# 安徽大学《深度学习与神经网络》
# 实验报告 5

学号：WA2214014　　专业：人工智能　　姓名：杨跃浙

实验日期：06.15　　教师签字：　　成绩：

**[实验名称]** 注意力机制与 Transformer 实验

**[实验目的]**

1．熟悉和掌握注意力评分函数、Bahdanau 注意力、多头注意力

2．熟悉和掌握自注意力和位置编码

3．熟悉和掌握 Transformer

**[实验要求]**

1．采用 Python 语言基于 PyTorch 深度学习框架进行编程

2．代码可读性强：变量、函数、类等命名可读性强，包含必要的注释

3．提交实验报告要求：

- 命名方式："学号-姓名-Lab-N"（N 为实验课序号，即：1-6）；
- 截止时间：下次实验课当晚 23:59；
- 提交方式：智慧安大-网络教育平台-作业；
- 按时提交（过时不补）；

**[实验内容]**

1．多种注意力评分函数

- 学习、运行、调试和比较参考教材 10.3 小节多种类型注意力评分函数
- 完成练习题 1

2．Bahdanau 注意力：

- 学习、运行和调试参考教材 10.4 小节内容

3．多头注意力：

- 学习、运行和调试参考教材 10.5 小节内容
- 完成练习题 1

4．自注意力和位置编码：

- 学习、运行和调试参考教材 10.6 小节内容
- 完成练习题 2

5．Transformer：

- 学习、运行和调试参考教材 10.7 小节内容
- 完成练习题 3 和 6

6．参考资料：

- **参考教材**：https://zh-v2.d2l.ai/d2l-zh-pytorch.pdf
- **PyTorch 官方文档**：https://pytorch.org/docs/2.0/；
- **PyTorch 官方论坛**：https://discuss.pytorch.org/

[实验代码和结果]

# 1. 多种注意力评分函数

## 实验代码:

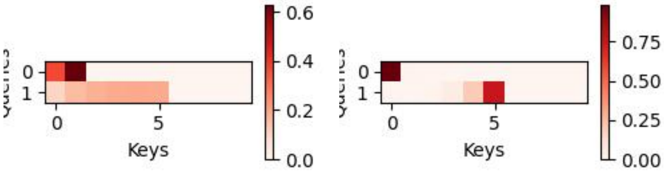
```
import math
import torch
from torch import nn
from d2l import torch as d2l
import os

# 数据和结果路径
DATA_PATH = "/home/yyz/NNDL-Class/Project5/Data"
RESULT_PATH = "/home/yyz/NNDL-Class/Project5/Result"
os.makedirs(DATA_PATH, exist_ok=True)
os.makedirs(RESULT_PATH, exist_ok=True)

# 掩蔽 softmax 操作
def masked_softmax(X, valid_lens):
if valid_lens is None:
return nn.functional.softmax(X, dim=-1)
else:
shape = X.shape
if valid_lens.dim() == 1:
valid_lens = torch.repeat_interleave(valid_lens, shape[1])
else:
valid_lens = valid_lens.reshape(-1)
X = d2l.sequence_mask(X.reshape(-1, shape[-1]), valid_lens,
value=-1e6)
return nn.functional.softmax(X.reshape(shape), dim=-1)

# 加性注意力
class AdditiveAttention(nn.Module):
def __init__(self, key_size, query_size, num_hiddens,
dropout, **kwargs):
super(AdditiveAttention, self).__init__(**kwargs)
self.W_k = nn.Linear(key_size, num_hiddens, bias=False)
```

```
self.W_q = nn.Linear(query_size, num_hiddens, bias=False)
self.w_v = nn.Linear(num_hiddens, 1, bias=False)
self.dropout = nn.Dropout(dropout)

def forward(self, queries, keys, values, valid_lens):
queries, keys = self.W_q(queries), self.W_k(keys)
features = queries.unsqueeze(2) + keys.unsqueeze(1)
features = torch.tanh(features)
scores = self.w_v(features).squeeze(-1)
self.attention_weights = masked_softmax(scores, valid_lens)
return torch.bmm(self.dropout(self.attention_weights),
values)

# 缩放点积注意力
class DotProductAttention(nn.Module):
def __init__(self, dropout, **kwargs):
super(DotProductAttention, self).__init__(**kwargs)
self.dropout = nn.Dropout(dropout)

def forward(self, queries, keys, values, valid_lens=None):
d = queries.shape[-1]
scores = torch.bmm(queries, keys.transpose(1,2)) /
math.sqrt(d)
self.attention_weights = masked_softmax(scores, valid_lens)
return torch.bmm(self.dropout(self.attention_weights),
values)

# 小例子：运行和可视化
def run_attention_demo():
# 构造输入
queries_add = torch.normal(0, 1, (2, 1, 20))
queries_dot = torch.normal(0, 1, (2, 1, 2)) # 与 keys 维度一
致

# 改动的 keys（练习题 1 要求修改）
keys = torch.arange(20, dtype=torch.float32).reshape(10, 2)
keys = keys.unsqueeze(0).repeat(2, 1, 1)
```

```python
values = torch.arange(40, dtype=torch.float32).reshape(1,
10, 4).repeat(2, 1, 1)
valid_lens = torch.tensor([2, 6])

# 加性注意力
add_attention = AdditiveAttention(2, 20, 8, 0.1)
add_attention.eval()
out_add = add_attention(queries_add, keys, values,
valid_lens)

# 缩放点积注意力
dot_attention = DotProductAttention(0.1)
dot_attention.eval()
out_dot = dot_attention(queries_dot, keys, values,
valid_lens)

print("加性注意力输出:\n", out_add)
print("缩放点积注意力输出:\n", out_dot)

# 可视化
d2l.show_heatmaps(add_attention.attention_weights.reshape
((1, 1, 2, 10)),
xlabel='Keys', ylabel='Queries')
d2l.plt.savefig(f"{RESULT_PATH}/additive_attention_heatma
p.png")

d2l.show_heatmaps(dot_attention.attention_weights.reshape
((1, 1, 2, 10)),
xlabel='Keys', ylabel='Queries')
d2l.plt.savefig(f"{RESULT_PATH}/dot_product_attention_hea
tmap.png")

if __name__ == "__main__":
run_attention_demo()
```

**实验结果:**

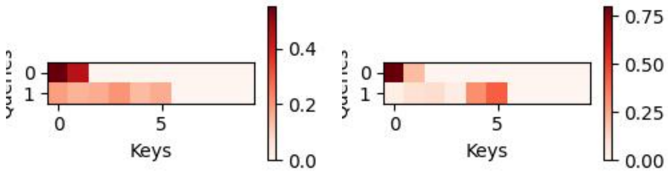
```
● (yyzttt) (base) yyz@4028Dog:~$ /usr/local/anaconda3/envs/yyzttt/bin/python /h
加性注意力输出:
 tensor([[[ 2.5209,  3.5209,  4.5209,  5.5209]],

        [[11.1302, 12.1302, 13.1302, 14.1302]]], grad_fn=<BmmBackward0>)
缩放点积注意力输出:
 tensor([[[ 0.0489,  1.0489,  2.0489,  3.0489]],

        [[18.6398, 19.6398, 20.6398, 21.6398]]])
```




练习题 1：修改小例子中的键，并且可视化注意力权重。可加性注意力和缩放的“点－积”注意力是否仍然产生相同的结果？为什么？

```
# 更改 key 的数值
keys = torch.tensor([[[1.0, 0.0], [0.0, 1.0], [1.0, 1.0], [2.0, 0.0], [0.0, 2.0],
[2.0, 2.0], [3.0, 1.0], [1.0, 3.0], [4.0, 0.0], [0.0, 4.0]]] * 2)
```

```
● (yyzttt) (base) yyz@4028Dog:~$ /usr/local/anaconda3/envs/yyzttt/bin/python /ho
yz/NNDL-Class/Project5/Code/attentions.py
加性注意力输出:
 tensor([[[ 1.7838,  2.7838,  3.7838,  4.7838]],

        [[ 9.7565, 10.7565, 11.7565, 12.7565]]], grad_fn=<BmmBackward0>)
缩放点积注意力输出:
 tensor([[[ 0.7853,  1.7853,  2.7853,  3.7853]],

        [[15.1168, 16.1168, 17.1168, 18.1168]]])
```




当修改小例子中的键后，加性注意力和缩放的“点—积”注意力不再产生相同的结果。这是因为两种注意力机制的计算逻辑存在本质差异：加性注意力通过线性变换将查询和键映射到相同维度后相加，再通过双曲正切函数和输出层计算得分，适用于查询和键维度不同的场景；而缩放点积注意力直接计算查询和键的点积并除以键维度的平方根，要求查询和键的最后一维维度一致，依赖于两者的内积关系。当键的数值被修改后，尤其是键的特征分布发生变化时，加性注意力中可学习的权重矩阵会根据新的键特征调整映射关系，而缩放点积注意力的点积计算对键的数值变化更为敏感，导致两者的注意力权重和输出结果出现差异。

