1 使用 Python 编写神经网络

徐辉，xuh@fudan.edu.cn

本章学习目标：

- ⋆⋆⋆ 理解神经网络的基本工作原理，包括网络结构、前向传播与反向传播机制。
- ⋆⋆⋆ 能够阅读并理解使用 Python 编写的基础神经网络模型代码。
- ⋆⋆⋆ 能够利用神经网络对简单数学函数进行建模与拟合。

1.1 问题定义与分析

本节以阶乘问题为例讲解如何用 Python 构建神经网络预测阶乘值。

问题 给定任意一个 x，其阶乘 $x!$ 的计算方式为

$$x! = \prod_{i=1}^{x} i.$$

如何通过神经网络模型预测 x 的阶乘?

分析 从数学角度看，函数可以形式化为集合之间的映射。阶乘函数定义为从自然数集合到自然数集合的映射：

$$f : \mathbb{N} \to \mathbb{N}, \quad f(x) = x!.$$

在实际建模过程中，直接对 $x!$ 进行预测会面临数值大小迅速增长的问题。为缓解这一困难，通常对目标函数进行对数变换，将问题转化为对 $\log(x!)$ 的预测：

$$\log(x!) = \log\left(\prod_{i=1}^{x} i\right) = \sum_{i=1}^{x} \log i.$$

该变换将原本的乘法运算转化为加法形式，从而显著缓解输出数值的增长速率，使其更适合神经网络进行建模。我们可以使用神经网络构建 $x \to \log f(x)$ 的映射。

一般而言，利用神经网络解决这类问题通常包括以下几个步骤：

1) **模型构建**：设计神经网络的输入形式、网络结构及输出表示；
2) **模型训练**：基于实际数据调整模型参数，最小化模型输出与真实值之间的差异；
3) **效果验证**：评估模型的预测准确度和泛化能力。

1.2 模型构建

1.2.1 网络结构

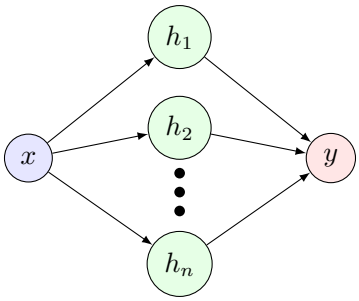


图 1.1: 前馈神经网络示例

我们采用前馈神经网络（feed-forward neural network）对该问题进行建模。前馈神经网络结构比较简单，其信息自输入层向输出层单向传播。图 1.1是一个前馈神经网络，该模型由一个输入节点、一个隐藏层以及一个输出节点组成，其中隐藏层包含 $h_1, h_2, \ldots, h_n$ 个神经元。每个隐藏层神经元执行如下非线性变换：

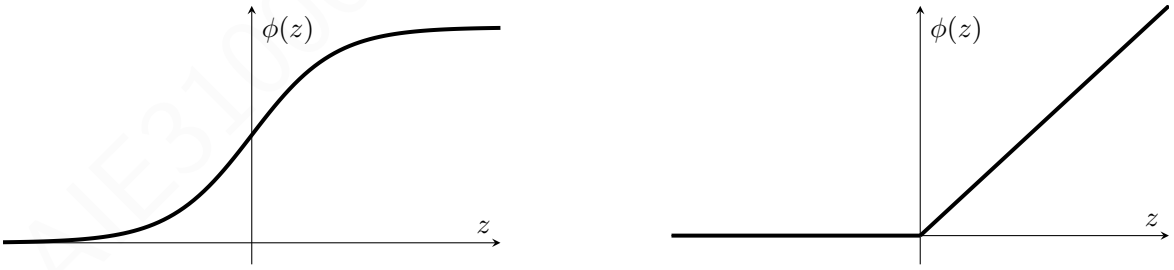
$$h_i = \sigma(w1_i \cdot x + b1_i),$$

其中 $\sigma(\cdot)$ 表示激活函数。输出层采用线性形式，对所有隐藏层神经元的输出进行加权求和作为 $\log(x!)$ 的预测结果。

$$\hat{y} = \sum_{i=1}^{n} w2_i \cdot h_i + b2$$

1.2.2 激活函数

激活函数 $\sigma(\cdot)$ 通过对输入的线性组合进行非线性变换，从而为模型引入非线性能力，使得神经网络能够逼近复杂的非线性关系。若不引入激活函数，即使堆叠多个神经元，整个模型仍然等价于一个线性模型。常见的激活函数包括 Sigmoid 函数和 ReLU（Rectified Linear Unit）。



