ENGINEERING HANDBOOK

Model Refactoring

for Software Engineers

Profiling, pruning, distillation, quantization,
and mobile deployment

VISION / LANGUAGE / AUDIO
IMAGE DIFFUSION / VIDEO DIFFUSION

CRASH-COURSE EDITION · COMPLETE PACKAGE

25 September 2026
Android-developer crash course, tested CPU labs,
linked glossary, and HDemucs experiment plan

ABOUT THIS EDITION

Scope and evidence

Crash-course edition | Complete companion package | 25 September 2026

Review status: This expanded edition adds a twelve-lesson crash course for Android developers, original CPU exercises, and recorded correctness tests. It retains the seventeen main chapters, linked technical glossary, original experiment records, and proposed HDemucs plan. Synthetic and mechanism tests are not production-model validation. The new ONNX and Kotlin exercises are unexecuted integration examples, not device benchmarks.

From an existing trained model to a smaller, measurable, deployable system.

This handbook treats a model change as an engineering change: define the problem, inspect the implementation, make one controlled modification, test the result, and preserve a rollback path.

The worked examples use Python and PyTorch. The methods apply to experiments with vision, language, audio, image diffusion, and video diffusion. Android deployment is one target, not the definition of success. A desktop GPU experiment and an on-device product use the same evidence discipline but different resource budgets.

> **Reading this edition**
>
> Start with Chapter 0 when moving from app development into model engineering. The contents and PDF bookmarks navigate the full book. Linked terminology opens the glossary; evidence and source markers open their records or reference entries.

The ZIP includes the manuscript, original and new code, generated teaching checkpoints, experiment records, and build source. CPU mechanism tests do not establish production-model quality or phone performance. ONNX export and Kotlin integration remain unexecuted examples.

Contents

How to use this handbook

New to model engineering? Begin with Chapter 0, the crash course for Android developers. It includes a CPU-based learning lab and a route into the existing chapters. Read Chapters 1 through 3 before changing a production checkpoint. Chapters 4 through 11 form a catalog of transformations. Chapter 12 addresses image and video diffusion. Chapters 13 and 14 address export, runtimes, and device integration. Chapter 15 reports the executed labs. Chapters 16 and 17 provide experiment plans and review criteria. Appendix A explains how to run the companion code. Appendix C provides the technical glossary used by the linked terminology throughout the handbook. Appendix D applies the method to an HDemucs experiment plan.

Chapter 0 assumes software development experience and introduces the Python and tensor concepts needed for the lab. For the main transformation chapters, you should be comfortable reading Python, using version control, writing automated tests, and interpreting performance measurements. You do not need to derive backpropagation to use the examples. You do need to understand tensor shapes and distinguish a trained parameter from a value computed during inference.

Model refactoring is not always behavior-preserving. Removing an unused graph node can preserve the intended function. Deleting a learned block usually changes it. In the latter case, your regression test is a distribution of outcomes, not just equality on a few sample inputs. Keep these two kinds of change separate in code review. ONNX Runtime explicitly distinguishes semantics-preserving graph rewrites from other optimizations, including optional approximations. **[R15]**

Evidence labels

Documented means the behavior is described in official documentation or the referenced implementation. **Reported** means an external study describes a result; this handbook has not reproduced that study. **Executed** means the companion code was run in the preparation environment and its outputs are included. **Proposed** means the procedure is an engineering recommendation or a template that still needs testing in your environment.

References such as **[R14]** identify external primary sources in the reference directory. References **[E01]**, **[E02]**, and **[E03]** identify the original experiment records. Reference **[E04]** identifies the added crash-course run and fifteen correctness tests. A citation establishes the specific claim associated with it. It does not validate an entire application or guarantee a particular compression ratio.

What has and has not been tested

The executed lab uses **RefactorNet 1.0**, an original, small residual classifier trained on generated numerical data. It demonstrates block ablation, physical feed-forward pruning, recovery training, checkpoint reconstruction, and CPU timing. A second executed example demonstrates chunked evaluation of one causal convolution. A third tests convolution-channel pruning, attention-head pruning, tokenwise feed-forward chunking, image tiling, and a deliberately incorrect normalization change. The supplied records include software versions and raw measurements. **[E01]**, **[E02]**, **[E03]**

No Android device, NPU, pretrained diffusion checkpoint, production language or speech model, or task-specific media dataset was used in those experiments. The ONNX export and quantization template was not executed because the required packages were unavailable and package installation could not reach the network. Device deployment and production-model

recipes are therefore proposed work, not benchmark results. This is a practical draft with executed mechanism tests, not an independently reviewed claim of universal compression performance.

The laboratory runtime was PyTorch 2.10.0+cpu on Python 3.13.5. Online documentation can describe newer releases. Version-specific PyTorch references are used where available, and mutable documentation is identified by its access date. Do not assume that a newly installed package is equivalent to the tested environment. PyTorch also cautions that reproducibility is not guaranteed across releases and platforms. **[R10]**

The added TinyNet course in Chapter 0 was executed separately in the same Python and PyTorch versions. Its recorded results, checkpoints, shape trace, CPU timing samples, and fifteen correctness tests appear under `evidence/crash_course/` and `evidence/crash_course_tests.txt`. The course uses synthetic numerical classification data. It does not add a production-media or Android benchmark. The original records remain preserved rather than replaced by the new run. **[E04]**

0. Crash course for Android developers

You can start with the engineering skills you already use: trace data, check contracts, isolate a change, measure a result, and preserve a rollback. The new skill is understanding a program whose useful behavior depends on learned numerical parameters. This course connects that idea to a complete, deliberately small experiment before you modify a production checkpoint.

The course uses **TinyNet**, an original classifier with 16 numerical inputs and three output classes. It is not an image recognizer, speech model, or diffusion model. Its purpose is to make shapes, gradients, pruning, and recovery observable without downloading a dataset or a pretrained checkpoint. The package contains the implementation, generated example checkpoints, tests, and recorded results. **[E04]**

Run status: The Python learning lab and 15 structural tests were executed on CPU with Python 3.13.5 and PyTorch 2.10.0+cpu. The optional ONNX exporter and Kotlin integration example were not executed. No Android, GPU, NPU, HDemucs, or diffusion performance result is implied. **[E04]**

Course map

Lesson	Engineering deliverable
0.1 Set up a small, reproducible workspace	An isolated environment and a successful test run
0.2 Read Python as an Android engineer	A readable model class and a verified checkpoint
0.3 Think in tensors and contracts	Correct shapes, axis meanings, and payload calculations
0.4 Read a network one operation at a time	A shape trace and parameter count
0.5 Understand one training step	A working forward, loss, backward, and update cycle
0.6 Separate learning from evaluation	A split policy that prevents avoidable leakage
0.7 Profile before changing the model	A baseline record with narrowly defined measurements
0.8 Make your first structural edit	A smaller model plus tests of the edit itself
0.9 Recover the model and test distillation	A controlled comparison of two recovery methods
0.10 Cross the export and quantization boundary	An explicit conversion contract and parity checklist
0.11 Integrate and measure on Android	A deployment plan with a golden input and output
0.12 Choose the next model and graduate	A model-specific experiment proposal

These are skill gates, not a six-month prerequisite. Move forward when you can explain the result and reproduce the exercise. A course can introduce a method; it cannot establish that a particular production model will tolerate that method.

0.1 Set up a small, reproducible workspace

Goal: Run a model experiment without first solving a GPU, dataset-download, or Android build problem.

Extract the ZIP and open a terminal in `Model_Refactoring_Complete`. The core course uses the Python standard library and PyTorch. It does not require Android Studio, CUDA, ONNX, or a paid service. That statement applies to the supplied CPU lab, not to training arbitrary models. **[E04]**

Create a virtual environment so this experiment has its own installed packages. The environment is similar in purpose to separating toolchain dependencies between projects, although it does not make results independent of the operating system or processor. Python documents `venv` and `pip` as the relevant environment and package tools. **[R67]**

BASH

```
python -m venv .venv
```

On Windows PowerShell, invoke the environment's interpreter directly. This avoids requiring an activation-script policy change:

POWERSHELL

```
.\.venv\Scripts\python.exe -m pip install -r crash_course/requirements-cpu.txt --index-url
https://download.pytorch.org/whl/cpu
.\.venv\Scripts\python.exe -m unittest crash_course.test_contracts -v
.\.venv\Scripts\python.exe -m crash_course.run --out runs/first
```

On macOS or Linux:

BASH

```
.venv/bin/python -m pip install -r crash_course/requirements-cpu.txt --index-url https://
download.pytorch.org/whl/cpu
.venv/bin/python -m unittest crash_course.test_contracts -v
.venv/bin/python -m crash_course.run --out runs/first
```

The requirement pins the tested base version, `torch==2.10.0`. A suitable wheel must exist for your operating system, processor, and Python version. Installing packages requires network access or a local wheel cache. Running the lab after installation does not download data. Keep the tested environment and a new environment distinct in your reports.

The command refuses a nonempty output directory. Use `runs/second` for another experiment instead of replacing the evidence from `runs/first`. The included reference run is in `evidence/crash_course`; do not use that path for a rerun.

Exercise: Run the tests, then the learning lab. Open `report.json` and locate the Python version, PyTorch version, seed, selected candidate, source hashes, and final test results.

Exit check: You can explain which files are code, which are model parameters, and which are evidence. You have not mistaken a package installation failure for a model failure.

0.2 Read Python as an Android engineer

Goal: Read a model implementation without treating every unfamiliar Python expression as a new ML concept.

A Python class can own other objects, expose methods, and validate its inputs. `self` refers to the instance. `__init__` initializes it. A type hint such as `width: int` documents the expected type but does not itself enforce all runtime constraints. Assignment can give two names to the same mutable object. These are language concerns, not learning algorithms. **[R68]**

In this course, a PyTorch model is an `nn.Module`. Submodules assigned to module attributes participate in PyTorch's parameter traversal. A `ModuleList` registers its contained modules; a plain Python list is not a substitute when those modules need to appear in the model state and optimizer parameter collection. Calling `model(x)` invokes the module call machinery around `forward`, including registered hooks. **[R62]**

Read this actual component from `crash_course/models.py`:

PYTHON

```python
class ResidualFFN(nn.Module):
def __init__(self, width: int, inner: int) -> None:
super().__init__()
self.up = nn.Linear(width, inner)
self.act = nn.GELU()
self.down = nn.Linear(inner, width)
def forward(self, x: torch.Tensor) -> torch.Tensor:
return x + self.down(self.act(self.up(x)))
```

Start with the control flow. The input goes through `up`, `act`, and `down`; the result is added to the original input. The class construction determines the layer dimensions. Loading a checkpoint supplies values for those layers. Neither operation, by itself, teaches the network a task.

A useful Android analogy is a component implementation plus its persisted state. The analogy stops at interchangeability: changing a neural layer's dimensions usually invalidates some saved parameter shapes. Renaming a configuration value does not resize the checkpoint.

PYTHON

```python
from pathlib import Path
from crash_course.models import load_model
model = load_model(Path("evidence/crash_course/selected.pt"))
print(model)
print(model.config)
```

The loader reconstructs the exact TinyNet configuration and uses strict state-dictionary loading. Save architecture metadata with the weights, not just a filename such as `small.pt`. PyTorch's save/load tutorial distinguishes the state dictionary from the model class needed to interpret it. Load only trusted artifacts. **[R69]**

Exercise: Find `stem`, `blocks`, and `head` in the printed model. Confirm that the selected model has two residual blocks while the original teacher has three.

Exit check: You can identify where the architecture is defined and where the learned numbers enter that architecture. You use `deepcopy` for a pruning candidate rather than `candidate = baseline`, which would alias the original object. **[R68]**

0.3 Think in tensors and contracts

Goal: Know what every axis means before changing its length.

A tensor is a numerical array. In PyTorch it also has properties such as a dtype and a device. Shapes alone do not tell you whether an axis means color channels, time samples, tokens, or classes. Document those meanings as part of the input contract. A float tensor holding RGB values in the wrong order can be structurally valid and semantically wrong. **[R59]**

Common conventions you will encounter are shown below. These are conventions to verify, not universal layouts that every model accepts.

Example contract	Meaning
`[B, C, H, W]`	Batch, channels, image height, image width
`[B, C, T]`	Batch, audio channels, time samples
`[B, T, D]`	Batch, sequence positions, feature width
`[B, C, F, H, W]`	One possible video layout with frame axis `F`
`[B, 16] -> [B, 3]`	TinyNet's features and class scores

The first four rows describe possible application contracts. A video checkpoint may use another order or operate on latent representations rather than pixels. Its implementation is the authority.

The following is a payload calculation, not a peak-RAM measurement:

PYTHON

```python
import torch
image = torch.zeros(1, 3, 32, 32, dtype=torch.float32)
logical_bytes = image.numel() * image.element_size()
assert logical_bytes == 12_288
```

There are 3,072 values, each represented by four bytes. The logical payload is therefore 12,288 bytes. Runtime allocations, object metadata, retained intermediates, and workspace are not included. Views can also share underlying storage, so summing every visible tensor's payload can double-count memory. **[R50]**, **[E04]**

Changing shape is not the same as changing axis order. `permute` rearranges axes. `reshape` requests a new shape while preserving the relevant element ordering; it is not an RGB-to-planar conversion just because the final dimensions match. A reshape may return a view or require a copy, depending on the existing layout. **[R50]**

PYTHON

```python
x = torch.arange(24).reshape(1, 2, 3, 4)
correct = x.permute(0, 2, 3, 1)
wrong = x.reshape(1, 3, 4, 2)
assert correct.shape == wrong.shape
assert not torch.equal(correct, wrong)
```

That assertion is part of the executed tests. It models a deployment bug worth learning early: both sides agree on dimensions while disagreeing on the meaning of the values. **[E04]**

Exercise: Calculate the FP32 payload of `[1, 32, 64, 64]`, then halve both spatial dimensions. The answers are 524,288 bytes and 131,072 bytes. Halving height and width quarters the number of values in this tensor. It does not prove that the whole model's peak memory falls by the same factor.

Exit check: For an input and an intermediate tensor, you can state the shape, axis meanings, dtype, device, and logical payload without calling that payload total process memory.

0.4 Read a network one operation at a time

Goal: Translate a model listing into a sequence of numerical interfaces.

For a batched input, `nn.Linear(16, 32)` produces 32 features from 16 input features. Its weight has shape `[32, 16]`, and its bias has shape `[32]`. PyTorch computes the corresponding affine mapping as `x @ weight.T + bias`. The layer therefore has `16 * 32 + 32 = 544` parameters. **[R63]**

TinyNet applies this sequence:

EXAMPLE

```
features [B, 16]
  -> stem Linear(16, 32)
  -> [B, 32]
  -> residual block, repeated three times
       Linear(32, 64) -> GELU -> Linear(64, 32)
       add the block input
  -> head Linear(32, 3)
  -> logits [B, 3]
```

Each residual branch expands and contracts the feature width without changing its external shape. One branch has `32*64 + 64 + 64*32 + 32 = 4,192` parameters. Three branches plus the stem and the 99-parameter head total 13,219 parameters. These counts are verified by the supplied tests. **[E04]**

GELU is a nonlinear activation function, not a trainable projection. Its standard definition is `x * Phi(x)`, where `Phi` is the standard normal cumulative distribution function. PyTorch also provides a configurable approximation. In this block, GELU changes values but not the tensor shape and adds no learned parameters. Replacing it with another function changes the computation even when all dimensions still match. **[R64]**

Convolution adds another kind of contract. A `Conv2d(3, 8, kernel_size=3, padding=1)` with its default stride and dilation maps `[B, 3, H, W]` to `[B, 8, H, W]`. Its filters mix local spatial neighborhoods and input channels. The tested example has 224 parameters. Convolution output dimensions depend on kernel, stride, padding, dilation, and grouping. **[R47]**, **[E04]**

A normalization layer is not just a harmless rescale. LayerNorm computes statistics over its configured final dimensions. Shrinking those dimensions or replacing whole-image statistics with tile-local statistics can change the result. Learn which axes participate before applying a memory optimization. **[R70]**

Attention is a separate operation for mixing information across positions. In the usual scaled dot-product form, queries and keys determine weights used to combine values. Changing the number of heads, the projection widths, or the positions each token can attend to are different transformations. Do not begin by deleting heads from an unfamiliar implementation. Chapter 7 develops the dependency checks. **[R27]**

Exercise: Use `parameter_count(model)` and `trace_shapes(model, torch.zeros(1, 16))`. Match every reported shape to the sequence above. The trace records metadata only; it does not store tensors for later inspection. **[E04]**

Exit check: You can explain the difference between a learned linear layer, a nonlinear activation function, a residual addition, and a tensor produced by any of them.

0.5 Understand one training step

Goal: Distinguish running a model from changing its learned parameters.

Inference computes an output using the current parameters. Training computes an objective, differentiates it, and applies an update. For this classifier, the head emits logits, which are unnormalized scores for the three classes. `CrossEntropyLoss` accepts those scores directly with the target class indices. Do not insert a softmax before this loss in the class-index example. **[R65]**

A loss function is the numerical objective you choose to minimize. Its value is not automatically the product metric. A lower classification loss does not measure the speed of your Android app, and a lower reconstruction loss does not establish that a video keeps the intended character identity.

Autograd records differentiable operations and computes gradients when `backward()` is called. A parameter's gradient describes the local derivative of the objective with respect to that parameter. It is not a reliable standalone ranking of how safely an entire layer can be deleted. **[R60]**

This is the core order used by the supplied training routine:

PYTHON

```
optimizer.zero_grad(set_to_none=True)
logits = model(features)
loss = torch.nn.functional.cross_entropy(logits, labels)
loss.backward()
optimizer.step()
```

`zero_grad` clears gradients from the previous update. `backward` computes new gradients. `step` applies the optimizer's update. The executed test checks that a weight has not changed after `backward` alone and does change after `step`. Learning rate controls the update scale; it is a training configuration value rather than a model weight. **[R61]**, **[E04]**

There are two controls you must keep separate:

PYTHON

```
model.eval()
with torch.inference_mode():
prediction = model(features)
```

`eval()` changes module training/evaluation behavior where that distinction exists, such as dropout or batch normalization. It does not disable autograd. `inference_mode()` changes gradient-tracking behavior and related overhead. Conversely, `no_grad()` does not call `eval()` for you. The course tests the distinction directly. During distillation, the teacher forward pass uses `no_grad()` so its outputs can safely be used as fixed training targets. **[R73]**, **[E04]**

The basic mathematics needed here is limited: vector and matrix dimensions, weighted sums, functions, the meaning of a derivative, and averages over examples. You can learn deeper optimization theory while doing experiments rather than postponing every experiment until you finish a mathematics curriculum.

Exercise: Set a breakpoint before `backward`, after `backward`, and after `step`. Inspect one parameter and its `.grad`. Explain why gradient accumulation occurs when gradients are not cleared. **[R61]**

Exit check: You can identify the forward pass, objective, gradient computation, parameter update, and evaluation mode as five separate concerns.

0.6 Separate learning from evaluation

Goal: Avoid selecting a smaller model using evidence that it has already been trained or repeatedly tuned to satisfy.

Use three roles for data. Training examples drive parameter updates. Validation examples guide choices such as which block to remove or which recovery recipe to retain. A held-out test set measures the final selected procedure. When you repeatedly use test results to make choices, that set is no longer an untouched test. Preprocessing learned from data must also be fitted without leaking evaluation information. **[R37]**

The course generates 2,048 training examples, 512 validation examples, and 512 final test examples using separate random seeds. Their labels come from an explicitly defined nonlinear rule, not from the teacher. The code selects the removed block and the recovery candidate using validation cross-entropy. Only after selection does it evaluate the teacher and selected candidate on the final test data. **[E04]**

That design demonstrates data roles, not real-world representativeness. For a production application, the unit of separation matters. Two adjacent frames from one video should not automatically be treated as independent examples across training and evaluation. Propose splits by video, scene, speaker, recording session, subject, or source according to what could leak in your task. Record why that unit was chosen.

Your evaluation also needs to match the actual function. A classifier can use accuracy and class-specific errors. A separator needs appropriate reference stems or a separately designed listening evaluation. Image and video generation require controlled prompts, seeds, output settings, and task-specific review. Do not replace all these questions with one generic metric or use a small set of random prompts as a universal quality certificate. See Chapter 3.

Exercise: Before inspecting any new candidate results, write this rule in your experiment note: select the CE or KD student with lower validation cross-entropy, with CE winning ties. Then confirm that `run.py` follows that rule. It does not select whichever method produces the more impressive story.

Exit check: You can name every dataset's role and explain which decisions it influenced. You have a final evaluation set or explicitly label the work as exploratory when no untouched test exists.

0.7 Profile before changing the model

Goal: Identify the quantity that actually needs to improve.

Model file size, parameter payload, an intermediate tensor's size, allocator reservations, and process memory are different measurements. A file can get smaller while runtime memory stays high. A model can use fewer arithmetic operations while an unsupported operator forces an expensive backend transition. Chapter 2 provides the full accounting framework. **[R16]**, **[R21]**

Begin with four records: the architecture and checkpoint identity, quality on the fixed evaluation cases, warmed inference timing for a defined input, and a memory measurement appropriate to your platform. Record initialization time separately. Record accelerator synchronization where applicable. Profiling tools can perturb execution, so perform the final timing run without expensive tracing attached. **[R09]**

The course exposes a deliberately modest inspection function:

PYTHON

```python
from crash_course.run import trace_shapes
rows = trace_shapes(model, torch.zeros(1, 16))
for row in rows:
print(row["module"], row["shape"], row["logical_output_bytes"])
```

The hooks retain names, shapes, and scalar byte counts, not output tensors. They are removed in a `finally` block. The sum of the rows is not peak memory: outputs can share storage or have nonoverlapping lifetimes, and the trace does not include every internal allocation. The recorded run does not claim a peak-RAM result. **[E04]**

The timing exercise uses one CPU thread, batch size one, 30 warmup calls per model, and 200 measured calls per model. It alternates model order and saves every sample. These measurements describe this small implementation in the recorded environment. They are not a forecast for a phone. **[E04]**

Exercise: Choose one objective for the next change: lower measured latency, lower measured peak memory, or smaller parameter storage. State what evidence would support improvement and what would falsify your hypothesis. Do not write only "make it lighter."