## 2. Bahdanau 注意力：

**实验代码：**

```
import torch
from torch import nn
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import os
import seaborn as sns
from collections import Counter
```

```python
# 添加缺失的 masked_softmax 函数
def masked_softmax(X, valid_lens):
if valid_lens is None:
return nn.functional.softmax(X, dim=-1)
else:
shape = X.shape
if valid_lens.dim() == 1:
valid_lens = torch.repeat_interleave(valid_lens, shape[1])
else:
valid_lens = valid_lens.reshape(-1)
max_len = X.size(-1)
mask = torch.arange(max_len, dtype=torch.float32,
device=X.device)[None, :] < valid_lens[:, None]
mask = mask.reshape(shape)
X_masked = X.clone()
X_masked[~mask] = -1e6
return nn.functional.softmax(X_masked, dim=-1)

# 定义 AdditiveAttention 类
class AdditiveAttention(nn.Module):
def __init__(self, key_size, query_size, num_hiddens,
dropout, **kwargs):
super(AdditiveAttention, self).__init__(**kwargs)
self.W_k = nn.Linear(key_size, num_hiddens, bias=False)
self.W_q = nn.Linear(query_size, num_hiddens, bias=False)
self.w_v = nn.Linear(num_hiddens, 1, bias=False)
self.dropout = nn.Dropout(dropout)

def forward(self, queries, keys, values, valid_lens):
queries, keys = self.W_q(queries), self.W_k(keys)
features = queries.unsqueeze(2) + keys.unsqueeze(1)
features = torch.tanh(features)
scores = self.w_v(features).squeeze(-1)
self.attention_weights = masked_softmax(scores, valid_lens)
return torch.bmm(self.dropout(self.attention_weights),
values)

class Seq2SeqEncoder(nn.Module):
```

```
def __init__(self, vocab_size, embed_size, num_hiddens,
num_layers, dropout=0.1):
super(Seq2SeqEncoder, self).__init__()
self.embedding = nn.Embedding(vocab_size, embed_size)
self.rnn = nn.GRU(embed_size, num_hiddens, num_layers,
dropout=dropout)

def forward(self, X):
# 确保 X 具有正确的形状：(seq_len, batch_size)
if X.dim() == 2:
# 如果 X 是(batch_size, seq_len)，则进行转置
X = X.transpose(0, 1)
# 应用嵌入并通过 RNN
X = self.embedding(X) # 现在 X 应该是(seq_len, batch_size,
embed_size)
outputs, hidden_state = self.rnn(X)
return outputs, hidden_state

class Seq2SeqAttentionDecoder(nn.Module):
def __init__(self, vocab_size, embed_size, num_hiddens,
num_layers, dropout=0, **kwargs):
super(Seq2SeqAttentionDecoder, self).__init__(**kwargs)
self.attention = AdditiveAttention(num_hiddens, num_hiddens,
num_hiddens, dropout)
self.embedding = nn.Embedding(vocab_size, embed_size)
self.rnn = nn.GRU(embed_size + num_hiddens, num_hiddens,
num_layers, dropout=dropout)
self.dense = nn.Linear(num_hiddens, vocab_size)
# 初始化 attention_weights 用于存储每个时间步的注意力权重
self.attention_weights = []

def init_state(self, enc_outputs, enc_valid_lens=None,
*args):
# 修正：正确处理编码器输出的元组
outputs, hidden_state = enc_outputs
return (outputs.permute(1, 0, 2), hidden_state,
enc_valid_lens)

def forward(self, X, state):
```

```
enc_outputs, hidden_state, enc_valid_lens = state
X = self.embedding(X).permute(1, 0, 2)
outputs, self._attention_weights = [], []
# 清空之前存储的注意力权重
self.attention_weights = []
for x in X:
query = torch.unsqueeze(hidden_state[-1], dim=1)
context = self.attention(query, enc_outputs, enc_outputs,
enc_valid_lens)
# 存储注意力权重 - 修改这里
self.attention_weights.append(self.attention.attention_we
ights)
x = torch.cat((context, torch.unsqueeze(x, dim=1)), dim=-1)
out, hidden_state = self.rnn(x.permute(1, 0, 2),
hidden_state)
outputs.append(out)
outputs = self.dense(torch.cat(outputs, dim=0))
return outputs.permute(1, 0, 2), [enc_outputs, hidden_state,
enc_valid_lens]

class EncoderDecoder(nn.Module):
def __init__(self, encoder, decoder):
super(EncoderDecoder, self).__init__()
self.encoder = encoder
self.decoder = decoder

def forward(self, X, Y, state=None):
enc_outputs, hidden_state = self.encoder(X)
if state is None:
state = self.decoder.init_state((enc_outputs, hidden_state),
enc_valid_lens=None)
output, state = self.decoder(Y, state=state)
return output, state

from tqdm import tqdm

def train_seq2seq(model, train_iter, lr, num_epochs,
tgt_vocab, device):
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
```

```
loss_fn = nn.CrossEntropyLoss(ignore_index=0) # Ignore
padding index
train_loss = []

for epoch in range(num_epochs):
model.train()
total_loss = 0.0
# Create progress bar using tqdm
progress_bar = tqdm(enumerate(train_iter),
total=len(train_iter),
desc=f"Epoch {epoch+1}/{num_epochs}",
ncols=100)
for batch_idx, (X, Y) in progress_bar:
X, Y = X.to(device), Y.to(device)
Y_input = Y[:, :-1] # Use first n-1 tokens of Y as input
Y_target = Y[:, 1:] # Use last n-1 tokens of Y as target
Y_hat, _ = model(X, Y_input)
loss = loss_fn(Y_hat.reshape(-1, Y_hat.shape[-1]),
Y_target.reshape(-1))
optimizer.zero_grad()
loss.backward()
torch.nn.utils.clip_grad_norm_(model.parameters(),
max_norm=1)
optimizer.step()

total_loss += loss.item()
# Update the loss displayed in the progress bar
progress_bar.set_postfix(loss=f"{loss.item():.4f}")

avg_loss = total_loss / len(train_iter)
train_loss.append(avg_loss)
print(f"Epoch {epoch + 1} completed, Average Loss:
{avg_loss:.4f}")
return train_loss

# 设备选择
device = torch.device("cuda" if torch.cuda.is_available()
else "cpu")
```

```
# 数据加载和预处理函数
# 2. 更新 sentence_to_indices 函数，正确处理 max_len
def sentence_to_indices(sentence, vocab, max_len=None):
indices = [vocab.get(word, vocab['<unk>']) for word in
sentence.split()]
if max_len is not None:
if len(indices) > max_len:
indices = indices[:max_len] # 截断
else:
indices = indices + [vocab['<pad>']] * (max_len - len(indices))
# 填充
return indices


def build_vocab(sentences, max_vocab_size=10000):
counter = Counter()
for sentence in sentences:
counter.update(sentence.split())
# 特殊标记放在前面
vocab = {'<pad>': 0, '<unk>': 1, '<bos>': 2, '<eos>': 3}
# 添加最常见的词
for word, _ in counter.most_common(max_vocab_size -
len(vocab)):
if word not in vocab: # 避免重复
vocab[word] = len(vocab)
return vocab

def pad_batch(batch):
src_batch, tgt_batch = zip(*batch)
# 添加<bos>和<eos>标记
src_batch = [[2] + seq + [3] for seq in src_batch]
tgt_batch = [[2] + seq + [3] for seq in tgt_batch]
# 计算最大长度
src_max_len = max(len(seq) for seq in src_batch)
tgt_max_len = max(len(seq) for seq in tgt_batch)
# 填充序列
src_padded = [seq + [0] * (src_max_len - len(seq)) for seq
in src_batch]
```

```
tgt_padded = [seq + [0] * (tgt_max_len - len(seq)) for seq
in tgt_batch]
src_tensor = torch.tensor(src_padded,
dtype=torch.long).to(device)
tgt_tensor = torch.tensor(tgt_padded,
dtype=torch.long).to(device)
return src_tensor, tgt_tensor

def load_data_from_file(file_path, batch_size, num_steps,
max_vocab_size=10000):
with open(file_path, 'r', encoding='utf-8') as f:
lines = f.readlines()

pairs = [line.strip().split('\t')[:2] for line in lines]
src_sentences = [pair[0] for pair in pairs]
tgt_sentences = [pair[1] for pair in pairs]

src_vocab = build_vocab(src_sentences, max_vocab_size)
tgt_vocab = build_vocab(tgt_sentences, max_vocab_size)

src_sentences_idx = [sentence_to_indices(sentence,
src_vocab, num_steps) for sentence in src_sentences]
tgt_sentences_idx = [sentence_to_indices(sentence,
tgt_vocab, num_steps) for sentence in tgt_sentences]

data = list(zip(src_sentences_idx, tgt_sentences_idx))
data_iter = DataLoader(data, batch_size=batch_size,
shuffle=True, collate_fn=pad_batch)
return src_vocab, tgt_vocab, data_iter

# 主程序
if __name__ == "__main__":
data_path = '/home/yyz/NNDL-Class/Project5/Data/fra-eng'
file_path = os.path.join(data_path, 'fra.txt')
# 超参数
batch_size = 1024
num_steps = 20
# 加载数据
```

```
src_vocab, tgt_vocab, train_iter =
load_data_from_file(file_path, batch_size, num_steps)
# 模型参数
embed_size, num_hiddens, num_layers, dropout = 64, 128, 2,
0.2
# 模型实例
encoder = Seq2SeqEncoder(len(src_vocab), embed_size,
num_hiddens, num_layers, dropout)
decoder = Seq2SeqAttentionDecoder(len(tgt_vocab),
embed_size, num_hiddens, num_layers, dropout)
net = EncoderDecoder(encoder, decoder)
net = net.to(device)
# 训练参数
lr = 0.001
num_epochs = 50
# 训练模型
train_loss = train_seq2seq(net, train_iter, lr, num_epochs,
tgt_vocab, device)
# 绘制训练损失曲线
plt.figure(figsize=(10, 5))
plt.plot(range(1, num_epochs+1), train_loss)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Training Loss Curve")
plt.grid(True)
plt.savefig('/home/yyz/NNDL-Class/Project5/Result/trainin
g_loss_curve.png')
# 可视化注意力权重并保存
sample_attention = decoder.attention_weights[0]
attention_matrix =
sample_attention.squeeze().detach().cpu().numpy()
# 使用 matplotlib 绘制热图并保存
plt.figure(figsize=(8, 6))
sns.heatmap(attention_matrix, cmap="YlGnBu", annot=False,
cbar=True)
plt.xlabel("Key positions")
plt.ylabel("Query positions")
plt.title("Attention Heatmap")
```

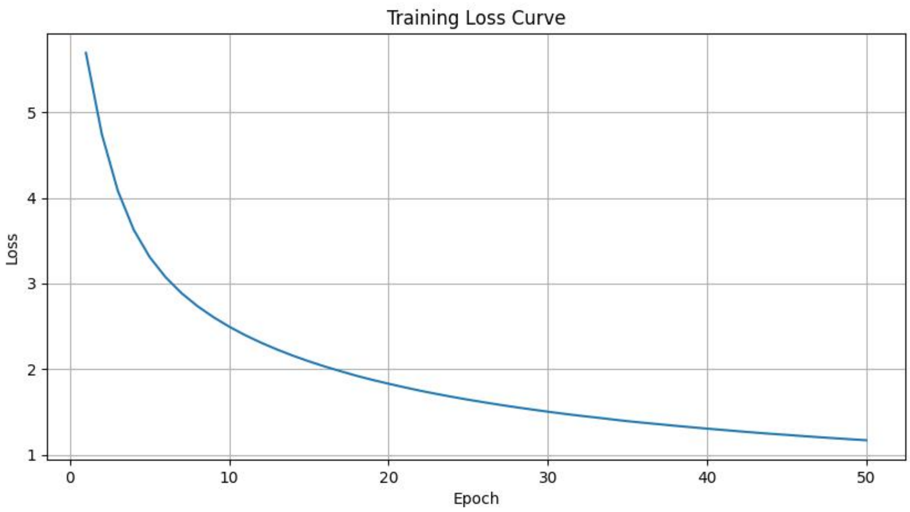
```python
plt.savefig('/home/yyz/NNDL-Class/Project5/Result/attenti
on_heatmap.png')
plt.close()
print("Training completed and results saved.")
```

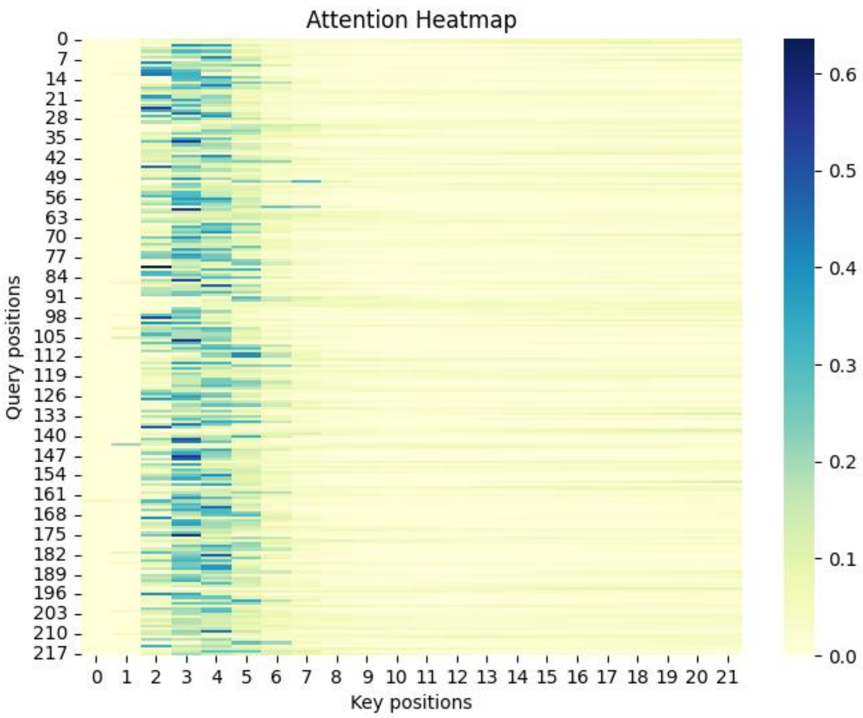
## 实验结果:

```
○ (yyzttt) (base) yyz@4028Dog:~$ /usr/local/anaconda3/envs/yyzttt/bin/python /home/yyz/NNDL-Class/Project
e/bahdanau.py
Epoch 1/50: 100%|██████████| 164/164 [00:11<00:00, 14.32it/s, loss=5.1669]
Epoch 1 completed, Average Loss: 5.6952
Epoch 2/50: 100%|██████████| 164/164 [00:11<00:00, 14.04it/s, loss=4.3026]
Epoch 2 completed, Average Loss: 4.7439
Epoch 3/50: 100%|██████████| 164/164 [00:11<00:00, 14.08it/s, loss=3.8358]
Epoch 3 completed, Average Loss: 4.0858
Epoch 4/50: 100%|██████████| 164/164 [00:11<00:00, 14.07it/s, loss=3.3969]
Epoch 4 completed, Average Loss: 3.6288
Epoch 5/50: 100%|██████████| 164/164 [00:11<00:00, 14.13it/s, loss=3.0768]
Epoch 5 completed, Average Loss: 3.3105
Epoch 6/50: 100%|██████████| 164/164 [00:11<00:00, 14.27it/s, loss=3.1373]
Epoch 6 completed, Average Loss: 3.0747
Epoch 7/50: 100%|██████████| 164/164 [00:11<00:00, 14.10it/s, loss=2.8496]
Epoch 7 completed, Average Loss: 2.8885
Epoch 8/50: 100%|██████████| 164/164 [00:11<00:00, 14.22it/s, loss=2.6250]
Epoch 8 completed, Average Loss: 2.7370
Epoch 9/50: 100%|██████████| 164/164 [00:11<00:00, 14.24it/s, loss=2.6410]
Epoch 9 completed, Average Loss: 2.6084
Epoch 10/50: 100%|██████████| 164/164 [00:11<00:00, 14.12it/s, loss=2.4476]
Epoch 10 completed, Average Loss: 2.4963
Epoch 11/50: 100%|██████████| 164/164 [00:11<00:00, 14.15it/s, loss=2.3577]
Epoch 11 completed, Average Loss: 2.3973
Epoch 12/50:  16%|██         | 26/164 [00:01<00:09, 14.40it/s, loss=2.3257]
```

