智能控制作业

WA2214014 杨跃浙 WA2214030 蔡文杰

November 2024

1 作业要求

已知网络结构如图 2所示，网络输入/输出如图 1所示。其中，$f(x)$ 为 x 的符号函数，网络的输出为：

$$f(\mathrm{net}) = \mathrm{sign}(w_1x_1 + w_2x_2 + w_3 \cdot 1)$$

其中 bias 取常数 1。设初始权值随机取为 $(0.75, 0.5, -0.6)$。给定十组训练参数表，使用如下的学习算法进行权值更新：

若对于每组输入 (x_1, x_2)，输出 $f(\mathrm{net})$ 和训练数据中的理想输出不一致，则使用有师学习算法：

$$w(n) = w(n-1) + \alpha(k) \cdot (d(n-1) - \mathrm{sign}(w(n-1) \cdot x(n-1))) \cdot x(n-1)$$

其中，$\alpha(k)$ 为学习率调度器。迭代更新直到十个训练参数的输出与理想输出一致，最终得到权值 w。

训练数据如下表所示：

输入 $(x1, x2)$	**输出** Y
([1.0, 1.0])	1
([9.4, 6.4])	-1
([2.5, 2.1])	1
([8.0, 7.7])	-1
([0.5, 2.2])	1
([7.9, 8.4])	-1
([7.0, 7.0])	-1
([2.8, 0.8])	1
([1.2, 3.0])	1
([7.8, 6.1])	-1

图 1: 训练数据表

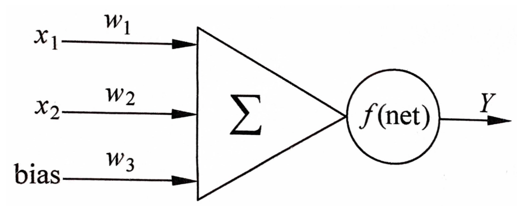

图 2: 神经网络结构示例图

2 实验原理

本文实现了一个简单的线性分类器，并使用三种不同的学习率调度策略（*SquareRootScheduler*、*FactorScheduler* 和 *CosineScheduler*）来动态调整学习率。其原理如下：

2.1 符号函数

定义了一个符号函数 sign，用于将输入值转换为 -1 或 1：

$$\text{sign}(x) = \begin{cases} 1 & \text{if } x > 0 \\ -1 & \text{if } x \leq 0 \end{cases}$$

2.2 初始化权重和偏置

初始化权重向量 $\mathbf{w}$ 和偏置 b。

2.3 训练数据

提供了一组训练数据，每个数据点由一个特征向量 $\mathbf{x}$ 和标签 y 组成：

$$\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$$

2.4 学习率调度器

定义了三种不同的学习率调度器：

- **SquareRootScheduler**: 根据平方根衰减学习率。
- **FactorScheduler**: 根据常数因子逐步减少学习率。
- **CosineScheduler**: 使用余弦函数调整学习率。

2.5 超参数

设置了以下训练超参数：

- 训练轮数：num_epochs。
- 初始学习率：learning_rate。

2.6 训练过程

训练过程的主要步骤如下：

- 对于每一轮训练，遍历所有训练数据。
- 对于每个数据点，向输入向量 $\mathbf{x}$ 添加偏置项，计算感知器的输出：

$$\text{output} = \text{sign}(\mathbf{w} \cdot \mathbf{x} + b)$$

- 如果预测结果与实际标签不一致，则计算误差并更新权重：

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(y_i - \text{output})\mathbf{x}$$

