two_layer_net.ipynb 의 code 와 markdown 중심으로 설명한다. 그 중 코드를 돌려 나온 결과는 파란색, 다른 코드에서 가져온 함수(코드)는 초록색 선으로 표시하겠다.

1) 기본 세팅

Implementing a Neural Network

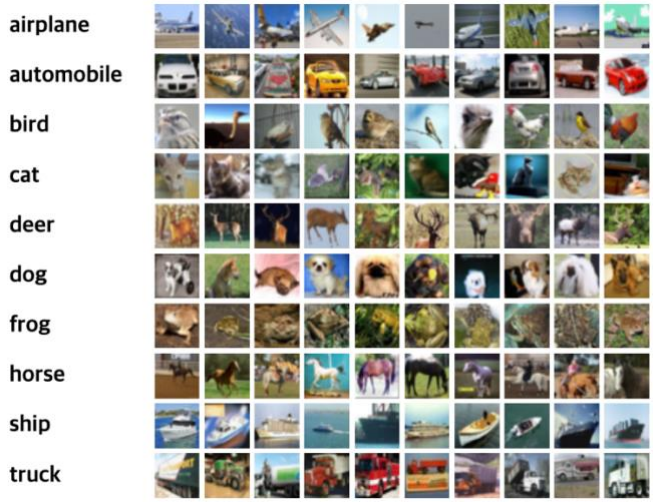
In this exercise we will develop a neural network with fully-connected layers to perform classification, and test it out on the CIFAR-10 dataset.

CIFAR-10 dataset 를 이용해 test 를 진행한다.

CIFAR-10 dataset 은 다음과 같다. 각 10 개의 클래스에 대해 3x3 의 컬러이미지 6000 장으로 구성된다. 50000 개의 training images 와 10000 개의 test images 가 존재한다.

ex 랜덤 10 개씩

cifar-10-batches-py/readme.html



기본적인 일부 세팅을 해준다

```
# 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.

행렬 활용을 위한 numpy 와 그래프를 위한 pyplot 을 import 하고, pyplot 기본세팅(크기, 보간, 색)을 해준다.

file classifiers/neural_net.py 의 TwoLayerNet 클래스를 사용할 것이다.
상대오차를 반환해주는 rel_error()함수를 정의한다

```
# 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()
```

random 을 이용해 초기화 함수 init_toy_model()과 init_toy_data()를 정의하고, net, x, y 초기화

2) Forward pass 로 score 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.

back-propagation 을 통해 각종 변수들의 gradient 를 구하기 위해서는 우선 계산 완료된 L 이 필요하므로, Forward pass 먼저 진행해준다. score, loss, gradient 등을 계산하자.

```
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)))
```

net 은 init_toy_model()로 초기화한 변수이고, init_toy_model()는 TwoLayerNet 를 반환한다. 즉, net.loss 는 TwoLayerNet class 의 loss 함수를 이용한다는 의미이다.
아래 loss 함수에 대한 설명이다

```
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.
  """
```

설명 요약:

two layer fully connected neural network 의 loss 와 gradient 반환

입력: X: (N, D)의 입력데이터

y: training labels 의 벡터

reg: 정규화 강도

반환: y 없으면 score 만, 있으면 loss 랑 gradient 까지 반환

```
# 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
```

변수설정

s=Wx+b 를 두번 수행, 사이에 ReLU 적용하면,

$W_1x + b_1$ -> $\max(0, W_1x + b_1)$ -> $W_2(\max(0, W_1x + b_1)) + b_2$

따라서 코드는

```
s1=X.dot(W1) + b1          # s=Wx+b
X2=np.maximum(0, s1)       # ReLU 적용
scores=X2.dot(W2)+b2       # s=Wx+b
```

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

# Compute the loss
loss = None
```

y값이 없다면 scores만 반환,

y가 있다면 loss와 gradient의 계산을 시작한다

softmax classifier loss를 구하는 식은 이와 같다.

