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

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./cifar-10-python.t
100%|██████████| 170M/170M [00:02<00:00, 63.8MB/s]
Extracting ./cifar-10-python.tar.gz to .
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([50
```

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)*****

    # Dot products between each training and test image for numerator of cosine similarity
    dot_products = torch.mm(train, test.t())

    # L2 norms of each vector for denominator of cosine similarity
    norm_train = torch.norm(train, dim=1, keepdim=True)  # shape: (num_train, 1)
    norm_test = torch.norm(test, dim=1, keepdim=True)    # shape: (num_test, 1)

    # Dot products / L2 norms = cosine similarity
    dists = dot_products / (norm_train * norm_test.t())

    # *****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
    for i in range(x_test.shape[0]):
        # For the i-th test image, extract its similarity scores with all training images.
        # Since dists has shape (num_train, num_test), i-th column = i-th test image.
        similarities = dists[:, i]

    # 2. Find the indices of the k most similar training samples (highest cosine similarit
        topk_sim, topk_idx = torch.topk(similarities, k=k, largest=True)

    # 3. Retrieve the labels of these k neighbors and compute their contributions as the s
        neighbor_labels = self.y_train[topk_idx]
        # Sum weighted votes for each label.
        weights = {}
        for sim, label in zip(topk_sim, neighbor_labels):
            label = int(label)
            weights[label] = weights.get(label, 0.0) + sim.item()

    # 4. Assign the label with the highest accumulated weight as the final prediction.
        y_test_pred[i] = max(weights.items(), key=lambda x: x[1])[0]

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

    # *****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)
```

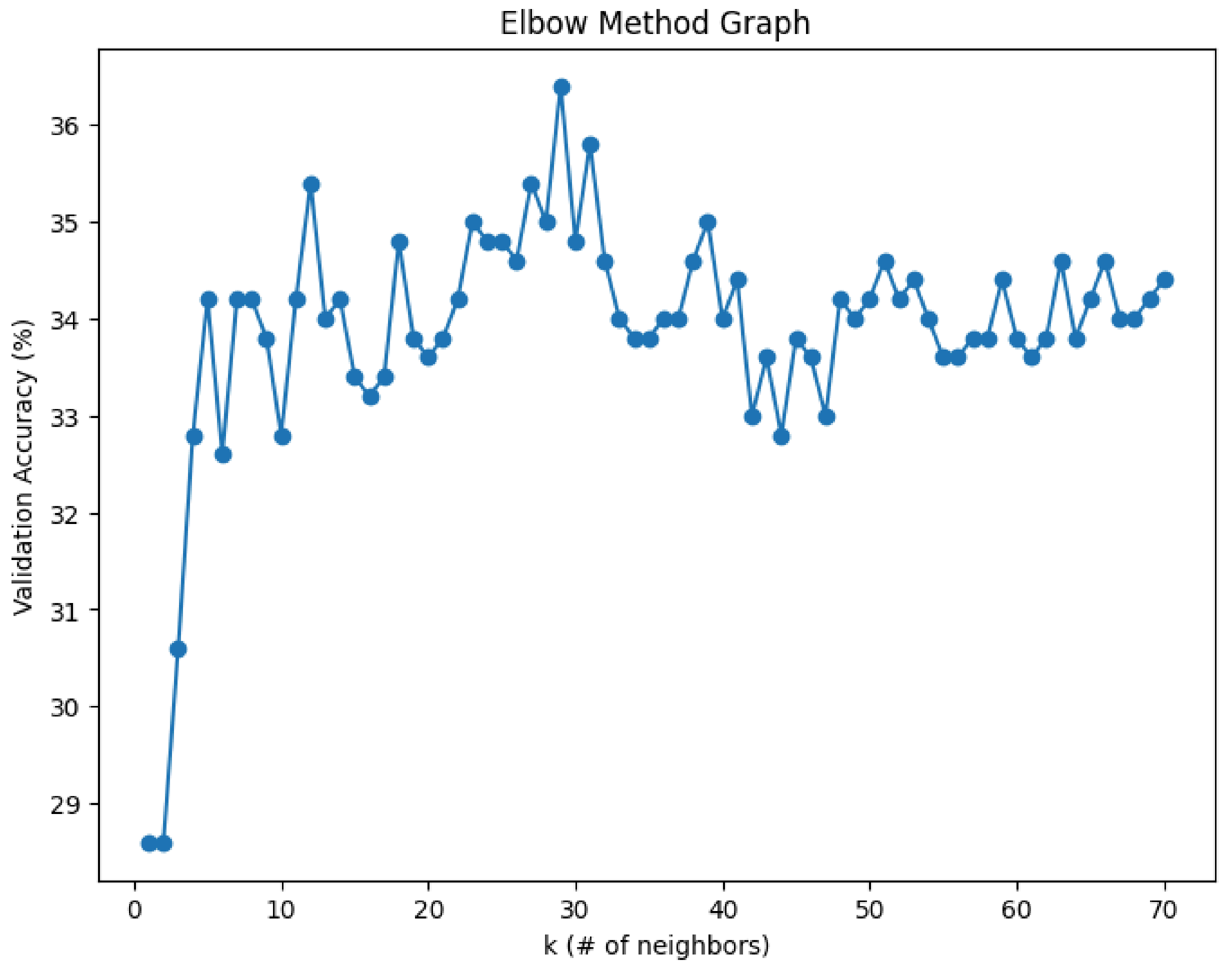
```
Got 168 / 500 correct; accuracy is 33.60%
33.6
```

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

import matplotlib.pyplot as plt

# Square root of the number of training images = upper bound for candidate k values
k_max = int(np.sqrt(num_train))
```

```
k_candidates = list(range(1, k_max + 1))

# Create a classifier trained on the subsampled training data.
classifier = KnnClassifier(knn_x_train, knn_y_train)

# Evaluate the classifier on the validation set for each candidate k.
for k in k_candidates:
    val_accuracy = classifier.check_accuracy(knn_x_val, knn_y_val, k=k, quiet=True)
    results[k] = val_accuracy
    if val_accuracy > best_val:
        best_val = val_accuracy
        best_k = k

# Plot the validation accuracy as a function of k (elbow plot).
plt.figure(figsize=(8, 6))
plt.plot(list(results.keys()), list(results.values()), marker='o')
plt.xlabel('k (# of neighbors)')
plt.ylabel('Validation Accuracy (%)')
plt.title('Elbow Method Graph')
plt.show()

# *****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 k:', best_k, '; with validation accuracy:', best_val)

print("Running accuracy check on the test images with best k value...")
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 2 val accuracy: 28.600000
k 3 val accuracy: 30.600000
k 4 val accuracy: 32.800000
k 5 val accuracy: 34.200000
k 6 val accuracy: 32.600000
k 7 val accuracy: 34.200000
k 8 val accuracy: 34.200000
k 9 val accuracy: 33.800000
k 10 val accuracy: 32.800000
k 11 val accuracy: 34.200000
k 12 val accuracy: 35.400000
k 13 val accuracy: 34.000000
k 14 val accuracy: 34.200000
k 15 val accuracy: 33.400000
k 16 val accuracy: 33.200000
k 17 val accuracy: 33.400000
k 18 val accuracy: 34.800000
k 19 val accuracy: 33.800000
k 20 val accuracy: 33.600000
k 21 val accuracy: 33.800000
k 22 val accuracy: 34.200000
k 23 val accuracy: 35.000000
k 24 val accuracy: 34.800000
k 25 val accuracy: 34.800000
k 26 val accuracy: 34.600000
k 27 val accuracy: 35.400000
k 28 val accuracy: 35.000000
k 29 val accuracy: 36.400000
k 30 val accuracy: 34.800000
k 31 val accuracy: 35.800000
```

```
k 32 val accuracy: 34.600000
k 33 val accuracy: 34.000000
k 34 val accuracy: 33.800000
k 35 val accuracy: 33.800000
k 36 val accuracy: 34.000000
k 37 val accuracy: 34.000000
k 38 val accuracy: 34.600000
k 39 val accuracy: 35.000000
k 40 val accuracy: 34.000000
k 41 val accuracy: 34.400000
k 42 val accuracy: 33.000000
k 43 val accuracy: 33.600000
k 44 val accuracy: 32.800000
k 45 val accuracy: 33.800000
k 46 val accuracy: 33.600000
k 47 val accuracy: 33.000000
k 48 val accuracy: 34.200000
k 49 val accuracy: 34.000000
k 50 val accuracy: 34.200000
k 51 val accuracy: 34.600000
k 52 val accuracy: 34.200000
k 53 val accuracy: 34.400000
k 54 val accuracy: 34.000000
k 55 val accuracy: 33.600000
k 56 val accuracy: 33.600000
k 57 val accuracy: 33.800000
k 58 val accuracy: 33.800000
k 59 val accuracy: 34.400000
k 60 val accuracy: 33.800000
k 61 val accuracy: 33.600000
k 62 val accuracy: 33.800000
k 63 val accuracy: 34.600000
k 64 val accuracy: 33.800000
k 65 val accuracy: 34.200000
k 66 val accuracy: 34.600000
k 67 val accuracy: 34.000000
k 68 val accuracy: 34.000000
k 69 val accuracy: 34.200000
k 70 val accuracy: 34.400000
Best k: 29 ; with validation accuracy: 36.4
Running accuracy check on the test images with best k value...
Got 176 / 500 correct; accuracy is 35.20%
Final test accuracy knn achieved: 35.200000
```

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,
    ):
        # num_train, dim = X.shape
        # print("Training on %d examples, each with %d features." % (num_train, dim))

        """
        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)*****

            # Sample batch_size elements from the training data and corresponding labels.
            indices = np.random.choice(num_train, batch_size, replace=True)
            X_batch = X[indices]
            y_batch = y[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 = np.dot(X, self.W)
    y_pred = np.argmax(scores, axis=1)

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

    N = X.shape[0]  # number of examples
    scores = X.dot(W)  # shape: (N, C)
    # Select the correct class scores (shape: (N,))
    correct_class_scores = scores[np.arange(N), y]

    # Compute margins: max(0, score_j - score_yi + delta)
    delta = 1.0
    margins = np.maximum(0, scores - correct_class_scores[:, np.newaxis] + delta)
    # Don't consider correct classes in the loss.
    margins[np.arange(N), y] = 0

    # Compute loss: average over all examples + regularization
    loss = np.sum(margins) / N + reg * np.sum(W * W)

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

    # Gradient calculation:
    # Create mask where margins > 0 are marked as 1.
    mask = np.zeros_like(margins)
    mask[margins > 0] = 1

    # For each example, count how many times we had a positive margin.
    row_sum = np.sum(mask, axis=1)
    # For the correct class, subtract the count.
    mask[np.arange(N), y] = -row_sum

    # The gradient is then computed as the dot product of X^T and the mask,
    # averaged over the number of examples, with the regularization gradient added.
    dW = X.T.dot(mask) / N + 2 * reg * W

    # *****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: 9.000850e+00 computed in 0.014595s
```

```
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 777.207030
iteration 100 / 1500: loss 290.904548
iteration 200 / 1500: loss 112.445255
iteration 300 / 1500: loss 46.959406
iteration 400 / 1500: loss 22.927159
iteration 500 / 1500: loss 14.110980
iteration 600 / 1500: loss 10.874773
iteration 700 / 1500: loss 9.687515
iteration 800 / 1500: loss 9.250832
iteration 900 / 1500: loss 9.091505
iteration 1000 / 1500: loss 9.031718
iteration 1100 / 1500: loss 9.011399
iteration 1200 / 1500: loss 9.003725
iteration 1300 / 1500: loss 9.001006
iteration 1400 / 1500: loss 8.999284
That took 9.921838s
```

```
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: 0.244755
validation accuracy: 0.253000
```

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)*****
# Fill in your own values
learning_rates = [0.005, 0.01, 0.02, 0.03] # [0.01, 0.05, 0.1, 0.2, 0.5]
regularization_strengths =[0.0005, 0.001, 0.002, 0.005] # [0.001, 0.01, 0.1, 1, 10]

results = {}
best_val = -1
best_svm = None

# Relatively small # of iterations for tuning so that the code runs quickly.
num_iters = 1500

for lr in learning_rates:
    for reg in regularization_strengths:
        svm = LinearSVM()
        loss_hist = svm.train(X_train.numpy(), y_train.numpy(),
                              learning_rate=lr, reg=reg,
                              num_iters=num_iters, verbose=False)
        train_pred = svm.predict(X_train.numpy())
        train_acc = np.mean(y_train.numpy() == train_pred)
        val_pred = svm.predict(X_val.numpy())
        val_acc = np.mean(y_val.numpy() == val_pred)
        results[(lr, reg)] = (train_acc, val_acc)
        if val_acc > best_val:
            best_lr = lr
            best_reg = reg
            best_val = val_acc
            best_svm = svm

# *****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 combo found was lr:', best_lr, '; with regularization:', best_reg)

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 5.000000e-03 reg 5.000000e-04 train accuracy: 0.405082 val accuracy: 0.393000
lr 5.000000e-03 reg 1.000000e-03 train accuracy: 0.402857 val accuracy: 0.392000
lr 5.000000e-03 reg 2.000000e-03 train accuracy: 0.401796 val accuracy: 0.395000
lr 5.000000e-03 reg 5.000000e-03 train accuracy: 0.397082 val accuracy: 0.403000
lr 1.000000e-02 reg 5.000000e-04 train accuracy: 0.406980 val accuracy: 0.408000
lr 1.000000e-02 reg 1.000000e-03 train accuracy: 0.407327 val accuracy: 0.397000
lr 1.000000e-02 reg 2.000000e-03 train accuracy: 0.404837 val accuracy: 0.392000
lr 1.000000e-02 reg 5.000000e-03 train accuracy: 0.407612 val accuracy: 0.407000
lr 2.000000e-02 reg 5.000000e-04 train accuracy: 0.405571 val accuracy: 0.393000
lr 2.000000e-02 reg 1.000000e-03 train accuracy: 0.406490 val accuracy: 0.402000
```

```
lr 2.000000e-02 reg 2.000000e-03 train accuracy: 0.408551 val accuracy: 0.400000
lr 2.000000e-02 reg 5.000000e-03 train accuracy: 0.404490 val accuracy: 0.393000
lr 3.000000e-02 reg 5.000000e-04 train accuracy: 0.404980 val accuracy: 0.401000
lr 3.000000e-02 reg 1.000000e-03 train accuracy: 0.399102 val accuracy: 0.382000
lr 3.000000e-02 reg 2.000000e-03 train accuracy: 0.389245 val accuracy: 0.386000
lr 3.000000e-02 reg 5.000000e-03 train accuracy: 0.399122 val accuracy: 0.381000
Best combo found was lr: 0.01 ; with regularization: 0.0005
best validation accuracy achieved: 0.408000
final test accuracy svm achieved: 0.377000
```

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

    # First fully-connected layer: compute linear combination
    z1 = X.mm(W1) + b1  # Shape: (N, H)

    # Compute the forward pass
    hidden = None
    scores = None
    def activation(z):
      # TODO: use sigmoid function [https://en.wikipedia.org/wiki/Sigmoid_function]
      # Produce hidden layer activations.
      return 1 / (1 + 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).                                                            #
    ############################################################################

    # Apply activation to get hidden layer features
    hidden = activation(z1)  # Shape: (N, H)

    # Second fully-connected layer: compute class scores
    scores = hidden.mm(W2) + b2  # Shape: (N, C)

    ###########################################################################
    #                              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/).   #
############################################################################
# Replace "pass" statement with your code

# Shift scores for numerical stability.
shifted_scores = scores - torch.max(scores, dim=1, keepdim=True)[0]
exp_scores = torch.exp(shifted_scores)
sum_exp = torch.sum(exp_scores, dim=1, keepdim=True)
probs = exp_scores / sum_exp

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

# Regularization loss (do not multiply by 1/2).
reg_loss = reg * (torch.sum(W1 * W1) + torch.sum(W2 * W2))
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                                                     #
    ###########################################################################
    # Replace "pass" statement with your code

    # Compute gradient on scores.
    dscores = probs.clone()
    dscores[torch.arange(N), y] -= 1
    dscores /= N  # scale gradients by the number of examples

    # Backprop into W2 and b2.
    dW2 = hidden.t().mm(dscores)  # (H, N) x (N, C) -> (H, C)
    db2 = torch.sum(dscores, dim=0)  # (C,)

    # Backprop into hidden layer.
    dhidden = dscores.mm(W2.t())  # (N, C) x (C, H) -> (N, H)

    # Backprop through the sigmoid activation.
    # Sigmoid derivative: sigmoid(x) * (1 - sigmoid(x))
    dz1 = dhidden * hidden * (1 - hidden)  # (N, H)

    # Backprop into W1 and b1.
    dW1 = X.t().mm(dz1)  # (D, N) x (N, H) -> (D, H)
    db1 = torch.sum(dz1, dim=0)  # (H,)

    # Add regularization gradient.
    dW2 += 2 * reg * W2
    dW1 += 2 * reg * W1

    # Store gradients in the grads dictionary.
    grads['W1'] = dW1
    grads['b1'] = db1
    grads['W2'] = dW2
    grads['b2'] = db2

    ###########################################################################
    #                              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 key in params:
    params[key] -= learning_rate * grads[key]
```

```
    ##########################################################################
    #                             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)

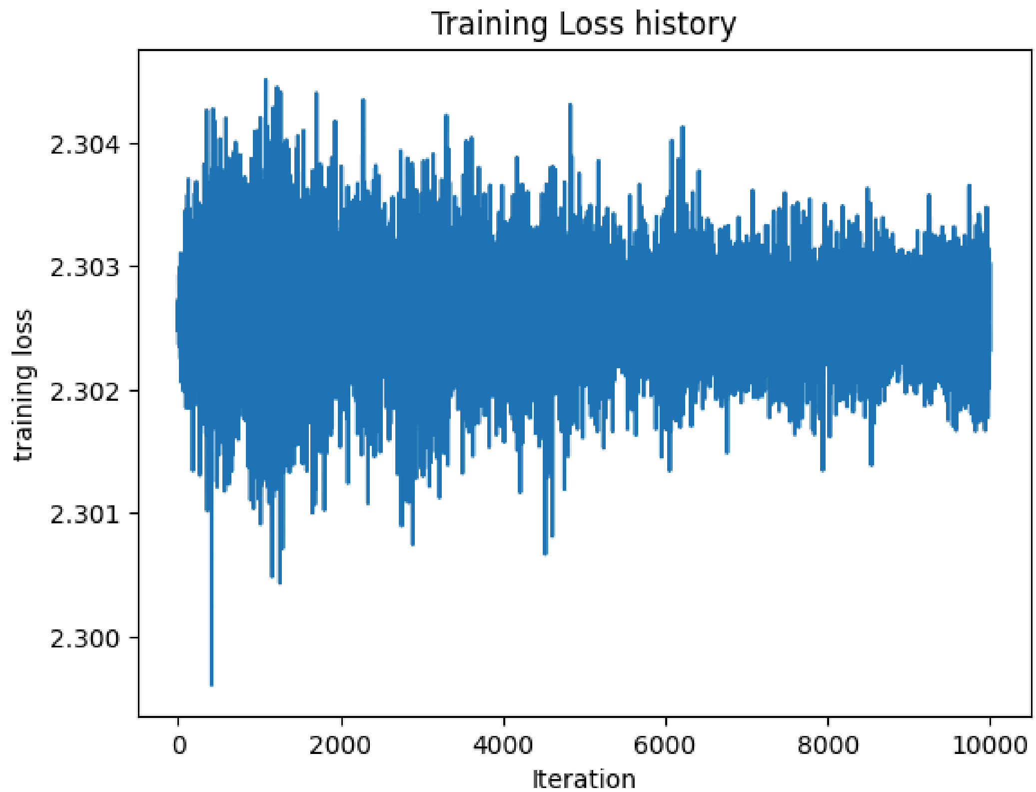
```
def nn_predict(params, loss_func, X):
  """
  Use the trained weights of this two-layer network to predict labels for
  data points. For each data point we predict scores for each of the C
  classes, and assign each data point to the class with the highest score.

  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!                #
```

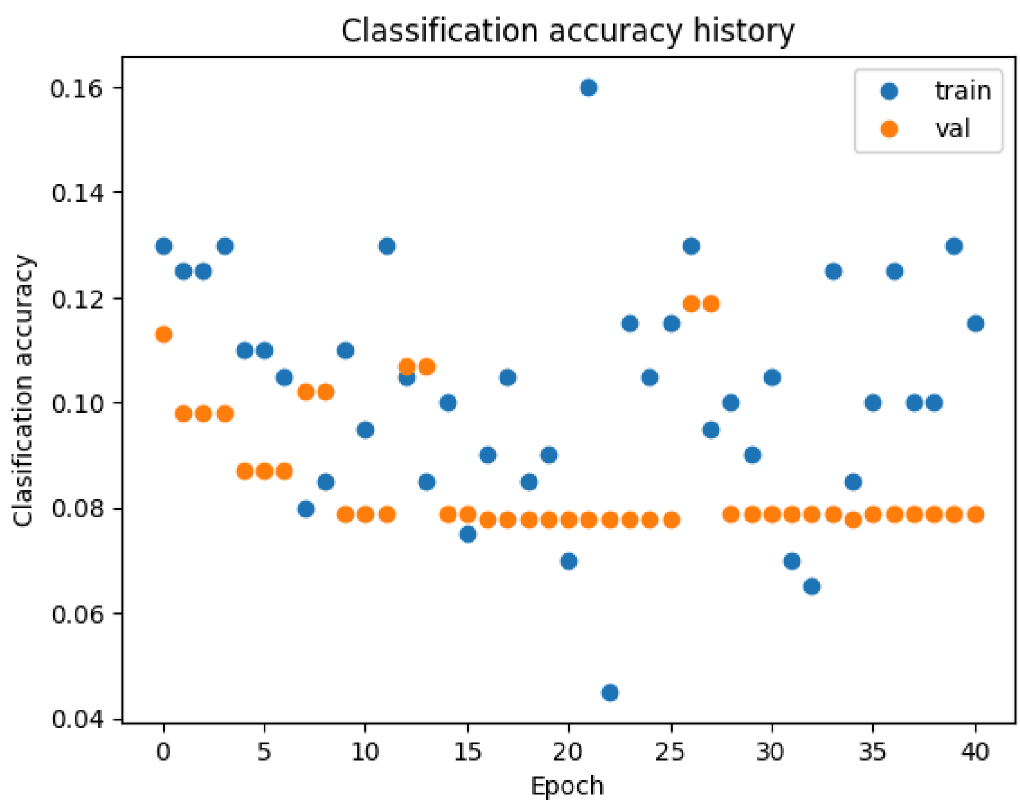
```
    ###########################################################################
    # Replace "pass" statement with your code

    scores, _ = nn_forward_pass(params, X)
    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 101.880891s
Final training loss:  2.3030202388763428
```





Hyperparameters tuning (10 pts)

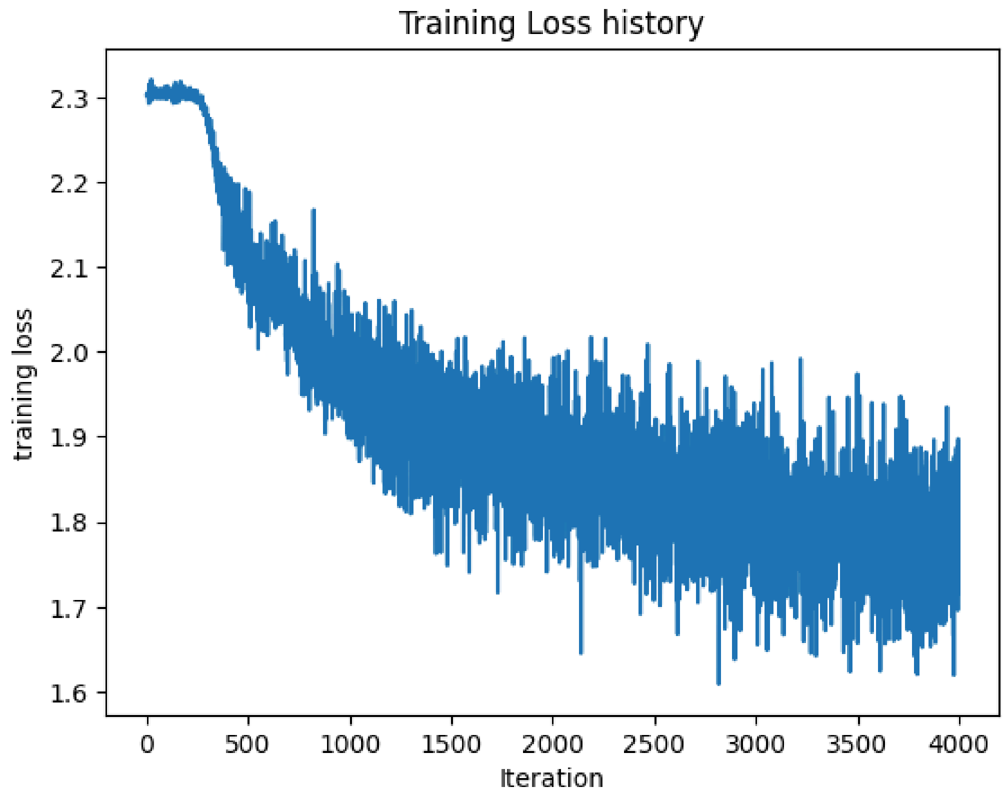
Tuning. Tuning the hyperparameters and developing intuition for how they affect the final performance is a large part of using Neural Networks, so we want you to get a lot of practice. Below, you should experiment with different values of the various hyperparameters, including hidden layer size, learning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

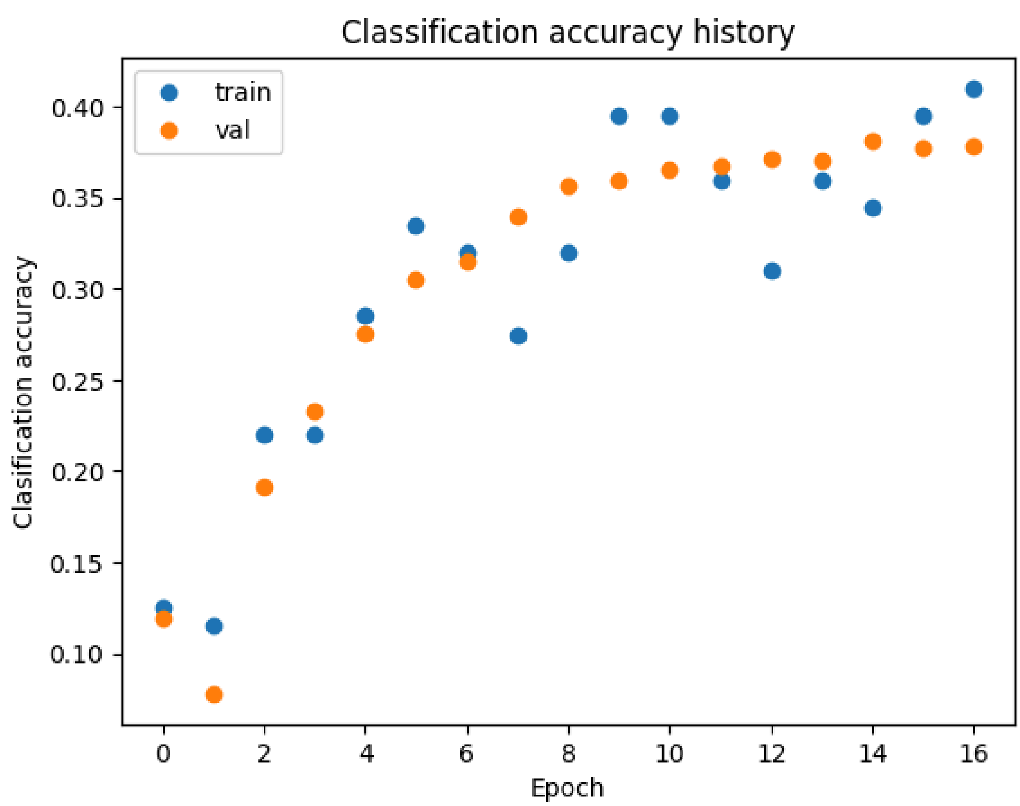
Approximate results. To get full credit for the assignment, you should achieve a classification accuracy above 50% on the validation set.

