Myles Johnson - 205868607

ECE C147/C247: Neural Networks & Deep Learning, Winter 2023 Homework #2

Noisy Linear Regression

a) Express the expectation of the modified loss over the gaussian noise, in terms of the original loss plus a term independent of the dataset $\mathcal{D}$

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

Simplify the inner term of the $\tilde{\mathcal{L}}(\theta)$ sum:

$$= (y^{(i)} - (x^{(i)} + \delta^{(i)})^T\theta^2)$$

$$= (y^{(i)} - (x^{(i)})^T\theta^2 - \delta^{(i)^T}\theta^2)$$

$$= (y^{(i)} - (x^{(i)})^T\theta)^2 - 2(y^{(i)} - (x^{(i)})^T\theta)((\delta^{(i)})^T\theta) + (\delta^{(i)^T}\theta^2)$$

Since $\mathbb{E}$ is a linear operator, we can apply it to each term in the sum:

$$\mathbb{E}_{\delta\sim\mathcal{N}}[(y^{(i)} - (x^{(i)} + \delta^{(i)})^T\theta^2)] = \mathbb{E}_{\delta\sim\mathcal{N}}[(y^{(i)} - (x^{(i)})^T\theta)^2] - \mathbb{E}_{\delta\sim\mathcal{N}}[2(y^{(i)} - (x^{(i)})^T\theta)((\delta^{(i)})^T\theta)] + \mathbb{E}_{\delta\sim\mathcal{N}}[(\delta^{(i)^T}\theta^2)]$$

Looking at the terms with δ in them, we can see that the first term is a constant, and the second term is a linear function of δ. So, we can apply the linearity of expectation to the second term:

$$\mathbb{E}_{\delta\sim\mathcal{N}}[-2(y^{(i)}-(x^{(i)})^T\theta)((\delta^{(i)})^T\theta)]$$

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

Since $\mathbb{E}_{\delta\sim\mathcal{N}}[\delta^{(i)}] = 0 \in \mathbb{R}$:

$$\mathbb{E}_{\delta\sim\mathcal{N}}[-2(y^{(i)}-(x^{(i)})^T\theta)((\delta^{(i)})^T\theta)] = 0$$

The third term also contains δ, we can apply the linearity of expectation to this term as well:

$$\mathbb{E}_{\delta\sim\mathcal{N}}[(\delta^{(i)^T}\theta^2)]$$

$$= \mathbb{E}_{\delta\sim\mathcal{N}}[(\theta^T\delta^{(i)}\delta^{(i)^T}\theta)]$$

$$= \theta^T\mathbb{E}_{\delta\sim\mathcal{N}}[(\delta^{(i)}\delta^{(i)^T})]\theta$$

Since $\mathbb{E}_{\delta\sim\mathcal{N}}[\delta^{(i)}\delta^{(i)^T}] = \sigma^2\mathbf{I}$:

$$\mathbb{E}_{\delta\sim\mathcal{N}}[(\delta^{(i)^T}\theta^2)] = \sigma^2\theta^T\mathbf{I}\theta = \sigma^2\|\theta\|_2^2$$

Therefore, the overall expectation of modified loss is:

$$\mathbb{E}_{\delta\sim\mathcal{N}}[(y^{(i)}-(x^{(i)}+\delta^{(i)})^T\theta^2)] = (y^{(i)}-(x^{(i)}+\delta^{(i)})^T\theta^2) + \sigma^2\|\theta\|_2^2$$

where $\mathcal{L}(\theta) = (y^{(i)}-(x^{(i)}+\delta^{(i)})^T\theta^2)$ so, $\mathcal{R} = \sigma^2\|\theta\|_2^2$ which is not a function of $\mathcal{D}$.

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

If R is equal to $\sigma^2\|\theta\|_2^2$, the addition of noise to the model's parameters, as a regularization technique, would have the effect of adding a term to the loss function which is proportional to the L2-norm of the parameters, multiplied by σ^2.

c) Suppose $\sigma \to 0$, what effect would this have on the model?

If $\sigma \to 0$, this term would become very small and have a negligible effect on the model. In this case, the model would not be regularized and could overfit to the training data.

d) Suppose $\sigma \to \infty$, what effect would this have on the model?

On the other hand, if $\sigma \to \infty$, this term would become very large, and it would have a significant effect on the model. In this case, the model would be heavily regularized and could underfit to the training data. The model would be more robust to the noise but would be less accurate.

