



CONFERENCE

—NORTH AMERICA 2026—

Deploying Robot Policies Across Many Targets with ExecuTorch

Anthony Shoumikhin, Jacob Szwejbka
PyTorch Team at Meta

ExecuTorch: PyTorch inference on the device

Start in PyTorch. Deploy through common runtime APIs.

Your model

Use your eager model definition. Iterate on it freely.

Target optimization

Use PyTorch export, quantization, and hardware compilers.

On-device inference

Run compiled programs from Python or C++.

A common runtime approach for embedded models, perception, and advanced policies.

Why we're investing in robotics

- Diverse models** Perception, action policies, and language-conditioned models share the robot and have different hardware needs.
- Local compute** Memory, power, and latency budgets shape what can run locally.
- Experimentation** Policies evolve in PyTorch. ExecuTorch optimizes and deploys from that same model definition.

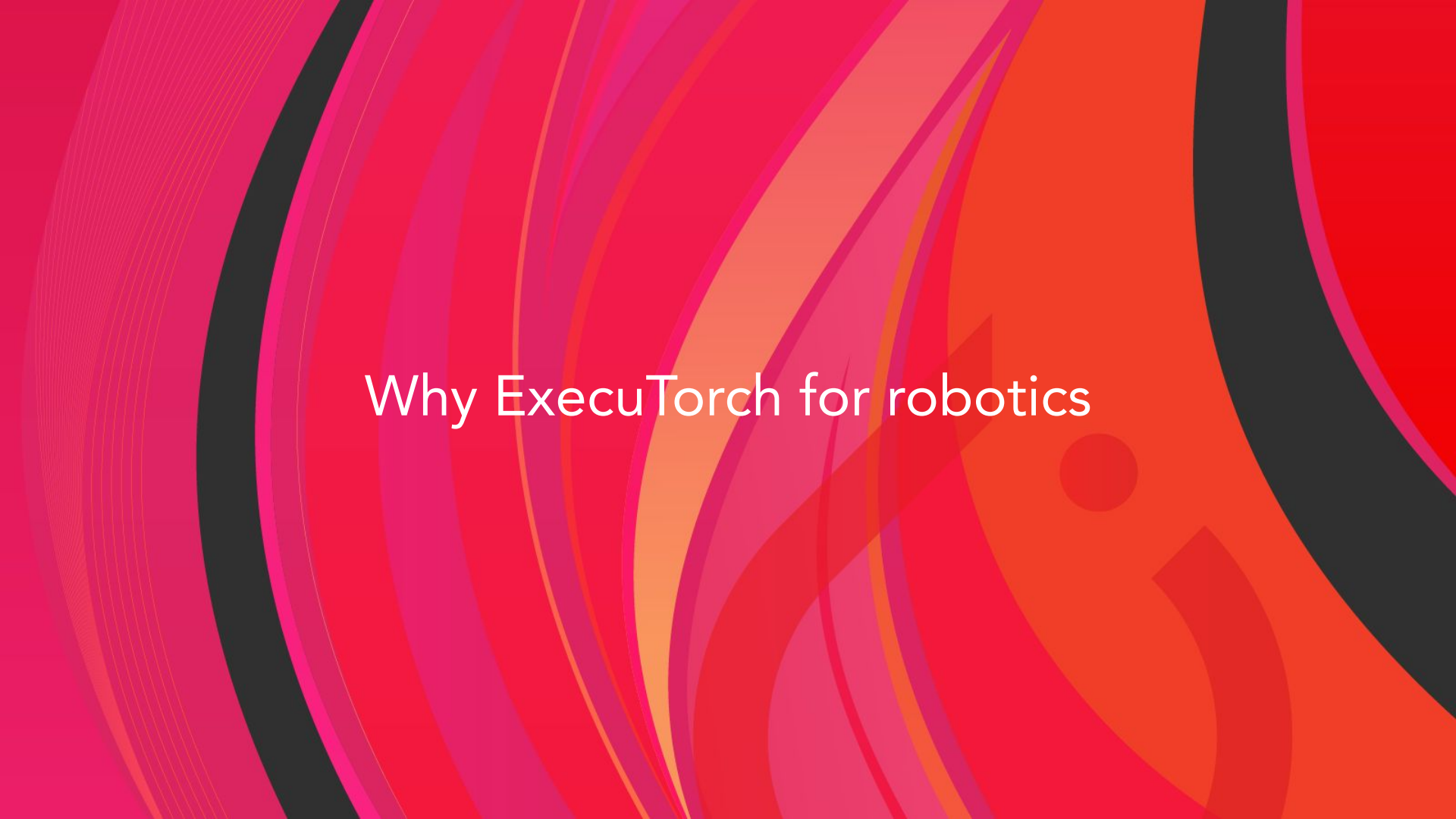
Landscape Today: Robotics Models

A policy maps observations to actions. An action chunk contains multiple future actions.

Model family	Inputs	Outputs	Compute
Vision-language-action	Images + instruction	Action sequence	GPU / NPU
Action policies	Observations	Action chunks	GPU / NPU
Motion controllers	State + targets	Motor commands	CPU / MCU
Perception	Cameras / sensors	Detections, depth, features	CPU / GPU / NPU

Landscape Today: Software and Deployment

- Training: PyTorch policies and robotics frameworks such as LeRobot
- Applications: sensor processing, policy inference, and action scheduling
- Systems: robot middleware, controllers, and hardware runtime libraries
- Deployment: direct PyTorch execution, ONNX, and vendor engines
- Integration: observation layouts, preprocessing, action units, and timing



