深度模型中的优化

朱明超
deityrayleigh@gmail.com

1 基本优化算法

整体框架

1.1 梯度

1.1.1 梯度下降

- 微积分中使用**梯度**表示函数增长最快的方向，因此，神经网络中使用**负梯度**来指示目标函数下降最快的方向。
  - **梯度**实际上是损失函数对网络中每个参数的**偏导**所组成的向量；
  - **梯度**仅仅指示了对于每个参数各自增长最快的方向，因此，梯度无法保证**全局方向**就是函数为了达到最小值应该前进的方向；
  - 使用**梯度**的具体计算方法即**反向传播**。
- **梯度下降**是一种优化算法，通过**迭代**的方式寻找使模型**目标函数**达到**最小值**时的最优参数 (也称**最速下降法**)：
  - 当目标函数是**凸函数**时，梯度下降的解是全局最优解，但在一般情况下，**梯度下降无法保证全局最优**；
  - 梯度下降最常用的形式是批量梯度下降法 (`Batch Gradient Descent`，BGD)，其做法是在更新参数时使用所有的样本来进行更新。
- 反过来，如果求解**目标函数**达到**最大值**时的最优参数，就需要用**梯度上升法**进行迭代。
- **负梯度**中的每一项可以认为传达了**两个信息**：
  - 正负号在告诉输入向量应该调大还是调小 (正调大，负调小)；
  - 每一项的相对大小表明每个参数对函数值达到最值的**影响程度**。

1.1.2 随机梯度下降

首先，梯度下降法沿着梯度的反方向进行搜索，利用了函数的一阶导数信息。基本的梯度下降法每次使用**所有训练样本**的**平均损失**来更新参数。因此，经典的梯度下降在每次对模型参数进行更新时，需要遍历所有数据。当训练样本的数量很大时，这需要消耗相当大的计算资源，在实际应用中基本不可行。

而**随机梯度下降** (`Stochastic Gradient Descent`，SGD) 是随机抽取一批样本 (Batch)，以此为根据来更新参数。随机梯度下降每次使用 m 个样本的损失来近似平均损失，更新规则：

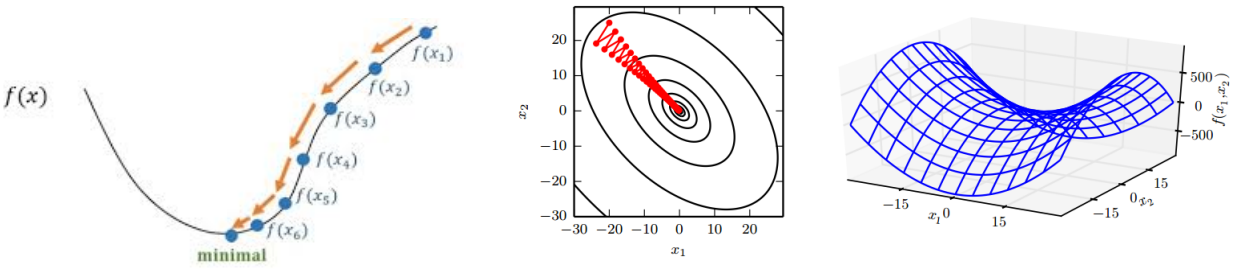
$$
\begin{aligned}
&g \leftarrow \frac{1}{m}\nabla_\theta \sum_i J(f(\boldsymbol{x}^{(i)};\theta),\boldsymbol{y}^{(i)}) \\
&\theta \leftarrow \theta - \epsilon g
\end{aligned}
\tag{1}
$$

随机梯度下降法存在的问题：

- 随机梯度下降 (SGD) 放弃了**梯度的准确性**，仅采用一部分样本来估计当前的梯度。因此 SGD 对梯度的估计常常出现偏差，造成目标函数收敛不稳定，甚至不收敛的情况。
- 无论是经典的梯度下降还是随机梯度下降，都可能陷入**局部极值点**。除此之外，SGD 还可能遇到 "**峡谷**" 和 "**鞍点**" 两种情况。
  - **峡谷**类似一个带有**坡度**的狭长小道，左右两侧是 "**峭壁**"；在**峡谷**中，准确的梯度方向应该沿着坡的方向向下，但粗糙的梯度估计使其稍有偏离就撞向两侧的峭壁，然后在两个峭壁间来回**震荡**。
  - **鞍点**的形状类似一个马鞍，一个方向两头翘，一个方向两头垂，而**中间区域近似平地**；一旦优化的过程中不慎落入鞍点，优化很可能就会停滞下来。

SGD 的改进遵循两个方向：**惯性保持**和**环境感知**。

- **惯性保持**指的是加入**动量**：**动量 (`Momentum`) 方法**。
- **环境感知**指的是根据不同参数的一些**经验性判断**，**自适应**地确定**每个参数的学习速率**：**自适应学习率的优化算法**。



(i) 梯度下降法和随机梯度下降法迭代过程 (ii) 随机梯度下降法存在的问题，峡谷与鞍点

图 1. 随机梯度下降法

[1]:
```
from abc import ABC, abstractmethod
import numpy as np
```

[2]:
```
class OptimizerBase(ABC):

    def __init__(self):
        pass

    def __call__(self):
        return self.update(params, params_grad, params_name)

    @abstractmethod
    def update(self, params, params_grad, params_name):
        """
        参数说明:
        params: 待更新参数， 如权重矩阵 W;
        params_grad: 待更新参数的梯度;
        params_name: 待更新参数名;
        """
        raise NotImplementedError
```

[3]:
```
class SGD(OptimizerBase):
    """
    sgd 优化方法
    """
    def __init__(self, lr=0.001):
        super().__init__()
        self.lr = lr
        self.cache = {}

    def __str__(self):
        return "SGD(lr={})".format(self.hyperparams["lr"])

    def update(self, params, params_grad, params_name):
        update_value = self.lr * params_grad
        return params - update_value

    @property
    def hyperparams(self):
        return {
            "op": "SGD",
            "lr": self.lr
        }
```

1.2 动量

1.2.1 Momentum 算法

引入**动量** (Momentum) 方法一方面是为了解决 “峡谷” 和 “鞍点” 问题，另一方面也可以用于 SGD 加速，特别是针对**高曲率、小幅但是方向一致**的梯度。

如果把原始的 SGD 想象成一个**纸团**在重力作用向下滚动，由于**质量小**受到山壁弹力的干扰大，导致来回震荡。或者在鞍点处因为**质量小**速度很快减为 0，导致无法离开这块平地。**动量**方法相当于把纸团换成了**铁球**。不容易受到外力的干扰，轨迹更加稳定，同时因为在鞍点处因为**惯性**的作用，更有可能离开平地。

参数更新公式

$$
\begin{aligned}
v &\leftarrow \alpha v - \epsilon \nabla_\theta (\frac{1}{m} \sum_{i=1}^{m} J(f(\boldsymbol{x^{(i)}}; \theta), \boldsymbol{y}^{(i)})) \\
\theta &\leftarrow \theta + v
\end{aligned}
\tag{2}
$$

- 从形式上看，动量算法引入了变量 v 充当速度角色，以及相关的超参数 α，决定了之前的梯度贡献衰减得有多快；
- 原始 SGD 每次更新的步长只是梯度乘以学习率。现在，步长还取决于**历史梯度序列**的大小和排列。当许多连续的梯度指向**相同的方向**时，**步长会被不断增大**。

速度 v 累积了当前梯度元素 $\nabla_\theta(\frac{1}{m} \sum_{i=1}^{m} J(f(\boldsymbol{x^{(i)}}; \theta), \boldsymbol{y}^{(i)}))$，相对于 ϵ，如果 α 越大，之前梯度对现在方向的影响也越大。实践中，α 一般取值为 0.5，0.9 和 0.99。

当前迭代点的下降方向不仅仅取决于**当前的梯度**，还受到**前面所有迭代点的影响**。动量方法以一种廉价的方式模拟了二阶梯度 (牛顿法)。

1.2.2 NAG 算法

Nesterov 提出了一个针对动量算法的改进措施。前面的动量算法是把历史的梯度和当前的梯度进行合并，来计算下降的方向。而 Nesterov 提出，让迭代点先按照历史梯度走一步，然后再合并。更新规则如下，改变主要在于梯度的计算上：

$$
\begin{aligned}
v &\leftarrow \alpha v - \epsilon \nabla_\theta (\frac{1}{m} \sum_{i=1}^{m} J(f(\boldsymbol{x^{(i)}}; \theta + \alpha v), \boldsymbol{y}^{(i)})) \\
\theta &\leftarrow \theta + v
\end{aligned}
\tag{3}
$$

