模型建立-Models

房价预测

线性预测模型

```
class Net(nn.Module):
    def __init__(self,n_input,n_output,n_hidden):
        super().__init__()

        # Hidden layer
        self.l1 = nn.Linear(n_input,n_hidden)
        # Output layer
        self.l2 = nn.Linear(n_hidden,n_output)
        # ReLU func
        self.relu = nn.ReLU(inplace=True)

        self.features = nn.Sequential(
            self.l1,
            self.relu,
            self.l2
        )

    def forward(self,x):
        x1 = self.features(x)
        return x1
```

```
# Print model outline
net = Net(n_input, n_output, n_hidden)
print(net)
```

```
Net(
  (l1): Linear(in_features=80, out_features=32, bias=True)
  (l2): Linear(in_features=32, out_features=1, bias=True)
  (relu): ReLU(inplace=True)
  (features): Sequential(
    (0): Linear(in_features=80, out_features=32, bias=True)
    (1): ReLU(inplace=True)
    (2): Linear(in_features=32, out_features=1, bias=True)
  )
)
```

房价预测-线性回归模型

普通多层感知机-MLP

```
# 定义MLP模型
class Net(nn.Module):
    def __init__(self, input_dim, hidden_dim,
output_dim):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim,
hidden_dim)
        self.fc3 = nn.Linear(hidden_dim,
hidden_dim)
        self.fc4 = nn.Linear(hidden_dim,
output_dim)
        self.relu = nn.ReLU()

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.relu(out)
        out = self.fc3(out)
        out = self.relu(out)
        out = self.fc4(out)
        return out
```

这个网络模型是一个多层感知机（MLP）模型，由输入层、三个隐层和一个输出层组成，每个隐层包含若干个神经元，并且每个神经元都使用ReLU作为激活函数。该模型通过前向传播进行计算，将输入数据传递给输出层，输出层返回预测结果。

多层感知机是一种比较常用的神经网络结构，其优势在于可以学习非线性函数关系，特别适合于处理高维数据。在本问题中，由于需要对多个特征值进行预测，因此采用多层感知机较为适合。另外，ReLU作为激活函数的优势在于训练速度快，收敛速度快，且在实际应用中表现优异。

```
: model
```

```
: Net(
    (fc1): Linear(in_features=149, out_features=512, bias=True)
    (fc2): Linear(in_features=512, out_features=512, bias=True)
    (fc3): Linear(in_features=512, out_features=512, bias=True)
    (fc4): Linear(in_features=512, out_features=1, bias=True)
    (relu): ReLU()
  )
```

普通多层感知机

带Dropout和BatchNorm层的MLP

```
# 定义MLP模型
class Net(nn.Module):
    def __init__(self, input_dim, hidden_dim,
output_dim):
        super(Net, self).__init__()
        self.fc = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),

 nn.BatchNorm1d(num_features=hidden_dim),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.5),
            nn.Linear(hidden_dim, hidden_dim//2),

 nn.BatchNorm1d(num_features=hidden_dim//2),
            nn.ReLU(inplace=True),
            nn.Linear(hidden_dim // 2, output_dim))

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

建立了一种具有Dropout和BatchNorm优化的MLP模型。该模型使用了4个全连接层，前3个层包含ReLU激活函数和BatchNorm正则化操作，最后一层没有激活函数，用来预测目标。其中第二个和第三个层使用了Dropout操作以减轻过拟合的发生。

BatchNorm会将每个特征归一化至0均值和1方差的范围内，有利于加速模型的学习、提升泛化性能，避免了梯度弥散和训练不稳定等问题的出现；Dropout则是一种正则化方法，随机从神经元中删除一些神经元，使得模型更加鲁棒，减少过拟合的发生。

该模型选取4层全连接层，是为了对多维特征进行建模，并提高模型的拟合能力。ReLU作为激活函数，可以在大多数情况下快速收敛，并且在实际应用中表现优异；最后一层没有激活函数，是为了预测该问题中的连续型目标值，采用均方误差作为损失函数，使用Adam优化算法进行迭代。

