

1. 深度神经网络

1.1 深度神经网络框架设计

1.2 二分类任务真实案例

1.2.1 准备数据

数据字段解读

数据处理

加载数据

1.2.2 搭建模型

1.2.3 训练结果

1.3 多分类任务 - MNIST手写体识别

1.3.1 数据读取

1.3.2 搭建模型

1.3.3 运行结果

1.4 网络优化

1.4.1 权重矩阵初始化

1.4.1.1 零初始化

1.4.1.2 标准初始化

1.4.1.3 Xavier初始化方法

1.4.1.4 MSRA初始化方法

1.4.1.5 小结

1.4.2 批量归一化

1.4.2.1 原理

1.4.2.2 批量归一化的优点

1.4.3 梯度下降优化算法

1.4.3.1 随机梯度下降 SGD

1.4.3.2 动量算法 Momentum

1.4.3.3 梯度加速算法 NAG

1.4.4 自适应学习率算法

1.4.4.1 AdaGrad

1.4.4.2 AdaDelta

1.4.4.3 均方根反向传播 RMSProp

1.4.4.4 Adam - Adaptive Moment Estimation

1.4.5 算法效果比较

1.5 正则化

1.5.1 过拟合

1.5.1.1 过拟合的原因

1.5.1.2 解决过拟合

1.5.2 偏差和方差

1.5.3 解决过拟合之早停法

1.5.4 解决过拟合之丢弃法

1.5.5 解决过拟合之数据拓展

1.5.5.1 图像数据增强主要包括：

1.5.5.2 多样本合成法

1.5.6 结果过拟合之集成学习

1. 深度神经网络

1.1 深度神经网络框架设计

1.2 二分类任务真实案例

用一个真实的数据级来实现一个二分类任务：收入调查与预测，即给定一个居民的各种情况，如工作、家庭、学历等，来预测该居民的年收入是否可以大于50K/年，所以大于50K的就是正例，而小于等于50K的就是负例。

1.2.1 准备数据

此数据集是从1994 Census数据库中提取的。

数据字段解读

标签值：>50K, <=50K。

属性字段：

- `age`，年龄：连续值
- `workclass`，工作性质：枚举型，类似私企、政府之类的
- `fnlwgt`，权重：连续值
- `education`，教育程度：枚举型，如学士、硕士等
- `education-num`，受教育的时长：连续值
- `marital-status`，婚姻状况：枚举型，已婚、未婚、离异等
- `occupation`，职业：枚举型，包含的种类很多，如技术支持、维修工、销售、农民渔民、军人等
- `relationship`，家庭角色：枚举型，丈夫、妻子等
- `sex`，性别：枚举型
- `capital-gain`，资本收益：连续值
- `capital-loss`，资本损失：连续值
- `hours-per-week`，每周工作时长：连续值
- `native-country`，祖籍：枚举型

数据处理

对于连续值，可以直接使用原始数据。对于枚举型，需要把它们转成连续值。以性别举例，`Female=0`，`Male=1` 即可。对于其它枚举型，都可以用从0开始的整数编码。

```
1 sex_list = ["Female", "Male"]
2 array_x[0,9] = sex_list.index(row[9].strip())
```

`strip()` 是trim掉前面的空格，因为是 `csv` 格式，读出来会是这个样子："`_Female`"，前面总有个空格。`index` 是取列表下标，这样对于字符串"`Female`"取出的下标为0，对于字符串"`Male`"取出的下标为1。

把所有数据按行保存到 `numpy` 数组中，最后用 `npz` 格式存储：

```
1 np.savez(data_npz, data=self.XData, label=self.YData)
```

原始数据已经把train data和test data分开了，所以针对两个数据集分别调用数据处理过程一次，保存为 `Income_Train.npz` 和 `Income_Test.npz`。

加载数据

```

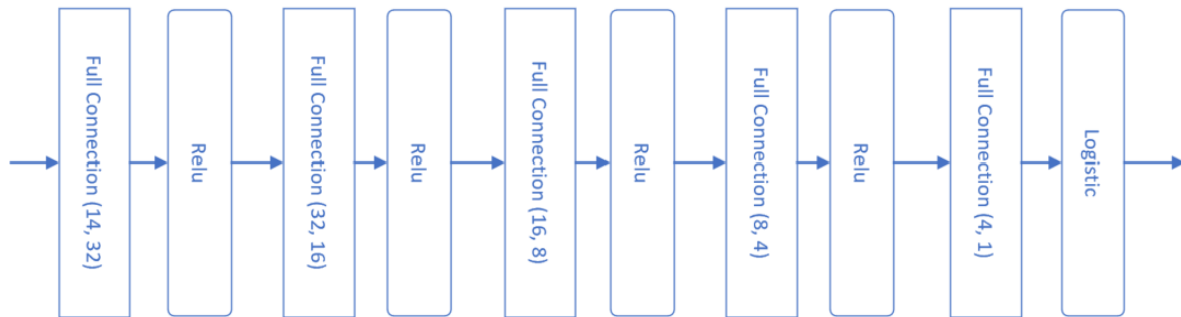
1 train_file = "../../Data/ch14.Income.train.npz"
2 test_file = "../../Data/ch14.Income.test.npz"
3
4 def LoadData():
5     dr = DataReader_2_0(train_file, test_file)
6     dr.ReadData()
7     dr.NormalizeX()
8     dr.Shuffle()
9     dr.GenerateValidationSet()
10    return dr

```

因为属性字段众多，取值范围相差很大，所以一定要先调用 `NormalizeX()` 函数做归一化。由于是二分类问题，在做数据处理时，我们已经把大于50K标记为1，小于等于50K标记为0，所以不需要做标签值的归一化。

1.2.2 搭建模型

搭建一个如下的网络结构



```

1 def model(dr):
2     num_input = dr.num_feature
3     num_hidden1 = 32
4     num_hidden2 = 16
5     num_hidden3 = 8
6     num_hidden4 = 4
7     num_output = 1
8
9     max_epoch = 100
10    batch_size = 16
11    learning_rate = 0.1
12
13    params = HyperParameters_4_0(
14        learning_rate, max_epoch, batch_size,
15        net_type=NetType.BinaryClassifier,
16        init_method=InitialMethod.MSRA,
17        stopper=Stopper(StopCondition.StopDiff, 1e-3))
18
19    net = NeuralNet_4_0(params, "Income")
20
21    fc1 = FcLayer_1_0(num_input, num_hidden1, params)
22    net.add_layer(fc1, "fc1")
23    a1 = ActivationLayer(Relu())
24    net.add_layer(a1, "relu1")
25    .....
26    fc5 = FcLayer_1_0(num_hidden4, num_output, params)
27    net.add_layer(fc5, "fc5")
28    logistic = ClassificationLayer(Logistic())

```

```

29     net.add_layer(logistic, "logistic")
30
31     net.train(dr, checkpoint=1, need_test=True)
32     return net

```

超参数说明：

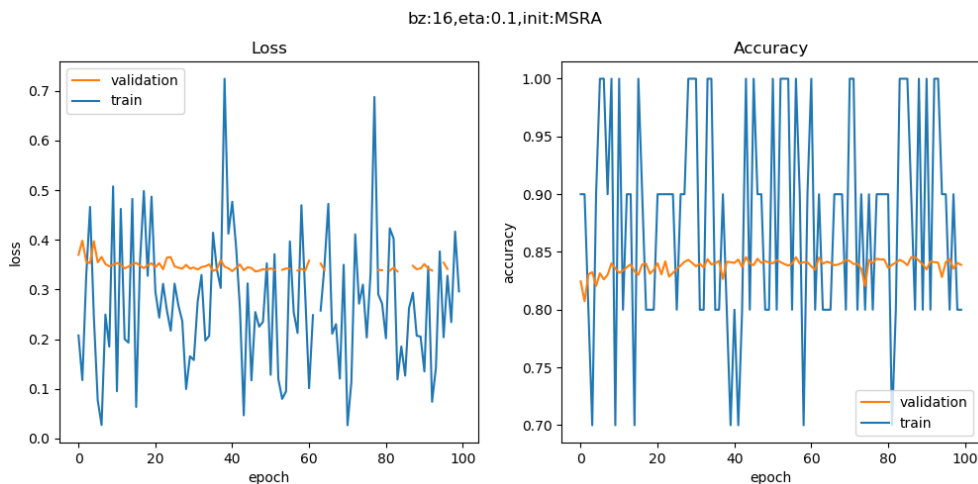
1. 学习率=0.1
2. 最大 epoch=100
3. 批大小=16
4. 二分类网络类型
5. MSRA初始化
6. 相对误差停止条件 $1e-3$

`net.train()` 函数是一个阻塞函数，只有当训练完毕后才返回。

1.2.3 训练结果

下图左边是损失函数图，右边是准确率图。忽略测试数据的波动，只看黄色的验证集的趋势，损失函数值不断下降，准确率不断上升。

为什么不把 `max_epoch` 设置为更大的数字，比如1000，以便得到更好的结果呢？实际上，训练更多的次数，因为过拟合的风险，不会得到更好的结果。



下面是最后的打印输出：

```

1  .....
2  epoch=99, total_iteration=169699
3  loss_train=0.296219, accuracy_train=0.800000
4  loss_valid=nan, accuracy_valid=0.838859
5  time used: 29.866002321243286
6  testing...
7  0.8431606905710491

```

最后用独立的测试集得到的结果是84%，与该数据集相关的其它论文相比，已经是一个不错的成绩了。

1.3 多分类任务 - MNIST手写体识别

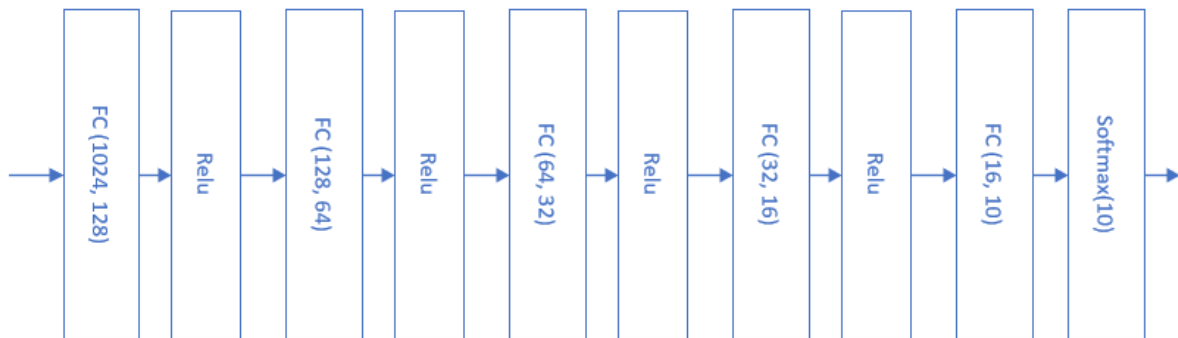
1.3.1 数据读取

MNIST数据本身是图像格式的，用 `mode="vector"` 去读取，转变成矢量格式。

```
1 def LoadData():
2     print("reading data...")
3     dr = MnistImageDataReader(mode="vector")
4     .....
```

