

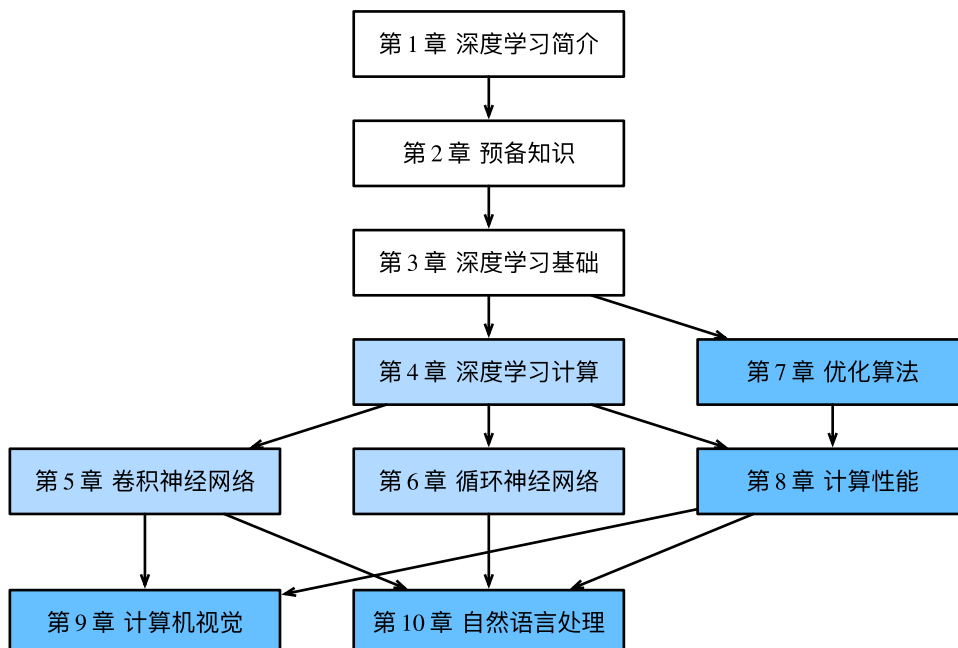


这是 [动手学深度学习\(PyTorch版\)](#) 的读书笔记. 后续可能会因个人懒的原因加入更多的DL笔记啥的..

```
# 给记性不好的我的Anaconda handbook
# 查看帮助
conda -h
# 基于python3.6版本创建一个名字为python36的环境
conda create --name python36 python=3.6
# 激活此环境
activate python36
source activate python36 # linux/mac
# 再来检查python版本, 显示是 3.6
python -V
# 退出当前环境
deactivate python36
# 删除该环境
conda remove -n python36 --all
# 或者
conda env remove -n python36

# 查看所以安装的环境
conda env list
```

动手学DL(PyTorch版)



1. 简介

- "用数据编程"

与其枯坐在房间里思考怎么设计一个识别猫的程序，不如收集一些已知包含猫与不包含猫的真实图像，然后我们的目标就转化成如何从这些图像入手得到一个可以推断出图像中是否有猫的函数。

- 算力增长 >> 存储增长

存储容量没能跟上数据量增长的步伐。与此同时，计算力的增长又盖过了数据量的增长。这样的趋势使得统计模型可以在优化参数上投入更多的计算力，但同时需要提高存储的利用效率，例如使用非线性处理单元。

- DL特点: 端到端

并不是将单独调试的部分拼凑起来组成一个系统，而是将整个系统组建好之后一起训练。

2. PyTorch基础操作

在PyTorch中，`torch.Tensor`是存储和变换数据的主要工具。`Tensor`近似与多维数组，具有自动求梯度与GPU计算功能。

- Tensor

创建Tensor

```
import torch
# 创建一个5 * 3未初始化的Tensor
x = torch.empty(5, 3)

# 创建一个5 * 3 全为long 0的Tensor
x = torch.zeros(5, 3, dtype=torch.long)

# 输出形状
print(x.size())
print(x.shape)
```

运算Tensor

```
# 加
z = x + y

torch.add(x, y, out=z)

y.add_(x) # y += x
```

索引(浅拷贝)

```
y = x[0, :]
y += 1
print(y)
print(x[0, :]) # 源tensor也被改了
```

改变形状

```
y = x.view(-1, 5)
```

clone(深拷贝)

```
x_cp = x.clone().view(15)
```

Tensor => Numpy

```
np = torch.ones(5).numpy()
```

Numpy => Tensor

```
ts = torch.from_numpy(np.ones(5))
```

Tensor with GPU

```
# Below codes CAN ONLY run with PyTorch-GPU
if torch.cuda.is_available():
    gpu = torch.device("cuda")
    y = torch.ones_like(x, device=gpu) # y is in gpu
    x = x.to(gpu)
    z = x + y
    print(z)
    print(z.to("cpu", torch.double))
```

- 梯度 (gradient)

Tensor是这个包的核心类，如果将其属性 `.requires_grad` 设置为 `True`，它将开始追踪(track)在其上的所有操作（这样就可以利用链式法则进行梯度传播了）。完成计算后，可以调用 `.backward()` 来完成所有梯度计算。此 `Tensor` 的梯度将累积到 `.grad` 属性中。`.detach()` 可以阻止追踪

e.g.

创建一个Tensor并设置requires_grad=True:

```
x = torch.ones(2, 2, requires_grad=True)
print(x)
print(x.grad_fn)
```

做运算操作

```
y = x + 2
print(y)
print(y.grad_fn)
```

注意x是直接创建的，所以它没有 `grad_fn`，而y是通过一个加法操作创建的，所以它有一个为`<AddBackward>`的 `grad_fn`

像x这种直接创建的被称为叶子节点，叶子节点的 `grad_fn` 为 `None`

```
print(x.is_leaf) # True
print(y.is_leaf) # False
```

通过 `.requires_grad_()` 来改变`requires_grad`属性

```
a = torch.randn(2, 2) # Default requires_grad = False
print(a.requires_grad) # False
a.requires_grad_(True)
print(a.requires_grad) # True
```

`backward()`自动计算梯度(grad)

```
x = torch.ones(2, 2, requires_grad=True) # x: [ [1, 1], [1, 1]]
y = x + 2 # y: [ [3, 3], [3, 3]]
z = y**2 * 3 # z: [ [27, 27], [27, 27]]
z_mean = z.mean() # z_mean: [27]
# z_mean 为一个标量，所以无需指定求导变量
z_mean.backward() # <=> out.backward(torch.tensor(1.))
print(x.grad) # [ [4.5, 4.5], [4.5, 4.5]]
```

如果我们想要修改 `tensor` 的数值，但是又不希望被 `autograd` 记录（即不会影响反向传播），那么我可以对 `tensor.data` 进行操作。

```

x = torch.ones(1, requires_grad=True)

print(x.data) # x.data is a tensor [1]
print(x.data.requires_grad) # False

y = x * 2
x.data *= 100 # Only changes x's value, DO NOT backward.

y.backward()
print(x) # x: [100]
print(x.grad) # 2

```

3. DL基础

线性回归 (Linear Regression)

- 线性回归的基本要素
 - 模型

设房屋的面积为 x_1 , 房龄为 x_2 , 售出面积为 y , 我们需要建立 y 关于 x_1 , x_2 的表达式, 也即是模型 (model)
 - 训练数据

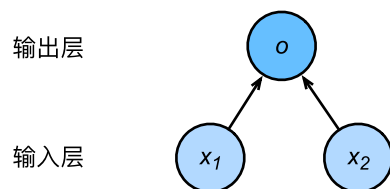
通过数据来寻找特定的模型参数值, 使模型在数据上的误差尽可能小。这个过程叫作模型训练 (model training)
 - 损失函数

