Introduction

In this assignment you will practice putting together a simple image classification pipeline with both non-parametric and parametric methods.

In paticular, we will work with the k-Nearest Neighbor, the SVM classifier and the 2-Layered Neural Network for CIFAR-10 dataset. The goals of this assignment are as follows:

- Understand the basic Image Classification pipeline and the data-driven approach (train/predict stages).
- Understand the train/val/test splits and the use of validation data for hyperparameter tuning.
- Implement and apply a Weighted k-Nearest Neighbor (kNN) classifier.
- Implement and apply a Multiclass Support Vector Machine (SVM) classifier.
- Implement and apply a 2-layered Neural Network.
- Understand the differences and tradeoffs between these classifiers.

Please fill in all the **TODO** code blocks. Once you are ready to submit:

- Export the notebook `CSCI677_spring25_assignment_2.ipynb` as a PDF `[Your USC ID]_CSCI677_spring25_assignment_2.pdf`
- Submit your PDF file through Brightspace.

Please make sure that the notebook have been run before exporting PDF, and your code and all cell outputs are visible in the your submitted PDF. Regrading request will not be accepted if your code/output is not visible in the original submission. Thank you!

Data Preparation

CIFAR-10 is a well known dataset composed of 60,000 colored 32x32 images. The utility function `cifar10()` returns the entire CIFAR-10 dataset as a set of four Torch tensors:

- `x_train` contains all training images (real numbers in the range [0,1])
- `y_train` contains all training labels (integers in the range [0,9])
- `x_test` contains all test images
- `y_test` contains all test labels

This function automatically downloads the CIFAR-10 dataset the first time you run it.

```
import os
import time
import torch
import numpy as np
from torchvision.datasets import CIFAR10
```

```
import random
import matplotlib.pyplot as plt

def _extract_tensors(dset, num=None):
    x = torch.tensor(dset.data, dtype=torch.float32).permute(0, 3, 1,
2).div_(255)
    y = torch.tensor(dset.targets, dtype=torch.int64)
    if num is not None:
        if num <= 0 or num > x.shape[0]:
          raise ValueError('Invalid value num=%d; must be in the range
[0, %d]'
                              % (num, x.shape[0]))
        x = x[:num].clone()
        y = y[:num].clone()
    return x, y

def cifar10(num_train=None, num_test=None):
    download = not os.path.isdir('cifar-10-batches-py')
    dset_train = CIFAR10(root='.', download=download, train=True)
    dset_test = CIFAR10(root='.', train=False)
    x_train, y_train = _extract_tensors(dset_train, num_train)
    x_test, y_test = _extract_tensors(dset_test, num_test)

    return x_train, y_train, x_test, y_test
```

Our data is going to be stored simply in the four variables: `x_train`, `x_test`, `y_train`, and `y_test`.

- Training set: `x_train` is composed of 50,000 images where `y_train` references the corresponding labels.
- Testing set: `x_test` is composed of 10,000 images where `y_test` references the corresponding labels.

```
torch.manual_seed(0)
num_train = 50000
num_test = 5000
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog',
'horse', 'ship', 'truck']

x_train, y_train, x_test, y_test = cifar10(num_train, num_test)

# Split the data into train, val, and test sets. In addition we will
# create a small development set as a subset of the training data;
# we can use this for development so our code runs faster.
num_training = 49000
num_validation = 1000
num_test = 1000
num_dev = 500

x_train_np = x_train.numpy()
```

```python
y_train_np = y_train.numpy()
x_test_np = x_test.numpy()
y_test_np = y_test.numpy()

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

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

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

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

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

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

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

```
# third: append the bias dimension of ones (i.e. bias trick) so that
our SVM
# only has to worry about optimizing a single weight matrix W.
X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

X_train, X_test, X_dev, X_val = torch.FloatTensor(X_train),
torch.FloatTensor(X_test), torch.FloatTensor(X_dev),
torch.FloatTensor(X_val)
y_train, y_test, y_dev, y_val = torch.LongTensor(y_train),
torch.LongTensor(y_test), torch.LongTensor(y_dev),
torch.LongTensor(y_val)
print(X_train.shape, X_val.shape, X_test.shape, X_dev.shape)
```

```
Training data shape:  (49000, 3072)
Validation data shape:  (1000, 3072)
Test data shape:  (1000, 3072)
dev data shape:  (500, 3072)
torch.Size([49000, 3073]) torch.Size([1000, 3073]) torch.Size([1000,
3073]) torch.Size([500, 3073])
```

k-Nearest Neighbor (kNN) (20 pts)

Subsampling

When implementing machine learning algorithms, it's usually a good idea to use a small sample of the full dataset. This way your code will run much faster, allowing for more interactive and efficient development. Once you are satisfied that you have correctly implemented the algorithm, you can then rerun with the entire dataset.

```
# Subsample size
def subsample(X, y, n):
    assert len(X) == len(y)
    indices = torch.randint(len(X), (n,))
    return X[indices],  y[indices]
ss_x_train, ss_y_train = subsample(X_train, y_train, 500)
print(ss_x_train.shape, ss_y_train.shape)
```

```
torch.Size([500, 3073]) torch.Size([500])
```

Compute Distance (5 pts)

Now that we have examined and prepared our data, it is time to implement the Weighted-kNN classifier. We can break the process down into two steps:

1. Compute the consine similarities between all training examples and all test examples
2. Given these pre-computed similarities, for each test example find its k nearest neighbors and have them vote for the label to output

NOTE: When implementing algorithms in PyTorch, it's best to avoid loops in Python if possible. Instead it is preferable to implement your computation so that all loops happen inside PyTorch functions. This will usually be much faster than writing your own loops in Python, since PyTorch functions can be internally optimized to iterate efficiently, possibly using multiple threads. This is especially important when using a GPU to accelerate your code.

```
def compute_distances(x_train, x_test):
    """
    Inputs:
    x_train: shape (num_train, C, H, W) tensor.
    x_test: shape (num_test, C, H, W) tensor.

    Returns:
    dists: shape (num_train, num_test) tensor where dists[j, i] is the
        cosine similarity between the ith training image and the jth
test
        image.
    """

    # Get the number of training and testing images
    num_train = x_train.shape[0]
    num_test = x_test.shape[0]

    # dists will be the tensor housing all distance measurements
between testing and training
    dists = x_train.new_zeros(num_train, num_test)

    # Flatten tensors
    train = x_train.flatten(1)
    test = x_test.flatten(1)


######################################################################
#
    # TODO (5 pts):
    # find the consine similarities between testing and training
images,
    # and save the computed distance in dists.

######################################################################
#
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    # Compute dot product between train and test samples
    dot_product = train @ test.T  # (num_train, num_test)

    # Compute the norms (L2 norms)
    train_norms = torch.norm(train, dim=1, keepdim=True)  # 
(num_train, 1)
    test_norms = torch.norm(test, dim=1, keepdim=True)    # (num_test, 
1)

    # Compute cosine similarity
    dists = dot_product / (train_norms * test_norms.T)  # (num_train, 
num_test)

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

    return dists
```

Implement Weighted-kNN (10 pts)

The Weighted-kNN classifier consists of two stages:

- Training: the classifier takes the training data and simply remembers it
- Testing: For each test sample, the classifier computes the similarity to all training samples and selects the k most similar neighbors. Instead of simple majority voting, each neighbor contributes to the final prediction based on its similarity with the test sample. This ensures that more similar neighbors have a greater influence on the classification decision.

