. This might however increase the
///   size of the model dramatically in cases like the one below, where we'll
///   end up storing two large constants instead of one while only getting a
///   neglectible performance boost from the operation.
///
/// ```text
///                                    +---------------------+
///                                +--^+ Simple operation 1  +-->
///             +---------------+  |   +---------------------+
///             | Const (large) +--+
///             +---------------+  |   +---------------------+
///                                +--^+ Simple operation 2  +-->
///                                    +---------------------+
/// ```
///
/// - We could also search for the lowest common ancestor of all the sinks in
///   each connected component, and prune every node and edge that isn't part
///   of a path between that ancestor and a sink. If no such ancestor exists,
///   we don't do anything. This way we guarantee that we don't increase the
///   size of the model, but we might miss some optimisations.
///
/// - Ideally, we would use a heuristic to find a middle ground between the
///   two strategies. This would allow the duplication of constants if the
///   size or performance gained from pruning compensates the size loss.
pub fn propagate_constants(analyser: &mut Analyser) -> Result<()> {
    let components: Vec<Component> = connected_components(analyser)?;
    info!("Detected {:?} connected components.", components.len());

    for component in components {
        for i in component.outputs {
            let edge = &mut analyser.edges[i];
            let tensor = edge.fact.value.concretize().unwrap().to_pb().unwrap();

            let node_id = analyser.nodes.len();
            let node_name = format!("generated_{}", i).to_string();
            let node = build_const_node(node_id, node_name, tensor);
            let old_node_id = edge.from_node.unwrap();

            // Detach the edge from its previous source.
            {
                let successors = &mut analyser.next_edges[old_node_id];
                let position = successors.iter().position(|&i| i == edge.id).unwrap();
                successors.remove(position);
            };

            // Detach the target node from its previous source.
            {
                let predecessors = &mut analyser.nodes[edge.to_node.unwrap()].inputs;
                let position = predecessors
                    .iter()
                    .position(|&(i, _)| i == old_node_id)
                    .unwrap();
                predecessors[position] = (node_id, None);
            }

            // Attach the edge to its new source.
            edge.from_node = Some(node_id);
            analyser.prev_edges.push(vec![]);
            analyser.next_edges.push(vec![edge.id]);
```

```
                analyser.nodes.push(node);
            }
        }

        analyser.reset_plan()?;

        Ok(())
    }
```

Appendix H: Rust implementation of $step_{\texttt{Conv2D}}$.

```rust
/// Evaluates one step of the operation on the given input tensors.
fn step(
    &self,
    mut inputs: Vec<(Option<usize>, Option<TensorView>)>,
    buffer: &mut Box<OpBuffer>,
) -> Result<Option<Vec<TensorView>>> {
    // We only support the VALID padding strategy for now, with the
    // streaming dimension being either the width or the height.

    // The idea is that, regardless of the strides, we need at least
    // as many chunks in the buffer as the size of the filter in the
    // streaming dimension to compute our first output chunk. Then,
    // we pop the min(buffer_size, k) first chunks from the buffer,
    // ignore the next max(k - buffer_size, 0) chunks, and wait for
    // the k following chunks to compte one output chunk, with k the
    // strides in the streaming dimension.

    let (mut data, mut filter) = args_2!(inputs);

    if filter.0.is_some() || filter.1.is_none() {
        bail!("Filter input should not be streamed.");
    }

    if data.0.is_none()  {
        bail!("Data input should be streamed.");
    }

    // Maybe there is no incoming chunk.
    if data.1.is_none() {
        return Ok(None);
    }

    // Maybe the data is streamed along the batch dimension.
    let dim = data.0.unwrap();
    if dim == 0 {
        let result = self.eval(vec![
            data.1.take().unwrap(),
            filter.1.take().unwrap()
        ])?;

        return Ok(Some(result))
    }

    if dim < 1 || dim > 2 {
        bail!("Conv2D only supports batch, width and height streaming.");
    }

    let data = data.1.take().unwrap().into_tensor();
    let data = into_4d(T::tensor_into_array(data)?)?;
    let data_size = data.shape()[dim];
    debug_assert!(data_size == 1);

    let filter = filter.1.take().unwrap();
    let filter = T::tensor_to_view(&*filter)?;
    let filter_size = filter.shape()[dim - 1];

    // Generates an empty 4-dimensional array of the right shape.
    let empty_array = || {
```

