Cs231n assignment1 보고서

1. KNN

```
def compute_distances_two_loops(self, X):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train using a nested loop over both the training data and the
    test data.

    Inputs:
    - X: A numpy array of shape (num_test, D) containing test data.

    Returns:
    - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
      is the Euclidean distance between the ith test point and the jth training
      point.
    """
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    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, nor use np.linalg.norm().          #
            #####################################################################
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
            dists[i][j] =  np.sqrt(np.sum(np.square(X[i]-self.X_train[j])))

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

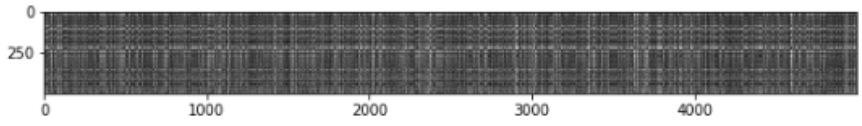
i번째 test 데이터와 j번째 train 데이터의 l2 distance를 구해줘서 dists[i][j]에 넣어준다.

주석에 제시된 것처럼 (num_test, num_train) (500, 5000)으로 결과가 나온다.

```
[12] # Open cs231n/classifiers/k_nearest_neighbor.py and implement
     # compute_distances_two_loops.

     # Test your implementation:
     dists = classifier.compute_distances_two_loops(X_test)
     print(dists.shape)
```

```
(500, 5000)
```

```
[13] # We can visualize the distance matrix: each row is a single test example and
     # its distances to training examples
     plt.imshow(dists, interpolation='none')
     plt.show()
```



Inline Question 1

Notice the structured patterns in the distance matrix, where some rows or columns are visibly brighter. (Note that with the default color scheme black indicates low distances while white indicates high distances.)

- What in the data is the cause behind the distinctly bright rows?
- What causes the columns?

Your Answer : *fill this in.*

- What in the data is the cause behind the distinctly bright rows?

  행(rows)부분은 test 데이터 1개와 여러 개의 train 데이터 간의 distance를 시각화한 것이다. 따라서 밝은 rows은 그 부분의 test가 여러 train과 먼 distance를 가진다고 유추해 볼 수 있다.

- What causes the columns?

  반대로 열 부분은 해당 train 데이터가 여러 test 데이터와 먼 distance를 가진다고 볼 수 있다.

```
def predict_labels(self, dists, k=1):
    """
    Given a matrix of distances between test points and training points,
    predict a label for each test point.

    Inputs:
    - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
      gives the distance betwen the ith test point and the jth training point.

    Returns:
    - y: A numpy array of shape (num_test,) containing predicted labels for the
      test data, where y[i] is the predicted label for the test point X[i].
    """
    num_test = dists.shape[0]
    y_pred = np.zeros(num_test)
    for i in range(num_test):
        # A list of length k storing the labels of the k nearest neighbors to
        # the ith test point.
        closest_y = []
        ########################################################################
        # TODO:                                                                #
        # Use the distance matrix to find the k nearest neighbors of the ith   #
        # testing point, and use self.y_train to find the labels of these      #
        # neighbors. Store these labels in closest_y.                          #
        # Hint: Look up the function numpy.argsort.                            #
        ########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

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

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        ########################################################################
        # TODO:                                                                #
        # Now that you have found the labels of the k nearest neighbors, you   #
        # need to find the most common label in the list closest_y of labels.  #
        # Store this label in y_pred[i]. Break ties by choosing the smaller    #
        # label.                                                               #
        ########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        y_pred[i]=max(set(closest_y), key = list(closest_y).count)


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

    return y_pred
```

함수 predict_labels 부분은 i번째 test 데이터에 대해 k개의 가까운 distance를 가지는 label을 가져와, 그 중 가장 빈도가 높은 라벨을 고르는 과정을 거친다.

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

➔ Argsort를 통해 앞에서 구한 dists를 통해 i번째 test 데이터와의 거리가 가장 짧은 순(오름차순)으로 정렬한다. 그 인덱스를 가져와 y_train에서 라벨을 가져온다. (argsort를 이용해 구현)

y_pred[i]=max(set(closest_y), key = list(closest_y).count)

➔ 이제 closest_y에는 i번째 test 데이터와 distance가 가장 짧은 순으로 k개의 라벨이 담겨있다. 이를 list로 변환해 count를 통해 빈도가 높은 라벨 하나를 뽑아 pred[i]에 넣어준다.

```
[32] # Now implement the function predict_labels and run the code below:
     # We use k = 1 (which is Nearest Neighbor).
     y_test_pred = classifier.predict_labels(dists, k=1)

     # Compute and print the fraction of correctly predicted examples
     num_correct = np.sum(y_test_pred == y_test)
     accuracy = float(num_correct) / num_test
     print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))

     Got 137 / 500 correct => accuracy: 0.274000
```

You should expect to see approximately `27%` accuracy. Now lets try out a larger `k`, say `k = 5`:

```
[33] y_test_pred = classifier.predict_labels(dists, k=5)
     num_correct = np.sum(y_test_pred == y_test)
     accuracy = float(num_correct) / num_test
     print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))

     Got 139 / 500 correct => accuracy: 0.278000
```

You should expect to see a slightly better performance than with `k = 1`.

제시된 것 대로 27%의 정확도를 보여주었으며 k=5일 때 약 0.004정도 상승한 것을 볼 수 있다.

Inline Question 2

We can also use other distance metrics such as L1 distance. For pixel values $p_{ij}^{(k)}$ at location (i, j) of some image I_k,

the mean μ across all pixels over all images is

$$\mu = \frac{1}{nhw}\sum_{k=1}^{n}\sum_{i=1}^{h}\sum_{j=1}^{w}p_{ij}^{(k)}$$

And the pixel-wise mean μ_{ij} across all images is

$$\mu_{ij} = \frac{1}{n}\sum_{k=1}^{n}p_{ij}^{(k)}.$$

The general standard deviation σ and pixel-wise standard deviation σ_{ij} is defined similarly.

Which of the following preprocessing steps will not change the performance of a Nearest Neighbor classifier that uses L1 distance? Select all that apply.

1. Subtracting the mean μ ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu$.)
2. Subtracting the per pixel mean μ_{ij} ($\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu_{ij}$.)
3. Subtracting the mean μ and dividing by the standard deviation σ.
4. Subtracting the pixel-wise mean μ_{ij} and dividing by the pixel-wise standard deviation σ_{ij}.
5. Rotating the coordinate axes of the data.

YourAnswer :

YourExplanation :

Answer : (1),(3)

Explanation : (1)번은 전체의 평균을 각 이미지의 픽셀에서 빼는 경우인데, 이는 L1 distance에 영향을 주지 않을 것이라고 생각된다. L1 distance를 구하는 과정에서 서로 빼게 되면 평균을 빼는 과정이 상쇄되기 때문이다. 마찬가지로 (3)번도 영향을 주지 않을 것이라고 생각된

다. (2)번은 각 픽셀의 평균을 빼는 과정인데, 이는 이미지마다 평균이 달라지므로 L1 distance가 변화될 것 이라고 생각된다. (4)도 마찬가지로 변화할 것이라고 생각된다. (5)은 45도로 회전할 경우 L1 distance가 변화하게 된다.