```
from collections import defaultdict

class KnnClassifier:
    def __init__(self, x_train, y_train):
        """
        x_train: shape (num_train, C, H, W) tensor where num_train is 
batch size,
          C is channel size, H is height, and W is width.
        y_train: shape (num_train) tensor where num_train is batch 
size providing labels
        """

        self.x_train = x_train
        self.y_train = y_train

    def predict(self, x_test, k=1):
        """
        x_test: shape (num_test, C, H, W) tensor where num_test is 
batch size,
          C is channel size, H is height, and W is width.
        k: The number of neighbors to use for prediction
        """

        # Init output shape
        y_test_pred = torch.zeros(x_test.shape[0], dtype=torch.int64)
        # Find & store Euclidean distance between test & train
        dists = compute_distances(self.x_train, x_test)


######################################################################
#
        # TODO (10 pts):
        # The goal is to return a tensor y_test_pred where the ith
index
        # is the assigned label to ith test image by the kNN
algorithm.
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS
LINE)*****
        # 1. Index over test images

        # 2. Find the indices of the k most similar training samples
(highest cosine similarity).

        # 3. Retrieve the labels of these k neighbors and compute
their contributions as the similarity scores

        # 4. Assign the label with the highest accumulated weight as
the final prediction.

######################################################################
#
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS
LINE)*****
        # 1. Index over test images
        for i in range(x_test.shape[0]):
            # 2. Find the indices of the k most similar training
samples (highest cosine similarity).
            top_k_indices = torch.argsort(dists[:, i],
descending=True)[:k]

            # 3. Retrieve the labels of these k neighbors and compute
their contributions as the similarity scores
            top_k_labels = self.y_train[top_k_indices]
            top_k_weights = dists[top_k_indices, i]  # Use similarity
as weight

            # 4. Assign the label with the highest accumulated weight
as the final prediction.
            label_weights = defaultdict(float)
            for label, weight in zip(top_k_labels, top_k_weights):
                label_weights[label.item()] += weight.item()

            y_test_pred[i] = max(label_weights, key=label_weights.get)

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

    def check_accuracy(self, x_test, y_test, k=1, quiet=False):
        """
        x_test: shape (num_test, C, H, W) tensor where num_test is
batch size,
          C is channel size, H is height, and W is width.
        y_test: shape (num_test) tensor where num_test is batch size
providing labels
        k: The number of neighbors to use for prediction
        quiet: If True, don't print a message.

        Returns:
        accuracy: Accuracy of this classifier on the test data, as a
percent.
          Python float in the range [0, 100]
        """

        y_test_pred = self.predict(x_test, k=k)
        num_samples = x_test.shape[0]
        num_correct = (y_test == y_test_pred).sum().item()
        accuracy = 100.0 * num_correct / num_samples
        msg = (f'Got {num_correct} / {num_samples} correct; '
              f'accuracy is {accuracy:.2f}%')
        if not quiet:
          print(msg)
        return accuracy
```

We've finished implementing kNN and can begin testing the algorithm on larger portions of the dataset to see how well it performs.

```
torch.manual_seed(0)
num_train = 5000
num_test = 500
num_val = 500
knn_x_train, knn_y_train = subsample(X_train, y_train, num_train)
knn_x_test, knn_y_test = subsample(X_test, y_test, num_test)
knn_x_val, knn_y_val = subsample(X_val, y_val, num_val)
classifier = KnnClassifier(knn_x_train, knn_y_train)
classifier.check_accuracy(knn_x_test, knn_y_test, k=5)
```

```
Got 168 / 500 correct; accuracy is 33.60%
```

```
33.6
```

Hyperparameter Tuning (5 pts)

Now we use the validation set to tune hyperparameters (number of nearest neighbors k). You should experiment with different ranges of k.

```
results = {}
best_val = -1   # The highest validation accuracy that we have seen so
far.
best_k = None # The value of k that achieved the highest validation
rate.

######################################################################
##########
# TODO (5 pts):
#
# Write code that chooses the best k value by tuning on the validation
#
# set. For each value of k, train a KnnClassifier on the
#
# training set, compute its accuracy on the training and validation
sets, and  #
# store these numbers in the results dictionary. In addition, store
the best   #
# validation accuracy in best_val and the best value of k in best_k.
#
######################################################################
##########
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
# fill in your own values
k_choices = [1, 3, 5, 7, 9, 15, 20, 50]
for k in k_choices:
    # Train classifier with current k
    classifier = KnnClassifier(knn_x_train, knn_y_train)

    # Compute accuracy on validation set
    val_accuracy = classifier.check_accuracy(knn_x_val, knn_y_val,
k=k, quiet=True)

    # Store results
    results[k] = val_accuracy

    # Track best validation accuracy
    if val_accuracy > best_val:
        best_val = val_accuracy
        best_k = k

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

# Print out results.
for k in sorted(results):
    val_accuracy = results[k]
    print('k %d val accuracy: %f' % (
                k, val_accuracy))

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

```
classifier = KnnClassifier(knn_x_train, knn_y_train)
test_acc = classifier.check_accuracy(knn_x_test, knn_y_test, k=best_k)
print('final test accuracy knn achieved: %f' % test_acc)
```

```
k 1 val accuracy: 28.600000
k 3 val accuracy: 30.600000
k 5 val accuracy: 34.200000
k 7 val accuracy: 34.200000
k 9 val accuracy: 33.800000
k 15 val accuracy: 33.400000
k 20 val accuracy: 33.600000
k 50 val accuracy: 34.200000
best validation accuracy achieved: 34.200000
Got 168 / 500 correct; accuracy is 33.60%
final test accuracy knn achieved: 33.600000
```

Define a General Classifier Class (15 pts)

Before implementing Support Vector Machine (SVM) Classifier. We define a general classifier class that contains the following main functions:

1. `train`: train this linear classifier using stochastic gradient descent.
2. `predict`: use the trained weights of this linear classifier to predict labels for data points.
3. `loss`: compute the loss function and its derivative.

We will define SVM and Softmax classifier as subclasses of this general linear classifier class. Subclasses will override the `loss` function.

```
class LinearClassifier(object):
    def __init__(self):
        self.W = None

    def train(
        self,
        X,
        y,
        learning_rate=1e-3,
        reg=1e-5,
        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.
        - reg: (float) regularization strength.
        - 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
        )  # assume y takes values 0...K-1 where K is number of
classes
        if self.W is None:
            # lazily initialize W
            self.W = 0.001 * np.random.randn(dim, num_classes)

        # Run stochastic gradient descent to optimize W
        loss_history = []
        for it in range(num_iters):
            X_batch = None
            y_batch = None


######################################################################
###
            # TODO (5 pts):
#
            # Sample batch_size elements from the training data and
their           #
            # corresponding labels to use in this round of gradient
descent.        #
            # Store the data in X_batch and their corresponding labels
in          #
            # y_batch; after sampling X_batch should have shape
(batch_size, dim)   #
            # and y_batch should have shape (batch_size,)
#

######################################################################
```

```
###
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS
LINE)*****
            batch_indices = np.random.choice(num_train, batch_size,
replace=True)
            X_batch = X[batch_indices]
            y_batch = y[batch_indices]
            # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS
LINE)*****

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

            # perform parameter update

######################################################################
###
            # TODO (5 pts):
#
            # Update the weights using the gradient and the learning
rate.          #

######################################################################
###
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS
LINE)*****
            self.W -= learning_rate * grad
            # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS
LINE)*****

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

        return loss_history

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

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

        Returns:
        - y_pred: Predicted labels for the data in X. y_pred is a 1-
```

```
dimensional
          array of length N, and each element is an integer giving the
predicted
          class.
        """
        y_pred = np.zeros(X.shape[0])

######################################################################
#####
        # TODO (5 pts):
#
        # Implement this method. Store the predicted labels in y_pred.
#

######################################################################
#####
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS
LINE)*****
        scores = X @ self.W  # Compute class scores
        y_pred = np.argmax(scores, axis=1)  # Select class with
highest score

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

    def loss(self, X_batch, y_batch, reg):
        """
        Compute the loss function and its derivative.
        Subclasses will override this.

        Inputs:
        - X_batch: A numpy array of shape (N, D) containing a
minibatch of N
          data points; each point has dimension D.
        - y_batch: A numpy array of shape (N,) containing labels for
the minibatch.
        - reg: (float) regularization strength.

        Returns: A tuple containing:
        - loss as a single float
        - gradient with respect to self.W; an array of the same shape
as W
        """
        pass
```

Multiclass Support Vector Machine (SVM) (25 pts)

Support vector machines (SVMs) are a set of supervised learning methods used for classification.

The advantages of support vector machines are:

- Effective in high dimensional spaces.
- Still effective in cases where number of dimensions is greater than the number of samples.
- Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.
- Versatile: different Kernel functions can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.

The disadvantages of support vector machines include:

- If the number of features is much greater than the number of samples, avoid over-fitting in choosing Kernel functions and regularization term is crucial.
- SVMs do not directly provide probability estimates, these are calculated using an expensive five-fold cross-validation (see Scores and probabilities, below).

In this section, we will first implement the loss function for SVM and use the validation set to tune hyperparameters.

NOTE: please use numpy, please do not use scikit-learn, PyTorch or other libraries.

