Problem 2: Incorporating CNNs

- Learning Objective: In this problem, you will learn how to deeply understand how Convolutional Neural Networks work by implementing one.
- Provided Code: We provide the skeletons of classes you need to complete. Forward checking and gradient checkings are provided for verifying your implementation as well.
- TODOs: you will implement a Convolutional Layer and a MaxPooling Layer to improve on your classification results in part 1.

In [ ]:
```
from lib.mlp.fully_conn import *
from lib.mlp.layer_utils import *
from lib.mlp.datasets import *
from lib.mlp.train import *
from lib.cnn.layer_utils import *
from lib.cnn.cnn_models import *
from lib.grad_check import *
from lib.optim import *
import numpy as np
import matplotlib.pyplot as plt

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

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

Loading the data (SVHN)

Run the following code block to download SVHN dataset and load in the properly splitted SVHN data. The script `get_datasets.sh` use `wget` to download the SVHN dataset. If you have a trouble with executing `get_datasets.sh` , you can manually download the dataset and extract files.

In [ ]:
```
!./get_datasets.sh
# !get_datasets.sh for windows users
```

```
--2022-02-20 18:53:53--  http://ufldl.stanford.edu/housenumbers/tr
ain_32x32.mat
Resolving ufldl.stanford.edu (ufldl.stanford.edu)... 171.64.68.10
Connecting to ufldl.stanford.edu (ufldl.stanford.edu)|171.64.68.1
0|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 182040794 (174M) [text/plain]
Saving to: ‘data/train_32x32.mat’

train_32x32.mat     100%[===================>] 173.61M   657KB/s
in 5m 5s

2022-02-20 18:58:58 (583 KB/s) - ‘data/train_32x32.mat’ saved [182
040794/182040794]

--2022-02-20 18:58:58--  http://ufldl.stanford.edu/housenumbers/te
st_32x32.mat
Resolving ufldl.stanford.edu (ufldl.stanford.edu)... 171.64.68.10
Connecting to ufldl.stanford.edu (ufldl.stanford.edu)|171.64.68.1
0|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 64275384 (61M) [text/plain]
Saving to: ‘data/test_32x32.mat’

test_32x32.mat      100%[===================>]  61.30M   591KB/s
in 1m 51s

2022-02-20 19:00:49 (567 KB/s) - ‘data/test_32x32.mat’ saved [6427
5384/64275384]
```

In [ ]:
```
data = SVHN_data()
for k, v in data.items():
    print ("Name: {} Shape: {}".format(k, v.shape))
```

```
Name: data_train Shape: (70000, 32, 32, 3)
Name: labels_train Shape: (70000,)
Name: data_val Shape: (3257, 32, 32, 3)
Name: labels_val Shape: (3257,)
Name: data_test Shape: (26032, 32, 32, 3)
Name: labels_test Shape: (26032,)
```

Convolutional Neural Networks

We will use convolutional neural networks to try to improve on the results from Problem 1. Convolutional layers make the assumption that local pixels are more important for prediction than far-away pixels. This allows us to form networks that are robust to small changes in positioning in images.

Convolutional Layer Output size calculation [2pts]

As you have learned, two important parameters of a convolutional layer are its stride and padding. To warm up, we will need to calculate the output size of a convolutional layer given its stride and padding. To do this, open the `lib/cnn/layer_utils.py` file and fill out the TODO section in the `get_output_size` function in the ConvLayer2D class.

Implement your function so that it returns the correct size as indicated by the block below.

In [ ]:
```
%reload_ext autoreload

input_image = np.zeros([32, 28, 28, 3]) # a stack of 32 28 by 28 r
gb images

in_channels = input_image.shape[-1] #must agree with the last dime
nsion of the input image
k_size = 4
n_filt = 16

conv_layer = ConvLayer2D(in_channels, k_size, n_filt, stride=2, pa
dding=3)
output_size = conv_layer.get_output_size(input_image.shape)

print("Received {} and expected [32, 16, 16, 16]".format(output_si
ze))
```

```
Received [32, 16, 16, 16] and expected [32, 16, 16, 16]
```

Convolutional Layer Forward Pass [5pts]

Now, we will implement the forward pass of a convolutional layer. Fill in the TODO block in the `forward` function of the ConvLayer2D class.

