ECE C147/C247, Winter 2025 **Homework #2**
Neural Networks & Deep Learning Prof. J.C. Kao
UCLA ECE TAs: B. Qu, K. Pang, S. Dong, S. Rajesh, T. Monsoor, X. Yan

Due Monday, 27 Jan 2025, by 11:59pm to Gradescope.
100 points total.

1. (10 points) **Noisy linear regression**

   A real estate company have assigned us the task of building a model to predict the house prices in Westwood. For this task, the company has provided us with a dataset $\mathcal{D}$:

   $$\mathcal{D} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(N)}, y^{(N)})\}$$

   where $x^{(i)} \in \mathbb{R}^d$ is a feature vector of the i^{th} house and $y^{(i)} \in \mathbb{R}$ is the price of the i^{th} house. Since we just learned about linear regression, so we have decided to use a linear regression model for this task. Additionally, the IT manager of the real estate company have requested us to design a model with small weights. In order to accommodate his request, we will design a linear regression model with parameter regularization. In this problem, we will navigate through the process of achieving regularization by adding noise to the feature vectors. Recall, that we define the cost function in a linear regression problem as:

   $$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - {(x^{(i)})}^T\theta)^2$$

   where $\theta \in \mathbb{R}^d$ is the parameter vector. As mentioned earlier, we will be adding noise to the feature vectors in the dataset. Specifically, we will be adding zero-mean gaussian noise of known variance σ^2 from the distribution

   $$\mathcal{N}(0, \sigma^2 I)$$

   where $I \in \mathbb{R}^{d\times d}$ and $\sigma \in \mathbb{R}$. With the addition of gaussian noise the modified cost function is given by,

   $$\tilde{\mathcal{L}}(\theta) = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - (x^{(i)} + \delta^{(i)})^T\theta)^2$$

   where $\delta^{(i)}$ are i.i.d noise vectors with $\delta^{(i)} \sim \mathcal{N}(0, \sigma^2 I)$.

   (a) (6 points) Express the expectation of the modified loss over the gaussian noise, $\mathbb{E}_{\delta\sim\mathcal{N}}[\tilde{\mathcal{L}}(\theta)]$, in terms of the original loss plus a term independent of the data $\mathcal{D}$. To be precise, your answer should be of the form:

   $$\mathbb{E}_{\delta\sim\mathcal{N}}[\tilde{\mathcal{L}}(\theta)] = \mathcal{L}(\theta) + R$$

   where R is not a function of $\mathcal{D}$. For answering this part, you might find the following result useful:

   $$\mathbb{E}_{\delta\sim\mathcal{N}}[\delta\delta^T] = \sigma^2 I$$

(b) (2 points) Based on your answer to (a), under expectation what regularization effect would the addition of the noise have on the model?

(c) (1 point) Suppose $\sigma \longrightarrow 0$, what effect would this have on the model?

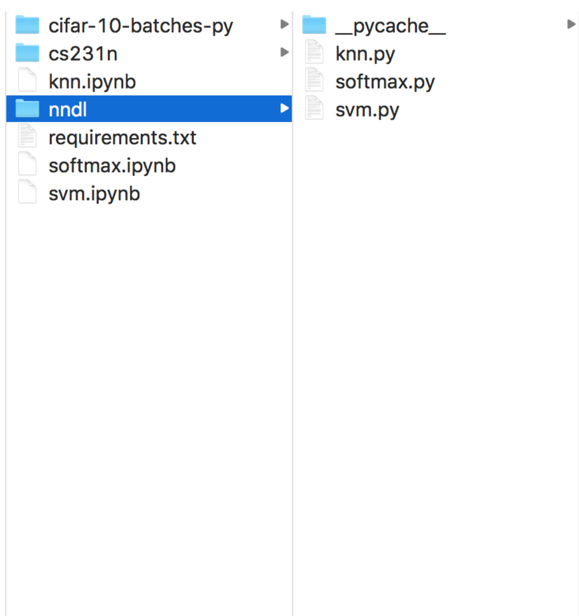
(d) (1 point) Suppose $\sigma \longrightarrow \infty$, what effect would this have on the model?

2. (20 points) ***k*-nearest neighbors.** Complete the k-nearest neighbors Jupyter notebook. The goal of this workbook is to give you experience with the CIFAR-10 dataset, training and evaluating a simple classifier, and k-fold cross validation. In the Jupyter notebook, we'll be using the CIFAR-10 dataset. Acquire this dataset by running:

```
wget http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
tar -xzvf cifar-10-python.tar.gz
rm cifar-10-python.tar.gz
```

If you don't have `wget` you can simply go to: `https://www.cs.toronto.edu/~kriz/cifar.html` and download it manually.

We have attached a screenshot of the paths the files ought to be in, in case helpful (though it should be apparent from the Jupyter notebook).



Print out the entire workbook and related code sections in knn.py, then submit them as a pdf to gradescope.

3. (30 points) **Softmax classifier gradient.** For softmax classifier, derive the gradient of the log likelihood.

Concretely, assume a classification problem with c classes

- Samples are $(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(m)}, y^{(m)})$, where $\mathbf{x}^{(j)} \in \mathbb{R}^n$, $y^{(j)} \in \{1, \ldots, c\}$, $j = 1, \ldots, m$

- Parameters are $\theta = \{\mathbf{w}_i, b_i\}_{i=1,\dots,c}$
- Probablistic model is

$$\Pr\left(y^{(j)} \mid \mathbf{x}^{(j)}, \theta\right) = \text{softmax}_{y^{(j)}}(\mathbf{x}^{(j)})$$

  where

$$\text{softmax}_{y^{(j)}}(\mathbf{x}^{(j)}) = \frac{e^{\mathbf{w}_{y^{(j)}}^T \mathbf{x}^{(j)} + b_{y^{(j)}}}}{\sum_{k=1}^{c} e^{\mathbf{w}_k^T \mathbf{x}^{(j)} + b_k}}$$

Derive the log-likelihood $\mathcal{L}$, and its gradient w.r.t. the parameters, $\nabla_{\mathbf{w}_i}\mathcal{L}$ and $\nabla_{b_i}\mathcal{L}$, for $i = 1, \dots, c$.

Note: We can group $\mathbf{w}_i$ and b_i into a single vector by augmenting the data vectors with an additional dimension of constant 1. Let $\tilde{\mathbf{x}} = \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}$, $\tilde{\mathbf{w}}_i = \begin{bmatrix} \mathbf{w}_i \\ b_i \end{bmatrix}$, then $a_i(\mathbf{x}) = \mathbf{w}_i^T \mathbf{x} + b_i = \tilde{\mathbf{w}}_i^T \tilde{\mathbf{x}}$. This unifies $\nabla_{\mathbf{w}_i}\mathcal{L}$ and $\nabla_{b_i}\mathcal{L}$ into $\nabla_{\tilde{\mathbf{w}}_i}\mathcal{L}$.

4. (10 points) **Hinge loss gradient.**

Due to the drastic changes in climate throughout the world, a weather forecasting organization wants our help to build a model that can classify the observed weather patterns as severe or not severe. They have accumulated data on various attributes of the weather pattern such as temperature, precipitation, humidity, wind speed, air pressure, and geographical location along with severity of weather. However, the contribution of the attributes to the classification of weather as severe or not is unknown.