2. K-Nearest Neighbors

Code sections for this part are in `knn.py` :

```
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 trainin
        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[
          is the Euclidean distance between the ith test point and the jth
          point.
        """
        if norm is None:
            norm = lambda x: np.sqrt(np.sum(x**2))
            # norm = 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):
                # ========================================================
                # YOUR CODE HERE:
                #   Compute the distance between the ith test point and th
                #   training point using norm(), and store the result in d
                # ========================================================

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

                # ========================================================
                # END YOUR CODE HERE
                # ========================================================

        return dists

    def compute_L2_distances_vectorized(self, X):
        """
        Compute the distance between each test point in X and each trainin
        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[
          is the Euclidean distance between the ith test point and the jth
          point.
        """
        num_test = X.shape[0]
        num_train = self.X_train.shape[0]
        dists = np.zeros((num_test, num_train))

        # ================================================================
        # YOUR CODE HERE:
        #   Compute the L2 distance between the ith test point and the jth
        #   training point and store the result in dists[i, j].  You may
        #   NOT use a for loop (or list comprehension).  You may only use
        #   numpy operations.
        #
        #   HINT: use broadcasting.  If you have a shape (N,1) array and
        #   a shape (M,) array, adding them together produces a shape (N,
        #   array.
        # ================================================================

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

        # ================================================================
        # END YOUR CODE HERE
        # ================================================================

        return dists

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

        Inputs:
        - dists: A numpy array of shape (num_test, num_train) where dists[
          gives the distance between the ith test point and the jth traini

        Returns:
        - y: A numpy array of shape (num_test,) containing predicted label
          test data, where y[i] is the predicted label for the test point
        """
        num_test = dists.shape[0]
        y_pred = np.zeros(num_test)
        for i in np.arange(num_test):
            # A list of length k storing the labels of the k nearest neigh
            # the ith test point.
            closest_y = []
            # ============================================================
            # YOUR CODE HERE:
            #   Use the distances to calculate and then store the labels o
            #   the k-nearest neighbors to the ith test point.  The functi
            #   numpy.argsort may be useful.
            #
            #   After doing this, find the most common label of the k-near
            #   neighbors.  Store the predicted label of the ith training
            #   as y_pred[i].  Break ties by choosing the smaller label.
            # ============================================================

            closest_y = list(self.y_train[np.argsort(dists[i])[:k]])
            y_pred[i] = max(sorted(list(set(closest_y))), key=closest_y.co

            # ============================================================
            # END YOUR CODE HERE
            # ============================================================

        return y_pred
```

(Attached workbook below)

3. Softmax Classifier Gradient

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 $i = 1, ..., c$

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 $\tilde{x} = \begin{bmatrix}\mathbf{x} \\ 1\end{bmatrix}$, $\tilde{w}_i = \begin{bmatrix}\mathbf{w_i} \\ b_i\end{bmatrix}$, then $a_i(x) = \mathbf{w_i}^T\mathbf{x} + b_i = \tilde{w}_i^T\tilde{x}$. This unifies $\nabla_{\mathbf{w_i}}\mathcal{L}$ and $\nabla_{b_i}\mathcal{L}$ into a single gradient $\nabla_{\tilde{w}_i}\mathcal{L}$.

For a softmax classifier, the log-likelihood function is (via Discussion 3):

$$\mathcal{L}(\mathbf{w_1}, ..., \mathbf{w_c}, b_1, ..., b_c) = \frac{1}{N}\sum_{n=1}^{N} \log\left(\frac{e^{(\mathbf{w_{y_n}}^T\mathbf{x_n}+b_{y_n})}}{\sum_{j=1}^{c} e^{(\mathbf{w_j}^T\mathbf{x_n}+b_j)}}\right)$$

Using the NOTE that gives the single gradient $\nabla_{\tilde{w}_i}\mathcal{L}$ notation, the log-likelihood function becomes:

$$\mathcal{L}(\tilde{w}_1, ..., \tilde{w}_c) = \frac{1}{N}\sum_{n=1}^{N} \log\left(\frac{e^{(\tilde{w}_{y_n}^T\ x_n)}}{\sum_{j=1}^{c} e^{(\tilde{w}_j^T\ x_n)}}\right)$$

And the gradient of the log-likelihood function with respect to $\tilde{w}_i$ becomes:

$$\nabla_{\tilde{w}_i}\mathcal{L} = \frac{1}{N}\sum_{n=1}^{N}\left(\frac{e^{(\tilde{w}_i^T\ x_n)}}{\sum_{j=1}^{c} e^{(\tilde{w}_j^T\ x_n)}} - \mathbb{I}_{\{y_n=i\}}\right)\ x_n \qquad \because \mathbb{I}_{\{y_n=i\}}$$ is an indicator function that is 1 if $y_n = i$ and 0 otherwise.

With this notation, we can express the gradient of the log-likelihood function with respect to a single vector $\tilde{w}_i$, which includes both the gradient with respect to the parameters of the i-th class, $\mathbf{w_i}$ and b_i, rather than expressing them separately. This gradient tells us how much the log-likelihood changes when we change the parameters $\tilde{w}_i$, and it takes into account all the data points. The first part of the equation, $\frac{e^{(\ w_i^T\ x_n)}}{\sum_{j=1}^{c} e^{(\ w_j^T\ x_n)}}$, tells us the predicted probability of the i-th class for the n-th data point, and the second part, $\mathbb{I}_{\{y_n=i\}}$, tells us the true label for the n-th data point. By subtracting the true label from the predicted probability, we can see how well our model is doing for each data point, and by adding up the results for all data points, we can see how well our model is doing overall.

