

Machine Learning (ML) 2021

课程: [台湾大学李宏毅ML2021](#)

作业:

Lecture 1: 机器学习/深度学习基本概念简介

机器学习基本概念简介

Machine Learning \approx *Looking for Function*——机器学习就是让机器（程序）具备找一个函数的能力。

Different types of Functions:

- Regression (回归) ——连续。最终得到标量 (scalar)
- Classification (分类) ——离散。得到一个选择 (options/classes)
- 除此两大任务外, 还有**Structured Learning**: 让机器不仅学会分类或者实现预测任务, 而且可以创造特定的“有结构”的物体, 譬如文章、图像等。

机器学习如何找到这个函数? (三个步骤)

- **Function with Unknown Parameters:**

譬如 $y = b + wx_1$, 该假设方程是基于 **domain knowledge** (领域知识) 各种定义:

- **Model**: 带有未知的参数(Parameters)的函数 (function)。
- x_1 是 **feature**, w 是 **weight**, b 是 **bias**, 后两个未知参数基于数据(data)学习得到。

- **Define Loss from Training Data:**

- Loss, 即损失函数, 一个仅带有函数未知的参数的方程, 记作 $L(b, w)$
- Loss 的值体现了函数的一组参数的设定的优劣
- 通过训练资料来计算 $\text{loss} = |\text{估计值} - \text{真正值}|$, **Label** 指的就是正确的数值 \hat{y} ,
$$e_i = |y - \hat{y}|, i = 1, 2, \dots, n$$
, 所以。 **Loss**: $L = \frac{1}{N} \sum_{i=1}^n e_i$ 。其中, 差值 e 的有不同的计算方法, 如上采用直接做差得绝对值 (Mean Absolute Error: MAE), 还有 $e = (y - \hat{y})^2$, 即 Mean Square Error: MSE。选择哪一种方法衡量 e 取决于我们的需求以及对于 task 的理解。
- 我们枚举不同参数组合 (w, b) 通过计算 Loss 值画出等高线图: **Error Surface**
- 如果 y 和 \hat{y} 都是概率 \Rightarrow **Cross-entropy**: 交叉熵, 通常用于分类任务
- loss 函数自定义设定, 如果有必要的话, loss 函数可以 output 负值

- **Optimization**

- $w^*, b^* = \arg \min_{w, b} L$
- 为了实现上述任务 (找到 w, b 使得 L 最小), 通常采用梯度下降法 (**Gradient Descent**)。譬如: 隐去其中一个参数 $w^* = \arg \min_w L$ 从而得到一个 $w - \text{Loss}(L)$ 的数值曲线, 记作 $L(w)$
 - 随机选取一个初始值: w_0
 - 计算: $\frac{\partial L}{\partial w} \Big|_{w=w_0}$, 该点位置在 Error Surface 的切线斜率: 若负值 (Negative), 左高右低 $\Rightarrow w$ 右移 η 使得 Loss 变小; 若正值 (Positive), 左底右高 $\Rightarrow w$ 左移 η 使得 Loss 变小。斜率大 \Rightarrow 步伐 η 跨大一些; 斜率小 \Rightarrow 步伐 η 跨小一些。 $w_1 \leftarrow w_0 - \eta \frac{\partial L}{\partial w} \Big|_{w=w_0}$

η : learning rate 学习率, 属于 **hyper parameters**: 超参数, 自己设定, 决定更新速率。

- 不断迭代更换 w

“假”问题: 囿于局部最优解 local minima, 忽略了实际的最优解 global minima (不过并非梯度下降法的真正痛点)

- 类似的, 将单参数随机梯度下降法推广到两参数上: $w^*, b^* = \arg \min_{w,b} L$

➤ (Randomly) Pick initial values w^0, b^0

➤ Compute

$$\begin{aligned} \frac{\partial L}{\partial w} \Big|_{w=w^0, b=b^0} \\ \frac{\partial L}{\partial b} \Big|_{w=w^0, b=b^0} \end{aligned}$$

$$w^1 \leftarrow w^0 - \eta \frac{\partial L}{\partial w} \Big|_{w=w^0, b=b^0}$$

$$b^1 \leftarrow b^0 - \eta \frac{\partial L}{\partial b} \Big|_{w=w^0, b=b^0}$$

Can be done in one line in most deep learning frameworks

确定更新方向: $(-\eta \frac{\partial L}{\partial w}, -\eta \frac{\partial L}{\partial b})$, η 为学习率

总结来说, 基本步骤如下

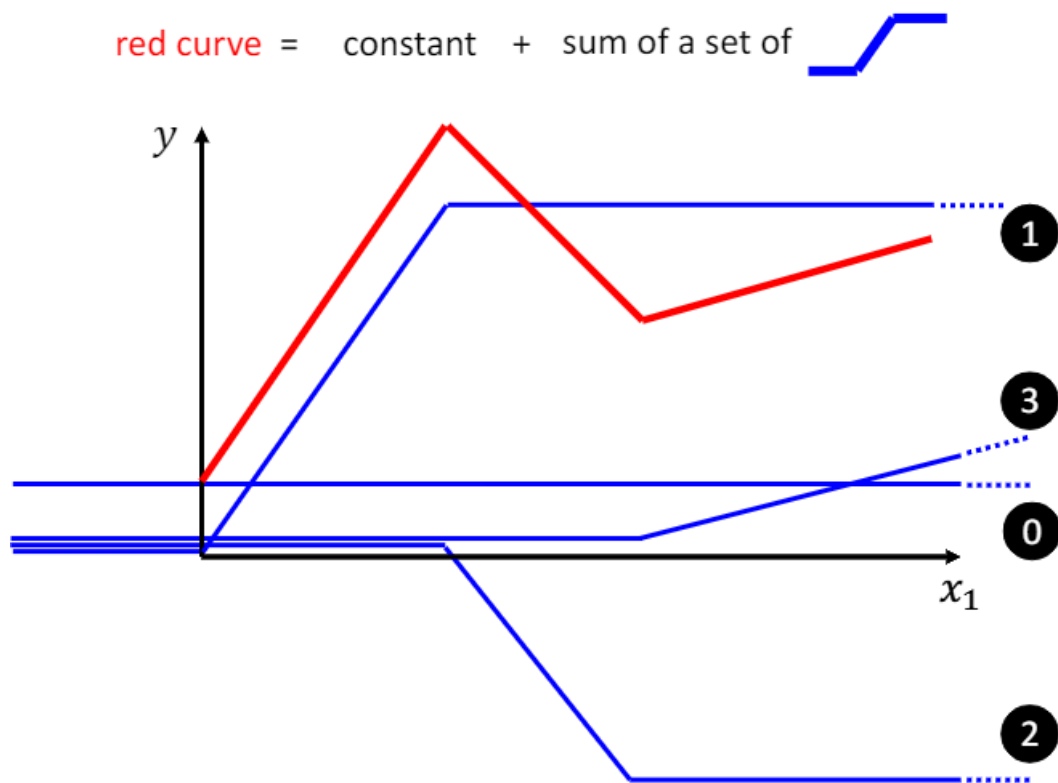


以上三步是机器学习最为基本的框架。基于此, 还需要理解任务, 摸索数据变化规律==>修改模型 (model)

机器学习可以认为是深度学习的超集, 后者是前者在多层网

深度学习基本概念简介

线性模型 (Linear Model) 过于简单, 无论参数组合如何可能总是无法完全拟合任务的 Model, 这里说明 Linear Model 具有 *severe limitation*, 这种局限被称之为 **Model Bias**。于是我们需要更为复杂的函数。

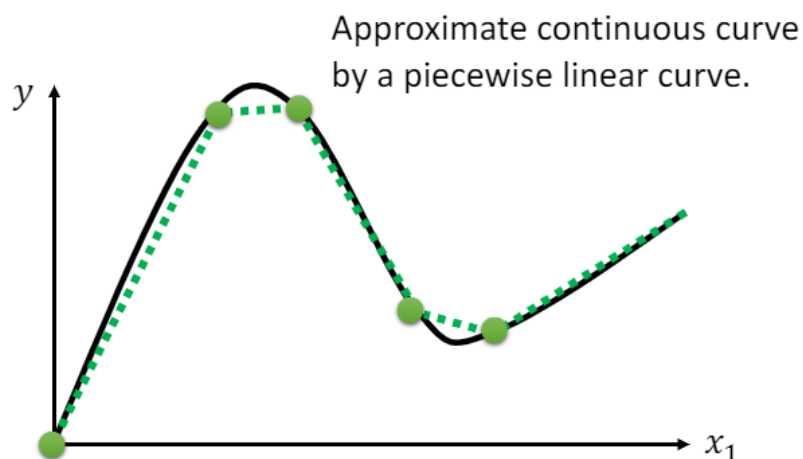


这里类似于使用阶跃函数的组合来表示分段函数, $\text{red curve} = 1 + 2 + 3 + 0$ (常数项), 这里归纳出一个常见的结论: 分段函数 $\text{All Piecewise Linear Curves} = \text{constant}(\text{常数项}) +$


sum of a set of



那么, 对于 *Beyond Piecewise Linear Curves* (这也是我们常见的一般函数的曲线), 我们使用许多多不一样的小线段去“逼近”连续的这条曲线:

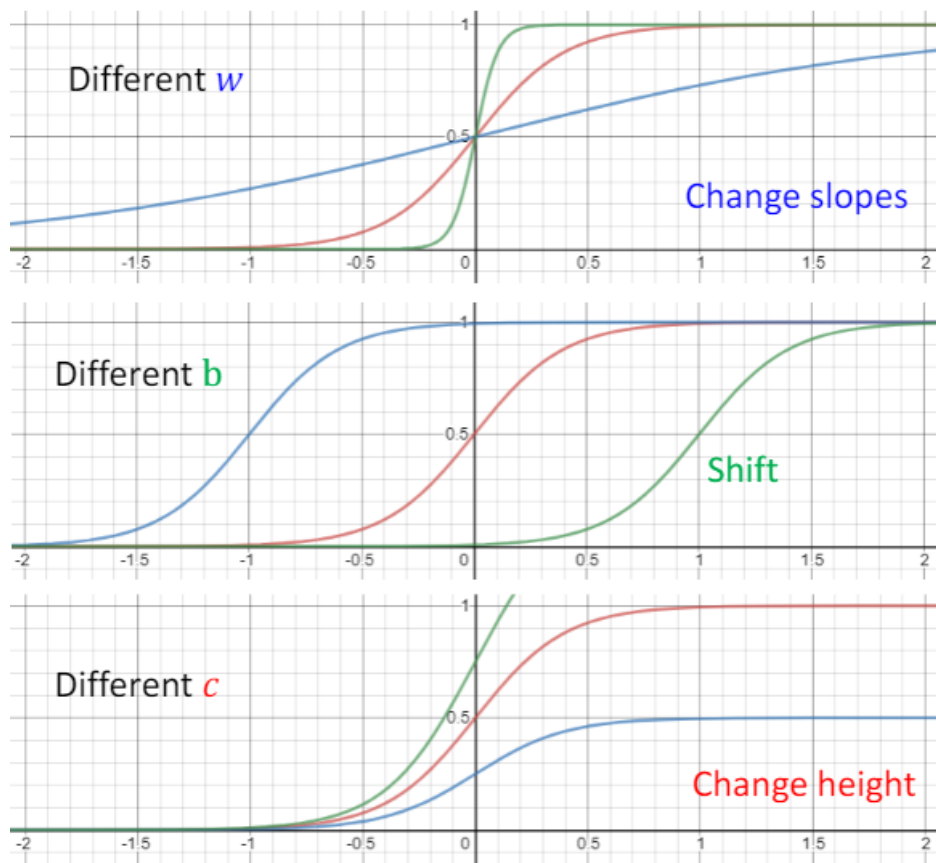


To have good approximation, we need sufficient pieces.

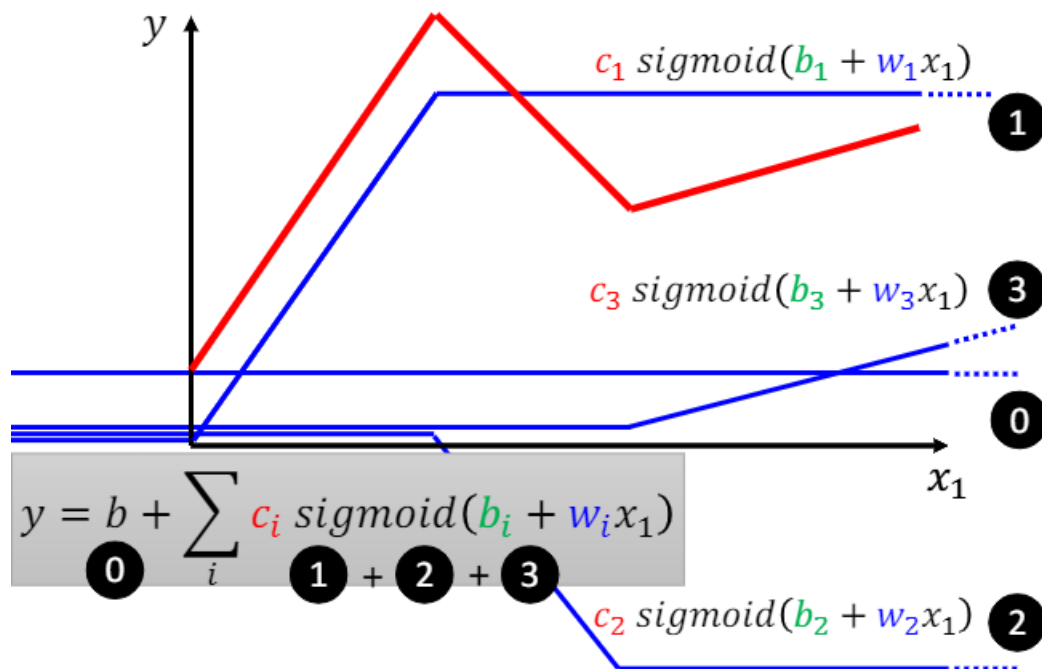
为了表示这样一个蓝色的函数 (小线段)  (被称之为 **Hard Sigmoid**), 这里用一个常见的指数函数来逼近——**Sigmoid Function**

$$y = c \frac{1}{1 + e^{-(b+wx_1)}} = c \cdot \text{sigmoid}(b + wx_1) \quad (1)$$

通过调整 w, b, c , 一组参数组合可以得到不同逼近的小线段 



这个引入超级棒！！由上易知，一个连续的复杂的函数曲线可以被分解成许多离散的小线段（**Hard Sigmoid**）和一个常数项的线性相加，然后每个小线段被一个三参数的**Sigmoid Function**所逼近。下图的函数曲线可以表示为一个含有10个未知参数的mode：



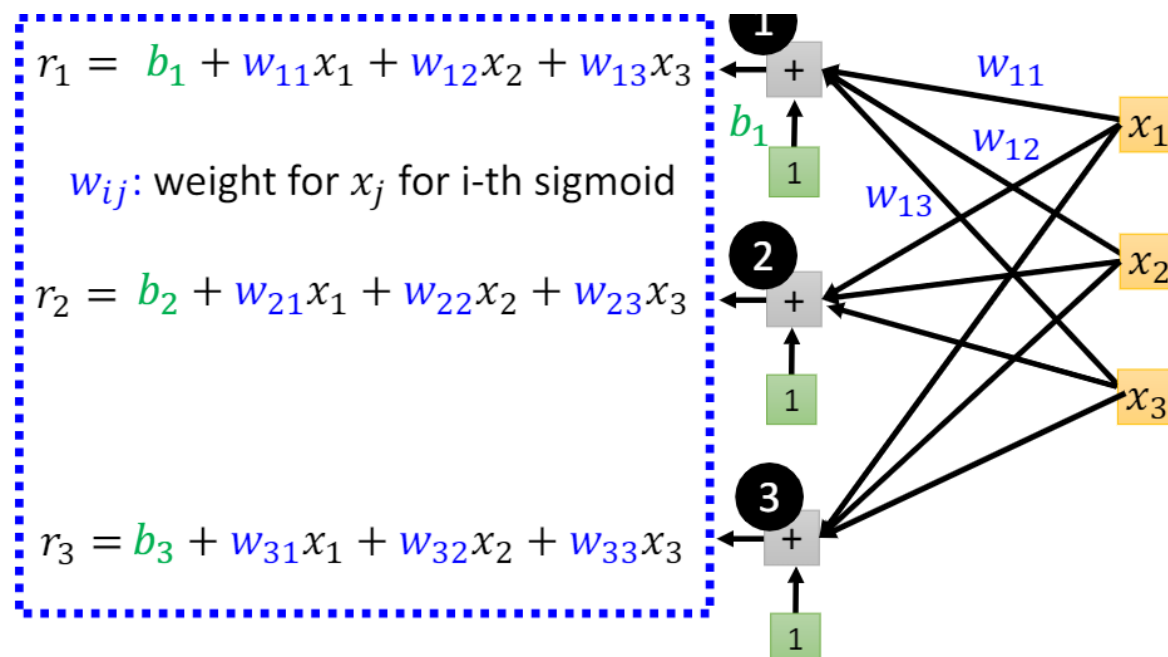
从而，可以产生一个从简单->复杂、单一->多元的函数模式。新的模型包含更多的特征。

$$y = b + wx_1 \Rightarrow y = b + \sum_i c_i \text{sigmoid}(b_i + w_i x_1) \quad (2)$$

由（2）式，考虑到多特征因素，进一步扩展得

$$y = b + \sum_j w_j x_j \Rightarrow y = b + \sum_i c_i \text{sigmoid}(b_i + \sum_j w_{ij} x_j) \quad (3)$$

其中 i 表示 i^{th} 个Sigmoid函数（模型的基函数个数）， x_j 表示一个函数中不同的特征或者预测的数据长度， w_j 表示对应特征权值。



总结：在通用的机器学习教程中，sigmoid函数普遍被视作一款常见的激活函数，在本课程中，从代表任务模型的非线性函数出发-->极限：分段的线性函数组合-->不同性质/特征的sigmoid函数逼近小分割的线性函数。如上图所示，我们有三个激活函数（sigmoid function）以及输出的一个方程组（矩阵/向量相乘表示），这里基本上可以视为一个具有三个神经元的全连接的一层神经网络。