Loss Function (20 pts)

We first structure the loss function for SVM. For detailed explanations of SVM loss, please check out this reading material.

```
def svm_loss(W, X, y, reg):
    """
    Structured SVM loss function implementation.

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

    Inputs:
    - W: A numpy array of shape (D, C) containing weights.
    - 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.
    - reg: (float) regularization strength
    Returns a tuple of:
    - loss as single float
    - gradient with respect to weights W; an array of same shape as W
    """
    loss = 0.0
    dW = np.zeros(W.shape)  # initialize the gradient as zero


######################################################################
#######
    # TODO (10 pts):
#
    # Implement a vectorized version of the structured SVM loss,
storing the    #
    # result in loss. Refer to https://cs231n.github.io/linear-
classify/        #

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

    # Get number of training samples
    num_train = X.shape[0]  # Fix potential issue with previous
num_train

    # Compute class scores
    scores = X @ W  # Shape (N, C)

    # Ensure y is an integer for indexing
    y = y.astype(int)

    # Extract correct class scores
    correct_class_scores = scores[np.arange(num_train), y].reshape(-1,
1)  # Shape (N, 1)

    # Compute margin (max(0, s_j - s_{y_i} + 1))
    margins = np.maximum(0, scores - correct_class_scores + 1)  #
Shape (N, C)
    margins[np.arange(num_train), y] = 0  # Ignore correct class

    # Compute loss
    loss = np.sum(margins) / num_train  # Average over all samples
    loss += reg * np.sum(W ** 2)  # Add regularization term

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


######################################################################
#######
    # TODO (10 pts):
```

```
#
    # Implement a vectorized version of the gradient for the
structured SVM     #
    # loss, storing the result in dW.
#
    #
#
    # Hint: Instead of computing the gradient from scratch, it may be
easier    #
    # to reuse some of the intermediate values that you used to
compute the     #
    # loss.
#

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

    # Compute mask where margins > 0
    margin_mask = (margins > 0).astype(float)

    # Count incorrect classifications for each sample
    margin_mask[np.arange(num_train), y] = -np.sum(margin_mask,
axis=1)

    # Compute gradient
    dW = (X.T @ margin_mask) / num_train  # Average over all samples
    dW += 2 * reg * W  # Regularization term
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    return loss, dW
```

Now, we can test our implementation of SVM loss.

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

tic = time.time()
loss, _ = svm_loss(W, X_dev.numpy(), y_dev.numpy(), 0.000005)
toc = time.time()
print('loss: %e computed in %fs' % (loss, toc - tic))
```

```
loss: 8.998826e+00 computed in 0.020505s
```

```
class LinearSVM(LinearClassifier):
    """ A subclass that uses the Multiclass SVM loss function """

    def loss(self, X_batch, y_batch, reg):
        return svm_loss(self.W, X_batch, y_batch, reg)
```

```
svm = LinearSVM()
tic = time.time()
loss_hist = svm.train(X_train.numpy(), y_train.numpy(),
learning_rate=1e-7, reg=2.5e4,
                      num_iters=1500, verbose=True)
toc = time.time()
print('That took %fs' % (toc - tic))
```

```
iteration 0 / 1500: loss 782.116353
iteration 100 / 1500: loss 292.702711
iteration 200 / 1500: loss 113.105350
iteration 300 / 1500: loss 47.201874
iteration 400 / 1500: loss 23.017616
iteration 500 / 1500: loss 14.143208
iteration 600 / 1500: loss 10.885566
iteration 700 / 1500: loss 9.691425
iteration 800 / 1500: loss 9.252887
iteration 900 / 1500: loss 9.092204
iteration 1000 / 1500: loss 9.032534
iteration 1100 / 1500: loss 9.011159
iteration 1200 / 1500: loss 9.003135
iteration 1300 / 1500: loss 9.000165
iteration 1400 / 1500: loss 8.999247
That took 8.266896s
```

```
y_train_pred = svm.predict(X_train.numpy())
print('training accuracy: %f' % (np.mean(y_train.numpy() ==
y_train_pred), ))
y_val_pred = svm.predict(X_val.numpy())
print('validation accuracy: %f' % (np.mean(y_val.numpy() ==
y_val_pred), ))
```

```
training accuracy: 0.242735
validation accuracy: 0.263000
```

Hyperparameter Tuning (5 pts)

Now we use the validation set to tune hyperparameters (regularization strength and learning rate). You should experiment with different ranges for the learning rates and regularization strengths.

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

```
# results is dictionary mapping tuples of the form
# (learning_rate, regularization_strength) to tuples of the form
# (training_accuracy, validation_accuracy). The accuracy is simply the
fraction
# of data points that are correctly classified.
```

```
results = {}
best_val = -1   # The highest validation accuracy that we have seen so
far.
best_svm = None # The LinearSVM object that achieved the highest
validation rate.

######################################################################
##########
# TODO (10 pts):
#
# Write code that chooses the best hyperparameters by tuning on the
validation #
# set. For each combination of hyperparameters, train a linear SVM on
the      #
# training set, compute its accuracy on the training and validation
sets, and  #
# store these numbers in the results dictionary. In addition, store
the best   #
# validation accuracy in best_val and the LinearSVM object that
achieves this  #
# accuracy in best_svm.
#
#
#
# Hint: You should use a small value for num_iters as you develop your
#
# validation code so that the SVMs don't take much time to train; once
you are #
# confident that your validation code works, you should rerun the
validation   #
# code with a larger value for num_iters.
                                                      #
######################################################################
##########
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
learning_rates = [0.005, 0.008, 0.01, 0.012, 0.015]
regularization_strengths = [5e-4, 8e-4, 1e-3, 3e-3, 5e-3]

# Try all combinations of hyperparameters
for lr in learning_rates:
   for reg in regularization_strengths:
       svm = LinearSVM()

       # Train with more iterations for better convergence
       loss_hist = svm.train(X_train.numpy(), y_train.numpy(),
                            learning_rate=lr, reg=reg,
                            num_iters=1500,
                            batch_size=200,
                            verbose=False)
```

```
        # Compute accuracies
        y_train_pred = svm.predict(X_train.numpy())
        train_acc = np.mean(y_train.numpy() == y_train_pred)

        y_val_pred = svm.predict(X_val.numpy())
        val_acc = np.mean(y_val.numpy() == y_val_pred)

        # Store results
        results[(lr, reg)] = (train_acc, val_acc)

        # Print current result to see progress
        print(f"lr {lr:.1e}, reg {reg:.1e}: train acc: {train_acc:.4f},
val acc: {val_acc:.4f}")

        # Update best model if validation accuracy improves
        if val_acc > best_val:
            best_val = val_acc
            best_svm = svm
            print(f"New best: {val_acc:.4f}")
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

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

print('best validation accuracy achieved: %f' % best_val)
y_test_pred = best_svm.predict(X_test.numpy())
test_acc = np.mean(y_test.numpy() == y_test_pred)
print('final test accuracy svm achieved: %f' % test_acc)
```

