CV2_HW3_CS4090

March 28, 2022

1 COMS 4732: Advanced Computer Vision Homework 3

1.1 Overview

In this assignment, we'll build a simple convolutional neural network in PyTorch and Train it to recognize natural objects using the CIFAR-10 dataset. Training a classifier on the CIFAR-10 dataset can be regarded as the hello world of image recognition. The structure of this assignment is:

```
1. Introduction
2. Setting up the Environment
3. Preparing the Data
4. Building the Network
5. Training the Model and Evaluating the Performance
6. Understand the Deep Neural Networks
```

(The full content can be better viewed in jupyter notebook).

1.1.1 Submission Instructions

- Please submit the notebook (ipynb and pdf) including the output of all cells. Please note that you should export your completed notebook as a PDF (CV2_HW3_UNI.pdf) and upload it to GradeScope.
- Then, please submit the executable completed notebook (CV2_HW3_UNI.ipynb) to Cousework.
- For both files, 1) remember to replace UNI with your own uni; 2) the completed notebook should show your code, as well as the log and image.

1.2 Introduction

Object Recognition typically aims to train a model to recognize, identify, and locate the objects in the images with a given degree of confidence. Object recognition is also called image recognition in many other kinds of literature, which is strictly related to computer vision, which we define as the art and science of making computers understand images.

Deep learning has contributed quite a lot to the development of object recognition algorithms and has achieved many surprising results on many benchmark datasets, such as ImageNet, MS COCO, LVIS. Also, different architectures are proposed to improve the accuracy and efficiency of the learned models. As a brief introduction, we first go through some basic concepts in object recognition and give the following tasks:

1. Classification.
2. Detection.
3. Segmentation.

In this assignment, we only dive into the image classification task and we start with a small model on a toy dataset. We do encourage you to explore the other tasks. Here are a few interesting references: 1. https://towardsdatascience.com/creating-your-own-object-detector-ad69dda69c85 2. https://medium.com/analytics-vidhya/iou-intersection-over-union-705a39e7acef 3. https://www.analyticsvidhya.com/blog/2018/11/implementation-faster-r-cnn-python-object-detection/ 4. https://blog.paperspace.com/mask-r-cnn-in-tensorflow-2-0/ 5. https://www.analyticsvidhya.com/blog/2021/05/an-introduction-to-few-shot-learning/

1.3 Name: Chandan Suri

1.4 UNI: CS4090

1.4.1 In Collaboration with: Gursifath Bhasin, GB2760

1.4.2 Classification

Classification is an important task in Object recognition and aims to identify what is in the image and with what level of confidence. The general pipeline of this task is straightforward. It starts with the definition of the ontology, i.e. the class of objects to detect. Then, the classification task identifies what is in the image (associated level of confidence) and picks up the class with the highest confidence. Usually, each test image used for classification has one dominant object in the image and we directly use the representation of the whole image for classification.

For the example given above, the classification algorithm will only remember that there is a dog, ignoring all other classes. Similar to classification, tagging also predicts confidence level for each class but aims to recognize multiple ones for a given image. For the example given below, it will try to return all the best classes corresponding to the image.

1.4.3 Detection and segmentation

Once identified what is in the image, we want to locate the objects. There are two ways to do so: detection and segmentation.Detection outputs a rectangle, also called bounding box, where the objects are. It is a very robust technology, prone to minor errors and imprecisions.

Segmentation identifies the objects for each pixel in the image, resulting in a very precise map. However, the accuracy of segmentation depends on an extensive and often time-consuming training of the neural network.

Different from Classification, Detection and Segmentation ask the algorithm to return the location of the objects in the image. However, as no prior is given to predict the location, a set of potential locations/regions are enumerated. Then, the algorithm performs classification and localization prediction on each of the enumerated subregions. To note, the enumerated sub-regions differ from each other regarding the center location, aspect ratio, and size, while all of them are pre-defined by humans and all of the enumerated sub-regions can densely cover the full image. The location prediction is in effect predicting the offset w.r.t. each the sub-region.

For object detection/segmentation, the two-stage and the one-stage frameworks have been dominating the related research area. Recently, the end-to-end object detector has been proposed to

perform prediction in a unified manner (i.e., no pre-defined sub-region will be densely enumerated).

1.4.4 Setting up the Environment

To learn a deep neural network, we need to first build proper environment and we use Pytorch. For some basic overview and features offered in Colab notebooks, check out: Overview of Colaboratory Features.

For this homework, you may need you Colab for your experiments. Under your columbia account, you can open the colab and then upload this jupyter notebook. You need to use the Colab GPU for this assignment by selecting:

Runtime → Change runtime type → Hardware Accelerator: GPU

You should be capable to directly run the following cells in most cases. However, in case you need, you can use the following code to install pytorch when you open the CoLab.

```
!pip install torch torchvision
!pip install Pillow==4.0.0
```

[1]:
```
"""
Below are some input statements which basically imports numpy, torch,␣
↪torchvision, matplotlib - This is a known standard and no need to memorise␣
↪this
"""

import cv2
import torch
import torchvision
import torchvision.transforms as transforms
from torchvision.datasets import CIFAR10
from torch.utils.data import DataLoader
import torchvision.models as models
import matplotlib.pyplot as plt
import numpy as np
from copy import deepcopy
from torchvision.utils import make_grid

import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import random_split

import math
import os
import argparse
```

1.4.5 Warm-Up for Pytorch

[2]:
```
#Pytorch is very similar to numpy. Basically, any operation in numpy can be␣
 ↪easily refered for pytorch operations
# The following is a few examples.

# Operations
y = torch.rand(2, 2)
x = torch.rand(2, 2)

# multiplication
z = x * y
z = torch.mul(x,y) #elementwise #must be broadcastable

#matrix multiplication
tensor1 = torch.randn(3, 4)
tensor2 = torch.randn(4, 3)
torch.matmul(tensor1, tensor2)

# NumPy conversion
x = torch.rand(2,2)
y = x.numpy()
print(type(y))

z1 = torch.from_numpy(y) #sharing the memory space with the numpy ndarray
z2 = torch.tensor(y) #a copy
print(type(z1))
print(type(z2))

# Pytorch attributes and functions for tensors
print(x.shape)
print(x.device)

# autograd

# requires grad equals true lets us compute gradients on the tenor
x = torch.tensor([2,3,5], dtype=float, requires_grad=True)
y =(5 * x**2).sum()

# When we finish our computation we can call .backward() and have all the␣
 ↪gradients computed automatically
# The gradient for this tensor will be accumulated into .grad attribute
y.backward()
#print(z.grad) # dz/dz
print(x.grad) # dz/dx

# autograd requires computational resources and can take time.
```

```
# disable autograd for model eval by writing your evaluation code in
# As such, with torch.no_grad() is usually used in evaluation part
```

```
<class 'numpy.ndarray'>
<class 'torch.Tensor'>
<class 'torch.Tensor'>
torch.Size([2, 2])
cpu
tensor([20., 30., 50.], dtype=torch.float64)
```

For repeatable experiments, we are recommended to set random seeds for anything using random number generation - this means numpy and random as well! It's also worth mentioning that cuDNN uses nondeterministic algorithms which can be disabled by setting torch.backends.cudnn.enabled = False.

[3]:
```
experiment_name = 'debug'  #Provide name to model experiment
model_name = 'basic' # Choose between [basic, alexnet]
batch_size = 5  #You may not need to change this but incase you do

torch.manual_seed(42)
```

[3]:
```
<torch._C.Generator at 0x7fbef5f9ec30>
```

1.5 Preparing the Data

We provide the data-loading functions and a few helper functions below.

To learn a deep learning algorithm, we need to prepare the data and this is where TorchVision comes into play. Usually, we have a training dataset and a test dataset. Each image in the training dataset is paired with a class label, serving as groundtruth to guide the updating of the deep neural network. In real-world test scenario, no label will be provided. However, we still have the labels in the test set to calculate the accuracy. Then, the accuracy on test data is used to quantitatively evaluate the performance, i.e., the generalization of the learned deep algorithm. After all, it will be unexpected if the model works perfectly on the training data but fails on the testing data.

During network training, we usually perform data augmentation on the training data by manipulating the value of pixels. To determine how we adjust the pixel values, we first determine the effects to be applied to the images. TorchVision offers a lot of handy transformations, such as cropping or normalization. For example, shifting the image, adjusting the light and contrast of an image, translating an RGB image into a grayscale image, image rotation, and so on. Also, torchvision can apply random sections among the effects.

To note, the effects to be added on the training data should also seriously consider the property of the dataset. For example, for the digit recognition in MNIST, can we perform 180-degree rotation on the image of digit 6 without altering the image label during network training?

[4]:
```
def get_transform(model_name):

    if model_name == 'alexnet':
        transform = transforms.Compose([
```

```
            transforms.Resize((227, 227)),
            transforms.ToTensor(),
            transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
        ])

    else:

        transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
        ])

    return transform


def get_dataset(model_name, train_percent=0.9):
    '''
    Returns the train, val and test torch.Datasetin addition to a list of␣
↪classes, where the idx of class name corresponds
    to the label used for it in the data


    Reference for transforms in torchvision: https://pytorch.org/vision/stable/
↪transforms.html
    @model_name: either 'basic' or 'alexnet'
    @train_percent: percent of training data to keep for training. Rest will be␣
↪validation.
    '''

    transform  = get_transform(model_name)

    train_data = CIFAR10(root='./data', train=True, download=first_run,␣
↪transform=transform)
    test_data  = CIFAR10(root='./data', train=False, download=first_run,␣
↪transform=transform)

    train_size = int(train_percent * len(train_data))
    val_size = len(train_data) - train_size


    train_data, val_data = random_split(train_data, [train_size, val_size])
    class_names = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog',␣
↪'horse', 'ship', 'truck')

    return train_data, val_data, test_data, class_names
```

```
def get_dataloader(batch_size, num_workers=1, model_name='basic'):
    '''
    Returns the train, val and test dataloaders in addition to a list of␣
 ↪classes, where the idx of class name corresponds
    to the label used for it in the data

    Reference for dataloader class: https://pytorch.org/docs/stable/data.html
    @batch_size: batch to be used by dataloader
    @num_workers: number of dataloader workers/instances used
    @model_name: either 'basic' or 'alexnet'
    '''

    train_set, val_set, test_set, class_names = get_dataset(model_name)
    trainloader = DataLoader(train_set, batch_size=batch_size, shuffle=True,␣
 ↪num_workers=num_workers, pin_memory=True)
    valloader = DataLoader(val_set, batch_size=batch_size, shuffle=False,␣
 ↪num_workers=num_workers, pin_memory=True)
    testloader = DataLoader(test_set, batch_size=batch_size, shuffle=False,␣
 ↪num_workers=num_workers, pin_memory=True)


    return trainloader, valloader, testloader, class_names

def makegrid_images(model_name='basic'):
    '''
    For visualization purposes

    @model_name: either 'basic' or 'alexnet'
    '''



    _, trial_loader, _, _ = get_dataloader(32, model_name=model_name)
    images, labels = iter(trial_loader).next()

    grid = make_grid(images)

    return grid

def show_img(img, mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5), viz=True,␣
 ↪norm=True):
    '''
    For visualization purposes

    B: batch size
    C: channels
    H: height
```

```
    W: width

    @img: torch.Tensor for the image of type (B, C, H, W)
    @mean: mean used for normalizing along 3 dimensions C, H, W in get_transform
    @std:  std. deviation used for normalizing along 3 dimensions C, H, WW in␣
↪get_transform
    @viz: whether or not to plt.plot or just return the unnormalized image
    @norm: whether or not unnormalize. Unnormalizes if true.

    Returns:
    Viewable image in (H, W, C) as a numpy array
    '''


    if norm:
        for idx in range(img.shape[0]):

            img[idx] = img[idx] * std[idx] + mean[idx]

    image = np.asarray(img)

    if viz:
        if len(image.shape) == 4:
            image = image.squeeze()

        plt.imshow(np.transpose(image, (1, 2, 0)))
        plt.show()

    return np.transpose(image.squeeze(), (1, 2, 0))
```

1.6 Building the Network

The network is essentially the key component of the deep learning design. For image classification, the network is typically a convolution neural network (i.e., CNN, ConvNet). CNN will take the full image as input and then predict the confidence level for each class, which is further used for classification. To learn a deep neural network, there are roughly two different ways, 1) training from scratch and 2) finetuning from a pre-trained model.