1.3.2 搭建模型

一共4个隐层，都用ReLU激活函数连接，最后的输出层接Softmax分类函数。



以下是主要的参数设置：

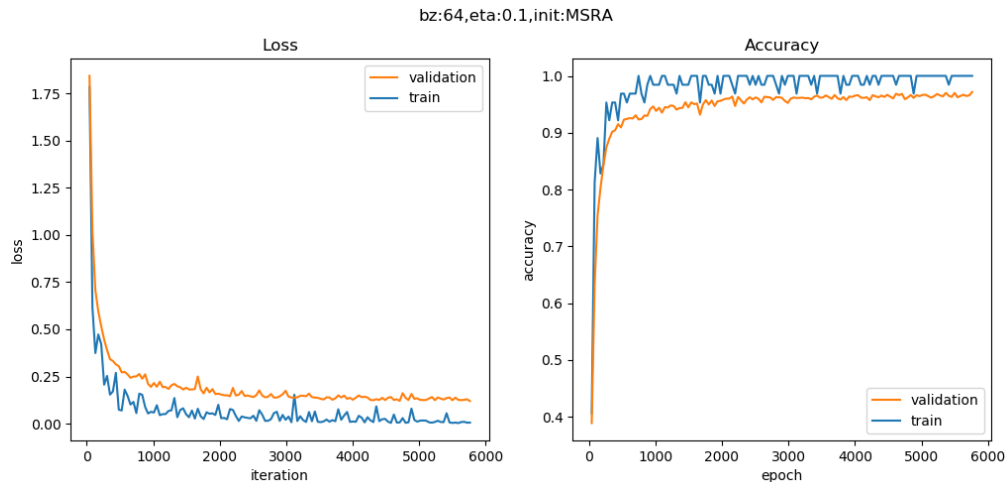
```
1 if __name__ == '__main__':
2     dataReader = LoadData()
3     num_feature = dataReader.num_feature
4     num_example = dataReader.num_example
5     num_input = num_feature
6     num_hidden1 = 128
7     num_hidden2 = 64
8     num_hidden3 = 32
9     num_hidden4 = 16
10    num_output = 10
11    max_epoch = 10
12    batch_size = 64
13    learning_rate = 0.1
14
15    params = HyperParameters_4_0(
16        learning_rate, max_epoch, batch_size,
17        net_type=NetType.MultipleClassifier,
18        init_method=InitialMethod.MSRA,
19        stopper=Stopper(StopCondition.StopLoss, 0.12))
20
21    net = NeuralNet_4_0(params, "MNIST")
22
23    fc1 = FcLayer_1_0(num_input, num_hidden1, params)
24    net.add_layer(fc1, "fc1")
25    r1 = ActivationLayer(Relu())
26    net.add_layer(r1, "r1")
27    .....
28    fc5 = FcLayer_1_0(num_hidden4, num_output, params)
29    net.add_layer(fc5, "fc5")
30    softmax = ClassificationLayer(Softmax())
31    net.add_layer(softmax, "softmax")
```

32
33
34

```
net.train(dataReader, checkpoint=0.05, need_test=True)  
net.ShowLossHistory(xcoord=XCoordinate.Iteration)
```

1.3.3 运行结果

停止条件是绝对Loss值达到0.12时，所以迭代到6个epoch时，达到了0.119的损失值，就停止训练了。



下面是最后几行的打印输出。

```
1 .....  
2 epoch=6, total_iteration=5763  
3 loss_train=0.005559, accuracy_train=1.000000  
4 loss_valid=0.119701, accuracy_valid=0.971667  
5 time used: 17.500738859176636  
6 save parameters  
7 testing...  
8 0.9697
```

最后用测试集得到的准确率为96.97%。

1.4 网络优化

随着网络的加深，训练变得越来越困难，时间越来越长，原因可能是：

- 参数多
- 数据量大
- 梯度消失
- 损失函数坡度平缓

为了解决上面这些问题，科学家们在深入研究网络表现的前提下，发现在下面这些方向上经过一些努力，可以给深度网络的训练带来或多或少的改善：

- 权重矩阵初始化
- 批量归一化
- 梯度下降优化算法
- 自适应学习率算法

1.4.1 权重矩阵初始化

1.4.1.1 零初始化

即把所有层的 w 值的初始值都设置为0。

$$W = 0$$

但是对于多层网络来说，绝对不能用零初始化，否则权重值不能学习到合理的结果。看下面的零值初始化的权重矩阵值打印输出：

```
1 w1= [[-0.82452497 -0.82452497 -0.82452497]]
2 b1= [[-0.01143752 -0.01143752 -0.01143752]]
3 w2= [[-0.68583865]
4       [-0.68583865]
5       [-0.68583865]]
6 b2= [[0.68359678]]
```

可以看到 $w1$ 、 $b1$ 、 $w2$ 内部3个单元的值都一样，这是因为初始值都是0，所以梯度均匀回传，导致所有 w 的值都同步更新，没有差别。这样的话，无论多少轮，最终的结果也不会正确。

1.4.1.2 标准初始化

标准正态初始化方法保证激活函数的输入均值为0，方差为1。将 W 按如下公式进行初始化：

$$W \sim N[0, 1]$$

其中的 W 为权重矩阵， N 表示高斯分布，Gaussian Distribution，也叫做正态分布，Normal Distribution，所以有的地方也称这种初始化为Normal初始化。

一般会根据全连接层的输入和输出数量来决定初始化的细节：

$$W \sim N\left(0, \frac{1}{\sqrt{n_{in}}}\right)$$

$$W \sim U\left(-\frac{1}{\sqrt{n_{in}}}, \frac{1}{\sqrt{n_{in}}}\right)$$

当目标问题较为简单时，网络深度不大，所以用标准初始化就可以了。

1.4.1.3 Xavier初始化方法

条件：正向传播时，激活值的方差保持不变；反向传播时，关于状态值的梯度的方差保持不变。

$$W \sim N\left(0, \sqrt{\frac{2}{n_{in} + n_{out}}}\right)$$

$$W \sim U\left(-\sqrt{\frac{6}{n_{in} + n_{out}}}, \sqrt{\frac{6}{n_{in} + n_{out}}}\right)$$

其中的 W 为权重矩阵， N 表示正态分布（Normal Distribution）， U 表示均匀分布（Uniform Distribution）。下同。

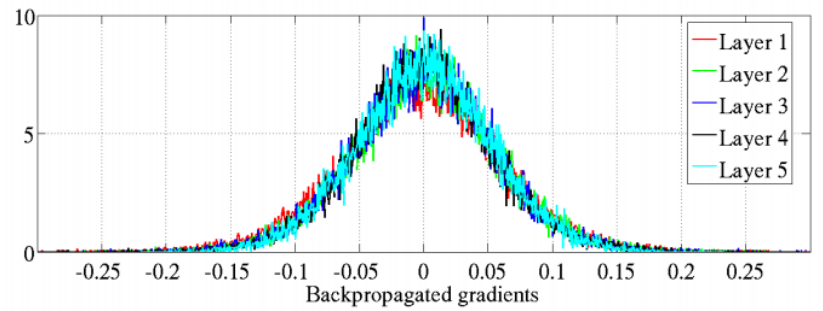
假设激活函数关于0对称，且主要针对于全连接神经网络。适用于tanh和softsign。

即权重矩阵参数应该满足在该区间内的均匀分布。其中的 W 是权重矩阵， U 是Uniform分布，即均匀分布。

Xavier初始化方法比直接用高斯分布进行初始化 W 的优势所在：

一般的神经网络在前向传播时神经元输出值的方差会不断增大，而使用Xavier等方法理论上可以保证每层神经元输入输出方差一致。

随机初始化和Xavier初始化的各层激活值与反向传播梯度比较：

	各层的激活值	各层的反向传播梯度
随机初始化	激活值分布渐渐集中	反向传播力度逐层衰退
Xavier初始化	激活值分布均匀	 <p>反向传播力度保持不变</p>

1.4.1.4 MSRA初始化方法

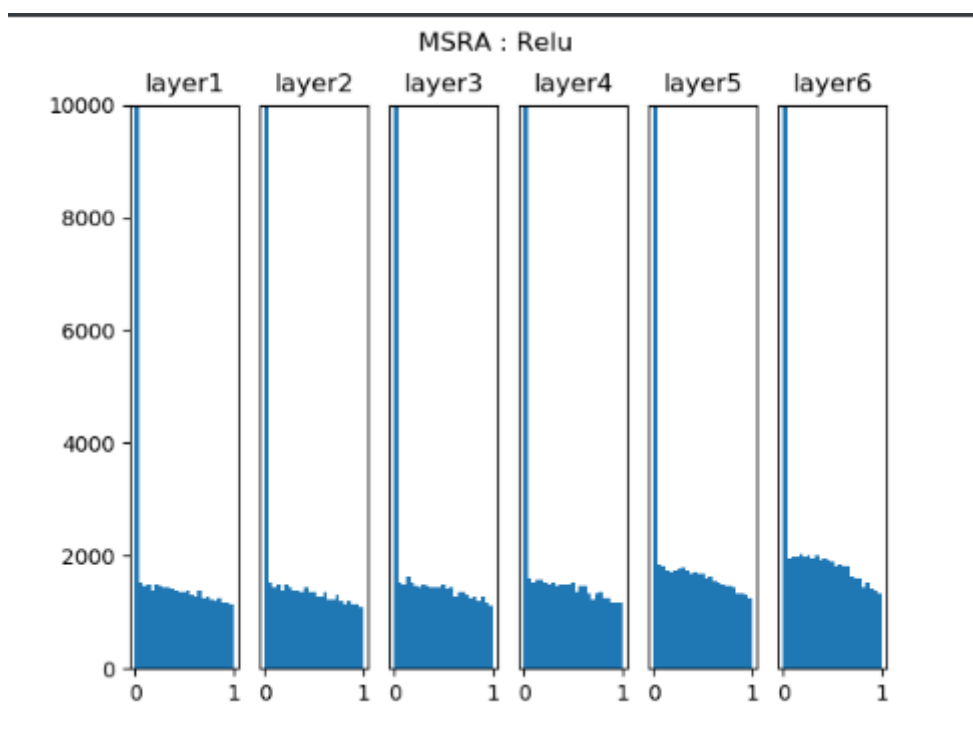
条件：正向传播时，状态值的方差保持不变；反向传播时，关于激活值的梯度的方差保持不变。