其中 α 为当前学习率。

- 使用学习率调度器更新学习率：

$$\alpha \leftarrow \text{Scheduler}(\alpha)$$

- 打印当前轮次的学习率和权重向量 $\mathbf{w}$。
- 如果正确分类的样本数达到 10 个，则提前终止训练。

2.7 最终权重

训练完成后，打印最终的权重 $\mathbf{w}$ 和每个训练数据的预测结果。

3 代码实现

以下是简单的线性分类器实现，使用三种不同的学习率调度策略。

```python
import math
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

# 定义符号函数
def sign(x):
    return torch.sign(x)

# 初始化权重和偏置
w = torch.tensor([0.75, 0.5, -0.6], requires_grad=True)
bias = 1.0

# 训练数据
training_data = [
    (torch.tensor([1.0, 1.0]), 1),
    (torch.tensor([9.4, 6.4]), -1),
    (torch.tensor([2.5, 2.1]), 1),
    (torch.tensor([8.0, 7.7]), -1),
    (torch.tensor([0.5, 2.2]), 1),
    (torch.tensor([7.9, 8.4]), -1),
    (torch.tensor([7.0, 7.0]), -1),
    (torch.tensor([2.8, 0.8]), 1),
    (torch.tensor([1.2, 3.0]), 1),
    (torch.tensor([7.8, 6.1]), -1)
]

class SquareRootScheduler:
    def __init__(self, lr=0.2):
        self.lr = lr

    def __call__(self, epoch):
        return self.lr * pow(epoch + 1.0, -0.5)

class FactorScheduler:
    def __init__(self, factor=0.9, stop_factor_lr=1e-2, base_lr=0.1):
        self.factor = factor
        self.stop_factor_lr = stop_factor_lr
        self.base_lr = base_lr

    def __call__(self, num_update):
        self.base_lr = max(self.stop_factor_lr, self.base_lr * self.factor)
```

```python
        return self.base_lr

class CosineScheduler:
    def __init__(self, max_update, base_lr=0.01, final_lr=0, warmup_steps=0, warmup_begin_lr=0):
        self.base_lr_orig = base_lr
        self.max_update = max_update
        self.final_lr = final_lr
        self.warmup_steps = warmup_steps
        self.warmup_begin_lr = warmup_begin_lr
        self.max_steps = self.max_update - self.warmup_steps

    def get_warmup_lr(self, epoch):
        increase = (self.base_lr_orig - self.warmup_begin_lr) \
                       * float(epoch) / float(self.warmup_steps)
        return self.warmup_begin_lr + increase

    def __call__(self, epoch):
        if epoch < self.warmup_steps:
            return self.get_warmup_lr(epoch)
        if epoch <= self.max_update:
            self.base_lr = self.final_lr + (
                self.base_lr_orig - self.final_lr) * (1 + math.cos(
                math.pi * (epoch - self.warmup_steps) / self.max_steps)) / 2
        return self.base_lr

# 学习率调度器
# def learning_rate_scheduler(epoch):
#     base_lr = 0.2
#     return base_lr / (1 + 0.1 * epoch)

# 超参数
num_epochs = 100
learning_rate = 0.2
# scheduler = SquareRootScheduler(lr=learning_rate)
# scheduler = FactorScheduler(base_lr=learning_rate)
scheduler = CosineScheduler(max_update=10, base_lr=learning_rate)

# 记录每个epoch的损失和学习率
losses = []
learning_rates = []

# 开始训练
for epoch in range(num_epochs):
    correct_count = 0
    for x, d in training_data:
        # 添加偏置项到输入向量
        x_with_bias = torch.cat((x, torch.tensor([bias])))
        net = w[0] * x_with_bias[0] + w[1] * x_with_bias[1] + w[2] * x_with_bias[2]
        output = sign(net)
        if output == d:
            correct_count += 1
        else:
            # 计算误差
            error = d - output
            # 更新权重
            with torch.no_grad():
                w += learning_rate * error * x_with_bias
    # 更新学习率
    learning_rate = scheduler.__call__(epoch)
    losses.append(correct_count / len(training_data))
    learning_rates.append(learning_rate)
```

```python
    print(f'Epoch {epoch+1}/{num_epochs}, Learning Rate: {learning_rate}')
    print(w)
    # 如果正确分类的样本数达到10个，则提前终止训练
    if correct_count >= 10:
        break

# 打印最终权重
print("Final weights:", w)
for x, d in training_data:
    # 添加偏置项到输入向量
    x_with_bias = torch.cat((x, torch.tensor([bias])))
    net = w[0] * x_with_bias[0] + w[1] * x_with_bias[1] + w[2] * x_with_bias[2]
    output = sign(net)
    print(output)

# 绘制损失和学习率的变化曲线
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(range(len(losses)), losses, label='Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training Accuracy Over Epochs')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(range(len(learning_rates)), learning_rates, label='Learning Rate', color='orange')
plt.xlabel('Epoch')
plt.ylabel('Learning Rate')
plt.title('Learning Rate Over Epochs')
plt.legend()

plt.tight_layout()
plt.show()
```

