1. 配置问题

考虑到很多朋友使用Windows系统（包括我这个VegetableBird...），这就直接导致数据集无法通过官网所给的教程下载，因此这里首先要解决作业的配置问题。
想要运行官网上的那段命令，第一步需要下载数据集。点击这里进行下载。下载后不需要解压，将这个压缩包放在*Assignment1/CS231n/datasets*目录下。第二步下载Git。在Git Bash中通过cd命令进入到*Assignment1/CS231n/datasets*这个文件夹下，输入*./get_datasets.sh*这个命令，配置完成！
当然，还有些朋友可能会缺少某些包，那就缺少哪些下载哪些就好。

2 代码分析

2.1 KNN

在KNN中，需要完成k_nearest_neighbor.py和knn.ipynb中相应的代码。

2.1.1 训练数据的可视化

```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',
'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show();
```

这段代码的主要目的在于将每个类别中的一些训练数据可视化。*idxs*通过np.flatnonzero获得第y类的索引值，在随机选出*samples_per_class*个样本绘制出来。

2.1.2 compute_distances_two_loops

这个函数在k_nearest_neighbor.py中，使用两个循环计算第i个测试数据与第j个训练数据的L2 distance，并放入$dist_{ij}$中，这个是非常简单的。

```
# 直接计算即可
dists[i][j] = np.sqrt(np.sum((X[i] - self.X_train[j]) ** 2))
```

2.1.3 predict_labels

这个函数的目的是通过计算出的dists求得第i个测试数据到训练数据最近的K个样本类别。
第一部分代码：

```
# 获得排序后的索引序列
sorted_index = np.argsort(dists[i, :])
top_K = sorted_index[:k]
closest_y = self.y_train[top_K]
```

这部分代码的意义是对于第i个测试数据，先将dists的第i行进行排序，并获得排序后的索引序列，那么top_K = sorted_index[:k]。c此时最近的K个类别就是在top_K索引下的y_train的值。

第二部分代码：

```
y_pred[i] = np.argmax(np.bincount(closest_y))
```

这里使用了np.bincount函数，这个函数是产生一个元素数量比closet_y最大值大一的numpy矩阵。也就是说若closet_y中最大值为9，通过这个函数产生的矩阵元素个数为10个，每个索引对应的元素值就是该索引值在closet_y中出现的个数。所以前K个类别中最多的类即为最大值的索引。

2.1.4 compute_distances_one_loop

这里用到了广播原则。

```
# 使用广播原则 注意每行的和是L2的值
dists[i, :] = np.sqrt(np.sum((X[i] - self.X_train) ** 2, axis = 1))
```

X[i]维度为（D,）X_train的维度是（num_train, D），所以X[i]会沿行方向扩展，即扩展成每一行的元素值都与X[i]相同。

2.1.5 compute_distances_no_loop

个人觉得还是挺难的。
朴素想法：

```
X = X.reshape(num_test, 1, X.shape[1])
dists = np.sqrt(np.sum((X - self.X_train) ** 2), axis = 2)
```

思路：$dists_{ij} = \sqrt{(\sum_{k=0}^{D-1}(X_{ik} - X\_train_{jk})^2)}$，这里的D是X_train的列数，也就是输入的特征维度数。根据这个公式，可以发现如果不使用循环，可以把X和X_train扩展为（num_test,num_train,D）的numpy矩阵。这样扩展的目的是使X在第i行第j列z轴方向上是X[i,:]，使X_train在第i行第j列z轴方向上是X[j,:]。这就使得当扩展后相减后获得的新矩阵$new_{ijk} = X_{ik} - X_train_{jk}$，所以将new平方后在z轴方向上求和在开方即为$dists_{ij}$。可能这样说依旧有朋友不太理解，我在换种解释方法。X原本为一个(num_test,D)的矩阵，我们把它沿行方向立起来，这时它就成为了一个(num_test,1,D)维的矩阵，此时将它按列方向扩展。X_train原本为一个(num_train,D)的矩阵，把它也沿行方向立起来，这时它就成为了一个(num_train,1,D)维的矩阵。不过这里略有不同的在于把它的行当作列，也就是说此时它是一个(1，num_train,D)的矩阵。将它沿行方向扩展，就是上述所扩展出的矩阵。
下面考虑以下如何使用广播原则实现这一过程。X是(num_test,D)的矩阵，X_train是(num_train,D)的矩阵，将这连个都扩展成（num_test,num_train,D）的矩阵，所以可以把X reshape成(num_test,1,D)的矩阵，这样通过广播就可以获得获得相应的矩阵了。不过遗憾的是这种做法会导致内存超限，不过这还是提供了一个思路。