```
def compute_distances_one_loop(self, X):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train using a single loop over the test data.

    Input / Output: Same as compute_distances_two_loops
    """
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    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, :].                        #
        # Do not use np.linalg.norm().                                        #
        #######################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

         dists[i,:] = np.sqrt(np.sum(np.square(self.X_train - X[i,:]), axis = 1))

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

dists[i,:] = np.sqrt(np.sum(np.square(self.X_train - X[i,:]), axis = 1))

```
def compute_distances_no_loops(self, X):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train using no explicit loops.

    Input / Output: Same as compute_distances_two_loops
    """
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    #########################################################################
    # TODO:                                                                 #
    # Compute the l2 distance between all test points and all training      #
    # points without using any explicit loops, and store the result in      #
    # dists.                                                                #
    #                                                                       #
    # You should implement this function using only basic array operations; #
    # in particular you should not use functions from scipy,                #
    # nor use np.linalg.norm().                                             #
    #                                                                       #
    # HINT: Try to formulate the l2 distance using matrix multiplication    #
    #       and two broadcast sums.                                         #
    #########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    # print(np.sum(X**2, axis=1,keepdims=True).shape) (500,1)
    # print(-2*X.dot(self.X_train.T).shape) ()
    # print(np.sum(self.X_train**2,axis=1,keepdims=True).T.shape) (1,5000)
    dists = np.sqrt(np.sum(X**2, axis=1,keepdims=True)-2*X.dot(self.X_train.T)+np.sum(self.X_train**2,axis=1,keepdims=True).T)

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

dists = np.sqrt(np.sum(X**2, axis=1,keepdims=True)

-2*X.dot(self.X_train.T)+np.sum(self.X_train**2,axis=1,keepdims=True).T)

➔ L2 distance에서 곱셈식을 전개해서 반복문 없이 L2 distance를 구하도록 해보았다.

```
# Let's compare how fast the implementations are
def time_function(f, *args):
    """
    Call a function f with args and return the time (in seconds) that it took to execute.
    """
    import time
    tic = time.time()
    f(*args)
    toc = time.time()
    return toc - tic

two_loop_time = time_function(classifier.compute_distances_two_loops, X_test)
print('Two loop version took %f seconds' % two_loop_time)

one_loop_time = time_function(classifier.compute_distances_one_loop, X_test)
print('One loop version took %f seconds' % one_loop_time)

no_loop_time = time_function(classifier.compute_distances_no_loops, X_test)
print('No loop version took %f seconds' % no_loop_time)

# You should see significantly faster performance with the fully vectorized implementation!

# NOTE: depending on what machine you're using,
# you might not see a speedup when you go from two loops to one loop,
# and might even see a slow-down.
```

```
Two loop version took 28.047133 seconds
One loop version took 20.995912 seconds
No loop version took 0.422804 seconds
```

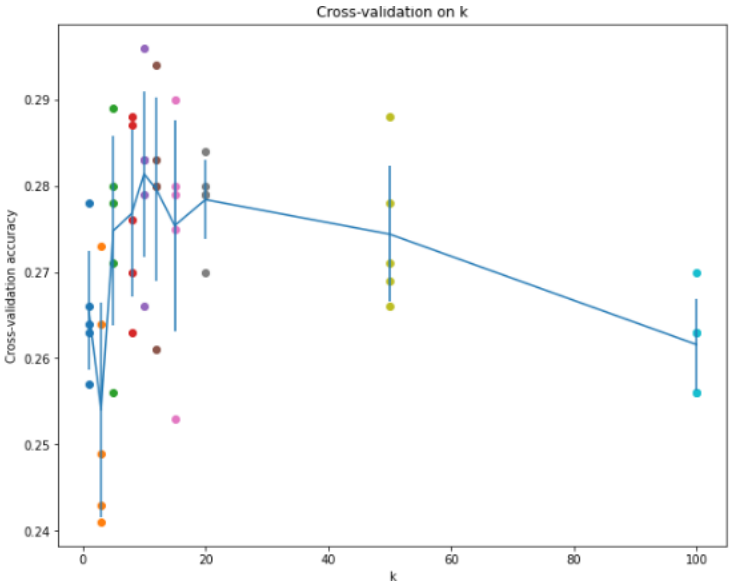
시간을 비교해보면 역시 루프를 덜 사용할 수 록 적은 시간을 가진다.

```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

X_train_folds = []
y_train_folds = []
################################################################################
# TODO:                                                                        #
# Split up the training data into folds. After splitting, X_train_folds and    #
# y_train_folds should each be lists of length num_folds, where                #
# y_train_folds[i] is the label vector for the points in X_train_folds[i].     #
# Hint: Look up the numpy array_split function.                                #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

X_train_folds = np.array_split(X_train, num_folds)
y_train_folds = np.array_split(y_train, num_folds)

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

교차검증을 위해 np.array_split으로 5분할 시켜준다.

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

for i in range(num_folds):
    X_train_batch= np.concatenate(X_train_folds[:i]+X_train_folds[i+1:])
    y_train_batch= np.concatenate(y_train_folds[:i]+y_train_folds[i+1:])
    X_val_batch= X_train_folds[i]
    y_val_batch= y_train_folds[i]

    classifier = KNearestNeighbor()
    classifier.train(X_train_batch, y_train_batch)


    dists_two = classifier.compute_distances_no_loops(X_val_batch)

    for k_choice in k_choices:
      y_val_batch_pred = classifier.predict_labels(dists_two, k=k_choice)
      num_correct = np.sum(y_val_batch_pred == y_val_batch)
      accuracy = float(num_correct) / y_val_batch.shape[0]
      if k_choice in k_to_accuracies:
         k_to_accuracies[k_choice].append(accuracy)
      else:
        k_to_accuracies[k_choice]= [accuracy]



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

i번째 데이터를 validation data로 해주고, 나머지 데이터는 train 데이터로 하여 반복문을 돌려 교차검증을 실시하도록 하였다.



