第十章深度学习

深度学习 (deep learning) 的基石是**神经网络 (neural network)**。神经网络在上世纪 80 年代被提出，中间由于 SVMs、Boosting 和随机森林等方法而沉寂、2010 年之后深度学习带来了新的技术，得益于现代更加庞大的数据集，深度学习成为了当下最活跃的研究对象

本章讨论用于图像领域的卷积神经网络 (CNNs)、用于时间或其他序列的循环神经网络 (RNNs) 等基础知识，并使用 `Python torch` 包来实现这些模型

10.1 单层神经网络

神经网络接受一个包含 p 个变量的输入向量 $X = (X_1, X_2, \dots, X_p)$ 并构造一个非线性函数 $f(X)$ 来预测响应变量 Y

神经网络与之前的非线性模型的区别在于它的结构，一个**前馈神经网络 (feed-forward neural network)** 的结构如下：

找不到“../../../_resources/044d520b31ff450a8068b8b467e48b12.png”。

用神经网络的术语描述，4 个特征 $X_1, \dots, X_4$ 组成了**输入层 (input layer)**，输入层被连接到 K 个**隐藏单元 (hidden units)**。神经网络的数学描述如下：

$$
\begin{aligned}
f(X) &= \beta_0 + \sum_{k=1}^{K} \beta_k h_k(X) \\
&= \beta_0 + \sum_{k=1}^{K} \beta_k g(\omega_{k0} + \sum_{j=1}^{p} \omega_{kj} X_j)
\end{aligned}
$$

其中 β_0 是偏置项，β_k 是输出层的权重，$h_k(X)$ 是第 k 个隐藏单元的输出。g 表示**激活函数 (activations)**，ω_{k0} 是第 k 个隐藏单元的偏置项，ω_{kj} 是第 k 个隐藏单元与输入 X_j 的权重

我们这样理解这个单层神经网络：

1. 首先，对于 K 个激活函数 A_k，在隐藏层中，我们计算每个特征的输入：

   $$
   A_k = h_k(X) = g(\omega_{k0} + \sum_{j=1}^{p} \omega_{kj} X_j)
   $$

   其中的每个 $g(z)$ 都是非线性的激活函数，它们被事先设置好了。
2. 当我们得到 A_k，作为之前特征的一个转换时 (就像第七章的样条)，然后将 K 个激活值输入到输出层，输出：

   $$
   f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k A_k
   $$

这是一个线性回归模型，拥有 K 个输入。所有的参数 $\beta_0, \ldots, \beta_K$ 和 $\omega_1 0, \ldots, \omega_{Kp}$ 都需要用训练数据估计

在早期的神经网络中，**sigmoid 激活函数 (sigmoid activation function)** 比较常用：

$$g(z) = \frac{e^z}{1+e^z} = \frac{1}{1+e^{-z}}$$

这实际上同逻辑回归中使用的函数相同，它将一个线性函数转换到 0 和 1 之间的一个概率。现代神经网络常用的激活函数是**修正线性单元 (rectified linear unit, ReLU)**，形式如下：

$$g(z) = (z)_+ = \begin{cases} 0, & \text{if } z < 0 \\ z, & \text{otherwise} \end{cases} = \max(0, z)$$

尽管 ReLU 激活函数有一半的值是 0，但是引入偏置项 ω_{k0} 可以改变这个阈值

神经网络得名于大脑的神经元，激活函数 A_k 接近 1 时，认为这个神经元被激发，就类似大脑中的活跃神经元；当激活值接近 0 时，模拟大脑中不活跃的神经元

> 因此，Sigmoid 激活函数适合模拟神经元，因为它将输入值映射到 $[0, 1]$ 区间上

非线性激活函数 $g(\cdot)$ 非常重要，没有它 $f(X)$ 就会退化为一个简单的关于 X_i 的线性模型。非线性激活函数使我们能够捕捉这些特征之间复杂的相互作用和非线性关系

拟合神经网络需要估计未知参数，他们通常用最小方差损失估计：

$$\text{minimize} \sum_{i=1}^{n} (y_i - f(x_i))^2$$

10.2 多层神经网络

现代神经网络通常具有一个以上的隐含层，每个层大小适中。这里理论上的大量单元的单隐藏层要好

本节使用著名的手写数字数据集 `MNIST` 来研究多层神经网络。这个数据集的特点是拥有 6 w 张训练照片和 1 w 张测试照片，每张照片有 $p = 28 \times 28 = 784$ 个像素，每个像素是一个取值在 $[0, 255]$ 的灰度，表示该像素的颜色有多深，响应变量 $Y = (Y_0, \ldots, Y_9)$ 是是个哑变量，使用了独热编码

下面是一个两层神经网络，用于执行数字分类任务：

找不到“../../../_resources/80ce2d83f55a4ac5d9b0f85a8278f23f.png”。

> 将截距 (机器学习中称为**偏差 (biases)**) 计算在内，此神经网络拥有 235,146 个参数 (机器学习中称为**权重 (weights)**)

它的特点是：

1. 有两层，分别有 $L_1 = 256$ 个 $L_2 = 128$ 个单元

2. 拥有十个定性输出变量，这些变量之间高度相关。在**多任务学习 (multi-task learining)** 中，单个神经网络可同时预测不同的想要，这些响应都会影响隐藏层的形成
3. 为多任务学习制定了相应的损失函数

第一个隐藏层的激活函数如下：

$$A_k^{(1)} = h_k^{(1)}(X) \qquad = g(\omega_{k0}^{(1)} + \sum_{j=1}^{p} \omega_{kj}^{(1)} X_j)$$

其中 $k = 1, 2, \ldots, K_1$。第二个隐藏层接受输入 $A_k^{(1)}$，激活函数如下：

$$A_l^{(2)} = h_l^{(2)}(X) \qquad = g(\omega_{l0}^{(2)} + \sum_{k=1}^{K_1} \omega_{lk}^{(2)} A_k^{(1)})$$

其中 $l = 1, \ldots, K_2$

> 第二层隐藏层也是 X 的函数，这表明多层神经网络能够得到相当复杂的 X 交互项的输出

图示记号 $\mathbf{W}_1$ 表示从输入层到第一个隐藏层 L_1 的整个权重矩阵。这个矩阵将会包含 $785 \times 256 = 200960$ 个元素，785 而不是 784 是因为计入了截距或称偏差项 (机器学习社区)

> 计算方式 $(p + 1) \cdot L_1$

在输出层，我们计算：

$$\begin{aligned} X_m &= \beta_{m0} + \sum_{l=1}^{K_2} \beta_{ml} h_l^{(2)}(X) \\ &= \beta_{m0} + \sum_{l=1}^{K_2} \beta_{ml} A_l^{(2)} \end{aligned}$$

这是一个线性模型。其中 $m = 0, 1, \ldots, 9$，这个权重矩阵则包括 $129 \times 10 = 1290$ 个参数

对于定量的输出，为了保证我们分类的输出情况类似于概率形式，我们还必须对输出层进行 softmax 激活：

$$f_m(X) = Pr(Y = m|X) = \frac{e^{Z_m}}{\sum_{l=0}^{9} e^{Z_l}}$$

为了训练这个网络，我们估计能使最小化：

$$-\sum_{i=1}^{n} \sum_{m=0}^{9} y_{im} \log(f_m(x_i))$$

负数多项对数似然值的参数，这也被称为**交叉熵 (corss-entropy)**，这是二元分类逻辑回归准测的推广。其中，y_{im} 是一个指示函数

> 熵的计算公式是：$H(p) = -\sum_x p(x) \log p(x)$，表示一个随机变量的不确定性；交叉熵被定义为衡量用分布 q 表示真实分布 p 所需的平均编码长度：$H(p, q) = -\sum_x p(x) \log q(x)$

神经网络的参数如此之多，以至于 6 w 个训练值不足以估计所有的参数。因此，为了避免过拟合，我们使用了两种形式的正则化：

- 岭正则化，类似于岭回归
- **Dropout**正则化

10.3 卷积神经网络

本章我们用 `CIFAR 100` 数据集研究**卷积神经网络 (Convolutional Neural Networks, CNNs)**。这个数据集包含分为 100 个细分子类，20 个超类的动物图像，每个图像包含 32×32 个像素，每个像素由 3 个 8 bits 数组成，分别表示红绿蓝，这些值存储在一个叫做**特征图 (feature map)** 的三维数组中，前两维表示空间信息，即该像素的高度和宽度，第三维表示**颜色通道 (channel)** ，这是一个三维数组，分别包含 RGB 的取值 $[0,255]$。数据包含 5 w 训练样本和 1 w 测试样本

CNNs 被发明用于这种任务，它通过识别图像任何位置的特征或者模式来区分每个对象的分类。它首先识别输入图像的低级特征，如小的边缘、颜色块等，然后用这些低级特征组合为高层次特征，如动物的不同部位，根据这些高层特征的存在和缺失来提高输出正确类的概率

CNNs 有两个特殊的隐藏层，分别是**卷积 (convolution)** 层和**池化 (pooling)** 层。卷积层在图像中搜索小型局部模式，不同层次的卷积层识别的模式不同；池化层则对卷积层的特征图进行**降采样 (downsample)** 以获取最突出的信息，提高稳健性

> 池化的常见方法包括最大池化 (保留窗口内的最大值) 和平均池化 (保留窗口内的平均值)

10.3.1 卷积层

卷积层是由许多**卷积滤波器 (convolutional filter)** 组成的。卷积滤波器依赖于一个非常简单的操作，即卷积，它可以认为是重复相乘矩阵元素，然后将结果相加

假设一个 4×3 的图像：

$$\text{Original Image} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{bmatrix}$$

我们考虑一个 2×2 的卷积核 (也就是卷积滤波器)：

$$\text{Convolution Filter} = \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix}$$

则我们用滤波器卷积得到的结果是：

$$\text{Convolved Image} = \begin{bmatrix} a\alpha + b\beta + d\gamma + e\delta & b\alpha + c\beta + e\gamma + f\delta \\ d\alpha + e\beta + g\gamma + h\delta & e\alpha + f\beta + h\gamma + i\delta \\ g\alpha + h\beta + j\gamma + k\delta & h\alpha + i\beta + k\gamma + l\delta \end{bmatrix}$$

卷积核用于模式匹配和特征提取。若一个子矩阵的结构和卷积核很相似，则计算输出的值就较大；不同的卷积核，可以分别识别图像的不同特征。卷积核的局部作用和大小的灵活变化使它在识别图像上非常好用