在机器学习里, 将衡量误差的函数称为损失函数 (loss function)
 - 优化算法

当模型和损失函数形式较为简单时, 上面的误差最小化问题的解可以直接用公式表达出来。这类解叫作解析解 (analytical solution)。

线性回归和平方误差刚好属于这个范畴。然而, 大多数深度学习模型并没有解析解, 只能通过优化算法有限次迭代模型参数来尽可能降低损失函数的值。这类解叫作数值解 (numerical solution)。

- 线性回归的表示方法



如图, 线性回归是个单层神经网络, 在线性回归中, o 的计算完全依赖于 x_1 和 x_2 .
所以, 这里的输出层又叫全连接层 (fully-connected layer) 或稠密层 (dense layer)

- 从0开始手撸线性回归

即只用 `Tensor` 和 `autograd` 实现线性回归的训练

见 `Source Code: LinearRegressTest.py`

- 线性回归简洁实现

见 `Source Code: LinearRegressTestSimple.py`

总结

一般过程:

准备数据集 -> 读入数据 -> 定义模型 -> 初始化模型参数 -> 定义损失函数 ->
定义优化算法 -> 训练

PyTorch框架对应:

`torch.utils.data`: 数据处理相关

`torch.nn`: 神经网络的层

`torch.nn.init`: 模块的初始化方法

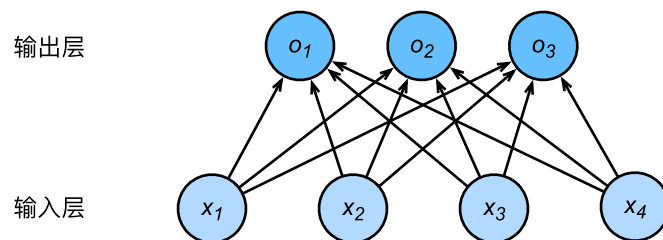
`torch.optim`: 优化算法

Softmax回归

- 简介

softmax回归同线性回归一样是个单层神经网络, 由于输出 o_1, o_2, o_3 的计算依赖于所有输入 x_1, x_2, x_3, x_4 , 故softmax回归的输出层也是个全连接层.

softmax回归也将输入特征与权重做线性叠加. 不同点在于, softmax回归的输出值个数等于标签中的类别数.



e.g.

$$o_1 = x_1 w_{11} + x_2 w_{21} + x_3 w_{31} + x_4 w_{41} + b_1,$$

$$o_2 = x_1 w_{12} + x_2 w_{22} + x_3 w_{32} + x_4 w_{42} + b_2,$$

$$o_3 = x_1 w_{13} + x_2 w_{23} + x_3 w_{33} + x_4 w_{43} + b_3.$$

softmax运算符 (softmax operator) 将输出层的输出值变为值为正且和为1的概率分布:

$$y_1, y_2, y_3 = \text{softmax}(o_1, o_2, o_3)$$

其中

$$y_1 = \frac{\exp(o_1)}{\sum_{i=1}^3 \exp(o_i)}, y_2 = \frac{\exp(o_2)}{\sum_{i=1}^3 \exp(o_i)}, y_3 = \frac{\exp(o_3)}{\sum_{i=1}^3 \exp(o_i)}$$

- 实现

见 `Source Code: SoftmaxTest.py` 和

见 `Source Code: SoftmaxTestSimple.py`

- 交叉熵损失函数 (cross entropy loss function)

衡量两个概率分布差异的方法.即不用判断概率达到某值,而是A概率比B, C都高即可.

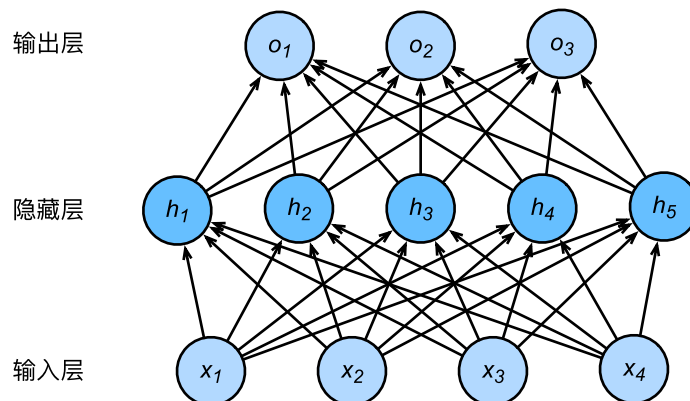
$$H(y_{(i)}, \hat{y}_{(i)}) = - \sum_{j=1}^q y_{(i)} \log \hat{y}_{(i)}$$

多层感知机 (MultiLayer Perceptron, MLP)

- 简介

多层感知机即在单层神经网络的基础上引入了一个到多个隐藏层 (hidden layer) 的网络结构.

如图所示的多层感知机中包含5个隐藏单元(hidden unit), 隐藏层和输出全连接层.



- 激活函数

多个线性函数的叠加 (多个简单隐藏层) 会导致仍然为一个线性函数.为了避免这种情况,需要对隐藏变量使用非线性函数.被称为激活函数 (activation function).

常用的激活函数有:

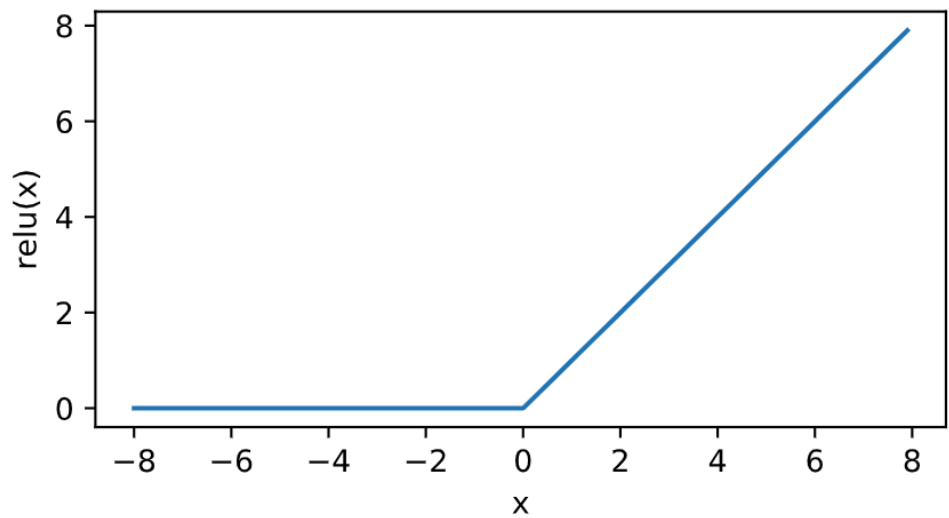
1. ReLU (Rectified Linear Unit)函数

给定元素 x , 该函数定义为

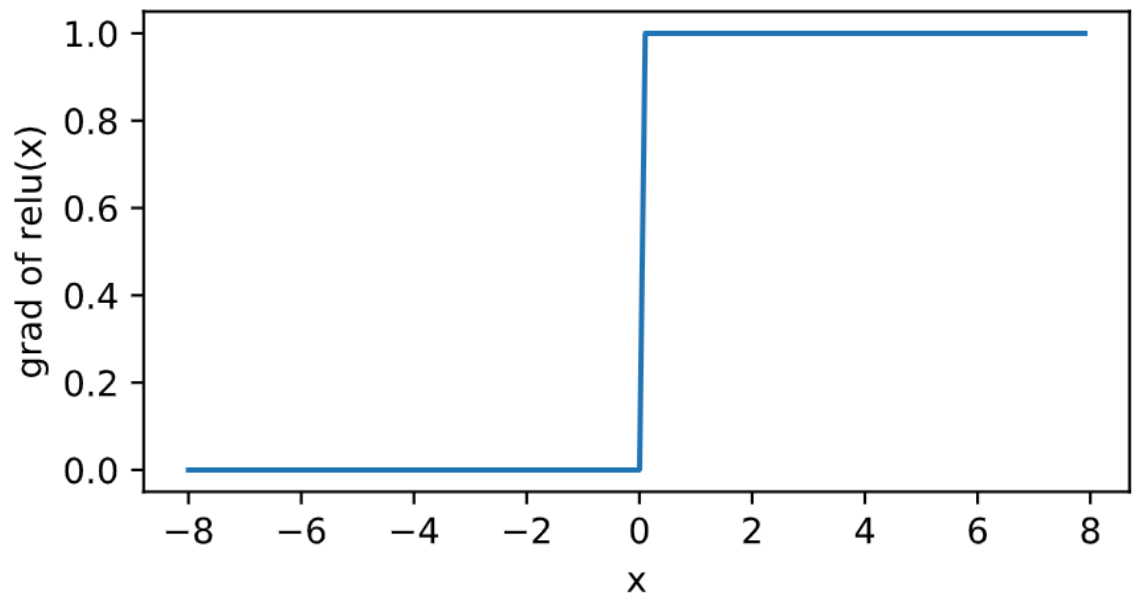
$$ReLU(x) = \max(x, 0).$$

即保留正数元素, 并将负数元素清0.

