面经

训练模型tricks

知乎 https://zhuanlan.zhihu.com/p/102817180

- Mixup
- 形变卷积
- GridMask
- 水平翻转
- 垂直翻转
- 90°旋转
- 亮度对比度
- 高斯模糊
- 运动模糊
- 随机平移
- 图像压缩
- RGB抖动

数学

马尔科夫链

马尔科夫**性质**：当一个随机过程在给定现在状态及所有过去状态情况下，其未来状态的条件概率分布仅依赖于当前状态；换句话说，在给定现在状态时，它与过去状态（即该过程的历史路径）是条件独立的，那么此随机过程即具有**马尔可夫性质**。具有马尔可夫性质的过程通常称之为**马尔可夫过程**。

马尔可夫链（英语：Markov chain），又称**离散时间马尔可夫链**（discrete-time Markov chain，缩写为**DTMC**[1]），为状态空间中经过从一个状态到另一个状态的转换的随机过程。该过程要求具备“无记忆”的性质：下一状态的概率分布只能由当前状态决定，在时间序列中它前面的事件均与之无关。这种特定类型的“无记忆性”称作马尔可夫性质。

马尔可夫网络，（马尔可夫随机场、无向图模型）是关于一组有马尔可夫性质随机变量X的全联合概率分布模型。

理论：

https://zhuanlan.zhihu.com/p/38343732 写得非常好！

贝叶斯定理

条件概率和贝叶斯定理

条件概率,我们使用这个概念的时候往往是用来解决**逆问题**，也就是已知结果反推原因的一个过程。而经典的贝叶斯定理公式：

$$P(B|A)=\frac{P(A|B)P(B)}{P(A)}.$$

我们常常利用这个公式去推导**我们无法直接观测的问题**，首先明白一个概念：上面的式子中，A*A*是结果，B*B*是原因。而上面这个式子我们要做的任务就是：

- 已知 $P(A|B)$ 和 $P(B)$（ $P(A)$ 可以通过 $P(A)=\sum^{n}P(A|B)\times P(B_i)$ 计算得到）
- 要求 $P(B|A)$

我们一般称 `P(原因)` 为**先验概率**， `P(原因|结果)` 称为**后验概率**，相应的为先验分布和后验分布。

点乘

点积（德语：Skalarprodukt，英语：Dot Product）又称**数量积**或**标量积**（德语：Skalarprodukt、英语：Scalar Product），是一种接受两个等长的数字序列（通常是坐标向量）、返回单个数字的代数运算。在欧几里得几何中，两个笛卡尔坐标向量的点积常称为**内积**（德语：inneres Produkt、英语：Inner Product）。

从代数角度看，先对两个数字序列中的每组对应元素求积，再对所有积求和，结果即为点积。从几何角度看，点积则是两个向量的长度与它们夹角余弦的积。这两种定义在笛卡尔坐标系中等价。

点积的名称源自表示点乘运算的点号($a\cdot b$)，读作a dot b，**标量积**的叫法则是在强调其运算结果为标量而非向量。向量的另一种乘法是叉乘（$a\times b$），其结果为向量，称为**叉积**或**向量积**。

代数定义 [编辑]

两个向量$\vec{a}=[a_1,a_2,\cdots,a_n]$和$\vec{b}=[b_1,b_2,\cdots,b_n]$的点积定义为：

$$\vec{a}\cdot\vec{b}=\sum_{i=1}^{n}a_ib_i=a_1b_1+a_2b_2+\cdots+a_nb_n$$

这里的Σ是求和符号，而*n*是向量空间的维数。

点积还可以写为：

$$\vec{a} \cdot \vec{b} = \vec{a}\vec{b}^{T}。$$

这里，$\vec{b}^{T}$ 是行向量$\vec{b}$的转置。

几何定义 [编辑]

在欧几里得空间中，点积可以直观地定义为

$$\vec{a} \cdot \vec{b} = |\vec{a}|\,|\vec{b}|\cos\theta$$

这里 $|\vec{x}|$ 表示$\vec{x}$的模（长度），θ表示两个向量之间的角度。

注意：点积的形式定义和这个定义不同；在形式定义中，$\vec{a}$和$\vec{b}$的夹角是通过上述等式定义的。

这样，两个互相垂直的向量的点积总是零。若$\vec{a}$和$\vec{b}$都是单位向量（长度为1），它们的点积就是它们的夹角的余弦。那么，给定两个向量，它们之间的夹角可以通过下列公式得到：

$$\cos\theta = \frac{\mathbf{a} \cdot \mathbf{b}}{|\vec{a}|\,|\vec{b}|}$$

这个运算可以简单地理解为：在点积运算中，第一个向量投影到第二个向量上（这里，向量的顺序是不重要的，点积运算是可交换的），然后通过除以它们的标量长度来“标准化”。这样，这个分数一定是小于等于1的，可以简单地转化成一个角度值。

线性代数

例 3 n 个变量 $x_1, x_2, \cdots, x_n$ 与 m 个变量 $y_1, y_2, \cdots, y_m$ 之间的关系式

$$\begin{cases} y_1 = a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n, \\ y_2 = a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n, \\ \cdots\cdots\cdots\cdots \\ y_m = a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{cases} \tag{2}$$

表示一个从变量 $x_1, x_2, \cdots, x_n$ 到变量 $y_1, y_2, \cdots, y_m$ 的线性变换，其中 a_{ij} 为常数. 线性变换(2)的系数 a_{ij} 构成矩阵 $\boldsymbol{A} = (a_{ij})_{m\times n}$.

给定了线性变换(2)，它的系数所构成的矩阵(称为系数矩阵)也就确定. 反之，如果给出一个矩阵作为线性变换的系数矩阵，则线性变换也就确定. 在这个意义上，线性变换和矩阵之间存在着一一对应的关系.

例如线性变换

$$\begin{cases} y_1 = x_1, \\ y_2 = x_2, \\ \cdots\cdots\cdots\cdots \\ y_n = x_n \end{cases}$$

叫做恒等变换，它对应的一个 n 阶方阵

$$\boldsymbol{E} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}$$

叫做 n 阶单位矩阵，简称单位阵. 这个方阵的特点是：从左上角到右下角的直线(叫做(主)对角线)上的元素都是 1，其他元素都是 0. 即单位阵 $\boldsymbol{E}$ 的 (i,j) 元为

$$\delta_{ij} = \begin{cases} 1, & \text{当 } i = j, \\ 0, & \text{当 } i \neq j \end{cases} (i, j = 1, 2, \cdots, n).$$

又如线性变换

$$\begin{cases} y_1 = \lambda_1 x_1, \\ y_2 = \lambda_2 x_2, \\ \cdots\cdots\cdots\cdots \\ y_n = \lambda_n x_n \end{cases}$$

对应 n 阶方阵

$$\boldsymbol{\Lambda} = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{pmatrix}.$$

这个方阵的特点是：不在对角线上的元素都是 0. 这种方阵称为对角矩阵，简称对角阵. 对角阵也记作

$$\boldsymbol{\Lambda} = \operatorname{diag}(\lambda_1, \lambda_2, \cdots, \lambda_n).$$

由于矩阵和线性变换之间存在一一对应的关系，因此可以利用矩阵来研究线性变换，也可以利用线性变换来解释矩阵的含义.

代数余子式

定义

在n阶行列式D中划去任意选定的k行、k列后，余下的元素按原来顺序组成的n-k阶行列式M，称为行列式D的k阶子式A的余子式。如果k阶子式A在行列式D中的行和列的标号分别为$i_1, i_2, \ldots, i_k$和$j_1, j_2, \ldots, j_k$。则在A的余子式M前面添加符号：

$$(-1)^{(i_1+i_2+,\ldots,+i_k)+(j_1+j_2+,\ldots,+j_k)}.$$

后,所得到的n-k阶行列式，称为行列式D的k阶子式A的代数余子式。 [2]

例题分析

例1 在五阶行列式 [1]

$$D=\begin{vmatrix} 3 & -1 & -1 & 1 & 1 \\ 2 & 1 & 1 & -1 & -1 \\ 0 & 0 & 5 & -2 & 2 \\ 0 & 0 & 6 & 2 & -1 \\ 0 & 0 & 1 & 0 & 1 \end{vmatrix}.$$

中，划定第二行、四行和第二列、三列，就可以确定D的一个二阶子行列式

$$A: A=\begin{vmatrix} 1 & 1 \\ 0 & 6 \end{vmatrix}.$$

A的相应的余子式M为：

$$M=\begin{vmatrix} 3 & 1 & 1 \\ 0 & -2 & 2 \\ 0 & 0 & 1 \end{vmatrix}.$$

子行列式A的相应的代数余子式为：

$$(-1)^{(2+4)+(2+3)}M=-\begin{vmatrix} 3 & 1 & 1 \\ 0 & -2 & 2 \\ 0 & 0 & 1 \end{vmatrix}.$$

