

The code for FAdam is publicly available on GitHub and is included as supplementary material.

```
from typing import Optional, Tuple
import torch
from torch.optim.optimizer import Optimizer
from torchtitan.logging_utils import logger

class FAdam(Optimizer):

    def __init__(
        self,
        params,
        lr: float = 1e-3,
        weight_decay: float = 1e-3,
        betas: Tuple[float, float] = (0.9, 0.999),
        clip: float = 1.0,
        p: float = 0.5,
        eps: float = 1e-15,
        momentum_dtype: torch.dtype = torch.float32,
        fim_dtype: torch.dtype = torch.float32,
    ):
        """Args:
            params (iterable): iterable of parameters to optimize or dicts
            defining
                parameter groups
            lr (float, optional): learning rate (default: 1e-3)
            betas (Tuple[float, float], optional): coefficients used for
            computing
                running averages of gradient and its square (default: (0.9,
            0.999))
            eps (float, optional): term added to the denominator to improve
                numerical stability (default: 1e-15)
            clip (float, optional): maximum norm of the gradient (default: 1.0)
        """
        defaults = dict(
            lr=lr,
            betas=betas,
            weight_decay=weight_decay,
            eps=eps,
```

```

        momentum_dtype=momentum_dtype,
        fim_dtype=fim_dtype,
        clip=clip,
        p=p,
    )

    super().__init__(params, defaults)

@torch.no_grad()
def step(self, closure: Optional[Callable] = None) -> Optional[float]:
    """Performs a single optimization step.

    Args:
        closure (callable, optional): A closure that reevaluates the model
and
        returns the loss.
    """
    loss = None
    if closure is not None:
        with torch.enable_grad():
            # to fix linter, we do not keep the returned loss for use atm.
            loss = closure()

    for group in self.param_groups:
        beta1, beta2 = group["betas"]
        lr = group["lr"]
        eps = group["eps"]
        clip = group["clip"]
        pval = group["p"]
        momentum_dtype = group["momentum_dtype"]
        fim_dtype = group["fim_dtype"]
        weight_decay = group["weight_decay"]

        for p in group["params"]:
            if p.grad is None:
                continue

            if p.grad.is_sparse:
                raise RuntimeError("FAdam does not support sparse gradients")

```

```

state = self.state[p]

# State initialization
if len(state) == 0:
    state.setdefault("step", torch.tensor(0.0))
    state.setdefault(
        "momentum", torch.zeros_like(p, dtype=momentum_dtype)
    )
    state.setdefault("fim", torch.ones_like(p, dtype=fim_dtype))

# main processing -----

# update the steps for each param group update
state["step"] += 1
step = state["step"]

momentum = state["momentum"]
fim = state["fim"]
grad = p.grad

# begin FAdam algo -----
# 6 - beta2 bias correction per Section 3.4.4
curr_beta2 = beta2 * (1 - beta2 ** (step - 1)) / (1 - beta2**step)

# 7 - update fim
fim.mul_(curr_beta2).add_(grad * grad, alpha=1 - curr_beta2)

# 9 - compute natural gradient
fim_base = fim**pval + eps

grad_nat = grad / fim_base

# 10 - clip the natural gradient
rms = torch.sqrt(torch.mean(grad_nat**2))
divisor = max(1, rms)
divisor = divisor / clip
grad_nat = grad_nat / divisor

# 11 - update momentum
momentum.mul_(beta1).add_(grad_nat, alpha=1 - beta1)

```

```
# 12 - weight decay
grad_weights = p / fim_base

# 13 - clip weight decay
rms = torch.sqrt(torch.mean(grad_weights**2))
divisor = max(1, rms)
divisor /= clip
grad_weights = grad_weights / divisor

# 14 - compute update
full_step = momentum + (weight_decay * grad_weights)
lr_step = lr * full_step

# 15 - update weights
p.sub_(lr_step)

return loss
```