只考虑输入个数时，MSRA初始化是一个均值为0，方差为 $2/n$ 的高斯分布，适合于ReLU激活函数：

$$W \sim N\left(0, \sqrt{\frac{2}{n}}\right)$$

$$W \sim U\left(-\sqrt{\frac{6}{n_{in}}}, \sqrt{\frac{6}{n_{out}}}\right)$$

下图的激活值从0到1的分布，在各层都非常均匀，不会由于层的加深而梯度消失，所以，在使用ReLU时，推荐使用MSRA法初始化。



1.4.1.5 小结

ID	网络深度	初始化方法	激活函数	说明
1	单层	零初始化	无	可以
2	双层	零初始化	Sigmoid	错误，不能进行正确的反向传播
3	双层	随机初始化	Sigmoid	可以
4	多层	随机初始化	Sigmoid	激活值分布成凹形，不利于反向传播
5	多层	Xavier初始化	Tanh	正确
6	多层	Xavier初始化	ReLU	激活值分布偏向0，不利于反向传播
7	多层	MSRA初始化	ReLU	正确

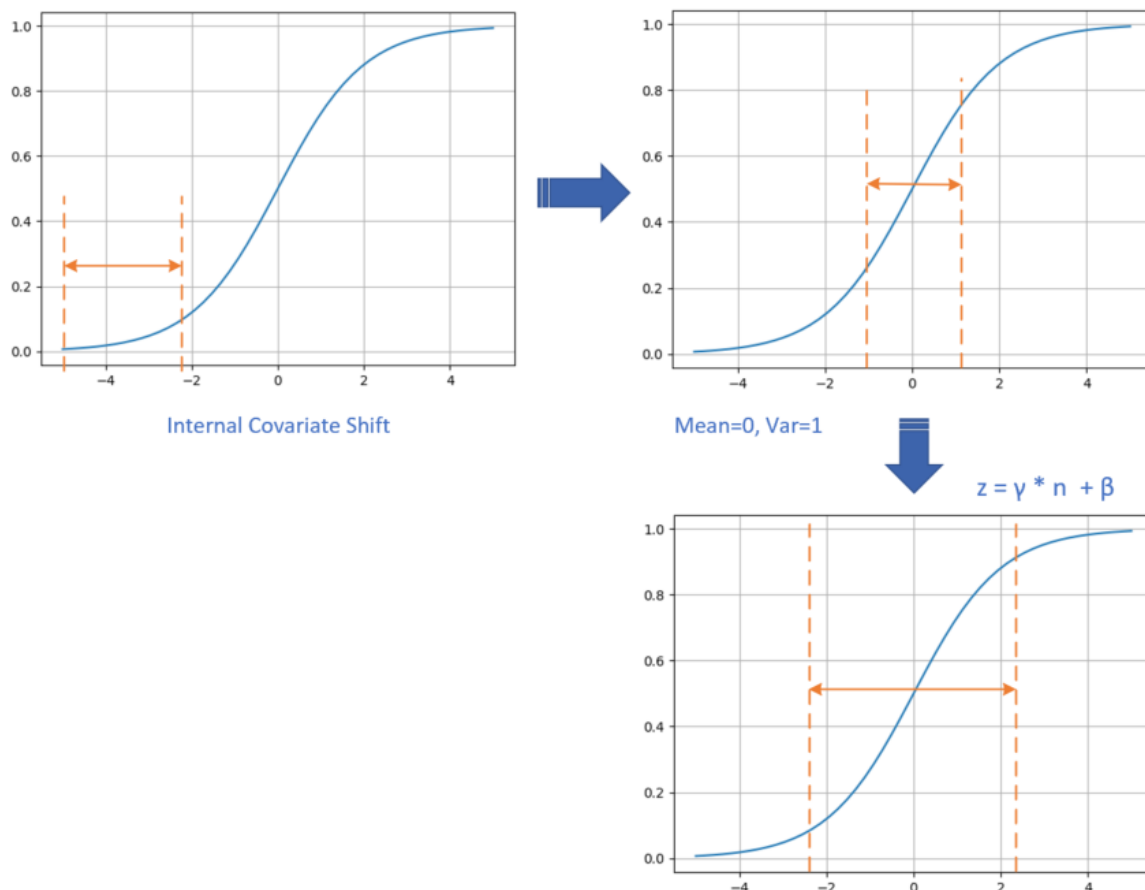
1.4.2 批量归一化

1.4.2.1 原理

既然可以把原始训练样本做归一化，那么如果在深度神经网络的每一层，都可以有类似的手段，也就是说把层之间传递的数据移到0点附近，那么训练效果就应该会很理想。这就是批归一化BN的想法的来源。

BN就是在深度神经网络训练过程中使得每一层神经网络的输入保持相同的分布，致力于将每一层的输入数据正则化成 $N(0, 1)$ 的分布。因此，每次训练的数据必须是mini-batch形式，一般取32，64等数值。

具体的数据处理过程如下图所示。



1.4.2.2 批量归一化的优点

1. 可以选择比较大的初始学习率，让你的训练速度提高。

以前还需要慢慢调整学习率，甚至在网络训练到一定程度时，还需要想着学习率进一步调小的比例选择多少比较合适，现在我们可以采用初始很大的学习率，因为这个算法收敛很快。当然这个算法即使你选择了较小的学习率，也比以前的收敛速度快，因为它具有快速训练收敛的特性；

2. 减少对初始化的依赖

一个不太幸运的初始化，可能会造成网络训练实际很长，甚至不收敛。

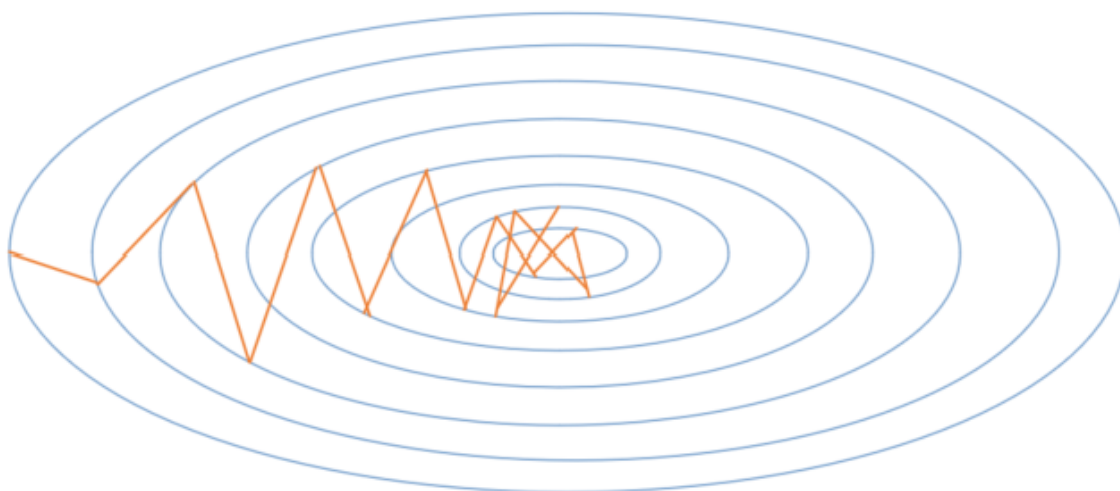
3. 减少对正则的依赖

采用BN算法后，我们会逐步减少对正则的依赖，比如令人头疼的dropout、L2正则项参数的选择问题，或者可以选择更小的L2正则约束参数了，因为BN具有提高网络泛化能力的特性；

1.4.3 梯度下降优化算法

1.4.3.1 随机梯度下降 SGD

下图为梯度搜索轨迹为示意图。



输入和参数

- η - 全局学习率

算法

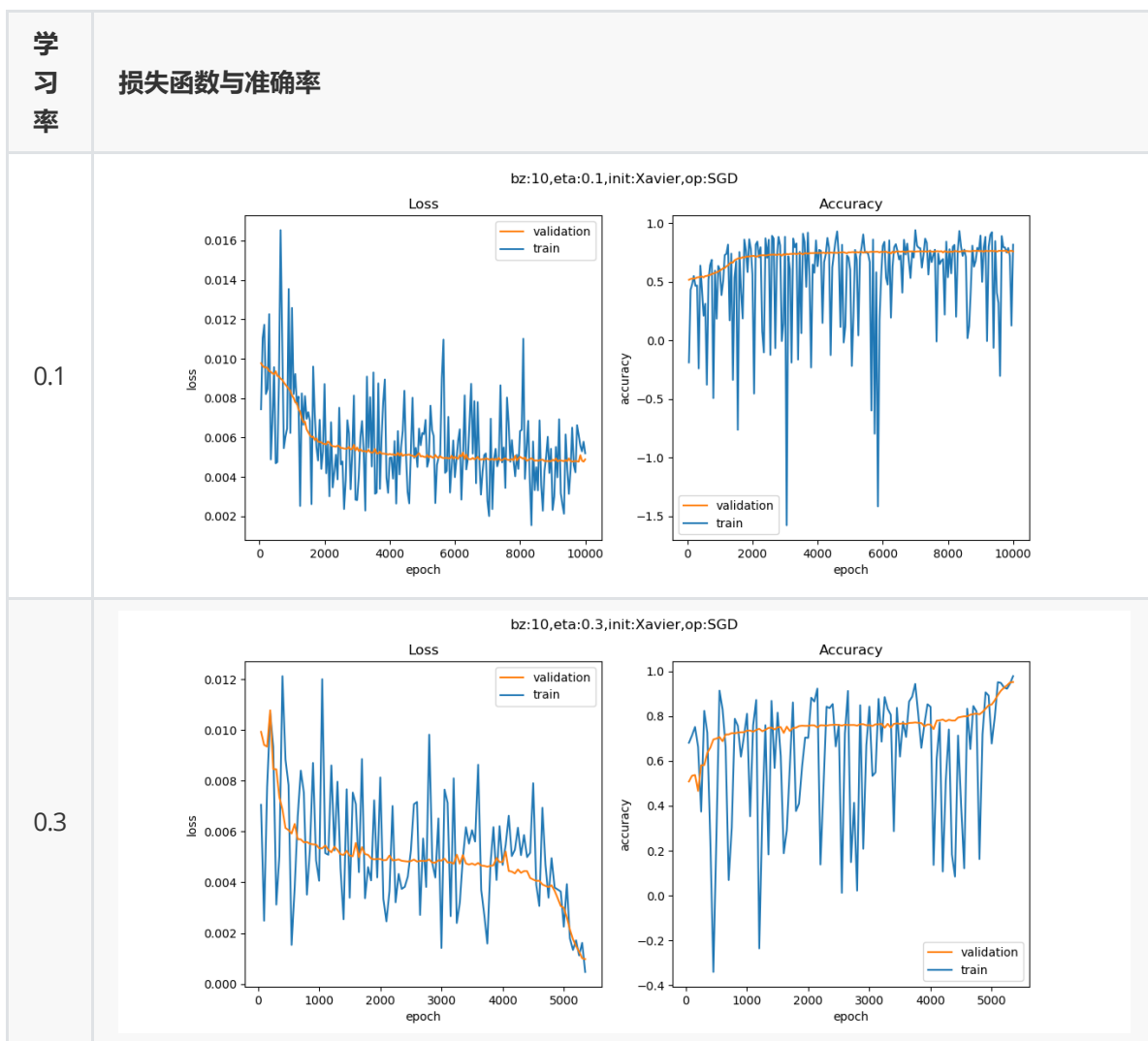
计算梯度: $g_t = \nabla_{\theta} J(\theta_{t-1})$

