Technical report-Assignment09

컴퓨터공학 류다현(2376087)

1. neural_net.py

TwoLayerNet 클래스에 two-layer fully-connected 신경망을 구현한다. 신경망의 구조는

입력 → FC → ReLU → FC → softmax

이며, 최종 예측 결과에 cross-entropy loss와 L2 regularization을 적용한다.

1.1 def __init__(self, input_size, hidden_size, output_size, std=1e-4)

__init__()에서는 가중치를 임의의 작은 값으로, 편향을 0으로 설정해 모델을 초기화한다. 입력 파라미터는 다음과 같다.

- input_size = 입력 차원 D
- hidden_size = 은닉층 뉴런 수 H
- output_size = 출력 차원(클래스 개수) C
- std = 초기값 표준편차

이후 아래와 같이 가중치와 편향을 초기화한다.

```
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)
```

이때 파라미터들을 딕셔너리에 저장함으로써 다른 함수에서의 파라미터 사용과 업데이트가 용이하도록 한다.

1.2 def loss(self, X, y=None, reg=0.0)

y를 입력받지 않은 경우 loss()에서는 순전파로 scores를 계산해 반환하고, y를 입력받았을 경우 scores, 예측 확률, 손실을 계산한 뒤 역전파를 통해 기울기를 계산하여 손실 loss와 기울기 grads를 반환한다.

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

1.1에서 self.params 딕셔너리에 저장되었던 파라미터들을 W1, b1과 W2, b2에 옮겨 저장하고, N과 D에 입력 X의 크기인X.shape를 저장한다.

```
scores = None

s1 = X.dot(W1) + b1
p1 = np.maximum(0, s1)
scores = p1.dot(W2) + b2
```

첫 번째 FC layer에서 Wx+b를 계산한 것을 s1에 저장하고, ReLU를 적용하기 위해 maximum()으로 0과 s1 중 더 큰 것만 유지하도록 한 것을 p1에 저장한다. 이후 p1과 W2를 dot product하고 b2를 더한 것을 scores에 저장하며, 이것이 두 번째 FC layer의 결과 점수이다.

```
if y is None:
      return scores
```

y가 None이라면, 즉 loss() 호출 시 y를 입력받지 않았다면 더 이상 진행하지 않고 지금까지 계산한 점수 scores를 반환하며 함수를 종료한다. y를 전달받았으면 softmax 활성화 함수와 cross-entropy 손실 함수, L2 regularization을 적용해 scores를 확률로 변환하고 손실과 기울기를 계산한다.

```
exp_s = np.exp(scores)
p2 = exp_s / np.sum(exp_s, axis=1, keepdims=True)
```

softmax를 적용하기 위해 점수 scores에 대해 지수 함수를 적용해 exp_s를 계산하고, 이를 각 행의 값으로 나눠 확률 벡터 p2를 구한다.

```
correct = -np.log(p2[np.arange(N), y])
data_loss = np.sum(correct) / N
regularization = reg * (np.sum(W1*W1) + np.sum(W2*W2))
loss = data_loss + regularization
```

L2 정규화할 경우 최종 손실 $L = \frac{1}{N}\sum_{i=1}^{N} L_i\left(f(x_i, \boldsymbol{W}), y_i\right) + \lambda R(\boldsymbol{W})$이다. 이때 score에 softmax를 적용한 것인 $f(x_i, \boldsymbol{W})$를 앞에서 계산해 p2에 저장했었고, 따라서 각 샘플 i에 대한 정답 클래스 y_i의 확률 p2[np.arange(N), y]에 -log를 취함으로써 cross-entropy 손실 함수를 적용한 뒤 샘플 개수 N으로 나눠 평균한 것을 것을 data_loss에, 가중치 제곱에 L2 정규화 강도 파라미터인 reg을 곱한 것을 reg_loss에 나누어 저장한다. 이후 data_loss와 reg_loss를 합쳐 최종 손실인 loss를 계산한다.

