

1 LSTM/GRU:

LSTM: 主要通过三个门，控制内部状态的更新与输出

(都是 sigmoid 函数，是当前输入和上一状态输出的函数，学习得到参数
门控都是 element-wise, 控制每一维的保留比例)

- At each time step, perform the following operations

Input: controls how much new cell info is written to cell

$$i_t = \sigma(W^{(i)}x_t + U^{(i)}h_{t-1})$$

Forget: controls how much previous cell info should be forgotten

$$f_t = \sigma(W^{(f)}x_t + U^{(f)}h_{t-1})$$

Output: controls how much info is written to hidden state

$$o_t = \sigma(W^{(o)}x_t + U^{(o)}h_{t-1})$$

New cell info: new content to write to cell

$$\tilde{c}_t = \tanh(W^{(c)}x_t + U^{(c)}h_{t-1})$$

Cell state (Memory): forget some of the previous cell info and write some new cell info

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$

Hidden state: output some bits of info from the cell state for next cell to "remember"

$$h_t = o_t \circ \tanh(c_t)$$

知乎 @乐天

© 2018 National University of Singapore. All Rights Reserved

i:输入门，控制新的内部状态有多少需要被累加。新状态类似于 RNN，由当前输入和之前状态得到

f:遗忘门，控制旧的内部状态，有多少需要被遗忘。

在遗忘门和输入们共同作用下，更新内部状态。

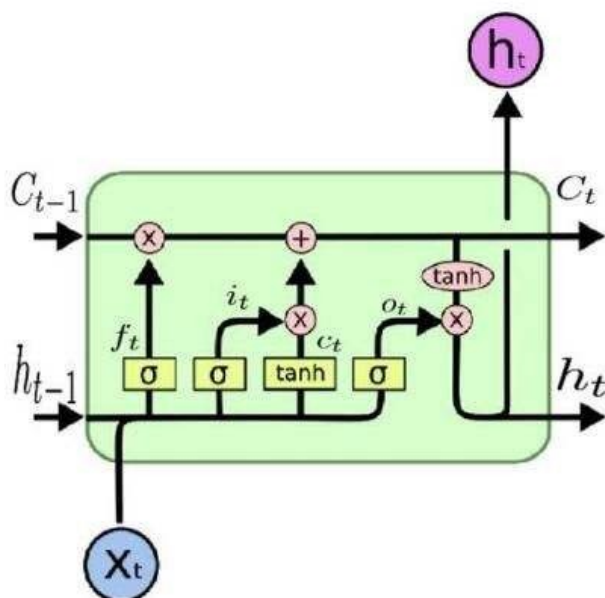
o:输出门：控制内部状态有多少需要被输出，成为新的隐状态 $h_t = o_t * \tanh(c_t)$ 激活后才被控制输出
ct~ 同样由 x_t, h_{t-1} 生成，并被 \tanh 激活 ($f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ $f(x)' = 1 - f(x)^2$)，之后被 it 控制大小

$$c_t = i_t * c_{t-1} + f_t * c_{t-1}$$

过去信息如果有必要记忆，遗忘门会保持 1，会使过去的记忆一直保留下来，从而实现长期记忆。

否则相反的话，新记忆被记忆，过去的不重要的被遗忘，从而学到记忆之间的依赖

生成记忆 h 时,选择的是 \tanh ,中心是 1，和多数特征吻合。0 附近梯度也更大



GRU:

Update: controls how much info in the new hidden states are kept or updated

$$\mathbf{u}^{(t)} = \sigma \left(\mathbf{W}_u \mathbf{h}^{(t-1)} + \mathbf{U}_u \mathbf{x}^{(t)} + \mathbf{b}_u \right)$$

Reset: controls how much info in the previous hidden states are kept

$$\mathbf{r}^{(t)} = \sigma \left(\mathbf{W}_r \mathbf{h}^{(t-1)} + \mathbf{U}_r \mathbf{x}^{(t)} + \mathbf{b}_r \right)$$

New Hidden state: **Reset** selects info from previous hidden state and combines it with the current input

$$\tilde{\mathbf{h}}^{(t)} = \tanh \left(\mathbf{W}_h (\mathbf{r}^{(t)} \circ \mathbf{h}^{(t-1)}) + \mathbf{U}_h \mathbf{x}^{(t)} + \mathbf{b}_h \right)$$

Hidden state: **Update** selects info from previous hidden state and combines it with the current input

$$\mathbf{h}^{(t)} = (1 - \mathbf{u}^{(t)}) \circ \mathbf{h}^{(t-1)} + \mathbf{u}^{(t)} \circ \tilde{\mathbf{h}}^{(t)}$$

知乎 @乐天

有两个门。没有用内部状态 c, 直接更新隐状态 h

u: 更新门: 表示隐状态更新的比例

r: 重置门: 生成新的隐状态时, 控制上一隐状态的保留比例 (u, r 共同左右达到遗忘门的效果, 实现长期记忆)

h 直接由更新门更新, 新的隐状态权重为 u, 旧的隐状态权重为 1-u

新的隐状态由上一时刻 h_{t-1} 和 x_t 生成, 但用重置门控制上一隐状态的保留比例

3 梯度消失, 梯度爆炸

激活函数导致的梯度消失:

sigmoid/Tanh : 值过大或者过小时, 梯度为 0, 梯度消失

Relu 为正时, 梯度不容易消失, 计算简单。而且单边抑制 (为负时不激活, 使网络稀疏)

