Image Classification

Team:
Ahmed Rizk(14)
Yahia Mohamed ELShahawy(87)

Contents

1. Problem statement

In this assignment we practice putting together a simple image classification pipeline, based on the k-Nearest Neighbor or the SVM/Softmax classifier. The goals of this assignment are as follows:

- Understand the basic Image Classification pipeline and the data-driven approach (train/predict stages)
- Understand the train/validation/test splits and the use of validation data for hyper parameter tuning.
- Develop proficiency in writing efficient vectorized code with numpy
- Implement and apply a k-Nearest Neighbor (kNN) classifier.
- Implement and apply a Multiclass Support Vector Machine (SVM) classifier.
- Implement and apply a Softmax classifier.
- Implement and apply a Two layer neural network classifier.
- Understand the differences and tradeoffs between these classifiers.
- Get a basic understanding of performance improvements from using higher-level representations than raw pixels (e.g. color histograms, Histogram of Gradient (HOG) features).

The image classification pipeline: Image Classification task is to take an array of pixels that represents a single image and assign a label to it. Our complete pipeline can be formalized as follows:

Input: Our input consists of a set of N images, each labeled with one of K different classes. We refer to this data as the training set.
Learning: Our task is to use the training set to learn what every one of the classes looks like. We refer to this step as training a classifier, or learning a model.
Evaluation: In the end, we evaluate the quality of the classifier by asking it to predict labels for a new set of images that it has never seen before. We will then compare the true labels of these images to the ones predicted by the classifier.

2. Implementation

2.1.KNN

The kNN classifier consists of two stages:

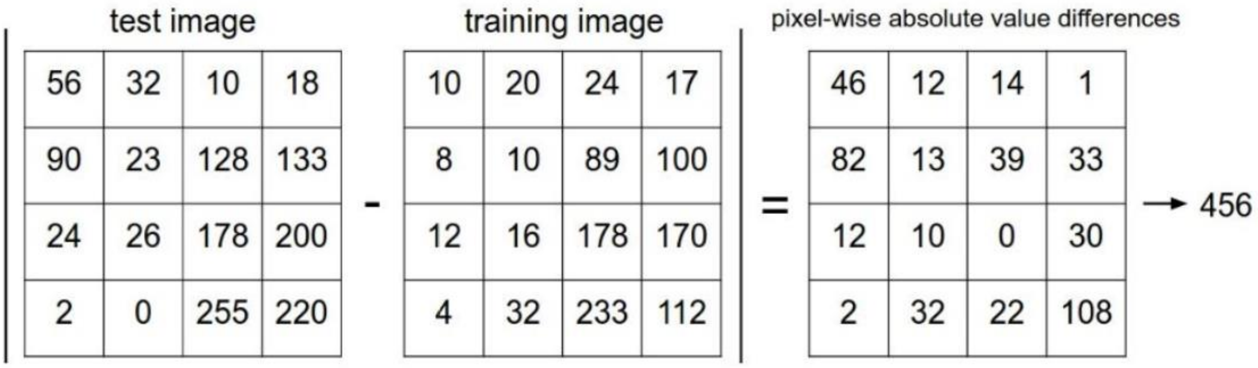
- During training, the classifier takes the training data and simply remembers it.
- During testing, kNN classifies every test image by comparing to all training images and transferring the labels of the k most similar training examples.
- The value of k is cross-validated.

Pseudocode

```
1. Load the training and test data
2. Choose the value of K
3. For each point in test data:
       - find the Euclidean distance to all training data points
       - store the Euclidean distances in a list and sort it
       - choose the first k points
       - assign a class to the test point based on the majority of classes
         present in the chosen points
4. End
```

L2-Distances:

$$d_2(I_1, I_2) = \sqrt{\sum_p \left(I_1^p - I_2^p\right)^2}$$



test image

56	32	10	18
90	23	128	133
24	26	178	200
2	0	255	220

-

training image

10	20	24	17
8	10	89	100
12	16	178	170
4	32	233	112

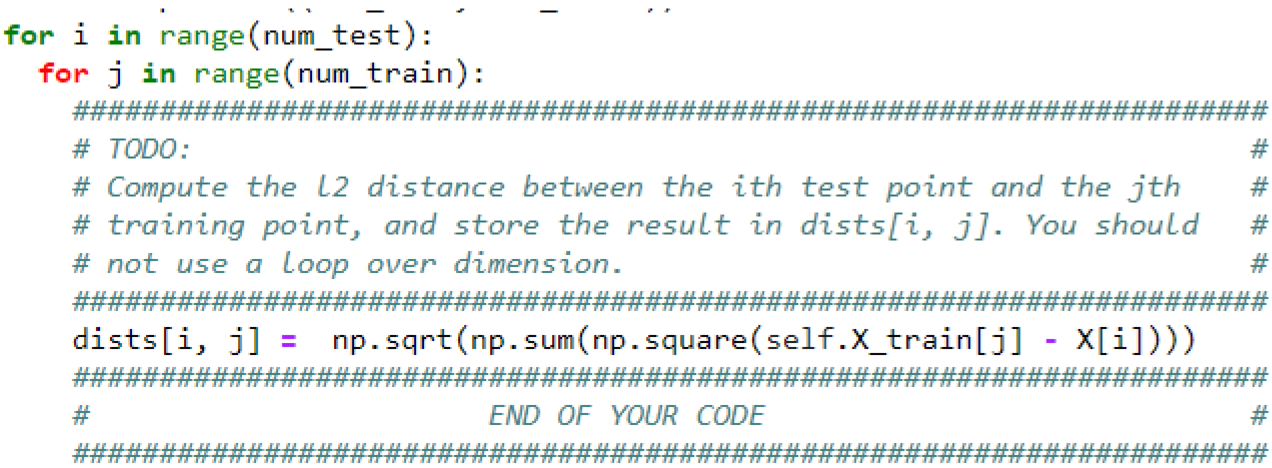
=

pixel-wise absolute value differences

46	12	14	1
82	13	39	33
12	10	0	30
2	32	22	108

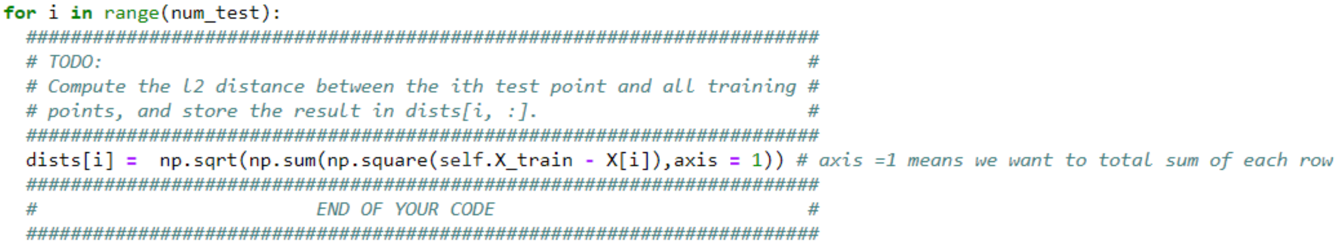
→ 456

Method 1: using 2 loops



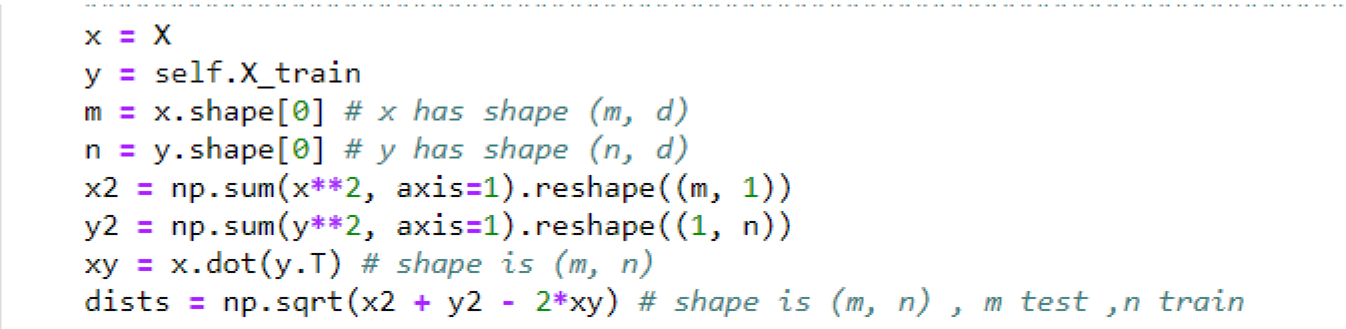
```
for i in range(num_test):
  for j in range(num_train):
    #####################################################################
    # TODO:                                                             #
    # Compute the l2 distance between the ith test point and the jth    #
    # training point, and store the result in dists[i, j]. You should   #
    # not use a loop over dimension.                                    #
    #####################################################################
    dists[i, j] =  np.sqrt(np.sum(np.square(self.X_train[j] - X[i])))
    #####################################################################
    #                       END OF YOUR CODE                            #
    #####################################################################
```

Method 2: using 1 loop



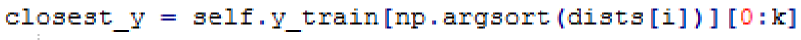
```
for i in range(num_test):
  #######################################################################
  # TODO:                                                               #
  # Compute the l2 distance between the ith test point and all training #
  # points, and store the result in dists[i, :].                        #
  #######################################################################
  dists[i] =  np.sqrt(np.sum(np.square(self.X_train - X[i]),axis = 1)) # axis =1 means we want to total sum of each row
  #######################################################################
  #                       END OF YOUR CODE                              #
  #######################################################################
```

Method3: Full vectorized version



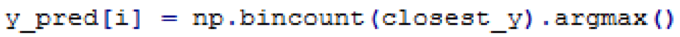
```
x = X
y = self.X_train
m = x.shape[0] # x has shape (m, d)
n = y.shape[0] # y has shape (n, d)
x2 = np.sum(x**2, axis=1).reshape((m, 1))
y2 = np.sum(y**2, axis=1).reshape((1, n))
xy = x.dot(y.T) # shape is (m, n)
dists = np.sqrt(x2 + y2 - 2*xy) # shape is (m, n) , m test ,n train
```

- get the predicted label
  - Sort the distances.
  - Take the lowest k distances.



```
closest_y = self.y_train[np.argsort(dists[i])][0:k]
```

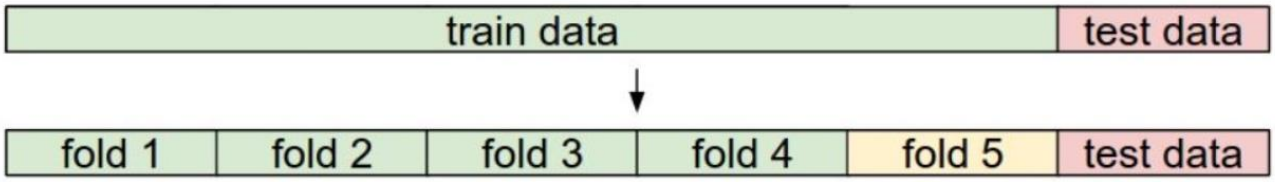
  - The predicted label is the most common label of the lowest k distances labels.



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

K-fold cross validation

Cross-validation is a resampling procedure used to evaluate machine learning models on a limited data sample. The procedure has a single parameter called k that refers to the number of groups that a given data sample is to be split into. As such, the procedure is often called k-fold cross-validation.
Cross-validation is primarily used in applied machine learning to estimate the skill of a machine learning model on unseen data.

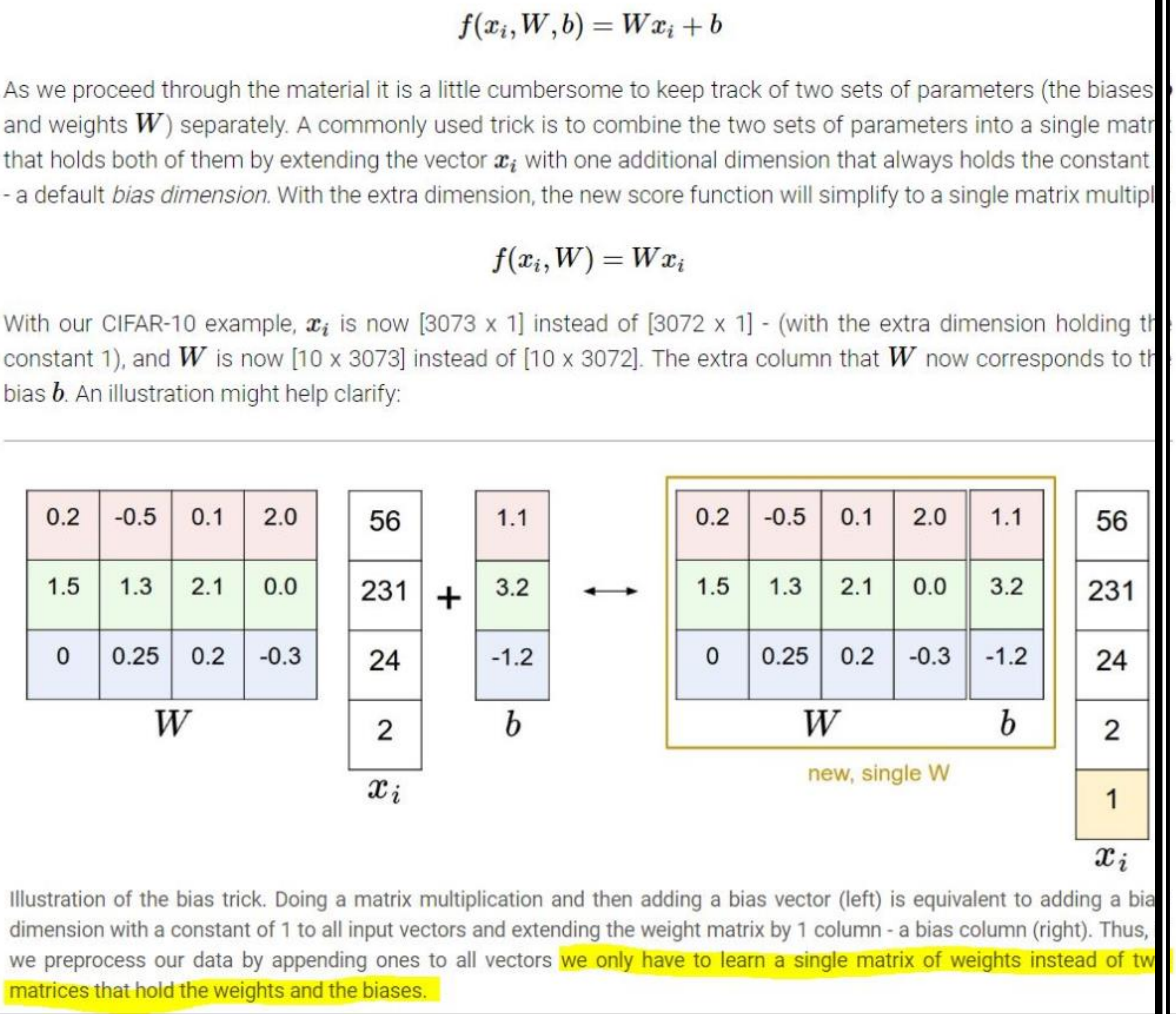


The general procedure is as follows:

- Shuffle the dataset randomly.
- Split the dataset into k groups
- For each unique group:
  - Take the group as a hold out or test data set
  - Take the remaining groups as a training data set
  - Fit a model on the training set and evaluate it on the test set
  - Retain the evaluation score and discard the model
- Summarize the skill of the model using the sample of model evaluation scores

