人工智能实践：TensorFlow笔记

第二讲

神经网络优化

tf.random.normal
tf.where

简介

本讲目标：学会神经网络优化过程，使用正则化减少过拟合，使用优化器更新网络参数。

1 神经网络复杂度

1.1 时间复杂度

即模型的运算次数，可用浮点运算次数（FPLOPs, FLoating-point OPerations）或者乘加运算次数衡量.

1.2 空间复杂度

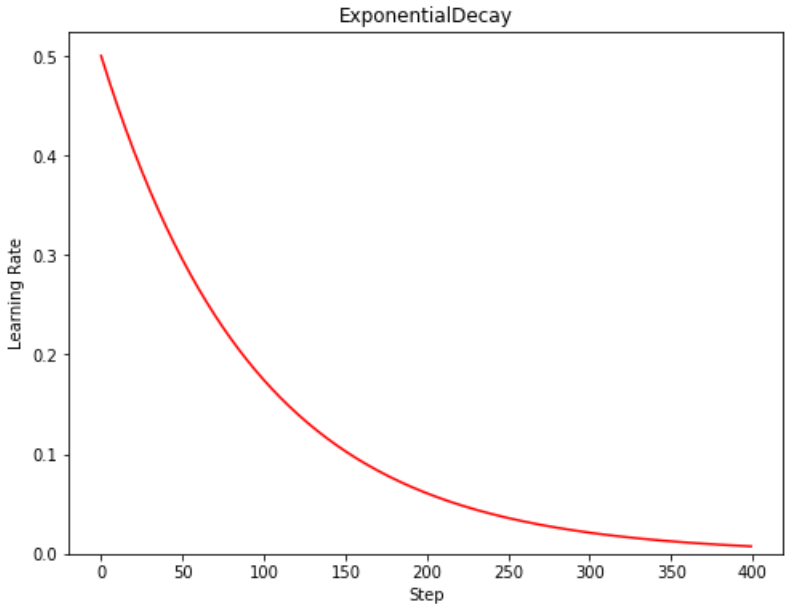
空间复杂度（访存量），严格来讲包括两部分：总参数量 + 各层输出特征图。

- **参数量**：模型所有带参数的层的权重参数总量；
- **特征图**：模型在实时运行过程中每层所计算出的输出特征图大小。

2 学习率策略

2.1 指数衰减

TensorFlow API: tf.keras.optimizers.schedules.ExponentialDecay



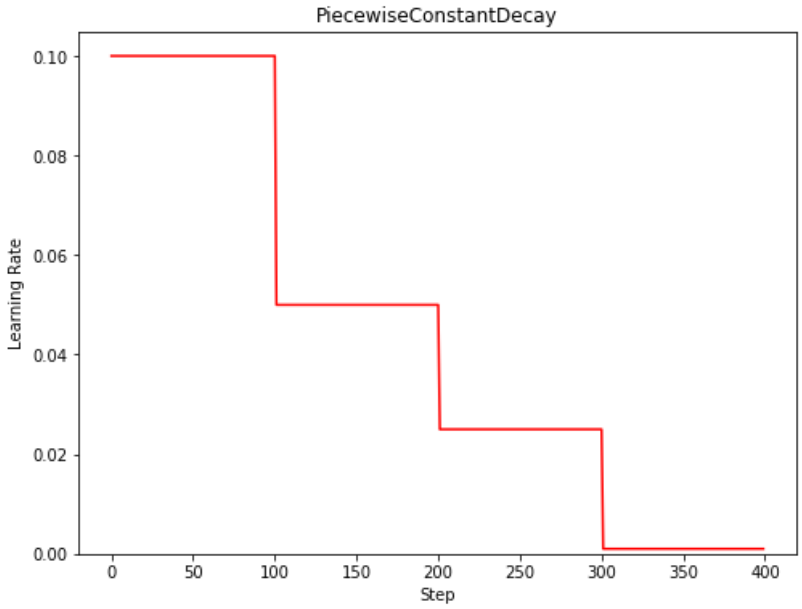
$$decayed\_learning\_rate = learning\_rate \cdot decay\_rate^{(global\_step/decay\_steps)}$$

其中，$learning\_rate$ 是初始学习率，$decay_rate$是衰减率，$global_step$表示从0到当前的训练次数，$decay_steps$用来控制衰减速度。

指数衰减学习率是先使用较大的学习率来快速得到一个较优的解，然后随着迭代的继续,逐步减小学习率，使得模型在训练后期 更加稳定。指数型学习率衰减法是最常用的衰减方法，在大量模型中都广泛使用。

2.2 分段常数衰减

TensorFlow API: tf.optimizers.schedules.PiecewiseConstantDecay



分段常数衰减可以让调试人员针对不同任务设置不同的学习率，进行**精细调参**，在任意步长后下降任意数值的learning rate，要求调试人员对模型和数据集有深刻认识.

3 激活函数

激活函数是用来加入非线性因素的，因为线性模型的表达能力不够。引入非线性激活函数，可使深层神经网络的表达能力更加强大。

优秀的激活函数应满足：

- **非线性：** 激活函数非线性时，多层神经网络可逼近所有函数
- **可微性：** 优化器大多用梯度下降更新参数
- **单调性：** 当激活函数是单调的，能保证单层网络的损失函数是凸函数
- **近似恒等性：** $f(x) \approx x$. 当参数初始化为随机小值时，神经网络更稳定

