

第四周

1. 权值初始化

1.1 梯度消失与爆炸

$$H_2 = H_1 \times W_2$$

$$\begin{aligned}\Delta W_2 &= \frac{\partial Loss}{\partial out} \times \frac{\partial out}{\partial H_2} \times \frac{\partial H_2}{\partial w_2} \\ &= \frac{\partial Loss}{\partial out} \times \frac{\partial out}{\partial H_2} \times H_1\end{aligned}$$

ΔW_2 的梯度取决于上一层的输出 H_1

梯度消失:

$$H_1 \rightarrow 0 \Rightarrow \Delta W_2 \rightarrow 0$$

梯度爆炸:

$$H_1 \rightarrow \infty \Rightarrow \Delta W_2 \rightarrow \infty$$

梯度消失与爆炸

Gradient Vanishing and Exploding

$H_{11} = \sum_{i=0}^n X_i * W_{1i} \quad D(X*Y) = D(X)*D(Y)$

$D(H_{11}) = \sum_{i=0}^n D(X_i) * D(W_{1i})$

$= n * (1 * 1)$

$= n$

$std(H_{11}) = \sqrt{D(H_{11})} = \sqrt{n}$

$D(H_1) = n * D(X) * D(W) = 1$

$D(W) = \frac{1}{n} \Rightarrow std(W) = \sqrt{\frac{1}{n}}$

Input layer Hidden layers Output layer

$X \quad W_1 \quad H_1 \quad W_2 \quad H_2 \quad W_3 \quad out$

deepshare.net
深度之眼

1.2 Xavier初始化

Xavier初始化

Xavier初始化

Xavier Initialization

方差一致性：保持数据尺度维持在恰当范围，通常方差为1

激活函数：饱和函数，如Sigmoid, Tanh

$$n_i * D(W) = 1$$

$$W \sim U[-a, a]$$

$$n_{i+1} * D(W) = 1$$

$$D(W) = \frac{(-a-a)^2}{12} = \frac{(2a)^2}{12} = \frac{a^2}{3}$$

$$\Rightarrow D(W) = \frac{2}{n_i + n_{i+1}}$$

$$\frac{2}{n_i + n_{i+1}} = \frac{a^2}{3} \Rightarrow a = \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}$$

$$\Rightarrow W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}, \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}\right]$$

参考文献：《Understanding the difficulty of training deep feedforward neural networks》

Kaiming初始化

Kaiming初始化

Kaiming Initialization

方差一致性：保持数据尺度维持在恰当范围，通常方差为1

激活函数：ReLU及其变种

$$D(W) = \frac{2}{n_i}$$

$$D(W) = \frac{2}{(1+a^2) * n_i}$$

$$\text{std}(W) = \sqrt{\frac{2}{(1+a^2) * n_i}}$$

参考文献：《Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification》

其中， a 为负半轴斜率

1.3 权值初始化方法

- Xavier均匀分布
- Xavier标准正态分布
- Kaiming均匀分布
- Kaiming标准正态分布
- 均匀分布
- 正态分布
- 常数分布
- 正交矩阵初始化
- 单位矩阵初始化
- 稀疏矩阵初始化

```

1 # 计算激活函数的方差变换尺度
2 nn.init.calculate_gain(nonlinearity, # 激活函数名称
3 param=None) # 激活函数参数, 如Leaky ReLU的negative_slope

```

2. 损失函数

2.1 损失函数概念

- **损失函数**: 衡量模型输出与真实标签的差异
- 损失函数 (Loss Function) : $Loss = f(y', y)$
- 代价函数 (Cost Function) : $Cost = \frac{1}{N} \sum_i^N f(y'_i, y_i)$
- 目标函数 (Objective Function) : $Obj = Cost + Regularization$ (L1/L2·正则项)

```

1 class _Loss(Module):
2     def __init__(self, reduction='mean'):
3         super(_Loss, self).__init__()
4         self.reduction = reduction