4. Hinge Loss Gradient

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 $\nabla_b\mathcal{L}$

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

Since the gradient is a linear operator, we can write the gradient of the loss function as the sum of the gradients of each term in the loss function:

$$\nabla_{\mathbf{w}}\mathcal{L} = \frac{1}{K}\sum_{i=1}^{K}\nabla_{\mathbf{w}}\text{hinge}_{y^{(i)}}(x^{(i)}) + \lambda\nabla_{\mathbf{w}}\|\mathbf{w}\|$$

Note that $\text{hinge}_{y^{(i)}}(x^{(i)}) = \max(0, 1 - y^{(i)}(\mathbf{w}^T x^{(i)} + b))$. So the gradient of the hinge loss is:

$$\nabla_{\mathbf{w}}\text{hinge}_{y^{(i)}}(x^{(i)}) = \begin{cases} -y^{(i)}x^{(i)} & \text{if } 1 > y^{(i)}(\mathbf{w}^T x^{(i)} + b) \\ 0 & \text{if } 1 < y^{(i)}(\mathbf{w}^T x^{(i)} + b) \end{cases}$$

And the gradient of the norm is:

$$\nabla_{\mathbf{w}}\|\mathbf{w}\| = \begin{cases} 1 & \text{if } \mathbf{w} > 0 \\ -1 & \text{if } \mathbf{w} < 0 \end{cases}$$

So the gradient of the loss function is:

$$\nabla_{\mathbf{w}}\mathcal{L} = \frac{1}{K}\sum_{i=1}^{K}\begin{cases} -y^{(i)}x^{(i)} & \text{if } 1 > y^{(i)}(\mathbf{w}^T x^{(i)} + b) \\ 0 & \text{if } 1 < y^{(i)}(\mathbf{w}^T x^{(i)} + b) \end{cases} + \lambda\begin{cases} 1 & \text{if } \mathbf{w} > 0 \\ -1 & \text{if } \mathbf{w} < 0 \end{cases}$$

5. Softmax Classifier

Code sections for this part are in `softmax.py` :

```
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 mi
        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]
          that X[i] has label c, where 0 <= c < C.

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

        # Initialize the loss to zero.
        loss = 0.0

        # ================================================================
        # YOUR CODE HERE:
        #   Calculate the normalized softmax loss.  Store it as the variab
        #   (That is, calculate the sum of the losses of all the training
        #   set margins, and then normalize the loss by the number of
        #   training examples.)
        # ================================================================

        # Keep track of the current training example
        i = 0

        # Iterates through each row of X multiplied by the transpose of th
        for row in X.dot(self.W.T):

            # Subtract the max value of the row to prevent overflow when t
            row -= np.max(row)

            # Loss is calculated as -log(exp(row[y[i]]) / sum(exp(row))),
            loss += -np.log(np.exp(row[y[i]]) / sum(np.exp(row)))
            i = i + 1

        # Total loss is divided by the number of examples to get the avera
        loss = loss / y.shape[0]

        # ==================================================================
        # END YOUR CODE HERE
        # ==================================================================

        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.
        """

        # Initialize the loss and gradient to zero.
        loss = 0.0
        grad = np.zeros_like(self.W)

        # ==================================================================
        # YOUR CODE HERE:
        #   Calculate the softmax loss and the gradient. Store the gradien
        #   as the variable grad.
        # ==================================================================

        # Calculate the dot product of W and X transpose
        activations = self.W.dot(X.T)

        # Calculate the element-wise exponential of a
        activations_exp = np.exp(activations)

        # Calculate the Score matrix
        score_matrix = activations_exp / np.sum(activations_exp, axis=0)

        # Subtract 1 from the corresponding element of Score where y=i
        np.subtract.at(score_matrix, (y, range(score_matrix.shape[1])), 1)

        # Calculate the gradient
        grad = np.dot(score_matrix, X)
        grad /= X.shape[0]
        # Calculate the loss
        loss = self.loss(X, y)

        # ================================================================
        # END YOUR CODE HERE
        # ================================================================

        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 outputs as loss_and_grad.
        """
        loss = 0.0
        grad = np.zeros(self.W.shape)  # initialize the gradient as zero
```

```
# ================================================================
# YOUR CODE HERE:
#   Calculate the softmax loss and gradient WITHOUT any for loops.
# ================================================================

# Compute the dot product of X and the transpose of W
activations = X.dot(self.W.T)

# Then subtract the maximum value of each row to prevent numerical
activations = (activations.T - np.amax(activations, axis=1)).T

num_train = y.shape[0]

# Compute the softmax scores for each sample
activations_exp = np.exp(activations)
score_matrix = np.zeros_like(activations_exp)
score_matrix = activations_exp / np.sum(activations_exp, axis=1, k


epsilon = 1e-7

# Compute the loss
loss = np.sum(
    -np.log(
        activations_exp[np.arange(activations.shape[0]), y]
        / (np.sum(activations_exp, axis=1) + epsilon)
    )
)

# Compute the gradient
score_matrix[range(num_train), y] -= 1
gradient_wrt_activations = score_matrix
grad = gradient_wrt_activations.T.dot(X)
grad /= num_train

# Average the loss and gradient over the number of training sample
loss = loss / num_train

# ================================================================
# END YOUR CODE HERE
# ================================================================
        return loss, grad

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

        Inputs:
        - X: A numpy array of shape (N, D) containing training data; there
          training samples each of dimension D.
        - y: A numpy array of shape (N,) containing training labels; y[i]
          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
        - verbose: (boolean) If true, print progress during optimization.

        Outputs:
        A list containing the value of the loss function at each training
        """
        num_train, dim = X.shape
        num_classes = (
            np.max(y) + 1
        )  # assume y takes values 0...K-1 where K is number of classes

        self.init_weights(
            dims=[np.max(y) + 1, X.shape[1]]
        )  # initializes the weights of self.W

        # Run stochastic gradient descent to optimize W
        loss_history = []

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

            # ==============================================================
            # YOUR CODE HERE:
            #   Sample batch_size elements from the training data for use
            #     gradient descent.  After sampling,
            #     - X_batch should have shape: (batch_size, dim)
            #     - y_batch should have shape: (batch_size,)
            #   The indices should be randomly generated to reduce correla
            #   in the dataset.  Use np.random.choice.  It's okay to sampl
            #   replacement.
            # ============================================================

            # Randomly select a batch of training examples to update the w
            index = np.random.choice(np.arange(num_train), batch_size)
            X_batch = X[index]
            y_batch = y[index]

            # ============================================================
            # END YOUR CODE HERE
            # ============================================================

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

            # ============================================================
            # YOUR CODE HERE:
            #   Update the parameters, self.W, with a gradient step
            # ============================================================

            # Update the weights using the calculated gradient
            self.W = self.W - grad * learning_rate

            # ============================================================
            # END YOUR CODE HERE
            # ============================================================

            if verbose and it % 100 == 0:
                print("iteration {} / {}: loss {}".format(it, num_iters, l

        return loss_history

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

