

第 5 章



深度学习

本章将简要介绍深度学习的一些基本内容和算法,现阶段关于深度学习的书籍或其他资料非常多,因此这里只简要阐述一些常见的神经网络算法。本章的算法和模型实现主要通过 PyTorch 模块实现。

5.1 PyTorch



视频讲解

PyTorch 模块是由 Facebook 在深度学习框架 Torch 的基础上,使用 Python 重写的一个全新的深度学习框架,不仅继承了 NumPy 模块的众多优点,同时还支持 GPU 加速计算,因此其计算效率明显好于 NumPy,而且相比于 TensorFlow,它的灵活性备受使用者喜欢。PyTorch 含有非常丰富的 API,可以轻松地完成深度神经网络模型的构建和实现,因此本章节的深度学习内容以 PyTorch 来实现。

5.1.1 PyTorch 安装

PyTorch 也是 Python 的一个模块,但其安装方式与之前介绍的内容稍有不同。需要登录官方网站(<https://pytorch.org/>)下载,它分为不同的系统版本,有 PyTorch 版本、Python 版本、显卡等安装形式。另外,需要根据自己的操作系统和软硬件配置进行搭配,如图 5.1 所示。

本书选用了稳定版本(1.0)的 PyTorch、Mac OS 系统,通过 pip 安装,Python 是 3.6 版本的,由于 MacBook Air 不支持 CUDA,因此无法进行 GPU 加速。图 5.1 给出的安装命令如下所示。

```
1 pip3 install torch torchvision
```

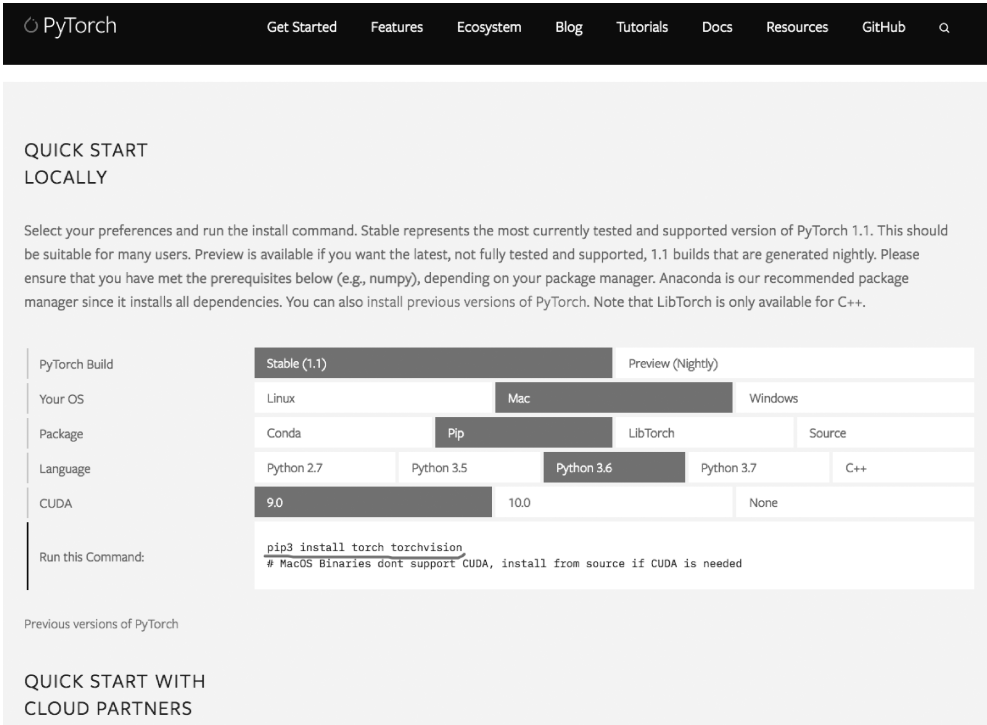


图 5.1 PyTorch 安装界面(实例图)

安装成功后,进入 Notebook 环境,并执行以下代码来验证是否成功,如图 5.2 所示。

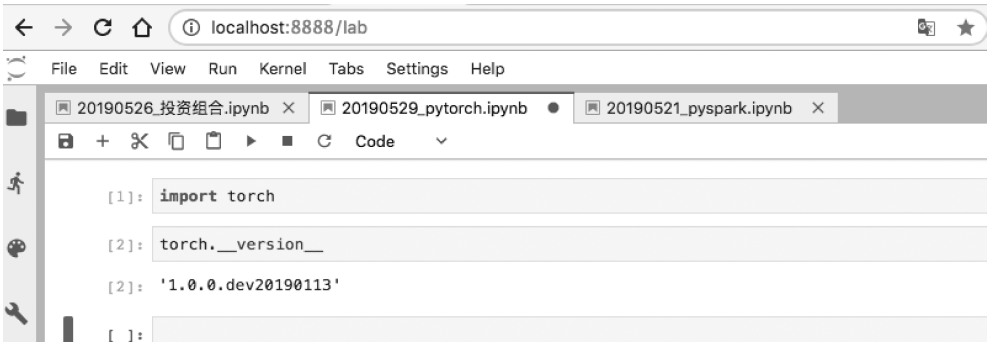


图 5.2 验证 PyTorch 是否成功安装

若安装失败,很可能是选择的安装版本与软硬件不相符造成的。安装成功后可通过以下命令验证是否支持 GPU 加速。

```
1 torch.cuda.is_available()      # 输出 ↓
2 False                        # 表示不支持
```

若支持 GPU 加速,在编程期间仅需添加以下命令,即可实现快速计算。

```

1  if torch.cuda.is_available():
2      x = x.cuda()                # GPU 加速
3      y = y.cuda()

```

5.1.2 创建 tensor

PyTorch 与 NumPy 在操作上非常相似,若读者熟悉 NumPy 模块的相关操作,那么在短时间内就能掌握并熟练应用 PyTorch。在这里简要阐述 tensor(张量)的一些基本操作,其代码如下。

```

1  # 构建 3×2 随机浮点数类型张量
2  a = torch.FloatTensor(3, 2)
3  a                                     # tensor 格式 输出 ↓
4  tensor([[ 1.4013e-45, 1.4013e-45],
5          [ 3.9883e-22, 1.4013e-45],
6          [-1.1426e-14, 4.5911e-41]])
7  a.numpy()                             # 转成数组 输出 ↓ (非常简单)
8  array([[ 1.4012985e-45, 1.4012985e-45],
9         [ 3.9882707e-22, 1.4012985e-45],
10        [-1.1426393e-14, 4.5910742e-41]], dtype=float32)
11 # 列表转 tensor
12 b = torch.FloatTensor([1, 0.618])
13 b                                     # 输出 ↓
14 tensor([1.0000, 0.6180])
15 # 构建 3×2 随机整数型
16 torch.IntTensor(3, 2)                 # 输出 ↓
17 tensor([[ 0, 1342177280],
18         [ 0, 1342177280],
19         [ 5,          0]], dtype=torch.int32)
20 # 列表转 tensor,列表元素为浮点型
21 torch.IntTensor([1.2, 3.2])           # 输出 ↓
22 tensor([1, 3], dtype=torch.int32)
23 # 构建 2×3 元素全为 0 的 tensor
24 torch.zeros(2, 3)                     # 输出 ↓
25 tensor([[0., 0., 0.],
26         [0., 0., 0.]])
27 # 构建 3×3 单位矩阵(格式: tensor)
28 torch.eye(3, 3)                       # numpy.eye(3, 3) 输出 ↓
29 tensor([[1., 0., 0.],
30         [0., 1., 0.],
31         [0., 0., 1.]])
32 # 构建 3×3 元素全为 1 的矩阵
33 torch.ones(3, 3)                      # 输出 ↓
34 tensor([[1., 1., 1.],
35         [1., 1., 1.],
36         [1., 1., 1.]])

```

```
37 # NumPy 转 tensor 转换非常简单直接
38 torch.from_numpy(np.array([[1, 2, 3], [3, 4, 5]])) # 输出 ↓
39 tensor([[1, 2, 3],
40         [3, 4, 5]])
```

5.1.3 基本运算

这里对 PyTorch 中的 tensor 基本运算进行简要阐述。

```
1  a = torch.randn(2, 3) # 随机数
2  a # 输出 ↓
3  tensor([[ 0.3786, 0.8703, 0.5232],
4          [ 0.2417, -0.7136, -1.2713]])
5  # 绝对值|x_{i}| 运算
6  torch.abs(a) # 输出 ↓
7  tensor([[0.3786, 0.8703, 0.5232],
8          [0.2417, 0.7136, 1.2713]])
9  # 加法 1, 维度要对应一致
10 a.add(b) # 输出 ↓
11 tensor([[0.7572, 1.7407, 1.0464],
12         [0.4834, 0.0000, 0.0000]])
13 # 加法 2
14 torch.add(a, b) # 输出 ↓
15 tensor([[0.7572, 1.7407, 1.0464],
16         [0.4834, 0.0000, 0.0000]])
17 # 加法 3, b ← a + b
18 b.add_(a) # 方法含有下画线的.method_ 会更新原有数组 输出 ↓
19 tensor([[0.7572, 1.7407, 1.0464],
20         [0.4834, 0.0000, 0.0000]])
21 b # 输出 ↓ (很强大, 方便)
22 tensor([[0.7572, 1.7407, 1.0464],
23         [0.4834, 0.0000, 0.0000]])
24 # 减法
25 b.sub_(a) # 复原 b 输出 ↓
26 b # 输出 ↓
27 tensor([[0.3786, 0.8703, 0.5232],
28         [0.2417, 0.7136, 1.2713]])
29 a - b # 减法, 同 a.sub(b)
30 # 乘法
31 a * b # 输出 ↓
32 tensor([[ 0.1433, 0.7575, 0.2738],
33         [ 0.0584, -0.5092, -1.6162]])
34 a.mul(b) # 乘法, 结果同上
35 # 除法
36 a.div(b) # 输出 ↓
```



```

37 tensor([[ 1.,  1.,  1.],
38         [ 1., -1., -1.]])
39 a / b                                # 除法,结果同上
40 # 余数
41 a % b                                # 输出 ↓
42 tensor([[0., 0., 0.],
43         [0., 0., 0.]])
44 a.pow(3)                             # 所有元素的3次方 输出 ↓
45 tensor([[ 0.0543, 0.6593, 0.1432],
46         [ 0.0141, -0.3634, -2.0546]])

```

从上面的代码不难发现,PyTorch 关于基本运算是非常方便的,但不局限于此,比如定义一个简单的分段函数。

$$f(x) = \begin{cases} a, & x \leq a \\ x, & a < x < b \\ b, & x \geq b \end{cases} \quad (5.1)$$

其中, a 和 b 是待定常数。下面通过 PyTorch 来实现这个函数。

```

1  a                                # tensor 输出 ↓
2  tensor([[ 0.3786, 0.8703, 0.5232],
3         [ 0.2417, -0.7136, -1.2713]])
4  torch.clamp(a, 0, 0.5)           # 取值 a = 0, b = 0.5 输出 ↓
5  tensor([[0.3786, 0.5000, 0.5000],
6         [0.2417, 0.0000, 0.0000]])
7  # 只操作最大值
8  torch.clamp_max(a, 0.5)          # 输出 ↓
9  tensor([[ 0.3786, 0.5000, 0.5000],
10         [ 0.2417, -0.7136, -1.2713]])
11 # 只操作最小值
12 torch.clamp_min(a, 0)             # 输出 ↓
13 tensor([[0.3786, 0.8703, 0.5232],
14         [0.2417, 0.0000, 0.0000]])
15 torch.clamp??                     # 查看命令的用法 输出 ↓ (notebook or ipython)
16 Docstring:
17 clamp(input, min, max, out = None) -> Tensor
18
19 Clamp all elements in :attr:'input' into the range '[':attr:'min', :attr:'max']' and return
20 a resulting tensor:
21 ...