```

2.2 交叉熵损失函数

损失函数

Loss Function

交叉熵 = 信息熵 + 相对熵

交叉熵: $H(P, Q) = -\sum_{i=1}^N P(x_i) \log Q(x_i)$

自信息: $I(x) = -\log[p(x)]$

熵: $H(P) = E_{x \sim p}[I(x)] = -\sum_{i=1}^N P(x_i) \log P(x_i)$

相对熵: $D_{KL}(P, Q) = E_{x \sim p} \left[\log \frac{P(x)}{Q(x)} \right]$

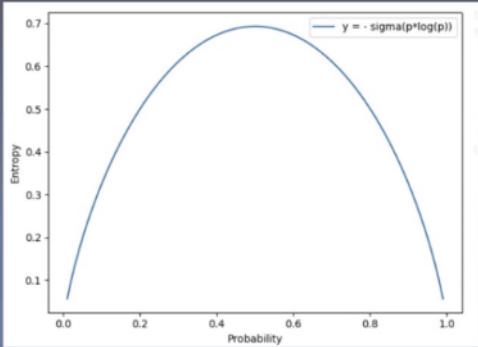
$$= E_{x \sim p} [\log P(x) - \log Q(x)]$$

$$= \sum_{i=1}^N P(x_i) [\log P(x_i) - \log Q(x_i)]$$

$$= \sum_{i=1}^N P(x_i) \log P(x_i) - \sum_{i=1}^N P(x_i) \log Q(x_i)$$

交叉熵: $H(P, Q) = D_{KL}(P, Q) + H(P)$

$$= H(P, Q) - H(P)$$



- 相对熵: KL散度
- P 为训练集数据分布; Q 为模型输出数据分布

```

1 # 计算交叉熵
2 nn.CrossEntropyLoss(weight=None, # 各类别loss设置的权值
3 ignore_index=-100, # 忽略某个类别
4 reduction='mean') # 计算模式, 如none/sum/mean
5 # none: 逐个元素计算
6 # sum: 所有元素求和, 返回标量
7 # mean: 加权平均, 返回标量

```

2.3 NLL/BCE/BCEWithLogits Loss

```

1 # 1. 实现负对数似然函数中的负号功能

```

```

2 nn.NLLLoss(weight=None,      # 各类别的loss设置权值
3           ignore_index=-100,  # 忽略某个类别
4           reduction='mean')   # 计算模式, 如none/sum/mean
5 # 2. 二分类交叉熵
6 # 注: 输入值为[0,1], 可配合sigmoid函数使用
7 nn.BCELoss(weight=None,      # 各类别loss设置的权值
8            ignore_index=-100, # 忽略某个类别
9            reduction='mean')  # 计算模式, 如none/sum/mean
10 # 3. 结合Sigmoid与二分类交叉熵
11 # 注: 不需要额外加入Sigmoid函数
12 nn.BCEWithLogitsLoss(weight=None,      # 各类别loss设置的权值
13                      ignore_index=-100, # 忽略某个类别
14                      reduction='mean',   # 计算模式, 如none/sum/mean
15                      pos_weight=None)    # 正样本的权值

```

2.4 其他损失函数

1. `nn.L1Loss`: $l_n = |x_n - y_n|$

2. `nn.MSELoss`: $l_n = (x_n - y_n)^2$

3. `nn.SmoothL1Loss`: $loss(x, y) = \frac{1}{n} \sum_{i=1}^n z_i$

其中, $z_i = \begin{cases} 0.5(x_i - y_i)^2, & \text{if } |x_i - y_i| < 1 \\ |x_i - y_i| - 0.5, & \text{otherwise} \end{cases}$

4. `nn.PoissonNLLLoss`

5. `nn.KLDivLoss`: $l_n = y_n \times (\log y_n - x_n)$

6. `nn.MarginRankingLoss`: $loss(x, y) = \max(0, -y \times (x_1 - x_2) + \text{margin})$