```
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)*****
# fill in your own values
learning_rates = [0.1, 0.5, 0.9] # [1e-3, 1e-2, 1e-1]
regularization_strengths = [5e-06, 1e-05, 5e-05, 1e-04] # [1e-5, 1e-4, 1e-3]
hidden_dims = [50, 100, 200] # [25, 50, 75]

num_iters = 4000 # 2000  # Use a reasonable number of iterations
batch_size = 200
learning_rate_decay = 0.95

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

print("Starting grid search over hyperparameters...\n")
for lr in learning_rates:
    for reg in regularization_strengths:
        for H in hidden_dims:
            print("------------------------------------------------")
            print("Training NN with lr = %e, reg = %e, hidden_dim = %d" % (lr, reg, H))
            model = TwoLayerNet(input_size=X_train.shape[1], hidden_size=H, output_size=10,
                                dtype=torch.float32, device='cpu', std=1e-4)
            stats = model.train(X_train, y_train, X_val, y_val,
                                learning_rate=lr,
                                learning_rate_decay=learning_rate_decay,
                                reg=reg,
                                num_iters=num_iters,
                                batch_size=batch_size,
                                verbose=False)
            train_acc = stats['train_acc_history'][-1]
            val_acc = stats['val_acc_history'][-1]
            # print("Finished training with lr = %e, reg = %e, hidden_dim = %d" % (lr, reg, H)
            print("Train accuracy: %f, Val accuracy: %f" % (train_acc, val_acc))

            # Visualize loss and accuracy history for this run.
            visualization(stats)

            results[(lr, reg, H)] = (train_acc, val_acc)
            if val_acc > best_val:
                best_val = val_acc
                best_nn = model
            print("------------------------------------------------\n")
```

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

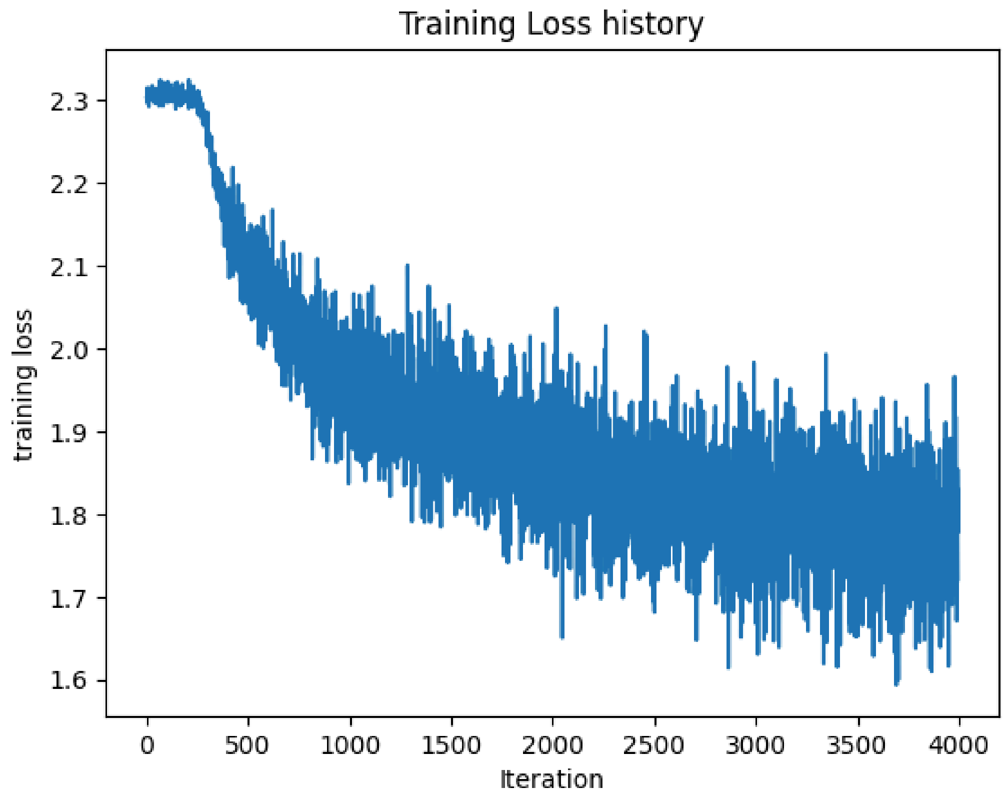
```
Starting grid search over hyperparameters...

--------------------------------------------------
Training NN with lr = 1.000000e-01, reg = 5.000000e-06, hidden_dim = 50
Train accuracy: 0.410000, Val accuracy: 0.378000
Final training loss:  1.7701492309570312
```

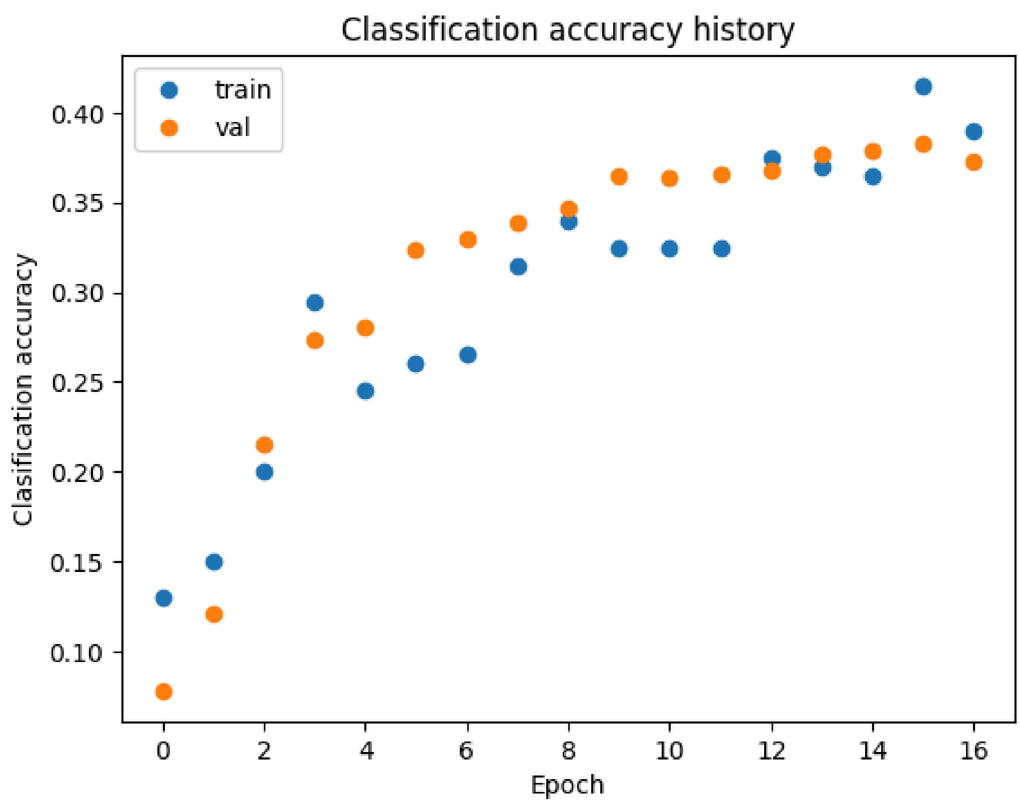





```
--------------------------------------------------

--------------------------------------------------
Training NN with lr = 1.000000e-01, reg = 5.000000e-06, hidden_dim = 100
```

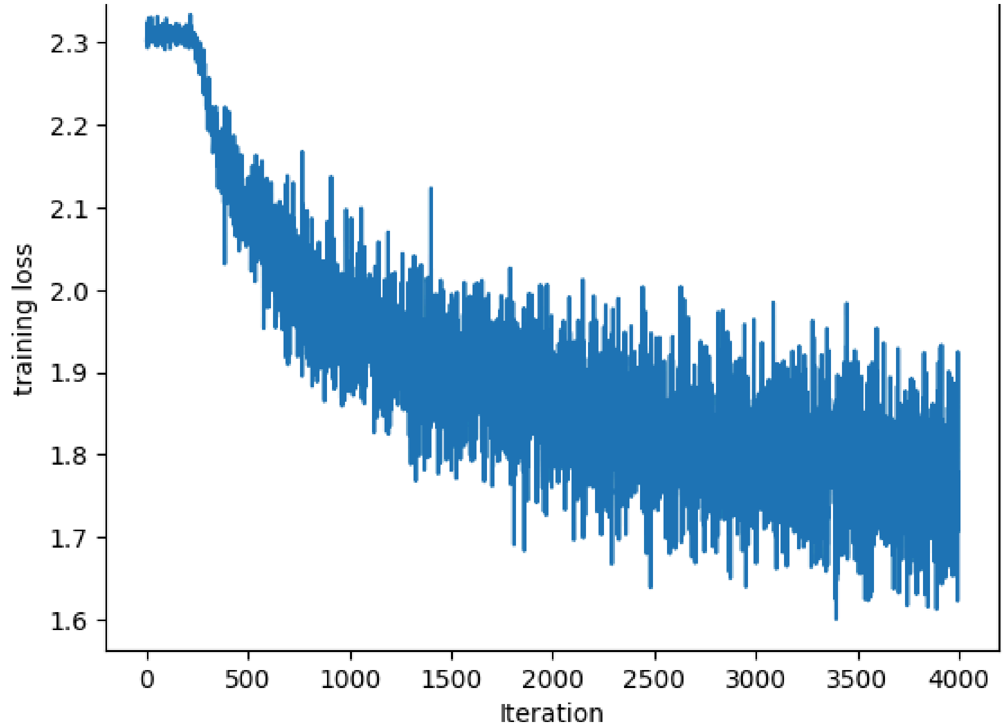
```
Train accuracy: 0.390000, Val accuracy: 0.373000
Final training loss:  1.828048825263977
```

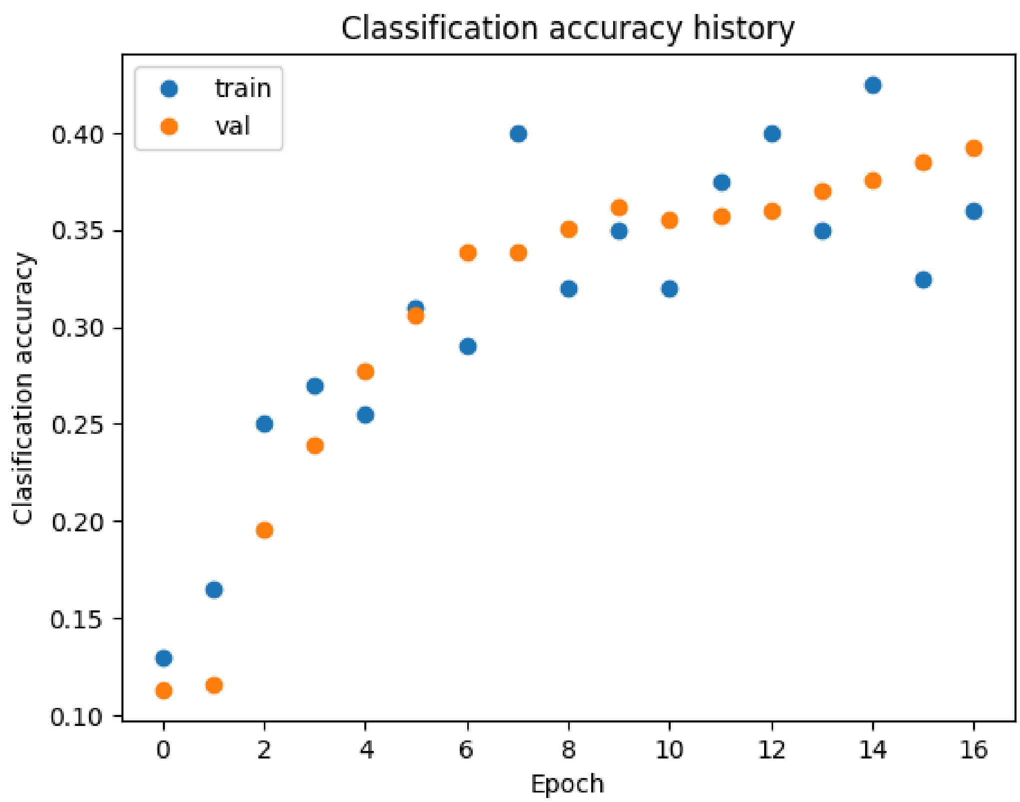




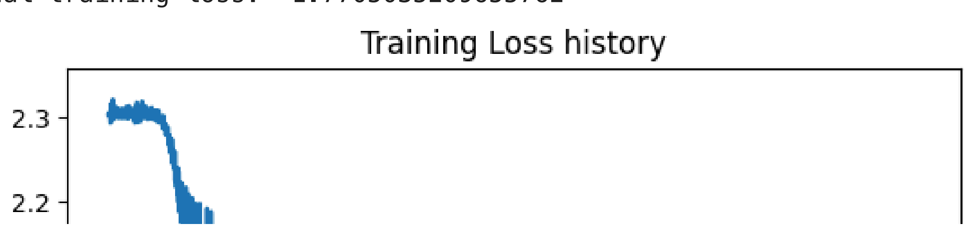

```
--------------------------------------------------

--------------------------------------------------
Training NN with lr = 1.000000e-01, reg = 5.000000e-06, hidden_dim = 200
Train accuracy: 0.360000, Val accuracy: 0.392000
Final training loss:  1.7081058025360107
```

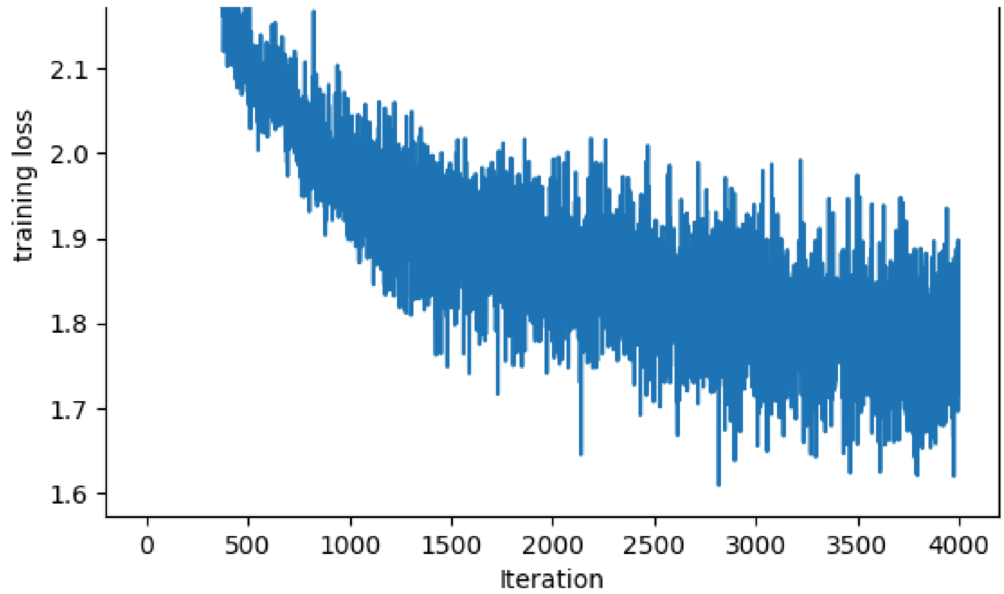
Training Loss history



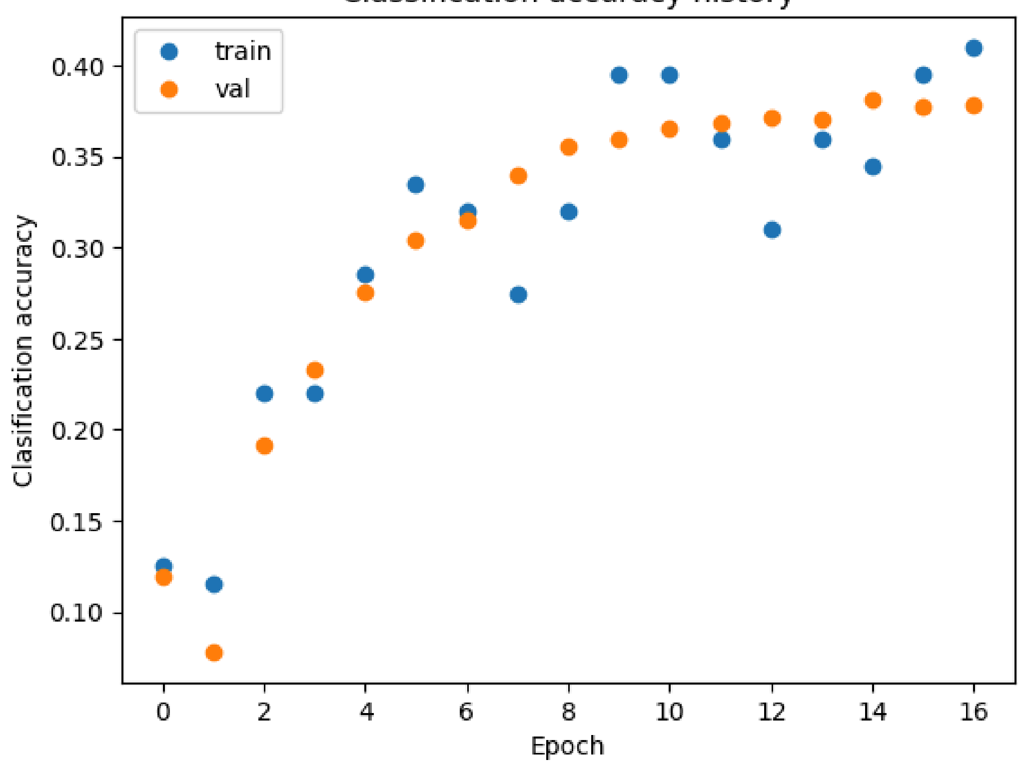



```
--------------------------------------------------

--------------------------------------------------
Training NN with lr = 1.000000e-01, reg = 1.000000e-05, hidden_dim = 50
Train accuracy: 0.410000, Val accuracy: 0.378000
Final training loss:  1.7705035209655762
```



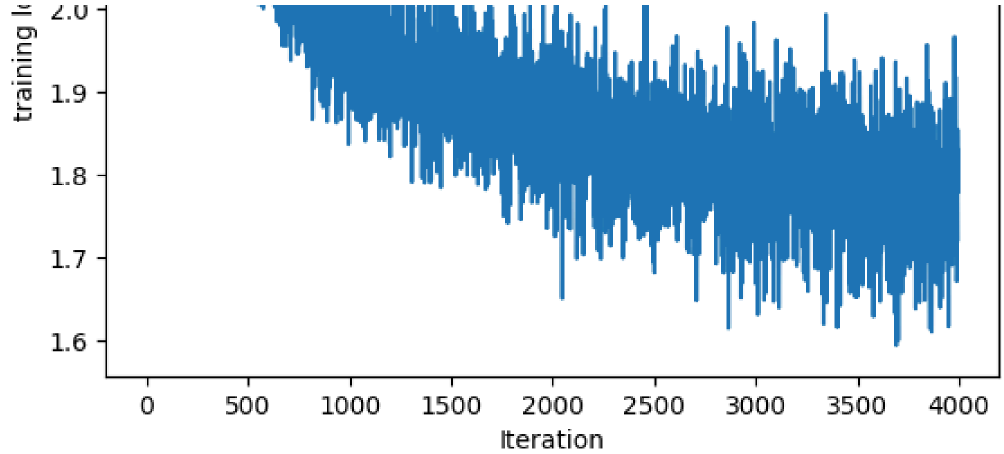


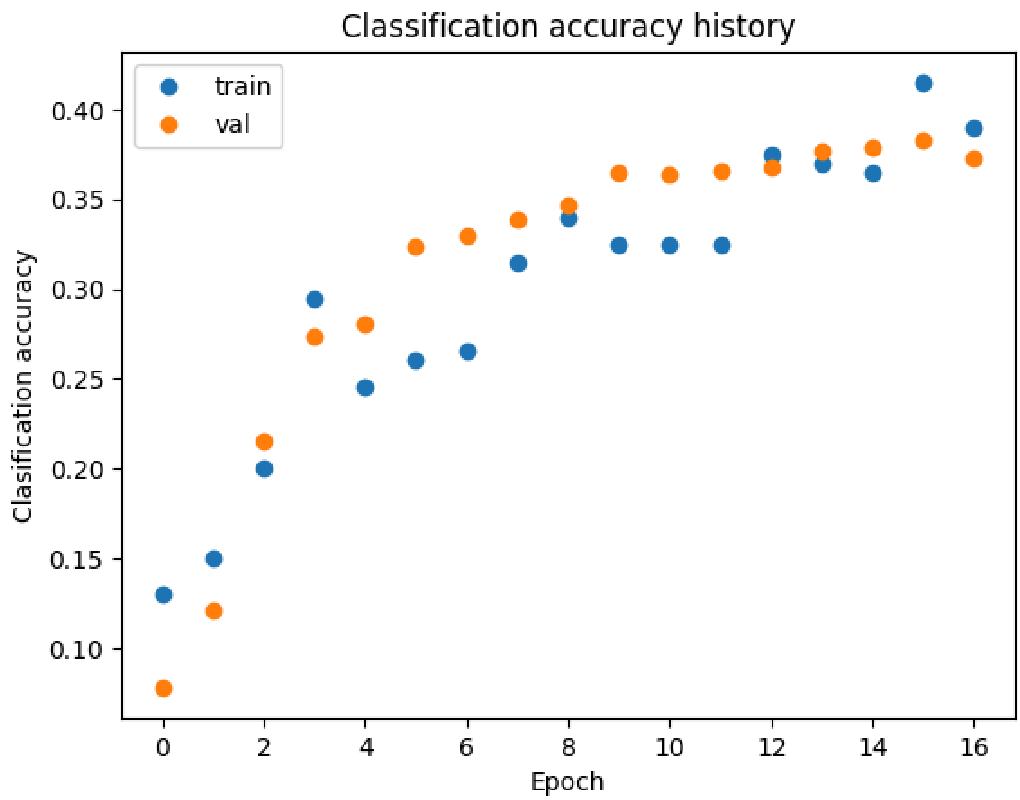
Classification accuracy history



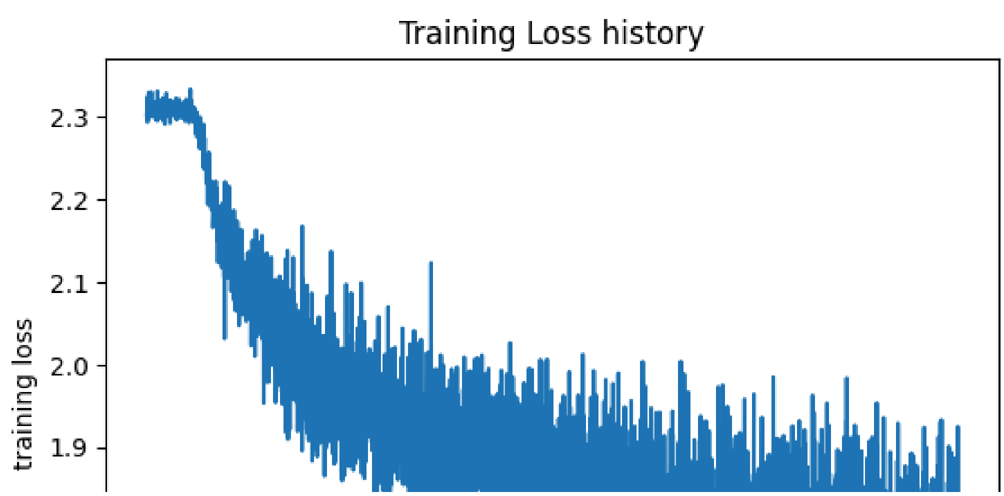