> 卷积特征图的值大小表示相似性，是因为卷积核的卷积过程相当于两个矩阵的点乘，如 $(a,b,c,d)^T \cdot (\alpha,\beta,\gamma,\delta)^T$。如果这个值较大，说明向量的方向约接近，即越相似

卷积特征图的输出高度等于输入高度减去卷积核高度加一，输出宽度等于输入宽度减去卷积核宽度加一

以竖向卷积核和横向卷积核为例，对于一副图像，它们分别生成凸显竖线特征和横线特征的特征图。在传统方法中，这是通过预定义的滤波器，如 Sobel 提取边缘，Gabor 提取纹理实现的，需要人工组合多种滤波器。CNNs 采用不同的权重来使用滤波器，能够针对特定分类任务进行优化

以下是一些更详细的介绍：

- 对于一张 RGB 图，其是一个 $32\times32\times3$ 的张量，我们同样设计一个 $3\times3\times3$ 的卷积核，来对图像特征进行卷积，这样能得到 3 副 30×30 的 R、G、B 特征图，接着对这些特征图对应求和，就能综合 RGB 三种颜色特征，得到最终的一张特征图

  > 此时，颜色作为特征已经全部使用，将不会传入到后续的卷积层中

- 在 CNN 的第一层，通常拥有 K 个卷积核，每个卷积核都会与输入图像进行卷积，共产生 K 个 2 D 图。这 K 个 2 D 图堆叠在一起，就形成了一个新的 3 D 特征图，然后继续应用卷积核

  > 卷积产生的 2 D 图就好像传统神经网络中隐藏层的激活值，不过区别在于这里的激活值是一个矩阵，保留了空间信息

- 为了引入非线性操作 (相对于卷积的线性操作)，卷积后通常施加 $ReLU$ 激活函数以去掉负数值，增加稀疏性便于计算。这样看来，$ReLU$ 就像一个特征过滤器，对于高响应的值保留，低响应的则置零。因此又将它视为一个**检测层 (detector layer)**

10.3.2 池化层

池化层在 CNN 中执行浓缩的功能。具体来说，**池化 (pooling)** 保留原有特征值的重要特征，降低后续运算量，例如：

$$\text{Max pool}\begin{bmatrix}1&2&5&3\\3&0&1&2\\2&1&3&4\\1&1&2&0\end{bmatrix}\rightarrow\begin{bmatrix}3&5\\2&4\end{bmatrix}$$

10.3.3 卷积神经网络架构

找不到“../../../_resources/859c959d0c796603815318fbe547d271.png”。

一个典型的 CNN 包含多个卷积层和池化层。其中，每个卷积层包含多个卷积核，生成多个 2D 图像；池化层提取这些图像的重要特征 (降低图像的宽度高度维数) 后，再次进行卷积，为了补偿池化层降维的部分，这里的卷积核通常比之前更多。经过多层的特征提取后，当空间维度足够低时，将所有的像素扁平化，视为单独的特征，接入 MLP 进行非线性操作，最后输出结果

> 示例图中卷积后图像空间维度不变，是因为使用了**填充 (padding)** 方法
> 有时候在池化层前接入多个卷积层，可以在不降低空间分辨率的前提下提取更复杂的特征。这等效于使用更大感受视野的卷积核

CNNs 的参数如此多，为了避免过拟合，我们需要调优的超参数包括：

- 卷积核个数，大小，层次
- 池化的方式，步长等
- 全连接层大小、Dropout 方法、L_2 正则化

10.3.4 数据增强

数据增强 (data augmentation) 是一种增大数据集的技术。CNNs 一来大规模数据，但真实的标注数据有限，为了减少过拟合，我们可用数据增强的方法扩大数据集。常见的有：

1. 几何变换。如旋转、平移、缩放等
2. 颜色变换。如亮度、对比度和噪声等
3. 伪造数据。如随机遮挡图像、混合图片和裁剪拼接等

数据增强可以看做是一种正则化的形式

10.3.5 预训练分类器

CNNs 的核心是学习卷积滤波器，这一学习过程通过反向传播学习实现。当 CNN 在大规模数据集上训练后，他学习到的卷积滤波器可以快速泛化到其他视觉任务上。这是因为低级和中级特征在不同任务中是通用的，只有最后的分类层要根据任务具体调整

迁移学习是指在大数据集上训练好的 CNN 作为特征提取器，然后在小数据集上微调以适应小数据任务的过程

权重冻结 (Weight Freezing) 的意思是不更新 CNN 预训练好的前几层权重，只训练最后几层 (通常是全连接层)。这可以减少训练量，并避免过拟合

10.4 文档分类

文档分类时一种在工业和生活中常见的场景，例如对医学期刊评级、对新闻稿、邮件进行分类等。下面使用 `IMDb` 数据集演示，这是一个关于电影评价和最终得分的数据集，一部电影的评价可能是积极的或者消极的

每个评论的长度不同，用语不同 (俚语甚至非词语)，我们需要寻找方法来对文档进行**特征提取 (featurize)**

最常见和简单的方案是使用**词袋 (bag-of-words)** 模型。我们用一个单词的出现与否对文档进行评分；若一个文档包含 M 个单词，那意味着他们都是一个 M 维度的向量，其中 1 表示单词存在，0 表示不存在。我们通过限定最常用的 1 w 个词来限定向量维度，其中不在这些词的字典中的词将会被标记为如 `<UNK>` 的样式

通过这种方法构造的向量特征很多，但是作为训练集的矩阵非零元素也很多，这被称为**稀疏矩阵 (sparse matrix format)**，可以用特别的方法存储以节省空间

实验中使用 lasso 和两个隐藏层的二分类神经网络来进行预测，注意到二分类神经网络可视为一个非线性逻辑回归模型：

$$\begin{aligned}\log\left(\frac{\Pr(Y=1|X)}{\Pr(Y=0|X)}\right) &= Z_1 - Z_0 \\ &= (\beta_{10}-\beta_{00}) + \sum_{l=1}^{K_2}(\beta_{1l}-\beta_{0l})A_l^{(2)}\end{aligned}$$

> Softmax 函数具有冗余性，因为 K 分类问题只需要 $K-1$ 组系数，最后一个可以通过概率相减得到

词袋模型只考虑的文章中是否出现单词，但是没考虑上下文，下面是另外两种常用的改进方案：

1. **bag-of-n-grams**模型。例如，我们可以每次将两对临近单词进行组合，联系上下文
2. 将文档视为一个序列，考虑单词出现的同时考虑上下文单词的位置

10.5 循环神经网络

许多数据实际上是序列化的，构建模型时需要特殊处理，例如：

- 文档，如书籍和电影评论。它们单词的顺序和相对位置捕捉了叙述主题和语气等特征
- 温度、降雨量等时间序列
- 金融时间序列
- 语音、录音等。文本转录和语言翻译需要考虑序列化
- 手写体

在**循环神经网络 (recurrent neural network, RNN)** 中，输入矩阵 X 就是一个序列。RNN 被设计用来处理这类有序数据，这就像 CNN 被设计得适用于图像数据一样。这些输入数据 $X=(X_1,X_2,\dots,X_L)$，元素之间的排列顺序和近邻程度可以衡量它们之间的关系。RNN 的输出可以是序列 (如文本生成)，也可以是标量 (分类)

找不到“../../../_resources/0c74b25faf8ae01d5053c69d6d1e98bf.png”。

对于数据序列 $X = \{X_1, X_2, \ldots, X_L\}$，其中每个 X_l 都是一个向量，隐藏层神经元序列 $\{A_l\}_1^L = \{A_1, A_2, \ldots, A_L\}$。用文本举例，则每个 X_l 都是一个单词的独热编码 (例如 $X_l = (0, 0, 1, 0, 0)$)。RNN 每次接受一个数据序列中的向量 X_l，并根据之前序列得到的激活值 A_{l-1} 输出当前序列的激活值 A_l，同时输出一个预测值 O_l。对于最后一次输出 O_L，它是和我们预测的目标最接近的

具体来说，我们考虑输入序列中的一个向量 X_l，它拥有 p 个元素 $X_l^T = (X_{l1}, X_{l2}, \ldots, X_{lp})$，隐藏层由 K 个神经元组成：$A_l^T = (A_{l1}, A_{l2}, \ldots, A_{lp})$。我们可以用矩阵表示权重：

- 输入层的权重矩阵 $\mathbf{W}$，维度 $K \times (p+1)$
- 隐藏层到隐藏层的权重矩阵 $\mathbf{U}$ 共有 $K \times K$ 维度，其中每个元素 u_{ks} 表示从隐藏层中的神经元 k 到另一个隐藏层中的神经元 s 的权重
- 输出层的权重向量 $\mathbf{B}$，有 $K+1$ 个权重 β_k 到输出结果上

因此，隐藏层的计算公式 (激活值) 为：

$$A_{lk} = g\left(\omega_{k0} + \sum_{j=1}^{p} \omega_{kj} X_{lj} + \sum_{s=1}^{K} u_{ks} A_{l-1,s}\right)$$

每次的输出 O_l 可以计算为：

$$O_l = \beta_0 + \sum_{k=1}^{K} \beta_k A_{lk}$$

对于定量变量等可以接入 Sigmoid 激活函数

上面的计算式中，$g(\cdot)$ 是一个激活函数，类似于 ReLU。注意权重矩阵 $\mathbf{W}, \mathbf{U}, \mathbf{B}$ 不是 l 的函数，它们在每次接受序列数据时保持一致。这是 RNNs 的特性，即**权重共享 (weight sharing)**

对于回归问题，每对观测值 (X, Y) 的损失函数为：

$$(Y - O_L)^2$$

其中 $O_L = \beta_0 + \sum_{k=1}^{K} \beta_k A_{Lk}$。注意这里只用到了最后一次的输出 O_L，而忽略了前面的输出。因为共享权重，每次的输入数据简介地对权重矩阵产生贡献。对于 n 对观测值 (x_i, y_i)，我们的参数需要最小化：

$$\sum_{i=1}^{n} (y_i - o_{iL})^2 = \sum_{i=1}^{n} \left(y_i - \left(\beta_0 + \sum_{k=1}^{K} \beta_k g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{iLj} + \sum_{s=1}^{K} u_{ks} a_{i,L-1,s}\right)\right)\right)^2$$