K가 10일 때 약 28%으로 고 성능이 나왔다.

```
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 28% accuracy on the test data.
best_k = 10

classifier = KNearestNeighbor()
classifier.train(X_train, y_train)
y_test_pred = classifier.predict(X_test, k=best_k)

# Compute and display the accuracy
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

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

Inline Question 3

Which of the following statements about k-Nearest Neighbor (k-NN) are true in a classification setting, and for all k? Select all that apply.

1. The decision boundary of the k-NN classifier is linear.
2. The training error of a 1-NN will always be lower than or equal to that of 5-NN.
3. The test error of a 1-NN will always be lower than that of a 5-NN.
4. The time needed to classify a test example with the k-NN classifier grows with the size of the training set.
5. None of the above.

YourAnswer :

YourExplanation :

Answer : (2),(4)

Explanation : 먼저 k-NN classifier의 decision boundary는 linear 하지 않으므로 (1)은 거짓이다. (2)의 training error를 접목시키면 1-nn은 항상 자기 자신을 발견해서 error가 0이 될 것이지만 5-NN은 이를 확답할 수 없으므로 1-NN의 에러가 5-NN보다 작다. 따라서 참이다.

(3)의 경우 test 데이터는 학습된 것이 아니라 처음보는 데이터이므로 확답할 수 없다. 따라서 거짓이다. (4)의 경우, k-NN classifier는 test 데이터를 predict할 때마다 distance를 구해야 하므로 데이터의 크기가 커지면 시간이 늘어난다. 따라서 참이다.

2. SVM

Linear_svm.py 중 Svm_loss_naive

linear_svm.py

```
  # Rather that 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)*****

  #X = (500, 3073)=(N,D), X[i] = (3073), W = (3073,10)=(D,C)
  #y = (500) = (N)
  #dW = (3073,10)

  for i in range(num_train):
      scores = X[i].dot(W) #(3073)*(3073,10)= (N,C)
      correct_class_score = scores[y[i]]
      for j in range(num_classes):
          if j == y[i]:
            continue
          margin = scores[j] - correct_class_score + 1
          if margin > 0:
            dW[:, j] += X[i].T
            dW[:,y[i]] += -X[i].T
            loss += margin
              |
  dW /= num_train
  loss /= num_train

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


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

  return loss, dW
```

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

nction that is one if the condition inside is true or zero otherv
written out, when you're implementing this in code you'd sir
e desired margin (and hence contributed to the loss function)
is the gradient. Notice that this is the gradient only with resp
class. For the other rows where $j \neq y_i$ the gradient is:

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

오른쪽 식에 맞추어 w의 gradient를 계산하는 식을 작성해준다. 또한 마지막으로 L2 regularization을 추가해 마무리해준다.

```
# Evaluate the naive implementation of the loss we provided for you:
from cs231n.classifiers.linear_svm import svm_loss_naive
import time

# generate a random SVM weight matrix of small numbers
W = np.random.randn(3073, 10) * 0.0001

loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.000005)
print('loss: %f' % (loss, ))
```

```
loss: 9.248582
```

그러면 다음과 같이 loss 값을 구할 수 있게 된다.

다음 svm_loss_vecotrized는 이를 반복(루프)없이 빠르게 계산하는 함수이다.

```
#############################################################################
# TODO:                                                                     #
# Implement a vectorized version of the structured SVM loss, storing the    #
# result in loss.                                                           #
#############################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

scores = X.dot(W) #(N,C)
num_train = X.shape[0]
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 #j=y[i] continue 부분
loss = np.sum(margins)
loss = loss/num_train
loss += reg * np.sum(W * W)

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

#############################################################################
# TODO:                                                                     #
# Implement a vectorized version of the gradient for the structured SVM     #
# loss, storing the result in dW.                                           #
#                                                                           #
# Hint: Instead of computing the gradient from scratch, it may be easier    #
# to reuse some of the intermediate values that you used to compute the     #
# loss.                                                                     #
#############################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

margins[margins>0]=1
margin_count = np.sum(margins, axis=1)
margins[np.arange(num_train), y] = -margin_count
dW = np.dot(X.T, margins)
dW = dW/num_train
dW += reg*2*W
#https://mainpower4309.tistory.com/28 참고


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

Loss 부분은 해결하였지만, dw 부분을 구하는 것이 이해되지 않아 해당 링크를 참조하여 공부하였다. 먼저 margin>0이면 1로 초기화하고 이의 합을 구해 몇 번 loop가 도는 지 알 아낸다. 그 후 그 만큼 correct label에서 -하도록 설정 한 뒤, dot 연산을 하면 앞선 과정과 똑같은 결과가 나오게 된다.

```
[10] # Next implement the function svm_loss_vectorized; for now only compute the loss;
     # we will implement the gradient in a moment.
     tic = time.time()
     loss_naive, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
     toc = time.time()
     print('Naive loss: %e computed in %fs' % (loss_naive, toc - tic))

     from cs231n.classifiers.linear_svm import svm_loss_vectorized
     tic = time.time()
     loss_vectorized, _ = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
     toc = time.time()
     print('Vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

     # The losses should match but your vectorized implementation should be much faster.
     print('difference: %f' % (loss_naive - loss_vectorized))

     Naive loss: 9.248582e+00 computed in 0.216628s
     Vectorized loss: 9.248582e+00 computed in 0.017290s
     difference: -0.000000
```

그러면 다음과 같이 차이 없이 동일한 값을 반환하는 것을 볼 수 있다.

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*

Your Answer : *fill this in.*

그러한 discrepancy(불일치)는 사실 svm의 손실함수가 미분할 수 없기 때문에 발생한다. 손실함수에는 max()가 있는데 이는 연속은 하지만 미분할 수 없는 점인 x=0을 포함하고 있다. 따라서 이러한 불일치가 발생하게 된다.

linear_classifier.py

```
            ###########################################################################
            # TODO:                                                                   #
            # Sample batch_size elements from the training data and their             #
            # corresponding labels to use in this round of gradient descent.          #
            # Store the data in X_batch and their corresponding labels in             #
            # y_batch; after sampling X_batch should have shape (batch_size, dim)     #
            # and y_batch should have shape (batch_size,)                             #
            #                                                                         #
            # Hint: Use np.random.choice to generate indices. Sampling with           #
            # replacement is faster than sampling without replacement.                #
            ###########################################################################
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
            indices = np.random.choice(np.arange(num_train),batch_size)
            X_batch = X[indices]
            y_batch = y[indices]

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

            # evaluate loss and gradient
            loss, grad = self.loss(X_batch, y_batch, reg)
            loss_history.append(loss)

            # perform parameter update
            ###########################################################################
            # TODO:                                                                   #
            # Update the weights using the gradient and the learning rate.            #
            ###########################################################################
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

            self.W += -learning_rate*grad

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

            if verbose and it % 100 == 0:
                print("iteration %d / %d: loss %f" % (it, num_iters, loss))

        return loss_history
```

Stochastic gradient descent를 위해 np.random.choice를 이용해 x_batch와 y_batch를 선언해준다. 그 후 learning rate와 gradient를 이용해 w를 업데이트 해준다.