이후 아래와 같이 역전파를 통해 기울기 grad를 계산한다.

```
# dL/dscore
dscores = p2
dscores[np.arange(N), y] -= 1 # one-hot 이므로 z=1, dL/ds_2=p-z
dscores /= N  # 위에서 1/N 했었으니까 여기서도 나눠 줘야
# -> dscores = dL/ds_2

# dL/dW2
grads['W2'] = p1.T.dot(dscores) + 2 * reg * W2
# dL/dW2 = dscore*transpose(T)
# 2*reg*W2 는 reg_loss 미분
grads['b2'] = np.sum(dscores, axis=0) # dL/db_2 = dL/ds_2 = sum(dscores)

# dL/dp_1
dp1 = dscores.dot(W2.T) # (W2)^T*dL/ds_2 = transpose(W2)*dscores
dp1[p1 <= 0] = 0  # ReLU 했었으므로 p1<=0 인 인덱스의 dp1 값들은 0

# dL/dW1
grads['W1'] = X.T.dot(dp1) + 2 * reg * W1
```

```
# dL/ds1 = diag((1-sigma(s))*sigma(s))*dp1
# dL/dW1 = dL/ds_1*x^T = transpose(x)*dp1
# 2*reg*W1 은 reg_loss 미분
grads['b1'] = np.sum(dp1, axis=0) # dL/db_1 = dL/ds_1 = sum(dp1)
```

전체 역전파 과정은 Lec12 강의 자료 슬라이드 33에 서술된 것을 그대로 구현하였으며, 각 코드가 의미하는 바는 주석으로 설명되어 있다.

```
return loss, grads
```

이후 계산된 손실 loss와 기울기 grads를 반환하고 loss() 함수를 종료한다.

1.3 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() 함수는 SGD optimizer를 이용해 위에서 구현한 신경망을 훈련한다. 입력 파라미터는 다음과 같다.

- X: 훈련 데이터
- y: 훈련 데이터 정답 클래스
- X_val: validation 데이터
- y_val: validation 데이터 정답 클래스
- learning_rate: 학습률
- learning_rate_decay: 가중치 decay 정도
- batch_size: 배치 크기
- verbose: 진행 상황 출력 여부

```
num_train = X.shape[0]
iterations_per_epoch = max(num_train / batch_size, 1)
```

훈련 데이터 크기와 epoch당 반복 횟수를 각각 num_train, iterations_per_epoch에 저장한다.

```
loss_history = []
train_acc_history = []
val_acc_history = []
```

loss_history는 계산된 손실 값, train_acc_history는 계산된 훈련 데이터셋 정확도, val_acc_history는 validation 데이터셋 정확도를 저장하는 배열이다.

```
for it in range(num_iters):
      X_batch = None
      y_batch = None

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

이후 반복 횟수 num_iters만큼 반복하며 무작위로 생성한 미니배치에 대해 손실과 기울기를 계산한 뒤 파라미터를 갱신한다. 0부터 num_train까지 배치 크기 batch_size개의 랜덤 인덱스를 추출해 random_indices에 저장하고, 이에 해당하는 데이터를 X_batch와 y_batch에 저장한다.

```
loss, grads = self.loss(X_batch, y=y_batch, reg=reg)
loss_history.append(loss)
```

이후 위에서 작성한 loss()에 X_batch와 y_batch, regularization 강도 파라미터를 전달해 손실과 기울기를 계산하고, 이를 loss와 grads에 저장한다. 매 단계 손실이 어떻게 변화하는지 기록하기 위해 loss_history 배열에 계산한 손실을 append한다.

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

$W' = W - \alpha\frac{\partial L}{\partial w}$ 공식에서 $\frac{\partial L}{\partial w}$가 grad이므로, learning_rate*grads[i]한 값으로 새로운 가중치를 계산해 업데이트한다.

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

중간 계산 결과 출력 여부 verbose가 true일 경우 100번 반복할 때마다 반복 횟수와 손실을 출력한다.

```
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
```

매 epoch마다 훈련 데이터셋 정확도와 validation 데이터셋 정확도를 출력하고, 현재 학습률 learning_rate에 학습률 감소 파라미터 learning_rate_decay를 곱해 학습이 진행됨에 따라 학습률을 줄인다. 이를 통해 학습 초반에는 큰 학습률로 손실 함수의 최소점으로 빠르게 수렴할 수 있고, 학습 후반에는 작은 학습률로 파라미터 업데이트 정도를 작게 해 미세 조정함으로써 과적합을 방지할 수 있다.

1.4 def predict(self, X)

predict()에서는 학습한 파라미터로 순전파를 수행해 예측값을 계산한다.

