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.train import *
from lib.cnn.layer_utils import *
from lib.cnn.cnn_models import *
from lib.datasets 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-modules-in
%load_ext autoreload
%autoreload 2
```

Loading the data (CIFAR-100 with 20 superclasses)

In this homework, we will be classifying images from the CIFAR-100 dataset into the 20 superclasses. More information about the CIFAR-100 dataset and the 20 superclasses can be found here.

Download the CIFAR-100 data files here, and save the `.mat` files to the `data/cifar100` directory.

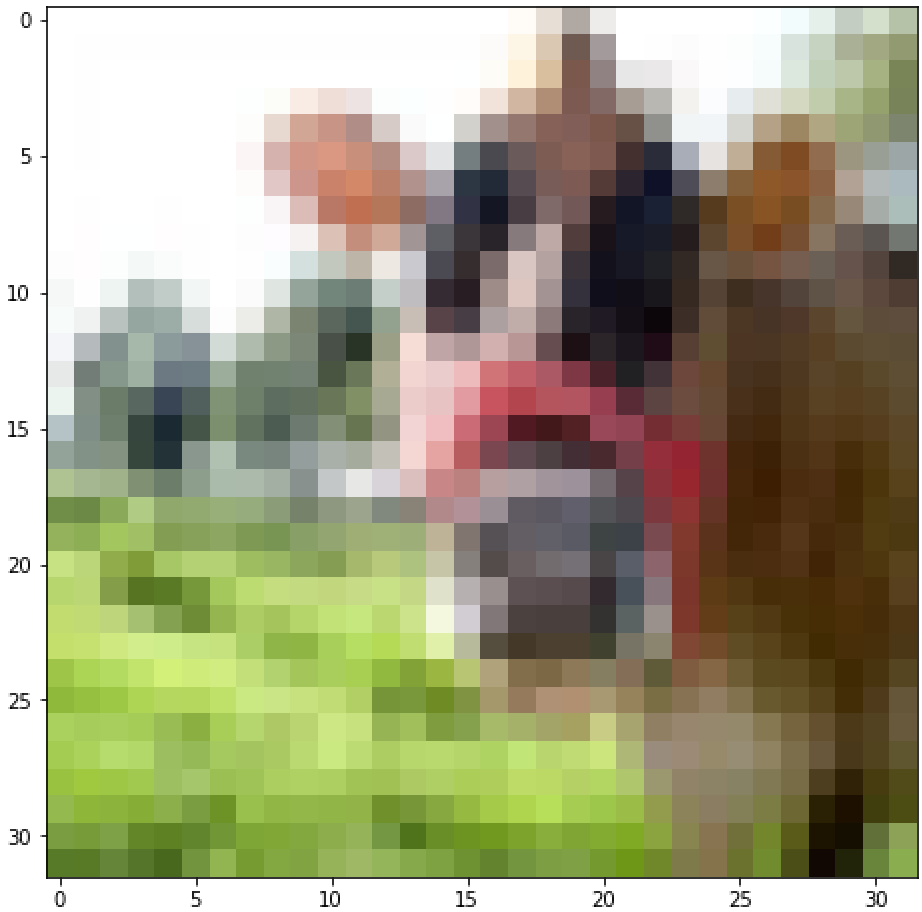
In [ ]:
```
data = CIFAR100_data('data/cifar100/')
for k, v in data.items():
    if type(v) == np.ndarray:
        print ("Name: {} Shape: {}, {}".format(k, v.shape, type(v)))
    else:
        print("{}: {}".format(k, v))
label_names = data['label_names']
mean_image = data['mean_image'][0]
std_image = data['std_image'][0]
```

