updated_two_layer_net

July 24, 2022

1 EECS 498-007/598-005 Assignment 2-2: Two Layer Neural Network

Before we start, please put your name and UMID in following format Firstname LASTNAME, #00000000 // e.g.) Justin JOHNSON, #12345678

Your Answer:
Joseph CAMACHO, #XXXXXXXX

1.1 Implementing a Neural Network

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

We train the network with a softmax loss function and L2 regularization on the weight matrices. The network uses a ReLU nonlinearity after the first fully connected layer.

In other words, the network has the following architecture:

input - fully connected layer - ReLU - fully connected layer - softmax

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

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

1.2 Install starter code

We will continue using the utility functions that we've used for Assignment 1: `coutils` package. Run this cell to download and install it.

[ ]:
```
!pip install git+https://github.com/deepvision-class/starter-code
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Collecting git+https://github.com/deepvision-class/starter-code
  Cloning https://github.com/deepvision-class/starter-code to /tmp/pip-req-
build-fu6fa_7o
  Running command git clone -q https://github.com/deepvision-class/starter-code
/tmp/pip-req-build-fu6fa_7o
Requirement already satisfied: pydrive in /usr/local/lib/python3.7/dist-packages
(from Colab-Utils==0.1.dev0) (1.3.1)
```

```
Requirement already satisfied: oauth2client>=4.0.0 in
/usr/local/lib/python3.7/dist-packages (from pydrive->Colab-Utils==0.1.dev0)
(4.1.3)
Requirement already satisfied: PyYAML>=3.0 in /usr/local/lib/python3.7/dist-
packages (from pydrive->Colab-Utils==0.1.dev0) (3.13)
Requirement already satisfied: google-api-python-client>=1.2 in
/usr/local/lib/python3.7/dist-packages (from pydrive->Colab-Utils==0.1.dev0)
(1.12.11)
Requirement already satisfied: six<2dev,>=1.13.0 in
/usr/local/lib/python3.7/dist-packages (from google-api-python-
client>=1.2->pydrive->Colab-Utils==0.1.dev0) (1.15.0)
Requirement already satisfied: uritemplate<4dev,>=3.0.0 in
/usr/local/lib/python3.7/dist-packages (from google-api-python-
client>=1.2->pydrive->Colab-Utils==0.1.dev0) (3.0.1)
Requirement already satisfied: google-api-core<3dev,>=1.21.0 in
/usr/local/lib/python3.7/dist-packages (from google-api-python-
client>=1.2->pydrive->Colab-Utils==0.1.dev0) (1.31.6)
Requirement already satisfied: httplib2<1dev,>=0.15.0 in
/usr/local/lib/python3.7/dist-packages (from google-api-python-
client>=1.2->pydrive->Colab-Utils==0.1.dev0) (0.17.4)
Requirement already satisfied: google-auth<3dev,>=1.16.0 in
/usr/local/lib/python3.7/dist-packages (from google-api-python-
client>=1.2->pydrive->Colab-Utils==0.1.dev0) (1.35.0)
Requirement already satisfied: google-auth-httplib2>=0.0.3 in
/usr/local/lib/python3.7/dist-packages (from google-api-python-
client>=1.2->pydrive->Colab-Utils==0.1.dev0) (0.0.4)
Requirement already satisfied: packaging>=14.3 in /usr/local/lib/python3.7/dist-
packages (from google-api-core<3dev,>=1.21.0->google-api-python-
client>=1.2->pydrive->Colab-Utils==0.1.dev0) (21.3)
Requirement already satisfied: requests<3.0.0dev,>=2.18.0 in
/usr/local/lib/python3.7/dist-packages (from google-api-
core<3dev,>=1.21.0->google-api-python-client>=1.2->pydrive->Colab-
Utils==0.1.dev0) (2.23.0)
Requirement already satisfied: setuptools>=40.3.0 in
/usr/local/lib/python3.7/dist-packages (from google-api-
core<3dev,>=1.21.0->google-api-python-client>=1.2->pydrive->Colab-
Utils==0.1.dev0) (57.4.0)
Requirement already satisfied: protobuf<4.0.0dev,>=3.12.0 in
/usr/local/lib/python3.7/dist-packages (from google-api-
core<3dev,>=1.21.0->google-api-python-client>=1.2->pydrive->Colab-
Utils==0.1.dev0) (3.17.3)
Requirement already satisfied: googleapis-common-protos<2.0dev,>=1.6.0 in
/usr/local/lib/python3.7/dist-packages (from google-api-
core<3dev,>=1.21.0->google-api-python-client>=1.2->pydrive->Colab-
Utils==0.1.dev0) (1.56.4)
Requirement already satisfied: pytz in /usr/local/lib/python3.7/dist-packages
(from google-api-core<3dev,>=1.21.0->google-api-python-
client>=1.2->pydrive->Colab-Utils==0.1.dev0) (2022.1)
```

```
Requirement already satisfied: cachetools<5.0,>=2.0.0 in
/usr/local/lib/python3.7/dist-packages (from google-auth<3dev,>=1.16.0->google-
api-python-client>=1.2->pydrive->Colab-Utils==0.1.dev0) (4.2.4)
Requirement already satisfied: pyasn1-modules>=0.2.1 in
/usr/local/lib/python3.7/dist-packages (from google-auth<3dev,>=1.16.0->google-
api-python-client>=1.2->pydrive->Colab-Utils==0.1.dev0) (0.2.8)
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.7/dist-
packages (from google-auth<3dev,>=1.16.0->google-api-python-
client>=1.2->pydrive->Colab-Utils==0.1.dev0) (4.8)
Requirement already satisfied: pyasn1>=0.1.7 in /usr/local/lib/python3.7/dist-
packages (from oauth2client>=4.0.0->pydrive->Colab-Utils==0.1.dev0) (0.4.8)
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in
/usr/local/lib/python3.7/dist-packages (from packaging>=14.3->google-api-
core<3dev,>=1.21.0->google-api-python-client>=1.2->pydrive->Colab-
Utils==0.1.dev0) (3.0.9)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.7/dist-packages (from requests<3.0.0dev,>=2.18.0->google-
api-core<3dev,>=1.21.0->google-api-python-client>=1.2->pydrive->Colab-
Utils==0.1.dev0) (2022.6.15)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-
packages (from requests<3.0.0dev,>=2.18.0->google-api-
core<3dev,>=1.21.0->google-api-python-client>=1.2->pydrive->Colab-
Utils==0.1.dev0) (2.10)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in
/usr/local/lib/python3.7/dist-packages (from requests<3.0.0dev,>=2.18.0->google-
api-core<3dev,>=1.21.0->google-api-python-client>=1.2->pydrive->Colab-
Utils==0.1.dev0) (1.24.3)
Requirement already satisfied: chardet<4,>=3.0.2 in
/usr/local/lib/python3.7/dist-packages (from requests<3.0.0dev,>=2.18.0->google-
api-core<3dev,>=1.21.0->google-api-python-client>=1.2->pydrive->Colab-
Utils==0.1.dev0) (3.0.4)
Building wheels for collected packages: Colab-Utils
  Building wheel for Colab-Utils (setup.py) … done
  Created wheel for Colab-Utils: filename=Colab_Utils-0.1.dev0-py3-none-any.whl
size=10306
sha256=35e6cc6f2665a9c74c768923aeb2d1757606e69df58466c4bfddd519116604d6
  Stored in directory: /tmp/pip-ephem-wheel-cache-
ewslc7ko/wheels/eb/3c/88/465b0d78ef4a63d1f487c4208bd4691a448f05923eda0ef5f6
Successfully built Colab-Utils
Installing collected packages: Colab-Utils
Successfully installed Colab-Utils-0.1.dev0
```

1.3 Setup code

Run some setup code for this notebook: Import some useful packages and increase the default figure size.