```
In [3]: model

Out[3]: Net(
          (fc): Sequential(
            (0): Linear(in_features=149, out_features=256, bias=True)
            (1): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        =True)
            (2): ReLU(inplace=True)
            (3): Dropout(p=0.5, inplace=False)
            (4): Linear(in_features=256, out_features=128, bias=True)
            (5): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
        =True)
            (6): ReLU(inplace=True)
            (7): Linear(in_features=128, out_features=1, bias=True)
          )
        )
```

更复杂的多层感知机

Transfomer+MLP

```
class Mlp(nn.Module):
    def __init__(self, in_features,
hidden_features=None, act_layer=nn.GELU, drop=0.,
pred=True):
        super().__init__()
        # 定义线性层，将输入进行计算
        hidden_features = hidden_features or
in_features
```

```
        self.q = nn.Linear(in_features,
in_features)
        self.k = nn.Linear(in_features,
in_features)
        self.v = nn.Linear(in_features,
in_features)
        # 定义全连接层，作为MLP中的隐藏层
        self.fc1 = nn.Linear(in_features,
hidden_features)
        # 激活函数
        self.act = act_layer()
        # 是否存在输出层
        self.pred = pred
        if pred==True:
            # 预测模式下，输出层为单一的线性层
            self.fc2 = nn.Linear(hidden_features,1)
        else:
            # 训练模式下，输出层为与输入形状相同的线性层
            self.fc2 = nn.Linear(hidden_features,
in_features)
        # 定义Dropout层，防止过拟合
        self.drop = nn.Dropout(drop)

    def forward(self, x):
        x0 = x
        # 计算Attention分布
        q = self.q(x).unsqueeze(2)
        k = self.k(x).unsqueeze(2)
        v = self.v(x).unsqueeze(2)
        attn = (q @ k.transpose(-2, -1))
        attn = attn.softmax(dim=-1)
        # 计算加权和
        x = (attn @ v).squeeze(2)
        x += x0
        # 计算MLP中的隐藏层与输出层
        x1 = x
        x = self.fc1(x)
        x = self.act(x)
        x = self.drop(x)
        x = self.fc2(x)
```

```
        x = self.drop(x)
        # 如果不存在输出层，则将输出与输入相加
        if self.pred==False:
            x += x1
        x = x.squeeze(0)
        # 返回输出值
        return x

class TF(nn.Module):
    def __init__(self, in_features, drop=0.):
        super().__init__()
        # 定义多层MLP结构，包括一个隐藏层和一个输出层
        self.Block1 = Mlp(in_features=in_features,
hidden_features=64, act_layer=nn.GELU, drop=drop,
pred=False)
        self.Block2 = Mlp(in_features=in_features,
hidden_features=64, act_layer=nn.GELU, drop=drop,
pred=True)

    def forward(self, x):
        # 前向传播，将输入经过多层MLP结构得到输出
        return self.Block2(self.Block1(x))
```

```
Out[22]: TF(
           (Block1): Mlp(
             (q): Linear(in_features=331, out_features=331, bias=True)
             (k): Linear(in_features=331, out_features=331, bias=True)
             (v): Linear(in_features=331, out_features=331, bias=True)
             (fc1): Linear(in_features=331, out_features=64, bias=True)
             (act): GELU(approximate='none')
             (fc2): Linear(in_features=64, out_features=331, bias=True)
             (drop): Dropout(p=0.0, inplace=False)
           )
           (Block2): Mlp(
             (q): Linear(in_features=331, out_features=331, bias=True)
             (k): Linear(in_features=331, out_features=331, bias=True)
             (v): Linear(in_features=331, out_features=331, bias=True)
             (fc1): Linear(in_features=331, out_features=64, bias=True)
             (act): GELU(approximate='none')
             (fc2): Linear(in_features=64, out_features=1, bias=True)
             (drop): Dropout(p=0.0, inplace=False)
           )
         )
```

更更复杂的MLP

网络模型是一种被称为Transformer+MLP的结构。Transformer+MLP结构在自然语言处理领域得到了广泛应用，并逐渐在其他领域得到了推广。该结构是Transformer和MLP的结合，充分利用了Transformer中的注意力机制和MLP中的全连接层。

具体来说，该结构包括一个Transformer block，一个MLP block以及相应的输入/输出层。Transformer block包括多头自注意力层和前向传输层，用于进行特征提取和编码；MLP block包括多个全连接层和激活函数，用于特征的处理和预测。两个block的输出被求和后作为最终的输出。

在本代码中，首先定义了多头注意力层（包括Q,K,V）用于捕捉输入数据之间的关系；其次定义了前向传输层，对数据进行全局特征提取和编码；然后，定义了传统的MLP层，实现特征转换和预测目标。最后，输入数据经过Transformer block和MLP block进行特征提取、编码、预测，得到最终输出。

在整个网络模型中，由于Transformer的自注意力机制能够很好的捕捉输入数据之间的关系，因此可以提高模型在数据集上的表示能力；而MLP block结构可以应对不同性质的特征，非常有效。此外，由于该结构没有使用卷积层，对于图像数据、时间序列数据等非结构化数据具有较强的适应性。

MNIST数字识别

CNN