```

除了以上基本运算,还有很多其他的运算,这里不再一一阐述,下面阐述矩阵运算。

```

1 A = torch.Tensor([[1, 2, 3]]) # 构建 1x3 矩阵
2 B = torch.Tensor([[1, 2, 3], [7, 8, 9]]) # 构建 2x3 矩阵
3 A.shape # 1 行 3 列 输出 ↓
4 torch.Size([1, 3])
5 B.shape # 2 行 3 列 输出 ↓
6 torch.Size([2, 3])
7 # 矩阵乘法 AB^T {T}: A: 1 × 3 B^T: 3 × 2
8 A.mm(B.reshape(3, 2)) # (同 torch.matmul(A, B.reshape(3, 2))) 输出 ↓
9 tensor([[31., 43.]])
10 # 矩阵与向量的乘法 Ax : mv(顺序不要乱): matrixvector
11 torch.mv(B, A[0]) # 输出 ↓
12 tensor([14., 50.])
13 # 方阵求逆
14 c = torch.randn(3, 3) # 构建 3×3 矩阵
15 c # 输出 ↓
16 tensor([[ 0.2634, 0.8859, -0.5662],
17         [-0.5709, -0.8612, -0.3837],
18         [-0.6534, -0.8777, 2.2320]])
19 c.inverse() # 矩阵 c 的逆 输出 ↓
20 tensor([[ -2.8565, -1.8718, -1.0465],
21         [ 1.9283, 0.2755, 0.5366],
22         [-0.0779, -0.4396, 0.3527]])
23 # 行列式
24 c.det() # 输出 ↓
25 tensor(0.7908)
26 # 验证逆矩阵乘法 AA^{-1} = I
27 torch.matmul(c.inverse(), c) # 输出 ↓
28 tensor([[ 1.0000e+00, -1.1921e-07, 4.7684e-07],
29         [ 0.0000e+00, 1.0000e+00, 0.0000e+00],
30         [-3.1692e-10, -1.5118e-08, 1.0000e+00]])
31 # 构建一个 2x4x4 张量
32 d = torch.randn(2, 4, 4)
33 d # 输出 ↓
34 tensor([[[[-0.0879, -0.6460, -0.1376, -1.2186],
35          [-0.9019, 1.1118, -0.0460, 2.1199],
36          [-1.2170, 0.0621, -1.1434, -1.1737],
37          [-1.3173, 0.6776, 0.2788, 0.4723]],
38         [[ 0.2511, -0.1904, -0.5661, -0.4801],
39          [-1.0341, -1.4300, 0.2876, 0.6154],
40          [-0.6928, 1.4691, -0.7081, -0.2806],
41          [-0.7519, -1.2448, 1.1138, -0.0593]]]])
42
43 d.shape # 输出 ↓
44 torch.Size([2, 4, 4])

```

```
45 d[0]                                # 切片输出 ↓
46 tensor([[ -0.0879, -0.6460, -0.1376, -1.2186],
47         [ -0.9019, 1.1118, -0.0460, 2.1199],
48         [ -1.2170, 0.0621, -1.1434, -1.1737],
49         [ -1.3173, 0.6776, 0.2788, 0.4723]])
```

关于 PyTorch 更多的内容,读者可以通过相关书籍或官方文档进行深入学习,这里不再大篇幅的详细介绍。

5.2 基础知识

在介绍神经网络之前,需要先对相关的思想方法和数学知识进行了解,因此需要补充一些相关的数学知识,以便能更好地理解后面的内容。

5.2.1 蒙特卡洛法

蒙特卡洛法也称统计模拟方法,曾被评为 20 世纪最美的算法之一。本书在前面介绍过关于蒙特卡洛法的实例(求 π 的近似值),其思想就是通过模拟参数(采样)来求解一个复杂问题的(相对)最优解。通常模拟(采样)数量越多越接近最优解,在关于线性回归的章节中通过理论推导出参数的值(精确值),但是一个含有很多变量的函数就很难处理,蒙特卡洛法可以在一定程度上解决这个问题。举个简单的例子,给定一个一元二次函数 $f(x)=x^2$,并添加随机噪声,其结果如图 5.3 所示。

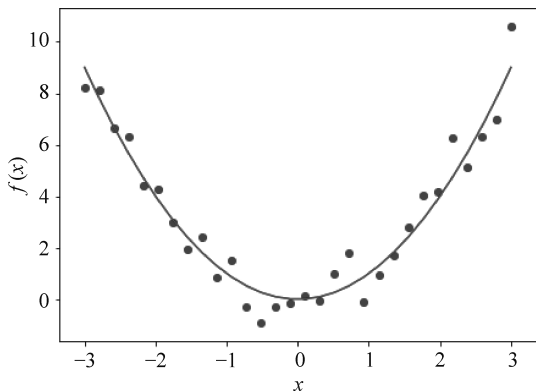


图 5.3 $f(x)$ 函数和添加随机噪声后的散点图

图 5.3 中的散点序列 $\{(x_i, y_i)\}$ 具有非常直观的现象。现在若想构造一个函数来描述这些散点,进而逼近原函数。众所周知:任何形式的曲线都可以通过多项式去逼近,问题在于如何构造一个合适的函数。对于一个变元问题,构建函数次数的高低会影响模型的性能:

- 构建函数的次数低,易造成模型的欠拟合问题;
- 构建函数的次数高,易造成模型的过拟合问题。

针对散点序列 $\{(x_i, y_i)\} (i=1, 2, \dots, m)$, 理论上希望散点尽可能地都落在已构造的函数上, 如式(5.2)所示。

$$y_i = a_0 + a_1 x_i + a_2 x_i^2 + \dots + a_n x_i^n + \epsilon_i \quad (5.2)$$

其中, $a_j (j=0, 1, 2, \dots, n)$ 是待定系数, ϵ_i 是扰动项。通过观察散点图, 不妨构造如下函数(一元二次函数)。

$$\hat{f}(x_i) = ax_i^2 + bx_i + c \quad (5.3)$$

其中, $\hat{f}$ 是拟合函数, a, b, c 是待定系数(需要求解的)。通过图形和函数性质分析, 不难推测出系数 $a > 0$, b 和 c 均比较小, 这样做的目的是可以在生成随机数组时进行剔除不满足条件的数, 下面的实例中暂不考虑这种情况。

根据式(5.3), 其损失函数如下所示。

$$J(a, b, c) = \frac{1}{m} \sum_{i=1}^m (\hat{f}(x_i) - y_i)^2 \quad (5.4)$$

要满足 $J(a, b, c)$ (代价函数, 又称损失函数) 最小, 即 $J(a^*, b^*, c^*) = \min J(a, b, c)$, 其中 (a^*, b^*, c^*) 是待求的“最佳系数组”。式(5.4)可通过理论进行推导, 现在并不打算这样做, 先仅通过蒙特卡洛法来生成 q 个(视情况而定)随机数组 $(a_r, b_r, c_r) (r=1, 2, \dots, q)$, 以遍历的形式(或并行计算)代入式(5.4)求解 J 的值, 在 q 个结果中筛选出最小值 J 对应的数组 (a_r, b_r, c_r) , 即求解的待定系数组 (a^*, b^*, c^*) 。

通过以上理论分析, 构造的式(5.3)针对散点序列 $\{(x_i, y_i)\}$ 结合蒙特卡洛法进行求解的代码如下所示。

```

1  import numpy as np
2  import torch
3  from torch.autograd import Variable
4  import matplotlib.pyplot as plt
5  %matplotlib inline
6  x = np.linspace(-3, 3, 30)                # x 轴数据
7  f = x ** 2                                # 原函数
8  f_noise = f + np.random.randn(x.__len__()) # 添加噪声后的数据
9  # 定义拟合函数
10 def f_prob(x, a, b, c):
11     return a * x ** 2 + b * x + c
12 x_tensor = torch.from_numpy(x)             # array2tensor
13 f_tensor = torch.from_numpy(f_noise)        # array2tensor
14 tmp_result_dict = {}                       # 创建存储计算结果字典
15 for _ in range(100000):                     # 这里模拟 100 000 次
16     arg_value = torch.randn(3)              # 生成随机数
17     # 不含 1/m 的 J 函数
18     J = (f_prob(x_tensor, *arg_value) - f_tensor).pow(2).sum().numpy()
19     tmp_result_dict[str(arg_value.tolist())] = J
20 loss_min_label = min(tmp_result_dict, key = tmp_result_dict.get)
21 tmp_result_dict[loss_min_label]             # 最小 J 值对应的待定系数组
                                              # 输出 ↓

```

```

22 array(14.71679089)          # 差值平方和(14.71679089/30)相当小,在可接受范围内
23 # 最佳系数组元素转换成数值类型 str2float
24 best_args_list = [round(float(i),4) for i in loss_min_label.strip('[]').split(',')]
25 print(best_args_list)        # 输出 ↓
26 [1.015, 0.1198, 0.0314]

```

通过 100000 次模拟可得到一个相对最佳的解,即 $a=1.015, b=0.1198, c=0.0314$ 。代入构造拟合函数(式(5.3))可得 $f(x)=1.015x^2+0.1198x+0.0314$ 。将数据代入函数对其进行绘图,结果如图 5.4 所示。

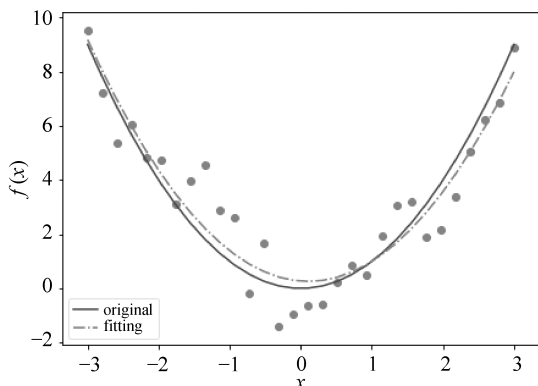


图 5.4 拟合函数与原函数对比图

实现图 5.4 的代码如下所示。

```

1  f_fitting = f_prob(x_tensor, *best_args_list)
2  fig = plt.figure(figsize=(7, 5))
3  plt.plot(x, f, 'r-', label='original')          # 函数图像
4  plt.scatter(x, f_noise)                         # 噪声点
5  plt.plot(x, f_fitting.numpy(), 'g-.', label='fitting') # 拟合函数
6  plt.legend()
7  plt.xlabel('$ x $ ')
8  plt.ylabel('$ f(x) $ ')

```

通过蒙特卡洛法求解出一个相对最佳的结果,但是通过 100 000 次模拟参数数组去计算的,尽管现在的计算机计算力比较强,但是相对于研究数学的人而言这并非首选方法,原因有以下几点:

- 尽管复杂度不大,但是计算量大;
- 求解的结果仅是一种近似解,且与模拟次数有很大的关系;
- 数学研究志在求解精确解,即使是一种近似,理论上也要确定误差范围。

读者可以尝试构造一元一次函数或一元高次(大于 2)函数进行实验,看效果如何。若通过蒙特卡洛法求解多元高次函数,且在待定系数取值范围不好确定的情况下,为求解最佳待定系数需要模拟的次数规模是一件可怕的事情,显然这不是最佳方法,但是在求解

极其复杂的问题时,蒙特卡洛法不失为一种有效的方法。

5.2.2 梯度下降法

在前面章节介绍过牛顿法(求解方程的根),这里再介绍另一种算法:梯度下降法。仍然以蒙特卡洛法中的实例,针对式(5.4)定义的 $J(a, b, c)$,先计算关于 a, b, c 的导数。

$$\begin{cases} \frac{\partial J(a, b, c)}{\partial a} = \frac{2}{m} \sum_{i=1}^m (ax_i^2 + bx_i + c - y_i)x_i^2 \\ \frac{\partial J(a, b, c)}{\partial b} = \frac{2}{m} \sum_{i=1}^m (ax_i^2 + bx_i + c - y_i)x_i \\ \frac{\partial J(a, b, c)}{\partial c} = \frac{2}{m} \sum_{i=1}^m (ax_i^2 + bx_i + c - y_i) \end{cases} \quad (5.5)$$