更新参数: $\theta_t = \theta_{t-1} - \eta \cdot g_t$

随机梯度下降算法，在当前点计算梯度，根据学习率前进到下一点。到中点附近时，由于样本误差或者学习率问题，会发生来回徘徊的现象，很可能会错过最优解。

实际效果

下图为学习率对SGD的影响



SGD的另外一个缺点就是收敛速度慢，如上，在学习率为0.1时，训练10000个epoch不能收敛到预定损失值；学习率为0.3时，训练5000个epoch可以收敛到预定水平。

1.4.3.2 动量算法 Momentum

SGD方法的一个缺点是其更新方向完全依赖于当前batch计算出的梯度，因而十分不稳定，因为数据有噪音。

Momentum算法会观察历史梯度，若当前梯度的方向与历史梯度一致（表明当前样本不太可能为异常点），则会增强这个方向的梯度。若当前梯度与历史梯度方向不一致，则梯度会衰减。

输入和参数

- η - 全局学习率
- α - 动量参数，一般取值为0.5, 0.9, 0.99
- v_t - 当前时刻的动量，初值为0

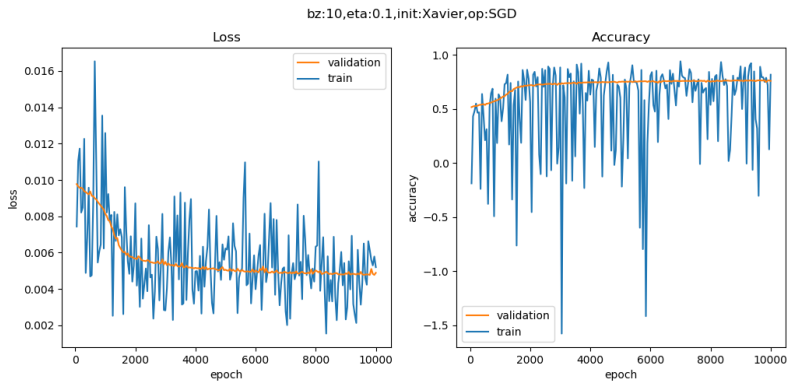
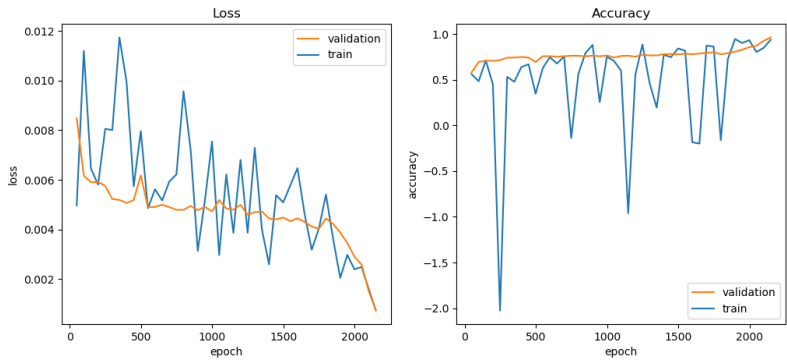
算法

计算梯度: $g_t = \nabla_{\theta} J(\theta_{t-1})$

计算速度更新: $v_t = \alpha \cdot v_{t-1} + \eta \cdot g_t$ (公式1)

更新参数: $\theta_t = \theta_{t-1} - v_t$ (公式2)

下图为SGD和动量法的比较

算法	损失函数和准确率
SGD	 <p>Figure 1: SGD performance plots. The left plot shows Loss vs. epoch (0 to 10000). The training loss (blue) is highly oscillatory, while the validation loss (orange) decreases and stabilizes around 0.005. The right plot shows Accuracy vs. epoch (0 to 10000). The training accuracy (blue) is highly oscillatory, while the validation accuracy (orange) increases and stabilizes around 0.8.</p>
Momentum	 <p>Figure 2: Momentum performance plots. The left plot shows Loss vs. epoch (0 to 2000). The training loss (blue) is highly oscillatory, while the validation loss (orange) decreases and stabilizes around 0.005. The right plot shows Accuracy vs. epoch (0 to 2000). The training accuracy (blue) is highly oscillatory, while the validation accuracy (orange) increases and stabilizes around 0.8.</p>

从比较可以看到，使用同等的超参数设置，普通梯度下降算法经过epoch=10000次没有到达预定0.001的损失值；动量算法经过2000个epoch迭代结束。

在损失函数历史数据图中，中间有一大段比较平坦的区域，梯度值很小，或者是随机梯度下降算法找不到合适的方向前进，只能慢慢搜索。而下侧的动量法，利用惯性，判断当前梯度与上次梯度的关系，如果方向相同，则会加速前进；如果不同，则会减速，并趋向平衡。所以很快地就达到了停止条件。

1.4.3.3 梯度加速算法 NAG

在小球向下滚动的过程中，我们希望小球能够提前知道在哪些地方坡面会上升，这样在遇到上升坡面之前，小球就开始减速。这方法就是Nesterov Momentum，其在凸优化中有较强的理论保证收敛。并且，在实践中Nesterov Momentum也比单纯的Momentum的效果好。

输入和参数

- η - 全局学习率
- α - 动量参数，缺省取值0.9
- v - 动量，初始值为0

算法

临时更新: $\hat{\theta} = \theta_{t-1} - \alpha \cdot v_{t-1}$

前向计算: $f(\hat{\theta})$

计算梯度: $g_t = \nabla_{\hat{\theta}} J(\hat{\theta})$

计算速度更新: $v_t = \alpha \cdot v_{t-1} + \eta \cdot g_t$

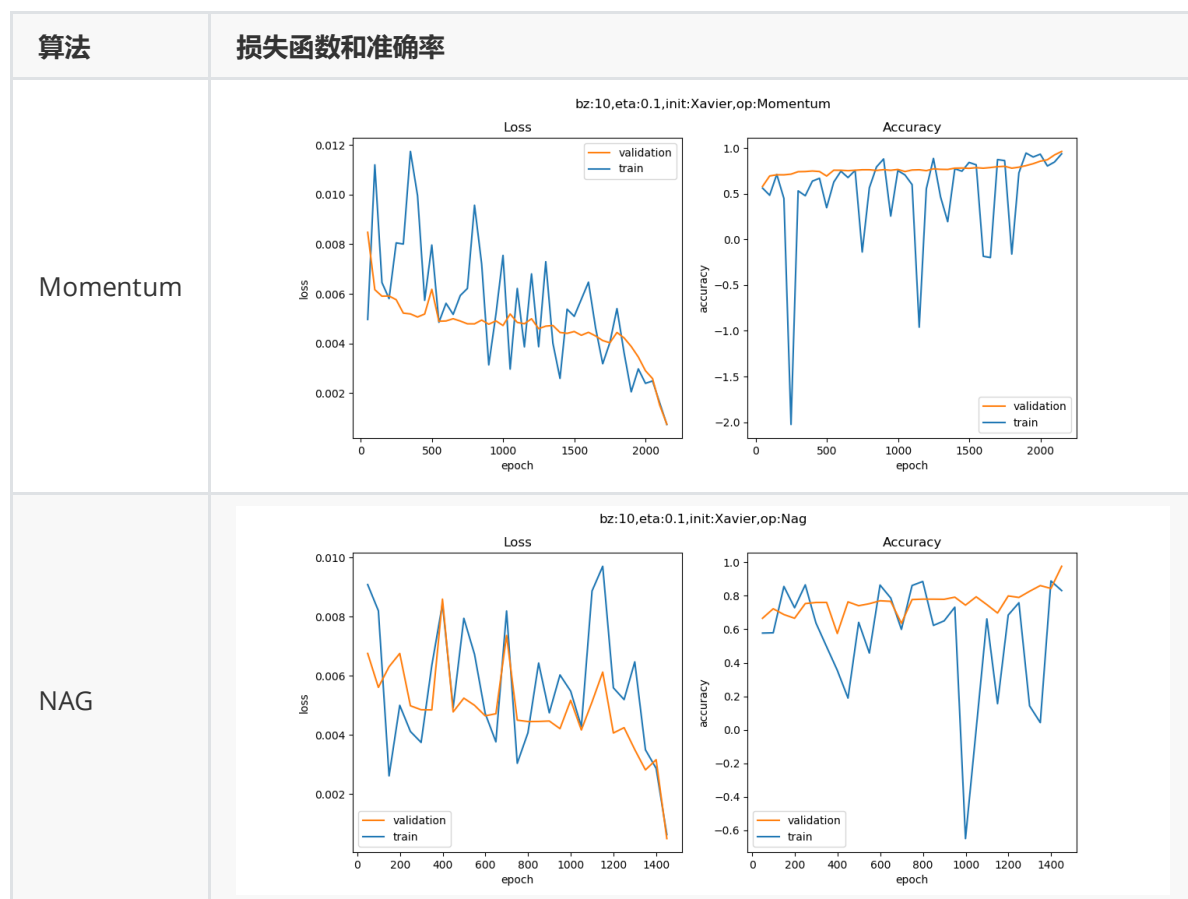
更新参数: $\theta_t = \theta_{t-1} - v_t$

其核心思想是：注意到 momentum 方法，如果只看 $\alpha \cdot v_{t-1}$ 项，那么当前的 θ 经过 momentum 的作用会变成 $\theta - \alpha \cdot v_{t-1}$ 。既然知道了下一步的走向，我们不妨先走一步，到达新的位置“展望”未来，然后在新位置上求梯度，而不是原始的位置。

