Universal_Pattern_Generator

December 14, 2025

[ ]:
```python
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import os


from models import *
from config import bargs
from data import *
from einops import rearrange


class ModelTrainer:
    """r
    Train and validate model;
    Extract a certain quantized layer into a specific format
    """

    def __init__(self, debug=True):
        if "VGG" in bargs.model_name:
            model = VGG_quant(
                model_name=bargs.model_name,
                model_config=bargs.model_config,
                weight_bits=bargs.weight_bits,
                act_bits=bargs.act_bits,
            ).to(bargs.device)
        else:
            model = ConvNext_quant(
                model_name=bargs.model_name,
                weight_bits=bargs.weight_bits,
                act_bits=bargs.act_bits,
            ).to(bargs.device)

        model_path = f"./path/{bargs.model_save_name}.pth"
        if os.path.exists(model_path):
            model.load_state_dict(
```

```
                torch.load(
                    model_path,
                    map_location=bargs.device,
                )
            )

        if debug:
            print(model)
            print(
                f"{bargs.layer_num}th layer of model is "
                + str(model.features[bargs.layer_num])
            )

        train_loader, test_loader = get_data_loaders()
        self.best_accuracy = 0
        self.model = model
        self.train_loader = train_loader
        self.test_loader = test_loader
        self.criterion = nn.CrossEntropyLoss().to(bargs.device)
        self.optimizer = optim.AdamW(
            model.parameters(), lr=bargs.init_lr, weight_decay=1e-5
        )

        self.scheduler = optim.lr_scheduler.CosineAnnealingLR(
            self.optimizer, T_max=bargs.epochs, eta_min=bargs.final_lr
        )

    def run(self):
        for epoch in range(1, bargs.epochs + 1):
            self.model.train()

            step = 0
            for input, target in self.train_loader:
                input, target = input.to(bargs.device), target.to(bargs.device)
                output = self.model(input)
                loss = self.criterion(output, target)
                loss.backward()
                step += 1

                if step % bargs.update_steps == 0:
                    self.optimizer.step()
                    self.optimizer.zero_grad()

            if step % bargs.update_steps != 0 and step > 0:
                self.optimizer.step()
                self.optimizer.zero_grad()
```

```
            if epoch % bargs.check_epoch == 0:
                self.validate(epoch, self.model)

            self.scheduler.step()

    def validate(self, epoch):
        self.model.eval()
        correct_count = 0
        total_count = 0

        with torch.no_grad():
            for test_input, test_target in self.test_loader:
                test_input = test_input.to(bargs.device)
                test_target = test_target.to(bargs.device)
                test_output = self.model(test_input)
                preds = test_output.argmax(dim=1)
                correct_count += (preds == test_target).sum().item()
                total_count += test_target.size(0)

        accuracy = (correct_count / total_count) * 100

        if accuracy > self.best_accuracy:
            torch.save(
                self.model.state_dict(),
                f"./path/{bargs.model_save_name}{accuracy * 100: .0f}.pth",
            )
            self.best_accuracy = accuracy

        print(f"Epoch {epoch}, Accuracy: {accuracy:.2f}%")

    def extract_layer(self):
        print(f"Extracting data from Layer {bargs.layer_num}...")

        # Only take 1st image of the batch
        conv_layer = self.model.features[bargs.layer_num]

        captured = {}
        h_conv = self.model.features[bargs.layer_num].register_forward_pre_hook(
            lambda m, i: captured.update({"conv_input": i[0].detach()})
        )
        h_relu = self.model.features[bargs.layer_num + 2].