伴随矩阵（伴随阵）A*

例 9 行列式$|A|$的各个元素的代数余子式A_{ij}所构成的如下的矩阵

$$A^{*}=\begin{pmatrix} A_{11} & A_{21} & \cdots & A_{n1} \\ A_{12} & A_{22} & \cdots & A_{n2} \\ \vdots & \vdots & & \vdots \\ A_{1n} & A_{2n} & \cdots & A_{nn} \end{pmatrix},$$

称为矩阵 A 的伴随矩阵,简称伴随阵.试证

$$AA^{*}=A^{*}A=|A|E.$$

证 设 $A=(a_{ij})$,记 $AA^{*}=(b_{ij})$,则

$$b_{ij}=a_{i1}A_{j1}+a_{i2}A_{j2}+\cdots+a_{in}A_{jn}=|A|\delta_{ij},$$

故

$$AA^{*}=(|A|\delta_{ij})=|A|(\delta_{ij})=|A|E.$$

类似有

$$A^{*}A=\left(\sum_{k=1}^{n}A_{ki}a_{kj}\right)=(|A|\delta_{ij})=|A|(\delta_{ij})=|A|E.$$

逆矩阵

定义 7 对于 n 阶矩阵 A，如果有一个 n 阶矩阵 B，使

$$AB = BA = E,$$

则说矩阵 A 是可逆的，并把矩阵 B 称为 A 的逆矩阵，简称逆阵.

如果矩阵 A 是可逆的，那么 A 的逆阵是惟一的. 这是因为：设 B，C 都是 A 的逆阵，则有

$$B = BE = B(AC) = (BA)C = EC = C,$$

所以 A 的逆阵是惟一的.

A 的逆阵记作 A^{-1}. 即若 $AB = BA = E$，则 $B = A^{-1}$.

定理 1 若矩阵 A 可逆，则 $|A| \neq 0$.

证 A 可逆，即有 A^{-1}，使 $AA^{-1} = E$. 故 $|A| \cdot |A^{-1}| = |E| = 1$，所以 $|A| \neq 0$.

定理 2 若 $|A| \neq 0$，则矩阵 A 可逆，且

$$A^{-1} = \frac{1}{|A|} A^*, \tag{10}$$

其中 A^* 为矩阵 A 的伴随阵.

证 由例 9 知

$$AA^* = A^* A = |A| E,$$

因 $|A| \neq 0$，故有

$$A \frac{1}{|A|} A^* = \frac{1}{|A|} A^* A = E,$$

所以，按逆阵的定义，即知 A 可逆，且有

$$A^{-1} = \frac{1}{|A|} A^*.$$

证毕

当 $|A| = 0$ 时，A 称为奇异矩阵，否则称非奇异矩阵. 由上面两定理可知：A 是可逆矩阵的充分必要条件是 $|A| \neq 0$，即可逆矩阵就是非奇异矩阵.

(i) 若 A 可逆，则 A^{-1} 亦可逆，且 $(A^{-1})^{-1}=A$.

(ii) 若 A 可逆，数 $\lambda\neq 0$，则 λA 可逆，且 $(\lambda A)^{-1}=\frac{1}{\lambda}A^{-1}$.

(iii) 若 A，B 为同阶矩阵且均可逆，则 AB 亦可逆，且

$$(AB)^{-1}=B^{-1}A^{-1}.$$

证 $(AB)(B^{-1}A^{-1})=A(BB^{-1})A^{-1}=AEA^{-1}=AA^{-1}=E$，由推论，即有 $(AB)^{-1}=B^{-1}A^{-1}$.

(iv) 若 A 可逆，则 A^{T} 亦可逆，且 $(A^{\mathrm{T}})^{-1}=(A^{-1})^{\mathrm{T}}$.

证 $$A^{\mathrm{T}}(A^{-1})^{\mathrm{T}}=(A^{-1}A)^{\mathrm{T}}=E^{\mathrm{T}}=E,$$

所以 $$(A^{\mathrm{T}})^{-1}=(A^{-1})^{\mathrm{T}}.$$ 证毕

当 A 可逆时，还可定义

$$A^{0}=E,\ A^{-k}=(A^{-1})^{k},$$

其中 k 为正整数. 这样，当 A 可逆，λ、μ 为整数时，有

$$A^{\lambda}A^{\mu}=A^{\lambda+\mu},\ (A^{\lambda})^{\mu}=A^{\lambda\mu}.$$

定义 3 在 $m\times n$ 矩阵 A 中，任取 k 行与 k 列（$k\leqslant m$，$k\leqslant n$），位于这些行列交叉处的 k^2 个元素，不改变它们在 A 中所处的位置次序而得的 k 阶行列式，称为矩阵 A 的 k 阶子式.

$m\times n$ 矩阵 A 的 k 阶子式共有 $C_m^k\cdot C_n^k$ 个.

定义 4 设在矩阵 A 中有一个不等于 0 的 r 阶子式 D，且所有 $r+1$ 阶子式（如果存在的话）全等于 0，那么 D 称为矩阵 A 的最高阶非零子式，数 r 称为矩阵 A 的秩，记作 $R(A)$. 并规定零矩阵的秩等于 0.

对于 n 阶矩阵 A，由于 A 的 n 阶子式只有一个 $|A|$，故当 $|A|\neq 0$ 时 $R(A)=n$，当 $|A|=0$ 时 $R(A)<n$. 可见可逆矩阵的秩等于矩阵的阶数，不可逆矩阵的秩小于矩阵的阶数. 因此，可逆矩阵又称满秩矩阵，不可逆矩阵（奇异矩阵）又称降秩矩阵.

特征值和特征向量

定义 6 设 A 是 n 阶矩阵，如果数 λ 和 n 维非零列向量 x 使关系式

$$Ax = \lambda x \tag{1}$$

成立，那么，这样的数 λ 称为矩阵 A 的特征值，非零向量 x 称为 A 的对应于特征值 λ 的特征向量.

(1) 式也可写成

$$(A - \lambda E)x = 0,$$

这是 n 个未知数 n 个方程的齐次线性方程组，它有非零解的充分必要条件是系数行列式

$$|A - \lambda E| = 0,$$

即

$$\begin{vmatrix} a_{11} - \lambda & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} - \lambda & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} - \lambda \end{vmatrix} = 0.$$

上式是以 λ 为未知数的一元 n 次方程，称为矩阵 A 的特征方程. 其左端 $|A - \lambda E|$ 是 λ 的 n 次多项式，记作 $f(\lambda)$，称为矩阵 A 的特征多项式. 显然，A 的特征值就是特征方程的解. 特征方程在复数范围内恒有解，其个数为方程的次数(重根按重数计算)，因此，n 阶矩阵 A 在复数范围内有 n 个特征值.

设 n 阶矩阵 $A = (a_{ij})$ 的特征值为 $\lambda_1, \lambda_2, \cdots, \lambda_n$，不难证明①

(i) $\lambda_1 + \lambda_2 + \cdots + \lambda_n = a_{11} + a_{22} + \cdots + a_{nn}$；

(ii) $\lambda_1 \lambda_2 \cdots \lambda_n = |A|$.

设 $\lambda = \lambda_i$ 为矩阵 A 的一个特征值，则由方程

$$(A - \lambda_i E)x = 0$$

可求得非零解 $x = p_i$，那么 p_i 便是 A 的对应于特征值 λ_i 的特征向量.(若 λ_i 为实数，则 p_i 可取实向量；若 λ_i 为复数，则 p_i 为复向量.)

相似矩阵

定义 7 设 A, B 都是 n 阶矩阵，若有可逆矩阵 P，使

$$P^{-1}AP = B,$$

则称 B 是 A 的相似矩阵，或说矩阵 A 与 B 相似. 对 A 进行运算 $P^{-1}AP$ 称为对 A 进行相似变换，可逆矩阵 P 称为把 A 变成 B 的相似变换矩阵.

定理 3 若 n 阶矩阵 A 与 B 相似，则 A 与 B 的特征多项式相同，从而 A 与 B 的特征值亦相同.

证 因 A 与 B 相似，即有可逆矩阵 P，使 $P^{-1}AP = B$. 故

$$|B - \lambda E| = |P^{-1}AP - P^{-1}(\lambda E)P| = |P^{-1}(A - \lambda E)P|$$

$$= |P^{-1}||A-\lambda E||P| = |A-\lambda E|.$$

推论　若 n 阶矩阵 A 与对角阵

$$\Lambda = \begin{pmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{pmatrix}$$

相似，则 $\lambda_1, \lambda_2, \cdots, \lambda_n$ 即是 A 的 n 个特征值.

正定矩阵

定义 10　设有二次型 $f(x) = x^{\mathrm{T}}Ax$，如果对任何 $x \neq 0$，都有 $f(x) > 0$（显然 $f(0) = 0$），则称 f 为正定二次型，并称对称阵 A 是正定的；如果对任何 $x \neq 0$ 都有 $f(x) < 0$，则称 f 为负定二次型，并称对称阵 A 是负定的.

