第5章

AlphaFold与蛋白质结构预测

长期以来，研究人员使用诸如X射线晶体衍射、核磁共振和冷冻电镜等实验方法确定蛋白质的结构，每种方法都需要大量的实验、错误试探和纠错，每一个蛋白质结构的成功解析，背后都是无数艰辛的科学探索与实验支撑。近年来，随着AI技术的崛起，借助AI方法，根据氨基酸残基序列直接计算预测蛋白质结构成为计算生物学的热点领域。如图5.1所示，直接根据残基序列推断蛋白质三维结构的方法，相当于用超强的计算过程取代了科学家们繁重的实验过程，是一条极富科学前景的探索之路。

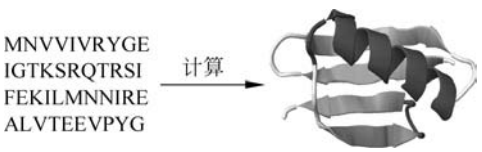

图5.1　基于残基序列直接预测蛋白质结构

5.1　什么是AlphaFold

蛋白质结构预测技术的关键评估（Critical Assessment of protein Structure Prediction，CASP），是基于残基序列直接预测蛋白质三维结构的全球科学盛会，目标是对来自世界各地的科学家团队提交的蛋白质结构预测模型的关键指标做出科学评估。

参加CASP比赛的团队一般由科学家、工程师和人工智能专家等组成，CASP每两年举办一次。从1994年的第1届CASP1到2018年的第13届CASP13，从未间断。在撰写本教材的过程中，第14届CASP14已经于2020年4月拉开序幕。有人将两年一度的CASP誉为结构生物学领域的“奥林匹克”。

Google旗下DeepMind公司的一个人工智能研究团队参加了CASP13，团队名称为A7D，其开发的蛋白质结构预测系统称为AlphaFold（又称A7D系统）。相关研究成果发表在*Nature*上（Senior，Evans，et al.，2020），深度学习模型的代码开源在GitHub

(https://github.com/deepmind/deepmind-research/tree/master/alphafold_casp13)。AlphaFold 系统的逻辑结构如图 5.2 所示。

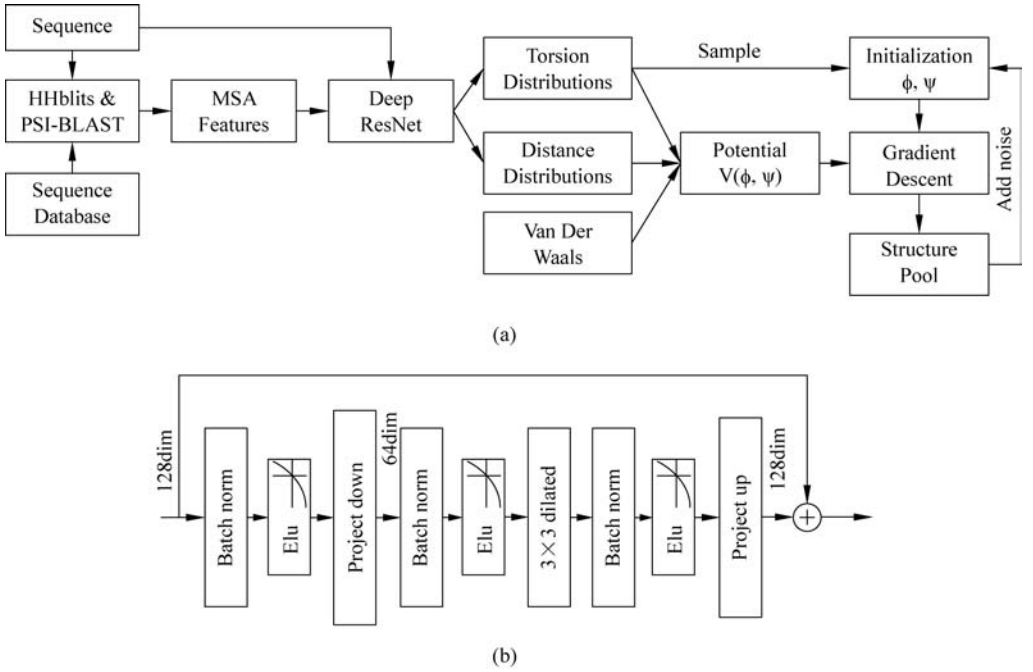

图 5.2 AlphaFold 系统的逻辑结构

图 5.2 源自 Andrew W. Senior 等人发表于 *Nature* 的论文。图 5.2(a)表示 AlphaFold 的系统逻辑,包括以下几个关键步骤。

(1) 特征提取阶段。使用 HHblits 和 BLAST 软件,通过对已有的蛋白质数据库的搜索、构建和计算,完成给定残基序列的多序列比对对齐(Multiple Sequence Alignment,MSA)分析,实现 MSA 特征提取。

(2) 模型训练与结构预测。定义了一个包含 220 个残差块的 ResNet 深度卷积神经网络,基于序列和 MSA 特征,训练和预测蛋白质残基对之间的距离,以及每个残基的二面角 Phi(ϕ)和 Psi(ψ)。

单个残差块的结构如图 5.2(b)所示。图 5.2(b)是一个 ResNet 的残差结构,输入层的通道数为 128,残差块包含 3 个卷积层,第一个卷积层采用了 1×1 的卷积降维到 64 通道,第 2 个卷积层采用 3×3 的卷积,第 3 个卷积层采用 1×1 的卷积上采样到 128 通道,卷积层采用 Batch norm 方法和 Elu 激励函数。

(3) 二次特征提取与整合。整合深度学习得到的二面角、残基对距离,结合原子的范德华半径,构建生成蛋白质结构的特征集。

(4) 生成蛋白质结构。根据模型预测的距离分布、二面角 Phi(ϕ)和 Psi(ψ)的分布以及范德华半径构建蛋白质的候选潜在结构,通过一个深度卷积网络估算候选结构的准确性,最终确定蛋白质结构。

AlphaFold 在全球约一百个研究团队提交的数万个模型中脱颖而出,取得综合排名第一的成绩,取得了用计算方法预测蛋白质结构前所未有的进步。图 5.3 所示为 AlphaFold

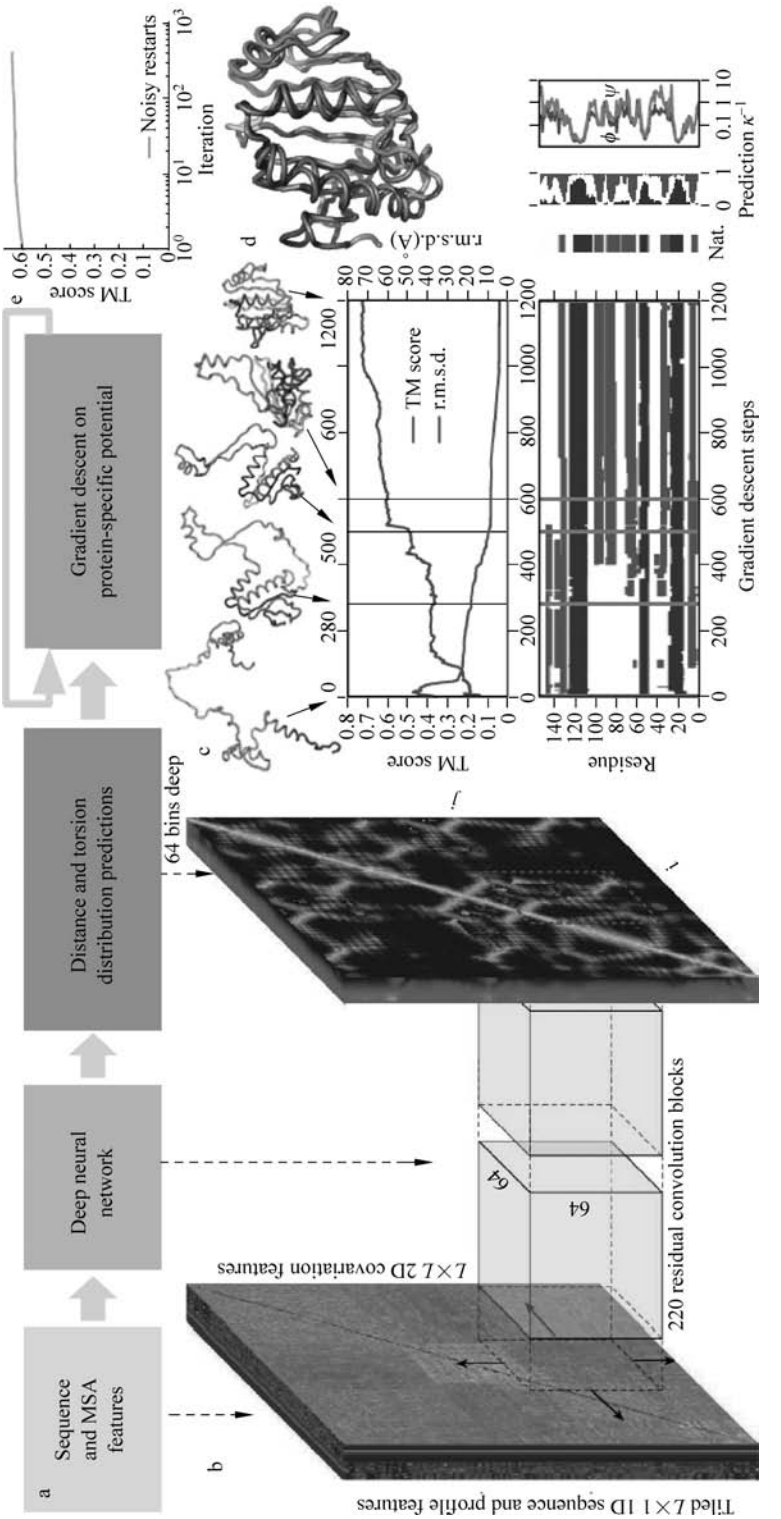

图 5.3 AlphaFold 的蛋白质结构建模过程解析

对 CASP13 中标签为 T0986s2 的蛋白质结构建模过程的解析。

图 5.3 源自 Andrew W. Senior 等人发表于 *Nature* 的论文。T0986s2 的序列长度为 $L=155$，PDB 数据库中的蛋白质 ID 为 6N9V。

图 5.3 中 a 表示结构预测步骤，依次是特征提取，深度神经网络进行距离和二面角预测，二次特征提取，深度神经网络对模型结构进行评估与筛选。

图 5.3 中 b 表示 ResNet 神经网络根据 MSA 特征和序列特征预测得到 $L\times L$ 的距离分布图，预测过程以 64×64 作为残基窗口进行累积预测。

图 5.3 中 c 显示的是经过 1200 步梯度下降迭代，TM 分数和均方根误差(r. m. s. d.)的变化图形，TM 分数逐步提升，均方根误差逐步下降，并且给出了第 0 步、第 280 步、第 500 步、第 600 步、第 1200 步预测的蛋白质二级结构形状。

TM 是 CASP 采用的蛋白质结构评分方法，表示预测的结构总体上与真实结构的匹配程度，取值范围为[0，1]。AlphaFold 在蛋白质结构自由建模领域遥遥领先于其他方法。

图 5.3 中 d 是对模型的预测结构与真实结构的匹配程度的直观展示，灰色表示预测的结构，彩色表示真实的结构。

图 5.3 中 e 表示整个测试集(377 个蛋白质样本)的 TM 平均分数随着迭代次数的变化趋势。

5.2 肽键、多肽与肽链

蛋白质几乎参与了所有的生命活动过程，是生命活动的物质基础，是构成生物体最基本的功能物质。蛋白质的主要组成元素占比如表 5.1 所示。

表 5.1 蛋白质组成元素占比

元素名称	占　比	元素名称	占　比
碳	50%～55%	氮	16%
氢	6.5%～7.3%	硫	0%～3%
氧	19%～24%	其他微量	剩余占比

蛋白质的基本构成单位为氨基酸，常见氨基酸有 20 种。用酸、碱或蛋白酶可以分别将蛋白质水解，生成游离的氨基酸。组成蛋白质的 20 种常见氨基酸中除脯氨酸外，均为 α-氨基酸，其基本结构如图 5.4 所示，除了侧链 R 以外，其他部分是固定不变的。

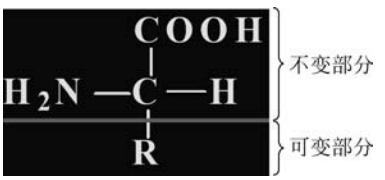

图 5.4 α-氨基酸的基本结构

肽是由一个氨基酸的羧基(α-COOH)与另一个氨基酸的氨基(α-NH_2)脱水缩合而成的化合物，氨基酸间脱水后生成的共价键称为肽键(Piptide bond)，其中的氨基酸单位称为氨基酸残基。由两个氨基酸缩合而成的肽称为二肽，少于 10 个氨基酸残基的肽称为寡肽，多于 10 个氨基酸残基的肽称为多肽。肽链上的各个侧链由不同氨基酸的 R 侧链构成。如图 5.5 所示，两个氨基酸脱水生成二肽。

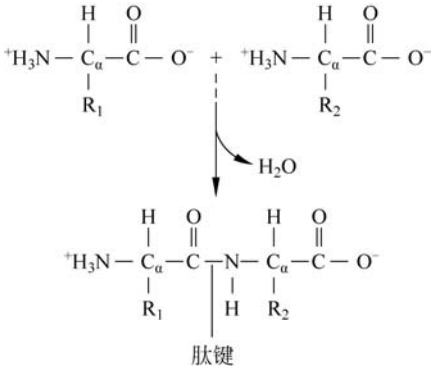

图 5.5　两个氨基酸组成二肽

由多个氨基酸通过肽键相互连接形成的链状结构称为肽链。通常在肽链的一端含有一个游离的 α-氨基，称为氨基端或 N-端；在另一端含有一个游离的 α-羧基，称为羧基端或 C-端。如图 5.6 所示，左端是 N-端，右端是 C-端。

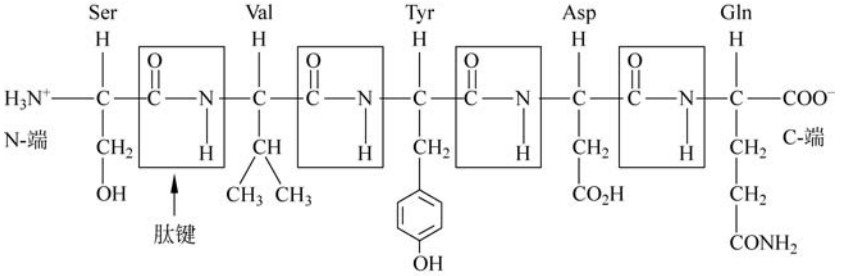

图 5.6　肽链的基本结构

肽链中的氨基酸残基按一定的顺序排列，这种排列顺序称为氨基酸顺序。

氨基酸顺序是以 N-端残基为起点、以 C-端残基为终点所形成的排列顺序。如图 5.6 所示的五肽按照残基顺序可表示为 Ser-Val-Tyr-Asp-Gln 或者 SVTAG。

