

# DeepLearning

---

## DeepLearning

- AIC 和 BIC 准则

  - Notes

    - AIC 准则

    - BIC 准则

    - 比较

  - Links

- 交叉熵、相对熵、JS 散度

  - 熵

  - 相对熵 (KL 散度)

  - 交叉熵

  - JS 散度

  - Wasserstein 距离

- Optimizer in Deep Learning

  - 梯度下降法 (Gradient Descent)

    - 批量梯度下降法 (BGD)

    - 随机梯度下降法 (SGD)

    - 小批量梯度下降法 (Mini-batch Gradient Descent)

  - 动量优化法

    - Momentum

    - NAG (Nesterov Accelerated Gradient)

  - 自适应学习率

    - AdaGrad

    - Adadelta

    - RMSProp

    - Adam (Adaptive Moment Estimation)

  - Links

- 神经网络中的 Normalization 的发展历程

  - Local Response Normalization

  - Batch Normalization

  - Weight Normalization

  - Layer Normalization

  - Instance Normalization

  - Consine Normalization

  - Group Normalization

- 人工智能前言讲习

- Problems in Coding

## AIC 和 BIC 准则

---

### Notes

模型选择问题在 **模型复杂度** 与 **模型对数据集描述能力** 之间寻求最佳平衡;

## AIC 准则

赤池信息准则 (Akaike Information Criterion, AIC) , AIC 定义为:

$$AIC = 2k - 2\ln(L)$$

其中  $k$  为参数个数,  $L$  为似然函数。从一组可供选择的模型中选择最佳模型时, **通常选择 AIC 最小的模型**:

- 当两个模型之间存在较大差异时, 差异主要体现在似然函数项, 当似然函数差异不显著时, 上式第一项, 即模型复杂度则起作用, 从而**参数个数少**的模型是较好的选择;
- 一般而言, 当模型复杂度提高 ( $k$  增大) 时, 似然函数  $L$  也会增大, 从而使 AIC 变小, 但是  $k$  过大时, 似然函数增速减缓, 导致 AIC 增大, 模型过于复杂容易造成过拟合现象;
- 目标是选取AIC最小的模型, AIC不仅要提高模型拟合度 (极大似然), 而且引入了惩罚项, 使模型参数尽可能少, 有助于降低过拟合的可能性;

## BIC 准则

贝叶斯信息准则 (Bayesian Information Criterion, BIC) , BIC 定义为:

$$BIC = k\ln(n) - 2\ln(L)$$

其中,  $k$  为模型参数个数,  $n$  为样本数量,  $L$  为似然函数。从一组可供选择的模型中选择最佳模型时, **通常选择 BIC 最小的模型**;

## 比较

AIC 和 BIC 的公式中后半部分是一样的; 当  $n \geq 10^2$  时,  $k\ln(n) \geq 2k$ , 所以, BIC 相比 AIC 在**大数据量时对模型参数惩罚得更多**, 导致 BIC 更倾向于选择参数少的简单模型。所以还是**考虑使用 BIC 准则**;

## Links

- 参考链接: [AIC和BIC准则详解](#)

## 交叉熵、相对熵、JS 散度

### 熵

熵 (信息熵) 指的是信息量的期望;

$$H(X) = - \sum_{i=1}^n p(x_i) \log(p(x_i))$$

### 相对熵 (KL 散度)

相对熵 (KL 散度) 用来衡量两个分布的差异;

$$D_{KL}(p||q) = \sum_{i=1}^n p(x_i) \log\left(\frac{p(x_i)}{q(x_i)}\right)$$

相对熵是非对称的, 使用时  $p(x)$  用来表示样本的真实分布, 而  $q(x)$  用来表示模型所预测的分布;

## 交叉熵

交叉熵可以通过相对熵变化而来，在机器学习中通常直接用交叉熵作为损失函数；

$$H(p, q) = - \sum_{i=1}^n p(x_i) \log(q(x_i))$$

## JS 散度

JS 散度用来衡量两个分布的相似度，是基于相对熵的变体，解决了相对熵的非对称问题；

$$JS(P_1 || P_2) = \frac{1}{2} KL(P_1 || \frac{P_1 + P_2}{2}) + \frac{1}{2} KL(P_2 || \frac{P_1 + P_2}{2})$$

