Neural Networks

July 11, 2022

In the Name of God

Sharif University of Technology - Departmenet of Computer Engineering

Artifical Intelligence - Dr. Mohammad Hossein Rohban Spring 2022

```
<font color=red size=6>
    <br />
```

Practical Assignment 6 Neural Networks Cheating is Strongly Prohibited Please run all the cells.

1 Personal Data

[ ]:
```
# Set your student number
student_number = 99102401
Name = 'Ali'
Last_Name = 'Nazari'
```

2 Rules

- You are not allowed to add or remove cells. You **must use the provided space to write your code**. If you don't follow this rule, **your Practical Assignment won't be graded**.
- You **are** allowed to use **for loops** only in the implementation of the **FullyConnectedNet** class.

[1]:
```
!pip install future
!pip install pandas
!pip install torchvision
```

```
Requirement already satisfied: future in f:\anaconda\installed\lib\site-packages
(0.18.2)
Requirement already satisfied: pandas in f:\anaconda\installed\lib\site-packages
(1.2.4)
Requirement already satisfied: python-dateutil>=2.7.3 in
f:\anaconda\installed\lib\site-packages (from pandas) (2.8.1)
Requirement already satisfied: pytz>=2017.3 in f:\anaconda\installed\lib\site-
packages (from pandas) (2021.1)
Requirement already satisfied: numpy>=1.16.5 in f:\anaconda\installed\lib\site-
packages (from pandas) (1.20.1)
```

```
Requirement already satisfied: six>=1.5 in f:\anaconda\installed\lib\site-
packages (from python-dateutil>=2.7.3->pandas) (1.15.0)
Requirement already satisfied: torchvision in f:\anaconda\installed\lib\site-
packages (0.13.0)
Requirement already satisfied: requests in f:\anaconda\installed\lib\site-
packages (from torchvision) (2.25.1)
Requirement already satisfied: torch==1.12.0 in f:\anaconda\installed\lib\site-
packages (from torchvision) (1.12.0)
Requirement already satisfied: typing-extensions in
f:\anaconda\installed\lib\site-packages (from torchvision) (3.7.4.3)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in
f:\anaconda\installed\lib\site-packages (from torchvision) (8.2.0)
Requirement already satisfied: numpy in f:\anaconda\installed\lib\site-packages
(from torchvision) (1.20.1)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in
f:\anaconda\installed\lib\site-packages (from requests->torchvision) (1.26.4)
Requirement already satisfied: certifi>=2017.4.17 in
f:\anaconda\installed\lib\site-packages (from requests->torchvision) (2020.12.5)
Requirement already satisfied: chardet<5,>=3.0.2 in
f:\anaconda\installed\lib\site-packages (from requests->torchvision) (4.0.0)
Requirement already satisfied: idna<3,>=2.5 in f:\anaconda\installed\lib\site-
packages (from requests->torchvision) (2.10)
```

[6]:
```
from Helper_codes.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from Helper_codes.MNIST_data import get_MNIST_data, get_normalized_MNIST_data
from builtins import range
import numpy as np
import matplotlib.pyplot as plt
from Helper_codes.solver import *
import pandas as pd
from sklearn.datasets import fetch_california_housing
from Helper_codes.california_housing import *


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

def print_mean_std(x,axis=0):
    print(f"  means: {x.mean(axis=axis)}")
    print(f"  stds:  {x.std(axis=axis)}\n")
```

3 Fully-Connected Neural Nets

In this exercise we will implement fully-connected networks using a modular approach. For each layer we will implement a `forward` and a `backward` function. The `forward` function will receive inputs, weights, and other parameters and will return both an output and a `cache` object storing data needed for the backward pass, like this:

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

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

  return out, cache
```

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

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

After implementing a bunch of layers this way, we will be able to easily combine them to build classifiers with different architectures.

4 Affine layer: forward

Implement the `affine_forward` function.

[7]:
```
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)
    """
    ###########################################################################
    # 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)*****
    modified_x = x.reshape(x.shape[0], -1) # to make x into rows
    out = modified_x.dot(w) + b

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

You can test your implementaion by running the following:

[8]:
```
# 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))
```

```
Testing affine_forward function:
difference:  9.769847728806635e-10
```

5 Affine layer: backward

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

[9]:
```
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,)
    """
    ###########################################################################
    # TODO: Implement the affine backward pass.                               #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    x, w, b = cache
    dot_dx = dout.dot(w.T)
    dx = dot_dx.reshape(x.shape)
    reshape_dw = x.reshape(x.shape[0], -1)
    transposed_dw = reshape_dw.T
    dw = transposed_dw.dot(dout)
    db = np.sum(dout, axis=0)
    return dx, dw, db
```

[10]:
```
# 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))
```

```
Testing affine_backward function:
dx error:  5.399100368651805e-11
dw error:  9.904211865398145e-11
db error:  2.4122867568119087e-11
```

6 ReLU activation: forward

Implement the forward pass for the ReLU activation function in the `relu_forward` function and test your implementation using numeric gradient checking.

[11]:
```
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
    """
    ############################################################################
    # TODO: Implement the ReLU forward pass.                                   #
    ############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    out = np.maximum(x, 0)
    return out, x
```

[12]:
```
# 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))
```

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

7 ReLU activation: backward

Now implement the backward pass for the ReLU activation function in the `relu_backward` function.

[13]:
```
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
    """
    ###########################################################################
    # TODO: Implement the ReLU backward pass.                                 #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    return dout * (cache > 0)
```

You can test your implementation using numeric gradient checking:

[14]:
```
# Test the relu_backward function
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))
```

```
Testing relu_backward function:
dx error:  3.2756349136310288e-12
```

8 Sigmoid activation: forward

Implement the forward pass for the Sigmoid activation function in the `sigmoid_forward` function and test your implementation using numeric gradient checking.

[15]:
```python
def sigmoid_forward(x):
    """
    Computes the forward pass for a layer of Sigmoid.

    Input:
    - x: Inputs, of any shape

    Returns a tuple of:
    - out: Output, of the same shape as x
    - cache: x
    """
    ###########################################################################
    # TODO: Implement the Sigmoid forward pass.                               #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    out = np.array(1.0 / (1.0 + np.exp(-x)))

    return out,x
```

[16]:
```python
# Test the sigmoid_forward function

x = np.linspace(-6, 6, num=12).reshape(3, 4)

out, _ = sigmoid_forward(x)
correct_out = np.array([[0.00247262, 0.00732514, 0.0214955 , 0.06138311],
                        [0.16296047, 0.36691963, 0.63308037, 0.83703953],
                        [0.93861689, 0.9785045 , 0.99267486, 0.99752738]])

# Compare your output with ours. The error should be on the order of e-7
print('Testing sigmoid_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing sigmoid_forward function:
difference:  6.383174040859927e-07
```

9 Sigmoid activation: backward

Now implement the backward pass for the Sigmoid activation function in the `sigmoid_backward` function.

[17]:
```
def sigmoid_backward(dout, cache):
    """
    Computes the backward pass for a layer of Sigmoid.

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

    Returns:
    - dx: Gradient with respect to x
    """
    ###########################################################################
    # TODO: Implement the Sigmoid backward pass.                              #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    # we know that to calculate derivate dout/dnetout we have out(1-out) so we use this approach for calculating this part
    gradian = (np.array(1.0 / (1.0 + np.exp(-cache))))*(1-(np.array(1.0 / (1.0 + np.exp(-cache)))))
    return dout * gradian
```

You can test your implementation using numeric gradient checking:

[18]:
```
# Test the sigmoid_backward function
np.random.seed(231)
x = np.random.randn(10, 10)
dout = np.random.randn(*x.shape)

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

_, cache = sigmoid_forward(x)
dx = sigmoid_backward(dout, cache)

# The error should be on the order of e-11
print('Testing sigmoid_backward function:')
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing sigmoid_backward function:
dx error:  3.446520386706568e-11
```

10 "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. Implement the forward and backward pass for the affine layer followed by a ReLU nonlinearity in the `affine_relu_forward` and `affine_relu_backward` functions.

[19]:
```
def affine_relu_forward(x, w, b):
    """
    Convenience layer that performs an affine transform followed by a ReLU

    Inputs:
    - x: Input to the affine layer
    - w, b: Weights for the affine layer

    Returns a tuple of:
    - out: Output from the ReLU
    - cache: Object to give to the backward pass
    """
    ############################################################################
    # TODO: Implement the affine-RELU forward pass.                            #
    ############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    # first we use our affine_forward function, because in explenation we have␣
 ↪this
    first_out, first_cache = affine_forward(x, w, b)
    # then we use relu_forward to get main output
    main_out, main_cache = relu_forward(first_out)
    return main_out, (first_cache, main_cache)
```

[22]:
```
def affine_relu_backward(dout, cache):
    """
    Backward pass for the affine-relu convenience layer

    Inputs:
    - dout: Upstream derivatives, of any shape
    - cache: (fc_cache, relu_cache)

    Returns a tuple of:
    - dx: Gradient with respect to x
    - dw: Gradient with respect to w
    - db: Gradient with respect to b
    """
    ############################################################################
    # TODO: Implement the affine-RELU backward pass.                           #
    ############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
    relu_backward_parameter = cache[1]
    affine_backward_parameter = cache[0]

    # after using relu_forward in previous section, now we should use␣
↪relu_backward to start propagation
    first_part_output = relu_backward(dout ,relu_backward_parameter)
    # after using affine_forward and relu_forward and relu_backward, we should␣
↪use affine_backward to finish the process
    second_part_output = affine_backward(first_part_output,␣
↪affine_backward_parameter)
    dx, dw, db = second_part_output
    return dx, dw, db
```

You can test your implementation using numeric gradient checking:

[23]:
```
# Test the affine_relu_backward function

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))
```

```
Testing affine_relu_forward and affine_relu_backward:
dx error:  6.750562121603446e-11
dw error:  8.162015570444288e-11
db error:  7.826724021458994e-12
```

11 Batch Normalization: Forward Pass

Implement the batch normalization forward pass in the function `batchnorm_forward`.