```
Name: data_train Shape: (40000, 32, 32, 3), <class 'numpy.ndarray'>
Name: labels_train Shape: (40000,), <class 'numpy.ndarray'>
Name: data_val Shape: (10000, 32, 32, 3), <class 'numpy.ndarray'>
Name: labels_val Shape: (10000,), <class 'numpy.ndarray'>
Name: data_test Shape: (10000, 32, 32, 3), <class 'numpy.ndarray'>
Name: labels_test Shape: (10000,), <class 'numpy.ndarray'>
label_names: ['aquatic_mammals', 'fish', 'flowers', 'food_containers',
'fruit_and_vegetables', 'household_electrical_devices', 'household_furni
ture', 'insects', 'large_carnivores', 'large_man-made_outdoor_things',
'large_natural_outdoor_scenes', 'large_omnivores_and_herbivores', 'mediu
m_mammals', 'non-insect_invertebrates', 'people', 'reptiles', 'small_mam
mals', 'trees', 'vehicles_1', 'vehicles_2']
Name: mean_image Shape: (1, 1, 1, 3), <class 'numpy.ndarray'>
Name: std_image Shape: (1, 1, 1, 3), <class 'numpy.ndarray'>
```

In [ ]:

```
idx = 0
image_data = data['data_train'][idx]
image_data = ((image_data*std_image + mean_image) * 255).astype(np.int32)
plt.imshow(image_data)
label = label_names[data['labels_train'][idx]]
print("Label:", label)
```

```
Label: large_omnivores_and_herbivores
```



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 rgb imag

in_channels = input_image.shape[-1] #must agree with the last dimension c
k_size = 4
n_filt = 16

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

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

```
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]) #
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, padding=

w = np.linspace(-0.2, 0.2, num=weight_size).reshape(k_size, k_size, in_ch
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)".fo

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-7
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
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.forward(im
dw_num = eval_numerical_gradient_array(lambda w: single_conv.forward(img)
db_num = eval_numerical_gradient_array(lambda b: single_conv.forward(img)

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-6
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:  1.3833634232292728e-08
dw Error:  1.4630336847153207e-07
db Error:  1.8189793665778192e-10
dimg Shape:  (15, 8, 8, 3) (15, 8, 8, 3)
```

In [ ]:
```
dimg_num.shape, dimg.shape
```

Out[ ]:
```
((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 si

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

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

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-7
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_test")

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

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:  3.2778449667326122e-12
dimg Shape:  (15, 8, 8, 3) (15, 8, 8, 3)
```

Test a Small Convolutional Neural 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 size
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("fc1_w").std() - std)
b2 = model.net.get_params("fc1_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("fc1_w", w2)
model.net.assign("fc1_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,  -9.20584342,
 [-11.44514171, -10.21200524 , -8.97886878 , -7.74573231 , -6.51259584],
 [ -9.03921048,  -8.89555231 , -8.75189413 , -8.60823596,  -8.46457778],
 [ -6.63327925 , -7.57909937 , -8.52491949 , -9.4707396 , -10.41655972]])
scores_diff = np.sum(np.abs(scores - correct_scores))
print(scores_diff)
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 might be wr
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
        print ('%s relative error: %.2e' % (name, rel_error(grad_num, lay
```

```
Testing initialization ...
Passed!
Testing test-time forward pass ...
5.3041079084437115e-08
Passed!
Testing the loss ...
Passed!
Testing the gradients (error should be no larger than 1e-6) ...
conv_b relative error: 3.90e-09
conv_w relative error: 8.59e-10
fc1_b relative error: 8.77e-11
fc1_w relative error: 3.83e-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 (such as fully-connected layers and non-linearities). You are also free to select any optimizer you have implemented (with any learning rate).

You will train your network on CIFAR-100 20-way superclass classification. Try to find a combination that is able to achieve 40% validation accuracy.

Since the CNN takes significantly longer to train than the fully connected network, it is suggested to start off with fewer filters in your Conv layers and fewer intermediate fully-connected layers so as to get faster initial results.

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