$$L_i = -\log_2\left(\frac{e^{s_{y_i}}}{\sum_{j=1}^{C} e^{s_j}}\right)$$

```
scores_exp=np.exp(scores)    # e^(s)
softmax_mat=scores_exp/np.sum(scores_exp, axis=1, keepdims=True) # e^(s)/전체 e^(s)합
loss=np.sum(-np.log(softmax_mat[np.arange(N), y]))  # 정답 클래스의 값에 로그 씌우고 -붙이기
```

이렇게 L을 계산했으므로 이제 back propagation을 진행하자

$$\frac{\partial L}{\partial \mathbf{W}_1} = \frac{\partial L}{\partial \boldsymbol{s}_1}\boldsymbol{x}^{\mathrm{T}} \qquad \frac{\partial L}{\partial \mathbf{W}_2} = \frac{\partial L}{\partial \boldsymbol{s}_2}\boldsymbol{p}_1^{\mathrm{T}} \qquad \frac{\partial L}{\partial \mathbf{W}} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial L_i(f(\boldsymbol{x}_i,\mathbf{W}),y_i)}{\partial \mathbf{W}} + \lambda\frac{\partial R(\mathbf{W})}{\partial \mathbf{W}}$$

식을 이용해 W1, W2, b1, b2를 구해준 뒤, 변수에 저장해 반환한다

```
softmax_mat[np.arange(N) ,y] -= 1
softmax_mat /= N

dW2 = X2.T.dot(softmax_mat)
db2 = softmax_mat.sum(axis=0)

dW1 = softmax_mat.dot(W2.T)
dfc1 = dW1 * (s1>0)
dW1 = X.T.dot(dfc1)
db1 = dfc1.sum(axis=0)

dW1 += reg * W1
dW2 += reg * W2

grads['W1'] = dW1
grads['b1'] = db1
grads['W2'] = dW2
grads['b2'] = db2
```

여기서는 매개변수로 X 만 보냈으므로, score 만 계산한 것이다. 구한 score 을 출력, correct score 값과의 편차를 출력한다.

```
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
```

Forward pass: compute loss

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

```
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:
5.025497369260268
```

위에서 초록색 창으로 설명했던 loss 함수를 이용해 다시한번 loss 와 gradient 를 계산한다. 제공된 correct loss 와의 차이도 출력한다.

2) back-propagation: loss 와 gradient 계산

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:

변수 W1, b1, W2 및 b2 에 대한 gradient 를 계산한다. forward pass 로 구한 값을 이용해 구할 수 있다.

```
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
```

loss 함수를 호출한다. 이번에는 매개변수 y 가 있으므로, loss 와 gradient 계산을 진행한다.

gradient_check.py 의 eval_numerical_gradient 함수를 호출한다.

```python
def eval_numerical_gradient(f, x, verbose=True, h=0.00001):
    """
    a naive implementation of numerical gradient of f at x
    - f should be a function that takes a single argument
    - x is the point (numpy array) to evaluate the gradient at
    """

    fx = f(x) # evaluate function value at original point # 계산할 함수
    grad = np.zeros_like(x)
    # iterate over all indexes in x
    it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
    while not it.finished:

        # evaluate function at x+h
        ix = it.multi_index
        oldval = x[ix]                              # f(x)
        x[ix] = oldval + h # increment by h
        fxph = f(x) # evalute f(x + h)              # f(x+h)
        x[ix] = oldval - h
        fxmh = f(x) # evaluate f(x - h)             # f(x-h)
        x[ix] = oldval # restore

        # compute the partial derivative with centered formula
        grad[ix] = (fxph - fxmh) / (2 * h) # the slope     # 중심차분 {f(x+h)-f(x-h)}/2h
        if verbose:
            print(ix, grad[ix])                     # 결과 축력
        it.iternext() # step to next dimension

    return grad
```

이 함수는 중심차분을 이용하여 함수 f 의 gradient 를 구하는 함수이다.

수치적인 gradient 를 이용하기 때문에 미분을 직접 계산하지 않고 기울기를 이용해 근사적으로 구한다.

$$D_h f(x) = \frac{f(x+h) - f(x-h)}{2h}.$$

이와 같은 결과가 나온다.

```
W1 max relative error: 1.000000e+00
b1 max relative error: 6.666667e-01
W2 max relative error: 1.000000e+00
b2 max relative error: 6.666667e-01
```

3) Training

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.

