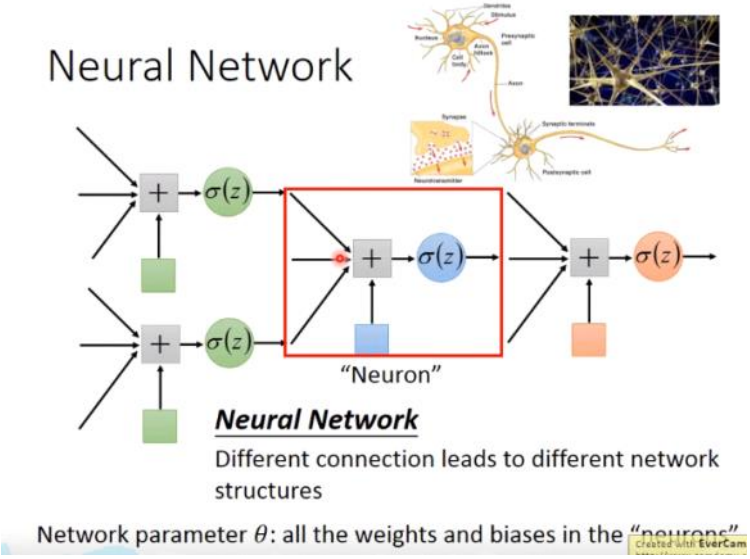
深度学习 Deep learning

2020年7月10日 20:28

深度学习的三个步骤，先定义不同的神经网络接着对其功能进行评估，最后挑选出最优功能的定义

Neural Network

根据权重：w 和偏置：b 对神经元进行运算



不同神经元的连接方式

1 把Neuron排成两排，每一组都有自己的权重weight和偏置bias，是根据trainng data找出来的

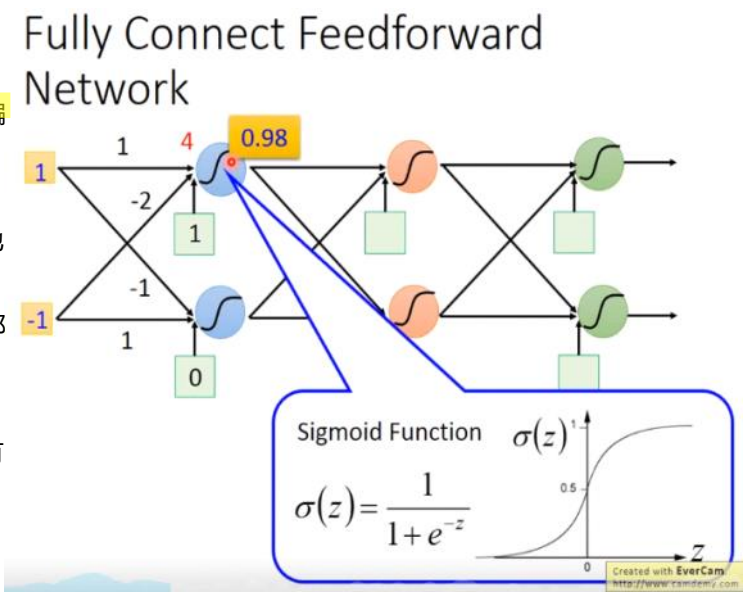
2 输入1，-1，经过若干个Neuron Network，使用其中的权重和偏置，再加上SIGMOD函数，得到新的值，不断地变换，到最终结果。

3 如果该Neuron Network中的参数（所有权重和偏置）都知道的话，则该Neuron Network就是一个function。其input是一个vector，output是另外一个vector。

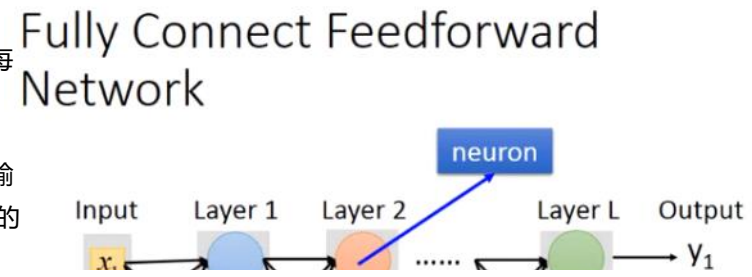
如果给一个Neuron Network结构设置不同的参数（所有权重和偏置），那么可以得到多个function，即定义了a function set。

类比，Neuron Network的function相对一般的机器学习算法中的function很大。

所以，决定了Neuron Network的结构，就决定了一个function set。

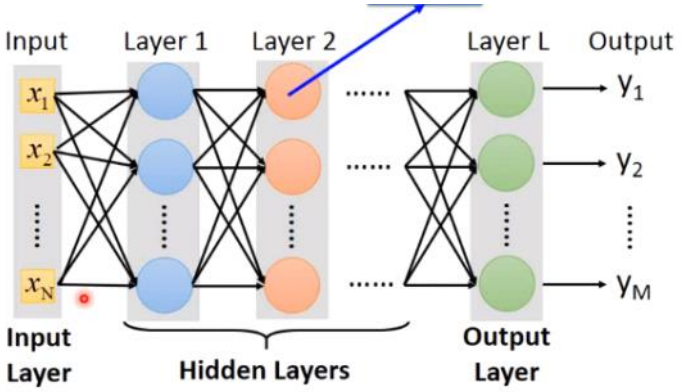


1 好多排的Neuron，每一排的Neuron的数目可能很多，每个球代表一个Neuron。



2 第一排Neuron1的输出会接给第二排每一个Neuron2的输出。Layer2的input是layer1的output。因为两个layer之间的neuron两两链接，所以叫fully connect network

2 第一排Neuron1的输出会接给第二排每一个Neuron2的输出。Layer2的input是layer1的output。因为两个layer之间的neuron两两链接，所以叫fully connect network



3 因为传奇的方向是有layer1，到layer2，到layer3，由后往前传，则叫feedforward network

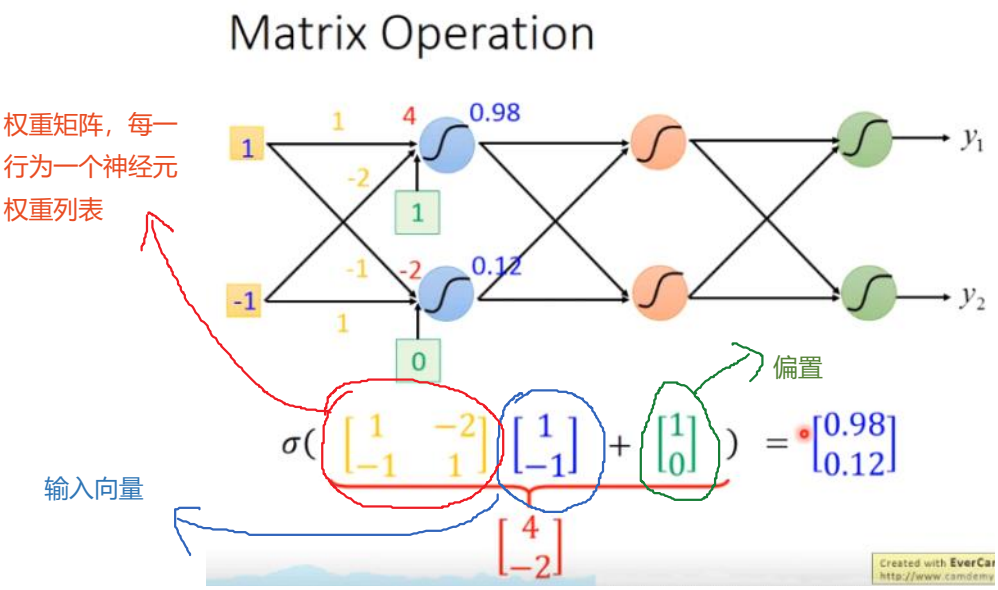
4 整个输入需要一个input，对layer1的每一个Neuron的input，就是input layer的每一个dimension。