定理 10　n 元二次型 $f = x^{\mathrm{T}}Ax$ 为正定的充分必要条件是：它的标准形的 n 个系数全为正，即它的规范形的 n 个系数全为 1，亦即它的正惯性指数等于 n.

推论　对称阵 A 为正定的充分必要条件是：A 的特征值全为正.

定理 11　对称阵 A 为正定的充分必要条件是：A 的各阶主子式都为正，即

$$a_{11} > 0, \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} > 0, \cdots, \begin{vmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nn} \end{vmatrix} > 0,$$

对称阵 A 为负定的充分必要条件是：奇数阶主子式为负，而偶数阶主子式为正，即

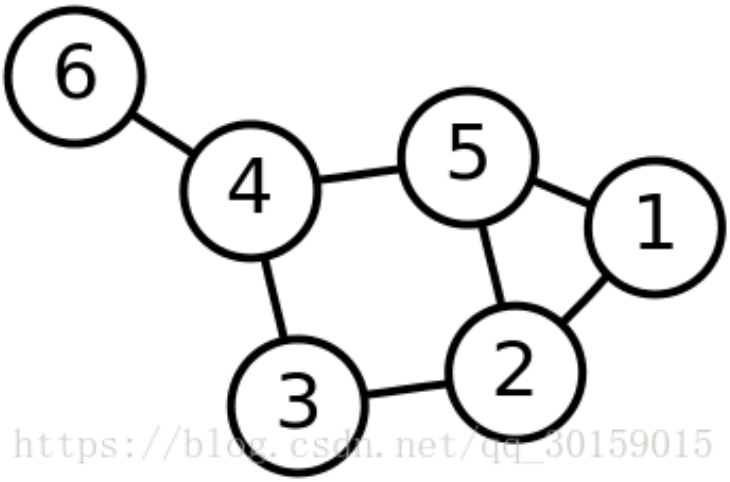
$$(-1)^r \begin{vmatrix} a_{11} & \cdots & a_{1r} \\ \vdots & & \vdots \\ a_{r1} & \cdots & a_{rr} \end{vmatrix} > 0 \ (r = 1, 2, \cdots, n).$$

这个定理称为赫尔维茨定理，这里不予证明.

特征分解（谱分解）：将矩阵分解为由其特征值和特征向量表示的矩阵之积的方法。需要注意只有对可对角化矩阵才可以施以特征分解。

拉普拉斯矩阵

拉普拉斯矩阵是图论中用到的一种重要矩阵，给定一个有n个顶点的图 G=(V,E)，其拉普拉斯矩阵被定义为 L = D-A，D 其中为图的度矩阵，A为图的邻接矩阵。例如，给定一个简单的图：



把此“图”转换为邻接矩阵的形式，记为A：

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

把W的每一列元素加起来得到N个数，然后把它们放在对角线上（其它地方都是零），组成一个N×N的对角矩阵，记为度矩阵D，如下图所示。其实度矩阵（对角线元素）表示的就是原图中每个点的度数，即由该点发出的边之数量：

$$\begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

根据拉普拉斯矩阵的定义L = D-A，可得拉普拉斯矩阵L 为：

$$\begin{pmatrix} 2 & -1 & 0 & 0 & -1 & 0 \\ -1 & 3 & -1 & 0 & -1 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3 & -1 & -1 \\ -1 & -1 & 0 & -1 & 3 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix}$$

显然，拉普拉斯矩阵都是对称的。此外，另外一种更为常用的拉普拉斯矩阵形式是正则化的拉普拉斯矩阵（Symmetric normalized Laplacian），定义为：

$$L^{\text{sym}} := D^{-1/2} L D^{-1/2} = I - D^{-1/2} A D^{-1/2}$$

该矩阵中的元素由下面的式子给出：

$$L^{\text{sym}}_{i,j} := \begin{cases} 1 & \text{if } i = j \text{ and } \deg(v_i) \neq 0 \\ -\frac{1}{\sqrt{\deg(v_i)\deg(v_j)}} & \text{if } i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise.} \end{cases}$$

傅里叶变换（Fourier transform）

https://www.bilibili.com/video/BV1pW411J7s8

牛顿法

牛顿法的原理是使用函数 $f(x)$ 的泰勒级数的前面几项来寻找方程 $f(x)=0$ 的根。

将函数 $f(x)$ 在 x_0 处展开成泰勒级数:

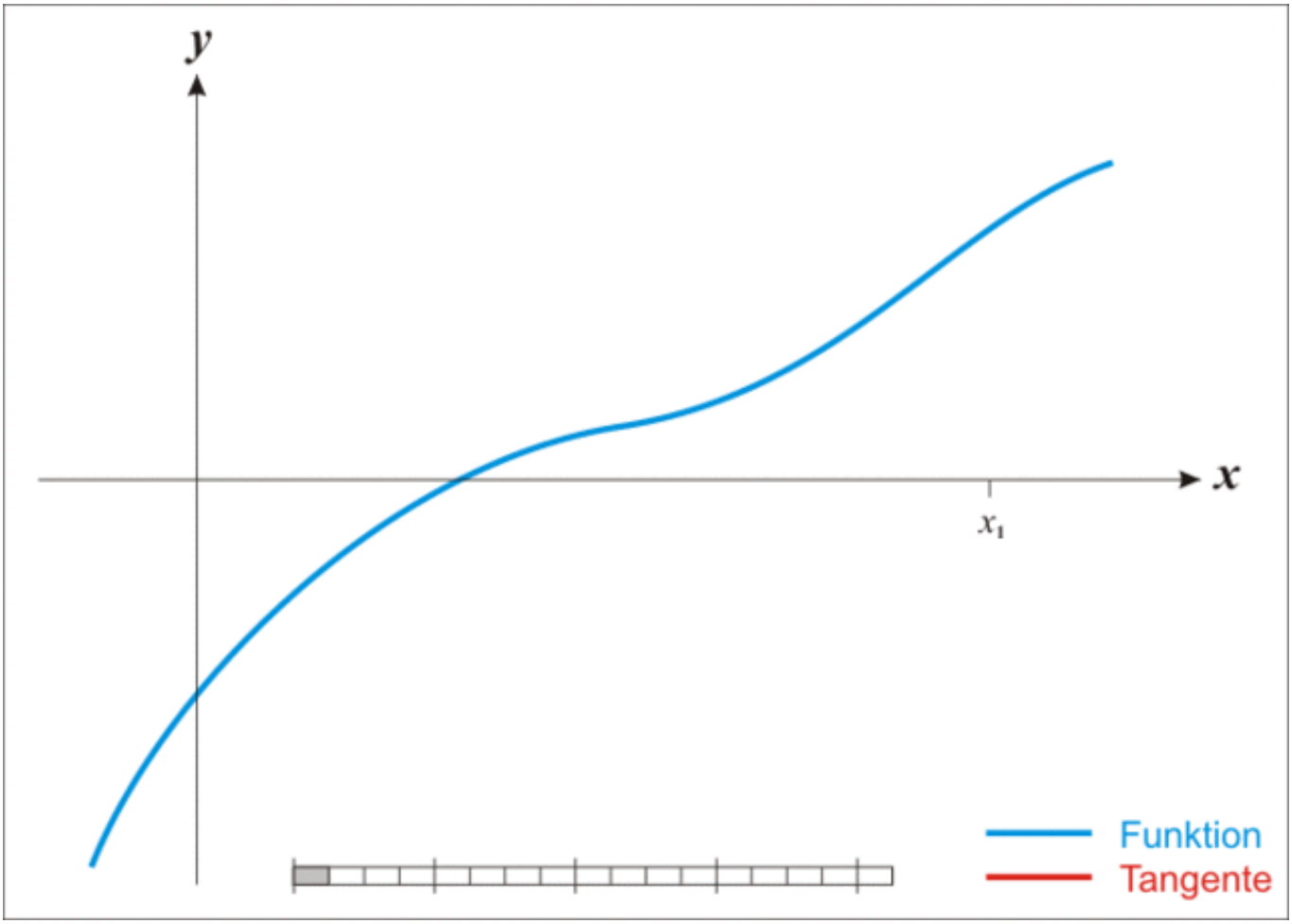
$$f(x)=f(x_0)+f^{'}(x_0)(x-x_0)+\frac{1}{2}f^{''}(x_0)(x-x_0)^2+\dots$$

取其线性部分，作为 $f(x)$ 的近似，则可用 $f(x_0)+f^{'}(x_0)(x-x_0)=0$ 的解来近似 $f(x)=0$ 的解，其解为 $x_1=x_0-\frac{f(x_0)}{f^{'}(x_0)}$ 。

