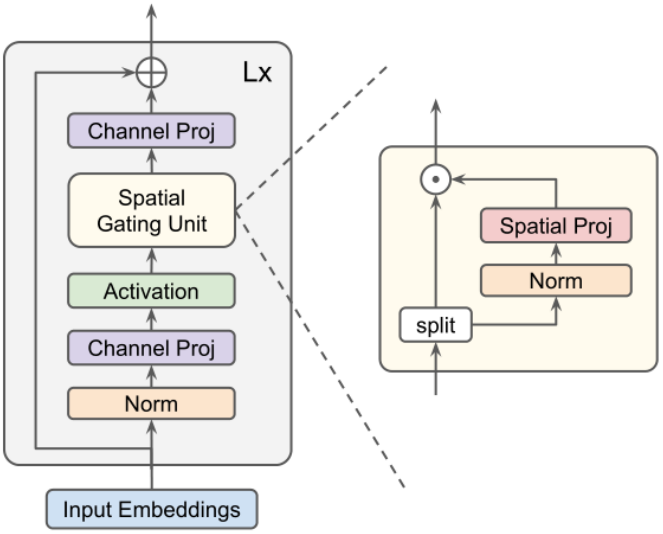
论文主要内容

自从Transformers和BERT被提出，计算机视觉和自然语言处理上的很多任务的最好水平都被刷新了。但为什么Transformers能达到这么好的效果？是注意力机制太有效了？还是它的前馈神经网络？本文的作者想要验证self-attention是不是必要的，提出了一个带有门机制的MLP架构。

模型结构如下



Pseudo-code for the gMLP block

```
def gmlp_block(x, d_model, d_ffn):
  shortcut = x
  x = norm(x, axis="channel")
  x = proj(x, d_ffn, axis="channel")
  x = gelu(x)
  x = spatial_gating_unit(x)
  x = proj(x, d_model, axis="channel")
  return x + shortcut

def spatial_gating_unit(x):
  u, v = split(x, axis="channel")
  v = norm(v, axis="channel")
  n = get_dim(v, axis="spatial")
  v = proj(v, n, axis="spatial", init_bias=1)
  return u * v
```

因为就是一个多层感知机，结构还是比较简单的。上图是一个block的结构，完整的gMLP由六个这样的block组成。拿NLP的输入输出举例。输入的X是一个n*d的张量，n是X的序列长度，d是词向量的维度。

对论文的理解和感想

理解：

1. 在NLP任务上难以超过Transformers，需要将参数量提升非常大才能缩小差距，但加入一个小小的注意力机制模块，效果就能变的很好。能用加大参数量提升效果，就说明注意力机制不必要。但加入注意力机制效果有很大的提升，说明它很重要。
2. 在图像任务上轻松地就比Transformers表现更好，但在自然语言处理上不行。这让我想到了CNN的提出。一张图片上的像素点很多，如果用全连接的方式来处理，虽然能考虑每一点信息，但这个参数量是我们的硬件无法承受的。CNN基于一些理论（如平移不变性）来对图像进行类似特征提取的操作，有效地降低了参数量且保留了重要信息。到现在，好些人提出这样的MLP架构，与现在硬件条件的发展是分不开的。我们已经有能力考虑全部的信息了，不需要针对不同的数据、不同的任务设定什么特殊的模型结构，直接全连接，让模型去学习重要的信息，就能达到很好的效果。在图像的任务上这样操作是没什么问题的。

但是在NLP中，这个模型在单句任务（如情感分类）上表现很好，在需要跨句子的任务上却不行。说明这种简单的结构还是没办法理解语言之间的语义联系，仍然需要注意力机制发挥作用。
3. 文中很多实验都提到，性能的差距可以通过提升参数量来弥补。作者想表达的意思是，有些模型就算提高复杂度性能也无法再提高了。这个模型的表现能一直通过增加深度来提升，说明它的潜力很大。

感想：

1. 我觉得深度学习/机器学习的发展与硬件水平的发展太相关了。二者像是在同一个方向上奔跑，有时候深度学习发展地太快，硬件跟不上了，促使人们努力研发硬件，就像是深度学习拉了硬件一把。有时候硬件跑地太快，深度学习就能受益于此，自由地快速发展。这也是今年来人工智能兴起的原因吧。深度学习一直在进行性能和计算资源上的权衡。MLP结构的又一次出现是人们在试探现有的计算能力到底能达到多好的效果。
2. 虽然在计算机视觉上gMLP的参数量少，性能还好。但在NLP上如果不用注意力机制需要三倍的参数才能弥补gMLP和transformers之间的差距。我还是更喜欢根据人类大脑真实的运转方式，或是自然界中一些广泛存在的规律，迁移到模型的设计上。根据不同的任务/不同的数据，设计不同结构的模型，即节约计算资源（相比直接一股脑全连接），又能取得好的效果。（比如CNN的发展过程中就对视觉的生理性原理做了很多实验，注意力机制也非常符合人类认知世界的方式）或许现在Transformers和MLP是想找到一个适用于所有情景的通用模型？我觉得有点怀疑...
3. 除此之外，作者也在文章提到静态参数化的MLP可以拟合任意函数。gMLP这种MLP结构现在取得了这么好的结果，好像又在强调深度学习是个黑盒。让模型自己去学习参数，自己提取重要信息，也不知道哪个像素点在哪一步就重要了。还是attention、CNN的感受野等原理，这些结构符合人的认知方式，更容易让人接受。
4. 听说"MLP结构简单，在工业界有很多成熟的优化的方式"，或许这些研究可以在业界发挥作用，从这个角度看还是挺好的。但我的认知有限，只能说是听说。