1.6.1 Training from scratch

First, we will train a shallow convolutional neural network from scratch for practice. We first define the neural network in **GradBasicNet**, randomly initialize the value of the parameters in the network (the reason why it is called ''from scratch"), and train it on the training dataset:

1. Fill out **get_conv_layers()** with a network of **2 conv layers** each with kernel size of 5. After the first convolutional layers, add a **max pooling layer** with kernel size of 2 and stride of 2. Remember to add the non linearities immediately after the conv layers. You must choose the in-channels and out-channels for both the layers yourself: remember that the image to

be input will be RGB so there is only one number that can be used for the in-channels of the first conv layer. Use the nn.Sequential API to combine all this into one layer, and return from **get_conv_layers()** method

2. Fill out **final_pool_layer()** with a max pooling layer. Typically after all the convolutional layers there is another max pooling layer. Use the same kernel size and stride as before and this time directly return the **nn.MaxPool2d**.
3. Fill out the **get_fc_layers()** method with a classifier containing 3 linear layers. Once again you are free to choose the in_channels and out_channels for these yourself. Once again use the **nn.Sequential** API and return the object from the method. Inside, alternate the Linear layers with ReLU activations. You do not need a ReLU after the final layer. Remember that the first Linear layer must take in the output of the final convolution layer so depending on your choice in (1.) there is only 1 value you can have for the in_channels of the first Linear layer. Also remember that the final Linear layer must have out_channels=10 as we are performing 10-way classification. Lastly, **remember to leave comments** on why you need to keep the relu of the first two layers while remove the relu of the last layer.
4. Finally, you should fill out the forward pass using these layers. Remember to use **self.conv_model**, **self.final_max_pool** and **self.fc_model** one after the other.

Here are a few useful reference for your implementation, which are useful for your implementation: 1. convolution layers: https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html 2. mas pooling layers: https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html 3. Sequential API: https://pytorch.org/docs/stable/generated/torch.nn.Sequential.html

[5]:
```
class GradBasicNet(nn.Module):

    def __init__(self):
        super().__init__()

        self.conv_model = self.get_conv_layers()
        self.final_max_pool = self.final_pool_layer()
        self.fc_model = self.get_fc_layers()

    def get_conv_layers(self):

        layers = nn.Sequential(
            nn.Conv2d(3, 32, 5),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(32, 64, 5),
            nn.ReLU()
        )

        return layers

    def final_pool_layer(self):

        layer = nn.MaxPool2d(2, 2)
```

```
        return layer

    def get_fc_layers(self):

        layers = nn.Sequential(
            nn.Linear(1600, 1028),
            nn.ReLU(),
            nn.Linear(1028, 512),
            nn.ReLU(),
            nn.Linear(512, 10)
        )

        # ================================================
        # ReLU activation is not added after the last fully connected layer␣
↪here because
        # it's not needed. As while training, we need to calculate loss through␣
↪Cross Entropy loss
        # which basically adds a sigmoid activation at the end to help classify␣
↪the image and find the
        # label for the image, thus, the ReLU activation is not required after␣
↪the final fully connected layer
        # due to the presence of the sigmoid activation function at the end of␣
↪the model layers.
        # ================================================
        return layers

    def register_grad_hook(self, grad):
        self.grad = grad


    def forward(self, x):

        x = self.conv_model(x)
        #ignore this: relevance for gradcam section
        h = x.register_hook(self.register_grad_hook)

        x = self.final_max_pool(x)
        x = torch.flatten(x, 1)
        x = self.fc_model(x)

        return x

    def get_gradient_activations(self):
        return self.grad

    def get_final_conv_layer(self, x):
        return self.conv_model(x)
```

1.6.2 Finetuning on a pre-trained model

Secondly, we build a stronger convolutional neural network by starting from a pre-trained model **AlexNet**:

In this subsection, we might have to take advantage of a pretrained network in a process called transfer learning, where we only train a few final layers of a neural network. Here we will use the AlexNet architecture (Krizhevsky et al.)that revolutionized Deep Learning. The pretrained model (on ImageNet) is available on torchvision library and all we need to is ask PyTorch to allow updates on a few of thee final layers during training. We will put the AlexNet model also into the API we have used for our model allow, except that we will need an additionl **transition layer** function for the added **AvgPool** layer. Visualize the model by running the code cell. You will notice: 1. (features) contains most of the conv layers. We need upto (11) to include every conv layer 2. (features) (12) is the final max pooling layer 3. (avgpool) is the transition average pooling layer 4. (classifier) is the collection of linear layers

Question After you finish the implementation and run the experiments, pls go back and answer the questions below: 1. what is the main difference between finetuning and training from scratch.

2. Compare the performance between finetuning from a pre-trained model and training from scratch. Explain why the better one outperforms the other?

Answer: 1. The main difference between finetuning and training from scratch is the training weights in the model. For the finetuning case, the layers in the pre-trained model have been trained to capture the coarse details in the image and also some fine details. Precisely, we just tune the weights of the last few layers for our use case and keep the weights to detect the coarse details in the image intact (no re-training). On the other hand, when we train from scratch, we basically start from randomized or zeroed weights which don't give us much information at first but after weights are learnt for all the layers in the model, the model starts capturing all kinds of details. For the same reason, sometimes a larger dataset is required to train a model from scratch in comparison to fine tuning a pre-trained model. 2. Performance of the pre-trained model: Testing Accuracy: 87.87%. Performance of the model trained from scratch: Testing Accuracy: 71.37%. By far, the performance of the fine-tuned model is better in comparison to the model trained from scratch. There are multiple reasons here that the fine tuned model performs better here: a. As the fine-tuned model already has learnt to capture the coarse details like edges and shapes in the image, fine tuning the later layers that capture finer details is easier and helps learn the model much faster. b. As we don't have a pretty large dataset here, fine tuning model because of it's pre-trained weights require much less to reach better performance in comparison to the model being trained from scratch which knows nothing about capturing any kinds of details in the image initially. This means that the model being trained from scratch would converge to an optimal solution at a later stage w.r.t the fine tuned model. c. Also, for our case here, the model being trained from scratch is not as deep and complex as the AlexNet which is quite a large model with more number of layers and nodes in the hidden layers. Thus, AlexNet is capable of capturing much more details in comparison to our basic model being trained from scratch. This also contributes to the fine tuned model performing better.

[6]:
```
example_model = models.alexnet(pretrained=True)
print(example_model)
```

```
Downloading: "https://download.pytorch.org/models/alexnet-owt-7be5be79.pth" to
```

```
/root/.cache/torch/hub/checkpoints/alexnet-owt-7be5be79.pth

  0%|          | 0.00/233M [00:00<?, ?B/s]

AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

Your task is two fold:

1. Implement the method **activate training layers** which sets the requires_grad of relevant parameters to True, so that training can occur. You can iterate over training parameters with

```
for name, param in self.conv_model.named_parameters():
```

and

```
for name, param in self.fc_model.named_parameters():
```

For the conv layers, every param should have requires_grad set to false except for the last layer (10). For the linear layers, all layers must be trainable aka requires_grad must be set to True.

2. Implement the forward pass in the following order: self.conv_model, self.final_max_pool, self.avg_pool, self.fc_model

3. Remember to fill in your comments under the question mentioned above.

[7]:
```
class GradAlexNet(nn.Module):

    def __init__(self):
        super().__init__()

        self.base_alex_net = models.alexnet(pretrained=True)


        self.conv_model = self.get_conv_layers()
        self.final_max_pool = self.final_pool_layer()
        self.avg_pool = self.transition_layer()
        self.fc_model = self.get_fc_layers()

        self.activate_training_layers()

    def activate_training_layers(self):

        for name, param in self.conv_model.named_parameters():
            print(name)
            # this is the number of every convolutional layer. From what model␣
↪printed above, what is
            # the last convolutional layer?
            # The last convolutional layer is the 10th one (index of 10)
            # with both in channels and out channels as 256.
            # (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),␣
↪padding=(1, 1))
            number = int(name.split('.')[0])

            if number < 10:
                param.requires_grad = False
            else:
                param.requires_grad = True

        for name, param in self.fc_model.named_parameters():
            param.requires_grad = True


    def get_conv_layers(self):

        return self.base_alex_net.features[:12]

    def final_pool_layer(self):
        return nn.MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,␣
↪ceil_mode=False)

    def transition_layer(self):
        return nn.AdaptiveAvgPool2d(output_size=(6, 6))
```