[ ]:
```
from __future__ import print_function
from __future__ import division

import torch
import coutils
import random
import math
import matplotlib.pyplot as plt
from torchvision.utils import make_grid

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

We will use GPUs to accelerate our computation in this notebook. Run the following to make sure GPUs are enabled:

[ ]:
```
if torch.cuda.is_available:
  print('Good to go!')
else:
  print('Please set GPU via Edit -> Notebook Settings.')
```

```
Good to go!
```

The inputs to our network will be a batch of N (`num_inputs`) D-dimensional vectors (`input_size`); the hidden layer will have H hidden units (`hidden_size`), and we will predict classification scores for C categories (`num_classes`). This means that the learnable weights and biases of the network will have the following shapes:

- 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,)

We will use the following function to generate random weights for a small toy model while we implement the model:

[ ]:
```
def get_toy_data(num_inputs=5, input_size=4, hidden_size=10, num_classes=3,
                 dtype=torch.float32):
  N = num_inputs
  D = input_size
  H = hidden_size
  C = num_classes

  # We set the random seed for repeatable experiments.
  coutils.utils.fix_random_seed()

  # Generate some random parameters, storing them in a dict
```

```
params = {}
params['W1'] = 1e-4 * torch.randn(D, H, device='cuda').to(dtype)
params['b1'] = torch.zeros(H, device='cuda').to(dtype)
params['W2'] = 1e-4 * torch.randn(H, C, device='cuda').to(dtype)
params['b2'] = torch.zeros(C, device='cuda').to(dtype)

# Generate some random inputs and labels
toy_X = 10.0 * torch.randn(N, D, device='cuda').to(dtype)
toy_y = torch.tensor([0, 1, 2, 2, 1], dtype=torch.int64, device='cuda')

return toy_X, toy_y, params
```

1.4 Forward pass: compute scores

Like in the Linear Classifiers exercise, we want to write a function that takes as input the model weights and a batch of images and labels, and returns the loss and the gradient of the loss with respect to each model parameter.

However rather than attempting to implement the entire function at once, we will take a staged approach and ask you to implement the full forward and backward pass one step at a time.

First we will implement the forward pass of the network which uses the weights and biases to compute scores for all inputs:

[ ]:
```
import torch.nn.functional as F

def nn_loss_part1(params, X, y=None, reg=0.0):
    """
    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)

    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.
    - 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 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
  ␣
↪##############################################################################
    # 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
    hidden = X @ W1 + b1[None, :]
    hidden = F.relu(hidden)

    scores = hidden @ W2 + b2[None, :]
    #scores = torch.max(scores, torch.tensor([0], device=scores.device))
  ␣
↪##############################################################################
    #                              END OF YOUR CODE                             ␣
↪#
  ␣
↪##############################################################################

    return scores, hidden
```

Compute the scores and compare with the answer. The distance gap should be smaller than 1e-10.

[ ]:
```
toy_X, toy_y, params = get_toy_data()

scores, _ = nn_loss_part1(params, toy_X)
print('Your scores:')
print(scores)
print()
print('correct scores:')
```

```
correct_scores = torch.tensor([
        [ 9.7003e-08, -1.1143e-07, -3.9961e-08],
        [-7.4297e-08,  1.1502e-07,  1.5685e-07],
        [-2.5860e-07,  2.2765e-07,  3.2453e-07],
        [-4.7257e-07,  9.0935e-07,  4.0368e-07],
        [-1.8395e-07,  7.9303e-08,  6.0360e-07]], dtype=torch.float32,
 ↪device=scores.device)
print(correct_scores)
print()

# The difference should be very small. We get < 1e-10
scores_diff = (scores - correct_scores).abs().sum().item()
print('Difference between your scores and correct scores: %.2e' % scores_diff)
```

```
Your scores:
tensor([[ 9.7003e-08, -1.1143e-07, -3.9961e-08],
        [-7.4297e-08,  1.1502e-07,  1.5685e-07],
        [-2.5860e-07,  2.2765e-07,  3.2453e-07],
        [-4.7257e-07,  9.0935e-07,  4.0368e-07],
        [-1.8395e-07,  7.9303e-08,  6.0360e-07]], device='cuda:0')

correct scores:
tensor([[ 9.7003e-08, -1.1143e-07, -3.9961e-08],
        [-7.4297e-08,  1.1502e-07,  1.5685e-07],
        [-2.5860e-07,  2.2765e-07,  3.2453e-07],
        [-4.7257e-07,  9.0935e-07,  4.0368e-07],
        [-1.8395e-07,  7.9303e-08,  6.0360e-07]], device='cuda:0')

Difference between your scores and correct scores: 2.24e-11
```

1.5 Forward pass: compute loss

Now, we implement the first part of `nn_loss_part2` that computes the data and regularization loss.

For the data loss, we will use the softmax loss. For the regularization loss we will use L2 regularization on the weight matrices `W1` and `W2`; we will not apply regularization loss to the bias vectors `b1` and `b2`.

[147]:
```
def nn_loss_part2(params, X, y=None, reg=0.0):
    """
    Compute the loss and gradients for a two layer fully connected neural
    network.

    Inputs: Same as nn_loss_part1

    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, h1 = nn_loss_part1(params, X, y, reg)
    # If the targets are not given then jump out, we're done
    if y is None:
      return scores

    # Compute the loss
    loss = None
  ␣
↪###########################################################################
    # TODO: Finish the forward pass, and compute the loss. 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 Softmax          ␣
↪#
    # classifier loss. When you implment the regularization over W, 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
    scores, hidden = nn_loss_part1(params, X, y, reg)

    # Softmax calculations #
    scores_probs = scores
    scores_max = scores_probs.max(-1, keepdim=True)
    scores_probs -= scores_max.values # numerical stability
    scores_probs = torch.exp(scores_probs)
```

```
    scores_probs /= scores_probs.sum(-1, keepdim=True)
    loss = -torch.log(scores_probs[torch.arange(N), y]).sum()

    loss /= N # We want an average loss, not total loss

    # Regularization
    loss += reg * W1.square().sum()
    loss += reg * W2.square().sum()

   ␣
↪###########################################################################
    #                              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

    # Calculate gradient for W2 and b2 using bias trick #
    hidden_extended = torch.cat((
        hidden,
        torch.ones_like(hidden[:, 0:1]), # Stack 1's for bias trick.
    ), dim=1)
    dW2_extended = hidden_extended.T @ scores_probs
    class_grad = torch.zeros_like(dW2_extended.T) # Gradient at correct class
    class_grad.index_add_(0, y, hidden_extended)
    dW2_extended -= class_grad.T

    dW2_extended /= N # Average over the entire batch
    dW2, dB2 = dW2_extended[:-1], dW2_extended[-1]
    dW2 += 2 * reg * W2 # gradient from regularization
    grads['W2'] = dW2
    grads['b2'] = dB2
```

```
    # Calculate gradient for W1 #
    h_grad = scores_probs @ W2.T - W2.T[y]
    h_grad *= hidden > 0
    X_extended = torch.cat((
        X,
        torch.ones_like(X[:, 0:1]), # Stack 1's for bias trick.
    ), dim=1)
    dW1_extended = X_extended.T @ h_grad
    dW1_extended /= N # Average over the entire batch

    dW1, dB1 = dW1_extended[:-1], dW1_extended[-1]
    dW1 += 2 * reg * W1 # gradient from regularization
    grads['W1'] = dW1
    grads['b1'] = dB1