Exit check: You can distinguish a calculated quantity from an observed one and can name the baseline against which the next measurement will be compared.

0.8 Make your first structural edit

Goal: Change the architecture deliberately and verify the edit independently of its task quality.

TinyNet's residual blocks preserve their input shape. That makes bypassing one block executable without adding a shape adapter. It does not make bypassing the block behavior-preserving. The course tests each block removal separately on validation data and selects the candidate with the lowest validation loss. Those single-removal results do not prove that removing several blocks together is safe. **[E04]**

PYTHON

```python
from crash_course.models import drop_block, shrink_inner
# Valid for TinyNet. Not a recipe for deleting an HDemucs encoder stage.
candidate = drop_block(teacher, index=chosen_index)
candidate = shrink_inner(candidate, inner=32)
```

The second transformation keeps the residual width at 32 while shrinking the internal branch width from 64 to 32. It removes matching rows from the first projection, matching entries from its bias, and matching columns from the second projection. It preserves the second bias. Simply setting the unwanted weights to zero would leave the original dense tensor dimensions in place. **[E04]**, **[R01]**, **[R02]**

The supplied pruning routine includes checks for empty, duplicate, negative, and out-of-range indices. Its correctness test compares the smaller branch with an original-sized branch whose removed intermediate units are masked to zero. That comparison should match within tolerance. Comparing the smaller branch with the unmodified original should not generally match. These are different tests with different purposes. **[E04]**

The course ranks retained inner units by the absolute weight sum of the first projection. This is an intentionally simple heuristic. It is not a claim that these are the most important units for another model or even the best units for this task.

After replacing parameter objects, build a new optimizer for the candidate. Otherwise an existing optimizer may still reference the original objects. Also update configuration metadata so saving and reloading reconstructs the smaller architecture. The course tests both the numerical transformation and the save/load round trip. **[E04]**

Exercise: Inspect parameter counts after block removal and after inner-width reduction. The recorded sizes are 13,219, 9,027, and 4,867 respectively. Then inspect quality separately. A successful shape test cannot establish preserved classification performance. **[E04]**

Exit check: You have an untouched baseline, a candidate with physically smaller tensors, an explicit architecture configuration, and tests that would fail if your connected tensor slices were wrong.

0.9 Recover the model and test distillation

Goal: Compare ordinary fine-tuning with teacher-guided recovery instead of assuming one must win.

Fine-tuning continues parameter updates from an existing checkpoint. Knowledge distillation adds information from a teacher to the student's training objective. The teacher and student need not have identical internal layers, but the selected target must be meaningfully comparable. For this classifier, both models predict the same three classes. **[R04]**

The lab starts two students from identical copied candidate weights. Each sees the same training examples in the same minibatch order and receives 200 optimizer steps. One uses only cross-entropy against the known labels. The other combines that loss with a soft-target objective from a frozen teacher. Online distillation also performs teacher inference, so equal student update counts do not mean equal total compute. **[E04]**

The soft-target calculation is:

PYTHON

```python
with torch.no_grad():
teacher_logits = teacher(features)
soft_loss = torch.nn.functional.kl_div(
torch.nn.functional.log_softmax(student_logits / temperature, dim=-1),
torch.nn.functional.softmax(teacher_logits / temperature, dim=-1),
reduction="batchmean",
) * temperature ** 2
```

Temperature controls the softened class distributions. PyTorch's KL loss expects log-probabilities for its first argument under the default target convention. The temperature-squared factor follows the standard soft-target distillation treatment. This classifier objective is not a drop-in loss for matching video frames, audio waveforms, or diffusion noise predictions. **[R05]**, **[R66]**

The recorded validation results are deliberately shown without choosing a favorable narrative:

Candidate	Parameters	Validation accuracy	Validation cross-entropy
Trained teacher	13,219	94.73%	0.1895
Block removed and inner width pruned, before recovery	4,867	91.80%	0.2682
Same candidate after supervised fine-tuning	4,867	94.73%	0.1503
Same candidate after distillation	4,867	94.53%	0.1682

These are rounded values from one synthetic CPU run. The prespecified selection rule retained the supervised student because it had lower validation cross-entropy. On the final 512-example test set, the teacher achieved 93.16% accuracy and the selected student 94.53%. This does not establish that pruning improves generalization or that distillation is inferior in general. The experiment is too small and narrow to support those claims. **[E04]**

The hardware budget for larger recovery runs must include more than the student's checkpoint. It can include resident teacher and student weights, student gradients, optimizer state, live activations, workspace, and inputs. Freezing the teacher removes its parameter-gradient updates but not the cost of its forward pass. Offline teacher targets can separate teacher inference from student training, but require storage and a precisely recorded target-generation contract. [R73; proposed planning procedure]

The course's CPU execution is evidence that **this lab** needs no GPU. It is not evidence that a video-diffusion teacher and student fit on the same hardware. Before buying hardware for a real model, run or obtain a representative memory profile for the exact resolution, sequence length, batch size, precision, and training method.

Exercise: Read the loss curves, ablation results, and final selection in `report.json`. Describe what the run establishes and at least two questions it does not answer.

Exit check: You can implement teacher-free and teacher-guided recovery, freeze the teacher correctly, and report an unfavorable distillation result without discarding it.

0.10 Cross the export and quantization boundary

Goal: Treat a deployed graph as another implementation that needs its own tests.

An ONNX file describes a model graph and its data. ONNX Runtime is an execution engine for compatible graphs. An Execution Provider supplies backend implementations. Exporting successfully does not establish that the target backend supports every operation or that it executes them efficiently. **[R49]**, **[R17]**

Before quantization, export a full-precision candidate and compare its outputs against PyTorch. Otherwise a quantization regression and an export regression arrive at the same time. The optional `crash_course/export_onnx.py` exports a fixed `[1, 16]` input contract, validates the graph, and compares 32 generated inputs using CPU ONNX Runtime. It writes a parity record only after those checks pass.

BASH

```
python -m crash_course.export_onnx \
 evidence/crash_course/selected.pt \
 runs/export/selected.onnx
```

Status: This script is provided for execution in your environment. The preparation environment did not contain the optional ONNX dependencies, so no exported model or parity result is claimed or included. Install compatible `onnx`, `onnxscript`, `onnxruntime`, and `numpy` packages, record their versions, and inspect their supported export path before use. `numpy` is required even though the core course does not need it, because the parity comparison converts a tensor with `.numpy()`. The module checks for the three ONNX packages and reports them; a missing `numpy` instead surfaces later as an `ImportError`. On Windows, set `PYTHONUTF8=1` for this command: PyTorch's exporter writes progress characters that a console using the legacy cp1252 encoding cannot represent, which raises `UnicodeEncodeError` from inside `torch.onnx` after the export itself has already succeeded. [R13; proposed exercise]

Quantization changes numerical representation. Weight-only compression and quantized activations are different interventions. A four-bit weight file does not establish that the target device performs four-bit arithmetic. Support depends on graph operations, quantization form, runtime version, and backend kernels. Static quantization also requires representative calibration data. **[R14]**

For the first deployment exercise, keep the float model as a reference and add quantization as a second candidate. Compare numerical drift, task quality, model storage, runtime memory, and latency. Do not require exact floating-point equality after an intentional approximation. Set tolerances and task acceptance criteria before interpreting the result.

Exercise: Write a deployment manifest with input/output names, shapes, dtypes, feature meanings, preprocessing, checkpoint digest, export settings, and runtime version. Explain why dimensions alone would not catch the layout bug from Lesson 0.3.

Exit check: You have separate records for PyTorch quality, exported-model parity, and quantized-model quality. An export script is not being presented as an Android benchmark.

0.11 Integrate and measure on Android

Goal: Reuse your Android engineering skills without letting the app hide a model-contract failure.

Start with a numeric fixture rather than camera or microphone input. The lab writes `golden.json`, containing one `[1, 16]` feature vector and its expected three logits. This isolates tensor construction, model loading, and output interpretation from sensor preprocessing. The features are synthetic numbers, not image pixels. **[E04]**

The optional `android_example/TinyNetRunner.kt` shows the ONNX Runtime Java interface from Kotlin. Its responsibility is narrow: load a verified model file, construct the input tensor, run the named output, copy the scores, and release per-call resources. It is a source example, not a complete Android Studio application or a compiled device artifact. ONNX Runtime documents Java session creation, tensors, inference, and resource cleanup. **[R72]**

Use this integration contract:

EXAMPLE

```
trusted exported model + golden input fixture
  -> Kotlin tensor construction
  -> ONNX Runtime CPU execution
  -> three raw output scores
  -> compare with expected output using stated tolerance
```

After this passes, integrate actual preprocessing for a real model. Keep inference off the UI thread, define who owns the session, close resources, handle invalid inputs, and prevent shutdown from racing with an active call. The included wrapper serializes access to its session; your application must still provide an appropriate lifecycle and worker context.

Only then investigate accelerators. Choose the runtime and provider using supported operations and measured behavior on the actual devices. NNAPI is deprecated starting with Android 15, so it should not be treated as the default future deployment strategy simply because older examples use it. The official Android guidance and runtime support for your selected version should guide backend selection. **[R71]**, **[R16]**

Measure initialization separately from repeated inference. Add sustained runs, memory pressure, foreground/background transitions, and thermal behavior to device testing. Android process-memory tools report different quantities from a tensor payload calculation; choose and label the measure used in your acceptance gate. **[R21]**, **[R22]**, **[R23]**

Exercise: Follow `android_example/README.md` after executing export locally. Compare the golden fixture before adding UI features. Fill the device-measurement template even when the outcome is that the smaller model is not faster.

Exit check: You can say whether a failure belongs to preprocessing, graph conversion, runtime support, application lifecycle, or model quality, rather than treating every failure as "the AI model is broken."

0.12 Choose the next model and graduate

Goal: Carry the method to another family without copying an inappropriate transformation.

Your next experiment should change one known bottleneck in a model you can already run and evaluate. The following routes are proposals, not executed production recipes.

Next family	First bounded experiment	New knowledge to acquire
Image classifier	Reproduce baseline evaluation and physically prune a supported channel group	Image preprocessing, label mapping, convolution dependencies
Image diffusion	Profile text encoder, denoiser, scheduler calls, and decoder separately before changing one component	Latent representations, timesteps, conditioning, denoising objectives
Video diffusion	Fix prompt, frame count, resolution, and sampling settings; inspect the actual temporal-memory bottleneck	Temporal context, frame consistency, motion evaluation
Audio separation	Reproduce the checkpoint's own sample-rate and stem contract; test supported segmentation settings first	STFT, complex values, overlap, waveform alignment, reference stems
Language model	Verify tokenizer and decoding settings, then measure a controlled context-length workload	Tokens, attention, KV state, task-specific evaluation

For diffusion, reducing denoising steps and removing neural layers are different changes. A scheduler determines how iterative updates are applied; the denoiser predicts a quantity at a particular noise level. A classifier's three-class KL loss is not automatically a diffusion-training objective. Read Chapter 12 before adapting the recovery loop. **[R31]**, **[R38]**, **[R42]**

For HDemucs, an encoder stage can change frequency or time resolution and participate in skip connections and the relationship between waveform and spectrogram paths. It is not interchangeable with the shape-preserving TinyNet branch. Use Appendix D to identify the architecture and its inference wrapper before deciding what to change. **[R51]**, **[R55]**

Build a short graduation record with your chosen checkpoint, the exact input contract, baseline measurements, one change, correctness tests, quality evaluation, and a keep-or-revert decision. A good decision can be to reject the modification because it damages quality or does not improve the measured bottleneck.

Readiness test: Explain, without relying on this text, why smaller weights may not reduce peak activations, why a passing shape test is insufficient, why the teacher can be frozen but still consume compute, and why desktop timing does not establish phone performance. Then run the course from a clean environment or explain any environment difference you introduced.

Where to continue: Chapters 1 through 3 formalize the baseline and evaluation process. Chapters 4 through 11 develop the transformation catalog. Chapters 12 through 14 address diffusion and deployment. Chapters 15 through 17 explain recorded experiments and acceptance. The glossary remains available through linked terminology throughout the book.

1. Establish a reproducible baseline

Problem

You have a trained model that appears too large or slow. The temptation is to delete layers immediately. Without a reproducible baseline, you cannot distinguish a useful optimization from a different input, a changed preprocessing step, a lucky timing result, or an export error.

Your first deliverable is not a smaller model. It is an experiment that another engineer can run and interpret.

Complete intake before planning any edit

Most of this handbook assumes you can run the model, retrain it, and measure its task quality. Establish that you can before committing a sprint to it. Record what you actually hold:

1. The training-time architecture source at a pinned revision, not only a serialized graph.
2. The float checkpoint and its digest, with the publisher or owner named.
3. The exact preprocessing and postprocessing code, including tokenizer or feature settings.
4. An evaluation set with references, and the right to use it for this purpose.
5. Inputs usable for calibration data and for recovery training.
6. Whether retraining is permitted at all, and on what hardware.

Items 1, 5, and 6 are preconditions, not conveniences. Every structural edit in Chapters 4 through 10 constructs new modules from the architecture definition and then recovers the model by training it. Without the source you cannot perform the edit; without data and permission you cannot recover from it. If any of the three is missing, the correct next action is to request it, naming what you will do with it, rather than to begin.

If all you have is the exported artifact. Being handed only a deployable file is the common case for an application engineer, and it is not a dead end. It does rule out Chapters 4 through 10: you cannot prune a graph you cannot rebuild, and a configuration field is not an architecture change. What remains is a shorter list, already covered later in this handbook: weight-only and dynamic quantization of the existing graph and the granularity choices that go with them in Chapter 11; graph optimization level, execution provider selection, the partitioning that decides what actually runs on an accelerator, and shape bucketing in Chapter 13; and thread configuration and application-level scheduling in Chapter 14. Those levers are frequently enough to meet a latency budget, and they are the ones to exhaust before asking for retraining rights.

State which case you are in at the top of the experiment record. A reader of your report cannot otherwise tell whether an untried transformation was rejected or was never available.

Define the product contract

Describe the task in observable terms. A detector may need to identify small objects. An image generator may need to preserve requested composition. A video generator may need to maintain identity and motion across a specified duration. A speech model may need intelligibility and speaker consistency. These are proposed product requirements, not properties established by a parameter count.

Define the operating envelope separately: supported devices, maximum input duration, supported resolutions, sequence lengths, channels, and sample rates, whether network access is allowed, and whether the job may pause. A batch application and a live translator have different deadlines even when they use the same model.

Avoid a requirement such as "the model must fit in 900 MB." Specify what the number measures: model download, mapped parameter data, peak process footprint, or one activation buffer. Android's managed heap limit varies by device and is not a universal allowance for native ML tensors. **[R21]**

Freeze the full implementation

A model is more than its checkpoint. Record the architecture source, configuration, tokenizer or phonemizer, preprocessing, checkpoint digest, inference arguments, dependency versions, and decoding settings. Two applications can load identical weights and produce different output because they interpret the input differently.

Open the checkpoint before trusting it

Loading a file you did not create is often your first direct encounter with a real model, and failures can arise before any modeling decision. Each has an easy-looking fix that is wrong.

Load with `weights_only=True`, which is the default on current PyTorch:

PYTHON

```python
payload = torch.load(path, map_location="cpu", weights_only=True)
print(type(payload), list(payload)[:10] if isinstance(payload, dict) else "")
```

A failure here is information, not an obstacle. It means the file contains code-bearing objects, such as a training namespace, a configuration object, or a whole pickled module, and it may raise an unpickling error without identifying that context. Setting `weights_only=False` executes whatever the file contains. That is a decision about trusting its origin: verify the digest against the publisher first, and never apply it to an artifact you merely downloaded. The safer route is to ask the owner for a plain state dictionary.

A training checkpoint is usually not a state dictionary but a dictionary containing one, alongside optimizer state, a step count, and often an exponential moving average of the weights. Print the top-level keys and choose deliberately. When an `ema` or `ema_state_dict` entry exists, it frequently contains the weights used to produce the published results; taking the raw weights instead can yield a quietly worse baseline that invalidates every later comparison.

Then check the key names. A checkpoint saved from a distributed wrapper carries a `module.` prefix on every key, and one saved from a compiled model carries `_orig_mod.`. Against your architecture those keys match nothing:

PYTHON

```python
state = {k.removeprefix("module.").removeprefix("_orig_mod."): v
for k, v in state.items()}
report = model.load_state_dict(state, strict=True)
assert not report.missing_keys and not report.unexpected_keys, report
```

The rule is that a key mismatch is fixed by renaming keys, never by relaxing `strict`. This matters because the relaxation fails silently in the worst possible way. With `module.` prefixed keys and `strict=False`, `load_state_dict` reports every parameter as missing and every supplied key as unexpected, loads **nothing**, and returns a network still at its random initialization. Verified: the weights are bit-identical to the untrained model afterwards. Nothing raises. You then measure a baseline so poor that every subsequent change looks like an improvement, and the entire experiment is inverted. `strict=True` instead raises immediately and names the mismatched keys, which is why this handbook requires it everywhere.

Assert on the returned report even under `strict=True`, and record the resolved checkpoint key, any prefix you stripped, and whether you took averaged or raw weights in the frozen implementation record above.

Use a manifest that your program can populate rather than a handwritten description that drifts from the implementation:

JSON

```json
{
 "architecture": "RefactorNet",
 "architecture_version": "1.0",
 "checkpoint_sha256": "computed_by_the_run",
 "input_contract": {"shape": [1, 16], "dtype": "float32"},
 "runtime": "PyTorch 2.10.0+cpu",
 "device_class": "Linux x86_64 CPU",
 "threads": 1,
 "seed": 20260924
}
```

This is a schema example. The companion run writes the actual digest and configuration into `evidence/run/report.json`; it does not retain the illustrative digest above. **[EO1]**

Establish two baselines

The **quality baseline** defines how the unmodified checkpoint behaves on the evaluation suite. The **performance baseline** defines how the deployed implementation behaves with a fixed workload and device configuration. Keep both, because a valid PyTorch model is not automatically an equivalent exported model.

Measure initialization separately from warm inference. Report the input shape, batch size, number of warmup calls, number of measured calls, thread configuration, and whether accelerator work was synchronized. For an audio stage, define real-time factor as processing seconds divided by input-audio seconds. For a multi-stage application, measure the complete job as well as individual stages.

The procedure matters as much as the reported fields. Time with a monotonic high-resolution clock, not wall-clock time of day. Discard warmup calls, because the first calls pay allocation and lazy initialization. Take many samples per measurement and report a median rather than a mean, which one outlier distorts. Set the thread count explicitly instead of inheriting it. Measure the models in a shuffled order across several rounds, so that drift in machine state does not attach itself to whichever model happened to run last. Report the round-to-round spread; it is your noise floor, and a difference smaller than it is not a result.

PYTHON

```
# Executed implementation is in lab.py.
for _ in range(30): # warmup, discarded
model(x)
samples = []
for _ in range(200):
start = time.perf_counter_ns() # monotonic, nanosecond resolution
model(x)
samples.append((time.perf_counter_ns() - start) / 1e6)
round_median = statistics.median(samples)
```

The companion `benchmark` function repeats that inner loop for five rounds per model, shuffles model order between rounds, and retains every raw sample along with the per-round medians, so the noise floor can be computed afterwards rather than assumed. **[E01]**

Do not replace measured latency with multiply-accumulate counts. NetAdapt specifically studies adaptation using direct platform measurements because indirect quantities such as operation count do not reliably determine latency and energy. Its mobile vision experiments are evidence for a hardware-aware procedure, not a performance forecast for speech models. **[R08]**

Define acceptance before selecting candidates

Write down which quality regressions are unacceptable, which performance objective matters, and which inputs must continue to work. Your thresholds are product decisions. There is no universal acceptable percentage loss for detection, semantic correctness, image fidelity, motion consistency, or voice identity.

For example, a release gate can require that a critical object category retain recall, that an image-editing mask remain respected, and that a chosen memory measurement fall. State how each condition will be measured and who reviews ambiguous outputs. Do not adopt numerical thresholds from an unrelated application.

Verification: rerun the unchanged baseline before beginning surgery. Confirm that the quality result is stable enough to detect the changes you care about. Treat inconsistent runs as an experiment problem first.

Decision and rollback: preserve the original checkpoint, environment manifest, reference outputs, and preprocessing. A candidate is not the new baseline until it passes the agreed evaluation.

2. Find the actual bottleneck

Problem

A model file contains the parameters needed to run the network, but the process also needs intermediate values, persistent state, runtime workspaces, and application buffers. A parameter reduction can leave the real memory bottleneck untouched.

Separate the quantities

For a dense tensor, its logical payload is the product of its dimensions multiplied by bytes per element. This arithmetic excludes allocator padding, views that share storage, compressed representations, and workspace. For example:

EXAMPLE

```
Shape: [1, 2000, 1024]
FP16 element size: 2 bytes
Payload: 1 * 2000 * 1024 * 2 = 4,096,000 bytes
Decimal MB: 4.096
Binary MiB: 3.90625
```

The distinction between decimal MB and binary MiB matters when comparing logs. State the unit and conversion rather than silently mixing them.