```
#构建网络模型
class CNN_Net(nn.Module):
    def __init__(self):
        super(CNN_Net,self).__init__()
        #图片 1*28*28
        self.conv1 = nn.Conv2d(1,6,5) #24*24*20
        self.pool = nn.MaxPool2d(2,2) # 12*12*20
        self.conv2 = nn.Conv2d(6,16,3)# 10*10*40
        #5*5*40
        self.fc1 = nn.Linear(5*5*16,120)
        self.fc2 = nn.Linear(120,84)
        self.fc3 = nn.Linear(84,10)
    def forward(self,x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
```

```
        x = x.view(-1,5*5*16)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
#创建模型
net = CNN_Net().to(device)
```

具体来说，该模型包括两个卷积层和三个全连接层。第一个卷积层使用55*的卷积核和6个卷积核，输出大小为24246；接着使用*22的最大池化层，将输出大小缩小一半；第二个卷积层使用33*的卷积核和16个卷积核，输出大小为101016；再次进行*22的最大池化，输出大小变为5516；最后经过两个全连接层和一个输出层，输出10个数字的概率分布，以表示识别到的数字。

因此，该模型是为了提高模型的表现能力、提高模型分类准确性而设计的，可以用于解决分类问题。其卷积层用于特征提取，池化层用于下采样；全连接层用于特征转换和预测目标。因此，该模型可以针对大多数图像分类问题使用，并具有较强的通用性。

```
In [5]: net
Out[5]: CNN_Net(
          (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
          (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fals
        e)
          (conv2): Conv2d(6, 16, kernel_size=(3, 3), stride=(1, 1))
          (fc1): Linear(in_features=400, out_features=120, bias=True)
          (fc2): Linear(in_features=120, out_features=84, bias=True)
          (fc3): Linear(in_features=84, out_features=10, bias=True)
        )
```

CNN模型

RNN (LSTM)

```
class RNNnet(nn.Module):
    def __init__(self):
        super(RNNnet, self).__init__()
        self.rnn = nn.LSTM(
            input_size=INPUT_SIZE,
            hidden_size=64,
            num_layers=1,
            batch_first=True
        )
```

```
        self.out = nn.Linear(64, 10)

    def forward(self, x):
        # x shape (batch, time_step, input_size)
        # r_out shape (batch, time_step,
output_size)
        # h_n shape (n_layers, batch, hidden_size)
        # h_c shape (n_layers, batch, hidden_size)
        r_out, (h_n, h_c) = self.rnn(x, None)
        # choose r_out at the last time step
        out = self.out(r_out[:, -1, :])
        return out
```

具体来说，该模型包括一个LSTM层和一个全连接层。LSTM层用于建模序列，对输入图像序列进行逐步处理并保留关键信息，而不仅仅是通过静态特征提取来处理整张图像。随后，将LSTM层的最后一个时间步输出传入全连接层进行分类，以输出输入的10个数字的概率分布，表示识别到的数字。

LSTM是RNN中的一种特殊形式，用于解决序列数据中的长依赖问题，并且具有较好的记忆性，因此常被用于文本、音频、图像序列处理。在本代码中，使用LSTM来处理MNIST图像序列，也是一种有效的解决方案。

因此，相较于卷积神经网络中常用的CNN模型，基于LSTM的RNN模型能够很好地利用时间序列信息，对图像识别具有更好的自适应性和鲁棒性。此外，对于图像中存在长距离依赖的问题，LSTM能够较好地处理，因此在某些情况下，LSTM也可以成为图像识别的解决方案之一。

