深度学习应用与工程实践
6. 训练-1
6. Training-Part 1

李冰
Bing Li
Tenure-track Associate Professor
Academy of Multidisciplinary Studies
Capital Normal University

回顾

深度学习应用

计算机视觉 (CV)

MLP　CNN　UNet　YoLo　MobileNet

自然语

RNN　LSTM/GRU　**Transformer**

生成式模型

GAN　**Diffusion**　**GPT**

工程实践

模型设计

PyTorch

训练关键技术

部署

- CNN架构的基本构成模块
- CNN设计原则

新增内容

这节课

深度学习应用

计算机视觉 (CV)

MLP | CNN | UNet | YoLo | MobileNet

自然语

RNN | LSTM/GRU | **Transformer**

生成式模型

GAN | **Diffusion** | **GPT**

工程实践

模型设计

PyTorch

训练关键技术

部署

- 构建容易训练的模型
- 训练成功需要考虑的问题

新增内容

优化策略

- 激活函数
- 神经网络架构
- 损失函数 & 优化
- 权重初始化

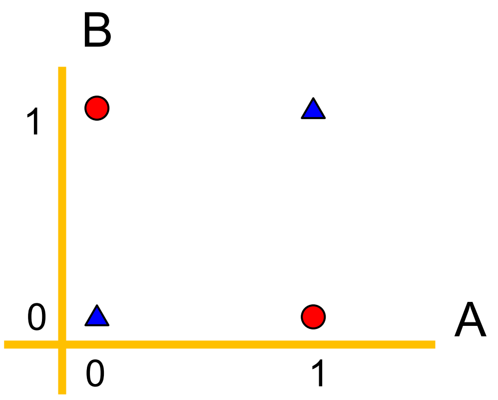
激活函数

- 激活函数使DNN能够学习更复杂的特征.
- 没有激活函数的话，一个多层的神经网络退化为单层神经网络，因为一个多层的线性变换与单层线性变换等价.

XNOR(异或门)的真值表

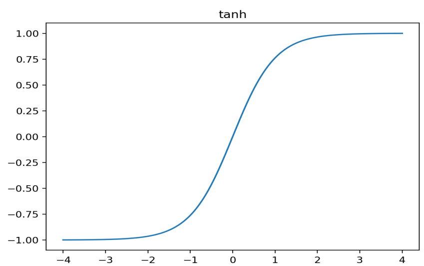
输入		输出
A	**B**	**Y**
0	0	0
0	1	1
1	0	1
1	1	0

Q: **线性**网络能够模拟异或门的工作吗?



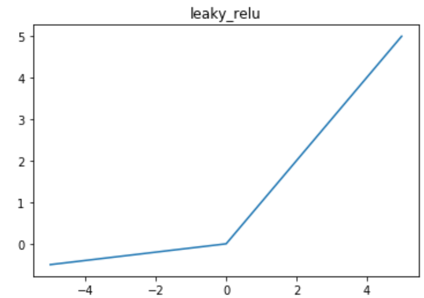
$$Y = w1\,A + w2\,B$$

激活函数一览



Tanh

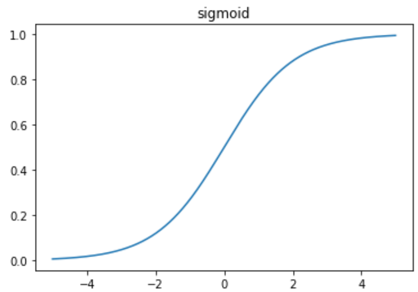
$a(x) = \tanh(x)$



Leaky ReLU

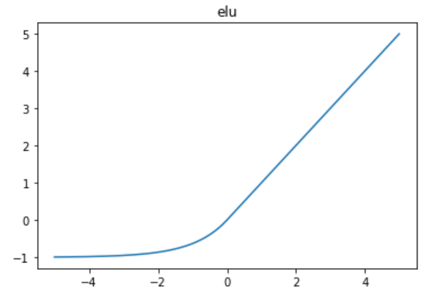
$a(x) = \max(\alpha x, x)$

$\alpha \in (0,1)$



Sigmoid

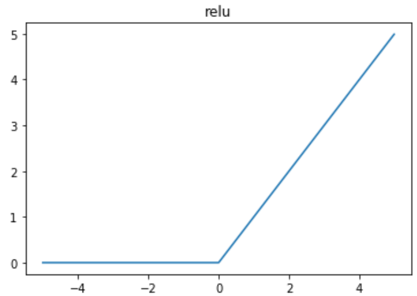
$$a(x) = \frac{1}{1 + e^{-x}}$$



ELU

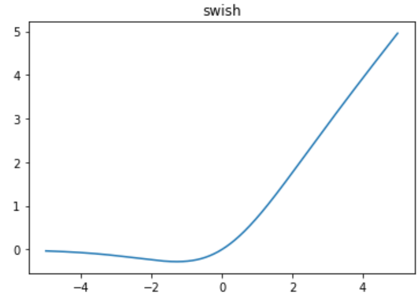
$$a(x) = \begin{cases} x & x \geq 0 \\ \alpha(e^{x} - 1) & x < 0 \end{cases}$$

$\alpha \in (0,1)$



Rectified linear unit(ReLU)

$a(x) = \max(0, x)$



Swish

$a(x) = x \cdot sigmoid(x)$

Tanh

- 水平方向压缩，垂直方向扩展。
- 当输入饱和的时候，梯度接近零

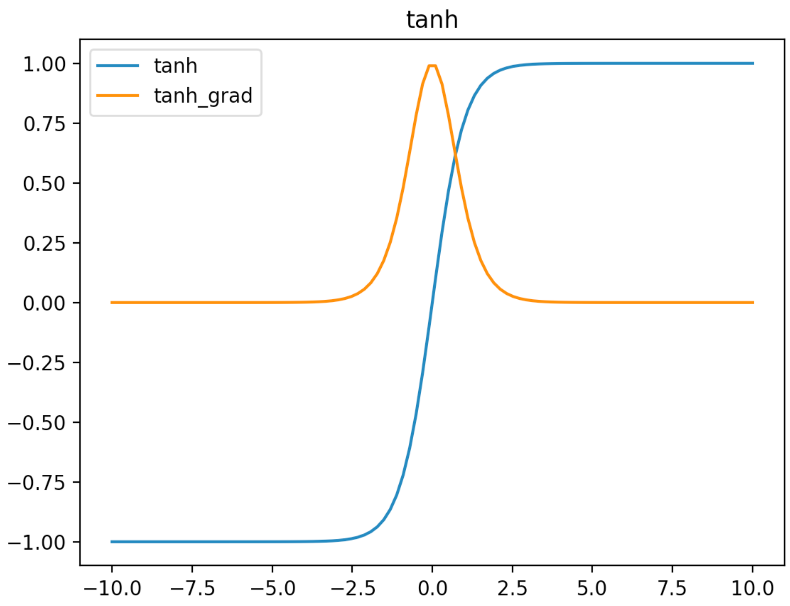
优点:

– 有界，零为中点

问题:

– 梯度消失

– 幂运算相对耗时




```
def tanh_grad(x):
    return 1 - np.power(np.tanh(x), 2)
```

Sigmoid

- 当输出饱和的时候，梯度接近零
  - 通常只用在最后一层，用来调整和限制预测值。

优点:

– 约束输出值。使有界

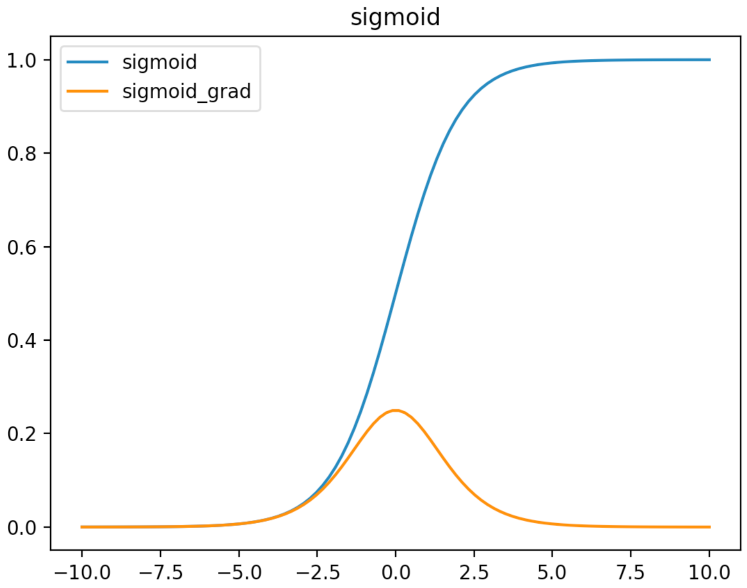
缺点:

– 梯度消失

– 幂运算相对耗时

– 输出不以零为中心点

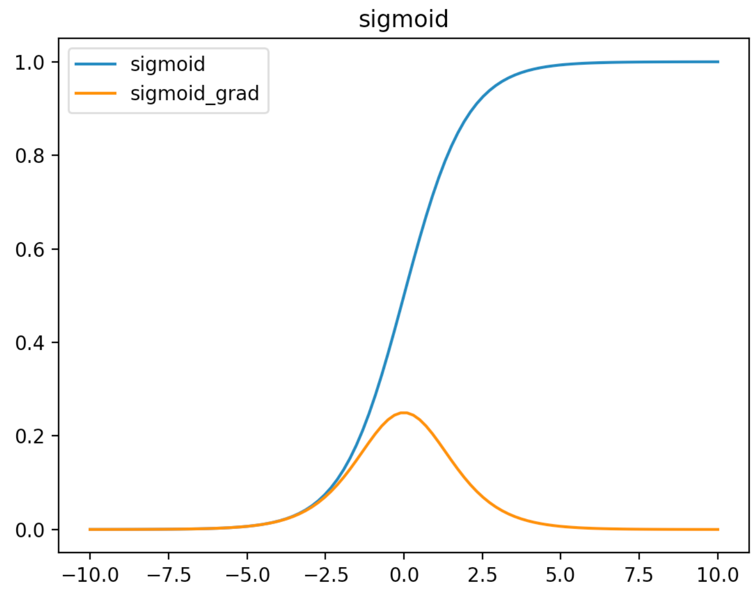
- 会导致神经网络收敛较慢



$$a(x) = \frac{1}{1 + e^{-x}}$$

Sigmoid

$$a(x) = \frac{1}{1 + e^{-x}}$$




```
def sigmoid(x):
    return 1/ (1 + np.exp(-x))


def sigmoid_grad(x):
    sig = sigmoid(x)
    return sig*(1 - sig)
```

Softmax

- Sigmoid 用在二分类问题中.
- 多分类问题，用Softmax, 预测结果是一个有界输出。

$$y = \begin{bmatrix} 2.0 \\ 1.0 \\ 0.1 \end{bmatrix} \longrightarrow S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}} \longrightarrow S(y) = \begin{bmatrix} 0.66 \\ 0.24 \\ 0.10 \end{bmatrix}$$

与Sigmoid类似, Softmax 只用在**最后一层**.

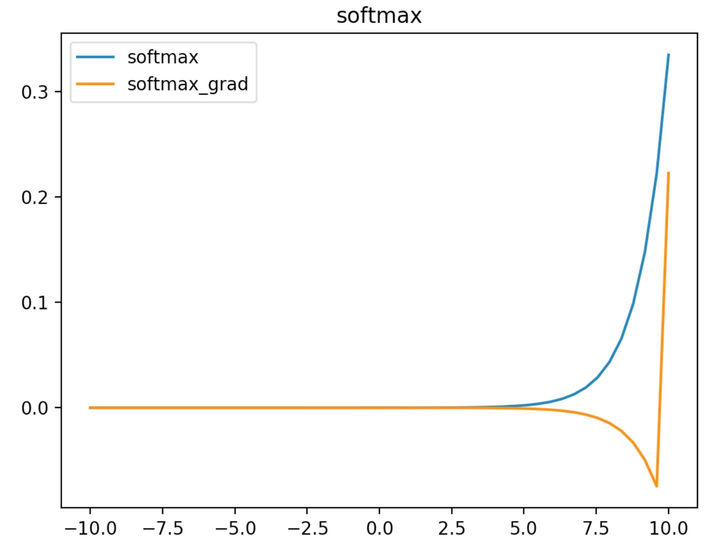
优点:

– 有界输出

缺点:

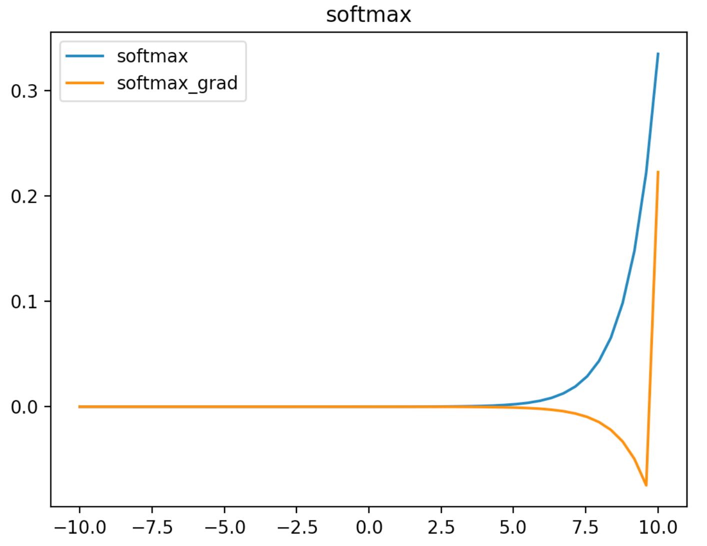
– 梯度消失

– 输出中心非零



Softmax

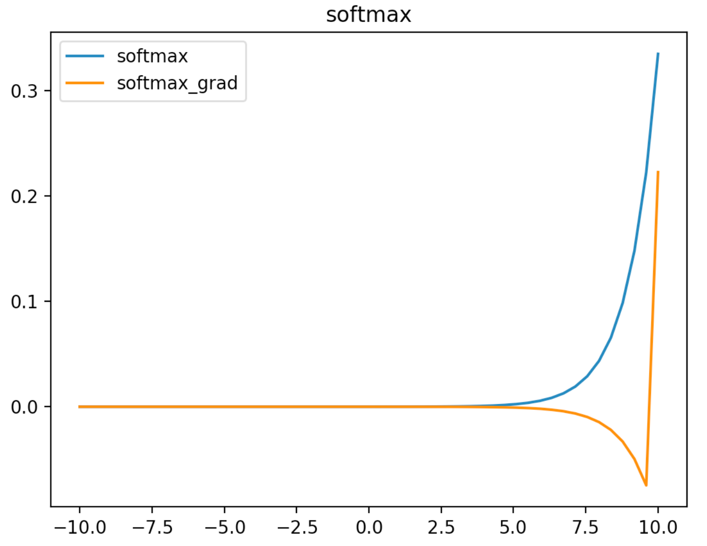
$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}} = \frac{e^{y_i - \max(y)}}{\sum_j e^{y_j - \max(y)}}$$




```
def softmax(x):
    orig_shape = x.shape
    if len(x.shape) > 1:
        # Matrix
        # shift max whithin each row
        constant_shift = np.max(x, axis=1).reshape(1, -1)
        x -= constant_shift
        x = np.exp(x)
        normlize = np.sum(x, axis=1).reshape(1, -1)
        x /= normlize
    else:
        # vector
        constant_shift = np.max(x)
        x -= constant_shift
        x = np.exp(x)
        normlize = np.sum(x)
        x /= normlize
    assert x.shape == orig_shape
    return x
```

Softmax

$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}} = \frac{e^{y_i - \max(y)}}{\sum_j e^{y_j - \max(y)}}$$




```python
def softmax_list(x):
    exp_v = np.exp(x)
    return exp_v/np.sum(exp_v)

def softmax_grad(x):
    softmax_v =softmax_list(x)
    gradients = np.zeros(np.shape(x))
    for i in range(len(x)):
        for j in range(len(softmax_v)):
            if i == j :
                gradients[i] = softmax_v[i]*(1-softmax_v[j])
            else:
                gradients[i] = -softmax_v[j]*softmax_v[i]
    return gradients
```

ReLu

- 只要有输出，梯度是常数.
- ReLU 除了不能用在最后一层，可以用在其他每一层.

