Deep Learning Lab3 Report

109511207 蔡宗儒

1. Task1-Transformer

Number of layers = 3

```
class Classifier(nn.Module):
    def __init__(self, d_model = 176, dim_feedforward = 1024, n_spks = 600, dropout = 0.1):
        super().__init__()
        # Project the dimension of features from that of input into d_model.
        self.prenet = nn.Linear(40, d_model)
        # TODO:
        #   Build vanilla transformer encoder block from scratch !!!
        #   You can't call the transformer encoder block function from PyTorch library !!!
        #   The number of encoder layers should be less than 3 !!!
        #   The parameter size of transformer encoder block should be less than 500k !!!
        self.encoder_layer = TransformerEncoderLayer(d_model, dim_feedforward, 16, dropout)
        self.encoder = TransformerEncoder(self.encoder_layer, 3)
        # Project the the dimension of features from d_model into speaker nums.
        self.pred_layer = nn.Sequential(
            nn.Linear(d_model, d_model),
            nn.ReLU(),
            nn.Linear(d_model, n_spks),
        )
```

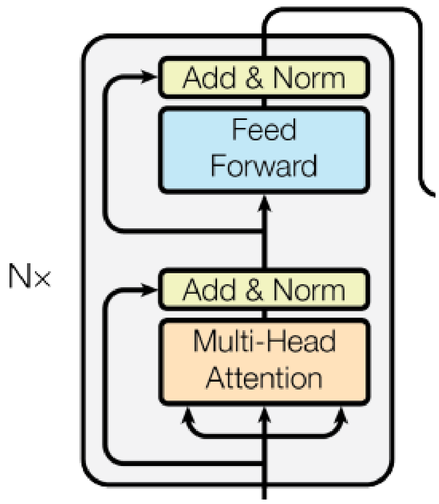
Parameter size = 486.96k

```
The parameter size of encoder block is 486.96k
```

Accuracy = 0.8047

```
Train:   1% 16/2000 [00:00<00:27, 72.66 step/s, accuracy=0.88, loss=0.59, step=6e+4]
Step 60000, best model saved. (accuracy=0.7927)
Train: 100% 2000/2000 [00:32<00:00, 61.13 step/s, accuracy=0.97, loss=0.17, step=62000]
Valid: 100% 5664/5667 [00:03<00:00, 1805.40 uttr/s, accuracy=0.80, loss=0.94]
Train: 100% 2000/2000 [00:32<00:00, 60.71 step/s, accuracy=0.94, loss=0.17, step=64000]
Valid: 100% 5664/5667 [00:03<00:00, 1838.26 uttr/s, accuracy=0.79, loss=0.98]
Train: 100% 2000/2000 [00:32<00:00, 61.66 step/s, accuracy=0.94, loss=0.19, step=66000]
Valid: 100% 5664/5667 [00:03<00:00, 1750.72 uttr/s, accuracy=0.80, loss=0.98]
Train: 100% 2000/2000 [00:32<00:00, 60.88 step/s, accuracy=0.91, loss=0.22, step=68000]
Valid: 100% 5664/5667 [00:03<00:00, 1773.39 uttr/s, accuracy=0.80, loss=0.95]
Train: 100% 2000/2000 [00:32<00:00, 61.64 step/s, accuracy=0.97, loss=0.16, step=7e+4]
Valid: 100% 5664/5667 [00:03<00:00, 1828.94 uttr/s, accuracy=0.80, loss=0.95]
Train:   0% 0/2000 [00:00<?, ? step/s]
Step 70000, best model saved. (accuracy=0.8047)
```

Task1 需實作 transformer encoder，對音檔資料做多層 encode，然後將其傳遞給後面的助教寫好的 fully connected network 做辨識，encoder layer 架構如下圖。



Screenshot of transformer code

程式碼如下，encoder 將 encoder layer 重複 3 層，其中我將 TransformerEncoderLayer 進一步拆成 MultiHeadAttention 和 FeedForward，並依照論文實作 residual connection，避免梯度消失。