为负时, 会导致神经元死亡, 无法激活

leaky Relu: 正时相同。为负时, $y = ax$. a 很小还是单边抑制, 但不容易消失 / a 不好选择 (leaky 漏的)

P-Relu: 正时相同。为负时, $y = ax$. a 学习得到

(P: 参数化的)

rnn 导致的梯度消失/爆炸:

最后层对输入层的导数, 是每一层雅克比矩阵连乘 W^n :

(因为 rnn 权值矩阵 W 相同, 所以没法抵消。cnn 每层参数不同, 独立同分布, 所以问题不如 rnn 严重)

当雅克比矩阵特征值大于 1 时:

$$W^n = \mathbf{Q} \mathbf{A} \mathbf{Q}^T * \mathbf{Q} \mathbf{A} \mathbf{Q}^T * \mathbf{Q} \mathbf{A} \mathbf{Q}^T \dots = \mathbf{Q} * \mathbf{A}^n * \mathbf{Q}^T$$

靠近输入端梯度会爆炸。反之靠近输入端梯度会消失。

如果使用 Relu 做激活函数, 最好将 W 初始化为接近单位矩阵, 不容易梯度爆炸/消失

解决:

梯度爆炸: grad_clip

梯度消失 Resnet 短接或者 highway network 加权

LSTM: 引入遗忘门, 控制旧信息的权重

4 防止过拟合

dropout: 随机舍弃神经元 如果共有 N 个神经元参与 dropout, 每个神经元都有两种可能, 相当于训练 2^N 个模型。这些模型可能每次共享一些神经元。每次神经元都可能和不同的神经元一起参

与训练, 会减弱全体神经元之间的相互依赖, 增加泛化能力, 防止过拟合

训练时, 以概率 p 随机丢弃。测试时, 不丢弃神经元, 可能导致网络输出比训练时大 p 倍。

所以测试时每个神经元的输出乘以概率 p , 使得总的输出量级等于训练时

Batch_normalize:

一般的: 输入数据 $X(m,n)$ 时, 对于每个样本, 有 n 个特征。对每一个特征进行归一化 (在所有样本上), 可以保证不同特征有相同的更新速率, 可以更快的收敛。

一种等比例: $x = x - x_{\min} / x_{\max} - x_{\min}$ 一种改变分布为 $N(0,1)$ $x = x - u / \sigma$

问题: 对不同批的输入, 前边层参数更新 (变化), 使得后边层接收的输入分布也会发生变化, 后边层每次都要拟合不同的分布, 使得学习速度很慢。训练复杂且容易过拟合。

如果每次在输入后一层之前, 对输入的数据进行归一化处理, 强制每一层输入都保持相同分布 (均值为 0, 方差为 1 的正态分布)。那么不同 batch 经过时, 网络不需要学习不同分布: ($k=1-n$)

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

对每个 batch, (B,n) 每个特征 (节点) 在该 batch 上做归一化, 那么这个 batch 的每个节点 (特征), 其均值为 0, 方差为 1. $E(x_k)$ 是特征 k 在该 batch 上的均值, σ 是该 batch 的方差。

(而且可以将输入移到线性区, 防止梯度消失 (sigmoid 等))

但直接归一化, 之前学到的某些分布特征可能丢失。为防止丢失, 同样对每一维度进行补偿, 通过

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

将每一维的分布恢复为均值为 β_k , 方差为 γ_k 的正态分布。需要学习得到, 从而学到的是最优的。之后预测之前用这些学到的最优的分布。

对 BN, 由于不同层有不同的分布, 因此每一层参数不同。

而对于 CNN, 每个卷积核作用相同, 尽管捕捉不同位置的特征。因此一个卷积核对应的输入对应一组 BN 参数。这个卷积核作用的所有 batch, normalize. n 组卷积核, n 组 BN 层。

Bagging / L1L2 / 降低模型复杂度 / 增加样本数量 /

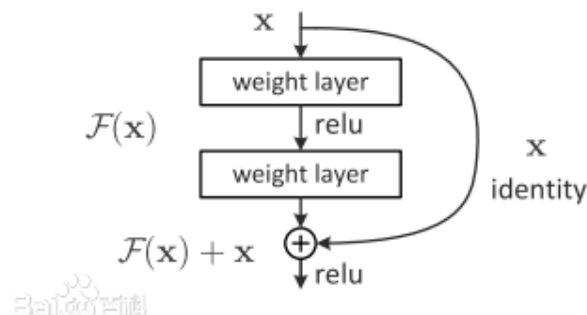
5 Resnet/highway network: 在深度学习过程中, 本来多加一层不会更差, 如果是恒等映射, 至少和原来一样好。但多层时, 由于导数的连乘可能导致的梯度消失/爆炸, 使得梯度回传到靠近输入端时, 梯度过大或者过小, 使得该层参数无法正确学习, 网络性能变差。

