# ECE 288: ML for Physical Applications - Assignment 1:

## Instructor: Yuanyuan Shi

## Teaching Assistants


- Srinivas Rao Daru, [sdaru@ucsd.edu]
- Tawaana Gustad Homavazir, [thomavaz@ucsd.edu]
- Rohin Garg, [rgarg@ucsd.edu]
- Rishabh Jangir, [rjangir@ucsd.edu]


# Instructions

1. This assignment must be completed individually.
2. All solutions must be written in this notebook
3. This notebook contains skeleton code, which should not be modified
4. You must submit this notebook exported as a pdf. You must also submit this notebook as an .ipynb file.
5. You must submit both files (.pdf and .ipynb) on Gradescope. You must mark each problem on Gradescope in the pdf.

# Description

In this assignment you will implement a Fully Connected Network and a Convolutional Neural Network in Python and Numpy. Implement the forward and backward pass for each layer as solved for in the theoretical questions above. We will also explore the impact of using momentum with SGD . Observe the difference in performance between the Fully Connected Network and Convolutional Network on the same task.

# Dataset

The MPIIFaceGaze Dataset contains 213,659 full face images and corresponding ground-truth gaze positions collected from 15 users during everyday laptop use over several months.

## Inputs

Each full face image is processed to retrieve the images of the two eyes, *left* and *right*. For the purposes of this assignment your input is an image formed by the side-by-side stacking of the two eyes.

## Output

The original dataset contains the 2D and 3D gaze positions corresponding to each full face image. For the purposes of this assignment you have been provided with classification labels

that divides the field of view into 4 quadrants where the origin at the center of the field. The label corresponds to the quadrant that the subject is gazing at.

## Objective

Your task is to train a set of networks to classify the eye gaze into the 4 quadrants.

# Instructions

1. Download the dataset from Dataset Google Drive
2. Copy and unzip it in the same folder as the .ipynb
3. The folder structure must look like -ECE228-Homeowrk-1/

```
-utils/
-Homework-1.ipynb
-/datasets/MPIIFaceGaze/
    -Image
    -Label
```

## Note:

Do not install any additional packages. The TAs will use the same environment as defined in the config file we provide you, so anything that's not in there by default will probably cause your code to break during grading. Failure to follow any of these instructions will lead to point deductions.

# Programming assignments start here

In [1]:

```
# Do not change this code, you should not need any additional imports
from utils.solver_gaze import Solver
from utils.data_utils import Data
import numpy as np
import matplotlib.pyplot as plt
import cv2
np.random.seed(228)
```

In [2]:
```
%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 extenrnal modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

In [3]:
```
batch_size = 256
dataset_directory = './datasets/MPIIFaceGaze/'
```

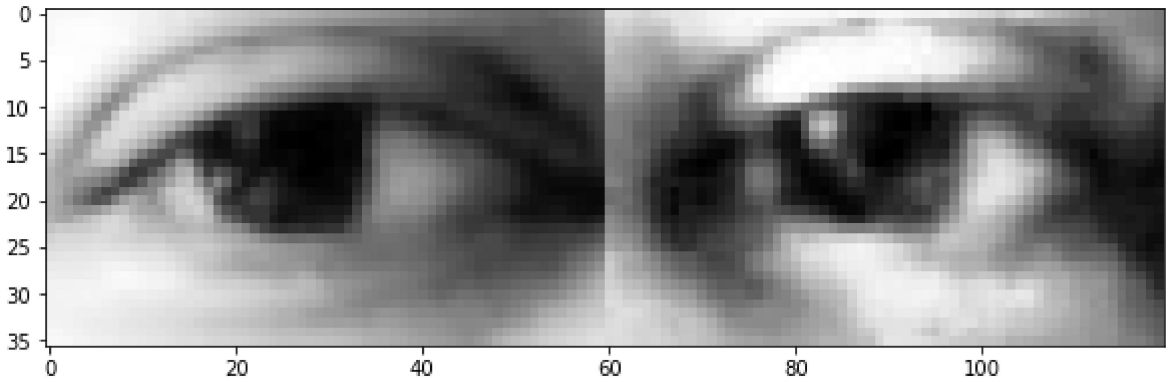
In [4]:
```
def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))

def eval_numerical_gradient(f, x, verbose=True, h=0.00001):
    """
      a naive implementation of numerical gradient of f at x
      - f should be a function that takes a single argument
      - x is the point (numpy array) to evaluate the gradient at
    """
    fx = f(x) # evaluate function value at original point
    grad = np.zeros_like(x)
    # iterate over all indexes in x
    it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])

    while not it.finished:
        # evaluate function at x+h
        ix = it.multi_index
        oldval = x[ix]
        x[ix] = oldval + h # increment by h
        fxph = f(x) # evalute f(x + h)
        x[ix] = oldval - h
        fxmh = f(x) # evaluate f(x - h)
        x[ix] = oldval # restore

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

    return grad

def eval_numerical_gradient_array(f, x, df, h=1e-5):
    """
    Evaluate a numeric gradient for a function that accepts a numpy
    array and returns a numpy array.
    """
    grad = np.zeros_like(x)
    it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
    while not it.finished:
        ix = it.multi_index
        oldval = x[ix]
        x[ix] = oldval + h
        pos = f(x).copy()
        x[ix] = oldval - h
        neg = f(x).copy()
        x[ix] = oldval
```

```
        grad[ix] = np.sum((pos - neg) * df) / (2 * h)
        it.iternext()
    return grad
```

In [5]:
```
# Sample data loading for a batch of length 1
inputs, labels = next(Data(batch_size=1, dataset_directory=dataset_directory).load_t

print('Training data shape: ', inputs[0].shape)
print('Training labels shape: ', labels[0])
plt.imshow(inputs[0])
```

