Quantization

You can quantize:

- Weights
- Activation

Challenges:

- Quantization Error
- Retraining
- Limited Hardware support
- Calibration dataset needed
- packing/unpacking

Linear Quantization

$$r = s(q - z)$$

s: scale z: zero point q: quantized number r: number to be quantized

z is stored in same data types as of quantized numbers

$$q = \text{int}\left(round\left(\frac{r}{s} + z\right)\right)$$

```
def linear_q_with_scale_and_zero_point(tensor, scale, zero_point, dtype =
   torch.int8):
    scaled_and_shifted_tensor = tensor / scale + zero_point
    rounded_tensor = torch.round(scaled_and_shifted_tensor)
    q_min = torch.iinfo(dtype).min
    q_max = torch.iinfo(dtype).max
    q_tensor = rounded_tensor.clamp(q_min,q_max).to(dtype)
    return q_tensor

def linear_dequantization(quantized_tensor, scale, zero_point):
    return scale * (quantized_tensor.float() - zero_point)
```

Calculation of Scale and zero point

$$z = \text{int}(round(q_{min} - r_{min}/s))$$

$$s = \frac{(r_{max} - r_{min})}{(q_{max} - q_{min})}$$

```
def get_q_scale_and_zero_point(tensor, dtype=torch.int8):
    q_min, q_max = torch.iinfo(dtype).min, torch.iinfo(dtype).max
    r_min, r_max = tensor.min().item(), tensor.max().item()
    scale = (r_max - r_min) / (q_max - q_min)
    zero_point = q_min - (r_min / scale)
    # clip the zero_point to fall in [quantized_min, quantized_max]
    if zero_point < q_min:
        zero_point = q_min
    elif zero_point > q_max:
        zero_point = q_max
    else:
        # round and cast to int
        zero_point = int(round(zero_point))
    return scale, zero_point

def linear_quantization(tensor, dtype=torch.int8):
    scale, zero_point = get_q_scale_and_zero_point(tensor, dtype=dtype)
    quantized_tensor = linear_q_with_scale_and_zero_point(tensor, scale, zero_point,
                                                          dtype=dtype)
    return quantized_tensor, scale, zero_point
```

Two modes in Linear Quantization:

- Asymmetric: Map $[r_{min}, r_{max}]$ to $[q_{min}, q_{max}]$

- Symmetric: Map $[-r_{max}, r_{max}]$ to $[-q_{max}, q_{max}]$, set $r_{max} = max(|r_{tensor}|)$

In Symmetric Quantization, zero point = 0

$$q = \text{int}(round(r/s))$$

$$s = r_{max}/q_{max}$$

```
def get_q_scale_symmetric(tensor, dtype=torch.int8):
    r_max = tensor.abs().max().item()
    q_max = torch.iinfo(dtype).max
    # return the scale
    return r_max/q_max
```

Per-channel Quantization

```
def linear_q_symmetric_per_channel(r_tensor, dim, dtype=torch.int8):
    output_dim = r_tensor.shape[dim]
    scale = torch.zeros(output_dim)
    for index in range(output_dim):
        sub_tensor = r_tensor.select(dim, index)
        scale[index] = get_q_scale_symmetric(sub_tensor, dtype=dtype)
    # reshape the scale
    scale_shape = [1] * r_tensor.dim()
    scale_shape[dim] = -1
    scale = scale.view(scale_shape)
    quantized_tensor = linear_q_with_scale_and_zero_point(
        r_tensor, scale=scale, zero_point=0, dtype=dtype)
    return quantized_tensor, scale
```

Per-Group Quantization

```
def linear_q_symmetric_per_group(tensor, group_size, dtype=torch.int8):
    t_shape = tensor.shape
    num_groups = (tensor.numel() + group_size - 1) // group_size
    tensor = tensor.view(num_groups, group_size)
    quantized_tensor, scale = linear_q_symmetric_per_channel(tensor, dim=0,
                                        dtype=dtype)
    quantized_tensor = quantized_tensor.view(t_shape)
    return quantized_tensor, scale
```

Custom 8-bit Quantizer

W8A16 linear layer

```
class W8A16LinearLayer(nn.Module):
    def __init__(self, in_features, out_features, bias=True, dtype=torch.float32):
        super().__init__()
        self.register_buffer("int8_weights", torch.randint(-128, 127, (out_features,
                  in_features), dtype=torch.int8))
        self.register_buffer("scales", torch.randn((out_features), dtype=dtype))
        if bias:
            self.register_buffer("bias", torch.randn((1, out_features), dtype=dtype))
        else:
            self.bias = None

    def quantize(self, weights):
        w_fp32 = weights.clone().to(torch.float32)
        scales = w_fp32.abs().max(dim=-1).values / 127
        scales = scales.to(weights.dtype)
        int8_weights = torch.round(weights/scales.unsqueeze(1)).to(torch.int8)
        self.int8_weights = int8_weights
        self.scales = scales

    def forward(self, input):
        casted_weights = self.int8_weights.to(input.dtype)
        output = F.linear(input, casted_weights) *  self.scales
```

```
        if bias is not None:
            output = output +  self.bias
        return output
```

Replacing layers in a model

```
class DummyModel(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.emb = torch.nn.Embedding(1, 1)
        self.linear_1 = nn.Linear(1, 1)
        self.linear_2 = nn.Linear(1, 1, bias=False)
        self.lm_head = nn.Linear(1, 1, bias=False)

def replace_linear_with_target(module, target_class, module_name_to_exclude):
    for name, child in module.named_children():
        if isinstance(child, nn.Linear) and not any([x == name for x in
                                    module_name_to_exclude]):
            old_bias = child.bias
            new_module = target_class(child.in_features, child.out_features,
                                      old_bias is not None, child.weight.dtype)
            setattr(module, name, new_module)
            getattr(module, name).quantize(old_weight)
            if old_bias is not None:
                getattr(module, name).bias = old_bias
        else:
            replace_linear_with_target(child, target_class, module_name_to_exclude)
```

Weight Packing

When you do `tensor = torch.tensor([0,1], dtype= torch.int4)` it is not supported. We have to use the lowest bit support, which is not ideal because tensor will occupy 8-bit per datapoint and might add a considerable overhead for large models.

```
def pack_weights(uint8tensor, bits):
    #Assuming 8/bits is natural number
    num_values = uint8tensor.shape[0] * bits // 8
    num_steps = 8 // bits
    unpacked_idx = 0
    packed_tensor = torch.zeros((num_values), dtype=torch.uint8)
    for i in range(num_values):
        for j in range(num_steps):
            packed_tensor[i] |= uint8tensor[unpacked_idx] << (bits * j)
            unpacked_idx += 1
    return packed_tensor
def unpack_weights(uint8tensor, bits):
    num_values = uint8tensor.shape[0] * 8 // bits
    num_steps = 8 // bits
    unpacked_tensor = torch.zeros((num_values), dtype=torch.uint8)
    unpacked_idx = 0
    mask = 2 ** bits - 1
    for i in range(uint8tensor.shape[0]):
        for j in range(num_steps):
            unpacked_tensor[unpacked_idx] |= uint8tensor[i] >> (bits * j)
            unpacked_idx += 1
    unpacked_tensor &= mask
    return unpacked_tensor
```