A useful conceptual accounting model is:

EXAMPLE

```
At time t:
working memory = resident parameters + live intermediate tensors
               + persistent state + temporary workspace
               + runtime and application allocations

Peak memory = maximum of that quantity over the execution
```

This is an accounting model, not a formula for adding RSS to parameter bytes. RSS already includes resident allocations. Some buffers share pages or storage; count those relationships explicitly when you need an accurate total. Android's PSS accounts proportionally for shared memory and is a different measurement from a tensor's logical payload. **[R21]**, **[R22]**

Tensor lifetimes matter

Suppose a layer reads a 300 MB tensor and writes a separate 250 MB tensor. Those buffers can overlap in lifetime. A following layer reads the 250 MB tensor and writes 350 MB. With no in-place operation, no additional state, and no scratch space, the two steps need approximately 550 MB and 600 MB respectively.

The required storage is not automatically 900 MB, because earlier buffers may be reused. It is also not automatically 350 MB, because input and output often coexist. This is a worked lifetime example, not a measurement of a particular runtime.

Deleting some sequential layers may reduce parameters and execution time while barely changing the largest live allocation. Removing a long-lived skip connection or shortening the sequence may affect the peak more directly. Confirm this against the actual execution schedule, rather than adding per-layer output sizes.

Profile the model without turning the profiler into the bottleneck

PyTorch's profiler can record operator time, input shapes, and tensor allocation activity. Its documentation warns that collecting shapes and stacks adds overhead and can retain references. Use profiling to locate expensive work, then benchmark without the profiler. **[R09]**

PYTHON

```python
# Integration pattern. Run against your own model and inputs.
model.eval()
with torch.inference_mode():
with torch.profiler.profile(
activities=[torch.profiler.ProfilerActivity.CPU],
record_shapes=True,
profile_memory=True,
) as prof:
model(example_input)
print(prof.key_averages().table(
sort_by="self_cpu_time_total", row_limit=15
))
```

The lab includes a metadata-only forward-hook pass that writes module output shapes. Those records locate large outputs; they are not a peak-activation measurement. Hooks on modules can miss functional operations, and two reported outputs may share underlying storage. The lab deliberately does not publish a fabricated peak from their sum. **[E01]**

Distinguish training from inference

Use `model.eval()` to select evaluation behavior where modules have separate training behavior. Disable gradient recording for inference as appropriate. PyTorch's `inference_mode` removes additional autograd overhead but does not itself call `eval()`. Do not apply it to gradient-based importance estimation or to tools that need autograd to analyze the graph. **[R12]**

Activation checkpointing is principally a training tradeoff: it saves selected activations and recomputes others for backward. It is not an automatic remedy for a forward-only mobile memory problem. **[R36]**

Choose the next experiment

If parameters dominate, investigate quantization, width reduction, low-rank factors, or fewer blocks. If long feature sequences dominate, investigate chunking, state management, or temporal resolution. If runtime copies dominate, inspect layout changes and execution-provider boundaries. If application buffers dominate, fix the surrounding code before modifying the model.

Verification: identify the exact operation or lifetime responsible for the chosen bottleneck.
Decision: proceed only when a proposed change can plausibly affect that bottleneck.
Rollback: preserve the trace and input that exposed it so the same condition can be reproduced.

3. Build an evaluation suite that can reject a bad change

Problem

A model can perform well on a demonstration and still fail on a small object, a long prompt, an unusual camera movement, a quiet speaker, or a rare output class. Your suite must expose the failures relevant to the experiment rather than reward the examples you happened to inspect.

Use data for distinct purposes

The **training set** updates parameters. A **calibration subset** estimates quantization ranges or importance statistics. The **validation set** guides choices such as which block to remove. The **test set** evaluates the selected procedure after those choices are fixed.

Calibration data can come from training data. It does not have to be held-out test data. Any procedure that fits statistics must not quietly fit them on the final test set. This separation is also emphasized in scikit-learn's guidance on data leakage. **[R37]**

A frozen validation set makes development comparisons consistent. It does not remain unbiased after repeated tuning. Reserve a separate final test, record its use, and do not revise the candidate after inspecting it while still calling it unseen.

Where the examples come from

Start with the original model's documented evaluation data when it is accessible and suitable. Add your own legally usable, representative inputs and reviewed expected outcomes. You do not need to reproduce the original training corpus to begin a regression experiment. You do need enough coverage to justify the claim you plan to make.

A classifier needs inputs and class labels. A segmentation model needs masks or another appropriate reference. A generator may use a fixed prompt suite, condition images, masks, random seeds, and human review rather than one uniquely correct output image. A teacher's generated outputs can be recovery targets, but agreement with the teacher does not independently establish quality.

Do not choose a universal sample count. Begin with a small diagnostic suite to catch implementation errors, then expand the held-out evaluation to cover the operating envelope and obtain useful uncertainty estimates. Ten attractive outputs cannot establish a broad quality claim.

Split at the unit that can leak

Do not scatter adjacent frames of the same video across training and testing. Decide whether the claim concerns new clips, new scenes, new subjects, or new source collections, then group the split accordingly. Apply the same reasoning to near-duplicate images, lines from the same speaker, and documents from the same source. This is a proposed dataset-design procedure, not a prescribed split ratio.

For prompt-based generation, separate prompt templates as well as individual strings when template generalization matters. A different noun inserted into a repeatedly tuned prompt is not necessarily a new kind of test.

Choose metrics that describe the task

Model family	Useful evaluation dimensions	Failure an average may conceal
Classification or detection	Correctness, calibration, per-class recall, small-object performance	A rare but important class is lost.
Segmentation or restoration	Boundary quality, structural fidelity, reference agreement	Thin structures disappear or edges shift.
Language	Task success, factual consistency where relevant, long-context behavior	The model passes short prompts but fails structured output.
Speech or audio	Intelligibility, identity, temporal alignment, audible artifacts	Words remain clear while a speaker's identity changes.
Image generation	Prompt following, composition, diversity, visual defects	A model produces pleasing images while ignoring requested relations.
Video generation	Identity, motion, temporal flicker, spatial quality, prompt following	Individual frames look good but the sequence is unstable.

The table is a proposed engineering checklist, not a definition of any named benchmark. VBench is a primary research example of evaluating video generation along distinct dimensions instead of collapsing all behavior into one visual score. ITU-T P.808 is a source for speech listening-study methodology, not a universal speaker-similarity measure. **[R43]**, **[R33]**

Make generative comparisons reproducible

Use the same prompt and conditioning case across baseline and candidate. For closely related generators, reset the random generator for each case and retain the initial latent tensor when the implementation permits. Reusing a mutable generator object without resetting it advances its state, so the same nominal seed recorded once is insufficient. Cross-device and cross-version equality is not guaranteed. **[R44]**, **[R10]**

When changing resolution, the initial noise tensor changes shape. The same integer seed no longer creates a directly aligned pixel-level experiment. Compare task outcomes across a fixed seed list rather than interpreting pixel differences as pure model error.

Record the evaluator's preprocessing. Image resizing and compression can change FID measurements, as demonstrated by the Clean-FID work. Report the metric implementation, feature extractor, reference set, sample count, and preprocessing. Do not attach a published FID label to an informal handful of pictures. **[R45]**

Make evaluation actionable

A record should connect an input to its conditions, candidate, and failure categories:

JSON

```
{
 "case_id": "heldout_video_014",
 "condition_groups": ["camera_pan", "occlusion", "two_subjects"],
 "reference_version": "reviewed_cases_v1",
 "candidate_digest": "computed_by_the_evaluator",
 "checks": ["identity", "motion", "prompt_following", "artifacts"]
}
```

This is a schema example, not an executed video result. Store failed outputs alongside aggregates. An accuracy change from 90% to 89% is one percentage point. State units rather than reporting an ambiguous "1% loss."

Interpret uncertainty

Use paired comparisons where possible. Repeat recovery seeds when a decision depends on a small difference. Estimate uncertainty at the independent sampling unit, such as a video or source group, rather than treating correlated frames as independent evidence.

Put a number on the noise before interpreting a difference. For a rate metric such as accuracy or recall measured on `n` independent cases, the standard error is `sqrt(p*(1-p)/n)`. At `p = 0.87` and `n = 1024` that is about 1.05 percentage points, so a one-sigma band is roughly plus or minus one point and a two-sigma band roughly plus or minus two. A two-point spread on a suite of that size is therefore indistinguishable from noise. Report `n` next to every subgroup number, because the same rule makes a three-point move on a subgroup of 40 cases meaningless.

For a paired comparison, the useful quantity is not the two rates but the disagreement: count the cases the baseline gets right and the candidate gets wrong, and the reverse. Those two counts, not the aggregate difference, are what a decision should rest on, and they also name the specific examples to inspect.

For a latency comparison, the noise floor is measured, not derived: take the spread of the per-round medians from the timing procedure in Chapter 1, and treat any candidate-versus-baseline difference smaller than that spread as unresolved. Repeating the measurement narrows the floor; it does not remove it.

For human comparisons, conceal candidate identities, randomize presentation order, and ask a specific question. Fidelity, naturalness, identity, and preference are different outcomes. Keep difficult cases in the suite after discovering them, but track that they have become development examples.

Verification: version the examples, split rules, and scoring code. **Decision:** reject critical subgroup regressions even when the mean improves. **Rollback:** retain failure outputs and the configuration that generated them.

4. Remove a residual block

Problem and preconditions

The model contains repeated blocks whose input and output contracts match. You want to remove one to reduce depth. A valid candidate is a block that can be bypassed without breaking tensor dimensions, required state, or the meaning of the next interface.

A residual block of the form `y = x + F(x)` has a natural bypass: return `x`. An encoder stage that also changes time resolution or channel count does not necessarily have that property. A block that produces a decoder cache or a second output cannot safely be replaced by a one-output identity module without an adapter.

LayerDrop is a reported training method designed to make Transformer depth adjustable. Its results do not establish that arbitrary layers in an ordinary pretrained checkpoint can be removed without recovery. **[R06]**

Rank candidates with ablation

Ablation means changing one candidate while holding the rest of the experiment fixed. For each removable block, build a separate model, run the validation set, and record the difference from the unmodified model. The companion lab uses validation cross-entropy as the selection criterion. It does not select a block using final test accuracy. **[E01]**

PYTHON

```python
# Executed implementation is in lab.py.
student = copy.deepcopy(model)
student.blocks = torch.nn.ModuleList([
block for i, block in enumerate(student.blocks)
if i != remove_index
])
```

The code is valid for RefactorNet because every block preserves a 32-value feature dimension and the forward method simply iterates over `blocks`. It is not a general Transformer-pruning function. A production architecture can also require changes to layer counts, cache indices, relative-position modules, serialization metadata, and generation helpers.

Separate ranking from acceptance

A layer can have the smallest loss increase and still be too important to remove. Ranking tells you which experiment to investigate first. Acceptance tells you whether the result meets the product contract.

Do not assume that ablation losses add. Two individually tolerable removals can damage complementary functions. Remove a small set, recover if appropriate, and rerun importance tests on the modified network.

Low residual magnitude, activation similarity, and gradient-based sensitivity can help shortlist candidates. They remain heuristics unless checked against task outcomes. First-order pruning criteria have research support in specific settings, including Molchanov and colleagues' CNN experiments. That support does not make small gradients a universal test of irrelevance. **[R07]**

What the executed lab found

The lab evaluated all six blocks. Removing block 5 gave the lowest validation loss. Some removals actually improved validation loss. That is possible in a finite, imperfectly trained model and is not proof that the removed layer was universally useless. The selected removal slightly reduced final test accuracy before recovery. The complete ablation table appears in Chapter 15. **[E01]**

This contrast is important: the validation set chooses a candidate; the test set measures whether the selected procedure generalizes. It is normal for the two not to move in exactly the same direction.

Verification and recovery

Run shape and finite-value checks first. Rebuild the optimizer after changing the parameter structure. Save the edited architecture configuration with its weights, then reconstruct it in a clean process and require strict loading. The lab tests exact output equality for that save-and-reload operation, not equality between the original and pruned models. **[E01]**

Recover the candidate with supervised fine-tuning or distillation. Compare recovery methods under an explicit training budget rather than allowing one candidate substantially more optimization without saying so.

Decision: keep the deletion only when quality and the target-device resource objective pass. **Rollback:** restore the original checkpoint and the original block ordering. Do not attempt to reverse a sequence of undocumented in-place edits.

5. Shrink feed-forward channels without changing the external interface

Problem and preconditions

A block contains a feed-forward network that expands from width `d` to width `h`, applies an elementwise activation, and projects back to `d`. You want to reduce `h` while leaving the residual interface unchanged.

This is a useful first structural edit because it is narrower in scope than changing the network's global hidden width. The worked implementation applies to an ordinary dense `Linear -> GELU -> Linear` path with no normalization across the intermediate feature dimension, no gated parallel path, and no tied parameters. **[E01]**

Remove the connected dimensions together

For a PyTorch linear layer, weight rows correspond to output features. If you retain hidden indices `K`, the connected changes are:

EXAMPLE

```
Original:
up.weight   [h, d]     up.bias   [h]
down.weight [d, h]     down.bias [d]

After retaining k hidden channels:
up.weight   [k, d]     up.bias   [k]
down.weight [d, k]     down.bias [d]
```

Copy rows `K` from the first weight and bias. Copy columns `K` from the second weight. Preserve the second bias. The lab's `shrink_ffn` function validates indices, creates smaller modules on the original device and dtype, copies the coupled parameters, carries over the normalization parameters, and restores the training mode. **[E01]**

PYTHON

```python
# Core of the executed transformation; index guards are in lab.py.
new.norm.load_state_dict(old.norm.state_dict()) # not on the pruned axis; still carried over
new.up.weight.copy_(old.up.weight[keep])
new.up.bias.copy_(old.up.bias[keep])
new.down.weight.copy_(old.down.weight[:, keep])
new.down.bias.copy_(old.down.bias)
new.train(old.training) # a fresh module defaults to train mode
```

Do not write directly into `.data` to hide shape inconsistencies. Construct the correct modules and preserve their configuration explicitly.

Account for every tensor in the block, not only the ones on the pruned axis. A freshly constructed module is default-initialized, so any parameter or registered buffer you do not copy is silently replaced with untrained values: `LayerNorm` affine weights become one and biases zero, `BatchNorm` running means reset to zero and running variances to one, and learned scalars or position tables revert to their defaults. Parameter count, tensor shapes, a strict checkpoint load, and export all still succeed when such a tensor is missed, so the masked comparison in the next section is what detects it.

Two properties that are not tensors travel with the same problem. `requires_grad` is not part of a `state_dict`, so a freshly built module is trainable regardless of what the original was: if you had frozen part of the network, pruning silently unfreezes it, and a later recovery step updates parameters you believe are pinned. Training mode is the same. Restore both explicitly.

Finally, mind which no-gradient context the surgery runs in. Tensors created inside `torch.inference_mode()` cannot later participate in an autograd graph, so a module built there looks correct, saves, and evaluates, and then raises at the first backward pass of the recovery step this handbook tells you to run next. Use `torch.no_grad()` for a transformation whose output must remain trainable, and reserve `inference_mode` for deployment paths that never train.

Prove the surgery did what you intended

There are two different comparisons. The structurally pruned FFN should match the original FFN with the removed intermediate channels masked to zero, within numerical tolerance. It does not generally match the unmasked original FFN.

The companion self-test checks the first equivalence. It also checks that the parameter count falls and that duplicate indices are rejected. These are implementation tests. They do not establish acceptable task quality. **[E01]**

PyTorch's standard pruning utilities demonstrate masking. Removing the pruning reparameterization makes that sparsity permanent in the values; it does not by itself produce smaller dense tensor dimensions. Torch-Pruning addresses structural removal and coupled dependencies. **[R01]**, **[R02]**

Choose channels, then test the choice

The lab uses a simple heuristic: average absolute hidden activation multiplied by the norm of the corresponding output-weight column. It computes this on 256 training examples, not on the final test set. This is original illustrative scoring code, not an implementation claimed to reproduce a published optimal pruning criterion. **[E01]**

Other candidate-selection strategies can use validation ablation, learned gates, or gradient-based importance. The important engineering property is that the score, data, and retained indices are recorded. Avoid saying "unimportant channels" when you have only measured low values under one criterion.

For a gated FFN, such as one with separate gate and value projections multiplied elementwise, corresponding hidden channels must be removed from both branches and the downstream projection. For convolutions, channel changes can propagate through batch normalization, residual additions, concatenations, and grouped operators. DepGraph formalizes this dependency problem, but custom modules still require inspection and tests. **[R03]**

Verification, acceptance, and rollback

Benchmark widths that the target kernel handles well, not only the mathematically smallest width. The number of parameters changes predictably from the shapes; latency remains a measurement. The lab’s width reduction removed substantially more parameters than the observed percentage improvement in warm CPU latency. **[E01]**

Retain the original indices in the experiment record, reconstruct the student from its new configuration, run recovery, and evaluate subgroup failures. Revert if task quality is unacceptable, if export fails, or if the changed shape does not improve the resource objective that justified the work.

6. Prune convolution channels without breaking the graph

Problem and preconditions

A convolutional stage produces more feature channels than your resource budget permits. Unlike deleting scattered weight values, reducing channel dimensions can create smaller dense tensors for the next stage. The engineering problem is identifying every consumer of those channels.

The executed `ConvPair` example has a deliberately narrow contract:

EXAMPLE

```
Input image [B, 3, H, W]
  -> Conv2d: 3 channels to 12
  -> ReLU
  -> Conv2d: 12 channels to 3
  -> Output image [B, 3, H, W]
```

It has no residual branch, concatenation, normalization, grouped convolution, or shared parameters. This makes the hidden channel axis easy to inspect. It is a mechanism test with random weights, not a trained image model. **[E03]**

Make one physical edit

Retain a set of hidden channels `K`. Copy the first convolution's output filters and biases at those indices. Copy the corresponding input-channel slices from the second convolution. Keep the second bias unchanged.

PYTHON

```python
# Executed for the exact ConvPair architecture in the companion lab.
new.first.weight.copy_(old.first.weight[keep])
new.first.bias.copy_(old.first.bias[keep])
new.second.weight.copy_(old.second.weight[:, keep])
new.second.bias.copy_(old.second.bias)
```

The full function validates the indices, creates the new modules on the same device and dtype, and preserves evaluation state. Its self-test compares the smaller model with the original after masking the removed hidden activations. Both produce the same result in the recorded test. That does not mean the smaller model equals the unmasked original. **[E03]**

Expand from a chain to a dependency group

A real image model is usually more connected than the example. Before pruning a channel, draw its consumers or inspect them in the captured graph. A residual addition requires compatible shapes on both branches. A concatenation changes the input offsets of a

downstream operator. A normalization layer can own channel-indexed parameters and state. Dependency-aware pruning groups these connected edits rather than treating each module independently. **[R03]**

Consider this proposed inspection:

EXAMPLE

```
A produces channels 0..63
  -> normalization
  -> residual branch
  -> concatenation with B
  -> projection
```

Your edit record should identify the retained channel order, the modified normalization state, the residual adapter if any, and the concatenation offsets seen by the projection. "Pruned 25% of layer A" is not enough to reconstruct the graph.

Torch-Pruning is useful for generating structural dependency groups. Its graph construction uses autograd, so do not wrap that analysis in inference mode. Use the package's documented workflow for the pinned commit, then inspect the proposed group before applying it. A graph-analysis tool does not know your application-specific quality contract. **[R02]**

Grouped operators require a different edit

For grouped convolution, channels are divided into groups rather than fully connected. Input and output channel counts must respect the group configuration. Depthwise convolution is a special grouping pattern, not merely an ordinary convolution with many zero weights. PyTorch documents these constraints for `Conv2d`. **[R47]**

Do not feed arbitrary retained indices into the `ConvPair` function and expect it to support a depthwise layer. Either retain valid groups and update the group metadata, or implement a tested transformation specifically for that operator. Removing only one member of a coupled group can invalidate both the shape and channel mapping.

Normalization deserves the same care. Shrinking the set of values over which a normalizer computes statistics changes the function. The fact that all shapes match does not prove equivalence. The tiling experiment in Chapter 9 makes this issue visible for GroupNorm. **[R28]**, **[E03]**

Replacing convolution is a new model, not a rename

Depthwise separable convolution applies spatial filtering per input channel and then mixes channels with a pointwise projection. MobileNet is a primary example of this architecture. It is an architectural choice with a different parameterization, not a general exact replacement for an arbitrary dense convolution. **[R29]**

Treat a proposed replacement as student design. Specify how it is initialized and trained. Compare it with simpler alternatives such as channel pruning, lower resolution, or a better-supported kernel before committing to a retraining effort.

Verification and decision

Run channel-mapping tests on a small controlled graph first. Then require strict checkpoint reload, finite outputs, task evaluation, and target-backend execution. Inspect whether the new shapes still select an efficient kernel.

Accept when the complete dependency group is valid, quality passes, and the measured bottleneck improves. **Revert** when the modification requires fragile runtime exceptions, loses critical image features, or produces no useful device-level benefit. Preserve the original group and retained indices so the transformation is reproducible.

7. Prune attention heads and distinguish them from model width

Problem

An attention module is expensive, and its configuration includes several numbers that appear interchangeable: model width, head count, head dimension, and key/value head count. They describe different tensor dimensions. Changing one field in a configuration file does not necessarily remove computation.

In a conventional multi-head projection, the internal attention width is:

EXAMPLE

```
internal_width = number_of_heads * head_dimension
```

If you halve the head count but double the dimension of each head, the internal width remains unchanged. The projection matrices may retain the same number of values. That edit is not the same as physically pruning half the heads.

Preserve the public interface

The executed attention lab keeps the external width at 32 and each head dimension at 8. It changes four internal heads to two. The query, key, and value projections shrink from 32 output features to 16; the output projection maps those 16 features back to the original 32-feature interface. **[E03]**

EXAMPLE

```
Original:
input 32 -> Q/K/V width 32 -> four 8-value heads -> output 32

Pruned:
input 32 -> Q/K/V width 16 -> two 8-value heads -> output 32
```

This preserves the surrounding residual shape without pretending that the unmasked attention function is unchanged.

Remove the corresponding slices

For separate dense Q, K, and V projections, a retained head corresponds to a contiguous group of projected feature indices. Retain those output rows in all three projections and the matching input columns in the output projection.

