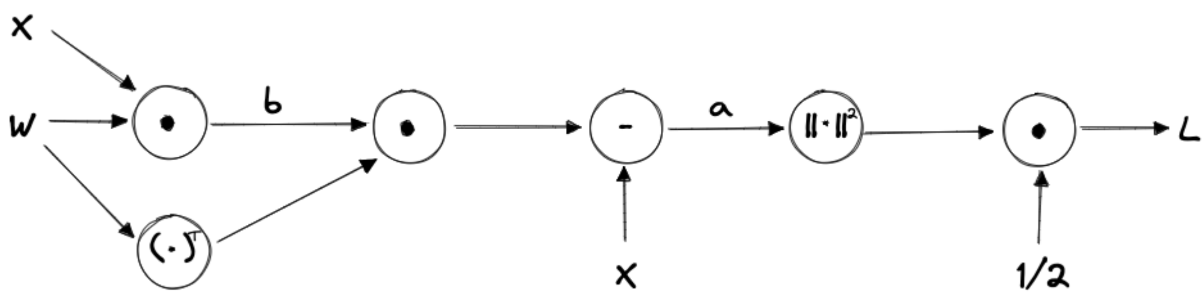
Problem 1

(a) The loss is defined as proportional to the difference between $\mathbf{W}^T\mathbf{W}\mathbf{x}$ and $\mathbf{x}$. We can think of $\mathbf{W}\mathbf{x}$ as bringing $\mathbf{x}$ down to a much lower dimensional space, then $\mathbf{W}^T\mathbf{W}\mathbf{x}$ as bringing that back to the original dimensionality. Thus, if we minimize this loss function, then $\mathbf{W}\mathbf{x}$ should preserve information about $\mathbf{x}$ in order to minimize the difference between the reconstituted vector and the original vector.

(b) The computational graph for $\mathcal{L}$ is as follows:



(c) By the law of total derivatives, the individual gradients of each path to $\mathbf{W}$ should be summed together. That is, if we let $\mathbf{a} = \mathbf{W}^T\mathbf{W}\mathbf{x} - \mathbf{x}$, then $\nabla_{\mathbf{W}}\mathcal{L} = \frac{\partial \mathbf{W}\mathbf{x}}{\partial \mathbf{W}}\frac{\partial \mathbf{a}}{\partial \mathbf{W}\mathbf{x}}\frac{\partial \mathcal{L}}{\partial \mathbf{a}} + \frac{\partial \mathbf{W}^T}{\partial \mathbf{W}}\frac{\partial \mathbf{a}}{\partial \mathbf{W}^T}\frac{\partial \mathcal{L}}{\partial \mathbf{a}}$.

(d) Let $\mathcal{L} = \frac{1}{2}||\mathbf{W}^T\mathbf{W}\mathbf{x} - \mathbf{x}||^2$. Also, let $\mathbf{a} = \mathbf{W}^T\mathbf{W}\mathbf{x} - \mathbf{x}$ and $\mathbf{b} = \mathbf{W}\mathbf{x}$.

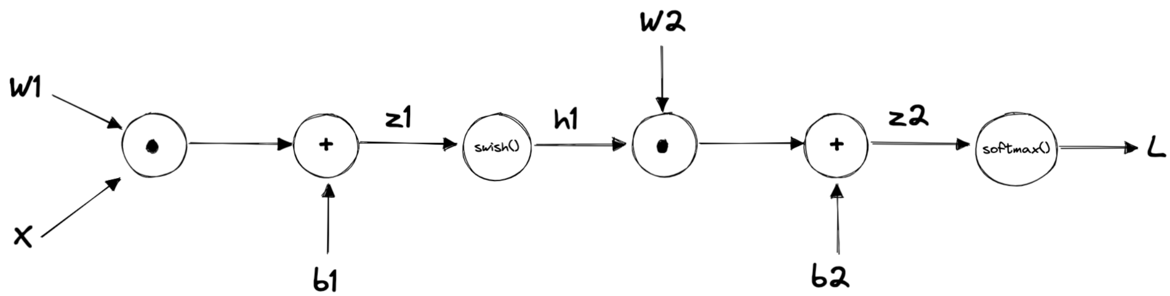
$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathbf{a}} &= \mathbf{a} \\
\frac{\partial \mathcal{L}}{\partial \mathbf{W}^T} &= \frac{\partial \mathbf{a}}{\partial \mathbf{W}^T}\frac{\partial \mathcal{L}}{\partial \mathbf{a}} = \mathbf{a}(\mathbf{W}\mathbf{x})^T \\
\frac{\partial \mathcal{L}}{\partial \mathbf{W}}_{\text{path 1}} &= \mathbf{W}\mathbf{x}\mathbf{a}^T \\
\frac{\partial \mathcal{L}}{\partial \mathbf{b}} &= \frac{\partial \mathbf{a}}{\partial \mathbf{b}}\frac{\partial \mathcal{L}}{\partial \mathbf{a}} = \mathbf{W}\mathbf{a} \\
\frac{\partial \mathcal{L}}{\partial \mathbf{W}}_{\text{path 2}} &= \frac{\partial \mathbf{b}}{\partial \mathbf{W}}\frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \mathbf{W}\mathbf{a}\mathbf{x}^T \\
\nabla_{\mathbf{W}}\mathcal{L} &= \mathbf{W}\mathbf{a}\mathbf{x}^T + \mathbf{W}\mathbf{x}\mathbf{a}^T \\
&= \mathbf{W}(\mathbf{W}^T\mathbf{W}\mathbf{x} - \mathbf{x})\mathbf{x}^T + \mathbf{W}\mathbf{x}(\mathbf{W}^T\mathbf{W}\mathbf{x} - \mathbf{x})^T
\end{aligned}
$$

Problem 2

I am a C147 student.

Problem 3

(a) The computational graph for L is as follows:



(b) Assume we already know $\frac{\partial L}{\partial z_2}$.

We know that $z_2 = W_2 h_1 + b_2$. We can first backpropagate to b_2, which is a simple addition operation. Therefore, $\nabla_{b_2} L = \frac{\partial z_2}{\partial b_2} = \frac{\partial L}{\partial z_2}$. This also means that $\frac{\partial L}{\partial W_2 h_1} = \frac{\partial L}{\partial z_2}$.

Then, we can backpropagate $\frac{\partial L}{\partial z_2}$ to $W_2 h_1$. Using the derivation from class, we know that $\frac{\partial L}{\partial W_2} = \frac{\partial W_2 h_1}{\partial W_2} \frac{\partial L}{\partial W_2 h_1} = \frac{\partial L}{\partial z_2} h_1^T$.

In summary, using backpropagation, we derive the gradients

$$\nabla_{b_2} L = \frac{\partial L}{\partial z_2},$$
$$\nabla_{W_2} L = \frac{\partial L}{\partial z_2} h_1^T.$$

(c) Continuing off from part (b), we can backpropagate $\frac{\partial L}{\partial z_2}$ to $W_2 h_1$, where if we use the derivation from class, we know that $\frac{\partial L}{\partial h_1} = \frac{\partial W_2 h_1}{\partial h_1} \frac{\partial L}{\partial W_2 h_1} = W_2^T \frac{\partial L}{\partial z_2}$.

h_1 is defined as being equal to Swish(z_1), where for a neuron $z \in z_1$, $\text{Swish}(z) = z\sigma(z)$. Then, $\frac{d}{dz}\text{Swish}(z) = \frac{d}{dz} z\sigma(z) = \sigma(z) + z\frac{d\sigma(z)}{z} = \sigma(z) + z\sigma(z)(1-\sigma(z)) = \text{Swish}(z) + \sigma(z)(1 - \text{Swish}(z))$. Therefore, $\frac{\partial h_1}{\partial z_1} = \text{Swish}(z_1) + \sigma(z_1)(1 - \text{Swish}(z_1))$, and $\frac{\partial L}{\partial z_1} = \frac{\partial h_1}{\partial z_1} \frac{\partial L}{\partial h_1} = \frac{\partial h_1}{\partial z_1} W_2^T \frac{\partial L}{\partial z_2}$.

We know that $z_1 = W_1 x + b_1$, so $\frac{\partial L}{\partial b_1} = \frac{\partial z_1}{\partial b_1} \frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial z_1}$. This also means that $\frac{\partial L}{\partial W_1 x} = \frac{\partial L}{\partial z_1}$.

Finally, we can backpropagate $\frac{\partial L}{\partial z_1}$ to $W_1 x$. Using the derivation from class, we know that $\frac{\partial L}{\partial W_1} = \frac{\partial W_1 x}{\partial W_1} \frac{\partial L}{\partial W_1 x} = \frac{\partial L}{\partial z_1} x^T$.