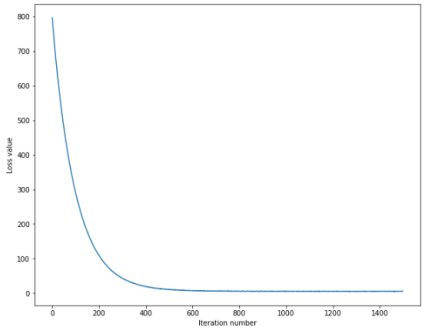
```python
def predict(self, X):
    """
    Use the trained weights of this linear classifier to predict labels for
    data points.

    Inputs:
    - X: A numpy array of shape (N, D) containing training data; there are N
      training samples each of dimension D.

    Returns:
    - y_pred: Predicted labels for the data in X. y_pred is a 1-dimensional
      array of length N, and each element is an integer giving the predicted
      class.
    """
    y_pred = np.zeros(X.shape[0])
    ###########################################################################
    # TODO:                                                                   #
    # Implement this method. Store the predicted labels in y_pred.            #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

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

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

Predict 함수는 X와 W를 dot 연산 해준다. 이 중에서 가장 높은 값을 가지는 인텍스를 불러와 y_pred에 저장 후 반환 해준다.

```python
[12] # In the file linear_classifier.py, implement SGD in the function
     # LinearClassifier.train() and then run it with the code below.
     from cs231n.classifiers import LinearSVM
     svm = LinearSVM()
     tic = time.time()
     loss_hist = svm.train(X_train, y_train, learning_rate=1e-7, reg=2.5e4,
                           num_iters=1500, verbose=True)
     toc = time.time()
     print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 796.526602
iteration 100 / 1500: loss 289.561476
iteration 200 / 1500: loss 108.681137
iteration 300 / 1500: loss 43.247970
iteration 400 / 1500: loss 19.458556
iteration 500 / 1500: loss 10.965502
iteration 600 / 1500: loss 6.845959
iteration 700 / 1500: loss 5.407618
iteration 800 / 1500: loss 5.571565
iteration 900 / 1500: loss 5.568017
iteration 1000 / 1500: loss 5.409647
iteration 1100 / 1500: loss 5.325478
iteration 1200 / 1500: loss 5.376328
iteration 1300 / 1500: loss 4.691358
iteration 1400 / 1500: loss 5.068792
That took 11.990225s
```

```python
[13] # A useful debugging strategy is to plot the loss as a function of
     # iteration number:
     plt.plot(loss_hist)
     plt.xlabel('Iteration number')
     plt.ylabel('Loss value')
     plt.show()
```




```python
[14] # Write the LinearSVM.predict function and evaluate the performance on both the
     # training and validation set
     y_train_pred = svm.predict(X_train)
     print('training accuracy: %f' % (np.mean(y_train == y_train_pred), ))
     y_val_pred = svm.predict(X_val)
     print('validation accuracy: %f' % (np.mean(y_val == y_val_pred), ))
```

```
training accuracy: 0.361918
validation accuracy: 0.377000
```

그럼 다음과 같은 결과가 나오게 된다.

또한 validation set을 이용해 최적의 hyper parameter를 구한 뒤 이를 적용시킬 수 도 있다.

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

#result 구하기 (learning_rate, regularization_strength) = (training_accuracy, validation_accuracy)
for lr in learning_rates:
  for reg in regularization_strengths:
    svm = LinearSVM()
    svm.train(X_train, y_train, learning_rate=lr, reg=reg,
                      num_iters=1500, verbose=True)
    y_train_pred = svm.predict(X_train)
    acc_tr = np.mean(y_train == y_train_pred)
    y_val_pred = svm.predict(X_val)
    acc_val = np.mean(y_val == y_val_pred)
    if best_val < acc_val:
      best_val = acc_val
      best_svm=svm
    results[(lr, reg)] = (acc_tr, acc_val)

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

루프를 돌며 learning rate와 reg를 선택한뒤, 이를 훈련 한 뒤 validation set에 적용해 최고의 정확도를 나타내는 모델을 저장한다.

```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.367245 val accuracy: 0.373000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.355939 val accuracy: 0.364000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.066694 val accuracy: 0.054000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.100265 val accuracy: 0.087000
best validation accuracy achieved during cross-validation: 0.373000
```

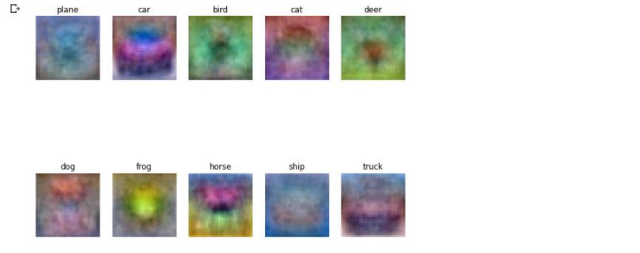
그럼 다음과 같이 약 37의 정확도를 보인다.

이를 실제 test set에 테스트 해보면 약 35%의 정확도를 보인다.

```
[17] # Evaluate the best svm on test set
     y_test_pred = best_svm.predict(X_test)
     test_accuracy = np.mean(y_test == y_test_pred)
     print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)

     linear SVM on raw pixels final test set accuracy: 0.349000
```

학습된 weight를 시각화하면 다음과 같이 나오게 되는데

```
# 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])
```



Inline question 2

Describe what your visualized SVM weights look like, and offer a brief explanation for why they look the way they do.

YourAnswer : fill this in

해당 가중치는 해당 레이블에 있는 데이터가 평균적으로 보이는 모양을 띄고 있다. w와 데이터를 내적하면 나오는 score가 그 레이블에서 높게 나와야 하기 때문이다.

3. Softmax

Softmax의 loss와 gradient는 다음 과정을 통해 구할 수 있다.

Softmax_loss_naive

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

num_classes = W.shape[1]
num_train = X.shape[0]
for i in range(num_train):
  scores = X[i].dot(W)
  scores -= np.max(scores)
  loss += -scores[y[i]] + np.log(np.sum(np.exp(scores)))
  for j in range(num_classes):
    #정답 레이블(y[i])가 아닌 경우
    dW[:, j] += X[i]*(np.exp(scores[j])/np.sum(np.exp(scores)))
  #정답 레이블(y[i])인 경우
  dW[:,y[i]] += -X[i]
dW /= num_train
loss /= num_train

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

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

return loss, dW
```

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

해당 식에 맞추어 loss와 dw를 구해준다. Loss는 오른쪽 식으로 구해주었고, dw는 정답 레이블인 경우, 아닐 경우를 나누어 구해주었다.