[31]:
```
def batchnorm_forward(x, gamma, beta, bn_param):
    """Forward pass for batch normalization.

    During training the sample mean and (uncorrected) sample variance are
    computed from minibatch statistics and used to normalize the incoming data.
    During training we also keep an exponentially decaying running mean of the
    mean and variance of each feature, and these averages are used to normalize
    data at test-time.

    At each timestep we update the running averages for mean and variance using
    an exponential decay based on the momentum parameter:

    running_mean = momentum * running_mean + (1 - momentum) * sample_mean
    running_var = momentum * running_var + (1 - momentum) * sample_var

    Input:
    - x: Data of shape (N, D)
    - gamma: Scale parameter of shape (D,)
    - beta: Shift paremeter of shape (D,)
    - bn_param: Dictionary with the following keys:
      - mode: 'train' or 'test'; required
      - eps: Constant for numeric stability
      - momentum: Constant for running mean / variance.
      - running_mean: Array of shape (D,) giving running mean of features
      - running_var Array of shape (D,) giving running variance of features

    Returns a tuple of:
    - out: of shape (N, D)
    - cache: A tuple of values needed in the backward pass
    """
    mode = bn_param["mode"]
    eps = bn_param.get("eps", 1e-5)
    momentum = bn_param.get("momentum", 0.9)

    N, D = x.shape
    running_mean = bn_param.get("running_mean", np.zeros(D, dtype=x.dtype))
    running_var = bn_param.get("running_var", np.zeros(D, dtype=x.dtype))

    out, cache = None, None
    if mode == "train":
        #############################################################################
        # TODO: Implement the training-time forward pass for batch norm.      #
        # Use minibatch statistics to compute the mean and variance, use      #
        # these statistics to normalize the incoming data, and scale and      #
        # shift the normalized data using gamma and beta.                     #
        #                                                                     #
        # You should store the output in the variable out. Any intermediates  #
```

```
        # that you need for the backward pass should be stored in the cache   #
        # variable.                                                            #
        #                                                                      #
        # You should also use your computed sample mean and variance together #
        # with the momentum variable to update the running mean and running   #
        # variance, storing your result in the running_mean and running_var   #
        # variables.                                                           #
        #                                                                      #
        # Note that though you should be keeping track of the running         #
        # variance, you should normalize the data based on the standard       #
        # deviation (square root of variance) instead!                         #
        ########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        normalized_pow = (x - np.mean(x, axis=0)) ** 2
        standard_deviation = np.sqrt(np.var(x, axis=0) + eps)
        final_normalized_data = (x - np.mean(x, axis=0)) * (1./
↪standard_deviation)
        out = (gamma * final_normalized_data) + beta
        cache = (final_normalized_data, gamma, (x - np.mean(x, axis=0)), (1./
↪standard_deviation), standard_deviation, standard_deviation *␣
↪standard_deviation, eps)
        running_mean = ((1 - momentum) * (np.mean(x, axis=0))) + running_mean *␣
↪momentum
        running_var = ((1 - momentum) * (np.var(x, axis=0))) + running_var *␣
↪momentum

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        ########################################################################
        #                           END OF YOUR CODE                           #
        ########################################################################
    elif mode == "test":
        ########################################################################
        # TODO: Implement the test-time forward pass for batch normalization. #
        # Use the running mean and variance to normalize the incoming data,   #
        # then scale and shift the normalized data using gamma and beta.      #
        # Store the result in the out variable.                                #
        ########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        normalized_mean = x - running_mean
        normalized_var = np.sqrt(running_var + eps)
        out = gamma * (normalized_mean / normalized_var)
        out += beta
        cache = None

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

```
        ##############################################################################
        #                             END OF YOUR CODE                               #
        ##############################################################################
    else:
        raise ValueError('Invalid forward batchnorm mode "%s"' % mode)

    # Store the updated running means back into bn_param
    bn_param["running_mean"] = running_mean
    bn_param["running_var"] = running_var

    return out, cache
```

Run the following to test your implementation.

[33]:
```
# Check the training-time forward pass by checking means and variances
# of features both before and after batch normalization

# Simulate the forward pass for a two-layer network.
np.random.seed(231)
N, D1, D2, D3 = 200, 50, 60, 3
X = np.random.randn(N, D1)
W1 = np.random.randn(D1, D2)
W2 = np.random.randn(D2, D3)
a = np.maximum(0, X.dot(W1)).dot(W2)

print('Before batch normalization:')
print_mean_std(a,axis=0)

gamma = np.ones((D3,))
beta = np.zeros((D3,))

# Means should be close to zero and stds close to one.
print('After batch normalization (gamma=1, beta=0)')
a_norm, _ = batchnorm_forward(a, gamma, beta, {'mode': 'train'})
print_mean_std(a_norm,axis=0)

gamma = np.asarray([1.0, 2.0, 3.0])
beta = np.asarray([11.0, 12.0, 13.0])

# Now means should be close to beta and stds close to gamma.
print('After batch normalization (gamma=', gamma, ', beta=', beta, ')')
a_norm, _ = batchnorm_forward(a, gamma, beta, {'mode': 'train'})
print_mean_std(a_norm,axis=0)
```

```
Before batch normalization:
  means: [ -2.3814598  -13.18038246   1.91780462]
  stds:  [27.18502186 34.21455511 37.68611762]
```

```
After batch normalization (gamma=1, beta=0)
  means: [4.44089210e-17 8.27116153e-17 4.57966998e-17]
  stds:  [0.99999999 1.         1.        ]

After batch normalization (gamma= [1. 2. 3.] , beta= [11. 12. 13.] )
  means: [11. 12. 13.]
  stds:  [0.99999999 1.99999999 2.99999999]
```

[35]:
```python
# Check the test-time forward pass by running the training-time
# forward pass many times to warm up the running averages, and then
# checking the means and variances of activations after a test-time
# forward pass.

np.random.seed(231)
N, D1, D2, D3 = 200, 50, 60, 3
W1 = np.random.randn(D1, D2)
W2 = np.random.randn(D2, D3)

bn_param = {'mode': 'train'}
gamma = np.ones(D3)
beta = np.zeros(D3)

for t in range(50):
    X = np.random.randn(N, D1)
    a = np.maximum(0, X.dot(W1)).dot(W2)
    batchnorm_forward(a, gamma, beta, bn_param)

bn_param['mode'] = 'test'
X = np.random.randn(N, D1)
a = np.maximum(0, X.dot(W1)).dot(W2)
a_norm, _ = batchnorm_forward(a, gamma, beta, bn_param)

# Means should be close to zero and stds close to one, but will be
# noisier than training-time forward passes.
print('After batch normalization (test-time):')
print_mean_std(a_norm,axis=0)
```

```
After batch normalization (test-time):
  means: [-0.03927354 -0.04349152 -0.10452688]
  stds:  [1.01531428 1.01238373 0.97819988]
```

12 Batch Normalization: Backward Pass

Now implement the backward pass for batch normalization in the function `batchnorm_backward`.

In the forward pass, given a set of inputs $X = \begin{bmatrix} x_1 \\ x_2 \\ ... \\ x_N \end{bmatrix}$,

we first calculate the mean μ and variance var. With μ and var calculated, we can calculate the standard deviation σ and normalized data Y. The equations and graph illustration below describe the computation (y_i is the i-th element of the vector Y).

$$\mu = \frac{1}{N}\sum_{k=1}^{N} x_k \qquad var = \frac{1}{N}\sum_{k=1}^{N}(x_k - \mu)^2 \tag{1}$$

$$\sigma = \sqrt{v + \epsilon} \qquad y_i = \frac{x_i - \mu}{\sigma} \tag{2}$$

You should make sure each of the intermediary gradient derivations are all as simplified as possible, for ease of implementation.

[40]:
```
def batchnorm_backward(dout, cache):
    """Backward pass for batch normalization.

    For this implementation, you should write out a computation graph for
    batch normalization on paper and propagate gradients backward through
    intermediate nodes.

    Inputs:
    - dout: Upstream derivatives, of shape (N, D)
    - cache: Variable of intermediates from batchnorm_forward.

    Returns a tuple of:
    - dx: Gradient with respect to inputs x, of shape (N, D)
    - dgamma: Gradient with respect to scale parameter gamma, of shape (D,)
    - dbeta: Gradient with respect to shift parameter beta, of shape (D,)
    """
    ###########################################################################
    # TODO: Implement the backward pass for batch normalization. Store the    #
    # results in the dx, dgamma, and dbeta variables.                         #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    before_shape = 0.5 * 1. / np.sqrt(cache[5]+cache[6]) * (-1. / (cache[4]**2)␣
↪* (np.sum(dout * cache[1] * cache[2], axis=0)))
    after_shape = 2 * cache[2] * (1. / dout.shape[0] * np.ones((dout.shape[0],␣
↪dout.shape[1])) * before_shape)
    first_main_part = (dout * cache[1] * cache[3]) + after_shape
    second_main_part = 1. / dout.shape[0] * np.ones((dout.shape[0], dout.
↪shape[1])) * (-1 * np.sum(first_main_part, axis=0))
```

```
    return first_main_part + second_main_part , np.sum(dout*cache[0], axis=0),␣
↪np.sum(dout, axis=0)
```

Run the following to numerically check your backward pass.

[43]:
```
# Gradient check batchnorm backward pass.
np.random.seed(231)
N, D = 4, 5
x = 5 * np.random.randn(N, D) + 12
gamma = np.random.randn(D)
beta = np.random.randn(D)
dout = np.random.randn(N, D)

bn_param = {'mode': 'train'}
fx = lambda x: batchnorm_forward(x, gamma, beta, bn_param)[0]
fg = lambda a: batchnorm_forward(x, a, beta, bn_param)[0]
fb = lambda b: batchnorm_forward(x, gamma, b, bn_param)[0]

dx_num = eval_numerical_gradient_array(fx, x, dout)
da_num = eval_numerical_gradient_array(fg, gamma.copy(), dout)
db_num = eval_numerical_gradient_array(fb, beta.copy(), dout)

_, cache = batchnorm_forward(x, gamma, beta, bn_param)
dx, dgamma, dbeta = batchnorm_backward(dout, cache)

# You should expect to see relative errors between 1e-13 and 1e-8.
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

```
dx error:  3.925968740641243e-06
dgamma error:  7.417225040694815e-13
dbeta error:  2.379446949959628e-12
```

13 Loss layer: Softmax

Now implement the loss and gradient for softmax in the `softmax_loss` function.