```
s1 = X.dot(self.params['W1']) + self.params['b1'] # W1x + b1
p1 = np.maximum(0, s1)  # ReLU
s2 = p1.dot(self.params['W2']) + self.params['b2']  # W2X + b2
y_pred = np.argmax(s2, axis=1)

return y_pred
```

s1은 첫 번째 FC layer의 결과, p1은 s1에 ReLU 함수를 적용한 값, s2는 p1을 입력으로 하여 두 번

째 FC layer에서 계산한 결과이며, 최종 예측 값 y_pred는 샘플별로 s2 중 값이 가장 큰 클래스 인덱스이다. y_pred를 반환함으로써 확률이 가장 높은 클래스, 즉 모델이 예측한 클래스를 반환한다.

2. 실행 결과

two_layer_net.ipnb는 개발 환경을 설정하고 사용할 라이브러리와 구현한 TwoLayerNet 클래스를 로드한 구현이 적절한지 단계별로 검증하며, 이 과정에서 loss()를 호출해 중간 결과들을 기준 값들과 비교하고 train()을 호출해 손실이 정상적으로 줄어드는지 확인한다. 이후 CIFAR-10 데이터셋을 불러와 훈련 데이터셋, validation 데이터셋, 테스트 데이터셋으로 분할하고 reshape해 전처리한 뒤 학습 및 예측한 결과를 출력한다

구글 colab 개발 환경에서 수행하기 위해 파일 상단에 구글 드라이브를 import하고 작업 경로에 과제 폴더인 '/content/drive/MyDrive/25-spring-openSWproject/Assignment09'를 연결하는 코드를 추가하였다. 또, 오류를 방지하기 위해 cifar10_dir의 경로를 절대 경로인 '/content/drive/MyDrive/25-spring-openSWproject/Assignment09/datasets/cifar-10-b'로 변경하였다.

CIFAR 데이터셋을 이용해 구현한 신경망을 테스트한 결과는 아래와 같으며, two_layer_net.ipynb의 코드와 실행 결과를 함께 첨부하였다. 위 경로 설정을 제외하면 원본 코드를 유지했으며, 마크다운에 각 코드가 설명되어 있으므로 다시 설명하지 않고 결과 이미지 전체를 첨부하는 것으로 대신한다.

```
from google.colab import drive
drive.mount('/content/drive')

import sys
sys.path.append('/content/drive/MyDrive/25-spring-openSWproject/Assignment09')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

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.

```
# 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.

```
# 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()
```

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.

```
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:
3.6802720745909845e-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:
1.7985612998927536e-13
```

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:

```
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: 4.447646e-11
W1 max relative error: 3.561318e-09
b1 max relative error: 2.738421e-09
```

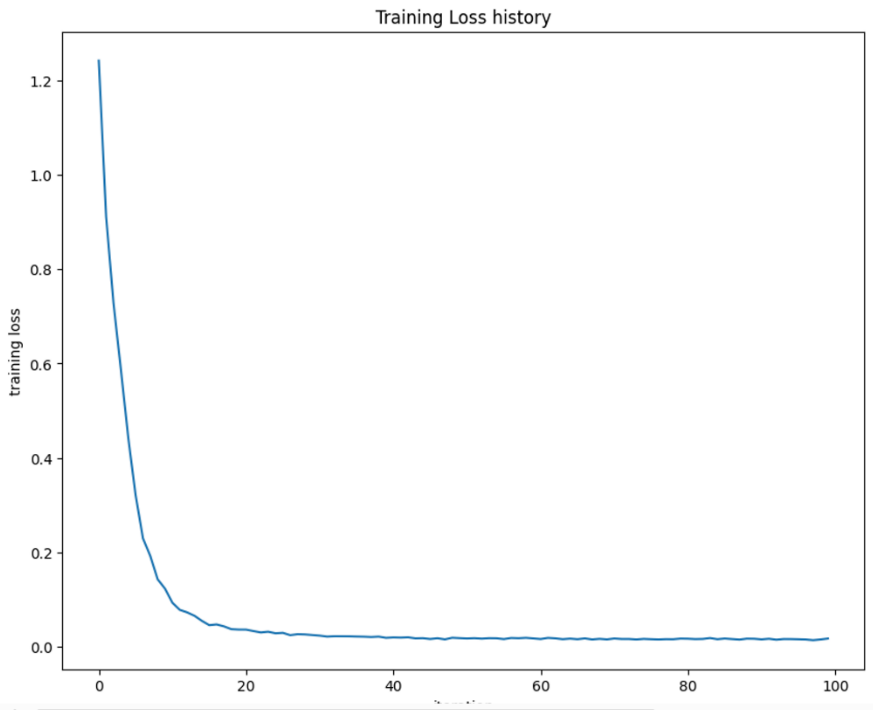
Train the network

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.

```
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:  0.017149607938732048
```



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.

```
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'
    cifar10_dir = '/content/drive/MyDrive/25-spring-openSWproject/Assignment09/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,)
```

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.