Why ExecuTorch for robotics

Why ExecuTorch fits this workflow

Keep the model	Export from the PyTorch definition you use in eager evaluation.
Keep the tools	Export preprocessing and postprocessing with the model, and use PyTorch's quantization tools.
Reach the hardware	Lower for supported CPUs, GPUs, NPUs, and MCU targets with minimal framework overhead within the same PyTorch ecosystem.
AoT Specialization	Achieve peak performance for your limited hardware targets

One model definition, multiple deployment targets

PyTorch model + supported processing

Export, quantize, and lower for each target

CPU program

GPU program

NPU program

ExecuTorch runtime APIs in the application

Evaluate the program you deploy

Python evaluation loop

```
from executorch.runtime import Runtime

runtime = Runtime.get()
program = runtime.load_program(
    "policy.pt")
policy = program.load_method(
    "forward")
outputs = policy.execute(inputs)
```

C++ robot application

```
using executorch::extension::Module;

Module policy("policy.pt");
auto result = policy.forward(inputs);
if (!result.ok()) return 1;
auto outputs = result.get();
```

Use the same target-specific program with matching inputs and runtime dependencies.



Hardware collaborations

Hardware Partners

Compiler integrations, runtime improvements, and robot-policy experiments.

Intel / OpenVINO

Bring Intel hardware optimization into the ExecuTorch deployment path.

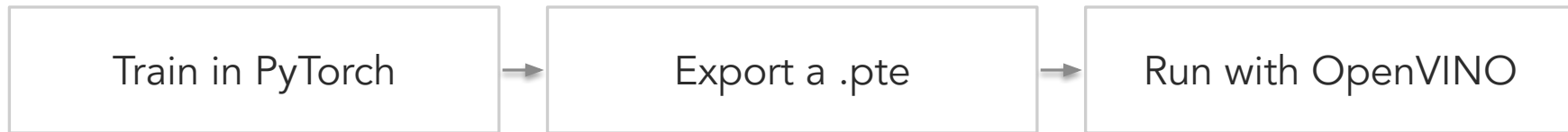
NVIDIA / Torch-TensorRT

Reuse TensorRT compilation and compose GPU execution through ExecuTorch.

Broader backend coverage includes NXP, Arm, Qualcomm, and many more.

Intel: Physical AI Studio and OpenVINO

From policy training to inference in the robot loop.