根据式(5.5)构造梯度 $\nabla J(a, b, c) = \left(\frac{\partial J(a, b, c)}{\partial a}, \frac{\partial J(a, b, c)}{\partial b}, \frac{\partial J(a, b, c)}{\partial c} \right)$ 。不妨令 $\theta = (a, b, c)$,则梯度下降法的迭代式为

$$\theta_n = \theta_{n-1} - \eta \nabla J(\theta_{n-1}) \quad (5.6)$$

其中, η (称为学习率)是指定常数,通常的范围为 $(0, 1)$, θ_0 是给定的初值, θ_n 是第 n 次迭代后的值($n=0, 1, 2, \dots$)。

用梯度下降法极小化 $J(a, b, c)$ 代码如下所示。在这个例子中, $\eta=0.05$ 。

```
1  # 梯度函数
2  def gradient(x, y, a, b, c):
3      x_len = len(x)
4      tmp_value = a * x.pow(2) + b * x + c - y          # tensor 计算
5      ja = 2 / x_len * (tmp_value * x.pow(2)).sum().numpy()
6      jb = 2 / x_len * (tmp_value * x).sum().numpy()
7      jc = 2 / x_len * tmp_value.sum().numpy()
8      return torch.Tensor([ja, jb, jc])
9  # 给定一组初值系数
10 coeff_value = torch.Tensor([1, 1, 1])
11 loss_result_list = []                                # 存储损失函数值
12 for ix in range(20):
13     coeff_value = coeff_value - 0.05 * gradient(x_tensor, f_tensor, *coeff_value)
14     loss_value = (f_prob(x_tensor, *coeff_value) - f_tensor).pow(2).sum().numpy()
15     loss_result_list.append(round(float(loss_value), 2))
16     print("迭代步数:{0}, 系数数组: {1}, 损失函数值: {2:.2f}".format(ix + 1,
17                                     coeff_value, loss_value)) # 打印输出 ↓
17 迭代步数:1, 系数数组: tensor([0.7178, 0.7044, 0.9046]), 损失函数值: 76.06
18 迭代步数:2, 系数数组: tensor([0.9878, 0.5037, 0.9092]), 损失函数值: 52.66
19 迭代步数:3, 系数数组: tensor([0.7573, 0.3673, 0.8268]), 损失函数值: 40.66
20 迭代步数:4, 系数数组: tensor([0.9793, 0.2746, 0.8266]), 损失函数值: 34.08
21 迭代步数:5, 系数数组: tensor([0.7910, 0.2117, 0.7552]), 损失函数值: 30.16
22 迭代步数:6, 系数数组: tensor([0.9737, 0.1689, 0.7513]), 损失函数值: 27.57
23 迭代步数:7, 系数数组: tensor([0.8199, 0.1399, 0.6892]), 损失函数值: 25.71
24 迭代步数:8, 系数数组: tensor([0.9703, 0.1202, 0.6826]), 损失函数值: 24.27
```

```

25 迭代步数:9, 系数数组: tensor([0.8448, 0.1068, 0.6285]), 损失函数值: 23.10
26 迭代步数:10, 系数数组: tensor([0.9686, 0.0977, 0.6200]), 损失函数值: 22.12
27 迭代步数:11, 系数数组: tensor([0.8663, 0.0915, 0.5727]), 损失函数值: 21.28
28 迭代步数:12, 系数数组: tensor([0.9683, 0.0873, 0.5629]), 损失函数值: 20.55
29 迭代步数:13, 系数数组: tensor([0.8849, 0.0844, 0.5214]), 损失函数值: 19.92
30 迭代步数:14, 系数数组: tensor([0.9689, 0.0825, 0.5108]), 损失函数值: 19.36
31 迭代步数:15, 系数数组: tensor([0.9011, 0.0812, 0.4742]), 损失函数值: 18.86
32 迭代步数:16, 系数数组: tensor([0.9704, 0.0803, 0.4632]), 损失函数值: 18.43
33 迭代步数:17, 系数数组: tensor([0.9151, 0.0797, 0.4309]), 损失函数值: 18.04
34 迭代步数:18, 系数数组: tensor([0.9723, 0.0793, 0.4197]), 损失函数值: 17.69
35 迭代步数:19, 系数数组: tensor([0.9274, 0.0790, 0.3912]), 损失函数值: 17.39
36 迭代步数:20, 系数数组: tensor([0.9746, 0.0788, 0.3800]), 损失函数值: 17.11
37 # 迭代后计算出的最佳系数
38 coeff_value                                # 输出 ↓
39 tensor([0.9746, 0.0788, 0.3800])

```

使用最后一次迭代(第20次,这里步长 $\eta=0.05$)计算的系数作为最佳的系数组代入函数,其结果如图5.5所示。

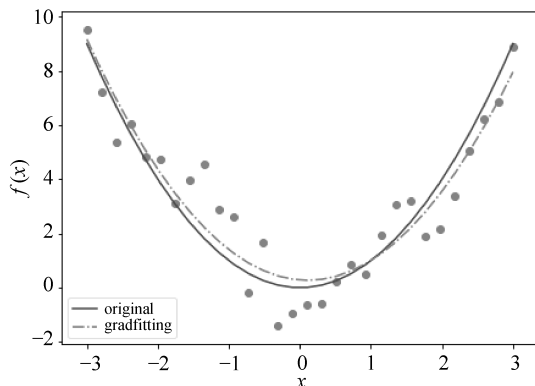


图 5.5 梯度下降法迭代 20 次的拟合函数(gradfitting)和原函数(original)图形

相应的损失函数值关于迭代步骤的曲线如图5.6所示。

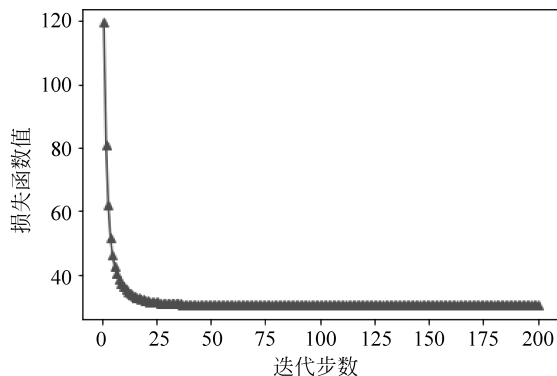


图 5.6 损失函数迭代图

图 5.5 和图 5.6 的代码实现如下所示。

```

1  # 拟合函数图
2  fig = plt.figure(figsize=(7, 5))
3  plt.plot(x, f, 'r-', label='original')           # 函数图像
4  plt.scatter(x, f_noise)                          # 噪声点
5  plt.plot(x, f_prob(x_tensor, *coeff_value).numpy(), 'g-.', label='gradfitting')
                                                    # 拟合函数

6  plt.legend()
7  plt.xlabel('$x$')
8  plt.ylabel('$f(x)$')
9  # 损失函数迭代图
10 plt.plot(list(range(1, len(loss_result_list) + 1)), loss_result_list, 'r-^')
11 plt.xlabel("迭代步数")
12 plt.ylabel("损失函数值")

```

这里简单讨论蒙特卡洛法和梯度下降法。通过以上实例不难看出,蒙特卡洛法更像是以“猜”或“试”的方法来求最值,但前提要确定待求系数的取值范围,这样才能保证“猜”的可靠性。梯度下降法是每次迭代时都找一个“最快”的方式来更新上一步的计算(给定)的待求系数,因此梯度下降法在计算成本上更有优势。

这里将所有的散点序列 $\{(x_i, y_i)\}$ 在每次迭代过程中都进行考虑,若每次随机采样(重复采样)来进行梯度计算,称为随机梯度下降法(stochastic gradient descent, SGD)。随机梯度下降法可以有效抑制过拟合问题,关于随机梯度法的内容不再阐述,读者可以查阅相关资料了解随机梯度法的相关定义和内容。除此之外,还有批量梯度下降法(batch gradient descent, BGD)和小批量梯度下降法(mini-batch gradient descent, MBGD)。

5.2.3 封装实现

前面通过自编代码来实现模型的构建和计算,那通过 PyTorch 模块如何实现呢? 这里给出其对应的代码。

```

1  x = x_tensor.float()           # 自变量
2  y = torch.from_numpy(f_noise).float()   # 因变量
3  # 定义拟合函数
4  def f_prob(x, w):
5      return w[0] * x ** 2 + w[1] * x + w[2]
6  # 初始化参数
7  w = torch.ones(3, requires_grad=True)
8  # 损失函数
9  criterion = torch.nn.MSELoss()
10 # 优化函数
11 optimizer = torch.optim.SGD([w], lr=0.05)   # 随机梯度下降法学习率 0.05
12 # 遍历
13 for iter_n in range(20):
14     loss = criterion(f_prob(x, w), y)
15     optimizer.zero_grad()                 # 初始化梯度

```



```

16     loss.backward()
17     optimizer.step()
18     if iter_n % 2 == 0:
19         print("迭代步数", iter_n, '损失值', loss.data, "参数变化", w.data)
                                     # 输出 ↓
20 迭代步数 0 损失值 tensor(4.1230) 参数变化 tensor([0.7178, 0.7045, 0.9046])
21 迭代步数 2 损失值 tensor(1.7554) 参数变化 tensor([0.7573, 0.3673, 0.8268])
22 迭代步数 4 损失值 tensor(1.1360) 参数变化 tensor([0.7910, 0.2117, 0.7552])
23 迭代步数 6 损失值 tensor(0.9190) 参数变化 tensor([0.8199, 0.1399, 0.6892])
24 迭代步数 8 损失值 tensor(0.8090) 参数变化 tensor([0.8448, 0.1068, 0.6285])
25 迭代步数 10 损失值 tensor(0.7373) 参数变化 tensor([0.8663, 0.0915, 0.5727])
26 迭代步数 12 损失值 tensor(0.6851) 参数变化 tensor([0.8849, 0.0844, 0.5214])
27 迭代步数 14 损失值 tensor(0.6452) 参数变化 tensor([0.9011, 0.0812, 0.4743])
28 迭代步数 16 损失值 tensor(0.6142) 参数变化 tensor([0.9151, 0.0797, 0.4310])
29 迭代步数 18 损失值 tensor(0.5898) 参数变化 tensor([0.9274, 0.0790, 0.3912])

```

将计算后的结果系数(参数) $w = [0.9274, 0.0790, 0.3912]$ 代入公式, 并绘制图片, 其结果如图 5.7 所示。

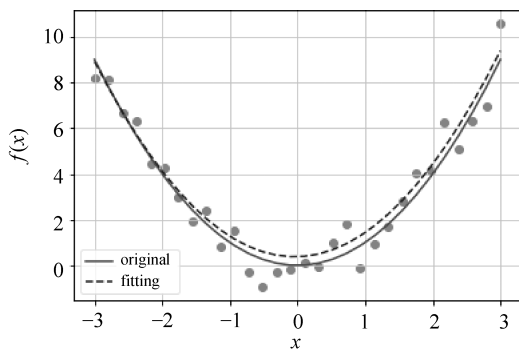


图 5.7 拟合函数与原函数对比图

这里通过 19 次迭代来求解参数, 因此最终结果与上面代码运算的结果略有差别, 但只要迭代次数足够大, 其结果差异性就会很小。

5.2.4 激活函数

激活函数(activation function)又称非线性函数(nonlinear function)。激活函数是类似于神经元与神经元传递信息时的一种信息处理手段。激活函数具有以下几种性质: ①非线性; ②单调性; ③值域有界性。