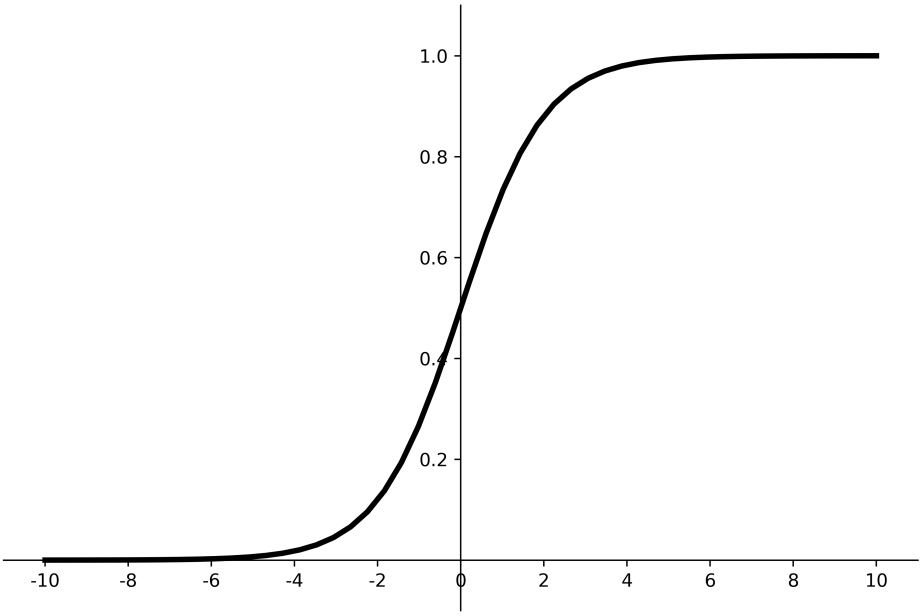
激活函数输出值的范围：

- 激活函数输出为有限值时，基于梯度的优化方法更稳定
- 激活函数输出为无限值时，建议调小学习率

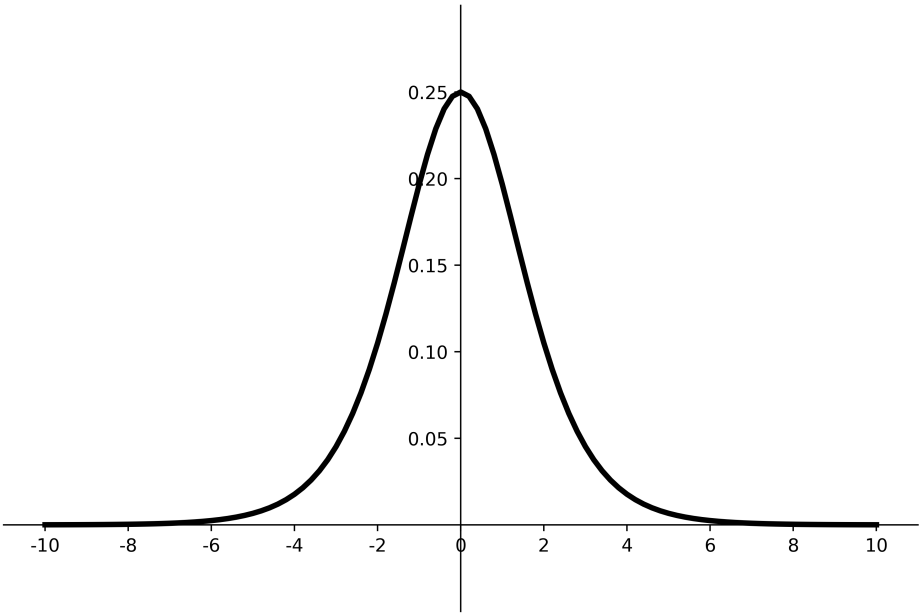
常见的激活函数有：sigmoid，tanh，ReLU，Leaky ReLU，PReLU，RReLU，ELU（Exponential Linear Units），softplus，softsign，softmax等，下面介绍几个典型的激活函数：

3.1 sigmoid

$$f(x) = \frac{1}{1 + e^{-x}}$$



函数图像



导数图像

TensorFlow API: tf.math.sigmoid

优点:

1. 输出映射在(0,1)之间，单调连续，输出范围有限，优化稳定，可用作输出层；
2. 求导容易。

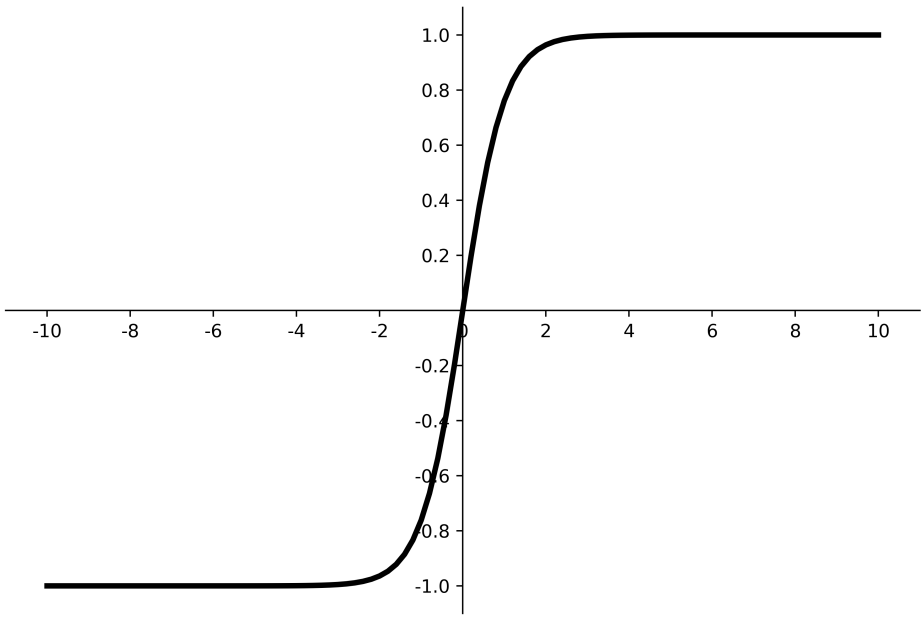
缺点:

1. 易造成梯度消失；
2. 输出非0均值，收敛慢；
3. 幂运算复杂，训练时间长。

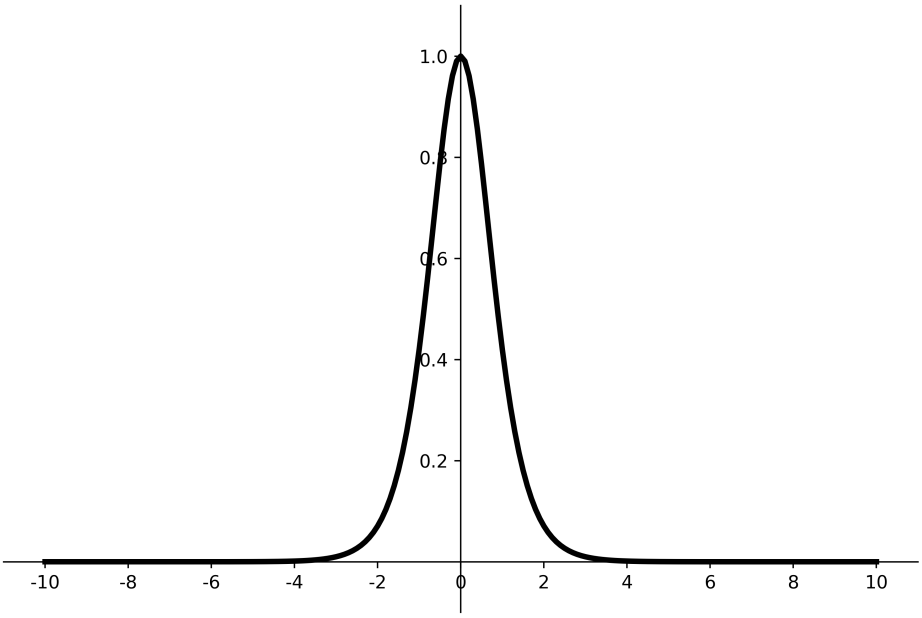
sigmoid函数可应用在训练过程中。然而，当处理分类问题作出输出时，sigmoid却无能为力。简单地说，sigmoid函数只能处理两个类，不适用于多分类问题。而softmax可以有效解决这个问题，并且softmax函数大都运用在神经网路中的最后一层网络中，使得值得区间在（0,1）之间，而不是二分类的。

3.2 tanh

$$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$



函数图像



导数图像

TensorFlow API: tf.math.tanh

优点:

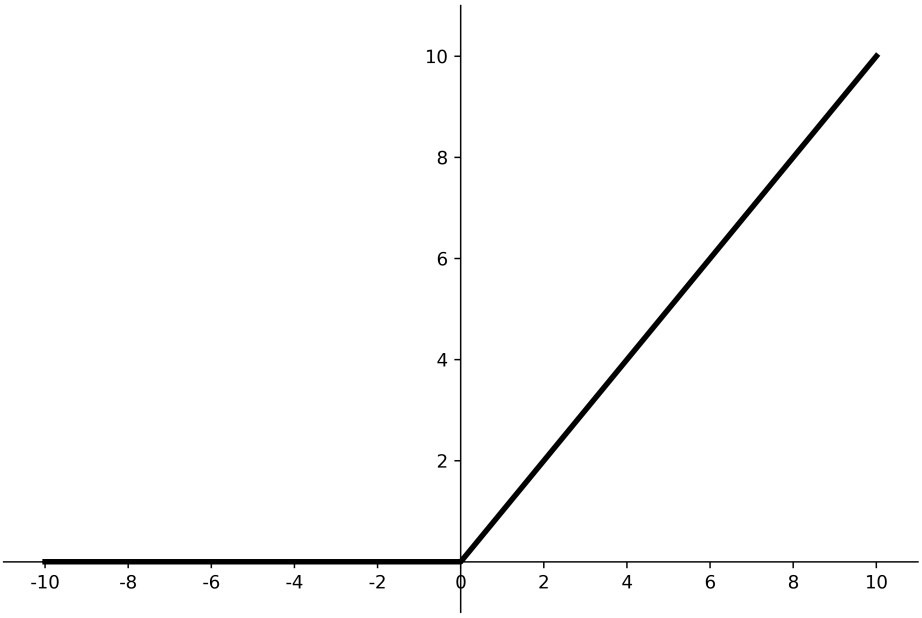
1. 比sigmoid函数收敛速度更快。
2. 相比sigmoid函数，其输出以0为中心。