Loss 식을 풀어보면 -f_yi + log(e^f0 + e^f1 +... e^fj)가 되는데 이를 W_i에 대해서 미분하면 i가 정답이 아닌 경우에는 dw = x*softmax, 정답인경우에는 dw = -x + x*softmax 가 된다.

```
[5]  # First implement the naive softmax loss function with nested loops.
     # Open the file cs231n/classifiers/softmax.py and implement the
     # softmax_loss_naive function.

     from cs231n.classifiers.softmax import softmax_loss_naive
     import time

     # Generate a random softmax weight matrix and use it to compute the loss.
     W = np.random.randn(3073, 10) * 0.0001
     loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

     # As a rough sanity check, our loss should be something close to -log(0.1).
     print('loss: %f' % loss)
     print('sanity check: %f' % (-np.log(0.1)))

     loss: 2.342792
     sanity check: 2.302585
```

```
# Complete the implementation of softmax_loss_naive and implement a (naive)
# version of the gradient that uses nested loops.
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

# As we did for the SVM, use numeric gradient checking as a debugging tool.
# The numeric gradient should be close to the analytic gradient.
from cs231n.gradient_check import grad_check_sparse
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)

# similar to SVM case, do another gradient check with regularization
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 5e1)
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 5e1)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)
```

```
numerical: 2.553467 analytic: 2.553467, relative error: 7.899575e-09
numerical: 0.388968 analytic: 0.388968, relative error: 2.304584e-08
numerical: 1.213103 analytic: 1.213103, relative error: 6.896397e-08
numerical: 0.045426 analytic: 0.045426, relative error: 1.691022e-06
numerical: -1.242787 analytic: -1.242787, relative error: 3.396805e-09
numerical: -1.221818 analytic: -1.221818, relative error: 2.964947e-08
numerical: -1.055135 analytic: -1.055136, relative error: 6.235704e-08
numerical: 2.174725 analytic: 2.174726, relative error: 1.026504e-08
numerical: 3.665958 analytic: 3.665958, relative error: 4.125389e-09
numerical: 0.247048 analytic: 0.247048, relative error: 5.422815e-08
numerical: -1.954185 analytic: -1.954185, relative error: 1.044108e-08
numerical: 0.024351 analytic: 0.024351, relative error: 1.250222e-06
```

다음과 같은 결과가 나오게 된다.

이를 vectorized 하면 다음과 같이 표현할 수 있다.

```
#############################################################################
# TODO: Compute the softmax loss and its gradient using no explicit loops.  #
# Store the loss in loss and the gradient in dW. If you are not careful     #
# here, it is easy to run into numeric instability. Don't forget the        #
# regularization!                                                           #
#############################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
num_classes = W.shape[1]
num_train = X.shape[0]
scores = X.dot(W)
scores -= np.max(scores,axis=1).reshape(-1,1)
loss = -np.sum(scores[np.arange(num_train), y]) + np.sum(np.log(np.sum(np.exp(scores),axis=1)))

softmax = np.exp(scores)/np.sum(np.exp(scores), axis=1).reshape(-1,1)
softmax[np.arange(num_train), y] -=1
dW = X.T.dot(softmax)

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


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

위의 두 과정을 비교해보면 동일하다는 것을 알 수 있다.

```
# Now that we have a naive implementation of the softmax loss function and its gradient,
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version should be
# much faster.
tic = time.time()
loss_naive, grad_naive = softmax_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('naive loss: %e computed in %fs' % (loss_naive, toc - tic))

from cs231n.classifiers.softmax import softmax_loss_vectorized
tic = time.time()
loss_vectorized, grad_vectorized = softmax_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

# As we did for the SVM, we use the Frobenius norm to compare the two versions
# of the gradient.
grad_difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print('Loss difference: %f' % np.abs(loss_naive - loss_vectorized))
print('Gradient difference: %f' % grad_difference)
```

```
naive loss: 2.342792e+00 computed in 0.179271s
vectorized loss: 2.342792e+00 computed in 0.018867s
Loss difference: 0.000000
Gradient difference: 0.000000
```

Inline Question 1

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

Your Answer : *Fill this in*

클래스의 수가 10이기 때문에 통상적으로 10%의 예측값을 기대할 수 있다. Softmax의 loss는 예측 확률값에 -log() 해주기 때문에 다음과 같은 loss를 기대해볼 수 있다.

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.

Your Answer :

Your Explanation :

True, svm은 margin을 이용해 일정 기준을 만족하면 개선되지 않지만, softmax는 확률이 1이 될때까지 개선하기 때문이다.

4. Two-Layer Neural Network

Layer.py

Affine_forward

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

out = x.reshape(x.shape[0],-1).dot(w)+b


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

Affine _backward

```
###########################################################################
# TODO: Implement the affine backward pass.                               #
###########################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

dx = dout.dot(w.T).reshape(x.shape)
dw = x.reshape(x.shape[0],-1).T.dot(dout)
db = np.sum(dout,axis=0)

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

Relu_forward

$f(x) = \max(0, x)$. 해당 식에 맞게 작성해준다.

```
###########################################################################
# TODO: Implement the ReLU forward pass.                                  #
###########################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

out = np.maximum(x, 0)

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

Relu_backward

```
###########################################################################
# TODO: Implement the ReLU backward pass.                                 #
###########################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

dx = dout*np.array(x>0)

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

Affine layer:forward

다음과 같이 올바른 결과가 나온 것을 볼 수 있다.

▾ Affine layer: forward

Open the file cs231n/layers.py and implement the affine_forward function.