We choose to use a binary support vector machine (SVM) classification model. The SVM model parameters are learned by optimizing a hinge loss. The company has provided us with a data-set

$$\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \cdots, (\mathbf{x}^{(K)}, y^{(K)})\}$$

where $\mathbf{x}^{(i)} \in \mathbb{R}^d$ is a feature vector of the i^{th} data sample and $y^{(i)} \in \{-1, 1\}$. We define the hinge loss per training sample as

$$\text{hinge}_{y^{(i)}}(\mathbf{x}^{(i)}) = \max\left(0, 1 - y^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b)\right) \tag{1}$$

, where $\mathbf{w} \in \mathbb{R}^d$ and bias $b \in \mathbb{R}$ are the model parameters. With the hinge loss per sample defined, we can then formulate the average loss for our model as:

$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{K} \sum_{i=1}^{K} \text{hinge}_{y^{(i)}}(\mathbf{x}^{(i)}) \tag{2}$$

Find the gradient of the loss function $\mathcal{L}(\mathbf{w}, b)$ with respect to the parameters i.e $\nabla_{\mathbf{w}}\mathcal{L}$ and $\nabla_b\mathcal{L}$.

Hint: An Indicator function, also known as a characteristic function, takes on the value of 1 at certain designated points and 0 at all other points. Mathematically, we can represent it as follows:

$$\mathbb{1}_{\{p<1\}} = \begin{cases} 1, & \text{if } p < 1 \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

5. (30 points) **Softmax classifier.** Complete the Softmax Jupyter notebook. Print out the entire workbook and related code sections in softmax.py, then submit them as a pdf to gradescope.

ECE C247

Sujit Silas Armstrong Suthahar

HW 2

1. a) $E_{\delta\sim N}[\tilde{\mathcal{L}}(\theta)] = \mathcal{L}(\theta) + R \;,\; E_{\delta\sim N}[\delta\delta^T] = \sigma^2 I$

$$(a-b)^2 = a^2 - 2ab + b^2$$

$$\tilde{\mathcal{L}}(\theta) = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - (x^{(i)} + \delta^{(i)})^T\theta)^2 \qquad \boxed{a = y^{(i)} - (x^{(i)})^T\theta \;,\; b = (\delta^{(i)})^T\theta}$$

$$\tilde{\mathcal{L}}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\Big((y^{(i)} - (x^{(i)})^T\theta)^2 - 2(y^{(i)} - (x^{(i)})^T\theta)(\delta^{(i)})^T\theta + ((\delta^{(i)})^T\theta)^2\Big)$$

$$E_{\delta\sim N}[\tilde{\mathcal{L}}(\theta)] = \frac{1}{N}\sum_{i=1}^{N}((y^{(i)} - (x^{(i)})^T\theta)^2 + E_{\delta\sim N}[((\delta^{(i)})^T\theta)^2])$$

we know, $E_{\delta\sim N}[\delta\delta^T] = \sigma^2 I$,

$$E_{\delta\sim N}[((\delta^{(i)})^T\theta)^2]) = \theta^T E_{\delta\sim N}[\delta^{(i)}\delta^{(i)T}]\theta$$

$$= \theta^T(\sigma^2 I)\theta = \sigma^2\|\theta\|_2^2$$

$$E_{\delta\sim N}[(y^{(i)} - x^{(i)T}\theta)^2] = (y^{(i)} - x^{(i)T}\theta)^2$$

$$E_{\delta\sim N}[(y^{(i)} - x^{(i)T}\theta)(\delta^{(i)T}\theta)] = 0$$

$\therefore$ we have,

$$E_{\delta\sim N}[\tilde{\mathcal{L}}(\theta)] = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - x^{(i)T}\theta)^2 - \sigma^2\|\theta\|_2^2$$

b) Addition of noise would have a L-2 regularization effect on the model based on σ^2's weightage

c) As $\sigma \to 0$, the effect of L-2 regularization decreases to no effect. This would lead to the overfitting of the data

d) As $\sigma \to \infty$, the L-2 norm term dominates leading to an over regularized model. This would lead to the underfitting of the data. Optimal θ will converge to 0, leading to a model that predicts y with no dependence on x. In other words, the model sacrifices fit to the data to minimize penalty. The objective of the cost function is to minimize the L2 norm of param θ & hence $\theta \to 0$ and the model will underfit the data.

2. On workbook

3. Softmax classifier gradient

Samples $\to$ $(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})$, where $x^{(j)} \in \mathbb{R}^n$, $y^{(j)} \in \{1, \dots, c\}, j = 1, \dots, m$

param $\Rightarrow$ $\theta = \{w_i, b_i\}_{i=1,\dots,c}$

$$\Pr(y^{(j)} \mid x^{(j)}, \theta) = \text{Softmax}_{y^{(j)}}(x^{(j)})$$

$$\text{Softmax}_{y^{(j)}}(x^{(j)}) = \frac{e^{w_{y^{(j)}}^T x^{(j)} + b_{y^{(j)}}}}{\sum_{k=1}^{c} e^{w_k^T x^{(j)} + b_k}}$$

$\nabla_{w_i} \mathcal{L}$ & $\nabla_{b_i} \mathcal{L}$, for $i = 1, \dots, c$

Let, $\tilde{x} = \begin{bmatrix} x \\ 1 \end{bmatrix}$, $\tilde{w}_i = \begin{bmatrix} w_i \\ b_i \end{bmatrix}$,

then $a_i(x) = w_i^T x + b_i = \tilde{w}_i^T \tilde{x}$.

$$\mathcal{L} = \log \prod_{j=1}^{m} P(y^{(j)} \mid x^{(j)}, \theta)$$

$$= \sum_{i=1}^{m} \log(\text{Softmax}_{y^{(i)}}(x^{(i)}))$$

$$= \sum_{i=1}^{m} \log\left(\frac{e^{w_{y^{(j)}}^T x^{(j)} + b_{y^{(j)}}}}{\sum_{k=1}^{c} e^{w_k^T x^{(j)} + b_k}}\right)$$

$$= \sum_{j=1}^{m} \left(\log(e^{w_{y^{(j)}}^T x^{(j)} + b_{y^{(j)}}}) - \log\left(\sum_{j=1}^{c} e^{w_k^T x^{(j)} + b_k}\right)\right)$$

$$= \sum_{j=1}^{m} \left((w_{y^{(j)}}^T x^{(j)} + b_y^{(j)}) - \log\left(\sum_{j=1}^{c} e^{w_k^T x^{(j)} + b_k}\right)\right)$$

Calculating $\nabla_{w_i} \mathcal{L}$ & $\nabla_{b_i} \mathcal{L}$

$$\frac{\partial \mathcal{L}}{\partial w_i} = \sum_{j=1}^{m} \left(\frac{\partial}{\partial w_i} \left[\left[w_{y(j)}^T x^{(j)} + b_y^{(j)} \right]\right] - \frac{\partial}{\partial w_i} \log \left(\sum_{j=1}^{c} e^{w_k^T x^{(j)} + b_k} \right) \right)$$

$$\frac{\partial}{\partial w_i} \left[w_{y(j)}^T x^{(j)} + b_y^{(j)} \right] = x^{(j)} \quad \text{if } i = y^{(j)} \text{ and } 0 \text{ otherwise}$$

$$\frac{\partial}{\partial w_i} \log \left(\sum_{j=1}^{c} e^{w_k^T x^{(j)} + b_k} \right) = \text{Softmax}\,(x^{(j)})\, x^{(j)}$$

$$\therefore \frac{\partial \mathcal{L}}{\partial w_i} = \sum_{j=1}^{m} \left(1_{y^{(j)}=i} - \text{Softmax}\,(x^{(j)}) \right) x^{(j)}$$

$$\text{Similarly, } \frac{\partial \mathcal{L}}{\partial b_i} = \sum_{j=1}^{m} \left(1_{y^{(j)}=i} - \text{Softmax}\,(x^{(j)}) \right)$$

