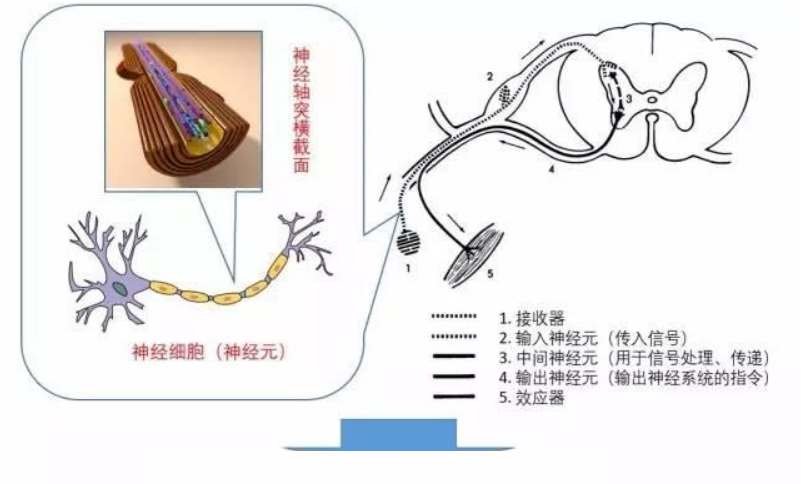
挖掘建模之分类与预测

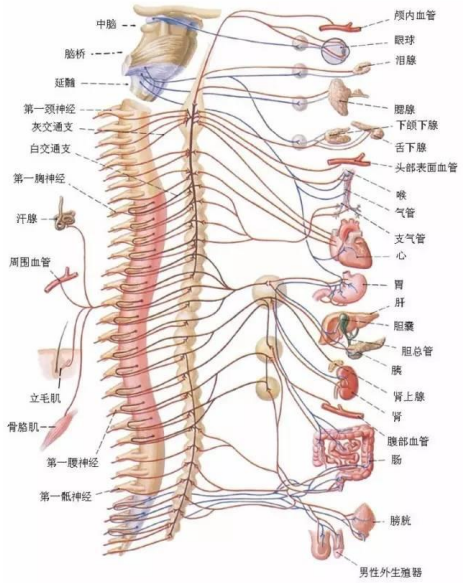
2021/11/15

目录

大脑及神经系统

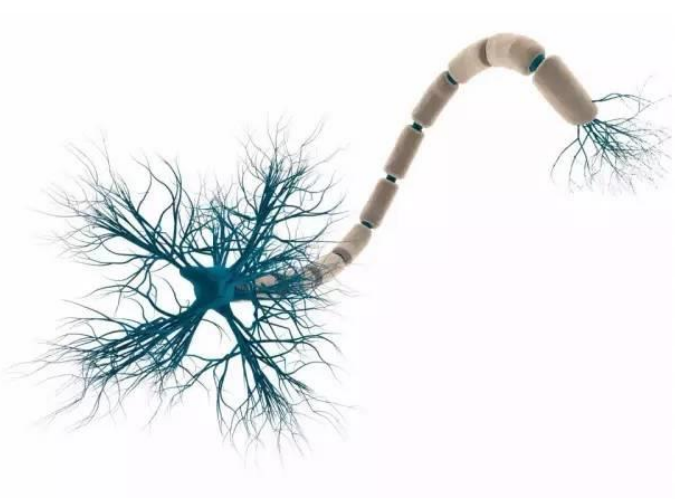
大脑



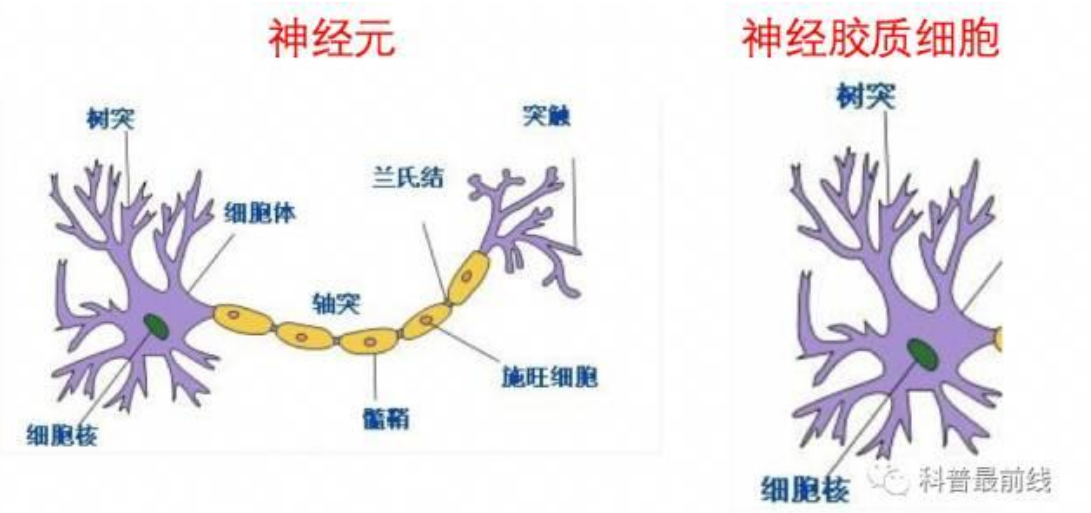


自主神经系概观

神经元



神经元



神经元具体结构

树突：有多个，主要用来接受传入信息

细胞核：对信号进行计算

轴突：只有一条，用于信号传递

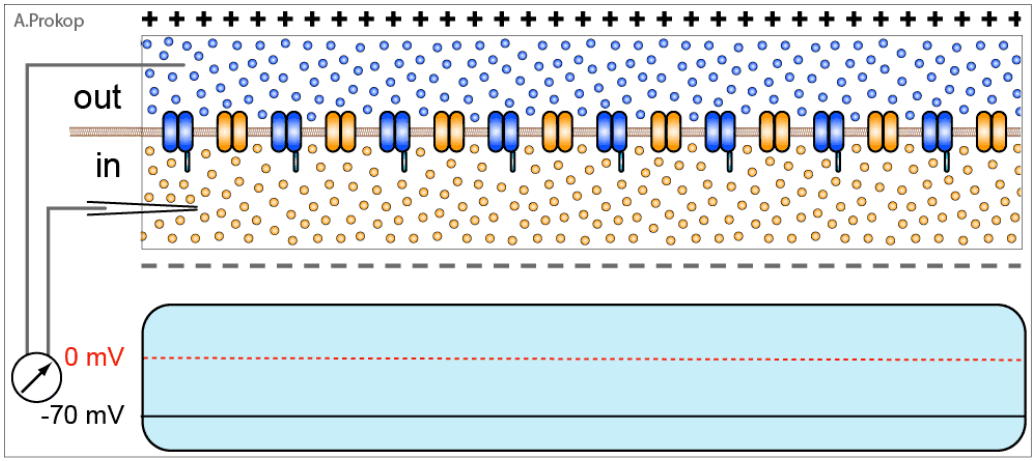
突触：轴突末梢跟其他神经元的树突产生连接，从而传递信号。这个连接的位置在生物学上叫做“突触”。

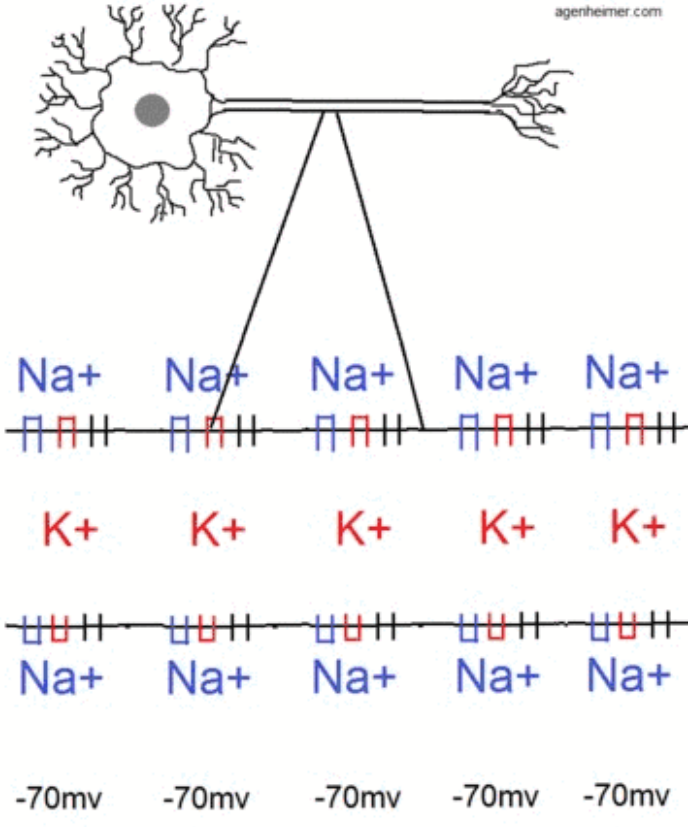
一个神经元接入了多个输入，经过计算最终只变成一个输出，传递给了后面的神经元

神经脉冲-细胞膜内外离子（钾和钠）的交替出入

神经信号：电信号，也即神经脉冲（如肌肉收缩），化学信号

神经脉冲：当我们受到刺激时，受体会发送神经脉冲，神经脉冲会经由神经元传到脊髓之后去到大脑，经过大脑分析后会发送一些神经脉冲到我们的肌肉让我们能作出反应。





神经脉冲的产生，依靠神经元内外电势差

神经科学-学科交叉的巅峰

Hodgkin

Huxley

英国生理学家Hodgkin和Huxley在1952年研究鱿鱼的大轴突后，提出了Hodgkin-Huxley模型以描述神经脉冲的产生和传到，并因此获得了1963年的诺贝尔医学或生理学奖。

产生的电流脉冲

相当于电量=电容*电压 $Q = C \times V$

附加电流，表示神经元受到的刺激

$$I = C_m \frac{dV_m}{dt} + \bar{g}_K n^4 (V_m - V_K) + \bar{g}_{Na} m^3 h (V_m - V_{Na}) + \bar{g}_l (V_m - V_l) - I_{App}$$

钾离子诱导的脉冲（非线性）

钠离子诱导的脉冲（非线性）

两种离子共同诱导的脉冲（线性）

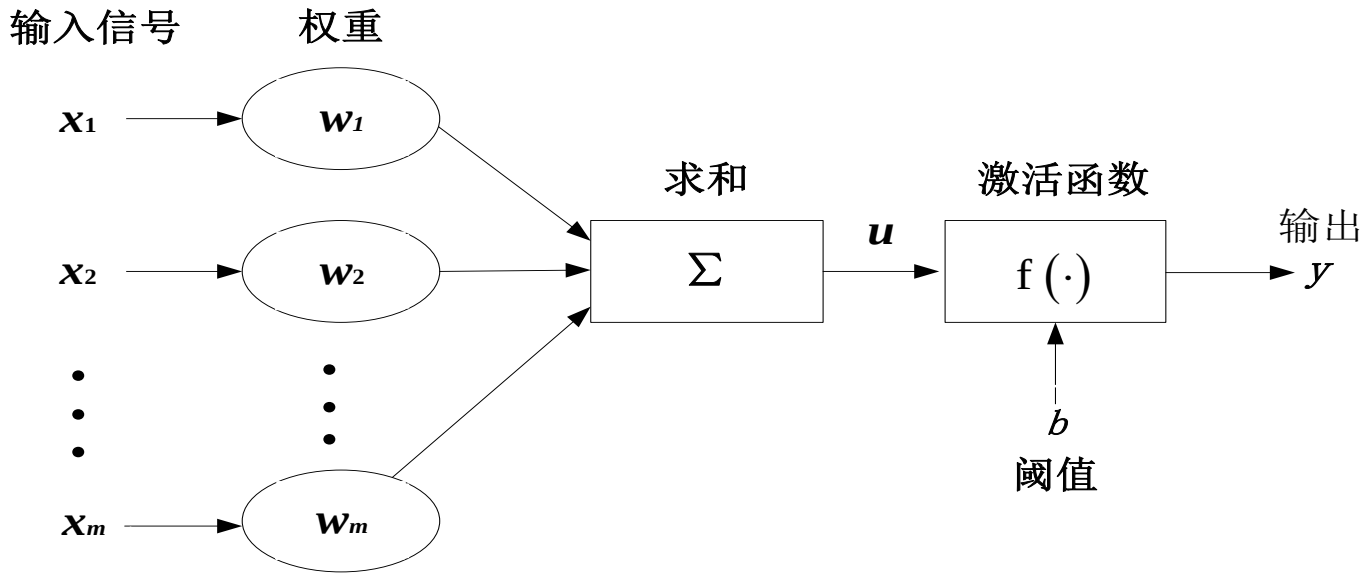
$$\frac{dn}{dt} = \alpha_n (V_m)(1 - n) - \beta_n (V_m) n$$

$$\frac{dm}{dt} = \alpha_m (V_m)(1 - m) - \beta_m (V_m) m$$

$$\frac{dh}{dt} = \alpha_h (V_m)(1 - h) - \beta_h (V_m) h$$

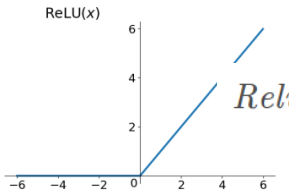
- m, n和h的不同表示钠通道和钾通道有不同结构
- 它们三个变量介于0和1之间，分别表示几个不同通道大门的开关概率
- α和β表示大门开关两状态之间的转化率

