2-layer Neural Network

2317038

허지원

개요

2-layer neural network는 2-layer fully-connected neural network을 구현하고, 이를 CIFAR-10 데이터 세트에 대한 분류에 적용하는 프로젝트이다. 하이퍼 파라미터 튜닝과 L2 정규화, SGD, Backpropagation 등의 기법을 적용하여 높은 정확도를 구현하였다.

2-layer neural network

: input layer → hidden layer → output layer의 구조를 갖는 신경망

- first Layer: input layer → hidden layer (W1, b1)
- second Layer: hidden layer → output layer (W2, b2)

Loss function

: 모델이 예측한 값과 실제 정답(레이블) 사이의 차이를 수치적으로 나타내는 함수로, 우리의 프로젝트에서는 softmax를 통해 확률로 변환한 뒤 cross-entropy 계산을 통해 확률과 레이블을 비교한다.

① softmax: 클래스 점수(score) 값을 확률(probability)로 변환하여 나타내는 함수

$$P(Y = k|X = x_i) = p_k = \frac{e^{s_k}}{\sum_{j=1}^{C} e^{s_j}}$$

- x_i: image, y_i: class label, $s = Wx_i + b$

② cross-entropy loss: 예측 확률과 실제 레이블의 차이를 계산하여 오차를 측정

$$L_i = -\sum_{j=1}^{C} \left(z_{ij} \log p_i + (1 - z_{ij}) \log(1 - p_i) \right)$$

- z_{ij}: i번째 이미지의 클래스 레이블
- p_i: i번째 이미지의 확률

③ softmax + cross-entropy의 미분 결과

1) softmax + cross-entropy를 통한 손실값

$$L = -\frac{1}{N} \log\left(p_{i,y_i}\right)$$

2) softmax + cross-entropy의 미분 결과

$$\frac{\partial L}{\partial z_{i,j}} = \frac{1}{N}\left(p_{i,j} - 1[y_i = j]\right)$$

- 정답 클래스에서는 $(p_i - 1)$이고, 나머지 클래스에서는 $(p_i - 0)$

L2 regularizaiton

: 손실 함수에서 모델의 가중치의 제곱합을 추가하는 방식의 정규화 기법

$$L = \frac{1}{N}\sum_{i=1}^{M} L_i(f(x_i, W), y_i) + \lambda R(W), \qquad R(W) = \sum_{k,l} W_{k,l}^2$$

Stochastic Gradient Descent(SGD)

: 전체 데이터 세트 내에서 무작위로 선택한 하나의 샘플에 대해 손실 함수의 기울기를 계산하고, 그 반대 방향으로 파라미터를 이동시키며 손실을 줄여 나가는 확률적 경사 하강법

$$W^{T+1} = W^T - \alpha\frac{\partial L}{\partial W^T}$$

- 대용량의 데이터 세트가 아니라 작은 샘플을 사용하여 근사합을 구하기 때문에 빠른 학습이 가능하다.

Backpropagation

: 학습의 결과값과 정답값 사이의 오차를 output layer→input layer 방향으로 propagation 하면서 각 가중치를 조정하는 방법. Chain Rule 개념을 활용한다.

코드 설명

1. neural_net.py

1) 라이브러리 불러오기

```
1 from __future__ import print_function
2
3 import numpy as np
4 import matplotlib.pyplot as plt
```

A. numpy: 선형대수 기반의 계산을 위한 라이브러리

B. matplotlib.pyplot: 시각화를 위한 라이브러리

2) 클래스 정의 및 초기화 함수

```python
class TwoLayerNet(object):
  """
  A two-layer fully-connected neural network. The net has an input dimension of
  N, 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.
  """

```

A. TwoLayerNet 클래스 정의: 두 개의 계층으로 구성된 신경망 클래스를 정의

i. 가중치 행렬에서 softmax loss function과 L2 정규화를 활용하여 network를 학습

ii. input layer → fully connected layer → ReLU → fully connected layer → softmax의 과정

```python
  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 (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.
      """

      # np.random.randn(shape)
      # - Return a sample (or samples) from the "standard normal" distribution following shape
      self.params = {}
      self.params['W1'] = std * np.random.randn(input_size, hidden_size)
      self.params['b1'] = np.zeros(hidden_size)
      self.params['W2'] = std * np.random.randn(hidden_size, output_size)
      self.params['b2'] = np.zeros(output_size)
```