```
lr 5.0e-03, reg 5.0e-04: train acc: 0.4002, val acc: 0.4030
New best: 0.4030
lr 5.0e-03, reg 8.0e-04: train acc: 0.3996, val acc: 0.3880
lr 5.0e-03, reg 1.0e-03: train acc: 0.4023, val acc: 0.3900
lr 5.0e-03, reg 3.0e-03: train acc: 0.4010, val acc: 0.3980
lr 5.0e-03, reg 5.0e-03: train acc: 0.4014, val acc: 0.4040
New best: 0.4040
lr 8.0e-03, reg 5.0e-04: train acc: 0.3989, val acc: 0.3860
lr 8.0e-03, reg 8.0e-04: train acc: 0.4080, val acc: 0.3930
lr 8.0e-03, reg 1.0e-03: train acc: 0.4037, val acc: 0.4030
lr 8.0e-03, reg 3.0e-03: train acc: 0.4058, val acc: 0.4030
lr 8.0e-03, reg 5.0e-03: train acc: 0.4051, val acc: 0.4110
New best: 0.4110
lr 1.0e-02, reg 5.0e-04: train acc: 0.4079, val acc: 0.3990
lr 1.0e-02, reg 8.0e-04: train acc: 0.4069, val acc: 0.4070
lr 1.0e-02, reg 1.0e-03: train acc: 0.4042, val acc: 0.4010
lr 1.0e-02, reg 3.0e-03: train acc: 0.4090, val acc: 0.3950
lr 1.0e-02, reg 5.0e-03: train acc: 0.4017, val acc: 0.4020
lr 1.2e-02, reg 5.0e-04: train acc: 0.4106, val acc: 0.4120
New best: 0.4120
lr 1.2e-02, reg 8.0e-04: train acc: 0.4082, val acc: 0.3930
lr 1.2e-02, reg 1.0e-03: train acc: 0.4104, val acc: 0.4010
lr 1.2e-02, reg 3.0e-03: train acc: 0.4058, val acc: 0.3940
lr 1.2e-02, reg 5.0e-03: train acc: 0.4081, val acc: 0.3950
lr 1.5e-02, reg 5.0e-04: train acc: 0.3963, val acc: 0.3900
lr 1.5e-02, reg 8.0e-04: train acc: 0.4090, val acc: 0.4070
lr 1.5e-02, reg 1.0e-03: train acc: 0.4089, val acc: 0.4030
lr 1.5e-02, reg 3.0e-03: train acc: 0.3987, val acc: 0.3850
lr 1.5e-02, reg 5.0e-03: train acc: 0.4079, val acc: 0.3970
lr 5.000000e-03 reg 5.000000e-04 train accuracy: 0.400224 val
accuracy: 0.403000
lr 5.000000e-03 reg 8.000000e-04 train accuracy: 0.399571 val
accuracy: 0.388000
lr 5.000000e-03 reg 1.000000e-03 train accuracy: 0.402265 val
accuracy: 0.390000
lr 5.000000e-03 reg 3.000000e-03 train accuracy: 0.401041 val
accuracy: 0.398000
lr 5.000000e-03 reg 5.000000e-03 train accuracy: 0.401367 val
accuracy: 0.404000
lr 8.000000e-03 reg 5.000000e-04 train accuracy: 0.398939 val
accuracy: 0.386000
lr 8.000000e-03 reg 8.000000e-04 train accuracy: 0.407980 val
accuracy: 0.393000
lr 8.000000e-03 reg 1.000000e-03 train accuracy: 0.403714 val
accuracy: 0.403000
lr 8.000000e-03 reg 3.000000e-03 train accuracy: 0.405776 val
accuracy: 0.403000
lr 8.000000e-03 reg 5.000000e-03 train accuracy: 0.405082 val
accuracy: 0.411000
lr 1.000000e-02 reg 5.000000e-04 train accuracy: 0.407898 val
accuracy: 0.399000
lr 1.000000e-02 reg 8.000000e-04 train accuracy: 0.406878 val
accuracy: 0.407000
lr 1.000000e-02 reg 1.000000e-03 train accuracy: 0.404224 val
accuracy: 0.401000
lr 1.000000e-02 reg 3.000000e-03 train accuracy: 0.409000 val
accuracy: 0.395000
lr 1.000000e-02 reg 5.000000e-03 train accuracy: 0.401735 val
accuracy: 0.402000
lr 1.200000e-02 reg 5.000000e-04 train accuracy: 0.410633 val
accuracy: 0.412000
lr 1.200000e-02 reg 8.000000e-04 train accuracy: 0.408245 val
accuracy: 0.393000
lr 1.200000e-02 reg 1.000000e-03 train accuracy: 0.410388 val
accuracy: 0.401000
lr 1.200000e-02 reg 3.000000e-03 train accuracy: 0.405755 val
accuracy: 0.394000
```

```
lr 1.200000e-02 reg 5.000000e-03 train accuracy: 0.408102 val
accuracy: 0.395000
lr 1.500000e-02 reg 5.000000e-04 train accuracy: 0.396265 val
accuracy: 0.390000
lr 1.500000e-02 reg 8.000000e-04 train accuracy: 0.408980 val
accuracy: 0.407000
lr 1.500000e-02 reg 1.000000e-03 train accuracy: 0.408898 val
accuracy: 0.403000
lr 1.500000e-02 reg 3.000000e-03 train accuracy: 0.398714 val
accuracy: 0.385000
lr 1.500000e-02 reg 5.000000e-03 train accuracy: 0.407878 val
accuracy: 0.397000
best validation accuracy achieved: 0.412000
final test accuracy svm achieved: 0.384000
```

Implementing a Neural Network (40 pts)

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

We train the network with a cross-entropy loss function and L2 regularization on the weight matrices. The network uses a Sigmoid nonlinearity after the first fully connected layer.

In other words, the network has the following architecture:

input -> fully connected layer -> Sigmoid -> fully connected layer -> softmax -> cross-entropy

The outputs of the second fully-connected layer are the scores for each class.

Note: When you implement the regularization over W, **please DO NOT multiply the regularization term by 1/2** (no coefficient).

```
# Template class modules that we will use later: Do not edit/modify
this class
class TwoLayerNet(object):
  def __init__(self, input_size, hidden_size, output_size,
               dtype=torch.float32, device='cuda', std=1e-4):
    """
    Initialize the model. Weights are initialized to small random
values and
    biases are initialized to zero. Weights and biases are stored in
the
    variable self.params, which is a dictionary with the following
keys:

    W1: First layer weights; has shape (D, H)
    b1: First layer biases; has shape (H,)
    W2: Second layer weights; has shape (H, C)
    b2: Second layer biases; has shape (C,)

    Inputs:
    - input_size: The dimension D of the input data.
    - hidden_size: The number of neurons H in the hidden layer.
    - output_size: The number of classes C.
    - dtype: Optional, data type of each initial weight params
    - device: Optional, whether the weight params is on GPU or CPU
    - std: Optional, initial weight scaler.
    """
    # reset seed before start
    random.seed(0)
    torch.manual_seed(0)

    self.params = {}
    self.params['W1'] = std * torch.randn(input_size, hidden_size,
dtype=dtype, device=device)
    self.params['b1'] = torch.zeros(hidden_size, dtype=dtype,
device=device)
    self.params['W2'] = std * torch.randn(hidden_size, output_size,
dtype=dtype, device=device)
    self.params['b2'] = torch.zeros(output_size, dtype=dtype,
device=device)

  def loss(self, X, y=None, reg=0.0):
    return nn_forward_backward(self.params, X, y, reg)

  def train(self, X, y, X_val, y_val,
            learning_rate=1e-3, learning_rate_decay=0.95,
            reg=5e-6, num_iters=100,
            batch_size=200, verbose=False):
    return nn_train(
            self.params,
            nn_forward_backward,
            nn_predict,
            X, y, X_val, y_val,
            learning_rate, learning_rate_decay,
            reg, num_iters, batch_size, verbose)

  def predict(self, X):
    return nn_predict(self.params, nn_forward_backward, X)

  def save(self, path):
    torch.save(self.params, path)
    print("Saved in {}".format(path))

  def load(self, path):
    checkpoint = torch.load(path, map_location='cpu')
    self.params = checkpoint
    print("load checkpoint file: {}".format(path))
```

Forward pass function (5 pts)

```
def nn_forward_pass(params, X):
    """
    The first stage of our neural network implementation: Run the
forward pass
    of the network to compute the hidden layer features and
classification
    scores. The network architecture should be:

    FC layer -> ReLU (hidden) -> FC layer (scores)

    As a practice, we will NOT allow to use torch.relu and torch.nn
ops
    just for this time (you can use it from A3).

    Inputs:
    - params: a dictionary of PyTorch Tensor that store the weights of
a model.
      It should have following keys with shape
          W1: First layer weights; has shape (D, H)
          b1: First layer biases; has shape (H,)
          W2: Second layer weights; has shape (H, C)
          b2: Second layer biases; has shape (C,)
    - X: Input data of shape (N, D). Each X[i] is a training sample.

    Returns a tuple of:
    - scores: Tensor of shape (N, C) giving the classification scores
for X
    - hidden: Tensor of shape (N, H) giving the hidden layer
representation
      for each input value (after the ReLU).
    """
    # Unpack variables from the params dictionary
    W1, b1 = params['W1'], params['b1']
    W2, b2 = params['W2'], params['b2']
    N, D = X.shape

    # Compute the forward pass
    hidden = None
    scores = None
    def activation(z):
      # TODO: use sigmoid function
[https://en.wikipedia.org/wiki/Sigmoid_function]
      return 1.0 / (1.0 + torch.exp(-z))

######################################################################
######
    # TODO: Perform the forward pass, computing the class scores for
the input.#
```

```
    # Store the result in the scores variable, which should be an
tensor of    #
    # shape (N, C).
#

######################################################################
######
    # Replace "pass" statement with your code
    # First layer: linear transformation
    z1 = X.mm(W1) + b1

    # Apply sigmoid activation
    hidden = activation(z1)

    # Second layer: linear transformation to get scores
    scores = hidden.mm(W2) + b2

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

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

    return scores, hidden
```

Loss function + Gradients computation (15 pts)

```
def nn_forward_backward(params, X, y=None, reg=0.0):
    """
    Compute the loss and gradients for a two layer fully connected
neural
    network. When you implement loss and gradient, please don't forget
to
    scale the losses/gradients by the batch size.

    Inputs: First two parameters (params, X) are same as
nn_forward_pass
    - params: a dictionary of PyTorch Tensor that store the weights of
a model.
      It should have following keys with shape
          W1: First layer weights; has shape (D, H)
          b1: First layer biases; has shape (H,)
          W2: Second layer weights; has shape (H, C)
          b2: Second layer biases; has shape (C,)
    - X: Input data of shape (N, D). Each X[i] is a training sample.
    - y: Vector of training labels. y[i] is the label for X[i], and
each y[i] is
```

```
      an integer in the range 0 <= y[i] < C. This parameter is
optional; if it
      is not passed then we only return scores, and if it is passed
then we
      instead return the loss and gradients.
    - reg: Regularization strength.

    Returns:
    If y is None, return a tensor scores of shape (N, C) where
scores[i, c] is
    the score for class c on input X[i].

    If y is not None, instead return a tuple of:
    - loss: Loss (data loss and regularization loss) for this batch of
training
      samples.
    - grads: Dictionary mapping parameter names to gradients of those
parameters
      with respect to the loss function; has the same keys as
self.params.
    """
    # Unpack variables from the params dictionary
    W1, b1 = params['W1'], params['b1']
    W2, b2 = params['W2'], params['b2']
    N, D = X.shape

    scores, hidden = nn_forward_pass(params, X)
    # If the targets are not given then jump out, we're done
    if y is None:
      return scores

    # Compute the loss
    loss = None

######################################################################
######
    # TODO: Compute the loss, based on the results from
nn_forward_pass.       #
    # This should include both the data loss and L2 regularization for
W1 and  #
    # W2. Store the result in the variable loss, which should be a
scalar. Use #
    # the Cross-entropy classifier loss.
#
    # Please DO NOT multiply the regularization term by 1/2 (no
coefficient).  #
    # If you are not careful here, it is easy to run into numeric
instability  #
    # (Check Numeric Stability in http://cs231n.github.io/linear-
classify/).   #
```

```
#######################################################################
######
    scores_shifted = scores - scores.max(dim=1, keepdim=True)[0]  #
For numerical stability
    exp_scores = torch.exp(scores_shifted)
    probs = exp_scores / exp_scores.sum(dim=1, keepdim=True)

    # Cross-entropy loss
    correct_logprobs = -torch.log(probs[torch.arange(N), y])
    data_loss = correct_logprobs.sum() / N

    # L2 regularization
    reg_loss = reg * (torch.sum(W1 * W1) + torch.sum(W2 * W2))

    # Total loss
    loss = data_loss + reg_loss

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

#######################################################################
#####
    # Backward pass: compute gradients
    grads = {}

#######################################################################
#####
    # TODO: Compute the backward pass, computing the derivatives of
the       #
    # weights and biases. Store the results in the grads dictionary.
#
    # For example, grads['W1'] should store the gradient on W1, and be
a      #
    # tensor of same size
#

#######################################################################
#####
    # Gradient of the loss with respect to scores
    dscores = probs.clone()
    dscores[torch.arange(N), y] -= 1
    dscores /= N

    # Backprop through the second layer
    # Gradient with respect to W2 and b2
    grads['W2'] = hidden.t().mm(dscores)
    grads['b2'] = dscores.sum(dim=0)
    # Gradient of the loss with respect to hidden
    dhidden = dscores.mm(W2.t())

    # Backprop through the sigmoid activation
    # Gradient of sigmoid: dsigmoid = sigmoid * (1-sigmoid)
    dsigmoid = dhidden * hidden * (1 - hidden)

    # Backprop through the first layer
    # Gradient with respect to W1 and b1
    grads['W1'] = X.t().mm(dsigmoid)
    grads['b1'] = dsigmoid.sum(dim=0)

    # Add regularization gradient contribution
    grads['W2'] += 2 * reg * W2
    grads['W1'] += 2 * reg * W1

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

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

    return loss, grads
```

Weight updates (5 pts)

```
def nn_train(params, loss_func, pred_func, X, y, X_val, y_val,
            learning_rate=1e-3, learning_rate_decay=0.95,
            reg=5e-6, num_iters=100,
            batch_size=200, verbose=False):
  """
  Train this neural network using stochastic gradient descent.

  Inputs:
  - params: a dictionary of PyTorch Tensor that store the weights of a
model.
    It should have following keys with shape
        W1: First layer weights; has shape (D, H)
        b1: First layer biases; has shape (H,)
        W2: Second layer weights; has shape (H, C)
        b2: Second layer biases; has shape (C,)
  - loss_func: a loss function that computes the loss and the
gradients.
    It takes as input:
    - params: Same as input to nn_train
    - X_batch: A minibatch of inputs of shape (B, D)
    - y_batch: Ground-truth labels for X_batch
    - reg: Same as input to nn_train
    And it returns a tuple of:
      - loss: Scalar giving the loss on the minibatch
      - grads: Dictionary mapping parameter names to gradients of the
loss with
        respect to the corresponding parameter.
  - pred_func: prediction function that im
  - X: A PyTorch tensor of shape (N, D) giving training data.
  - y: A PyTorch tensor f shape (N,) giving training labels; y[i] = c
means that
    X[i] has label c, where 0 <= c < C.
  - X_val: A PyTorch tensor of shape (N_val, D) giving validation
data.
  - y_val: A PyTorch tensor of shape (N_val,) giving validation
labels.
  - learning_rate: Scalar giving learning rate for optimization.
  - learning_rate_decay: Scalar giving factor used to decay the
learning rate
    after each epoch.
  - reg: Scalar giving regularization strength.
  - num_iters: Number of steps to take when optimizing.
  - batch_size: Number of training examples to use per step.
  - verbose: boolean; if true print progress during optimization.

  Returns: A dictionary giving statistics about the training process
  """
  num_train = X.shape[0]
  iterations_per_epoch = max(num_train // batch_size, 1)

  # Use SGD to optimize the parameters in self.model
  loss_history = []
  train_acc_history = []
  val_acc_history = []

  for it in range(num_iters):
    indices = torch.randint(num_train, (batch_size,))
    y_batch = y[indices]
    X_batch = X[indices]

    # Compute loss and gradients using the current minibatch
    loss, grads = loss_func(params, X_batch, y=y_batch, reg=reg)
    loss_history.append(loss.item())


######################################################################
###
    # TODO: Use the gradients in the grads dictionary to update the
#
    # parameters of the network (stored in the dictionary self.params)
#
```

```
    # using stochastic gradient descent. You'll need to use the
gradients   #
    # stored in the grads dictionary defined above.
#

######################################################################
###
    # Replace "pass" statement with your code
    for param_name in params:
        params[param_name] -= learning_rate * grads[param_name]

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

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

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

    # Every epoch, check train and val accuracy and decay learning
rate.
    if it % iterations_per_epoch == 0:
      # Check accuracy
      y_train_pred = pred_func(params, loss_func, X_batch)
      train_acc = (y_train_pred == y_batch).float().mean().item()
      y_val_pred = pred_func(params, loss_func, X_val)
      val_acc = (y_val_pred == y_val).float().mean().item()
      train_acc_history.append(train_acc)
      val_acc_history.append(val_acc)

      # Decay learning rate
      learning_rate *= learning_rate_decay

  return {
    'loss_history': loss_history,
    'train_acc_history': train_acc_history,
    'val_acc_history': val_acc_history,
  }
```

Predict function (5 pts)