5.3　蛋白质的四级结构

每一种天然蛋白质都有自己特有的空间结构，这种空间结构称为蛋白质的(天然)构象。蛋白质的结构层次分为四级，如图 5.7 所示。

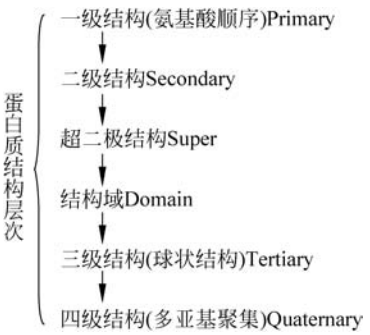

图 5.7　蛋白质的结构层次

如图 5.8 所示为蛋白质四级结构示意图。

蛋白质的一级结构是指氨基酸残基的排列顺序，这是蛋白质生物功能的基础。一级结构(残基序列)中含有形成高级结构全部必需的信息，一级结构决定高级结构及其功能。所以，根据一级机构(残基序列)可以推断蛋白质的高级空间结构，根据高级空间结构可以进一步推断蛋白质的功能性质。

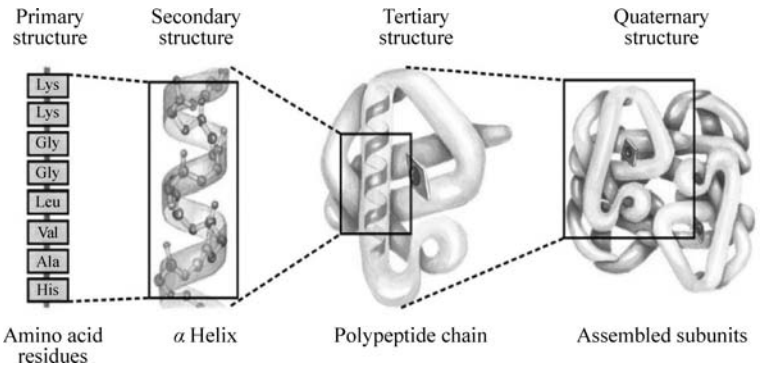

图 5.8 蛋白质四级结构示意图

蛋白质的二级结构是多肽链中各个肽段借助氢键形成有规则的构象，主要包括 α-螺旋、β-折叠、β-转角、无规卷曲等。

超二级结构指蛋白质中相邻的二级结构单位（α-螺旋或 β-折叠或 β-转角）组合在一起，形成有规则的、在空间上能够辨认的二级结构组合体。超二级结构的基本类型有：αα、βαβ、ββββ。

结构域是多肽链在二级结构或超二级结构基础上形成三级结构的局部折叠区。结构域是蛋白质的独立折叠单位，一般由 100～200 个氨基酸残基构成。

蛋白质的三级结构是多肽链借助各种非共价键的作用力，通过弯曲、折叠，形成具有一定走向的紧密球状构象。球状构象的表面积最低，使蛋白质与周围环境的相互作用力最小。

蛋白质的四级结构是寡聚蛋白中各亚基之间在空间上的相互关系和结合方式。

决定蛋白质二级构象的主要因素是二面角和残基之间的距离（接触）。目前，以 AlphaFold 为代表的机器学习软件，主要是基于二面角和距离（接触）对蛋白质的空间结构做出预测。

5.4 数据集

AlphaFold 开源在 GitHub 上的模型，数据集采用已经处理好的 TFRecords 格式，模型的计算需求较大，初学者的入门门槛较高，因此，本章案例以作者 Eric Alcaide 发布的 MinFold（https://github.com/EricAlcaide/MiniFold）开源项目为起点进行探索，并做了局部修改和优化。

MiniFold 可以看作是 AlphaFold 的一个简易版本，特别是在距离预测与角度预测方面，与 AlphaFold 的理念与方法保持一致，采用的网络结构也是源自 AlphaFold，在此向 Eric Alcaide 的杰出工作表示致敬！

数据集下载页面（https://github.com/aqlaboratory/proteinnet）包含 CASP7～CASP12 的比赛数据集。每个数据集包含 Text-based 与 TFRecords 两种格式，本章案例选择的是 CASP7 中的 Text-based 数据集。

CASP7 数据集的下载文件为 casp7.tar.gz 压缩包，压缩包大小为 3.18GB。下载完成后，解压到 chapter5\dataset 目录下，相关文件描述如表 5.2 所示。

表 5.2 CASP7 蛋白质结构数据集

文 件 名	数 据 规 模	大 小	功 能
training_30	10 333 个蛋白质样本	675MB	包含训练集 30%的数据
training_50	13 024 个蛋白质样本	898MB	包含训练集 50%的数据
training_70	15 207 个蛋白质样本	1.03GB	包含训练集 70%的数据
training_90	17 611 个蛋白质样本	1.2GB	包含训练集 90%的数据
training_95	17 938 个蛋白质样本	1.23GB	包含训练集 95%的数据
training_100	34 557 个蛋白质样本	2.42GB	包含训练集 100%的数据
validation	224 个蛋白质样本	15.2MB	验证集数据
testing	93 个蛋白质样本	6.5MB	测试集数据

以 training_30 文件为例，文件结构组织如表 5.3 所示。

表 5.3 training_30 文件的组织结构

字 段 域 名	行 数	功 能
[ID]	单独占 1 行	蛋白质的 ID 编号
[PRIMARY]	单独占 1 行	氨基酸残基序列(单字母表示)
[EVOLUTIONARY]	单独占 21 行	残基序列的 PSSM 评分矩阵
[TERTIARY]	单独占 3 行	主链上 N—Calpha—C 原子的 x,y,z 坐标
[MASK]	单独占 1 行	残基掩码('+'：有实验数据，'-'：数据缺失)

5.5 筛选蛋白质序列

从 training_30 数据集中筛选出残基序列长度不超过 136 的蛋白质作为训练集。之所以选择长度为 136 的残基序列，一方面是考虑计算力的问题，另一方面是因为后面计算二面角时采用了宽度为 34 的剪辑窗口，136 是 34 的整数倍。

为了组织项目文件结构，按照如下步骤创建子目录。

(1) 在 chapter5 目录下新建两个子目录：preprocessing 和 models。preprocessing 用于数据预处理和特征提取，models 用于存放距离预测模型和角度预测模型。

(2) 在 models 目录下创建 angles 和 distance 两个子目录，分别用于存放角度预测和距离预测的相关模型程序。

在 preprocessing 目录下创建程序 calculate_aa_distance.ipynb。执行程序段 P5.1 导入库。

P5.1 #导入库

```
001  import numpy as np
002  import matplotlib.pyplot as plt
003  import os
```

执行程序段 P5.2,读取数据集。

```
P5.2  #读入文件
004   filename = '../dataset/training_30'
005   with open(filename, 'r') as f:
006       lines = f.readlines()     #读取所有行
007   print('文件中共有 {0} 行'.format(len(lines)))
```

运行结果显示：文件中共有 340 989 行数据。

由于[EVOLUTIONARY]字段域和[TERTIARY]字段域都包含多行数据,这些数据需要解析为浮点型数据列表,所以用程序段 P5.3 定义一个行数据解析函数。

```
P5.3  #行数据析取函数
008   def lines_split(lines, splice) :
          '''
          功能: 按照分隔符 splice,将字符行 lines 中的数值拆分为列表,数据转为 float 类型
          参数:
          lines:蛋白质结构文件中的若干行
          splice:分隔符
          返回值:
          return: 浮点型数据列表
          '''
009       data = []
010       for line in lines :
011           data.append([float(x) for x in line.split(splice)])
012       return data
```

执行程序段 P5.4,读取数据集中所有蛋白质的结构信息,分类存放。

```
P5.4  #读取数据集中所有的蛋白质结构信息
013   names = []       #蛋白质 ID
014   seqs = []        #氨基酸序列
015   coords = []      #主链上原子的坐标
016   pssms = []       #PSSM 评分矩阵
017   masks = []       #掩码序列
018   for i in range(len(lines)):
019       if lines[i] == '[ID]\n' :
020           names.append(lines[i + 1])
021       elif lines[i] == '[PRIMARY]\n' :
022           seqs.append(lines[i + 1])
023       elif lines[i] == '[TERTIARY]\n' :
024           coords.append(lines_split(lines[i + 1:i + 4], "\t"))
025       elif lines[i] == '[EVOLUTIONARY]\n' :
026           pssms.append(lines_split(lines[i + 1:i + 22], "\t"))
027       elif lines[i] == '[MASK]\n' :
028           masks.append(lines[i + 1])
029   print('数据集中包含{0}个蛋白质序列'.format(len(seqs)))
030   print('第 1 个蛋白质的序列长度为: {0}'.format(len(seqs[0]) - 1))
031   print('第 1 个蛋白质主链原子坐标矩阵的维度为: {0}'.format(np.array(coords[0]).shape))
032   print('第 1 个蛋白质的 Mask 掩码长度为: {0}'.format(len(masks[0]) - 1))
```

程序运行结果如下。

```
数据集中包含 10333 个蛋白质序列
第 1 个蛋白质的序列长度为: 307
第 1 个蛋白质主链原子坐标矩阵的维度为: (3, 921)
第 1 个蛋白质的 Mask 掩码长度为: 307
```

执行程序段 P5.5,按照残基序列长度统计 training_30 数据集中的蛋白质分布,如图 5.9 所示,该图给出的是残基序列长度在 6 个范围段上的汇总情况。

P5.5 ＃统计蛋白质分布

```
033  under64,under128,under136,under200,under300,above300 = 0,0,0,0,0,0
034  for i in range(len(seqs)):
035      if (len(seqs[i]) - 1 <= 64) :
036          under64 += 1
037      elif (len(seqs[i]) - 1 <= 128) :
038          under128 += 1
039      elif (len(seqs[i]) - 1 <= 136) :
040          under136 += 1
041      elif (len(seqs[i]) - 1 <= 200) :
042          under200 += 1
043      elif (len(seqs[i]) - 1 <= 300) :
044          under300 += 1
045      else :
046          above300 += 1
047  x = ['<= 64','(64, 128]','(128, 136]','(136, 200]','(200, 300]','> 300']
048  y = [under64, under128, under136, under200, under300, above300]
049  plt.ylabel('Counts',fontsize = 15)
050  plt.title('Amino Acid Sequence Length Counts',fontsize = 15)
051  plt.bar(x, y, alpha = 0.7)
```

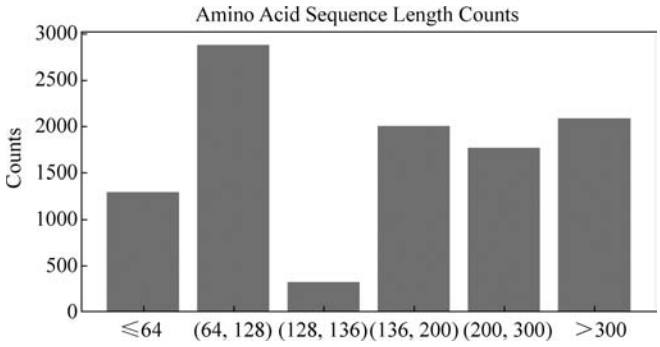

图 5.9 training_30 数据集的蛋白质序列分布

执行程序段 P5.6,删除连续两个原子坐标为 0 的蛋白质,因为连续坐标为 0,说明相邻原子数据缺失,无法计算角度。

```
P5.6  #删除连续两个原子坐标为 0 的蛋白质
052   flag = 1
053   index = []
054   for i in range(len(seqs)):              #遍历所有蛋白质
055       for j in range(len(seqs[i])):   #第 j 个原子与第 j + 1 个原子坐标均为 0
056           if ((coords[i][0][j] == 0.0 and coords[i][0][j + 1] == 0.0) and
057               (coords[i][1][j] == 0.0 and coords[i][1][j + 1] == 0.0) and
058               (coords[i][2][j] == 0.0 and coords[i][2][j + 1] == 0.0)):
059               flag = 0
060               break
061       if (flag == 1):
062           index.append(i)
063       flag = 1
064   print('筛选前包含的蛋白质数量: {0}'.format(len(names)))
065   print('筛选后包含的蛋白质数量: {0}'.format(len(index)))
```

运行结果如下。

```
筛选前包含的蛋白质数量: 10333
筛选后包含的蛋白质数量: 6241
```

执行程序段 P5.7,根据残基序列长度和 MASK 筛选蛋白质。

```
P5.7  #根据氨基酸序列长度筛选序列,长度小于或等于 L,并且 Mask 全部为" + "
066   print("原有的蛋白质结构总数:{0} ".format(len(seqs)))
067   L = 136 #指定长度
068   under = []
069   for i in range(len(index)):
070       k = index[i]
071       if (len(seqs[k] - 1) <= L) and (masks[k].find('-') < 0):
072           under.append(k)   #记录满足条件的蛋白质序号
073   print('氨基酸序列长度小于或等于 {0} 且 Mask 全部为 \" + \" 的蛋白质总数:
          {1}'.format( L, len(under)))
```

运行结果如下。

```
原有的蛋白质结构总数:10333
氨基酸序列长度小于或等于 136 且 Mask 全部为 " + " 的蛋白质总数: 2919
```

如果计算力不够,可以将 L 设置为更小的值,例如 $L=68$,正好是剪辑窗口宽度 34 的两倍。

5.6 计算残基之间的距离

N^i、C_α^i、C^i 表示第 i 个残基的三个原子,N^{i-1}、C_α^{i-1}、C^{i-1} 表示第 $i-1$ 个残基的三个原子,N^{i+1}、C_α^{i+1}、C^{i+1} 表示第 $i+1$ 个残基的三个原子,这些原子都分布在肽链的主链上。第 i 个残基与第 $i-1$ 个残基之间的距离,定义为 C_α^i 与 C_α^{i-1} 两个原子之间的距

离，第 i 个残基与第 $i+1$ 个残基之间的距离，定义为 C_α^i 与 C_α^{i+1} 两个原子之间的距离，如图 5.10 所示，虚线表示相邻两个残基之间的距离。