参数 α 和 ϵ 发挥了和标准动量方法中类似的作用，Nesterov 动量和标准动量之间的区别在于梯度的计算上。**NAG 把梯度计算放在对参数施加当前速度之后**，这个“**提前量**”的设计让算法有了对前方环境“**预判**”的能力，可以理解为 Nesterov 动量往标准动量方法中添加了一个校正因子。在凸优化问题使用批量梯度下降的情况下，Nesterov 动量将 k 步之后额外误差收敛率从 $O(\frac{1}{k})$ 提高到 $O(\frac{1}{k^2})$，对 SGD 没有改进收敛率。

如图 2 所示，左图显示了 Momentum 方法的轨迹，右图显示了 Nesterov 方法的轨迹。

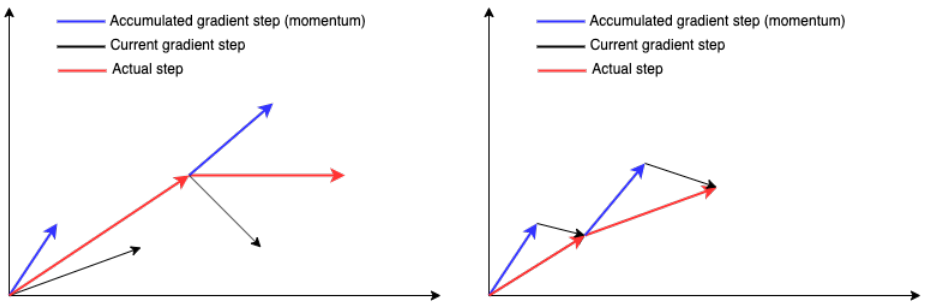


图 2. 动量方法的轨迹。蓝色线为历史的梯度，黑色线为当前的梯度，红色线为实际轨迹。

[4]:
```
"""
在 SGD 中考虑 momentum
"""
class Momentum(OptimizerBase):

    def __init__(
        self, lr=0.001, momentum=0.0, **kwargs
    ):
        """
        参数说明:
        lr: 学习率, float (default: 0.001)
        momentum: 考虑 Momentum 时的 alpha, 决定了之前的梯度贡献衰减得有多快, 取值范围 [0, 1], 默认 0
        """
        super().__init__(lr)
        self.momentum = momentum
        self.cache = {}

    def __str__(self):
        return "Momentum(lr={}, momentum={})".format(self.lr, self.momentum)

    def update(self, param, param_grad, param_name):
        C = self.cache
```

```
        lr, momentum = self.lr, self.momentum

        if param_name not in C:  # save v
            C[param_name] = np.zeros_like(param_grad)

        update = momentum * C[param_name] - lr * param_grad
        self.cache[param_name] = update
        return param + update

    @property
    def hyperparams(self):
        return {
            "op": "Momentum",
            "lr": self.lr,
            "momentum": self.momentum
        }
```

1.3 自适应学习率

由于 SGD 中随机采样 Batch 会引入噪声源，在极小点处梯度并不会消失。因此，随着梯度的降低，有必要逐步减小学习率。

1.3.1 3.1 AdaGrad 算法

算法的思想是独立地适应模型的每个参数：**一直较大偏导的参数相应有一个较小的学习率，初始学习率会下降的较快；而一直小偏导的参数则对应一个较大的学习率，初始学习率会下降的较慢**。具体来说，每个参数的学习率会缩放各参数反比于其**历史梯度平方值总和的平方根**，更新公式：

$$
\begin{aligned}
g &\leftarrow \frac{1}{m}\nabla_\theta \sum_i J(f(\mathrm{x}^{(\mathrm{i})};\theta),\mathrm{y}^{(\mathrm{i})}) \\
r &\leftarrow r + g \odot g \\
\theta &\leftarrow \theta - \frac{\epsilon}{\delta + \sqrt{r}} \odot g
\end{aligned}
\tag{4}
$$

r 为**累积平方梯度**。学习率相当于 $\frac{\epsilon}{\delta+\sqrt{r}}$，对于更新不频繁的参数，单次步长更大；对于更新频繁的参数，步长较小，使得学习到的参数更稳定，不至于被单个样本影响太多。

AdaGrad 算法存在的问题，历史梯度在分母上的**累积会越来越大**，所以**学习率会越来越小**，使得在中后期网络的学习能力越来越弱。

[5]:
```
class AdaGrad(OptimizerBase):

    def __init__(self, lr=0.001, eps=1e-7, **kwargs):
        """
        参数说明:
        lr: 学习率, float (default: 0.001)
        eps: delta 项, 防止分母为 0
        """
        super().__init__(lr)
        self.lr = lr
        self.eps = eps
        self.cache = {}

    def __str__(self):
        return "AdaGrad(lr={}, eps={})".format(self.lr, self.eps)

    def update(self, param, param_grad, param_name):
        C = self.cache
        lr, eps = self.hyperparams["lr"], self.hyperparams["eps"]
        if param_name not in C:  # save r
            C[param_name] = np.zeros_like(param_grad)
        C[param_name] += param_grad ** 2
        update = lr * param_grad / (np.sqrt(C[param_name]) + eps)
        self.cache = C
```

```
        return param - update

    @property
    def hyperparams(self):
        return {
            "op": "AdaGrad",
            "lr": self.lr,
            "eps": self.eps
        }
```

1.3.2 RMSProp 算法

RMSProp 主要是为了解决 AdaGrad 方法中**学习率过度衰减**的问题—— AdaGrad 根据平方梯度的整个历史来收缩学习率，可能使得学习率在达到局部最小值之前就变得太小而难以继续训练。RMSProp 使用**指数衰减平均** (递归定义) 以丢弃遥远的历史，使其能够在找到某个 “凸” 结构后快速收敛。此外，RMSProp 还加入了一个超参数 ρ 用于控制衰减速率：

$$
\begin{aligned}
g &\leftarrow \frac{1}{m}\nabla_\theta \sum_i J(f(\mathrm{x}^{(\mathrm{i})};\theta),\mathrm{y}^{(\mathrm{i})}) \\
r &\leftarrow \rho r + (1-\rho) g \odot g \\
\theta &\leftarrow \theta - \frac{\epsilon}{\sqrt{\delta + r}} \odot g
\end{aligned} \tag{5}
$$

RMSProp 建议的**初始值**：全局学习率 $\epsilon = 1e-3$，衰减速率 $\rho = 0.9$。

[6]:
```
class RMSProp(OptimizerBase):

    def __init__(
        self, lr=0.001, decay=0.9, eps=1e-7, **kwargs
    ):
        """
        参数说明:
        lr: 学习率, float (default: 0.001)
        eps: delta 项, 防止分母为 0
        decay: 衰减速率
        """
        super().__init__(lr)
        self.lr = lr
        self.eps = eps
        self.decay = decay
        self.cache = {}

    def __str__(self):
        return "RMSProp(lr={}, eps={}, decay={})".format(
            self.lr, self.eps, self.decay
        )

    def update(self, param, param_grad, param_name):
        C = self.cache
        lr, eps = self.hyperparams["lr"], self.hyperparams["eps"]
        decay = self.hyperparams["decay"]
        if param_name not in C:  # save r
            C[param_name] = np.zeros_like(param_grad)
        C[param_name] = decay * C[param_name] + (1 - decay) * param_grad ** 2
        update = lr * param_grad / (np.sqrt(C[param_name]) + eps)
        self.cache = C
        return param - update

    @property
    def hyperparams(self):
        return {
            "op": "RMSProp",
```

```
            "lr": self.lr,
            "eps": self.eps,
            "decay": self.decay
        }
```

1.3.3 AdaDelta 算法

AdaDelta 和 RMSProp 一样使用**指数衰减平均** (递归定义) 以丢弃遥远的历史，但 AdaDelta 算法没有学习率这一超参数。AdaDelta 算法维护一个额外的变量 $\Delta\theta$，来计算自变量变化量按元素平方的指数加权移动平均：

$$
\begin{aligned}
g &\leftarrow \frac{1}{m}\nabla_\theta \sum_i J(f(\mathrm{x}^{(\mathrm{i})};\theta),\mathrm{y}^{(\mathrm{i})}) \\
r &\leftarrow \rho r + (1-\rho) g \odot g \\
g' &\leftarrow \frac{\sqrt{\delta+\Delta\theta}}{\sqrt{\delta+r}} \odot g \\
\theta &\leftarrow \theta - g' \\
\Delta\theta &\leftarrow \rho\Delta\theta + (1-\rho) g' \odot g'
\end{aligned}
\tag{6}
$$