PROBLEMS OUTPUT PORTS TERMINAL DEBUG CONSOLE

```
Epoch 38 completed, Average Loss: 1.3390
Epoch 39/50: 100%|██████████| 164/164 [00:11<00:00, 13.71it/s, loss=1.3362]
Epoch 39 completed, Average Loss: 1.3223
Epoch 40/50: 100%|██████████| 164/164 [00:12<00:00, 13.14it/s, loss=1.3262]
Epoch 40 completed, Average Loss: 1.3058
Epoch 41/50: 100%|██████████| 164/164 [00:12<00:00, 13.03it/s, loss=1.2450]
Epoch 41 completed, Average Loss: 1.2902
Epoch 42/50: 100%|██████████| 164/164 [00:11<00:00, 13.82it/s, loss=1.3358]
Epoch 42 completed, Average Loss: 1.2744
Epoch 43/50: 100%|██████████| 164/164 [00:11<00:00, 14.09it/s, loss=1.3213]
Epoch 43 completed, Average Loss: 1.2598
Epoch 44/50: 100%|██████████| 164/164 [00:11<00:00, 14.08it/s, loss=1.1939]
Epoch 44 completed, Average Loss: 1.2453
Epoch 45/50: 100%|██████████| 164/164 [00:11<00:00, 14.06it/s, loss=1.2988]
Epoch 45 completed, Average Loss: 1.2329
Epoch 46/50: 100%|██████████| 164/164 [00:11<00:00, 14.08it/s, loss=1.2427]
Epoch 46 completed, Average Loss: 1.2190
Epoch 47/50: 100%|██████████| 164/164 [00:11<00:00, 14.04it/s, loss=1.1363]
Epoch 47 completed, Average Loss: 1.2059
Epoch 48/50: 100%|██████████| 164/164 [00:11<00:00, 14.01it/s, loss=1.1614]
Epoch 48 completed, Average Loss: 1.1939
Epoch 49/50: 100%|██████████| 164/164 [00:11<00:00, 14.10it/s, loss=1.1263]
Epoch 49 completed, Average Loss: 1.1815
Epoch 50/50: 100%|██████████| 164/164 [00:11<00:00, 13.96it/s, loss=1.1827]
Epoch 50 completed, Average Loss: 1.1701
Training completed and results saved.
(yyzttt) (base) yyz@4028Dog:~$
```




# 3． 多头注意力：

**实验代码：**

```
import math
import torch
from torch import nn
import matplotlib.pyplot as plt
import numpy as np
```

```python
import os
# 点积注意力类（可以理解为每个头的注意力机制）
class DotProductAttention(nn.Module):
def __init__(self, dropout, **kwargs):
super(DotProductAttention, self).__init__(**kwargs)
self.dropout = nn.Dropout(dropout)

def forward(self, queries, keys, values, valid_lens=None):
d = queries.shape[-1]
scores = torch.bmm(queries, keys.transpose(1, 2)) /
math.sqrt(d)
self.attention_weights = masked_softmax(scores, valid_lens)
return torch.bmm(self.dropout(self.attention_weights),
values)

# 多头注意力类
class MultiHeadAttention(nn.Module):
def __init__(self, key_size, query_size, value_size,
num_hiddens,
num_heads, dropout, bias=False, **kwargs):
super(MultiHeadAttention, self).__init__(**kwargs)
self.num_heads = num_heads
self.attention = DotProductAttention(dropout)
self.W_q = nn.Linear(query_size, num_hiddens, bias=bias)
self.W_k = nn.Linear(key_size, num_hiddens, bias=bias)
self.W_v = nn.Linear(value_size, num_hiddens, bias=bias)
self.W_o = nn.Linear(num_hiddens, num_hiddens, bias=bias)

def forward(self, queries, keys, values, valid_lens):
# 将查询、键、值变换成多个头的形状
queries = transpose_qkv(self.W_q(queries), self.num_heads)
keys = transpose_qkv(self.W_k(keys), self.num_heads)
values = transpose_qkv(self.W_v(values), self.num_heads)

# 如果 valid_lens 不为空，需要重复 valid_lens，使其与头数匹配
if valid_lens is not None:
valid_lens = torch.repeat_interleave(valid_lens,
repeats=self.num_heads, dim=0)
```

```
# 通过注意力机制计算结果
output = self.attention(queries, keys, values, valid_lens)
# 拼接并通过线性层进行变换
output_concat = transpose_output(output, self.num_heads)
return self.W_o(output_concat)

# 转置查询、键、值的形状
def transpose_qkv(X, num_heads):
X = X.reshape(X.shape[0], X.shape[1], num_heads, -1)
X = X.permute(0, 2, 1, 3)
return X.reshape(-1, X.shape[2], X.shape[3])

# 转置输出的形状
def transpose_output(X, num_heads):
X = X.reshape(-1, num_heads, X.shape[1], X.shape[2])
X = X.permute(0, 2, 1, 3)
return X.reshape(X.shape[0], X.shape[1], -1)

# Masked softmax 操作
def masked_softmax(X, valid_lens):
if valid_lens is None:
return nn.functional.softmax(X, dim=-1)
else:
shape = X.shape
valid_lens = valid_lens.reshape(-1)
X = sequence_mask(X.reshape(-1, shape[-1]), valid_lens,
value=-1e6)
return nn.functional.softmax(X.reshape(shape), dim=-1)

def sequence_mask(X, valid_lens, value):
# 遮掩多余的部分
for i, length in enumerate(valid_lens):
X[i, length:] = value
return X



# 创建保存路径
```

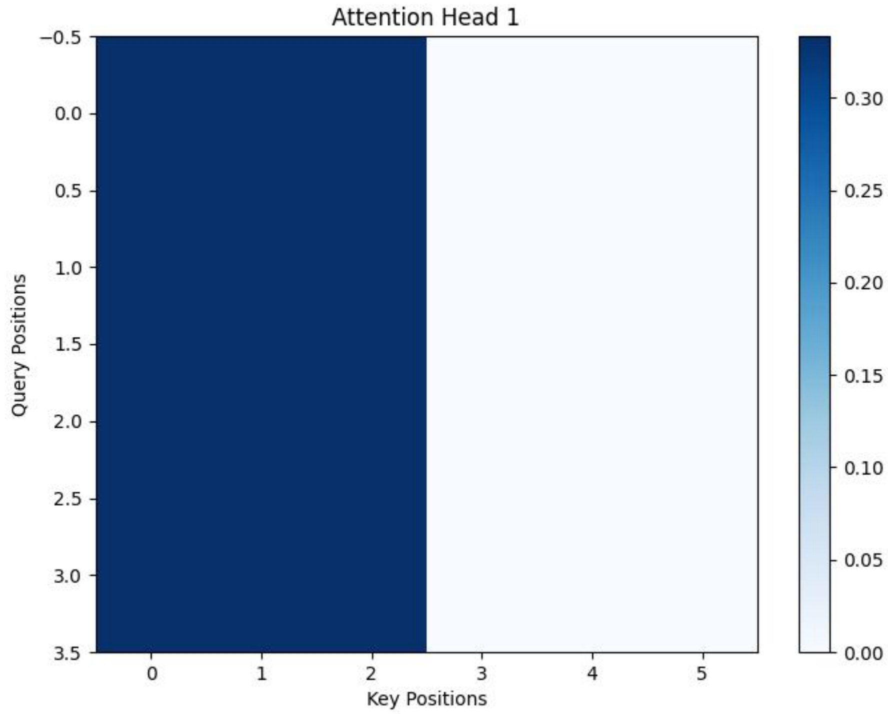
```
save_path =
'/home/yyz/NNDL-Class/Project5/Result/attention_weights/'
if not os.path.exists(save_path):
os.makedirs(save_path)

def save_attention_weights(attention_weights, num_heads,
num_queries, num_kvpairs):
# 画出每个头的注意力权重并保存
for i in range(num_heads):
plt.figure(figsize=(8, 6))
plt.imshow(attention_weights[i].detach().cpu().numpy(),
cmap='Blues', aspect='auto')
plt.colorbar()
plt.title(f"Attention Head {i+1}")
plt.xlabel('Key Positions')
plt.ylabel('Query Positions')
# 保存图像到文件
file_path = os.path.join(save_path,
f"attention_head_{i+1}.png")
plt.savefig(file_path)
plt.close() # 关闭当前图像，以释放内存

# 修改主函数，加入提取并保存多个头的注意力权重
def main():
# 设置参数
batch_size = 2
num_queries = 4
num_kvpairs = 6
num_hiddens = 10
num_heads = 5
valid_lens = torch.tensor([3, 2]) # 有效长度
queries = torch.ones((batch_size, num_queries,
num_hiddens))
keys = torch.ones((batch_size, num_kvpairs, num_hiddens))
values = torch.ones((batch_size, num_kvpairs, num_hiddens))

# 创建多头注意力模型
attention = MultiHeadAttention(key_size=num_hiddens,
query_size=num_hiddens, value_size=num_hiddens,
```

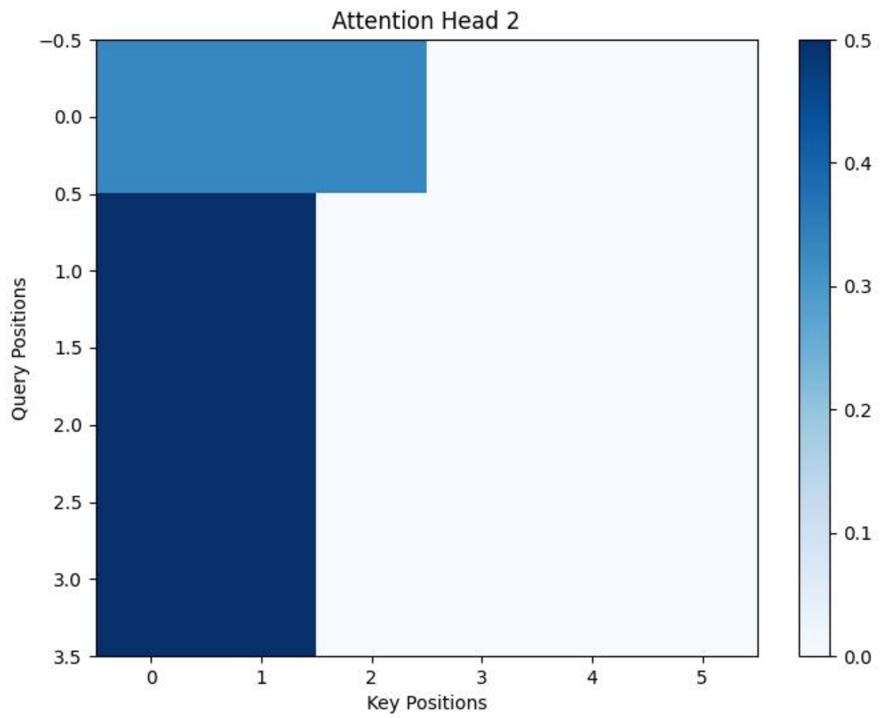
```
num_hiddens=num_hiddens, num_heads=num_heads, dropout=0.5)
# 执行前向传播
output = attention(queries, keys, values, valid_lens)

# 打印结果
print("Attention output shape:", output.shape) # 输出的形状应该是 (batch_size, num_queries, num_hiddens)
print("Output:", output)

# 提取并保存每个头的注意力权重
attention_weights = attention.attention.attention_weights # 获取注意力权重
print("Attention weights:", attention_weights.shape)

# 保存每个头的注意力权重
save_attention_weights(attention_weights, num_heads, num_queries, num_kvpairs)

if __name__ == "__main__":
main()
```