进阶想法：

```
# 方案二
d1 = np.sum(X ** 2, axis = 1)
d2 = np.sum(self.X_train ** 2, axis = 1)
d = d1.reshape(-1, 1) + d2.reshape(1, -1)
dists = np.sqrt(d - 2 * np.dot(X, self.X_train.T))
```

思路：这个方案的主要思路就是利用了$(a-b)^2=a^2+b^2-2ab$。回到这个问题上：

$$dist_{ij}=\Sigma_{k=0}^{D-1}(X_{ik}^2+X\_train_{jk}^2)-2\Sigma_{k=0}^{D-1}X_{ik}X\_train_{jk}$$

将这个式子分为两个部分，第一部分：

$$\Sigma_{k=0}^{D-1}(X_{ik}^2+X\_train_{jk}^2)$$

显然，我们可以把X平方后沿列方向求和获得d1，把X_train按平方后沿列方向求和获得d2，将d1变为列向量，d2变为行向量，二式相加通过广播就是上面这一部分。

第二部分：

$$2\Sigma_{k=0}^{D-1}X_{ik}X\_train_{jk}$$

这个实际上就是X的行元素与X_train的列元素相乘求和，这就是XX_train^T。
所以，通过这些运算就可以在不使用循环的情况下求出dists。

2.1.6 Cross-validation

这一部分就相对简单了，交叉验证。

```
# 第一部分
X_train_folds = np.array_split(X_train, num_folds)
y_train_folds = np.array_split(y_train, num_folds)

# 第二部分
for k in k_choices:
    k_to_accuracies[k] = []
    for j in range(num_folds):
        # 注意X_train_folds是一个列表 不是numpy类型
        # 交叉验证集
        X_cv_train = np.vstack(X_train_folds[:j] + X_train_folds[j + 1:])
        y_cv_train = np.hstack(y_train_folds[:j] + y_train_folds[j + 1:])
        # 交叉测试集
        X_cv_test = X_train_folds[j]
        y_cv_test = y_train_folds[j]
        classifier = KNearestNeighbor()
        classifier.train(X_cv_train, y_cv_train)
        dists = classifier.compute_distances_no_loops(X_cv_test)
        y_predict = classifier.predict_labels(dists, k)
        correct_number = np.sum(y_predict == y_cv_test)
        accuracy = float(correct_number) / y_cv_test.shape[0]
        k_to_accuracies[k].append(accuracy)
```

交叉验证的思想就是将数据分为k份，循环验证k次，每次取出第i份数据作为测试集进行验证。
所以，首先是将数据集分为k折，这里利用np.array_split。这里还要对超参数k进行一个最优选择，所以首先是对k值进行选择。之后对测试集进行选择，第j次选择第j份数据，将剩余数据进行合并。这一部分比较简单，就不在继续赘述了（好吧实际上是因为我懒...）。

2.1.7 plot the raw observations

```
# plot the raw observations
for k in k_choices:
    accuracies = k_to_accuracies[k]
    plt.scatter([k] * len(accuracies), accuracies)

# plot the trend line with error bars that correspond to standard deviation
accuracies_mean = np.array([np.mean(v) for k,v in
sorted(k_to_accuracies.items())])
accuracies_std = np.array([np.std(v) for k,v in
sorted(k_to_accuracies.items())])
plt.errorbar(k_choices, accuracies_mean, yerr=accuracies_std)
plt.title('Cross-validation on k')
plt.xlabel('k')
plt.ylabel('Cross-validation accuracy')
plt.show()
```

这部分代码实际上就是画出一个errorbar图，以五次准确率的方差作为误差。这里提醒一下（也有可能是我个人问题），[k] * len(accuracies)是五个k的列表，numpy用多了就忘了列表的特性了.....