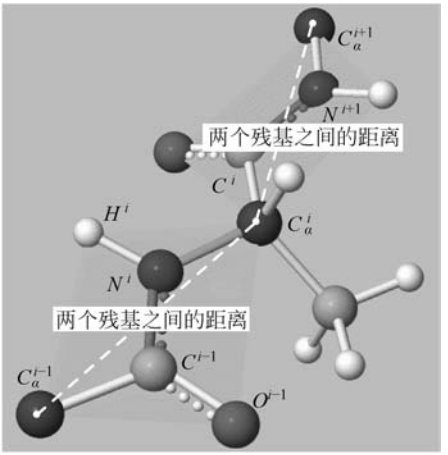

图 5.10　两个残基之间的距离

如图 5.10 所示，N^i-C^{i-1} 是连接两个氨基酸的肽键，C^i-N^{i+1} 也是肽键，肽键不能自由旋转，所以肽键连接的 6 个原子（C_α^{i-1}、C^{i-1}、O^{i-1}、H^i、N^i、C_α^i）处于同一平面内。执行程序段 P5.8，完成残基对之间的距离计算。

```
P5.8  #根据氨基酸 C-alpha 原子的坐标计算氨基酸残基对之间的距离
074   dists = []
075   protein_number = len(under)
076   for k in range(protein_number) :
077       dist = []
078       key = under[k]
079       coords_length = np.array(coords[key]).shape[1]
080       for i in range(coords_length) :
081           if i%3 == 1 : #每个氨基酸在主链上有 N-Calpha-C 三个原子,只处理 Calpha
                            #的坐标
082               aad = [] #存储第 i 个 AA 到所有 AA 之间的距离
083               for j in range( coords_length) :
084                   if j%3 == 1 : #只处理 Calpha 的坐标
                                  #第 i 个 AA 的 Calpha 的坐标
085                       coordi = np.array([coords[key][0][i],coords[key][1][i],
                          coords[key][2][i]])
                                  #第 j 个 AA 的 Calpha 的坐标
086                       coordj = np.array([coords[key][0][j],coords[key][1][j],
                          coords[key][2][j]])
087                       aa_dist = np.sqrt(np.sum(np.square(coordi - coordj)))  #计算距离
088                       aad.append(aa_dist) #第 i 个 AA 到第 j 个 AA 的距离加入列表
089               dist.append(aad) #第 i 个 AA 到所有 AA 的距离加入列表
090   dists.append(dist)  #当前序列的 AA 距离加入总列表
```

执行程序段 P5.9，对前面计算的数据做一个检测。

```
P5.9  #数据测试与检验
n = 0
key = under[n]
print(key)
print("蛋白质 ID: ", names[key])
print("残基序列: ", seqs[key])
print("序列长度", len(seqs[key]) - 1)
print("当前序列第 1 个残基 N 原子的坐标(x,y,z) = [{0}, {1}, {2}]".
          format(coords[key][0][0], coords[key][1][0], coords[key][1][0]))
print("当前序列第 1 个残基 Calpha 原子的坐标(x,y,z) = [{0}, {1}, {2}]".
          format(coords[key][0][1], coords[key][1][1], coords[key][1][1]))
print("当前序列第 1 个残基 C 原子的坐标(x,y,z) = [{0}, {1}, {2}]".
          format(coords[key][0][2], coords[key][1][2], coords[key][1][2]))
print("当前序列第 1 个残基与第 9 个残基之间的距离: ", dists[0][0][9])
print("当前序列生成的距离矩阵的维度: ", np.array(dists[0]).shape)
print("当前序列的最大距离: ",np.array(dists[0]).max())
print("当前序列的最小距离: ",np.array(dists[0]).min())
print("当前序列的平均距离: ",np.array(dists[0]).mean())
```

运行结果如下。

```
key = 1
蛋白质 ID: 2EUL_d2euld1
残基序列: MAREVKLTKAGYERLMQQLERERERLQEATKILQELMESSDDY
DDSGLEAAKQEKARIEARIDSLEDILSRAVILEE
序列长度 77
当前序列第 1 个残基 N 原子的坐标(x,y,z) = [981.8, 4076.1, 4076.1]
当前序列第 1 个残基 Calpha 原子的坐标(x,y,z) = [1093.7, 4174.2, 4174.2]
当前序列第 1 个残基 C 原子的坐标(x,y,z) = [1219.7, 4106.1, 4106.1]
当前序列第 1 个残基与第 9 个残基之间的距离: 2526.233815386058
当前序列生成的距离矩阵的维度: (77, 77)
当前序列的最大距离: 5536.502383274119
当前序列的最小距离: 0.0
当前序列的平均距离: 2063.4028892563697
```

执行程序段 P5.10,构建训练集文件 distance_aa_under136.txt,存放到 dataset/distances/目录下。

```
P5.10  #将处理完成的数据保存到新文件中
out_path = '../dataset/distances/'
if not os.path.exists(out_path):
    os.makedirs(out_path)
filename = out_path + 'distance_aa_under136.txt'
with open(filename, 'w') as f:
    for k in range(len(under)) :
        key = under[k]
        #保存 ID
```

```
112             f.write("\n[ID]\n")
113             f.write(names[key])
                #保存序列
114             f.write("\n[PRIMARY]\n")
115             f.write(seqs[key])
                #保存 PSSM 矩阵,PSSM 有 21 行
116             f.write("\n[EVOLUTIONARY]\n")
117             for line in range(21):
118                 for item in pssms[key][line]:
119                     f.write(str(item) + '\t')
120                 f.write('\n')
                #保存坐标矩阵,坐标有三行,分别代表 x,y,z
121             f.write("\n[TERTIARY]\n")
122             for line in range(3):
123                 for item in coords[key][line]:
124                     f.write(str(item) + '\t')
125                 f.write('\n')
                #保存距离矩阵,距离矩阵的维度与序列长度 L 有关: (L,L)
126             f.write("\n[DIST]\n")
127             for line in range(len(dists[k])):
128                 for item in dists[k][line]:
129                     f.write(str(item) + '\t')
130                 f.write('\n')
```

5.7 二面角与拉氏构象图

肽键连接的 6 个原子处于同一平面，如图 5.11 所示，Ca、C、N、H、O、Ca 六个原子共处同一平面。肽键中的 C-N 键具有部分双键性质，不能自由旋转，结果使肽键处在一个刚性的平面上，此平面被称为肽键平面(酰胺平面)。

两个肽键平面之间的 α 碳原子，可以作为一个旋转点形成二面角。二面角的变化，决定着多肽主键在三维空间的排列方式，是形成不同蛋白质构象的基础。

组成肽键的 6 个原子处于同一平面，每个残基的 C_α-N 键和 C_α-C 键是单键，可以相对自由旋转，围绕 C_α-N 键轴旋转产生的角度称为 Phi(Φ)，围绕 C_α-C 键轴旋转产生的角度称为 Psi(Ψ)。

根据拉氏构象图(Ramachandran plot)原理，二面角(Φ、Ψ)的变化范围为 $-180°$～$180°$，但是 Φ、Ψ 不能任意取值，有部分(Φ、Ψ)构象是不存在的，根据原子的范德华半径，在旋转过程中，原子间会发生碰撞冲突，无法实现(Φ、Ψ)的自由变化，如图 5.12 所示的拉氏构象图，以 Φ 为横坐标，以 Ψ 为纵坐标，显示了(Φ、Ψ)的分布规律。

图 5.12 显示了 100 000 个(Φ、Ψ)数据点的分布，每个数据点代表出现在单个氨基酸中的 Φ 和 Ψ 角度的组合。这些数据点不包括甘氨酸、脯氨酸，因为这两种氨基酸的拉氏构象分布具有自己的特点。

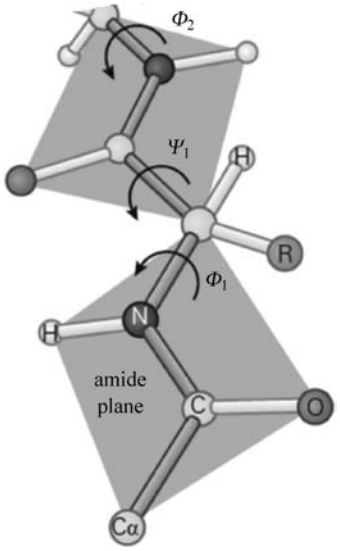

图 5.11 二面角 Phi(Φ)和 Psi(Ψ)

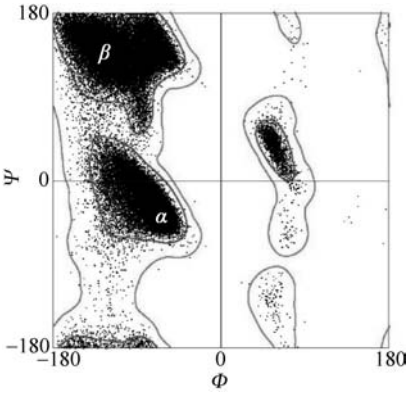

图 5.12 二面角拉氏构象图

图 5.12 中 α 螺旋构象的二面角区域标记为 α，β 折叠构象的二面角区域标记为 β。右上象限中的数据簇主要表示残基之间的转角。

5.8 计算二面角 Phi(Φ)和 Psi(Ψ)

在 preprocessing 目录下创建程序 calculate_phi_psi.ipynb。执行程序段 P5.11 导入库。

```
P5.11  #导入库
131    import numpy as np
132    import matplotlib.pyplot as plt
```

执行程序段 P5.12，定义函数，解析行数据。

```
P5.12  #定义函数,将行数据解析为 float 型数据列表
133    def parse_line(raw):
134        return np.array([[float(x) for x in line.split("\t") if x != ""] for line in raw])
```

执行程序段 P5.13，读取数据集中的蛋白质 ID、残基序列、PSSM 评分矩阵和原子坐标。

```
P5.13  #读取数据集
135    names = []        #ID
136    seqs = []         #序列
137    psis = []         #Psi
138    phis = []         #Phi
139    pssms = []        #PSSM
140    coords = []       #坐标
```

```
path = "../dataset/distances/distance_aa_under136.txt"
with open(path, "r") as f:
    lines = f.read().split('\n')  #读取文件内容,按行拆分
#从文本文件析取蛋白质的 ID、序列、PSSM 和坐标
for i,line in enumerate(lines):
    if line == "[ID]":
        names.append(lines[i+1])
    elif line == "[PRIMARY]":
        seqs.append(lines[i+1])
    elif line == "[EVOLUTIONARY]":
        pssms.append(parse_line(lines[i+1:i+22])) #PSSM
    elif line == "[TERTIARY]":
        coords.append(parse_line(lines[i+1:i+4])) #坐标
print (len(names))
```

运行结果显示数据集中有 2919 个序列长度不超过 136 的蛋白质。

执行程序段 P5.14,定义函数获取单种类型原子的坐标。

```
P5.14  #定义函数获取单种类型原子的坐标
def separate_coords(full_coords, pos):
    '''
    full_coords: 主链上所有原子的坐标
    pos: 0: n_term 原子, 1: calpha 原子, 2: cterm 原子
    return: 单种类型原子的坐标
    '''
    res = []
    for i in range(len(full_coords[1])):
        if i%3 == pos:
            res.append([full_coords[j][i] for j in range(3)]) #坐标(x,y,z)
    return np.array(res)
#根据原子类型分离坐标
coords_nterm = [separate_coords(full_coords, 0) for full_coords in coords] #n_term 原子
coords_calpha = [separate_coords(full_coords, 1) for full_coords in coords] #calpha 原子
coords_cterm = [separate_coords(full_coords, 2) for full_coords in coords] #cterm 原子
#坐标检查
print("第一个蛋白质第 1 个残基的三个原子 N-Ca-C 的坐标: ")
print(coords_nterm[0][0])
print(coords_calpha[0][0])
print(coords_cterm[0][0])
```

运行结果如下。

```
第一个蛋白质第 1 个残基的三个原子 N-Ca-C 的坐标:
[ 981.8 4076.1  -7423.1]
[ 1093.7 4174.2  -7419.3]
[ 1219.7 4106.1  -7366.7]
```

执行程序段 P5.15,定义函数,用向量法计算二面角。

```
P5.15  #定义函数,用向量法计算二面角
167    def get_dihedral(coords1, coords2, coords3, coords4):
           """
           coords1, coords2, coords3, coords4: 对应主链上四个原子的坐标
           计算 phi 角时: C - N - Calpha - C
           计算 psi 角时: N - Calpha - C - N
           Returns: 返回 phi 角或 psi 角
           """
168        a1 = coords2 - coords1
169        a2 = coords3 - coords2
170        a3 = coords4 - coords3
171        v1 = np.cross(a1, a2)
172        v1 = v1 / (v1 * v1).sum(-1) ** 0.5
173        v2 = np.cross(a2, a3)
174        v2 = v2 / (v2 * v2).sum(-1) ** 0.5
175        porm = np.sign((v1 * a3).sum(-1))
176        rad = np.arccos((v1 * v2).sum(-1) / ((v1 ** 2).sum(-1) * (v2 ** 2).sum(-1))
           ** 0.5)
177        if not porm == 0:
178            rad = rad * porm
179        return rad
```

执行程序段 P5.16,完成所有蛋白质序列主链上二面角的计算。

```
P5.16  #完成所有蛋白质序列主链上二面角的计算
180    phis, psis = [], []
181    ph_angle_dists, ps_angle_dists = [], []
       #遍历每一个蛋白质
182    for k in range(len(coords)):
183        phi, psi = [0.0], [] # phi 从 0 开始,psi 以 0 结束
           #遍历每一个氨基酸
184        for i in range(len(coords_calpha[k])):
               #计算 phi,第 1 个残基不计算 phi
185            if i>0:
186                phi.append(get_dihedral(coords_cterm[k][i-1], coords_nterm[k][i],
                           coords_calpha[k][i], coords_cterm[k][i]))
               #计算 psi,最后 1 个残基不计算 psi
187            if i<len(coords_calpha[k])-1:
188                psi.append(get_dihedral(coords_nterm[k][i], coords_calpha[k][i],
                           coords_cterm[k][i], coords_nterm[k][i+1]))
           #最后一个残基的 psi 为 0
189        psi.append(0)
           #添加到列表
190        phis.append(phi)
191        psis.append(psi)
```

执行程序段 P5.17,绘制 Ramachandran 图形,观察 100 个蛋白质的 Phi 和 Psi 分布。

P5.17 #绘制 Ramachandran 图形,观察 100 个蛋白质的 Phi 和 Psi 分布

```
n = 100  #蛋白质的数量
test_phi = []
for i in range(n):
    for test in phis[i]:
        test_phi.append(test)
test_phi = np.array(test_phi)
test_psi = []
for i in range(n):
    for test in psis[i]:
        test_psi.append(test)
test_psi = np.array(test_psi)
#按照象限统计 Phi 和 Psi 角度分布
quads = [0,0,0,0]
for i in range(len(test_phi)):
    if test_phi[i] >= 0 and test_psi[i] >= 0:
        quads[0] += 1
    elif test_phi[i] < 0 and test_psi[i] >= 0:
        quads[1] += 1
    elif test_phi[i] < 0 and test_psi[i] < 0:
        quads[2] += 1
    else:
        quads[3] += 1
print("象限分布: ", quads, " 总数: ", len(test_phi))
#绘制 Ramachandran 分布图
plt.scatter(test_phi, test_psi, marker = ".")
plt.xlim(-np.pi, np.pi)
plt.ylim(-np.pi, np.pi)
plt.xlabel("Phi")
plt.ylabel("Psi")
plt.show()
```

运行结果显示这 100 个蛋白质共包含 3980 对 Phi 和 Psi 角度,一、二、三、四象限分布数量分别为 294、1585、1965、136,对应的拉氏构象如图 5.13 所示。