```
In [4]: model=RNNnet()

In [5]: model

Out[5]: RNNnet(
          (rnn): LSTM(28, 64, batch_first=True)
          (out): Linear(in_features=64, out_features=10, bias=True)
        )
```

LSTM模型也可以用于图像领域

泰坦尼克号存活预测

简单线性回归模型

```
class LinearRegression(nn.Module):
    def __init__(self,input_size,output_size):
        super(LinearRegression,self).__init__()
        self.f1 = nn.Linear(input_dim, 2000)
        self.f2 = nn.Linear(2000, output_dim)


    def forward(self,x):
        x = self.f1(x)
        x = F.leaky_relu(x)
        x = F.dropout(x, p = 0.3)
        x = self.f2(x)
        return  F.sigmoid(x)
```

该代码中的网络模型是一个简单的线性回归模型，用于解决Titanic存活率问题。

具体来说，该模型包括两个全连接层。第一个全连接层接收输入数据并将其映射到一个大小为2000的中间层；中间层采用leaky ReLU激活函数进行激活，以帮助提高模型的表达能力。同时，为了减轻过拟合，还使用了dropout技术进行正则化。第二个全连接层将中间层映射到输出层，并采用sigmoid激活函数进行输出，表示输出数据的概率。

因此，该模型是一个二分类模型，用于预测Titanic乘客是否存活。模型输入的特征包括从Titanic船上抽取的各种信息，如幸存者的等级、性别、年龄、船票费用、登船口岸等。这些特征将作为模型的输入，模型将通过学习抽取最相关的特征，并输出存活的概率。该模型使用了全连接层进行特征的线性组合、leaky ReLU进行非线性激活，以及dropout进行正则化，能够形成比较合理的特征抽取和模型预测，并在Titanic存活率问题中取得了较好的效果。

In [35]:
```
input_dim = 1730
output_dim = 2
learning_rate = 1
model = LinearRegression(input_dim,output_dim)
error = nn.CrossEntropyLoss()  #交叉熵损失
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum = 0.5)
```

In [36]:
```
model
```

简单先行回归模型，但效率良好

Out[36]:
```
LinearRegression(
  (f1): Linear(in_features=1730, out_features=2000, bias=True)
  (f2): Linear(in_features=2000, out_features=2, bias=True)
)
```

MLP多层感知机

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

        self.fc1 = nn.Linear(13, 13)
        self.dropout1 = nn.Dropout(0.1)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(13, 13)
        self.dropout2 = nn.Dropout(0.1)
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(13, 10)
        self.dropout3 = nn.Dropout(0.1)
        self.relu3 = nn.ReLU()
        self.fc4 = nn.Linear(10, 5)
        self.relu4 = nn.ReLU()
        self.fc5 = nn.Linear(5, 1)
        self.sigmoid1 = nn.Sigmoid()

    def forward(self, x):
        x = self.fc1(x)
        x = self.dropout1(x)
        x = self.relu1(x)
        x = self.fc2(x)
        x = self.dropout2(x)
        x = self.relu2(x)
        x = self.fc3(x)
        x = self.dropout3(x)
        x = self.relu3(x)
```

```
        x = self.fc4(x)
        x = self.relu4(x)
        x = self.fc5(x)
        x = self.sigmoid1(x)
        return x
```

代码中的网络模型是一个多层感知机（MLP）模型，用于解决回归问题。

具体来说，该模型包括5个全连接层，其中每个层之间使用了dropout技术进行正则化，以避免模型过拟合。每个全连接层都使用ReLU激活函数进行激活。最后一层是一个只有一个输出的全连接层，使用了sigmoid激活函数进行输出。