人工神经元



人工神经网络-典型结构

激活函数，引入非线性



$Relu = max(0, x)$

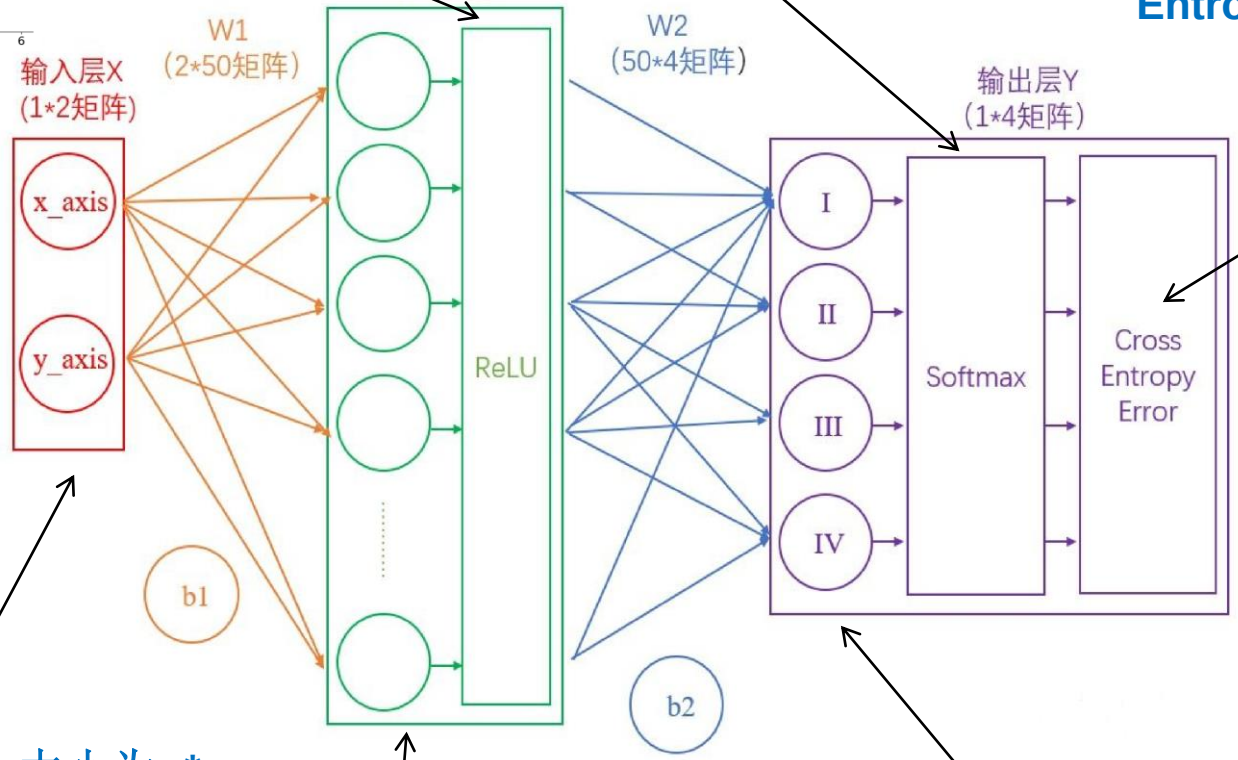
将输出转换为属于每个类的概率，**softmax**

$p=\frac{e^{z_c}}{\sum_c e^{z_c}}$

损失函数:计算预测值与类标的误差

交叉熵损失（**Cross Entropy Error**）:

$$L_i = -\sum_{c=1}^{k} y_{ic} \log p_{ic}$$



输入数据，大小为**n*m**的矩阵，其中**n**为样本个数，**m**为特征个数

隐含层 $H=X * W_1 + b_1$

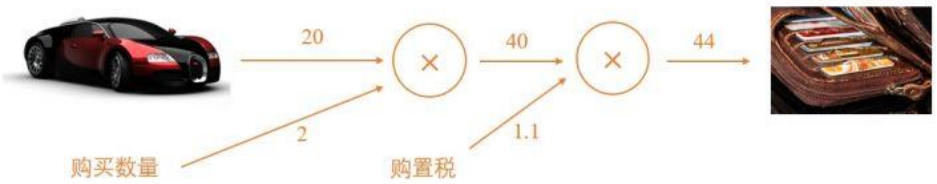
输出层 $Z=\text{Relu}(H) * W_2 + b_2$

一个典型的三层神经网络

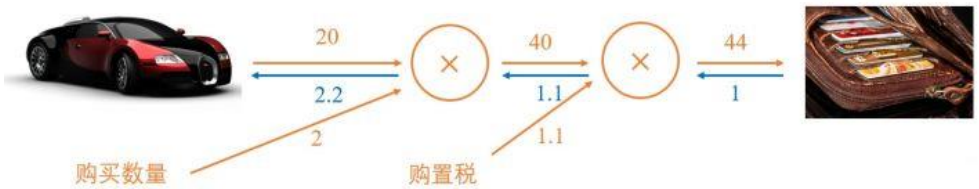
人工神经网络-优化

对于神经网络的优化算法，主要需要两步：

- **前向传播**(Forward Propagation)：从输入层到输出层，计算每一层每一个神经元的输出，最终计算损失函数
- **反向传播**(Back Propagation)：根据前向传播计算出来的激活值，来计算每一层参数的梯度，并从后往前进行参数的更新。



正向传播



反向传播

链式法则：如果某个函数由复合函数表示，则该复合函数的导数可以用构成复合函数的各个函数的导数的乘积表示。

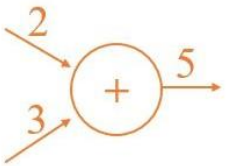
$$w_i^2 = w_i^2 - \gamma \frac{\partial L_i}{\partial w_i^2} \qquad \frac{\partial L_i}{\partial w_i^2} = \frac{\partial L_i}{\partial p_i}\frac{\partial p_i}{\partial z_i}\frac{\partial z_i}{\partial w_i^2}$$

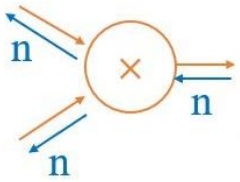
$$w_i^1 = w_i^1 - \gamma \frac{\partial L_i}{\partial w_i^1} \qquad \frac{\partial L_i}{\partial w_i^1} = \frac{\partial L_i}{\partial p_i}\frac{\partial p_i}{\partial z_i}\frac{\partial z_i}{\partial h_i}\frac{\partial h_i}{\partial w_i^1}$$

人工神经网络

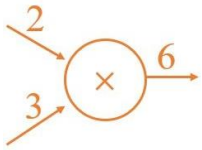
- 反向传播
- 激活函数
- 学习率
- 优化方法
- 过拟合

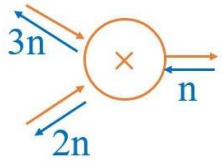
人工神经网络-反向传播1





z对x求导等于1，对y求导也等于1，所以在加法节点反向传递时，输入的值会**原封不动**地流入下一个节点。

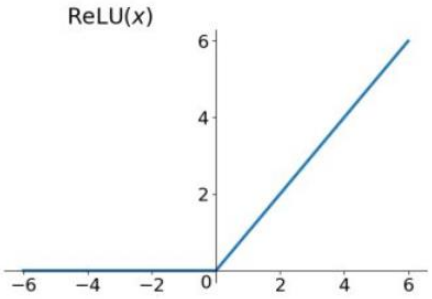


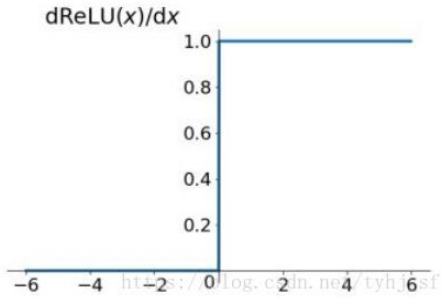


z对x求导等于1，对y求导也等于1，所以在乘法节点反向传递时，输入的值会**原封不动**地流入下一个节点。

$H=X*W_1+b_1$

对X求导，结果就是W1；对W1求导，结果就是X，和乘法节点是一样的；对b1求导，结果为1，原封不动地流入即可。





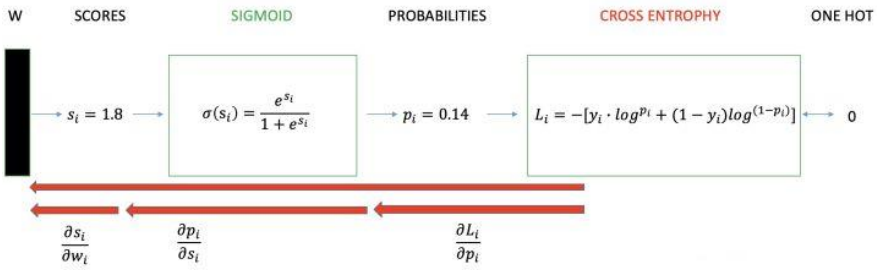
当x>0时，y=x，求导为1，也就是原封不动传递。
当x<=0时，y=0，求导为0，也就是传递值为0。

$$W = W - epsilon * dW$$

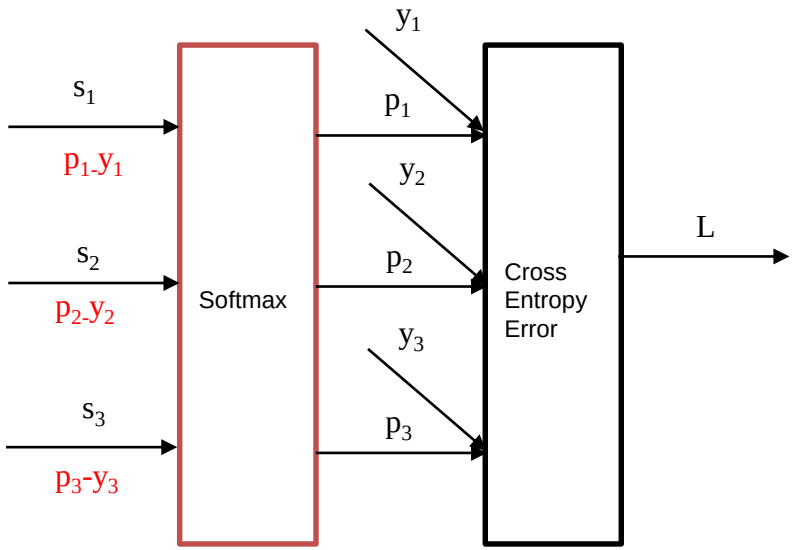
$$b = b - epsilon * db$$

Epsilon是学习率

人工神经网络-反向传播2



Softmax-with-Loss



Softmax-with-Loss反向传播

$$L_i = -[y_i \cdot log(p_i) + (1-y_i) \cdot log(1-p_i)]$$

$$\begin{aligned}\frac{\partial L_i}{\partial p_i} &= \frac{\partial -[y_i \cdot log(p_i) + (1-y_i) \cdot log(1-p_i)]}{\partial p_i} \\ &= -\frac{y_i}{p_i} - [(1-y_i) \cdot \frac{1}{1-p_i} \cdot (-1)] \\ &= -\frac{y_i}{p_i} + \frac{1-y_i}{1-p_i}\end{aligned}$$

$$\frac{\partial s_i}{\partial w_i} = x_i$$

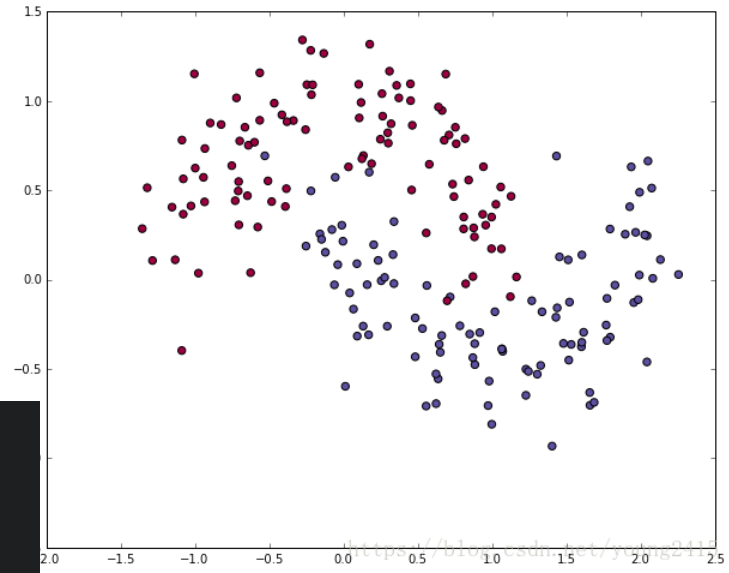
$$\begin{aligned}\frac{\partial p_i}{\partial s_i} &= \frac{(e^{s_i})' \cdot (1+e^{s_i}) - e^{s_i} \cdot (1+e^{s_i})'}{(1+e^{s_i})^2} \\ &= \frac{e^{s_i} \cdot (1+e^{s_i}) - e^{s_i} \cdot e^{s_i}}{(1+e^{s_i})^2} \\ &= \frac{e^{s_i}}{(1+e^{s_i})^2} \\ &= \frac{e^{s_i}}{1+e^{s_i}} \cdot \frac{1}{1+e^{s_i}} \\ &= \sigma(s_i) \cdot [1-\sigma(s_i)]\end{aligned}$$

$$\begin{aligned}\frac{\partial L_i}{\partial w_i} &= \frac{\partial L_i}{\partial p_i} \cdot \frac{\partial p_i}{\partial s_i} \cdot \frac{\partial s_i}{\partial w_i} \\ &= [-\frac{y_i}{p_i} + \frac{1-y_i}{1-p_i}] \cdot \sigma(s_i) \cdot [1-\sigma(s_i)] \cdot x_i \\ &= [-\frac{y_i}{\sigma(s_i)} + \frac{1-y_i}{1-\sigma(s_i)}] \cdot \sigma(s_i) \cdot [1-\sigma(s_i)] \cdot x_i \\ &= [-\frac{y_i}{\sigma(s_i)} \cdot \sigma(s_i) \cdot (1-\sigma(s_i)) + \frac{1-y_i}{1-\sigma(s_i)} \cdot \sigma(s_i) \cdot (1-\sigma(s_i))] \cdot x_i \\ &= [-y_i + y_i \cdot \sigma(s_i) + \sigma(s_i) - y_i \cdot \sigma(s_i)] \cdot x_i \\ &= [\sigma(s_i) - y_i] \cdot x_i\end{aligned}$$

Softmax-with-Loss的反向传播的结果为(p1 − y1, p2 − y2, p3 − y3)。

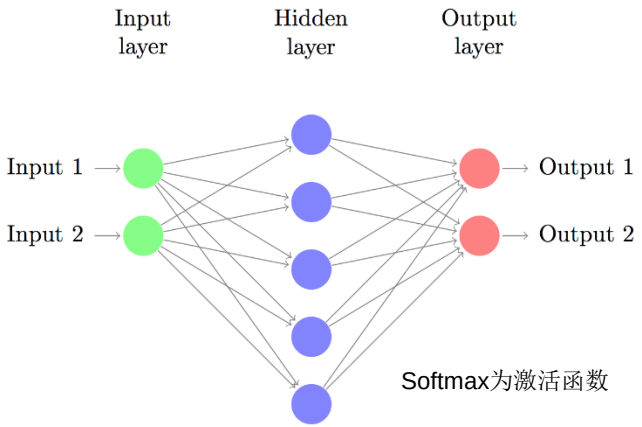
人工神经网络-代码1

```
# 生成数据集并绘制出来
np.random.seed(0)
X, y = sklearn.datasets.make_moons(200, noise=0.20)
plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=plt.cm.Spectral)
```




```
#参数设置
num_examples = len(X) # 训练样本的数量
nn_input_dim = 2 # 输入层的维度
nn_output_dim = 2 # 输出层的维度

# 梯度下降的参数
epsilon = 0.01 # 梯度下降的学习率
reg_lambda = 0.01 # 正则化的强度
```

```
# 预测输出（0或1）
def predict(model, x):
    W1, b1, W2, b2 = model['W1'], model['b1'], model['W2'], model['b2']
    # 正向传播
    z1 = x.dot(W1) + b1
    a1 = np.tanh(z1)
    z2 = a1.dot(W2) + b2
    exp_scores = np.exp(z2)
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
    return np.argmax(probs, axis=1)
```



Softmax为激活函数

```
#帮助我们在数据集上估算总体损失的函数
def calculate_loss(model):
    W1, b1, W2, b2 = model['W1'], model['b1'], model['W2'], model['b2']
    #正向传播，计算预测值
    z1 = X.dot(W1) + b1
    a1 = np.tanh(z1)
    z2 = a1.dot(W2) + b2
    exp_scores = np.exp(z2)
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
    # 计算损失
    corect_logprobs = -np.log(probs[range(num_examples), y])
    data_loss = np.sum(corect_logprobs)
    #在损失上加上正则项（可选）
    data_loss += reg_lambda/2 * (np.sum(np.square(W1)) + np.sum(np.square(W2)))
    return 1./num_examples * data_loss
```

