In [1]:

```
import torch
import torchvision
import torchvision.transforms as transforms
from torch import nn
from tqdm import tqdm
import matplotlib.pyplot as plt
import numpy as np
import multiprocessing

transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.4914, 0.4822, 0.4465),  (0.2023, 0.1994, 0.2010))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128,
                                          shuffle=True, num_workers=multiprocessing

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=128,
                                         shuffle=False, num_workers=multiprocessing

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

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

In [2]:

```
if torch.cuda.is_available():
    print("Training on gpu")
    mode = 'cuda'
else:
    print("Training on cpu")
    mode = 'cpu'
```

```
Training on gpu
```

Exercise 1. Convolutional Neural Networks (CNN)

In [3]:

```
class BasicBlock(nn.Module):
    def __init__(self, in_channels, out_channels, is_last):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=(3, 3), paddi
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=(3, 3), padd
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.conv3 = nn.Conv2d(out_channels, out_channels, kernel_size=(3, 3), padd
        self.bn3 = nn.BatchNorm2d(out_channels)
        if is_last:
            self.conv4 = nn.Conv2d(out_channels, out_channels, kernel_size=(3, 3),
        else:
            self.conv4 = nn.Conv2d(out_channels, out_channels, kernel_size=(3, 3),
        self.bn4 = nn.BatchNorm2d(out_channels)

        self.in_channels = in_channels
        self.out_channels = out_channels

        if in_channels != out_channels:
            self.upsample_filters = True
            self.transform_skip_conv = nn.Conv2d(in_channels, out_channels, kernel_
            self.transform_skip_bn   = nn.BatchNorm2d(out_channels)
        else:
            self.upsample_filters = False
        if is_last:
            self.transform_skip_conv = nn.Conv2d(out_channels, out_channels, kernel
            self.transform_skip_bn = nn.BatchNorm2d(out_channels)
        self.is_last = is_last
        self.mse_loss = torch.nn.MSELoss()

    def forward(self, x, show_mse=False):
        x_skip = x
        x = self.conv1(x)
        x = torch.nn.functional.relu(x)
        x = self.bn1(x)
        x = self.conv2(x)

        if self.upsample_filters:
            x_skip = self.transform_skip_conv(x_skip)
            x_skip = torch.nn.functional.relu(x_skip)
            x_skip = self.transform_skip_bn(x_skip)

        x = x + x_skip
        if show_mse:
            loss = self.mse_loss(x, x_skip)
            print(f"MSE IS: {loss}")
        x = torch.nn.functional.relu(x)
        x = self.bn2(x)
        x_skip = x

        x = self.conv3(x)
        x = torch.nn.functional.relu(x)
        x = self.bn3(x)
        x = self.conv4(x)

        if self.is_last:
            x_skip = self.transform_skip_conv(x_skip)
            x_skip = torch.nn.functional.relu(x_skip)
            x_skip = self.transform_skip_bn(x_skip)
```

```
        x = x + x_skip
        if show_mse:
            loss = self.mse_loss(x, x_skip)
            print(f"MSE IS: {loss}")
        x = torch.nn.functional.relu(x)
        x = self.bn4(x)
        return x



class ResNetBlock(nn.Module):
    def __init__(self, in_channels, out_channels, n):
        super().__init__()
        layers = []
        layers.append(BasicBlock(in_channels, out_channels, is_last=False))
        for j in range(2 * n - 2):
            layers.append(BasicBlock(out_channels, out_channels, is_last=False))
        layers.append(BasicBlock(out_channels, out_channels, is_last=True))
        self.layers = nn.ModuleList(layers)

    def forward(self, x, show_mse):
        for layer in self.layers:
            x = layer(x, show_mse=show_mse)
        return x

class ResNet(nn.Module):
    def __init__(self, in_channels, n):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels, 16, kernel_size=(3, 3), padding=(1, 1),
        self.bn1 = nn.BatchNorm2d(16)
        self.fc = nn.Linear(64, 10)
        self.rn_block1 = ResNetBlock(16, 16 * (2 ** 0), n)
        self.rn_block2 = ResNetBlock(16 * (2 ** 0), 16 * (2 ** 1), n)
        self.rn_block3 = ResNetBlock(16 * (2 ** 1), 16 * (2 ** 2), n)
        self.global_avgpool = nn.AdaptiveAvgPool2d((1, 1))

    def forward(self, x, is_training=False, show_mse=False):
        x = self.conv1(x)
        x = torch.nn.functional.relu(x)
        x = self.bn1(x)
        x = self.rn_block1(x, show_mse=show_mse)
        x = self.rn_block2(x, show_mse=show_mse)
        x = self.rn_block3(x, show_mse=show_mse)
        x = self.global_avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)
        if not is_training:
            # CategoricalCrossentropy in pytorch already applies softmnax
            x = torch.nn.functional.softmax(x, dim=1)
        return x
```

In [4]:

```
def train_loop(model, optimizer, criterion, epochs, trainloader, testloader):
    train_losses = []
    val_losses = []
    train_accs = []
    val_accs = []
    for epoch in range(1, epochs + 1):
        # epoch
        total_train_loss = 0
        total_train_acc = 0
        train_iterations = 0
        for x, y in tqdm(trainloader):
            optimizer.zero_grad()
            x, y = x.to(mode), y.to(mode)
            y_pred = clf(x, is_training=True)
            loss = criterion(y_pred, y)
            loss.backward()
            optimizer.step()

            total_train_loss += loss.item()
            acc = (torch.argmax(y_pred, dim=1) == y).sum().item() / y.shape[0]
            total_train_acc += acc
            train_iterations += 1

        total_val_loss = 0
        total_val_acc = 0
        test_iterations = 0
        for x, y in tqdm(testloader):
            x, y = x.to(mode), y.to(mode)
            with torch.no_grad():
                y_pred = clf(x)
                acc = (torch.argmax(y_pred, dim=1) == y).sum().item() / y.shape[0]
                loss = criterion(y_pred, y)
                total_val_acc += acc
                total_val_loss += loss.item()
            test_iterations += 1


        train_losses.append(total_train_loss / train_iterations)
        train_accs.append(total_train_acc / train_iterations)
        val_losses.append(total_val_loss / test_iterations)
        val_accs.append(total_val_acc / test_iterations)

        print(f"train loss at epoch {epoch}: {train_losses[-1]}")
        print(f"val loss at epoch {epoch}: {val_losses[-1]}")
        print(f"train acc at epoch {epoch}: {train_accs[-1]}")
        print(f"val acc at epoch {epoch}: {val_accs[-1]}")

    return train_losses, val_losses, train_accs, val_accs
```

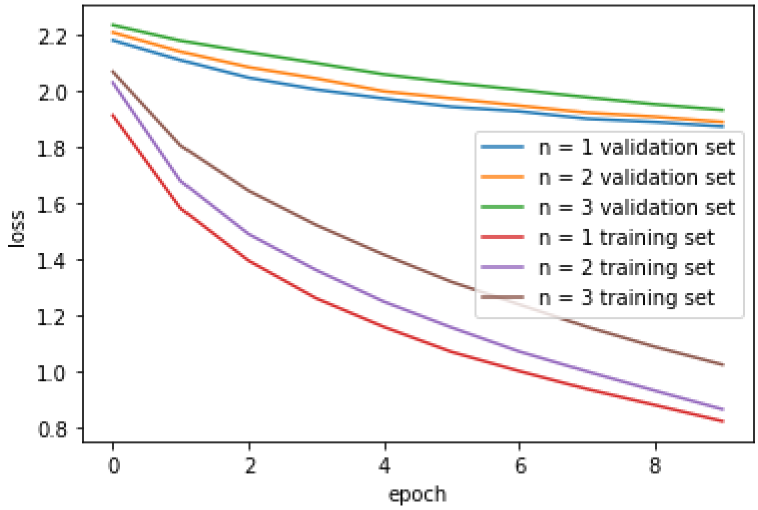
In [5]:

```
metrics = {}
for n in range(1, 4):
    clf = ResNet(in_channels=3, n=n).to(mode)
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(clf.parameters(), lr=0.002, momentum=0.9)
    train_losses, val_losses, train_accs, val_accs = train_loop(
        clf,
        optimizer,
        criterion,
        10,
        trainloader,
        testloader
    )
    metrics[n] = {
        "train_losses": train_losses,
        "val_losses": val_losses,
        "train_accs": train_accs,
        "val_accs": val_accs
    }
```

```
train acc at epoch 8: 0.5798853120204604
val acc at epoch 8: 0.5709058544303798

100%|██████████| 391/391 [00:35<00:00, 11.15it/s]
100%|██████████| 79/79 [00:02<00:00, 31.13it/s]
  0%|          | 0/391 [00:00<?, ?it/s]

train loss at epoch 9: 1.0889641971844237
val loss at epoch 9: 1.9504002589213698
train acc at epoch 9: 0.6082201086956522
val acc at epoch 9: 0.5877175632911392