Resnet:为了能让靠近输入端的网络能正确学习, 在远端和近端直接短接, 让梯度能够直接回传到近端。

残差网络: 原始 $x \rightarrow F(x) = H(x)$

现在输出时是 $x + F(x) \rightarrow H(x)$

$F(x)$ 是多层之后的结果。之前 $F(x)$ 要直接拟合 $H(x)$, 而现在 $F(x)$ 只需要拟合 $H(x)$ 和 x 的残差。



好处:

1: 解决网络过深梯度消失/爆炸

回传梯度时, 即使多层网络 $f(x)$ 梯度较小或消失, 仍能通过 short-cut 路径, 把梯度回传给近端输入

2 对不起作用的层, 该层相当于直接学习一个恒等映射, 不会使网络性能变差。该层无表达能力也对网络整体性能无影响。

3 跳过的 $F(x)$, 只需要拟合上层输入和目标之间的残差。相对于直接学习 $H(x)$ 变得简单

4: 加入短接还能在一定程度上能改变网络的对称性 (sometimes 对称无鉴别能力)

5 相比于 highway network, 不引入额外参数

Highway network:

相比于直接加一层短接, highway network 的输出是 x 和 $F(x)$ 的加权输出, 权重由门控进行控制 (类似于 LSTM, 门控由上层输入 x 决定, 参数通过学习得到)。也能解决梯度消失, 但需要引入额外参数学习门控

$$y = H(x, W_H) \cdot T(x, W_T) + x \cdot (1 - T(x, W_T)). \quad (3)$$

6 CNN

卷积: 对文本, 应该是 $1, kn$. Ngram 大多数情况下用 k, n 的卷积核, 本质相同

池化: 多个卷积核 (n 个) 得到固定维度的向量, 是局部信息向量
之后得到定长的向量表示。

(提取最有用的特征表示, 防止过拟合)

(对粗提取的特征降采样, 得到更有效的特征表示)

文本:

1-max pooling: 每个卷积核得到的向量中, 选择最大的, 拼成 n 维向量

k-max pooling: 1-max pooling 每个卷积核可能丢掉其他该核的重要特征
每个向量选最大 K 个特征, 拼成 Kn 维向量

平均池化： 所有卷积核得到的特征向量取平均

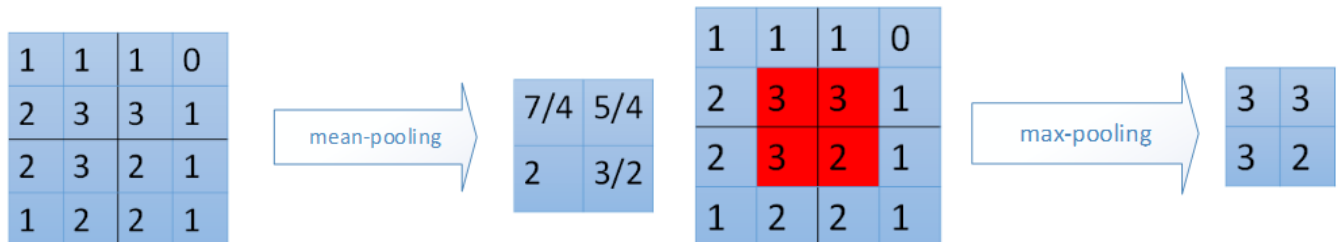
图像：

卷积后，对每个卷积核得到的新特征矩阵进行池化操作，将原来的较大特征矩阵提取为较小特征矩阵

一般选取不重叠区域池化。如果选取 2×2 不重叠区域
(可以降低参数数量/保持平移不变性)

平均池化： 每次选取每个区域的均值作为该区域的特征值，得到新的特征值（保留低频特征）

最大池化： 每次选取每个区域的最大值作为该区域的特征值，得到新的特征值（保留高频特征）



7 优化算法 sgd/momentum/adam/adagrad/rmsprop

1 Gradient Descent Optimizer

Gradient Descent Optimizer 即梯度下降法, 通过计算参数的梯度来更新参数, 公式可以表示为:

$$\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \quad (1)$$

如果样本量巨大且参数特别多, 每次实行梯度下降时计算可能会非常耗时, 所以 GD 又可以衍变为 SGD 以及 Mini-Batch SGD 两种形式, 这两种方法的思想也很简单, SGD 就是每次随机选择一个样本来更新参数, 这样速度会非常快, 但是梯度更新的方向随机性较大, 可能不会很快收敛, 甚至无法收敛到局部最优解, 一般得到的是近似最优解. Mini-Batch SGD 对于 GD 和 SGD 做了一个折中, 它每次随机选择一部分样本进行梯度计算, 更新参数, 这样既保证了计算速度, 又保证了可以较快地收敛, 也是实际采用较多的办法.

