



Export Pipelines: PyTorch to Every Runtime

Edge deployment | torch.export and the ONNX exporter, opsets, then Core ML, LiteRT, TensorRT, OpenVINO and ExecuTorch with their commands, operator-coverage and dynamic-shape pitfalls, and a parity check (versions as of 2026)

#ONNX #CoreML #LiteRT #TensorRT #OpenVINO #ExecuTorch

One model, many runtimes

Translating a novel into several languages

The author's manuscript with margin notes

A clean typeset master copy

One translator per language

Idioms with no equivalent

Proof-reading against the original

≈

Exporting a PyTorch model

Python model code with control flow

torch.export graph or ONNX file

One converter per runtime

Operators the target does not support

Numerical parity test on real inputs

WHERE THE ANALOGY BREAKS

A translator can paraphrase a missing idiom; a converter either rewrites the operator into supported ones or leaves it on the CPU, silently slowing the model.

Exporting as translating a book

Before Ready to export, can you explain...

	Not yet	Partly	Yes
MUST KNOW model.eval() and no training-only branches	☐	☐	☐
MUST KNOW Know which inputs vary in shape	☐	☐	☐
MUST KNOW ONNX graph, nodes, initialisers	☐	☐	☐
HELPFUL Delegates and execution providers	☐	☐	☐
HELPFUL fp16 and int8 formats	☐	☐	☐

Tool	Input format	Entry from PyTorch	NPU / accel.	Dynamic shapes	Targets
ONNX Runtime	.onnx	torch.onnx.export	⚠	✓	all OS, web
Core ML	.mlpackage	coremltools	✓	⚠	Apple only
LiteRT	.tflite	ai-edge-torch	⚠	⚠	Android, iOS, MCU
TensorRT	.engine	ONNX → trtexec	✓	profiles	NVIDIA GPU, Jet...
OpenVINO	.xml + .bin	ov.convert_model	✓	✓	Intel CPU/GPU/N...
ExecuTorch	.pte	torch.export	⚠	⚠	mobile, embedded

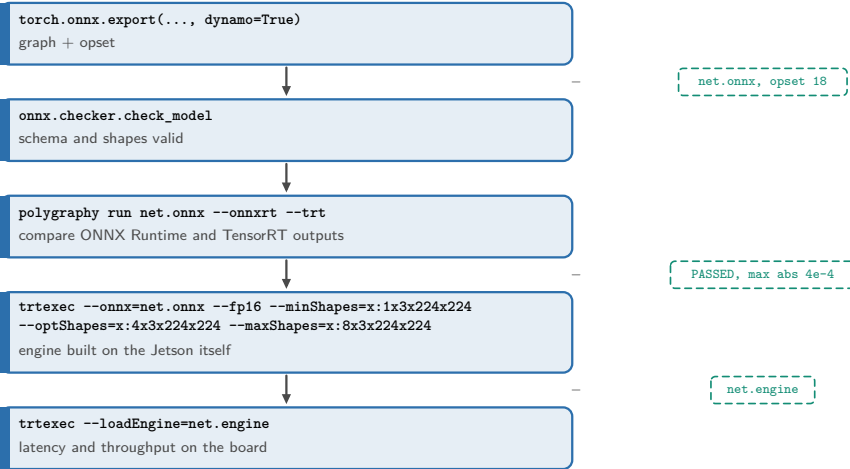
✓ yes / full ⚠ partial ✗ no / none

Six runtimes against what deployment needs (support as of 2026; partial = vendor- or version-dependent)

The commands, runtime by runtime

```
Step 0: export the graph once (PyTorch 2.x)
import torch
model = Net().eval()
x = torch.randn(1, 3, 224, 224)
batch = torch.export.Dim("batch", min=1, max=8)
ep = torch.export.export(
    model, (x,), dynamic_shapes={"x": {0: batch}})
# ONNX via the dynamo exporter
onnx_prog = torch.onnx.export(model, (x,), dynamo=True,
    dynamic_shapes={"x": {0: batch}})
onnx_prog.save("net.onnx")
```