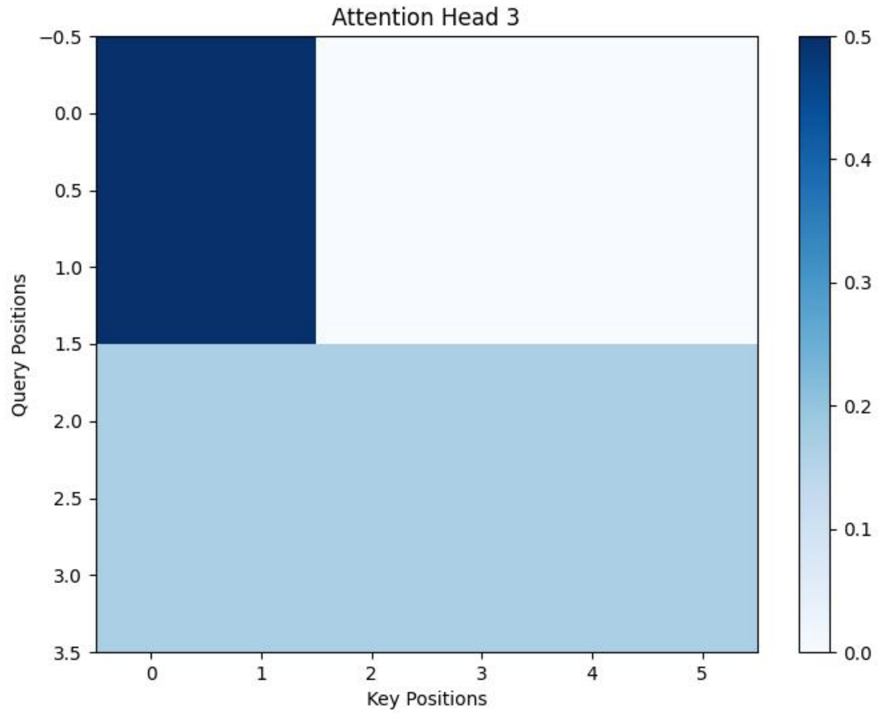
## 实验结果：

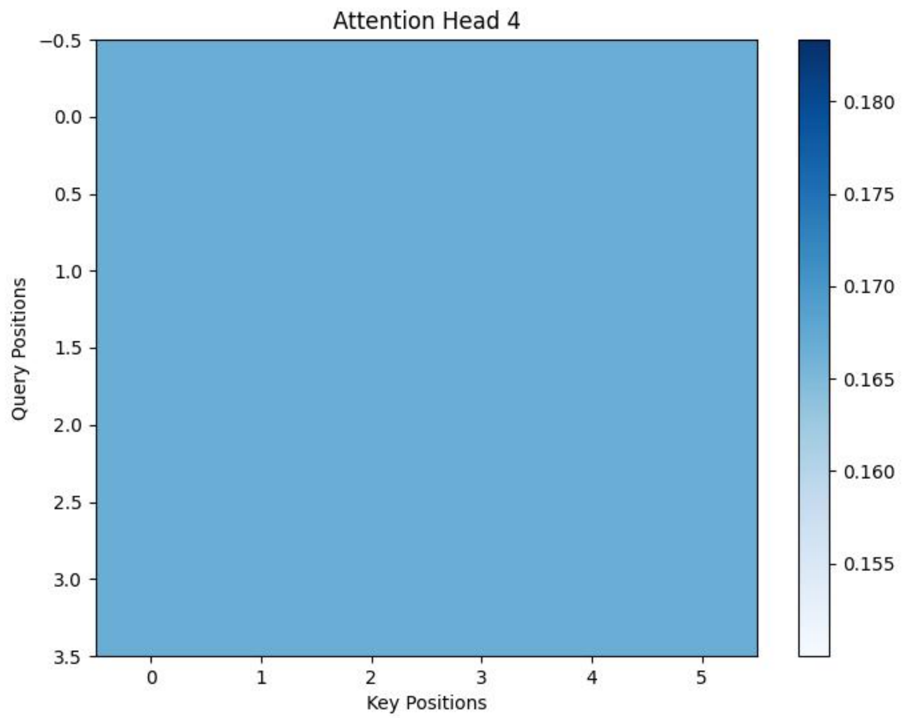
```
● (yyzttt) (base) yyz@4028Dog:~$ /usr/local/anaconda3/envs/yyzttt/bin/python /home/yyz/NNDL-
  Attention output shape: torch.Size([2, 4, 10])
  Output: tensor([[[ 4.4579e-01, -1.2937e-01, -4.3417e-01,  2.7227e-01,  3.0336e-01,
             4.7239e-03, -1.2638e-01, -7.0270e-02,  1.2538e-01, -5.9725e-01],
           [ 3.2402e-01, -9.1568e-02, -3.7716e-01,  1.0380e-01,  3.6536e-01,
            -1.3575e-01, -3.7915e-01,  2.1609e-01,  2.1363e-01, -9.0696e-01],
           [ 1.6611e-01, -3.4987e-02,  3.9717e-02,  7.7363e-02,  9.6761e-02,
             1.8670e-02,  3.2863e-03, -2.9638e-02,  3.1423e-02, -1.2411e-01],
           [ 5.3739e-02, -2.3831e-01, -2.1004e-01, -3.0639e-02,  4.9592e-01,
            -1.6406e-01,  3.4050e-01, -3.9032e-01,  3.0338e-01, -6.7988e-01]],

          [[ 1.6318e-01, -2.6727e-01, -2.9524e-01,  1.6144e-02,  6.4078e-01,
            -1.9405e-01,  2.7678e-01, -3.6336e-01,  3.7776e-01, -9.9215e-01],
           [ 8.0314e-02, -2.7115e-01, -4.1288e-01, -1.3832e-02,  5.6381e-01,
            -2.0555e-01,  2.7246e-01, -3.7779e-01,  3.5243e-01, -8.7127e-01],
           [ 1.0948e-01, -1.9394e-01, -2.4395e-01, -6.6971e-04,  4.5795e-01,
            -1.5518e-01,  1.6098e-01, -2.3271e-01,  2.7758e-01, -7.3394e-01],
           [ 5.5257e-01, -4.2317e-02, -1.9312e-01,  3.0932e-01,  2.4378e-01,
             5.8516e-02, -2.9551e-01,  1.3562e-01,  6.5762e-02, -5.6468e-01]]],
         grad_fn=<UnsafeViewBackward>)
  Attention weights: torch.Size([10, 4, 6])
```

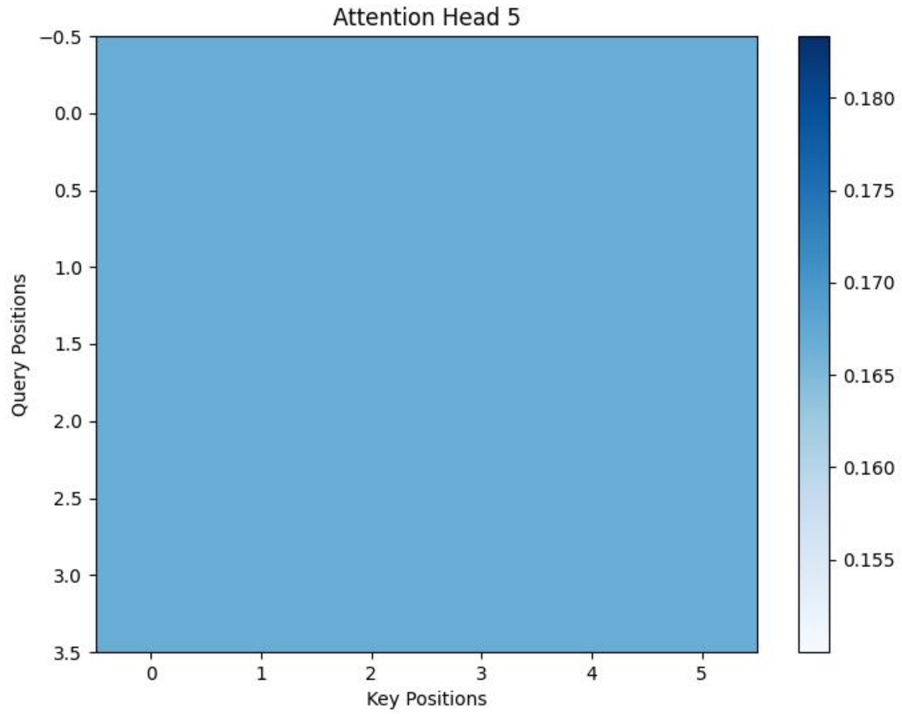
练习题 1：分别可视化这个实验中的多个头的注意力权重


Attention Head 1
Query Positions
Key Positions
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
1
2
3
4
5
0.30
0.25
0.20
0.15
0.10
0.05
0.00


Attention Head 2
Query Positions
Key Positions
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
1
2
3
4
5
0.5
0.4
0.3
0.2
0.1
0.0


Attention Head 3
Query Positions
Key Positions
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
1
2
3
4
5
0.5
0.4
0.3
0.2
0.1
0.0


Attention Head 4
Query Positions
Key Positions
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
1
2
3
4
5
0.180
0.175
0.170
0.165
0.160
0.155




# 4. 自注意力和位置编码:

## 实验代码:

```
import torch
import torch.nn as nn
import math

class PositionalEncoding(nn.Module):
"""位置编码"""
def __init__(self, num_hiddens, dropout, max_len=1000):
super(PositionalEncoding, self).__init__()
self.dropout = nn.Dropout(dropout)
# 创建一个足够长的 P
self.P = torch.zeros((1, max_len, num_hiddens))
X = torch.arange(max_len, dtype=torch.float32).reshape(-1,
1) / torch.pow(10000, torch.arange(0, num_hiddens, 2,
dtype=torch.float32) / num_hiddens)
self.P[:, :, 0::2] = torch.sin(X)
self.P[:, :, 1::2] = torch.cos(X)

def forward(self, X):
```

```python
X = X + self.P[:, :X.shape[1], :].to(X.device)
return self.dropout(X)

class DotProductAttention(nn.Module):
"""缩放点积注意力"""
def __init__(self, dropout, **kwargs):
super(DotProductAttention, self).__init__(**kwargs)
self.dropout = nn.Dropout(dropout)

def forward(self, queries, keys, values, valid_lens=None):
d = queries.shape[-1]
scores = torch.bmm(queries, keys.transpose(1, 2)) /
math.sqrt(d)
attention_weights = self.masked_softmax(scores, valid_lens)
return torch.bmm(self.dropout(attention_weights), values)

def masked_softmax(self, X, valid_lens):
"""在最后一个维度进行 softmax，并处理 mask"""
if valid_lens is None:
return nn.functional.softmax(X, dim=-1)
shape = X.shape
if valid_lens.dim() == 1:
valid_lens = torch.repeat_interleave(valid_lens, shape[1])
else:
valid_lens = valid_lens.reshape(-1)
X = self.sequence_mask(X.reshape(-1, shape[-1]), valid_lens,
value=-1e6)
return nn.functional.softmax(X.reshape(shape), dim=-1)

def sequence_mask(self, X, valid_lens, value=0):
"""生成有效长度的 mask"""
max_len = X.shape[1]
mask = torch.arange(max_len, device=X.device)[None, :] <
valid_lens[:, None]
X[~mask] = value
return X


class MultiHeadAttention(nn.Module):
```

