Problem Set 6: Neural Networks

Warning! Some of the problems in this problem set require heavy computation - you are encouraged to start early so that you don't get stuck at the last minute.

Note: If you are using Google Colab for this problem set, make sure to turn off the Gemini assistant.

Note: Lab 9 is your friend -- many of the examples will help you answers in this problem set.

Truck v/s Cars: Neural Networks and Image Classification

Your goal for this problem set is to train neural network models for image classification. Specifically, your task is to train models that correctly predict where the vehicle in a given image is a truck, or a car / automobile.

It might be useful to start by implementing this entire problem set on a relatively small subset of all of the images first, before using the full dataset.

From a coding perspective, it will be easiest to complete all the problems using pytorch; however, if you prefer to use keras / tensorflow or any other deeplearning API, feel free to do so.

Part 0

Question 1 - upload your ipynb to bcourses

Question 2 - upload your pdf to bcourses

Question 3 - please provide a summary of any resources consulted, and people with whom you worked in the completion of the problem set.

Highlight any specific LLM prompts/resources used at the point of use.

Part I.

Question 4: Load Data + Exploratory Analysis

For this problem, we'll load the CIFAR 10 dataset . This dataset has been widely used in ML and computer vision research -- you can read more about the state of the art model performance (and how this has improved over time) here.

The CIFAR 10 dataset originally has 10 classes -- we've provided helper code below to load the data, and remove images belonging to unnecessary classes. We will use this dataset for a supervised binary classification problem.

Your tasks:

- Create a subset of the CIFAR 10 data, keeping only the car (automobile) and truck classes.
- Select 9 random images from your training set. Plot these images in a 3 X 3 grid, along with the corresponding category / label
- Plot the distribution of labels in your training, validation and test sets.

In [1]:
```
from torchvision import datasets, transforms
import torch.nn as nn
import numpy as np
def load_cifar10(data_path):
    """
    Helper code to clean the CIFAR 10 dataset, and remove the unnecessary cl
    """
    class_names = ['airplane','automobile','bird','cat','deer',
                'dog','frog','horse','ship','truck']

    cifar10 = datasets.CIFAR10(
        data_path, train=True, download=True,
        transform=transforms.Compose([
            transforms.ToTensor(),
            # transforms.Normalize((0.4915, 0.4823, 0.4468),
            #                      (0.2470, 0.2435, 0.2616))
        ]))

    cifar10_val = datasets.CIFAR10(
    data_path, train=False, download=True,
    transform=transforms.Compose([
        transforms.ToTensor(),
        # transforms.Normalize((0.4915, 0.4823, 0.4468),
        #                      (0.2470, 0.2435, 0.2616))
    ]))

    return cifar10, cifar10_val
```

In [6]:
```
from torch.utils.data import Subset
import torch
import matplotlib.pyplot as plt
```

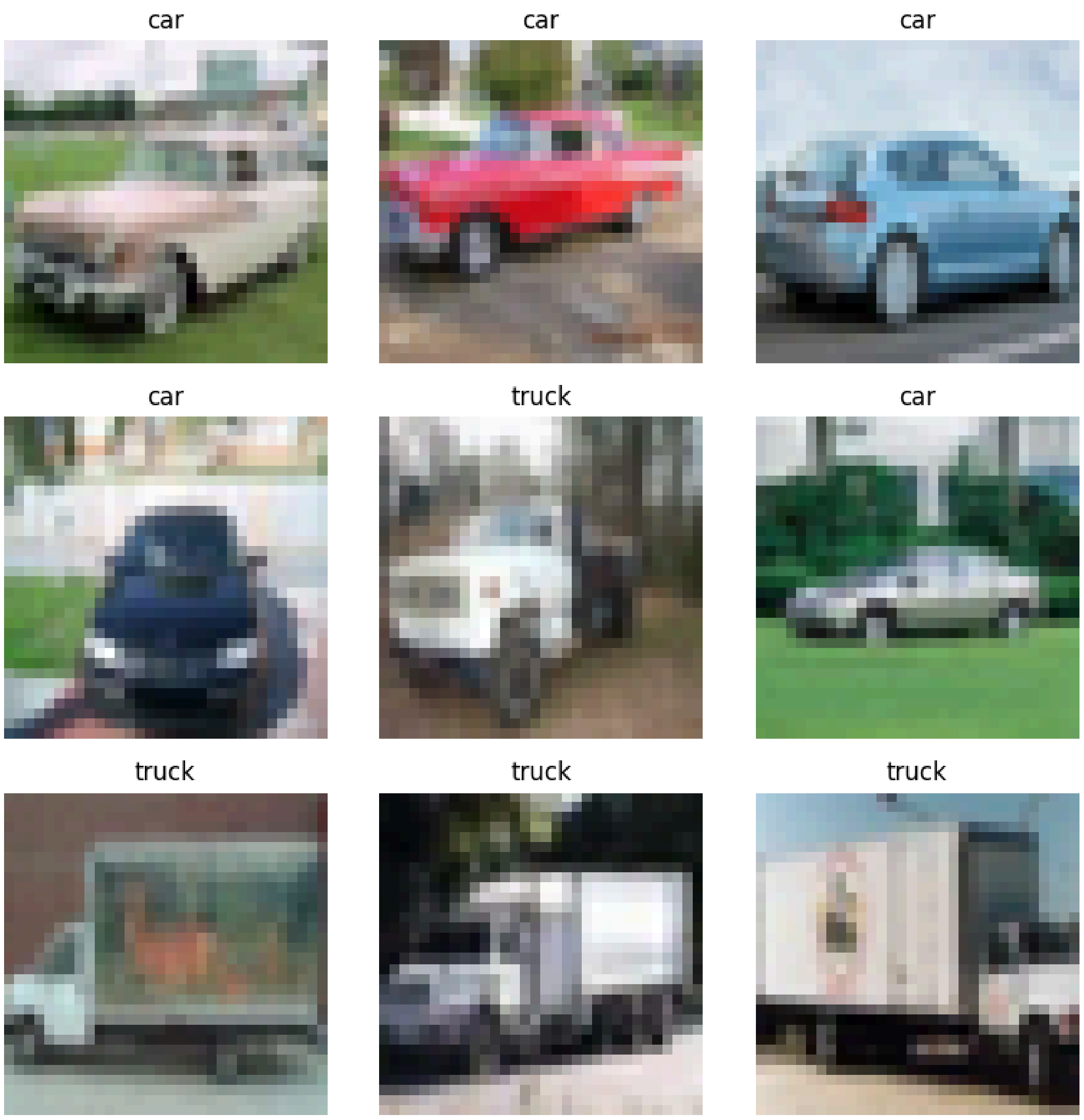
In [2]:
```
## Load CIFAR data
datapath = '../data-unversioned/p1ch6/'
cifar10, cifar10_val = load_cifar10(datapath)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ../da
ta-unversioned/p1ch6/cifar-10-python.tar.gz
100%|█████████████████████████████████████████| 170M/170M [04:08<00:00, 686k
B/s]
Extracting ../data-unversioned/p1ch6/cifar-10-python.tar.gz to ../data-unver
sioned/p1ch6/
Files already downloaded and verified
```