缺点:

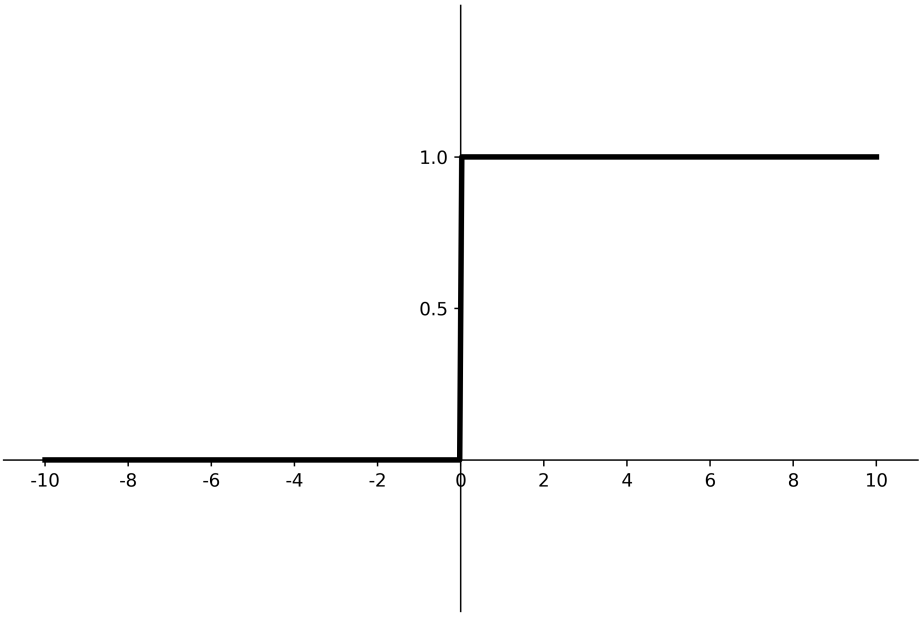
1. 易造成梯度消失；
2. 幂运算复杂，训练时间长。

3.3 ReLU

$$f(x) = max(0, x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$



函数图像



导数图像

TensorFlow API: tf.nn.relu

优点:

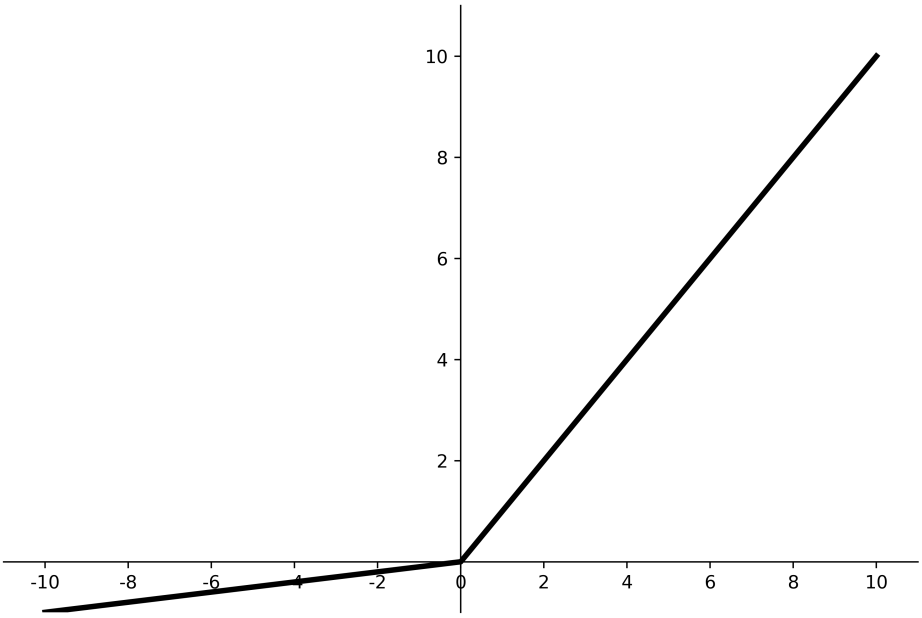
1. 解决了梯度消失问题(在正区间);
2. 只需判断输入是否大于0，计算速度快;
3. 收敛速度远快于sigmoid和tanh，因为sigmoid和tanh涉及很多expensive的操作;
4. 提供了神经网络的稀疏表达能力。

缺点:

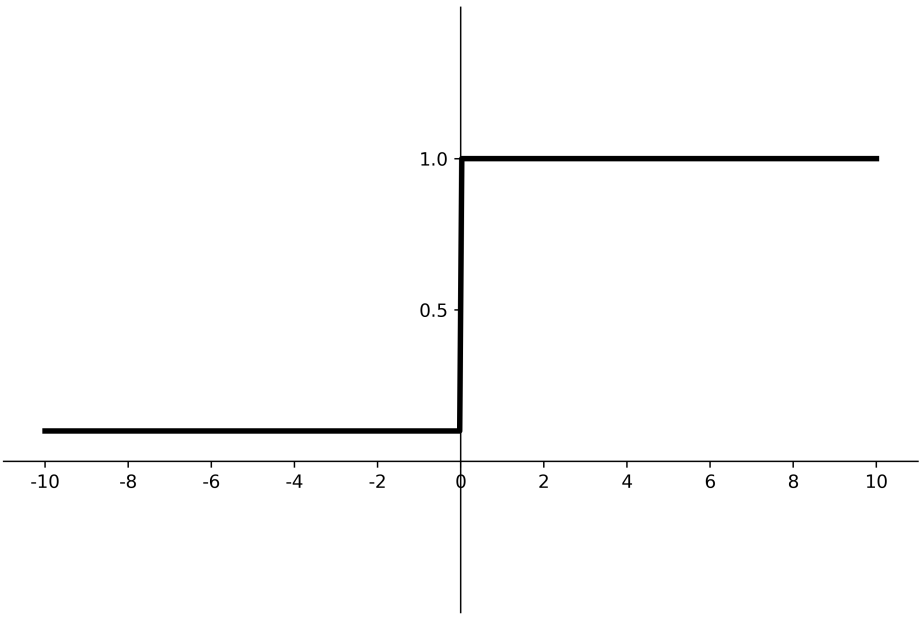
1. 输出非0均值，收敛慢；
2. Dead ReLU问题：某些神经元可能永远不会被激活，导致相应的参数永远不能被更新。

3.4 Leaky ReLU

$$f(x) = max(\alpha x, x)$$



函数图像



导数图像

TensorFlow API: tf.nn.leaky_relu

理论上来讲，Leaky ReLU有ReLU的所有优点，外加不会有Dead ReLU问题，但是在实际操作当中，并没有完全证明Leaky ReLU总是好于ReLU。

3.5 softmax

$$\sigma(\boldsymbol{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad for\ j = 1, \dots, K$$

TensorFlow API: tf.nn.softmax

对神经网络全连接层输出进行变换，使其服从概率分布，即每个值都位于[0,1]区间且和为1。

3.6 建议

对于初学者的建议：

1. 首选ReLU激活函数；
2. 学习率设置较小值；
3. 输入特征标准化，即让输入特征满足以0为均值，1为标准差的正态分布；
4. 初始化问题：初始参数中心化，即让随机生成的参数满足以0为均值，$\sqrt{\frac{2}{\text{当前层输入特征个数}}}$为标准差的正态分布。