```
Data shape: (40000, 32, 32, 3)
Flattened data input size: 3072
Number of data classes: 20
```

In [ ]:
```python
%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 network
# You may only adjust the hyperparameters within this block
################################################################################
optimizer = Adam(model.net, 1e-3)

batch_size = 16
epochs = 5
lr_decay = 0.9
lr_decay_every = 1000
regularization = "l2"
reg_lambda = 0.01
################################################################################
#                               END OF YOUR CODE
################################################################################
results = train_net(data_dict, model, loss_f, optimizer, batch_size, epoc
                    lr_decay, lr_decay_every, show_every=4000, verbose=Tr
opt_params, loss_hist, train_acc_hist, val_acc_hist = results
```

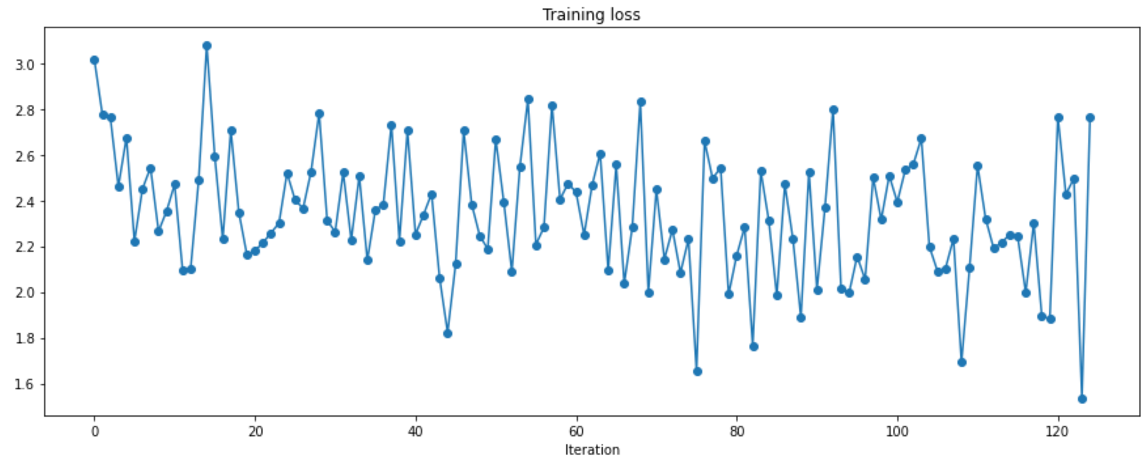
```
  0%|          | 1/2500 [00:00<13:58,  2.98it/s]
(Iteration 1 / 12500) Average loss: 2.9827090569067747
100%|██████████| 2500/2500 [14:35<00:00,  2.85it/s]
(Epoch 1 / 5) Training Accuracy: 0.316325, Validation Accuracy: 0.3063
 60%|██████    | 1501/2500 [13:38<04:40,  3.56it/s]
(Iteration 4001 / 12500) Average loss: 2.449195675770429
100%|██████████| 2500/2500 [18:23<00:00,  2.27it/s]
(Epoch 2 / 5) Training Accuracy: 0.332125, Validation Accuracy: 0.3253
100%|██████████| 2500/2500 [22:09<00:00,  1.88it/s]
(Epoch 3 / 5) Training Accuracy: 0.34215, Validation Accuracy: 0.3247
 20%|██        | 501/2500 [02:46<08:57,  3.72it/s]
(Iteration 8001 / 12500) Average loss: 2.3310430103591786
100%|██████████| 2500/2500 [12:33<00:00,  3.32it/s]
(Epoch 4 / 5) Training Accuracy: 0.342275, Validation Accuracy: 0.3279
 80%|████████  | 2001/2500 [10:59<02:41,  3.09it/s]
(Iteration 12001 / 12500) Average loss: 2.301840297749229
100%|██████████| 2500/2500 [13:42<00:00,  3.04it/s]
(Epoch 5 / 5) Training Accuracy: 0.359225, Validation Accuracy: 0.3435
```

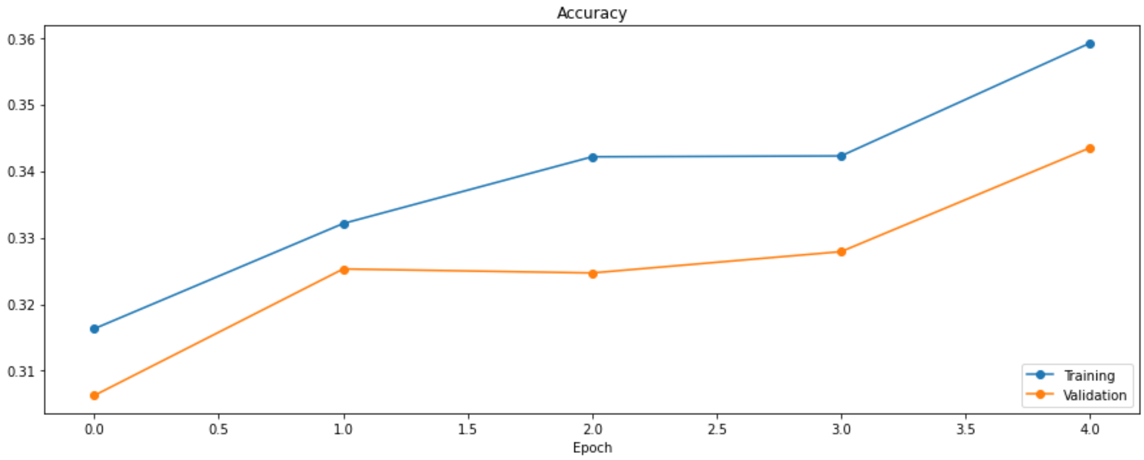
Run the code below to generate the training plots.

In [ ]:

```
%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()
```



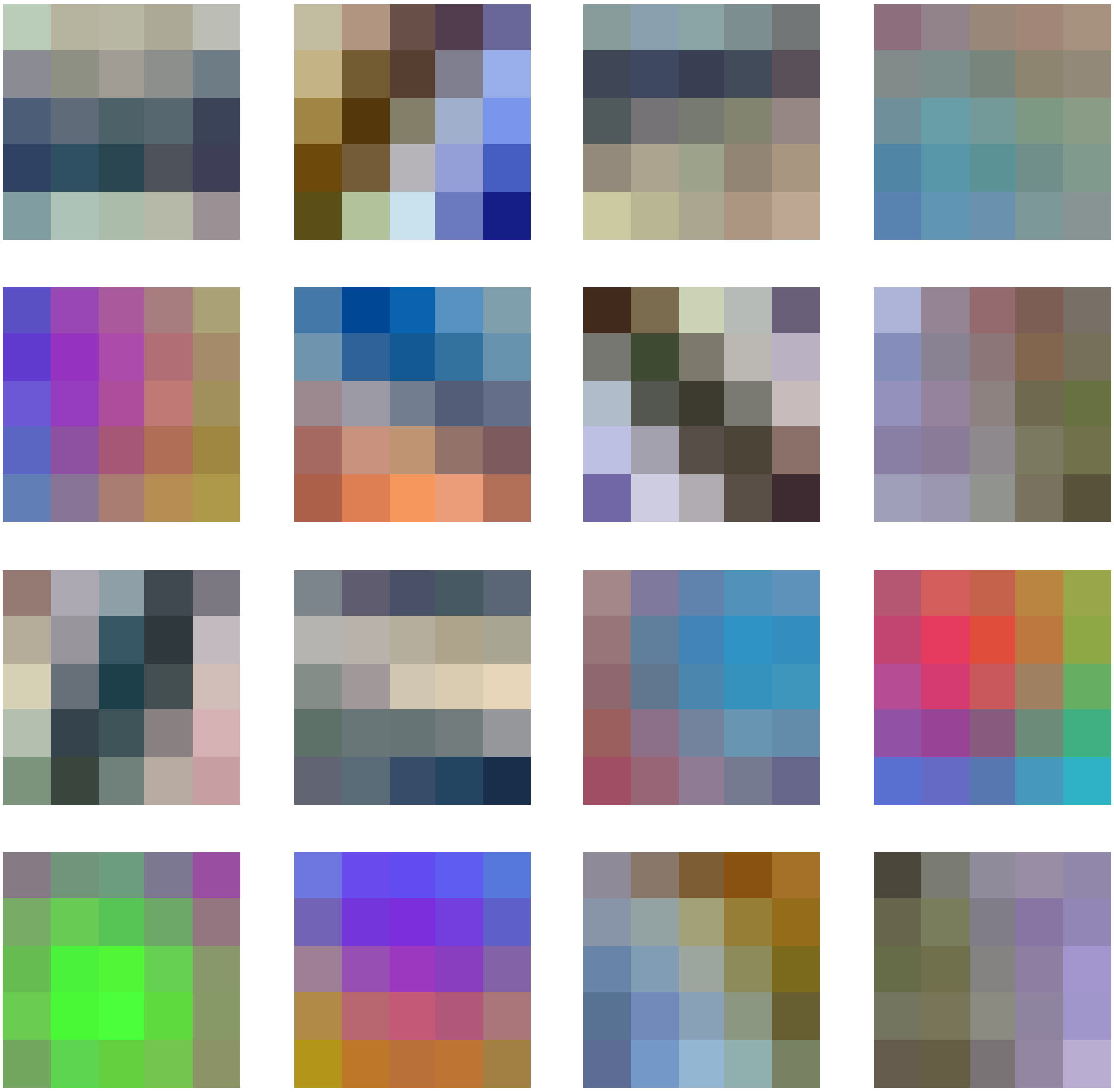


Visualizing Layers [5pts]

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.                            #
####################################################

# get the filters from the first convolutional layer
filters = model.net.get_params("conv_w")

# normalize the filters to be between 0 and 1
filters -= filters.min()
filters /= filters.max()
filter_transposed = filters.transpose(3, 0, 1, 2)

num_rows, num_cols = 4, 4
```