```
Training data shape:  (36, 120)
Training labels shape:  1
```

Out[5]: <matplotlib.image.AxesImage at 0x20d70354e20>




# Part 1: Fully-connected neural networks (60 points)

## Part 1.1: Linear layer and activation functions

**Task 1.1: Affine layer: foward pass (no for loops are allowed) (3 points).**

In [6]:
```
def fc_forward(x, w, b):
    """
    Computes the forward pass for a 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)
    """
    N,D = x.shape[0], w.shape[0]
    out = None
    # TODO: Implement a fully-connected forward pass. Store the result in out. You
    # will need to reshape the input into rows.
```

```
    out = np.dot(x.reshape((N,D)),w) + b
    cache = (x, w, b)
    return out, cache
```

In [7]:

```
# Test the fc_forward function
# This is an example auto-grader and try to develop such tests for all later on sub-
num_inputs = 1
input_shape = (4, 3, 3)
output_dim = 10

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

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

out, _ = fc_forward(x, w, b)
correct_out = [[0.04735376,
                0.0917982,
                0.13624265,
                0.18068709,
                0.22513154,
                0.26957598,
                0.31402043,
                0.35846487,
                0.40290932,
                0.44735376]]

# Compare your output with ours. The error should be around e-7 or less.
print('Testing fc_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing fc_forward function:
difference:  2.663519958184904e-08
```

## Task 1.2: Affine layer: backward pass (no for loops are allowed) (5 points).

In [30]:

```
def fc_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
    N, D = x.shape[0], w.shape[0]
    # TODO: Implement the affine backward pass.
    dx = np.dot(dout, w.T).reshape(x.shape)
    dw = np.dot(x.reshape((N,D)).T, dout)
```

```
    db = np.sum(dout, axis = 0)
    return dx, dw, db
```

In [31]:

```
# Test the fc_backward function
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: fc_forward(x, w, b)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: fc_forward(x, w, b)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: fc_forward(x, w, b)[0], b, dout)

_, cache = fc_forward(x, w, b)
dx, dw, db = fc_backward(dout, cache)

# The error should be around e-9 or less
print('Testing fc_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 fc_backward function:
dx error:  1.057662444541783e-09
dw error:  1.2381371616015458e-10
db error:  4.3199822443612255e-11
```

## Task 1.3: ReLU activation: forward pass (no for loops are allowed) (2 points).

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

In [32]:

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

    cache = x
    return out, cache
```

In [33]:

```
# 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
```

## Task 1.4: ReLU activation: backward pass (no for loops are allowed) (4 points).

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

In [34]:

```
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.
    dx = dout.copy()
    dx[x <= 0] = 0

    return dx
```

In [35]:

```
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:  3.275624595460045e-12
```

## Task 1.5: Softmax loss layer (no for loops are allowed) (8 points).

In [67]:

```
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,
    # including the forward and backward passes.
    # forward
    N, C = x.shape
    y_one_hot = np.eye(C)[y]

    scores = x - np.max(x, axis=-1, keepdims=True)  # avoid numeric instability

    # Calculate softmax outputs e_i/sum(e_j)
    x_softmax = np.exp(scores) / np.sum(np.exp(scores),axis=1,keepdims=True)

    loss = - np.sum(np.sum(y_one_hot * np.log(x_softmax)))

    # Normalize the loss by dividing by the total number of samples N
    loss /= N

    # backward
    dx = x_softmax - y_one_hot
    # Normalize the gradient by dividing with the total number of samples N
    dx /= N
    return loss, dx
```

In [68]:

```
# Let's check your implementation
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 aro
print('\nTesting softmax_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing softmax_loss:
loss:  2.3026309764021677
dx error:  7.289867215410699e-09
```

In [69]:

```
def check_accuracy(model, dataloader, model_forward='loss'):
    ''' Check the accuracy of the classification task
    model: the model that must be run for the outputs
    dataloader: the dataloader for train_data, val_data or test_data
    model_forward: flag to decide the function that will be called on the model
                   'loss': For the numpy based models
                   'forward': For the pytorch based models
    '''
    y_pred = []
    y = []
```

```
    if model_forward!='loss':
        model.eval()
    for batch, (inputs, labels) in enumerate(dataloader, 0):
        if model_forward=='loss':
            scores = model.loss(inputs)
        elif model_forward=='forward':
            inputs = torch.tensor(inputs).unsqueeze(1).float()
            with torch.no_grad():
                scores = model(torch.tensor(inputs)).detach().numpy()
        y.append(labels)
        y_pred.append(np.argmax(scores, axis=1))

    y_pred = np.hstack(y_pred)
    y = np.hstack(y)
    acc = np.mean(y_pred == y)

    return acc*100
```

## Part 1.2: Fully-Connected Neural Networks

### Task 1.6: Two-layer network (no for loops are allowed in your implementation) (10 points)

Complete the implementation of the `TwoLayerNet` class. This class will serve as a model for the other networks you will implement in this assignment, so read through it to make sure you understand the API. You can run the cell below to test your implementation.

In [111…

```
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 FC - relu - FC - 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=1*36*120, hidden_dim=128, num_classes=4,
                 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.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.num_classes = num_classes
        self.weight_scale = weight_scale
        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'.
        self.params['W1'] = np.random.normal(0, weight_scale, size = (input_dim, hid
        self.params['b1'] = np.zeros(hidden_dim)
        self.params['W2'] = np.random.normal(0, weight_scale, size = (hidden_dim, nu
        self.params['b2'] = np.zeros(num_classes)


    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.
        out, first_cache = fc_forward(X, self.params['W1'], self.params['b1'])
        scores, second_cache = fc_forward(out, self.params['W2'], self.params['b2'])

        # 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.

        loss, dx2 = softmax_loss(scores, y)
        loss += 0.5*self.reg*(np.linalg.norm(self.params['W1'])**2 +
                              np.linalg.norm(self.params['W2'])**2)
```

```
        dx1, dW2, db2 = fc_backward(dx2, second_cache)
        dW2 += self.reg*self.params['W2']
        grads['W2'], grads['b2'] = dW2, db2

        dx, dW1, db1 = fc_backward(dx1, first_cache)
        dW1 += self.reg*self.params['W1']
        grads['W1'], grads['b1'] = dW1, db1

        return loss, grads
```