4 损失函数

神经网络模型的效果及优化的目标是通过损失函数来定义的。回归和分类是监督学习中的两个大类。

4.1 均方误差损失函数

均方误差（Mean Square Error）是回归问题最常用的损失函数。回归问题解决的是对具体数值的预测，比如房价预测、销量预测等。这些问题需要预测的不是一个事先定义好的类别，而是一个任意实数。均方误差定义如下：

$$MSE(y, y^{'}) = \frac{\Sigma_{i=1}^{n}(y_i - y_i^{'})^2}{n}$$

其中 y_i 为一个batch中第i个数据的真实值，而 $y_i^{'}$ 为神经网络的预测值。

TensorFlow API: tf.keras.losses.MSE

4.2 交叉熵损失函数

交叉熵（Cross Entropy）表征两个概率分布之间的距离，交叉熵越小说明二者分布越接近，是分类问题中使用较广泛的损失函数。

$$H(y\_, y) = -\Sigma y\_ * lny$$

其中 $y\_$ 代表数据的真实值，y 代表神经网络的预测值。

对于多分类问题，神经网络的输出一般不是概率分布，因此需要引入softmax层，使得输出服从概率分布。TensorFlow中可计算交叉熵损失函数的API有：

TensorFlow API: tf.keras.losses.categorical_crossentropy

TensorFlow API: tf.nn.softmax_cross_entropy_with_logits

TensorFlow API: tf.nn.sparse_softmax_cross_entropy_with_logits

4.3 自定义损失函数

根据具体任务和目的，可设计不同的损失函数。从老师课件和讲解中对于酸奶预测损失函数的设计，我们可以得知损失函数的定义能极大影响模型预测效果。好的损失函数设计对于模型训练能够起到良好的引导作用。

例如，我们可以看目标检测中的多种损失函数。目标检测的主要功能是定位和识别，损失函数的功能主要就是让定位更精确，识别准确率更高。目标检测任务的损失函数由分类损失（Classificition Loss）和回归损失（Bounding Box Regeression Loss）两部分构成。近几年来回归损失主要有Smooth L1 Loss(2015), IoU Loss(2016 ACM), GIoU Loss(2019 CVPR), DIoU Loss & CIoU Loss(2020 AAAI)等，分类损失有交叉熵、softmax loss、logloss、focal loss等。在此由于篇幅原因不细究，有兴趣的同学可自行研究。主要是给大家一个感性的认知：需要针对特定的背景、具体的任务设计损失函数。

5 欠拟合与过拟合

欠拟合的解决方法：

- 增加输入特征项
- 增加网络参数
- 减少正则化参数

过拟合的解决方法：

- 数据清洗
- 增大训练集
- 采用正则化
- 增大正则化参数

6 优化器

优化算法可以分成一阶优化和二阶优化算法，其中一阶优化就是指的梯度算法及其变种，而二阶优化一般是用二阶导数（Hessian 矩阵）来计算，如牛顿法，由于需要计算Hessian阵和其逆矩阵，计算量较大，因此没有流行开来。这里主要总结一阶优化的各种梯度下降方法。

深度学习优化算法经历了SGD -> SGDM -> NAG ->AdaGrad -> AdaDelta -> Adam -> Nadam这样的发展历程。

> 定义：待优化参数 ω，损失函数 $f(\omega)$，初始学习率 α，每次迭代一个batch，t表示当前batch迭代的总次数。
>
> 1. 计算损失函数关于当前参数的梯度：$g_t = \nabla f(\omega_t) = \frac{\partial f}{\partial \omega_t}$
> 2. 根据历史梯度计算**一阶动量**和**二阶动量**：$m_t = \phi(g_1, g_2, \ldots, g_t)$ ，$V_t = \psi(g_1, g_2, \ldots, g_t)$
> 3. 计算当前时刻的下降梯度：$\eta_t = \alpha \cdot m_t / \sqrt{V_t}$
> 4. 根据下降梯度进行更新：$\omega_{t+1} = \omega_t - \eta_t$
>
> 步骤3，4对于各算法都是一致的，主要差别体现在步骤1和2上。

6.1 SGD

TensorFlow API: tf.keras.optimizers.SGD

6.1.1 vanilla SGD

最初版本的SGD没有动量的概念，

$$m_t = g_t \ , \ V_t = 1$$

梯度下降是最简单的：

$$\eta_t = \alpha \cdot g_t$$

vanilla SGD最大的缺点是下降速度慢，而且可能会在沟壑的两边持续震荡，停留在一个局部最优点。

代码实现：

```
# sgd
w1.assign_sub(learning_rate * grads[0])
b1.assign_sub(learning_rate * grads[1])
```

6.1.2 SGD with Momentum

动量法是一种使**梯度向量**向相关方向加速变化，抑制震荡，最终实现加速收敛的方法。(Momentum is a method that helps accelerate SGD in the right direction and dampens oscillations. It adds a fraction β of the update vector of the past time step to the current update vector. The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions.)

为了抑制SGD的震荡，SGDM认为梯度下降过程可以加入惯性。下坡的时候，如果发现是陡坡，那就利用惯性跑的快一些。SGDM全称是SGD with Momentum，在SGD基础上引入了一阶动量：

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$

一阶动量是各个时刻梯度方向的指数移动平均值，约等于最近$1/(1-\beta_1)$个时刻的梯度向量和的平均值。也就是说，t 时刻的下降方向，不仅由当前点的梯度方向决定，而且由此前累积的下降方向决定。β_1 的经验值为0.9，这就意味着下降方向主要偏向此前累积的下降方向，并略微偏向当前时刻的下降方向。

Image 2: SGD without momentum

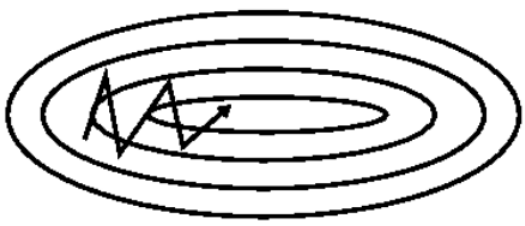


Image 3: SGD with momentum

代码实现：

```
# sgd-momentun
beta = 0.9
m_w = beta * m_w + (1 - beta) * grads[0]
m_b = beta * m_b + (1 - beta) * grads[1]
w1.assign_sub(learning_rate * m_w)
b1.assign_sub(learning_rate * m_b)
```

6.1.3 SGD with Nesterov Acceleration

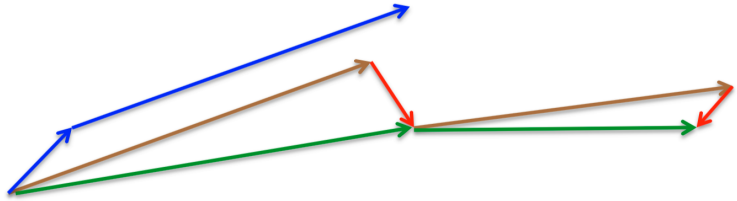
SGD 还有一个问题是会被困在一个局部最优点里。就像被一个小盆地周围的矮山挡住了视野，看不到更远的更深的沟壑。

NAG全称Nesterov Accelerated Gradient，是在SGD、SGDM的基础上的进一步改进，改进点在于步骤1。我们知道在时刻t的主要下降方向是由累积动量决定的，自己的梯度方向说了也不算，那与其看当前梯度方向，不如先看看如果跟着累积动量走了一步，那个时候再怎么走。因此，NAG在步骤1不计算当前位置的梯度方向，而是计算如果按照累积动量走了一步，考虑这个新地方的梯度方向。此时的梯度就变成了：

$$g_t = \nabla f(\omega_t - \alpha \cdot m_{t-1})$$

我们用这个梯度带入 SGDM 中计算 m_t 的式子里去，然后再计算当前时刻应有的梯度并更新这一次的参数。

其基本思路如下图（转自Hinton的Lecture slides）：



首先，按照原来的更新方向更新一步（棕色线），然后计算该新位置的梯度方向（红色线），然后用这个梯度方向修正最终的更新方向（绿色线）。上图中描述了两步的更新示意图，其中蓝色线是标准momentum更新路径。

