Week1

本周完成

1.根据视频了解学习了深度学习的一些概念（回归算法，梯度下降优化，损失函数，神经网络）

2.根据教程实现并了解liner和softmax的回归算法的代码具体实现过程。

3.学习了transforms和tensorboard的使用

下周计划

1.学习多层感知机与卷积神经网络

2.阅读一篇论文

二分分类(Binary Classification)

  逻辑回归是一个用于**二分类(binary classification)**的算法。

首先我们从一个问题开始说起，这里有一个二分类问题的例子，假如你有一张猫猫图片作为输入如果识别这张图片为猫，则输出标签**1**作为结果；如果识别出不是猫，那么输出标签**0**作为结果。

图片在计算机中表示：

保存一张图片，需要保存三个矩阵，它们分别对应图片中的红、绿、蓝三种颜色通道，如果你的图片大小为64x64像素，那么你就有三个规模为64x64的矩阵，分别对应图片中红、绿、蓝三种像素的强度值。

为了把这些像素值转换为特征向量x,我们要把所有的像素按照一定顺序都取出来，假设用**上图数据**

那么x的维度就是$(n_x, 1)$,其中$n_x = 64 \times 64 \times 3$

我们的最终目的习得一个分类器，它以图片的特征向量作为输入，然后预测输出结果y为1还是0.

逻辑回归(Logistic Regression)

现在我们有了输入数据x，想要得到输出数据y。那么就得有一个算法能够输出预测。

假定为$\bar{y}$为对y的预测，即可能性。$\bar{y}$的取值范围就应该在$0-1$之间.

如果我们设计的算法为$\bar{y}=w^tx+b, w_t$为我们自行设计的权重，b为偏差值。此时即为一个关于**x的线性函数**，实际上这是你在做**线性回归**时所用到的。$\bar{y}$的值可能并不在$0-1$之间。所以我们要用**sigmoid**函数将其转化在我们的需求之间

```
import numpy as np
def sig(x):
    return 1/(1 + np.exp(-x))
```

逻辑回归的代价函数（Logistic Regression Cost Function）

由于对参数w，b的未知，我们需要代价函数来训练获得w，b

$$\hat{y}^{(i)} = \sigma(w^T x^{(i)} + b), \text{where } \sigma(z) = \frac{1}{1+e^{-z}}$$

$$\text{Given } \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}, \text{ want } \hat{y}^{(i)} \approx y^{(i)}.$$

逻辑回归的输出函数

损失函数又叫做误差函数，用来衡量算法的运行情况 $L(\bar{y}, y)$

我们通过这个称为L的损失函数，来衡量预测输出值和实际值有多接近。
一般我们用
预测值和实际值的平方差或者它们**平方差的一半**，但是**通常在逻辑回归中我们不这么做**，因为当我们在学习逻辑回归参数的时候，会发现我们的优化目标不是凸优化，只能找到多个局部最优值，梯度下降法很可能找不到全局最优值，虽然平方差是一个不错的损失函数，但是我们在逻辑回归模型中会定义另外一个损失函数。

逻辑回归中用到的损失函数

$$L(\hat{y}, y) = -ylog(\hat{y}) - (1-y)log(1-\hat{y})$$

因为我们y取值仅仅为1和0，并且我们通过sigmoid函数让$\hat{y}$处于0-1之间。我们可以通过调整$\bar{y}$来让我们的损失尽可能的小。

但L是单个样本的损失函数，我们想要知道在全部训练样本上的总体代价。即可记为

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^i, y^i) = -y^i log(\hat{y}^i) - (1-y^i) log(1-\hat{y}^i)$$

以在训练逻辑回归模型时候，我们需要找到合适的w和b，来让代价函数J的总代价降到最低。

```
def Costcompute(w, b, X, Y):
    """
    以处理图片为列子
    Arguments:
    w -- 权重矩阵 (num_px * num_px * 3, 1)
    b -- 偏差值
    X -- 样本矩阵 (num_px * num_px * 3, number of examples)
    Y -- 样本对应判断矩阵 (0，1组成) (1, number of examples)
    """

    m = X.shape[1]  #获取数量
    A = sigmoid(np.dot(w.T, X) + b)      # 用sigmoid归1
    cost = -(1.0 / m) * np.sum(Y * np.log(A) + (1 - Y) * np.log(
    cost = np.squeeze(cost)
    return cost
```

