Basic Knowledge

Axis

- 计算时，保留该轴的数量元素；0是横轴，1是纵轴

```
In [1]: import numpy as np
#生成一个3行4列的数组
In [2]: a = np.arange(12).reshape(3,4)
In [3]: a
Out[3]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
#axis= 0 对a的横轴进行操作，在运算的过程中其运算的方向表现为纵向运算
In [4]: a.sum(axis = 0)
Out[4]: array([12, 15, 18, 21])
#axis= 1 对a的纵轴进行操作，在运算的过程中其运算的方向表现为横向运算
In [5]: a.sum(axis = 1)
Out[5]: array([ 6, 22, 38])
```

多层感知机（MLP：Multilayer Perceptron）

- MLP由三层或更多层非线性激活节点组成(一个输入层和一个具有一个或多个隐藏层的输出层)。由于多层互连是完全连接的，所以一层中的每个节点都以一定的权重 w(ij) 连接到下一层的每个节点。
- MLP 在感知器中进行学习，通过每次处理数据后改变连接权重，降低输出与预测结果的误差量。这是有监督学习的一个例子，通过反向传播来实现，反向传播是线性感知器中最小均方算法的推广。

1. 我们可以将输出节点 j 的第 n 个数据点的误差表示为

$$e_j(n) = d_j(n) - y_j(n)$$

其中 d 是目标值，y 是由感知器预测的值。调整节点权重的方式是，尝试通过修正节点权重最小化输出的整体误差

$$\mathcal{E}(n) = \frac{1}{2}\sum_j e_j^2(n)$$

2. 使用梯度下降，每个权重的修正量为

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathcal{E}(n)}{\partial v_j(n)} y_i(n)$$

其中 y(i) 是前一个神经元的输出，η是学习率。η需要精心挑选，保证权重可以快速收敛而不发生震荡。

式中的导数取决于局部场 v(j)。场是变化的。很容易证明输出节点的导数可以简化为

$$-\frac{\partial \mathcal{E}(n)}{\partial v_j(n)} = e_j(n)\phi'(v_j(n))$$

其中

$$\phi'$$

是激活函数的导数,是不变的。对于隐藏节点的权重变化，分析更加困难，但是可以看出相关的导数是

$$-\frac{\partial \mathcal{E}(n)}{\partial v_j(n)} = \phi'(v_j(n)) \sum_k -\frac{\partial \mathcal{E}(n)}{\partial v_k(n)} w_{kj}(n)$$

.

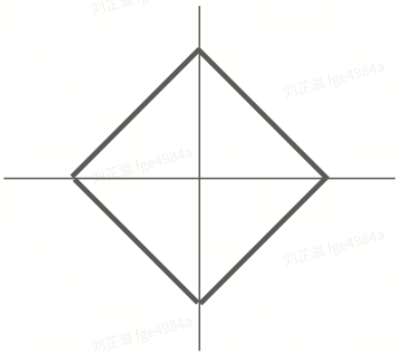
代表输出层的第k个节点的权重变化会影响这个导数。因此，为了改变隐藏层权重，输出层权重根据激活函数的导数而改变，因此该算法代表激活函数的反向传播。