因此，该模型被设计为可以接受大小为13的输入，然后输出一个由sigmoid激活函数映射到[0,1]区间的标量值，该标量值用于回归预测问题中。输输入特征包括房屋面积、房龄、房间数量等13个房屋信息，预测房屋价格。

总体而言，该模型采用了多层感知机的结构，将一系列隐藏单元串联在一起，并在每个层之间加入了非线性激活函数和dropout正则化技术。这种结构使模型具有更强的表达能力，有助于提高回归预测的准确性和性能。

```
In [28]: model=MLP()

In [29]: model
```

MLP多层感知机模型

```
Out[29]: MLP(
           (fc1): Linear(in_features=13, out_features=13, bias=True)
           (dropout1): Dropout(p=0.1, inplace=False)
           (relu1): ReLU()
           (fc2): Linear(in_features=13, out_features=13, bias=True)
           (dropout2): Dropout(p=0.1, inplace=False)
           (relu2): ReLU()
           (fc3): Linear(in_features=13, out_features=10, bias=True)
           (dropout3): Dropout(p=0.1, inplace=False)
           (relu3): ReLU()
           (fc4): Linear(in_features=10, out_features=5, bias=True)
           (relu4): ReLU()
           (fc5): Linear(in_features=5, out_features=1, bias=True)
           (sigmoid1): Sigmoid()
         )
```

股票预测问题

LSTM模型

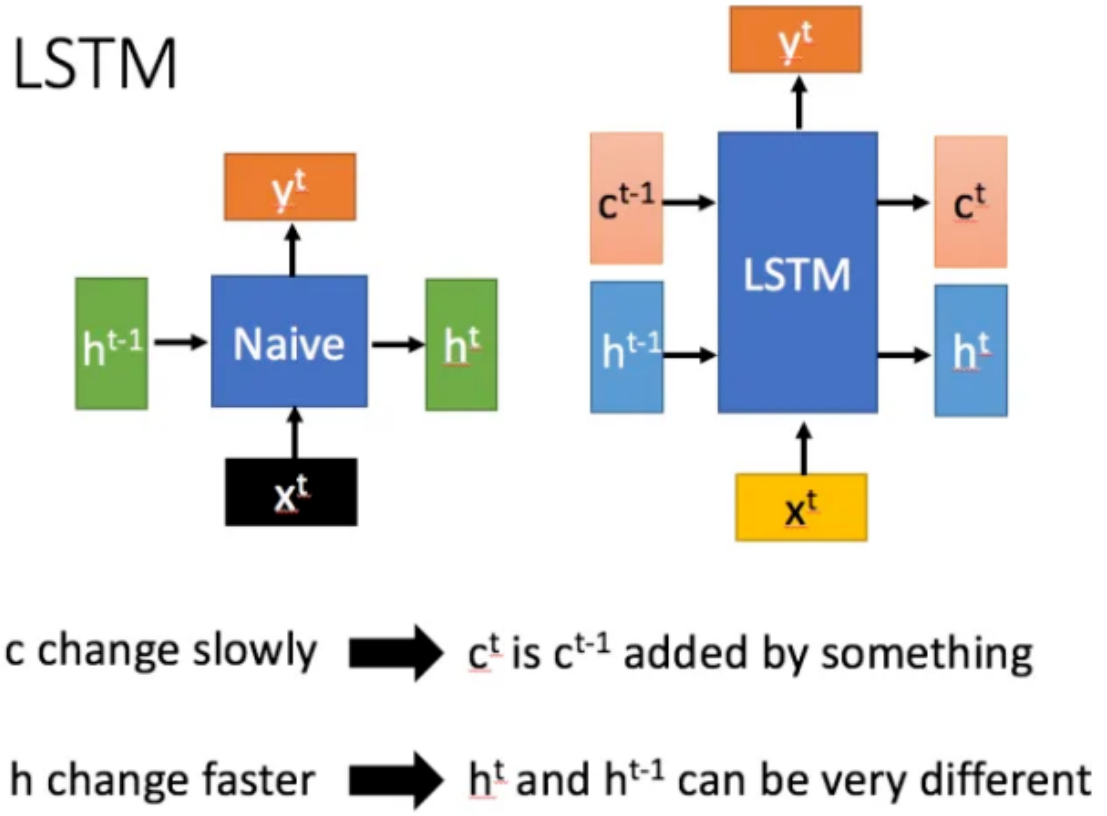
```
class LSTM(nn.Module):
    def __init__(self, input_size=8,
hidden_size=32, num_layers=1 , output_size=1 ,
dropout=0, batch_first=True):
        super(LSTM, self).__init__()
        # lstm的输入 #batch,seq_len, input_size
        self.hidden_size = hidden_size
        self.input_size = input_size
        self.num_layers = num_layers
        self.output_size = output_size
        self.dropout = dropout
        self.batch_first = batch_first
        self.rnn =
nn.LSTM(input_size=self.input_size,
hidden_size=self.hidden_size,
num_layers=self.num_layers,
batch_first=self.batch_first, dropout=self.dropout
)
        self.linear = nn.Linear(self.hidden_size,
self.output_size)

    def forward(self, x):
        out, (hidden, cell) = self.rnn(x)  #
x.shape : batch, seq_len, hidden_size , hn.shape
and cn.shape : num_layes * direction_numbers,
batch, hidden_size
        # a, b, c = hidden.shape
        # out = self.linear(hidden.reshape(a * b,
c))
        out = self.linear(hidden)
        return out
```