```
Returns:
- y_pred: Predicted labels for the data in X. y_pred is a 1-dimens
  array of length N, and each element is an integer giving the pre
  class.
"""
y_pred = np.zeros(X.shape[1])
# ================================================================
# YOUR CODE HERE:
#   Predict the labels given the training data.
# ================================================================

# Compute the scores for each sample
scores = X.dot(self.W.T)

# Take the class with the highest score as the prediction
y_pred = np.argmax(scores, axis=1)

# ================================================================
# END YOUR CODE HERE
# ================================================================

return y_pred
```

(Attached the softmax_nosol workbook below knn_nosol workbook)

knn_nosol

January 29, 2023

0.1 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.

0.2 Import the appropriate libraries

[61]:
```
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.

# 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
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

[62]:
```
# Set the path to the CIFAR-10 data
cifar10_dir = "/Users/mylesthemonster/Documents/ece_c247/hw2/hw2_code/
  ↪cifar-10-batches-py"
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,)
```

[63]:
```
# 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()
```

[64]:
```
# 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)
```

1 K-nearest neighbors

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

[65]:
```
# Import the KNN class
from nndl import KNN
```

[66]:
```
# 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)
```

1.1 Questions

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

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

1.2 Answers

(1) The inside the knn.train() function looks like:

```
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
```

All this being done is that the training data is being stored in the class.

(2) The pros of this training step are that it is simple. A con of this training step is that it will take up memory.

1.3 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.

[67]:
```
# 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: 14.396498918533325
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

1.3.1 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.

[68]:
```
# 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.norm(dists_L2 - dists_L2_vectorized, "fro")
    )
)
```

```
Time to run code: 0.06827902793884277
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.

1.3.2 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.

[69]:
```
# 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
```

```
error = 1

# ================================================================ #
# YOUR CODE HERE:
#   Calculate the error rate by calling predict_labels on the test
#   data with k = 1.  Store the error rate in the variable error.
# ================================================================ #
yPredicted = knn.predict_labels(dists_L2_vectorized, 1)
error = np.count_nonzero(y_test - yPredicted) / float(len(y_test))
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

print(error)
```

```
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%.

2 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.

2.0.1 Create training and validation folds

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

[70]:
```
# Create the dataset folds for cross-validation.
num_folds = 5

X_train_folds = []
y_train_folds = []

# ================================================================ #
# YOUR CODE HERE:
#   Split the training data into num_folds (i.e., 5) folds.
#   X_train_folds is a list, where X_train_folds[i] contains the
#      data points in fold i.
#   y_train_folds is also a list, where y_train_folds[i] contains
#      the corresponding labels for the data in X_train_folds[i]
# ================================================================ #

# Calculate the size of each fold
n = X_train.shape[0]
```

```
# Divide the number of examples by the number of folds
fold_size = int(n / num_folds)

# Iterate over the number of folds
for i in range(num_folds):

    # Append each fold of the training data to the X_train_folds list
    X_train_folds.append(X_train[i * fold_size : i * fold_size + fold_size])

    # Append each fold of the corresponding labels to the y_train_folds list
    y_train_folds.append(y_train[i * fold_size : i * fold_size + fold_size])

print("==>> y_train.shape: ", y_train.shape)
print("==>> X_train.shape: ", X_train.shape)
print("==>> y_train_folds[0].shape: ", y_train_folds[0].shape)
print("==>> X_train_folds[0].shape: ", X_train_folds[0].shape)
print("==>> Labels in each fold: ", y_train_folds[0].shape[0])
print("==>> Training data entries in each fold: ", X_train_folds[0].shape[0])