4 实验结果

4.1 CosineScheduler:

```
C:\pytorch\anaconda3\envs\cpytorch\python.exe C:\python\testwork\test.py
Epoch 1/100, Learning Rate: 0.2
tensor([-3.3300, -1.2600,  0.2000], requires_grad=True)
Epoch 2/100, Learning Rate: 0.19510565162951538
tensor([-3.7300, -1.2600,  1.4000], requires_grad=True)
Epoch 3/100, Learning Rate: 0.18090169943749476
tensor([-1.2717,  0.2618,  2.5706], requires_grad=True)
Epoch 4/100, Learning Rate: 0.15877852522924732
tensor([-1.6335,  0.4065,  3.6560], requires_grad=True)
Epoch 5/100, Learning Rate: 0.13090169943749475
tensor([-3.2213, -1.2765,  3.6560], requires_grad=True)
Epoch 6/100, Learning Rate: 0.1
tensor([-1.5719, -0.2555,  4.4415], requires_grad=True)
Epoch 7/100, Learning Rate: 0.06909830056250527
tensor([-1.0719,  0.1645,  4.6415], requires_grad=True)
Epoch 8/100, Learning Rate: 0.0412214747707527
tensor([-1.0719,  0.1645,  4.6415], requires_grad=True)
Final weights: tensor([-1.0719,  0.1645,  4.6415], requires_grad=True)
tensor(1.)
tensor(-1.)
tensor(1.)
tensor(-1.)
tensor(1.)
tensor(-1.)
tensor(-1.)
tensor(1.)
tensor(1.)
tensor(-1.)

进程已结束，退出代码为 0
```

图 3: CosineScheduler 的运行结果

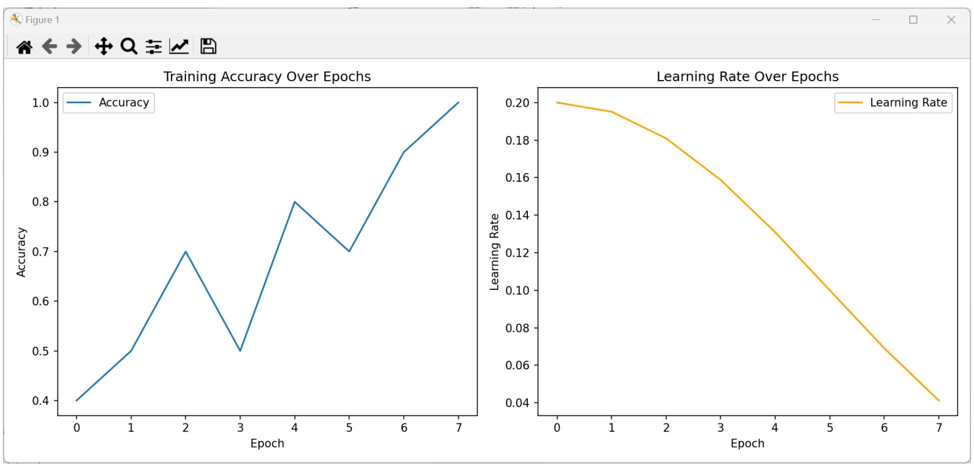

图 4: CosineScheduler 的优化曲线

4.2 SquareRootScheduler:

```
C:\pytorch\anaconda3\envs\cpytorch\python.exe C:\python\testwork\1.py
Epoch 1/100, Learning Rate: 0.2
tensor([-3.3300, -1.2600,  0.2000], requires_grad=True)
Epoch 2/100, Learning Rate: 0.14142135623730953
tensor([-3.7300, -1.2600,  1.4000], requires_grad=True)
Epoch 3/100, Learning Rate: 0.11547005383792515
tensor([-1.8067,  0.4653,  2.5314], requires_grad=True)
Epoch 4/100, Learning Rate: 0.1
tensor([-2.0376,  0.5577,  3.2242], requires_grad=True)
Epoch 5/100, Learning Rate: 0.08944271909999159
tensor([-2.5376, -0.0823,  3.4242], requires_grad=True)
Epoch 6/100, Learning Rate: 0.08164965809277261
tensor([-1.5895,  0.4365,  3.7820], requires_grad=True)
Epoch 7/100, Learning Rate: 0.07559289460184546
tensor([-1.1323,  0.5671,  3.9453], requires_grad=True)
Epoch 8/100, Learning Rate: 0.07071067811865477
tensor([-1.1323,  0.5671,  3.9453], requires_grad=True)
Final weights: tensor([-1.1323,  0.5671,  3.9453], requires_grad=True)
tensor(1., grad_fn=<SignBackward0>)
tensor(-1., grad_fn=<SignBackward0>)
tensor(1., grad_fn=<SignBackward0>)
tensor(-1., grad_fn=<SignBackward0>)
tensor(1., grad_fn=<SignBackward0>)
tensor(-1., grad_fn=<SignBackward0>)
tensor(-1., grad_fn=<SignBackward0>)
tensor(1., grad_fn=<SignBackward0>)
tensor(1., grad_fn=<SignBackward0>)
tensor(-1., grad_fn=<SignBackward0>)
```