```
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 2.302984
iteration 100 / 1000: loss 2.302664
iteration 200 / 1000: loss 2.299105
iteration 300 / 1000: loss 2.278315
iteration 400 / 1000: loss 2.214040
iteration 500 / 1000: loss 2.135265
iteration 600 / 1000: loss 2.071108
iteration 700 / 1000: loss 2.081200
iteration 800 / 1000: loss 2.075023
iteration 900 / 1000: loss 2.008127
Validation accuracy:  0.241
```

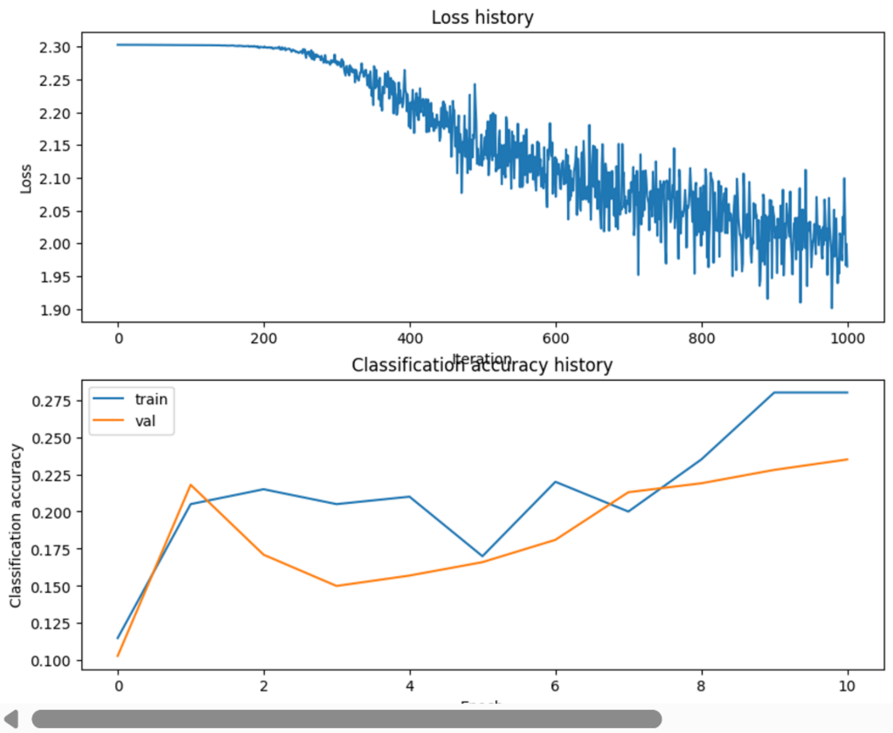
Debug the training

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.

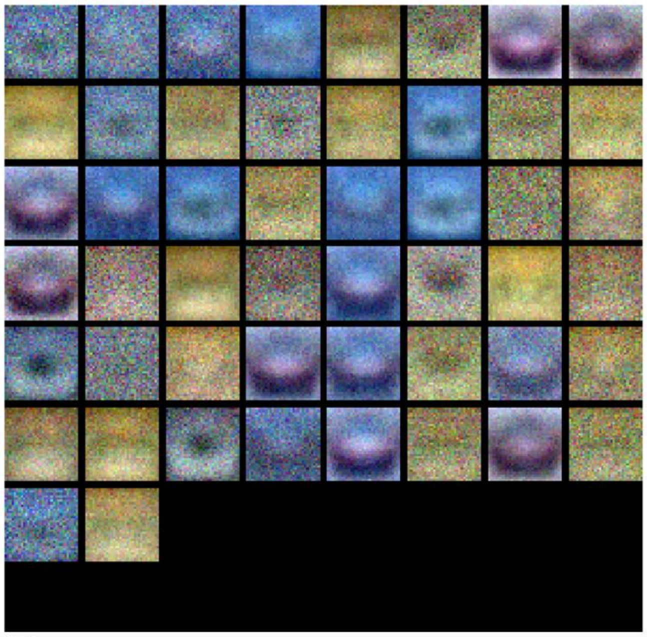
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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.