Once you are done you can test your implementaion by running the following:

```
[4]  # Test the affine_forward function

     num_inputs = 2
     input_shape = (4, 5, 6)
     output_dim = 3

     input_size = num_inputs * np.prod(input_shape)
     weight_size = output_dim * np.prod(input_shape)

     x = np.linspace(-0.1, 0.5, num=input_size).reshape(num_inputs, *input_shape)
     w = np.linspace(-0.2, 0.3, num=weight_size).reshape(np.prod(input_shape), output_dim)
     b = np.linspace(-0.3, 0.1, num=output_dim)

     out, _ = affine_forward(x, w, b)
     correct_out = np.array([[ 1.49834967,  1.70660132,  1.91485297],
                             [ 3.25553199,  3.5141327,   3.77273342]])

     # Compare your output with ours. The error should be around e-9 or less.
     print('Testing affine_forward function:')
     print(out.shape)
     print(correct_out.shape)
     print('difference: ', rel_error(out, correct_out))

     Testing affine_forward function:
     (2, 3)
     (2, 3)
     difference:  9.769849468192957e-10
```

Affine layer: backward

Now implement the `affine_backward` function and test your implementation using numeric grad

```
# Test the affine_backward function
np.random.seed(231)
x = np.random.randn(10, 2, 3)
w = np.random.randn(6, 5)
b = np.random.randn(5)
dout = np.random.randn(10, 5)

dx_num = eval_numerical_gradient_array(lambda x: affine_forward(x, w, b)[0], x, dou
dw_num = eval_numerical_gradient_array(lambda w: affine_forward(x, w, b)[0], w, dou
db_num = eval_numerical_gradient_array(lambda b: affine_forward(x, w, b)[0], b, dou

_, cache = affine_forward(x, w, b)
dx, dw, db = affine_backward(dout, cache)

# The error should be around e-10 or less
print('Testing affine_backward function:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_backward function:
dx error:  5.399100368651805e-11
dw error:  9.904211865398145e-11
db error:  2.4122867568119087e-11
```

ReLU activation: forward

Implement the forward pass for the ReLU activation function in the `relu_forward` function and test following:

```
[6] # Test the relu_forward function

    x = np.linspace(-0.5, 0.5, num=12).reshape(3, 4)

    out, _ = relu_forward(x)
    correct_out = np.array([[ 0.,          0.,          0.,          0.,        ],
                            [ 0.,          0.,          0.04545455,  0.13636364,],
                            [ 0.22727273,  0.31818182,  0.40909091,  0.5,       ]])

    # Compare your output with ours. The error should be on the order of e-8
    print('Testing relu_forward function:')
    print('difference: ', rel_error(out, correct_out))
```

```
Testing relu_forward function:
difference:  4.999999798022158e-08
```

ReLU activation: backward

Now implement the backward pass for the ReLU activation function in the `relu_backward` fun using numeric gradient checking:

```
[7] np.random.seed(231)
    x = np.random.randn(10, 10)
    dout = np.random.randn(*x.shape)

    dx_num = eval_numerical_gradient_array(lambda x: relu_forward(x)[0], x, dout)

    _, cache = relu_forward(x)
    dx = relu_backward(dout, cache)

    # The error should be on the order of e-12
    print('Testing relu_backward function:')
    print('dx error: ', rel_error(dx_num, dx))
```

```
Testing relu_backward function:
dx error:  3.2756349136310288e-12
```

Inline Question 1:

We've only asked you to implement ReLU, but there are a number of different activation functions that one could use in neural networks, each with its pros and cons. In particular, an issue commonly seen with activation functions is getting zero (or close to zero) gradient flow during backpropagation. Which of the following activation functions have this problem? If you consider these functions in the one dimensional case, what types of input would lead to this behaviour?

1. Sigmoid
2. ReLU
3. Leaky ReLU

1,2번이 해당 현상이 발생할 수 있다. Local gradient가 매우 작다면, gradient를 상쇄시키게 된다.

Loss layers: Softmax and SVM

Now implement the loss and gradient for softmax and SVM in the `softmax_loss` and `svm_loss` function in `cs231n/layers.py`. These should be similar to what you implemented in `cs231n/classifiers/softmax.py` and `cs231n/classifiers/linear_svm.py`.

You can make sure that the implementations are correct by running the following:

```
np.random.seed(231)
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)

dx_num = eval_numerical_gradient(lambda x: svm_loss(x, y)[0], x, verbose=False)
loss, dx = svm_loss(x, y)

# Test svm_loss function. Loss should be around 9 and dx error should be around the order of e-9
print('Testing svm_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))

dx_num = eval_numerical_gradient(lambda x: softmax_loss(x, y)[0], x, verbose=False)
loss, dx = softmax_loss(x, y)

# Test softmax_loss function. Loss should be close to 2.3 and dx error should be around e-8
print('\nTesting softmax_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing svm_loss:
loss:  8.999602749096233
dx error:  1.402156600651672e-09

Testing softmax_loss:
loss:  2.3025458445007376
dx error:  8.234144091578429e-09
```

앞에서 진행했던 svm과 softmax 로 loss와 dx를 구하는 과정이다. 주석에 표시된 것과 비슷한 수치가 나왔다.

```python
def svm_loss(x, y):
    """
    Computes the loss and gradient using for multiclass SVM classification.

    Inputs:
    - x: Input data, of shape (N, C) where x[i, j] is the score for the jth
      class for the ith input.
    - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
      0 <= y[i] < C

    Returns a tuple of:
    - loss: Scalar giving the loss
    - dx: Gradient of the loss with respect to x
    """
    loss, dx = None, None

    ############################################################################
    # TODO: Copy over your solution from A1.
    ############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    num_train = x.shape[0]
    correct_class_scores = x[np.arange(num_train), y].reshape(num_train,1)
    margins = np.maximum(0, x - correct_class_scores + 1)
    margins[np.arange(num_train), y] = 0
    loss = np.sum(margins) / num_train
    margin_count = np.sum(margins > 0, axis=1)
    dx = np.zeros_like(x)
    dx[margins > 0] = 1
    dx[np.arange(num_train), y] -= margin_count
    dx /= num_train


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

```python
def softmax_loss(x, y):
    """
    Computes the loss and gradient for softmax classification.

    Inputs:
    - x: Input data, of shape (N, C) where x[i, j] is the score for the jth
      class for the ith input.
    - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
      0 <= y[i] < C

    Returns a tuple of:
    - loss: Scalar giving the loss
    - dx: Gradient of the loss with respect to x
    """
    loss, dx = None, None

    ############################################################################
    # TODO: Copy over your solution from A1.
    ############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    num_train = x.shape[0]
    probs = np.exp(x - np.max(x,axis=1).reshape(-1,1))
    probs /= np.sum(probs, axis=1, keepdims=True)
    loss = -np.sum(np.log(probs[np.arange(num_train), y])) / num_train
    dx = probs.copy()
    dx[np.arange(num_train), y] -= 1
    dx /= num_train

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

앞부분과 똑같은 과정을 거쳐 작성해주었다.