4. Hinge loss gradient

$$D = \{ (x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots (x^{(k)}, y^{(k)}) \}$$

$$\text{where } x^{(i)} \in \mathbb{R}^d \;\&\; y^{(i)} \in \{-1, 1\}$$

$$\text{hinge}_{y(i)}(x^{(i)}) = \max(0, 1 - y^{(i)} (w^T x^{(i)} + b))$$

$$\text{where } w \in \mathbb{R}^d \;\&\; \text{bias } b \in \mathbb{R}$$

$$\mathcal{L}(W, b) = \frac{1}{k} \sum_{i=1}^{K} \text{hinge}_{y(i)} (x^{(i)})$$

Finding : $\nabla_w \mathcal{L}$ & $\nabla_b \mathcal{L}$

$$1_{\{p<1\}} = \begin{cases} 1, & \text{if } p<1 \\ 0, & \text{otherwise} \end{cases}$$

we are given,

$$\mathcal{L}(W,b) = \frac{1}{k}\sum_{i=1}^{K} hinge_{y^{(i)}}(x^{(i)})$$

$$= \max(0, 1 - y^{(i)}(w^T x^{(i)} + b))$$

we can treat it as a piece-wise function

$$hinge_{y^{(i)}}(x^{(i)}) = \begin{cases} 0 & , \text{If } y^{(i)}(w^T x^{(i)}+b) \geq 1 \\ 1 - y^{(i)}(w^T x^{(i)} + b, & \text{If } y^{(i)}(w^T x^{(i)}+b) < 1 \end{cases}$$

when $y^{(i)}(w^T x^{(i)}+b) < 1$,

$$\nabla_w hinge(x^{(i)}) = -y^{(i)} x^{(i)}$$

$$\therefore \nabla_W \mathcal{L} = \frac{1}{k}\sum_{i=k}^{k} 1\{y^{(i)}(w^T x^{(i)}+b) < 1\}(-y^{(i)} x^{(i)})$$

$$\nabla_b \mathcal{L} = \frac{1}{k}\sum_{i=k}^{k} 1\{y^{(i)}(w^T x^{(i)}+b) < 1\}(-y^{(i)})$$

5. Softmax Classifier Implementation on workbook

```
import numpy as np
import pdb


class KNN(object):

  def __init__(self):
    pass

  def train(self, X, y):
    """
    Inputs:
    - X is a numpy array of size (num_examples, D)
    - y is a numpy array of size (num_examples, )
    """
    self.X_train = X
    self.y_train = y

  def compute_distances(self, X, norm=None):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train.

    Inputs:
    - X: A numpy array of shape (num_test, D) containing test data.
    - norm: the function with which the norm is taken.

    Returns:
    - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
      is the Euclidean distance between the ith test point and the jth training
      point.
    """
    if norm is None:
      norm = lambda x: np.sqrt(np.sum(x**2))


    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    for i in np.arange(num_test):

      for j in np.arange(num_train):

        dists[i,j] = norm(X[i] - self.X_train[j])

    return dists

  def compute_L2_distances_vectorized(self, X):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train WITHOUT using any for loops.

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

    Returns:
    - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
      is the Euclidean distance between the ith test point and the jth training
      point.
    """

    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))

    X_squared = np.sum(X**2, axis=1).reshape(-1, 1)  # Shape (num_test, 1)
    X_train_squared = np.sum(self.X_train**2, axis=1).reshape(1, -1)  # Shape (1, num_train)
    cross_term = 2 * np.dot(X, self.X_train.T)  # Shape (num_test, num_train)
    dists = np.sqrt(X_squared + X_train_squared - cross_term)


    return dists


  def predict_labels(self, dists, k=1):
    """
```

```
Given a matrix of distances between test points and training points,
predict a label for each test point.

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

Returns:
- y: A numpy array of shape (num_test,) containing predicted labels for the
  test data, where y[i] is the predicted label for the test point X[i].
"""
num_test = dists.shape[0]
y_pred = np.zeros(num_test, dtype=int)

for i in range(num_test):
    closest_indices = np.argsort(dists[i])[:k]
    closest_y = self.y_train[closest_indices]
    y_pred[i] = np.bincount(closest_y).argmax()

return y_pred
```

This is the k-nearest neighbors workbook for ECE C147/C247 Assignment #2

Please follow the notebook linearly to implement k-nearest neighbors.

Please print out the workbook entirely when completed.

The goal of this workbook is to give you experience with the data, training and evaluating a simple classifier, k-fold cross validation, and as a Python refresher.

Import the appropriate libraries

In [1]:

```
import numpy as np # for doing most of our calculations
import matplotlib.pyplot as plt# for plotting
from utils.data_utils import load_CIFAR10 # function to load the CIFAR-10 dataset.
import numpy as np
import matplotlib.pyplot as plt
import time

# Load matplotlib images inline
%matplotlib inline

# These are important for reloading any code you write in external .py files.
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

In [2]:

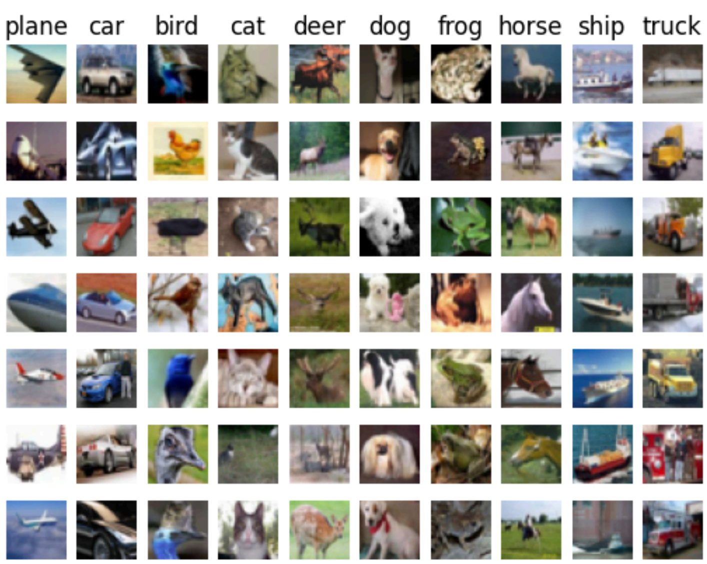
```
# Set the path to the CIFAR-10 data
cifar10_dir = '/Users/sujitsilas/Desktop/UCLA/Winter 2025/EE ENGR 247/Homeworks/HW2/student_copy/cifar-10-
X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

# 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,)
```

In [3]:

```python
# 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()
```

```
/opt/homebrew/lib/python3.11/site-packages/IPython/core/pylabtools.py:152: MatplotlibDeprecationWarning: save
fig() got unexpected keyword argument "orientation" which is no longer supported as of 3.3 and will become an
error two minor releases later
  fig.canvas.print_figure(bytes_io, **kw)
/opt/homebrew/lib/python3.11/site-packages/IPython/core/pylabtools.py:152: MatplotlibDeprecationWarning: save
fig() got unexpected keyword argument "dpi" which is no longer supported as of 3.3 and will become an error t
wo minor releases later
  fig.canvas.print_figure(bytes_io, **kw)
/opt/homebrew/lib/python3.11/site-packages/IPython/core/pylabtools.py:152: MatplotlibDeprecationWarning: save
fig() got unexpected keyword argument "facecolor" which is no longer supported as of 3.3 and will become an e
rror two minor releases later
  fig.canvas.print_figure(bytes_io, **kw)
/opt/homebrew/lib/python3.11/site-packages/IPython/core/pylabtools.py:152: MatplotlibDeprecationWarning: save
fig() got unexpected keyword argument "edgecolor" which is no longer supported as of 3.3 and will become an e
rror two minor releases later
  fig.canvas.print_figure(bytes_io, **kw)
/opt/homebrew/lib/python3.11/site-packages/IPython/core/pylabtools.py:152: MatplotlibDeprecationWarning: save
fig() got unexpected keyword argument "bbox_inches_restore" which is no longer supported as of 3.3 and will b
ecome an error two minor releases later
  fig.canvas.print_figure(bytes_io, **kw)
```



In [4]:

```python
# 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)
```

K-nearest neighbors

In the following cells, you will build a KNN classifier and choose hyperparameters via k-fold cross-validation.

In [5]:
```
# Import the KNN class

from nndl import KNN
```

In [6]:
```
# Declare an instance of the knn class.
knn = KNN()

# Train the classifier.
#   We have implemented the training of the KNN classifier.
#   Look at the train function in the KNN class to see what this does.
knn.train(X=X_train, y=y_train)
```

Questions

(1) Describe what is going on in the function knn.train().

(2) What are the pros and cons of this training step?

Answers

(1) The knn function simply stores the training data (X_train) and the corresponding labels (y_train) so they can be used later during the prediction phase to compute distances and make predictions.

(2) The pros are that it is relatively simple, straight forward, and fast. With its ability to store the entier dataset, it can adapt to different test cases without retraining. The cons are that the notion of diatance in a higher dimensional space becomes less intuitive. KNN takes up a lot of memory, may not be scalable and can involve high computational costs

KNN prediction

In the following sections, you will implement the functions to calculate the distances of test points to training points, and from this information, predict the class of the KNN.

In [7]:
```
# Implement the function compute_distances() in the KNN class.
# Do not worry about the input 'norm' for now; use the default definition of the norm
# in the code, which is the 2-norm.
# You should only have to fill out the clearly marked sections.

import time
time_start =time.time()

dists_L2 = knn.compute_distances(X=X_test)

print('Time to run code: {}'.format(time.time()-time_start))
print('Frobenius norm of L2 distances: {}'.format(np.linalg.norm(dists_L2, 'fro')))
```

```
Time to run code: 16.23002815246582
Frobenius norm of L2 distances: 7906696.077040902
```

Really slow code

Note: This probably took a while. This is because we use two for loops. We could increase the speed via vectorization, removing the for loops.

If you implemented this correctly, evaluating np.linalg.norm(dists_L2, 'fro') should return: ~7906696

KNN vectorization

The above code took far too long to run. If we wanted to optimize hyperparameters, it would be time-expensive. Thus, we will speed up the code by vectorizing it, removing the for loops.

Formula: ||x_i - x_j||^2 = ||x_i||^2 + ||x_j||^2 - 2 * x_i * x_j

In [8]:

```
# Implement the function compute_L2_distances_vectorized() in the KNN class.
# In this function, you ought to achieve the same L2 distance but WITHOUT any for loops.
# Note, this is SPECIFIC for the L2 norm.

time_start =time.time()
dists_L2_vectorized = knn.compute_L2_distances_vectorized(X=X_test)
print('Time to run code: {}'.format(time.time()-time_start))
print('Difference in L2 distances between your KNN implementations (should be 0): {}'.format(np.linalg.nor
```

```
Time to run code: 0.49208498001098633
Difference in L2 distances between your KNN implementations (should be 0): 0.0
```

Speedup

Depending on your computer speed, you should see a 10-100x speed up from vectorization. On our computer, the vectorized form took 0.36 seconds while the naive implementation took 38.3 seconds.

Implementing the prediction

Now that we have functions to calculate the distances from a test point to given training points, we now implement the function that will predict the test point labels.

In [9]:

```
# Implement the function predict_labels in the KNN class.
# Calculate the training error (num_incorrect / total_samples)
#   from running knn.predict_labels with k=1

# Calculate the training error for k=1
y_pred = knn.predict_labels(dists_L2_vectorized, k=1)

# Calculate the number of incorrect predictions
num_incorrect = np.sum(y_pred != y_test)

# Calculate the training error rate
error = (num_incorrect / y_test.shape[0])

print(f"Training error rate: {error}")


```

```
Training error rate: 0.726
```

If you implemented this correctly, the error should be: 0.726.

This means that the k-nearest neighbors classifier is right 27.4% of the time, which is not great, considering that chance levels are 10%.

Optimizing KNN hyperparameters

In this section, we'll take the KNN classifier that you have constructed and perform cross-validation to choose a best value of k, as well as a best choice of norm.

Create training and validation folds

First, we will create the training and validation folds for use in k-fold cross validation.

In [10]:

```python
# Create the dataset folds for cross-valdiation.
num_folds = 5

X_train_folds = []
y_train_folds =  []

# Number of folds for cross-validation
num_folds = 5

# Shuffle the data indices
num_train = X_train.shape[0]
indices = np.arange(num_train)
np.random.shuffle(indices)

# Split the indices into folds
fold_size = num_train // num_folds
X_train_folds = []
y_train_folds = []

for i in range(num_folds):

    start_idx = i * fold_size
    if i == num_folds - 1:
        end_idx = num_train
    else:
        end_idx = start_idx + fold_size

    fold_indices = indices[start_idx:end_idx]

    # Append the corresponding data and labels to the folds
    X_train_folds.append(X_train[fold_indices])
    y_train_folds.append(y_train[fold_indices])