```
def nn_predict(params, loss_func, X):
  """
  Use the trained weights of this two-layer network to predict labels
for
```

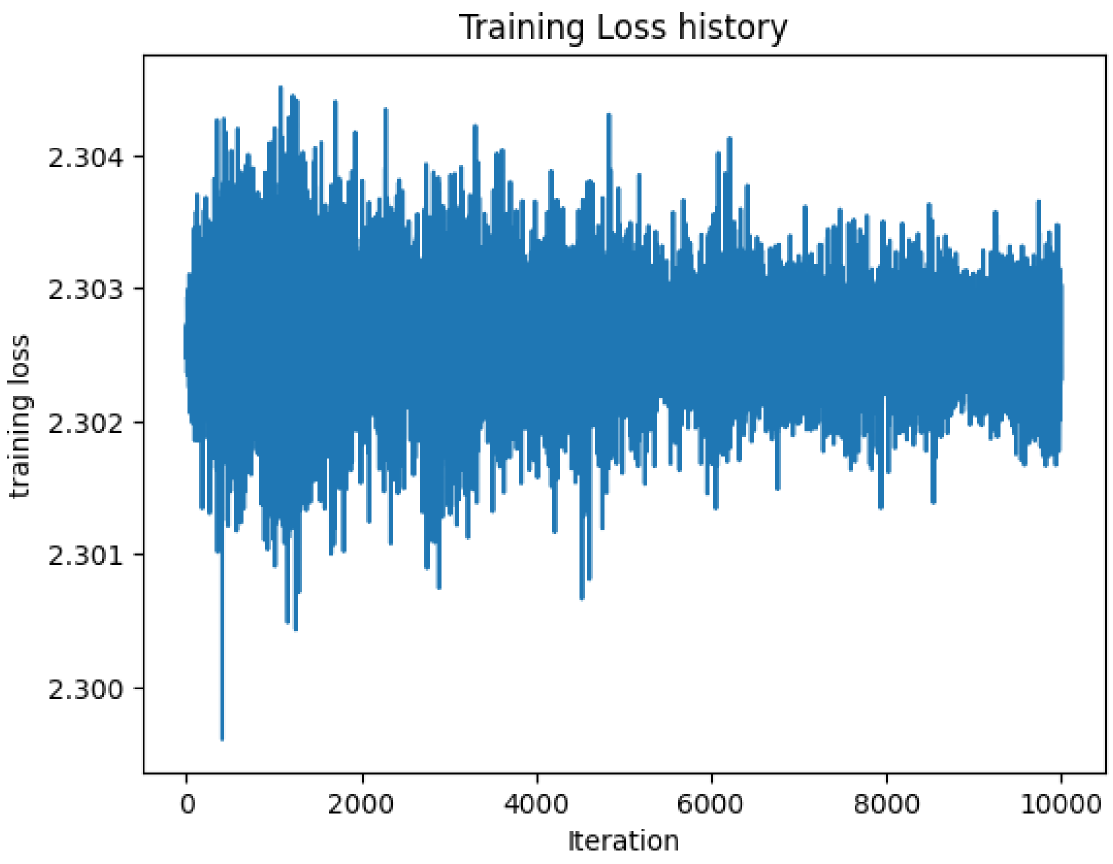
```
  data points. For each data point we predict scores for each of the C
  classes, and assign each data point to the class with the highest
score.

  Inputs:
  - params: a dictionary of PyTorch Tensor that store the weights of a
model.
    It should have following keys with shape
        W1: First layer weights; has shape (D, H)
        b1: First layer biases; has shape (H,)
        W2: Second layer weights; has shape (H, C)
        b2: Second layer biases; has shape (C,)
  - loss_func: a loss function that computes the loss and the
gradients
  - X: A PyTorch tensor of shape (N, D) giving N D-dimensional data
points to
    classify.

  Returns:
  - y_pred: A PyTorch tensor of shape (N,) giving predicted labels for
each of
    the elements of X. For all i, y_pred[i] = c means that X[i] is
predicted
    to have class c, where 0 <= c < C.
  """
  y_pred = None


######################################################################
#####
  # TODO: Implement this function; it should be VERY simple!
#

######################################################################
#####
  # Replace "pass" statement with your code
  # Get scores from the forward pass by calling the loss function with
y=None
  scores = loss_func(params, X)

  # Get predictions by taking argmax of scores
  y_pred = torch.argmax(scores, dim=1)

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

######################################################################
#####
```

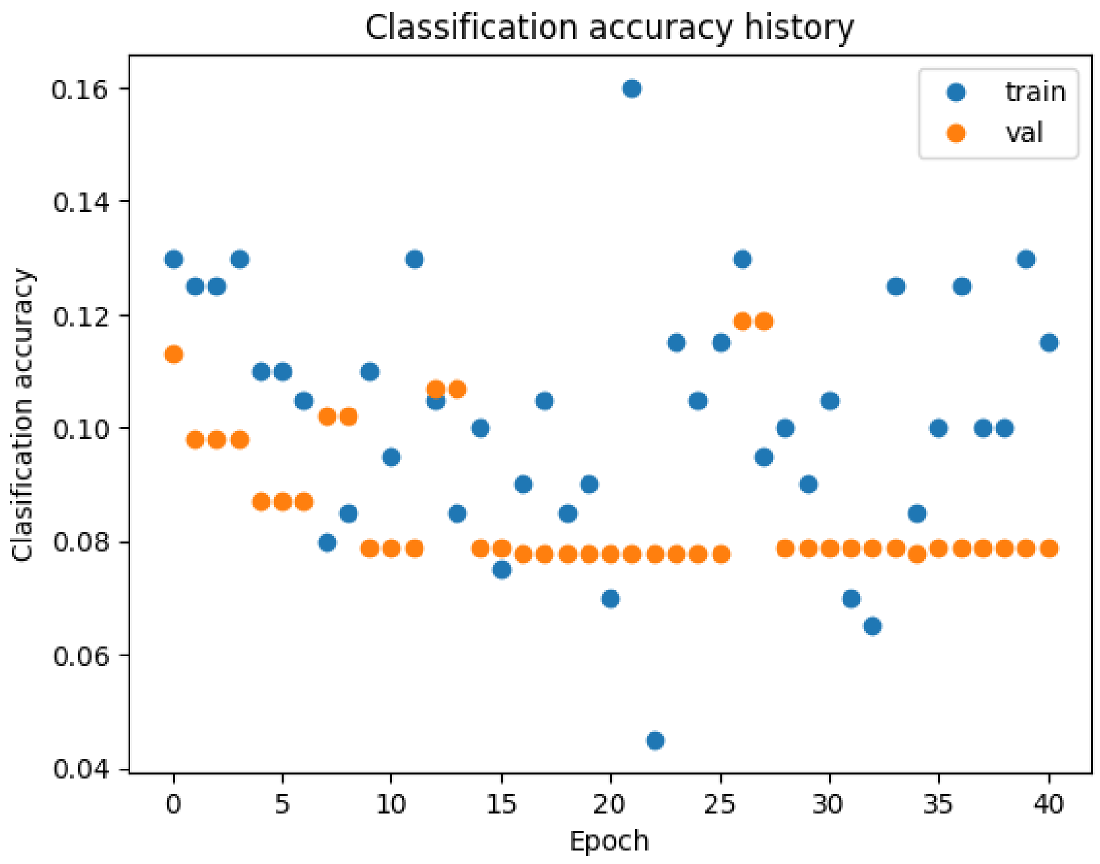
```
    return y_pred

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

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

    # Plot the loss function and train / validation accuracies
    plt.plot(stats['train_acc_history'], 'o', label='train')
    plt.plot(stats['val_acc_history'], 'o', label='val')
    plt.title('Classification accuracy history')
    plt.xlabel('Epoch')
    plt.ylabel('Clasification accuracy')
    plt.legend()
    plt.show()
```

Now, we can test our implementation of the neural network.

```
model = TwoLayerNet(input_size=X_train.shape[1], hidden_size=128,
output_size=10, device='cpu')
tic = time.time()
stats = model.train(X_train, y_train, X_val, y_val, verbose=False,
num_iters=10000)
toc = time.time()
print('That took %fs' % (toc - tic))
visualization(stats)
```

```
That took 93.635873s
Final training loss:  2.3030202388763428
```


Training Loss history
training loss
Iteration
2.300
2.301
2.302
2.303
2.304
0
2000
4000
6000
8000
10000



Hyperparameters tuning (10 pts)

Tuning. Tuning the hyperparameters and developing intuition for how they affect the final performance is a large part of using Neural Networks, so we want you to get a lot of practice. Below, you should experiment with different values of the various hyperparameters, including hidden layer size, learning rate, and regularization strength. You might also consider tuning other parameters such as num_iters as well.

Approximate results. To get full credit for the assignment, you should achieve a classification accuracy above 50% on the validation set.

