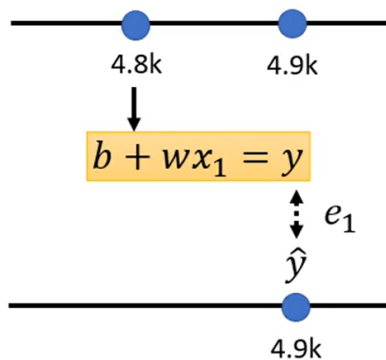


7月10日

一、李宏毅2021春机器学习课程-第一节

2. Define Loss from Training Data

- Loss is a function of parameters $L(b, w)$
- Loss: how good a set of values is.



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

$$e = |y - \hat{y}| \quad L \text{ is mean absolute error (MAE)}$$

$$e = (y - \hat{y})^2 \quad L \text{ is mean square error (MSE)}$$

- 试了不同的参数,然后计算它的Loss, 画出来的这个等高线图叫做**Error Surface**, 是机器学习的第二步。



- 来自model本身的限制叫做**model bias**

•

$$y = b + \sum_i c_i \operatorname{sigmoid}\left(b_i + \sum_j w_{ij} x_j\right) \quad \begin{matrix} i: 1,2,3 \\ j: 1,2,3 \end{matrix}$$

$$\begin{aligned} r_1 &= b_1 + w_{11}x_1 + w_{12}x_2 + w_{13}x_3 \\ r_2 &= b_2 + w_{21}x_1 + w_{22}x_2 + w_{23}x_3 \\ r_3 &= b_3 + w_{31}x_1 + w_{32}x_2 + w_{33}x_3 \end{aligned}$$

$$\begin{bmatrix} r_1 \\ r_2 \\ r_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} + \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$\mathbf{r} = \mathbf{b} + \mathbf{W} \mathbf{x}$$

• Optimization of New Model

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

➤ (Randomly) Pick initial values θ^0

➤ Compute gradient $\mathbf{g} = \nabla L^1(\theta^0)$

update $\theta^1 \leftarrow \theta^0 - \eta \mathbf{g}$

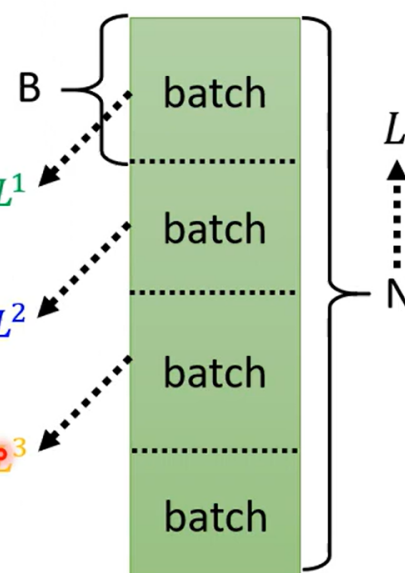
➤ Compute gradient $\mathbf{g} = \nabla L^2(\theta^1)$

update $\theta^2 \leftarrow \theta^1 - \eta \mathbf{g}$

➤ Compute gradient $\mathbf{g} = \nabla L^3(\theta^2)$

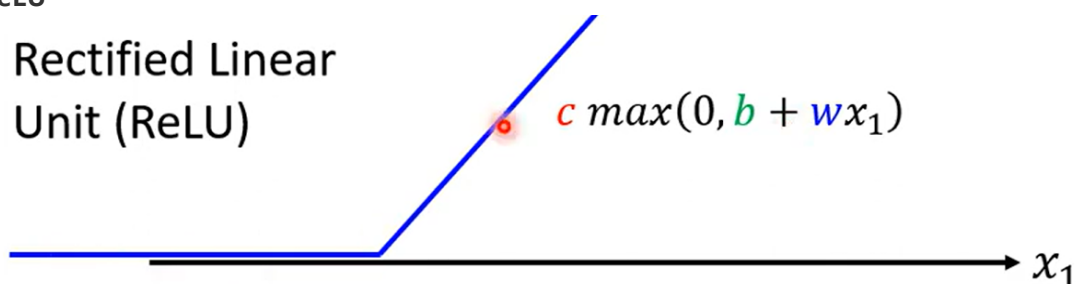
update $\theta^3 \leftarrow \theta^2 - \eta \mathbf{g}$

1 epoch = see all the batches once



• ReLU

Rectified Linear Unit (ReLU)



• ReLU 和 sigmoid 哪一个更好呢?