图 5: SquareRootScheduler 的运行结果

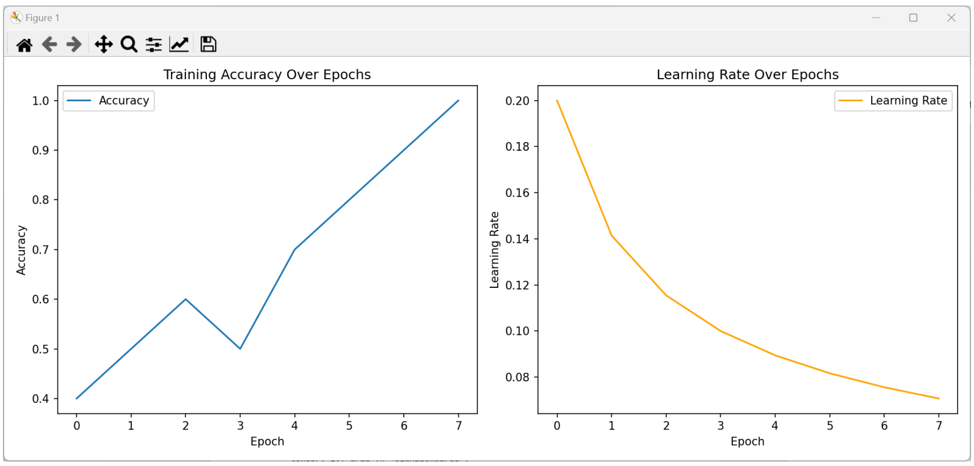

图 6: SquareRootScheduler 的优化曲线

4.3 FactorScheduler:

```
C:\pytorch\anaconda3\envs\cpytorch\python.exe C:\python\testwork\1.py
Epoch 1/100, Learning Rate: 0.18000000000000002
tensor([-3.3300, -1.2600,  0.2000], requires_grad=True)
Epoch 2/100, Learning Rate: 0.16200000000000003
tensor([-3.6900, -1.2600,  1.2800], requires_grad=True)
Epoch 3/100, Learning Rate: 0.14580000000000004
tensor([-1.6488,  0.0036,  2.2520], requires_grad=True)
Epoch 4/100, Learning Rate: 0.13122000000000003
tensor([-2.0570, -0.7254,  2.8352], requires_grad=True)
Epoch 5/100, Learning Rate: 0.11809800000000004
tensor([-0.6661,  0.0357,  3.3601], requires_grad=True)
Epoch 6/100, Learning Rate: 0.10628820000000004
tensor([-0.6661,  0.0357,  3.3601], requires_grad=True)
Final weights: tensor([-0.6661,  0.0357,  3.3601], requires_grad=True)
tensor(1., grad_fn=<SignBackward0>)
tensor(-1., grad_fn=<SignBackward0>)
tensor(1., grad_fn=<SignBackward0>)
tensor(-1., grad_fn=<SignBackward0>)
tensor(1., grad_fn=<SignBackward0>)
tensor(-1., grad_fn=<SignBackward0>)
tensor(-1., grad_fn=<SignBackward0>)
tensor(1., grad_fn=<SignBackward0>)
tensor(1., grad_fn=<SignBackward0>)
tensor(-1., grad_fn=<SignBackward0>)
```