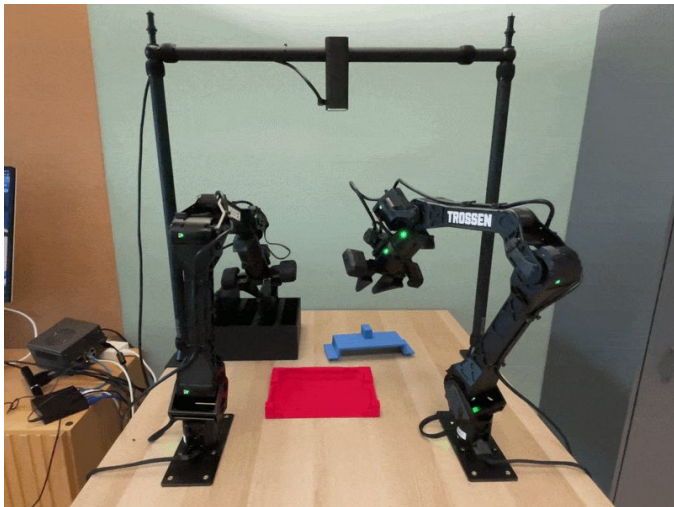
```
policy.export("./policy", backend="executorch")
```

The application closes the loop from camera and state inputs to actions.

