실습3: two layer net

설정

```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'A11/'
FOLDERNAME = None
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/A11/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from A11.classifiers.fc_net import *
from A11.data_utils import get_CIFAR10_data
from A11.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
from A11.solver import Solver

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

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

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

```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in list(data.items()):
  print(('%s: ' % k, v.shape))
```

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

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

  return out, cache
```

```python
def layer_backward(dout, cache):
  """
  Receive dout (derivative of loss with respect to outputs) and cache,
  and compute derivative with respect to inputs.
  """
  # Unpack cache values
  x, w, z, out = cache

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

  return dx, dw
```

구현1: affine_forward (layers.py)

Affine layer: forward

Open the file `AI/layers.py` and implement the `affine_forward` function.

Once you are done you can test your implementaion by running the following:

```
[ ]  # Test the affine_forward function

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

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

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

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

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

구현1: affine_forward (layers.py)

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

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

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

    Returns a tuple of:
    - out: output, of shape (N, M)
    - cache: (x, w, b)
    """
    out = None
    ###########################################################################
    # TODO: Implement the affine forward pass. Store the result in out. You   #
    # will need to reshape the input into rows.                               #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    pass

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    cache = (x, w, b)
    return out, cache
```

구현2: affine_backward (layers.py)

- $Y = WX$
- $dW = dY \cdot X^T$
- $dX = W^T \cdot dY$

Affine layer: backward

Now implement the `affine_backward` function and test your implementation using numeric gradient checking.

```
[ ]  # Test the affine_backward function
     np.random.seed(231)
     x = np.random.randn(10, 2, 3)
     w = np.random.randn(6, 5)
     b = np.random.randn(5)
     dout = np.random.randn(10, 5)

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

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

     # The error should be around e-10 or less
     print('Testing affine_backward function:')
     print('dx error: ', rel_error(dx_num, dx))
     print('dw error: ', rel_error(dw_num, dw))
     print('db error: ', rel_error(db_num, db))
```

구현2: affine_backward (layers.py)

- x': x를 재배열한 것
- x': (N,D), w: (D,M), b: (M,), out: (N,M)
- out = x'w + b
- $dx = dout \cdot w^T$
  - dx는 이후 (N, d1, ..., dk) 형태로 재배열
- $dw = x'^T \cdot dout$
- db는 $dout$의 행들을 합한 것

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

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

    Returns a tuple of:
    - dx: Gradient with respect to x, of shape (N, d1, ..., d_k)
    - dw: Gradient with respect to w, of shape (D, M)
    - db: Gradient with respect to b, of shape (M,)
    """
    x, w, b = cache
    dx, dw, db = None, None, None
    ###########################################################################
    # TODO: Implement the affine backward pass.                               #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    pass

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    return dx, dw, db
```

구현3: relu_forward (layers.py)

ReLU activation: forward

Implement the forward pass for the ReLU activation function in the `relu_forward` function and test your implementation using the following:

```
[ ]  # Test the relu_forward function

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

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

     # Compare your output with ours. The error should be on the order of e-8
     print('Testing relu_forward function:')
     print('difference: ', rel_error(out, correct_out))
```

구현3: relu_forward (layers.py)

- ReLU(x) = max(0, x)

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

    Input:
    - x: Inputs, of any shape

    Returns a tuple of:
    - out: Output, of the same shape as x
    - cache: x
    """
    out = None
    ###########################################################################
    # TODO: Implement the ReLU forward pass.                                  #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    pass

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    cache = x
    return out, cache
```

구현4: relu_backward (layers.py)

ReLU activation: backward

Now implement the backward pass for the ReLU activation function in the `relu_backward` function and test your implementation using numeric gradient checking:

```
[ ]  np.random.seed(231)
     x = np.random.randn(10, 10)
     dout = np.random.randn(*x.shape)

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

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

     # The error should be on the order of e-12
     print('Testing relu_backward function:')
     print('dx error: ', rel_error(dx_num, dx))
```