所以，同 Momentum 相比，梯度不是根据当前位置 θ 计算出来的，而是在移动之后的位置 $\theta - \alpha \cdot v_{t-1}$ 计算梯度。理由是，既然已经确定会移动 $\theta - \alpha \cdot v_{t-1}$ ，那不如之前去看移动后的梯度。

这个改进的目的就是为了提前看到前方的梯度。如果前方的梯度和当前梯度目标一致，那我直接大步迈过去；如果前方梯度同当前梯度不一致，那我就小心点更新。

动量法和NAG法的比较



上图显示，使用动量算法经过2000个epoch迭代结束，NAG算法是加速的动量法，因此只用1400个epoch迭代结束。

1.4.4 自适应学习率算法

1.4.4.1 AdaGrad

AdaGrad是一个基于梯度的优化算法，它的主要功能是：它对不同的参数调整学习率，具体而言，对低频出现的参数进行大的更新，对高频出现的参数进行小的更新。因此，他很适合于处理稀疏数据。

输入和参数

- η - 全局学习率
- ϵ - 用于数值稳定的小常数，建议缺省值为 `1e-6`
- $r = 0$ 初始值

算法

计算梯度： $g_t = \nabla_{\theta} J(\theta_{t-1})$

累计平方梯度: $r_t = r_{t-1} + g_t \odot g_t$

计算梯度更新: $\Delta\theta = \frac{\eta}{\epsilon + \sqrt{r_t}} \odot g_t$

更新参数: $\theta_t = \theta_{t-1} - \Delta\theta$

从AdaGrad算法中可以看出,随着算法不断迭代, r 会越来越大,整体的学习率会越来越小。所以,一般来说AdaGrad算法一开始是激励收敛,到了后面就慢慢变成惩罚收敛,速度越来越慢。 r 值的变化如下:

0. $r_0 = 0$
1. $r_1 = g_1^2$
2. $r_2 = g_1^2 + g_2^2$
3. $r_3 = g_1^2 + g_2^2 + g_3^2$

在SGD中,随着梯度的增大,我们的学习步长应该是增大的。但是在AdaGrad中,随着梯度 g 的增大, r 也在逐渐的增大,且在梯度更新时 r 在分母上,也就是整个学习率是减少的,这是为什么呢?

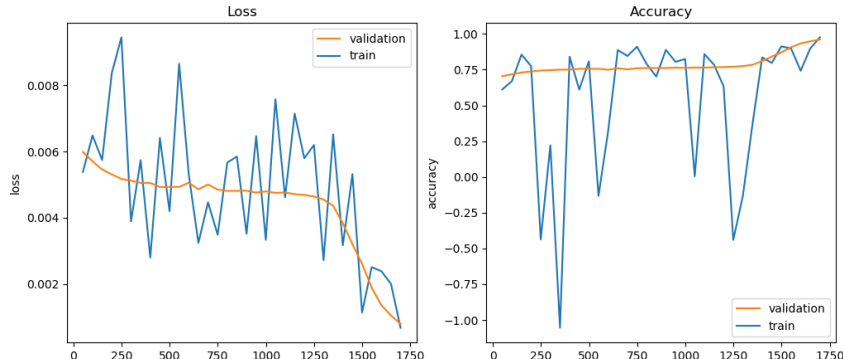
这是因为随着更新次数的增大,我们希望学习率越来越慢。因为我们认为在学习率的最初阶段,我们距离损失函数最优解还很远,随着更新次数的增加,越来越接近最优解,所以学习率也随之变慢。

但是当某个参数梯度较小时,累积和也会小,那么更新速度就大。

经验上已经发现,对于训练深度神经网络模型而言,从训练开始时积累梯度平方会导致有效学习率过早和过量的减小。AdaGrad在某些深度学习模型上效果不错,但不是全部。

实际效果

AdaGrad算法的学习率设置如下:

初始学习率	损失函数值变化
eta=0.3	
eta=0.5	
eta=0.7	<div><p>bz:10,eta:0.7,init:Xavier,op:AdaGrad</p></div>

上面表明,我们设定不同的初始学习率,分别为0.3、0.5、0.7,可以看到学习率为0.7时,收敛得最快,只用1750个epoch;学习率为0.5时用了3000个epoch;学习率为0.3时用了8000个epoch。所以,对于AdaGrad来说,可以在开始时把学习率的值设置大一些,因为它会衰减得很快。

1.4.4.2 AdaDelta

AdaDelta法是AdaGrad 法的一个延伸，它旨在解决它学习率不断单调下降的问题。相比计算之前所有梯度值的平方和，AdaDelta法仅计算在一个大小为w的时间区间内梯度值的累积和。

输入和参数

- ϵ - 用于数值稳定的小常数，建议缺省值为1e-5
- $\alpha \in [0, 1)$ - 衰减速率，建议0.9
- s - 累积变量，初始值0
- r - 累积变量变化量，初始为0

算法

计算梯度: $g_t = \nabla_{\theta} J(\theta_{t-1})$

累积平方梯度: $s_t = \alpha \cdot s_{t-1} + (1 - \alpha) \cdot g_t \odot g_t$

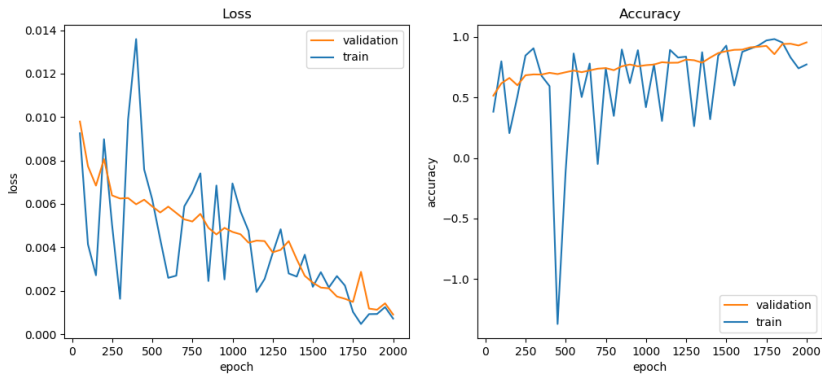
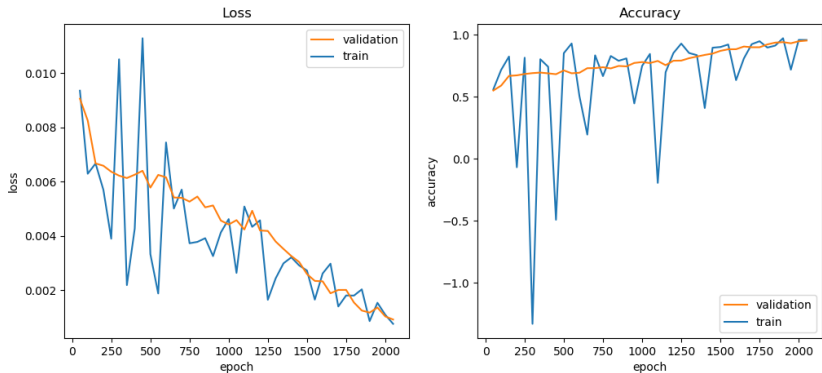
计算梯度更新: $\Delta\theta = \sqrt{\frac{r_{t-1} + \epsilon}{s_t + \epsilon}} \odot g_t$

更新梯度: $\theta_t = \theta_{t-1} - \Delta\theta$

更新变化量: $r = \alpha \cdot r_{t-1} + (1 - \alpha) \cdot \Delta\theta \odot \Delta\theta$

实际效果

下表为AdaDelta法的学习率设置

初始学习率	损失函数值
eta=0.1	<p>bz:10,eta:0.1,init:Xavier,op:AdaDelta</p> 
eta=0.01	<p>bz:10,eta:0.01,init:Xavier,op:AdaDelta</p> 

从上可以看到，初始学习率设置为0.1或者0.01，对于本算法来说都是一样的，这是因为算法中用 r 来代替学习率。

1.4.4.3 均方根反向传播 RMSProp

RMSprop法要解决AdaGrad的学习率缩减问题。

输入和参数

- η - 全局学习率，建议设置为0.001
- ϵ - 用于数值稳定的小常数，建议缺省值为1e-8
- α - 衰减速率，建议缺省取值0.9
- r - 累积变量矩阵，与 θ 尺寸相同，初始化为0

算法

计算梯度: $g_t = \nabla_{\theta} J(\theta_{t-1})$

累计平方梯度: $r = \alpha \cdot r + (1 - \alpha)(g_t \odot g_t)$

计算梯度更新: $\Delta\theta = \frac{\eta}{\sqrt{r+\epsilon}} \odot g_t$

更新参数: $\theta_t = \theta_{t-1} - \Delta\theta$

RMSprop也将学习率除以一个指数衰减的衰减均值。为了进一步优化损失函数在更新中存在摆动幅度过大的问题，并且进一步加快函数的收敛速度，RMSProp算法对权重 W 和偏置 b 的梯度使用了微分平方加权平均数，这种做法有利于消除了摆动幅度大的方向，用来修正摆动幅度，使得各个维度的摆动幅度都较小。另一方面也使得网络函数收敛更快。