[70]:
```
def softmax_loss(x, y):
    """
    Computes the loss and gradient for softmax classification.

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

```
    Returns a tuple of:
    - loss: Scalar giving the loss
    - dx: Gradient of the loss with respect to x
    """
    ############################################################################
    # TODO: Implement the softmax_loss function.                               #
    ############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    section1 = (x-np.max(x,axis=1,keepdims=True))-np.log(np.sum(np.exp(x-np.
 ↪max(x,axis=1,keepdims=True)),axis=1,keepdims=True))
    x_derivative = np.exp(section1)
    x_derivative[np.arange(x.shape[0]), y.astype(int)] -= 1
    return (-np.sum(section1[np.arange(x.shape[0]), y.astype(int)],␣
 ↪keepdims=True) / x.shape[0])[0], x_derivative / x.shape[0]
```

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

[71]:
```
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: 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))
```

```
Testing softmax_loss:
loss:  2.302545844500738
dx error:  9.384673161989355e-09
```

14 Loss layer: MSE

Now implement the loss and gradient for mean squared error in the `mse_loss` function.

[86]:
```
def mse_loss(x, y):
    """
    Computes the loss and gradient for MSE loss.
```

```
    Inputs:
    - x: Input data, of shape (N,) where x[i] is the predicted vector for
        the ith input.
    - y: Vector of target values, of shape (N,) where y[i] is the target value
        for the ith input.

    Returns a tuple of:
    - loss: Scalar giving the loss
    - dx: Gradient of the loss with respect to x
    """
    ###########################################################################
    # TODO: Implement the mse_loss function.                                  #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    return np.sum(np.power(y-x, 2))/x.shape[0], (2*x-2*y)/x.shape[0]
```

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

[87]:
```
np.random.seed(231)
num_inputs = 50
x = np.random.randn(num_inputs)
y = np.random.randn(num_inputs)

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

# Test mse_loss function. Loss should be close to 1.9 and dx error should be␣
 ↪around e-9
print('\nTesting mse_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing mse_loss:
loss:  1.8672282748726519
dx error:  2.8607953262121067e-09
```

15 Multi-Layer Fully Connected Network

In this part, you will implement a fully connected network with an arbitrary number of hidden layers.

[98]:
```
class FullyConnectedNet(object):
    """Class for a multi-layer fully connected neural network.

    Network contains an arbitrary number of hidden layers, ReLU nonlinearities,
```

```
    and a softmax loss function for a classification problem or the MSE loss␣
↪function for
    a regression problem. This will also implement batch normalization as an␣
↪option.
    For a network with L layers, the architecture will be

    {affine - [batchnorm] - relu} x (L - 1) - affine - softmax/mse

    where batch normalization is optional in each layer and the {...} block is
    repeated L - 1 times.

    Learnable parameters are stored in the self.params dictionary and will be␣
↪learned
    using the Solver class.
    """

    def __init__(
        self,
        category,
        hidden_dims,
        normalization,
        input_dim=784,
        output_dim=10,
        reg=0.0,
        weight_scale=1e-2,
        dtype=np.float32,
    ):
        """Initialize a new FullyConnectedNet.

        Inputs:
        - category: The type of the problem. Valid values are "classification",
            "regression".
        - hidden_dims: A list of integers giving the size of each hidden layer.
        - normalization: A list of booleans which shows that we have batch
            normalization after the affine layer.
        - input_dim: An integer giving the size of the input.
        - output_dim: An integer giving the number of classes to classify. It
            is 1 for a regression problem.
        - reg: Scalar giving L2 regularization strength.
        - weight_scale: Scalar giving the standard deviation for random
            initialization of the weights.
        - dtype: A numpy datatype object; all computations will be performed␣
↪using
            this datatype. float32 is faster but less accurate, so you should␣
↪use
            float64 for numeric gradient checking.
        """
```

```
        self.category = category
        self.normalization = normalization
        self.reg = reg
        self.num_layers = 1 + len(hidden_dims)
        self.dtype = dtype
        self.params = {}
        self.bn_params = []

        ␣
↪############################################################################
        # TODO: Initialize the parameters of the network, storing all values in␣
↪    #
        # the self.params dictionary. Store weights and biases for the first␣
↪layer #
        # in W1 and b1; for the second layer use W2 and b2, etc. Weights should␣
↪be #
        # initialized from a normal distribution centered at 0 with standard   ␣
↪    #
        # deviation equal to weight_scale. Biases should be initialized to zero.
↪    #
        #                                                                       ␣
↪    #
        # When using batch normalization, store scale and shift parameters for␣
↪the #
        # first layer in gamma1 and beta1; for the second layer use gamma2 and ␣
↪    #
        # beta2, etc. Scale parameters should be initialized to ones and shift ␣
↪    #
        # parameters should be initialized to zeros.                            ␣
↪    #
        ␣
↪############################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        for i in range(0, self.num_layers):
            main_dim = [input_dim] + hidden_dims + [num_classes]
            self.params['W' + str(i+1)] = np.random.
↪normal(scale=weight_scale,size=(main_dim[i],main_dim[i+1]))
            self.params['b' + str(i+1)] = np.zeros(main_dim[i+1])
            if i != self.num_layers - 1:
                self.params['gamma' + str(i+1)] = np.ones(main_dim[i])
                self.params['beta' + str(i+1)] = np.zeros(main_dim[i])

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

```
        #                               END OF YOUR CODE                              ␣
↪    #
      ␣
↪##############################################################################

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

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

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

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

        Returns:
        If y is None, then run a test-time forward pass of the model and return
        scores for a classification problem or the predicted_values for
        a regression problem:
        - out: Array of shape (N, C) / (N, ) giving classification scores /␣
↪predicted values, where
        scores[i, c] is the classification score for X[i] and class c /␣
↪predicted_values[i]
        is the predicted value for X[i].


        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.
        """
        X = X.astype(self.dtype)
        mode = "test" if y is None else "train"
```

```
        # Set train/test mode for batchnorm params since they
        # behave differently during training and testing.
        for bn_param in self.bn_params:
            bn_param["mode"] = mode

        ############################################################################
        # TODO: Implement the forward pass for the fully connected net, computing  #
        # the class scores / target values for X and storing them in the out       #
        #  variable.                                                               #
        #                                                                          #
        # When using batch normalization, you'll need to pass self.bn_params[0] to #
        # the forward pass for the first batch normalization layer, pass           #
        # self.bn_params[1] to the forward pass for the second batch normalization #
        # layer, etc.                                                              #
        ############################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        self.cache = dict()
        self.batchnorm_cache = dict()
        scores = X
        for i in range(1, self.num_layers+1):
            if i != self.num_layers:
                scores, self.batchnorm_cache['batchnorm' + str(i)] = batchnorm_forward(scores, self.params['gamma' + str(i)], self.params['beta' + str(i)], self.bn_params[i-1])
                scores, cache = affine_relu_forward(scores, self.params['W' + str(i)], self.params['b' + str(i)])
            else:
                scores, cache = affine_forward(scores, self.params['W' + str(i)], self.params['b' + str(i)])
            self.cache['c' + str(i)] = cache
        out = scores

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

```
        #                                END OF YOUR CODE                              ␣
↪    #
        ␣
↪##############################################################################

        # If test mode return early.
        if mode == "test":
            return out

        loss, grads = 0.0, {}
        ␣
↪##############################################################################
        # TODO: Implement the backward pass for the fully connected net. Store␣
↪the #
        # loss in the loss variable and gradients in the grads dictionary.␣
↪Compute #
        # data loss using softmax/mse, and make sure that grads[k] holds the   ␣
↪    #
        # gradients for self.params[k]. Don't forget to add L2 regularization! ␣
↪    #
        #                                                                       ␣
↪    #
        # When using batch normalization, you don't need to regularize the␣
↪scale   #
        # and shift parameters.                                                 ␣
↪    #
        #                                                                       ␣
↪    #
        # 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)*****

        main_parameter = {}
        for i in range(self.num_layers, 0, -1):
            main_parameter['W' + str(i)] = i
        loss, der = softmax_loss(scores, y)
        if self.category != "classification":
            y = y.reshape(out.shape[0], 1)
            loss, der = mse_loss(out, y)
        for i in range(self.num_layers, 0, -1):
```

```
            first_parameter = i
            second_parameter = i+1
            if i != self.num_layers:
                der, grads['W' + str(i)], grads['b' + str(i)] =␣
↪affine_relu_backward(der, self.cache['c' + str(i)])
                first_parameter = 'gamma' + str(i)
                der, grads['gamma' + str(i)], grads['beta' + str(i)] =␣
↪batchnorm_backward(der, self.batchnorm_cache['batchnorm' + str(i)])
            else:
                second_parameter = 'b' + str(i)
                der, grads['W' + str(i)], grads['b' + str(i)] =␣
↪affine_backward(der, self.cache['c' + str(i)])
            grads['W' + str(i)] += self.reg*self.params['W' + str(i)]
            loss += 0.5*self.reg*np.sum(np.power(self.params['W' + str(i)], 2))

