CS280 Spring 2025 Assignment 1
Part A

Basics

March 4, 2025

Name: Chen Xuanxin

Student ID: 2024233125

1. ***Maximum Likelihood Estimation*** **(10 points).**

Consider a dataset $\mathcal{D}$ consisting of n independent and identically distributed samples:

$$\mathcal{D} = \left\{((x_1^1, x_2^1), y^1), ((x_1^2, x_2^2), y^2), \ldots, ((x_1^n, x_2^n), y^n)\right\}, \tag{1}$$

where $(x_1^i, x_2^i) \in \mathbb{R}^2$ are input features and $y^i \in \mathbb{R}$ is an output.
Assume that every output y^i in $\mathcal{D}$ is generated by inputting (x_1^i, x_2^i) into a model:

$$y = f_{\theta_1,\theta_2}(x_1, x_2) + \epsilon, \tag{2}$$

where the function f_{θ_1,θ_2} is a mapping from features $(x_1, x_2) \in \mathbb{R}^2$ to a value in $\mathbb{R}$, which has two parameters θ_1 and θ_2. Here we assume that the random noise $\epsilon \sim N(0, \sigma^2)$ is independent and distributed according to a Gaussian distribution with zero mean and variance σ^2.

(a) Show that the log likelihood of the data given the parameters is:

$$l(\mathcal{D}; \theta_1, \theta_2) = -\frac{1}{2\sigma^2} \sum_{i=1}^{n} (y^i - f_{\theta_1,\theta_2}(x_1^i, x_2^i))^2 - n \log(\sqrt{2\pi}\sigma). \tag{3}$$

Recall the probability density function of the Gaussian distribution $N(\mu, \sigma^2)$ is:

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right). \tag{4}$$

(b) To find the maximum likelihood estimates of θ_1 and θ_2 using gradient descent, compute the gradient of the log likelihood with respect to θ_1 and θ_2. Express you answer in terms of:

$$y^i, \;\; f_{\theta_1,\theta_2}(x_1^i, x_2^i), \;\; \frac{\partial}{\partial\theta_1} f_{\theta_1,\theta_2}(x_1^i, x_2^i), \;\; \frac{\partial}{\partial\theta_2} f_{\theta_1,\theta_2}(x_1^i, x_2^i)$$

(c) Given the learning rate η, what update rule would you use in gradient descent to *maximize* the likelihood.

1 Answer for Maximum Likelihood Estimation

(a) Derivation of Log-Likelihood

Given the noise $\epsilon \sim N(0, \sigma^2)$, the conditional distribution of each sample y^i is:

$$y^i \mid x_1^i, x_2^i \sim N\left(f_{\theta_1,\theta_2}(x_1^i, x_2^i), \sigma^2\right).$$

Using the probability density function of the Gaussian distribution (Equation (4)), the likelihood for a single sample is:

$$p(y^i \mid x_1^i, x_2^i, \theta_1, \theta_2) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^i - f_{\theta_1,\theta_2}(x_1^i, x_2^i))^2}{2\sigma^2}\right).$$

Since the samples are independent and identically distributed (i.i.d.), the total likelihood is the product of individual likelihoods:

$$L(\mathcal{D}; \theta_1, \theta_2) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^i - f_{\theta_1,\theta_2}(x_1^i, x_2^i))^2}{2\sigma^2}\right).$$

Taking the logarithm and simplifying:

$$\begin{aligned}
l(\mathcal{D}; \theta_1, \theta_2) &= \sum_{i=1}^{n} \ln\left(\frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^i - f_{\theta_1,\theta_2}(x_1^i, x_2^i))^2}{2\sigma^2}\right)\right) \\
&= \sum_{i=1}^{n} \left(-\ln(\sqrt{2\pi}\sigma) - \frac{(y^i - f_{\theta_1,\theta_2}(x_1^i, x_2^i))^2}{2\sigma^2}\right) \\
&= -\frac{1}{2\sigma^2} \sum_{i=1}^{n} (y^i - f_{\theta_1,\theta_2}(x_1^i, x_2^i))^2 - n \ln(\sqrt{2\pi}\sigma).
\end{aligned}$$

Thus, the log-likelihood is:

$$l(\mathcal{D}; \theta_1, \theta_2) = -\frac{1}{2\sigma^2} \sum_{i=1}^{n} (y^i - f_{\theta_1,\theta_2}(x_1^i, x_2^i))^2 - n \log(\sqrt{2\pi}\sigma)$$

(b) Gradient Computation

The partial derivative with respect to θ_1 is:

$$\begin{aligned}
\frac{\partial l}{\partial \theta_1} &= -\frac{1}{2\sigma^2} \sum_{i=1}^{n} \frac{\partial}{\partial \theta_1} \left(y^i - f_{\theta_1,\theta_2}(x_1^i, x_2^i)\right)^2 \\
&= -\frac{1}{2\sigma^2} \sum_{i=1}^{n} 2\left(y^i - f_{\theta_1,\theta_2}(x_1^i, x_2^i)\right) \cdot \left(-\frac{\partial f_{\theta_1,\theta_2}}{\partial \theta_1}\right) \\
&= \frac{1}{\sigma^2} \sum_{i=1}^{n} \left(y^i - f_{\theta_1,\theta_2}(x_1^i, x_2^i)\right) \frac{\partial f_{\theta_1,\theta_2}}{\partial \theta_1}.
\end{aligned}$$

Similarly, the partial derivative with respect to θ_2 is:

$$\frac{\partial l}{\partial \theta_2} = \frac{1}{\sigma^2} \sum_{i=1}^{n} \left(y^i - f_{\theta_1,\theta_2}(x_1^i, x_2^i)\right) \frac{\partial f_{\theta_1,\theta_2}}{\partial \theta_2}.$$

The gradient vector is:

$$\nabla_{\theta_1,\theta_2} l = \left[\frac{1}{\sigma^2} \sum_{i=1}^{n} \left(y^i - f_{\theta_1,\theta_2}(x_1^i, x_2^i)\right) \frac{\partial f_{\theta_1,\theta_2}}{\partial \theta_1}, \ \frac{1}{\sigma^2} \sum_{i=1}^{n} \left(y^i - f_{\theta_1,\theta_2}(x_1^i, x_2^i)\right) \frac{\partial f_{\theta_1,\theta_2}}{\partial \theta_2} \right]$$

(c) Gradient Descent Update Rule To maximize the likelihood, we use gradient ascent. The update rule for parameters is:

$$\theta_j^{(k+1)} = \theta_j^{(k)} + \eta \cdot \frac{\partial l}{\partial \theta_j}, \quad j = 1, 2.$$

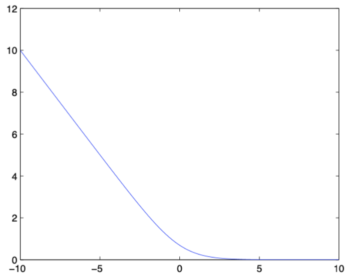
Substituting the gradient expressions, we get:

$$\theta_j^{(k+1)} = \theta_j^{(k)} + \frac{\eta}{\sigma^2} \sum_{i=1}^{n} \left(y^i - f_{\theta_1,\theta_2}(x_1^i, x_2^i)\right) \frac{\partial f_{\theta_1,\theta_2}}{\partial \theta_j}, \quad j = 1, 2$$

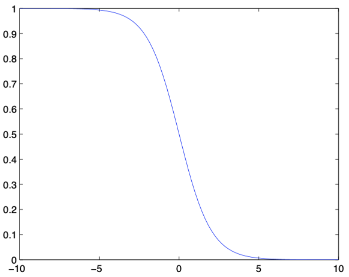
2. *Loss Function* (10 points).

Assume that a classifier is written as $H(x) = sign(F(x))$, where $H(x) : \mathbb{R}^d \rightarrow \{-1, 1\}$, $sign()$ is a sign function, and $F(x) : \mathbb{R}^d \rightarrow \mathbb{R}$. To obtain the parameters in $F(x)$, we need to minimize the loss function averaged over the training set: $\sum_i L(y^i F(x^i))$. Here L is a function of $yF(x)$. For example, for linear classifiers, $F(x) = w_0 + \sum_{j=1}^d w_j x_j$, and $yF(x) = y(w_0 + \sum_{j=1}^d w_j x_j)$.

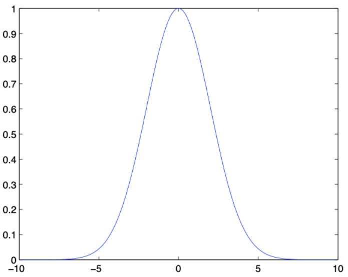
(a) Which loss functions below are appropriate to use in classification? For the ones that are not appropriate, explain why not. In general, what conditions does L have to satisfy in order to be an appropriate loss function? The x axis is $yF(x)$, and the y axis is $L(yF(x))$.



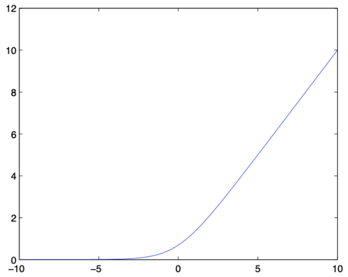
(a)



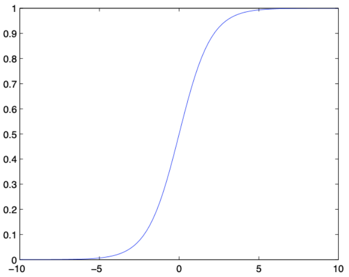
(b)



(c)



(d)



(e)

(b) Among the above loss functions appropriate to use in classification, which one is the most robust to outliers? Justify your answer.

(c) Let $F(x) = w_0 + \sum_{j=1}^d w_j x_j$ and $L(yF(x)) = \frac{1}{1+exp(yF(x))}$. Suppose you use gradient descent to obtain the optimal values for w_0 and w_j. Give the update rules for these parameters.

2 Answer for Loss Functions

(a) Appropriate loss functions

From the provided plots (a)–(e) (with the x-axis being $yF(x)$ and the y-axis being $L(yF(x))$), we see that only (a) and (b) are monotonically decreasing and approach 0 as $yF(x) \to +\infty$. These two satisfy the typical requirements for classification loss:

- $L(t) \geq 0$ for all t,
- $L(t)$ is decreasing in t,
- $L(t) \to 0$ as $t \to +\infty$ (correct classification with large margin),
- $L(t)$ becomes large when $t \ll 0$ (incorrect classification with large negative margin).

Analysis of subfigures (a)-(e):

Subfigure	Characteristics	Appropriate?
(a)	Exponentially decreasing curve: $L \approx 10 \to 0$ as $yF(x)$ increases	✓
(b)	Saturated loss: Flat regions for $\lvert yF(x)\rvert > 2$, sharp transition near $yF(x) = 0$	✓
(c)	Bell-shaped curve: Peaks at $yF(x) = 0$ (*opposite* to required monotonicity)	×
(d)	Increasing linear function: $L \propto yF(x)$ (*violates* the decreasing requirement)	×
(e)	Logistic-like curve but reversed sign: does not decrease as $yF(x)$ grows	×

> Only (a) and (b) are *monotonically decreasing* in $yF(x)$ and approach 0 when $yF(x) \to +\infty$. Hence, (a) and (b) are appropriate for classification; (c), (d), and (e) are not.

(b) Robustness to outliers

Among the appropriate loss functions, (a) resembles the exponential loss, while (b) resembles a logistic-type loss. The exponential loss $\exp(-yF(x))$ grows very large for highly negative margins and is therefore more sensitive to outliers. The logistic loss saturates more smoothly, making it more robust to outliers. Hence, (b) is the most robust among them.

(c) Gradient descent updates for $L\big(yF(x)\big) = \frac{1}{1+\exp\big(yF(x)\big)}$

Let the training set be $\{(x^i,\, y^i)\}_{i=1}^n$, where $y^i \in \{-1, +1\}$, and define

$$F(x^i) \;=\; w_0 \;+\; \sum_{j=1}^{d} w_j\, x_j^i.$$

Then the loss for sample i is

$$L_i \;=\; L\big(y^i F(x^i)\big) \;=\; \frac{1}{1 + \exp\big(y^i F(x^i)\big)}.$$

We want to minimize the sum $\sum_{i=1}^{n} L_i$.

Partial derivatives. Let $z_i = y^i F(x^i)$. Then

$$L_i \;=\; \frac{1}{1+e^{z_i}}, \quad \frac{\partial L_i}{\partial z_i} = -\,\frac{e^{z_i}}{\big(1+e^{z_i}\big)^2} = \,-\,L_i\,(1-L_i).$$

Note also that $\dfrac{\partial z_i}{\partial w_0} = y^i$, and $\dfrac{\partial z_i}{\partial w_j} = y^i\,x^i_j$.

Hence,

$$\frac{\partial L_i}{\partial w_0} = \frac{\partial L_i}{\partial z_i}\,\frac{\partial z_i}{\partial w_0} = \,-\,y^i\,L_i\,(1-L_i),$$

$$\frac{\partial L_i}{\partial w_j} = \,-\,y^i\,x^i_j\,L_i\,(1-L_i).$$

Gradient descent updates. Let α be the learning rate. Then the update rules for each iteration are:

$$w_0 \;\leftarrow\; w_0 \;-\; \alpha\sum_{i=1}^{n}\frac{\partial L_i}{\partial w_0} \;=\; w_0 \;+\; \alpha\sum_{i=1}^{n}\Big[y^i\,L_i\,\big(1-L_i\big)\Big],$$

$$w_j \;\leftarrow\; w_j \;-\; \alpha\sum_{i=1}^{n}\frac{\partial L_i}{\partial w_j} \;=\; w_j \;+\; \alpha\sum_{i=1}^{n}\Big[y^i\,x^i_j\,L_i\,\big(1-L_i\big)\Big],$$

where $L_i = \dfrac{1}{1+\exp\!\big(y^iF(x^i)\big)}$.

Perceptron Learning Algorithm

The perceptron is a simple supervised machine learning algorithm and one of the earliest neural network architectures. It was introduced by Rosenblatt in the late 1950s. A perceptron represents a binary linear classifier that maps a set of training examples (of d dimensional input vectors) onto binary output values using a $d-1$ dimensional hyperplane. But Today, we will implement **Multi-Classes Perceptron Learning Algorithm Given:**

- dataset $\{(x^i, y^i)\}$, $i \in (1, M)$
- x^i is d dimension vector, $x^i = (x^i_1, \ldots x^i_d)$
- y^i is multi-class target varible $y^i \in \{0, 1, 2\}$

A perceptron is trained using gradient descent. The training algorithm has different steps. In the beginning (step 0) the model parameters are initialized. The other steps (see below) are repeated for a specified number of training iterations or until the parameters have converged.

Step0: Initial the weight vector and bias with zeros
Step1: Compute the linear combination of the input features and weight.
$y^i_{pred} = \arg\max_k W_k * x^i + b_k, foreachclassk \in 0, 1, 2$
Step2: Compute the gradients for parameters W_k, b_k. **Derive the parameter update equation Here (5 points)**

##################################
My Answer:

We begin by noting that the prediction for a given input x^i is made as follows:

$$\hat{y} = \arg\max_{k \in \{0,1,2\}} \left(W_k \cdot x^i + b_k\right)$$

If the prediction $\hat{y}$ is correct (i.e. $\hat{y} = y^i$), no update is made. However, if the prediction is incorrect (i.e. $\hat{y} \neq y^i$), we want to adjust the weights and biases so that the score for the true class increases and the score for the incorrect (predicted) class decreases.

One common way to view this is by considering a loss function that reflects the misclassification. One formulation is to use the following (non-differentiable) loss for a single example:

$$L(W, b; x^i, y^i) = \max\left(0,\ \left[W_{\hat{y}} \cdot x^i + b_{\hat{y}}\right] - \left[W_{y^i} \cdot x^i + b_{y^i}\right]\right)$$

For the case when $\hat{y} \neq y^i$ (i.e. the loss is positive), we can compute the subgradients with respect to the parameters of the involved classes:

- **For the correct class y^i:**
    - $\frac{\partial L}{\partial W_{y^i}} = -x^i$
    - $\frac{\partial L}{\partial b_{y^i}} = -1$
- **For the misclassified (predicted) class $\hat{y}$:**
    - $\frac{\partial L}{\partial W_{\hat{y}}} = x^i$
    - $\frac{\partial L}{\partial b_{\hat{y}}} = 1$
- **For all other classes k (where $k \neq y^i$ and $k \neq \hat{y}$)**, the gradients are zero.

Using gradient descent with a learning rate η, we update the parameters by moving in the negative direction of the gradient. Thus, the update equations are:

- **For the true class y^i:** $W_{y^i} \leftarrow W_{y^i} - \eta\left(-x^i\right) = W_{y^i} + \eta x^i \; b_{y^i} \leftarrow b_{y^i} - \eta\left(-1\right) = b_{y^i} + \eta$
- **For the predicted (incorrect) class $\hat{y}$:** $W_{\hat{y}} \leftarrow W_{\hat{y}} - \eta\left(x^i\right) = W_{\hat{y}} - \eta x^i \; b_{\hat{y}} \leftarrow b_{\hat{y}} - \eta\left(1\right) = b_{\hat{y}} - \eta$
- **For any other class k not involved in the misclassification (i.e. $k \neq y^i$ and $k \neq \hat{y}$)**, the parameters remain unchanged.