为什么保留每次的输出 O_l，尽管我们只用到了 O_L？这是因为它们是必要计算的，而且有时候我们希望得到的结果也是一个变化的序列 $\{O_1, O_2, \ldots, O_L\}$

10.5.1 用于文档分类的序列模型

使用独热编码来编码各类单词存在维度灾难的问题，因为每个单词都会表示为一个非常稀疏的向量。一种解决方案是使用**嵌入 (embedding)** 。词嵌入是指用一个低维度的向量 (维度为

$m << n$) 来表示之前的独热编码，这需要用到一个嵌入矩阵 $E_{m\times n}$，其中每列表示第 i 个单词转换后的嵌入向量 e_i，他们之间的关系满足：

$$\mathbf{e}_i = E\mathbf{v}_i$$

其中 $\mathbf{v}_i$ 表示单词 i 的独热编码向量

为了获取矩阵 E，我们首先需要有大规模标注的语料集，然后将 E 作为神经网络优化的一部分。在这种情况下，E 就成为了**嵌入层 (embedding layer)**。或者我们可以在嵌入层中加入一个预训练的矩阵，也即是**权重冻结 (weight freezing)** 的一个过程。两个广泛使用的预训练矩阵是 `word 2 vec` 和 `GloVe`，它们是主成分分析 PCA 的变体，要求同义词需要出现在映射空间的附近

下一步，我们将每个文档限制到最后的 L 个单词，然后将 $< L$ 的文档前置补 0，这样每个文档都是由 L 个向量 $X = \{X_1, X_2, \ldots, X_L\}$ 组成的序列了，每个 X_l 拥有 m 个分量

我们用基础 RNN 拟合这个模型，它的参数如下：

- 常规的权重矩阵 W，拥有 $K\times(m+1)$ 个参数
- 矩阵 U 拥有 $K\times K$ 个参数
- 向量 B 拥有 $2(K+1)$ 个参数 (2 分类问题下)
- 如果要训练嵌入矩阵 E，它有 $m\times D$ 的参数，D 表示词典单词数，是最大的一个矩阵

实验表明，简单 RNN 的训练正确率为 76%。一种改进的 RNN 模型是**长短期记忆模型 (long term and short term memory， LSTM)**，一定程度上克服了简单 RNN 的梯度下降和爆炸问题。LSTM 模型在此数据集上的性能提高到了 87%，与词袋模型 88% 相当

对 LSTM 进行调参是一个很花时间的工作。RNNs 是一系列模型的基础，在此基础上，一个领先的 RNNs 配置在 `IMDb` 数据集上报告了 95% 以上的准确率。本书未提及细节

10.5.2 时间序列预测

下面用一份纽约股票交易所的交易数据来演示时间序列预测。它的三份时间序列如下：

- 对数交易量。表示当天交易流通股票相对于过去 100 天交易量的移动平均值的对数，计算公式为 $\log(\frac{\text{当日交易量}}{\text{100天平均交易量}})$
- Dow Jones 指数汇报。表示连续交易日之间 Dow Jones 工业指数对数值的差异，计算公式为 $\log(\text{today}) - \log(\text{yesterday})$
- 对数波动率，表示每日价格变动绝对值的对数

日期 t 的观测值 (v_t, r_t, z_t) 分别代表对数交易量，道琼指数和对数波动率，共有 $T = 6051$ 个这样的观测对。在这种情况下，观测值们倾向于拥有相似的表现，这与其他训练数据之间相互独立的假设不同。我们假设观测对 $(v_t, v_{t-l}$，其中 l 表示 l 天的**滞后 (lag)**，获得连续的这样的观测对，我们可以计算它们之间的自相关性

> 时间序列的自相关性使我们能够直接利用它的历史数据来预测未来的数据

我们希望通过 v_t 的过去值 $v_{t-1}, v_{t-2}, \dots$ 来预测 v_t，同时也要考虑 r_{t-1}, r_{t-2} 和 $z_{t-1}, z_{t-2}, \dots$ 等历史数据。这与文本预测不同的是，我们只有一条时间序列 (不同于文本的 n 条评论时间序列)，因此我们考虑利用滞后 L 来划分历史数据：

$$X_1 = \begin{pmatrix} v_{t-L} \\ r_{t-L} \\ z_{t-L} \end{pmatrix}, X_2 = \begin{pmatrix} v_{t-L+1} \\ r_{t-L+1} \\ z_{t-L+1} \end{pmatrix}, \dots, X_L = \begin{pmatrix} v_{t-1} \\ r_{t-1} \\ z_{t-1} \end{pmatrix}$$

我们的预测目标 $Y = v_t$。对于时间 t 的变化，我们的数据可以划分出相当多的训练数据对。注意 L 在这里是一个关键参数，可能需要通过调优获得。用 RNN 拟合的模型 $R^2 = 0.42$，是一个不错的效果

刚才的 RNN 模型与传统的**自回归模型 (autoregression, AR)** 有许多共同指出，我们将历史数据改造为响应值向量和设计矩阵：

$$y = \begin{bmatrix} v_{L+1} \\ v_{L+2} \\ \vdots \\ v_T \end{bmatrix}, M = \begin{bmatrix} 1 & v_L & v_{L-1} & \cdots & v_1 \\ 1 & v_{L+1} & v_L & \cdots & v_2 \\ 1 & v_{L+2} & v_{L+1} & \cdots & v_3 \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & v_{T-1} & v_{T-2} & \cdots & v_{T-L} \end{bmatrix}$$

这与用滞后值来预测当前值的思路一致，我们用 y 和 M 拟合回归模型：

$$\hat{v}_t = \hat{\beta}_0 + \hat{\beta}_1 v_{t-1} + \beta_2 v_{t-2} + \cdots + \hat{\beta}_L v_{t-L}$$

这被称为顺序 L 自回归模型，缩写为 AR (L)，将特征改为包括另外两个时间序列的向量，我们就能得到 AR 版本的预测模型，其 $R^2 = 0.41$，仅比 RNN 少 0.01

与 RNN 类似，AR 也接受 $L = 5, p = 3$ 的数据，但是它将 L 个元素同等对待，作为一个 $L \times p$ 的预测变量的向量，这个过程在神经网络中称为**展平 (flatening)**。RNN 理论上能够记住前面每次预测的信息，并作用到后续的预测结果上，但实际上受到梯度的限制

现代常用的手段是扩展 AR 模型，如将滞后变量作为输入向量，输入到一个普通神经网络中，增强模型灵活性；或者增加模型的特征，如星期，可用独热编码表示；也可以用 RNN 的改进版本 LSTM 来预测

10.5.3 RNNs 的总结

上面的讨论仅仅涉及了 RNNs 的很小部分，RNN 的扩展性非常强

一维卷积神经网络可以用于处理序列，例如，我们将单词表示为嵌入空间的向量组，并用一维 CNN 滑动处理，也能识别短语和时间序列的趋势

在 RNN 中添加多个隐藏层，可以增强模型的表达能力和学习能力，例如第一个隐藏层的输出序列 A_l 再输入到下一个隐藏层中；还可以扩展为**双向 RNN (bidirectional RNNs)**，从两个方向扫描序列，缓解梯度问题

在语言翻译中，目标也是一个词序列，但是它使用的语言与输入序列的语言不同。在**序列到序列 (Seq 2 Seq)** 学习中，隐藏单元被认为能够捕捉句子的语义含义。语言建模和翻译中的一些重大突破源于这类 RNN（循环神经网络）相对较近的改进

> 这种模型能解决 $n \to m$ 的问题

10.6 何时使用深度学习

深度学习在现代的多种任务中取得了相当好的性能，真正成为了当代的一项革新技术。然而，它本质上是一个黑箱模型，解释性相比我们之前学习的方法差了很多

尽管深度学习在恰当的参数下表现得十分优异，但是这往往十分耗时。**Occam's razor**指出，当模型的效果相近时，选择最简单的那个

通常来说，当训练集的样本量非常大，且模型的可解释性不重要时，我们选择深度学习模型

10.7 神经网络的拟合

我们从简单神经网络开始。对于 10.1 节的简单神经网络，我们的目的是最小化参数 $\beta = (\beta_0, \beta_1, \dots, \beta_K)$ 和 $\omega_{k0}, \omega_{k1}, \dots, \omega_{kp}$ 在 n 对观测值 (x_i, y_i) 下的损失函数：

$$\underset{\{\omega_k\}_1^K}{\text{minimize}} \frac{1}{2} \sum_{i=1}^{n} (y_i - f(x_i))^2$$

其中：

$$f(x_i) = \beta_0 + \sum_{k=1}^{K} \beta_k g(\omega_{k0} + \sum_{j=1}^{p} \omega_{kj} x_{ij})$$

> 损失函数在参数上是非凸的，因此存在多个解。例如，对于最小值，我们可能得到**局部最小值 (local minimum)** 或者 **全局最小值 (global minimum)**

为了避免最小化损失函数过程中的过拟合，常采用下面两种方法：

- **慢学习 (slow learning)** 缓慢地进行**梯度下降 (gradient descent)** 过程，在探测到过拟合时停止优化
- 正则化在损失函数中添加惩罚项

假设我们的所有参数都放在一个向量 θ 中，则上面的损失函数改写为：

$$R(\theta) = \frac{1}{2} \sum_{i=1}^{n} (y_i - f_\theta(x_i))^2$$

通过上式，我们很容易用梯度下降的方法来获取一个全局 (局部) 最小值作为参数向量的选择：

1. 猜测 θ 的一个子集元素 θ^0，设置时间 $t = 0$
2. 迭代，直到上式下降到合适的一个值

1. 找到一个向量 δ 作为 θ 的一个微小变化值，使得 $\theta^{t+1}=\theta^t+\delta$ 能使优化函数下降，即 $R(\theta^{t+1})<R(\theta^t)$
2. 修改 $t\leftarrow t+1$

一般来说，我们在复杂问题中得到梯度下降的最小值是一个较优局部最小值

10.7.1 反向传播

为了找到最合适的 θ 向量，使得 $R(\theta)$ 的值减小，我们定义目标函数的**梯度 (gradient)** 表示函数的偏导数向量：

$$\nabla R(\theta^m)=\frac{\partial R(\theta)}{\partial\theta}|_{\theta=\theta^m}$$

