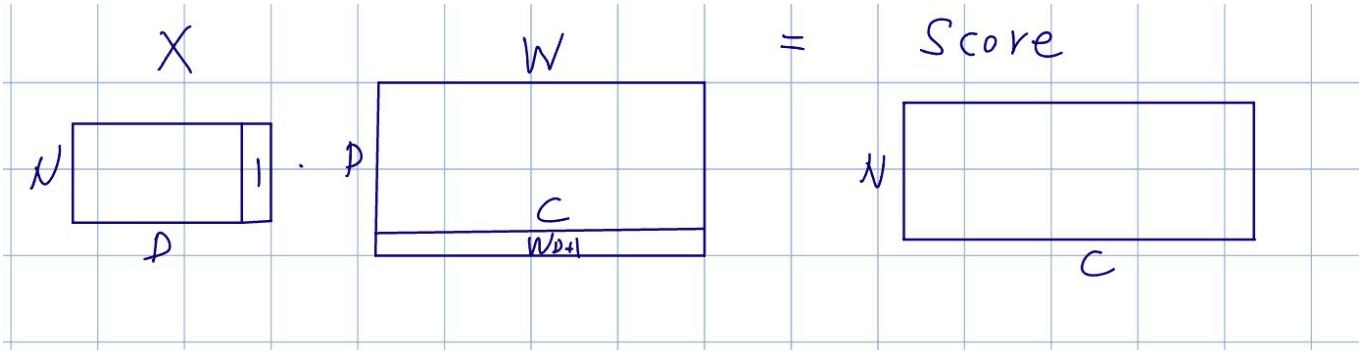
支持向量机

先处理数据，把图像reshape成1维的

```
# 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)
```

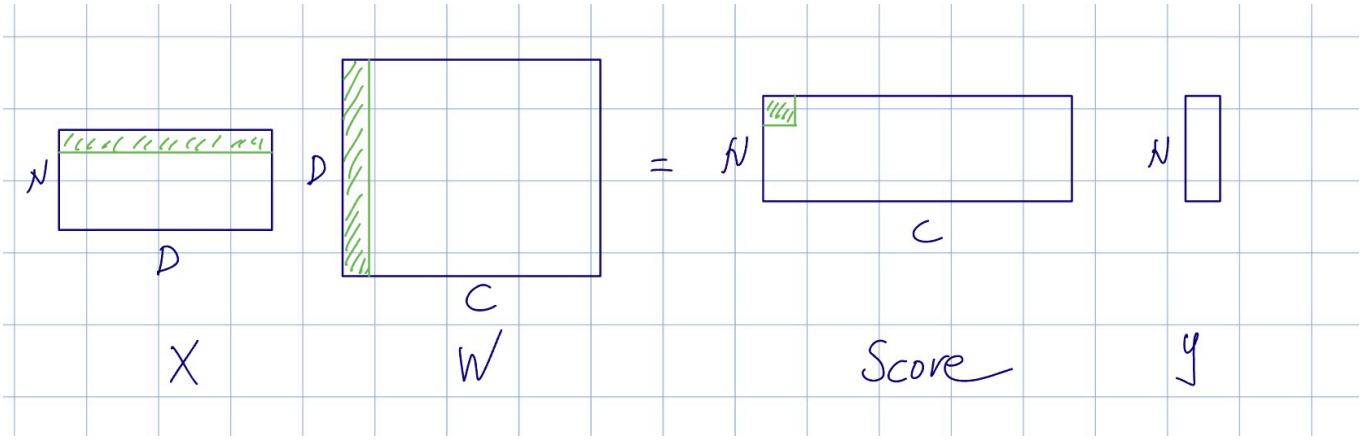
接下来要减掉图像平均值，加快计算速度

接下来给X多加一维，给W多加一行，把b放在W中一起训练




```
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))])
```

核心是计算损失函数和L关于W的导数



X.shape() = (N,D)，表示X这一组mini-batch有N个用来训练的数据，每个数据是D维
W.shape() = (D,C)，表示全部的权重矩阵，有C组数据，表示所有可能的选择为C个，D表示每种数据的维度为D，每一列为一种数据
score.shape() = (N,C)，表示 X W 矩阵乘法得到的分值矩阵，每一行为X[i] W，表示X中第 i 个数据和W相乘，为第 i 个数据对所有种类 C 分别的得分
y.shape() = (N,)，表示标注好的数据 X 对应的标签