KNN

- 寻找距离给定目标最近的节点，将该节点的结果赋予目标节点
- Distance Metric

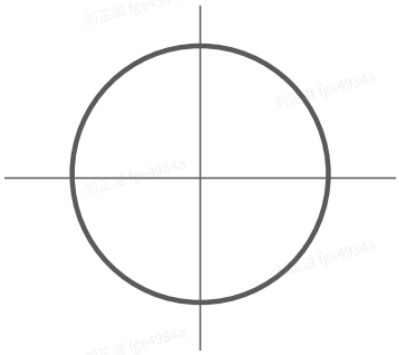
L1 (Manhattan) distance

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$



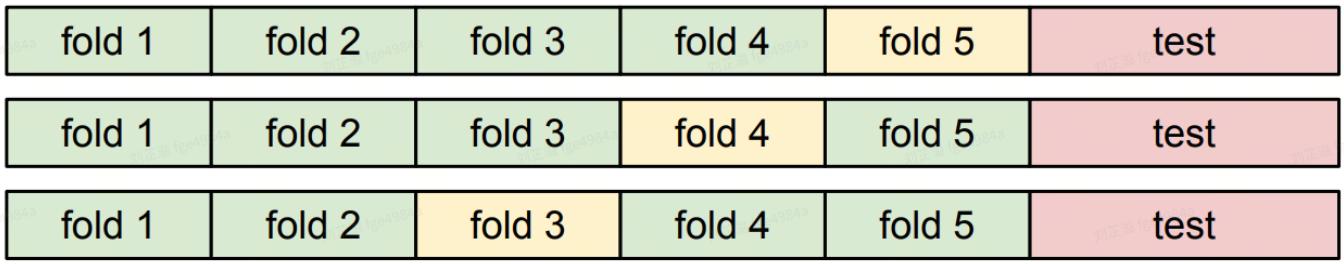
L2 (Euclidean) distance

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$



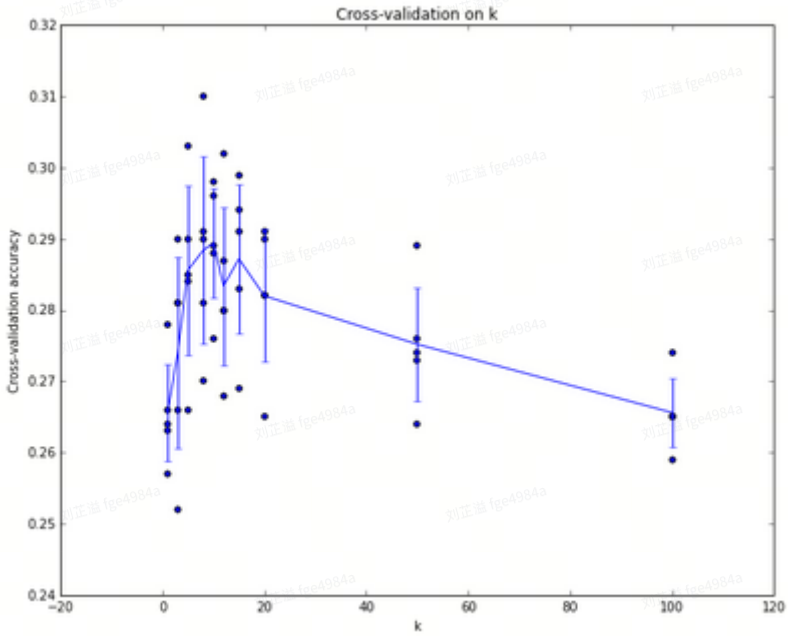
- 超参数：**K和距离函数的选取**
  - 将数据分成train, validation, test三部分
  - 选择在validation上表现好的超参数，在test上预测

  - 或者将数据打乱分组，训练集和验证集交叉排列



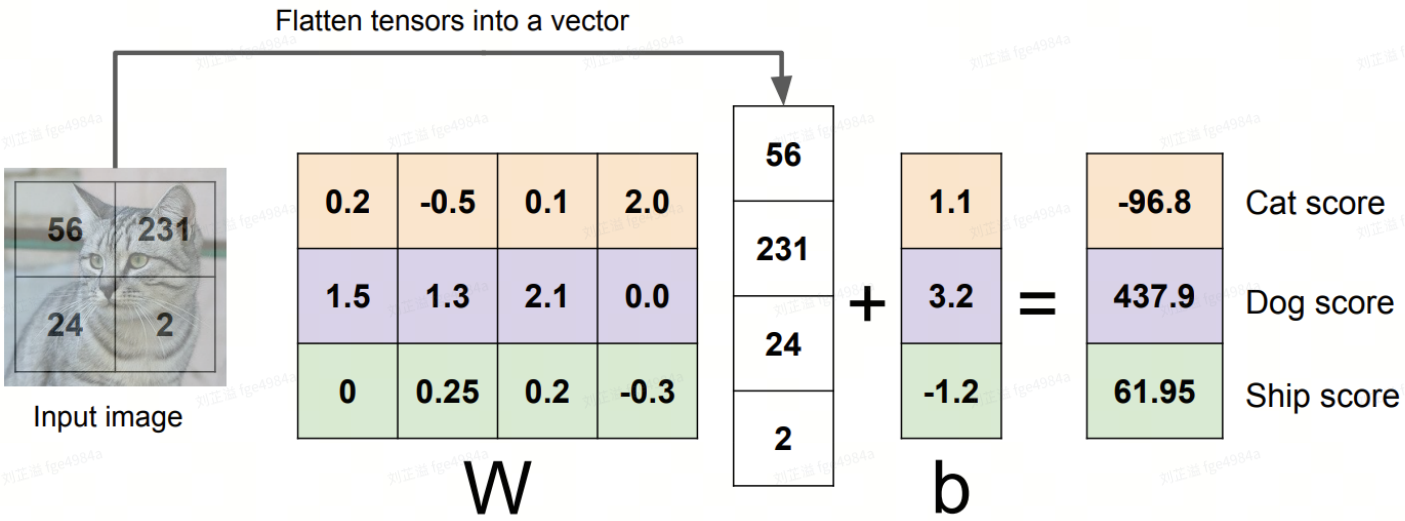
Useful for small datasets, but not used too frequently in deep learning

