GMDL HW4

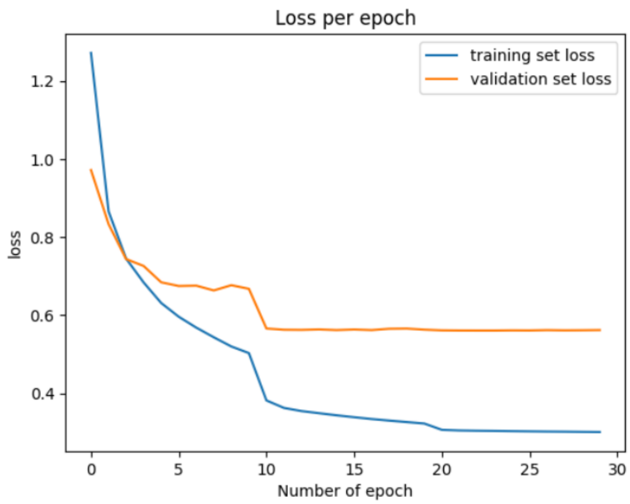
Yair Gross, Neta Elmaliach, Sharon Hendy

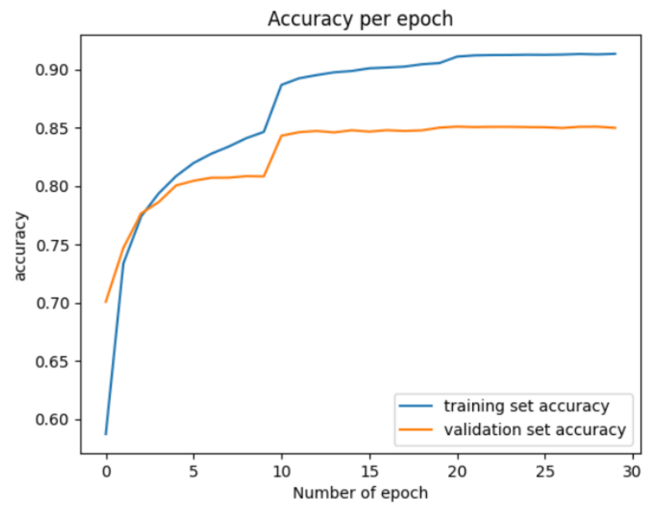
22/06/2023

4 Training

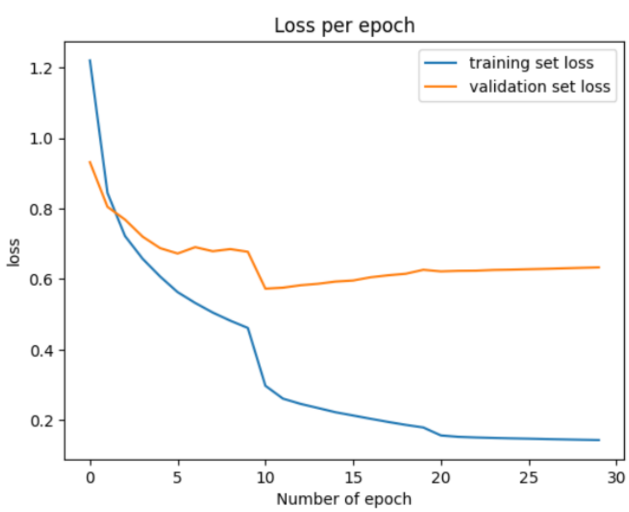
Problem 1:

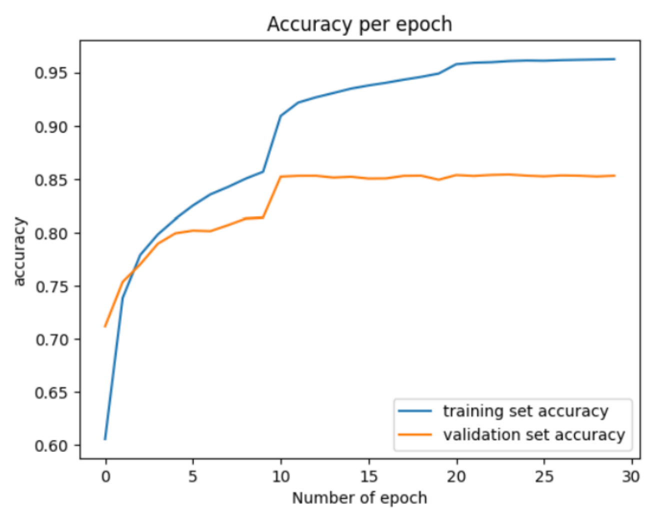
Loss and accuracy for the first FC network
layers of sizes: 128, 64