2.2 SVM

在SVM中，需要完成linear_svm.py、linear_classifier.py和svm.ipynb中相应的代码。

2.2.1 Preprocessing

```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# As a sanity check, print out the shapes of the data
print('Training data shape: ', X_train.shape)
print('Validation data shape: ', X_val.shape)
print('Test data shape: ', X_test.shape)
print('dev data shape: ', X_dev.shape)


# Preprocessing: subtract the mean image
# first: compute the image mean based on the training data
mean_image = np.mean(X_train, axis=0)
print(mean_image[:10]) # print a few of the elements
plt.figure(figsize=(4,4))
plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize the mean
image
plt.show()

# second: subtract the mean image from train and test data
X_train -= mean_image
```

```
X_val -= mean_image
X_test -= mean_image
X_dev -= mean_image

# third: append the bias dimension of ones (i.e. bias trick) so that our SVM
# only has to worry about optimizing a single weight matrix W.
X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

print(X_train.shape, X_val.shape, X_test.shape, X_dev.shape)
```

这段代码很简单，但是需要注意一下，这里的训练数据集和测试数据集全部被拉伸成了一维矩阵。在第二次预处理中图片全部减去了像素值的均值。最后又添加了一个全一列，这是因为SVM的方程是$w^T x + b$，通过增添一个全一列就可以使w和b合并，使得方程变为$w^T x$，这样就不必考虑b了。

2.2.2 svm_loss_naive

这一部分损失比较好求，但是梯度可能有些复杂。注意梯度的一部分代码放入可for循环中，并没有全部放在TODO这里。

```
    # compute the loss and the gradient
    num_classes = W.shape[1]
    num_train = X.shape[0]
    loss = 0.0
    for i in range(num_train):
        scores = X[i].dot(W)
        correct_class_score = scores[y[i]]
        for j in range(num_classes):
            if j == y[i]:
                continue
            margin = scores[j] - correct_class_score + 1 # note delta = 1
            if margin > 0:
                loss += margin

                # 计算梯度
                dW[:, j] += X[i].T
                dW[:, y[i]] -= X[i].T

    # Right now the loss is a sum over all training examples, but we want it
    # to be an average instead so we divide by num_train.
    loss /= num_train

    # Add regularization to the loss.
    loss += reg * np.sum(W * W)


 #############################################################################
    # TODO:
#
    # Compute the gradient of the loss function and store it dW.
#
    # Rather than first computing the loss and then computing the derivative,
#
    # it may be simpler to compute the derivative at the same time that the
#
```

```
    # loss is being computed. As a result you may need to modify some of the
#
    # code above to compute the gradient.
#

 ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    dw /= num_train
    dw += 2 * reg * W

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    return loss, dw
```

先将损失函数列出来（C是样本种类数）：

$$loss = \Sigma_{k=0}^{C-1}(S_k - S_{y_i} + \Delta)(k \neq y_i)$$

从这个损失函数可以看出，loss的计算只需要利用两层循环即可把每一个训练样本的loss值求出来。现在把重点放在梯度上。
在这个损失函数中，$S = XW$，这里由于写成了循环的形式，所以就将该式写为$S = X_iW$，也就是说$S_j = \Sigma_{k=0}^{D-1}X_{ik}W_{kj}$，D是X的特征维度数。现在我们通过这一个样本产生的损失值$loss_i$对W的第j列求偏导。

$$loss_i = \Sigma_{j=0}^{C-1}max(0, W_{:,j}^TX_i - W_{:,y_i}^TX_i + 1)(j \neq y_i)$$

考察$W_{:,j}$产生的损失值$loss_{ij}$

$$loss_{ij} = \Sigma_{k=0}^{D-1}max(0, X_{ik}W_{kj})$$

显然如果间隔小于0，也就是$W_{:,j}^TX_i + 1 < 0$，此时是不会产生梯度的
但是如果比0大，那么$loss_{ij} = \Sigma_{k=0}^{D-1}X_{ik}W_{kj}$

$$\frac{\partial loss_{ij}}{\partial W_{:,j}} = X_i^T$$

$$\frac{\partial loss_{ij}}{\partial W_{:,y_i}} = -X_i^T$$

$$\frac{\partial loss_{ij}}{\partial W_{:,k}} = 0(k \neq j, y_i)$$

而$loss_i$的值是$loss_{ij}$的累加和
所以$loss_{ij}$对W的偏导和就是$loss_i$对W的偏导

在循环中计算出所有训练数据的偏导并求和。但是此时无论是loss还是grad都没有完，因为此时还没有求平均以及加入正则化项。求均值只需除以num_train即可，loss的正则化项是W中所有元素平方之和再与reg相乘，这个正则化项获得的梯度就是2*reg*W。
最后，我想补充一点：有些情况下并不把W和b合并，这时候就需要对b求偏导。先说结论，b的偏导是1或-1。这可以从上述式子看出，此时b就等同于W的最后一行。因为X在最后一列又添加了一个全一列，所以W最后一行添加b就等同于$w^Tx + b$。而最后一行的梯度值与训练数据的最后一列有关，最后一列全为1。所以最后一行的元素梯度在y_i列是-1，其他列是1。累加求平均之后由于并不是所有的训练数据都为同一类，所以最终b的梯度也就可能不是1或-1了，当然这一切的前提都是损失值大于0。
举个简单的例子：