B. __init__ 함수: 모델 파라미터를 초기화 하기 위한 함수

i. 파라미터

- input_size: 입력 데이터의 차원(D)
- hidden_size: hidden layer의 뉴런의 개수(H)
- output_size: 클래스의 개수(C)
- std: 가중치 초기화 표준편차

ii. self.params 딕셔너리 정의

- W1: input layer → hidden layer의 가중치 (D, H)
- b1: hidden layer의 편향 (H,)
- W2: hidden layer → output layer의 가중치 (H, C)
- b2: output layer의 편향 (C,)

3) 손실(loss) 함수

```python
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
      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
```

A. 함수 파라미터
   i. X: 입력 데이터 (N, D)
   ii. y: 정답 레이블 (N,)
   iii. reg: 정규화 강도 (이 코드에서는 L2 정규화 계수)
B. W1, b1, W2, b2: 계층별 가중치, 편향 값

```python
    # Compute the forward pass
    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 array of      #
    # shape (N, C).                                                             #
    ##############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    z = X.dot(W1) + b1 # (N, H)
    h = np.maximum(z, 0) # ReLu
    scores = h.dot(W2) + b2 # (N, C)


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

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

C. Forward pass 구현: 입력 데이터에 대해 각 클래스의 점수를 계산
   i. z: 첫 번째 계층 (hidden layer의 입력을 계산)
   ii. h: ReLU 활성화 함수를 적용하여 hidden layer 출력
   iii. scores: 최종 클래스별 점수 (N, C)
D. 레이블이 없으면 예측 점수(scores)만 반환

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

    scores_exp = np.exp(scores)
    scores_expsum = np.sum(scores_exp, axis=1).reshape(N,1)

    p = scores_exp / scores_expsum # softmax probabilities

    loss = np.sum(-np.log(p[np.arange(N), y]))/N # cross-entropy loss

    # L2 Regularization
    loss += 0.5 * reg * (np.sum(W1**2) + np.sum(W2**2))

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

E. 손실 계산 (softmax + L2 Regularizaiton)

i. Data loss 계산: softmax를 사용하여 data loss 계산

ii. Regularization 계산: L2 regularization을 사용하여 정규화

iii. loss는 스칼라 값

```
    # 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 matrix of same size #
    ############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    # Let z be the result after W * X, z after activation function becomes y.

    # back propagation
    b_p = np.copy(p) # (N, C)
    b_p[np.arange(N), y] -= 1 # 정답 클래스 확률에 -1
    b_p /= N # 샘플 수로 평균

    grads['W2'] = h.T.dot(b_p) + reg * W2 # H x C
    grads['b2'] = np.sum(b_p, axis=0) # C x 1

    dh = b_p.dot(W2.T)
    ds = dh * (z > 0)  # ReLU 미분

    grads['W1'] = X.T.dot(ds) + reg * W1 # D x H
    grads['b1'] = np.sum(ds, axis=0) # H x 1

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

    return loss, grads
```

F. Backward pass 구현

i. softmax를 사용하여 계산한 data loss 값의 미분 결과를 b_p에 저장

- softmax + cross-entropy → 정답 클래스의 확률에 -1

ii. W2, b2: output layer의 가중치와 편향의 gradient 계산

iii. dh, ds: hidden layer에 전달하기 위한 gradient 계산

- ReLU 활성화 함수에 대한 gradient 적용

iv. W1, b1: hidden layer의 가중치와 편향의 gradient 계산

G. 반환값: loss, grads

4) 학습(train) 함수

```
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.
    - 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 = []
```

A. 파라미터
   i. X, y: 학습 데이터와 학습 레이블
   ii. X_val, y_val: 검증 데이터와 검증 레이블
   iii. learning_rate: 최적화를 위한 학습률
   iv. learning_rate_decay: 학습률 감소 계수
   v. reg: 정규화 강도
   vi. num_iters: 최적화 과정에서의 전체 반복 수
   vii. batch_size: 단계 별로 사용할 학습 샘플의 수
   viii. verbose: 최적화 과정 출력 여부

B. Stochastic Gradient Descent(SGD)를 활용하여 파라미터를 학습

