P02: Multi-class SVM Classifier on CIFAR10

In this practical, you will practice putting together a simple classification pipeline. We shall work with the SVM linear classifier. The goals of this practical are as follows:

1. Preprocess the data by **subtracting the mean image**
2. Implement and apply a Multiclass Support Vector Machine (**SVM**) linear classifier
3. Optimize the loss function with **SGD**
4. Perform hyperparameter tuning of the learning rate and regularization strength using **random search**

Once we have completed the task, we shall visualize the weights learnt by gradient descent.

In [ ]:
```
from lib import cifar10
from lib import common
import numpy as np
import matplotlib.pyplot as plt
```

In [ ]:
```
# set the plot
plt.rcParams['image.interpolation'] = 'nearest'

# set to automatic reload
%load_ext autoreload
%autoreload 2
```

Data preparation

Loading and preparing the data

First, we load and prepare our data.

In [ ]:
```
X_train, y_train, X_test, y_test = cifar10.load_data (r'.\data\cifar-10-batche
s-py')
classes = cifar10.get_classes()
num_classes = len(classes)
print('classes:', classes)
print('Shape of training samples:', X_train.shape)
print('Shape of testing samples:',  X_test.shape)
```

Get the training, validation, testing and development set

The following code split the data into *train*, *val*, and *test* sets. In addition we will create a small **development set** as a subset of the training data. We shall use this for code development (particularly to develop code to compute the loss function) so our code runs faster.

In [ ]:
```
num_training = 49000
num_validation = 1000
num_test = 1000
num_dev = 500

# Our validation set will be num_validation points from the original
# training set.
mask = range(num_training, num_training + num_validation)
X_val = X_train[mask]
y_val = y_train[mask]

# Our training set will be the first num_train points from the original
# training set.
mask = range(num_training)
X_train = X_train[mask]
y_train = y_train[mask]

# We will also make a development set, which is a small subset of
# the training set.
mask = np.random.choice(num_training, num_dev, replace=False)
X_dev = X_train[mask]
y_dev = y_train[mask]

# We use the first num_test points of the original test set as our
# test set.
mask = range(num_test)
X_test = X_test[mask]
y_test = y_test[mask]

print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

Exercise 1: convert the dataset (X_train, X_val, X_test and X_dev) from uint8 to double

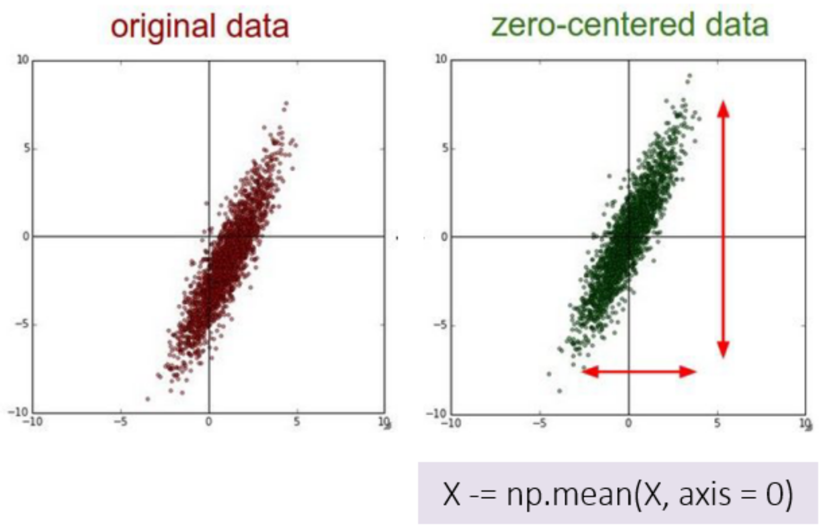
In [ ]:
```
X_train.dtype      # uint8
```

In [ ]:
```
# Your code here
```

In [ ]:
```
X_train.dtype      # expecting float64
```

Center the data

One of the common step is to **center the data**. This preprocessing step is important if the input is an image which has a range of [0, 255] where all values are positive. When all inputs are positive, this will slow down the training process. More details can be found in the lecture on "Regular Neural Network".



Exercise 2: Center all samples

First,compute the mean image by computing mean of all the images in the training samples. Then, display the mean image.