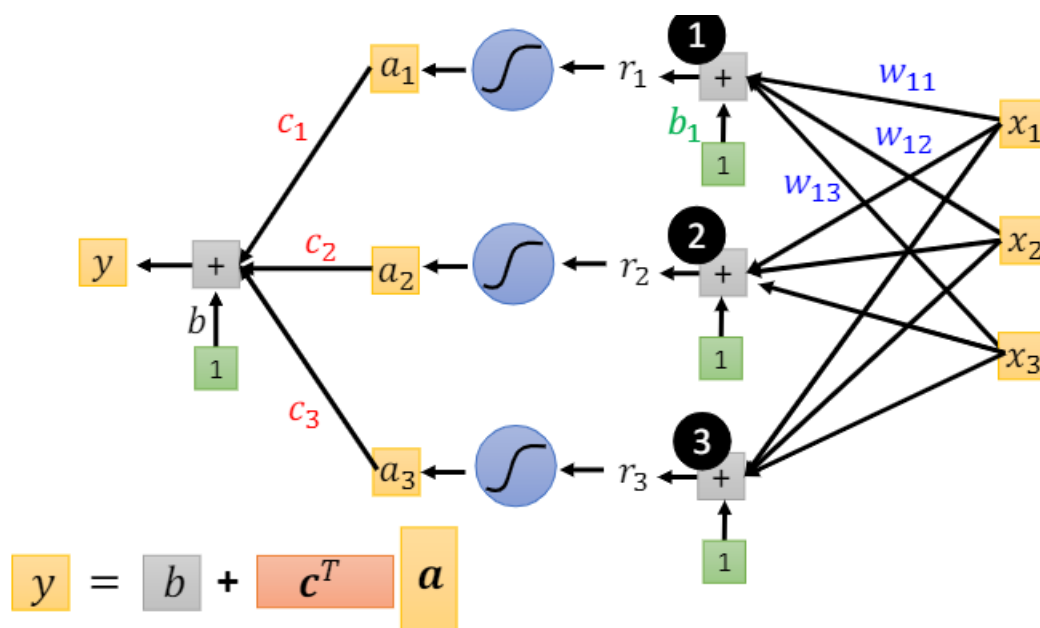
$$[r_1, r_2, r_3]^T = [b_1, b_2, b_3]^T + \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \cdot [x_1, x_2, x_3]^T \quad (4)$$

总之，

$$r = b + w \cdot x \quad (5)$$

接下来，将该方程组 r 通过激活函数输出向量 a ，这里

$$a = \sigma(r) \quad (6)$$



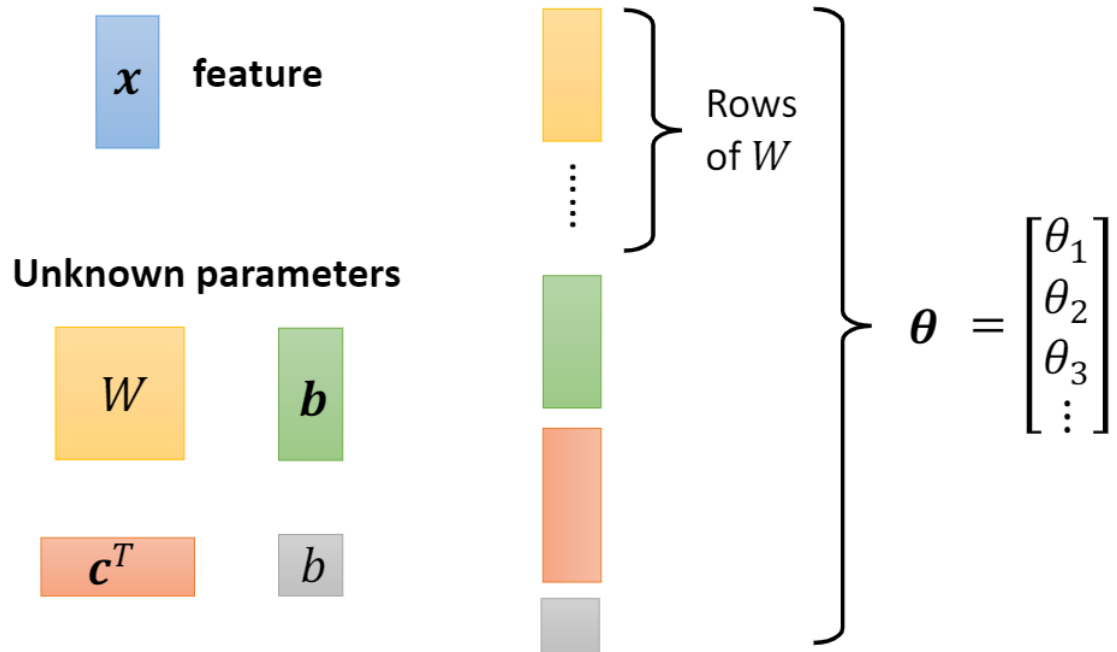
由(5)、(6)得

$$\begin{aligned} \text{由 } a &= \sigma(b + w \cdot x) \\ \Rightarrow y &= b + [c_1, c_2, c_3] \cdot \sigma(b + w \cdot x) \end{aligned} \quad (7)$$

注意， σ 中的 b 是向量，外面的 b 是数值，结果 y 也是数值（标量）。

Step 1: unknown parameters的引入

在上述例子中， x 表示特征， c 、 b 、 W 、 b 为未知参数。为了把未知参数统一起来处理，我们进行如下泛化，比方说， $\theta_1 = [c_1, b_1, w_{11}, w_{12}, w_{13}, b]^T$

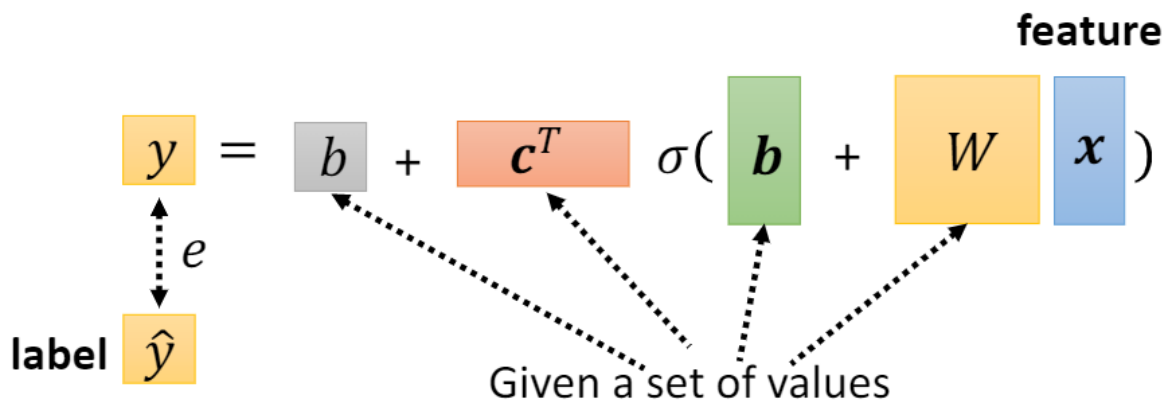


θ 是一个很长的向量，里面的第一个向量为 θ_1 ，以此类推。只要是未知参数都统称在 θ 内。

在参数很少的时候，可以直接穷举参数组合，寻找最优解；但是当机器学习问题中的参数较多时，梯度下降法更为合理。隐含层神经元节点个数（*sigmoid*函数个数）自己决定，其本身个数数值也为超参数之一。

Step 2: 确定loss函数

- loss是一个未知参数的函数： $L(\theta)$
- loss衡量一组参数值表示模型效果优劣



$$\text{Loss: } L = \frac{1}{N} \sum_n e_n$$

同以上介绍的步骤无区别。

Step 3: Optimization

新模型的optimization步骤和之前介绍的无任何区别。对于 $\theta = [\theta_1, \theta_2, \theta_3 \dots]^T$

- 随机选取初始值 θ^0 ，**gradient**梯度记为 $g = [\frac{\partial L}{\partial \theta_1}|_{\theta=\theta^0}, \frac{\partial L}{\partial \theta_2}|_{\theta=\theta^0}, \dots]^T$ ，可简化为 $g = \nabla L(\theta^0)$ 向量长度=参数个数。
- 更新参数 \hookrightarrow (η 当然是学习率啦)

$$\theta = [\theta_1^1, \theta_2^1, \dots]^T \leftarrow \theta = [\theta_1^0, \theta_2^0, \dots]^T - [\eta \frac{\partial L}{\partial \theta_1}|_{\theta=\theta^0}, \eta \frac{\partial L}{\partial \theta_2}|_{\theta=\theta^0}, \dots]^T \quad (8)$$
$$\theta^1 \leftarrow \theta^0 - \eta g$$

不断迭代 $\theta^2 \leftarrow \theta^1 - \eta g, \theta^3 \leftarrow \theta^2 - \eta g, \dots$ ，直到找到不想做或者梯度最后是zero vector（后者不太可能）。

实际上在做梯度下降的时候，我们要把数据 N 分成若干**Batch**（称之为**批量**），如何分？随便分。原先是把所有data拿来算一个loss，现在是在一个Batch上算loss，那么对于 B_1, B_2, \dots 我们可以得到 L^1, L^2, \dots

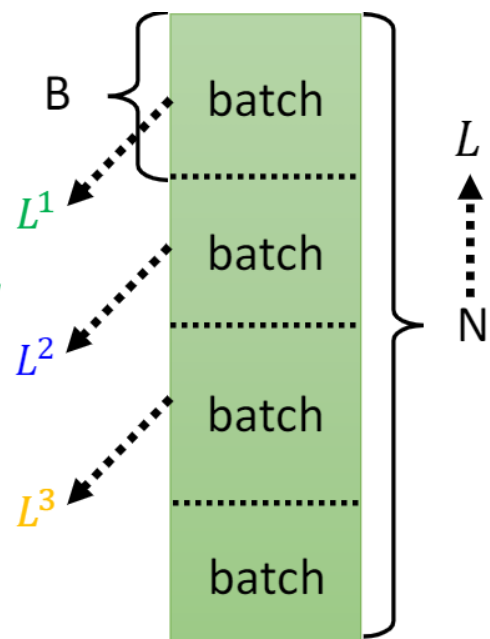
$$\theta^* = \arg \min_{\theta} L$$

➤ (Randomly) Pick initial values θ^0

➤ Compute gradient $g = \nabla L^1(\theta^0)$ L^1
update $\theta^1 \leftarrow \theta^0 - \eta g$

➤ Compute gradient $g = \nabla L^2(\theta^1)$ L^2
update $\theta^2 \leftarrow \theta^1 - \eta g$

➤ Compute gradient $g = \nabla L^3(\theta^2)$ L^3
update $\theta^3 \leftarrow \theta^2 - \eta g$



把所有batch算过一次，称之为一个**epoch**：1 **epoch** = see all the batches once。以上即为**批量梯度下降**。注意区别：一次update指的是每次更新一次参数，而把所有的Batch看过一遍则是epoch。

另外，**Batch Size**大小也是一个超参数。

对模型做更多的变形：

Sigmoid \rightarrow *ReLU*: **Rectified Linear Unit (ReLU)** : $c \cdot \max(0, b + wx_1)$ 曲线。不同的是，我们需要两个*ReLU*曲线才能合成一个**Hard Sigmoid**函数曲线（蓝色的小线段）。无论是*Sigmoid*还是*ReLU*都是**激活函数 (Activation Function)**。

上面的长篇大论仅仅讲述了一层神经网络是如何搭建的，那么多层神经网络的耦合（或者是逐步构建隐藏层） \rightarrow **深度学习 (Deep Learning)**。老师表示这里的层数也是个超参数哦。层数越多，参数越多。

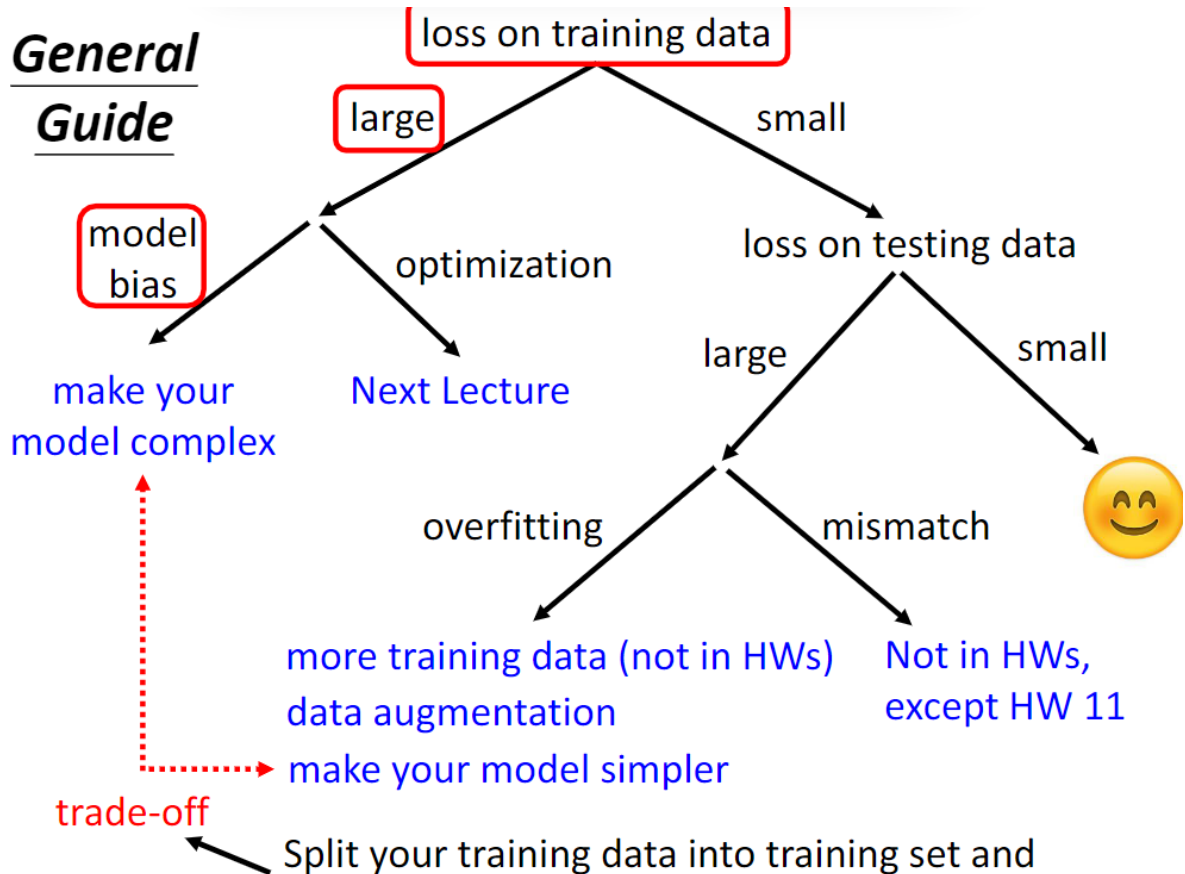
同一层好多个激活函数（Neruon）就是一个hidden layer，多个hidden layer组成了Neural Network。这一整套技术就是deep learning。

之后的神经网络层数越来越多（AlexNet、GoogLeNet等等）那么为什么是深度学习而不是**宽（肥）度**学习？另外，当层数变多了，就会**overfitting (过拟合)**。这些是我们之后课程要讨论的问题。

Lecture 2: 机器学习任务攻略

Training Data \Rightarrow Training (Lecture 1: 三个步骤) \Rightarrow Testing data

General Guide



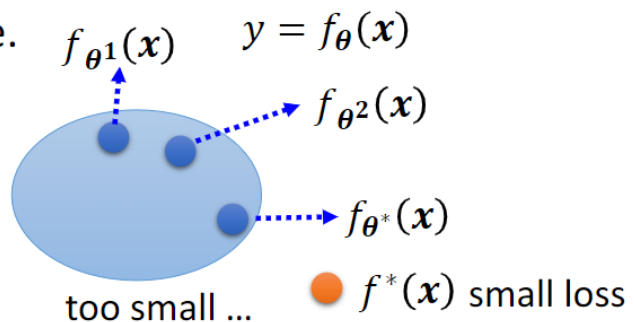
1. 从 loss on training data 着手

1.1 Model Bias

模型过于简单或者与实际相差过多，无论如何迭代，loss值无法降低。需要让模型更加flexible。在课程里层数越多模型越有弹性。

- The model is too simple.

find a needle in a haystack ...
... but there is no needle



- Solution: redesign your model to make it more flexible

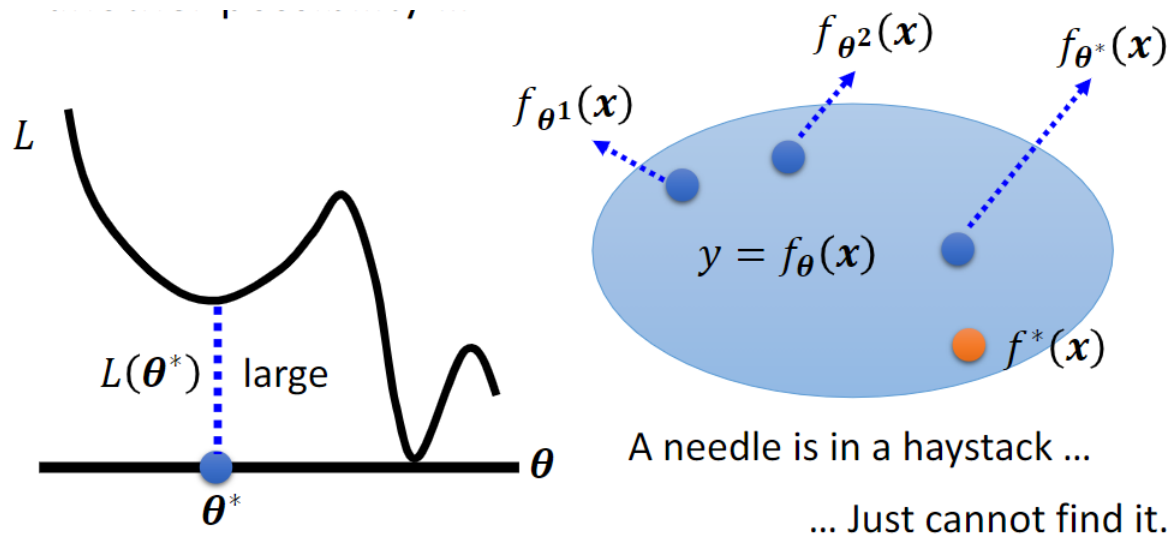
$$y = b + wx_1 \xrightarrow{\text{More features}} y = b + \sum_{j=1}^{56} w_j x_j$$

Deep Learning
(more neurons, layers)

$$y = b + \sum_i c_i \text{sigmoid} \left(b_i + \sum_j w_{ij} x_j \right)$$

1.2 优化问题 (Optimization Issue)

寻找loss陷入局部最优解



关于两者的比较和判断，介绍了文章[Population imbalance in the extended Fermi-Hubbard model](#)。当两个网络A、B，A在B的基础上有更多的层数，但是在任务上A的loss要比B大，这说明A网络的Optimization没有做好。

从对比中，我们可以获得更确切的认知；我们可以从较为浅的model开始着手；如果更深的网络并没有得到更小的loss，那么该网络有optimization issue。

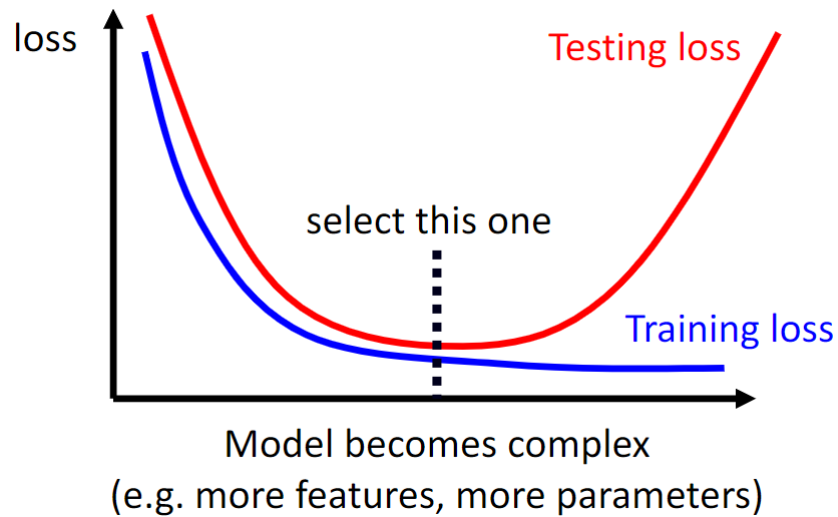
当我们在training data上得到良好的loss，我们就可以着手在testing data上降低loss。

2. 从 loss on testing data 着手

2.1 overfitting 过拟合

- 增加training data (作业里不行)
- Data Augmentation, 根据自己对任务的理解，人为创造出一些新的数据。例如：图像识别训练中可以把训练图片左右翻转，裁剪获得新的训练数据
- 给予模型一定限制，使其不那么flexible
 - 更少的参数
 - 更少的features
 - Early stopping、Regularization、Dropout (Lecture 4)

Bias-Complexity Trade-off: 模型复杂的程度 (或曰模型的弹性) ——function比较多，随着复杂度增加，training的loss越来越小，然而testing的loss是一个凹状的曲线 (先小后大)。



机器学习比赛（例如Kaggle）分为两个Leaderboard：public和private（A、B榜），在两个测试集上的分数的差别过大在于model不够鲁棒。换言之，在公用数据集上达到较高的准确率，不见得在落地使用上能完全实现其测试的level（骗骗麻瓜的商业蜜口）。

每日限制上传次数主要是为了防止各位水模型不断test公用数据集刷分数（无意义~~）

Cross Validation 交叉验证

把training data分成两半：training data和validation data。如何分呢？可以随机分；另外，可以用**N-折交叉验证（N-fold Cross Validation）**



2.2 mismatch

Mismatch表示训练数据和测试数据的**分布（distributions）**不一致。

也可以认为是一种overfitting。通常在预定的机器学习任务中不会出现。

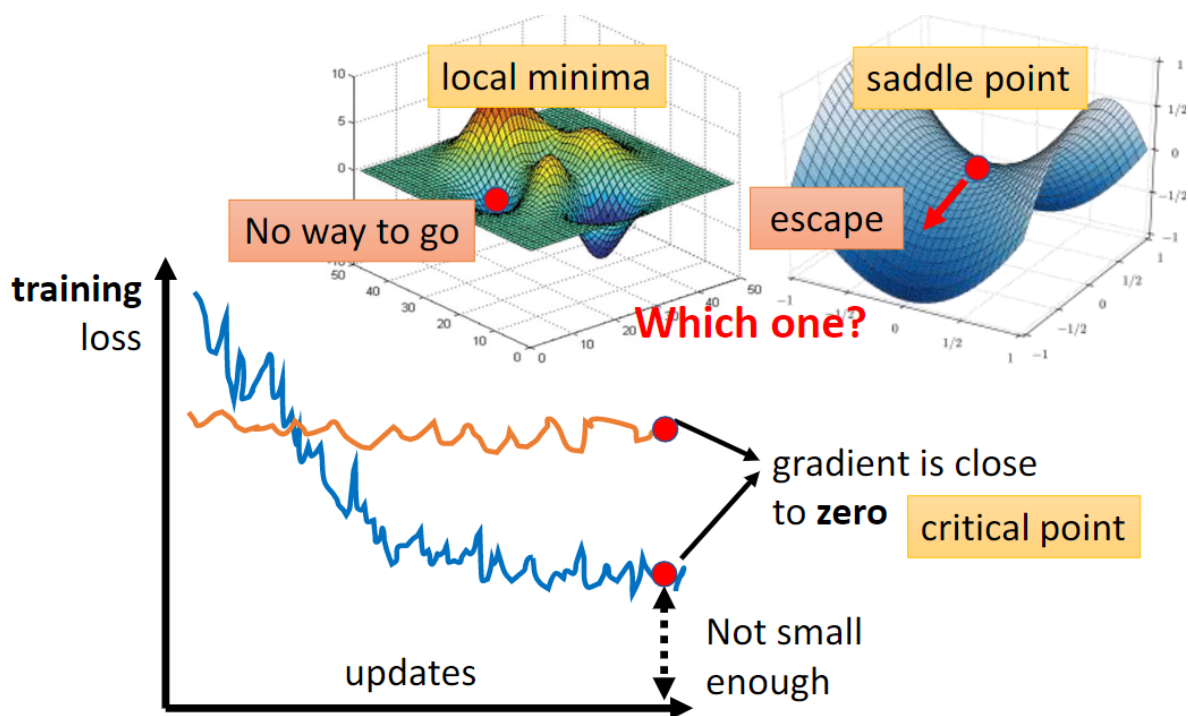
（HW11针对这个问题）

Lecture 2*：如何训练好类神经网络

When gradient is small: Local Minimum and Saddle Point

如果Optimization失败了...——随着不断update而training loss不再下降，你不满意其较小值；或者一开始update时loss下降不下去

Why? ——很有可能update到一个地方（**critical point**），gradient微分后参数为0（或相当接近0）



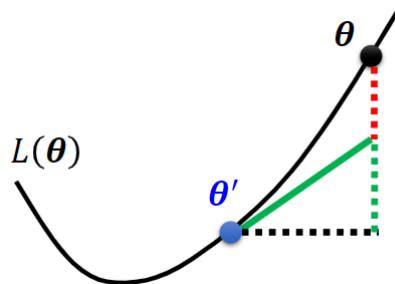
这个点可能是local minima或是saddle point (鞍点)

那么，如何知道这个点 (critical point) 是上述两种的哪一种？(数学上分析如下)

Taylor Series Approximation

对于 $L(\theta)$ ，当 $\theta \approx \theta'$ 时，以下可以约为成立：

$$L(\theta) \approx L(\theta') + (\theta - \theta')^T g + \frac{1}{2}(\theta - \theta')^T H(\theta - \theta')$$



- 梯度Gradient g 是向量，用来弥补 θ 和 θ' 之间的差距。 $g = \nabla L(\theta')$, $g_i = \frac{\partial L(\theta')}{\partial \theta_i}$
- Hessian H 是一个矩阵。 $H_{ij} = \frac{\partial^2}{\partial \theta_i \partial \theta_j} L(\theta')$ ，即 L 的二次微分 (海塞矩阵)

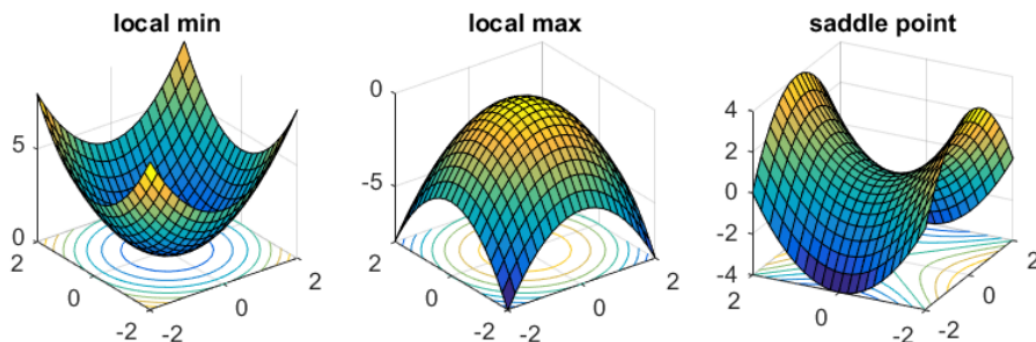
Hessian

$L(\theta)$ around $\theta = \theta'$ can be approximated below

$$L(\theta) \approx L(\theta') + \cancel{(\theta - \theta')^T g} + \frac{1}{2} (\theta - \theta')^T H (\theta - \theta')$$

At critical point

telling the properties of critical points



当梯度 g 为0时, 令 $(\theta - \theta') = v$: ①对于任何可能的 v , 若都有 $v^T H v > 0$, 所以 $L(\theta) > L(\theta')$, 说明是**Local minima**, 等价于 H 是一个称之为**positive definite**的矩阵 (其所有特征值[eigenvalue]为正), 由此也可以判断是否**local minima**; ②对于任何可能的 v , 若都有 $v^T H v < 0$, 所以 $L(\theta) < L(\theta')$, 说明是**Local maxima**, 等价于 H 是一个称之为**negative definite**的矩阵 (其所有特征值[eigenvalue]为负), 由此也可以判断是否**local maxima**; ③对于任何可能的 v , 可能 $v^T H v > 0$, 也可能 $v^T H v < 0$, 说明是**saddle point**. 等价于矩阵有着 H 的特征值有正有负。

所以, 如果更新时走到了saddle point, 这时候梯度为0, 那么就可以看 H : (H 可以告诉我们参数更新的方向)

$$u \text{ 是 } H \text{ 的特征向量, } \lambda \text{ 是 } u \text{ 的特征值. } \Rightarrow u^T H u = u^T (\lambda u) = \lambda \|u\|^2 \quad (*)$$

若 $\lambda < 0$, 那么 $(*) < 0$, $\Rightarrow L(\theta) < L(\theta')$, 这里假设 $\theta - \theta' = u$, 即只要让下一步更新到 $\theta = \theta' + u$, L 就会变小。

如上, 需要计算二次微分, 计算量较大, 所以之后会有计算量更小的方法。

之后, 老师讲了三体里的一个故事 (魔术师, 君士坦丁堡), 淦。。。引入了在高维空间提供参数学习的视角。参数越多, error surface维度越来越高。当在一个相当的维度下做训练任务时, 如果update下去loss不再下降, 大概率是卡在了saddle point上, local minima并没有如此常见。

Tips For training: BATCH and MOMENTUM

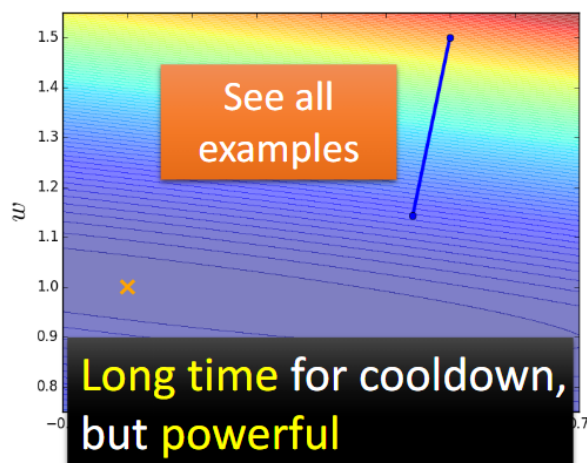
关于BATCH

回顾之前的介绍 (Lecture 1), 1 **epoch** = see all the batches once \rightarrow **Shuffle** after each epoch, 即在每一次epoch开始之前都会分一次batch, 导致每次epoch的batches都不完全一样。Batch大小的设置可以分成两种情况。

Small Batch v.s. Large Batch, 假设总数为 $N=20$:

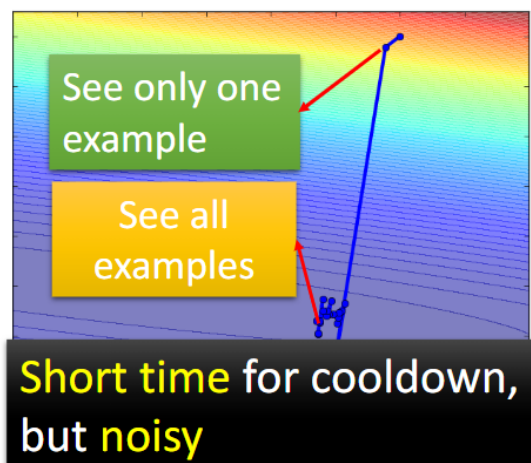
Batch size = N (Full batch)

Update after seeing all the 20 examples






Batch size = 1

Update for each example
Update 20 times in an epoch



两者都很极端，左边就看一遍，蓄力太长；而右边，看一个就蓄力一次，频繁瞬发，方向不定，乱枪打鸟。

算力的进步带来并行计算的能力增强①在如上条件下，epoch较大的batch的训练速度可以更快（反直觉）。②而小一点的batch的Optimization的结果会更好。（可能的解释：loss function是略有差异的，即使update到了critical point，不容易陷入局部最优解）；③在两者batch上train的效果相近，而test结果相差很大（大batch较差），说明发生overfitting。小的batch的泛化性更好些。

| | Small | Large |
|--------------------------------------|--|--|
| Speed for one update (no parallel) | Faster | Slower |
| Speed for one update (with parallel) | Same | Same (not too large) |
| Time for one epoch | Slower | Faster  |
| Gradient | Noisy | Stable |
| Optimization | Better  | Worse |
| Generalization | Better  | Worse |

Batch size是我们决定的超参数。如何确定两者平衡（鱼与熊掌）呢？（提供以下阅读资料可供学习参考）

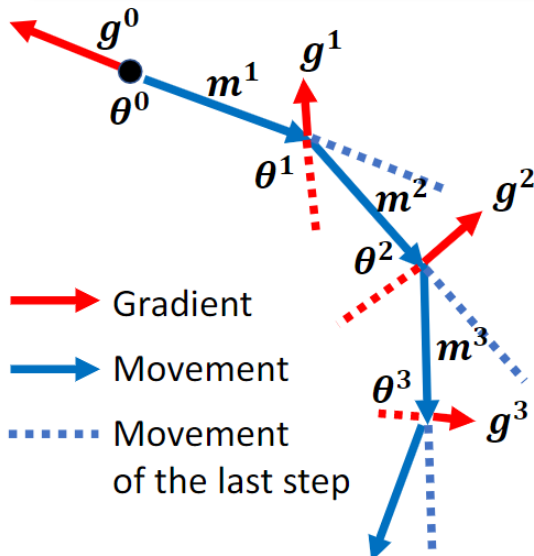
关于Momentum

update时有一个“动量”或惯性，使得接近critical point时，不陷入其中，可以继续update。（不一定会被卡住）

- 一般的Gradient Descent，回顾Lecture 1
- **Gradient Descent + Momentum**

每次移动：不只往gradient反方向移动，同时加上前一步移动的方向，从而调整构成我们的参数。

Movement: movement of last step minus gradient at present



m^i 是所有之前梯度序列 $\{g^0, g^1, \dots, g^{i-1}\}$ 的加权和。

Starting at θ^0

Movement $m^0 = 0$

Compute gradient g^0

Movement $m^1 = \lambda m^0 - \eta g^0$

Move to $\theta^1 = \theta^0 + m^1$

Compute gradient g^1

Movement $m^2 = \lambda m^1 - \eta g^1$

Move to $\theta^2 = \theta^1 + m^2$

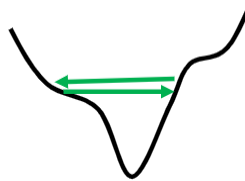
Movement not just based on gradient, but previous movement.

总结一下上两节所学:

- critical points表明该处梯度为0
- critical point可能是saddle point或是local minima: 取决于Hessian matrix; 通过Hessian matrix的特征向量我们可以在梯度为0的点重新更新方向; 另外, local minima可能并不常见
- Smaller batch size以及momentum可以帮助逃开critical points.

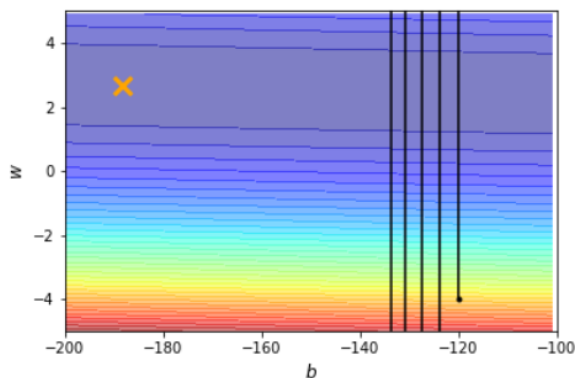
Tips for Training: Adaptive Learning Rate:

引入: Training Stuck \neq Small Gradient. 以下图为例, update后并没有卡在critical point, 而是在两个等高位置“反复横跳”, gradient任然很大, 而loss无法下降。

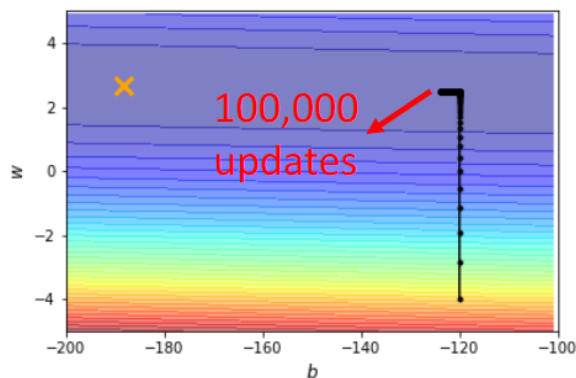


一般的gradient descent的方法下, 在到达critical point之前train就停止了。所以在实做中出现的问题往往不应该怪罪critical point。

由于Learning Rate(LR:学习率)决定每次update的步伐大小, 以下图error surface为例 (目标local minima即是图中橘色小叉叉) learning rate过大, train时一直在两边震荡, loss下降不下去; 当 learning rate较小时, 在梯度较小的地带, 无法得到有效update (走不过去了...)



$$\eta = 10^{-2}$$



$$\eta = 10^{-7}$$

以上说明学习率 (Learning Rate) 不能够 **one-size-fits-all**。应该是，学习率应当为每个参数客质化。
——Different parameters need different learning rate

原来的: $\theta_i^{t+1} \leftarrow \theta_i^t - \eta g_i^t$, $g_i^t = \frac{\partial L}{\partial \theta_i} |_{\theta=\theta^t}$, 改进后: $\theta_i^{t+1} \leftarrow \theta_i^t - \frac{\eta}{\sigma_i^t} g_i^t$ 。我们可以看到从 η 改进为 $\frac{\eta}{\sigma_i^t}$, 分号下的 σ_i^t : 其中不同的参数给出不同的 σ , 同时不同的 iteration 给出不同的 σ , 以上便是 parameter dependent 的 learning rate。