最后一个layer L直接输出对应的yi值，Output Layer。

具体计算方法：

1 一个Neuron Network所做的工作就是 一连串的vector乘以matrix，再加上matrix

2 写成矩阵方式，可使用GPU并行计算技术，加速计算。当需要做矩阵运算的时候，就调用一下GPU，使其完成该计算工作，比CPU快。

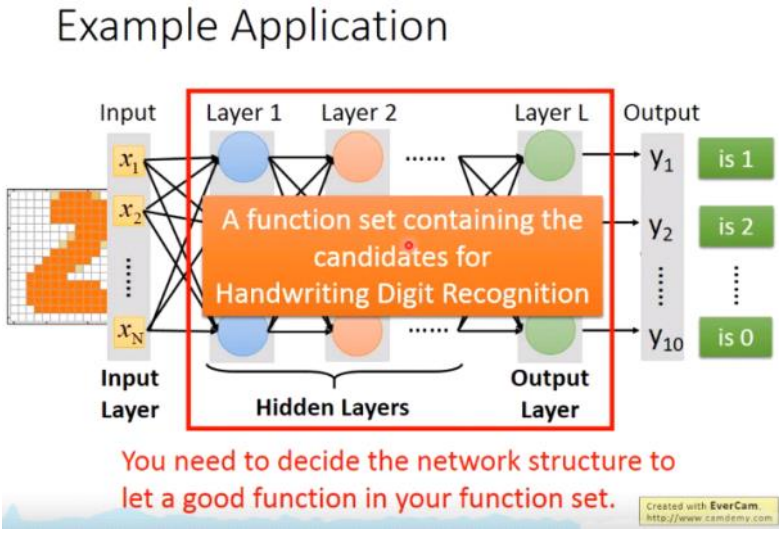


示例：

1 输入向量x：2 中有图黑的地方就是1，没有就是0，共256维。

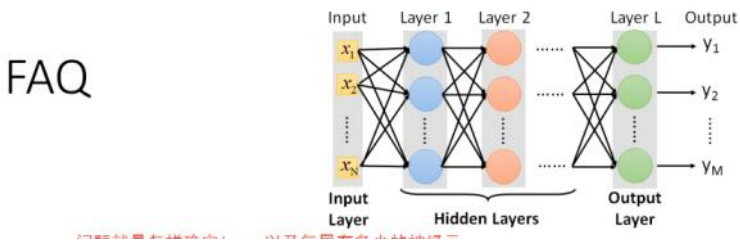
2 output如果是十维的话，代表1,2,3....0的几率，the confidence of a digit。

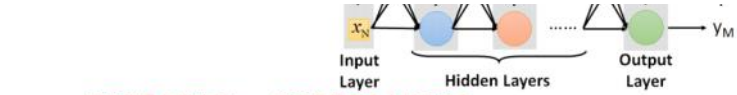
3 需要一个function，将256维度的输入，输出为10维的数字几率。所以，设计network structure（很关键），构建出一个function set，寻找适用于手写上是别的最佳候选function。



有了DeepLearning，不需要再做feature selection/手动transform，而是全部丢进去，但需要解决新问题，即：多少个layer，每个层多少个Neuron，经验+直觉+试错。

如果是语音辨识，图像识别的，则design network





问题就是怎样确定layer以及每层有多少的神经元

- Q: How many layers? How many neurons for each layer?

通常的情况就是不断的尝试，做好提前的函数分析

Trial and Error + Intuition

- Q: Can the structure be automatically determined?
  - E.g. Evolutionary Artificial Neural Networks

这个可以自己确定吗？ 比如阿尔法狗

- Q: Can we design the network structure?

我们也可以按照自己的方式设置，搭建神经网络（如CNN）

Convolutional Neural Network (CNN)

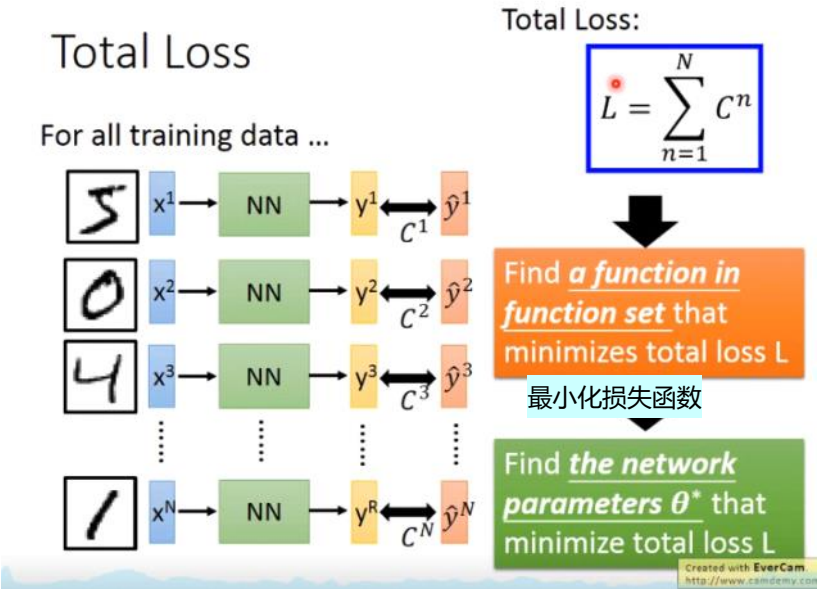
如果是语音辨识，图像识别的，则design network structure，比 feature engeering容易。对人来说，抽取一组好的feature，来做分类或回归更难。不如用design network structure让其自动抽取出一组好的feature。

如果是dl在nlp方面，DeepLearning（仅）可能没有其他的算法好，因为人的文字处理能力较强，可控，可基于先验经验指定规则，而语音却抽象。

能不能自动学network structure？可利用gradient descent，只是方向变复杂了，其他更一般的机器学习方法差不多。

$$C(y,\hat{y}) = -\sum_{i=1}^{10} \hat{y}_i ln y_i$$

单独计算每个样本的交叉熵，然后计算总体的交叉熵



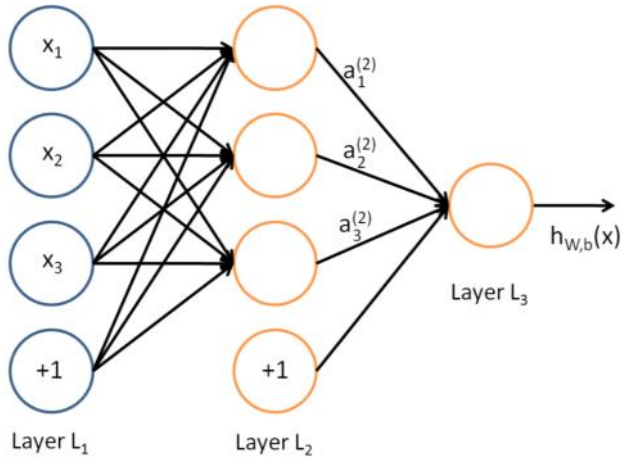
根据梯度下降法进行最优化选择，寻找一组最合适该模型的参数

示例

2020年7月10日 21:41

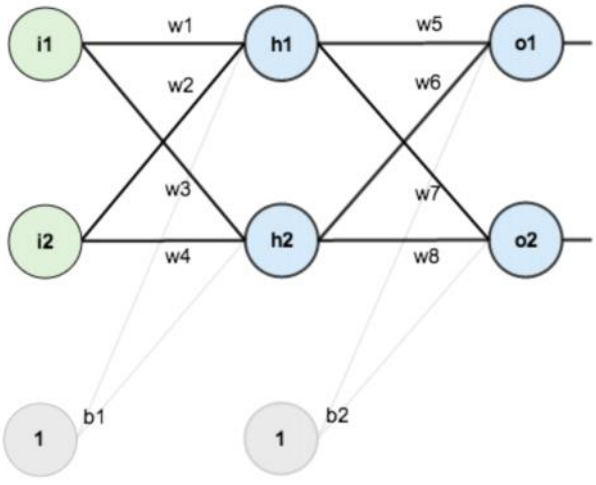
来源：https://www.cnblogs.com/charlotte77/p/5629865.html

说到神经网络，大家看到这个图应该不陌生：



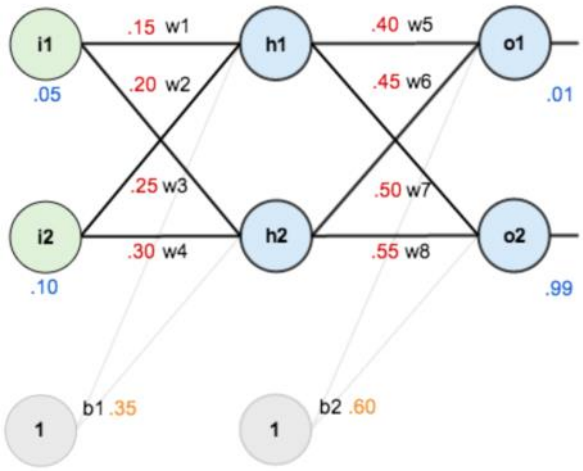
这是典型的三层神经网络的基本构成，Layer L1是输入层，Layer L2是隐含层，Layer L3是隐含层，我们现在手里有一堆数据{x1,x2,x3,...,xn},输出也是一堆数据{y1,y2,y3,...,yn},现在要他们在隐含层做某种变换，让你把数据灌进去后得到你期望的输出。如果你希望你的输出和原始输入一样，那么就是最常见的自编码模型（Auto-Encoder）。可能有人会问，为什么要输入输出都一样呢？有什么用啊？其实应用挺广的，在图像识别，文本分类等等都会用到，我会专门再写一篇Auto-Encoder的文章来说明，包括一些变种之类的。如果你的输出和原始输入不一样，那么就是很常见的人工神经网络了，相当于让原始数据通过一个映射来得到我们想要的输出数据，也就是我们今天要讲的话题。

假设，你有这样一个网络层：



第一层是输入层，包含两个神经元i1，i2，和截距项b1；第二层是隐含层，包含两个神经元h1,h2和截距项b2，第三层是输出o1,o2，每条线上标的wi是层与层之间连接的权重，激活函数我们默认为sigmoid函数。

现在对他们赋上初值，如下图：



其中，输入数据 i1=0.05，i2=0.10;

输出数据 o1=0.01,o2=0.99;