可以看到，如不考虑 δ 的影响，AdaDelta 算法与 RMSProp 算法的不同之处在于使用 $\sqrt{\Delta\theta}$ 来替代超参数 ϵ。

[7]:
```
class AdaDelta(OptimizerBase):

    def __init__(
        self, lr=0.001, decay=0.95, eps=1e-7, **kwargs
    ):
        """
        参数说明:
        lr: 学习率，float (default: 0.001)
        eps: delta 项，防止分母为 0
        decay: 衰减速率
        """
        super().__init__(lr)
        self.lr = lr
        self.eps = eps
        self.decay = decay
        self.cache = {}

    def __str__(self):
        return "AdaDelta(eps={}, decay={})".format(self.eps, self.decay)

    def update(self, param, param_grad, param_name):
        C = self.cache
        eps = self.hyperparams["eps"]
        decay = self.hyperparams["decay"]
        if param_name not in C:  # save r, delta_theta
            C[param_name] = {
                "r": np.zeros_like(param_grad),
                "d": np.zeros_like(param_grad)
            }
        C[param_name]["r"] = decay * C[param_name]["r"] + (1 - decay) * param_grad ** 2
        update = (np.sqrt(C[param_name]["d"] + eps)) * param_grad / (np.sqrt(C[param_name]["r"]) + eps)
        C[param_name]["d"] = decay * C[param_name]["d"] + (1 - decay) * update ** 2
        self.cache = C
        return param - update

    @property
    def hyperparams(self):
        return {
            "op": "AdaDelta",
            "eps": self.eps,
```

```
            "decay": self.decay
        }
```

1.3.4 Adam 算法

Adam 在 RMSProp 方法的基础上更进一步：除了加入**历史梯度平方的指数衰减平均** (s) 外，还保留了**历史梯度的指数衰减平均** (r)，相当于**动量**。Adam 行为就像一个带有摩擦力的小球，在误差面上倾向于平坦的极小值。

$$
\begin{aligned}
g &\leftarrow \frac{1}{m}\nabla_\theta \sum_i J(f(\mathrm{x}^{(\mathrm{i})};\theta),\mathrm{y}^{(\mathrm{i})}) \\
t &\leftarrow t+1 \\
r &\leftarrow \rho_1 r + (1-\rho_1)g \\
s &\leftarrow \rho_2 s + (1-\rho_2)g \odot g \\
\hat{r} &\leftarrow \frac{r}{1-\rho_1^t} \\
\hat{s} &\leftarrow \frac{s}{1-\rho_2^t} \\
\Delta\theta &\leftarrow -\epsilon\frac{\hat{r}}{\sqrt{\hat{s}}+\delta} \\
\theta &\leftarrow \theta + \Delta\theta
\end{aligned}
\tag{7}
$$

偏差修正

这里的 s 和 r 初始化为 0，且 ρ_1 和 ρ_2 推荐的初始值都很接近 1 (0.9 和 0.999)。注意到，在时间步 t 我们得到 $r_t = (1-\rho_1)\sum_{i=1}^{t}\rho_1^{t-i}g_i$，将过去各时间步小批量随机梯度的权值相加，得到 $(1-\rho_1)\sum_{i=1}^{t}\rho_1^{t-i} = 1-\rho_1^t$。可以看出，当 t 较小时，过去各时间步小批量随机梯度权值之和会较小。假设取 $\rho_1 = 0.9$，可以得到 $t=1$ 时 $r_1 = 0.1g_1$。因此我们需要偏差修正 (见式 7 第 5，6 行)，使过去各时间步小批量随机梯度权值之和为 1。

[8]:
```
class Adam(OptimizerBase):

    def __init__(
        self,
        lr=0.001,
        decay1=0.9,
        decay2=0.999,
        eps=1e-7,
        **kwargs
    ):
        """
        参数说明:
        lr: 学习率，float (default: 0.01)
        eps: delta 项，防止分母为 0
        decay1: 历史梯度的指数衰减速率，可以理解为考虑梯度均值 (default: 0.9)
        decay2: 历史梯度平方的指数衰减速率，可以理解为考虑梯度方差 (default: 0.999)
        """
        super().__init__(lr)
        self.lr = lr
        self.decay1 = decay1
        self.decay2 = decay2
        self.eps = eps
        self.cache = {}

    def __str__(self):
        return "Adam(lr={}, decay1={}, decay2={}, eps={})".format(
            self.lr, self.decay1, self.decay2, self.eps
        )

    def update(self, param, param_grad, param_name, cur_loss=None):
        C = self.cache
        d1, d2 = self.hyperparams["decay1"], self.hyperparams["decay2"]
        lr, eps = self.hyperparams["lr"], self.hyperparams["eps"]
        if param_name not in C:
            C[param_name] = {
```

```
                "t": 0,
                "mean": np.zeros_like(param_grad),
                "var": np.zeros_like(param_grad),
            }
        t = C[param_name]["t"] + 1
        mean = C[param_name]["mean"]
        var = C[param_name]["var"]
        C[param_name]["t"] = t
        C[param_name]["mean"] = d1 * mean + (1 - d1) * param_grad
        C[param_name]["var"] = d2 * var + (1 - d2) * param_grad ** 2
        self.cache = C
        m_hat = C[param_name]["mean"] / (1 - d1 ** t)
        v_hat = C[param_name]["var"] / (1 - d2 ** t)
        update = lr * m_hat / (np.sqrt(v_hat) + eps)
        return param - update

    @property
    def hyperparams(self):
        return {
            "op": "Adam",
            "lr": self.lr,
            "eps": self.eps,
            "decay1": self.decay1,
            "decay2": self.decay2
        }
```

使用上述优化算法，MNIST 数据集测试

[9]:
```
from chapter6 import DFN
```

[10]:
```
"""
载入数据
"""
def load_data(path="../data/mnist/mnist.npz"):
    f = np.load(path)
    X_train, y_train = f['x_train'], f['y_train']
    X_test, y_test = f['x_test'], f['y_test']
    f.close()
    return (X_train, y_train), (X_test, y_test)

(X_train, y_train), (X_test, y_test) = load_data()
y_train = np.eye(10)[y_train.astype(int)]
y_test = np.eye(10)[y_test.astype(int)]
X_train = X_train.reshape(-1, X_train.shape[1]*X_train.shape[2]).astype('float32')
X_test = X_test.reshape(-1, X_test.shape[1]*X_test.shape[2]).astype('float32')
print(X_train.shape, y_train.shape)
N = 20000   # 取 20000 条数据用以训练
indices = np.random.permutation(range(X_train.shape[0]))[:N]
X_train, y_train = X_train[indices], y_train[indices]
print(X_train.shape, y_train.shape)
X_train /= 255
X_train = (X_train - 0.5) * 2
X_test /= 255
X_test = (X_test - 0.5) * 2
```

```
(60000, 784) (60000, 10)
(20000, 784) (20000, 10)
```

[11]:
```
"""
SGD 优化
"""
```

```
model = DFN(hidden_dims_1=200, hidden_dims_2=10, optimizer="sgd(lr=0.01)")
model.fit(X_train, y_train, n_epochs=20, batch_size=64, epo_verbose=False)
print("sgd -- accuracy:{}".format(model.evaluate(X_test, y_test)))
```

```
sgd -- accuracy:0.8951
```

[12]:
```
"""
Momentum 优化
"""
model = DFN(hidden_dims_1=200, hidden_dims_2=10, optimizer="momentum(lr=0.01, momentum=0.95)")
model.fit(X_train, y_train, n_epochs=20, batch_size=64, epo_verbose=False)
print("momentum -- accuracy:{}".format(model.evaluate(X_test, y_test)))
```

```
momentum -- accuracy:0.9604
```

[13]:
```
"""
AdaGrad 优化
"""
model = DFN(hidden_dims_1=200, hidden_dims_2=10, optimizer="adagrad(lr=0.01, eps=1e-7)")
model.fit(X_train, y_train, n_epochs=20, batch_size=64, epo_verbose=False)
print("adagrad -- accuracy:{}".format(model.evaluate(X_test, y_test)))
```

```
adagrad -- accuracy:0.9503
```

[14]:
```
"""
RMSProp 优化
"""
model = DFN(hidden_dims_1=200, hidden_dims_2=10, optimizer="rmsprop(lr=0.001, eps=1e-7, decay=0.95)")
model.fit(X_train, y_train, n_epochs=20, batch_size=64, epo_verbose=False)
print("rmsprop -- accuracy:{}".format(model.evaluate(X_test, y_test)))
```

```
rmsprop -- accuracy:0.9614
```