# Print the number of samples in each fold to verify
for i in range(num_folds):
    print(f"Fold {i+1}: {X_train_folds[i].shape[0]} samples")

```

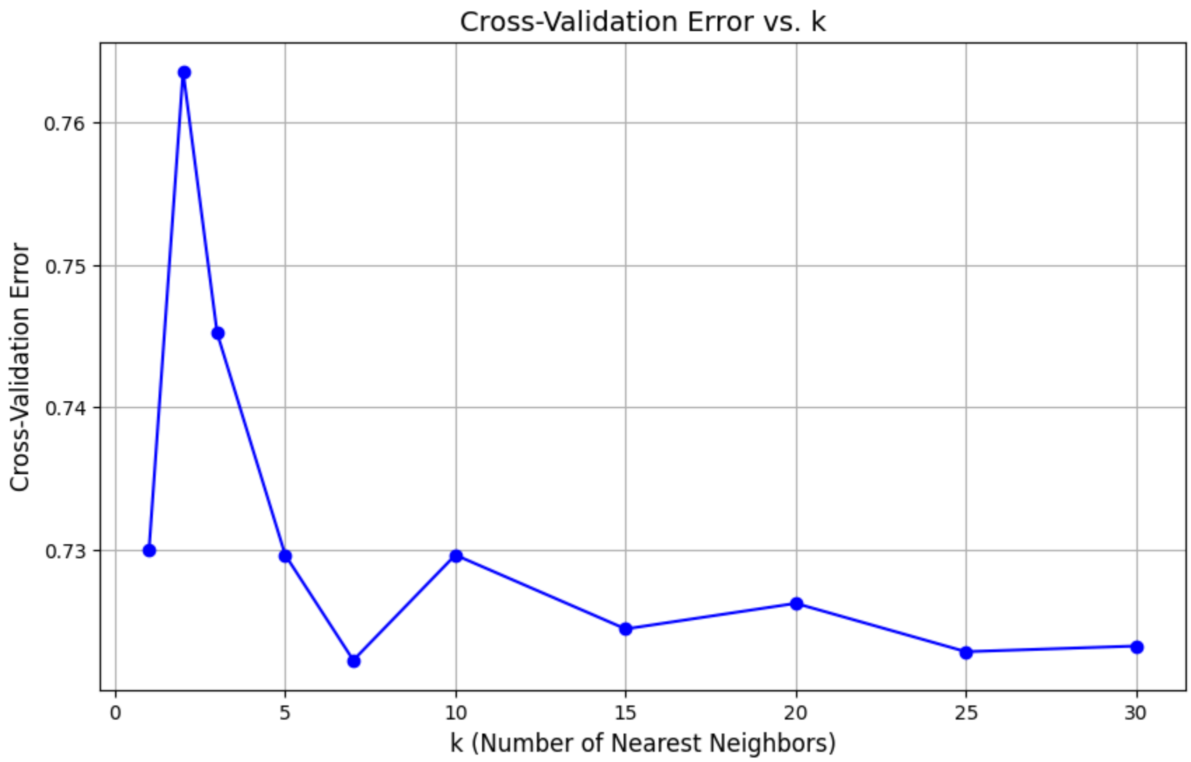
```
Fold 1: 1000 samples
Fold 2: 1000 samples
Fold 3: 1000 samples
Fold 4: 1000 samples
Fold 5: 1000 samples
```

Optimizing the number of nearest neighbors hyperparameter.

In this section, we select different numbers of nearest neighbors and assess which one has the lowest k-fold cross validation error.

In [11]:

```
import numpy as np
import matplotlib.pyplot as plt
import time

time_start = time.time()


ks = [1, 2, 3, 5, 7, 10, 15, 20, 25, 30]

cv_errors = []

for k in ks:
    fold_errors = []

    for i in range(num_folds):

        X_val_fold = X_train_folds[i]
        y_val_fold = y_train_folds[i]
        X_train_fold = np.concatenate(X_train_folds[:i] + X_train_folds[i+1:], axis=0)
        y_train_fold = np.concatenate(y_train_folds[:i] + y_train_folds[i+1:], axis=0)


        knn = KNN()
        knn.train(X_train_fold, y_train_fold)


        dists = knn.compute_L2_distances_vectorized(X_val_fold)


        y_pred = knn.predict_labels(dists, k=k)


        fold_error = np.mean(y_pred != y_val_fold)
        fold_errors.append(fold_error)


    avg_error = np.mean(fold_errors)
    cv_errors.append(avg_error)

# Plot the results
plt.figure(figsize=(10, 6))
plt.plot(ks, cv_errors, marker='o', linestyle='-', color='b')
plt.xlabel('k (Number of Nearest Neighbors)', fontsize=12)
plt.ylabel('Cross-Validation Error', fontsize=12)
plt.title('Cross-Validation Error vs. k', fontsize=14)
plt.grid(True)
plt.show()

print('Computation time: %.2f seconds' % (time.time() - time_start))
```




```
Computation time: 40.55 seconds
```

Questions:

(1) What value of k is best amongst the tested k's?

(2) What is the cross-validation error for this value of k?

In [12]:
```python
print(f"Best k: {ks[np.argmin(cv_errors)]}")
print(f"Lowest error: {np.min(cv_errors)}")
```

```
Best k: 7
Lowest error: 0.7222000000000001
```

Answers:

(1) k=20

(2) The lowest error rate was 0.722

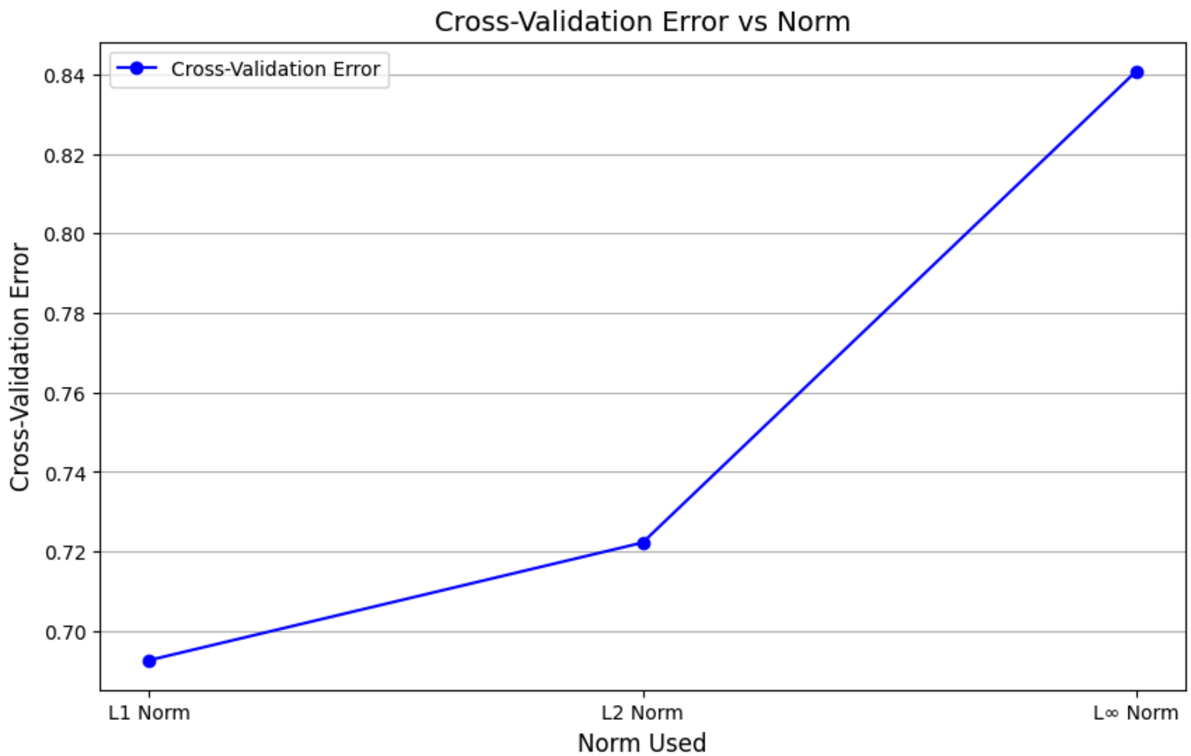
Optimizing the norm

Next, we test three different norms (the 1, 2, and infinity norms) and see which distance metric results in the best cross-validation performance.

In [21]:

```python
time_start =time.time()
best_k = 7
L1_norm = lambda x: np.linalg.norm(x, ord=1)
L2_norm = lambda x: np.linalg.norm(x, ord=2)
Linf_norm = lambda x: np.linalg.norm(x, ord= np.inf)
norms = [L1_norm, L2_norm, Linf_norm]

# Initialize storage for errors
cv_errors = []

time_start = time.time()

# Perform cross-validation for each norm
for norm in norms:
    fold_errors = []

    for i in range(num_folds):
        X_val_fold = X_train_folds[i]
        y_val_fold = y_train_folds[i]
        X_train_fold = np.concatenate(X_train_folds[:i] + X_train_folds[i+1:], axis=0)
        y_train_fold = np.concatenate(y_train_folds[:i] + y_train_folds[i+1:], axis=0)

        knn = KNN()
        knn.train(X_train_fold, y_train_fold)

        dists = knn.compute_distances(X=X_val_fold, norm=norm)

        y_pred = knn.predict_labels(dists, k=best_k)

        fold_error = np.mean(y_pred != y_val_fold)
        fold_errors.append(fold_error)

    avg_error = np.mean(fold_errors)
    cv_errors.append(avg_error)

# Plot the results
norm_names = ['L1 Norm', 'L2 Norm', 'L∞ Norm']
plt.figure(figsize=(10, 6))
plt.plot(norm_names, cv_errors, marker='o', linestyle='-', color='b', label="Cross-Validation Error")
plt.xlabel('Norm Used', fontsize=12)
plt.ylabel('Cross-Validation Error', fontsize=12)
plt.title('Cross-Validation Error vs Norm', fontsize=14)
plt.legend()
plt.grid(axis='y')
plt.show()

# Print computation time
print('Computation time: %.2f seconds' % (time.time() - time_start))

# Output the best norm and corresponding error
best_norm_idx = np.argmin(cv_errors)
print(f"Best norm: {norm_names[best_norm_idx]}")
print(f"Lowest cross-validation error: {cv_errors[best_norm_idx]:.4f}")
```

```
/opt/homebrew/lib/python3.11/site-packages/IPython/core/pylabtools.py:152: MatplotlibDeprecationWarning: save
fig() got unexpected keyword argument "facecolor" which is no longer supported as of 3.3 and will become an e
rror two minor releases later
  fig.canvas.print_figure(bytes_io, **kw)
/opt/homebrew/lib/python3.11/site-packages/IPython/core/pylabtools.py:152: MatplotlibDeprecationWarning: save
fig() got unexpected keyword argument "edgecolor" which is no longer supported as of 3.3 and will become an e
rror two minor releases later
  fig.canvas.print_figure(bytes_io, **kw)
/opt/homebrew/lib/python3.11/site-packages/IPython/core/pylabtools.py:152: MatplotlibDeprecationWarning: save
fig() got unexpected keyword argument "bbox_inches_restore" which is no longer supported as of 3.3 and will b
ecome an error two minor releases later
  fig.canvas.print_figure(bytes_io, **kw)
```




```
Computation time: 583.88 seconds
Best norm: L1 Norm
Lowest cross-validation error: 0.6926
```

Questions:

(1) What norm has the best cross-validation error?

(2) What is the cross-validation error for your given norm and k?

Answers:

(1) L1 norm has the lowest error

(2) The corss vlalidation error for the the L1 norm was 0.6926

Evaluating the model on the testing dataset.

Now, given the optimal k and norm you found in earlier parts, evaluate the testing error of the k-nearest neighbors model.

In [22]:

```
k_optimal = 7
norm_optimal = L1_norm

# Initialize the KNN model with the optimal k
knn = KNN()
knn.train(X_train, y_train)
dists = knn.compute_distances(X=X_test, norm=norm_optimal)

# Predict labels for the test set based on the optimal k
y_pred = []
for dist in dists:
    nearest_neighbors = np.argsort(dist)[:k_optimal]
    label = np.bincount(y_train[nearest_neighbors]).argmax()
    y_pred.append(label)
y_pred = np.array(y_pred)


error = np.mean(y_pred != y_test)


print('Error rate achieved: {}'.format(error))
```

```
Error rate achieved: 0.698
```

In [23]:

```
naive_error = 0.726  # Training error for naive k=1, L2 norm
improvement = naive_error - error
print(f"Error improvement by cross-validation: {improvement:.4f}")

```

```
Error improvement by cross-validation: 0.0280
```

Question:

How much did your error improve by cross-validation over naively choosing $k = 1$ and using the L2-norm?

Answer:

The error changed by 0.0280

```python
import numpy as np


class Softmax(object):

  def __init__(self, dims=[10, 3073]):
    self.init_weights(dims=dims)

  def init_weights(self, dims):
    """
    Initializes the weight matrix of the Softmax classifier.
    Note that it has shape (C, D) where C is the number of
    classes and D is the feature size.
    """
    self.W = np.random.normal(size=dims) * 0.0001

  def loss(self, X, y):
    """
    Calculates the softmax loss.

    Inputs have dimension D, there are C classes, and we operate on minibatches
    of N examples.

    Inputs:
    - X: A numpy array of shape (N, D) containing a minibatch of data.
    - y: A numpy array of shape (N,) containing training labels; y[i] = c means
      that X[i] has label c, where 0 <= c < C.

    Returns a tuple of:
    - loss as single float
    """

    loss = 0.0

    N = X.shape[0]

    scores = X.dot(self.W.T)

    exp_scores = np.exp(scores)
    sums = np.sum(exp_scores, axis=1, keepdims=True)
    probs = exp_scores / sums


    correct_log_probs = -np.log(probs[np.arange(N), y])
    loss = np.sum(correct_log_probs) / N

    return loss

  def loss_and_grad(self, X, y):
    """
    Same as self.loss(X, y), except that it also returns the gradient.

    Output: grad -- a matrix of the same dimensions as W containing
      the gradient of the loss with respect to W.
    """

    loss = 0.0
    N = X.shape[0]

    grad = np.zeros_like(self.W)

    scores = X.dot(self.W.T)
    #scores -= np.max(scores, axis=1, keepdims=True)

    exp_scores = np.exp(scores)
    sums = np.sum(exp_scores, axis=1, keepdims=True)
    probs = exp_scores / sums

    correct_log_probs = -np.log(probs[np.arange(N), y])
    loss = np.sum(correct_log_probs) / N

    dscores = probs.copy()
    dscores[np.arange(N), y] -= 1
    dscores /= N

    # Now backprop into W
```

```
  # W is of shape (C, D), X is (N, D), dscores is (N, C)
  # => grad = dscores^T * X, shape = (C, D)
  grad = dscores.T.dot(X)

  return loss, grad


def grad_check_sparse(self, X, y, your_grad, num_checks=10, h=1e-5):
  """
  sample a few random elements and only return numerical
  in these dimensions.
  """

  for i in np.arange(num_checks):
    ix = tuple([np.random.randint(m) for m in self.W.shape])

    oldval = self.W[ix]
    self.W[ix] = oldval + h # increment by h
    fxph = self.loss(X, y)
    self.W[ix] = oldval - h # decrement by h
    fxmh = self.loss(X,y) # evaluate f(x - h)
    self.W[ix] = oldval # reset

    grad_numerical = (fxph - fxmh) / (2 * h)
    grad_analytic = your_grad[ix]
    rel_error = abs(grad_numerical - grad_analytic) / (abs(grad_numerical) + abs(grad_analytic))
    print('numerical: %f analytic: %f, relative error: %e' % (grad_numerical, grad_analytic, rel_error))

def fast_loss_and_grad(self, X, y):
  """
  A vectorized implementation of loss_and_grad. It shares the same
  inputs and ouptuts as loss_and_grad.
  """
  loss = 0.0
  grad = np.zeros(self.W.shape)