图 7: FactorScheduler 的运行结果

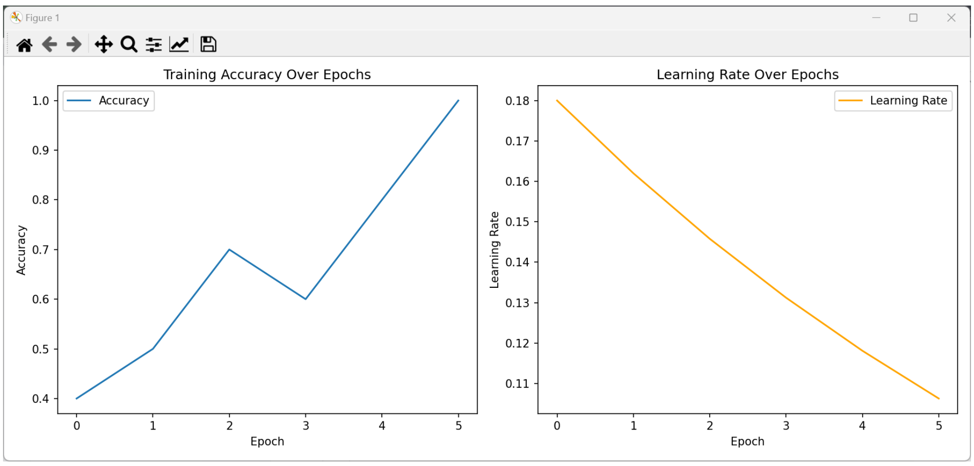

图 8: FactorScheduler 的优化曲线

4.4 运行时间对比:

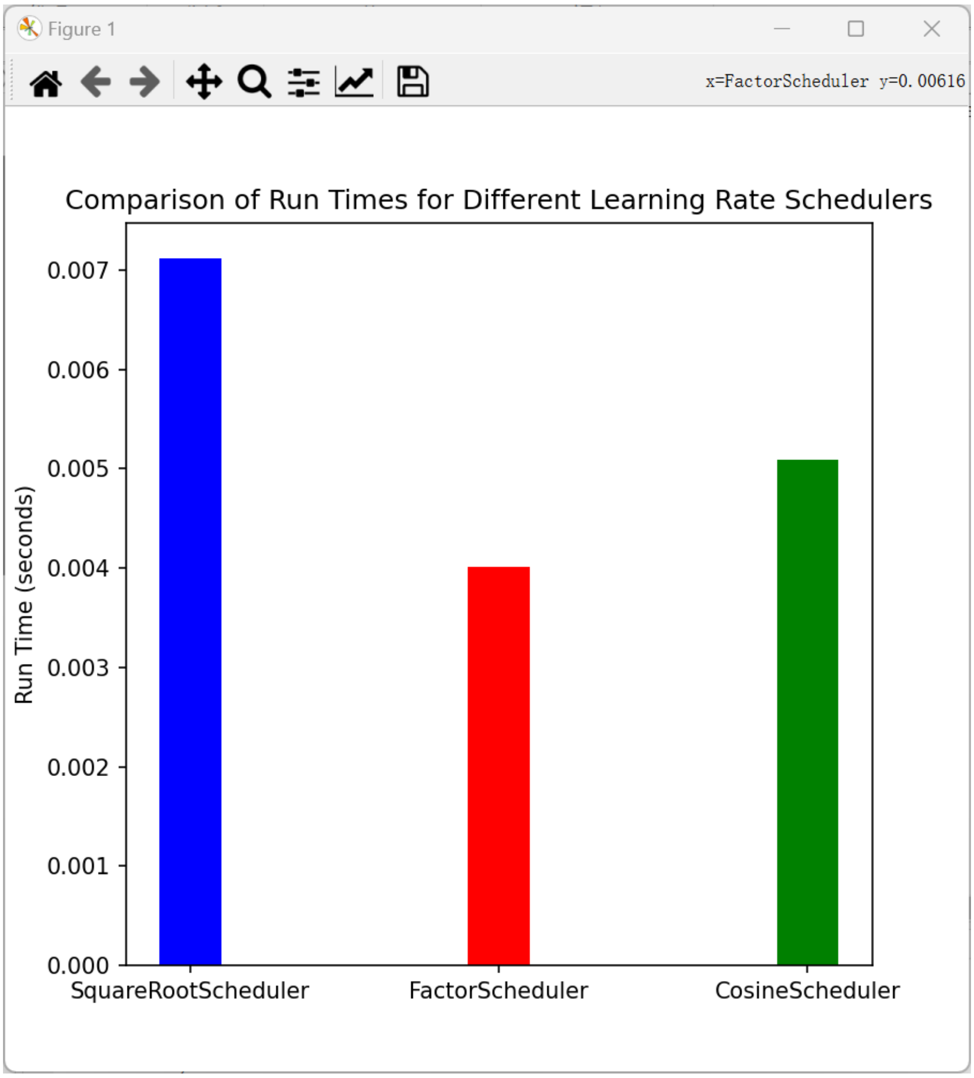

图 9: 三种不同学习率调度器的运行时间对比