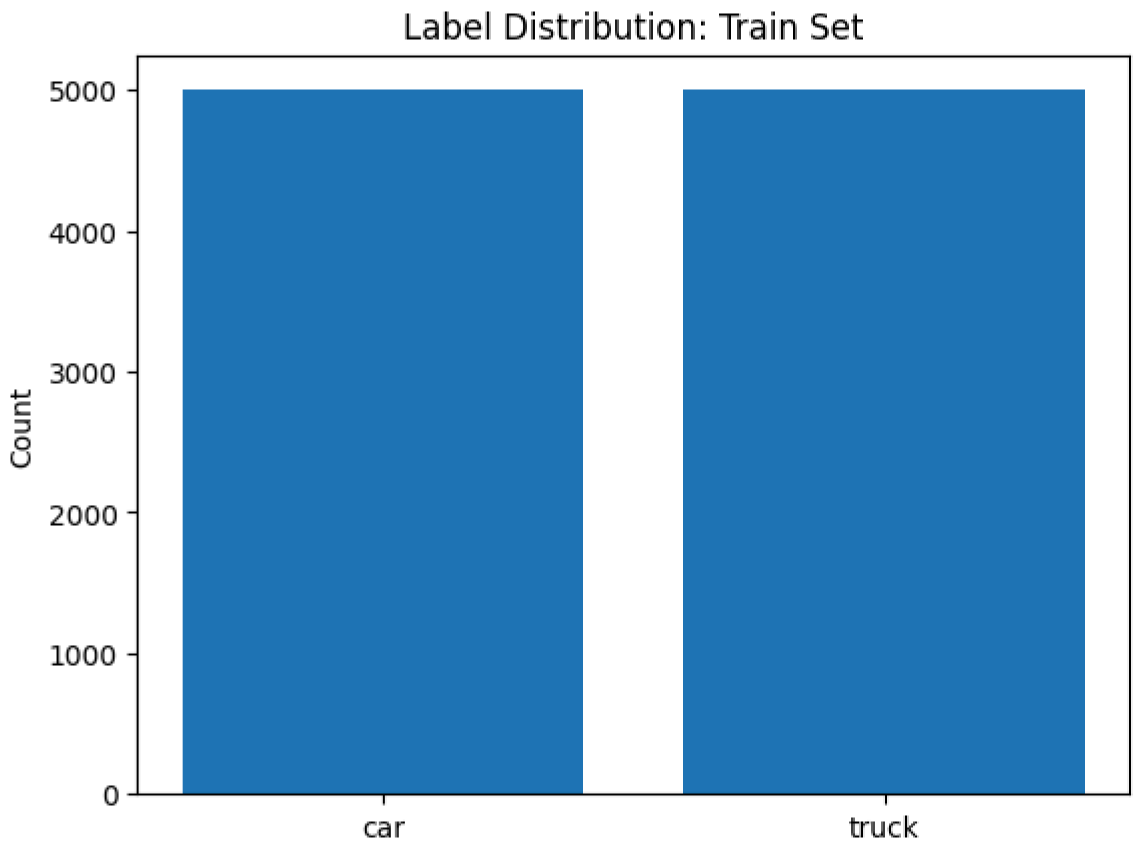
In [27]:
```
def filterCar(dataset):
    indices = []
    targets = []

    for i, (_, label) in enumerate(dataset):
        if label in [1, 9]:  #automobile or truck
            indices.append(i)
            targets.append(0 if label == 1 else 1)  #0 for car, 1 for truck

    filtered_data = Subset(dataset, indices)
    filtered_data.targets = torch.tensor(targets)
    return filtered_data


def random_images(dataset):
    fig, axs = plt.subplots(3, 3, figsize=(8, 8))
    indices = torch.randint(0, len(dataset), (9,))
    labels_map = {1: 'car', 9: 'truck'}
#we wanna run them for all
    for ax, idx in zip(axs.flatten(), indices):
        img, label = dataset[idx]
        ax.imshow(img.permute(1, 2, 0))  #convert from (C, H, W) to (H, W, C
        ax.set_title(labels_map[label])
        ax.axis('off')
    plt.tight_layout()
    plt.show()



def plot_distr(dataset, title=''):
    labels = dataset.targets
    unique, counts = torch.unique(labels, return_counts=True)

    plt.bar(['car', 'truck'], counts.tolist())
    plt.title(f'Label Distribution: {title}')
    plt.ylabel('Count')
    plt.show()
```

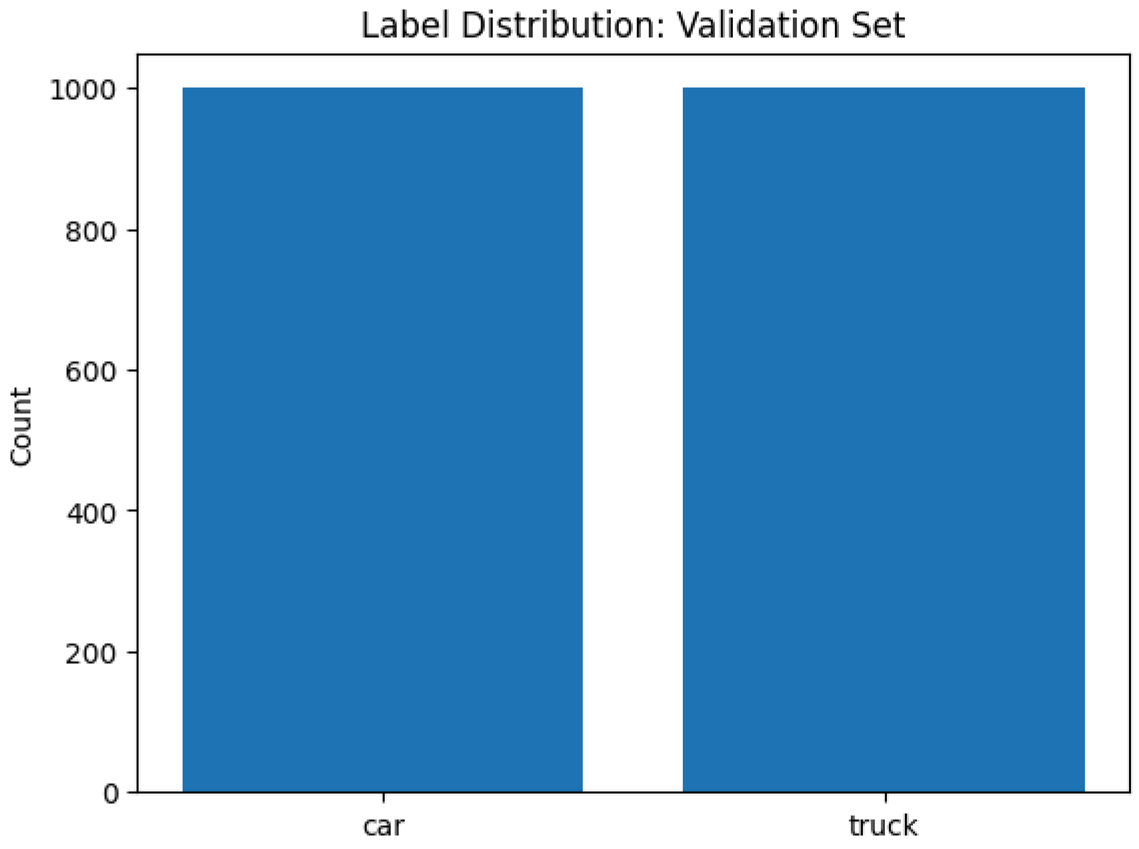
In [5]:
```
fil_train = filterCar(cifar10)
fil_val   = filterCar(cifar10_val)
```

In [26]:
```
random_images(fil_train)
```



```
In [28]: plot_distr(fil_train, title='Train Set')
         plot_distr(fil_val, title='Validation Set')
```





Question 5: Convolutional Filters

Let's build some intuition for what convolutions actually do. First, complete the following steps. Then, answer the questions below.