```
            match dim {
                1 => Array::zeros((data.shape()[0], 0, data.shape()[2], data.shape()[3])),
                2 => Array::zeros((data.shape()[0], data.shape()[1], 0, data.shape()[3])),
                _ => panic!()
            }
        };

        let buffer = buffer.downcast_mut::<Buffer<T>>()
            .ok_or("The buffer can't be downcasted to Buffer<T>.")?;

        if buffer.prev.is_none() {
            buffer.prev = Some(empty_array());
        }

        let skip = &mut buffer.skip;
        let prev = buffer.prev.as_mut().unwrap();

        if *skip > 0 {
            *skip -= 1;
            return Ok(None)
        }

        let mut next = stack(Axis(dim), &[prev.view(), data.view()])?;
        let next_size = next.shape()[dim];

        // Maybe we don't have enough chunks to compute the convolution yet.
        if next_size < filter_size {
            *skip = 0;
            *prev = next;
            return Ok(None)
        }

        // Otherwise we compute the convolution using the non-streaming implementation.
        let result = self.convolve(&next, filter, dim != 1, dim != 2)?.into_dyn();
        let stride = [self.0.v_stride, self.0.h_stride][dim - 1];

        if stride > next_size {
            // Maybe we must pop more chunks from the buffer than it currently contains.
            *skip = stride - next_size;
            *prev = empty_array();
        } else {
            // Otherwise we pop the right number of chunks to prepare the next iteration.
            next.slice_axis_inplace(Axis(dim), Slice::from(stride..));
            *skip = 0;
            *prev = next;
        }

        Ok(Some(vec![T::array_into_tensor(result).into()]))
    }
```

▷ *Edited for clarity, see the full version at* `https://github.com/kali/tensorflow-deploy-rust/blob/master/src/ops/nn/conv2d.rs`.

Appendix I: Rust implementation of the streaming inference algorithm.

```rust
/// The state of a model during streaming evaluation.
#[derive(Clone)]
pub struct StreamingState {
    model: Model,
    output: usize,
    mapping: Vec<Option<usize>>,
    buffers: Vec<Box<OpBuffer>>,
    dimensions: HashMap<(usize, usize), usize>,
    successors: Vec<Vec<(usize, usize)>>,
}

/// The type of an input during streaming evaluation.
#[derive(Debug, Clone)]
pub enum StreamingInput {
    // The input is being streamed. We pass its datatype and shape along,
    // using None to denote the streaming dimension.
    Streamed(tfpb::types::DataType, Vec<Option<usize>>),

    // The input will remain constant during the evaluation.
    Constant(Tensor),
}

impl StreamingState {
    /// Initializes the streaming evaluation of a model.
    ///
    /// For each input in the model, you must either provide a constant
    /// value or specify the dimension along which to stream.
    ///
    /// You will only be able to fetch the results of the evaluation step
    /// for the output node. If `output` is None, the output node will be
    /// guessed automatically.
    pub fn start(
        model: Model,
        inputs: Vec<(usize, StreamingInput)>,
        output: Option<usize>,
    ) -> Result<StreamingState> {
        use StreamingInput::*;

        let output = output
            .or(analyser::detect_output(&model)?)
            .ok_or("Unable to auto-detect output node.")?;

        let mut analyser = Analyser::new(model, output)?;

        // Pre-compute the constant part of the graph using the analyser.
        for input in inputs {
            match input {
                (i, Streamed(dt, shape)) =>
                    analyser.hint(i, &TensorFact {
                        datatype: typefact!(dt),
                        shape: shape.iter().cloned().collect(),
                        value: valuefact!(_),
                    })?,
                (i, Constant(tensor)) => analyser.hint(i, &tensor_to_fact(tensor))?,
            }
        }

        analyser.run()?;
```

```
        analyser.propagate_constants()?;