由于对 $f(x)$ 的近似只是一阶展开，因此 x_1 并非 $f(x)=0$ 的解，只能说 $f(x_1)$ 比 $f(x_0)$ 更接近0。于是，考虑迭代求解:

$$x_{n+1}=x_n-\frac{f(x_n)}{f^{'}(x_n)}$$

迭代过程可参考下图:



Python

glob

glob模块的主要方法就是glob,该方法返回所有匹配的文件路径**列表**（list）；该方法需要一个参数用来指定匹配的路径字符串（字符串可以为绝对路径也可以为相对路径），其返回的文件名只包括当前目录里的文件名，不包括子文件夹里的文件。

```
# 返回list
glob.glob('c:*.txt')
glob.glob('E:\pic**.jpg')
# 返回迭代器
f = glob.iglob(r'../*.py')

# 示例
glob.glob('./[0-9].*')
glob.glob('*.gif')
glob.glob('?.gif')
```

os.path

sorted

```
# 按值(value)排序:
sorted(key_value.items(), key = lambda kv:(kv[1], kv[0]))

# 通过age和name排序
lis = [{ "name" : "Taobao", "age" : 100},
{ "name" : "Runoob", "age" : 7 },
{ "name" : "Google", "age" : 100 },
{ "name" : "Wiki" , "age" : 200 }]

sorted(lis, key = lambda i: (i['age'], i['name']))

# 逆序
sorted(lis, key = lambda i: i['age'],reverse=True)
```

char2int int2char

```
chr(97) # -> 'a'
ord('a') # -> 97
```

GIL (Global Interpreter Lock)

全局锁。。。保证了多线程的线程安全，但由于是全局锁导致多线程性能极差。

> 解决多线程之间数据完整性和状态同步的最简单方法自然就是加锁。 于是有了GIL这把超级大锁，而当越来越多的代码库开发者接受了这种设定后，他们开始大量依赖这种特性（即默认python内部对象是thread-safe的，无需在实现时考虑额外的内存锁和同步操作）。

GIL原理

其实，由于 Python 的线程就是 C 语言的 pthread，它是通过操作系统调度算法调度执行的。

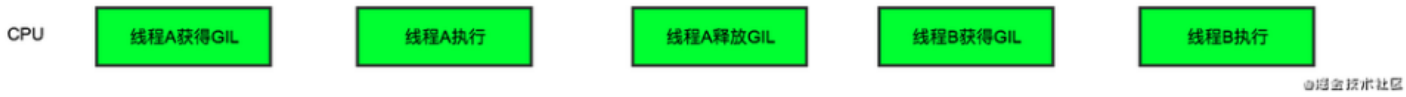
Python 2.x 的代码执行是基于 opcode 数量的调度方式，简单来说就是每执行一定数量的字节码，或遇到系统 IO 时，会强制释放 GIL，然后触发一次操作系统的线程调度。

虽然在 Python 3.x 进行了优化，基于固定时间的调度方式，就是每执行固定时间的字节码，或遇到系统 IO 时，强制释放 GIL，触发系统的线程调度。

但这种线程的调度方式，都会导致同一时刻只有一个线程在运行。

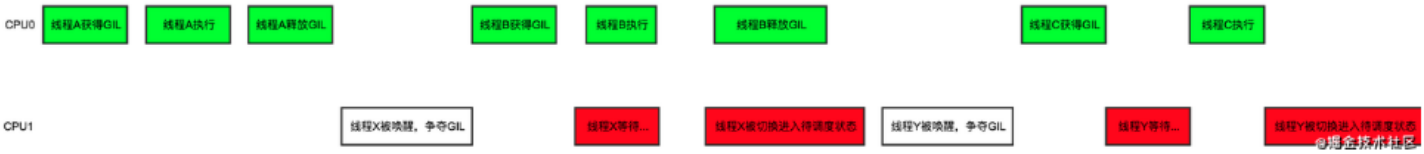
而线程在调度时，又依赖系统的 CPU 环境，也就是在单核 CPU 或多核 CPU 下，多线程在调度切换时的成本是不同的。

如果是在单核 CPU 环境下，多线程在执行时，线程 A 释放了 GIL 锁，那么被唤醒的线程 B 能够立即拿到 GIL 锁，线程 B 可以无缝接力继续执行，执行流程如下图：



而如果在在多核 CPU 环境下，当多线程执行时，线程 A 在 CPU0 执行完之后释放 GIL 锁，其他 CPU 上的线程都会进行竞争。

但 CPU0 上的线程 B 可能又马上获取到了 GIL，这就导致其他 CPU 上被唤醒的线程，只能眼巴巴地看着 CPU0 上的线程愉快地执行着，而自己只能等待，直到又被切换到待调度的状态，这就会产生多核 CPU 频繁进行线程切换，消耗资源，这种情况也被叫做「CPU颠簸」。整个执行流程如下图：



图中绿色部分是线程获得了 GIL 并进行有效的 CPU 运算，红色部分是被唤醒的线程由于没有争夺到 GIL，只能无效等待，无法充分利用 CPU 的并行运算能力。

这就是多线程在多核 CPU 下，执行效率还不如单线程或单核 CPU 效率高的原因。

到此，我们可以得出一个结论：**如果使用多线程运行一个 CPU 密集型任务，那么 Python 多线程是无法提高运行效率的。**

别急，你以为事情就这样结束了吗?

我们还需要考虑另一种场景：如果多线程运行的不是一个 CPU 密集型任务，而是一个 **IO 密集型**的任务，结果又会如何呢?

答案是，**多线程可以显著提高运行效率！**

其实原因也很简单，因为 IO 密集型的任务，大部分时间都花在等待 IO 上，并没有一直占用 CPU 的资源，所以并不会像上面的程序那样，进行无效的线程切换。

例如，如果我们想要下载 2 个网页的数据，也就是发起 2 个网络请求，如果使用单线程的方式运行，只能是依次串行执行，其中等待的总耗时是 2 个网络请求的时间之和。

而如果采用 2 个线程的方式同时处理，这 2 个网络请求会同时发送，然后同时等待数据返回（IO等待），最终等待的时间取决于耗时最久的线程时间，这会比串行执行效率要高得多。

所以，**如果需要运行 IO 密集型任务，Python 多线程是可以提高运行效率的。**

解决方案

- 用multiprocessing替代Thread

它完整的复制了一套thread所提供的接口方便迁移。唯一的不同就是它使用了多进程而不是多线程。

-

  ## 解决方案

  既然 GIL 的存在会导致这么多问题，那我们在开发时，需要注意哪些地方，避免受到 GIL 的影响呢?

  我总结了以下几个方案:

  1. IO 密集型任务场景，可以使用多线程可以提高运行效率
  2. CPU 密集型任务场景，不使用多线程，推荐使用多进程方式部署运行
  3. 更换没有 GIL 的 Python 解释器，但需要提前评估运行结果是否与 CPython 一致
  4. 编写 Python 的 C 扩展模块，把 CPU 密集型任务交给 C 模块处理，但缺点是编码较为复杂
  5. 更换其他语言 :)

Pytorch

Tensor

expand

```
>>> x = torch.tensor([[1], [2], [3]])
>>> x.size()
torch.Size([3, 1])
>>> x.expand(3, 4)
tensor([[ 1,  1,  1,  1],
        [ 2,  2,  2,  2],
        [ 3,  3,  3,  3]])
>>> x.expand(-1, 4)   # -1 means not changing the size of that dimension
tensor([[ 1,  1,  1,  1],
        [ 2,  2,  2,  2],
        [ 3,  3,  3,  3]])
```

expand不会分配内存

repeat

repeat会分配内存

```
>>> a
tensor([[ 1.,  2.,  3.,  4.],
        [ 5.,  6.,  7.,  8.],
        [ 9., 10., 11., 12.]])
>>> a.repeat(2,3).shape
torch.Size([6, 12])
>>> a.repeat(2,2,3).shape
torch.Size([2, 6, 12])
```

is_leaf

是否为叶子节点。

可以看做是bp时最末端的节点（例如输入tensor等）。

判定条件：

- 如果requires_grad=False，则是叶子
- 如果requires_grad=True，且是用户生成的tensor（不是计算得到的中间量），则是叶子