1. initizialize a convolutional layer (using nn.Conv2d):
   - in_channels: 3
   - out_channels: 1
   - kernel_size: 3
   - padding: 1
2. assign the following weights to the convolutional layer

$$\begin{bmatrix} -1.0 & -1.0 & -1.0 \\ 0.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 1.0 \end{bmatrix}$$

3. Pass the first image from the training dataset through this convolutional layer

4. Plot the raw image, and the resulting output

Then, answer the following questions

1. What do you notice about the output image? After applying the convolutional filter, what kind of features are more prominent / enhanced? Why is this the case (or in other words, what does step 2 help us achieve?)?
2. Here, we are explicitly assigning weights to this convolutional filter. This is never really done in practice; the goal is for the network estimate different filters from data in whatever way the discrimination is most effective. However, this does bring up some questions about different kinds of weight initializations and their consequences:
   - suppose we initialize convolutional layers with all the weights to zero. what would be the consequence?
   - instead, suppose we initialize the weights with random numbers (as is commonly done) -- what are some of the drawbacks of this approach?

Hint 1: Review the documentation for nn.Conv2d
Hint 2: Use conv.weight[:] to set the weights, where "conv" is the name of your convolutional layer.

In [29]:
```
kernel = torch.tensor([
                        [-1.0, -1.0, -1.0],
                        [ 0.0,  0.0,  0.0],
                        [ 1.0,  1.0,  1.0]], dtype=torch.float32)
```

In [30]:
```
conv = nn.Conv2d(in_channels=3, out_channels=1, kernel_size=3, padding=1, bi
```

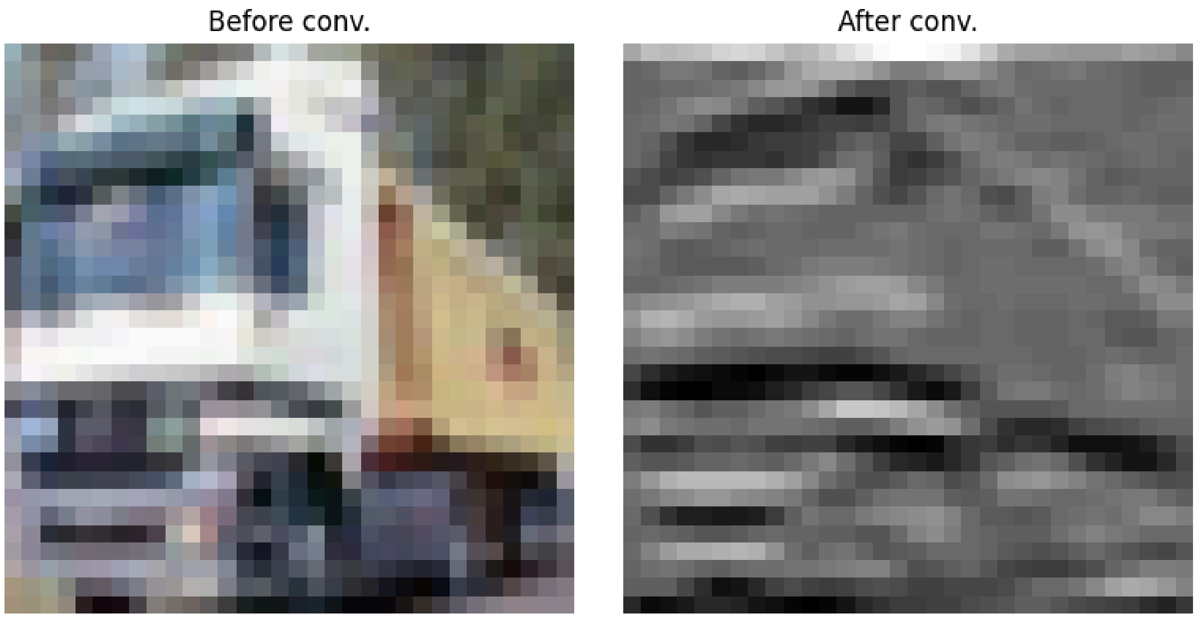
In [31]:
```
with torch.no_grad():
    conv.weight[:] = torch.stack([kernel] * 3).unsqueeze(0)  #here the will
```

In [34]:
```
img, _ = fil_train[0]  #just first image and alsa its label label
img_batch = img.unsqueeze(0)  #batch dim: (1, 3, 32, 32)
print(img_batch.shape)
out = conv(img_batch)  ###output shape will be in (1, 1, 32, 32)
out.shape
```

```
torch.Size([1, 3, 32, 32])
```

Out[34]:
```
torch.Size([1, 1, 32, 32])
```

In [35]:
```
fig, axs = plt.subplots(1, 2, figsize=(8, 4))

axs[0].imshow(img.permute(1, 2, 0))
axs[0].set_title("Before conv.")
axs[0].axis("off")

axs[1].imshow(out.squeeze().detach().numpy(), cmap='gray')
axs[1].set_title("After conv.")
axs[1].axis("off")

plt.tight_layout()
plt.show()
```



II. Neural Networks

Question 6: Helper Functions

Complete the following functions:

- training: which invokes the training loop, similar to what we saw in Lab 9.
- validate: given a trained model and data loader, generate performance metrics

Here is an example of how you will use these functions:

test_loader = ... (Define your data loader)

learning_rate = .. (define learning rate)
neural_network = .. (initialize your neural network)
optimizer = .. (set up your optimizer)
loss_fn = .. (set up your loss function)

training(....)

validate(...)

In [38]:
```
from sklearn.metrics import precision_score, recall_score, f1_score, accurac
from torch.utils.data import DataLoader
```

In [65]:
```
def training(model, n_epochs, optimizer, fn_loss, data_loader):
    """
    Train a neural network model. (i.e this function invokes the training lo
    ----------
    Parameters:

        model: torch model object
        n_epochs: number of epochs
        optimizer: torch.optim object
        fn_loss: torch loss function (torch.nn.modules.loss object)
        data_loader: torch data loader
    ----------
    Returns:
        N/A
    """
    model.train()
    for epoch in range(n_epochs):
        total_loss = 0.0

        for inputs, labels in data_loader:
            optimizer.zero_grad()  #in each loop we wanna clear previous gra

            outputs = model(inputs)  # forward pass

            labels = labels.float().unsqueeze(1)  #shape will be (batch_size
            # print(labels.shape)
            loss = fn_loss(outputs, labels)  # compute loss

            loss.backward()  # #backpropag.
            optimizer.step()  ##update weights

            total_loss += loss.item()

        avg_loss = total_loss / len(data_loader)
        print(f"Epoch [{epoch+1}/{n_epochs}] - Loss: {avg_loss:.4f}")

def validate(model, data_loader):
    """
    Given a model and data loader, generate performance metrics.
    -----------
    Parameters:
        model: (trained) neural network model
        data_loader: torch data loader
    ----------
    Returns:
        Precision, Recall, Accuracy and F1 Score
    """
    model.eval()
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for inputs, labels in data_loader:
            outputs = model(inputs)
            preds = torch.sigmoid(outputs)  # convert logits to probabilitie
            preds = (preds > 0.5).int()  # threshold at 0.5

            all_preds.extend(preds.squeeze().tolist())
            all_labels.extend(labels.tolist())

    precision = precision_score(all_labels, all_preds)
    recall = recall_score(all_labels, all_preds)
    f1 = f1_score(all_labels, all_preds)
    acc = accuracy_score(all_labels, all_preds)

    return precision, recall, acc, f1
```

Question 7: Feedforward Neural Networks

Build a feedforward neural network with 2 hidden layers. You are expected to:

- Complete the FFN Class, based on the network architecture below (some scaffolding for this class is provided below, but feel free to write your own).
- Train / compile the network, following the guidelines below.
- Report the accuracy, precision, recall and f1-score for the training and validation sets.