初始权重 w1=0.15,w2=0.20,w3=0.25,w4=0.30;

w5=0.40,w6=0.45,w7=0.50,w8=0.55

目标：给出输入数据i1,i2(0.05和0.10)，使输出尽可能与原始输出o1,o2(0.01和0.99)接近。

Step 1 前向传播

1.输入层----->隐含层:

计算神经元h1的输入加权和:

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

$$net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

神经元h1的输出o1:(此处用到激活函数为sigmoid函数):

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-0.3775}} = 0.593269992$$

同理，可计算出神经元h2的输出o2:

$$out_{h2} = 0.596884378$$

2.隐含层----->输出层:

计算输出层神经元o1和o2的值:

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967$$

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}} = \frac{1}{1+e^{-1.105905967}} = 0.75136507$$

$$out_{o2} = 0.772928465$$

这样前向传播的过程就结束了，我们得到输出值为[0.75136079 ， 0.772928465]，与实际值[0.01 ， 0.99]相差还很远，现在我们对误差进行反向传播，更新权值，重新计算输出。

Step 2 反向传播

1.计算总误差

总误差：(square error)

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

但是有两个输出，所以分别计算o1和o2的误差，总误差为两者之和：

$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2 = \frac{1}{2}(0.01 - 0.75136507)^2 = 0.274811083$$

$$E_{o2} = 0.023560026$$

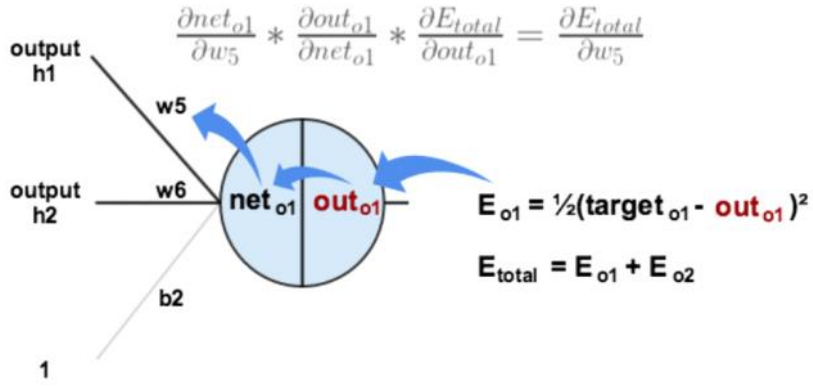
$$E_{total} = E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = 0.298371109$$

2.隐含层---->输出层的权值更新：

以权重参数w5为例，如果我们想知道w5对整体误差产生了多少影响，可以用整体误差对w5求偏导求出：（链式法则）

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

下面的图可以更直观的看清楚误差是怎样反向传播的：



现在我们来分别计算每个式子的值：

计算

$\frac{\partial E_{total}}{\partial out_{o1}}$

：

$$E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o2} - out_{o2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2 * \frac{1}{2}(target_{o1} - out_{o1})^{2-1} * -1 + 0$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = -(target_{o1} - out_{o1}) = -(0.01 - 0.75136507) = 0.74136507$$

计算

$$\frac{\partial out_{o1}}{\partial net_{o1}}$$

：

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}}$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = 0.75136507(1 - 0.75136507) = 0.186815602$$

（这一步实际上就是对sigmoid函数求导，比较简单，可以自己推导一下）

计算

$$\frac{\partial net_{o1}}{\partial w_5}$$

：

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial w_5} = 1 * out_{h1} * w_5^{(1-1)} + 0 + 0 = out_{h1} = 0.593269992$$

最后三者相乘：

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

$$\frac{\partial E_{total}}{\partial w_5} = 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041$$

这样我们就计算出整体误差E(total)对w5的偏导值。

回过头来再看看上面的公式，我们发现：

$$\frac{\partial E_{total}}{\partial w_5} = -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1}) * out_{h1}$$

为了表达方便，用

δ_{o1}

来表示输出层的误差：

$$\delta_{o1} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} = \frac{\partial E_{total}}{\partial net_{o1}}$$

$$\delta_{o1} = -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1})$$

因此，整体误差E(total)对w5的偏导公式可以写成：

$$\frac{\partial E_{total}}{\partial w_5} = \delta_{o1} out_{h1}$$

如果输出层误差计为负的话，也可以写成：

$$\frac{\partial E_{total}}{\partial w_5} = -\delta_{o1} out_{h1}$$

最后我们来更新w5的值：

$$w_5^+ = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.082167041 = 0.35891648$$

（其中，

η

是学习速率，这里我们取0.5）

同理，可更新w6,w7,w8：

$$w_6^+ = 0.408666186$$

$$w_7^+ = 0.511301270$$

$$w_8^+ = 0.561370121$$

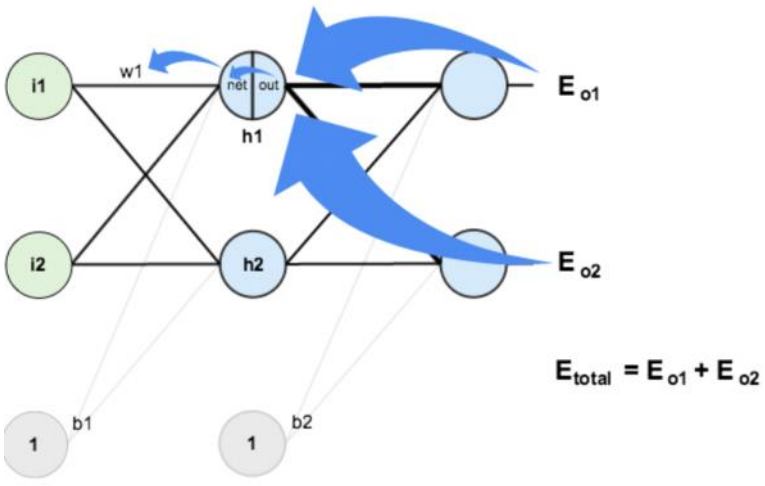
3.隐含层---->隐含层的权值更新：

方法其实与上面说的差不多，但是有个地方需要变一下，在上文计算总误差对w5的偏导时，是从out(o1)---->net(o1)---->w5,但是在隐含层之间的权值更新时，是out(h1)---->net(h1)---->w1,而out(h1)会接受E(o1)和E(o2)两个地方传来的误差，所以这个地方两个都要计算。

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\downarrow$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



计算

$\frac{\partial E_{total}}{\partial out_{h1}}$

：

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

先计算

$\frac{\partial E_{o1}}{\partial out_{h1}}$

：

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}}$$

$$\frac{\partial E_{o1}}{\partial net_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} = 0.74136507 * 0.186815602 = 0.138498562$$

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial out_{h1}} = w_5 = 0.40$$

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}} = 0.138498562 * 0.40 = 0.055399425$$

同理，计算出：

$\frac{\partial E_{o2}}{\partial out_{h1}} = -0.019049119$

两者相加得到总值：

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} = 0.055399425 + -0.019049119 = 0.036350306$$

再计算

$\frac{\partial out_{h1}}{\partial net_{h1}}$

：

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}}$$

$$\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1}) = 0.59326999(1 - 0.59326999) = 0.241300709$$

再计算

$\frac{\partial net_{h1}}{\partial w_1}$

：

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

$$\frac{\partial net_{h1}}{\partial w_1} = i_1 = 0.05$$

最后，三者相乘：

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$$

为了简化公式，用sigma(h1)表示隐含层单元h1的误差：

$$\frac{\partial E_{total}}{\partial w_1} = (\sum_o \frac{\partial E_{total}}{\partial out_o} * \frac{\partial out_o}{\partial net_o} * \frac{\partial net_o}{\partial out_{h1}}) * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = (\sum_o \delta_o * w_{ho}) * out_{h1}(1 - out_{h1}) * i_1$$

$$\frac{\partial E_{total}}{\partial w_1} = \delta_{h1} i_1$$

最后，更新w1的权值：

$$w_1^+ = w_1 - \eta * \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.5 * 0.000438568 = 0.149780716$$

同理，额可更新w2,w3,w4的权值：

$$w_2^+ = 0.19956143$$

$$w_3^+ = 0.24975114$$

$$w_4^+ = 0.29950229$$

这样误差反向传播法就完成了，最后我们再把更新的权值重新计算，不停地迭代，在这个例子中第一次迭代之后，总误差E(total)由0.298371109下降至0.291027924。迭代10000次后，总误差为0.000035085，输出为[0.015912196,0.984065734](原输入为[0.01,0.99]),证明效果还是不错的。