## Wasserstein 距离

Wasserstein 距离用来度量两个概率分布之间的距离，解决了当两个分布  $P, Q$  相差很远时，KL 散度和 JS 散度梯度消失的问题；

$$W(P_1, P_2) = \inf_{\gamma \in \Pi(P_1, P_2)} \mathbb{E}_{(x, y) \sim \gamma} [\|x - y\|]$$

# Optimizer in Deep Learning

## 梯度下降法 (Gradient Descent)

梯度下降法的计算过程就是沿梯度下降的方向求解极小值，也可以沿梯度上升方向求解最大值。使用梯度下降法更新参数：

$$\theta_{t+1} = \theta_t - \alpha \cdot \nabla_{\theta} J(\theta)$$

## 批量梯度下降法 (BGD)

在整个训练集上计算梯度，对参数进行更新：

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{1}{n} \cdot \sum_{i=1}^n \nabla_{\theta} J_i(\theta, x^i, y^i)$$

因为要计算整个数据集，收敛速度慢，但其优点在于更趋近于全局最优解；

## 随机梯度下降法 (SGD)

每次只随机选择一个样本计算梯度，对参数进行更新：

$$\theta_{t+1} = \theta_t - \alpha \cdot \nabla_{\theta} J_i(\theta, x^i, y^i)$$

训练速度快，但是容易陷入局部最优点，导致梯度下降的波动非常大；

## 小批量梯度下降法 (Mini-batch Gradient Descent)

每次随机选择  $n$  个样本计算梯度，对参数进行更新：

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{1}{m} \cdot \sum_{i=x}^{i=x+m-1} \nabla_{\theta} J_i(\theta, x^i, y^i)$$

这种方法是 BGD 和 SGD 的折衷；

## 动量优化法

### Momentum

参数更新时在一定程度上保留之前更新的方向，同时又利用当前 batch 的梯度微调最终的更新方向。在 SGD 的基础上增加动量，则参数更新公式如下：

$$\begin{aligned}m_{t+1} &= \mu \cdot m_t + \alpha \cdot \nabla_{\theta} J(\theta) \\ \theta_{t+1} &= \theta_t - m_{t+1}\end{aligned}$$

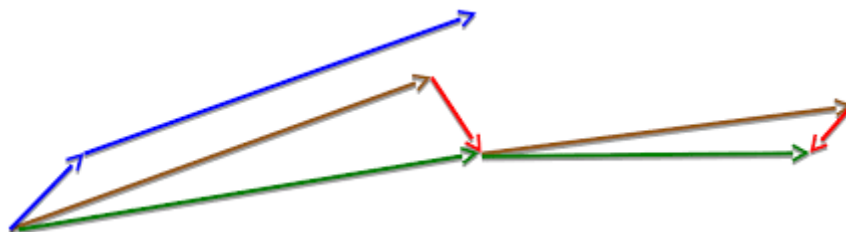
在梯度方向发生改变时，Momentum 能够降低参数更新速度，从而减少震荡；在梯度方向相同时，Momentum 可以加速参数更新，从而加速收敛。

### NAG (Nesterov Accelerated Gradient)

与 Momentum 不同的是，NAG 是在更新梯度是做一个矫正，即提前往前探一步，并对梯度作修改。参数更新公式如下：

$$\begin{aligned}m_{t+1} &= \mu \cdot m_t + \alpha \cdot \nabla_{\theta} J(\theta - \mu \cdot m_t) \\ \theta_{t+1} &= \theta_t - m_{t+1}\end{aligned}$$

两者的对比：蓝色为 Momentum，剩下的是 NAG；



## 自适应学习率

### AdaGrad

AdaGrad 算法期望在模型训练时有一个较大的学习率，而随着训练的不断增多，学习率也跟着下降。参数更新公式如下：

$$\begin{aligned}g &\leftarrow \nabla_{\theta} J(\theta) \\ r &\leftarrow r + g^2 \\ \Delta\theta &\leftarrow \frac{\delta}{\sqrt{r + \epsilon}} \cdot g \\ \theta &\leftarrow \theta - \Delta\theta\end{aligned}$$

学习率随着 **梯度的平方和** ( $r$ ) 的增加而减少。缺点：

- 需要手动设置学习率  $\delta$ ，如果  $\delta$  过大，会使得正则化项  $\frac{\delta}{\sqrt{r + \epsilon}}$  对梯度的调节过大；
- 中后期，参数的更新量会趋近于 0，使得模型无法学习；