其中 $\theta=\theta^m$ 表示当猜测完当前的 θ 后，再计算梯度值。梯度表示某个点 $\theta=\theta^m$ 处函数增长最快的方向，然后我们希望减小 $R(\theta)$ 的值，于是反方向移动 θ：

$$\theta^{m+1}\leftarrow\theta^m-\rho\nabla R(\theta^m)$$

对于足够小的**学习率 (learning rate)**，这一步变动将会得到更小的 $R(\theta)$ 值，即 $R(\theta^{m+1})\leq R(\theta^m)$。当梯度向量为 0 时，我们达到了目标最小值

> 计算梯度的过程需要用到求偏导数的**链式法则 (chain rule)**

总的损失函数 $R(\theta)$ 被分解为 $R(\theta)=\sum_{i=1}^n R_i(\theta)=\frac{1}{2}\sum_{i=1}^n(y_i-f_\theta(x_i))^2$，对于每对观测值 (x_i,y_i)，它的损失函数：

$$R_i(\theta)=\frac{1}{2}\left(y_i-\beta_0-\sum_{k=1}^K\beta_k g(\omega_{k0}+\sum_{j=1}^p\omega_{kj}x_{ij})\right)^2$$

为了简化上式，我们用 $z_{ik}=\omega_{k0}+\sum_{j=1}^p\omega_{kj}x_{ij}$ 来代替激活函数 $g(\cdot)$，则关于 β_k 的梯度：

$$\begin{aligned}\frac{\partial R_i(\theta)}{\partial\beta_k}&=\frac{\partial R_i(\theta)}{\partial f_\theta(x_i)}\cdot\frac{\partial f_\theta(x_i)}{\partial\beta_k}\\&=-(y_i-f_\theta(x_i))\cdot g(z_{ik})\end{aligned}$$

其中 $g(z_{ik})$ 表示第 k 个隐藏层神经元的输出，这一项体现了第 k 个神经元对误差的贡献。ω_{kj} 的梯度：

$$\begin{aligned}\frac{\partial R_i(\theta)}{\partial\omega_{kj}}&=\frac{\partial R_i(\theta)}{\partial f_\theta(x_i)}\cdot\frac{\partial f_\theta(x_i)}{\partial g(z_{ik})}\cdot\frac{\partial g(z_{ik})}{\partial z_{ik}}\cdot\frac{\partial z_{ik}}{\partial\omega_{kj}}\\&=-(y_i-f_\theta(x_i))\cdot\beta_k\cdot g'(z_{ik})x_{ij}\end{aligned}$$

其中 β_k 表示输出层权重，调节残差分配到隐藏层神经元 k 的比例，激活函数的导数 $g'(z_{ik})$ 衡量激活函数对输入的敏感度，输入值 x_{ij} 表示输入数据对隐藏层神经元 k 的贡献

反向传播 (backpropagation) 的起点是残差 $(y_i-f(x_i))$，之后通过梯度逐层分解到每层上。通过逐层分解的方式，反向传播计算了所有参数的梯度，使得神经网络能够通过梯度下降优化

10.7.2 正则化和随机梯度下降

计算大量数据的梯度是非常麻烦的。在实践中，常使用**随机梯度下降 (Stochastic Gradient Descent, SGD)** 的方法进行加速。随机梯度下降不计算所有样本以进行反向传播，而是每次迭代只使用一个小样本，也称为**小批量 (minibatch)** 样本进行训练

考虑数字分类使用的双隐藏层神经网络，它的参数数量是训练数据的 4 倍，因此必须通过正则化来避免过拟合。我们引入 L_2 正则化的交叉熵公式如下：

$$R(\theta;\lambda)=-\sum_{i=1}^{n}\sum_{m=0}^{9}y_{im}\log(f_m(x_i))+\lambda\sum_j\theta_j^2$$

这里的 λ 可以通过设置一个较小值或者通过验证集方法找到。对于不同层的权重组，也可以使用不同的 λ 值；也可以引入 lasso 回归使用的 L_1 正则化项

> 文中的实验指出，SGD 对模型的作用效果近似于施加一个二次的 L_2 正则化项

SGD 在训练过程中将整个数据集划分成多个小批量，并对每个小批量计算损失函数和梯度，更新模型参数。当整个训练集被遍历时，我们说完成了一个**epoch**，通常训练多个 epoch 以获得最佳模型

当训练到一定的 epoch 后，使用验证集方法表明模型的验证误差出现了增加，表明潜在的过拟合发生。因此，**早停 (early stopping)** 策略常常被应用于及时停止迭代，防止过拟合

10.7.3 丢弃学习

丢弃 (dropout) 是一种正则化技术，它受到随机森林的启发，每次拟合模型时随机移除一层神经元中的某个部分 Φ，这些神经元不参与计算，不影响梯度更新，因此每次反向传播仅对未被丢弃的神经元进行梯度更新。进行测试时则不进行丢弃，而对所有神经元的输出乘以丢弃概率 p 以保持期望输出一致

丢弃学习可以防止神经元变得过拟合。在实际应用中，“丢弃”的操作是将某些神经元激活值设置为 0 实现的

10.7.4 网络调优

在神经网络的训练过程中，我们有许多需要考虑的参数：

- 隐藏层的数量和每层的神经元数量。现代观点认为，每层隐藏层可以容纳大量的神经元
- 正则化参数。包括丢弃率 Φ，正则化强度 λ 等，它们通常在不同隐藏层中单独设置
- SGD 的细节。包括批量的大小，训练的 epochs 数，可能被应用的数据增强的细节

我们需要对这些超参数进行调优，因为它们的设置直接影响模型性能。网络调优的目的即是寻找最优策略提高模型准确率，减少过拟合和欠拟合并优化计算效率

10.8 差值和双下降

本书讨论的一个核心观点是偏差-方差权衡。这个观点认为，在训练误差接近零 (这一区域被称为**插值 (interpolate)** 区) 时，测试误差将会增大。一般来说，我们期望训练误差呈现下降的趋

势，而测试误差则会形成一个 U 型曲线

然而，事实证明，在某些特定的设置中，插值训练的统计学习方法可以表现得很好，这种现象就是**双下降 (double descent)**。

原书用自然样条进行了实验。结果表明，当自由度与训练数据相同时，拟合曲线震荡程度很大；而当自由度远超过训练数据时，拟合曲线的震荡反而减小，即出现了双下降现象。大量参数的神经网络也会出现双下降现象，这是因为 SGD 等技术倾向于选择一个平滑的插值模型，在这类问题上具有良好的测试集性能

> 自然样条选择过程中，提到了**最小范数解 (minimum norm solution)** 的概念，即在方程个数少于变量的方程组中，选择欧几里得范数 $l = \sqrt{\sum_{i=1}^{n} x_i^2}$ 最小的解

下面是一些值得注意的点：

- 双下降与偏差-方差权衡不矛盾。在书中提到的例子中，双下降区域是因为横轴的显示为样条基函数的数量，这不能完全真实地表示模型的灵活性，因为在插值区域，后续模型的最小自然范数样条比之前更低，表明方差的再次下降
- 正则化方法同样适用。正则化方法不需要插值数据，就能获得优秀的泛化能力，获得更好的测试误差。正则化方法避免了双下降，且双下降不是所有学习方法的普遍特性
- SVM 等方法能够达到零训练误差，且泛化依然较好，是因为它们找到的是最小范数解。最小范数解兼具平滑性和低复杂度，但是零训练误差并不总是最优
- 零训练误差的效果取决于数据的信噪比。对于高信噪比的数据，训练数据噪声很少，做到零误差是可能的；但对于低信噪比的数据，模型非常容易过拟合。避免过拟合的方法包括正则化的早停等

10.9 实验：深度学习

本章使用的包主要来自 `Python torch` 和 `pytorch_ligntning`，它们提供了方便的深度学习框架。关于更多有关 `pytorch` 的教程，参见文档Pytorch 文档

本章的基础引入包括：

```
import numpy as np, pandas as pd
from matplotlib.pyplot import subplots
from sklearn.linear_model import \
    (LinearRegression,
     LogisticRegression,
     Lasso)
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
from sklearn.pipeline import Pipeline
from ISLP import load_data
from ISLP.models import ModelSpec as MS
from sklearn.model_selection import \
    (train_test_split,
     GridSearchCV)
```

关于 `pytorch` 的包必须单独引入

首先是 `pytorch` 的基础库：

```
import torch
from torch import nn
from torch.optim import RMSprop
from torch.utils.data import TensorDataset
```

`pytorch` 拥有很多辅助函数，例如 `torchmetrics` 可以计算不同的矩阵来评估模型效果，`torchinfo` 提供了许多关于模型每层的总结信息， `read_image` 可以用于加载图片：

```
from torchmetrics import (MeanAbsoluteError,
R2Score)
from torchinfo import summary
from torchvision.io import read_image
```

`pytorch_lightning` 是 `pytorch` 的一个更高级别的API，它能更加简化训练模型的过程而不关注底层实现细节：

```
from pytorch_lightning import Trainer
from pytorch_lightning.loggers import CSVLogger
```

为了重现结果，我们使用 `seed_everything()` 函数。同时，我们尽可能让 `pytorch` 使用确定性算法：

```
from pytorch_lightning import seed_everything
```

```
#使用全局种子，便于复现结果
seed_everything(0, workers=True)
torch.use_deterministic_algorithms(True, warn_only=True)
```

我们使用几个 `torchvision` 中包含的数据集和一些预训练模型：

```
from torchvision.datasets import MNIST,CIFAR100
from torchvision.models import (resnet50,
                                ResNet50_Weights)
from torchvision.transforms import (Resize,
                                    Normalize,
                                    CenterCrop,
                                    ToTensor)
```

`ISLP` 开发了一些专门针对深度模型的对象，其中 `SimpleDataModule` 和 `SimpleModule` 是 `pytorch_lightning` 中对象的简单版本，它们不考虑关于GPU并行计算的问题(本章实验不涉及)；此外， `Error Tracker` 在验证或测试阶段处理小批量的目标和预测的集合，允许在整个验证或测试数据集上度量：

```
from ISLP.torch import (SimpleDataModule,
                        SimpleModule,
                        ErrorTracker,
                        rec_num_workders)
```

`ISLP` 还开发了针对 `IMDb` 数据集的辅助函数，以及通过映射整数到特定键的查找函数(关于数据库)：

```
from ISLP.torch.imdb import (load_lookup,
                             load_tensor,
                             load_sparse,
                             load_sequential)
```

最后是一些其他的函数。 `glob` 包中的 `glob` 函数可以查找所有与 `glob()` 通配符字符匹配的文件，可让我们用 `ResNet50` 拟合我们自己的图像； `json` 包可以帮助我们加载 `json` 文件：

```
from glob import glob
import json
```