Intel: policy inference in the robot loop

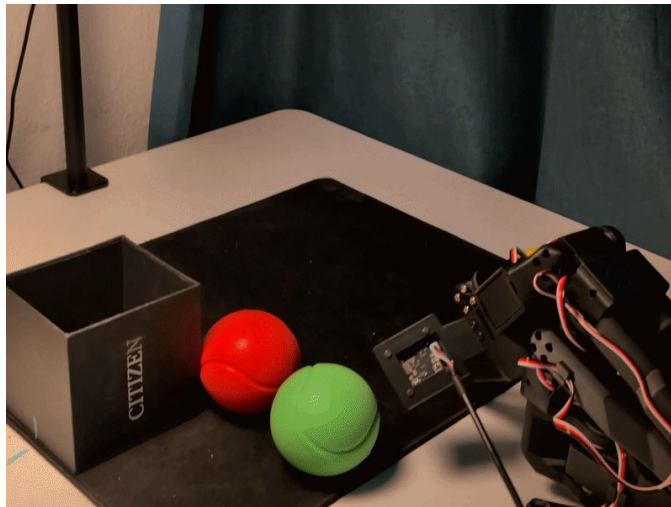
Tray-to-jig pick-and-place

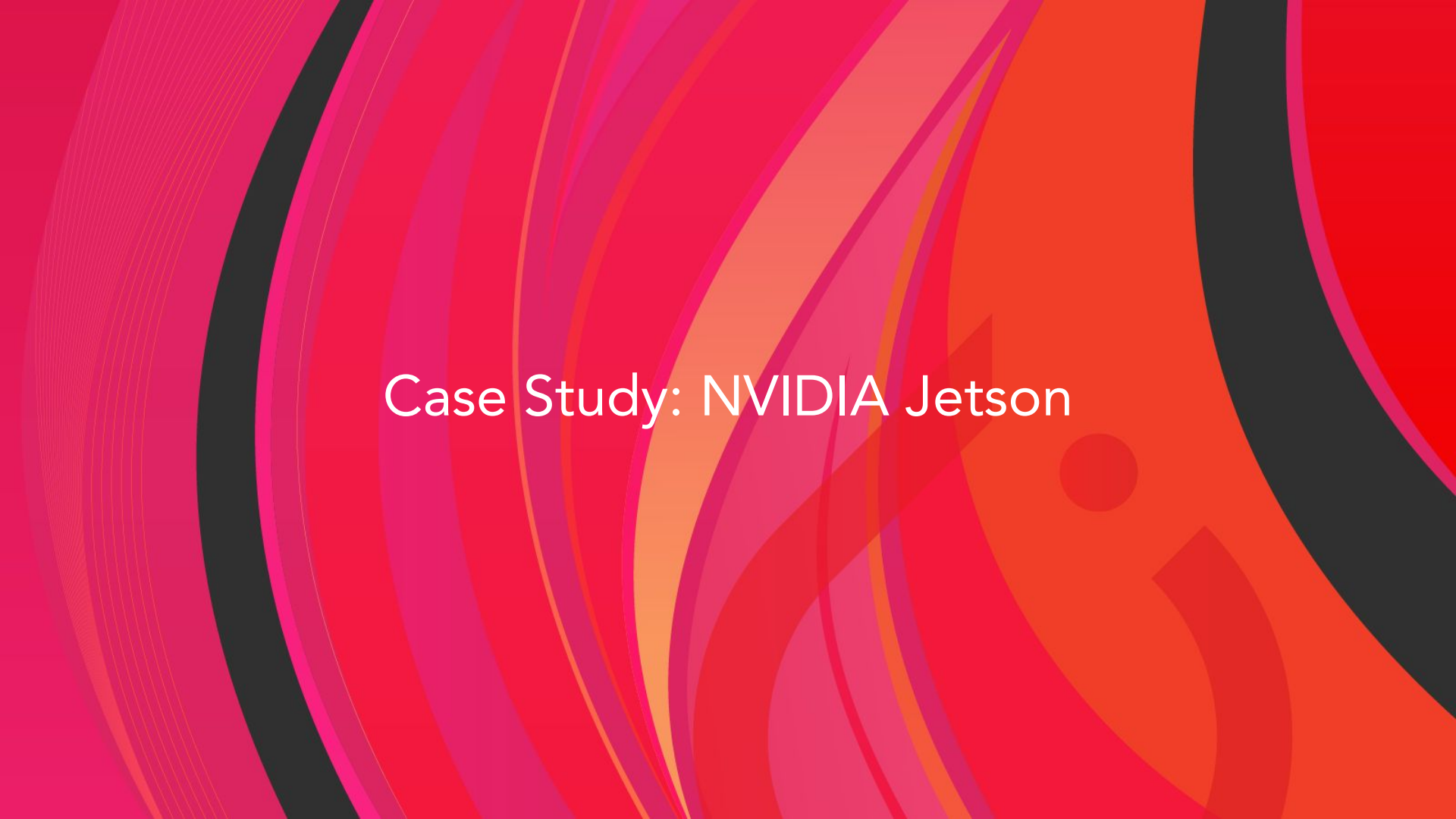
Trossen arms + Intel Core Ultra Series 3



Ball placement in a box

SO-101 arm + Intel Arc B580 graphics



The background is an abstract composition of fluid, overlapping curves in various shades of red, pink, and magenta. A prominent dark grey or black curved shape is visible on the right side, partially enclosing the text. The overall effect is dynamic and modern.

Case Study: NVIDIA Jetson

From PyTorch to a robot arm

Trained in PyTorch

Compiled by TensorRT

Run by ExecuTorch

pi0.5 on an 8 GB Jetson Orin Nano. Plain PyTorch runs out of memory

All it took

```
# Export: one script per policy
python examples/export/pi05_executorch_tensorrt.py \
    --policy.path=<checkpoint> ...
# Run: the same command as for a PyTorch checkpoint
lerobot-rollout --policy.path=<export folder> ...
```