具体来说，该模型包含一个LSTM层和一个全连接层。LSTM层由输入大小、隐藏状态大小、层数和dropout等参数定义。LSTM层将输入序列映射到隐藏状态序列，并输出最后一个时刻的隐藏状态。输出隐藏状态后通过全连接层进行映射，以预测股票价格。

在训练过程中，模型可以接受包含价格历史数据的时间序列作为输入数据。每个时间步的输入包括过去的价格、交易量等8个特征。LSTM层对这些特征进行建模，从而捕捉时间序列中的相关性和周期性，以此预测未来的股票价格。

由于LSTM具有记忆功能和时间序列建模的能力，因此适用于预测具有时间相关性的问题。同时，该模型采用了一个全连接层进行计算输出，实现回归预测并输出股票的价格。




```
In [23]: model = LSTM(input_size=8, hidden_size=32, num_layers=2, output_size=1, dropout=0.1,

In [24]: model                                    LSTM模型

Out[24]: LSTM(
           (rnn): LSTM(8, 32, num_layers=2, batch_first=True, dropout=0.1)
           (linear): Linear(in_features=32, out_features=1, bias=True)
         )

In [25]: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

影视评论问题

RNN-GRU

```
#定义RNN分类器
class RNNClassifier(torch.nn.Module):
    def __init__(self, input_size, hidden_size,
output_size, n_layers=1, bidirectional=True):
        super(RNNClassifier, self).__init__()
        self.hidden_size = hidden_size
        self.n_layers = n_layers
        self.n_direction = 2 if bidirectional else
1
        self.embedding =
torch.nn.Embedding(input_size, hidden_size)
        self.gru = torch.nn.GRU(hidden_size,
hidden_size, n_layers, bidirectional=bidirectional)
        self.fc =
torch.nn.Linear(hidden_size*self.n_direction,
output_size)

    def _init_hidden(self, batch_size):
        hidden =
torch.zeros(self.n_layers*self.n_direction,
batch_size, self.hidden_size)
        return hidden.to(device)

    def forward(self, input, seq_lengths):
        input = input.t()
        batch_size = input.size(1)
        hidden = self._init_hidden(batch_size)
        embedding = self.embedding(input)
        gru_input = pack_padded_sequence(embedding,
seq_lengths.cpu())
        output, hidden = self.gru(gru_input,
hidden)
        if self.n_direction == 2:
```