这里介绍 3 个常见的激活函数: sigmoid、tanh 和 ReLU。

1. 常见的激活函数

1) sigmoid

$$g(x) = \frac{1}{1 + e^{-x}} \quad (5.7)$$

2) tanh

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (5.8)$$

3) ReLU

$$\text{relu}(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} \quad (5.9)$$

通过 Python 对以上 3 个函数进行绘图,其代码如下所示,图像如图 5.8 所示。

```
1  # sigmoid 函数
2  def g(x):
3      return 1 / (1 + torch.exp(-x))
4  # tanh 激活函数
5  def tanh(x):
6      a1 = torch.exp(x)
7      a2 = torch.exp(-x)
8      return (a1 - a2) / (a1 + a2)
9  # ReLU 激活函数
10 def relu(x):
11     return torch.clamp_min(x, 0)
12
13 x = torch.linspace(-5, 5, 100)
14 fig, axes = plt.subplots(1, 3, figsize=(12, 5))
15 axes[0].plot(x.numpy(), g(x).numpy(), 'r-.', label='sigmoid')
16 axes[1].plot(x.numpy(), tanh(x).numpy(), 'g', label='tanh')
17 axes[2].plot(x.numpy(), relu(x).numpy(), 'b', label='relu')
```

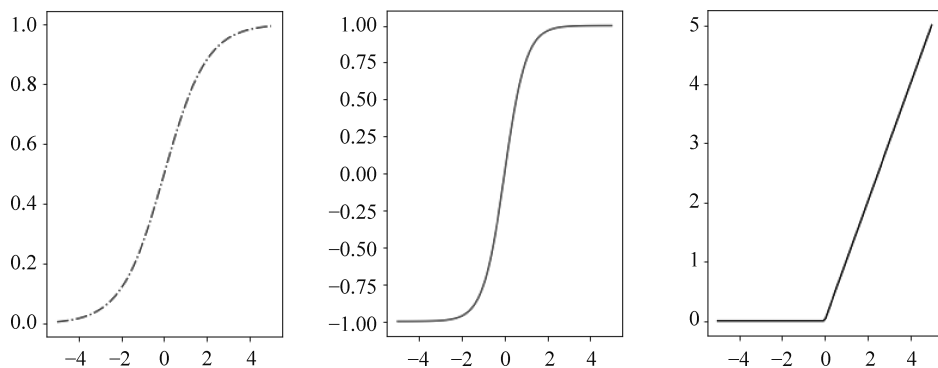


图 5.8 激活函数图

2. 激活函数的优缺点

以上 3 个激活函数都有各自的优缺点,这里对其简要介绍。

1) sigmoid 函数

优点:

- 值域为 $(0,1)$,单调连续,优化稳定;
- 易求导。

缺点:

- 易造成计算的梯度消失,从而导致训练出现问题;
- 其输出不是以0为中心(zero-centered)。

2) tanh 函数

优点:

- 相比于 sigmoid 函数,其收敛速度更快;
- 其输出是以0为中心。

缺点: 由于饱和性产生的梯度消失(同 sigmoid 函数)。

注意: 对于函数 $f(x)$, 当 $x \rightarrow -\infty$ 时, 其导数 $f'(x) \rightarrow 0$, 则称为左饱和, 相应地, 还有右饱和。若左右都满足饱和, 称为两端饱和。

3) ReLU 函数

优点:

- 计算复杂度低,不需要进行指数运算;
- 收敛速度比 sigmoid 和 tanh 快。

缺点:

- 输出不是以0为中心;
- 不会对数据做幅度压缩,因此数据幅度会随着模型层数的增加不断扩大。

除了以上函数、还有 ELU、SELU、Threshold、PReLU 和 Leaky ReLU 等激活函数,这里不再一一阐述。

5.2.5 softmax

softmax 函数又称归一化指数函数,是逻辑函数中的一种。Softmax 函数本质上是对有限项离散概率分布的梯度对数归一化^①。softmax 函数在人工神经网络的多分类问题中有着非常广泛的应用。其意图是将一个含有任意实数的 k 维向量 $\mathbf{z}$, 映射到另一个 k 维实向量中,但其元素范围为 $(0,1)$, k 个元素的和为1。

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{i=1}^k e^{z_i}} \quad (5.10)$$

其 Python 代码实现非常简单,如下所示。

```
1 # 导入模块包
2 import NumPy as np
3 # 构建数组
4 z = np.array([1.0, 2.0, 1.0, 5.0, 6.0])
5 print(np.exp(z) / sum(np.exp(z)))
```

^① <https://zh.wikipedia.org/wiki/softmax> 函数。



视频讲解

5.3 前馈神经网络

5.3.1 思想原理

人工神经网络(artificial neural network, ANN)简称神经网络^①。人工神经网络有很多类型,这里仅阐述一些经典的方法。下面先介绍全连接神经网络,其简单的流程如图 5.9 所示。

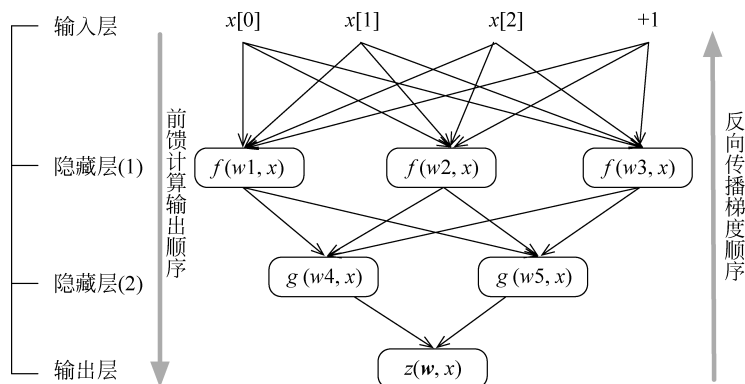


图 5.9 全连接神经网络

如图 5.9 所示,这是一个全连接神经网络(fully connected neural network),全连接神经网络是仅由全连接层组成的前馈神经网络。图 5.9 含有输入层、隐藏层(2 个)和输出层,这里有 3 个特征 $x_i (i=0, 1, 2)$ 和 1 个常数项 b , 存在一个权重向量 $\mathbf{w} = (w_0, w_1, w_2)$, 记

$$f(w_1, x) = w_0 x_0 + w_1 x_1 + w_2 x_2 + b = \sum_{i=0}^2 w_i x_i + b \quad (5.11)$$

式(5.11)是第一个隐藏层(含有 3 个神经元),第一个隐藏层到第二个隐藏层传递之前,需要通过激活函数 $h(x)$ 计算 $h(f(w_1, x))$, 然后作为“输入元”再进行类似于式(5.11)的处理 $g(w_4, h(f(w_1, x)))$, 最终通过输出层来计算出输出结果。

前馈神经网络有以下特点:

- 每层含有不同数量的神经元,同层之间的神经元无连接;
- 相邻两层之间的神经元全部两两连接;
- 整个网络中无反馈环节,可以理解为一个有向无环图。

5.3.2 手写体识别实例

这里通过经典数据集——手写体(MNIST)数据集来进行实验,不妨先随机取出 6 个样本进行观察,其图像如图 5.10 所示。

^① <https://zh.wikipedia.org/wiki/人工神经网络>。

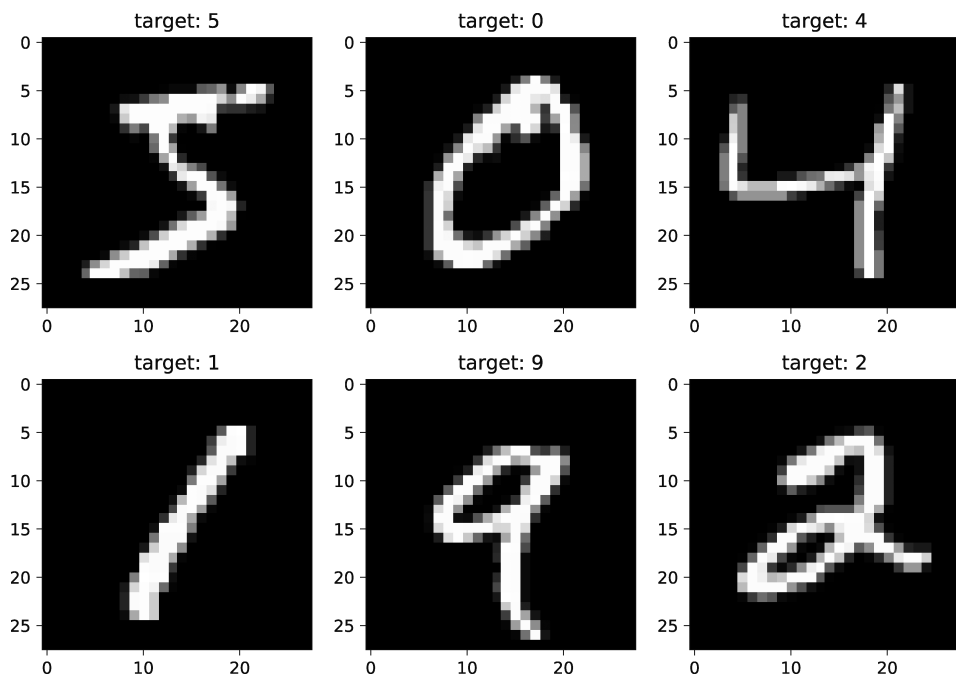


图 5.10 6 个样本的图像数据

MNIST 数据集是一个手写体数据集,共有 10 分类标签,每一个样本的标签值为 0~9 的一个数。训练集有 60 000 个样本,测试集有 10 000 个样本,每个样本都可以看成 28×28 的矩阵,因此将其展平成 1×784 向量,即 784 个变元。

下面通过 PyTorch 模块对其进行神经网络训练,先导入模块以及基本参数设置。

```

1  # 模块导入
2  import torch
3  import torch.nn as nn
4  import torchvision
5  import torchvision.transforms as transforms
6  # 基本参数设置
7  input_size = 784          # 变量数量
8  num_classes = 10         # 分类数目
9  num_epochs = 5           # 迭代次数
10 batch_size = 100         # 训练批次
11 learning_rate = 1e-3     # 学习率

```

torchvision 模块作为 PyTorch 的辅助模块,不仅包含经典的深度学习模型,而且含有常见的数据集,现在下载并导入 MNIST 数据集。

```

1  data_path = '../data/MNIST'          # 在 data 路径下创建 MNIST 文件
2  # 训练集若没有则下载

```

```

3  train_dataset = torchvision.datasets.MNIST(
4      root = data_path,                # 数据集路径
5      train = True,                    # 是否为训练集
6      transform = transforms.ToTensor(), # 将数据张量化处理
7      download = True)                # 进行下载
8  # 测试集
9  test_dataset = torchvision.datasets.MNIST(
10     root = data_path,
11     train = False,
12     transform = transforms.ToTensor(),
13     download = True)
14  # 构建训练数据加载器
15  train_loader = torch.utils.data.DataLoader(dataset = train_dataset,
16                                              batch_size = batch_size,
17                                              shuffle = True)
18  # 测试数据加载器
19  test_loader = torch.utils.data.DataLoader(dataset = test_dataset,
20                                           batch_size = batch_size,
21                                           shuffle = False)

```

通过 PyTorch 模块进行数据处理时,该数据集的输入格式都已处理好,其中 train_loader 为数据加载器,它可以很好地进行批处理运算。通过 PyTorch 先搭建一个极简的全连接网络,即输入层到输出层。

```

1  # 线性模型
2  model = nn.Linear(input_size, num_classes) # 输出 ↓
3  Linear(in_features = 784, out_features = 10, bias = True)

