自然语言处理中的神经网络基础

汇报人：程 征

内容概要

- 感知器
- 常用激活函数
- Softmax回归
- 多层感知器
- 卷积神经网络

感知器

感知器（Perceptron）是最简单也是最早出现的机器学习模型，其灵感直接来源于生产生活的实践。例如：在公司面试时，经常由多位面试官对一位面试者打分，最终将多位面试官的打分求和，如果分数超过一定的阈值，则录用该面试者，否则不予录用。

感知器

假设有n位面试官，每人的打分分别为$x_1, x_2, \dots, x_n$，则总分$s = x_1 + x_2 + \cdots + x_n$，如果$s \geq t$，则给与录用，其中$t$被称为阈值，$\boldsymbol{x_1}, \boldsymbol{x_2}, \dots, \boldsymbol{x_n}$被称为输入，可以使用向量$\boldsymbol{x} = [\boldsymbol{x_1}, \boldsymbol{x_2}, \dots, \boldsymbol{x_n}]$表示。然而，在这些面试官中，有一些经验比较丰富，一些则是刚入门的新手，如果简单地将他们的打分进行相加，最终的得分显然不够客观，因此可以通过对面试官的打分进行加权的方法解决，即为经验丰富的面试官赋予较高的权重，而为新手赋予较低的权重。假设n为面试官的权重分别为$w_1, w_2, \dots, w_n$，则最终的分数为$\boldsymbol{s} = \boldsymbol{w_1 x_1} + \boldsymbol{w_2 x_2} + \cdots + \boldsymbol{w_n x_n}$，同样使用向量$\mathbf{w} = [\boldsymbol{w_1}, \boldsymbol{w_2}, \dots, \boldsymbol{w_n}]$表示n个权重，则分数可以写成权重向量和输入向量的点积，即$s = w \cdot x$，于是最终的输入y为：

$$y = \begin{cases} 1, & \text{如果} s \geq t \\ 0, & \text{否则} \end{cases} = \begin{cases} 1, & \text{如果} w \cdot x \geq t \\ 0, & \text{否则} \end{cases}$$

感知器

式中，输出$y = 1$表示录用，$y = 0$表示不录用。这就是感知器模型，其实还可以写成以下的形式：

$$y = \begin{cases} 1, & \text{如果} w \cdot x + b \geq 0 \\ 0, & \text{否则} \end{cases}$$

式中，$\mathrm{b} = -t$,又被称为偏差项（Bias）。

当使用感知器模型时，有两个棘手的问题需要加以解决。首先是如何将一个问题的原始输入（Raw Input）转换成输入向量x，此过程又被称为特征提取（Feature Extraction）；其次是如何合理地设置权重w和偏差项b（也被称为模型参数），此过程又被称为参数学习（也称参数优化或模型训练）。