```
------------------------------------------------

------------------------------------------------
Training NN with lr = 1.000000e-01, reg = 1.000000e-05, hidden_dim = 100
Train accuracy: 0.390000, Val accuracy: 0.373000
Final training loss:  1.8283641338348389
```

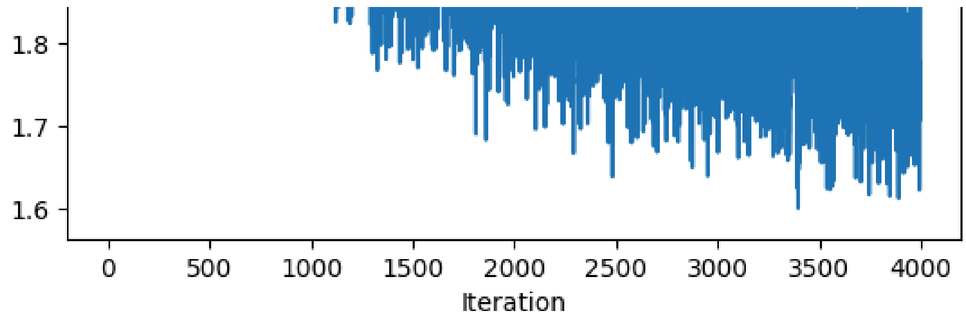
Training Loss history

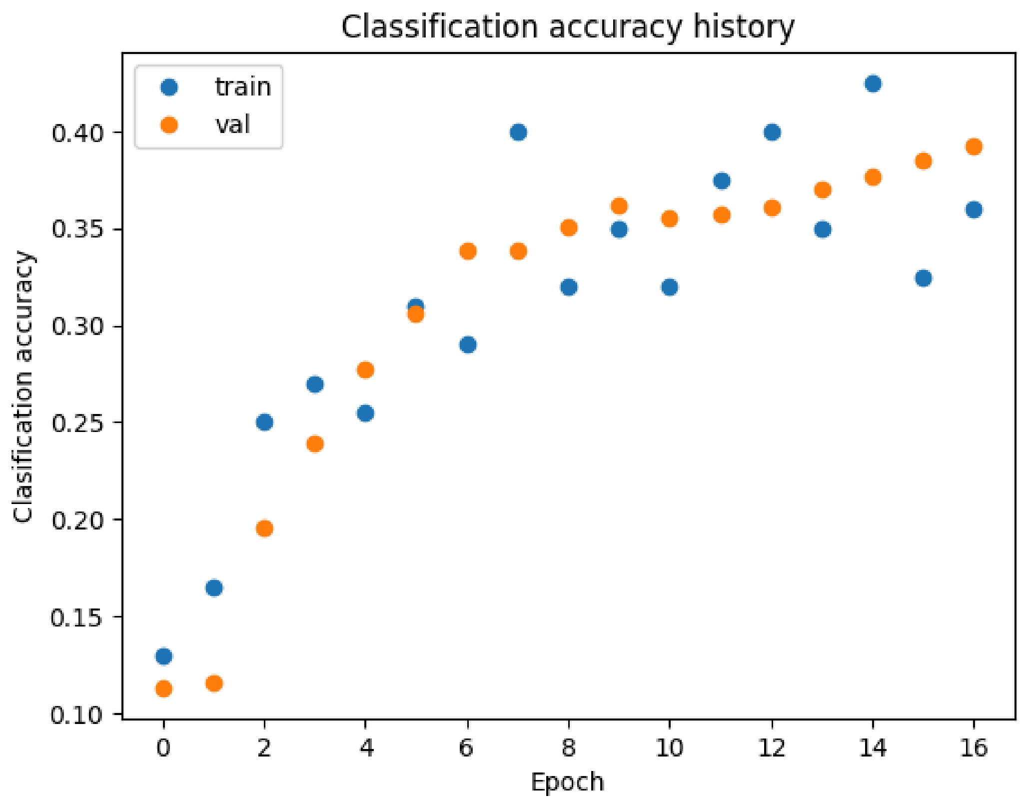




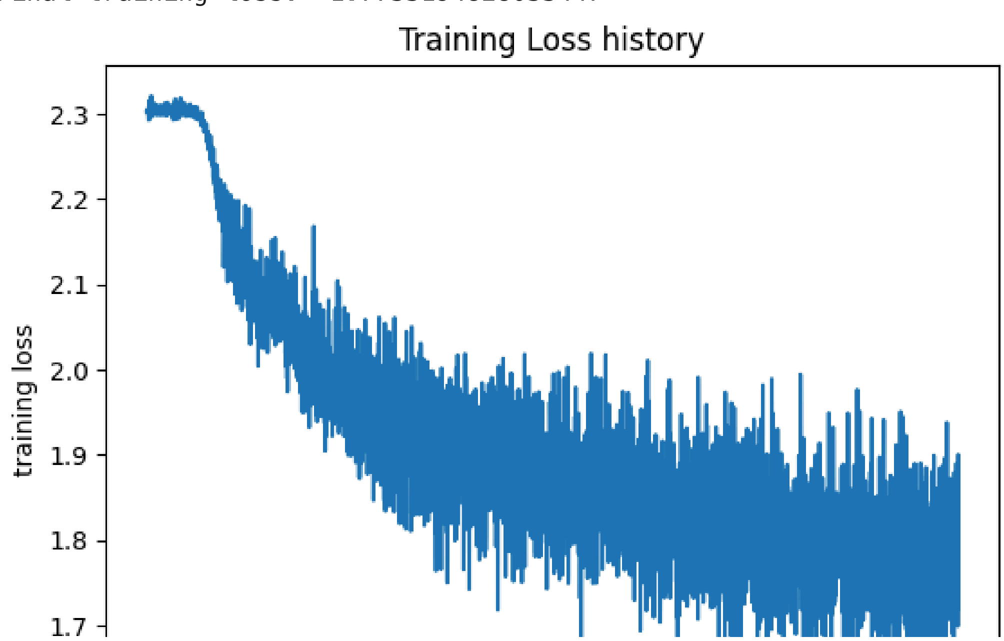

```
------------------------------------------------

------------------------------------------------
Training NN with lr = 1.000000e-01, reg = 1.000000e-05, hidden_dim = 200
Train accuracy: 0.360000, Val accuracy: 0.392000
Final training loss:  1.7084879875183105
```

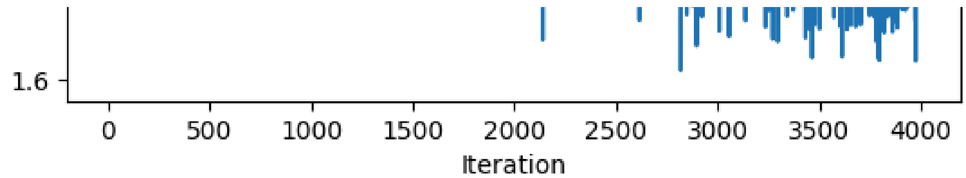


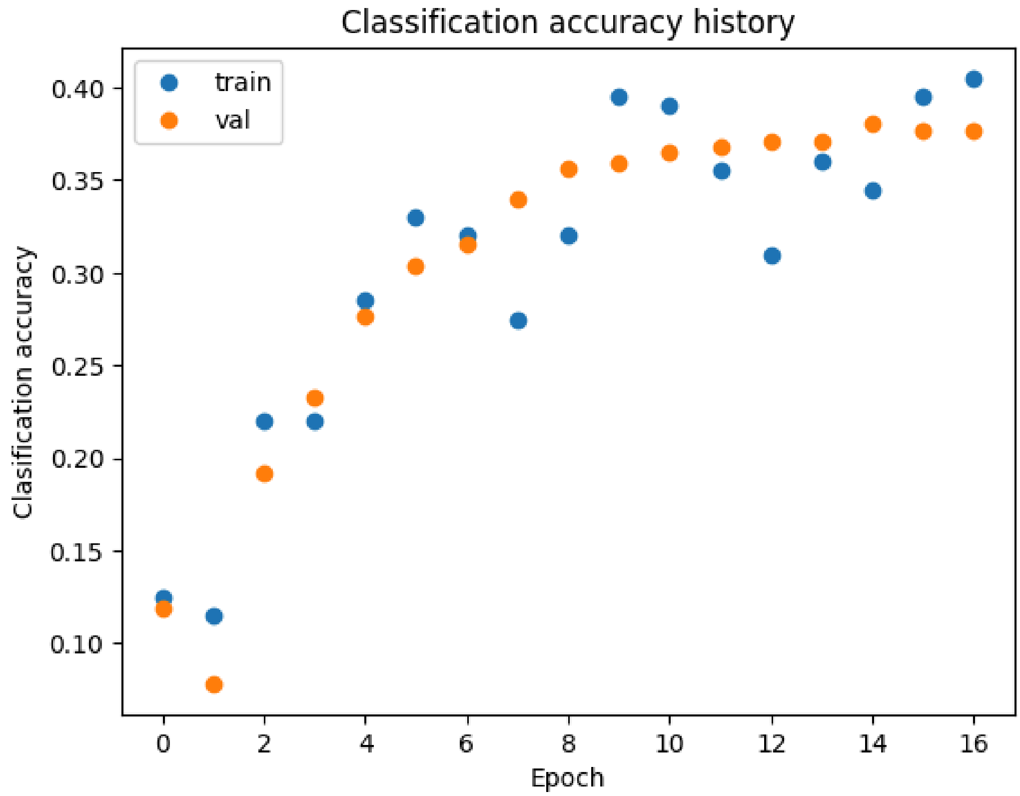




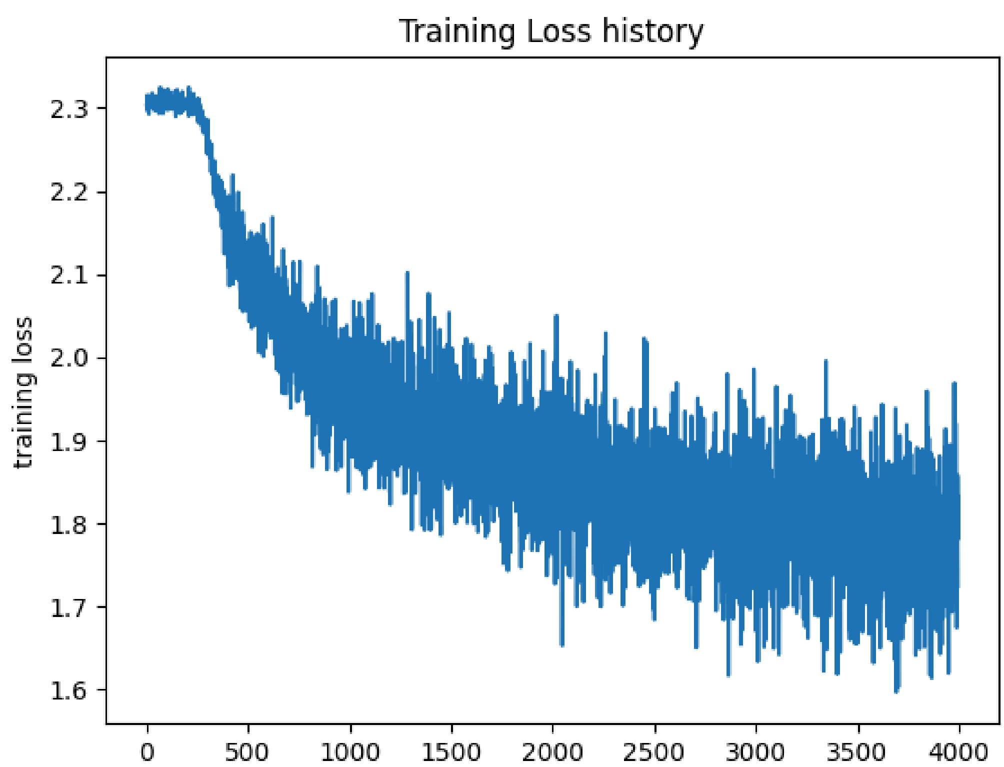

```
------------------------------------------------

------------------------------------------------
Training NN with lr = 1.000000e-01, reg = 5.000000e-05, hidden_dim = 50
Train accuracy: 0.405000, Val accuracy: 0.377000
Final training loss:  1.7733194828033447
```



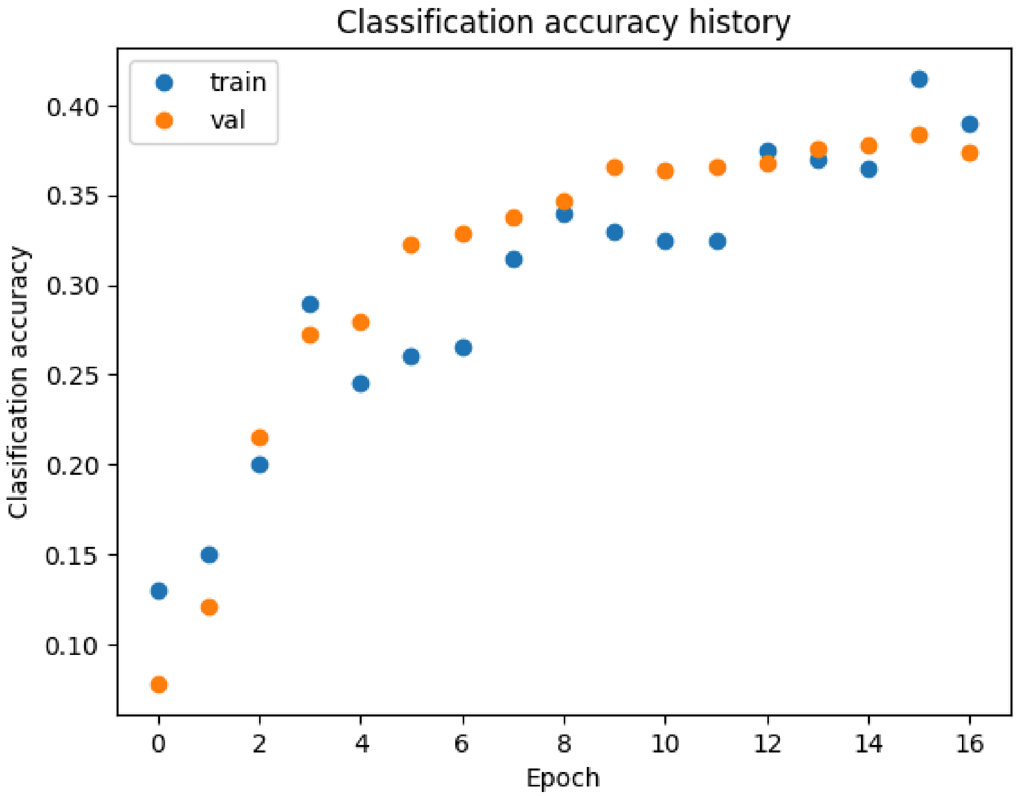




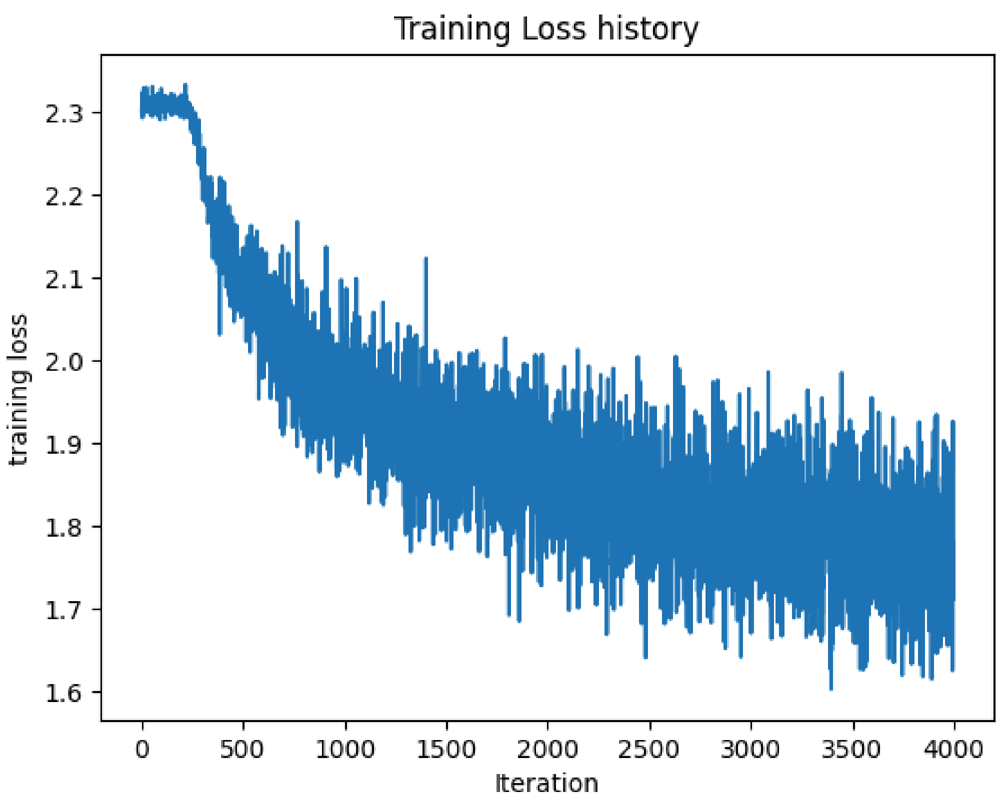

```
------------------------------------------------

------------------------------------------------
Training NN with lr = 1.000000e-01, reg = 5.000000e-05, hidden_dim = 100
Train accuracy: 0.390000, Val accuracy: 0.374000
Final training loss:  1.8308676481246948
```

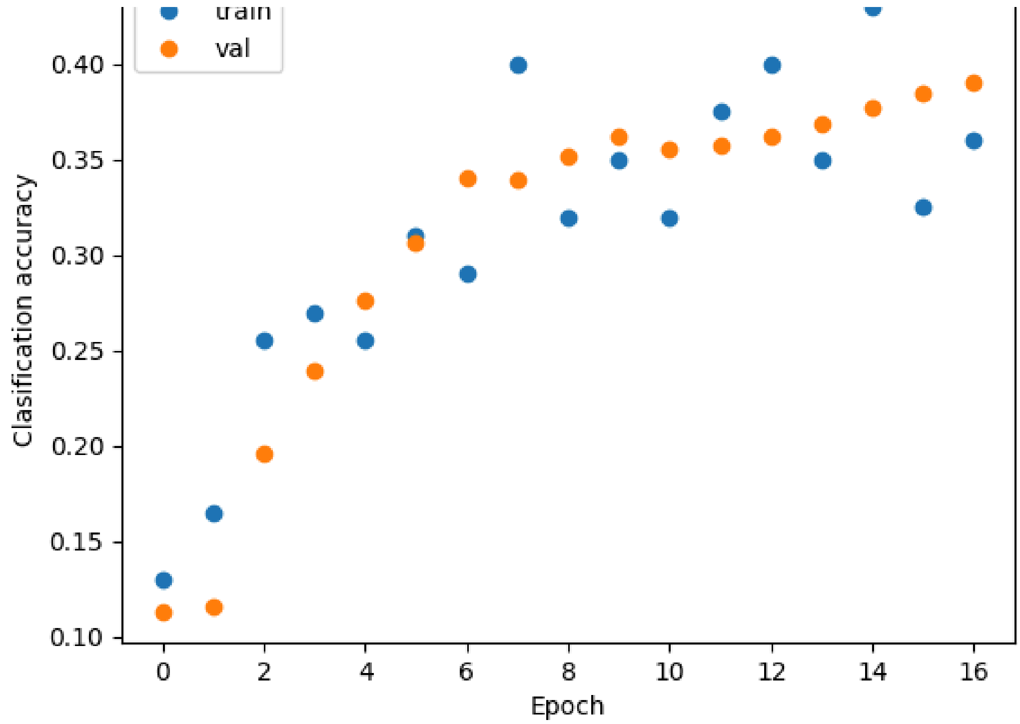


Iteration

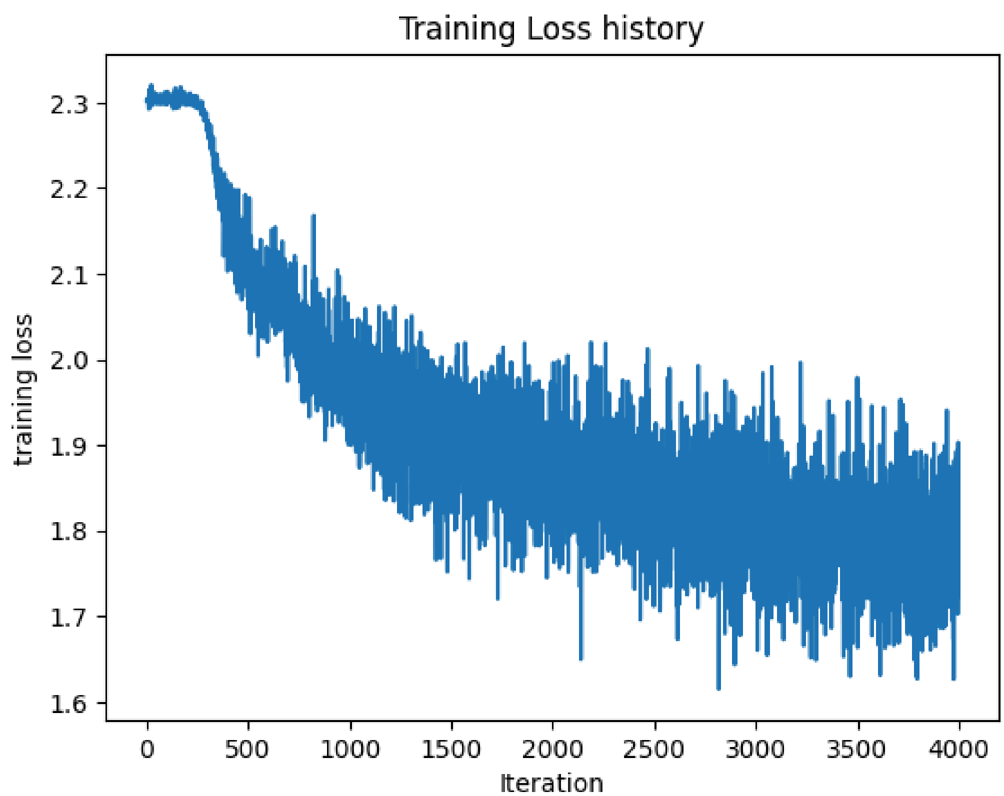


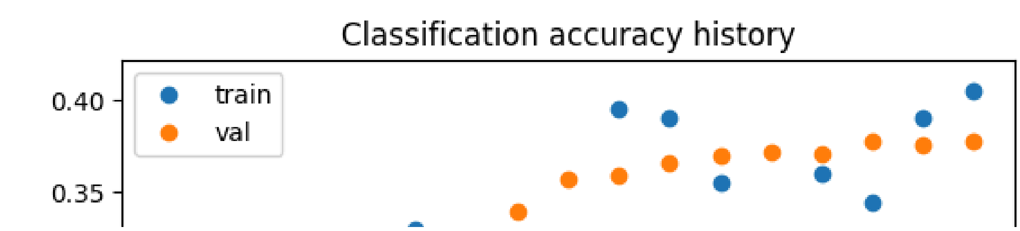

```
------------------------------------------------

------------------------------------------------
Training NN with lr = 1.000000e-01, reg = 5.000000e-05, hidden_dim = 200
Train accuracy: 0.360000, Val accuracy: 0.390000
Final training loss:  1.7115272283554077
```

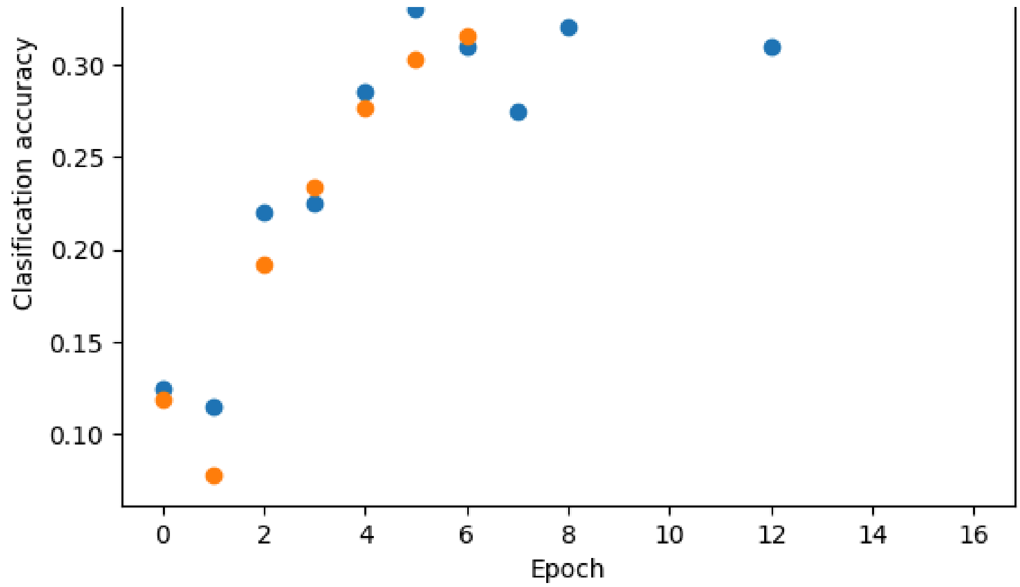




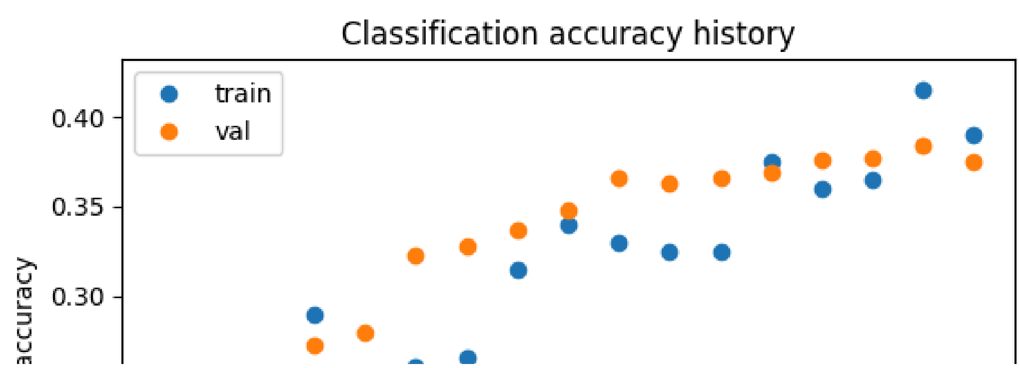