梯度下降法（Gradient Descent）

由于我们的代价函数为凸函数（国外教材与国内凹凸性定义相反）

我们可以通过使用**梯度**不断**迭代**的方法来快速达到最优点（函数在该点处沿着该方向（此梯度的方向）变化最快，变化率最大），也称为优化算法。

$$w := w - \alpha \frac{\partial J(w\ ,b)}{\partial w}$$

$$b := b - \alpha \frac{\partial J(w\ ,b)}{\partial b}$$

α代表学习率，一般为常量，用来控制步长。$\frac{\partial J(w,b)}{\partial w}$是对w的偏导，$\frac{\partial J(w,b)}{\partial b}$是对b的偏导。

由于使用梯度下降在整个训练集上每次都要对训练集求导，计算量太大（李沐），所以可以用随机采样来近似损失。小批量随机梯度下降是深度学习的一般求解方法。

在pytorch中可以设置自动计算梯度。

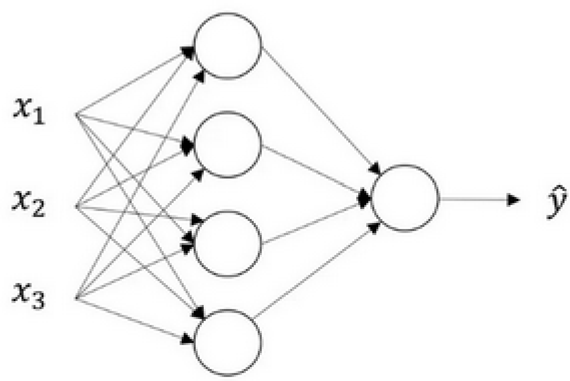
```
#使用pytorch与封装好的库
import numpy as np
import torch
from torch import nn

net = nn.Sequential(nn.Linear(2, 1))#设置线性模型，输入2个，输出1个
net[0].weight.data.normal_(0, 0.01) #随机数据
net[0].bias.data.fill_(0)  #填充
loss = nn.MSELoss() #设置均方误差
trainer = torch.optim.SGD(net.parameters(), lr=0.03) #设置梯度优化
num_epochs = 3
for epoch in range(num_epochs):
    for X, y in data_iter:
        l = loss(net(X) ,y)
        trainer.zero_grad() #梯度设置为0
        l.backward()  #计算
        trainer.step() #更新
    l = loss(net(features), labels)
    print(f'epoch {epoch + 1}, loss {l:f}')
```

```
#传统手写派
def Costcompute(w, b, X, Y):
```

```
    """
    以处理图片为列子
    Arguments:
    w -- 权重矩阵 (num_px * num_px * 3, 1)
    b -- 偏差值
    X -- 样本矩阵 (num_px * num_px * 3, number of examples)
    Y -- 样本对应判断矩阵 (0，1组成) (1, number of examples)
    """
    m = X.shape[1]  #获取数量
    A = sigmoid(np.dot(w.T, X) + b)      # 用sigmoid归1
    cost = -(1.0 / m) * np.sum(Y * np.log(A) + (1 - Y) * np.log(
      dw = (1.0 / m) * np.dot(X, (A - Y).T)
    db = (1.0 / m) * np.sum(A - Y)
    grads = {"dw": dw,
             "db": db}
    return cost,grads # 返回值增加了对d，w的偏导
def optimize(w, b, X, Y, num_iterations, learning_rate):
    """
    使用梯度下降来优化
    Arguments:
    w -- 权重矩阵 (num_px * num_px * 3, 1)
    b -- 偏差值
    X -- 样本矩阵 (num_px * num_px * 3, number of examples)
    Y -- 样本对应判断矩阵 (0，1组成) (1, number of examples)
    num_iterations -- 优化次数
    learning_rate -- 学习率
    """
    costs = []
    for i in range(num_iterations):
        grads, cost = Costcompute(w, b, X, Y)
        dw = grads["dw"]
        db = grads["db"]
        w = w - learning_rate * dw  #迭代更新
        b = b - learning_rate * db
        # Record the costs
        if i % 100 == 0:
```

```
            costs.append(cost)  #记录
    params = {"w": w,
              "b": b}
    grads = {"dw": dw,
             "db": db}
    return params, grads, costs
```