```
# 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)
```



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.).

Explain your hyperparameter tuning process below.

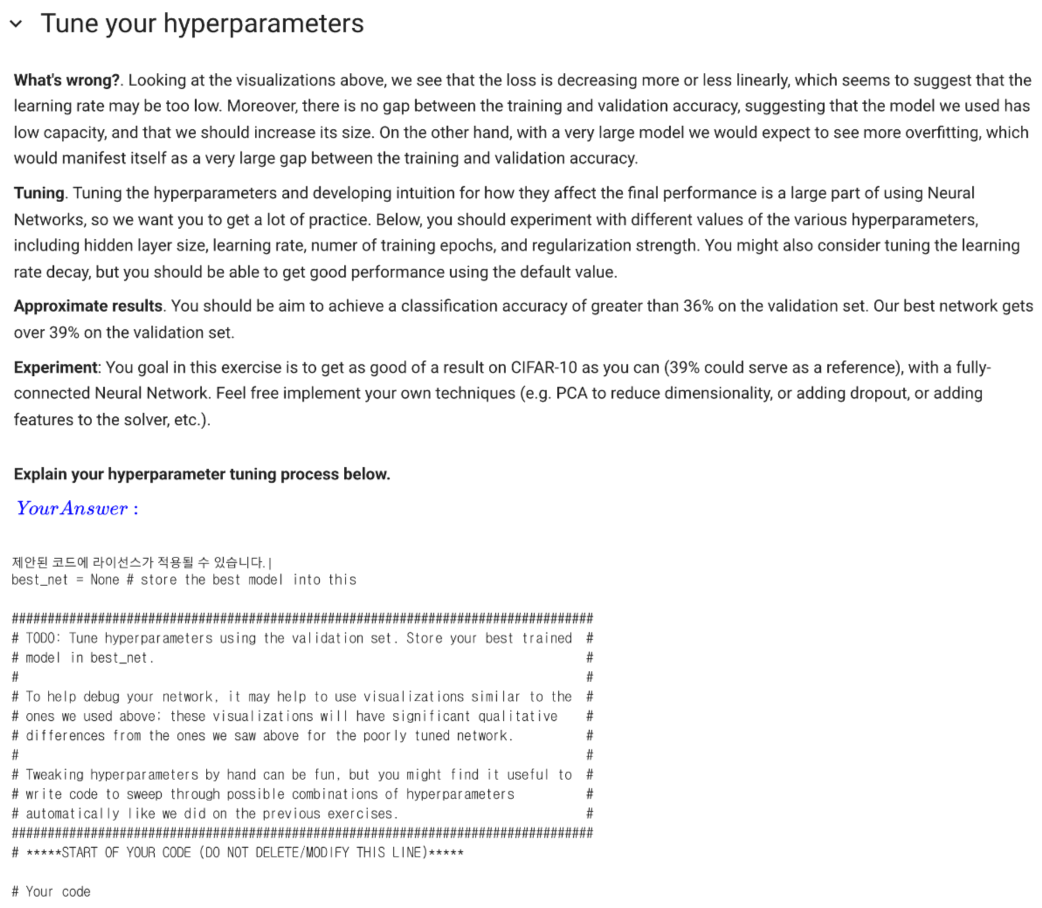
$Your Answer :$

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

# Your code
```

Forward pass: compute scores에서 목표 오차 1e-7보다 작은 약 3.7e-8, Forward pass: compute loss에서 목표 1e-12보다 작은 약 1.8e-13의 오차가 계산되었으며 Backward pass에서 W2, b2, W1, b1 오차 또한 각각 약 3.4e-9, 4.4e-11, 3.6e-9, 2.7e-9로 목표 오차 1e-8보다 작았다. 따라서 neural_net.py의 loss()가 적절히 구현된 것을 확인할 수 있다. Train the network에서도 훈련 손실이 약 0.017로 목표인 0.02보다 작으므로 train() 함수 또한 올바르게 동작한다.

그러나 CIFAR-10 데이터셋으로 훈련한 결과 정확도가 약 24%로 매우 낮았으며, 손실 곡선과 훈련 데이터셋, vaildation 데이터셋의 정확도 그래프를 확인하면 손실이 느리게 감소하고 훈련 데이터 정확도와 validation 데이터 정확도 사이 큰 차이가 없었다. 이를 통해 train() 호출 시 학습률을 지나치게 낮게 설정했고 모델 크기 또한 작았기 때문에 이러한 문제가 발생했음을 추론할 수 있다.

이 문제를 해결하기 위해 하이퍼파라미터를 조정하며, 은닉층 크기와 학습률, L2 정규화 강도를 다양하게 설정하고 이들을 적절히 조합하여 가장 좋은 성능을 가지는 모델을 결정한다. 기존 호출 시 은닉층 크기=50, 학습률=1e-4, L2 정규화 강도=0.25였으므로 은닉층 크기 후보는 100, 150으로, 학습률 후보는 1e-3, 3e-3으로, L2 정규화 강도는 0.1과 0.5로 하여 이들을 hidden_sizes, learning_rates, regs 배열에 저장한 뒤 3중 for문으로 8개의 조합을 만들어 테스트한다. 전체 테스트 코드와 결과는 아래와 같다.



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.).

Explain your hyperparameter tuning process below.