神经网络层中的权值w的tensor都是叶子节点。torch会计算从loss到叶子节点的梯度，而后修改叶子节点的tensor的值。

is_contiguous

判断tensor是否是连续的，这里的连续是指在内存中存成一个一维数组。

在使用transpose、permute等操作后，tensor不再是行优先存储，因此在调用view方法时会报错。

可以通过调用contiguous()方法生成新的tensor，来解决transpose等操作后无法view的问题。

参考：https://zhuanlan.zhihu.com/p/64551412

Dataloader

- num_workers
  - dataloader一次性创建`num_worker`个worker，（也可以说dataloader一次性创建`num_worker`个工作进程，worker也是普通的工作进程），并用`batch_sampler`将指定batch分配给指定worker，worker将它负责的batch加载进RAM。然后，dataloader从RAM中找本轮迭代要用的batch，如果找到了，就使用。如果没找到，就要`num_worker`个worker继续加载batch到内存，直到dataloader在RAM中找到目标batch。一般情况下都是能找到的，因为`batch_sampler`指定batch时当然优先指定本轮要用的batch。
  - 读取完成后，主进程使用fori读取各个worker中的数据，如果没有读到worker0的数据，其他数据就会暂存在缓冲区，worker0读取完成后其他的才会输入进来。worker1，worker2等同理。因此通常会在worker_num次后阻塞几秒时间。

```
while True:
    assert (not self.shutdown and self.batches_outstanding > 0)
    # 阻塞式的从data_queue里面获取处理好的批数据
    idx, batch = self._get_batch()
    # 任务数减一
    self.batches_outstanding -= 1
    # 这一步就是造成的周期阻塞现象的原因
    # 因为该DataLoader设计要保证模型复现性(个人猜测)，因此数据读取的顺序也是需要保证可复现的
    # 因此每次获取data以后，要校验和rcvd_idx是否一致
    # 若不一致，则先把获取到的数据放到reorder_dict这个缓存dict中，继续死循环
    # 直到获取到相应的idx编号于rcvd_idx可以对应上，并将数据返回
    if idx != self.rcvd_idx:
        # store out-of-order samples
        self.reorder_dict[idx] = batch
        continue
    return self._process_next_batch(batch)
```

  - `num_worker`设置得大，好处是寻batch速度快，因为下一轮迭代的batch很可能在上一轮/上上一轮...迭代时已经加载好了。坏处是内存开销大，也加重了CPU负担（worker加载数据到RAM的进程是CPU复制的嘛）。`num_workers`的经验设置值是自己电脑/服务器的CPU核心数，如果CPU很强、RAM也很充足，就可以设置得更大些。
  - 如果`num_worker`设为0，意味着每一轮迭代时，dataloader不再有自主加载数据到RAM这一步骤（因为没有worker了），而是在RAM中找batch，找不到时再加载相应的batch。缺点当然是速度更慢。

- pin_memory
  - pin_memory就是锁页内存，创建DataLoader时，设置pin_memory=True，则意味着生成的Tensor数据最开始是属于内存中的锁页内存，这样将内存的Tensor转义到GPU的显存就会更快一些。
  - 主机中的内存，有两种存在方式，一是锁页，二是不锁页，锁页内存存放的内容在任何情况下都不会与主机的虚拟内存进行交换（注：虚拟内存就是硬盘），而不锁页内存在主机内存不足时，数据会存放在虚拟内存中。
  - 在使用多卡训练时，如果使用pin_memory，并且没有将tensor转换回cpu的需要，则可以在调用cuda()的时候加上non_blocking=True，这样可以避免一个卡慢的时候由于barrier导致所有卡都卡在这里。
- collate_fn
  - 将一个list的sample组成一个mini-batch的函数

conv2d的参数及含义

Parameters:

- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*int or tuple*) – Size of the convolving kernel
- **stride** (*int or tuple, optional*) – Stride of the convolution. Default: 1
- **padding** (*int or tuple, optional*) – Zero-padding added to both sides of the input. Default: 0
- **dilation** (*int or tuple, optional*) – Spacing between kernel elements. Default: 1
- **groups** (*int, optional*) – Number of blocked connections from input channels to output channels. Default: 1
- **bias** (*bool, optional*) – If `True`, adds a learnable bias to the output. Default: `True`

dilation：从输入中每隔dilation-1个元素取一个元素

卷积层输出：

$$\begin{cases} height_{out} &= (height_{in} - height_{kernel} + 2 * padding) \ / \ stride + 1 \\ width_{out} &= (width_{in} - width_{kernel} + 2 * padding) \ / \ stride + 1 \end{cases}$$

计算量：

分为加法和乘法，乘加。每个点每一个卷积核的计算：ks*ks次乘，ks*ks-1次加。

Optimization

博客 https://blog.csdn.net/xiaoxifei/article/details/87797935

```
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

微调

- **局部微调**：加载了模型参数后，只想调节最后几层，其它层不训练，也就是不进行梯度计算，pytorch提供的requires_grad使得对训练的控制变得非常简单

```
model = torchvision.models.resnet18(pretrained=True)
for param in model.parameters():
    param.requires_grad = False
# 替换最后的全连接层， 改为训练100类
# 新构造的模块的参数默认requires_grad为True
model.fc = nn.Linear(512, 100)

# 只优化最后的分类层
optimizer = optim.SGD(model.fc.parameters(), lr=1e-2, momentum=0.9)
```

- **全局微调**：对全局微调时，只不过我们希望改换过的层和其他层的学习速率不一样，这时候把其它层和新层在optimizer中单独赋予不同的学习速率。

```
ignored_params = list(map(id, model.fc.parameters()))
base_params = filter(lambda p: id(p) not in ignored_params,
                     model.parameters())

optimizer = torch.optim.SGD([
            {'params': base_params},
            {'params': model.fc.parameters(), 'lr': 1e-3}
            ], lr=1e-2, momentum=0.9)
```

model.train() model.eval()

model.train()

- 启用 Batch Normalization 和 Dropout。

model.eval()

- 不启用 Batch Normalization 和 Dropout。

model.eval()和torch.no_grad()的区别

在eval模式下，dropout层会让所有的激活单元都通过，而BN层会停止计算和更新mean和var，直接使用在训练阶段已经学出的mean和var值。
该模式不会影响各层的gradient计算行为，即gradient计算和存储与training模式一样，只是不进行反向传播（back probagation)。

而with torch.no_grad()则主要是用于停止autograd模块的工作，以起到加速和节省显存的作用。它的作用是将该with语句包裹起来的部分停止梯度的更新，从而节省了GPU算力和显存，但是并不会影响dropout和BN层的行为。

如果不在意显存大小和计算时间的话，仅仅使用model.eval()已足够得到正确的validation/test的结果；而with torch.no_grad()则是更进一步加速和节省gpu空间（因为不用计算和存储梯度），从而可以更快计算，也可以跑更大的batch来测试。

eval后也可以反传，但没有了dropout和bn的更新。

torch.no_grad()和requires_grad()的区别

- no_grad()不计算所有的梯度，因此如果前面有层需要梯度，是传递不到的
- requires_grad()不计算权重的梯度，但如果前面层需要梯度，还是会计算x的梯度。

pytorch实现梯度更新

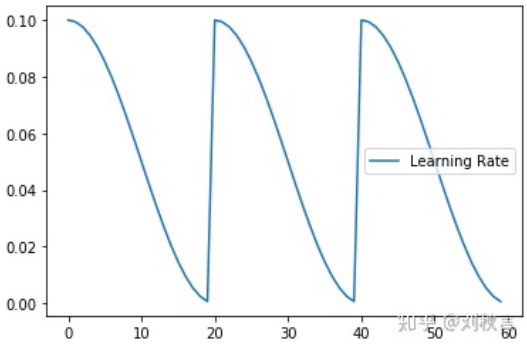
pytorch bp实现

lr更新

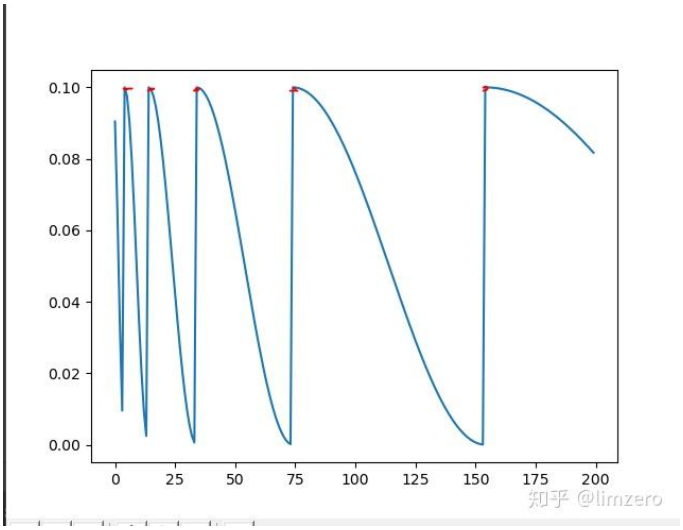
```
torch.optim.lr_scheduler.StepLR
scheduler = StepLR(optimizer, step_size=30, gamma=0.1)
for epoch in range(100):
    train(...)
    validate(...)
    scheduler.step()

torch.optim.lr_scheduler.StepLR(optimizer, step_size, gamma=0.1, last_epoch=-1)
torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones, gamma=0.1, last_epoch=-1)
torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma, last_epoch=-1)
torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max, eta_min=0,
last_epoch=-1)
torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(optimizer, T_0, T_mult=1,
eta_min=0, last_epoch=- 1, verbose=False)
torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.1,
patience=10, verbose=False, threshold=0.0001, threshold_mode='rel', cooldown=0,
min_lr=0, eps=1e-08) # 多epoch不降后调整lr
torch.optim.lr_scheduler.LambdaLR(optimizer, lr_lambda, last_epoch=-1) # 多组参数时可以
设置不同的lr变化策略
```

余弦退火



余弦退火+warm restarts：T_mult=2



知乎 多种更新方式 https://zhuanlan.zhihu.com/p/69411064

多卡训练

知乎 https://zhuanlan.zhihu.com/p/86441879

更深入使用（包括sync BN，all_reduce使用、barrior使用、infer实现等）https://zhuanlan.zhihu.com/p/178402798

torch.multinomial(input, num_samples, replacement, out)

对input取num_samples值，input中每个位置的值代表权重，返回的是indexes。replacement代表是否放回。

in-place

通常在方法后加一个下划线，可以原地计算，而不是返回一个值：