[15]:
```
"""
AdaDelta 优化
"""
model = DFN(hidden_dims_1=200, hidden_dims_2=10, optimizer="adadelta(eps=1e-7, decay=0.95)")
model.fit(X_train, y_train, n_epochs=20, batch_size=64, epo_verbose=False)
print("adadelta -- accuracy:{}".format(model.evaluate(X_test, y_test)))
```

```
adadelta -- accuracy:0.9624
```

[16]:
```
"""
Adam 优化
"""
model = DFN(hidden_dims_1=200, hidden_dims_2=10, optimizer="adam(lr=0.001, decay1=0.9, decay2=0.999, eps=1e-7)")
model.fit(X_train, y_train, n_epochs=20, batch_size=64, epo_verbose=False)
print("adam -- accuracy:{}".format(model.evaluate(X_test, y_test)))
```

```
adam -- accuracy:0.9667
```

1.4 二阶近似方法

1.4.1 牛顿法

梯度下降使用的梯度信息实际上是**一阶导数**，而牛顿法除了一阶导数外，还会使用**二阶导数**的信息。根据导数的定义，一阶导描述的是函数值的变化率，即**斜率**；二阶导描述的则是斜率的变化率，即曲线的弯曲程度——**曲率**。

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^\top \nabla_\theta J(\theta_0) + \frac{1}{2}(\theta - \theta_0)^\top \mathrm{H}(\theta - \theta_0) \tag{8}$$

更新公式：

$$
\begin{aligned}
g &\leftarrow \frac{1}{m}\nabla_\theta \sum_i J(f(\mathrm{x}^{(\mathrm{i})};\theta),\mathrm{y}^{(\mathrm{i})}) \\
H &\leftarrow \frac{1}{m}\nabla_\theta^2 \sum_i J(f(\mathrm{x}^{(\mathrm{i})};\theta),\mathrm{y}^{(\mathrm{i})}) \\
\Delta\theta &\leftarrow -\mathrm{H}^{-1}g \\
\theta &\leftarrow \theta + \Delta\theta
\end{aligned}
\tag{9}
$$

几何理解

牛顿法就是用一个**二次曲面**去拟合当前所处位置的局部曲面，而梯度下降法是用一个平面去拟合当前的局部曲面。

通常情况下，二次曲面的拟合会比平面更好，所以牛顿法选择的**下降路径**会更符合真实的最优下降路径。

通俗理解

找一条最短的路径走到一个盆地的最底部，梯度下降法每次只从当前所处位置选一个坡度最大的方向走一步；牛顿法在选择方向时，不仅会考虑坡度是否够大，还会考虑走了一步之后，**坡度是否会变得更大**。所以，牛顿法比梯度下降法看得更远，能**更快**地走到最底部。

- **优点**
  - 收敛速度快，能用更少的迭代次数找到最优解
- **缺点**
  - 每一步都需要求解目标函数的 Hessian 矩阵的**逆矩阵**，计算复杂。
  - 牛顿法是局部收敛的，当初始点选择不当时，往往导致不收敛。
  - 二阶 Hessian 矩阵必须可逆，否则算法进行困难。

1.4.2 拟牛顿法

拟牛顿法的本质思想是改善牛顿法每次需要求解复杂的 Hessian 矩阵的逆矩阵的缺陷，它使用正定矩阵来近似 Hessian 矩阵的逆，从而简化了运算的复杂度。

拟牛顿法和最速下降法一样只要求每一步迭代时知道目标函数的梯度。通过测量梯度的变化，构造一个目标函数的模型使之足以产生超线性收敛性。这类方法大大优于最速下降法，尤其对于困难的问题。另外，因为拟牛顿法不需要二阶导数的信息，所以有时比牛顿法更为有效。

2 优化策略

2.1 参数初始化

神经网络的训练过程中的参数学习是基于梯度下降法进行优化的。梯度下降法需要在开始训练时给每一个参数赋一个初始值。这个初始值的选取十分关键。一般希望**数据和参数的均值都为 0，输入和输出数据的方差一致**。在实际应用中，参数服从高斯分布或者均匀分布都是比较有效的初始化方式。

初始参数需要在不同单元间**破坏对称性**。如果具有相同激活函数的两个隐藏单元连接到相同的输入，那么这些单元必须**具有不同的初始参数**。**如果它们具有相同的初始参数**，然后应用到确定性损失和模型的确定性学习算法将**一直以相同的方式更新这两个单元**。

更大的初始权重具有**更强的破坏对称性**的作用，但也会在前向传播或反向传播中产生**爆炸的值**。如在循环网络中，很大的权重也可能导致混沌。因此，一般不要全零初始化或者具有较大数的随机初始化。

初始化策略有：

1、随机初始化方法

- 正态化的随机初始化 (Random Normal)：它是从以 0 为中心，标准差为 std 的正态分布中抽取样本。
- 标准化的随机初始化 (Random Uniform)：它是从 $[-1,1]$ 中的均匀分布中抽取样本。

2、Glorot 初始化方法

Glorot 初始化，也称为 Xavier 初始化。首先，我们考虑隐藏层 $\boldsymbol{z}=\boldsymbol{W}^\top\boldsymbol{x}$ 以及 $\boldsymbol{a}=\sigma(\boldsymbol{z})$，Glorot 初始化认为一个好的初始化应该在各个层的激活值 $\boldsymbol{a}$ 的方差保持一致，即对于第 i 层和第 j 层，有 $\mathrm{Var}(a^i)=\mathrm{Var}(a^j)$，同时各个层对状态 $\boldsymbol{z}$ 的梯度的方差要保持一致，即 $\mathrm{Var}(\frac{\partial J}{\partial z^i})=\mathrm{Var}(\frac{\partial J}{\partial z^j})$。我们先分析一边，我们视每个神经元为一个特征。假设单层神经元上的特征的方差一样，即 $\mathrm{Var}(x_k)=\mathrm{Var}(\boldsymbol{x})$，每一层的输入 $\boldsymbol{a}$ 均值为 0，且初始

时 $\boldsymbol{z}$ 落在激活函数的线性区域，即 $\sigma'(z_l) \approx 1$，$l = \{1, ..., m\}$，m 为该层神经元数目。于是：

$$\begin{aligned}
\mathrm{Var}(a_l^i) &= \mathrm{Var}(\sigma(z_l^i)) \\
&= \mathrm{Var}(z_l^i) \\
&= \mathrm{Var}(\sum_{k=1}^{n} W_{lk} * a_k^{i-1}) \\
&= \sum_{k=1}^{n} \mathrm{Var}(W_{lk} * a_k^{i-1}) \\
&= n\,\mathrm{Var}(\mathrm{W}_{lk})\,\mathrm{Var}(a_k^{i-1}) \\
&= n\,\mathrm{Var}(\boldsymbol{W})\,\mathrm{Var}(a^{i-1})
\end{aligned} \tag{10}$$

由于 $\mathrm{Var}(\boldsymbol{a}_i) = \mathrm{Var}(\boldsymbol{a}_{i-1})$，故 $n\,\mathrm{Var}(\boldsymbol{W}) = 1$。于是考虑一边的情况下，$\mathrm{Var}(\boldsymbol{W}) = \frac{1}{n}$，$n$ 为输入神经元 (特征) 数目。同样考虑另一边，得到 $\mathrm{Var}(\boldsymbol{W}) = \frac{1}{m}$，$m$ 为输出神经元 (特征) 数目。我们综合考虑二者，得到 $\mathrm{Var}(\boldsymbol{W}) = \frac{2}{n+m}$。

- 正态分布的 Glorot 初始化 (Glorot Normal)：它是从以 0 为中心，标准差为 $std = \sqrt{\frac{2}{fan_{in} + fan_{out}}}$ 的截断正态分布中抽取样本，其中 fan_{in} 是权值张量中的输入单位的数量，fan_{out} 是权值张量中的输出单位的数量。
- 均匀分布的 Glorot 初始化 (Glorot Uniform)：它是从 $[-limit, limit]$ 中的均匀分布中抽取样本，其中 $limit = \sqrt{\frac{6}{fan_{in} + fan_{out}}}$ (将均匀分布的方差代入计算可得)，fan_{in} 是权值张量中的输入单位的数量，fan_{out} 是权值张量中的输出单位的数量。