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
==>> y_train.shape:  (5000,)
==>> X_train.shape:  (5000, 3072)
==>> y_train_folds[0].shape:  (1000,)
==>> X_train_folds[0].shape:  (1000, 3072)
==>> Labels in each fold:  1000
==>> Training data entries in each fold:  1000
```

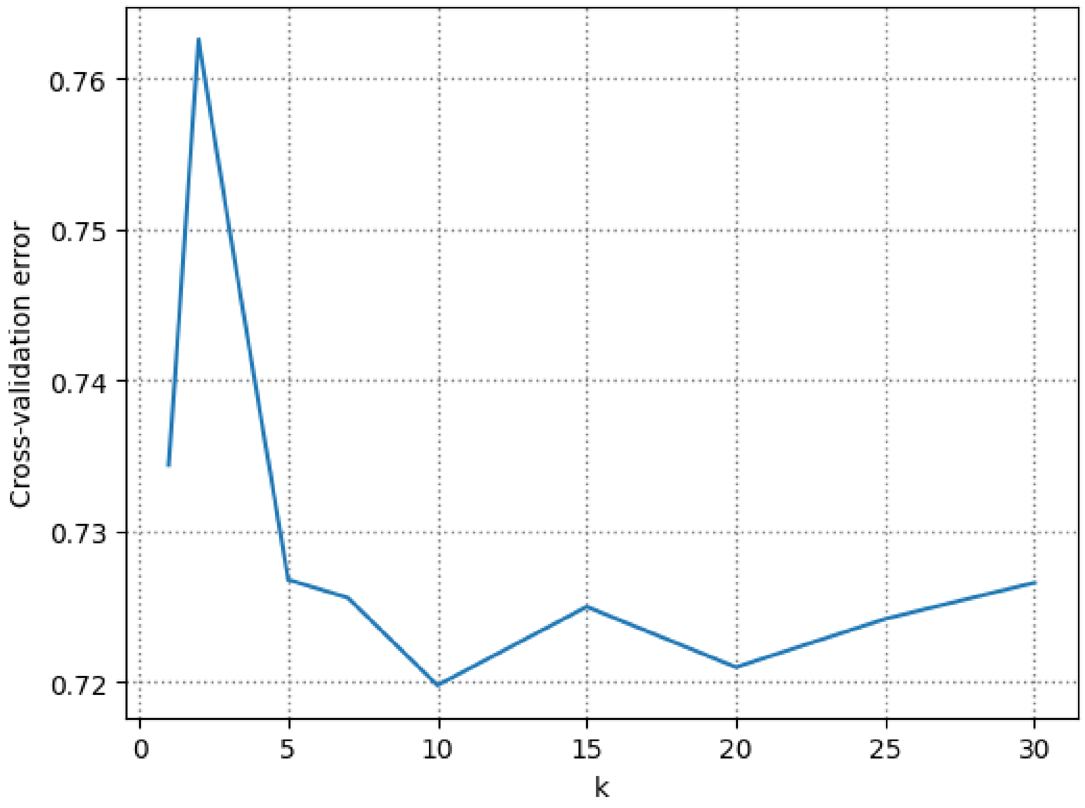
2.0.2 Optimizing the number of nearest neighbors hyperparameter.

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

[75]:
```
time_start = time.time()

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

# ================================================================ #
# YOUR CODE HERE:
#   Calculate the cross-validation error for each k in ks, testing
#   the trained model on each of the 5 folds.  Average these errors
#   together and make a plot of k vs. cross-validation error. Since
#   we are assuming L2 distance here, please use the vectorized code!
#   Otherwise, you might be waiting a long time.
# ================================================================ #
# List to store the average cross-validation error for each k
avr_cross_val_err = []
entries_per_fold = y_train_folds[0].shape[0]

# Loop through each k
for k in ks:
    total_error = 0

    # Loop through each fold
    for i in range(num_folds):
        # Declare an instance of the knn class.
        knn = KNN()

        # Create the training and testing sets for the current fold
        X_test_fold = X_train_folds[i]
        y_test_fold = y_train_folds[i]

        X_train_fold = np.concatenate(X_train_folds[:i] + X_train_folds[i + 1 :
 ↪])
        y_train_fold = np.concatenate(y_train_folds[:i] + y_train_folds[i + 1 :
 ↪])

        # Train the model on the current training set
        knn.train(X=X_train_fold, y=y_train_fold)

        # Compute the L2 distances and predict the labels using the current␣
 ↪value of k
        dists_fold = knn.compute_L2_distances_vectorized(X_test_fold)
        y_est_fold = knn.predict_labels(dists_fold, k)

        # Calculate the number of correct predictions
        total_correct = np.sum(y_test_fold == y_est_fold)

        # Calculate the error for the current fold
        error = (entries_per_fold - total_correct) / entries_per_fold

        # Add the error for the current fold to the total error
        total_error += error

    # Append the average error for the current value of k to the list of errors
    avr_cross_val_err.append(total_error / num_folds)

index_min_error = np.argmin(avr_cross_val_err)
print(f"The optimal k is k = {ks[index_min_error]}, with a cross-validation␣
 ↪error of {avr_cross_val_err[index_min_error]}")
```

```
# Plot k vs. Cross-validation Error
plt.plot(ks, avr_cross_val_err)
plt.xlabel("k")
plt.ylabel("Cross-validation error")
plt.grid(color="grey", linestyle=":", linewidth=1)
plt.show()

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

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

```
The optimal k is k = 10, with a cross-validation error of 0.7198
```



```
Computation time: 14.99
```

2.1 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?