```
In [ ]: %reload_ext autoreload

        # Test the convolutional forward function
        input_image = np.linspace(-0.1, 0.4, num=1*8*8*1).reshape([1, 8,
        8, 1]) # a single 8 by 8 grayscale image
        in_channels, k_size, n_filt = 1, 5, 2

        weight_size = k_size*k_size*in_channels*n_filt
        bias_size = n_filt

        single_conv = ConvLayer2D(in_channels, k_size, n_filt, stride=1, p
        adding=0, name="conv_test")

        w = np.linspace(-0.2, 0.2, num=weight_size).reshape(k_size, k_siz
        e, in_channels, n_filt)
        b = np.linspace(-0.3, 0.3, num=bias_size)

        single_conv.params[single_conv.w_name] = w
        single_conv.params[single_conv.b_name] = b

        out = single_conv.forward(input_image)

        print("Received output shape: {}, Expected output shape: (1, 4, 4,
        2)".format(out.shape))

        correct_out = np.array([[
           [[-0.03874312, 0.57000324],
           [-0.03955296, 0.57081309],
           [-0.04036281, 0.57162293],
           [-0.04117266, 0.57243278]],

          [[-0.0452219, 0.57648202],
           [-0.04603175, 0.57729187],
           [-0.04684159, 0.57810172],
           [-0.04765144, 0.57891156]],

          [[-0.05170068, 0.5829608 ],
           [-0.05251053, 0.58377065],
           [-0.05332038, 0.5845805 ],
           [-0.05413022, 0.58539035]],

          [[-0.05817946, 0.58943959],
           [-0.05898931, 0.59024943],
           [-0.05979916, 0.59105928],
           [-0.06060901, 0.59186913]]]])


        # Compare your output with the above pre-computed ones.
        # The difference should not be larger than 1e-8
        print ("Difference: ", rel_error(out, correct_out))
```

```
Received output shape: (1, 4, 4, 2), Expected output shape: (1, 4,
4, 2)
Difference:  5.110565335399418e-08
```

Conv Layer Backward [5pts]

Now complete the backward pass of a convolutional layer. Fill in the TODO block in the `backward` function of the ConvLayer2D class. Check you results with this code and expect differences of less than 1e-6.

In [ ]:
```
%reload_ext autoreload

# Test the conv backward function
img = np.random.randn(15, 8, 8, 3)
w = np.random.randn(4, 4, 3, 12)
b = np.random.randn(12)
dout = np.random.randn(15, 4, 4, 12)

single_conv = ConvLayer2D(input_channels=3, kernel_size=4, number_
filters=12, stride=2, padding=1, name="conv_test")
single_conv.params[single_conv.w_name] = w
single_conv.params[single_conv.b_name] = b

dimg_num = eval_numerical_gradient_array(lambda x: single_conv.for
ward(img), img, dout)
dw_num = eval_numerical_gradient_array(lambda w: single_conv.forwa
rd(img), w, dout)
db_num = eval_numerical_gradient_array(lambda b: single_conv.forwa
rd(img), b, dout)

out = single_conv.forward(img)

dimg = single_conv.backward(dout)
dw = single_conv.grads[single_conv.w_name]
db = single_conv.grads[single_conv.b_name]

# The error should be around 1e-8
print("dimg Error: ", rel_error(dimg_num, dimg))
# The errors should be around 1e-8
print("dw Error: ", rel_error(dw_num, dw))
print("db Error: ", rel_error(db_num, db))
# The shapes should be same
print("dimg Shape: ", dimg.shape, img.shape)
```