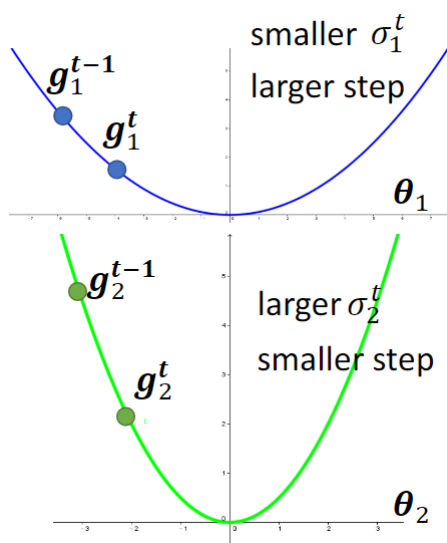
Root Mean Square: σ $\theta_i^{t+1} \leftarrow \theta_i^t - \frac{\eta}{\sigma_i^t} g_i^t$, 以下介绍如何计算 σ

1. 第一步, 当 $t = 0$ 时, $\theta_i^1 \leftarrow \theta_i^0 - \frac{\eta}{\sigma_i^0} g_i^0$, $\sigma_i^0 = \sqrt{(g_i^0)^2} = |g_i^0|$
2. 第二步, 当 $t = 1$ 时, $\theta_i^2 \leftarrow \theta_i^1 - \frac{\eta}{\sigma_i^1} g_i^1$, $\sigma_i^1 = \sqrt{\frac{1}{2}[(g_i^0)^2 + (g_i^1)^2]}$
3. 如上归纳, $\sigma_i^t = \sqrt{\frac{1}{t+1}[(g_i^0)^2 + (g_i^1)^2 + \dots + (g_i^{t-1})^2 + (g_i^t)^2]}$

Adagrad 算法: $\theta_i^{t+1} \leftarrow \theta_i^t - \frac{\eta}{\sigma_i^t} g_i^t$ 且 $\sigma_i^t = \sqrt{\frac{1}{t+1} \sum_{s=0}^t (g_i^s)^2}$

[Deep Learning 最优化方法之AdaGrad - 知乎 \(zhihu.com\)](https://zhuanlan.zhihu.com/p/25000000)

当梯度小时, 根据Adagrad算法, σ 就小, 导致LR较大; 反之, 梯度大, σ 大, LR小。不过它的缺点在于对于稠密的update下, 不断地叠加梯度平方和使得 σ 快速增大而LR随之快速趋于0



另外, 对于具体问题下就算对于同一个参数, 同一个更新方向, LR也被期望可以动态调整——**RMSProp**算法, 来自Hinton在Coursera的授课 (没有论文可引)

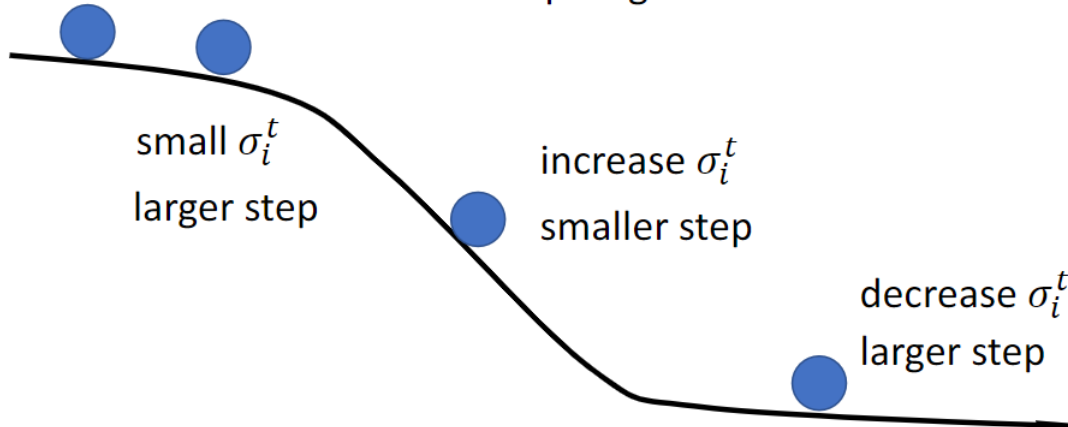
1. 第一步, 当 $t = 0$ 时, $\theta_i^1 \leftarrow \theta_i^0 - \frac{\eta}{\sigma_i^0} g_i^0$, $\sigma_i^0 = \sqrt{(g_i^0)^2} = |g_i^0|$

2. 第二步, 当 $t = 1$ 时, $\theta_i^2 \leftarrow \theta_i^1 - \frac{\eta}{\sigma_i^1} g_i^1$, $\sigma_i^1 = \sqrt{\alpha(\sigma_i^0)^2 + (1 - \alpha)(g_i^1)^2}$, $0 < \alpha < 1$

3. 如上归纳, $\theta_i^{t+1} \leftarrow \theta_i^t - \frac{\eta}{\sigma_i^t} g_i^t$ 时, $\sigma_i^t = \sqrt{\alpha(\sigma_i^{t-1})^2 + (1 - \alpha)(g_i^t)^2}$, $0 < \alpha < 1$

通过 α 这一项, 可以动态调整平衡梯度和前一步 σ 的影响

The recent gradient has larger influence,
and the past gradients have less influence.



目前, 我们最常用的动态调整LR的算法就是**Adam**: RMSProp + Momentum推荐阅读录入ICLR2015的Adam文献。相关算法已经写入pytorch里了 (调包叭\xdm)

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 β_1^t and β_2^t we denote β_1 and β_2 to the power t .

Require: α : Stepsize

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

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

Require: θ_0 : Initial parameter vector

$m_0 \leftarrow 0$ (Initialize 1st moment vector) → for momentum

$v_0 \leftarrow 0$ (Initialize 2nd moment vector) → for RMSprop

$t \leftarrow 0$ (Initialize timestep)

while θ_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 θ_t (Resulting parameters)

事实上在实际操作时, LR并不像我们预期那样很顺利的到达local minima, 而是在梯度较小的地段发生向左右两边“井喷”的现象 (原因没怎么听懂), 因此做出以下优化:

Learning Rate Scheduling: η^t

- **Learning Rate Decay** $\theta_i^{t+1} \leftarrow \theta_i^t - \frac{\eta^t}{\sigma_i^t} g_i^t$, 即让 η 和 σ 一同变化

- **Warm Up** 黑科技?? 总的来说: **LR先变大后变小** (至于要变到多大以及变化的速率[超参数]也是需要调的) DeepLearning远古时期的文章就有Warm Up了, 例如Residual Network【这篇文章LR初始设0.01之后设0.1】、以及Transformer

为什么使用Warm Up会有好一些的训练效果？目前为止没有一个完美的解答。有一个解释是：由于 σ 在Adagrad或是Adam中表现出的主要是统计意义，所以在初始时期其相关统计的数据不够多时，先让其不要过于远离初始点，探索获取更多的情报——到后期累计的数据比较多，所以可以LR大一些。[RAdam](#)有相关更深入的讨论。

LR优化方法的总结

Root Mean Square (RMS)： σ 只考虑了梯度的大小，忽略了方向；而Momentum： m_i^t 还考虑到了梯度的方向。总的来说，momentum表达了历史运动的惯性，而RMS则致力于将梯度下降趋于平缓。

(Vanilla) Gradient Descent

$$\theta_i^{t+1} \leftarrow \theta_i^t - \eta g_i^t$$

Various Improvements

$$\theta_i^{t+1} \leftarrow \theta_i^t - \frac{\eta^t}{\sigma_i^t} m_i^t$$

Learning rate scheduling

Momentum: weighted sum of the previous gradients

Consider direction

root mean square of the gradients

only magnitude

17

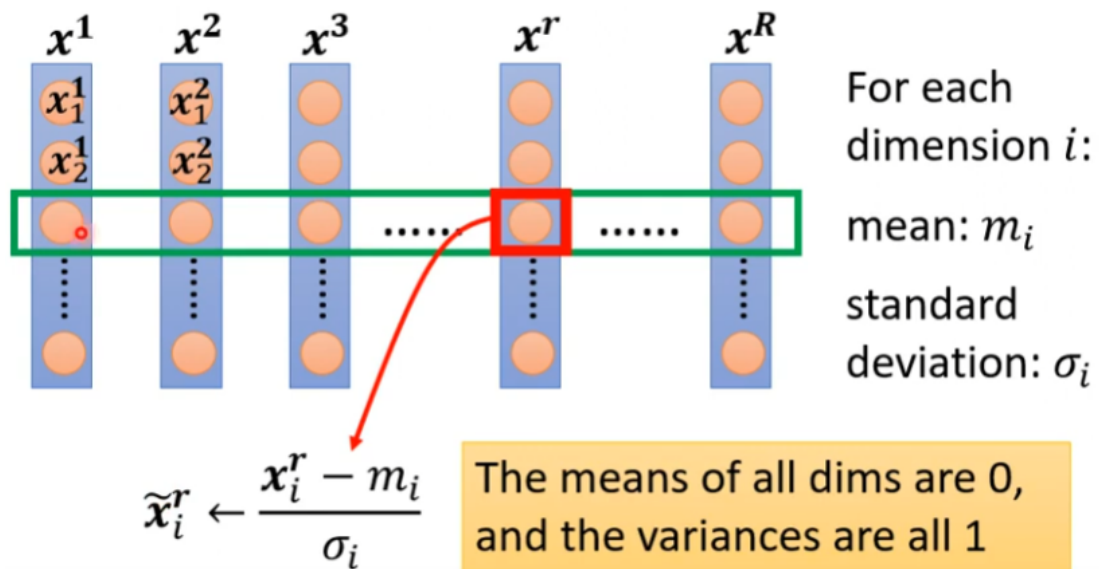
这节主要探讨了在Error Surface坑坑洼洼状态下，如何达成有效优化。下一节则讲授如何优化Error Surface（解决问题的源头？），使其平滑。