优点:

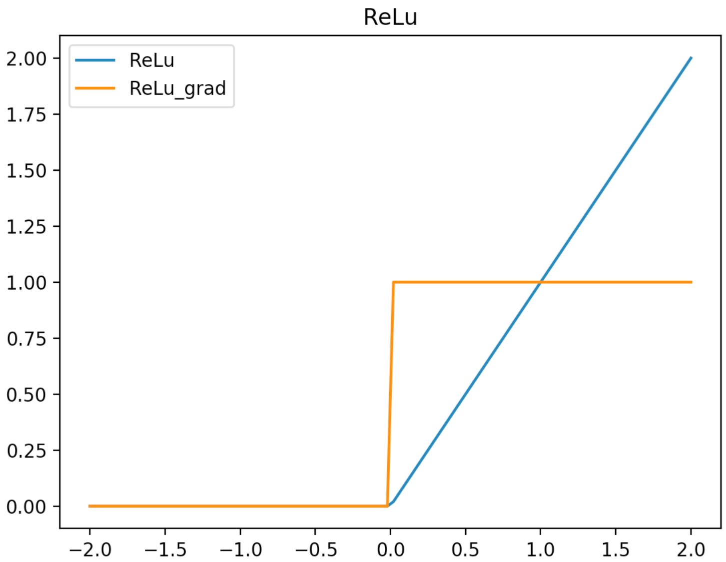
\+ 计算开销小

\+ 生物神经元的工作类似

缺点:

– 某些神经元无法更新 (Dead Neuron)

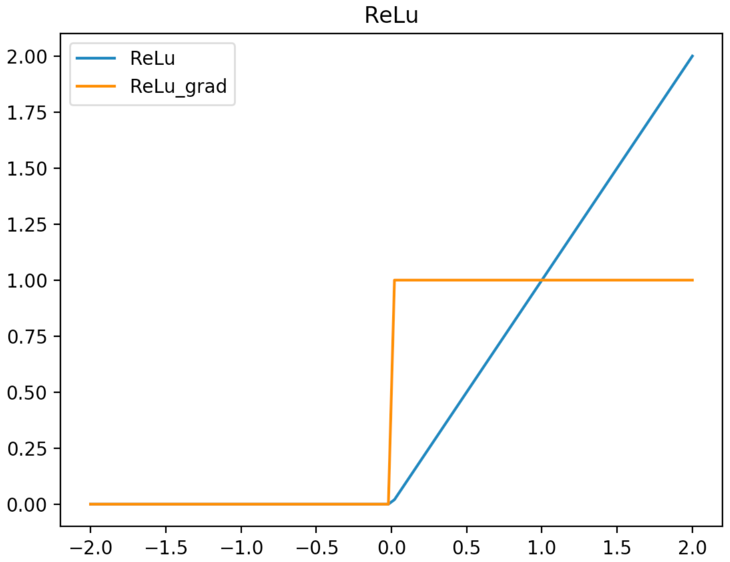
– 输出非零中点.



$$a(x) = \max(0, x)$$

ReLu

$$a(x) = \max(0, x)$$




```
def relu(x):
    y = x.copy()
    y[y<0]=0
    return y
def relu_grad(x, alpha):
    y = x.copy()
    y[x<=0]=0
    y[x>0]=1
    return y
```

Leaky ReLU

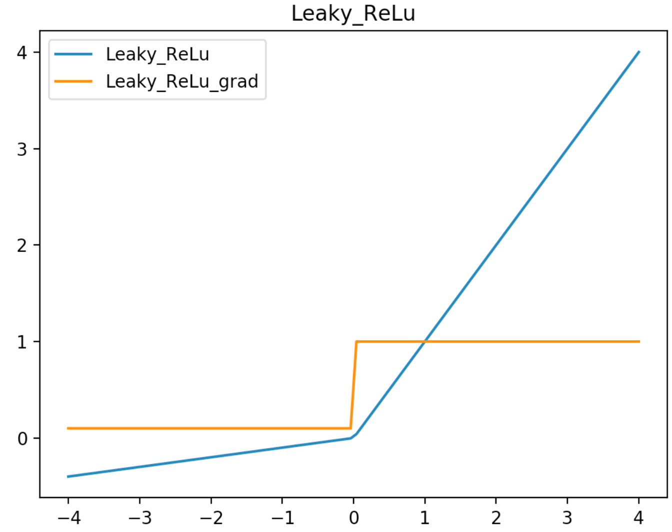
- 当 x < 0 时，它有稳定的梯度

优点:

+ 计算高效
+ 没有Dead Neuron

缺点:

– 输出中点非零.
– 收敛耗费时间长.

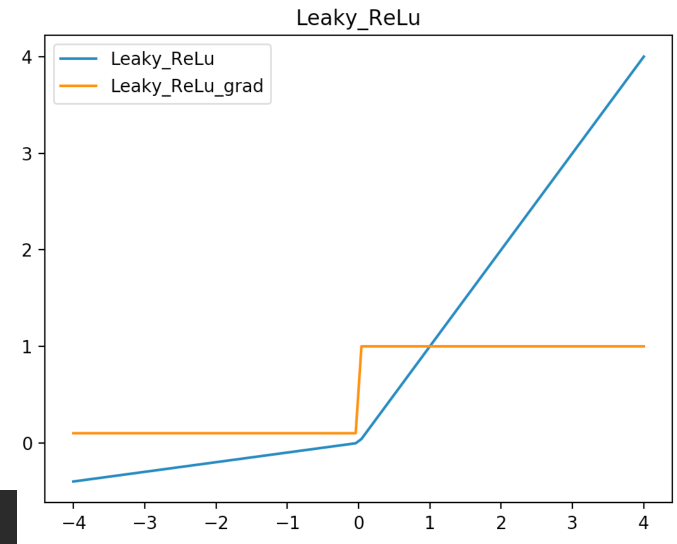


$$a(x) = \max(\boldsymbol{\alpha} x, x)$$

$$\boldsymbol{\alpha} \in (0,1)$$

Leaky ReLU

$$a(x) = \max(\alpha x, x)$$
$$\alpha \in (0,1)$$




```
def leaky_relu(x,alpha):
    y=x.copy()
    alpha_y = alpha*x
    y[x<alpha_y] = alpha_y[x<alpha_y]
    return y

def leaky_relu_grad(x, alpha):
    y = x.copy()
    alpha_y = alpha * y
    y[x<alpha_y]  = alpha
    y[x>alpha_y] = 1
    return y
```

Exponential linear unit (ELU)

ELU使用指数运算来增加DNN的表示能力。

ELU为负饱和部分的噪声提供了更高的鲁棒性。

优点:

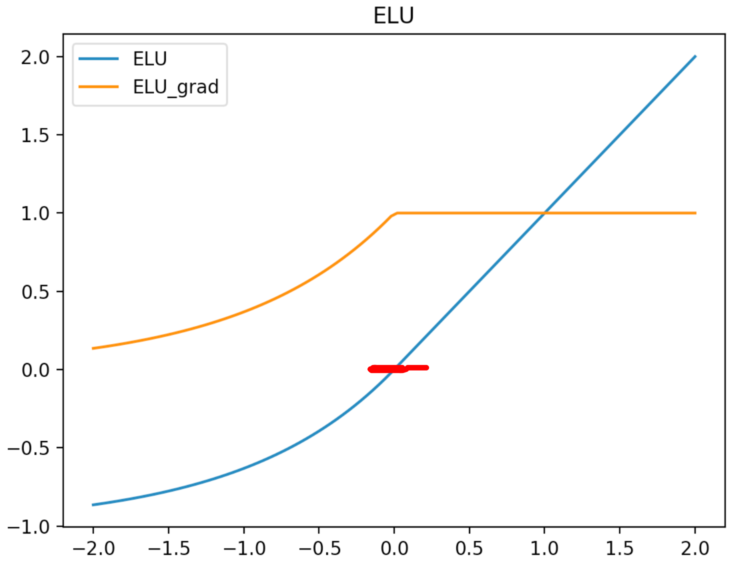
\+ 输出中点接近为零

\+ 神经网络对权重的初始值不敏感

\+ 收敛速度快

缺点:

– 计算复杂.