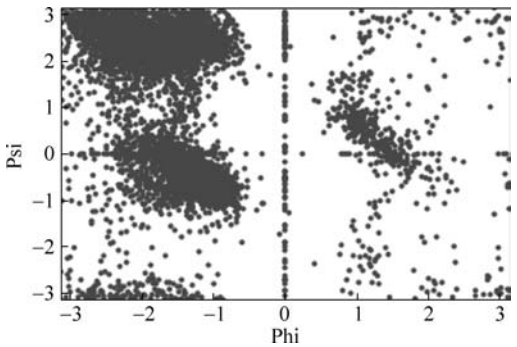

图 5.13 100 个蛋白质的拉氏构象分布

执行程序段 P5.18,定义函数将向量转换为字符串。

```
P5.18  #定义函数,将向量格式转换为字符串行
220    def stringify(vec):
221        line = ""
222        for v in vec:
223            line = line + str(v) + " "
224        return line
225    print([stringify([1,2,3,4,5,6])])
```

执行程序段 P5.19,保存文件,包含蛋白质 ID、氨基酸序列、评分矩阵以及二面角 Phi 和 Psi。

```
P5.19  #保存文件,包含蛋白质 ID、氨基酸序列、评分矩阵以及二面角 Phi 和 Psi
226    with open("../dataset/angles/angles_phi_psi_under64.txt", "w") as f:
227        for k in range(len(names)):
228            f.write("\n[ID]\n")
229            f.write(names[k])     #ID
230            f.write("\n[PRIMARY]\n")
231            f.write(seqs[k])     #序列
232            f.write("\n[EVOLUTIONARY]\n")
233            for j in range(len(pssms[k])):
234                f.write(stringify(pssms[k][j]) + "\n")   #PSSM
235            f.write("[PHI]\n")
236            f.write(stringify(phis[k]))     #PHI
237            f.write("\n[PSI]\n")
238            f.write(stringify(psis[k]) + '\n')     #PSI
```

5.9 裁剪残基序列的 One-Hot 矩阵

在 preprocessing 目录下创建程序 prepare_angle_data.ipynb。执行程序段 P5.20,将行数据解析为列表。

```
P5.20  #将行数据解析为列表
239    import numpy as np
240    def parse_lines(raw):
241        return np.array([[float(x) for x in line.split(" ") if x != ""] for line in raw])
242    def parse_line(line):
243        return np.array([float(x) for x in line.split(" ") if x != ""])
```

执行程序段 P5.21,读取 5.8 节完成的 Phi 和 Psi 数据集,包含 2919 个蛋白质的相关数据。

```
P5.21  #打开文件,读取 Phi 和 Psi 数据集
244    path = "../dataset/angles/angles_phi_psi_under136.txt"
245    with open(path, "r") as f:
246        lines = f.read().split('\n')
```

```
    names = []
    seqs = []
    psis = []
    phis = []
    pssms = []
    #读取每一个蛋白质的结构信息
    for i,line in enumerate(lines):
        if line == "[ID]":
            names.append(lines[i+1])
        elif line == "[PRIMARY]":
            seqs.append(lines[i+1])
        elif line == "[EVOLUTIONARY]":
            pssms.append(parse_lines(lines[i+1:i+22]))
        elif lines[i] == "[PHI]":
            phis.append(parse_line(lines[i+1]))
        elif lines[i] == "[PSI]":
            psis.append(parse_line(lines[i+1]))
    print(len(names))
```

DNA 翻译为蛋白质的编码图谱如图 5.14 所示，根据这个图谱不难看出，自然界已知氨基酸（不考虑最新发现的）只有 20 种，将这 20 种氨基酸按照一定顺序排列，例如 HRKDENQSYTCPAVLIGFWM。根据这个字母排列可以定义残基序列的 One-Hot 编码。

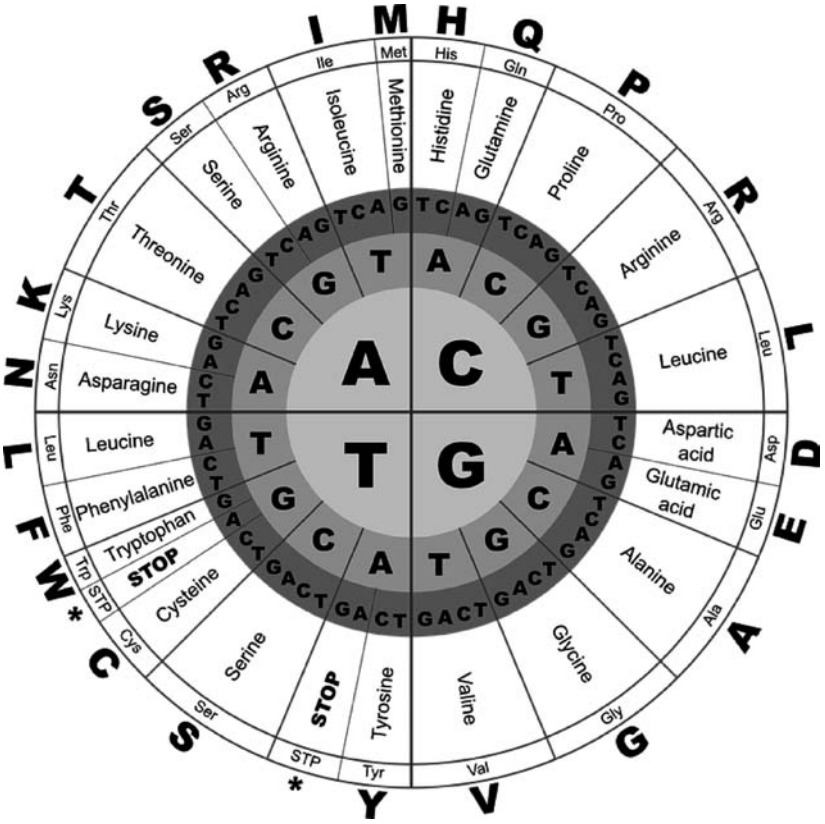

图 5.14　DNA 翻译为蛋白质的编码图谱

氨基酸序列通过肽键构成了肽链，肽链折叠形成蛋白质。根据氨基酸序列预测二面角，需要首先完成残基序列的 One-Hot 编码。以数据集中 ID 为 1KX6_1_A 的蛋白质序列(长度为 29)为例，对应的 One-Hot 编码如图 5.15 所示。

	H	S	Q	G	T	F	T	S	D	Y	S	K	Y	L	D	S	R	R	A	Q	D	F	V	Q	W	L	M	N	T
H	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	0	0
R	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
K	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
D	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
E	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	0	0	0
N	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	0	1	0
Q	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0
S	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Y	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
T	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
C	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	0	0	0
P	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	0	0	0
A	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
V	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
L	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
I	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	0	0	0
G	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
F	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
W	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
M	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

图 5.15　残基序列的 One-Hot 编码

执行程序段 P5.22，完成残基序列片段的 One-Hot 编码和范德华半径编码。单个残基的 One-Hot 编码为(20,1)的向量，后面加上范德华半径和范德华表面半径，所以维度为(22,1)，并对给定的残基序列 seq，从 pos 位置向左向右按照(34,22)的窗口裁剪。

```
P5.22  #残基序列片段的 One-Hot 编码和范德华半径编码,并裁剪
264    def onehotter_aa(seq, pos):
           '''
           功能: 每次截取长度为 34(17×2 个残基)的氨基酸序列片段
           seq: 氨基酸序列
           pos:截取片段的中心位置,截取起点为 pos-17,截取终点为 pos+17
           return: 截取片段的 One-Hot 编码
           '''
265        pad = 17
           #定义 20 种氨基酸序列表
266        key = "HRKDENQSYTCPAVLIGFWM"
           #20 种氨基酸的范德华半径
267        vdw_radius = {"H": 118, "R": 148, "K": 135, "D": 91, "E": 109, "N": 96, "Q": 114,
                         "S": 73, "Y": 141, "T": 93, "C": 86, "P": 90, "A": 67, "V": 105,
                         "L": 124, "I": 124, "G": 48, "F": 135, "W": 163, "M": 124}
268        radius_rel = vdw_radius.values()
269        basis = min(radius_rel)/max(radius_rel)
           #20 种氨基酸的范德华表面半径
270        surface = {"H": 151, "R": 196, "K": 167, "D": 106, "E": 138, "N": 113, "Q": 144,
                      "S": 80, "Y": 187, "T": 102, "C": 104, "P": 105, "A": 67, "V": 117,
                      "L": 137, "I": 140, "G": 0, "F": 175, "W": 217, "M": 160}
```

```
        surface_rel = surface.values()
        surface_basis = min(surface_rel)/max(surface_rel)
        #One-Hot 编码
        one_hot = []
        for i in range(pos-pad, pos+pad): #遍历截取的片段,片段长度为 pad×2
            vec = [0 for i in range(22)]  #先将 One-Hot 编码置为 0
            #将当前氨基酸与 20 种氨基酸序列表一一比对
            for j in range(len(key)):
                if seq[i] == key[j]:
                    vec[j] = 1 #此处标定为 1,表示氨基酸类型
                    #在 One-Hot 编码的末尾添加范德华半径和范德华表面半径,并归一化
                    vec[-2] = vdw_radius[key[j]]/max(radius_rel)-basis
                    vec[-1] = surface[key[j]]/max(surface_rel)-surface_basis
            one_hot.append(vec)  #将当前氨基酸的维度为(1,22)的编码向量加入列表 one_hot
        return np.array(one_hot)  #One_Hot 的维度为(34,22)
```

5.10　裁剪评分矩阵和二面角标签

本节对氨基酸序列片段进行裁剪,完成模型训练的数据准备工作,主要包括:

(1) 氨基酸序列的裁剪矩阵,单个片段维度为(34,22),如图 5.16(a)所示。

(2) 评分裁剪矩阵,单个片段裁剪维度为(34,21),如图 5.16(b)所示。

(3) 定义二面角标签,单个标签维度为(1,2),如图 5.16(c)所示。

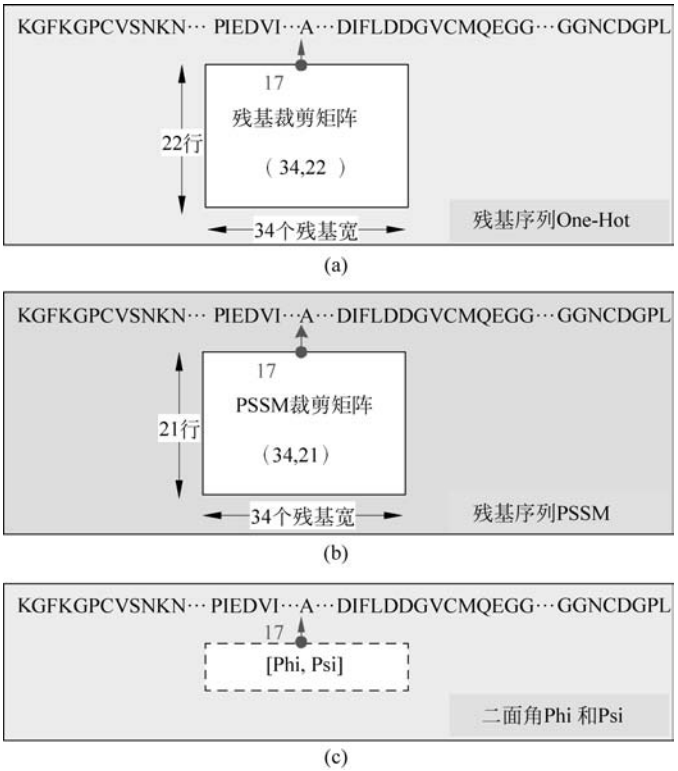

图 5.16　二面角训练集的裁剪方法

执行程序段 P5.23，裁剪 PSSM 评分矩阵。

```
P5.23  #裁剪 PSSM 评分矩阵
283    def pssm_cropper(pssm, pos):
           '''
           功能：每次截取长度为 34(17×2 个残基)的 PSSM 评分序列片段
           pssm：评分矩阵
           pos:截取片段的中心位置,截取起点为 pos－17,截取终点为 pos＋17
           return：截取的片段矩阵,维度为(21,34)
           '''
284        pssm_out = []
285        pad = 17
286        for i,row in enumerate(pssm):
287            pssm_out.append(row[pos－pad:pos＋pad])
288        return np.array(pssm_out)
       #检查所有的矩阵是否包含相同的蛋白质结构数
289    print("Names：", len(names))
290    print("Seqs：", len(seqs))
291    print("PSSMs：", len(pssms))
292    print("Phis：", len(phis))
293    print("Psis：", len(psis))
```

运行结果如下。

```
Names：2919
Seqs：2919
PSSMs：2919
Phis：2919
Psis：2919
```

执行程序段 P5.24，裁剪氨基酸序列、评分矩阵和二面角矩阵。

```
P5.24  #裁剪氨基酸序列、评分矩阵和二面角矩阵
294    input_aa = []
295    input_pssm = []
296    outputs = []
297    count = 0 #计数需要裁剪多少次
       #遍历每一个蛋白质结构
298    for i in range(len(seqs)):
299        if len(seqs[i])>17＊2：#序列长度大于 34
300            count ＋= len(seqs[i])－17＊2
301            for j in range(17,len(seqs[i])－17)：#设定 pos 的位置 j
               #根据 pos 的位置 j 向左向右裁剪序列
302                input_aa.append(onehotter_aa(seqs[i], j))
303                input_pssm.append(pssm_cropper(pssms[i], j))
304                outputs.append([phis[i][j], psis[i][j]])
305    print("全部的蛋白序列,共裁剪 {0} 次,这是样本的数量".format(count))
       #得到的矩阵长度
306    print("标签矩阵 outputs 的长度：", len(outputs))
307    print("残基片段矩阵 Inputs_aa 的长度：", len(input_aa))
308    print("评分片段矩阵 input_pssm 的长度：", len(input_pssm))
```

运行结果如下。

```
全部的蛋白序列,共裁剪 140756 次,这是样本的数量
标签矩阵 outputs 的长度: 140756
残基片段矩阵 Inputs_aa 的长度: 140756
评分片段矩阵 input_pssm 的长度: 140756
```

执行程序段 P5.25,调整特征矩阵的维度。

```
P5.25  #调整特征矩阵的维度
input_aa = np.array(input_aa).reshape(len(input_aa), 17 * 2, 22)
print('氨基酸序列片段矩阵的维度: {0}'.format(input_aa.shape))
input_pssm = np.array(input_pssm).reshape(len(input_pssm), 17 * 2, 21)
print('评分片段矩阵的维度: {0}'.format(input_pssm.shape))
```

运行结果如下。

```
氨基酸序列片段矩阵的维度: (140756, 34, 22)
评分片段矩阵的维度: (140756, 34, 21)
```

执行程序段 P5.26,将向量转换为字符串,以空格分隔数据项。

```
P5.26  #将向量转换为字符串,以空格分隔数据项
def stringify(vec):
    return "".join(str(v) + " " for v in vec)
```