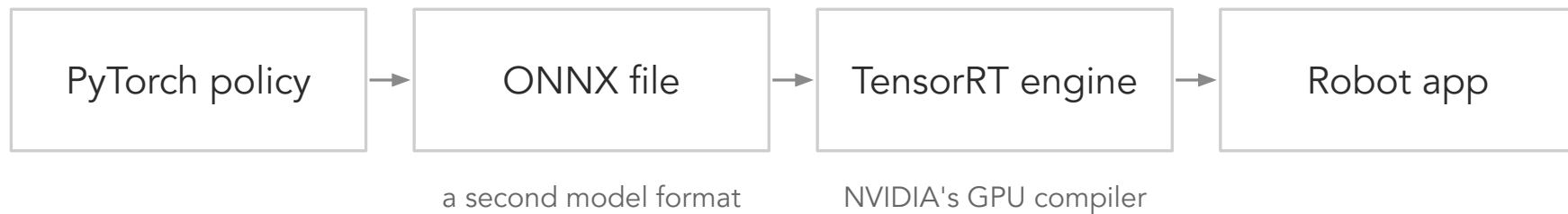
Scan for the code and the README

Ready-made scripts for **ACT**, **SmolVLA**, **pi0.5** and **GR00T N1.7**

Try it at our booth: the arm hands you a candy

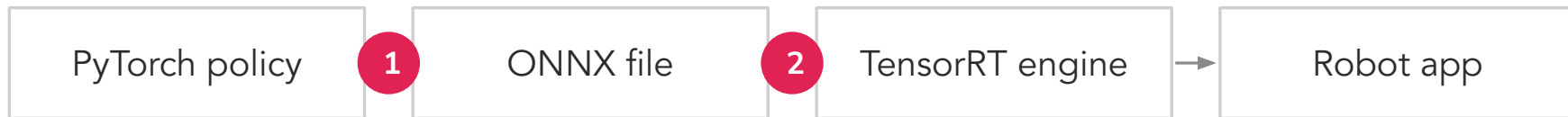


How a policy reaches a Jetson today



Fast and proven. This is the route to match

Two gates your model must pass



1 PyTorch's ONNX exporter

Every operation needs an ONNX translation, in an ONNX version the exporter can write

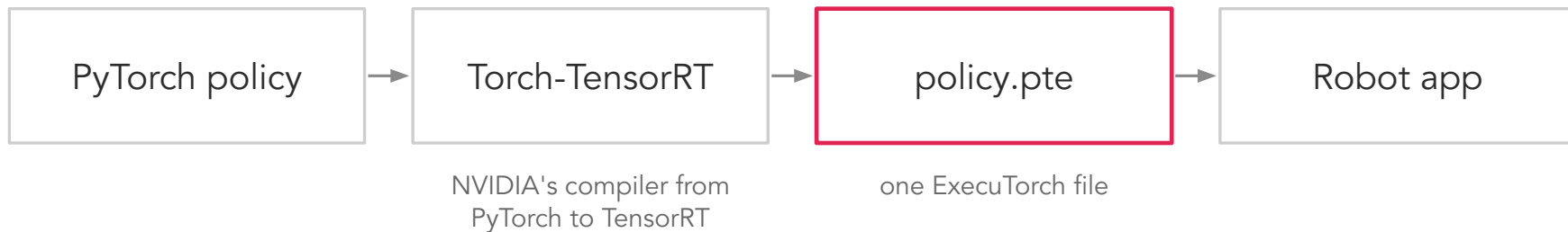
2 TensorRT's ONNX parser

One unknown node fails the whole build. The usual fix is a C++ plugin, often hundreds of lines

The problem: each new model can fail a gate you do not own

Sources: onnx/onnx-tensorrt operator matrix and parser source; NVIDIA/TensorRT plugin samples

The ExecuTorch route skips ONNX



No ONNX file

PyTorch's own export feeds TensorRT directly

Nothing left behind

What TensorRT cannot run goes to ExecuTorch's CUDA backend, in the same file

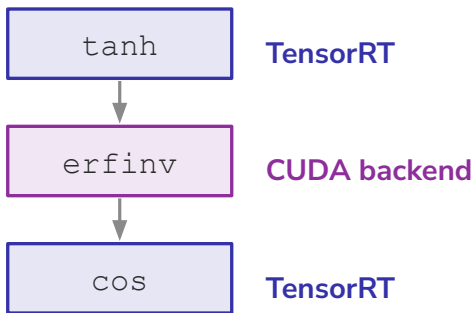
Data stays on the GPU

Inputs and outputs can live in GPU memory

Inside the export: split, compile, run

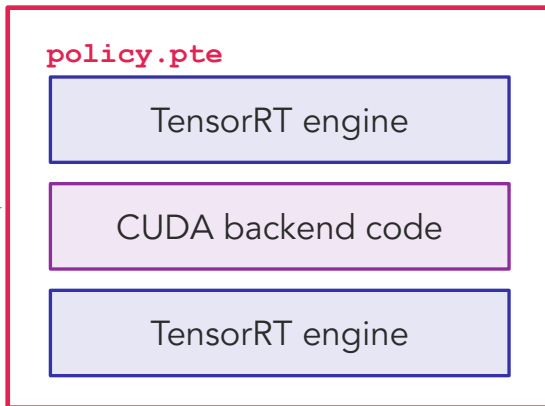
1 Capture and split

```
y = cos(erfinv(tanh(x)))
```