 ␣
↪###########################################################################
    #                              END OF YOUR CODE                         ␣
↪#
 ␣
↪###########################################################################

    return loss, grads
```

First, implement the forward pass in the function `nn_loss_part2` above. Then run the following to check your implementation.

We compute the loss for the toy data, and compare with the answer computed by our implementation. The difference between the correct and computed loss should be less than `1e-4`.

[148]:
```
toy_X, toy_y, params = get_toy_data()

loss, _ = nn_loss_part2(params, toy_X, toy_y, reg=0.05)
print('Your loss: ', loss.item())
correct_loss = 1.0986
print('Correct loss: ', correct_loss)
diff = (correct_loss - loss).item()

# should be very small, we get < 1e-4
print('Difference: %.4e' % diff)
```

```
Your loss:  1.0986121892929077
Correct loss:  1.0986
Difference: -1.2159e-05
```

1.6 Backward pass

Now implement the backward pass for the entire network in `nn_loss_part2`.

After doing so, we will use numeric gradient checking to see whether the analytic gradient computed by our backward pass mateches a numeric gradient.

First we define a couple utility functions for our numeric gradient check:

[149]:
```python
def compute_numeric_gradient(f, x, h=1e-7):
  """
  Compute the numeric gradient of f at x using a finite differences
  approximation. We use the centered difference:

  df/dx ~= (f(x + h) - f(x - h)) / (2 * h)

  Inputs:
  - f: A function that inputs a torch tensor and returns a torch scalar
  - x: A torch tensor giving the point at which to compute the gradient

  Returns:
  - grad: A tensor of the same shape as x giving the gradient of f at x
  """
  fx = f(x) # evaluate function value at original point
  flat_x = x.contiguous().view(-1)
  grad = torch.zeros_like(x)
  flat_grad = grad.view(-1)
  # iterate over all indexes in x
  for i in range(flat_x.shape[0]):
    oldval = flat_x[i].item() # Store the original value
    flat_x[i] = oldval + h    # Increment by h
    fxph = f(x).item()        # Evaluate f(x + h)
    flat_x[i] = oldval - h    # Decrement by h
    fxmh = f(x).item()        # Evaluate f(x - h)
    flat_x[i] = oldval        # Restore original value

    # compute the partial derivative with centered formula
    flat_grad[i] = (fxph - fxmh) / (2 * h)

  return grad


def rel_error(x, y, eps=1e-10):
  """ returns relative error between x and y """
  top = (x - y).abs().max().item()
  bot = (x.abs() + y.abs()).clamp(min=eps).max().item()
  return top / bot
```

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

You should see relative errors less than `1e-4` for all parameters.

[150]:
```
reg = 0.05
toy_X, toy_y, params = get_toy_data(dtype=torch.float64)
loss, grads = nn_loss_part2(params, toy_X, toy_y, reg=reg)

for param_name, grad in grads.items():
  param = params[param_name]
  f = lambda w: nn_loss_part2(params, toy_X, toy_y, reg=reg)[0]
  grad_numeric = compute_numeric_gradient(f, param)
  error = rel_error(grad, grad_numeric)
  print('%s max relative error: %e' % (param_name, error))
```

```
W2 max relative error: 9.052869e-07
b2 max relative error: 3.845715e-09
W1 max relative error: 1.077773e-06
b1 max relative error: 1.277993e-05
```

1.7 Train the network

To train the network we will use stochastic gradient descent (SGD), similar to the SVM and Softmax classifiers. Look at the function `nn_train` and fill in the missing sections to implement the training procedure. This should be very similar to the training procedure you used for the SVM and Softmax classifiers.

[151]:
```
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):
  X_batch = None
  y_batch = None

  #########################################################################
  # TODO: Create a random minibatch of training data and labels, storing  #
  # them in X_batch and y_batch respectively.                             #
  # hint: torch.randint                                                   #
  #########################################################################
  # Replace "pass" statement with your code
  batch_idxes = torch.randint(num_train, (batch_size,))
  X_batch = X[batch_idxes, :]
  y_batch = y[batch_idxes]
  #########################################################################
  #                             END OF YOUR CODE                          #
  #########################################################################

  # 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
    params['W1'] -= learning_rate * grads['W1']
    params['b1'] -= learning_rate * grads['b1']
    params['W2'] -= learning_rate * grads['W2']
    params['b2'] -= learning_rate * grads['b2']
    #########################################################################
    #                              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,
  }
```

You will also have to implement `nn_predict`, as the training process periodically performs prediction to keep track of accuracy over time while the network trains.

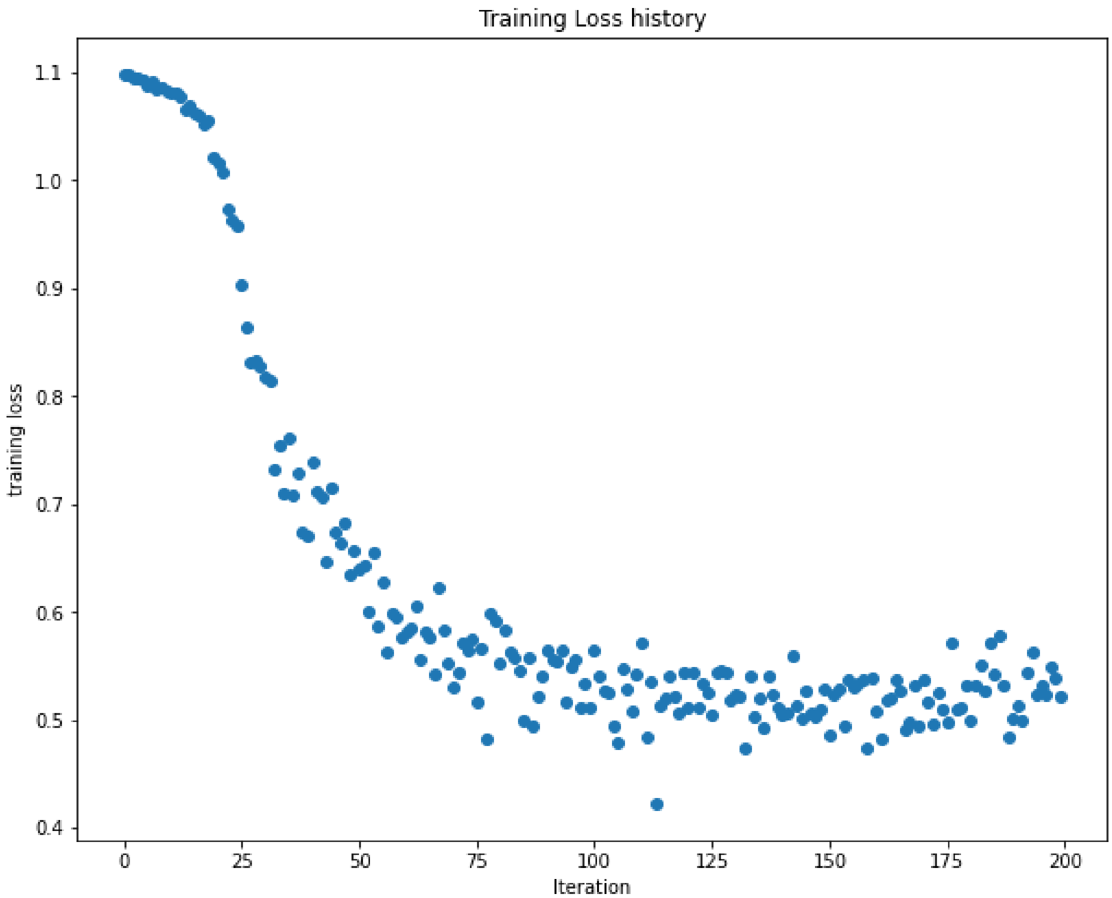
[152]:
```
def nn_predict(params, loss_func, X):
  """
  Use the trained weights of this two-layer network to predict labels for
  data points. For each data point we predict scores for each of the C
  classes, and assign each data point to the class with the highest score.