$$
\begin{aligned}
&X, y: \\
&X = \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} && y = [0, 1] \\
&W, b: \\
&W = \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{bmatrix} && b = [b_1, b_2] \\
&S: \\
&S = XW + b = \begin{bmatrix} W_{11}X_{11} + b_1 & W_{12}X_{12} + b_2 \\ W_{21}X_{21} + b_1 & W_{22}X_{22} + b2 \end{bmatrix} \\
&\text{因此对}b\text{的梯度即为（假设损失值都大于}0\text{）：} \\
&grad_b = [(-1+1)/2, (1+(-1))/2] = [0, 0]
\end{aligned}
$$

2.2.3 svm_loss_vectorized

将损失函数和梯度计算转化为向量化计算，有了上面的基础，我觉得还是比较简单的。

```
    # 第一部分
    num_train = X.shape[0]
    pred_score = X.dot(W)
    true_pred_socre = pred_score[range(num_train), y].reshape(-1, 1)
    margin = np.maximum(pred_score - true_pred_socre + 1, 0)
    margin[range(num_train), y] = 0
    loss = np.sum(margin) / num_train + reg * np.sum(W ** 2)

    # 第二部分
    margin[margin > 0] = 1
    margin[range(num_train), y] = -np.sum(margin, axis = 1)
    dW = (X.T).dot(margin)
    dW = dW / num_train + 2 * reg * W
```

先求loss。XW就是训练数据的每个分类的预测得分情况，真实分类就是pred_score[range(num_train), y].reshape(-1, 1)，用得分减去真实分类得分再加一就是margin值。将小于0的元素全部更新为0。由于损失值不包含$margin_{iy_i}$，所以将$margin_{iy_i}$也更新为0。全部求和求平均再加入正则项就是损失值。再求grad。正如上面分析的，只有margin的值大于0，才会产生梯度。所以对于X_i，产生如果$margin_{ij}$不为0，那么X_i在$W_{:,j}$产生了X^T的梯度，在$W_{:,y_i}$上产生了$-X^T$的梯度。也就是说，在$W_{:,y_i}$上产生的梯度和$margin_{i,:}$的所有非0值有关。具体说来，就是有多少个非0值，就是有多少$-X^T$。因此，把$margin_{iy_i}$更新为$-\Sigma_{k=0}^{C-1} margin_{ik}$，C表示分类数。所以$X^T margin$就是所有样本的梯度和，求平均再加入正则化项就是dW。

2.2.4 LinearClassifier

2.2.4.1 train

```
# 第一部分
indexes = np.random.choice(num_train, batch_size)
X_batch = X[indexes, :]
y_batch = y[indexes]

# 第二部分
self.W -= learning_rate * grad
```

实际上这就是SGD，每次只选取一部分数据进行梯度下降。

2.2.4.2 predict

```
y_pred = X.dot(self.W)
y_pred = np.argmax(y_pred, axis = 1)
```

2.2.5 超参数选择

这个比较简单，就不过多解释了。

```
for learning_rate in learning_rates:
    for regularization_strength in regularization_strengths:
        svm_clf = LinearSVM()
        loss_hist = svm_clf.train(X_train, y_train, learning_rate =
learning_rate,
                                  reg = regularization_strength, num_iters =
1500, verbose = False)
        train_accuracy = np.mean(svm_clf.predict(X_train) == y_train)
        val_accuracy = np.mean(svm_clf.predict(X_val) == y_val)
        results[(learning_rate, regularization_strength)] = (train_accuracy,
val_accuracy)
        if val_accuracy > best_val:
            best_val = val_accuracy
            best_svm = svm_clf
```

2.2.6 W的可视化

```
# Visualize the learned weights for each class.
# Depending on your choice of learning rate and regularization strength, these
may
# or may not be nice to look at.
w = best_svm.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)
w_min, w_max = np.min(w), np.max(w)
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',
'ship', 'truck']
for i in range(10):
    plt.subplot(2, 5, i + 1)

    # Rescale the weights to be between 0 and 255
    wimg = 255.0 * (w[:, :, :, i].squeeze() - w_min) / (w_max - w_min)
    plt.imshow(wimg.astype('uint8'))
    plt.axis('off')
    plt.title(classes[i])
```