PYTHON

```python
# Core of the executed SplitSelfAttention transformation.
offsets = torch.arange(old.head_dim, device=keep.device)
indices = (keep[:, None] * old.head_dim + offsets).reshape(-1)
for name in ("q", "k", "v"):
source = getattr(old, name)
target = getattr(new, name)
target.weight.copy_(source.weight[indices])
target.bias.copy_(source.bias[indices])
new.out.weight.copy_(old.out.weight[:, indices])
new.out.bias.copy_(old.out.bias)
```

The full implementation rejects invalid head indices and tests equivalence to the original with the removed head outputs masked. It has no dropout, rotary encoding, cache, fused QKV storage, or shared key/value heads. Do not substitute it into a pretrained Transformer without adapting those contracts. **[E03]**

Choose heads by task impact

The paper *Are Sixteen Heads Really Better than One?* studies head pruning and shows that importance differs across attention components in the tasks evaluated. It supports investigating head redundancy, not assuming every attention layer can lose the same fraction of heads. **[R25]**

For your model, begin with validation ablation or another explicitly recorded importance criterion. Re-evaluate the entire task after recovery. A head that appears dispensable on short inputs may matter on longer contexts or different conditioning.

For diffusion, include multiple noise levels and conditioning modes in the evaluation. For a language decoder, include both prompt processing and token generation. These are proposed coverage requirements derived from the execution paths you intend to support.

Do not confuse a better kernel with a different attention rule

An IO-aware implementation can compute the same mathematical attention without storing all of its intermediate matrices at once. FlashAttention is a primary example. Windowed attention instead changes which positions can interact. One is an implementation strategy; the other can change the model's behavior. **[R26]**

PyTorch's scaled-dot-product attention API can select among implementations subject to input and backend support. It is not a promise that a particular Android runtime has the same kernels. It also applies dropout according to the supplied probability, so explicitly use zero dropout for an inference-only call. **[R27]**

Before pruning, test whether a supported memory-efficient attention implementation addresses the actual bottleneck. That experiment may avoid a structural change, but still needs numerical and task checks.

Global width is a larger migration

Changing model width from 1024 to 512 affects embeddings, residual paths, normalizers, attention projections, feed-forward layers, and often output heads. Shared or tied parameters make the dependency wider. Treat it as an architecture migration with a new configuration, not a local edit.

A practical sequence is to try internal FFN width and internal attention width independently before shrinking the public hidden dimension. This is a proposed risk-reduction strategy, not a universal optimal pruning order.

Inspect cache costs separately

For a simple decoder with the same cache layout in every layer, the logical key/value payload is approximately:

EXAMPLE

```
2 * layer_count * batch_size * cached_tokens
  * kv_head_count * head_dimension * bytes_per_value
```

The factor of two accounts for keys and values. This accounting excludes padding, temporary tensors, allocator overhead, and alternative cache layouts. Query-head pruning alone need not reduce the key/value cache in an architecture with shared KV heads. Hugging Face documents distinct cache strategies and their memory/latency tradeoffs. **[R46]**

Verification: test attention masks, position handling, cache construction, and checkpoint reload. **Decision:** accept only measured gains on the relevant inference path. **Rollback:** restore projection slices and cache configuration together.

8. Replace a dense matrix with low-rank factors

Problem and contract

A large dense linear layer contributes heavily to parameter storage or matrix multiplication. You want a smaller representation while preserving its input and output dimensions.

For a layer with weight shape `[m, n]`, replace one mapping from `n` to `m` with two mappings:

EXAMPLE

```
n -> rank r -> m
```

The dense weight contains `m*n` values. The factors contain `r*(m+n)` values. Ignoring a retained output bias, the factors are smaller when:

EXAMPLE

```
r < (m*n) / (m+n)
```

For a 4096-by-4096 matrix and rank 512, the weight count changes from 16,777,216 to 4,194,304. This is arithmetic, not a measured fourfold speedup.

Initialize with SVD

Singular-value decomposition expresses the weight using orthogonal factors and singular values. The companion function uses `torch.linalg.svd` to construct two linear modules from the retained components. The first has no bias; the second retains the original bias. **[R11]**, **[E01]**

PYTHON

```python
# Executed implementation, simplified to show the factorization.
u, s, vh = torch.linalg.svd(layer.weight.float(), full_matrices=False)
first.weight.copy_(vh[:rank].to(layer.weight))
second.weight.copy_((u[:, :rank] * s[:rank]).to(layer.weight))
if layer.bias is not None:
second.bias.copy_(layer.bias)
```

There must be no extra activation between the two factors if the objective is a low-rank approximation to the original linear operation. Adding a ReLU changes the hypothesis and requires a different test.

Test the implementation before the approximation

At full rank, the reconstructed operation should match the original within numerical tolerance. This validates factor order, transpose conventions, and bias placement. Full-rank reconstruction is not itself a useful compression result.

At a lower rank, output differences are expected. The lab checks the output shape but does not claim task quality for its low-rank candidate. A small weight-reconstruction error is not a guarantee that the errors occur in task-irrelevant directions. Evaluate the actual model and recover it if necessary. **[E01]**

Rank selection is an experiment

A singular-value spectrum can help shortlist ranks. Choose several candidates that satisfy backend shape constraints, evaluate them under the same protocol, and compare against other ways of spending the same parameter budget.

Two small matrix multiplications can be slower than one optimized large multiplication because the new graph adds an intermediate tensor and another operator invocation. This follows from the changed execution schedule; determine the net result by measurement rather than parameter arithmetic.

LoRA is not this transformation

LoRA learns low-rank updates while retaining the pretrained base weight. It reduces the number of parameters trained for adaptation. It does not, by itself, replace the base model with a small low-rank inference model. Merging a low-rank update into the base weight preserves the base matrix dimensions. **[R30]**

Low-rank **replacement** and low-rank **adaptation** can both be useful, but record which one the experiment performs. A small adapter file does not establish a small total model footprint.

Verification: require full-rank numerical agreement, valid low-rank shapes, strict reload, and task evaluation. **Decision:** retain a rank only when the deployment metric improves under the quality contract. **Rollback:** preserve the original dense weight and factorization configuration.

9. Reduce resolution and bound intermediate memory

Problem

The weights fit, but long sequences, high-resolution feature maps, or video frames create a large working set. Pruning a few layers may not address the largest simultaneously live tensors. Your next experiment should change the amount of data processed at once or the dimensions produced by the network.

Name the dimension you intend to reduce

An image tensor may have shape `[batch, channels, height, width]`. A video tensor adds a frame or time dimension, whose position depends on the implementation. An attention module may flatten image patches or video patches into a token sequence. Write down the actual layout before changing a shape.

For a fixed number of channels, halving both image dimensions produces one quarter as many feature values. For a dense attention score matrix over flattened tokens, quartering the token count reduces the number of score entries by a factor of sixteen. These are shape calculations. Fused attention may avoid materializing that score matrix, and end-to-end latency includes other operations.

Halving generated frames reduces temporal coverage unless playback rate or another part of the product changes. It is not a free memory optimization when the required output is a fixed-duration video.

Separate four different interventions

Lower input resolution changes what information reaches the model. **Lower internal resolution** changes the architecture or representation. **Tiling** changes the execution region while attempting to preserve an output resolution. **Chunking** changes how a sequence is scheduled, sometimes with carried state.

These interventions have different risks. Record them as separate candidates instead of labeling all four "activation pruning."

Begin with transformations that preserve local computation

The companion image test evaluates one stride-one convolution with an odd square kernel. For each output tile, it reads enough neighboring input pixels to cover the kernel radius. That neighborhood is the tile's **halo**. It applies padding at the original image boundary, not at every internal tile boundary. **[E03]**

EXAMPLE

```
Desired output tile
  + required surrounding input halo
  -> convolution
  -> write only that tile's output region
```

For the tested single convolution, tiled and full outputs match. The test retains the complete input and output to compare them, so it does not demonstrate bounded total process memory. It establishes the numerical boundary rule for that operator. **[E03]**

In a deeper network, determine the complete receptive field and any stride alignment. A halo sufficient for one convolution is not automatically sufficient for ten layers. A spatially global operation can make a finite local halo insufficient.

The halo is also the cost. Every interior layer of a tiled region is evaluated over the enlarged input the tile requires, so the halo is recomputed once per tile rather than shared. For `T` by `T` tiles through a sub-network of receptive-field radius `R`, the spatial work is roughly `((T + 2R) / T)^2` times the untiled work. At `T = 64` and `R = 16` that is about 2.25 times the compute for the same output. Small tiles and deep tiled regions are the expensive combination, because `R` grows with depth while `T` is what you reduced to fit the budget.

This is why tiling belongs to the memory half of the budget and not the latency half. A tiled build can fit a memory limit and still miss a latency target by a multiple. Measure latency before and after tiling, not only peak memory and output quality, and if the result is too slow the lever is a larger tile or a shallower tiled region rather than a smaller one.

A failing test is part of the lesson

The image lab deliberately adds GroupNorm independently inside each tile. This fails to reproduce GroupNorm applied to the complete convolution output. Each tile supplies different statistics. Calling `eval()` does not turn GroupNorm into fixed whole-image statistics. **[R28]**, **[E03]**

The recorded maximum absolute difference is about 0.678 for that random test. The number is not an image-quality score. It is a concrete counterexample to the claim that any convolutional model can be tiled exactly with a small overlap. **[E03]**

Likewise, global pooling, attention, and coordinate-dependent operations need explicit analysis. Blending overlapping outputs may conceal seams without restoring the original mathematical function.

Use model-supported tiling where available

Diffusers documents VAE tiling and slicing as distinct mechanisms. Tiling splits spatial work; slicing addresses batches. Its documentation notes possible tile-to-tile tone variation. Therefore, treat a documented tiling switch as a supported approximation or implementation feature that still needs visual validation, not an automatic equivalence certificate. **[R34]**

Do not assume enabling VAE tiling changes the denoiser’s largest attention allocation. Profile the stage responsible for the peak and select the matching intervention.

Chunk operations that are independent across positions

A tokenwise feed-forward module processes each token independently while sharing weights. Splitting the sequence for that module and concatenating the outputs can preserve its function when no cross-token operation is introduced. The companion test verifies this for `Linear -> GELU -> Linear` at inference. **[E03]**

PYTHON

```python
# Executed for a tokenwise FFN with no cross-token operations.
parts = [ffn(part) for part in tokens.split(5, dim=1)]
chunked = torch.cat(parts, dim=1)
```

This test retains all output pieces for comparison. A deployment implementation should avoid retaining unnecessary intermediates. It cannot apply the same independent split to full self-attention without changing which tokens see each other or introducing an appropriate attention algorithm.

One split of attention is exact

The naive split fails because it splits keys and values along with queries, which changes which tokens each query can see. A different split is exact. Each query's output depends on its own query vector and on all keys and values, but not on any other query. Splitting only the **query** axis, and passing the complete keys and values to every chunk, therefore reproduces the unsplit result while bounding the largest intermediate.

PYTHON

```python
# Exact for standard softmax attention. Splits queries only; K and V stay whole.
parts = [attention(q_chunk, k, v) for q_chunk in q.split(chunk, dim=2)]
out = torch.cat(parts, dim=2)
```

The tensor this bounds is the attention probability matrix. Its full form is `[batch, heads, queries, keys]`, and for a vision transformer, a diffusion transformer, or a long-context decoder that tensor is frequently the peak the memory budget is failing on. Splitting the query axis into `c` chunks divides that peak by `c` without changing the output contract, the architecture, or the weights, so no recovery training is required. This is the mechanism behind memory-efficient and fused attention implementations.

Two conditions bound the claim. With a causal or otherwise masked attention, each query chunk must receive the key and value prefix its mask allows and the corresponding slice of the mask; passing the whole sequence with the wrong mask slice silently changes the result. And any operation that is genuinely global over queries, such as a normalization computed across the query axis, is not covered by this split. Verify with the same masked comparison used elsewhere in this chapter rather than assuming the rearrangement was faithful.

Carry state for causal processing

The causal-convolution lab retains only the required input history between chunks. For a stride-one convolution with kernel size `k` and dilation `d`, that history has `(k-1)*d` positions. It pads only the beginning of the stream. It tests ordinary kernels, dilation, and a kernel size of one. **[E02]**

EXAMPLE

```
previous state + current chunk
  -> causal convolution
  -> output chunk
  -> retain only the required tail as next state
```

The implementation clones the retained tail. A view would share storage with the larger joined input, potentially keeping that allocation alive. PyTorch documents that views share their base storage. **[R50]**

This is not a streaming adapter for an arbitrary audio model or video Transformer. Bidirectional context, lookahead, normalization, striding, and encoder-decoder skips can require a different contract. Decide whether the target is exact equivalence, a bounded-context approximation, or a newly trained streaming student.

Resolution changes can require recovery

Changing a sample rate is not equivalent to relabeling an audio file. Changing patch size can alter projection shapes. Changing spectrogram bins can alter both encoder and vocoder interfaces. Changing a latent channel count can affect the encoder, denoiser, and decoder together.

For every proposed representation change, name the producer and all consumers. Build adapters or retrain the affected components explicitly. A shape-compatible tensor is not necessarily semantically compatible.

Verification and acceptance

For exact scheduling changes, compare full and chunked outputs at boundaries, on short inputs, on partial final chunks, and across supported shapes. For approximate changes, evaluate the final task and boundary-specific failures separately.

Accept when the specific peak falls and quality remains within the contract. **Revert** when tiling creates tone shifts, chunking creates temporal discontinuities, or lower resolution removes information the product must retain. Never report savings from shortened or downscaled output as same-task acceleration without disclosing the changed output contract.

10. Recover a changed model with fine-tuning and distillation

Problem

A structural change executes correctly but loses task quality. You now have two separate engineering tasks: confirm that the transformation is implemented correctly, and train the modified architecture to use its remaining capacity. Training cannot reliably compensate for an unnoticed channel-mapping bug.

Fine-tuning updates the student using the task objective. **Distillation** adds learning signals from a teacher. The teacher may be the original checkpoint or another suitable model. The original distillation paper and PyTorch's tutorial provide primary descriptions and implementation examples. **[R05]**, **[R04]**

Define what the student is learning

For classification, the student can learn from labels and the teacher's class probabilities. A typical objective combines ordinary cross-entropy with a temperature-scaled divergence:

PYTHON

```python
# This calculation is executed inside fit() in lab.py.
logits = student(x)
with torch.no_grad():
teacher_logits = teacher(x)
temperature = 2.0
supervised = F.cross_entropy(logits, labels)
soft_targets = F.kl_div(
F.log_softmax(logits / temperature, dim=-1),
F.softmax(teacher_logits / temperature, dim=-1),
reduction="batchmean",
) * temperature**2
loss = 0.5 * supervised + 0.5 * soft_targets
```

Choose the recovery budget and step size

The learning rate that trained a model from random initialization is the wrong learning rate for recovering one. A converged checkpoint sits at a point that a large step destroys within a few hundred updates, and the resulting model can be worse than the unrecovered candidate. Start one to two orders of magnitude below the rate the checkpoint was originally trained with, or below the published fine-tuning rate for that model, and add warmup and decay.

Run one control before trusting any recovery number: apply the identical recipe to the **unmodified** original. If the original also degrades, the recipe is destroying the model and the structural edit is not what you are measuring. The companion experiment includes exactly this control as `baseline_extra_ce`, an unpruned model given the same extra supervised budget, so that a recovered candidate is compared against more training rather than against no training. **[E01]**

Decide the budget the same way. Record quality against update count and stop when it plateaus rather than at a round number. A recovery that needs a budget comparable to original training is evidence that the edit removed something load-bearing, which is a result about the edit, not a reason to keep training.

The temperature and weights above are fixed choices for the synthetic lab, not recommended defaults for every model. The teacher is held in evaluation mode and receives no gradient updates. The student receives a fresh optimizer after surgery. **[E01]**

Use `no_grad()` for the teacher branch when its outputs participate in a student loss. Do not indiscriminately wrap the student training step in inference mode. That would prevent the gradient computation the recovery step needs.

Distillation targets depend on the task

For a regression model, matching class probabilities may make no sense. You might compare a teacher's continuous outputs, selected representations, or task-specific features. When teacher and student widths differ, an explicit projection can align intermediate dimensions. Remove training-only adapters from the deployment graph unless they are deliberately part of the student.

For detection or segmentation, preserve the meaning and alignment of spatial outputs. For sequence models, align valid positions and mask padding consistently. For audio synthesis, waveform alignment and perceptual criteria require more care than applying mean squared error to arbitrary samples.

These are proposed design checks. The presence of a loss function in a paper does not establish that it is appropriate for a different representation or application.

Diffusion recovery needs aligned noise and conditioning

A denoiser receives a noisy representation, a noise level or timestep, and possibly text, image, or other conditioning. For a teacher-student comparison, feed both the same noisy state and conditioning. Match the same prediction parameterization, such as noise, clean sample, or velocity, rather than comparing tensors with different meanings. Scheduler documentation exposes these prediction-type distinctions. **[R42]**

A generic proposal is:

EXAMPLE

```
sample a training example and noise level
construct the same noisy input for teacher and student
run both with the same conditioning
compute a compatible prediction or feature loss
update only the student
periodically evaluate complete generated outputs
```

This is not a complete reproduction of progressive distillation or consistency training. Those methods define specific sampling and training objectives. Use their exact method when claiming to implement them. **[R38]**, **[R39]**

Matching one denoising call can support recovery experiments, but it does not prove that errors remain acceptable across the complete generation trajectory. Inspect completed samples as well as the training loss.

Compare recovery against fair controls

At minimum, compare the original checkpoint, the changed model before recovery, and the changed model after recovery. To understand the contribution of distillation, compare it with ordinary fine-tuning under a documented budget. Additional training of the original model is also informative.

The lab includes both cross-entropy and distillation recovery for the depth-pruned model. It also trains the original architecture for additional epochs. This prevents an improvement after pruning from being automatically attributed to pruning itself. **[E01]**

An equal number of epochs is not necessarily equal compute. Distillation evaluates the teacher and student, while ordinary fine-tuning may evaluate only one model. State whether the comparison equalizes steps, examples, wall time, or compute allocation.

Preserve independent evaluation

Use training data for recovery. Use validation for stopping or candidate selection if that is the recorded protocol. Keep the final test independent. Do not fine-tune on failure examples from a previously reported test and continue calling the result held out.

When using teacher-generated data, record its origin and sampling procedure. Keep independently reviewed task examples in the final suite. A student that reproduces teacher errors is successful at imitation but not necessarily at the application.

Stop when the evidence says to stop

A student can lack sufficient capacity for the required operating envelope. A teacher can lack the capability you need to preserve. Distillation does not guarantee recovery of every removed function or create a fixed percentage of retained quality.

Accept recovery only when the complete candidate passes the same task contract used for the baseline. **Revert** or choose a less aggressive architecture when improvements require excessive training, critical behavior remains missing, or the deployed implementation fails to deliver the resource gain.

11. Quantize deliberately, not by file extension

Problem

You need fewer parameter bytes, less memory traffic, or a backend that requires quantized inputs. Numeric precision is a separate experiment from architecture pruning. A structurally smaller FP32 model and an equally shaped INT8 model save different resources.

Distinguish storage, activations, and arithmetic

A label such as W4A16 means four-bit weights and sixteen-bit activations in the stated scheme. It does not say that every operator uses that representation or that accumulation occurs at four bits. Some components can remain at higher precision.

For dense weight payload alone, 100 million values occupy approximately 400 MB at FP32, 200 MB at FP16, 100 MB at INT8, or 50 MB when packed at four bits. This ignores scales, zero points, alignment, metadata, unquantized tensors, and runtime repacking. It is not a total-memory estimate.

FP16 and BF16 are floating-point formats. INT16 is an integer format. An audio file using sixteen-bit PCM is a representation of the input or output signal, not evidence that a neural model computes using INT16.

Understand what calibration estimates

In an affine quantizer, an integer value is interpreted using a scale and a zero point. Values outside the represented range can be clipped. The choice of range therefore matters as well as the bit count. ONNX Runtime documents this mapping and its static and dynamic quantization workflows. **[R14]**

Granularity and sign are separate choices from bit width

A scale and zero point can be shared by an entire tensor, or held one per output channel of a weight. Per-tensor is the smaller representation. Per-channel costs one scale per output channel and usually recovers most of the accuracy lost on layers whose output channels have very different value ranges. ONNX Runtime’s static and dynamic helpers default to per-tensor. **[R14]**

This interacts directly with channel pruning. After Chapter 6 removes channels, a single tensor-wide scale is set by whichever surviving channel has the widest range, and the narrower channels lose resolution. If post-training quantization costs several points of accuracy on a model you pruned, test per-channel weight scales before concluding that the pruned architecture cannot be quantized. Granularity is one argument; it is cheaper to test than a new architecture.

A symmetric scheme fixes the zero point at zero and represents a range centered on zero. An affine or asymmetric scheme carries a nonzero zero point and can represent a one-sided range, which is what an activation after a ReLU actually needs. Signed and unsigned eight-

bit integers differ the same way. Record the granularity and the scheme alongside the bit width, because INT8 alone does not identify the representation, and two INT8 models can differ by several points of accuracy for this reason alone.

For static activation quantization, build a calibration subset that represents the inputs and intermediate conditions expected in deployment. Keep it separate from the final test. Calibration is not recovery training, although both can consume training examples.

For image models, include supported resolutions, contrast ranges, and image conditions. For video, include motion and temporal lengths. For language, include representative context lengths and token distributions. For audio, include the intended signal conditions. These are proposed coverage dimensions, not a guarantee that a small curated set is sufficient.