```
-------------------------------------------------

-------------------------------------------------
Training NN with lr = 1.000000e-01, reg = 1.000000e-04, hidden_dim = 50
Train accuracy: 0.405000, Val accuracy: 0.378000
Final training loss:  1.7767891883850098
```

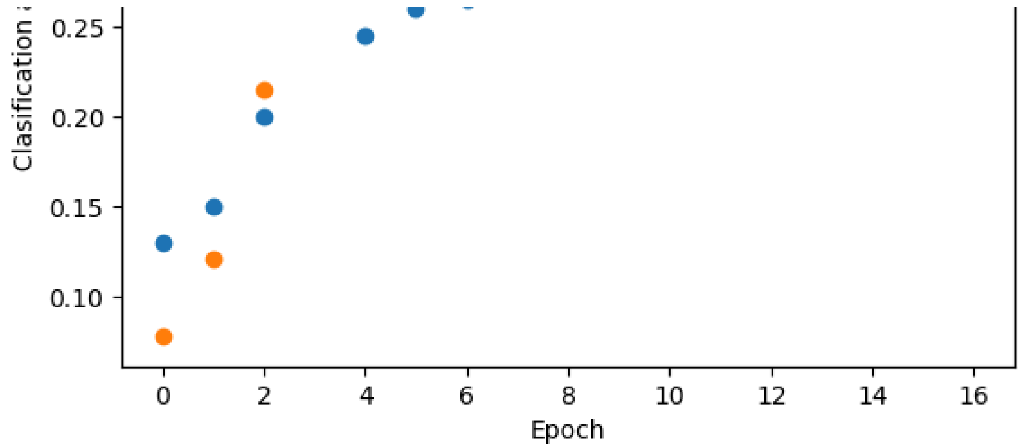




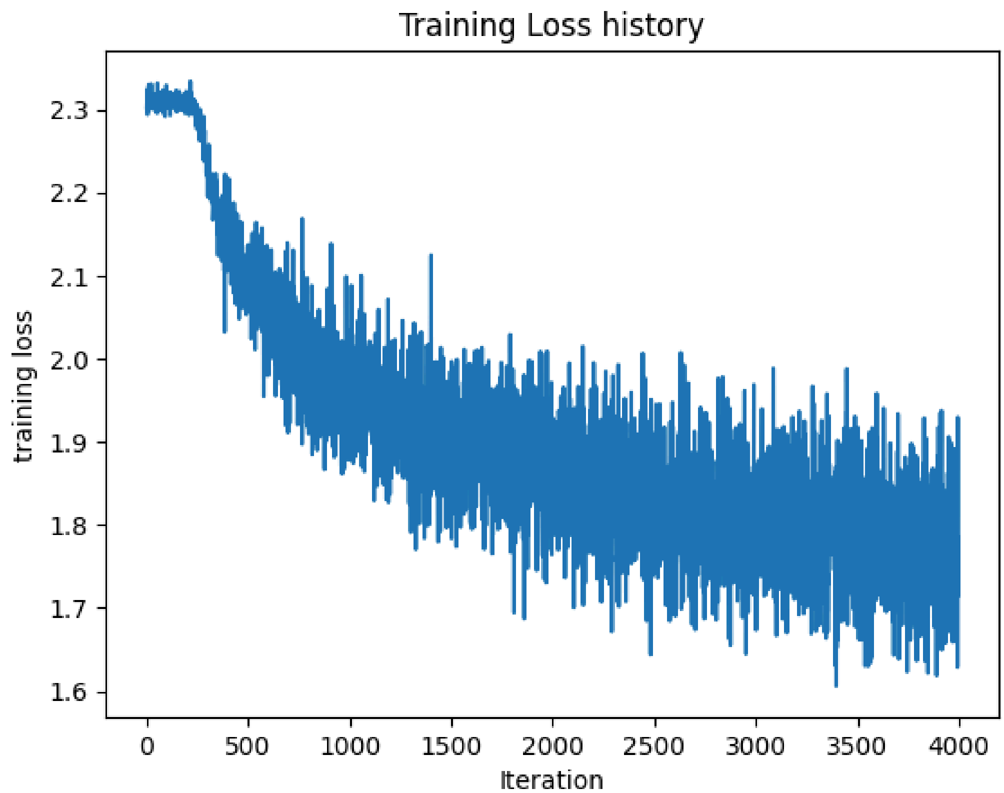


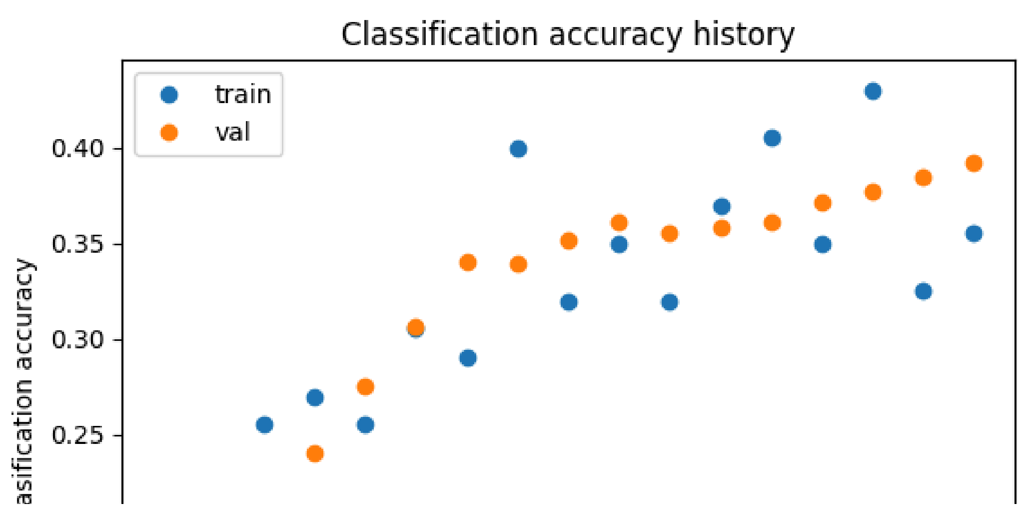

```
----------------------------------------------

----------------------------------------------
Training NN with lr = 1.000000e-01, reg = 1.000000e-04, hidden_dim = 100
Train accuracy: 0.390000, Val accuracy: 0.375000
Final training loss:  1.8339554071426392
```

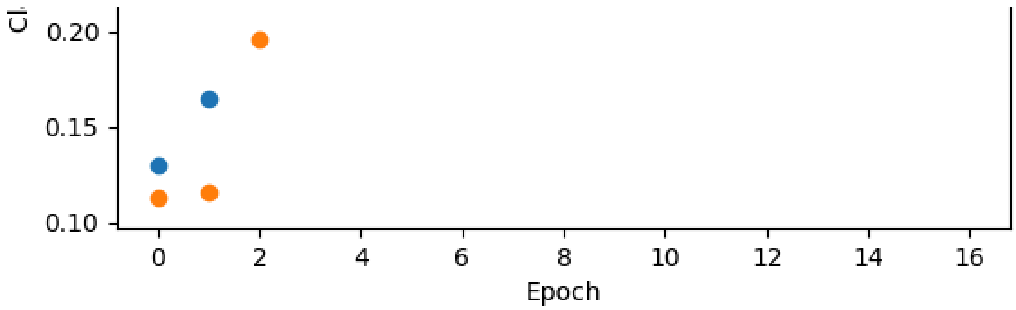




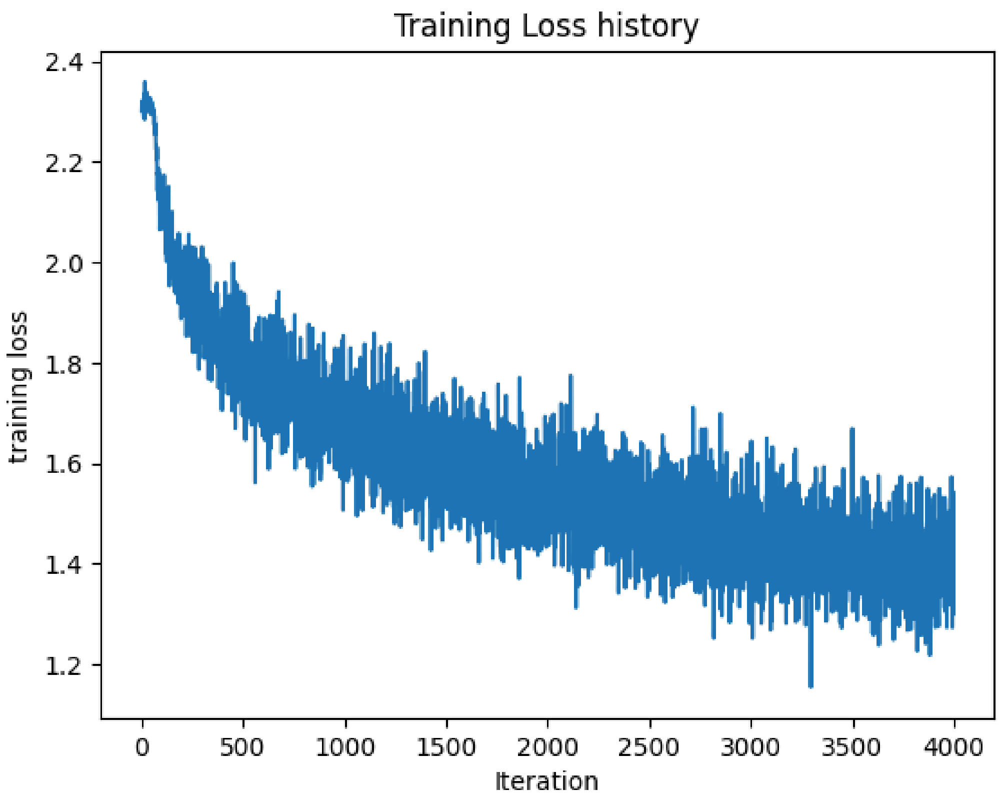

```
--------------------------------------------------

--------------------------------------------------
Training NN with lr = 1.000000e-01, reg = 1.000000e-04, hidden_dim = 200
Train accuracy: 0.355000, Val accuracy: 0.392000
Final training loss:  1.7152793407440186
```

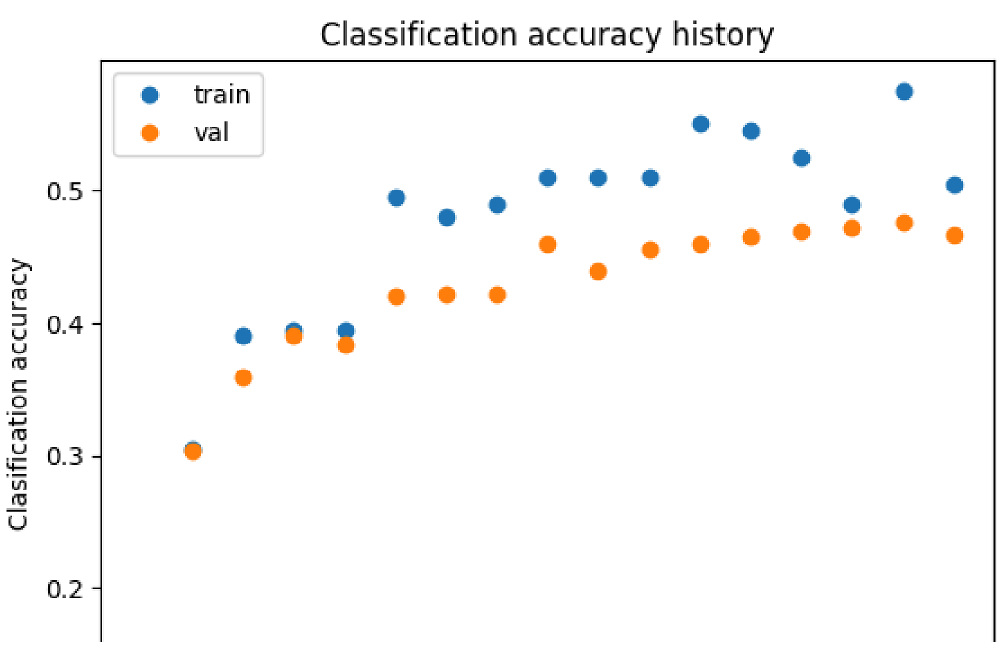


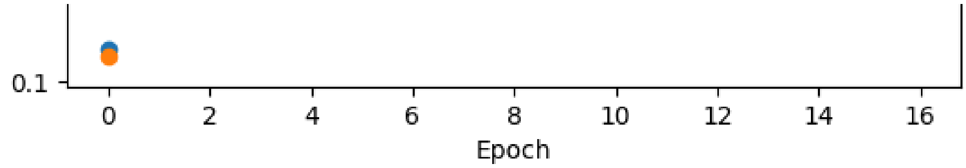




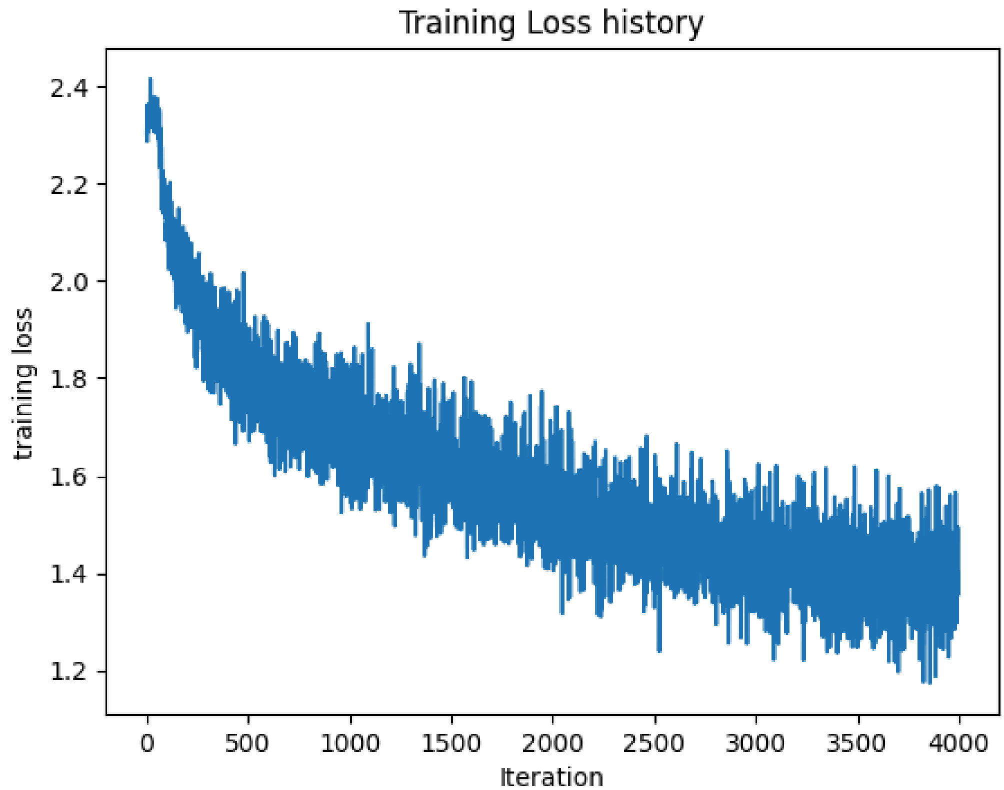

```
-----------------------------------------------

-----------------------------------------------
Training NN with lr = 5.000000e-01, reg = 5.000000e-06, hidden_dim = 50
Train accuracy: 0.505000, Val accuracy: 0.467000
Final training loss:  1.4457263946533203
```

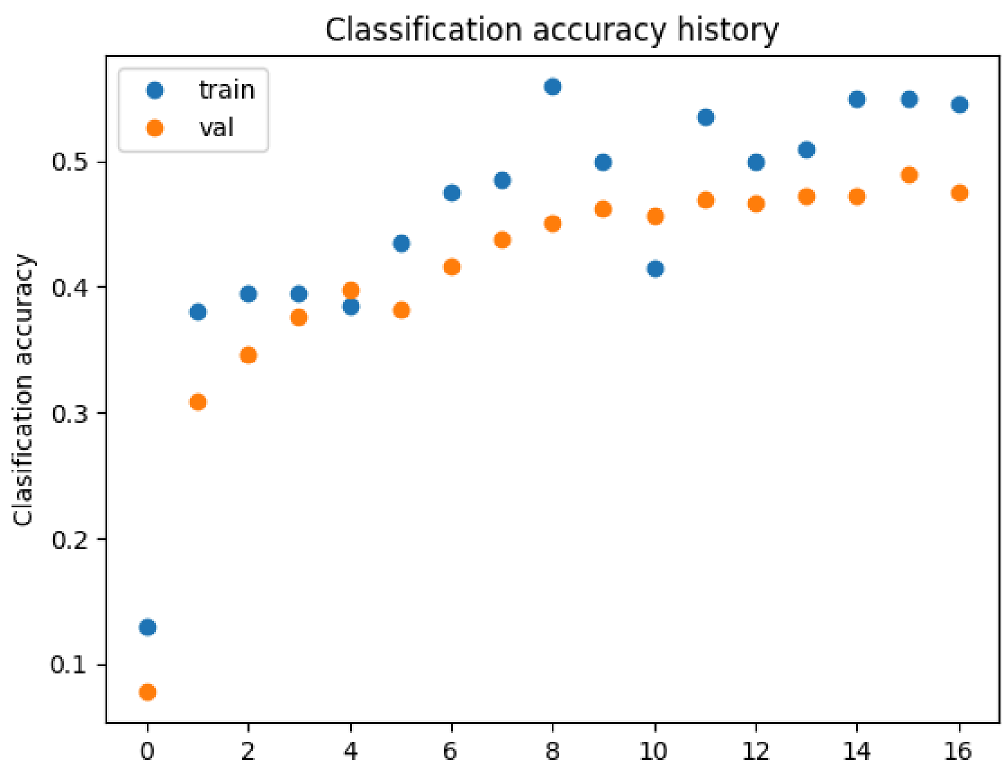






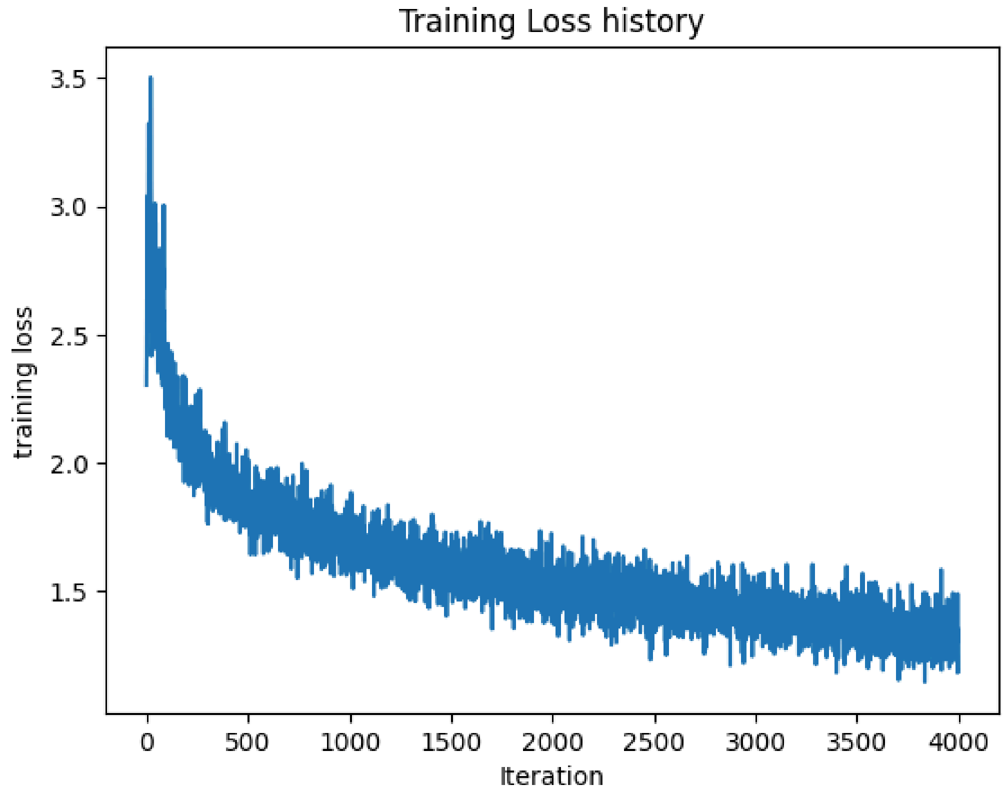