2 Momentum Optimizer

Momentum Optimizer, 即动量优化法, 它与 GD 最大的不同是, GD 一般具有恒定的 learning rate, 而且每次只利用了当前梯度的信息, 这样可能会存在收敛速度非常慢, 甚至无法收敛的情况. Momentum 引入了一个 momentum vector, 每次参数的更新不仅与本次计算的梯度相关, 还与之前的梯度相关, 这样参数更新的方向会更加朝着有利于收敛到最优解的方向, 收敛速度会更快. Momentum 的具体计算公式如下:

$$m \leftarrow \beta m + \eta \nabla_{\theta} J(\theta) \quad (2)$$

$$\theta \leftarrow \theta - m \quad (3)$$

公式 (2) 中的 β 是一个超参数, 可以理解为之前的动量对于当前参数更新影响的大小, η 是学习率, 假设每次梯度恒定, 那么根据式 (2)(3) 可以推出 Momentum 参数更新的速度是 GD 的 $\frac{1}{1-\beta}$ 倍. 实际过程中, 如果没有使用 Batch Normalization 的情况下, 输入数据各个维度尺寸往往差异会很大, 这更加不利于收敛, 这个时候如果使用 Momentum 替代 GD 会更有利于收敛 (减小震荡), 而且有助于跳出局部最优解. 实际经验表明, 超参数 β 一般设置为 0.9 比较好.

SGD: 每次更新只取决于当前 batch 梯度, 随机性很大, 不容易收敛

主要考虑下降稳定:

momentum: 每次更新的方向和上次更新方向大致相同, 只用当前 batch 梯度 $J(\theta)$ 部分调整。

更新更加稳定, 减小震荡, 使收敛速度加快. 一般选择 $\beta=0.9$.

NAG: 类似于 momentum, 也保留部分上一次的梯度来减小震荡, 但如果已经可以猜到下一个更加正确的点在什么位置, 那么可以直接用更加正确的点处的梯度, 来当做该点处的梯度, 进行更新。

具体来说, 用本次 batch 的梯度时, momenta 直接加上当前位置 θ 处的梯度, NAG 则是加上 (当前位置往后一点 $\theta + \beta m$) 的梯度, 因为往后一点的点 $\theta + \beta m$ 是通过 momenta 预测得到的, 会指向更加正确的方向. 使收敛更加迅速。

3 Nesterov Accelerated Gradient(NAG)

Nesterov Accelerated Gradient, 简称 NAG, 是在动量优化算法的基础上改进得到的一个算法, 它与 Momentum 形式上非常相近, 最大的不同在于: m 每次更新时加上梯度的不同, Momentum 是加上当前位置 θ 的梯度, 而 NAG 是加上当前位置之后一点点 $\theta + \beta m$ 位置的梯度, 理由是一般动量 m 会指向更加正确的 (局部) 最优点的方向, 因此这样改进会更加快速地收敛到极值点. NAG 具体的更新公式如下:

$$m \leftarrow \beta m + \eta \nabla_{\theta} J(\theta + \beta m) \quad (4)$$

$$\theta \leftarrow \theta - m \quad (5)$$

我们也可以通过下面的图 1 来说明为什么 NAG 比 Momentum 更好: 而

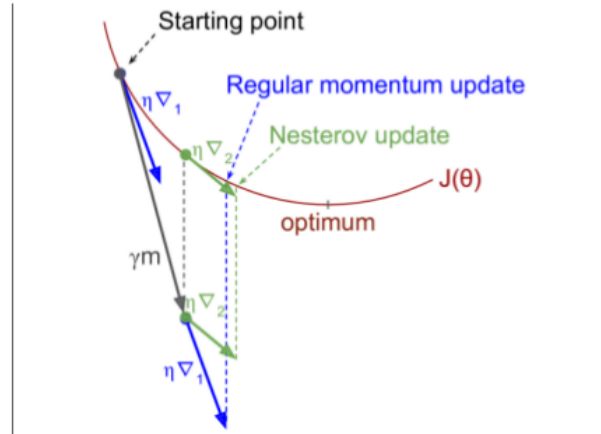


图 1: NAG vs Momentum

且, NAG 在接近极值的时候, 如果超出了极值点, 会把更新的方向往回拉, 而不是继续向前, 这有利于减小震荡, 加速收敛.

4 AdaGrad Optimizer

AdaGrad Optimizer, 即自适应梯度下降法, 其参数更新公式如下:

$$s \leftarrow s + \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta) \quad (6)$$

主要考虑能自适应调整每一维的更新幅度:

Adagrad: 对每一维梯度, 更新时除以该维梯度累积的模值。 (根据每个维度已经更新的程度, 自适应调整该维度本次更新的幅度)。容易使梯度减小的快

RMSP: 同样每一维, 更新时除以该维梯度累积的模值, 自适应调整该维的更新速率。 (Adadelat 的变体)

为了防止模值增大太快, 每一维用的模值 s 都是在上一次 s 模值的基础上, 用本次梯度的模值进行微调, 防止累积的梯度模值增加太快, 使该维学习率降得太快。之前累加模值 s 在本次更新中所占的比例 b 随着时间指数衰减, 距离目前越远的 s , 影响越小.

$$\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{s + \varepsilon} \quad (7)$$