  N = X.shape[0]

  # 1) Compute scores
  scores = X.dot(self.W.T)
  scores -= np.max(scores, axis=1, keepdims=True)

  # 2) Exponentiate and normalize
  exp_scores = np.exp(scores)
  sums = np.sum(exp_scores, axis=1, keepdims=True)
  probs = exp_scores / sums

  # 3) Compute loss
  correct_log_probs = -np.log(probs[np.arange(N), y])
  loss = np.sum(correct_log_probs) / N

  # 4) Compute gradient
  dscores = probs
  dscores[np.arange(N), y] -= 1
  dscores /= N
  grad = dscores.T.dot(X)

  return loss, grad


def train(self, X, y, learning_rate=1e-3, num_iters=100,
          batch_size=200, verbose=False):
  """
  Train this linear classifier using stochastic gradient descent.

  Inputs:
  - X: A numpy array of shape (N, D) containing training data; there are N
    training samples each of dimension D.
  - y: A numpy array of shape (N,) containing training labels; y[i] = c
    means that X[i] has label 0 <= c < C for C classes.
  - learning_rate: (float) learning rate for optimization.
  - num_iters: (integer) number of steps to take when optimizing
  - batch_size: (integer) number of training examples to use at each step.
  - verbose: (boolean) If true, print progress during optimization.

  Outputs:
  A list containing the value of the loss function at each training iteration.
  """
```

```
    num_train, dim = X.shape
    num_classes = np.max(y) + 1

    self.init_weights(dims=[np.max(y) + 1, X.shape[1]])

    loss_history = []

    for it in np.arange(num_iters):
      X_batch = None
      y_batch = None

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

      loss, grad = self.fast_loss_and_grad(X_batch, y_batch)
      loss_history.append(loss)

      self.W -= learning_rate * grad

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


    return loss_history

  def predict(self, X):
    """
    Inputs:
    - X: N x D array of training data. Each row is a D-dimensional point.

    Returns:
    - y_pred: Predicted labels for the data in X. y_pred is a 1-dimensional
      array of length N, and each element is an integer giving the predicted
      class.
    """
    y_pred = np.zeros(X.shape[1])

    scores = X.dot(self.W.T)

    y_pred = np.argmax(scores, axis=1)

    return y_pred
```

This is the softmax workbook for ECE C147/C247 Assignment #2

Please follow the notebook linearly to implement a softmax classifier.

Please print out the workbook entirely when completed.

The goal of this workbook is to give you experience with training a softmax classifier.

In [1]:

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

%matplotlib inline
%load_ext autoreload
%autoreload 2
```

In [2]:

```python
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 = '/Users/sujitsilas/Desktop/UCLA/Winter 2025/EE ENGR 247/Homeworks/HW2/student_copy/cifar
    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,)
```

Training a softmax classifier.

The following cells will take you through building a softmax classifier. You will implement its loss function, then subsequently train it with gradient descent. Finally, you will choose the learning rate of gradient descent to optimize its classification performance.

In [3]:

```python
from nndl import Softmax
```

In [4]:
```
# Declare an instance of the Softmax class.
# Weights are initialized to a random value.
# Note, to keep people's first solutions consistent, we are going to use a random seed.

np.random.seed(1)

num_classes = len(np.unique(y_train))
num_features = X_train.shape[1]

softmax = Softmax(dims=[num_classes, num_features])
```

Softmax loss

In [5]:
```
## Implement the loss function of the softmax using a for loop over
#  the number of examples

loss = softmax.loss(X_train, y_train)
```

In [6]:
```
print(loss)
```

```
2.3277607028048757
```

Question:

You'll notice the loss returned by the softmax is about 2.3 (if implemented correctly). Why does this make sense?

Answer:

These are results from the untrained classifer that are assigning probabilities based on the number of classes present in the dataset. The probability for 1/10 (-log(1/10)). In the case of 10 classes, the classifier is predicting each class with probability 1/10.

Softmax gradient

In [7]:
```
## Calculate the gradient of the softmax loss in the Softmax class.
# For convenience, we'll write one function that computes the loss
#   and gradient together, softmax.loss_and_grad(X, y)
# You may copy and paste your loss code from softmax.loss() here, and then
#   use the appropriate intermediate values to calculate the gradient.

loss, grad = softmax.loss_and_grad(X_dev,y_dev)

# Compare your gradient to a gradient check we wrote.
# You should see relative gradient errors on the order of 1e-07 or less if you implemented the gradient co
softmax.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: 0.512844 analytic: 0.512843, relative error: 2.129439e-08
numerical: 0.321468 analytic: 0.321468, relative error: 8.974866e-08
numerical: -1.458204 analytic: -1.458204, relative error: 1.216753e-08
numerical: -0.413745 analytic: -0.413745, relative error: 1.364065e-08
numerical: 1.454856 analytic: 1.454856, relative error: 3.809682e-08
numerical: 0.387829 analytic: 0.387829, relative error: 1.361799e-07
numerical: -0.112319 analytic: -0.112319, relative error: 3.713372e-07
numerical: -0.559600 analytic: -0.559600, relative error: 2.054512e-08
numerical: 0.279579 analytic: 0.279579, relative error: 2.465414e-08
numerical: -2.448996 analytic: -2.448996, relative error: 1.569323e-08
```

A vectorized version of Softmax

To speed things up, we will vectorize the loss and gradient calculations. This will be helpful for stochastic gradient descent.