Network Architecture:

- Input layer dimensions: 32 X 32 X 3 (W X H X C, where W = width, H = Height, C = Number of Channels)
- Hidden layer 1: 256 nodes, relu activation

Compile:

- Optimizer: Adam

- Batch size: 32
- You may run the model for a maximum of 50 epochs.

Here is an example of how to implement your neural network:

test_loader = ... (Define your data loader)

learning_rate = .. (define learning rate)
neural_network = .. (initialize your neural network)
optimizer = .. (set up your optimizer)
loss_fn = .. (set up your loss function)

training(....)

validate(...)

Note: Refer to lab 9 for an example of how to complete the FFN class

In [61]:
```
from torch.utils.data import Dataset

class CarTruck(Dataset):
    def __init__(self, dataset):
        self.data = []
        self.targets = []
        for img, label in dataset:
            if label == 1:  # car >> 0
                self.data.append(img)
                self.targets.append(0)
            elif label == 9:  # truck >> 1
                self.data.append(img)
                self.targets.append(1)

    def __getitem__(self, idx):
        return self.data[idx], self.targets[idx]

    def __len__(self):
        return len(self.targets)
# I end up doing this to make more reasonable results for the next steps
```

In [62]:
```
fil_train = CarTruck(cifar10)
fil_val = CarTruck(cifar10_val)
```

In [63]:
```
class FFN(nn.Module):
    def __init__(self):
        super(FFN, self).__init__()
        self.flatten = nn.Flatten()
        self.layers = nn.Sequential(nn.Linear(32 * 32 * 3, 256),
            nn.ReLU(),
            nn.Linear(256, 64),
            nn.ReLU(),
            nn.Linear(64, 1))  #the output will be like binary output (logit
    def forward(self, x):
        x = self.flatten(x)  #first convert image to 1D vector
        x = self.layers(x)
        return x
```

In [42]:
```
train_loader = DataLoader(fil_train, batch_size=32, shuffle=True)
val_loader = DataLoader(fil_val, batch_size=32)
model = FFN()
loss_fn = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

In [69]:
```
training(model, n_epochs=50, optimizer=optimizer, fn_loss=loss_fn, data_load

print("Training-------:")
train_metrics = validate(model, train_loader)
print(f"Precision: {train_metrics[0]:.2f}, Recall: {train_metrics[1]:.2f}, A

print("Validation---------:")
val_metrics = validate(model, val_loader)
print(f"Precision: {val_metrics[0]:.2f}, Recall: {val_metrics[1]:.2f}, Accur
Epoch [1/50] - Loss: 0.6135
Epoch [2/50] - Loss: 0.5198
Epoch [3/50] - Loss: 0.4398
Epoch [4/50] - Loss: 0.4038
Epoch [5/50] - Loss: 0.3797
Epoch [6/50] - Loss: 0.3596
Epoch [7/50] - Loss: 0.3415
Epoch [8/50] - Loss: 0.3312
Epoch [9/50] - Loss: 0.3152
Epoch [10/50] - Loss: 0.3058
Epoch [11/50] - Loss: 0.2905
Epoch [12/50] - Loss: 0.2819
Epoch [13/50] - Loss: 0.2794
Epoch [14/50] - Loss: 0.2674
Epoch [15/50] - Loss: 0.2610
Epoch [16/50] - Loss: 0.2536
Epoch [17/50] - Loss: 0.2487
Epoch [18/50] - Loss: 0.2454
Epoch [19/50] - Loss: 0.2384
Epoch [20/50] - Loss: 0.2328
Epoch [21/50] - Loss: 0.2278
Epoch [22/50] - Loss: 0.2229
Epoch [23/50] - Loss: 0.2188
Epoch [24/50] - Loss: 0.2169
Epoch [25/50] - Loss: 0.2140
Epoch [26/50] - Loss: 0.2086
Epoch [27/50] - Loss: 0.2052
Epoch [28/50] - Loss: 0.2077
Epoch [29/50] - Loss: 0.1995
Epoch [30/50] - Loss: 0.2044
Epoch [31/50] - Loss: 0.1951
Epoch [32/50] - Loss: 0.1952
Epoch [33/50] - Loss: 0.1948
Epoch [34/50] - Loss: 0.1883
Epoch [35/50] - Loss: 0.1884
Epoch [36/50] - Loss: 0.1845
Epoch [37/50] - Loss: 0.1816
Epoch [38/50] - Loss: 0.1790
Epoch [39/50] - Loss: 0.1806
Epoch [40/50] - Loss: 0.1795
Epoch [41/50] - Loss: 0.1788
Epoch [42/50] - Loss: 0.1726
Epoch [43/50] - Loss: 0.1716
Epoch [44/50] - Loss: 0.1721
Epoch [45/50] - Loss: 0.1627
Epoch [46/50] - Loss: 0.1658
Epoch [47/50] - Loss: 0.1653
Epoch [48/50] - Loss: 0.1646
Epoch [49/50] - Loss: 0.1605
Epoch [50/50] - Loss: 0.1602
Training-------:
Precision: 0.95, Recall: 0.94, Accuracy: 0.95, f1: 0.95
Validation---------:
Precision: 0.90, Recall: 0.90, Accuracy: 0.90, f1: 0.90
```

Question 8: Convolutional Neural Networks

Build a convolutional neural network with two blocks, where each block is a convolutional layer followed by a max pooling layer. You are expected to:

- Complete the CNN Class, based on the network architecture below (some scaffolding for this class is provided below, but feel free to write your own).
- Train / compile the network, following the guidelines below.
- Report the accuracy, precision, recall and f1-score for the training and validation sets.

Network Architecture:

- Input layer dimensions: 32 X 32 X 3 (W X H X C, where W = width, H = Height, C = Number of Channels)
- Convolutional layer1:
  - Number of filters / kernels: 20
  - Dimensions: 3 x 3
  - Activation: ReLU
- Convolutional layer 2:
  - Number of filters / kernels: 10
  - Dimensions: 3 x 3
  - Activation: ReLU

Compile:

- Optimizer: Adam
- Batch size: 32

You may run the model for a maximum of 50 epochs.

Here is an example of how to implement your neural network:

test_loader = ... (Define your data loader)

learning_rate = .. (define learning rate)
neural_network = .. (initialize your neural network)
optimizer = .. (set up your optimizer)
loss_fn = .. (set up your loss function)

training(....)

validate(...)

Note: Refer to lab 9 for an example of how to complete the CNN class

In [46]:
```
import torch.nn.functional as F
```

In [48]:
```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=20, kernel_size=3
        self.pool1 = nn.MaxPool2d(kernel_size=2)

        self.conv2 = nn.Conv2d(in_channels=20, out_channels=10, kernel_size=
        self.pool2 = nn.MaxPool2d(kernel_size=2)
        #here we run pooling two times:: so after two poolings of 32x32 >> 1

        self.fc = nn.Linear(10 * 8 * 8, 1)

