

深度学习中的优化

概览

1. 学习与优化
2. 基本优化算法
3. 自适应学习率算法

1. 学习与优化

“学习”定义

Mitchell (1997) 提出“学习”的简洁定义：对于某类任务 T 和性能度量 P ，一个计算机程序被认为可以从经验 E 中学习是指，通过经验 E 改进后，它在任务 T 上由性能度量 P 衡量的性能有所提升。

需要注意的是：性能度量 P 并不是在经验 E 中进行衡量的。

代价函数

我们的目标是提高模型性能，即最小化取自数据生成分布 p_{data} 的期望。则其目标函数为：

$$J^*(\theta) = \mathbb{E}_{(x,y) \sim p_{data}} L(f(x;\theta), y)$$

其中 L 是每个样本的损失函数， $f(x;\theta)$ 是输入 x 时所预的输出， p_{data} 是真实分布。监督学习中， y 是目标输出。

代价函数

事实上，我们无法使用 P_{data} 来进行模型的训练，以期达到最好的性能，我们只能使用有限的训练集对模型进行训练。所以模型在经验分布 \hat{P}_{data} 上的代价函数为：

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(x,y) \sim \hat{P}_{data}} L(f(\mathbf{x}; \boldsymbol{\theta}), y)$$

与纯优化 J 不同，我们希望通过降低代价函数 J 来提高性能 P 。

1.1 经验风险最小化

经验风险最小化

我们的目标是降低泛化误差 J^* ，这个数据量被称为**风险 (risk)**。然而通常，我们并不知道真实分布 p_{data} 。这意味着我们只能使用训练集的经验分布 \hat{p}_{data} 替代真实分布。现在我们将最小化**经验分布 (empirical risk)**：

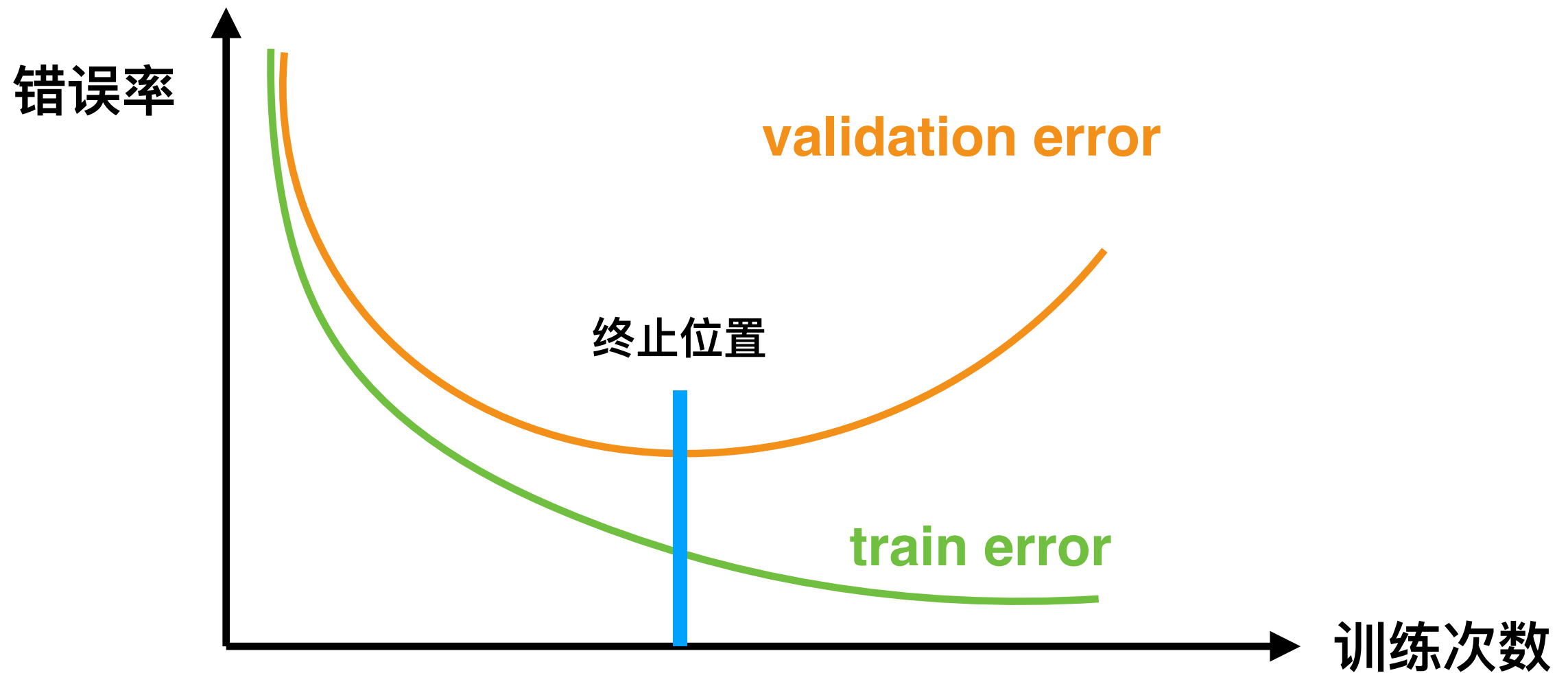
$$E_{(x,y) \sim \hat{p}_{data}} [L(f(\mathbf{x}; \boldsymbol{\theta}), y)] = \frac{1}{m} \sum_i^m L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$$