  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
  class_preds = loss_func(params, X)
  y_pred = torch.argmax(class_preds, dim=1)
  ###########################################################################
  #                              END OF YOUR CODE                           #
  ###########################################################################

  return y_pred
```

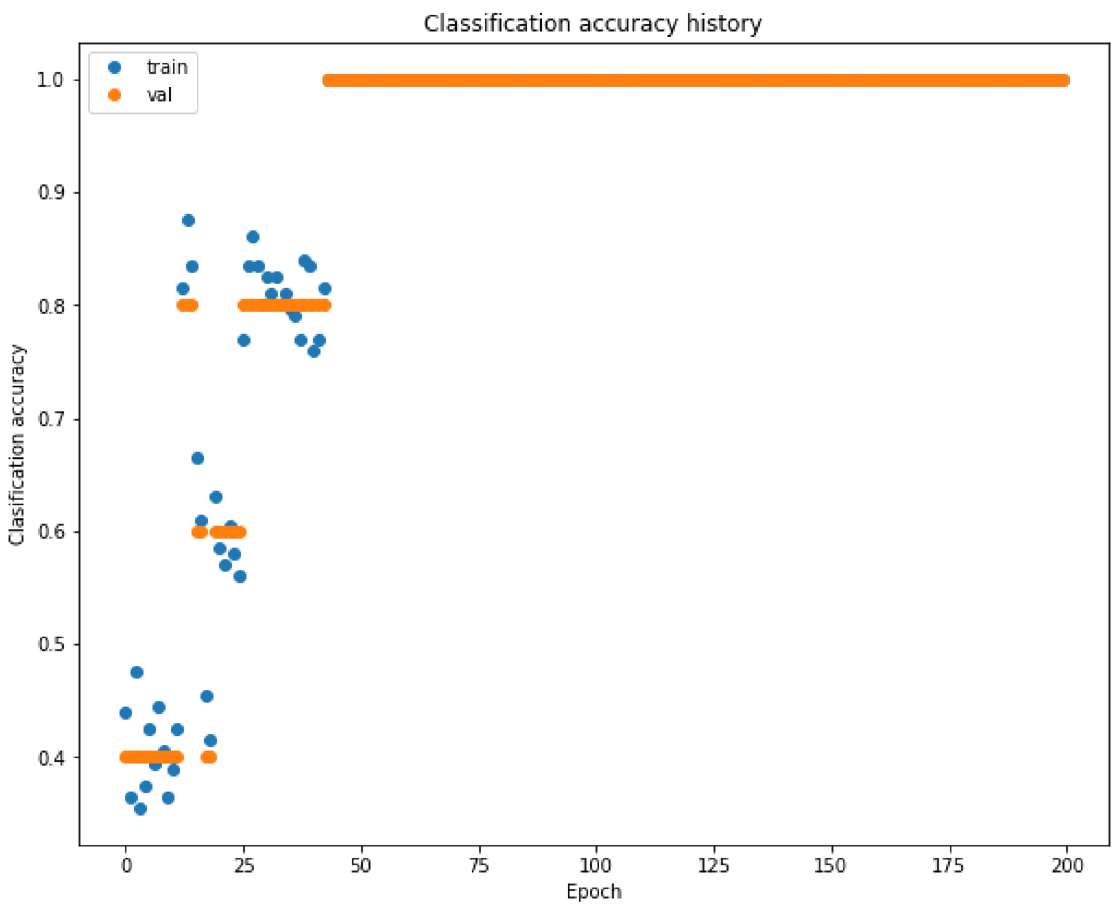
Once you have implemented the method, run the code below to train a two-layer network on toy data. Your final training loss should be less than 1.05.

[153]:
```
toy_X, toy_y, params = get_toy_data()

stats = nn_train(params, nn_loss_part2, nn_predict, toy_X, toy_y, toy_X, toy_y,
                 learning_rate=1e-1, reg=1e-6,
                 num_iters=200, verbose=False)

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

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

```
Final training loss:  0.5211756825447083
```



[154]:
```
# 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()
```



1.8 Wrap all function into a Class

We will use the class `TwoLayerNet` to represent instances of our network. The network parameters are stored in the instance variable `self.params` where keys are string parameter names and values are PyTorch tensors.

[155]:
```
class TwoLayerNet(object):
  def __init__(self, input_size, hidden_size, output_size, 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.
    """
    # fix random seed before we generate a set of parameters
    coutils.utils.fix_random_seed()

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

  def _loss(self, params, X, y=None, reg=0.0):
    return nn_loss_part2(params, X, y, reg)

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


  def _train(self, 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):
    return nn_train(params, loss_func, pred_func, X, y, X_val, y_val,
            learning_rate, learning_rate_decay,
            reg, num_iters, batch_size, verbose)

  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 self._train(self.params, self._loss, self._predict,
                       X, y, X_val, y_val,
                       learning_rate, learning_rate_decay,
                       reg, num_iters, batch_size, verbose)

  def _predict(self, params, loss_func, X):
    return nn_predict(params, loss_func, X)

  def predict(self, X):
    return self._predict(self.params, self._loss, X)
```