代码示例

```
#coding:utf-8
import random
import math

#
#  参数解释:
#  "pd_"： 偏导的前缀
#  "d_"： 导数的前缀
#  "w_ho"： 隐含层到输出层的权重系数索引
#  "w_ih"： 输入层到隐含层的权重系数的索引

class NeuralNetwork:
    LEARNING_RATE = 0.5

    def __init__(self, num_inputs, num_hidden, num_outputs, hidden_layer_weights = None, hidden_layer_bias = None,
output_layer_weights = None, output_layer_bias = None):
        self.num_inputs = num_inputs

        self.hidden_layer = NeuronLayer(num_hidden, hidden_layer_bias)
        self.output_layer = NeuronLayer(num_outputs, output_layer_bias)

        self.init_weights_from_inputs_to_hidden_layer_neurons(hidden_layer_weights)
```

```
        self.init_weights_from_hidden_layer_neurons_to_output_layer_neurons(output_layer_weights)

    def init_weights_from_inputs_to_hidden_layer_neurons(self, hidden_layer_weights):
        weight_num = 0
        for h in range(len(self.hidden_layer.neurons)):
            for i in range(self.num_inputs):
                if not hidden_layer_weights:
                    self.hidden_layer.neurons[h].weights.append(random.random())
                else:
                    self.hidden_layer.neurons[h].weights.append(hidden_layer_weights[weight_num])
                weight_num += 1

    def init_weights_from_hidden_layer_neurons_to_output_layer_neurons(self, output_layer_weights):
        weight_num = 0
        for o in range(len(self.output_layer.neurons)):
            for h in range(len(self.hidden_layer.neurons)):
                if not output_layer_weights:
                    self.output_layer.neurons[o].weights.append(random.random())
                else:
                    self.output_layer.neurons[o].weights.append(output_layer_weights[weight_num])
                weight_num += 1

    def inspect(self):
        print('------')
        print('* Inputs: {}'.format(self.num_inputs))
        print('------')
        print('Hidden Layer')
        self.hidden_layer.inspect()
        print('------')
        print('* Output Layer')
        self.output_layer.inspect()
        print('------')

    def feed_forward(self, inputs):
        hidden_layer_outputs = self.hidden_layer.feed_forward(inputs)
        return self.output_layer.feed_forward(hidden_layer_outputs)

    def train(self, training_inputs, training_outputs):
        self.feed_forward(training_inputs)

        # 1. 输出神经元的值
        pd_errors_wrt_output_neuron_total_net_input = [0] * len(self.output_layer.neurons)
        for o in range(len(self.output_layer.neurons)):

            # ∂E/∂z_j
            pd_errors_wrt_output_neuron_total_net_input[o] =
self.output_layer.neurons[o].calculate_pd_error_wrt_total_net_input(training_outputs[o])

        # 2. 隐含层神经元的值
        pd_errors_wrt_hidden_neuron_total_net_input = [0] * len(self.hidden_layer.neurons)
        for h in range(len(self.hidden_layer.neurons)):

            # dE/dy_j = Σ ∂E/∂z_j * ∂z/∂y_j = Σ ∂E/∂z_j * w_ij
            d_error_wrt_hidden_neuron_output = 0
            for o in range(len(self.output_layer.neurons)):
                d_error_wrt_hidden_neuron_output += pd_errors_wrt_output_neuron_total_net_input[o] *
self.output_layer.neurons[o].weights[h]

            # ∂E/∂z_j = dE/dy_j * ∂z_j/∂
            pd_errors_wrt_hidden_neuron_total_net_input[h] = d_error_wrt_hidden_neuron_output *
self.hidden_layer.neurons[h].calculate_pd_total_net_input_wrt_input()

        # 3. 更新输出层权重系数
        for o in range(len(self.output_layer.neurons)):
            for w_ho in range(len(self.output_layer.neurons[o].weights)):

                # ∂E_j/∂w_ij = ∂E/∂z_j * ∂z_j/∂w_ij
                pd_error_wrt_weight = pd_errors_wrt_output_neuron_total_net_input[o] *
self.output_layer.neurons[o].calculate_pd_total_net_input_wrt_weight(w_ho)
```

```
            # Δw = α * ∂E_j/∂w_i
            self.output_layer.neurons[o].weights[w_ho] -= self.LEARNING_RATE * pd_error_wrt_weight

        # 4. 更新隐含层的权重系数
        for h in range(len(self.hidden_layer.neurons)):
            for w_ih in range(len(self.hidden_layer.neurons[h].weights)):

                # ∂E_j/∂w_i = ∂E/∂z_j * ∂z_j/∂w_i
                pd_error_wrt_weight = pd_errors_wrt_hidden_neuron_total_net_input[h] *
self.hidden_layer.neurons[h].calculate_pd_total_net_input_wrt_weight(w_ih)

                # Δw = α * ∂E_j/∂w_i
                self.hidden_layer.neurons[h].weights[w_ih] -= self.LEARNING_RATE * pd_error_wrt_weight

    def calculate_total_error(self, training_sets):
        total_error = 0
        for t in range(len(training_sets)):
            training_inputs, training_outputs = training_sets[t]
            self.feed_forward(training_inputs)
            for o in range(len(training_outputs)):
                total_error += self.output_layer.neurons[o].calculate_error(training_outputs[o])
        return total_error

class NeuronLayer:
    def __init__(self, num_neurons, bias):

        # 同一层的神经元共享一个截距项b
        self.bias = bias if bias else random.random()

        self.neurons = []
        for i in range(num_neurons):
            self.neurons.append(Neuron(self.bias))

    def inspect(self):
        print('Neurons:', len(self.neurons))
        for n in range(len(self.neurons)):
            print(' Neuron', n)
            for w in range(len(self.neurons[n].weights)):
                print('  Weight:', self.neurons[n].weights[w])
            print('  Bias:', self.bias)

    def feed_forward(self, inputs):
        outputs = []
        for neuron in self.neurons:
            outputs.append(neuron.calculate_output(inputs))
        return outputs

    def get_outputs(self):
        outputs = []
        for neuron in self.neurons:
            outputs.append(neuron.output)
        return outputs

class Neuron:
    def __init__(self, bias):
        self.bias = bias
        self.weights = []

    def calculate_output(self, inputs):
        self.inputs = inputs
        self.output = self.squash(self.calculate_total_net_input())
        return self.output

    def calculate_total_net_input(self):
        total = 0
        for i in range(len(self.inputs)):
            total += self.inputs[i] * self.weights[i]
        return total + self.bias
```

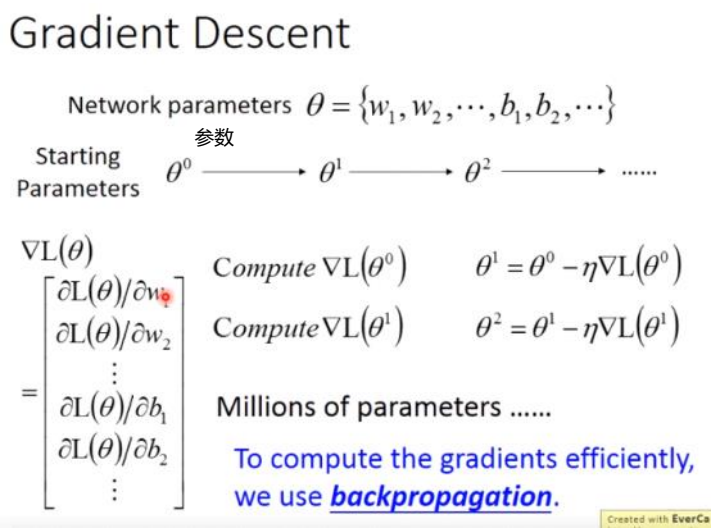
```
    # 激活函数sigmoid
    def squash(self, total_net_input):
        return 1 / (1 + math.exp(-total_net_input))


    def calculate_pd_error_wrt_total_net_input(self, target_output):
        return self.calculate_pd_error_wrt_output(target_output) * self.calculate_pd_total_net_input_wrt_input();

    # 每一个神经元的误差是由平方差公式计算的
    def calculate_error(self, target_output):
        return 0.5 * (target_output - self.output) ** 2


    def calculate_pd_error_wrt_output(self, target_output):
        return -(target_output - self.output)


    def calculate_pd_total_net_input_wrt_input(self):
        return self.output * (1 - self.output)


    def calculate_pd_total_net_input_wrt_weight(self, index):
        return self.inputs[index]


# 文中的例子:

nn = NeuralNetwork(2, 2, 2, hidden_layer_weights=[0.15, 0.2, 0.25, 0.3], hidden_layer_bias=0.35,
output_layer_weights=[0.4, 0.45, 0.5, 0.55], output_layer_bias=0.6)
for i in range(10000):
    nn.train([0.05, 0.1], [0.01, 0.09])
    print(i, round(nn.calculate_total_error([[[0.05, 0.1], [0.01, 0.09]]]), 9))


#另外一个例子，可以把上面的例子注释掉再运行一下:

# training_sets = [
#     [[0, 0], [0]],
#     [[0, 1], [1]],
#     [[1, 0], [1]],
#     [[1, 1], [0]]
# ]

# nn = NeuralNetwork(len(training_sets[0][0]), 5, len(training_sets[0][1]))
# for i in range(10000):
#     training_inputs, training_outputs = random.choice(training_sets)
#     nn.train(training_inputs, training_outputs)
#     print(i, nn.calculate_total_error(training_sets))
```