In [8]:
```
import time
```

In [9]:

```
## Implement softmax.fast_loss_and_grad which calculates the loss and gradient
#    WITHOUT using any for loops.

# Standard loss and gradient
tic = time.time()
loss, grad = softmax.loss_and_grad(X_dev, y_dev)
toc = time.time()
print('Normal loss / grad_norm: {} / {} computed in {}s'.format(loss, np.linalg.norm(grad, 'fro'), toc - t

tic = time.time()
loss_vectorized, grad_vectorized = softmax.fast_loss_and_grad(X_dev, y_dev)
toc = time.time()
print('Vectorized loss / grad: {} / {} computed in {}s'.format(loss_vectorized, np.linalg.norm(grad_vector

# The losses should match but your vectorized implementation should be much faster.
print('difference in loss / grad: {} /{} '.format(loss - loss_vectorized, np.linalg.norm(grad - grad_vecto

# You should notice a speedup with the same output.
```

```
Normal loss / grad_norm: 2.332541806941961 / 276.4220540057411 computed in 0.049748897552490234s
Vectorized loss / grad: 2.332541806941961 / 276.4220540057411 computed in 0.056962013244628906s
difference in loss / grad: 0.0 /3.972403559145082e-14
```

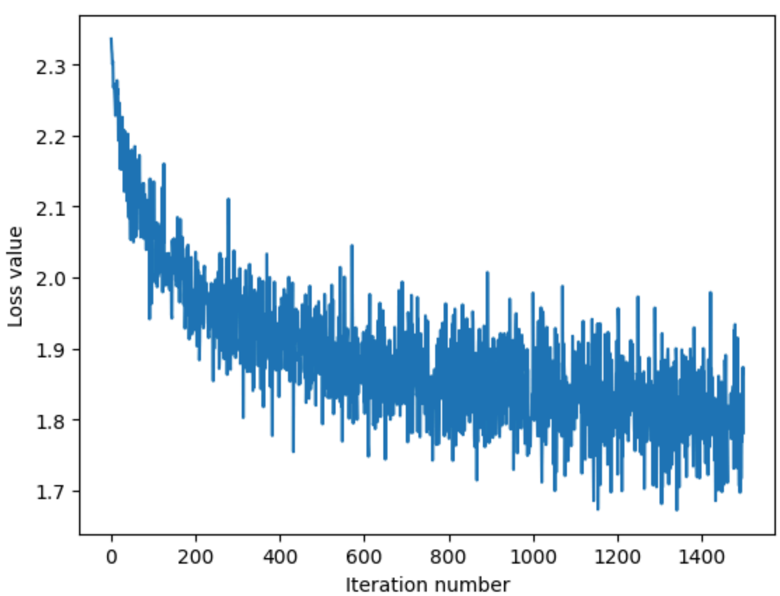
Stochastic gradient descent

We now implement stochastic gradient descent. This uses the same principles of gradient descent we discussed in class, however, it calculates the gradient by only using examples from a subset of the training set (so each gradient calculation is faster).

In [10]:

```python
# Implement softmax.train() by filling in the code to extract a batch of data
# and perform the gradient step.
import time

tic = time.time()
loss_hist = softmax.train(X_train, y_train, learning_rate=1e-7,
                      num_iters=1500, verbose=True)
toc = time.time()
print('That took {}s'.format(toc - tic))

plt.plot(loss_hist)
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```

```
iteration 0 / 1500: loss 2.336593
iteration 100 / 1500: loss 2.055722
iteration 200 / 1500: loss 2.035775
iteration 300 / 1500: loss 1.981335
iteration 400 / 1500: loss 1.958314
iteration 500 / 1500: loss 1.862265
iteration 600 / 1500: loss 1.853261
iteration 700 / 1500: loss 1.835306
iteration 800 / 1500: loss 1.829389
iteration 900 / 1500: loss 1.899216
iteration 1000 / 1500: loss 1.978350
iteration 1100 / 1500: loss 1.847080
iteration 1200 / 1500: loss 1.841145
iteration 1300 / 1500: loss 1.791040
iteration 1400 / 1500: loss 1.870580
That took 54.61989188194275s

/opt/homebrew/lib/python3.11/site-packages/IPython/core/pylabtools.py:152: MatplotlibDeprecationWarning: save
fig() got unexpected keyword argument "orientation" which is no longer supported as of 3.3 and will become an
error two minor releases later
  fig.canvas.print_figure(bytes_io, **kw)
/opt/homebrew/lib/python3.11/site-packages/IPython/core/pylabtools.py:152: MatplotlibDeprecationWarning: save
fig() got unexpected keyword argument "dpi" which is no longer supported as of 3.3 and will become an error t
wo minor releases later
  fig.canvas.print_figure(bytes_io, **kw)
/opt/homebrew/lib/python3.11/site-packages/IPython/core/pylabtools.py:152: MatplotlibDeprecationWarning: save
fig() got unexpected keyword argument "facecolor" which is no longer supported as of 3.3 and will become an e
rror two minor releases later
  fig.canvas.print_figure(bytes_io, **kw)
/opt/homebrew/lib/python3.11/site-packages/IPython/core/pylabtools.py:152: MatplotlibDeprecationWarning: save
fig() got unexpected keyword argument "edgecolor" which is no longer supported as of 3.3 and will become an e
rror two minor releases later
  fig.canvas.print_figure(bytes_io, **kw)
/opt/homebrew/lib/python3.11/site-packages/IPython/core/pylabtools.py:152: MatplotlibDeprecationWarning: save
fig() got unexpected keyword argument "bbox_inches_restore" which is no longer supported as of 3.3 and will b
ecome an error two minor releases later
  fig.canvas.print_figure(bytes_io, **kw)
```



Evaluate the performance of the trained softmax classifier on the validation data.

In [11]:
```
## Implement softmax.predict() and use it to compute the training and testing error.

y_train_pred = softmax.predict(X_train)
print('training accuracy: {}'.format(np.mean(np.equal(y_train,y_train_pred), )))
y_val_pred = softmax.predict(X_val)
print('validation accuracy: {}'.format(np.mean(np.equal(y_val, y_val_pred)), ))
```

```
training accuracy: 0.3811428571428571
validation accuracy: 0.398
```

Optimize the softmax classifier

In [12]:
```
np.finfo(float).eps
```

Out[12]:
```
2.220446049250313e-16
```

In [14]:

```python
# Example set of candidate learning rates
learning_rates = [1e-5, 5e-5, 1e-4, 5e-4, 1e-3]

best_lr = None
best_val_acc = -1
best_softmax = None

# Loop through each learning rate
for lr in learning_rates:
    # Create a new Softmax instance for each learning rate
    classifier = Softmax(dims=[num_classes, X_train.shape[1]])

    # Train the softmax classifier
    _ = classifier.train(
        X_train,
        y_train,
        learning_rate=lr,
        num_iters=1500,
        batch_size=200,
        verbose=False
    )

    # Evaluate on the validation set
    y_val_pred = classifier.predict(X_val)
    val_acc = np.mean(y_val_pred == y_val)

    # Print validation accuracy for this learning rate
    print(f"Learning rate: {lr:.1e}, Validation accuracy: {val_acc:.4f}")

    # Update the best classifier if this one is better
    if val_acc > best_val_acc:
        best_val_acc = val_acc
        best_lr = lr
        best_softmax = classifier

# Output the best learning rate and validation accuracy
print("\nBest learning rate:", best_lr)
print(f"Best validation accuracy: {best_val_acc:.4f}")

# Evaluate the best model on the test set
y_test_pred = best_softmax.predict(X_test)
test_acc = np.mean(y_test_pred == y_test)
test_error_rate = 1.0 - test_acc

# Output test performance of the best model
print("\nTest accuracy of the best model:", test_acc)
print(f"Test error rate of the best model: {test_error_rate:.4f}")

```

```
Learning rate: 1.0e-05, Validation accuracy: 0.2920
Learning rate: 5.0e-05, Validation accuracy: 0.2790
Learning rate: 1.0e-04, Validation accuracy: 0.2800
Learning rate: 5.0e-04, Validation accuracy: 0.2650
Learning rate: 1.0e-03, Validation accuracy: 0.2940

Best learning rate: 0.001
Best validation accuracy: 0.2940

Test accuracy of the best model: 0.28
Test error rate of the best model: 0.7200
```