其中， r 值的变化如下：

$$0. r_0 = 0$$

$$1. r_1 = 0.1g_1^2$$

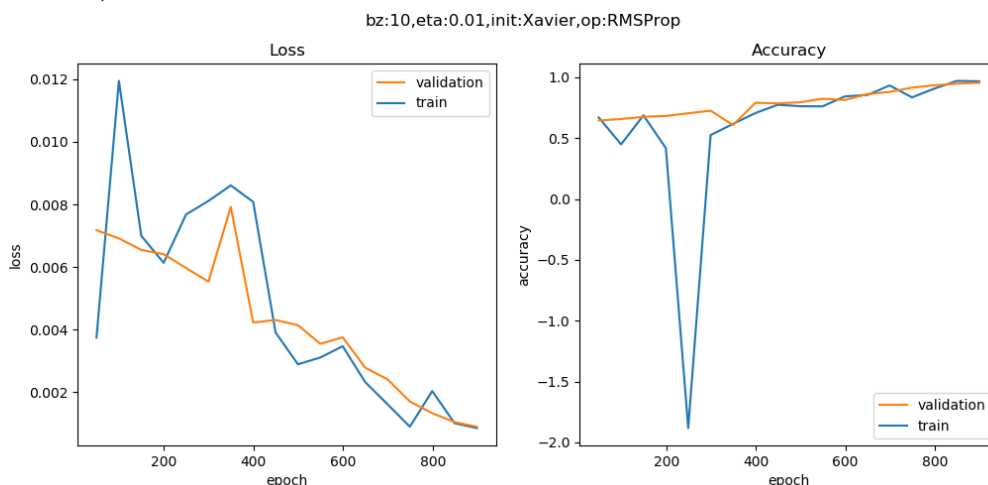
$$2. r_2 = 0.9r_1 + 0.1g_2^2 = 0.09g_1^2 + 0.1g_2^2$$

$$3. r_3 = 0.9r_2 + 0.1g_3^2 = 0.081g_1^2 + 0.09g_2^2 + 0.1g_3^2$$

与AdaGrad相比， r_3 要小很多，那么计算出来的学习率也不会衰减的太厉害。注意，在计算梯度更新时，分母开始时是个小于1的数，而且非常小，所以如果全局学习率设置过大的话，比如0.1，将会造成开始的步子迈得太大，而且久久不能收缩步伐，损失值也降不下来。

实际效果

|eta=0.01|



1.4.4.4 Adam - Adaptive Moment Estimation

计算每个参数的自适应学习率，相当于RMSProp + Momentum的效果，Adam算法在RMSProp算法基础上对小批量随机梯度也做了指数加权移动平均。和AdaGrad算法、RMSProp算法以及AdaDelta算法一样，目标函数自变量中每个元素都分别拥有自己的学习率。

输入和参数

- t - 当前迭代次数
- η - 全局学习率，建议缺省值为0.001
- ϵ - 用于数值稳定的小常数，建议缺省值为1e-8
- β_1, β_2 - 矩估计的指数衰减速率， $\in [0, 1)$ ，建议缺省值分别为0.9和0.999

算法

计算梯度: $g_t = \nabla_{\theta} J(\theta_{t-1})$

计数器加一: $t = t + 1$

更新有偏一阶矩估计: $m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$

更新有偏二阶矩估计: $v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2)(g_t \odot g_t)$

修正一阶矩的偏差: $\hat{m}_t = m_t / (1 - \beta_1^t)$

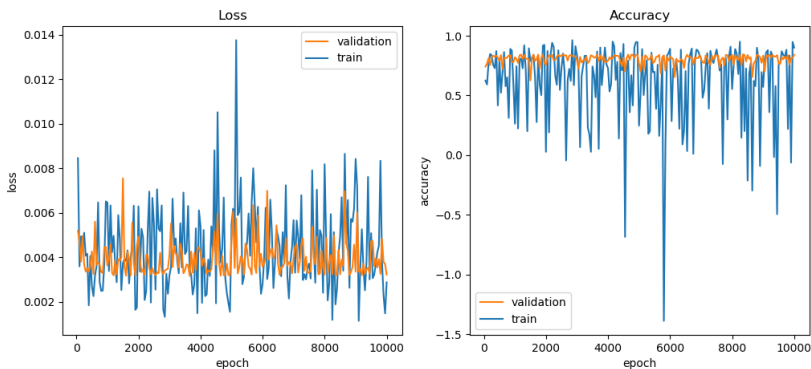
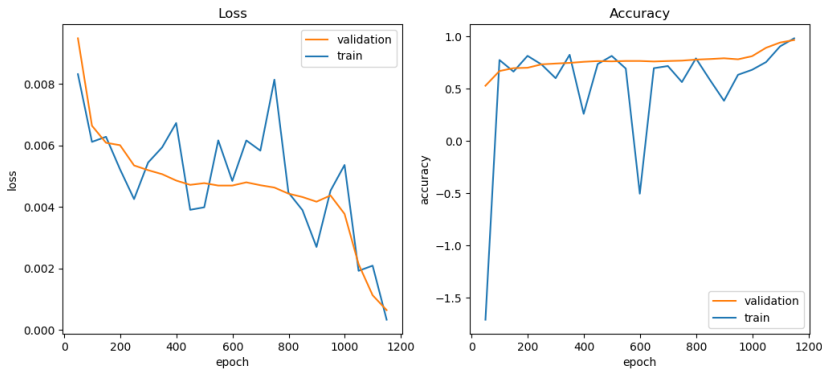
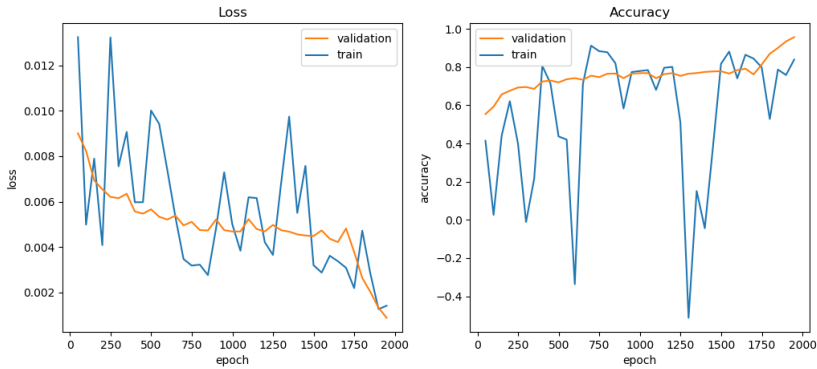
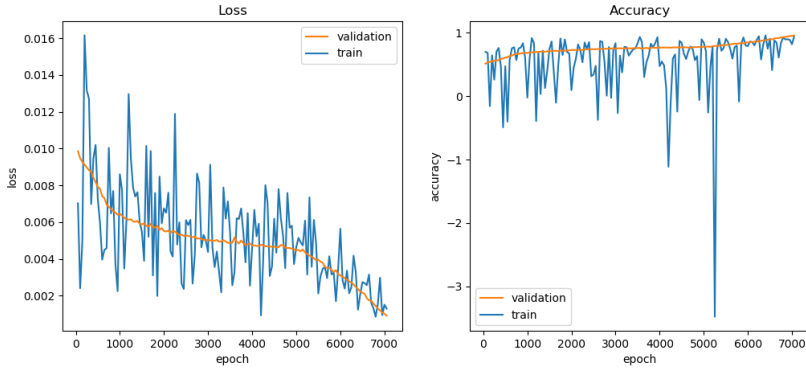
修正二阶矩的偏差: $\hat{v}_t = v_t / (1 - \beta_2^t)$

计算梯度更新: $\Delta\theta = \eta \cdot \hat{m}_t / (\epsilon + \sqrt{\hat{v}_t})$

更新参数: $\theta_t = \theta_{t-1} - \Delta\theta$

实际效果

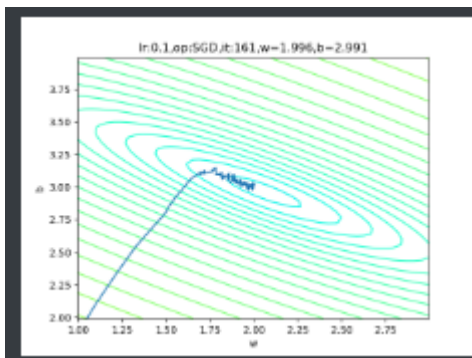
下表为Adam法的学习率设置

初始学习率	损失函数值
eta=0.1	<p>bz:10,eta:0.1,init:Xavier,op:Adam</p> 
	迭代了10000次，但是损失值没有降下来，因为初始学习率0.1太高了
eta=0.01	<p>bz:10,eta:0.01,init:Xavier,op:Adam</p> 
	比较合适的学习率
eta=0.005	<p>bz:10,eta:0.005,init:Xavier,op:Adam</p> 
	学习率较低
eta=0.001	<p>bz:10,eta:0.001,init:Xavier,op:Adam</p> 
	初始学习率太低，收敛到目标损失值的速度慢

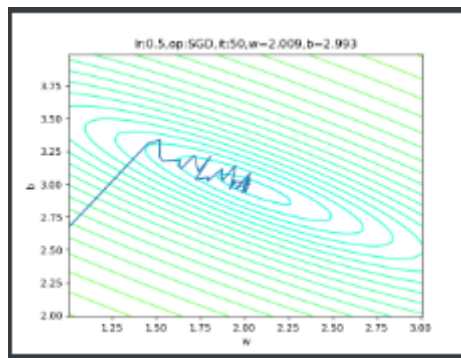
由于Adam继承了RMSProp的传统，所以学习率不宜设置太高，从上面的比较可以看到，初始学习率设置为0.01时比较理想。