3、Kaiming 初始化方法

Kaiming 初始化，也称为 He 初始化，也称之为 MSRA 初始化。前面的 Glorot 初始化要求了激活函数关于 0 对称，因此不适用于 ReLU。Kaiming 初始化的想法是一个好的初始化应该在各个层的状态值 $\boldsymbol{z}$ 的方差保持一致，即对于第 i 层和第 j 层，有 $\mathrm{Var}(z^i) = \mathrm{Var}(z^j)$。这边只假设初始化的均值为 0，即 $\mathbb{E}(\boldsymbol{W}) = 0$：

$$\begin{aligned}
\mathrm{Var}(z^i) &= n\,\mathrm{Var}(\boldsymbol{W}^\top a^{i-1}) \\
&= n\left[\mathbb{E}(\boldsymbol{W}^\top a^{i-1})^2 - (\mathbb{E}(\boldsymbol{W}^\top a^{i-1}))^2\right] \\
&= n\left[\mathbb{E}(\boldsymbol{W}^2)\mathbb{E}((a^{i-1})^2) - (\mathbb{E}(\boldsymbol{W}))^2(\mathbb{E}(a^{i-1}))^2\right] \\
&= n\,\mathbb{E}(\boldsymbol{W}^2)\mathbb{E}((a^{i-1})^2) \\
&= n\,\mathrm{Var}(\boldsymbol{W})\mathbb{E}((a^{i-1})^2)
\end{aligned} \tag{11}$$

而再计算 $\mathbb{E}((a^{i-1})^2)$，需要注意的是，由于 $\mathbb{E}(\boldsymbol{W}) = 0$ 且 $\boldsymbol{W}$ 与 a^{i-1} 独立，故 $\mathbb{E}(z^i) = 0$：

$$\begin{aligned}
\mathbb{E}(a^{i-1})^2 &= \mathbb{E}(g(z^{i-1}))^2 \\
&= \int_{-\infty}^{\infty} p(z^{i-1})(g(z^{i-1}))^2 dz^{i-1} \\
&= \int_{-\infty}^{0} p(z^{i-1})(g(z^{i-1}))^2 dz^{i-1} + \int_{0}^{\infty} p(z^{i-1})(g(z^{i-1}))^2 dz^{i-1} \\
&= 0 + \int_{0}^{\infty} p(z^{i-1})(z^{i-1})^2 dz^{i-1} \\
&= \frac{1}{2}\int_{-\infty}^{\infty} p(z^{i-1})(z^{i-1})^2 dz^{i-1} \\
&= \frac{1}{2}\mathbb{E}(z^{i-1})^2 \\
&= \frac{1}{2}\mathrm{Var}(z^{i-1})
\end{aligned} \tag{12}$$

于是我们得到 $\mathrm{Var}(z^i) = \frac{1}{2} n\,\mathrm{Var}(\boldsymbol{W})\,\mathrm{Var}(z^{i-1})$，由于 $\mathrm{Var}(z^i) = \mathrm{Var}(z^{i-1})$，可以得到 $\mathrm{Var}(\boldsymbol{W}) = \frac{2}{n}$。

- 正态分布的 Kaiming 初始化 (He Normal)：它是从以 0 为中心，标准差为 $std = \sqrt{\frac{2}{fan_{in}}}$ 的截断正态分布中抽取样本，其中 fan_{in} 是权值张量中的输入单位的数量。
- 均匀分布的 Kaiming 初始化 (He Uniform)：它是从 $[-limit, limit]$ 中的均匀分布中抽取样本，其中 $limit = \sqrt{\frac{6}{fan_{in}}}$，$fan_{in}$ 是权值张量中的输入单位的数量。

4、Batch Normalization 初始化方法

Batch Normalization 会在后文介绍，使用 BN 时，减少了网络对参数初始值尺度的依赖，此时使用较小的标准差 (如 0.01) 进行初始化即可。

5、Pre-Train 初始化方法

借助预训练模型中参数作为新任务参数初始化的方式是一种简便易行且十分有效的模型参数初始化方法，这也是目前的主流方法之一。

[17]:
```python
def calc_fan(weight_shape):
    """
    对权重矩阵计算 fan-in 和 fan-out

    参数说明:
    weight_shape: 权重形状
    """
```

```
    if len(weight_shape) == 2:  # 暂先考虑维数为 2
        fan_in, fan_out = weight_shape
    else:
        raise ValueError("Unrecognized weight dimension: {}".format(weight_shape))
    return fan_in, fan_out
```

[18]:
```
def random_uniform(weight_shape, b=1.0):
    """
    初始化网络权重 W--- 基于 Uniform(-b, b)

    参数说明:
    weight_shape: 权重形状
    """
    return np.random.uniform(-b, b, size=weight_shape)
```

[19]:
```
def random_normal(weight_shape, std=1.0):
    """
    初始化网络权重 W--- 基于 TruncatedNormal(0, std)

    参数说明:
    weight_shape: 权重形状
    std: 权重标准差
    """
    return truncated_normal(0, std, weight_shape)
```

[20]:
```
def he_uniform(weight_shape):
    """
    初始化网络权重 W--- 基于 Uniform(-b, b)，其中 b=sqrt(6/fan_in)，常用于 ReLU 激活层

    参数说明:
    weight_shape: 权重形状
    """
    fan_in, fan_out = calc_fan(weight_shape)
    b = np.sqrt(6 / fan_in)
    return np.random.uniform(-b, b, size=weight_shape)
```

[21]:
```
def he_normal(weight_shape):
    """
    初始化网络权重 W--- 基于 TruncatedNormal(0, std)，其中 std=2/fan_in，常用于 ReLU 激活层

    参数说明:
    weight_shape: 权重形状
    """
    fan_in, fan_out = calc_fan(weight_shape)
    std = np.sqrt(2 / fan_in)
    return truncated_normal(0, std, weight_shape)
```

[22]:
```
def glorot_uniform(weight_shape, gain=1.0):
    """
    初始化网络权重 W--- 基于 Uniform(-b, b)，其中 b=gain*sqrt(6/(fan_in+fan_out))，
                        常用于 tanh 和 sigmoid 激活层

    参数说明:
    weight_shape: 权重形状
    """
    fan_in, fan_out = calc_fan(weight_shape)
    b = gain * np.sqrt(6 / (fan_in + fan_out))
    return np.random.uniform(-b, b, size=weight_shape)
```

[23]:
```
def glorot_normal(weight_shape, gain=1.0):
    """
```

```
    初始化网络权重 W--- 基于 TruncatedNormal(0, std)，其中 std=gain^2*2/(fan_in+fan_out)，
                        常用于 tanh 和 sigmoid 激活层

    参数说明:
    weight_shape: 权重形状
    """
    fan_in, fan_out = calc_fan(weight_shape)
    std = gain * np.sqrt(2 / (fan_in + fan_out))
    return truncated_normal(0, std, weight_shape)
```

[24]:
```
def truncated_normal(mean, std, out_shape):
    """
    通过拒绝采样生成截断正态分布

    参数说明:
    mean: 正态分布均值
    std: 正态分布标准差
    out_shape: 矩阵形状
    """
    samples = np.random.normal(loc=mean, scale=std, size=out_shape)
    reject = np.logical_or(samples >= mean + 2 * std, samples <= mean - 2 * std)
    while any(reject.flatten()):
        resamples = np.random.normal(loc=mean, scale=std, size=reject.sum())
        samples[reject] = resamples
        reject = np.logical_or(samples >= mean + 2 * std, samples <= mean - 2 * std)
    return samples
```

3 批标准化

批标准化 (Batch Normalization) 并不是一个优化算法，而是一个**自适应的重参数化**的方法，试图解决训练**非常深的模型**的困难。

BatchNorm 是基于 SGD 的，因为**深层神经网络**在**非线性变换前**的激活输入值随着网络深度加深或者在训练过程中，其**分布逐渐发生偏移或者变动** (Covariate Shift)。之所以训练收敛慢，一般是**整体分布**逐渐往**非线性函数的取值区间的上下限两端靠近** (如对于 sigmoid 函数来说，激活输入值是大的负值或正值)，所以这导致反向传播时低层神经网络的**梯度消失**。

BatchNorm 是通过一定的规范化手段，把每层神经网络任意神经元这个输入值的分布强行拉回到**均值为 0、方差为 1 的标准正态分布**，把越来越偏的分布拉回标准分布，这样使得激活输入值落在**非线性函数对输入比较敏感的区域**，避免梯度消失问题产生，梯度变大使得学习收敛速度快。