7. `nn.MultiLabelMarginLoss`: $loss(x, y) = \sum_{ij} \frac{\max(0, 1 - (x[y[j]] - x[i]))}{x.size(0)}$

8. `nn.SoftMarginLoss`: $loss(x, y) = \sum_i \frac{\log(1 + \exp(-y[i] * x[i]))}{x.nelement()}$

9. `nn.MultiLabelSoftMarginLoss`:

$loss(x, y) = -\frac{1}{x.nelement()} * \sum_i y[i] * \log((1 + \exp(-x[i]))^{-1}) + (1 - y[i]) * \log\left(\frac{\exp(-x[i])}{(1 + \exp(-x[i]))}\right)$

10. `nn.MultiMarginLoss`: $loss(x, y) = \frac{\sum_i \max(0, \text{margin} - x[y[i]] + x[i])^p}{x.size(0)}$

11. `nn.TripletMarginLoss`: $L(a, p, n) = \max\{d(a_i, p_i) - d(a_i, n_i) + \text{margin}, 0\}$ 其中,
 $d(x_i, y_i) = \|x_i - y_i\|_p$

12. `nn.HingeEmbeddingLoss`: $l_n = \begin{cases} x_n, & \text{if } y_n = 1 \\ \max\{0, \text{margin} - x_n\}, & \text{if } y_n = -1 \end{cases}$

13. `nn.CosineEmbeddingLoss`: $loss(x, y) = \begin{cases} 1 - \cos(x_1, x_2), & \text{if } y = 1 \\ \max(0, \cos(x_1, x_2) - \text{margin}), & \text{if } y = -1 \end{cases}$

14. `nn.CTCLoss` (新添加)

```

1 # 1. nn.L1Loss
2 # 计算inputs与target之差的绝对值
3 nn.L1Loss(reduction='mean')    # 计算模式, 如none/sum/mean
4 # 2. nn.MSELoss
5 # 计算inputs与target之差的平方
6 nn.MSELoss(reduction='mean')
7 # 3. SmoothL1Loss
8 # 平滑的L1Loss
9 nn.SmoothL1Loss(reduction='mean')

```

```

10 # 4. PoissonNLLLoss
11 # 泊松分布的负对数似然损失函数
12 # log_input = True: loss(input,target)=exp(input)-target*input
13 # log_input = False: loss(input,target)=input-target*log(input+eps)
14 nn.PoissonNLLLoss(log_input=True,    # 输入是否为对数形式
15                   full=False,       # 是否计算所有loss
16                   eps=1e-08,        # 修正项, 避免log为nan
17                   reduction='mean')
18 # 5. KLDivLoss
19 # 计算KL散度, 相对熵
20 # 注: 需提前输入计算log-probabilities, 如通过nn.logsoftmax()计算
21 # batchmean: 在batchsize维度求平均值
22 nn.KLDivLoss(reduction='mean')    # 计算模式增加batchmean
23 # 6. MarginRankingLoss
24 # 计算两个向量之间的相似度, 用于排序任务, 返回一个n*n的loss矩阵
25 # y=1时, 当x1>x2, 不产生loss; y=-1时, 当x2>x1, 不产生loss
26 nn.MarginRankingLoss(margin=0,     # 边界值, x1与x2之间的差异值
27                      reduction='mean')
28 # 7. MultiLabelMarginLoss
29 # 多标签边界损失函数
30 # 例: 四分类任务/样本属于0类和3类, 标签为[0,3,-1,-1]
31 nn.MultiLabelMarginLoss(reduction='mean')
32 # 8. SoftMarginLoss
33 # 计算二分类的logistic损失
34 nn.SoftMarginLoss(reduction='mean')
35 # 9. MultiLabelSoftMarginLoss
36 # SoftMarginLoss多标签版本, 标签为[1,0,0,1]
37 nn.MultiLabelSoftMarginLoss(weight=None,    # 各类别的loss设置权重
38                             reduction='mean')
39 # 10. MultiMarginLoss
40 # 计算多分类的折页损失
41 nn.MultiMarginLoss(p=1,             # 可选1或2
42                   margin=1.0,       # 边界值
43                   weight=None,      # 各类别的loss设置权重
44                   reduction='mean')
45 # 11. TripletMarginLoss
46 # 计算三元组损失, 人脸验证中常用
47 nn.TripletMarginLoss(margin=1.0,    # 边界值
48                     p=2,            # 范数的阶
49                     eps=1e-06,
50                     swap=False,
51                     reduction='mean')
52 # 12. HingeEmbeddingLoss
53 # 计算两个输入的相似性, 常用于非线性embedding和半监督学习
54 # 注: 输入x为两个输入之差的绝对值
55 nn.HingeEmbeddingLoss(margin=1.0,
56                      reduction='mean')
57 # 13. CosineEmbeddingLoss
58 # 采用余弦相似度计算两个输入的相似性
59 nn.CosineEmbeddingLoss(margin=0,    # 可取值[-1,1], 推荐[0,0.5]
60                       reduction='mean')
61 # 14. CTCLoss
62 # 计算CTC损失, 解决时序类数据的分类
63 nn.CTCLoss(blank=0,               # blank label
64            reduction='mean',      # 无穷大的值或梯度置0
65            zero_infinity=False)