In summary, using backpropagation we derive the gradients

$$\nabla_{b_1} L = \frac{\partial L}{\partial z_1},$$
$$\nabla_{W_1} L = \frac{\partial L}{\partial z_1} x^T, \text{ where}$$
$$\frac{\partial L}{\partial z_1} = \Big[\text{Swish}(z_1) + \sigma(z_1)(1 - \text{Swish}(z_1))\Big] \odot W_2^T \frac{\partial L}{\partial z_2}.$$

Problem 4

neural_net.py:

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


class TwoLayerNet(object):
  """
  A two-layer fully-connected neural network. The net has an input dimension of
  D, a hidden layer dimension of H, and performs classification over C classes.
  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.
  """

  def __init__(self, input_size, hidden_size, output_size, 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 (H, D)
    b1: First layer biases; has shape (H,)
    W2: Second layer weights; has shape (C, H)
    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.
    """
    self.params = {}
    self.params['W1'] = std * np.random.randn(hidden_size, input_size)
    self.params['b1'] = np.zeros(hidden_size)
    self.params['W2'] = std * np.random.randn(output_size, hidden_size)
    self.params['b2'] = np.zeros(output_size)


  def loss(self, X, y=None, reg=0.0):
    """
    Compute the loss and gradients for a two layer fully connected neural
    network.

    Inputs:
    - 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
```

```python
      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 matrix 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 = self.params['W1'], self.params['b1']
    W2, b2 = self.params['W2'], self.params['b2']
    N, D = X.shape

    # Compute the forward pass
    scores = None

    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the output scores of the neural network.  The result
    #   should be (N, C). As stated in the description for this class,
    #   there should not be a ReLU layer after the second FC layer.
    #   The output of the second FC layer is the output scores. Do not
    #   use a for loop in your implementation.
    # ================================================================ #

    relu_out = np.maximum(0, (X @ W1.T + b1))
    scores = relu_out @ W2.T + b2

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


    # If the targets are not given then jump out, we're done
    if y is None:
      return scores

    # Compute the loss
    loss = None

    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the loss of the neural network.  This includes the
    #   softmax loss and the L2 regularization for W1 and W2. Store the
    #   total loss in teh variable loss.  Multiply the regularization
    #   loss by 0.5 (in addition to the factor reg).
    # ================================================================ #
```

```python
    # scores is num_examples by num_classes

    scores_exp = np.exp(scores)
    probs = scores_exp / np.sum(scores_exp, axis=1, keepdims=True)
    loss = -np.sum(np.log(probs[np.arange(N), y])) / N
    loss += 0.5*reg*(np.sum(W1**2) + np.sum(W2**2))

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

    grads = {}

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the backward pass.  Compute the derivatives of the
    #   weights and the biases.  Store the results in the grads
    #   dictionary.  e.g., grads['W1'] should store the gradient for
    #   W1, and be of the same size as W1.
    # ================================================================ #

    # A is the (N x C) matrix of activations i.e. the outputs of the second FC layer
    dLdA = probs
    dLdA[np.arange(N), y] -= 1
    dLdA /= N
    grads['W2'] = dLdA.T @ relu_out + reg*W2
    grads['b2'] = np.sum(dLdA, axis=0)

    # B is the (N x H) matrix output of the first FC layer
    dLdB = (X @ W1.T > 0) * (dLdA @ W2)
    grads['W1'] = dLdB.T @ X + reg*W1
    grads['b1'] = np.sum(dLdB, axis=0)

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

    return loss, grads

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

    Inputs:
    - X: A numpy array of shape (N, D) giving training data.
    - y: A numpy array f shape (N,) giving training labels; y[i] = c means that
      X[i] has label c, where 0 <= c < C.
    - X_val: A numpy array of shape (N_val, D) giving validation data.
    - y_val: A numpy array of shape (N_val,) giving validation labels.
    - learning_rate: Scalar giving learning rate for optimization.
```

```python
    - 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.
    """
    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 np.arange(num_iters):
      X_batch = None
      y_batch = None

      # ================================================================ #
      # YOUR CODE HERE:
      #   Create a minibatch by sampling batch_size samples randomly.
      # ================================================================ #

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

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

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

      # ================================================================ #
      # YOUR CODE HERE:
      #   Perform a gradient descent step using the minibatch to update
      #   all parameters (i.e., W1, W2, b1, and b2).
      # ================================================================ #

      for p in self.params:
          self.params[p] -= learning_rate * grads[p]

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

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

      # Every epoch, check train and val accuracy and decay learning rate.
      if it % iterations_per_epoch == 0:
```

```python
            # Check accuracy
            train_acc = (self.predict(X_batch) == y_batch).mean()
            val_acc = (self.predict(X_val) == y_val).mean()
            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,
    }

  def predict(self, 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:
    - X: A numpy array of shape (N, D) giving N D-dimensional data points to
      classify.

    Returns:
    - y_pred: A numpy array 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

    # ================================================================ #
    # YOUR CODE HERE:
    #   Predict the class given the input data.
    # ================================================================ #

    relu_out = np.maximum(0, (X @ self.params['W1'].T) + self.params['b1'])
    scores = (relu_out @ self.params['W2'].T) + self.params['b2']
    y_pred = np.argmax(scores, axis=1)

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

    return y_pred
```

This is the 2-layer neural network notebook for ECE C147/C247 Homework #3

Please follow the notebook linearly to implement a two layer neural network.

Please print out the notebook entirely when completed.

The goal of this notebook is to give you experience with training a two layer neural network.

In [1]:
```
import random
import numpy as np
from utils.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

%matplotlib inline
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

Toy example

Before loading CIFAR-10, there will be a toy example to test your implementation of the forward and backward pass. Make sure to read the description of TwoLayerNet class in neural_net.py file , understand the architecture and initializations

In [11]:
```
from nndl.neural_net import TwoLayerNet
```

In [12]:
```
# Create a small net and some toy data to check your implementations.
# Note that we set the random seed for repeatable experiments.

input_size = 4
hidden_size = 10
num_classes = 3
num_inputs = 5

def init_toy_model():
    np.random.seed(0)
    return TwoLayerNet(input_size, hidden_size, num_classes, std=1e-1)

def init_toy_data():
    np.random.seed(1)
    X = 10 * np.random.randn(num_inputs, input_size)
    y = np.array([0, 1, 2, 2, 1])
    return X, y
```

```
net = init_toy_model()
X, y = init_toy_data()
```

Compute forward pass scores

In [13]:
```
## Implement the forward pass of the neural network.
## See the loss() method in TwoLayerNet class for the same

# Note, there is a statement if y is None: return scores, which is why
# the following call will calculate the scores.
scores = net.loss(X)
print('Your scores:')
print(scores)
print()
print('correct scores:')
correct_scores = np.asarray([
    [-1.07260209,  0.05083871, -0.87253915],
    [-2.02778743, -0.10832494, -1.52641362],
    [-0.74225908,  0.15259725, -0.39578548],
    [-0.38172726,  0.10835902, -0.17328274],
    [-0.64417314, -0.18886813, -0.41106892]])
print(correct_scores)
print()

# The difference should be very small. We get < 1e-7
print('Difference between your scores and correct scores:')
print(np.sum(np.abs(scores - correct_scores)))
```

```
Your scores:
[[-1.07260209  0.05083871 -0.87253915]
 [-2.02778743 -0.10832494 -1.52641362]
 [-0.74225908  0.15259725 -0.39578548]
 [-0.38172726  0.10835902 -0.17328274]
 [-0.64417314 -0.18886813 -0.41106892]]

correct scores:
[[-1.07260209  0.05083871 -0.87253915]
 [-2.02778743 -0.10832494 -1.52641362]
 [-0.74225908  0.15259725 -0.39578548]
 [-0.38172726  0.10835902 -0.17328274]
 [-0.64417314 -0.18886813 -0.41106892]]

Difference between your scores and correct scores:
3.381231197113754e-08
```

Forward pass loss

In [14]:
```
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.071696123862817

# should be very small, we get < 1e-12
print("Loss:",loss)
print('Difference between your loss and correct loss:')
print(np.sum(np.abs(loss - correct_loss)))
```

```
Loss: 1.071696123862817
Difference between your loss and correct loss:
0.0
```

Backward pass

Implements the backwards pass of the neural network. Check your gradients with the gradient check utilities provided.