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

        return loss, grads
```

15.1 Initial Loss and Gradient Check

As a sanity check, run the following to check the initial loss and to gradient check the network both with and without regularization. This is a good way to see if the initial losses seem reasonable.

For gradient checking, you should expect to see errors around 1e-7 or less.

[100]:
```
np.random.seed(231)
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

for reg in [0, 3.14]:
    print("Running check with reg = ", reg)
    model = FullyConnectedNet(
        "classification",
        [H1, H2],
        [False, False],
        input_dim=D,
        output_dim=C,
        reg=reg,
        weight_scale=5e-2,
```

```
        dtype=np.float64
    )

    loss, grads = model.loss(X, y)
    print("Initial loss: ", loss)

    # Most of the errors should be on the order of e-7 or smaller.
    # NOTE: It is fine however to see an error for W2 on the order of e-5
    # for the check when reg = 0.0
    for name in sorted(grads):
        f = lambda _: model.loss(X, y)[0]
        grad_num = eval_numerical_gradient(f, model.params[name],␣
↪verbose=False, h=1e-5)
        print(f"{name} relative error: {rel_error(grad_num, grads[name])}")
```

```
Running check with reg =  0
Initial loss:  2.3309802038647067
W1 relative error: 0.1999794237856981
W2 relative error: 3.9093686179961866e-09
W3 relative error: 2.826479487714071e-07
b1 relative error: 0.19997458679995195
b2 relative error: 1.757994127476511e-09
b3 relative error: 1.4044618389324762e-10
beta1 relative error: 0.21734246875745225
beta2 relative error: 4.9099728341377236e-08
gamma1 relative error: 0.27282982079972345
gamma2 relative error: 7.906083804373158e-09
Running check with reg =  3.14
Initial loss:  7.059999870212515
W1 relative error: 0.4536505810701704
W2 relative error: 4.1797136534095544e-08
W3 relative error: 2.6147518955068794e-07
b1 relative error: 0.1999260872960772
b2 relative error: 4.674673363287726e-09
b3 relative error: 2.0910464777481823e-10
beta1 relative error: 0.1966015506006677
beta2 relative error: 1.4060053878165976e-08
gamma1 relative error: 0.1981997665768208
gamma2 relative error: 7.71058018171681e-07
```

[101]:
```
np.random.seed(231)
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

# You should expect losses between 1e-4~1e-10 for W,
# losses between 1e-08~1e-10 for b,
```

```
# and losses between 1e-08~1e-09 for beta and gammas.
for reg in [0, 3.14]:
    print('Running check with reg = ', reg)
    model = FullyConnectedNet(
        "classification",
        [H1, H2],
        [True, True],
        input_dim=D,
        output_dim=C,
        reg=reg,
        weight_scale=5e-2,
        dtype=np.float64
    )

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

    for name in sorted(grads):
        f = lambda _: model.loss(X, y)[0]
        grad_num = eval_numerical_gradient(f, model.params[name],␣
↪verbose=False, h=1e-5)
        print('%s relative error: %.2e' % (name, rel_error(grad_num,␣
↪grads[name])))
    if reg == 0: print()
```

```
Running check with reg =  0
Initial loss:  2.3309802038647067
W1 relative error: 2.00e-01
W2 relative error: 3.91e-09
W3 relative error: 2.83e-07
b1 relative error: 2.00e-01
b2 relative error: 1.76e-09
b3 relative error: 1.40e-10
beta1 relative error: 2.17e-01
beta2 relative error: 4.91e-08
gamma1 relative error: 2.73e-01
gamma2 relative error: 7.91e-09

Running check with reg =  3.14
Initial loss:  7.059999870212515
W1 relative error: 4.54e-01
W2 relative error: 4.18e-08
W3 relative error: 2.61e-07
b1 relative error: 2.00e-01
b2 relative error: 4.67e-09
b3 relative error: 2.09e-10
beta1 relative error: 1.97e-01
```

```
beta2 relative error: 1.41e-08
gamma1 relative error: 1.98e-01
gamma2 relative error: 7.71e-07
```

15.2 SGD+Momentum

Stochastic gradient descent with momentum is a widely used update rule that tends to make deep networks converge faster than vanilla stochastic gradient descent.

Implement the SGD+momentum update rule in the function `sgd_momentum`.

[110]:
```python
def sgd_momentum(w, dw, config=None):
    """
    Performs stochastic gradient descent with momentum.
    Inputs:
    - w: A numpy array giving the current weights.
    - dw: A numpy array of the same shape as w giving the gradient of the
    loss with respect to w.
    - config: A dictionary containing hyperparameter values such as learning
    rate, momentum.

    Returns:
      - next_w: The next point after the update.
      - config: The config dictionary to be passed to the next iteration of the
        update rule.

    config format:
    - learning_rate: Scalar learning rate.
    - momentum: Scalar between 0 and 1 giving the momentum value.
      Setting momentum = 0 reduces sgd_momentum to stochastic gradient descent.
    - velocity: A numpy array of the same shape as w and dw used to store a
      moving average of the gradients.
    """
    if config is None:
        config = {}
    config.setdefault("learning_rate", 1e-2)
    config.setdefault("momentum", 0.9)
    v = config.get("velocity", np.zeros_like(w))

    next_w = None
    ###########################################################################
    # TODO: Implement the momentum update formula. Store the updated value in #
    # the next_w variable. You should also use and update the velocity v.     #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    v = config['momentum'] * v - config['learning_rate'] * dw
    next_w = w + v
```

```
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ##############################################################################
    #                               END OF YOUR CODE                             #
    ##############################################################################
    config["velocity"] = v

    return next_w, config
```

Run the following to check your implementation. You should see errors less than e-8.

[111]:
```
N, D = 4, 5
w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
v = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

config = {"learning_rate": 1e-3, "velocity": v}
next_w, _ = sgd_momentum(w, dw, config=config)

expected_next_w = np.asarray([
  [ 0.1406,      0.20738947,  0.27417895,  0.34096842,  0.40775789],
  [ 0.47454737,  0.54133684,  0.60812632,  0.67491579,  0.74170526],
  [ 0.80849474,  0.87528421,  0.94207368,  1.00886316,  1.07565263],
  [ 1.14244211,  1.20923158,  1.27602105,  1.34281053,  1.4096    ]])
expected_velocity = np.asarray([
  [ 0.5406,      0.55475789,  0.56891579, 0.58307368,  0.59723158],
  [ 0.61138947,  0.62554737,  0.63970526,  0.65386316,  0.66802105],
  [ 0.68217895,  0.69633684,  0.71049474,  0.72465263,  0.73881053],
  [ 0.75296842,  0.76712632,  0.78128421,  0.79544211,  0.8096    ]])

# Should see relative errors around e-8 or less
print("next_w error: ", rel_error(next_w, expected_next_w))
print("velocity error: ", rel_error(expected_velocity, config["velocity"]))
```

```
next_w error:  8.882347033505819e-09
velocity error:  4.269287743278663e-09
```

16 MNIST

MNIST is a widely used dataset of handwritten digits that contains 60,000 handwritten digits for training a machine learning model and 10,000 handwritten digits for testing the model.

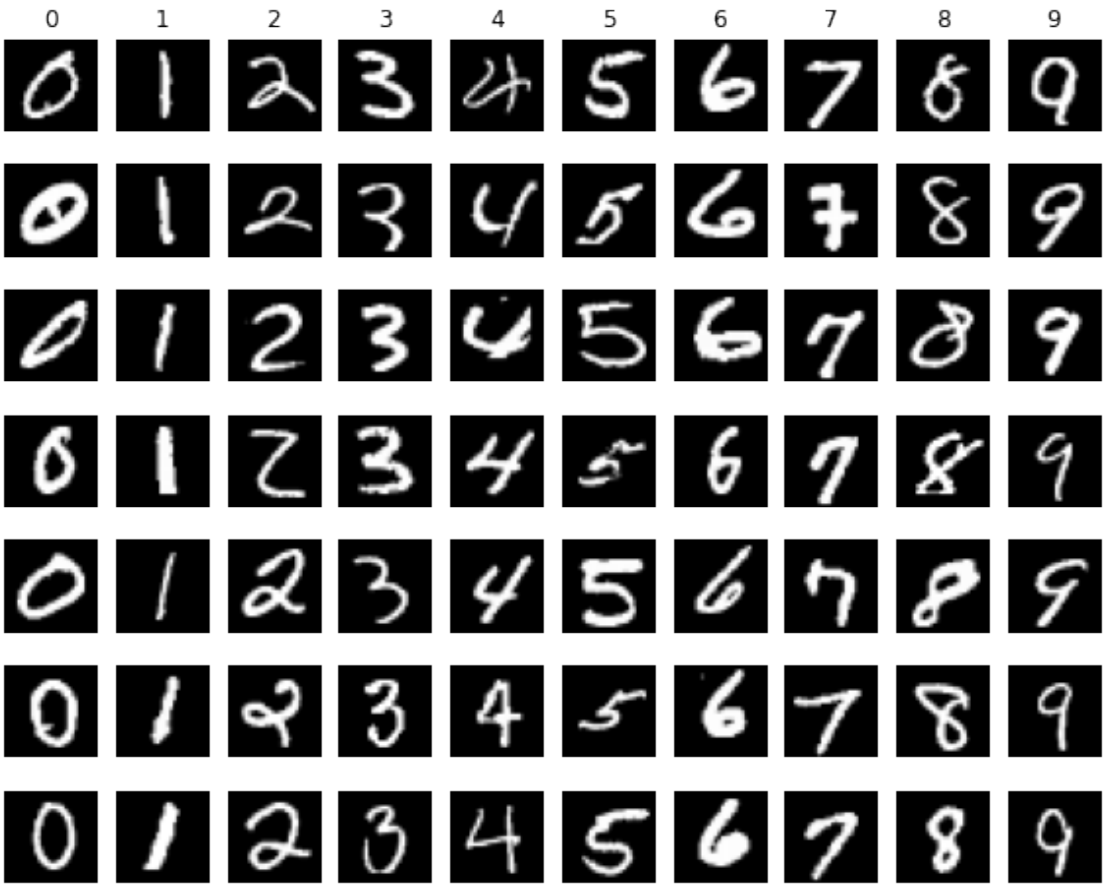
[112]:
```
X_train, y_train, X_val, y_val, X_test, y_test = get_MNIST_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
```