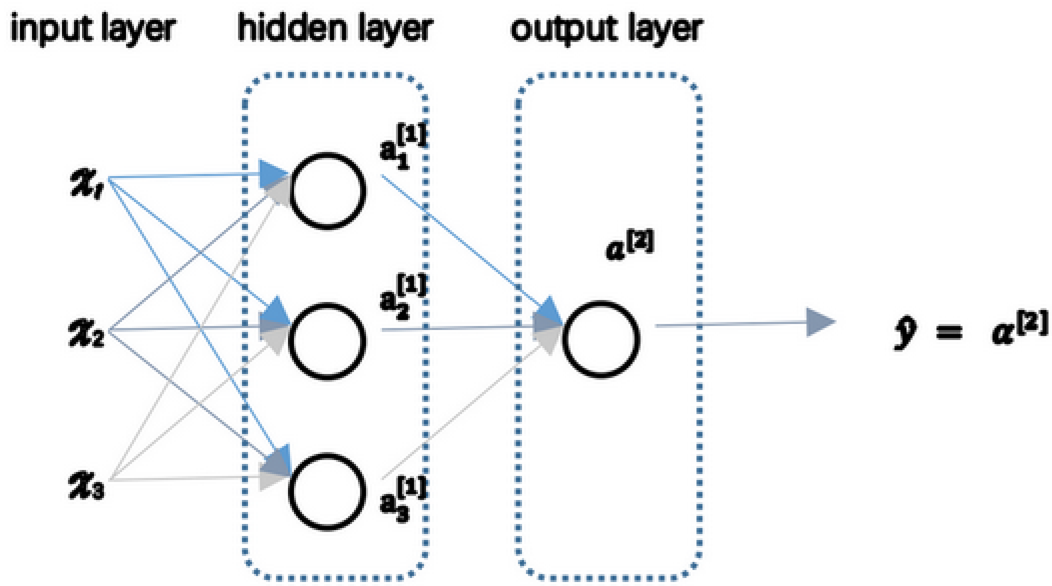
神经网络表示（Neural Network Representation）



本例中的神经网络只包含一个隐藏层

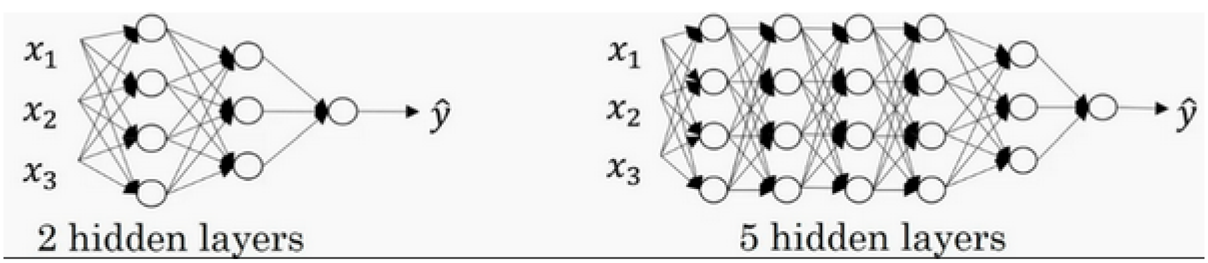
输入特征x_1、x_2、x_3，它们被竖直地堆叠起来，这叫做神经网络的**输入层**。它包含了神经网络的输入；然后这里有另外一层我们称之为**隐藏层**。最后一层只由一个结点构成，而这个只有一个结点的层被称为**输出层**，它负责产生预测值。

隐藏层的含义是在训练集中，这些中间结点的准确值我们是不知道到的，也就是说你看不见它们在训练集中应具有的值。
若我们用向量
x表示输入特征。这里有个可代替的记号$a^{[0]}$可以用来表示输入特征,它意味着网络中不同层的值会传递到它们后面的层中.下一层即隐藏层也同样会产生一些激活值。在本例中，我们有四个结点或者单元，或者称为四个隐藏层单元。最后输出层将产生某个数值a，它只是一个单独的实数.
这个例子，只能叫做一个两层的神经网络,原因是当我们计算网络的层数时，输入层是不算入总层数内，所以隐藏层是第一层，输出层是第二层。第二个惯例是我们将输入层称为第零层，所以在技术上，这仍然是一个三层的神经网络，因为这里有输入层、隐藏层，还有输出层。