BatchNorm 训练阶段的实现包括两个步骤：

- 对于某个神经元，如果 Batch 为 m 个样本，首先计算在当前神经元处的均值 μ 与方差 σ。再对数据进行规范化，使得输入每个特征的分布均值为 0，方差为 1。再将输入经过非线性函数。
- 步骤一让每一层网络的输入数据分布都变得稳定，但却导致了数据表达能力的缺失，因为通过变换操作改变了原有数据的信息表达。因此，引入参数 scale γ 和 offset β，再对规范化后的数据进行线性变换，**恢复数据本身的表达能力**。

具体如下：

$$
\begin{aligned}
\mu &\leftarrow \frac{1}{m}\sum_{i=1}^{m}\boldsymbol{x}^{(i)} \\
\sigma^2 &\leftarrow \frac{1}{m}\sum_{i=1}^{m}(\boldsymbol{x}^{(i)}-\mu)^2 \\
\hat{\boldsymbol{x}}^{(i)} &\leftarrow \frac{\boldsymbol{x}^{(i)}-\mu}{\sqrt{\sigma^2+\epsilon}} \\
\hat{\boldsymbol{z}}^{(i)} &\leftarrow \gamma\hat{\boldsymbol{x}}^{(i)}+\beta \\
\boldsymbol{a}^{(i)} &\leftarrow f(\hat{\boldsymbol{z}}^{(i)})
\end{aligned}
\tag{13}
$$

测试阶段使用训练阶段**整个样本的统计量**来对测试数据归一化。训练阶段保留每组 Batch 的 μ_{batch} 和 σ_{batch}，于是可以得到均值与方差的**无偏估计**。如下：

$$
\begin{aligned}
\mu &\leftarrow \mathbb{E}(\mu_{batch}) \\
\sigma^2 &\leftarrow \frac{m}{m-1}\mathbb{E}(\sigma^2_{batch})
\end{aligned}
\tag{14}
$$

反向传播过程：

- 首先，可以计算得出：

$$\frac{\partial J}{\partial \gamma} = \frac{\partial J}{\partial \hat{\boldsymbol{z}}^{(i)}} \cdot \frac{\partial \hat{\boldsymbol{z}}^{(i)}}{\partial \gamma} = \boxed{\sum_{i=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{z}}^{(i)}} \cdot \hat{\boldsymbol{x}}^{(i)}} \tag{15}$$

$$\frac{\partial J}{\partial \beta} = \frac{\partial J}{\partial \hat{\boldsymbol{z}}^{(i)}} \cdot \frac{\partial \hat{\boldsymbol{z}}^{(i)}}{\partial \beta} = \boxed{\sum_{i=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{z}}^{(i)}}} \tag{16}$$

$$\frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(i)}} = \frac{\partial J}{\partial \hat{\boldsymbol{z}}^{(i)}} \cdot \frac{\partial \hat{\boldsymbol{z}}^{(i)}}{\partial \hat{\boldsymbol{x}}^{(i)}} = \boxed{\frac{\partial J}{\partial \hat{\boldsymbol{z}}^{(i)}} \cdot \gamma} \tag{17}$$

然后，根据

$$\begin{cases} \dfrac{\partial \hat{\boldsymbol{x}}^{(i)}}{\partial \mu} = \dfrac{1}{\sqrt{\sigma^2 + \epsilon}} \cdot (-1) \\ \dfrac{\partial \sigma^2}{\partial \mu} = \dfrac{1}{m} \displaystyle\sum_{i=1}^{m} 2 \cdot (\boldsymbol{x}^{(i)} - \mu) \cdot (-1) \\ \dfrac{\partial \hat{\boldsymbol{x}}}{\partial \sigma^2} = \displaystyle\sum_{i=1}^{m} (\boldsymbol{x}^{(i)} - \mu) \cdot (-0.5) \cdot (\sigma^2 + \epsilon)^{-0.5-1} \end{cases} \tag{18}$$

可以得到：

$$\frac{\partial J}{\partial \sigma^2} = \boxed{\frac{\partial J}{\partial \hat{\boldsymbol{x}}} \cdot \frac{\partial \hat{\boldsymbol{x}}}{\partial \sigma^2}} \tag{19}$$

以及：

$$\begin{aligned} \frac{\partial J}{\partial \mu} &= \left(\sum_{i=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(i)}} \cdot \frac{-1}{\sqrt{\sigma^2 + \epsilon}} \right) + \left(\frac{\partial J}{\partial \sigma^2} \cdot \frac{1}{m} \sum_{i=1}^{m} -2(\boldsymbol{x}^{(i)} - \mu) \right) \\ &= \left(\sum_{i=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(i)}} \cdot \frac{-1}{\sqrt{\sigma^2 + \epsilon}} \right) + \left(\frac{\partial J}{\partial \sigma^2} \cdot (-2) \cdot \left(\frac{1}{m} \sum_{i=1}^{m} \boldsymbol{x}^{(i)} - \frac{1}{m} \sum_{i=1}^{m} \mu \right) \right) \\ &= \left(\sum_{i=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(i)}} \cdot \frac{-1}{\sqrt{\sigma^2 + \epsilon}} \right) + \left(\frac{\partial J}{\partial \sigma^2} \cdot (-2) \cdot \left(\mu - \frac{m \cdot \mu}{m} \right) \right) \\ &= \boxed{\sum_{i=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(i)}} \cdot \frac{-1}{\sqrt{\sigma^2 + \epsilon}}} \end{aligned} \tag{20}$$

再根据

$$\begin{cases} \dfrac{\partial \hat{\boldsymbol{x}}^{(i)}}{\partial \boldsymbol{x}^{(i)}} = \dfrac{1}{\sqrt{\sigma^2 + \epsilon}} \\ \dfrac{\partial \mu}{\partial \boldsymbol{x}^{(i)}} = \dfrac{1}{m} \\ \dfrac{\partial \sigma^2}{\partial \boldsymbol{x}^{(i)}} = \dfrac{2(\boldsymbol{x}^{(i)} - \mu)}{m} \end{cases} \tag{21}$$

计算得出：

$$\begin{aligned} \frac{\partial J}{\partial \boldsymbol{x}^{(i)}} &= \left(\frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(i)}} \cdot \frac{1}{\sqrt{\sigma^2 + \epsilon}} \right) + \left(\frac{\partial J}{\partial \mu} \cdot \frac{1}{m} \right) + \left(\frac{\partial J}{\partial \sigma^2} \cdot \frac{2(\boldsymbol{x}^{(i)} - \mu)}{m} \right) \\ &= \left(\frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(i)}} \cdot \frac{1}{\sqrt{\sigma^2 + \epsilon}} \right) + \left(\frac{1}{m} \sum_{j=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(j)}} \cdot \frac{-1}{\sqrt{\sigma^2 + \epsilon}} \right) - \left(0.5 \sum_{j=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(j)}} \cdot (\boldsymbol{x}^{(j)} - \mu) \cdot (\sigma^2 + \epsilon)^{-1.5} \cdot \frac{2(\boldsymbol{x}^{(i)} - \mu)}{m} \right) \\ &= \left(\frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(i)}} \cdot (\sigma^2 + \epsilon)^{-0.5} \right) - \left(\frac{(\sigma^2 + \epsilon)^{-0.5}}{m} \sum_{j=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(j)}} \right) - \left(\frac{(\sigma^2 + \epsilon)^{-0.5}}{m} \cdot \frac{\boldsymbol{x}^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}} \sum_{j=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(j)}} \cdot \frac{(\boldsymbol{x}^{(j)} - \mu)}{\sqrt{\sigma^2 + \epsilon}} \right) \\ &= \left(\frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(i)}} \cdot (\sigma^2 + \epsilon)^{-0.5} \right) - \left(\frac{(\sigma^2 + \epsilon)^{-0.5}}{m} \sum_{j=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(j)}} \right) - \left(\frac{(\sigma^2 + \epsilon)^{-0.5}}{m} \cdot \hat{\boldsymbol{x}}^{(i)} \sum_{j=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(j)}} \cdot \hat{\boldsymbol{x}}^{(j)} \right) \\ &= \boxed{\frac{m \dfrac{\partial J}{\partial \hat{\boldsymbol{x}}^{(i)}} - \displaystyle\sum_{j=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(j)}} - \hat{\boldsymbol{x}}^{(i)} \sum_{j=1}^{m} \frac{\partial J}{\partial \hat{\boldsymbol{x}}^{(j)}} \cdot \hat{\boldsymbol{x}}^{(j)}}{m\sqrt{\sigma^2 + \epsilon}}} \end{aligned} \tag{22}$$