For diffusion, calibrating only clean images or a single noise level can miss the distribution actually seen during denoising. Q-Diffusion specifically studies diffusion quantization and timestep-dependent behavior. This supports using diffusion-aware calibration, not treating a generic image-classification recipe as established for a denoiser. **[R40]**

Choose the workflow according to the backend

Post-training quantization starts from trained weights and estimates a lower-precision representation. **Quantization-aware training** includes simulated quantization effects during training before conversion to a deployable representation. The correct operators and quantization scheme depend on the target toolchain.

The three post-training paths differ in where the activation range comes from, and that difference decides which one suits a backend. **Static** fixes activation ranges ahead of time from a calibration set, needs representative data, and yields a fully integer graph; it is the path an NPU or a quantized convolution kernel generally requires. **Dynamic** recomputes activation ranges on every call, needs no calibration set, and pays that estimation cost at each inference; it suits matrix-multiplication-dominated language models and is a poor fit for convolutional vision models, where the per-call overhead can exceed the arithmetic saved. **Weight-only** compresses stored weights and leaves activations in floating point, reducing download size and memory without necessarily reducing compute time.

The one-line dynamic helper is the most discoverable of the three and is therefore the most common wrong choice. Decide according to the constraint that actually binds the budget: download size, resident memory, or latency.

For ONNX Runtime, examine the supported static, dynamic, and weight-only paths for the exact operators and execution provider. INT4 support in its documented tooling is operator-specific, including suitable constant-weight matrix multiplications. An INT4 model type in the interchange format is not a universal INT4 convolution kernel. **[R14]**, **[R49]**

Do not combine a quantizer intended for one backend with another backend and assume the resulting graph will be accelerated. Shape, layout, precision, and operator support must agree throughout the deployment path.

Test selective precision before changing everything

Create a floating-point reference export first. Then quantize a supported component and compare it with that reference. Add more components only when the first change is understood. This staged approach is a proposed debugging strategy.

A mixed-precision candidate may retain sensitive operations at higher precision while compressing large projections. The relevant question is not whether 100% of the model is INT8. It is whether the executed graph meets quality and resource requirements.

Repeated dequantization and requantization can add overhead. Weight-only compression may reduce storage without reducing all activation allocations. Measure the runtime rather than assuming that a smaller checkpoint implies faster execution. ONNX Runtime explicitly notes that quantization performance depends on hardware and model behavior. **[R14]**

Debug the first divergence

Compare tensors at a small number of meaningful boundaries. Find where the candidate first differs substantially from the floating-point reference, then inspect saturation, range estimation, precision exclusions, and unsupported operations around that boundary.

Do not retain every activation from a large model just to debug it. Sample a small diagnostic batch or capture one boundary at a time. Record the comparison metric and tolerance. Relative error near zero can be misleading, so include absolute error or another meaningful normalization.

For a generator, passing local tensor tolerances does not replace final-sample evaluation. Small recurrent deviations can produce different outputs. A numerical difference can also be harmless if the task contract allows it. Separate implementation fidelity from product quality.

Recalibrate after structural surgery

Pruning or recovery training can change activation distributions. Reusing quantization statistics from the unmodified checkpoint is an additional hypothesis that must be tested. The conservative workflow is to recalibrate the final recovered structure, then rerun the complete evaluation.

Verification: inspect converted operators, precision boundaries, runtime logs, and final task outputs. **Decision:** keep a quantized candidate only when the actual backend improves the chosen metric. **Rollback:** preserve the recovered floating-point checkpoint and calibration manifest alongside the quantized artifact.

12. Refactor image and video diffusion systems

Problem

A diffusion system can consume substantial resources in several different places: condition encoding, repeated denoising, latent decoding, temporal processing, and output assembly. “Make the diffusion model smaller” is too broad to specify a useful experiment.

Latent diffusion uses an encoded representation rather than operating entirely in image pixels. A Diffusion Transformer can operate on latent patches instead of a U-Net backbone. These architectures have different structural dependencies, even when both are part of an image-generation pipeline. **[R31]**, **[R32]**

Draw the complete pipeline first

A representative pipeline is:

EXAMPLE

```
prompt and conditioning inputs
  -> tokenizer / image preprocessing
  -> condition encoder
  -> initial latent or noisy input
  -> repeated denoiser and scheduler updates
  -> latent decoder
  -> image or video postprocessing
```

Not every pipeline contains every component. Some use additional encoders, adapters, or temporal modules. Inspect the implementation you have rather than treating this diagram as a universal interface.

Measure condition encoding, one denoiser evaluation, the complete denoising loop, decoding, and final encoding separately. Record model-loading time as well. A memory peak in the decoder needs a different intervention from a peak in the denoiser’s attention.

Sampling steps are not network layers

A model can contain 24 blocks and be evaluated 30 times during sampling. Removing a block changes one network evaluation. Reducing sampling steps changes how many evaluations are requested and where they occur along the trajectory.

Do not report “30 layers reduced to 10” when you changed `num_inference_steps`. Preserve the distinction in configuration, benchmark names, and review comments.

A simplified cost model is:

EXAMPLE

```
total time = condition encoding
           + sum of actual denoiser evaluation times
           + scheduler work
           + decoding and postprocessing
```

The number of user-visible sampling steps need not equal the number of denoiser calls. Solver behavior and guidance implementation can affect the executed work. Count actual calls or effective evaluated batches instead of inferring them from one setting. Diffusers exposes scheduler configurations and inference-performance options for this purpose. **[R42]**, **[R48]**

Fewer steps usually change total repeated work more directly than the largest single-step activation. If the same denoiser and shape remain resident, a step-count reduction need not lower that peak.

First experiment: a compatible sampler configuration

Start with an unchanged checkpoint and the sampler configuration recommended for it. Record prediction type, timestep or sigma schedule, solver settings, guidance behavior, seed list, output shape, and actual denoiser evaluations.

Test a small number of compatible step counts. Do not switch scheduler, precision, resolution, and checkpoint in the same candidate. Evaluate prompt following, details, and failures on the same suite. Scheduler documentation is explicit about prediction types and solver configuration; arbitrary combinations are not interchangeable. **[R42]**

Do not turn an ordinary many-step model into a one-step model by setting the loop count to one and calling it distilled. Progressive distillation trains a model to reproduce a longer sampler with fewer learned steps. Consistency models also use a specific training construction. These are learned changes, not merely loop optimizations. **[R38]**, **[R39]**

Second experiment: isolate precision and attention implementation

For a supported device, test the precision and attention kernel independently while retaining the same sampling configuration. Measure numerical differences on a fixed noisy input, then evaluate complete outputs across the suite.

A kernel that avoids materializing large attention intermediates can address memory without changing the attention neighborhood. Replacing global attention with local attention changes the interaction pattern and is a different model hypothesis. **[R26]**

Treat compilation as another separate candidate. Report compilation and warmup cost rather than excluding them from a one-image user workflow without explanation. The benefit of a compiled steady-state loop may matter for a batch tool and be less useful for a short session. **[R48]**

Third experiment: decoder and feed-forward scheduling

Try model-supported VAE tiling when the spatial decoder is the bottleneck. Try slicing when a supported decoder processes multiple images as a batch. Neither is a blanket solution to denoiser memory, and tiling can change local tone. **[R34]**

For video, distinguish decoding frames in smaller groups from changing the temporal model itself. Diffusers' Stable Video Diffusion documentation describes feed-forward chunking and `decode_chunk_size`, and warns that very small decode chunks can produce flickering. This is an implementation-specific tradeoff, not evidence that all video models can decode frames independently. **[R35]**

Use the normalization counterexample in the companion lab as a reminder to test the mathematical boundary of any custom tiler. Production decoder behavior requires its own tests.

Fourth experiment: structural pruning of the denoiser

Choose one structurally valid target: repeated residual blocks, internal FFN channels, compatible attention heads, or a coupled convolution group. Establish importance under representative noise levels and conditioning before removing it.

A diffusion denoiser is reused across the generation trajectory. A block that appears inactive at one noise level can matter at another. The Diff-Pruning paper proposes a timestep-aware structural-pruning method, providing primary evidence that diffusion pruning requires attention to this reuse. Its experiments do not validate arbitrary block deletion in a different video generator. **[R41]**

Use the following proposed protocol:

EXAMPLE

```
freeze checkpoint and sampler
  -> collect representative noisy states and conditions
  -> rank a small set of structurally valid candidates
  -> remove one candidate group
  -> run shape and prediction checks
  -> recover with a compatible objective
  -> evaluate complete samples and performance
```

For a U-Net, inspect resolution transitions and skip connections before deleting a block. For a DiT, inspect adaptive conditioning, position handling, token shapes, and the final projection. A matching hidden width is necessary in many edits but not sufficient to establish a valid bypass.

Fifth experiment: lower spatial or temporal resolution

Reducing height and width can substantially reduce the number of latent positions. Increasing patch size can also reduce token count, but changes patch projection and reconstruction interfaces. Treat it as an architecture change rather than a convenient inference flag. The DiT paper explicitly studies depth, width, and token count as different scaling axes. **[R32]**

For video, record generated frame count, conditioning frame rate, playback frame rate, and temporal compression separately. Lowering playback frame rate does not retroactively reduce the model's generation work. Generating fewer frames and interpolating them adds another model and a different artifact profile.

Review fast movement, camera motion, occlusion, object persistence, and scene transitions. A framewise visual score cannot alone establish temporal quality. VBench provides a primary reference for separating these dimensions. **[R43]**

Caching changes the validity conditions

Caching a condition embedding is exact only when every input and model state determining that embedding is unchanged. Include tokenizer version, truncation, encoder weights, prompt text, and relevant conditioning in the cache key.

Reusing a denoiser activation at a different timestep is usually an approximation unless the method establishes a special invariant. Do not confuse this with caching an unchanged text embedding. Treat cross-step activation reuse as its own proposed experiment with an explicit refresh policy and quality checks.

Offloading is not the same as reducing total RAM

A desktop pipeline can move inactive weights between GPU and CPU memory. Diffusers documents several offloading strategies and their transfer costs. This can reduce device-local memory without eliminating the host copy or reducing the size of the model itself. **[R34]**

For a mobile system, identify the actual memory domains and copy behavior rather than transplanting a desktop CUDA recipe. An app that no longer exceeds a GPU allocation can still exceed the phone's tolerable total footprint. Measure both the accelerator and process-level behavior where those measurements are available.

What a successful report contains

Publish the exact checkpoint and component revisions, complete generation settings, prompt and conditioning suite, seed procedure, output shapes, task-specific evaluation, cold and warm timing, resource metrics, and failed examples.

Accept only a candidate that meets the stated generation contract. **Revert** a candidate that improves static images while introducing video flicker, reduces fidelity outside the tuned prompts, or claims same-task speed by silently shortening the output.

This chapter provides experiment designs supported by primary methods and documentation. No production diffusion checkpoint was compressed or benchmarked in the preparation environment.

13. Export the candidate and inspect the executed graph

Problem

The changed model works in eager PyTorch. You now need an artifact that another runtime can load, execute, and optimize. Export is an interface migration with its own tests, not the final proof that a candidate is mobile-compatible.

ONNX describes a model representation. ONNX Runtime executes supported graphs using its available kernels and execution providers. The interchange format and the runtime are different layers of the system. **[R49]**, **[R17]**

Export after reconstructing the architecture

Load the changed architecture from its saved configuration and require strict checkpoint loading. Do not export a notebook object whose class definitions or in-place mutations cannot be reconstructed.

Define input names, shapes, dtypes, output names, and any state tensors. For a recurrent decoder, include the cache contract. For a denoiser, expose the noisy state, timestep, and conditioning consistently. For a multi-stage system, consider exporting stable component boundaries rather than capturing a Python-heavy orchestration layer.

The PyTorch 2.10 exporter supports a `torch.export`-based path, shape constraints, a diagnostic report, and optional ONNX Runtime verification. These facilities help detect export problems; they do not evaluate the product's task quality. **[R13]**

PYTHON

```
# Integration template. ONNX execution was not available in this environment.
torch.onnx.export(
model.eval(),
(example_input,),
"candidate.onnx",
input_names=["input"],
output_names=["output"],
opset_version=target_opset,
dynamo=True,
report=True,
verify=True,
)
```

Set `target_opset` from the supported export/runtime combination. Do not copy the newest opset into an older deployment runtime without checking compatibility.

Then verify the artifact, because the request is not always honored. With `dynamo=True` the exporter builds against a recent operator library and attempts a down-conversion to the requested opset. That conversion can fail without failing the export: the file is still written, at the higher opset. Measured with torch 2.10.0 and onnx 1.23.0 on this package's own model, requesting opset 15 or 17 produced a model at 15 or 17, while requesting opset 13

produced a model at 18. A pinned `onnxruntime-android` build or a vendor converter that caps below the saved opset will then reject the model, or silently fall back, after both the export and the desktop parity check passed.

PYTHON

```python
# Verify the saved opset rather than trusting the request.
import onnx
saved = onnx.load("candidate.onnx")
print([(o.domain or "ai.onnx", o.version) for o in saved.opset_import])
```

Read that value, compare it against the pinned runtime's supported maximum, and treat a mismatch as a failed export rather than a warning. Store any external weight files with the graph and include all of them in artifact integrity checks.

Test three boundaries separately

First compare the reconstructed PyTorch model with the original edited object. Then compare the exported floating-point graph with reconstructed PyTorch. Finally compare the quantized or backend-optimized graph with the exported reference.

This isolates structural errors, export errors, and conversion errors. If all three changes happen at once, a bad output does not identify which stage introduced it.

For numerical checks, use representative inputs and the supported shape envelope. Random input can catch shape errors, but it does not establish task fidelity. For generators, compare one step on identical state and also compare completed outputs.

Dynamic shapes are not universally free

A dynamic dimension makes the model more flexible, but the backend may impose restrictions or compile shape-specific variants. ONNX Runtime's QNN documentation states that its documented path requires fixed shapes and supports only a subset of operators. Verify those constraints against the pinned toolchain. **[R18]**

A proposed deployment strategy is to define a small set of shape buckets. Each bucket needs correct preprocessing, padding or cropping rules, and its own tests. Do not improve benchmark results by padding away difficult cases or silently truncating user input.

Read the execution-provider allocation

An accelerator can execute only the graph portions it supports. The runtime may partition supported subgraphs and leave other work on another provider. Cross-provider transitions can introduce transfers and synchronization. ONNX Runtime documents this execution-provider model and recommends testing the actual mobile combination. **[R17]**, **[R16]**

Inspect the execution trace rather than assuming that requesting an NPU means every operation ran on it. Record unsupported operators, their attributes and shapes, and the cost of crossing each boundary.

Two mechanisms produce that information and neither is on by default. Per-node placement, and the reasons individual operators were rejected, appear only at verbose session logging, which is selected when the environment and session options are constructed rather than afterwards. Separately, the runtime can write a profiling file for a

session containing per-node timings and the provider that executed each node, read after the run rather than during it. Both are reachable from the same Java and Kotlin API used to load the model, so this check belongs in an instrumented debug build rather than in a desktop script.

Requesting a provider, observing a latency improvement, and recording that the model ran on the accelerator is the specific mistake this section exists to prevent. A partial placement can be faster than CPU alone while still leaving most of the graph off the accelerator, which changes what happens on the next device, the next driver version, or the next operator added to the model. Record the provider for every node, or record that the allocation was not measured.

Sometimes changing an export decomposition is enough. Sometimes a CPU-only graph is preferable to a fragmented accelerated graph. Sometimes the architecture needs to change. These are measurements to perform, not a fixed ranking of CPU, GPU, and NPU.

Verify graph optimizations independently

Constant folding, dead-node removal, and supported operator fusions can reduce execution overhead without the same learning problem as pruning. ONNX Runtime documents graph-optimization levels. Some optimizations are hardware-dependent, so retain the target platform in the artifact metadata. **[R15]**

Do not equate an operator disappearing from a graph viewer with a learned capability disappearing. It may have been fused into another kernel. Conversely, a smaller serialized graph can still require a large temporary workspace.

Runtime size is a different budget

ONNX Runtime can use a reduced operator configuration to build a runtime containing only the required operator and type support. This reduces the runtime package, not necessarily the model's activation peak. Generate the configuration from all model variants the application must support. **[R19]**

ExecuTorch is another edge-deployment stack with its own export, lowering, and backend support. It is not simply a different reader for every ONNX artifact. Evaluate toolchains against the graph and devices you actually need, rather than selecting one by a general popularity claim. **[R20]**

Verification: strict reload, export equivalence, operator coverage, provider trace, and final task tests. **Decision:** keep the candidate only after it runs through the intended deployment path. **Rollback:** retain the recovered checkpoint, export settings, runtime build, and previous deployable artifact.

14. Integrate the model into a device application

Problem

A model benchmark succeeds in isolation, but the application also loads media, allocates buffers, displays previews, writes files, and responds to lifecycle events. Device compatibility is a system property, not just a tensor-shape property.

The procedures below are proposed integration tests. No Android device was available for the accompanying experiments.

Define an application memory envelope

Measure process-level behavior separately from tensor payloads. Keep managed heap, native allocations, mapped files, accelerator memory, and shared accounting distinct. Android documents memory-pressure behavior and device-dependent heap constraints; it does not define a universal 900 MB CPU-activation allowance. **[R21]**

A useful initial diagnostic is:

```bash
# Replace the package name with the application under test.
adb shell dumpsys meminfo com.example.modelapp
```

This produces a snapshot, not a guaranteed record of the peak between snapshots. Use a trace or a suitable profiler for short-lived spikes, and keep the measurement method in the report. The meaning of the fields is documented in Android's `dumpsys` reference. **[R22]**

Do not add parameter bytes to RSS and call the result total RAM. The resident weights may already be included in RSS. Do not add independently measured stage peaks and claim they are simultaneously live.

Schedule stages deliberately

When components run sequentially, consider loading only the active stage and retaining minimal intermediate state. For diffusion, the denoiser is reused many times, so repeatedly unloading it between every step can add avoidable overhead. For a multi-model media pipeline, persistent outputs can connect stages without keeping every model resident.

Unloading an object does not guarantee an immediate fall in the measured process footprint. Allocator caches, mapped storage, shared references, and runtime workspaces can remain. Test repeated jobs in a clean process and in a long-lived process. Record whether the working set stabilizes or grows.

Bound application queues. Decoding an entire video to uncompressed frames can dominate memory before the model runs. Limit pending inputs and outputs, and make the producer wait when the consumer cannot keep up. This is a proposed application-level memory policy.

Benchmark sustained work

Measure cold initialization, first inference, warm calls, and complete jobs. Include the supported workload lengths, not only a single short input. Record device model, SoC, OS build, runtime, thread count, charging state, and relevant thermal conditions.

Android's Thermal API provides status and headroom signals for adapting workload. It does not guarantee the same sustainable performance on all devices. Define a pause, reduced-concurrency, or quality-tier policy and test that it preserves a coherent job state. **[R23]**

Do not infer sustained latency from a desktop CPU result or a phone's advertised peak accelerator throughput. Your application executes a particular graph with particular memory movement and scheduling.

Test concurrency rather than maximizing threads

Use a documented range of thread and concurrency configurations. Test the entire application, since a codec, a rendering thread, and an inference runtime may compete for the same cores and bandwidth.

Separate throughput goals from response-time goals. Running two model instances can improve completed jobs per unit time while worsening per-job latency and memory. The correct configuration is the one that satisfies the product contract.

Treat cancellation and process death as normal cases

An offline media job should record its completed stage, input digest, model versions, and resumable artifacts. Write final output atomically where the platform permits so a partial file is not presented as a successful export.

Use the appropriate platform mechanism for user-visible or deferred long-running work. Android's long-running worker guidance includes version-specific constraints; it does not authorize an unlimited background computation simply because the task uses ML. Check the rules for the application's actual target SDK. **[R24]**

Test cancellation during model loading, inference, and output writing. Test insufficient storage and damaged model files. These failures belong in the deployment suite even when all tensor calculations are correct.

Ship a complete artifact contract

Deliver the model file

A validated candidate still has to reach the device, and this is the step at which someone other than the engineer who compressed the model can block the release. Choose between three delivery routes before committing to a model size. A model bundled in the application package is the simplest and counts against the store's install-size limit. Install-time or on-demand asset delivery keeps the base package small and moves the model into a separate, platform-managed download. An application-managed download gives the most control over versioning and rollback, and requires handling storage, integrity, retries, and the first-run state in which no model is present yet.

Two costs are easy to miss. A compressed asset inside the application package has to be extracted before it can be used, which adds a second copy on disk and a cold-start delay on first use; the usual remedy is to leave the model's file extension uncompressed so the

runtime can map the file directly. And a model that ships inside the application can only be replaced by shipping the application, which is why a separate delivery route is what makes a model rollback independent of an application rollback.

Measure the installed size and the cold-start time on a device rather than the file size on a build machine. A quantized model that meets its latency target and fails the install-size limit has not met its acceptance criteria.

Package the architecture variant, checkpoint digest, preprocessing version, runtime compatibility, input envelope, and output format. Validate downloads before loading. Retain a known-good model version and a way to select it when a candidate fails a device-specific check.

Accept a deployment only after complete jobs pass quality, lifecycle, storage, memory, and sustained-performance tests on the supported device matrix. **Revert** a model update independently from an application update when the packaging and compatibility design permit it.

15. Inspect the executed experiments

What the experiments establish

The companion programs are deliberately small. They let you inspect the complete transformation, test the shape contract, and separate a programming error from an approximation error. They are not miniature evidence that a particular pretrained video generator can be compressed successfully.

There are three original experiment records. RefactorNet is a trained synthetic classifier. The causal-convolution example tests stateful chunking. The vision and attention example tests structural equivalence on randomly initialized modules. Their scope and failure conditions differ, so their results must not be combined into one compression claim. The TinyNet crash course in Chapter 0 is a fourth record with its own separate scope; Appendix B describes it. **[E01]**, **[E02]**, **[E03]**, **[E04]**

Experiment A: block removal and feed-forward pruning

The classifier maps 16 input values to four classes. Its stem produces 32 features. Six residual blocks preserve that width and each expands internally to 96 features. The labels come from a fixed nonlinear function implemented in `make_data`, not a downloaded media dataset.

