CS231n Assignment #1

● KNN:

Q1：KNN 分类器工作流程如下

- 记住所有的训练数据和其标签
- 对于测试数据，选择 K 个和其距离（例如欧式距离或其它距离度量方式）最近的数据，选择这 K 个数据中代表标签最多的，就是测试数据的标签

要求使用二层循环，单层循环，无循环三种方法实现测试数据和训练数据 L2 距离的计算。

A1：双层/单层循环实现较为简单，均可通过循环对每一项/行进行 L2 距离计算得到结果，在此不进行展示。

无循环的方法需要用到矩阵运算，考虑公式（a – b）^2 = a^2 + b^2 – 2ab，即可对两矩阵的 L2 距离公式做如下变换：

$$\begin{aligned}&\mathbf{dist[i][j]}\\&=\sqrt{(T_{i1}-P_{j1})^2+(T_{i2}-P_{j2})^2+\cdots+(T_{ik}-P_{jk})^2+\cdots+(T_{iD}-P_{jD})^2}\\&=\sqrt{(T_{i1}^2+T_{i2}^2+\cdots+T_{iD}^2)+(P_{j1}^2+P_{j2}^2+\cdots+P_{jD}^2)-2*(T_{i1}P_{j1}+T_{i2}P_{j2}+\cdots+T_{iD}P_{jD})}\\&=\sqrt{\|T_i^2\|+\|P_j^2\|-2*T_iP_j^T}\end{aligned}$$

将其转化为矩阵各元素平方和，矩阵乘法等运算，相应代码如下：

```
思路：
 (X1  -  X2) ^2  =  X1^2  +  X2^2  -  2  *  X1  *  X2
"""
num_test  =  X.shape[0]
num_train  =  self.X_train.shape[0]
dists  =  np.zeros((num_test,  num_train))
pass
dists  +=  np.sum(self.X_train  **  2,  axis  =  1).reshape(1,  num_train)
dists  +=  np.sum(X**2,  axis  =  1).reshape(num_test,  1)
dists  -=  2  *  np.dot(X,  self.X_train.T)
dists  =  np.sqrt(dists)
```

Q2：实现 K 折交叉验证。

A1：在机器学习中，将数据集 A 分为训练集 B（training set）和测试集 C（test set），在样本量不充足的情况下，为了充分利用数据集对算法效果进行测试，将数据集 A 随机分为 k 个包，每次将其中一个包作为测试集，剩下 k-1 个包作为训练集进行训练。划分部分代码如下：

```
#  将原始数据分为num_flods份
X_train_folds  =  np.array_split(X_train,num_folds)
y_train_folds  =  np.array_split(y_train,num_folds)
pass
```

交叉验证部分代码如下：

```
for  k  in  k_choices:
        #  将字典值部分初始化为一个大小为num_folds的数组
        k_to_accuracies[k]  =  np.zeros(num_folds)
        acc  =  []
        for  i  in  range(num_folds):
                #  把第i个空出来,作为验证集
                X_tr  =  X_train_folds[0:i]  +  X_train_folds[i+1:]
                y_tr  =  y_train_folds[0:i]  +  y_train_folds[i+1:]
                #  使用concatenate函数将剩余num_folds  -  1个训练集拼在一起
                X_tr  =  np.concatenate(X_tr,axis=0)
                y_tr  =  np.concatenate(y_tr,axis=0)
                X_cv  =  X_train_folds[i]
                y_cv  =  y_train_folds[i]


                classifier  =  KNearestNeighbor()
                #  将训练数据保存
                classifier.train(X_tr,y_tr)
                #  计算距离矩阵
                dists  =  classifier.compute_distances_no_loops(X_cv)
                #  预测结果
                y_cv_pred  =  classifier.predict_labels(dists,k=k)
                #  计算准确率
                num_correct  =  np.mean(y_cv_pred  ==  y_cv)

                acc.append(num_correct)
        k_to_accuracies[k]  =  acc
pass
```

总结：

1、 本实验中，通过 L2 距离循环实现和矩阵实现的运算时间对比，发现在数据量较大时，矩阵实现的运算速度远快于循环实现，故在今后代码实现中应多采用向量化运算。

2、 当训练数据集规模不大时，可采用 K 折交叉验证的方法，进行调优。

3、 KNN 分类器的决策边界是非线性的，不存在将两个类分开的超平面。

• SVM

Q1：实现 SVM 分类器的 loss 函数（梯度及损失值）

A1： 多类支持向量机（Support Vector Machine，SVM）的分类目标是使得正确类别的得分比其它类别的得分尽可能的大一个间隔(margin)，用△来表示。

对于单个样本，其损失函数表示为：

$$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + \Delta)$$

将第 j 类的得分和第 y 类别的得分带入上式有：

$$L_i = \sum_{j \neq y_i} max(0, W_j^T x_i - W_{y_i}^T x_i + \Delta)$$

同时为了防止过拟合，我们引入正则项：

$$L_i = \sum_{j \neq y_i} max(0, W_j^T x_i - W_{y_i}^T x_i + \Delta) + \lambda \sum_k \sum_l W_{k,l}^2$$

其中 L2 范数正则项限制权重 W 过大，使其更小且分布均匀。而 L1 正则项使得 W 分布得更稀疏，更离散。本实验采用 L2 正则。

由上图易得 loss 值代码实现如下：

```
# 计算得分矩阵
scores = X.dot(W)
# 矩阵行数
num_train = X.shape[0]

# 取出每一行的真实分类的那一列的得分,并且reshape成(num_train * 1)
current_score = scores[np.arange(num_train), y].reshape(num_train, 1)
# 损失函数计算
margins = np.maximum(0, scores - current_score + 1)
# 将其中真实分类对应列的值置为0
margins[np.arange(num_train), y] = 0
# 对所有的列求和，并对所有行求和
loss = np.sum(margins)
loss /= num_train
# 正则化
loss += reg * np.sum(W ** 2)
```

求 W 梯度时，只有 i==j 的那一行，才有可能对梯度产生贡献，i<>j 时整个式子均为常数，而常数求导为零。公式推导如下：

$$\begin{cases} \nabla_{w_{y_i}} L_i = -\left(\sum_{j \neq y_i} 1(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) \right) x_i & j = y_i \\ \nabla_{w_j} L_i = 1(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) x_i & j \neq y_i \end{cases}$$

之后还应加上正则项的导数 2*W，故最终代码实现如下：

```
# 计算梯度
dS = np.zeros_like(scores)
# 找到大于0的坐标，二维坐标
idx = np.where(scores - current_score + 1 > 0)
dS[idx] = 1
dS[np.arange(num_train), y] = -1 * (np.sum(scores - current_score + 1 > 0, axis=1) - 1)
dW = X.T.dot(dS)
dW /= num_train
dW += 2 * reg * W
```

Q2：SGD
A2：使用梯度下降时，则每次独立变量迭代的计算成本 O(n)，随 n 线性增长。因此，当训练数据集较大时，每次迭代的梯度下降成本将更高。随机梯度下降 (SGD) 可降低每次迭代时的计算成本。在随机梯度下降的每次迭代中，我们随机统一采样一个指数 $i \in \{1,\ldots,n\}$以获取数据示例，并计算渐变 $\nabla f_i(\mathbf{x})$以更新 $\mathbf{x}$，公式如下：

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f_i(\mathbf{x}),$$

其中 η是学习率。我们可以看到，每次迭代的计算成本从梯度下降的 O(n) 降至常数 O(1)。此外，随机梯度 $\nabla f_i(\mathbf{x})$是对完整梯度 $\nabla f(\mathbf{x})$的公正估计，因为：

$$\mathbb{E}_i \nabla f_i(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\mathbf{x}) = \nabla f(\mathbf{x}).$$

即平均而言，随机梯度是对梯度的良好估计值。代码实现如下：

```
self.W = self.W - learning_rate * grad
```

总结：

1、 对于凸函数优化，我们可以证明，随机梯度下降将能收敛到最佳解决方案。对于深度学习而言，情况通常并非如此。但是，对凸问题的分析使我们能够深入了解如何进行优化，即逐步降低学习率。
2、 如果学习率太小或太大，就会出现问题。实际上，通常只有经过多次实验后才能找到合适的学习率。
3、 当训练数据集中有更多示例时，计算渐变下降的每个迭代的成本更高，因此在这些情况下，首选随机梯度下降。

• Softmax

Q1:实现 Softmax 损失函数，计算其损失值及梯度。
A1：多分类交叉熵损失 和 SVM 损失不一样的是其在计算交叉熵损失之前需要将输出归一化（即统一成一个概率分布），具体的函数表达式如下：

$$Softmax(x)_i = \frac{e^{s_i}}{\sum_{j=1}^{C} e^{s_j}} \quad (i = 1, \ldots, C)$$

即其表示是 C 类中每类的概率。然后才是我们对于每一样本的损失函数：

$$L_i = -\log \frac{e^{s_{y_i}}}{\sum_{j=1}^{C} e^{s_j}} = -s_{y_i} + \log \sum_{j=1}^{C} e^{s_j}$$

实际运算中，因为有指数运算，为了防止运算结果过大，所以我们需要进行处理：

$$\frac{e^{s_{y_i}}}{\sum_{j=1}^{C} e^{s_j}} = \frac{De^{s_{y_i}}}{D\sum_{j=1}^{C} e^{s_j}} = \frac{e^{s_{y_i}+logD}}{\sum_{j=1}^{C} e^{s_j+logD}}$$

其中：

$$logD = -max(s_j)$$

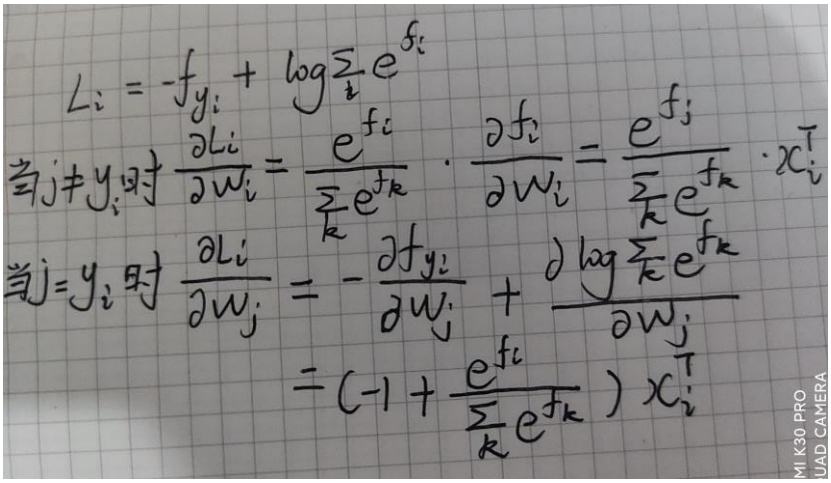
为了防止过拟合，我们还需要在最后的损失上加正则项。故最终表达为：

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + \lambda R(W)$$

相应代码如下：

```
num_train = X.shape[0]
score = X.dot(W)
#数值稳定
score -= np.max(score, axis = 1)[:, np.newaxis]
correct_score = score[range(num_train), y]
exp_score = np.exp(score)
sum_exp_score = np.sum(exp_score, axis = 1)
loss = np.sum(np.log(sum_exp_score) - correct_score)
loss /= num_train
loss += 0.5 * reg * np.sum(W * W)
```

梯度推导如下：

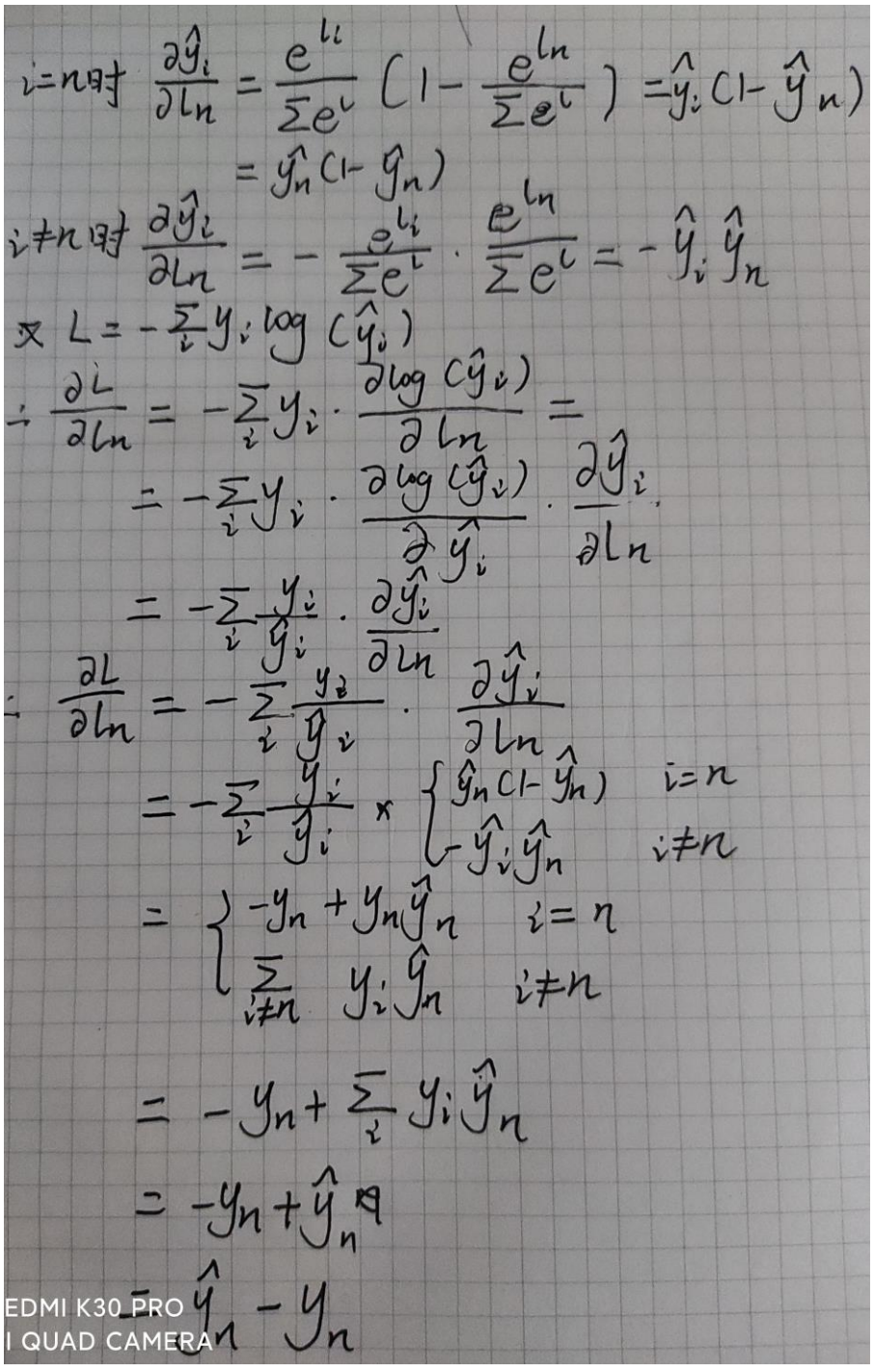


之后还需加上正则项，最终代码实现如下：

```
margin = np.exp(score) / sum_exp_score.reshape(num_train, 1)
margin[np.arange(num_train), y] += -1
dw = X.T.dot(margin)
dw /= num_train
dw += reg * w
```

补充：为什么在 mxnet，pytorch 等深度学习架构中，softmax 通常会和交叉熵损失函数结合在一起。

推导过程如下：



Softmax 和交叉熵函数的结合，在反向传播的求导过程中，能取得较为简便的表达。

- Two layer net

Q1：实现单隐藏层神经网络的 loss 函数，求其损失值及相关梯度。

A1：单隐藏层神经网络是一种非线性分类器，区别于 SVM 分类器和 Softmax 分类器，它的中间网络层通过激活函数等手段引入了非线性性，它只有一个中间隐藏层。一般的神经网络模型都包括**前向传播**和**反向传播**两个过程。
前向传播接收输入 X，然后映射到中间隐藏层 H：

$$H = f(W_1 X + b_1)$$

然后中间隐藏层 H 再经过映射得到我们的预测分数层 S

$$S = f(W_2 H + b_2)$$

这里的 S 我们可以看成之前线性分类器的得分，然后后面使用 SVM 或者 Softmax 进行分类。

反向传播阶段就是根据计算图和链式法则计算出损失函数对于参数的梯度，然后用 SGD 来更新梯度。

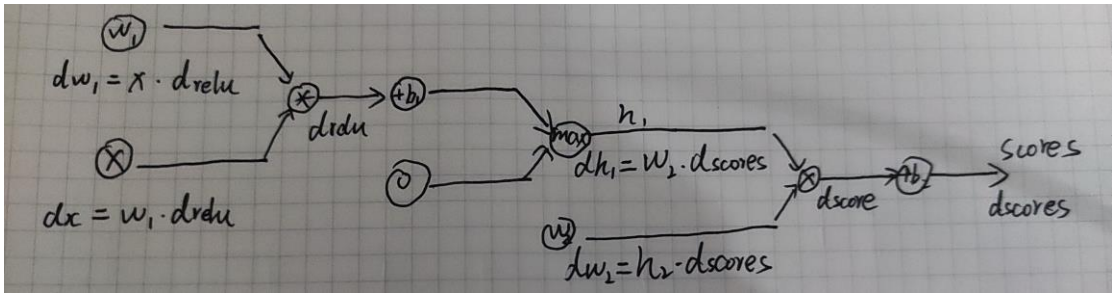
计算 S 代码实现如下：

```
W1, b1 = self.params['W1'], self.params['b1']
W2, b2 = self.params['W2'], self.params['b2']
N, D = X.shape
h1 = np.maximum(0, np.dot(X, W1) + b1)
scores = np.dot(h1, W2) + b2
```

计算 loss 代码实现如下（使用 Softmax）：

```
exp_scores = np.exp(scores)
row_sum = np.sum(exp_scores, axis=1).reshape(N, 1)
norm_scores = exp_scores / row_sum
data_loss = - 1 / N * np.sum(np.log(norm_scores[np.arange(N),y]))
reg_loss = 0.5 * (np.sum(W1 * W1) + np.sum(W2 * W2))
loss = data_loss + reg_loss
```

反向传播阶段计算梯度时，计算图如下所示：



故有如下代码实现：

```
dscores = norm_scores.copy()
dscores[range(N),y] -= 1
dscores /= N                                        #(N,C)
grads['b2'] = np.sum(dscores,axis=0)                #(C,)
dh1 = dscores.dot(W2.T)                             #(N,H)
grads['W2'] = h1.T.dot(dscores) + W2    #(H,C)
dRelu = (h1 > 0) * dh1                              #(N,H)
grads['W1'] = X.T.dot(dRelu) + W1           #(D,H)
grads['b1'] = np.sum(dRelu,axis=0)                  #(H,)
```

总结：

1、 单隐藏层神经网络通过激活函数等手段引入了非线性，能够解决部分线性不可分问题。

2、 计算较为复杂的神经网络的反向传播时，可以通过构筑计算图进行简化。

- # 本章总结

一开始进行本章实验时，觉得实验要求及题目设置非常简单，可以高效完成。但在实际试验过程中，在求 loss 值，模型调参等方面较为顺利，在反向传播的推导，形状计算等方面均产生了不少问题。经过反思，应该是自己过度依赖 pytorch 框架所致（在《动手深度学》代码实践中，使用 pytorch 时，仅需实现正向传播，该框架会自动实现反向传播，因此忽视了反向传播背后的原理和细节）。通过本章实验，复习巩固了 KNN、Softmax、SVM、MLP，加深了对反向传播的理解，为接下来的实验打好了基础。