这一部分实际上就是把W的在每一个分类上的特征权重绘制成图像。这个W由于加入了偏置项b，所以先把走后一行剔除，再reshape成(32,32,3,10)的矩阵。(32,32,3)代表着3通道的32*32维矩阵图像，10代表10个分类。对W的值进行归一化调整至0~255绘制出来，就是W的图片。

2.3 Softmax

在Softmax中需要完成softmax.ipynb和softmax.py。

2.3.1 softmax_loss_naive

```
    num_train = X.shape[0]
    num_classes = W.shape[1]
    for i in range(num_train):
        scores = X[i].dot(W)
        scores -= np.max(scores)
        sum_i = np.sum(np.exp(scores))
        loss += -np.log(np.exp(scores[y[i]]) / sum_i)
        for j in range(num_classes):
            probility = np.exp(scores[j]) / sum_i
            if j != y[i]:
                dW[:, j] += X[i] * probility
            else:
                dW[:, j] += X[i] * (probility - 1)

    loss /= num_train
    loss += reg * np.sum(W * W)
    dW /= num_train
    dW += 2 * reg * W
```

利用循环实现softmax函数loss值计算，这个相对比较简单，套用公式再加入正则化项就好，重点在于梯度的计算。
先把由X_i产生的损失值列出来

$$loss_i = -log(\frac{e^S_{y_i}}{\Sigma_{j=0}^{C-1} e^{S_j}})(C是分类数)$$

考虑到e的幂指数有可能会很大，而$\frac{e^{S_{y_i}}}{\Sigma_{j=0}^{C-1} e^{S_j}} = \frac{Ke^{S_{y_i}}}{\Sigma_{j=0}^{C-1} Ke^{S_j}} = \frac{e^{S_{y_i}+logK}}{\Sigma_{j=0}^{C-1} e^{S_j+logK}}$

这个K常常取为$e^{-max(S_k)}$，所以损失值就变成了$\frac{e^{S_{y_i}-max(S_k)}}{\Sigma_{j=0}^{C-1} e^{S_j-max(S_k)}}$

如此一来不妨直接将$scores$减去其中的最大值

现在对$loss_i$求偏导

$$\frac{\partial loss_i}{\partial W_{:,j}} = \frac{\partial(-log\frac{e^{S_{y_i}}}{\Sigma_{j=0}^{C-1} e^{S_j}})}{\partial W_{:,j}} = \frac{\partial log(\Sigma_{j=0}^{C-1} e^{S_j} - loge^{S_{y_i}})}{\partial W_{:,j}}$$

对于$W_{:,j}$来说，只有$S_j = W_{:,j}^T X_i$出现了$W_{:,j}$的元素

所以$\frac{\partial loss_i}{\partial W_{:,j}} = \frac{\partial loss_i}{\partial S_j}\frac{\partial S_j}{\partial W_{:,j}}$

$$\frac{\partial loss_i}{\partial S_j} = \frac{e^{S_j}}{\Sigma_{j=0}^{C-1} e^{S_j}} - \frac{\partial loge^{S_{y_i}}}{\partial S_j}$$

当$j \neq y_i$时

$$\frac{\partial loge^{S_{y_i}}}{\partial S_j} = 0$$

当$j = y_i$

$$\frac{\partial loge^{S_{y_i}}}{\partial S_j} = \frac{\partial S_{y_i}}{\partial S_j} = 1$$

$$\frac{\partial S_j}{\partial W_{:,j}} = \frac{\partial W_{:,j}^T X_i}{\partial W_{:,j}} = X_i^T$$

令$probility = \frac{e^{S_j}}{\Sigma_{j=0}^{C-1} e^{S_j}} = \frac{e^{S_{y_i}+logK}}{\Sigma_{j=0}^{C-1} e^{S_j+logK}}$

当$j \neq y_i$时

$$\frac{\partial loss_i}{\partial W_{:,j}} = probility * X^T$$

当$j = y_i$时

$$\frac{\partial loss_i}{\partial W_{:,j}} = (probility - 1) * X^T$$

最后，加入正则化项的偏导即为所求梯度。
同样的，再补充一点：这里的X也是在最后一列增加了一个全一列。如果将式子写为$w^T x + b$这种形式，b的梯度就是probility或（probility-1），原因和SVM里所讲的一致。