6.2 AdaGrad

TensorFlow API: tf.keras.optimizers.Adagrad

上述SGD算法一直存在一个超参数（Hyper-parameter），即学习率。超参数是训练前需要手动选择的参数，前缀“hyper”就是用于区别训练过程中可自动更新的参数。学习率可以理解为参数 ω 沿着梯度 g 反方向变化的步长。

SGD对所有的参数使用统一的、固定的学习率，一个自然的想法是对每个参数设置不同的学习率，然而在大型网络中这是不切实际的。因此，为解决此问题，AdaGrad算法被提出，其做法是给学习率一个缩放比例，从而达到了自适应学习率的效果（Ada = Adaptive）。其思想是：对于频繁更新的参数，不希望被单个样本影响太大，我们给它们很小的学习率；对于偶尔出现的参数，希望能多得到一些

信息，我们给它较大的学习率。

那怎么样度量历史更新频率呢？为此引入二阶动量——该维度上，所有梯度值的平方和：

$$V_t = \sum_{\tau=1}^{t} g_\tau^2$$

回顾步骤 3 中的下降梯度：$\eta_t = \alpha \cdot m_t / \sqrt{V_t}$ 可视为 $\eta_t = \frac{\alpha}{\sqrt{V_t}} \cdot m_t$，即对学习率进行缩放。（一般为了防止分母为 0，会对二阶动量加一个平滑项，即$\eta_t = \alpha \cdot m_t / \sqrt{V_t + \varepsilon}$，$\varepsilon$是一个非常小的数。）

AdaGrad 在稀疏数据场景下表现最好。因为对于频繁出现的参数，学习率衰减得快；对于稀疏的参数，学习率衰减得更慢。然而在实际很多情况下，二阶动量呈单调递增，累计从训练开始的梯度，学习率会很快减至 0，导致参数不再更新，训练过程提前结束。

代码实现：

```
 # adagrad
v_w += tf.square(grads[0])
v_b += tf.square(grads[1])
w1.assign_sub(learning_rate * grads[0] / tf.sqrt(v_w))
b1.assign_sub(learning_rate * grads[1] / tf.sqrt(v_b))
```

6.3 RMSProp

TensorFlow API: tf.keras.optimizers.RMSprop

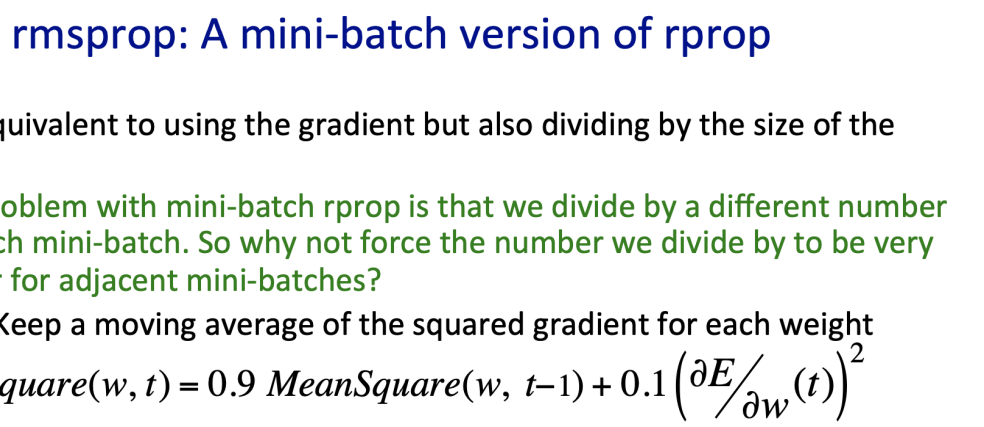
RMSProp算法的全称叫 Root Mean Square Prop，是由Geoffrey E. Hinton提出的一种优化算法（Hinton的课件见下图）。由于 AdaGrad 的学习率衰减太过激进，考虑改变二阶动量的计算策略：不累计全部梯度，只关注过去某一窗口内的梯度。修改的思路很直接，前面我们说过，指数移动平均值大约是过去一段时间的平均值，反映“局部的”参数信息，因此我们用这个方法来计算二阶累积动量：

$$m_t = g_t \qquad V_t = \beta_2 \cdot V_{t-1} + (1-\beta_2) \cdot g_t^2$$

$$\eta_t = \frac{\alpha}{\sqrt{V_t}} \cdot m_t = \frac{\alpha}{\sqrt{\beta_2 \cdot V_{t-1} + (1-\beta_2) \cdot g_t^2}} \cdot g_t$$

$$\omega_{t+1} = \omega_t - \eta_t$$

下图是来自Hinton的Lecture：



代码实现：

```
# RMSProp
beta = 0.9
v_w = beta * v_w + (1 - beta) * tf.square(grads[0])
v_b = beta * v_b + (1 - beta) * tf.square(grads[1])
w1.assign_sub(learning_rate * grads[0] / tf.sqrt(v_w))
b1.assign_sub(learning_rate * grads[1] / tf.sqrt(v_b))
```

6.4 AdaDelta

TensorFlow API: tf.keras.optimizers.Adadelta

为解决AdaGrad的学习率递减太快的问题，RMSProp和AdaDelta几乎同时独立被提出。

我们先看论文的AdaDelta算法，下图来自原论文：

Algorithm 1 Computing ADADELTA update at time t

Require: Decay rate ρ, Constant ϵ

Require: Initial parameter x_1

1: Initialize accumulation variables $E[g^2]_0 = 0$, $E[\Delta x^2]_0 = 0$

2: **for** $t = 1 : T$ **do** %% Loop over # of updates

3: Compute Gradient: g_t

4: Accumulate Gradient: $E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho)g_t^2$

5: Compute Update: $\Delta x_t = -\frac{\text{RMS}[\Delta x]_{t-1}}{\text{RMS}[g]_t} g_t$

6: Accumulate Updates: $E[\Delta x^2]_t = \rho E[\Delta x^2]_{t-1} + (1-\rho)\Delta x_t^2$

7: Apply Update: $x_{t+1} = x_t + \Delta x_t$

8: **end for**

对于上图算法的一点解释，$RMS[g]_t$ 是梯度 g 的均方根（Root Mean Square），$RMS[\Delta x]_{t-1}$ 是 Δx 的均方根：

$$RMS[g]_t = \sqrt{E[g^2]_t}$$

$$RMS[\Delta x]_{t-1} = \sqrt{E[\Delta x^2]_{t-1}}$$

我们可以看到AdaDelta与RMSprop仅仅是分子项不同，为了与前面公式保持一致，在此用 $\sqrt{U_t}$ 表示 η 的均方根：

$$m_t = g_t \qquad V_t = \beta_2 \cdot V_{t-1} + (1-\beta_2) \cdot g_t^2$$

$$\eta_t = \frac{\sqrt{U_{t-1}}}{\sqrt{V_t}} \cdot m_t = \frac{\sqrt{U_{t-1}}}{\sqrt{\beta_2 \cdot V_{t-1} + (1-\beta_2) \cdot g_t^2}} \cdot g_t$$

$$U_t = \beta_2 \cdot U_{t-1} + (1-\beta_2) \cdot \eta_t^2$$

$$\omega_{t+1} = \omega_t - \eta_t$$

代码实现：

```
# AdaDelta
beta = 0.999
v_w = beta * v_w + (1 - beta) * tf.square(grads[0])
v_b = beta * v_b + (1 - beta) * tf.square(grads[1])

delta_w = tf.sqrt(u_w) * grads[0] / tf.sqrt(v_w)
delta_b = tf.sqrt(u_b) * grads[1] / tf.sqrt(v_b)

u_w = beta * u_w + (1 - beta) * tf.square(delta_w)
u_b = beta * u_b + (1 - beta) * tf.square(delta_b)

w1.assign_sub(delta_w)
b1.assign_sub(delta_b)
```