10.9.1 单层神经网络在Hitters数据上的应用

本节使用 `Hitters` 数据集作为示例：

```
Hitters = load_data('Hitters').dropna()
n = Hitters.shape[0]
```

我们用最小二乘回归和lasso回归来比较单层神经网络的效果，其衡量指标是平均绝对误差：

$$\mathrm{MAE}(y,\hat{y}) = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

定义模型和设计矩阵：

```
model = MS(Hitters.columns.drop('Salary'), intercept=False)
X = model.fit_transform(Hitters).to_numpy()
Y = Hitters['Salary'].to_numpy()
```

这里用 `to_numpy()` 方法将 `pandas` 中的数据框或者列转换为 `numpy` 中的矩阵，这是为了便于使用 `sklearn` 中的接口进行比较，而不是用之前的 `statsmodels` 包

现在，我们将数据划分为测试集和训练集并满足随机划分假设：

```
(X_train,
 X_test,
 Y_train,
 Y_test) = train_test_split(X,
                            Y,
                            test_size=1/3,
                            random_state=1)
```

下面直接拟合线性模型，并评估测试误差：

```
hit_lm = LinearRegression().fit(X_train, Y_train)
Yhat_test = hit_lm.predict(X_test)
np.abs(Yhat_test - Y_test).mean()
```

接着，拟合lasso模型。注意此处使用绝对平均误差评估模型性能，因此需要额外使用一个交叉检验网格寻找最优模型：

```
scaler = StandardScaler(with_mean=True, with_std=True) #创建标准比较器
lasso = Lasso(warm_start=True, max_iter=30000) #获取Lasso回归模型
standard_lasso = Pipeline(steps=[('scaler', scaler),
                                 ('lasso', lasso)]) #得到标准化lasso回归
```

`StandardScaler` 是一个 `sklearn` 中的类，用于数据标准化处理， `with_mean` 和 `with_std` 分别指定是否转换为均值为0和标准差为1的分布； `Lasso` 模型参数 `warm_start` 表示每次调用 `fit` 方法时从上一次训练结果继续训练，常用于多次迭代， `max_iter` 指定最大迭代次数

按照通常的做法，我们需要为λ值设计一个网格，这里使用的100个网格值在对数尺度上均匀分布，其中 `lam_max` 表示使得lasso回归系数全为0的最小λ值(在数学上等于任何预测变量与中心化响应变量之间最大绝对内积，超过本书内容)，搜索范围是$[0.01\mathrm{lam_max}, \mathrm{lam_max}]$：

```
X_s = scaler.fit_transform(X_train) #标准化设计矩阵
n = X_s.shape[0]
lam_max = np.fabs(X_s.T.dot(Y_train - Y_train.mean())).max() / n #获得最大
lambda值
param_grid = {'lasso__alpha': np.exp(np.linspace(0, np.log(0.01), 100))
              * lam_max}
```

> 必须先对数据进行标准化，因为数据规模影响λ值选取

现在进行网格搜索，拟合最优lasso模型：

```
cv = KFold(10,
           shuffle=True,
           random_state=1)
grid = GridSearchCV(standard_lasso,
                    param_grid,
                    cv=cv,
                    scoring='neg_mean_absolute_error')
grid.fit(X_train, Y_train)
```

显示最优lasso模型的平均绝对误差：

```
trained_lasso = grid.best_estimator_
Yhat_test = trained_lasso.predict(X_test)
print(np.fabs(Yhat_test - Y_test).mean())
```

这个值与最小二乘回归非常接近。然而，使用不同的随机种子，我们可能得到不一样的结果

为了拟合神经网络，我们需要建立描述网络的模型结构，这需要我们针对我们希望拟合的模型定义新的类，这通常是在 `pytorch` 中通过对网络的泛型表示进行子类化来完成的：

```
class HittersModel(nn.Module): #指定模型大小参数

    def __init__(self, input_size): #类的构造函数
        super(HittersModel, self).__init__() #继承父类的所有属性
        self.flatten = nn.Flatten() #定义flatten的属性，来自nn.Flatten()
        self.sequential = nn.Sequential( #定义sequential属性
            nn.Linear(input_size, 50),
            nn.ReLU(),
            nn.Dropout(0.4),
            nn.Linear(50, 1))

    def forward(self, x): #定义forward函数
```

```
        x = self.flatten(x)
        return torch.flatten(self.sequential(x))
```

对象 `nn.Module` 还有许多强大的功能，例如 `eval()` 可以使我们不对神经元进行dropout而评估测试数据

新建一个神经网络实例：

```
hit_model = HittersModel(X.shape[1])
```

对象 `self.sequential` 由四个映射组成。第一个映射将输入特征转换为50个维度，共有$50\times19+50$个参数；这一层然后被ReLU层映射，以实现40%的神经元丢弃，共有50个参数；最后是一个线性映射，将隐藏层的数据线性组合为一个输出结果，包含$50+1$个参数

`torchinfo` 包提供的 `summary()` 函数可以整洁地显示模型信息：

```
print(summary(hit_model,
              input_size=X_train.shape,
              col_names=['input_size',
                         'output_size',
                         'num_params']))
```

为了输入我们的训练数据，需要将他们转换为 `pytorch` 可以接受的形式。 `pytorch` 接受32位浮点数的张量，这与我们之前用到的 `ndarray` 很类似。为了转换数据，我们用到 `.tensor()` 方法，最后需要将X和Y同时转换到一个数据集中：

```
X_train_t = torch.tensor(X_train.astype(np.float32))
Y_train_t = torch.tensor(Y_train.astype(np.float32))
hit_train = TensorDataset(X_train_t, Y_train_t) #合并X和Y

X_test_t = torch.tensor(X_test.astype(np.float32))
Y_test_t = torch.tensor(Y_test.astype(np.float32))
hit_test = TensorDataset(X_test_t, Y_test_t)
```

然后，数据集将会被传送到 `DataLoader()` 上，并由它加载到我们的神经网络中。这一过程看似繁琐，但对于不同机器上的数据和一些必须用GPU训练的模型是很有帮助的。 `ISLP` 中的辅助函数 `SimpleDataModule` 能够帮助我们更容易地进行这个任务，其中的 `num_workers` 参数指定我们加载数据的进程数。 `torch` 包能够为我们决定运行任务的最大进程数量，同时函数 `rec_num_workers()` 可以计算多少进程是合理的：

```
max_num_workers = rec_num_workers()
```

`Pytorch Lightning` 的一般训练过程包括训练数据，验证数据和测试数据三个步骤，这些数据由不同的加载器表示。在每个训练周期(epoch)中，我们运行一个训练步骤，并运行一个验

证步骤跟踪误差，最后用测试数据评估模型

本节用到的例子数据只被划分为了测试集和训练集，因此用测试集来代替验证集，这可用 `validation=hit_test` 参数设置实现。 `validation` 参数可以接受一个0-1之间的浮点数，一个整数或者一个数据集。如果是浮点数和整数会被认为是训练观察值的百分比或数量，若为数据集则直接传递给数据加载器：

```
hit_dm = SimpleDataModule(
    hit_train,
    hit_test,
    batch_size=32,
    num_workers=min(4, max_num_workers),
    validation=hit_test
)
```

其中 `batch_size` 指定每个训练周期中的批次大小， `num_workers` 限定了运行进程数量

下一步，我们需要提供一个 `pytorch_lightning` 模块记录训练过程中的每一步，`SimpleModule` 已经实现了一些基本方法来记录每个周期结束时的损失函数值和其他指标(由 `SimpleModule.[training/test/validation]_step()` 控制，本节不修改这些指标)：

```
hit_module = SimpleModule.regression(hit_model,
                             metrics={'mae':MeanAbsoluteError()})
```

`SimpleModule.regression()` 方法简化了定义回归模型的过程，且指定我们的损失函数为均方误差； `metrics` 是一个字典，用于指定训练过程中要跟踪的指标

用 `CSVLogger()` 加载结果日志，他将会将结果保存为一个csv文件，之后可用 `pd.DataFrame()` 可视化：

```
hit_logger = CSVLogger('logs', name='hitters')
```

最后，使用 `pytorch lightning` 的 `Trainer` 对象管理模型的训练、验证和测试过程。`datamodule` 参数告诉训练器如何生成训练、验证和测试日志， `hit_module` 参数定义网络架构和训练步骤等，指定训练模型， `callbacks` 参数允许我们在训练过程中执行多个任务，这里的 `ErrorTracker()` 召回函数在训练过程中计算验证误差，训练结束后计算测试误差：

```
hit_trainer = Trainer(deterministic=True,
                      max_epochs=50, #最大训练周期数
                      log_every_n_steps=5, #日志记录频率为5步
                      logger=hit_logger, #指定日志加载器
                      callbacks=[ErrorTracker()]) #使用的回调函数列表
hit_trainer.fit(hit_module, datamodule=hit_dm)
```

在模型拟合完毕后，使用 `test()` 函数进行评估：

```
hit_trainer.test(hit_module, datamodule=hit_dm)
```

现在，创建一个MAE的图作为训练周期的函数。首先检索日志摘要：

```
hit_results = pd.read_csv(hit_logger.experiment.metrics_file_path)
```

为了简化后续的操作，这里写了一个函数来创建类似的图：

```
def summary_plot(results,
                 ax,
                 col='loss',
                 valid_legend='Validation',
                 training_legend='Training',
                 ylabel='Loss',
                 fontsize=20):
    for(column,
        color,
        label) in zip([f'train_{col}_epoch',
                       f'valid_{col}'],
                      ['black',
                       'red'],
                      [training_legend,
                       valid_legend]):
        results.plot(x='epoch',
                     y=column,
                     label=label,
                     marker='o',
                     color=color,
                     ax=ax)
    ax.set_xlabel('Epoch')
    ax.set_ylabel(ylabel)
    return ax
```

然后查看图片：

```
fig, ax = subplots(1, 1, figsize=(6, 6))
ax = summary_plot(hit_results,
                  ax,
                  col='mae',
                  ylabel='MAE',
                  valid_legend='Validation (=Test)')
ax.set_ylim([0, 400])
ax.set_xticks(np.linspace(0, 50, 11).astype(int))
```

使用 `eval()` 方法修改最终模型，告诉 `torch` 神经元不必随机丢弃，以适应新数据的预测：

```
hit_model.eval()
preds = hit_module(X_test_t)
print(torch.abs(Y_test_t - preds).mean())
```