深层神经网络（Deep L-layer neural network）

神经网络的层数是这么定义的：**从左到右，由0开始定义**，比如上边右图，x1、x2、x3，这层是第0层，这层左边的隐藏层是第1层，由此类推。如下图左边是两个隐藏层的神经网络，右边是5个隐藏层的神经网络。



深度神经网络能解决好多问题，其实并不需要很大的神经网络，但是得有深度，得有比较多的隐藏层.

假设在建一个人脸识别或是人脸检测系统，深度神经网络所做的事就是，当你输入一张脸部的照片，然后你可以把**深度神经网络的第一层**，当成一个**特征探测器或者边缘探测器**。在这个例子里，我会建一个大概有20个隐藏单元的深度神经网络，是怎么针对这张图计算的。隐藏单元就是这些图里这些小方块（第一张大图），举个例子，这个小方块（第一行第一列）就是一个隐藏单元，它会去找这张照片里“|”边缘的方向。那么这个隐藏单元（第四行第四列），可能是在找（“—”）水平向的边缘在哪里。你可以先把神经网络的第一层当作看图，然后去找这张照片的各个边缘。我们可以把照片里组成边缘的像素们放在一起看，然后它可以把被探测到的边缘组合成面部的不同部分（第二张大图）。比如说，

可能有一个神经元会去找眼睛的部分，另外还有别的在找鼻子的部分，然后把这许多的边缘结合在一起，就可以开始检测人脸的不同部分。最后再把这些部分放在一起，比如鼻子眼睛下巴，就可以识别或是探测不同的人脸（第三张大图）。（吴恩达）

深度神经网络的这许多隐藏层中，较早的前几层能学习一些低层次的简单特征，等到后几层，就能把简单的特征结合起来，去探测更加复杂的东西。

激活函数（Activation functions）

上面用过**sigmoid**激活函数，但是，有时其他的激活函数效果会更好。
更通常的情况下，使用不同的函数，比如除了
sigmoid函数以外的非线性函数。**tanh**函数或者双曲正切函数是总体上都优于**sigmoid**函数的激活函数。

tanh函数值域是位于+1和-1之间。 $a = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

tanh函数是**sigmoid**的向下平移和伸缩后的结果。对它进行了变形后，穿过了点$(0, 0)$，并且值域介于+1和-1之间。
结果表明，如果在隐藏层上使用函数
tanh 效果总是优于**sigmoid**函数。因为函数值域在-1和+1的激活函数，其均值是更接近零均值的。在训练一个算法模型时，如果使用**tanh**函数代替**sigmoid**函数中心化数据，使得数据的平均值更接近0而不是0.5。但有一个例外：在二分类的问题中，对于输出层，因为的值是0或1，所以想让的数值介于0和1之间，而不是在-1和+1之间。所以需要使用**sigmoid**激活函数。