ReLU:



ReLU gradient:

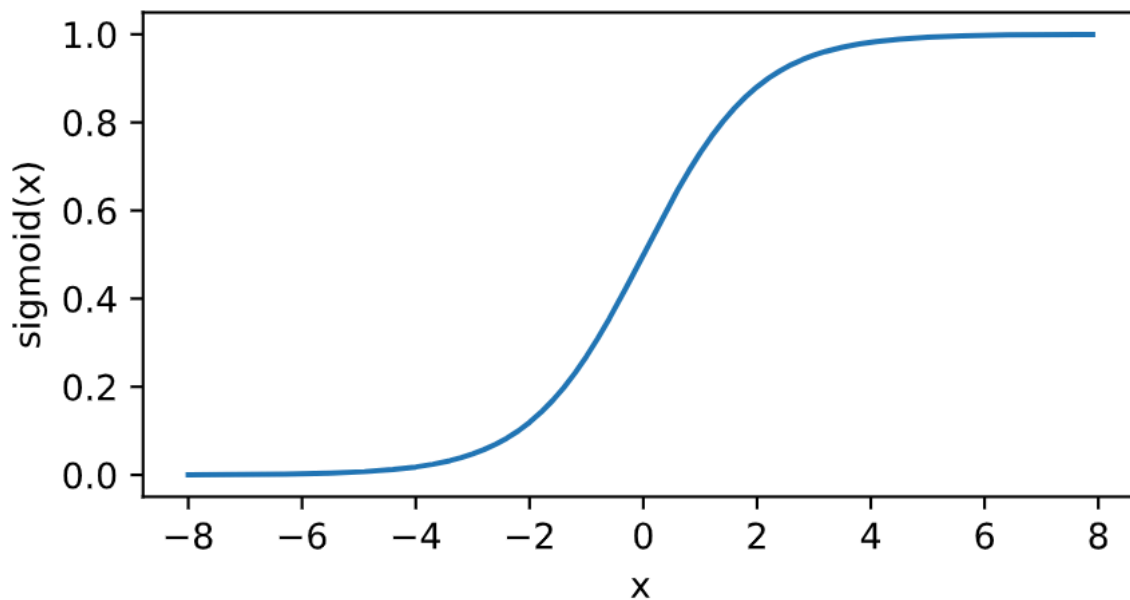


2. sigmoid函数

sigmoid将元素 x 变换到 **0~1**, 定义:

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}$$

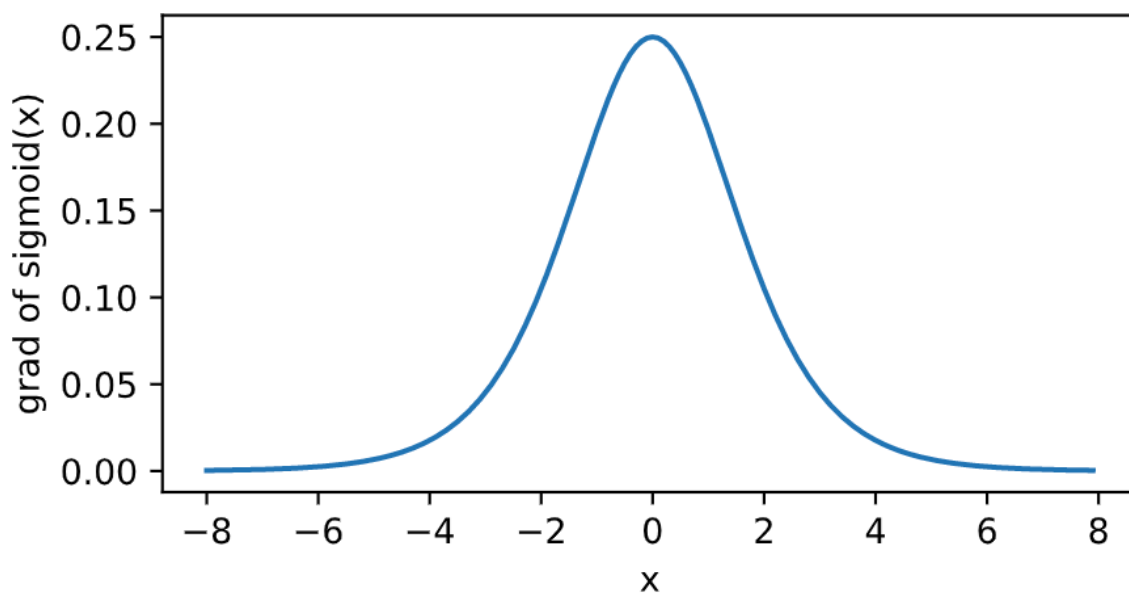
sigmoid example:



且sigmoid的导函数为:

$$\text{sigmoid}'(x) = \text{sigmoid}(x)(1 - \text{sigmoid}(x)).$$

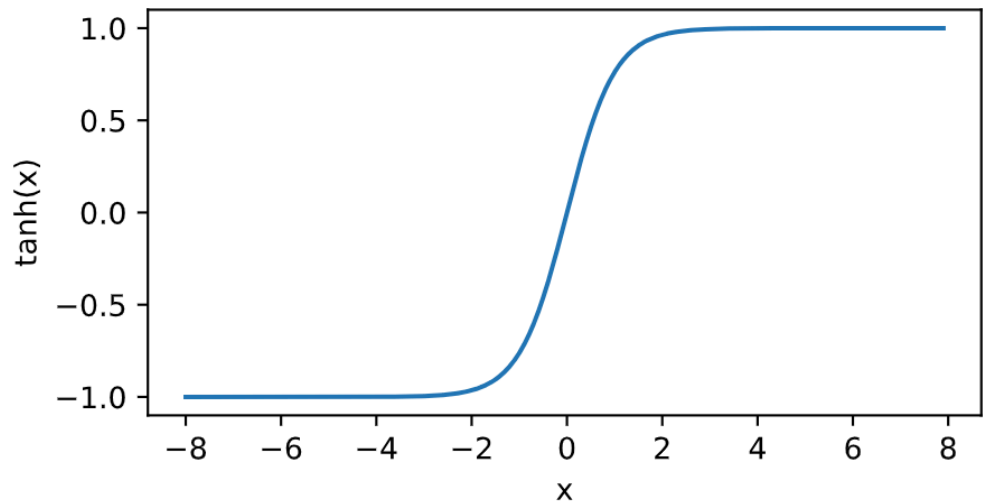
这个导函数值域为 $(0, 0.25]$, 且越接近0越大, $\text{sigmoid}'(0)=\max=0.25$