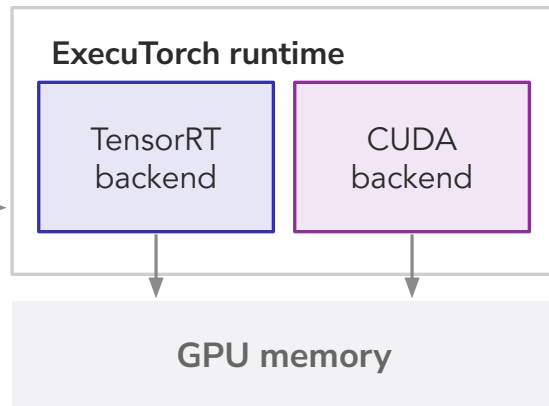
TensorRT has no converter for erfinv, so the CUDA backend takes it

2 Compile and pack



TensorRT builds the engines. The CUDA backend compiles the rest. Both go in one file

3 Load and run



C++ or Python, no PyTorch. Data stays on the GPU between the parts

Deployment workflow

Install Nightly wheels today. Stable releases at the end of October

```
pip install "torch-tensorrt[executorch]"
```

Export, in Python

```
trt = torch_tensorrt.compile(  
    policy, inputs=inputs)  
  
torch_tensorrt.save(  
    trt, "policy.pt",  
    output_format="executorch")
```

Run, in C++

```
Module policy("policy.pt");  
auto actions = policy.forward(obs);
```

Link, with CMake

```
find_package(executorch ...)  
find_package(executorch_backend_tensorrt)
```

Python runs it too, without PyTorch

What ships in Torch-TensorRT 2.15

Built with NVIDIA's Torch-TensorRT team: **Naren Dasan, Lan Luo and Adrian Wang**

Prebuilt for Jetson	pip install on Linux Arm64 and x86-64, and Windows x86-64
One file, many parts	TensorRT engines, CUDA backend code, several methods
Big policies	Engines over 2 GB. Weights can sit in a .ptd beside the .pte
Your CUDA stream	Even inside a green context: a fixed slice of the GPU
CUDA graphs	Replay a whole engine with one launch. Opt-in
Shared engines	Load a policy twice, keep its weights once: 122 MiB, not 806
Small-board memory	One activation buffer for many engines. Weight streaming

In nightly wheels now. Stable with PyTorch 2.15 at the end of October

ONNX-TensorRT speed, on an 8 GB board



Image: NVIDIA

Jetson Orin Nano Super. Milliseconds per action chunk, lower is better. Not quantized

Policy	ExecuTorch-TensorRT	ONNX-TensorRT	Torch-TensorRT	PyTorch
ACT	25	25	26	64
SmolVLA	125	121	137	749
pi0.5	630	612	out of memory	out of memory
GR00T N1.7	266	274	275, needs swap	out of memory

Median of 3 runs of 100 chunks. Bold: best in the row. Source: github.com/shoumikhin/lerobot, export branch

The same memory as a bare TensorRT engine

Orin Nano Super. Memory the whole board uses while the policy runs, MiB

Policy	ExecuTorch-TensorRT .pte	ONNX-TensorRT bare engine	Torch-TensorRT .pt2, needs PyTorch	PyTorch no compiler
ACT	207	222	841	1136
SmolVLA	373	475	1764	2447
pi0.5	4983	4882	out of memory	out of memory
GR00T N1.7	4402	4408	7534, needs swap	out of memory

The CPU and GPU share this memory on a Jetson. Bold is best in the row. Source: github.com/shoumikhin/lerobot, export branch

Large policies fit on the 8 GB board

pi0.5: 9 programs



GR00T N1.7: 9 programs



One engine build needs the whole model in memory. One program at a time fits in 8 GB