这有一些选择激活函数的**经验法则**（吴恩达）：

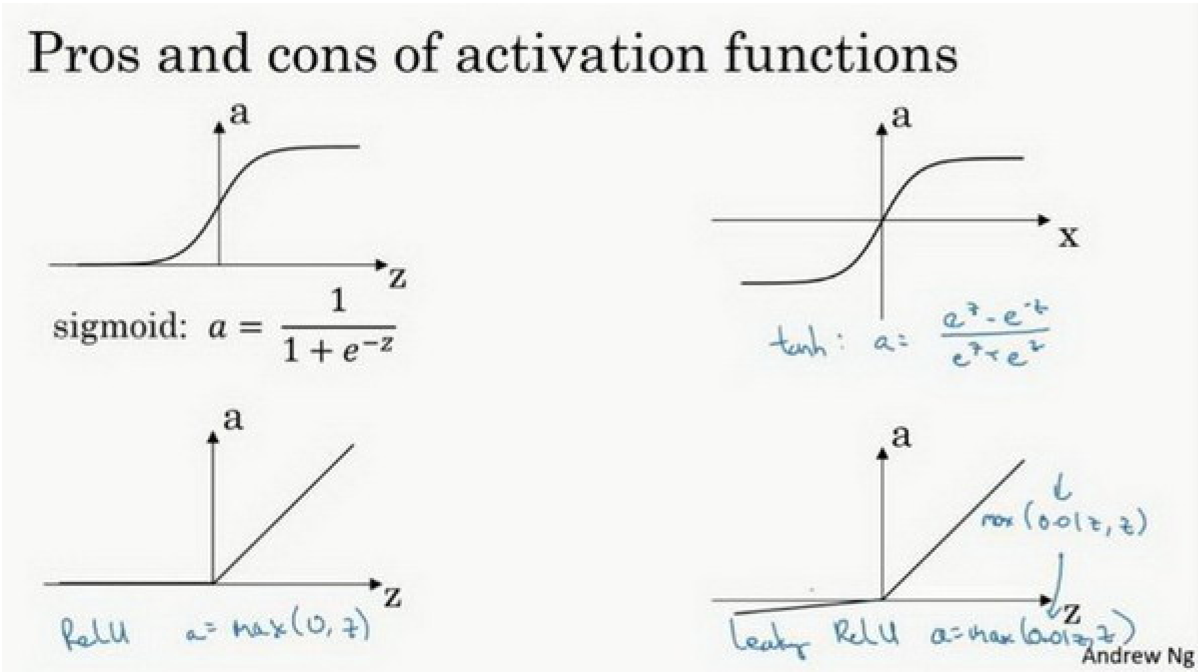
如果输出是0、1值（二分类问题），则输出层选择**sigmoid**函数，然后其它的所有单元都选择**Relu**函数。

这是很多激活函数的默认选择，如果在隐藏层上不确定使用哪个激活函数，那么通常会使用**Relu**激活函数。有时，也会使用**tanh**激活函数，但**Relu**的一个优点是：当是负值的时候，导数等于0。

这里也有另一个版本的**Relu**被称为**Leaky Relu**。

当是负值时，这个函数的值不是等于0，而是轻微的倾斜，如图。

这个函数通常比**Relu**激活函数效果要好，尽管在实际中**Leaky ReLu**使用的并不多。



随机初始化（Random+Initialization）

训练神经网络时，权重随机初始化是很重要的。对于逻辑回归，把权重初始化为0当然也是可以的。但是对于一个神经网络，如果你把权重或者参数都初始化为0，那么梯度下降将不会起作用。

如果初始化成0，由于所有的隐含单元都是对称的，无论你运行梯度下降多久，他们一直计算同样的函数。这没有任何帮助，因为你想要两个不同的隐含单元计算不同的函数，这个问题的解决方法就是随机初始化参数。

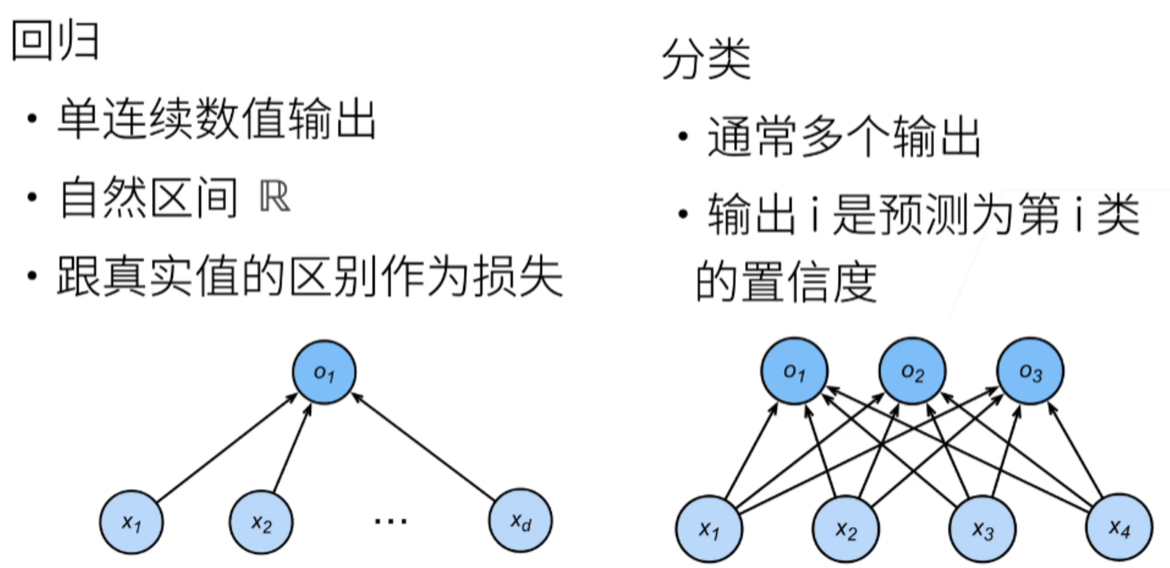
通常倾向于初始化为很小的随机数。因为如果你用**tanh**或者**Sigmoid**激活函数，或者说只在输出层有一个**Sigmoid.**如果W(权重)很大**，**输出**z**就会很大或者很小**，**地方梯度很小也就意味着梯度下降会很慢。