In summary, the parameter update rule for a misclassified example is:

$$\begin{aligned} W_{y^i} &\leftarrow W_{y^i} + \eta\, x^i, \qquad & b_{y^i} &\leftarrow b_{y^i} + \eta, \\ W_{\hat{y}} &\leftarrow W_{\hat{y}} - \eta\, x^i, \qquad & b_{\hat{y}} &\leftarrow b_{\hat{y}} - \eta. \end{aligned}$$

This update increases the score for the correct class while decreasing the score for the incorrect class, guiding the perceptron toward making the right classification in subsequent iterations.

#################################

In [1]:

```
from sklearn import datasets
import numpy as np
from sklearn.model_selection import train_test_split
# newer version sklearn change the name of cross_validation
import matplotlib.pyplot as plt
import random

np.random.seed(0)
random.seed(0)
```

In [2]:

```
iris = datasets.load_iris()
X = iris.data
print(type(X))
y = iris.target
y = np.array(y)
print('X_Shape:', X.shape)
print('y_Shape:', y.shape)
print('Label Space:', np.unique(y))
```

```
<class 'numpy.ndarray'>
X_Shape: (150, 4)
y_Shape: (150,)
Label Space: [0 1 2]
```

In [3]:

```
## split the training set and test set
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.3, random_state=0)
print('X_train_Shape:', X_train.shape)
print('X_test_Shape:',  X_test.shape)
print('y_train_Shape:', y_train.shape)
print('y_test_Shape:',  y_train.shape)

print(type(y_train))
```

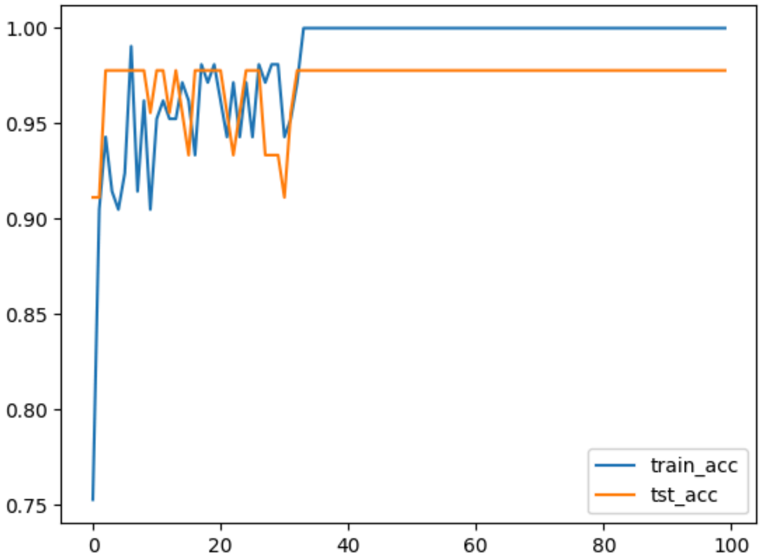
```
X_train_Shape: (105, 4)
X_test_Shape: (45, 4)
y_train_Shape: (105,)
y_test_Shape: (105,)
<class 'numpy.ndarray'>
```

In [16]:

```
class MultiClsPLA(object):

    ## We recommend to absorb the bias into weight.  W = [w, b]

    def __init__(self, X_train, y_train, X_test, y_test, lr, num_epoch, weight_dimension, num_cls):
        super(MultiClsPLA, self).__init__()
        self.X_train = X_train
        self.y_train = y_train
        self.X_test = X_test
        self.y_test = y_test
        self.weight = self.initial_weight(weight_dimension, num_cls)
        self.sample_mean = np.mean(self.X_train, 0)
        self.sample_std = np.std(self.X_train, 0)
        self.num_epoch = num_epoch
        self.lr = lr
        self.total_acc_train = []
        self.total_acc_tst = []

    def initial_weight(self, weight_dimension, num_cls):
        # weight = None
        #########################################
        ##  ToDO: Initialize the weight with   ##
        ##  small std and zero mean gaussian   ##
        #########################################
        weight = np.random.normal(loc=0.0, scale=0.01, size=(num_cls, weight_dimension))
        return weight

    def data_preprocessing(self, data):
        #####################################
        ##  ToDO: Normlize the data        ##
        #####################################
        norm_data = (data-self.sample_mean)/self.sample_std
        return norm_data

    def train_step(self, X_train, y_train, shuffle_idx):
        # np.random.shuffle(shuffle_idx)
        # X_train = X_train[shuffle_idx]
        # y_train = y_train[shuffle_idx]
        # train_acc = None
        np.random.shuffle(shuffle_idx)
        X_train = X_train[shuffle_idx]
        y_train = y_train[shuffle_idx]
        correct = 0
        n = X_train.shape[0]
        ##############################################
        ## TODO: to implement the training process  ##
        ## and update the weights                   ##
        ##############################################

        for i in range(n):
            x = X_train[i]
            true_label = y_train[i]
            scores = np.dot(self.weight, x)
            predicted_label = np.argmax(scores)
            if predicted_label == true_label:
                correct += 1
            else:
                self.weight[true_label] += self.lr * x
                self.weight[predicted_label] -= self.lr * x
        train_acc = correct / n

        return train_acc

    def test_step(self, X_test, y_test):
        # X_test = self.data_preprocessing(data=X_test)
        # num_sample = X_test.shape[0]
        # test_acc = None

        #########################################
        ##  ToDO: Evaluate the test set and    ##
        ##  return the test acc                ##
        #########################################
        X_test = self.data_preprocessing(data=X_test)
        X_test = np.concatenate([X_test, np.ones((X_test.shape[0], 1))], axis=1)
        num_sample = X_test.shape[0]
        scores = np.dot(self.weight, X_test.T)
        y_pred = np.argmax(scores, axis=0)
        correct = np.sum(y_pred == y_test)
        test_acc = correct / num_sample

        return test_acc

    def train(self):
        self.X_train = self.data_preprocessing(data=self.X_train)
        self.X_train = np.concatenate([self.X_train, np.ones((self.X_train.shape[0], 1))], axis=1)
        num_sample = self.X_train.shape[0]
        # self.X_train = self.data_preprocessing(data=self.X_train)
        # num_sample = self.X_train.shape[0]

        ######################################################
        ### TODO: In order to absorb the bias into weights ###
        ###  we need to modify the input data.            ###
        ###  So You need to transform the input data       ###
        ######################################################


        shuffle_index = np.array(range(0, num_sample))
        for epoch in range(self.num_epoch):
            training_acc = self.train_step(X_train=self.X_train, y_train=self.y_train, shuffle_idx=shuffle_index)
            tst_acc = self.test_step(X_test=self.X_test,  y_test=self.y_test)
            self.total_acc_train.append(training_acc)
            self.total_acc_tst.append(tst_acc)
            print('epoch:', epoch, 'traing_acc:%.3f'%training_acc, 'tst_acc:%.3f'%tst_acc)

    def vis_acc_curve(self):
        train_acc = np.array(self.total_acc_train)
        tst_acc = np.array(self.total_acc_tst)
        plt.plot(train_acc)
        plt.plot(tst_acc)
        plt.legend(['train_acc', 'tst_acc'])
        plt.show()


np.random.seed(0)
random.seed(0)
#######################################################
### TODO:
### 1. You need to import the model and pass some parameters.
### 2. Then training the model with some epoches.
### 3. Visualize the training acc and test acc verus epoches

lr = 0.01
num_epoch = 100

weight_dimension = X_train.shape[1] + 1
num_cls = len(np.unique(y_train))

model = MultiClsPLA(X_train, y_train, X_test, y_test, lr, num_epoch, weight_dimension, num_cls)
model.train()
model.vis_acc_curve()
```

```
epoch: 0 traing_acc:0.752 tst_acc:0.911
epoch: 1 traing_acc:0.905 tst_acc:0.911
epoch: 2 traing_acc:0.943 tst_acc:0.978
epoch: 3 traing_acc:0.914 tst_acc:0.978
epoch: 4 traing_acc:0.905 tst_acc:0.978
epoch: 5 traing_acc:0.924 tst_acc:0.978
epoch: 6 traing_acc:0.990 tst_acc:0.978
epoch: 7 traing_acc:0.914 tst_acc:0.978
epoch: 8 traing_acc:0.962 tst_acc:0.978
epoch: 9 traing_acc:0.905 tst_acc:0.956
epoch: 10 traing_acc:0.952 tst_acc:0.978
epoch: 11 traing_acc:0.962 tst_acc:0.978
epoch: 12 traing_acc:0.952 tst_acc:0.956
epoch: 13 traing_acc:0.952 tst_acc:0.978
epoch: 14 traing_acc:0.971 tst_acc:0.956
epoch: 15 traing_acc:0.962 tst_acc:0.933
epoch: 16 traing_acc:0.933 tst_acc:0.978
epoch: 17 traing_acc:0.981 tst_acc:0.978
epoch: 18 traing_acc:0.971 tst_acc:0.978
epoch: 19 traing_acc:0.981 tst_acc:0.978
epoch: 20 traing_acc:0.962 tst_acc:0.978
epoch: 21 traing_acc:0.943 tst_acc:0.956
epoch: 22 traing_acc:0.971 tst_acc:0.933
epoch: 23 traing_acc:0.943 tst_acc:0.956
epoch: 24 traing_acc:0.971 tst_acc:0.978
epoch: 25 traing_acc:0.943 tst_acc:0.978
epoch: 26 traing_acc:0.981 tst_acc:0.978
epoch: 27 traing_acc:0.971 tst_acc:0.933
epoch: 28 traing_acc:0.981 tst_acc:0.933
epoch: 29 traing_acc:0.981 tst_acc:0.933
epoch: 30 traing_acc:0.943 tst_acc:0.911
epoch: 31 traing_acc:0.952 tst_acc:0.956
epoch: 32 traing_acc:0.971 tst_acc:0.978
epoch: 33 traing_acc:1.000 tst_acc:0.978
epoch: 34 traing_acc:1.000 tst_acc:0.978
epoch: 35 traing_acc:1.000 tst_acc:0.978
epoch: 36 traing_acc:1.000 tst_acc:0.978
epoch: 37 traing_acc:1.000 tst_acc:0.978
epoch: 38 traing_acc:1.000 tst_acc:0.978
epoch: 39 traing_acc:1.000 tst_acc:0.978
epoch: 40 traing_acc:1.000 tst_acc:0.978
epoch: 41 traing_acc:1.000 tst_acc:0.978
epoch: 42 traing_acc:1.000 tst_acc:0.978
epoch: 43 traing_acc:1.000 tst_acc:0.978
epoch: 44 traing_acc:1.000 tst_acc:0.978
epoch: 45 traing_acc:1.000 tst_acc:0.978
epoch: 46 traing_acc:1.000 tst_acc:0.978
epoch: 47 traing_acc:1.000 tst_acc:0.978
epoch: 48 traing_acc:1.000 tst_acc:0.978
epoch: 49 traing_acc:1.000 tst_acc:0.978
epoch: 50 traing_acc:1.000 tst_acc:0.978
epoch: 51 traing_acc:1.000 tst_acc:0.978
epoch: 52 traing_acc:1.000 tst_acc:0.978
epoch: 53 traing_acc:1.000 tst_acc:0.978
epoch: 54 traing_acc:1.000 tst_acc:0.978
epoch: 55 traing_acc:1.000 tst_acc:0.978
epoch: 56 traing_acc:1.000 tst_acc:0.978
epoch: 57 traing_acc:1.000 tst_acc:0.978
epoch: 58 traing_acc:1.000 tst_acc:0.978
epoch: 59 traing_acc:1.000 tst_acc:0.978
epoch: 60 traing_acc:1.000 tst_acc:0.978
epoch: 61 traing_acc:1.000 tst_acc:0.978
epoch: 62 traing_acc:1.000 tst_acc:0.978
epoch: 63 traing_acc:1.000 tst_acc:0.978
epoch: 64 traing_acc:1.000 tst_acc:0.978
epoch: 65 traing_acc:1.000 tst_acc:0.978
epoch: 66 traing_acc:1.000 tst_acc:0.978
epoch: 67 traing_acc:1.000 tst_acc:0.978
epoch: 68 traing_acc:1.000 tst_acc:0.978
epoch: 69 traing_acc:1.000 tst_acc:0.978
epoch: 70 traing_acc:1.000 tst_acc:0.978
epoch: 71 traing_acc:1.000 tst_acc:0.978
epoch: 72 traing_acc:1.000 tst_acc:0.978
epoch: 73 traing_acc:1.000 tst_acc:0.978
epoch: 74 traing_acc:1.000 tst_acc:0.978
epoch: 75 traing_acc:1.000 tst_acc:0.978
epoch: 76 traing_acc:1.000 tst_acc:0.978
epoch: 77 traing_acc:1.000 tst_acc:0.978
epoch: 78 traing_acc:1.000 tst_acc:0.978
epoch: 79 traing_acc:1.000 tst_acc:0.978
epoch: 80 traing_acc:1.000 tst_acc:0.978
epoch: 81 traing_acc:1.000 tst_acc:0.978
epoch: 82 traing_acc:1.000 tst_acc:0.978
epoch: 83 traing_acc:1.000 tst_acc:0.978
epoch: 84 traing_acc:1.000 tst_acc:0.978
epoch: 85 traing_acc:1.000 tst_acc:0.978
```

```
epoch: 86 traing_acc:1.000 tst_acc:0.978
epoch: 87 traing_acc:1.000 tst_acc:0.978
epoch: 88 traing_acc:1.000 tst_acc:0.978
epoch: 89 traing_acc:1.000 tst_acc:0.978
epoch: 90 traing_acc:1.000 tst_acc:0.978
epoch: 91 traing_acc:1.000 tst_acc:0.978
epoch: 92 traing_acc:1.000 tst_acc:0.978
epoch: 93 traing_acc:1.000 tst_acc:0.978
epoch: 94 traing_acc:1.000 tst_acc:0.978
epoch: 95 traing_acc:1.000 tst_acc:0.978
epoch: 96 traing_acc:1.000 tst_acc:0.978
epoch: 97 traing_acc:1.000 tst_acc:0.978
epoch: 98 traing_acc:1.000 tst_acc:0.978
epoch: 99 traing_acc:1.000 tst_acc:0.978
```



knn

March 6, 2025

[6]:

```python
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = "cs231n/assignments/assignment1/"
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
--2025-03-03 17:11:20--  http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)… 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80…
connected.
HTTP request sent, awaiting response… 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: ‘cifar-10-python.tar.gz’

cifar-10-python.tar 100%[===================>] 162.60M  48.5MB/s    in 3.6s

2025-03-03 17:11:24 (44.8 MB/s) - ‘cifar-10-python.tar.gz’ saved
[170498071/170498071]

cifar-10-batches-py/
cifar-10-batches-py/data_batch_4
```

```
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
/content/drive/My Drive/cs231n/assignments/assignment1
```

1 k-Nearest Neighbor (kNN) exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the assignments page on the course website.

The kNN classifier consists of two stages:

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

In this exercise you will implement these steps and understand the basic Image Classification pipeline, cross-validation, and gain proficiency in writing efficient, vectorized code.