1.9 Load CIFAR-10 data

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

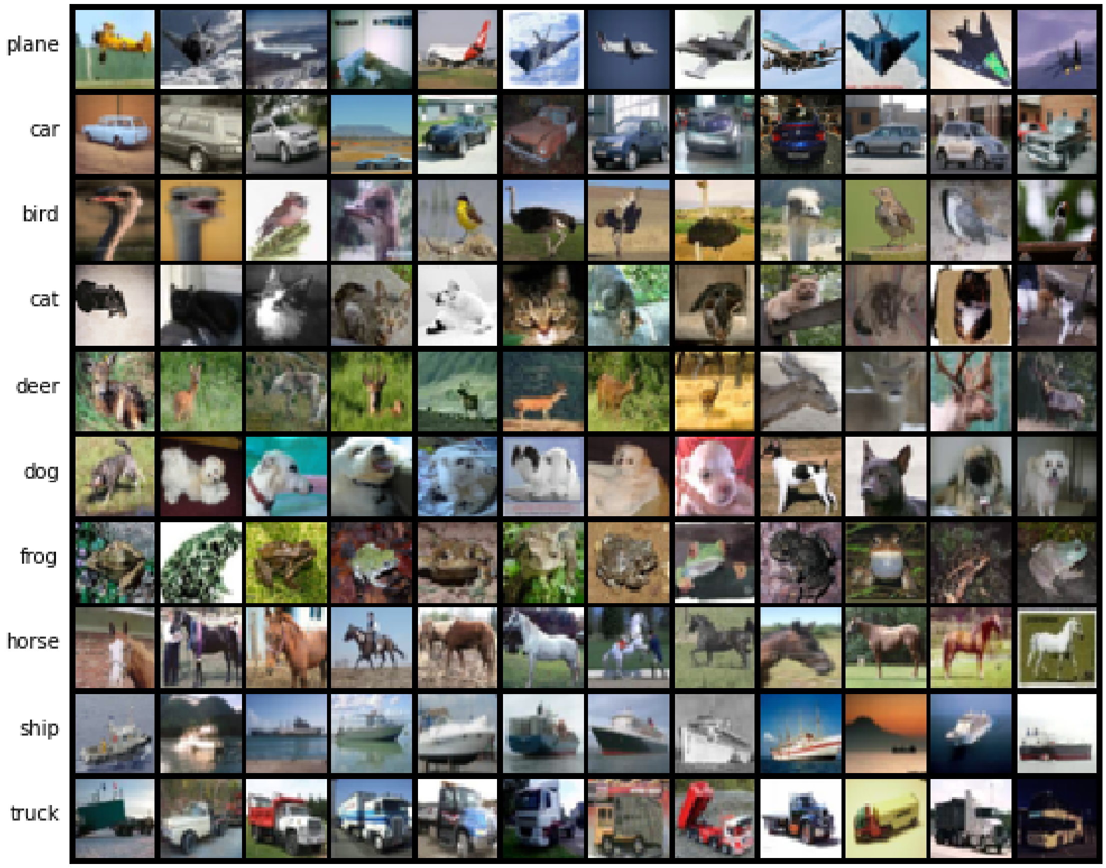
[156]:
```
def get_CIFAR10_data(validation_ratio = 0.05):
  """
  Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
  it for the linear classifier. These are the same steps as we used for the
  SVM, but condensed to a single function.
  """
  X_train, y_train, X_test, y_test = coutils.data.cifar10()

  # load every data on cuda
  X_train = X_train.cuda()
  y_train = y_train.cuda()
  X_test = X_test.cuda()
  y_test = y_test.cuda()

  # 0. Visualize some examples from the dataset.
  class_names = [
      'plane', 'car', 'bird', 'cat', 'deer',
      'dog', 'frog', 'horse', 'ship', 'truck'
  ]
  img = coutils.utils.visualize_dataset(X_train, y_train, 12, class_names)
  plt.imshow(img)
  plt.axis('off')
  plt.show()

  # 1. Normalize the data: subtract the mean RGB (zero mean)
  mean_image = X_train.mean(dim=0, keepdim=True).mean(dim=2, keepdim=True).
↪mean(dim=3, keepdim=True)
  X_train -= mean_image
  X_test -= mean_image

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

  # 3. take the validation set from the training set
  # Note: It should not be taken from the test set
  # For random permutation, you can use torch.randperm or torch.randint
  # But, for this homework, we use slicing instead.
  num_training = int( X_train.shape[0] * (1.0 - validation_ratio) )
  num_validation = X_train.shape[0] - num_training

  # return the dataset
```

```
  data_dict = {}
  data_dict['X_val'] = X_train[num_training:num_training + num_validation]
  data_dict['y_val'] = y_train[num_training:num_training + num_validation]
  data_dict['X_train'] = X_train[0:num_training]
  data_dict['y_train'] = y_train[0:num_training]

  data_dict['X_test'] = X_test
  data_dict['y_test'] = y_test
  return data_dict

# Invoke the above function to get our data.
data_dict = get_CIFAR10_data()
print('Train data shape: ', data_dict['X_train'].shape)
print('Train labels shape: ', data_dict['y_train'].shape)
print('Validation data shape: ', data_dict['X_val'].shape)
print('Validation labels shape: ', data_dict['y_val'].shape)
print('Test data shape: ', data_dict['X_test'].shape)
print('Test labels shape: ', data_dict['y_test'].shape)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
./cifar-10-python.tar.gz
```

```
  0%|          | 0/170498071 [00:00<?, ?it/s]
```

```
Extracting ./cifar-10-python.tar.gz to .
```




```
Train data shape:  torch.Size([47500, 3072])
Train labels shape:  torch.Size([47500])
Validation data shape:  torch.Size([2500, 3072])
Validation labels shape:  torch.Size([2500])
Test data shape:  torch.Size([10000, 3072])
Test labels shape:  torch.Size([10000])
```

1.10 Train a network

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

[157]:
```
input_size = 3 * 32 * 32
hidden_size = 36
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network
stats = net.train(data_dict['X_train'], data_dict['y_train'],
```

```
                  data_dict['X_val'], data_dict['y_val'],
                  num_iters=500, batch_size=1000,
                  learning_rate=1e-2, learning_rate_decay=0.95,
                  reg=0.25, verbose=True)

# Predict on the validation set
y_val_pred = net.predict(data_dict['X_val'])
val_acc = 100.0 * (y_val_pred == data_dict['y_val']).float().mean().item()
print('Validation accuracy: %.2f%%' % val_acc)
```

```
iteration 0 / 500: loss 2.302864
iteration 100 / 500: loss 2.302708
iteration 200 / 500: loss 2.302672
iteration 300 / 500: loss 2.302584
iteration 400 / 500: loss 2.302634
Validation accuracy: 8.76%
```

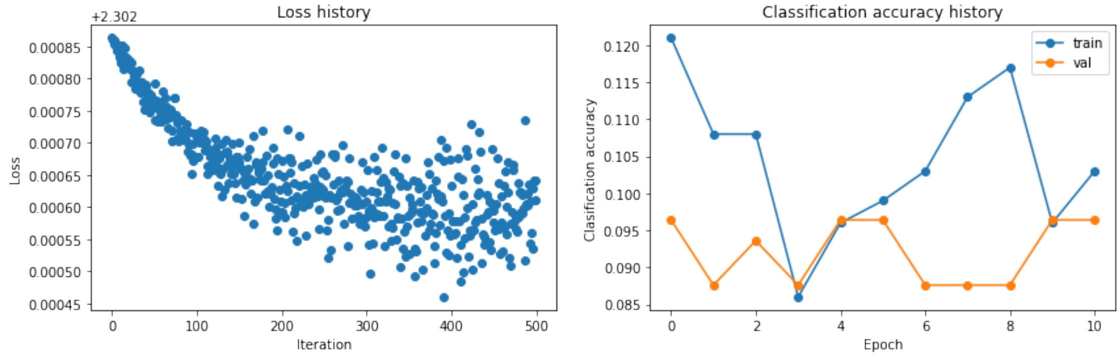
1.11 Debug the training

With the default parameters we provided above, you should get a validation accuracy of about 8.76% on the validation set. This isn't very good.

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

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

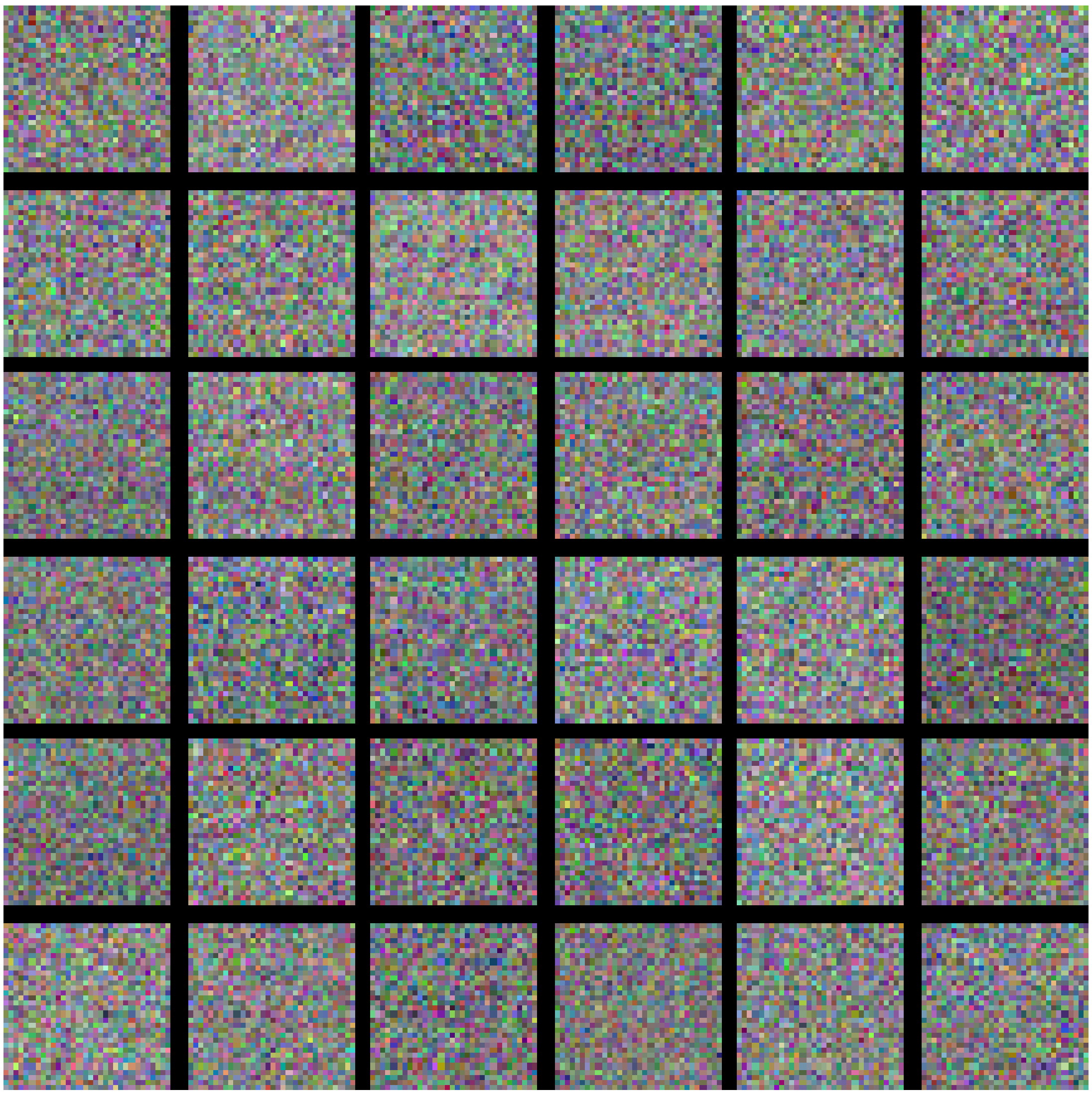
[158]:
```
# Plot the loss function and train / validation accuracies
def plot_stats(stat_dict):
  plt.subplot(1, 2, 1)
  plt.plot(stat_dict['loss_history'], 'o')
  plt.title('Loss history')
  plt.xlabel('Iteration')
  plt.ylabel('Loss')

  plt.subplot(1, 2, 2)
  plt.plot(stat_dict['train_acc_history'], 'o-', label='train')
  plt.plot(stat_dict['val_acc_history'], 'o-', label='val')
  plt.title('Classification accuracy history')
  plt.xlabel('Epoch')
  plt.ylabel('Clasification accuracy')
  plt.legend()

  plt.gcf().set_size_inches(14, 4)
  plt.show()

plot_stats(stats)
```



Similar to SVM and Softmax classifier, let's visualize the weights.

[159]:
```
def visualize_grid(Xs, ubound=255.0, padding=1):
  """
  Reshape a 4D tensor of image data to a grid for easy visualization.

  Inputs:
  - Xs: Data of shape (N, H, W, C)
  - ubound: Output grid will have values scaled to the range [0, ubound]
  - padding: The number of blank pixels between elements of the grid
  """
  (N, H, W, C) = Xs.shape
  # print(Xs.shape)
  grid_size = int(math.ceil(math.sqrt(N)))
  grid_height = H * grid_size + padding * (grid_size - 1)
  grid_width = W * grid_size + padding * (grid_size - 1)
  grid = torch.zeros((grid_height, grid_width, C), device=Xs.device)
  next_idx = 0
  y0, y1 = 0, H
  for y in range(grid_size):
    x0, x1 = 0, W
    for x in range(grid_size):
      if next_idx < N:
        img = Xs[next_idx]
        low, high = torch.min(img), torch.max(img)
        grid[y0:y1, x0:x1] = ubound * (img - low) / (high - low)
        # grid[y0:y1, x0:x1] = Xs[next_idx]
        next_idx += 1
      x0 += W + padding
      x1 += W + padding
    y0 += H + padding
    y1 += H + padding
  # print(grid.shape)
  return grid
```

```python
# Visualize the weights of the network
def show_net_weights(net):
  W1 = net.params['W1']
  W1 = W1.reshape(3, 32, 32, -1).transpose(0, 3)
  plt.imshow(visualize_grid(W1, padding=3).type(torch.uint8).cpu())
  plt.gca().axis('off')
  plt.show()

show_net_weights(net)
```



1.12 What's wrong?

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

1.12.1 Capacity?

Our initial model has very similar performance on the training and validation sets. This suggests that the model is underfitting, and that its performance might improve if we were to increase its capacity.

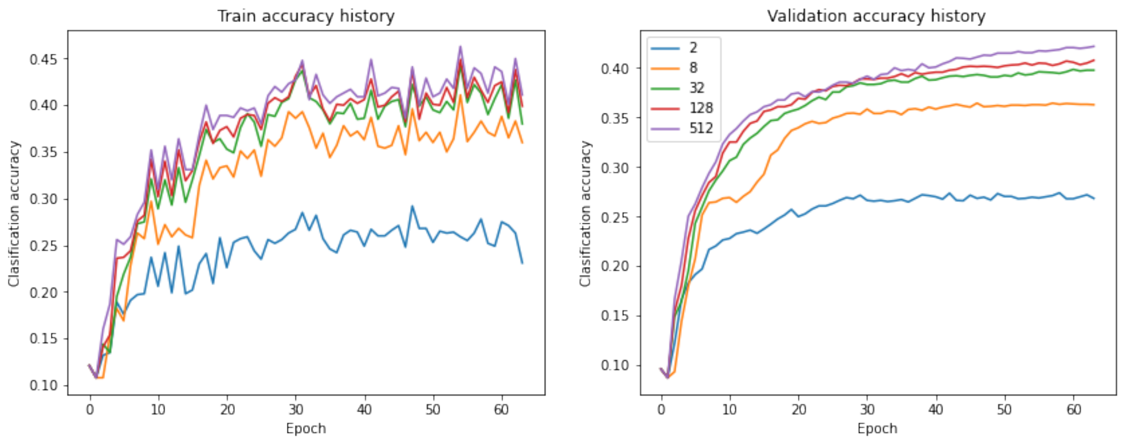
One way we can increase the capacity of a neural network model is to increase the size of its hidden layer. Here we investigate the effect of increasing the size of the hidden layer. The performance (as measured by validation-set accuracy) should increase as the size of the hidden layer increases; however it may show diminishing returns for larger layer sizes.