2.2 Answers:

(1) The best value of k amongst the tested k's is $k = 10$

(2) The cross-validation error for $k = 10$ is 0.7198

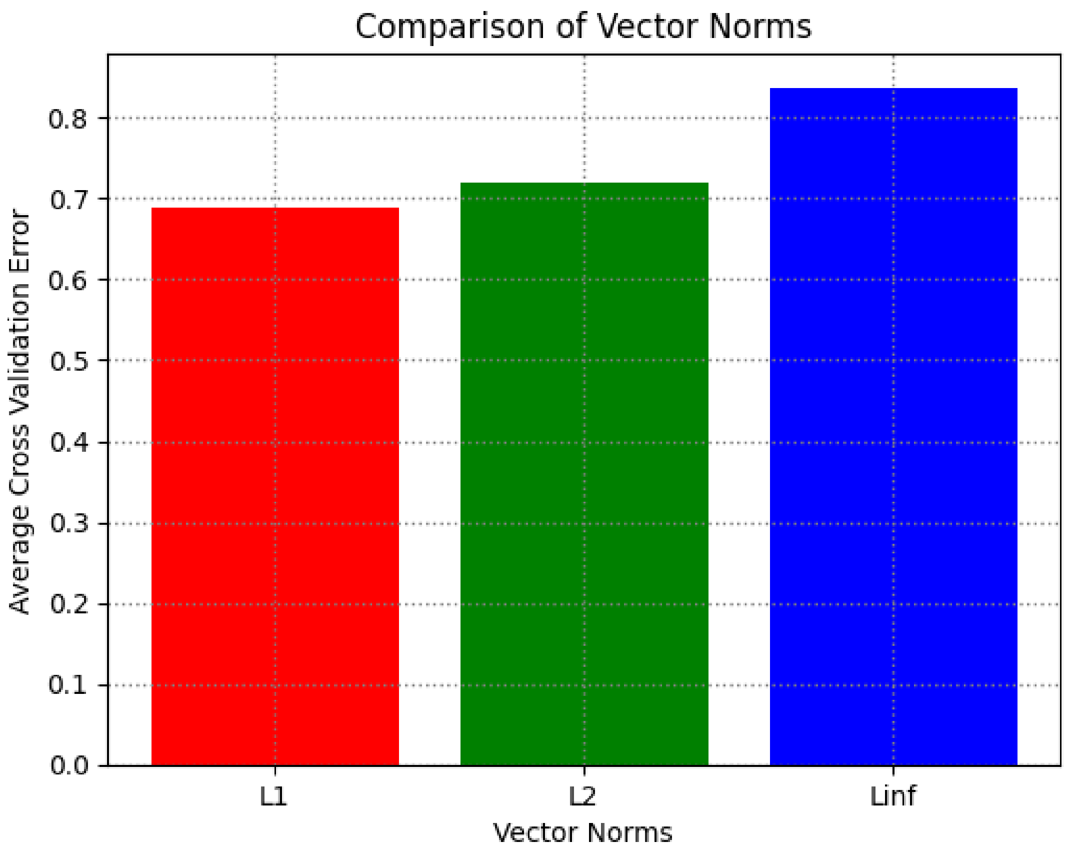
2.2.1 Optimizing the norm

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

[72]:
```
time_start = time.time()

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]

# ================================================================ #
# YOUR CODE HERE:
#   Calculate the cross-validation error for each norm in norms, testing
#   the trained model on each of the 5 folds.  Average these errors
#   together and make a plot of the norm used vs the cross-validation error
#   Use the best cross-validation k from the previous part.
#
#   Feel free to use the compute_distances function.  We're testing just
#   three norms, but be advised that this could still take some time.
#   You're welcome to write a vectorized form of the L1- and Linf- norms
#   to speed this up, but it is not necessary.
# ================================================================ #

# Vectorized form of the L1- and Linf- norms
Vec_L1_norm = lambda x: np.sum(np.abs(x))
Vec_L2_norm = lambda x: np.sqrt(np.sum(x**2))
Vec_Linf_norm = lambda x: np.max(np.abs(x))
vec_norms = [Vec_L1_norm, Vec_L2_norm, Vec_Linf_norm]
vec_norms_names = ["L1", "L2", "Linf"]

# List to store the average cross-validation error for each norm
avr_cross_val_err = []
entries_per_fold = y_train_folds[0].shape[0]

# Best k from the previous part
k = 10

# Iterate over each norm
for l in vec_norms:
    total_error = 0
    # Iterate over each fold
    for i in range(num_folds):
        # Initialize KNN classifier
        knn = KNN()

        # Create the training and testing sets for the current fold
        X_test_fold = X_train_folds[i]
        y_test_fold = y_train_folds[i]

        X_train_fold = np.concatenate(X_train_folds[:i] + X_train_folds[i + 1 :
↪])
        y_train_fold = np.concatenate(y_train_folds[:i] + y_train_folds[i + 1 :
↪])

        # Train the model on the current training set
        knn.train(X=X_train_fold, y=y_train_fold)

        # Compute the distances between the test data and the train data using␣
↪the current norm
        dists_fold = knn.compute_distances(X_test_fold, l)

        # Predict the labels for the test data using the current norm
        y_est_fold = knn.predict_labels(dists_fold, k)

        # Calculate the error for the current fold
        y_diff_fold = y_test_fold - y_est_fold

        # Calculate the number of correct predictions
        total_correct = np.sum(y_test_fold == y_est_fold)

        # Calculate the error for the current fold
        error = (entries_per_fold - total_correct) / entries_per_fold

        # Add the error for the current fold to the total error
        total_error += error

    # Append the average error for the current value of k to the list of errors
    avr_cross_val_err.append(total_error / num_folds)

# Print the errors for each norm
for j in np.arange(len(avr_cross_val_err)):
    print(
        f"For the {vec_norms_names[j]} vectorized norm , the cross-validation␣
↪error is {avr_cross_val_err[j]}"
    )

# Plot the error vs the norm used
```

```
plt.figure()
plt.bar(vec_norms_names, avr_cross_val_err, color=['red', 'green', 'blue'])
plt.xlabel("Vector Norms")
plt.ylabel("Average Cross Validation Error")
plt.title("Comparison of Vector Norms")
plt.grid(color="grey", linestyle=":", linewidth=1)
plt.show()

# =============================================================== #
# END YOUR CODE HERE
# =============================================================== #
print("Computation time: %.2f" % (time.time() - time_start))
```

```
For the L1 vectorized norm , the cross-validation error is 0.6886000000000001
For the L2 vectorized norm , the cross-validation error is 0.7198
For the Linf vectorized norm , the cross-validation error is 0.8370000000000001
```



```
Computation time: 311.81
```