네트워크를 훈련하기 위해 SVM 및 Softmax 분류기와 유사한 확률적 경사 하강법(SGD)을 사용할 것이다.

```
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()
```

neural_net.py 의 train 함수를 호출해 training 을 진행한다

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

확률적 경사 하강법을 이용해 신경망을 훈련한다.

training 데이터에서 랜덤으로 minibatch 를 생성한다.

```
batch_ran = np.random.choice(num_train, batch_size)  # 랜덤으로 data를 batch_size만큼 뽑는다
X_batch = X[batch_ran]   # 변수에 저장
y_batch = y[batch_ran]
```

구한 random data 세트에 대해 loss 와 gradient 를 계산한다

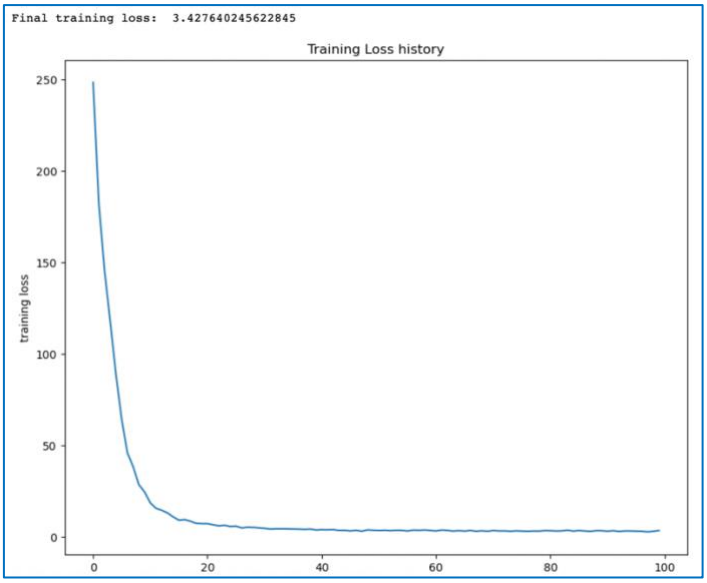
```
loss, grads = self.loss(X_batch, y=y_batch, reg=reg) # loss function you completed above
loss_history.append(loss)
```

구한 gradient 를 이용해 W 와 b parameter 를 업데이트한다.
params 에는 W1, W2, b1,b2 가 저장되어있다. for 문으로 모든 parameter 에 대해 아래의 식과 같이 update 해준다.

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

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

결과를 그래프로 출력한다.



100 번의 값 update 를 통해 loss 가 줄어드는 것을 확인할 수 있다.

4) CIFAR-10 data 로 training 수행

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.

CIFAR-10 데이터를 로드하여 실제 데이터 세트에서 classifier 를 train 해보자.

```python
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
```

이 함수는 CIFAR-10 의 데이터셋을 로드하고 변수를 초기화 한 뒤, 데이터를 학습, 검증, 테스트 데이터로 나눈다. 데이터에서 평균 이미지를 빼서 정규화하고, 이미지를 벡터로 변환해 모델에 입력할 수 있게 한다.

```
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,)
```

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.

SGD 를 이용해 training 한다.

SGD 의 식은 다음과 같다

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i(f(\boldsymbol{x}_i, \mathbf{W}), y_i) + \lambda R(\mathbf{W})$$

```
from classifiers.neural_net import TwoLayerNet
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)
```

위 함수에서 형태를 맞춰준 데이터들을 train 함수를 이용해 training 해준다.

neural_net.py 의 predict 함수를 호출한다.

```
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.
  """
```

이 함수는 two-layer 네트워크의 trained weight 를 사용해 각 클래스의 점수를 예측하고 가장 높은 점수를 가진 클래스를 반환한다.

```
y_pred = np.argmax( self.loss(X), axis=1)

# 가장 큰 점수를 가지는 클래스를 찾아 y_pred 변수에 저장한다
```

```
iteration 0 / 1000: loss 460.519927
iteration 100 / 1000: loss 460.456135
iteration 200 / 1000: loss 459.735859
iteration 300 / 1000: loss 455.512344
iteration 400 / 1000: loss 442.514281
iteration 500 / 1000: loss 426.636459
iteration 600 / 1000: loss 413.647164
iteration 700 / 1000: loss 415.558436
iteration 800 / 1000: loss 414.222924
iteration 900 / 1000: loss 400.799266
Validation accuracy:  0.243
```