```
dimg Error:  2.6294797003498437e-07
dw Error:  1.5793360396074264e-08
db Error:  3.029810082779965e-10
dimg Shape:  (15, 8, 8, 3) (15, 8, 8, 3)
```

Max pooling Layer

Now we will implement maxpooling layers, which can help to reduce the image size while preserving the overall structure of the image.

Forward Pass max pooling [5pts]

Fill out the TODO block in the `forward` function of the MaxPoolingLayer class.

In [ ]:
```
# Test the convolutional forward function
input_image = np.linspace(-0.1, 0.4, num=64).reshape([1, 8, 8, 1])
# a single 8 by 8 grayscale image

maxpool= MaxPoolingLayer(pool_size=4, stride=2, name="maxpool_tes
t")
out = maxpool.forward(input_image)

print("Received output shape: {}, Expected output shape: (1, 3, 3,
1)".format(out.shape))

correct_out = np.array([[
   [[0.11428571],
   [0.13015873],
   [0.14603175]],

  [[0.24126984],
   [0.25714286],
   [0.27301587]],

  [[0.36825397],
   [0.38412698],
   [0.4       ]]]])

# Compare your output with the above pre-computed ones.
# The difference should not be larger than 1e-8
print ("Difference: ", rel_error(out, correct_out))
```

```
Received output shape: (1, 3, 3, 1), Expected output shape: (1, 3,
3, 1)
Difference:  1.8750000280978013e-08
```

Backward Pass Max pooling [5pts]

Fill out the `backward` function in the MaxPoolingLayer class.

In [ ]:
```
img = np.random.randn(15, 8, 8, 3)

dout = np.random.randn(15, 3, 3, 3)

maxpool= MaxPoolingLayer(pool_size=4, stride=2, name="maxpool_tes
t")

dimg_num = eval_numerical_gradient_array(lambda x: maxpool.forward
(img), img, dout)

out = maxpool.forward(img)
dimg = maxpool.backward(dout)

# The error should be around 1e-8
print("dimg Error: ", rel_error(dimg_num, dimg))
# The shapes should be same
print("dimg Shape: ", dimg.shape, img.shape)
```

```
dimg Error:  3.2772667075661996e-12
dimg Shape:  (15, 8, 8, 3) (15, 8, 8, 3)
```

Test a Small Fully Connected Network [3pts]

Please find the `TestCNN` class in `lib/cnn/cnn_models.py` . Again you only need to complete few lines of code in the TODO block. Please design a Convolutional --> Maxpool --> flatten --> fc network where the shapes of parameters match the given shapes. Please insert the corresponding names you defined for each layer to param_name_w, and param_name_b respectively. Here you only modify the param_name part, the _w, and _b are automatically assigned during network setup.

In [ ]:
```
%reload_ext autoreload

seed = 1234
np.random.seed(seed=seed)

model = TestCNN()
loss_func = cross_entropy()

B, H, W, iC = 4, 8, 8, 3 #batch, height, width, in_channels
k = 3 #kernel size
oC, Hi, O = 3, 27, 5 # out channels, Hidden Layer input, Output si
ze
std = 0.02
x = np.random.randn(B,H,W,iC)
y = np.random.randint(O, size=B)

print ("Testing initialization ... ")

###################################################
# TODO: param_name should be replaced accordingly  #
###################################################
w1_std = abs(model.net.get_params("conv_w").std() - std)
b1 = model.net.get_params("conv_b").std()
w2_std = abs(model.net.get_params("fc_w").std() - std)
b2 = model.net.get_params("fc_b").std()
###################################################
#                 END OF YOUR CODE                 #
###################################################

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 ("Passed!")

print ("Testing test-time forward pass ... ")
w1 = np.linspace(-0.7, 0.3, num=k*k*iC*oC).reshape(k,k,iC,oC)
w2 = np.linspace(-0.2, 0.2, num=Hi*O).reshape(Hi, O)
b1 = np.linspace(-0.6, 0.2, num=oC)
b2 = np.linspace(-0.9, 0.1, num=O)

###################################################
# TODO: param_name should be replaced accordingly  #
###################################################
model.net.assign("conv_w", w1)
model.net.assign("conv_b", b1)
model.net.assign("fc_w", w2)
model.net.assign("fc_b", b2)
###################################################
#                 END OF YOUR CODE                 #
###################################################

feats = np.linspace(-5.5, 4.5, num=B*H*W*iC).reshape(B,H,W,iC)
scores = model.forward(feats)
correct_scores = np.asarray([[-13.85107294, -11.52845818,  -9.2058
4342,  -6.88322866,  -4.5606139 ],
 [-11.44514171, -10.21200524 , -8.97886878 , -7.74573231 , -6.5125
9584],
 [ -9.03921048,  -8.89555231 , -8.75189413 , -8.60823596,  -8.4645
7778],
 [ -6.63327925 , -7.57909937 , -8.52491949 , -9.4707396 , -10.4165
5972]])
scores_diff = np.sum(np.abs(scores - correct_scores))
assert scores_diff < 1e-6, "Your implementation might be wrong!"
print ("Passed!")