구현4: relu_backward (layers.py)

- y=ReLU(x) = max(0,x)
- x>0이면 dx = dy
- x<0이면 dx = 0

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

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

    Returns:
    - dx: Gradient with respect to x
    """
    dx, x = None, cache
    ###########################################################################
    # TODO: Implement the ReLU backward pass.                                 #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    pass

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

affine_relu_forward, affine_relu_backward (layers_utils.py)

"Sandwich" layers

There are some common patterns of layers that are frequently used in neural nets. For example, affine layers are frequently followed by a ReLU nonlinearity. To make these common patterns easy, we define several convenience layers in the file `cs231n/layer_utils.py`.

For now take a look at the `affine_relu_forward` and `affine_relu_backward` functions, and run the following to numerically gradient check the backward pass:

```
[ ]  from cs231n.layer_utils import affine_relu_forward, affine_relu_backward
     np.random.seed(231)
     x = np.random.randn(2, 3, 4)
     w = np.random.randn(12, 10)
     b = np.random.randn(10)
     dout = np.random.randn(2, 10)

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

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

     # Relative error should be around e-10 or less
     print('Testing affine_relu_forward and affine_relu_backward:')
     print('dx error: ', rel_error(dx_num, dx))
     print('dw error: ', rel_error(dw_num, dw))
     print('db error: ', rel_error(db_num, db))
```

svm_loss, softmax_loss (layers.py)

Loss layers: Softmax and SVM

Now implement the loss and gradient for softmax and SVM in the `softmax_loss` and `svm_loss` function in `cs231n/layers.py`. These should be similar to what you implemented in `cs231n/classifiers/softmax.py` and `cs231n/classifiers/linear_svm.py`.

You can make sure that the implementations are correct by running the following:

```
[ ]  np.random.seed(231)
     num_classes, num_inputs = 10, 50
     x = 0.001 * np.random.randn(num_inputs, num_classes)
     y = np.random.randint(num_classes, size=num_inputs)

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

     # Test svm_loss function. Loss should be around 9 and dx error should be around the order of e-9
     print('Testing svm_loss:')
     print('loss: ', loss)
     print('dx error: ', rel_error(dx_num, dx))

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

     # Test softmax_loss function. Loss should be close to 2.3 and dx error should be around e-8
     print('\nTesting softmax_loss:')
     print('loss: ', loss)
     print('dx error: ', rel_error(dx_num, dx))
```

구현5: TwoLayerNet 클래스 (fc_net.py)