  - KNN在高维空间运行会出现”**维度诅咒**”的问题，那是因为在高维空间太广阔，高维空间的数据点不趋向接近另外的数据点

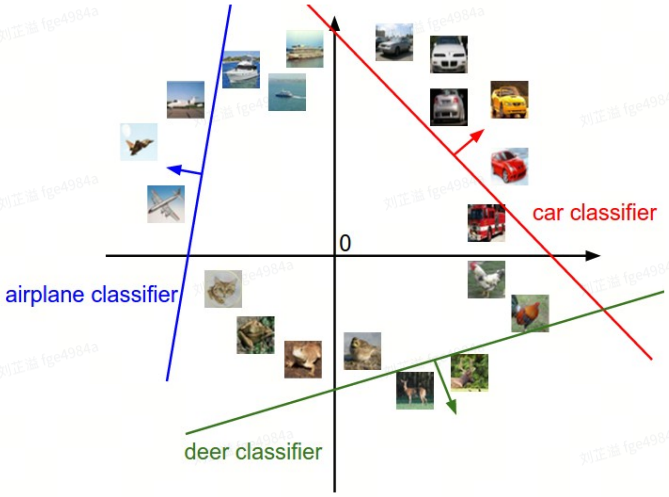


Linear Classifier

- 将图片展开成一个向量



- 选择损失函数和优化方式，寻找使得loss最小的**W**
- 可以看作是高维的图像映射成二维的点



- 可以将线性分类器看作模板匹配；而且是将多个相同类别不同外形的特征作为模板，但对不同颜色的同类别事物鉴别较差
- 图像数据预处理：将图像像素的像素值限定到[-127, 127]，特征限定到[-1, 1]

损失函数

Multiclass SVM loss

and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \begin{cases} 0 & \text{if } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1 & \text{otherwise} \end{cases}$$

$$= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$L = \underbrace{\frac{1}{N}\sum_i L_i}_{\text{data loss}} + \underbrace{\lambda R(W)}_{\text{regularization loss}}$$

Or expanding this out in its full form:

$$L = \frac{1}{N}\sum_i \sum_{j \neq y_i} \left[\max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + \Delta)\right] + \lambda \sum_k \sum_l W_{k,l}^2$$

Relation to Binary Support Vector Machine. You may be coming to this class with previous experience with Binary Support Vector Machines, where the loss for the i-th example can be written as:

$$L_i = C\max(0, 1 - y_i w^T x_i) + R(W)$$

- 求导

$$j \neq y_i, w_j * x_i^T - w_{y_i} * x_i^T + \delta > 0 => \frac{\partial L_i}{\partial w_j} = x_i^T$$

$$j \neq y_i, w_j * x_i^T - w_{y_i} * x_i^T + \delta > 0 => \frac{\partial L_i}{\partial w_{y_i}} = -x_i^T$$

Sofamax classifier (Multinomial Logistic Regression)

- *Maximum Likelihood Estimation* (MLE)：极大似然估计

Want to interpret raw classifier scores as **probabilities**

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax Function

- 损失函数

Maximize probability of correct class

$$L_i = -\log P(Y = y_i|X = x_i)$$

Putting it all together:

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{s_j}}\right)$$

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \qquad \text{or equivalently} \qquad L_i = -f_{y_i} + \log\sum_j e^{f_j}$$

- Softmax分类器最小化估计类别概率之间的交叉熵

$$H(p,q) = H(p) + D_{KL}(p||q)$$

  - 相当于最小化KL散度

Information theory view. The *cross-entropy* between a "true" distribution p and an estimated distribution q is defined as:

$$H(p,q) = -\sum p(x)\log q(x)$$

Practical issues: Numeric stability. When you're writing code for computing the Softmax function in practice, the intermediate terms $e^{f_{y_i}}$ and $\sum_j e^{f_j}$ may be very large due to the exponentials. Dividing large numbers can be numerically unstable, so it is important to use a normalization trick. Notice that if we multiply the top and bottom of the fraction by a constant C and push it into the sum, we get the following (mathematically equivalent) expression:

$$\frac{e^{f_{y_i}}}{\sum_j e^{f_j}} = \frac{Ce^{f_{y_i}}}{C\sum_j e^{f_j}} = \frac{e^{f_{y_i}+\log C}}{\sum_j e^{f_j+\log C}}$$

We are free to choose the value of C. This will not change any of the results, but we can use this value to improve the numerical stability of the computation. A common choice for C is to set $\log C = -\max_j f_j$. This simply states that we should shift the values inside the vector f so that the highest value is zero. In code:

- 求导：y是只有一个元素为1，z是输入；a是经过softmax的结果

（1）当 $i \neq j$

$$\frac{\partial a_j}{\partial z_i} = \frac{0 - e^{z_j} e^{z_i}}{(\sum_k^n e^{z_k})^2} = -a_j a_i$$

$$\frac{\partial L}{\partial z_i} = -a_j a_i * -\frac{1}{a_j} = a_i$$

（2）当 $i = j$

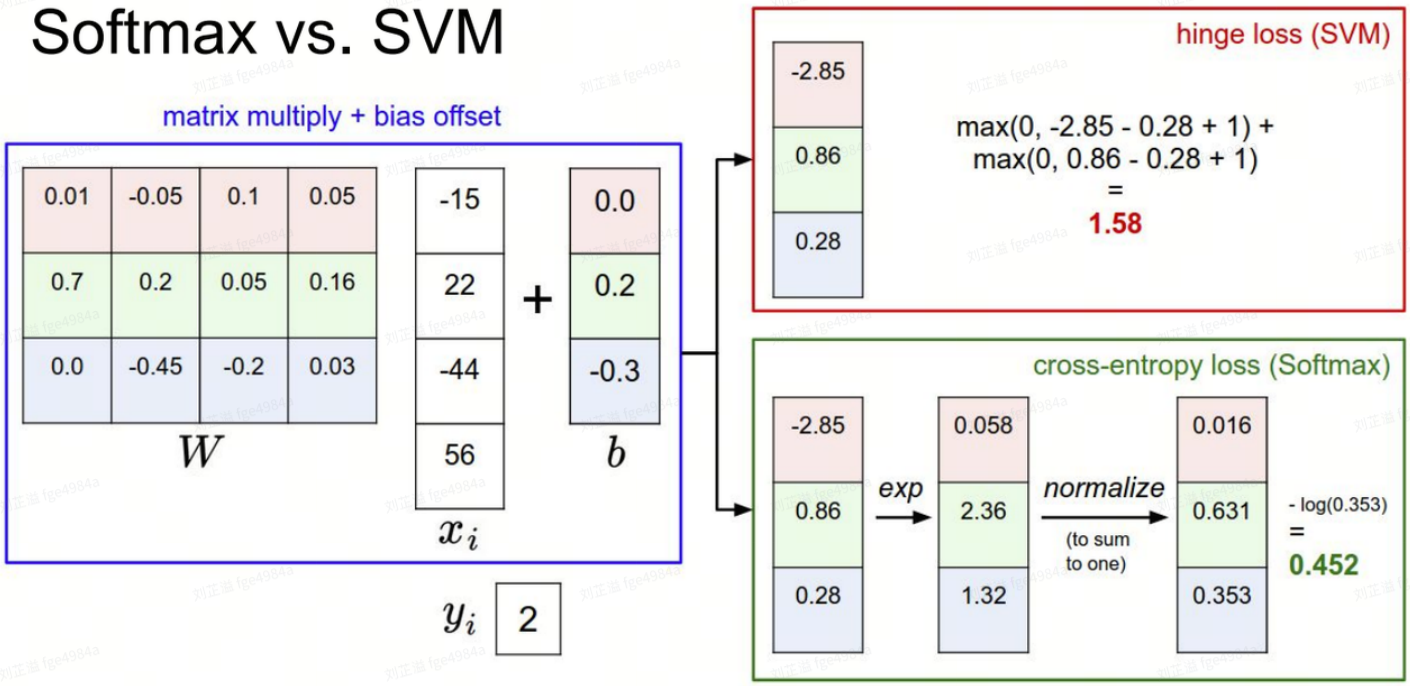
$$\frac{\partial a_j}{\partial z_j} = \frac{e^{z_j} \sum_k^n e^{z_k} - e^{z_j} e^{z_j}}{(\sum_k^n e^{z_k})^2} = a_j - a_j^2$$