反向传播BP算法 Backpropagation

2020年7月12日 14:08

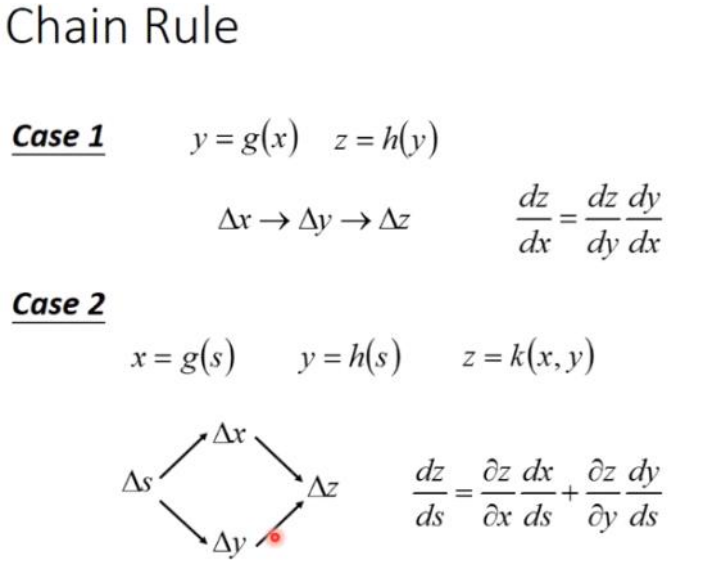
反向传播算法可以说是神经网络最基础也是最重要的知识点。基本上所以的优化算法都是在反向传播算出梯度之后进行改进的。
反向传播算法是一个递归的形式，一层一层的向后传播误差
记住反向传播就是梯度下降的一种方法，最后目的就是得到参数，然后更新参数，每次迭代都要算一次

Backpropagation这个方法其实也是Gradient Descent，只是为了让神经网络有效率地train出来，所以才有了反向传播。



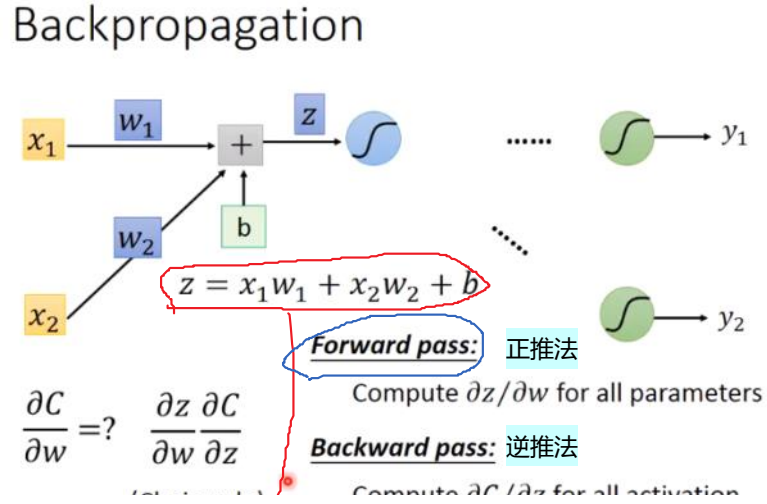
为了计算梯度的效率我们使用反向传播

使用的方法是链式求导法则
考研数学复习过



- 损失函数：定义在单个训练样本上的，单指一个样本的误差
- 代价函数：定义在整个训练集上的，即所有样本的误差的总和的平均
- 总体损失函数：定义在整个训练集上的，即所有样本的误差的总和，也是我们反向传播需要最小化的值

损失函数Loss，代价函数Cost，总体损失函数Total Loss



$\frac{\partial C}{\partial w} = ?$ $\frac{\partial z}{\partial w}\frac{\partial C}{\partial z}$ (Chain rule)

Backward pass: 逆推法

Compute $\partial C/\partial z$ for all activation function inputs z

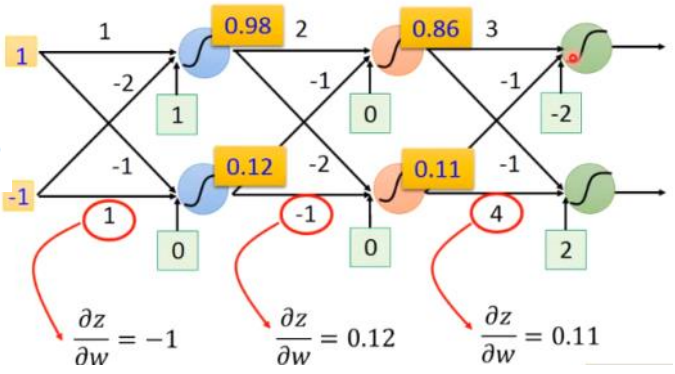
$\partial z/\partial w_1 = ?\ x_1$

$\partial z/\partial w_2 = ?\ x_2$

The value of the input connected by the weight

Backpropagation – Forward pass

Compute $\partial z/\partial w$ for all parameters



Backward pass

根据链式求导法则$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial z}\frac{\partial z}{\partial w}$，其中$z = x_1w_1 + x_2w_2 + b$，因此$\frac{\partial z}{\partial w}$可以很轻松计求解。

Backward pass

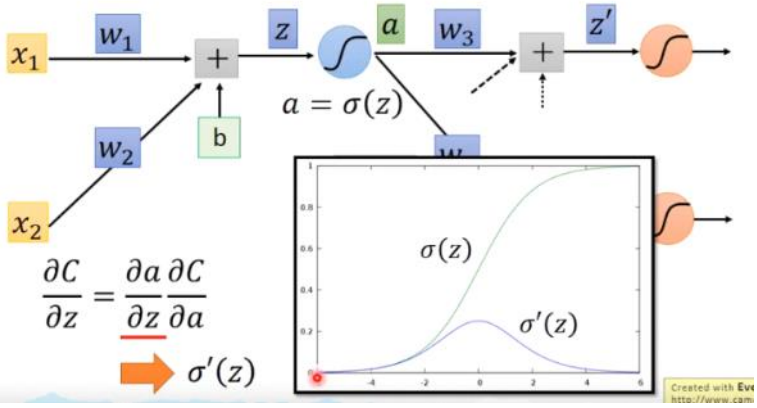
这一部分就非常的困难复杂，因为我们要求$\frac{\partial C}{\partial z}$，我们先用链式法则，把它分解

- 假设神经元function是sigmoid，然后$a = \sigma(z)$
- a通过weight w_3，加上别的value，得到z'
- a还会乘上别的weight w_4，加上别的value，得到z''
- $\frac{\partial C}{\partial z} = \frac{\partial a}{\partial z}\frac{\partial C}{\partial a}$

$\frac{\partial a}{\partial z}$其实就是对sigmoid函数做偏微分

Backpropagation – Backward pass

Compute $\partial C/\partial z$ for all activation function inputs z



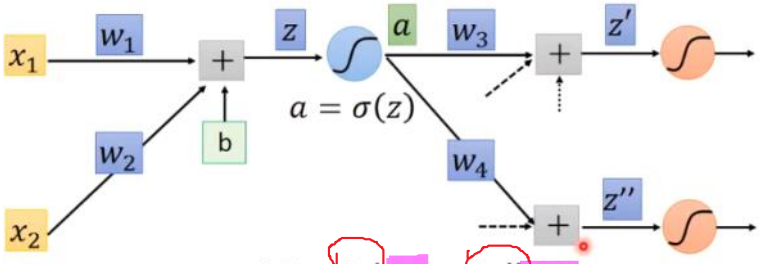
a会通过影响z'和z'' 来影响C，通过上面链式法则中case 2的形式，就会获得上面最后一条式子，如果有很N个neuron，就会加上N个，这里的例子就只假设了两项。

- $\frac{\partial z'}{\partial a} = w_3$
- $\frac{\partial z''}{\partial a} = w_4$

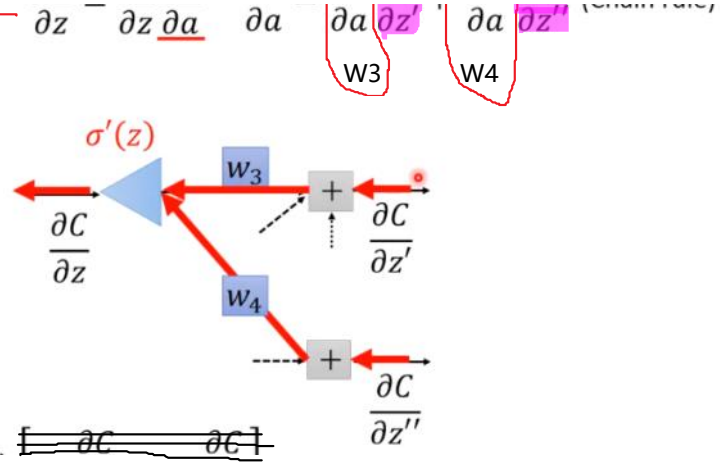
现在我们还是不知道$\frac{\partial C}{\partial z'}$和$\frac{\partial C}{\partial z''}$，不过我们假设已经知道了它们的值，继续运算下去。

Backpropagation – Backward pass

Compute $\partial C/\partial z$ for all activation function inputs z



$$\frac{\partial C}{\partial z} = \sigma'(z)\left[w_3\frac{\partial C}{\partial z'} + w_4\frac{\partial C}{\partial z''}\right]$$