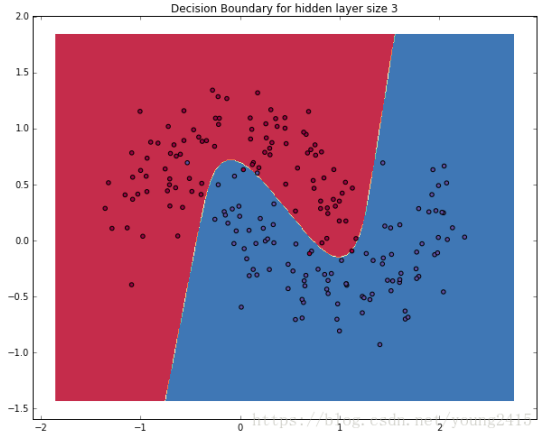
人工神经网络-代码2

```
# 这个函数为神经网络学习参数并且返回模型
# - nn_hdim: 隐藏层的节点数
# - num_passes: 通过训练集进行梯度下降的次数
# - print_loss: 如果是True, 那么每1000次迭代就打印一次损失值
def build_model(nn_hdim, num_passes=20000, print_loss=False):

    # 用随机值初始化参数。
    np.random.seed(0)
    W1 = np.random.randn(nn_input_dim, nn_hdim) / np.sqrt(nn_input_dim)
    b1 = np.zeros((1, nn_hdim))
    W2 = np.random.randn(nn_hdim, nn_output_dim) / np.sqrt(nn_hdim)
    b2 = np.zeros((1, nn_output_dim))

    # 梯度下降
    for i in xrange(0, num_passes):

        # 正向传播
        z1 = X.dot(W1) + b1
        a1 = np.tanh(z1)
        z2 = a1.dot(W2) + b2
        exp_scores = np.exp(z2)
        probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
```

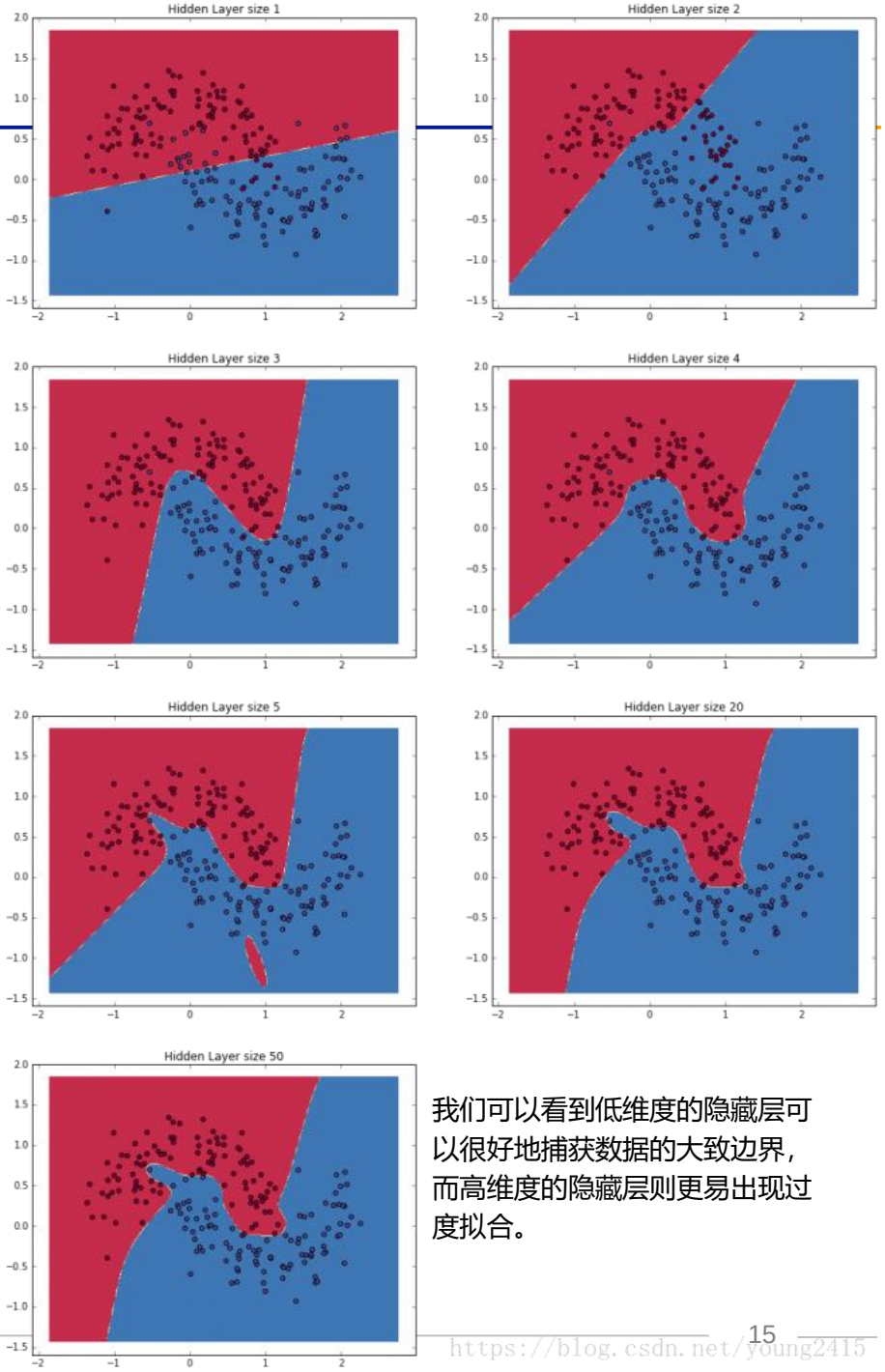
```
        # 反向传播
        delta3 = probs
        delta3[range(num_examples), y] -= 1
        dW2 = (a1.T).dot(delta3)
        db2 = np.sum(delta3, axis=0, keepdims=True)
        delta2 = delta3.dot(W2.T) * (1 - np.power(a1, 2))
        dW1 = np.dot(X.T, delta2)
        db1 = np.sum(delta2, axis=0)

        # 添加正则项 (b1 和 b2 没有正则项)
        dW2 += reg_lambda * W2
        dW1 += reg_lambda * W1

        # 梯度下降更新参数
        W1 += -epsilon * dW1
        b1 += -epsilon * db1
        W2 += -epsilon * dW2
        b2 += -epsilon * db2

        # 为模型分配新的参数
        model = { 'W1': W1, 'b1': b1, 'W2': W2, 'b2': b2}

        # 选择性地打印损失
        if print_loss and i % 1000 == 0:
          print "Loss after iteration %i: %f" %(i,
calculate_loss(model))

    return model
```

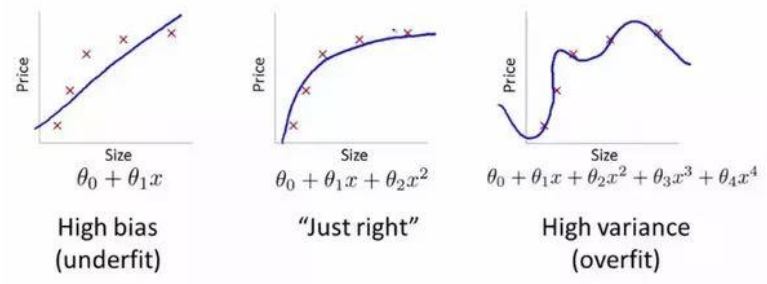
人工神经网络-代码3

```
# 搭建一个3维隐藏层的模型
model = build_model(3, print_loss=True)

# 画出决策边界
plot_decision_boundary(lambda x: predict(model, x))
plt.title("Decision Boundary for hidden layer size 3")
```




```
plt.figure(figsize=(16, 32))
hidden_layer_dimensions = [1, 2, 3, 4, 5, 20, 50]
for i, nn_hdim in enumerate(hidden_layer_dimensions):
    plt.subplot(5, 2, i+1)
    plt.title('Hidden Layer size %d' % nn_hdim)
    model = build_model(nn_hdim)
    plot_decision_boundary(lambda x: predict(model, x))
plt.show()
```



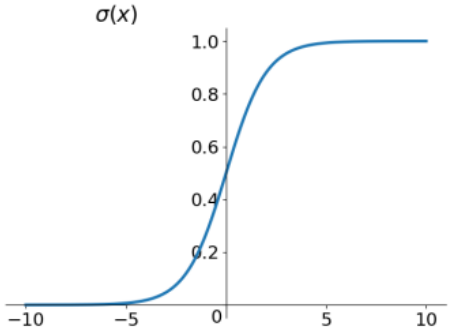


我们可以看到低维度的隐藏层可以很好地捕获数据的大致边界，而高维度的隐藏层则更易出现过度拟合。

人工神经网络-Sigmoid激活函数

如果不用激励函数，在这种情况下你每一层节点的输入都是上层输出的线性函数，很容易验证，无论你神经网络有多少层，输出都是输入的线性组合，与没有隐藏层效果相当，这种情况就是最原始的感知机（Perceptron）了，那么网络的逼近能力就相当有限，不能处理非线性数据。

激活函数的目的是引入非线性



Sigmoid函数

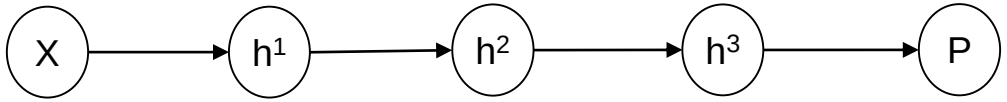
$$f(z) = \frac{1}{1 + e^{-z}}$$

- 缺点1：Sigmoid 的 output 不是0均值（即zero-centered），会导致后一层的神经元将得到上一层输出的非0均值的信号作为输入。如x>0, $f=w^Tx+b$,那么对w求局部梯度则都为正，这样在反向传播的过程中w都往正方向更新，导致有一种捆绑的效果，使得收敛缓慢。
- 缺点2：其解析式中含有幂运算，计算机求解时相对来讲比较耗时。对于规模比较大的深度网络，这会较大地增加训练时间。
- 缺点3：在深度神经网络中梯度反向传递时导致梯度爆炸和梯度消失，其中梯度爆炸发生的概率非常小，而梯度消失发生的概率比较大。

人工神经网络-梯度消失

如下是一个极简单的五层神经网络：w是权重，h是隐含层输出，L是某个代价函数， σ 是某个激活函数。

第j个神经元的输出 $h_j = \sigma(z_j) = \sigma(w_j * h_{j-1})$



$$\frac{\partial L_i}{\partial w_1} = \frac{\partial L_i}{\partial p_i}\frac{\partial p_i}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial w_1}$$

$$\frac{\partial h_3}{\partial h_2} = w_3\sigma'(z_3)$$

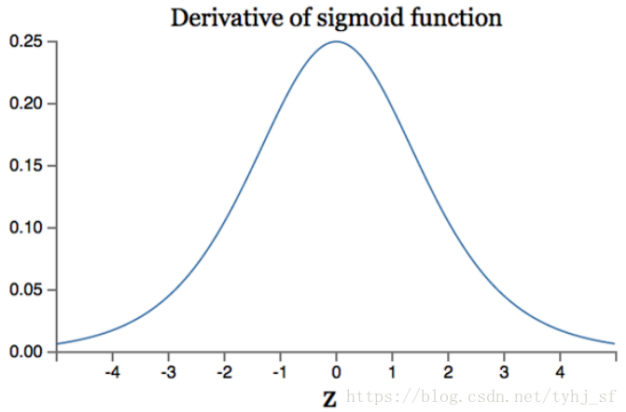
$$\frac{\partial h_2}{\partial h_1} = w_2\sigma'(z_2)$$

由于一般 $|w_j| < 1$ $\sigma'(z_j) < 1/4$，我们有 $|w_j\sigma'(z_j)| < 1/4$

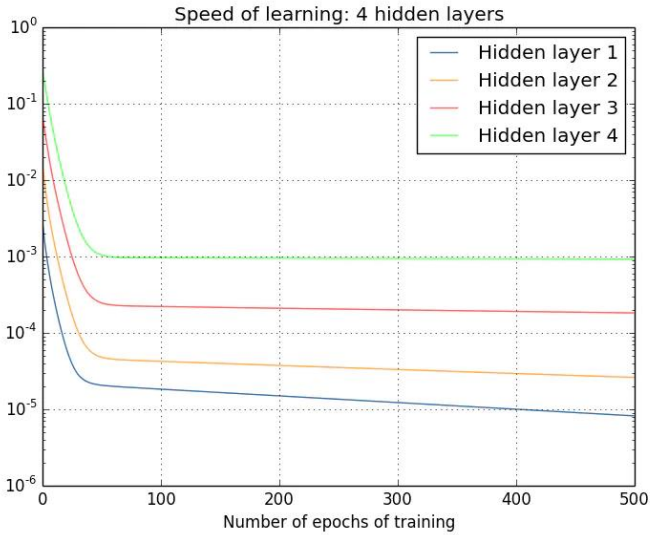
$$\frac{\partial L_i}{\partial w_1} = \frac{\partial L_i}{\partial p_i}\frac{\partial p_i}{\partial h_3}\left(\frac{1}{4}\right)^2\frac{\partial h_1}{\partial w_1}$$

如何解决梯度消失问题?