5) 개선

Debug the training

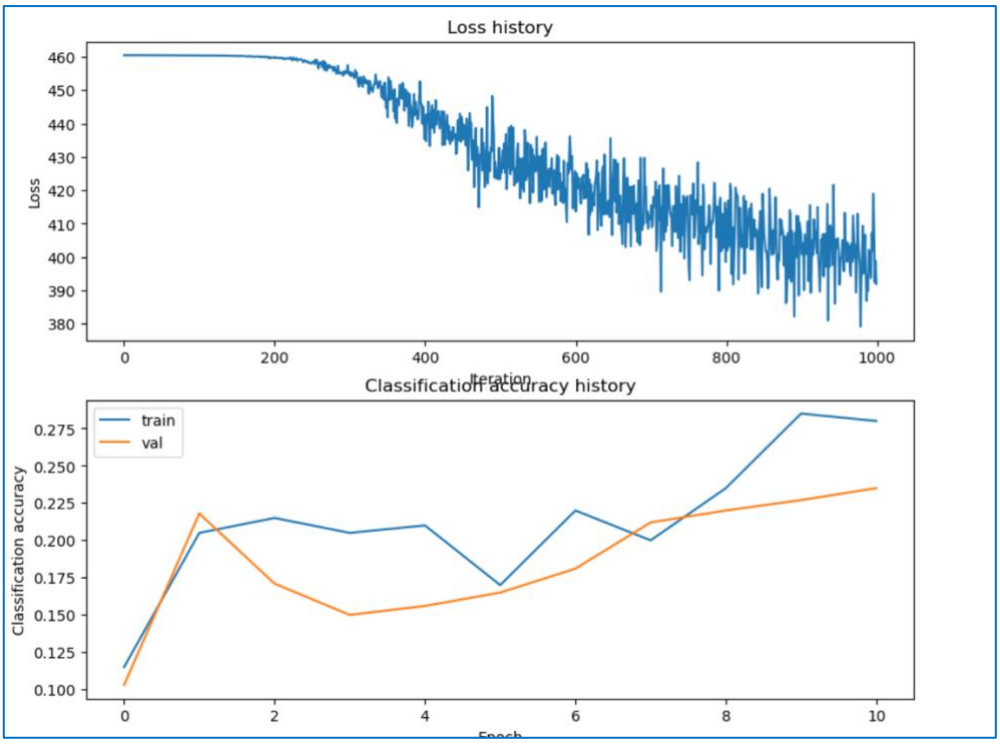
With the default parameters we provided above, you should get a validation accuracy of about 0.27 on the validation set. This isn't very good.

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.

유효성 검사 결과 0.243 을 얻어 별로 좋지 않은 결과이므로, loss 함수와 training 과 validation 의 정확성을 표시하자. 또한 첫번째 계층에서 학습된 가중치를 시각화하자

```
import matplotlib.pyplot as plt
# 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()
```

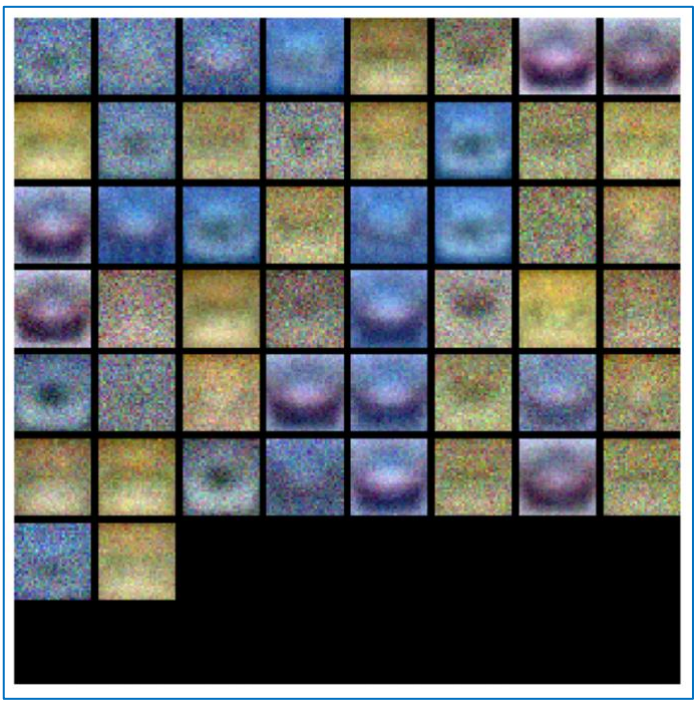



```
from vis_utils import visualize_grid

# 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)
```