[7]:

```python
# Run some setup code for this notebook.

import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

# This is a bit of magic to make matplotlib figures appear inline in the␣
 ↪notebook
# rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# Some more magic so that the notebook will reload external python modules;
# see http://stackoverflow.com/questions/1907993/
 ↪autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

[8]:

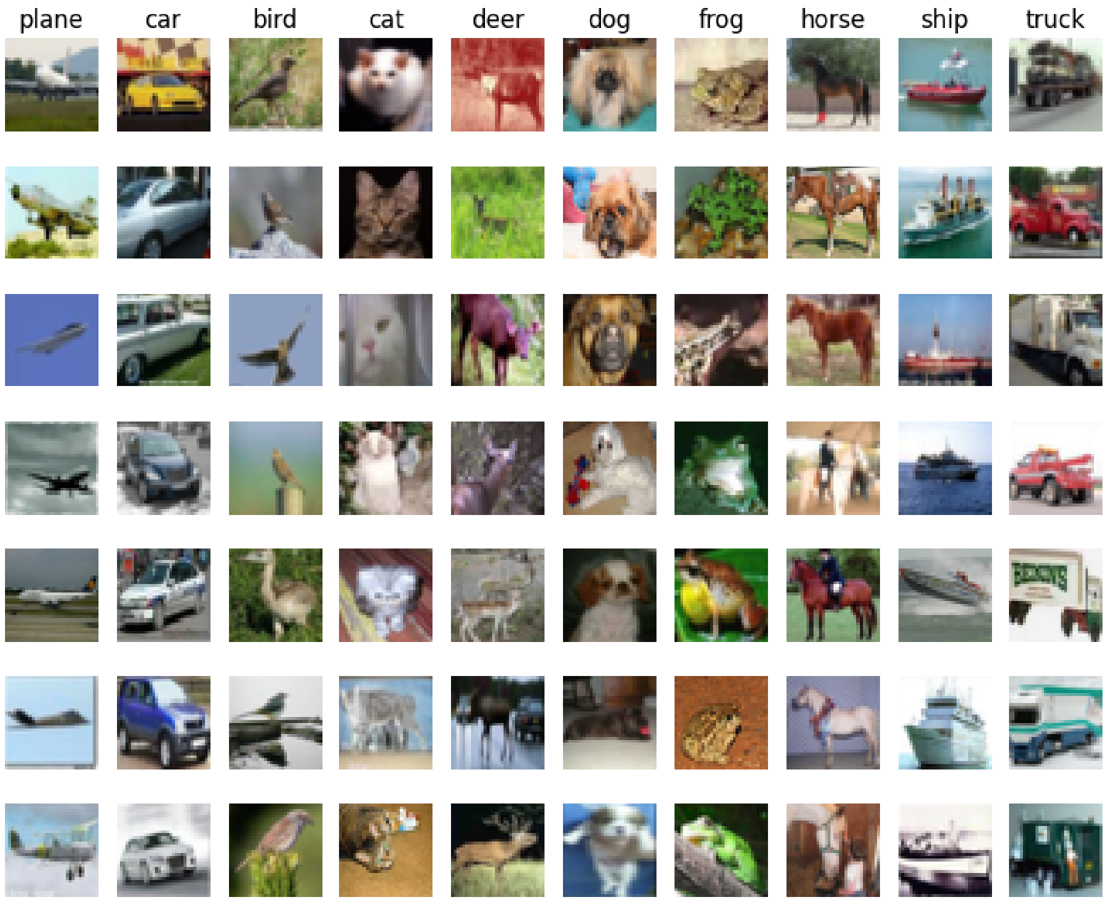
```python
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'
```

```
# Cleaning up variables to prevent loading data multiple times (which may cause␣
 ↪memory issue)
try:
   del X_train, y_train
   del X_test, y_test
   print('Clear previously loaded data.')
except:
   pass

X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

# As a sanity check, we print out the size of the training and test data.
print('Training data shape: ', X_train.shape)
print('Training labels shape: ', y_train.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

[9]:

```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```



[10]:
```
# Subsample the data for more efficient code execution in this exercise
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
y_train = y_train[mask]

num_test = 500
mask = list(range(num_test))
X_test = X_test[mask]
y_test = y_test[mask]

# Reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
print(X_train.shape, X_test.shape)
```

```
(5000, 3072) (500, 3072)
```

[11]:
```
from cs231n.classifiers import KNearestNeighbor

# Create a kNN classifier instance.
# Remember that training a kNN classifier is a noop:
# the Classifier simply remembers the data and does no further processing
classifier = KNearestNeighbor()
classifier.train(X_train, y_train)
```

We would now like to classify the test data with the kNN classifier. Recall that we can break down this process into two steps:

1. First we must compute the distances between all test examples and all train examples.
2. Given these distances, for each test example we find the k nearest examples and have them vote for the label

Lets begin with computing the distance matrix between all training and test examples. For example, if there are **Ntr** training examples and **Nte** test examples, this stage should result in a **Nte x Ntr** matrix where each element (i,j) is the distance between the i-th test and j-th train example.

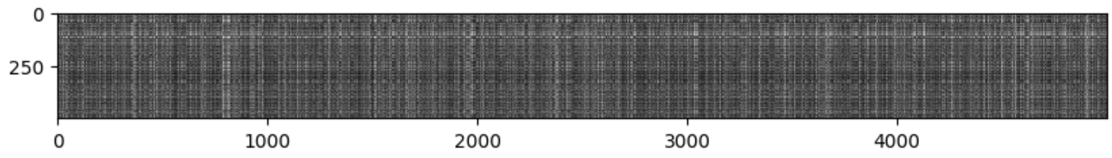
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

First, open `cs231n/classifiers/k_nearest_neighbor.py` and implement the function `compute_distances_two_loops` that uses a (very inefficient) double loop over all pairs of (test, train) examples and computes the distance matrix one element at a time.

[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 :

Bright rows in the distance matrix arise from test images that are overall very dissimilar to the training examples (i.e. they are outliers or contain atypical features). In contrast, bright columns correspond to training images that are, in general, far from most of the test images—again typically because they are unrepresentative or contain unusual variations.

Recall that in our distance matrix, each entry represents the L1 distance between a test image (row) and a training image (column). Since white indicates high distance, a bright row means that a particular test image has high distances to nearly all training images. This suggests that the test image is atypical—it may be noisy, misaligned, or simply an outlier compared to the common patterns in the training set.

Similarly, a bright column means that a certain training image is far from most test images. Such a training example likely embodies unusual characteristics or is an outlier within the training distribution, making it generally dissimilar from the bulk of test images.

Therefore, the structured brightness patterns reveal underlying data irregularities: bright rows pinpoint problematic or atypical test cases, while bright columns indicate unrepresentative training examples.

[14]:

```
# 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`:

[15]:

```
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`.

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. To clarify, both training and test examples are preprocessed in the same way.

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, which means rotating all the images by the same angle. Empty regions in the image caused by rotation are padded with a same pixel value and no interpolation is performed.

$YourAnswer:$ 1, 2, and 3

$YourExplanation:$

1. **Subtracting the mean μ:**
   This operation shifts all pixel values by the same constant. For any two images (x) and (y),
   $$\|(x-\mu)-(y-\mu)\|_1 = \|x-y\|_1.$$
   Thus, the relative L1 distances are unchanged.

2. **Subtracting the per-pixel mean μ_{ij}:**
   Here, each pixel is shifted by a constant specific to its position. For two images, the difference at each pixel is
   $$\left|p_{ij}^{(x)} - \mu_{ij} - (p_{ij}^{(y)} - \mu_{ij})\right| = \left|p_{ij}^{(x)} - p_{ij}^{(y)}\right|,$$
   so the overall L1 distance remains the same.

3. **Subtracting the mean μ and dividing by the standard deviation σ:**
   This rescales every pixel by the same constant $1/\sigma$, so
   $$\left\|\frac{x-\mu}{\sigma} - \frac{y-\mu}{\sigma}\right\|_1 = \frac{1}{\sigma}\|x-y\|_1.$$
   Multiplying by a positive constant does not change the order of distances used in nearest neighbor classification.

4. **Subtracting the pixel-wise mean μ_{ij} and dividing by the pixel-wise standard deviation σ_{ij}:**
   This results in a weighted L1 norm:

$$\sum_{ij} \frac{\left|p_{ij}^{(x)} - p_{ij}^{(y)}\right|}{\sigma_{ij}}.$$

   where each pixel is scaled by a different factor. Since these factors are not constant across pixels, the relative distances between images can change, potentially altering the nearest neighbor relationships.

5. **Rotating the images:**
   Rotating all images changes the order of the pixels. Unless the rotation is a perfect permutation (as with a 90° rotation on a square image without any padding issues), the L1 distance is not preserved. The padding with a constant value and any loss or redistribution of pixel information can alter the distances, affecting the classifier's performance.

Thus, only preprocessing steps 1, 2, and 3 are L1-distance invariant (up to a constant scaling) when applied uniformly to both training and test data.

[ ]:

```
from google.colab import drive
drive.mount('/content/drive')
```

[16]:

```
# Now lets speed up distance matrix computation by using partial vectorization
# with one loop. Implement the function compute_distances_one_loop and run the
# code below:
dists_one = classifier.compute_distances_one_loop(X_test)

# To ensure that our vectorized implementation is correct, we make sure that it
# agrees with the naive implementation. There are many ways to decide whether
# two matrices are similar; one of the simplest is the Frobenius norm. In case
# you haven't seen it before, the Frobenius norm of two matrices is the square
# root of the squared sum of differences of all elements; in other words,␣
 ↪reshape
# the matrices into vectors and compute the Euclidean distance between them.
difference = np.linalg.norm(dists - dists_one, ord='fro')
print('One loop difference was: %f' % (difference, ))
if difference < 0.001:
    print('Good! The distance matrices are the same')
else:
    print('Uh-oh! The distance matrices are different')
```

```
One loop difference was: 0.000000
Good! The distance matrices are the same
```

[17]:

```
# Now implement the fully vectorized version inside compute_distances_no_loops
# and run the code
dists_two = classifier.compute_distances_no_loops(X_test)

# check that the distance matrix agrees with the one we computed before:
```

```
difference = np.linalg.norm(dists - dists_two, ord='fro')
print('No loop difference was: %f' % (difference, ))
if difference < 0.001:
    print('Good! The distance matrices are the same')
else:
    print('Uh-oh! The distance matrices are different')
```

```
No loop difference was: 0.000000
Good! The distance matrices are the same
```

[18]:

```
# 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 36.998218 seconds
One loop version took 35.071851 seconds
No loop version took 0.597502 seconds
```

1.0.1 Cross-validation

We have implemented the k-Nearest Neighbor classifier but we set the value k = 5 arbitrarily. We will now determine the best value of this hyperparameter with cross-validation.

[19]:

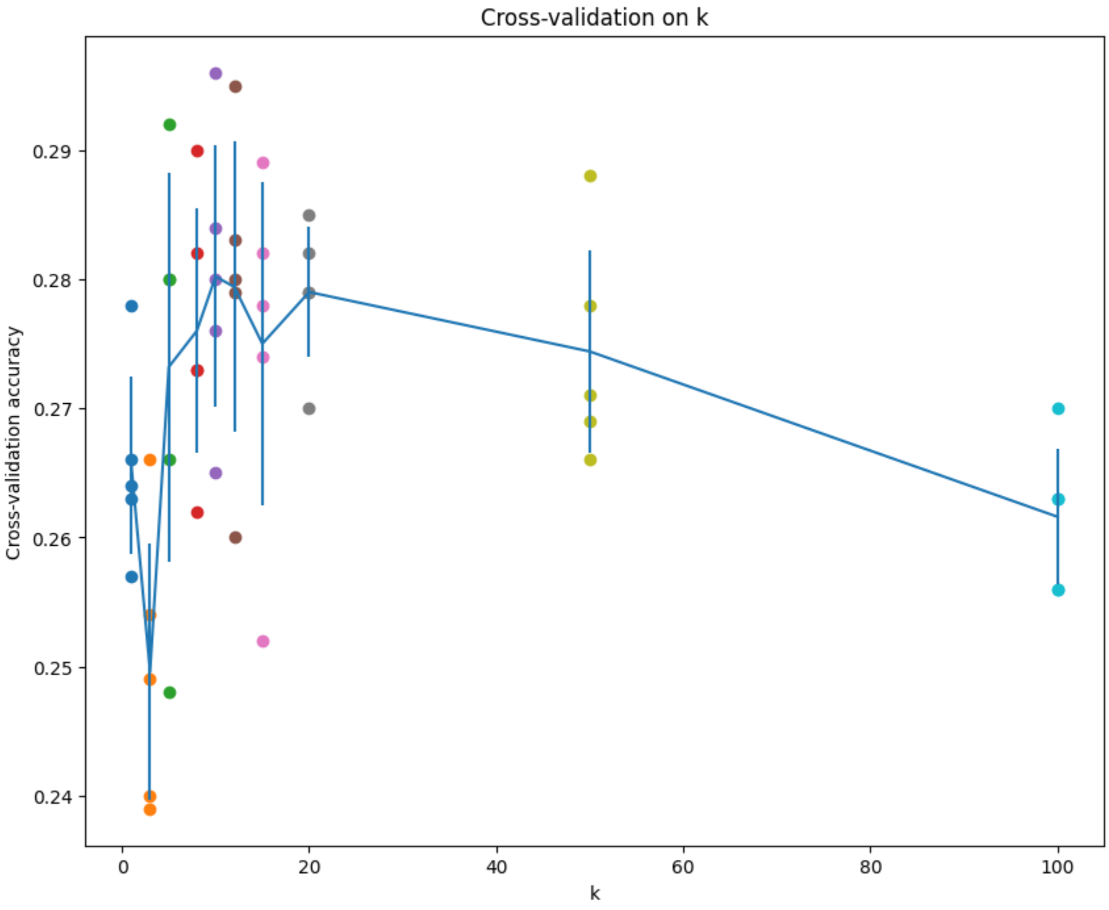
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

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)*****

avg_size = int(X_train.shape[0] / num_folds) # will abandon the rest if not␣
 ↪divided evenly.
for i in range(num_folds):
  X_train_folds.append(X_train[i * avg_size : (i+1) * avg_size])
  y_train_folds.append(y_train[i * avg_size : (i+1) * avg_size])

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

# A dictionary holding the accuracies for different values of k that we find
# when running cross-validation. After running cross-validation,
# k_to_accuracies[k] should be a list of length num_folds giving the different
# accuracy values that we found when using that value of k.
k_to_accuracies = {}


################################################################################
# TODO:                                                                        #
# Perform k-fold cross validation to find the best value of k. For each        #
# possible value of k, run the k-nearest-neighbor algorithm num_folds times,   #
# where in each case you use all but one of the folds as training data and the #
# last fold as a validation set. Store the accuracies for all fold and all     #
# values of k in the k_to_accuracies dictionary.                               #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
for k in k_choices:
  accuracies = []
  for i in range(num_folds):
      X_train_cv = np.vstack(X_train_folds[0:i] + X_train_folds[i+1:])
      y_train_cv = np.hstack(y_train_folds[0:i] + y_train_folds[i+1:])
      X_valid_cv = X_train_folds[i]
      y_valid_cv = y_train_folds[i]

      classifier.train(X_train_cv, y_train_cv)
      dists = classifier.compute_distances_no_loops(X_valid_cv)
      accuracy = float(np.sum(classifier.predict_labels(dists, k) ==␣
 ↪y_valid_cv)) / y_valid_cv.shape[0]
      accuracies.append(accuracy)
  k_to_accuracies[k] = accuracies
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# Print out the computed accuracies
for k in sorted(k_to_accuracies):
    for accuracy in k_to_accuracies[k]:
        print('k = %d, accuracy = %f' % (k, accuracy))
```

```
k = 1, accuracy = 0.263000
k = 1, accuracy = 0.257000
k = 1, accuracy = 0.264000
k = 1, accuracy = 0.278000
k = 1, accuracy = 0.266000
k = 3, accuracy = 0.239000
k = 3, accuracy = 0.249000
k = 3, accuracy = 0.240000
k = 3, accuracy = 0.266000
k = 3, accuracy = 0.254000
k = 5, accuracy = 0.248000
k = 5, accuracy = 0.266000
k = 5, accuracy = 0.280000
k = 5, accuracy = 0.292000
k = 5, accuracy = 0.280000
k = 8, accuracy = 0.262000
k = 8, accuracy = 0.282000
k = 8, accuracy = 0.273000
k = 8, accuracy = 0.290000
k = 8, accuracy = 0.273000
k = 10, accuracy = 0.265000
k = 10, accuracy = 0.296000
k = 10, accuracy = 0.276000
k = 10, accuracy = 0.284000
k = 10, accuracy = 0.280000
k = 12, accuracy = 0.260000
k = 12, accuracy = 0.295000
k = 12, accuracy = 0.279000
k = 12, accuracy = 0.283000
k = 12, accuracy = 0.280000
k = 15, accuracy = 0.252000
k = 15, accuracy = 0.289000
k = 15, accuracy = 0.278000
k = 15, accuracy = 0.282000
k = 15, accuracy = 0.274000
k = 20, accuracy = 0.270000
```

```
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.282000
k = 20, accuracy = 0.285000
k = 50, accuracy = 0.271000
k = 50, accuracy = 0.288000
k = 50, accuracy = 0.278000
k = 50, accuracy = 0.269000
k = 50, accuracy = 0.266000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.270000
k = 100, accuracy = 0.263000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.263000
```

[20]:

```
# plot the raw observations
for k in k_choices:
    accuracies = k_to_accuracies[k]
    plt.scatter([k] * len(accuracies), accuracies)

# plot the trend line with error bars that correspond to standard deviation
accuracies_mean = np.array([np.mean(v) for k,v in sorted(k_to_accuracies.
 ↪items())])
accuracies_std = np.array([np.std(v) for k,v in sorted(k_to_accuracies.
 ↪items())])
plt.errorbar(k_choices, accuracies_mean, yerr=accuracies_std)
plt.title('Cross-validation on k')
plt.xlabel('k')
plt.ylabel('Cross-validation accuracy')
plt.show()
```



[21]:

```
# 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 = 1

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

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.

$YourAnswer :$ 2 and 4

$YourExplanation :$

1. **False** - The decision boundary of the k-NN classifier is not linear. It depends on the spatial distribution of the training data points and can be non-linear, especially in cases where data distributions are complex.
2. **True** - The training error of a 1-NN classifier is always zero because each training example is its own nearest neighbor, ensuring perfect classification on the training set. For 5-NN, the training error can be higher due to the possibility of neighboring points (including itself) being incorrectly labeled.
3. **False** - The test error of a 1-NN classifier can be higher than that of a 5-NN classifier, especially in noisy datasets. A larger k (like 5) can reduce overfitting and improve generalization, leading to lower test error.
4. **True** - The computational time required to classify a test example with k-NN increases with the size of the training set. This is because k-NN must compute distances to all training examples to find the nearest neighbors, leading to higher computational complexity as the training set grows.

svm

March 6, 2025

1 Multiclass Support Vector Machine exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the assignments page on the course website.

In this exercise you will:

- implement a fully-vectorized **loss function** for the SVM
- implement the fully-vectorized expression for its **analytic gradient**
- **check your implementation** using numerical gradient
- use a validation set to **tune the learning rate and regularization** strength
- **optimize** the loss function with **SGD**
- **visualize** the final learned weights

[2]:

```
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = "cs231n/assignments/assignment1/"
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
--2025-03-05 06:48:15--  http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)… 128.100.3.30
```

```
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80…
connected.
HTTP request sent, awaiting response… 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: ‘cifar-10-python.tar.gz’

cifar-10-python.tar 100%[===================>] 162.60M  14.8MB/s    in 13s

2025-03-05 06:48:28 (13.0 MB/s) - ‘cifar-10-python.tar.gz’ saved
[170498071/170498071]

cifar-10-batches-py/
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
/content/drive/My Drive/cs231n/assignments/assignment1
```

[3]:

```python
# Run some setup code for this notebook.
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

# This is a bit of magic to make matplotlib figures appear inline in the
# notebook rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# Some more magic so that the notebook will reload external python modules;
# see http://stackoverflow.com/questions/1907993/
  ↪autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

[ ]:

```python
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

1.1 CIFAR-10 Data Loading and Preprocessing

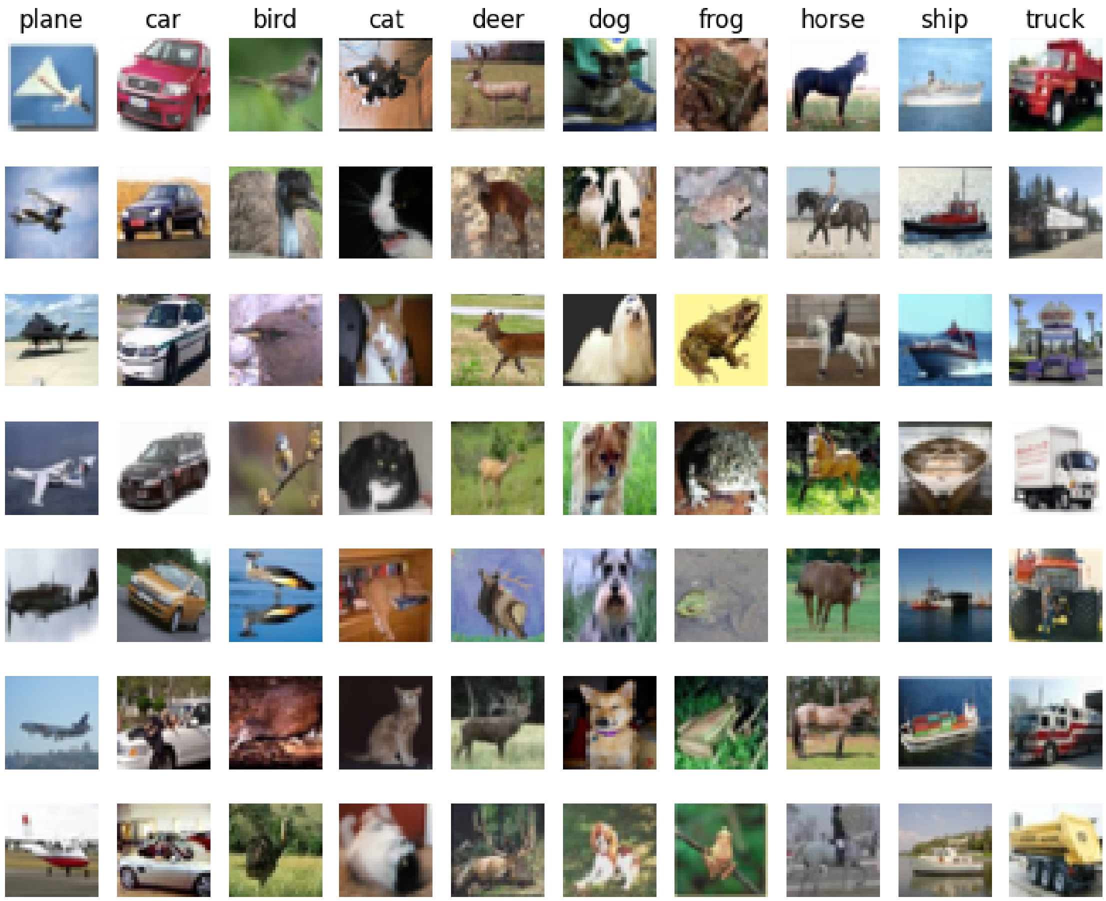
[4]:

```
# Load the raw CIFAR-10 data.
cifar10_dir = '/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/
 ↪datasets/cifar-10-batches-py'

# Cleaning up variables to prevent loading data multiple times (which may cause␣
 ↪memory issue)
try:
   del X_train, y_train
   del X_test, y_test
   print('Clear previously loaded data.')
except:
   pass

X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

# As a sanity check, we print out the size of the training and test data.
print('Training data shape: ', X_train.shape)
print('Training labels shape: ', y_train.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Training data shape:  (50000, 32, 32, 3)
Training labels shape:  (50000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

[5]:

```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```



[6]:

```
# 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 can use this for development so our code runs faster.
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)
```

```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

[7]:

```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

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

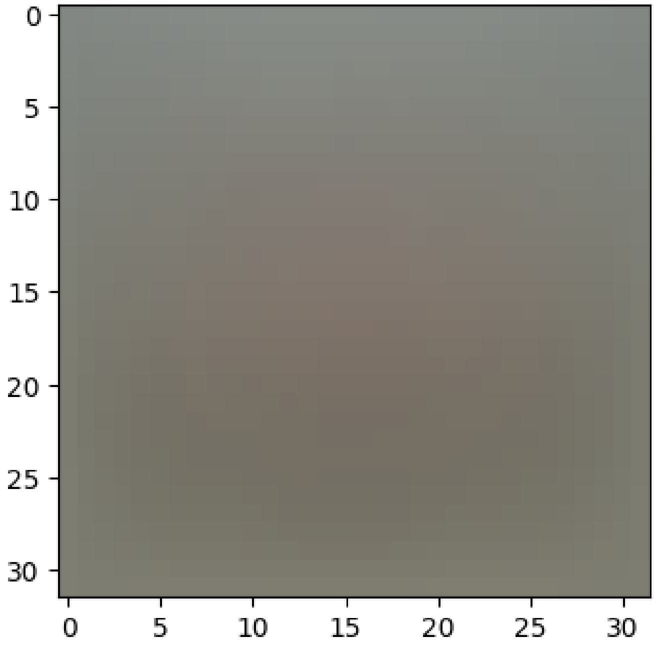
```
Training data shape:  (49000, 3072)
Validation data shape:  (1000, 3072)
Test data shape:  (1000, 3072)
dev data shape:  (500, 3072)
```

[8]:

```
# Preprocessing: subtract the mean image
# first: compute the image mean based on the training data
mean_image = np.mean(X_train, axis=0)
```

```
print(mean_image[:10]) # print a few of the elements
plt.figure(figsize=(4,4))
plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize the mean␣
  ↪image
plt.show()

# second: subtract the mean image from train and test data
X_train -= mean_image
X_val -= mean_image
X_test -= mean_image
X_dev -= mean_image

# third: 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.
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)
```

```
[130.64189796 135.98173469 132.47391837 130.05569388 135.34804082
 131.75402041 130.96055102 136.14328571 132.47636735 131.48467347]
```



```
(49000, 3073) (1000, 3073) (1000, 3073) (500, 3073)
```

1.2 SVM Classifier

Your code for this section will all be written inside `cs231n/classifiers/linear_svm.py`.

As you can see, we have prefilled the function `svm_loss_naive` which uses for loops to evaluate the multiclass SVM loss function.

[33]:

```python
# 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: 8.754895
```

The `grad` returned from the function above is right now all zero. Derive and implement the gradient for the SVM cost function and implement it inline inside the function `svm_loss_naive`. You will find it helpful to interleave your new code inside the existing function.

To check that you have correctly implemented the gradient correctly, you can numerically estimate the gradient of the loss function and compare the numeric estimate to the gradient that you computed. We have provided code that does this for you:

[34]:

```python
# Once you've implemented the gradient, recompute it with the code below
# and gradient check it with the function we provided for you

# Compute the loss and its gradient at W.
loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.0)

# Numerically compute the gradient along several randomly chosen dimensions, and
# compare them with your analytically computed gradient. The numbers should␣
 ↪match
# almost exactly along all dimensions.
from cs231n.gradient_check import grad_check_sparse
f = lambda w: svm_loss_naive(w, X_dev, y_dev, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad)

# do the gradient check once again with regularization turned on
# you didn't forget the regularization gradient did you?
loss, grad = svm_loss_naive(W, X_dev, y_dev, 5e1)
f = lambda w: svm_loss_naive(w, X_dev, y_dev, 5e1)[0]
grad_numerical = grad_check_sparse(f, W, grad)
```

```
numerical: -5.499235 analytic: -5.499235, relative error: 9.919427e-12
numerical: 4.207009 analytic: 4.207009, relative error: 2.861345e-11
numerical: -8.387595 analytic: -8.387595, relative error: 6.428698e-12
```

```
numerical: 9.873773 analytic: 9.873773, relative error: 2.618244e-11
numerical: 32.736160 analytic: 32.736160, relative error: 3.383830e-13
numerical: -44.071831 analytic: -44.071831, relative error: 6.929076e-12
numerical: 7.954467 analytic: 7.954467, relative error: 1.531845e-11
numerical: 23.921498 analytic: 23.921498, relative error: 1.226471e-11
numerical: -12.327717 analytic: -12.327717, relative error: 1.530947e-11
numerical: -0.597515 analytic: -0.597515, relative error: 5.387105e-10
numerical: 13.459376 analytic: 13.459376, relative error: 2.894586e-11
numerical: -6.281317 analytic: -6.281317, relative error: 9.906627e-12
numerical: 1.500570 analytic: 1.500570, relative error: 1.950670e-10
numerical: 11.303950 analytic: 11.303950, relative error: 1.549228e-11
numerical: 3.186412 analytic: 3.186412, relative error: 2.950658e-11
numerical: 13.317858 analytic: 13.317858, relative error: 8.665334e-12
numerical: 17.814628 analytic: 17.814628, relative error: 5.517341e-12
numerical: -2.431046 analytic: -2.431046, relative error: 1.217943e-10
numerical: -20.622950 analytic: -20.622950, relative error: 7.635002e-13
numerical: -8.409510 analytic: -8.409510, relative error: 5.211091e-13
```

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*

$\color{blue}{\textit{Your Answer :}}$ The discrepancy in the gradient check could be caused by the non-differentiability of the SVM loss function. The SVM loss function is not strictly differentiable, especially when the margin is exactly zero. In such cases, the gradient is not uniquely defined, and different implementations might choose different subgradients, leading to small discrepancies between the numerical and analytical gradients.

This is not necessarily a reason for concern if the discrepancy is small. However, if the discrepancy is large, it might indicate an error in the gradient implementation.

Changing the margin would affect the frequency of this happening. A larger margin would make the loss function less likely to have points where the margin is exactly zero, reducing the frequency of non-differentiable points and thus the frequency of gradient check failures. Conversely, a smaller margin would increase the likelihood of such points, potentially increasing the frequency of discrepancies.

The discrepancy is caused by the non-differentiability of the SVM loss function when the margin is zero. This is not a major concern if the discrepancy is small. A one-dimensional example is when the margin is exactly zero, causing the numerical and analytical gradients to differ. Increasing the margin reduces the frequency of such occurrences.

[35]:
```
# 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: 8.754895e+00 computed in 0.098619s
Vectorized loss: 8.754895e+00 computed in 0.012916s
difference: 0.000000
```

[12]:

```
# Complete the implementation of svm_loss_vectorized, and compute the gradient
# of the loss function in a vectorized way.

# The naive implementation and the vectorized implementation should match, but
# the vectorized version should still be much faster.
tic = time.time()
_, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss and gradient: computed in %fs' % (toc - tic))

tic = time.time()
_, grad_vectorized = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Vectorized loss and gradient: computed in %fs' % (toc - tic))

# The loss is a single number, so it is easy to compare the values computed
# by the two implementations. The gradient on the other hand is a matrix, so
# we use the Frobenius norm to compare them.
difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print('difference: %f' % difference)
```

```
Naive loss and gradient: computed in 0.021401s
Vectorized loss and gradient: computed in 0.000521s
difference: 0.000000
```

1.2.1 Stochastic Gradient Descent

We now have vectorized and efficient expressions for the loss, the gradient and our gradient matches the numerical gradient. We are therefore ready to do SGD to minimize the loss. Your code for this part will be written inside `cs231n/classifiers/linear_classifier.py`.

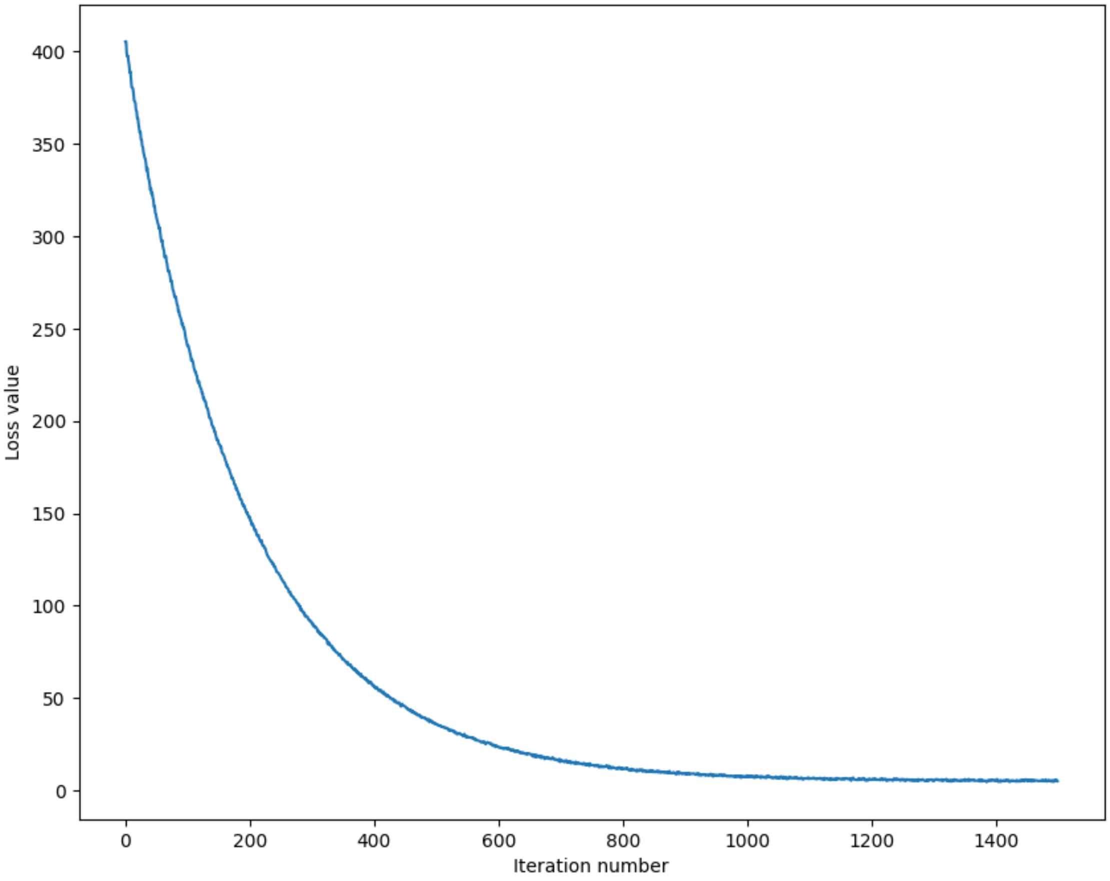
[47]:

```
# 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 405.322677
iteration 100 / 1500: loss 240.435149
iteration 200 / 1500: loss 148.033423
iteration 300 / 1500: loss 90.772534
iteration 400 / 1500: loss 56.081496
iteration 500 / 1500: loss 35.883183
iteration 600 / 1500: loss 23.349041
iteration 700 / 1500: loss 16.517434
iteration 800 / 1500: loss 12.034489
iteration 900 / 1500: loss 8.961969
iteration 1000 / 1500: loss 8.089569
iteration 1100 / 1500: loss 6.624396
iteration 1200 / 1500: loss 5.768331
iteration 1300 / 1500: loss 5.044981
iteration 1400 / 1500: loss 5.198462
That took 9.142997s
```

[48]:

```
# 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()
```



[38]:

```
# 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: 0.377408
validation accuracy: 0.374000
```

[43]:

```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of about 0.39 on the validation set.

# Note: you may see runtime/overflow warnings during hyper-parameter search.
# This may be caused by extreme values, and is not a bug.

# results is dictionary mapping tuples of the form
# (learning_rate, regularization_strength) to tuples of the form
# (training_accuracy, validation_accuracy). The accuracy is simply the fraction
# of data points that are correctly classified.
results = {}
best_val = -1   # The highest validation accuracy that we have seen so far.
best_svm = None # The LinearSVM object that achieved the highest validation␣
 ↪rate.