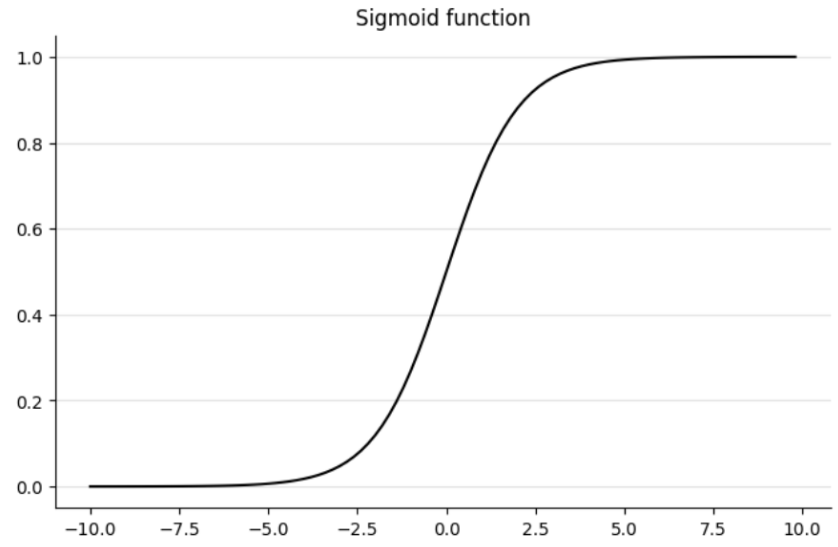
常用激活函数

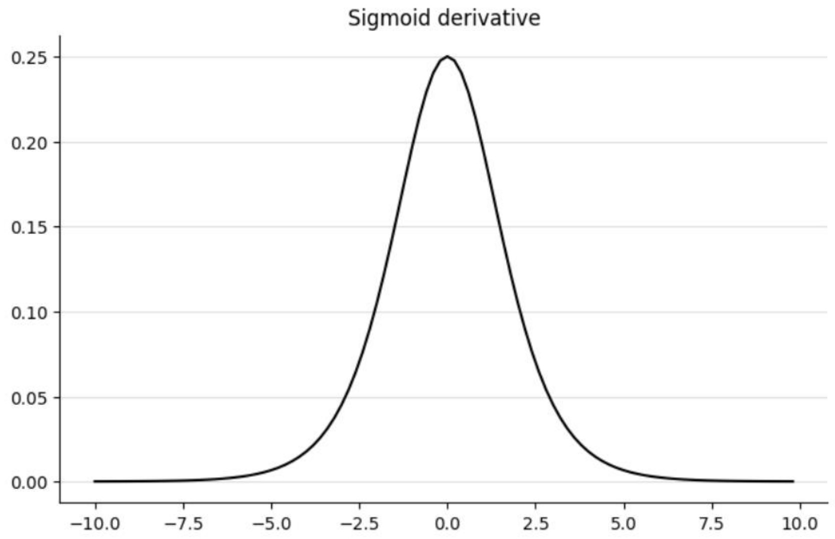
Sigmoid函数

$$\sigma = \frac{1}{1+e^{-z}}$$

其中e为自然常数（约为2.71828），其中z是自变量，σ是因变量，z的值通常为线性模型的输出结果（如线性回归），Sigmoid函数是一个S型的图像。

$$sigmoid'(z) = \sigma(z) * (1-\sigma(z))$$

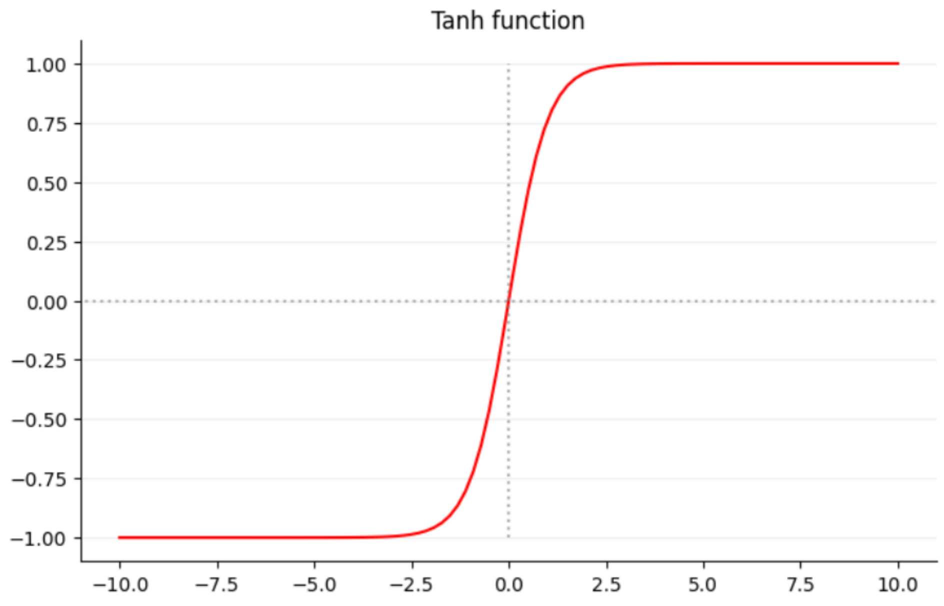


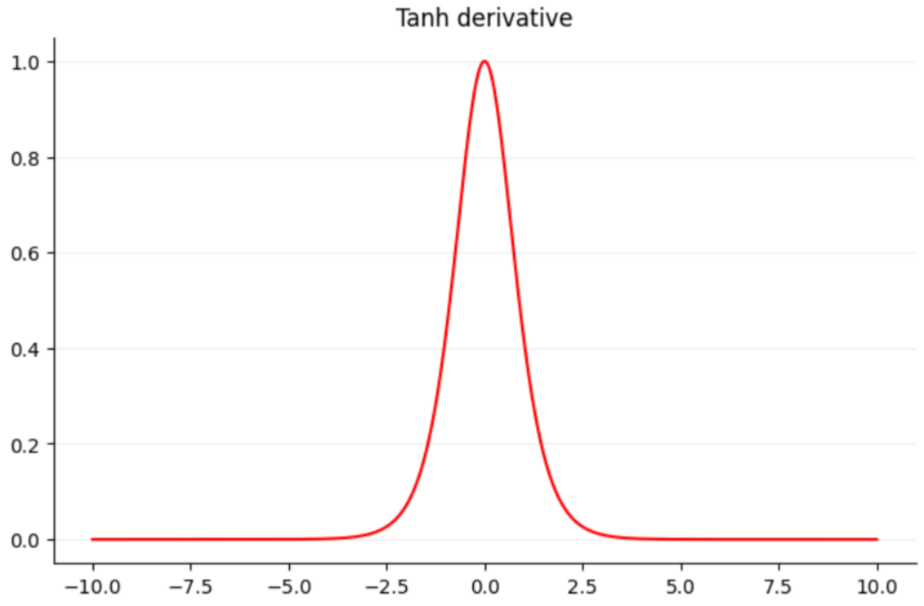


tanh函数

tanh，双曲正切函数，其性质与Sigmoid类似，将值压缩到（-1,1）区间内，公式如下：

$$tanh\colon \sigma = \frac{e^{2z}-1}{e^{2z}+1} \qquad tanh'(z) = 1 - tanh^2(z)$$