修改激活函数，改变其梯度。



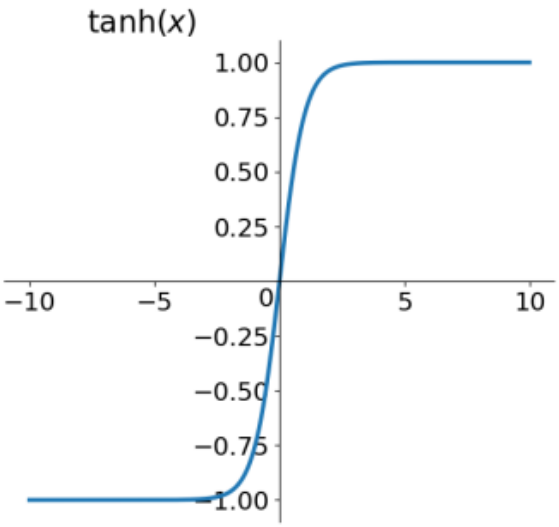
Sigmoid函数求导

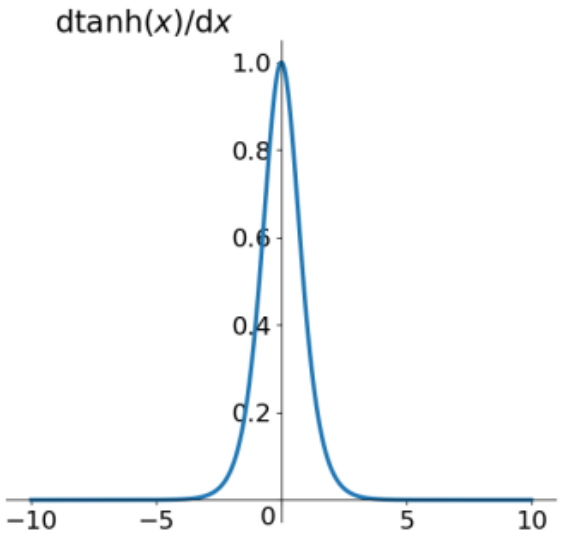


学习速度与隐含层关系

人工神经网络-Tanh激活函数

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



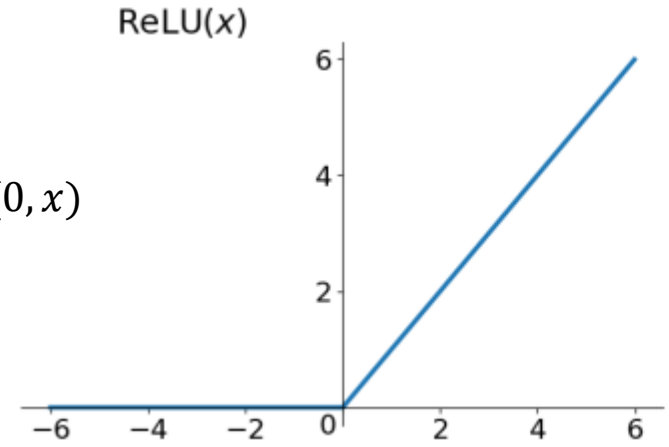


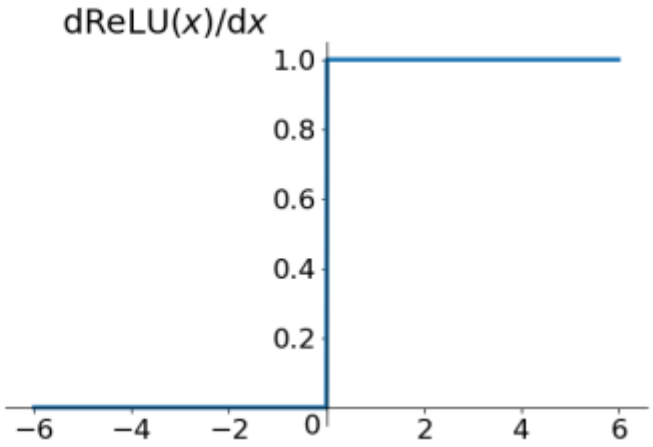
Tanh（Hyperbolic Tangent）函数

它解决了Sigmoid函数的不是zero-centered输出问题，然而，梯度消失的问题和幂运算的问题仍然存在。

人工神经网络-Relu激活函数

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





ReLU优点：

- 解决了gradient vanishing问题 (在正区间)
- 计算速度非常快，只需要判断输入是否大于0
- 收敛速度远快于sigmoid和tanh

ReLU也有几个需要特别注意的问题：

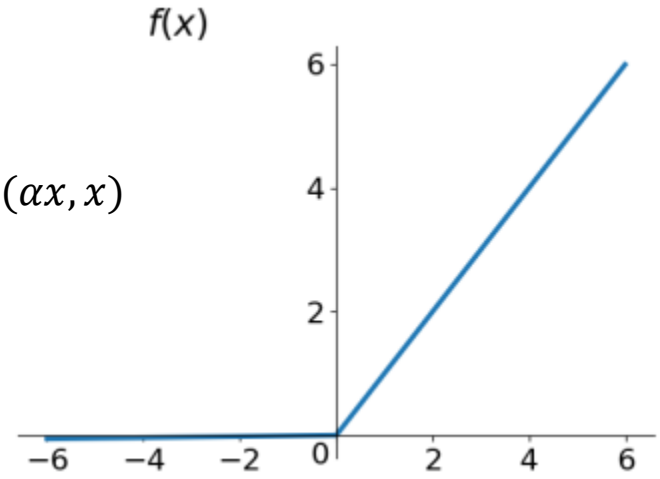
1）ReLU的输出不是zero-centered

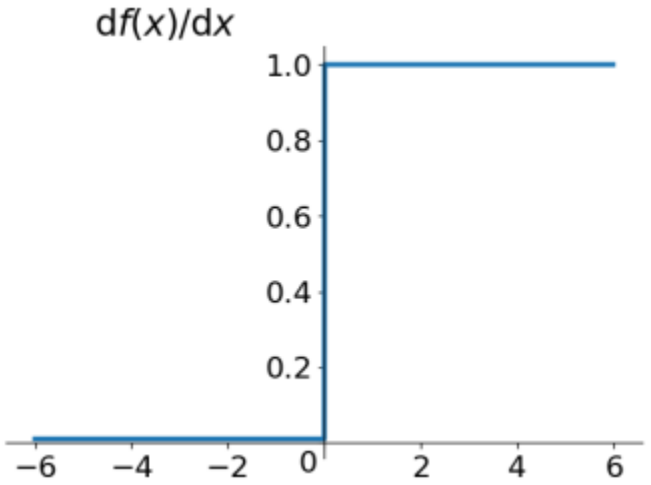
2）Dead ReLU Problem，指的是某些神经元可能永远不会被激活，导致相应的参数永远不能被更新。有两个主要原因可能导致这种情况产生: (1) 非常不幸的参数初始化，这种情况比较少见 (2) learning rate太高导致在训练过程中参数更新太大，不幸使网络进入这种状态。解决方法是可以采用Xavier初始化方法，以及避免将learning rate设置太大或使用adagrad等自动调节learning rate的算法。

ReLU是最常用的activation function，在搭建人工神经网络的时候推荐优先尝试！

人工神经网络-Leaky Relu激活函数

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

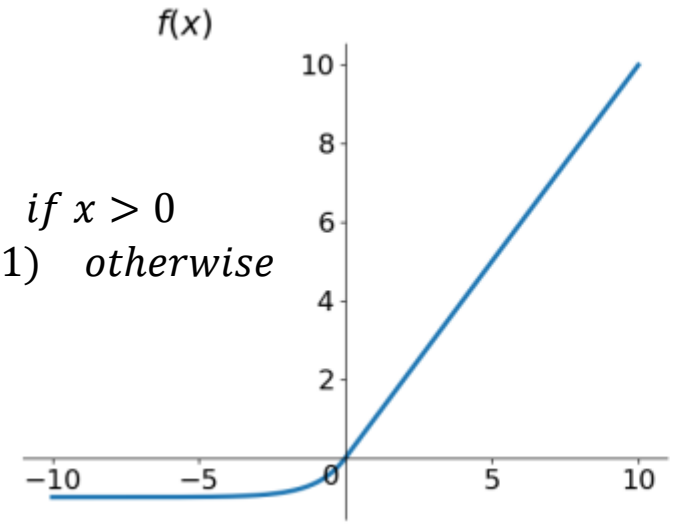


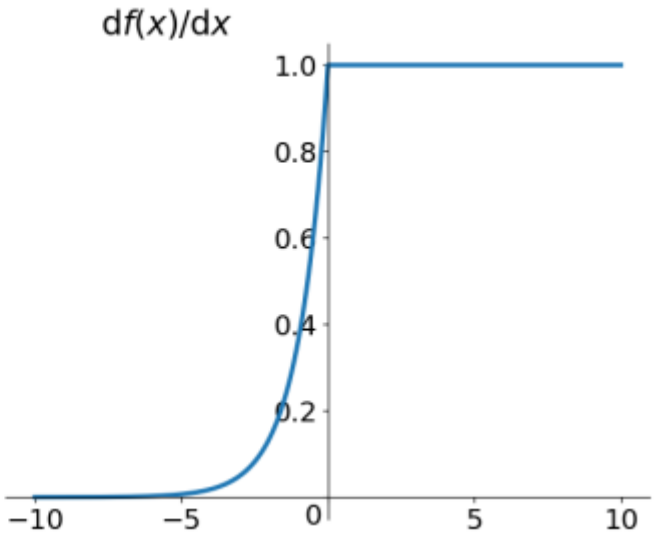


理论上来讲，Leaky ReLU有ReLU的所有优点，外加不会有Dead ReLU问题，但是在实际操作当中，并没有完全证明Leaky ReLU总是好于ReLU。

人工神经网络-ELU (Exponential Linear Units)激活函数

$$f(x) = \begin{cases} x & if\ x > 0 \\ \alpha(e^x - 1) & otherwise \end{cases}$$





ELU优点：

- 不会有Dead ReLU问题
- 输出的均值接近0，zero-centered

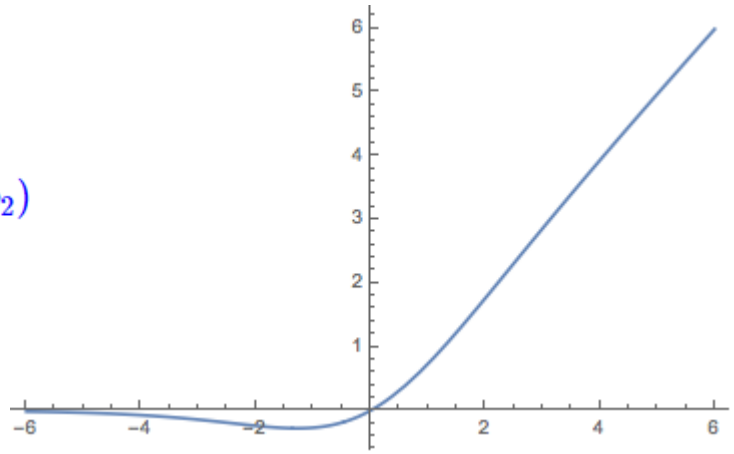
ELU缺点：

1）计算量稍大

理论上ELU虽然好于ReLU，但在实际使用中目前并没有好的证据ELU总是优于ReLU。

人工神经网络-swish激活函数

$$(\boldsymbol{W}_1\boldsymbol{x}+\boldsymbol{b}_1)\otimes\sigma(\boldsymbol{W}_2\boldsymbol{x}+\boldsymbol{b}_2)$$



谷歌团队的测试结果表明该函数在很多模型都优于ReLu。

人工神经网络-其他激活函数

$$h_i(x) = \max_{j \in [1,k]} z_{ij}$$

MaxOut函数

$$h(X) = (XW + b) \otimes \delta(XV + c)$$

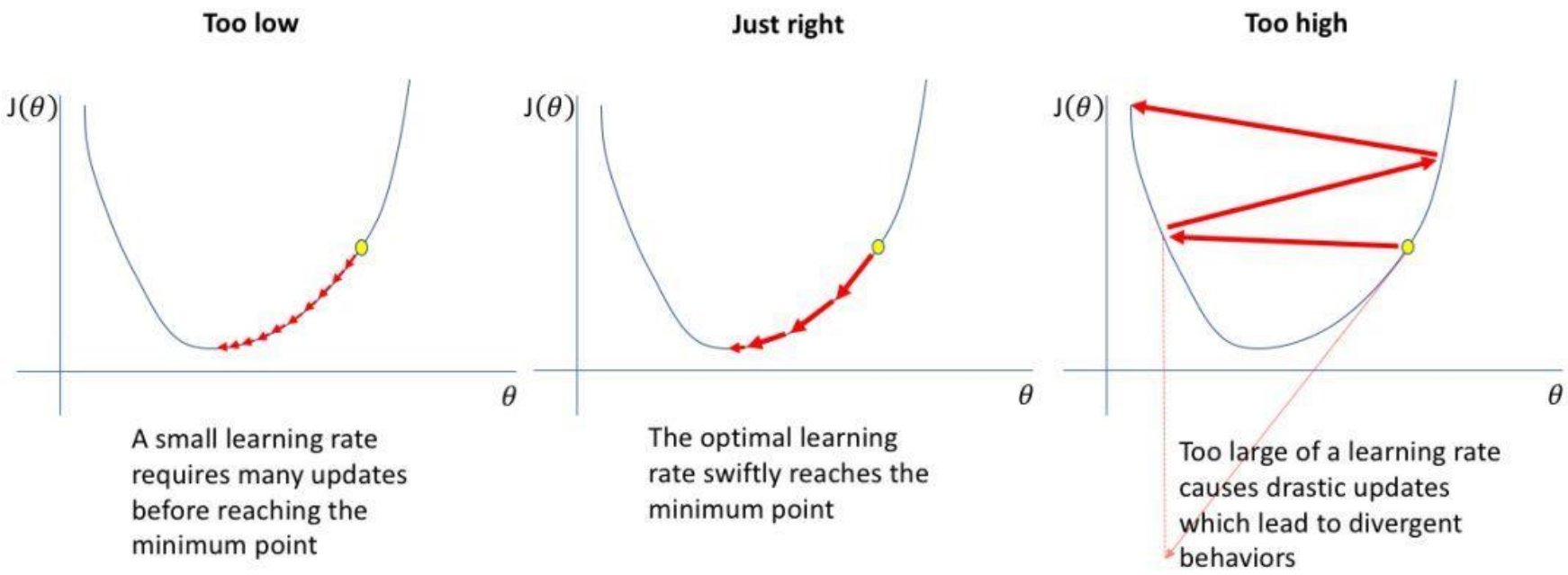
Gated linear units(GLU)

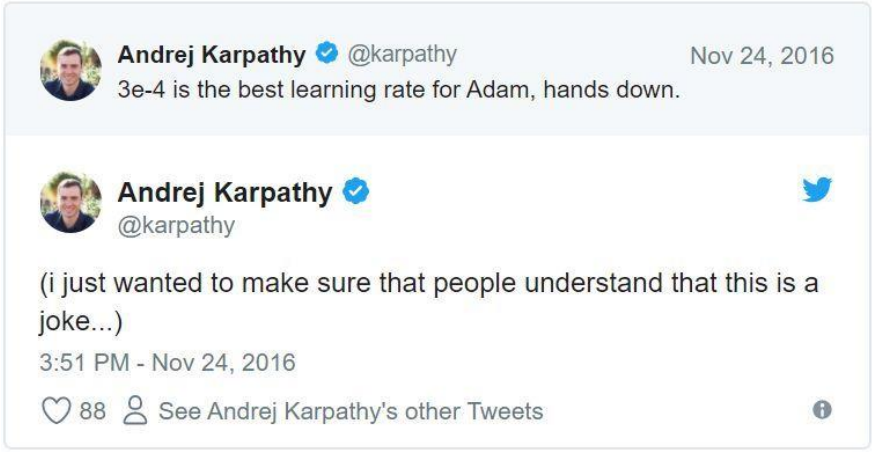
$$h(X) = \tanh(XW + b) \otimes \delta(XV + c)$$