In [112…

```
# Let's check your implementation
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
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.49994135
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.66781506
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])))
```

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.52e-08
W2 relative error: 3.25e-10
b1 relative error: 8.37e-09
b2 relative error: 8.99e-11
Running numeric gradient check with reg =  0.7
W1 relative error: 2.53e-07
W2 relative error: 2.85e-08
b1 relative error: 1.56e-08
b2 relative error: 1.97e-09
```

## Task 1.7: Solver (3 points).

Following a more modular design, we have split the logic for training models into a separate class.

Open the file `utils/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 at least `50%` accuracy on the validation set.

In [114…

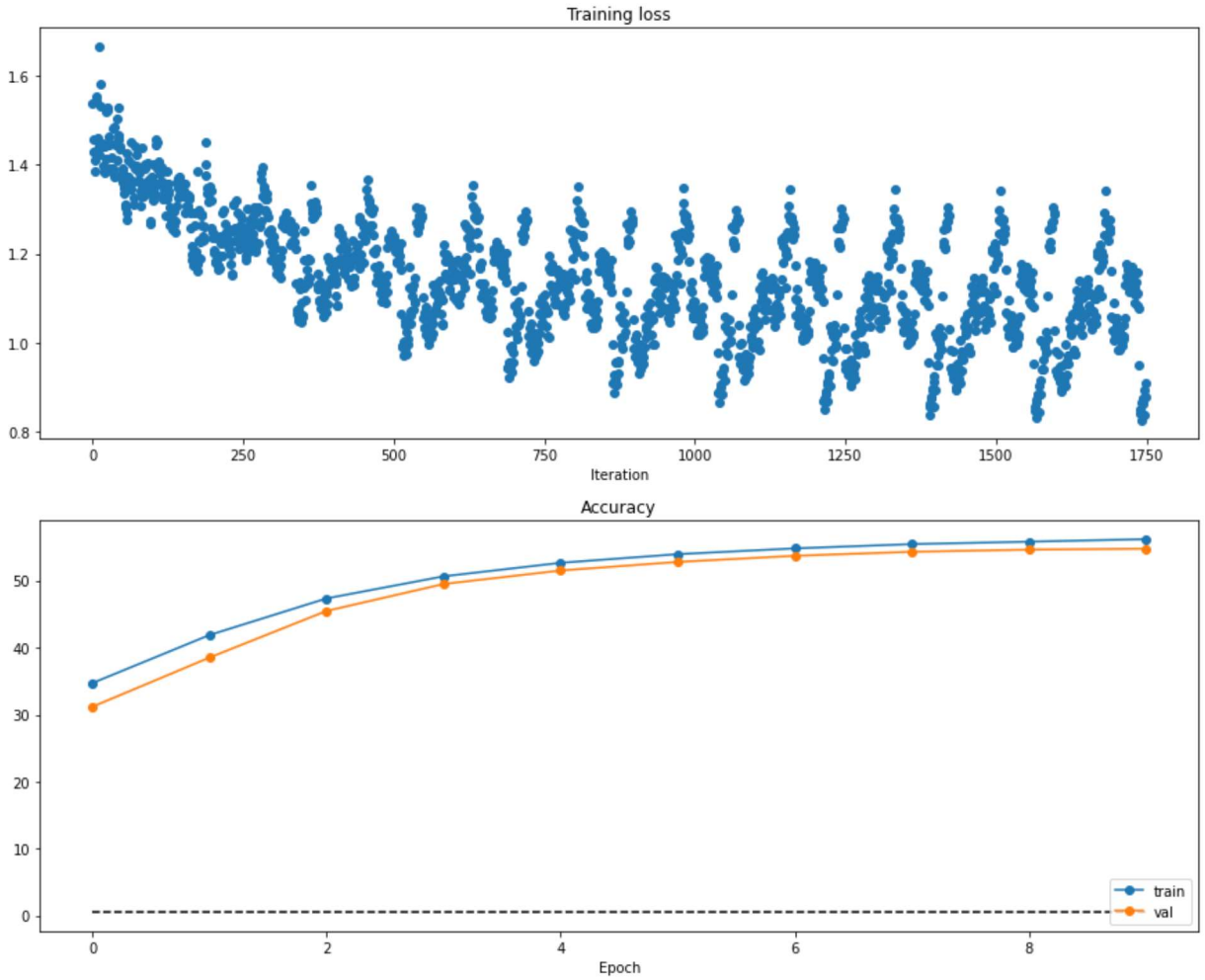
```
model = TwoLayerNet(weight_scale=5e-2)
solver = None
learning_rate = 1e-3 # TODO: Tune learning rate
update_rule = 'sgd'

solver = Solver(
    model,
    update_rule=update_rule,
    optim_config={'learning_rate': learning_rate,},
    lr_decay=0.8,
    dataset_directory=dataset_directory,
    num_epochs=10,
    batch_size=batch_size,
    print_every=50
)
solver.train()
```

```
(Epoch 0 Batch 1) loss: 1.537706
(Epoch 0 Batch 51) loss: 1.392027
(Epoch 0 Batch 101) loss: 1.399371
(Epoch 0 Batch 151) loss: 1.337121
(Epoch 0 / 10) train acc: 34.731111; val_acc: 31.200000
(Epoch 1 Batch 1) loss: 1.386596
(Epoch 1 Batch 51) loss: 1.245963
(Epoch 1 Batch 101) loss: 1.304185
(Epoch 1 Batch 151) loss: 1.272808
(Epoch 1 / 10) train acc: 41.895556; val_acc: 38.533333
(Epoch 2 Batch 1) loss: 1.250147
(Epoch 2 Batch 51) loss: 1.164412
(Epoch 2 Batch 101) loss: 1.259390
(Epoch 2 Batch 151) loss: 1.237533
(Epoch 2 / 10) train acc: 47.360000; val_acc: 45.488889
(Epoch 3 Batch 1) loss: 1.170260
(Epoch 3 Batch 51) loss: 1.113345
(Epoch 3 Batch 101) loss: 1.235789
```