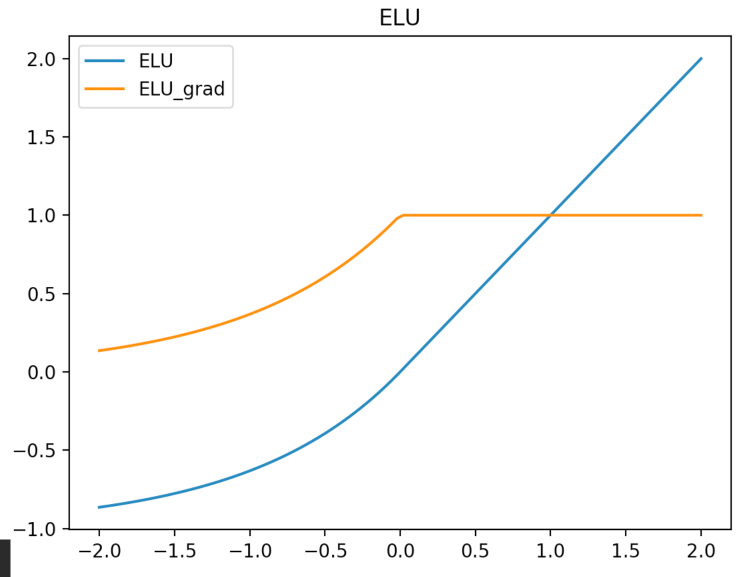
$$a(x) = \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

$\alpha \in (0,1]$, 通常我们用α=1

Exponential linear unit (ELU)

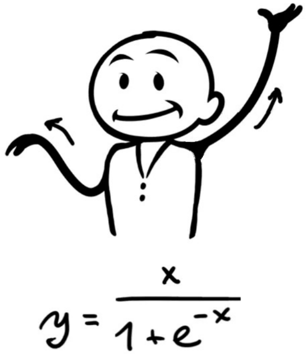
$$a(x) = \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

$\alpha \in (0,1]$, 通常我们用α=1




```
def ELU(x, alpha):
    y = x.copy()
    alpha_y = alpha * (np.exp(x)-1)
    y[x<=0] = alpha_y[x<=0]
    return y
def ELU_grad(x, alpha):
    y= x.copy()
    y[x>0]=1
    y[x<=0]=ELU(x[x<=0],alpha) + alpha
    return y
```

Swish



- Swish 通过搜索激活值.
- 非单调的突起“bump” 提升了性能，保留更多的信息.

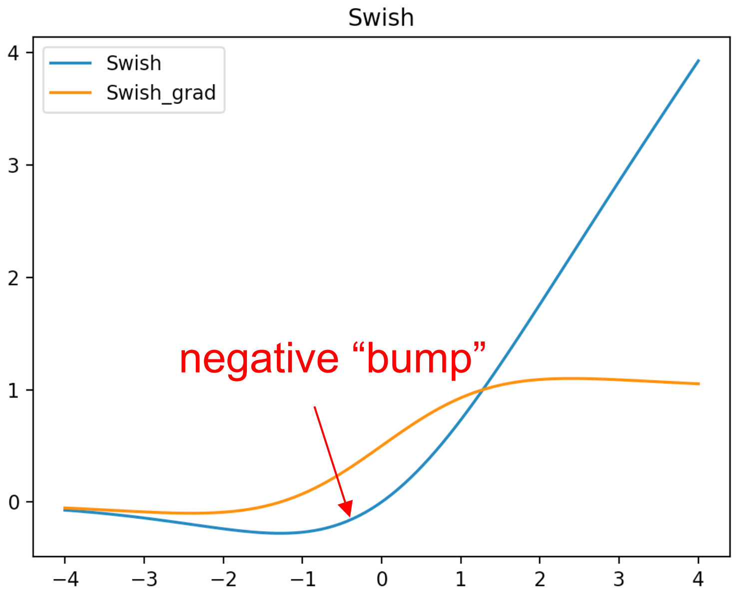
优点:

\+ 提升了DNN的表现能力，尤其是小
模型.

\+ 性能提升是通过这个负突起“bump”得到的.

缺点:

– 计算开销大.

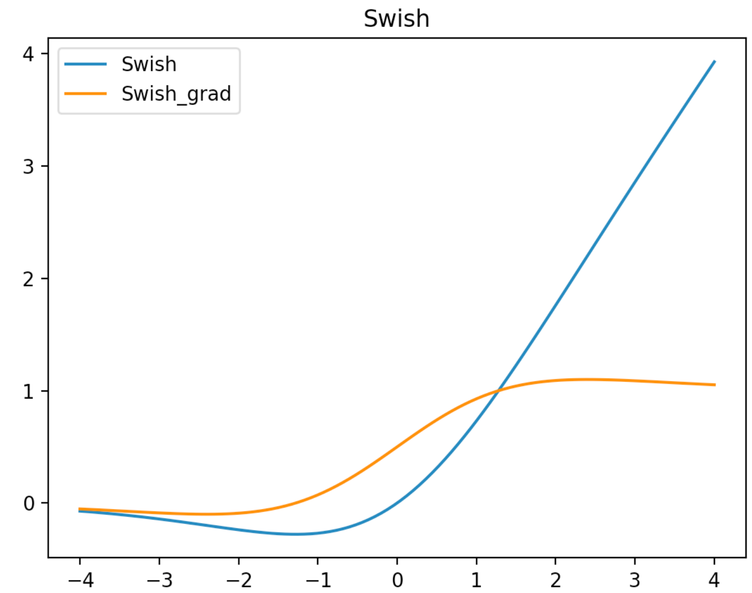


$$a(x) = x\sigma(x)$$

https://youtu.be/1Du1XScHCww
Dance Moves of Deep Learning Activation Functions

Swish

$$a(x) = x\sigma(x)$$




```
def Swish(x):
    return x*sigmoid(x)

def Swish_grad(x):
    #f (x) = f (x) + σ (x) (1 − f (x))

    return Swish(x) + sigmoid(x)*(1-Swish(x))
```

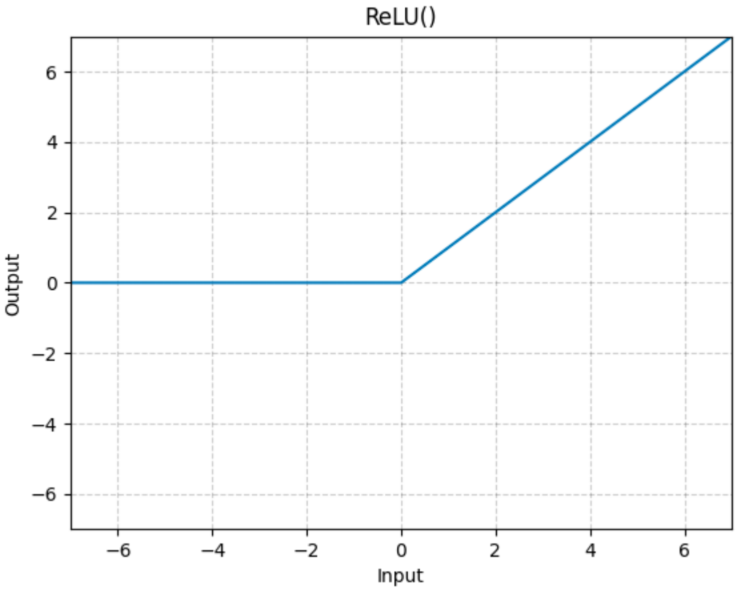
Pytroch 中的激活函数

import torch.nn as nn

torch.nn.ReLU(*inplace=False*)

$$\mathrm{ReLU}(x) = (x)^+ = \max(0, x)$$

```
m = nn.ReLU()
in = torch.randn(2)
output = m(in)
```



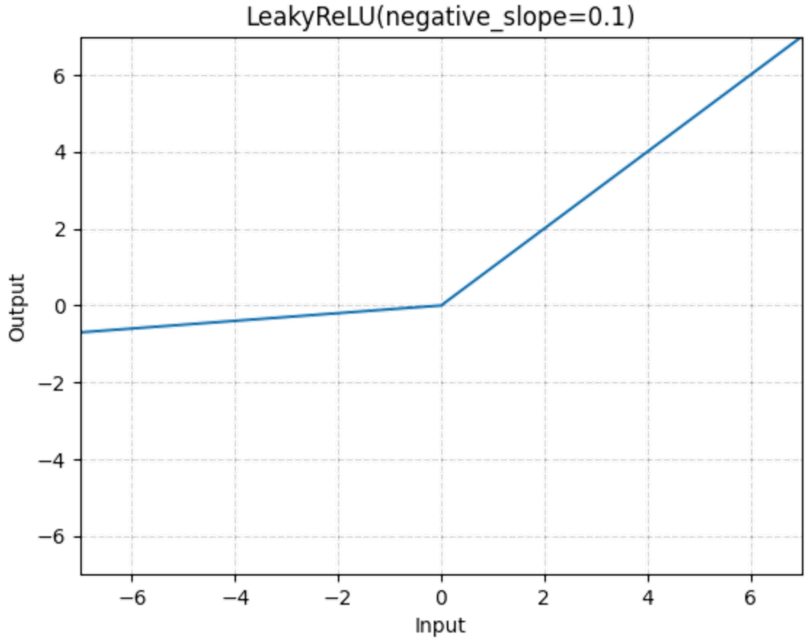
Pytorch 中的激活函数

import torch.nn as nn

torch.nn.LeakyReLU(*negative_slope=0.01*, *inplace=False*)

$$\text{LeakyReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \text{negative\_slope} \times x, & \text{otherwise} \end{cases}$$

```
m = nn.LeakyReLU(0.1)
in = torch.randn(2)
output = m(in)
```