$$\frac{\partial L}{\partial z_j} = (a_j - a_j^2) * -\frac{1}{a_j} = a_j - 1$$

所以，$\frac{\partial L}{\partial \boldsymbol{z}} = [a_1, a_2, \ldots, a_j - 1, \ldots a_n] = \boldsymbol{a} - \boldsymbol{y}$ 。

SVM vs. Softmax

- 二者区别不大；但是SVM只在乎正确的类别，其他类别损失均为0；而Softmax则会计算每个类别的损失



Regularization and Optimization

Regularizaion

- 防止模型训练过拟合，增加超参数**λ**
- 减小权重W，进行权重衰减，使网络复杂度降低

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

- 常见的正则化

Simple examples

L2 regularization: $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization: $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2): $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

- L2正则化的作用

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w$$
$$\frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b}.$$

$$\begin{aligned} w &\rightarrow w - \eta \frac{\partial C_0}{\partial w} - \frac{\eta \lambda}{n} w \\ &= \left(1 - \frac{\eta \lambda}{n}\right) w - \eta \frac{\partial C_0}{\partial w}. \end{aligned}$$

- 可以发现L2正则化项对b的更新没有影响，但是对于w的更新有影响:
- **所以 1－ηλ/n小于1，它的效果是减小w，这也就是权重衰减（weight decay）的由来；更小的权值w，从某种意义上说，表示网络的复杂度更低，对数据的拟合刚刚好（这个法则也叫做奥卡姆剃刀）**

Optimization

- Numerical gradient：数值计算，求导来计算梯度
- Analytic gradient：梯度分析
- **Always use analytic gradient, but check implementation with numerical gradient.** This is called a gradient check.
- 梯度下降需要考虑步长：后面叫做learning rate，如何选择合适的lr是一个重大的问题

Mini-batch 梯度下降

- 小batch的梯度是目标梯度的良好近似
- 可以实现更快地收敛

随机梯度下降

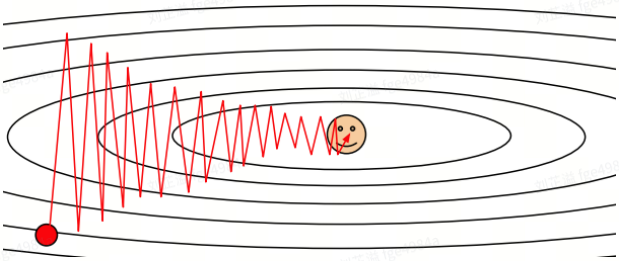
- Stichastic Gradient Descent (SGD)
- 核心思想：迭代精细化，最开始使用随机的W，不断迭代精细化，每次迭代得到更好的W

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \qquad [f(x+h) - f(x-h)]/2h$$