[25]:
```
from chapter6 import LayerBase, CrossEntropy, OrderedDict, softmax, FullyConnected
```

[26]:
```
class BatchNorm1D(LayerBase):


    def __init__(self, momentum=0.9, epsilon=1e-5, optimizer=None):
        """
        参数说明:
```

```
        momentum: 动量项，越趋于 1 表示对当前 Batch 的依赖程度越小，running_mean 和 running_var 的计算越平滑
                  float 型 (default: 0.9)

        epsilon: 避免除数为 0，float 型 (default : 1e-5)
        optimizer: 优化器
        """
        super().__init__(optimizer)
        self.n_in = None
        self.n_out = None
        self.epsilon = epsilon
        self.momentum = momentum
        self.params = {
            "scaler": None,
            "intercept": None,
            "running_var": None,
            "running_mean": None,
        }
        self.is_initialized = False

    def _init_params(self):
        scaler = np.random.rand(self.n_in)
        intercept = np.zeros(self.n_in)
        running_mean = np.zeros(self.n_in)
        running_var = np.ones(self.n_in)
        self.params = {
            "scaler": scaler,
            "intercept": intercept,
            "running_mean": running_mean,
            "running_var": running_var,
        }
        self.gradients = {
            "scaler": np.zeros_like(scaler),
            "intercept": np.zeros_like(intercept),
        }
        self.is_initialized = True

    def reset_running_stats(self):
        self.params["running_mean"] = np.zeros(self.n_in)
        self.params["running_var"] = np.ones(self.n_in)

    def forward(self, X, is_train=True, retain_derived=True):
        """
        Batch 训练时 BN 的前向传播，原理见上文。

        [train]: Y = scaler * norm(X) + intercept，其中 norm(X) = (X - mean(X)) / sqrt(var(X) + epsilon)

        [test]: Y = scaler * running_norm(X) + intercept,
                    其中 running_norm(X) = (X - running_mean) / sqrt(running_var + epsilon)

        参数说明:
        X: 输入数组，为 (n_samples, n_in)，float 型
        is_train: 是否为训练阶段，bool 型
        retain_derived: 是否保留中间变量，以便反向传播时再次使用，bool 型
        """
        if not self.is_initialized:
            self.n_in = self.n_out = X.shape[1]
            self._init_params()
        epsi, momentum = self.hyperparams["epsilon"], self.hyperparams["momentum"]
        rm, rv = self.params["running_mean"], self.params["running_var"]
```

```
        scaler, intercept = self.params["scaler"], self.params["intercept"]
        X_mean, X_var = self.params["running_mean"], self.params["running_var"]
        if is_train and retain_derived:
            X_mean, X_var = X.mean(axis=0), X.var(axis=0)
            self.params["running_mean"] = momentum * rm + (1.0 - momentum) * X_mean
            self.params["running_var"] = momentum * rv + (1.0 - momentum) * X_var
        if retain_derived:
            self.X.append(X)
        X_hat = (X - X_mean) / np.sqrt(X_var + epsi)
        y = scaler * X_hat + intercept
        return y

    def backward(self, dLda, retain_grads=True):
        """
        BN 的反向传播，原理见上文。

        参数说明：
        dLda：关于损失的梯度，为 (n_samples, n_out)，float 型
        retain_grads：是否计算中间变量的参数梯度，bool 型
        """
        if not isinstance(dLda, list):
            dLda = [dLda]
        dX = []
        X = self.X
        for da, x in zip(dLda, X):
            dx, dScaler, dIntercept = self._bwd(da, x)
            dX.append(dx)
            if retain_grads:
                self.gradients["scaler"] += dScaler
                self.gradients["intercept"] += dIntercept
        return dX[0] if len(X) == 1 else dX

    def _bwd(self, dLda, X):
        scaler = self.params["scaler"]
        epsi = self.hyperparams["epsilon"]
        n_ex, n_in = X.shape
        X_mean, X_var = X.mean(axis=0), X.var(axis=0)
        X_hat = (X - X_mean) / np.sqrt(X_var + epsi)
        dIntercept = dLda.sum(axis=0)
        dScaler = np.sum(dLda * X_hat, axis=0)
        dX_hat = dLda * scaler
        dX = (n_ex * dX_hat - dX_hat.sum(axis=0) - X_hat * (dX_hat * X_hat).sum(axis=0)) / (
            n_ex * np.sqrt(X_var + epsi)
        )
        return dX, dScaler, dIntercept

    @property
    def hyperparams(self):
        return {
            "layer": "BatchNorm1D",
            "acti_fn": None,
            "n_in": self.n_in,
            "n_out": self.n_out,
            "epsilon": self.epsilon,
            "momentum": self.momentum,
            "optimizer": {
                "cache": self.optimizer.cache,
                "hyperparams": self.optimizer.hyperparams,
            },
```

```
        }
```

引入 BatchNorm，MNIST 数据集测试

[27]:
```
def load_data(path="../data/mnist/mnist.npz"):
    f = np.load(path)
    X_train, y_train = f['x_train'], f['y_train']
    X_test, y_test = f['x_test'], f['y_test']
    f.close()
    return (X_train, y_train), (X_test, y_test)

(X_train, y_train), (X_test, y_test) = load_data()
y_train = np.eye(10)[y_train.astype(int)]
y_test = np.eye(10)[y_test.astype(int)]
X_train = X_train.reshape(-1, X_train.shape[1]*X_train.shape[2]).astype('float32')
X_test = X_test.reshape(-1, X_test.shape[1]*X_test.shape[2]).astype('float32')
print(X_train.shape, y_train.shape)
N = 20000    # 取 20000 条数据用以训练
indices = np.random.permutation(range(X_train.shape[0]))[:N]
X_train, y_train = X_train[indices], y_train[indices]
print(X_train.shape, y_train.shape)
X_train /= 255
X_train = (X_train - 0.5) * 2
X_test /= 255
X_test = (X_test - 0.5) * 2
```

```
(60000, 784) (60000, 10)
(20000, 784) (20000, 10)
```

[28]:
```
def minibatch(X, batchsize=256, shuffle=True):
    """
    函数作用：将数据集分割成 batch， 基于 mini batch 训练。
    """
    N = X.shape[0]
    idx = np.arange(N)
    n_batches = int(np.ceil(N / batchsize))
    if shuffle:
        np.random.shuffle(idx)
    def mb_generator():
        for i in range(n_batches):
            yield idx[i * batchsize : (i + 1) * batchsize]
    return mb_generator(), n_batches
```

```
class DFN(object):

    def __init__(
        self,
        hidden_dims_1=None,
        hidden_dims_2=None,
        optimizer="sgd(lr=0.01)",
        init_w="std_normal",
        loss=CrossEntropy()
    ):
        self.optimizer = optimizer
        self.init_w = init_w
        self.loss = loss
        self.hidden_dims_1 = hidden_dims_1
        self.hidden_dims_2 = hidden_dims_2
        self.is_initialized = False
```

```
def _set_params(self):
    """
    函数作用：模型初始化
    FC1 -> Sigmoid -> BN -> FC2 -> Softmax
    """
    self.layers = OrderedDict()
    self.layers["FC1"] = FullyConnected(
        n_out=self.hidden_dims_1,
        acti_fn="sigmoid",
        init_w=self.init_w,
        optimizer=self.optimizer
    )
    self.layers["BN"] = BatchNorm1D(optimizer=self.optimizer)
    self.layers["FC2"] = FullyConnected(
        n_out=self.hidden_dims_2,
        acti_fn="affine(slope=1, intercept=0)",
        init_w=self.init_w,
        optimizer=self.optimizer
    )
    self.is_initialized = True

def forward(self, X_train, is_train=True):
    Xs = {}
    out = X_train
    for k, v in self.layers.items():
        Xs[k] = out
        try:  # 考虑 BN
            out = v.forward(out, is_train=is_train)
        except:
            out = v.forward(out)
    return out, Xs

def backward(self, grad):
    dXs = {}
    out = grad
    for k, v in reversed(list(self.layers.items())):
        dXs[k] = out
        out = v.backward(out)
    return out, dXs

def update(self):
    """
    函数作用：梯度更新
    """
    for k, v in reversed(list(self.layers.items())):
        v.update()
    self.flush_gradients()

def flush_gradients(self, curr_loss=None):
    """
    函数作用：更新后重置梯度
    """
    for k, v in self.layers.items():
        v.flush_gradients()

def fit(self, X_train, y_train, n_epochs=20, batch_size=64, verbose=False):
    """
    参数说明：
```

```
        X_train: 训练数据
        y_train: 训练数据标签
        n_epochs: epoch 次数
        batch_size: 每次 epoch 的 batch size
        verbose: 是否每个 batch 输出损失
        """
        self.verbose = verbose
        self.n_epochs = n_epochs
        self.batch_size = batch_size
        if not self.is_initialized:
            self.n_features = X_train.shape[1]
            self._set_params()
        prev_loss = np.inf
        for i in range(n_epochs):
            loss, epoch_start = 0.0, time.time()
            batch_generator, n_batch = minibatch(X_train, self.batch_size, shuffle=True)
            for j, batch_idx in enumerate(batch_generator):
                batch_len, batch_start = len(batch_idx), time.time()
                X_batch, y_batch = X_train[batch_idx], y_train[batch_idx]
                out, _ = self.forward(X_batch, is_train=True)
                y_pred_batch = softmax(out)
                batch_loss = self.loss(y_batch, y_pred_batch)
                grad = self.loss.grad(y_batch, y_pred_batch)
                _, _ = self.backward(grad)
                self.update()
                loss += batch_loss
                if self.verbose:
                    fstr = "\t[Batch {}/{}] Train loss: {:.3f} ({:.1f}s/batch)"
                    print(fstr.format(j + 1, n_batch, batch_loss, time.time() - batch_start))
            loss /= n_batch
            fstr = "[Epoch {}] Avg. loss: {:.3f}  Delta: {:.3f} ({:.2f}m/epoch)"
            print(fstr.format(i + 1, loss, prev_loss - loss, (time.time() - epoch_start) / 60.0))
            prev_loss = loss

    def evaluate(self, X_test, y_test, batch_size=128):
        acc = 0.0
        batch_generator, n_batch = minibatch(X_test, batch_size, shuffle=True)
        for j, batch_idx in enumerate(batch_generator):
            batch_len, batch_start = len(batch_idx), time.time()
            X_batch, y_batch = X_test[batch_idx], y_test[batch_idx]
            y_pred_batch, _ = self.forward(X_batch, is_train=False)
            y_pred_batch = np.argmax(y_pred_batch, axis=1)
            y_batch = np.argmax(y_batch, axis=1)
            acc += np.sum(y_pred_batch == y_batch)
        return acc / X_test.shape[0]

    @property
    def hyperparams(self):
        return {
            "init_w": self.init_w,
            "loss": str(self.loss),
            "optimizer": self.optimizer,
            "hidden_dims_1": self.hidden_dims_1,
            "hidden_dims_2": self.hidden_dims_2,
            "components": {k: v.params for k, v in self.layers.items()}
        }
```