$Your Answer :$

```
제안된 코드에 라이선스가 적용될 수 있습니다.|
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.                          #
#################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# Your code
```

```
'''
기존 설정
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)
'''

best_val_acc = -np.inf  # 최고 정확도

input_size = 32 * 32 * 3

hidden_sizes = [100, 150]     # 은닉층 증가
learning_rates = [1e-3, 3e-3] # 학습률 변경
regs = [0.1, 0.5]             # L2 정규화
num_classes = 10

for h in hidden_sizes:
    for l in learning_rates:
      for r in regs:
          net = TwoLayerNet(input_size, h, num_classes)

          stats = net.train(X_train, y_train, X_val, y_val,
                            num_iters=1000, batch_size=200,
                            learning_rate=l, learning_rate_decay=0.95,
                            reg=r, verbose=True)

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

          if val_acc > best_val_acc:
              best_val_acc = val_acc
              best_net = net

          print(f'Hidden size: {h}, Learning rate: {l}, Regularization strength: {r}, Validation accuracy: {val_acc:.4f}')


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

```
iteration 0 / 1000: loss 2.302863
iteration 100 / 1000: loss 1.950277
iteration 200 / 1000: loss 1.809036
iteration 300 / 1000: loss 1.713532
iteration 400 / 1000: loss 1.586785
iteration 500 / 1000: loss 1.455712
iteration 600 / 1000: loss 1.502838
iteration 700 / 1000: loss 1.396357
iteration 800 / 1000: loss 1.443033
iteration 900 / 1000: loss 1.408119
Hidden size: 100, Learning rate: 0.001, Regularization strength: 0.1, Validation accuracy: 0.4510
iteration 0 / 1000: loss 2.304108
iteration 100 / 1000: loss 1.969180
iteration 200 / 1000: loss 1.759264
iteration 300 / 1000: loss 1.788179
iteration 400 / 1000: loss 1.712420
iteration 500 / 1000: loss 1.656601
iteration 600 / 1000: loss 1.573993
iteration 700 / 1000: loss 1.581476
iteration 800 / 1000: loss 1.510320
iteration 900 / 1000: loss 1.614684
Hidden size: 100, Learning rate: 0.001, Regularization strength: 0.5, Validation accuracy: 0.4410
iteration 0 / 1000: loss 2.302896
iteration 100 / 1000: loss 1.804546
iteration 200 / 1000: loss 1.646184
iteration 300 / 1000: loss 1.514177
iteration 400 / 1000: loss 1.532872
iteration 500 / 1000: loss 1.631565
iteration 600 / 1000: loss 1.399923
iteration 700 / 1000: loss 1.539778
iteration 800 / 1000: loss 1.375952
iteration 900 / 1000: loss 1.380553
Hidden size: 100, Learning rate: 0.003, Regularization strength: 0.1, Validation accuracy: 0.4330
iteration 0 / 1000: loss 2.304127
iteration 100 / 1000: loss 1.719146
iteration 200 / 1000: loss 1.619190
iteration 300 / 1000: loss 1.748378
iteration 400 / 1000: loss 1.659286
iteration 500 / 1000: loss 1.519856
iteration 600 / 1000: loss 1.664643
iteration 700 / 1000: loss 1.625185
```

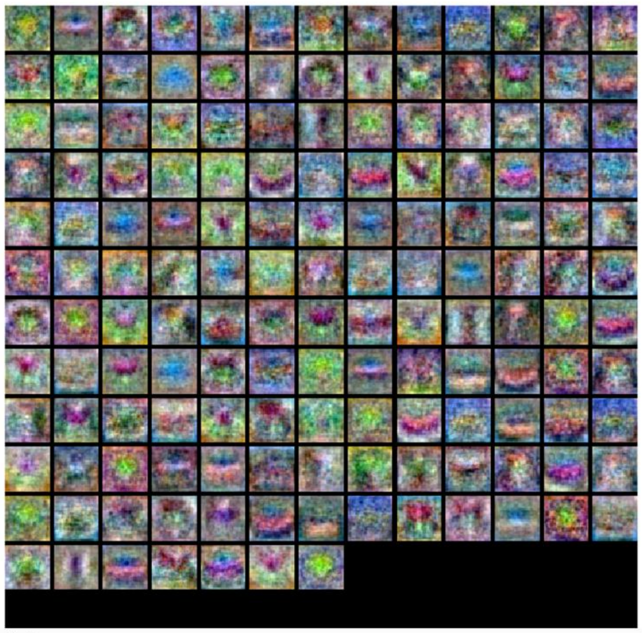
전체 테스트 결과(이미지에 누락된 부분이 있어 별도로 첨부)