基于最小化平均训练误差的训练过程称之为**经验风险最小化 (empirical risk minimization)**。我们希望直接优化经验风险，使得大大降低风险。

经验风险最小化的缺点

在很多情况下，经验风险最小化并不可行。经验风险最小化很容易导致过拟合。高容量的模型会简单的记住数据集。最有效的现代优化算法是基于梯度下降的。

提前终止



与纯优化不同的是，提前终止时，损失函数仍然有较大的导数，而纯优化终止时导数较小。

1.2 批量算法与小批量 算法

批量与小批量算法

机器学习算法的代价函数通常可以分解为训练样本上的求和，再使用优化算法计算参数的每一次更新时通常仅使用一部分样本来估计代价函数的期望。

例如，最大似然估计问题可以在对数空间上分解成对各个样本的总和：

$$\boldsymbol{\theta}_{\text{ML}} = \arg \max_{\boldsymbol{\theta}} \sum_{i=1}^m \log p_{\text{model}}(\mathbf{x}^{(i)}, y^{(i)}; \boldsymbol{\theta})$$

最大化这个总和等价于最大化训练集在经验分布上的期望：

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}, y \sim \hat{P}_{\text{data}}} \log p_{\text{model}}(\mathbf{x}^{(i)}, y^{(i)}; \boldsymbol{\theta})$$

批量与小批量算法

优化算法常用代价函数中的梯度，即梯度为：

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\mathbf{x}, y \sim \hat{P}_{data}} \nabla_{\theta} \log p_{\text{model}}(\mathbf{x}^{(i)}, y^{(i)}; \theta)$$

准确计算这个期望的代价非常大，因为需要遍历整个数据集上的每个样本。

通常我们使用小批量样本获得梯度的统计估计。因为首先使用更多样本来估计梯度的回报是小于线性的，其次由于训练集往往是冗余的，训练集中大多数样本对梯度做出了非常相似的贡献。

批量与小批量术语解释

- 使用整个训练集的优化算法被称为**批量 (batch)** 或**确定性 (deterministic)** 算法；
- **小批量 (mini batch)** 算法是指使用小批量训练集的优化算法；
- 每次只使用单个样本的优化算法被称为**随机 (stochastic)** 或**在线 (online)** 算法；
- 在线算法通常是指从连续产生样本的数据流中抽取样本的情况；
- 随机算法通常是指从固定大小的训练集中遍历多次采样的情况。

小练习

- 判断以下任务中分别使用的是哪种算法？
 - 从手写数字数据集MNIST中随机抽取32个样本完成一次训练；
 - 根据用户访问一个网站的实时行为训练此用户的偏好属性模型
 - 使用收集到的100张肿瘤图片同时进行肿瘤性质判断模型的训练；
 - 从收集到的100张肿瘤图片中每次随机抽取一张进行肿瘤性质判断模型的训练。

小批量大小的决定因素

- 更大的批量会计算更精确的梯度估计，但是回报却是小于线性的；
- 极小批量通常难以充分利用计算机的多核架构。这促使我们使用一些绝对最小批量，低于这个值的小批量处理不会减少计算时间；
- 如果批量处理中的所有样本都可以并行地处理，那么内存消耗和批量大小会成正比。对于很多硬件设施，这是批量大小的限制因素；
- 在GPU等某些硬件上使用特定大小的批次大小可以更快运行，通常使用2的指数幂作为批量大小，通常大于16小于512；
- 可能是由于小批量在学习过程中加入了噪声，它们会有一些正则化效果(Wilson and Martinez, 2003)。泛化误差通常在批量大小为1时最好，此时因为梯度估计的高方差，小批量学习的学习率应该设置小一些以保持稳定性。
- 不同的算法使用不同的方法从小批量中获取不同的信息。

小批量抽取数据的注意事项

- 小批量数据是随机抽取的，要尽量避免连续样本之间具有高度的相关性，通常可以在抽取小批量样本时对样本顺序进行一次打乱。
- 当没有重复使用样本时，小批量数据对梯度的估计是无偏的，即它遵循着真实泛化误差的梯度。

2. 基本优化算法

2.1 小批量梯度下降

小批量梯度下降

小批量梯度下降 (Min Batch Gradient Descent, MBGD) , 有时也被不严谨的称为随机梯度下降, 是机器学习中应用最多的优化算法。其利用小批量样本对梯度进行估计, 并使用梯度的反方向更新模型参数。

算法描述

算法 小批量梯度下降算法

Require: 学习率 ϵ

Require: 初始参数 θ

while 停止准则未满足 **do**

从训练集中采包含 m 个样本 $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ 的小批量, 其中 $\mathbf{x}^{(i)}$ 对应目标为 $\mathbf{y}^{(i)}$ 。

计算梯度估计: $\hat{\mathbf{g}} \leftarrow +\frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