在建立数据模块时，我们启动过的进程将会持续工作。在任务结束后，我们需要删除这些进程：

```
del(Hitters,
    hit_model, hit_dm,
    hit_logger,
    hit_test, hit_train,
    X, Y,
    X_test, X_train,
    Y_test, Y_train,
    X_test_t, Y_test_t,
    hit_trainer, hit_module)
```

10.9.2 多层神经网络拟合MNIST数字数据集

`torchvision`包自带很多数据集，例如`MNIST`数据集。工作流的第一步是合成训练和测试数据集，`MNIST()`函数可以实现这些步骤。首次执行函数将会下载数据集到`data/MNIST`目录中：

```
(mnist_train,
 mnist_test) = [MNIST(root='data',
                      train=train,
                      download=True,
                      transform=ToTensor())
               for train in [True, False]]
mnist_train
```

神经网络对输入数据的尺度有敏感性，这是因为正则化项。现在需要`MNIST`数据集中的图像像素映射到单位区间。如同单层神经网络，现在划分数据集：

```
mnist_dm = SimpleDataModule(mnist_train,
                            mnist_test,
                            validation=0.2, #20%的测试数据
                            num_workers=max_num_workers,
                            batch_size=256)
```

查看我们的数据集，显示前两个批次：

```
for idx, (X_ ,Y_) in enumerate(mnist_dm.train_dataloader()):
    print('X: ', X_.shape)
    print('Y: ', Y_.shape)
```

```
    if idx >= 1:
        break
```

输出的结果如下：

```
X:  torch.Size([256, 1, 28, 28])
Y:  torch.Size([256])
X:  torch.Size([256, 1, 28, 28])
Y:  torch.Size([256])
```

每个批次的X由256张大小为28×28的图像组成，数字1表示单通道(灰度)。对于`CIFAR100`这样的RGB图像，通道数为3。现在构造针对这个数据集的神经网络：

```
class MNISTModel(nn.Module):
    def __init__(self):
        super(MNISTModel, self).__init__()
        self.layer1 = nn.Sequential( #第一层属性设置
            nn.Flatten(),
            nn.Linear(28*28, 256), #将28×28的图像展平为一维向量
            nn.ReLU(),
            nn.Dropout(0.4))
        self.layer2 = nn.Sequential( #第二层属性设置
            nn.Linear(256, 128), #将256维映射到128维
            nn.ReLU(),
            nn.Dropout(0.3))
        self._forward = nn.Sequential(
            self.layer1,
            self.layer2,
            nn.Linear(128, 10)) #将128维映射到10个类别

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

#使用模型
minst_model = MNISTModel()
```

根据已有的批次`X_`检验模型是否产生期望大小的输出：

```
mnist_model(X_).size()
```

显示模型摘要：

```
summary(mnist_model,
        input_size=X_,
        col_names=['input_size',
```

```
                    'output_size',
                    'num_params'])
```

到这一步，我们的剩余步骤与单层神经网络几乎类似。一个不同的点是我们使用 `SimpleModule.classification()` 来执行分类任务，它使用交叉熵来优化模型：

```
mnist_module = SimpleModule.classification(mnist_model,
                                           num_classes=10)
mnist_logger = CSVLogger('logs', name='MNIST') #设置日志提取器
```

准备工作完成了，最后一步是用训练数据训练模型：

```
mnist_trainer = Trainer(deterministic=True,
                        max_epochs=30,
                        logger=mnist_logger,
                        enable_progress_bar=False,
                        callbacks=[ErrorTracker()])
mnist_trainer.fit(mnist_module,
                  datamodule=mnist_dm)
```

训练这个模型需要花费一定的时间。`SimpleModule.classification()` 默认输出一个精确度度量，此外指定 `torchmetrics` 参数可以显示其他的指标：

```
mnist_results = pd.read_csv(mnist_logger.experiment.metrics_file_path)
fig, ax = subplots(1, 1, figsize=(6, 6))
summary_plot(mnist_results,
             ax,
             col='accuracy',
             ylabel='Accuracy')
ax.set_ylim([0.5, 1])
ax.set_ylabel('Accuracy')
ax.set_xticks(np.linspace(0, 30, 7).astype(int))
```

再次使用 `test()` 函数来生成测试误差：

```
mnist_trainer.test(mnist_module,
                   datamodule=mnist_dm)
```

一个特别的想法是用 `torch` 架构拟合LDA模型，尽管我们可以用 `sklearn` 中的 `LogisticRegression()` 来实现。一个只有输入和输出层的网络能够满足要求：

```
#定义神经网络，此处没有隐藏层
class MNIST_MLR(nn.Module):
    def __init__(self):
        super(MNIST_MLR, self).__init__()
```

```
        self.linear = nn.Sequential(nn.Flatten(),
                                    nn.Linear(784, 10))
    def forward(self, x):
        return self.linear(x)

#使用模型
mlr_model = MNIST_MLR()
mlr_module = SimpleModule.classification(mlr_model,
                                         num_classes=10)
mlr_logger = CSVLogger('logs', name='MNIST_MLR')

#进行训练
mlr_trainer = Trainer(deterministic=True,
                      max_epochs=30,
                      enable_progress_bar=False,
                      callbacks=[ErrorTracker()])
mlr_trainer.fit(mlr_module, datamodule=mnist_dm)

#输出误差
mlr_trainer.test(mlr_module,
                 datamodule=mnist_dm)
```

这个简单的模型准确率也达到了90%以上。最后我们删除进程：

```
del(mnist_test,
    mnist_train,
    mnist_model,
    mnist_dm,
    mnist_trainer,
    mnist_module,
    mnist_results,
    mlr_model,
    mlr_module,
    mlr_trainer)
```

10.9.3 CNNs

本节用CNN拟合 `torchvision` 中的 `CIFAR100` 数据集，这个数据集的结构与 `MNIST` 类似：

```
(cifar_train,
 cifar_test) = [CIFAR100(root="data",
                         train=train,
                         download=True)
               for train in [True, False]]
```

`CAFAR100` 数据集由50000张训练数据图像组成，每张图像是三维张量。下面用 `ToTensor()` 转换器实现标准化处理，且保留数组的结构：

```
transform = ToTensor() #数据转换器能够处理图像数据，映射像素到[0, 1]上
cifar_train_X = torch.stack([transform(x) for x in
                             cifar_train.data])
cifar_test_X = torch.stack([transform(x) for x in
                             cifar_test.data])
cifar_train = TensorDataset(cifar_train_X, #TensorDataset()将数据张量组合为数据
集
                             torch.tensor(cifar_train.targets))
cifar_test = TensorDataset(cifar_test_X,
                             torch.tensor(cifar_test.targets))
```

建立数据模块与 `MNIST` 类似：

```
cifar_dm = SimpleDataModule(cifar_train,
                             cifar_test,
                             validation=0.2,
                             num_workers=max_num_workers,
                             batch_size=128)
```

检查每批次的数据是否符合我们的预期：

```
for idx, (X_ ,Y_) in enumerate(cifar_dm.train_dataloader()):
    print('X: ', X_.shape)
    print('Y: ', Y_.shape)
    if idx >= 1:
        break
```

现在我们可以查看一些训练图像：

```
fig, axes = subplots(5, 5, figsize=(10,10))
rng = np.random.default_rng(4)
indices = rng.choice(np.arange(len(cifar_train)), 25,
                     replace=False).reshape((5,5)) #生成所有索引的数组，并随机选
取25个元素
for i in range(5):
    for j in range(5):
        idx = indices[i,j]
        axes[i,j].imshow(np.transpose(cifar_train[idx][0],
                                      [1,2,0]),
                                      interpolation=None)
        axes[i,j].set_xticks([]) #移除当前子图的刻度
        axes[i,j].set_yticks([])
```

这将生成一个5×5的网格，并且在每个子图中随机显示 `CIFAR` 图像。 `imshow()` 方法从参数形状中识别三维数组，并且根据维度绘制图像

现在建立一个中等大小的CNN模型用于演示，这里包括多个层，每层由卷积、ReLU和max-pooling步骤组成，下面是模块中一个层的定义：

```
class BuildingBlock(nn.Module):

    def __init__(self,
                 in_channels, #输入特征图的通道数
                 out_channels): #输出特征图的通道数

        super(BuildingBlock, self).__init__()
        self.conv = nn.Conv2d(in_channels=in_channels, #二维卷积层
                              out_channels=out_channels,
                              kernel_size=(3,3),
                              padding='same')
        self.activation = nn.ReLU() #激活函数层
        self.pool = nn.MaxPool2d(kernel_size=(2,2)) #池化层

    def forward(self, x):
        return self.pool(self.activation(self.conv(x)))
```

这里的 `padding='same'` 参数表示确保输出特征图与输入特征图大小相同(使用了填充方法)。第一个隐藏层有32通道，输入层中有3个通道，每层使用3×3卷积滤波器，卷积后接上2×2池化层

有了构建模块，我们可以按顺序使用 `BuildingBlock()` 模块进行组合：

```
class CIFARModel(nn.Module):

    def __init__(self):
        super(CIFARModel, self).__init__()
        sizes = [(3,32), #这里存储了每个卷积层的输入通道数和输出通道数
                 (32,64),
                 (64,128),
                 (128,256)]
        self.conv = nn.Sequential(*[BuildingBlock(in_, out_) #依次将参数传入到构建模块中
                                    for in_, out_ in sizes])

        self.output = nn.Sequential(nn.Dropout(0.5),
                                    nn.Linear(2*2*256, 512),
                                    nn.ReLU(),
                                    nn.Linear(512, 100))
    def forward(self, x):
        val = self.conv(x)
        val = torch.flatten(val, start_dim=1)
        return self.output(val)
```

> `*`是一种解包方式

构建模型，并查看摘要：

```
cifar_model = CIFARModel()
summary(cifar_model,
        input_data=X_,
        col_names=['input_size',
                   'output_size',
                   'num_params'])
```

摘要提示参数共有964516个。池化操作每次将通道从两个维度上减半，最后一次我们得到256个2×2通道图层，并压平为1024大小的致密层(长度为4的向量)

目前我们使用的都是`SimpleModule()`中的默认优化器。实验表明，较小的学习了比默认的0.01性能更好，下面自定义一个学习率为0.001的自定义优化器：

```
cifar_optimizer = RMSprop(cifar_model.parameters(), lr=0.001) #创建一个优化器实例，通过调整学习率来优化算法
cifar_module = SimpleModule.classification(cifar_model,
                                           num_classes=100,
                                           optimizer=cifar_optimizer) #使用新的优化器
cifar_logger = CSVLogger('logs', name='CIFAR100')
```

训练模型：

```
cifar_trainer = Trainer(deterministic=True,
                        max_epochs=30,
                        logger=cifar_logger,
                        enable_progress_bar=False,
                        callbacks=[ErrorTracker()])
cifar_trainer.fit(cifar_module,
                  datamodule=cifar_dm)
```

> 此模型需要10mins以上运行，测试准确率为42%。对于网络上75%左右的分类结果，需要花费大量时间构筑模型结构、正则化等

查看随着训练周期增加的验证和训练准确性：

```
log_path = cifar_logger.experiment.metrics_file_path
cifar_results = pd.read_csv(log_path)
fig, ax = subplots(1, 1, figsize=(6, 6))
summary_plot(cifar_results,
             ax,
             col='accuracy',
             ylabel='Accuracy')
```

```
ax.set_xticks(np.linspace(0, 10, 6).astype(int))
ax.set_ylabel('Accuracy')
ax.set_ylim([0, 1])
```

模型评估：

```
cifar_trainer.test(cifar_module,
                   datamodule=cifar_dm)
```

部分硬件可以用来加速深度学习的训练过程。例如具有M1芯片的MAC OS设备可能启用了Metal编程框架，可以加速 `torch` 运算，下面是一个示例：

```
try:
    for name, metric in cifar_module.metrics.items():
        cifar_module.metrics[name] = metric.to('mps')
    cifar_trainer_mps = Trainer(accelerator='mps',
                                deterministic=True,
                                enable_progress_bar=False,
                                max_epochs=30)
    cifar_trainer_mps.fit(cifar_module,
                          datamodule=cifar_dm)
    cifar_trainer_mps.test(cifar_module,
                          datamodule=cifar_dm)
except:
    pass
```

`try` 和 `except` 语句用于保护代码块，如果加速有效则执行，否则不执行任何操作

我们可以使用预训练模型分类自己的图像。下面以 `resnet50` 为例，对书中的六张图片进行分类。首先我们需要将图像转换为 `torch` 中要求的数组格式：

```
resize = Resize((232,232), antialias=True) #Resize对象用于调整图像大小，
antialias指定抗锯齿算法
crop = CenterCrop(224) #用于裁剪中心区域，大小为224×224
normalize = Normalize([0.485,0.456,0.406],
                      [0.229,0.224,0.225]) #用于图像标准化，第一行为每个通道均值，
第二行为标准差
imgfiles = sorted([f for f in glob('book_images/*')]) #获取文件夹下的所有文件
imgs = torch.stack([torch.div(crop(resize(read_image(f))), 255)
                    for f in imgfiles]) #在处理图像后，堆叠为一个张量
imgs = normalize(imgs) #标准化
imgs.size() #显示大小
```

使用模型：

```
resnet_model = resnet50(weights=ResNet50_Weights.DEFAULT)
summary(resnet_model,
        input_data=imgs,
        col_names=['input_size',
                   'output_size',
                   'num_params'])
```

调整模型为`eval()`模式，以预测新的数据：

```
resnet_model.eval()
img_preds = resnet_model(imgs) #获取输出
```

想要查看前3个图像中每个选项的预测概率，需要先将softmax应用于`img_preds`计算概率，需要我们在张量上调用`detach()`方法将其转换为`ndarray`：

```
img_probs = np.exp(np.asarray(img_preds.detach()))
img_probs /= img_probs.sum(1)[:,None]
```

查看类标签需要先下载与`imagent`关联的索引文件：

```
labs = json.load(open('imagenet_class_index.json'))
class_labels = pd.DataFrame([(int(k), v[1]) for k, v in
                             labs.items()],
                             columns=['idx', 'label'])
class_labels = class_labels.set_index('idx')
class_labels = class_labels.sort_index()
```

为每个图像建立一个`DataFrame`，包括最高概率的三个标签：

```
for i, imgfile in enumerate(imgfiles):
    img_df = class_labels.copy()
    img_df['prob'] = img_probs[i]
    img_df = img_df.sort_values(by='prob', ascending=False)[:3]
    print(f'Image: {imgfile}')
    print(img_df.reset_index().drop(columns=['idx']))
```

最后，清理进程：

```
del(cifar_test,
    cifar_train,
    cifar_dm,
    cifar_module,
    cifar_logger,
```

```
    cifar_optimizer,
    cifar_trainer)
```

10.9.4 文档分类

这一节使用 `IMDB` 数据集进行文档分类演示，它来自 `keras` 包。由于 `keras` 使用的是 `tensorflow` 架构，所有我们需要对其进行转换以适合 `torch` 。用于转换的代码来自 `ISLP.torch._make_imdb` ，依赖于 `keras` 包运行

对于评论数据，本书将其分为三类：

- `load_tensor()` ，可被 `torch` 应用的稀疏张量
- `load_spares()` ，可应用于 `sklearn` 的稀疏矩阵(后续用于对比lasso)
- `load_sequential()` ，原始序列的填充版本，仅限于每条评论的最后500单词

```
(imdb_seq_train,
 imdb_seq_test) = load_sequential(root='data/IMDB')
padded_sample = np.asarray(imdb_seq_train.tensors[0][0])
sample_review = padded_sample[padded_sample > 0][:12]
sample_review[:12]
```

数据集 `imdb_seq_train` 和 `imdb_seq_test` 都是类 `TensorDataset` 的实例。用于构造它们的张量可以在 tensors 属性中找到，第一个张量是特征 X，第二个张量是结果 Y。我们获取了第一行特征并将其存储为 padded_sample。在用于形成这些数据的预处理中，如果序列不够长，则在开始时用 0 填充，因此我们通过限制为 `padded_sample` > 0 的条目来删除此填充。然后，我们提供样本评论的前 12 个单词

用 `ISLP.torch.imdb` 中的 `lookup` 函数查看这些单词：

```
lookup = load_lookup(root='data/IMDB')
' '.join(lookup[i] for i in sample_review)
```

我们的首个模型将数据集中的10000个单词中的每个单词创建了一个二进制特征，如果单词j出现在评论i中，则i,j会包括这样它的值。这个特征矩阵超过98%都是0：

```
max_num_workers=10
(imdb_train,
 imdb_test) = load_tensor(root='data/IMDB')
imdb_dm = SimpleDataModule(imdb_train,
                           imdb_test,
                           validation=2000,
                           num_workers=min(6, max_num_workers),
                           batch_size=512)
```

下面使用两层的模型来拟合：

```
class IMDBModel(nn.Module):

    def __init__(self, input_size):
        super(IMDBModel, self).__init__()
        self.dense1 = nn.Linear(input_size, 16)
        self.activation = nn.ReLU()
        self.dense2 = nn.Linear(16, 16)
        self.output = nn.Linear(16, 1)

    def forward(self, x):
        val = x
        for _map in [self.dense1,
                     self.activation,
                     self.dense2,
                     self.activation,
                     self.output]:
            val = _map(val)
        return torch.flatten(val)
```

初始化模型，并查看摘要：

```
imdb_model = IMDBModel(imdb_test.tensors[0].size()[1])
summary(imdb_model,
        input_size=imdb_test.tensors[0].size(),
        col_names=['input_size',
                   'output_size',
                   'num_params'])
```

此处仍然使用较小的学习率，因此初始化学习器模型：

```
imdb_optimizer = RMSprop(imdb_model.parameters(), lr=0.001)
imdb_module = SimpleModule.binary_classification(
                        imdb_model,
                        optimizer=imdb_optimizer)
```

注意这里的模型是`SimpleModule.binary_classification()`，表示二元分类问题(将评论分为好评和恶评)

后续步骤同前面类似：

```
imdb_logger = CSVLogger('logs', name='IMDB')
imdb_trainer = Trainer(deterministic=True,
                       max_epochs=30,
                       logger=imdb_logger,
                       enable_progress_bar=False,
                       callbacks=[ErrorTracker()])
```

```
imdb_trainer.fit(imdb_module,
                 datamodule=imdb_dm)
```

进行评估，准确率在86%左右：

```
print(test_results = imdb_trainer.test(imdb_module , datamodule=imdb_dm))
```

我们现在用 `sklearn` 中的 `LogisticRegression()` 模型来拟合Lasso回归，以进行对比：

```
((X_train, Y_train),
 (X_valid, Y_valid),
 (X_test, Y_test)) = load_sparse(validation=2000,
                                 random_state=0,
                                 root='data/IMDB')
```

上面的代码划分了数据集，使用 `ISLP` 中的数据转换方法

与之前做法类似，我们划分一个均匀的λ搜索区间：

```
lam_max = np.abs(X_train.T * (Y_train - Y_train.mean())).max()
lam_val = lam_max * np.exp(np.linspace(np.log(1),
                                       np.log(1e-4), 50))
```

使用 `LogisticRegression()` 时，正则化参数 `C` 指定为λ的倒数；用 `liblinear` 求解器能够好地适应稀疏矩阵数据：

```
logit = LogisticRegression(penalty='l1',
                           C=1/lam_max,
                           solver='liblinear',
                           warm_start=True,
                           fit_intercept=True)
```

这大约花费40s来运行，搜索了50个值的路径；系数和截取具有无关维度， `np.squeeze()` 可以删除它们：

```
coefs = []
intercepts = []

for l in lam_val:
    logit.C = 1/l
    logit.fit(X_train, Y_train)
    coefs.append(logit.coef_.copy())
    intercepts.append(logit.intercept_)
```

```
coefs = np.squeeze(coefs)
intercepts = np.squeeze(intercepts)
```

下面我们来绘制图像以比较这两个模型：

```
%%capture
fig, axes = subplots(1, 2, figsize=(16, 8), sharey=True)
for ((X_, Y_),
     data_,
     color) in zip([(X_train, Y_train),
                    (X_valid, Y_valid),
                    (X_test, Y_test)],
                    ['Training', 'Validation', 'Test'],
                    ['black', 'red', 'blue']):
    linpred_ = X_ * coefs.T + intercepts[None,:]
    label_ = np.array(linpred_ > 0)
    accuracy_ = np.array([np.mean(Y_ == l) for l in label_.T])
    axes[0].plot(-np.log(lam_val / X_train.shape[0]),
                 accuracy_,
                 '.--',
                 color=color,
                 markersize=13,
                 linewidth=2,
                 label=data_)
axes[0].legend()
axes[0].set_xlabel(r'$-\log(\lambda)$', fontsize=20)
axes[0].set_ylabel('Accuracy', fontsize=20)
```