```
(Epoch 3 Batch 151) loss: 1.215157
(Epoch 3 / 10) train acc: 50.655556; val_acc: 49.511111
(Epoch 4 Batch 1) loss: 1.118851
(Epoch 4 Batch 51) loss: 1.079281
(Epoch 4 Batch 101) loss: 1.221098
(Epoch 4 Batch 151) loss: 1.199761
(Epoch 4 / 10) train acc: 52.666667; val_acc: 51.522222
(Epoch 5 Batch 1) loss: 1.083654
(Epoch 5 Batch 51) loss: 1.056025
(Epoch 5 Batch 101) loss: 1.209584
(Epoch 5 Batch 151) loss: 1.188763
(Epoch 5 / 10) train acc: 53.957778; val_acc: 52.800000
(Epoch 6 Batch 1) loss: 1.058034
(Epoch 6 Batch 51) loss: 1.040406
(Epoch 6 Batch 101) loss: 1.199036
(Epoch 6 Batch 151) loss: 1.180902
(Epoch 6 / 10) train acc: 54.820000; val_acc: 53.711111
(Epoch 7 Batch 1) loss: 1.038024
(Epoch 7 Batch 51) loss: 1.030511
(Epoch 7 Batch 101) loss: 1.189156
(Epoch 7 Batch 151) loss: 1.175443
(Epoch 7 / 10) train acc: 55.457778; val_acc: 54.311111
(Epoch 8 Batch 1) loss: 1.021340
(Epoch 8 Batch 51) loss: 1.024876
(Epoch 8 Batch 101) loss: 1.180371
(Epoch 8 Batch 151) loss: 1.171842
(Epoch 8 / 10) train acc: 55.822222; val_acc: 54.633333
(Epoch 9 Batch 1) loss: 1.006864
(Epoch 9 Batch 51) loss: 1.022153
(Epoch 9 Batch 101) loss: 1.173067
(Epoch 9 Batch 151) loss: 1.169621
(Epoch 9 / 10) train acc: 56.197778; val_acc: 54.777778
```

In [115…

```
# 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()
```




## Task 1.8: Multilayer network (15 points).

Next you will implement a fully-connected network with an arbitrary number of hidden layers.

Implement the initialization, the forward pass, and the backward pass. For the moment don't worry about implementing dropout or batch/layer normalization.

In [303…

```
def fc_relu_forward(x, w, b):
    """
    Convenience layer that perorms 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
    """

    fc_out, fc_cache = fc_forward(x, w, b)
    out, relu_cache = relu_forward(fc_out)
    cache = (fc_cache, relu_cache)
    return out, cache
```

```
def fc_relu_backward(dout, cache):
    """
    Backward pass for the affine-relu convenience layer
    """
    fc_cache, relu_cache = cache

    relu_dout = relu_backward(dout,relu_cache)
    dx, dw, db= fc_backward(relu_dout, fc_cache)

    return dx, dw, db
```

In [304…

```
class FullyConnectedNet(object):
    """
    A fully-connected neural network with an arbitrary number of hidden layers,
    ReLU nonlinearities, and a softmax loss function. For a network with L layers,
    the architecture will be

    {affine - relu} x (L - 1) - affine - softmax

    where batch/layer normalization and dropout are optional, and the {...} block is
    repeated L - 1 times.

    Similar to the TwoLayerNet above, learnable parameters are stored in the
    self.params dictionary and will be learned using the Solver class.
    """

    def __init__(self, hidden_dims, input_dim=1*36*120, num_classes=4, reg=0.0,
                 weight_scale=1e-2, seed=None):
        """
        Initialize a new FullyConnectedNet.

        Inputs:
        - hidden_dims: A list of integers giving the size of each hidden layer.
        - input_dim: An integer giving the size of the input.
        - num_classes: An integer giving the number of classes to classify.
        - 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.
        - seed: If not None, then pass this random seed to the dropout layers. This
          will make the dropout layers deteriminstic so we can gradient check the
          model.
        """
        self.reg = reg
        self.num_layers = 1 + len(hidden_dims)
        self.params = {}
        self.dtype = np.float64
        # 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.

        for i in range(1,self.num_layers):
            if i == 1: # first layer initialization
                self.params['W'+str(i)] = np.random.normal(0, weight_scale, size = (
                self.params['b'+str(i)] = np.zeros(hidden_dims[i-1])
            else: # hidden layers initialization
                self.params['W'+str(i)] = np.random.normal(0, weight_scale, size = (
                self.params['b'+str(i)] = np.zeros(hidden_dims[i-1])
```

```
        # final layer
        self.params['W'+ str(self.num_layers)] = np.random.normal(0, weight_scale, s
        self.params['b'+ str(self.num_layers)] = np.zeros(num_classes)

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


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

        Input / output: Same as TwoLayerNet above.
        """
        X = X.astype(self.dtype)
        mode = 'test' if y is None else 'train'

        scores = None

        # TODO: Implement the forward pass for the fully-connected net, computing
        # the class scores for X and storing them in the scores variable.