[160]:
```
def plot_acc_curves(stat_dict):
  plt.subplot(1, 2, 1)
  for key, single_stats in stat_dict.items():
    plt.plot(single_stats['train_acc_history'], label=str(key))
  plt.title('Train accuracy history')
  plt.xlabel('Epoch')
  plt.ylabel('Clasification accuracy')

  plt.subplot(1, 2, 2)
  for key, single_stats in stat_dict.items():
    plt.plot(single_stats['val_acc_history'], label=str(key))
  plt.title('Validation accuracy history')
  plt.xlabel('Epoch')
  plt.ylabel('Clasification accuracy')
  plt.legend()

  plt.gcf().set_size_inches(14, 5)
  plt.show()
```

[161]:
```
hidden_sizes = [2, 8, 32, 128, 512]
lr = 0.1
reg = 0.001

stat_dict = {}
for hs in hidden_sizes:
  print('train with hidden size: {}'.format(hs))
  net = TwoLayerNet(3 * 32 * 32, hs, 10, device=data_dict['X_train'].device)
  stats = net.train(data_dict['X_train'], data_dict['y_train'],␣
 ↪data_dict['X_val'], data_dict['y_val'],
```

```
            num_iters=3000, batch_size=1000,
            learning_rate=lr, learning_rate_decay=0.95,
            reg=reg, verbose=False)
  stat_dict[hs] = stats

plot_acc_curves(stat_dict)
```

```
train with hidden size: 2
train with hidden size: 8
train with hidden size: 32
train with hidden size: 128
train with hidden size: 512
```



1.12.2 Regularization?

Another possible explanation for the small gap we saw between the train and validation accuracies of our model is regularization. In particular, if the regularization coefficient were too high then the model may be unable to fit the training data.

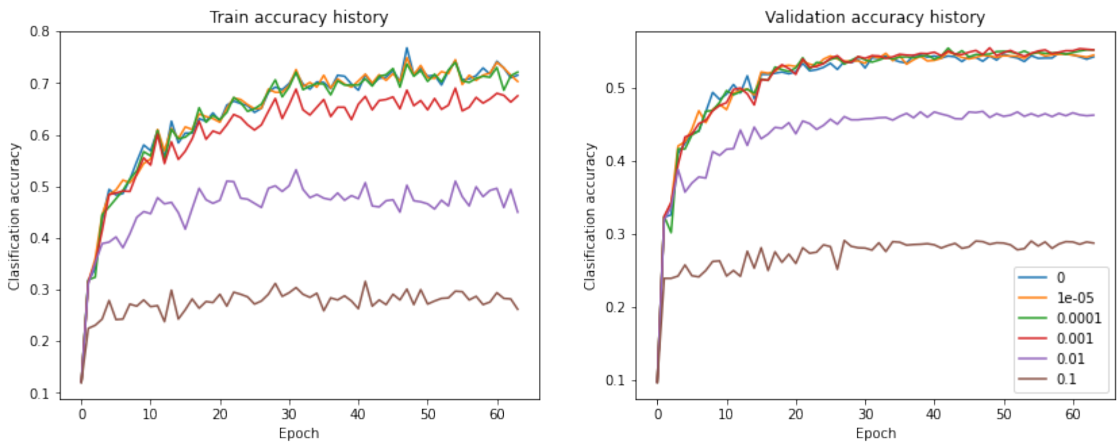
We can investigate the phenomenon empirically by training a set of models with varying regularization strengths while fixing other hyperparameters.

You should see that setting the regularization strength too high will harm the validation-set performance of the model:

[162]:
```
hs = 128
lr = 1.0
regs = [0, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1]

stat_dict = {}
for reg in regs:
  print('train with regularization: {}'.format(reg))
  net = TwoLayerNet(3 * 32 * 32, hs, 10, device=data_dict['X_train'].device)
```

```
  stats = net.train(data_dict['X_train'], data_dict['y_train'],␣
↪data_dict['X_val'], data_dict['y_val'],
            num_iters=3000, batch_size=1000,
            learning_rate=lr, learning_rate_decay=0.95,
            reg=reg, verbose=False)
  stat_dict[reg] = stats

plot_acc_curves(stat_dict)
```

```
train with regularization: 0
train with regularization: 1e-05
train with regularization: 0.0001
train with regularization: 0.001
train with regularization: 0.01
train with regularization: 0.1
```



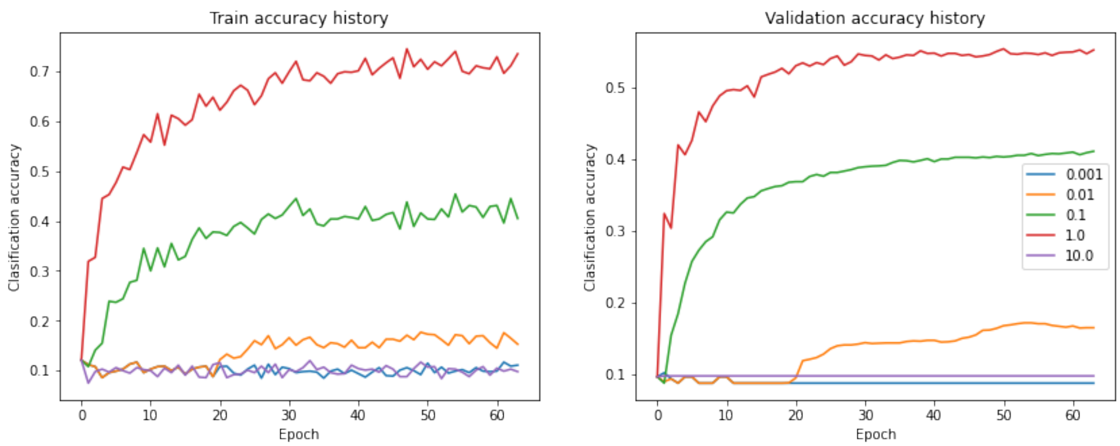
1.12.3 Learning Rate?

Last but not least, we also want to see the effect of learning rate with respect to the performance.

[163]:
```
hs = 128
lrs = [1e-3, 1e-2, 1e-1, 1e0, 1e1]
reg = 1e-4

stat_dict = {}
for lr in lrs:
  print('train with learning rate: {}'.format(lr))
  net = TwoLayerNet(3 * 32 * 32, hs, 10, device=data_dict['X_train'].device)
  stats = net.train(data_dict['X_train'], data_dict['y_train'],␣
↪data_dict['X_val'], data_dict['y_val'],
            num_iters=3000, batch_size=1000,
            learning_rate=lr, learning_rate_decay=0.95,
```

```
            reg=reg, verbose=False)
  stat_dict[lr] = stats

plot_acc_curves(stat_dict)
```

```
train with learning rate: 0.001
train with learning rate: 0.01
train with learning rate: 0.1
train with learning rate: 1.0
train with learning rate: 10.0
```



1.13 Tune your hyperparameters

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, number of training epochs, and regularization strength. You might also consider tuning the learning rate decay, but you should be able to get good performance using the default value.

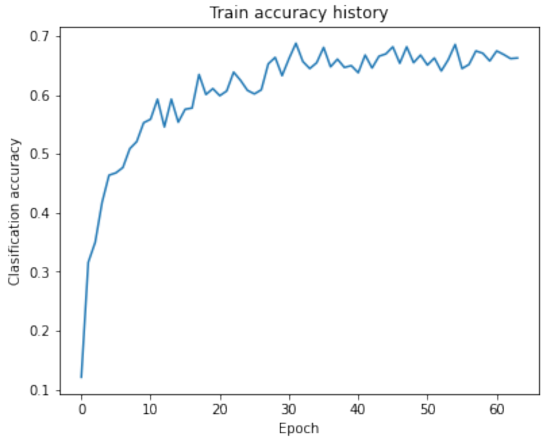
Plots. To guide your hyperparameter search, you might consider making auxiliary plots of training and validation performance as above, or plotting the results arising from different hyperparameter combinations as we did in the Linear Classifier notebook. You should feel free to plot any auxiliary results you need in order to find a good network, but we don't require any particular plots from you.