################################################################################
# TODO:                                                                        #
# Write code that chooses the best hyperparameters by tuning on the validation #
# set. For each combination of hyperparameters, train a linear SVM on the      #
# training set, compute its accuracy on the training and validation sets, and  #
# store these numbers in the results dictionary. In addition, store the best   #
# validation accuracy in best_val and the LinearSVM object that achieves this  #
# accuracy in best_svm.                                                        #
#                                                                              #
# Hint: You should use a small value for num_iters 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 value for num_iters.                                      #
################################################################################

# Provided as a reference. You may or may not want to change these␣
 ↪hyperparameters
learning_rates = [1e-7, 5e-5]
regularization_strengths = [2.5e4, 5e4]

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

for lr in learning_rates:
    for rs in regularization_strengths:
        svm = LinearSVM()
        loss_hist = svm.train(X_train, y_train, learning_rate=lr, reg=rs,
                      num_iters=1500, verbose=True)
        y_train_pred = svm.predict(X_train)
        train_acc = np.mean(y_train == y_train_pred)
        y_val_pred = svm.predict(X_val)
        val_acc = np.mean(y_val == y_val_pred)

        results[(lr, rs)] = (train_acc, val_acc)

        if val_acc > best_val:
            best_val = val_acc
            best_svm = svm

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

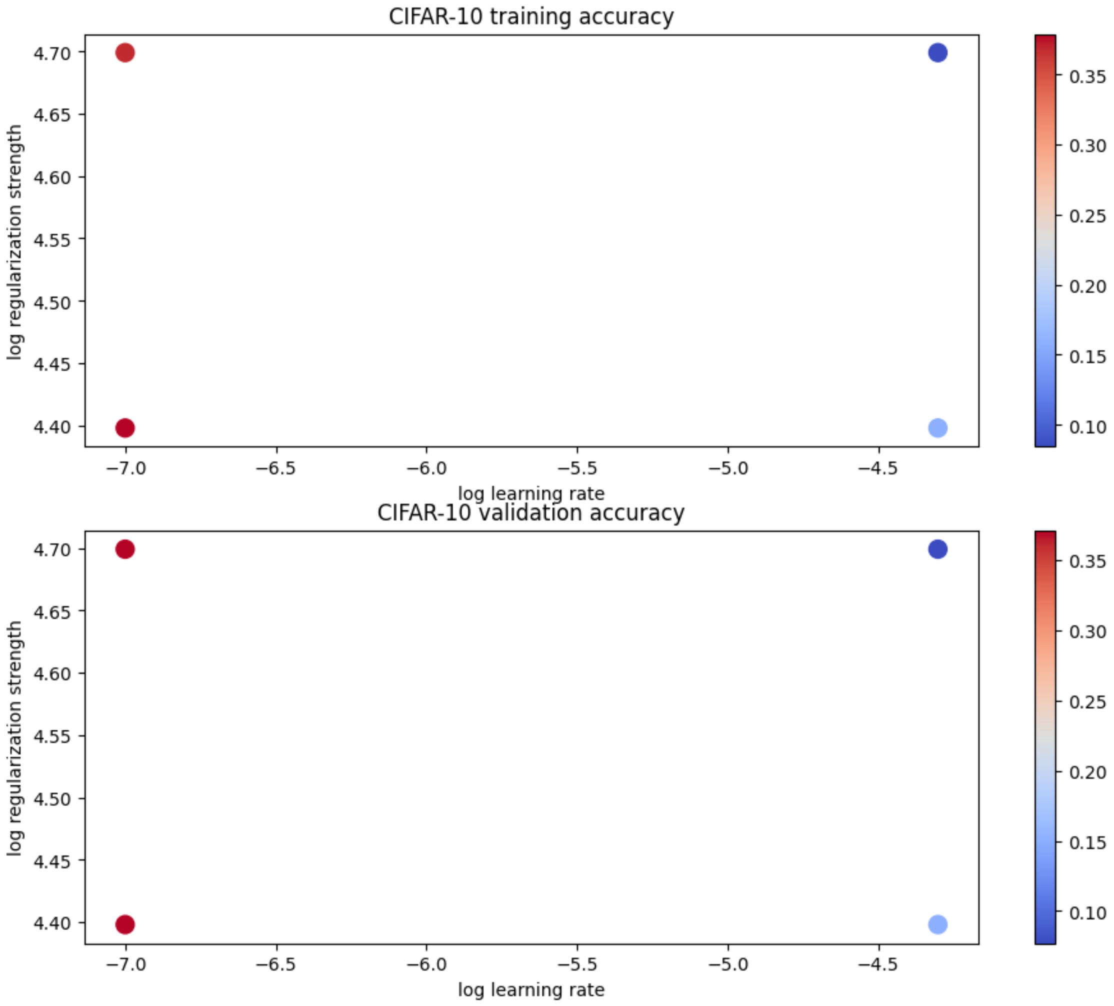
```python
# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy))

print('best validation accuracy achieved during cross-validation: %f' %␣
 ↪best_val)
```

```
iteration 0 / 1500: loss 407.184330
iteration 100 / 1500: loss 240.587674
iteration 200 / 1500: loss 146.621398
iteration 300 / 1500: loss 90.482457
iteration 400 / 1500: loss 57.092392
iteration 500 / 1500: loss 35.536968
iteration 600 / 1500: loss 23.280010
iteration 700 / 1500: loss 16.573400
iteration 800 / 1500: loss 11.120189
iteration 900 / 1500: loss 8.771876
iteration 1000 / 1500: loss 7.603269
iteration 1100 / 1500: loss 6.625076
iteration 1200 / 1500: loss 6.433018
iteration 1300 / 1500: loss 5.338744
iteration 1400 / 1500: loss 5.175254
iteration 0 / 1500: loss 803.762411
iteration 100 / 1500: loss 290.377368
iteration 200 / 1500: loss 108.764013
iteration 300 / 1500: loss 42.958484
iteration 400 / 1500: loss 19.385431
iteration 500 / 1500: loss 9.578206
iteration 600 / 1500: loss 6.786690
iteration 700 / 1500: loss 5.943175
iteration 800 / 1500: loss 5.384997
iteration 900 / 1500: loss 4.939267
iteration 1000 / 1500: loss 5.301167
iteration 1100 / 1500: loss 4.789306
iteration 1200 / 1500: loss 5.252710
iteration 1300 / 1500: loss 5.462931
iteration 1400 / 1500: loss 5.099795
iteration 0 / 1500: loss 408.755478
iteration 100 / 1500: loss 813.483139
iteration 200 / 1500: loss 1099.060462
iteration 300 / 1500: loss 1107.650917
iteration 400 / 1500: loss 1051.238664
iteration 500 / 1500: loss 1258.253564
iteration 600 / 1500: loss 928.126392
iteration 700 / 1500: loss 885.313258
iteration 800 / 1500: loss 1029.197797
iteration 900 / 1500: loss 1169.235797
iteration 1000 / 1500: loss 1326.910182
iteration 1100 / 1500: loss 1069.788620
iteration 1200 / 1500: loss 1114.406718
iteration 1300 / 1500: loss 1069.792073
iteration 1400 / 1500: loss 1176.919929
iteration 0 / 1500: loss 790.699360
iteration 100 / 1500: loss 422183012388442770692759374541648560128.000000
iteration 200 / 1500: loss 697834707173412798808128212897339656111722146410062484
06135896444663496704.000000
iteration 300 / 1500: loss 115346488192601159297486444997548785922530678433149221
68994241799419081437019356605093719632729678984928821248.000000
iteration 400 / 1500: loss 190658506973050375165316839422354549580154974218381703
12372298075857667803027955251873316653355794461982717206515421655539146595369658
79923212288.000000
iteration 500 / 1500: loss 315143242336912469434726458717459045692057202281290704
57436905190395883454105106712013261283240708813250961305888301393056193123626346
737845707824263161058374611149377526797277593 6.000000
iteration 600 / 1500: loss 520906540009044245891327375079789159214920255788280097
50550135670368392678144070480768985648932104727677711879900630925543198698748389
89037965577053149547410932732172474524951302128359250597631646496756963257771622
4.000000
iteration 700 / 1500: loss 861016791640757497114801616904876155593381430474285836
25703463157203979956334110501218338202325541971659772576485142420534126923842754
57161365077993459778556122426304362053319381917589313134554155015375799386478618
46100117313692150210316156365753548 8.000000
iteration 800 / 1500: loss 142319179842601246979878469908804774898071373192583911
57602970644663919442879440963789152140498938269791387843109465609074573677891717
36000737873510532625487319891343610032454537500268287953624256856589489536565314
5985292197910354715416500938879321653069483399991377593111461530183925 76.000000
iteration 900 / 1500: loss inf
iteration 1000 / 1500: loss inf
iteration 1100 / 1500: loss inf
iteration 1200 / 1500: loss inf
iteration 1300 / 1500: loss inf
iteration 1400 / 1500: loss inf
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.378918 val accuracy: 0.371000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.368449 val accuracy: 0.370000
lr 5.000000e-05 reg 2.500000e+04 train accuracy: 0.157020 val accuracy: 0.151000
lr 5.000000e-05 reg 5.000000e+04 train accuracy: 0.084347 val accuracy: 0.077000
best validation accuracy achieved during cross-validation: 0.371000
```

[44]:

```
# Visualize the cross-validation results
import math
import pdb
```

```
# pdb.set_trace()

x_scatter = [math.log10(x[0]) for x in results]
y_scatter = [math.log10(x[1]) for x in results]

# plot training accuracy
marker_size = 100
colors = [results[x][0] for x in results]
plt.subplot(2, 1, 1)
plt.tight_layout(pad=3)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap=plt.cm.coolwarm)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 training accuracy')

# plot validation accuracy
colors = [results[x][1] for x in results] # default size of markers is 20
plt.subplot(2, 1, 2)
plt.scatter(x_scatter, y_scatter, marker_size, c=colors, cmap=plt.cm.coolwarm)
plt.colorbar()
plt.xlabel('log learning rate')
plt.ylabel('log regularization strength')
plt.title('CIFAR-10 validation accuracy')
plt.show()
```



[45]:

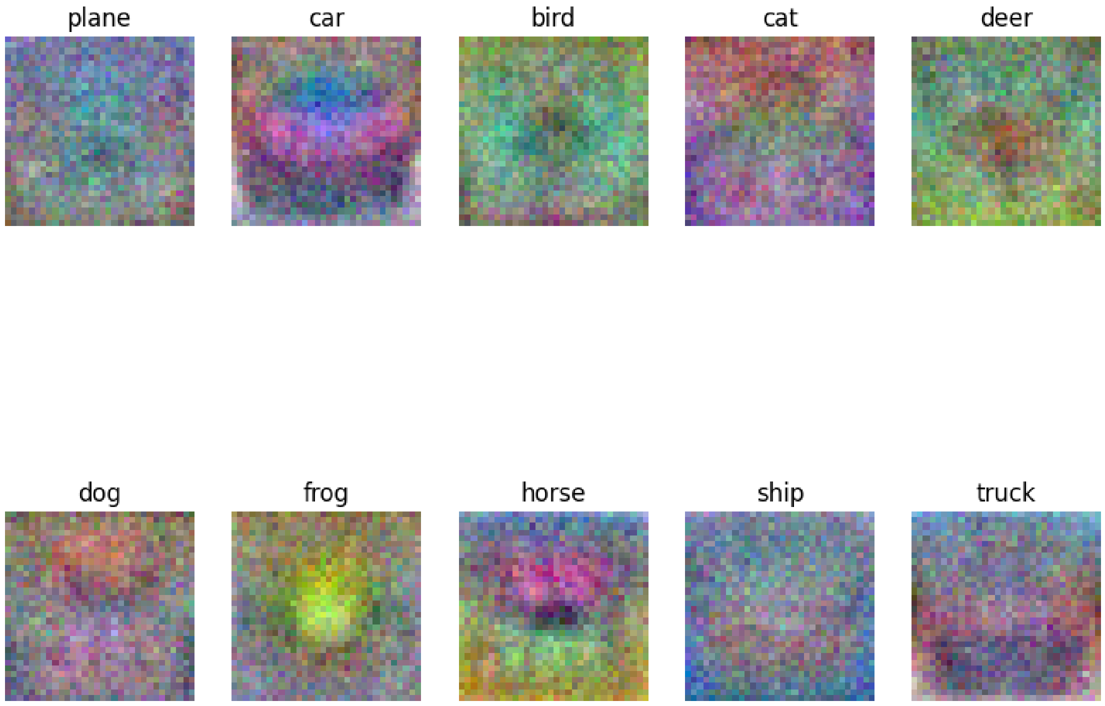
```
# Evaluate the best svm on test set
y_test_pred = best_svm.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('linear SVM on raw pixels final test set accuracy: %f' % test_accuracy)
```

```
linear SVM on raw pixels final test set accuracy: 0.365000
```

[46]:

```
# 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 they way that they do.

YourAnswer : The visualized SVM weights appear as a grid of images, each corresponding to a specific class (e.g., plane, car, bird, etc.). These images show patterns that the SVM has learned to associate with each class. The patterns are often somewhat abstract and may not immediately resemble the actual objects they represent. They can appear as random noise or have regions of color and texture that hint at the features the SVM uses to distinguish between classes.

The reason they look this way is due to the nature of the SVM algorithm. SVMs learn to separate different classes by finding the optimal hyperplane in the feature space. The weights represent the importance of each feature (pixel, in this case) for the decision boundary. When visualized, these weights show the patterns that the SVM has identified as important for classification. The abstract appearance is because the weights are not directly representing the images but rather the coefficients that, when combined with the input features, maximize the margin between different classes.

The visualized SVM weights appear as abstract patterns that the SVM has learned to associate with each class. They look this way because the weights represent the coefficients that define the decision boundary, which may not directly resemble the actual images but rather the features important for classification.

Softmax exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the assignments page on the course website.

This exercise is analogous to the SVM exercise. You will:

- implement a fully-vectorized **loss function** for the Softmax classifier
- implement the fully-vectorized expression for its **analytic gradient**
- **check your implementation** with numerical gradient
- use a validation set to **tune the learning rate and regularization** strength
- **optimize** the loss function with **SGD**
- **visualize** the final learned weights

In [2]:

```
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = "cs231n/assignments/assignment1/"
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
--2025-03-05 08:09:34--  http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: ‘cifar-10-python.tar.gz’

cifar-10-python.tar 100%[===================>] 162.60M  50.2MB/s    in 3.4s

2025-03-05 08:09:38 (47.5 MB/s) - ‘cifar-10-python.tar.gz’ saved [170498071/170498071]

cifar-10-batches-py/
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
/content/drive/My Drive/cs231n/assignments/assignment1
```

In [3]:

```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading extenrnal modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

In [4]:

```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, num_dev=500):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the linear classifier. These are the same steps as we used for the
    SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

    # Cleaning up variables to prevent loading data multiple times (which may cause memory issue)
    try:
       del X_train, y_train
       del X_test, y_test
       print('Clear previously loaded data.')
    except:
       pass

    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]
    mask = np.random.choice(num_training, num_dev, replace=False)
    X_dev = X_train[mask]
    y_dev = y_train[mask]

    # Preprocessing: reshape the image data into rows
    X_train = np.reshape(X_train, (X_train.shape[0], -1))
    X_val = np.reshape(X_val, (X_val.shape[0], -1))
    X_test = np.reshape(X_test, (X_test.shape[0], -1))
    X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis = 0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image
    X_dev -= mean_image

    # add bias dimension and transform into columns
    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))])

    return X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev = get_CIFAR10_data()
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)
print('dev data shape: ', X_dev.shape)
print('dev labels shape: ', y_dev.shape)
```

```
Train data shape:  (49000, 3073)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3073)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3073)
Test labels shape:  (1000,)
dev data shape:  (500, 3073)
dev labels shape:  (500,)
```

Softmax Classifier

Your code for this section will all be written inside `cs231n/classifiers/softmax.py`.