        cache = [None] * (self.num_layers)
        for i in range(1,self.num_layers):
            if i == 1:
                out,cache[i-1] = fc_relu_forward(X,self.params['W'+str(i)],self.para
            else:
                out,cache[i-1] = fc_relu_forward(out,self.params['W'+str(i)],self.pa

        scores,cache[-1] = fc_forward(out,self.params['W'+str(self.num_layers)],self

        # If test mode return early
        if mode == 'test':
            return scores

        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, 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.
        loss, dx = softmax_loss(scores, y)

        for i in range(1, self.num_layers+1):
            loss += 0.5*self.reg*np.linalg.norm(self.params['W'+str(i)])**2

        dx, dw, db = fc_backward(dx,cache[self.num_layers-1])
        dw += self.reg * self.params['W'+str(self.num_layers)]
        grads['W'+str(self.num_layers)], grads['b'+str(self.num_layers)] = dw, db

        i = self.num_layers - 1
        while i > 0:
            dx, dw, db = fc_relu_backward(dx, cache[i-1])
            dw += self.reg * self.params['W'+str(i)]
            grads['W'+str(i)], grads['b'+str(i)] = dw, db
            i -= 1

        return loss, grads
```

In [305…

```
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(
        [H1, H2],
        input_dim=D,
        num_classes=C,
        reg=reg,
        weight_scale=5e-2
    )

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

    # Most of the errors should be on the order of 1e-7 or smaller.
    # NOTE: It is fine however to see an error for W2 on the order of 1e-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=1
        print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

```
Running check with reg =  0
Initial loss:  2.3003803003032077
W1 relative error: 1.72e-07
W2 relative error: 1.26e-06
W3 relative error: 8.12e-08
b1 relative error: 1.93e-08
b2 relative error: 5.02e-09
b3 relative error: 8.07e-11
Running check with reg =  3.14
Initial loss:  6.660009106538036
W1 relative error: 9.47e-09
W2 relative error: 6.13e-08
W3 relative error: 1.33e-07
b1 relative error: 9.86e-08
b2 relative error: 1.47e-09
b3 relative error: 2.38e-10
```

## Task 1.9: SGD+Momentum (5 points)

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

Open the file `utils/optim.py` and read the documentation at the top of the file to make sure you understand the API. Implement the SGD+momentum update rule below and run the following to check your implementation. You should see errors less than 1e-8.

In [122…

```
def sgd_momentum(w, dw, config=None):
    """
    Performs stochastic gradient descent with momentum.
```

```
    config format:
    - learning_rate: Scalar learning rate.
    - momentum: Scalar between 0 and 1 giving the momentum value.
      Setting momentum = 0 reduces to sgd.
    - 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.
    v = config['momentum']*v - config['learning_rate']*dw

    next_w = w + v
    config['velocity'] = v

    return next_w, config
```

## Notice:

In this task, the momentum update formula is different from the one mentioned in class. I implement another fomular cited from piazza to get the matched answer. The link is: https://piazza.com/class/l1704onp9dx6n5?cid=111

In [123…

```
# Let's check your implementation
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
```

In [344…

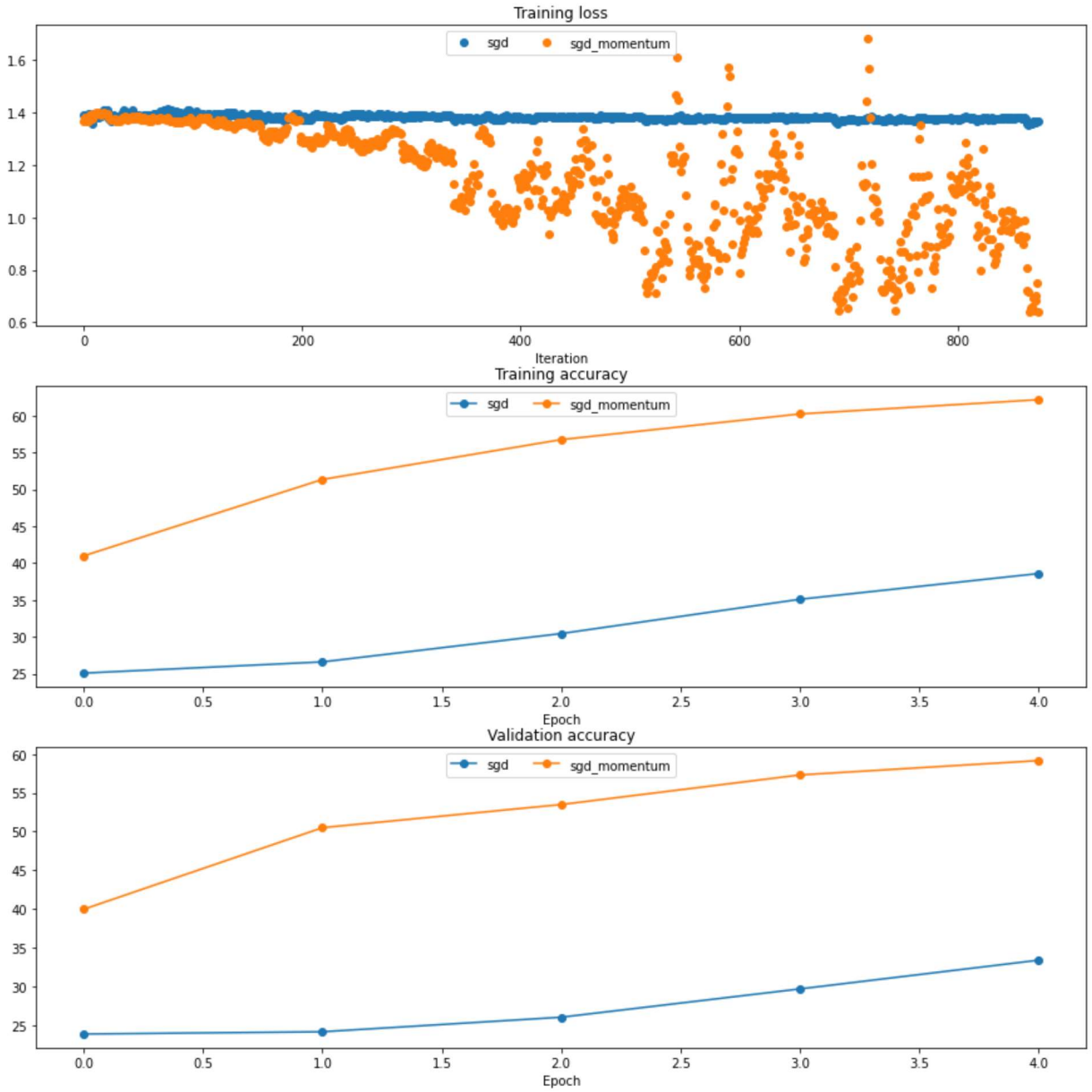
```
solvers = {}

learning_rate = 1e-3# TODO: Tune learning rate
momentum = 0.9# TODO: Tune value of momentum
```

```
for update_rule in ['sgd','sgd_momentum']:
    print('running with ', update_rule)
    model = FullyConnectedNet([256, 128, 64], weight_scale=5e-2, reg=0.0)

    solver = Solver(
        model,
        num_epochs=5,
        batch_size=256,
        update_rule=update_rule if update_rule == 'sgd' else sgd_momentum,
        optim_config={'learning_rate': learning_rate,'momentum':momentum},
        print_every=100,
        dataset_directory=dataset_directory,
        verbose=True
    )

    solvers[update_rule] = solver
    solver.train()

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

for update_rule, solver in list(solvers.items()):
    plt.subplot(3, 1, 1)
    plt.plot(solver.loss_history, 'o', label=update_rule)

    plt.subplot(3, 1, 2)
    plt.plot(solver.train_acc_history, '-o', label=update_rule)

    plt.subplot(3, 1, 3)
    plt.plot(solver.val_acc_history, '-o', label=update_rule)

for i in [1, 2, 3]:
    plt.subplot(3, 1, i)
    plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