应用更新: $\theta \leftarrow \theta - \epsilon \hat{\mathbf{g}}$

end while

TensorFlow实现GD算法

方法一：

1. 使用 `tf.train.GradientDescentOptimizer()` 创建优化器对象。
2. 使用优化器对象的方法 `minimize()` 应用求解梯度并参数更新的过程。

通常用法：

```
opt = tf.train.GradientDescentOptimizer(learning_rate)
train_op = minimize(loss)
```

TensorFlow实现GD算法

方法二：

1. 使用 `tf.train.GradientDescentOptimizer()` 创建优化器对象。

2. 使用优化器对象的梯度计算方法求得梯度 `compute_gradients()`

3. 使用优化器对象的应用梯度方法更新参数 `apply_gradients()`

2、3步骤等价于使用 `minimize()`

通常用法：

```
learning_rate = 0.01

cus_opt = tf.train.GradientDescentOptimizer(learning_rate)
grads_and_vars = cus_opt.compute_gradients(loss, [w])

train_op = cus_opt.apply_gradients(grads_and_vars)
```

TensorFlow实现GD算法

方法三：

1. 使用优化器基类创建优化器对象 `tf.train.Optimizer()`
2. 使用优化器对象的梯度计算方法求得梯度 `compute_gradients()`
3. 利用参数更新规则和求得的梯度更新参数

通常用法：

```
learning_rate = 0.01

cus_opt = tf.train.Optimizer(use_locking=False, name='CustomOptimizer')
grads_and_vars = cus_opt.compute_gradients(loss, [w])

train_op = [gv[1].assign_sub(gv[0] * learning_rate) for gv in grads_and_vars]
```


实验一

- 设计线性回归模型，使用方法二实现GD算法，并完成训练；
- （作业）设计线性回归模型，使用方法三实现GD算法，并完成训练；

2.1.1 学习率衰减

学习率衰减

学习率衰减 (Learning rate decay) 是在执行小批量梯度下降算法或随机梯度下降法时，设置动态的学习率，使其随着算法的迭代减小学习率，收敛于最小值附近。

目的：解决在SGD或MBGD中梯度估计引入噪声使得在极小值点附近无法收敛的问题。批量梯度下降不存在此问题，无需使用学习率衰减。

学习率衰减方法

- 线性衰减学习率
- 指数衰减学习率
- 其它

线性衰减

线性衰减学习率即每次迭代使得学习率线性减少。实践中，一般会使用线性衰减学习率到 τ 次迭代，之后使用固定的学习率即可。

前 τ 次迭代时，学习率衰减方法为：

$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha\epsilon_\tau$$

初始学习率

1%

当迭代次数大于 τ 时，学习率固定为： ϵ_τ 稳定学习率

其中 $\alpha = \frac{k}{\tau}$ 训练次数

线性衰减实例

```
start_lr = 0.3 # 初始学习率
stable_lr = start_lr * 0.01 # 稳定后的学习率
decay_step = 20 # 线性衰减步数
train_step = 50 # 训练步数

# 训练模型
for i in range(train_step):
    if i < decay_step:
        lr = (1 - i / decay_step) * start_lr + i / decay_step * stable_lr
    else:
        lr = stable_lr

    print(lr)
    # training ...
    pass
```

线性衰减学习率的优缺点

- 需要确定三个超参数，需要根据经验尝试。例如初始学习率太大模型可能无法收敛，太小可能导致收敛速度太慢；
- 合理的超参数可以使模型更好更快的收敛。

指数衰减

指数衰减学习率的式子如下：

$$\varepsilon_k = \varepsilon_0 d^k$$

初始学习率

衰减率

训练次数

为了避免学习率衰减过快，通常，每训练n步执行一次指数衰减，同时衰减率值域一般为(0, 1)，且衰减率接近于1。

指数衰减学习率的优缺点

- 需要确定初始学习率、衰减率、衰减间隔3个超参数，若超参数设置不合理可能导致模型训练缓慢甚至无法收敛；
- 需要警惕使用衰减间隔，过于频繁的衰减学习率可能会使得学习率接近于0，导致模型更新接近停止。
- 衰减率通常设置与0.96-0.99的数值，这时候通常学习率衰减100次左右即可。

指数衰减实例

```
start_lr = 0.3    # 初始学习率
decay_rate=0.96   # 衰减率
decay_times = 5   # 指数衰减间隔
train_step = 50   # 训练步数

lr = start_lr
# 训练模型
for i in range(train_step):
    if i % decay_times == decay_times - 1:
        lr = stable_lr * decay_rate ** ((i + 1) / decay_times)

    print(lr)
    # training ...
    pass
```

实验二

- 根据上述“实例”实现线性衰减学习率；
- 根据上述“实例”实现指数衰减学习率；
- （**作业**）使用TensorFlow内置的指数衰减方法改进实验一中的学习率。

主要使用方法： `tf.train.exponential_decay()`

2.2 动量梯度下降

带动量的梯度下降法

动量 (Momentum) 方法引入速度变量 \boldsymbol{v} 代表参数在参数空间的移动方向和速率。当前速度受到之前速度的影响，即当前位置梯度与之前速度的方向一致时，当前速度增大，反之当前速度减小。