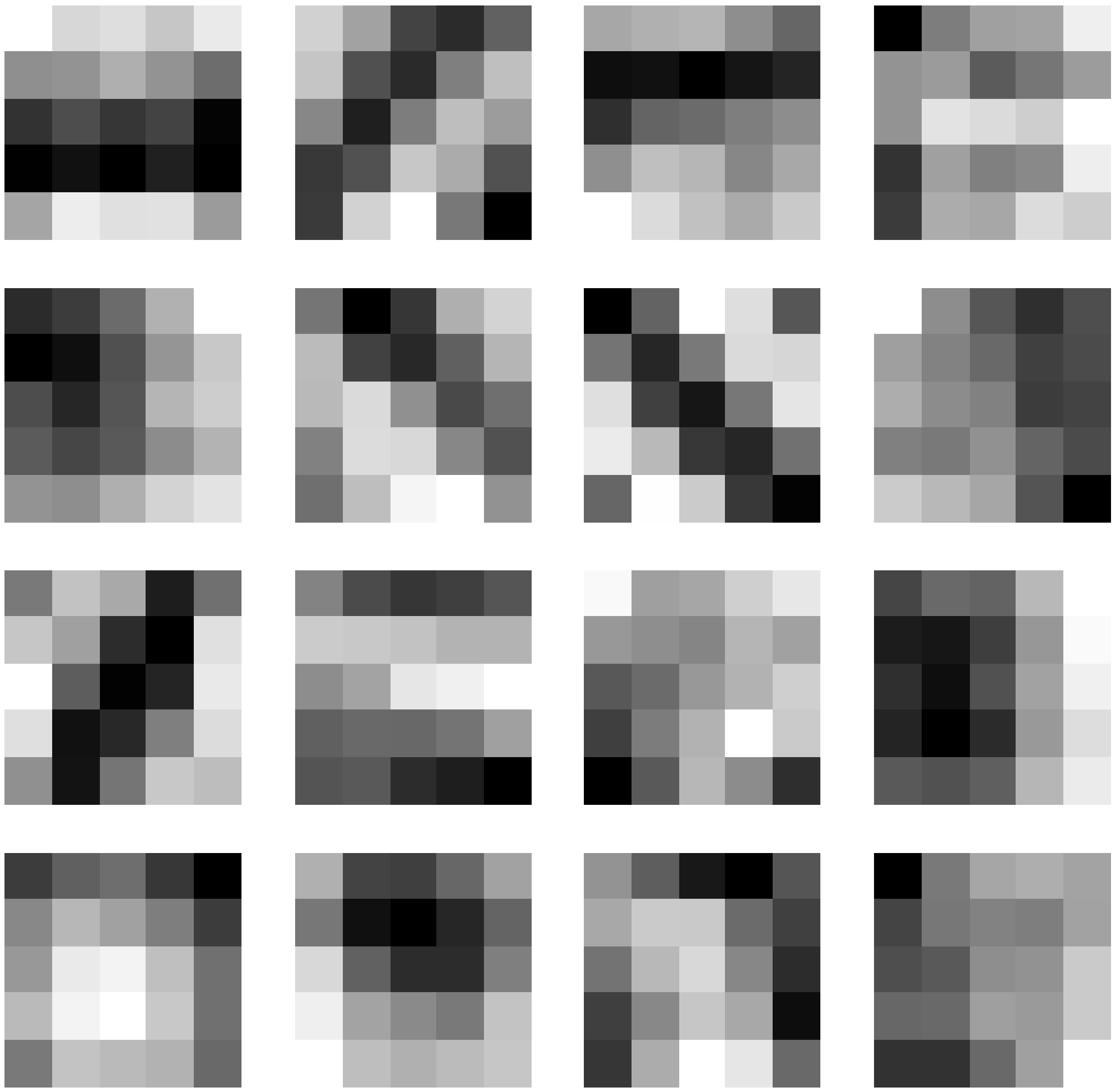
```
fig, axes = plt.subplots(num_rows, num_cols, figsize=(10, 10))
for i in range(num_rows):
    for j in range(num_cols):
        img_idx = i * num_cols + j
        if img_idx < filter_transposed.shape[0]:
            img = filter_transposed[img_idx]
            axes[i][j].imshow(img)
        axes[i][j].axis('off')

###################################################
#                 END OF YOUR CODE                #
###################################################


plt.show()
```