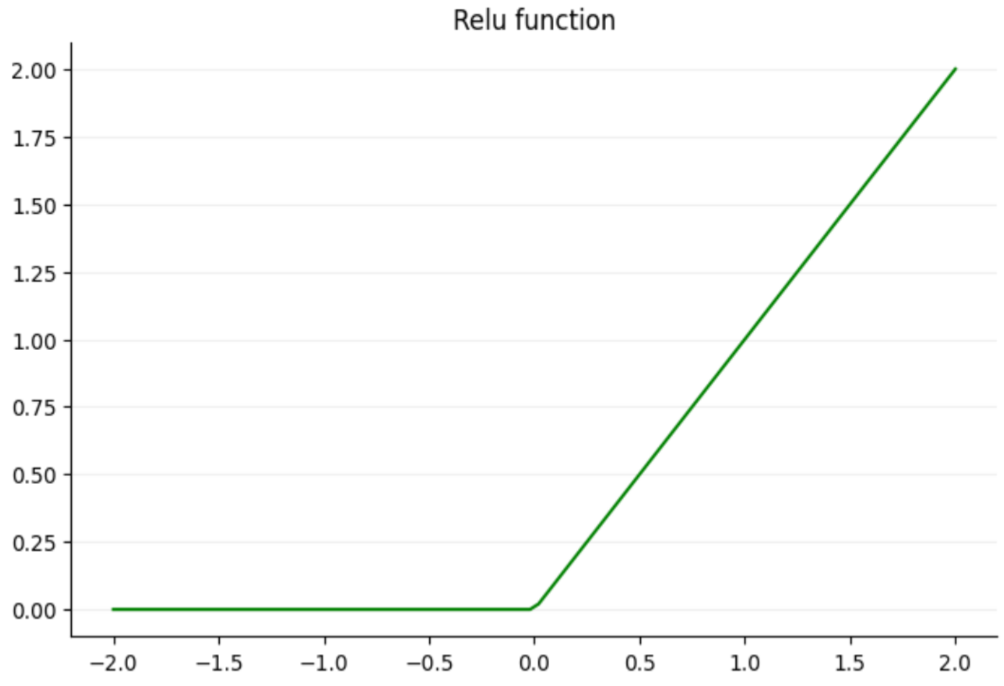
ReLU函数

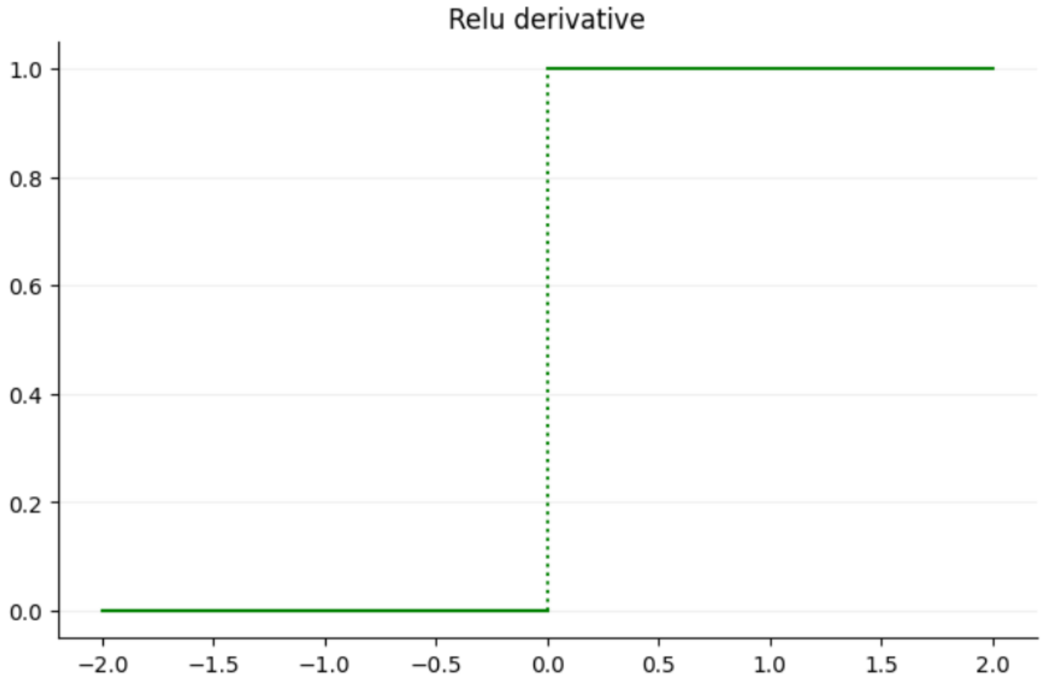
ReLU(Rectified Linear Unit)，函数又名整流线型单元函数，是现在神经网络领域中的宠儿，应用甚至比sigmoid更广泛。ReLU提供了一个很简单的非线性变换：当输入的自变量大于0时，直接输出该值；当输入的自变量小于等于0时，输出0。公式如下：

$$ReLU:\ \sigma = \begin{cases} z, & z > 0 \\ 0, & z \leq 0 \end{cases}$$

ReLU函数是一个非常简单的函数，本质就是max(0,z)，max函数会从输入的数值中选择较大的那个值进行输出，以达到保留正数元素，将负元素清零的作用。

ReLU函数





ReLU函数的图像如上左图所示。当输入为正数时，ReLU函数的导数为1，当为负数时，ReLU函数的导数为0，当输入为0时，ReLU函数不可导，其导函数图像如上右图所示。

Softmax回归

Sigmoid回归虽然可以用于处理二元分类问题，但是很多现实问题的类别可能不止两个，如手写数字识别，输出属于0 ~ 9共10个数字中的一个，即有10个类别。那么，如何处理多元分类问题呢？其中一种方法和Sigmoid回归的思想类似，即对第i个类别使用线性回归打一个分数，$z_i = w_{i1}x_1 + w_{i2}x_2 + \cdots + w_{in}x_n + b_i$。式中，$w_{ij}$表示第$i$个类别对应的第$j$个输入的权重。然后，对多个分数使用指数函数进行归一化计算，并获得一个输入属于某个类别的概率。该方法又称Softmax回归，具体公式为：

$$y_i = \mathrm{Softmax}(\boldsymbol{z})_i = \frac{\mathrm{e}^{z_i}}{\mathrm{e}^{z_1} + \mathrm{e}^{z_2} + \cdots + \mathrm{e}^{z_m}}$$