2.3.2 softmax_loss_vectorized

这里用向量化的计算方法，分析和SVM很像，就不在过多介绍了，直接上代码

```
num_train = X.shape[0]
num_classes = W.shape[1]
scores = X.dot(W)
scores -= np.max(scores, axis = 1).reshape(-1, 1)
scores = np.exp(scores)
sum_scores = np.sum(scores, axis = 1).reshape(-1, 1)
class_prob = scores / sum_scores
L_i = class_prob[range(num_train), y]
loss = np.sum(-np.log(L_i)) / num_train
loss += reg * np.sum(W * W)
```

```
    class_prob[range(num_train), y] -= 1
    dW = X.T.dot(class_prob)
    dW /= num_train
    dW += 2 * reg * W
```

2.3.3 超参数选择

```
for lr in learning_rates:
    for reg in regularization_strengths:
        softmax = Softmax()
        softmax.train(X_train, y_train, lr, reg, num_iters=1000)
        y_train_pred = softmax.predict(X_train)
        train_accuracy = np.mean(y_train == y_train_pred)
        y_val_pred = softmax.predict(X_val)
        val_accuracy = np.mean(y_val == y_val_pred)
        if val_accuracy > best_val:
            best_val = val_accuracy
            best_softmax = softmax
        results[(lr, reg)] = train_accuracy, val_accuracy
```

2.4 two_layer_net

2.4.1 loss

计算loss的第一步首先要计算scores。在第一层神经网络中的激活函数是ReLU激活函数，这比较简单就不过多赘述了

```
        h1 = X.dot(W1) + b1
        h1 = np.maximum(0, h1)
        scores = h1.dot(W2) + b2
```

第二层的激活函数是softmax函数，获得scores后根据softmax函数对应的损失函数很容易就求出损失值了。

```
        scores -= np.argmax(scores, axis = 1).reshape(-1, 1)
        scores = np.exp(scores)
        sum_scores = np.sum(scores, axis = 1).reshape(-1, 1)
        class_prob = scores / sum_scores
        loss = np.sum(-np.log(class_prob[range(N), y]))
        loss /= N
        loss += reg * (np.sum(W1 * W1) + np.sum(W2 * W2))
```

现在通过反向传播计算梯度。反向传播还是比较难的，建议多看些资料。

```
        class_prob[range(N), y] -= 1
        dscores = class_prob
        grads['W2'] = h1.T.dot(class_prob) / N + 2 * reg * W2
        grads['b2'] = np.sum(class_prob, axis = 0) / N

        dH = dscores.dot(W2.T)
        dh = dH * (h1 > 0)
        grads['W1'] = X.T.dot(dh) / N + 2 * reg * W1
        grads['b1'] = np.sum(dh, axis=0)
```

首先来看一下前向传播的过程:

$X \longrightarrow XW_1 + b_1 \longrightarrow h \longrightarrow ReLU \longrightarrow H \longrightarrow HW_2 + b_2 \longrightarrow scores \longrightarrow softmax \longrightarrow class\_prob \longrightarrow loss$

现在我们通过反向传播，求出$\frac{\partial loss}{W_1}, \frac{\partial loss}{b_1}, \frac{\partial loss}{W_2}, \frac{\partial loss}{b_2}$。

反向传播：

1.$\frac{\partial loss}{\partial W_2}$

显然，这个就是$softmax$函数的梯度，过程不再细写

$\nabla W_2 = H^T addjust\_class\_prob$

注意，这里的输入是H而不是X

2.$\frac{\partial loss}{\partial b_2}$

实际上，如果同SVM和$Softmax$把H添加一个全一列，W添加一行

那么W的最后一行就是b，也就是说,$\nabla b_2 = \nabla W_2[-1,:]$

而该值就是$addjust_class_prob$沿行方向求和并求均值所得的矩阵

3.$\frac{\partial loss}{\partial W_1}$

由反向传播公式，需要求出$\frac{\partial scores}{\partial H}$

$scores = HW_2 + b_2$，所以$\frac{\partial scores}{\partial H} = W_2^T$

此时再求出$\frac{\partial H}{\partial h}$

这里使用的是$ReLU$激活函数，所以获得的偏导就是$(h > 0)$，大于0为1，小于等于0为0

注意H和h的关系，$\frac{\partial loss}{\partial h} = \frac{\partial loss}{\partial H} .* \frac{\partial H}{\partial h}$，注意是点乘