```
    def get_fc_layers(self):
        return nn.Sequential(
            nn.Dropout(p=0.5, inplace=False),
            nn.Linear(in_features=9216, out_features=4096, bias=True),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.5, inplace=False),
            nn.Linear(in_features=4096, out_features=4096, bias=True),
            nn.ReLU(inplace=True),
            nn.Linear(in_features=4096, out_features=1000, bias=True),
            nn.ReLU(inplace=True),
            nn.Linear(in_features=1000, out_features=10, bias=True),
        )

    def register_grad_hook(self, grad):
        self.grad = grad

    def forward(self, x):

        x = self.conv_model(x)
        h = x.register_hook(self.register_grad_hook)

        x = self.final_max_pool(x)
        x = self.avg_pool(x)
        x = torch.flatten(x, 1)
        x = self.fc_model(x)

        return x

    def get_gradient_activations(self):
        return self.grad

    def get_final_conv_layer(self, x):
        return self.conv_model(x)
```

1.7 Training the Model and Evaluating the Model's Performance

Performing network training is intuitive, i.e., feed the training data into the network and then use the groundtruth label to guide the training of the network parameters. In practice, we need to measure the inconsistency between the network output and groundtruth label and then update the network through gradient back propagation. As such, during a network training, a few components are necessary for the implementataion.

1. criterion (loss function): measure the inconsistency between the network output and label
2. optimizer: calculates gradients and then update the network parameters.

Similar to the human vision system which may need to obtain new knowledge by repeating the process of learning and practicing, purely feeding the training data into the model once is far from enough. Meanwhile, we also need to check whether the model overfits to the training data. As

such, we monitor the status of the learned network by checking its performance on the test dataset at each training loop.

To note, one training loop is denoted as one epoch and means all of the training data has been used to train the deep neural network once. Now we will fill out the classifier we use to train these networks. Study the general framework for the code below well. 1. In the **init** method fill out the **self.criterion** and **self.optimizer**. Remember this is a classification problem so we will need a cross entropy loss. For the optimizer, I would recommend stochastic gradient descent with a learning rate of 0.001 and momentum of 0.9. The rest of **init** has been filled out for you. 2. Next, fill out the **training loop**. Here you are expected to iterate over **self.dataloaders['train']** and optimize on the loss with the groundtruth labels. The **validation** aspect of the training loop has been provided for you and so has the evaluate method. In the training loop, print the average loss every 1000 images you process. Remeber to zero out gradients on the model before you do loss.backward(), and then only after this backward step, make a step in the right direction using the optimizer.

At this stage ignore all methods, after evaluate. They will be relevant to later sections and you will have to return to them when you have more instructions.

[8]:
```
class Classifier():

    def __init__(self, name, model, dataloaders, class_names, use_cuda=False):

        '''
        @name: Experiment name. Will define stored results etc.
        @model: Either a GradBasicNet() or a GradAlexNet()
        @dataloaders: Dictionary with keys train, val and test and␣
↪corresponding dataloaders
        @class_names: list of classes, where the idx of class name corresponds␣
↪to the label used for it in the data
        @use_cuda: whether or not to use cuda
        '''

        self.name = name
        if use_cuda and not torch.cuda.is_available():
            raise Exception("Asked for CUDA but GPU not found")

        self.use_cuda = use_cuda

        self.model = model.to('cuda' if use_cuda else 'cpu')

        self.criterion = nn.CrossEntropyLoss()
        self.optim = optim.SGD(self.model.parameters(), lr=0.001, momentum=0.9)

        self.dataloaders = dataloaders
        self.class_names = class_names
        self.activations_path = os.path.join('activations', self.name)
        self.kernel_path = os.path.join('kernel_viz', self.name)
```

```
        save_path = os.path.join(os.getcwd(), 'models', self.name)
        if not os.path.exists(save_path):
            os.makedirs(save_path)

        if not os.path.exists(self.activations_path):
            os.makedirs(self.activations_path)

        if not os.path.exists(self.kernel_path):
            os.makedirs(self.kernel_path)

        self.save_path = save_path

    def train(self, epochs, save=True):
        '''
        @epochs: number of epochs to train
        @save: whether or not to save the checkpoints
        '''

        best_val_accuracy = - math.inf

        for epoch in range(epochs):

            self.model.train()

            batches_in_pass = len(self.dataloaders['train'])

            #You may comment these two lines if you do not wish to use them
            loss_total = 0.0 # Record the total loss within a few steps
            epoch_loss = 0.0 # Record the total loss for each epoch

            for idx, data in enumerate(self.dataloaders['train']):

                inputs, labels = data
                inputs = inputs.to('cuda' if self.use_cuda else 'cpu')
                labels = labels.to('cuda' if self.use_cuda else 'cpu')

                self.optim.zero_grad()

                outputs = self.model(inputs)
                loss = self.criterion(outputs, labels)
                loss.backward()
                self.optim.step()

                loss_total = loss.item()
                epoch_loss += loss_total
                if idx % 1000 == 999:
```

```
                print(f"[Epoch {epoch + 1}: {idx + 1}] Loss: {loss_total}")

        '''Give validation'''
        epoch_loss /= batches_in_pass

        self.model.eval()

        #DO NOT modify this part
        correct = 0.0
        total = 0.0
        for idx, data in enumerate(self.dataloaders['val']):

            inputs, labels = data
            inputs = inputs.to('cuda' if self.use_cuda else 'cpu')
            labels = labels.to('cuda' if self.use_cuda else 'cpu')

            outputs = self.model(inputs)
            _, predicted = torch.max(outputs, 1)

            total += labels.shape[0]
            correct += (predicted == labels).sum().item()

        epoch_accuracy = 100 * correct / total

        print(f'Train Epoch Loss (Avg): {epoch_loss}')
        print(f'Validation Epoch Accuracy:{epoch_accuracy}')

        if save:
            #  Make sure that your saving pipeline is working well.
            # Is os library working on your file system?
            # Is your model being saved and reloaded fine?
            # When you do the kernel viz, activation maps,
            # and GradCAM you must be using the model you have saved before.

            torch.save(self.model.state_dict(), os.path.join(self.
↪save_path, f'epoch_{epoch}.pt'))

            if epoch_accuracy > best_val_accuracy:

                torch.save(self.model.state_dict(), os.path.join(self.
↪save_path, 'best.pt'))
                best_val_accuracy = epoch_accuracy

    print('Done training!')


def evaluate(self):
```

```
        try:
            assert os.path.exists(os.path.join(self.save_path, 'best.pt'))

        except:
            print('It appears you are testing the model without training.␣
↪Please train first')
            return

        self.model.load_state_dict(torch.load(os.path.join(self.save_path,␣
↪'best.pt')))
        self.model.eval()

        #total = len(self.dataloaders['test'])

        # Addendum Code for analysis:
        labels_max_count_map = {idx: 0 for idx in range(10)}
        correct_preds_labels_count_map = {idx: 0 for idx in range(10)}

        correct = 0.0
        total = 0.0
        for idx, data in enumerate(self.dataloaders['test']):
            inputs, labels = data
            inputs = inputs.to('cuda' if self.use_cuda else 'cpu')
            labels = labels.to('cuda' if self.use_cuda else 'cpu')

            outputs = self.model(inputs)
            _, predicted = torch.max(outputs, 1)

            total += labels.shape[0]
            correct += (predicted == labels).sum().item()

            # Addendum Code for analysis:
            # Get total counts of all the labels and
            # update the correct predictions made for each of the labels.
            for prediction, label in zip(predicted.tolist(), labels.tolist()):
                labels_max_count_map[int(label)] += 1
                if int(prediction) == int(label):
                    correct_preds_labels_count_map[int(prediction)] += 1

        print(f'Accuracy: {100 * correct/total}%')

        # Addendum Code for analysis:
        for class_index in range(10):
            correct_pred_count = correct_preds_labels_count_map[class_index]
            max_label_count = labels_max_count_map[class_index]
            accuracy_prop = correct_pred_count/max_label_count
```

```
            print(f'Accuracy for Class Label {class_index}: {100 *␣
↪accuracy_prop}')

    def grad_cam_on_input(self, img):

        try:
            assert os.path.exists(os.path.join(self.save_path, 'best.pt'))

        except:
            print('It appears you are testing the model without training.␣
↪Please train first')
            return

        self.model.load_state_dict(torch.load(os.path.join(self.save_path,␣
↪'best.pt')))


        self.model.eval()
        img = img.to('cuda' if self.use_cuda else 'cpu')


        out = self.model(img)

        _, pred = torch.max(out, 1)

        predicted_class = self.class_names[int(pred)]
        print(f'Predicted class was {predicted_class}')

        out[:, pred].backward()
        gradients = self.model.get_gradient_activations()

        print('Gradients shape: ', f'{gradients.shape}')

        mean_gradients = torch.mean(gradients, [0, 2, 3]).cpu()
        activations = self.model.get_final_conv_layer(img).detach().cpu()

        print('Activations shape: ', f'{activations.shape}')

        for idx in range(activations.shape[1]):
            activations[:, idx, :, :] *= mean_gradients[idx]

        final_heatmap = np.maximum(torch.mean(activations, dim=1).squeeze(), 0)

        final_heatmap /= torch.max(final_heatmap)

        return final_heatmap
```

```
    def trained_kernel_viz(self):

        all_layers = [0, 3]
        all_filters = []
        for layer in all_layers:

            filters = self.model.conv_model[layer].weight
            all_filters.append(filters.detach().cpu().clone()[:8, :8, :, :])

        for filter_idx in range(len(all_filters)):

            filter = all_filters[filter_idx]
            print(filter.shape)
            filter = filter.contiguous().view(-1, 1, filter.shape[2], filter.
↪shape[3])
            image = show_img(make_grid(filter))
            image = 255 * image
            cv2.imwrite(os.path.join(self.kernel_path,␣
↪f'filter_layer{all_layers[filter_idx]}.jpg'), image)


    def activations_on_input(self, img):

        img = img.to('cuda' if self.use_cuda else 'cpu')

        all_layers = [0,3,6,8,10]
        all_viz = []

        for each in all_layers:

            current_model = self.model.conv_model[:each+1]
            current_out = current_model(img)
            all_viz.append(current_out.detach().cpu().clone()[:, :64, :, :])

        for viz_idx in range(len(all_viz)):

            viz = all_viz[viz_idx]
            viz = viz.view(-1, 1, viz.shape[2], viz.shape[3])
            image = show_img(make_grid(viz))
            image = 255 * image
            cv2.imwrite(os.path.join(self.activations_path,␣
↪f'sample_layer{all_layers[viz_idx]}.jpg'), image)
```

Run the classifier for the code using the basic model by running the following snippets. If all goes well, you should have a test accuracy of about ~60-70% at the end of it. It should take <10 mins to run on a GPU.

[10]:
```
experiment_name = 'basic_debug'  #Provide name to model experiment
model_name = 'basic' #Choose between [basic, alexnet]
batch_size = 5  #You may not need to change this but incase you do
first_run = True #whether or not first time running it

trainloader, valloader, testloader, class_names =␣
↪get_dataloader(batch_size=batch_size, model_name=model_name)
dataloaders = {'train': trainloader, 'val' : valloader, 'test': testloader,␣
↪'mapping': class_names}

if model_name == 'basic':
    model = GradBasicNet()
elif model_name == 'alexnet':
    model = GradAlexNet()
else:
    raise NotImplementedError("This option has not been implemented. Choose␣
↪between 'basic' and 'alexnet' ")

classifier = Classifier(experiment_name, model, dataloaders, class_names,␣
↪use_cuda=True)
```

```
Files already downloaded and verified
Files already downloaded and verified
```

[11]:
```
# When you develop your code, to save your time, you can choose to run the model
# for 5 cpoches. The accuracy after training for 5 epoches has already been high
# and close to the model after 20-epoch traiing.
# (In my case, Validation Epoch Accuracy is above 61 after training 5 epoches)

# To note, for your final submission, you should make sure to train the model
# for 20 epoches and analysis that model in the later sections.