参数更新规则

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^m L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right),$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}.$$

动量小批量梯度下降法流程

算法 带动量的小批量梯度下降法

Require: 学习率 ϵ , 动量参数 α , 其中 $\alpha \in [0, 1)$

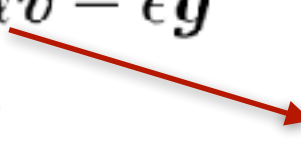
Require: 初始参数 θ , 初始速度 v

while 没有达到停止准则 **do**

从训练集中采包含 m 个样本 $\{x^{(1)}, \dots, x^{(m)}\}$ 的小批量, 对应目标为 $y^{(i)}$ 。

计算梯度估计: $g \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(x^{(i)}; \theta), y^{(i)})$

计算速度更新: $v \leftarrow \alpha v - \epsilon g$

应用更新: $\theta \leftarrow \theta + v$  **指数衰减速度**

end while

超参数 α 的最大贡献

例如：

一个参数对应的速度为0，每次更新梯度时的值均为-1， $\alpha=0.1$ ，则每次更新梯度对应的速度为：

$$v_0 = 0$$

$$v_1 = 0 + 1 = 1$$

$$v_2 = 0.1 * v_1 + 1 = 1.1$$

$$v_3 = 0.1 * v_2 + 1 = 1.11$$

.....

最大速度成为原来的 $\frac{1}{1-\alpha}$ 倍。

```
class MomentumOpt(tf.train.Optimizer):

    def __init__(self, learning_rate=0.001, momentum=0.9,
                  use_locking=False, name='CusMom'):
        super(MomentumOpt, self).__init__(use_locking, name)
        self._lr = learning_rate
        self._momentum = momentum

        self._lr_t = None
        self._mom_t = None

    def _prepare(self):
        self._lr_t = tf.convert_to_tensor(self._lr)
        self._mom_t = tf.convert_to_tensor(self._momentum)

    def _create_slots(self, var_list):
        for v in var_list:
            self._zeros_slot(v, 'v', self._name)

    def _apply_dense(self, grad, var):
        self._lr_t = tf.cast(self._lr_t, var.dtype.base_dtype)
        self._mom_t = tf.cast(self._mom_t, var.dtype.base_dtype)

        v = self.get_slot(var, 'v')
        v_t = v.assign(v * self._mom_t - grad * self._lr_t)
        var_update = var.assign_add(v_t)
        return var_update
```


实验对比

- 分别使用相同的学习率与初始化参数，使用小批量梯度下降法与动量方法训练线性回归模型，对比前10次的代价下降情况。**一般的，在开始时小批量梯度下降更快。**
- 分别使用相同的学习率与初始化参数，使用小批量梯度下降法与动量方法训练线性回归模型直到收敛，对比两个算法的收敛速度（收敛所用步数）。**一般的，动量方法收敛更快。**

实验对比

```
train_op = tf.train.GradientDescentOptimizer(0.003).minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(5000):
        loss_res, _ = sess.run([loss, train_op])
        if i < 10:
            print('step %d: %f' % (i, loss_res))
        if loss_res < 1e-6:
            print('number of steps: %d' % i)
            break
```

```
step 0: 622.923340
step 1: 3.509555
step 2: 0.163335
step 3: 0.143392
step 4: 0.141430
step 5: 0.139592
step 6: 0.137777
step 7: 0.135986
step 8: 0.134218
step 9: 0.132473
number of steps: 911
```

实验模型：二元线性回归
优化方法：MBGD
前期下降速度：较快
收敛步数：911步

实验对比

```
train_op = tf.train.MomentumOptimizer(0.003, 0.9).minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(5000):
        loss_res, _ = sess.run([loss, train_op])
        if i < 10:
            print('step %d: %f' % (i, loss_res))
        if loss_res < 1e-6:
            print('number of steps: %d' % i)
            break
```

```
step 0: 622.923340
step 1: 3.509555
step 2: 574.967834
step 3: 330.138885
step 4: 43.210865
step 5: 474.238281
step 6: 146.277573
step 7: 92.352264
step 8: 354.251617
step 9: 47.843769
number of steps: 144
```

实验模型：二元线性回归
优化方法：动量方法
前期下降速度：较慢
收敛步数：144步

动量方法的特点

- 通常的，与GD相比，加快了训练速度；
- 初始训练速度较慢；
- 对处理高曲率、小但一致的梯度或是带噪声的梯度，加速效果更好；
- 超参数 α 决定了之前梯度的贡献衰减的有多快，当总观察到梯度为一个常数，则其最终速度会成为原来的 $\frac{1}{1-\alpha}$ ；实践中 α 一般取值为0.5、0.9或0.99。
- 动量方法会在极值点附近抖动。