$h = XW_1 + b_1$，所以$\frac{\partial h}{\partial W_1} = X^T$

此时将所有的偏导相乘即为：

$X^T * dscores * W_2^T .* (h > 0)$，$.*$表示点乘，$*$表示矩阵乘法

这里写得比较简单，我另写了一份详细一点的过程，不过这需要对矩阵求导有一定的了解

4.$\frac{\partial loss}{\partial b_1}$

同b_2，$dscores * W_2^T .* (h > 0)$沿行方向求和并求均值的矩阵

5.细节部分

注意$\nabla W_1 \nabla b_1 \nabla W_2 \nabla b_2$均要求均值,这是因为这是所有样本数据点梯度之和

在最后要加入正则化部分，正则化的损失值和$softmax$损失值是相加的关系，所以梯度也是相加关系

前向传播：$X \xrightarrow{XW_1+b_1} h \xrightarrow{ReLU} H \xrightarrow{HW_2+b_2} scores \xrightarrow{softmax} class\_prob \xrightarrow{f} loss$

把箭头上的运算当作一个函数可能会更好地理解反向传播

1. $\frac{\partial loss}{\partial scores}$，这里不对 class_prob 求偏导是因为 class_prob 每行元素是相关联的！

$loss = \sum_{i=0}^{N}\left(-\log \frac{e^{s[i][y[i]]}}{sum[i]}\right)$，$sum[i]$ 是 e^{scores} 的第 i 行元素之和，N 是样本数

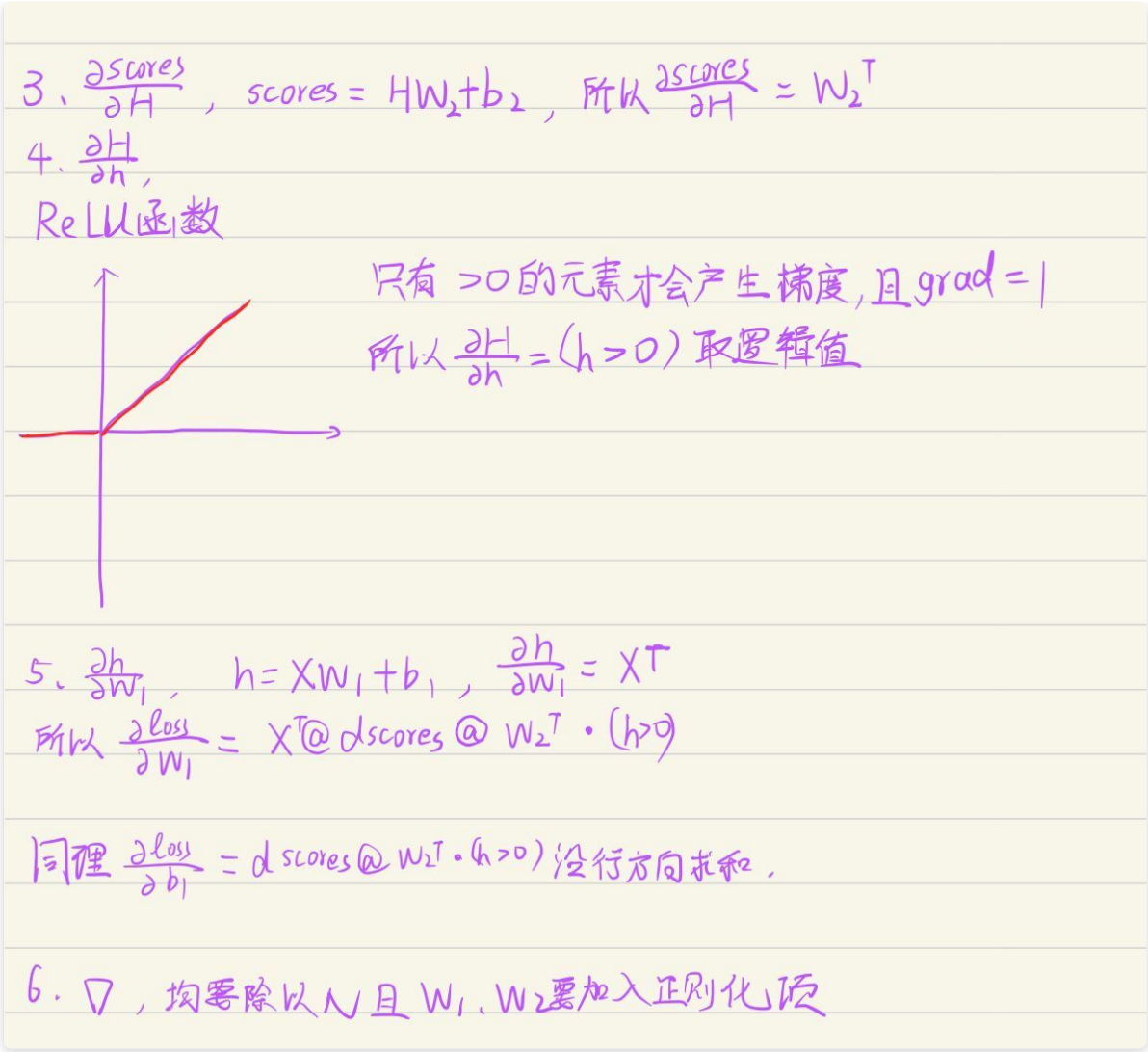
$$\frac{\partial loss}{\partial scores} = \begin{bmatrix} \frac{e^{s[0][0]}}{sum[0]} & \cdots & \frac{e^{s[0][C-1]}}{sum[0]} \\ \vdots & & \\ \frac{e^{s[n][0]}}{sum[n]} & \cdots & \frac{e^{s[n][C-1]}}{sum[n]} \end{bmatrix} = dscores$$，其中 $dscores[i][y[i]] -= 1$