其中 \otimes 表示按元素相乘 (elementwise-multiply), \oslash 表示按元素相除. 根据上述公式, AdaGrad 本质上就是对每次更新的各维梯度做了一个自适应缩放操作, 这个缩放操作是通过除以 $\sqrt{s + \varepsilon}$ 来完成的, s 一直在累加各维梯度之和 (会越来越来), ε 是平滑因子, 作用是为了防止分母为 0, 一般取非常小的值 (比如 10^{-9}). 通过这样的操作, 保证了随着迭代的增加, 参数的更新速度最终会越来越慢, 这样可以减小震荡, 避免因步长太大而无法收敛到极值点. 但是它也有一个很大的缺点, 就是在复杂函数优化 (比如 Deep Learning) 中, 它容易过早收敛 (early stop), 这样就无法收敛到全局极小值点, 而是收敛到局部极小值点. 所以一般我们不使用它来做特别复杂函数的优化 (训练神经网络之类), 但是用来求解一些简单的问题, 比如线性回归还是可以的.

5 RMSProp Optimizer

之前说过, AdaGrad 的最主要的问题是它的梯度更新减小速度太快了, 以至于它很容易陷入局部极值点. RMSProp 通过只累加最近一次的梯度来解决这个问题 (AdaGrad 是累加之前所有的梯度), 具体的做法是引入了一个衰减因子 (decay rate) β (实际上是指数衰减), 其参数更新公式如下:

$$s \leftarrow \beta s + (1 - \beta) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta) \quad (8)$$

$$\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{s + \varepsilon} \quad (9)$$

RMSProp 和 AdaGrad 最大的不同在于 s 的更新方式, 这里 β 是一个超参数, 表示之前的梯度累计和对于当前的影响大小, 且这个大小是按照指数速率衰减的 (β 就是衰减率), 也就是说每次 s 由两部分组成: 一部分是之前的梯度累计和, 另外一部分是当前的梯度大小 (起主要作用的部分), 离当前越远的梯度对当前的影响越小 (因为是指数衰减嘛), 这样从直观上理解也是合理的. 实际上, 除了对于特别简单的函数来说, RMSProp 几乎总是优于 AdaGrad, 而且一般情况下也比 Momentum 和 NAG 更优, 实际上它是 Adam Optimizer 出现之前人们使用最多的一种优化算法之一.

Adam: 相比于 sgd, 主要有两方面的更新:

- 1: 类似于 momentum, 为了使训练更稳定, 每次更新用的梯度不只是当前 batch 的梯度, 还考虑之前的梯度, 使得收敛更快
- 2: 也结合了 RMSP 的思想, 根据已经学习过的每一维的梯度和, 自适应调整每一维在本次的更新步长, 且同样在累积每一维之前梯度的和 s 时, 用之前的 s 进行调整

因此学习率可以自适应调整+相对来说学习更稳定, 收敛更快。

sgd 训练时间更长, 但是在好的初始化和学习率调度方案的情况下, 结果更可靠

6 Adam Optimizer

Adam 的全称是 adptive moment estimation, 它结合了 Momentum 和 RMSProp 的思想, 就像 Momentum 一样, 它记录了之前梯度的指数平均, 就像 RMSProp 一样, 它记录了过去梯度平方和的指数平均, 其参数更新公式如下:

$$m \leftarrow \beta_1 m + (1 - \beta_1) \nabla_{\theta} J(\theta) \quad (10)$$

$$s \leftarrow \beta_2 s + (1 - \beta_2) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta) \quad (11)$$

$$m \leftarrow \frac{m}{1 - \beta_1} \quad (12)$$

$$s \leftarrow \frac{s}{1 - \beta_2} \quad (13)$$

$$\theta \leftarrow \theta - \eta m \oslash \sqrt{s + \varepsilon} \quad (14)$$

如果只看 (10) 的话, 其形式很像 Momentum, 如果只看 (11) 的话, 其形式很像 RMSProp, (14) 的参数更新与 RMSProp 相比是将当前的梯度 $\nabla_{\theta} J(\theta)$ 换成了当前动量 m , 这就相当于是 Momentum 与 RMSProp 的巧妙结合, 而考虑到初始时 m 和 s 都是 0, 最开始都比较小, 因此 (12), (13) 的作用是初始时给 m, s 进行加速. 这里 β_1 和 β_2 都是超参数, 一般根据经验我们将 β_1 设置为 0.9, β_2 设置为 0.999, 平滑系数 ε 一般设置为一个非常小的数 (比如 10^{-8}), 由于 Adam 也是一个自适应的优化算法, 所以我们不需要对学习率进行手动调节, 这样有的时候 Adam 比 GD 更加易于使用. 综合来说, Adam 是上面所说的 6 种优化算法中最优的一个 (它借鉴了其他算法的优点并集成在一起), 所以当你不确定使用哪一种优化算法的时候, 就使用 Adam 吧.

7 Jacobians and Hessians

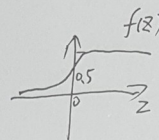
以上所说的六种优化算法, 有一个共同点就是它们只使用了损失函数一阶导数的信息 (Jacobians), 还有一种方式是使用二阶导数的信息 (Hessians), 其收敛速度要比 Jacobians 要快得多, 但是这种方式的时间复杂度是 $O(n^2)$ (n 是参数个数), 而深度学习中往往有成千上万的参数, $O(n^2)$ 的时间复杂度是无法接受的, 而且 Hessians 的计算非常复杂, 这就是我们为什么一般只选择 Jacobians 类型的优化算法的原因.