```

2.5 作业

[第四周作业1](#)

3. 优化器

3.1 优化器的概念

- 优化器：管理并更新模型中可学习参数的值，使得模型输出更接近真实标签
- 导数：函数在指定坐标轴上的变化率
- 方向导数：函数在指定方向上的变化率
- 梯度：向量，方向为方向导数取得最大值的方向

3.2 优化器的属性

```
1 class Optimizer(object):
2     def __init__(self, params, defaults):
3         self.defaults = defaults    # 优化器超参数
4         self.state = defaultdict(dict)    # 参数的缓存，如momentum的缓存
5         # param_groups = [{'params': param_groups}]
6         self.param_groups = []    # 管理的参数组，包含字典元素的list
7         # _step_count: 记录更新次数，学习率调整中使用
8         ...
```

3.3 优化器的方法

- `zero_grad()`：清空管理参数的梯度 注：pytorch中张量梯度不自动清零
- `step()`：执行一步更新
- `add_param_group`：添加参数组
- `state_dict()`：获取优化器当前状态信息字典
- `load_state_dict()`：加载状态信息字典

4. 随机梯度下降

4.1 learning rate 学习率

- 控制更新的步伐

4.2 momentum 动量

- 结合当前梯度与上一次更新信息，用于当前更新
- 指数加权平均： $v_t = \beta \times v_{t-1} + (1 - \beta) \times \theta_t$

4.3 torch.optim.SGD

$$v_i = m \times v_{i-1} + g(w_i)$$

$$w_{i+1} = w_i - lr \times v_i$$

其中， w_{i+1} 为第 $i + 1$ 次更新的参数； lr 为学习率； v_i 为更新量； m 为momentum系数； $g(w_i)$ 为 w_i 的梯度

```
1 torch.optim.SGD(params,      # 管理的参数组
2     lr=<object object>,      # 初始学习率
3     momentum=0,              # 动量系数beta
4     dampening=0,             # L2正则化系数
5     weight_decay=0,
6     nesterov=False)          # 是否采用NAG
```

4.4 Pytorch的十种优化器

- `optim.SGD`：随机梯度下降法
- `optim.Adagrad`：自适应学习率梯度下降法
- `optim.RMSprop`：Adagrad的改进
- `optim.Adadelta`：Adagrad的改进
- `optim.Adam`：RMSprop结合Momentum
- `optim.Adamax`：Adam增加学习率上限
- `optim.SparseAdam`：稀疏版的Adam
- `optim.ASGD`：随机平均梯度下降
- `optim.Rprop`：弹性反向传播
- `optim.LBFGS`：BFGS的改进

4.5 作业

[第四周作业3](#)

注：作业2为整理损失函数笔记