6.5 Adam

TensorFlow API: tf.keras.optimizers.Adam

Adam名字来源是adaptive moment estimation。Our method is designed to combine the advantages of two recently popular methods: AdaGrad (Duchi et al., 2011), which works well with sparse gradients, and RMSProp (Tieleman & Hinton, 2012), which works well in on-line and non-stationary settings。也就是说，adam融合了Adagrad和RMSprop的思想。

谈到这里，Adam的出现就很自然而然了——它们是前述方法的集大成者。我们看到，SGDM在SGD基础上增加了一阶动量，AdaGrad、RMSProp和AdaDelta在SGD基础上增加了二阶动量。把一阶动量和二阶动量结合起来，再修正偏差，就是Adam了。

SGDM的一阶动量：

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$

加上RMSProp的二阶动量：

$$V_t = \beta_2 \cdot V_{t-1} + (1 - \beta_2) \cdot g_t^2$$

其中，参数经验值是 $\beta_1 = 0.9, \beta_2 = 0.999$ 。

一阶动量和二阶动量都是按照指数移动平均值进行计算的。初始化 $m_0 = 0, V_0 = 0$ ，在初期，迭代得到的 m_t, V_t 会接近于0。我们可以通过对 m_t, V_t 进行偏差修正来解决这一问题：

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{V}_t = \frac{V_t}{1 - \beta_2^t}$$

代码实现：

```
# adam
m_w = beta1 * m_w + (1 - beta1) * grads[0]
m_b = beta1 * m_b + (1 - beta1) * grads[1]
v_w = beta2 * v_w + (1 - beta2) * tf.square(grads[0])
v_b = beta2 * v_b + (1 - beta2) * tf.square(grads[1])

m_w_correction = m_w / (1 - tf.pow(beta1, int(global_step)))
m_b_correction = m_b / (1 - tf.pow(beta1, int(global_step)))
v_w_correction = v_w / (1 - tf.pow(beta2, int(global_step)))
v_b_correction = v_b / (1 - tf.pow(beta2, int(global_step)))

w1.assign_sub(learning_rate * m_w_correction / tf.sqrt(v_w_correction))
b1.assign_sub(learning_rate * m_b_correction / tf.sqrt(v_b_correction))
```

各优化器来源：

- SGD（1952）：https://projecteuclid.org/euclid.aoms/1177729392（源自回答）
- SGD with Momentum（1999）：https://www.sciencedirect.com/science/article/abs/pii/S0893608098001166
- SGD with Nesterov Acceleration（1983）：由Yurii Nesterov提出
- AdaGrad（2011）：http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf
- RMSProp（2012）：http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf
- AdaDelta（2012）：https://arxiv.org/abs/1212.5701
- Adam:（2014） https://arxiv.org/abs/1412.6980

（对上述算法非常好的可视化：https://imgur.com/a/Hqolp）

6.5 优化器选择

很难说某一个优化器在所有情况下都表现很好，我们需要根据具体任务选取优化器。一些优化器在计算机视觉任务表现很好，另一些在涉及RNN网络时表现很好，甚至在稀疏数据情况下表现更出色。

总结上述，基于原始SGD增加动量和Nesterov动量，RMSProp是针对AdaGrad学习率衰减过快的改进，它与AdaDelta非常相似，不同的一点在于AdaDelta采用参数更新的均方根（RMS）作为分子。Adam在RMSProp的基础上增加动量和偏差修正。如果数据是稀疏的，建议用自适用方法，即Adagrad, RMSprop, Adadelta, Adam。RMSprop, Adadelta, Adam 在很多情况下的效果是相似的。随着梯度变的稀疏，Adam 比 RMSprop 效果会好。总的来说，Adam整体上是最好的选择。

然而很多论文仅使用不带动量的vanilla SGD和简单的学习率衰减策略。SGD通常能够达到最小点，但是相对于其他优化器可能要采用更长的时间。采取合适的初始化方法和学习率策略，SGD更加可靠，但也有可能陷于鞍点和极小值点。因此，当在训练大型的、复杂的深度神经网络时，我们想要快速收敛，应采用自适应学习率策略的优化器。

如果是刚入门，优先考虑Adam或者SGD+Nesterov Momentum。

算法没有好坏，最适合数据的才是最好的，永远记住：No free lunch theorem。

6.6 优化算法的常用tricks

1. **首先，各大算法孰优孰劣并无定论。**如果是刚入门，优先考虑SGD+Nesterov Momentum或者Adam.（Standford 231n：*The two recommended updates to use are either SGD+Nesterov Momentum or Adam*）
2. **选择你熟悉的算法**——这样你可以更加熟练地利用你的经验进行调参。
3. **充分了解你的数据**——如果模型是非常稀疏的，那么优先考虑自适应学习率的算法。
4. **根据你的需求来选择**——在模型设计实验过程中，要快速验证新模型的效果，可以先用Adam进行快速实验优化；在模型上线或者结果发布前，可以用精调的SGD进行模型的极致优化。
5. **先用小数据集进行实验。**有论文研究指出，随机梯度下降算法的收敛速度和数据集的大小的关系不大。（*The mathematics of stochastic gradient descent are amazingly independent of the training set size. In particular, the asymptotic SGD convergence rates are independent from the sample size.*）因此可以先用一个具有代表性的小数据集进行实验，测试一下最好的优化算法，并通过参数搜索来寻找最优的训练参数。
6. **考虑不同算法的组合。**先用Adam进行快速下降，而后再换到SGD进行充分的调优。
7. **充分打乱数据集（shuffle）。**这样在使用自适应学习率算法的时候，可以避免某些特征集中出现，而导致的有时学习过度、有时学习不足，使得下降方向出现偏差的问题。在每一轮迭代后对训练数据打乱是一个不错的主意。
8. 训练过程中**持续监控训练数据和验证数据**上的目标函数值以及精度或者AUC等指标的变化情况。对训练数据的监控是要保证模型进行了充分的训练——下降方向正确，且学习率足够高；对验证数据的监控是为了避免出现过拟合。
9. **制定一个合适的学习率衰减策略。**可以使用分段常数衰减策略，比如每过多少个epoch就衰减一次；或者利用精度或者AUC等性能指标来监控，当测试集上的指标不变或者下跌时，就降低学习率。
10. Early stopping。如Geoff Hinton所说：“Early Stopping是美好的免费午餐”。你因此必须在训练的过程中时常在验证集上监测误差，在验证集上如果损失函数不再显著地降低，那么应该提前结束训练。
11. **算法参数的初始值选择。**初始值不同，获得的最小值也有可能不同，因此梯度下降求得的只是局部最小值；当然如果损失函数是凸函数则一定是最优解。由于有局部最优解的风险，需要多次用不同初始值运行算法，关键损失函数的最小值，选择损失函数最小化的初值。

参考链接

1. TensorFlow官网
2. 卷积神经网络的复杂度分析
3. 一个框架看懂优化算法之异同 SGD/AdaGrad/Adam
4. 梯度下降法家族
5. Some State of the Art Optimizers in Neural Networks
6. An overview of gradient descent optimization algorithms