```
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to
./data\MNIST\raw\train-images-idx3-ubyte.gz

  0%|          | 0/9912422 [00:00<?, ?it/s]

Extracting ./data\MNIST\raw\train-images-idx3-ubyte.gz to ./data\MNIST\raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to
./data\MNIST\raw\train-labels-idx1-ubyte.gz

  0%|          | 0/28881 [00:00<?, ?it/s]

Extracting ./data\MNIST\raw\train-labels-idx1-ubyte.gz to ./data\MNIST\raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to
./data\MNIST\raw\t10k-images-idx3-ubyte.gz

  0%|          | 0/1648877 [00:00<?, ?it/s]

Extracting ./data\MNIST\raw\t10k-images-idx3-ubyte.gz to ./data\MNIST\raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to
./data\MNIST\raw\t10k-labels-idx1-ubyte.gz

  0%|          | 0/4542 [00:00<?, ?it/s]

Extracting ./data\MNIST\raw\t10k-labels-idx1-ubyte.gz to ./data\MNIST\raw

Train data shape:  (50000, 784)
Train labels shape:  (50000,)
Validation data shape:  (10000, 784)
Validation labels shape:  (10000,)
Test data shape:  (10000, 784)
Test labels shape:  (10000,)
```

[113]:
```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = list(range(10))
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
```

```
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].reshape((28, 28)))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```



Data normalization is an important step which ensures that each input parameter has a similar data distribution. This makes convergence faster while training the network.

[114]:
```
X_train, X_val, X_test = get_normalized_MNIST_data(X_train, X_val, X_test)
```

17 Train a Good Model!

Open the file `solver.py` and read through it to familiarize yourself with the API. After doing so, use a `Solver` instance to train the best fully connected model that you can on MNIST, storing your best model in the `MNIST_best_model` variable. We require you to get at least 95% accuracy on the validation set using a fully connected network.