The run used 4,096 training examples, 1,024 validation examples, and 1,024 independently generated test examples. The first 256 training inputs supplied channel-importance statistics. The original model trained for 24 epochs. Recovery variants received eight additional epochs with fresh optimizers. The seed was 20260924. The saved protocol records the exact learning rate, batch size, and distillation settings. **[E01]**

Each removable block was tested separately against validation cross-entropy. Indices below are zero-based. A negative delta means the ablated model had lower validation loss than the original in this experiment.

Removed block	Validation loss	Loss delta	Validation accuracy
0	0.6585	+0.1801	80.96%
1	0.6566	+0.1781	82.03%
2	0.5262	+0.0478	83.59%
3	0.5585	+0.0800	83.20%
4	0.4447	-0.0338	85.55%
5	0.3978	-0.0807	87.01%

Block 5 was selected by the recorded loss criterion. The selection did not use test accuracy. The run then compared depth reduction, internal width reduction from 96 to 64, their combination, and recovery variants. The classifier’s public hidden width remained 32. **[E01]**

Recorded quality and timing

In the table, **raw** means no recovery after surgery. **CE** means recovery with supervised cross-entropy. **KD** means the executed mixture of supervised loss and teacher-student distillation. These names describe the training recipe, not a claim that one method is always superior.

Variant	Parameters	Validation accuracy	Test accuracy	Median ms	p95 ms
`baseline`	38,756	86.82%	85.25%	0.1705	0.2966
`depth_raw`	32,420	87.01%	84.96%	0.1456	0.2353
`depth_ce`	32,420	85.74%	85.16%	0.1456	0.2985
`depth_kd`	32,420	87.01%	86.13%	0.1450	0.2255
`width_raw`	26,276	84.96%	86.52%	0.1581	0.2394
`width_kd`	26,276	87.01%	87.40%	0.1680	0.2731
`combined_raw`	22,020	84.47%	85.55%	0.1433	0.2578
`combined_kd`	22,020	86.23%	87.40%	0.1434	0.2583
`baseline_extra_ce`	38,756	85.55%	87.21%	0.1699	0.2218

Timing was measured on a Linux x86_64 host reporting an AMD EPYC 9V74 CPU, using PyTorch 2.10.0+cpu and one intra-op thread. The workload was one input of shape `[1, 16]`. Each model received 30 warmup calls, followed by five rounds of 200 measured calls. Model order was shuffled between rounds. All models remained resident. These are warm Python/PyTorch call timings, not initialization, peak-memory, GPU, or Android measurements. The raw samples and round ordering are included. **[E01]**

This harness's own noise floor can be read from those records. Across models, the gap between each model's smallest and largest per-round medians ranged from 0.59% to 2.51% of that model's overall median. The `baseline` and `baseline_extra_ce` variants have the same architecture and parameter count, yet their medians differ by 0.37%. Differences at that scale are measurement noise. The 15.9% median reduction discussed below is far outside it; a one-point comparison between two pruned variants generally is not.

The `p95` column is reported for completeness and should not be read as a tail-latency ranking. Each value is a single 95th percentile over pooled samples from one host, and its ordering contradicts the medians: `baseline_extra_ce` records the lowest `p95` of all nine models while `baseline`, the same architecture with the same parameter count, records nearly the highest. Warm in-process Python timings have a long right tail driven by the host rather than by the model. A tail or frame-deadline number that a release gate can rely on has to be measured on the target device, over many runs, with the percentile itself repeated. **[E01]**

Interpretation without overstating the result

The combined KD model has 22,020 parameters compared with 38,756 in the original, a reduction of 43.2%. Its recorded median call time was 15.9% lower. These quantities did not fall by the same proportion. This is one small-model CPU observation, not a predicted speedup for a larger model or another runtime. **[E01]**

The original model's recorded test accuracy was 85.25%. The combined KD variant recorded 87.40%, but the unpruned model with the extra supervised training budget recorded 87.21%. The difference between the latter two is two examples out of 1,024. One seed and one test split do not establish that the compressed model is generally more accurate. **[E01]**

The ablation selected using validation loss slightly reduced test accuracy before recovery. Some recovery runs improved test accuracy while their validation results did not improve in the same way. Preserve these results instead of selecting only a favorable metric. They illustrate why the selection criterion, final test, and training controls must remain explicit.

The run does not include matched-compute recovery, repeated training seeds, confidence intervals, or a student trained from scratch. It therefore does not establish that distillation was necessary, that it was more compute-efficient, or that this architecture is optimally compressed. It also did not measure peak process memory or peak live activations. Parameter bytes and module-output sizes are not substitutes for those measurements.

Experiment B: chunked causal convolution

`chunking.py` tests a single causal one-dimensional convolution with persistent left context. The three tested configurations include a kernel of size five, dilation one or two, and a kernel of size one. The largest recorded absolute error was approximately `2.38e-7`, within the test tolerance. **[E02]**

This supports the implementation of the state buffer for the tested operation. It does not establish that an arbitrary noncausal audio encoder can be made causal or streamed without changing its behavior. The test collects outputs for comparison and does not measure end-to-end peak memory.

Experiment C: image, attention, and chunking contracts

The convolution-channel test reduces a two-convolution module from 663 to 333 parameters. The attention-head test reduces its custom module from 4,224 to 2,128 parameters while preserving its external 32-feature interface. Both match their specifically masked reference functions within the recorded checks. Neither is asserted to match the original unmasked function. **[E03]**

The image tiler reproduces a single convolution for four tested tile sizes when it includes the required input halo. One of those sizes exceeds both image dimensions and therefore executes as a single tile; the record marks which sizes actually cross a seam, because a tile larger than the image tests no seam at all. Tokenwise feed-forward chunking also matches the unchunked calculation for the tested module. These tests use random weights and inputs, so they establish implementation behavior, not learned task quality. **[E03]**

The deliberately incorrect per-tile GroupNorm calculation produced a maximum absolute difference of approximately 0.678 from normalization over the complete feature map. The test passes by confirming the expected mismatch. It is a regression test against an invalid assumption, not a successful approximation to deploy. **[E03]**

What to reproduce next

Run the self-tests first. Then reproduce the synthetic training experiment in a new output directory. Do not overwrite the supplied evidence. Once you understand the tests, replace the toy architecture and data with a specific checkpoint and task suite, retaining the same distinction between structural checks, task evaluation, and device measurements.

The first useful production experiment is a small, reversible change with a clearly measured bottleneck. There is no requirement to begin with the most aggressive compression technique.

16. Plan experiments across model families

A reusable experiment contract

Before coding, write one sentence that connects an observed cost to a proposed change. For example: “The decoder’s spatial activation peak exceeds our budget; test its supported tiling path without modifying the denoiser.” This is narrower and more testable than “optimize the model.”

Use the following template. Empty measurement fields are intentional. They must be populated by execution, not estimated from the desired outcome.

YAML

```yaml
experiment_id: exp_001
status: proposed
baseline:
 architecture_revision: REQUIRED
 checkpoint_sha256: REQUIRED
 runtime_and_backend: REQUIRED
 input_contract: REQUIRED
hypothesis: REQUIRED
one_primary_change: REQUIRED
held_constant:
 - evaluation_cases
 - preprocessing
 - unrelated_inference_settings
structural_checks: []
quality_gates: []
performance_objective: REQUIRED
recovery_budget: REQUIRED_OR_NONE
measurements: {}
known_limitations: []
decision: not_evaluated
rollback_artifact: REQUIRED
```

Keep an experiment directory containing this contract, the patch, the generated architecture configuration, outputs, environment details, and the final decision. A failed experiment is useful when the failure is reproducible.

Image classification, detection, and segmentation

A proposed first structural experiment is to prune one compatible convolution-channel group or one internal FFN width while preserving the prediction interface. Use dependency analysis for branches and coupled parameters. DepGraph provides a concrete implementation and research basis for identifying such dependencies. It does not choose your product’s acceptable recall loss. **[R02]**, **[R03]**

Keep input resolution unchanged for that experiment. Measure per-class outcomes and boundary or small-object behavior where relevant. Then test resolution separately. Combining channel pruning and reduced resolution in the first run makes it difficult to identify which change removed fine detail.

For a segmentation model, a matching output shape is only a structural check. Examine the output alignment, thin structures, and task-specific errors. For a detector, preserve label mappings, box coordinate conventions, and postprocessing settings.

Image diffusion

Begin by locating the expensive stage. A condition encoder, denoiser, and latent decoder need different interventions. Try a compatible inference configuration or supported memory-scheduling option before committing to a new architecture. The diffusion chapter describes these as separate hypotheses. **[R34]**, **[R42]**, **[R48]**

For structural pruning, capture representative noisy states and conditioning, identify a valid dependency group, test a local prediction boundary, and recover using a compatible objective. Then evaluate completed samples. Noise prediction error alone does not establish composition, prompt adherence, or visual diversity.

Keep a fixed prompt and seed suite for comparison. Record failures such as omitted subjects, broken spatial relations, mask violations, or unwanted changes outside an edit region. Do not evaluate only whether a generated image looks attractive.

Video diffusion

First record temporal length, spatial dimensions, conditioning frame rate, playback frame rate, and decoder grouping. Keep these quantities separate. A proposal to generate fewer frames is a workload change. A proposal to decode the same latent sequence in groups is a scheduling change whose validity depends on the decoder.

Test identity, motion, flicker, occlusion, and scene changes independently. VBench is a reference for this multidimensional evaluation. A framewise score cannot establish all of those properties. **[R43]**

Temporal chunking deserves a dedicated test. State how context crosses chunk boundaries, whether normalization depends on the complete sequence, and whether a frame can attend to information outside the chunk. The image tiling counterexample is a useful warning, not a direct proof about your video architecture.

Language models

Separate prompt processing from autoregressive generation. Profile weights, intermediate allocations, and cache growth rather than publishing one memory number for all context lengths. A head-pruning change can affect query projections without shrinking a shared key/value cache. **[R46]**

Begin with a compatible internal FFN or block ablation. Verify attention masks, position handling, cache indices, and tied parameters. Evaluate the actual tasks, including structured output and longer contexts when they are supported. Short-prompt fluency is not evidence that the full operating envelope survived.

Keep decoding settings fixed when comparing model changes. A different sampling temperature can conceal or imitate a behavioral change in the network.

Audio models

State whether the model operates on waveforms, spectrograms, tokens, or another latent representation. Sample rate, frame hop, channel count, and causal context are part of the contract. Changing them can alter the task rather than merely making the same calculation cheaper.

Test boundary behavior when chunking. Preserve required context and inspect timing or phase behavior as appropriate. For generated speech, intelligibility, speaker identity, and prosody are separate acceptance dimensions. For separation, inspect leakage and artifacts rather than assuming clearer speech means better source recovery.

The supplied causal-convolution test is a starting point for reasoning about state. It is not an exportable replacement for a production separator or speech model.

Compare candidates without hiding tradeoffs

Maintain a table of quality, latency, memory, energy when measured, and recovery cost. Mark every unmeasured field as unmeasured. Do not fill a memory column with a parameter-count estimate simply because the table looks incomplete.

Prefer reporting tradeoffs to inventing one universal score. A candidate can be useful for batch generation and unsuitable for live interaction. A smaller checkpoint can be valuable for distribution even when execution is not faster. Keep the deployment objective attached to the decision.

17. Review, accept, or revert

Review the claim before the code

An optimization review starts with a claim such as "reduces warm batch-one latency for these input shapes on this backend." Check that the evidence actually measures that claim. A desktop MAC count does not prove phone latency. A masked-equivalence test does not prove unchanged task quality.

Use three review questions: Is the transformation implemented correctly? Does the task still meet its requirements? Does the intended deployment benefit exist? Evidence for one question does not answer the other two.

Review the transformation

Inspect dependent dimensions, residual interfaces, output contracts, state, and serialization. Confirm that the pruned architecture can be reconstructed from a clean process. New parameters require a deliberate optimizer policy during recovery. Reusing an optimizer that still references removed tensors is not a valid migration.

Check that code snippets marked as illustrative are not silently promoted to executed recipes. A custom module in this handbook is not a drop-in patch for a pretrained implementation merely because both contain a `Linear` layer.

Review the evaluation

Confirm that candidate selection and final testing are separated. Look for duplicate or correlated inputs across splits. Verify that model, metric, and preprocessing versions are recorded. Inspect examples where the candidate fails, not only aggregate improvements.

For generative outputs, verify the prompt list, conditioning, seed handling, and review procedure. For recovery, compare against an appropriate extra-training control. State when repeated seeds or independent review were not performed.

Review deployment evidence

Read the executed graph and backend trace, then inspect full-application behavior. Include model loading, repeated jobs, cancellation, and long inputs. Confirm that timing excludes or includes compilation and initialization intentionally rather than accidentally.

Treat missing measurements as open work. "Peak memory not measured" is an acceptable limitation in a prototype report. "Peak memory reduced" without a measurement is not.

Write the decision record

A useful final record names the candidate, identifies the measured improvement, states the accepted quality tradeoff, lists known limitations, and links the rollback artifact. It also identifies the supported environment. Avoid declaring a general winner when the experiment tested one device and one input shape.

A release decision may be **accept**, **accept for a restricted operating envelope**, **investigate**, or **revert**. These are engineering decisions against a written contract, not universal judgments about an architecture.

The endpoint is not the smallest possible network. It is a deployable model whose behavior and costs are understood well enough for the intended application.

Appendix A. Run the companion code

Package structure

The complete package contains the updated PDF and source, the beginner course, the original scripts, and recorded evidence. Checkpoints are generated teaching examples, not third-party pretrained weights.

EXAMPLE

```
Model_Refactoring_Complete/
  README.md
  CHANGELOG.md
  Model_Refactoring_for_Software_Engineers.pdf
  Model_Refactoring_for_Software_Engineers.md
  book.html
  requirements-lab.txt
  crash_course/
    README.md
    COURSE.md
    __init__.py
    models.py
    run.py
    test_contracts.py
    export_onnx.py
    requirements-cpu.txt
  android_example/
    README.md
    TinyNetRunner.kt
  code/
    lab.py
    chunking.py
    vision_attention_lab.py
  evidence/
    crash_course/
      report.json
      golden.json
      ... generated checkpoints ...
    crash_course_tests.txt
    run/
      report.json
      ... original records and checkpoints ...
    sources.json
    sources_current.json
  templates/
  build/
  archive/
  SHA256SUMS.txt
```

`crash_course/README.md` is the beginner entry point. `archive/` preserves the preceding manuscripts and original package metadata for provenance; it is not the current reading edition. The Kotlin and ONNX examples have explicit unexecuted status. No pretrained production model, dataset, APK, or font file is bundled.

The manuscript's `evidence/` references are relative to this package. Downloading only the Markdown manuscript provides the text and bibliography, not the referenced local experiment files.

Environment and installation

The recorded environment used Python 3.13.5 and PyTorch 2.10.0+cpu on Linux x86_64. The scripts use the Python standard library and PyTorch. Install an appropriate PyTorch build for your platform. The requirement file identifies the tested base package version; it is not a complete platform lockfile and does not guarantee an identical wheel on another operating system.

BASH

```
python -m venv .venv
# Activate the environment using your platform's command.
python -m pip install -r requirements-lab.txt
python -c "import torch; print(torch.__version__)"
```

Package installation requires a suitable package source and network or cached wheels. It is separate from running the examples. The recorded environment could not install the additional ONNX packages, so no ONNX or quantization result is included.

Run the crash course

The new beginner lab is independent of the original RefactorNet scripts. Run it from the package root after installing PyTorch:

BASH

```
python -m unittest crash_course.test_contracts -v
python -m crash_course.run --out runs/my_course_run
```

The included E04 record is immutable reference evidence. Reruns must use a fresh directory. The course records source and checkpoint hashes, the train/validation/test policy, ablation choices, supervised and distillation recovery, and a final selected-candidate evaluation.

Run the mechanism tests

From the extracted package directory:

BASH

```
python code/lab.py --self-test
python code/chunking.py
python code/vision_attention_lab.py --out vision_attention_rerun.json
```

A test confirming that naive normalization differs is expected to pass by detecting the mismatch. Do not change that test into an equivalence assertion to make an incorrect implementation appear acceptable.

Run the learning experiment

BASH

```
python code/lab.py --out new_run
```

The command refuses an existing output directory. Choose a new name for every run. It generates its own numerical dataset, trains the baseline, selects the block using validation loss, constructs the candidates, runs the fixed recovery recipes, and writes results and checkpoints.

Compare the new report with the supplied one. Expect hardware-dependent timing and possible numerical differences across releases or platforms. The supplied evidence is a record of a particular run, not a promise of bitwise equality on every machine. **[R10]**

Load a saved example

PYTHON

```
from pathlib import Path
import sys
import torch
sys.path.insert(0, "code")
from lab import load_model
model = load_model(Path("evidence/run/combined_kd.pt"))
with torch.inference_mode():
logits = model(torch.zeros(1, 16))
assert logits.shape == (1, 4)
```

The loader reconstructs the architecture from its versioned configuration and uses strict state-dictionary loading. Only load checkpoints whose origin you trust. The supplied loader is intended for these generated examples, not arbitrary objects or architecture files.

Extend rather than overwrite

Create a new experiment directory for a pretrained model. Retain its architecture revision, checkpoint digest, licenses, preprocessing, and evaluator configuration. Add model-specific tests before adapting a transformation. Keep this handbook’s recorded results unchanged so the difference between the example and your experiment remains visible.

Appendix B. Evidence directory

E01. RefactorNet 1.0 CPU learning lab

Origin: Original code in `code/lab.py`, executed in the preparation environment. **Primary records:** `evidence/run/report.json` and `evidence/run/timing.json`. **Supporting artifacts:** checkpoint files and `module_outputs.json`.

The report records the source digest, seed, environment, split sizes, training protocol, ablation ranking, retained channel indices, parameter counts, checkpoint digests, quality values, and limitations. Timing samples are included rather than only the aggregate values. This is one synthetic classification experiment, not a pretrained-model benchmark.

E02. Causal-convolution chunking checks

Origin: Original code in `code/chunking.py`. **Record:** `evidence/chunking_results.json`.

Three configurations test a single causal convolution with carried input context. The record contains kernel, dilation, chunk size, and maximum absolute error. No process-memory or production-audio measurement is claimed.

E03. Vision and attention mechanism tests

Origin: Original code in `code/vision_attention_lab.py`. **Record:** `evidence/vision_attention.json`.

The record includes the environment, source digest, parameter changes, numerical equivalence checks, and the intentionally nonequivalent normalization case. Randomly initialized modules and random inputs were used. The tests do not measure image-generation quality, temporal coherence, or device performance.

E04. TinyNet crash course and correctness suite

Origin: Original teaching code in `crash_course/models.py`, `crash_course/run.py`, and `crash_course/test_contracts.py`, executed for the expanded edition on 25 September 2026. **Primary records:** `evidence/crash_course/report.json`, `evidence/crash_course_tests.txt`, and `evidence/crash_course_stdout.json`.

The suite contains fifteen passing tests of tensor payloads, axis ordering, linear algebra, known parameter counts, forward interfaces, safe block copying, physical pruning against a masked reference, invalid-index handling, evaluation-mode behavior, gradient updates, frozen-teacher behavior, checkpoint reconstruction, hook cleanup, and finite gradients. It does not certify preserved production quality.

The learning run uses 2,048 synthetic training examples, 512 validation examples, and 512 held-out test examples. Validation selects the block removal and supervised-versus-distillation candidate. The final test is evaluated only after these choices. The teacher has 13,219 parameters; the combined student has 4,867. Supervised recovery, not distillation, wins the specified validation-loss selection in this single run. The report includes unrounded values, source hashes, checkpoint hashes, raw CPU timing samples, and explicit limitations.

`golden.json` contains a fixed feature input and expected logits for a future integration test. Its existence does not imply that the optional ONNX exporter or Kotlin wrapper has been executed. The preparation environment had no ONNX packages, and no Android device was used.

Appendix C. Technical glossary and lookup index

This appendix is an alphabetical lookup for terminology used in model refactoring, compression, deployment, and performance work. It also includes common terms that a software engineer is likely to encounter while applying the techniques in this handbook. Definitions are intentionally implementation-oriented. When behavior depends on a specific framework, runtime, hardware backend, or model revision, verify it against the pinned implementation used by the experiment.

Quick abbreviation index

Abbreviation	Meaning
BF16	Brain floating point, 16-bit floating-point format
CPU	Central processing unit
CUDA	NVIDIA parallel computing platform and programming model
EP	Execution Provider in ONNX Runtime terminology
FFN	Feed-forward network
FLOPs	Floating-point operations, usually used as an approximate compute measure
FP16	IEEE half-precision floating point
FP32	IEEE single-precision floating point
GELU	Gaussian Error Linear Unit
GEMM	General matrix multiplication
GPU	Graphics processing unit
INT4	4-bit integer representation
INT8	8-bit integer representation
KV cache	Key-value cache used by autoregressive attention models
MAC	Multiply-accumulate operation
NPU	Neural processing unit
ONNX	Open Neural Network Exchange

Abbreviation	Meaning
PTQ	Post-training quantization
QAT	Quantization-aware training
QNN	Qualcomm AI software stack and ONNX Runtime execution provider
Q/K/V	Query, key, and value projections in attention
RAM	Random-access memory
RSS	Resident Set Size
SIMD	Single Instruction, Multiple Data
SVD	Singular Value Decomposition
TFLite	TensorFlow Lite
VAE	Variational Autoencoder

A

Activation

A tensor produced while data passes through a model layer. Activations are temporary inference values rather than persistent learned parameters. Their size and lifetime can dominate peak memory even when model weights are small. See Chapters 2 and 9.

Activation function

A nonlinear function applied to a layer output. Common examples include ReLU, GELU, SiLU, and tanh. Activation choice can affect accuracy, numerical range, quantization behavior, and runtime operator support.