```
x = torch.rand(2)
y = torch.rand(2)
x.add_(y)
```

在官方文档中提到，如果使用in-place并且没有报错的话，可以确定梯度计算是正确的。

由此可以引出另一个tensor中的概念：叶子节点和非叶子节点

归一化方法

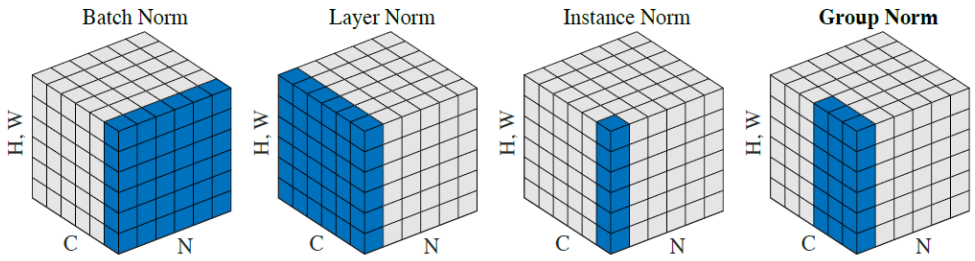


Figure 2. **Normalization methods**. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

Batch Normalization（2015年）、Layer Normalization（2016年）、Instance Normalization（2017年）、Group Normalization（2018年）、Switchable Normalization（2018年）

将输入的图像shape记为[N, C, H, W]，这几个方法主要的区别就是在，

Batch Norm是在batch上，对NHW做归一化，就是对每个单一通道输入进行归一化，这样做对小batchsize效果不好；
Layer Norm在通道方向上，对CHW归一化，就是对每个深度上的输入进行归一化，主要对RNN作用明显；
Instance Norm在图像像素上，对HW做归一化，对一个图像的长宽即对一个像素进行归一化，用在风格化迁移；
Group Norm将channel分组，有点类似于LN，只是GN把channel也进行了划分，细化，然后再做归一化；
Switchable Norm是将BN、LN、IN结合，赋予权重，让网络自己去学习归一化层应该使用什么方法。

BN

消除**奇异样本数据**导致的不良影响。

优缺点：BN实际使用时需要计算并且保存某一层神经网络batch的均值和方差等统计信息，对于对一个固定深度的前向神经网络（DNN，CNN）使用BN，很方便；但对于RNN来说，sequence的长度是不一致的，换句话说**RNN的深度不是固定的**，不同的time-step需要保存不同的statics特征，可能存在一个特殊sequence比其他sequence长很多，这样training时，计算很麻烦。

防止过拟合：可以通过BN使数据受batch影响，近似可看作数据增强。

原理

对于神经网络中的第 l 层，我们有：

$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$$

$$\mu = \frac{1}{m} \sum_{i=1}^{m} Z^{[l](i)}$$

$$\sigma^2 = \frac{1}{m} \sum_{i=1}^{m} (Z^{[l](i)} - \mu)^2$$

$$\tilde{Z}^{[l]} = \gamma \cdot \frac{Z^{[l]} - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$$

$$A^{[l]} = g^{[l]}(\tilde{Z}^{[l]})$$

LN

LN不依赖于batch的大小和输入sequence的深度，因此可以用于batchsize为1和RNN中对边长的输入sequence的normalize操作。常用于RNN。

NLP中只在hiden_size这一维上计算，CV中在1,2,3维计算，可学习参数数量是2*hidden_size

a是当前层输出，g和b是可学习参数

$$\mathbf{h} = f(\frac{\mathbf{g}}{\sqrt{\sigma^2 + \epsilon}} \odot (\mathbf{a} - \mu) + \mathbf{b})$$

IN

图像风格中，生成结果主要依赖某个图像实例，所以此时对整个batch归一化不适合了，需要对但像素进行归一化，可以加速模型的收敛，并且保持每个图像实例之间的独立性。

GN

主要是针对Batch Normalization对小batchsize效果差，GN将channel方向分group，然后每个group内做归一化，算(C//G)*H*W的均值，这样与batchsize无关，不受其约束。

GN与LN和IN有关，这两种标准化方法在**训练循环（RNN / LSTM）或生成（GAN）模型**方面特别成功。

SN

- 归一化虽然提高模型泛化能力，然而归一化层的操作是人工设计的。在实际应用中，解决不同的问题原则上需要设计不同的归一化操作，并没有一个通用的归一化方法能够解决所有应用问题；
- 一个深度神经网络往往包含几十个归一化层，通常这些归一化层都使用同样的归一化操作，因为手工为每一个归一化层设计操作需要进行大量的实验。

激活函数

Sigmoid

$$\begin{aligned} \sigma(x) &= \frac{e^x}{1+e^x} = \frac{1}{1+e^{-x}} \\ \sigma'(x) &= \sigma(x)(1-\sigma(x)) \end{aligned} \tag{1}$$

SoftMax

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \tag{2}$$

Loss

熵 Entropy

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

Cross Entropy CE

- 二分类

$$L = \frac{1}{N}\sum_i L_i = \frac{1}{N}\sum_i -[y_i \cdot log(p_i) + (1-y_i) \cdot log(1-p_i)]$$

- 多分类，M为类别数量

$$L = \frac{1}{N}\sum_i L_i = -\frac{1}{N}\sum_i \sum_{c=1}^{M} y_{ic}\log(p_{ic})$$

求导

- 二分类

  分为3个子过程