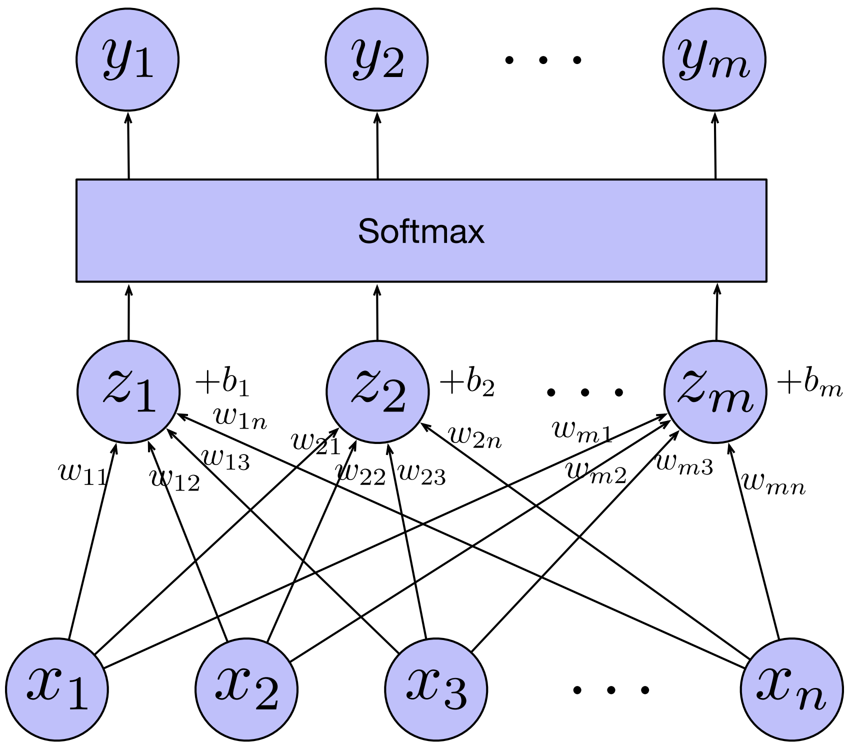
式中，z表示向量$[z_1, z_2, \dots, z_m]$；m表示类别数；y_i表示第i个类别的概率。

Softmax回归

当$m = 2$，即处理二元分类问题时，上述公式可以写为：

$$y_1 = \frac{e^{z1}}{e^{z1} + e^{z2}} = \frac{1}{1 + e^{-(z_1 - z_2)}}$$

此公式即Sigmoid函数形式，也就是Sigmoid函数是Softmax函数在处理二元分类问题时的一个特例。



Softmax回归模型示意图

Softmax回归

进一步地，将Softmax回归模型公式展开，其形式为：

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \mathrm{Softmax}\begin{pmatrix} w_{11}x_1 + w_{12}x_2 + \cdots + w_{1n}x_n + b_1 \\ w_{21}x_1 + w_{22}x_2 + \cdots + w_{2n}x_n + b_2 \\ \vdots \\ w_{m1}x_1 + w_{m2}x_2 + \cdots + w_{mn}x_n + b_m \end{pmatrix}$$

然后，可以使用矩阵乘法的形式重写该公式，具体为：

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \mathrm{Softmax}\left(\begin{bmatrix} w_{11}, w_{12}, \cdots, w_{1n} \\ w_{21}, w_{22}, \cdots, w_{2n} \\ \vdots \\ w_{m1}, w_{m2}, \cdots, w_{mn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}\right)$$

Softmax回归

更进一步地，可以使用张量表示输入、输出以及其中的参数，即：

$$\boldsymbol{y} = \mathrm{Softmax}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

式中，$x = [x_1, x_2, \dots, x_n]^{\mathrm{T}}$， $\mathrm{y} = [y_1, y_2, \dots, y_m]^{\mathrm{T}}$，$W = \begin{bmatrix} w_{11}, w_{12}, \dots, w_{1n} \\ w_{21}, w_{22}, \dots, w_{2n} \\ \vdots \\ w_{m1}, w_{m2}, \dots, w_{mn} \end{bmatrix}$，
$\mathrm{b} = [b_1, b_2, \dots, b_m]^{\mathrm{T}}$，对向量$x$进行执行$Wx + b$运算又被称为对$x$的线性映射或线性变换。

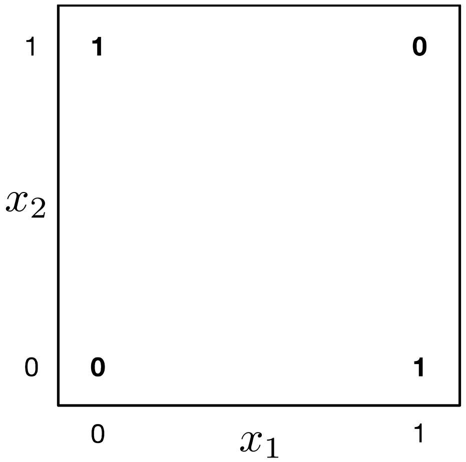
代码实现

Sigmoid、tanh、ReLu等各种激活函数包在torch.nn.functional中，实现对输入按元素进行非线性计算，调用方式如下。

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

torch.manual_seed(1234)

linear = nn.Linear(16,2)  # 输入维度16，输出维度2
inputs = torch.rand(4,16)  # 创建一个形状为（4，16）的随机张量
outputs = linear(inputs)