```
results = {}
best_val = -1
best_nn = None
######################################################################
##########
# TODO (10 pts):
#
# Use the validation set to set the learning rate and regularization
strength. #
# This should be identical to the validation that you did for the SVM;
save    #
# the best trained softmax classifer in best_softmax.
```

```
#
######################################################################
##########
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
# Extended hyperparameter search space
learning_rates = [0.5, 0.7, 1.0]  # Try even higher learning rates
regularization_strengths = [0, 1e-7, 1e-6, 1e-5]  # Try lower
regularization
hidden_dims = [384, 512, 768, 1024]  # Try even larger hidden layers
# Try combinations of hyperparameters
print("Starting hyperparameter tuning...")
for lr in learning_rates:
    for reg in regularization_strengths:
        for hidden_dim in hidden_dims:
            print(f"Training with lr={lr}, reg={reg},
hidden_dim={hidden_dim}")

            # Create a new network with current hyperparameters
            model = TwoLayerNet(
                input_size=X_train.shape[1],
                hidden_size=hidden_dim,
                output_size=10,
                device='cpu'
            )

            # Train with current hyperparameters (use fewer iterations
for faster tuning)
            # Apply more aggressive learning rate decay for higher
learning rates
            lr_decay = 0.9 if lr >= 0.3 else 0.95  # More aggressive
decay for higher learning rates

            stats = model.train(
                X_train, y_train, X_val, y_val,
                learning_rate=lr,
                learning_rate_decay=lr_decay,  # Apply more aggressive
LR decay
                reg=reg,
                num_iters=2000,  # Reduced number of iterations for
faster tuning
                batch_size=200,
                verbose=False
            )

            # Get validation accuracy (take the last epoch's accuracy)
            train_accuracy = stats['train_acc_history'][-1]
            val_accuracy = stats['val_acc_history'][-1]

            # Store results
            results[(lr, reg, hidden_dim)] = (train_accuracy,
val_accuracy)

            # Update best validation accuracy and model
            if val_accuracy > best_val:
                best_val = val_accuracy
                best_nn = model
                print(f"New best model: val_accuracy = 
{val_accuracy:.4f}")

# After finding the best hyperparameters, retrain with more iterations
if best_nn is not None:
    best_lr, best_reg, best_hidden = max(results.keys(), key=lambda k: 
results[k][1])
    print(f"Best hyperparameters: lr={best_lr}, reg={best_reg}, 
hidden_dim={best_hidden}")

    # Set appropriate learning rate decay based on the learning rate
    best_lr_decay = 0.9 if best_lr >= 0.3 else 0.95

    # Retrain the best model with more iterations
    best_nn = TwoLayerNet(
        input_size=X_train.shape[1],
        hidden_size=best_hidden,
        output_size=10,
        device='cpu'
    )

    # Longer training period (8000-10000 iterations)
    best_stats = best_nn.train(
        X_train, y_train, X_val, y_val,
        learning_rate=best_lr,
        learning_rate_decay=best_lr_decay,
        reg=best_reg,
        num_iters=8000,  # Increased iterations for final model
        batch_size=200,
        verbose=True
    )
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

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

print('best validation accuracy achieved: %f' % best_val)

y_test_pred = best_nn.predict(X_test)
test_acc = (y_test_pred == y_test).double().mean().item()
```

```python
print('final test accuracy 2-layered neural network achieved: %f' %
test_acc)
```

```
Starting hyperparameter tuning...
Training with lr=0.5, reg=0, hidden_dim=384
New best model: val_accuracy = 0.4480
Training with lr=0.5, reg=0, hidden_dim=512
Training with lr=0.5, reg=0, hidden_dim=768
Training with lr=0.5, reg=0, hidden_dim=1024
Training with lr=0.5, reg=1e-07, hidden_dim=384
Training with lr=0.5, reg=1e-07, hidden_dim=512
Training with lr=0.5, reg=1e-07, hidden_dim=768
Training with lr=0.5, reg=1e-07, hidden_dim=1024
Training with lr=0.5, reg=1e-06, hidden_dim=384
New best model: val_accuracy = 0.4510
Training with lr=0.5, reg=1e-06, hidden_dim=512
Training with lr=0.5, reg=1e-06, hidden_dim=768
Training with lr=0.5, reg=1e-06, hidden_dim=1024
Training with lr=0.5, reg=1e-05, hidden_dim=384
New best model: val_accuracy = 0.4530
Training with lr=0.5, reg=1e-05, hidden_dim=512
Training with lr=0.5, reg=1e-05, hidden_dim=768
Training with lr=0.5, reg=1e-05, hidden_dim=1024
Training with lr=0.7, reg=0, hidden_dim=384
New best model: val_accuracy = 0.4720
Training with lr=0.7, reg=0, hidden_dim=512
Training with lr=0.7, reg=0, hidden_dim=768
Training with lr=0.7, reg=0, hidden_dim=1024
Training with lr=0.7, reg=1e-07, hidden_dim=384
Training with lr=0.7, reg=1e-07, hidden_dim=512
Training with lr=0.7, reg=1e-07, hidden_dim=768
Training with lr=0.7, reg=1e-07, hidden_dim=1024
Training with lr=0.7, reg=1e-06, hidden_dim=384
Training with lr=0.7, reg=1e-06, hidden_dim=512
Training with lr=0.7, reg=1e-06, hidden_dim=768
Training with lr=0.7, reg=1e-06, hidden_dim=1024
Training with lr=0.7, reg=1e-05, hidden_dim=384
Training with lr=0.7, reg=1e-05, hidden_dim=512
Training with lr=0.7, reg=1e-05, hidden_dim=768
Training with lr=0.7, reg=1e-05, hidden_dim=1024
Training with lr=1.0, reg=0, hidden_dim=384
New best model: val_accuracy = 0.4960
Training with lr=1.0, reg=0, hidden_dim=512
Training with lr=1.0, reg=0, hidden_dim=768
Training with lr=1.0, reg=0, hidden_dim=1024
Training with lr=1.0, reg=1e-07, hidden_dim=384
New best model: val_accuracy = 0.4980
Training with lr=1.0, reg=1e-07, hidden_dim=512
Training with lr=1.0, reg=1e-07, hidden_dim=768
Training with lr=1.0, reg=1e-07, hidden_dim=1024
Training with lr=1.0, reg=1e-06, hidden_dim=384
Training with lr=1.0, reg=1e-06, hidden_dim=512
New best model: val_accuracy = 0.5000
Training with lr=1.0, reg=1e-06, hidden_dim=768
Training with lr=1.0, reg=1e-06, hidden_dim=1024
Training with lr=1.0, reg=1e-05, hidden_dim=384
Training with lr=1.0, reg=1e-05, hidden_dim=512
New best model: val_accuracy = 0.5010
Training with lr=1.0, reg=1e-05, hidden_dim=768
Training with lr=1.0, reg=1e-05, hidden_dim=1024
Best hyperparameters: lr=1.0, reg=1e-05, hidden_dim=512
iteration 0 / 8000: loss 2.302712
iteration 100 / 8000: loss 2.277854
iteration 200 / 8000: loss 2.040915
iteration 300 / 8000: loss 1.831343
iteration 400 / 8000: loss 1.776872
iteration 500 / 8000: loss 1.749563
iteration 600 / 8000: loss 1.661909
iteration 700 / 8000: loss 1.730476
iteration 800 / 8000: loss 1.656007
iteration 900 / 8000: loss 1.657022
iteration 1000 / 8000: loss 1.531593
iteration 1100 / 8000: loss 1.532537
iteration 1200 / 8000: loss 1.407128
iteration 1300 / 8000: loss 1.429646
iteration 1400 / 8000: loss 1.504884
iteration 1500 / 8000: loss 1.379798
iteration 1600 / 8000: loss 1.300159
iteration 1700 / 8000: loss 1.406824
iteration 1800 / 8000: loss 1.456739
iteration 1900 / 8000: loss 1.409551
iteration 2000 / 8000: loss 1.182867
iteration 2100 / 8000: loss 1.412113
iteration 2200 / 8000: loss 1.426808
iteration 2300 / 8000: loss 1.400980
iteration 2400 / 8000: loss 1.241284
iteration 2500 / 8000: loss 1.304014
iteration 2600 / 8000: loss 1.343309
iteration 2700 / 8000: loss 1.239685
iteration 2800 / 8000: loss 1.265587
iteration 2900 / 8000: loss 1.346947
iteration 3000 / 8000: loss 1.339279
iteration 3100 / 8000: loss 1.307005
iteration 3200 / 8000: loss 1.227780
iteration 3300 / 8000: loss 1.109372
iteration 3400 / 8000: loss 1.135645
iteration 3500 / 8000: loss 1.216051
iteration 3600 / 8000: loss 1.245631
iteration 3700 / 8000: loss 1.210782
iteration 3800 / 8000: loss 1.275728
iteration 3900 / 8000: loss 1.172280
iteration 4000 / 8000: loss 1.267400
iteration 4100 / 8000: loss 1.219921
iteration 4200 / 8000: loss 1.059330
iteration 4300 / 8000: loss 1.167317
iteration 4400 / 8000: loss 1.088710
iteration 4500 / 8000: loss 1.123137
iteration 4600 / 8000: loss 1.114026
iteration 4700 / 8000: loss 1.196509
iteration 4800 / 8000: loss 1.119819
iteration 4900 / 8000: loss 1.127917
iteration 5000 / 8000: loss 1.200658
iteration 5100 / 8000: loss 1.325636
iteration 5200 / 8000: loss 1.216488
iteration 5300 / 8000: loss 1.191004
iteration 5400 / 8000: loss 1.154697
iteration 5500 / 8000: loss 1.206100
iteration 5600 / 8000: loss 1.127197
iteration 5700 / 8000: loss 1.189977
iteration 5800 / 8000: loss 1.152224
iteration 5900 / 8000: loss 1.077864
iteration 6000 / 8000: loss 1.179621
iteration 6100 / 8000: loss 1.119994
iteration 6200 / 8000: loss 1.270168
iteration 6300 / 8000: loss 1.172428
iteration 6400 / 8000: loss 1.135115
iteration 6500 / 8000: loss 1.098096
iteration 6600 / 8000: loss 1.137819
iteration 6700 / 8000: loss 1.021511
iteration 6800 / 8000: loss 1.260520
iteration 6900 / 8000: loss 1.162058
iteration 7000 / 8000: loss 1.167908
iteration 7100 / 8000: loss 1.041457
iteration 7200 / 8000: loss 1.154747
iteration 7300 / 8000: loss 1.195556
iteration 7400 / 8000: loss 1.142495
iteration 7500 / 8000: loss 1.161962
iteration 7600 / 8000: loss 1.124055
iteration 7700 / 8000: loss 1.012036
iteration 7800 / 8000: loss 1.132277
iteration 7900 / 8000: loss 1.185427
lr 5.000000e-01 reg 0.000000e+00 H 3.840000e+02 train accuracy:
0.450000 val accuracy: 0.448000
lr 5.000000e-01 reg 0.000000e+00 H 5.120000e+02 train accuracy:
0.515000 val accuracy: 0.441000
lr 5.000000e-01 reg 0.000000e+00 H 7.680000e+02 train accuracy:
0.505000 val accuracy: 0.434000
lr 5.000000e-01 reg 0.000000e+00 H 1.024000e+03 train accuracy:
```