GTU（Gated Tanh Unit）

- ◆深度学习往往需要大量时间来处理大量数据，模型的收敛速度是尤为重要的。所以，总体上来讲，训练深度学习网络尽量使用zero-centered数据（可以经过数据预处理实现）和zero-centered输出。所以要尽量选择输出具有zero-centered特点的激活函数以加快模型的收敛速度。
- ◆如果使用 ReLU，那么一定要小心设置 learning rate，而且要注意不要让网络出现很多 “dead” 神经元，如果这个问题不好解决，那么可以试试 Leaky ReLU、PReLU 或者 Maxout.
- ◆在深度网络中最好不要用sigmoid，你可以试试 tanh，不过可以预期它的效果会比不上 ReLU 和 Maxout.
- ◆根据经验，一般可以从ReLU激活函数开始，但是如果ReLU不能很好的解决问题，再去尝试其他的激活函数
- ◆ReLU永远只在隐藏层中使用
- ◆RELU如果遇到了一些死的神经元，我们可以使用Leaky ReLU函数

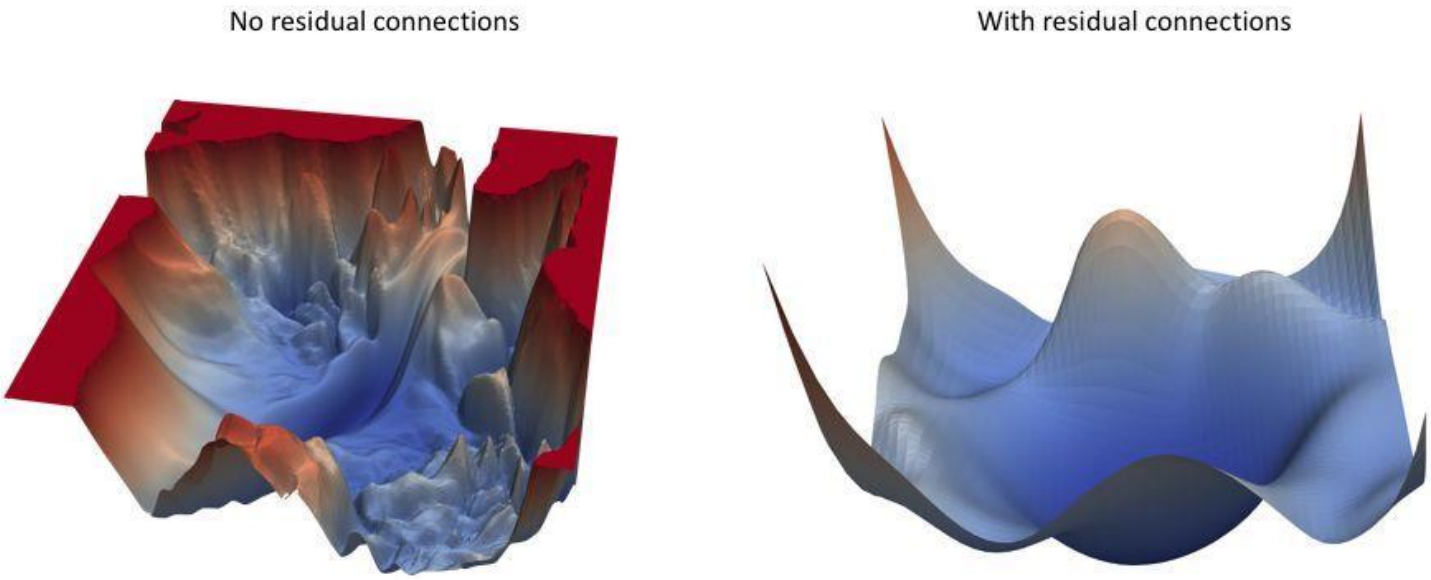
人工神经网络-学习率


Too low
Just right
Too high
J(θ)
θ
A small learning rate requires many updates before reaching the minimum point
The optimal learning rate swiftly reaches the minimum point
Too large of a learning rate causes drastic updates which lead to divergent behaviors


Andrej Karpathy @karpathy Nov 24, 2016
3e-4 is the best learning rate for Adam, hands down.
Andrej Karpathy
@karpathy
(i just wanted to make sure that people understand that this is a joke...)
3:51 PM - Nov 24, 2016
88
See Andrej Karpathy's other Tweets

人工神经网络-学习率调整

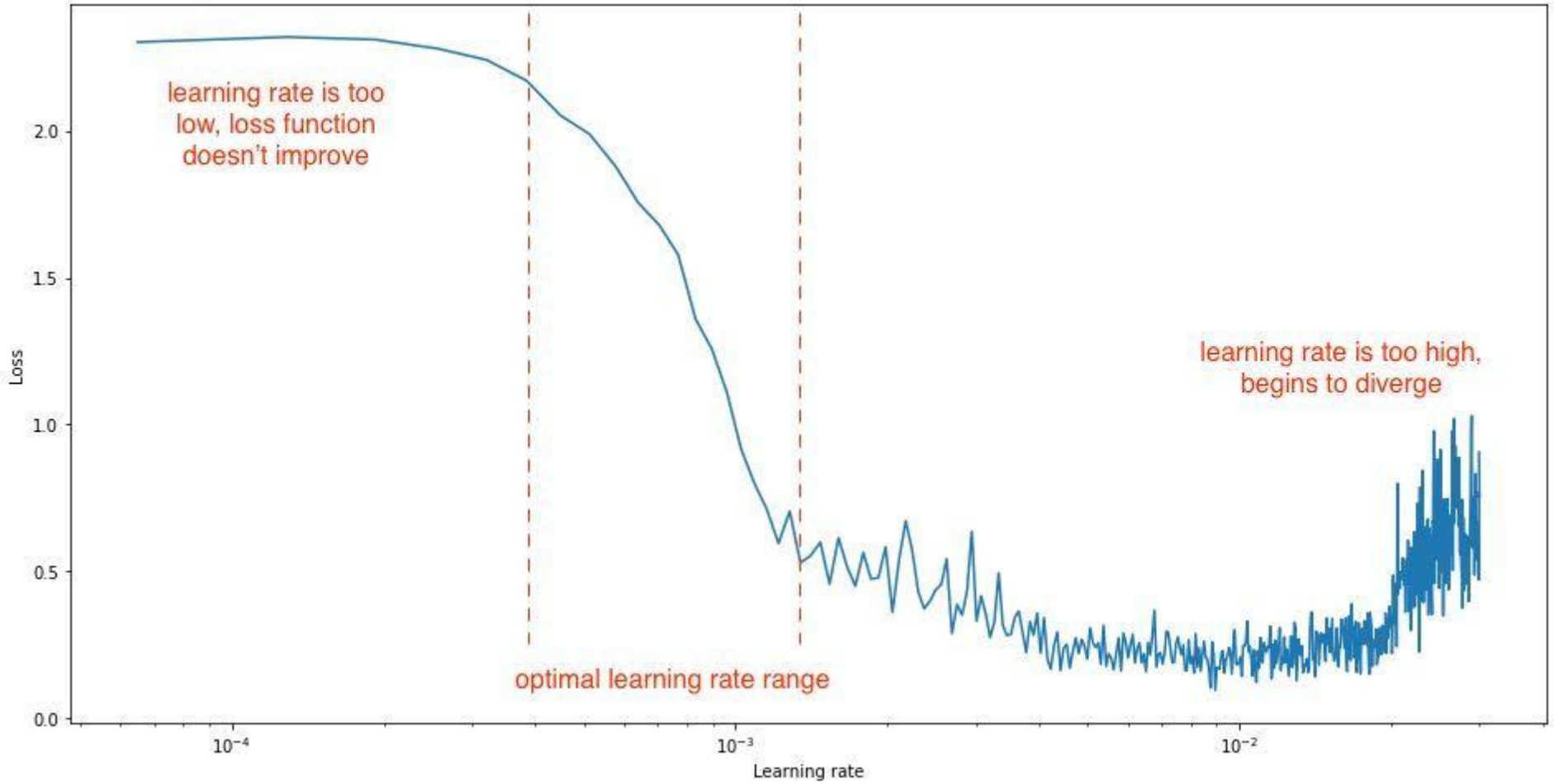
最优学习率取决于损失地图的拓扑结构



残差连接可产生更平滑的拓扑结构

《Visualizing the Loss Landscape of Neural Nets》

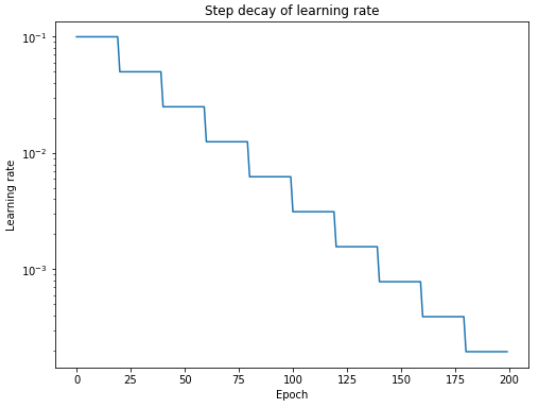
人工神经网络-一个找寻最优学习速率的系统化方法



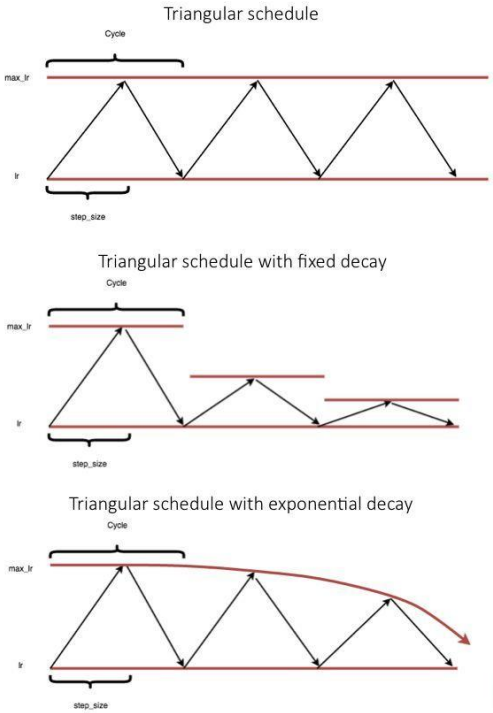
在逐步提高每一次小批量（迭代）的学习速率的同时通过做一个简单实验来观察，记录每一次增量之后的损失。这个逐步的增长可以是线性或指数的。

人工神经网络-学习速率退火（learning rate annealing）

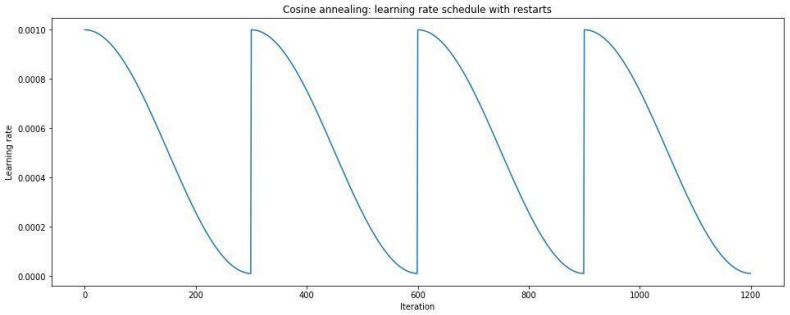
先从一个比较高的学习速率开始然后慢慢地在训练中降低学习速率。
这个方法背后的思想是我们喜欢快速地从初始参数移动到一个参数值「好」的范围，但这之后我们又想要一个学习速率小到我们可以发掘「损失函数上更深且窄的地方」。



学习速率退火的最流行方式是「步衰减」（Step Decay），其中学习率经过一定数量的训练 epochs 后下降了一定的百分比。



周期性学习率，三角形更新规则《Cyclical Learning Rates for Training Neural Networks》



带有热重启的随机梯度下降（SGDR）与周期性方法很相似，其中一个积极的退火表与周期性「再启动」融合到原始的初始学习率之中。

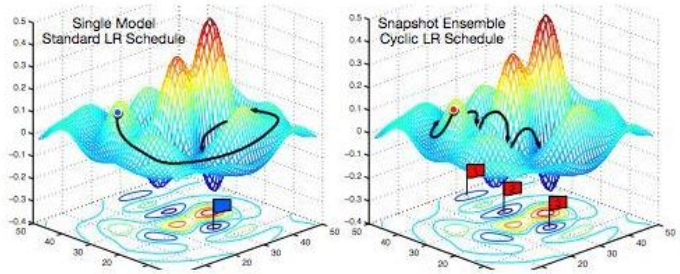


Figure 1: **Left:** Illustration of SGD optimization with a typical learning rate schedule. The model converges to a minimum at the end of training. **Right:** Illustration of Snapshot Ensembling. The model undergoes several learning rate annealing cycles, converging to and escaping from multiple local minima. We take a snapshot at each minimum for test-time ensembling.

在每一轮循环后截图一下权重，研究员可以通过训练单个模型去建立一个全套模型。这是因为从一个周期到另一个周期，这个网络「沉淀」在不同的局部最优，像在下面图中画的一样。

人工神经网络-损失函数

classification error

$$L_i = \frac{count\ of\ error\ items}{count\ of\ all\ items}$$

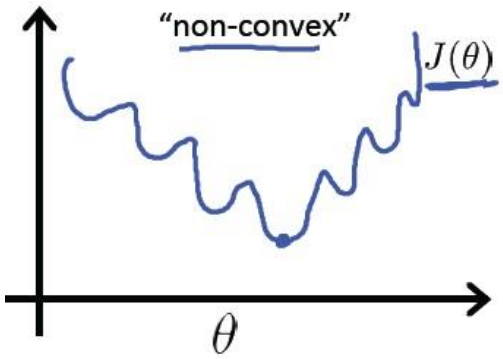
COMPUTED	TARGETS	CORRECT?
0.3 0.3 0.4	0 0 1 (democrat)	yes
0.3 0.4 0.3	0 1 0 (republican)	yes
0.1 0.2 0.7	1 0 0 (other)	no

COMPUTED	TARGETS	CORRECT?
0.1 0.2 0.7	0 0 1 (democrat)	yes
0.1 0.7 0.2	0 1 0 (republican)	yes
0.3 0.4 0.3	1 0 0 (other)	no

classification error都为1/3，无法测量概率分布的差异

Mean Squared Error

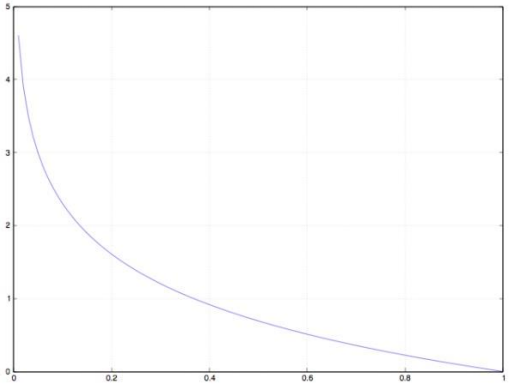
$$L_i = \sum_{c=1}^{k} (y_{ic} - p_{ic})^2$$



如果用 MSE 计算 loss，输出的曲线是波动的，有很多局部的极值点，即非凸优化问题（non-convex）。

Cross Entropy

$$L_i = -\sum_{c=1}^{k} y_{ic} \log p_{ic}$$



cross entropy 计算 loss，则依旧是一个凸优化问题，用梯度下降求解时，凸优化问题有很好的收敛特性。

为什么损失函数不用MSE
1.MSE会有梯度消失现象
2.MSE的导数非凸函数，求解最优解困难

https://www.jianshu.com/p/af1e5cff21b9

优化算法-三种梯度下降算法

梯度下降（Gradient descent, GD, aka **batch gradient descent）**，梯度在整个数据集上计算得到：

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

```
for i in range(nb_epochs):
    params_grad = evaluate_gradient(loss_function, data, params)
    params = params - learning_rate * params_grad
```