### Adadelta

Adadelta 算法将 梯度的平方和 改为 **梯度平方的加权平均值**。参数更新公式如下：

$$\begin{aligned}
g_t &\leftarrow \nabla_{\theta} J(\theta) \\
E[g^2]_t &\leftarrow \gamma \cdot E[g^2]_{t-1} + (1 - \gamma) \cdot g_t^2 \\
\Delta \theta_t &\leftarrow -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t \\
\theta_{t+1} &\leftarrow \theta_t + \Delta \theta_t
\end{aligned}$$

上式中仍存在一个需要自己设置的全局学习率  $\eta$ ，可以通过下式消除全局学习率的影响：

$$\begin{aligned}
E[\Delta \theta^2]_t &\leftarrow \gamma \cdot E[\Delta \theta^2]_{t-1} + (1 - \gamma) \cdot \Delta \theta_t^2 \\
\Delta \theta_t &\leftarrow -\frac{\sqrt{E[\Delta \theta^2]_{t-1} + \epsilon}}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t \\
\theta_{t+1} &\leftarrow \theta_t + \Delta \theta_t
\end{aligned}$$

## RMSProp

RMSProp 算法是 AdaGrad 算法的改进，修改 梯度平方和 为 **梯度平方的指数加权移动平均**，解决了学习率急剧下降的问题。参数更新公式如下：

$$\begin{aligned}
g_t &\leftarrow \nabla_{\theta} J(\theta) \\
E[g^2]_t &\leftarrow \rho \cdot E[g^2]_{t-1} + (1 - \rho) \cdot g_t^2 \\
\Delta \theta &\leftarrow -\frac{\delta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g \\
\theta &\leftarrow \theta - \Delta \theta
\end{aligned}$$

## Adam (Adaptive Moment Estimation)

Adam 算法在动量的基础上，结合了偏置修正。参数更新公式如下：

$$\begin{aligned}
g_t &\leftarrow \nabla_{\theta} J(\theta) \\
m_t &\leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \\
v_t &\leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \\
\hat{m}_t &\leftarrow \frac{m_t}{1 - (\beta_1)^t} \\
\hat{v}_t &\leftarrow \frac{v_t}{1 - (\beta_2)^t} \\
\theta_{t+1} &= \theta_t - \frac{\delta}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t
\end{aligned}$$

论文伪代码：

**Algorithm 1:** *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation.  $g_t^2$  indicates the elementwise square  $g_t \odot g_t$ . Good default settings for the tested machine learning problems are  $\alpha = 0.001$ ,  $\beta_1 = 0.9$ ,  $\beta_2 = 0.999$  and  $\epsilon = 10^{-8}$ . All operations on vectors are element-wise. With  $\beta_1^t$  and  $\beta_2^t$  we denote  $\beta_1$  and  $\beta_2$  to the power  $t$ .

**Require:**  $\alpha$ : Stepsize

**Require:**  $\beta_1, \beta_2 \in [0, 1)$ : Exponential decay rates for the moment estimates

**Require:**  $f(\theta)$ : Stochastic objective function with parameters  $\theta$

**Require:**  $\theta_0$ : Initial parameter vector

$m_0 \leftarrow 0$  (Initialize 1<sup>st</sup> moment vector)

$v_0 \leftarrow 0$  (Initialize 2<sup>nd</sup> moment vector)

$t \leftarrow 0$  (Initialize timestep)

**while**  $\theta_t$  not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$  (Get gradients w.r.t. stochastic objective at timestep  $t$ )

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$  (Update biased first moment estimate)

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$  (Update biased second raw moment estimate)

$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$  (Compute bias-corrected first moment estimate)

$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$  (Compute bias-corrected second raw moment estimate)

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$  (Update parameters)

**end while**

**return**  $\theta_t$  (Resulting parameters)

## Links

- 参考链接: [优化算法 Optimizer 比较和总结](#)
- Adam 论文链接: [Kingma D P, Ba J. Adam: A method for stochastic optimization\[J\]. arXiv preprint arXiv:1412.6980, 2014.](#)

## 神经网络中的 Normalization 的发展历程

参考链接: [\[笔记\] 神经网络中 Normalization 的发展历程](#)

### Local Response Normalization

(1) 重要思想: 借鉴“侧抑制” (Lateral Inhibition) 的思想实现**局部神经元抑制**, 即使得局部的神经元产生竞争机制, 使其中响应值大的变得更大, 响应值小的变得更小 (这一点我从数值上并没有看出来); 在运用 Local Response Normalization 过程中, 我觉得需要注意的是它是对 **FeatureMap** 的一个正则化操作