```
--------------------------------------------------

--------------------------------------------------
Training NN with lr = 5.000000e-01, reg = 5.000000e-06, hidden_dim = 100
Train accuracy: 0.545000, Val accuracy: 0.475000
Final training loss:  1.402327537536621
```

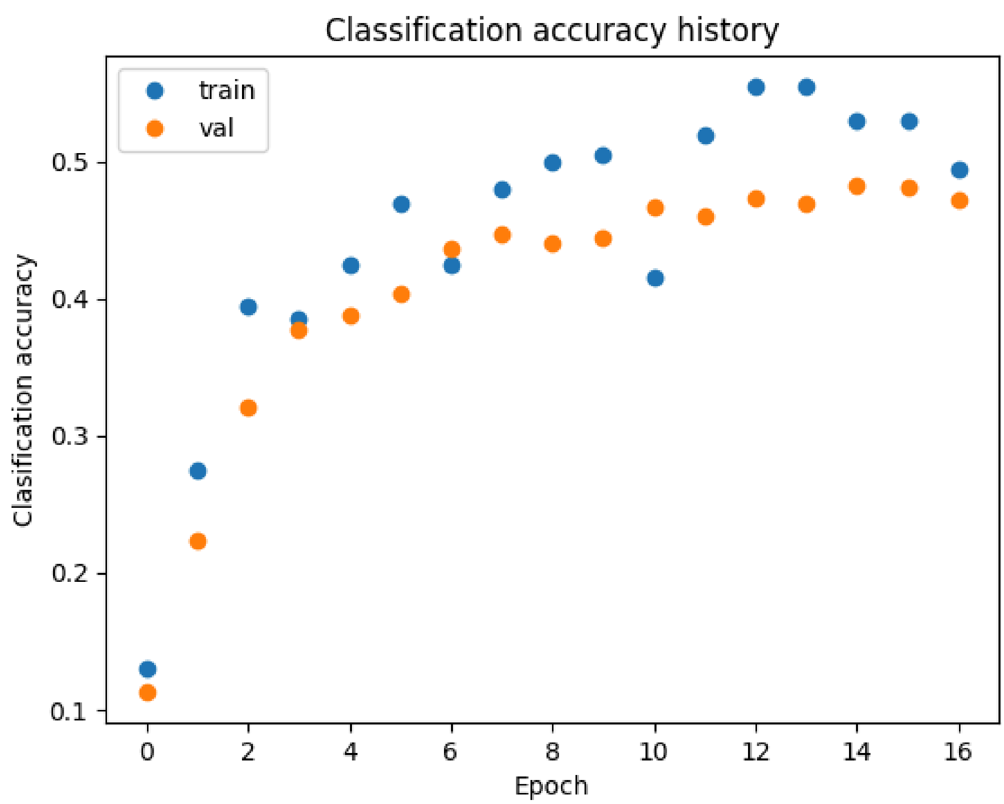




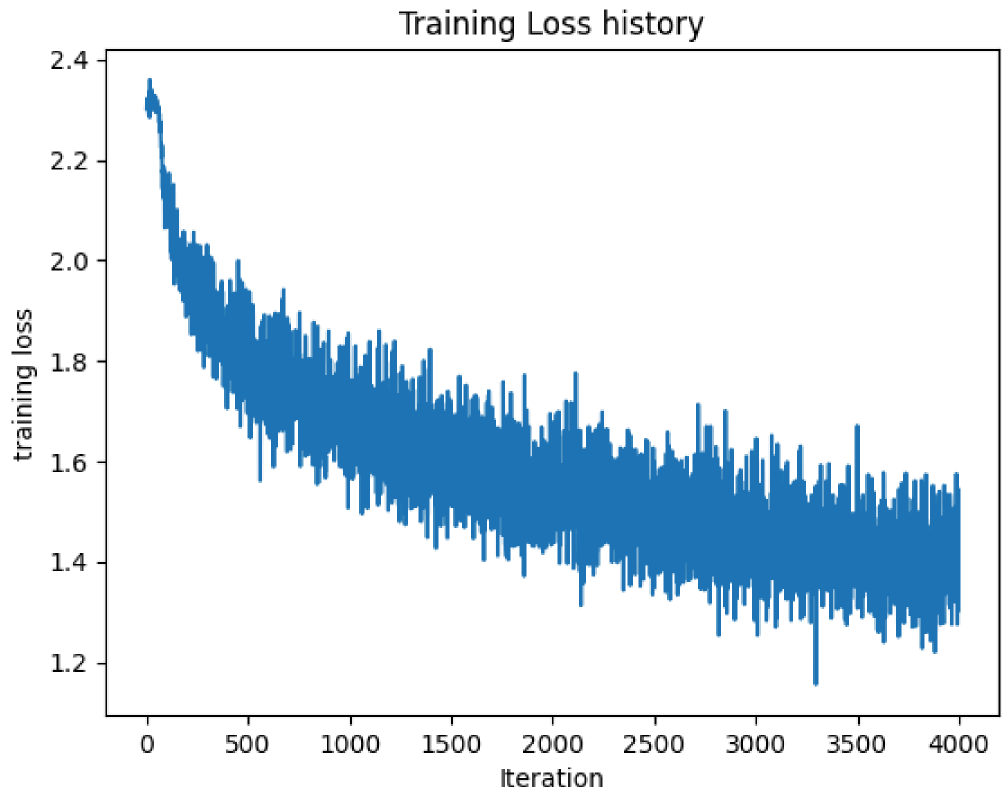
Epoch

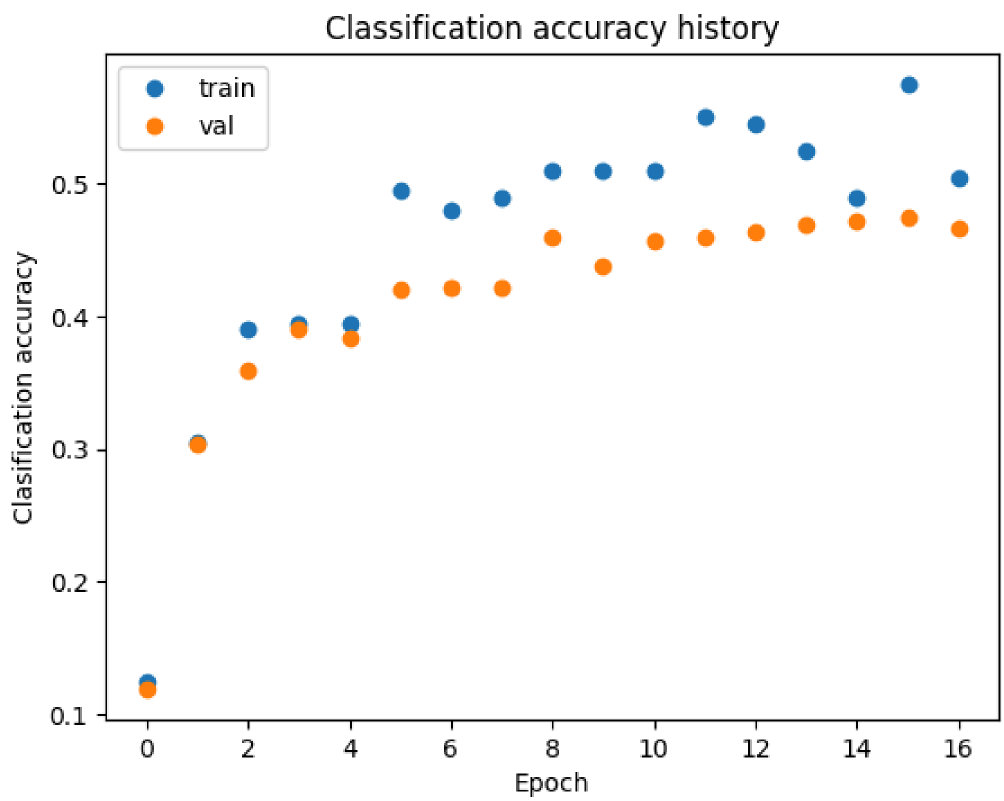
```
Training NN with lr = 5.000000e-01, reg = 5.000000e-06, hidden_dim = 200
Train accuracy: 0.495000, Val accuracy: 0.472000
Final training loss:  1.1863486766815186
```





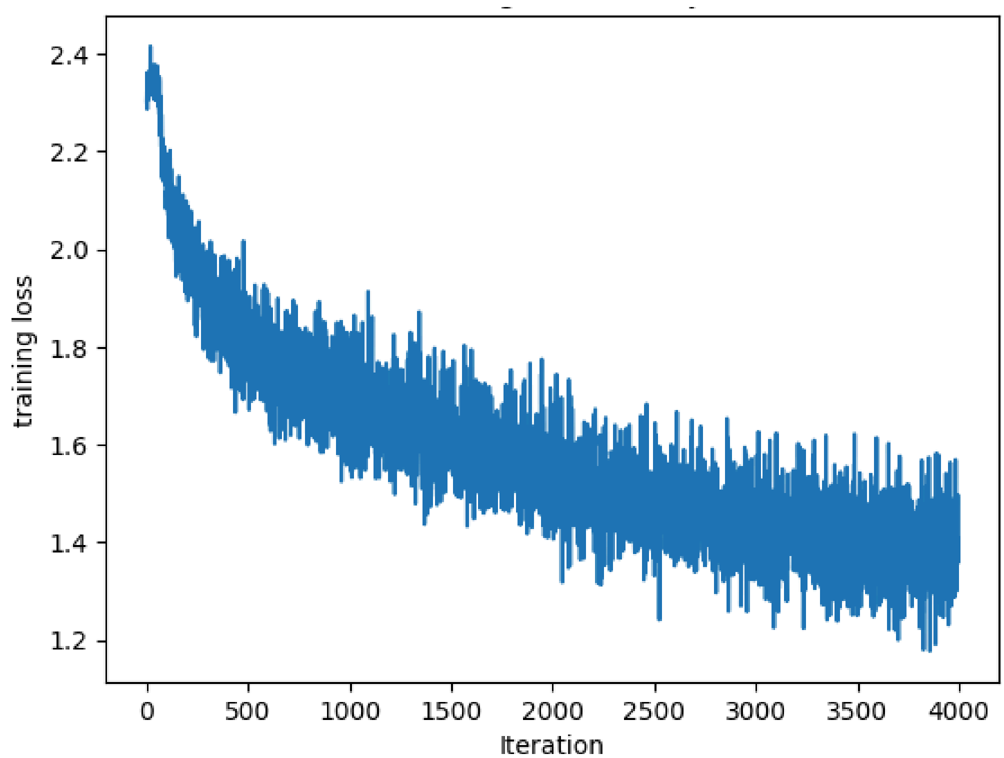
```
Training NN with lr = 5.000000e-01, reg = 1.000000e-05, hidden_dim = 50
Train accuracy: 0.505000, Val accuracy: 0.466000
Final training loss:  1.4485291242599487
```

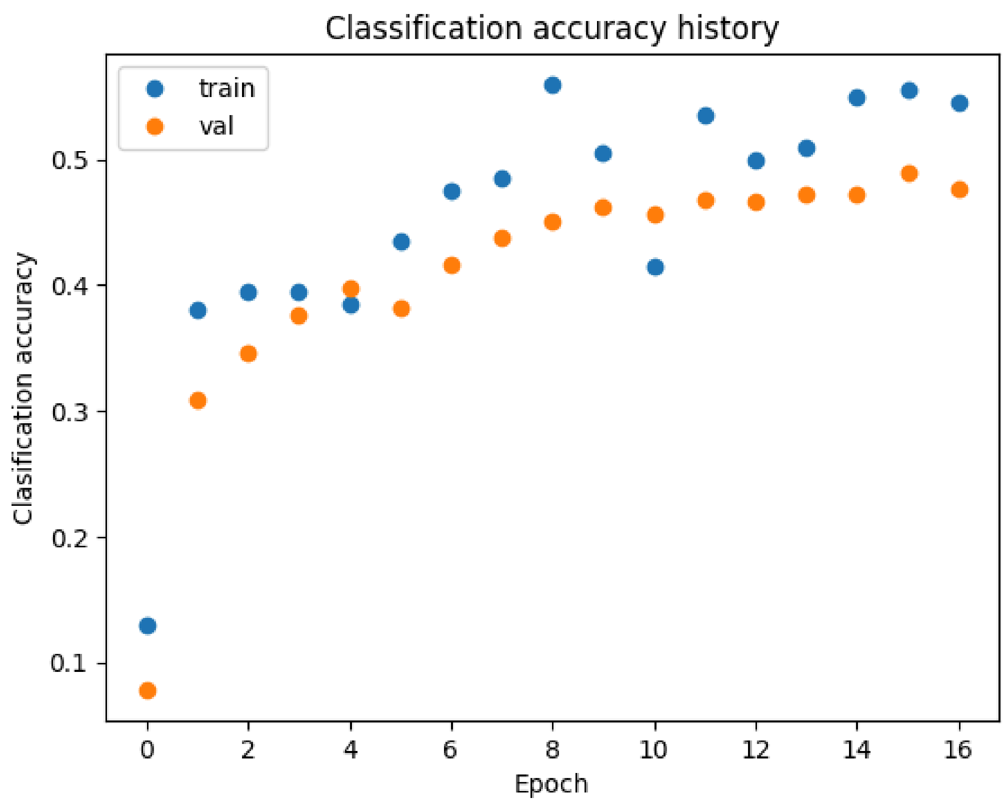




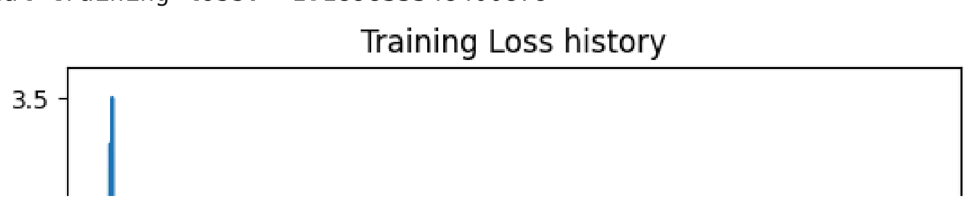

```
------------------------------------------------

------------------------------------------------
Training NN with lr = 5.000000e-01, reg = 1.000000e-05, hidden_dim = 100
Train accuracy: 0.545000, Val accuracy: 0.477000
Final training loss:  1.4054808616638184
```

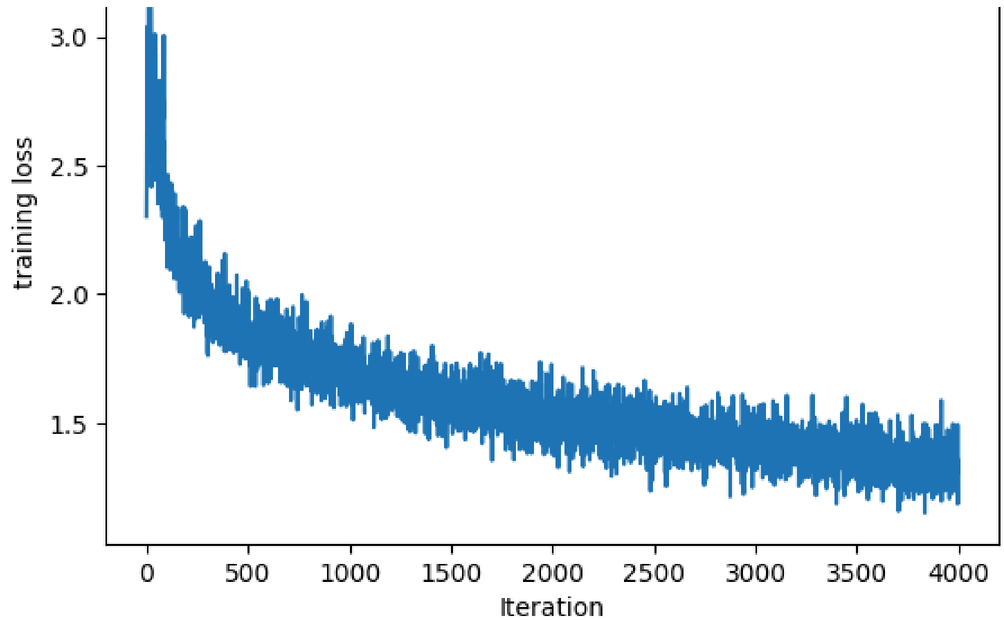
Training Loss history



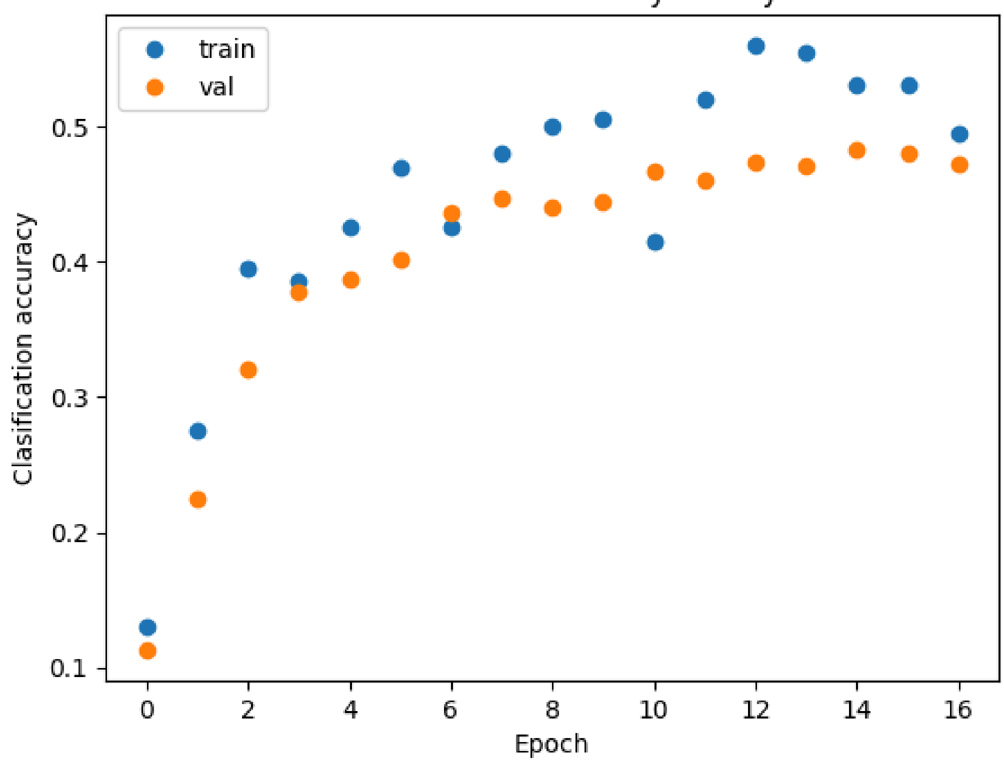



```
------------------------------------------------

------------------------------------------------
Training NN with lr = 5.000000e-01, reg = 1.000000e-05, hidden_dim = 200
Train accuracy: 0.495000, Val accuracy: 0.472000
Final training loss:  1.189835548400879
```



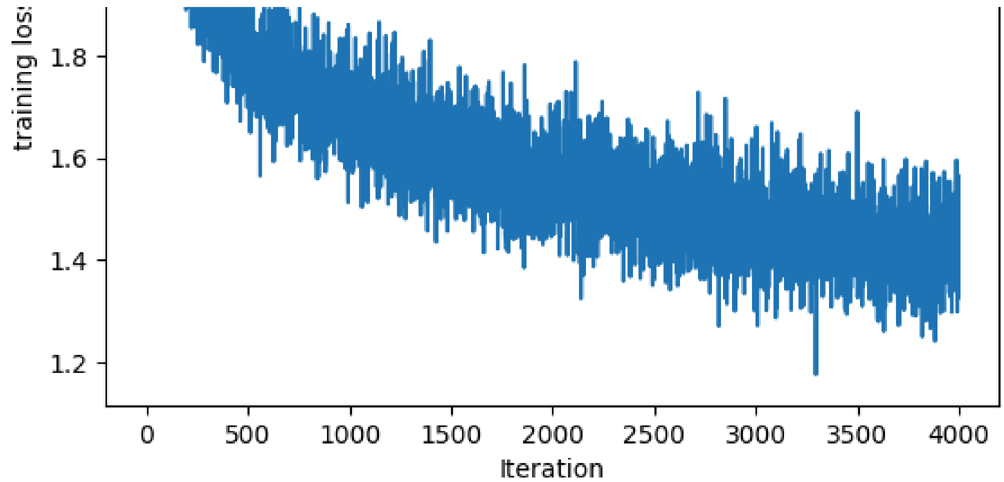


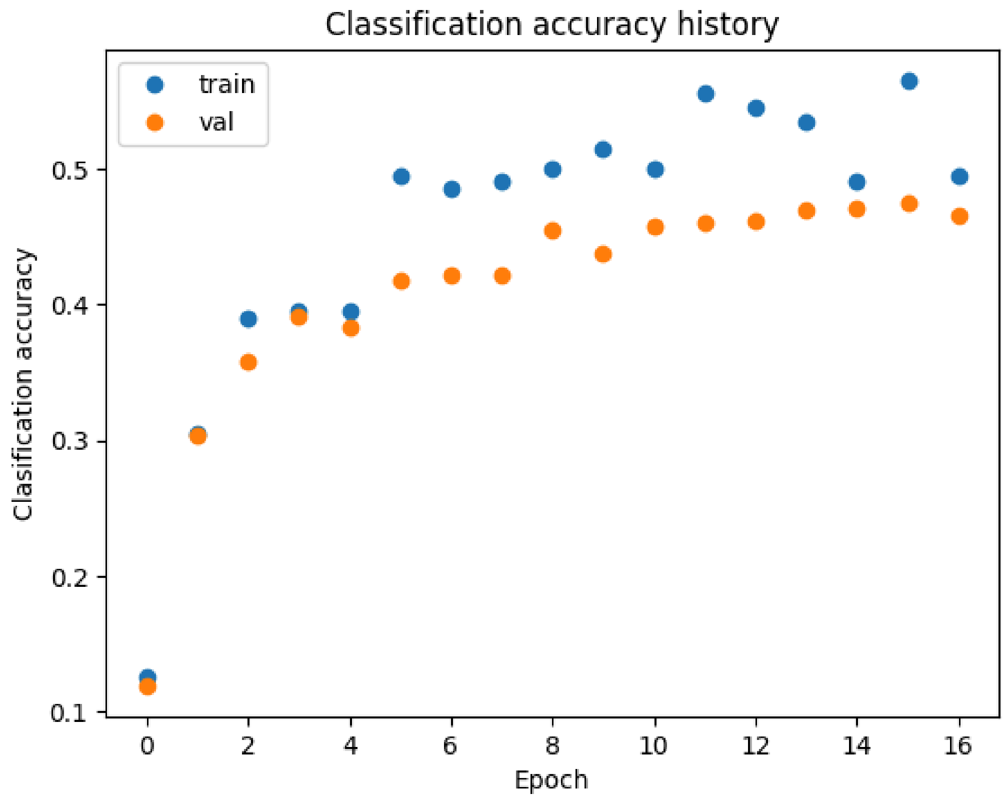
Classification accuracy history



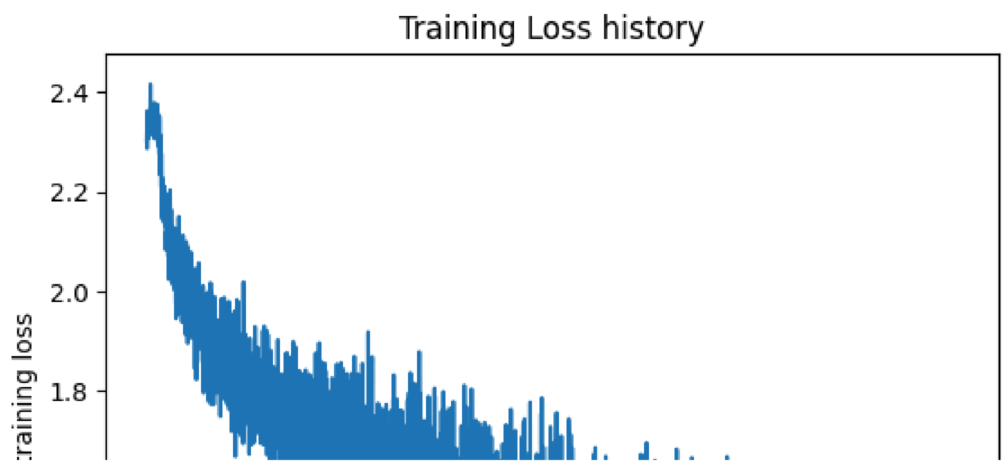

```
------------------------------------------------

------------------------------------------------
Training NN with lr = 5.000000e-01, reg = 5.000000e-05, hidden_dim = 50
Train accuracy: 0.495000, Val accuracy: 0.466000
Final training loss:  1.4699710607528687
```

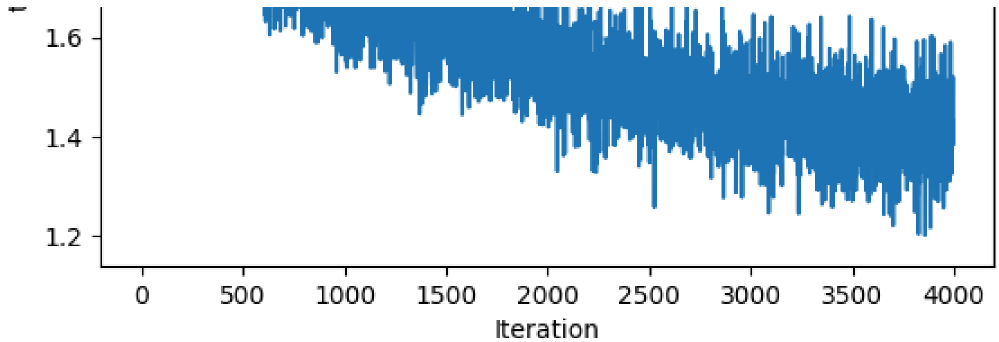
Training Loss history

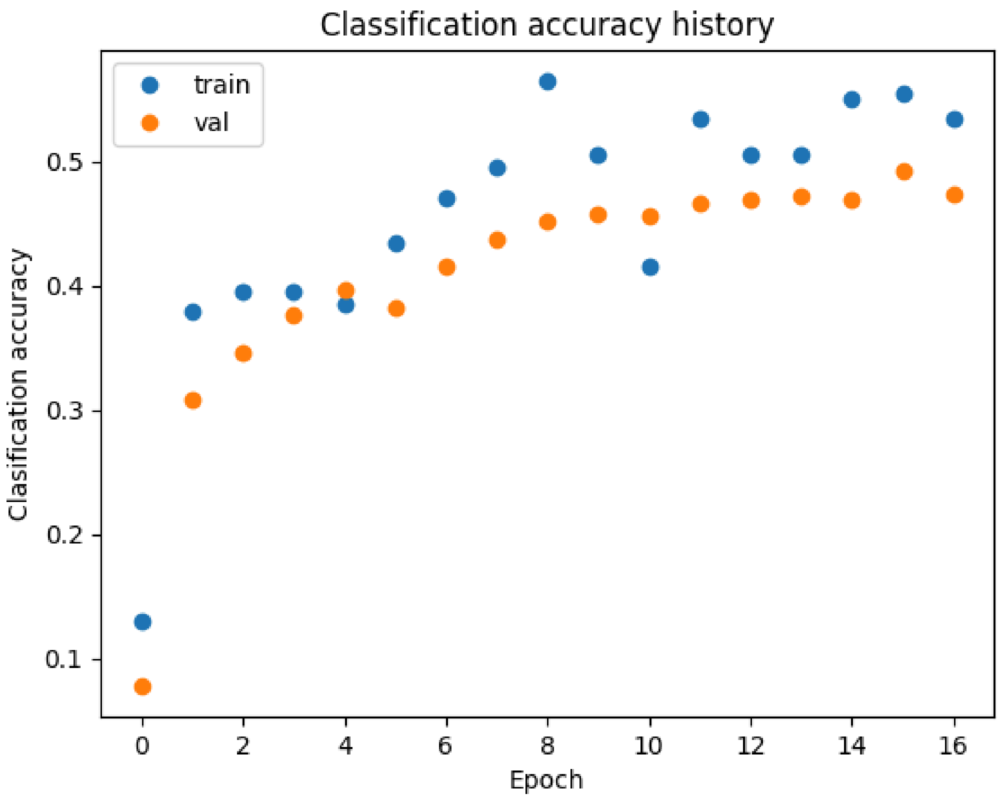




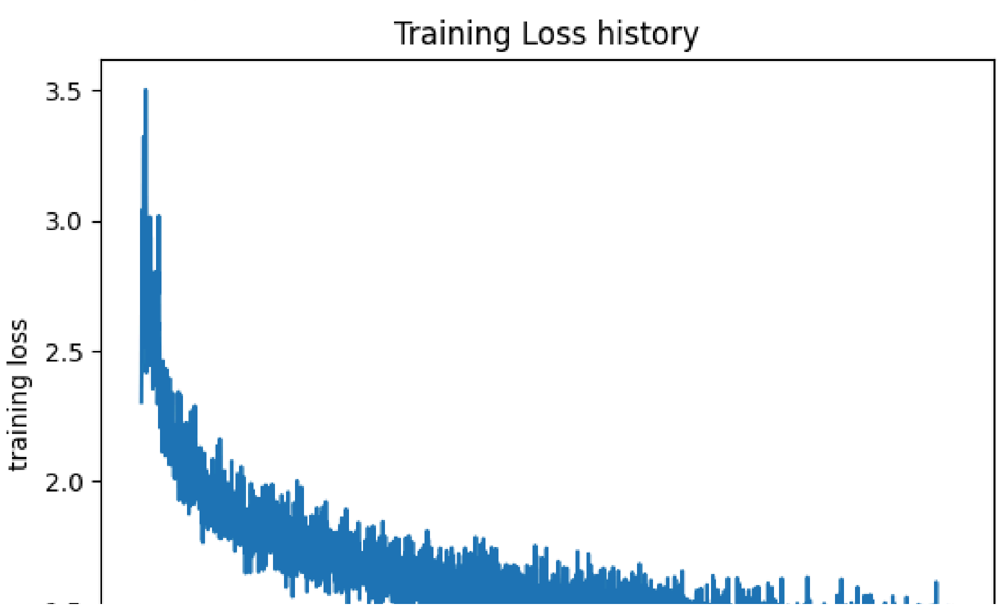

```
--------------------------------------------------

--------------------------------------------------
Training NN with lr = 5.000000e-01, reg = 5.000000e-05, hidden_dim = 100
Train accuracy: 0.535000, Val accuracy: 0.473000
Final training loss:  1.4302115440368652
```

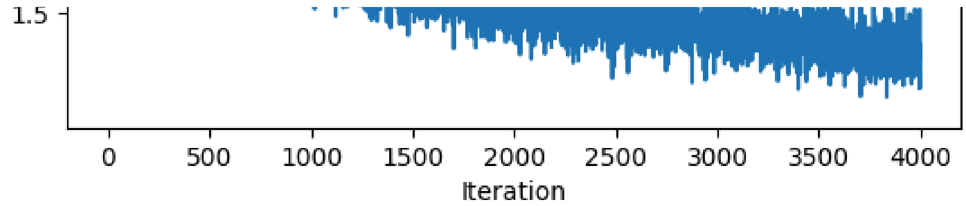


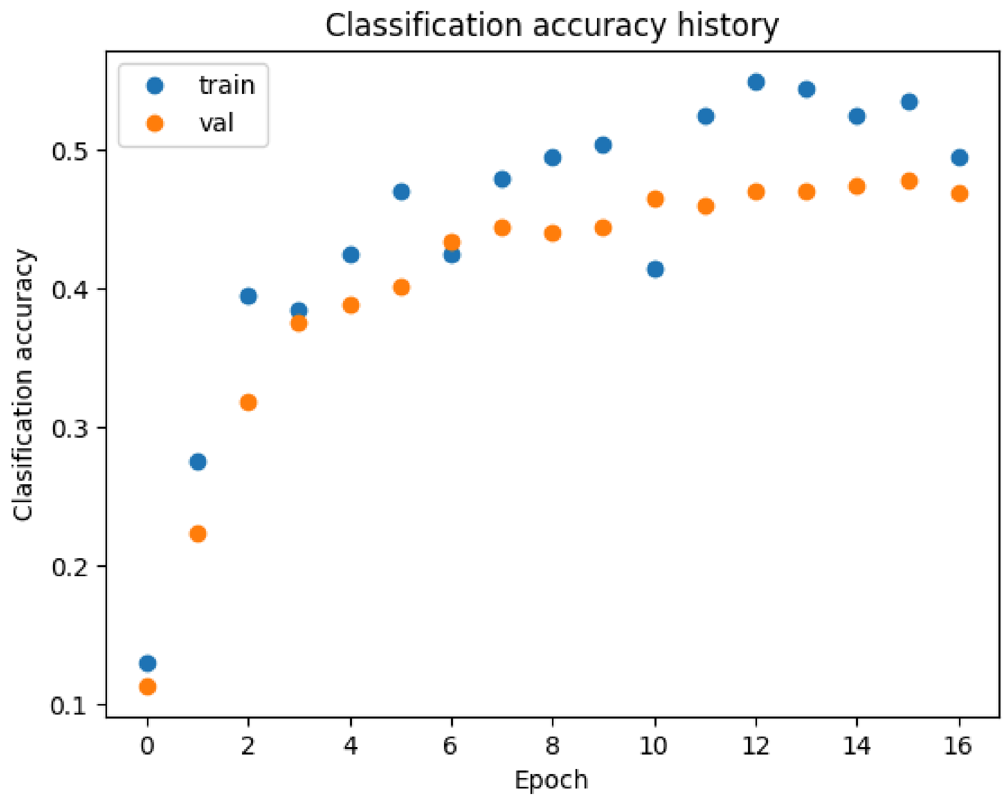




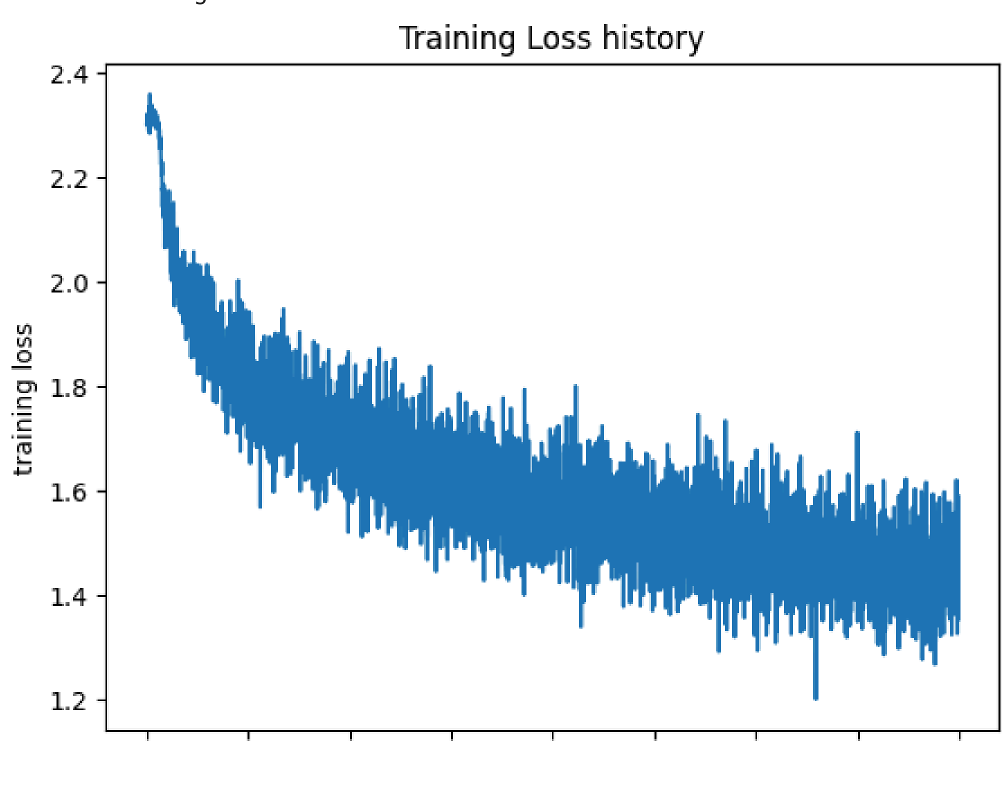

```
------------------------------------------------

------------------------------------------------
Training NN with lr = 5.000000e-01, reg = 5.000000e-05, hidden_dim = 200
Train accuracy: 0.495000, Val accuracy: 0.469000
Final training loss:  1.2164244651794434
```

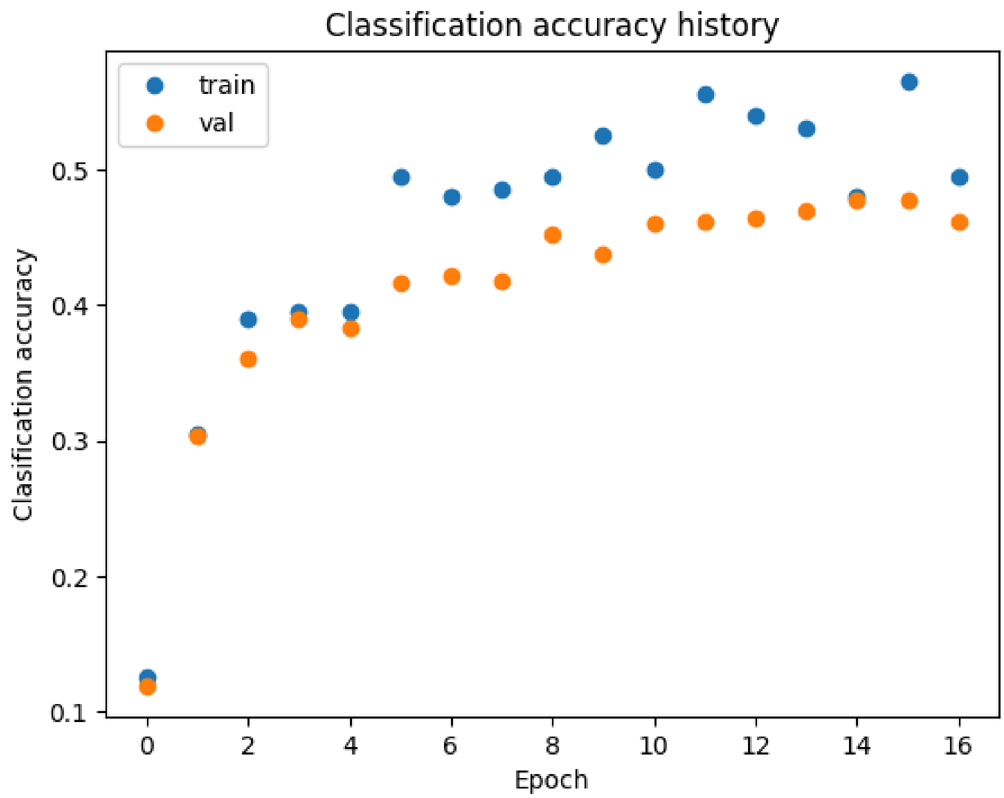




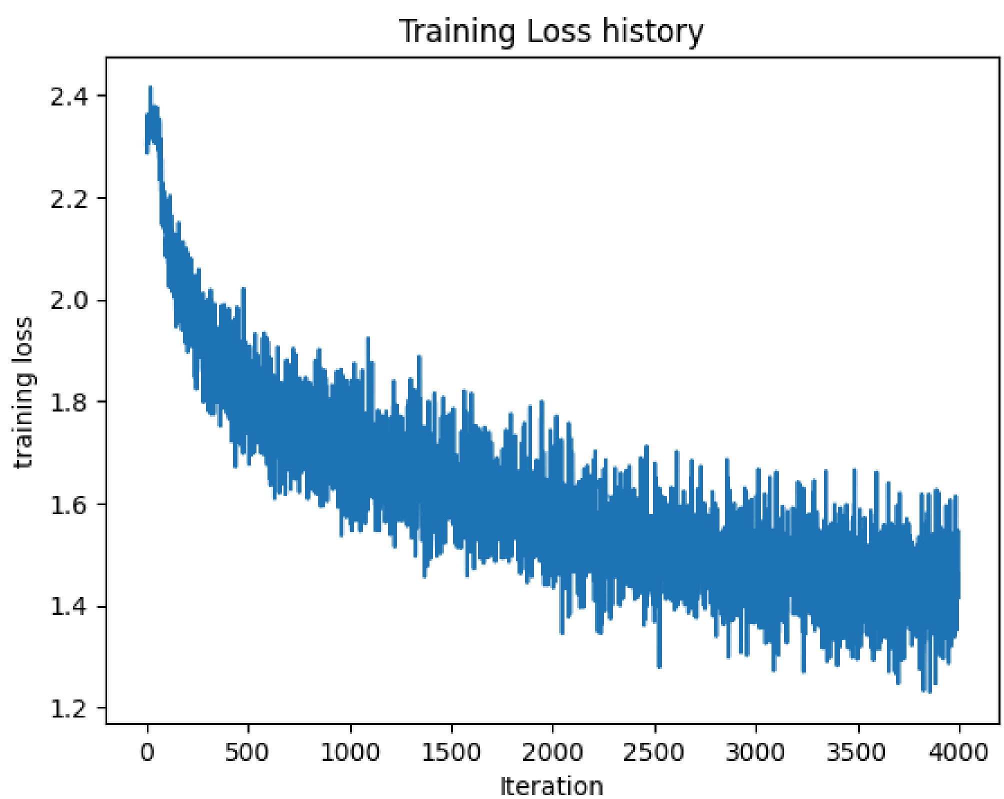


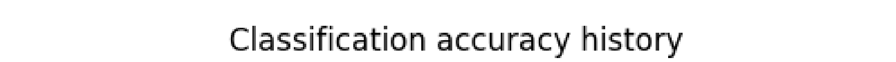

```
------------------------------------------------

------------------------------------------------
Training NN with lr = 5.000000e-01, reg = 1.000000e-04, hidden_dim = 50
Train accuracy: 0.495000, Val accuracy: 0.462000
Final training loss:  1.4946109056472778
```

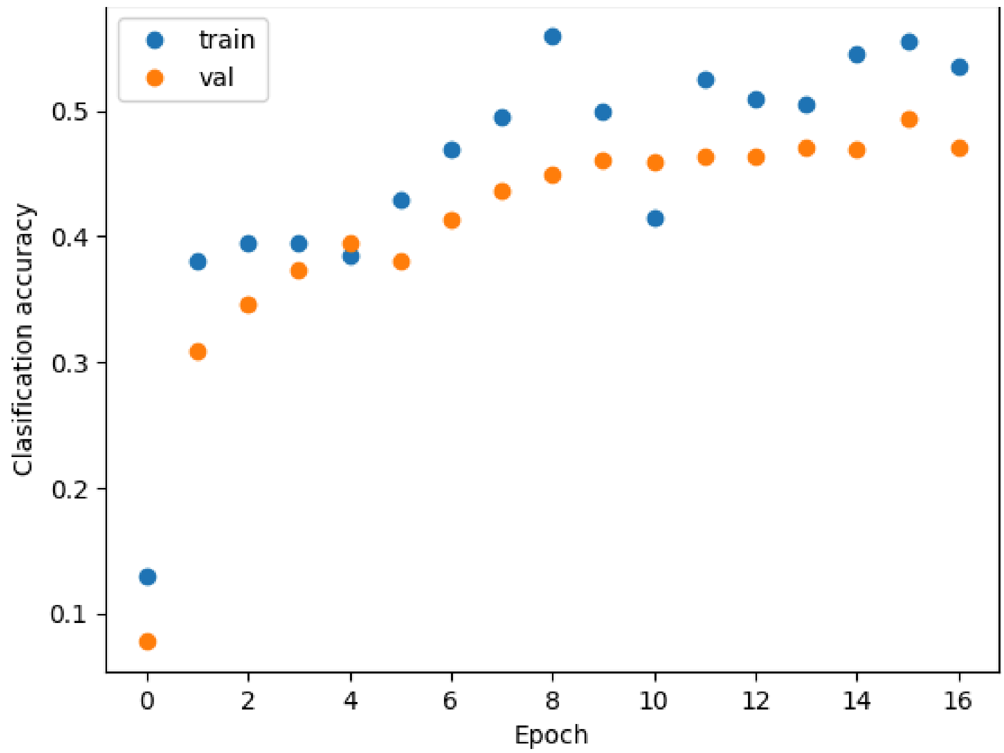





```
------------------------------------------------

------------------------------------------------
Training NN with lr = 5.000000e-01, reg = 1.000000e-04, hidden_dim = 100
Train accuracy: 0.535000, Val accuracy: 0.471000
Final training loss:  1.4596526622772217
```

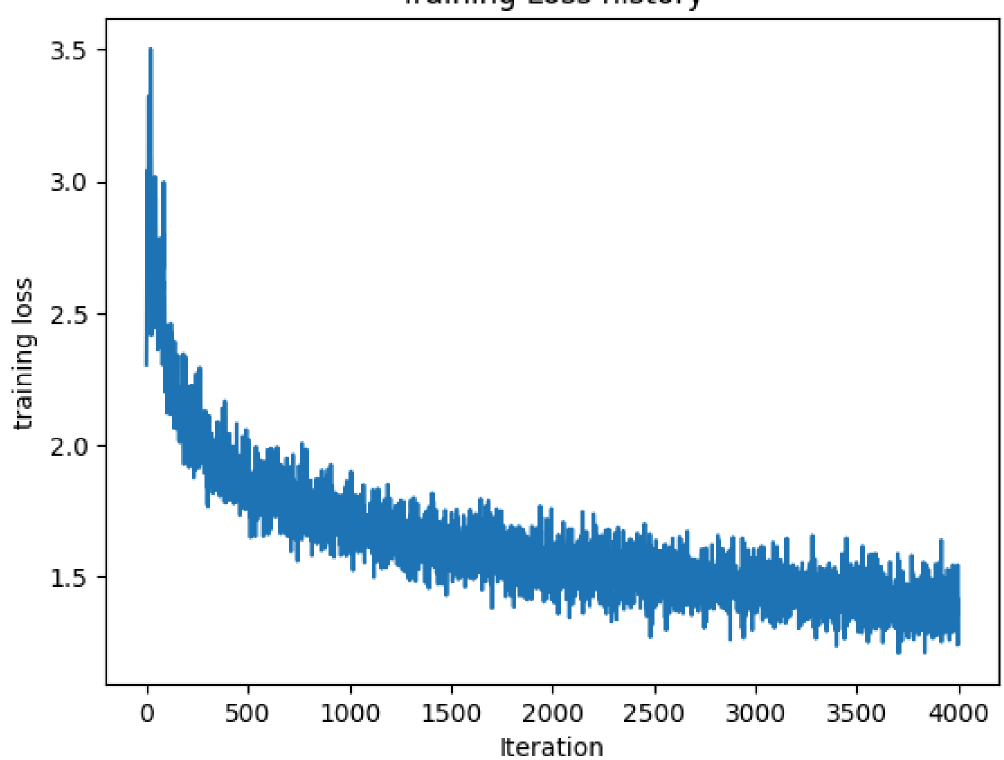




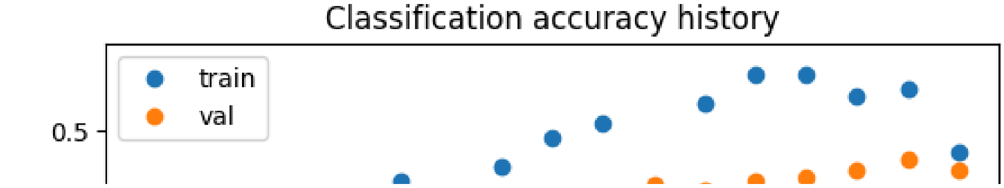


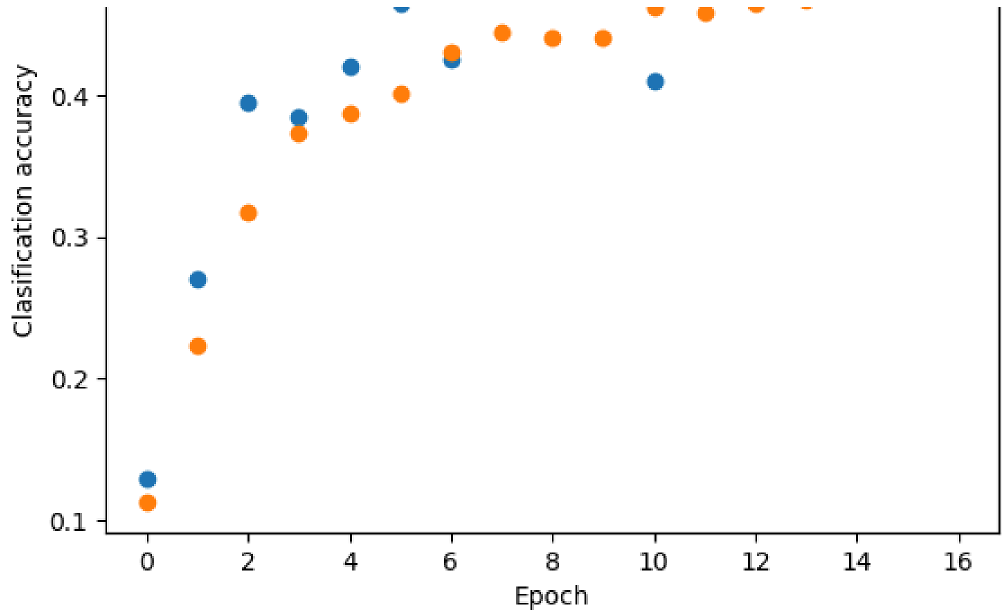
```
Training NN with lr = 5.000000e-01, reg = 1.000000e-04, hidden_dim = 200
Train accuracy: 0.485000, Val accuracy: 0.472000
Final training loss:  1.2458816766738892
```

Training Loss history

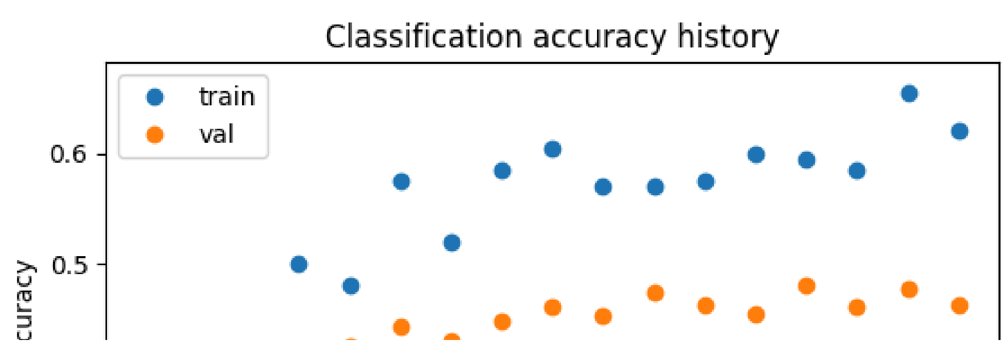


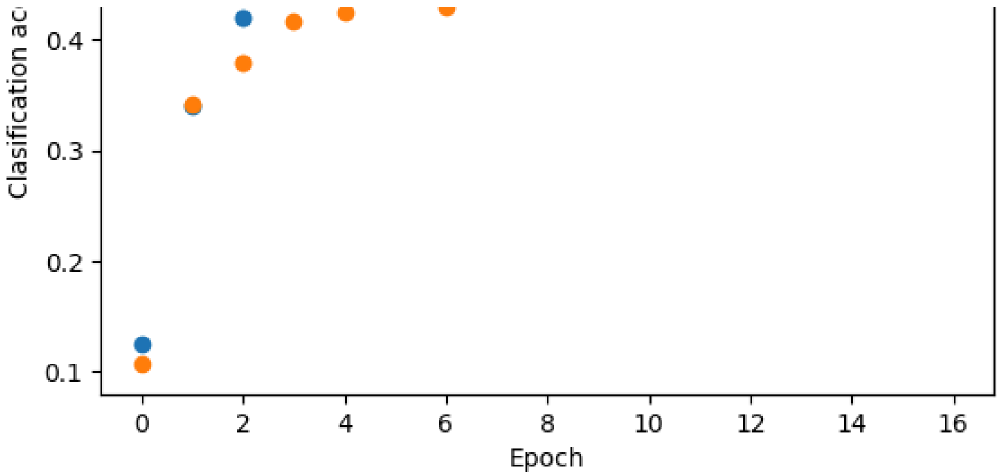
Classification accuracy history



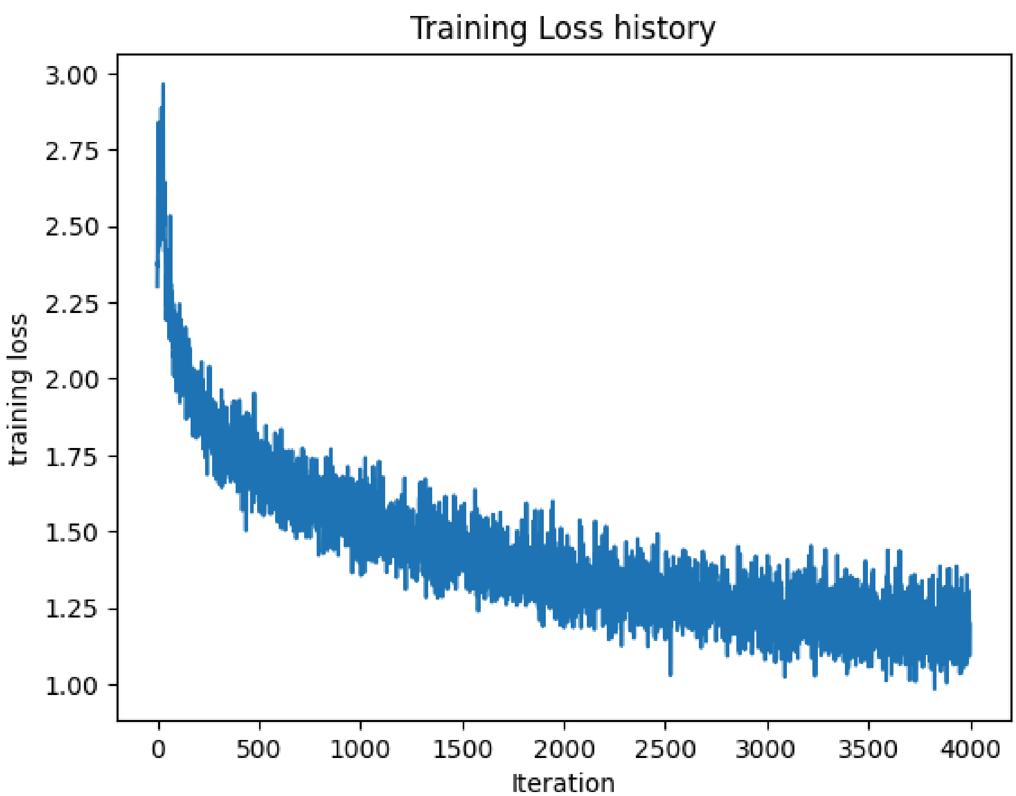


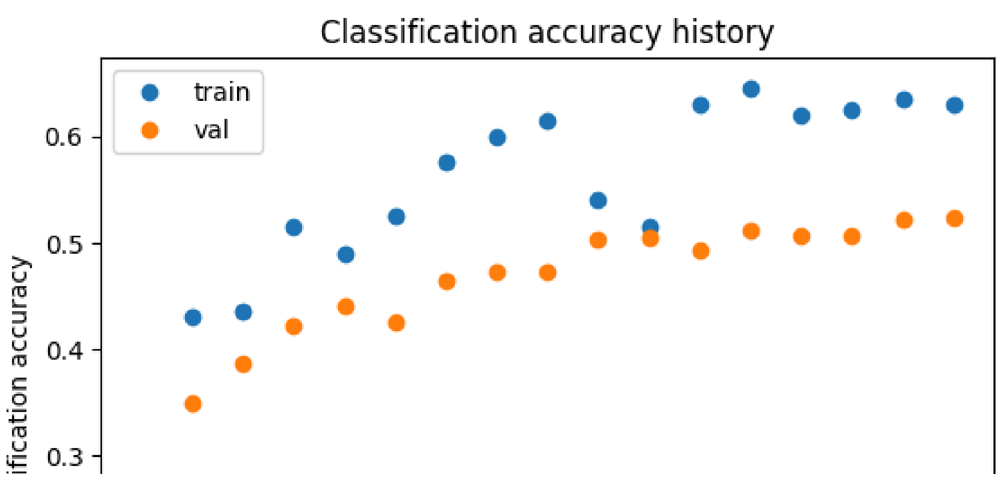