```
    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.                             #
        #########################################################################

        random_idxs = np.random.choice(num_train, batch_size)
        X_batch = X[random_idxs]
        y_batch = y[random_idxs]

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

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

        #########################################################################
        # 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.                         #
        #########################################################################

        self.params['W2'] -= learning_rate * grads['W2']
        self.params['b2'] -= learning_rate * grads['b2']
        self.params['W1'] -= learning_rate * grads['W1']
        self.params['b1'] -= learning_rate * grads['b1']

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

C. 미니 배치 샘플링: numpy의 random.choice 함수를 활용하여 선택된 랜덤 인덱스를 기반으로 배치의 크기(X_batch, y_bvatch)를 샘플링
D. loss 함수를 호출하여 손실(loss) 및 gradient(grad) 계산
E. Gradient Descent: 파라미터를 업데이트

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

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

F. 매 epoch마다 학습 정확도와 검증 정확도를 평가
G. 학습률 감소
H. 반환값: loss_history, train_acc_history, val_acc_history

5) 예측(predict) 함수

```
    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
        params = self.params
        ###########################################################################
        # TODO: Implement this function; it should be VERY simple!                #
        ###########################################################################

        z = X.dot(params['W1']) + params['b1']
        h = np.maximum(z, 0)
        p = h.dot(params['W2']) + params['b2']

        y_pred = np.argmax(p, axis=1) # 각 행에서 가장 큰 값이 있는 열의 인덱스

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

        return y_pred
```

A. forward pass 연산 후 가장 점수가 높은 클래스의 인덱스를 반환

2. two_layer_net.ipynb

1) 기본 설정 및 유틸리티 함수 정의

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.

In [1]:
```
# A bit of setup

import numpy as np
import matplotlib.pyplot as plt # library for plotting figures

from classifiers.neural_net import TwoLayerNet

%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'

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))))
```

We will use the class `TwoLayerNet` in the file `classifiers/neural_net.py` 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 numpy arrays. Below, we initialize toy data and a toy model that we will use to develop your implementation.

A. 라이브러리 불러오기
   i. numpy, matplotlib.pyplot 모듈을 불러옴
   ii. neural_net.py에서 생성한 TwoLayerNex 클래스를 불러옴
B. 그래프의 기본 설정: 그래프의 기본 크기와 보간 방식, 색상맵 설정
C. 유틸리티 함수 정의: 상대 오차를 계산

2) 가중치 및 점수 검증

In [2]:
```
# 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()
```

A. 파라미터 지정
B. init_toy_model 함수 정의: 지정된 파라미터로 TwoLayerNet 객체를 초기화
C. init_toy_data 함수 정의: 테스트 데이터를 입력하고 레이블 데이터를 생성하기 위한 함수 정의

3) forward pass 수행: 점수(scores) 계산

Forward pass: compute scores

Open the file `classifiers/neural_net.py` and look at the method `TwoLayerNet.loss`. This function is very similar to the loss functions you have written for the SVM and Softmax exercises: It takes the data and weights and computes the class scores, the loss, and the gradients on the parameters.

Implement the first part of the forward pass which uses the weights and biases to compute the scores for all inputs.

In [3]:
```
scores = net.loss(X)
print('Your scores:')
print(scores)
print()
print('correct scores:')
correct_scores = np.asarray([
  [-0.81233741, -1.27654624, -0.70335995],
  [-0.17129677, -1.18803311, -0.47310444],
  [-0.51590475, -1.01354314, -0.8504215 ],
  [-0.15419291, -0.48629638, -0.52901952],
  [-0.00618733, -0.12435261, -0.15226949]])
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:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]

correct scores:
[[-0.81233741 -1.27654624 -0.70335995]
 [-0.17129677 -1.18803311 -0.47310444]
 [-0.51590475 -1.01354314 -0.8504215 ]
 [-0.15419291 -0.48629638 -0.52901952]
 [-0.00618733 -0.12435261 -0.15226949]]

Difference between your scores and correct scores:
3.680272087688147e-08
```

A. TwolayerNet 클래스의 손실 함수를 호출하여 scores에 저장
B. 정답 점수값(correct_scores)과 비교하여 구현 정확도를 확인
C. 상대 오차 확인

➔ 실행 결과: scores에 저장한 결과와 정답 간의 절대값 차이의 총합을 계산하여 출력하였다. 결과값이 1e-7보다 작은 값이기 때문에 forward pass가 정확하게 구현되었다고 할 수 있다.