$$\frac{\partial L_i}{\partial w_i} = \frac{\partial L_i}{\partial p_i} \cdot \frac{\partial p_i}{\partial s_i} \cdot \frac{\partial s_i}{\partial w_i}$$

3.1.4 计算结果 $\frac{\partial L_i}{\partial w_i}$

$$\begin{aligned}
\frac{\partial L_i}{\partial w_i} &= \frac{\partial L_i}{\partial p_i} \cdot \frac{\partial p_i}{\partial s_i} \cdot \frac{\partial s_i}{\partial w_i} \\
&= [-\frac{y_i}{\sigma(s_i)} + \frac{1-y_i}{1-\sigma(s_i)}] \cdot \sigma(s_i) \cdot [1-\sigma(s_i)] \cdot x_i \\
&= [-\frac{y_i}{\sigma(s_i)} \cdot \sigma(s_i) \cdot (1-\sigma(s_i)) + \frac{1-y_i}{1-\sigma(s_i)} \cdot \sigma(s_i) \cdot (1-\sigma(s_i))] \cdot x_i \\
&= [-y_i + y_i \cdot \sigma(s_i) + \sigma(s_i) - y_i \cdot \sigma(s_i)] \cdot x_i \\
&= [\sigma(s_i) - y_i] \cdot x_i
\end{aligned}$$

- 多分类

  如上图所示，求导过程可以分为三个子过程：

$$\frac{\partial L_i}{\partial w_{ic}} = \frac{\partial L_i}{\partial p_{ik}} \cdot \frac{\partial p_{ik}}{\partial s_{ic}} \cdot \frac{\partial s_{ic}}{\partial w_{ic}}$$

  其中，令y_{ik}为1，其余类别为0

  即$L_i = -\log(p_{ik})$

  重点！！！ 即使其他类别不对Loss产生影响，但BP的时候梯度会从y_{ik}传到其他类别

  - 第一项

$$\begin{aligned}\frac{\partial L_i}{\partial p_{ik}} &= \frac{\partial -\log(p_{ik})}{\partial p_{ik}} \\ &= -\frac{1}{p_{ik}}\end{aligned}$$

- 第二项

  分为2类：

  情况1: $c = k$

  则第二项的求导式子，可以写成：

  $$\frac{\partial p_{ik}}{\partial s_{ic}} = \frac{\partial p_{ik}}{\partial s_{ik}}$$

  求导后得

  $$\begin{aligned}\frac{\partial p_{ik}}{\partial s_{ik}} &= \frac{\frac{\partial e^{s_{ik}}}{\partial s_{ik}} \cdot \sum e^{s_{ij}} - e^{s_{ik}} \cdot \frac{\partial \sum e^{s_{ij}}}{\partial s_{ik}}}{(\sum e^{s_{ij}})^2} \\ &= \frac{e^{s_{ik}} \cdot \sum e^{s_{ij}} - e^{s_{ik}} \cdot e^{s_{ik}}}{(\sum e^{s_{ij}})^2} \\ &= \frac{e^{s_{ik}}}{\sum e^{s_{ij}}} - (\frac{e^{s_{ik}}}{\sum e^{s_{ij}}})^2 \\ &= \frac{e^{s_{ik}}}{\sum e^{s_{ij}}} \cdot (1 - \frac{e^{s_{ik}}}{\sum e^{s_{ij}}}) \\ &= p_{ik} \cdot (1 - p_{ik})\end{aligned}$$

情况2: $c \neq k$

此时 s_{ic} 这一项只在分母中存在，求导后得：

$$
\begin{aligned}
\frac{\partial p_{ik}}{\partial s_{ic}} &= \frac{\frac{\partial e^{s_{ik}}}{\partial s_{ic}} \cdot \sum e^{s_{ij}} - e^{s_{ik}} \cdot \frac{\partial \sum e^{s_{ij}}}{\partial s_{ic}}}{(\sum e^{s_{ij}})^2} \\
&= \frac{0 \cdot \sum e^{s_{ij}} - e^{s_{ik}} \cdot e^{s_{ic}}}{(\sum e^{s_{ij}})^2} \\
&= -\frac{e^{s_{ik}} \cdot e^{s_{ic}}}{(\sum e^{s_{ij}})^2} \\
&= -\frac{e^{s_{ik}}}{\sum e^{s_{ij}}} \cdot \frac{e^{s_{ic}}}{\sum e^{s_{ij}}} \\
&= -p_{ik} \cdot p_{ic}
\end{aligned}
$$

最后得到的结果为

3.2.4 计算结果 $\frac{\partial L_i}{\partial w_{ic}}$

$$\frac{\partial L_i}{\partial w_{ic}} = \frac{\partial L_i}{\partial p_{ik}} \cdot \frac{\partial p_{ik}}{\partial s_{ic}} \cdot \frac{\partial s_{ic}}{\partial w_{ic}}$$