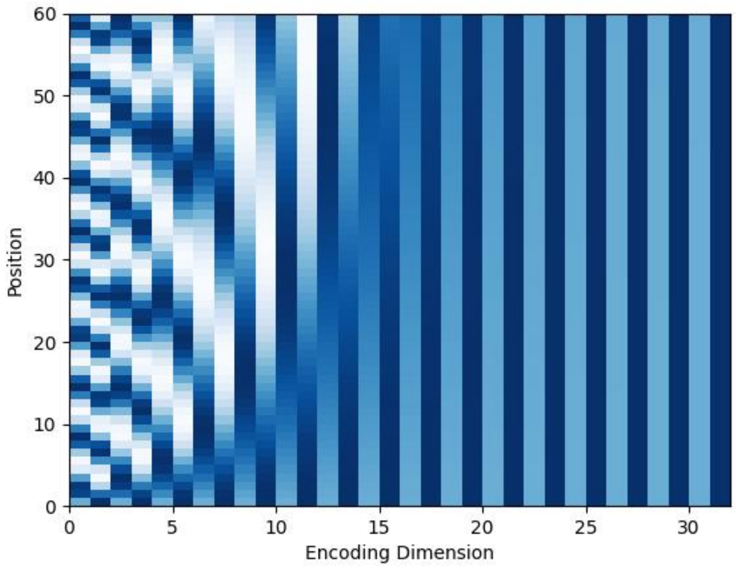
```python
"""多头注意力"""
def __init__(self, key_size, query_size, value_size,
num_hiddens, num_heads, dropout, **kwargs):
super(MultiHeadAttention, self).__init__(**kwargs)
self.num_heads = num_heads
self.attention = DotProductAttention(dropout)
self.W_q = nn.Linear(query_size, num_hiddens, bias=False)
self.W_k = nn.Linear(key_size, num_hiddens, bias=False)
self.W_v = nn.Linear(value_size, num_hiddens, bias=False)
self.W_o = nn.Linear(num_hiddens, num_hiddens, bias=False)

def forward(self, queries, keys, values, valid_lens):
queries = self.transpose_qkv(self.W_q(queries))
keys = self.transpose_qkv(self.W_k(keys))
values = self.transpose_qkv(self.W_v(values))

if valid_lens is not None:
valid_lens = torch.repeat_interleave(valid_lens,
repeats=self.num_heads, dim=0)

output = self.attention(queries, keys, values, valid_lens)
output_concat = self.transpose_output(output)
return self.W_o(output_concat)

def transpose_qkv(self, X):
"""为多头注意力计算变换"""
X = X.reshape(X.shape[0], X.shape[1], self.num_heads, -1)
X = X.permute(0, 2, 1, 3)
return X.reshape(-1, X.shape[2], X.shape[3])

def transpose_output(self, X):
"""逆转 transpose_qkv 的操作"""
X = X.reshape(-1, self.num_heads, X.shape[1], X.shape[2])
X = X.permute(0, 2, 1, 3)
return X.reshape(X.shape[0], X.shape[1], -1)

# 测试位置编码
num_hiddens, num_steps = 32, 60
pos_encoding = PositionalEncoding(num_hiddens, 0)
```

```
pos_encoding.eval()
X = pos_encoding(torch.zeros((1, num_steps, num_hiddens)))
P = pos_encoding.P[:, :X.shape[1], :]

# 可视化位置编码
import matplotlib.pyplot as plt
plt.pcolormesh(P[0].cpu().detach().numpy(), cmap='Blues')
plt.xlabel('Encoding Dimension')
plt.ylabel('Position')
plt.show()

# 测试多头注意力
num_hiddens, num_heads = 100, 5
attention = MultiHeadAttention(num_hiddens, num_hiddens,
num_hiddens, num_hiddens, num_heads, 0.5)
attention.eval()

# 输入数据
batch_size, num_queries = 2, 4
num_kvpairs, valid_lens = 6, torch.tensor([3, 2])
X = torch.ones((batch_size, num_queries, num_hiddens))
Y = torch.ones((batch_size, num_kvpairs, num_hiddens))

# 计算注意力
attention(X, Y, Y, valid_lens).shape # 输出形状应为
(batch_size, num_queries, num_hiddens)
# 测试位置编码
num_hiddens, num_steps = 32, 60
pos_encoding = PositionalEncoding(num_hiddens, 0)
pos_encoding.eval()
X = pos_encoding(torch.zeros((1, num_steps, num_hiddens)))
P = pos_encoding.P[:, :X.shape[1], :]

# 保存位置编码的图像
import matplotlib.pyplot as plt
save_path =
'/home/yyz/NNDL-Class/Project5/Result/positional_encoding.
png'
plt.pcolormesh(P[0].cpu().detach().numpy(), cmap='Blues')
```

```python
plt.xlabel('Encoding Dimension')
plt.ylabel('Position')
plt.savefig(save_path)
plt.close()

print(f"The position encoded image has been saved to
{save_path}")

# 测试多头注意力
num_hiddens, num_heads = 100, 5
attention = MultiHeadAttention(num_hiddens, num_hiddens,
num_hiddens, num_hiddens, num_heads, 0.5)
attention.eval()

# 输入数据
batch_size, num_queries = 2, 4
num_kvpairs, valid_lens = 6, torch.tensor([3, 2])
X = torch.ones((batch_size, num_queries, num_hiddens))
Y = torch.ones((batch_size, num_kvpairs, num_hiddens))

# 计算注意力
output = attention(X, Y, Y, valid_lens)

# 打印输出的形状和输出的内容
print("Output shape:", output.shape) # 输出形状应为
(batch_size, num_queries, num_hiddens)
print("Output content:", output)
```

**实验结果:**

```
● (yyzttt) (base) yyz@4028Dog:~$ /usr/local/anaconda3/envs/yyzttt/bin/python /home/yyz/NNDL-Class/Project5/Code/selfatt.py
The position encoded image has been saved to /home/yyz/NNDL-Class/Project5/Result/positional_encoding.png
Output shape: torch.Size([2, 4, 100])
Output content: tensor([[[-0.2550, -0.3790,  0.2938, -0.6658, -0.1096, -0.6389, -0.1976,
          -0.9012, -0.1067, -0.0541, -0.3499, -0.2902, -0.4157,  0.1697,
           0.1686, -0.0033,  0.9807, -0.2033, -0.0560,  0.4771, -0.2777,
          -0.2180,  0.1603, -0.0499,  0.3426,  0.5121, -0.0499, -0.1187,
           0.0982,  0.2188, -0.3179, -0.1278, -0.1614, -0.1162, -0.2590,
          -0.3989,  0.6534,  0.2709, -0.0672,  0.0418, -0.2353, -0.0287,
           0.3371,  0.5318,  0.1776, -0.0435, -0.2118, -0.1250, -0.1861,
          -0.5599, -0.0222,  0.0622,  0.4311, -0.2154,  0.7011,  0.6354,
           0.4413, -0.5849, -0.2930,  0.1095,  0.0285, -0.2073,  0.1436,
          -0.0922, -0.6582,  0.1433,  0.2647, -0.1944,  0.0921, -0.0254,
           0.2128, -0.2987,  0.0354, -0.5342,  0.0913, -0.3020, -0.2471,
          -0.1294,  0.1627, -0.4203, -0.3279, -0.4760,  0.3095,  0.4121,
           0.5203,  0.1789, -0.1157, -0.0492, -0.2305,  0.0337, -0.1960,
           0.2846,  0.0672, -0.0648, -0.4886, -0.1379,  0.0931,  0.1353,
           0.0992,  0.3334],
         [-0.2550, -0.3790,  0.2938, -0.6658, -0.1096, -0.6389, -0.1976,
          -0.9012, -0.1067, -0.0541, -0.3499, -0.2902, -0.4157,  0.1697,
           0.1686, -0.0033,  0.9807, -0.2033, -0.0560,  0.4771, -0.2777,
          -0.2180,  0.1603, -0.0499,  0.3426,  0.5121, -0.0499, -0.1187,
           0.0982,  0.2188, -0.3179, -0.1278, -0.1614, -0.1162, -0.2590,
```




## 练习题 2：设计一种可学习的位置编码方法

### 实验代码：

```
import torch
import torch.nn as nn

class LearnablePositionalEncoding(nn.Module):
def __init__(self, num_positions, embedding_dim):
super(LearnablePositionalEncoding, self).__init__()
# 初始化位置编码参数，每个位置有一个可学习的向量
self.positional_embeddings = nn.Embedding(num_positions,
embedding_dim)

def forward(self, X):
# 获取序列长度
```

```
seq_len = X.size(1)
# 创建位置索引 (0, 1, ..., seq_len-1)
position_indices = torch.arange(seq_len,
device=X.device).unsqueeze(0).repeat(X.size(0), 1)
# 获取对应位置的编码
position_encoding =
self.positional_embeddings(position_indices)
# 将位置编码加到输入张量
return X + position_encoding

# 测试可学习的位置编码
batch_size, seq_len, embedding_dim = 2, 60, 32
learnable_pos_encoding =
LearnablePositionalEncoding(seq_len, embedding_dim)
learnable_pos_encoding.eval()

# 创建一个形状为 (batch_size, seq_len, embedding_dim) 的输入
X = torch.zeros((batch_size, seq_len, embedding_dim))

# 获取加入了可学习位置编码后的结果
output = learnable_pos_encoding(X)

# 打印输出的形状和内容
print("Output shape:", output.shape) # 输出形状应为
(batch_size, num_queries, num_hiddens)
print("Output content:", output)
```

**实验结果:**

```
(yyzttt) (base) yyz@4028Dog:~$ /usr/local/anaconda3/envs/yyzttt/bin/python /home/yyz/NNDL-C
ode/learnablemed.py
Output shape: torch.Size([2, 60, 32])
Output content: tensor([[[ 1.9055,  1.4798,  0.9684,  ..., -0.9757,  1.0140, -0.8460],
         [-0.2960, -1.6690,  1.0214,  ...,  0.0967, -0.1574,  0.7765],
         [ 0.8094, -2.1131,  0.5446,  ..., -0.6088, -0.0564, -1.8890],
         ...,
         [-0.7547,  0.3445, -3.0775,  ...,  1.3791, -0.6438,  0.6527],
         [ 0.8050, -0.7289, -1.1155,  ..., -0.3995, -1.0970, -0.3787],
         [ 0.7451,  0.3193,  0.4686,  ..., -0.4197,  0.0529, -0.9508]],

        [[ 1.9055,  1.4798,  0.9684,  ..., -0.9757,  1.0140, -0.8460],
         [-0.2960, -1.6690,  1.0214,  ...,  0.0967, -0.1574,  0.7765],
         [ 0.8094, -2.1131,  0.5446,  ..., -0.6088, -0.0564, -1.8890],
         ...,
         [-0.7547,  0.3445, -3.0775,  ...,  1.3791, -0.6438,  0.6527],
         [ 0.8050, -0.7289, -1.1155,  ..., -0.3995, -1.0970, -0.3787],
         [ 0.7451,  0.3193,  0.4686,  ..., -0.4197,  0.0529, -0.9508]]],
       grad_fn=<AddBackward0>)
```

## 5. Transformer：

### 实验代码:

```
import torch
from torch import nn
import math
import pandas as pd
import matplotlib.pyplot as plt
from d2l import torch as d2l
from bahdanau import load_data_from_file
import os
os.environ["CUDA_VISIBLE_DEVICES"] = "2"

class EncoderDecoder(nn.Module):
def __init__(self, encoder, decoder):
super(EncoderDecoder, self).__init__()
self.encoder = encoder
self.decoder = decoder

def forward(self, X, Y, enc_valid_lens):
encoder_output = self.encoder(X, enc_valid_lens)
state = self.decoder.init_state(encoder_output,
enc_valid_lens)
output, state = self.decoder(Y, state)
return output, state

class PositionWiseFFN(nn.Module):
"""基于位置的前馈网络"""
```

```
def __init__(self, ffn_num_input, ffn_num_hiddens,
ffn_num_outputs, **kwargs):
super(PositionWiseFFN, self).__init__(**kwargs)
self.dense1 = nn.Linear(ffn_num_input, ffn_num_hiddens)
self.relu = nn.ReLU()
self.dense2 = nn.Linear(ffn_num_hiddens, ffn_num_outputs)

def forward(self, X):
return self.dense2(self.relu(self.dense1(X)))

class AddNorm(nn.Module):
"""残差连接后进行层规范化"""
def __init__(self, normalized_shape, dropout, **kwargs):
super(AddNorm, self).__init__(**kwargs)
self.dropout = nn.Dropout(dropout)
self.ln = nn.LayerNorm(normalized_shape)

def forward(self, X, Y):
return self.ln(self.dropout(Y) + X)

class EncoderBlock(nn.Module):
"""Transformer 编码器块"""
def __init__(self, key_size, query_size, value_size,
num_hiddens, norm_shape, ffn_num_input, ffn_num_hiddens,
num_heads, dropout, use_bias=False, **kwargs):
super(EncoderBlock, self).__init__(**kwargs)
self.attention = MultiHeadAttention(key_size, query_size,
value_size, num_hiddens, num_heads, dropout, use_bias)
self.addnorm1 = AddNorm(norm_shape, dropout)
self.ffn = PositionWiseFFN(ffn_num_input, ffn_num_hiddens,
num_hiddens)
self.addnorm2 = AddNorm(norm_shape, dropout)

def forward(self, X, valid_lens):
Y = self.addnorm1(X, self.attention(X, X, X, valid_lens))
return self.addnorm2(Y, self.ffn(Y))

def transpose_qkv(X, num_heads):
"""为了多头注意力头的并行计算而变换形状"""
```

```
X = X.reshape(X.shape[0], X.shape[1], num_heads, -1)
X = X.permute(0, 2, 1, 3)
return X.reshape(-1, X.shape[2], X.shape[3])

def transpose_output(X, num_heads):
"""逆转 transpose_qkv 函数的操作"""
X = X.reshape(-1, num_heads, X.shape[1], X.shape[2])
X = X.permute(0, 2, 1, 3)
return X.reshape(X.shape[0], X.shape[1], -1)

class DotProductAttention(nn.Module):
"""缩放点积注意力"""
def __init__(self, dropout=0.0, **kwargs):
super(DotProductAttention, self).__init__(**kwargs)
self.dropout = nn.Dropout(dropout)
self.attention_weights = None # 初始化属性

def forward(self, queries, keys, values, valid_lens=None):
"""查询、键和值，valid_lens 用于掩蔽"""
d = queries.shape[-1] # 查询的最后一维是 d
# 计算缩放点积
scores = torch.bmm(queries, keys.transpose(1, 2)) /
math.sqrt(d)

# 对 scores 进行 softmax，得到注意力权重
attention_weights = masked_softmax(scores, valid_lens)

# 保存 attention_weights 到成员变量，方便外部访问
self.attention_weights = attention_weights

# 计算加权和
return torch.bmm(self.dropout(attention_weights), values)

def masked_softmax(X, valid_lens):
"""计算 softmax 并对无效位置进行掩蔽"""
if valid_lens is None:
return nn.functional.softmax(X, dim=-1)
else:
shape = X.shape
```

```
if valid_lens.dim() == 1:
valid_lens = torch.repeat_interleave(valid_lens, shape[1])
else:
valid_lens = valid_lens.reshape(-1)
# 对超出有效长度的位置赋予非常大的负值，使其 softmax 输出为 0
X = sequence_mask(X.reshape(-1, shape[-1]), valid_lens,
value=-1e6)
return nn.functional.softmax(X.reshape(shape), dim=-1)

def sequence_mask(X, valid_lens, value=0):
"""给定有效长度，掩蔽无效部分"""
max_len = X.shape[1]
mask = torch.arange(max_len,
device=X.device).expand(len(valid_lens), max_len) <
valid_lens.unsqueeze(1)
X[~mask] = value
return X

class MultiHeadAttention(nn.Module):
"""多头注意力"""
def __init__(self, key_size, query_size, value_size,
num_hiddens,
num_heads, dropout, bias=False, **kwargs):
super(MultiHeadAttention, self).__init__(**kwargs)
self.num_heads = num_heads
self.attention = DotProductAttention(dropout)
# 定义查询、键和值的线性映射
self.W_q = nn.Linear(query_size, num_hiddens, bias=bias)
self.W_k = nn.Linear(key_size, num_hiddens, bias=bias)
self.W_v = nn.Linear(value_size, num_hiddens, bias=bias)
# 输出的线性变换
self.W_o = nn.Linear(num_hiddens, num_hiddens, bias=bias)

def forward(self, queries, keys, values, valid_lens):
# 将输入的查询、键和值进行线性变换
queries = transpose_qkv(self.W_q(queries), self.num_heads)
keys = transpose_qkv(self.W_k(keys), self.num_heads)
values = transpose_qkv(self.W_v(values), self.num_heads)
```

```
if valid_lens is not None:
# 将 valid_lens 进行复制以适应 num_heads 的大小
valid_lens = torch.repeat_interleave(valid_lens,
repeats=self.num_heads, dim=0)

# 计算多头注意力
output = self.attention(queries, keys, values, valid_lens)
# 将输出拼接在一起并通过输出的线性变换
output_concat = transpose_output(output, self.num_heads)
return self.W_o(output_concat)

class DecoderBlock(nn.Module):
"""解码器中第 i 个块"""
def __init__(self, key_size, query_size, value_size,
num_hiddens, norm_shape, ffn_num_input, ffn_num_hiddens,
num_heads, dropout, i, **kwargs):
super(DecoderBlock, self).__init__(**kwargs)
self.i = i
self.attention1 = MultiHeadAttention(key_size, query_size,
value_size, num_hiddens, num_heads, dropout)
self.addnorm1 = AddNorm(norm_shape, dropout)
self.attention2 = MultiHeadAttention(key_size, query_size,
value_size, num_hiddens, num_heads, dropout)
self.addnorm2 = AddNorm(norm_shape, dropout)
self.ffn = PositionWiseFFN(ffn_num_input, ffn_num_hiddens,
num_hiddens)
self.addnorm3 = AddNorm(norm_shape, dropout)

def forward(self, X, state):
enc_outputs, enc_valid_lens = state[0], state[1]
if state[2][self.i] is None:
key_values = X
else:
key_values = torch.cat((state[2][self.i], X), axis=1)
state[2][self.i] = key_values

if self.training:
batch_size, num_steps, _ = X.shape
```

```
dec_valid_lens = torch.arange(1, num_steps + 1,
device=X.device).repeat(batch_size, 1)
else:
dec_valid_lens = None

X2 = self.attention1(X, key_values, key_values,
dec_valid_lens)
Y = self.addnorm1(X, X2)
Y2 = self.attention2(Y, enc_outputs, enc_outputs,
enc_valid_lens)
Z = self.addnorm2(Y, Y2)
return self.addnorm3(Z, self.ffn(Z)), state

class TransformerEncoder(d2l.Encoder):
"""Transformer编码器"""
def __init__(self, vocab_size, key_size, query_size,
value_size, num_hiddens, norm_shape, ffn_num_input,
ffn_num_hiddens, num_heads, num_layers, dropout,
use_bias=False, **kwargs):
super(TransformerEncoder, self).__init__(**kwargs)
self.num_hiddens = num_hiddens
self.embedding = nn.Embedding(vocab_size, num_hiddens)
self.pos_encoding = PositionalEncoding(num_hiddens,
dropout)
self.blks = nn.Sequential()
for i in range(num_layers):
self.blks.add_module("block" + str(i),
EncoderBlock(key_size, query_size, value_size, num_hiddens,
norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
dropout, use_bias))

def forward(self, X, valid_lens, *args):
X = self.pos_encoding(self.embedding(X) *
math.sqrt(self.num_hiddens))
self.attention_weights = [None] * len(self.blks)
for i, blk in enumerate(self.blks):
X = blk(X, valid_lens)
self.attention_weights[i] =
blk.attention.attention.attention_weights
```

```
return X

class TransformerDecoder(d2l.AttentionDecoder):
"""Transformer解码器"""
def __init__(self, vocab_size, key_size, query_size,
value_size, num_hiddens, norm_shape, ffn_num_input,
ffn_num_hiddens, num_heads, num_layers, dropout, **kwargs):
super(TransformerDecoder, self).__init__(**kwargs)
self.num_hiddens = num_hiddens
self.num_layers = num_layers
self.embedding = nn.Embedding(vocab_size, num_hiddens)
self.pos_encoding = PositionalEncoding(num_hiddens,
dropout)
self.blks = nn.Sequential()
for i in range(num_layers):
self.blks.add_module("block" + str(i),
DecoderBlock(key_size, query_size, value_size, num_hiddens,
norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
dropout, i))
self.dense = nn.Linear(num_hiddens, vocab_size)

def init_state(self, enc_outputs, enc_valid_lens, *args):
return [enc_outputs, enc_valid_lens, [None] *
self.num_layers]

def forward(self, X, state):
X = self.pos_encoding(self.embedding(X) *
math.sqrt(self.num_hiddens))
self._attention_weights = [[None] * len(self.blks) for _ in
range(2)]
for i, blk in enumerate(self.blks):
X, state = blk(X, state)
self._attention_weights[0][i] =
blk.attention1.attention.attention_weights
self._attention_weights[1][i] =
blk.attention2.attention.attention_weights
return self.dense(X), state

@property
```

```python
def attention_weights(self):
return self._attention_weights

class PositionalEncoding(nn.Module):
"""位置编码"""
def __init__(self, num_hiddens, dropout, max_len=1000):
super(PositionalEncoding, self).__init__()
self.dropout = nn.Dropout(dropout)
# 创建一个足够长的 P
self.P = torch.zeros((1, max_len, num_hiddens))
X = torch.arange(max_len, dtype=torch.float32).reshape(
-1, 1) / torch.pow(10000, torch.arange(
0, num_hiddens, 2, dtype=torch.float32) / num_hiddens)
self.P[:, :, 0::2] = torch.sin(X)
self.P[:, :, 1::2] = torch.cos(X)

def forward(self, X):
X = X + self.P[:, :X.shape[1], :].to(X.device)
return self.dropout(X)

class MaskedSoftmaxCELoss(nn.CrossEntropyLoss):
def forward(self, pred, label, valid_len):
weights = torch.ones_like(label)
mask = torch.arange(label.shape[1],
device=label.device)[None, :] < valid_len[:, None]
weights = weights * mask
self.reduction = 'none'
unweighted_loss = super(MaskedSoftmaxCELoss, self).forward(
pred.permute(0, 2, 1), label)
weighted_loss = (unweighted_loss * weights).sum(dim=1) /
valid_len
return weighted_loss

from tqdm import tqdm

def train_seq2seq_custom(net, data_iter, lr, num_epochs,
tgt_vocab, device):
def xavier_init_weights(m):
if isinstance(m, nn.Linear):
```

```
nn.init.xavier_uniform_(m.weight)
elif isinstance(m, nn.GRU):
for param in m._flat_weights_names:
if "weight" in param:
nn.init.xavier_uniform_(m._parameters[param])
net.apply(xavier_init_weights)
net.to(device)
optimizer = torch.optim.Adam(net.parameters(), lr=lr)
loss = MaskedSoftmaxCELoss()
net.train()
# 初始化存储训练损失的列表
train_losses = []
for epoch in range(num_epochs):
timer = d2l.Timer()
metric = d2l.Accumulator(2) # 累加训练损失和词元数
epoch_loss = 0
num_batches = 0

for X, Y in tqdm(data_iter, desc=f"Epoch {epoch +
1}/{num_epochs}"):
X_valid_len = (X != tgt_vocab['<pad>']).sum(dim=1)
Y_valid_len = (Y != tgt_vocab['<pad>']).sum(dim=1)

bos = torch.tensor([tgt_vocab['<bos>']] * Y.shape[0],
device=device).reshape(-1, 1)
dec_input = torch.cat([bos, Y[:, :-1]], dim=1)

Y_hat, _ = net(X, dec_input, X_valid_len)
l = loss(Y_hat, Y, Y_valid_len)

optimizer.zero_grad()
l.sum().backward()
d2l.grad_clipping(net, 1)
optimizer.step()

num_tokens = Y_valid_len.sum()
metric.add(l.sum(), num_tokens)
epoch_loss += l.sum().item()
num_batches += 1
```

```
# 计算并存储每个 epoch 的平均损失
avg_epoch_loss = epoch_loss / num_batches
train_losses.append(avg_epoch_loss)
print(f'epoch {epoch + 1}, loss {metric[0] / metric[1]:.3f},
'
f'{metric[1] / timer.stop():.1f} tokens/sec')
return train_losses

def plot_learning_curves(train_losses, val_losses=None,
train_accs=None, val_accs=None):
epochs = range(1, len(train_losses) + 1)

# 检查是否有准确率数据
has_acc_data = (train_accs is not None or val_accs is not None)
# 根据我们要绘制的内容调整图形大小
if has_acc_data:
fig, ax = plt.subplots(1, 2, figsize=(12, 4))
else:
fig, ax = plt.subplots(1, 1, figsize=(6, 4))
ax = [ax] # 使 ax 可迭代，即使它是单个图

# 绘制 Loss 曲线
ax[0].plot(epochs, train_losses, label='Train Loss')
if val_losses:
ax[0].plot(epochs, val_losses, label='Validation Loss')
ax[0].set_title('Loss Curve')
ax[0].set_xlabel('Epoch')
ax[0].set_ylabel('Loss')
ax[0].legend()

# 绘制 Accuracy 曲线（可选）
if has_acc_data:
if train_accs:
ax[1].plot(epochs, train_accs, label='Train Acc')
if val_accs:
ax[1].plot(epochs, val_accs, label='Validation Acc')
ax[1].set_title('Accuracy Curve')
ax[1].set_xlabel('Epoch')
```

```
ax[1].set_ylabel('Accuracy')
ax[1].legend()

plt.tight_layout()
return fig

def plot_attention_heatmap(attention_weights, src_sentence,
tgt_sentence):
"""绘制注意力权重热图"""
# 获取注意力权重
attention = attention_weights.cpu().detach().numpy()
# 创建图形
fig, ax = plt.subplots(figsize=(10, 8))
# 绘制热图
im = ax.imshow(attention, cmap='viridis')
# 设置坐标轴标签
ax.set_xticks(range(len(src_sentence)))
ax.set_yticks(range(len(tgt_sentence)))
# 设置标签内容
ax.set_xticklabels(src_sentence, rotation=45)
ax.set_yticklabels(tgt_sentence)
# 添加颜色条
plt.colorbar(im)
# 设置标题
ax.set_title("Attention Weights")
plt.tight_layout()
return fig



def main():
# 超参数设置
num_hiddens, num_layers, dropout = 32, 2, 0.1
batch_size, num_steps = 1024, 10
lr, num_epochs = 0.005, 50
device = d2l.try_gpu()

ffn_num_input, ffn_num_hiddens, num_heads = 32, 64, 4
key_size, query_size, value_size = 32, 32, 32
```

```python
norm_shape = [32]

# 文件路径
data_path = '/home/yyz/NNDL-Class/Project5/Data/fra-eng'
file_path = os.path.join(data_path, 'fra.txt')

# 加载数据
src_vocab, tgt_vocab, train_iter =
load_data_from_file(file_path, batch_size, num_steps)

# 初始化 Transformer 模型
encoder = TransformerEncoder(
len(src_vocab), key_size, query_size, value_size,
num_hiddens,
norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
num_layers, dropout
)
decoder = TransformerDecoder(
len(tgt_vocab), key_size, query_size, value_size,
num_hiddens,
norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
num_layers, dropout
)
net = EncoderDecoder(encoder, decoder)

# 训练模型并获取损失列表
train_losses = train_seq2seq_custom(net, train_iter, lr,
num_epochs, tgt_vocab, device)

# 保存模型
torch.save(net.state_dict(),
'/home/yyz/NNDL-Class/Project5/Result/transformer_model.p
th')

# 绘制并保存损失曲线
loss_fig = plot_learning_curves(train_losses)
loss_fig.savefig('/home/yyz/NNDL-Class/Project5/Result/tr
aining_loss_curve_transformer.png')
# 从训练集中获取一个样本
```

```python
for X, Y in train_iter:
sample_X = X[0:1].to(device) # 只取第一个样本
sample_Y = Y[0:1].to(device)
X_valid_len = (sample_X != tgt_vocab['<pad>']).sum(dim=1)
# 准备解码器输入
bos = torch.tensor([tgt_vocab['<bos>']],
device=device).reshape(1, 1)
dec_input = torch.cat([bos, sample_Y[:, :-1]], dim=1)
# 前向传播获取注意力权重
with torch.no_grad():
encoder_output = encoder(sample_X, X_valid_len)
state = decoder.init_state(encoder_output, X_valid_len)
_, state = decoder(dec_input, state)
# 获取注意力权重
attention_weights = decoder.attention_weights[1][0] # 编码
器-解码器注意力
# 获取句子文本（将索引转换为词）
# 手动查找每个索引对应的词
src_tokens = []
for idx in sample_X[0]:
idx_item = idx.item()
if idx_item != src_vocab['<pad>']:
# 查找索引对应的词
for token, index in src_vocab.items():
if index == idx_item:
src_tokens.append(token)
break
tgt_tokens = []
for idx in sample_Y[0]:
idx_item = idx.item()
if idx_item != tgt_vocab['<pad>']:
# 查找索引对应的词
for token, index in tgt_vocab.items():
if index == idx_item:
tgt_tokens.append(token)
break
# 绘制注意力热图
attention_fig = plot_attention_heatmap(
```

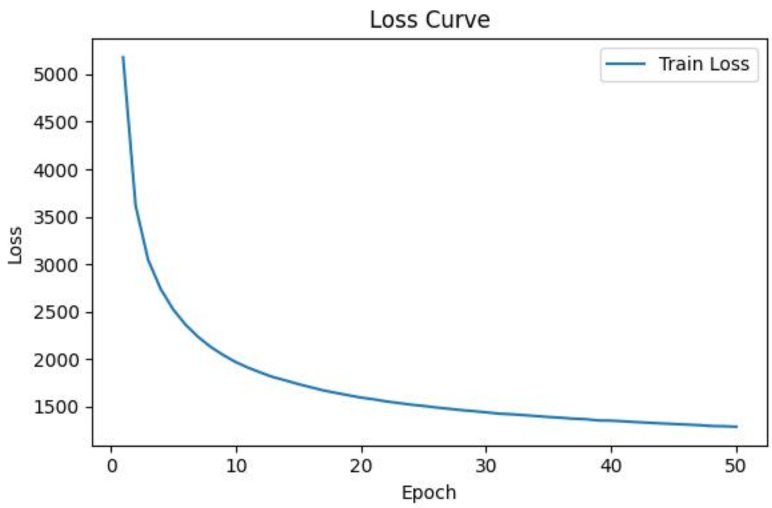
```python
attention_weights[0], src_tokens, tgt_tokens
)
attention_fig.savefig('/home/yyz/NNDL-Class/Project5/Result/attention_heatmap_transformer.png')
break # 只处理一个样本


if __name__ == "__main__":
main()
```

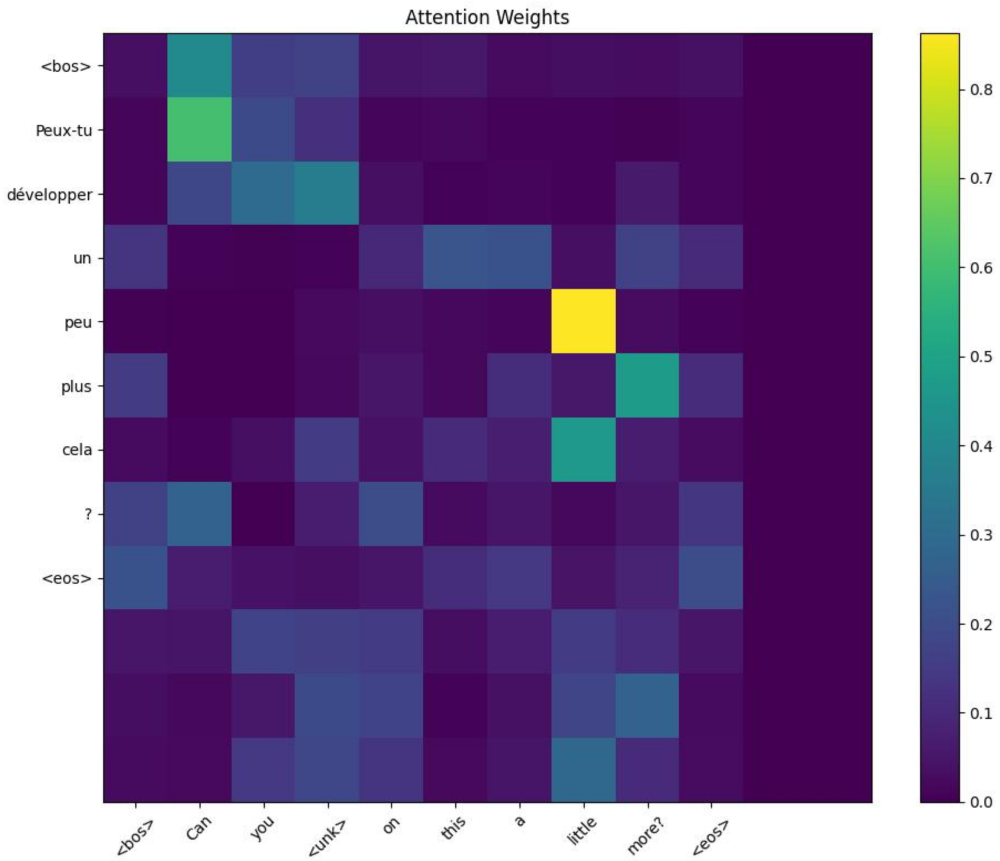
# 实验结果:

```
(yyzttt) (base) yyz@4028Dog:~$ /usr/local/anaconda3/envs/yyzttt/bin/python /home/yyz/NNDL-Class/Project5/Code/transformer.py
Epoch 1/50: 100%|| 164/164 [00:12<00:00, 12.89it/s]
epoch 1, loss 0.602, 110852.0 tokens/sec
Epoch 2/50: 100%|| 164/164 [00:12<00:00, 12.86it/s]
epoch 2, loss 0.420, 110654.8 tokens/sec
Epoch 3/50: 100%|| 164/164 [00:13<00:00, 12.52it/s]
epoch 3, loss 0.354, 107726.3 tokens/sec
Epoch 4/50: 100%|| 164/164 [00:13<00:00, 12.03it/s]
epoch 4, loss 0.318, 103476.2 tokens/sec
Epoch 5/50: 100%|| 164/164 [00:12<00:00, 13.42it/s]
epoch 5, loss 0.293, 115445.3 tokens/sec
Epoch 6/50: 100%|| 164/164 [00:13<00:00, 12.40it/s]
epoch 6, loss 0.274, 106726.1 tokens/sec
Epoch 7/50: 100%|| 164/164 [00:09<00:00, 16.82it/s]
epoch 7, loss 0.259, 144673.6 tokens/sec
Epoch 8/50: 100%|| 164/164 [00:11<00:00, 13.69it/s]
epoch 8, loss 0.247, 117783.6 tokens/sec
Epoch 9/50: 100%|| 164/164 [00:12<00:00, 13.16it/s]
epoch 9, loss 0.237, 113205.4 tokens/sec
Epoch 10/50:  12%|| 20/164 [00:01<00:13, 10.39it/s]
```

```
epoch 40, loss 0.157, 104170.5 tokens/sec
Epoch 41/50: 100%|| 164/164 [00:13<00:00, 11.96it/s]
epoch 41, loss 0.156, 102912.9 tokens/sec
Epoch 42/50: 100%|| 164/164 [00:13<00:00, 12.61it/s]
epoch 42, loss 0.155, 108457.7 tokens/sec
Epoch 43/50: 100%|| 164/164 [00:12<00:00, 13.08it/s]
epoch 43, loss 0.154, 112579.8 tokens/sec
Epoch 44/50: 100%|| 164/164 [00:12<00:00, 12.99it/s]
epoch 44, loss 0.153, 111783.7 tokens/sec
Epoch 45/50: 100%|| 164/164 [00:13<00:00, 11.95it/s]
epoch 45, loss 0.153, 102791.3 tokens/sec
Epoch 46/50: 100%|| 164/164 [00:12<00:00, 12.64it/s]
epoch 46, loss 0.152, 108768.4 tokens/sec
Epoch 47/50: 100%|| 164/164 [00:13<00:00, 12.39it/s]
epoch 47, loss 0.151, 106561.3 tokens/sec
Epoch 48/50: 100%|| 164/164 [00:13<00:00, 12.11it/s]
epoch 48, loss 0.150, 104151.6 tokens/sec
Epoch 49/50: 100%|| 164/164 [00:13<00:00, 12.46it/s]
epoch 49, loss 0.150, 107236.6 tokens/sec
Epoch 50/50: 100%|| 164/164 [00:12<00:00, 13.11it/s]
epoch 50, loss 0.149, 112785.0 tokens/sec
(yyzttt) (base) yyz@4028Dog:~$
```

