2023 Fall Deep Learning Lab3 Report

109651066 林柏佑

Screenshot of task1 (Transformer)

- Number of layers：3

```
self.encoder = TransformerEncoder(
    num_layers=3,
```

- Parameter size：497.43K

```
The parameter size of encoder block is 497.43k
```

- Accuracy：0.7781

```
Train: 100% 2000/2000 [02:25<00:00, 13.70 step/s, accuracy=0.89, loss=0.39, step=7e+4]
Valid:  99% 5632/5667 [00:02<00:00, 2758.22 uttr/s, accuracy=0.77, loss=1.09]
Step 70000, best model saved. (accuracy=0.7781)
```

Screenshot of task2 (Conformer)

- Number of layers：3

```
self.encoder = ConformerEncoder(
    num_layers=3,
```

- Parameter size：448.504K

```
The parameter size of encoder block is 448.504k
```

- Accuracy：0.7995

```
Train: 100% 2000/2000 [02:40<00:00, 12.50 step/s, accuracy=0.97, loss=0.11, step=7e+4]
Valid:  99% 5632/5667 [00:02<00:00, 2660.82 uttr/s, accuracy=0.80, loss=1.09]
Step 70000, best model saved. (accuracy=0.7995)
```

In task2

- Which kind of transformer-like model do you choose？
- The reason why you choose this model.
- The advantage of chosen model.

我會選擇 Conformer。因為 Transformer 是基於 Self-attention 設計，在針對大範圍前後有相關的特徵資訊，雖有較好的效果，但缺乏局部細微的特徵。而 CNN 提取局部細微特徵的效果非常好。而正好 Conformer 的架構是將 Self-attention 和 Convolution layer 這兩著結合，各自擷取其優點。

Anything you do to improve the performance

我認為在 ADD 和 Norm 那層，可以改成先做 Norm 再做 Residual，效果上會有提升！而這個想法在 2020 年也有學者證實，並發表成論文。如下：

參考論文：On Layer Normalization in the Transformer Architecture

Screenshot of your transformer code for both encoder layer and encoder

- **Plagiarism is forbidden !!!**
- **There should be comment in your code**
- **For both Task1 and Task2**

Task1：

```
class TransformerEncoderLayer(nn.Module):
    def __init__(self, d_model, nheads, dim_feedforaward, dropout):
        super(TransformerEncoderLayer, self).__init__()
        self.multi_attention = MultiAttention(d_model, nheads)
        self.feed_forward = FeedForward(d_model, dim_feedforaward)
        self.norm = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        output = self.multi_attention(x, x, x)
        x = self.norm(x + self.dropout(output))  # add & norm
        output = self.feed_forward(x)
        return self.norm(x + self.dropout(output))  # add & norm


class TransformerEncoder(nn.Module):
    def __init__(
        self, num_layers, input_size, d_model, n_heads, dim_feedforaward, dropout
    ):
        super(TransformerEncoder, self).__init__()
        self.embedding = nn.Embedding(input_size, d_model)
        self.positional_encoding = PositionalEncoding(d_model)
        self.layers = nn.ModuleList(
            [
                TransformerEncoderLayer(d_model, n_heads, dim_feedforaward, dropout)
                for _ in range(num_layers)
            ]
        )
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        x = self.dropout(self.positional_encoding(self.embedding(x)))
        for layer in self.layers:
            output = layer(x)
        return output
```

Task2：

```
class ConformerEncoderLayer(nn.Module):
    def __init__(self, d_model, nheads, dim_feedforward, dropout=0.1):
        super(ConformerEncoderLayer, self).__init__()
        self.multi_attention = MultiAttention(d_model, nheads)
        self.norm1 = nn.LayerNorm(d_model)
        self.feedforward = nn.Sequential(
            nn.Linear(d_model, dim_feedforward),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(dim_feedforward, d_model),
            nn.Dropout(dropout),
        )
```

```python
        self.norm2 = nn.LayerNorm(d_model)
        self.conv1d = nn.Conv1d(
            d_model, 2 * d_model, kernel_size=3, padding=1, bias=False
        )
        self.glu = nn.GLU(dim=1)
        self.linear1 = nn.Linear(d_model, dim_feedforward)
        self.linear2 = nn.Linear(dim_feedforward, d_model)
        self.norm3 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        # multi-head self-attention
        attn_output = self.multi_attention(x, x, x)
        x = x + attn_output
        x = self.norm1(x)

        # feedforward network
        feedforward_output = self.feedforward(x)
        x = x + feedforward_output
        x = self.norm2(x)

        # convolutinal layer
        residual = x
        x = x.transpose(1, 2)
        x = self.conv1d(x)
        x = self.glu(x)
        x = x.transpose(1, 2)
        x = self.linear2(F.relu(self.linear1(x)))
        x = self.norm3(x + residual)

        return self.dropout(x)
```

```python
class ConformerEncoder(nn.Module):
    def __init__(
        self,
        input_size,
        output_size,
        d_model,
        nheads,
        dim_feedforward,
        num_layers,
        dropout=0.1,
    ):
        super(ConformerEncoder, self).__init__()
        self.conv1d = nn.Conv1d(
            input_size, d_model, kernel_size=3, padding=1  # , bias=False
        )
        self.positional_encoding = nn.Parameter(torch.zeros(1, 1, d_model))
        self.dropout = nn.Dropout(dropout)
        self.layers = nn.ModuleList(
            [
                ConformerEncoderLayer(d_model, nheads, dim_feedforward, dropout)
                for _ in range(num_layers)
            ]
        )
        self.fc = nn.Linear(d_model, output_size)

    def forward(self, x):
        x = self.conv1d(x)  # convolutional layer
        x = x + self.positional_encoding[:, :, : x.size(2)]  # positional encoding
        for layer in self.layers:  # conformer blocks
            x = layer(x)
        x = x.mean(dim=2)  # global average pooling
        return self.fc(x)  # linear layer
```