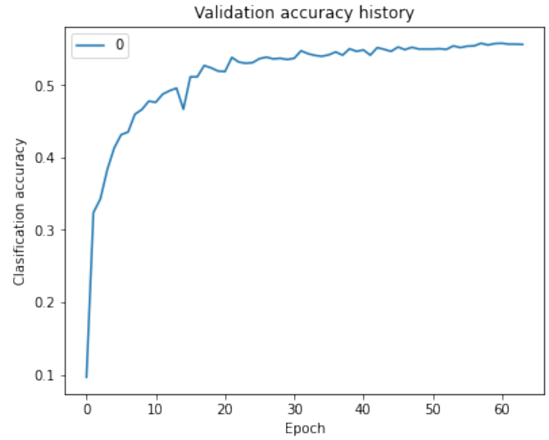
Approximate results. To get full credit for the assignment, you should achieve a classification accuracy above 50% on the validation set.

(Our best model gets a validation-set accuracy above 58% – did you beat us?)

[167]:
```
best_net = None # store the best model into this

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

```
# TODO: Tune hyperparameters using the validation set. Store your best trained ␣
↪#
# model in best_net.                                                          ␣
↪#
#                                                                             ␣
↪#
# To help debug your network, it may help to use visualizations similar to the ␣
↪#
# ones we used above; these visualizations will have significant qualitative  ␣
↪#
# differences from the ones we saw above for the poorly tuned network.        ␣
↪#
#                                                                             ␣
↪#
# Tweaking hyperparameters by hand can be fun, but you might find it useful to ␣
↪#
# write code to sweep through possible combinations of hyperparameters        ␣
↪#
# automatically like we did on the previous exercises.                        ␣
↪#
################################################################################
# Replace "pass" statement with your code
reg, lr, hs = 0.001, 1.0, 128
best_net = TwoLayerNet(3 * 32 * 32, hs, 10, device=data_dict['X_train'].device)
stats = best_net.train(data_dict['X_train'], data_dict['y_train'],␣
↪data_dict['X_val'], data_dict['y_val'],
          num_iters=3000, batch_size=1000,
          learning_rate=lr, learning_rate_decay=0.95,
          reg=reg, verbose=False)
stat_dict = {0: stats}

plot_acc_curves(stat_dict)
################################################################################
#                              END OF YOUR CODE                               ␣
↪#
################################################################################
```





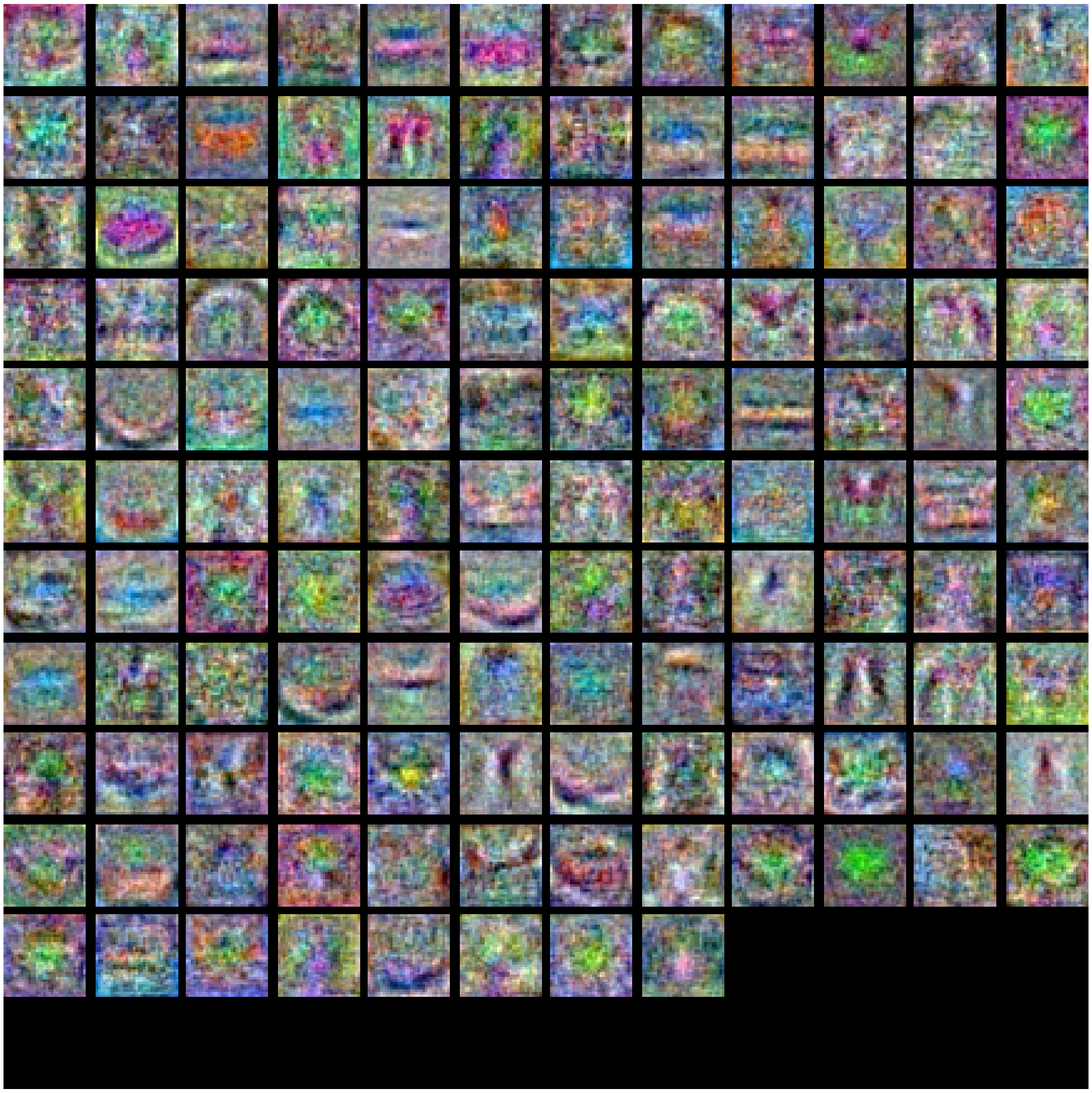
[168]:

```
# Check the validation-set accuracy of your best model
y_val_preds = best_net.predict(data_dict['X_val'])
val_acc = 100 * (y_val_preds == data_dict['y_val']).float().mean().item()
print('Best val-set accuracy: %.2f%%' % val_acc)
```

```
Best val-set accuracy: 55.36%
```

[169]:

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



1.14 Run on the test set

When you are done experimenting, you should evaluate your final trained network on the test set. To get full credit for the assignment, you should achieve over 50% classification accuracy on the test set.

(Our best model gets 54.1% test-set accuracy – did you beat us?)

[170]:
```
y_test_preds = best_net.predict(data_dict['X_test'])
test_acc = 100 * (y_test_preds == data_dict['y_test']).float().mean().item()
print('Test accuracy: %.2f%%' % test_acc)
```

```
Test accuracy: 54.13%
```