[130]:
```
MNIST_best_model = None

################################################################################
# TODO: Train the best FullyConnectedNet that you can on MNIST. You might      #
# find batch normalization. Store your best model in                           #
# the best_model variable.                                                     #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

parameters = [[20], [True], 0.0001]
model = FullyConnectedNet('classification', parameters[0], parameters[1],␣
 ↪reg=parameters[2])
names = ['X_train', 'X_val', 'y_train', 'y_val', 'X_test', 'y_test']
values = [X_train, X_val, y_train, y_val, X_test, y_test]
data = {}
for i in range(6):
    data[names[i]] = values[i]
MNIST_solver = Solver(model, data, update_rule=sgd_momentum)
MNIST_solver.train()
MNIST_best_model = model
```

```
(Iteration 1 / 5000) loss: 2.302661
(Epoch 0 / 10) train acc: 0.092000; val_acc: 0.093600
(Iteration 11 / 5000) loss: 2.297867
(Iteration 21 / 5000) loss: 2.294341
(Iteration 31 / 5000) loss: 2.266065
(Iteration 41 / 5000) loss: 2.195250
(Iteration 51 / 5000) loss: 2.053968
(Iteration 61 / 5000) loss: 1.684931
(Iteration 71 / 5000) loss: 1.184081
(Iteration 81 / 5000) loss: 1.116254
(Iteration 91 / 5000) loss: 0.917548
(Iteration 101 / 5000) loss: 0.709875
(Iteration 111 / 5000) loss: 0.746112
(Iteration 121 / 5000) loss: 0.653636
(Iteration 131 / 5000) loss: 0.638611
(Iteration 141 / 5000) loss: 0.460241
(Iteration 151 / 5000) loss: 0.488071
(Iteration 161 / 5000) loss: 0.341996
(Iteration 171 / 5000) loss: 0.427220
(Iteration 181 / 5000) loss: 0.350957
(Iteration 191 / 5000) loss: 0.377413
(Iteration 201 / 5000) loss: 0.181834
(Iteration 211 / 5000) loss: 0.367366
(Iteration 221 / 5000) loss: 0.341361
(Iteration 231 / 5000) loss: 0.315849
(Iteration 241 / 5000) loss: 0.251462
```

```
(Iteration 251 / 5000) loss: 0.229804
(Iteration 261 / 5000) loss: 0.380522
(Iteration 271 / 5000) loss: 0.348616
(Iteration 281 / 5000) loss: 0.504498
(Iteration 291 / 5000) loss: 0.169218
(Iteration 301 / 5000) loss: 0.557958
(Iteration 311 / 5000) loss: 0.231069
(Iteration 321 / 5000) loss: 0.240029
(Iteration 331 / 5000) loss: 0.252061
(Iteration 341 / 5000) loss: 0.485101
(Iteration 351 / 5000) loss: 0.203685
(Iteration 361 / 5000) loss: 0.251386
(Iteration 371 / 5000) loss: 0.203693
(Iteration 381 / 5000) loss: 0.327580
(Iteration 391 / 5000) loss: 0.258355
(Iteration 401 / 5000) loss: 0.234555
(Iteration 411 / 5000) loss: 0.168655
(Iteration 421 / 5000) loss: 0.286708
(Iteration 431 / 5000) loss: 0.350575
(Iteration 441 / 5000) loss: 0.353172
(Iteration 451 / 5000) loss: 0.535741
(Iteration 461 / 5000) loss: 0.328897
(Iteration 471 / 5000) loss: 0.287128
(Iteration 481 / 5000) loss: 0.286437
(Iteration 491 / 5000) loss: 0.252431
(Epoch 1 / 10) train acc: 0.936000; val_acc: 0.928600
(Iteration 501 / 5000) loss: 0.317505
(Iteration 511 / 5000) loss: 0.114836
(Iteration 521 / 5000) loss: 0.360029
(Iteration 531 / 5000) loss: 0.257276
(Iteration 541 / 5000) loss: 0.122578
(Iteration 551 / 5000) loss: 0.291218
(Iteration 561 / 5000) loss: 0.177452
(Iteration 571 / 5000) loss: 0.312963
(Iteration 581 / 5000) loss: 0.427823
(Iteration 591 / 5000) loss: 0.456080
(Iteration 601 / 5000) loss: 0.334557
(Iteration 611 / 5000) loss: 0.224592
(Iteration 621 / 5000) loss: 0.209881
(Iteration 631 / 5000) loss: 0.302203
(Iteration 641 / 5000) loss: 0.152229
(Iteration 651 / 5000) loss: 0.265676
(Iteration 661 / 5000) loss: 0.172714
(Iteration 671 / 5000) loss: 0.236323
(Iteration 681 / 5000) loss: 0.195849
(Iteration 691 / 5000) loss: 0.262063
(Iteration 701 / 5000) loss: 0.281968
(Iteration 711 / 5000) loss: 0.257573
```

```
(Iteration 721 / 5000) loss: 0.274067
(Iteration 731 / 5000) loss: 0.293366
(Iteration 741 / 5000) loss: 0.187390
(Iteration 751 / 5000) loss: 0.190743
(Iteration 761 / 5000) loss: 0.166664
(Iteration 771 / 5000) loss: 0.299682
(Iteration 781 / 5000) loss: 0.265708
(Iteration 791 / 5000) loss: 0.157915
(Iteration 801 / 5000) loss: 0.197524
(Iteration 811 / 5000) loss: 0.333827
(Iteration 821 / 5000) loss: 0.221945
(Iteration 831 / 5000) loss: 0.201086
(Iteration 841 / 5000) loss: 0.238976
(Iteration 851 / 5000) loss: 0.122704
(Iteration 861 / 5000) loss: 0.137131
(Iteration 871 / 5000) loss: 0.247349
(Iteration 881 / 5000) loss: 0.332140
(Iteration 891 / 5000) loss: 0.309816
(Iteration 901 / 5000) loss: 0.203995
(Iteration 911 / 5000) loss: 0.256087
(Iteration 921 / 5000) loss: 0.138414
(Iteration 931 / 5000) loss: 0.206925
(Iteration 941 / 5000) loss: 0.237345
(Iteration 951 / 5000) loss: 0.145176
(Iteration 961 / 5000) loss: 0.176485
(Iteration 971 / 5000) loss: 0.255023
(Iteration 981 / 5000) loss: 0.146622
(Iteration 991 / 5000) loss: 0.136614
(Epoch 2 / 10) train acc: 0.940000; val_acc: 0.944900
(Iteration 1001 / 5000) loss: 0.324771
(Iteration 1011 / 5000) loss: 0.314644
(Iteration 1021 / 5000) loss: 0.245609
(Iteration 1031 / 5000) loss: 0.216715
(Iteration 1041 / 5000) loss: 0.179346
(Iteration 1051 / 5000) loss: 0.183896
(Iteration 1061 / 5000) loss: 0.244951
(Iteration 1071 / 5000) loss: 0.134152
(Iteration 1081 / 5000) loss: 0.237698
(Iteration 1091 / 5000) loss: 0.183907
(Iteration 1101 / 5000) loss: 0.214306
(Iteration 1111 / 5000) loss: 0.146761
(Iteration 1121 / 5000) loss: 0.184400
(Iteration 1131 / 5000) loss: 0.166653
(Iteration 1141 / 5000) loss: 0.160175
(Iteration 1151 / 5000) loss: 0.272649
(Iteration 1161 / 5000) loss: 0.132612
(Iteration 1171 / 5000) loss: 0.203641
(Iteration 1181 / 5000) loss: 0.230129
```

```
(Iteration 1191 / 5000) loss: 0.211195
(Iteration 1201 / 5000) loss: 0.328073
(Iteration 1211 / 5000) loss: 0.197906
(Iteration 1221 / 5000) loss: 0.066077
(Iteration 1231 / 5000) loss: 0.274453
(Iteration 1241 / 5000) loss: 0.169911
(Iteration 1251 / 5000) loss: 0.132228
(Iteration 1261 / 5000) loss: 0.132390
(Iteration 1271 / 5000) loss: 0.211947
(Iteration 1281 / 5000) loss: 0.180476
(Iteration 1291 / 5000) loss: 0.245308
(Iteration 1301 / 5000) loss: 0.127634
(Iteration 1311 / 5000) loss: 0.195560
(Iteration 1321 / 5000) loss: 0.180443
(Iteration 1331 / 5000) loss: 0.270526
(Iteration 1341 / 5000) loss: 0.233856
(Iteration 1351 / 5000) loss: 0.038411
(Iteration 1361 / 5000) loss: 0.248095
(Iteration 1371 / 5000) loss: 0.215527
(Iteration 1381 / 5000) loss: 0.106490
(Iteration 1391 / 5000) loss: 0.138062
(Iteration 1401 / 5000) loss: 0.170132
(Iteration 1411 / 5000) loss: 0.122523
(Iteration 1421 / 5000) loss: 0.070025
(Iteration 1431 / 5000) loss: 0.194639
(Iteration 1441 / 5000) loss: 0.115503
(Iteration 1451 / 5000) loss: 0.136643
(Iteration 1461 / 5000) loss: 0.200164
(Iteration 1471 / 5000) loss: 0.193956
(Iteration 1481 / 5000) loss: 0.111635
(Iteration 1491 / 5000) loss: 0.103716
(Epoch 3 / 10) train acc: 0.948000; val_acc: 0.949100
(Iteration 1501 / 5000) loss: 0.048530
(Iteration 1511 / 5000) loss: 0.153122
(Iteration 1521 / 5000) loss: 0.186963
(Iteration 1531 / 5000) loss: 0.165351
(Iteration 1541 / 5000) loss: 0.103652
(Iteration 1551 / 5000) loss: 0.086432
(Iteration 1561 / 5000) loss: 0.105317
(Iteration 1571 / 5000) loss: 0.065514
(Iteration 1581 / 5000) loss: 0.305351
(Iteration 1591 / 5000) loss: 0.223433
(Iteration 1601 / 5000) loss: 0.184575
(Iteration 1611 / 5000) loss: 0.228834
(Iteration 1621 / 5000) loss: 0.153871
(Iteration 1631 / 5000) loss: 0.144551
(Iteration 1641 / 5000) loss: 0.158112
(Iteration 1651 / 5000) loss: 0.178091
```

```
(Iteration 1661 / 5000) loss: 0.070481
(Iteration 1671 / 5000) loss: 0.117658
(Iteration 1681 / 5000) loss: 0.108190
(Iteration 1691 / 5000) loss: 0.240401
(Iteration 1701 / 5000) loss: 0.225515
(Iteration 1711 / 5000) loss: 0.148956
(Iteration 1721 / 5000) loss: 0.085945
(Iteration 1731 / 5000) loss: 0.282833
(Iteration 1741 / 5000) loss: 0.152339
(Iteration 1751 / 5000) loss: 0.261931
(Iteration 1761 / 5000) loss: 0.211762
(Iteration 1771 / 5000) loss: 0.223554
(Iteration 1781 / 5000) loss: 0.341998
(Iteration 1791 / 5000) loss: 0.136698
(Iteration 1801 / 5000) loss: 0.124293
(Iteration 1811 / 5000) loss: 0.183823
(Iteration 1821 / 5000) loss: 0.152424
(Iteration 1831 / 5000) loss: 0.193008
(Iteration 1841 / 5000) loss: 0.154182
(Iteration 1851 / 5000) loss: 0.141084
(Iteration 1861 / 5000) loss: 0.290268
(Iteration 1871 / 5000) loss: 0.138835
(Iteration 1881 / 5000) loss: 0.242727
(Iteration 1891 / 5000) loss: 0.121471
(Iteration 1901 / 5000) loss: 0.183151
(Iteration 1911 / 5000) loss: 0.121418
(Iteration 1921 / 5000) loss: 0.164544
(Iteration 1931 / 5000) loss: 0.150246
(Iteration 1941 / 5000) loss: 0.115300
(Iteration 1951 / 5000) loss: 0.232217
(Iteration 1961 / 5000) loss: 0.228673
(Iteration 1971 / 5000) loss: 0.163716
(Iteration 1981 / 5000) loss: 0.054108
(Iteration 1991 / 5000) loss: 0.167189
(Epoch 4 / 10) train acc: 0.939000; val_acc: 0.950700
(Iteration 2001 / 5000) loss: 0.150035
(Iteration 2011 / 5000) loss: 0.144395
(Iteration 2021 / 5000) loss: 0.220130
(Iteration 2031 / 5000) loss: 0.226728
(Iteration 2041 / 5000) loss: 0.140372
(Iteration 2051 / 5000) loss: 0.200541
(Iteration 2061 / 5000) loss: 0.154787
(Iteration 2071 / 5000) loss: 0.162459
(Iteration 2081 / 5000) loss: 0.171591
(Iteration 2091 / 5000) loss: 0.161994
(Iteration 2101 / 5000) loss: 0.268932
(Iteration 2111 / 5000) loss: 0.115727
(Iteration 2121 / 5000) loss: 0.116320
```

```
(Iteration 2131 / 5000) loss: 0.282516
(Iteration 2141 / 5000) loss: 0.156541
(Iteration 2151 / 5000) loss: 0.215957
(Iteration 2161 / 5000) loss: 0.267018
(Iteration 2171 / 5000) loss: 0.236376
(Iteration 2181 / 5000) loss: 0.112104
(Iteration 2191 / 5000) loss: 0.159685
(Iteration 2201 / 5000) loss: 0.230413
(Iteration 2211 / 5000) loss: 0.065371
(Iteration 2221 / 5000) loss: 0.138908
(Iteration 2231 / 5000) loss: 0.278822
(Iteration 2241 / 5000) loss: 0.185231
(Iteration 2251 / 5000) loss: 0.233337
(Iteration 2261 / 5000) loss: 0.087415
(Iteration 2271 / 5000) loss: 0.146666
(Iteration 2281 / 5000) loss: 0.264767
(Iteration 2291 / 5000) loss: 0.082737
(Iteration 2301 / 5000) loss: 0.133665
(Iteration 2311 / 5000) loss: 0.193988
(Iteration 2321 / 5000) loss: 0.114659
(Iteration 2331 / 5000) loss: 0.093351
(Iteration 2341 / 5000) loss: 0.176438
(Iteration 2351 / 5000) loss: 0.141902
(Iteration 2361 / 5000) loss: 0.255191
(Iteration 2371 / 5000) loss: 0.157543
(Iteration 2381 / 5000) loss: 0.194882
(Iteration 2391 / 5000) loss: 0.080678
(Iteration 2401 / 5000) loss: 0.233245
(Iteration 2411 / 5000) loss: 0.107619
(Iteration 2421 / 5000) loss: 0.174294
(Iteration 2431 / 5000) loss: 0.227489
(Iteration 2441 / 5000) loss: 0.149281
(Iteration 2451 / 5000) loss: 0.171218
(Iteration 2461 / 5000) loss: 0.118622
(Iteration 2471 / 5000) loss: 0.120361
(Iteration 2481 / 5000) loss: 0.153397
(Iteration 2491 / 5000) loss: 0.151505
(Epoch 5 / 10) train acc: 0.959000; val_acc: 0.951800
(Iteration 2501 / 5000) loss: 0.136027
(Iteration 2511 / 5000) loss: 0.162532
(Iteration 2521 / 5000) loss: 0.152097
(Iteration 2531 / 5000) loss: 0.062900
(Iteration 2541 / 5000) loss: 0.165505
(Iteration 2551 / 5000) loss: 0.244690
(Iteration 2561 / 5000) loss: 0.163028
(Iteration 2571 / 5000) loss: 0.110109
(Iteration 2581 / 5000) loss: 0.206471
(Iteration 2591 / 5000) loss: 0.268631
```

```
(Iteration 2601 / 5000) loss: 0.092094
(Iteration 2611 / 5000) loss: 0.237822
(Iteration 2621 / 5000) loss: 0.109550
(Iteration 2631 / 5000) loss: 0.121881
(Iteration 2641 / 5000) loss: 0.115873
(Iteration 2651 / 5000) loss: 0.181656
(Iteration 2661 / 5000) loss: 0.086244
(Iteration 2671 / 5000) loss: 0.127814
(Iteration 2681 / 5000) loss: 0.170659
(Iteration 2691 / 5000) loss: 0.130027
(Iteration 2701 / 5000) loss: 0.069813
(Iteration 2711 / 5000) loss: 0.128398
(Iteration 2721 / 5000) loss: 0.102713
(Iteration 2731 / 5000) loss: 0.165818
(Iteration 2741 / 5000) loss: 0.087741
(Iteration 2751 / 5000) loss: 0.246523
(Iteration 2761 / 5000) loss: 0.088781
(Iteration 2771 / 5000) loss: 0.223274
(Iteration 2781 / 5000) loss: 0.094715
(Iteration 2791 / 5000) loss: 0.069793
(Iteration 2801 / 5000) loss: 0.154011
(Iteration 2811 / 5000) loss: 0.096112
(Iteration 2821 / 5000) loss: 0.076893
(Iteration 2831 / 5000) loss: 0.093638
(Iteration 2841 / 5000) loss: 0.198517
(Iteration 2851 / 5000) loss: 0.122076
(Iteration 2861 / 5000) loss: 0.177861
(Iteration 2871 / 5000) loss: 0.206739
(Iteration 2881 / 5000) loss: 0.127544
(Iteration 2891 / 5000) loss: 0.172314
(Iteration 2901 / 5000) loss: 0.047132
(Iteration 2911 / 5000) loss: 0.075055
(Iteration 2921 / 5000) loss: 0.095614
(Iteration 2931 / 5000) loss: 0.070959
(Iteration 2941 / 5000) loss: 0.232828
(Iteration 2951 / 5000) loss: 0.107345
(Iteration 2961 / 5000) loss: 0.168442
(Iteration 2971 / 5000) loss: 0.152529
(Iteration 2981 / 5000) loss: 0.118399
(Iteration 2991 / 5000) loss: 0.130793
(Epoch 6 / 10) train acc: 0.964000; val_acc: 0.952500
(Iteration 3001 / 5000) loss: 0.178498
(Iteration 3011 / 5000) loss: 0.104557
(Iteration 3021 / 5000) loss: 0.269629
(Iteration 3031 / 5000) loss: 0.088691
(Iteration 3041 / 5000) loss: 0.117175
(Iteration 3051 / 5000) loss: 0.111475
(Iteration 3061 / 5000) loss: 0.150796
```

```
(Iteration 3071 / 5000) loss: 0.166839
(Iteration 3081 / 5000) loss: 0.156805
(Iteration 3091 / 5000) loss: 0.076671
(Iteration 3101 / 5000) loss: 0.087640
(Iteration 3111 / 5000) loss: 0.146123
(Iteration 3121 / 5000) loss: 0.115022
(Iteration 3131 / 5000) loss: 0.220737
(Iteration 3141 / 5000) loss: 0.065539
(Iteration 3151 / 5000) loss: 0.153288
(Iteration 3161 / 5000) loss: 0.186098
(Iteration 3171 / 5000) loss: 0.142742
(Iteration 3181 / 5000) loss: 0.154013
(Iteration 3191 / 5000) loss: 0.169676
(Iteration 3201 / 5000) loss: 0.205801
(Iteration 3211 / 5000) loss: 0.129685
(Iteration 3221 / 5000) loss: 0.200520
(Iteration 3231 / 5000) loss: 0.188837
(Iteration 3241 / 5000) loss: 0.181224
(Iteration 3251 / 5000) loss: 0.119469
(Iteration 3261 / 5000) loss: 0.164483
(Iteration 3271 / 5000) loss: 0.082256
(Iteration 3281 / 5000) loss: 0.073595
(Iteration 3291 / 5000) loss: 0.199401
(Iteration 3301 / 5000) loss: 0.133149
(Iteration 3311 / 5000) loss: 0.095109
(Iteration 3321 / 5000) loss: 0.131454
(Iteration 3331 / 5000) loss: 0.244604
(Iteration 3341 / 5000) loss: 0.112423
(Iteration 3351 / 5000) loss: 0.103428
(Iteration 3361 / 5000) loss: 0.098017
(Iteration 3371 / 5000) loss: 0.145047
(Iteration 3381 / 5000) loss: 0.216100
(Iteration 3391 / 5000) loss: 0.095773
(Iteration 3401 / 5000) loss: 0.247716
(Iteration 3411 / 5000) loss: 0.065640
(Iteration 3421 / 5000) loss: 0.044682
(Iteration 3431 / 5000) loss: 0.111591
(Iteration 3441 / 5000) loss: 0.223207
(Iteration 3451 / 5000) loss: 0.181708
(Iteration 3461 / 5000) loss: 0.188894
(Iteration 3471 / 5000) loss: 0.101772
(Iteration 3481 / 5000) loss: 0.094802
(Iteration 3491 / 5000) loss: 0.169017
(Epoch 7 / 10) train acc: 0.961000; val_acc: 0.951500
(Iteration 3501 / 5000) loss: 0.074830
(Iteration 3511 / 5000) loss: 0.134608
(Iteration 3521 / 5000) loss: 0.150279
(Iteration 3531 / 5000) loss: 0.106805
```

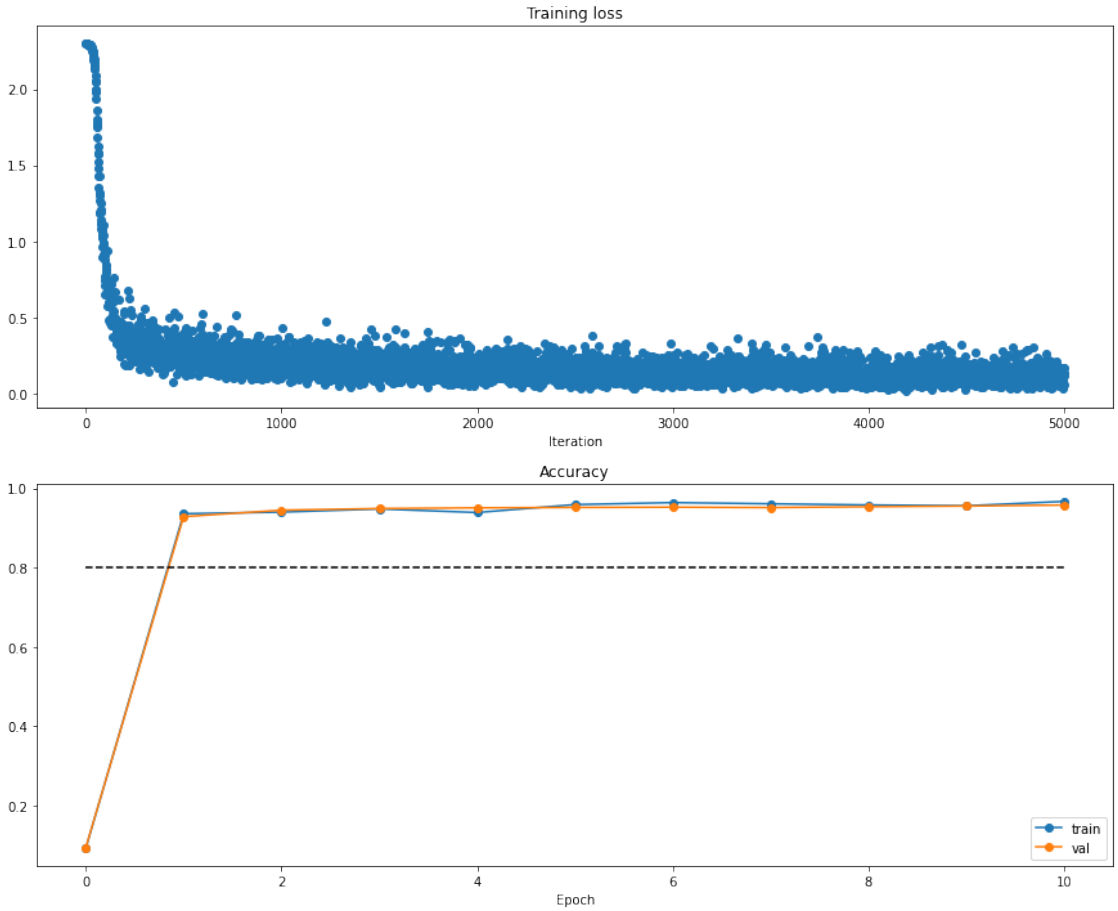
```
(Iteration 3541 / 5000) loss: 0.239294
(Iteration 3551 / 5000) loss: 0.088649
(Iteration 3561 / 5000) loss: 0.097054
(Iteration 3571 / 5000) loss: 0.091022
(Iteration 3581 / 5000) loss: 0.123961
(Iteration 3591 / 5000) loss: 0.100910
(Iteration 3601 / 5000) loss: 0.096909
(Iteration 3611 / 5000) loss: 0.081345
(Iteration 3621 / 5000) loss: 0.101146
(Iteration 3631 / 5000) loss: 0.123245
(Iteration 3641 / 5000) loss: 0.157589
(Iteration 3651 / 5000) loss: 0.108694
(Iteration 3661 / 5000) loss: 0.137832
(Iteration 3671 / 5000) loss: 0.090302
(Iteration 3681 / 5000) loss: 0.149655
(Iteration 3691 / 5000) loss: 0.205302
(Iteration 3701 / 5000) loss: 0.145298
(Iteration 3711 / 5000) loss: 0.113681
(Iteration 3721 / 5000) loss: 0.081105
(Iteration 3731 / 5000) loss: 0.083665
(Iteration 3741 / 5000) loss: 0.139595
(Iteration 3751 / 5000) loss: 0.138693
(Iteration 3761 / 5000) loss: 0.199336
(Iteration 3771 / 5000) loss: 0.129869
(Iteration 3781 / 5000) loss: 0.244757
(Iteration 3791 / 5000) loss: 0.199076
(Iteration 3801 / 5000) loss: 0.069333
(Iteration 3811 / 5000) loss: 0.241318
(Iteration 3821 / 5000) loss: 0.145392
(Iteration 3831 / 5000) loss: 0.169210
(Iteration 3841 / 5000) loss: 0.083865
(Iteration 3851 / 5000) loss: 0.258331
(Iteration 3861 / 5000) loss: 0.158819
(Iteration 3871 / 5000) loss: 0.068023
(Iteration 3881 / 5000) loss: 0.210536
(Iteration 3891 / 5000) loss: 0.071758
(Iteration 3901 / 5000) loss: 0.108948
(Iteration 3911 / 5000) loss: 0.155360
(Iteration 3921 / 5000) loss: 0.085552
(Iteration 3931 / 5000) loss: 0.168532
(Iteration 3941 / 5000) loss: 0.106982
(Iteration 3951 / 5000) loss: 0.108281
(Iteration 3961 / 5000) loss: 0.079555
(Iteration 3971 / 5000) loss: 0.106527
(Iteration 3981 / 5000) loss: 0.125052
(Iteration 3991 / 5000) loss: 0.166296
(Epoch 8 / 10) train acc: 0.958000; val_acc: 0.953600
(Iteration 4001 / 5000) loss: 0.128321
```

```
(Iteration 4011 / 5000) loss: 0.146398
(Iteration 4021 / 5000) loss: 0.130061
(Iteration 4031 / 5000) loss: 0.083713
(Iteration 4041 / 5000) loss: 0.223885
(Iteration 4051 / 5000) loss: 0.091173
(Iteration 4061 / 5000) loss: 0.256145
(Iteration 4071 / 5000) loss: 0.057791
(Iteration 4081 / 5000) loss: 0.068386
(Iteration 4091 / 5000) loss: 0.065509
(Iteration 4101 / 5000) loss: 0.024145
(Iteration 4111 / 5000) loss: 0.100115
(Iteration 4121 / 5000) loss: 0.087892
(Iteration 4131 / 5000) loss: 0.110097
(Iteration 4141 / 5000) loss: 0.077890
(Iteration 4151 / 5000) loss: 0.247436
(Iteration 4161 / 5000) loss: 0.050326
(Iteration 4171 / 5000) loss: 0.079490
(Iteration 4181 / 5000) loss: 0.045343
(Iteration 4191 / 5000) loss: 0.162700
(Iteration 4201 / 5000) loss: 0.067255
(Iteration 4211 / 5000) loss: 0.106507
(Iteration 4221 / 5000) loss: 0.117433
(Iteration 4231 / 5000) loss: 0.192870
(Iteration 4241 / 5000) loss: 0.140840
(Iteration 4251 / 5000) loss: 0.140154
(Iteration 4261 / 5000) loss: 0.078855
(Iteration 4271 / 5000) loss: 0.043446
(Iteration 4281 / 5000) loss: 0.163992
(Iteration 4291 / 5000) loss: 0.120459
(Iteration 4301 / 5000) loss: 0.090421
(Iteration 4311 / 5000) loss: 0.120337
(Iteration 4321 / 5000) loss: 0.111721
(Iteration 4331 / 5000) loss: 0.218496
(Iteration 4341 / 5000) loss: 0.116848
(Iteration 4351 / 5000) loss: 0.134550
(Iteration 4361 / 5000) loss: 0.186991
(Iteration 4371 / 5000) loss: 0.071530
(Iteration 4381 / 5000) loss: 0.189970
(Iteration 4391 / 5000) loss: 0.129055
(Iteration 4401 / 5000) loss: 0.095946
(Iteration 4411 / 5000) loss: 0.162496
(Iteration 4421 / 5000) loss: 0.087092
(Iteration 4431 / 5000) loss: 0.103126
(Iteration 4441 / 5000) loss: 0.156250
(Iteration 4451 / 5000) loss: 0.152561
(Iteration 4461 / 5000) loss: 0.142447
(Iteration 4471 / 5000) loss: 0.322884
(Iteration 4481 / 5000) loss: 0.204457
```

```
(Iteration 4491 / 5000) loss: 0.183331
(Epoch 9 / 10) train acc: 0.956000; val_acc: 0.955300
(Iteration 4501 / 5000) loss: 0.063245
(Iteration 4511 / 5000) loss: 0.124144
(Iteration 4521 / 5000) loss: 0.120450
(Iteration 4531 / 5000) loss: 0.083822
(Iteration 4541 / 5000) loss: 0.093633
(Iteration 4551 / 5000) loss: 0.075805
(Iteration 4561 / 5000) loss: 0.147220
(Iteration 4571 / 5000) loss: 0.130483
(Iteration 4581 / 5000) loss: 0.109378
(Iteration 4591 / 5000) loss: 0.086313
(Iteration 4601 / 5000) loss: 0.075867
(Iteration 4611 / 5000) loss: 0.058456
(Iteration 4621 / 5000) loss: 0.117620
(Iteration 4631 / 5000) loss: 0.076936
(Iteration 4641 / 5000) loss: 0.166701
(Iteration 4651 / 5000) loss: 0.116602
(Iteration 4661 / 5000) loss: 0.060929
(Iteration 4671 / 5000) loss: 0.194854
(Iteration 4681 / 5000) loss: 0.131531
(Iteration 4691 / 5000) loss: 0.150114
(Iteration 4701 / 5000) loss: 0.231506
(Iteration 4711 / 5000) loss: 0.132195
(Iteration 4721 / 5000) loss: 0.034741
(Iteration 4731 / 5000) loss: 0.152539
(Iteration 4741 / 5000) loss: 0.110821
(Iteration 4751 / 5000) loss: 0.093792
(Iteration 4761 / 5000) loss: 0.115045
(Iteration 4771 / 5000) loss: 0.138836
(Iteration 4781 / 5000) loss: 0.258117
(Iteration 4791 / 5000) loss: 0.121686
(Iteration 4801 / 5000) loss: 0.163133
(Iteration 4811 / 5000) loss: 0.285849
(Iteration 4821 / 5000) loss: 0.197464
(Iteration 4831 / 5000) loss: 0.060346
(Iteration 4841 / 5000) loss: 0.095297
(Iteration 4851 / 5000) loss: 0.140385
(Iteration 4861 / 5000) loss: 0.089366
(Iteration 4871 / 5000) loss: 0.038849
(Iteration 4881 / 5000) loss: 0.141058
(Iteration 4891 / 5000) loss: 0.164312
(Iteration 4901 / 5000) loss: 0.115676
(Iteration 4911 / 5000) loss: 0.191653
(Iteration 4921 / 5000) loss: 0.066303
(Iteration 4931 / 5000) loss: 0.033886
(Iteration 4941 / 5000) loss: 0.151648
(Iteration 4951 / 5000) loss: 0.084077
```

```
(Iteration 4961 / 5000) loss: 0.061302
(Iteration 4971 / 5000) loss: 0.125203
(Iteration 4981 / 5000) loss: 0.153827
(Iteration 4991 / 5000) loss: 0.051257
(Epoch 10 / 10) train acc: 0.967000; val_acc: 0.957800
```