附录：常用TensorFlow API及代码实现

学习率策略

tf.keras.optimizers.schedules.ExponentialDecay

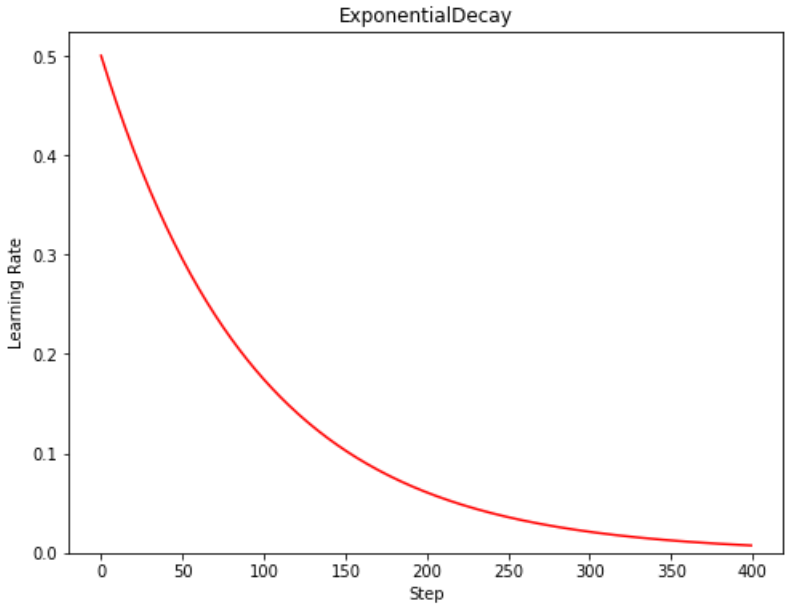
```
tf.keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate, decay_steps, decay_rate, staircase=False, name=None
)
```

功能：指数衰减学习率策略.
等价API：tf.optimizers.schedules.ExponentialDecay
参数：

- initial_learning_rate: 初始学习率.
- decay_steps: 衰减步数, staircase为True时有效.
- decay_rate: 衰减率.
- staircase: Bool型变量.如果为True, 学习率呈现阶梯型下降趋势.

返回：tf.keras.optimizers.schedules.ExponentialDecay(step)返回计算得到的学习率.
链接： tf.keras.optimizers.schedules.ExponentialDecay
例子：

```
N = 400
lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
    0.5,
    decay_steps=10,
    decay_rate=0.9,
    staircase=False)
y = []
for global_step in range(N):
    lr = lr_schedule(global_step)
    y.append(lr)
x = range(N)
plt.figure(figsize=(8,6))
plt.plot(x, y, 'r-')
plt.ylim([0,max(plt.ylim())])
plt.xlabel('Step')
plt.ylabel('Learning Rate')
plt.title('ExponentialDecay')
plt.show()
```



tf.keras.optimizers.schedules.PiecewiseConstantDecay

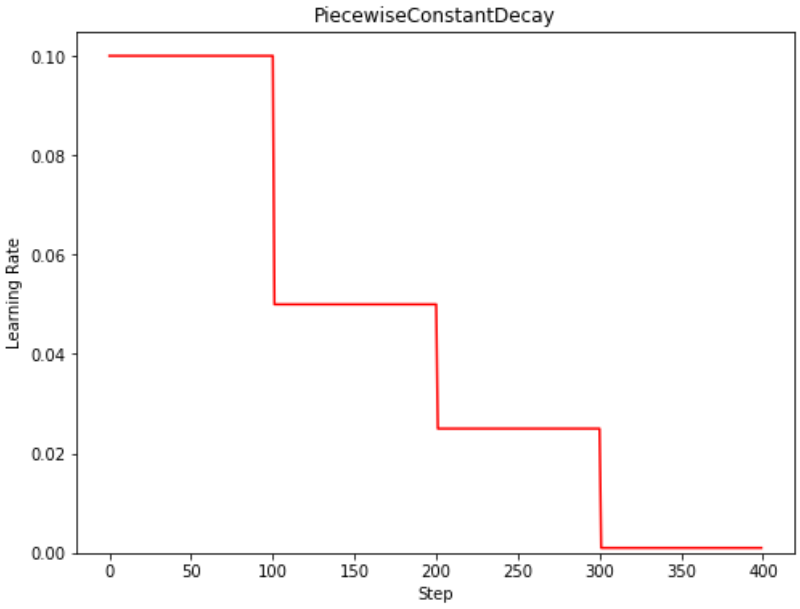
```
tf.keras.optimizers.schedules.PiecewiseConstantDecay(
    boundaries, values, name=None
)
```

功能：分段常数衰减学习率策略.
等价API：tf.optimizers.schedules.PiecewiseConstantDecay
参数：

- boundaries: [step_1, step_2, ..., step_n]定义了在第几步进行学习率衰减.
- values: [val_0, val_1, val_2, ..., val_n]定义了学习率的初始值和后续衰减时的具体取值.

返回：tf.keras.optimizers.schedules.PiecewiseConstantDecay(step)返回计算得到的学习率.
链接： tf.keras.optimizers.schedules.PiecewiseConstantDecay
例子：

```
N = 400
lr_schedule = tf.keras.optimizers.schedules.PiecewiseConstantDecay(
    boundaries=[100, 200, 300],
    values=[0.1, 0.05, 0.025, 0.001])
y = []
for global_step in range(N):
    lr = lr_schedule(global_step)
    y.append(lr)
x = range(N)
plt.figure(figsize=(8,6))
plt.plot(x, y, 'r-')
plt.ylim([0,max(plt.ylim())])
plt.xlabel('Step')
plt.ylabel('Learning Rate')
plt.title('PiecewiseConstantDecay')
plt.show()
```



激活函数

tf.math.sigmoid

```
tf.math.sigmoid(
    x, name=None
)
```

功能：计算x每一个元素的sigmoid值.
等价API：tf.nn.sigmoid, tf.sigmoid
参数：

- x: 张量x.

返回：与x shape相同的张量.
链接： tf.math.sigmoid
例子：

```
x = tf.constant([1., 2., 3.], )
print(tf.math.sigmoid(x))
```

```
>>> tf.Tensor([0.7310586  0.880797   0.95257413], shape=(3,), dtype=float32)
```

```
# 等价实现
print(1/(1+tf.math.exp(-x)))
```

```
>>> tf.Tensor([0.7310586  0.880797   0.95257413], shape=(3,), dtype=float32)
```

tf.math.tanh

```
tf.math.tanh(
    x, name=None
)
```

功能：计算x每一个元素的双曲正切值.
等价API：tf.nn.tanh, tf.tanh
参数：

- x: 张量x.

返回：与x shape相同的张量.
链接： tf.math.tanh
例子：

```
x = tf.constant([-float("inf"), -5, -0.5, 1, 1.2, 2, 3, float("inf")])
print(tf.math.tanh(x))
```

```
>>> tf.Tensor([-1. -0.99990916 -0.46211717 0.7615942 0.8336547 0.9640276
0.9950547 1.], shape=(8,), dtype=float32)
```

```
# 等价实现
print((tf.math.exp(x)-tf.math.exp(-x))/(tf.math.exp(x)+tf.math.exp(-x)))
```

```
>>> tf.Tensor([nan -0.9999091 -0.46211714 0.7615942 0.83365464 0.9640275
0.9950547 nan], shape=(8,), dtype=float32)
```

tf.nn.relu

```
tf.nn.relu(
    features, name=None
)
```

功能：计算修正线性值(rectified linear)： max(features, 0).
参数：

- features: 张量.

返回：与features shape相同的张量.
链接： tf.nn.relu
例子：

```
print(tf.nn.relu([-2., 0., -0., 3.]))
```

```
>>> tf.Tensor([0. 0. -0. 3.], shape=(4,), dtype=float32)
```

tf.nn.leaky_relu

```
tf.nn.leaky_relu(
    features, alpha=0.2, name=None
)
```

功能：计算Leaky ReLU值.
参数：

- features: 张量.
- alpha: x<0时的斜率值.

返回：与features shape相同的张量.
链接： tf.nn.leaky_relu
例子：

```
print(tf.nn.leaky_relu([-2., 0., -0., 3.]))
```

```
>>> tf.Tensor([-0.4 0. -0. 3.], shape=(4,), dtype=float32)
```

tf.nn.softmax

```
tf.nn.softmax(
    logits, axis=None, name=None
)
```

功能：计算softmax激活值.
等价API：tf.math.softmax
参数：

- logits: 张量.
- axis: 计算softmax所在的维度. 默认为-1，即最后一个维度.