3. tanh函数

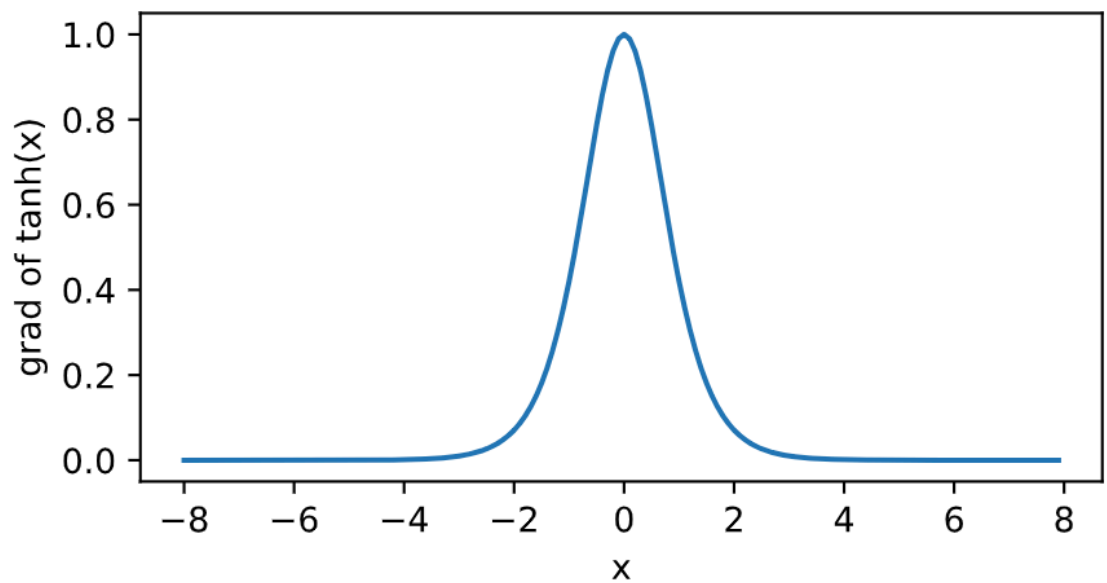
\tanh (双曲正切) 将元素的值变换到 **-1~1** 之间:

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}$$



tan导函数:

$$\tanh_1(x) = 1 - \tanh_2^2(x).$$



- MLP定义

MLP的隐藏层通过激活函数进行变换, MLP的层数和隐藏层中的隐藏单元个数为超参数.

MLP定义如下:

$$H = \phi(XW_h + b_h),$$

$$O = HW_o + b_o,$$

其中 ϕ 表示激活函数, O 为输出. 分类时可以对 O 做softmax预案算, 并使用softmax回归中的交叉熵损失函数.

4.深度学习计算

模型构造

- 继承 `Module` 类来构造

需要重载 `Module.__init__()`, `Module.forward()`

e.g.

```
import torch
import torch.nn as nn
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.hidden = nn.Linear(784, 256)
        self.act = nn.ReLU()
        self.output = nn.Linear(256, 10)

    def forward(self, x):
        a = self.act(self.hidden(x))
        return self.output(a)

## Use
X = torch.rand(2, 784)
net = MLP()
print(net)
net(X)
```

- 直接使用预构建好的类
 - `Sequential`

```
net = Sequential(
    nn.Linear(784, 256),
    nn.ReLU(),
    nn.Linear(256, 10)
)
```

- **ModuleList**

```
## Use
net = nn.ModuleList([nn.Linear(784, 256), nn.ReLU()])
net.append(nn.Linear(256, 10))
print(net[-1])
print(net)
```

P.S

ModuleList 仅仅是一个存储各种模块的列表, 模块之间没有联系也没有顺序, 需要自己实现 **forward()**, 所以直接执行 **net(X)** 会报错.

而 **Sequential** 内的模块要按顺序排列, **forward()** 功能已经实现.

- **ModuleDict**

ModuleDict 接收一个子模块的字典作为输入, 可以用key访问

```
net = nn.ModuleDict({
    'linear' : nn.Linear(784, 256),
    'act' : nn.ReLU()
})
net['output'] = nn.Linear(256, 10)
print(net['linear'])
print(net.output)
print(net)
```

- 构建灵活复杂的模型

get_constant() 可以创建不被迭代的常参数

e.g.

```

import torch
import torch.nn as nn
class FancyMLP(nn.Module):
    def __init__(self, **kwargs):
        super(FancyMLP, self).__init__(**kwargs)

        self.rand_weight = torch.rand((20, 20), requires_grad=False)
        self.linear = nn.Linear(20, 20)

    def forward(self, x):
        x = self.linear(x)
        x = nn.functional.relu(torch.mm(x, self.rand_weight.data) + 1)

        # Reuse full connect layer.
        x = self.linear(x)
        # Get scalar to compare.
        while x.norm().item() > 1:
            x /= 2
        if x.norm().item() < 0.8:
            x *= 10
        return x.sum()

## Use
X = torch.rand(2, 20)
net = FancyMLP()
print(net)
net(X)

```

模型参数的访问, 初始化

- `parameters()`, `named_parameters()`

前者可以返回参数, 后者返回参数与参数名

```

for name, param in net.named_parameters():
    print(name, param.size())

```

- `init.normal_()`

`nn.init.normal_()`可以用来初始化模型参数:

```

for name, param in net.named_parameters():
    if 'weight' in name:
        init.normal_(param, mean=0, std=0.01)

```

或者使用参数来初始化:

```
for name, param in net.named_parameters():
    if 'bias' in name:
        init.constant_(param, val=0)
```

自定义层

- 不含模型参数

```
class CenteredLayer(nn.Module):
    def __init__(self, **kwargs):
        super(CenteredLayer, self).__init__(**kwargs)
    def forward(self, x):
        return x - x.mean()

## Instantiate layer.
layer = CenteredLayer()
layer(torch.tensor([1, 2, 3, 4]), dtype=torch.float)

## Or be used to construct another model.
net = nn.Sequential(nn.Linear(8, 128), CenteredLayer())
```

- 含模型参数

```
class MyDense(nn.Module):
    def __init__():
        super(MyDense, self).__init__()
        self.params = nn.ParameterList([nn.Parameter(torch.randn(4, 4))
                                         for i in range(3)])
        self.params.append(nn.Parameter(torch.randn(4, 1)))

    def forward(self, x):
        for i in range(len(self.params)):
            x = torch.mm(x, self, params[i])
        return x

net = MyDense()
print(net)
```

读取与存储

- 读写Tensor

Write

```
torch.save(x, 'x.pt')
```

Read

```
x = torch.load('x.pt')
```

- 读写model

1. 读写权重 `state_dict`

Write

```
torch.save(model.state_dict(), 'dict.pt')
```

Read

```
model = MyModel()
```

```
model.load_state_dict(torch.load('dict.pt'))
```

2. 读写整个模型

Write

```
torch.save(model, 'net.pt')
```

Read

```
model = torch.load('net.pt')
```

GPU相关

- GPU是否可用

```
torch.cuda.is_available()
```

- 转换到GPU

```
x = torch.tensor([1, 2, 3])
```

```
x = x.cuda(0)
```

Or

```
x = torch.tensor([1, 2, 3], device=torch.device('cuda' if torch.cuda.is_available() else 'cpu'))
```

P.S.

`x.to(device)` 这个方法为**Non-InPlace**方法,即赋值后才生效

6.循环神经网络

(Recurrent Neural Network, RNN)

语言模型

- 定义

$$P(w_1, w_2, \dots, w_T) = \prod_{t=1}^T P(w_t | w_1, \dots, w_{t-1}).$$

e.g. 一段含有三个字的文本序列的概率

$$P(w_1, w_2, w_3) = P(w_1)P(w_2 | w_1)P(w_3 | w_1, w_2).$$

- n元语法

通过n阶马尔科夫链简化语言模型的计算, 即假设一个词的出现只与前面的n个词相关.

e.g.

n = 1时, 有

$$P(w_3 | w_1, w_2) = P(w_3 | w_2)$$

注: n较小时, n元语法往往不准确. 当n较大时, n元语法需要存储大量的词频和相邻概率.