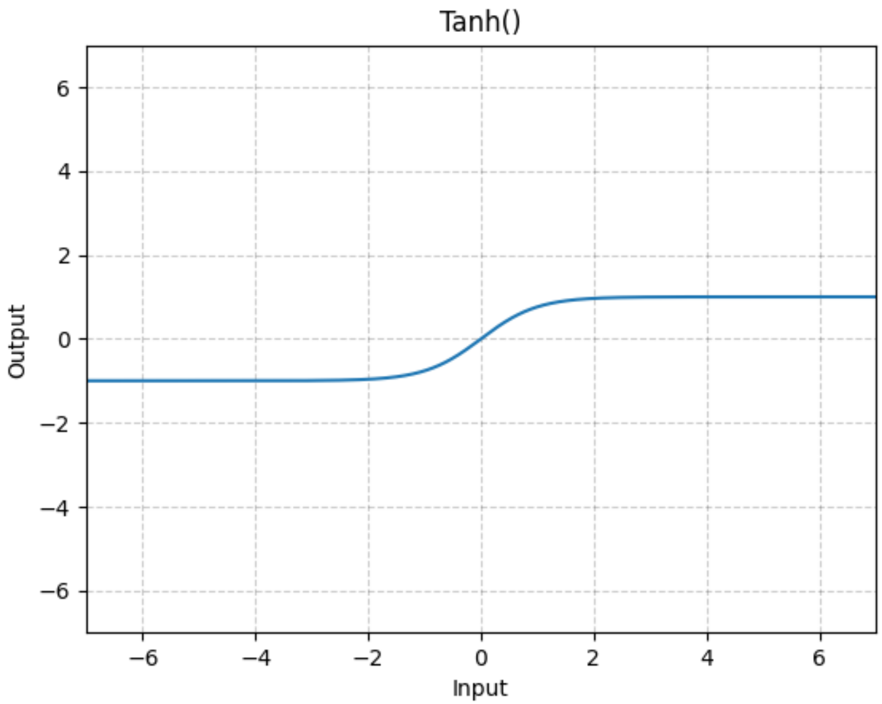
Pytorch 中的激活函数

import torch.nn as nn

torch.nn.Tanh()

$$\text{Tanh}(x) = \tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

```
m = nn.Tanh()
in = torch.randn(2)
output = m(in)
```



Pytorch 中的激活函数

import torch.nn as nn

torch.nn.Softmax(dim=None)

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

参数：被计算的维度

```
m = nn.Softmax(dim=1)
in = torch.randn(2,3)
output = m(in)
```

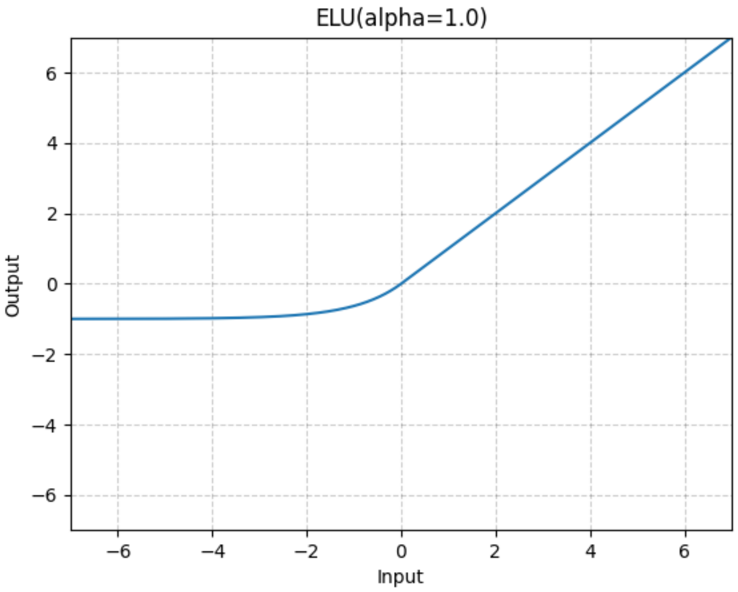
Pytorch 中的激活函数

import torch.nn as nn

torch.nn.ELU(alpha=1.0)

$$\mathrm{ELU}(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha * (\exp(x) - 1), & \text{if } x \le 0 \end{cases}$$

```
m = nn.ELU()
in = torch.randn(2)
output = m(in)
```



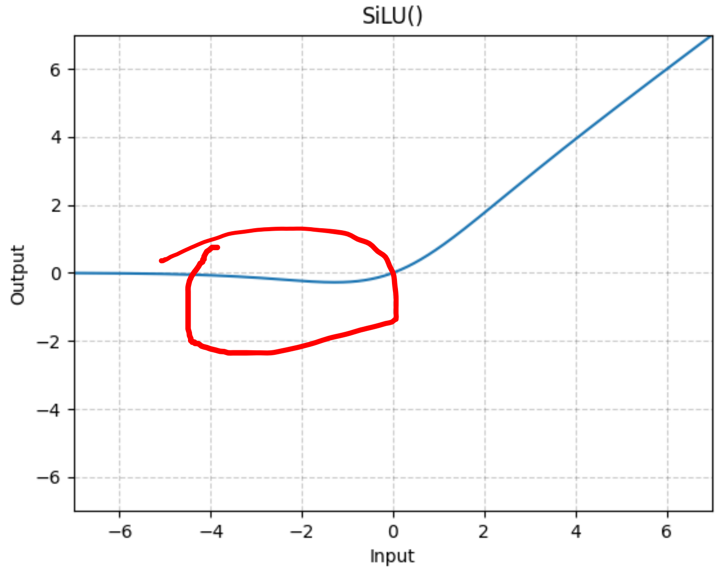
Pytorch 中的激活函数

import torch.nn as nn

torch.nn.SiLU()

$\mathrm{silu}(x) = x * \sigma(x)$, where $\sigma(x)$ is the logistic sigmoid.

```
m = nn.SiLU()
in = torch.randn(2)
output = m(in)
```

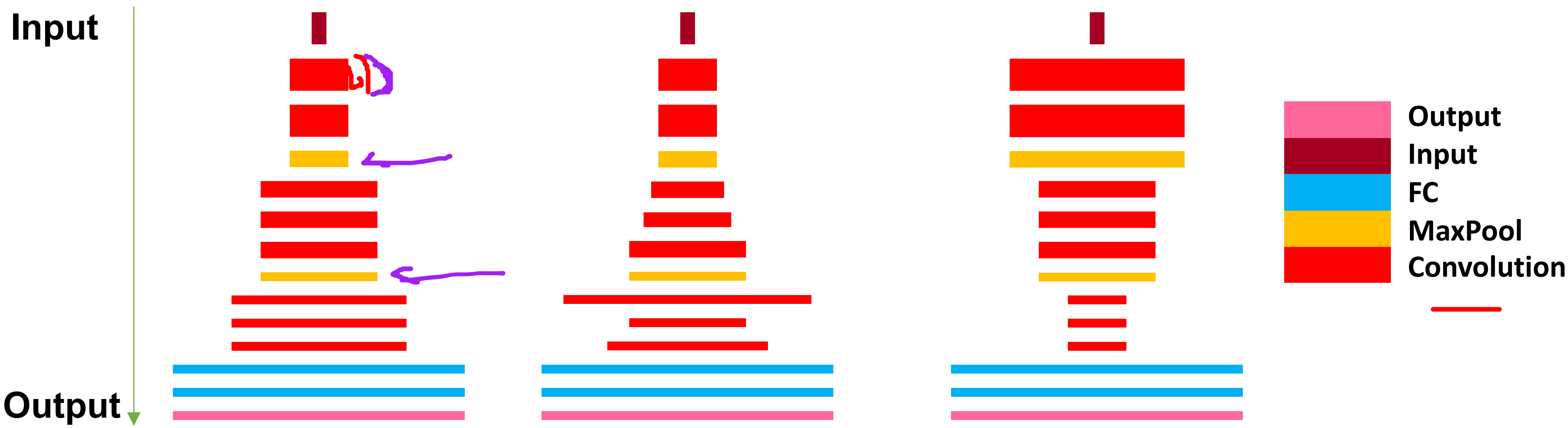


优化策略

- 激活函数
- 神经网络架构
- 损失函数 & 优化
- 权重初始化

神经网络设计

- 好的什么网络长什么样子
- “上窄下宽”



良好的神经网络设计。

底部的较宽架构可补偿因较小的特征图而导致的信息丢失。

不建议。

信息如何在过滤器中累积和保留不清楚。

不建议。

在最终输出层之前没有任何信息保留下来。

神经网络设计

如何选择卷积核的数目?