[29]:
```
model = DFN(hidden_dims_1=200, hidden_dims_2=10)
model.fit(X_train, y_train, n_epochs=20, batch_size=64)
```

```
[Epoch 1] Avg. loss: 0.979  Delta: inf (0.01m/epoch)
[Epoch 2] Avg. loss: 0.473  Delta: 0.506 (0.01m/epoch)
[Epoch 3] Avg. loss: 0.395  Delta: 0.078 (0.01m/epoch)
[Epoch 4] Avg. loss: 0.362  Delta: 0.033 (0.01m/epoch)
[Epoch 5] Avg. loss: 0.342  Delta: 0.019 (0.02m/epoch)
[Epoch 6] Avg. loss: 0.326  Delta: 0.016 (0.01m/epoch)
[Epoch 7] Avg. loss: 0.315  Delta: 0.011 (0.02m/epoch)
[Epoch 8] Avg. loss: 0.307  Delta: 0.008 (0.01m/epoch)
[Epoch 9] Avg. loss: 0.299  Delta: 0.008 (0.01m/epoch)
[Epoch 10] Avg. loss: 0.292  Delta: 0.008 (0.02m/epoch)
[Epoch 11] Avg. loss: 0.287  Delta: 0.005 (0.02m/epoch)
[Epoch 12] Avg. loss: 0.280  Delta: 0.007 (0.02m/epoch)
[Epoch 13] Avg. loss: 0.275  Delta: 0.005 (0.01m/epoch)
[Epoch 14] Avg. loss: 0.270  Delta: 0.005 (0.01m/epoch)
[Epoch 15] Avg. loss: 0.262  Delta: 0.008 (0.02m/epoch)
[Epoch 16] Avg. loss: 0.254  Delta: 0.008 (0.02m/epoch)
[Epoch 17] Avg. loss: 0.249  Delta: 0.005 (0.01m/epoch)
[Epoch 18] Avg. loss: 0.244  Delta: 0.005 (0.02m/epoch)
[Epoch 19] Avg. loss: 0.234  Delta: 0.010 (0.01m/epoch)
[Epoch 20] Avg. loss: 0.232  Delta: 0.002 (0.01m/epoch)
```

[30]:
```
print("accuracy:{}".format(model.evaluate(X_test, y_test)))
```

```
accuracy:0.9258
```

4 坐标下降

坐标下降 (Coordinate Descent) 是一种非梯度优化算法。算法在每次迭代中，在当前点处**沿一个坐标方向**进行**一维搜索**以求得一个函数的局部极小值。在整个过程中**循环使用不同的坐标方向**。如我们相对某个单一变量 x_i 最小化 $f(x)$，然后再相对另一个变量 x_j 计算，反复循环所有的变量，会保证到达 (局部) 极小值。这种做法被称为坐标下降，因为我们**一次优化一个坐标**。

对于不可拆分的函数而言，算法可能无法在较小的迭代步数中求得最优解。如果当**一个变量的值很大程度地影响另一个变量的最优值**时，坐标下降不是一个很好的方法。但在**稀疏矩阵上**的计算速度非常快，同时也是 Lasso 回归最快的解法。

5 Polyak 平均

Polyak 平均会平均优化算法在参数空间访问轨迹中的几个点。如果 t 次迭代梯度下降访问了点 $\theta^{(1)} \dots \theta^{(t)}$，那么 Polyak 平均算法输出的是：

$$\hat{\theta}^{(t)} = \frac{1}{t} \sum_i \theta^{(i)} \tag{23}$$

在梯度下降应用于某些问题，比如凸问题时，这种方法具有较强的收敛保证。当 Polyak 平均于**非凸问题时**，通常会用**指数衰减计算平均值**：

$$\hat{\theta}^{(t)} = \alpha \hat{\theta}^{(t-1)} + (1 - \alpha)\theta^{(t)} \tag{24}$$

滑动平均 (Exponential Moving Average)，或者叫做指数加权平均，可以用来估计变量的局部均值，使得**变量的更新与一段时间内的历史取值**有关。

对于 SGD，在**训练过程**仍然使用原来**不带滑动平均的权重**，在**测试过程**中使用**带滑动平均的权重**作为神经网络的权重，这样在测试数据上效果更好，带滑动平均的权重更新更加平滑。

6 监督预训练

如果模型太复杂难以优化，或是如果**任务非常困难**，直接训练模型来解决特定任务的挑战可能太大。**训练模型来求解一个简化的问题**，然后**转移**到最后的问题，有时也会更有效些。这些在直接训练目标模型求解目标问题之前，训练简单模型求解简化问题的方法统称为**预训练**。

贪心算法 (Greedy Algorithm) 将**问题分解成许多部分**，然后**独立地在每个部分求解最优值**。结合各个最佳的部分不能保证得到一个最佳的完整解，但贪心算法计算上比求解最优联合解的算法高效得多，并且贪心算法的解在不是最优的情况下，往往也是可以接受的。贪心算法也可以紧接一个**精调** (fine-tuning) 阶段，联合优化算法搜索全问题的最优解。使用贪心解初始化联合优化算法，可以极大地加速算法，并提高寻找到的解的质量。

一个与监督预训练有关的方法扩展了**迁移学习的想法**，另一条相关的工作线是 **FitNets 方法**。

7 设计有助于优化的模型

改进优化的最好方法并不总是改进优化算法，相反，在深度学习中的许多改进来自设计易于优化的模型。在实践中，**选择一族容易优化的模型**比使用一个强大的优化算法重要。

现代神经网络的设计选择体现在**层之间的线性变换**，**几乎处处可导的激活函数**，和**大部分定义域都有明显的梯度**，特别是**创新的模型**，比如 LSTM，整流线性单元和 maxout 单元都比先前的模型，比如 sigmoid 单元的深度网络，使用更多的线性函数，使得这些模型都有简化优化的作用。现代神经网络的设计方案旨在使其**局部梯度信息合理地对应着移动向一个遥远的解**。

[32]:
```
import numpy


print("numpy:", numpy.__version__)
```

```
numpy: 1.14.5
```