Two-layer network

Fc_net.py

```python
############################################################################
# TODO: Initialize the weights and biases of the two-layer net. Weights    #
# should be initialized from a Gaussian centered at 0.0 with               #
# standard deviation equal to weight_scale, and biases should be           #
# initialized to zero. All weights and biases should be stored in the      #
# dictionary self.params, with first layer weights                         #
# and biases using the keys 'W1' and 'b1' and second layer                 #
# weights and biases using the keys 'W2' and 'b2'.                         #
############################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

self.params['W1']=np.random.normal(scale=weight_scale, size=(input_dim, hidden_dim))
self.params['W2']=np.random.normal(scale=weight_scale, size=(hidden_dim, num_classes))
self.params['b1']=np.zeros(hidden_dim)
self.params['b2']=np.zeros(num_classes)

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

W1과 w2는 size에 맞추어 랜덤으로 초기화하고, b1과 b2는 역시 사이즈를 맞추어 9으로 초기화하였다.

```python
scores = None
############################################################################
# TODO: Implement the forward pass for the two-layer net, computing the    #
# class scores for X and storing them in the scores variable.              #
############################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

out, cache1 = affine_relu_forward(X,self.params['W1'],self.params['b1'])
scores, cache2 = affine_forward(out,self.params['W2'],self.params['b2'])

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

첫번째 레이어는 relu를 사용하고 두번째 레이어는 기본 forward를 해주었다. 각각의 결과를 out과 cache에 저장한다.

```
############################################################################
# TODO: Implement the backward pass for the two-layer net. Store the loss  #
# in the loss variable and gradients in the grads dictionary. Compute data #
# loss using softmax, and make sure that grads[k] holds the gradients for  #
# self.params[k]. Don't forget to add L2 regularization!                   #
#                                                                          #
# NOTE: To ensure that your implementation matches ours and you pass the   #
# automated tests, make sure that your L2 regularization includes a factor #
# of 0.5 to simplify the expression for the gradient.                      #
############################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

loss, dx = softmax_loss(scores, y)
loss += 0.5*self.reg * np.sum(self.params['W1'] * self.params['W1']) + 0.5*self.reg * np.sum(self.params['W2'] * self.params['W2'])
dx2,grads['W2'], grads['b2']  = affine_backward(dx,cache2)
dx1,grads['W1'], grads['b1'] = affine_relu_backward(dx2,cache1)

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

Softmax_loss를 통해 loss와 dx 값을 구해주고, reg 값도 더해주어 loss를 구해준다.

Grad 값도 backward를 통해 구해준다.

다음과 같이 동일한 값으로 나온 것을 볼 수 있다.

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.83e-08
W2 relative error: 3.20e-10
b1 relative error: 9.83e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 1.00e+00
W2 relative error: 1.00e+00
b1 relative error: 1.56e-08
b2 relative error: 9.09e-10
```

다음은 sgd_momentum의 식을 작성하는 부분이다.

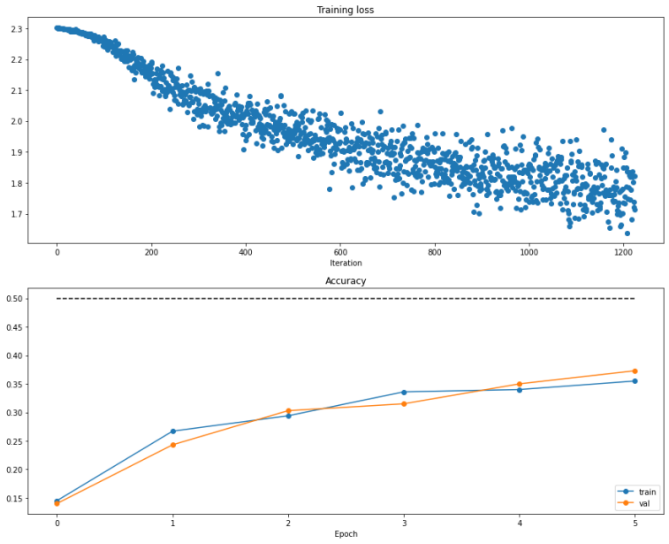
```
############################################################################
# TODO: Implement the momentum update formula. Store the updated value in  #
# the next_w variable. You should also use and update the velocity v.      #
############################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

v = config["momentum"]*v - config["learning_rate"]*dw
next_w +=v

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

그럼 다음과 같이 solver 라는 instance를 통해 모델을 훈련할 수 있다.

```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
model = TwoLayerNet(input_size, hidden_size, num_classes)
solver = None

################################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves about 36% #
# accuracy on the validation set.                                              #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
solver = Solver(model, data,
                update_rule='sgd',
                optim_config={
                  'learning_rate': 1e-4,
                },
                lr_decay=0.95,
                num_epochs=5, batch_size=200,
                print_every=100)
solver.train()

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

```
(Iteration 1 / 1225) loss: 2.301725
(Epoch 0 / 5) train acc: 0.145000; val_acc: 0.140000
(Iteration 101 / 1225) loss: 2.241923
(Iteration 201 / 1225) loss: 2.187425
(Epoch 1 / 5) train acc: 0.267000; val_acc: 0.243000
(Iteration 301 / 1225) loss: 2.056790
(Iteration 401 / 1225) loss: 1.937978
(Epoch 2 / 5) train acc: 0.294000; val_acc: 0.303000
(Iteration 501 / 1225) loss: 1.924555
(Iteration 601 / 1225) loss: 1.933743
(Iteration 701 / 1225) loss: 1.832777
(Epoch 3 / 5) train acc: 0.336000; val_acc: 0.315000
(Iteration 801 / 1225) loss: 1.960827
(Iteration 901 / 1225) loss: 1.832752
(Epoch 4 / 5) train acc: 0.340000; val_acc: 0.350000
(Iteration 1001 / 1225) loss: 1.739182
(Iteration 1101 / 1225) loss: 1.940517
(Iteration 1201 / 1225) loss: 1.848443
(Epoch 5 / 5) train acc: 0.355000; val_acc: 0.373000
```

하지만, 아래 그래프를 참고하면 train과 val 데이터의 차이가 별로 없으며 loss 감소 폭이 적다는 것을 알 수 있다. 따라서 학습률이 다소 작았던 것으로 볼 수 있다.



```
################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained  #
# model in best_model.                                                          #
#                                                                               #
# To help debug your network, it may help to use visualizations similar to the  #
# ones we used above; these visualizations will have significant qualitative    #
# differences from the ones we saw above for the poorly tuned network.          #
#                                                                               #
# Tweaking hyperparameters by hand can be fun, but you might find it useful to  #
# write code to sweep through possible combinations of hyperparameters          #
# automatically like we did on thexs previous exercises.                        #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10

best_model = TwoLayerNet(input_size, hidden_size, num_classes)
solver = None
solver = Solver(best_model, data,
                update_rule='sgd',
                optim_config={
                  'learning_rate': 1e-3,
                },
                lr_decay=0.95,
                num_epochs=5, batch_size=200,
                print_every=100)
solver.train()


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