```

这是一个非常简单的模型,即输入 784 个参数和 1 个偏置项(bias),输出为 10 个分类数目。截止到目前,关于该数据集的预处理和模型构造已完成,下面开始进行模型训练。

```

1  # 损失函数
2  # nn.CrossEntropyLoss() 内部集成了 softmax 函数
3  criterion = nn.CrossEntropyLoss()
4  # 优化方式: 随机梯度下降法
5  optimizer = torch.optim.SGD(model.parameters(), lr = learning_rate)
6  # 训练模型: 遍历
7  for epoch in range(num_epochs):
8      for i, (images, labels) in enumerate(train_loader):
9          # 转换数据格式
10         images = images.reshape(-1, 28 * 28) # 28x28 → 1x784
11         # 前向传播
12         outputs = model(images) # 导入数据,输出计算结果值
13         loss = criterion(outputs, labels) # 代价函数(损失函数)

```

```

14         # 反向传播及优化
15         optimizer.zero_grad()           # 初始化梯度
16         loss.backward()                 # 计算代价函数的损失函数
17         optimizer.step()               # 更新梯度
18         # 打印部分日志
19         if (i + 1) % 300 == 0:
20             print("Epoch: {0}, Step: {1}, loss: {2}".format(epoch + 1, i + 1, loss.item()))
21     # 输出 ↓
22     Epoch: 1, Step: 300, loss: 1.999853491783142
23     Epoch: 1, Step: 600, loss: 1.8600431680679321
24     Epoch: 2, Step: 300, loss: 1.6646180152893066
25     Epoch: 2, Step: 600, loss: 1.4261541366577148
26     Epoch: 3, Step: 300, loss: 1.3930109739303589
27     Epoch: 3, Step: 600, loss: 1.3381340503692627
28     Epoch: 4, Step: 300, loss: 1.1542218923568726
29     Epoch: 4, Step: 600, loss: 1.2540661096572876
30     Epoch: 5, Step: 300, loss: 1.0619515180587769
31     Epoch: 5, Step: 600, loss: 0.9185544848442078

```

下面通过测试集来验证模型的训练情况,其代码如下所示。

```

1     # PyTorch 默认每一次前向传播都会计算梯度
2     with torch.no_grad():
3         correct = 0                       # 初始化正确数量
4         total = 0                       # 初始化总数
5         for images, labels in test_loader:
6             images = images.reshape(-1, 28 * 28)   # 测试集格式
7             outputs = model(images)               # 代入训练好的模型
8             _, predicted = torch.max(outputs.data, 1) # 取最大值
9             total += labels.size(0)
10            correct += (predicted == labels).sum()   # 正确数
11        print('Accuracy of the model on the 10000 test images: {} %'.format(100 * correct/total))
12    # 输出 ↓
13    Accuracy of the model on the 10000 test images: 83 %

```

这里通过这个全连接神经网络模型可以获得 83% 的准确率。从其代码结构上不难发现是非常简洁的,易读性很高。若读者对其感到生疏,建议查阅官方的 PyTorch 文档进行学习。

有时候构建的神经网络比较复杂,建议继承 `torch.nn.Module` 类来构建神经网络,其代码构建是非常方便的。

```

1     # 导入模块
2     import torch.nn as nn
3     import torch.nn.functional as F
4     # 构建一个 MnistNet 类

```

```

5 class MnistNet(nn.Module):
6     def __init__(self, input_size, hidden_size, num_classes):
7         super(MnistNet, self).__init__()
8         self.fc1 = nn.Linear(input_size, hidden_size)
9         self.fc2 = nn.Linear(hidden_size, num_classes)
10    def forward(self, x):
11        x = self.fc1(x)           # 全连接
12        x = F.relu(x)             # ReLU 激活函数
13        x = self.fc2(x)           # 全连接
14        return x                 # 输出结果

```

以上自定义类都做了哪些工作？下面简要阐述。

- 导入必要的模块；
- 构建名为 MnistNet 的类，并继承 torch.nn.Module 类；
- 定义两个线性函数 fc1, fc2；
- 重构前馈神经网络函数 forward；
- 前馈神经网络中先对输入数据通过第一个线性函数线性计算，然后通过激活函数 ReLU 处理，再通过第 2 个线性函数进行线性计算，将最后的计算结果输出。

现在将对象实例化，其代码如下所示。

```

1 model1 = MnistNet(784, 20, 10)   # 输入层：784，隐含层：20 个神经元，输出层：10
2 model1                           # 输出 ↓ 神经网络结构
3 MnistNet(
4     (fc1): Linear(in_features = 784, out_features = 20, bias = True)
5     (fc2): Linear(in_features = 20, out_features = 10, bias = True)
6 )

```

构建相应的损失函数和梯度下降法，并对模型进行训练，其代码如下所示。

```

1 criterion = nn.CrossEntropyLoss()
2 optimizer1 = torch.optim.SGD(model1.parameters(), lr = learning_rate)
3 # 训练模型
4 total_step = len(train_loader)
5 for epoch in range(num_epochs):
6     for i, (images, labels) in enumerate(train_loader):
7         # 转换数据格式
8         images = images.reshape(-1, 28 * 28)
9         # 前向传播
10        outputs = model1(images)
11        loss = criterion(outputs, labels)
12        # 反向传播及优化
13        optimizer1.zero_grad()
14        loss.backward()

```



```

15         optimizer1.step()
16         # 打印部分日志
17         if (i + 1) % 300 == 0:
18             print("Epoch: {0}, Step: {1}, loss: {2}".format(epoch + 1, i + 1, loss.item()))

```

这里不再给出打印结果,不难看出 PyTorch 模块搭建神经网络是非常系统和简洁的。这也是 PyTorch 模块发行后备受欢迎的主要原因之一。

5.4 卷积神经网络



视频讲解

卷积神经网络(convolutional neural network, CNN)是一种具有局部连接、权重共享等特性的深层前馈神经网络。CNN 算法主要用于图像处理问题,因为图像可以看成含有多个颜色通道的二维矩阵以及特征处理过程中需要满足视野域的概念,但也可以用于一维数据。卷积的主要功能是在给定的图像上滑动一个卷积核(卷积核函数,又称滤波器),通过卷积计算后获得一组新的特征。

全连接前馈神经网络中,其权重矩阵的参数非常多致使训练效率低下。

5.4.1 核函数

介绍卷积神经网络之前,非常有必要介绍卷积核函数。卷积核函数有很多种,这里介绍 5 种常见的 3×3 卷积核函数。

1. identity

identity 卷积核是非常简单的。

$$\mathbf{kernel} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad (5.12)$$

该卷积核函数不改变图像的任何元素,不妨给定一个图像(单通道或灰度)的矩阵 $\mathbf{I}$,如下所示。

$$\mathbf{I} = \begin{pmatrix} 8 & 0 & 0 & 2 & 1 & 0 & 0 & 0 & 3 \\ 0 & 0 & 5 & 14 & 2 & 0 & 0 & 3 & 0 \\ 0 & 3 & 0 & 13 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 6 & 3 & 0 & 13 & 3 & 7 & 2 \\ 0 & 16 & 0 & 18 & 7 & 0 & 11 & 3 & 0 \\ 0 & 0 & 0 & 1 & 1 & 7 & 0 & 0 & 11 \\ 12 & 6 & 0 & 20 & 0 & 7 & 0 & 19 & 15 \\ 2 & 0 & 5 & 0 & 0 & 12 & 14 & 0 & 8 \\ 0 & 4 & 19 & 0 & 0 & 4 & 5 & 0 & 0 \end{pmatrix} \quad (5.13)$$

对矩阵 $\mathbf{I}$ 进行卷积函数处理时,其计算方式是非常简单的(暂不考虑边界元素),

$I_{i=1,j=1}$ 的分块矩阵

$$\begin{pmatrix} 8 & 0 & 0 \\ 0 & 0 & 5 \\ 0 & 3 & 0 \end{pmatrix} \quad (5.14)$$

$I_{i=1,j=1}$ 与卷积核对应元素相乘后再相加:

$$0 \times 8 + 0 \times 0 + 0 \times 0 + 0 \times 0 + 1 \times 0 + 0 \times 5 + 0 \times 0 + 0 \times 3 + 0 \times 0 = 0$$

即通过卷积核 kernel 函数在图像 $I_{i=1,j=1}$ 位置的值为 0。

这里给定一个图像,其图像读取效果如图 5.11 所示。



图 5.11 原图(源于 MATLAB 内置图像)

以上图像是一个含有 3 个渠道(R、G、B)的 256×256 的方形图像,其代码如下所示。

```
1  # 模块
2  import numpy as np
3  import torch
4  from PIL import Image, ImageShow
5  from torchvision import transforms
6  import matplotlib.pyplot as plt
7  %matplotlib inline
8  # 读取图像
9  path = "../chap3/girl.png"
10 girl_pic = Image.open(path)
11 girl_pic # 输出 ↓
```

将以上图像转换成数组,这里通过 NumPy 模块的 array 成员函数将图像数据转换成数组类型。

```
1  np.array(girl_pic).shape # 输出 ↓
2  (256, 256, 3)
3  type(girl_pic) # 输出 ↓
4  PIL.PngImagePlugin.PngImageFile
```

```

5  # 数据(width, height, channel) 转换成(channel, width, height)
6  data2channel_tensor = transforms.functional.to_tensor(np.array(girl_pic))
7  data2channel_tensor.shape          # 输出 ↓
8  torch.Size([3, 256, 256])

```

下面通过卷积核函数 kernel1 对图像的矩阵 **I** 做处理,其代码如下。

```

1  kernel_tensor = torch.FloatTensor(3,3,3)
2  for i in range(3):
3      # 核函数
4      kernel_tensor[i] = torch.Tensor([[0,0,0],[0,1,0],[0,0,0]])
5
6  # 构建一个等同, 元素全为 0 的数组
7  new_data_tensor = torch.zeros_like(data2channel_tensor)
8  c, m, n = data2channel_tensor.size()          # 渠道, 宽, 高
9  for layer in range(3):                        # 遍历渠道
10     tmp_data = data2channel_tensor[layer]
11     kernel = kernel_tensor[layer]
12     # 遍历计算
13     for i in range(1, m-1):
14         for j in range(1, n-1):
15             ele_data = tmp_data[i-1:i+2,j-1:j+2]
16             # 计算值结果
17             result_value = float((ele_data * kernel.float()).sum().numpy())
18             new_data_tensor[layer][i, j] = result_value

```

2. edge detection

边缘检测(edge detection)卷积核矩阵有很多形式,常见的 3 种形式如下所示。

$$\begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{pmatrix} \quad (5.15)$$

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} \quad (5.16)$$

$$\begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix} \quad (5.17)$$

通过以上 3 种边缘检测卷积核矩阵对图像的矩阵 **I** 分别处理,步长设置为 1,其结果如图 5.12 所示。

通过图 5.12 不难看出,其边缘轮廓的差异非常明显。



图 5.12 边缘检测卷积核函数处理结果

3. sharpen

锐化(sharpen)卷积核函数如下所示。

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix} \quad (5.18)$$

其结果如图 5.13 所示。



图 5.13 锐化卷积核函数处理结果

4. box blur

盒子模糊(box blur)是将 3×3 的矩阵元素全部设置为 $\frac{1}{9}$, 可以看作一种简单的加权平均方式, 其卷积和函数如下所示。

$$\begin{pmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{pmatrix} \quad (5.19)$$

关于其效果图这里不再表述,因为其效果图和下面要阐述的高斯模糊比较相近。

5. Gaussian blur

高斯模糊(Gaussian blur)是一种比较常见的模糊处理方法,它考虑到了距离问题,距离中心元素最近的4个元素全部为 $\frac{2}{16}$,剩余的4个角元素全部为 $\frac{1}{16}$,中心元素数值为 $\frac{4}{16}$ 。

$$\begin{pmatrix} \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \\ \frac{2}{16} & \frac{4}{16} & \frac{2}{16} \\ \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \end{pmatrix} \quad (5.20)$$