print ("Testing the loss ...",)
y = np.asarray([0, 2, 1, 4])
loss = loss_func.forward(scores, y)
dLoss = loss_func.backward()
correct_loss = 4.56046848799693
assert abs(loss - correct_loss) < 1e-10, "Your implementation migh
t be wrong!"
print ("Passed!")

print ("Testing the gradients (error should be no larger than 1e-
6) ...")
din = model.backward(dLoss)
for layer in model.net.layers:
    if not layer.params:
        continue
    for name in sorted(layer.grads):
        f = lambda _: loss_func.forward(model.forward(feats), y)
        grad_num = eval_numerical_gradient(f, layer.params[name],
verbose=False)
        print ('%s relative error: %.2e' % (name, rel_error(grad_n
um, layer.grads[name])))
```

```
Testing initialization ...
Passed!
Testing test-time forward pass ...
Passed!
Testing the loss ...
Passed!
Testing the gradients (error should be no larger than 1e-6) ...
conv_b relative error: 3.95e-09
conv_w relative error: 9.10e-10
fc_b relative error: 9.76e-11
fc_w relative error: 3.89e-07
```

Training the Network [25pts]

In this section, we defined a `SmallConvolutionalNetwork` class for you to fill in the TODO block in `lib/cnn/cnn_models.py`.

Here please design a network with at most two convolutions and two maxpooling layers (you may use less). You can adjust the parameters for any layer, and include layers other than those listed above that you have implemented. You are also free to select any optimizer you have implemented (with any learning rate).

Try to find a combination that is able to achieve 88% validation accuracy.

In [ ]:
```
# Arrange the data
data_dict = {
    "data_train": (data["data_train"], data["labels_train"]),
    "data_val": (data["data_val"], data["labels_val"]),
    "data_test": (data["data_test"], data["labels_test"])
}
```

In [ ]:
```
print("Data shape:", data_dict["data_train"][0].shape)
print("Flattened data input size:", np.prod(data["data_train"].sha
pe[1:]))
print("Number of data classes:", max(data['labels_train']) + 1)
```

```
Data shape: (70000, 32, 32, 3)
Flattened data input size: 3072
Number of data classes: 10
```

In [ ]:
```
%reload_ext autoreload

seed = 123
np.random.seed(seed=seed)

model = SmallConvolutionalNetwork()
loss_f = cross_entropy()


results = None
####################################################################
###########
# TODO: Use the train_net function you completed to train a networ
k         #
####################################################################
###########
optimizer =  Adam(model.net, 5e-4)

batch_size = 128
epochs = 16
lr_decay = .999
lr_decay_every = 10

####################################################################
###########
#                             END OF YOUR CODE
#
####################################################################
###########
results = train_net(data_dict, model, loss_f, optimizer, batch_siz
e, epochs,
                    lr_decay, lr_decay_every, show_every=100, verb
ose=True)
opt_params, loss_hist, train_acc_hist, val_acc_hist = results
```