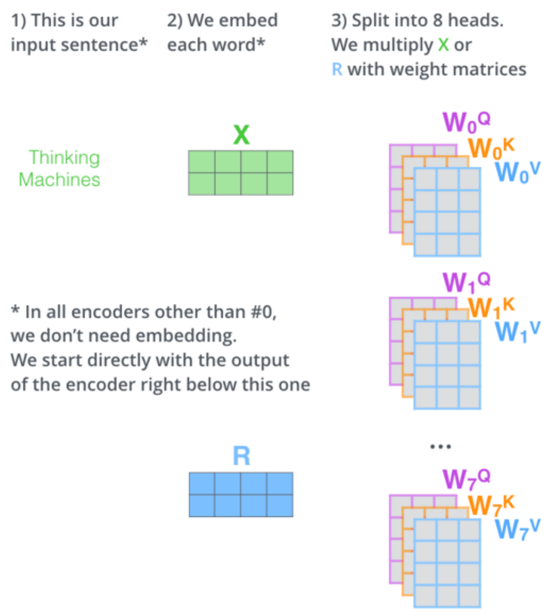
```
# Encoder Layer
class TransformerEncoderLayer(nn.Module):
  def __init__(self, d_model, dim_feedforward, nhead, dropout):
    super().__init__()
    # Calculate dk
    self.dk = d_model // nhead
    # Multi Head Attention
    self.attn = MultiHeadAttention(d_model, nhead, self.dk, dropout)
    # Feed Forward
    self.ff = FeedForward(d_model, dim_feedforward, dropout)
    # Layernorm
    self.ln = nn.LayerNorm(d_model)

  def forward(self, x):
    # Residual Connection & Multi Head Attention
    x = x + self.attn(x)
    # Layernorm
    x = self.ln(x)
    # Residual Connection & Feed Forward
    x = x + self.ff(x)
    # Layernorm
    x = self.ln(x)
    return x

# Encoder
class TransformerEncoder(nn.Module):
  def __init__(self, encoder_layer, num_layers):
    super().__init__()
    # Encoder Layer Blocks * N
    self.encoderlayers = nn.ModuleList([
        encoder_layer
        for _ in range(num_layers)])

  def forward(self, x):
    return self.encoderlayers(x)
```

Multi Head Attention 將 input 經 Layernorm(我自己多加的)後再經過 3 個線性轉換後生成 q、k 和 v，再將 q、k 和 v 切成 nhead 個(即論文中的 h)、dimension = dk(即 d_model // nhead)，如下圖，即從 shape 為 length x batch x d_model 切成 length x batch x nhead x dk。這個切分的步驟可以增加整個 encoder 的複雜度，讓模型可以同時關注不同部分的 input 序列、更好地理解 input 的特徵，並且可以讓計算平行進行，加速了 training 的速度。



接著再將其 transpose 成 shape 為 nhead x batch x length x dk，做 nhead 個 Scaled Dot Product Attention 後 concatenate 回 length x batch x d_model，最後再經過一層 Linear 以及一層 Dropout 後輸出，程式碼如下。

```
# Multi Head Attention
class MultiHeadAttention(nn.Module):
  def __init__(self, d_model, nhead, dk, dropout):
    super().__init__()
    # Layernorm
    self.ln = nn.LayerNorm(d_model)
    # Define self variable
    self.nhead = nhead
    self.dk = dk
    # Linear layer of weight query, key, value
    self.q = nn.Linear(d_model, d_model)
    self.k = nn.Linear(d_model, d_model)
    self.v = nn.Linear(d_model, d_model)
    # Linear layer
    self.l = nn.Linear(d_model, d_model)
    # Dropout layer
    self.dropout = nn.Dropout(dropout)

  def forward(self, x):
    # Layernorm
    x = self.ln(x)
    # Get length & batch
    length = x.size(0)
    batch = x.size(1)
    # Weight query, key, value
    # length x batch x d_model to length x batch x nhead x dk to nhead x batch x length x dk
    q = self.q(x).view(length, batch, self.nhead, self.dk).transpose(0, 2)
    k = self.k(x).view(length, batch, self.nhead, self.dk).transpose(0, 2)
    v = self.v(x).view(length, batch, self.nhead, self.dk).transpose(0, 2)
    # Scaled Dot Product Attention
    # Score = query dot key
    out = torch.matmul(q, k.transpose(-2, -1))
    # Divided by sqrt(dk)
    out = out / math.sqrt(self.dk)
    # Softmax
    out = F.softmax(out, dim=-1)
    # Softmax * value
    out = torch.matmul(out, v)
    # nhead x batch x length x dk to length x batch x nhead x dk to length x batch x d_model
    out = out.transpose(0, 2).contiguous().view(length, batch, -1)
    # Linear layer
    out = self.l(out)
    # Dropout Layer
    out = self.dropout(out)
    return out
```