(a) Sigmoid 激活函数　　(b) ReLU 激活函数

图 1.2: 常见激活函数示意图

***Sigmoid* 函数**　Sigmoid 函数的定义为：

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

其输出值严格位于 $(0, 1)$ 区间内，因此常被解释为概率值。

ReLU ReLU 在输入为负时输出零，在输入为正时呈线性增长，其定义为：

$$\sigma(z) = \max(0, z)$$

1.2.3 Python 实现

在本节中，我们使用纯 Python（不依赖任何深度学习框架）实现上述模型。我们将模型设计为一个 `class`，并将其名称定义为 `FactorialNN`。接下来，我们先设计模型的主要规格和初始化方法，然后再定义其前向传播过程。

模型初始化

代码 1.1 定义了模型类 `FactorialNN`。其中，`__init__` 函数是 Python 语言中的类构造器，用于在对象创建时对模型结构进行初始化。该函数接收 `hidden_size` 参数，用于定义隐藏层的神经元个数。例如，可以通过 `FactorialNN(20)` 创建一个模型实例，该代码会自动调用 `__init__` 函数，并将隐藏层神经元个数设置为 20。也可以在创建实例时省略该参数，即直接调用 `FactorialNN()`，则模型将采用默认设置，此时隐藏层神经元个数为 10。

```
class FactorialNN:
    def __init__(self, hidden_size=10):
        """
        __init__ 是 Python 类的构造函数（constructor），
        在创建对象时会被自动调用，用于初始化对象的成员变量。

        参数:
            hidden_size (int): 隐藏层神经元的个数。
                               若未显式指定，则默认取值为 10。
        """

        # 将传入的隐藏层规模保存为对象属性
        self.hidden_size = hidden_size

        # ----------------------------------------------------
        # 输入层 -> 隐藏层 的权重和偏置初始化
        # ----------------------------------------------------

        # w1[i] 表示从输入 x 到第 i 个隐藏层神经元的权重
        # 使用列表推导式生成 hidden_size 个随机初始权重
        # 等价于如下 for 循环（仅为说明，非实际执行代码）：
        #
        #   w1 = []
        #   for i in range(hidden_size):
        #       w1.append(random.uniform(-1, 1))
        #
        # 列表推导式在语义上完全等价，但写法更加简洁、清晰
        # random.uniform(-1, 1) 表示在 [-1, 1] 区间内均匀采样
        self.w1 = [random.uniform(-1, 1) for _ in range(hidden_size)]

        # b1[i] 表示第 i 个隐藏层神经元对应的偏置项
        # 偏置同样采用随机初始化
        self.b1 = [random.uniform(-1, 1) for _ in range(hidden_size)]
```

```
        # ------------------------------------------------
        # 隐藏层 -> 输出层 的权重和偏置初始化
        # ------------------------------------------------

        # w2[i] 表示第 i 个隐藏层神经元到输出层神经元的连接权重
        self.w2 = [random.uniform(-1, 1) for _ in range(hidden_size)]

        # b2 为输出层神经元的偏置项（标量）
        self.b2 = random.uniform(-1, 1)
```

代码 1.1: 定义神经网络模型结构

在 `__init__` 函数中，我们定义并随机初始化了模型各层之间的权重和偏置。其中，列表 `w1` 和 `b1` 分别表示输入层到隐含层的权重和偏置，`w2` 和 `b2` 则表示隐含层到输出层的权重和偏置。`uniform` 函数用于从指定区间内的均匀分布中随机采样初始参数值，在本例中权重和偏置均初始化为区间 $[-1, 1]$ 内的随机数，以避免所有神经元在初始阶段具有完全相同的行为。另外，若将权重初始化为区间过大，容易导致前向传播时激活值过大，使非线性函数进入饱和区，从而在反向传播阶段产生梯度消失等问题。