```
running with  sgd
(Epoch 0 Batch 1) loss: 1.389960
(Epoch 0 Batch 101) loss: 1.385686
(Epoch 0 / 5) train acc: 25.088889; val_acc: 23.844444
(Epoch 1 Batch 1) loss: 1.383227
(Epoch 1 Batch 101) loss: 1.382757
(Epoch 1 / 5) train acc: 26.608889; val_acc: 24.133333
(Epoch 2 Batch 1) loss: 1.380078
(Epoch 2 Batch 101) loss: 1.380489
(Epoch 2 / 5) train acc: 30.446667; val_acc: 26.011111
(Epoch 3 Batch 1) loss: 1.377332
(Epoch 3 Batch 101) loss: 1.377748
(Epoch 3 / 5) train acc: 35.093333; val_acc: 29.666667
(Epoch 4 Batch 1) loss: 1.374489
(Epoch 4 Batch 101) loss: 1.374129
(Epoch 4 / 5) train acc: 38.582222; val_acc: 33.377778
running with  sgd_momentum
(Epoch 0 Batch 1) loss: 1.366662
(Epoch 0 Batch 101) loss: 1.365053
```

```
(Epoch 0 / 5) train acc: 40.986667; val_acc: 39.966667
(Epoch 1 Batch 1) loss: 1.306781
(Epoch 1 Batch 101) loss: 1.321004
(Epoch 1 / 5) train acc: 51.337778; val_acc: 50.500000
(Epoch 2 Batch 1) loss: 1.112662
(Epoch 2 Batch 101) loss: 1.223287
(Epoch 2 / 5) train acc: 56.733333; val_acc: 53.488889
(Epoch 3 Batch 1) loss: 0.950554
(Epoch 3 Batch 101) loss: 1.168425
(Epoch 3 / 5) train acc: 60.213333; val_acc: 57.311111
(Epoch 4 Batch 1) loss: 0.846914
(Epoch 4 Batch 101) loss: 1.116084
(Epoch 4 / 5) train acc: 62.144444; val_acc: 59.166667
```

```
<ipython-input-344-5aabf33f70fe>:37: MatplotlibDeprecationWarning: Adding an axes us
ing the same arguments as a previous axes currently reuses the earlier instance.  In
a future version, a new instance will always be created and returned.  Meanwhile, th
is warning can be suppressed, and the future behavior ensured, by passing a unique l
abel to each axes instance.
  plt.subplot(3, 1, 1)
<ipython-input-344-5aabf33f70fe>:40: MatplotlibDeprecationWarning: Adding an axes us
ing the same arguments as a previous axes currently reuses the earlier instance.  In
a future version, a new instance will always be created and returned.  Meanwhile, th
is warning can be suppressed, and the future behavior ensured, by passing a unique l
abel to each axes instance.
  plt.subplot(3, 1, 2)
<ipython-input-344-5aabf33f70fe>:43: MatplotlibDeprecationWarning: Adding an axes us
ing the same arguments as a previous axes currently reuses the earlier instance.  In
a future version, a new instance will always be created and returned.  Meanwhile, th
is warning can be suppressed, and the future behavior ensured, by passing a unique l
abel to each axes instance.
  plt.subplot(3, 1, 3)
<ipython-input-344-5aabf33f70fe>:47: MatplotlibDeprecationWarning: Adding an axes us
ing the same arguments as a previous axes currently reuses the earlier instance.  In
a future version, a new instance will always be created and returned.  Meanwhile, th
is warning can be suppressed, and the future behavior ensured, by passing a unique l
abel to each axes instance.
  plt.subplot(3, 1, i)
```




## Task 1.10: Train a good model! (5 points)

Train the best fully-connected model that you can on the dataset, storing your best model in the `best_model` variable. We require you to get at least 55% accuracy on the validation set using a fully-connected net within 50 epochs.

If you are careful it should be possible to get accuracies above 60%, but we don't require it for this part and won't assign extra credit for doing so.

In [345…

```
best_model = None
learning_rate = 1e-3# TODO: Tune learning rate
lr_decay = 0.1# TODO: Tune learning decay
momentum = 0.9# TODO: Tune momentum

model = FullyConnectedNet([100, 100, 100], weight_scale=5e-2)

solver = Solver(
    model,
```

```
    print_every=100,
    num_epochs=20,
    batch_size=batch_size,
    update_rule=sgd_momentum,
    optim_config={'learning_rate': learning_rate, 'lr_decay': lr_decay, 'momentum':
    dataset_directory=dataset_directory,
    verbose=True
)
solver.train()

best_model = model
best_model.params.update(solver.best_params)