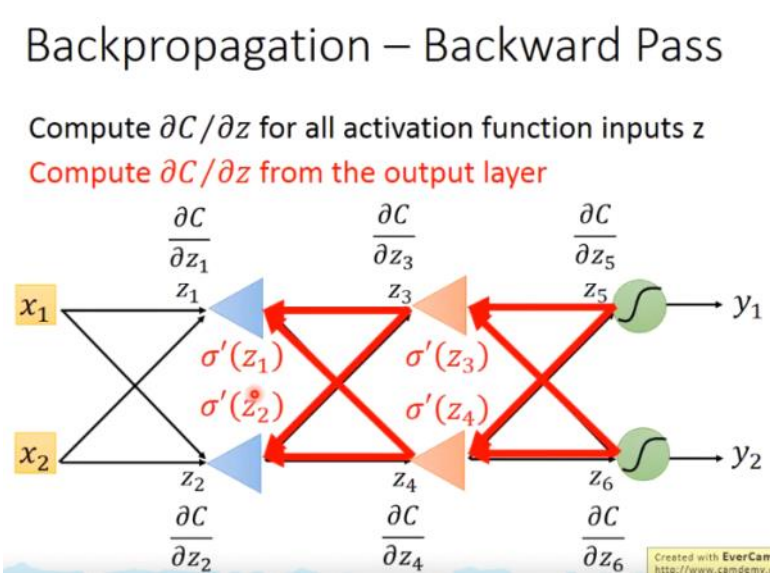


要求z对C的偏微分，需要知道z1和z2对C的偏微分，要知道前面这个，又要知道z3和z4对C的偏微分，以此类推，所以其实我们可以反过来思考，从output层往前计算。

$$\frac{\partial C}{\partial z} = \sigma'(z)\left[w_3\frac{\partial C}{\partial z'} + w_4\frac{\partial C}{\partial z''}\right]$$

$\sigma'(zn)$就像个放大器一样，通过乘以这个可以得z对C的微分值，这个是取决于你的激活函数的，这里只是刚好用上了sigmoid函数

实际上进行backward pass时候和向前传播的计算量差不多



深度学习技巧 Tips for Training DNN

2020年7月12日 15:42

基本步骤：

先定义不同的神经网络接着对其功能进行评估，最后挑选出最优功能的定义

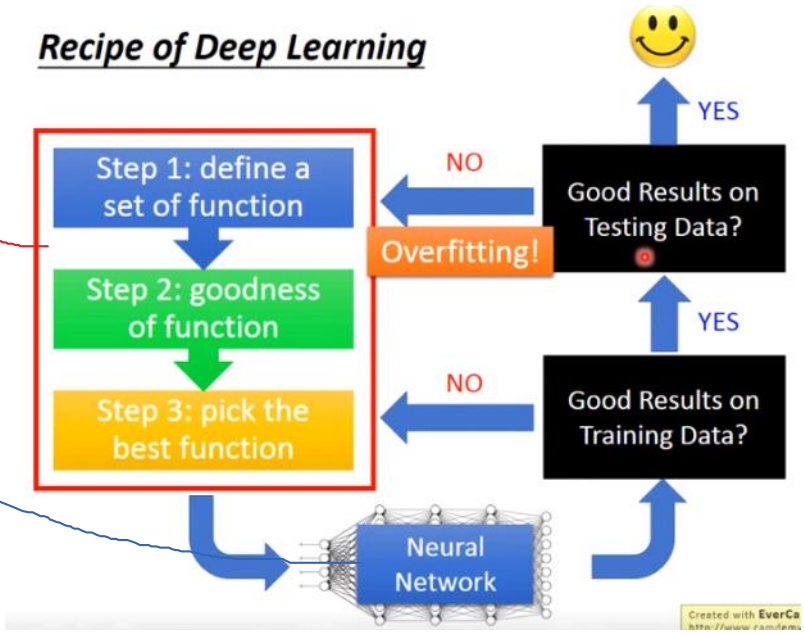
之后会得到一个神经网络

然后去检查再training data上的结果

若结果不好，则重复第一步

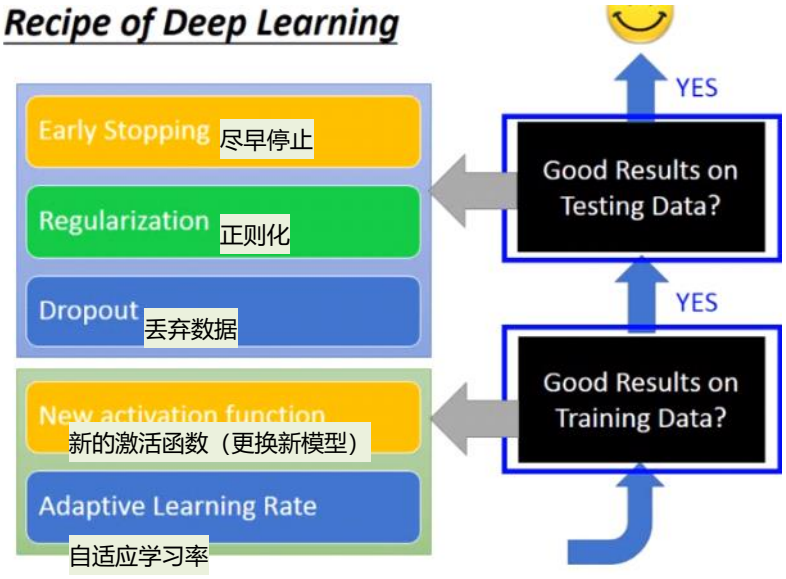
若结果较好，则进行testing data的训练

若testing data结果不好则需要考虑overfitting，结果较好，则可用于实际应用



值得注意的是并不是得到的所有不好的结果都是overfitting，测试集上的表现较差，或许是因为在训练集上的表现也不好；这也不太可能是欠拟合问题，因为欠拟合一般是指模型的参数不够多，所以其能力不能解出这种问题。

在训练集和测试集上较差结果对应的解决方式



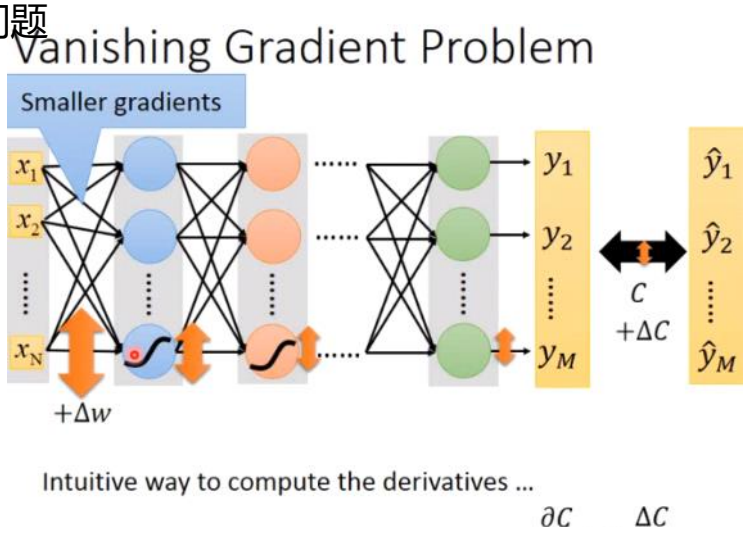
更换新的Activation function的问题

梯度消失问题（vanishing gradient）

后几层梯度很大，所以前几层参数更新几次后，后面的参数就已经达到了收敛（可能是局部最优，因为前几层的参数几乎和初始化的参数一致），所以这不是初始化参数的问题

原因：梯度弥散（Vanishing Gradient Problem）：
在靠近input的部分梯度较小，在远离input的地方，梯度较大。因此，靠近input的参数更新慢、收敛也慢，远离input的参数更新快、收敛得也很早。

为什么会有这个现象发生呢？



梯度较大。因此，靠近input的参数更新慢、收敛也慢，远离input的参数更新快、收敛得也很早。

为什么会有这个现象发生呢？

如果我们把BP算法的式子写出来，就会发现：用sigmoid作为激活函数就是会出现这样的问题。

直觉上想也可以理解：某个参数的下降梯度取决于损失函数对参数的偏导，而偏导的含义是：当该参数变化很小一点，对应的损失函数向相应变化的幅度。所以我们假设对第一层的某个参数加上△w，

看它对损失函数值的影响。那么，当△w经过一层sigmoid函数激活时，它的影响就会衰减一次。（sigmoid会将负无穷到正无穷的值都压缩到(-1,1)区间）

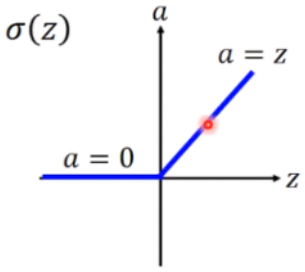
于是，怎么解决这个问题呢：换一个不会逐层消除掉增量的激活函数，比如ReLU.