前向传播

下面，我们为 `FactorialNN` 定义前向传播的计算过程，即代码 1.2中的 `forward` 函数。该函数描述了数据从输入层 → 隐含层 → 输出层的逐层计算过程。

```
class FactorialNN:
    ...
    def forward(self, x):
        # 参数:
            x: 模型输入，对应一个样本

        # ---------- 隐藏层 ----------
        # z 用于存储隐藏层应用激活函数之前每个神经元的线性输出
        # 用于反向传播计算梯度
        z = []
        # h 用于存储隐藏层每个神经元的输出
        h = []

        # 依次计算每个隐藏神经元的输出
        for i in range(self.hidden_size):
            # 线性变换
            # z_i = w1[i] * x + b1[i]
            # 其中:
            #   w1[i]: 输入到第 i 个隐藏神经元的权重
            #   b1[i]: 第 i 个隐藏神经元的偏置
            zi = self.w1[i] * x + self.b1[i]
            z.append(zi)
            # 引入非线性激活函数，使网络能够表示非线性函数
            # 若使用 sigmoid 激活函数，可替换为: h.append(sigmoid(z))
            h.append(relu(zi))


        # ---------- 输出层 ----------
        # 输出层对隐藏层的输出进行线性组合
```

```
        # y = sum(w2[i] * h[i]) + b2
        # 其中：
        #   w2[i]：第 i 个隐藏神经元到输出层的权重
        #   b2：输出层偏置
        y = sum(self.w2[i] * h[i] for i in range(self.hidden_size)) + self.b2

        # 返回：
        #   z：隐藏层应用激活函数之前每个神经元的线性输出（用于ReLU反向传播）
        #   h：隐藏层所有神经元的输出（用于Sigmoid反向传播）
        #   y：模型的最终预测结果
        return z, h, y
```

代码 1.2: 前向传播函数

1.3 模型训练

模型训练的目的是得到一组参数值 θ，使神经网络能够尽可能准确地预测输入 x 对应的输出 y。训练过程通常通过最小化预测值 $\hat{y}$ 与真实值 y 之间的误差来实现，并利用梯度下降（Gradient Descent）对模型参数进行迭代优化。在此过程中，反向传播（Backpropagation）算法被用来计算损失函数对各参数的梯度，从而指导参数更新。

1.3.1 反向传播

反向传播是神经网络训练的核心机制，其本质是利用链式求导法则，将输出误差逐层向前传播，从而计算各层参数对损失函数的梯度。反向传播算法的核心思想是：

1) 误差计算：根据预测值 $\hat{y}$ 与真实值 y，计算损失 $\mathcal{L}(\hat{y}, y)$。

2) 梯度计算：利用链式法则逐层计算损失函数对各参数 θ_i 的偏导数，从而得到梯度向量 $\nabla_\theta \mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial \theta_1}, \frac{\partial \mathcal{L}}{\partial \theta_2}, \dots\right)$。

3) 参数更新：根据梯度和学习率更新参数。

误差计算

在回归任务中，均方误差（Mean Squared Error, MSE）是一种常用的误差度量方式，其定义为：

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2,$$

其中 y_i 表示第 i 个样本的真实值，$\hat{y}_i$ 表示模型对该样本的预测值。

通过对误差进行平方，MSE 不仅消除了误差的正负号影响，还对较大的预测偏差给予更高的惩罚。此外，平方运算保证了损失函数在整个定义域内可导，使其能够与梯度下降和反向传播算法自然结合，从而实现高效的参数优化。

为了说明梯度的计算，我们可以先考虑单个样本的平方误差（Squared Error, SE）：

$$\mathcal{L}_{\text{SE}_i} = (\hat{y}_i - y_i)^2.$$

对该单样本误差求偏导，得到：

$$\frac{\partial \mathcal{L}_{\text{SE}_i}}{\partial \hat{y}_i} = 2(\hat{y}_i - y_i),$$

在省略常数系数 2 时，可以简化为：

$$\frac{\partial \mathcal{L}_{\text{SE}_i}}{\partial \hat{y}_i} = \hat{y}_i - y_i.$$

此外，平均绝对误差（Mean Absolute Error, MAE）也是一种常用的误差统计方法，其定义为：

$$\mathcal{L}_{\text{MAE}} = \frac{1}{N}\sum_{i=1}^{N}|\hat{y}_i - y_i|.$$

MAE 通过对预测误差取绝对值来消除正负号的影响。若只考虑单个样本的绝对误差（Absolute Error, AE），其对预测值 $\hat{y}_i$ 的偏导为：

$$\frac{\partial \mathcal{L}_{\text{AE}_i}}{\partial \hat{y}_i} = \begin{cases} -1, & y < \hat{y}; \\ \text{undefined}, & \hat{y} = y; \\ 1, & y > \hat{y}. \end{cases}$$