循环神经网络

现有一个含单隐藏层的MLP, 将通过添加隐藏状态将其变为循环神经网络:

$$H = \phi(XW_{xh} + b_h)$$

隐藏层权重参数为 W_{xh} , 隐藏层偏差参数为 b_h

其输出层输出为:

$$O = HW_{hq} + b_q$$

若是分类问题, 可以使用softmax(O)来计算输出类别的概率分布.

考虑输入数据存在时间相关性的情况:

$$H_t = \phi(X_t W_{xh} + H_{t-1} W_{hh} + b_h)$$

保存上一时间步的隐藏变量 H_{t-1} , 并引入新的权重参数 W_{hh} 来描述如何使用 H_{t-1}

在时间步 t , 输出层的输出类似:

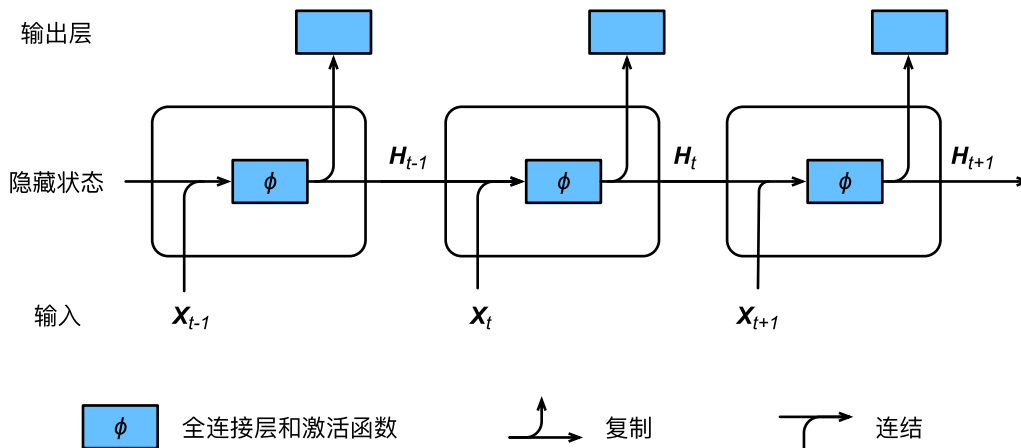
$$O_t = H_t W_{hq} + b_q$$

注:

即使在不同时间步, 循环神经网络也四种使用这些模型参数. 即, RNN的模型参数不随时间步的增加而增长.

如图为RNN在3个相邻时间步的计算逻辑.

在时间 t 时, 相当于将输入 X_t 和 H_{t-1} 连结后输入一个激活函数为 ϕ 的全连接层. 此全连接层的输出为 H_{t+1}



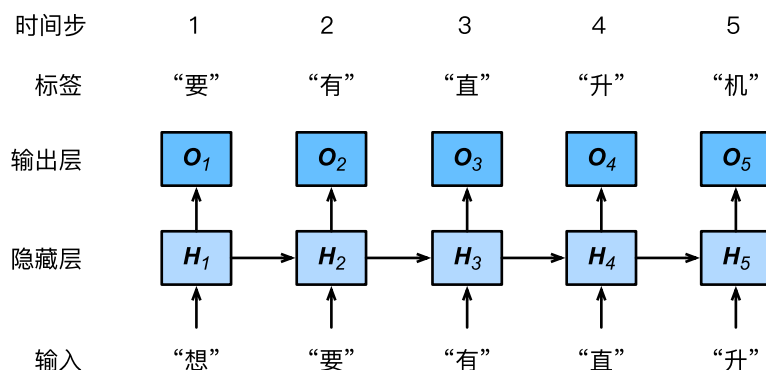
- 字符级循环神经网络

(character-level RNN)

设样本序列为"想要有直升机". 在训练时,

对每个时间步的输出层输出使用**softmax**运算, 然后使用交叉熵损失函数计算它与标签的误差.

对于时间步3, 输出 O_3 取决于基于"想", "要", "有"生成下一个词的概率分布与该时间步的标签"直"



- one-hot向量

将字符转化为向量输入到神经网络中的简易方法

e.g.

根据字符的index生成list如: [1, 0, 2] # ['你', '是', '谁']

生成的one-hot向量如:

0, 1, 0], [1, 0, 0], [0, 0, 1

- 困惑度 (perplexity)

模型的预测结果与样本的契合程度, 困惑度越低契合越准确.

Perplexity可以认为是average branch factor (平均分支系数).

模型的PPL下降到90, 可以直观地理解为, 在模型生成一句话时下一个词有90个合理选择, 可选词数越少, 我们大致认为模型越准确。

任何一个有效模型的困惑度必须小于类别个数.

- 实现

见 Source Code: RNNTest.py 和

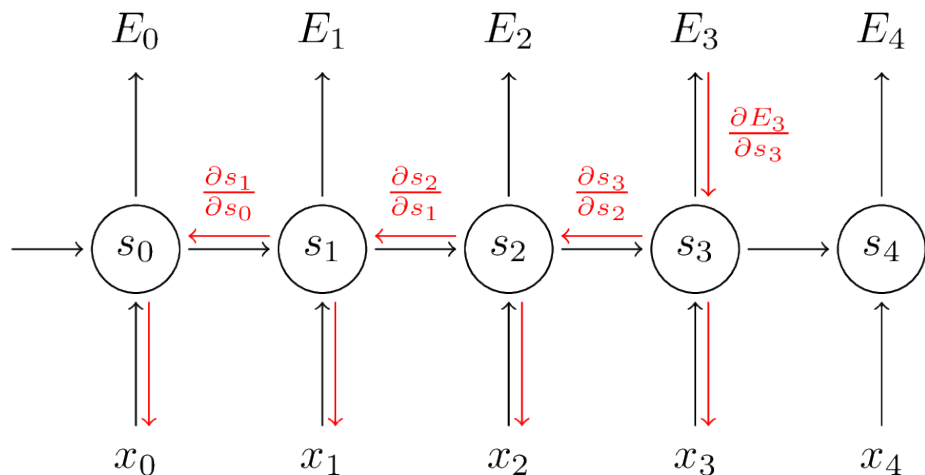
见 Source Code: RNNTestSimple.py

- RNN中的随时间反向传播和梯度消失 / 爆炸

(Back-Propagation Through Time, BPTT)

[ref](#)

1. BPTT是什么



对于 E_3 的偏导, 有如下推导过程:

$$\begin{aligned}
 \frac{\partial E_3}{\partial W} &= \frac{\partial E_3}{\partial \hat{y}^3} \frac{\partial \hat{y}^3}{\partial s_3} \frac{\partial s_3}{\partial W} \\
 &= \sum_{k=0}^3 \frac{\partial E_3}{\partial \hat{y}^3} \frac{\partial \hat{y}^3}{\partial s_3} \frac{\partial s_3}{\partial s_k} \frac{\partial s_k}{\partial W}
 \end{aligned} \tag{4}$$