7 逻辑回归, softmax, RNN 导数推导

逻辑回归:

$$P(X=1) = \frac{1}{1+e^{-z}} = f(z) \quad z = \theta^T x$$

$$P(X=0) = 1 - P(X=1) = \frac{e^{-z}}{1+e^{-z}} = \frac{1}{e^z+1} = f(-z)$$



前向: 损失函数: $L = P(X=1)^{y_i} \cdot P(X=0)^{1-y_i}$ (最大化)

对数形式: $L = y_i \log \frac{P(X=1)}{f(z)} + (1-y_i) \log \frac{P(X=0)}{(1-f(z))}$ \uparrow $(f(z) = \text{sigmoid}(z))$

$$= y_i \log f(z) + (1-y_i) \log (1-f(z))$$

后向: $\frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial \theta}$

其中 $\frac{\partial L}{\partial z} = y_i \cdot \frac{1}{f(z)} \cdot f'(z) + (1-y_i) \cdot \frac{1}{1-f(z)} \cdot -1 \cdot f'(z)$

$$f'(z) = f(z) \cdot (1-f(z))$$

$$\therefore \frac{\partial L}{\partial z} = y_i \cdot (1-f(z)) + (1-y_i) \cdot \frac{1}{f(z)-1} \cdot f(z) \cdot (1-f(z))$$

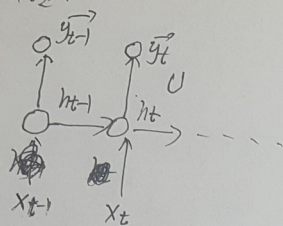
$$= y_i (1-f(z)) + (y_i-1) \cdot f(z)$$

$$= y_i - f(z)$$

$$\frac{\partial z}{\partial \theta} = x \quad \frac{\partial z}{\partial x} = \theta$$

$$\therefore \frac{\partial L}{\partial \theta} = [y_i - f(z)] \cdot x \quad \frac{\partial L}{\partial x} = [y_i - f(z)] \cdot \theta$$

RNN 中反向传播:



前向: $loss = -\sum_{i=1}^K y_i \log y_t \quad \downarrow$

向量 $\vec{y}_t = \text{softmax}(U \cdot \vec{h}_t + \vec{b}_1)$ $(1 \times K)$

其中 $\vec{h}_t = \tanh/\text{sigmoid}(W_1 \cdot \vec{h}_{t-1} + W_2 \cdot \vec{x}_t + \vec{b}_2)$

反向: 假设 ground-truth 类别为 K .

$loss = -1 \cdot \log y_K = -\log y_K$ 一维.

$y_K = \frac{\text{softmax}(U \cdot \vec{h}_t + \vec{b}_1)}{(K)}$

其中 $\vec{z} = U \cdot \vec{h}_t + \vec{b}_1$, $g(\vec{z}) = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}}$ g 为 j 元素

$\frac{\partial loss}{\partial \vec{h}_t} = \frac{\partial loss}{\partial y_K} \cdot \frac{\partial y_K}{\partial \vec{z}} \cdot \frac{\partial \vec{z}}{\partial \vec{h}_t}$
 $= -\frac{1}{y_K} \cdot \frac{\partial y_K}{\partial \vec{z}} \cdot \frac{\partial \vec{z}}{\partial \vec{h}_t}$

其中 $\frac{\partial y_K}{\partial \vec{z}} = \frac{\partial \frac{e^{z_K}}{\sum}}{\partial \vec{z}}$

对每个元素 z^j : 当 $j=K$

$\frac{\partial \frac{e^{z_K}}{\sum}}{\partial z^K} = \frac{e^{z_K} \cdot \sum - e^{z_K} \cdot e^{z_K}}{\sum^2} = \frac{e^{z_K}}{\sum} - \frac{e^{z_K}}{\sum} \cdot \frac{e^{z_K}}{\sum}$
 $= g(z^K) - g^2(z^K) = g(z^K)(1 - g(z^K)) = y^K(1 - y^K)$

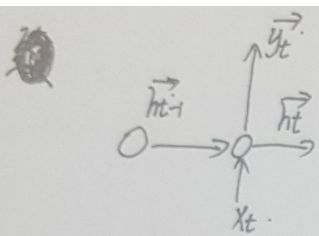
当 $j \neq K$
 $\frac{\partial \frac{e^{z_K}}{\sum}}{\partial z^j} = \frac{0 - e^{z_K} \cdot e^{z_j}}{\sum^2} = -\frac{e^{z_K}}{\sum} \cdot \frac{e^{z_j}}{\sum} = -g(z^K) \cdot g(z^j)$
 $= -y^K \cdot y^j$