In [15]:

```
from utils.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the backward
# If your implementation is correct, the difference between the numeric and
# analytic gradients should be less than 1e-8 for each of W1, W2, b1, and b2

loss, grads = net.loss(X, y, reg=0.05)

# these should all be less than 1e-8 or so
for param_name in grads:
    f = lambda W: net.loss(X, y, reg=0.05)[0]
    param_grad_num = eval_numerical_gradient(f, net.params[param_name], verb
    print('{} max relative error: {}'.format(param_name, rel_error(param_gra
```

```
W2 max relative error: 2.9632233460136427e-10
b2 max relative error: 1.2482624742512528e-09
W1 max relative error: 1.283285235125835e-09
b1 max relative error: 3.172680285697327e-09
```

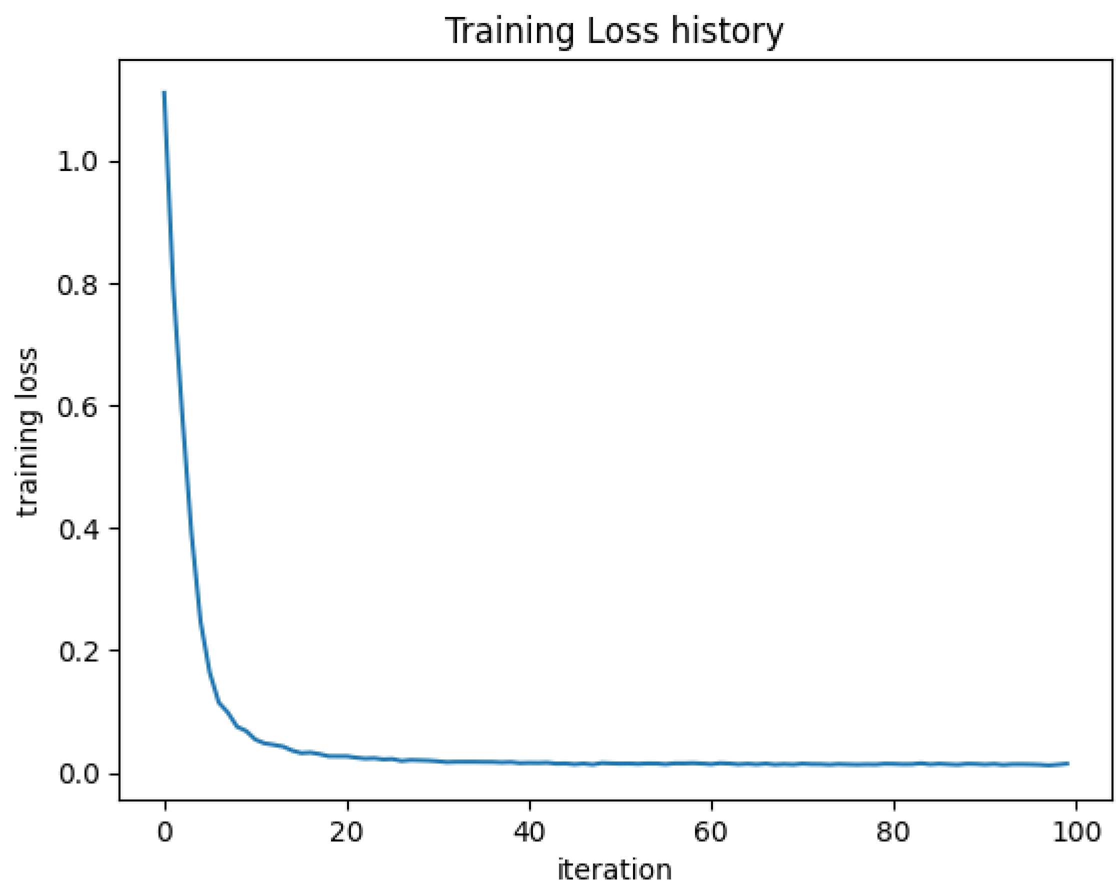
Training the network

Implement neural_net.train() to train the network via stochastic gradient descent, much like the softmax and SVM.

In [16]:

```
net = init_toy_model()
stats = net.train(X, y, X, y,
            learning_rate=1e-1, reg=5e-6,
            num_iters=100, verbose=False)

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()
```

```
Final training loss:  0.014498902952971663
```



Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

In [17]:
```
from utils.data_utils import load_CIFAR10

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000)
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the two-layer neural net classifier.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = '../hw2/cifar-10-batches-py'
    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # Subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]
```

```
    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis=0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image

    # Reshape data to rows
    X_train = X_train.reshape(num_training, -1)
    X_val = X_val.reshape(num_validation, -1)
    X_test = X_test.reshape(num_test, -1)

    return X_train, y_train, X_val, y_val, X_test, y_test


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

```
Train data shape:  (49000, 3072)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3072)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3072)
Test labels shape:  (1000,)
```

Running SGD

If your implementation is correct, you should see a validation accuracy of around 28-29%.

In [18]:

```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network
stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=1000, batch_size=200,
            learning_rate=1e-4, learning_rate_decay=0.95,
            reg=0.25, verbose=True)

# Predict on the validation set
val_acc = (net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)

# Save this net as the variable subopt_net for later comparison.
subopt_net = net
```

```
iteration 0 / 1000: loss 2.302757518613176
iteration 100 / 1000: loss 2.302120159207236
iteration 200 / 1000: loss 2.2956136007408703
iteration 300 / 1000: loss 2.2518259043164135
iteration 400 / 1000: loss 2.188995235046776
iteration 500 / 1000: loss 2.1162527791897743
iteration 600 / 1000: loss 2.064670827698217
iteration 700 / 1000: loss 1.990168862308394
iteration 800 / 1000: loss 2.002827640124685
iteration 900 / 1000: loss 1.9465176817856495
Validation accuracy:  0.283
```

Questions:

The training accuracy isn't great.

(1) What are some of the reasons why this is the case? Take the following cell to do some analyses and then report your answers in the cell following the one below.

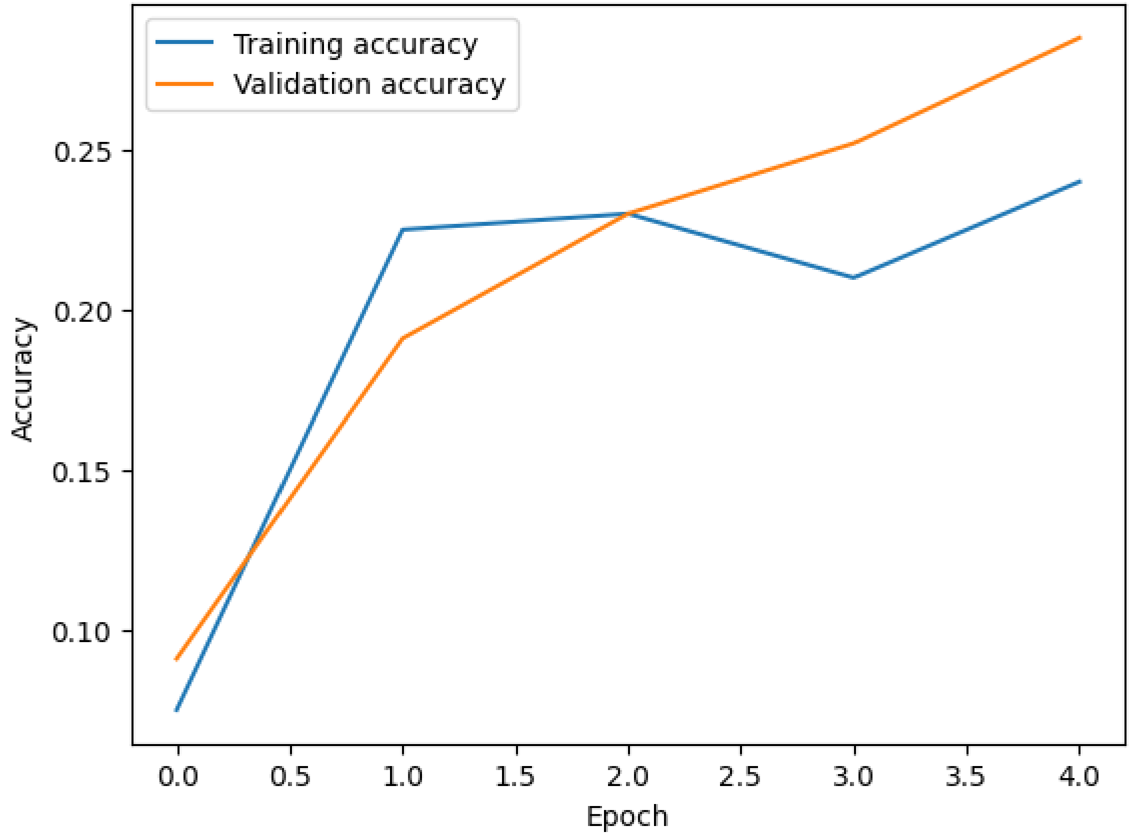
(2) How should you fix the problems you identified in (1)?

In [25]:
```
stats['train_acc_history']
```

Out[25]:
```
[0.075, 0.225, 0.23, 0.21, 0.24]
```

In [30]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Do some debugging to gain some insight into why the optimization
#   isn't great.
# ================================================================ #

# Plot the loss function and train / validation accuracies

x = np.arange(len(stats['train_acc_history']))
plt.plot(x, stats['train_acc_history'], label='Training accuracy')
plt.plot(x, stats['val_acc_history'], label='Validation accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

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

Out[30]:
```
<matplotlib.legend.Legend at 0x10bb0b9a0>
```



Answers:

(1) According to the plot of training accuracy and validation accuracy, the training accuracy remains very low (< 0.25 by the final epoch). This means that the model is very likely underfitting the training data and hasn't learned a particularly good model.

(2) The low training accuracy suggests that we can attempt to make the model more complex to better fit the training data by, for example, reducing the strength of regularization. Another possible solution would be to increase the number of training iterations.

Optimize the neural network

Use the following part of the Jupyter notebook to optimize your hyperparameters on the validation set. Store your nets as best_net.

In [46]:

```
best_net = None # store the best model into this

# ================================================================ #
# YOUR CODE HERE:
#   Optimize over your hyperparameters to arrive at the best neural
#   network.  You should be able to get over 50% validation accuracy.
#   For this part of the notebook, we will give credit based on the
#   accuracy you get.  Your score on this question will be multiplied by:
```