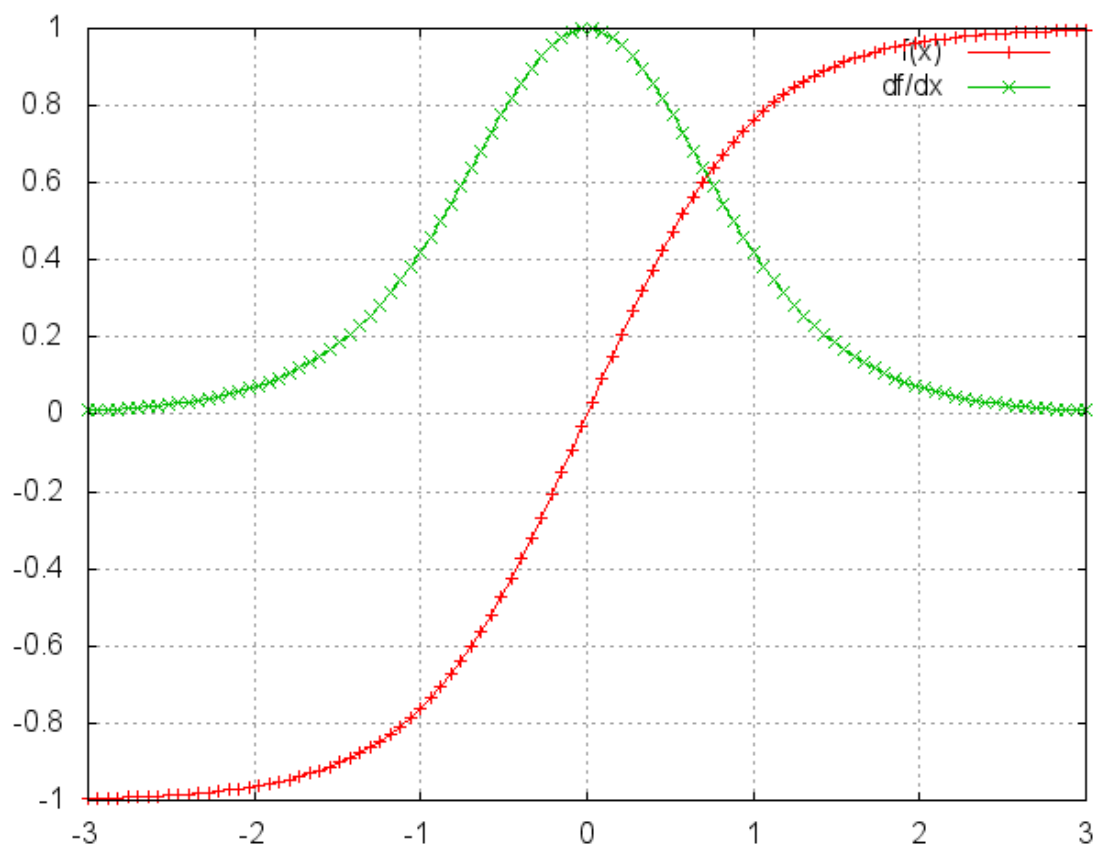
这就是BPTT的计算方式, 由于需要对各时间步求和, 而序列又可能很长, 所以会出现RNN难以训练的问题。

2. 梯度消失 / 爆炸问题

将上式重写为:

$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^3 \frac{\partial E_3}{\partial \hat{y}^3} \frac{\partial \hat{y}^3}{\partial s_3} \left(\prod_{j=k+1}^3 \frac{\partial s_j}{\partial s_{j-1}} \right) \frac{\partial s_k}{\partial W} \tag{6}$$

由于激活函数为tanh, 其图像与导函数图像如下:



可以看到, \tanh 的两端都有接近0的导数, 参数矩阵将快速收敛到0, 即梯度消失.
当参数矩阵的值很大时, 权重将变为NaN, 即梯度爆炸.

P.S.

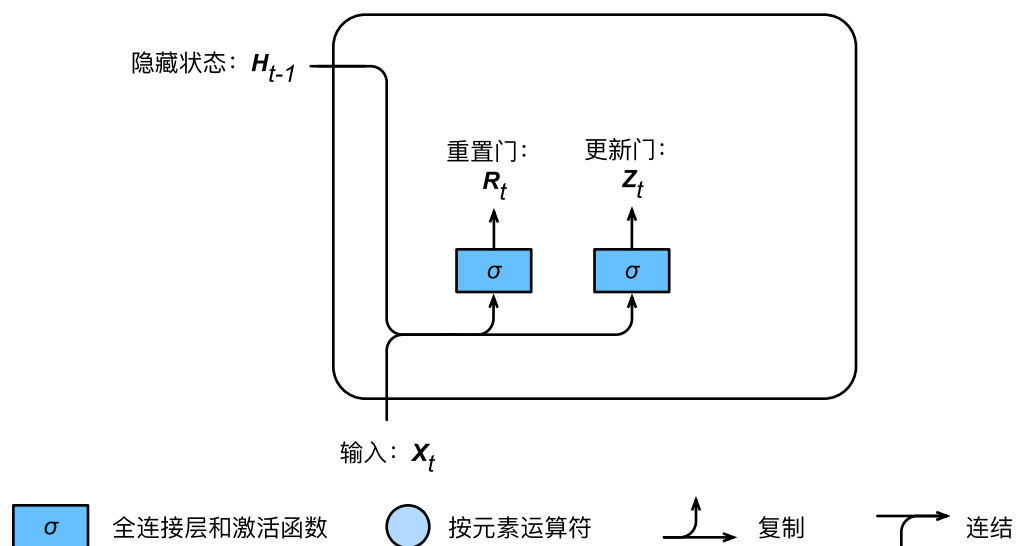
对于梯度消失, 可以采用ReLU函数代替 \tanh 作为激活函数, 或是采用LSTM或GRU架构.

对于梯度爆炸, 可以采用梯度剪切 (超过阈值时重设)

门控循环单元 (GRU)

(Gated Recurrent Unit, GRU)

- 重置门和更新门



如图, GRU中的重置门与更新门的输入为时间步输入 X_t 与上一时间步隐藏状态 H_{t-1} , 输出由激活函数为sigmoid的全连接层得到.