梯度计算

在神经网络训练过程中，我们的目标是计算损失函数对模型参数的偏导数，从而利用梯度下降法对参数进行更新。以本章定义的阶乘预测模型为例，训练过程中需要计算的梯度包括

$$\frac{\partial \mathcal{L}}{\partial w1_i}, \quad \frac{\partial \mathcal{L}}{\partial b1_i}, \quad \frac{\partial \mathcal{L}}{\partial w2_i}, \quad \frac{\partial \mathcal{L}}{\partial b2}$$

需要注意的是，损失函数 $\mathcal{L}$ 并不是模型参数的显式函数，而是通过一系列中间变量（如隐藏层的输出以及模型的预测值 $\hat{y}$）间接依赖这些参数。因此，参数梯度的计算需要借助链式求导法则。

以输出层权重 $w2_i$ 为例，其对损失函数的影响路径可以表示为：

$$w2_i \longrightarrow \hat{y} \longrightarrow \mathcal{L}.$$

其中输出层的计算形式为：

$$\hat{y} = \sum_{i=1}^{H} w2_i \cdot h_i + b2.$$

根据链式法则，有：

$$\frac{\partial \mathcal{L}}{\partial w2_i} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w2_i}.$$

假设采用均方误差作为损失函数，可得：

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \hat{y} - y, \qquad \frac{\partial \hat{y}}{\partial w2_i} = h_i.$$

因此，输出层权重的梯度为：

$$\frac{\partial \mathcal{L}}{\partial w2_i} = (\hat{y} - y) \cdot h_i.$$

同理，可以计算：

$$\frac{\partial \mathcal{L}}{\partial b2} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b2} = \hat{y} - y.$$

对于输入层权重 $w1_i$，其对损失函数的影响需要经过多层计算，其依赖路径为：

$$w1_i \longrightarrow z_i \longrightarrow h_i \longrightarrow \hat{y} \longrightarrow \mathcal{L}.$$

其中

$$z_i = w1_i \cdot x + b1_i, \qquad h_i = \sigma(z_i)$$

根据链式求导法则，损失函数对 $w_{1,i}$ 的偏导数可写为：

$$\frac{\partial \mathcal{L}}{\partial w1_i} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial h_i} \cdot \frac{\partial h_i}{\partial z_i} \cdot \frac{\partial z_i}{\partial w1_i}.$$

其中各项局部导数分别为：

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \hat{y} - y, \qquad \frac{\partial \hat{y}}{\partial h_i} = w2_i, \qquad \frac{\partial h_i}{\partial z_i} = \sigma'(z_i), \qquad \frac{\partial z_i}{\partial w1_i} = x.$$

其中，$\sigma'(z_i)$ 为激活函数的导数。将上述结果代入，可得：

$$\frac{\partial \mathcal{L}}{\partial w1_i} = (\hat{y} - y) \cdot w2_i \cdot \sigma'(z_i) \cdot x.$$

同理，可以计算

$$\frac{\partial \mathcal{L}}{\partial b1_i} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial h_i} \cdot \frac{\partial h_i}{\partial z_i} \cdot \frac{\partial z_i}{\partial b1_i} = (\hat{y} - y) \cdot w2_i \cdot \sigma'(z_i).$$

若激活函数采用 ReLU，其定义为 $\mathrm{ReLU}(z)=\max(0,z)$。因此，其关于 z 的导数为分段函数：

$$\frac{d}{dz}\mathrm{ReLU}(z)=\begin{cases}1, & z>0;\\ 0, & z<0.\end{cases}$$

注意，在 $z=0$ 处，ReLU 函数不可导。实际的神经网络实现中，通常约定：

$$\frac{d}{dz}\mathrm{ReLU}(0)=0.$$

因此，在使用 ReLU 激活函数时，反向传播中的梯度传播表现为一种 “门控” 机制。当神经元被激活（$z>0$）时，梯度可以无衰减地通过；当神经元未被激活（$z\le 0$）时，梯度被直接截断。

如果激活函数采用 Sigmoid，其计算方式为 $\sigma(z)=\frac{1}{1+e^{-z}}$。对该函数关于 z 求导，有

$$\frac{d}{dz}\sigma(z)=\frac{d}{dz}(1+e^{-z})^{-1}=-(1+e^{-z})^{-2}\cdot(-e^{-z})=\frac{e^{-z}}{(1+e^{-z})^2}.$$