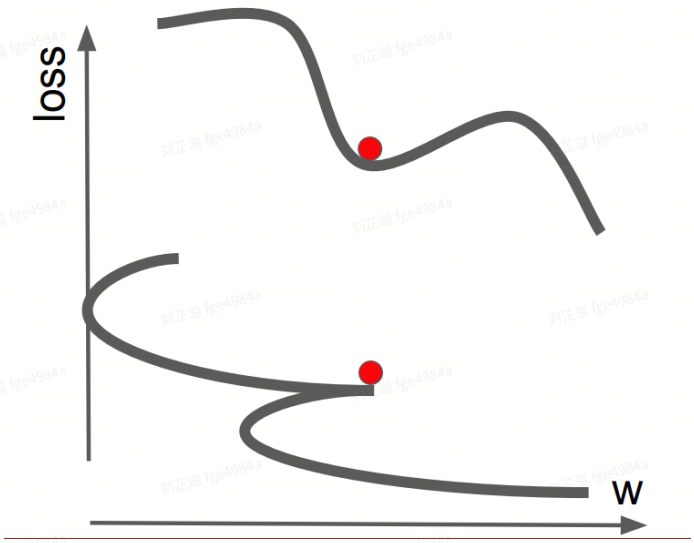
$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

- 出现问题
  - 在某一方向变化剧烈，会造成收敛波动，收敛速度变慢



  - 在局部最小值点（鞍点）收敛，不能达到最优值



  - Our gradients come from minibatches so they can be noisy

SGD

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

SGD + Momentum (动量)：

- 避免在鞍点收敛，尽可能在最优点收敛

SGD+Momentum

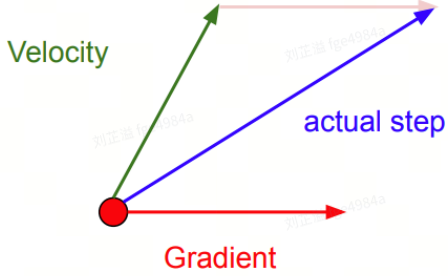
$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

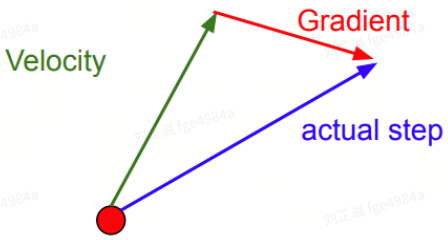
- 两种动量

Momentum update:



Combine gradient at current point with velocity to get step used to update weights

Nesterov Momentum



"Look ahead" to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

Nesterov, "A method of solving a convex programming problem with convergence rate O(1/k^2)", 1983
Nesterov, "Introductory lectures on convex optimization: a basic course", 2004
Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

Nesterov Momentum

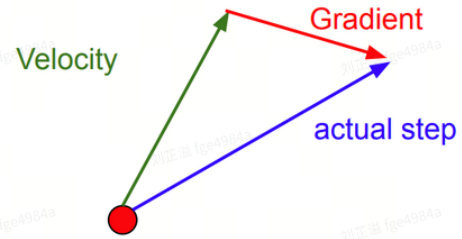
- 校正动量

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$

Change of variables $\tilde{x}_t = x_t + \rho v_t$ and rearrange:

$$v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t)$$
$$\tilde{x}_{t+1} = \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1}$$
$$= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)$$



"Look ahead" to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

AdaGrad

- 根据每个维度的历史平方和数据增加了梯度的逐元素缩放
- G是平方和的对角矩阵

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

- Leaky AdaGrad，解决Adagrad学习率急剧下降问题
- **使用的是指数加权平均，旨在消除梯度下降中的摆动，与Momentum的效果一样，某一维度的导数比较大，则指数加权平均就大，某一维度的导数比较小，则其指数加权平均就小，这样就保证了各维度导数都在一个量级，进而减少了摆动。允许使用一个更大的学习率η**

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2.$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t.$$

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Adam (Almost use)

- Momentum和RMSprop结合起来的一种算法

除了像 Adadelta 和 RMSprop 一样存储了过去梯度的平方 vt 的指数衰减平均值，也像 momentum 一样保持了过去梯度 mt 的**指数衰减平均值**：

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

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

如果 mt 和 vt 被初始化为 0 向量，那它们就会向 0 偏置，所以做了**偏差校正**，通过计算偏差校正后的 mt 和 vt 来抵消这些偏差：

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

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

梯度更新规则：

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t.$$

Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models!