# 方式一 （sigmoid、tanh会报警告信息）
activation = F.sigmoid(outputs)
activation = F.tanh(outputs)
activation = F.relu(outputs)
activation = F.softmax(outputs, dim=1)  # 沿第2维进行Softmax运算
# 方式二
activation = torch.sigmoid(outputs)
activation = torch.tanh(outputs)
activation = torch.relu(outputs)
activation = torch.softmax(outputs, dim=1)
```

多层感知器

线性回归、逻辑回归、Softmax回归等模型本质上都是线性模型，然而现实世界中很多真实的问题不都是线性可分的，即无法使用一条直接、平面或超平面分割不同的类别，其中典型的问题就是异或问题（Exclusive OR，XOR），即假设输入为 x_1和x_2， 如果它们相同，即当$x_1 = 0$、 $x_2 = 0$或$x_1 = 1$、 $x_2 = 1$时，输出$y = 0$；如果它们不相同，即当$x_1 = 0$、$x_2 = 1$或$x_1 = 1$、 $x_2 = 0$时，输出$y = 1$。此时，无法使用线性分类器恰当地将输入划分到正确的类别。



多层感知器

多层感知器（Multi-layer Perception，MLP）是解决线性不可分问题的一种解决方案。多层感知器指的是堆叠多层线性分类器，并在中间层（也叫隐含层，Hidden layer）增加非线性激活函数。例如，可以设计如下的多层感知器：

$$\boldsymbol{z} = \boldsymbol{W}^{[1]}\boldsymbol{x} + \boldsymbol{b}^{[1]}$$
$$\boldsymbol{h} = \mathrm{ReLU}(\boldsymbol{z})$$
$$\boldsymbol{y} = \boldsymbol{W}^{[2]}\boldsymbol{h} + \boldsymbol{b}^{[2]}$$

如果将相应的参数进行如下的设置：$W^{[1]} = \begin{bmatrix} 1,1 \\ 1,1 \end{bmatrix}$，$b^{[1]} = [0, -1]^{\mathrm{T}}$，$W^{[2]} = [1, -2]$，$b^{[2]} = [0]$，即可解决异或问题。

多层感知器

多层感知器解决异或问题推导:

x_1	x_2	y
0	0	0
1	1	0
0	1	1
1	0	1