4) forward pass 수행: 손실(loss) 계산

Forward pass: compute loss

In the same function, implement the second part that computes the data and regularization loss.

In [4]:
```
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.30378789133

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

```
Difference between your loss and correct loss:
0.018965419606062905
```

A. loss 함수 호출하여 net 클래스의 손실 값 계산
B. 정답 손실 값(correct_loss)와 비교

➔ 실행 결과: loss에 저장한 결과와 정답 손실 값 간의 차이를 절댓값으로 출력하였다. 결과값이 1e-12보다 매우 큰 값이므로, 손실 정도가 크다고 할 수 있다.

5) Backward pass 수행

Backward pass

Implement the rest of the function. This 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:

In [5]:
```
from gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the backward pass.
# 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], verbose=False)
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num, grads[param_name])))
```

```
W2 max relative error: 3.440708e-09
b2 max relative error: 4.447677e-11
W1 max relative error: 3.561318e-09
b1 max relative error: 2.738421e-09
```

A. eval_numerical_gradient(f, W): 주어진 함수 f에 대해 변수 W의 수치적인 gradient를 계산
B. grads 내 모든 파라미터에서 수치 미분과 backward pass의 오차 비교

➔ 실행 결과: backward pass 구현을 통해 구한 기울기와 eval_numerical_gradient 함수를 통해 수치적으로 계산한 기울기 간의 차이를 비교한 값을 rel_error 함수를 통해 출력하였다. 모든 파라미터에서의 상대 오차 값이 1e-8보다 작으므로, backward pass가 정확하게 구현되었다고 할 수 있다.

6) 네트워크 학습 및 시각화

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

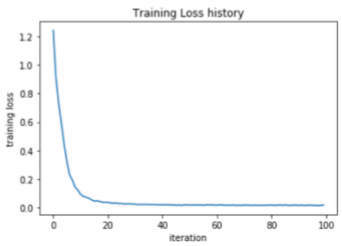
Once you have implemented the method, run the code below to train a two-layer network on toy data. You should achieve a training loss less than 0.02.

In [6]:
```
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:  0.017143643532923757
```



A. train 함수 호출을 통해 net 클래스의 2계층 신경망을 학습
B. stats 딕셔너리를 통해 반복별 손실값 리스트(stats['loss_history'])를 확인
C. 그래프를 통해 학습 중 손실이 줄어드는 경향을 시각화

➔ 실행 결과: 최종 학습 데이터의 손실 값은 0.02보다 작은 0.01710이다.

7) CIFAR-10 데이터 불러오기 및 전처리

Load the 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.

```
In [7]: from data_utils import load_CIFAR10

def get_CIFAR10_data(num_training=19000, 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. These are the same steps as
    we used for the SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'datasets/cifar-10-batches-py'

    # Cleaning up variables to prevent loading data multiple times (which may cause memory issue)
    try:
        del X_train, y_train
        del X_test, y_test
        print('Clear previously loaded data.')
    except:
        pass

    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # Subsample the data
    # train / test -> train + val / test
    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:  (19000, 3072)
Train labels shape:  (19000,)
Validation data shape:  (1000, 3072)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3072)
Test labels shape:  (1000,)
```

A. get_CIFAR10_data 함수 생성: disk 내에 있는 CIFAR-10 데이터 세트를 로드하고, 변수를 정리하기 위한 함수
B. load_CIFAR10 함수를 통해 CIFAR-10 데이터를 읽어오기
C. 데이터 전처리
   i. 학습/검증/테스트 데이터를 분리
   ii. 평균 이미지를 제거하여 정규화
   iii. 데이터를 (N, D) 형태로 펼침

➔ 실행 결과: CIFAR-10 데이터 세트의 shape

8) CIFAR-10에서 TwoLayerNet 네트워크 학습

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.

In [8]:
```
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)
```