Intuitive way to compute the derivatives ...

$$\frac{\partial C}{\partial w} =? \frac{\Delta C}{\Delta w}$$

ReLU

- Rectified Linear Unit (ReLU)



[Xavier Glorot, AISTATS'11]
[Andrew L. Maas, ICML'13]
[Kaiming He, arXiv'15]

Reason:

计算速度快

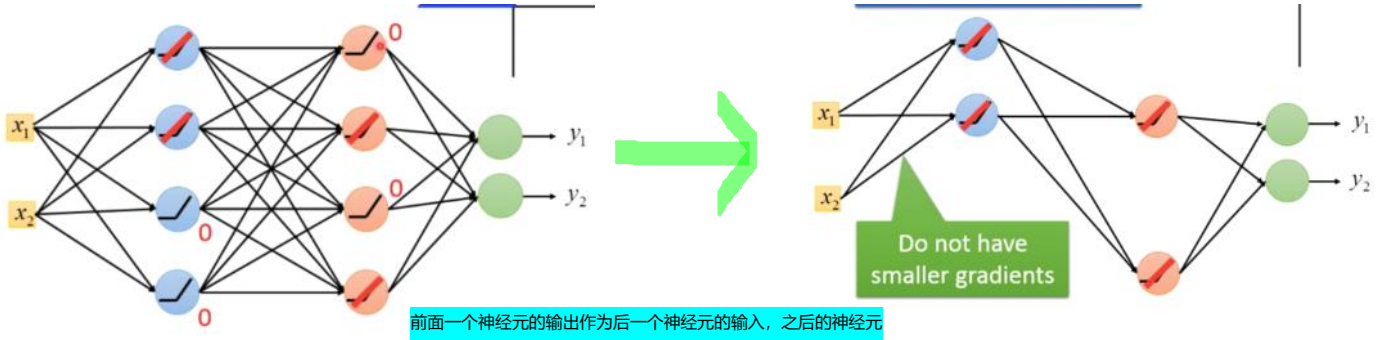
1. Fast to compute

生物学原因

2. Biological reason
3. Infinite sigmoid with different biases
4. Vanishing gradient problem

那ReLU具体怎么解决梯度弥散问题呢？

如下图，假设图中网络的激活函数都是ReLU。而网络中必然有一些神经元对应的输出会是0，它们对于网络的贡献也是零，故可以逻辑上直接从网络中删除，这样我们就获得了一个“更瘦长”的线性网络（输出等于输入）。



前面一个神经元的输出作为后一个神经元的输入，之后的神经元对输入数据进行操作，若干输入数据一直都是无效数据那么这个神经元会一直不被使用，除非它前面的神经元输出发生改变

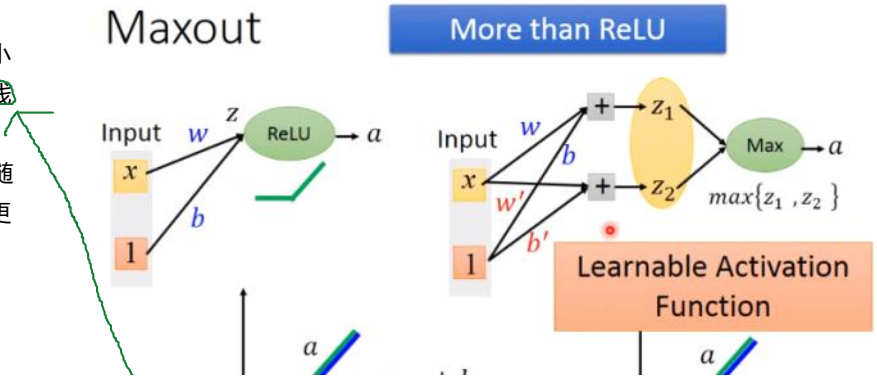
实际上并不是这样，我们只是一次只更新与输出非零的神经元相连的权重而已，当权重改变后，下一轮训练时可能输出为0的神经元就发生变化了，也可以看作是网络架构发生了变化。从这个角度来说，使用ReLU训练出来的神经网络依然是一个非线性函数。

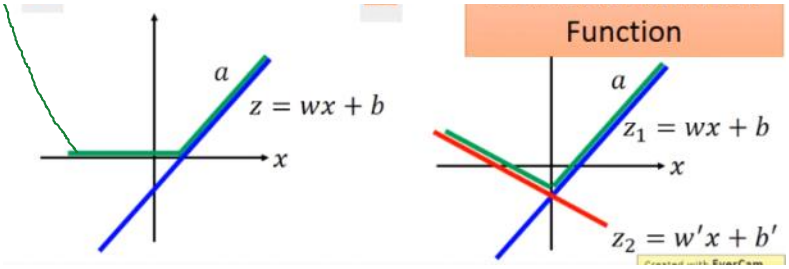
但是ReLU函数依然是存在缺点的，比如**“神经元死亡”**的问题。所谓神经元死亡就是这个神经元永远不会再被激活了，相当于这一层就少了一个神经元。这个问题的产生也很直观，以上面第一张图中第二层的第三个输出为0的神经元来说，首先在当前轮次中与之相连的权重都是不会更新的，而与之相连的第一层的两个输出为0的神经元的输出也会保持为0，因为这两个神经元与输入相连的权重并没有发生变化。因此，第三层的这个神经元要想被再次激活，只能寄希望于第一层前两个神经元的输出发生较大变化，但由于与这两个神经元连接的权重可能只是随机初始化的比较小的值，因此第三层的这个神经元要想被再次激活是很困难的。

综上所述，改进ReLU的思路应该是让z<0时输出不为0，而是一个较小的数。

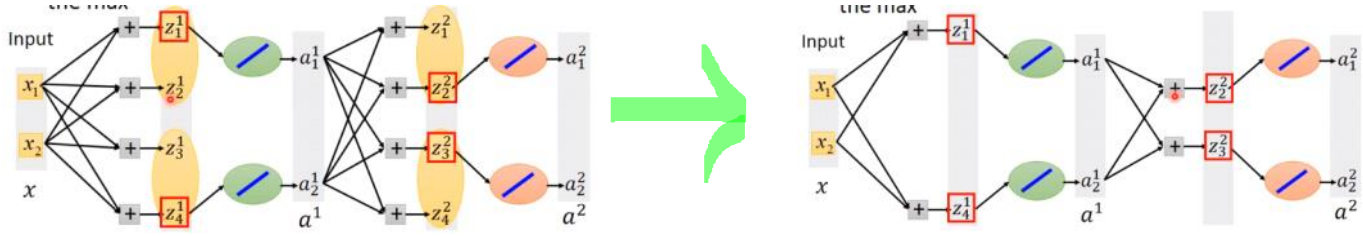
某个神经元的激活函数实际上就是其组内几个小神经元对应的线性函数的最大值，图中绿色的线就是激活函数的图像。

显然，maxout可以得到ReLU激活函数，而且随着一组内神经元个数的增多，还可以得到形式更复杂的激活函数。





假设一个神经元接受的数据又好几组，最后只选择一个最大的数据进行操作，那么未被选择的数据实际相当于对模型训练结果没有任何影响，类似于上面的神经元死亡

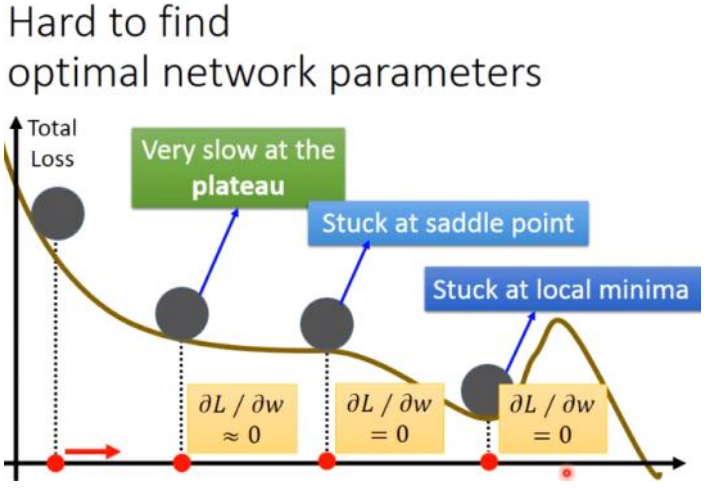


自适应学习率的问题adaptive leaning rate

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sqrt{\sum_{i=0}^{t}(g^i)^2}} g^t$$

adagrad的含义是：对于梯度一般比较小的参数，我们给它较大的学习率；对于梯度一般比较大的参数，我们给它较小的学习率。

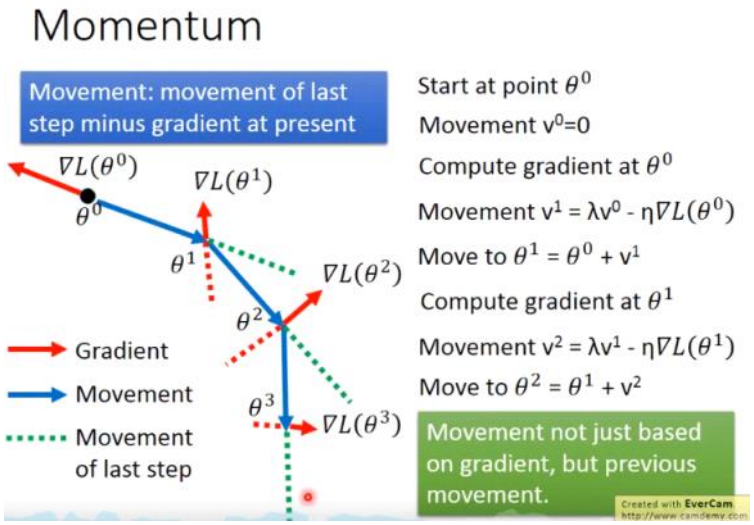
当小球从高处滑下的时候，可能不会因为一个小的凹槽停下来，因为其之前经历了很陡峭的斜坡，其有一个较大的动量，因此会倾向于继续上坡直到动量为0，把这样的机制引入梯度下降法中就可以以一定概率逃离平坦点、鞍点或局部最优点。