$$\begin{cases} z = w^1 x + b^1 \\ h = \mathrm{ReLu}(z) \\ y = w^2 h + b^2 \end{cases}$$

令 $X = \begin{pmatrix} 0 & 0 \\ 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}$, $w^1 = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$, $b^1 = (0, -1)^T$,

$w^2 = (1, -2)$, $b^2 = 0$.

则有 $z = w^1 x + b^1$

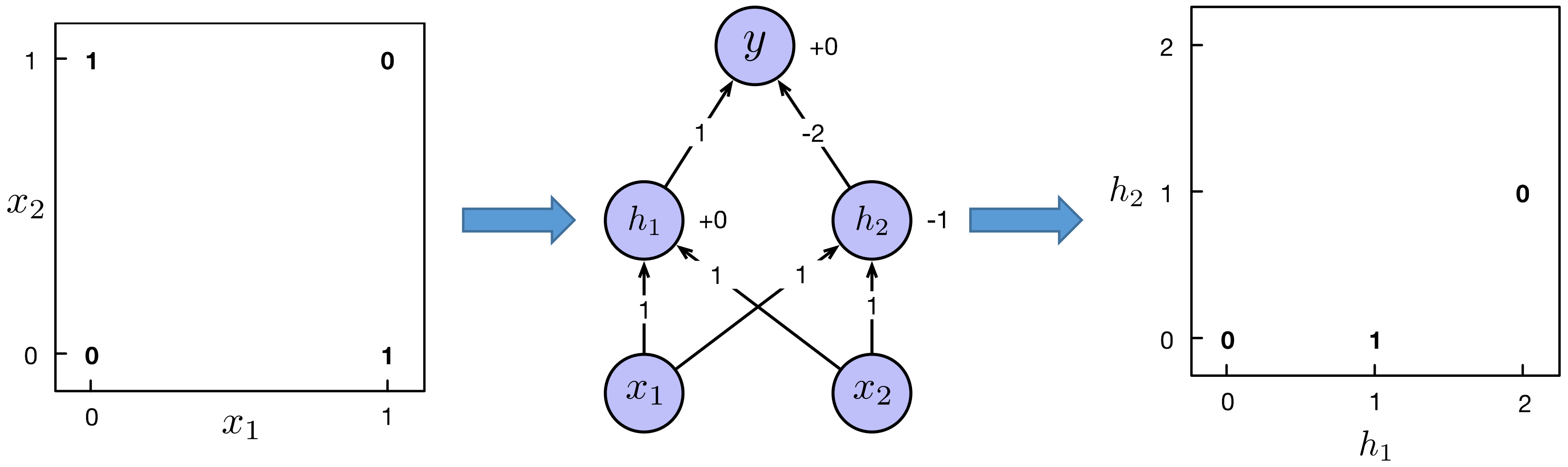
$$= \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & 2 & 1 & 1 \\ 0 & 2 & 1 & 1 \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & 2 & 1 & 1 \\ -1 & 1 & 0 & 0 \end{pmatrix}$$

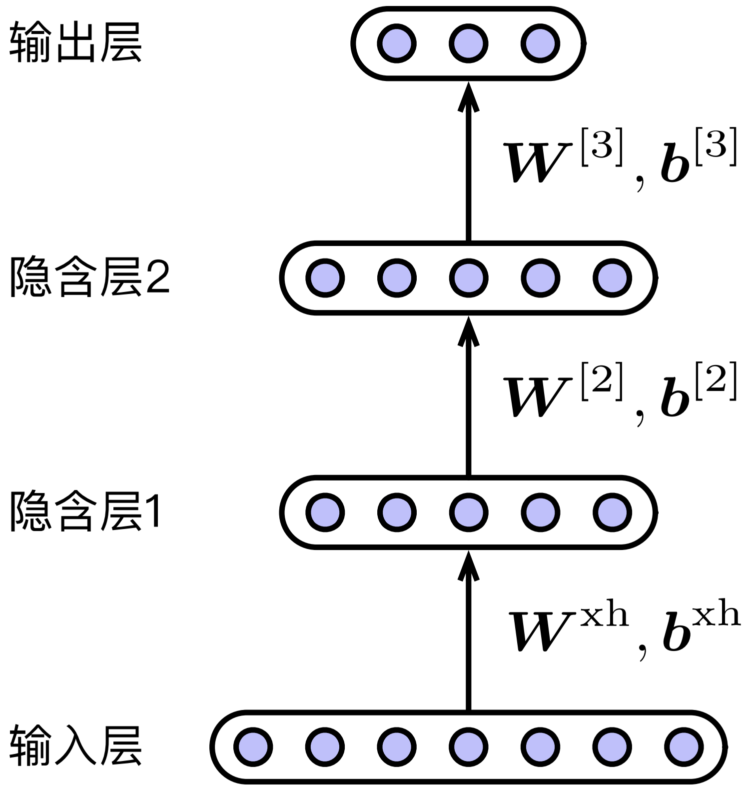
$$\xrightarrow{\mathrm{ReLu}} \underbrace{\begin{pmatrix} 0 & 2 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}}_{h} \xrightarrow{w^2 h + b^2} (1, -2)\begin{pmatrix} 0 & 2 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} + 0$$

$$= \underbrace{(0\ 0\ 1\ 1)}_{即\ \hat{y}}$$



多层感知器

更一般的多层感知器如右图所示，其中引入了更多的隐含层（没有画出非线性激活函数），并将输出层设置为多类分类层（使用Softmax函数）。输入层和输出层的大小一般是固定的，与输入数据的维度以及所处理的问题类别相对应，而隐含层的大小、层数和激活函数的类型需要根据经验以及实验结果设置，它们又被称为超参数（Hyper-parameter），一般来讲，隐含层越大，层数越多，即模型的参数越多、容量越大，多层感知器的表达能力就越强，但是此时较难优化网络的参数。而如果隐含层太小、层数过少，则模型的表达能力会不足。为了在模型容量和学习难度中间找到一个平衡点，需要根据不同的问题和数据，通过调参过程寻找合适的超参数组合。



多层感知器示意图

模型实现