FeedForward 的功能是增加整個模型的非線性度，我依照論文做 Linear -> ReLU -> Linear -> Dropout 程式碼如下。

```
# Feed Forward
class FeedForward(nn.Module):
  def __init__(self, d_model, dim_feedforward, dropout):
    super().__init__()
    # All layers
    self.layer = nn.Sequential(
      nn.Linear(d_model, dim_feedforward),  # Linear Layer
      nn.ReLU(),  # Relu Activation
      nn.Linear(dim_feedforward, d_model),  # Linear Layer
      nn.Dropout(dropout) # Dropout
    )

  def forward(self, x):
    x = self.layer(x)
    return x
```

How to improve the performance

我實作的數據如下，number of layers 皆= 3。

d_model	dim_feedforward	nhead	val acc
80	4 * 80	8	0.6967
128	4 * 128	8	0.7509
160	4 * 160	8	0.7745
128	8 * 128	16	0.7805
64	32 * 64	16	0.7286
128	12 * 128	16	0.7735
144	8 * 144	16	0.7864
192	4 * 192	16	0.7800
160	1024	16	0.7851

可以觀察到普遍來說 d_model 越大，parameter size 越大時，acc 會越高，而我在研究 task2 的 conformer 架構時，發現在進入每個 module 之前，都會先經過 Layernorm，但 task1 的 transformer 在進入 Multi Head Attention 之前，並沒有經過 Layernorm，所以我將 Multi Head Attention 多加入了一層 Layernorm，讓 input 可以先經過這層 Layernorm，再得出 q、k 和 v，並讓 parameter size 盡可能接近 500k。最後我將 d_model 設為 176、dim_feedforward 設為 1024、nhead = 16、parameter size = 486.96k，成功將 val acc 提高至 0.8047。

2. Task2-Conformer

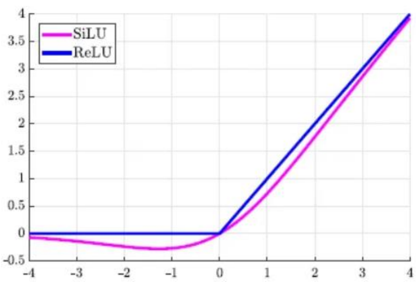
Which kind of transformer-like model do I choose?

A: Conformer

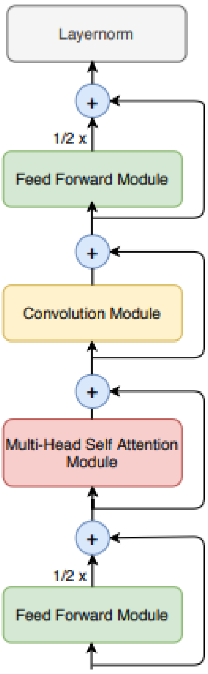
The reason why I choose this model & the advantage of chosen model

Transformer 的 self attention 的設計在針對長距離前後有相關的特徵有較好的效果，但缺乏了提取局部細節特徵的能力；CNN 則相反。而 conformer 是一種結合了 transformer 和 CNN 各自優點的架構，它的架構接近 task1 的 transformer，多了 convolution module，這能彌補 transformer 的缺點，且讓 task2 能在 task1 的 code 的基礎上實作，這也是我這個 task 選擇 conformer 的主因之一。

在 conformer 的 convolution module 中引入了 pointwise convolution 和 depthwise convolution 兩種不同的 convolution 方式。pointwise convolution 可以保留空間維度並改變特徵的維度，這樣能學習 channel 之間的特徵，depthwise convolution 則可以對每個 channel 進行單獨的 convolution，這樣能減少模型的參數數量和計算量並學習各自 channel 內的特徵。另外 conformer 也使用了 GLU 和 SiLU 兩種 activation function，GLU 能將輸入劃分為兩個部分，一邊通過 sigmoid，這樣可以有效地過濾和控制資料的流動。這種機制有助於模型更好地選擇和保留重要的特徵。而 SiLU 在接近 0 時有更平滑的曲線，能夠更快的收斂，如下圖。通常在語音識別中 SiLU 會有比 ReLU 更好的效果。