- 一般情况下，卷积核数目设置为2^n.
- 经过下采样层(e.g. MaxPool)后卷积核数目以2的倍数递增

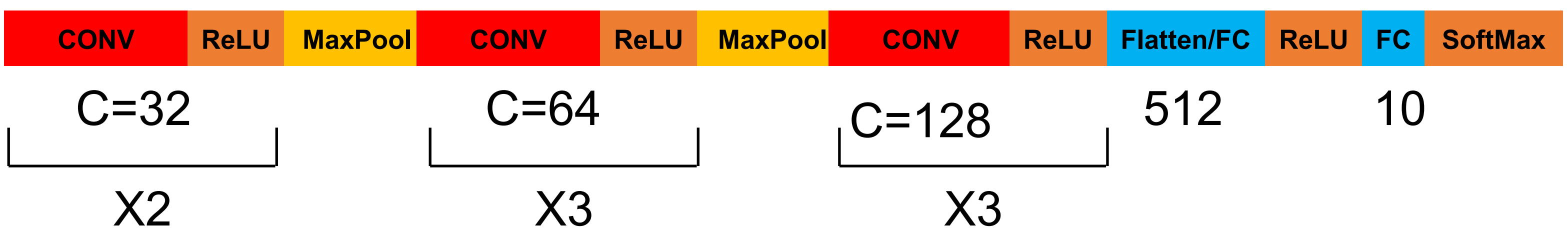
CONV	**ReLU**	**MaxPool**	**CONV**	**ReLU**	**MaxPool**	**CONV**	**ReLU**	**Flatten/FC**	**ReLU**	**FC**	**SoftMax**
C=32			C=64			C=128		512		10	

- 比如, 上面的配置在MNIST上的效果很好。

神经网络设计

每对下采样模块之间应该插入多少个卷积？

- 在靠后的下采样模块对之间应插入更多的卷积。
- 不要插入太多操作以形成非常深的卷积网络。
  - 会使训练难度增大。



神经网络设计

应该选择哪种激活函数?

- 最好一开始选择ReLU. ReLu可行的时候，尝试其他类型的激活函数以提高神经网络性能。
- 使用ReLU以外的激活函数会增加调试的难度 因为它们可以尝试在任何类型的神经网络上实现“收敛”。
  - 例如，即使神经网络配置不正确，ELU仍趋于收敛，但只能达到较差的性能.
- 最后一层选择 Softmax.

优化策略

- 激活函数
- 神经网络架构
- 损失函数 & 优化
- 权重初始化

损失函数

- 均方误差损失函数 (分类)

$$\mathrm{L} = \frac{1}{n}\sum (y - y)^2$$

torch.nn.MSELoss(*size_average*=*None*, *reduce*=*None*, *reduction* =*'mean'*)

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = (x_n - y_n)^2,$$

$$\ell(x, y) = \begin{cases} \operatorname{mean}(L), & \text{if reduction} = \text{'mean';} \\ \operatorname{sum}(L), & \text{if reduction} = \text{'sum'.} \end{cases}$$

损失函数

交叉熵(Softmax) 损失函数 (分类)

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

$$L_i = -\log P(Y = y_i | X = x_i) = -\log \frac{e^{s_{y_i}}}{\sum_j e^{s_{y_j}}}$$

假设我们有N 个类别。 如果权重**W**用均匀随机数初始化，则损失函数的值是多少?

损失函数

交叉熵(Softmax) 损失函数 (分类)

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

$$L_i = -\log P(Y = y_i | X = x_i) = -\log \frac{e^{s_{y_i}}}{\sum_j e^{s_{yj}}}$$

假设我们有N 个类别。 如果权重**W**用均匀随机数初始化，则损失函数的值是多少?

$\log N$. 因为当DNN一无所知时，将执行随机猜测。 所有s大约相等。

建议在开始训练之前检查初始损失值.

例如，对于N = 10类的CIFAR-10数据集，初始损失应为**2.3**左右。

损失函数

交叉熵(Softmax) 损失函数 (分类)

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

$$L_i = -\log P(Y = y_i | X = x_i) = -\log \frac{e^{s_{y_i}}}{\sum_j e^{s_{y_j}}}$$

torch.nn.CrossEntropyLoss(*weight=None, size_average=None, ignore_index=- 100, reduce=None, reduction='mean', label_smoothing=0.0*)

$$l_n = -w_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C} \exp(x_{n,c})} \cdot 1\{y_n \neq \text{ignore\_index}\}$$

```
loss = nn.CrossEntropyLoss()
input = torch.randn(3, 5, requires_grad=True)
target = torch.empty(3, dtype=torch.long).random_(5)
output = loss(input, target)
output.backward()
```

优化策略

- 激活函数
- 神经网络架构
- 损失函数 & 优化
- 权重初始化

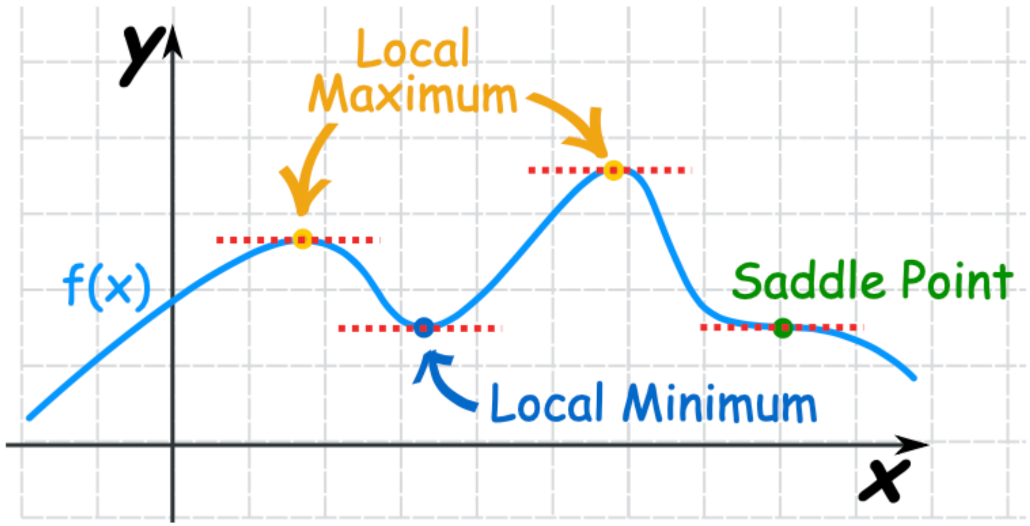
随机梯度下降（SGD）

权重通过计算的原始梯度进行更新。

$$W^{t+1} = W^t - \alpha \nabla_W L(W^t)$$

α: 学习率

SGD 可能陷入在局部最优点和马鞍点



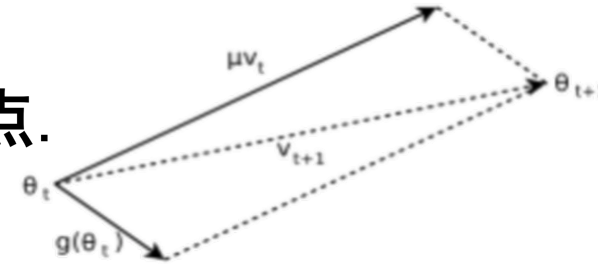
需要合适的初始化和step size的

带动量的SGD

计算和积累‘动量’ 来加速收敛过程

$$v^{t+1} = \mu v_t - \alpha \nabla_{W^t} L(W^t)$$
$$W^{t+1} = W^t + v^{t+1}$$

α: 学习率
μ: 动量因子



带动量的SGD 带有累积的动量来避免鞍点.