    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2(F.relu(self.conv2(x)))
        x = x.view(x.size(0), -1)
        x = self.fc(x)  #we wanna have linear output
        return x
```

In [68]:
```
train_loader = DataLoader(fil_train, batch_size=32, shuffle=True)
val_loader = DataLoader(fil_val, batch_size=32)
model = CNN()
loss_fn = nn.BCEWithLogitsLoss() # no sigmoid here
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)



model.train()
inputs, labels = next(iter(train_loader))

print("Inputs shape:", inputs.shape)  #hould be [32, 3, 32, 32]
print("Labels shape:", labels.shape)  ##should be [32]

#cheching labels type (?)
print("Label dtype:", labels.dtype)  # Should be torch.int64 or torch.long

#convert labels for loss
labels = labels.float().unsqueeze(1)
print("Labels reshaped for BCE:", labels.shape, "| dtype:", labels.dtype)

#forward pass
outputs = model(inputs)
print("Outputs (logits) shape:", outputs.shape)  # Should be [32, 1]

# check Sigmoid + predictions
probs = torch.sigmoid(outputs)
preds = (probs > 0.5).int()

print("Probabilities:", probs[:5].squeeze().tolist())
print("Predictions:", preds[:5].squeeze().tolist())
print("True Labels:", labels[:5].squeeze().tolist())
```

```
loss = loss_fn(outputs, labels)
print("Loss value:", loss.item())
```

```
Inputs shape: torch.Size([32, 3, 32, 32])
Labels shape: torch.Size([32])
Label dtype: torch.int64
Labels reshaped for BCE: torch.Size([32, 1]) | dtype: torch.float32
Outputs (logits) shape: torch.Size([32, 1])
Probabilities: [0.5078256130218506, 0.5175284147262573, 0.5185617804527283,
0.5146922469139099, 0.5164749622344971]
Predictions: [1, 1, 1, 1, 1]
True Labels: [0.0, 1.0, 1.0, 0.0, 0.0]
Loss value: 0.6981905698776245
```

In [67]:
```
training(model, n_epochs=50, optimizer=optimizer, fn_loss=loss_fn, data_loac

print("\nTraining---------:")
train_metrics = validate(model, train_loader)
print(f"Precision: {train_metrics[0]:.2f}, Recall: {train_metrics[1]:.2f}, A

print("\nValidation----------------:")
val_metrics = validate(model, val_loader)
print(f"Precision: {val_metrics[0]:.2f}, Recall: {val_metrics[1]:.2f}, Accur
Epoch [1/50] - Loss: 0.6161
Epoch [2/50] - Loss: 0.5337
Epoch [3/50] - Loss: 0.4704
Epoch [4/50] - Loss: 0.4418
Epoch [5/50] - Loss: 0.4071
Epoch [6/50] - Loss: 0.3887
Epoch [7/50] - Loss: 0.3679
Epoch [8/50] - Loss: 0.3477
Epoch [9/50] - Loss: 0.3366
Epoch [10/50] - Loss: 0.3251
Epoch [11/50] - Loss: 0.3066
Epoch [12/50] - Loss: 0.2950
Epoch [13/50] - Loss: 0.2918
Epoch [14/50] - Loss: 0.2804
Epoch [15/50] - Loss: 0.2720
Epoch [16/50] - Loss: 0.2662
Epoch [17/50] - Loss: 0.2604
Epoch [18/50] - Loss: 0.2567
Epoch [19/50] - Loss: 0.2528
Epoch [20/50] - Loss: 0.2496
Epoch [21/50] - Loss: 0.2416
Epoch [22/50] - Loss: 0.2373
Epoch [23/50] - Loss: 0.2347
Epoch [24/50] - Loss: 0.2290
Epoch [25/50] - Loss: 0.2276
Epoch [26/50] - Loss: 0.2243
Epoch [27/50] - Loss: 0.2212
Epoch [28/50] - Loss: 0.2195
Epoch [29/50] - Loss: 0.2206
Epoch [30/50] - Loss: 0.2165
Epoch [31/50] - Loss: 0.2096
Epoch [32/50] - Loss: 0.2094
Epoch [33/50] - Loss: 0.2090
Epoch [34/50] - Loss: 0.2006
Epoch [35/50] - Loss: 0.2042
Epoch [36/50] - Loss: 0.2009
Epoch [37/50] - Loss: 0.1958
Epoch [38/50] - Loss: 0.1982
Epoch [39/50] - Loss: 0.1920
Epoch [40/50] - Loss: 0.1967
Epoch [41/50] - Loss: 0.1922
Epoch [42/50] - Loss: 0.1893
Epoch [43/50] - Loss: 0.1899
Epoch [44/50] - Loss: 0.1850
Epoch [45/50] - Loss: 0.1847
Epoch [46/50] - Loss: 0.1835
Epoch [47/50] - Loss: 0.1815
Epoch [48/50] - Loss: 0.1804
Epoch [49/50] - Loss: 0.1780
Epoch [50/50] - Loss: 0.1796

Training---------:
Precision: 0.92, Recall: 0.95, Accuracy: 0.93, F1: 0.93

Validation----------------:
Precision: 0.89, Recall: 0.92, Accuracy: 0.90, F1: 0.90
```

Question 9: your turn!

Now, implement NewNet -- a convolutional network that you will build from scratch. The goal should be to exceed an F1 Score of 0.85. Note: We do not expect you to implement cross-validation or any formal hyperparameter optimization techniques. Rather, the goal is to arrive at a model architecture that is acceptable to you via trial and error.

Remember that you have a number of hyperparameters to work with, including

- the number / dimension of hidden layers
- choice of activation functions,
- type regularization,
- optimization techniques
- batch normalization
- and other relevant aspects(adding data augmentation etc.)

You may run the model for a maximum of 50 epochs.

Outputs:

- Complete the NewNet class below with your model
- Report the accuracy, recall, precision and f1-score on the training and validation sets.
- Briefly explain your model architecture / choices you made in tuning your CNN (No more than 3 - 4 sentences)
- Compare your model performance (precision, recall, and overall classification accuracy), in comparison to the feed forward neural networks in Questions 7 and 8?

In [86]:
```python
### Helper code, modify as necessary
class NewNet(nn.Module):
    def __init__(self):
        super(NewNet, self).__init__()

        self.block1 = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Dropout(0.2))

        self.block2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Dropout(0.3))

        self.block3 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Dropout(0.4))

        self.pool = nn.AdaptiveAvgPool2d((1, 1))  # the size after this will
        #print()
        self.classifier = nn.Sequential(nn.Flatten(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(64, 1))

    def forward(self, x):
        x = self.block1(x)
        x = self.block2(x)
        x = self.block3(x)
        x = self.pool(x)
        x = self.classifier(x)
        return x
```

In [87]:
```
model = NewNet()
loss_fn = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

train_loader = DataLoader(fil_train, batch_size=32, shuffle=True)
val_loader = DataLoader(fil_val, batch_size=32)
```

In [88]:
```
training(model, n_epochs=50, optimizer=optimizer, fn_loss=loss_fn, data_load