Nesterov方法

算法 带Nesterov动量的小批量梯度下降法

Require: 学习率 ϵ , 动量参数 α , 其中 $\alpha \in [0, 1)$

Require: 初始参数 θ , 初始速度 v

while 没有达到停止准则 do

从训练集中采包含 m 个样本 $\{x^{(1)}, \dots, x^{(m)}\}$ 的小批量, 对应目标为 $y^{(i)}$ 。

应用临时更新: $\tilde{\theta} \leftarrow \theta + \alpha v$

计算梯度 (在临时点): $g \leftarrow \frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L(f(x^{(i)}; \tilde{\theta}), y^{(i)})$

计算速度更新: $v \leftarrow \alpha v - \epsilon g$

应用更新: $\theta \leftarrow \theta + v$

end while

通过估计下一个更新位置的梯度, 来修正当前更新速度, 避免梯度大幅震荡。

实验

```
train_op = tf.train.MomentumOptimizer(0.003, 0.9, use_nesterov=True).minimize(loss)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(5000):
        loss_res, _ = sess.run([loss, train_op])
        if i < 10:
            print('step %d: %f' % (i, loss_res))
        if loss_res < 1e-6:
            print('number of steps: %d' % i)
            break
```

```
step 0: 622.923340
step 1: 673.233215
step 2: 27.938208
step 3: 6.145210
step 4: 0.604189
step 5: 0.185756
step 6: 0.117241
step 7: 0.102781
step 8: 0.093400
step 9: 0.084743
number of steps: 36
```

实验模型：二元线性回归
优化方法：Nesterov方法
前期下降速度：较快
收敛步数：36步

实验三

- 使用TensorFlow实现带动量的梯度下降法，并使用其优化线性回归模型；
- 与使用GD算法进行对比，观察哪个算法优化速度快？
- 改进上述实现为Nesterov方法。
- 学习TensorFlow内置的动量优化法的用法。

3. 自适应学习率算法

背景

- 学习率是神经网络中难以设置的超参数之一，不同的学习率会影响模型的收敛速度和最终性能；
- 不同的参数的梯度对损失的敏感度不同，但学习率是相同的；
- 动量算法在一定程度上解决了敏感度不同的问题，但其引入了新的超参数。

3.1 AdaGrad

AdaGrad

AdaGrad (Adaptive Gradient) 方法对每个变量使用不同的学习率，这个学习率在一开始时比较大，用于快速下降，随着优化的进行，使已经“下降”较多的变量的学习率更多的减小，使下降较少的变量的学习率更少的减小。

AdaGrad算法流程

算法 AdaGrad

Require: 全局学习率 ϵ

Require: 初始参数 θ

Require: 小常数 δ , 为了数值稳定大约设为 10^{-7}

初始化梯度累积变量 $\mathbf{r} = 0$

while 没有达到停止准则 **do**

从训练集中采包含 m 个样本 $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ 的小批量, 对应目标为 $\mathbf{y}^{(i)}$ 。

计算梯度: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

累积平方梯度: $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{g} \odot \mathbf{g}$

计算更新: $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \mathbf{g}$ (逐元素地应用除和求平方根)

应用更新: $\theta \leftarrow \theta + \Delta\theta$

end while

AdaGrad算法的特点

- 为每个参数制定了一个动态的学习率；
- 应用于凸问题时，模型可以快速收敛；
- 经验上发现，在非凸的神经网络中，从训练开始积累梯度平方会导致有效学习率过早和过量减小，导致训练速度变慢。

算法实现

```
class AdaGrad(tf.train.Optimizer):
    def __init__(self, learning_rate=0.001, epsilon=1e-7, use_locking=False, name='CusAdaGrad'):
        super(AdaGrad, self).__init__(use_locking, name)
        self._lr = learning_rate
        self._ep = epsilon

        self._lr_t = None
        self._ep_t = None

    def _prepare(self):
        self._lr_t = tf.convert_to_tensor(self._lr)
        self._ep_t = tf.convert_to_tensor(self._ep)

    def _create_slots(self, var_list):
        for r in var_list:
            self._zeros_slot(r, 'r', self._name)

    def _apply_dense(self, grad, var):
        r = self.get_slot(var, 'r')
        print(self.get_slot_names())
        update_r = r.assign_add(grad * grad)
        delta_theta = - (self._lr_t * grad) / (tf.sqrt(update_r) + self._ep_t)
        var_update = var.assign_add(delta_theta)
        return var_update
```

实验四

- 使用TensorFlow实现AdaGrad算法，并使用此算法完成线性回归模型训练。
- 与使用GD算法进行对比，观察哪个算法优化速度快？
- （**作业**）学习使用TensorFlow内置的AdaGrad优化方法的使用法。

3.2 RMSProp

RMSProp

RMSProp算法修改AdaGrad以在非凸设定下效果更好，其改变梯度平方累积为指数加权平均，以丢弃遥远过去的历史，使其能够快速收敛。

RMSProp算法流程

算法 RMSProp

Require: 全局学习率 ϵ , 衰减速率 ρ

Require: 初始参数 θ

Require: 小常数 δ , 通常设为 10^{-6} (用于被小数除时的数值稳定)

初始化累积变量 $r = 0$

while 没有达到停止准则 **do**

从训练集中采包含 m 个样本 $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ 的小批量, 对应目标为 $\mathbf{y}^{(i)}$ 。