1.4.5 算法效果比较

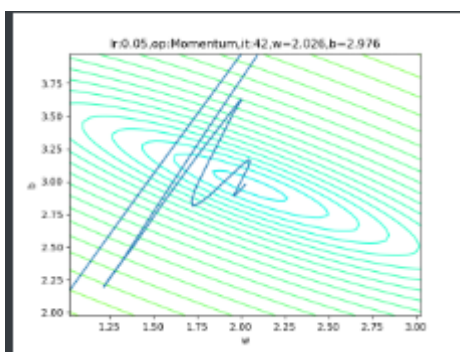
表15-10 各种算法的效果比较



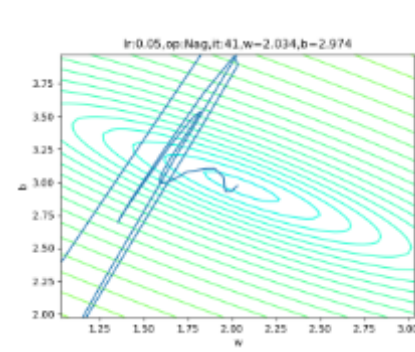
SGD当学习率为0.1时，需要很多次迭代才能逐渐向中心靠近



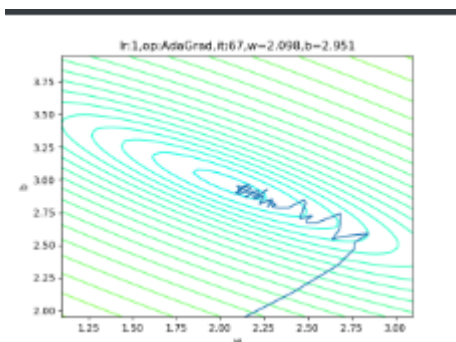
SGD当学习率为0.5时，会比较快速地向中心靠近，但是在中心的附近有较大震荡



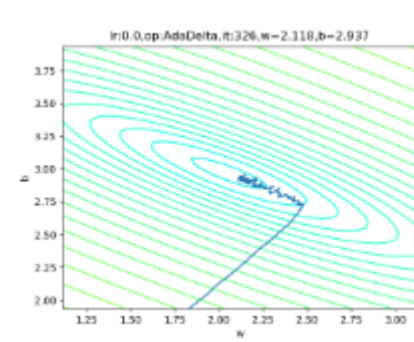
Momentum由于惯性存在，一下子越过了中心点，但是很快就会得到纠正



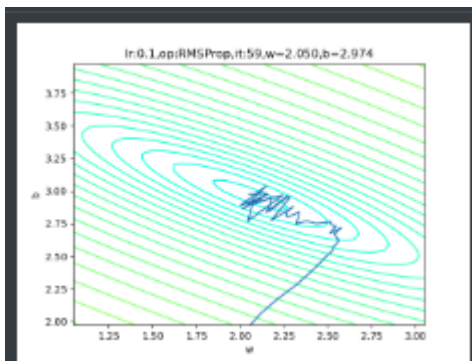
Nag是Momentum的改进，有预判方向功能



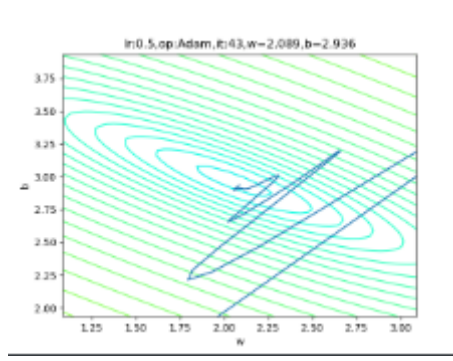
AdaGrad的学习率在开始时可以设置大一些，因为会很快衰减



AdaDelta即使把学习率设置为0，也不会影响，因为有内置的学习率策略



RMSProp解决AdaGrad的学习率下降问题，即使学习率设置为0.1，收敛也会快



Adam到达中点的路径比较直接

结果

- SGD: 接近中点的过程很曲折, 步伐很慢, 甚至有反方向的, 容易陷入局部最优。
- Momentum: 快速接近中点, 但中间跳跃较大。
- RMSProp: 接近中点很曲折, 但是没有反方向的, 用的步数比SGD少, 跳动较大, 有可能摆脱局部最优解的。
- Adam: 快速接近中点, 难怪很多人喜欢用这个优化器。

1.5 正则化

正则化的英文为Regularization, 用于防止过拟合。

1.5.1 过拟合

泛化:

在深度神经网络中, 我们遇到的另外一个挑战, 就是网络的泛化问题。所谓泛化, 就是模型在测试集上的表现要和训练集上一样好。经常有这样的例子: 一个模型在训练集上千锤百炼, 能到达99%的准确率, 拿到测试集上一试, 准确率还不到90%。这说明模型过度拟合了训练数据, 而不能反映真实世界的情况。解决过度拟合的手段和过程, 就叫做泛化。

1.5.1.1 过拟合的原因

1. 训练集的数量和模型的复杂度不匹配, 样本数量级小于模型的参数
2. 训练集和测试集的特征分布不一致
3. 样本噪音大, 使得神经网络学习到了噪音, 正常样本的行为被抑制
4. 迭代次数过多, 过分拟合了训练数据, 包括噪音部分和一些非重要特征

既然模型过于复杂, 那么我们简化模型不就行了吗? 为什么要用复杂度不匹配的模型呢? 有两个原因:

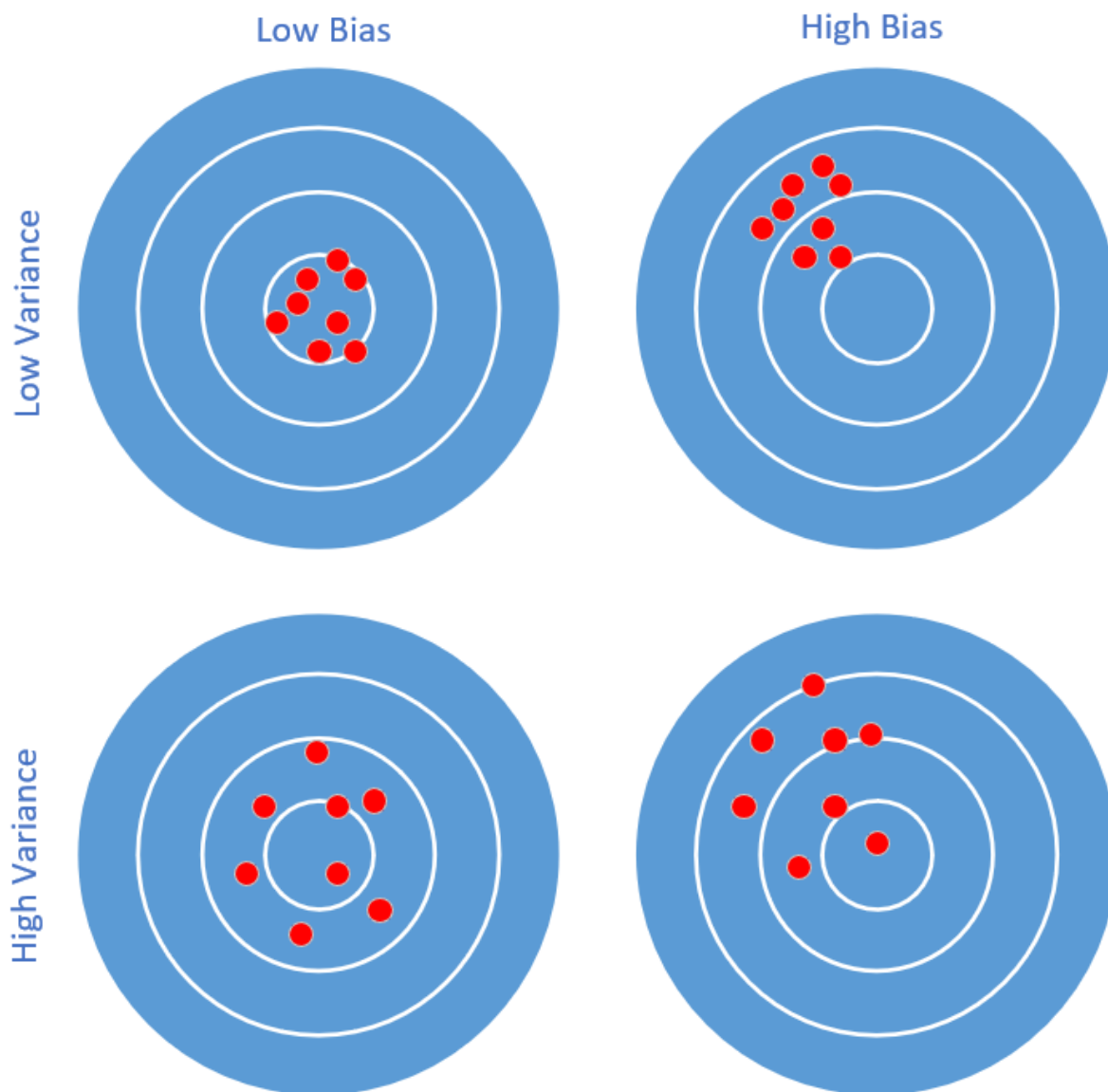
1. 因为有的模型以及非常成熟了, 比如VGG16, 可以不调参而直接用于你自己的数据训练, 此时如果你的数据数量不够多, 但是又想使用现有模型, 就需要给模型加正则项了。
2. 使用相对复杂的模型, 可以比较快速地使得网络训练收敛, 以节省时间。

1.5.1.2 解决过拟合

1. 数据扩展
2. 正则
3. 丢弃法
4. 早停法
5. 集成学习法
6. 特征工程 (属于传统机器学习范畴, 不在此处讨论)
7. 简化模型, 减小网络的宽度和深度

1.5.2 偏差和方差

先用一个直观的例子来理解偏差和方差。



总结一下，不同偏差和方差反映的射手的特点如下表所示。

	低偏差	高偏差
低方差	射手很稳，枪的准星也很准。	射手很稳，但是枪的准星有问题，所有子弹都固定地偏向一侧。
高方差	射手不太稳，但枪的准星没问题，虽然弹着点分布很散，但没有整体偏移。	射手不稳，而且枪的准星也有问题，弹着点分布很散且有规律地偏向一侧。

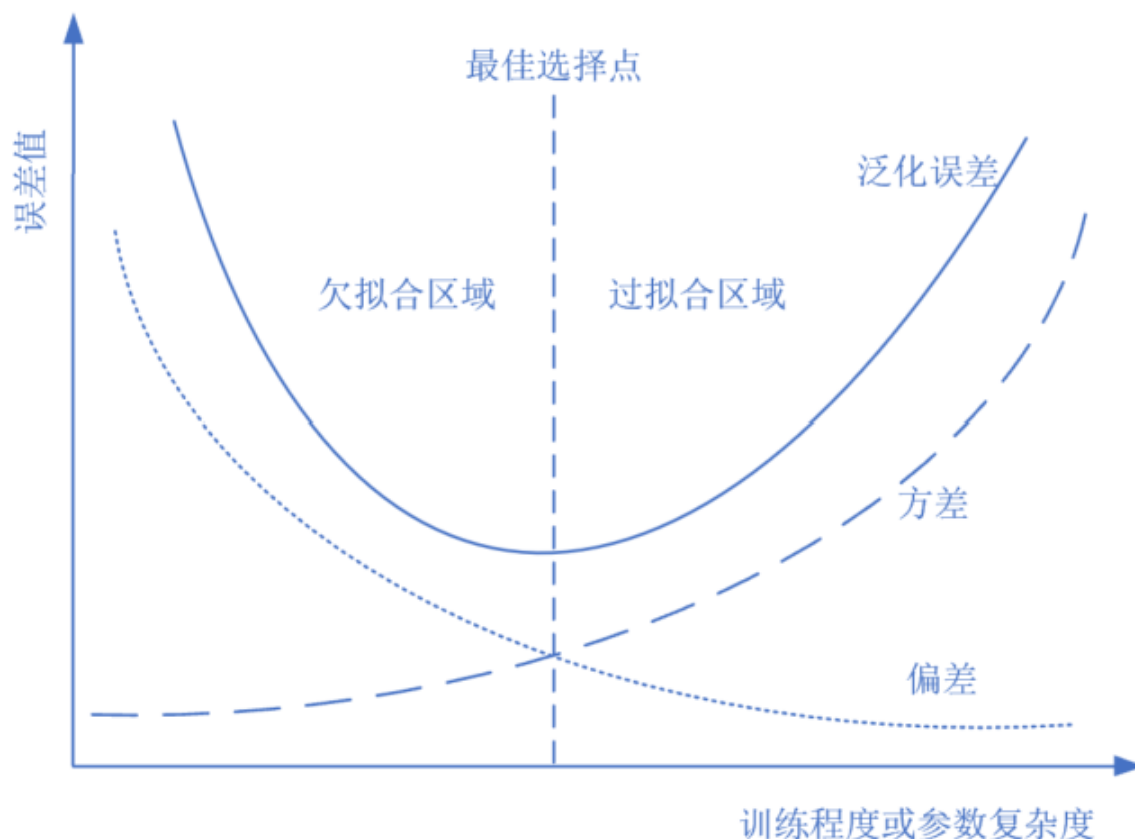
说明如下：

- 偏差：度量了学习算法的期望与真实结果的偏离程度，即学习算法的拟合能力。
- 方差：训练集与验证集的差异造成的模型表现的差异。
- 噪声：当前数据集上任何算法所能到达的泛化误差的下线，即学习问题本身的难度。