        // Keep track of the relation between old and new node indexes, as the
        // analyser replaces the constant parts of the graph with Const nodes.
        let mapping = analyser.prune_unused();
        let output = mapping[output].ok_or("The output node doesn't exist in the streaming graph.")?;

        let successors = analyser.next_edges.iter()
            .map(|s| {
                s.iter()
                    .filter_map(|&e| {
                        let e = &analyser.edges[e];
                        e.to_node.map(|dest| (e.from_out, dest))
                    })
                    .collect::<Vec<_>>()
            })
            .collect::<Vec<_>>();

        let buffers = analyser.nodes.iter()
            .map(|n| n.op.new_buffer())
            .collect::<Vec<_>>();

        let mut dimensions = HashMap::with_capacity(analyser.edges.len());
        for edge in &analyser.edges {
            let source = &analyser.nodes[edge.from_node.unwrap()];
            let streamed = edge.fact.shape.dims.iter().position(|d| d.is_streamed());

            if source.op_name != "Const" && streamed.is_some() {
                debug!(
                    "Found streaming dimension {:?} for ({}, {:?}).",
                    streamed.unwrap(),
                    source.name,
                    edge.from_out,
                );

                dimensions.insert((source.id, edge.from_out), streamed.unwrap());
            }
        }

        let model = analyser.into_model();

        Ok(StreamingState {model, output, mapping, buffers, dimensions, successors})
    }

    /// Runs one streaming evaluation step.
    ///
    /// The step starts by feeding a new chunk of data into one of the
    /// non-constant inputs of the model, which gets propagated to all
    /// the nodes in the graph in breadth-first ordering.
    ///
    /// The method will return a Vec<Vec<Tensor>>, which will contain
    /// a Vec<Tensor> for every chunk that was produced by the output
    /// during the evaluation step, with one Tensor per output port.
    pub fn step(&mut self, input: usize, input_chunk: Tensor) -> Result<Vec<Vec<Tensor>>> {
        let mut queue = VecDeque::new();
        let mut outputs = vec![];

        let input = self.mapping[input].ok_or("The input node doesn't exist in the streaming graph.")?;
        let input_view = Into::<TensorView>::into(input_chunk).into_shared();

        for &(port, target) in &self.successors[input] {
            queue.push_back((input, port, target, input_view.clone()));
        }
```

```
while let Some((source, port, target, chunk)) = queue.pop_front() {
    debug!(
        "Executing new edge: source={:?}, port={:?}, target={:?}, chunk={:?}",
        source, port, target, chunk
    );

    let target = self.model.get_node_by_id(target)?;
    let mut inputs = vec![];

    // We wrap chunk in an option because we want to capture
    // its value in one and only one of the iterations, but
    // the borrow checker doesn't know that.
    let mut chunk = Some(chunk);

    for &(k, kp) in &target.inputs {
        let pred = self.model.get_node_by_id(k)?;
        let dimension = self.dimensions.get(&(k, kp.unwrap_or(0))).map(|i| *i);

        let value = if pred.op_name == "Const" {
            // The input is not streamed, and so was turned into a constant
            // node by the analyser when performing StreamingState::start.
            Some(pred.op.eval(vec![])?.pop().unwrap())
        } else if k == source && kp.is_some() && kp.unwrap() == port {
            // The input is streamed, and we've got a new chunk to give it.
            // FIXME(liautaud): This doesn't work well if there are multiple
            // edges from node source to node k, because the condition above
            // will get verified for all edges but only one actually "holds"
            // the chunk. The others will be None, and the unwrap will fail.
            let chunk = chunk.take().unwrap();

            // We only allow chunks of size 1 along the streaming dimension.
            if chunk.as_tensor().shape()[dimension.unwrap()] != 1 {
                bail!("Trying to consume a chunk of size != 1 along the streaming dimension.");
            }

            Some(chunk)
        } else {
            // The input is streamed, but we don't have anything to give it yet.
            None
        };

        inputs.push((dimension, value));
    }

    let buffer = &mut self.buffers[target.id];

    if let Some(mut output_chunks) = target.op.step(inputs, buffer)? {
        if target.id == self.output {
            // If we've reached the output, just save the chunks.
            outputs.push(output_chunks.clone());
        }

        // Propagate the chunks to the successors.
        for &(port, successor) in &self.successors[target.id] {
            queue.push_back((target.id, port, successor, output_chunks[port].share()));
        }
    }
}

// Convert the output TensorViews to Tensors.
let outputs = outputs
    .into_iter()
```

```
            .map(|chunks| chunks
                .into_iter()
                .map(|tv| tv.into_tensor())
                .collect()
            )
            .collect();

        Ok(outputs)
    }

    /// Resets the model state.
    pub fn reset(&mut self) -> Result<()> {
        unimplemented!()
    }
}
```

▷ *Edited for clarity, see the full version at* `https://github.com/kali/tensorflow-deploy-rust/blob/master/src/lib.rs`.