Activation memory

Memory occupied by live intermediate tensors during inference or training. It depends on tensor shape, dtype, execution order, retained skip connections, attention state, and runtime memory planning. See Chapters 2 and 9.

Adam and AdamW

Adaptive gradient-based optimizers widely used for training neural networks. AdamW separates weight decay from the gradient update. Optimizer state can require substantially more training memory than inference because moment estimates are stored for parameters.

Attention

A mechanism that computes interactions between sequence elements. Standard self-attention forms query, key, and value tensors and combines values according to similarity between queries and keys. See Chapter 7.

Attention head

One parallel attention subspace inside multi-head attention. Removing a head only saves meaningful compute when the associated projection dimensions or operations are physically reduced. Disabling a head while preserving the same dense matrices may provide little speedup. See Chapter 7.

Autograd

Automatic differentiation used to compute derivatives through recorded tensor operations. In PyTorch, backward populates gradients for participating leaf tensors; an optimizer performs the later parameter update. See Lesson 0.5. **[R60]**, **[R73]**

Autoregressive model

A model that generates each new token, frame, or sample conditioned on previous outputs. Autoregressive decoding often has sequential latency and may use persistent state such as a KV cache.

B

Backpropagation

The reverse computation of derivatives from a scalar objective through a computational graph. It computes gradients; it does not by itself choose the optimizer update or prove that a layer is safe to remove. See Lesson 0.5. **[R60]**

Batch size

The number of samples processed together. Larger batches can improve accelerator utilization but increase activation memory. Mobile inference commonly uses batch size 1.

BF16

A 16-bit floating-point format with the same exponent width as FP32 but fewer mantissa bits. It provides a wide dynamic range and is often useful for training or inference on hardware with native BF16 support.

Bottleneck block

A block that reduces a representation to a smaller width for expensive computation and then expands it again. Bottlenecks are common in efficient convolutional and transformer architectures.

Buffer reuse

A runtime optimization that reuses memory after an intermediate tensor is no longer live. Effective reuse can reduce peak memory without changing model mathematics. See Chapter 2.

C

Calibration data

Representative inputs used to estimate numerical ranges for post-training quantization. Poor calibration data can produce inaccurate scales and larger quality regressions. See Chapter 11.

Channel

A feature dimension in convolutional or other tensor representations. Structured channel pruning physically removes selected channels and dependent weights. See Chapter 6.

Checkpoint

A saved representation of learned model state, usually parameters and sometimes optimizer state, scheduler state, configuration, or metadata. A checkpoint alone may not fully specify inference behavior. See Chapter 1.

Chunking

Processing a long input as smaller windows while preserving required context. Chunking can bound activation memory and enable streaming, but correctness depends on overlap, receptive field, state handling, and boundary treatment. See Chapter 9.

Compute graph

The network of operations and tensor dependencies executed by a framework or runtime. Graph structure determines operator compatibility, scheduling opportunities, and tensor lifetimes.

Convolution

An operation that applies learned filters across spatial, temporal, or other structured dimensions. Convolutional cost depends on channels, kernel size, input resolution, groups, stride, and output size.

CPU

A general-purpose processor. Mobile CPUs usually contain heterogeneous cores with different performance and power characteristics. CPU inference performance depends heavily on vectorized kernels, memory bandwidth, thread scheduling, and sustained thermals.

Cross-entropy

A loss for comparing a predicted class distribution with a target. In the class-index PyTorch example, CrossEntropyLoss takes raw logits and integer class labels, not a softmax output supplied in place of logits. **[R65]**

CUDA

NVIDIA’s platform for executing parallel workloads on NVIDIA GPUs. A model that relies on custom CUDA operators usually requires modification before it can run on non-NVIDIA mobile hardware.

D

Data leakage

Information from evaluation examples entering training, preprocessing, or model-selection decisions in a way that invalidates the intended independent evaluation. Split policies must consider shared source videos, speakers, or other correlated units. **[R37]**

Decoder

The part of a model that transforms an internal representation into an output representation. Depending on the architecture, a decoder may generate tokens, spectrograms, images, latent tensors, masks, or other outputs.

Dequantization

Conversion of a quantized integer representation back to a floating-point representation or to another numerical domain used for computation. Frequent quantize-dequantize transitions can reduce expected performance gains.

Depthwise convolution

A grouped convolution in which each input channel is filtered independently, commonly followed by a pointwise convolution. It is a core building block of many mobile-friendly vision architectures.

Diffusion model

A generative model that learns to reverse a noise process, typically through repeated denoising steps. Image and video diffusion systems often contain a denoiser, text encoder, scheduler, and VAE. See Chapter 12.

Distillation

Training a smaller or otherwise constrained student model using signals from a teacher model. The student may imitate logits, hidden representations, features, predicted noise, audio features, or other task-specific targets. See Chapter 10.

Dtype

The numerical data type of a tensor, such as FP32, FP16, BF16, INT8, or INT4. Dtype affects storage, bandwidth, numerical behavior, supported kernels, and sometimes activation memory.

Dynamic shape

A tensor dimension that can vary at runtime. Dynamic shapes improve flexibility but can restrict some graph optimizations, static memory planning, or accelerator compilation strategies.

E

Embedding

A learned vector representation for an item such as a token, speaker, class, timestep, or condition. Embedding dimensions contribute to parameter size and sometimes to persistent state.

Encoder

The portion of a model that converts an input into an internal representation. Encoders are used in language, vision, audio, and multimodal systems.

Epoch

One traversal of the training dataset under a conventional training loop. A step is one optimizer update and is not the same quantity. The crash-course lab specifies a fixed number of sampled minibatch steps rather than epochs. **[R61]**, **[E04]**

Evaluation mode

The state set by model.eval() in PyTorch. It changes modules with distinct training/evaluation behavior but does not disable gradient recording. Use no_grad or inference_mode separately when appropriate. **[R73]**

Execution Provider (EP)

An ONNX Runtime backend that implements execution on a specific class of hardware or software stack, such as CPU, CUDA, or Qualcomm QNN. Unsupported operators may cause graph partitioning across providers. See Chapter 13.

F

Feed-forward network (FFN)

A per-position multilayer network used inside transformer blocks, commonly expanding the hidden dimension and then projecting it back down. FFNs can account for a large fraction of transformer parameters and compute. See Chapter 5.

Fine-tuning

Additional training of an existing model on selected data or objectives. Fine-tuning is often used after structural pruning or other surgery to recover quality.

FLOPs

An estimate of floating-point operation count. FLOPs can help compare arithmetic workload, but they do not directly predict latency, energy use, memory traffic, or accelerator utilization. See Chapter 2.

FP16

A 16-bit floating-point format. Compared with FP32, FP16 usually halves raw tensor storage and memory traffic. Whether it is faster depends on hardware and kernel support.

FP32

A 32-bit floating-point format commonly used as a reference precision for training and inference.

Fused operator

A runtime or compiler operation that combines multiple graph operations into one kernel or optimized execution region. Fusion can reduce memory traffic and launch overhead.

G

GELU

Gaussian Error Linear Unit. A smooth activation function common in transformer architectures. In conceptual form, GELU multiplies the input by the probability that a standard normal variable is less than that input. Implementations often use an exact or approximate formulation. When refactoring a model, preserve the formulation expected by the checkpoint unless equivalence has been validated.

GEMM

General matrix multiplication. Dense neural network layers, transformer projections, and many convolutions ultimately map to GEMM-like kernels. GEMM efficiency is strongly influenced by tensor dimensions, dtype, memory layout, and hardware support.

Golden fixture

A recorded input, expected output, and comparison tolerance used to test an implementation boundary. In this course, golden.json checks numeric inference integration, not performance or task coverage. **[E04]**

GPU

A processor optimized for highly parallel workloads. GPUs can accelerate many neural operations but are sensitive to kernel launch overhead, memory transfer, tensor layout, supported precision, and sustained thermal limits on mobile devices.

Gradient

The derivative of a training objective with respect to model parameters. Gradients are used by optimizers to update parameters. They are normally not required during inference.

Graph optimization

A transformation applied to the computational graph to improve execution while preserving intended semantics, unless explicitly documented as approximate. Examples include constant folding, operator fusion, and eliminating redundant nodes. See Chapter 13.

GroupNorm

Group Normalization. A normalization layer that divides channels into groups and normalizes within each group. It is widely used in diffusion and vision models because it does not depend on batch statistics in the same way as BatchNorm.

H

Head dimension

The dimensionality assigned to one attention head. In many implementations, hidden size equals the number of heads multiplied by head dimension.

Hidden size

The width of the main internal representation of a model. Reducing hidden size can reduce parameters, compute, and activations, but it usually requires coordinated changes throughout the architecture.

I

Inference

Running a trained model to produce outputs without performing parameter updates. Inference usually requires much less memory than training because gradients and optimizer state are absent.

INT4

A 4-bit integer representation used for aggressive quantization, most commonly for weights. INT4 cuts raw weight storage to one eighth of FP32, excluding scales and metadata. Runtime speedup depends on native kernel support and whether unpacking or dequantization is required.

INT8

An 8-bit integer representation commonly used for mobile and edge quantization. INT8 can reduce weight and activation bandwidth and may use integer matrix kernels when supported. See Chapter 11.

Intermediate tensor

A temporary value produced between model operations. Intermediate tensors are often what engineers mean when discussing activation memory.

K

Kernel

A low-level implementation of an operation executed by a CPU, GPU, NPU, or other backend. Two runtimes can execute the same model graph with very different performance because their kernels differ.

Knowledge distillation

See Distillation.

KV cache

Key-value cache. Persistent attention state stored during autoregressive generation so earlier tokens do not need to be recomputed at every decoding step. KV cache memory grows with sequence length, layer count, head configuration, batch size, and cache dtype.

L

Latency

Elapsed time required to complete an operation or inference request. Report latency with input shape, batch size, device, thread configuration, warmup procedure, and percentile or averaging method. See Chapters 1 and 2.

LayerNorm

Layer Normalization. A normalization operation over selected feature dimensions, widely used in transformer architectures.

Learning rate

An optimizer configuration controlling the scale of parameter updates. It is not a learned layer weight. A rate suitable for one training setup is not automatically appropriate after structural surgery. **[R61]**

Logit

An unnormalized class score emitted before a probability transformation such as softmax. A logit may be negative and multiple logits need not sum to one. TinyNet emits three logits for each example. **[R65]**, **[E04]**

LoRA

Low-Rank Adaptation. A parameter-efficient fine-tuning method that trains low-rank update matrices while leaving the original dense weights frozen. LoRA is primarily an adaptation technique, although low-rank ideas are also used for compression. See Chapter 8 and reference R30.

Loss function

A numerical objective used to compare predictions and training targets. Its gradient guides parameter updates. The training loss and the product acceptance metric may be different quantities. **[R61]**

Low-rank factorization

Approximating a large matrix with the product of smaller matrices. It can reduce parameters and sometimes compute when a sufficiently small rank preserves acceptable behavior. See Chapter 8.

M

MAC

Multiply-accumulate operation. A common unit for describing neural network arithmetic. MAC counts do not capture memory traffic, control flow, or device-specific efficiency.

Memory bandwidth

The rate at which data can be moved between memory and compute units. Neural inference can be bandwidth-bound even when the processor has unused arithmetic capacity.

Memory-mapped weights

Model parameters accessed through operating-system virtual memory mapping rather than copied into a separately allocated buffer. Mapped file size and resident memory are different measurements.

Minibatch

A subset of examples processed for one training update. This course samples 128 training examples for each optimizer step. Equal minibatch order and step counts help control a recovery comparison, but online teacher inference still adds compute. **[E04]**

Mixed precision

Using more than one numerical precision within a model, such as INT8 for robust encoder layers and FP16 for numerically sensitive output layers.

Model surgery

An engineering term for changing a trained model’s structure, such as removing blocks, pruning channels, replacing layers, or changing internal dimensions, followed by validation and often recovery training.

N

NPU

Neural processing unit. A specialized accelerator designed for machine-learning workloads. Actual model acceleration depends on supported operators, tensor shapes, precision, compiler behavior, and graph partitioning.

Normalization

An operation that rescales or recenters activations according to defined statistics. Examples include LayerNorm, GroupNorm, BatchNorm, and RMSNorm. Replacing one normalization method with another is generally not behavior-preserving without retraining or validation.

O

ONNX

Open Neural Network Exchange. A model representation format used to describe graphs of operators and tensors across frameworks and runtimes. ONNX itself is not the execution engine. See Chapter 13.

ONNX Runtime

An inference runtime that executes ONNX models through one or more Execution Providers. It performs graph optimization, memory planning, kernel dispatch, and backend partitioning.

Operator

A graph-level computation such as MatMul, Conv, Add, Softmax, LayerNormalization, or Reshape. Mobile deployment often fails or slows down when required operators are unsupported by the intended accelerator.

Operator fusion

Combining adjacent operations into a single optimized execution unit. Fusion can reduce intermediate memory traffic and dispatch overhead.

Optimizer

The component that updates parameters using gradients and its own configuration or state. Replacing model parameters during surgery requires ensuring that the optimizer references the new parameters. The course constructs a new optimizer after surgery. **[R61]**, **[E04]**

P

Parameter

A learned value stored in the model, such as a weight or bias. Parameters persist across inference calls and differ from temporary activations.

Parameter count

The number of learned scalar values in a model. Parameter count is useful for estimating dense weight storage but is not a complete measure of runtime memory or latency.

Peak memory

The maximum measured memory usage during a defined workload. Always state what is measured, such as process RSS, accelerator allocation, framework allocator peak, or tensor payload estimate.

Pointwise convolution

A convolution with a 1 × 1 spatial kernel, commonly used to mix channels after depthwise convolution.

Post-training quantization (PTQ)

Quantizing a trained model without fully retraining it. PTQ often uses calibration data to estimate activation ranges. See Chapter 11.

Pruning

Removing or suppressing model parameters or structures judged unnecessary for the target objective. See Chapters 4 through 7.

Prosody

Speech properties such as rhythm, stress, intonation, timing, and pitch contour. For speech models, prosody may be a separate quality dimension from intelligibility or speaker identity.

Q

QAT

Quantization-aware training. Training or fine-tuning a model while simulating quantized numerical behavior so the model can adapt to expected quantization error.

QNN

Qualcomm’s AI software stack and the name used by ONNX Runtime for its Qualcomm QNN Execution Provider. Supported behavior depends on device, SDK, operator set, and model configuration. See Chapter 13.

Quantization

Representing weights or activations with lower-precision numerical formats. Quantization can reduce storage and bandwidth, but quality and speed effects depend on calibration, operator support, hardware kernels, and where conversions occur. See Chapter 11.

Quantization scale

A numerical factor used to map between real-valued and integer representations in quantized inference. A quantized tensor is reconstructed from its integers using a scale and, in an affine scheme, a zero point. Scales have a granularity: one scale for the whole tensor (per-tensor), or one per output channel or chosen axis (per-channel, per-axis). Per-channel weight scales cost a scale vector and usually recover accuracy on layers whose output channels have very different ranges, which includes layers left behind by channel pruning. Per-tensor is the default in common tooling, so granularity is a choice that is easy to make by accident.

Query, key, value (Q/K/V)

The three projected representations used by attention. Queries are compared with keys to produce attention weights that combine values.

R

Receptive field

The region of an input that can influence a given output. In temporal or spatial chunking, the required overlap depends partly on the model's effective receptive field.

ReLU

Rectified Linear Unit. An activation function defined as the maximum of zero and the input. It is inexpensive and widely supported by inference runtimes.

Residual connection

A path that adds or otherwise combines an earlier representation with the output of a later block. Residual connections improve trainability but can extend activation lifetimes because earlier tensors must remain available until the merge operation.

RMSNorm

Root Mean Square Normalization. A normalization method that scales activations using their root mean square without subtracting the mean. It is used in several modern transformer families.

RSS

Resident Set Size. An operating-system measure of the physical memory currently resident for a process. RSS is useful but does not directly equal model activation memory.

Runtime

Software that loads and executes an exported model, such as ONNX Runtime, TensorFlow Lite, ExecuTorch, Core ML, or ncnn.

S

Scheduler

In diffusion systems, the algorithm that defines how denoising timesteps and updates are traversed during sampling. Changing the scheduler or number of steps can alter quality and runtime without modifying model weights. See Chapter 12.

Self-attention

Attention in which queries, keys, and values are derived from the same sequence.

SiLU

Sigmoid Linear Unit, also called the swish activation in a common parameterization. It computes the input multiplied by its sigmoid and is widely used in convolutional and diffusion architectures.

SIMD

Single Instruction, Multiple Data. A CPU execution model in which one instruction operates on multiple values in parallel. Efficient mobile inference libraries depend heavily on SIMD vectorization.

Softmax

A function that converts a vector of values into normalized positive weights that sum to one. It is commonly used to normalize attention scores or class logits.

Sparsity

The fraction or pattern of zero or removed values in a tensor. Unstructured sparsity does not guarantee speedup unless the runtime and hardware use sparse kernels effectively.

Speaker embedding

A compact vector intended to represent speaker identity or voice characteristics. In multi-speaker speech generation, the same acoustic model can often be conditioned on different speaker embeddings.

State dictionary

PyTorch mapping of parameter and registered-buffer names to their values. It does not replace the architecture definition and input contract needed to interpret those values. The course saves configuration alongside the state dictionary. **[R69]**

Static shape

A tensor shape fully known at export or compile time. Static shapes can enable stronger memory planning or accelerator compilation, at the cost of flexibility.

Structured pruning

Pruning that physically removes complete structures such as channels, heads, neurons, or blocks so tensor dimensions become smaller. This is often more useful for mobile deployment than merely inserting zeros. See Chapters 4 through 7.

SVD

Singular Value Decomposition. A matrix decomposition that can be used to construct low-rank approximations of dense matrices. See Chapter 8.

T

Tensor

A multidimensional array with a shape and dtype. Model parameters, inputs, outputs, activations, caches, and intermediate values are represented as tensors.

Tensor shape

The ordered dimensions of a tensor, such as `[batch, sequence, hidden]` or `[batch, channels, height, width]`. Shape is one of the strongest determinants of activation memory and operator cost.

Test set

Evaluation examples reserved for the final selected procedure. Repeatedly using their results to select architecture or training settings turns them into development feedback rather than an untouched final test. **[R37]**

TFLite

TensorFlow Lite, now commonly associated with Google's LiteRT ecosystem, is a mobile and edge inference stack for TensorFlow-compatible models and related deployment workflows.

Throughput

The amount of work completed per unit time, such as images per second, tokens per second, or audio seconds processed per second. Throughput and single-request latency are different metrics.

Token

A discrete unit processed by many language and multimodal models. A token may represent a word fragment, character, code unit, image patch index, or another learned symbol depending on the tokenizer and model.

Tokenizer

Software that maps raw input text or another discrete representation to token IDs expected by a model. A checkpoint and tokenizer must be kept compatible.

Training set

Examples used to compute the objective and update model parameters. In the course, these are generated separately from validation and test data; production tasks require their own representative, authorized data. **[R37]**, **[E04]**

U

Unstructured pruning

Pruning individual scalar weights without changing the physical tensor shape. It can create many zeros but may not reduce latency or memory unless sparse storage and sparse kernels are used.

Upsampling

Increasing spatial, temporal, or latent resolution. Upsampling stages can create large activations, especially in image, video, and audio decoders.

V

VAE

Variational Autoencoder. In latent diffusion systems, a VAE commonly encodes images or video frames into lower-dimensional latent tensors and decodes generated latents back to pixel space.

Validation loss

The selected objective measured on validation examples without training on those examples. It can guide ablation and recovery choices but is not an untouched final-test result after repeated selection. **[R37]**, **[E04]**

Validation set

A fixed collection of representative examples used to measure whether a model change preserves required behavior. It should not be optimized against so aggressively that it stops representing unseen production cases. See Chapters 1 and 3.

Vectorization

Organizing computation so a processor can operate on several values per instruction. Tensor dimensions aligned with efficient vector widths can materially affect CPU performance.

Vocoder

A model or signal-processing component that converts an acoustic representation such as a mel spectrogram into a waveform. In TTS pipelines, the vocoder can be a major source of latency and quality sensitivity.

W

Weight

A learned parameter used by an operation such as a linear projection or convolution. Weight storage is usually easy to estimate from parameter count and dtype, but runtime memory includes more than weights.

Weight-only quantization

Quantization in which model weights use a reduced precision such as INT8 or INT4 while activations remain in floating point or another format. It is common for large language models and some matrix-heavy architectures.

Working set

The memory actively required during a workload, including resident parameters, live activations, caches, temporary workspace, runtime allocations, and relevant application buffers.

X

XNNPACK

A library of optimized neural network operators for CPU execution, especially on ARM and mobile-class processors. Performance depends on operator, shape, dtype, threading, and integration in the chosen runtime.

Practical lookup by engineering problem

If you are investigating…	Start with these terms
Model file is too large	Parameter count, dtype, quantization, weight-only quantization, low-rank factorization
Peak RAM is too high	Activation memory, tensor shape, buffer reuse, chunking, receptive field, working set

If you are investigating…	Start with these terms
Transformer is too slow	FFN, attention head, hidden size, GEMM, Q/K/V, KV cache, structured pruning
CNN is too slow	Channel, convolution, depthwise convolution, pointwise convolution, structured pruning
Diffusion is too slow	Diffusion model, scheduler, VAE, resolution, quantization, distillation
Android NPU is not accelerating	Execution Provider, operator, graph partitioning, QNN, dynamic shape, fused operator
INT8 quality is poor	Calibration data, quantization scale, PTQ, QAT, mixed precision
TTS quality changed after compression	Prosody, speaker embedding, vocoder, validation set, mixed precision
Long input causes OOM	Chunking, receptive field, activation memory, dynamic shape, KV cache
FLOPs fell but latency did not	Kernel, memory bandwidth, operator fusion, vectorization, execution provider

Appendix D. HDemucs: a concrete refactoring experiment

Evidence status: proposed experiment plan. This appendix applies the handbook to the original hybrid Demucs family. No HDemucs checkpoint was pruned, retrained, exported, or benchmarked while preparing this PDF. The observations below come from the referenced implementation. The proposed measurements and acceptance decisions are work for the experimenter.