```
In [ ]: # Your code here
```

Subtract the mean image from all samples

```
In [ ]: # your code here
```

Exercise 3: reshape the image data into rows

First, reshape the image data [?, 32, 32, 3] into rows [?, 3072]

```
In [56]: # your code here
```

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

Adding the bias term into W

Lastly we append the bias dimension of ones (i.e. bias trick) so that our SVM only has to worry about optimizing a single weight matrix W.

```
In [ ]: X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
        X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
        X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
        X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

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

Coding the SVM Classifier

Exercise 4: Develop the function to compute the SVM loss function (non-vectorized version)

Now, let's develop the code to build our SVM classifier. Complete the function **`compute_loss_naive`** in **`lib/linear_svm.py`** which implements the multi-svm loss.

$$L = \frac{1}{N}\sum_i L_i + \lambda R(W)$$

$$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)$$

where L is the multi-class SVM loss across the whole dataset, L_i is the data loss for sample i and $R(W)$ is the regularizer function, y_i is the true label for sample i and s_j is the score of class j for sample i.

For regularization, we shall use the L2 norm.

$$R(W) = \sum_j \sum_k W_{j,k}^2$$

Complete the function **`compute_loss_naive`** which uses two `for` loops to evaluate the multiclass SVM loss function. Note that the **non-vectorized** implementation is not a very efficient one. We shall develop a more efficient vectorized version later.

```
In [ ]: from lib.linear_svm import svm_loss_naive

        W = np.random.randn(3073, 10) * 0.0000000001
        loss = svm_loss_naive(W, X_dev, y_dev, 0)

        print('loss: {:f}'.format(loss))
```

Note that we have used a very small `W` for the code above.

Question: Since W is small and the regularization parameter has been turned off, what is the expected value for your SVM loss? ____________________

Use your answer to check if your implementation above is correct.

Mini-batch Gradient Descent

You have implemented the loss function. We are now ready to implement mini-batch gradient descent to find the best parameter W^* that minimizes the loss function L.

Exercise 5: Write the mini-batch gradient descent algorithm

Now, let's develop the code for the mini-batch gradient descent. Complete the function **`LinearClassifier.train`** in `lib/linear_classifier.py` which implements the gradient descent.

If you completed the task successfully, you should see that the loss decreases from around 780 to 5.5 over the iterations.

```
In [ ]: from lib.linear_classifier import LinearSVM
        import time

        tic = time.time()

        svm = LinearSVM()
        loss_hist = svm.train(X_train, y_train,
                              batch_size=200,
                              learning_rate=1e-7,
                              reg=2.5e4,
                              num_epochs=5,
                              vectorized = True,
                              verbose=True)

        toc = time.time()
        print('That took %fs' % (toc - tic))
```

A useful debugging strategy is to plot the loss as a function of iteration number. The following code plots the graph loss vs iteration. The loss should decrease over time.

```
In [ ]: plt.plot(loss_hist)
        plt.xlabel('Iteration number')
        plt.ylabel('Loss value')
        plt.show()
```

Exercise 6: Write the prediction function

Now, let's develop the code to predict the labels for any samples. Complete the function **`LinearClassifier.train`** in `lib/linear_classifier.py` and then evaluate the performance on training and validation set.

If you completed the task successfully, We are expecting an accuracy of about 3.6 or above for the model we created above.

In [ ]:
```
y_train_pred = svm.predict(X_train)
print('Training accuracy: {:f}'.format(np.mean(y_train == y_train_pred)))

y_val_pred = svm.predict(X_val)
print('validation accuracy: {:f}'.format(np.mean(y_val == y_val_pred)))
```

Training and hyperparameter tuning

Now that we have developed all the necessary code, it's time to perform training. We shall use the validation set to finetune hyperparameters (regularization strength and learning rate).

Exercise 7: Write the cross-validation code (Random Search) for hyperparameter tuning

Write the code that chooses the best hyperparameters by tuning on the validation set.

In the previous practical, we have used **Grid Search** for cross-validation. Today, we are going to use **Random Search** to determine our best hyperprameter on the validation set. Random search has been shown to produce comparable performance with lots of computational saving. Complete the code below to find the best hyperparameter settings.