```
iteration 0 / 1000: loss 2.302792
iteration 100 / 1000: loss 2.302473
iteration 200 / 1000: loss 2.298877
iteration 300 / 1000: loss 2.277810
iteration 400 / 1000: loss 2.212994
iteration 500 / 1000: loss 2.133865
iteration 600 / 1000: loss 2.069171
iteration 700 / 1000: loss 2.078976
iteration 800 / 1000: loss 2.072534
iteration 900 / 1000: loss 2.005639
Validation accuracy:  0.243
```

A. 파라미터 설정
   i. input_size, hidden_size, num_classes를 설정
   ii. TwoLayerNet 클래스를 통해 net 설정
B. train 함수 호출을 통해 전체 네트워크 학습 수행
C. predict 함수 호출을 통해 검증 세트에서 예측 정확도 계산

9) 학습 과정 시각화

Debug the training

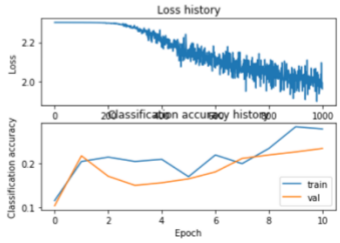
With the default parameters we provided above, you should get a validation accuracy of about 0.27 on the validation set. This isn't very good.

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.

In [9]:
```
# Plot the loss function and train / validation accuracies
plt.subplot(2, 1, 1)
plt.plot(stats['loss_history'])
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')

plt.subplot(2, 1, 2)
plt.plot(stats['train_acc_history'], label='train')
plt.plot(stats['val_acc_history'], label='val')
plt.title('Classification accuracy history')
plt.xlabel('Epoch')
plt.ylabel('Classification accuracy')
plt.legend()
plt.show()
```



A. 함수에서 저장한 반복 별 손실 리스트를 그래프로 시각화
B. 각 Epoch별로 저장된 학습/검증 정확도를 그래프로 시각화

➔ 실행 결과

➢ 반복에 따른 손실 값의 변화

- 초기 손실 값은 약 2.302이고, 점차 감소하며 마지막 손실 값은 약 2.00이다. 이를 통해 학습을 통해 예측을 개선하고 있다는 것을 알 수 있다.

➢ 검증 정확도: 0.243

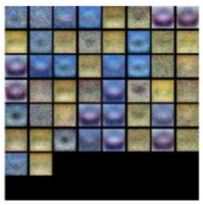
10) 학습된 가중치 시각화

```
In [10]: from vis_utils import visualize_grid

         # Visualize the weights of the network

         def show_net_weights(net):
             W1 = net.params['W1']
             W1 = W1.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(net)
```



A. show_net_weights 함수 생성: 클래스의 인스턴스 내의 첫 번째 층의 가중치를 이미지로 시각화

11) 하이퍼파라미터 조정을 통한 최상의 모델 저장

In [11]:
```
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.                          #
################################################################################

best_acc = 0
best_stats = None
results = {}

hidden_sizes = [200, 500]
learning_rates = [1e-3, 5e-4, 1e-4]
num_iters = [1000, 2000]
regularization_strengths = [0.5, 1.0]

input_size = 32 * 32 * 3
num_classes = 10


for hidden_size in hidden_sizes:
    for lr in learning_rates:
        for iters in num_iters:
            for reg in regularization_strengths:
                    net = TwoLayerNet(input_size, hidden_size, num_classes)

                    # Train the network
                    stats = net.train(X_train, y_train, X_val, y_val,
                                num_iters=iters, batch_size=200,
                                learning_rate=lr, learning_rate_decay=0.95,
                                reg=reg, verbose=True)

                    # Predict on the validation set
                    val_acc = (net.predict(X_val) == y_val).mean()

                    params = (hidden_size, lr, iters, reg)
                    results[params] = val_acc

                    #Predict on the best validation set
                    if val_acc > best_acc:
                        best_acc = val_acc
                        best_net = net
                        best_stats = stats
                        best_params = params