Conformer encoder layer 架構如下圖。



Number of layers = 3

```
class Classifier(nn.Module):
    def __init__(self, d_model= 160, dim_feedforward = 480, n_spks=600, dropout=0.1):
        super().__init__()
        # Project the dimension of features from that of input into d_model.
        self.prenet = PreNet(40, d_model, dropout)
        #self.prenet = nn.Linear(40,d_model)
        # TODO:
        #   Build vanilla transformer encoder block from scratch !!!
        #   You can't call the transformer encoder block function from PyTorch library !!!
        #   The number of encoder layers should be less than 3 !!!
        #   The parameter size of transformer encoder block should be less than 500k !!!
        self.encoder_layer = ConformerEncoderLayer(d_model, dim_feedforward, 16, 63, dropout)
        self.encoder = ConformerEncoder(self.encoder_layer, 3)
        # Project the the dimension of features from d_model into speaker nums.
        self.pred_layer = nn.Sequential(
            nn.Linear(d_model, d_model),
            nn.ReLU(),
            nn.Linear(d_model, n_spks),
        )
```

Parameter size = 495.84k

```
The parameter size of encoder block is 495.84k
```

Accuracy = 0.8377

```
Train:   0% 10/2000 [00:00<00:44, 44.46 step/s, accuracy=0.94, loss=0.32, step=6e+4]
Step 60000, best model saved. (accuracy=0.8229)
Train: 100% 2000/2000 [00:36<00:00, 54.60 step/s, accuracy=0.88, loss=0.26, step=62000]
Valid: 100% 5664/5667 [00:03<00:00, 1493.02 uttr/s, accuracy=0.83, loss=0.77]
Train: 100% 2000/2000 [00:38<00:00, 52.57 step/s, accuracy=0.91, loss=0.29, step=64000]
Valid: 100% 5664/5667 [00:03<00:00, 1603.19 uttr/s, accuracy=0.83, loss=0.79]
Train: 100% 2000/2000 [00:36<00:00, 54.60 step/s, accuracy=0.94, loss=0.11, step=66000]
Valid: 100% 5664/5667 [00:03<00:00, 1576.61 uttr/s, accuracy=0.82, loss=0.82]
Train: 100% 2000/2000 [00:37<00:00, 53.25 step/s, accuracy=0.91, loss=0.29, step=68000]
Valid: 100% 5664/5667 [00:03<00:00, 1595.61 uttr/s, accuracy=0.84, loss=0.77]
Train: 100% 2000/2000 [00:38<00:00, 51.59 step/s, accuracy=0.88, loss=0.50, step=7e+4]
Valid: 100% 5664/5667 [00:03<00:00, 1457.84 uttr/s, accuracy=0.83, loss=0.80]
Train:   0% 0/2000 [00:00<?, ? step/s]

Train:   0% 0/2000 [00:00<?, ? step/s]
Step 70000, best model saved. (accuracy=0.8377)
```

Screenshot of transformer code

程式碼如下，encoder 將 encoder layer 重複 3 層，其中我將 ConformerEncoderLayer 進一步拆成 MultiHeadAttention、FeedForward 和 ConvolutionModule，並依照論文實作 residual connection，避免梯度消失。

```
class ConformerEncoderLayer(nn.Module):
  def __init__(self, d_model, dim_feedforward, nhead, kernel_size, dropout):
    super().__init__()
    # Calculate dk
    self.dk = d_model // nhead
    # Feed Forward
    self.ff1 = FeedForward(d_model, dim_feedforward, dropout)
    # Multi Head Self Attention
    self.attn = MultiHeadAttention(d_model, nhead, self.dk, dropout)
    # Convolution Module
    self.conv = ConvolutionModule(d_model, 31, dropout)
    # Feed Forward
    self.ff2 = FeedForward(d_model, dim_feedforward, dropout)
    # Layer Norm
    self.ln = nn.LayerNorm(d_model)

  def forward(self, x):
    # Residual Connection & Feed Forward
    x = x + 0.5 * self.ff1(x)
    # Residual Connection & Multi Head Attention
    x = x + self.attn(x)
    # Residual Connection & Convolution
    x = x + self.conv(x)
    # Residual Connection & Feed Forward
    x = x + 0.5 * self.ff2(x)
    # Layer Norm
    x = self.ln(x)
    return x

class ConformerEncoder(nn.Module):
  def __init__(self, encoder_layer, num_layers):
    super().__init__()

    # Encoder Layer Blocks * N
    self.encoderlayers = nn.ModuleList([
        encoder_layer
        for i in range(num_layers)])
  def forward(self, x):
    return self.encoderlayers(x)
```