Batch Normalization (Quick Introduction)

简短介绍Batch Normalization，以及一些tips==>找到一个满意的Error Surface

由于训练中 x 取值变化很大，所以导致斜率变化“多端”，反差很大，于是使用固定的LR训练效果很差，上一节探讨了如何用优化：动态调整LR。这里介绍下调整range的方法：

- *Feature Normalization*：假设 $x^1, x^2, x^3, \dots, x^r, \dots, x^R$ ：所有训练集的Feature Vector



我们把不同vector下的同一个dimension里面的数值去做一个**平均** m_i ，再做一个**标准差** (standard deviation) 记为 σ_i ，这里就可以做一个**标准化** (Standardization)：

$\tilde{x}_i^r \leftarrow \frac{x_i^r - m_i}{\sigma_i}$ 。好处：同一个dimension上平均值为0，方差为1。在deeplearning里，（小tip）我们可以对特征行做Normalization（即Standardization），这个操作在激活函数前或后都可以，实战上差别不大。

Feature Normalization导致独立输入的初始input相互关联起来，即后面的输出和前面的所有input都有关系（因为input共同决定均值和方差）。有一条弹幕：batch内部每隔sample互相关，batch和batch之间相互独立。

实战中，考虑到GPU的实际内存，我们一般在一个batch上做Feature Normalization，所以这招也叫**Batch Normalization**。当然这会导致batch之间的异质性。

另外，经验之谈， $\tilde{x}^i = \gamma \odot \tilde{x}^i + \beta$ （初始时 γ 为单位向量， β 为零向量）。pyTorch在算Batch Normalization时会把 μ 和 σ 拿出来做moving average。

Batch Normalization用在CNN上，训练速度会变快。

serendipitous（机缘巧合的）discovery

- Layer Normalization
- Instance Normalization
- Group Normalization
- Weight Normalization
- Spectrum Normalization

Lecture 2**：分类 (Classification) BRIEF版

- Regression: $x \Rightarrow \text{model} \Rightarrow y \Leftrightarrow \hat{y}$
- Classification: 奇妙的方法---把分类当作回归

Class as one-hot vector，举例：每个类作为一个 **one-hot vector**

Class 1 Class 2 Class 3

$\hat{y} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ or $\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$ or $\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$

Regression

$$\begin{array}{c} \text{label} \\ \hat{y} \end{array} \longleftrightarrow y = b + c^T \sigma \left(\begin{array}{c} \text{feature} \\ b + Wx \end{array} \right)$$

Classification

$$y = b' + W' \sigma \left(\begin{array}{c} \text{feature} \\ b + Wx \end{array} \right)$$

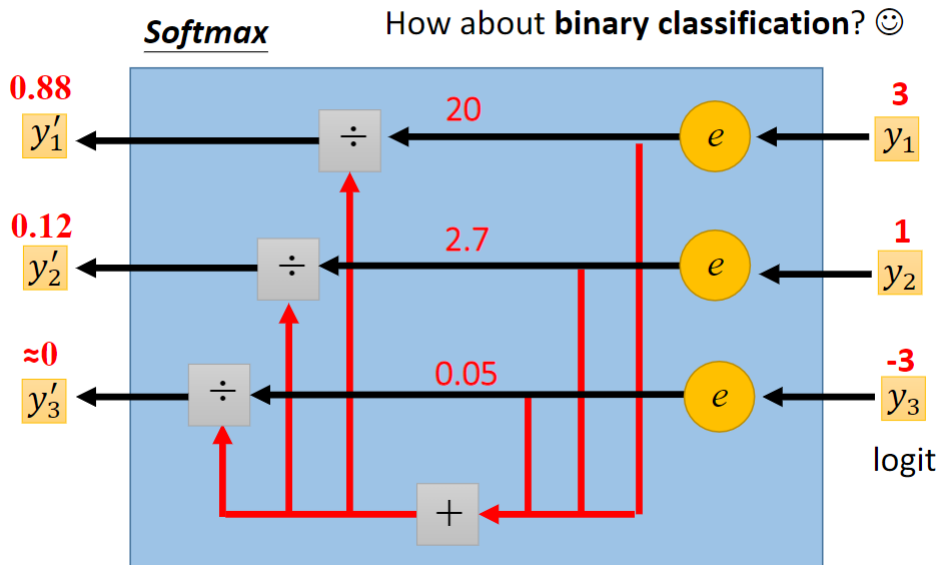
$$\begin{array}{c} \text{label} \\ \hat{y} \end{array} \longleftrightarrow y' = \text{softmax}(y)$$

0 or 1 Make all values between 0 and 1 Can have any value

classification里的y是一个向量（而非数值），另外与Regression不同的是， $y' = \text{softmax}(y)$ 。
 softmax 的作用是将y值映射到[0, 1]里，其原理原因自行探讨。

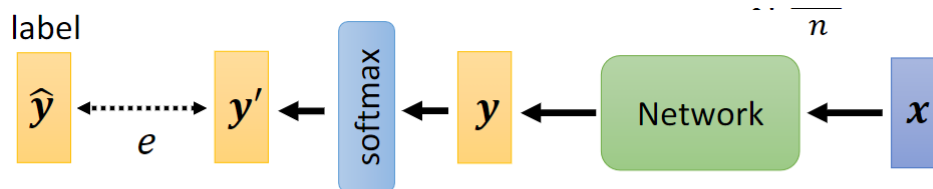
Soft-max

$$y'_i = \frac{\exp(y_i)}{\sum_j \exp(y_j)} \quad \begin{array}{l} \blacksquare 1 > y'_i > 0 \\ \blacksquare \sum_i y'_i = 1 \end{array}$$



除了正则化的效果外，**softmax**还可以让大值和小值差距更大。当只有两个class时，就直接用**sigmoid**了，我们可以认为**softmax**是**sigmoid**的扩展，可以用在三个及以上class的情形。

- **Loss of Classification** $L = \frac{1}{N} \sum_i e_n$ ，以下介绍了MSE和交叉熵



Mean Square Error (MSE) $e = \sum_i (\hat{y}_i - y'_i)^2$

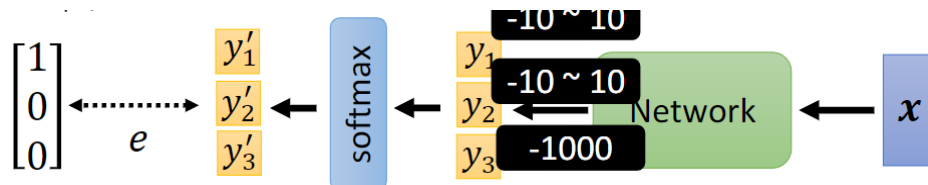
Cross-entropy  $e = - \sum_i \hat{y}_i \ln y'_i$

Minimizing cross-entropy is equivalent to maximizing likelihood.

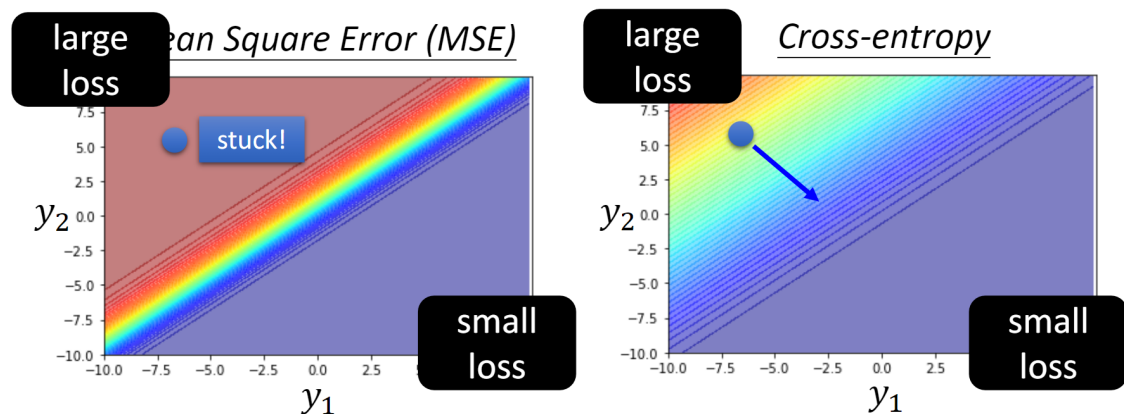
这里交叉熵 (Cross-entropy) 更优, **交叉熵最小 (Minimizing cross-entropy)** 等价于**最大似然 (Maximizing likelihood)**

交叉熵和**softmax**在使用时通常绑定在一起 (pytorch的设计如此)

相比于MSE, cross-entropy更被常使用在分类任务上, 以下从Optimization的角度的解释



这里e可能是MSE或cross-entropy。



Changing the loss function can change the difficulty of optimization.

两者的任务都是从左上角一路到右下角, 但是在MSE上, loss很大的地方非常平坦 (梯度小), 很容易被stuck走不下去; 而cross-entropy则相比起来好很多

###