情况1: $c = k$

$$\begin{aligned}
\frac{\partial L_i}{\partial w_{ic}} &= \frac{\partial L_i}{\partial p_{ik}} \cdot \frac{\partial p_{ik}}{\partial s_{ic}} \cdot \frac{\partial s_{ic}}{\partial w_{ic}} \\
&= (-\frac{1}{p_{ik}}) \cdot [p_{ik} \cdot (1 - p_{ik})] \cdot x_{ik} \\
&= (p_{ik} - 1) \cdot x_{ik} \\
&= (p_{ik} - y_{ik}) \cdot x_{ik} \\
&= [\sigma(s_{ik}) - y_{ik}] \cdot x_{ik}
\end{aligned}$$

情况2: $c \neq k$

$$\begin{aligned}
\frac{\partial L_i}{\partial w_{ic}} &= \frac{\partial L_i}{\partial p_{ik}} \cdot \frac{\partial p_{ik}}{\partial s_{ic}} \cdot \frac{\partial s_{ic}}{\partial w_{ic}} \\
&= (-\frac{1}{p_{ik}}) \cdot [-p_{ik} \cdot p_{ic}] \cdot x_{ic} \\
&= p_{ic} \cdot x_{ic} \\
&= (p_{ic} - 0) \cdot x_{ic} \\
&= (p_{ic} - y_{ic}) \cdot x_{ic} \\
&= [\sigma(s_{ic}) - y_{ic}] \cdot x_{ic}
\end{aligned}$$

即

$$\frac{\partial L_i}{\partial w_i} = [\sigma(s_i) - y_i] \cdot x_i$$

MSE

$$MSE = \frac{1}{n}\sum_{i}^{n}(\hat{y}_i - y_i)^2$$

KL散度 Kullback-Leibler Divergence

p为观察到的概率分布，要用另一个分布q来近似p。注意：KL散度不是对称的，即$D_{KL}(p||q)! = D_{KL}(q||p)$。

$$D_{KL}(p||q) = \sum_{i=1}^{N} p(x_i) \cdot (\log p(x_i) - \log q(x_i))$$

以下书写方式更常见：

$$D_{KL}(p||q) = \sum_{i=1}^{N} p(x_i) \cdot log\frac{p(x_i)}{q(x_i)}$$

防止过拟合

- 数据增强（Data Augmentation）
- 正则化（L0正则、L1正则和L2正则），也叫限制权值Weight-decay
- Dropout
- Early Stopping
- 简化模型
- 增加噪声
- Bagging
- 贝叶斯方法
- 决策树剪枝
- 集成方法，随机森林
- Batch Normalization

防止欠拟合

- 添加新特征
- 添加多项式特征
- 减少正则化参数
- 增加网络复杂度
- 使用集成学习方法，如Bagging

梯度爆炸梯度消失

原因：

- 梯度消失：隐藏层过多、不合适的激活函数
- 梯度爆炸：隐藏层过多、权重初始值过大、不合适的激活函数

GPU

知乎 https://zhuanlan.zhihu.com/p/51380356

Weight Decay

在设置上，Weight Decay是一个L2 penalty，是对参数取值平方和的惩罚。

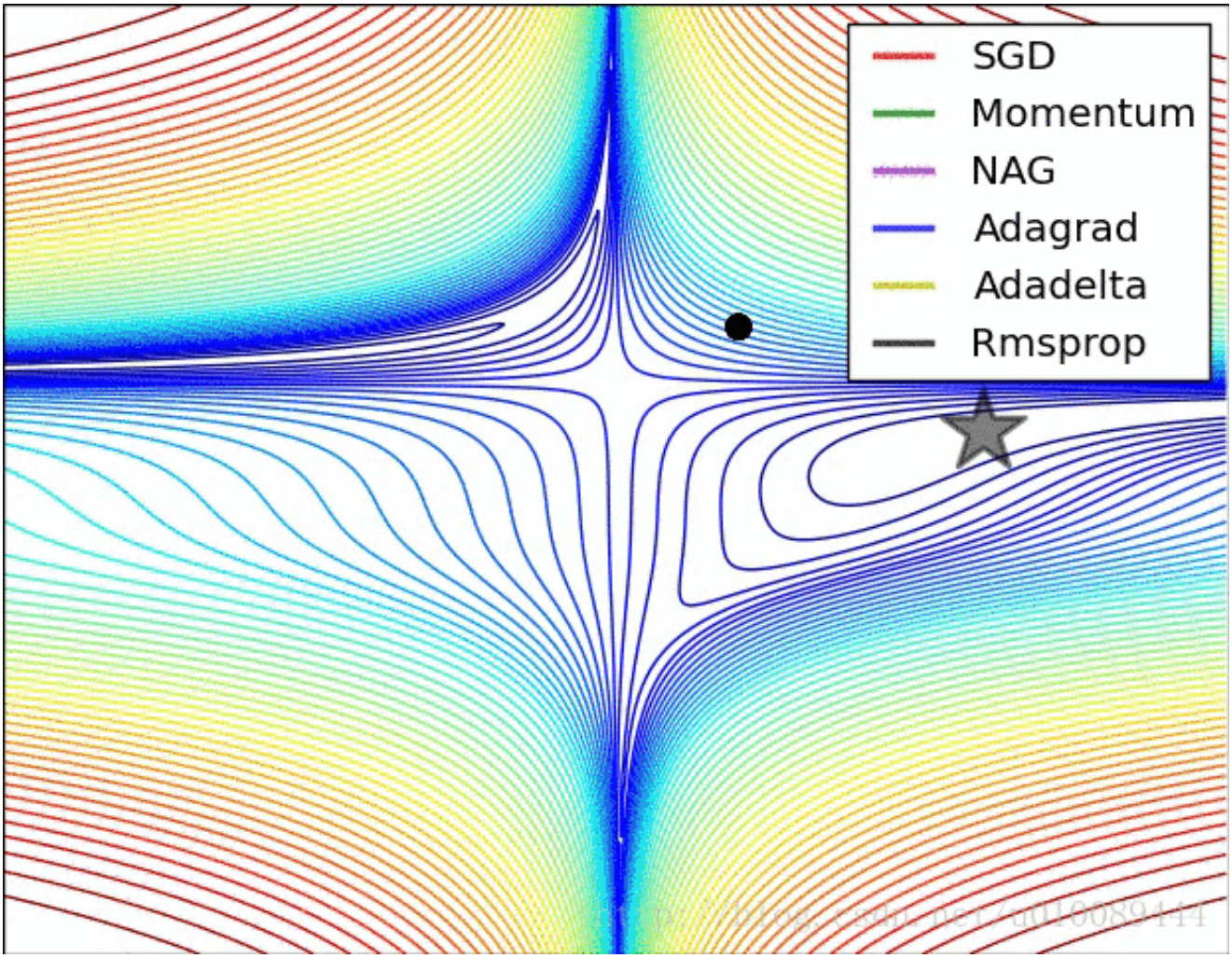
根据Loss function的设置求导之后，我们得到一个公式：

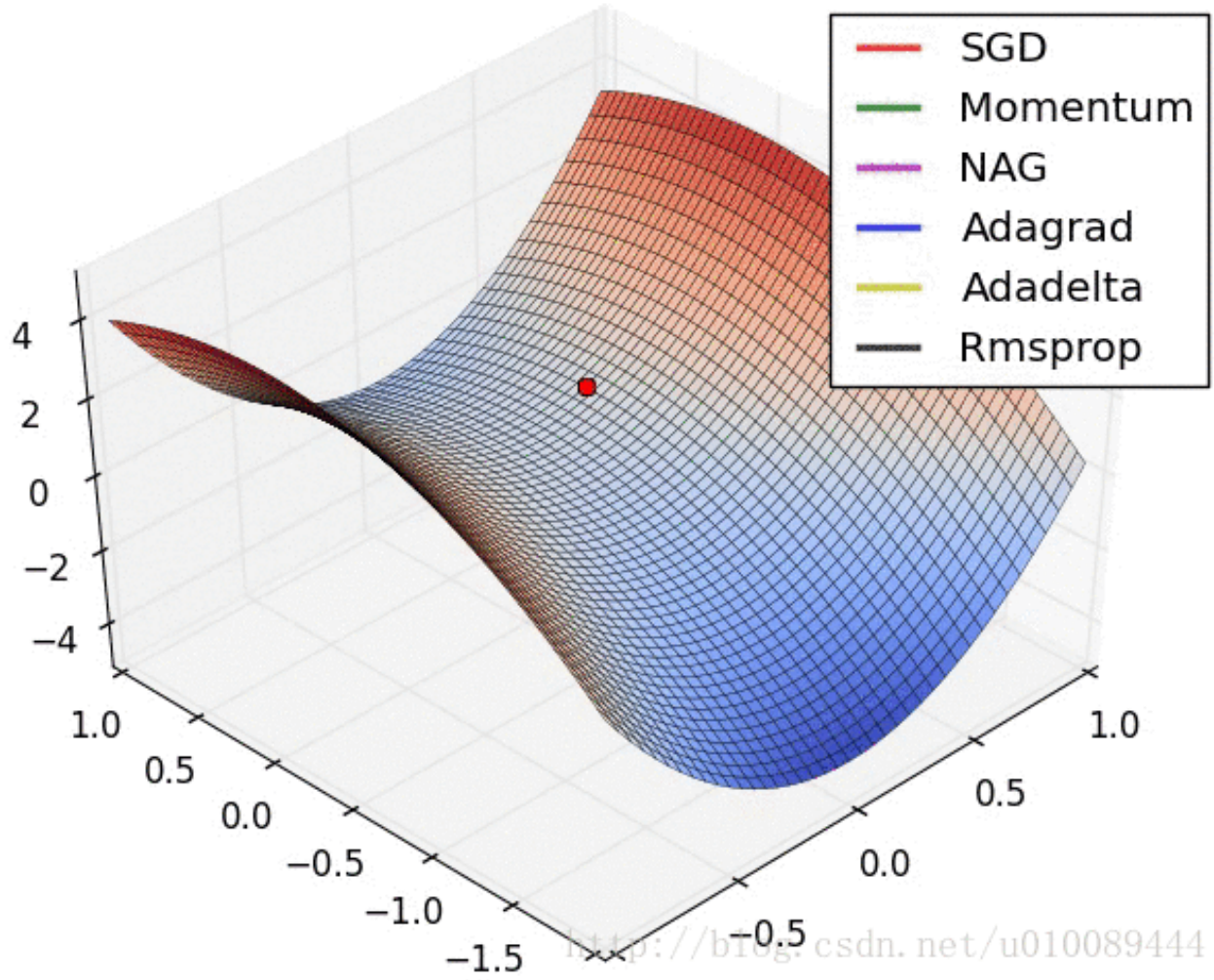
$W(t+1) = W(t) - lr * delta(W) - lr * \mathbf{wd} * W(t)$

也就是说，实际上我们可以把它化简为W(t)首先以（1- lr * wd ）的比例等比例收缩，然后再根据学习率、梯度方向进行一次调整。

优化器

CSDN https://blog.csdn.net/u010089444/article/details/76725843





Momentum SGD

$$v_t = \gamma \cdot v_{t-1} + \alpha \nabla_\Theta J(\Theta)$$
$$\Theta = \Theta - v_t \tag{3}$$

Adam

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

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

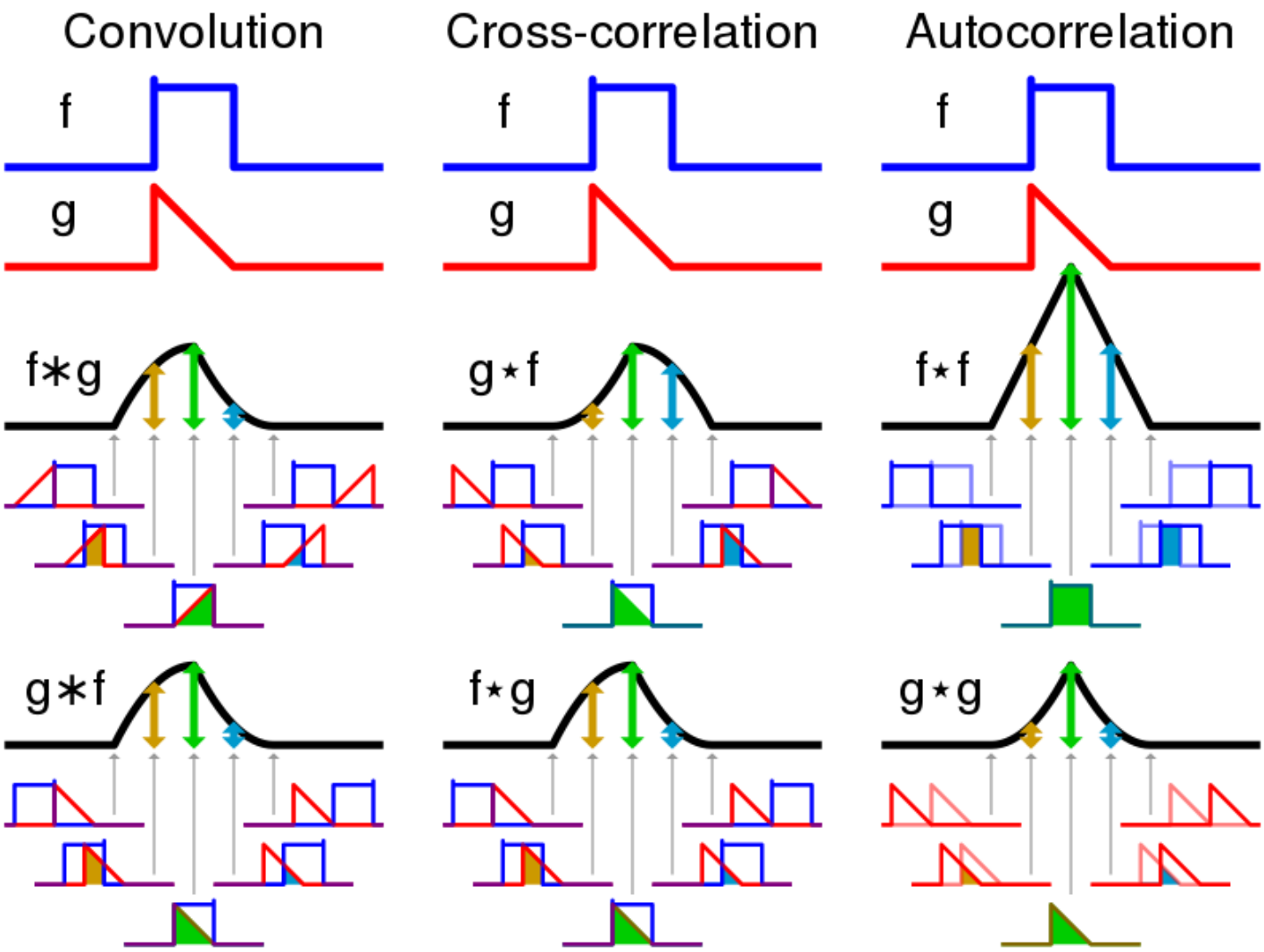
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\Theta_{t+1} = \Theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

adam求一阶和二阶动量，一阶用来确定动量的大小和方向，二阶用来确定当前参数最近的梯度的大小，大的话通过除以这个值来减速，从而使参数的更新保持稳定。

Convolution / Cross-correlation


Convolution
Cross-correlation
Autocorrelation
f
g
f∗g
g⋆f
f⋆f
g∗f
f⋆g
g⋆g