这是 dscores 的由来，换成代码表示 $dscores = (class\_prob[i][y[i]] -= 1)$

2. $\frac{\partial scores}{\partial W_2}$，$HW_2 + b_2 = scores$，$\frac{\partial scores}{\partial W_2} = H^T$，这是矩阵的求导，

所以 $\frac{\partial loss}{\partial W_2} = H^T @ dscores$，而 $\frac{\partial loss}{\partial b_2} =$（dscores 沿列方向求和的值）

这里令 $H = [H, 1]$，$dscores\ H^T$ 的最后一行就是 b，同样也等于（dscores 沿列方向求和的值）



2.4.2 train

比较简单，就不细讲了。

```
            # 第一部分
            indexes = np.random.choice(num_train, batch_size)
            X_batch = X[indexes, :]
            y_batch = y[indexes]

            # 第二部分
            self.params['W2'] -= learning_rate * grads['W2']
            self.params['b2'] -= learning_rate * grads['b2']
            self.params['W1'] -= learning_rate * grads['W1']
            self.params['b1'] -= learning_rate * grads['b1']
```

2.4.3 predict

执行前向传播过程，不过softmax倒是没有必要加入进来了。

```
        W1, b1 = self.params['W1'], self.params['b1']
        W2, b2 = self.params['W2'], self.params['b2']
        h1 = X.dot(W1) + b1
        h1 = np.maximum(h1, 0)
        scores = h1.dot(W2) + b2
        y_pred = np.argmax(scores, axis = 1)
```

2.4.4 寻找超参数

```
best_val_acc = 0
best_lr = 0
best_hs = 0
best_reg = 0

learning_rates = [i/10000 for i in range(10, 21, 2)]
hidden_sizes = range(60, 110, 10)
regs = [i/4 for i in range(1, 5)]

for hs in hidden_sizes:
    for lr in learning_rates:
        for reg in regs:
            nn = TwoLayerNet(input_size, hs, num_classes)
            results = net.train(X_train, y_train, X_val, y_val,
                                num_iters=1500, batch_size=200,
                                learning_rate=lr, learning_rate_decay=0.95,
                                reg=reg, verbose=False)
            val_acc = np.mean(net.predict(X_val) == y_val)
            print("hs:%d, lr:%f, reg:%f, val accuracy:%f"%(hs, lr, reg,
val_acc))
            if val_acc > best_val_acc:
                best_val_acc = val_acc
                best_net = net
                best_hs = hs
                best_lr = lr
                best_reg = reg
print("best model is:")
print("hs:%d, lr:%f, reg:%f, val accuracy:%f"%(best_hs, best_lr, best_reg,
best_val_acc))
```

2.5 features

这一部分代码非常简单，都是寻找最优超参数的代码。但是features.py值得看一看，这份文件中包含了特征提取的一些方法，不过碍于篇幅就不在赘述了（实际上我也不太会...）。

3 Star

码字排版不易，给颗Star吧...