计算梯度: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

累积平方梯度: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho) \mathbf{g} \odot \mathbf{g}$

计算参数更新: $\Delta \theta = -\frac{\epsilon}{\sqrt{\delta + \mathbf{r}}} \odot \mathbf{g}$ ($\frac{1}{\sqrt{\delta + \mathbf{r}}}$ 逐元素应用)

应用更新: $\theta \leftarrow \theta + \Delta \theta$

end while

RMSProp算法特点

- 为每个参数制定了一个动态的学习率；
- 经验上，RMSProp已被证明是一种有效且实用的神经网络优化算法；
- 其引入了衰减率超参数，这个参数默认使用0.9即可。
- 初始训练阶段对梯度的二阶矩估计的偏差很大。

RMSProp算法实现

```
class RMSProp(tf.train.Optimizer):
    def __init__(self, learning_rate=0.001, decay=0.9, epsilon=1e-6,
                 use_locking=False, name='CusRMSProp'):
        super(RMSProp, self).__init__(use_locking, name)
        self._lr = learning_rate
        self._decay = decay
        self._ep = epsilon

        self._lr_t = None
        self._decay_t = None
        self._ep_t = None

    def _prepare(self):
        self._lr_t = tf.convert_to_tensor(self._lr)
        self._decay_t = tf.convert_to_tensor(self._decay)
        self._ep_t = tf.convert_to_tensor(self._ep)

    def _create_slots(self, var_list):
        for r in var_list:
            self._zeros_slot(r, 'r', self._name)

    def _apply_dense(self, grad, var):
        r = self.get_slot(var, 'r')
        update_r = r.assign(r * self._decay_t + (1. - self._decay_t) * grad * grad)
        delta_theta = - (self._lr_t * grad) / (tf.sqrt(update_r) + self._ep_t)
        update_var = var.assign_add(delta_theta)
        return update_var
```

实验五

- 使用TensorFlow实现RMSProp算法，并使用此算法完成线性回归模型训练。
- （作业）学习使用TensorFlow内置的AdaGrad优化方法的使用法。

了解

带Nesterov动量的RMSProp算法

算法 使用Nesterov动量的RMSProp算法

Require: 全局学习率 ϵ , 衰减速率 ρ , 动量系数 α

Require: 初始参数 θ , 初始参数 v

初始化累积变量 $r = 0$

while 没有达到停止准则 **do**

从训练集中采包含 m 个样本 $\{x^{(1)}, \dots, x^{(m)}\}$ 的小批量, 对应目标为 $y^{(i)}$ 。

计算临时更新: $\tilde{\theta} \leftarrow \theta + \alpha v$

计算梯度: $g \leftarrow \frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L(f(x^{(i)}; \tilde{\theta}), y^{(i)})$

累积梯度: $r \leftarrow \rho r + (1 - \rho) g \odot g$

计算速度更新: $v \leftarrow \alpha v - \frac{\epsilon}{\sqrt{r}} \odot g$ ($\frac{1}{\sqrt{r}}$ 逐元素应用)

应用更新: $\theta \leftarrow \theta + v$

end while

思考

- 思考动量与学习率自适应分别为梯度下降带来了哪些益处？

3.3 Adam

Adam

自适应矩估计 (Adaptive Moment Estimation, Adam) 在 RMSProp 的基础上增加梯度一阶矩估计（相当于加入动量部分）和偏差修正，解决了在训练初期矩估计偏差较大的问题。

算法 Adam算法流程

Require: 步长 ϵ (建议默认为: 0.001)

Require: 矩估计的指数衰减速率, ρ_1 和 ρ_2 在区间 $[0, 1)$ 内。(建议默认为: 分别为 0.9 和 0.999)

Require: 用于数值稳定的小常数 δ (建议默认为: 10^{-8})

Require: 初始参数 θ

初始化一阶和二阶矩变量 $\mathbf{s} = 0, \mathbf{r} = 0$


初始化时间步 $t = 0$

while 没有达到停止准则 **do**

从训练集中采包含 m 个样本 $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ 的小批量, 对应目标为 $\mathbf{y}^{(i)}$ 。

计算梯度: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

$t \leftarrow t + 1$

更新有偏一阶矩估计: $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1) \mathbf{g}$  类似于动量

更新有偏二阶矩估计: $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2) \mathbf{g} \odot \mathbf{g}$

修正一阶矩的偏差: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \rho_1^t}$

修正二阶矩的偏差: $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1 - \rho_2^t}$

 偏差修正使得在初始阶段受一阶矩与二阶矩初始值的影响较小

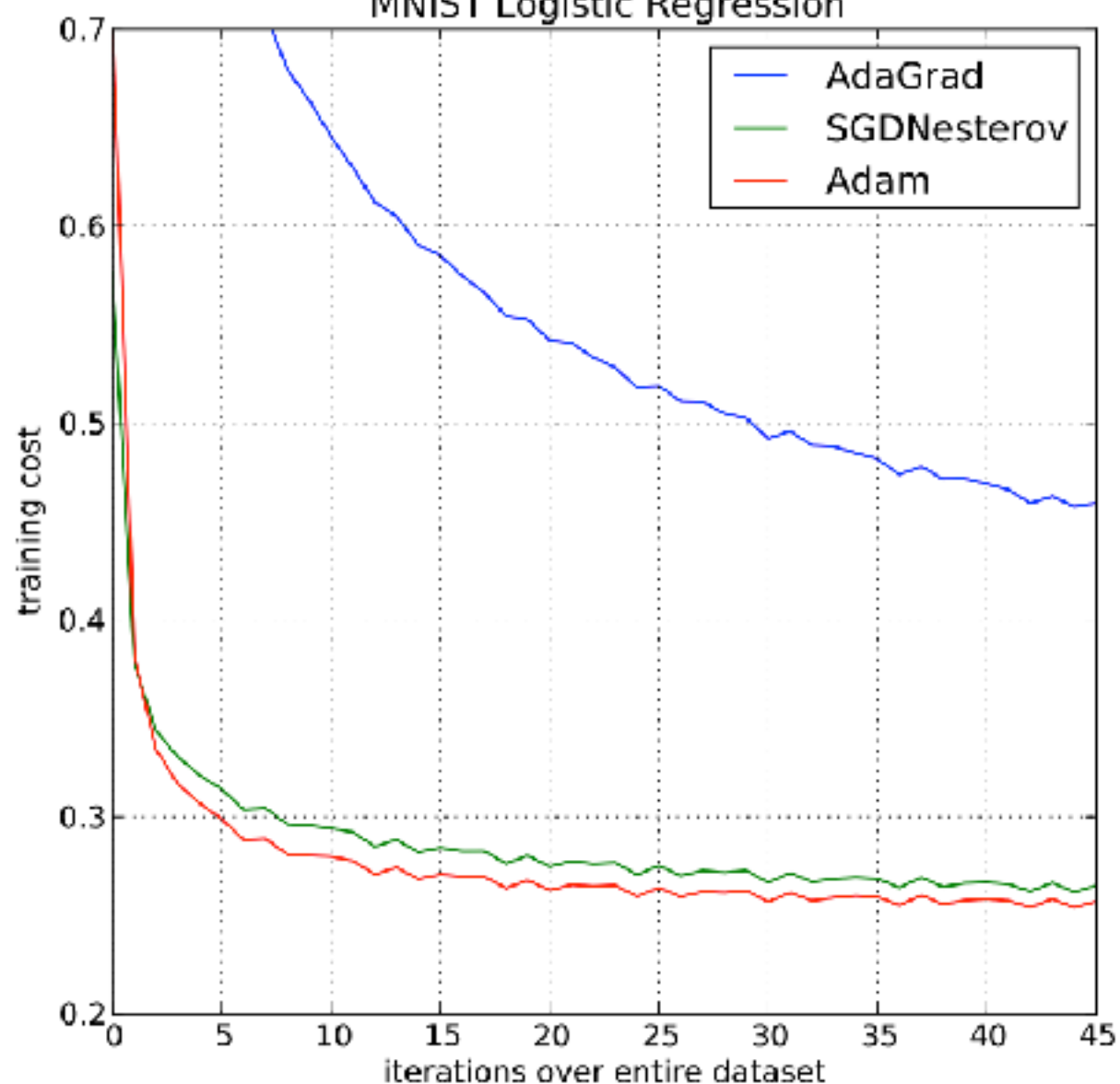
计算更新: $\Delta \theta = -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}} + \delta}}$ (逐元素应用操作)