```
(Iteration 1 / 8736) loss: 2.3025736957897607
(Iteration 101 / 8736) loss: 2.2591803444014857
(Iteration 201 / 8736) loss: 2.205958347853379
(Iteration 301 / 8736) loss: 1.696692227506987
(Iteration 401 / 8736) loss: 1.585874639311335
(Iteration 501 / 8736) loss: 1.1732460202422692
(Epoch 1 / 16) Training Accuracy: 0.6985857142857143, Validation A
ccuracy: 0.6911268038071845
(Iteration 601 / 8736) loss: 0.9067702846900861
(Iteration 701 / 8736) loss: 0.8581504522006463
(Iteration 801 / 8736) loss: 0.8141147996311852
(Iteration 901 / 8736) loss: 0.7086537658894622
(Iteration 1001 / 8736) loss: 0.6570129824709264
(Epoch 2 / 16) Training Accuracy: 0.7880571428571429, Validation A
ccuracy: 0.7817009517961314
(Iteration 1101 / 8736) loss: 0.7045922069347934
(Iteration 1201 / 8736) loss: 0.6741389875514159
(Iteration 1301 / 8736) loss: 0.8318776785027838
(Iteration 1401 / 8736) loss: 0.6473599770112083
(Iteration 1501 / 8736) loss: 0.6004213433716638
(Iteration 1601 / 8736) loss: 0.7449792262861505
(Epoch 3 / 16) Training Accuracy: 0.8079857142857143, Validation A
ccuracy: 0.8013509364445809
(Iteration 1701 / 8736) loss: 0.751971183148553
(Iteration 1801 / 8736) loss: 0.6156255371489677
(Iteration 1901 / 8736) loss: 0.9059816703212304
(Iteration 2001 / 8736) loss: 0.5304335094133626
(Iteration 2101 / 8736) loss: 0.7175114891569875
(Epoch 4 / 16) Training Accuracy: 0.8172285714285714, Validation A
ccuracy: 0.8087196806877495
(Iteration 2201 / 8736) loss: 0.8410531598476951
(Iteration 2301 / 8736) loss: 0.7585857692906276
(Iteration 2401 / 8736) loss: 0.47014066515093267
(Iteration 2501 / 8736) loss: 0.5988181223711321
(Iteration 2601 / 8736) loss: 0.6788401867154363
(Iteration 2701 / 8736) loss: 0.6207954511009204
(Epoch 5 / 16) Training Accuracy: 0.8238285714285715, Validation A
ccuracy: 0.8114829597789377
(Iteration 2801 / 8736) loss: 0.6002333214760249
(Iteration 2901 / 8736) loss: 0.6091293238179888
(Iteration 3001 / 8736) loss: 0.4068201242173936
(Iteration 3101 / 8736) loss: 0.49572492395683615
(Iteration 3201 / 8736) loss: 0.6918847188367822
(Epoch 6 / 16) Training Accuracy: 0.8272857142857143, Validation A
ccuracy: 0.8151673319005219
(Iteration 3301 / 8736) loss: 0.5532325241596414
(Iteration 3401 / 8736) loss: 0.635359698032371
(Iteration 3501 / 8736) loss: 0.6353531976662823
(Iteration 3601 / 8736) loss: 0.6393259819620984
(Iteration 3701 / 8736) loss: 0.5753410248775878
(Iteration 3801 / 8736) loss: 0.6371156410274871
(Epoch 7 / 16) Training Accuracy: 0.8285285714285714, Validation A
ccuracy: 0.8206938900828984
(Iteration 3901 / 8736) loss: 0.5374879320063028
(Iteration 4001 / 8736) loss: 0.5908884212648675
(Iteration 4101 / 8736) loss: 0.422254274204206
(Iteration 4201 / 8736) loss: 0.5096793619058617
(Iteration 4301 / 8736) loss: 0.5587539888282044
(Epoch 8 / 16) Training Accuracy: 0.8375, Validation Accuracy: 0.8
271415412956709
```