```
--------------------------------------------------

--------------------------------------------------
Training NN with lr = 9.000000e-01, reg = 5.000000e-06, hidden_dim = 50
Train accuracy: 0.620000, Val accuracy: 0.462000
Final training loss:  1.3458080291748047
```

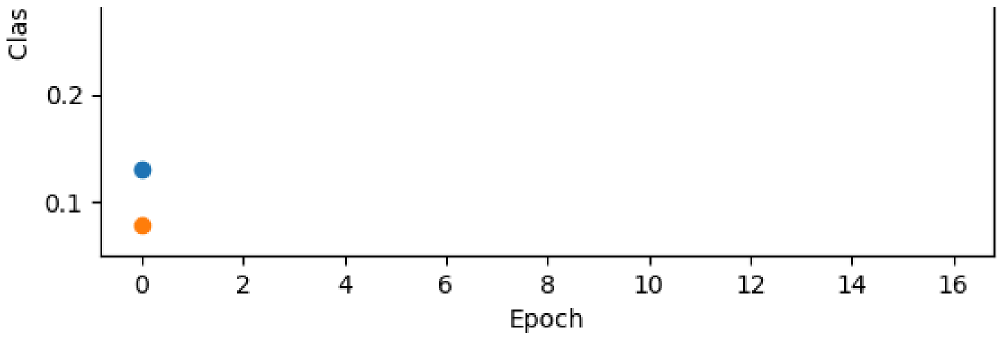




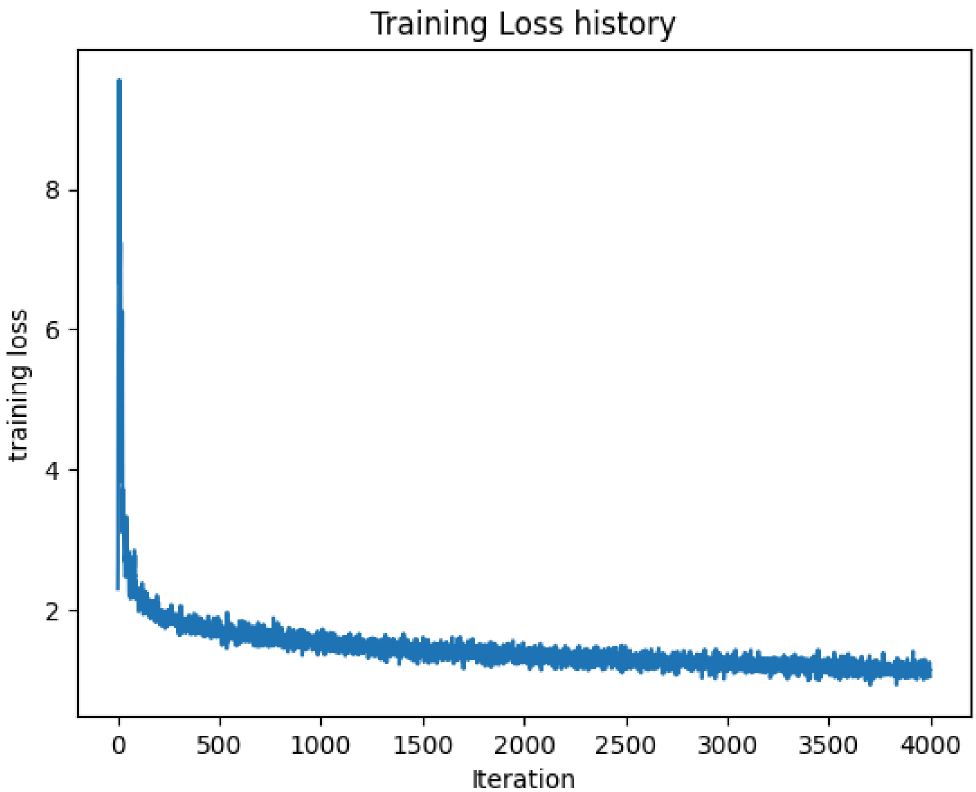

```
---------------------------------------------------

---------------------------------------------------
Training NN with lr = 9.000000e-01, reg = 5.000000e-06, hidden_dim = 100
Train accuracy: 0.630000, Val accuracy: 0.523000
Final training loss:  1.1975157260894775
```

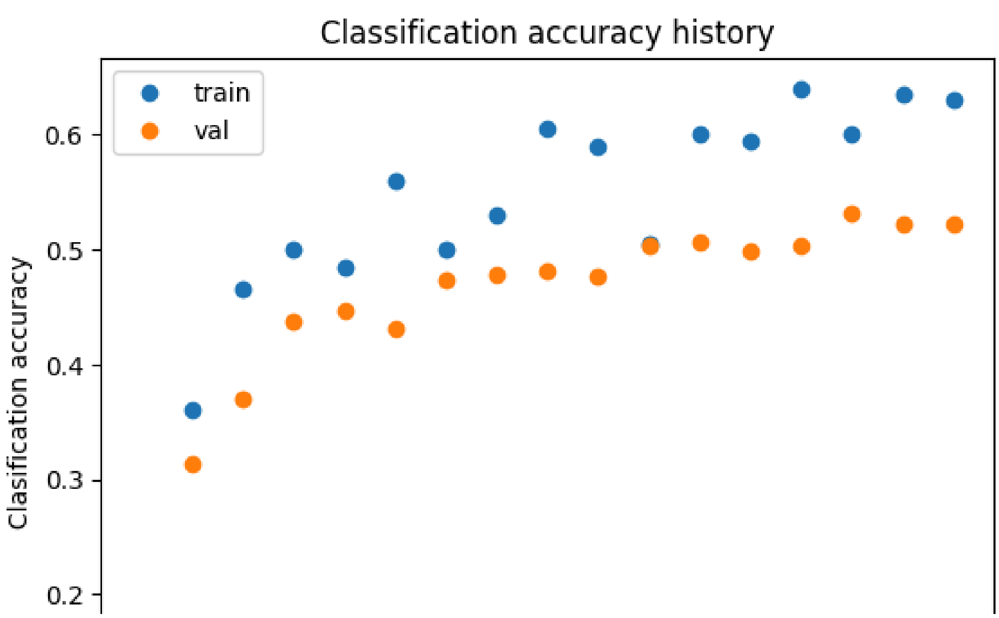


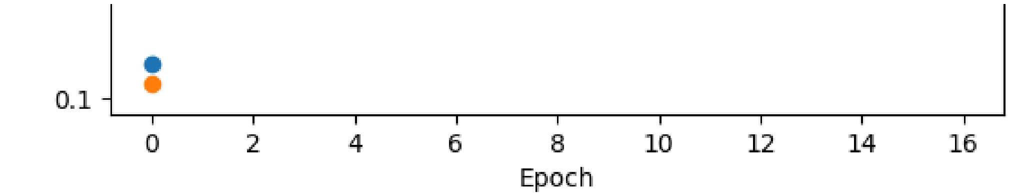




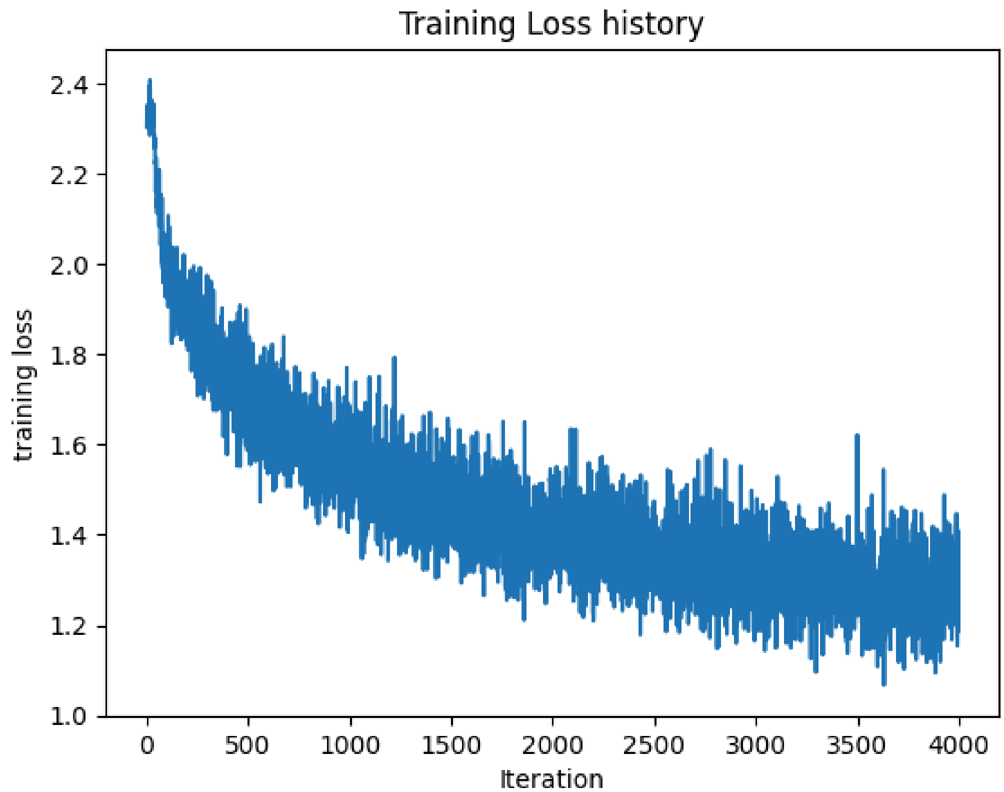
```
Training NN with lr = 9.000000e-01, reg = 5.000000e-06, hidden_dim = 200
Train accuracy: 0.630000, Val accuracy: 0.523000
Final training loss:  1.0480387210845947
```

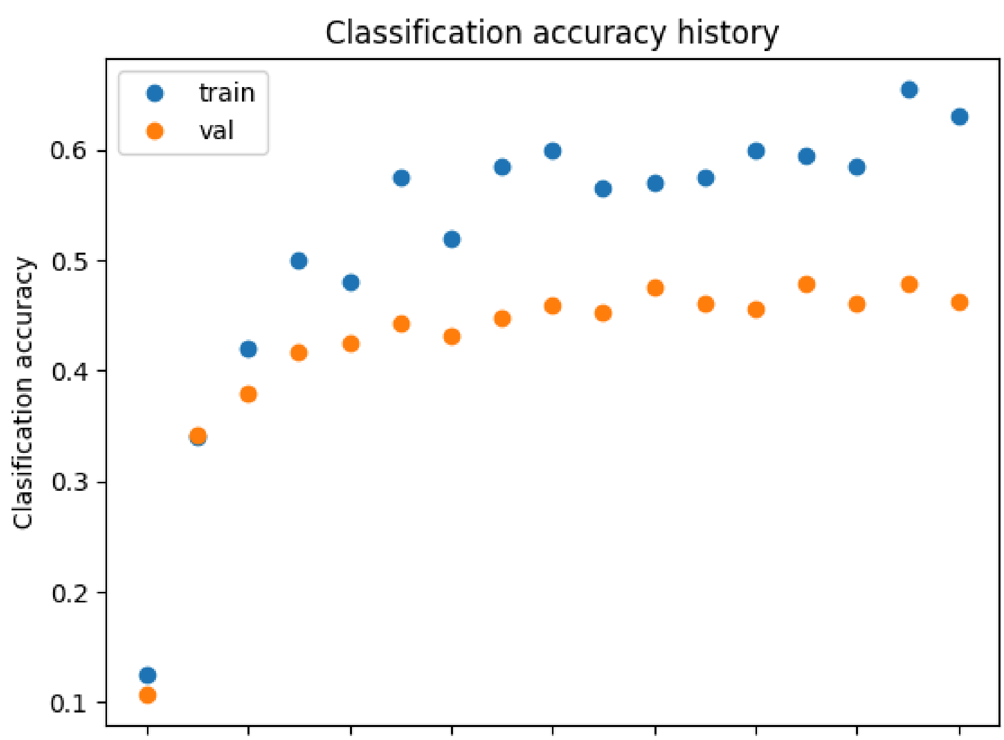




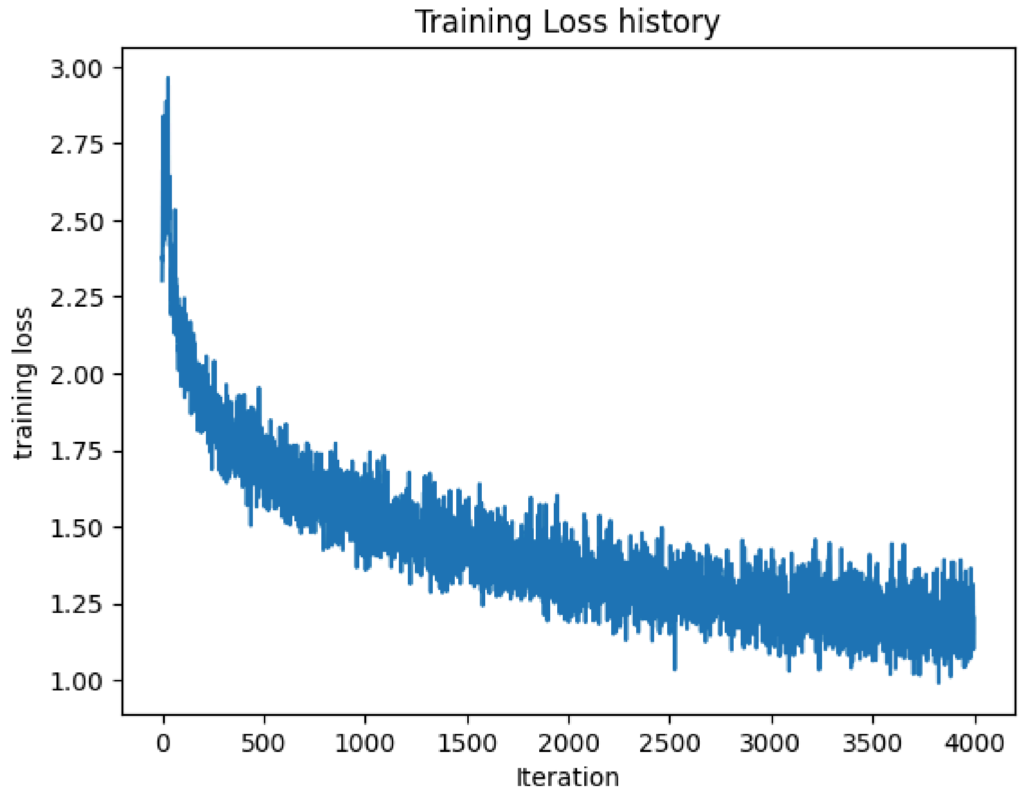


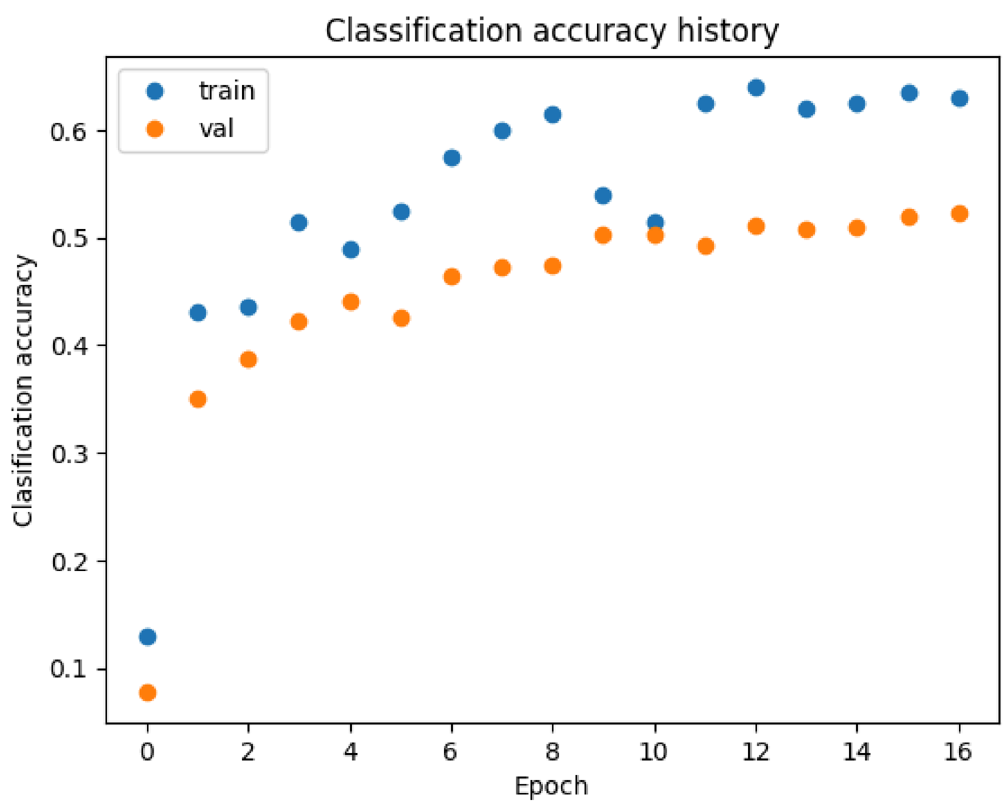
```
Training NN with lr = 9.000000e-01, reg = 1.000000e-05, hidden_dim = 50
Train accuracy: 0.630000, Val accuracy: 0.463000
Final training loss:  1.3492642641067505
```





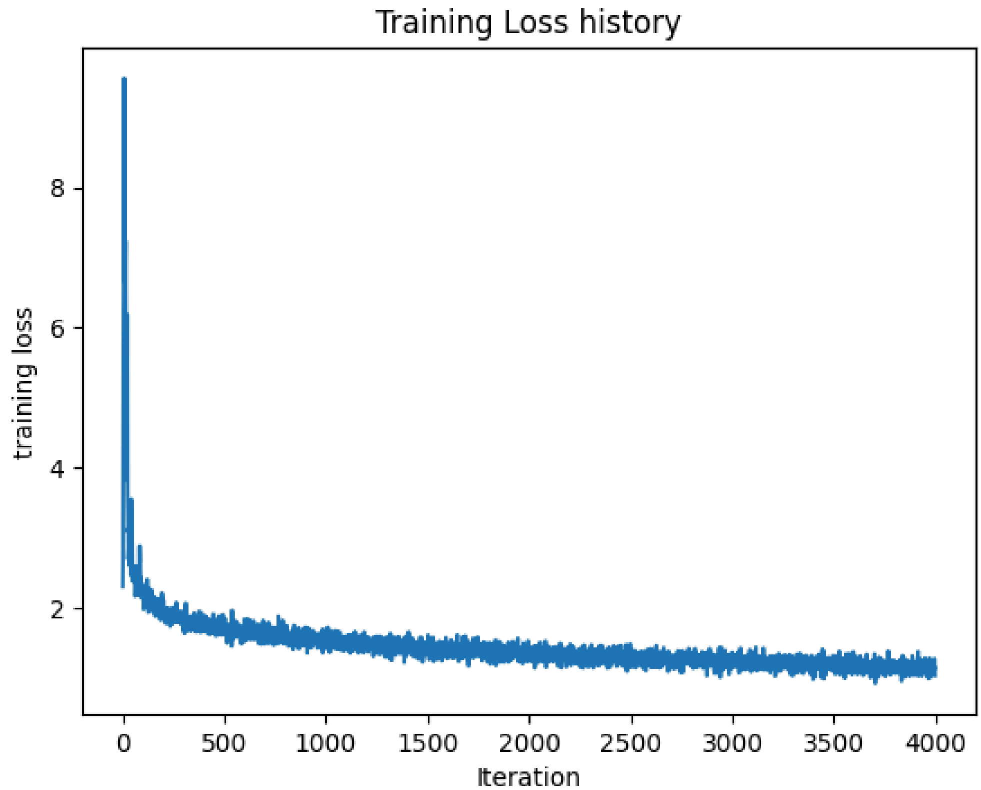

```
------------------------------------------------

------------------------------------------------
Training NN with lr = 9.000000e-01, reg = 1.000000e-05, hidden_dim = 100
Train accuracy: 0.630000, Val accuracy: 0.523000
Final training loss:  1.203563928604126
```

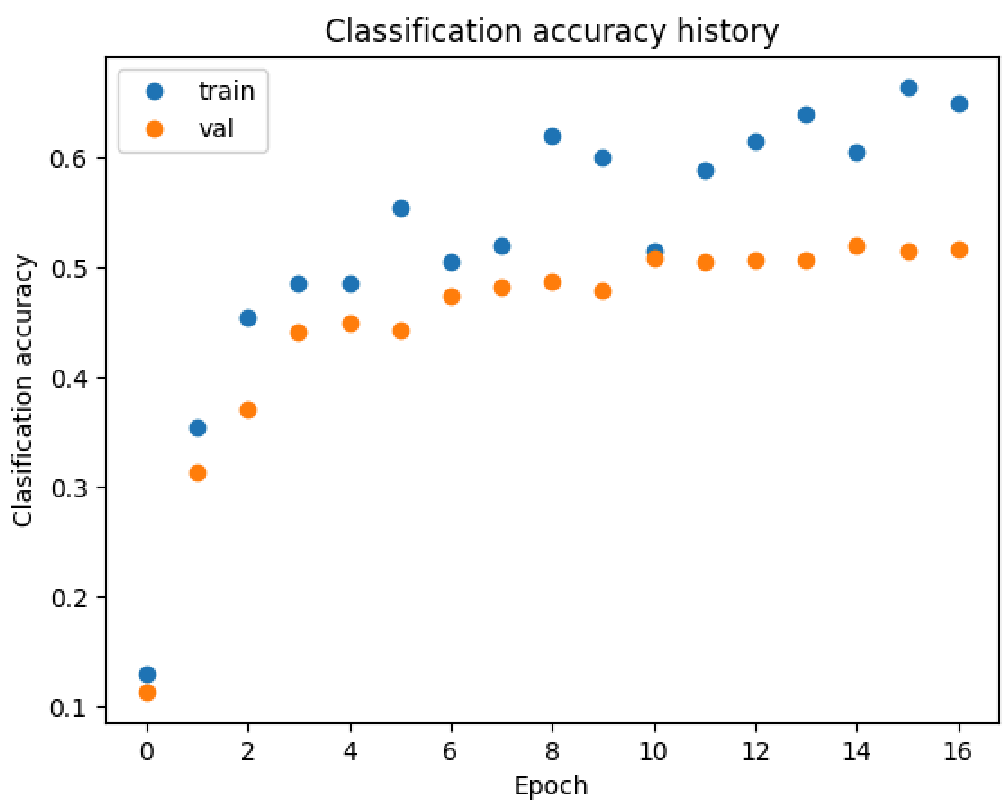




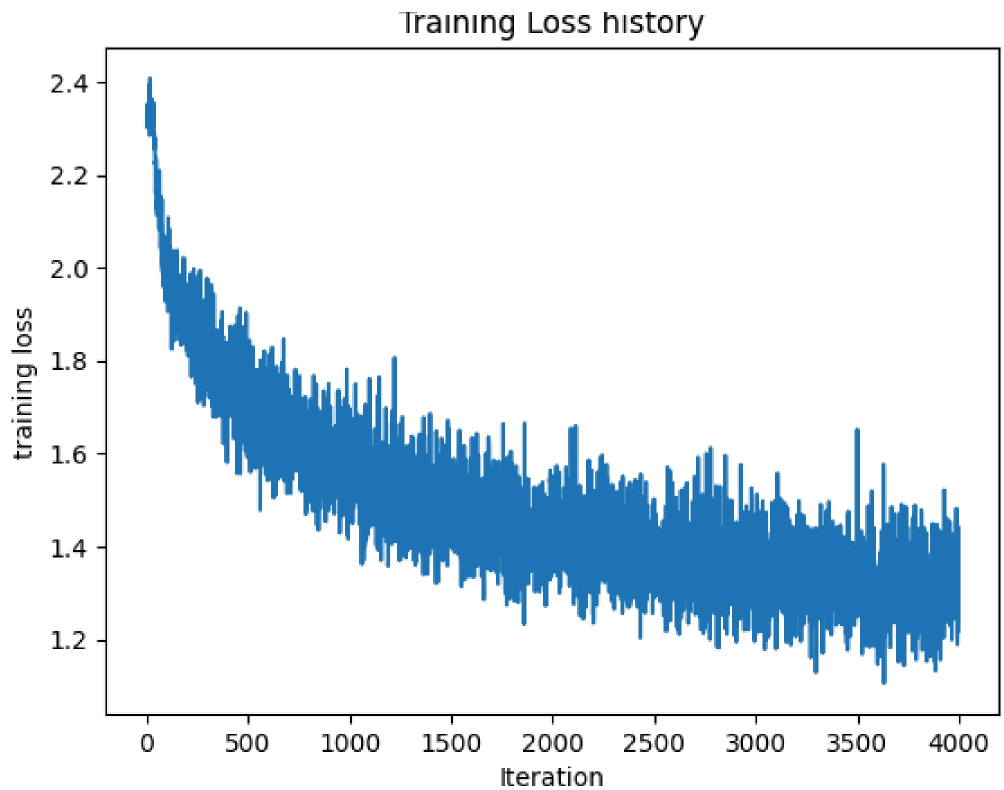