2.2.SVM

Score



$$f(x_i, W, b) = Wx_i + b$$

As we proceed through the material it is a little cumbersome to keep track of two sets of parameters (the biases and weights W) separately. A commonly used trick is to combine the two sets of parameters into a single matr that holds both of them by extending the vector x_i with one additional dimension that always holds the constant - a default *bias dimension*. With the extra dimension, the new score function will simplify to a single matrix multipl

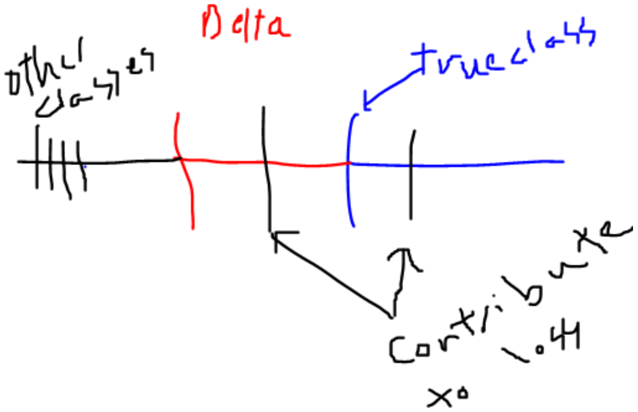
$$f(x_i, W) = Wx_i$$

With our CIFAR-10 example, x_i is now [3073 x 1] instead of [3072 x 1] - (with the extra dimension holding th constant 1), and W is now [10 x 3073] instead of [10 x 3072]. The extra column that W now corresponds to th bias b. An illustration might help clarify:

$$\underbrace{\begin{bmatrix}0.2 & -0.5 & 0.1 & 2.0\\ 1.5 & 1.3 & 2.1 & 0.0\\ 0 & 0.25 & 0.2 & -0.3\end{bmatrix}}_{W}\underbrace{\begin{bmatrix}56\\ 231\\ 24\\ 2\end{bmatrix}}_{x_i}+\underbrace{\begin{bmatrix}1.1\\ 3.2\\ -1.2\end{bmatrix}}_{b}\longleftrightarrow\underbrace{\begin{bmatrix}0.2 & -0.5 & 0.1 & 2.0 & 1.1\\ 1.5 & 1.3 & 2.1 & 0.0 & 3.2\\ 0 & 0.25 & 0.2 & -0.3 & -1.2\end{bmatrix}}_{W \quad b}\underbrace{\begin{bmatrix}56\\ 231\\ 24\\ 2\\ 1\end{bmatrix}}_{x_i}$$

Illustration of the bias trick. Doing a matrix multiplication and then adding a bias vector (left) is equivalent to adding a bia dimension with a constant of 1 to all input vectors and extending the weight matrix by 1 column - a bias column (right). Thus, we preprocess our data by appending ones to all vectors we only have to learn a single matrix of weights instead of tw matrices that hold the weights and the biases.

Loss



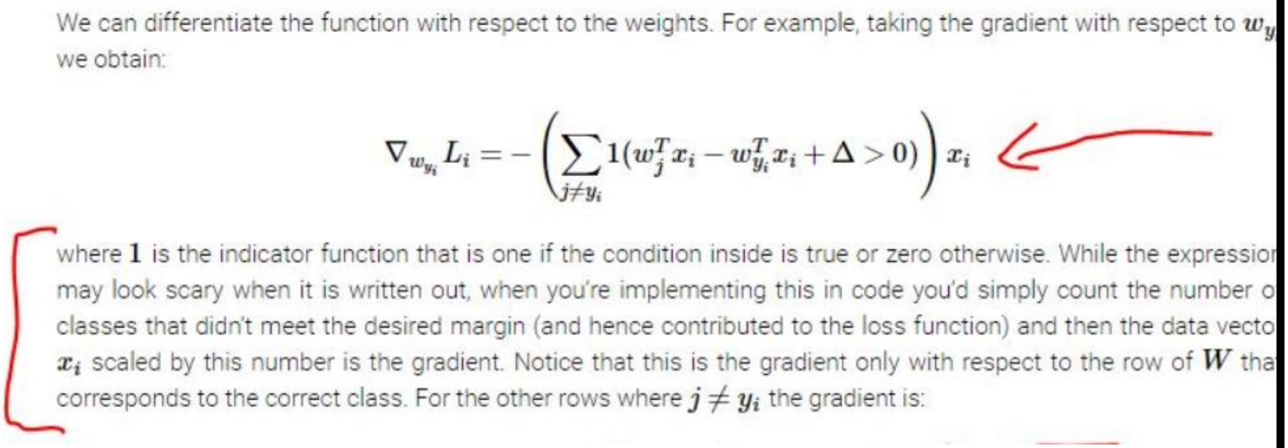
$$L_i = \sum_{j \neq y_i} \left[\max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta) \right]$$

Computing the gradient

There are two ways to compute the gradient: A slow, approximate but easy way **(numerical gradient)**, and a fast, exact but more error-prone way that requires calculus **(analytic gradient)**.

Computing the gradient analytically with Calculus:

That form calculates gradient for the true class:

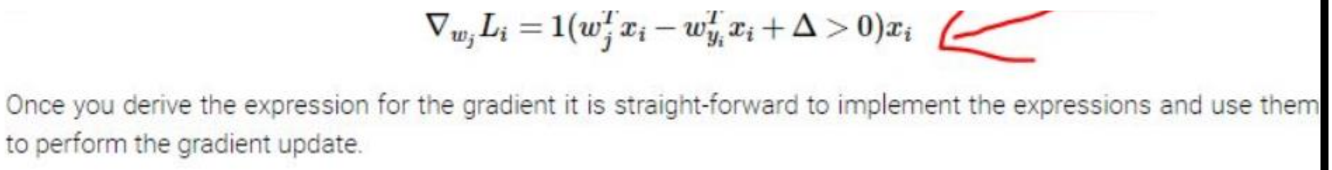


We can differentiate the function with respect to the weights. For example, taking the gradient with respect to w_y we obtain:

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

where $\mathbb{1}$ is the indicator function that is one if the condition inside is true or zero otherwise. While the expressio may look scary when it is written out, when you're implementing this in code you'd simply count the number o classes that didn't meet the desired margin (and hence contributed to the loss function) and then the data vecto x_i scaled by this number is the gradient. Notice that this is the gradient only with respect to the row of W tha corresponds to the correct class. For the other rows where $j \neq y_i$ the gradient is:

This form for the other classes, we would put 1*xi if that class contributed to the loss



$$\nabla_{w_j} L_i = \mathbb{1}(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) x_i$$

Once you derive the expression for the gradient it is straight-forward to implement the expressions and use them to perform the gradient update.

Regularization

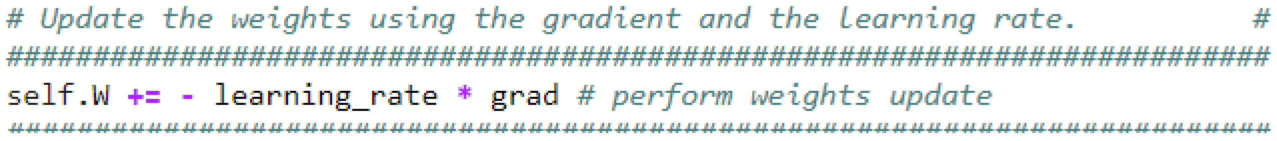
Used to prevent overfitting

L2 regularization:

For every weight w in the network, we add the term 0.5λw^2 to the objective, where λ is the regularization strength.