# Check accuracy on test data
check_accuracy(best_model, Data(batch_size=batch_size, dataset_directory=dataset_dir
```

```
(Epoch 0 Batch 1) loss: 1.382601
(Epoch 0 Batch 101) loss: 1.368477
(Epoch 0 / 20) train acc: 43.375556; val_acc: 38.422222
(Epoch 1 Batch 1) loss: 1.353818
(Epoch 1 Batch 101) loss: 1.323609
(Epoch 1 / 20) train acc: 49.880000; val_acc: 45.000000
(Epoch 2 Batch 1) loss: 1.263482
(Epoch 2 Batch 101) loss: 1.214537
(Epoch 2 / 20) train acc: 55.873333; val_acc: 53.544444
(Epoch 3 Batch 1) loss: 1.024125
(Epoch 3 Batch 101) loss: 1.217197
(Epoch 3 / 20) train acc: 58.853333; val_acc: 54.811111
(Epoch 4 Batch 1) loss: 0.926713
(Epoch 4 Batch 101) loss: 1.119541
(Epoch 4 / 20) train acc: 59.853333; val_acc: 55.700000
(Epoch 5 Batch 1) loss: 0.801682
(Epoch 5 Batch 101) loss: 1.065447
(Epoch 5 / 20) train acc: 60.462222; val_acc: 56.511111
(Epoch 6 Batch 1) loss: 0.756074
(Epoch 6 Batch 101) loss: 1.031719
(Epoch 6 / 20) train acc: 61.353333; val_acc: 57.377778
(Epoch 7 Batch 1) loss: 0.754533
(Epoch 7 Batch 101) loss: 1.016758
(Epoch 7 / 20) train acc: 62.442222; val_acc: 58.777778
(Epoch 8 Batch 1) loss: 0.785451
(Epoch 8 Batch 101) loss: 1.024918
(Epoch 8 / 20) train acc: 62.735556; val_acc: 58.422222
(Epoch 9 Batch 1) loss: 0.844625
(Epoch 9 Batch 101) loss: 1.028286
(Epoch 9 / 20) train acc: 62.093333; val_acc: 57.566667
(Epoch 10 Batch 1) loss: 0.906949
(Epoch 10 Batch 101) loss: 1.025449
(Epoch 10 / 20) train acc: 61.788889; val_acc: 57.266667
(Epoch 11 Batch 1) loss: 0.943158
(Epoch 11 Batch 101) loss: 1.018501
(Epoch 11 / 20) train acc: 61.726667; val_acc: 57.344444
(Epoch 12 Batch 1) loss: 0.950068
(Epoch 12 Batch 101) loss: 1.009122
(Epoch 12 / 20) train acc: 62.000000; val_acc: 57.944444
(Epoch 13 Batch 1) loss: 0.941261
(Epoch 13 Batch 101) loss: 0.998296
(Epoch 13 / 20) train acc: 62.464444; val_acc: 58.922222
(Epoch 14 Batch 1) loss: 0.924203
(Epoch 14 Batch 101) loss: 0.989199
(Epoch 14 / 20) train acc: 63.004444; val_acc: 60.100000
(Epoch 15 Batch 1) loss: 0.908587
(Epoch 15 Batch 101) loss: 0.980245
(Epoch 15 / 20) train acc: 63.513333; val_acc: 61.033333
(Epoch 16 Batch 1) loss: 0.890865
(Epoch 16 Batch 101) loss: 0.972437
(Epoch 16 / 20) train acc: 63.817778; val_acc: 61.866667
(Epoch 17 Batch 1) loss: 0.877817
```

```
(Epoch 17 Batch 101) loss: 0.963056
(Epoch 17 / 20) train acc: 64.171111; val_acc: 62.588889
(Epoch 18 Batch 1) loss: 0.862684
(Epoch 18 Batch 101) loss: 0.956239
(Epoch 18 / 20) train acc: 64.435556; val_acc: 63.111111
(Epoch 19 Batch 1) loss: 0.848694
(Epoch 19 Batch 101) loss: 0.950080
(Epoch 19 / 20) train acc: 64.695556; val_acc: 63.733333
```

Out[345… 67.27777777777779

# Part 2: Convolutional Neural Network (30 points)

In this second part of the assignment, you need to use Pytorch to build a Convolution Neural Network with 2 Convolution layers and 2 Fully Connected layers. For each convolution layer, we add a pooling layer afterwards for both tasks and a batch normalization layer for Task 2.3. The detailed neural network structure is given for each task.

In [320…

```
import torch
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F
```

## Task 2.1: CNN without batch normalization layers (10 points)

# Network Architecture

In [331…

```
class CNNWithoutBatchNorm(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1,6,3)
        self.pool1 = nn.MaxPool2d(2,2)

        self.conv2 = nn.Conv2d(6,16,5)
        self.pool2 = nn.MaxPool2d(2,2)

        self.fc1 = nn.Linear(16*6*27,256)
        self.fc2 = nn.Linear(256,4)

    def forward(self, x):
        x = F.relu(self.conv1(x))  # input(1,36,120)  output(6,34,118)
        x = self.pool1(x)          # output(6, 17, 59)
        x = F.relu(self.conv2(x))  # output(16, 13, 55)
        x = self.pool2(x)          # output(16, 6, 27)
        x = x.view(-1,16*6*27)

        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return x
```

## Task 2.2: Train CNN without Batch norm(5 points)

As a sanity check for your model, train your model. Observe how it performs in comparison with the previous networks.

In [332…