由于

$$\sigma(z)=\frac{1}{1+e^{-z}},\qquad 1-\sigma(z)=\frac{e^{-z}}{1+e^{-z}},$$

因此

$$\frac{e^{-z}}{(1+e^{-z})^2}=\sigma(z)\big(1-\sigma(z)\big).$$

最终得到 Sigmoid 函数导数的简洁形式：

$$\frac{d}{dz}\sigma(z)=\sigma(z)\big(1-\sigma(z)\big).$$

在神经网络的反向传播过程中，通常已知隐含层神经元的输出 $h=\sigma(z)$，因此其导数常直接写为

$$\frac{dh}{dz}=h(1-h).$$

1.3.2 参数更新

在完成梯度计算之后，神经网络的参数通过梯度下降法进行更新，使得损失函数 $\mathcal{L}$ 在训练过程中不断减小，从而实现模型的优化。设学习率为 η，则权重和偏置的更新公式为：

$$w2_i \leftarrow w2_i-\eta\frac{\partial\mathcal{L}}{\partial w2_i},$$

$$b2 \leftarrow b2-\eta\frac{\partial\mathcal{L}}{\partial b2},$$

$$w1_i \leftarrow w1_i-\eta\frac{\partial\mathcal{L}}{\partial w1_i},$$

$$b1_i \leftarrow b1_i-\eta\frac{\partial\mathcal{L}}{\partial b1_i}.$$

1.3.3 Python 实现

在本节，我们用 Python 实现模型训练过程，包括反向传播学习算法、数据准备、模型训练和效果测试几部分。

代码 1.3和 1.4分别给出了 ReLU 和 Sigmoid 两种常用激活函数及其对应的导数实现。这里使用 `0.0`、`1.0` 是为了确保输出为浮点数，从而在反向传播计算梯度时保持数值类型一致。

```
def relu(x):
    # 前向传播：返回 ReLU 激活值
    return max(0.0, x)

def relu_derivative(z):
    # 反向传播：ReLU 的导数
    return 1.0 if z > 0 else 0.0
```

代码 1.3: ReLU 激活函数及其导数

```
def sigmoid(x):
    # 前向传播：返回 Sigmoid 激活值
    return 1.0 / (1.0 + math.exp(-x))

def sigmoid_derivative(h):
    # 反向传播：Sigmoid 的导数
    return h * (1 - h)
```

代码 1.4: Sigmoid 激活函数及其导数

代码 1.5 给出了 `FactorialNN` 的反向传播与参数更新过程。

```
class FactorialNN:
    ...
    def train(self, x, target, lr=0.01):
         """
        对单个样本执行前向传播、反向传播并更新网络参数。

        参数：
        x      : 输入单样本。
        target : 样本的真实标签值。
        lr     : float，默认 0.01
                 学习率，用于梯度下降更新权重和偏置。
        """

        # ---------- 前向传播 ----------
        # z: 隐藏层应用激活函数前的输出列表
        # h: 隐藏层输出列表
        # y: 网络预测输出
        z, h, y = self.forward(x)

        # ---------- 输出层梯度 ----------
        # 使用均方误差（MSE）
        dy = y - target
        mse = dy ** 2

        # ---------- 隐藏层梯度 ----------
```

```
        dh = []
        for i in range(self.hidden_size):              # 链式法则计算隐藏层梯度
            # dL/dh_i = dL/dy * dy/dh_i * dh_i/dz_i
            # dL/dy = dy
            # dy/dh_i = self.w2[i]
            # dh_i/dz_i = 激活函数的导数
            # 假设使用ReLu作为激活函数
            grad = dy * self.w2[i] * relu_derivative(z[i])
            # 如果使用 Sigmoid 作为激活函数:
            # grad = dy * self.w2[i] * sigmoid_derivative(h[i])
            dh.append(grad)


        # ---------- 更新隐藏层 -> 输出层参数 ----------
        for i in range(self.hidden_size):
            self.w2[i] -= lr * dy * h[i]
        self.b2 -= lr * dy

        # ---------- 更新输入层 -> 隐藏层参数 ----------
        for i in range(self.hidden_size):
            self.w1[i] -= lr * dh[i] * x
            self.b1[i] -= lr * dh[i]

        # 返回损失值，方便观察训练过程
        return mse
```