Update

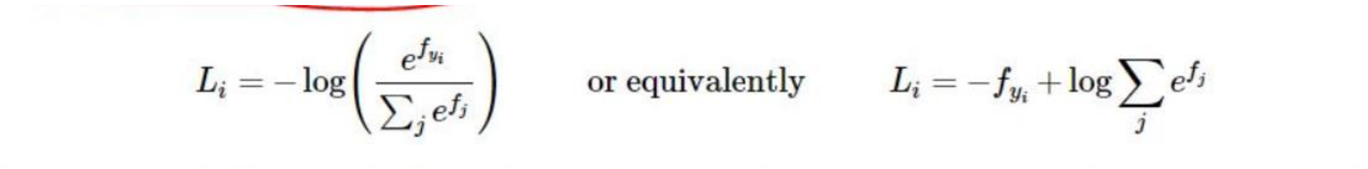
- Here we will notice that the true class previously was a -1*count of class contributing in loss*X
- That means that dw is -ve and when applying the following update we will get higher W for the true class
- That will give us higher score since score of class j f(j) = W[j]xi



```
# Update the weights using the gradient and the learning rate.          #
#########################################################################
self.W += - learning_rate * grad # perform weights update
#########################################################################
```

2.3.Softmax

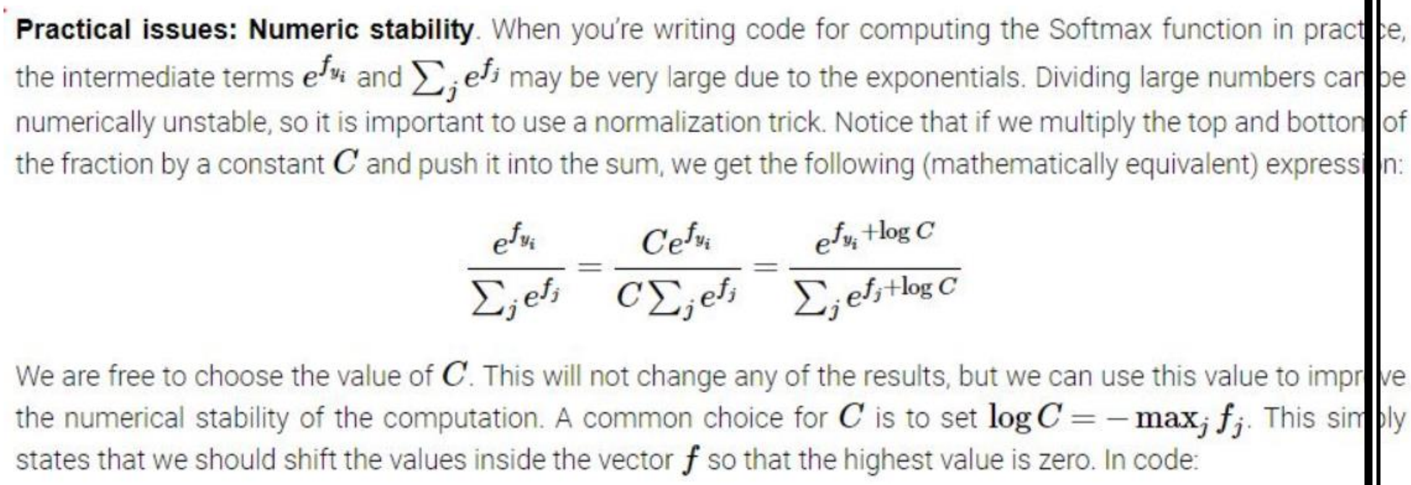
Loss



$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \qquad \text{or equivalently} \qquad L_i = -f_{y_i} + \log\sum_j e^{f_j}$$

Here the f denotes class score, so f_yi is true class score if the less than all other classes scores fj

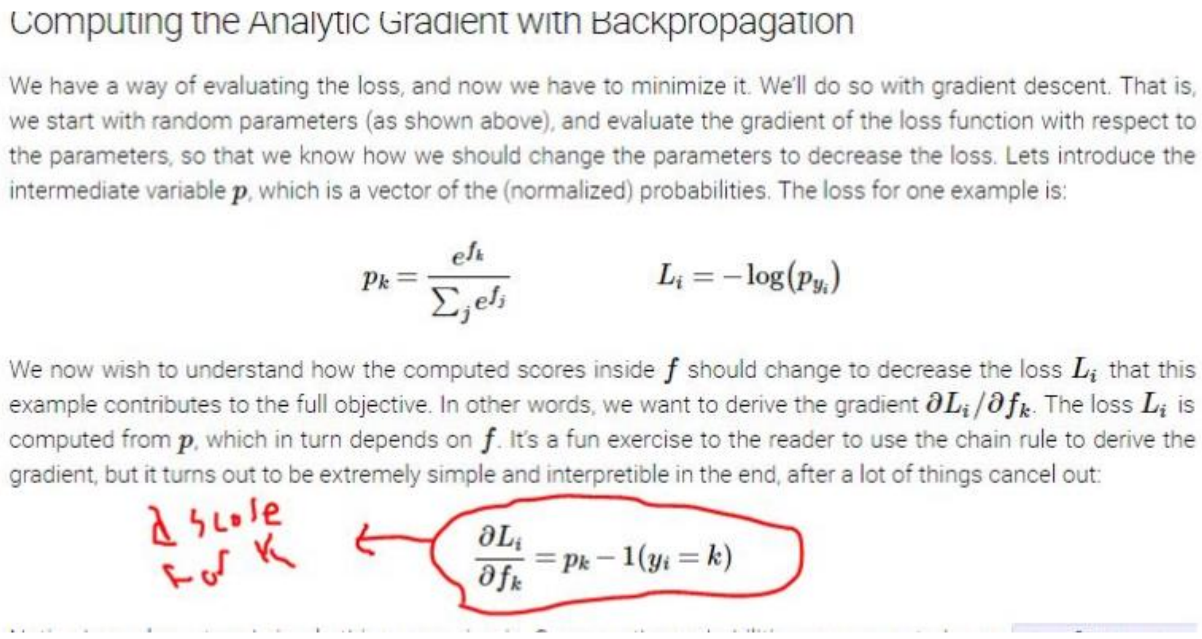
Then the Li would accumulate higher loss since log becomes -ve



Practical issues: Numeric stability. When you're writing code for computing the Softmax function in practice, the intermediate terms $e^{f_{y_i}}$ and $\sum_j e^{f_j}$ may be very large due to the exponentials. Dividing large numbers can be numerically unstable, so it is important to use a normalization trick. Notice that if we multiply the top and bottom of the fraction by a constant C and push it into the sum, we get the following (mathematically equivalent) expression:

$$\frac{e^{f_{y_i}}}{\sum_j e^{f_j}} = \frac{Ce^{f_{y_i}}}{C\sum_j e^{f_j}} = \frac{e^{f_{y_i}+\log C}}{\sum_j e^{f_j+\log C}}$$

We are free to choose the value of C. This will not change any of the results, but we can use this value to improve the numerical stability of the computation. A common choice for C is to set $\log C = -\max_j f_j$. This simply states that we should shift the values inside the vector f so that the highest value is zero. In code:



Computing the Analytic Gradient with Backpropagation

We have a way of evaluating the loss, and now we have to minimize it. We'll do so with gradient descent. That is, we start with random parameters (as shown above), and evaluate the gradient of the loss function with respect to the parameters, so that we know how we should change the parameters to decrease the loss. Lets introduce the intermediate variable p, which is a vector of the (normalized) probabilities. The loss for one example is:

$$p_k = \frac{e^{f_k}}{\sum_j e^{f_j}} \qquad\qquad L_i = -\log(p_{y_i})$$

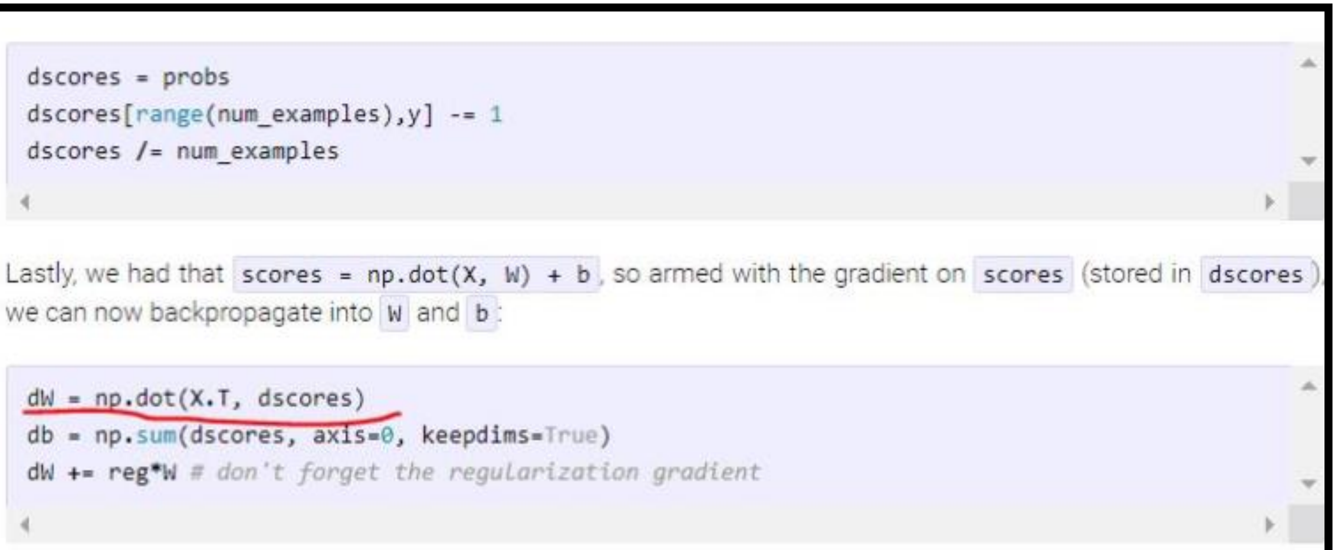
We now wish to understand how the computed scores inside f should change to decrease the loss L_i that this example contributes to the full objective. In other words, we want to derive the gradient $\partial L_i / \partial f_k$. The loss L_i is computed from p, which in turn depends on f. It's a fun exercise to the reader to use the chain rule to derive the gradient, but it turns out to be extremely simple and interpretible in the end, after a lot of things cancel out:

d score
for k

$$\boxed{\frac{\partial L_i}{\partial f_k} = p_k - \mathbb{1}(y_i = k)}$$

The previous partial derivative would be called dscores then to calculate the gradient

We simply get dot product X to dscores to get dW



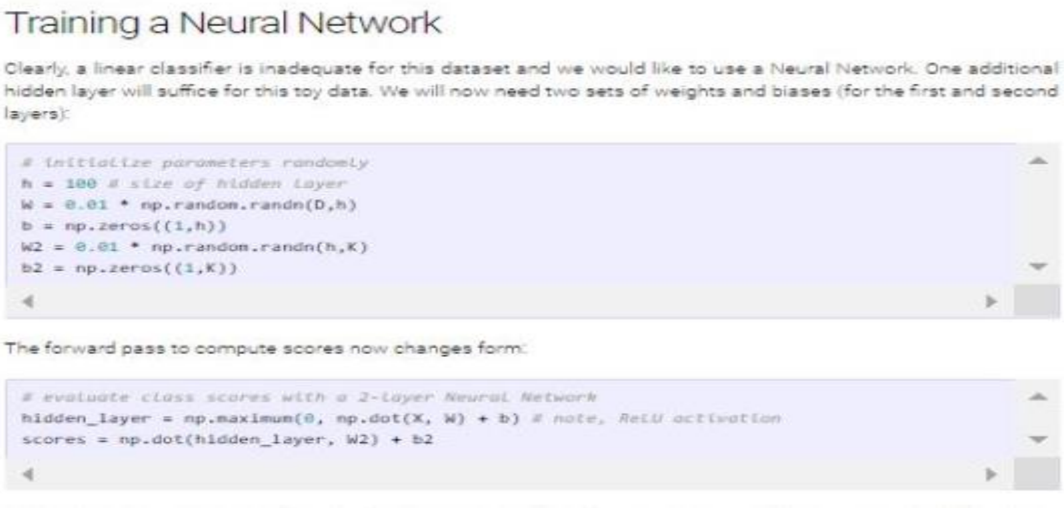
```
dscores = probs
dscores[range(num_examples),y] -= 1
dscores /= num_examples
```

Lastly, we had that `scores = np.dot(X, W) + b`, so armed with the gradient on `scores` (stored in `dscores`), we can now backpropagate into `W` and `b`:

```
dW = np.dot(X.T, dscores)
db = np.sum(dscores, axis=0, keepdims=True)
dW += reg*W # don't forget the regularization gradient
```

2.4.Two layer neural network

- Forward pass



Training a Neural Network

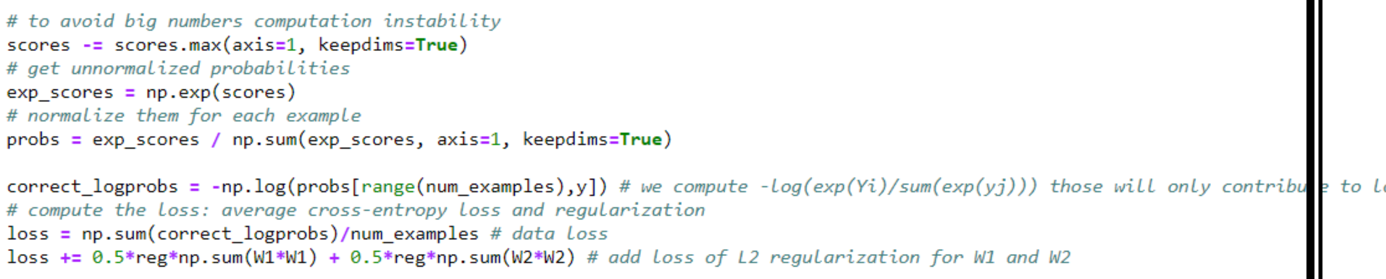
Clearly, a linear classifier is inadequate for this dataset and we would like to use a Neural Network. One additional hidden layer will suffice for this toy data. We will now need two sets of weights and biases (for the first and second layers):

```
# initialize parameters randomly
h = 100 # size of hidden layer
W = 0.01 * np.random.randn(D,h)
b = np.zeros((1,h))
W2 = 0.01 * np.random.randn(h,K)
b2 = np.zeros((1,K))
```

The forward pass to compute scores now changes form:

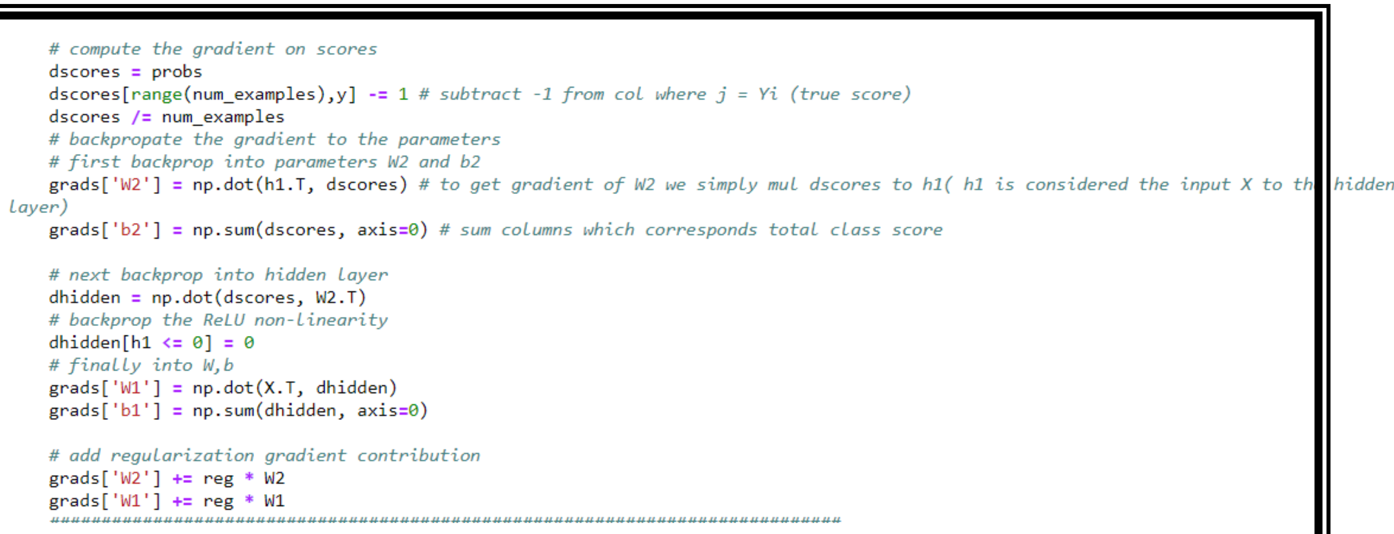
```
# evaluate class scores with a 2-layer Neural Network
hidden_layer = np.maximum(0, np.dot(X, W) + b) # note, ReLU activation
scores = np.dot(hidden_layer, W2) + b2
```

- Loss (softmax loss)



```
# to avoid big numbers computation instability
scores -= scores.max(axis=1, keepdims=True)
# get unnormalized probabilities
exp_scores = np.exp(scores)
# normalize them for each example
probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

correct_logprobs = -np.log(probs[range(num_examples),y]) # we compute -log(exp(Yi)/sum(exp(yj))) those will only contribu e to l
# compute the loss: average cross-entropy loss and regularization
loss = np.sum(correct_logprobs)/num_examples # data loss
loss += 0.5*reg*np.sum(W1*W1) + 0.5*reg*np.sum(W2*W2) # add loss of L2 regularization for W1 and W2
```

- Backward pass
    - H1 is considered input of the hidden layer
    - After getting probs at last layer
    - Get dscore by subtracting -1
    - Then gradient of last weights dw2 = h1(input of last layer)* dscores
    - Then to get the props of the hidden layer we multiply dscores to w2
    - Then we get it’s dscores (dhidden) by applying relU
    - Then to get gradient of W1 >> dw1 = X*dhidden



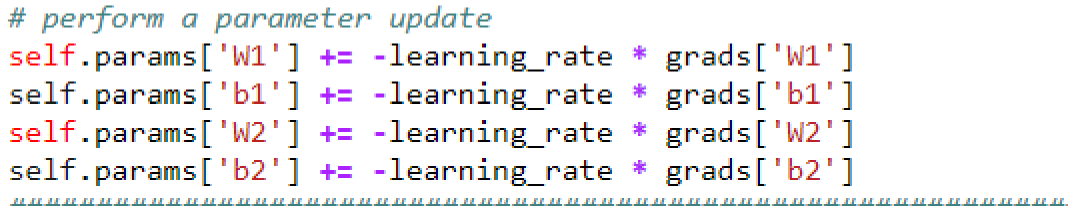
```
    # compute the gradient on scores
    dscores = probs
    dscores[range(num_examples),y] -= 1 # subtract -1 from col where j = Yi (true score)
    dscores /= num_examples
    # backpropate the gradient to the parameters
    # first backprop into parameters W2 and b2
    grads['W2'] = np.dot(h1.T, dscores) # to get gradient of W2 we simply mul dscores to h1( h1 is considered the input X to the hidden
layer)
    grads['b2'] = np.sum(dscores, axis=0) # sum columns which corresponds total class score

    # next backprop into hidden layer
    dhidden = np.dot(dscores, W2.T)
    # backprop the ReLU non-linearity
    dhidden[h1 <= 0] = 0
    # finally into W,b
    grads['W1'] = np.dot(X.T, dhidden)
    grads['b1'] = np.sum(dhidden, axis=0)

    # add regularization gradient contribution
    grads['W2'] += reg * W2
    grads['W1'] += reg * W1
    #############################################################################
```

- Train

  Create a random minibatch of training data and labels

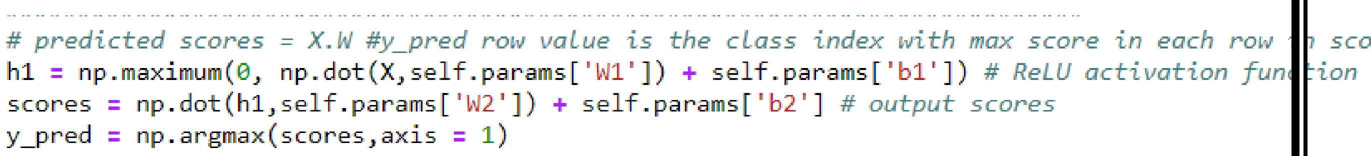
```
batch_indices = np.random.choice(num_train, batch_size)
X_batch = X[batch_indices]
y_batch = y[batch_indices]
```

  Use the gradients in the grads dictionary to update the parameters of the network



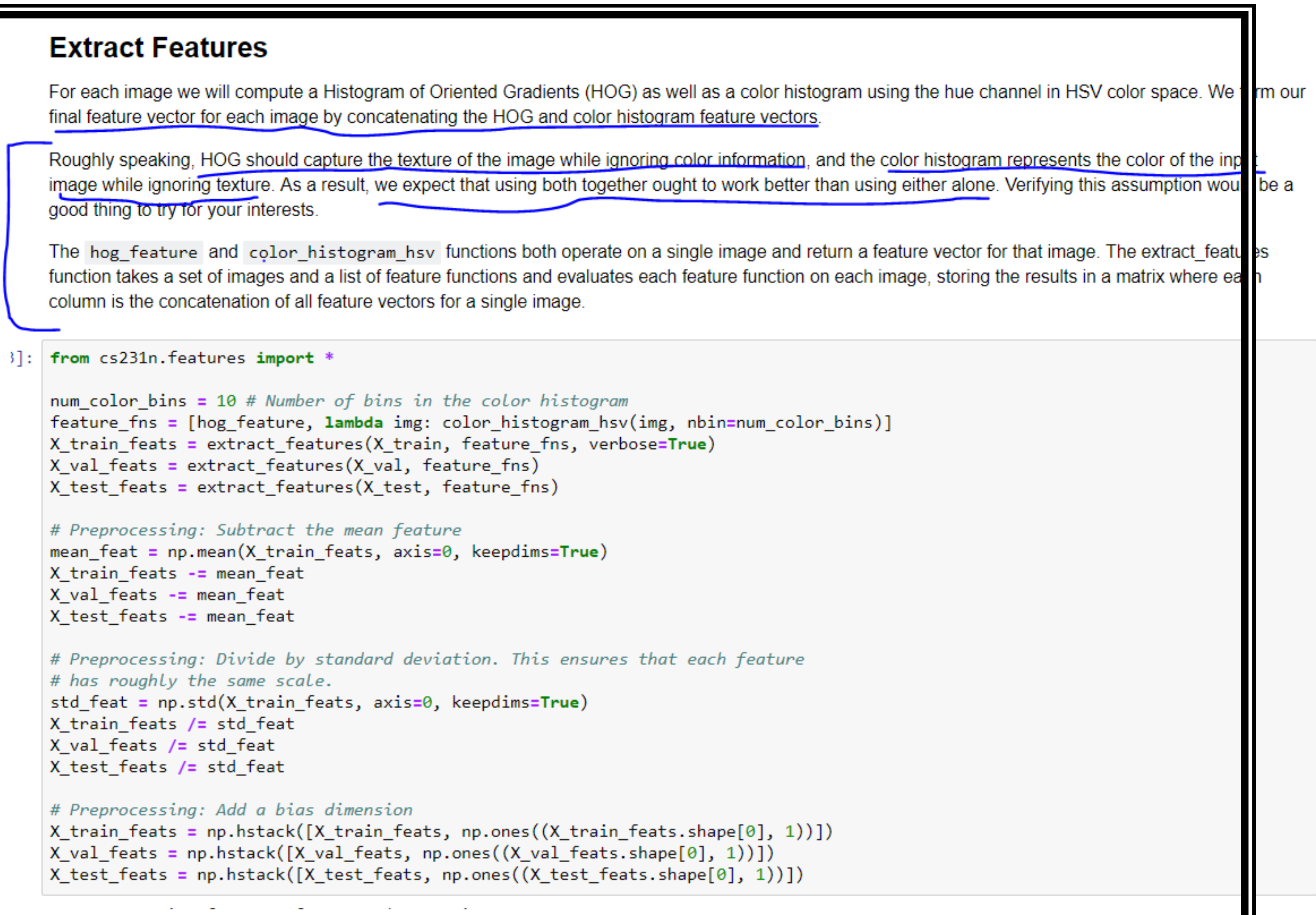
```
# perform a parameter update
self.params['W1'] += -learning_rate * grads['W1']
self.params['b1'] += -learning_rate * grads['b1']
self.params['W2'] += -learning_rate * grads['W2']
self.params['b2'] += -learning_rate * grads['b2']
#########################################################################
```