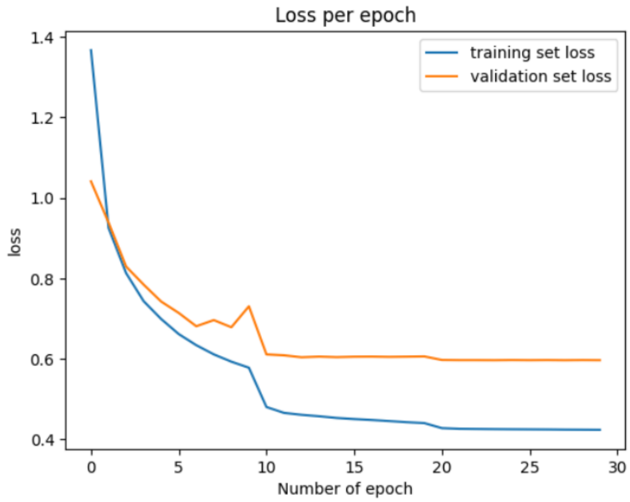


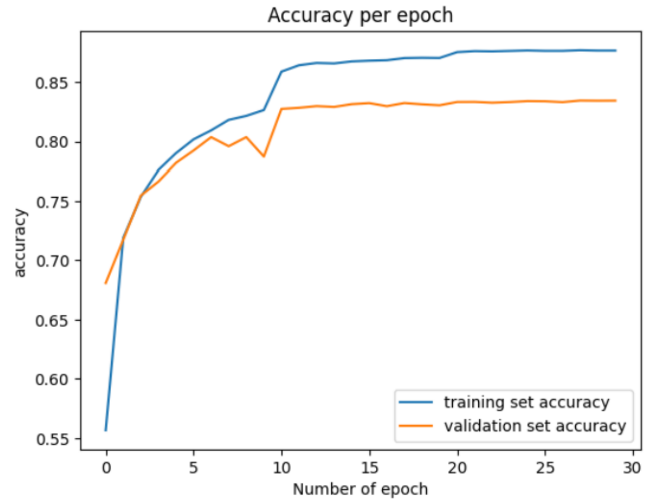
Loss and accuracy for the second FC network:
layers of sizes: 512, 256





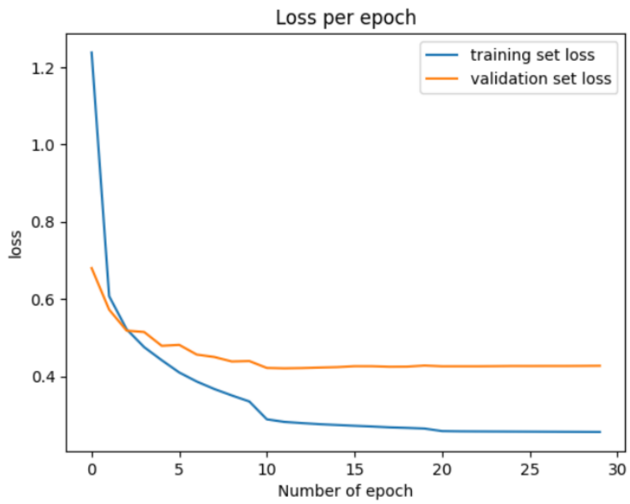
Loss and accuracy for the third FC network:
layers of sizes: 64, 32

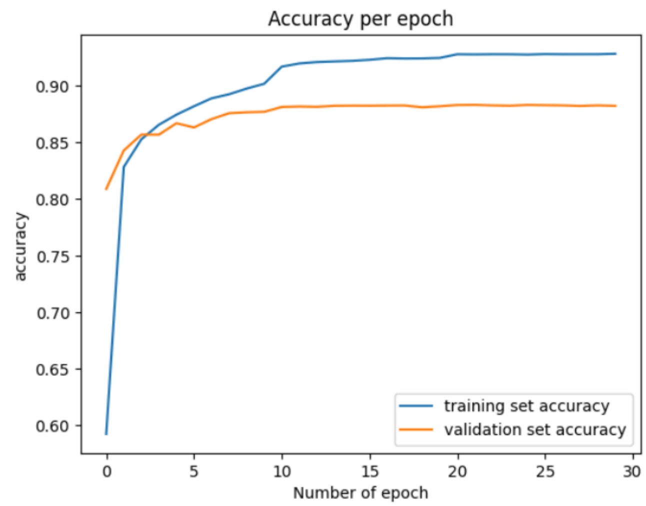




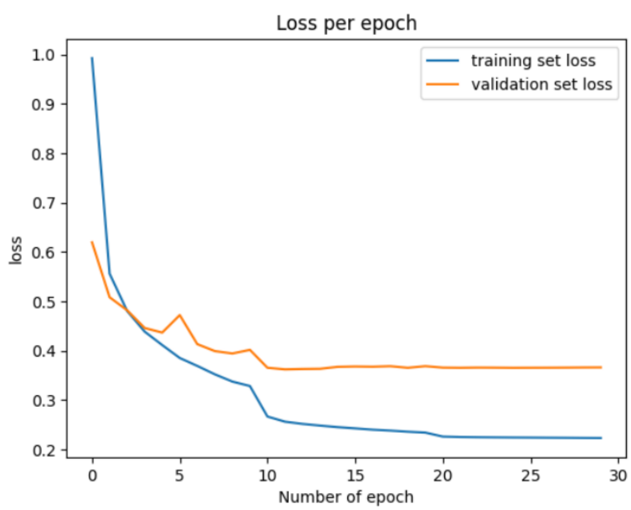
The first and the second architectures performed well on the validation set, according to the increased number of neurons in each layer.
This means that the models can classify new examples that are not from the training data with better accuracy.
The accuracy results are very high so there is no underfitiing.
The third architecture shows more overfitting due to its relatively fewer neurons in each layer.

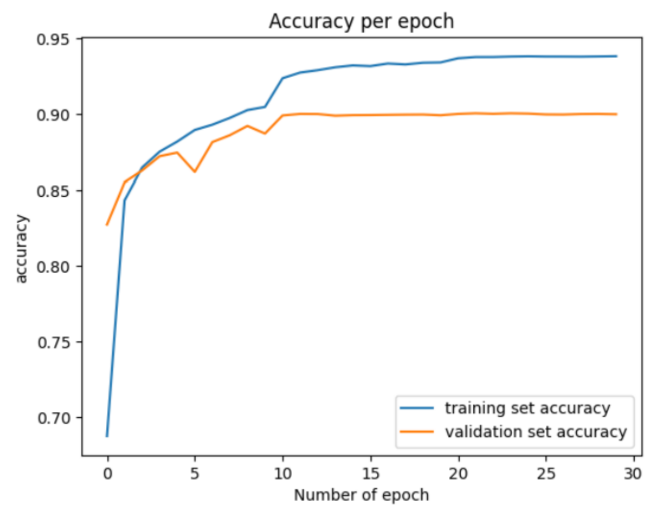
Loss and accuracy for the first CNN:
kernel: 3



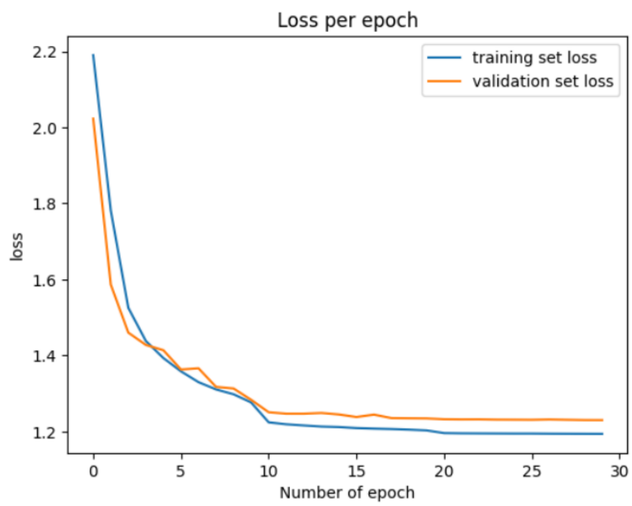


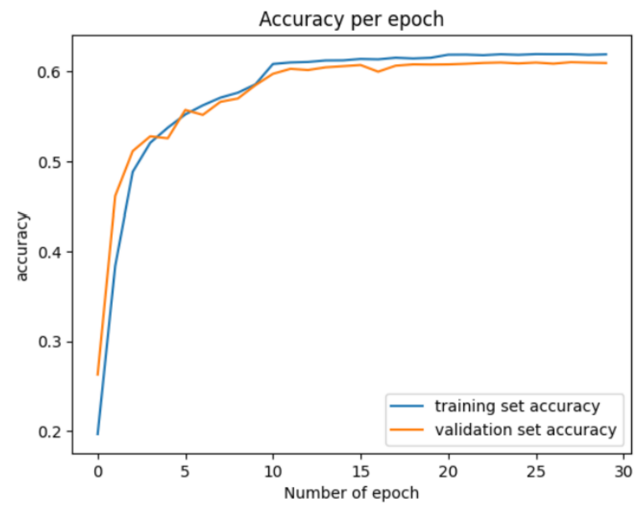
Loss and accuracy for the second CNN:
kernel: 10





Loss and accuracy for the third CNN:
kernel: 1





The net with a kernel size of 3 and the net with a kernel size 10 performed better than the network with a kernel size of 1.
The larger kernel size of 3 helped prevent overfitting and allowed the net to generalize better by capturing more complex patterns in the images.
This worked better for this specific dataset,
where there weren't significant variations in small details.

5 Evaluations

Problem 2:

It is not guaranteed that the model chosen based on the validation set
will give the best performance on the test set.
The validation set might be very small and not representing well all of the examples space.
When using the validation set to choose hyperparameter values,
we make the basic assumption that the validation set is not used for training.

6 Transfer learning

Problem 3:

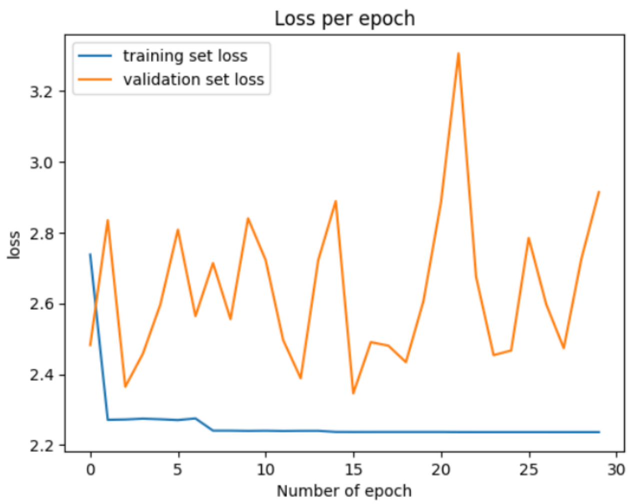
We chose the fine-tuning transfer learning method, using the pre-trained ResNet model.The ResNet model has been trained on a large dataset with various images, which helps it learn general features and patterns. This method is better for our purpose because we want to change the models parameters to be more suitable for the SVHN dataset and to our classification problem

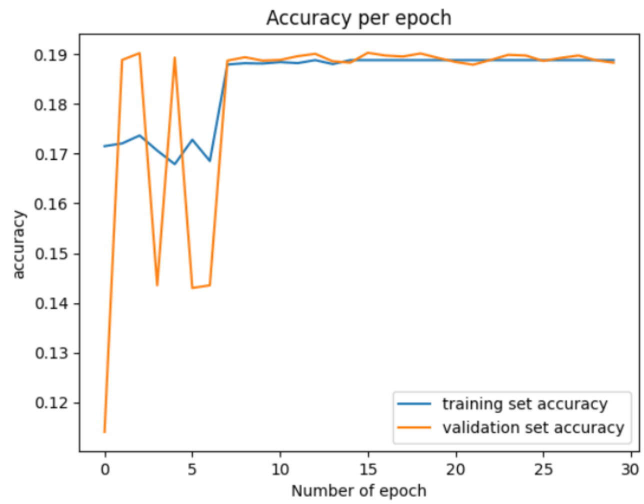
Problem 4:

In the fine-tuning transfer learning method,the starting point of finding the parameters is better because the model is pre-trained. Its more likely that we will find a global maximum instead of a local one.

Problem 5:

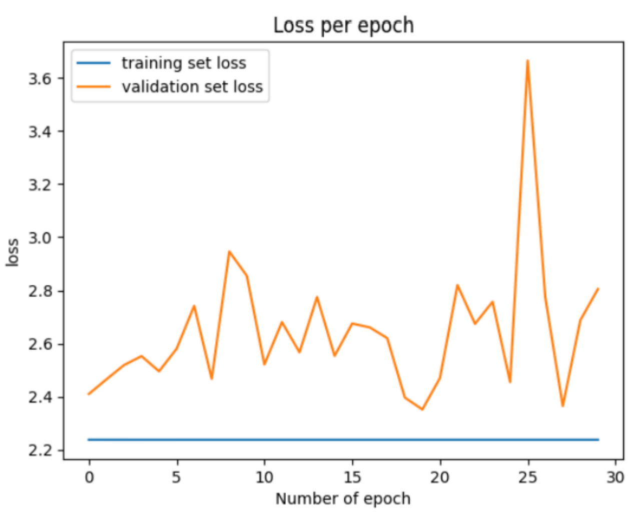
Loss and accuracy for ResNet with leaning rate 1:

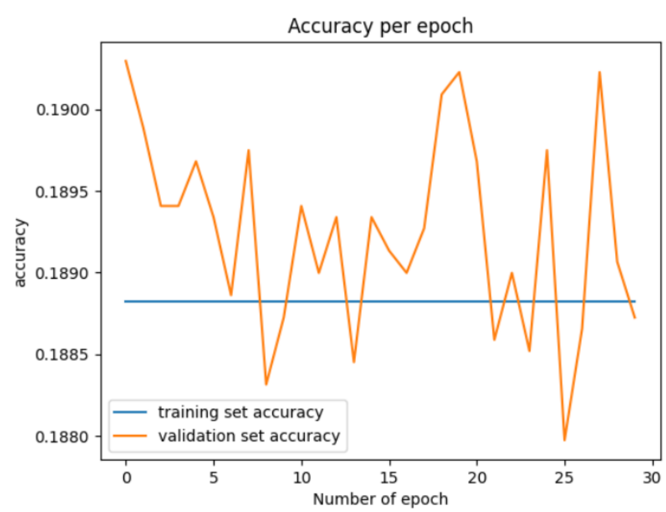




We can see that the increase in the accuracy over the epochs was not significant ,which could be because the net might be skipping the minimum due to taking overly large steps during training.
This is the same reason why the loss did not change significantly during the training process.

Loss and accuracy for ResNet with leaning rate 0.00001:





The reason why we didn't see a significant improvement in accuracy is that the network was taking small steps during training and couldn't reach the optimal solution effectively.
This resulted in a higher loss throughout the training process.

In [1]:
```python
import numpy as np
import torch
import torchvision
import matplotlib.pyplot as plt
from time import time
from torchvision import datasets, transforms
from torch import nn, optim
import torch.nn.functional as F
from tqdm import tqdm
import random
from torchsummary import summary
from torch.optim import lr_scheduler
import time
import os
import copy

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
os.environ['CUDA_LAUNCH_BLOCKING'] = '1'
```

In [ ]:
```python
# *** Computer Exercise 1 *** #
```

In [3]:
```python
# Loading the data

# Torch/PIL transformations which are applied to each image in the dataset
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,), (0.5,)),
                                ])

batch_size = 64

initial_trainset = torchvision.datasets.SVHN(root='./data', split="train",
                                        download=True, transform=transform)

testset = torchvision.datasets.SVHN(root='./data', split="test",
                                       download=True, transform=transform)


classes = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9')
```

```
Using downloaded and verified file: ./data/train_32x32.mat
Using downloaded and verified file: ./data/test_32x32.mat
```

In [5]:
```python
# Splitting into train and validation

generator = torch.Generator().manual_seed(42)
trainset, validationset = torch.utils.data.random_split(initial_trainset, [0.8, 0.2], generator=generator)
```

In [6]:
```python
# Creating data loaders for train, validation and test sets

trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)

validationloader = torch.utils.data.DataLoader(validationset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)
```

In [26]:
```python
def normalize_image(image):
  image_min = image.min()
  image_max = image.max()
  image_normalized = (image - image_min) / (image_max - image_min)
  return image_normalized

def display_images(dataloader):
  # get a single batch
  dataiter = iter(dataloader)
  images, labels = next(dataiter)

  figure = plt.figure()
  num_of_images = 60
  for index in range(1, num_of_images + 1):
      img = np.transpose(images[index].numpy(), (1,2,0))
      img_normalized = normalize_image(img)
      plt.subplot(6, 10, index)
      plt.axis('off')
      plt.imshow(img_normalized)
```

```
    print(labels[1:61].reshape((6,10)))
    print("------------------------------------------------")
    plt.show()
```

In [27]:
```
# Displaying some images from a batch of the train, test and validation set

print("Images and labels from train set:")
display_images(trainloader)
```

```
Images and labels from train set:
tensor([[2, 2, 4, 8, 0, 1, 8, 1, 8, 2],
        [5, 6, 0, 2, 1, 7, 8, 7, 8, 2],
        [3, 5, 2, 1, 7, 1, 6, 2, 4, 4],
        [4, 1, 4, 4, 5, 1, 2, 4, 9, 8],
        [9, 0, 3, 0, 1, 4, 2, 1, 7, 4],
        [6, 4, 3, 7, 7, 4, 2, 2, 7, 1]])
------------------------------------------------
```

In [28]:
```
print("Images and labels from test set:")
display_images(testloader)
```

```
Images and labels from test set:
tensor([[2, 1, 0, 6, 1, 9, 1, 1, 8, 3],
        [6, 5, 1, 4, 4, 1, 6, 3, 4, 2],
        [0, 1, 3, 2, 5, 4, 1, 4, 2, 8],
        [3, 8, 6, 0, 1, 5, 1, 1, 2, 9],
        [1, 6, 9, 2, 6, 1, 2, 0, 6, 9],
        [1, 5, 1, 9, 8, 1, 5, 1, 5, 2]])
------------------------------------------------
```

In [29]:
```
print("Images and labels from validation set:")
display_images(validationloader)
```

```
Images and labels from validation set:
tensor([[5, 3, 7, 2, 4, 1, 9, 3, 5, 1],
        [6, 5, 9, 1, 2, 4, 3, 5, 4, 1],
        [1, 2, 8, 6, 8, 4, 8, 3, 0, 9],
        [9, 3, 5, 0, 0, 0, 1, 1, 3, 3],
        [0, 1, 1, 2, 1, 4, 7, 0, 1, 5],
        [4, 5, 4, 5, 2, 7, 6, 3, 6, 2]])
------------------------------------------------
```

In [ ]:

```
# *** Computer Exercise 2 *** #
```

In [7]:

```
# Fully-connected network

input_size = 3072
hidden_sizes = [128, 64]
output_size = 10

class Net(nn.Module):
    def __init__(self, hidden_size1, hidden_size2):
      super(Net, self).__init__()
      self.input = nn.Linear(input_size, hidden_size1)
      self.hidden1 = nn.Linear(hidden_size1, hidden_size2)
      self.hidden2 = nn.Linear(hidden_size2, output_size)

    # x represents our data
    def forward(self, x):
      x = self.input(x)
      x = F.relu(x)
      x = self.hidden1(x)
      x = F.relu(x)
      x = self.hidden2(x)
      output = x
      return output


# Defining the first FC network, with hidden layers of sizes 128 and 64
fcNet = Net(hidden_sizes[0], hidden_sizes[1]).to(device)
summary(fcNet, input_size=(1, input_size))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1               [-1, 1, 128]         393,344
            Linear-2                [-1, 1, 64]           8,256
            Linear-3                [-1, 1, 10]             650
================================================================
Total params: 402,250
Trainable params: 402,250
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 0.00
Params size (MB): 1.53
Estimated Total Size (MB): 1.55
----------------------------------------------------------------
```

In [32]:
```python
# Convolutional network

class ConvNet(nn.Module):

    def __init__(self, kernel_size):
        super(ConvNet, self).__init__()
        # Conv2d(in_channels, out_channels, kernel_size)


        self.conv1 = nn.Conv2d(in_channels=3, out_channels=10, kernel_size=kernel_size, stride=1, padding=1)
        self.conv3 = nn.Conv2d(in_channels=10, out_channels=20, kernel_size=kernel_size, stride=1, padding=1
        # FC layers
        #  input to the FC layer = #output_features of the second conv layer
        self.fc1 = nn.Linear(1280, 64)
        self.fc2 = nn.Linear(64, 10)

    def forward(self, x):
        x = F.max_pool2d(input=F.relu(self.conv1(x)), stride=2, padding=0, kernel_size=2)
        x = F.max_pool2d(input=F.relu(self.conv3(x)), stride=2, padding=0, kernel_size=2)
        x = torch.flatten(x, 1) # flatten all dimensions except the batch dimension
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Defining the first CNN, with kernel size 3 for the convolutional layers
convNet = ConvNet(kernel_size=3).to(device)
print(convNet)
```

```
ConvNet(
  (conv1): Conv2d(3, 10, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(10, 20, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=1280, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=10, bias=True)
)
```

In [ ]:
```python
# *** Computer Exercise 3 *** #
```

In [9]:
```python
dataloaders = {'train': trainloader, 'val': validationloader}
dataset_sizes = {'train': len(trainset), 'val':len(validationset)}
criterion = nn.CrossEntropyLoss()
```