↪register_forward_pre_hook(
            lambda m, i: captured.update({"relu_output": i[0].detach()})
        )

        self.model.eval()
        images, _ = next(iter(self.test_loader))
```

```
        with torch.no_grad():
            self.model(images.to(bargs.device))
        h_conv.remove()
        h_relu.remove()

        # Quantization
        weight_int = weight_quantize_fn(bargs.weight_bits, training=False)(
            conv_layer.weight, conv_layer.w_alpha
        )
        input_float = captured["conv_input"][0]  # (c,h,w)
        act_int = unsigned_quantization(bargs.act_bits, training=False)(
            input_float, conv_layer.act_alpha
        ).unsqueeze(0)

        # Different Outputs
        output_quant = captured["relu_output"][0]
        output_int = F.relu_(
            F.conv2d(
                act_int,
                weight_int,
                stride=conv_layer.stride,
                padding=conv_layer.padding,
            )
        ).squeeze(0)
        output_ref = F.relu_(
            F.conv2d(
                input_float,
                conv_layer.weight,
                stride=conv_layer.stride,
                padding=conv_layer.padding,
            )
        ).squeeze(0)

        # Save Files
        if bargs.pe_config == "ws":
            act_tile = rearrange(
                F.pad(act_int, pad=(1, 1, 1, 1), value=0),
                "1 (th ts) h w -> th ts (h w)",
                th=bargs.tile_image_size,
                ts=bargs.tile_size,
            )  # (cin/t, t, hw)
            w_tile = rearrange(
                weight_int,
                "(th tsh) (tw tsw) h w -> tw th tsh tsw (h w)",
                tsh=bargs.tile_size,
                tsw=bargs.tile_size,
            )  # (cin/t, cout/t, t, t, k^2)
```

```
            psum = torch.einsum(
                "abpqk, apn -> abpnk", w_tile, act_tile
            )  # (cin/t, cout/t, t, hw, k^2)
            for cin_tile in range(bargs.tile_image_size):
                self._save_file(
                    data=act_tile[cin_tile],  # (t, (h+2p)(w+2p))
                    filename="./Files/"
                    + str(bargs.act_bits)
                    + "bit/"
                    + bargs.pe_config
                    + "/activation_tile"
                    + str(cin_tile)
                    + ".txt",
                    bits=bargs.act_bits,
                )
                for cout_tile in range(bargs.tile_image_size):
                    for kij in range(9):
                        self._save_file(
                            data=w_tile[cin_tile, cout_tile, :, :, kij],  #␣
↪(t,t)
                            filename="./Files/"
                            + str(bargs.act_bits)
                            + "bit/"
                            + bargs.pe_config
                            + "/weight_tile_"
                            + str(cin_tile * bargs.tile_image_size + cout_tile)
                            + "_kij_"
                            + str(kij)
                            + ".txt",
                            bits=bargs.weight_bits,
                        )
                        self._save_file(
                            data=psum[
                                cin_tile, cout_tile, :, :, kij
                            ],  # [t, (h+2p)(w+2p)]
                            filename="./Files/"
                            + str(bargs.act_bits)
                            + "bit/"
                            + bargs.pe_config
                            + "/psum_"
                            + str(cin_tile * bargs.tile_image_size + cout_tile)
                            + "_kij_"
                            + str(kij)
                            + ".txt",
                            bits=bargs.weight_bits,
                        )
```

```
        elif bargs.pe_config == "os":
            act_padded = F.pad(act_int, (1, 1, 1, 1))
            for i in range(bargs.channel // bargs.tile_size):
                for j in range(2):
                    self._save_file(
                        data=act_padded[
                            0,
                            bargs.tile_size * i : bargs.tile_size * i + bargs.