```
(Iteration 4401 / 8736) loss: 0.4968727650898371
(Iteration 4501 / 8736) loss: 0.47184823736780757
(Iteration 4601 / 8736) loss: 0.42854850569170233
(Iteration 4701 / 8736) loss: 0.4626370296647558
(Iteration 4801 / 8736) loss: 0.42469070002814885
(Iteration 4901 / 8736) loss: 0.5455848065547951
(Epoch 9 / 16) Training Accuracy: 0.8409857142857143, Validation A
ccuracy: 0.8256063862450107
(Iteration 5001 / 8736) loss: 0.7899093624945172
(Iteration 5101 / 8736) loss: 0.5502730511234887
(Iteration 5201 / 8736) loss: 0.5958190731077371
(Iteration 5301 / 8736) loss: 0.8316381022577392
(Iteration 5401 / 8736) loss: 0.5574427005410697
(Epoch 10 / 16) Training Accuracy: 0.8434142857142857, Validation
Accuracy: 0.828062634326067
Decaying learning rate of the optimizer to 0.0004995
(Iteration 5501 / 8736) loss: 0.6606717023067376
(Iteration 5601 / 8736) loss: 0.7727548567663055
(Iteration 5701 / 8736) loss: 0.4211305318002012
(Iteration 5801 / 8736) loss: 0.47555146604220033
(Iteration 5901 / 8736) loss: 0.6883270901486119
(Iteration 6001 / 8736) loss: 0.6948954857087859
(Epoch 11 / 16) Training Accuracy: 0.8430571428571428, Validation
Accuracy: 0.829290758366595
(Iteration 6101 / 8736) loss: 0.39826275592302396
(Iteration 6201 / 8736) loss: 0.40957773404898706
(Iteration 6301 / 8736) loss: 0.44472449216358445
(Iteration 6401 / 8736) loss: 0.526226869639065
(Iteration 6501 / 8736) loss: 0.5583256197813617
(Epoch 12 / 16) Training Accuracy: 0.8460714285714286, Validation
Accuracy: 0.8338962235185754
(Iteration 6601 / 8736) loss: 0.5051805034432353
(Iteration 6701 / 8736) loss: 0.6085562374413177
(Iteration 6801 / 8736) loss: 0.5237933084366818
(Iteration 6901 / 8736) loss: 0.598182529623854
(Iteration 7001 / 8736) loss: 0.44914766540014744
(Epoch 13 / 16) Training Accuracy: 0.8454571428571429, Validation
Accuracy: 0.8314399754375192
(Iteration 7101 / 8736) loss: 0.4357485755565206
(Iteration 7201 / 8736) loss: 0.429171839424588
(Iteration 7301 / 8736) loss: 0.4060706781566854
(Iteration 7401 / 8736) loss: 0.6180231202466824
(Iteration 7501 / 8736) loss: 0.5082554995922834
(Iteration 7601 / 8736) loss: 0.3170641554006881
(Epoch 14 / 16) Training Accuracy: 0.8507857142857143, Validation
Accuracy: 0.8348173165489714
(Iteration 7701 / 8736) loss: 0.644127312925496
(Iteration 7801 / 8736) loss: 0.47936058724260167
(Iteration 7901 / 8736) loss: 0.5449422933782173
(Iteration 8001 / 8736) loss: 0.4540320500257258
(Iteration 8101 / 8736) loss: 0.6761736033068332
(Epoch 15 / 16) Training Accuracy: 0.8532571428571428, Validation
Accuracy: 0.8375805956401596
(Iteration 8201 / 8736) loss: 0.4650179127314918
(Iteration 8301 / 8736) loss: 0.43590194510800867
(Iteration 8401 / 8736) loss: 0.46112058165194914
(Iteration 8501 / 8736) loss: 0.487792454632618
(Iteration 8601 / 8736) loss: 0.38261648168934437
(Iteration 8701 / 8736) loss: 0.5380163441115201
(Epoch 16 / 16) Training Accuracy: 0.854, Validation Accuracy: 0.8
```