应用更新: $\theta \leftarrow \theta + \Delta \theta$

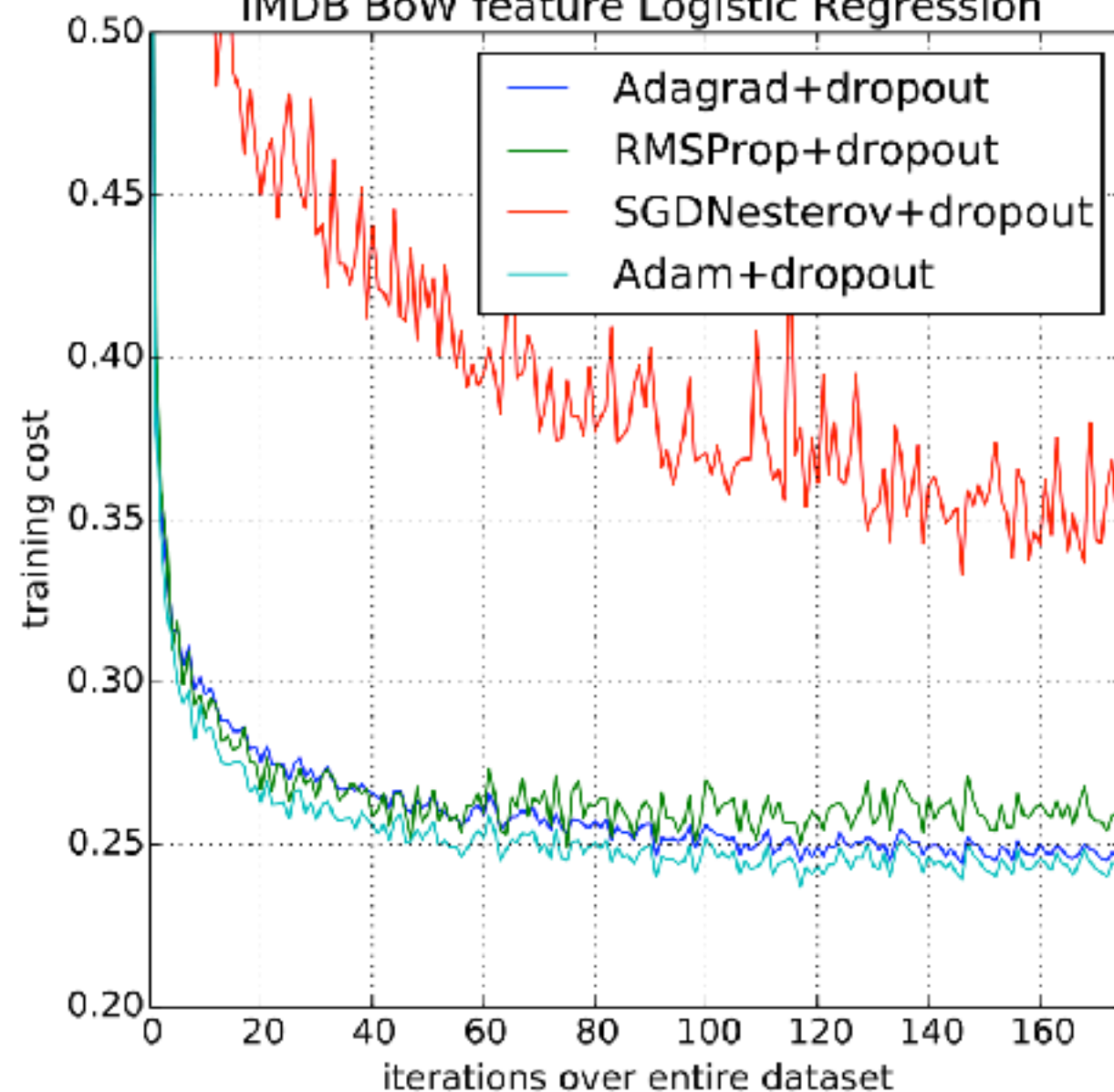
end while

Adam效果

MNIST Logistic Regression



IMDB BoW feature Logistic Regression



```
class Adam(tf.train.Optimizer):
    def __init__(self, learning_rate=0.001, beta1=0.9, beta2=0.999,
                  epsilon=1e-08, use_locking=False, name='Adam'):
        super(Adam, self).__init__(use_locking, name)
        self._lr = learning_rate
        self._b1 = beta1
        self._b2 = beta2
        self._ep = epsilon

        self._lr_t = None
        self._b1_t = None
        self._b2_t = None
        self._ep_t = None
        self._t = None

    def _prepare(self):
        self._lr_t = tf.convert_to_tensor(self._lr)
        self._b1_t = tf.convert_to_tensor(self._b1)
        self._b2_t = tf.convert_to_tensor(self._b2)
        self._ep_t = tf.convert_to_tensor(self._ep)
        self._t = tf.Variable(0.)

    def _create_slots(self, var_list):
        for var in var_list:
            self._zeros_slot(var, 's', self._name)
            self._zeros_slot(var, 'r', self._name)

    def _apply_dense(self, grad, var):
        update_t = self._t.assign_add(1)
        s = self.get_slot(var, 's')
        update_s = s.assign(s * self._b1_t + grad * (1 - self._b1_t))
        r = self.get_slot(var, 'r')
        update_r = r.assign(r * self._b2_t + grad * grad * (1 - self._b2_t))
        fix_s = update_s / (1 - tf.pow(self._b1_t, update_t))
        fix_r = update_r / (1 - tf.pow(self._b2_t, update_t))
        delta_theta = - (self._lr_t * fix_s) / (self._ep_t + tf.sqrt(fix_r))
        update_var = var.assign_add(delta_theta)
        return update_var
```

实验六

- 使用TensorFlow实现Adam算法，并使用此算法完成线性回归模型训练。
- （作业）学习使用TensorFlow内置的Adam优化方法的使用法。

Adam算法特点

- 为每个参数制定了一个动态的学习率；
- Adam算法的实际效果很好，是最常用的优化方法；
- 通常的超参数保持默认值即可，即超参数调整简单。

3.4 算法选择

算法选择

通常，并没有一个准确的结论说明哪个优化算法比其他优化算法更好。经验数据表明具有自适应学习率的算法表现的相当鲁棒。目前最常用的方法主要是带有Nesterov动量的SGD、RMSProp、Adam等。

其他优化方法

- 二阶优化法：牛顿法、拟牛顿法等；
- 坐标下降法；
- 其它。

小结

- 学习与优化的关系；
- 小批量算法流程与实现；
- 学习率衰减有助于小批量算法收敛；
- 使用动量方法改进SGD加快了训练速度；
- 使用自适应学习率算法可以高效训练模型。