Two-layer network

Open the file `AI/classifiers/fc_net.py` and complete the implementation of the `TwoLaye` understand the API. You can run the cell below to test your implementation.

```
[ ]  np.random.seed(231)
     N, D, H, C = 3, 5, 50, 7
     X = np.random.randn(N, D)
     y = np.random.randint(C, size=N)

     std = 1e-3
     model = TwoLayerNet(input_dim=D, hidden_dim=H, num_classes=C, weight_scale=std)

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

     print('Testing test-time forward pass ... ')
     model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
     model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
     model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
     model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
     X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
     scores = model.loss(X)
     correct_scores = np.asarray(
       [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434, 15.33206765,  16.09215096],
        [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.49994135,  16.18839143],
        [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.66781506,  16.2846319 ]])
     scores_diff = np.abs(scores - correct_scores).sum()
     assert scores_diff < 1e-6, 'Problem with test-time forward pass'

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

     model.reg = 1.0
     loss, grads = model.loss(X, y)
     correct_loss = 26.5948426952
     assert abs(loss - correct_loss) < 1e-10, 'Problem with regularization loss'

     # Errors should be around e-7 or less
     for reg in [0.0, 0.7]:
       print('Running numeric gradient check with reg = ', reg)
       model.reg = reg
       loss, grads = model.loss(X, y)

       for name in sorted(grads):
         f = lambda _: model.loss(X, y)[0]
         grad_num = eval_numerical_gradient(f, model.params[name], verbose=False)
         print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

구현5: TwoLayerNet 클래스 (fc_net.py)

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

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

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

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

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

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

        ############################################################################
        # TODO: Initialize the weights and biases of the two-layer net. Weights    #
        # should be initialized from a Gaussian centered at 0.0 with               #
        # standard deviation equal to weight_scale, and biases should be           #
        # initialized to zero. All weights and biases should be stored in the      #
        # dictionary self.params, with first layer weights                         #
        # and biases using the keys 'W1' and 'b1' and second layer                 #
        # weights and biases using the keys 'W2' and 'b2'.                         #
        ############################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        pass

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

- self.params['W1'], self.params['W2']
  - 작은 랜덤으로 초기화 (random.randn 함수 활용)
- self.params['b1'], self.params['b2']
  - 0으로 초기화

구현5: TwoLayerNet 클래스 (fc_net.py)

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

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

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

    If y is not None, then run a training-time forward and backward pass and
    return a tuple of:
    - loss: Scalar value giving the loss
    - grads: Dictionary with the same keys as self.params, mapping parameter
      names to gradients of the loss with respect to those parameters.
    """
    scores = None
    ############################################################################
    # TODO: Implement the forward pass for the two-layer net, computing the    #
    # class scores for X and storing them in the scores variable.              #
    ############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    pass

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

- TwoLayerNet: affine, relu, affine으로 구성됨
- 첫 번째 pass: affine_forward, relu_forward 활용
- 2번째 pass: affine_backward, relu_backward, softmax_loss 함수 활용

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

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

    loss, grads = 0, {}
    ############################################################################
    # TODO: Implement the backward pass for the two-layer net. Store the loss  #
    # in the loss variable and gradients in the grads dictionary. Compute data #
    # loss using softmax, and make sure that grads[k] holds the gradients for  #
    # self.params[k]. Don't forget to add L2 regularization!                   #
    #                                                                          #
    # NOTE: To ensure that your implementation matches ours and you pass the   #
    # automated tests, make sure that your L2 regularization includes a factor #
    # of 0.5 to simplify the expression for the gradient.                      #
    ############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    pass

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

    return loss, grads
```

구현6: Solver 사용

- Solver 클래스를 활용하여 TwoLayerNet 학습

Solver

Open the file `AI/solver.py` and read through it to familiarize yourself with the API. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves about `36%` accuracy on the validation set.

```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
model = TwoLayerNet(input_size, hidden_size, num_classes)
solver = None

################################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves about 36% #
# accuracy on the validation set.                                              #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

pass

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

구현6: Solver 사용

```
data = {
  'X_train': # training data
  'y_train': # training labels
  'X_val': # validation data
  'y_val': # validation labels
}
model = MyAwesomeModel(hidden_size=100, reg=10)
solver = Solver(model, data,
                update_rule='sgd',
                optim_config={
                  'learning_rate': 1e-4,
                },
                lr_decay=0.95,
                num_epochs=5, batch_size=200,
                print_every=100)
solver.train()
```

구현6: Solver 사용

Solver

Open the file `AI/solver.py` and read through it to familiarize yourself with the API. After doing so, use a `Solver` instance to train a `TwoLayerNet` that achieves about `36%` accuracy on the validation set.

```
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
model = TwoLayerNet(input_size, hidden_size, num_classes)
solver = None

################################################################################
# TODO: Use a Solver instance to train a TwoLayerNet that achieves about 36% #
# accuracy on the validation set.                                              #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

pass

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

훈련 디버깅

Debug the training

With the default parameters we provided above, you should get a validation accuracy of about 0.36 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.