```
403438747313479
```

Run the code below to generate the training plots.

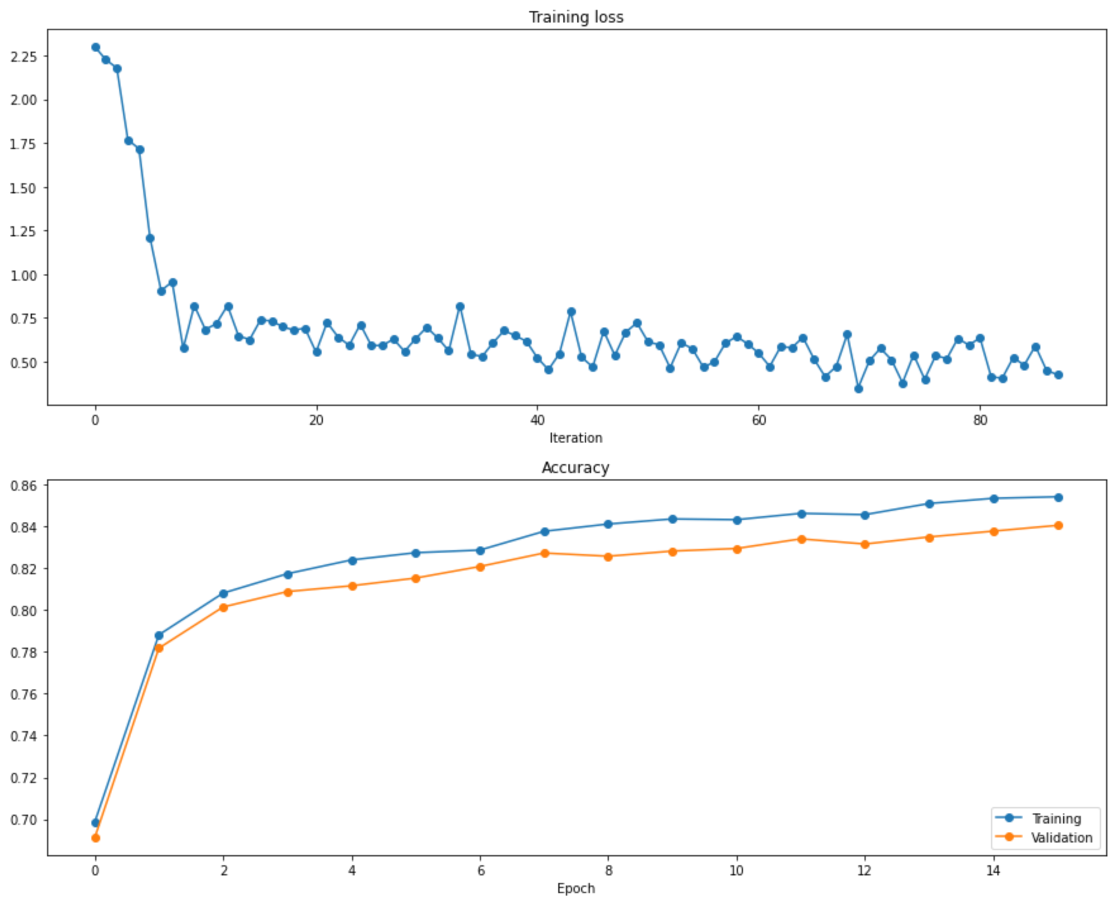
In [ ]:
```python
%reload_ext autoreload

opt_params, loss_hist, train_acc_hist, val_acc_hist = results

# Plot the learning curves
plt.subplot(2, 1, 1)
plt.title('Training loss')
loss_hist_ = loss_hist[1::100]  # sparse the curve a bit
plt.plot(loss_hist_, '-o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(train_acc_hist, '-o', label='Training')
plt.plot(val_acc_hist, '-o', label='Validation')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)