```
iteration 0 / 1000: loss 2.302863
iteration 100 / 1000: loss 1.950277
iteration 200 / 1000: loss 1.809036
iteration 300 / 1000: loss 1.713532
iteration 400 / 1000: loss 1.586785
iteration 500 / 1000: loss 1.455712
iteration 600 / 1000: loss 1.502838
iteration 700 / 1000: loss 1.396357
iteration 800 / 1000: loss 1.443033
iteration 900 / 1000: loss 1.408119
Hidden size: 100, Learning rate: 0.001, Regularization strength: 0.1, Validation accuracy:
0.4510
iteration 0 / 1000: loss 2.304108
iteration 100 / 1000: loss 1.969180
iteration 200 / 1000: loss 1.759264
iteration 300 / 1000: loss 1.788179
iteration 400 / 1000: loss 1.712420
iteration 500 / 1000: loss 1.656601
iteration 600 / 1000: loss 1.573993
iteration 700 / 1000: loss 1.581476
iteration 800 / 1000: loss 1.510320
iteration 900 / 1000: loss 1.614684
Hidden size: 100, Learning rate: 0.001, Regularization strength: 0.5, Validation accuracy:
0.4410
iteration 0 / 1000: loss 2.302896
iteration 100 / 1000: loss 1.804546
iteration 200 / 1000: loss 1.646184
iteration 300 / 1000: loss 1.514177
iteration 400 / 1000: loss 1.532872
iteration 500 / 1000: loss 1.631565
iteration 600 / 1000: loss 1.399923
iteration 700 / 1000: loss 1.539778
iteration 800 / 1000: loss 1.375952
iteration 900 / 1000: loss 1.380553
Hidden size: 100, Learning rate: 0.003, Regularization strength: 0.1, Validation accuracy:
0.4330
iteration 0 / 1000: loss 2.304127
iteration 100 / 1000: loss 1.719146
iteration 200 / 1000: loss 1.619190
iteration 300 / 1000: loss 1.748378
iteration 400 / 1000: loss 1.659286
iteration 500 / 1000: loss 1.519856
iteration 600 / 1000: loss 1.664643
iteration 700 / 1000: loss 1.625185
iteration 800 / 1000: loss 1.542223
iteration 900 / 1000: loss 1.555276
Hidden size: 100, Learning rate: 0.003, Regularization strength: 0.5, Validation accuracy:
```

```
0.4150
iteration 0 / 1000: loss 2.303044
iteration 100 / 1000: loss 1.844370
iteration 200 / 1000: loss 1.677221
iteration 300 / 1000: loss 1.644546
iteration 400 / 1000: loss 1.614431
iteration 500 / 1000: loss 1.510503
iteration 600 / 1000: loss 1.467470
iteration 700 / 1000: loss 1.443159
iteration 800 / 1000: loss 1.468036
iteration 900 / 1000: loss 1.373881
Hidden size: 150, Learning rate: 0.001, Regularization strength: 0.1, Validation accuracy:
0.4510
iteration 0 / 1000: loss 2.304899
iteration 100 / 1000: loss 1.973487
iteration 200 / 1000: loss 1.819147
iteration 300 / 1000: loss 1.840234
iteration 400 / 1000: loss 1.626568
iteration 500 / 1000: loss 1.669651
iteration 600 / 1000: loss 1.670260
iteration 700 / 1000: loss 1.528695
iteration 800 / 1000: loss 1.474362
iteration 900 / 1000: loss 1.555569
Hidden size: 150, Learning rate: 0.001, Regularization strength: 0.5, Validation accuracy:
0.4610 <- best_net
iteration 0 / 1000: loss 2.302991
iteration 100 / 1000: loss 1.726525
iteration 200 / 1000: loss 1.793733
iteration 300 / 1000: loss 1.690658
iteration 400 / 1000: loss 1.534091
iteration 500 / 1000: loss 1.353477
iteration 600 / 1000: loss 1.525393
iteration 700 / 1000: loss 1.539864
iteration 800 / 1000: loss 1.574362
iteration 900 / 1000: loss 1.416311
Hidden size: 150, Learning rate: 0.003, Regularization strength: 0.1, Validation accuracy:
0.4310
iteration 0 / 1000: loss 2.304899
iteration 100 / 1000: loss 1.704497
iteration 200 / 1000: loss 1.745501
iteration 300 / 1000: loss 1.623663
iteration 400 / 1000: loss 1.663855
iteration 500 / 1000: loss 1.622108
iteration 600 / 1000: loss 1.602889
iteration 700 / 1000: loss 1.517640
iteration 800 / 1000: loss 1.469133
iteration 900 / 1000: loss 1.599080
Hidden size: 150, Learning rate: 0.003, Regularization strength: 0.5, Validation accuracy:
0.4330
```