Command	What it does
<code>python -m onnxruntime.tools.check_onnx_model_mobile_ready net.onnx</code>	ONNX Runtime: lists operators and opset the mobile build supports
<code>ct.convert(ep, minimum_deployment_target=ct.target.iOS17)</code>	Core ML (coremltools): ExportedProgram or traced model → .mlpackage
<code>ai_edge_torch.convert(model, (x,)).export('net.tflite')</code>	LiteRT: PyTorch → .tflite, NHWC layout handled by the converter
<code>trtexec --onnx=net.onnx --fp16 --save Engine=net.engine</code>	TensorRT: builds a GPU-specific engine from ONNX; add shape flags for dynamic inputs
<code>ovc net.onnx --output_model net.xml</code>	OpenVINO: ONNX (or PyTorch via <code>ov.convert_model</code>) → IR; weights saved in fp16 by default
<code>to_edge_transform_and_lower(ep, partitioner=[XnnpackPartitioner()])</code>	ExecuTorch: lower the graph to a backend, then <code>.to_executorch()</code> and write the .pte buffer



The same chain for a Jetson board: every stage has its own failure mode

⚠️ **Build where you run**

A TensorRT engine is tied to the GPU architecture and TensorRT version that built it: build on the Jetson (or the same GPU family and JetPack), never copy an engine from a data-centre GPU. Core ML compiles .mlpackage to .mlmodelc per device; OpenVINO and ONNX Runtime compile per session, so cache the compiled blob to cut cold start.

🔧 **What breaks: opsets, operators, shapes**

⚠️ Export fails or the graph is frozen to one input HIGH COMMON		
LIKELY CAUSES - Python control flow on tensor values (if <code>x.sum() > 0</code>) - Shapes read with <code>.item()</code> or <code>len()</code> on a tensor	HOW TO CONFIRM Run <code>torch.export</code> with strict mode and read the error trace <pre>torch.export.export(model, (x,), strict=True)</pre>	FIX - Replace data-dependent if with <code>torch.where</code> or <code>torch.cond</code> - Declare dynamic dims with <code>torch.export.Dim</code>
⚠️ Runtime refuses to load the ONNX file HIGH COMMON		
LIKELY CAUSES - Opset newer than the runtime implements - Custom or contrib operator not registered	HOW TO CONFIRM Read the opset and node types <pre>python -c "import onnx;m=onnx.load('net.onnx');print(m.opset_import)"</pre>	FIX - Export with an opset the target runtime supports - Rewrite or register the custom operator
⚠️ Converted, but slower than expected on the NPU MEDIUM COMMON		
LIKELY CAUSES - Unsupported operators fall back to CPU, splitting the graph - Dynamic shapes force the CPU or GPU path	HOW TO CONFIRM Count partitions and the processor of each <pre>Xcode Core ML performance report; ORT log_severity_level=0</pre>	FIX - Replace the operator (e.g. GELU-tanh, fixed-size resize) - Fix input shapes or bucket them
⚠️ Outputs differ from PyTorch HIGH OCCASIONAL		
LIKELY CAUSES - Preprocessing mismatch: RGB/BGR, mean/std, resize mode - fp16 overflow in softmax, layer norm or large sums	HOW TO CONFIRM Feed identical tensors to both and compare <pre>np.testing.assert_allclose(ref, out, rtol=1e-3, atol=1e-4)</pre>	FIX - Move preprocessing into the graph or share one spec - Keep sensitive layers in fp32

The four export failures met most often

Runtime	Trap	Workaround
ONNX Runtime	Execution provider supports a subset of ops; rest run on CPU EP	Read the session log for node placement; pick ops the EP lists
Core ML	ONNX input is no longer supported by coreml-tools (since version 6)	Convert from PyTorch directly (traced or <code>torch.export</code>)
Core ML	Neural Engine needs fp16 and static or enumerated shapes	Use <code>ct.EnumeratedShapes</code> or <code>RangeDim</code> with few sizes
LiteRT	NHWC layout; NNAPI delegate deprecated in Android 15	Let the converter transpose; use GPU or vendor NPU delegates
TensorRT	Dynamic dims need an optimisation profile; int64 tensors cast to int32	Declare min/opt/max shapes; keep indices small
OpenVINO	NPU plug-in compiles static shapes best	Reshape the model to fixed input size before compiling
ExecuTorch	Ops not claimed by the partitioner run on the portable CPU kernels	Print the delegation summary; add a second backend or rewrite