随机梯度下降（ Stochastic gradient descent, SGD），梯度在单个样本上计算得到：

$$\theta = \theta - \eta \cdot \nabla_\theta J\left(\theta; x^{(i)}; y^{(i)}\right)$$

```
for i in range(nb_epochs):
    np.random.shuffle(data)
    for example in data:
        params_grad = evaluate_gradient(loss_function, example, params)
        params = params - learning_rate * params_grad
```

小批量梯度下降（Mini-batch gradient descent），梯度在一个批次数据上计算得到：

$$\theta = \theta - \eta \cdot \nabla_\theta J\left(\theta; x^{(i:i+m)}; y^{(i:i+m)}\right)$$

```
for i in range(nb_epochs):
    np.random.shuffle(data)
    for batch in get_batches(data, batch_size=50):
        params_grad = evaluate_gradient(loss_function, batch, params)
        params = params - learning_rate * params_grad
```

1）**batchsize**：一个迭代批次样本数量的大小。即每次迭代使用的样本数量；

2）**iteration**：1个iteration是指一次迭代过程，即上面的一个批次batchsize的样本完成前向和反向的整个过程；

3）**epoch**：1个epoch是指训练集中的全部样本都训练了一次，训练集中所有的样本都被迭代了一次就是完成了一个epoch，通常将的几个epoch就是指训练集中的所有样本被迭代了几次。

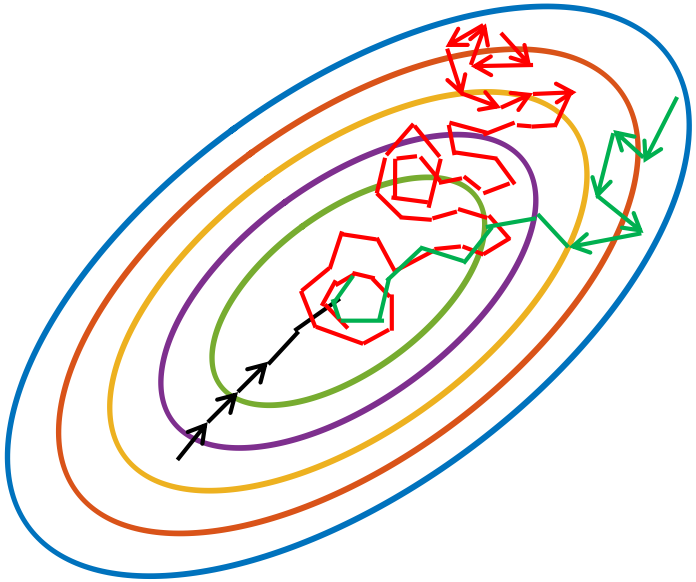
Gradients check: https://cs231n.github.io/neural-networks-3/

https://ruder.io/optimizing-gradient-descent/index.html

优化算法-小批量梯度下降

如果小批量大小=样本数n， 退化为批梯度下降

如果小批量大小=1， 退化为随机梯度下降

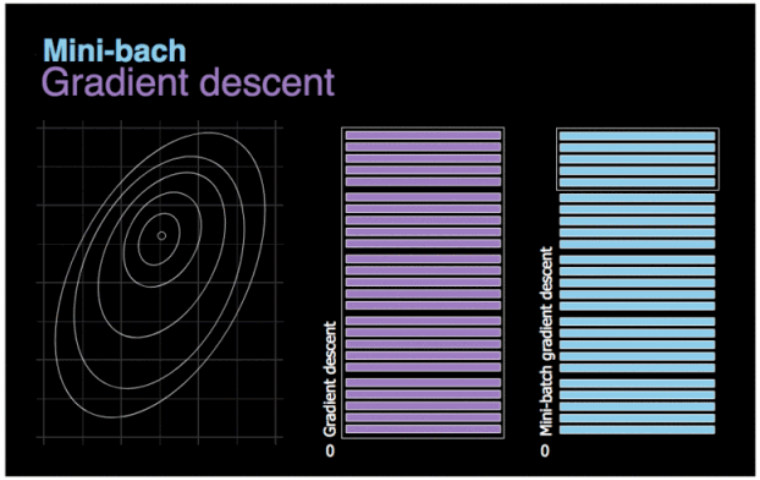


Batch gradient descent: *too long per iteration*

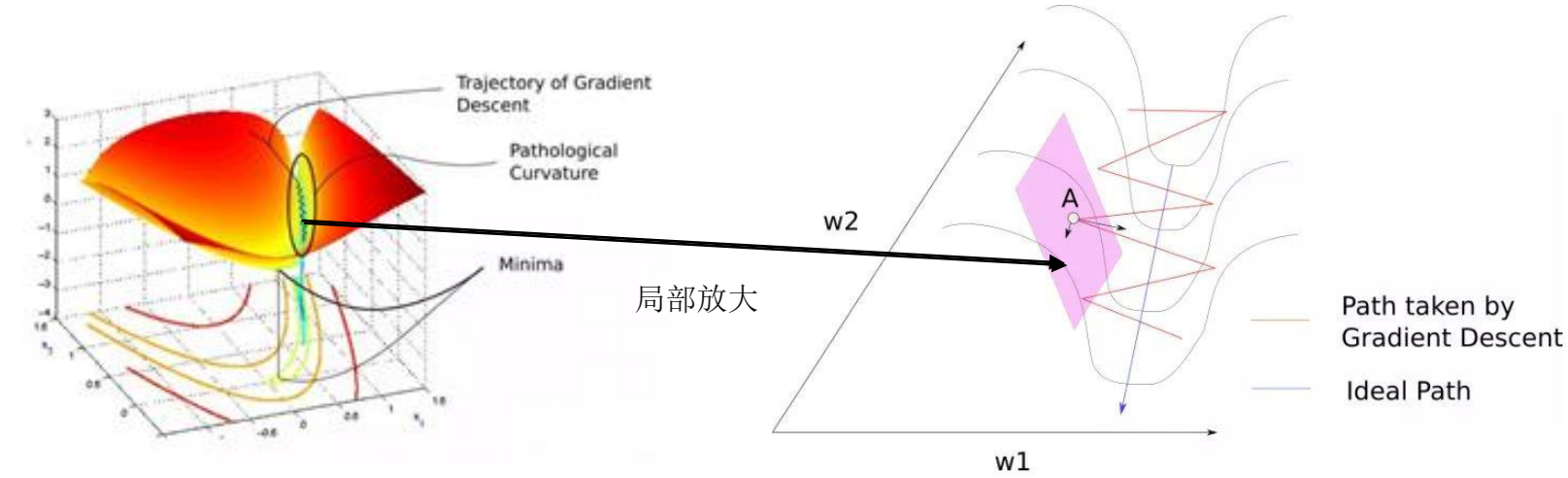
SGD: *lose speeding from vectorization*

Mini-batch gradient descent:

- *Take advantage of vectorization speeding*
- *Update parameter without processing entire training set.*

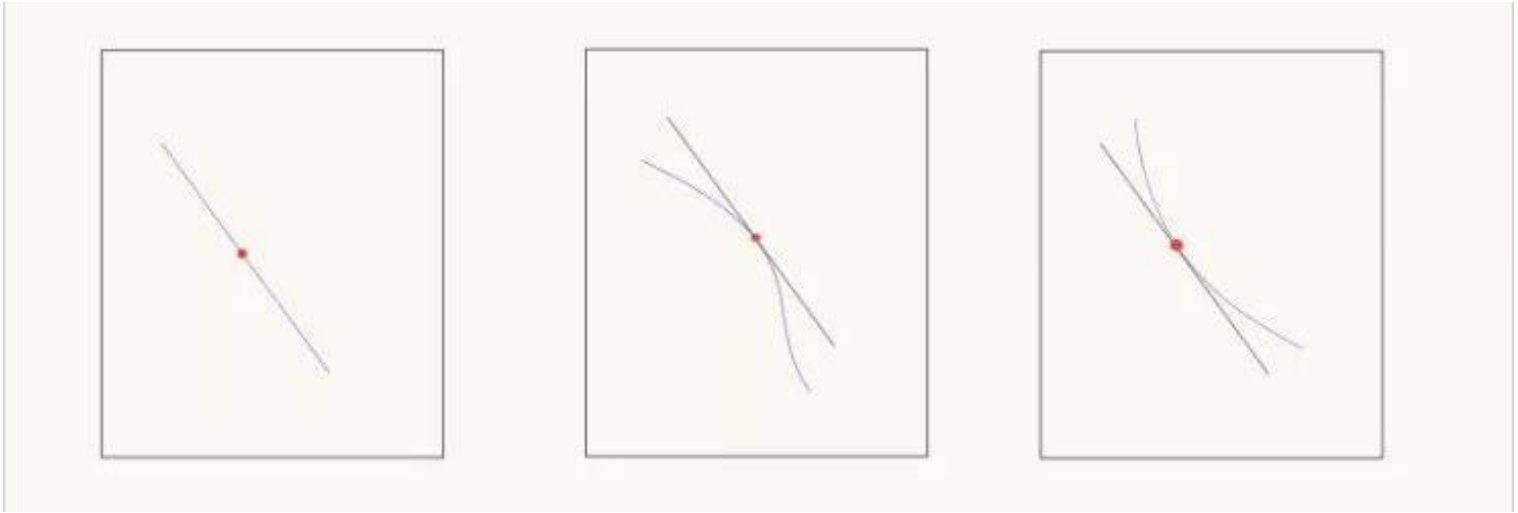


优化算法-病态曲率



损失曲线图。

损失曲线局部放大图。



梯度下降是一阶优化方法，只能说明损失是否下降以及下降的速度，而不能区分曲线是平坦的，向上的，还是向下的（不知道损失函数的曲率）。

优化算法-为什么不用牛顿法?

$$H(e)=\begin{bmatrix} \frac{\partial^2 e}{\partial w_1^2} & \frac{\partial^2 e}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 e}{\partial w_1 \partial w_n} \\ \frac{\partial^2 e}{\partial w_2 \partial w_1} & \frac{\partial^2 e}{\partial w_2^2} & \cdots & \frac{\partial^2 e}{\partial w_2 \partial w_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial^2 e}{\partial w_n \partial w_1} & \frac{\partial^2 e}{\partial w_n \partial w_2} & \cdots & \frac{\partial^2 e}{\partial w_n^2} \end{bmatrix}$$

Hessian 矩阵

Hessian 矩阵需要计算损失函数对所有权值组合的梯度。在组合已知的情况下，要求的值的数量**约是神经网络中权值数量的平方**。

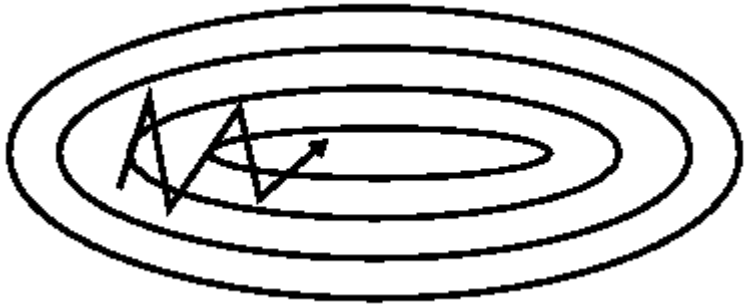
优化算法-Momentum

Momentum 不仅会使用当前梯度，还会积累之前的梯度以确定走向。

$$m_t = \gamma m_{t-1} + \eta g_t \qquad g_t = \nabla_\theta J(\theta)$$

$$\theta_t = \theta_{t-1} - m_t$$

SGD without momentum



SGD with momentum

每一步的梯度更新方向可以被进一步分解为 w1 和 w2 分量。如果我们单独的将这些向量求和，沿 w1 方向的的分量将抵消，沿 w2 方向的分量将得到加强。

- 前后梯度一致的时候能够加速学习；
- 前后梯度不一致的时候能够抑制震荡，越过局部极小值。（加速收敛，减小震荡。）

Gradients check: https://cs231n.github.io/neural-networks-3/

https://ruder.io/optimizing-gradient-descent/index.html

优化算法-AdaGrad （自适应梯度算法）

随机梯度和动量随机梯度算法都是使用全局的学习率，如何自动调节学习率？

$$v_{t,j} = v_{t-1,j} + g_{t,j}^2 \qquad g_{t,j} = \nabla_{\theta_j} J_t(\theta)$$

$$\theta_{t,j} = \theta_{t-1,j} - \frac{\alpha}{\sqrt{v_{t,j} + \delta}} g_{t,j}$$

其中 $\delta = 10^{-7}$，防止数值溢出。

在参数空间更为平缓的方向，会取得更大的进步（因为平缓，所以历史梯度平方和较小，对应学习下降的幅度较小），并且能够使得陡峭的方向变得平缓，从而加快训练速度。

同时，网络刚开始时学习率很大，当走完一段距离后小心翼翼。

AdaGrad的缺点：容易过分的降低学习率率使其提前停止更新。

Gradients check: https://cs231n.github.io/neural-networks-3/

https://ruder.io/optimizing-gradient-descent/index.html

优化算法-RMSprop

RMSProp 算法在AdaGrad基础上引入了衰减因子，如下式，RMSProp在梯度累积的时候，会对“过去”与“现在”做一个平衡，通过超参数β进行调节衰减量

$$v_{t,j} = \beta v_{t-1,j} + (1-\beta)\mathrm{g}_{\mathrm{t,j}}^2 \qquad \mathrm{g}_{t,\mathrm{j}} = \nabla_{\theta_j} J_t(\theta)$$

$$\theta_{t,j} = \theta_{t-1,j} - \frac{\alpha}{\sqrt{v_{t,j}+\delta}} g_{t,j}$$

其中 $\delta = 10^{-7}$，防止数值溢出。

β通常设为 0.001。

Gradients check: https://cs231n.github.io/neural-networks-3/

https://ruder.io/optimizing-gradient-descent/index.html

优化算法- Adaptive Moment Estimation (Adam)