vis_utils.py 의 visualize_grid 함수를 호출한다.

이 함수는 이미지 데이터를 그리드 형태로 시각화 해준다.

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

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



Tune your hyperparameters

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

Tuning. Tuning the hyperparameters and developing intuition for how they affect the final performance is a large part of using Neural Networks, so we want you to get a lot of practice. Below, you should experiment with different values of the various hyperparameters, including hidden layer size, learning rate, numer of training epochs, and regularization strength. You might also consider tuning the learning rate decay, but you should be able to get good performance using the default value.

Approximate results. You should be aim to achieve a classification accuracy of greater than 36% on the validation set. Our best network gets over 39% on the validation set.

Experiment: You goal in this exercise is to get as good of a result on CIFAR-10 as you can (39% could serve as a reference), with a fully-connected Neural Network. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

요약:

loss 가 선형적으로 감소한다. 학습률이 너무 낮다. 훈련과 검증 정확도 사이의 차이가 없으므로, 사용한 모델의 용량의 크기를 늘려야 한다. 하지만 모델이 너무 크면 overfitting 의 위험이 있다 많은 practice 로 실험해야한다.

classification accurary 36%이상을 목표로 한다.

YourAnswer : 랜덤으로 learing rate, reg, hidden_size를 정하고, training을 수행한다. 20번 수행 후 classification accuracy가 가장 큰 경우를 best_net으로 정한다

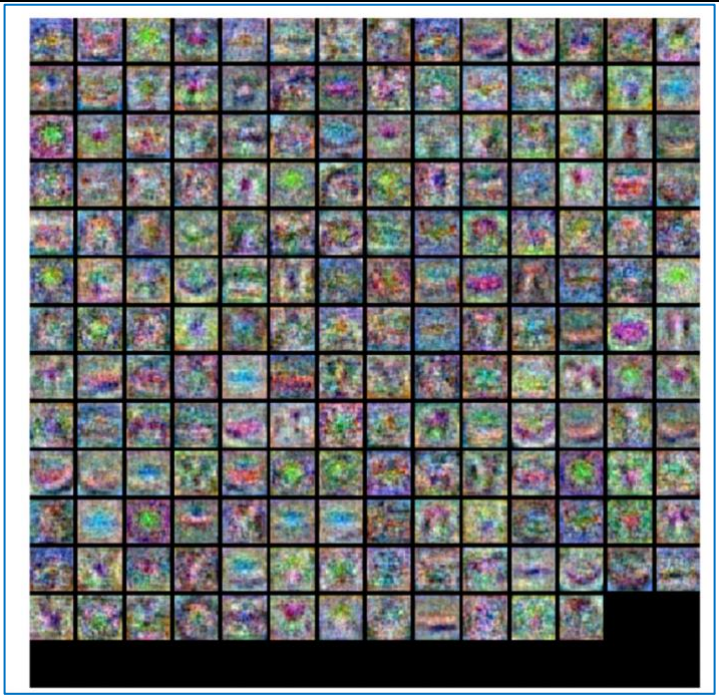
랜덤으로 parameter 를 뽑아서 20 번 training 을 수행하고 가장 accurary 가 큰 경우를 답으로 정하자

```
def random_search_hyperparams(lr_values, reg_values, h_values): # 랜덤하게 lr, reg, hidden을 정하는 함수
    lr = lr_values[np.random.randint(0,len(lr_values))]
    reg = reg_values[np.random.randint(0,len(reg_values))]
    hidden = h_values[np.random.randint(0,len(h_values))]
    return lr, reg, hidden

input_size = 32 * 32 * 3
num_classes = 10

np.random.seed(0)

for i in range(20):
    lr, reg, hidden_size = random_search_hyperparams([0.001], [0.05, 0.1, 0.15], [50, 80, 100, 120, 150, 180, 200])
    net = TwoLayerNet(input_size, hidden_size, num_classes)
    stats = net.train(X_train, y_train, X_val, y_val,              # 반환받은 랜덤한 수에 따라 train진행
                num_iters=2000, batch_size=200,
                learning_rate=lr, learning_rate_decay=0.95,
                reg=reg, verbose=False)

    train_accuracy = (net.predict(X_train) == y_train).mean()

    val_accuracy = (net.predict(X_val) == y_val).mean()

    if val_accuracy > best_val:    # best_val 보다 크다면 그것을 답으로 지정
        best_val = val_accuracy
        best_net = net
        best_stats = stats

    print('lr %e reg %e hid %d  train accuracy: %f val accuracy: %f' % (
                lr, reg, hidden_size, train_accuracy, val_accuracy))
print('best validation accuracy achieved: %f' % best_val)
```

