BLG 506E COMPUTER VISION ASSIGNMENT 2

Linear Classifier

We work on CIFAR10-dataset at this assignment. It has 50000 Train Data and 10000 Test Data. Firstly, we load CIFAR10 dataset. Then we visualize some examples from the dataset. We show 7 examples of training images from each class. Then we split the data into 49000 train, 1000 validation and 1000 test sets.

Then we make flat 3072 from 32*32*3. And shape of our training set (49000, 3072), shape of our validation set (1000, 3072), shape of our test. set (1000, 3072),

At “Preprocessing: subtract the mean image”, we compute the image that mean based on the training data and subtract the mean image from train and test data. Finally we append the bias dimension of ones.

SWM Classifier

At this section we compute swm loss.

Firstly, we add “compute_loss_naive” function at linear_svm.py file. Here the loss is calculated but the gradient is not calculated. We calculate gradient.

Computing the gradient analytically with Calculus

loss=max(0,something$-w_y*x$)

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

In the first (non-negative) case the loss "something$-w_y*$x" is linear in w_y, so the gradient is just the slope of this function of w_y, that is , $-$x.

In the second (negative) case the loss 0 is constant, so its derivative is also 0.

To write all this cases in one equation, we invent a function I(x), which equals 1 if x is true, and 0 otherwise. With this function, we can write

gradient=I(something$-w_y*x$>0)$*$(+x)

If “something$-w_y*x$”>0, the first multiplier equals 1, and gradient equals x. Otherwise, the first multiplier equals 0, and gradient as well. So I just rewrote the two cases in a single line.

gradient = sum(I(something - w_y*x[i] > 0) * (x[i]))

---

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:

            dW[:,j] += X[i] # j!=i case

            dW[:,y[i]] -= X[i] # j=i case because of its value must be 0

We have computed “`loss: 8.738247`”.

`Then we compute the loss and its gradient at W while regularization is zero and we compare numerically compute the gradient and analytically computed gradient.`

“`loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.0)`”

```
numerical: -10.896271 analytic: -10.896271, relative error: 2.159860e-11
numerical: 19.922841 analytic: 19.922841, relative error: 1.694342e-12
numerical: 6.180614 analytic: 6.180614, relative error: 8.507817e-11
numerical: 5.848896 analytic: 5.848896, relative error: 5.070573e-11
numerical: 26.728636 analytic: 26.728636, relative error: 1.393378e-12
numerical: 8.904048 analytic: 8.904048, relative error: 2.401301e-11
numerical: -8.875843 analytic: -8.875843, relative error: 4.111310e-11
numerical: -22.914664 analytic: -22.914664, relative error: 1.729483e-12
numerical: 1.728589 analytic: 1.728589, relative error: 1.253630e-10
numerical: 9.610851 analytic: 9.610851, relative error: 1.040999e-11
numerical: 9.312384 analytic: 9.312384, relative error: 1.072485e-11
numerical: 17.543361 analytic: 17.543361, relative error: 2.684671e-11
numerical: -10.548834 analytic: -10.548834, relative error: 8.060163e-13
numerical: -9.009409 analytic: -9.009409, relative error: 1.153042e-11
numerical: 7.455756 analytic: 7.455756, relative error: 3.837185e-12
numerical: 9.790559 analytic: 9.790559, relative error: 1.982856e-11
numerical: 2.932944 analytic: 2.932944, relative error: 3.842536e-11
numerical: 1.125820 analytic: 1.135684, relative error: 4.362042e-03
numerical: 6.891339 analytic: 6.891339, relative error: 5.720807e-12
numerical: -2.060628 analytic: -2.060628, relative error: 1.002699e-10
```

Inline Question 1: It is possible that once in a while a dimension in the gradcheck will not match exactly. What could such a discrepancy be caused by? Is it a reason for concern? What is a simple example in one dimension where a gradient check could fail? How would change the margin affect of the frequency of this happening? *Hint: the SVM loss function is not strictly speaking differentiable*

Answer*:* I thinks it is possible. How do we calculate? We make a calculation like "max (0, sj-syi +1)". Here, if "sj-syi + 1" is a negative number, we set the loss function to zero, but when this number turns positive, we take this number as the loss function. If the expression "sj-syi + 1" here is close to 0, we can see the error. For example, the expression "sj-syi + 1" is 0.0000003. Loss function and gradient out 0.0000003. But when the expression "sj-syi + 1" is -0.0000003, both the loss function and the gradient are 0. From such small values, margin can give minor errors.

Then we code “the function svm_loss_vectorized” and compute the naive loss.