(K,1) $\frac{\partial loss}{\partial y_K} \cdot \frac{\partial y_K}{\partial \vec{z}} = -\frac{1}{y^K} \cdot y^K \cdot \begin{cases} 1 - y^j & j=K \\ 0 - y^j & j \neq K \end{cases} = \vec{y}_t - \vec{y}_i = y^K \cdot (0 - y^j)$
 $(0, 0, \dots, 0, 1, 0, 0)$

$\frac{\partial \vec{z}}{\partial \vec{h}_t} = U^T$

$\Rightarrow \frac{\partial loss}{\partial \vec{h}_t} = U^T (\vec{y}_t - \vec{y}_i)$
 $(m, K) \quad (K, 1)$

$(\vec{z} = U \cdot \vec{h}_t + \vec{b}_1)$
 $K, 1 \quad K, m \quad m, 1$



$$\vec{h}_t = f(W_1 \vec{h}_{t-1} + W_2 \vec{x}_t + b) \quad f = \text{sigmoid/tanh}$$

$$\frac{\partial \vec{h}_t}{\partial \vec{z}} = \frac{\partial \vec{h}_t}{\partial f(\vec{z})} \frac{\partial f(\vec{z})}{\partial \vec{z}} = \begin{bmatrix} f'(\vec{z}_1) & & 0 \\ & f'(\vec{z}_2) & \\ 0 & & f'(\vec{z}_m) \end{bmatrix}_{m \times m} = \text{diag}[f'(\vec{z})]$$

其中 $\vec{z} = W_1 \vec{h}_{t-1} + W_2 \vec{x}_t + b$

$$\frac{\partial \vec{z}}{\partial \vec{h}_{t-1}} = W_1^T \quad (n \times m)$$

$$\therefore \frac{\partial \vec{h}_t}{\partial \vec{h}_{t-1}} = \underbrace{W_1^T}_{(n \times m)} \cdot \underbrace{\text{diag}[f'(\vec{z})]}_{(m \times m)}$$

对 $f = \text{ReLU}(\vec{z})$ $\frac{\partial \vec{h}_t}{\partial \vec{h}_{t-1}} = W_1^T \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

$$\therefore \frac{\partial \text{loss}}{\partial \vec{h}_{t-1}} = \frac{\partial \vec{h}_t}{\partial \vec{h}_{t-1}} \cdot \frac{\partial \text{loss}}{\partial \vec{h}_t}$$

$$(n, 1) = \underbrace{W_1^T}_{(n \times m)} \cdot \underbrace{\text{diag}[f'(\vec{z})]}_{(m \times m)} \cdot \underbrace{U^T}_{(m \times k)} \cdot \underbrace{(\vec{y}_t - \vec{y}_i)}_{(k, 1)}$$

8 S2S:

beam search: 只用在测试中。因为训练有 ground truth

每次输出的 y_{t-1} 中, 选择概率最大的两个, 作为下一次可能的输入

如果 $K=2$ 分别选择本次的 x_t 为 $y_{t-1_biggest}$, 得到下一个 y_t 的所有概率

再选 x_t 为 $y_{t-1_second_biggest}$, 下一个 y_t 的所有概率

最后选 $P(y_{t-1_biggest}) * P(y_t | y_{t-1_biggest})$ 的所有序列,

和 $P(y_{t-1_second_biggest}) * P(y_t | y_{t-1_second_biggest})$ 的所有序列 中, 概率最高

的

两个序列对应的 y_{t-1}, y_t 作为候选序列, 分别传到之后的 x_t 里, 重复算上述过程, 直到结束, 选概率最高的序列作为输出。

Beam search 好处: 如果采用 greedy 的方法, 会使得当前输出单词只取决于上一单词

$P(y_t | y_{t-1})$ 而非之前序列 $P(y_t | y_{t-1}, \dots, y_1)$, 使得输出效果降低。

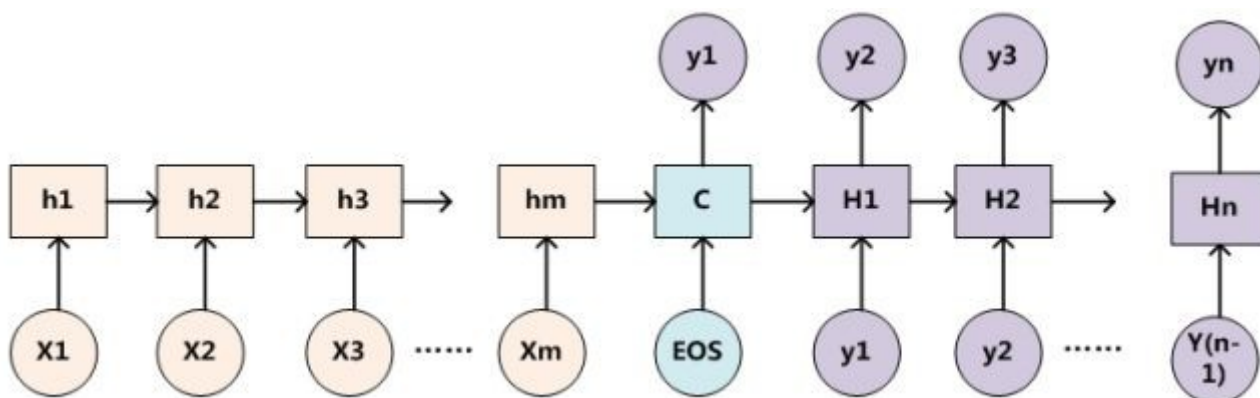
但穷举 $P(y_t | y_{t-1}, \dots, y_1)$ 复杂度很高。

decoder 初始输入 x_0 和 h_0 : 一般将 encoder 得到的向量 c 作为 decoder 的初始隐状态向量 h_0 。初始 x_0 一般是 $\langle s \rangle$ 从而启动解码。