(2) 公式:

$$b_{x,y}^i = a_{x,y}^i / \left( k + \alpha \sum_{j=\max(0, i-n/2)}^{\min(N-1, i+n/2)} (a_{x,y}^j)^2 \right)^{\beta}$$

其中,  $i, j$  表示 FeatureMap 的索引,  $a_{(x,y)}^j$  表示第  $j$  个 FeatureMap 中位于  $(x, y)$  处的响应值; 常用的值为:  $k = 2, n = 5, \alpha = 10^{-4}, \beta = 0.75$ 。

(3) 2015 年的 VGG 中发现 Local Response Normalization 的方法并没有什么作用, 故后来很少再被采用;

(4) 参考链接:

- [深度学习饱受争议的局部响应归一化\(LRN\)详解](#)

## Batch Normalization

(1) 主要思想：使用 Batch Normalization 层能够**使网络的训练更加稳定，加速神经网络的收敛速度**（当网络训练的收敛速度慢时可以考虑使用 BN 层），使得神经网络**对于初始化的敏感度下降**，并且具**有一定的正则化效果**；知乎大佬说 BN 层的优势是可以**防止“梯度弥散”**（我感觉不出来）；

(2) 公式：

<b>Input:</b> Values of $x$ over a mini-batch: $\mathcal{B} = \{x_1 \dots x_m\}$ ;	
Parameters to be learned: $\gamma, \beta$	
<b>Output:</b> $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$	
$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i$	// mini-batch mean
$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2$	// mini-batch variance
$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$	// normalize
$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i)$	// scale and shift

**Algorithm 1:** Batch Normalizing Transform, applied to activation  $x$  over a mini-batch.

其中， $\gamma$  和  $\beta$  两个仿射参数，是为了经过 BN 层处理后的数据仍**可以**恢复到之前的分布，从而**提升**了网络结构的 **Capacity**，即在做出一定改变的同时，仍保留之前的能力。注意，上面用的是“**可以**”这个词，具体“**是不是**”，还是要看模型训练的结果，训练的过程即**正常的求导梯度下降法**，公式如下：

$$\begin{aligned} \frac{\partial \ell}{\partial \hat{x}_i} &= \frac{\partial \ell}{\partial y_i} \cdot \gamma \\ \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} &= \sum_{i=1}^m \frac{\partial \ell}{\partial \hat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2} (\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2} \\ \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} &= \sum_{i=1}^m \frac{\partial \ell}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \\ \frac{\partial \ell}{\partial x_i} &= \frac{\partial \ell}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m} \\ \frac{\partial \ell}{\partial \gamma} &= \sum_{i=1}^m \frac{\partial \ell}{\partial y_i} \cdot \hat{x}_i \\ \frac{\partial \ell}{\partial \beta} &= \sum_{i=1}^m \frac{\partial \ell}{\partial y_i} \end{aligned}$$

(3) 代码实现：

- 使用注意点：
  - `batch_size` 应该尽可能设置得大一些；
  - 建议将bn层放在卷积层和激活层（例如 `ReLU`）之间，且卷积层不要使用偏置 bias；

- [PyTorch](#):