执行程序段 P5.27,保存二面角训练集的标签,即将 Phi 和 Psi 保存到文件 outputs.txt。

```
P5.27  #保存二面角训练集的标签文件
out_path = '../dataset/angles/'
with open(out_path + "outputs.txt", "w") as f:
    for o in outputs:
        f.write(stringify(o) + "\n")
```

执行程序段 P5.28,将残基片段序列的矩阵保存到文件 input_aa.txt。

```
P5.28  #将残基片段序列的矩阵保存到文件 input_aa.txt
with open(out_path + "input_aa.txt", "w") as f:
    for aas in input_aa:
        f.write("\nNEW\n")
        for j in range(len(aas)):
            f.write(stringify(aas[j]) + "\n")
```

执行程序段 P5.29,将 PSSM 片段序列的矩阵保存到文件 input_pssm.txt。

```
P5.29  #将 PSSM 片段序列的矩阵保存到文件 input_pssm.txt
with open(out_path + "input_pssm.txt", "w") as f:
    for k in range(len(input_pssm)):
```

```
326             f.write("\nNEW\n")
327             for j in range(len(input_pssm[k])):
328                 f.write(stringify(input_pssm[k][j]) + "\n")
```

5.11 定义二面角预测模型

二面角预测模型采用 ResNet 结构，卷积层采用一维卷积，残差块结构如图 5.17 所示。

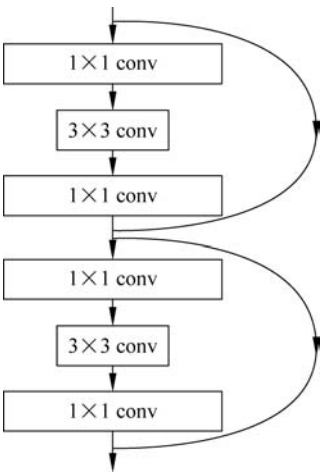

图 5.17 残差块结构

二面角预测模型结构如图 5.18 所示。样本的特征矩阵维度为(34,42)，标签维度为(1,4)。标签由原来的(Φ,Ψ)调整为(sinΦ,cosΦ,sinΨ,cosΨ)，可以借助角度的正弦值和余弦值，将标签值归一化为[−1,1]的值。

模型定义存放于 models/angles/resnet_1d_angles.py 文件，程序源码如程序段 P5.30 所示，将在二面角模型训练和预测程序 models/angles/predicting_angles.ipynb 中对其引用。

P5.30 ＃**ResNet 1D 卷积模型定义**

```
329 import keras
330 from keras.models import Model
331 from keras.regularizers import l2
332 from keras.losses import mean_squared_error, mean_absolute_error
333 from keras.layers.convolutional import Conv1D
334 from keras.layers import Dense, Flatten, Input, BatchNormalization, Activation
335 from keras.layers.pooling import AveragePooling1D
336 def custom_mse_mae(y_true, y_pred):
        """ 自定义损失函数 - MSE + MAE """
337     return mean_squared_error(y_true, y_pred) + mean_absolute_error(y_true, y_pred)
    ＃定义 1D 卷积层
```

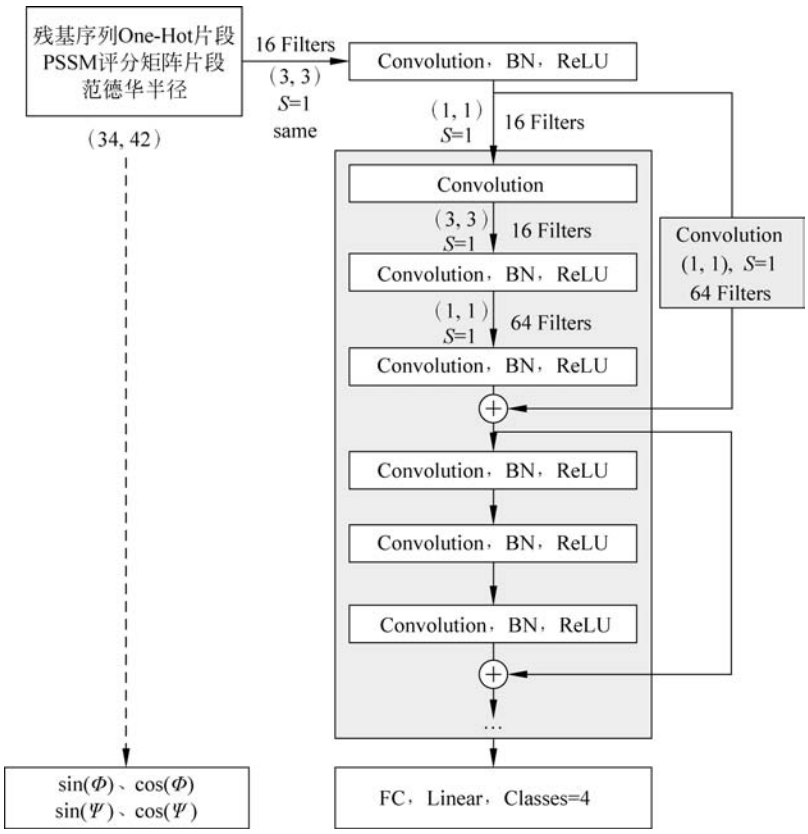

图 5.18　二面角预测模型结构

```
338    def resnet_layer(inputs,
                        num_filters=16,
                        kernel_size=3,
                        strides=1,
                        activation='relu',
                        batch_normalization=True,
                        conv_first=False):
        """卷积层有两种设计顺序,即: BN-Relu-Conv 或者 Conv-BN-Relu

        #参数
            inputs (tensor): 输入层或来自上一层的输入
            num_filters (int): 过滤器数量
            kernel_size (int): 过滤器尺寸
            strides (int): 步长
            activation (string): 激励函数名称
            batch_normalization (bool): 是否包含 BN 层
            conv_first (bool): conv-bn-activation (True) 或者 bn-activation-conv (False)
        #Returns
            x (tensor): 输出向量 x,作为下一层的输入
        """
        #定义 1D 卷积
```

```
339         conv = Conv1D(num_filters,
                          kernel_size=kernel_size,
                          strides=strides,
                          padding='same',
                          kernel_initializer='he_normal',
                          kernel_regularizer=l2(1e-4))
340         x = inputs
341         if conv_first:
342             x = conv(x)
343             if batch_normalization:
344                 x = BatchNormalization()(x)
345             if activation is not None:
346                 x = Activation(activation)(x)
347         else:
348             if batch_normalization:
349                 x = BatchNormalization()(x)
350             if activation is not None:
351                 x = Activation(activation)(x)
352             x = conv(x)
353         return x
        #定义 resnet_v2 的 1D 卷积网络
354     def resnet_v2(input_shape, depth, num_classes=4, conv_first=True):
            """ResNet Version 2 卷积网络的结构:
            瓶颈残差块采用 (1×1)-(3×3)-(1×1)卷积堆叠
            每个 stage 第 1 个残差块的直连采用 1×1 卷积做维度变换
            后续残差块采用恒等变换
            在 stage 的第 1 层,采用 strides=2 的步长下采样将特征图尺寸减半,过滤器数量翻倍
            同一 stage 内部,过滤器数量和特征图尺寸不变
            #参数
                input_shape (tensor): 输入层的维度
                depth (int): 核心卷积层的数量
                num_classes (int): 类别的数量
            #Returns
                model (Model): Keras model 实例
            """
355         if (depth - 2) % 9 != 0:
356             raise ValueError('网络深度 depth 按照公式 depth = 9n+2 计算 例如 depth=
                56 或 110')
            #初始化模型参数
357         num_filters_in = 16
358         num_res_blocks = int((depth - 2) / 9)
359         inputs = Input(shape=input_shape)
            #resnet_v2 第一层采用 conv-bn-activation 卷积
360         x = resnet_layer(inputs=inputs,
                             num_filters=num_filters_in,
                             conv_first=True)
            #一个 resnet 分为多个 stage,每个 stage 中有多个 block,每个 block 中包含多个层
361         for stage in range(3):
                #残差块 block
```

```
362             for res_block in range(num_res_blocks):
363                 activation = 'relu'
364                 batch_normalization = True
365                 strides = 1
366                 if stage == 0:
367                     num_filters_out = num_filters_in * 4  #第1个stage的过滤器数量
368                     if res_block == 0:  #第1个stage的第1层
369                         activation = None
370                         batch_normalization = False
371                 else:
372                     num_filters_out = num_filters_in * 2
373                     if res_block == 0: #非第1个stage的第1层
374                         strides = 2    #下采样
                    #定义瓶颈残差块
                    #残差块第1层:1×1的1D卷积
375                 y = resnet_layer(inputs=x,
                                     num_filters=num_filters_in,
                                     kernel_size=1,
                                     strides=strides,
                                     activation=activation,
                                     batch_normalization=batch_normalization,
                                     conv_first=conv_first)
                    #残差块第2层:3×3的1D卷积
376                 y = resnet_layer(inputs=y,
                                     num_filters=num_filters_in,
                                     conv_first=conv_first)
                    #残差块第3层:1×1的1D卷积
377                 y = resnet_layer(inputs=y,
                                     num_filters=num_filters_out,
                                     kernel_size=1,
                                     conv_first=conv_first)
378                 if res_block == 0:
                        #不同的stage之间需要做维度变换,用1×1卷积完成
379                     x = resnet_layer(inputs=x,
                                         num_filters=num_filters_out,
                                         kernel_size=1,
                                         strides=strides,
                                         activation=None,
                                         batch_normalization=False)
380                 x = keras.layers.add([x, y]) #完成残差块直连
381             num_filters_in = num_filters_out #后续每个stage,过滤器数量翻倍
382         x = BatchNormalization()(x)
383         x = Activation('relu')(x)
384         x = AveragePooling1D(pool_size=3)(x) #1D平均池化
385         y = Flatten()(x)
            #定义输出层
386         outputs = Dense(num_classes,
                            activation='linear',
                            kernel_initializer='he_normal')(y)
            #实例化模型
387         model = Model(inputs=inputs, outputs=outputs)
388         return model
```

5.12 二面角模型参数设定与训练

执行程序段 P5.31,导入库。

```
P5.31  #导入库
389    import numpy as np
390    import matplotlib.pyplot as plt
391    from keras.callbacks import EarlyStopping
       #导入 ResNet 1D 模型结构
392    from resnet_1d_angles import *
```

执行程序段 P5.32,读取 Phi 和 Psi 的标签文件,显示标签数据集的维度为(140756,2)。

```
P5.32  #读取 Phi 和 Psi 的标签文件
393    outputs = np.genfromtxt("../../dataset/angles/outputs.txt")
394    outputs[np.isnan(outputs)] = 0.0
395    outputs.shape
```

计算角度的 sin 值和 cos 值,用 Phi 和 Psi 的正余弦作为标签,如图 5.19 所示,单个残基对应的标签向量由(2,1)扩展为(4,1)。

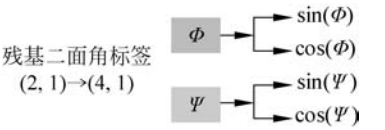

图 5.19 训练集标签扩展为(4,1)向量

执行程序段 P5.33,整个训练集的标签维度变为(140756,4)。

```
P5.33  #计算角度的 sin 值和 cos 值,用 Phi 和 Psi 的正余弦作为标签
396    out = []
397    out.append(np.sin(outputs[:,0]))
398    out.append(np.cos(outputs[:,0]))
399    out.append(np.sin(outputs[:,1]))
400    out.append(np.cos(outputs[:,1]))
401    out = np.array(out).T
402    print(out.shape)
```

执行程序段 P5.34,定义函数,读取残基序列的特征集或评分矩阵的特征集。

```
P5.34  #定义函数,读取残基序列的特征集或评分矩阵的特征集
403    def get_ins(path = "../../dataset/angles/input_aa.txt", pssm = None):
404        #pssm 文件路径
405        if pssm: path = "../../dataset/angles/input_pssm.txt"
406        with open(path, "r") as f:
407            lines = f.read().split('\n')
408        pre_ins = []
           #遍历每一个样本的特征矩阵
409        for i,line in enumerate(lines):
```

```
            if line == "NEW":
                prot = []
                raw = lines[i+1:i+(17*2+1)]
                #对每一个特征行做解析
                for r in raw:
                    prot.append(np.array([float(x) for x in r.split(" ") if x != ""]))
                #汇总每一个样本的特征矩阵
                pre_ins.append(np.array(prot))
        return np.array(pre_ins)
```

二面角特征矩阵的构建方法如图 5.20 所示。

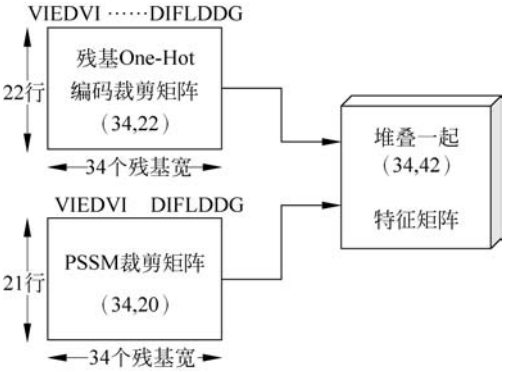

图 5.20 二面角训练集特征矩阵的构建方法

执行程序段 P5.35,堆叠残基序列和 PSSM 评分矩阵。

```
P5.35  #堆叠残基序列和 PSSM 评分矩阵
417    aas = get_ins() #读取残基序列矩阵
418    pssms = get_ins(pssm = True)  #读取 PSSM 评分矩阵
       #检查矩阵维度
419    print('残基序列维度: ', aas.shape, '\n 评分矩阵维度: ',pssms.shape)
       #特征矩阵堆叠
420    inputs = np.concatenate((aas[:, :, :20], pssms[:, :, :20], aas[:, :, 20:]), axis = 2)
421    print('堆叠矩阵维度: ', inputs.shape)
```

运行结果如下。

```
残基序列维度: (140756, 34, 22)
评分矩阵维度: (140756, 34, 21)
堆叠矩阵维度: (140756, 34, 42)
```

执行程序段 P5.36,模型定义和编译,设定优化算法、学习率等超参数。

```
P5.36  #模型定义和编译,设定超参数
422    adam = keras.optimizers.Adam(lr = 0.001, beta_1 = 0.9, beta_2 = 0.999,
           epsilon = 1e-8, decay = 0.0, amsgrad = True)  #采用 AMSGrad 优化算法
       #创建模型
```

```
model = resnet_v2(input_shape = (17 * 2,42), depth = 29, num_classes = 4, conv_first = True)
model.compile(optimizer = adam, loss = custom_mse_mae, metrics = ['accuracy'])
model.summary()
```

模型结构摘要显示,设定模型深度为 29 时,模型共有 31 个卷积层,需要学习训练的参数总量为 473 108 个。

执行程序段 P5.37,划分训练集与验证集,由于样本数量为 14 万条左右,所有设定训练集与验证集的划分比例为 8∶2。

```
P5.37 #训练集与验证集划分
split = int(len(inputs) * 0.8)
x_train, x_val = inputs[:split], inputs[split:]
y_train, y_val = out[:split], out[split:]
```