[131]:
```
# Run this cell to visualize training loss and train / val accuracy

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

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(MNIST_solver.train_acc_history, '-o', label='train')
plt.plot(MNIST_solver.val_acc_history, '-o', label='val')
plt.plot([0.8] * len(MNIST_solver.val_acc_history), 'k--')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```



18 Test Your Model!

Run your best model on the validation and test sets. You should achieve at least 95% accuracy on the validation set.

[132]:
```
y_test_pred = np.argmax(MNIST_best_model.loss(data['X_test']), axis=1)
y_val_pred = np.argmax(MNIST_best_model.loss(data['X_val']), axis=1)
print('Validation set accuracy: ', (y_val_pred == data['y_val']).mean())
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

```
Validation set accuracy:  0.9578
Test set accuracy:  0.9566
```

19 California housing dataset

This is a dataset obtained from the StatLib repository. The data pertains to the houses found in a given California district and some summary stats about them based on the 1990 census data.

[133]:
```
california_housing = fetch_california_housing(as_frame=True)
california_housing.frame.head()
```

[133]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude	MedHouseVal
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23	4.526
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22	3.585
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24	3.521
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25	3.413
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25	3.422

[134]:
```
X_train, y_train, X_val, y_val, X_test, y_test = get_california_housing_data()
print('Train data shape: ', X_train.shape)
print('Train target values shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation target values shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test target values shape: ', y_test.shape)
```

```
Train data shape:  (15640, 8)
Train target values shape:  (15640,)
Validation data shape:  (2500, 8)
Validation target values shape:  (2500,)
Test data shape:  (2500, 8)
Test target values shape:  (2500,)
```

[135]:
```
X_train, X_val, X_test = get_california_housing_normalized__data(X_train,␣
↪X_val, X_test)
```

20 Train a Good Model!

Train the best fully connected model that you can on california housing, storing your best model in the `california_housing_best_model` variable.

[138]:
```
california_housing_best_model = None

################################################################################
# TODO: Train the best FullyConnectedNet that you can on california housing.   #
# You might find batch normalization useful. Store your best model in          #
# the best_model variable.                                                     #
################################################################################
```

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

parameters = [[20], [True], 0.0001]
model = FullyConnectedNet('classification', parameters[0], parameters[1],␣
 ↪reg=parameters[2])
names = ['X_train', 'X_val', 'y_train', 'y_val', 'X_test', 'y_test']
values = [X_train, X_val, y_train, y_val, X_test, y_test]
data = {}
for i in range(6):
    data[names[i]] = values[i]
MNIST_solver = Solver(model, data, update_rule=sgd_momentum)
MNIST_solver.train()
MNIST_best_model = model
```

[139]:
```
Run this cell to visualize training loss and train / val RMS error

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

plt.subplot(2, 1, 2)
plt.title('RMS Error')
plt.plot(california_housing_solver.train_acc_history, '-o', label='train')
plt.plot(california_housing_solver.val_acc_history, '-o', label='val')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```