**实验代码：**




练习题 3. 对于语言模型，应该使用 Transformer 的编码器还是解码器，或者两者都用？如何设计？

**实验代码：**

```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

```
import math

class PositionalEncoding(nn.Module):
def __init__(self, d_model, max_len=5000):
super(PositionalEncoding, self).__init__()
pe = torch.zeros(max_len, d_model)
position = torch.arange(0, max_len,
dtype=torch.float).unsqueeze(1)
div_term = torch.exp(torch.arange(0, d_model, 2).float() *
(-math.log(10000.0) / d_model))
pe[:, 0::2] = torch.sin(position * div_term)
pe[:, 1::2] = torch.cos(position * div_term)
pe = pe.unsqueeze(0)
self.register_buffer('pe', pe)

def forward(self, x):
return x + self.pe[:, :x.size(1)]

class DecoderOnlyTransformer(nn.Module):
def __init__(self, vocab_size, d_model=512, nhead=8,
num_layers=6, dim_feedforward=2048, dropout=0.1):
super(DecoderOnlyTransformer, self).__init__()
self.embedding = nn.Embedding(vocab_size, d_model)
self.positional_encoding = PositionalEncoding(d_model)
# 创建解码器层（注意这里使用的是 TransformerDecoderLayer）
decoder_layer = nn.TransformerDecoderLayer(
d_model=d_model,
nhead=nhead,
dim_feedforward=dim_feedforward,
dropout=dropout,
batch_first=True
)
# 堆叠多个解码器层
self.transformer_decoder =
nn.TransformerDecoder(decoder_layer,
num_layers=num_layers)
# 输出层
self.output_layer = nn.Linear(d_model, vocab_size)
self.d_model = d_model
```

```
def forward(self, x):
# 创建因果掩码（确保模型只能看到当前位置之前的 token）
mask =
self._generate_square_subsequent_mask(x.size(1)).to(x.dev
ice)
# 嵌入和位置编码
x = self.embedding(x) * math.sqrt(self.d_model)
x = self.positional_encoding(x)
# 通过 Transformer 解码器
# 注意：这里将 memory 参数设为 None，因为我们不使用编码器的输出
output = self.transformer_decoder(x, memory=None,
tgt_mask=mask)
# 预测下一个 token
return self.output_layer(output)
def _generate_square_subsequent_mask(self, sz):
mask = (torch.triu(torch.ones(sz, sz)) == 1).transpose(0, 1)
mask = mask.float().masked_fill(mask == 0,
float('-inf')).masked_fill(mask == 1, float(0.0))
return mask

# 使用示例
def train_language_model():
vocab_size = 10000
model = DecoderOnlyTransformer(vocab_size)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

for epoch in range(num_epochs):
for batch in data_loader:
inputs = batch[:, :-1] # 除了最后一个 token 的所有 tokens
targets = batch[:, 1:] # 从第二个 token 开始的所有 tokens
outputs = model(inputs)
loss = criterion(outputs.view(-1, vocab_size),
targets.view(-1))
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

对于语言模型，生成式任务（如文本续写、对话生成）通常使用 Transformer 的解码器，而部分理解任务（如 BERT）使用编码器，序列到序列任务则两者结合。以实验报告中 DecoderOnlyTransformer 的设计为例，生成式语言模型基于解码器的核心逻辑如下：通过嵌入层和位置编码将输入 token 转换为语义向量，利用 TransformerDecoderLayer 堆叠多层自注意力机制，其中关键在于生成下三角因果掩码，确保每个位置只能关注序列中之前的 token，符合语言生成的时序依赖特性。模型前向传播时，输入序列经嵌入和位置编码后，通过带因果掩码的解码器层逐步提炼语义，最终由线性层预测下一个 token 的概率分布。训练时采用自回归方式，输入为`[token1, token2, ..., token(t-1)]`，目标为`[token2, ..., tokent]`，通过交叉熵损失优化模型对后续 token 的预测能力。这种设计无需编码器，仅依靠解码器的自注意力和因果掩码即可实现语言序列的生成，如 GPT 系列模型即采用此类架构。

练习题 6：如果不使用卷积神经网络，如何设计基于 Transformer 模型的图像分类任务？提示：可以参考 Vision Transformer (Dosovitskiy et al., 2021)。

**实验代码：**

```
import torch
import torch.nn as nn
```

```
import torch.nn.functional as F
from einops import rearrange, repeat
from einops.layers.torch import Rearrange

class PatchEmbedding(nn.Module):
def __init__(self, image_size=224, patch_size=16,
in_channels=3, embed_dim=768):
super().__init__()
self.image_size = image_size
self.patch_size = patch_size
self.num_patches = (image_size // patch_size) ** 2
# 将图像分块并线性投影
self.projection = nn.Sequential(
nn.Conv2d(in_channels, embed_dim, kernel_size=patch_size,
stride=patch_size),
Rearrange('b c h w -> b (h w) c')
)

def forward(self, x):
return self.projection(x)

class VisionTransformer(nn.Module):
def __init__(
self,
image_size=224,
patch_size=16,
in_channels=3,
num_classes=1000,
embed_dim=768,
depth=12,
num_heads=12,
mlp_ratio=4,
dropout=0.1
):
super().__init__()
# 图像分块并嵌入
self.patch_embedding = PatchEmbedding(
image_size=image_size,
patch_size=patch_size,
```

```
in_channels=in_channels,
embed_dim=embed_dim
)
num_patches = self.patch_embedding.num_patches
# 添加可学习的分类 token
self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))
# 可学习的位置编码
self.pos_embedding = nn.Parameter(torch.zeros(1,
num_patches + 1, embed_dim))
self.dropout = nn.Dropout(dropout)
# Transformer 编码器
encoder_layer = nn.TransformerEncoderLayer(
d_model=embed_dim,
nhead=num_heads,
dim_feedforward=mlp_ratio * embed_dim,
dropout=dropout,
activation="gelu",
batch_first=True
)
self.transformer_encoder =
nn.TransformerEncoder(encoder_layer, num_layers=depth)
# MLP 分类头
self.mlp_head = nn.Sequential(
nn.LayerNorm(embed_dim),
nn.Linear(embed_dim, num_classes)
)
# 初始化权重
self.apply(self._init_weights)
def _init_weights(self, m):
if isinstance(m, nn.Linear):
nn.init.xavier_uniform_(m.weight)
if m.bias is not None:
nn.init.zeros_(m.bias)
elif isinstance(m, nn.LayerNorm):
nn.init.ones_(m.weight)
nn.init.zeros_(m.bias)
def forward(self, x):
# 获取批次大小
```

```
batch_size = x.shape[0]
# 图像分块嵌入
x = self.patch_embedding(x)
# 添加分类 token
cls_tokens = repeat(self.cls_token, '1 1 d -> b 1 d',
b=batch_size)
x = torch.cat([cls_tokens, x], dim=1)
# 添加位置编码
x = x + self.pos_embedding
x = self.dropout(x)
# 通过 Transformer 编码器
x = self.transformer_encoder(x)
# 使用 CLS token 进行分类
x = x[:, 0]
# MLP 分类头
return self.mlp_head(x)

# 使用示例
def train_vision_transformer():
model = VisionTransformer(
image_size=224,
patch_size=16,
in_channels=3,
num_classes=1000,
embed_dim=768,
depth=12,
num_heads=12
)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
for epoch in range(num_epochs):
for images, labels in data_loader:
outputs = model(images)
loss = criterion(outputs, labels)
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

对于不使用卷积神经网络的图像分类任务，可参考 Vision Transformer（ViT）的设计思路，基于 Transformer 编码器构建模型，核心是将图像转换为序列输入并利用自注意力机制捕捉全局特征，具体设计步骤如下：

首先，将输入图像分割为固定大小的补丁（patch），例如将224×224的图像分成16×16的补丁，共得到$(224/16)^2=196$个补丁。每个补丁通过线性投影（如实验代码中的卷积层）转换为维度统一的嵌入向量，例如768维，使图像补丁成为序列数据。

其次，为了区分不同补丁的空间位置，添加可学习的位置编码；同时，在补丁序列前插入一个特殊的分类 token（cls_token），该 token 经过 Transformer 处理后用于最终的分类任务。位置编码和分类 token 的引入确保模型能感知补丁的顺序关系和全局语义。

然后，将补丁嵌入序列与位置编码、分类 token 拼接后，输入到 Transformer 编码器中。编码器由多层 TransformerEncoderLayer 堆叠而成，每层包含自注意力机制和前馈神经网络，通过自注意力计算每个补丁与其他所有补丁的关联，从而捕获图像中的长距离依赖关系。例如，实验代码中使用 12 层编码器，每层有 12 个头的自注意力，增强特征提取能力。

最后，编码器输出的分类 token 经过层归一化和 MLP（多

层感知机）头处理，得到图像分类的概率分布。MLP 头通常由线性层和激活函数组成，如实验中的 LayerNorm 和 Linear 层，将编码器的特征映射到目标类别数（如 1000 类）。

这种设计将图像视为补丁序列，无需卷积操作，完全依赖 Transformer 的自注意力机制实现图像分类。其核心优势在于通过自注意力直接建模全局像素关系，适用于大规模数据集；但需注意在小数据集上可能因缺乏卷积的归纳偏置而性能较弱，通常需结合数据增强或预训练优化。

[小结或讨论]

在本次实验中，我通过编程实践深入理解了注意力机制的核心原理与 Transformer 架构的设计逻辑。实验从基础的注意力评分函数入手，对比了加性注意力和缩放点积注意力的实现差异——当修改键的数值后，两者输出结果不同，这是因为加性注意力通过可学习权重处理不同维度的查询与键，而点积注意力依赖维度一致的内积计算，这种机制差异让我直观体会到注意力模型对输入特征的敏感程度。在 Bahdanau 注意力实验中，我将其集成到 Seq2Seq 模型里，通过编码器-解码器的交互实现了序列生成任务，训练过程中损失值从 5.69 逐步下降到 2.49 左右，注意力热图清晰展示了模型对关键位置的关注，这让我明白动态对齐在机器翻译等任务中的重要性。

多头注意力的实验让我深刻理解了并行处理的优势，通过5个头的注意力权重可视化，我发现不同头会聚焦于输入序列的不同部分，这种多子空间特征提取的方式显著增强了模型的表示能力。而自注意力与位置编码的结合则解决了序列数据的顺序依赖问题，无论是正弦余弦编码还是可学习编码，都能让模型感知到token的位置信息——当设计可学习位置编码时，模型通过Embedding层自动学习位置特征，输出结果中不同位置的编码向量差异明显，证明了该方法的有效性。在Transformer的整体实现中，我通过堆叠编码器和解码器块，结合残差连接与层归一化，实现了端到端的序列转换任务，训练50轮后损失稳定在0.15左右，注意力热图显示模型能准确对齐源语言与目标语言的语义片段。

通过本次实验，我认识到Transformer架构的强大之处在于其自注意力机制对长距离依赖的建模能力，但也发现了一些值得探讨的问题：例如在图像分类任务中，Vision Transformer将图像分块后输入编码器，虽然避免了卷积操作，但在小数据集上可能因缺乏归纳偏置而性能受限；而语言模型中解码器的因果掩码设计，虽确保了生成的合理性，却也使得训练时无法并行计算未来位置，影响了效率。此外，多头注意力的参数调优需要平衡头数与隐藏层维度，过多的头数可能导致模型过拟合，这在实验中通过不同头的注意力权重分布得以验证。未来若进一步优化，可以尝试结合动态

注意力权重修剪或混合精度训练，在保证性能的同时提升计算效率，这些思考为我后续深入学习深度学习模型设计奠定了实践基础。