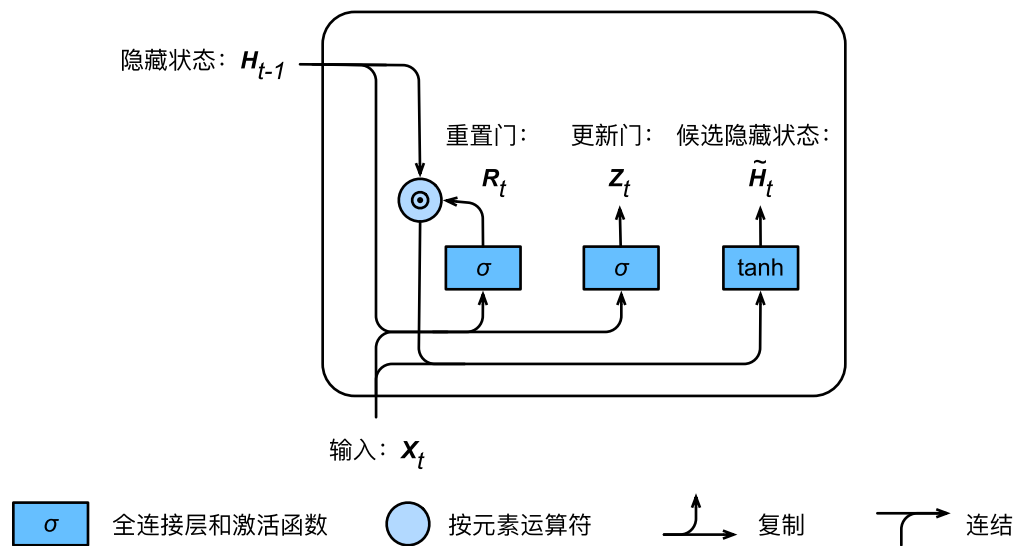
其中,

$$R_t = \sigma(X_t W_{xr} + H_{t-1} W_{hr} + b_r),$$

$$Z_t = \sigma(X_t W_{xz} + H_{t-1} W_{hz} + b_z),$$

因为 $\text{sigmoid}()$ 的值域为 $[0, 1]$, 所以 R_t, Z_t 均在 $[0, 1]$

- 候选隐藏状态

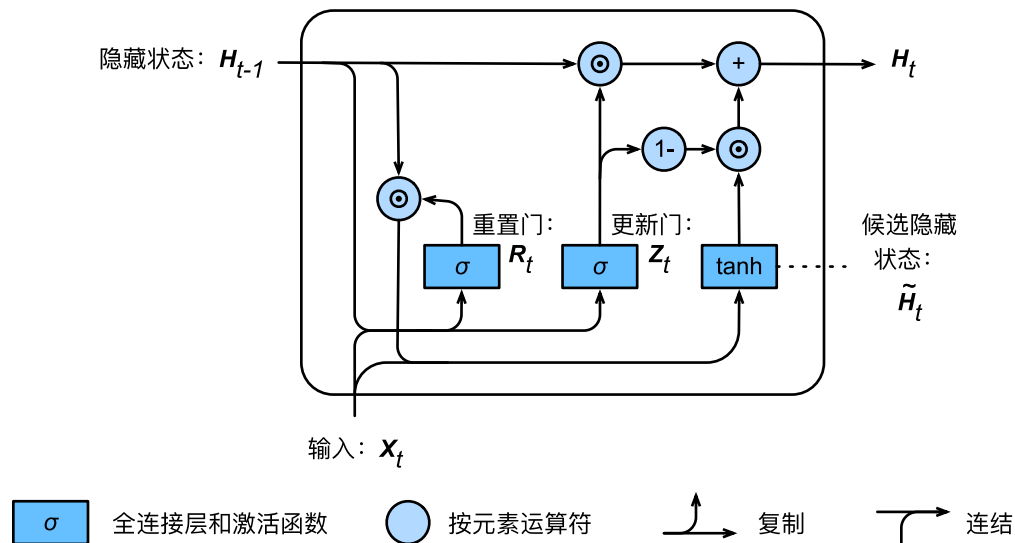


通过重置门的输出来决定保留多少来自上一时间步的隐藏状态,再讲通过tanh计算出候选隐藏状态输出.值域为[-1, 1]

用公式表达为:

$$\tilde{H}_t = \tanh(X_t W_{xh} + (R_t \odot H_{t-1}) W_{hh} + b_h),$$

- 隐藏状态



最后当前时间步的 H_t 为:

$$H_t = Z_t \odot H_{t-1} + (1 - Z_t) \odot \tilde{H}_t.$$

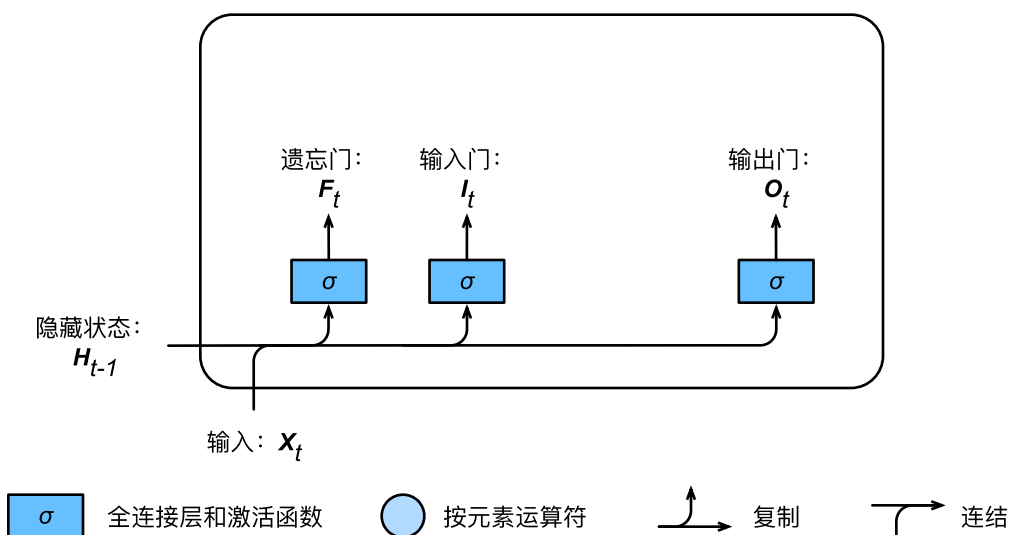
总结:

- 重置门有助于捕捉**短期**内的依赖关系
- 更新门有助于捕捉**长期**内的依赖关系

长短期记忆 (LSTM)

(Long Short-Term Memory, LSTM)

- 输入门, 遗忘门, 输出门



输入: X_t, H_{t-1}

输出: $F_t, I_t, O_t \in [0, 1]$

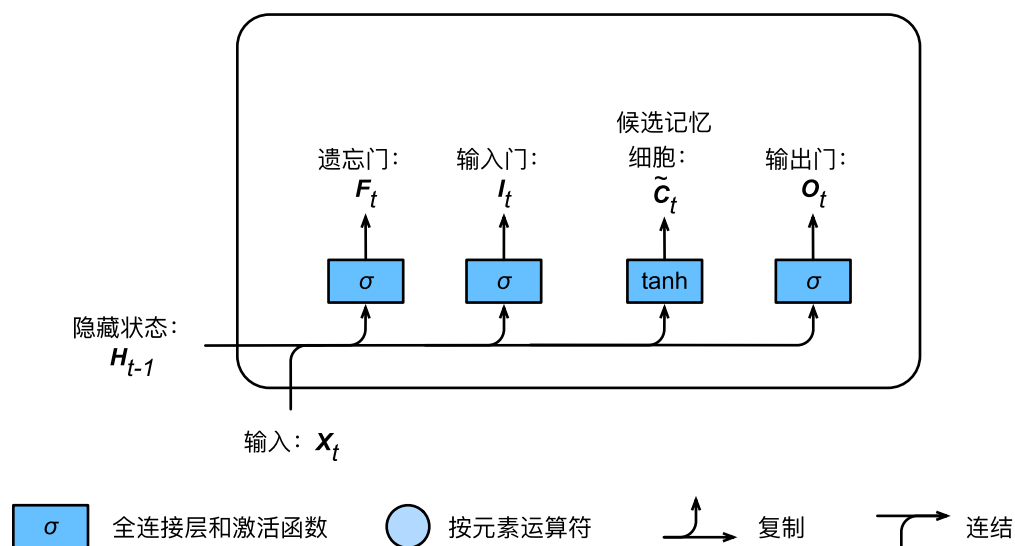
公式如下:

$$I_t = \sigma(X_t W_{xi} + H_{t-1} W_{hi} + b_i), \in [0, 1]$$

$$F_t = \sigma(X_t W_{xf} + H_{t-1} W_{hf} + b_f), \in [0, 1]$$

$$O_t = \sigma(X_t W_{xo} + H_{t-1} W_{ho} + b_o), \in [0, 1]$$

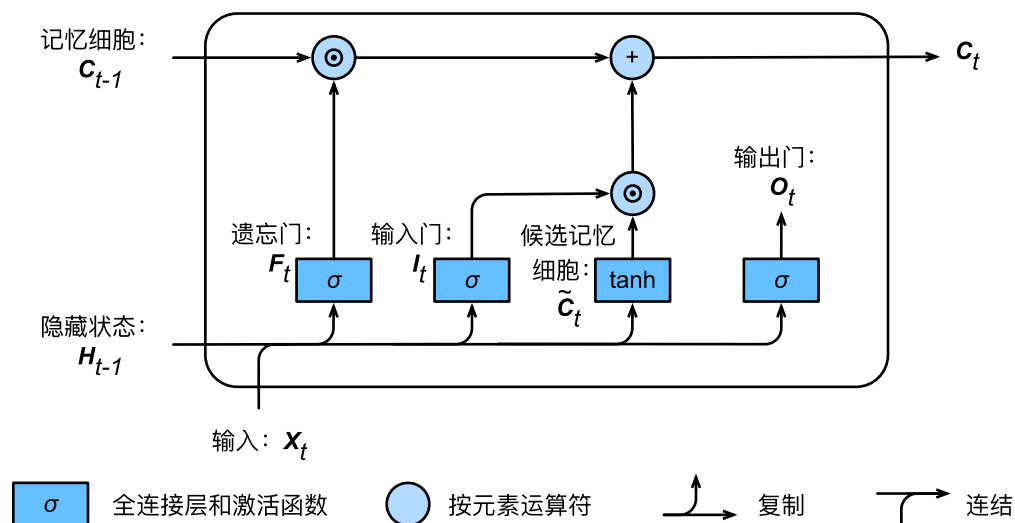
- 候选记忆细胞



候选记忆细胞为 \tilde{C}_t :

