Advanced NLP Assignment 2

12112627 李乐平

Q1

Write the pytorch code for the bind network, the input is image_feature and output is transformered_image_feature . Define the __init__() and forward() function. You can add anything/arguments you like in the code. Note that the bind network has three blocks ( ).

Solution:

In [1]:
```python
import torch
import torch.nn as nn

# https://github.com/bzhangGo/rmsnorm/blob/master/rmsnorm_torch.py
class RMSNorm(nn.Module):
    def __init__(self, d, p=-1., eps=1e-8, bias=False):
        """
            Root Mean Square Layer Normalization
        :param d: model size
        :param p: partial RMSNorm, valid value [0, 1], default -1.0 (disabled)
        :param eps:  epsilon value, default 1e-8
        :param bias: whether use bias term for RMSNorm, disabled by
            default because RMSNorm doesn't enforce re-centering invariance.
        """
        super(RMSNorm, self).__init__()

        self.eps = eps
        self.d = d
        self.p = p
        self.bias = bias

        self.scale = nn.Parameter(torch.ones(d))
        self.register_parameter("scale", self.scale)

        if self.bias:
            self.offset = nn.Parameter(torch.zeros(d))
            self.register_parameter("offset", self.offset)

    def forward(self, x):
        if self.p < 0. or self.p > 1.:
            norm_x = x.norm(2, dim=-1, keepdim=True)
            d_x = self.d
        else:
            partial_size = int(self.d * self.p)
            partial_x, _ = torch.split(x, [partial_size, self.d - partial_size], dim=-1)

            norm_x = partial_x.norm(2, dim=-1, keepdim=True)
            d_x = partial_size

        rms_x = norm_x * d_x ** (-1. / 2)
        x_normed = x / (rms_x + self.eps)

        if self.bias:
            return self.scale * x_normed + self.offset

        return self.scale * x_normed

class bind_network(nn.Module):
    def __init__(self, args):
        super(bind_network, self).__init__()
        self.image_dim = args.image_dim    # CI
        self.model_dim = args.model_dim    # C
        self.ffn_dim = self.model_dim * 4  # Ch

        self.linear0 = nn.Linear(self.image_dim, self.model_dim, bias = False)  # w0 CI*C

        self.projection_blocks = nn.ModuleList([
            nn.Sequential(
                RMSNorm(self.model_dim),                                        # RMSNorm
                nn.Linear(self.model_dim, self.ffn_dim, bias = False),          # w1 C*Ch
                nn.Linear(self.model_dim, self.ffn_dim, bias = False),          # w2 C*Ch
                nn.Linear(self.ffn_dim, self.model_dim, bias = False),          # w3 Ch*C
                nn.SiLU()                                                       # SiLU
            ) for _ in range(3)
        ])

    def forward(self, image_feature):
        F_I_0 = self.linear0(image_feature)
        F_I_i = F_I_0
        for block in self.projection_blocks:
            F_I_i = block[0](F_I_i)
            F_I_i = F_I_i + block[3](block[2](F_I_i) * block[4](block[1](F_I_i)))

        return F_I_i


class Args:
    def __init__(self, image_dim, model_dim):
        self.image_dim = image_dim
        self.model_dim = model_dim

args = Args(image_dim = 64, model_dim = 512)

bind_net = bind_network(args)

image_feature = torch.randn(1, args.image_dim)

output = bind_net(image_feature)

print(f"""
Image Shape: {image_feature.shape}
Image Feature: {image_feature}
Output Shape: {output.shape}
Output: {output}
""")
```

```
Image Shape: torch.Size([1, 64])
Image Feature: tensor([[ 1.3874, -1.5726, -1.3388, -1.0086,  0.7970,  2.5626,  0.2656, -0.2886,
          1.9697,  1.8278, -0.6941, -0.3214, -2.7398,  0.2542,  0.3267,  0.5992,
         -1.6390, -0.5445, -1.8141, -0.4989,  0.0957, -0.1325, -1.3822, -1.2853,
          1.6776,  1.7085, -0.7028,  1.0429,  0.9523, -0.3974,  1.1003, -0.0918,
          0.2438, -0.3119, -0.0368,  1.0498, -1.4760,  0.9999,  0.4980, -1.9578,
          0.6779,  0.2079,  0.4291, -0.4033,  2.9337, -0.7550,  1.9450, -0.8512,
          1.2409,  0.7904,  0.3042, -1.3916, -0.5970, -0.8465, -1.0813, -0.9439,
          0.9729,  0.3848,  0.5232, -0.0292,  0.5456, -1.1885, -0.6250, -0.3410]])
Output Shape: torch.Size([1, 512])
Output: tensor([[-0.6790, -0.4598, -0.3136,  2.2281,  0.2798,  0.7672,  0.5229, -0.1448,
          0.1670,  1.0647, -0.3899, -0.6772, -1.4385,  0.2184,  0.7808, -1.1050,
          0.7648,  0.7238, -0.8145,  0.6065,  1.2087,  1.4226,  0.1107, -1.1490,
          0.6066,  1.6156,  0.2304, -2.9906,  0.6499, -0.3870, -0.7582,  1.2478,
          1.0797,  0.6800,  2.4183,  1.3473,  0.2718,  1.4041,  0.5853, -0.2894,
          1.2984, -0.3243, -0.7392, -0.3412,  0.6381,  1.0599, -0.6536,  1.3608,
         -0.8283,  0.5268,  1.3062, -0.3730,  0.1469,  0.8579,  0.6458, -1.2272,
         -0.7672,  0.6986, -0.9615, -0.4958,  1.5251, -0.8008,  1.0973, -0.6409,
```