Multi Head Attention 和 Transformer 的做法相同，程式碼如下。

```
# Multi Head Attention
class MultiHeadAttention(nn.Module):
  def __init__(self, d_model, nhead, dk, dropout):
    super().__init__()
    # Define self variable
    self.nhead = nhead
    self.dk = dk
    # Layernorm
    self.ln = nn.LayerNorm(d_model)
    # Linear Layer of weight query, key, value
    self.q = nn.Linear(d_model, d_model)
    self.k = nn.Linear(d_model, d_model)
    self.v = nn.Linear(d_model, d_model)
    # Linear Layer
    self.l = nn.Linear(d_model, d_model)
    # Dropout
    self.dropout = nn.Dropout(dropout)

  def forward(self, x):
    # Get length & batch
    length = x.size(0)
    batch = x.size(1)
```

```
    # Layernorm
    x = self.ln(x)
    # Weight query, key, value
    # length x batch x d_model to length x batch x nhead x dk to nhead x batch x length x dk
    q = self.q(x).view(length, batch, self.nhead, self.dk).transpose(0, 2)
    k = self.k(x).view(length, batch, self.nhead, self.dk).transpose(0, 2)
    v = self.v(x).view(length, batch, self.nhead, self.dk).transpose(0, 2)
    # Scaled Dot Product Attention
    # Score = query dot key
    out = torch.matmul(q, k.transpose(-2, -1))
    # Divided by sqrt(dk)
    out = out / math.sqrt(self.dk)
    # Softmax
    out = F.softmax(out, dim=-1)
    # Softmax * value
    out = torch.matmul(out, v)
    # nhead x batch x length x dk to length x batch x nhead x dk to length x batch x d_model
    out = out.transpose(0, 2).contiguous().view(length, batch, -1)
    # Linear Layer
    out = self.l(out)
    # Dropout
    out = self.dropout(out)
    return out
```

FeedForward，和 Tranformer 不同的是將 ReLU 改成 SiLU，FeedForward 程式碼如下。

```
# Feed Forward
class FeedForward(nn.Module):
  def __init__(self, d_model, dim_feedforward, dropout):
    super().__init__()
    # All layers
    self.layer = nn.Sequential(
      nn.LayerNorm(d_model),  # Layernorm
      nn.Linear(d_model, dim_feedforward),  # Linear Layer
      nn.SiLU(),  # Swish Activation
      nn.Dropout(dropout),  # Dropout
      nn.Linear(dim_feedforward, d_model),  # Linear Layer
      nn.Dropout(dropout)  # Dropout
    )

  def forward(self, x):
    x = self.layer(x)
    return x
```

ConvolutionModule，先透過 pointwise convolution 改變特徵的維度，將 channel 數變為兩倍，而後透過 GLU 將 channel 數變回 d_model，再經 depthwise convolution 捕捉特徵並減少參數量和計算量，進一步提高效率。程式碼如下。

```
class ConvolutionModule(nn.Module):
  def __init__(self, d_model, kernel_size, dropout):
    super().__init__()
    # Layermorm
    self.ln = nn.LayerNorm(d_model)
    # All Remained Layers
    self.layer = nn.Sequential(
      # batch x d_model x length to
      # batch x (d_model x 2) x length
      # Pointwise Convolution with expansion = 2
      nn.Conv1d(in_channels=d_model, out_channels=d_model * 2, kernel_size=1),
      # Glu Activation, d_model /= 2
      nn.GLU(dim=1),
      # 1D Depthwise Convolution
      nn.Conv1d(in_channels=d_model, out_channels=d_model, kernel_size=kernel_size,
padding='same', groups=d_model),
      nn.BatchNorm1d(d_model),  # Batchnorm
      nn.SiLU(),  # Swish Activation
      # Pointwise Convolution
      nn.Conv1d(in_channels=d_model, out_channels=d_model, kernel_size=1),
      nn.Dropout(dropout) # Dropout
    )
  def forward(self, x):
```

```
    # Layernorm
    x = self.ln(x)
    # length x batch x d_model to
    # batch x d_model x length
    x = x.permute(1, 2, 0)
    # All Remained Layers
    x = self.layer(x)
    # batch x d_model x length to
    # length x batch x d_model
    x = x.permute(2, 0, 1)
    return x
```