返回：与logits shape相同的张量.
链接：tf.nn.softmax
例子：

```
logits = tf.constant([4., 5., 1.])
print(tf.nn.softmax(logits))
```

```
>>> tf.Tensor([0.26538792 0.7213992 0.01321289], shape=(3,), dtype=float32)
```

```
# 等价实现
print(tf.exp(logits) / tf.reduce_sum(tf.exp(logits)))
```

```
>>> tf.Tensor([0.26538792 0.72139925 0.01321289], shape=(3,), dtype=float32)
```

损失函数

tf.keras.losses.MSE

```
tf.keras.losses.MSE(
    y_true, y_pred
)
```

功能：计算y_true和y_pred的均方误差.
链接：tf.keras.losses.MSE
例子：

```
y_true = tf.constant([0.5, 0.8])
y_pred = tf.constant([1.0, 1.0])
print(tf.keras.losses.MSE(y_true, y_pred))
```

```
>>> tf.Tensor(0.145, shape=(), dtype=float32)
```

```
# 等价实现
print(tf.reduce_mean(tf.square(y_true - y_pred)))
```

```
>>> tf.Tensor(0.145, shape=(), dtype=float32)
```

tf.keras.losses.categorical_crossentropy

```
tf.keras.losses.categorical_crossentropy(
    y_true, y_pred, from_logits=False, label_smoothing=0
)
```

功能：计算交叉熵.
等价API：tf.losses.categorical_crossentropy
参数：

- y_true: 真实值.

- y_pred: 预测值.
- from_logits: y_pred是否为logits张量.
- label_smoothing: [0,1]之间的小数.

返回：交叉熵损失值.
链接： tf.keras.losses.categorical_crossentropy
例子：

```
y_true = [1, 0, 0]
y_pred1 = [0.5, 0.4, 0.1]
y_pred2 = [0.8, 0.1, 0.1]
print(tf.keras.losses.categorical_crossentropy(y_true, y_pred1))
print(tf.keras.losses.categorical_crossentropy(y_true, y_pred2))
```

```
>>> tf.Tensor(0.6931472, shape=(), dtype=float32)
tf.Tensor(0.22314353, shape=(), dtype=float32)
```

```
# 等价实现
print(-tf.reduce_sum(y_true * tf.math.log(y_pred1)))
print(-tf.reduce_sum(y_true * tf.math.log(y_pred2)))
```

```
>>> tf.Tensor(0.6931472, shape=(), dtype=float32)
tf.Tensor(0.22314353, shape=(), dtype=float32)
```

tf.nn.softmax_cross_entropy_with_logits

```
tf.nn.softmax_cross_entropy_with_logits(
    labels, logits, axis=-1, name=None
)
```

功能：logits经过softmax后，与labels进行交叉熵计算.

> 在机器学习中，对于多分类问题，把未经softmax归一化的向量值称为logits。logits经过softmax层后，输出服从概率分布的向量。（源自回答）

参数：

- labels: 在类别这一维度上，每个向量应服从有效的概率分布. 例如，在labels的shape为[batch_size, num_classes]的情况下，labels[i]应服从概率分布.
- logits: 每个类别的激活值，通常是线性层的输出. 激活值需要经过softmax归一化.
- axis: 类别所在维度，默认是-1，即最后一个维度.

返回：softmax交叉熵损失值.

链接： tf.nn.softmax_cross_entropy_with_logits
例子：

```
labels = [[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]]
logits = [[4.0, 2.0, 1.0], [0.0, 5.0, 1.0]]
print(tf.nn.softmax_cross_entropy_with_logits(labels=labels, logits=logits))
```

```
>>> tf.Tensor([0.16984604 0.02474492], shape=(2,), dtype=float32)
```

```
# 等价实现
print(-tf.reduce_sum(labels * tf.math.log(tf.nn.softmax(logits)), axis=1))
```

```
>>> tf.Tensor([0.16984606 0.02474495], shape=(2,), dtype=float32)
```

tf.nn.sparse_softmax_cross_entropy_with_logits

```
tf.nn.sparse_softmax_cross_entropy_with_logits(
    labels, logits, name=None
)
```

功能：labels经过one-hot编码，logits经过softmax，两者进行交叉熵计算. 通常labels的shape为[batch_size]，logits的shape为[batch_size, num_classes]. sparse可理解为对labels进行稀疏化处理(即进行one-hot编码).
参数：

- labels: 标签的索引值.
- logits: 每个类别的激活值，通常是线性层的输出. 激活值需要经过softmax归一化.

返回：softmax交叉熵损失值.
链接：[tf.nn.sparse_softmax_cross_entropy_with_logits]
例子：（下例中先对labels进行one-hot编码为[[1,0,0], [0,1,0]]，logits经过softmax变为[[0.844, 0.114，0.042], [0.007,0.976,0.018]]，两者再进行交叉熵运算）

```
labels = [0, 1]
logits = [[4.0, 2.0, 1.0], [0.0, 5.0, 1.0]]
print(tf.nn.sparse_softmax_cross_entropy_with_logits(labels1, logits))
```

```
>>> tf.Tensor([0.16984604 0.02474492], shape=(2,), dtype=float32)
```

```
# 等价实现
print(-tf.reduce_sum(tf.one_hot(labels, tf.shape(logits)[1]) *
tf.math.log(tf.nn.softmax(logits)), axis=1))
```

```
>>> tf.Tensor([0.16984606 0.02474495], shape=(2,), dtype=float32)
```

其它

tf.cast

```
tf.cast(
    x, dtype, name=None
)
```

功能：转换数据（张量）类型。
参数：

- x: 待转换的数据（张量）.
- dtype: 目标数据类型.

[tf.nn.sparse_softmax_cross_entropy_with_logits]: tf.nn.sparse_softmax_cross_entropy_with_logits

- name: 定义操作的名称（可选参数）.

返回：数据类型为dtype，shape与x相同的张量.

链接： tf.cast
例子：

```
x = tf.constant([1.8, 2.2], dtype=tf.float32)
print(tf.cast(x, tf.int32))
```

```
>>> tf.Tensor([1 2], shape=(2,), dtype=int32)
```

tf.random.normal

```
tf.random.normal(
    shape, mean=0.0, stddev=1.0, dtype=tf.dtypes.float32, seed=None, name=None
)
```

功能：生成服从正态分布的随机值。
参数：

- x: 一维张量.
- mean: 正态分布的均值.
- stddev: 正态分布的方差.

返回：满足指定shape并且服从正态分布的张量.

链接： tf.random.normal
例子：

```
tf.random.normal([3, 5])
```

```
>>> <tf.Tensor: id=7, shape=(3, 5), dtype=float32, numpy=
array([[-0.3951666 , -0.06858674,  0.29626969,  0.8070933 , -0.81376624],
       [ 0.09532423, -0.20840745,  0.37404788,  0.5459829 ,  0.17986278],
       [-1.0439969 , -0.8826001 ,  0.7081867 , -0.40955627, -2.6596873 ]],
      dtype=float32)>
```

tf.where

```
tf.where(
    condition, x=None, y=None, name=None
)
```

功能：根据condition，取x或y中的值。如果为True，对应位置取x的值；如果为False，对应位置取y的值。
参数：

- condition: bool型张量.
- x: 与y shape相同的张量.
- y: 与x shape相同的张量.

返回：shape与x相同的张量.

链接： tf.where
例子：

```
print(tf.where([True, False, True, False], [1,2,3,4], [5,6,7,8]))
```

```
>>> tf.Tensor([1 6 3 8], shape=(4,), dtype=int32)
```