```python
class MLP(nn.Module):
    def __init__(self, input_dim=300, hidden_dim=256, num_class=10):
        super(MLP, self).__init__()
        self.input_layer = nn.Linear(input_dim, hidden_dim)
        self.hidden_layer = nn.Linear(hidden_dim, hidden_dim)
        self.classifier = nn.Linear(hidden_dim, num_class)
        self.activate = torch.relu
    def forward(self, inputs):
        inputs = inputs.view(-1, 28*28)  # 维度变换，目的是为了便于计算
        z1 = self.input_layer(inputs)
        p1 = self.activate(z1)
        z2 = self.hidden_layer(p1)
        p2 = self.activate(z2)
        outputs = self.classifier(p2)
        return outputs
```

卷积神经网络

在多层感知器中，每层输入的各个元素都需要乘以一个独立的参数（权重），这一层又叫作全连接层（Full Connected Layer）或稠密层（Dense Layer）。然而，对于某些类型的任务，这样做并不合适，如在图像识别任务中，如果对每个像素赋予独立的参数，一旦待识别物体的位置出现轻微移动，识别结果可能会发生较大的变化；在情感分类任务中，句子的情感极性往往由个别词或短语决定，而这些决定性的词或短语在句子中的位置并不固定，使用全连接层很难捕捉这种关键的局部信息。

为了解决上述问题，一个非常直接的想法是使用一个小的稠密层提取这些局部特征，如图像中固定大小的像素区域、文本中的词的N-gram等。

N-gram：源于马尔科夫假设，“下一个词出现的概率只依赖于它前面n-1个词”，满足这种假设的模型，被称为N元语法（N-gram）模型。

卷积神经网络

为了解决关键信息位置不固定的问题，可以依次扫描输入的每个区域，该操作又被称为卷积（Convolution）操作。其中，每个小的、用于提取局部特征的稠密层又被称为卷积核（Kernel）或者滤波器（Filter）。卷积操作的结果还可以进行进一步聚合，这一过程被称为池化（Pooling）操作，常用的池化操作有最大池化、平均池化和加和池化等。以最大池化为例，其含义是仅保留最有意义的局部特征。如在情感分类任务中，保留的是句子中对于分类最关键的N-gram信息。

然而，如果仅使用一个卷积核，则只能提取单一类型的局部特征。而在实际问题中，往往需要提取很多种局部特征，如在情感分类中不同的情感词或者词组等。因此，在进行卷积操作时，可以使用多个卷积核提取不同种类的局部特征。

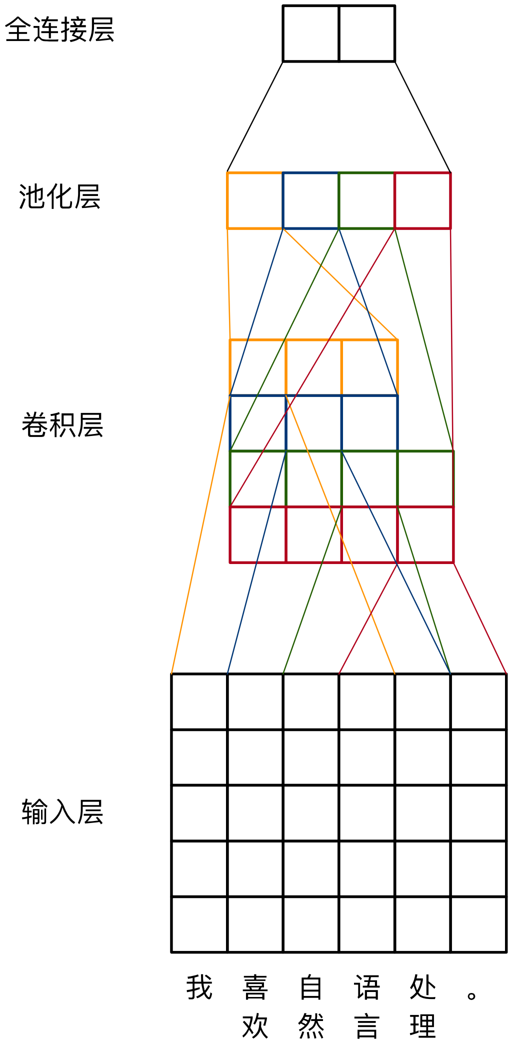
卷积神经网络

卷积核的构造方式大致有两种，一种是使用不同组的参数，并且使用不同的初始化参数，获得不同的卷积核；另一种是提取不同尺度的局部特征，如在情感分类中提取不同大小的N-gram。

既然多个卷积核输出多个特征，那么这些特征对于最终分类结果的判断，到底哪些比较重要，哪些不重要呢？其实只要在经过一个全连接的分类层就可以做出最终的决策。最后，还可以将多个卷积层加池化层堆叠起来，形成更深层的网络，这些网络统称为卷积神经网络（Convolutional Neural Network，CNN）。

卷积神经网络

由图给出了一个卷积神经网络的示意图，用于对输入的句子分类。其中，输入为“我 喜欢 自然 语言 处理 。”6个词，首先将每个词映射为一个词向量，此处假设每个词向量的维度为5（图中输入层的每列表示一个词向量，每个方框表示向量的一个元素）。然后，分别使用4个卷积核对输入进行局部特征提取，其中**前两个卷积核的宽度（N-gram中N的大小）为4**（黄色和蓝色），后**两个卷积核的宽度为3**（绿色和红色），卷积操作每次滑动一个词，则每个卷积核的输出长度为$\boldsymbol{L-N+1}$，其中L为单词的个数，N为卷积核的宽度，简单计算可以得到前两组卷积核的输出长度为3，后两组卷积核的输出长度为4。接下来进行**最大池化**操作，将不同卷积核的输出分别聚合为1个输出，并拼接成一个特征向量，最终经过全连接层分类。



卷积神经网络示意图

卷积神经网络

上述这种沿单一方向滑动的卷积操作又叫作一维卷积，适用于自然语言等序列数据。而对于图像等数据，由于卷积核不但需要横向滑动，还需要纵向滑动，此类卷积叫作二维卷积，类似的还有三维卷积。

与多层感知器模型类似，卷积神经网络中的信息也是从输入层经过隐含层，然后传递给输出层，按照一个方向流动，因此它们都被称为前馈神经网络（Feed-Forward Network，FFN）

模型实现

Pytorch的torch.nn包中使用Conv1d、Conv2d或Conv3d类实现卷积层，它们分别表示一维卷积、二维卷积和三维卷积。此处仅介绍自然语言处理中常用的一维卷积（Conv1d），其构造函数至少需要提供三个参数：

- in_channels: 输入通道的个数，在输入层对应词向量的维度；
- out_channels: 输出通道的个数，对应卷积核的个数；
- Kernel_size: 每个卷积核的宽度。

当调用该Conv1d对象时，输入数据形状为（batch, in_channels, seq_len）,输出数据形状为（batch, out_channels, seq_len），其中在输入数据和输出数据中，seq_len分别表述输入的序列长度和输出的序列长度。

模型实现