```
cnnWithoutBatchNormNet = CNNWithoutBatchNorm()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(cnnWithoutBatchNormNet.parameters(), lr=0.01, momentum=0.9)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=2, gamma=0.9)

for epoch in range(10):  # loop over the dataset multiple times
    trainloader = Data(batch_size=batch_size, dataset_directory=dataset_directory).l
    running_loss = 0.0
    for i, (inputs, labels) in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]

        inputs, labels = torch.tensor(inputs).unsqueeze(1).float(), torch.tensor(lab
        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = cnnWithoutBatchNormNet(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        batch_size = len(labels)
    print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / batch_size:.3f}')
    running_loss = 0.0
    scheduler.step()

test_acc = check_accuracy(cnnWithoutBatchNormNet, Data(batch_size=batch_size, datase
print('Finished Training with test acc: ', test_acc)
```

```
[1,   175] loss: 0.819
[2,   175] loss: 0.646
[3,   175] loss: 0.635
[4,   175] loss: 0.580
[5,   175] loss: 0.560
[6,   175] loss: 0.526
[7,   175] loss: 0.508
[8,   175] loss: 0.482
[9,   175] loss: 0.466
[10,   175] loss: 0.458
<ipython-input-69-ba0def6a1605>:20: UserWarning: To copy construct from a tensor, it
is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach
().requires_grad_(True), rather than torch.tensor(sourceTensor).
  scores = model(torch.tensor(inputs)).detach().numpy()
Finished Training with test acc:  61.044444444444444
```

## Task 2.3: CNN using batch normalization layers (5 points)

# Network Architecture

In [337…

```
class CNNWithBatchNorm(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1,6,3)
        self.pool1 = nn.MaxPool2d(2,2)
        self.bn1 = nn.BatchNorm2d(6)

        self.conv2 = nn.Conv2d(6,16,5)
        self.pool2 = nn.MaxPool2d(2,2)
        self.bn2 = nn.BatchNorm2d(16)

        self.fc1 = nn.Linear(16*6*27,256)
        self.fc2 = nn.Linear(256,4)

    def forward(self, x):
        x = F.relu(self.conv1(x))  # input(1,36,120)  output(6,34,118)
        x = self.pool1(x)          # output(6, 17, 59)
        x = self.bn1(x)

        x = F.relu(self.conv2(x))  # output(16, 13, 55)
        x = self.pool2(x)          # output(16, 6, 27)
        x = self.bn2(x)

        x = x.view(-1,16*6*27)
```

```
        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return x
```

## Task 2.4: Train CNN with Batch norm(5 points)

As a sanity check for your model, train your model. Observe how it performs in comparison with the previous networks.

In [338…

```
cnnWithBatchNormNet = CNNWithBatchNorm()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(cnnWithBatchNormNet.parameters(), lr=0.01, momentum=0.9)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=2, gamma=0.9)

for epoch in range(10):  # loop over the dataset multiple times
    trainloader = Data(batch_size=batch_size, dataset_directory=dataset_directory).l
    running_loss = 0.0
    for i, (inputs, labels) in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]

        inputs, labels = torch.tensor(inputs).unsqueeze(1).float(), torch.tensor(lab
        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = cnnWithBatchNormNet(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        batch_size = len(labels)
    print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / batch_size:.3f}')
    running_loss = 0.0
    scheduler.step()

test_acc = check_accuracy(cnnWithBatchNormNet, Data(batch_size=batch_size, dataset_d
print('Finished Training with test acc: ', test_acc)
```

```
[1,   175] loss: 0.596
[2,   175] loss: 0.558
[3,   175] loss: 0.500
[4,   175] loss: 0.451
[5,   175] loss: 0.424
[6,   175] loss: 0.405
[7,   175] loss: 0.392
[8,   175] loss: 0.386
[9,   175] loss: 0.365
[10,   175] loss: 0.333
<ipython-input-69-ba0def6a1605>:20: UserWarning: To copy construct from a tensor, it
is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().detach
().requires_grad_(True), rather than torch.tensor(sourceTensor).
  scores = model(torch.tensor(inputs)).detach().numpy()
Finished Training with test acc:  76.0
```

## Task 2.4: Visualize some of the convolutional filters (5 points)

In [346…

```
# # You can visualize the first-layer convolutional filters from the trained network
from utils.vis_utils import visualize_grid

conv1_weights = cnnWithBatchNormNet.conv1.weight.data.cpu().numpy()# TODO: weights o

grid = visualize_grid(conv1_weights)
plt.imshow(grid.astype('uint8'))
plt.axis('off')
plt.gcf().set_size_inches(5, 5)
plt.show()
```

## Part 3: What did you observe from the previous experiments?

**Task 3.1: Calculate the number of learnable parameters in both the CNN and the Fully Connected Network of Task 1.10 and explain which is better and why? (5 points)**

params in CNN without batch norm = 667,052 = (1*3*3*6+6)+0+(6*5*5*16+16)+0+(16*6*27*256)+256*4

params in CNN with batch norm = 667,052

params in FCN = 452,400 = 36*120*100 + 100*100 + 100*100 + 100*4

Obviously, CNN with batch norm has better performance than the others have. By checking the number of learnable parameters in these two types of neural networks, it tells us that CNN can extract more features from the same dataset. This advantage gives CNN a more ambitious learning rate while preventing from overfitting. In this experiments, the learning rate of CNN is 0.01 while the learning rate of FCN is 0.001 with the same optimization method.

**Task 3.2: Compare SGD with SGD+momentum and report your observations? Why does one work better than the other? (5 points)**

Compare both optimization methods, SGD with momentum performs better than SGD with no momentum in both accuracy and speed of loss decline. The momentum adds an additional

hyperparameter that controls the amount of historical momentum to include in the update equation to accelerate the optimization process. It reduces the chance of bouncing around a specific search space, such as a plateaus space, when searches the direction of gradient descent.

# Part 4: Improving the CNN (Extra credit: 5 points)

In [ ]:

```
# Improve the CNN trained in the previous step
# You can use one or more of the following techniques or something else
# 1. Change the architecture: Increase the depth, change the kernel sizes etc
# 2. Fine tune the hyper parameters
# 3. Use a better optimizer
```