```
#      min(floor((X - 28%)) / %22, 1)
#   where if you get 50% or higher validation accuracy, you get full
#   points.
#
#   Note, you need to use the same network structure (keep hidden_size = 50)
# ================================================================= #

best_stats = None
best_val_acc = 0
for learning_rate in [1e-3, 1e-2, 1e-1]:
    for reg in [1e-7, 1e-5, 1e-3]:
        print(f'Testing lr={learning_rate}, reg={reg}...')
        net = TwoLayerNet(input_size, hidden_size, num_classes)
        stats = net.train(X_train, y_train, X_val, y_val,
            learning_rate=learning_rate,
            learning_rate_decay=0.95, reg=reg,
            num_iters=1500, batch_size=200, verbose=False)
        val_acc = (net.predict(X_val) == y_val).mean()
        if val_acc > best_val_acc:
            best_net = net
            best_stats = stats
            best_val_acc = val_acc

# ================================================================= #
# END YOUR CODE HERE
# ================================================================= #
val_acc = (best_net.predict(X_val) == y_val).mean()
print('Best validation accuracy: ', val_acc)
```

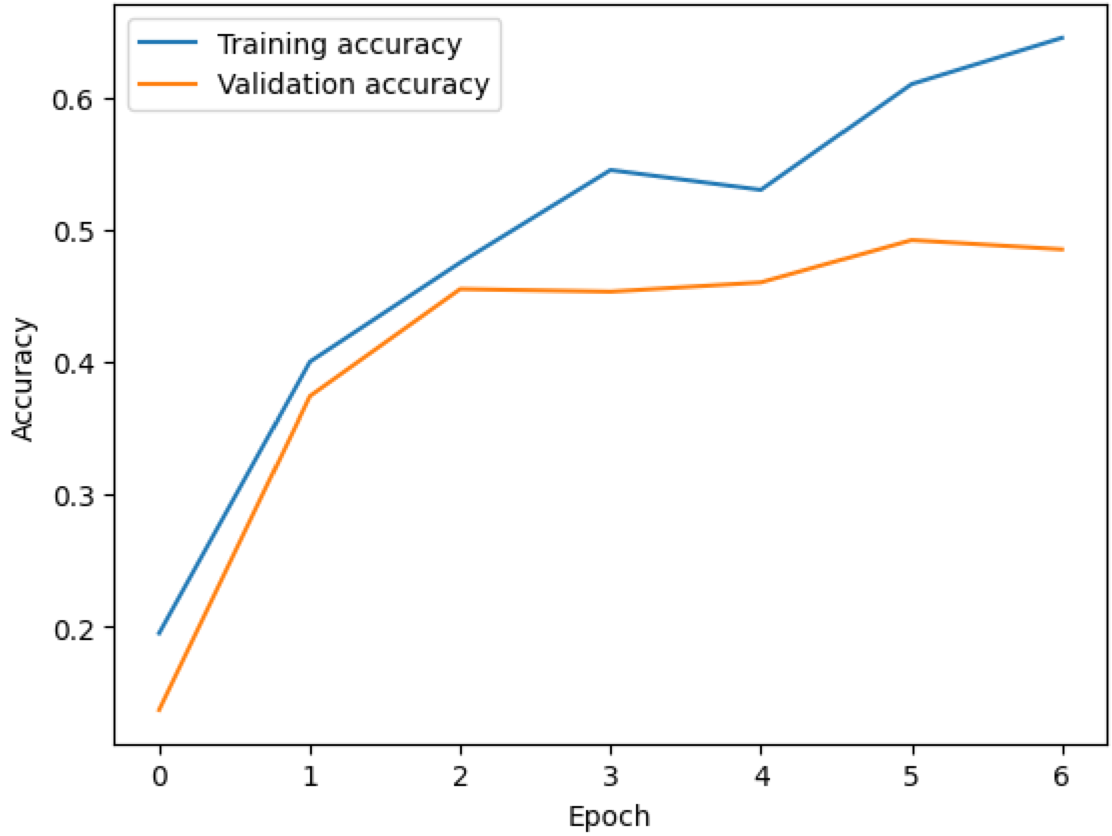
```
Testing lr=0.001, reg=1e-07...
Testing lr=0.001, reg=1e-05...
Testing lr=0.001, reg=0.001...
Testing lr=0.01, reg=1e-07...
Testing lr=0.01, reg=1e-05...
Testing lr=0.01, reg=0.001...
Testing lr=0.1, reg=1e-07...
Testing lr=0.1, reg=1e-05...
Testing lr=0.1, reg=0.001...
Best validation accuracy:  0.503
```

In [47]:

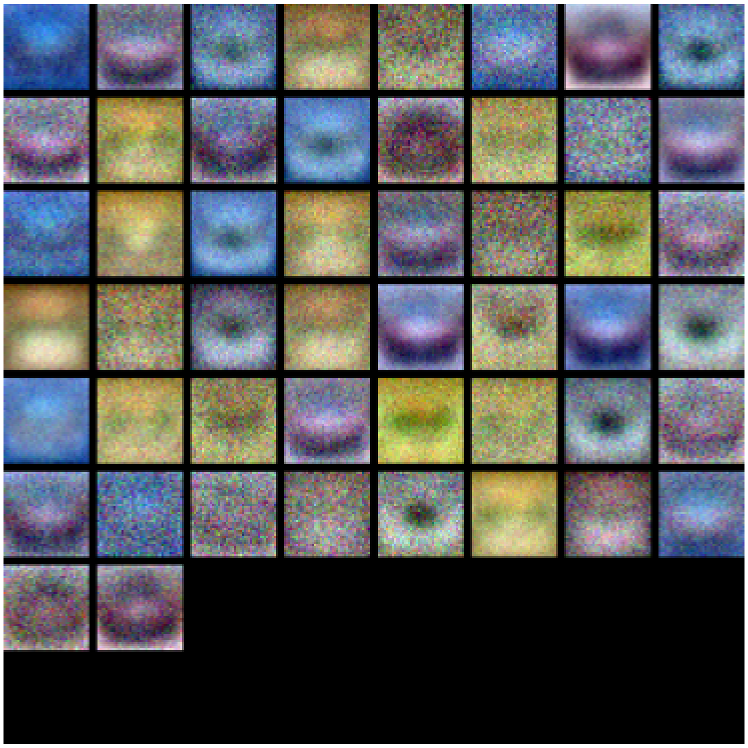
```
x = np.arange(len(best_stats['train_acc_history']))
plt.plot(x, best_stats['train_acc_history'], label='Training accuracy')
plt.plot(x, best_stats['val_acc_history'], label='Validation accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
```

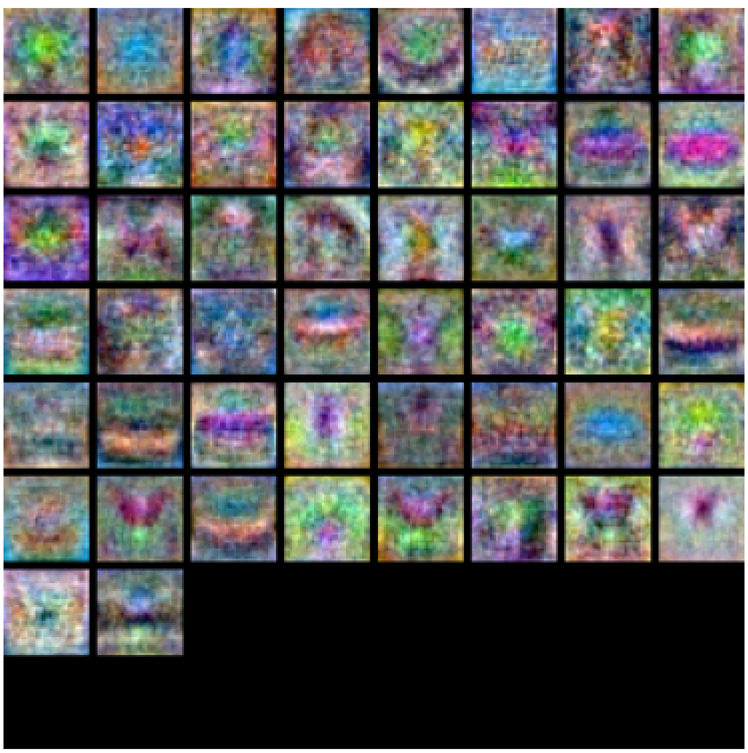
Out[47]:

```
<matplotlib.legend.Legend at 0x10f617580>
```



In [48]:
```
from utils.vis_utils import visualize_grid

# Visualize the weights of the network

def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.T.reshape(32, 32, 3, -1).transpose(3, 0, 1, 2)
    plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
    plt.gca().axis('off')
    plt.show()

show_net_weights(subopt_net)
show_net_weights(best_net)
```





Question:

(1) What differences do you see in the weights between the suboptimal net and the best net you arrived at?

Answer:

(1) The weights of the suboptimal net generally look very similar to one another, meaning that the model likely hasn't learned a good idea of what each class looks like. In comparison, the weights of the best net look more varied and defined.

Evaluate on test set

In [49]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.485
```