↪tile_size,
                            2 * j : 2 * j + 4,
                            :,
                        ].reshape(bargs.tile_size, -1),
                        filename=(
                            "./Files/"
                            + str(bargs.act_bits)
                            + "bit/"
                            + bargs.pe_config
                            + "/channel_group_"
                            + str(i)
                            + ("_upper" if j == 0 else "_lower")
                            + ".txt"
                        ),
                        bits=bargs.act_bits,
                    )

                reshaped_weight = rearrange(
                    weight_int,
                    "(tn par_cout) (ts par_cin) k1 k2 -> tn par_cin (k1 k2) ts␣
↪par_cout",
                    tn=bargs.tile_image_size,
                    ts=bargs.tile_size,
                )
                for tn in range(bargs.tile_image_size):
                    for kij in range(9):
                        self._save_file(
                            data=reshaped_weight[tn, i, kij, :, :].reshape(8,␣
↪-1),
                            filename="./Files/"
                            + str(bargs.act_bits)
                            + "bit/"
                            + bargs.pe_config
                            + "/weight_channel_group_"
                            + str(i)
                            + "_kij_"
                            + str(kij)
                            + "_tile_"
```

```
                        + str(tn)
                        + ".txt",
                        bits=bargs.weight_bits,
                    )

        for i in range(bargs.tile_image_size):
            self._save_file(
                data=rearrange(
                    output_int, "(tn ts) h w -> tn ts (h w)", ts=bargs.tile_size
                )[i],
                filename="./Files/"
                + str(bargs.act_bits)
                + "bit/output_"
                + str(i)
                + ".txt",
                bits=16,
            )

            if "bn" in bargs.model_config:
                bn = self.model.features[bargs.layer_num + 1]
                mu = bn.running_mean.reshape(bargs.tile_size, -1).to(torch.
↪float16)
                sigma = (
                    torch.sqrt(bn.running_var + bn.eps)
                    .reshape(bargs.tile_size, -1)
                    .to(torch.float16)
                )
                gamma = bn.weight.reshape(bargs.tile_size, -1).to(torch.float16)
                beta = bn.bias.reshape(bargs.tile_size, -1).to(torch.float16)

                self._save_file(
                    data=mu[:, i].unsqueeze(-1).unsqueeze(-1),
                    filename="./Files/"
                    + str(bargs.act_bits)
                    + "bit/bn_mu_"
                    + str(i)
                    + ".txt",
                    bits=16,
                )
                self._save_file(
                    data=sigma[:, i].unsqueeze(-1),
                    filename="./Files/"
                    + str(bargs.act_bits)
                    + "bit/bn_sigma_"
                    + str(i)
                    + ".txt",
                    bits=16,
```

```
                )
                self._save_file(
                    data=gamma[:, i].unsqueeze(-1),
                    filename="./Files/"
                    + str(bargs.act_bits)
                    + "bit/bn_gamma_"
                    + str(i)
                    + ".txt",
                    bits=16,
                )
                self._save_file(
                    data=beta[:, i].unsqueeze(-1),
                    filename="./Files/"
                    + str(bargs.act_bits)
                    + "bit/bn_beta_"
                    + str(i)
                    + ".txt",
                    bits=16,
                )

        # Error Calculation
        total_scale = (
            conv_layer.w_alpha
            * conv_layer.act_alpha
            / (2 ** (bargs.weight_bits - 1) - 1)
            / (2**bargs.act_bits - 1)
        )
        training_error = (output_quant - output_ref).abs().mean()
        quant_error = (output_quant - output_int * total_scale).abs().mean()
        print(
            f"Training Error: {training_error:.6f},    Quant Error:␣
↪{quant_error:.6f}"
        )

    def _save_file(self, data, filename, bits):
        """r
        data: (tile_size, -1)
        """
        os.makedirs(os.path.dirname(filename), exist_ok=True)

        file = open(filename, "w")
        file.write("#time0row7[msb-lsb],time0row6[msb-lst],....
↪,time0row0[msb-lst]#\n")
        file.write("#time1row7[msb-lsb],time1row6[msb-lst],....
↪,time1row0[msb-lst]#\n")
        file.write("#................#\n")
        fmt_str = f"0{bits}b"
```

```
        for i in range(data.size(1)):
            for j in range(data.size(0)):
                data_value = round(data[7 - j, i].item())
                if data_value < 0:
                    data_bin = format(data_value & (2**bits - 1), fmt_str)
                else:
                    data_bin = format(data_value, fmt_str)
                for k in range(bits):
                    file.write(data_bin[k])
            file.write("\n")
        file.close()


if __name__ == "__main__":
    trainer = ModelTrainer(debug=True)
    trainer.validate(0)
    # trainer.run()
    # trainer.extract_layer()
```