代码 1.5: 反向传播及参数更新

代码 1.6 展示了训练数据准备过程。

```
MAX_N = 10  # 定义最大阶乘值 n

def log_factorial(n):
    """
    计算 log(n!)
    使用对数避免阶乘值快速增长导致溢出
    """
    return math.log(math.factorial(n))

# 训练数据列表
# 输入：归一化的 n（除以 MAX_N）
# 输出：log(n!)，作为回归目标
training_data = [
    (n / MAX_N, log_factorial(n))
    for n in range(1, MAX_N + 1)
]
```

代码 1.6: 构造阶乘训练数据

神经网络初始化及训练过程如代码 1.7 所示，每轮迭代遍历全部训练样本并更新参数，同时可输出总损失观察训练收敛情况。

```
nn = FactorialNN(hidden_size=10)  # 隐藏层10个神经元
```

```python
for epoch in range(20000):          # 迭代次数
    total_loss = 0.0                # 累计本轮总损失
    for x, y in training_data:      # 遍历每个训练样本
        total_loss += nn.train(x, y)  # 前向传播 + 反向传播 + 参数更新

    if epoch % 2000 == 0:           # 每2000轮打印一次
        print(f"Epoch {epoch}, Loss = {total_loss:.4f}")
```

代码 1.7: 训练神经网络

训练完成后，可通过前向传播得到预测的 $\log(n!)$，再使用指数函数还原为阶乘值。代码 1.8 展示了测试过程及预测结果与真实值对比。

```python
print("\nPrediction Results:")

for n in range(1, MAX_N + 1):
    x = n / MAX_N                   # 输入归一化
    _, y_pred = nn.forward(x)       # 前向传播得到 log(n!) 预测值
    fact_pred = math.exp(y_pred)    # 将预测值还原为 n!

    # 打印预测结果与真实阶乘值对比
    print(f"{n}!   {fact_pred:.1f} (true: {math.factorial(n)})")
```

代码 1.8: 测试神经网络预测效果

下面给出了使用 ReLU 和 Sigmoid 作为隐藏层激活函数时，阶乘回归任务的训练过程与预测结果。可以看到，不同激活函数在损失收敛速度以及拟合能力上存在显著差异。

```
#: python factorial-relu.py
Epoch 0, Loss = 661.5144
Epoch 2000, Loss = 5.0790
Epoch 4000, Loss = 4.8953
Epoch 6000, Loss = 4.8947
Epoch 8000, Loss = 4.8939
Epoch 10000, Loss = 4.8934
Epoch 12000, Loss = 4.8928
Epoch 14000, Loss = 4.8922
Epoch 16000, Loss = 4.8911
Epoch 18000, Loss = 4.8905

Prediction Results (ReLU):
1!    0.3   (true: 1)
2!    1.5   (true: 2)
3!    8.2   (true: 6)
4!    45.9  (true: 24)
5!    257.3 (true: 120)
6!    1442.9 (true: 720)
7!    8092.5 (true: 5040)
8!    45386.9 (true: 40320)
9!    254551.8 (true: 362880)
10!   1379211.7 (true: 3628800)
```

代码 1.9: ReLU 激活函数的训练与预测结果

```
#: python factorial-sigmoid.py
Epoch 0, Loss: 705.7546
Epoch 2000, Loss: 0.0695
Epoch 4000, Loss: 0.0622
Epoch 6000, Loss: 0.0573
Epoch 8000, Loss: 0.0531
Epoch 10000, Loss: 0.0493
Epoch 12000, Loss: 0.0459
Epoch 14000, Loss: 0.0429
Epoch 16000, Loss: 0.0404
Epoch 18000, Loss: 0.0383

Prediction Results (Sigmoid):
1!    0.9    (true: 1)
2!    2.2    (true: 2)
3!    6.5    (true: 6)
4!    24.2   (true: 24)
5!    115.0  (true: 120)
6!    684.2  (true: 720)
7!    4955.3 (true: 5040)
8!    41680.6 (true: 40320)
9!    381847.8 (true: 362880)
10!   3497393.9 (true: 3628800)
```

代码 1.10: Sigmoid 激活函数的训练与预测结果

练习

1) 设计一个神经网络程序，预测第 x 个斐波那契数，分析效果。

2) 设计一个神经网络程序，预测任意 x 的正弦值 $y = \sin(x)$，分析效果。