适应性动量估计法（Adam）除了存储类似 Adadelta 法或 RMSprop 中指数衰减的过去梯度平方均值 外，也存储像动量法中的指数衰减的过去梯度值均值。

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

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\mathrm{g_t^2}$$

m_t和v_t分别是梯度的一阶矩（均值）和二阶矩（表示不确定度的方差）。

由于m_t和v_t一开始被初始化为0时，会有趋向0的偏差。所以使用偏差纠正系数，来修正一阶矩和二阶矩的偏差。

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

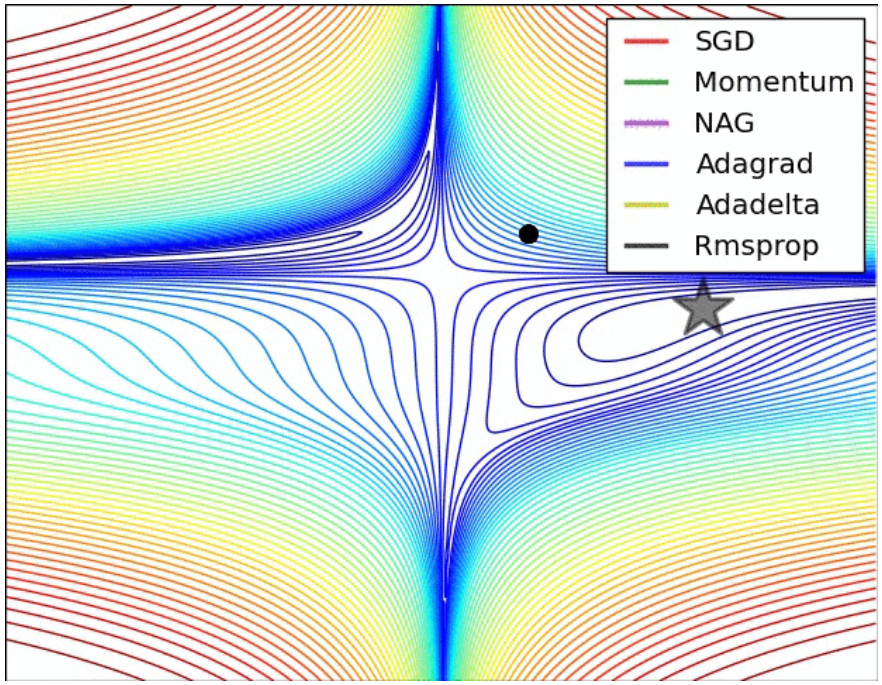
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t$$

默认值， β_1 =0.9 ， β_2 =0.999， $\epsilon = 10^{-8}$

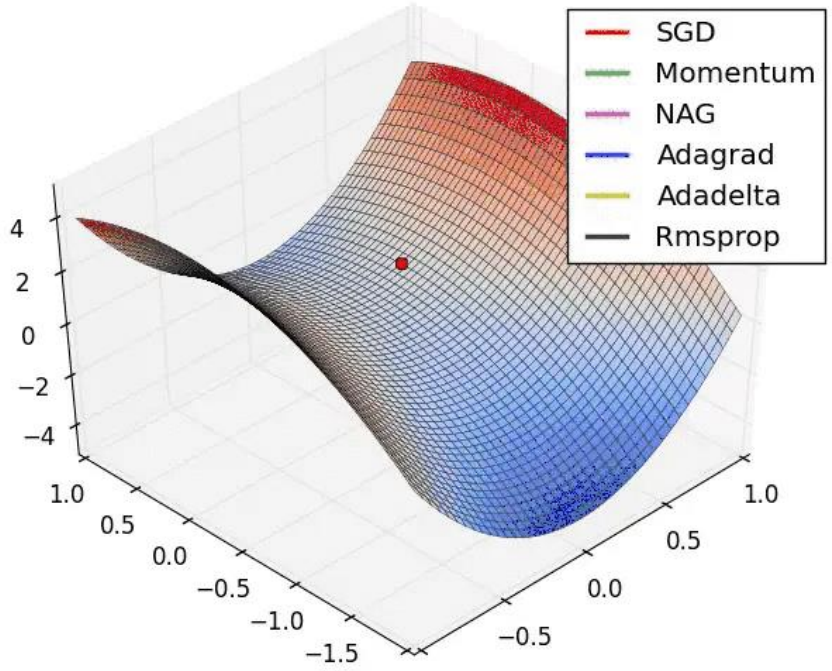
Gradients check: https://cs231n.github.io/neural-networks-3/

https://ruder.io/optimizing-gradient-descent/index.html

优化算法- Summary



Contours of a loss surface and time evolution of different optimization algorithms.



A visualization of a saddle point in the optimization landscape, where the curvature along different dimension has different signs (one-dimension curves up and another down)

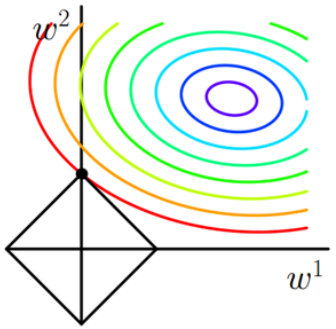
As we can see, the adaptive learning-rate methods, i.e. Adagrad, Adadelta, RMSprop, and Adam are most suitable and provide the best convergence for these scenarios.

Gradients check: https://cs231n.github.io/neural-networks-3/

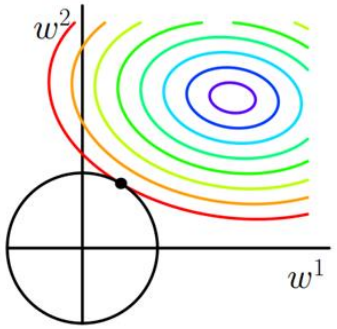
https://ruder.io/optimizing-gradient-descent/index.html

过拟合解决方法

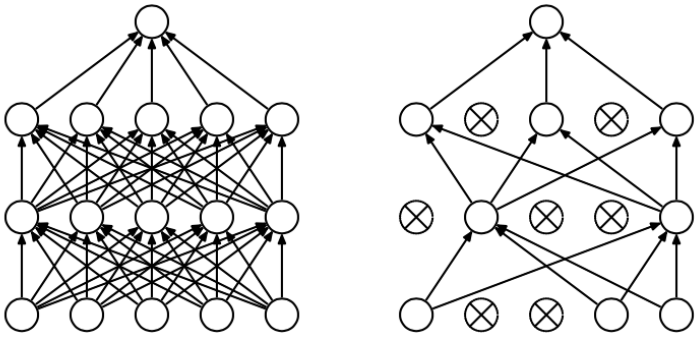
1. 正则化
   - L1正则化
   - L2正则化
2. 剪枝处理
   - 对决策树模型进行剪枝
3. 提前终止迭代（Early stopping）
   - 在神经网络训练中，提前终止迭代可以有效的控制权值参数的大小，从而降低模型的复杂度。
4. 权值共享
   - 在神经网络中，权值共享的目的旨在减小模型中的参数，同时还能较少计算量。
5. 增加噪声
   - 对数据或者权值添加噪声
6. Batch Normalization
   - 将每一层的输入值做归一化处理，并且会重构归一化处理之后的数据，确保数据的分布不会发生变化。
7. Bagging和Boosting
   - 集成方法
8. Dropout
   - 在一定的概率上（通常设置为0.5，原因是此时随机生成的网络结构最多）隐式的去除网络中的神经元。



L1正则化



L2正则化



Dropout

Gradients check: https://cs231n.github.io/neural-networks-3/

https://ruder.io/optimizing-gradient-descent/index.html

神经网络-开发工具


Chainer
PYTORCH
mxnet
GLUON
theano
Caffe
K
Caffe2
Microsoft
CNTK
TensorFlow
before
2012
2013
2014
2015
2016
2017

神经网络-Caffe

ResNet-101-deploy.prototext

```
layer {
        bottom: "data"
        top: "conv1"
        name: "conv1"
        type: "Convolution"
        convolution_param {
                num_output: 64
                kernel_size: 7
                pad: 3
                stride: 2
```

- Protobuf 作为界面
- 强大的视觉模型覆盖率
- 灵活可移植
- 不易开发

神经网络-Tensorflow

实现Adam算法

```
# m_t = beta1 * m + (1 - beta1) * g_t
m = self.get_slot(var, "m")
m_scaled_g_values = grad.values * (1 - beta1_t)
m_t = state_ops.assign(m, m * beta1_t,
                       use_locking=self._use_locking)
m_t = state_ops.scatter_add(m_t, grad.indices, m_scaled_g_values,
                            use_locking=self._use_locking)
```

- Python的域特定语言（DSL）
- 丰富的算子
- 丰富的功能
- 代码可读性有限

神经网络-Keras

```
model = Sequential()
model.add(Dense(512, activation='relu',
                input_shape=(784,)))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))

model.compile(...)
model.fit(...)
```

- 简单的DSL for Python，可以使用多个后端(TensoFlow, MXNet, CNTK…)
- 比 TensorFlow 简易
- 相对更慢
- 不易开发和调试

神经网络-Pytorch

```
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out
```

- Torch tensors + chainer 神经网络
- 易开发和调试
- 不易配置部署

神经网络-MXNet

实现Resnet算法

```
bn1 = sym.BatchNorm(data=data, fix_gamma=Fal
act1 = sym.Activation(data=bn1, act_type='re
conv1 = sym.Convolution(data=act1, num_filte
```

实现Adam算法

```
coef2 = 1. - self.beta2**t
lr *= math.sqrt(coef2)/coef1

weight -= lr*mean/(sqrt(variance) + self.epsilon)
```

- 像 Numpy 一样的 Tensor + 像 Keras 一样的神经网络
- 高性能
- 高实用性

神经网络-PyTorch安装

系统
- Win → 参照 https://zhuanlan.zhihu.com/p/26871672 用conda安装
- Linux
- OSX

参照官网 http://pytorch.org/ 根据需求选择代码安装

Get Started.

Select your preferences, then run the PyTorch install command.

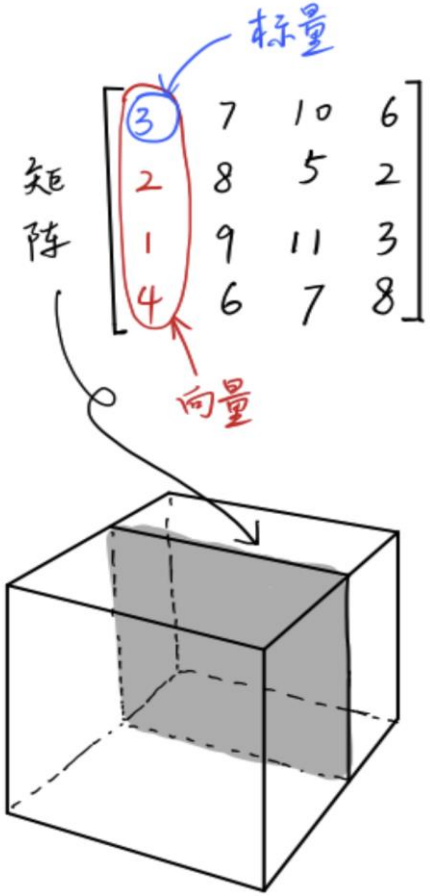
Please ensure that you are on the latest pip and numpy packages.
Anaconda is our recommended package manager

OS	Linux	OSX	
Package Manager	conda	pip	Source
Python	2.7	3.5	3.6
CUDA	7.5	8.0	None

Run this command: conda install pytorch torchvision -c soumith

神经网络-PyTorch中的Tensor

1.Tensor



张量与numpy的ndarray（多维数组）类似，还可以使用GPU加速计算。
标量是零维的张量，向量是一维的张量，矩阵是二维的张量……除此之外，张量还可以是四维的、五维的。

神经网络-PyTorch中的Tensor

创建Tensor

函数	功能
Tensor(*size)	直接从参数构造一个的张量，支持 list,numpy 数组
eye(row, column)	创建指定行数，列数的二维单位 tensor
linspace(start,end,steps)	从 step 到 end，均匀切分成 steps 份
logspace(start,end,steps)	从 10^step，到 10^end，均匀切分成 steps 份
rand/randn(*size)	生成[0,1)均匀分布/标准正态分布数据
ones(*size)	返回指定 shape 的张量，元素初始为 1
zeros(*size)	返回指定 shape 的张量，元素初始为 0
ones_like(t)	返回与 t 的 shape 相同的张量，且元素初始为 1
zeros_like(t)	返回与 t 的 shape 相同的张量，且元素初始为 0
arange(start,end,step)	在区间[start,end)上以间隔 step 生成一个序列张量
from_numpy(*ndarray*)	从 ndarray 创建一个 tensor

神经网络-PyTorch中的Tensor

创建Tensor

```
import torch


#生成一个单位矩阵
torch.eye(2,2)
#自动生成全是0的矩阵
torch.zeros(2,3)
#根据规则生成数据
torch.linspace(1,10,4)
#生成满足均匀分布随机数
torch.rand(2,3)
#生成满足标准分布随机数
torch.randn(2,3)
#返回所给数据形状相同，值全为0的张量
torch.zeros_like(torch.rand(2,3))
```

神经网络-PyTorch中的Tensor

创建Tensor

torch.Tensor与torch.tensor的几点区别

- torch.Tensor是torch.empty和torch.tensor之间的一种混合，但是，当传入数据时，torch.Tensor使用全局默认dtype（FloatTensor），torch.tensor从数据中推断数据类型。
- torch.tensor(1)返回一个固定值1，而torch.Tensor(1)返回一个大小为1的张量，它是随机初始化的值。

神经网络-PyTorch中的Tensor

修改Tensor形状

函数	说明
size()	返回张量的 shape 属性值，与函数 shape(0.4 版新增)等价
numel(input)	计算 tensor 的元素个数
view(*shape)	修改 tensor 的 shape，与 reshape(0.4 版新增)类似，但 view 返回的对象与源 tensor 共享内存，修改一个另一个同时修改。Reshape 将生成新的 tensor，而且不要求源 tensor 是连续的。View(-1)展平数组。
resize	类似于 view，但在 size 超出时会重新分配内存空间
item	若 tensor 为单元素，则返回 pyton 的标量
unsqueeze	在指定维度增加一个"1"
squeeze	在指定维度压缩一个"1"

神经网络-PyTorch中的Tensor

修改Tensor形状

```
import torch

#生成一个形状为2x3的矩阵
x = torch.randn(2, 3)
#查看矩阵的形状
x.size()   #结果为torch.Size([2, 3])

#查看x的维度
x.dim()    #结果为2
#把x变为3x2的矩阵
x.view(3,2)
#把x展平为1维向量
y=x.view(-1)
y.shape
#添加一个维度
z=torch.unsqueeze(y,0)
#查看z的形状
z.size()   #结果为torch.Size([1, 6])
#计算Z的元素个数
z.numel()   #结果为6
```

修改Tensor形状

torch.view与torch.reshpae的异同

- reshape()可以由torch.reshape(),也可由torch.Tensor.reshape()调用。view()只可由torch.Tensor.view()来调用。
- 对于一个将要被view的Tensor，新的size必须与原来的size与stride兼容。否则，在view之前必须调用contiguous()方法。
- 同样也是返回与input数据量相同，但形状不同的tensor。若满足view的条件，则不会copy，若不满足，则会copy
- 如果只想重塑张量，请使用torch.reshape。如果还关注内存使用情况并希望确保两个张量共享相同的数据，请使用torch.view。

索引操作

函数	说明
index_select(input, dim, index)	在指定维度上选择一些行或列
nonzero(input)	获取非 0 元素的下标
masked_select(input, mask)	使用二元值进行选择
gather(input, dim, index)	在指定维度上选择数据，输出的形状与 index（index 的类型必须是 LongTensor 类型的）一致
scatter_(input, dim, index, src)	为 gather 的反操作，根据指定索引补充数据