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.                           #
###################################################

# get the filters from the first convolutional layer
```

```
filters = model.net.get_params("conv_w")

# normalize the filters to be between 0 and 1
filters -= filters.min()
filters /= filters.max()
filter_transposed = filters.transpose(3, 0, 1, 2)

num_rows, num_cols = 4, 4
fig, axes = plt.subplots(num_rows, num_cols, figsize=(10, 10))
for i in range(num_rows):
    for j in range(num_cols):
        img_idx = i * num_cols + j
        if img_idx < filter_transposed.shape[0]:
            img = filter_transposed[img_idx]
            img = 0.2126 * img[:,:,0] + 0.7152 * img[:,:,1] + 0.0722 * im
            axes[i][j].imshow(img)
        axes[i][j].axis('off')

###################################################
#                 END OF YOUR CODE                #
###################################################


plt.show()
```



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

Answer. I have plot the gray scale version of the feature maps for better visulaization. The dark squares signify weights that are small or inhibitory, while the light squares denote weights that are large or excitatory.

(Row 1, Col 1) and (Row 1, Col 3) : Represents a horizontal lines feature detector.
(Row 1, Col 2) and (Row 2, Col 2) and (Row 2, Col 3) and (Row 3, Col 1) : Represents an inclined lines feature detector.
(Row 2, Col 1) and (Row 2, Col 4) and (Row 3, Col 4) and (Row 4, Col 4) : Captures the gradient.
(Row 4, Col 1) : Captures the edges.

Extra-Credit: Analysis on Trained Model [5pts]

For extra credit, you can perform some additional analysis of your trained model. Some suggested analyses are:

1. Plot the confusion matrix of your model's predictions on the test set. Look for trends to see which classes are frequently misclassified as other classes (e.g. are the two vehicle superclasses frequently confused with each other?).
2. Implement BatchNorm and analyze how the models train with and without BatchNorm.
3. Introduce some small noise in the labels, and investigate how that affects training and validation accuracy.

You are free to choose any analysis question of interest to you. We will not be providing any starter code for the extra credit. Include your extra-credit analysis as the final section of your report pdf, titled "Extra Credit".

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.

In [ ]: