華東理工大學

模式识别大作业

题　　目	蛋白质二级结构预测
学　　院	信息科学与工程学院
专　　业	信息与通信工程
组　　员	秦祥翔 马磊 付锡欣 汤强 杨昊晨
指导教师	赵海涛

完成日期：　2018 年 12 月 14 日

模式识别大作业报告——蛋白质二级结构预测

组员：秦祥翔 马磊 付锡欣 汤强 杨昊晨

本次大作业使用的题目是 LintCode 网站上的一道 AI 题：蛋白质的二级结构预测，题目给定了氨基酸的序列以及相应的二级结构作为训练数据，要求训练一个分类器用于蛋白质二级结构的预测。

本次实验考虑了多种方案，其中包括 BP_NN 和 CNN。BP 神经网络实质上实现了一个从输入到输出的映射功能，数学理论证明三层的神经网络就能够以任意精度逼近任何非线性连续函数。这使得其特别适合于求解内部机制复杂的问题，例如本次实验要做的蛋白质内部的氨基酸之间就有着复杂的相互关系从而使蛋白质有着不同的二级结构。也就是说，BP 神经网络具有较强的非线性映射能力。卷积神经网络（CNN）常用于图像分类领域，但随着深度学习的不断发展，CNN 也在自然语言处理、语音识别等多领域取得了重大突破，因此，本次实验也使用了 CNN 对蛋白质的二级结构进行预测。本实验使用 BP_NN 预测的准确率为 0.73831，在网站的成绩排名中暂列第二，使用 CNN 进行预测的准确率为 0.66134，CNN 准确率较低的原因可能是网络的结构（比如卷积层的数量，卷积核的大小）不够完善，这也说明了有时候最复杂的模型并不是解决问题的最好办法。

一、 题目分析

1.1 题目介绍

氨基酸序列是一串由多个氨基酸单体排列组成的链式结构。一共有 20 种常见的氨基酸单体，其单个字母缩写表示为：[A R N D C Q E G H I L K M F P S T W Y V]。而每一条氨基酸序列在折叠成蛋白质的过程中，会逐渐显现出其二级结构，在这里我们使用较为简单的 3 分类模型：[α-螺旋，β-折叠以及无规卷曲（random coil）]，分别用[H，E，C]来表示。

当一条氨基酸序列经过折叠后，序列中不同的区域会显现出不同的二级结构，因此，序列中的每个氨基酸单体都可以找到其对应的二级结构类型，而相邻的氨基酸单体往往会属于同一个二级结构，因为不论是α-螺旋还是β-折叠，都是多个氨基酸单体折叠在一起后才产生的结构。

1.2 数据分析并选择合适的模型

由于要求的是对每个氨基酸单体进行分类，那么最简单的想法就是一条氨基酸链的每一个单体都会是一个训练样本，但是通过分析所给的数据，数据中每个氨基酸序列的长度在 50~100 之间，即氨基酸链的长度不相等，且同一条氨基酸链的单体之间是有联系的，这直接影响了单体的二级结构类别，也就是将每个单体作为独立的一个样本去训练是不可取的。

既然一条氨基酸链的单体之间是有联系的，那么，我们考虑将每一个氨基酸链作为一个样本去训练，那么，每个单体的类型将会作为氨基酸链样本的特征，同时因为每个氨基酸链的长度不一样，根据题目给出的信息，将每个氨基酸链转换成 2 维的数组，其中每一行表示一个氨基酸单体，每一列为对应的单体的种类，即对于每一行来说，只会有某一列的值为 1，表示了该行对应单体是 20 种氨基酸单体[A R N D C Q E G H I L K M F P S T W Y V]中的哪一个，该行其余列应为 0（因为对于每一行来说，氨基酸单体是确定的，只会对应 20 个字母中的一个，不可能同时对应多个字母）。按照这种数据格式，对应的标签的每一列为该行单体对应的三个类别[H，E，C]，同理每一行也只会有一列为 1，其余列为 0（因为一个确定的氨基酸单体只会与三个类别字母中的一个对应）。

本次实验将会根据给出的每一条氨基酸序列以及每一个氨基酸单体对应的二级结构类别，训练一个分类器，对给定的测试数据中每一条氨基酸序列的所有单体进行二级结构类别预测。考虑到 CNN 是可以直接对二维的数组进行运算的，而对于 BP 神经网络，则需要将上述的每一条二维氨基酸链拉成一个长向量的形式作为网络的输入。但是这两种方法都是以一条氨基酸链作为一个样本来运行的。

二、 实验原理

2.1 BP 神经网络

BP(back propagation)神经网络是 1986 年由 Rumelhart 和 McClelland 为首的科学家提出的概念，是一种按照误差逆向传播算法训练的多层前馈神经网络，是目前应用最广泛的神经网络。BP 网络能学习和存贮大量的输入-输出模式映射关系，而无需事前揭示描述这种映射关系的数学方程。它通过反向传播来不断调整网络的权值和阈值，使网络的误差平方和最小。BP 神经网络的典型结构模型如图 2-1 所示。

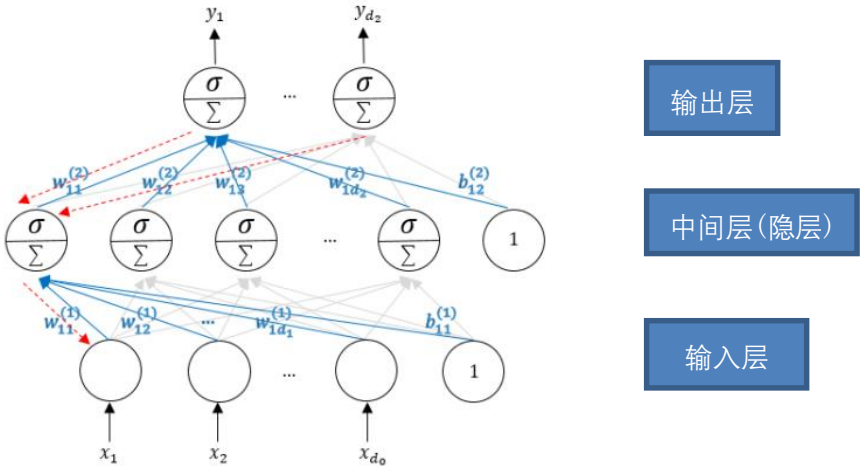

图 2-1. BP 神经网络模型

BP 神经网络是一种具有三层或者三层以上神经元的前馈神经网络，包括输入层、中间层（隐层）和输出层，上下层神经元之间实现全连接，同层的神经元无连接。图中的每个神经元的输出为：

$$y = \sigma(\sum_{i=1}^{n} w_i x_i + b)$$

其中 xi 为神经元的输入,wi 为各神经元之间的权重，表示前一个神经元对后一个神经元的影响程度，b为神经元的阈值（这里约定阈值写成与权值相似的形式，即把阈值看成样本输入为 1 的随机数），y 神经元的输出，σ(•)为神经元的激活函数，可以理解为某种非线性变换，常用的转化函数如图 2-2 所示。

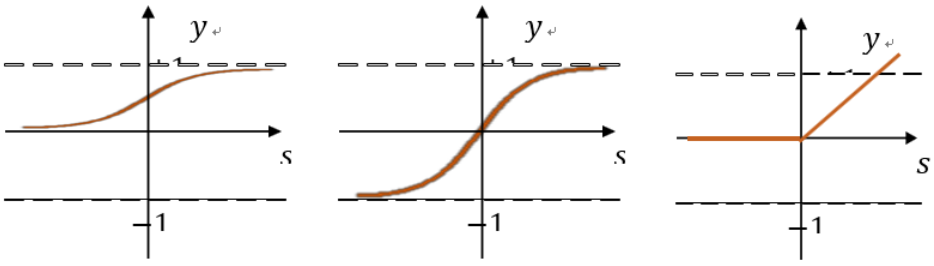

图 2-2. 常用的激活函数（从左到右依次为 sigmod 函数，tanh 函数，ReLU 函数）

BP 算法的学习过程由信息正向传播和误差反向传播两部分组成，核心是数学中的“梯度下降法”，即 BP 网络的误差调整方向总是沿着误差下降最快的方向进行。表一列出了网络的符号约定。

表 1 神经网络符号约定

层名	符号	约定
输入层（d_0个神经元）	$x_1, x_2, \dots x_{d_0}$	输入数据
中间层（d_1个神经元）	$w_1^{(1)}, w_2^{(1)}, \dots w_{d_1}^{(1)}$	权值
	$b_1^{(1)}, b_2^{(1)}, \dots b_{d_1}^{(1)}$	阈值
	$z_1^{(1)}, z_2^{(1)}, \dots z_{d_1}^{(1)}$	求和输出
	$h_1, h_2, \dots h_{d_1}$	激活函数变换输出
输出层（d_2个神经元）	$w_1^{(2)}, w_2^{(2)}, \dots w_{d_2}^{(2)}$	权值
	$b_1^{(2)}, b_2^{(2)}, \dots b_{d_2}^{(2)}$	阈值
	$z_1^{(2)}, z_2^{(2)}, \dots z_{d_2}^{(2)}$	求和输出
	$y_1, y_2, \dots y_{d_2}$	激活函数变换输出
	$t_1, t_2, \dots t_{d_2}$	真实值

在正向传播过程中，隐含层或输出层任一神经元将来自前一神经元传来的信息进行加权整合，通常还会在整合过的信息中添加一个阈值，这主要是模仿生物学中神经元必须达到一定的阈值才会触发的原理。再经激活函数变换后传到下一层神经元。

（1）从输入层到中间层

$$z_i^{(1)} = \boldsymbol{w_i^{(1)}} \boldsymbol{x} = \sum_{j=1}^{d_0} w_{ij}^{(1)} x_j + b_i^{(1)} \quad \text{求和}$$

$$h_i = \sigma\left(z_i^{(1)}\right) = \sigma\left(\sum_{j=1}^{d_0} w_{ij}^{(1)} x_j + b_i^{(1)}\right) \quad \text{变换}$$

其中：$i = 1,2 \dots d_1$，$\boldsymbol{x} = [x_1, x_2, \dots, x_{d_0}]^T$，$\boldsymbol{w_i^{(1)}} = [w_{i1}^{(1)}, w_{i2}^{(1)}, \dots, w_{id_0}^{(1)}]$

（2）从中间层到输出层

$$z_i^{(2)} = \boldsymbol{w_i^{(2)}} \boldsymbol{h} = \sum_{j=1}^{d_1} w_{ij}^{(2)} h_j + b_i^{(2)} \quad \text{求和}$$

$$y_i = \sigma\left(z_i^{(2)}\right) = \sigma\left(\sum_{j=1}^{d_1} w_{ij}^{(2)} h_j + b_i^{(2)}\right) \quad \text{变换}$$

其中：$i = 1,2 \dots d_2$，$\boldsymbol{h} = [h_1, h_2, \dots, h_{d_1}]^T$，$\boldsymbol{w_i^{(2)}} = [w_{i1}^{(2)}, w_{i2}^{(2)}, \dots, w_{id_0}^{(2)}]$

在反向传播过程中，构造关于输出值与真实值的目标函数，按照减少输出值与真实值之间误差的方向，依托梯度下降法，从输出层反向经过各中间层回到输入层，逐步修正各连接权值与阈值。构造目标函数（目标函数应为多个样本的和，

下面为简化形式）：

$$J = \frac{1}{2}\sum_{i=1}^{d_2}(t_i - y_i)^2$$

（1）从输出层到中间层

$$\frac{\partial J}{\partial z_i^{(2)}} = \frac{\partial J}{\partial y_i}\frac{\partial y_i}{\partial z_i^{(2)}} = -(t_i - y_i)\sigma'(z_i^{(2)})$$

其中：$i = 1,2,\dots d_2$，为下面书写方便，令：$\delta_i^{(2)} = -(t_i - y_i)\sigma'\left(z_i^{(2)}\right)$

$$\frac{\partial J}{\partial w_{ij}^{(2)}} = \frac{\partial J}{\partial z_i^{(2)}}\frac{\partial z_i^{(2)}}{\partial w_{ij}^{(2)}} = \delta_i^{(2)} h_j$$

$$w_{ij}^{(2)} \leftarrow w_{ij}^{(2)} - \alpha\delta_i^{(2)} h_j$$

修正权值

$$\frac{\partial J}{\partial b_i^{(2)}} = \frac{\partial J}{\partial z_i^{(2)}}\frac{\partial z_i^{(2)}}{\partial b_i^{(2)}} = \delta_i^{(2)}$$

$$b_i^{(2)} \leftarrow b_i^{(2)} - \alpha\delta_i^{(2)}$$

修正阈值

其中：$i = 1,2,\dots d_2$，$j = 1,2,\dots d_1$

（2）从中间层到输入层

因为$z_i^{(1)}$和每个$z_k^{(2)}(i = 1,2,\dots d_2)$都有关系（可以参考图 4 中的红色虚线），

因此求导时要对每个$z_k^{(2)}$进行。

$$\frac{\partial J}{\partial z_i^{(1)}} = \sum_{k=1}^{d_2}\frac{\partial J}{\partial z_k^{(2)}}\frac{\partial z_k^{(2)}}{\partial z_i^{(2)}} = \sum_{k=1}^{d_2}\frac{\partial J}{\partial z_k^{(2)}}\frac{\partial z_k^{(2)}}{\partial h_i}\frac{\partial h_i}{\partial z_i^{(1)}} = \sum_{k=1}^{d_2}\delta_k^{(2)} w_{k_i}^{(2)}\sigma'(z_i^{(1)})$$

其中：$i = 1,2,\dots d_1$，为下面书写方便，令：$\delta_i^{(1)} = \sum_{k=1}^{d_2}\delta_k^{(2)} w_{k_i}^{(2)}\sigma'(z_i^{(1)})$

$$\frac{\partial J}{\partial w_{ij}^{(1)}} = \frac{\partial J}{\partial z_i^{(1)}}\frac{\partial z_i^{(1)}}{\partial w_{ij}^{(1)}} = \delta_i^{(1)} x_j$$

$$w_{ij}^{(1)} \leftarrow w_{ij}^{(1)} - \alpha\delta_i^{(1)} x_j$$

修正权值

$$\frac{\partial J}{\partial b_i^{(1)}} = \frac{\partial J}{\partial z_i^{(1)}}\frac{\partial z_i^{(1)}}{\partial b_i^{(1)}} = \delta_i^{(1)}$$

$$b_i^{(1)} \leftarrow b_i^{(1)} - \alpha\delta_i^{(1)}$$

修正阈值

其中：$i = 1,2,\dots d_1$，$j = 1,2,\dots d_0$

BP 学习算法可以归纳如下：

(1) 初始化权值。把所有的权值和神经元节点的阈值设置成较小的随机数。

(2) 正向计算网络实际输出。逐层计算中间层及输出层各节点的输出。

(3) 反向递推更新参数。先从输出层开始，然后从输出端向输入端调整各层权值。

(4) 重复（2）-（3）过程，直到迭代次数足够或者误差达到规定指标以下。

2.2 卷积神经网络(CNN)

卷积神经网络(CNN)和常规神经网络非常相似：它们都是由神经元组成，神经元中有具有学习能力的权重和偏差。每个神经元都得到一些输入数据，进行内积运算后再进行激活函数运算。整个网络依旧是一个可导的评分函数：该函数的输入是原始的图像像素，输出是不同类别的评分。在最后一层（往往是全连接层），网络依旧有一个损失函数（比如SVM或Softmax），并且在神经网络中我们实现的各种技巧和要点依旧适用于卷积神经网络。

常规神经网络的输入是一个向量，然后在一系列的隐层中对它做变换。每个隐层都是由若干的神经元组成，每个神经元都与前一层中的所有神经元连接。但是在一个隐层中，神经元相互独立不进行任何连接。最后的全连接层被称为“输出层”，在分类问题中，它输出的值被看做是不同类别的评分值。

常规神经网络对于大尺寸图像效果不尽人意。例如，图像的尺寸是32*32*3（宽高均为32像素，3个颜色通道），对应的的常规神经网络的第一个隐层中，每一个单独的全连接神经元就有32*32*3=3072个权重。这个数量看起来还可以接受，但是很显然这个全连接的结构不适用于更大尺寸的图像。举例说来，一个尺寸为200*200*3的图像，会让神经元包含200*200 *3=120,000个权重值。而网络中肯定不止一个神经元，那么参数的量就会快速增加显而易见，这种全连接方式效率低下，大量的参数也很快会导致网络过拟合。

卷积神经网络针对输入全部是图像的情况，将结构调整得更加合理，获得了不小的优势。与常规神经网络不同，卷积神经网络的各层中的神经元是3维排列的：宽度、高度和深度（这里的深度指的是激活数据体的第三个维度，而不是整个网络的深度，整个网络的深度指的是网络的层数）。图2-3展示了CNN与常规的神经网络的结构比较。

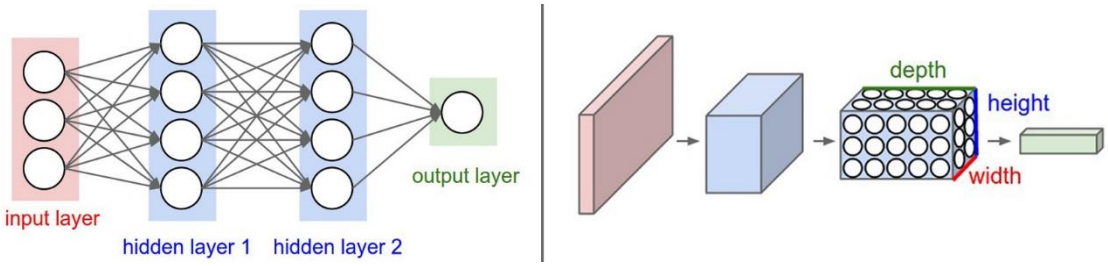

图 2-3. 左：常规神经网络 右：卷积神经网络

一个简单的卷积神经网络是由各种层按照顺序排列组成，网络中的每个层使用一个可以微分的函数将激活数据从一个层传递到另一个层。卷积神经网络主要由三种类型的层构成：卷积层，池化（Pooling）层和全连接层（全连接层和常规神经网络中的一样）。通过将这些层叠加起来，就可以构建一个完整的卷积神经

网络。

卷积层是构建卷积神经网络的核心层，它产生了网络中大部分的计算量。卷积层的参数是有一些可学习的卷积核集合构成的。每个核在空间上（宽度和高度）都比较小，但是深度和输入数据一致。在前向传播的时候，让每个卷积核都在输入数据的宽度和高度上滑动（更精确地说是卷积），然后计算整个卷积核和输入数据任一处的内积。当沿着输入数据的宽度和高度滑过后，会生成一个 2 维的激活图（activation map），激活图给出了在每个空间位置处的反应。直观地来说，网络会让卷积核学习到当它看到某些类型的视觉特征时就激活，具体的视觉特征可能是某些方位上的边界，或者在第一层上某些颜色的斑点，甚至可以是网络更高层上的蜂巢状或者车轮状图案。具体的可以用一个简单的例子来解释卷积层的运算过程。

原图像为$\begin{bmatrix}11&1&7&2&2\\1&3&9&6&7\\7&3&9&6&1\\4&3&2&6&3\\4&1&3&4&5\end{bmatrix}$，卷积核为$\begin{bmatrix}1&5&2\\2&6&3\\7&1&1\end{bmatrix}$，从图像的左上角开始，和卷积核对应元素相乘，然后相加，卷积结果为：

$$11\times2+1\times5+7\times2+1\times2+3\times6+9\times3+7\times7+3\times1+9\times1=138$$

接下来在待卷积图像上向右滑动一列（根据步长的不同，也可滑动不同的长度），继续进行上述操作，待卷积核滑动到图像的右下角时，得到最终的结果为：$\begin{bmatrix}138&154&166\\126&167&133\\104&110&121\end{bmatrix}$经过卷积运算之后，图像尺寸变小了。我们也可以先对图像进行扩充（padding），例如在周边补 0，然后用尺寸扩大后的图像进行卷积，保证卷积结果图像和原图像尺寸相同。

卷积运算显然是一个线性操作，而神经网络要拟合的是非线性的函数，因此和全连接网络类似，我们需要加上激活函数，常用的有 sigmoid 函数，tanh 函数，ReLU 函数等。

以上是单通道图像的卷积，输入是二维数组。实际应用时我们遇到的经常是多通道图像，如 RGB 彩色图像有三个通道，另外由于每一层可以有多个卷积核，产生的输出也是多通道的特征图像，此时对应的卷积核也是多通道的。具体做法是用卷积核的各个通道分别对输入图像的各个通道进行卷积，然后把对应位置处的像素值按照各个通道累加。图 2-4 是多通道卷积的一个例子。

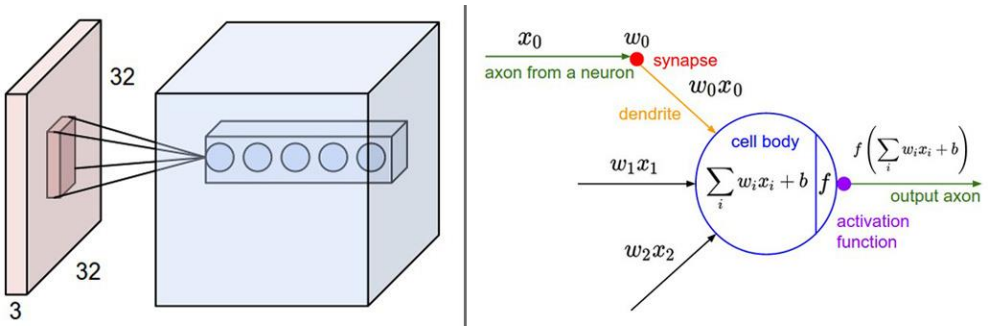

图 2-4. 左：红色的是输入数据体，蓝色的部分是第一个卷积层中的神经元。卷积层中的每个神经元都只是与输入数据体的一个局部在空间上相连，但是与输入数据体的所有深度维度全部相连（所有颜色通道）。在深度方向上有多个神经元，它们都接受输入数据的同一块区域（感受野相同）。右：神经元计算权重和输入的内积，然后进行激活函数运算，它们的连接被限制在一个局部空间。

通过卷积操作，我们完成了对输入向图像的降维和特征抽取，但特征图像的维数还是很高。维数高不仅计算耗时，而且容易导致过拟合。为此引入了下采样技术，即池化操作。**池化层**的做法是对图像的某一个区域用一个值代替，如最大值或平均值。如果采用最大值，叫做 max pooling；如果采用均值，叫做 mean pooling。除了降低图像尺寸之外，下采样带来的另外一个好处是平移、旋转不变性，因为输出值由图像的一片区域计算得到，对于平移和旋转并不敏感。原图像为$\begin{bmatrix}11 & 1 & 7 & 2\\ 1 & 3 & 9 & 6\\ 7 & 3 & 9 & 6\\ 4 & 3 & 2 & 6\end{bmatrix}$，如果进行 max pooling 且池化尺寸为 2*2，步长为 2，那么最大池化后的结果为$\begin{bmatrix}11 & 9\\ 7 & 9\end{bmatrix}$。池化层的具体实现是在进行卷积操作之后对得到的特征图像进行分块，计算这些块内的最大值或平均值，得到池化后的图像。

在**全连接层**中，神经元对于前一层中的所有激活数据是全部连接的，这个常规神经网络中一样。它们的激活可以先用矩阵乘法，再加上偏差。

典型的卷积神经网络就是由上述三种层组成的，在实际应用中，网络的结构会因为使用不同数量和在不同位置使用不同的层而有所差异，图 2-5 展示了一种最简单的用于图像分类的 CNN 模型结构。

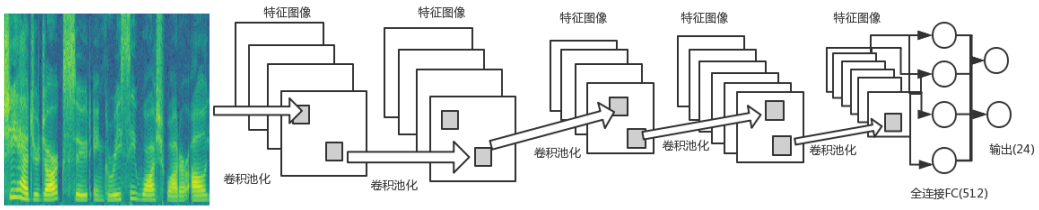

图 2-5 CNN 的网络架构

三、实验过程

3.1 数据预处理

原始数据为两个 txt 文件，分别是训练集：ss100_train.txt 和测试集：ss100_test.txt。训练集（training set）包含 10000 条氨基酸序列以及其对应的二级结构，而测试集（test set）中只包含了 400 条氨基酸序列，用于最终预测并上传答案。每一条氨基酸序列的长度在[50,100]之间。ss100_train.txt 中数据以如下形式储存：氨基酸序列数据以 4 行为一周期，其中第 1、3 行是标签，用来告诉你该氨基酸序列的 ID，以及下一行是氨基酸序列还是其对应的二级结构。4 行中的第 2 行是氨基酸单体序列，第 4 行是其对应的二级结构。如图 3-1 所示。

```
>1A02:J:sequence
MKAERKRMRNRIAASKSRKRKLERIARLEEKVKTLKAQNSELASTANMLREQVAQL
>1A02:J:secstr
CCCCCHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHCC
>1A02:F:sequence
MKRRIRRERNKMAAAKSRNRRRELTDTLQAETDQLEDEKSALQTEIANLLKEKEKL
>1A02:F:secstr
CCCHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHCC
>1A03:A:sequence
MASPLDQAIGLLIGIFHKYSGKEGDKHTLSKKELKELIQKELTIGSKLQDAEIVKLMDDLDRNKDQEVNFQEYITFLGALAMIYNEALKG
>1A03:A:secstr
CCCHHHHHHHHHHHHHHHHCCCCCCCCCEEHHHHHHHHHHHHHCCCCCHHHHHHHHHHHHHCCCCCCEECHHHHHHHHHHHHHHCHHHHHC
```

图 3-1 原始数据格式

第一行表示这个氨基酸序列代号为：1A02:J，sequence 表示它的下一行是 amino acid sequence，即氨基酸序列；第二行是该氨基酸序列的字母缩写表达；第三行里的 secstr 为 Secondary Structure 的缩写，表示下一行是该氨基酸序列所对应的二级结构；第四行就是其二级结构的字母缩写表达。

第二行的氨基酸单体字母缩写一共有 20 种可能，为：[A R N D C Q E G H I L K M F P S T W Y V]。而第四行中的二级结构一共有 3 类——[H，E，C]，其分别代表[α-螺旋，β-折叠以及无规卷曲（random coil）]。第二行与第四行的长度一定是完全相同的，因为每一个氨基酸单体都会在二级结构中找到其对应的分类。

因为这里提供的是 txt 文件，因此需要手动将其转换成适合神经网络读入的格式。根据之前的数据分析，我们将每一个氨基酸序列读成一个二维数组，图 3-2 表示了我们读成的二维数组的形式。

序号\缩写	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	X	UNKONW
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
3																						
4																						
5																						
6																						
7																						
8																						
9																						
10																						

图 3-2 氨基酸链转换成二维数组

因为每个氨基酸链的长度是不等的，但最长不会超过 100，所以，我们将每个氨基酸链读成一个 100*22 的数组，其中前 20 列为氨基酸名字的缩写，在数据处理的过程中发现，给出的文档中，氨基酸链还有第 21 种缩写：X，其表示了该位置可以是任意的单体，故读出的数组的第 21 列为 X，对于长度小于 100 的氨基酸链，剩下没有对于单体的行对应着第 22 列：Unkonw，即该条氨基酸链在这个位置是没有单体的。可以看出，我们读取的数据每一行只会有一列的值为 1，其余列值为 0，是一个稀疏矩阵。相应的，读取的标签格式如图 3-3 所示。

序号/类别	H	E	C	UNKONW
1	0	0	1	0
2	1	0	0	0
3	0	0	1	0
4	0	1	0	0
5				
6				
7				
8				
9				
10				

图 3-2 标签转换成二维数组

数据读取部分使用 Python 编写，先将 txt 文件按行读取存成 list，再对 list 进行具体的处理。将读取到的 10000 个氨基酸序列的 75%作为训练集，25%作为验证集。数据处理代码如图 3-3 所示，读取后的数据格式如图 3-4 所示。

```
169  f = open("./data/ss100_train.txt", "r")   #设置文件对象
170  data = f.readlines()  #直接将文件中按行读到list里
171  f.close()             #关闭文件
172
173  x_train = []
174  y_train = []
175  x_y = []
176  for index, seq in enumerate(data):
177      if index % 2 == 1:
178          x_y.append(seq)  # 将偶数行取出来
179
180  for index, train_data in enumerate(x_y):
181      if index % 2 == 0:
182          x_train.append(train_data)  # 每个氨基酸序列
183      elif index % 2 == 1:
184          y_train.append(train_data)  # 对应的标签
185
186  x_data, y_data = train_to_array(x_train, y_train)  # 将数据处理成二维数组
187  train_X, validation_X, train_y, validation_y = train_test_split(x_data, y_data, test_size=0.25)
```

```
def train_to_array(data_x, data_y):
    X = []
    Y = []
    for one_sequence in data_x:
        one_sample = []
        one_sequence = one_sequence[:-1]
        print(one_sequence)
        for feature in one_sequence:
            rows = np.zeros(22)
            if feature == 'A':  # 根据对应的字母确定该行的某一列为1
                rows[0] = 1
            elif feature == 'R':
                rows[1] = 1
            elif feature == 'N':
                rows[2] = 1
            elif feature == 'D':
                rows[3] = 1
            elif feature == 'C':
                rows[4] = 1
```

```
            elif feature == 'Y':
                rows[18] = 1
            elif feature == 'V':
                rows[19] = 1
            elif feature == 'X':
                rows[20] = 1
            else:
                break
            one_sample.append(rows)

        one_sample = np.array(one_sample)
        print(one_sample.shape)
        non_seq = np.zeros(22)
        non_seq[-1] = 1
        non_seq.shape = (1, 22)
        # 需要填充长度小于100的序列，填充的每一行的最后一列为1
        for i in range(100-one_sample.shape[0]):
            one_sample = np.r_[one_sample, non_seq]
        X.append(one_sample)
    X = np.array(X)
    print(X.shape)

```

```
    for one_sequence in data_y:
        one_sample = []
        one_sequence = one_sequence[:-1]
        print(one_sequence)
        for label in one_sequence:
            rows = np.zeros(4)
            if label == 'H':
                rows[0] = 1
            elif label == 'E':
                rows[1] = 1
            elif label == 'C':
                rows[2] = 1
            else:
                break
            one_sample.append(rows)
        one_sample = np.array(one_sample)
        print(one_sample.shape)
        non_seq = np.zeros(4)
        non_seq[-1] = 1
        non_seq.shape = (1, 4)
        for i in range(100 - one_sample.shape[0]):
            one_sample = np.r_[one_sample, non_seq]
        Y.append(one_sample)
    Y = np.array(Y)
    print(Y.shape)
    return X, Y
```

图 3-3 数据预处理代码

```
E:\Python\Python36-32\python.exe C:/Users/ALEX/PycharmProjects/模式识别课程代码/蛋白质/predic_process.py
(7500, 100, 22)
[[[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.
    0.  0.  0.  0.]
  [ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
    0.  0.  0.  0.]
  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.
    0.  0.  0.  0.]
  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.
    0.  0.  0.  0.]
  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.
    0.  0.  0.  0.]
  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.
    0.  0.  0.  0.]
  [ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
    0.  0.  0.  0.]
  [ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
    0.  0.  0.  0.]
```

```
E:\Python\Python36-32\python.exe C:/Users/ALEX/PycharmProjects/模式识别课程代码/蛋白质/predic_process.py
(7500, 100, 4)
[[[ 0.  0.  1.  0.]
  [ 0.  1.  0.  0.]
  [ 0.  1.  0.  0.]
  [ 0.  1.  0.  0.]
  [ 0.  0.  1.  0.]
  [ 0.  0.  1.  0.]
  [ 0.  0.  1.  0.]
  [ 0.  0.  1.  0.]
  [ 0.  0.  1.  0.]
  [ 0.  1.  0.  0.]
  [ 0.  1.  0.  0.]
  [ 0.  1.  0.  0.]
  [ 0.  1.  0.  0.]
  [ 0.  1.  0.  0.]
  [ 0.  1.  0.  0.]
  [ 0.  0.  1.  0.]
```

图 3-4 读取好的数据

3.2 分类网络的结构设计与程序实现

3. 2. 1. BP 神经网络

根据读取数据的格式，BP 神经网络的输入数据需要向量的形式，而我们读取好的数据为二维的数组，所以对于每一个 100×22 的氨基酸链样本，先将其拉长成 2200×1 的向量，对应的 100×4 的标签拉长成 400×1 的向量。因此，设计的神经网络的输入神经元个数为 2200，隐藏层为 1 层，神经元个数设定为 3000，输出层的神经元个数为 400，隐藏层与输出层的激活函数都为 sigmoid 函数。搭建的 BP 神经网络如图 3-5 所示。

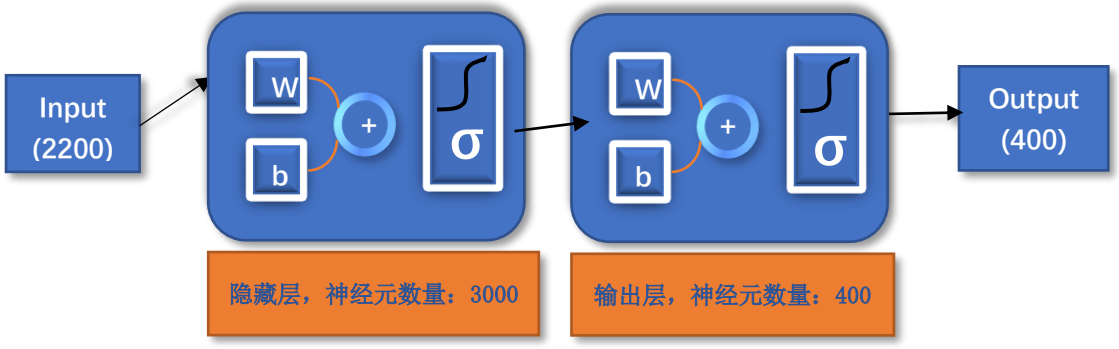

图 3-5 BP_NN 结构图

在本次实验中，还使用了附加动量因子的梯度下降学习函数来自适应调节网络的学习速率。传统的梯度下降法在修正权值时，只是按照 k 时刻的负梯度方向修正，并没有考虑到以前积累的经验，即以前时刻的梯度方向，从而常常使学习过程发生振荡，收敛缓慢。使用动量法的权值修正可以表示为：

$$w_{ji}(k+1)=w_{ji}(k)+\eta[(1-\alpha)D(k)+\alpha D(k-1)]$$

其中，D(k)表示 k 时刻的负梯度，α 为动量因子，范围在[0,1]之间，可以看出，当 α =0 时，权值的修正只与当前的梯度有关，当当 α =1 时，权值的修正完全取决于上一次循环的梯度。这种方法所加入的动量项实质上相当于阻尼项，它减少了学习过程的振荡趋势，从而改善了网络的收敛性。在 Matlab 中，使用的是 'traingdx' 来自适应调整学习速率。

数据的向量化处理使用 python 编写，BP 网络的实现部分使用 Matlab 编写。在数据分析部分已经说过，由于 CNN 可以将一条氨基酸序列看成一副图片，其输入数据的结构可以为 N*100*22 的三维数组，但这样的数据类型无法作为 BP 神经网络的输入数据。核心思想是将该氨基酸形成的图片按行拉成一条向量，由此作为 BP 神经网络的输入。修改数据格式的 python 代码如下：

```
from scipy import io
import numpy as np
mat = np.load('C:/AMD/train_X.npy')
print mat.shape
trainvector_X=np.zeros((7500,2200))
# 若为标签数据，则 2200 修改为 400，7500 为条目数
for i in range(0,len(mat)):
    trainvector_X[i,:]=np.reshape(mat[i],2200, 'C')
# 按行拉成向量，若为标签数据，2200 改为 400 即可
io.savemat('train_x.mat', {'train_x': trainvector_X})
# 原来的 7500*100*22 由此被拉成 7500*2200 的行向量，每一行代表一条氨基酸序列
```

上述代码得到了 7500 条被拉长成向量的测试集数据，剩余的 2500 条验证集数据的修改也采用相同的操作。

BP 神经网络的构造与结果分析代码如下：

```
clc;  clear all;  load datax;
%特征归一化
[input,xmin,xmax] = premnmx(trainvectorx');
%构造输出矩阵
output=trainvectory;
%隐藏层和输出层使用 Sigmoid 传输函数
%使用  'traingdx'  自适应调整学习速率
% learn 属性  'learngdm'  附加动量因子的梯度下降学习函数
net = newff( minmax(input) , [3000 400], {'tansig' 'tansig'} , 'traingdx' , 'learngdm') ;
%设置训练参数
net.trainparam.show = 1000;%每间隔步显示一次训练结果
net.trainparam.epochs = 15000 ;%允许最大训练步数 15000 步
net.trainparam.goal = 0.01 ;%训练目标最小误差 0.01
net.trainParam.lr = 0.05 ;%学习速率 0.05
%开始训练
net = train( net, input , output' ) ;
%读取测试数据，测试数据归一化并输入到网络得到预测类别输出
[input,amin,xmax] = premnmx( testvectorx')   ;
testInput = tramnmx( testvectorx' , amin,xmax) ;
Y = sim( net , testInput ) ;
%统计识别正确率
[s1 , s2] = size( Y ) ;
[d1 , d2] = size(testvectory');
testvectory=testvectory';
hitNum = 0 ;%用于记录预测正确的个数
point=1;jk=1;
for i = 1 : s2%按列遍历输出的矩阵，每 4 个进行一次截断，用于生成最终预测序列
  while point<400
  [m , Index] = max( Y(point :point+3 , i ) ) ;
  result(jk)=Index;
  jk=jk+1;
  point=point+4;
  end
if point>400
        point=1
  end
end
point=1;jk=1;
```

```
for i = 1 : d2%按列遍历标签矩阵，每 4 个进行一次截断，用于对比生成准确率
  while point<400
  [m , Index2] = max(testvectory(point :point+3 ,i)) ;
  Y_test(jk)=Index2;
  jk=jk+1;
  point=point+4;
  end
if point>400
        point=1
  end
end
for i=1:length(Y_test)
       if Y_test(i)==result(i);
             hitNum=hitNum + 1 ;
       end
end
sprintf('识别率是 %.3f%%',100 * hitNum / (d1*d2/4) )
```

使用均方误差作为网络的损失函数，一共迭代训练 15000 次，最终在验证集上的准确率为 80.641%，在网站所给的测试集上的准确率为 73.831%，具体的分析见 3.3 的结果对比分析。

3. 2. 2. 卷积神经网络(CNN)

前面介绍了 CNN 在图像分类中的应用，那么我们可以将一条氨基酸序列看成一副图片，其输入数据的结构可以为 N*100*22 的三维数组，只是输出的分类不再是对整个图像的一个类别，而是对图像的每一行进行分类（有点类似于图像分割，对图像的每个像素点进行分类），即输出的类别信息的是 N*100*4 的数组。

传统的多分类 CNN 最后的全连接层会接一个 softmax 分类器输出分类信息，我们设计了一个含有 3 个卷积层的网络，卷积层后接一个 ReLU 激活层，但是并没有加入池化层，原因是池化会使得数据的尺寸变小，而我们最终还是需要对数据的每一行进行分类，所以没有加入池化层。实际上，可以通过池化以后再进行上采样或者反卷积恢复成原数据的尺寸，但我们的数据终究是氨基酸序列转换成的稀疏矩阵，说到底还不是图像，先池化再上采样对原数据会造成巨大程度上的破坏，对分类结果是毁灭性的打击。

每个卷积层的卷积核尺寸为 11*11，之所以将卷积核的尺寸设定为 11，是因为在分析数据的时候我们发现，在很多氨基酸链中经常会出现连续 11 个单体的结构是相同的情况。同时卷积层的卷积核通道数为 40，为了保证能够得到尽可能多的有效信息。

网络使用交叉熵函数作为 loss，并使用 Adam 算法最小化 loss。Adam 算法根据损失函数对每个参数的梯度的一阶矩估计和二阶矩估计动态调整针对于每

个参数的学习速率。TensorFlow 提供的 tf.train.AdamOptimizer 可控制学习速度。Adam 也是基于梯度下降的方法，但是每次迭代参数的学习步长都有一个确定的范围，不会因为很大的梯度导致很大的学习步长，参数的值比较稳定。令 M(t)为梯度的第一时刻平均值（一阶矩），V(t)为梯度的第二时刻非中心方差值（二阶矩）。其计算公式为：

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t},$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}.$$

参数更新的公式为：

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t.$$

其中，β1 设为 0.9，β2 设为 0.9999，ϵ 设为 10^{-8}。与其他自适应学习率算法相比，其收敛速度更快，学习效果更为有效，而且可以纠正其他优化技术中存在的问题，如学习率消失、收敛过慢或是高方差的参数更新导致损失函数波动较大等问题。

本实验使用 python 的深度学习库 TensorFlow 完成 CNN 的框架的搭建，网络结构的主要代码如下：

```
#  定义网络结构

# Input and output
X = tf.placeholder(tf.float32, [None, 100, 22], name="X")
y = tf.placeholder(tf.float32, [None, 100, 4], name='y')

# 模型
filter_width = 11  # 卷积核尺寸
filter_input_size = 22  # 输入数据是22维的
filter_channels = 40    # 卷积核通道数
final_filter_channels = 4   # 输出通道数应为对应的分类种类
# 定义每一层卷积的卷积核参数
W1 = tf.get_variable(name="W1", shape=[filter_width, filter_input_size, filter_channels], initializer=tf.contrib.layers.xavier_initiali
b1 = tf.get_variable("b1",[filter_channels], initializer=tf.random_normal_initializer())
W2 = tf.get_variable(name="W2", shape=[filter_width, filter_input_size, filter_channels], initializer=tf.contrib.layers.xavier_initiali
b2 = tf.get_variable("b2",[filter_channels], initializer=tf.random_normal_initializer())
W3 = tf.get_variable(name="W3", shape=[filter_width, filter_input_size, final_filter_channels], initializer=tf.contrib.layers.xavier_in
b3 = tf.get_variable("b3",[final_filter_channels], initializer=tf.random_normal_initializer())
```

```
# 卷积层1
conv1 = tf.nn.conv1d(value=X, filters=W1, stride=1, padding='SAME')
a1 = tf.nn.bias_add(conv1, b1)
z1 = tf.nn.relu(a1)
# 卷积层2
conv2 = tf.nn.conv1d(value=X, filters=W2, stride=1, padding='SAME')
a2 = tf.nn.bias_add(conv2, b2)
z2 = tf.nn.relu(a2)
# 卷积层3
conv3 = tf.nn.conv1d(value=X, filters=W3, stride=1, padding='SAME')
a3 = tf.nn.bias_add(conv3, b3)
z3 = tf.nn.relu(a3)
# 通过softmax输出类别
y_ = tf.nn.softmax(z3)

mask = tf.not_equal(tf.argmax(y, 2), 3)

y_masked = tf.boolean_mask(y, mask)
z3_masked = tf.boolean_mask(z3, mask)
y__masked = tf.boolean_mask(y_, mask)

# loss(交叉熵函数)
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(labels=y_masked, logits=z3_masked))
```

```
# 学习率
learning_rate = tf.placeholder(tf.float32)

# Adam优化算法进行梯度下降最小化loss
# optimizer = tf.train.GradientDescentOptimizer(learning_rate = learning_rate).minimize(loss)
optimizer = tf.train.AdamOptimizer(learning_rate = learning_rate).minimize(loss)

# 输出的预测值与准确率的定义
prediction = tf.argmax(y__masked, 1)
accuracy = tf.reduce_mean(tf.cast(tf.equal(prediction, tf.argmax(y_masked, 1)), tf.float32))

# 初始化条件
init = tf.global_variables_initializer()

n_parameters = np.sum([np.prod(v.get_shape().as_list()) for v in tf.trainable_variables()])
print("Number of parameters:", n_parameters)
```

网络的训练过程中，batch_size 设置为 100，即每次输入 100*100*22 的数据进行训练，整个网络迭代训练 200 次，最终在 test 数据上的准确率为 66.134% 具体的分析见 3.3 的结果对比分析。训练代码如下：

```
# Start as session
with tf.Session() as session:

    batch_size = 100
    # 先对模型的参数初始化
    session.run(init)

    for epoch in range(100):
        print("Epoch:", epoch)
        for b in range(0, X_train.shape[0], batch_size):
            _, loss_value = session.run([optimizer, loss], feed_dict={X: X_train[b:b+batch_size],
                                                                      y: y_train[b:b+batch_size],
                                                                      learning_rate: 0.0001})
            if b % 1000 == 0:
                validation_accuracy = session.run(accuracy, feed_dict={X: X_validation, y: y_validation})
                print("loss[b=%04d] = %f, val_acc = %f" % (b, loss_value, validation_accuracy))

    print("Optimization done")

    # 计算准确率
    train_accuracy_value, pred_train = session.run([accuracy, prediction], feed_dict={X: X_train, y: y_train})
    print("Train accuracy:", train_accuracy_value)

    # Calculate test accuracy
    val_accuracy_value, pred_val = session.run([accuracy, prediction], feed_dict={X: X_validation, y: y_validation})
    print("Val accuracy:", val_accuracy_value)
```

3.3 结果分析对比

当网络训练完后将网站提供的 test 数据进行和之前处理 train 数据一样的操作，并将输出存储在提交的 csv 文件中。这里需要注意的是，由于我们在之前对所有的氨基酸链的长度都统一成了 100，那么在进行测试的时候，也需要将测试

集的 400 条氨基酸长度统一，那么得到的分类数据的长度会是 400*100=40000，所以在最后输出到 csv 文件的过程中，需要将我们添加的那一部分长度所对应的结果去掉（即删除每个测试氨基酸链中不存在的那部分序列的分类结果），最终得到的是与测试集长度相同的类别信息 32508。这部分的处理代码如下：

```
prediction = np.load('./prediction.npy')
prediction = prediction.reshape(400, 100)

f = open("./data/ss100_test.txt", "r")   # 读取test数据
data = f.readlines()  # 直接将文件中按行读到list里
f.close()             # 关闭文件
x_test = []
for index, seq in enumerate(data):
    if index % 2 == 1:
        x_test.append(seq)

# x_test = np.array(x_test)
re_predict = []
for index, one_seq in enumerate(x_test):
    one_seq = one_seq[:-1]
    seq_len = len(one_seq)  # 根据每条氨基酸链的长度将裁剪预测类别数据
    pre_list = list(prediction[index])
    del pre_list[seq_len:100]  # 删除不存在的单体对应的类别
    re_predict.append(pre_list)

re_predict = list(_flatten(re_predict))
# 存储预测数据
data = pd.DataFrame({'class': re_predict})
data.to_csv('submit_2.csv', index=False)
```

提交的 submission.csv 文件，将使用[准确率(accuracy)]作为最后评判标准。准确率(accuracy)是用来衡量算法预测结果的准确程度，具体指测试集中算法预测正确的数量占总数的比例。即：

$$Accuracy = \frac{Number of Correct Predictions}{Number of Total Predictions}$$

本实验中，将二级结构分为 3 类（α-螺旋，β-折叠以及无规卷曲（random coil））。因此每一个氨基酸的二级结构预测值都应当属于这 3 类之一。在统计准确率时，这里我们将比较每一条氨基酸序列中，每一个氨基酸单体所对应的二级结构预测值和真实值。即，在一条长度为 k 的氨基酸序列中，它包含 k 个氨基酸单体，其 number of prediction = k。因此，Number of Correct Predictions 为预测正确的氨基酸单体的总个数，而 Number of Total Predictions 为所有的氨基酸序列的长度总和，accuracy_score 即为此二者之比。

3. 3. 1. BP_NN 结果

BP 网络在迭代训练了 15000 次以后再验证集上的准确率达到了 80.641%，此外，网络训练的 loss 图以及最终误差与梯度实验图如下所示：

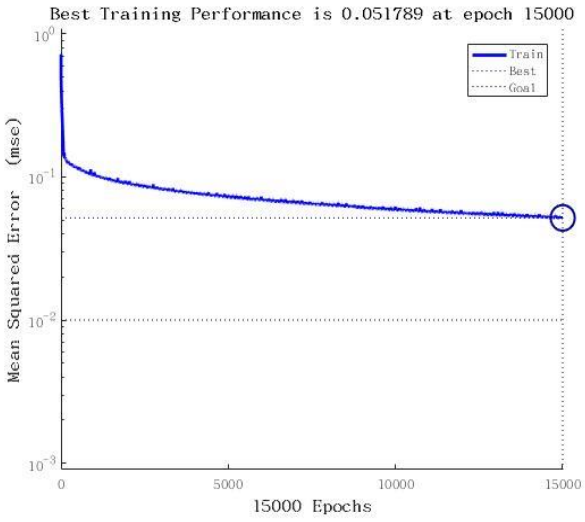

图 3-6 BP-NN loss 曲线图

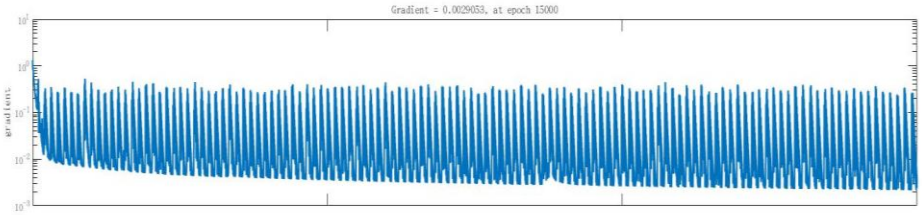

图 3-7 BP-NN 梯度变化曲线图

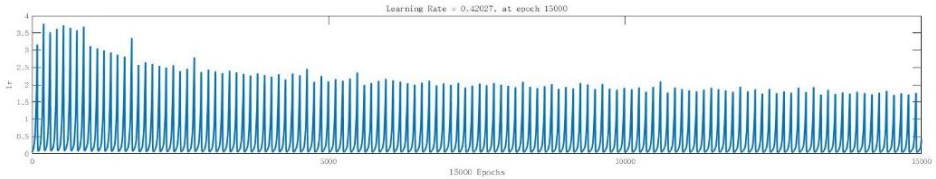

图 3-8 BP-NN 学习率变化曲线图

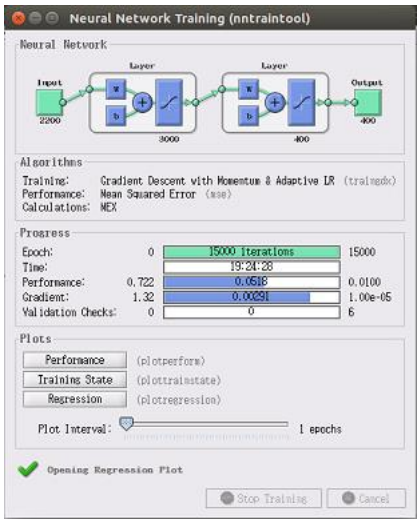

图 3-9 BP-NN 网络训练结果图

从图中可以看到，在迭代 15000 次后均方误差为 0.051789，仍然没有达到设

定的目标误差值 0.01，尽管增加迭代次数，loss 下降程度十分有限甚至几乎不下降，但与此同时梯度在第 15000 次迭代后为 0.0029053 已非常小。值得一提的是网络的训练时间很长，15000 次迭代大约耗时 20 小时。

将训练好的网络应用于最终测试集，并进行最后的数据尾处理操作，并上传至 LintCode 网站，得到的在测试数据上的准确率为 73.831%。

3. 3. 2. CNN 结果

CNN 经过 100 次迭代后，在验证集上的准确率为 65.957%，网络的 loss 曲线和准确率曲线如下图所示。

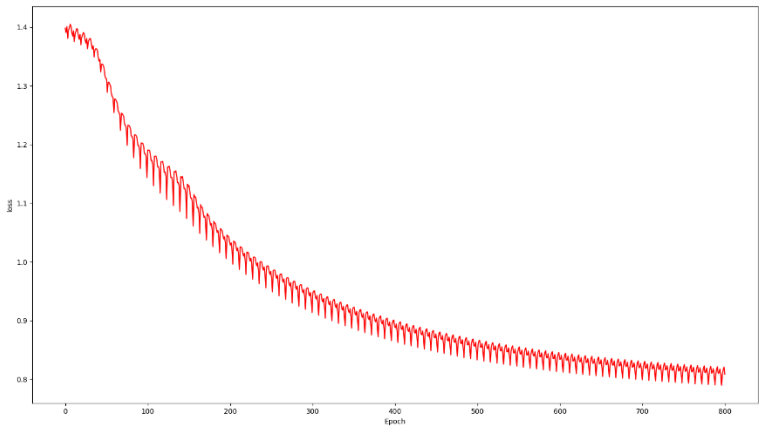

图 3-10 CNN loss 下降曲线

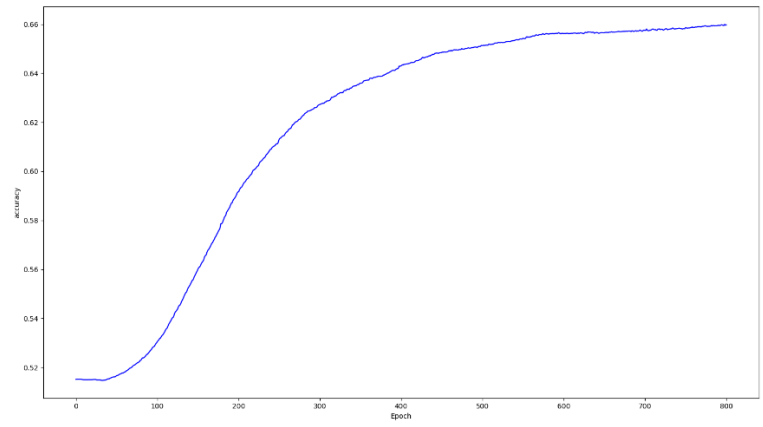

图 3-11 CNN 准确率变化曲线

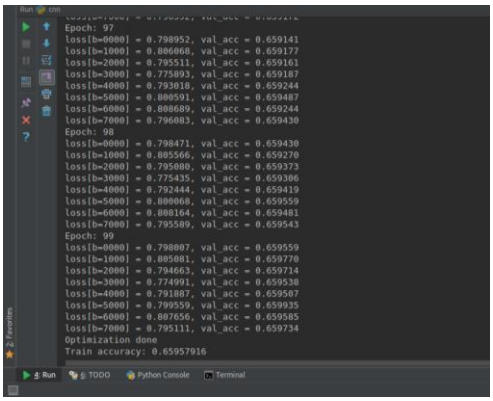

图 3-12 CNN 训练结果

将训练好的网络应用于最终测试集，并进行最后的数据尾处理操作，并上传至 LintCode 网站，得到的在测试数据上的准确率为 66.134 %。这个准确率低于 BP_NN 的结果，说明了有时候复杂的网络也不一定是解决问题的最好的办法，这个结果也可能说明了将氨基酸链看做成一个二维 2 值化图像用 CNN 去分类可能还不够完善，而相反，BP 神经网络具有较强的非线性映射能力，这使得其特别适合于求解内部机制复杂的问题，例如本次实验要做的蛋白质内部的氨基酸之间就有着复杂的相互关系从而使蛋白质有着不同的二级结构。这也解释了为什么在 BP_NN 上能取得更好的结果。但是本次实验的 BP_NN 的训练十分耗时，而 CNN 在进行的 100 轮迭代只用了不到一分钟的时间。但由于时间关系，我们没有进一步优化 BP_NN 的训练时间和对 CNN 结构进行优化。

3. 3. 3. 最终结果排名

将最终预测结果上传到网站上并得到在测试集上的准确率为 0.73831，排名第二，其中第一名的成绩存在质疑（为 100%），故我们的结果目前来看在网站上的排名是第一的，如图所示。

排名

刷新

#	队伍名称	分数	成员	提交次数	最后提交时间
1	jiangzuo	1.00000		12	24 天前
2	QinXiangxiang	0.73831		7	8 天前
3	serenetu	0.73803		7	24 天前
4	lanxx	0.47022		6	1 个月前

四、小组分工

程序设计及编写：秦祥翔、马磊、付锡欣

程序调试及结果分析：秦祥翔、马磊、付锡欣

资料查找及报告撰写：汤强、杨昊晨

五、总结

本次实验使用 BP 神经网络和卷积神经网络模型分别进行了蛋白质的二级结构预测，其中在 BP 神经网络上取得的结果更好，准确率为 73.831%，这说明有时候简单的才是最好的。因为本次实验所给出的数据非常贴合实际，且与之前我们所见到的数据格式有点不同，所以在数据预处理时我们组经过了很长时间的讨论，最终确定了将每一条氨基酸链作为一个二维数组来训练。

在 BP 神经网络的设计部分，由于之前讨论的数据格式不太适用于 BP 神经网络的输入格式，所以我们又将读取成的二维数组拉长成一维的向量用作 BP 神经网络的训练，这种方法使得 BP_NN 的输入向量维度很大，也导致了网络的训练时间很长，故接下来如果有机会，我们将进一步的考虑如何去优化数据的格式以降低 BP_NN 的训练时间。

在 CNN 的设计过程中，由于一般的 CNN 中都会有池化层的存在，在一开始，我们也将池化层加入到网络结构中，并且为了保证输入输出的尺寸一致，额外添加了上采样部分，但是这么做导致网络的分类结果很差，准确率不到 10%，经过仔细的分析，我们发现在进行池化并上采样以后，原始数据的分布结构被完全破坏了，这与我们输入的是一个稀疏矩阵而不是真正的图像有关，故在最后我们将池化和上采样层去掉，只使用了 3 层卷积层和 ReLU 激活层来搭建我们的 CNN 模型，同时我们注意到很多条氨基酸的连续 11 个或以上的单体都会有相同的二级结构，为了保证不破坏这些单体间可能存在的联系，我们将卷积核的大小设置为 11。但是最后的分类结果却没有 BP_NN 的高，虽然在训练时间上远优于 BP_NN，故接下来如果有机会，我们将进一步的考虑更加适合的结构来获得更好的结果。

最后，感谢赵老师的教学帮助，并且在作业过程中给了我们大家很多指导。因为上了您的课程，使我们对模式识别产生了浓厚的兴趣，虽然课程已经结束，但我们也将在模式识别领域继续探索下去。