## BATCHNORM2D

```
CLASS torch.nn.BatchNorm2d(num_features, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

[SOURCE]

Applies Batch Normalization over a 4D input (a mini-batch of 2D inputs with additional channel dimension) as described in the paper [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](#).

$$y = \frac{x - E[x]}{\sqrt{\text{Var}[x] + \epsilon}} * \gamma + \beta$$

The mean and standard-deviation are calculated per-dimension over the mini-batches and  $\gamma$  and  $\beta$  are learnable parameter vectors of size C (where C is the input size). By default, the elements of  $\gamma$  are set to 1 and the elements of  $\beta$  are set to 0. The standard-deviation is calculated via the biased estimator, equivalent to `torch.var(input, unbiased=False)`.

Also by default, during training this layer keeps running estimates of its computed mean and variance, which are then used for normalization during evaluation. The running estimates are kept with a default `momentum` of 0.1.

If `track_running_stats` is set to `False`, this layer then does not keep running estimates, and batch statistics are instead used during evaluation time as well.

### • NOTE

This `momentum` argument is different from one used in optimizer classes and the conventional notion of momentum. Mathematically, the update rule for running statistics here is  $\hat{x}_{\text{new}} = (1 - \text{momentum}) \times \hat{x} + \text{momentum} \times x_t$ , where  $\hat{x}$  is the estimated statistic and  $x_t$  is the new observed value.

- 可以设置 `affine=False`，则不再设置后面的仿射参数；
  - 当设置（默认值）`track_running_state=True` 时，模型会使用动量法更新**两个全局的动态统计量** `running_mean` 和 `running_var` 来归一化输入的 `mini_batch`；如何设置 `track_running_state=False` 时，则模型直接使用 `mini_batch` 作为统计量；（所以在测试的时候，最好就是将这个设置为 `False`）
  - 在模型预测阶段，我们需要设置 BN 层 `training=False`，即直接使用 `model.eval()`；
- [Tensorflow](#)：（这里以 Tensorflow 2 为主，具体分析的话，可以发现它和 Tensorflow 1 的很多形为存在不同之处）

## tf.keras.layers.BatchNormalization

✓ See Stable

See Nightly



TensorFlow 1 version



View source on GitHub

Layer that normalizes its inputs.

Inherits From: [Layer](#), [Module](#)