```
iteration 800 / 1000: loss 1.542223
iteration 900 / 1000: loss 1.555276
Hidden size: 100, Learning rate: 0.003, Regularization strength: 0.5, Validation accuracy: 0.4150
iteration 0 / 1000: loss 2.303044
iteration 100 / 1000: loss 1.844370
iteration 200 / 1000: loss 1.677221
iteration 300 / 1000: loss 1.644546
iteration 400 / 1000: loss 1.614431
iteration 500 / 1000: loss 1.510503
iteration 600 / 1000: loss 1.467470
iteration 700 / 1000: loss 1.443159
iteration 800 / 1000: loss 1.468036
iteration 900 / 1000: loss 1.373881
Hidden size: 150, Learning rate: 0.001, Regularization strength: 0.1, Validation accuracy: 0.4510
iteration 0 / 1000: loss 2.304899
iteration 100 / 1000: loss 1.973487
iteration 200 / 1000: loss 1.819147
```

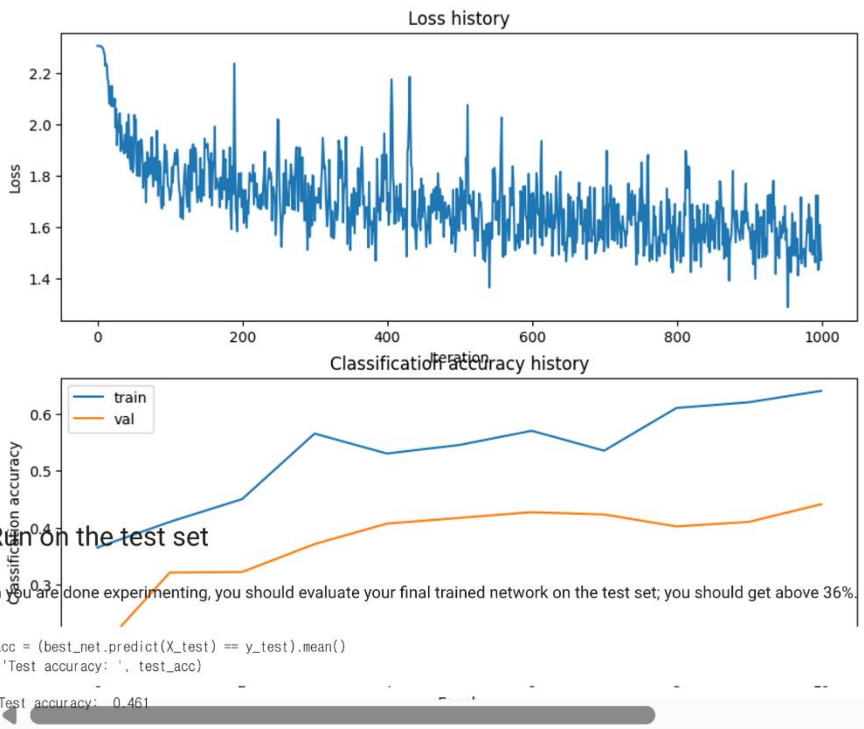
```
# visualize the weights of the best network
show_net_weights(best_net)
```



for문이 종료되면 가장 우수한 모델이 best_net에 저장되며, 이후 best_net의 가중치를 시각화해 출력한다.

위 계산 결과에서 빨간색으로 표시한 부분이 가장 우수한 모델인 best_net의 결과이며, 은닉층 크기가 150, 학습률이 0.001(=1e-3), L2 정규화 강도가 0.5인 모델이 validation 데이터셋에 대해 46%로 가장 높은 정확도를 보였다. 가중치 시각화 결과 또한 첫 번째 실행 결과보다 뚜렷하고 세밀하며, best_net의 손실 함수 그래프와 훈련/validation 정확도 그래프를 출력하면 다음과 같다.

```
# 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()
```



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%.

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

```
Test accuracy:  0.461
```

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.

$Your Answer:$

$Your Explanation:$

이전 테스트 결과에서 훈련 초반에 학습이 정체되었던 것과 달리 전 구간에서 손실 함수가 꾸준히 감소하며, 훈련/validation 정확도 차이가 존재한다. 테스트 데이터셋으로 예측한 결과 모델이 약 46%의 정확도로 이미지를 분류하는 것을 확인할 수 있다.

Inline Quesiton 란의 정답은 1번과 3번이며, 은닉층을 추가해 모델을 더욱 크게(=deeper) 만들 경우 가중치 개수가 커지고 모델이 복잡해져 오히려 과적합이 심화되므로 2번은 틀린 답이다.