In [ ]:
```
```

Problem 5

layers.py:

```python
import numpy as np
import pdb

def affine_forward(x, w, b):
  """
  Computes the forward pass for an affine (fully-connected) layer.

  The input x has shape (N, d_1, ..., d_k) and contains a minibatch of N
  examples, where each example x[i] has shape (d_1, ..., d_k). We will
  reshape each input into a vector of dimension D = d_1 * ... * d_k, and
  then transform it to an output vector of dimension M.

  Inputs:
  - x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
  - w: A numpy array of weights, of shape (D, M)
  - b: A numpy array of biases, of shape (M,)

  Returns a tuple of:
  - out: output, of shape (N, M)
  - cache: (x, w, b)
  """

  # ================================================================ #
  # YOUR CODE HERE:
  #   Calculate the output of the forward pass.  Notice the dimensions
  #   of w are D x M, which is the transpose of what we did in earlier
  #   assignments.
  # ================================================================ #

  N, D = x.shape[0], w.shape[0]
  x_reshaped = np.reshape(x, (N, D))
  out = x_reshaped @ w + b

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

  cache = (x, w, b)
  return out, cache


def affine_backward(dout, cache):
  """
  Computes the backward pass for an affine layer.

  Inputs:
  - dout: Upstream derivative, of shape (N, M)
  - cache: Tuple of:
    - x: Input data, of shape (N, d_1, ... d_k)
    - w: Weights, of shape (D, M)
```

```python
    Returns a tuple of:
    - dx: Gradient with respect to x, of shape (N, d1, ..., d_k)
    - dw: Gradient with respect to w, of shape (D, M)
    - db: Gradient with respect to b, of shape (M,)
    """
    x, w, b = cache
    dx, dw, db = None, None, None

    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the gradients for the backward pass.
    # ================================================================ #

    # dout is N x M
    # dx should be N x d1 x ... x dk; it relates to dout through multiplication with w, which is D x M
    # dw should be D x M; it relates to dout through multiplication with x, which is N x D after reshapin
    # db should be M; it is just the sum over dout examples

    N, D = x.shape[0], w.shape[0]
    dx = np.reshape(dout @ w.T, x.shape)
    dw = np.reshape(x, (N, D)).T @ dout
    db = np.sum(dout, axis=0)

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

    return dx, dw, db

def relu_forward(x):
    """
    Computes the forward pass for a layer of rectified linear units (ReLUs).

    Input:
    - x: Inputs, of any shape

    Returns a tuple of:
    - out: Output, of the same shape as x
    - cache: x
    """
    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the ReLU forward pass.
    # ================================================================ #

    out = np.maximum(0, x)

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

    cache = x
    return out, cache
```

```python
def relu_backward(dout, cache):
  """
  Computes the backward pass for a layer of rectified linear units (ReLUs).

  Input:
  - dout: Upstream derivatives, of any shape
  - cache: Input x, of same shape as dout

  Returns:
  - dx: Gradient with respect to x
  """
  x = cache

  # ================================================================ #
  # YOUR CODE HERE:
  #   Implement the ReLU backward pass
  # ================================================================ #

  # ReLU directs linearly to those > 0
  dx = (x > 0) * dout

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

  return dx


def softmax_loss(x, y):
  """
  Computes the loss and gradient for softmax classification.

  Inputs:
  - x: Input data, of shape (N, C) where x[i, j] is the score for the jth class
    for the ith input.
  - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
    0 <= y[i] < C

  Returns a tuple of:
  - loss: Scalar giving the loss
  - dx: Gradient of the loss with respect to x
  """

  probs = np.exp(x - np.max(x, axis=1, keepdims=True))
  probs /= np.sum(probs, axis=1, keepdims=True)
  N = x.shape[0]
  loss = -np.sum(np.log(probs[np.arange(N), y])) / N
  dx = probs.copy()
  dx[np.arange(N), y] -= 1
  dx /= N
  return loss, dx
```

fc_net.py:

```
import numpy as np

from .layers import *
from .layer_utils import *


class TwoLayerNet(object):
  """
  A two-layer fully-connected neural network with ReLU nonlinearity and
  softmax loss that uses a modular layer design. We assume an input dimension
  of D, a hidden dimension of H, and perform classification over C classes.

  The architecure should be affine - relu - affine - softmax.

  Note that this class does not implement gradient descent; instead, it
  will interact with a separate Solver object that is responsible for running
  optimization.

  The learnable parameters of the model are stored in the dictionary
  self.params that maps parameter names to numpy arrays.
  """

  def __init__(self, input_dim=3*32*32, hidden_dims=100, num_classes=10,
               dropout=0, weight_scale=1e-3, reg=0.0):
    """
    Initialize a new network.

    Inputs:
    - input_dim: An integer giving the size of the input
    - hidden_dims: An integer giving the size of the hidden layer
    - num_classes: An integer giving the number of classes to classify
    - dropout: Scalar between 0 and 1 giving dropout strength.
    - weight_scale: Scalar giving the standard deviation for random
      initialization of the weights.
    - reg: Scalar giving L2 regularization strength.
    """
    self.params = {}
    self.reg = reg

    # ================================================================ #
    # YOUR CODE HERE:
    #   Initialize W1, W2, b1, and b2.  Store these as self.params['W1'],
    #   self.params['W2'], self.params['b1'] and self.params['b2']. The
    #   biases are initialized to zero and the weights are initialized
    #   so that each parameter has mean 0 and standard deviation weight_scale.
    #   The dimensions of W1 should be (input_dim, hidden_dim) and the
    #   dimensions of W2 should be (hidden_dims, num_classes)
    # ================================================================ #

    self.params['W1'] = np.random.normal(loc=0, scale=weight_scale, size=(input_dim, hidden_dims))
    self.params['b1'] = np.zeros(hidden_dims)
    self.params['W2'] = np.random.normal(loc=0, scale=weight_scale, size=(hidden_dims, num_classes))
```

```
    self.params['b2'] = np.zeros(num_classes)

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

  def loss(self, X, y=None):
    """
    Compute loss and gradient for a minibatch of data.

    Inputs:
    - X: Array of input data of shape (N, d_1, ..., d_k)
    - y: Array of labels, of shape (N,). y[i] gives the label for X[i].

    Returns:
    If y is None, then run a test-time forward pass of the model and return:
    - scores: Array of shape (N, C) giving classification scores, where
      scores[i, c] is the classification score for X[i] and class c.

    If y is not None, then run a training-time forward and backward pass and
    return a tuple of:
    - loss: Scalar value giving the loss
    - grads: Dictionary with the same keys as self.params, mapping parameter
      names to gradients of the loss with respect to those parameters.
    """
    scores = None

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the forward pass of the two-layer neural network. Store
    #   the class scores as the variable 'scores'.  Be sure to use the layers
    #   you prior implemented.
    # ================================================================ #

    layer_1_out, layer_1_cache = affine_relu_forward(X, self.params['W1'], self.params['b1'])
    layer_2_out, layer_2_cache = affine_forward(layer_1_out, self.params['W2'], self.params['b2'])
    scores = layer_2_out

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

    # If y is None then we are in test mode so just return scores
    if y is None:
      return scores

    loss, grads = 0, {}
    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the backward pass of the two-layer neural net.  Store
    #   the loss as the variable 'loss' and store the gradients in the
    #   'grads' dictionary.  For the grads dictionary, grads['W1'] holds
    #   the gradient for W1, grads['b1'] holds the gradient for b1, etc.
    #   i.e., grads[k] holds the gradient for self.params[k].
```

```python
    #
    #   Add L2 regularization, where there is an added cost 0.5*self.reg*W^2
    #   for each W.  Be sure to include the 0.5 multiplying factor to
    #   match our implementation.
    #
    #   And be sure to use the layers you prior implemented.
    # ================================================================ #

    loss, dL = softmax_loss(scores, y)
    loss += 0.5 * self.reg * (np.sum(self.params['W1']**2) +  np.sum(self.params['W2']**2))

    dh, dw2, db2 = affine_backward(dL, layer_2_cache)
    _, dw1, db1 = affine_relu_backward(dh, layer_1_cache)

    grads['W1'] = dw1 + self.reg * self.params['W1']
    grads['b1'] = db1.T
    grads['W2'] = dw2 + self.reg * self.params['W2']
    grads['b2'] = db2.T

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

    return loss, grads


class FullyConnectedNet(object):
  """
  A fully-connected neural network with an arbitrary number of hidden layers,
  ReLU nonlinearities, and a softmax loss function. This will also implement
  dropout and batch normalization as options. For a network with L layers,
  the architecture will be

  {affine - [batch norm] - relu - [dropout]} x (L - 1) - affine - softmax

  where batch normalization and dropout are optional, and the {...} block is
  repeated L - 1 times.

  Similar to the TwoLayerNet above, learnable parameters are stored in the
  self.params dictionary and will be learned using the Solver class.
  """

  def __init__(self, hidden_dims, input_dim=3*32*32, num_classes=10,
               dropout=0, use_batchnorm=False, reg=0.0,
               weight_scale=1e-2, dtype=np.float32, seed=None):
    """
    Initialize a new FullyConnectedNet.

    Inputs:
    - hidden_dims: A list of integers giving the size of each hidden layer.
    - input_dim: An integer giving the size of the input.
    - num_classes: An integer giving the number of classes to classify.
    - dropout: Scalar between 0 and 1 giving dropout strength. If dropout=0 then
      the network should not use dropout at all.
```

```
- use_batchnorm: Whether or not the network should use batch normalization.
- reg: Scalar giving L2 regularization strength.
- weight_scale: Scalar giving the standard deviation for random
  initialization of the weights.
- dtype: A numpy datatype object; all computations will be performed using
  this datatype. float32 is faster but less accurate, so you should use
  float64 for numeric gradient checking.
- seed: If not None, then pass this random seed to the dropout layers. This
  will make the dropout layers deteriminstic so we can gradient check the
  model.
"""
self.use_batchnorm = use_batchnorm
self.use_dropout = dropout > 0
self.reg = reg
self.num_layers = 1 + len(hidden_dims)
self.dtype = dtype
self.params = {}

# ================================================================ #
# YOUR CODE HERE:
#   Initialize all parameters of the network in the self.params dictionary.
#   The weights and biases of layer 1 are W1 and b1; and in general the
#   weights and biases of layer i are Wi and bi. The
#   biases are initialized to zero and the weights are initialized
#   so that each parameter has mean 0 and standard deviation weight_scale.
# ================================================================ #

in_dim = input_dim
for i, out_dim in enumerate(hidden_dims + [num_classes]):
  self.params[f'W{i+1}'] = np.random.normal(loc=0, scale=weight_scale, size=(in_dim, out_dim))
  self.params[f'b{i+1}'] = np.zeros((out_dim,))
  in_dim = out_dim

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

# When using dropout we need to pass a dropout_param dictionary to each
# dropout layer so that the layer knows the dropout probability and the mode
# (train / test). You can pass the same dropout_param to each dropout layer.
self.dropout_param = {}
if self.use_dropout:
  self.dropout_param = {'mode': 'train', 'p': dropout}
  if seed is not None:
    self.dropout_param['seed'] = seed

# With batch normalization we need to keep track of running means and
# variances, so we need to pass a special bn_param object to each batch
# normalization layer. You should pass self.bn_params[0] to the forward pass
# of the first batch normalization layer, self.bn_params[1] to the forward
# pass of the second batch normalization layer, etc.
self.bn_params = []
if self.use_batchnorm:
  self.bn_params = [{'mode': 'train'} for i in np.arange(self.num_layers - 1)]
```

```python
    # Cast all parameters to the correct datatype
    for k, v in self.params.items():
      self.params[k] = v.astype(dtype)


  def loss(self, X, y=None):
    """
    Compute loss and gradient for the fully-connected net.

    Input / output: Same as TwoLayerNet above.
    """
    X = X.astype(self.dtype)
    mode = 'test' if y is None else 'train'

    # Set train/test mode for batchnorm params and dropout param since they
    # behave differently during training and testing.
    if self.dropout_param is not None:
      self.dropout_param['mode'] = mode
    if self.use_batchnorm:
      for bn_param in self.bn_params:
        bn_param[mode] = mode

    scores = None

    # ================================================================ #
    # YOUR CODE HERE:
    #   Implement the forward pass of the FC net and store the output
    #   scores as the variable "scores".
    # ================================================================ #

    L = self.num_layers
    in_layer = X
    caches = dict()

    for i in range(1, L):
      W, b = self.params[f'W{i}'], self.params[f'b{i}']
      out_layer, layer_cache = affine_relu_forward(in_layer, W, b)
      caches[i] = layer_cache
      in_layer = out_layer

    scores, layer_cache = affine_forward(in_layer, self.params[f'W{L}'], self.params[f'b{L}'])
    caches[L] = layer_cache

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

    # If test mode return early
    if mode == 'test':
      return scores

    loss, grads = 0.0, {}
    # ================================================================ #
```

```python
# YOUR CODE HERE:
#   Implement the backwards pass of the FC net and store the gradients
#   in the grads dict, so that grads[k] is the gradient of self.params[k]
#   Be sure your L2 regularization includes a 0.5 factor.
# ================================================================ #

loss, dL = softmax_loss(scores, y)
for p in self.params:
  if p[0] == 'W':
    loss += 0.5 * self.reg * np.sum(self.params[p]**2)

backward = affine_backward
dhi, dwi, dbi = dL, None, None
for i in range(L, 0, -1):
  dhi, dwi, dbi = backward(dhi, caches[i])
  grads[f'W{i}'] = dwi + self.reg * self.params[f'W{i}']
  grads[f'b{i}'] = dbi.T
  backward = affine_relu_backward

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
return loss, grads
```