下一步要走的方向是梯度的方向和惯性的方向的矢量和方向

在什么地方停止取决于两个向量的矢量和为0

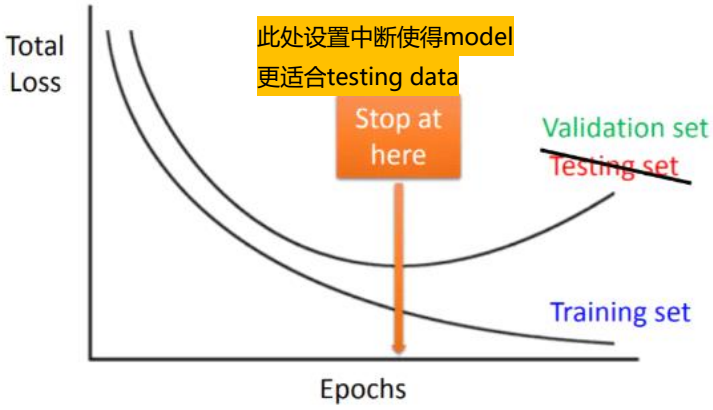
每次计算动量时，就是在之前动量上乘以一个系数这个系数在0到1之间，起到衰减作用，因为上一步已经沿该方向走了一段距离，会消耗部分动量，然后再减去学习率乘以梯度。



以上是当training data出现问题时的解决方法，以下是当testing data出现问题时的解决方法

Early stopping

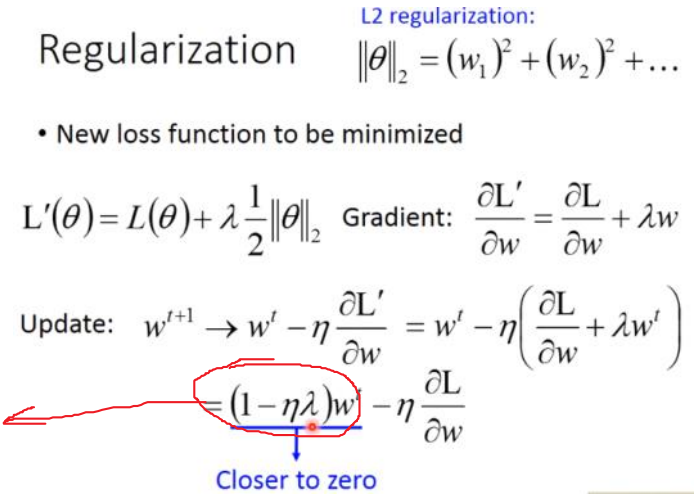
当learning rate的设置合理，那么total loss一般会越来越小，但是当learning rate设置存在问题时loss可能会变大；
但是由于training data的局限性或许在training data中比较合适的learning rate或许会在testing data中出现问题



Regularization

那就是在正则化中通常是不考虑偏置的，这是因为正则化是为了让学得的函数尽可能“平滑”（smooth），而偏置只是将函数上下平移，并不影响函数的平滑程度。
在神经网络的损失函数中加入正则项并使用梯度下降法可以得到上图结果，可以看出，相比不加正则项，梯度下降法只是在每次运行过程中先进行了一次**“权重衰减”**，即给当前训练的参数乘上一个略小于1的数值。于是算法倾向于得到各个权重都不太大的一个平滑函数。

每次update之前使得参数较上一次减小一点（乘一个小于1但是接近1的数）



正则化例子

L2给当前训练的参数乘上一个略小于1的数值，而L1是给当前训练参数改变一个固定的较小的值nλ，当参数为正，减去nλ，当参数为负，加上nλ。

当w很大时L2会下降的很快

Regularization

L1 regularization:

$$\|\theta\|_1 = |w_1| + |w_2| + \dots$$

- New loss function to be minimized

$$L'(\theta) = L(\theta) + \lambda \frac{1}{2}\|\theta\|_1 \qquad \frac{\partial L'}{\partial w} = \frac{\partial L}{\partial w} + \lambda \,\mathrm{sgn}(w)$$

Update:

$$w^{t+1} \rightarrow w^t - \eta \frac{\partial L'}{\partial w} = w^t - \eta\left(\frac{\partial L}{\partial w} + \lambda \,\mathrm{sgn}(w^t)\right)$$

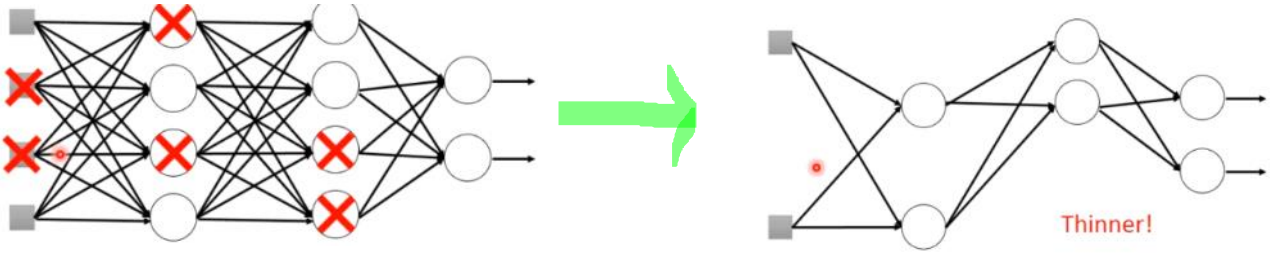
$$= w^t - \eta \frac{\partial L}{\partial w} - \eta\lambda \,\mathrm{sgn}(w^t)$$ Always delete

$$= (1 - \eta\lambda) w^t - \eta \frac{\partial L}{\partial w}$$ L2

Dropout

具体操作：

Dropout的思路很简单，就是在训练过程中每一层都drop一些神经元，具体实现过程是让每个神经元都有p的概率被drop。



直观上说，我们每次用一个比较简单的网络（总网络的一个子集）进行训练，这就相当于把在这部分权重训练到最好，让它们有自己上战场也能顶得住的觉悟，每部分权重都这样训练，最后效果自然会强大一些。

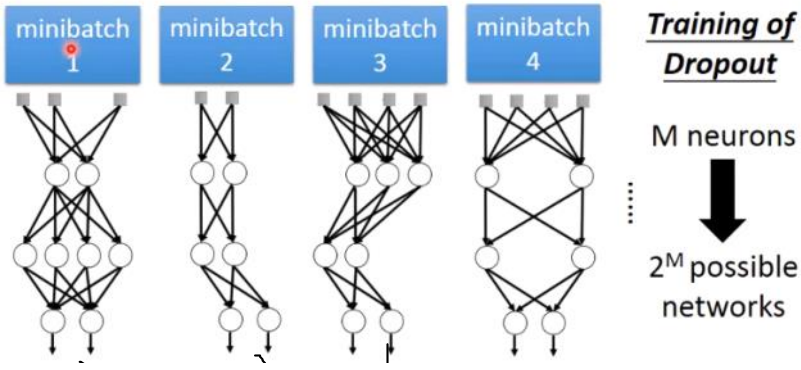
理论上来说，**可以把Dropout视为一种很多网络架构训练结果的ensemble**，因此相比单一结构的神经网络表现会有所提升。

No dropout

- If the dropout rate at training is p%, all the weights times 1-p%
- Assume that the dropout rate is 50%. If a weight $w = 1$ by training, set $w = 0.5$ for testing.

同一组数据，在进行不同的训练数据后通过dropout丢掉一部分，剩余数据结果进行整合出最终结果

Dropout is a kind of ensemble.

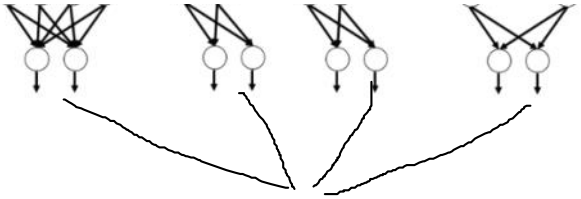


直观上说，我们每次用一个比较简单的网络（总网络的一个子集）进行训练，这就相当于把在这部分权重训练到最好，让它们有自己上战场也能顶得住的觉悟，每部分权重都这样训练，最后效果自然会强大一些。

理论上来说，**可以把Dropout视为一种很多网络架构训练结果的ensemble**，因此相比单一结构

果自然会强大一些。

理论上来说，**可以把Dropout视为一种很多网络架构训练结果的ensemble**，因此相比单一结构的神经网络表现会有所提升。



networks

average得到最终结果