Q2

The Transformer applies self-attention mechanism. Here is the corresponding code in the Fairseq-toolkit. Please write the line number(s) for the code(s) which do the scaling with the coefficient.

Solution:

In [ ]:
```
"""
self.scaling = self.head_dim ** 0.5  # Line 114
q *= self.scaling                    # Line 602
"""
```

Q3

RMSNorm is another regularizer which can replace the layer normalization. Read the paper and briefly illustrate the calculation process of RMSNorm and its advantages over Layer Normalization

Solution: The RNSNorm is calculated as below

$$\bar{a}_i = \frac{a_i}{\text{RMS}(\vec{a})} g_i, \text{where RMS}(\vec{a}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} a_i^2}, i = 1, 2, \ldots, n.$$

where g_i is the gain parameter used to re-scale the standardized summed inputs, and is set to 1 at the beginning.

One of the advantages of RMSNorm to LayerNorm is that RMSNorm simplifies computation, reducing running time by 6.9% to 40% on different models, while it performs comparable or better performance than LayerNorm on various tasks and neural architectures. Also, RMSNorm is invariant to the scaling of inputs and weights, which stabilizes the layer activations and the model gradients. It also has an implicit learning rate adaptation ability that avoids large-norm weight matrix and improves model convergence.

Appendix

LLM used:

ChatGPT 3.5

请根据以下文字编写相应的代码：

Bind Network.

In Figure 3, we present the details of the bind network, which aims to align the embedding space between ImageBind and LLaMA. Specifically, we denote the CI -dimensional global image feature encoded by ImageBind as FI ∈ R^(1×CI). In the bind network, we first adopt a linear projection layer with a weight matrix w0 ∈ R^(CI×C) , formulated as F_I^0 = F_I w_0 ∈ R(1×C), where C denotes the feature dimension of LLaMA. Inspired by the Feed-Forward Network (FFN) in LLaMA, we then cascade three projection blocks with RMSNorm [43], SiLU activation functions [44], and residual connections [45]. For the (i + 1)-th block with F_I^i as input, we formulate the calculation of F_I^(i+1) as (the normalization is omitted for simplicity)

F_I^(i+1) = F_I^i + (F_I^i w_2 · SiLU(F_I^i w_1))w_3, 0 ≤ i < 3 (1)

where w1, w2 ∈ R^(C×Ch) and w3 ∈ R^(Ch×C) , with Ch denoting the hidden dimension. After the bind network, we obtain the transformed image feature, T_I ∈ R^(1×C) , which learns to align the embedding space from ImageBind to LLaMA

New Bing

Read the paper and summarize the advantages of RMSNorm when compared with LayerNorm