Fully connected networks

In the previous notebook, you implemented a simple two-layer neural network class. However, this class is not modular. If you wanted to change the number of layers, you would need to write a new loss and gradient function. If you wanted to optimize the network with different optimizers, you'd need to write new training functions. If you wanted to incorporate regularizations, you'd have to modify the loss and gradient function.

Instead of having to modify functions each time, for the rest of the class, we'll work in a more modular framework where we define forward and backward layers that calculate losses and gradients respectively. Since the forward and backward layers share intermediate values that are useful for calculating both the loss and the gradient, we'll also have these function return "caches" which store useful intermediate values.

The goal is that through this modular design, we can build different sized neural networks for various applications.

In this HW #3, we'll define the basic architecture, and in HW #4, we'll build on this framework to implement different optimizers and regularizations (like BatchNorm and Dropout).

Modular layers

This notebook will build modular layers in the following manner. First, there will be a forward pass for a given layer with inputs (`x`) and return the output of that layer (`out`) as well as cached variables (`cache`) that will be used to calculate the gradient in the backward pass.

```
def layer_forward(x, w):
  """ Receive inputs x and weights w """
  # Do some computations ...
  z = # ... some intermediate value
  # Do some more computations ...
  out = # the output

  cache = (x, w, z, out) # Values we need to compute gradients

  return out, cache
```

The backward pass will receive upstream derivatives and the `cache` object, and will return gradients with respect to the inputs and weights, like this:

```
def layer_backward(dout, cache):
  """
```

```
  Receive derivative of loss with respect to outputs and cache,
  and compute derivative with respect to inputs.
  """
  # Unpack cache values
  x, w, z, out = cache

  # Use values in cache to compute derivatives
  dx = # Derivative of loss with respect to x
  dw = # Derivative of loss with respect to w

  return dx, dw
```

In [1]:
```
## Import and setups

import time
import numpy as np
import matplotlib.pyplot as plt
from nndl.fc_net import *
from utils.data_utils import get_CIFAR10_data
from utils.gradient_check import eval_numerical_gradient, eval_numerical_gra
from utils.solver import Solver

%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'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ip
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
  """ returns relative error """
  return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

In [2]:
```
# Load the (preprocessed) CIFAR10 data.
data = get_CIFAR10_data()
for k in data.keys():
  print('{}: {} '.format(k, data[k].shape))
```

```
X_train: (49000, 3, 32, 32)
y_train: (49000,)
X_val: (1000, 3, 32, 32)
y_val: (1000,)
X_test: (1000, 3, 32, 32)
y_test: (1000,)
```

Linear layers

In this section, we'll implement the forward and backward pass for the linear layers.

The linear layer forward pass is the function `affine_forward` in `nndl/layers.py` and the backward pass is `affine_backward`.

After you have implemented these, test your implementation by running the cell below.

Affine layer forward pass

Implement `affine_forward` and then test your code by running the following cell.

In [3]:
```
# Test the affine_forward function

num_inputs = 2
input_shape = (4, 5, 6)
output_dim = 3

input_size = num_inputs * np.prod(input_shape)
weight_size = output_dim * np.prod(input_shape)

x = np.linspace(-0.1, 0.5, num=input_size).reshape(num_inputs, *input_shape)
w = np.linspace(-0.2, 0.3, num=weight_size).reshape(np.prod(input_shape), ou
b = np.linspace(-0.3, 0.1, num=output_dim)

out, _ = affine_forward(x, w, b)
correct_out = np.array([[ 1.49834967,  1.70660132,  1.91485297],
                        [ 3.25553199,  3.5141327,   3.77273342]])

# Compare your output with ours. The error should be around 1e-9.
print('Testing affine_forward function:')
print('difference: {}'.format(rel_error(out, correct_out)))
```

```
Testing affine_forward function:
difference: 9.7698500479884e-10
```

Affine layer backward pass

Implement `affine_backward` and then test your code by running the following cell.

In [4]:
```
# Test the affine_backward function

x = np.random.randn(10, 2, 3)
w = np.random.randn(6, 5)
b = np.random.randn(5)
dout = np.random.randn(10, 5)

dx_num = eval_numerical_gradient_array(lambda x: affine_forward(x, w, b)[0],
dw_num = eval_numerical_gradient_array(lambda w: affine_forward(x, w, b)[0],
db_num = eval_numerical_gradient_array(lambda b: affine_forward(x, w, b)[0],

