ckkissane first commit 5bde1d5 · 4 months ago History

Code Blame 115 lines (103 loc) · 4.68 KB Raw

```python
from utils import *
from transformer_lens import ActivationCache
import tqdm

class Buffer:
    """
    This defines a data buffer, to store a stack of acts across both model that car
    """

    def __init__(self, cfg, model_A, model_B, all_tokens):
        assert model_A.cfg.d_model == model_B.cfg.d_model
        self.cfg = cfg
        self.buffer_size = cfg["batch_size"] * cfg["buffer_mult"]
        self.buffer_batches = self.buffer_size // (cfg["seq_len"] - 1)
        self.buffer_size = self.buffer_batches * (cfg["seq_len"] - 1)
        self.buffer = torch.zeros(
            (self.buffer_size, 2, model_A.cfg.d_model),
            dtype=torch.bfloat16,
            requires_grad=False,
        ).to(cfg["device"]) # hardcoding 2 for model diffing
        self.cfg = cfg
        self.model_A = model_A
        self.model_B = model_B
        self.token_pointer = 0
        self.first = True
```

Symbols

Find definitions and references for functions and other symbols in this file by clicking a symbol below or in the code.

Filter symbols r

- class Buffer
  - func __init__
  - func estimate_norm_scaling_f...
  - func refresh
  - func next

```
        self.normalize = True
        self.all_tokens = all_tokens

        estimated_norm_scaling_factor_A = self.estimate_norm_scaling_factor(cfg["mc
        estimated_norm_scaling_factor_B = self.estimate_norm_scaling_factor(cfg["mc

        self.normalisation_factor = torch.tensor(
        [
            estimated_norm_scaling_factor_A,
            estimated_norm_scaling_factor_B,
        ],
        device="cuda:0",
        dtype=torch.float32,
        )
        self.refresh()

    @torch.no_grad()
    def estimate_norm_scaling_factor(self, batch_size, model, n_batches_for_norm_es
        # stolen from SAELens https://github.com/jbloomAus/SAELens/blob/6d6eaef343f
        norms_per_batch = []
        for i in tqdm.tqdm(
            range(n_batches_for_norm_estimate), desc="Estimating norm scaling factc
        ):
            tokens = self.all_tokens[i * batch_size : (i + 1) * batch_size]
            _, cache = model.run_with_cache(
                tokens,
                names_filter=self.cfg["hook_point"],
                return_type=None,
            )
            acts = cache[self.cfg["hook_point"]]
            # TODO: maybe drop BOS here
            norms_per_batch.append(acts.norm(dim=-1).mean().item())
        mean_norm = np.mean(norms_per_batch)
        scaling_factor = np.sqrt(model.cfg.d_model) / mean_norm

        return scaling_factor

    @torch.no_grad()
    def refresh(self):
```

```
        self.pointer = 0
        print("Refreshing the buffer!")
        with torch.autocast("cuda", torch.bfloat16):
            if self.first:
                num_batches = self.buffer_batches
            else:
                num_batches = self.buffer_batches // 2
            self.first = False
            for _ in tqdm.trange(0, num_batches, self.cfg["model_batch_size"]):
                tokens = self.all_tokens[
                    self.token_pointer : min(
                        self.token_pointer + self.cfg["model_batch_size"], num_batc
                    )
                ]
                _, cache_A = self.model_A.run_with_cache(
                    tokens, names_filter=self.cfg["hook_point"]
                )
                cache_A: ActivationCache

                _, cache_B = self.model_B.run_with_cache(
                    tokens, names_filter=self.cfg["hook_point"]
                )
                cache_B: ActivationCache

                acts = torch.stack([cache_A[self.cfg["hook_point"]], cache_B[self.c
                acts = acts[:, :, 1:, :] # Drop BOS
                assert acts.shape == (2, tokens.shape[0], tokens.shape[1]-1, self.m
                acts = einops.rearrange(
                    acts,
                    "n_layers batch seq_len d_model -> (batch seq_len) n_layers d_m
                )

                self.buffer[self.pointer : self.pointer + acts.shape[0]] = acts
                self.pointer += acts.shape[0]
                self.token_pointer += self.cfg["model_batch_size"]

        self.pointer = 0
        self.buffer = self.buffer[
            torch.randperm(self.buffer.shape[0]).to(self.cfg["device"])
```

```
104             ]
105
106         @torch.no_grad()
107         def next(self):
108             out = self.buffer[self.pointer : self.pointer + self.cfg["batch_size"]].flo
109             # out: [batch_size, n_layers, d_model]
110             self.pointer += self.cfg["batch_size"]
111             if self.pointer > self.buffer.shape[0] // 2 - self.cfg["batch_size"]:
112                 self.refresh()
113             if self.normalize:
114                 out = out * self.normalisation_factor[None, :, None]
115             return out
```