对每一个分值 score[i][j]，其计算得出的损失函数
Li = max(0,score[i][j] - score[i][y[i]] + 1)
也就是第 j 类的分值减去正确类别 y[i] 的分值再加一
注意其中 i 不等于 y[i]，因为这会变成正确的类别和自己算，这整个一组数据跳过，不对W的修正产生任何影响
其意义在于如果错误的类别比正确类别还高，或者没有低于一个区间（这里是1），那么就判定为这次预测存在不准确之处，要累加损失函数
最后计算总损失函数
L = 1/N Σ*Li* + *reg* ΣW^2
后方为正则化项，用于控制模型的过拟合，reg为正则化超参数

先计算损失函数

```
def svm_loss_vectorized(W, X, y, reg):
    """
    Structured SVM loss function, vectorized implementation.
    Inputs and outputs are the same as svm_loss_naive.    """
    loss = 0.0
    dW = np.zeros(W.shape)  # initialize the gradient as zero

    """
    calculate the loss"""
    N = X.shape[0]
    score_matrix = np.dot(X,W)
    loss_matrix = score_matrix - ...
```

```
    score_matrix[np.arange(X.shape[0]),y].reshape(N,1) + 1
    margin = np.maximum(loss_matrix,0)
    margin[np.arange(X.shape[0]),y] = 0
    loss = margin.sum() / X.shape[0] + reg * np.sum(W ** 2)
```

这里代码中对整个矩阵一起操作，numpy可以GPU加速并行计算
先算出来每一项的损失矩阵，`score_matrix[np.arange(X.shape[0]),y].reshape(N,1)` 是先通过 np.arange生成行的索引，y为正确的标签的索引，这样就把正确的class的分数提取成一个单独的 (N,) 向量，重新变形成 (N,1)，然后让 score_matrix 的每一行减去该行正确class的分数，再加以，这时候正确的class的分数是 1 ，后面要消除掉影响
margin为损失函数，对loss矩阵逐元素取其和 0 的最大值，为hinge损失函数
再用 `margin[np.arange(X.shape[0]),y] = 0` 把正确类别的损失函数全部设为0,防止对 dW 和 L 有影响
最后加上正则化项 `reg * np.sum(W ** 2)`

接下来计算梯度

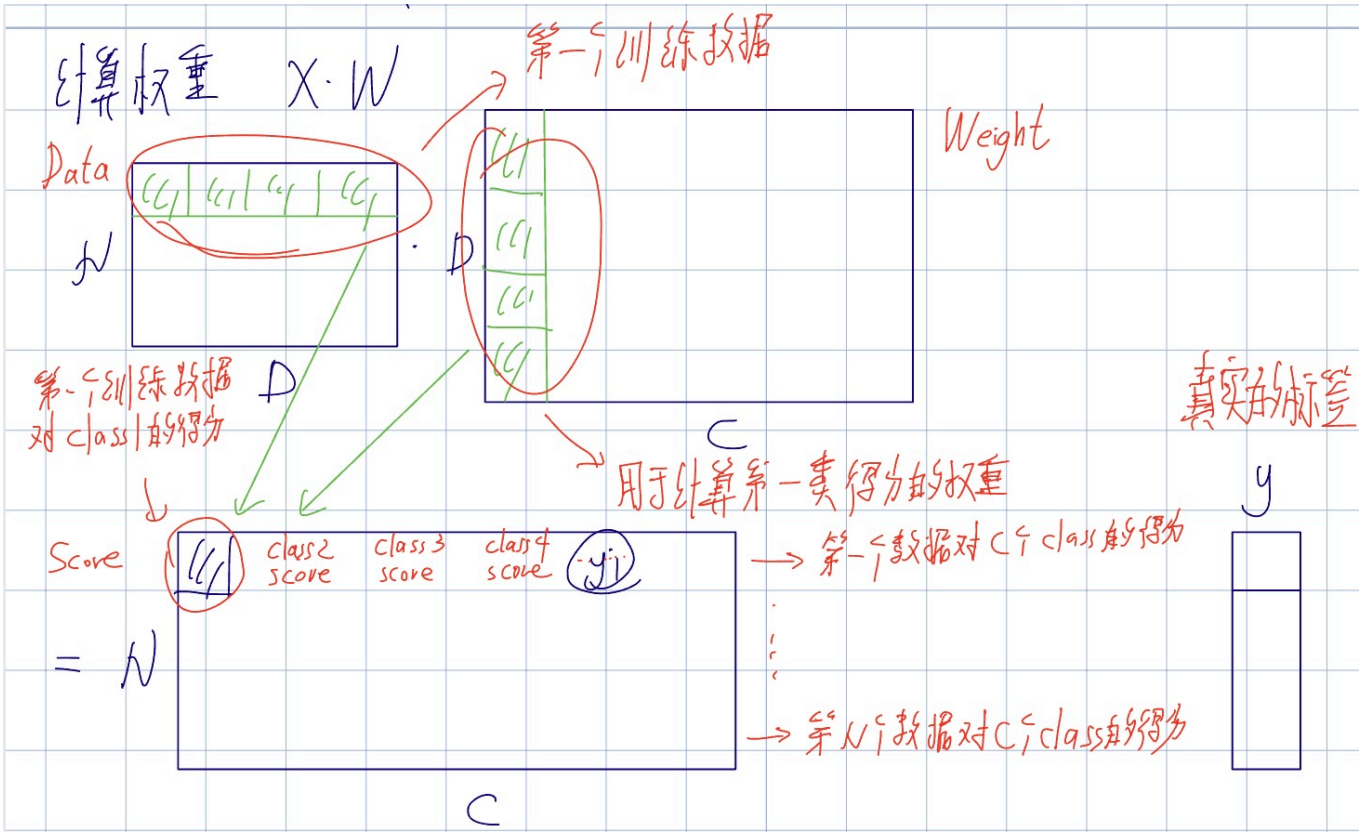
```
    """
    calculate the gradient of W dW, use for SVG later"""
    N = X.shape[0]
    binary = (margin > 0).astype(int)
    binary[np.arange(N), y] = -np.sum(binary, axis=1)
    dW = (X.T.dot(binary) / N) + (2 * reg * W)

    return loss, dW
```