想当然地，如果希望偏差与方差越小越好，但实际并非如此。一般来说，偏差与方差是有冲突的，称为偏差-方差窘境 (bias-variance dilemma)。

- 给定一个学习任务，在训练初期，由于训练不足，网络的拟合能力不够强，偏差比较大，也是由于拟合能力不强，数据集的特征也无法使网络产生显著变化，也就是欠拟合的情况。
- 随着训练程度的加深，网络的拟合能力逐渐增强，训练数据的特征也能够渐渐被网络学到。

- 充分训练后，网络的拟合能力已非常强，训练数据的微小特征都会导致网络发生显著变化，当训练数据自身的、非全局的特征被网络学到了，则将发生过拟合。



在上图中，随着训练程度的增加，偏差（点线）一路下降，但是方差（虚线）一路上升，整体误差（实线，偏差+方差+噪音误差）呈U形，最佳平衡点就是U形的最低点。

1.5.3 解决过拟合之早停法

一般的做法是，在训练的过程中，记录到目前为止最好的validation 准确率，当连续N次Epoch（比如N=10或者更多次）没达到最佳准确率时，则可以认为准确率不再提高了。此时便可以停止迭代了（Early Stopping）。这种策略也称为“No-improvement-in-N”，N即Epoch的次数，可以根据实际情况取，如10、20、30.....

算法描述如下：

```
1  初始化
2      初始权重均值参数: theta = theta_0
3      迭代次数: i = 0
4      忍耐次数: patience = N (e.g. N=10)
5      忍耐次数计数器: counter = 0
6      验证集损失函数值: lastLoss = 10000 (给一个特别大的数值)
7
8  while (epoch < maxEpoch) 循环迭代训练过程
9      正向计算，反向传播更新theta
10     迭代次数加1: i++
11     计算验证集损失函数值: newLoss = loss
12     if (newLoss < lastLoss) // 新的损失值更小
13         忍耐次数计数器归零: counter = 0
14         记录当前最佳权重矩阵训练参数: theta_best = theta
15         记录当前迭代次数: i_best = i
16         更新最新验证集损失函数值: lastLoss = newLoss
```

```
17     else // 新的损失值大于上一步的损失值
18         忍耐次数计数器加1: counter++
19         if (counter >= patience) 停止训练!!!
20     end if
21 end while
```

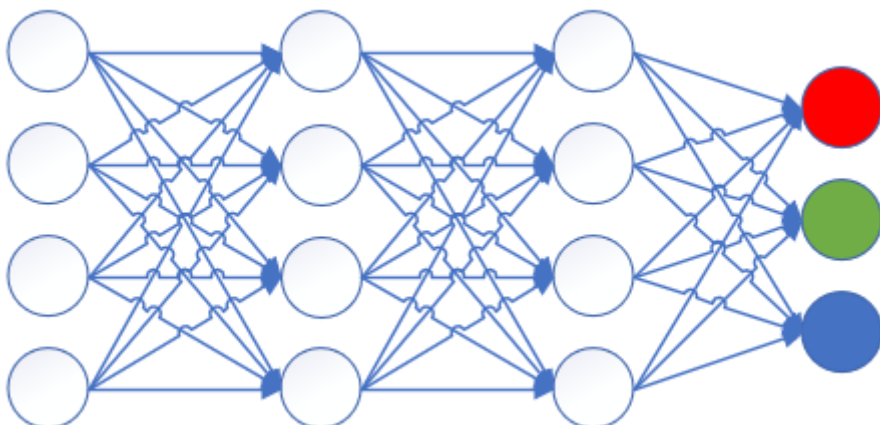
此时，`theta_best` 和 `i_best` 就是最佳权重值和迭代次数。

要注意的问题

1. 门限值 `patience` 不能太小，比如小于5，因为很可能在5个 `epoch` 之外，损失函数值又会再次下降
2. `patience` 不能太大，比如大于30，因为在这30个 `epoch` 之内，由于样本数量少和数据 `shuffle` 的关系，很可能某个 `epoch` 的损失函数值会比上一次低，这样忍耐次数计数器 `counter` 就清零了，从而不能及时停止。
3. 当样本数量少时，为了获得平滑的变化曲线，可以考虑使用加权平均的方式处理当前和历史损失函数值，以避免某一次的高低带来的影响。

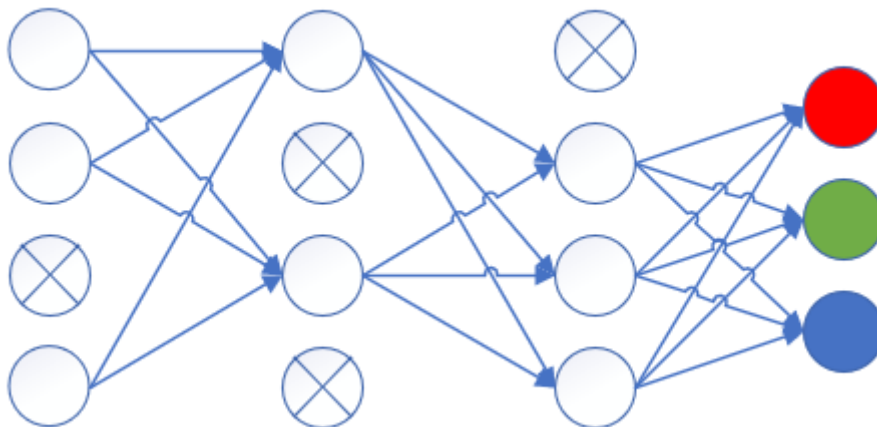
1.5.4 解决过拟合之丢弃法

我们假设原来的神经网络是这个结构，最后输出三分类结果，如图所示。



Dropout可以作为训练深度神经网络的一种正则方法供选择。在每个训练批次中，通过忽略一部分的神经元（让其隐层节点值为0），可以明显地减少过拟合现象。这种方式可以减少隐层节点间的相互作用，高层的神经元需要低层的神经元的输出才能发挥作用，如果高层神经元过分依赖某个低层神经元，就会有过拟合发生。在一次正向/反向的过程中，通过随机丢弃一些神经元，迫使高层神经元和其它的一些低层神经元协同工作，可以有效地防止神经元因为接收到过多的同类型参数而陷入过拟合的状态，来提高泛化程度。

丢弃后的结果如图所示。



其中有叉子的神经元在本次迭代训练中被暂时的封闭了，在下次迭代训练中，再随机地封闭一些神经元，同一个神经元也许被连续封闭两次，也许一次都没有被封闭，完全随机。封闭多少个神经元是由一个超参来控制的，叫做丢弃率。

1.5.5 解决过拟合之数据拓展

过拟合的原因之一是训练数据不够，而在现代的机器学习中，数据量却是不成问题，因为通过互联网上用户的交互行为，或者和手机App的交互行为，可以收集大量的数据用于网络训练。

但是对于一些图片类数据，不是很容易从原始渠道搞到，所以可以采用增加一些假数据的方式来满足需要，尤其是当这个任务是分类任务时，更加适合。

对于拟合任务，在当前样本数据附近增加一些假的样本数据并无意义，相当于把整个样本数据变“粗”。对于概率密度计算任务，增加假样本很可能破坏原始样本的概率密度。

通过丰富的图像处理手段，我们往往可以把样本数量翻好几倍。下面我们通过手写数字识别的例子，来说明如何做简单的图片增强。

1.5.5.1 图像数据增强主要包括：

1. 旋转
2. 缩放
3. 平移和添加噪音
4. 其他

- 翻转图像：即左右镜像，或者上下镜像，但是对于数字识别来说不合适
- 剪裁图像：从图像中随机选择一部分，再调整为原始图像大小，对于本例也不适合
- 颜色变化：对图像进行颜色抖动，即对RGB值进行随机扰动，如椒盐噪声和高斯噪声
- 对比度变化：通过修改HSV空间中的色调和饱和度来改变图像的对比度，也可以用直方图均衡化
- 亮度变化：改变整个图像的亮度
- 颜色增强：对于颜色暗淡的图片进行全图的颜色增强

1.5.5.2 多样本合成法

SMOTE：Synthetic Minority Over-sampling Technique，通过人工合成新样本来处理样本不平衡问题，提升分类器性能。

SamplePairing：训练集中随机抽取两张图片分别经过基础数据增强操作（如随机翻转等）处理后经像素取平均值的形式叠加合成一个新的样本，标签为原样本标签中的一种。

Mixup: Mixup是基于邻域风险最小化 (VRM) 原则的数据增强方法, 使用线性插值得到新样本数据。在邻域风险最小化原则下, 根据特征向量线性插值将导致相关目标线性插值的先验知识, 可得出简单且与数据无关的mixup公式。

1.5.6 结果过拟合之集成学习

当数据集有问题, 或者网络学习能力不足, 或准确度不够时, 我们可以采取集成学习的方法, 来提升性能。说得通俗一些, 就是发挥团队的智慧, 根据团队中不同背景、不同能力的成员的独立意见, 通过某种决策方法来解决一个问题。所以集成学习也称为多分类器系统(multi-classifier system)、基于委员会的学习(committee-based learning)等

一个简单的集成学习的示意图。