通过高斯模糊处理好的图像相比于原图有一定的模糊,但是高斯模糊可以在一定程度上有效降低高斯噪声,其图像如图 5.14 所示。

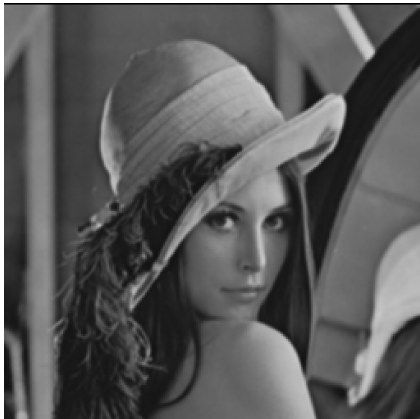


图 5.14 高斯模糊卷积核处理结果

除了以上5种常见的 3×3 卷积核函数,还有其他类似的 5×5 和 7×7 等卷积核函数。读者可以自行查阅相关资料深入了解和学习。

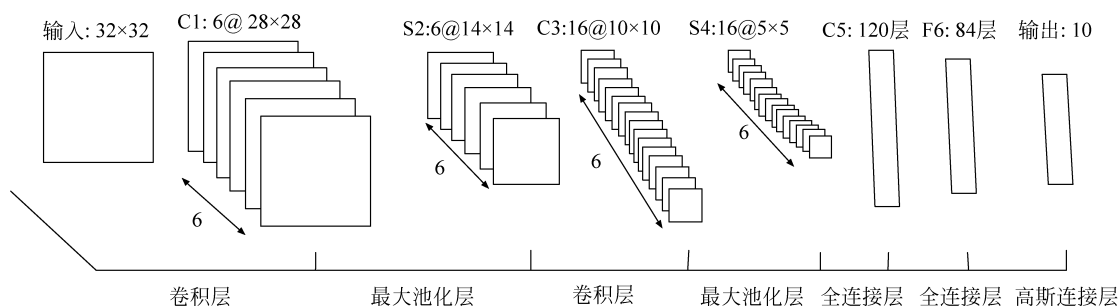
5.4.2 池化层

除了上面的卷积核函数和激活函数,卷积网络中另一个重要的结构是池化层(pooling)。其本质是将图片变小,达到一种降维的目的,从而有效提高计算效率,并能保留原图像相应的特征。池化层没有任何参数,易于实现,其形式也有多种,比如最大值池化、最小值池化、均值池化等,并且其尺寸大小以及步长可进行人为调整或设置。卷积网络中采用最大池化层来做处理。

5.4.3 LeNet

LeNet 神经网络由深度学习三巨头之一的 Yan LeCun 提出,该方法标志着 CNN 的

真正面世。LeNet 主要用于手写体的识别与分类,并在美国银行得到广泛应用。由于当时计算机硬件不佳,以及缺少大规模的训练集,致使 LeNet 模型在处理较复杂的问题时不太理想,但是其思想方法非常值得借鉴。LeNet 神经网络结构如图 5.15 所示。



根据图 5.15 的神经网络,不难看出它涉及 2 个卷积层和 2 个全连接层,其 PyTorch 代码也是非常简单的,如下所示。

```

1  # 导入模块
2  import torch.nn as nn
3  import torch.nn.functional as F
4  # 构建一个 LeNet 类
5  class LeNet(nn.Module):
6      def __init__(self):
7          super(LeNet, self).__init__()
8          self.tmp_dict = {}
9          # 3: 输入图片单通道, 6: 输出通道数, 5: 卷积核为 5 * 5
10         self.conv1 = nn.Conv2d(3, 6, 5)
11         self.conv2 = nn.Conv2d(6, 16, 5)
12         # 全连接层 y = wx + b
13         self.fc1 = nn.Linear(16 * 5 * 5, 120)
14         self.fc2 = nn.Linear(120, 84)
15         self.fc3 = nn.Linear(84, 10)
16     def forward(self, x):
17         # 卷积 -> 激活 -> 池化
18         x = self.conv1(x)          # 5x5 卷积核
19         x = F.relu(x)              # ReLU 激活函数
20         x = F.max_pool2d(x, 2)     # 最大池化层 2x2
21         x = self.conv2(x)          # 卷积 6 个神经元扩展到 16 个神经元, 5x5 卷积核
22         x = F.relu(x)              # 激活函数
23         x = F.max_pool2d(x, 2)     # 2x2 池化层 # S4
24         x = x.view(x.size()[0], -1) # 数据展平 C5
25         x = F.relu(self.fc1(x))    # 线性模型 1
26         x = F.relu(self.fc2(x))    # 线性模型 2
27         return self.fc3(x)         # 输出层

```

现在根据以上构建的神经网络代码,对 LeNet 算法进行阐述:

- 输入层：含有 3 个颜色通道的图像大小为 32×32 ；
- C1 层：6 个 5×5 的卷积核，即 6 组 28×28 特征映射结果；
- S2 层：使用最大池化层 2×2 ，生成 6 组 14×14 个神经元；
- C3 层：由 6 组扩展到 16 组大小为 10×10 的特征映射，这里采用了部分连接（非全连接），使用了 60 个 5×5 的卷积核，若是全连接则是 $6 \times 16 = 96$ 个卷积核；
- S4 层：继续采用 2×2 的池化层，得到 16 组 5×5 的特征映射；
- C5 层：对数据进行展平，得到 120 个 1×1 的特征映射，使用 16×120 个 5×5 个卷积核；
- F6 层：这是一个全连接层，共有 84 个神经元；
- 输出层：输出 10 个类别数组结果。

5.4.4 AlexNet

AlexNet 由 Hilton 的学生 Alex Krizhevsky 提出，它可被视为首个现代深度卷积网络模型，主要体现在使用 GPU 进行并行训练、利用 Dropout 防止过拟合。AlexNet 在 2012 年的 ImageNet 图像分类大赛中获得冠军。AlexNet 模型如图 5.16 所示。

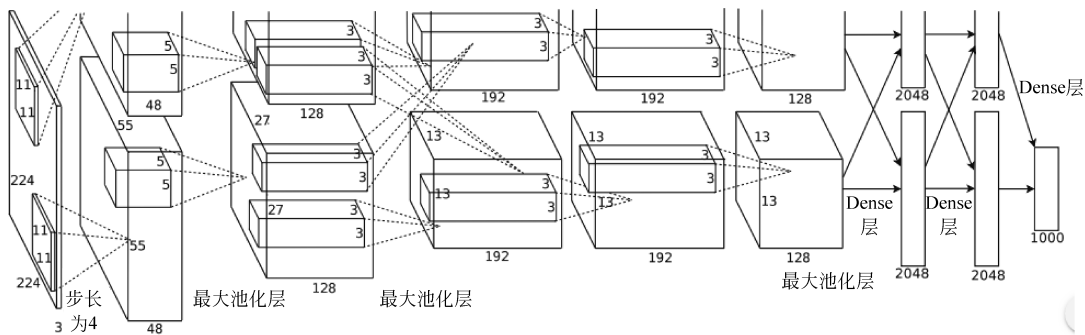


图 5.16 AlexNet 模型(图片来源见参考文献 [11])

关于 AlexNet 模块的代码如下所示。

```
1 import torch.nn as nn
2 import torchvision.transforms as transforms
3 class AlexNet(nn.Module):
4     def __init__(self, num_classes):
5         super(AlexNet, self).__init__()
6         self.features = nn.Sequential(
7             # (227 - 11 + 2 * 2) / 4 = 55
8             nn.Conv2d(3, 96, kernel_size=11, stride=4, padding=2),
9             # 输出 55 * 55 * 96
10            nn.ReLU(inplace=True),
11            nn.MaxPool2d(kernel_size=3, stride=2),
```

```

12         nn.Conv2d(64, 256, kernel_size=5, padding=2),
13         nn.ReLU(inplace=True),
14         nn.MaxPool2d(kernel_size=3, stride=2),
15         nn.Conv2d(192, 384, kernel_size=3, padding=1),
16         nn.ReLU(inplace=True),
17         nn.Conv2d(384, 256, kernel_size=3, padding=1),
18         nn.ReLU(inplace=True),
19         nn.Conv2d(256, 256, kernel_size=3, padding=1),
20         nn.ReLU(inplace=True),
21         nn.MaxPool2d(kernel_size=3, stride=2)
22     )
23     self.classifier = nn.Sequential(
24         nn.Dropout(),
25         nn.Linear(256 * 6 * 6, 4096),
26         nn.ReLU(inplace=True),
27         nn.Dropout(),
28         nn.Linear(4096, 4096),
29         nn.ReLU(inplace=True),
30         nn.Linear(4096, num_classes)
31     )
32     def forward(self, x):
33         x = self.features(x)
34         x = x.view(x.size(0), 256 * 6 * 6)
35         x = self.classifier(x)
36         return x

```

5.4.5 ResNet

2015 年,在 ImageNet 竞赛中获得冠军的是由微软亚洲研究院的研究员设计的更加简单的网络 ResNet,该模型可有效地解决深度神经网络难以训练的问题^①,可训练高达 1000 层的卷积网络。由于网络资源中有很多现成的介绍,这里不再一一赘述。

```

1  from torch import nn
2  import torch as t
3  from torch.nn import functional as F
4  class ResidualBlock(nn.Module):
5      '''
6      实现子模块: Residual Block
7      '''
8      def __init__(self, in_channel, out_channel, stride = 1, short_cut = None):
9          super(ResidualBlock, self).__init__()

```

① 网络之所以难以训练,是因为存在梯度消失的问题。


```
10         self.left = nn.Sequential(
11             nn.Conv2d(in_channel, out_channel, 3, stride, padding = 1, bias = False),
12             nn.BatchNorm2d(out_channel),
13             nn.ReLU(inplace = True),
14             nn.Conv2d(out_channel, out_channel, 3, 1, 1, bias = False),
15             nn.BatchNorm2d(out_channel)
16         )
17         self.right = short_cut
18     def forward(self, x):
19         out = self.left(x)
20         residual = x if self.right is None else self.right(x)
21         out += residual
22         return F.relu(out)
23 class ResNet(nn.Module):
24     '''
25     主模块: ResNet34
26     '''
27     def __init__(self, num_classes = 1000):
28         super(ResNet, self).__init__()
29         # 前几层图像转换
30         self.pre = nn.Sequential(
31             nn.Conv2d(3, 64, 7, 2, 3, bias = False),
32             nn.BatchNorm2d(64),
33             nn.ReLU(inplace = True),
34             nn.MaxPool2d(3, 2, 1))
35         # 重复 layer: 分别为 3, 4, 6, residual block
36         self.layer1 = self._make_layer(64, 128, 3)
37         self.layer2 = self._make_layer(128, 256, 4, stride = 2)
38         self.layer3 = self._make_layer(256, 512, 6, stride = 2)
39         self.layer4 = self._make_layer(512, 512, 3, stride = 2)
40         # 分类用的全连接
41         self.fc = nn.Linear(512, num_classes)
42     def _make_layer(self, in_channel, out_channel, block_num, stride = 1):
43         '''
44         构建 layer, 包含多个 residual block
45         :param in_channel:
46         :param out_channel:
47         :param block_num:
48         :param stride:
49         :return:
50         '''
51         shortcut = nn.Sequential(
52             nn.Conv2d(in_channel, out_channel, 1, stride, bias = False),
53             nn.BatchNorm2d(out_channel)
54         )
55         layers = []
56         layers.append(ResidualBlock(in_channel, out_channel, stride, shortcut))
```

```

57         for i in range(1, block_num):
58             layers.append(ResidualBlock(out_channel, out_channel))
59         return nn.Sequential(*layers)
60     def forward(self, x):
61         '''
62         :param x: 样本
63         :return: 前反馈数据结果
64         '''
65         x = self.pre(x)
66         x = self.layer1(x)
67         x = self.layer2(x)
68         x = self.layer3(x)
69         x = self.layer4(x)
70         x = F.avg_pool2d(x, 7)
71         x = x.view(x.size(0), -1)
72         return self.fc(x)

```

关于 ResNet 神经网络的变型较多,比如 ResNet18、ResNet3、ResNet50、ResNet101 以及 ResNet152。ResNet 神经网络在处理实际问题中具有较强的表现能力,很大程度归功于 Residual Block。

5.4.6 GoogLeNet

VGG 在 2014 年的 ImageNet 比赛中获得了亚军,冠军为 GoogLeNet,其模型由 Google 的研究人员提出。该模型在当时影响非常大,其根源在于颠覆了对卷积神经网络的常规想法。除此之外,它采用了一种非常高效的 inception 模块,得到了比 VGG 更深的网络结构,并且其参数比 VGG 的参数更少。这里给出高效的 inception 模块示意图,如图 5.17 所示。

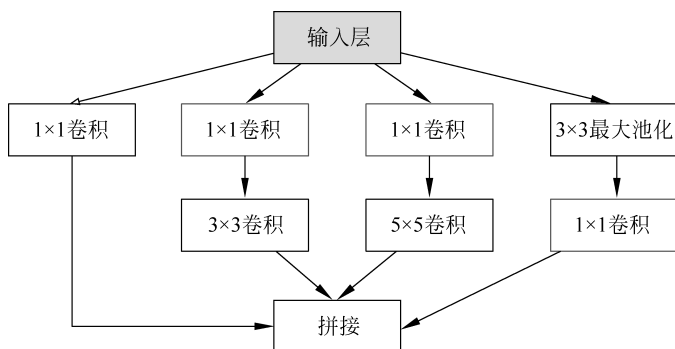


图 5.17 inception 模块示意图

图 5.17 含有 4 个并行线路:

- 一个 1×1 的卷积,一个小的感受野进行卷积提取特征;
- 一个 1×1 的卷积加上一个 3×3 的卷积, 1×1 的卷积降低输入的特征通道,减少参数计算量,然后接一个 3×3 的卷积,做一个较大感受野的卷积;

- 一个 1×1 的卷积加上一个 5×5 的卷积,做一个更大感受野的卷积;
- 一个 3×3 的最大池化加上一个 1×1 的卷积,最大池化改变输入的特征序列, 1×1 的卷积进行特征提取。

最终将 4 个并行线路得到的特征进行拼接,再进行下一步处理。

5.4.7 垃圾分类实例

垃圾分类问题是当下研究的一个热点问题,这里通过自行构建的神经网络(与 LeNet 神经网络非常相似)来实验垃圾分类问题。本实例的样本数据来源于网络资源^①,共有 6 个分类:纸板(箱)(cardboard)、玻璃瓶(glass)、金属类(metal)、纸质类(paper)、塑料类(plastic)以及垃圾(trash),其训练集的各类样本数量如表 5.1 所示。

表 5.1 垃圾样本对应的样本量

label	cardboard	glass	metal	paper	plastic	trash
N	403	501	410	594	482	137

通过表 5.1 不难发现,训练集的各类样本数量并不均衡,但这里并不想深入探究这个问题。现在随机读取几个样本观察其图像,如图 5.18 所示。



图 5.18 训练集样本图像

图 5.18 中含有 6 张图像,对应的类别分别为: cardboard、glass、metal、paper、plastic 以及 trash,由于训练集的数据都处理成尺寸大小一致的图像,因此不需要再次处理。下面通过 PyTorch 来进行数据预处理。

```

1  from torchvision import transforms, datasets
2  from torch.utils.data import Dataset, DataLoader
3  import torch.optim as optim
4  import torch.nn.functional as F
5  path = "./dataset-resized/"                # 训练集数据路径

```

^① 链接为 https://pan.baidu.com/s/1rWl_odFKFAnNBrlNUxDo8g, 密码: uhv8。

```

6  data_transform = transforms.Compose([
7      transforms.Resize(84),           # 重置成尺寸大小为 84 × 84 的图像
8      transforms.CenterCrop(84),       # 剪切
9      transforms.ToTensor(),           # 转换成张量形式
10     transforms.Normalize(mean = [0.485, 0.456, 0.406], std = [0.229, 0.224, 0.225])
                                     # 标准化处理
11 ])
12 train_dataset = datasets.ImageFolder(root = path, transform = data_transform)

```

这里借助 torchvision 中 transforms 来进一步处理数据,比如设置尺寸、剪切、张量化处理以及标准化处理等,利用 dataset 来导入数据集。其效果可以通过打印 train_dataset 来查看。其代码如下所示。

```

1  print(train_dataset)                # 输出 ↓
2  Dataset ImageFolder
3      Number of datapoints: 2527
4      Root location: ./dataset-resized/
5      StandardTransform
6  Transform: Compose(
7      Resize(size = 84, interpolation = PIL.Image.BILINEAR)
8      CenterCrop(size = (84, 84))
9      ToTensor()
10     Normalize(mean = [0.485, 0.456, 0.406], std = [0.229, 0.224, 0.225])
11 )

```

完成训练集数据的处理工作,下面就需要对其进行训练方式的设置,比如以什么样的方式来进行训练模型。这里选择以批次为 2(batch_size=2)、每次迭代的批次样本顺序随机变动(shuffle=True),以及采用 2 个进程(num_workers=2)来训练,其代码实现是非常简单的,如下所示。

```

1  train_loader = torch.utils.data.DataLoader(train_dataset, batch_size = 2,
                                             shuffle = True, num_workers = 2)

```

下面给定自定义的神经网络,其代码如下所示。

```

1  class MyNet(nn.Module):
2      def __init__(self):
3          super(MyNet, self).__init__()
4          self.features = nn.Sequential(
5              nn.Conv2d(in_channels = 3, out_channels = 18, kernel_size = 5, stride = 1,
                        dilation = 1),
6              nn.ReLU(inplace = True),
7              nn.MaxPool2d(kernel_size = 2, stride = 2),

```

```

8         nn.Conv2d(in_channels = 18, out_channels = 30, kernel_size = 5, stride = 1,
                    dilation = 1),
9         nn.ReLU(inplace = True),
10        nn.MaxPool2d(kernel_size = 2, stride = 2)
11    )
12    self.linear = nn.Sequential(
13        nn.Linear(in_features = 30 * 18 * 18, out_features = 1024),
14        nn.ReLU(inplace = True),
15        nn.Linear(in_features = 1024, out_features = 512),
16        nn.ReLU(inplace = True),
17        nn.Linear(in_features = 512, out_features = 6)
18    )
19    def forward(self, x):
20        x = self.features(x)
21        x = x.view(-1, 30 * 18 * 18)
22        x = self.linear(x)
23        return x

```

由于数据量较大,现在通过 GPU 进行加速训练,其代码如下所示。

```

1  net = MyNet().cuda()
2  # 函数
3  criterion = nn.CrossEntropyLoss()
4  # 优化函数
5  optimizer = optim.SGD(net.parameters(), lr = 0.0001)
6  # 迭代遍历
7  for epoch in range(100):
8      for i, data in enumerate(train_loader, 0):
9          inputs, labels = data                # x, y
10         outputs = net(inputs.cuda())          # 训练, 输出预测\hat{y}
11         loss = criterion(outputs.cpu(), labels) # 损失函数
12         optimizer.zero_grad()
13         loss.backward()
14         optimizer.step()
15         loss_val = loss.data.numpy()
16         if i % 10 == 0:
17             print('epoch:[ %d | %d], loss: %.3f' % (epoch + 1, i + 1, loss_val))

```

该模型采用随机梯度下降法来进行批次训练,并且迭代次数(epoch)为100,这里不考虑这些设置是否可以获得本问题的最优解,仅考虑整个神经网络的正常运行。截至目前,即完成了整个训练过程,net中的参数即是待求系数。关于模型的验证这里不再赘述,通过PyTorch的相关模块看一下整个模型的构造,如图5.19所示。

图5.19的神经网络模型结构图可以通过PyTorch相应的代码来实现,其代码也非常简单,如下所示。

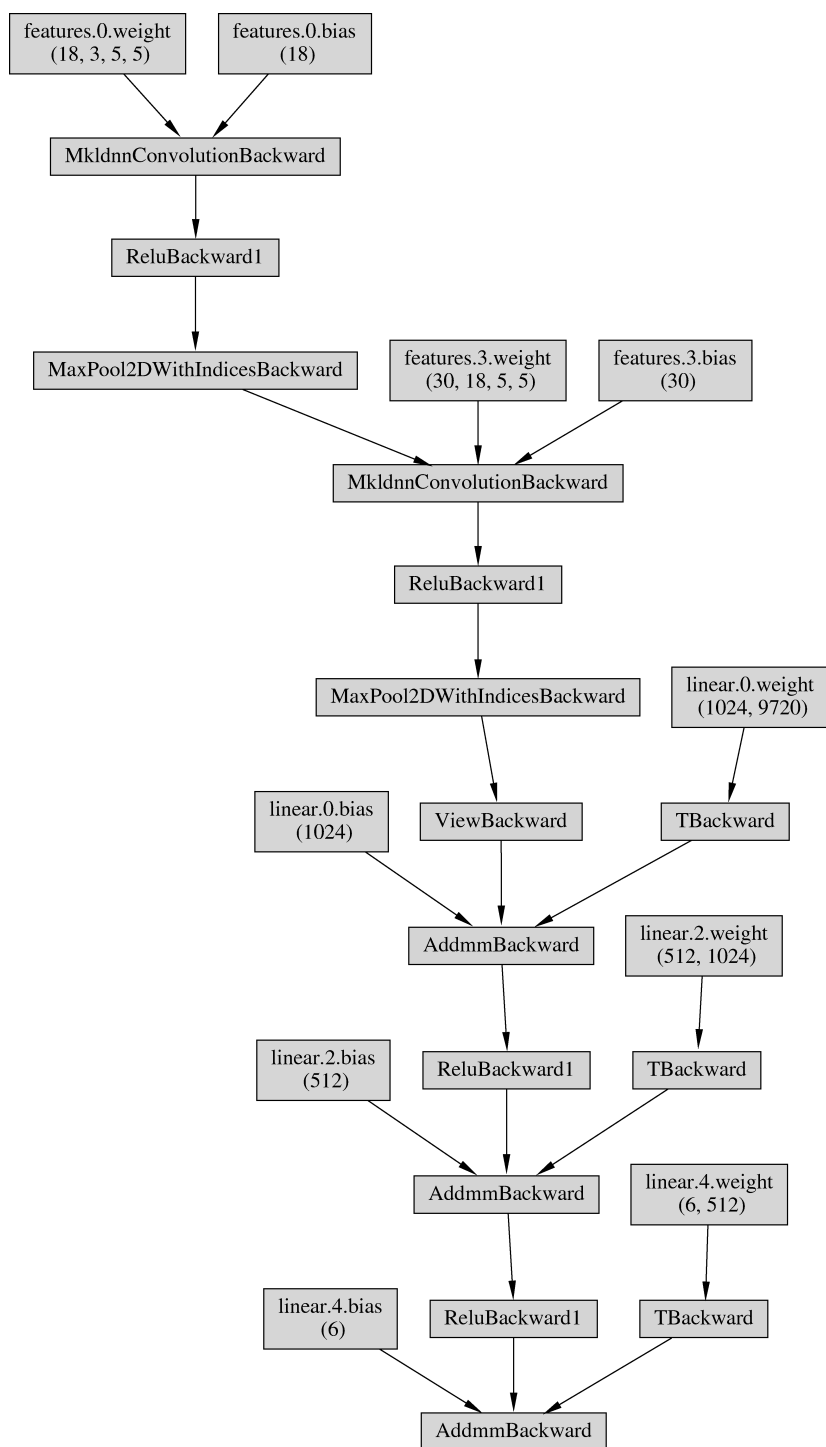


图 5.19 MyNet 神经网络模型

```

1  from torchviz import make_dot
2  from torch.autograd import Variable
3  # 随机生成一个满足图片尺寸大小和数据结构类型的随机张量
4  x = Variable(torch.randn(1, 3, 84, 84))
5  vis_graph = make_dot(net(x), params=dict(net.named_parameters()))
6  # 将神经网络结构模型保存成 pdf 文件格式
7  vis_graph.view()

