

Activation checkpointing (AC) saves memory by discarding a block's activations in the forward pass and recomputing them before backward. **Selective AC** (SAC) goes further, recomputing only the cheap ops and saving the expensive ones. Although PyTorch core already provides a SAC API in `create_selective_checkpoint_contexts`, it has some fundamental limitations that motivate a rewrite: per-op-instance policies can be difficult to express, plain `autograd.Function` is not well supported, some outputs can be saved unnecessarily. **torch_remat** is a new activation checkpointing library built on top of `torch.utils.checkpoint` that provides improved expressivity, saves only what is needed, and more!



github.com/meta-pytorch/remat

↩️ Comparison	create_selective_checkpoint_contexts	torch_remat
Where the policy lives	in a policy function, passed to AC via context_fn.	inline in code; wrap Python functions inside AC'd block w/ <code>remat.region</code>
Per-op instance handling	per-instance policies are hard to express, hard for policy to tell op instances apart; either save all mms or recompute all mms	per-instance policies are easy to express, wrap the regions you want potentially saved
<code>autograd.Function</code> handling	need to wrap in a custom op for proper handling	works as-is, e.g. you can just wrap <code>Fn.apply</code> with <code>remat.region</code>
What is skipped during recompute	aten ops: All code is rerun, except saved aten ops	regions of code: anything wrapped in <code>remat.region</code> is skipped if <code>recompute=False</code>
What persists after forward	every saved op's outputs	what is saved for backward in the region AND any outputs needed during recompute (see "what stays resident after forward")
Offloading / Saved Tensor Hooks handling	compose with native saved tensor hooks	use <code>torch_remat</code> 's saved tensor hooks, which can provide extra metadata to inform offloading policies

<> Basic usage

```
import torch_remat as remat
class Block(nn.Module):
    def forward(self, h):
        x = self.attention_norm(h)
        attn = remat.region(
            self.attention, "attention",
            recompute=False, # skipped during recompute
        )(x)
        remat.recompute_needs_tensor(attn)
        h = h + attn
        # unwrapped code is always recomputed
        return h + self.ffn(self.ffn_norm(h))
# one checkpoint per transformer block
out = remat.checkpoint(region_name="layers.0")(blk)(h)
```

1. Call `remat.checkpoint` instead of `torch.utils.checkpoint`. `remat.checkpoint` will call into `utils.checkpoint` underneath

2. Wrap regions of code with `remat.region`; this allows them to be conditionally skipped

3. Unwrapped code that consume the output of `remat.regions` (with `recompute=False`) are always recomputed and thus should have their inputs marked with `recompute_needs_tensor`. This triggers the `recompute=False` region to actually persist its output. If we do not persist an output which was actually needed, an error is thrown.

»» Replay skips recompute=False regions

Forward

norm

attention

+

norm

MoE

+

Replay

norm

attention skipped

+

norm

MoE

+

☐ recompute (default)

☒ recompute=False

☐ not re-run

Before backward, the block is replayed with `recompute=False` regions skipped. A skipped region returns a placeholder with the right shape and dtype; running a kernel on it is an error. Its real output is kept only when later recomputed code needs it.

📁 What stays resident after forward?

x

→

region A
bwd needs p

→ y

region B
bwd needs q

→ z

region C

→

output

	A	B	Stays resident	Recomputed
1	recompute=False	recompute=False	x, z, p, q	nothing
2	recompute=False	recompute=True	x, y, p	z, q
3	recompute=True	recompute=False	x, z, q	y, p
4	recompute=True	recompute=True	x	y, z, p, q

Notes: in row 1, the output of region A (y) was not saved since its not needed for recompute, its only consumer, region B is itself recompute=False. p and q are activations produced by the A/B regions (not inputs to the regions).

🚀 Getting started

```
pip install git+https://github.com/meta-pytorch/remat
```

Requires PyTorch 2.10+ and Python 3.10+. BSD-3-Clause.

🔍 Inspecting saves and memory

Check which regions were saved or recomputed

```
with remat.collect_trace() as trace:
    output = model(inputs)
    print(trace.format())
```

```
torch_remat trace
scope [test_flag]
sin: save
cos: recompute
```

Generate a breakdown of each saved tensor and its size:

```
if not remat.is_recomputing():
    remat.print_current_memory_report()
```

```
layers.0: 28 B resident in 2 storage(s)
layers.0::attn.softmax: 28 B
12 B lse (3,) float32
16 B probs (4,) float32
```

- `print_saved_tensors_report(loss)` covers live `remat` regions plus reachable ordinary `autograd` saves.
- Attach `remat.oom_observer` to print this report automatically on OOM.

🎲 Handling RNG and other side effectful code

You should set `recompute_state_hooks=` if the block uses randomness (e.g. dropout): a skipped `recompute=False` region doesn't advance the RNG during replay. `remat` snapshots state on the forward and restores it at checkpoint entry and after each skipped region:

```
class CudaRNGStateHook:
    def snapshot(self):
        return torch.cuda.get_rng_state()

    def restore(self, state):
        torch.cuda.set_rng_state(state)

out = remat.checkpoint(
    region_name="layers.0",
    recompute_state_hooks=(CudaRNGStateHook(),),
)(block)(h)
```

`remat` sets `preserve_rng_state=False` when it calls into `torch.utils.checkpoint`

↕ Offloading with saved-tensor hooks

Offload what `remat` keeps (e.g. to CPU) with a pack/unpack pair:

```
def pack(t):
    kind = remat.current_saved_tensor_info().kind # why remat keeps t
    return t.detach().to("cpu"), t.device

def unpack(saved):
    t, device = saved
    return t.to(device)

with remat.saved_tensors_hooks(pack, unpack):
    out = remat.checkpoint(region_name="layers.0")(blk)(x)
```

- Hooks see every tensor `remat` keeps, and `remat.current_saved_tensor_info()` tells you why it's kept: `CHECKPOINT_INPUT`, `BACKWARD`, or `SAVE_OUTPUT`.
- To offload checkpoint inputs too, pass the pair to `checkpoint(saved_tensors_hooks=...)`.
- `capture_context` records where each tensor was produced (e.g. which layer), even if the pack runs later.