自己的理解不足：

1. 本科接触的东西还是太少了，导致看见一些东西无法产生相关的联想。在网上看见有人说SpatialGatingUnit就类似于注意力机制或是卷积操作，但我看这个门的公式没有太大感觉，也不是很理解channel projection和spatial projection的区别。
2. 这篇文章之前也有一些探究Transformers结构的文章了，比如同样是MLP架构的MLP-Mixer，用傅里叶变换代替attention的FNet...如果最近没有那些期末大作业的话，想一起看看，对比一下。

我掌握的知识还是很浅显，但还是把自己想的东西都写出来了。如果有说的不对的地方，希望老师可以不吝赐教。谢谢老师！

代码实现

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
"""
B: batch_size
L: seq_len
D: d_model
H: d_ffn
V: vocab_size
"""


class GELU(nn.Module):
    def __init__(self):
        super(GELU, self).__init__()

    def forward(self, x):
        return 0.5*x*(1+F.tanh(np.sqrt(2/np.pi)*(x+0.044715*torch.pow(x,3))))
# https://blog.csdn.net/w137093940/article/details/112756141


class SpatialGatingUnit(nn.Module):
    def __init__(self, seq_len, d_ffn):
        super().__init__()
        # 输入的x的维度是B * L *H ,那么LayerNorm的参数是H(与x的最后一个维度相同)
        self.norm = nn.LayerNorm(d_ffn//2)
        self.proj = nn.Conv1d(seq_len, seq_len, kernel_size=1)
        # 输入通道是seq_len，输出通道是seq_len（卷积核个数），卷积核大小为1
        nn.init.constant_(self.proj.weight, 0)
        nn.init.constant_(self.proj.bias, 1)

    def forward(self,x):
        # x B * L * H
        u, v = x.chunk(2, dim=-1)  # 在最后一个维度上切分
        # u,v B * L * H/2
        v = self.norm(v)
        v = self.proj(v) # v B * L * H/2
        return u * v  # B * L * H/2
```

```
class gMLPblock(nn.Module):
    def __init__(self, seq_len, d_model, d_ffn):
        # seq_len是序列的长度
        # d_model是词向量的维度
        # d_ffn是前馈神经网络中隐藏层的维度
        super().__init__()
        self.norm = nn.LayerNorm(d_model)
        self.channel_proj1 = nn.Linear(d_model, d_ffn)
        self.sgu = SpatialGatingUnit(seq_len, d_ffn)
        self.channel_proj2 = nn.Linear(d_ffn//2, d_model)
        self.act = GELU()


    def forward(self,x):
        # 输入的x的维度是 B * L * D
        shortcut = x
        x = self.norm(x)
        x = self.channel_proj1(x)  # U d_model * seq_len x变为batch_size * seq_len * d_ffn
        x = self.act(x)  # x batch_size * seq_len * d_ffn
        x = self.sgu(x)  # x batch_size * seq_len * d_ffn/2
        x = self.channel_proj2(x)  # V batch_size * seq_len * d_model
        return shortcut + x # batch_size * seq_len * d_model


class gMLP(nn.Module):
    def __init__(self, seq_len=256, d_model=256, d_ffn=512, num_layers=6):
        super().__init__()
        self.model = nn.Sequential(*[gMLPblock(seq_len, d_model, d_ffn)]*num_layers)

    def forward(self,x):
        x = self.model(x)
        return x


class gMLPofLanguageModel(gMLP):
    def __init__(self, vocab_size=20000, seq_len=256, d_model=256, d_ffn=512, num_layers=6):
        super().__init__(seq_len, d_model, d_ffn, num_layers)
        self.embedding = nn.Embedding(vocab_size, d_model)
        self.fc = nn.Linear(d_model, vocab_size)

    def forward(self, x):
        emb = self.embedding(x)  # B * L * D
        out = self.model(emb) # B * L * D
        out = self.fc(out) # B * L * V
        return out


vocab_size = 10000
batch_size = 16
seq_len = 20
num_layers = 6
input = torch.randint(vocab_size, (batch_size, seq_len))
gmlp = gMLPofLanguageModel(vocab_size=vocab_size,seq_len=seq_len,d_model=512,d_ffn=1024)
output=gmlp(input)
print(output.shape)
```

torch.Size([16, 20, 10000])

如果是aMLP

```
class TinyAttn(nn.Module):
    def __init__(self, d_out,d_model,d_attn=64):
        super().__init__()
        self.proj1 = nn.Linear(d_model, 3 * d_attn)
        self.proj2 = nn.Linear(d_attn, d_out)
        self.d_attn = d_attn

    def forward(self, x):
        qkv = self.proj1(x) # B * L * 3attn
        q, k, v = qkv.chunk(3, dim=-1) # B * L * attn
        k = k.permute(0,2,1) # B * attn *L
        w = torch.matmul(q,k) # B * L * L
        a = F.softmax(w) # B * L * L
        x = torch.matmul(a,v) # B * L * attn
        x = self.proj2(x) # B * L * d_out
        return x
```

在门控单元中加入

```
self.attn = TinyAttn(d_out=d_ffn // 2, d_model=d_ffn)
```

对初始x进行attention操作后，与v相加，结果再与u相乘