$$W = np.random.randn(2,2) * 0.01$$

事实上有时有比0.01更好的常数，训练一个只有一层隐藏层的网络时（这是相对浅的神经网络，没有太多的隐藏层），设为0.01可能也可以。但当训练一个非常非常深的神经网络，可能要试试0.01以外的常数。但是无论如何它通常都会是个相对小的数。

Softmax回归

实际上仍然是分类问题。



假设有一个图像问题需要判别图像属于类别“猫”“鸡”和“狗”中的哪一个。可以使用

独热编码（one-hot encoding）独热编码是一个向量，它的分量和类别一样多。 类别对应的分量设置为1，其他所有分量设置为0。

在上面这个例子中，标签y是一个三维向量，其中$(1,0,0)$对应猫，$(0,1,0)$对应鸡，$(0,0,1)$对应狗

y ∈ {(1, 0, 0),(0, 1, 0),(0, 0, 1)}

为了估计所有可能类别的条件概率，我们需要一个有多个输出的模型，每个类别对应一个输出。

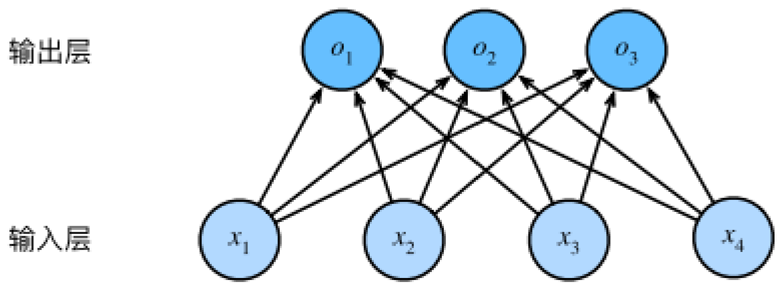
在上例子中，由于我们有4个特征和3个可能的输出类别，我们将需要12个标量来表示权重（带下标
的w），3个标量来表示偏置（带下标的b）。下面我们为每个输入计算三个未规范化的预测（logit）：o1、o2和o3。用向量表示即为

$o = Wx + b$

$o_1 = x_1w_{11} + x_2w_{12} + x_3w_{13} + x_4w_{14} + b_1$

$o_2 = x_1w_{21} + x_2w_{22} + x_3w_{23} + x_4w_{24} + b_2$

$o_3 = x_1w_{31} + x_2w_{32} + x_3w_{33} + x_4w_{34} + b_3$



模型的输出y^j可以视为属于类j的概率，然后选择具有最大输出值的类别$argmax_j y_j$作为我们的预
测。例如，如果
y^1、y^2和y^3分别为0.1、0.8和0.1，那么我们预测的类别是2，在上例子中代表“鸡”

尽管softmax是一个非线性函数，但softmax回归的输出仍然由输入特征的仿射变换决定。因此，softmax回归是一个线性模型（linear model）。

在softmax中我们就使用最大似然估计作为**损失函数**

$$l(y,\hat{y}) = -\sum_{j=1}^{q} y_j log\widehat{y_j}$$