```
            hidden_cat = torch.cat((hidden[-1],
hidden[-2]), dim=1)
        else:
            hidden_cat = hidden[-1]
        fc_output = self.fc(hidden_cat)
        return fc_output
```

具体来说，该模型由一个嵌入层、一个GRU层和一个全连接层组成。嵌入层将输入的单词索引转换为词向量，GRU层对输入的词向量进行建模，从而捕捉词序列中的句法和语法信息，全连接层用于计算输出，以预测评论的情感。

在训练过程中，模型可以接受一个由单词索引组成并经过填充的序列作为输入数据。为了更好地处理变长的输入序列，模型采用pack_padded_sequence方法对序列长度进行压缩，以避免计算冗余的填充数据。该方法会将序列长度发给GRU层，以便其根据序列长度动态地计算每个时刻的隐藏状态。

该模型还支持双向部署，即双向GRU，通过同时考虑正向和反向的隐藏状态，从而进一步捕捉序列中的信息。

总体而言，该模型采用了GRU网络结构，可以对变长序列进行建模，并结合嵌入层和全连接层计算模型输出，实现情感分类预测任务。

```
In [17]: classifier

Out[17]: RNNClassifier(
           (embedding): Embedding(128, 128)
           (gru): GRU(128, 128, num_layers=2, bidirectional=True)
           (fc): Linear(in_features=256, out_features=5, bias=True)
         )
```

RNN分类器

拓展-基于BERT+Transformer的文本分类模型

```
# 定义Bert分类器
class BertClassifier(nn.Module):
    def __init__(self,bert_config,num_labels):
        super().__init__()
        #定义BERT模型
```

```
        self.bert = BertModel(config = bert_config)
        #定义分类器 线性分类器
        self.classifier =
nn.Linear(bert_config.hidden_size,num_labels)

    def forward(self, input_ids, attention_mask,
token_type_ids):
        # BERT的输出
        bert_output =
self.bert(input_ids=input_ids,
attention_mask=attention_mask,
token_type_ids=token_type_ids)
        # 取[CLS]位置的pooled output
        pooled = bert_output[1]
        # 分类
        logits = self.classifier(pooled)
        # 返回softmax后结果
        return torch.softmax(logits, dim=1)

# Bert+BiLSTM，用法与BertClassifier一样，可直接在train
里面调用
class BertLstmClassifier(nn.Module):
    def __init__(self, bert_config, num_labels):
        super().__init__()
        self.bert = BertModel(config=bert_config)
        self.lstm =
nn.LSTM(input_size=bert_config.hidden_size,
hidden_size=bert_config.hidden_size, num_layers=2,
batch_first=True, bidirectional=True)
        self.classifier =
nn.Linear(bert_config.hidden_size*2, num_labels)  #
双向LSTM 需要乘以2
        self.softmax = nn.Softmax(dim=1)

    def forward(self, input_ids, attention_mask,
token_type_ids):
        outputs = self.bert(input_ids=input_ids,
attention_mask=attention_mask,
token_type_ids=token_type_ids)
```

```
        last_hidden_state =
outputs.last_hidden_state
        out, _ = self.lstm(last_hidden_state)
        logits = self.classifier(out[:, -1, :]) # 取
最后时刻的输出
        return self.softmax(logits)
```

Bert分类器的主要结构是基于BERT模型，包含一个预训练的BERT模型和一个线性分类器。在前向传播过程中，输入文本经过BERT模型转换为对每个token的特征向量表示，然后在该模型的分类器中计算相应的输出概率分布。

Bert+BiLSTM分类器则在BERT分类器的基础上加入了一个双向LSTM层。LSTM层接收BERT模型输出的特征表示作为输入，进一步捕捉输入序列中的时间依赖关系。随后，最后时刻的LSTM输出作为分类器的输入，用于预测文本分类。

两个模型都是基于预训练模型进行微调，可以通过不同的参数调整和fine-tuning策略改进性能。特别的，在BERT模型中，该模型通过对大规模语料库进行预训练，可以在多种自然语言处理任务中取得优秀的表现，包括文本分类、情感分析等任务。在深度学习中，预训练技术是一种重要的方法，它可以充分利用大规模无标注语料库的统计信息，预训练出通用的语义表示，进而加速和提高特定任务的训练效果。

In [8]:
```
model
```

Out[8]:
```
BertClassifier(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(21128, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0-11): 12 x BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
  (classifier): Linear(in_features=768, out_features=10, bias=True)
)
```

Transfomer+BERT BiLSTM