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

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

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

시각화

```
from AI.vis_utils import visualize_grid

# Visualize the weights of the network

def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.reshape(3, 32, 32, -1).transpose(3, 1, 2, 0)
    plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
    plt.gca().axis('off')
    plt.show()

show_net_weights(model)
```

구현7: 하이퍼파라미터 튜닝

- 아직까지는 좋은 성능을 달성하지 못함
- 하이퍼파라미터를 조정하면서 그것이 최종 성능에 어떻게 영향을 미치는지에 대한 직관을 얻는 연습을 할 것
- 은닉층 크기, 학습률, 훈련 에폭 수, 정규화 강도 등 다양한 하이퍼파라미터의 값을 다르게 설정하여 실험해야함
- 학습률 감쇠(learning rate decay)을 조절하는 것도 고려해볼 수 있지만, 기본 값을 사용하여도 좋은 성능을 얻을 수 있어야함
- 검증 세트에서 48% 이상의 분류 정확도 달성이 권장됨
- 이 연습에서의 목표는 완전연결신경망을 사용하여 CIFAR-10에서 최대한 좋은 결과를 얻는 것

구현7: 하이퍼파라미터 튜닝

```
best_model = None


################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained  #
# model in best_model.                                                          #
#                                                                               #
# 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 thexs previous exercises.                          #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

pass

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

테스트

Test your model!

Run your best model on the validation and test sets. You should achieve above 48% accuracy on the validation set and the test set.

```
[ ]  y_val_pred = np.argmax(best_model.loss(data['X_val']), axis=1)
     print('Validation set accuracy: ', (y_val_pred == data['y_val']).mean())
```

```
[ ]  y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
     print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

과제3

- 고정 파라미터
  - batch_size: 200
  - lr_decay: 0.95
- 조정하는 파라미터
  - hidden_size
  - learning_rate
  - num_epochs
  - reg

과제3

hidden_size	learning_rate	lr_decay	num_epochs	reg	훈련정확도	검증정확도	결과 분석 및 고찰
50	1.00E-04	0.95	5	0	?	?	?
200	1.00E-04	0.95	5	0	?	?	?
				...			

과제3

- 그리디 알고리즘(Greedy algorithm)
  - 그리디 알고리즘은 최적의 값을 구해야하는 상황에서 사용되는 근시안적인 방법론
  - 각 단계에서 최적이라고 생각되는 것을 선택해 나가는 방식으로 진행하여 최종적인 해답에 도달하는 알고리즘
  - 최적의 답을 얻는다는 보장은 없지만, 어느 정도 최적에 근사한 값을 빠르게 도달할 수 있는 장점이 있음

과제3

- 에폭 수
  - 에폭이 많으면 시간이 많이 걸린다는 단점이 있음
  - 또한, 과적합 되는 경우도 있음
  - 과적합이 되는 것 같으면 더 이상 반복을 하지 않는 것이 좋고, 반면 과적합은 안 되면서 손실 값이 덜 수렴 된 것으로 판단되면 더 반복을 해야할 수 있음
- 은닉 계층 수
  - 은닉계층 수는 기본적으로 크게 늘리는 것이 좋음. 특히, 과적합이 나타나지 않으면 더욱 그럼
  - 과적합이 나타나면 정규화 강도 높이는 것을 고려해야할 수 있음
  - 은닉계층 수가 커지면 시간이 많이 걸린다는 단점이 있음

과제3

- 학습률
  - 학습률이 작으면 좋기는 하지만 그만큼 반복이 훨씬 많이 필요할 것이므로 적절한 값 설정 필요
  - 지그재그인 것 같으면 작게 하는 것이 좋음
  - 로그 스케일로 여러 값을 시도해보는 것이 좋음
    - ex) 0.001, 0.01, 0.1, 1
  - 초기에는 큰 학습률로 시작하여 점차 작은 값으로 줄여가면서 실험하는 것이 좋음
- 정규화 강도
  - 과적합 방지에 도움이 됨
  - 과적합이 발생 안 하더라도 성능을 더 높일 수도 있음
  - 로그 스케일로 여러 값 시도하는 것이 좋음
    - ex) 0.0001, 0.001, 0.01, 0.1, 0.1