In [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: 2.360174
sanity check: 2.302585
```

Inline Question 1

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

$Your Answer:$ It expects our loss to be close to $-log(0.1)$, the reason is that: with random weights, the model's predictions are essentially random guesses.

In a classification problem with 10 classes, if the model has not learned anything and assigns equal probability to each class, the probability for the correct class would be approximately 0.1. The loss function used here is the cross-entropy loss, which for a single example with true label y and predicted probability p_y is given by $-\log(p_y)$. If $p_y = 0.1$, then the loss would be $-\log(0.1) \approx 2.3026$. This serves as a baseline to check if the loss is reasonable before any training has occurred.

With random weights, the model assigns approximately equal probability to each of the 10 classes. The expected loss is close to $-log(0.1)$ because the correct class has a probability of about 0.1, and the cross-entropy loss for this probability is $-log(0.1)$.

In [9]:

```
# 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: 0.080408 analytic: 0.080408, relative error: 4.117212e-07
numerical: 0.667580 analytic: 0.667580, relative error: 8.101678e-08
numerical: 1.139377 analytic: 1.139377, relative error: 4.304391e-08
numerical: -0.565619 analytic: -0.565619, relative error: 5.863109e-08
numerical: -1.630414 analytic: -1.630414, relative error: 4.994217e-08
numerical: 1.593042 analytic: 1.593042, relative error: 1.127354e-08
numerical: 0.804008 analytic: 0.804008, relative error: 4.003464e-08
numerical: -0.919757 analytic: -0.919757, relative error: 1.501712e-09
numerical: -0.589818 analytic: -0.589818, relative error: 6.566720e-08
numerical: 1.383674 analytic: 1.383674, relative error: 5.198213e-09
numerical: 2.500708 analytic: 2.500708, relative error: 1.388971e-09
numerical: -0.090659 analytic: -0.090659, relative error: 6.224561e-07
numerical: -1.292699 analytic: -1.292699, relative error: 2.593231e-08
numerical: 2.830130 analytic: 2.830130, relative error: 2.350799e-08
numerical: -1.111680 analytic: -1.111680, relative error: 1.878130e-08
numerical: 2.614292 analytic: 2.614292, relative error: 2.520747e-08
numerical: -3.523202 analytic: -3.523202, relative error: 9.244864e-09
numerical: 3.170524 analytic: 3.170523, relative error: 1.612754e-08
numerical: 0.376067 analytic: 0.376067, relative error: 5.955728e-08
numerical: 0.796215 analytic: 0.796215, relative error: 4.942707e-09
```

In [10]:

```
# 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: 2.360174e+00 computed in 0.282045s
vectorized loss: 2.360174e+00 computed in 0.031356s
Loss difference: 0.000000
Gradient difference: 0.000000
```

In [11]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of over 0.35 on the validation set.

from cs231n.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None

################################################################################
# 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 softmax classifer in best_softmax.                          #
################################################################################

# Provided as a reference. You may or may not want to change these hyperparameters
learning_rates = [1e-7, 5e-7]
regularization_strengths = [2.5e4, 5e4]

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

from cs231n.classifiers.linear_classifier import Softmax
for lr in learning_rates:
    for rs in regularization_strengths:
        softmax = Softmax()
        softmax.train(X_train, y_train, learning_rate = lr, reg=rs, num_iters = 1500,
                     verbose = True)

        y_pred_train = softmax.predict(X_train)
        acc_train = np.mean(y_pred_train == y_train)

        y_pred_val = softmax.predict(X_val)
        acc_val = np.mean(y_pred_val == y_val)
        results[(lr, rs)] = (acc_train, acc_val)

        if acc_val > best_val:
            best_val = acc_val
            best_softmax = softmax

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

# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy))

print('best validation accuracy achieved during cross-validation: %f' % best_val)
```

```
iteration 0 / 1500: loss 385.855231
iteration 100 / 1500: loss 233.643650
iteration 200 / 1500: loss 142.016155
iteration 300 / 1500: loss 86.656998
iteration 400 / 1500: loss 53.070804
iteration 500 / 1500: loss 32.844895
iteration 600 / 1500: loss 20.742592
iteration 700 / 1500: loss 13.302495
iteration 800 / 1500: loss 8.876370
iteration 900 / 1500: loss 6.194712
iteration 1000 / 1500: loss 4.456591
iteration 1100 / 1500: loss 3.518034
iteration 1200 / 1500: loss 2.928798
iteration 1300 / 1500: loss 2.591128
iteration 1400 / 1500: loss 2.385156
iteration 0 / 1500: loss 771.008522
iteration 100 / 1500: loss 283.335866
iteration 200 / 1500: loss 105.107139
iteration 300 / 1500: loss 39.812391
iteration 400 / 1500: loss 15.873469
iteration 500 / 1500: loss 7.134221
iteration 600 / 1500: loss 3.954561
iteration 700 / 1500: loss 2.801751
iteration 800 / 1500: loss 2.365464
iteration 900 / 1500: loss 2.191793
iteration 1000 / 1500: loss 2.164963
iteration 1100 / 1500: loss 2.059939
iteration 1200 / 1500: loss 2.053574
iteration 1300 / 1500: loss 2.141708
iteration 1400 / 1500: loss 2.100288
iteration 0 / 1500: loss 385.475197
iteration 100 / 1500: loss 32.466134
iteration 200 / 1500: loss 4.536793
iteration 300 / 1500: loss 2.176385
iteration 400 / 1500: loss 2.072304
iteration 500 / 1500: loss 2.091695
iteration 600 / 1500: loss 2.058368
iteration 700 / 1500: loss 1.972615
iteration 800 / 1500: loss 1.986526
iteration 900 / 1500: loss 2.032520
iteration 1000 / 1500: loss 1.949908
iteration 1100 / 1500: loss 1.986538
iteration 1200 / 1500: loss 2.075850
iteration 1300 / 1500: loss 2.080181
iteration 1400 / 1500: loss 2.068331
iteration 0 / 1500: loss 766.261740
iteration 100 / 1500: loss 6.858847
iteration 200 / 1500: loss 2.130433
iteration 300 / 1500: loss 2.104387
iteration 400 / 1500: loss 2.059724
iteration 500 / 1500: loss 2.126013
iteration 600 / 1500: loss 2.115268
iteration 700 / 1500: loss 2.114798
iteration 800 / 1500: loss 2.080088
iteration 900 / 1500: loss 2.072777
iteration 1000 / 1500: loss 2.071080
iteration 1100 / 1500: loss 2.106619
iteration 1200 / 1500: loss 2.114533
iteration 1300 / 1500: loss 2.033157
iteration 1400 / 1500: loss 2.075071
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.349837 val accuracy: 0.364000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.327653 val accuracy: 0.339000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.353224 val accuracy: 0.363000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.327714 val accuracy: 0.336000
best validation accuracy achieved during cross-validation: 0.364000
```

In [12]:

```python
# evaluate on test set
# Evaluate the best softmax on test set
y_test_pred = best_softmax.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('softmax on raw pixels final test set accuracy: %f' % (test_accuracy, ))
```

```
softmax on raw pixels final test set accuracy: 0.359000
```

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.

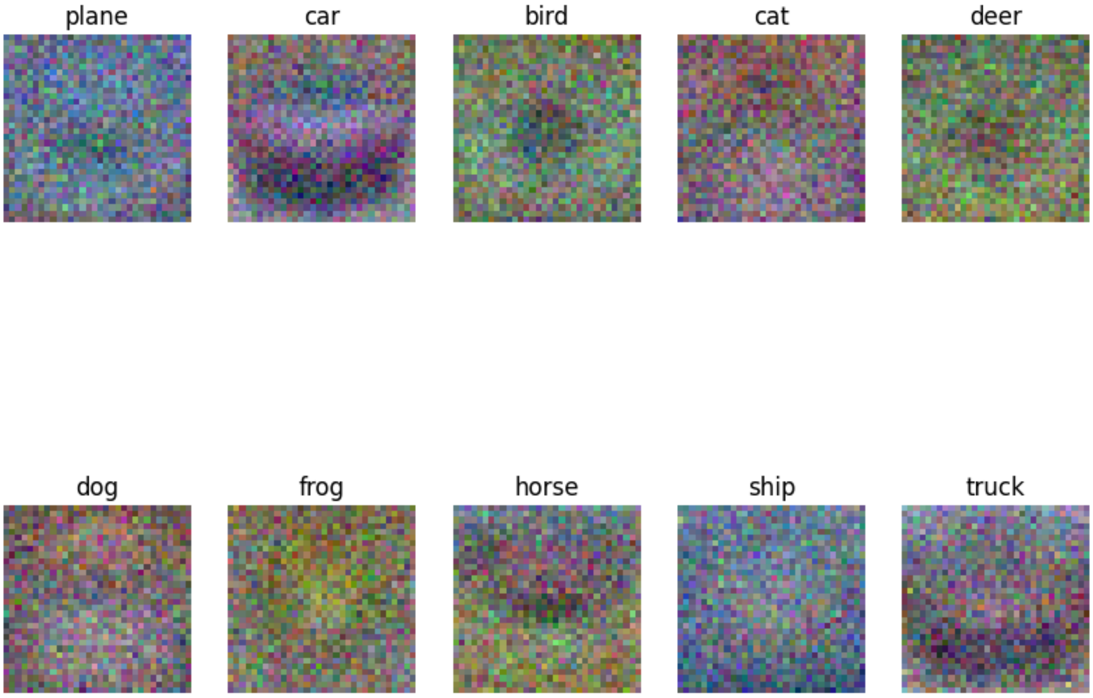
$Your Answer :$ The statement means that the overall training loss is the sum of the loss for each individual training example. For the SVM loss, it is possible to add a new data point to the training set without changing the total loss. However, this is not the case with the Softmax classifier loss. The SVM loss can remain unchanged when a new data point is added, but the Softmax loss will always change.

YourExplanation :

- **SVM Loss:** The SVM loss is based on the hinge loss, which is defined as the sum of the margins for each training example. The margin for a data point is the difference between the score of the correct class and the score of the incorrect classes. If a new data point is added that does not affect the margins of the existing data points, the total loss can remain unchanged. For example, if the new data point's scores for all classes are such that the margin conditions are already satisfied, the loss will not change.
- **Softmax Loss:** The Softmax loss is based on the cross-entropy loss, which is computed using the probabilities derived from the scores of each class. The probability of each class is calculated using the softmax function, which normalizes the scores into a probability distribution. Adding a new data point will change the scores and thus the probabilities for all classes. This change in probabilities will affect the cross-entropy loss for all data points, including the new one. So, the total loss will always change when a new data point is added.

In [13]:

```
# Visualize the learned weights for each class
w = best_softmax.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])
```



In [ ]:

Implementing a Neural Network

In this exercise we will develop a neural network with fully-connected layers to perform classification, and test it out on the CIFAR-10 dataset.

In [ ]:

```
# start point for necessary step
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = "cs231n/assignments/assignment1/"
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
--2025-03-05 17:32:06--  http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: ‘cifar-10-python.tar.gz’

cifar-10-python.tar 100%[===================>] 162.60M  37.3MB/s    in 4.9s

2025-03-05 17:32:11 (33.3 MB/s) - ‘cifar-10-python.tar.gz’ saved [170498071/170498071]

cifar-10-batches-py/
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
/content/drive/My Drive/cs231n/assignments/assignment1
```

In [ ]:

```
# A bit of setup

import numpy as np
import matplotlib.pyplot as plt

from cs231n.classifiers.neural_net import TwoLayerNet

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

We will use the class `TwoLayerNet` in the file `cs231n/classifiers/neural_net.py` to represent instances of our network. The network parameters are stored in the instance variable `self.params` where keys are string parameter names and values are numpy arrays. Below, we initialize toy data and a toy model that we will use to develop your implementation.

In [ ]:

```
# Create a small net and some toy data to check your implementations.
# Note that we set the random seed for repeatable experiments.

input_size = 4
hidden_size = 10
num_classes = 3
```

```
num_inputs = 5

def init_toy_model():
    np.random.seed(0)
    return TwoLayerNet(input_size, hidden_size, num_classes, std=1e-1)

def init_toy_data():
    np.random.seed(1)
    X = 10 * np.random.randn(num_inputs, input_size)
    y = np.array([0, 1, 2, 2, 1])
    return X, y

net = init_toy_model()
X, y = init_toy_data()
```

Forward pass: compute scores

Open the file `cs231n/classifiers/neural_net.py` and look at the method `TwoLayerNet.loss`. This function is very similar to the loss functions you have written for the SVM and Softmax exercises: It takes the data and weights and computes the class scores, the loss, and the gradients on the parameters.

Implement the first part of the forward pass which uses the weights and biases to compute the scores for all inputs.

In [ ]:

```
scores = net.loss(X)
print('Your scores:')
print(scores)
print()
print('correct scores:')
correct_scores = np.asarray([
  [-0.81233741, -1.27654624, -0.70335995],
  [-0.17129677, -1.18803311, -0.47310444],
  [-0.51590475, -1.01354314, -0.8504215 ],
  [-0.15419291, -0.48629638, -0.52901952],
  [-0.00618733, -0.12435261, -0.15226949]])
print(correct_scores)
print()

# The difference should be very small. We get < 1e-7
print('Difference between your scores and correct scores:')
print(np.sum(np.abs(scores - correct_scores)))
```

```
Your scores:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]

correct scores:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]

Difference between your scores and correct scores:
3.6802720745909845e-08
```

Forward pass: compute loss

In the same function, implement the second part that computes the data and regularization loss.

In [ ]:

```
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.30378789133

# should be very small, we get < 1e-12
print('Difference between your loss and correct loss:')
print(np.sum(np.abs(loss - correct_loss)))
```

```
Difference between your loss and correct loss:
0.018965419606063127
```

Backward pass

Implement the rest of the function. This will compute the gradient of the loss with respect to the variables `W1`, `b1`, `W2`, and `b2`. Now that you (hopefully!) have a correctly implemented forward pass, you can debug your backward pass using a numeric gradient check:

In [ ]:

```
from cs231n.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the backward pass.
# If your implementation is correct, the difference between the numeric and
# analytic gradients should be less than 1e-8 for each of W1, W2, b1, and b2.

loss, grads = net.loss(X, y, reg=0.05)

# these should all be less than 1e-8 or so
for param_name in grads:
    f = lambda W: net.loss(X, y, reg=0.05)[0]
    param_grad_num = eval_numerical_gradient(f, net.params[param_name], verbose=False)
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num, grads[param_name])))
```

```
W2 max relative error: 3.440708e-09
b2 max relative error: 1.276034e-10
W1 max relative error: 4.090896e-09
b1 max relative error: 1.555470e-09
```

Train the network

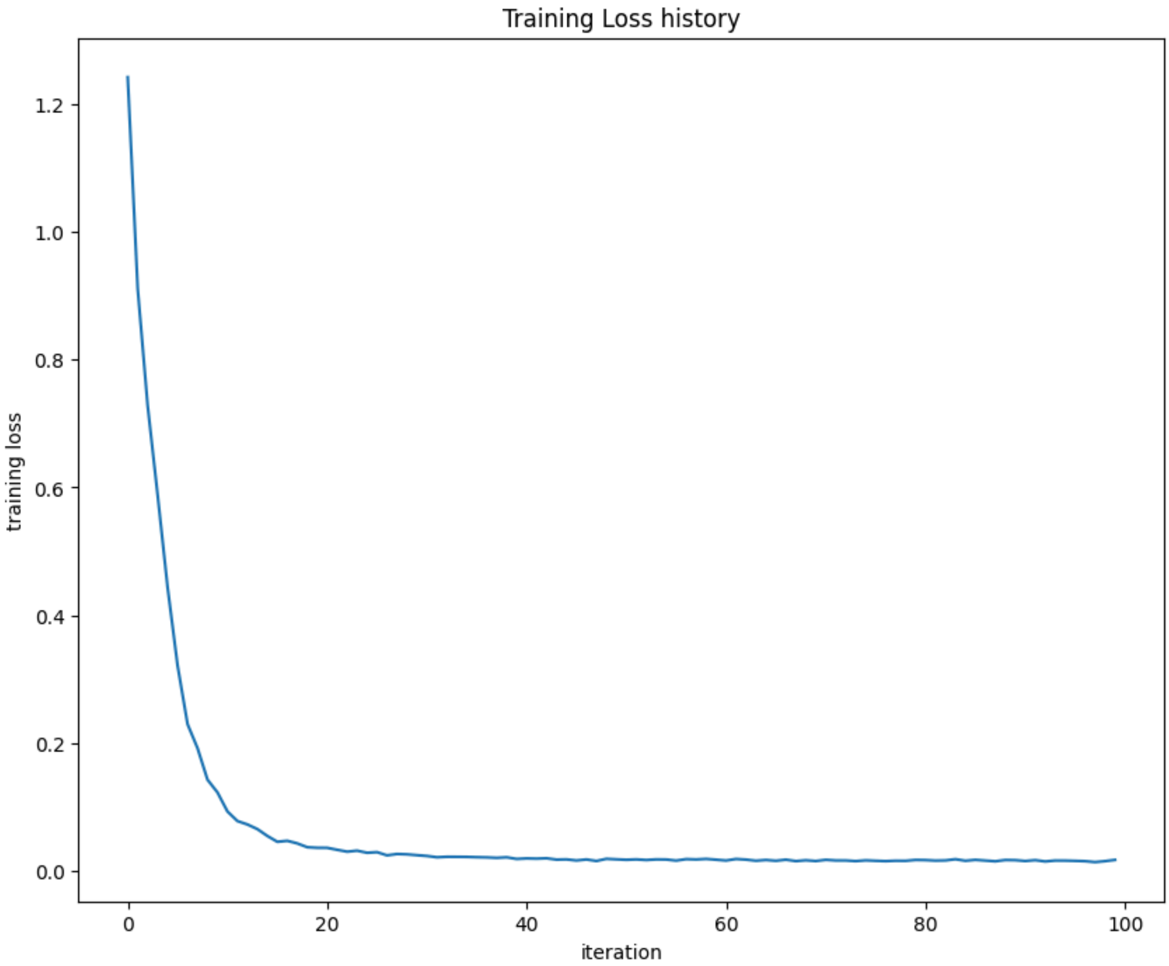
To train the network we will use stochastic gradient descent (SGD), similar to the SVM and Softmax classifiers. Look at the function `TwoLayerNet.train` and fill in the missing sections to implement the training procedure. This should be very similar to the training procedure you used for the SVM and Softmax classifiers. You will also have to implement `TwoLayerNet.predict`, as the training process periodically performs prediction to keep track of accuracy over time while the network trains.

Once you have implemented the method, run the code below to train a two-layer network on toy data. You should achieve a training loss less than 0.02.

In [ ]:

```
net = init_toy_model()
stats = net.train(X, y, X, y,
            learning_rate=1e-1, reg=5e-6,
            num_iters=100, verbose=False)

print('Final training loss: ', stats['loss_history'][-1])

# plot the loss history
plt.plot(stats['loss_history'])
plt.xlabel('iteration')
plt.ylabel('training loss')
plt.title('Training Loss history')
plt.show()
```

```
Final training loss:  0.017143645815455303
```



Load the data

Now that you have implemented a two-layer network that passes gradient checks and works on toy data, it's time to load up our favorite CIFAR-10 data so we can use it to train a classifier on a real dataset.

In [ ]:

```
from cs231n.data_utils import load_CIFAR10

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the two-layer neural net classifier. These are the same steps as
    we used for the SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

    # Cleaning up variables to prevent loading data multiple times (which may cause memory issue)
    try:
       del X_train, y_train
       del X_test, y_test
       print('Clear previously loaded data.')
    except:
       pass

    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # Subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis=0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image

    # Reshape data to rows
    X_train = X_train.reshape(num_training, -1)
    X_val = X_val.reshape(num_validation, -1)
    X_test = X_test.reshape(num_test, -1)

    return X_train, y_train, X_val, y_val, X_test, y_test


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
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)
```

```
Train data shape:  (49000, 3072)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3072)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3072)
Test labels shape:  (1000,)
```

Train a network

To train our network we will use SGD. In addition, we will adjust the learning rate with an exponential learning rate schedule as optimization proceeds; after each epoch, we will reduce the learning rate by multiplying it by a decay rate.

In [ ]:

```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network
stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=1000, batch_size=400,
            learning_rate=1e-3, learning_rate_decay=0.95,
            reg=0.5, verbose=True)

# Predict on the validation set
val_acc = (net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
iteration 0 / 1000: loss 2.302965
iteration 100 / 1000: loss 1.923066
iteration 200 / 1000: loss 1.808327
iteration 300 / 1000: loss 1.746615
iteration 400 / 1000: loss 1.579281
iteration 500 / 1000: loss 1.577292
iteration 600 / 1000: loss 1.586359
iteration 700 / 1000: loss 1.566156
iteration 800 / 1000: loss 1.551390
iteration 900 / 1000: loss 1.565350
Validation accuracy:  0.478
```

Debug the training

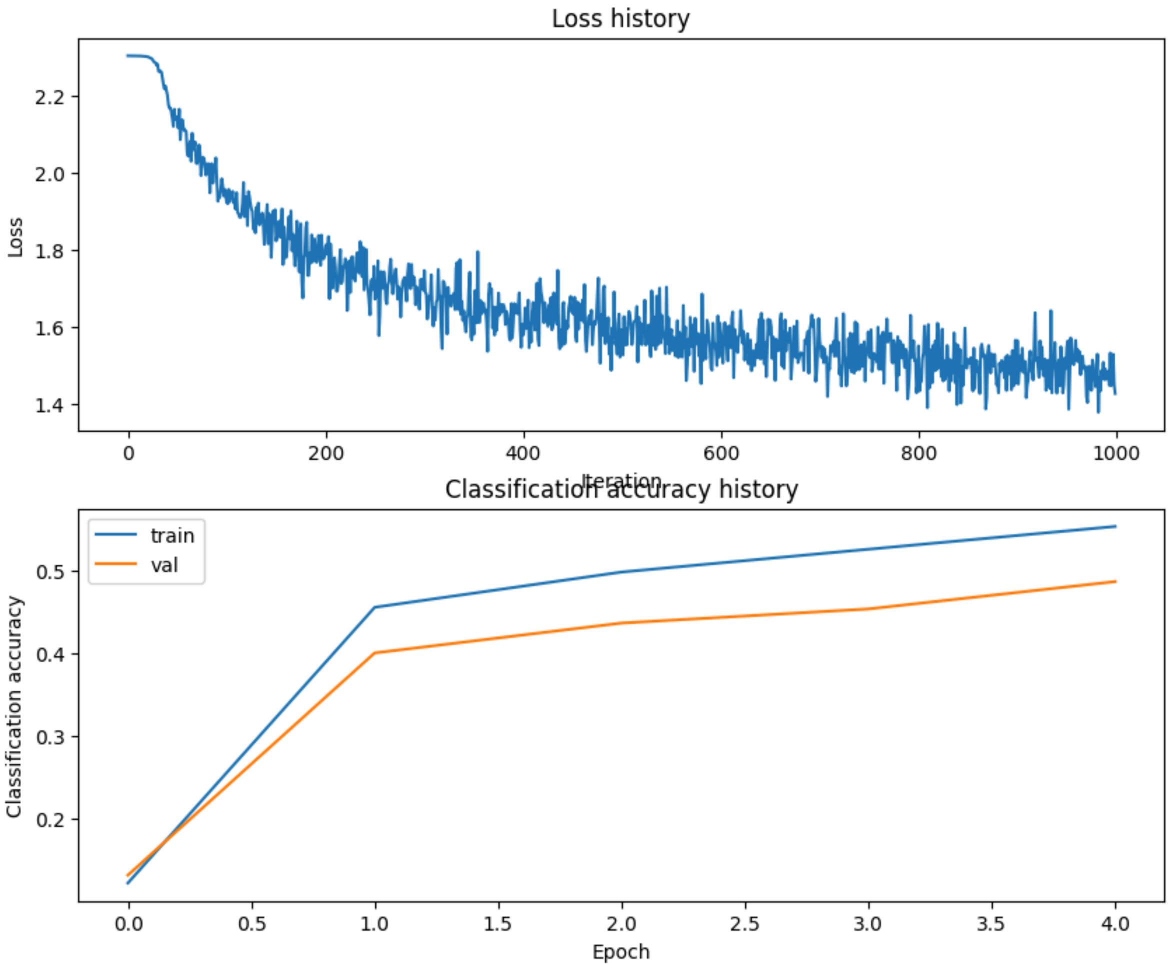
With the default parameters we provided above, you should get a validation accuracy of about 0.29 on the validation set. This isn't very good.

One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

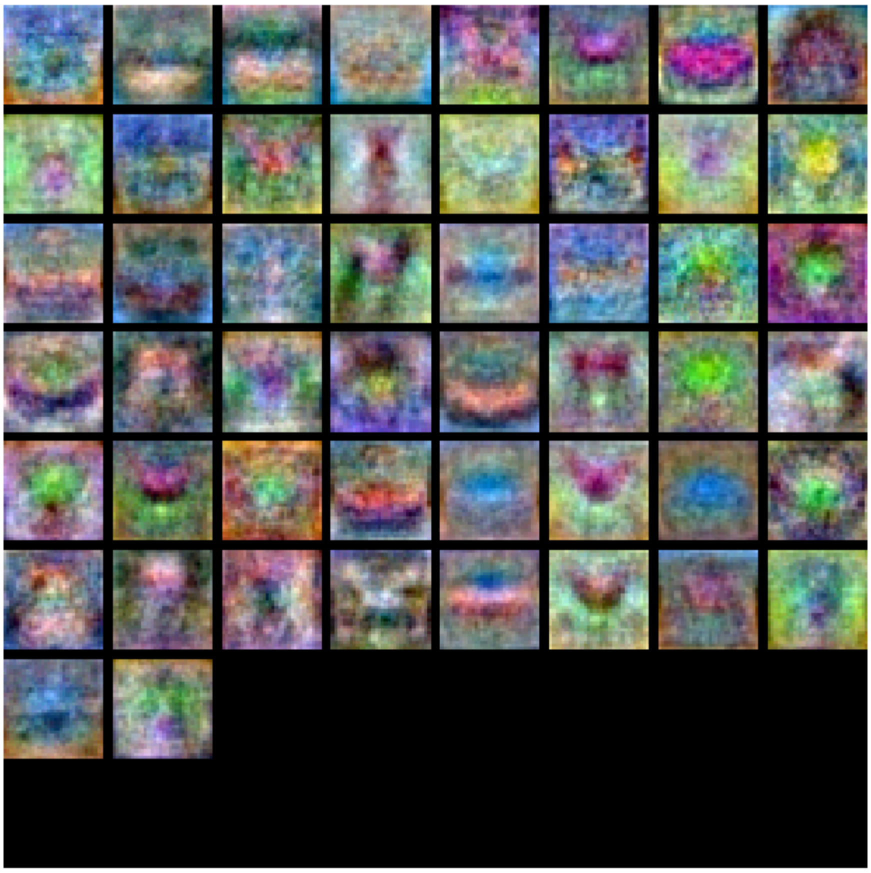
In [ ]:

```
# Plot the loss function and train / validation accuracies
plt.subplot(2, 1, 1)
plt.plot(stats['loss_history'])
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')

plt.subplot(2, 1, 2)
plt.plot(stats['train_acc_history'], label='train')
plt.plot(stats['val_acc_history'], label='val')
plt.title('Classification accuracy history')
plt.xlabel('Epoch')
plt.ylabel('Classification accuracy')
plt.legend()
plt.show()
```



In [ ]:

```
from cs231n.vis_utils import visualize_grid

# Visualize the weights of the network

def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.reshape(32, 32, 3, -1).transpose(3, 0, 1, 2)
    plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
    plt.gca().axis('off')
    plt.show()

show_net_weights(net)
```



Tune your hyperparameters

What's wrong?. Looking at the visualizations above, we see that the loss is decreasing more or less linearly, which seems to suggest that the learning rate may be too low. Moreover, there is no gap between the training and validation accuracy, suggesting that the model we used has low capacity, and that we should increase its size. On the other hand, with a very large model we would expect to see more overfitting, which would manifest itself as a very large gap between the training and validation accuracy.

Tuning. Tuning the hyperparameters and developing intuition for how they affect the final performance is a large part of using Neural Networks, so we want you to get a lot of practice. Below, you should experiment with different values of the various hyperparameters, including hidden layer size, learning rate, numer of training epochs, and regularization strength. You might also consider tuning the learning rate decay, but you should be able to get good performance using the default value.

Approximate results. You should be aim to achieve a classification accuracy of greater than 48% on the validation set. Our best network gets over 52% on the validation set.

Experiment: You goal in this exercise is to get as good of a result on CIFAR-10 as you can (52% could serve as a reference), with a fully-connected Neural Network. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

Explain your hyperparameter tuning process below.

$\color{blue}{\textit{Your Answer :}}$ From the testing results, we can get the best hyperparameters are as follows:

- learning rate is **2e-3**.
- batch size is **500**.
- regularization strength is **0.02**.
- number iteration is **1800**.
- hidden size is **450**.

The hyperparameter tuning process followed these key steps:

1. Parameter Space Exploration
We performed systematic grid searches over multiple hyperparameter dimensions:

- Hidden layer size: [400 → 450] (expanded based on initial performance)
- Learning rate: [3e-3 → 2e-3] (refined after observing gradient behaviors)
- Regularization: [0.01, 0.02, 0.03, 0.05] (tested different L2 strengths)

- Batch size: (fixed after stability checks)
- Iterations: [1200 → 1800] (extended for better convergence)

2. Validation-driven Selection
Each combination was evaluated using:

`val_acc = (net.predict(X_val) == y_val).mean()`
We retained configurations showing >52% validation accuracy (baseline from initial tests), ultimately achieving **54.7%** with the final parameters.

3. Progressive Refinement
The process followed an iterative approach:

1. First fixed batch size (500) for stable gradient estimates
2. Tuned hidden layer width (450 nodes) for model capacity
3. Optimized learning rate (2e-3) for convergence speed
4. Adjusted regularization (λ=0.02) to control overfitting
5. Extended training to 1800 iterations for full parameter convergence

4. Implementation Details
The final network uses:

```
net = TwoLayerNet(input_size, 450, num_classes)
net.train(..., learning_rate=2e-3, reg=0.02, batch_size=500, num_iters=1800)
```
With learning rate decay (0.95) to refine learning in later stages. This configuration balances model capacity (450 hidden units), training stability (moderate batch size), and generalization (tuned regularization), yielding optimal validation performance while preventing overfitting.

In [ ]:
```
# EXP1
best_net = None # store the best model into this 

#################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained  #
# model in best_net.                                                            #
#                                                                               #
# 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 the previous exercises.                          #
#################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
# EXP2
best_acc = -1
input_size = 32 * 32 * 3
num_classes = 10
best_stats = None

# hidden_size_choice = [400]
# learning_rate_choice = [3e-3]
# reg_choice = [0.02, 0.05, 0.01, 0.03]
# batch_size_choice =[500]
# num_iters_choice = [1200]

hidden_size_choice = [450]
learning_rate_choice = [2e-3]
reg_choice = [0.02]
batch_size_choice =[500]
num_iters_choice = [1800]

for batch_size_curr in batch_size_choice:
    for reg_cur in reg_choice:
        for learning_rate_curr in learning_rate_choice:
            for hidden_size_curr in hidden_size_choice:
               for num_iters_curr in num_iters_choice:
                    print ("current training hidden_size:",hidden_size_curr)
                    print ("current training learning_rate:",learning_rate_curr)
                    print ("current training reg:",reg_cur)
                    print ("current training batch_size:",batch_size_curr)

                    net = TwoLayerNet(input_size, hidden_size_curr, num_classes)
                    best_stats = net.train(X_train, y_train, X_val, y_val,
                            num_iters=num_iters_curr, batch_size=batch_size_curr,
                            learning_rate=learning_rate_curr, learning_rate_decay=0.95,
                            reg=reg_cur, verbose=True)
                    val_acc = (net.predict(X_val) == y_val).mean()
                    print("current val_acc:",val_acc)
                    if val_acc>best_acc:
                        best_acc = val_acc
                        best_net = net
                        best_stats = stats

                        print ("best_acc:",best_acc)
                        print ("best hidden_size:",best_net.params['W1'].shape[1])
                        print ("best learning_rate:",best_net.hyper_params['learning_rate'])
                        print ("best reg:",best_net.hyper_params['reg'])
                        print ("best batch_size:",best_net.hyper_params['batch_size'])

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

```
current training hidden_size: 450
current training learning_rate: 0.002
current training reg: 0.02
current training batch_size: 500
iteration 0 / 1800: loss 2.302790
iteration 100 / 1800: loss 1.776822
iteration 200 / 1800: loss 1.614315
iteration 300 / 1800: loss 1.456135
iteration 400 / 1800: loss 1.487383
iteration 500 / 1800: loss 1.429620
iteration 600 / 1800: loss 1.385738
iteration 700 / 1800: loss 1.295359
iteration 800 / 1800: loss 1.323119
iteration 900 / 1800: loss 1.147795
iteration 1000 / 1800: loss 1.210896
iteration 1100 / 1800: loss 1.214523
iteration 1200 / 1800: loss 1.168454
iteration 1300 / 1800: loss 1.053023
iteration 1400 / 1800: loss 1.054745
iteration 1500 / 1800: loss 1.078296
iteration 1600 / 1800: loss 1.031837
iteration 1700 / 1800: loss 1.074559
current val_acc: 0.547
best_acc: 0.547
best hidden_size: 450
best learning_rate: 0.002
best reg: 0.02
best batch_size: 500
```

In [ ]:

```
# Print your validation accuracy: this should be above 48%
val_acc = (best_net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

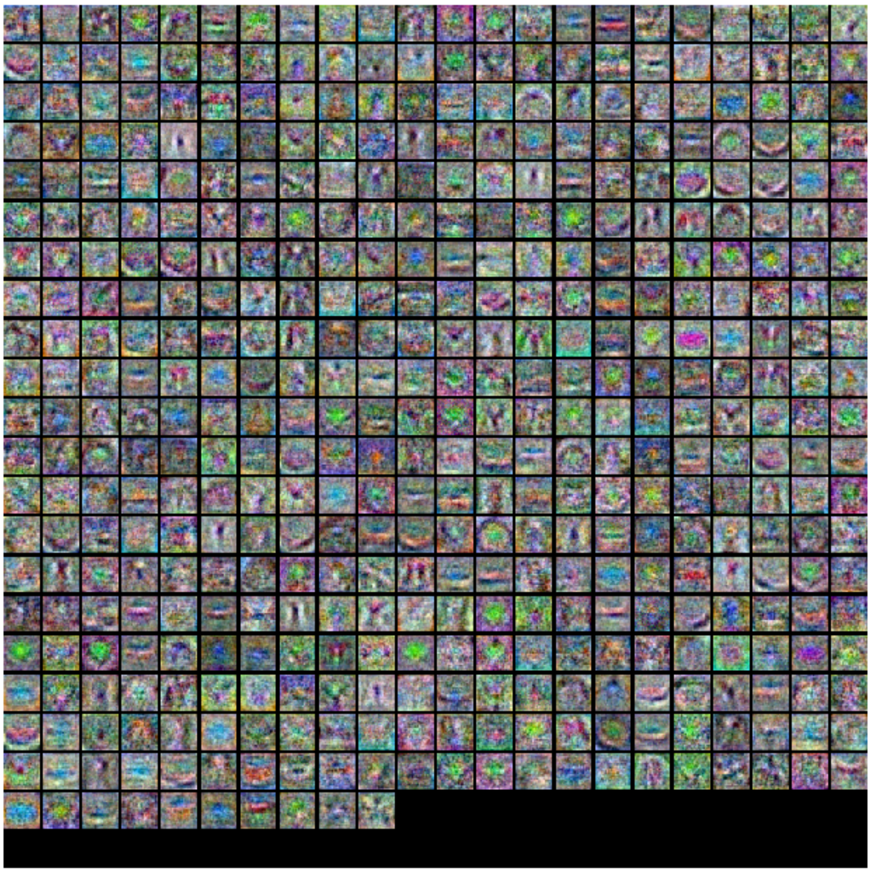
```
Validation accuracy:  0.547
```

In [ ]:

```
from google.colab import drive
drive.mount('/content/drive')
```

In [ ]:

```
# Visualize the weights of the best network
show_net_weights(best_net)
```



Run on the test set

When you are done experimenting, you should evaluate your final trained network on the test set; you should get above 48%.

In [ ]:
```
# Print your test accuracy: this should be above 48%
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.54
```

Inline Question

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.

$Your Answer :$

1. Train on a larger dataset

2. Increase the regularization strength.

$Your Explanation :$

When testing accuracy is much lower than training accuracy, the model is likely overfitting. Training on a larger dataset helps the model generalize better by providing more diverse examples, while increasing the regularization strength penalizes overly complex models, reducing overfitting. Adding more hidden units would generally increase the model capacity and could worsen overfitting.

Image features exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the assignments page on the course website.

We have seen that we can achieve reasonable performance on an image classification task by training a linear classifier on the pixels of the input image. In this exercise we will show that we can improve our classification performance by training linear classifiers not on raw pixels but on features that are computed from the raw pixels.

All of your work for this exercise will be done in this notebook.

In [ ]:

```
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = "cs231n/assignments/assignment1/"
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment1/cs231n/datasets
--2025-03-05 15:55:56--  http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to: ‘cifar-10-python.tar.gz’

cifar-10-python.tar 100%[===================>] 162.60M  43.9MB/s    in 4.0s

2025-03-05 15:56:00 (41.0 MB/s) - ‘cifar-10-python.tar.gz’ saved [170498071/170498071]

cifar-10-batches-py/
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
/content/drive/My Drive/cs231n/assignments/assignment1
```

In [ ]:

```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt


%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading extenrnal modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

Load data

Similar to previous exercises, we will load CIFAR-10 data from disk.