执行程序段 P5.38,模型训练,设定 epoch 数量为 20,mini-batch 大小为 16,如果连续 3 次迭代的损失不下降,则提前停止模型训练。

```
P5.38 #模型训练,如果连续 3 次迭代的损失不下降,则提前停止模型训练
early_stopping = EarlyStopping(monitor = 'val_loss', patience = 3)
his = model.fit(x_train, y_train, epochs = 20, batch_size = 16, verbose = 1, shuffle = True,
                validation_data = (x_val, y_val), callbacks = [early_stopping])
```

训练集规模为 112 604 个样本,验证集包含 28 152 个样本。训练主机 CPU 配置为 Intel® Core™ i7-6700 CPU@3.40Hz,内存配置为 16GB。模型的训练提前终止于第 20 代,完成 20 代训练,用时 1h11min31s。读者可根据自己的计算力,修正数据集规模。一般情况下,数据集越大,取得的效果会越好。

执行程序段 P5.39,绘制准确率曲线如图 5.21 所示,绘制损失函数曲线如图 5.22 所示。

```
P5.39 #绘制准确率和损失函数曲线
x = range(1, len(his.history['accuracy']) + 1)
plt.plot(x, his.history['accuracy'])
plt.plot(x, his.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.xticks(x)
plt.legend(['Train', 'Val'], loc = 'upper left')
plt.show()
plt.plot(x, his.history['loss'])
plt.plot(x, his.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.xticks(x)
plt.legend(['Train', 'Val'], loc = 'lower left')
plt.show()
```

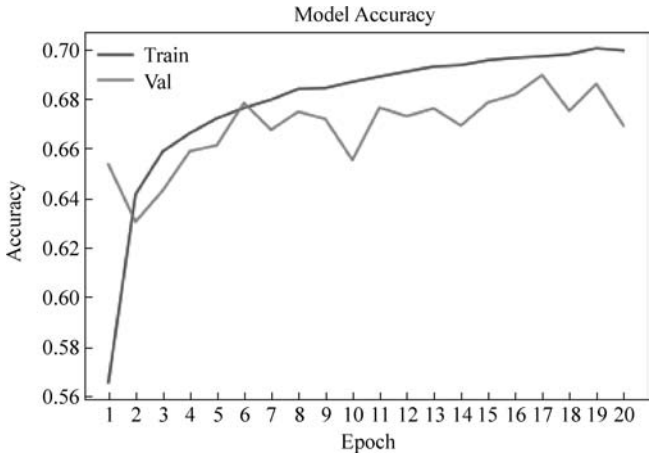

图 5.21　训练集和验证集准确率曲线对比

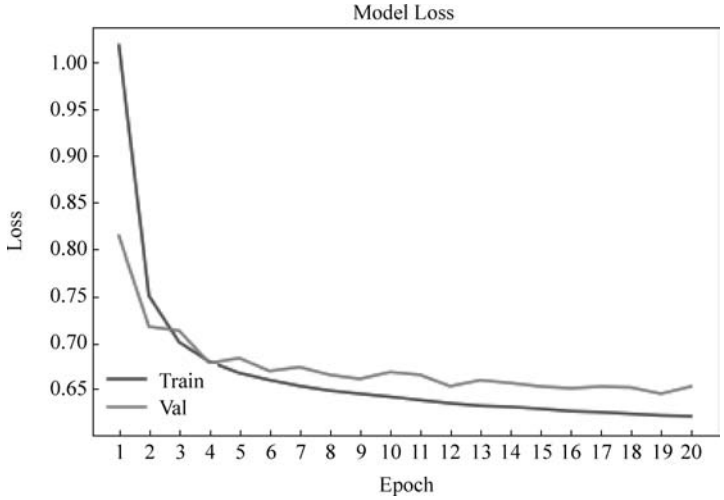

图 5.22　训练集和验证集损失函数下降曲线对比

准确率曲线显示模型的准确率起点较高，随着迭代次数增加，呈缓慢上升趋势，过拟合现象不明显。

损失函数曲线显示模型在验证集和训练集上的趋势基本一致，方差较小，模型的泛化能力较好。

5.13　二面角模型预测与评价

执行程序段 P5.40，在验证集上做预测，并将预测后得到的 sin(Φ)、cos(Φ)、sin(Ψ)、cos(Ψ)还原为角度值 Φ 和 Ψ，维度变为(28152,2)。

```
P5.40  #在验证集上做预测
448    preds = model.predict(x_val)
       #将验证集的预测结果 sin 值和 cos 值还原为二面角 Phi 和 Psi
449    refactor = []
450    for pred in preds:
451        angles = []
452        phi_sin, phi_cos, psi_sin, psi_cos = pred[0], pred[1], pred[2], pred[3]
       #Phi 分布在一、四象限
453        if (phi_sin>=0 and phi_cos>=0) or (phi_cos>=0 and phi_sin<=0):
454            angles.append(np.arctan(phi_sin/phi_cos))
455        elif (phi_cos<=0 and phi_sin>=0) :
456            angles.append(np.pi + np.arctan(phi_sin/phi_cos)) # Phi 在第二象限
457        else:
458            angles.append(-np.pi + np.arctan(phi_sin/phi_cos)) # Phi 在第三象限
       #Psi 分布在一、四象限
459        if (psi_sin>=0 and psi_cos>=0) or (psi_cos>=0 and psi_sin<=0):
460            angles.append(np.arctan(psi_sin/psi_cos))
461        elif (psi_cos<=0 and psi_sin>=0) :
462            angles.append(np.pi + np.arctan(psi_sin/psi_cos)) # Psi 在第二象限
463        else:
464            angles.append(-np.pi + np.arctan(psi_sin/psi_cos)) # Psi 在第三象限
465        refactor.append(angles)
       #将预测的角度限定在(-pi, pi)
466    refactor = np.array(refactor)
467    refactor[refactor>np.pi] = np.pi
468    refactor[refactor<-np.pi] = -np.pi
469    print(refactor.shape)
```

执行程序段 P5.41,绘制真实值与预测值的拉氏分布对比图,如图 5.23 所示。

```
P5.41  #绘制真实值与预测值的拉氏分布对比图
470    plt.scatter(outputs[split:,0], outputs[split:,1], marker=".")
471    plt.scatter(refactor[:,0], refactor[:,1], marker=".")
472    plt.legend(["Truth distribution", "Predictions distribution"], loc="lower right")
473    plt.xlim(-np.pi, np.pi)
474    plt.ylim(-np.pi, np.pi)
475    plt.xlabel("Phi")
476    plt.ylabel("Psi")
477    plt.show()
```

执行程序段 P5.42,判断预测值与真实值的相关性。

```
P5.42  #判断预测值与真实值的相关性
478    cos_phi = np.corrcoef(np.cos(refactor[:,0]), np.cos(outputs[split:,0]))
479    cos_psi = np.corrcoef(np.cos(refactor[:,1]), np.cos(outputs[split:,1]))
480    print("理想的相关系数应该是: Phi: 0.65 ,Psi: 0.7")
481    print("模型的 Phi 相关系数: ", cos_phi[0,1])
482    print("模型的 Psi 相关系数: ", cos_psi[0,1])
```

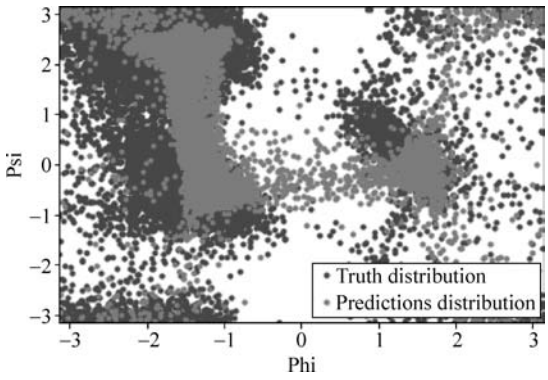

图 5.23　验证集上的预测值与真实值对比

运行结果如下。

```
理想的相关系数应该是: Phi: 0.65 ,Psi: 0.7
模型的 Phi 相关系数: 0.45145041419891035
模型的 Psi 相关系数: 0.5169855298544158
```

显然在 2919 个蛋白质上的训练结果还是比较理想的,增大蛋白质数据集规模,可望取得更好的效果。

执行程序段 P5.43,保存训练模型。

```
P5.43  #保存训练模型
483    model.save("resnet_1d_angles.h5")
```

根据需要,可以执行程序段 P5.44,加载预训练模型。

```
P5.44  #加载预训练模型
484    from keras.models import load_model
485    model = load_model("resnet_1d_angles.h5", custom_objects =
               {'custom_mse_mae': custom_mse_mae})
```

5.14　定义距离预测模型

距离预测模型仍然采用 ResNet 结构,与二面角预测模型采用 1D 卷积不同,距离预测模型采用 2D 卷积,结构如图 5.24 所示。

模型编码定义如程序段 P5.45 所示。

```
P5.45  #定义距离预测模型
486    import numpy as np
487    import keras
488    import keras.backend as K
```

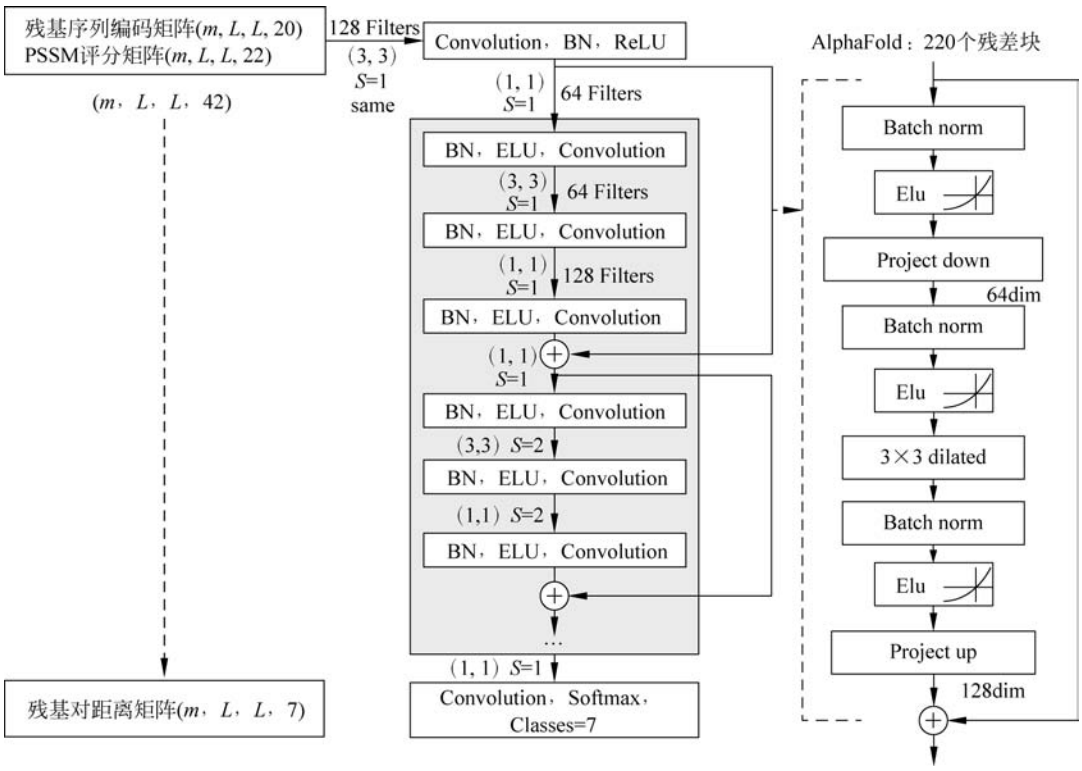

图 5.24 距离预测模型结构

```
489  from keras.models import Model
490  from keras.regularizers import l2
491  from keras.activations import softmax
492  from keras.layers.convolutional import Conv2D, Conv2DTranspose
493  from keras.layers import Input, BatchNormalization, Activation
     #softmax 回归函数
494  def softMaxAxis2(x):
         """ 在 axis 2 维度上计算 softmax """
495      return softmax(x,axis = 2)
     #带权重的损失函数定义
496  def weighted_categorical_crossentropy(weights):
         """
         功能:根据类别的权重计算损失
         参数:
             weights: 权重系数向量,维度为(C,) ,C 是类别的数量
         """
497      weights = K.variable(weights)
498      def loss(y_true, y_pred):
             #预测值归一化
499          y_pred /= K.sum(y_pred, axis = -1, keepdims = True)
             #数据范围剪辑
```

```
            y_pred = K.clip(y_pred, K.epsilon(), 1 - K.epsilon())
            #计算损失
            loss = y_true * K.log(y_pred) * weights
            loss = -K.sum(loss, -1)
            return loss
        return loss
    #残差块定义
    def resnet_block(inputs,
                         num_filters=64,
                         kernel_size=3,
                         strides=1,
                         activation='elu',
                         batch_normalization=True,
                         conv_first=False):
        """#参数:
            inputs (tensor): 输入层或上一层的输入
            num_filters (int): Conv2D 过滤器数量
            kernel_size (int): Conv2D 过滤器尺寸
            strides (int): Conv2D 步长
            activation (string): 激励函数名称
            batch_normalization (bool):是否进行 BN 计算
            conv_first (bool): conv-bn-activation (True) 或者 bn-activation-conv
            (False)
        """
        x = inputs
        #下采样
        x = BatchNormalization()(x)
        x = Activation(activation)(x)
        x = Conv2D(num_filters//2, kernel_size=1, strides=1, padding='same',
                   kernel_initializer='he_normal', kernel_regularizer=l2(1e-4))(x)
        #卷积计算
        x = BatchNormalization()(x)
        x = Activation(activation)(x)
        x = Conv2D(num_filters//2, kernel_size=3, strides=strides, padding='same',
                   kernel_initializer='he_normal', kernel_regularizer=l2(1e-4))(x)
        #上采样
        x = BatchNormalization()(x)
        x = Activation(activation)(x)
        x = Conv2DTranspose(num_filters, kernel_size=1, strides=strides, padding='same',
                   kernel_initializer='he_normal', kernel_regularizer=l2(1e-4))(x)
        return x
    #定义 ResNet 二维卷积层
    def resnet_layer(inputs,
                         num_filters=32,
                         kernel_size=3,
                         strides=1,
                         activation='relu',
                         batch_normalization=True,
                         conv_first=False):
```

```
        """
            功能：二维卷积：按照 BN－Activation－Conv 顺序
            Returns: x (tensor):输入下一层的向量
        """
518     conv = Conv2D(num_filters,
                      kernel_size = kernel_size,
                      strides = strides,
                      padding = 'same',
                      kernel_initializer = 'he_normal',
                      kernel_regularizer = l2(1e－4))
519     x = inputs
520     if conv_first:
521         x = conv(x)
522         if batch_normalization:
523             x = BatchNormalization()(x)
524         if activation is not None:
525             x = Activation(activation)(x)
526     else:
527         if batch_normalization:
528             x = BatchNormalization()(x)
529         if activation is not None:
530             x = Activation(activation)(x)
531         x = conv(x)
532     return x
    ＃RestNet 残差网络定义
533 def resnet_v2(input_shape, depth, num_classes = 4, conv_first = True):
        """
          采用 ELU 作为激励函数的 ResNet,网络深度参数 depth 应是 4 的倍数
        """
534     if depth % 4 != 0:
535         raise ValueError('depth should be 4n (eg 8 or 16)')
        ＃模型定义
536     num_filters_in = 128
537     inputs = Input(shape = input_shape)
        ＃定义第一个卷积层,Conv－BN－ReLU
538     x = resnet_layer(inputs = inputs,
                         num_filters = num_filters_in,
                         conv_first = True)
        ＃定义残差块的卷积步长
539     striding = [1,2,4,8]
        ＃遍历每一个 stage
540     for stage in range(depth):
541         activation = 'elu'
542         batch_normalization = True
            ＃瓶颈残差块
543         y = resnet_block(inputs = x,
                             num_filters = 128,
                             kernel_size = 3,
                             strides = striding[stage % 4])
```

```
544             x = keras.layers.add([x, y]) #直连求和
                #顶层添加一个线性卷积分类器
545             y = Conv2D(num_classes, kernel_size=1, strides=1, padding='same',
                        kernel_initializer='he_normal', kernel_regularizer=l2(1e-4))(x)
546             outputs = Activation(softMaxAxis2)(y)
                #初始化模型
547             model = Model(inputs=inputs, outputs=outputs)
548             return model
```