优点:

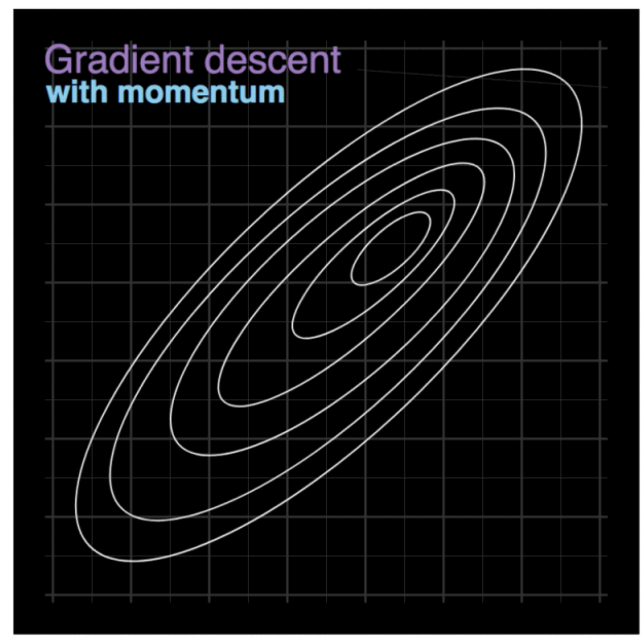
\+ 权重更新时候的计算开销小

\+ 经常用来取得最好的性能

缺点:

\- 需要更多的条件时间

\- 可能会越过全局最优.



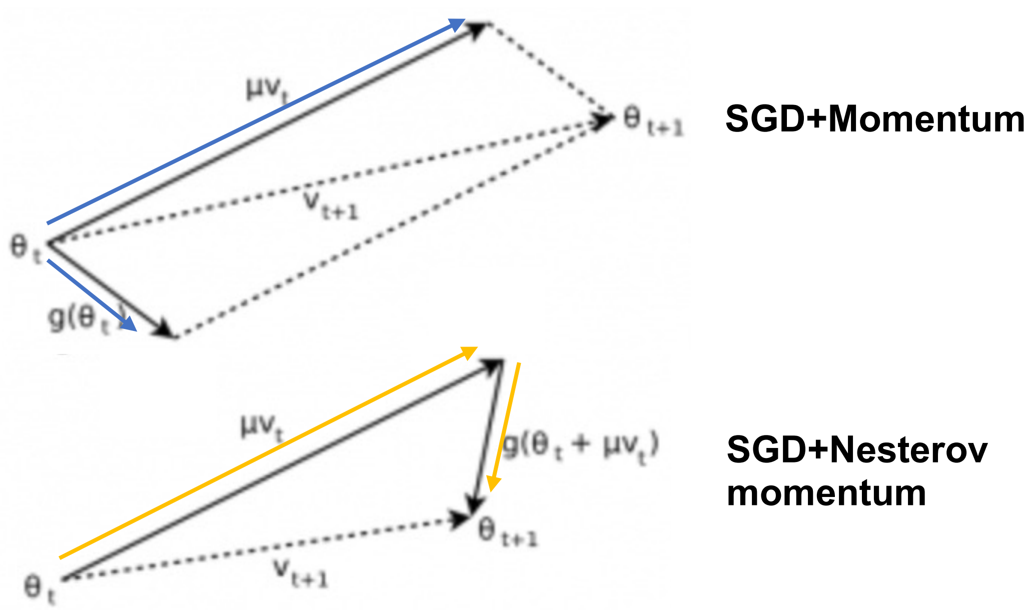
Nesterov动量的SGD

动量积累和 '预测': 当前权重参数乘以一个速率用于计算梯度。

$$v^{t+1} = \mu v_t - \alpha \nabla_W L(W^t + \mu v_t)$$
$$W^{t+1} = W^t + v^{t+1}$$

α: 学习率
μ: 动量因子

优点:
+ 避免优化过程的震荡
+ 非常好的优化性能



自适应Adagrad (Adaptive gradient)

为了加速学习过程, 自适应调整学习率调整权重参数

$$W^{t+1} = W^t - \frac{\alpha \nabla_W L(W^t)}{\sqrt{G+\epsilon}}$$

$$G = \sum_{i=0}^{t} \left(\nabla_W L(W^t)\right)^2$$

α: 学习率
ϵ : 一个很小的数值避免零除
G : 梯度的二范式

优点:

\+ 增加更新的平滑性

\+ 适合处理稀疏梯度

缺点:

\- 随着分母中的梯度累积，学习率会减小的很快， 使得训练提前结束

RMSprop (root mean square propagation)

为了解决Adagrad中的问题，RMSprop使用输入梯度的移动平均值作为分母。

$$G^{t+1} = \mu G^{t} + (1-\mu)\left(\nabla_W L(W^{t})\right)^{2}$$

$$W^{t+1} = W^{t} - \frac{\alpha \nabla_W L(W^{t})}{\sqrt{G^{t+1}+\epsilon}}$$

α: 学习率
μ: 动量因子
G^{t}: 在t步的梯度范式.
ϵ :一个很小的数值避免零除

优点:

+ 避免学习率的缩减
+ 对RNN网络很好

缺点:

- 在最后收敛时刻会震荡.

Adam (Adaptive Momentum Estimation)

带动量的SGD 和 RMSprop 的特点结合.

- $g_t = \nabla_W L(W^t)$
- $m_{t+1} = \beta_1 m_t + (1-\beta_1) g_t$ 一阶矩估计
- $v_{t+1} = \beta_2 v_t + (1-\beta_2) g_t^2$ 二阶矩估计
- $\hat{m}_{t+1} = \frac{m_{t+1}}{1-\beta_1^{t+1}}$ 偏差修正的一阶矩估计
- $\hat{v}_{t+1} = \frac{v_{t+1}}{1-\beta_2^{t+1}}$ 偏差修正的二阶矩估计
- $W^{t+1} = W^t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}$ 更新, ϵ一个很小的数值避免零(如 10E-8)

一阶和二阶动量估计使Adam快速收敛. 但 Adam 极其不稳定，会在收敛结束时loss会不断振荡。 与基于SGD的优化器相比，这可能会导致一些性能损失。

Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." (2015)

Adam (Adaptive Momentum Estimation)

原论文列举了将 Adam 优化算法应用在非凸优化问题中所获得的优势：

- 直截了当地实现
- 高效的计算
- 所需内存少
- 梯度对角缩放的不变性（第二部分将给予证明）
- 适合解决含大规模数据和参数的优化问题
- 适用于非稳态（non-stationary）目标
- 适用于解决包含很高噪声或稀疏梯度的问题
- 超参数可以很直观地解释，并且基本上只需极少量的调参

如何选择优化方法

- **Adam** 最快，所以大多情况下默认选择这个方法
- **带动量的SGD/Nesterov 动量SGD**性能表现会超出**Adam**, 但需要更多的时间来调节权重值达到收敛.