```
tf.keras.layers.BatchNormalization(
    axis=-1, momentum=0.99, epsilon=0.001, center=True, scale=True,
    beta_initializer='zeros', gamma_initializer='ones',
    moving_mean_initializer='zeros',
    moving_variance_initializer='ones', beta_regularizer=None,
    gamma_regularizer=None, beta_constraint=None, gamma_constraint=None,
    renorm=False, renorm_clipping=None, renorm_momentum=0.99, fused=None,
    trainable=True, virtual_batch_size=None, adjustment=None, name=None, **kwargs
)
```



Batch normalization applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1.

Importantly, batch normalization works differently during training and during inference.

**During training** (i.e. when using `fit()` or when calling the layer/model with the argument `training=True`), the layer normalizes its output using the mean and standard deviation of the current batch of inputs. That is to say, for each channel being normalized, the layer returns  $(\text{batch} - \text{mean}(\text{batch})) / (\text{var}(\text{batch}) + \text{epsilon}) * \text{gamma} + \text{beta}$ , where:

- `epsilon` is small constant (configurable as part of the constructor arguments)
- `gamma` is a learned scaling factor (initialized as 1), which can be disabled by passing `scale=False` to the constructor.
- `beta` is a learned offset factor (initialized as 0), which can be disabled by passing `center=False` to the constructor.

**During inference** (i.e. when using `evaluate()` or `predict()` or when calling the layer/model with the argument `training=False` (which is the default), the layer normalizes its output using a moving average of the mean and standard deviation of the batches it has seen during training. That is to say, it returns  $(\text{batch} - \text{self.moving\_mean}) / (\text{self.moving\_var} + \text{epsilon}) * \text{gamma} + \text{beta}$ .

`self.moving_mean` and `self.moving_var` are non-trainable variables that are updated each time the layer is called in training mode, as such:

- `moving_mean = moving_mean * momentum + mean(batch) * (1 - momentum)`
- `moving_var = moving_var * momentum + var(batch) * (1 - momentum)`

- 可以看到，Tensorflow 在设计上和 pytorch 有所不同，tensorflow 直接将 BN 层设计成了 `training` 模式和 `inference` 模式；在 `training` 模式下，模型会不断更新 `moving_mean` 和 `moving_var` 这两个变量，`mini_batch` 的统计量直接由输入得到；而在 `inference` 模式下，模型则固定 `moving_mean` 和 `moving_var` 两个变量作为 `mini_batch` 的统计量（这可能就是基于了同分布的假设）；至于这个模式的变换，通过在调用 BN 层时设置一个 `training` 参数实现，下面具体看一下他的文档

Call arguments:

- `inputs`: Input tensor (of any rank).
- `training`: Python boolean indicating whether the layer should behave in training mode or in inference mode.
  - `training=True`: The layer will normalize its inputs using the mean and variance of the current batch of inputs.
  - `training=False`: The layer will normalize its inputs using the mean and variance of its moving statistics, learned during training.

Input shape: Arbitrary. Use the keyword argument `input_shape` (tuple of integers, does not include the samples axis) when using this layer as the first layer in a model.

Output shape: Same shape as input.

**About setting `layer.trainable = False` on a `BatchNormalization` layer:**

The meaning of setting `layer.trainable = False` is to freeze the layer, i.e. its internal state will not change during training: its `trainable` weights will not be updated during `fit()` or `train_on_batch()`, and its state updates will not be run.

Usually, this does not necessarily mean that the layer is run in inference mode (which is normally controlled by the `training` argument that can be passed when calling a layer). "Frozen state" and "inference mode" are two separate concepts.

Usually, this does not necessarily mean that the layer is run in inference mode (which is normally controlled by the `training` argument that can be passed when calling a layer). "Frozen state" and "inference mode" are two separate concepts.

However, in the case of the `BatchNormalization` layer, setting `trainable = False` on the layer means that the layer will be subsequently run in inference mode (meaning that it will use the moving mean and the moving variance to normalize the current batch, rather than using the mean and variance of the current batch).

This behavior has been introduced in TensorFlow 2.0, in order to enable `layer.trainable = False` to produce the most commonly expected behavior in the convnet fine-tuning use case.

Note that:

- This behavior only occurs as of TensorFlow 2.0. In 1.\*, setting `layer.trainable = False` would freeze the layer but would not switch it to inference mode.