Operator and layout traps by runtime (as of 2026)

PROBLEM

A segmentation model runs on the NPU in 6.0 ms when fully delegated. One unsupported resize operator in the middle falls back to the CPU. The CPU takes 1.5 ms for it, and each NPU ↔ CPU switch copies a 4 MB activation at 8 GB/s with 0.3 ms of synchronisation overhead. New latency?

1. NPU part

⇒ $6.0 - 0.2 = 5.8$ ms

why: the resize leaves the NPU

2. Switch cost

⇒ $2 \times (0.5 + 0.3) = 1.6$ ms

why: two partitions boundaries

3. Total

⇒ $5.8 + 1.5 + 1.6 = 8.9$ ms

why: sum of the serial parts

GIVEN

- $t_{\text{npu}} = 6.0$ ms (resize itself ≈ 0.2 ms of it on the NPU)
- $t_{\text{cpu}} = 1.5$ ms; switches $k = 2$ (to CPU and back)
- copy time = $4 \text{ MB} \div 8 \text{ GB/s} = 0.5$ ms; sync = 0.3 ms per switch

ANSWER

About 8.9 ms instead of 6.0 ms: one fallback operator costs about 48 % more latency

Check: Replacing the resize by a supported one (e.g. fixed-scale nearest) restores about 6 ms

Cost of one CPU fallback in the middle of a graph. Timings are illustrative assumptions

✓ Myths, recap and self-test

🗨 Interview favourite

Asked how to validate an export, name three checks: the runtime loads it (opset, operators), outputs match PyTorch within a tolerance on real samples, and the profiler shows every operator on the intended processor.

MYTH OR FACT?

“Always use the newest ONNX opset”

MYTH: Use the newest opset your target runtime supports
Mobile and NPU builds often lag the desktop runtime by several opsets.

MYTH OR FACT?

“One ONNX file serves every device”

MYTH: ONNX is portable, performance is not
Each EP or engine needs its own shapes, precision and sometimes rewritten operators.

MYTH OR FACT?

“A TensorRT engine can be shipped like a model file”

MYTH: It is built per GPU and TensorRT version
Ship ONNX and build the engine on or for the exact target.

Three export misreadings

Export checklist

Tick a box only when you could explain the point to someone else



torch.export first

fix graph breaks once, reuse for every converter



Pin opset and runtime versions

record them with the artifact



Declare or fix dynamic shapes

NPUs prefer static



Parity test on real inputs

rtol 1e-3 class tolerance, preprocessing included



Profile operator placement

zero unexpected CPU fallbacks

Q1 Common entry point of modern PyTorch converters?

A torch.export, which produces an ExportedProgram (ATen graph, no Python).

Q2 How to declare a variable batch dimension?

A torch.export.Dim('batch', min=1, max=8) passed in dynamic_shapes.

Q3 Why can a newer opset break deployment?

A The target runtime may not implement it yet and refuses to load the model.

Q4 Can coremltools convert ONNX today?

A No, ONNX input was dropped in coremltools 6; convert from PyTorch.

Q5 What does trtexec --saveEngine produce?

A A TensorRT engine valid only for that GPU architecture and TensorRT version.

Q6 OpenVINO IR files?

A .xml topology plus .bin weights.

Q7 ExecuTorch output file and lowering call?

A .pte, via to_edge_transform_and_lower with a partitioner, then to_executorch().

Q8 Most common cause of accuracy loss after export?

A Preprocessing mismatch (colour order, normalisation, resize) between training and the app.

Q9 Tool that compares ONNX Runtime and TensorRT outputs?

A Polygraphy (polygraphy run --onnxrt --trt).