                    # 진행 상황 출력
                    print('=====================================================')
                    print('hidden size :',hidden_size, 'learning rate:',lr, 'iteration:', iters, 'reg:', reg)
                    print('Validation accuracy:', val_acc)
                    print('=====================================================')
```

A. 기본 파라미터 설정 및 초기화
   i. best_net: 가장 성능이 좋은 모델을 저장하기 위한 변수
   ii. best_acc: 최고의 validation accuracy
   iii. best_stats: 가장 좋은 모델의 학습 로그(loss, accuracy)
   iv. results: (hidden_size, lr, iters, reg) 조합의 validation accuracy

B. 하이퍼 파라미터
   i. hidden_sizes: hidden layer의 뉴런 수 후보
   ii. learning_rates: 학습률 후보
   iii. num_iters: 전체 학습 반복 횟수의 후보
   iv. regularization_strengths: L2 정규화 계수 후보

C. for문을 통한 하이퍼파라미터 탐색
   i. 신경망 생성 → train 함수를 통한 학습
   ii. 결과를 val_acc에 저장 후 if문을 활용하여 최고의 성능 모델을 갱신
   iii. 진행 상황마다 조합에 대한 결과 출력

12) 학습 과정 시각화

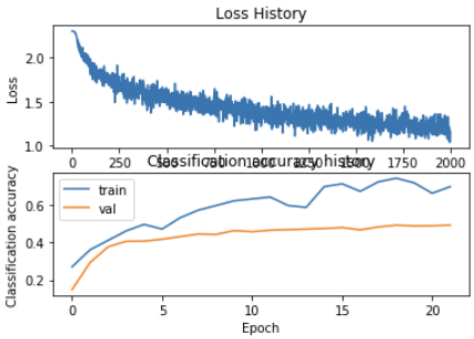
```
print('*****************************************************')
print('BEST MODEL HYPERPARAMETERS')
print('*****************************************************')
print('hidden size :',best_params[0])
print('learning rate:', best_params[1])
print('iteration:', best_params[2])
print('reg:', best_params[3])
print('Best validation accuracy:', best_acc)
print('*****************************************************')


# Plot the loss function and train / validation accuracies
plt.subplot(2, 1, 1)
plt.plot(best_stats['loss_history'])
plt.title('Loss History')
plt.xlabel('Iteration')
plt.ylabel('Loss')

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


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

```
*****************************************************
BEST MODEL HYPERPARAMETERS
*****************************************************
hidden size : 500
learning rate: 0.001
iteration: 2000
reg: 0.5
Best validation accuracy: 0.487
*****************************************************
```



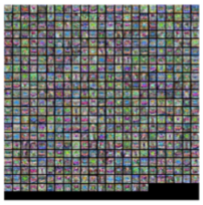
A. 하이퍼파라미터 조정을 통해 구한 최상의 모델에서의 반복 별 손실 리스트를 그래프로 시각화
B. 각 Epoch별로 저장된 학습/검증 정확도를 그래프로 시각화

➔ 실행 결과: 파라미터를 변화하기 전 검증 정확도인 0.243과 비교하였을 때, 검증 정확도가 0.487로 매우 높아졌다.

- hidden size의 값을 50→500로 증가시킴으로써 모델 용량이 증가하여 복잡한 패턴에 대해서도 학습이 가능하게 되었다.
- learning rate의 값을 1e-4→1e-3로 증가시킴으로써 초기의 학습 속도가 빨라졌다.
- iteration의 값을 1000→2000로 증가시킴으로써 최적화 시간을 확보하여 손실을 최소화하였다.
- reg의 값을 0.25→0.5로 증가시킴으로써 과적합을 방지하였다.

13) 학습된 가중치 시각화 및 테스트 세트에서 정확도 평가

In [12]:
```
# visualize the weights of the best network
show_net_weights(best_net)
```



Run on the test set

When you are done experimenting, you should evaluate your final trained network on the test set; you should get above 36%.

In [13]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.478
```

A. 최상의 네트워크에서의 가중치 시각화
B. 학습이 완료된 net을 이용하여 테스트 데이터 X_test의 각 샘플에 대해 예측 수행

➔ 실행 결과: 학습 데이터에서의 검증 정확도 0.487과 비교하였을 때, 0.478로 미세하게 낮은 정확도를 가진다. 이를 통해 학습 과정에서 과적합이 발생하였음을 알 수 있다. 이 문제를 개선하기 위한 방법으로는 첫 번째, 더 큰 데이터 세트를 사용하여 데이터의 다양성을 증가할 수 있다. 두 번째, 정규화 강도를 증가시켜 가중치의 크기를 제한함으로써 모델을 단순화시킬 수 있다.