- Setting `trainable` on an model containing other layers will recursively set the `trainable` value of all inner layers.
- If the value of the `trainable` attribute is changed after calling `compile()` on a model, the new value doesn't take effect for this model until `compile()` is called again.

- BN 层含有两个比较关键的参数，`trainable` 用来表示 BN 层是否可训练，控制的是上面的仿射参数的状态；`training` 参数则用来控制 `inference` 状态；另外，当设置 `trainable=False` 时，tensorflow 2 中的 BN 层将自动进入 `inference` 状态（这个和 tensorflow 1 不同）；
- 在 `model.compile()` 之后修改 BN 层的 `trainable` 属性，需要重新调用 `model.compile()` 才会起作用；

(4) 参考链接：

- [深度学习中 Batch Normalization 为什么效果好？](#)
- [Batch Normalization 详解以及 pytorch 实验](#)

## Weight Normalization

(1) 主要思想：相比于 BN 层，WN 层并不是对输入的特征数据进行归一化操作，而是**对神经网络中指定的层的参数做归一化操作**；WN 层摆脱了对于 Batch 的依赖，这意味着 WN 层完全可以用在 RNN 网络中，以及对于噪声敏感的任务；WN 层的计算成本低，可以**减少模型的运行时间**；

(2) 公式：将参数  $w$  拆分成方向和长度两个部分

$$y = \Phi(w * x + b) \Rightarrow y = \Phi(g * \frac{v}{||v||} * x + b)$$

我们把公式转换一下，就可以发现 **BN 层其实等同于使用参数的统计值对数据进行归一化**，

$$\Rightarrow y = \Phi(gv * \frac{v}{||v||} + b)$$

参数的学习，直接通过梯度下降的方法即可

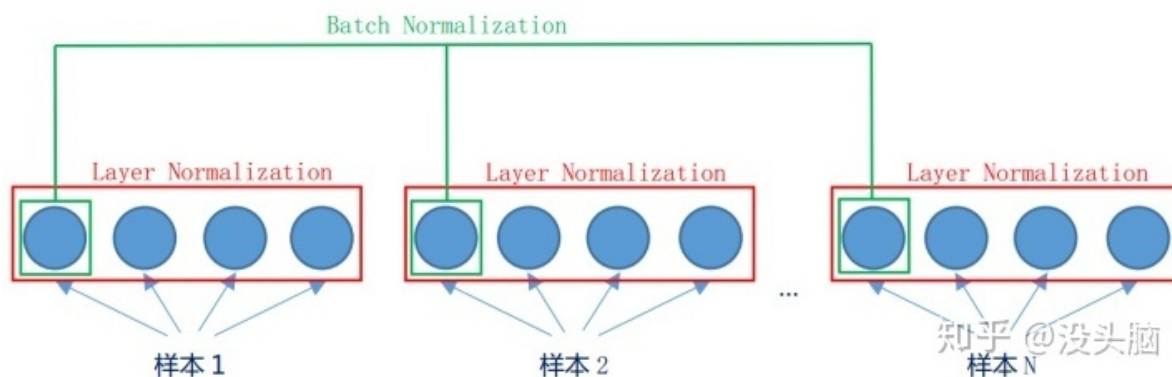
$$\nabla_g L = \frac{\nabla_w L \cdot v}{||v||} \quad \nabla_v L = \frac{g}{||v||} \nabla_w L - \frac{g \nabla_g L}{||v||^2} v$$

(3) 参考链接：

- [模型优化之Weight Normalization](#)
- [详解深度学习中的Normalization，BN/LN/WN](#)

## Layer Normalization

(1) 主要思想: LN 层和 BN 层非常相似, 不同之处在于, BN 层是对一个 Batch 中的**所有样本的不同维度**做 Normalization, 而 LN 是对**单个样本的所有维度**做 Normalization; LN 层是为了解决 BN 层对 **batch 数据和内存的依赖**, 并**减少 normalization 的时间**;



(2) 公式: (注意, 这里的统计量是一个标量)

$$\mu^l = \frac{1}{H} \sum_{i=1}^H a_i^l, \quad \sigma^l = \sqrt{\frac{1}{H} \sum_{i=1}^H (a_i^l - \mu^l)^2}$$

在 RNN 中, 我们对每个时间片的输入都使用 LN 层来进行归一化,  $t$  时刻循环网络层的输入可以表示为:

$$\mathbf{a}^t = W_{hh} h^{t-1} + W_{xh} \mathbf{x}^t$$

则可以在  $\mathbf{a}^t$  上应用 LN 层:

$$\mathbf{h}^t = f\left(\frac{\mathbf{g}}{\sqrt{(\sigma^t)^2 + \epsilon}} \odot (\mathbf{a}^t - \mu^t) + \mathbf{b}\right) \quad \mu^t = \frac{1}{H} \sum_{i=1}^H a_i^t \quad \sigma^t = \sqrt{\frac{1}{H} \sum_{i=1}^H (a_i^t - \mu^t)^2}$$

(3) 参考链接:

- [模型优化之Layer Normalization](#)

## Instance Normalization

(1) 主要思想: IN 层和 LN 层相似, 仅对**单个样本进行归一化**, 但是 IN 层并不进行仿射变换;

(2) 公式:

$$\mu_{nc}(x) = \frac{1}{HW} \sum_{h=1}^H \sum_{w=1}^W x_{nchw}$$
$$\sigma_{nc}(x) = \sqrt{\frac{1}{HW} \sum_{h=1}^H \sum_{w=1}^W (x_{nchw} - \mu_{nc}(x))^2 + \epsilon}$$

知乎 @没头脑

这边需要注意, IN 层和 LN 层还有一点不同是, IN 层是作用在 FeatureMap 的**单个 Channel 上**的, 所以它计算出来的统计量是一个向量;

(3) 代码:

- [PyTorch](#):

# INSTANCENORM2D

**CLASS** `torch.nn.InstanceNorm2d(num_features, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)`

[SOURCE]

Applies Instance Normalization over a 4D input (a mini-batch of 2D inputs with additional channel dimension) as described in the paper [Instance Normalization: The Missing Ingredient for Fast Stylization](#).

$$y = \frac{x - E[x]}{\sqrt{\text{Var}[x] + \epsilon}} * \gamma + \beta$$

The mean and standard-deviation are calculated per-dimension separately for each object in a mini-batch.  $\gamma$  and  $\beta$  are learnable parameter vectors of size C (where C is the input size) if `affine` is `True`. The standard-deviation is calculated via the biased estimator, equivalent to `torch.var(input, unbiased=False)`.

By default, this layer uses instance statistics computed from input data in both training and evaluation modes.

If `track_running_stats` is set to `True`, during training this layer keeps running estimates of its computed mean and variance, which are then used for normalization during evaluation. The running estimates are kept with a default `momentum` of 0.1.

## • NOTE

This `momentum` argument is different from one used in optimizer classes and the conventional notion of momentum. Mathematically, the update rule for running statistics here is  $\hat{x}_{\text{new}} = (1 - \text{momentum}) \times \hat{x} + \text{momentum} \times x_t$ , where  $\hat{x}$  is the estimated statistic and  $x_t$  is the new observed value.

## • NOTE

`InstanceNorm2d` and `LayerNorm` are very similar, but have some subtle differences. `InstanceNorm2d` is applied on each channel of channeled data like RGB images, but `LayerNorm` is usually applied on entire sample and often in NLP tasks. Additionally, `LayerNorm` applies elementwise affine transform, while `InstanceNorm2d` usually don't apply affine transform.

可以看到，代码中也有一个 `track_running_stats` 参数；并且，可以设置 `affine=True` 来额外添加仿射参数；

- [Tensorflow](#):

## tfa.layers.InstanceNormalization



View source on GitHub

Instance normalization layer.

Inherits From: [GroupNormalization](#)