Calculate vectorized version of the structured SVM loss:

$$L_i = \sum_{j \neq y_i} \left[\max(0, w_j^T x_i - w_{y_i}^T x_i + 1) \right]$$

```
num_classes = W.shape[1]
num_train = X.shape[0]


scores = X.dot(W)
correct_class_scores = scores[np.arange(num_train), y].reshape(num_train,1)
margins = np.maximum(0, scores - correct_class_scores +1)
margins[ np.arange(num_train), y] = 0 # like if j = y[i]


loss = margins.sum() / num_train #avarage loss
loss += reg * np.sum(W * W) # add regularization
```

Calculate vectorized version of the gradient for the structured SVM loss

---

```
margins[margins >0] =1 #we calculated margins when calculating loss above
margins_sum = margins.sum(axis=1)
margins[np.arange(num_train),y] -= margins_sum #subtract j == y[i] case
```

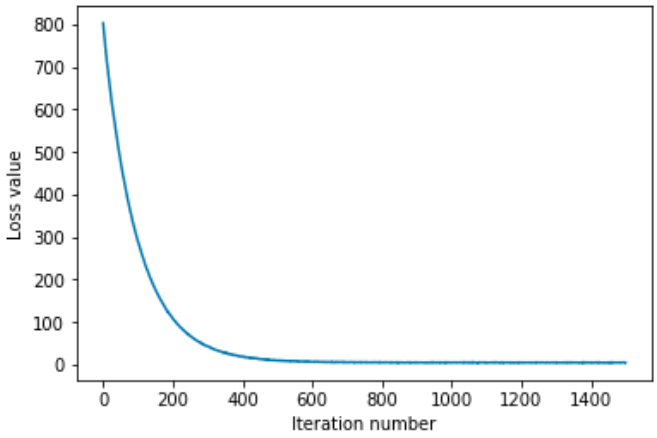
```
    dW = (X.T).dot(margins)


    dW /= num_train #average dW
    dW = dW + reg * 2 * W # Add regularization
```

Results:

```
Naive loss and gradient: computed in 0.261958s
Vectorized loss and gradient: computed in 0.007492s
difference: 0.000000
```

Stochastic Gradient Descent

We calculated loss and gradient with split batch train data. We use np.random.choice to generate indices.




```
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
          batch = np.random.choice(num_train,batch_size) #given as batch_size = 1500
          X_batch = X[batch,:]
          y_batch = y[batch]
   # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

we update W with learning data and grad.

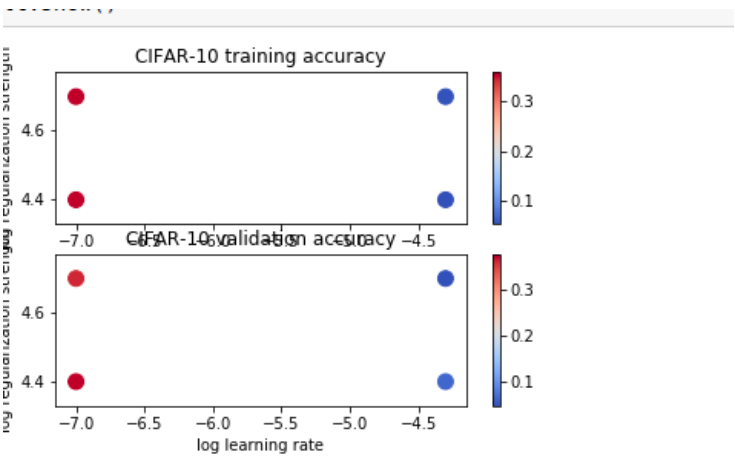
self.W -= learning_rate*grad

calculate best value accuracy and svm model

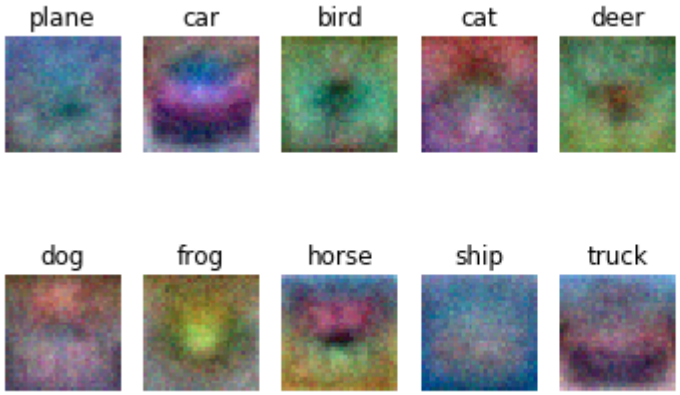
```
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

#given as svm = LinearSVM() and loss_hist = svm.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,

#num_iters=1500, verbose=True) and we create model with new parameters each iterasion.


for lr in learning_rates:
    for reg in regularization_strengths:
        svm = LinearSVM()
        lost_hist = svm.train(X_train, y_train, learning_rate = lr,reg= reg, num_iters = 500)
        y_train_pred = svm.predict(X_train) # Predict values for train set
        train_accuracy = np.mean(y_train == y_train_pred)  # Calculate train for set accuracy
        y_val_pred = svm.predict(X_val) # Predict values for validation set
        val_accuracy = np.mean(y_val_pred == y_val) # Calculate validation for set accuracy


        results[(lr,reg)] = train_accuracy, val_accuracy #store results dictionary.
        if(best_val < val_accuracy):
            best_val = val_accuracy
            best_svm = svm


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

```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.358612 val accuracy: 0.373000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.360857 val accuracy: 0.374000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.063735 val accuracy: 0.080000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.062286 val accuracy: 0.067000
best validation accuracy achieved during cross-validation: 0.374000
```

Then we visualize the cross-validation results



Finally we visualize the learned weights for each class.