$$\tilde{C}_t = \tanh(X_t W_{xc} + H_{t-1} W_{hc} + b_c), \in [-1, 1]$$

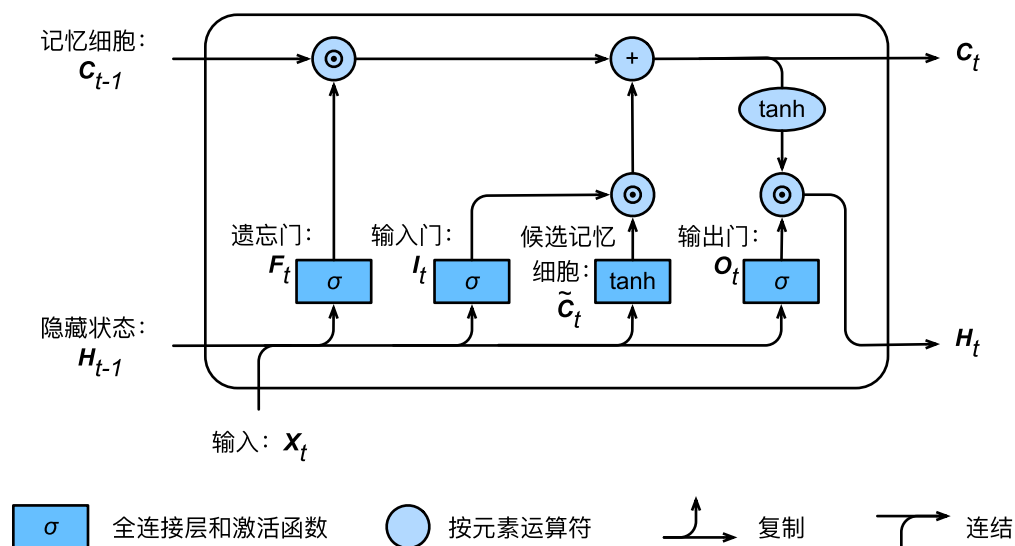
• 记忆细胞



$$C_t = F_t \odot C_{t-1} + I_t \odot \tilde{C}_t$$

此处如果将遗忘门 F_t 近似1, 并将输入门 I_t 近似0, 就可以记忆更长时间步的信息.

• 隐藏状态



输出状态表示为:

$$H_t = O_t \odot \tanh(C_t).$$

当输出门近似1时, C_t 将输出为隐藏状态, 近似0时, 只在记忆细胞自己保留.

自然语言处理 (NLP)

词向量

ref

- 神经概率语言模型

构造 最大对数似然函数, 来对N-gram模型进行函数化:

$$L = \sum_{w \in C} \log F(w, \text{Context}(w), \theta)$$

其中,

C 为语料库,

$\text{Context}(w)$ 是词 w 的上下文,

θ 为待训练参数

接下来就是采用『神经网络』来构造『函数 $F()$ 』

神经概率语言模型

$$(\text{Context}(w), w) \rightarrow X_w \rightarrow Z_w \rightarrow y_w$$

其中,

$(\text{Context}(w), w)$ 为训练样本, $\text{Context}(x)$ 取前面 $n-1$ 个词

X_w 为投影层向量, 将训练样本的前 $n-1$ 个词的词向量收尾拼接在一起构成 X_w

Z_w 为隐藏层向量, $Z_w = \tanh(WX_w + p)$

y_w 为输出层向量, $y_w = Uz_w + q$

在对 y_w 做Softmax归一化后, y_w 的分量就表示当前词是 w 的概率.

该神经网络的参数有『词向量 $v(w)$ 』,『神经网络参数 W, p, U, q 』. 确定了这些参数, 就相当于确定了『函数 F 』的参数, 就能求得整个句子的概率.

优点: 可以求出词之间的相似度, 概率不可能为0

缺点: 计算量太大

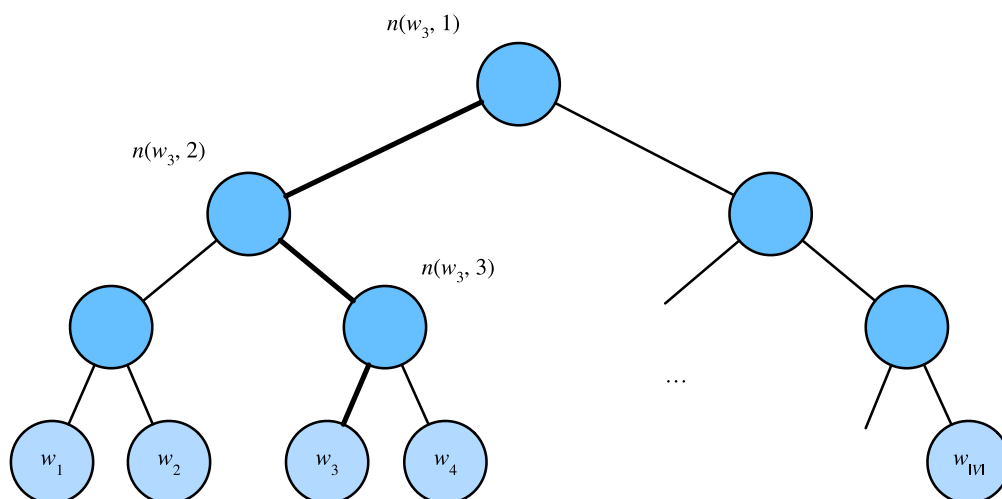
- Word2vec相比于神经概率语言模型的改进

- 优化网络结构

删去隐藏层, 将投影层的节点数由 $\text{num_context} * \text{dimen_v}$ 变成 dimen_v , 简化计算从而能训练更多的数据

- 优化Softmax

采用树型Softmax Hierarchical Softmax + Huffman树 或是

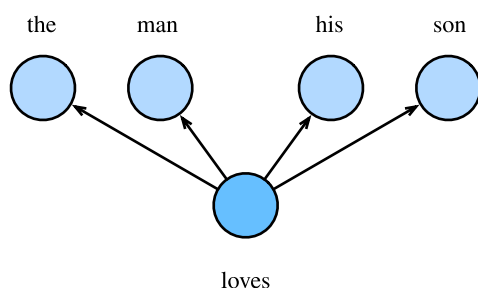


负采样的方法简化Softmax的计算 (不使用全部的词向量来做Softmax的分母, 只随机选用一些负样本)

根据上述的改进, 产生了两个模型: 跳字模型 与 连续词袋模型.

- 跳字模型 (skip-gram)

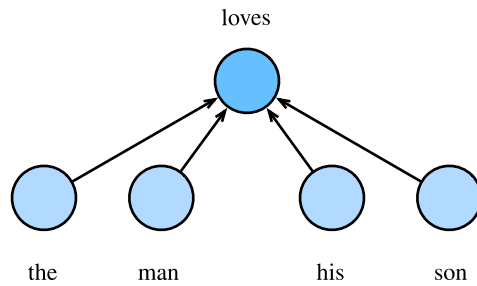
跳字模型关心的是, 给定中心词A, 生成邻近的B, C, D的概率.



即:

$$P(\text{"the", "man", "his", "sun", | "loves"}).$$

- 连续词袋模型(Continuous Bag-of-Words Model, CBOW)
连续词袋模型与跳字模型类型, 最大的不同在于, 后者是基于某个中心词前后的背景词来生成该中心词.



即:

$$P(\text{"loves"} | \text{"the", "man", "his", "son"}).$$

- Word2vec实现
见 `Source Code: word2vec.py`
- GloVe模型
当词典中含有大量生僻词的时候, 交叉熵损失函数会带来过大的计算开销, 并且预测结果也并不准确.
为了应对这个问题, GloVe(Global Vectors) 模型采用平方损失函数, 并且最终最小化损失函数.
- 实现: Sentiment classification on Large Movie Review
见 `Source Code: IMDB_Emotions.py`