2.3 Questions:

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

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

2.4 Answers:

(1) The $L1$ norm has the best cross-validation error

(2) the cross-validation error for the $L1$ norm and $k = 10$ is 0.6886000000000001

3 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.

[73]:
```
error = 1

# ================================================================ #
# YOUR CODE HERE:
#   Evaluate the testing error of the k-nearest neighbors classifier
#   for your optimal hyperparameters found by 5-fold cross-validation.
# ================================================================ #

knn = KNN()
knn.train(X=X_train, y=y_train)

dists_L1 = knn.compute_distances(X_test, Vec_L1_norm)
y_pred = knn.predict_labels(dists_L1, k)
error = np.count_nonzero(y_test - y_pred) / float(len(y_test))

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

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

```
Error rate achieved: 0.722
```

3.1 Question:

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

3.2 Answer:

My error from by cross-validation with $k = 1$ and using the $L2$-norm was 0.726 and my error from cross-validation with the optimal $k = 10$ and $L1$-norm is 0.722. This means that my error improved by 0.004.

softmax_nosol

January 29, 2023

0.1 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.

[134]:
```
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
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

[135]:
```
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/mylesthemonster/Documents/ece_c247/hw2/hw2_code/
↪cifar-10-batches-py"
    )
    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,)
```

0.2 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.

[136]:
```
from nndl import Softmax
```

[137]:
```
# 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

[138]:
```
## Implement the loss function of the softmax using a for loop over
#  the number of examples

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

[139]:
```
print(loss)
```

```
2.327760702804897
```

0.3 Question:

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

0.4 Answer:

The loss from the softmax being 2.327760702804897 makes sense because the softmax is a multiclass classifier. The loss is the average of the loss for each class. Since there are 10 classes, the loss for each class is $\frac{2.3}{10} = 0.23$. The loss for each class is the negative log of the probability of the correct class. Since the probability of the correct class is $\frac{1}{10}$, the loss for each class is $-\log(\frac{1}{10}) = 2.3$.

Softmax gradient

[140]:
```
## 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 correctly.
softmax.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: -1.265198 analytic: -1.265198, relative error: 2.288884e-08
numerical: 0.118820 analytic: 0.118820, relative error: 1.340117e-07
numerical: 0.345094 analytic: 0.345094, relative error: 8.097804e-09
numerical: 1.080327 analytic: 1.080327, relative error: 1.824787e-08
numerical: -1.255915 analytic: -1.255915, relative error: 4.078140e-08
numerical: 1.819504 analytic: 1.819504, relative error: 6.604489e-09
numerical: -0.849827 analytic: -0.849827, relative error: 6.472480e-08
numerical: 0.237496 analytic: 0.237496, relative error: 2.136340e-08
numerical: 0.673580 analytic: 0.673580, relative error: 6.425449e-09
numerical: -2.647151 analytic: -2.647151, relative error: 1.810721e-08
```

0.5 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.

[141]:
```
import time
```

[142]:
```
## 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 - tic))

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_vectorized, 'fro'), toc - tic))

# 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_vectorized)))

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

```
Normal loss / grad_norm: 2.329977339069047 / 341.4968789453495 computed in
0.006153821945190430s
Vectorized loss / grad: 2.3299773550631815 / 341.4968789453495 computed in
0.0028061866760253906s
difference in loss / grad: -1.5994134461294607e-08 /1.0698549924945794e-13
```

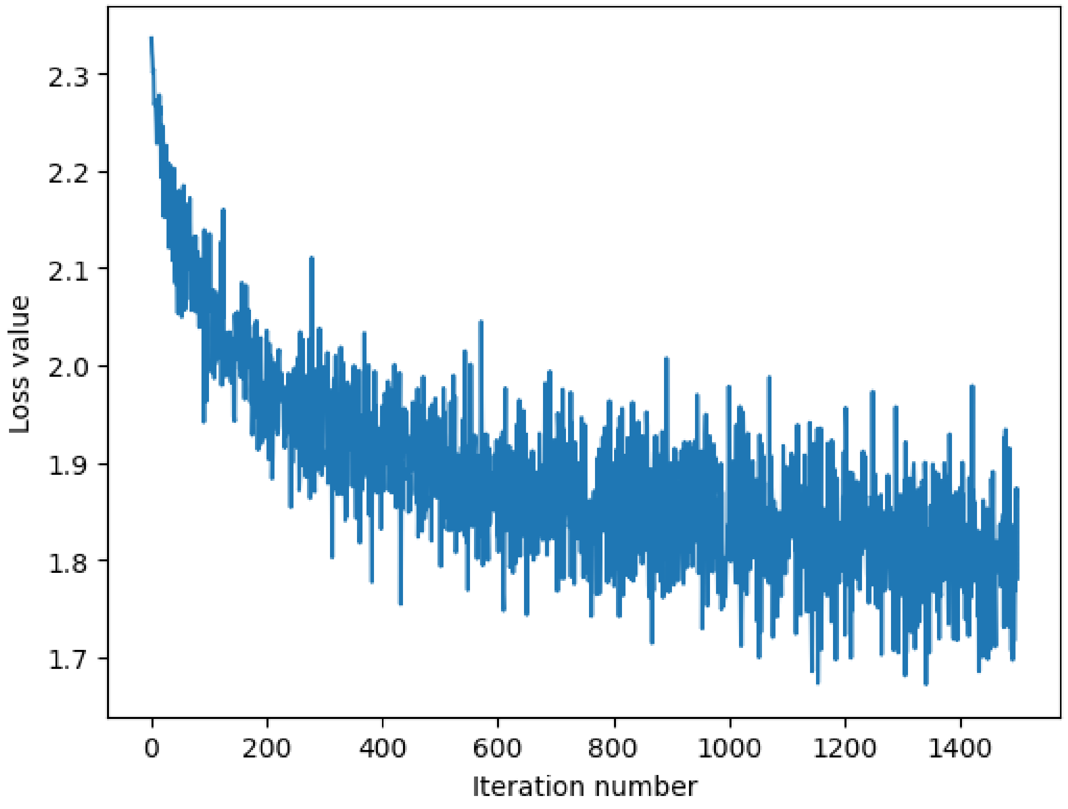
0.6 Stochastic gradient descent

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).

[143]:
```
# 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 2.3365926768353664
iteration 100 / 1500: loss 2.055722282615281
iteration 200 / 1500: loss 2.0357745361814166
iteration 300 / 1500: loss 1.9813348420788213
iteration 400 / 1500: loss 1.9583142707532786
iteration 500 / 1500: loss 1.8622653355158354
iteration 600 / 1500: loss 1.8532611744112568
iteration 700 / 1500: loss 1.8353062499718755
iteration 800 / 1500: loss 1.829389275758732
iteration 900 / 1500: loss 1.8992158822347505
iteration 1000 / 1500: loss 1.9783503842474557
iteration 1100 / 1500: loss 1.8470798201432712
iteration 1200 / 1500: loss 1.8411450584382825
iteration 1300 / 1500: loss 1.7910402816542166
iteration 1400 / 1500: loss 1.8705803352042043
```

```
That took 1.916717767715454s
```



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

[144]:
```python
## 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
```

0.7 Optimize the softmax classifier

[145]:
```
np.finfo(float).eps
```

[145]:
```
2.220446049250313e-16
```

[146]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Train the Softmax classifier with different learning rates and
#     evaluate on the validation data.
#   Report:
#     - The best learning rate of the ones you tested.
#     - The best validation accuracy corresponding to the best validation error.
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
# ================================================================ #

# Define a list of learning rates to test
learning_rates = [10**i for i in range(-15, -1)]

# Initialize variables to keep track of the best learning rate and its␣
 ↪corresponding accuracy
best_rate, best_val_accuracy, best_test_accuracy = 0, 0, 0

# Iterate over all learning rates
for rate in learning_rates:

    # Train the classifier with the current learning rate
    softmax.train(X_train, y_train, learning_rate=rate, num_iters=1500,␣
 ↪verbose=False)

    # Predict the labels of the validation set
    y_val_pred = softmax.predict(X_val)

    # Calculate the accuracy of the predictions
    val_accuracy = np.mean(np.equal(y_val, y_val_pred))

    # Print the current learning rate and its corresponding accuracy
    print(f"Learning rate: {rate}, Validation accuracy: {val_accuracy}")
    # If the current accuracy is better than the best accuracy so far, update␣
  ↪the best accuracy
    if val_accuracy > best_val_accuracy:
        best_rate = rate
        best_val_accuracy = val_accuracy

# Re-train the classifier with the best learning rate
softmax.train(X_train, y_train, learning_rate=best_rate, num_iters=1500,␣
  ↪verbose=False)

# Predict the labels of the test set
y_test_pred = softmax.predict(X_test)

# Calculate the accuracy of the predictions
best_test_accuracy = np.mean(np.equal(y_test, y_test_pred))

# Print the best learning rate, test accuracy and test error rate
print(
    f"\nThe Best learning rate is {best_rate} which has a Test accuracy of␣
  ↪{best_test_accuracy}, and a Test error rate of {1.0 - best_test_accuracy}"
)

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
Learning rate: 1e-15, Validation accuracy: 0.122
Learning rate: 1e-14, Validation accuracy: 0.049
Learning rate: 1e-13, Validation accuracy: 0.068
Learning rate: 1e-12, Validation accuracy: 0.075
Learning rate: 1e-11, Validation accuracy: 0.083
Learning rate: 1e-10, Validation accuracy: 0.08
Learning rate: 1e-09, Validation accuracy: 0.178
Learning rate: 1e-08, Validation accuracy: 0.313
Learning rate: 1e-07, Validation accuracy: 0.392
Learning rate: 1e-06, Validation accuracy: 0.415
Learning rate: 1e-05, Validation accuracy: 0.317
Learning rate: 0.0001, Validation accuracy: 0.278
Learning rate: 0.001, Validation accuracy: 0.305
Learning rate: 0.01, Validation accuracy: 0.243

The Best learning rate is 1e-06 which has a Test accuracy of 0.399, and a Test
error rate of 0.601
```