Pytorch 中的优化方法

```
import torch
import torch.optim as optim
optim.SGD(params,lr, momentum, dampening=0, weight_decay=0,
nesterov=False,…)
```

参数：

weight_decay: L2正则化

maximize: 布尔型，最大化参数

```
optimizer =
torch.optim.SGD(model.parameters(), lr=0.1,
momentum=0.9)
optimizer.zero_grad()
loss_fn(model(input), target).backward()
optimizer.step()
```

---

input : γ (lr), θ_0 (params), $f(\theta)$ (objective), λ (weight decay),
μ (momentum), τ (dampening), *nesterov*, *maximize*

---

for $t = 1$ **to** $\ldots$ **do**
- $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
- **if** $\lambda \neq 0$
  - $g_t \leftarrow g_t + \lambda\theta_{t-1}$
- **if** $\mu \neq 0$
  - **if** $t > 1$
    - $\mathbf{b}_t \leftarrow \mu\mathbf{b}_{t-1} + (1-\tau)g_t$
  - **else**
    - $\mathbf{b}_t \leftarrow g_t$
  - **if** *nesterov*
    - $g_t \leftarrow g_t + \mu\mathbf{b}_t$
  - **else**
    - $g_t \leftarrow \mathbf{b}_t$
- **if** *maximize*
  - $\theta_t \leftarrow \theta_{t-1} + \gamma g_t$
- **else**
  - $\theta_t \leftarrow \theta_{t-1} - \gamma g_t$

---

return θ_t

Pytorch 中的优化方法

```
import torch
import torch.optim as optim
optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08,
weight_decay=0, amsgrad=False, …)
```

参数:

weight_decay: L2正则化

maximize: 布尔型，最大化参数

```
optimizer =
torch.optim.SGD(model.parameters(), lr=0.1,
momentum=0.9)
optimizer.zero_grad()
loss_fn(model(input), target).backward()
optimizer.step()
```

---

input : γ (lr), β_1, β_2 (betas), θ_0 (params), $f(\theta)$ (objective)
λ (weight decay), *amsgrad*, *maximize*

initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$

---

for $t = 1$ **to** $\ldots$ **do**
  if *maximize* :
    $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
  else
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
  $\widehat{m_t} \leftarrow m_t / (1-\beta_1^t)$
  $\widehat{v_t} \leftarrow v_t / (1-\beta_2^t)$
  if *amsgrad*
    $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$
    $\theta_t \leftarrow \theta_{t-1} - \gamma\widehat{m_t} / (\sqrt{\widehat{v_t}^{max}} + \epsilon)$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma\widehat{m_t} / (\sqrt{\widehat{v_t}} + \epsilon)$

---

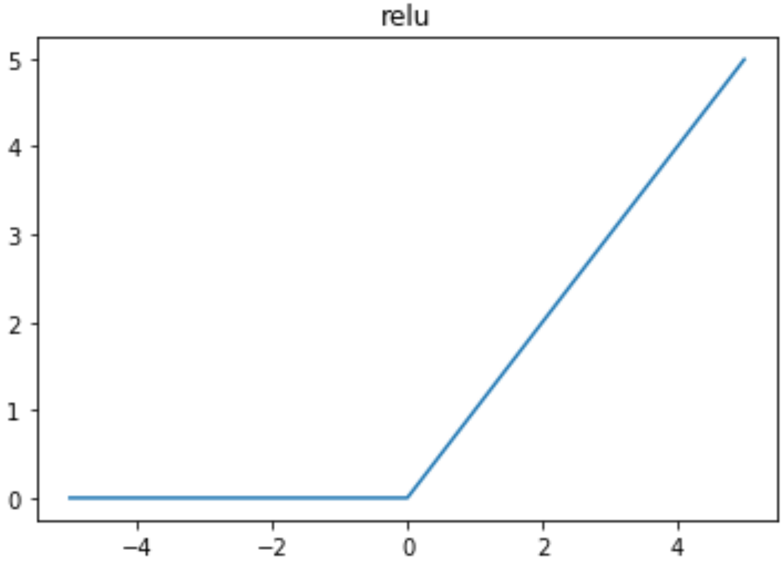
return θ_t

优化策略

- 激活函数
- 神经网络架构
- 损失函数 & 优化
- 权重初始化

权重初始化

- 没有权重初始化，DNN可能很难训练。
  - 假设我们使用ReLU作为所有DNN层的激活函数。
  - 问：如果在训练开始时将所有权重参数和偏差参数都设置为0，将会发生什么？

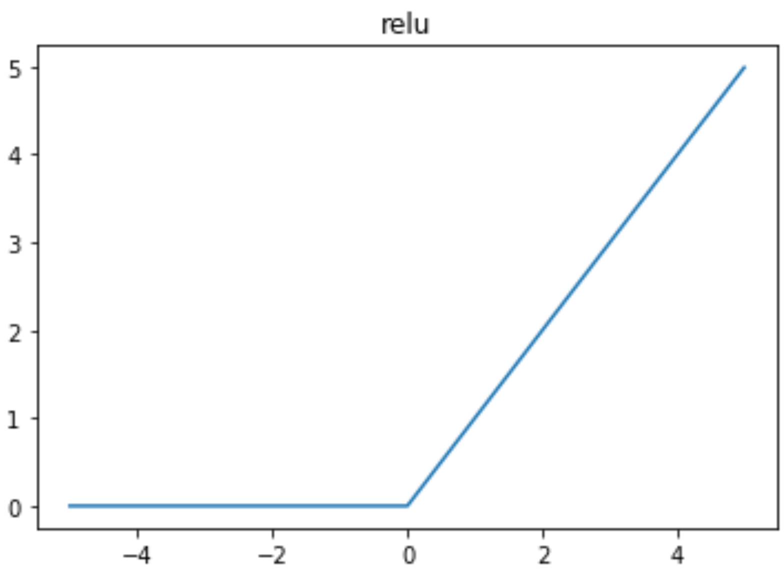


答:
所有输出都固定为0，所有神经元永远不会更新和激活（Dead Neuron).
尽管输入了训练数据，权重将永远不会更新.

权重初始化能激活一些神经元以生成一些非零输出并开始权重更新过程。

权重初始化

- 假设我们使用ReLU作为所有DNN层的激活函数。如果在训练开始时将所有权重参数和偏差参数设置**为相同的符号**，将会发生什么？



由于我们使用ReLU激活，因此负值神经元将不会被激活。它们将始终产生零。
这样，我们无法充分利用DNN的表示能力。

随机初始化

- 训练开始时随机初始化. 通常，平均值为0，标准差偏小的随机分布(e.g. 0.03). 两种常用的随机初始化函数:
  - 随机均匀初始化Random uniform initializer
  - 随机标准初始化Random normal initializer
- **随机初始化**不利用网络结构信息。
- 因此，除非仔细调整，否则此方法对提高网络性能没有用。一般情况无法通过这个方法提高网络性能。

Xavier 初始化

为了利用网络结构信息，根据不同的输入单元和输出单元初始化权重。

两种Xavier初始化函数.

- Xavier (Glorot) 标准初始化

$$W \sim N\left(0, \sqrt{\frac{2}{fan\_in + fan\_out}}\right)$$

- Xavier (Glorot) 均匀初始化

$$W \sim U\left[-\sqrt{\frac{6}{fan\_in + fan\_out}}, \sqrt{\frac{6}{fan\_in + fan\_out}}\right]$$

Xavier 初始化都是假设用线性或者拟线性的激活函数.
因此，Xavier初始化在数学上是有意义的。
但是，此方法不适用于具有级联结构的较新模型。

Xavier 初始化: 均匀 vs.标准

- 在大多数情况下，初始DNN权重的分布在训练后仍然保持。因此，决定我们要使用均匀权重初始化程序还是标准权重初始化很重要。

- 我们建议使用**标准权重初始化**，因为它符合自然规律。在PyTorch中，默认的权重初始化方法是xavier_uniform。
- 如果将初始化方法更改为xavier_normal，则大多数神经网络都将获得性能提升。

torch.nn.init.xavier_uniform_(tensor, gain=1)

torch.nn.init.xavier_normal_(tensor, gain=1)

MSRA 初始化 (方差缩放)

仅按输入单位数缩放当前权重的方差。

这种初始化不限制输出单元的大小，因此对有级联层和累加层的网络更友好。

对全连接层有 *fan_in* 个输入单元, MSRA 初始化权重的公式如下：

$$W \sim N\left(0, \sqrt{\frac{1}{fan\_in}}\right)$$

对卷积层，卷积核大小 $K \times K$ 输入通道数是C, MSRA初始化权重公式如下：

$$W \sim N\left(0, \sqrt{\frac{1}{K \times K \times C}}\right)$$

He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." (2015)

建议

- 最开始用 Xavier标准初始化
- 除非原文章作者指明, 不要用任何形式的均匀初始化, 因为这会导致性能下降.
- 有了Batch Normalization批处理规范化之后，初始化方法的重要程度降低了.

这节课，我们学了

- DNN 训练设置
  - 激活函数
  - 神经网络设计
- 优化方法
  - 基于SGD的优化方法
  - Adagrad, RMSprop, Adam
  - 如何选择优化方法
- 权重初始化
  - Xavier 均匀/标准初始化
  - MSRA方法(方差缩放)
  - 如何选择权重初始化方法