```
tfa.layers.InstanceNormalization(  
    **kwargs  
)
```



Tensorflow 2 中，IN 层则在 `tensorflow_addons` 包中；

(4) 有两个场景建议不要使用 IN 层：

- **MLP 或者 RNN 中**：因为在 MLP 或者 RNN 中，每个通道上只有一个数据，这时会自然不能使用 IN；
- **FeatureMap 比较小时**：因为此时 IN 的采样数据非常少，得到的归一化统计量将不再具有代表性；

(5) 参考链接：

- [模型优化之Instance Normalization](#)

## Consine Normalization

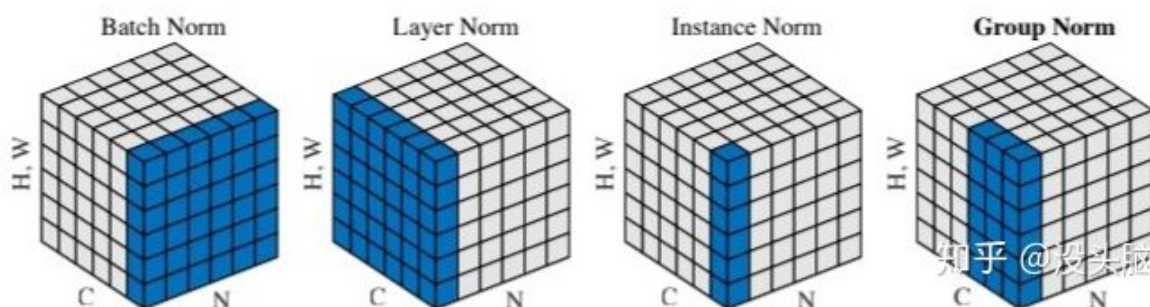
(1) 主要思想: CN 层和前面的思想都不太一样, 其不对输入数据做归一化, 也不对参数做归一化, 而是对输入数据和参数之间的点乘做出改动, 改为**计算两者的余弦相似度**  $\cos(\theta)$ ; **CN 层将模型的输出进行归一化, 使得输出有界, 但是也因此丢弃了原本输出中所含的 Scale 信息, 所以其作用有待探讨;**

(2) 公式:

$$w * x \Rightarrow \frac{w * x}{|w| * |x|}$$

## Group Normalization

(1) 主要思想: **GN 层更像是 LN 层和 IN 层的一般化形式, 通过参数  $G$  进行控制;** 当  $G = 1$ , GN 层即等价于 IN 层; 当  $G = C$  时, GN 层即等价于 LN 层;



(2) 公式:

$$\mu_{ng}(x) = \frac{1}{(C/G)HW} \sum_{c=gC/G}^{(g+1)C/G} \sum_{h=1}^H \sum_{w=1}^W x_{nchw}$$

$$\sigma_{ng}(x) = \sqrt{\frac{1}{(C/G)HW} \sum_{c=gC/G}^{(g+1)C/G} \sum_{h=1}^H \sum_{w=1}^W (x_{nchw} - \mu_{ng}(x))^2 + \epsilon}$$

## 人工智能前言讲习

“他山之石, 可以攻玉”, 站在巨人的肩膀才能看得更高, 走的更远。

- [【他山之石】整理 Deep Learning 调参 tricks](#)
- [【他山之石】深度学习中的那些 Trade-off](#)
- [【他山之石】tensorflow2.4性能调优最佳实践](#)
  - [【梦想做个翟老师】浅谈Tensorflow分布式架构: ring all-reduce算法](#)
  - [【瓦特兰蒂斯】单机多卡的正确打开方式 \(二\): TensorFlow](#)
- [【强基固本】机器学习常用评价指标总览](#)

## Problems in Coding

- Keras 和 Multiprocessing 组合 Bug:

我在 windows 上面运行的很好，但是放到 Linux 服务器上面后，子进程中的 `keras.models.load_model()` 就卡住不动了。Bug 原理和解决方法参考博客 [“keras使用多进程”](#)，写的非常棒，体会到了进程拷贝的问题。Bug 在 github 上面的链接参考 [Keras is not multi-processing safe](#) ；