```

待模型训练完成后,需要对其进行保存,代码如下所示。

```

1  # 保存和加载整个模型, 包括网络结构、模型参数等
2  torch.save(net, 'MyNet_model.pkl')
3  # 加载已训练好的模型
4  model = torch.load('MyNet_model.pkl')
5  # 保存和加载网络中的参数
6  torch.save(net.state_dict(), 'params.pkl')
7  # 加载已训练好的模型
8  resnet.load_state_dict(torch.load('params.pkl'))

```

5.5 生成对抗网络

5.5.1 思想原理

生成对抗网络(generative adversarial net, GAN)作为深度学习的热门方向之一,深受研究学者和部分市场的欢迎。Ian Goodfellow(称为 GAN 之父)在 2014 年首次提出生成对抗网络,并给出了一个通过 GAN 实现的手写数字。GAN 可以有效解决非监督学习的一个有名的问题:指定一定规模的样本,训练一个系统能够生成类似的新样本。

下面给出 GAN 的流程图,如图 5.20 所示。

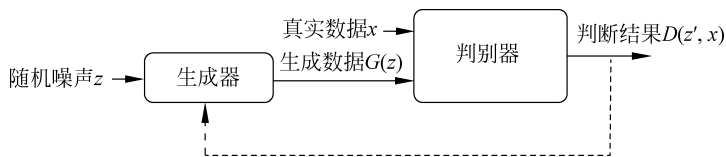


图 5.20 GAN 流程图

根据图 5.20,给出其优化目标函数:

$$\min_{G} \max_{D} V(D, G) = E_{x \sim p_{\text{data}}(x)} [\log D(x)] + E_{z \sim p_z(z)} [\log(1 - (D(G(z))))] \quad (5.21)$$

GAN 模型在训练期间容易陷入损失函数值 NaN 现象, G 网络和 D 网络在训练过程中要保持在平衡状态, 若一方训练过好会致使另一方训练无法进行。相关研究人员在 GAN 的基础上提出了多种变种模型, 比如 WGAN、DCGAN、CGAN 和 LSGAN 等。感兴趣的读者可以通过文献检索来进行了解和学习。

5.5.2 对抗网络实例

通过对手写字段进行训练得到新的样本集,其结果如图 5.21 所示。

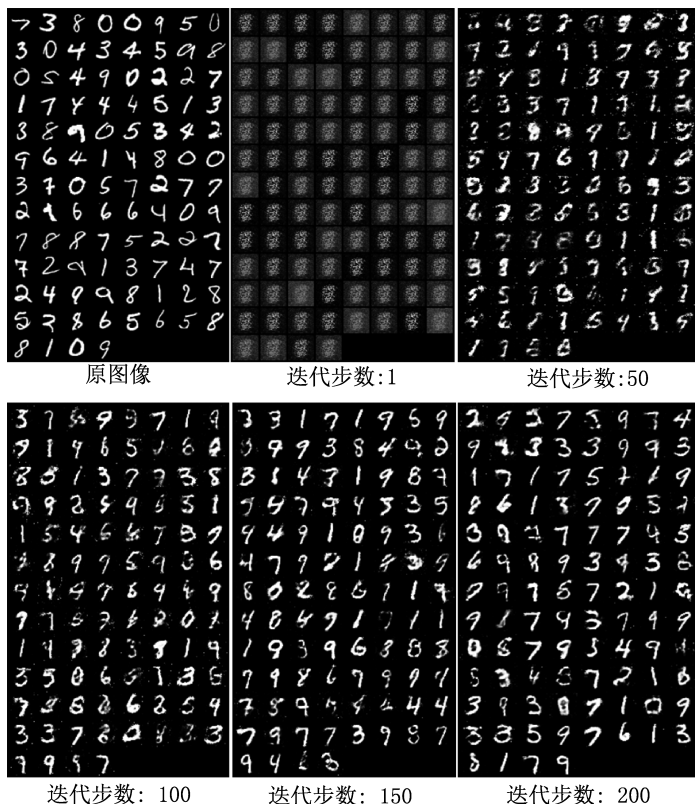


图 5.21 手写字体原图像和部分训练次数结果

通过图 5.21 的结果不难看出, GAN 模型在训练过程生成的样本与原样本非常相近, 并且训练次数越高其相似度越大, 从而到达真假难分的目的。其代码(该代码借鉴了 github 资源)如下所示。

```

1  # 导入模块
2  import torch
3  import torch.nn as nn
4  from torchvision import datasets
5  from torchvision import transforms
6  from torchvision.utils import save_image
7  from torch.autograd import Variable
8  from torch.utils.data import DataLoader
9  # 对图像进行处理
10 def img_norm(x):
11     '''分段函数'''
12     out = (x + 1) / 2

```



```
13     return out.clamp(0, 1)
14 # 标准化, 均值为 0.5, 标准差为 0.5
15 transform = transforms.Compose([
16     transforms.ToTensor(),
17     transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.05,0.05,0.05))])
18 # 下载数据
19 path = "../data"
20 mnist = datasets.MNIST(
21     root = path,
22     train = True,
23     transform = transform,
24     download = True
25 )
26 # 数据批量, 随机批次
27 train_loader = DataLoader(mnist, batch_size=100, shuffle=True)
28 # 加载数据, 对输入数据做线性变换
29 D = nn.Sequential(
30     nn.Linear(784, 256),
31     # max(0, x) + 0.2 * min(0, x)
32     nn.LeakyReLU(0.2),
33     nn.Linear(256, 128),
34     nn.LeakyReLU(0.2),
35     nn.Linear(128, 1),
36     nn.Sigmoid()
37 )
38 # 生成数据的取值范围与真实数据相似
39 G = nn.Sequential(
40     nn.Linear(64, 128),
41     nn.LeakyReLU(0.2),
42     nn.Linear(128, 256),
43     nn.LeakyReLU(0.2),
44     nn.Linear(256, 784),
45     nn.Tanh()
46 )
47 # 损失函数以及优化函数
48 criterion = nn.BCELoss()
49 d_optimizer = torch.optim.Adam(D.parameters(), lr=1e-6)
50 g_optimizer = torch.optim.Adam(G.parameters(), lr=1e-6)
51 # 遍历
52 for epoch in range(200):
53     for i, (images, _) in enumerate(train_loader): # 不需要 label
54         batch_size = images.size(0)
55         # 转成 784 * 1 向量
56         images = Variable(images.view(batch_size, -1))
57         # 构建元素全为 1 的矩阵
58         real_labels = Variable(torch.ones(batch_size))
59         # 元素全为 0 的矩阵
60         fake_labels = Variable(torch.zeros(batch_size))
```

```

61         outputs = D(images)
62         d_loss_real = criterion(outputs.view(1, -1), real_labels)
63         real_score = outputs
64         # 构建随机矩阵
65         z = Variable(torch.randn(batch_size, 64))
66         # 先 G 网络后 D 网络
67         fake_images = G(z)
68         outputs = D(fake_images)
69         d_loss_fake = criterion(outputs.view(1, -1), fake_labels)
70         fake_score = outputs
71         d_loss = d_loss_real + d_loss_fake
72         D.zero_grad()
73         d_loss.backward()
74         d_optimizer.step()
75         z = Variable(torch.randn(batch_size, 64))
76         # 先 G 网络后 D 网络
77         fake_images = G(z)
78         outputs = D(fake_images)
79         g_loss = criterion(outputs.view(1, -1), real_labels)
80         D.zero_grad()
81         G.zero_grad()
82         g_loss.backward()
83         g_optimizer.step()
84         if (i + 1) % 300 == 0:
85             print("epoch: ", epoch, "d_loss: ", d_loss_real.data.numpy(),
86                   fake_score.data.mean())
87
88     # 对图片进行训练
89     if (epoch + 1) == 1:
90         images = images.view(images.size(0), 1, 28, 28)
91         save_image(img_norm(images.data), './real_images.png')
92     fake_images = fake_images.view(fake_images.size(0), 1, 28, 28)
93     save_image(img_norm(fake_images.data), './fake_image_{0}.png'.format(epoch + 1))

```

5.6 其他神经网络

5.6.1 循环神经网络

循环神经网络(recurrent neural network,RNN)是一种具有短期记忆功能的神经网络。不同于前馈神经网络和卷积神经网络,RNN 不仅可接收其他神经元传递的信息,也可接收自身的历史信息,从而形成一个具有环路的网络结构。RNN 在语音识别和自然语言领域有广泛的应用。这里只介绍一种简单的循环网络。

给定一个时间序列 $x = (x^1, x^2, \dots, x^t)$, 其更新并含有反馈的函数为:

$$h^t = f(h^{t-1}, x^t) \quad (5.22)$$

其中, $h^0 = 0$, f 可以是一个非线性函数,也可以是一个前馈网络。式(5.22)可视为一种动

力系统。理论上,循环神经网络在一定程度上可以视为非线性动力系统。

注意:式(5.22)形式上类似于统计学中的自回归模型(autoregressive module, AR)。

循环神经网络的流程图如图 5.22 所示。

关于循环神经网络的其他内容在这里不再进行过多阐述,若读者对循环神经网络深感兴趣,可以查阅其他相关的书籍或与自然语言(NLP)相关的书籍。

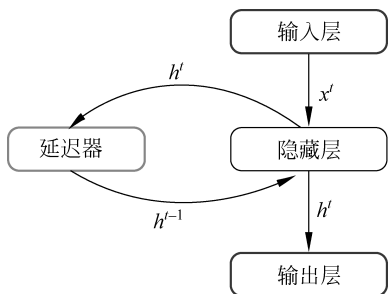


图 5.22 循环神经网络的流程图

5.6.2 风格迁移神经网络

风格迁移(neural style)神经网络是由德国图宾根大学的 Bethge 实验室的 3 个研究员 Leon Gatys、Alexander Ecker 和 Matthias Bethge 于 2015 年提出的一种算法。其算法实现过程非常的复杂,处理像素比较大的图片时往往需要几个小时甚至几十个小时的训练。2016 年斯坦福大学的 Justin Johnson、Alexandre Alahi 和李飞飞提出了一种快速风格迁移算法。该方法结合 GPU 可以快速完成训练。

由于网上有很多关于该算法的优质代码,读者可以通过 github 网络查询优质开源代码,这里不再对其进行赘述。这里给出训练的一组校园图片,如图 5.23 所示。



图 5.23 风格图像(星空)

图 5.23 是大画家梵高的星空。杭州师范大学(仓前校区)的图书馆照片如图 5.24 所示。

通过风格迁移神经网络的算法原理,将风格图像(图 5.23)迁移到图像 5.24 中,其训



图 5.24 杭州师范大学(仓前校区)图书馆一角

练 100 次的结果如图 5.25 所示。



图 5.25 风格图像迁移训练 100 次的效果图

5.7 本章小结

本章主要介绍了 PyTorch 模块和部分神经网络模型。PyTorch 在学术研究方面要优于 TensorFlow,再者它与 NumPy 模块可以很方便地进行转换,这也是使用它做神经

网络的重要原因之一,但在大型的神经网络项目部署中,建议采用 TensorFlow。神经网络历史悠久,但是其兴盛时期也就在最近几年,其涉及领域广泛,应用前景非常大,不同的模型用于不同的实际问题,更甚者同一问题可以通过多种神经网络模型来研究。若读者对神经网络非常感兴趣,可以查阅斯坦福大学李飞飞老师的相关教程,因此这里不再详细阐述关于神经网络的相关内容。