plt.show()
```



Visualizing Layers [5pts]

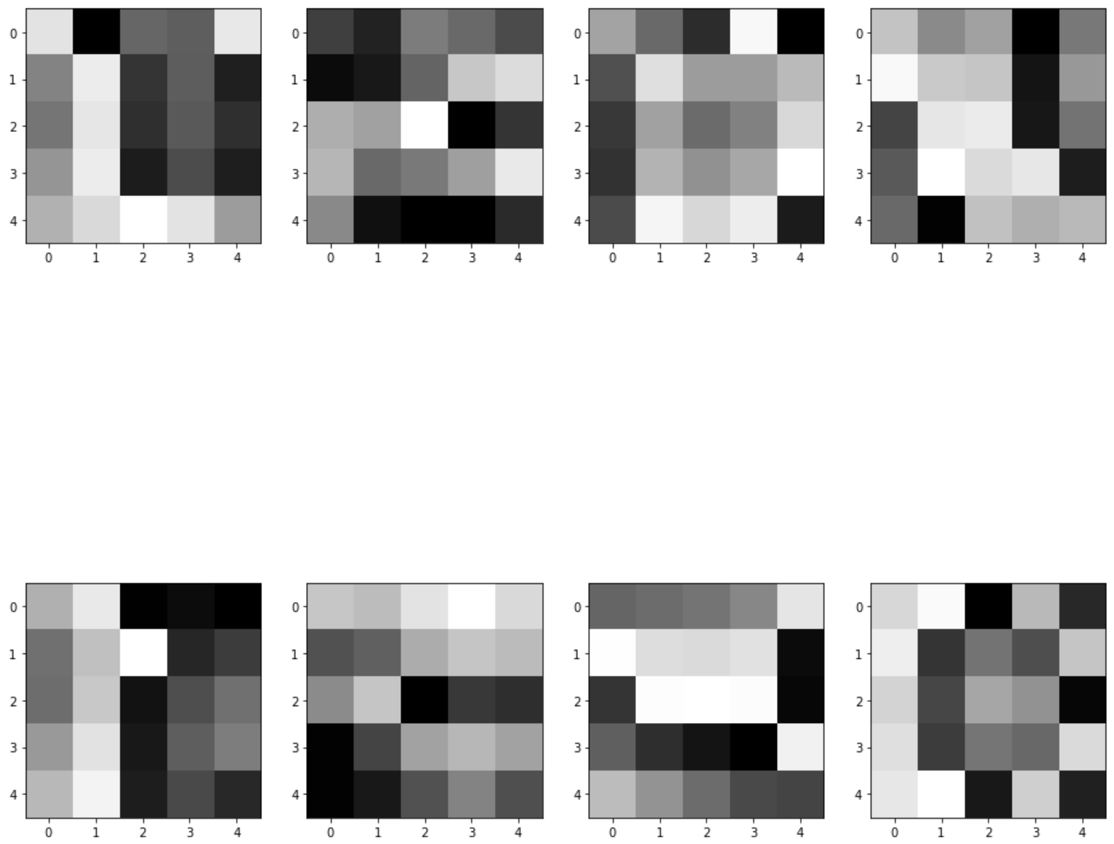
An interesting finding from early research in convolutional networks was that the learned convolutions resembled filters used for things like edge detection. Complete the code below to visualize the filters in the first convolutional layer of your best model.

```
In [ ]: im_array = None
        nrows, ncols = None, None

        ###################################################
        # TODO: read the weights in the convolutional     #
        # layer and reshape them to a grid of images to   #
        # view with matplotlib.                           #
        ###################################################
        filters = model.net.get_params("conv1_w")
        filters = np.sum(filters, axis=2) * 255
        filters = filters.astype(np.uint8)
        filters_shaped = np.moveaxis(filters, -1, 0).tolist()

        _, axs = plt.subplots(2, 4, figsize = (16, 16))
        axs = axs.flatten()
        for img, ax in zip(filters_shaped, axs):
            ax.imshow(img)
        plt.show()
        ###################################################
        #                 END OF YOUR CODE                #
        ###################################################

        # plt.imshow(im_array)
```



Inline Question: Comment below on what kinds of filters you see. Include your response in your submission [5pts]

Consider the darker areas corresponding to background/noise while whiter represents the edges and greyish representing some kind of noise, below is my interpretation of the 8 filters from the 1st convolutional layer (going from left to right, row by row)

1. The first filter looks like it is highlighting a slight curve which might correspond to the curved areas of 6, 8, 9 or 0.
2. The filter looks like to have captured more noise than the others with a single dot denoted by the white pixel. It can be either a noise or some information captured from the dataset.
3. It seems like the L-shaped area is corresponding to a edge of the digit 5
4. The bright parts of the filter looks like a small curving edge which might be corresponding to regions in the digit 6.
5. The bright vertical pixels seems to be representing a straight vertical line. It might be corresponding to the same in digits - 1, 5, 7, 9.
6. The bright horizontal line seems to be corresponding to an horizontal edge. Like the top line of 5. or 7
7. The curve looking area seems to be corresponding to a downward curve or a small curve like in the ending of digit 5.
8. The kind of C shaped bright pixels looks like edges with a bit of sides coming out of it. Like in the middle part of 5.

Note: I have used digits in the explanation to aid the visualization of the reader. It might not correspond to that actual digit.

Submission

Please prepare a PDF document `problem_2_solution.pdf` in the root directory of this repository with all plots and inline answers of your solution. Concretely, the document should contain the following items in strict order:

1. Training loss / accuracy curves for CNN training
2. Visualization of convolutional filters
3. Answers to inline questions about convolutional filters

Note that you still need to submit the jupyter notebook with all generated solutions. We will randomly pick submissions and check that the plots in the PDF and in the notebook are equivalent.