神经网络-PyTorch中的Tensor

索引操作

```
import torch

#设置一个随机种子
torch.manual_seed(100)
#生成一个形状为2x3的矩阵
x = torch.randn(2, 3)
#根据索引获取第1行，所有数据
x[0,:]
#获取最后一列数据
x[:,-1]
#生成是否大于0的Byter张量
mask=x>0
#获取大于0的值
torch.masked_select(x,mask)
#获取非0下标,即行，列索引
torch.nonzero(mask)
#获取指定索引对应的值,输出根据以下规则得到
#out[i][j] = input[index[i][j]][j]  # if dim == 0
#out[i][j] = input[i][index[i][j]]  # if dim == 1
index=torch.LongTensor([[0,1,1]])
torch.gather(x,0,index)
index=torch.LongTensor([[0,1,1],[1,1,1]])
a=torch.gather(x,1,index)
#把a的值返回到一个2x3的0矩阵中
z=torch.zeros(2,3)
z.scatter_(1,index,a)
```

广播机制

```
import torch
import numpy as np

A = np.arange(0, 40,10).reshape(4, 1)
B = np.arange(0, 3)
#把ndarray转换为Tensor
A1=torch.from_numpy(A)  #形状为4x1
B1=torch.from_numpy(B)  #形状为3
#Tensor自动实现广播
C=A1+B1
#我们可以根据广播机制，手工进行配置
#根据规则1，B1需要向A1看齐，把B变为（1,3）
B2=B1.unsqueeze(0)  #B2的形状为1x3
#使用expand函数重复数组，分别的4x3的矩阵
A2=A1.expand(4,3)
B3=B2.expand(4,3)
#然后进行相加,C1与C结果一致
C1=A2+B3
```

逐元素操作

函数	说明
abs/add	绝对值/加法
addcdiv(t, v, t1, t2)	t1 与 t2 的按元素除后，乘 v 加 t
addcmul(t, v, t1, t2)	t1 与 t2 的按元素乘后，乘 v 加 t
ceil/floor	向上取整/向下取整
clamp(t, min, max)	将张量元素限制在指定区间
exp/log/pow	指数/对数/幂
mul(或*)/neg	逐元素乘法/取反
sigmoid/tanh/softmax	激活函数
sign/sqrt	取符号/开根号

逐元素操作

```
import torch


t = torch.randn(1, 3)
t1 = torch.randn(3, 1)
t2 = torch.randn(1, 3)
#t+0.1*(t1/t2)
torch.addcdiv(t, 0.1, t1, t2)
#计算sigmoid
torch.sigmoid(t)
#将t限制在[0,1]之间
torch.clamp(t,0,1)
#t+2进行就地运算
t.add_(2)
```

归并操作

函数	说明
cumprod(t, axis)	在指定维度对 t 进行累积
cumsum	在指定维度对 t 进行累加
dist(a,b,p=2)	返回 a,b 之间的 p 阶范数
mean/median	均值/中位数
std/var	标准差/方差
norm(t,p=2)	返回 t 的 p 阶范数
prod(t)/sum(t)	返回 t 所有元素的积/和

归并操作

```
import torch


#生成一个含6个数的向量
a=torch.linspace(0,10,6)
#使用view方法，把a变为2x3矩阵
a=a.view((2,3))
#沿y轴方向累加，即dim=0
b=a.sum(dim=0)  #b的形状为[3]
#沿y轴方向累加，即dim=0,并保留含1的维度
b=a.sum(dim=0,keepdim=True) #b的形状为[1,3]
```

比较操作

函数	说明
eq	比较 tensor 是否相等，支持 broadcast
equal	比较 tensor 是否有相同的 shape 与值
ge/le/gt/lt	大于/小于比较/大于等于/小于等于比较
max/min(t,axis)	返回最值，若指定 axis，则额外返回下标
topk(t,k,axis)	在指定的 axis 维上取最高的 K 个值

```
import torch

x=torch.linspace(0,10,6).view(2,3)
#求所有元素的最大值
torch.max(x)   #结果为10
#求y轴方向的最大值
torch.max(x,dim=0)  #结果为[6,8,10]
#求最大的2个元素
torch.topk(x,1,dim=0)  #结果为[6,8,10],对应索引为tensor([[1, 1, 1]
```

矩阵操作

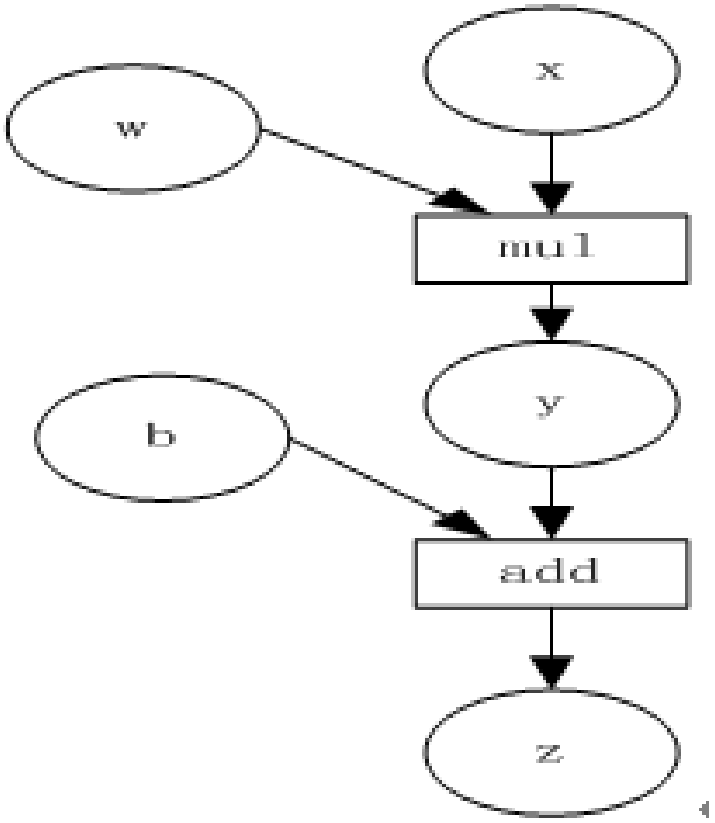
函数	说明
dot(t1, t2)	计算张量(1D)的内积或点积
mm(mat1, mat2)/bmm(batch1,batch2)	计算矩阵乘法/含 batch 的 3D 矩阵乘法
mv(t1, v1)	计算矩阵与向量乘法
t	转置
svd(t)	计算 t 的 SVD 分解

```
import torch

a=torch.tensor([2, 3])
b=torch.tensor([3, 4])

torch.dot(a,b)  #运行结果为18
x=torch.randint(10,(2,3))
y=torch.randint(6,(3,4))
torch.mm(x,y)
x=torch.randint(10,(2,2,3))
y=torch.randint(6,(2,3,4))
torch.bmm(x,y)
```

Pytorch与Numpy比较

操作类别	Numpy	PyTorch
数据类型	np.ndarray	torch.Tensor
	np.float32	torch.float32; torch.float
	np.float64	torch.float64; torch.double
	np.int64	torch.int64; torch.long
从已有数据构建	np.array([3.2, 4.3], dtype=np.float16)	torch.tensor([3.2, 4.3], dtype=torch.float16)
	x.copy()	x.clone()
	np.concatenate	torch.cat
线性代数	np.dot	torch.mm
属性	x.ndim	x.dim()
	x.size	x.nelement()
形状操作	x.reshape	x.reshape; x.view
	x.flatten	x.view(-1)
类型转换	np.floor(x)	torch.floor(x); x.floor()
比较	np.less	x.lt
	np.less_equal/np.greater	x.le/x.gt
	np.greater_equal/np.equal /np.not_equal	x.ge/x.eq/x.ne
随机种子	np.random.seed	torch.manual_seed

计算图



z=wx+b

等价于

y=wx

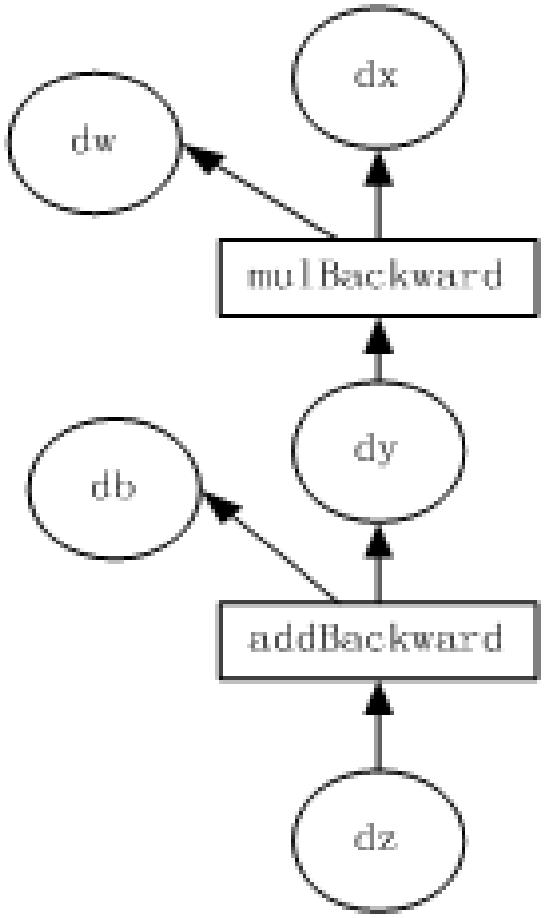
Z=y+b

计算图

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x} = w$$

$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial w} = x$$

$$\frac{\partial z}{\partial b} = 1$$



标量反向传播

(1) 定义叶子节点及算子节点

```
import torch


#定义输入张量x
x=torch.Tensor([2])
#初始化权重参数W,偏移量b、并设置require_grad属性为True，为自动求导
w=torch.randn(1,requires_grad=True)
b=torch.randn(1,requires_grad=True)
#实现前向传播
y=torch.mul(w,x)  #等价于w*x
z=torch.add(y,b)  #等价于y+b
#查看x,w，b页子节点的requite_grad属性
print("x,w,b的require_grad属性分别为：
{},{},{}".format(x.requires_grad,w.requires_grad,b.requires_grad))
```

运行结果：x, w, b的require_grad属性分别为： False,True,True

标量反向传播

(2) 查看叶子节点、非叶子节点的其他属性

```
#查看非叶子节点的requres_grad属性,
print("y，z的requires_grad属性分别为：{},{}".format(y.requires_grad,z.requires_grad))
#因与w，b有依赖关系，故y，z的requires_grad属性也是：True,True
#查看各节点是否为叶子节点
print("x，w，b，y，z的是否为叶子节点：
{},{},{},{},{}".format(x.is_leaf,w.is_leaf,b.is_leaf,y.is_leaf,z.is_leaf))
#x，w，b，y，z的是否为叶子节点：True,True,True,False,False
#查看叶子节点的grad_fn属性
print("x，w，b的grad_fn属性：{},{},{}".format(x.grad_fn,w.grad_fn,b.grad_fn))
#因x，w，b为用户创建的，为通过其他张量计算得到，故x，w，b的grad_fn属性：
None,None,None
#查看非叶子节点的grad_fn属性
print("y，z的是否为叶子节点：{},{}".format(y.grad_fn,z.grad_fn))
#y，z的是否为叶子节点：,
```

标量反向传播

(3)自动求导，实现梯度方向传播，即梯度的反向传播。

```
#基于z张量进行梯度反向传播,执行backward之后计算图会自动清空，
z.backward()
#如果需要多次使用backward，需要修改参数retain_graph为True，此时梯度是累加的
#z.backward(retain_graph=True)

#查看叶子节点的梯度，x是叶子节点但它无需求导，故其梯度为None
print("参数w,b的梯度分别为:{},{},{}".format(w.grad,b.grad,x.grad))
#参数w,b的梯度分别为:tensor([2.]),tensor([1.]),None

#非叶子节点的梯度，执行backward之后，会自动清空
print("非叶子节点y,z的梯度分别为:{},{}".format(y.grad,z.grad))
#非叶子节点y,z的梯度分别为:None,None
```

非标量反向传播

backward(gradient=None, retain_graph=None, create_graph=False)

```
import torch

#定义叶子节点张量x，形状为1x2
x= torch.tensor([[2, 3]], dtype=torch.float,
requires_grad=True)
#初始化Jacobian矩阵
J= torch.zeros(2 ,2)
#初始化目标张量，形状为1x2
y = torch.zeros(1, 2)
#定义y与x之间的映射关系：
#y1=x1**2+3*x2，y2=x2**2+2*x1
y[0, 0] = x[0, 0] ** 2 + 3 * x[0 ,1]
y[0, 1] = x[0, 1] ** 2 + 2 * x[0, 0]

#生成y1对x的梯度
y.backward(torch.Tensor([[1, 0]]),retain_graph=True)
J[0]=x.grad
#梯度是累加的，故需要对x的梯度清零
x.grad = torch.zeros_like(x.grad)
#生成y2对x的梯度
y.backward(torch.Tensor([[0, 1]]))
J[1]=x.grad
#显示jacobian矩阵的值
print(J)
```

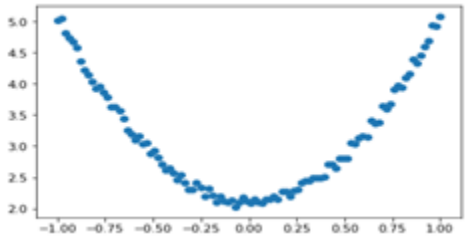
实现机器学习

（1）导入需要的库

```
import torch as t
%matplotlib inline
from matplotlib import pyplot as plt
```

（2）生成训练数据，并可视化数据分布情况

```
t.manual_seed(100)
dtype = t.float
#生成x坐标数据，x为tenor，需要把x的形状转换为100x1
x = t.unsqueeze(torch.linspace(-1, 1, 100), dim=1)
#生成y坐标数据，y为tenor，形状为100x1，另加上一些噪音
y = 3*x.pow(2) +2+ 0.2*torch.rand(x.size())

# 画图，把tensor数据转换为numpy数据
plt.scatter(x.numpy(), y.numpy())
plt.show()
```



实现机器学习

（3）初始化权重参数

```
# 随机初始化参数，参数w，b为需要学习的，故需requires_grad=True
w = t.randn(1,1, dtype=dtype,requires_grad=True)
b = t.zeros(1,1, dtype=dtype, requires_grad=True)
```

（4）训练模型

```
lr =0.001 # 学习率

for ii in range(800):
    # 前向传播，并定义损失函数loss
    y_pred = x.pow(2).mm(w) + b
    loss = 0.5 * (y_pred - y) ** 2
    loss = loss.sum()

    # 自动计算梯度，梯度存放在grad属性中
    loss.backward()

    # 手动更新参数，需要用torch.no_grad()，使上下文环境中切断自动求导的计算
    with t.no_grad():
        w -= lr * w.grad
        b -= lr * b.grad

    # 梯度清零
        w.grad.zero_()
        b.grad.zero_()
```

实现机器学习

（5）可视化训练结果

```
plt.plot(x.numpy(), y_pred.detach().numpy(),'r-
',label='predict')#predict
plt.scatter(x.numpy(),
y.numpy(),color='blue',marker='o',label='true') # true data
plt.xlim(-1,1)
plt.ylim(2,6)
plt.legend()
plt.show()

print(w, b)
```