결과:

```
lr 1.000000e-03 reg 5.000000e-02 hid 180  train accuracy: 0.660263 val accuracy: 0.487000
lr 1.000000e-03 reg 1.500000e-01 hid 180  train accuracy: 0.649421 val accuracy: 0.478000
lr 1.000000e-03 reg 1.500000e-01 hid 100  train accuracy: 0.629579 val accuracy: 0.485000
lr 1.000000e-03 reg 5.000000e-02 hid 180  train accuracy: 0.653684 val accuracy: 0.480000
lr 1.000000e-03 reg 1.000000e-01 hid 200  train accuracy: 0.658842 val accuracy: 0.481000
lr 1.000000e-03 reg 5.000000e-02 hid 200  train accuracy: 0.659211 val accuracy: 0.474000
lr 1.000000e-03 reg 5.000000e-02 hid 180  train accuracy: 0.657316 val accuracy: 0.471000
lr 1.000000e-03 reg 1.500000e-01 hid 50  train accuracy: 0.584789 val accuracy: 0.457000
lr 1.000000e-03 reg 5.000000e-02 hid 100  train accuracy: 0.637474 val accuracy: 0.479000
lr 1.000000e-03 reg 5.000000e-02 hid 80  train accuracy: 0.617684 val accuracy: 0.460000
lr 1.000000e-03 reg 1.500000e-01 hid 180  train accuracy: 0.647211 val accuracy: 0.491000
lr 1.000000e-03 reg 5.000000e-02 hid 180  train accuracy: 0.655632 val accuracy: 0.479000
lr 1.000000e-03 reg 1.000000e-01 hid 120  train accuracy: 0.635474 val accuracy: 0.479000
lr 1.000000e-03 reg 5.000000e-02 hid 200  train accuracy: 0.658737 val accuracy: 0.482000
lr 1.000000e-03 reg 1.000000e-01 hid 100  train accuracy: 0.632526 val accuracy: 0.460000
lr 1.000000e-03 reg 1.000000e-01 hid 80  train accuracy: 0.615737 val accuracy: 0.475000
lr 1.000000e-03 reg 1.500000e-01 hid 150  train accuracy: 0.643105 val accuracy: 0.488000
lr 1.000000e-03 reg 1.000000e-01 hid 150  train accuracy: 0.649158 val accuracy: 0.476000
```

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



6) 최종 평가

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%.

최종 훈련된 네트워크를 평가

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

```
Test accuracy:  0.497
```

Inline Question

Now that you have trained a Neural Network classifier, you may find that your testing accuracy is much lower than the training accuracy. In what ways can we decrease this gap? Select all that apply.

1. Train on a larger dataset.
2. Add more hidden units.
3. Increase the regularization strength.
4. None of the above.

Your Answer : 1, 3

Your Explanation : overfitting을 줄이기 위해 training size를 늘리거나 정규화 정도를 높일 수 있다.