print("\nTraining ---------:")
train_metrics = validate(model, train_loader)
print(f"Precision: {train_metrics[0]:.2f}, Recall: {train_metrics[1]:.2f}, A

print("\nValidation -------------------:")
val_metrics = validate(model, val_loader)
print(f"Precision: {val_metrics[0]:.2f}, Recall: {val_metrics[1]:.2f}, Accur
Epoch [1/50] - Loss: 0.5827
Epoch [2/50] - Loss: 0.4668
Epoch [3/50] - Loss: 0.4075
Epoch [4/50] - Loss: 0.3695
Epoch [5/50] - Loss: 0.3465
Epoch [6/50] - Loss: 0.3236
Epoch [7/50] - Loss: 0.3098
Epoch [8/50] - Loss: 0.2962
Epoch [9/50] - Loss: 0.2892
Epoch [10/50] - Loss: 0.2665
Epoch [11/50] - Loss: 0.2538
Epoch [12/50] - Loss: 0.2499
Epoch [13/50] - Loss: 0.2402
Epoch [14/50] - Loss: 0.2383
Epoch [15/50] - Loss: 0.2245
Epoch [16/50] - Loss: 0.2194
Epoch [17/50] - Loss: 0.2099
Epoch [18/50] - Loss: 0.2143
Epoch [19/50] - Loss: 0.2119
Epoch [20/50] - Loss: 0.2016
Epoch [21/50] - Loss: 0.1915
Epoch [22/50] - Loss: 0.1855
Epoch [23/50] - Loss: 0.1913
Epoch [24/50] - Loss: 0.1796
Epoch [25/50] - Loss: 0.1835
Epoch [26/50] - Loss: 0.1718
Epoch [27/50] - Loss: 0.1729
Epoch [28/50] - Loss: 0.1746
Epoch [29/50] - Loss: 0.1651
Epoch [30/50] - Loss: 0.1627
Epoch [31/50] - Loss: 0.1532
Epoch [32/50] - Loss: 0.1593
Epoch [33/50] - Loss: 0.1587
Epoch [34/50] - Loss: 0.1534
Epoch [35/50] - Loss: 0.1499
Epoch [36/50] - Loss: 0.1393
Epoch [37/50] - Loss: 0.1498
Epoch [38/50] - Loss: 0.1443
Epoch [39/50] - Loss: 0.1382
Epoch [40/50] - Loss: 0.1453
Epoch [41/50] - Loss: 0.1398
Epoch [42/50] - Loss: 0.1471
Epoch [43/50] - Loss: 0.1293
Epoch [44/50] - Loss: 0.1322
Epoch [45/50] - Loss: 0.1226
Epoch [46/50] - Loss: 0.1278
Epoch [47/50] - Loss: 0.1259
Epoch [48/50] - Loss: 0.1315
Epoch [49/50] - Loss: 0.1221
Epoch [50/50] - Loss: 0.1152

Training ---------:
Precision: 0.97, Recall: 1.00, Accuracy: 0.98, F1: 0.98

Validation -------------------:
Precision: 0.90, Recall: 0.96, Accuracy: 0.93, F1: 0.93
```

In [89]:
```
torch.save(model.state_dict(), "newNet_q9_weights.pth")
```

Question 10: Feature Maps

A feature map, or an activation map allows us to examine the result of applying the filters to a given input. The broad intuition is that feature maps closer to the input image detect fine-grained detail, whereas feature maps closer to the output of the model capture more generic aspects.

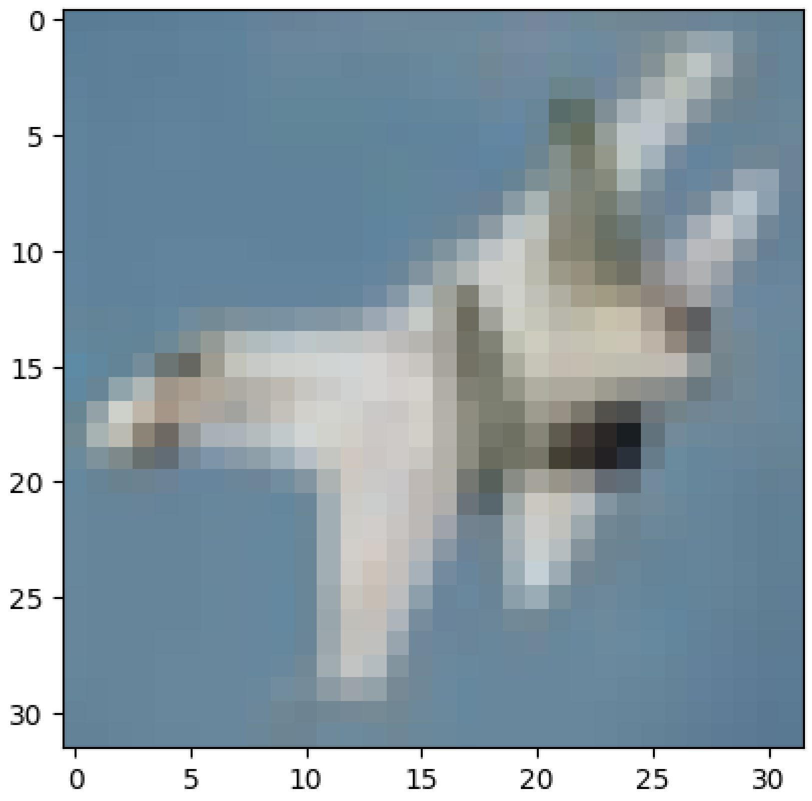
Your task is to create and visualize a feature map (i.e the outputs) from the first convolutional layer in your trained CNN.

In order to do this, proceed as follows:

- Identify a nice image from your training data -- ideally, something that has some distinguishing properties to the naked eye.
- Pass this image through your trained CNN from **Question 9**, and store the output from the first convolutional layer -- this is your feature map! Note that there are multiple ways to do this -- this tutorial is particularly helpful.

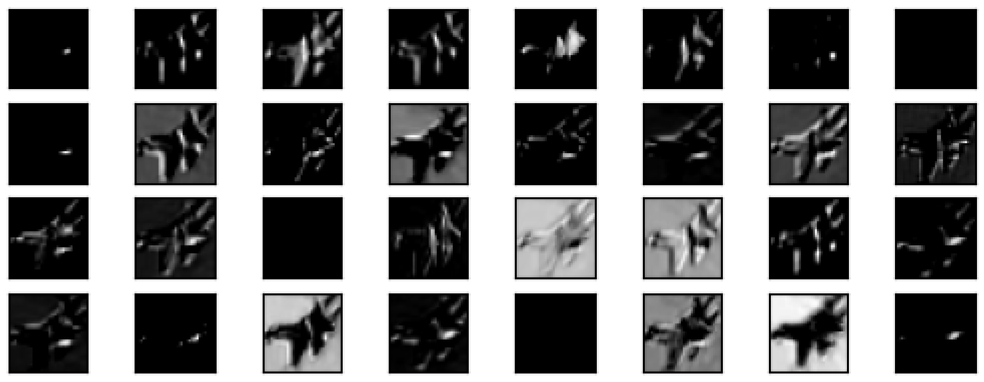
Note that the size of the feature map depends on how many filters you have in the layer.

Outputs: - plot 1) The raw image from the training data, and 2) the feature map. An example is shown below: - what do you observe about the feature maps?



Raw Image

Feature Map



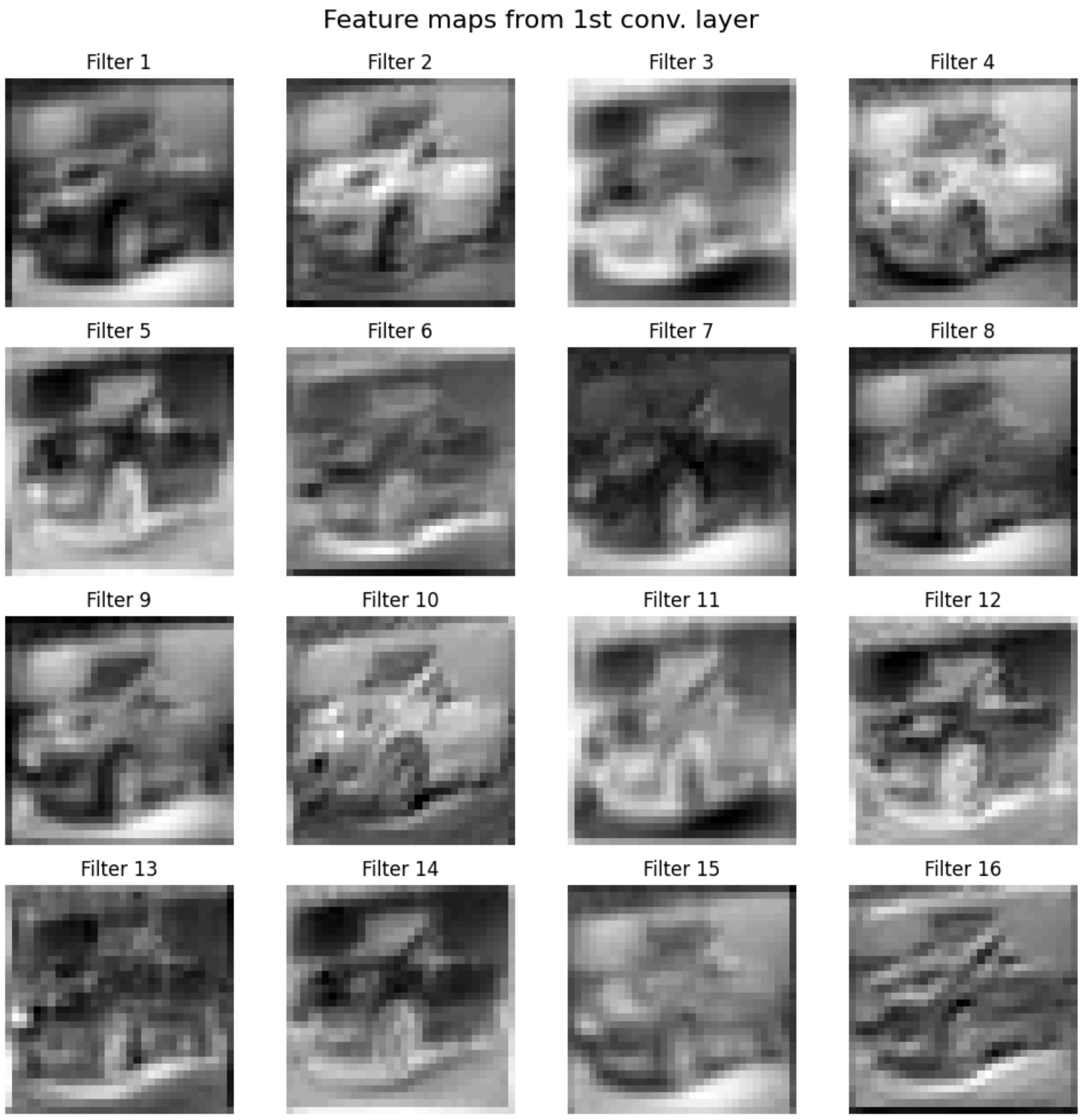
```
In [76]: sample_img, label = fil_train[10]
         sample_img_batch = sample_img.unsqueeze(0)
```