_, cache = affine_forward(x, w, b)
dx, dw, db = affine_backward(dout, cache)

# The error should be around 1e-10
print('Testing affine_backward function:')
```

```
print('dx error: {}'.format(rel_error(dx_num, dx)))
print('dw error: {}'.format(rel_error(dw_num, dw)))
print('db error: {}'.format(rel_error(db_num, db)))
```

```
Testing affine_backward function:
dx error: 7.31465044978936e-10
dw error: 9.088876066260483e-11
db error: 9.55575431106778e-12
```

Activation layers

In this section you'll implement the ReLU activation.

ReLU forward pass

Implement the `relu_forward` function in `nndl/layers.py` and then test your code by running the following cell.

In [5]:
```
# Test the relu_forward function

x = np.linspace(-0.5, 0.5, num=12).reshape(3, 4)

out, _ = relu_forward(x)
correct_out = np.array([[ 0.,          0.,          0.,          0.,
                        [ 0.,          0.,          0.04545455,  0.13636364,
                        [ 0.22727273,  0.31818182,  0.40909091,  0.5,

# Compare your output with ours. The error should be around 1e-8
print('Testing relu_forward function:')
print('difference: {}'.format(rel_error(out, correct_out)))
```

```
Testing relu_forward function:
difference: 4.999999798022158e-08
```

ReLU backward pass

Implement the `relu_backward` function in `nndl/layers.py` and then test your code by running the following cell.

In [6]:
```
x = np.random.randn(10, 10)
dout = np.random.randn(*x.shape)

dx_num = eval_numerical_gradient_array(lambda x: relu_forward(x)[0], x, dout

_, cache = relu_forward(x)
dx = relu_backward(dout, cache)

# The error should be around 1e-12
print('Testing relu_backward function:')
print('dx error: {}'.format(rel_error(dx_num, dx)))
```

```
Testing relu_backward function:
dx error: 3.2756218628675432e-12
```

Combining the affine and ReLU layers

Often times, an affine layer will be followed by a ReLU layer. So let's make one that puts them together. Layers that are combined are stored in `nndl/layer_utils.py`.

Affine-ReLU layers

We've implemented `affine_relu_forward()` and `affine_relu_backward` in `nndl/layer_utils.py`. Take a look at them to make sure you understand what's going on. Then run the following cell to ensure its implemented correctly.

In [7]:
```
from nndl.layer_utils import affine_relu_forward, affine_relu_backward

x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

out, cache = affine_relu_forward(x, w, b)
dx, dw, db = affine_relu_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: affine_relu_forward(x, w, b
dw_num = eval_numerical_gradient_array(lambda w: affine_relu_forward(x, w, b
db_num = eval_numerical_gradient_array(lambda b: affine_relu_forward(x, w, b

print('Testing affine_relu_forward and affine_relu_backward:')
print('dx error: {}'.format(rel_error(dx_num, dx)))
print('dw error: {}'.format(rel_error(dw_num, dw)))
print('db error: {}'.format(rel_error(db_num, db)))
```

```
Testing affine_relu_forward and affine_relu_backward:
dx error: 1.4005251456258178e-10
dw error: 4.764910208249058e-10
db error: 7.826645559542029e-12
```

Softmax loss

You've already implemented it, so we have written it in `layers.py`. The following code will ensure they are working correctly.

In [8]:
```
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)



dx_num = eval_numerical_gradient(lambda x: softmax_loss(x, y)[0], x, verbose
loss, dx = softmax_loss(x, y)
```

```
# Test softmax_loss function. Loss should be 2.3 and dx error should be 1e-8
print('\nTesting softmax_loss:')
print('loss: {}'.format(loss))
print('dx error: {}'.format(rel_error(dx_num, dx)))
```

```
Testing softmax_loss:
loss: 2.3025779226361585
dx error: 6.197432934655467e-09
```

Implementation of a two-layer NN

In `nndl/fc_net.py`, implement the class `TwoLayerNet` which uses the layers you made here. When you have finished, the following cell will test your implementation.

In [9]:

```
N, D, H, C = 3, 5, 50, 7
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

std = 1e-2
model = TwoLayerNet(input_dim=D, hidden_dims=H, num_classes=C, weight_scale=

print('Testing initialization ... ')
W1_std = abs(model.params['W1'].std() - std)
b1 = model.params['b1']
W2_std = abs(model.params['W2'].std() - std)
b2 = model.params['b2']
assert W1_std < std / 10, 'First layer weights do not seem right'
assert np.all(b1 == 0), 'First layer biases do not seem right'
assert W2_std < std / 10, 'Second layer weights do not seem right'
assert np.all(b2 == 0), 'Second layer biases do not seem right'

print('Testing test-time forward pass ... ')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.loss(X)
correct_scores = np.asarray(
  [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434, 15.
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'

print('Testing training loss (no regularization)')
y = np.asarray([0, 5, 1])
loss, grads = model.loss(X, y)
correct_loss = 3.4702243556
assert abs(loss - correct_loss) < 1e-10, 'Problem with training-time loss'

model.reg = 1.0
loss, grads = model.loss(X, y)
```

```
correct_loss = 26.5948426952
assert abs(loss - correct_loss) < 1e-10, 'Problem with regularization loss'

for reg in [0.0, 0.7]:
  print('Running numeric gradient check with reg = {}'.format(reg))
  model.reg = reg
  loss, grads = model.loss(X, y)

  for name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False)
    print('{} relative error: {}'.format(name, rel_error(grad_num, grads[nam
```

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg = 0.0
W1 relative error: 1.8265914102168292e-08
W2 relative error: 3.3458521609206087e-10
b1 relative error: 8.008665234879004e-09
b2 relative error: 1.9376073688240078e-10
Running numeric gradient check with reg = 0.7
W1 relative error: 2.5279152310200606e-07
W2 relative error: 1.3678364482660862e-07
b1 relative error: 1.564680139421095e-08
b2 relative error: 9.089616810648186e-10
```

Solver

We will now use the utils Solver class to train these networks. Familiarize yourself with the API in `utils/solver.py`. After you have done so, declare an instance of a TwoLayerNet with 200 units and then train it with the Solver. Choose parameters so that your validation accuracy is at least 50%.

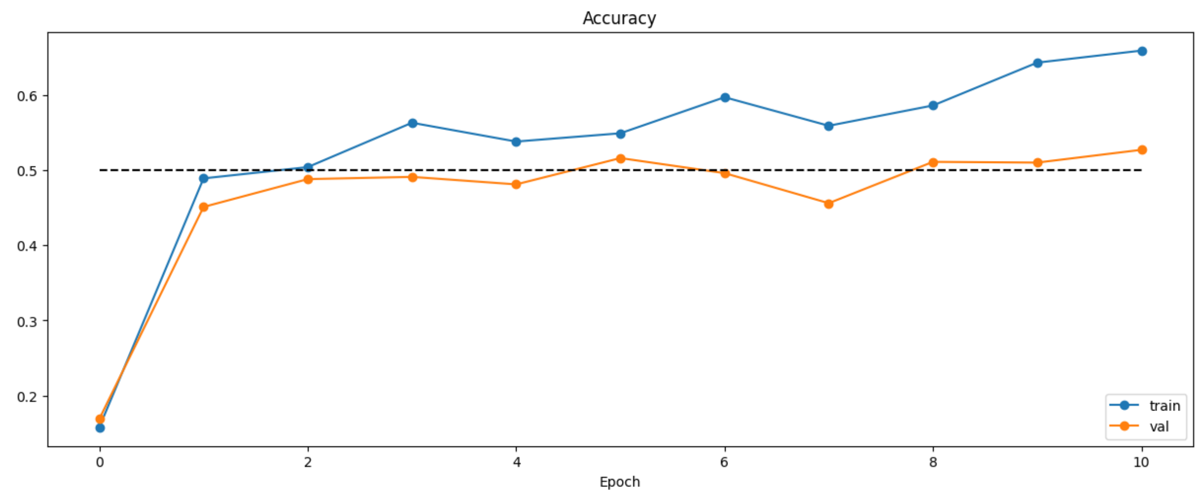
In [10]:
```
model = TwoLayerNet()
solver = None

# ================================================================ #
# YOUR CODE HERE:
#   Declare an instance of a TwoLayerNet and then train
#   it with the Solver. Choose hyperparameters so that your validation
#   accuracy is at least 50%.  We won't have you optimize this further
#   since you did it in the previous notebook.
#
# ================================================================ #

model = TwoLayerNet(hidden_dims=200, reg=1e-4)
solver = Solver(model, data,
                update_rule='sgd',
                optim_config={
                  'learning_rate': 1e-3,
                },
                lr_decay=0.97,
                num_epochs=10,
```

```
                batch_size=100,
                print_every=500)
solver.train()

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