5.15 构建残基序列3D特征矩阵

前面筛选蛋白质序列时，设定的蛋白质序列长度为136，考虑到距离模型的计算需求较大，因此本节将蛋白质序列的长度范围限定为不超过64。

重新打开preprocessing目录下的calculate_aa_distance.ipynb程序，将程序段P5.7中L的值由136修改为64，同时将程序段P5.10中的文件名称修改为distance_aa_under64.txt，重新运行程序calculate_aa_distance.ipynb，生成新的距离数据集。

在model/distances目录下新建程序文档predicting_distances.ipynb，执行程序段P5.46，导入库。

P5.46 #导入库

```
549     import numpy as np
550     import matplotlib.pyplot as plt
551     from keras.callbacks import EarlyStopping
        #导入距离预测模型结构
552     from elu_resnet_2d_distances import *
```

执行程序段P5.47，定义行解析函数，将行数据转换为float型数据向量。

P5.47 #行解析函数，将行数据转换为float型数据向量

```
553     def parse_lines(raw):
554         return [[float(x) for x in line.split("\t") if x != ""] for line in raw]
555     def parse_line(line):
556         return [float(x) for x in line.split("\t") if x != ""]
```

执行程序段P5.48，读取文件，解析为行的集合。为了降低计算需求，程序段P5.48指定从distance_aa_under64.txt中读取数据。

P5.48 #读取文件，解析为行的集合

```
557     path = "../../dataset/distances/distance_aa_under64.txt"
558     with open(path, "r") as f:
559         lines = f.read().split('\n')
```

执行程序段 P5.49，读取所有蛋白质的结构信息，结果显示数据集共包含 890 个蛋白质结构信息。

```
P5.49  #读取所有蛋白质的结构信息
560    names = []
561    seqs = []
562    dists = []
563    pssms = []
       #遍历数据集文件的每一行
564    for i,line in enumerate(lines):
565        if line == "[ID]":
566            names.append(lines[i+1])
567        elif line == "[PRIMARY]":
568            seqs.append(lines[i+1])
569        elif line == "[EVOLUTIONARY]":
570            pssms.append(parse_lines(lines[i+1:i+21]))
571        elif line == "[DIST]":
572            dists.append(parse_lines(lines[i+1:i+len(seqs[-1])+1]))
573    print('蛋白质数量: {0}'.format(len(seqs)))
```

构建氨基酸序列的 3D 特征矩阵，即行列均为氨基酸序列，高和宽均为 L，不足 L 的部分用 0 补齐。深度为氨基酸的 One-Hot 编码向量，如图 5.25 所示。

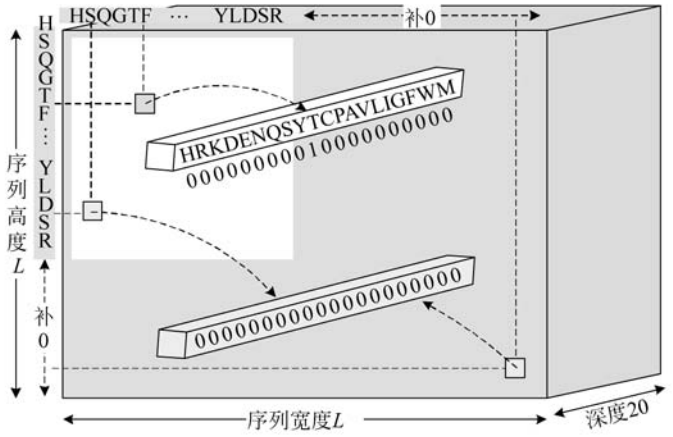

图 5.25　残基序列的 3D 特征矩阵

执行程序段 P5.50，将单个氨基酸序列转换为 $L \times L \times N$ 的特征矩阵。

```
P5.50  #将单个氨基酸序列转换为 L×L×N 的特征矩阵
574    def wider(seq, L=64, N=20):
           """ 将氨基酸序列转换为 One-Hot 编码,维度为: L×L×N,不是 L×N
               seq: 氨基酸序列
               L: 氨基酸序列的最大长度
               N: 20 种氨基酸序列长度
               return: L×L×N 特征矩阵
           """
```

```
575         key = "HRKDENQSYTCPAVLIGFWM"  #20 种氨基酸
576         tensor = []
577         for i in range(L):
578             d2 = []
579             for j in range(L):
580                 d1 = [1 if (j<len(seq) and i<len(seq) and key[x] == seq[i] and key
                    [x] == seq[j]) else 0
                          for x in range(N)]
581                 d2.append(d1) #d1 是维度为(1,20)的 One-Hot 特征向量
582             tensor.append(d2) #d2 是维度为(L,20)的特征矩阵
583         return np.array(tensor) #tensor 是维度为(L,L,20)的特征矩阵
```

执行程序段 P5.51,将所有蛋白质的残基序列编码为 $L\times L\times N$ 的特征矩阵。

```
P5.51  #将所有蛋白质的残基序列转换为 L×L×N 的特征矩阵
584    inputs_aa = np.array([wider(seq) for seq in seqs])
585    print('所有蛋白质残基序列构成的特征矩阵维度为:{0}'.format(inputs_aa.shape))
```

结果显示所有蛋白质残基序列构成的特征矩阵维度为(890,64,64,20)。

5.16 构建 3D 评分矩阵

将单个蛋白质的评分矩阵 PSSM 转换为 $L\times L\times N$ 的 3D 特征矩阵,L 表示序列的长度,N 表示氨基酸种类,固定为 20,如图 5.26 所示。

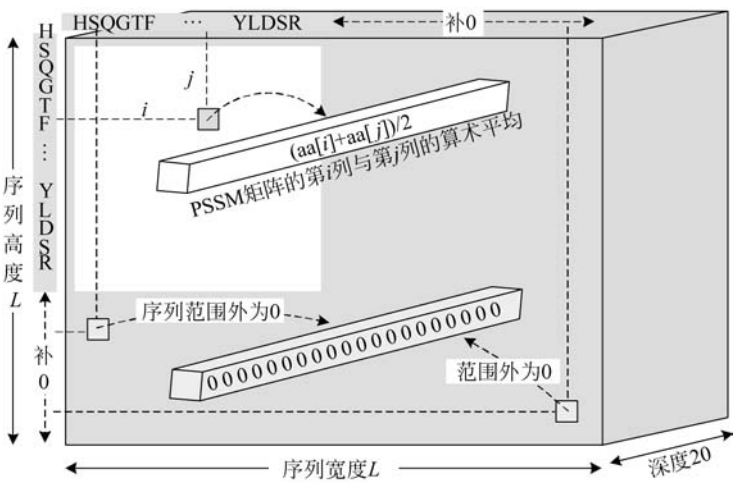

图 5.26 3D 评分矩阵的定义方法

实践中,在图 5.26 的基础上,根据两个残基间的相对关系,增加两个平衡值,扩展评分向量,深度方向的维度调整为 22。执行程序段 P5.52,将单个蛋白质的评分矩阵 PSSM 转换为 $L\times L\times(N+2)$的特征矩阵。

```
P5.52  #将单个蛋白质的评分矩阵 PSSM 转换为 L×L×(N+2)的特征矩阵
586    def wider_pssm(pssm, seq, L=64, N=20):
           """ 将 PSSM 转换为 L×L×N 矩阵,而不是 L×N """
587        key_alpha = "ACDEFGHIKLMNPQRSTVWY"
588        tensor = []
589        for i in range(L):
590            d2 = []
591            for j in range(L):
                   #残基序列范围内,添加一个维度为(1,20)的评分向量
592                if j<len(seq) and i<len(seq):
593                    d1 = [(aa[i]+aa[j])/2 for aa in pssm] #取 i、j 列的平均值
594                else: #范围外向量(1,20)的元素取值均为 0
595                    d1 = [0 for i in range(N)]
                   #残基序列范围内,扩展评分向量,取值 pssm[i]*pssm[j]
596                if j<len(seq) and i<len(seq):
597                    d1.append(pssm[key_alpha.index(seq[i])][i] *
598                              pssm[key_alpha.index(seq[j])][j]) # i、j 列评分乘积
599                else:    # 残基序列范围外,取值 0
600                    d1.append(0)
                   #根据残基序列 i、j 位置的相对关系扩展评分向量
601                if j<len(seq) and i<len(seq):
602                    d1.append(1-abs(i-j)/L)
603                else:
604                    d1.append(0)
605                d2.append(d1)  #d1 是一个(1,22)的向量
606            tensor.append(d2)  #d2 是一个(L,22)的矩阵
607        return np.array(tensor)  #tensor 是(L, L,22)的矩阵
```

执行程序段 P5.53,将所有蛋白质的评分矩阵转换为 $L\times L\times(N+2)$的特征矩阵。

```
P5.53  #将所有蛋白质的评分矩阵转换为 L×L×(N+2)的特征矩阵
608    inputs_pssm = np.array([wider_pssm(pssms[i], seqs[i]) for i in range(len(pssms))])
609    print('所有蛋白质构成的评分特征矩阵维度为:{0}'.format(inputs_pssm.shape))
```

结果显示所有蛋白质构成的评分特征矩阵维度为(890,64,64,22)。

执行程序段 P5.54,将残基序列矩阵和评分矩阵堆叠在一起,构建训练集特征矩阵(m,L,L,42)。

```
P5.54  #构建训练集特征矩阵(m,L,L,42)
610    inputs = np.concatenate((inputs_aa, inputs_pssm), axis=3)
611    print('训练集特征矩阵维度为:{0}'.format(inputs.shape))
       #删除不再使用的数据集,释放内存
612    del inputs_pssm
613    del inputs_aa
```

结果显示训练集特征矩阵维度为(890,64,64,42)。

5.17　定义距离标签的3D矩阵

距离是表征残基对之间关系的一种度量方法，5.6节计算得到的残基对之间的距离是一个维度为(L,L)的2D矩阵，距离是非负实数。本节根据距离的分段范围，将距离的2D矩阵转换为3D矩阵。

首先对单个蛋白质距离矩阵的维度进行扩展，如果残基序列的实际长度小于L，则空白处填充－1，如图5.27所示。

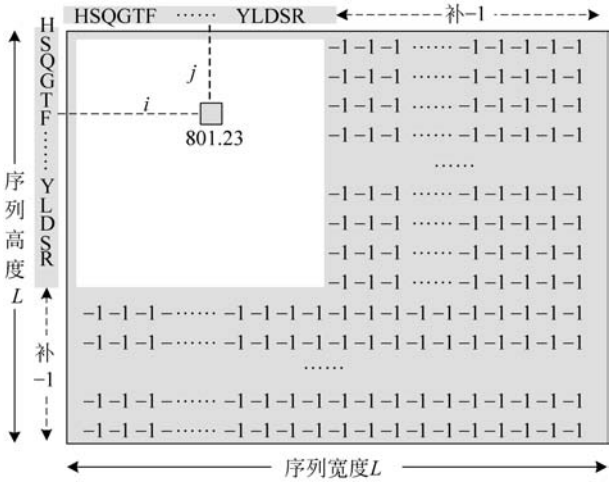

图5.27　残基距离矩阵扩展

执行程序段P5.55，将单个蛋白质的距离矩阵的维度扩展为(L,L)。

```
P5.55   #将单个蛋白质的距离矩阵的维度扩展为(L,L)
614     def embedding_matrix(matrix, L=64):
            #列方向扩展
615         for i in range(len(matrix)):
616             if len(matrix[i])<L:
617                 matrix[i].extend([-1 for i in range(L-len(matrix[i]))])
            #行方向扩展
618         while len(matrix)<L:
619             matrix.append([-1 for x in range(L)])
620         return np.array(matrix)
```

执行程序段P5.56，将所有蛋白质的距离矩阵拓展，构成训练集的标签矩阵(m,L,L)。

```
P5.56   #将所有蛋白质的距离矩阵拓展,构成训练集的标签矩阵(m,L,L)
621     dists = np.array([embedding_matrix(matrix) for matrix in dists])
622     print('训练集的距离标签矩阵维度为:{0}'.format(dists.shape))
```

运行结果显示训练集的距离标签矩阵维度为(890,64,64)。

以[<−0.5,≤500,≤750,≤1000,≤1400,≤1700,>1700]作为距离分段的标准,将距离划分为7个类别,对如图5.27所示的距离矩阵进行One-Hot编码,从而将2D距离矩阵拓展为3D标签矩阵。如图5.28所示,以距离为801.23为例,其对应的One-Hot向量为[0,0,0,1,0,0,0],序列范围以外的距离值−1,均转换为向量[1,0,0,0,0,0,0]。