```
0.485000 val accuracy: 0.438000
lr 5.000000e-01 reg 1.000000e-07 H 3.840000e+02 train accuracy:
0.460000 val accuracy: 0.447000
lr 5.000000e-01 reg 1.000000e-07 H 5.120000e+02 train accuracy:
0.500000 val accuracy: 0.445000
lr 5.000000e-01 reg 1.000000e-07 H 7.680000e+02 train accuracy:
0.505000 val accuracy: 0.430000
lr 5.000000e-01 reg 1.000000e-07 H 1.024000e+03 train accuracy:
0.495000 val accuracy: 0.440000
lr 5.000000e-01 reg 1.000000e-06 H 3.840000e+02 train accuracy:
0.460000 val accuracy: 0.451000
lr 5.000000e-01 reg 1.000000e-06 H 5.120000e+02 train accuracy:
0.505000 val accuracy: 0.441000
lr 5.000000e-01 reg 1.000000e-06 H 7.680000e+02 train accuracy:
0.495000 val accuracy: 0.434000
lr 5.000000e-01 reg 1.000000e-06 H 1.024000e+03 train accuracy:
0.490000 val accuracy: 0.447000
lr 5.000000e-01 reg 1.000000e-05 H 3.840000e+02 train accuracy:
0.470000 val accuracy: 0.453000
lr 5.000000e-01 reg 1.000000e-05 H 5.120000e+02 train accuracy:
0.520000 val accuracy: 0.442000
lr 5.000000e-01 reg 1.000000e-05 H 7.680000e+02 train accuracy:
0.505000 val accuracy: 0.427000
lr 5.000000e-01 reg 1.000000e-05 H 1.024000e+03 train accuracy:
0.490000 val accuracy: 0.442000
lr 7.000000e-01 reg 0.000000e+00 H 3.840000e+02 train accuracy:
0.495000 val accuracy: 0.472000
lr 7.000000e-01 reg 0.000000e+00 H 5.120000e+02 train accuracy:
0.560000 val accuracy: 0.460000
lr 7.000000e-01 reg 0.000000e+00 H 7.680000e+02 train accuracy:
0.555000 val accuracy: 0.450000
lr 7.000000e-01 reg 0.000000e+00 H 1.024000e+03 train accuracy:
0.525000 val accuracy: 0.460000
lr 7.000000e-01 reg 1.000000e-07 H 3.840000e+02 train accuracy:
0.490000 val accuracy: 0.469000
lr 7.000000e-01 reg 1.000000e-07 H 5.120000e+02 train accuracy:
0.540000 val accuracy: 0.467000
lr 7.000000e-01 reg 1.000000e-07 H 7.680000e+02 train accuracy:
0.560000 val accuracy: 0.447000
lr 7.000000e-01 reg 1.000000e-07 H 1.024000e+03 train accuracy:
0.530000 val accuracy: 0.468000
lr 7.000000e-01 reg 1.000000e-06 H 3.840000e+02 train accuracy:
0.475000 val accuracy: 0.470000
lr 7.000000e-01 reg 1.000000e-06 H 5.120000e+02 train accuracy:
0.560000 val accuracy: 0.457000
lr 7.000000e-01 reg 1.000000e-06 H 7.680000e+02 train accuracy:
0.545000 val accuracy: 0.453000
lr 7.000000e-01 reg 1.000000e-06 H 1.024000e+03 train accuracy:
0.520000 val accuracy: 0.459000
```

```
lr 7.000000e-01 reg 1.000000e-05 H 3.840000e+02 train accuracy:
0.470000 val accuracy: 0.464000
lr 7.000000e-01 reg 1.000000e-05 H 5.120000e+02 train accuracy:
0.550000 val accuracy: 0.463000
lr 7.000000e-01 reg 1.000000e-05 H 7.680000e+02 train accuracy:
0.550000 val accuracy: 0.451000
lr 7.000000e-01 reg 1.000000e-05 H 1.024000e+03 train accuracy:
0.515000 val accuracy: 0.462000
lr 1.000000e+00 reg 0.000000e+00 H 3.840000e+02 train accuracy:
0.530000 val accuracy: 0.496000
lr 1.000000e+00 reg 0.000000e+00 H 5.120000e+02 train accuracy:
0.580000 val accuracy: 0.493000
lr 1.000000e+00 reg 0.000000e+00 H 7.680000e+02 train accuracy:
0.600000 val accuracy: 0.484000
lr 1.000000e+00 reg 0.000000e+00 H 1.024000e+03 train accuracy:
0.560000 val accuracy: 0.468000
lr 1.000000e+00 reg 1.000000e-07 H 3.840000e+02 train accuracy:
0.510000 val accuracy: 0.498000
lr 1.000000e+00 reg 1.000000e-07 H 5.120000e+02 train accuracy:
0.585000 val accuracy: 0.495000
lr 1.000000e+00 reg 1.000000e-07 H 7.680000e+02 train accuracy:
0.595000 val accuracy: 0.486000
lr 1.000000e+00 reg 1.000000e-07 H 1.024000e+03 train accuracy:
0.550000 val accuracy: 0.473000
lr 1.000000e+00 reg 1.000000e-06 H 3.840000e+02 train accuracy:
0.510000 val accuracy: 0.494000
lr 1.000000e+00 reg 1.000000e-06 H 5.120000e+02 train accuracy:
0.585000 val accuracy: 0.500000
lr 1.000000e+00 reg 1.000000e-06 H 7.680000e+02 train accuracy:
0.575000 val accuracy: 0.477000
lr 1.000000e+00 reg 1.000000e-06 H 1.024000e+03 train accuracy:
0.555000 val accuracy: 0.473000
lr 1.000000e+00 reg 1.000000e-05 H 3.840000e+02 train accuracy:
0.520000 val accuracy: 0.490000
lr 1.000000e+00 reg 1.000000e-05 H 5.120000e+02 train accuracy:
0.575000 val accuracy: 0.501000
lr 1.000000e+00 reg 1.000000e-05 H 7.680000e+02 train accuracy:
0.600000 val accuracy: 0.484000
lr 1.000000e+00 reg 1.000000e-05 H 1.024000e+03 train accuracy:
0.575000 val accuracy: 0.483000
best validation accuracy achieved: 0.501000
final test accuracy 2-layered neural network achieved: 0.544000
```

Acknowledgement

Credits to UMichigan's 498/598 Deep Learning for Computer Vision and Stanfords's CS231n: Convolutional Neural Networks for Visual Recognition, some code is adapted from their courses's assignments.