最後我也有照論文架構實作 PreNet，藉由兩次的 Conv2d with kernel_size = 3, stride = 2 將 size 從 40 減少為 9，並將 dimension 拓展為 d_model，再經過一層 Linear 和一層 Dropout 給 encoder，程式碼如下。

```
# Convolution Subsampling
class ConvSubsampling(nn.Module):
  def __init__(self, d_model):
    super(ConvSubsampling, self).__init__()
    self.layer = nn.Sequential(
      # batch x 1 x length x 40 to
      # batch x d_model x length x ((40-1)//2)
      nn.Conv2d(in_channels=1, out_channels=d_model, kernel_size=3, stride=2),
      nn.ReLU(),
      # batch x d_model x length x ((40-1)//2) to
      # batch x d_model x length x (((40-1)//2-1)//2)
      nn.Conv2d(in_channels=d_model, out_channels=d_model, kernel_size=3, stride=2),
      nn.ReLU(),
    )
  def forward(self, x):
    # batch x length x 40 to
    # batch x 1 x length x 40 to
    # batch x d_model x length x (((40-1)//2-1)//2)
    x = self.layer(x.unsqueeze(1))
    # Get batch, d_model, length & d_input
    batch = x.size(0)
    d_model = x.size(1)
    length = x.size(2)
    d_input = x.size(3)
    # batch x d_model x length x d_input to
    # batch x length x (d_model x d_input) to
    x = x.permute(0, 2, 1, 3)
    x = x.contiguous().view(batch, length, d_model * d_input)
    return x

# PreNet
class PreNet(nn.Module):
  def __init__(self, d_input, d_model, dropout):
    super().__init__()
    # Convolution Subsampling
    # batch x length x 40 to
    # batch x length x (d_model x (((d_input - 1) // 2 - 1) // 2))
    self.conv_subsample = ConvSubsampling(d_model)
    # Linear layer
    # batch x length x (d_model x (((d_input - 1) // 2 - 1) // 2)) to
    # batch x length x d_model
    self.linear = nn.Linear(d_model * (((d_input - 1) // 2 - 1) // 2), d_model)
    # Dropout layer
    self.dropout = nn.Dropout(dropout)
  def forward(self, x):
    # Convolution Subsampling
    x = self.conv_subsample(x)
    # Linear layer
    x = self.linear(x)
    # Dropout layer
    x = self.dropout(x)
    return x
```

How to improve the performance

我實作的數據如下，number of layers 皆= 3。

d_model	dim_feedforward	nhead	kernel_size	val acc
128	4 * 128	8	31	0.8053
128	4 * 128	16	31	0.8070
128	4 * 128	16	63	0.8164
80	4 * 80	16	63	0.7717
128	4 * 128	32	63	0.8143
64	16 * 64	16	63	0.7479
144	4 * 144	16	63	0.8333
160	2 * 160	16	63	0.8377

可以觀察到和 transformer 一樣，普遍來說 d_model 越大，parameter size 越大時，acc 會越高。最後我將 d_model 設為 160、dim_feedforward 設為 480、nhead = 16、kernel_size = 63、parameter size = 495.84k，成功將 val acc 提高至 0.8377。

3. Reference

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser "Attention Is All You Need"

[2] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, Ruoming Pang "Conformer: Convolution-augmented Transformer for Speech Recognition"

[3] [論文筆記] Conformer Layer 介紹

[4] Depthwise 卷积与 Pointwise 卷积

[5] 激活函数 ReLU 和 SiLU 的区别

[6] Brief Review — SiLU: Sigmoid-weighted Linear Unit