```
(Iteration 1 / 4900) loss: 2.301530
(Epoch 0 / 10) train acc: 0.158000; val_acc: 0.169000
(Epoch 1 / 10) train acc: 0.489000; val_acc: 0.451000
(Iteration 501 / 4900) loss: 1.651384
(Epoch 2 / 10) train acc: 0.504000; val_acc: 0.488000
(Iteration 1001 / 4900) loss: 1.340158
(Epoch 3 / 10) train acc: 0.563000; val_acc: 0.491000
(Iteration 1501 / 4900) loss: 1.617317
(Epoch 4 / 10) train acc: 0.538000; val_acc: 0.481000
(Iteration 2001 / 4900) loss: 1.159907
(Epoch 5 / 10) train acc: 0.549000; val_acc: 0.516000
(Iteration 2501 / 4900) loss: 1.099285
(Epoch 6 / 10) train acc: 0.597000; val_acc: 0.496000
(Iteration 3001 / 4900) loss: 0.999669
(Epoch 7 / 10) train acc: 0.559000; val_acc: 0.456000
(Iteration 3501 / 4900) loss: 1.196356
(Epoch 8 / 10) train acc: 0.586000; val_acc: 0.511000
(Iteration 4001 / 4900) loss: 1.138254
(Epoch 9 / 10) train acc: 0.643000; val_acc: 0.510000
(Iteration 4501 / 4900) loss: 0.855623
(Epoch 10 / 10) train acc: 0.659000; val_acc: 0.527000
```

In [11]:

```
# Run this cell to visualize training loss and train / val accuracy

plt.subplot(2, 1, 1)
plt.title('Training loss')
plt.plot(solver.loss_history, 'o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(solver.train_acc_history, '-o', label='train')
plt.plot(solver.val_acc_history, '-o', label='val')
plt.plot([0.5] * len(solver.val_acc_history), 'k--')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```



Multilayer Neural Network

Now, we implement a multi-layer neural network.

Read through the `FullyConnectedNet` class in the file `nndl/fc_net.py`.

Implement the initialization, the forward pass, and the backward pass. There will be lines for batchnorm and dropout layers and caches; ignore these all for now. That'll be in HW #4.

In [12]:
```
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

for reg in [0, 3.14]:
  print('Running check with reg = {}'.format(reg))
  model = FullyConnectedNet([H1, H2], input_dim=D, num_classes=C,
                            reg=reg, weight_scale=5e-2, dtype=np.float64)

  loss, grads = model.loss(X, y)
  print('Initial loss: {}'.format(loss))

  for name in sorted(grads):
```

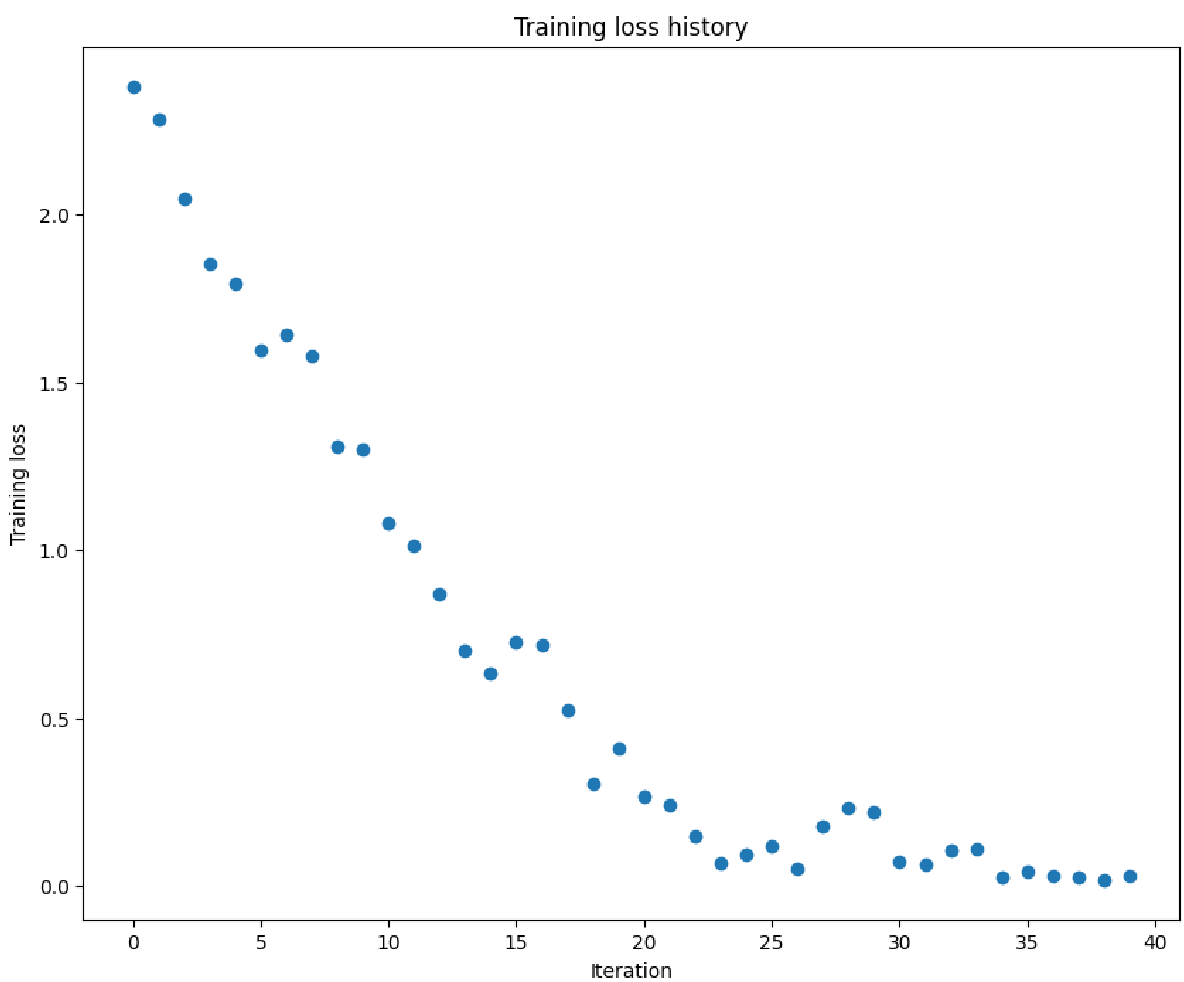
```
    f = lambda _: model.loss(X, y)[0]
    grad_num = eval_numerical_gradient(f, model.params[name], verbose=False,
    print('{} relative error: {}'.format(name, rel_error(grad_num, grads[nam
```

```
Running check with reg = 0
Initial loss: 2.301039294372159
W1 relative error: 1.7669291193587022e-06
W2 relative error: 9.571653999079532e-06
W3 relative error: 3.813095605592845e-07
b1 relative error: 5.8842141120884016e-08
b2 relative error: 4.8381987778526565e-09
b3 relative error: 1.0354616622811734e-10
Running check with reg = 3.14
Initial loss: 6.890666355348374
W1 relative error: 3.522896396863251e-08
W2 relative error: 2.8962458969078816e-08
W3 relative error: 1.5715536797469338e-08
b1 relative error: 1.003128487146125e-07
b2 relative error: 9.329412999788734e-09
b3 relative error: 4.1517298487597025e-10
```

In [13]:

```
# Use the three layer neural network to overfit a small dataset.

num_train = 50
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}


#### !!!!!!
# Play around with the weight_scale and learning_rate so that you can overfi
# Your training accuracy should be 1.0 to receive full credit on this part.
weight_scale = 1e-2
learning_rate = 1e-2

model = FullyConnectedNet([100, 100],
              weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                  'learning_rate': learning_rate,
                }
         )
solver.train()

plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

```
(Iteration 1 / 40) loss: 2.380206
(Epoch 0 / 20) train acc: 0.240000; val_acc: 0.095000
(Epoch 1 / 20) train acc: 0.220000; val_acc: 0.138000
(Epoch 2 / 20) train acc: 0.320000; val_acc: 0.150000
(Epoch 3 / 20) train acc: 0.380000; val_acc: 0.134000
(Epoch 4 / 20) train acc: 0.620000; val_acc: 0.195000
(Epoch 5 / 20) train acc: 0.740000; val_acc: 0.204000
(Iteration 11 / 40) loss: 1.081375
(Epoch 6 / 20) train acc: 0.800000; val_acc: 0.160000
(Epoch 7 / 20) train acc: 0.740000; val_acc: 0.174000
(Epoch 8 / 20) train acc: 0.880000; val_acc: 0.192000
(Epoch 9 / 20) train acc: 0.960000; val_acc: 0.196000
(Epoch 10 / 20) train acc: 0.960000; val_acc: 0.207000
(Iteration 21 / 40) loss: 0.266158
(Epoch 11 / 20) train acc: 0.980000; val_acc: 0.201000
(Epoch 12 / 20) train acc: 0.960000; val_acc: 0.194000
(Epoch 13 / 20) train acc: 0.960000; val_acc: 0.217000
(Epoch 14 / 20) train acc: 0.920000; val_acc: 0.170000
(Epoch 15 / 20) train acc: 0.980000; val_acc: 0.197000
(Iteration 31 / 40) loss: 0.075206
(Epoch 16 / 20) train acc: 0.980000; val_acc: 0.190000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.197000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.189000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.187000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.188000
```



```
In [ ]:
```