# classifier.train(epochs=5) # For your reference
classifier.train(epochs=20)
classifier.evaluate()
```

```
[Epoch 1: 1000] Loss: 1.839289665222168
[Epoch 1: 2000] Loss: 1.075409173965454
[Epoch 1: 3000] Loss: 1.998486876487732
[Epoch 1: 4000] Loss: 0.8425053358078003
[Epoch 1: 5000] Loss: 1.3410965204238892
[Epoch 1: 6000] Loss: 0.9979670643806458
[Epoch 1: 7000] Loss: 0.8590003252029419
[Epoch 1: 8000] Loss: 1.108230471611023
[Epoch 1: 9000] Loss: 0.6925342082977295
Train Epoch Loss (Avg): 1.6105484359098805
Validation Epoch Accuracy:54.06
[Epoch 2: 1000] Loss: 0.9636129140853882
```

```
[Epoch 2: 2000] Loss: 0.7447077035903931
[Epoch 2: 3000] Loss: 1.316241979598999
[Epoch 2: 4000] Loss: 0.6342312693595886
[Epoch 2: 5000] Loss: 0.8974155187606812
[Epoch 2: 6000] Loss: 0.9856387972831726
[Epoch 2: 7000] Loss: 0.41958656907081604
[Epoch 2: 8000] Loss: 0.7833250761032104
[Epoch 2: 9000] Loss: 1.728666067123413
Train Epoch Loss (Avg): 1.1517426948630147
Validation Epoch Accuracy:61.48
[Epoch 3: 1000] Loss: 0.6462609767913818
[Epoch 3: 2000] Loss: 1.5981109142303467
[Epoch 3: 3000] Loss: 1.1835819482803345
[Epoch 3: 4000] Loss: 0.8605234026908875
[Epoch 3: 5000] Loss: 1.7419397830963135
[Epoch 3: 6000] Loss: 1.241926670074463
[Epoch 3: 7000] Loss: 0.5782861709594727
[Epoch 3: 8000] Loss: 1.476057767868042
[Epoch 3: 9000] Loss: 0.8558582067489624
Train Epoch Loss (Avg): 0.9259636046851261
Validation Epoch Accuracy:66.46
[Epoch 4: 1000] Loss: 0.268186092376709
[Epoch 4: 2000] Loss: 0.6980152130126953
[Epoch 4: 3000] Loss: 0.8150874376296997
[Epoch 4: 4000] Loss: 0.618560254573822
[Epoch 4: 5000] Loss: 0.32110366225242615
[Epoch 4: 6000] Loss: 1.637263298034668
[Epoch 4: 7000] Loss: 0.04520019143819809
[Epoch 4: 8000] Loss: 0.4168470501899719
[Epoch 4: 9000] Loss: 0.3747004270553589
Train Epoch Loss (Avg): 0.7532294510656761
Validation Epoch Accuracy:70.2
[Epoch 5: 1000] Loss: 0.5003529787063599
[Epoch 5: 2000] Loss: 0.9156892895698547
[Epoch 5: 3000] Loss: 0.6951195001602173
[Epoch 5: 4000] Loss: 2.1024765968322754
[Epoch 5: 5000] Loss: 0.7229236364364624
[Epoch 5: 6000] Loss: 1.0482171773910522
[Epoch 5: 7000] Loss: 0.06440386176109314
[Epoch 5: 8000] Loss: 0.1492009311914444
[Epoch 5: 9000] Loss: 0.38160574436187744
Train Epoch Loss (Avg): 0.6043931998691341
Validation Epoch Accuracy:72.32
[Epoch 6: 1000] Loss: 0.0789753869175911
[Epoch 6: 2000] Loss: 0.853803813457489
[Epoch 6: 3000] Loss: 0.8908075094223022
[Epoch 6: 4000] Loss: 0.13001447916030884
[Epoch 6: 5000] Loss: 0.04225284233689308
```

```
[Epoch 6: 6000] Loss: 0.20981648564338684
[Epoch 6: 7000] Loss: 0.9528225660324097
[Epoch 6: 8000] Loss: 0.5052703619003296
[Epoch 6: 9000] Loss: 0.3961432874202728
Train Epoch Loss (Avg): 0.4703508247951007
Validation Epoch Accuracy:71.52
[Epoch 7: 1000] Loss: 0.15290606021881104
[Epoch 7: 2000] Loss: 0.23877671360969543
[Epoch 7: 3000] Loss: 0.051776014268398285
[Epoch 7: 4000] Loss: 0.03228997439146042
[Epoch 7: 5000] Loss: 0.3133305311203003
[Epoch 7: 6000] Loss: 0.33541765809059143
[Epoch 7: 7000] Loss: 0.05639665201306343
[Epoch 7: 8000] Loss: 0.6990998387336731
[Epoch 7: 9000] Loss: 0.5042819380760193
Train Epoch Loss (Avg): 0.3483702787133925
Validation Epoch Accuracy:70.66
[Epoch 8: 1000] Loss: 0.048941470682621
[Epoch 8: 2000] Loss: 0.2583470046520233
[Epoch 8: 3000] Loss: 0.13920670747756958
[Epoch 8: 4000] Loss: 0.16362841427326202
[Epoch 8: 5000] Loss: 0.20375525951385498
[Epoch 8: 6000] Loss: 0.010893427766859531
[Epoch 8: 7000] Loss: 0.32540035247802734
[Epoch 8: 8000] Loss: 0.03432852774858475
[Epoch 8: 9000] Loss: 0.43127885460853577
Train Epoch Loss (Avg): 0.26027248872993025
Validation Epoch Accuracy:71.5
[Epoch 9: 1000] Loss: 0.18862944841384888
[Epoch 9: 2000] Loss: 0.12818178534507751
[Epoch 9: 3000] Loss: 0.01781461201608181
[Epoch 9: 4000] Loss: 0.00018945429474115372
[Epoch 9: 5000] Loss: 0.03158103674650192
[Epoch 9: 6000] Loss: 0.03658009320497513
[Epoch 9: 7000] Loss: 0.6283212900161743
[Epoch 9: 8000] Loss: 0.08914465457201004
[Epoch 9: 9000] Loss: 0.46313410997390747
Train Epoch Loss (Avg): 0.19324884827307118
Validation Epoch Accuracy:70.4
[Epoch 10: 1000] Loss: 0.11728458106517792
[Epoch 10: 2000] Loss: 0.1560174524784088
[Epoch 10: 3000] Loss: 0.1612754762172699
[Epoch 10: 4000] Loss: 0.08368493616580963
[Epoch 10: 5000] Loss: 0.5843814015388489
[Epoch 10: 6000] Loss: 1.4267776012420654
[Epoch 10: 7000] Loss: 0.1318790763616562
[Epoch 10: 8000] Loss: 0.13022395968437195
[Epoch 10: 9000] Loss: 0.1571357697248459
```

```
Train Epoch Loss (Avg): 0.1457612246131796
Validation Epoch Accuracy:70.4
[Epoch 11: 1000] Loss: 0.055141352117061615
[Epoch 11: 2000] Loss: 0.06422601640224457
[Epoch 11: 3000] Loss: 0.002086097840219736
[Epoch 11: 4000] Loss: 0.24652640521526337
[Epoch 11: 5000] Loss: 0.013263210654258728
[Epoch 11: 6000] Loss: 0.0034316019155085087
[Epoch 11: 7000] Loss: 0.18845054507255554
[Epoch 11: 8000] Loss: 0.030243193730711937
[Epoch 11: 9000] Loss: 0.1850178986787796
Train Epoch Loss (Avg): 0.1251217217356284
Validation Epoch Accuracy:71.32
[Epoch 12: 1000] Loss: 0.0026425844989717007
[Epoch 12: 2000] Loss: 0.3827405273914337
[Epoch 12: 3000] Loss: 0.10337887704372406
[Epoch 12: 4000] Loss: 0.3289756178855896
[Epoch 12: 5000] Loss: 0.013572342693805695
[Epoch 12: 6000] Loss: 0.011606165207922459
[Epoch 12: 7000] Loss: 0.004146096762269735
[Epoch 12: 8000] Loss: 0.09607820957899094
[Epoch 12: 9000] Loss: 0.0003378721885383129
Train Epoch Loss (Avg): 0.11245241704359099
Validation Epoch Accuracy:71.58
[Epoch 13: 1000] Loss: 0.010517491027712822
[Epoch 13: 2000] Loss: 5.702177440980449e-05
[Epoch 13: 3000] Loss: 0.006150527857244015
[Epoch 13: 4000] Loss: 0.19340787827968597
[Epoch 13: 5000] Loss: 0.0018818371463567019
[Epoch 13: 6000] Loss: 1.2564317330543417e-05
[Epoch 13: 7000] Loss: 0.003162441309541464
[Epoch 13: 8000] Loss: 0.0003678981738630682
[Epoch 13: 9000] Loss: 0.0976262241601944
Train Epoch Loss (Avg): 0.08888193207077402
Validation Epoch Accuracy:72.54
[Epoch 14: 1000] Loss: 0.0006436306284740567
[Epoch 14: 2000] Loss: 0.04441498965024948
[Epoch 14: 3000] Loss: 0.0004627453163266182
[Epoch 14: 4000] Loss: 0.13770990073680878
[Epoch 14: 5000] Loss: 0.06772267818450928
[Epoch 14: 6000] Loss: 0.00650255149230361
[Epoch 14: 7000] Loss: 0.005267380736768246
[Epoch 14: 8000] Loss: 0.0027501597069203854
[Epoch 14: 9000] Loss: 0.0009340624092146754
Train Epoch Loss (Avg): 0.07629131440054798
Validation Epoch Accuracy:71.7
[Epoch 15: 1000] Loss: 0.00023254481493495405
[Epoch 15: 2000] Loss: 0.015753349289298058
```

```
[Epoch 15: 3000] Loss: 0.01821734569966793
[Epoch 15: 4000] Loss: 0.00037575262831524014
[Epoch 15: 5000] Loss: 0.0010607788572087884
[Epoch 15: 6000] Loss: 0.000147955899592489
[Epoch 15: 7000] Loss: 0.008989282883703709
[Epoch 15: 8000] Loss: 0.3354683220386505
[Epoch 15: 9000] Loss: 0.015010637231171131
Train Epoch Loss (Avg): 0.06319480144511867
Validation Epoch Accuracy:71.1
[Epoch 16: 1000] Loss: 0.006644558161497116
[Epoch 16: 2000] Loss: 0.0002451086766086519
[Epoch 16: 3000] Loss: 0.020070618018507957
[Epoch 16: 4000] Loss: 0.0004414243740029633
[Epoch 16: 5000] Loss: 0.0007731933728791773
[Epoch 16: 6000] Loss: 0.011521576903760433
[Epoch 16: 7000] Loss: 0.004387249238789082
[Epoch 16: 8000] Loss: 2.0670351659646258e-05
[Epoch 16: 9000] Loss: 0.005170539487153292
Train Epoch Loss (Avg): 0.0545581430842287
Validation Epoch Accuracy:72.2
[Epoch 17: 1000] Loss: 0.22778069972991943
[Epoch 17: 2000] Loss: 0.015964072197675705
[Epoch 17: 3000] Loss: 5.018395677325316e-05
[Epoch 17: 4000] Loss: 0.0003521873150020838
[Epoch 17: 5000] Loss: 0.007924935780465603
[Epoch 17: 6000] Loss: 0.003676327411085367
[Epoch 17: 7000] Loss: 0.00286726257763803
[Epoch 17: 8000] Loss: 2.50338939622452e-06
[Epoch 17: 9000] Loss: 0.0016335969557985663
Train Epoch Loss (Avg): 0.056758010431596964
Validation Epoch Accuracy:71.64
[Epoch 18: 1000] Loss: 0.0039728921838104725
[Epoch 18: 2000] Loss: 8.064766007009894e-05
[Epoch 18: 3000] Loss: 0.05002915859222412
[Epoch 18: 4000] Loss: 0.007583253085613251
[Epoch 18: 5000] Loss: 0.000753117143176496
[Epoch 18: 6000] Loss: 2.2291558707365766e-05
[Epoch 18: 7000] Loss: 0.28286388516426086
[Epoch 18: 8000] Loss: 1.7404177924618125e-05
[Epoch 18: 9000] Loss: 0.0007566021522507071
Train Epoch Loss (Avg): 0.046448715660686896
Validation Epoch Accuracy:71.04
[Epoch 19: 1000] Loss: 0.0006841083522886038
[Epoch 19: 2000] Loss: 0.0015126210637390614
[Epoch 19: 3000] Loss: 1.492446335760178e-05
[Epoch 19: 4000] Loss: 0.01308728288859129
[Epoch 19: 5000] Loss: 0.0007415938889607787
[Epoch 19: 6000] Loss: 0.011963007971644402
```

```
[Epoch 19: 7000] Loss: 0.0026111118495464325
[Epoch 19: 8000] Loss: 0.0013404979836195707
[Epoch 19: 9000] Loss: 0.0009425074094906449
Train Epoch Loss (Avg): 0.04433458956788221
Validation Epoch Accuracy:71.48
[Epoch 20: 1000] Loss: 0.001125797163695097
[Epoch 20: 2000] Loss: 0.038346003741025925
[Epoch 20: 3000] Loss: 0.00010809725063154474
[Epoch 20: 4000] Loss: 0.010262656025588512
[Epoch 20: 5000] Loss: 4.162545155850239e-05
[Epoch 20: 6000] Loss: 9.899096039589494e-05
[Epoch 20: 7000] Loss: 0.005860269535332918
[Epoch 20: 8000] Loss: 0.000637676625046879
[Epoch 20: 9000] Loss: 6.948698137421161e-05
Train Epoch Loss (Avg): 0.04371596721373543
Validation Epoch Accuracy:72.22
Done training!
Accuracy: 71.37%
Accuracy for Class Label 0: 78.2
Accuracy for Class Label 1: 81.39999999999999
Accuracy for Class Label 2: 62.2
Accuracy for Class Label 3: 45.300000000000004
Accuracy for Class Label 4: 63.7
Accuracy for Class Label 5: 63.3
Accuracy for Class Label 6: 84.6
Accuracy for Class Label 7: 74.0
Accuracy for Class Label 8: 85.39999999999999
Accuracy for Class Label 9: 75.6
```

Now run the classifier for the code using the alexnet model specified above. You should notice a notable performance increase. On a GPU, this trained for about 30 mins.

[12]:
```
experiment_name = 'alexnet_debug'  #Provide name to model experiment
model_name = 'alexnet' #Choose between [basic, alexnet]
batch_size = 5  #You may not need to change this but incase you do
first_run = True #whether or not first time running it

trainloader, valloader, testloader, class_names =␣
 ↪get_dataloader(batch_size=batch_size, model_name=model_name)
dataloaders = {'train': trainloader, 'val' : valloader, 'test': testloader,␣
 ↪'mapping': class_names}

#model = models.alexnet(pretrained=True)
if model_name == 'basic':

    model = GradBasicNet()

elif model_name == 'alexnet':
```

```
    model = GradAlexNet()

else:
    raise NotImplementedError("This option has not been implemented. Choose␣
 ↪between 'basic' and 'alexnet' ")

classifier = Classifier(experiment_name, model, dataloaders, class_names,␣
 ↪use_cuda=True)
```

```
Files already downloaded and verified
Files already downloaded and verified
0.weight
0.bias
3.weight
3.bias
6.weight
6.bias
8.weight
8.bias
10.weight
10.bias
```

[13]:
```
# When you develop your code, to save your time, you can choose to run the model
# for 3 cpoches. The accuracy after training for 3 epoches has already been high
# and close to the model after 20-epoch traiing.

# To note, for your final submission, it is recommended to run the model for 20␣
 ↪epochs.
# However, if it takes time, you should at least run the model for 5 epochs.

# classifier.train(epochs=3) # For your reference
classifier.train(epochs=20)
classifier.evaluate()
```

```
[Epoch 1: 1000] Loss: 0.900800347328186
[Epoch 1: 2000] Loss: 0.8746529817581177
[Epoch 1: 3000] Loss: 0.5221883058547974
[Epoch 1: 4000] Loss: 0.1240619570016861
[Epoch 1: 5000] Loss: 0.6995822191238403
[Epoch 1: 6000] Loss: 0.5663000345230103
[Epoch 1: 7000] Loss: 0.40648913383483887
[Epoch 1: 8000] Loss: 1.5863357782363892
[Epoch 1: 9000] Loss: 1.241487979888916
Train Epoch Loss (Avg): 0.853360242331116
Validation Epoch Accuracy:80.44
[Epoch 2: 1000] Loss: 1.934861421585083
[Epoch 2: 2000] Loss: 0.13458743691444397
```

```
[Epoch 2: 3000] Loss: 0.6234680414199829
[Epoch 2: 4000] Loss: 0.8939400911331177
[Epoch 2: 5000] Loss: 0.17099061608314514
[Epoch 2: 6000] Loss: 0.7612079381942749
[Epoch 2: 7000] Loss: 1.1172593832015991
[Epoch 2: 8000] Loss: 0.3519730567932129
[Epoch 2: 9000] Loss: 0.28895193338394165
Train Epoch Loss (Avg): 0.5254577976211311
Validation Epoch Accuracy:83.76
[Epoch 3: 1000] Loss: 0.4217618405818939
[Epoch 3: 2000] Loss: 0.03785059601068497
[Epoch 3: 3000] Loss: 0.11643385887145996
[Epoch 3: 4000] Loss: 0.3456801474094391
[Epoch 3: 5000] Loss: 0.23408587276935577
[Epoch 3: 6000] Loss: 0.05331174656748772
[Epoch 3: 7000] Loss: 0.10111900418996811
[Epoch 3: 8000] Loss: 0.030931467190384865
[Epoch 3: 9000] Loss: 0.004304581321775913
Train Epoch Loss (Avg): 0.41669621227255915
Validation Epoch Accuracy:85.82
[Epoch 4: 1000] Loss: 0.32429441809654236
[Epoch 4: 2000] Loss: 0.07733885198831558
[Epoch 4: 3000] Loss: 0.6808425188064575
[Epoch 4: 4000] Loss: 0.3023378252983093
[Epoch 4: 5000] Loss: 0.18488173186779022
[Epoch 4: 6000] Loss: 0.11451826244592667
[Epoch 4: 7000] Loss: 0.03869118541479111
[Epoch 4: 8000] Loss: 0.4221018850803375
[Epoch 4: 9000] Loss: 0.02779165469110012
Train Epoch Loss (Avg): 0.3400439777661636
Validation Epoch Accuracy:85.9
[Epoch 5: 1000] Loss: 0.021928995847702026
[Epoch 5: 2000] Loss: 0.7427435517311096
[Epoch 5: 3000] Loss: 0.00040206563426181674
[Epoch 5: 4000] Loss: 1.1534035205841064
[Epoch 5: 5000] Loss: 0.09560342133045197
[Epoch 5: 6000] Loss: 0.6544028520584106
[Epoch 5: 7000] Loss: 0.004064240958541632
[Epoch 5: 8000] Loss: 0.13489696383476257
[Epoch 5: 9000] Loss: 0.21985892951488495
Train Epoch Loss (Avg): 0.27919685249180154
Validation Epoch Accuracy:85.66
[Epoch 6: 1000] Loss: 0.07241272926330566
[Epoch 6: 2000] Loss: 0.1988300383090973
[Epoch 6: 3000] Loss: 0.2986968457698822
[Epoch 6: 4000] Loss: 0.07790499180555344
[Epoch 6: 5000] Loss: 0.09562958776950836
[Epoch 6: 6000] Loss: 0.014206988736987114
```

```
[Epoch 6: 7000] Loss: 0.48977822065353394
[Epoch 6: 8000] Loss: 0.8435360193252563
[Epoch 6: 9000] Loss: 0.398817241191864
Train Epoch Loss (Avg): 0.23901012240194672
Validation Epoch Accuracy:86.66
[Epoch 7: 1000] Loss: 0.8948383331298828
[Epoch 7: 2000] Loss: 0.00021748675499111414
[Epoch 7: 3000] Loss: 0.1407715082168579
[Epoch 7: 4000] Loss: 0.4846281409263611
[Epoch 7: 5000] Loss: 0.012136803939938545
[Epoch 7: 6000] Loss: 0.28601497411727905
[Epoch 7: 7000] Loss: 0.297566682100296
[Epoch 7: 8000] Loss: 0.42006438970565796
[Epoch 7: 9000] Loss: 0.002700451295822859
Train Epoch Loss (Avg): 0.19736509846086334
Validation Epoch Accuracy:86.76
[Epoch 8: 1000] Loss: 0.02304180897772312
[Epoch 8: 2000] Loss: 0.7363632917404175
[Epoch 8: 3000] Loss: 0.00950632058084011
[Epoch 8: 4000] Loss: 0.17087696492671967
[Epoch 8: 5000] Loss: 0.03370711952447891
[Epoch 8: 6000] Loss: 0.8534795045852661
[Epoch 8: 7000] Loss: 0.06144728511571884
[Epoch 8: 8000] Loss: 1.016886591911316
[Epoch 8: 9000] Loss: 0.000865709618665278
Train Epoch Loss (Avg): 0.16708435845620573
Validation Epoch Accuracy:86.98
[Epoch 9: 1000] Loss: 0.7810961008071899
[Epoch 9: 2000] Loss: 0.5823220610618591
[Epoch 9: 3000] Loss: 0.45389240980148315
[Epoch 9: 4000] Loss: 0.005158939398825169
[Epoch 9: 5000] Loss: 1.2862999439239502
[Epoch 9: 6000] Loss: 0.006002237554639578
[Epoch 9: 7000] Loss: 0.0007602741243317723
[Epoch 9: 8000] Loss: 0.1297457069158554
[Epoch 9: 9000] Loss: 0.1760658472776413
Train Epoch Loss (Avg): 0.14006514479532883
Validation Epoch Accuracy:86.62
[Epoch 10: 1000] Loss: 0.6788601875305176
[Epoch 10: 2000] Loss: 0.01068951841443777
[Epoch 10: 3000] Loss: 0.014035874977707863
[Epoch 10: 4000] Loss: 0.13345420360565186
[Epoch 10: 5000] Loss: 0.08882652223110199
[Epoch 10: 6000] Loss: 0.06099139526486397
[Epoch 10: 7000] Loss: 0.0007792532560415566
[Epoch 10: 8000] Loss: 0.01025826670229435
[Epoch 10: 9000] Loss: 0.0019521958893164992
Train Epoch Loss (Avg): 0.12412469682267278
```

```
Validation Epoch Accuracy:86.54
[Epoch 11: 1000] Loss: 0.0002562324807513505
[Epoch 11: 2000] Loss: 0.08755121380090714
[Epoch 11: 3000] Loss: 0.18455040454864502
[Epoch 11: 4000] Loss: 0.0015137714799493551
[Epoch 11: 5000] Loss: 0.16459038853645325
[Epoch 11: 6000] Loss: 0.035647593438625336
[Epoch 11: 7000] Loss: 0.0056436434388160706
[Epoch 11: 8000] Loss: 0.00655529135838151
[Epoch 11: 9000] Loss: 0.043333958834409714
Train Epoch Loss (Avg): 0.10953050222443196
Validation Epoch Accuracy:87.34
[Epoch 12: 1000] Loss: 0.042890764772892
[Epoch 12: 2000] Loss: 0.0012238083872944117
[Epoch 12: 3000] Loss: 0.10142247378826141
[Epoch 12: 4000] Loss: 0.0032226010225713253
[Epoch 12: 5000] Loss: 0.0004397241282276809
[Epoch 12: 6000] Loss: 0.0027083733584731817
[Epoch 12: 7000] Loss: 0.020443622022867203
[Epoch 12: 8000] Loss: 0.0004493575543165207
[Epoch 12: 9000] Loss: 0.05720806121826172
Train Epoch Loss (Avg): 0.0963194829538691
Validation Epoch Accuracy:87.84
[Epoch 13: 1000] Loss: 0.0009993982966989279
[Epoch 13: 2000] Loss: 0.0001638838875805959
[Epoch 13: 3000] Loss: 0.27431216835975647
[Epoch 13: 4000] Loss: 2.567722913227044e-05
[Epoch 13: 5000] Loss: 0.005479642190039158
[Epoch 13: 6000] Loss: 7.249309419421479e-05
[Epoch 13: 7000] Loss: 0.0055174147710204124
[Epoch 13: 8000] Loss: 0.005508631467819214
[Epoch 13: 9000] Loss: 0.00026355573209002614
Train Epoch Loss (Avg): 0.08669428245710381
Validation Epoch Accuracy:87.34
[Epoch 14: 1000] Loss: 0.04353197664022446
[Epoch 14: 2000] Loss: 0.004364088177680969
[Epoch 14: 3000] Loss: 9.15579657885246e-05
[Epoch 14: 4000] Loss: 0.0759681761264801
[Epoch 14: 5000] Loss: 0.047907836735248566
[Epoch 14: 6000] Loss: 0.0025883850175887346
[Epoch 14: 7000] Loss: 0.38393428921699524
[Epoch 14: 8000] Loss: 0.017977604642510414
[Epoch 14: 9000] Loss: 0.00917638186365366
Train Epoch Loss (Avg): 0.0753164661753414
Validation Epoch Accuracy:87.64
[Epoch 15: 1000] Loss: 1.186378836631775
[Epoch 15: 2000] Loss: 0.044163189828395844
[Epoch 15: 3000] Loss: 0.0017103452701121569
```

```
[Epoch 15: 4000] Loss: 0.06990233063697815
[Epoch 15: 5000] Loss: 0.005790588445961475
[Epoch 15: 6000] Loss: 0.16004127264022827
[Epoch 15: 7000] Loss: 0.23076558113098145
[Epoch 15: 8000] Loss: 0.004823838360607624
[Epoch 15: 9000] Loss: 0.020988570526242256
Train Epoch Loss (Avg): 0.06964746869455932
Validation Epoch Accuracy:87.08
[Epoch 16: 1000] Loss: 0.021626751869916916
[Epoch 16: 2000] Loss: 0.02345733717083931
[Epoch 16: 3000] Loss: 0.0030901399441063404
[Epoch 16: 4000] Loss: 0.0009080382296815515
[Epoch 16: 5000] Loss: 0.07276405394077301
[Epoch 16: 6000] Loss: 0.0017753075808286667
[Epoch 16: 7000] Loss: 0.00203865859657526
[Epoch 16: 8000] Loss: 0.001237576361745596
[Epoch 16: 9000] Loss: 0.0006276428466662765
Train Epoch Loss (Avg): 0.059869062581892826
Validation Epoch Accuracy:86.68
[Epoch 17: 1000] Loss: 0.0007305651670321822
[Epoch 17: 2000] Loss: 0.0046820747666060925
[Epoch 17: 3000] Loss: 0.02255082316696644
[Epoch 17: 4000] Loss: 0.0006501612369902432
[Epoch 17: 5000] Loss: 0.0010400604223832488
[Epoch 17: 6000] Loss: 0.005981555208563805
[Epoch 17: 7000] Loss: 1.0490396107343258e-06
[Epoch 17: 8000] Loss: 1.0442480743222404e-05
[Epoch 17: 9000] Loss: 0.05089370533823967
Train Epoch Loss (Avg): 0.05637745734020978
Validation Epoch Accuracy:87.84
[Epoch 18: 1000] Loss: 0.7586684823036194
[Epoch 18: 2000] Loss: 0.05034303665161133
[Epoch 18: 3000] Loss: 0.010383526794612408
[Epoch 18: 4000] Loss: 0.008767670020461082
[Epoch 18: 5000] Loss: 7.67829260439612e-05
[Epoch 18: 6000] Loss: 6.198877713359252e-07
[Epoch 18: 7000] Loss: 0.043259043246507645
[Epoch 18: 8000] Loss: 0.00017843949899543077
[Epoch 18: 9000] Loss: 0.0006920318119227886
Train Epoch Loss (Avg): 0.05251324371277371
Validation Epoch Accuracy:86.74
[Epoch 19: 1000] Loss: 0.0001625808363314718
[Epoch 19: 2000] Loss: 0.0036474987864494324
[Epoch 19: 3000] Loss: 0.0019604037515819073
[Epoch 19: 4000] Loss: 0.00437150988727808
[Epoch 19: 5000] Loss: 0.0007999733206816018
[Epoch 19: 6000] Loss: 0.6970851421356201
[Epoch 19: 7000] Loss: 0.0024113920517265797
```

```
[Epoch 19: 8000] Loss: 0.017996713519096375
[Epoch 19: 9000] Loss: 0.0006301200482994318
Train Epoch Loss (Avg): 0.04897270775095918
Validation Epoch Accuracy:88.62
[Epoch 20: 1000] Loss: 0.0009317377698607743
[Epoch 20: 2000] Loss: 0.003337359754368663
[Epoch 20: 3000] Loss: 0.002277124673128128
[Epoch 20: 4000] Loss: 0.00133138510864228
[Epoch 20: 5000] Loss: 1.1444059282439412e-06
[Epoch 20: 6000] Loss: 0.285722941160202
[Epoch 20: 7000] Loss: 0.011542106047272682
[Epoch 20: 8000] Loss: 0.0005023442208766937
[Epoch 20: 9000] Loss: 8.157146658049896e-05
Train Epoch Loss (Avg): 0.04147615693376899
Validation Epoch Accuracy:87.86
Done training!
Accuracy: 87.87%
Accuracy for Class Label 0: 91.7
Accuracy for Class Label 1: 94.5
Accuracy for Class Label 2: 85.2
Accuracy for Class Label 3: 75.4
Accuracy for Class Label 4: 90.4
Accuracy for Class Label 5: 79.0
Accuracy for Class Label 6: 90.8
Accuracy for Class Label 7: 87.6
Accuracy for Class Label 8: 92.80000000000001
Accuracy for Class Label 9: 91.3
```

Before you move forward to the next step: Try and see what classes your model does well on (you can modify the testing code for this). This will help you pick the best visualization to show later.

Analysis: Class Label 3 has the least accuracy in the test dataset on which the evaluation is being performed. This class 3 label corresponds to the label of "cat".

1.8 Understand the Deep Neural Networks

The deep learning is quite powerful and can achieve great performance after training for a few epochs. However, we still don't know why it works. Thus, we first study activation map to analysis the region that triggers the final classification. Then, we visualze a few kernels to infer what is learned in the neural network. (The activtion map and kernel visualization are two popurlar directions. However, much more effort is still needed to comprehensively understand deep neural networks and reach the ultimate goal.)

1.8.1 Activation Map

For interpreting our alexnet model, we will be using a simple version of Grad-CAM (Gradient based Class activation mapping) by Selvaraju et al. (https://arxiv.org/abs/1610.02391). This will help us see what region of the input image the output is 'focusing on' while making its key choice. I

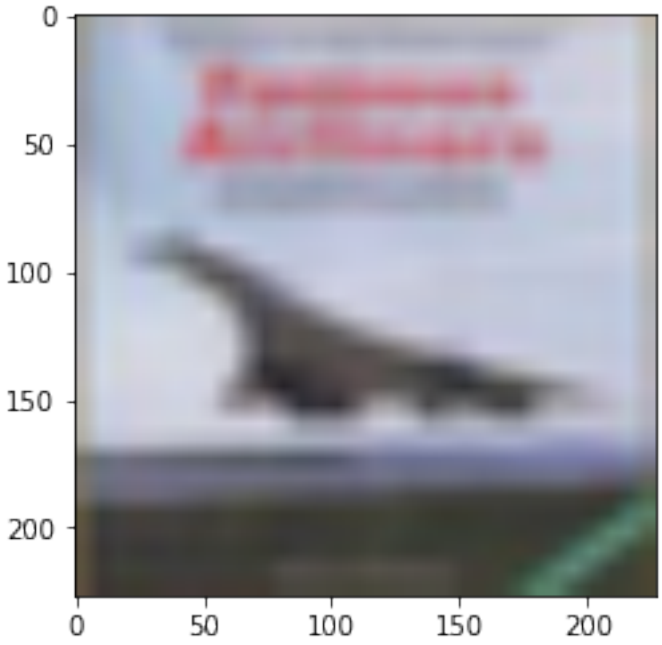
recommend reading the paper for those interested: however, the following instructions should also suffice. If you go step-by-step then all information is given in the intructions. You will receive one img to the method.

1. First, move the img to cuda if you are using GPU.
2. The code to load the model trained from before has been provided to you. Use self.model to output the predictions to the variable out. Your output should have dimension (1, 10).
3. The predicted class is the index of the highest value of of these 10 values. Use **torch.max** along dim 1 to get the argument of the max value. This method will return two values, the latter of them is the required argument. The next two lines have been provided to you: they indicate the predicted loss
4. Call the **backward** method on out[:, pred]. Previously during the forward passes, we have applied a gradient hook on the last convolutional layer. This will mean that during the applied backward pass, we will record the value of the gradient for the **maximum predicted class** with respect to the **output** of the last convolutional layer. This will be the same size as the **output** of the last convolutional layer. Do you see why?
5. After the **backward** call, get the value of the gradient above using the **get_gradient_activations** function on **self.model** and store it in gradients. Now use the torch.mean method to get the mean value of gradients across all dimensionss except the channels dimension (number of filters) and store this in **mean_gradients**. Your output should be of shape (1, 64, 1, 1). If your tensors have been on GPU, you should move them to CPU using .detach().cpu()
6. Use the self.model.get_final_conv_layer(img) to store the activations at the final layer to activations. This should be of shape (1, 64, H, W).
7. Now for each of the 64 filters, **scale** the activations at that filter, with the corresponding **mean_gradients** value for that filter. Your output should have size (1, 64, H, W) Iterate over the 64 filters in the following way: `for idx in range(activations.shape[1]):`
8. As a final step, we will take mean across the 64 filters and do a ReLU to get rid of negative activations before normalzing one last time to get the heatmap. This has been done for you.

[16]:
```
'''Sample an image from the test set'''

#You may change the sampling code to sample an image as you desire.
#Make sure to NOT move the sampling code to a different cell.

img_batch, labels_batch = next(iter(testloader))
img = img_batch[3]
img = img.unsqueeze(0)

classifier = Classifier(experiment_name, model, dataloaders, class_names,␣
 ↪use_cuda=True)
heatmap = classifier.grad_cam_on_input(img)

def visualize(img, heatmap):
    heatmap = heatmap.cpu().numpy()

    img = show_img(img)
    img = np.uint8(255 * img)
```

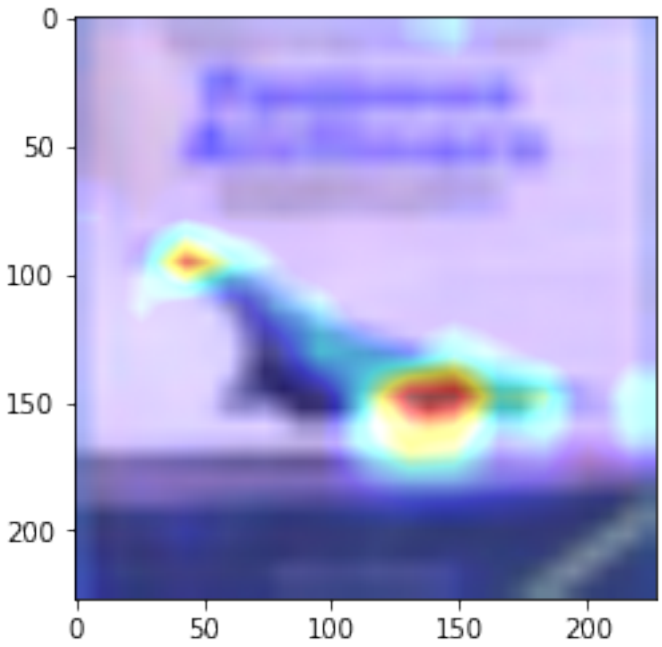
```
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    print(img.shape)
    print(heatmap.shape)
    heatmap = cv2.resize(heatmap, (img.shape[1], img.shape[0]))
    heatmap = np.uint8(255 * heatmap)
    heatmap = cv2.applyColorMap(heatmap, cv2.COLORMAP_JET)
    heatmap = cv2.cvtColor(heatmap, cv2.COLOR_BGR2RGB)

    combine = 0.5 * heatmap + img

    plt.imshow(combine/255)
    plt.show()


visualize(img, heatmap)
```

```
Predicted class was plane
Gradients shape:  torch.Size([1, 256, 13, 13])
Activations shape:  torch.Size([1, 256, 13, 13])
```




```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

```
(227, 227, 3)
(13, 13)
```

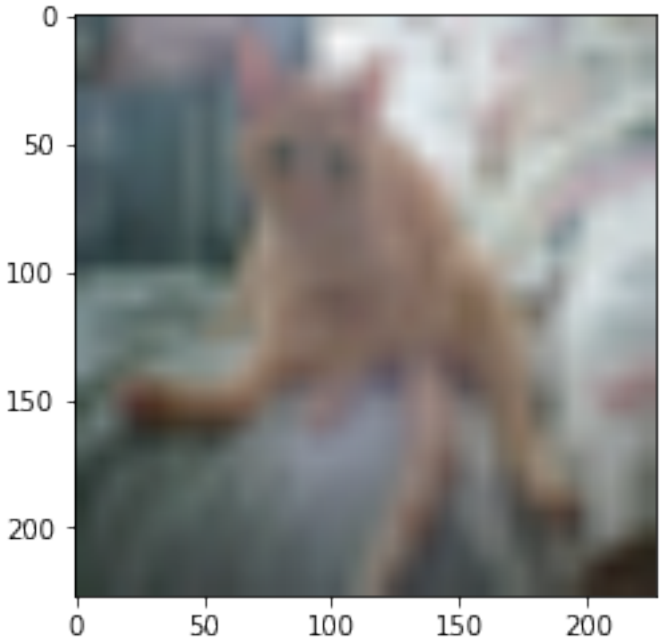


What do you observe? Show an example of an image where the method is looking at the object in question and another where it appear to be completely unrelated. In the latter case, it might have learnt a spurious correlation- aka a bias in the data which always appears to be correlated with a given label. For the ship class, this **might** be the surrounding water or for a **horse** it might be the surrounding grass. In such cases, do you think the model would predict correctly for a ship on sand or a horse in the air? Answer in a text snippet below. **Causally trained neural networks** (https://www.cmu.edu/dietrich/causality/neurips20ws/) are an exciting direction to solve this problem

[35]:
```
# Your code here to show an failure case.
# You can refer the steps and functions implemented in the previous cell and␣
 ↪reuse them.
import random

wrongly_classified_class = 3
matched_data = list()

for batch in testloader:
    img_batch, labels_batch = batch
    matched_data.extend([img_batch[idx] for idx, label in␣
 ↪enumerate(labels_batch) \
                         if int(label) == wrongly_classified_class])

img = matched_data[random.choice(list(range(len(matched_data))))]
img = img.unsqueeze(0)
```

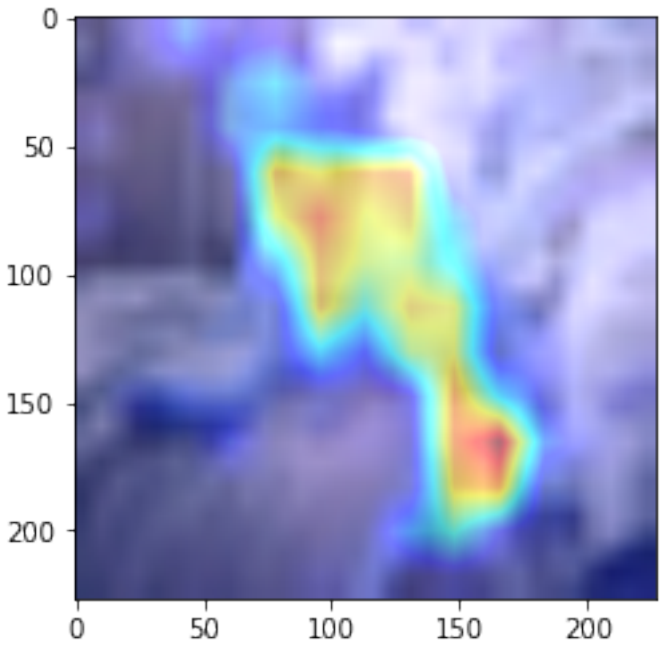
```
classifier = Classifier(experiment_name, model, dataloaders, class_names,␣
 ↪use_cuda=True)
heatmap = classifier.grad_cam_on_input(img)

visualize(img, heatmap)
```

```
Predicted class was horse
Gradients shape:  torch.Size([1, 256, 13, 13])
Activations shape:  torch.Size([1, 256, 13, 13])
```




```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

```
(227, 227, 3)
(13, 13)
```



Answer: In the prediction of a plane above, we can see that the model sees the wings of the plane as something important along with the sky behind it. This is mainly used to classify the image as having a plane in it. Also, as we can see here that the plane prediction is also dependent on having a sky in the neighborhood of the wings of the plane, it seems that the model prediction here not only depends on the object but also on the surrounding. If that's the case here, then the model won't always be able to predict a ship on sand correctly or even horse in the air correctly. Also, regarding the failure case above, we can see that the image contains a cat but the model predicts it to be a horse. I chose this class mainly as the predictions for the "cat" class had the least accuracy in terms of the model trained above. The accuracies for all the class predictions is shown above in the model training and evaluation outputs.

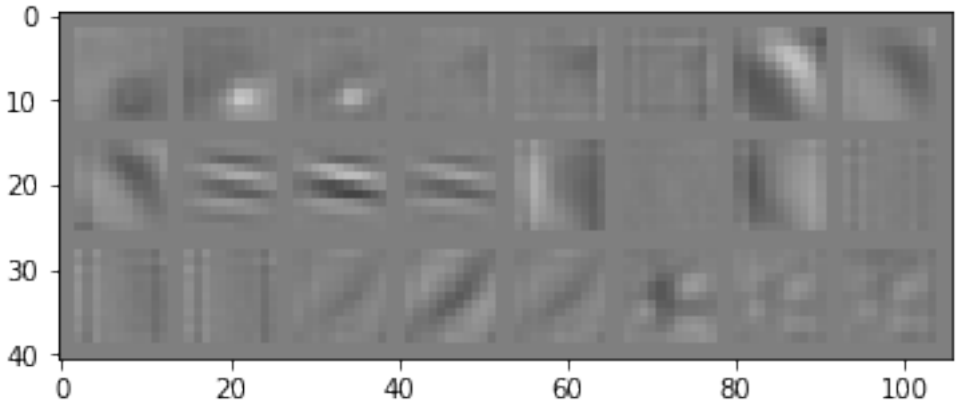
1.8.2 Kernel and Activation Visualizations

We will visualize some learned convolutional kernels for two layers in the conv-net. Study the code provided for **trained_kernel_viz** carefully. You only have to fill out the line for **filter**. All you are expected to do is to access the relevantt layer from **self.conv_model** and set **filter** equal to its weight parameter.

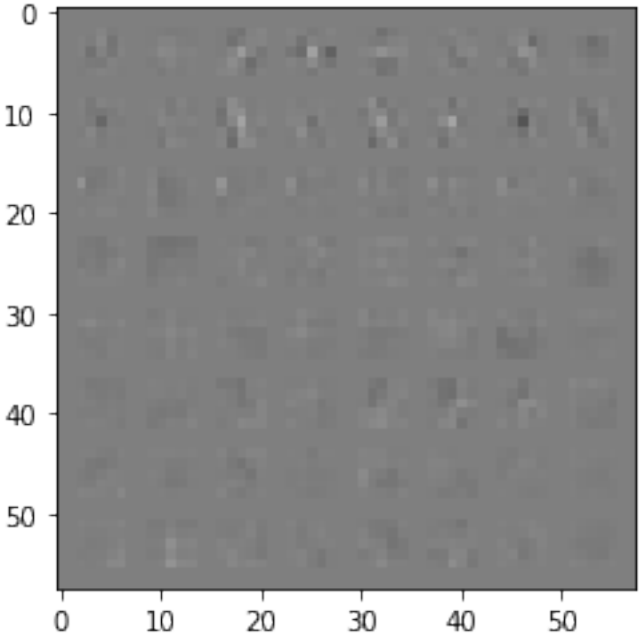
Call this function on the alexnet classifier. What do you observe?

[17]:
```
classifier.trained_kernel_viz()
```

```
torch.Size([8, 3, 11, 11])
```




```
torch.Size([8, 8, 5, 5])
```



Answer: 1. The first kernel trained with kernel visualizations shown above tells us that the kernel here is trying to capture the coarse details like the change in gradient as a line. Basically, this kernel tries and capture the edges in the image. 2. The second kernel trained with the kernel visualizations shown above tells us that the kernel is trying to capture a little more complex details and/or shapes in the image. This makes complete sense as the second kernel should try and capture more complex details in the image after the coarse details like lines/edges has been captured by the first kernel.

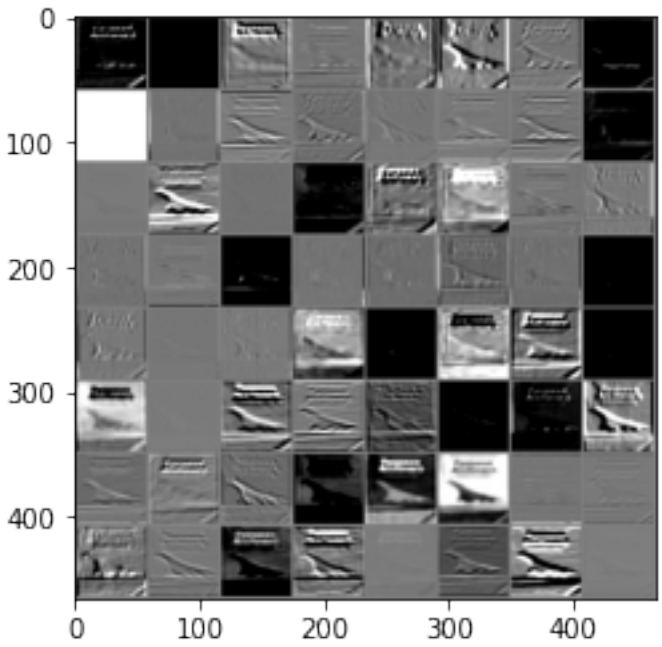
Now with the kernel viz filled out, write the method for activation visualizations **activations on**

input. The structure for the code is very similar to the kernel viz, except that we are actually viewing the output of the model at each stage and not for the kernel at that stage. Once filled, please call this method on a few sample images. What do you observe? Answer generally in a text snippet below.

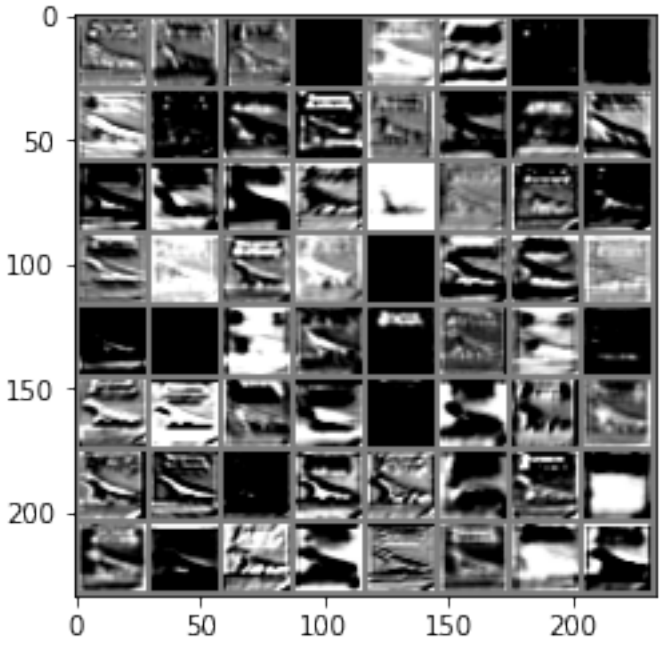
Answer: 1. It is observed that the outputs from the first few layers looks a lot like the original image. For instance, the outputs from the first layer shows that the edges have been detected (horizontal and vertical) and some coarse features have also been detected. 2. As we go deeper in the network, we can see that the outputs of the layers start capturing finer details and patterns in the images rather than coarse details like edges and shapes. 3. Also, the outputs of the layers clearly show us the correlation between the kernel learnt and the outputs formed. This correlation helps us interpret things like that the first few layers help capture the feautures like edges and shapes. Moving deeper, the model layers tries and learns to capture the details like small objects in the image which help us define class of the image on the whole.

[18]:
```
'''Sample an image from the test set'''
#You may change the sampling code to sample an image as you desire.
#Make sure to NOT move the sampling code to a different cell.
img_batch, labels_batch = next(iter(testloader))
img = img_batch[3]
img = img.unsqueeze(0)

classifier = Classifier(experiment_name, model, dataloaders, class_names,␣
↪use_cuda=True)
classifier.activations_on_input(img)
```

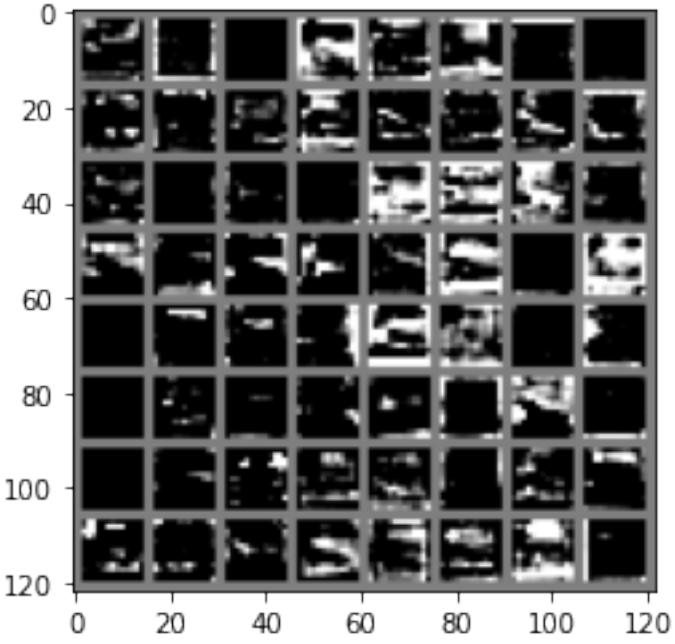
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



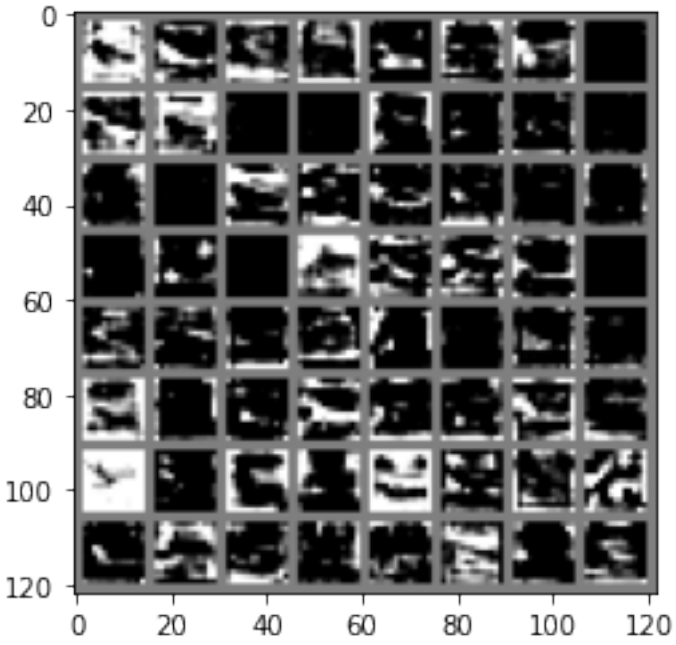
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



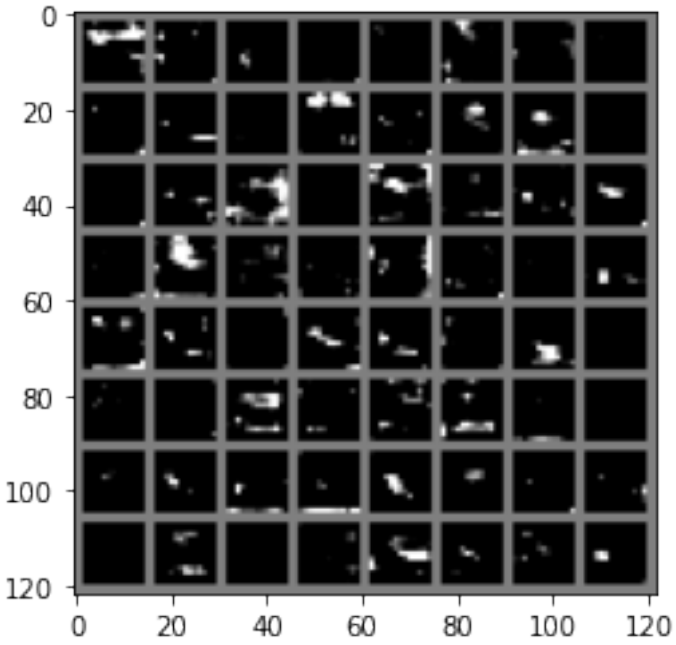

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```




```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



What do you observe about early layers v. later layers? Answer in a text snippet below.

Answer:

Early layers are basically learning to detect the features like edges/line in the images. On the other hand, later layers try and capture more finer details like shapes, then particular objects in the images and smaller but significant details in the images. This kind of learning on the whole lets us classify the image in a more comprehensive way. The first few layers helps us capture or detect the global features in the image like edges and shapes. The later layer helps us detect the local features in the image like some specific objects in the image or some complex shapes in the image.