```python
import torch
from torch import nn

# 定义一个一维卷积，输入通道大小为5（词向量维度），输出通道为2（卷积核个数），卷积核宽度为4
conv1 = nn.Conv1d(in_channels=5, out_channels=2, kernel_size=4)
# 再定义一个一维卷积，输入通道大小为5（词向量维度），输出通道为2（卷积核个数），卷积核宽度为3
conv2 = nn.Conv1d(in_channels=5, out_channels=2, kernel_size=3)
# 输入数据批次大小为2，即有两个序列，每个序列的长度为6，每个输入的维度为5
inputs = torch.rand(2,5,6)
inputs
conv1
conv2
```

```
[11]: tensor([[[0.4434, 0.5814, 0.7887, 0.6474, 0.5172, 0.1627],
         [0.1911, 0.4894, 0.5862, 0.5612, 0.2112, 0.0339],
         [0.4320, 0.5440, 0.3454, 0.2841, 0.5551, 0.8298],
         [0.9306, 0.6849, 0.7773, 0.7610, 0.4076, 0.6229],
         [0.0349, 0.6368, 0.4980, 0.0863, 0.9436, 0.5144]],

        [[0.8967, 0.5074, 0.0858, 0.9775, 0.8771, 0.2851],
         [0.2556, 0.9577, 0.4026, 0.4392, 0.4552, 0.6845],
         [0.4362, 0.8828, 0.8006, 0.8827, 0.1735, 0.6876],
         [0.5075, 0.9415, 0.6443, 0.9013, 0.6784, 0.5593],
         [0.2296, 0.1762, 0.0107, 0.4427, 0.8551, 0.9488]]])
[11]: Conv1d(5, 2, kernel_size=(4,), stride=(1,))
[11]: Conv1d(5, 2, kernel_size=(3,), stride=(1,))
```

```python
outputs1 = conv1(inputs)
outputs1 # torch.Size([2, 2, 3]) 两个序列，每个序列长度为3，大小为2
```

```
tensor([[[ 0.0660,  0.1636, -0.0205],
         [ 0.1396,  0.0247,  0.2542]],

        [[-0.1227,  0.0109, -0.0070],
         [ 0.4781,  0.3485,  0.5947]]], grad_fn=<ConvolutionBackward0>)
```

```python
outputs2 = conv2(inputs)
outputs2  # torch.Size([2, 2, 4]) 两个序列，每个序列长度为4，大小为2
```

```
tensor([[[ 0.2446, -0.0356,  0.2326,  0.1224],
         [-0.1560, -0.3676, -0.4097,  0.0371]],

        [[-0.0463,  0.1925,  0.3130,  0.1299],
         [-0.3818, -0.2284, -0.2250, -0.2976]]],
       grad_fn=<ConvolutionBackward0>)
```

模型实现

接下来需要调用torch.nn包中定义的池化层类，主要有最大池化、平均池化等。与卷积层类似，各种池化方法也分为一维、二维和三维三种。例如MaxPool1d是一维最大池化，其构造函数至少需要提供一个参数——kernel_size，即池化层核的大小，也就是对多大范围内的输入进行聚合。如果对整个输入序列进行池化，则其大小应为卷积层输出的序列长度。

```python
from torch.nn import MaxPool1d

# 第一个池化层核的大小为3，即卷积层的输出序列长度
pool1 = MaxPool1d(3)
# 执行一维最大池化操作，即取每行输入的最大值
outputs_pool1 = pool1(outputs1)
outputs_pool1
```

```
[20]: tensor([[[0.1636],
               [0.2542]],

              [[0.0109],
               [0.5947]]], grad_fn=<SqueezeBackward1>)
```

```python
# 第二个池化层核的大小为4
pool2 = MaxPool1d(4)
outputs_pool2 = pool2(outputs2)
outputs_pool2
```

```
[21]: tensor([[[ 0.2446],
               [ 0.0371]],

              [[ 0.3130],
               [-0.2250]]], grad_fn=<SqueezeBackward1>)
```

模型实现

除了使用池化层对象实现池化，Pytorch还在torch.nn.functional中实现了池化函数，如max_pool1d等，即无须定义一个池化层对象，就可以直接调用池化功能。这两种实现方式基本一致，一个显著区别在于使用池化函数实现无须事先指定池化层核的大小，只要在调用时提供即可。当处理不定长的序列长度时，此种实现方式更加适合，具体实例如下。

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

# outputs1的最后一维恰好为其序列的长度
outputs_pool1 = F.max_pool1d(outputs1, kernel_size=outputs1.shape[2])
outputs_pool1
```

```
[15]: tensor([[[0.1636],
               [0.2542]],

              [[0.0109],
               [0.5947]]], grad_fn=<SqueezeBackward1>)
```

```python
outputs_pool2 = F.max_pool1d(outputs2, kernel_size=outputs1.shape[2])
outputs_pool2
```

```
[22]: tensor([[[ 0.2446],
               [-0.1560]],

              [[ 0.3130],
               [-0.2250]]], grad_fn=<SqueezeBackward1>)
```

```python
outputs_pool1.shape
outputs_pool2.shape
```

```
[23]: torch.Size([2, 2, 1])
[23]: torch.Size([2, 2, 1])
```

模型实现

由于outputs_pool1和outputs_pool2是两个独立的张量，为了进行下一步操作，还需要调用torch.cat函数将它们拼接起来。在此之前，还需要调用squeeze函数将最后一个为1的维度删除，即将2行1列的矩阵变成一个向量。

```
# squeeze 降维
outputs_pool_squeeze1 = outputs_pool1.squeeze(dim=2)
outputs_pool_squeeze2 = outputs_pool2.squeeze(dim=2)

outputs_pool_squeeze1
outputs_pool_squeeze2
```

```
[24]: tensor([[0.1636, 0.2542],
        [0.0109, 0.5947]], grad_fn=<SqueezeBackward1>)
[24]: tensor([[ 0.2446, -0.1560],
        [ 0.3130, -0.2250]], grad_fn=<SqueezeBackward1>)
```

```
outputs_pool = torch.cat([outputs_pool_squeeze1, outputs_pool_squeeze2], dim=1)
outputs_pool
```

```
[25]: tensor([[ 0.1636,  0.2542,  0.2446, -0.1560],
        [ 0.0109,  0.5947,  0.3130, -0.2250]], grad_fn=<CatBackward0>)
```

模型实现

池化后，再连接一个全连接层，实现分类功能。

```python
# 全连接层，输入维度为4，即池化层输出的维度
linear = nn.Linear(in_features=4, out_features=2)
outputs_linear = linear(outputs_pool)

outputs_linear
```

```
[26]: tensor([[-0.4208,  0.2721],
              [-0.5234,  0.3199]], grad_fn=<AddmmBackward0>)
```