```
In [77]: def hook_fn(module, input, output):
             activation['conv1'] = output
```

In [78]:
```
activation = {}
hook = model.block1[0].register_forward_hook(hook_fn)

model.eval()
with torch.no_grad():
    _ = model(sample_img_batch)

hook.remove()
feature_map = activation['conv1'].squeeze(0)
```

In [79]:
```
plt.figure(figsize=(4, 4))
plt.imshow(sample_img.permute(1, 2, 0))  # convert to HWC for plotting
plt.title("Original Image")
plt.axis('off')
plt.show()
fig, axs = plt.subplots(4, 4, figsize=(10, 10))

for i in range(16):
    axs[i//4, i%4].imshow(feature_map[i].detach().numpy(), cmap='gray')
    axs[i//4, i%4].axis('off')
    axs[i//4, i%4].set_title(f'Filter {i+1}')

plt.tight_layout()
plt.suptitle("Feature maps from 1st conv. layer", fontsize=16)
plt.subplots_adjust(top=0.92)
plt.show()
```

Original Image



III. Transfer Learning / Fine tuning

Suppose you have a slightly different classification task at hand: to correctly separate trucks from airplanes.

We'll examine how we can use an already trained model to do this, instead of coding up a new neural network from scratch.

You are required to implement two approaches -- the first simply updates the weights for the model you have trained in Question 9, while the second involves loading a larger pre-trained model (e.g. ResNet)

Warning! Note that the second approach could be slow / time-consuming. If you are attempting it, please ensure that you budget ~20 mins to 1hour (worst case) for the code to complete running for this part.

Question 11: Planes and Trucks

Create a subset of the CIFAR 10 data, keeping only the airplane and truck classes.

In [80]:
```
#do the same thing we had before but this time I used the label for airplane
class AirplaneTruck(Dataset):
    def __init__(self, original_dataset):
        self.data = []
        self.targets = []
        for img, label in original_dataset:
            if label == 0:  ###airplane
                self.data.append(img)
                self.targets.append(0)
            elif label == 9:  #### truck
                self.data.append(img)
                self.targets.append(1)

    def __getitem__(self, idx):
        return self.data[idx], self.targets[idx]

    def __len__(self):
        return len(self.targets)
```

In [ ]:
```
# cifar10, cifar10_val = load_cifar10(datapath)
```

In [82]:
```
airtruck_train = AirplaneTruck(cifar10)
airtruck_val = AirplaneTruck(cifar10_val)
```

In [83]:
```
model = NewNet()
loss_fn = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

train_loader = DataLoader(airtruck_train, batch_size=32, shuffle=True)
val_loader = DataLoader(airtruck_val, batch_size=32)
```

In [84]:
```
training(model, n_epochs=20, optimizer=optimizer, fn_loss=loss_fn, data_loac
Epoch [1/20] - Loss: 0.3873
Epoch [2/20] - Loss: 0.2928
Epoch [3/20] - Loss: 0.2670
Epoch [4/20] - Loss: 0.2402
Epoch [5/20] - Loss: 0.2299
Epoch [6/20] - Loss: 0.2104
Epoch [7/20] - Loss: 0.2037
Epoch [8/20] - Loss: 0.1956
Epoch [9/20] - Loss: 0.1914
Epoch [10/20] - Loss: 0.1825
Epoch [11/20] - Loss: 0.1824
Epoch [12/20] - Loss: 0.1663
Epoch [13/20] - Loss: 0.1652
Epoch [14/20] - Loss: 0.1634
Epoch [15/20] - Loss: 0.1621
Epoch [16/20] - Loss: 0.1470
Epoch [17/20] - Loss: 0.1492
Epoch [18/20] - Loss: 0.1499
Epoch [19/20] - Loss: 0.1427
Epoch [20/20] - Loss: 0.1366
```

In [85]:
```
print("Evaluation----------------------")
validate(model, val_loader)
```