- Predict



```
#########################################################################
# predicted scores = X.W #y_pred row value is the class index with max score in each row in sco
h1 = np.maximum(0, np.dot(X,self.params['W1']) + self.params['b1']) # ReLU activation function
scores = np.dot(h1,self.params['W2']) + self.params['b2'] # output scores
y_pred = np.argmax(scores,axis = 1)
```

2.5. Feature Extraction:



Extract Features

For each image we will compute a Histogram of Oriented Gradients (HOG) as well as a color histogram using the hue channel in HSV color space. We form our final feature vector for each image by concatenating the HOG and color histogram feature vectors.

Roughly speaking, HOG should capture the texture of the image while ignoring color information, and the color histogram represents the color of the input image while ignoring texture. As a result, we expect that using both together ought to work better than using either alone. Verifying this assumption would be a good thing to try for your interests.

The `hog_feature` and `color_histogram_hsv` functions both operate on a single image and return a feature vector for that image. The extract_features function takes a set of images and a list of feature functions and evaluates each feature function on each image, storing the results in a matrix where each column is the concatenation of all feature vectors for a single image.

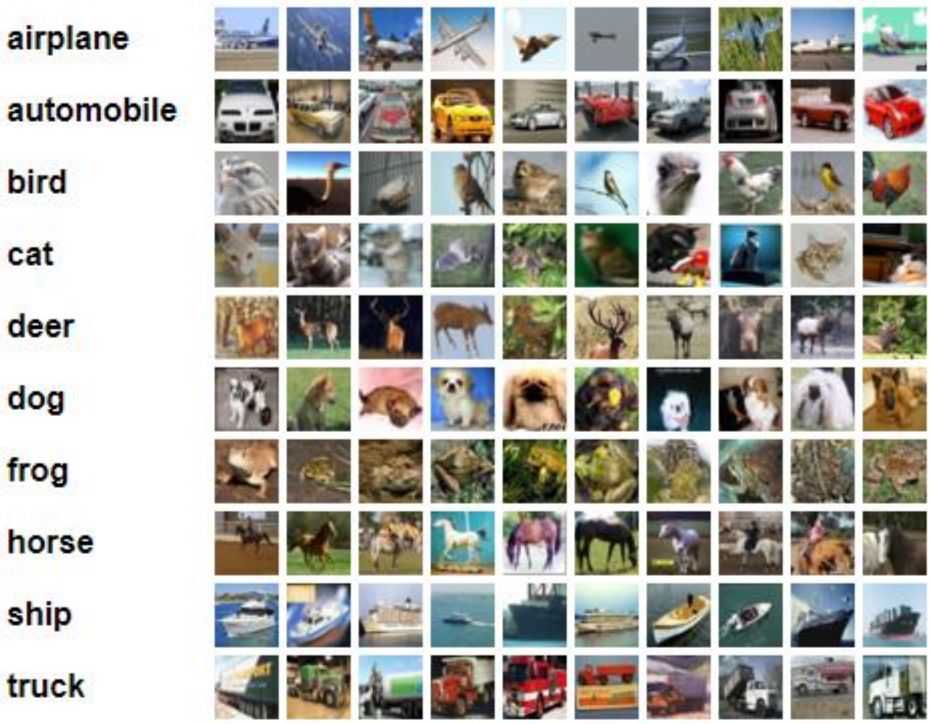
```
from cs231n.features import *

num_color_bins = 10 # Number of bins in the color histogram
feature_fns = [hog_feature, lambda img: color_histogram_hsv(img, nbin=num_color_bins)]
X_train_feats = extract_features(X_train, feature_fns, verbose=True)
X_val_feats = extract_features(X_val, feature_fns)
X_test_feats = extract_features(X_test, feature_fns)

# Preprocessing: Subtract the mean feature
mean_feat = np.mean(X_train_feats, axis=0, keepdims=True)
X_train_feats -= mean_feat
X_val_feats -= mean_feat
X_test_feats -= mean_feat

# Preprocessing: Divide by standard deviation. This ensures that each feature
# has roughly the same scale.
std_feat = np.std(X_train_feats, axis=0, keepdims=True)
X_train_feats /= std_feat
X_val_feats /= std_feat
X_test_feats /= std_feat

# Preprocessing: Add a bias dimension
X_train_feats = np.hstack([X_train_feats, np.ones((X_train_feats.shape[0], 1))])
X_val_feats = np.hstack([X_val_feats, np.ones((X_val_feats.shape[0], 1))])
X_test_feats = np.hstack([X_test_feats, np.ones((X_test_feats.shape[0], 1))])
```

3. Dataset

The CIFAR-10 dataset

The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

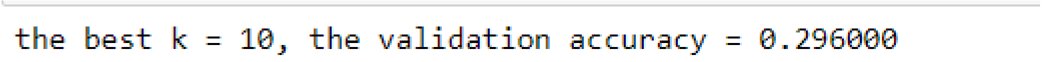
Here are the classes in the dataset, as well as 10 random images from each:



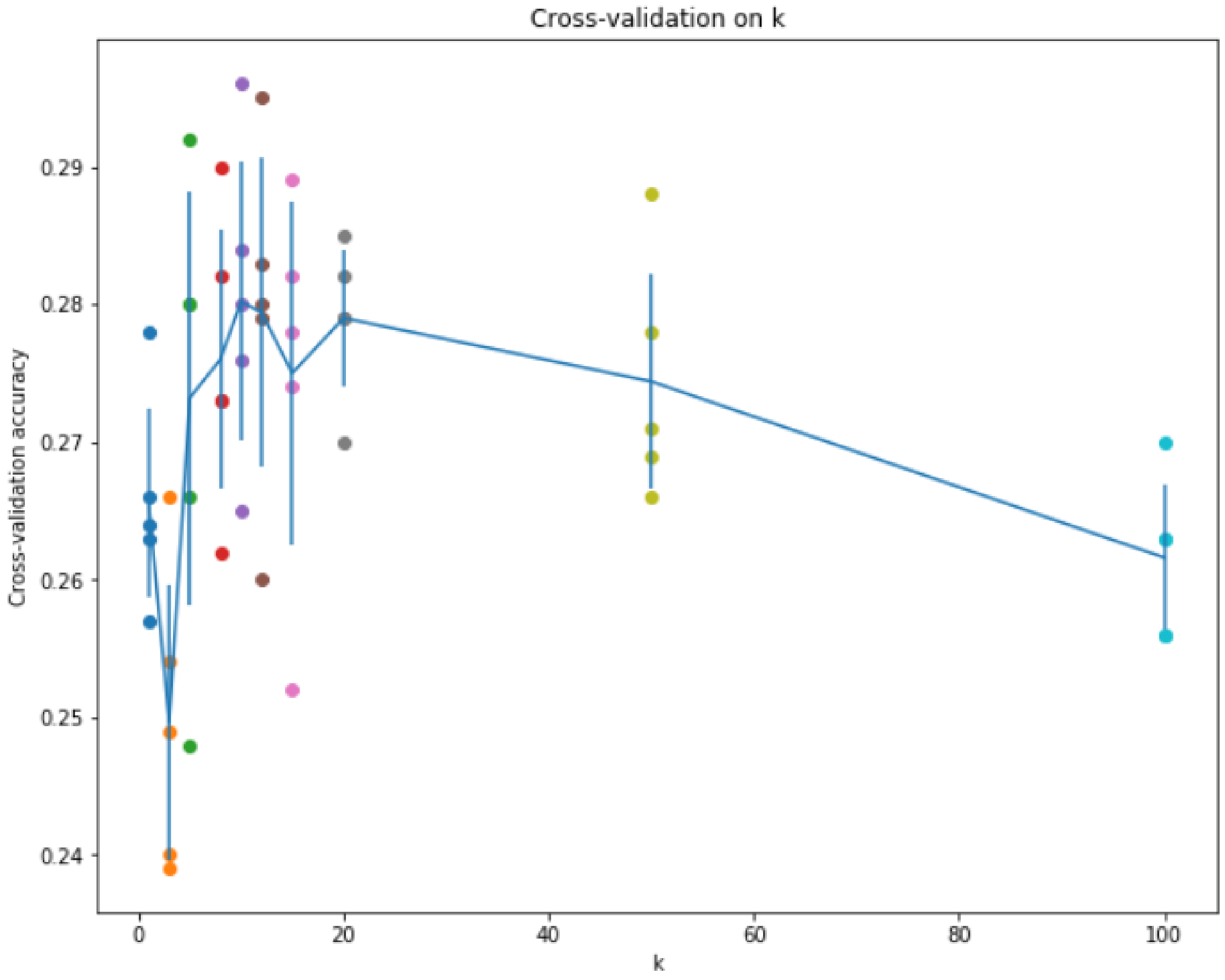
4. Results

4.1. KNN (Goal test score is 28%)

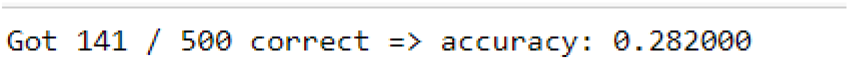
Validation score



```
the best k = 10, the validation accuracy = 0.296000
```



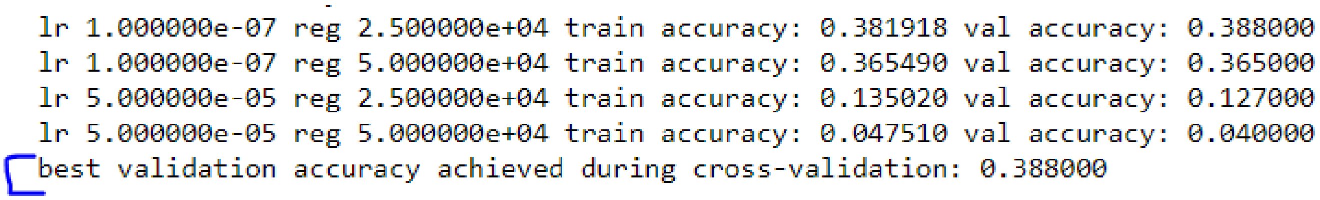
Test score



```
Got 141 / 500 correct => accuracy: 0.282000
```

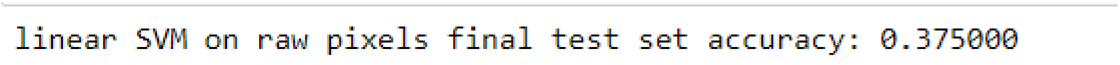
4.2. SVM (goal was val_score > 38%)

Validation score (we got 38.8%)



```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.381918 val accuracy: 0.388000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.365490 val accuracy: 0.365000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.135020 val accuracy: 0.127000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.047510 val accuracy: 0.040000
best validation accuracy achieved during cross-validation: 0.388000
```

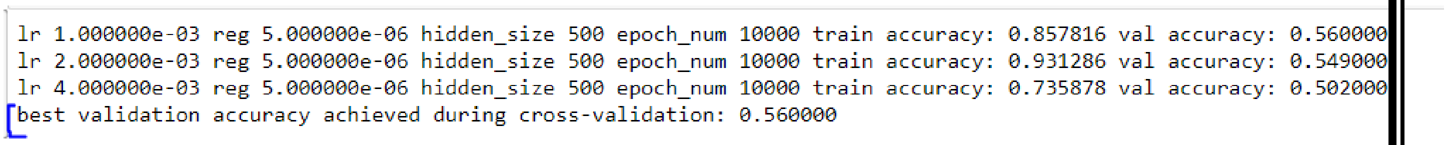
Test score (37.5%)



```
linear SVM on raw pixels final test set accuracy: 0.375000
```

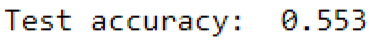
4.3. Neural Net (Goal was val_score > 48% .. the best was 52%)

Validation score (we got 56%)



```
lr 1.000000e-03 reg 5.000000e-06 hidden_size 500 epoch_num 10000 train accuracy: 0.857816 val accuracy: 0.560000
lr 2.000000e-03 reg 5.000000e-06 hidden_size 500 epoch_num 10000 train accuracy: 0.931286 val accuracy: 0.549000
lr 4.000000e-03 reg 5.000000e-06 hidden_size 500 epoch_num 10000 train accuracy: 0.735878 val accuracy: 0.502000
best validation accuracy achieved during cross-validation: 0.560000
```

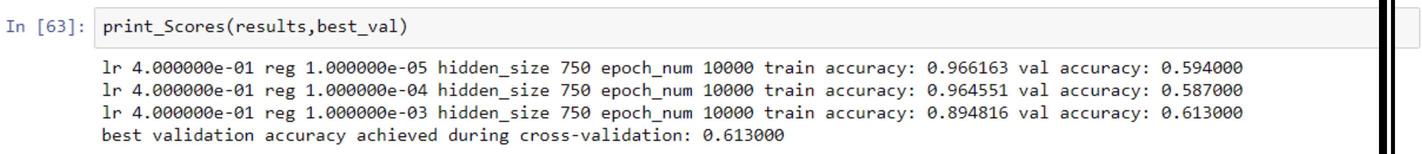
Test score (we got 55.3%)



```
Test accuracy:  0.553
```

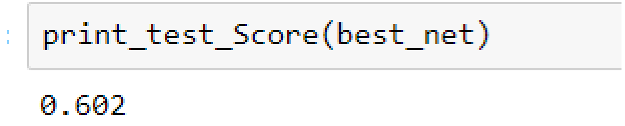
4.4. Feature Extraction(Goal was to achieve val_score>55% .. the best was 60%)

Validation score (we got 61.3%)



```
In [63]: print_Scores(results,best_val)

         lr 4.000000e-01 reg 1.000000e-05 hidden_size 750 epoch_num 10000 train accuracy: 0.966163 val accuracy: 0.594000
         lr 4.000000e-01 reg 1.000000e-04 hidden_size 750 epoch_num 10000 train accuracy: 0.964551 val accuracy: 0.587000
         lr 4.000000e-01 reg 1.000000e-03 hidden_size 750 epoch_num 10000 train accuracy: 0.894816 val accuracy: 0.613000
         best validation accuracy achieved during cross-validation: 0.613000
```

Test score (60.2%)



```
print_test_Score(best_net)

0.602
```