```
(Iteration 1 / 1225) loss: 2.302208
(Epoch 0 / 5) train acc: 0.125000; val_acc: 0.131000
(Iteration 101 / 1225) loss: 1.848463
(Iteration 201 / 1225) loss: 1.694937
(Epoch 1 / 5) train acc: 0.412000; val_acc: 0.407000
(Iteration 301 / 1225) loss: 1.706805
(Iteration 401 / 1225) loss: 1.598236
(Epoch 2 / 5) train acc: 0.448000; val_acc: 0.441000
(Iteration 501 / 1225) loss: 1.564486
(Iteration 601 / 1225) loss: 1.466235
(Iteration 701 / 1225) loss: 1.401038
(Epoch 3 / 5) train acc: 0.458000; val_acc: 0.449000
(Iteration 801 / 1225) loss: 1.549872
(Iteration 901 / 1225) loss: 1.450629
(Epoch 4 / 5) train acc: 0.504000; val_acc: 0.484000
(Iteration 1001 / 1225) loss: 1.359496
(Iteration 1101 / 1225) loss: 1.419782
(Iteration 1201 / 1225) loss: 1.420421
(Epoch 5 / 5) train acc: 0.509000; val_acc: 0.502000
```

학습률을 줄여 다시 시도해보았더니 정확도가 상승된 것을 볼 수 있었다.

테스트 결과도 제시된 대로 48%가 나왔다.

Test your model!

Run your best model on the validation and test sets. You should achieve above 48% accuracy on the validation se

```
[15] y_val_pred = np.argmax(best_model.loss(data['X_val']), axis=1)
     print('Validation set accuracy: ', (y_val_pred == data['y_val']).mean())

     Validation set accuracy:  0.502
```

```
[16] y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
     print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())

     Test set accuracy:  0.481
```

Inline Question 2:

Now that you have trained a Neural Network classifier, you may find that your testing accuracy is much lower than the training accuracy. In what ways can we decrease this gap? Select all that apply.

1. Train on a larger dataset.
2. Add more hidden units.
3. Increase the regularization strength.
4. None of the above.

Your Answer :

Your Explanation :

1,3 / test 데이터의 정확도가 train 데이터의 정확도보다 훨씬 낮다는 것을 오버피팅, 즉 과적합이 되었다는 뜻이다. 따라서 과적합을 해소할 수 있는 더 큰 데이터 셋을 적용하거나 regularization을 적용할 수 있다, 따라서 1,3이다.

5. FC_net

Features.ipynb는 데이터의 픽셀값들을 넣는 것이 아니라 특징값을 뽑아 그것을 데이터로 사용하는 과정이다.

다음과 같이 특징값들을 뽑아주고

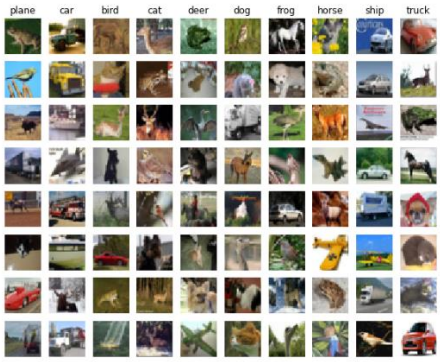
```
[4] from cs231n.features import *

    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))])

    Done extracting features for 1000 / 49000 images
    Done extracting features for 2000 / 49000 images
    Done extracting features for 3000 / 49000 images
    Done extracting features for 4000 / 49000 images
```

Svm을 이용하여 훈련하는 과정이다. Lr과 reg를 조정하여 파라미터 튜닝하는 과정을 거쳐준다. Validation 데이터에서 가장 높은 정확도는 42%가 나왔다.

```
################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# This should be identical to the validation that you did for the SVM; save    #
# the best trained classifer in best_svm. You might also want to play          #
# with different numbers of bins in the color histogram. If you are careful    #
# you should be able to get accuracy of near 0.44 on the validation set.       #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

for lr in learning_rates:
  for reg in regularization_strengths:
    svm = LinearSVM()
    svm.train(X_train_feats, y_train, learning_rate=lr, reg=reg,num_iters=1500)
    y_train_pred = svm.predict(X_train_feats)
    acc_tr = np.mean(y_train == y_train_pred)
    y_val_pred = svm.predict(X_val_feats)
    acc_val = np.mean(y_val == y_val_pred)
    if best_val < acc_val:
      best_val = acc_val
      best_svm=svm
    results[(lr, reg)] = (acc_tr, acc_val)

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

# Print out results.
```

test데이터에서는 약 42%의 정확도를 보였다.

```
[6] # Evaluate your trained SVM on the test set: you should be able to get at least 0.40
    y_test_pred = best_svm.predict(X_test_feats)
    test_accuracy = np.mean(y_test == y_test_pred)
    print(test_accuracy)

    0.424
```



Inline question 1:

Describe the misclassification results that you see. Do they make sense?

Your Answer :

분류된 이미지들을 보면 대체로 분류가 잘 이뤄지지 않았다는 것을 볼 수 있다. Car class에 경우에는 비교적 잘 분류되었지만, frog class, cat class 등에서 분류가 제대로 이뤄지지 않았음을 볼 수 있었다.

다음은 앞선 two layer network로 훈련하는 과정이다. 이번에는 lr을 튜닝하는 과정을 거쳤다.

```
data = {
    'X_train': X_train_feats,
    'y_train': y_train,
    'X_val': X_val_feats,
    'y_val': y_val,
    'X_test': X_test_feats,
    'y_test': y_test,
}

best_net = None
learning_rates = [1e-3, 1e-2, 1e-1]
################################################################################
# TODO: Train a two-layer neural network on image features. You may want to    #
# cross-validate various parameters as in previous sections. Store your best   #
# model in the best_net variable.                                              #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
best_val_acc=0
for lr in learning_rates:
    model = TwoLayerNet(input_dim, hidden_dim, num_classes)
    solver = Solver(model, data,
                update_rule='sgd',
                optim_config={
                  'learning_rate': lr,
                },
                lr_decay=0.95,
                num_epochs=20, batch_size=100,
                print_every=100)
    solver.train()
    if solver.best_val_acc>best_val_acc:
       best_model = model

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

Validation data에 대하여 최고 약 60%의 정확도를 보였다.

```
# Run your best neural net classifier on the test set. You should be able
# to get more than 55% accuracy.

y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
test_acc = (y_test_pred == data['y_test']).mean()
print(test_acc)
```

```
0.579
```

Test 데이터에서 약 57%의 정확도를 보였다.