Where the whole policy fits, one program is faster

On Jetson Thor it ran 10 to 15% faster than the chain, in earlier runs

Next: fewer programs, maybe just one

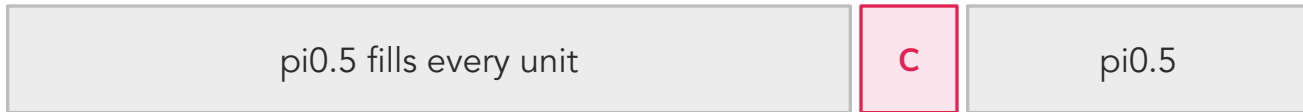


Jetson AGX Thor. Image: NVIDIA

Critical policies keep their own part of the GPU

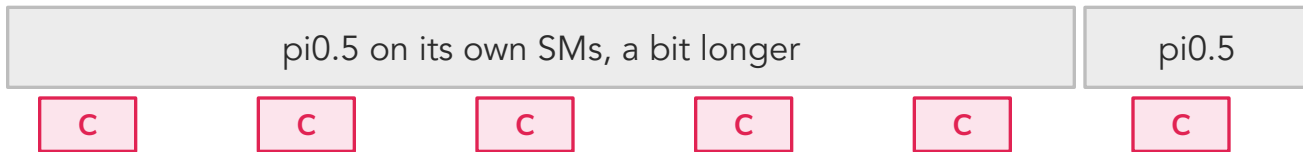
A green context gives a policy a fixed set of the GPU's compute units (SMs)

Without



The controller (C) waits for a gap

With green contexts



The controller, on its own SMs, always finds free units

```
using namespace executorch::extension::cuda;  
CallerStreamGuard guard(green_stream); // only its SMs  
method->execute();
```

Works with both the TensorRT and
CUDA backends

Source: NVIDIA's CUDA guide, green contexts. Other GPU resources stay shared, so delays shrink but do not vanish

Why ExecuTorch on a Jetson

Same speed and memory

as ONNX-TensorRT, the route used today

No ONNX in the middle

No second operator list. No plugins to write

Nothing left behind

What TensorRT cannot run still runs on the GPU

Small runtime

No PyTorch on the robot. C++ or Python

Many policies, one board

Shared engines. Own GPU slice for control

Stay in PyTorch

One toolchain for NVIDIA, Intel, Apple and more

Backup: the same picture on Jetson AGX Thor

Milliseconds per chunk				Board memory, MiB			
Policy	ExecuTorch-TensorRT	ONNX-TensorRT	PyTorch	Policy	ExecuTorch-TensorRT	ONNX-TensorRT	Torch-TensorRT
ACT	3	3	16	ACT	545	551	1523
SmolVLA	29	30	164	SmolVLA	1247	1310	2836
pi0.5	125	115	237	pi0.5	6050	5932	7600
GR00T	94	96	211	GR00T	5405	5420	6068

Source: github.com/shoumikhin/lerobot, export branch. Median of 3 runs of 100 chunks



What we're building next

LeRobot export and rollout prototype

Step	Prototype behavior
Export	Wrap the policy; capture processing and inference
Package	Write program(s), metadata, and a reference test case
Check	Compare compiled actions with saved eager outputs
Run	<code>lerobot-rollout --policy.path=<export-folder></code>

```
python examples/export/groot_executorch_tensorrt.py \  
  --policy.path="$CHECKPOINT" \  
  --dataset.repo_id="$DATASET" \  
  --output_dir="$EXPORT_DIR"
```

Working NVIDIA prototype;
collaborating with Hugging Face on
upstream integration

What we're building toward

Frameworks

Export commands and compiled-policy adapters.

LeRobot, Isaac Lab, mjlab, openpi, Isaac-GR00T

Policy execution

Reusable conditioning and repeated action-head execution.

Robot applications

ROS 2 nodes for inference and action-chunk handoff.

Validate a common interface on hardware from multiple vendors.

Summary

- Robots run many models on many chips; PyTorch is where they're trained
- ExecuTorch: one export path from the policy's own PyTorch code to every chip
- Same .pte in your eval loop and on the robot.
- Many of our hardware partners are investing heavily in this space
- NVIDIA: Torch-TensorRT exports .pte today; GR00T N1.7 runs on an 8 GB Orin Nano
- Next: VLA sessions, framework and ROS 2 integrations, robot runtime features

Thank you

- docs.pytorch.org/executorch
- github.com/pytorch/executorch
- docs.pytorch.org/TensorRT
- github.com/shoumikhin/lerobot — export branch