```
Evaluation----------------------
```

Out[85]:
```
(np.float64(0.8859964093357271),
 np.float64(0.987),
 0.93,
 np.float64(0.9337748344370861))
```

Question 12: Update Weights

First, use your model from Question 9 above -- and update the weights. To be precise:

- Save the model weights from Question 9.
- Then, create a new instance of the model from Question 9.
- Load the saved weights into the new model (Hint: the load_state_dict function is your friend here)
- Train this model for 10 epochs.
- Report the accuracy, recall, precision and f1-score on the training and validation sets.

In [90]:
```
model_q12 = NewNet()
model_q12.load_state_dict(torch.load("newNet_q9_weights.pth"))
```

```
/tmp/ipykernel_572816/117006147.py:2: FutureWarning: You are using `torch.lo
ad` with `weights_only=False` (the current default value), which uses the de
fault pickle module implicitly. It is possible to construct malicious pickle
data which will execute arbitrary code during unpickling (See https://githu
b.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more detail
s). In a future release, the default value for `weights_only` will be flippe
d to `True`. This limits the functions that could be executed during unpickl
ing. Arbitrary objects will no longer be allowed to be loaded via this mode
unless they are explicitly allowlisted by the user via `torch.serialization.
add_safe_globals`. We recommend you start setting `weights_only=True` for an
y use case where you don't have full control of the loaded file. Please open
an issue on GitHub for any issues related to this experimental feature.
  model_q12.load_state_dict(torch.load("newNet_q9_weights.pth"))
```

Out[90]:
```
<All keys matched successfully>
```

In [91]:
```
loss_fn = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model_q12.parameters(), lr=0.001)

train_loader = DataLoader(airtruck_train, batch_size=32, shuffle=True)
val_loader = DataLoader(airtruck_val, batch_size=32)
```

In [92]:
```
training(model_q12, n_epochs=10, optimizer=optimizer, fn_loss=loss_fn, data_
```

```
Epoch [1/10] - Loss: 0.2259
Epoch [2/10] - Loss: 0.1848
Epoch [3/10] - Loss: 0.1686
Epoch [4/10] - Loss: 0.1608
Epoch [5/10] - Loss: 0.1531
Epoch [6/10] - Loss: 0.1408
Epoch [7/10] - Loss: 0.1361
Epoch [8/10] - Loss: 0.1292
Epoch [9/10] - Loss: 0.1238
Epoch [10/10] - Loss: 0.1213
```

In [93]:
```
train_metrics = validate(model_q12, train_loader)
val_metrics = validate(model_q12, val_loader)
```

In [95]:
```
print("\nTraining ------------:")
print(f"Precision: {train_metrics[0]:.2f}, Recall: {train_metrics[1]:.2f}, A

print("\nValidation-------------------------:")
print(f"Precision: {val_metrics[0]:.2f}, Recall: {val_metrics[1]:.2f}, Accur
```

```
Training ------------:
Precision: 0.96, Recall: 0.99, Accuracy: 0.97, F1: 0.97

Validation-------------------------:
Precision: 0.92, Recall: 0.96, Accuracy: 0.94, F1: 0.94
```

Question 13: Fine Tuning [Extra Credit]

Second, we'll load a pre-trained model (e.g. ResNet50, or VGG19). While these models haven't seen the exact images in this dataset, they have been trained on a large general corpus. Since these models have millions of weights, so we'll implement the following approach:

- Load the pretrained model (pick any from several available models)
- Freeze the weights for all the layers (in pytorch, you can do this using model_object.parameters())
- Replace the final fully connected layer, since you have a binary classification task. You can add additional fully connected layers if you wish, alongwith related activation functions etc.
- Finally, train this new model for 10 epochs.
- Report the accuracy, recall, precision and f1-score on the training and validation sets.

In [97]:
```
from torchvision import models, transforms
```

In [98]:
```
resnet_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor()])
```

In [99]:
```
#loading with transforms##
cifar10 = datasets.CIFAR10(datapath, train=True, download=True, transform=re
cifar10_val = datasets.CIFAR10(datapath, train=False, download=True, transfo


airtruck_train = AirplaneTruck(cifar10)
airtruck_val = AirplaneTruck(cifar10_val)

train_loader = DataLoader(airtruck_train, batch_size=32, shuffle=True)
val_loader = DataLoader(airtruck_val, batch_size=32)
```

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

In [100…
```
resnet = models.resnet50(pretrained=True)
```

```
/new/benpyenv/lib/python3.10/site-packages/torchvision/models/_utils.py:208:
UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be
removed in the future, please use 'weights' instead.
  warnings.warn(
/new/benpyenv/lib/python3.10/site-packages/torchvision/models/_utils.py:223:
UserWarning: Arguments other than a weight enum or `None` for 'weights' are
deprecated since 0.13 and may be removed in the future. The current behavior
is equivalent to passing `weights=ResNet50_Weights.IMAGENET1K_V1`. You can a
lso use `weights=ResNet50_Weights.DEFAULT` to get the most up-to-date weight
s.
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/resnet50-0676ba61.pth" to
/home/javad/.cache/torch/hub/checkpoints/resnet50-0676ba61.pth
100%|██████████████████████████████████████| 97.8M/97.8M [00:23<00:00, 4.37M
B/s]
```

In [101…
```
for param in resnet.parameters():
    param.requires_grad = False
```

In [103…

```
resnet.fc = nn.Sequential(nn.Linear(resnet.fc.in_features, 128),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(128, 1))
```

In [104…

```
loss_fn = nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(resnet.fc.parameters(), lr=0.001) #training is
```

In [105…

```
training(resnet, n_epochs=10, optimizer=optimizer, fn_loss=loss_fn, data_loa
```

```
Epoch [1/10] - Loss: 0.2221
Epoch [2/10] - Loss: 0.1690
Epoch [3/10] - Loss: 0.1474
Epoch [4/10] - Loss: 0.1474
Epoch [5/10] - Loss: 0.1354
Epoch [6/10] - Loss: 0.1396
Epoch [7/10] - Loss: 0.1326
Epoch [8/10] - Loss: 0.1380
Epoch [9/10] - Loss: 0.1346
Epoch [10/10] - Loss: 0.1265
```

In [106…

```
train_metrics = validate(resnet, train_loader)
val_metrics = validate(resnet, val_loader)
```

In [110…

```
print("\nTraining ----------------")
print(f"Precision: {train_metrics[0]:.2f}, Recall: {train_metrics[1]:.2f}, A

print("\nValidation -----------------:")
print(f"Precision: {val_metrics[0]:.2f}, Recall: {val_metrics[1]:.2f}, Accur
```

```
Training ----------------
Precision: 0.98, Recall: 0.96, Accuracy: 0.97, F1: 0.97

Validation -----------------:
Precision: 0.97, Recall: 0.94, Accuracy: 0.95, F1: 0.95
```

Question 14: Conclusion [Extra Credit]

Compare model performance from Questions 12 and 13, and briefly summarize your observations.