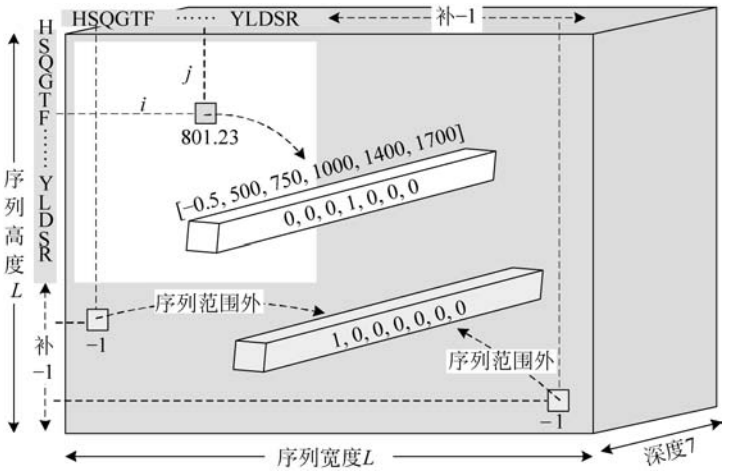

图5.28 构建距离标签的3D矩阵

定义程序段P5.57,将残基间的距离编码为7个类别。

```
P5.57  #将残基间的距离编码为7个类别
623    def treshold(matrix, cuts = [ - 0.5, 500, 750, 1000, 1400, 1700], L = 64):
           #将(L,L)特征矩阵转换为(L,L,7)特征矩阵
624        trash = (np.array(matrix)< cuts[0]).astype(np.int)   #<- 0.5 段 0
625        first = (np.array(matrix)<= cuts[1]).astype(np.int) - trash   #0.5～500 段 1
626        sec = (np.array(matrix)<= cuts[2]).astype(np.int) - trash - first   #500～750 段 2
627        third = (np.array(matrix)<= cuts[3]).astype(np.int) - trash - first - sec #7500～
                                                                          #1000 段 3
628        fourth = (np.array(matrix)<= cuts[4]).astype(np.int) - trash - first - sec -
           third   #1000～1400 段 4
629        fifth = (np.array(matrix)<= cuts[5]).astype(np.int) - trash - first - sec -
           third - fourth #1400～1700 段 5
630        sixth = np.array(matrix)> cuts[5]   #>1700 段 6
631        return np.concatenate((trash.reshape(L,L,1),
632                               first.reshape(L,L,1),
633                               sec.reshape(L,L,1),
634                               third.reshape(L,L,1),
635                               fourth.reshape(L,L,1),
636                               fifth.reshape(L,L,1),
637                               sixth.reshape(L,L,1)),axis = 2) #矩阵堆叠为(L,L,7)
```

执行程序段 P5.58,将所有蛋白质的距离矩阵编码为$(m,L,L,7)$的标签矩阵。

```
P5.58  #将所有蛋白质的距离矩阵编码为(m,L,L,7)的标签矩阵
638    outputs = np.array([treshold(d) for d in dists])
639    print('距离标签矩阵维度为:{0}'.format(outputs.shape))
640    del dists  #释放内存
```

运行结果显示距离标签矩阵维度为(890,64,64,7)。

5.18 距离模型参数设定与训练

执行程序段 P5.59,完成模型参数定义与编译。

```
P5.59  #模型参数定义与编译
       #设定优化算法
641    adam = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-8,
                  decay=0.0, amsgrad=True)
       #创建模型
642    model = resnet_v2(input_shape=(64,64, 42), depth=16, num_classes=7)
       #[1e-07, 0.45, 1.65, 1.75, 0.73, 0.77, 0.145]为输出标签的权重参数,用于损失值的计算
643    model.compile(optimizer=adam, loss=weighted_categorical_crossentropy(
                  np.array([1e-07, 0.45, 1.65, 1.75, 0.73, 0.77, 0.145])), metrics=
                  ["accuracy"])
644    model.summary()
```

运行结果显示模型包含 16 个残差块,50 个卷积层,需要学习训练的参数为 913 927 个。

执行程序段 P5.60,划分训练集与验证集,蛋白质总数为 890 个,训练集设定为 600 个,验证集设定为 290 个。

```
P5.60  #划分训练集与验证集
645    split = 600
646    x_train, x_val = inputs[:split], inputs[split:]
647    y_train, y_val = outputs[:split], outputs[split:]
```

执行程序段 P5.61,设定模型训练参数,开启训练过程,设定 epochs 为 50 代,early_stopping 参数的作用是,如果连续 5 次在验证集的损失值不下降,则提前停止训练。

```
P5.61  #设定模型训练参数,开启训练过程
       #连续5次验证集损失值不下降,停止训练
648    early_stopping = EarlyStopping(monitor='val_loss', patience=5)
649    his = model.fit(x_train, y_train, epochs=50, batch_size=4, verbose=1, shuffle=True,
                  validation_data=(x_val, y_val), callbacks=[early_stopping])
```

训练主机 CPU 配置为 Intel® Core™ i7-6700 CPU@3.40Hz,内存配置为 16GB,训

练过程提前终止于第 21 个 epoch，用时 1h21min51s。读者可根据自己的计算能力，修正数据集规模。数据集越大，取得的效果越好。

执行程序段 P5.62，保存模型，绘制准确率曲线如图 5.29 所示，损失函数曲线如图 5.30 所示。

P5.62 ＃保存模型，绘制准确率曲线和损失函数曲线

```
model.save("model_under_64.h5")
plt.figure(figsize=(8,4))
x=range(1, len(his.history['accuracy'])+1)
plt.plot(x,his.history["accuracy"])
plt.plot(x,his.history["val_accuracy"])
plt.legend(["accuracy", "val_accuracy"], loc="lower right")
plt.xlabel('Epoch')
plt.xticks(x)
plt.show()
plt.figure(figsize=(8,4))
plt.plot(x,his.history["loss"])
plt.plot(x,his.history["val_loss"])
plt.legend(["loss", "val_loss"], loc="upper right")
plt.xlabel('Epoch')
plt.xticks(x)
plt.show()
```

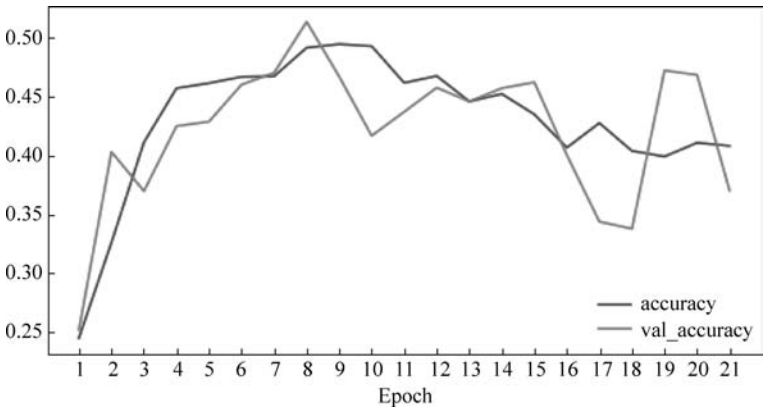

图 5.29 训练集与验证集的准确率曲线对比

如图 5.29 所示，验证集的准确率波动较大，应该是数据集样本的个体间的差异造成的，随着迭代次数的增加，准确率曲线在训练集与验证集均有下降趋势，再次证明数据集的样本不够充分。准确率在训练集与验证集的趋势保持一致，证明模型的方差控制在合理范围内。

如图 5.30 所示，训练集与验证集的损失函数下降趋势保持了高度一致，证明模型的方差小，泛化能力好。

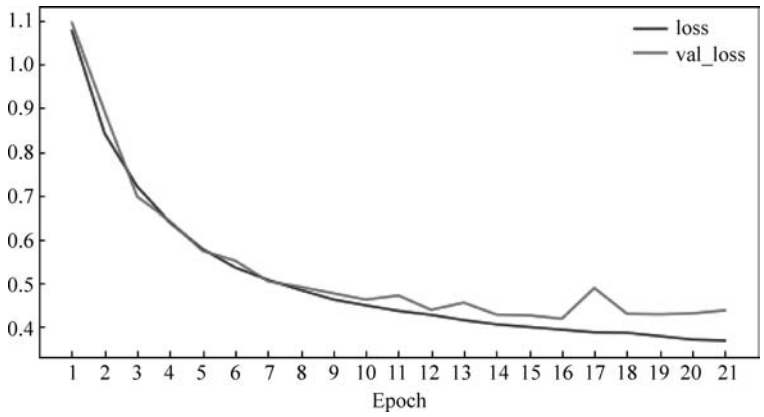

图 5.30　训练集与验证集的损失函数曲线对比

5.19　距离模型预测与评价

执行程序段 P5.63，对给定范围的蛋白质做出距离预测。

```
P5.63  #对给定范围的蛋白质做出距离预测
666    start, num = 0, 5 #预测范围: start 表示起点,num 表示数量
667    sample_pred = model.predict([inputs[start : start + num]])
```

执行程序段 P5.64，根据预测结果与真实标签中的最大值索引重新编码，将预测结果与真实标签中的 trash 类别的索引值 0 设为 7，从而用 1、2、3、4、5、6、7 表示 7 种距离值。

```
P5.64  #预测结果与真实标签的再次编码,用 1、2、3、4、5、6、7 表示 7 种距离值
668    preds_matrix = np.argmax(sample_pred, axis = 3)
669    preds_matrix[preds_matrix == 0] = 7 #将 trash 类别设置为 7,表示无穷远
670    outs_matrix = np.argmax(outputs[start : start + num], axis = 3)
671    outs_matrix[outs_matrix == 0] = 7 #将标签中的 trash 类别设为 7,表示无穷远
```

执行程序段 P5.65，根据准确率选择最好的 5 个预测结果。

```
P5.65  #根据准确率选择最好的 5 个预测结果
672    results = [np.sum(np.equal(pred[:len(seqs[start + j]),:len(seqs[start + j])],
                   outs_matrix[j,:len(seqs[start + j]),:len(seqs[start + j]),]),
                   axis = (0,1))/len(seqs[start + j]) * *2 for j,pred in enumerate(preds_matrix)]
673    best_score = max(results)
674    print("准确率最高值为: ", best_score)
675    sorted_scores = [acc for acc in sorted(results, key = lambda x: x, reverse = True)]
676    print("准确率最好的 5 个值为: ", sorted_scores[:5])
677    print("准确率最好的蛋白质序号为: ", [results.index(x) for x in sorted_scores[:5]])
678    best_score_index = results.index(best_score)
679    print("准确率最高的蛋白质序号为: ", best_score_index)
```

运行结果如下。

```
准确率最高值为: 0.7552083333333334
准确率最好的 5 个值为: [0.7552083333333334, 0.48404542996214167, 0.4822485207100592,
                    0.4359438660027162, 0.4351961950059453]
准确率最好的蛋白质序号为: [1, 2, 4, 3, 0]
准确率最高的蛋白质序号为: 1
```

执行程序段 P5.66,分别根据预测值与标签值绘制蛋白质距离图,真实距离分布如图 5.31 所示,预测距离分布如图 5.32 所示。

```
P5.66  #绘制蛋白质距离图(真实图和预测图对比)
680    best_score_index = 3 #显示序号为 3 的蛋白质
681    plt.title('Ground Truth of ' + names[best_score_index])     #真实图
682    norm = plt.Normalize(1, 7)
683    plt.imshow(outs_matrix[best_score_index, :len(seqs[start + best_score_index]),
               :len(seqs[start + best_score_index])], cmap = 'viridis_r', interpolation =
               'nearest', norm = norm)
684    plt.colorbar()
685    plt.show()
686    plt.title("Prediction by model of " + names[best_score_index])  #预测图
687    plt.imshow(preds_matrix[best_score_index, :len(seqs[start + best_score_index]),
                :len(seqs[start + best_score_index])], cmap = 'viridis_r', interpolation =
                'nearest', norm = norm)
688    plt.colorbar()
689    plt.show()
```

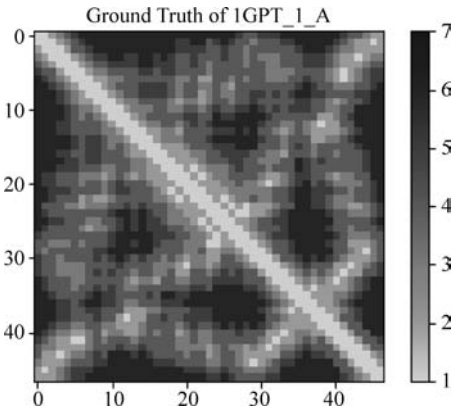

图 5.31 蛋白质 1GPT_1_A 的距离真实图

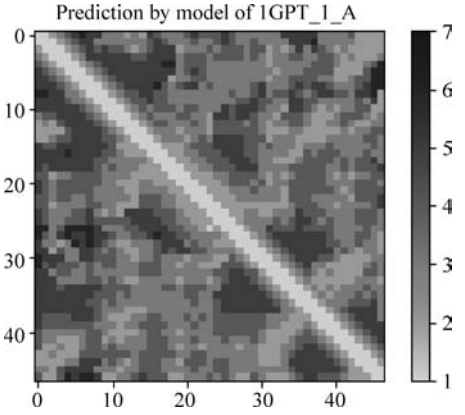

图 5.32 蛋白质 1GPT_1_A 的距离预测图

图 5.31 和图 5.32 中亮色区域(黄色区域)表示残基对的距离较近,深色区域(蓝色区域)表示距离较远。虽然预测图形比较模糊,与真实图形有较大误差,但是仍然可以看到轮廓骨架还是具有较高的相似性。

执行程序段 P5.67,通过混淆矩阵,进一步观察预测结果与真实值的偏差,得到混淆

矩阵如图 5.33 所示。

```
P5.67  #用混淆矩阵观察预测值与真实值的偏差
690    from sklearn.metrics import confusion_matrix
691    from sklearn.utils.multiclass import unique_labels
692    preds_crop = np.concatenate( [pred[:len(seqs[start + j]), :len(seqs[start + j])].
       flatten()
                   for j,pred in enumerate(preds_matrix)] )
693    outs_crop = np.concatenate( [outs_matrix[j, :len(seqs[start + j]), :len(seqs
       [start + j])].flatten()
                   for j,pred in enumerate(preds_matrix)] )
694    matrix = cm = confusion_matrix(outs_crop, preds_crop)
695    classes = [i + 1 for i in range(7)]
696    title = "Comfusion matrix"
697    cmap = "YlOrRd"
698    normalize = True
699    if normalize:
700        cm = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]
701        print("归一化的混淆矩阵")
702    else:
703        print('非归一化的混淆矩阵')
704    fig, ax = plt.subplots()
705    im = ax.imshow(cm, interpolation = 'nearest', cmap = cmap)
706    ax.figure.colorbar(im, ax = ax)
       #设置刻度,表示距离类型
707    ax.set(xticks = np.arange(cm.shape[1]),
708           yticks = np.arange(cm.shape[0]),
709           xticklabels = classes, yticklabels = classes,
710           title = title,
711           ylabel = 'True label',
712           xlabel = 'Predicted label')
713    plt.setp(ax.get_xticklabels(), rotation = 45, ha = "right",
714             rotation_mode = "anchor")
       #设置显示格式
715    fmt = '.2f' if normalize else 'd'
716    thresh = cm.max() / 2.
717    for i in range(cm.shape[0]):
718        for j in range(cm.shape[1]):
719            ax.text(j, i, format(cm[i, j], fmt),
720                    ha = "center", va = "center",
721                    color = "white" if cm[i, j] > thresh else "black")
722    fig.tight_layout()
723    print("总均方误差: ", np.linalg.norm(outs_crop - preds_crop))
724    print("平均均方误差: ", np.linalg.norm(outs_crop - preds_crop)/len(preds_matrix))
```

图 5.33 的行方向为真实值,列方向为预测值。不难看出,模型对 1、2、3、5、6 这五种距离的准确率都超过了 50%。对距离 1、2 的预测效果最好。对距离 4 的预测效果较差。