100%|██████████| 391/391 [00:34<00:00, 11.17it/s]
100%|██████████| 79/79 [00:02<00:00, 31.50it/s]

train loss at epoch 10: 1.0255918079020117
val loss at epoch 10: 1.9302597423143024
train acc at epoch 10: 0.6292039641943734
val acc at epoch 10: 0.6092761075949367
```

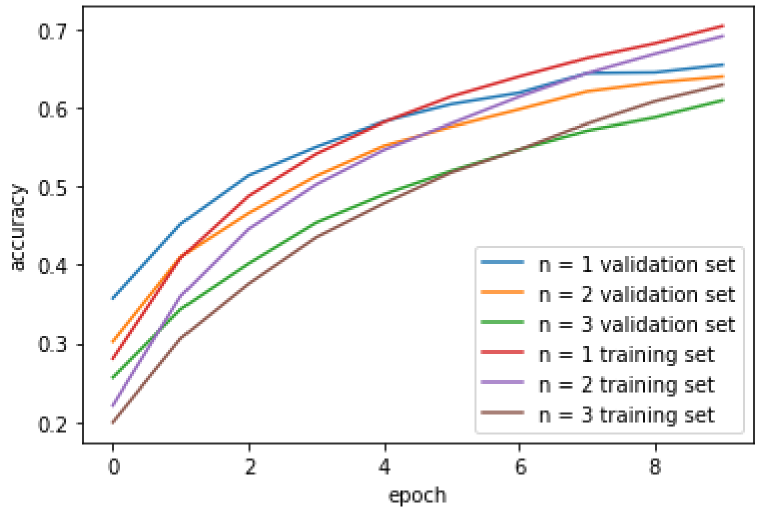
In [6]:

```
plt.plot(list(range(len(metrics[1]["val_losses"]))), metrics[1]["val_losses"], labe
plt.plot(list(range(len(metrics[2]["val_losses"]))), metrics[2]["val_losses"], labe
plt.plot(list(range(len(metrics[3]["val_losses"]))), metrics[3]["val_losses"], labe
plt.plot(list(range(len(metrics[1]["train_losses"]))), metrics[1]["train_losses"],
plt.plot(list(range(len(metrics[2]["train_losses"]))), metrics[2]["train_losses"],
plt.plot(list(range(len(metrics[3]["train_losses"]))), metrics[3]["train_losses"],
plt.xlabel("epoch")
plt.ylabel("loss")
plt.legend()
plt.show()
```



In [7]:

```
plt.plot(list(range(len(metrics[1]["val_accs"]))), metrics[1]["val_accs"], label="n
plt.plot(list(range(len(metrics[1]["val_accs"]))), metrics[2]["val_accs"], label="n
plt.plot(list(range(len(metrics[1]["val_accs"]))), metrics[3]["val_accs"], label="n
plt.plot(list(range(len(metrics[1]["train_accs"]))), metrics[1]["train_accs"], labe
plt.plot(list(range(len(metrics[1]["train_accs"]))), metrics[2]["train_accs"], labe
plt.plot(list(range(len(metrics[1]["train_accs"]))), metrics[3]["train_accs"], labe
plt.xlabel("epoch")
plt.ylabel("accuracy")
plt.legend()
plt.show()
```



(a) Plot the filters of the first layer. What kind of features do they

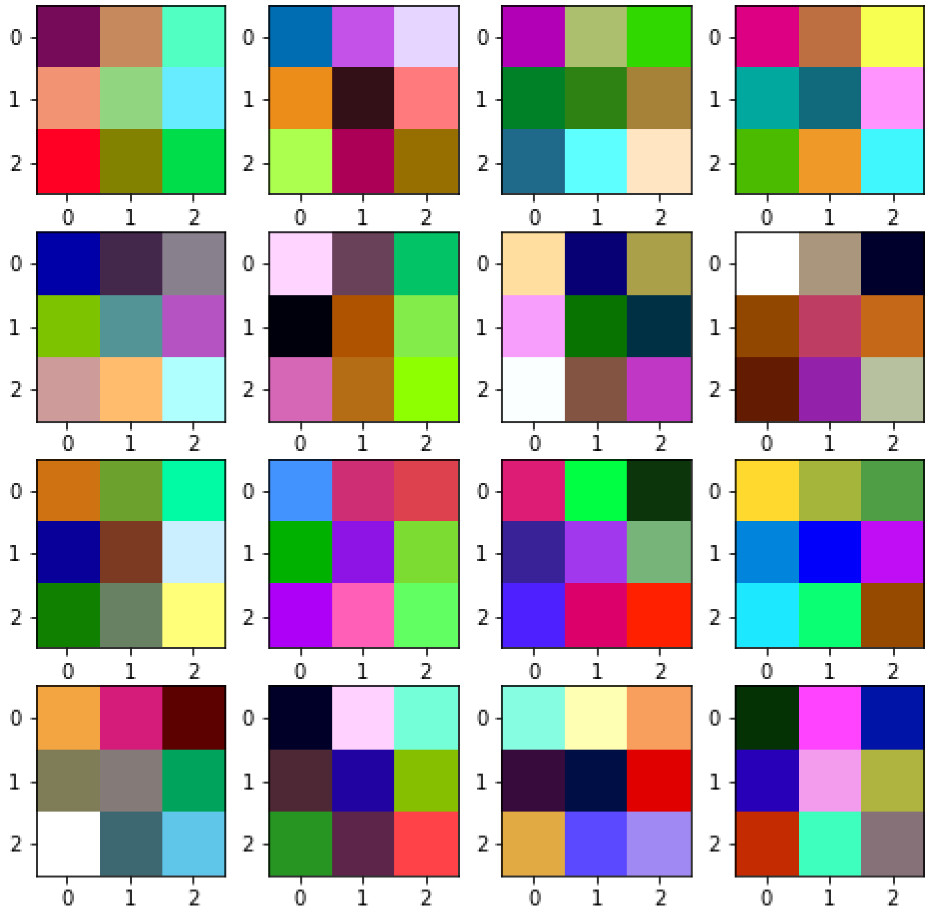
extract?

In [8]:

```
def normalise(x):
    minimum = np.min(x)
    maximum = np.max(x)
    return (x - minimum) / (maximum - minimum)

def normalise_img(img):
    img[:,:,0] = normalise(img[:,:,0])
    img[:,:,1] = normalise(img[:,:,1])
    img[:,:,2] = normalise(img[:,:,2])
    return img
```

In [9]:

```
filters = clf.conv1.weight.detach().cpu().clone().numpy()

w=10
h=10
fig=plt.figure(figsize=(8, 8))
columns = 4
rows = 4
for i in range(1, columns*rows +1):
    img = normalise_img(filters[i - 1])
    fig.add_subplot(rows, columns, i)
    plt.imshow(img)
plt.show()
```



(b) For every two convolutions with skip connection calculate the

MSE of the input of thoselayerxinand the outputxout: MSE(xin, xout). Does your network have layers that were learned to be the identity?

In [10]:

```
for x, y in trainloader:
    x = x.to(mode)
    clf(x, show_mse=True)
    break
```

```
MSE IS: 0.6238371133804321
MSE IS: 0.5665003061294556
MSE IS: 0.5593308210372925
MSE IS: 0.40274566411972046
MSE IS: 0.5262214541435242
MSE IS: 0.5639868974685669
MSE IS: 0.4649959206581116
MSE IS: 0.5885398387908936
MSE IS: 0.4307798147201538
MSE IS: 0.5467429161071777
MSE IS: 0.6793290376663208
MSE IS: 0.6809646487236023
MSE IS: 0.4855375289916992
MSE IS: 0.4010744094848633
MSE IS: 0.4426094889640808
MSE IS: 0.39885276556015015
MSE IS: 0.4395354390144348
MSE IS: 0.43303707242012024
MSE IS: 0.5229395627975464
MSE IS: 0.5231478810310364
MSE IS: 0.47984281182289124
MSE IS: 0.46972355246543884
MSE IS: 0.6970547437667847
MSE IS: 0.7026304006576538
MSE IS: 0.4426412582397461
MSE IS: 0.4262027144432068
MSE IS: 0.4286178648471832
MSE IS: 0.4170571565628052
MSE IS: 0.38850677013397217
MSE IS: 0.43397799134254456
MSE IS: 0.41576969623565674
MSE IS: 0.4961211383342743
MSE IS: 0.46461477875709534
MSE IS: 0.46489816904067993
MSE IS: 0.704309344291687
MSE IS: 1.0863127708435059
```

Our Network doesn't have layers which learnt to be the identity

c) Is deeper always better? Provide some evidence for your answer and explain why that is the case.

In theory it should be better, since our network can learn more complex functions, however in practise it makes the network overfit faster, makes the network suffer from the vanishing gradient problem and increases computational complexity. We can see that our network actually performed worse after 10 epochs when it had more layers.