这一部分比较抽象，为了不用循环用numpy加速运算进行了一点点数学变形
首先得出binary矩阵，存储所有margin > 0 的部分并转换成int类型
此时每一组中正确class对应的值是0，第三行重设数值，每个正确class的数值改为这一行所有数字之和的相反数
因为原来每个score[i][j] 可以展开为 Xij * Wij 对 j ≠ i 求和，再减去 Xi,y[i] * Wi,y[i]
因为对每个不为0的margin都要这样进行操作 margin，所以会减去这一行margin数字和的次数的 Xi,y[i] * Wi,y[i]，对应到原本的运算，就是每个非正确class且不为零的 score[i][j]，都会逐个改变 dW 中的元素，如下：

```
for i in range(num_train):
    '''dealing with training classes i
    '''     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
            dW[:,j] += X[i,:]
            dW[:,y[i]] -= X[i,:]
        else:
            '''loss stay unchanged'''
            '''dW stay unchanged'''
```

对score矩阵中的每一行来说，逐个计算 score[j]的影响，score[j]由X中的第i行和W中的第j列相乘得到，如果 j == y[i]，则跳过，该元素不参数运算
其他情况下，score[j] = Xi1Wi1 + Xi2Wi2 + ... + XiDWiD - score[y[i]]，则这一项损失对W的导数为向量[Xi1,Xi2 ..... XiD]，对应代码
`dW[:,j] += X[i,:]` 对dW的第j列逐个加上对应的Xi
`dW[:,y[i]] -= X[i,:]` 对dW的第y[i]列全部减去对应X行的值，-score[y[i]]要像前面一样对这对应的y[i]列逐个减去X的第y[i]行