```
------------------------------------------------
```

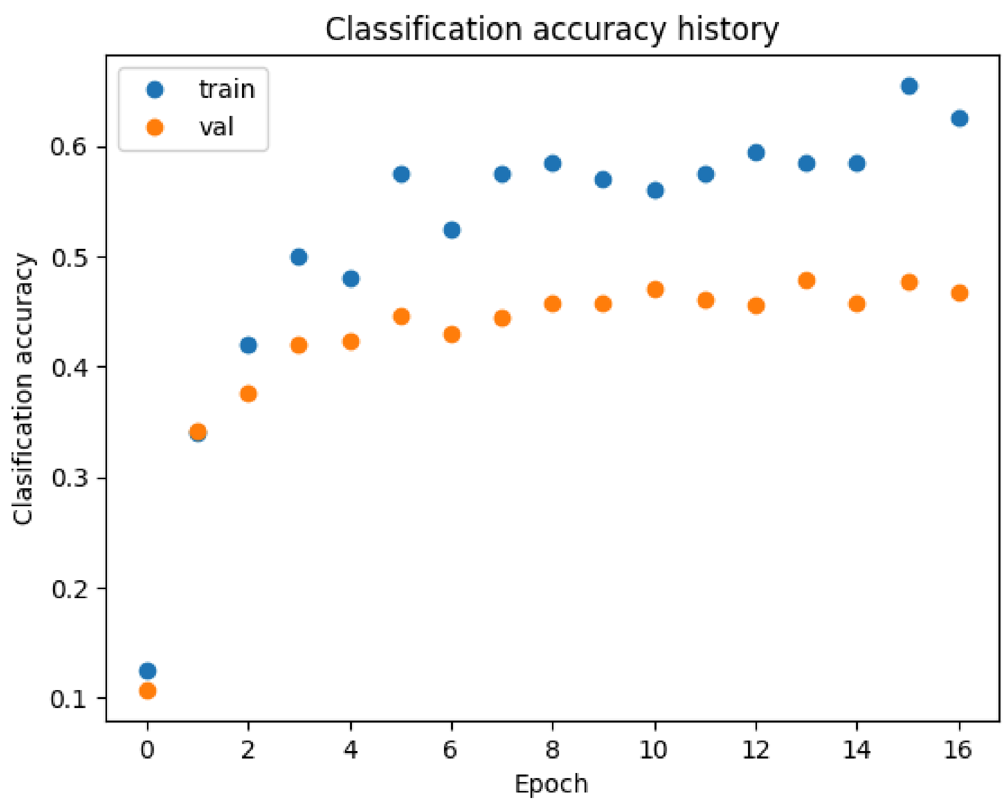
```
Training NN with lr = 9.000000e-01, reg = 1.000000e-05, hidden_dim = 200
Train accuracy: 0.650000, Val accuracy: 0.517000
Final training loss:  1.0313235521316528
```



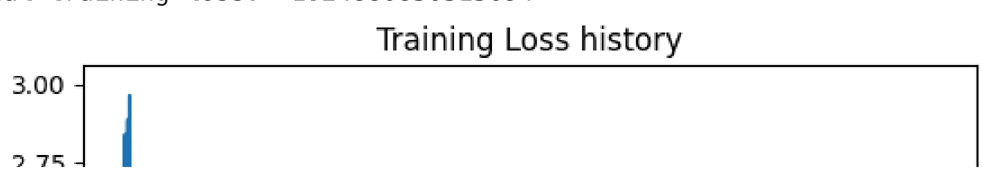


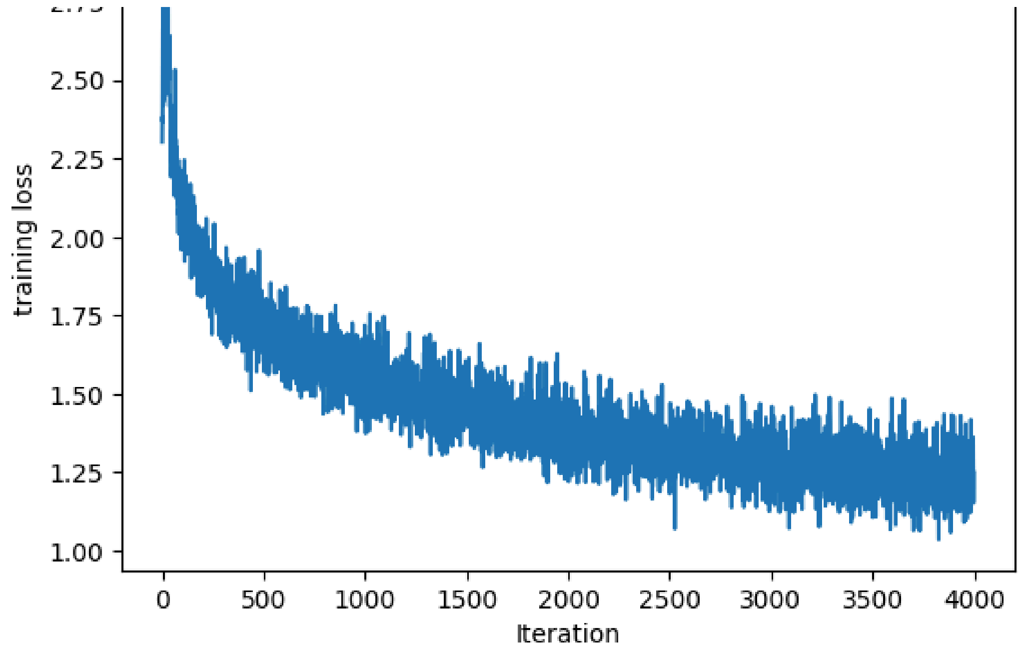
```
Training NN with lr = 9.000000e-01, reg = 5.000000e-05, hidden_dim = 50
Train accuracy: 0.625000, Val accuracy: 0.467000
Final training loss:  1.379137396812439
```

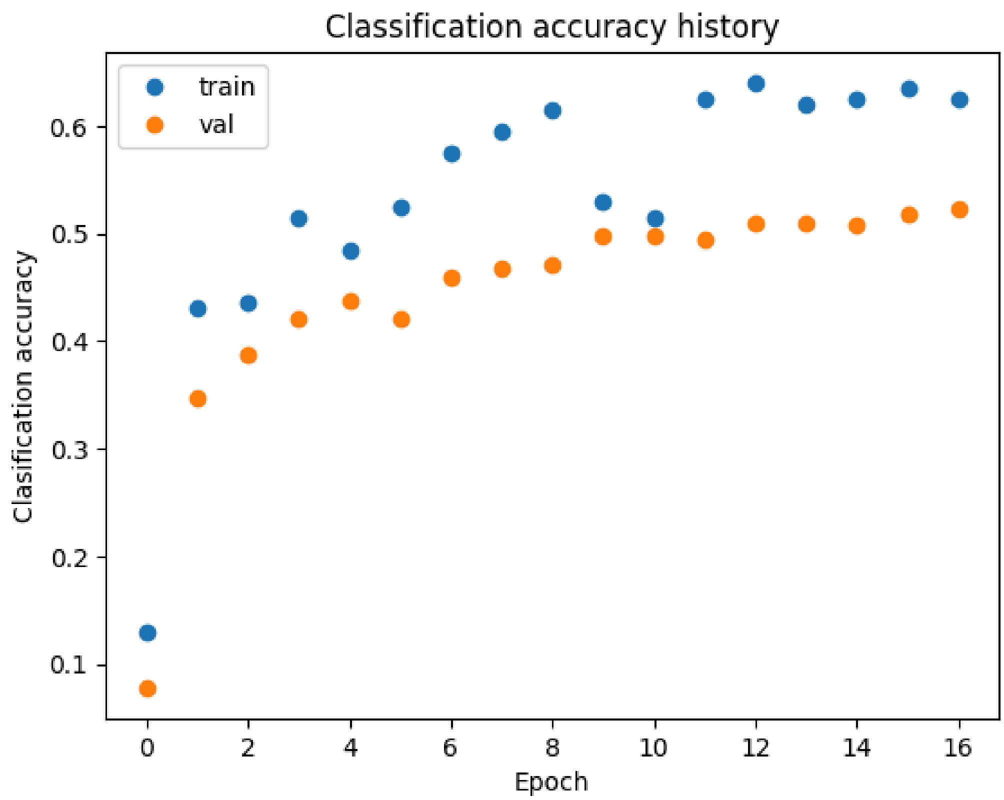




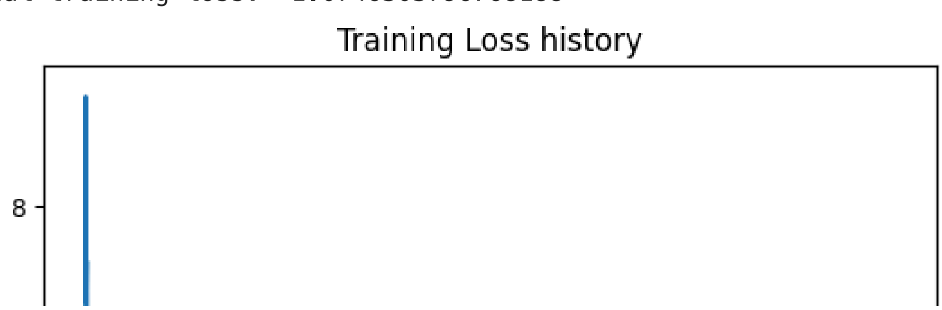

```
------------------------------------------------

------------------------------------------------
Training NN with lr = 9.000000e-01, reg = 5.000000e-05, hidden_dim = 100
Train accuracy: 0.625000, Val accuracy: 0.522000
Final training loss:  1.24880850315094
```

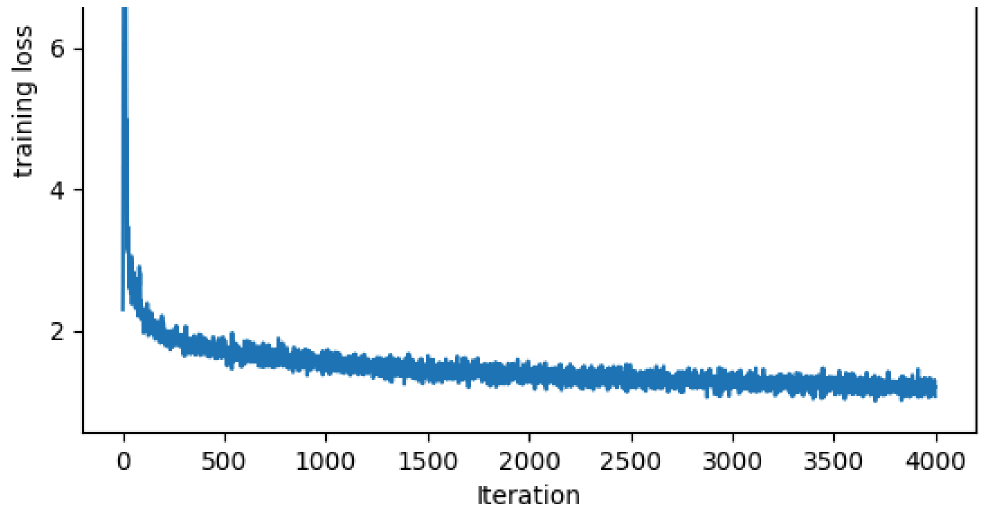


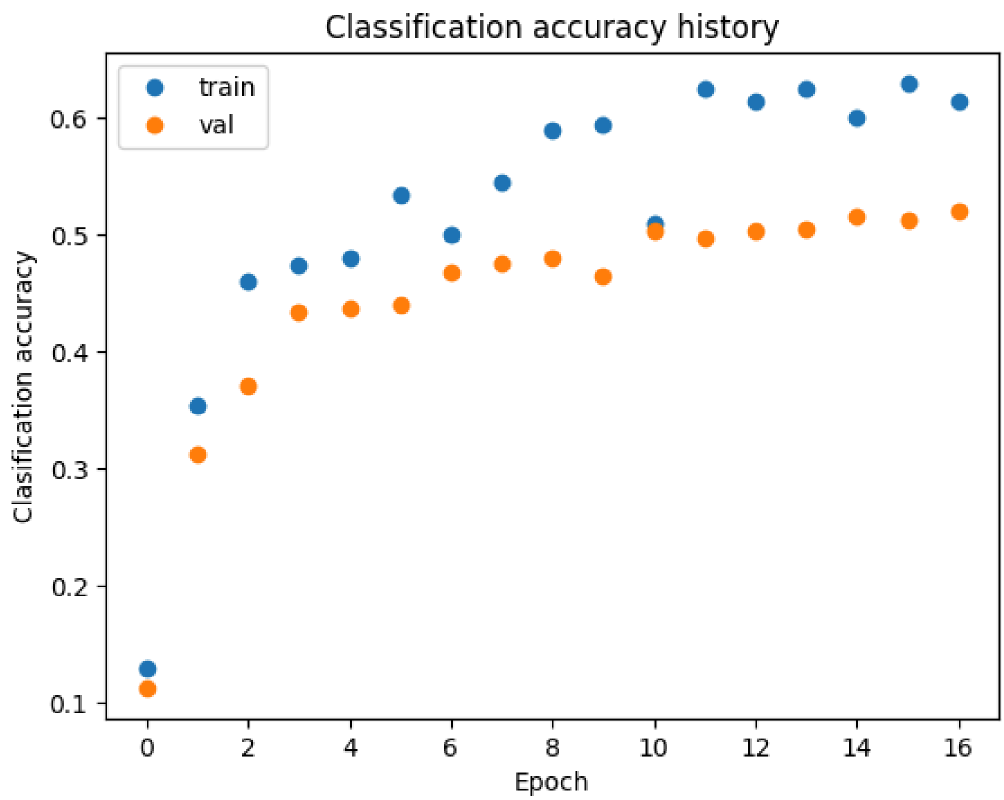




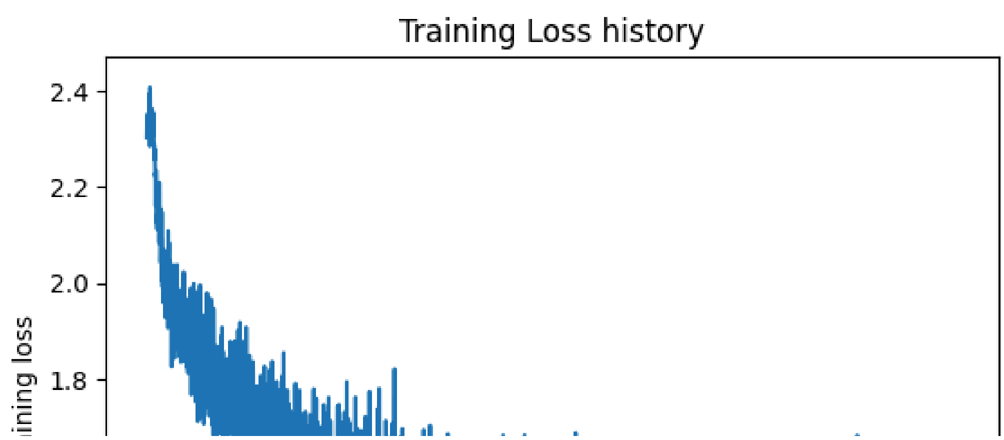

```
------------------------------------------------

------------------------------------------------
Training NN with lr = 9.000000e-01, reg = 5.000000e-05, hidden_dim = 200
Train accuracy: 0.615000, Val accuracy: 0.521000
Final training loss:  1.0746303796768188
```

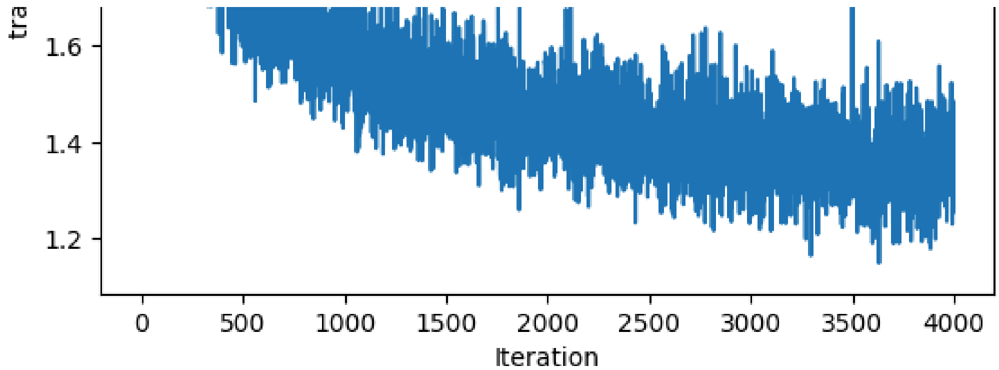


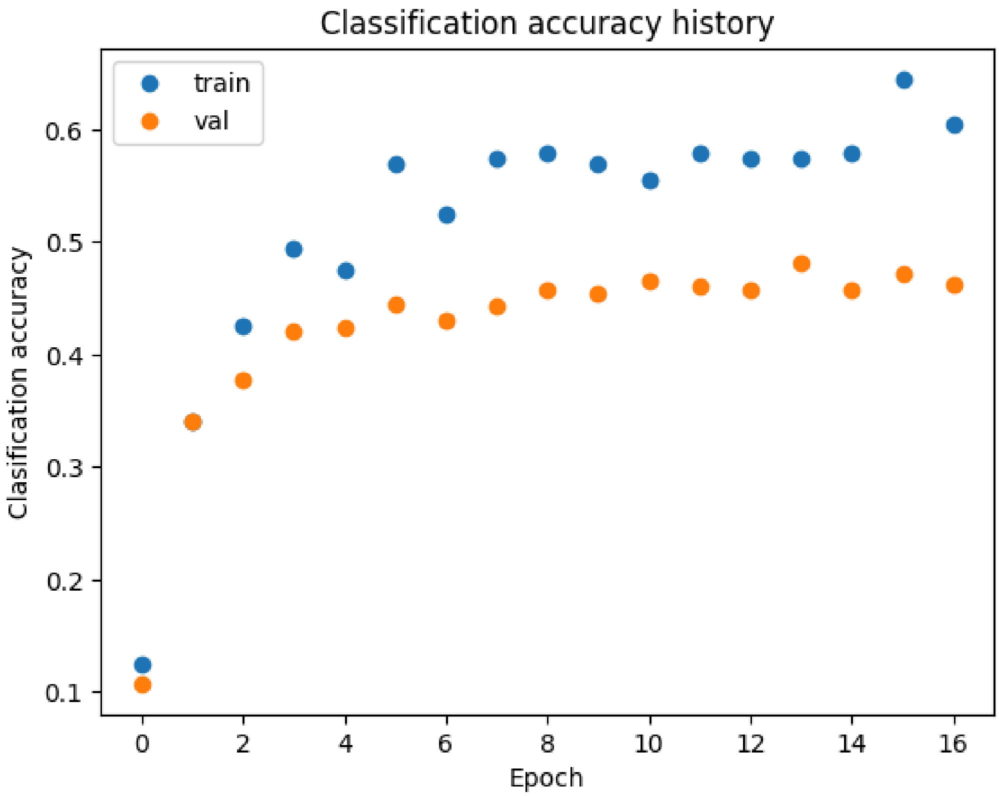




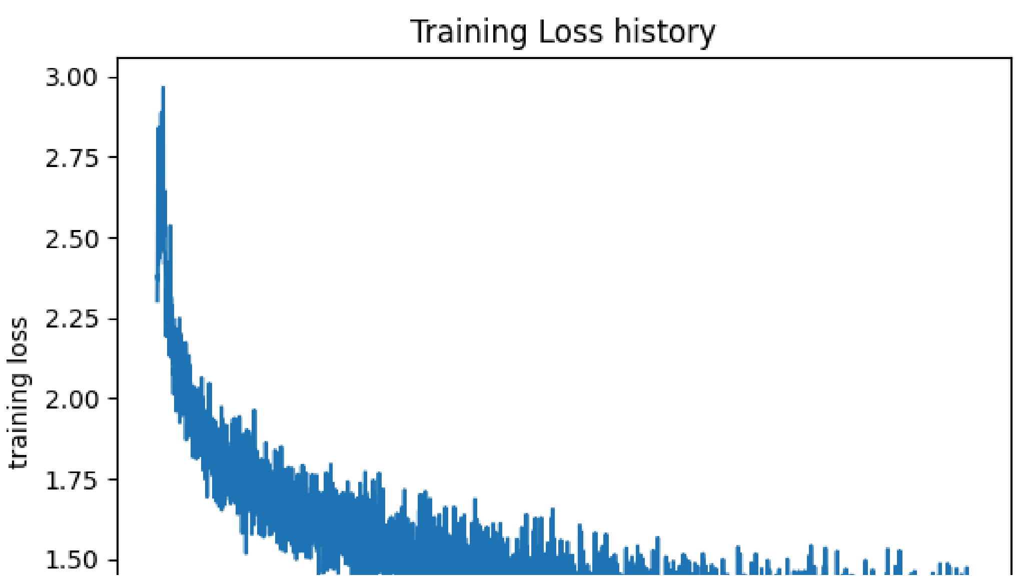

```
------------------------------------------------

------------------------------------------------
Training NN with lr = 9.000000e-01, reg = 1.000000e-04, hidden_dim = 50
Train accuracy: 0.605000, Val accuracy: 0.463000
Final training loss:  1.416151523590088
```

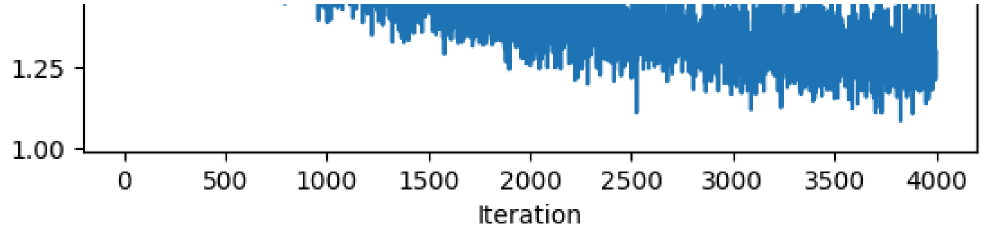


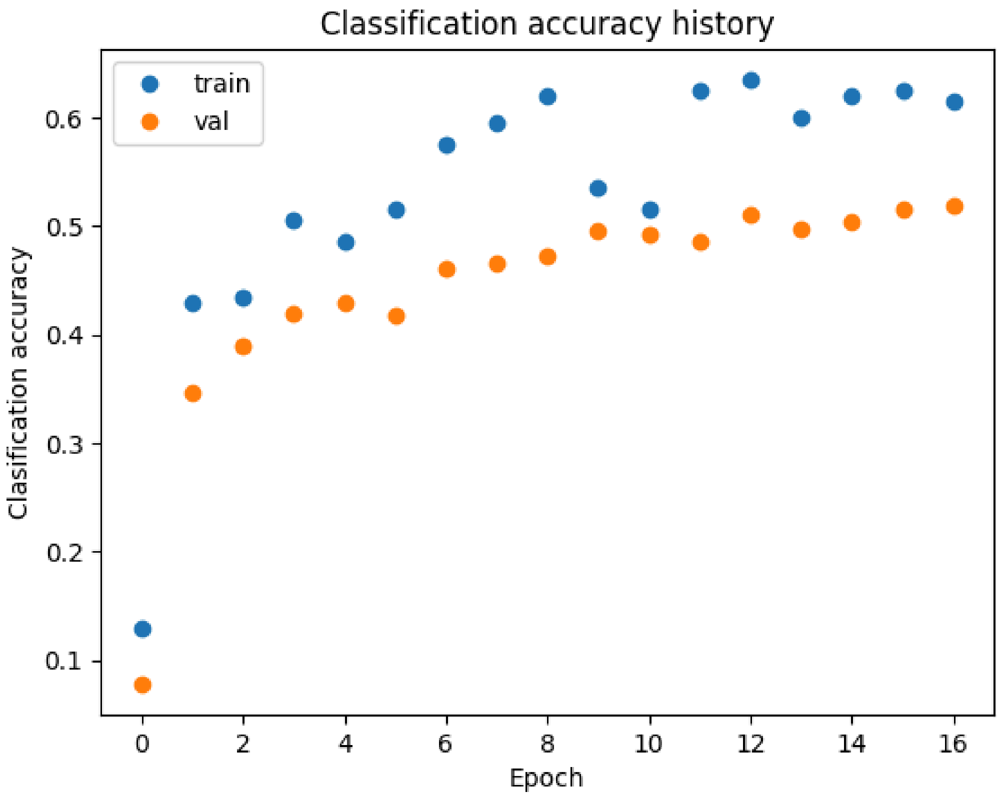




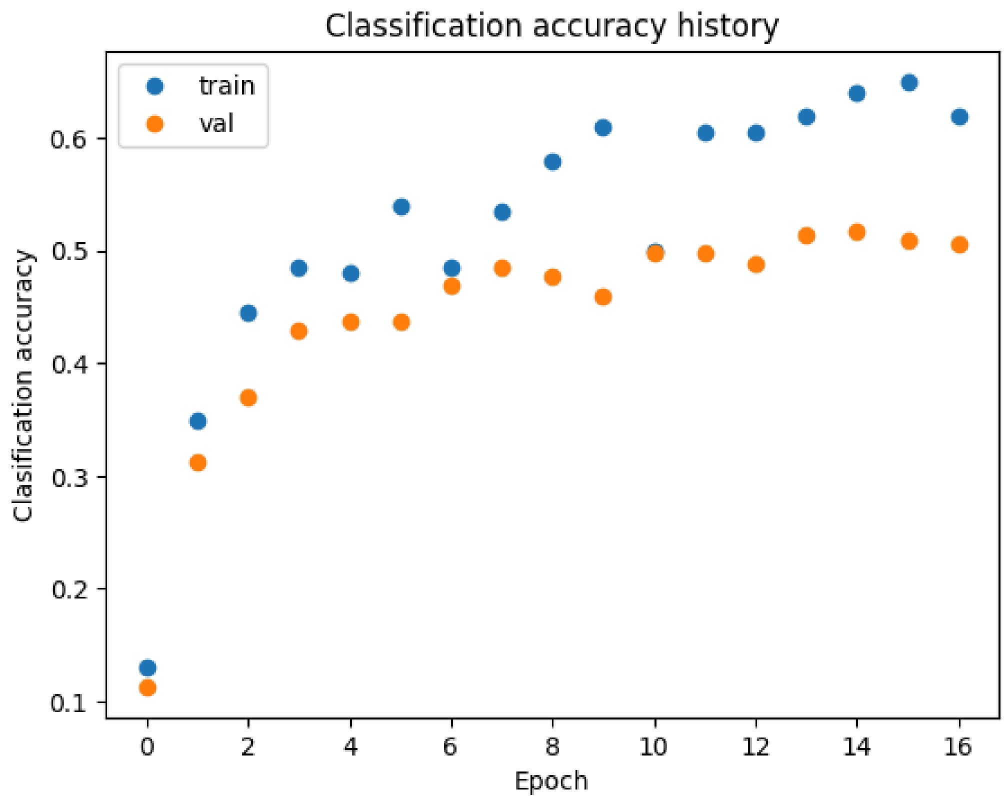

```
------------------------------------------------

------------------------------------------------
Training NN with lr = 9.000000e-01, reg = 1.000000e-04, hidden_dim = 100
Train accuracy: 0.615000, Val accuracy: 0.519000
Final training loss:  1.2986786365509033
```








```
------------------------------------------------

------------------------------------------------
Training NN with lr = 9.000000e-01, reg = 1.000000e-04, hidden_dim = 200
Train accuracy: 0.620000, Val accuracy: 0.506000
Final training loss:  1.131883978843689
```




```
------------------------------------------------

Best validation accuracy achieved: 0.523000
Final test accuracy 2-layered neural network achieved: 0.533000
```

Acknowledgement