注意 %%capture 会抑止部分完成的图像显示，这在制作复杂图像时非常有用

添加lasso精度图像：

```
imdb_results = pd.read_csv(imdb_logger.experiment.metrics_file_path)
summary_plot(imdb_results,
             axes[1],
             col='accuracy',
             ylabel='Accuracy')
axes[1].set_xticks(np.linspace(0, 30, 7).astype(int))
axes[1].set_ylabel('Accuracy', fontsize=20)
axes[1].set_xlabel('Epoch', fontsize=20)
axes[1].set_ylim([0.5, 1]);
axes[1].axhline(test_results[0]['test_accuracy'],
                color='blue',
                linestyle='--',
                linewidth=3)
```

最后，清除进程：

```
del(imdb_model,
    imdb_trainer,
    imdb_logger,
    imdb_dm,
    imdb_train,
    imdb_test)
```

10.9.5 RNNs

首先介绍用于文档分类的RNN。这里将一个简单的LSTM RNN拟合到`IMDb`数据集中。由于超过 90% 的文档少于 500 字，因此我们将文档长度设置为 500。对于较长的文档，我们使用了最后 500 个单词，对于较短的文档，我们在前面填充了空格：

```
imdb_seq_dm = SimpleDataModule(imdb_seq_train,
                               imdb_seq_test,
                               validation=2000,
                               batch_size=300,
                               num_workers=min(6, max_num_workers)
                               )
```

RNN的第一层是大小为32的嵌入层，独热编码将每个文档映射为500×10003的矩阵，然后将这些 10,003 维度向下映射到 32 。{额外的 3 个维度对应于评论中常见的非单词条目。由于每个单词都由一个整数表示，因此可以通过创建 size 10,003×32 的嵌入矩阵来有效地实现这一点;然后，通过索引此矩阵的相应行，将文档中的 500 个整数中的每一个映射到适当的 32 个实数

第二层是32个单元的LSTM，输出层是二元分类任务的单个logit：

```
class LSTMModel(nn.Module):
    def __init__(self, input_size):
        super(LSTMModel, self).__init__()
        self.embedding = nn.Embedding(input_size, 32)
        self.lstm = nn.LSTM(input_size=32,
                            hidden_size=32,
                            batch_first=True)
        self.dense = nn.Linear(32, 1)
    def forward(self, x):
        val, (h_n, c_n) = self.lstm(self.embedding(x))
        return torch.flatten(self.dense(val[:,-1]))
```

实例化，查看模型摘要：

```
lstm_model = LSTMModel(X_test.shape[-1])
summary(lstm_model,
        input_data=imdb_seq_train.tensors[0][:10],
        col_names=['input_size',
```

```
                    'output_size',
                    'num_params'])
```

剩余的部分与其他网络类似，性能测试表明它达到了85%左右的准确率：

```
lstm_module = SimpleModule.binary_classification(lstm_model)
lstm_logger = CSVLogger('logs', name='IMDB_LSTM')

lstm_trainer = Trainer(deterministic=True,
                       max_epochs=20,
                       logger=lstm_logger,
                       enable_progress_bar=False,
                       callbacks=[ErrorTracker()])
lstm_trainer.fit(lstm_module,
                 datamodule=imdb_seq_dm)

lstm_trainer.test(lstm_module, datamodule=imdb_seq_dm)
```

下面展示训练过程的学习进度图像，然后清理进程：

```
lstm_results = pd.read_csv(lstm_logger.experiment.metrics_file_path)
fig, ax = subplots(1, 1, figsize=(6, 6))
summary_plot(lstm_results,
             ax,
             col='accuracy',
             ylabel='Accuracy')
ax.set_xticks(np.linspace(0, 20, 5).astype(int))
ax.set_ylabel('Accuracy')
ax.set_ylim([0.5, 1])

del(lstm_model,
    lstm_trainer,
    lstm_logger,
    imdb_seq_dm,
    imdb_seq_train,
    imdb_seq_test)
```

之后，我们进行时间序列预测。首先加载数据：

```
NYSE = load_data('NYSE')
cols = ['DJ_return', 'log_volume', 'log_volatility']
X = pd.DataFrame(StandardScaler(
                     with_mean=True,
                     with_std=True).fit_transform(NYSE[cols]),
                 columns=NYSE[cols].columns,
                 index=NYSE.index)
```

现在需要设置数据的滞后版本，使用 `dropna()` 删除任何具有确实值的行：

```
for lag in range(1, 6):
    for col in cols:
        newcol = np.zeros(X.shape[0]) * np.nan
        newcol[lag:] = X[col].values[:-lag]
        X.insert(len(X.columns), "{0}_{1}".format(col, lag), newcol)
X.insert(len(X.columns), 'train', NYSE['train'])
X = X.dropna()
```

提取响应，训练指标。当天的两类指标 `DJ_return` 和 `log_volidatility` 要从前一天数据中预测：

```
Y, train = X['log_volume'], X['train']
X = X.drop(columns=['train'] + cols)
print(X.columns)
```

你和一个简单的线性模型，并用 `score()` 方法计算R^2：

```
M = LinearRegression()
M.fit(X[train], Y[train])
M.score(X[~train], Y[~train])
```

我们重新调整了这个模型，包括因子变量 `day_of_week` 。对于 `pandas` 中的 分类序列，我们可以使用该方法 `get_dummies()` 形成指标：

```
X_day = pd.concat([X,
                   pd.get_dummies(NYSE['day_of_week'])],
                   axis=1).dropna()

M.fit(X_day[train], Y[train])
M.score(X_day[~train], Y[~train])
```

注意，我们不必重新实例化线性回归模型，因为它 `fit()` 的方法直接接受设计矩阵和响应

为了拟合 RNN，我们必须重塑数据，因为它将期望每个特征的 5 个滞后版本，如下面层 `nn.RNN()` 的 `input_shape` 参数所示。我们首先确保数据框的列是这样的，这样重塑的矩阵将正确滞后于变量。我们使用 this `reindex()` method 来执行此操作:

```
ordered_cols = []
for lag in range(5,0,-1):
    for col in cols:
        ordered_cols.append('{0}_{1}'.format(col, lag))
X = X.reindex(columns=ordered_cols)
X.columns
```

重塑数据：

```
X_rnn = X.to_numpy().reshape((-1,5,3))
```

构建RNN模型，包括12个隐藏单元和10%的随机丢弃：

```
class NYSEModel(nn.Module):
    def __init__(self):
        super(NYSEModel, self).__init__()
        self.rnn = nn.RNN(3,
                          12,
                          batch_first=True)
        self.dense = nn.Linear(12, 1)
        self.dropout = nn.Dropout(0.1)
    def forward(self, x):
        val, h_n = self.rnn(x)
        val = self.dense(self.dropout(val[:,-1]))
        return torch.flatten(val)
nyse_model = NYSEModel()
```

形成类似`Hitters`的数据集：

```
datasets = []
for mask in [train, ~train]:
    X_rnn_t = torch.tensor(X_rnn[mask].astype(np.float32))
    Y_t = torch.tensor(Y[mask].astype(np.float32))
    datasets.append(TensorDataset(X_rnn_t, Y_t))
nyse_train, nyse_test = datasets
```

输出摘要：

```
summary(nyse_model,
        input_data=X_rnn_t,
        col_names=['input_size',
                   'output_size',
                   'num_params'])
```

合成数据到一个模块，批次大小为64：

```
nyse_dm = SimpleDataModule(nyse_train,
                           nyse_test,
                           num_workers=min(4, max_num_workers),
                           validation=nyse_test,
                           batch_size=64)

#输出前3个批次的数据，确保格式无误
```

```
for idx, (x, y) in enumerate(nyse_dm.train_dataloader()):
    out = nyse_model(x)
    print(y.size(), out.size())
    if idx >= 2:
        break
```

按照前面的示例设置回归训练器，并跟踪每个周期的R^2指标：

```
nyse_optimizer = RMSprop(nyse_model.parameters(),
                         lr=0.001)
nyse_module = SimpleModule.regression(nyse_model,
                                      optimizer=nyse_optimizer,
                                      metrics={'r2':R2Score()})
```

检测线性结果：

```
nyse_trainer = Trainer(deterministic=True,
                       max_epochs=200,
                       enable_progress_bar=False,
                       callbacks=[ErrorTracker()])
nyse_trainer.fit(nyse_module,
                 datamodule=nyse_dm)
nyse_trainer.test(nyse_module,
                  datamodule=nyse_dm)
```

只使用一个`nn.Flatten()`的无层`nn.RNN()`模型可以代替之前提到的非线性AR模型，无隐藏层则为线性AR模型。这里用包含`day_of_week`指标的特征集`X_day`拟合非线性AR模型：

```
#获取符合格式的数据集
datasets = []
for mask in [train, ~train]:
    X_day_t = torch.tensor(
                  np.asarray(X_day[mask]).astype(np.float32))
    Y_t = torch.tensor(np.asarray(Y[mask]).astype(np.float32))
    datasets.append(TensorDataset(X_day_t, Y_t))
day_train, day_test = datasets

#创建数据模块
day_dm = SimpleDataModule(day_train,
                          day_test,
                          num_workers=min(4, max_num_workers),
                          validation=day_test,
                          batch_size=64)

#创建模型
```

```
class NonLinearARModel(nn.Module):
    def __init__(self):
        super(NonLinearARModel, self).__init__()
        self._forward = nn.Sequential(nn.Flatten(),
                                      nn.Linear(20, 32),
                                      nn.ReLU(),
                                      nn.Dropout(0.5),
                                      nn.Linear(32, 1))
    def forward(self, x):
        return torch.flatten(self._forward(x))

#创建模型实例
nl_model = NonLinearARModel()
nl_optimizer = RMSprop(nl_model.parameters(),
                           lr=0.001)
nl_module = SimpleModule.regression(nl_model,
                                        optimizer=nl_optimizer,
                                        metrics={'r2':R2Score()})

#训练模型, 查看误差
nl_trainer = Trainer(deterministic=True,
                     max_epochs=20,
                     enable_progress_bar=False,
                     callbacks=[ErrorTracker()])
nl_trainer.fit(nl_module, datamodule=day_dm)
nl_trainer.test(nl_module, datamodule=day_dm)
```

#CS
#ML