百度无人车第 10 组

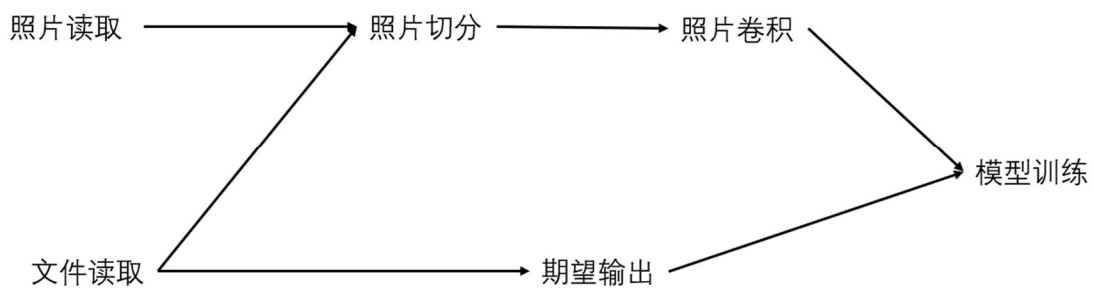
小组成员：王一舟，文启豪，朱家骆，王乙，李佳奇

1. 项目总体框架：

   1.1 寻找自动驾驶数据集：

   数据集来源：阿里云

   1.2 代码编写：



> 数据预处理 ---> 训练 <--(反复循环这两个步骤)--> 检测正确率 ---> 输出

2. 代码框架：

   2.1 文件读取：

   通过 opencv 库读取图片，以 numpy 矩阵的形式保存

   2.2 图像卷积：

   将图像进行卷积，形成 10*10 像素矩阵

   2.3 模型训练：

   尝试使用自己编写的 BP 算法进行训练，使用 sigmoid transfer function：

   层数：4 层，2 隐含层

   每层节点数：[300,30,10,3]

   结果：不理想

   尝试使用外部 BP 算法进行训练，使用 tanh transfer function：

   层数：4 层，2 隐含层

   每层节点数：[300,30,10,3]

来源：https://www.bbsmax.com/A/LPdoMYXyJ3/

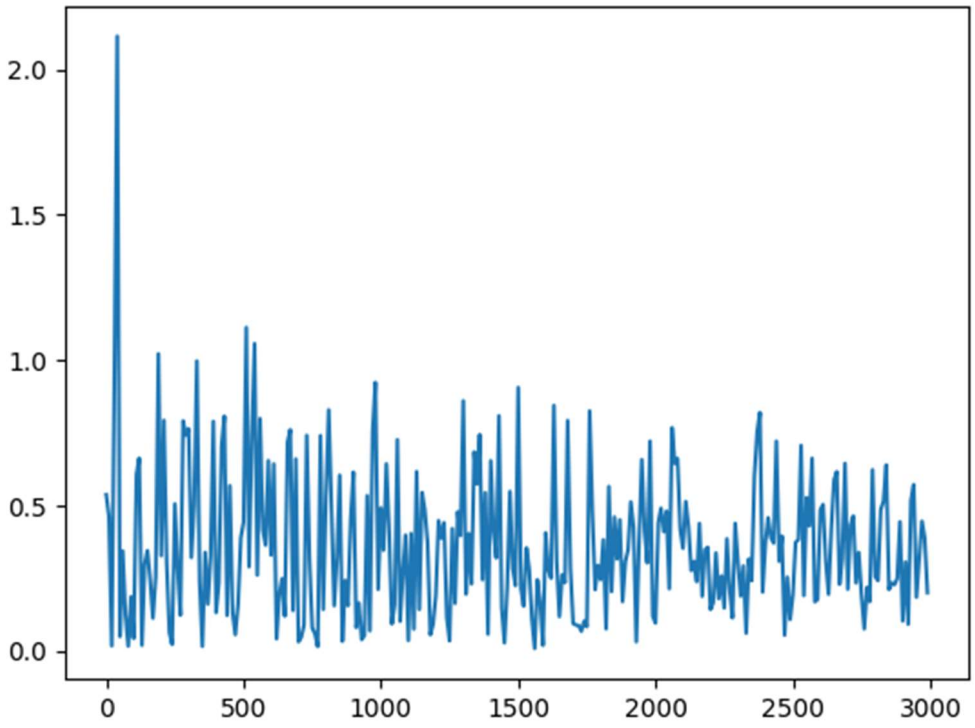
结果：较为理想

经过测试，learning rate 取 0.1，总共 1000 测试样例，将其打乱顺序后固定 900 个作为训练集，剩下 100 用来计算准确率。如果准确率不足 90%则将测试集重新学习。每隔 10 次学习后测量 error 并记录。

2.5 效果可视化：

每次循环打出其正确率。使用 matplotlib 进行结果可视化，将输出层 error 与学习次数的关系显示成图像。

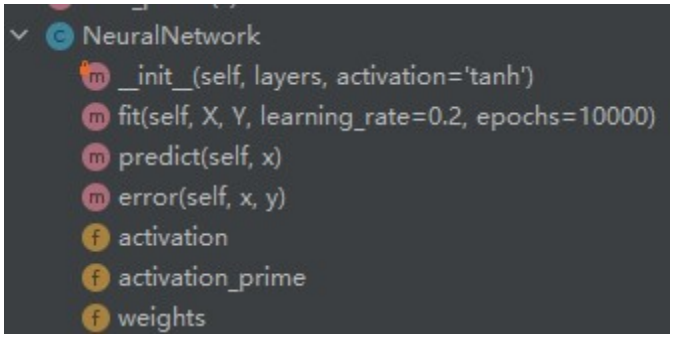
3. 运行结果：



可见输出层 error 随着学习次数增加而减少。

循环了三次，准确率随之升高，最终准确率达 92%

4.训练模型代码:




```python
class NeuralNetwork:
    def __init__(self, layers, activation='tanh'):
        """
        :参数layers: 神经网络的结构(输入层-隐含层-输出层包含的结点数列表)
        :参数activation: 激活函数类型
        """
        if activation == 'tanh':  # 也可以用其它的激活函数
            self.activation = tanh
            self.activation_prime = tanh_prime
        else:
            pass

        # 存储权值矩阵
        self.weights = []

        # range of weight values (-1,1)
        # 初始化输入层和隐含层之间的权值
        for i in range(1, len(layers) - 1):
            r = 2 * np.random.random((layers[i - 1] + 1, layers[i] + 1)) - 1  # add 1 for bias node
            self.weights.append(r)

        # 初始化输出层权值
        r = 2 * np.random.random((layers[i] + 1, layers[i + 1])) - 1
        self.weights.append(r)
```

```python
def fit(self, X, Y, learning_rate=0.2, epochs=10000):
    # Add column of ones to X
    # This is to add the bias unit to the input layer
    X = np.hstack([np.ones((X.shape[0], 1)), X])

    for k in range(epochs):  # 训练固定次数
        #if k % 1000 == 0: print('epochs:', k)

        # Return random integers from the discrete uniform distribution in the interval [0, low).
        i = np.random.randint(X.shape[0], high=None)
        a = [X[i]]  # 从m个输入样本中随机选一组

        for l in range(len(self.weights)):
            dot_value = np.dot(a[l], self.weights[l])  # 权值矩阵中每一列代表该层中的一个结点与上一层所有结点之间的权值
            activation = self.activation(dot_value)
            a.append(activation)

        # 反向递推计算delta:从输出层开始,先算出该层的delta,再向前计算
        error = Y[i] - a[-1]  # 计算输出层delta
        deltas = [error * self.activation_prime(a[-1])]

        # 从倒数第2层开始反向计算delta
        for l in range(len(a) - 2, 0, -1):
            deltas.append(deltas[-1].dot(self.weights[l].T) * self.activation_prime(a[l]))

        # [level3(output)->level2(hidden)]  => [level2(hidden)->level3(output)]
        deltas.reverse()  # 逆转列表中的元素

        # backpropagation
        # 1. Multiply its output delta and input activation to get the gradient of the weight.
        # 2. Subtract a ratio (percentage) of the gradient from the weight.
        for i in range(len(self.weights)):  # 逐层调整权值
            layer = np.atleast_2d(a[i])  # View inputs as arrays with at least two dimensions
            delta = np.atleast_2d(deltas[i])
            self.weights[i] += learning_rate * np.dot(layer.T, delta)  # 每输入一次样本,就更新一次权值

def predict(self, x):
    a = np.concatenate((np.ones(1), np.array(x)))  # a为输入向量(行向量)
    for l in range(0, len(self.weights)):  # 逐层计算输出
        a = self.activation(np.dot(a, self.weights[l]))
    return a
```