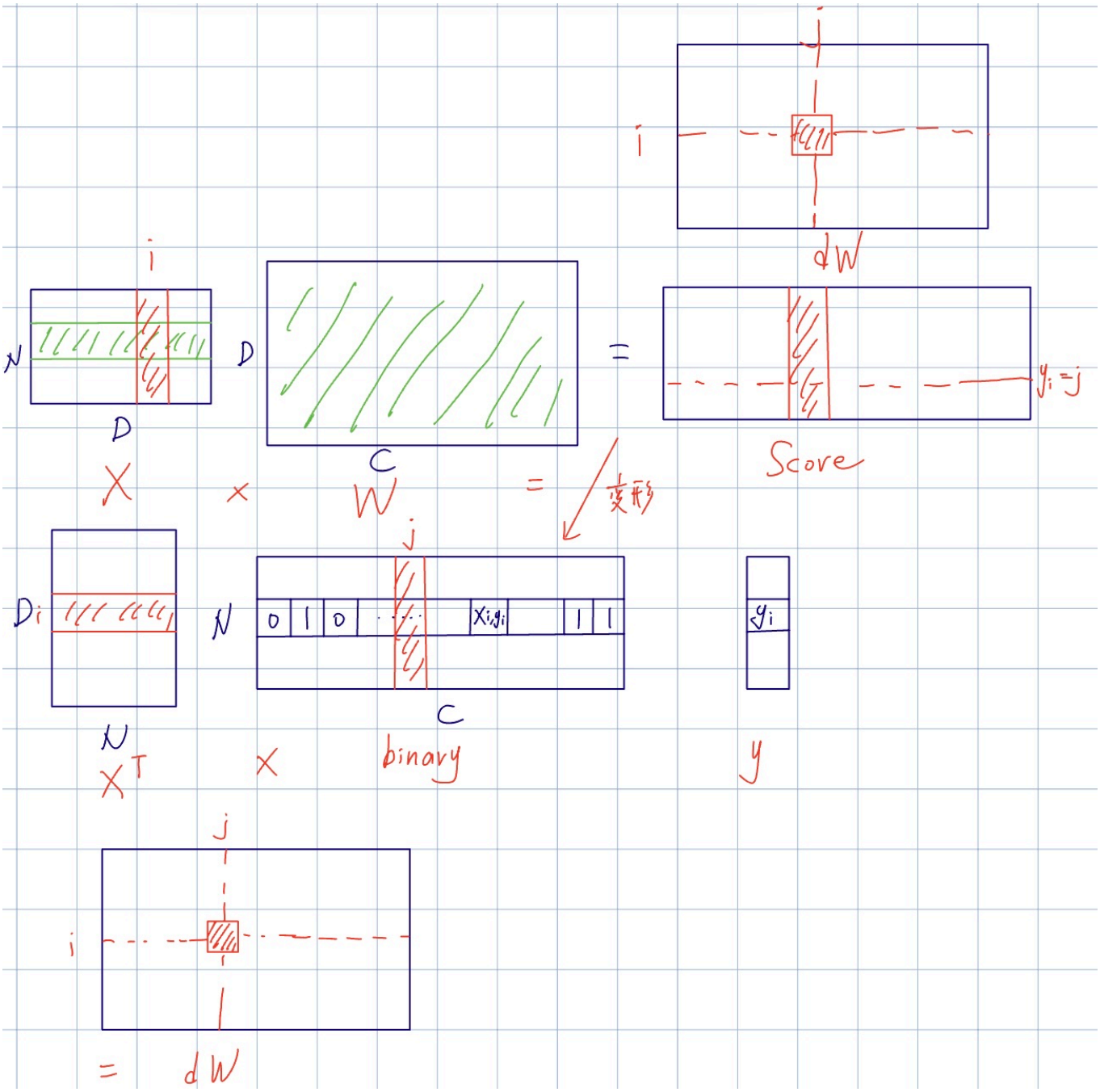
回头看向量化的代码的原理

```
    """
    calculate the gradient of W dW, use for SVG later"""
```

```
    N = X.shape[0]
    binary = (margin > 0).astype(int)
    binary[np.arange(N), y] = -np.sum(binary, axis=1)
    dW = (X.T.dot(binary) / N) + (2 * reg * W)

    return loss, dW
```

先看这个binary矩阵，score[i][j]的格子里是1代表要修正dW，dW[:,j] += X[i,:],同时修正dW[:,y[i]] -= X[y[i],:]，如果是0则代表无任何变化
所以拿这个binary操作矩阵操作后，我们期望的结果应该是对一个格子[i][j]，其值为binary[i][j]修正dW中第j列的值，dW += binary[i][j] * X[i,:]，注意这里把后面减去score[i][y[i]]那一项包含在了全组和的那一格
验证X.T.dot(binary)满足：
循环版本中，score中的每一列的正向修正效果是将X转置后沿axis = 1方向求和，再沿着修正dW的对应列
而binary数组中每一列（第j列）的正向修正效果是将X转置后做矩阵乘法，也是求和后正向修正而负向修正的部分是一样的，原来score矩阵中每个大于0的元素要额外修正一次正确的类别，现在在向量化版本中，一行的修正被压缩到了一个格子里一次性修正完

因此二者是等价的


j
i
dW
i
N
D
D
X
C
W
=
Score
$y_i=j$
×
=
变形
j
D_i
N
0
1
0
x_{ij}
1
1
y_i
N
X^T
×
binary
C
y
j
i
=
dW