Hint: You should use a small value for the number of batch iterations (`max_iter = 100`) as you develop your validation code so that the SVMs don't take much time to train. Once you are confident that your validation code works, you should rerun the validation code with a larger number of iterations (`max_iter = -1`).

```
In [ ]: results = {}                  # results is dictionary mapping (learning_rate, r
        egularization_strength)
                                      # to (training_accuracy, validation_accuracy). Th
        e accuracy is simply the
                                      #fraction of data points that are correctly class
        ified.
        best_val = -1                 # The highest validation accuracy that we have se
        en so far.
        best_svm = None               # The LinearSVM object that achieved the highest
         validation rate.

        # hyperparameter settings
        learning_rates = [-7, -2]
        regularization_strengths = [-4, 3]
        num_trials = 10

        for i in range(num_trials):

            ############################################################################
        ########
            # TODO:
               #
            # sample the learning rate and regularization hyperparameters in log spa
        ce     #
            ############################################################################
        ########
            # lr  = ...
            # reg = ...
```

```
    ########################################################################
########
    #                        END OF YOUR CODE
       #
    ########################################################################
########

    ########################################################################
########
    # TODO:
       #
    # Train on the training set. Set number of epochs to 7, maximum number o
f      #
    #     iteration to 100, verbose to False. Set to use the vectorized vers
ion if #
    #     possible.
       #
    ########################################################################
########
    # Your code here
    ########################################################################
########
    #                        END OF YOUR CODE
       #
    ########################################################################
########

    ########################################################################
########
    # TODO:
       #
    # Evaluate your model on both the training and validation set
       #
    ########################################################################
########
    # your code here
    ########################################################################
########
    #                        END OF YOUR CODE
       #
    ########################################################################
########

    results[(lr, reg)] = (train_accuracy, val_accuracy)

    if val_accuracy > best_val:
        best_val = val_accuracy
        best_svm = svm

    print('iter {:d}: lr {:e} reg {:e} train accuracy: {:.4f} val accuracy:
{:.4f}'.format(
                i, lr, reg, train_accuracy, val_accuracy))


# Sort the validation result for easy viewing
import operator
sorted_results = sorted(results.items(), key=operator.itemgetter(1))
```

```
print('\n-------------------------------- Sorted result ------------------
----------------')
for ((lr, reg), (train_acc, val_acc)) in sorted_results:
    print('lr {:e}, reg{:e}: train accuracy: {:.4f} val accuracy: {:.4f}'.fo
rmat(lr, reg, train_acc, val_acc))
print('>> Best validation: {:.4f}'.format(best_val))
```

Exercise 8: Optimize your hyperparameters on the validation set

Hyperparameter tuning is performed in stages. The first stage is to find the range of working range for hyperparameters (we have done this for you). Then, yYou should repeat the above process using a finer ranges for the learning rate and regularization strength. You may repeat this several times until you are satisfied with the performance of your system on the validation set. If you are careful you should be able to get a classification accuracy of about 0.4 on the validation set.

What is the best accuracy that you can achieve? What is the finest range of learning rates and regularization strength that you evaluate on?
Answer: ____________________________

Evaluation and Analysis

Evaluating on the testing set

Exercise 9: Evaluate on the test set

From above, we determine the best hyperparameter settings through cross-validation. The best model has also been saved as `best_svm`. The following code shows the prediction resut on the test set

```
In [57]: # your code here
```

Visualizing the template

Now, let's visualize the learnt weights for each class. Depending on your choice of learning rate and regularization strength, these may or may not be nice to look at.

In [ ]:
```
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', 'shi
p', 'truck']

plt.figure(figsize=(12, 5))
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])
plt.show()
```

Course Assignment 2 [2 marks]

Submission:

This is the second coursework assignment which extends this practical. Complete this section and upload the following file to WBLE by 11 Feb 2018.

- `linear_svm.py`

Task: Vectorized version of the multi-class SVM loss.

For this assignment, your task is to implement the vectorized version of the multiclass svm loss function. Complete the function **`svm_loss_vectorized`** in `linear_svm.py`. The function should NOT contain any for loops.

After you have completed your assignment, verify your implementation by running the code below. The loss computed by the vectorized version must be equal to the naive version. The vectorized version should be much more efficient than the non-vectorized version.