Sigmoid → ReLU

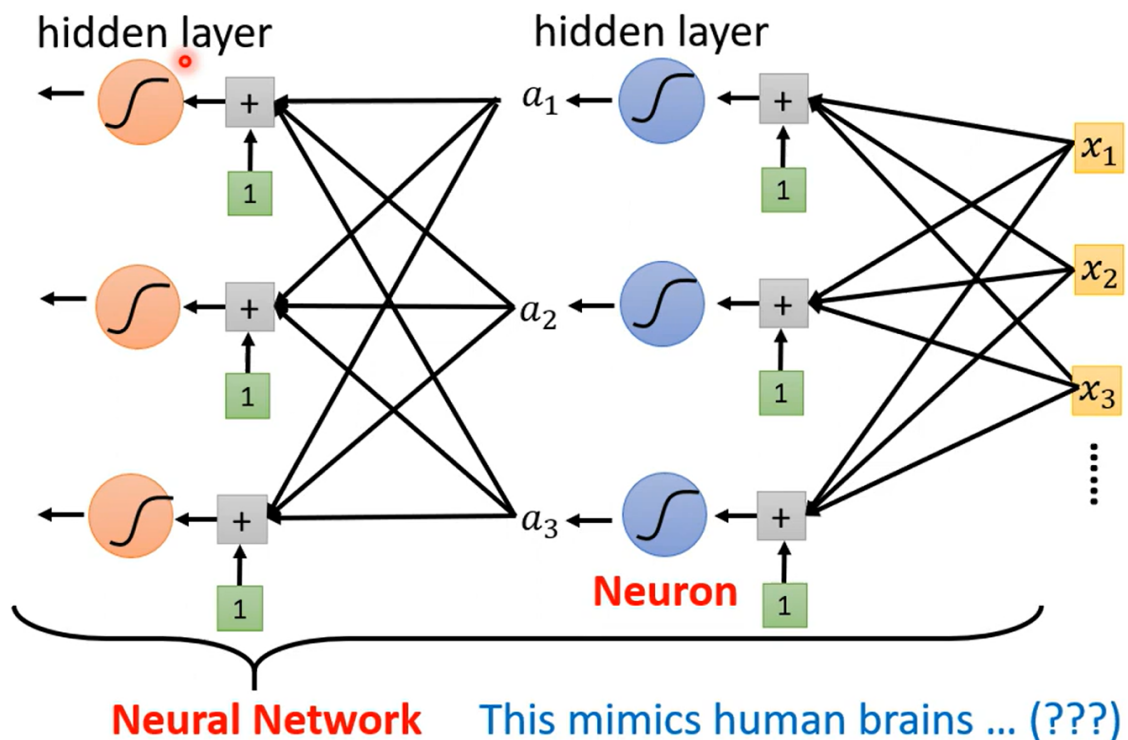
$$y = b + \sum_i c_i \text{sigmoid} \left(b_i + \sum_j w_{ij} x_j \right)$$

Activation function

$$y = b + \sum_{2i} c_i \max \left(0, b_i + \sum_j w_{ij} x_j \right)$$

Which one is better?

- Deep Learning名称的由来



Many layers means **Deep** ➡ Deep Learning



- 过拟合 (overfitting) 问题
- 反向传播 (Backpropagation) : [\(3条消息\) 反向传播——通俗易懂chengchaowei的博客-CSDN博客反向传播](#)以权重参数 w_5 为例，如果我们想知道 w_5 对整体误差产生了多少影响，可以用整体误差对 w_5 求偏导求出：（链式法则）

二、Pytorch Tutorial

• What is PyTorch?

- An open source **machine learning framework**.
- A Python package that provides two high-level features:
 - **Tensor** computation (like NumPy) with strong **GPU acceleration**
 - Deep neural networks built on a **tape-based autograd** system

• PyTorch v.s. TensorFlow

	PyTorch 	TensorFlow 
Developer	Facebook AI	Google Brain
Interface	Python & C++	Python, C++, JavaScript, Swift
Debug	Easier	Difficult (easier in 2.0)
Application	Research	Production

• [一个框架看懂优化算法之异同 SGD/AdaGrad/Adam - 知乎 \(zhihu.com\)](#)

- 首先定义：待优化参数： w ，目标函数： $f(w)$ ，初始学习率 α 。

而后，开始进行迭代优化。在每个epoch t ：

1. 计算目标函数关于当前参数的梯度： $g_t = \nabla f(w_t)$
2. 根据历史梯度计算一阶动量和二阶动量： $m_t = \phi(g_1, g_2, \dots, g_t); V_t = \psi(g_1, g_2, \dots, g_t)$
3. 计算当前时刻的下降梯度： $\eta_t = \alpha \cdot m_t / \sqrt{V_t}$
4. 根据下降梯度进行更新： $w_{t+1} = w_t - \eta_t$

掌握了这个框架，你可以轻轻松松设计自己的优化算法。

- **SGD**：SGD最大的缺点是下降速度慢，而且可能会在沟壑的两边持续震荡，停留在一个局部最优点。
- **SGD with Momentum**：为了抑制SGD的震荡，SGDM认为梯度下降过程可以加入惯性。t时刻的下降方向，不仅由当前点的梯度方向决定，而且由此前累积的下降方向决定。 β_1 的经验值为0.9，这就意味着下降方向主要是此前累积的下降方向，并略微偏向当前时刻的下降方向。想象高速公路上汽车转弯，在高速向前的同时略微偏向，急转弯可是要出事的。
- **SGD with Nesterov Acceleration**：NAG全称Nesterov Accelerated Gradient，是在SGD、SGD-M的基础上的进一步改进，改进点在于步骤1。我们知道在时刻t的**主要下降方向是由累积动量决定的，自己的梯度方向说了也不算，那与其看当前梯度方向，不如先看看如果跟着累积动量走了一步，那个时候再怎么走**。因此，**NAG在步骤1，不计算当前位置的梯度方向，而是计算如果按照累积动量走了一步，那个时候的下降方向**：

$$g_t = \nabla f(w_t - \alpha \cdot m_{t-1} / \sqrt{V_{t-1}})$$

然后用下一个点的梯度方向，与历史累积动量相结合，计算步骤2中当前时刻的累积动量。

- **AdaGrad**：二阶动量的出现，才意味着“自适应学习率”优化算法时代的到来。怎么样去度量历史更新频率呢？那就是二阶动量——该维度上，迄今为止所有梯度值的平方和：

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

我们再回顾一下步骤3中的下降梯度：

$$\eta_t = \alpha \cdot m_t / \sqrt{V_t}$$

可以看出，此时实质上的学习率由 α 变成了 $\alpha / \sqrt{V_t}$ 。一般为了避免分母为0，会在分母上加一个小的平滑项。因此 $\sqrt{V_t}$ 是恒大于0的，而且参数更新越频繁，二阶动量越大，学习率就越小。

这一方法在稀疏数据场景下表现非常好。但也存在一些问题：因为 $\sqrt{V_t}$ 是单调递增的，会使得学习率单调递减至0，可能会使得训练过程提前结束，即便后续还有数据也无法学到必要的知识。

- **AdaDelta / RMSProp**：不累积全部历史梯度，而只关注过去一段时间窗口的下降梯度。这也就是AdaDelta名称中Delta的来历。修改的思路很简单。前面我们讲到，指数移动平均值大约就是过去一段时间的平均值，因此我们用这一方法来计算二阶累积动量：

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

这就避免了二阶动量持续累积、导致训练过程提前结束的问题了。

- **Adam**：谈到这里，Adam和Nadam的出现就很自然而然了——它们是前述方法的集大成者。我们看到，SGD-M在SGD基础上增加了一阶动量，AdaGrad和AdaDelta在SGD基础上增加了二阶动量。把一阶动量和二阶动量都用起来，就是Adam了——Adaptive + Momentum。

SGD的一阶动量：

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

加上AdaDelta的二阶动量：

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

优化算法里最常见的两个超参数 β_1, β_2 就都在这里了，前者控制一阶动量，后者控制二阶动量。

- **Nadam**：最后是Nadam。我们说Adam是集大成者，但它居然遗漏了Nesterov，这还能忍？必须给它加上，按照NAG的步骤1：

$$g_t = \nabla f(w_t - \alpha \cdot m_{t-1} / \sqrt{V_t})$$

这就是Nesterov + Adam = Nadam了。

- [An overview of gradient descent optimization algorithms \(ruder.io\)](https://www.ruder.io/an-overview-of-gradient-descent-optimization-algorithms/)
- 技术的进步，使得傻瓜式操作就可以得到不错的效果，但是在特定的场景下，要拍出最好的效果，依然需要深入地理解光线、理解结构、理解器材。[Adam那么棒，为什么还对SGD念念不忘 \(2\)——Adam的两宗罪 - 知乎 \(zhihu.com\)](#)
 - 主要是后期Adam的学习率太低，影响了有效的收敛。他们试着对Adam的学习率的下界进行控制，发现效果好了很多。于是他们提出了一个用来改进Adam的方法：**前期用Adam，享受Adam快速收敛的优势；后期切换到SGD，慢慢寻找最优解。**
- [Adam那么棒，为什么还对SGD念念不忘 \(3\)——优化算法的选择与使用策略 - 知乎 \(zhihu.com\)](#)

- **先用Adam快速下降，再用SGD调优**，一举两得？思路简单，但里面有两个技术问题：
 1. **什么时候切换优化算法？**——如果切换太晚，Adam可能已经跑到自己的盆地里去了，SGD再怎么好也跑不出来了。
 2. **切换算法以后用什么样的学习率？**——Adam用的是自适应学习率，依赖的是二阶动量的累积，SGD接着训练的话，用什么样的学习率？
- **Useful github repositories using PyTorch**
 - [Huggingface Transformers](#) (transformer models: **BERT**, **GPT**, ...)
 - [Fairseq](#) (sequence modeling for **NLP** & speech)
 - [ESPnet](#) (speech recognition, translation, synthesis, ...)
 - Many implementation of papers

三、优化算法常用的tricks

1. **首先，各大算法孰优孰劣并无定论。**如果是刚入门，**优先考虑SGD+Nesterov Momentum或者Adam**。([Stanford 231n](#) : *The two recommended updates to use are either SGD+Nesterov Momentum or Adam*)
2. **选择你熟悉的算法**——这样你可以更加熟练地利用你的经验进行调参。
3. **充分了解你的数据**——如果模型是非常稀疏的，那么优先考虑自适应学习率的算法。
4. **根据你的需求来选择**——在模型设计实验过程中，要快速验证新模型的效果，可以先用Adam进行快速实验优化；在模型上线或者结果发布前，可以用精调的SGD进行模型的极致优化。
5. **先用小数据集进行实验。**有论文研究指出，随机梯度下降算法的收敛速度和数据集的大小的关系不大。(*The mathematics of stochastic gradient descent are amazingly independent of the training set size. In particular, the asymptotic SGD convergence rates are independent from the sample size. [2]*) 因此可以先用一个具有代表性的小数据集进行实验，测试一下最好的优化算法，并通过参数搜索来寻找最优的训练参数。
6. **考虑不同算法的组合。**先用Adam进行快速下降，而后再换到SGD进行充分的调优。切换策略可以参考本文介绍的方法。
7. **数据集一定要充分的打散 (shuffle)。**这样在使用自适应学习率算法的时候，可以避免某些特征集中出现，而导致的有时学习过度、有时学习不足，使得下降方向出现偏差的问题。
8. 训练过程中**持续监控训练数据和验证数据**上的目标函数值以及精度或者AUC等指标的变化情况。对训练数据的监控是要保证模型进行了充分的训练——下降方向正确，且学习率足够高；对验证数据的监控是为了避免出现过拟合。
9. **制定一个合适的学习率衰减策略。**可以使用定期衰减策略，比如每过多少个epoch就衰减一次；或者利用精度或者AUC等性能指标来监控，当测试集上的指标不变或者下跌时，就降低学习率。

四、HW1 COVID-19 Cases Prediction (Regression)

- `torch.backends.cudnn.deterministic = True` #将这个 flag 置为True的话，每次返回的卷积算法将是确定的，即默认算法。如果配合上设置 Torch 的随机种子为固定值的话，应该可以保证每次运行网络的时候相同输入的输出是固定的
- `torch.backends.cudnn.benchmark = False` #[\(3条消息\) torch.backends.cudnn.benchmark ?!Alan的博客CSDN博客torch.backends.cudnn.benchmark](#)
- `self.data[:, 40:].std(dim=0, keepdim=True)`
- [\(3条消息\) Pytorch \(五\) 入门: DataLoader 和 Dataset 嘿芝麻的树洞-CSDN博客](#)


```
dataloader = DataLoader(dataset, batch_size, shuffle=(mode == 'train'),
drop_last=False, num_workers=n_jobs, pin_memory=True)
```

- 使用了**drop_last = True**, 可能会有以下后果: 比如 test set中有229个数据, 如果batch size = 40, 那么就会有29个数据不会被使用。在每一个epoch中, 就只有200个数据。 $229 - 40 * 5 = 29$
- **num_workers**: 加载数据的时候使用几个子进程
- **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), 而不锁页内存存在主机内存不足时, 数据会存放在虚拟内存中。显卡中的显存全部是锁页内存, 当计算机的内存充足的时候, 可以设置pin_memory=True。当系统卡住, 或者交换内存使用过多的时候, 设置pin_memory=False。因为pin_memory与电脑硬件性能有关, pytorch开发者不能确保每一个炼丹玩家都有高端设备, 因此pin_memory默认为False。
- [\(3条消息\) 机器学习-----L1、L2规范化 \(L1 Regularization、L1 Regularization\) shuangyumelody的博客 CSDN 博客 2 regularization](#)
- [\(3条消息\) 正则化方法: L1和L2 regularization、数据集扩增、dropout wepon的专栏-CSDN博客 L2 regularization](#)
- [\(91 封私信 / 80 条消息\) 怎么选取训练神经网络时的Batch size? - 知乎 \(zhihu.com\)](#)