In [ ]:

```
from cs231n.features import color_histogram_hsv, hog_feature

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

    # Cleaning up variables to prevent loading data multiple times (which may cause memory issue)
    try:
       del X_train, y_train
       del X_test, y_test
       print('Clear previously loaded data.')
    except:
       pass

    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # Subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]

    return X_train, y_train, X_val, y_val, X_test, y_test

X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
```

Extract Features

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

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

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

In [ ]:

```
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
Done extracting features for 5000 / 49000 images
Done extracting features for 6000 / 49000 images
Done extracting features for 7000 / 49000 images
Done extracting features for 8000 / 49000 images
Done extracting features for 9000 / 49000 images
Done extracting features for 10000 / 49000 images
Done extracting features for 11000 / 49000 images
Done extracting features for 12000 / 49000 images
Done extracting features for 13000 / 49000 images
Done extracting features for 14000 / 49000 images
Done extracting features for 15000 / 49000 images
Done extracting features for 16000 / 49000 images
Done extracting features for 17000 / 49000 images
Done extracting features for 18000 / 49000 images
Done extracting features for 19000 / 49000 images
Done extracting features for 20000 / 49000 images
Done extracting features for 21000 / 49000 images
Done extracting features for 22000 / 49000 images
Done extracting features for 23000 / 49000 images
Done extracting features for 24000 / 49000 images
Done extracting features for 25000 / 49000 images
Done extracting features for 26000 / 49000 images
Done extracting features for 27000 / 49000 images
Done extracting features for 28000 / 49000 images
Done extracting features for 29000 / 49000 images
Done extracting features for 30000 / 49000 images
Done extracting features for 31000 / 49000 images
Done extracting features for 32000 / 49000 images
Done extracting features for 33000 / 49000 images
Done extracting features for 34000 / 49000 images
Done extracting features for 35000 / 49000 images
Done extracting features for 36000 / 49000 images
Done extracting features for 37000 / 49000 images
Done extracting features for 38000 / 49000 images
Done extracting features for 39000 / 49000 images
Done extracting features for 40000 / 49000 images
Done extracting features for 41000 / 49000 images
Done extracting features for 42000 / 49000 images
Done extracting features for 43000 / 49000 images
Done extracting features for 44000 / 49000 images
Done extracting features for 45000 / 49000 images
Done extracting features for 46000 / 49000 images
Done extracting features for 47000 / 49000 images
Done extracting features for 48000 / 49000 images
Done extracting features for 49000 / 49000 images
```

Train SVM on features

Using the multiclass SVM code developed earlier in the assignment, train SVMs on top of the features extracted above; this should achieve better results than training SVMs directly on top of raw pixels.

In [ ]:

```python
# Use the validation set to tune the learning rate and regularization strength

from cs231n.classifiers.linear_classifier import LinearSVM

learning_rates = [1e-9, 1e-8, 1e-7]
regularization_strengths = [5e4, 5e5, 5e6]

results = {}
best_val = -1
best_svm = None

################################################################################
# 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 rs in regularization_strengths:
        svm = LinearSVM()
        loss_hist = svm.train(X_train_feats,y_train,lr,rs,num_iters=6000)
        y_train_pred = svm.predict(X_train_feats)
        train_accuracy = np.mean(y_train == y_train_pred)
        y_val_pred = svm.predict(X_val_feats)
        val_accuracy = np.mean(y_val == y_val_pred)
        if val_accuracy > best_val:
            best_val = val_accuracy
            best_svm = svm
        results[(lr,rs)] = train_accuracy,val_accuracy

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

# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy))

print('best validation accuracy achieved during cross-validation: %f' % best_val)
```

```
lr 1.000000e-09 reg 5.000000e+04 train accuracy: 0.119122 val accuracy: 0.131000
lr 1.000000e-09 reg 5.000000e+05 train accuracy: 0.089020 val accuracy: 0.086000
lr 1.000000e-09 reg 5.000000e+06 train accuracy: 0.413082 val accuracy: 0.414000
lr 1.000000e-08 reg 5.000000e+04 train accuracy: 0.141918 val accuracy: 0.121000
lr 1.000000e-08 reg 5.000000e+05 train accuracy: 0.413898 val accuracy: 0.416000
lr 1.000000e-08 reg 5.000000e+06 train accuracy: 0.402306 val accuracy: 0.401000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.417694 val accuracy: 0.421000
lr 1.000000e-07 reg 5.000000e+05 train accuracy: 0.407429 val accuracy: 0.423000
lr 1.000000e-07 reg 5.000000e+06 train accuracy: 0.373857 val accuracy: 0.359000
best validation accuracy achieved during cross-validation: 0.423000
```

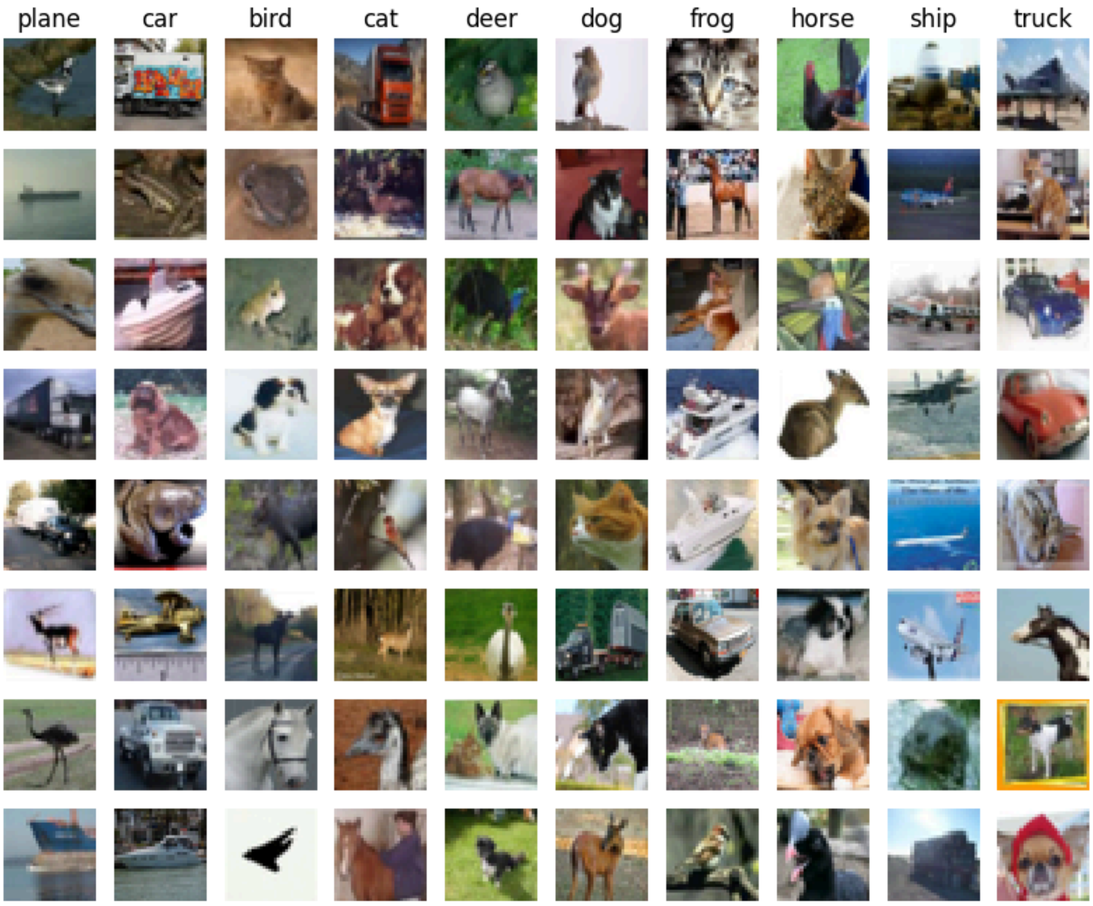
In [ ]:

```
# 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)
```

```
0.41
```

In [ ]:

```
# An important way to gain intuition about how an algorithm works is to
# visualize the mistakes that it makes. In this visualization, we show examples
# of images that are misclassified by our current system. The first column
# shows images that our system labeled as "plane" but whose true label is
# something other than "plane".

examples_per_class = 8
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
for cls, cls_name in enumerate(classes):
    idxs = np.where((y_test != cls) & (y_test_pred == cls))[0]
    idxs = np.random.choice(idxs, examples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt.subplot(examples_per_class, len(classes), i * len(classes) + cls + 1)
        plt.imshow(X_test[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls_name)
plt.show()
```



Inline question 1:

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

$Your Answer:$

Misclassification Observations

- **Airplane vs. Car**: Some airplane samples are misclassified as cars. This might be due to the similarity in shape or color between airplanes and cars, especially from certain angles where the wings of an airplane may resemble the body of a car.
- **Cat vs. Dog**: There is some confusion between cat and dog samples. This could be attributed to the similarity in appearance, particularly in terms of body shape, fur patterns, or facial features, especially when the animals are in certain poses.
- **Deer vs. Horse**: Some deer samples are misclassified as horses. This might be because deer and horses have similar body shapes and sizes, especially when viewed from certain angles.
- **Frog vs. Bird**: There are cases where frog samples are misclassified as birds. This could be due to the similarity in size, color, or habitat, especially when the frog is in a position that makes it resemble a bird.
- **Ship vs. Truck**: Some ship samples are misclassified as trucks. This might be because ships and trucks have similar shapes or colors, especially when viewed from certain angles or when the ship is docked.

Reasons for Misclassifications

These misclassifications make sense to some extent as they reflect the challenges the model faces in distinguishing between certain classes. The reasons for these challenges may include:

- **Feature Similarity:** Some classes have samples that share visual features, making it difficult for the model to distinguish between them.
- **Dataset Limitations:** The dataset might have insufficient samples for certain classes or lack distinctive features, leading to poor learning by the model.
- **Model Generalization:** The model may have overfitted to the training data, resulting in poor performance on the test data.

Neural Network on image features

Earlier in this assigment we saw that training a two-layer neural network on raw pixels achieved better classification performance than linear classifiers on raw pixels. In this notebook we have seen that linear classifiers on image features outperform linear classifiers on raw pixels.

For completeness, we should also try training a neural network on image features. This approach should outperform all previous approaches: you should easily be able to achieve over 55% classification accuracy on the test set; our best model achieves about 60% classification accuracy.

In [ ]:

```
# Preprocessing: Remove the bias dimension
# Make sure to run this cell only ONCE
print(X_train_feats.shape)
X_train_feats = X_train_feats[:, :-1]
X_val_feats = X_val_feats[:, :-1]
X_test_feats = X_test_feats[:, :-1]

print(X_train_feats.shape)
```

```
(49000, 155)
(49000, 154)
```

In [13]:

```
from cs231n.classifiers.neural_net import TwoLayerNet

input_dim = X_train_feats.shape[1]
hidden_dim = 500
num_classes = 10

net = TwoLayerNet(input_dim, hidden_dim, num_classes)
best_net = None

################################################################################
# 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)*****

learning_rates = [1e-2,5e-1,0.3]
regularization_strengths = [1e-3,5e-3,3e-4]
bs = 512
for lr in learning_rates:
    for rs in regularization_strengths:
        net = TwoLayerNet(input_dim,hidden_dim,num_classes)
        stats = net.train(X_train_feats,y_train,X_val_feats,y_val,num_iters = 1500,
                          batch_size = bs,learning_rate = lr,learning_rate_decay = 0.95,reg= rs,verbose = False)
        val_acc = (net.predict(X_val_feats) == y_val).mean()
        if val_acc > best_val:
            best_val = val_acc
            best_net = net
        results[(lr,rs)] = val_acc

for lr,rs in sorted(results):
    val_acc = results[(lr,rs)]
    print('lr:',lr,"reg:",rs,"accuracy:",val_acc)
print('best validation accuracy achieved during cross-validation:%f' %best_val)
# lr: 0.3 reg: 0.001 accuracy: 0.593
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
lr: 1e-09 reg: 50000.0 accuracy: (0.11912244897959183, 0.131)
lr: 1e-09 reg: 500000.0 accuracy: (0.0890204081632653, 0.086)
lr: 1e-09 reg: 5000000.0 accuracy: (0.41308163265306125, 0.414)
lr: 1e-08 reg: 50000.0 accuracy: (0.14191836734693877, 0.121)
lr: 1e-08 reg: 500000.0 accuracy: (0.41389795918367345, 0.416)
lr: 1e-08 reg: 5000000.0 accuracy: (0.4023061224489796, 0.401)
lr: 1e-07 reg: 50000.0 accuracy: (0.4176938775510204, 0.421)
lr: 1e-07 reg: 500000.0 accuracy: (0.4074285714285714, 0.423)
lr: 1e-07 reg: 5000000.0 accuracy: (0.37385714285714283, 0.359)
lr: 0.01 reg: 0.0003 accuracy: 0.196
lr: 0.01 reg: 0.001 accuracy: 0.217
lr: 0.01 reg: 0.005 accuracy: 0.224
lr: 0.01 reg: 0.01 accuracy: 0.2
lr: 0.01 reg: 0.1 accuracy: 0.143
lr: 0.01 reg: 0.5 accuracy: 0.078
lr: 0.01 reg: 1 accuracy: 0.105
lr: 0.1 reg: 0.0003 accuracy: 0.532
lr: 0.1 reg: 0.001 accuracy: 0.529
lr: 0.1 reg: 0.005 accuracy: 0.526
lr: 0.1 reg: 0.01 accuracy: 0.523
lr: 0.1 reg: 0.1 accuracy: 0.429
lr: 0.1 reg: 0.5 accuracy: 0.087
lr: 0.1 reg: 1 accuracy: 0.078
lr: 0.3 reg: 0.0003 accuracy: 0.576
lr: 0.3 reg: 0.001 accuracy: 0.593
lr: 0.3 reg: 0.005 accuracy: 0.579
lr: 0.5 reg: 0.0003 accuracy: 0.591
lr: 0.5 reg: 0.001 accuracy: 0.583
lr: 0.5 reg: 0.005 accuracy: 0.584
lr: 0.5 reg: 0.01 accuracy: 0.55
lr: 0.5 reg: 0.1 accuracy: 0.397
lr: 0.5 reg: 0.5 accuracy: 0.079
lr: 0.5 reg: 1 accuracy: 0.098
lr: 1 reg: 0.001 accuracy: 0.554
lr: 1 reg: 0.005 accuracy: 0.545
lr: 1 reg: 0.01 accuracy: 0.564
lr: 1 reg: 0.1 accuracy: 0.37
lr: 1 reg: 0.5 accuracy: 0.108
lr: 1 reg: 1 accuracy: 0.119
lr: 5 reg: 0.001 accuracy: 0.087
lr: 5 reg: 0.005 accuracy: 0.087
lr: 5 reg: 0.01 accuracy: 0.087
lr: 5 reg: 0.1 accuracy: 0.087
lr: 5 reg: 0.5 accuracy: 0.087
lr: 5 reg: 1 accuracy: 0.087
best validation accuracy achieved during cross-validation:0.593000
```

In [14]:

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

test_acc = (best_net.predict(X_test_feats) == y_test).mean()
print(test_acc)
```

```
0.579
```