计算 h_t 时, attention ct 一般加在 h_{t-1} 上: $h_t = h_{t-1} + ct / \text{concat/gate(乘)}$

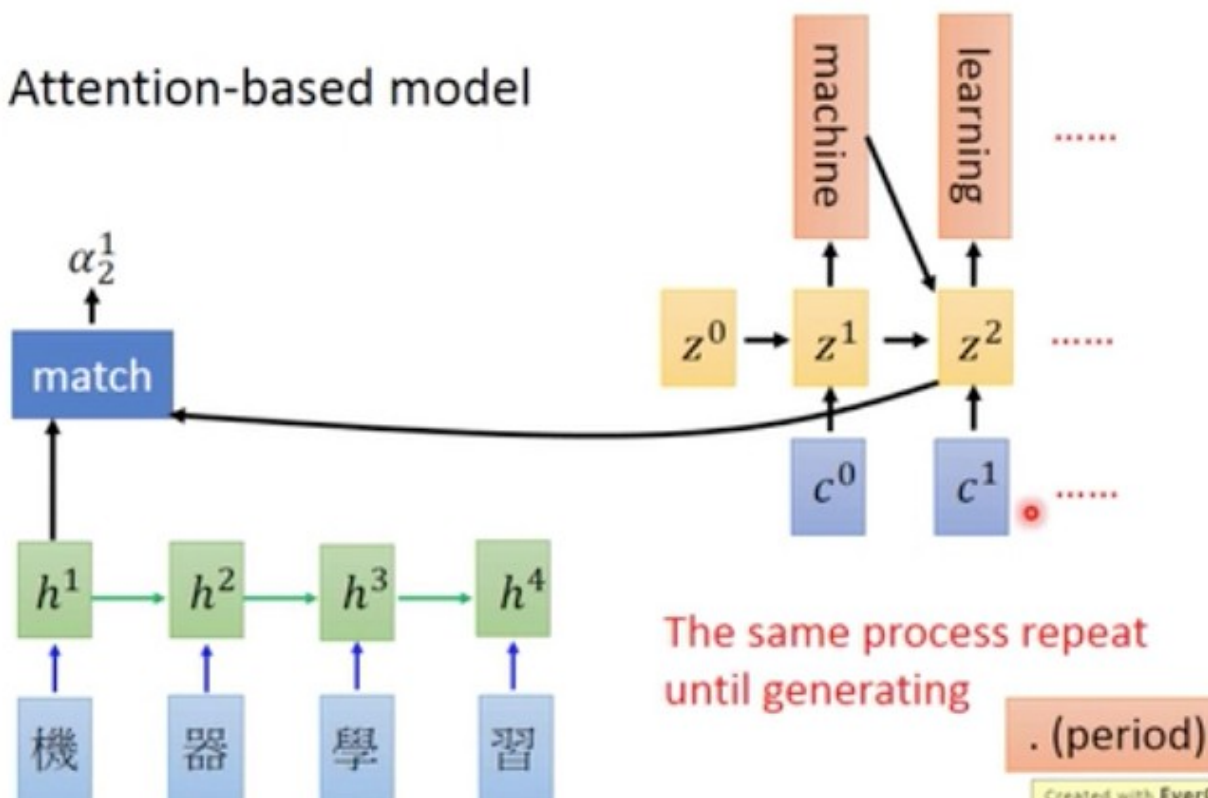
原文中 ct 独立, 和 y_{t-1}, h_{t-1} 一起, 分别乘新矩阵得到

新的 h_t



原文中 c_t 独立，和 y_{t-1} (machine), h_{t-1} (z_{t-1}) 一起，分别乘新矩阵得到新的 h_t

• Attention-based model



9 wordvec

对于层级化 softmax。对任意两个向量 v_c, v_i 。从 v_c 到 v_i 的概率 $P(v_i|v_c)$ 是该路径上一系列概率的相乘。每个节点处的概率可以看作由一个二分类器得出，该二分类器向量的参数是 node 的 vector。

平均路径长为 $\log|V|$ 。该树是根据词频建立的 huffman 编码树，所有词都是叶子节点，对应一条路径。高频词路径短。 $P(v_i|v_c)$ 是 v_c 到每一个叶子节点 i 的概率。该概率计算中，分类器 node 对应分类器的参数， v_c 对应分类器的输入样本向量。训练结束后，同时得到了 node vector 和 input vector。

10 EM 算法 (解决有变量 z 的极大似然)

11 HMM/CRF/维特比算法

12 fasttext

13 GNN

14 conv 1*1 意义 和 FC 区别