Inline question 2 : Describe what your visualized SVM weights look like, and offer a brief explanation for why they look they way that they do.

Answer: The SVM weights look like a combination of images and it's color. Considering how we calculate SVM, we may be seeing a template that we calculate with 500-500 examples and weights. Each template here may contain information from all classes, mostly from their own classes.

Softmax exercise

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

We calculate softmax loss and gradient of softmax loss at this section.
p = np.exp(scores[i]) / np.sum(np.exp(scores[i]))

loss -= np.log(p[y[i]])

We calculate loss of softmax as above for 1:num_train.

And we calculate gradient of softmax loss

if i != j gradient of softmax = Xi*pj
if i = j, gradient of softmax = Xi*(1 - pi) so Xi - Xi*pi and subtract Xi from i!=j case

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

    num_train = X.shape[0]
    num_classes = W.shape[1]
    scores = X.dot(W)
    for i in range(num_train):
        p = np.exp(scores[i]) / np.sum(np.exp(scores[i]))
        loss -= np.log(p[y[i]])

        for j in range(num_classes):
            dW[:,j] += X[i] * p[j] #gradient of softmax = Xi*pj if i != j
        dW[:,y[i]] -= X[i]   #gradient of softmax = Xi*(1 - pi)  if i = j, so Xi - Xi*pi and subtract Xi from
i!=j case

    # Average loss and gradient
    loss /= num_train
    dW /= num_train

    # add Regularization to loss and gradient
    loss += reg * np.sum(W * W)
    dW += reg * 2 * W



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


loss: 2.336361
sanity check: 2.302585
```

Inline Question 1: Why do we expect our loss to be close to -log(0.1)? Explain briefly.**

Answer: Since the loss of Softmax is calculated with -log (p) and the probability that a class of 10 classes will come is 0.1, it is normal for -log (0.1) to be equal to our loss value.

Calculate vectorized version of the structured Softmax loss:

This time, we calculate without using a loop.

```
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    num_train = X.shape[0]
    scores = X.dot(W)
    p = np.exp(scores) / np.sum(np.exp(scores), axis = 1).reshape(-1, 1)
    loss -= np.sum(np.log(p[range(num_train), y]))


    p[range(num_train), y] += -1
    dW = X.T.dot(p)


    # Average loss and gradient
    loss /= num_train #avarage loss
    dW /= num_train  #avarage gradient


    # add Regularization to loss and gradient
    loss += reg * np.sum(W * W) #add regularization to loss
    dW += reg * 2 * W #add regularization to gradient


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

```
naive loss: 2.336361e+00 computed in 0.157919s
vectorized loss: 2.336361e+00 computed in 0.008885s
Loss difference: 0.000000
Gradient difference: 0.000000
```

calculate best value accuracy and softmax model

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


for lr in learning_rates:
    for reg in regularization_strengths:
        softmax = Softmax()


        lost_hist = softmax.train(X_train, y_train, learning_rate = lr,reg= reg, num_iters = 500)
```

```
        y_train_pred = softmax.predict(X_train) # Predict values for train set
        train_accuracy = np.mean(y_train == y_train_pred)  # Calculate train for set accuracy
        y_val_pred = softmax.predict(X_val) # Predict values for validation set
        val_accuracy = np.mean(y_val_pred == y_val) # Calculate validation for set accuracy

        results[(lr,reg)] = train_accuracy, val_accuracy #store results dictionary.
        if(best_val < val_accuracy):
            best_val = val_accuracy
            best_softmax = softmax

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

```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.305776 val accuracy: 0.310000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.306122 val accuracy: 0.329000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.322224 val accuracy: 0.335000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.306980 val accuracy: 0.328000
best validation accuracy achieved during cross-validation: 0.335000
```

```
softmax on raw pixels final test set accuracy: 0.334000
```

Inline Question 2: *True or False*

Suppose the overall training loss is defined as the sum of the per-datapoint loss over all training examples. It is possible to add a new datapoint to a training set that would leave the SVM loss unchanged, but this is not the case with the Softmax classifier loss.

Answer : True

Explanation: In the SVM if the new data point has a score that is out of the margin range from the correct class score the loss wouldn't change but in the Softmax loss if the score of the new added datapoint be close to +infinity it will adversely affect the loss, but definitely the loss of Softmax will change.(exponentials)