D.1 Identify the model before changing it

HDemucs and HTDemucs are different architectures. The original hybrid design combines waveform and spectrogram processing. The repository also distributes a later hybrid Transformer family. A command that loads the default checkpoint is therefore not an adequate specification of an HDemucs experiment. The published model list identifies `hdemucs_mmi` as a hybrid baseline, while the inspected loader defaults to `htdemucs`. **[R51]**, **[R52]**, **[R53]**

Record the repository commit, installed package versions, checkpoint digest, model class, source names, sample rate, channel count, and every member of a model ensemble. The following inspection template is not a tested environment or a pinned installation recipe. Run it only after installing your chosen, trusted revision and recording its dependencies.

PYTHON

```python
# Proposed inspection template; not executed for this appendix.
from demucs.apply import BagOfModels
from demucs.hdemucs import HDemucs
from demucs.pretrained import get_model
loaded = get_model("hdemucs_mmi")
members = list(loaded.models) if isinstance(
loaded, BagOfModels
) else [loaded]
for index, model in enumerate(members):
if not isinstance(model, HDemucs):
raise TypeError(f"Unexpected model: {type(model).__name__}")
print({
"member": index,
"class": type(model).__name__,
"sources": model.sources,
"sample_rate": model.samplerate,
"audio_channels": model.audio_channels,
"parameters": sum(p.numel() for p in model.parameters()),
})
```

A successful inspection verifies identity and metadata. It does not demonstrate separation quality, runtime compatibility, or mobile feasibility. Inspect any ensemble rather than silently replacing it with its first member. The loader and inference wrapper explicitly support model bags. **[R53]**, **[R54]**

D.2 Do not apply the residual-block recipe to an encoder stage

The inspected HDemucs implementation has frequency and time encoders, corresponding decoders, skip connections, and a point where the branches meet. Stages change dimensions and resolution. Deleting `encoder[2]` is not equivalent to removing a shape-preserving residual block. The decoder and branch alignment can become invalid. The source also warns that changing the FFT size requires coordinated shape calculations. **[R55]**

For an initial experiment, leave the input contract, stage count, branch alignment, FFT settings, source ordering, and output reconstruction unchanged. Profile the actual checkpoint before choosing a transformation.

Inside supported stages, the implementation can contain `DConv` modules. Their internal residual steps preserve the external channel count and are a more localized place to investigate ablation. The inspected forward loop adds each step's output to its input. Replacing the step with `Identity` would therefore double that input, not remove the residual contribution. Removing a step requires rebuilding the step list or returning a zero residual, followed by verification. **[R56]**

Proposed selection procedure: inspect which residual modules exist, measure their costs, ablate one compatible step on a copied model, and evaluate that candidate before recovery training. Keep the original checkpoint immutable. Reconstruction must encode the changed structure, not just save a state dictionary against the original architecture.

D.3 Separate inference settings from architecture changes

Start with the existing inference wrapper. It exposes splitting, segment duration, overlap, random shifts, and worker configuration. In the inspected code, `shifts=0` disables random-shift augmentation. Reducing repeated evaluations and reducing segment length are different experiments. Neither operation changes the stored parameter count. **[R54]**

The following order is a proposed investigation sequence, not a measured ranking. Start from your pinned baseline and change one variable per comparison.

Experiment	Change under test	Keep fixed	Required check
A. Segment length	Test a shorter supported segment	Weights, overlap, shifts, threads	Peak memory, boundary artifacts, separation quality
B. Shift count	Reduce augmentation passes when enabled	Segment length, weights, overlap	Total runtime and quality
C. Residual step	Remove one compatible internal step	Stage interfaces and preprocessing	Shape validity and quality before and after recovery
D. Internal width	Rebuild one dependency group with fewer channels	Source contract and evaluation data	Coupled dimensions, quality, deployed latency
E. Precision	Quantize a supported subgraph	Accepted architecture and inputs	Numerical parity, actual kernels, audio artifacts

Do not interpret the `--two-stems` output option as a two-source architecture. In the inspected command-line implementation, source selection occurs after separation. Similarly, `--int24` controls the saved audio representation, not neural-network weight precision. These flags do not establish a smaller inference model. **[R57]**

D.4 Account for memory outside the network

The splitting implementation allocates an output tensor for the full requested duration. Shorter segments can reduce the size of an individual model evaluation without making the complete application constant-memory. **[R54]**

For a hypothetical four-source, two-channel output lasting 20 minutes at 44,100 samples per second, the FP32 payload alone is:

EXAMPLE

```
4 sources * 2 channels * 1,200 seconds * 44,100 samples/second
          * 4 bytes/sample
= 1,693,440,000 bytes
= approximately 1.69 GB, or 1.58 GiB
```

This is tensor-size arithmetic, not a measured HDemucs allocation. It excludes the mixture, weights, activations, workspaces, overlap buffers, and application memory.

For a proposed bounded-memory application, read bounded input windows, keep only the overlap region still needed for blending, and write finalized output progressively. Validate its waveform output against the reference wrapper. Preserve normalization, padding, weighting, and boundary conventions. Chunked processing is not automatically equivalent to causal streaming or full-track inference.

D.5 Treat resolution changes as new model experiments

Do not relabel 44.1 kHz samples as 16 kHz input. Do not reduce `nfft` and assume the existing checkpoint still represents the same learned operation. A lower-rate student requires an explicit resampling policy, compatible architecture, recovery or new training, and a fresh evaluation contract. Read the actual checkpoint metadata rather than assuming constructor defaults describe every release. **[R55]**

For a mobile export experiment, test the spectral frontend and reconstruction separately. The repository's spectral wrapper uses a Hann window, centered transforms, normalization, and complex-valued STFT output. A native DSP replacement must match the selected implementation's conventions before it replaces those operations in a larger pipeline. **[R58]**

Proposed verification: compare frontend tensors on silence, an impulse, tones, short clips, and representative mixtures. Compare reconstruction lengths and numerical error before evaluating end-to-end audio quality. Use declared tolerances and preserve the reference implementation for rollback.

D.6 Design an evaluation that can reject the optimization

The original research concerns music-source separation. Its evidence does not establish accurate dialogue, music, and sound-effect separation for cartoons or films. Define the intended sources and evaluate that domain independently. **[R51]**

For this proposed study, partition evaluation data by complete recording or source identity before extracting segments. Keep recovery-training data separate from the final test set. Where isolated references exist, report a specified separation metric per source and describe its implementation, aggregation, and handling of silent references. Without clean references, do not fabricate an objective reference-based quality score.

Add blinded listening comparisons for leakage, missing content, transients, stereo changes, and boundary clicks. Record failures as well as averages. Pair the quality report with the target-device configuration, complete-job time, warm inference latency, and a precisely named peak memory measurement. Compare candidates under the same workload and operating conditions.

Acceptance decision: retain a change only when it satisfies the previously declared quality and resource requirements. A successful export, a smaller checkpoint, or a passing shape test is insufficient. No compression ratio or Android speedup is asserted by this appendix.

References

Primary sources are listed below with the claims they support. Official documentation can change after the recorded access date. Pin the actual dependency and architecture revisions before reproducing a production experiment. A research reference describes the authors' evaluated conditions, not a guarantee for another checkpoint.

R01. Pruning Tutorial

PyTorch contributors. Official tutorial. Accessed 2026-09-24.

Supports: Mask-based pruning and removal of reparameterization; mutable tutorial.

Primary source

R02. Torch-Pruning

Gongfan Fang and Torch-Pruning contributors. Primary implementation. Accessed 2026-09-24.

Supports: Dependency-aware structural pruning; pin a commit before using the package.

Primary source

R03. DepGraph: Towards Any Structural Pruning

Fang et al. Research paper, 2023. Accessed 2026-09-24.

Supports: Coupled parameter groups across neural network architectures.

Primary source

R04. Knowledge Distillation Tutorial

PyTorch contributors. Official tutorial. Accessed 2026-09-24.

Supports: Teacher-student training examples; tutorial is not a guarantee of recovery.

Primary source

R05. Distilling the Knowledge in a Neural Network

Hinton, Vinyals, and Dean. Research paper, 2015. Accessed 2026-09-24.

Supports: Soft targets and temperature-based distillation.

Primary source

R06. Reducing Transformer Depth on Demand with Structured Dropout

Fan, Grave, and Joulin. Research paper, 2019. Accessed 2026-09-24.

Supports: LayerDrop is a training method, not evidence for arbitrary layer deletion.

Primary source

R07. Pruning Convolutional Neural Networks for Resource Efficient Inference

Molchanov et al. Research paper, ICLR 2017. Accessed 2026-09-24.

Supports: First-order sensitivity and iterative pruning in studied CNN tasks.

Primary source

R08. NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications

Yang et al. Research paper, ECCV 2018. Accessed 2026-09-24.

Supports: Direct platform measurements instead of relying only on operation counts.

Primary source

R09. torch.profiler

PyTorch contributors. Official documentation, 2.10. Accessed 2026-09-24.

Supports: Operator profiling, shapes, memory events and instrumentation overhead.

Primary source

R10. Reproducibility

PyTorch contributors. Official documentation, 2.10. Accessed 2026-09-24.

Supports: Determinism controls and limits across platforms and versions.

Primary source

R11. torch.linalg.svd

PyTorch contributors. Official documentation, 2.10. Accessed 2026-09-24.

Supports: Singular-value decomposition API and numerical caveats.

Primary source

R12. inference_mode

PyTorch contributors. Official documentation, 2.10. Accessed 2026-09-24.

Supports: Inference-mode behavior; evaluation mode remains separate.

Primary source

R13. torch.onnx

PyTorch contributors. Official documentation, 2.10. Accessed 2026-09-24.

Supports: Export options, shape constraints, verification and external weights.

Primary source

R14. Quantize ONNX models

ONNX Runtime contributors. Official documentation. Accessed 2026-09-24.

Supports: Static/dynamic quantization, QDQ, calibration and operator-specific INT4 support.

Primary source

R15. Graph optimizations

ONNX Runtime contributors. Official documentation. Accessed 2026-09-24.

Supports: Graph optimization levels and semantics-preserving rewrites.

Primary source

R16. Deploy on mobile

ONNX Runtime contributors. Official documentation. Accessed 2026-09-24.

Supports: Mobile execution-provider selection and device benchmarking.

Primary source

R17. Execution Providers

ONNX Runtime contributors. Official documentation. Accessed 2026-09-24.

Supports: Backend selection and allocation of supported subgraphs.

Primary source

R18. Qualcomm QNN Execution Provider

ONNX Runtime contributors. Official documentation. Accessed 2026-09-24.

Supports: Supported operators, static shapes and backend-specific quantization requirements.

Primary source

R19. Reduced operator config file

ONNX Runtime contributors. Official documentation. Accessed 2026-09-24.

Supports: Selective runtime builds using required operator/type lists.

Primary source

R20. ExecuTorch documentation

PyTorch contributors. Official documentation. Accessed 2026-09-24.

Supports: Edge deployment and supported hardware backends; mutable stable documentation.

Primary source

R21. Overview of memory management

Android Developers. Official documentation. Accessed 2026-09-24.

Supports: Managed heap, memory pressure and proportional set size.

Primary source

R22. dumpsys

Android Developers. Official documentation. Accessed 2026-09-24.

Supports: Inspecting Android process memory; snapshots are not continuous peak traces.

Primary source

R23. Thermal API

Android Developers. Official documentation. Accessed 2026-09-24.

Supports: Thermal status and headroom signals for adapting workload.

Primary source

R24. Support for long-running workers

Android Developers. Official documentation. Accessed 2026-09-24.

Supports: Long-running work, foreground execution and platform-version constraints.

Primary source

R25. Are Sixteen Heads Really Better than One?

Michel, Levy, and Neubig. Research paper, 2019. Accessed 2026-09-24.

Supports: Attention-head importance and pruning in evaluated Transformer tasks.

Primary source

R26. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness

Dao et al. Research paper, 2022. Accessed 2026-09-24.

Supports: Exact attention with an IO-aware implementation; not an Android guarantee.

Primary source

R27. scaled_dot_product_attention

PyTorch contributors. Official documentation, 2.10. Accessed 2026-09-24.

Supports: Kernel dispatch, dropout behavior and supported attention implementations.

Primary source

R28. GroupNorm

PyTorch contributors. Official documentation, 2.10. Accessed 2026-09-24.

Supports: Groupwise input statistics, including evaluation-time behavior.

Primary source

R29. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications

Howard et al. Research paper, 2017. Accessed 2026-09-24.

Supports: Depthwise separable convolution and width/resolution tradeoffs.

Primary source

R30. LoRA: Low-Rank Adaptation of Large Language Models

Hu et al. Research paper, 2021. Accessed 2026-09-24.

Supports: Low-rank updates to a retained pretrained weight matrix.

Primary source

R31. High-Resolution Image Synthesis with Latent Diffusion Models

Rombach et al. Research paper, CVPR 2022. Accessed 2026-09-24.

Supports: Latent autoencoder and denoising pipeline architecture.

Primary source

R32. Scalable Diffusion Models with Transformers

Peebles and Xie. Research paper, 2023 version. Accessed 2026-09-24.

Supports: Diffusion Transformer depth, width and latent patch-token architecture.

Primary source

R33. P.808: Subjective evaluation of speech quality with a crowdsourcing approach

ITU-T. Recommendation. Accessed 2026-09-24.

Supports: Speech listening-study methodology; not a universal voice-identity metric.

Primary source

R34. Reduce memory usage

Hugging Face Diffusers contributors. Official documentation. Accessed 2026-09-24.

Supports: VAE tiling/slicing, offloading, limitations and tradeoffs.

Primary source

R35. Stable Video Diffusion

Hugging Face Diffusers contributors. Official documentation. Accessed 2026-09-24.

Supports: Feed-forward chunking, frame decoding and possible flicker.

Primary source

R36. torch.utils.checkpoint

PyTorch contributors. Official documentation, 2.10. Accessed 2026-09-24.

Supports: Activation checkpointing exchanges training memory for recomputation.

Primary source

R37. Common pitfalls and recommended practices

scikit-learn contributors. Official documentation. Accessed 2026-09-24.

Supports: Data leakage and separation of fitting from evaluation.

Primary source

R38. Progressive Distillation for Fast Sampling of Diffusion Models

Salimans and Ho. Research paper, ICLR 2022. Accessed 2026-09-24.

Supports: Learning to replace a multi-step sampler with fewer learned steps.

Primary source

R39. Consistency Models

Song et al. Research paper, 2023. Accessed 2026-09-24.

Supports: Consistency training and distillation for few-step generation.

Primary source

R40. Q-Diffusion: Quantizing Diffusion Models

Li et al. Research paper, 2023. Accessed 2026-09-24.

Supports: Diffusion-specific calibration and quantization of studied denoisers.

Primary source

R41. Structural Pruning for Diffusion Models

Fang, Ma, and Wang. Research paper, NeurIPS 2023. Accessed 2026-09-24.

Supports: Time-aware importance selection and structural diffusion pruning.

Primary source

R42. DPMSolverMultistepScheduler

Hugging Face Diffusers contributors. Official documentation. Accessed 2026-09-24.

Supports: Prediction types, schedules and solver configuration.

Primary source

R43. VBench: Comprehensive Benchmark Suite for Video Generative Models

Huang et al. Research paper. Accessed 2026-09-24.

Supports: Separate temporal, spatial and semantic video-evaluation dimensions.

Primary source

R44. Reproducibility

Hugging Face Diffusers contributors. Official documentation. Accessed 2026-09-24.

Supports: Random-generator state and limits of repeated seeded generation.

Primary source

R45. On Aliased Resizing and Surprising Subtleties in GAN Evaluation

Parmar, Zhang, and Zhu. Research paper, CVPR 2022. Accessed 2026-09-24.

Supports: Resizing and compression can alter FID comparisons.

Primary source

R46. Cache strategies

Hugging Face Transformers contributors. Official documentation. Accessed 2026-09-24.

Supports: Static, dynamic, offloaded and quantized cache tradeoffs.

Primary source

R47. Conv2d

PyTorch contributors. Official documentation, 2.10. Accessed 2026-09-24.

Supports: Convolution shape and group constraints.

Primary source

R48. Accelerate inference

Hugging Face Diffusers contributors. Official documentation. Accessed 2026-09-24.

Supports: Precision, compilation and memory-efficient attention guidance.

Primary source

R49. ONNX Intermediate Representation Specification

ONNX contributors. Official specification. Accessed 2026-09-24.

Supports: Graph representation, operators and tensor types; not runtime kernel guarantees.

Primary source

R50. Tensor Views

PyTorch contributors. Official documentation, 2.10. Accessed 2026-09-24.

Supports: Storage sharing and non-contiguous views.

Primary source

R51. Hybrid Spectrogram and Waveform Source Separation

Alexandre Defossez. Original research paper, arXiv:2111.03600. Accessed 2026-09-24.

Supports: The original hybrid waveform and spectrogram architecture and the music-separation scope of its evaluation. Results are not reproduced in this appendix.

Primary source

R52. Demucs model descriptions

Meta / Facebook Research. Official repository README. Accessed 2026-09-24.

Supports: Distinguishing `hdemucs_mmi` from the `htdemucs` family in the published model list.

Primary source

R53. Demucs pretrained model loader

Meta / Facebook Research. Official `demucs/pretrained.py` implementation, inspected `main` snapshot. Accessed 2026-09-24.

Supports: Model loading, model-bag resolution, source labels, and the inspected default model name. Pin the selected revision before execution.

Primary source

R54. Demucs inference wrapper

Meta / Facebook Research. Official `demucs/apply.py` implementation, inspected `main` snapshot. Accessed 2026-09-24.

Supports: Segment splitting, overlap weighting, shift augmentation, ensemble evaluation, and full-duration output allocation. Pin the selected revision before execution.

Primary source

R55. HDemucs architecture implementation

Meta / Facebook Research. Official `demucs/hdemucs.py` implementation, inspected `main` snapshot. Accessed 2026-09-24.

Supports: Coupled time and frequency branches, encoder and decoder interfaces, FFT-dependent shapes, and reconstruction behavior. Constructor defaults are not a substitute for checkpoint metadata.

Primary source

R56. DConv residual implementation

Meta / Facebook Research. Official `demucs/demucs.py` implementation, inspected `main` snapshot. Accessed 2026-09-24.

Supports: Shape-preserving internal residual steps and additive forward behavior. No pruning-quality result is claimed.

Primary source

R57. Demucs command-line separation implementation

Meta / Facebook Research. Official `demucs/separate.py` implementation, inspected `main` snapshot. Accessed 2026-09-24.

Supports: Source-output selection after separation and the distinction between saved-audio precision and model precision.

Primary source

R58. Demucs spectral transform wrapper

Meta / Facebook Research. Official `demucs/spec.py` implementation, inspected `main` snapshot. Accessed 2026-09-24.

Supports: STFT and inverse-STFT windowing, normalization, centering, and complex representation conventions.

Primary source

R59. Tensors

PyTorch contributors. Official tutorial. Accessed 2026-09-25.

Supports: Tensor properties, basic operations, data layout, and device concepts.

Primary source

R60. Automatic differentiation with torch.autograd

PyTorch contributors. Official tutorial. Accessed 2026-09-25.

Supports: Gradient recording and backward computation.

Primary source

R61. Optimizing model parameters

PyTorch contributors. Official tutorial. Accessed 2026-09-25.

Supports: Training objectives, gradients, optimizers, and update order.

Primary source

R62. Module

PyTorch contributors. Versioned PyTorch 2.10 API reference. Accessed 2026-09-25.

Supports: Module registration, forward calls, parameter traversal, and hooks.

Primary source

R63. Linear

PyTorch contributors. Versioned PyTorch 2.10 API reference. Accessed 2026-09-25.

Supports: Affine computation, weight orientation, and feature dimensions.

Primary source

R64. GELU

PyTorch contributors. Versioned PyTorch 2.10 API reference. Accessed 2026-09-25.

Supports: GELU definition and approximation option.

Primary source

R65. CrossEntropyLoss

PyTorch contributors. Versioned PyTorch 2.10 API reference. Accessed 2026-09-25.

Supports: Raw-logit input and target conventions for classification.

Primary source

R66. KLDivLoss

PyTorch contributors. Versioned PyTorch 2.10 API reference. Accessed 2026-09-25.

Supports: Log-probability input convention and batchmean reduction.

Primary source

R67. Virtual environments and packages

Python Software Foundation. Python 3.13 tutorial. Accessed 2026-09-25.

Supports: venv and environment-local package installation.

Primary source

R68. Classes

Python Software Foundation. Python 3.13 tutorial. Accessed 2026-09-25.

Supports: Class instances, methods, assignment, and object aliasing.

Primary source

R69. Save and load the model

PyTorch contributors. Official tutorial. Accessed 2026-09-25.

Supports: State dictionaries and architecture reconstruction.

Primary source

R70. LayerNorm

PyTorch contributors. Versioned PyTorch 2.10 API reference. Accessed 2026-09-25.

Supports: Normalization dimensions and computation.

Primary source

R71. Neural Networks API

Android Developers. Official NDK documentation. Accessed 2026-09-25.

Supports: NNAPI deprecation beginning with Android 15.

Primary source

R72. Java inference API

ONNX Runtime contributors. Official documentation. Accessed 2026-09-25.

Supports: Java sessions, tensors, inference, and resource cleanup.

Primary source

R73. Autograd mechanics

PyTorch contributors. Versioned PyTorch 2.10 documentation. Accessed 2026-09-25.

Supports: Gradient modes, no_grad, inference mode, and separation from evaluation mode.

Primary source