In [10]:
```python
def train_model(model, criterion, optimizer, scheduler, is_fc_net, num_epochs=30):
    since = time.time()

    train_epoch_losses = torch.zeros(num_epochs)
    train_epoch_accs = torch.zeros(num_epochs)
    val_epoch_losses = torch.zeros(num_epochs)
    val_epoch_accs = torch.zeros(num_epochs)
    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0
    for epoch in range(num_epochs):
        # print('Epoch {}/{}'.format(epoch, num_epochs - 1))
        # print('-' * 10)

        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluate mode

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data.
            for inputs, labels in dataloaders[phase]:
                if(is_fc_net):
                  inputs = inputs.view(inputs.shape[0], -1)
                inputs = inputs.to(device)
                labels = labels.to(device)

                # zero the parameter gradients
                optimizer.zero_grad()

                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == 'train'):
                    # logits: The raw predictions from the last layer
                    logits = model(inputs)
```

```
                    _, preds = torch.max(logits, 1)
                    # using CE criterion, thus input is the logits
                    loss = criterion(logits, labels)

                    # backward + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                # statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)
            if phase == 'train':
                scheduler.step()


            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects.double() / dataset_sizes[phase]


            if phase == 'train':
              # print('{} Loss: {:.4f} Acc: {:.4f}'.format(
              #     phase, epoch_loss, epoch_acc))
              train_epoch_losses[epoch] = epoch_loss
              train_epoch_accs[epoch] = epoch_acc
            else:
              val_epoch_losses[epoch] = epoch_loss
              val_epoch_accs[epoch] = epoch_acc

            # deep copy the model the best accuracy based on the validation set
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())


    time_elapsed = time.time() - since
    # print('Training complete in {:.0f}m {:.0f}s'.format(
    #     time_elapsed // 60, time_elapsed % 60))
    # #print('Best val Acc: {:4f}'.format(best_acc))

    # plot losses and accuracies for training and validation sets
    plt.title("Loss per epoch")
    plt.plot(list(range(num_epochs)), train_epoch_losses, label="training set loss")
    plt.plot(list(range(num_epochs)), val_epoch_losses, label="validation set loss")
    plt.xlabel("Number of epoch")
    plt.ylabel("loss")
    plt.legend()
    plt.show()

    plt.title("Accuracy per epoch")
    plt.plot(list(range(num_epochs)), train_epoch_accs, label="training set accuracy")
    plt.plot(list(range(num_epochs)), val_epoch_accs, label="validation set accuracy")
    plt.xlabel("Number of epoch")
    plt.ylabel("accuracy")
    plt.legend()
    plt.show()

    # load best model weights
    model.load_state_dict(best_model_wts)
    return model
```

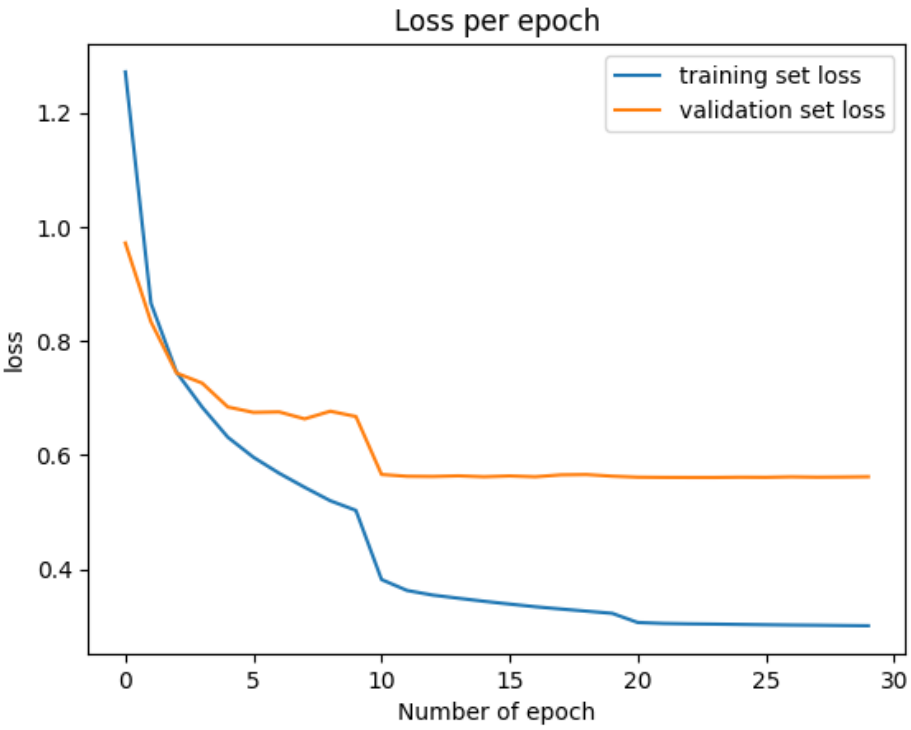
In [11]:

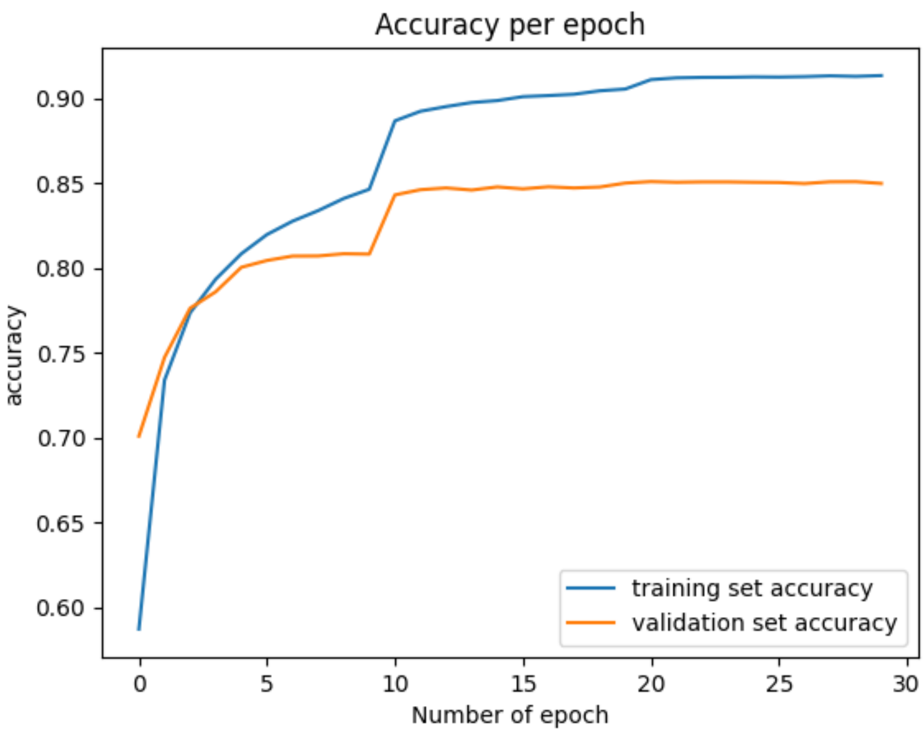
```
def call_train_model(net, PATH, is_fc_net):
  optimizer = optim.Adam(params=net.parameters(), lr=0.001)
  exp_lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)
  net = train_model(net, criterion, optimizer, exp_lr_scheduler, is_fc_net, num_epochs=30)
  torch.save(net.state_dict(), PATH)
  return net
```

In [30]:

```
# Training the FC network (with hidden layers of size 128 and 64)
print("Loss and accuracy for the first FC network:")
fcNet = call_train_model(fcNet, './fcNet.pth', True)
```

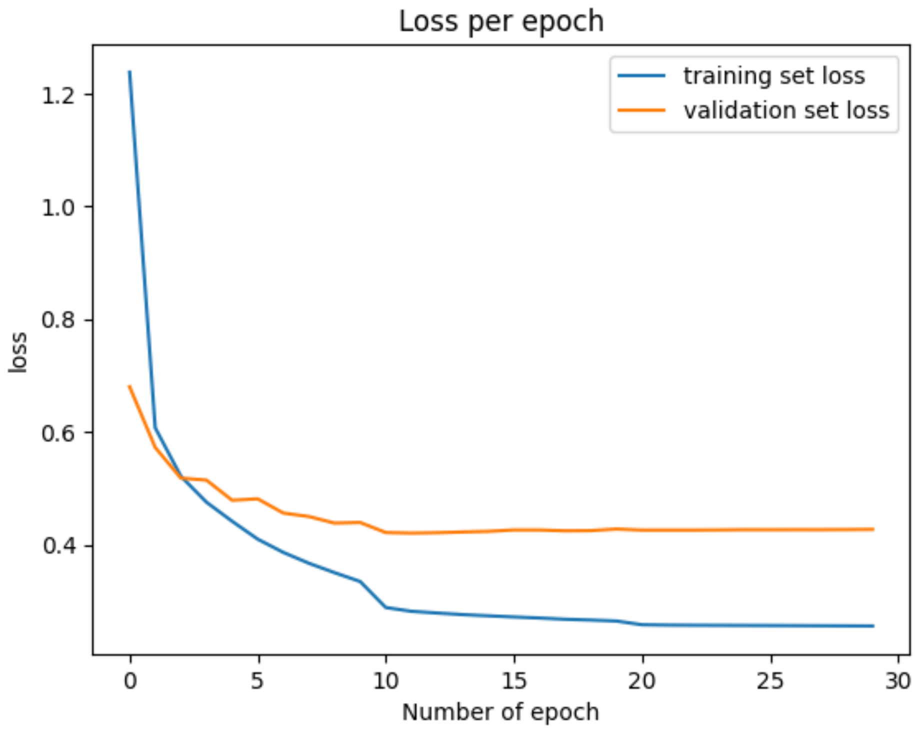
```
Loss and accuracy for the first FC network:
```

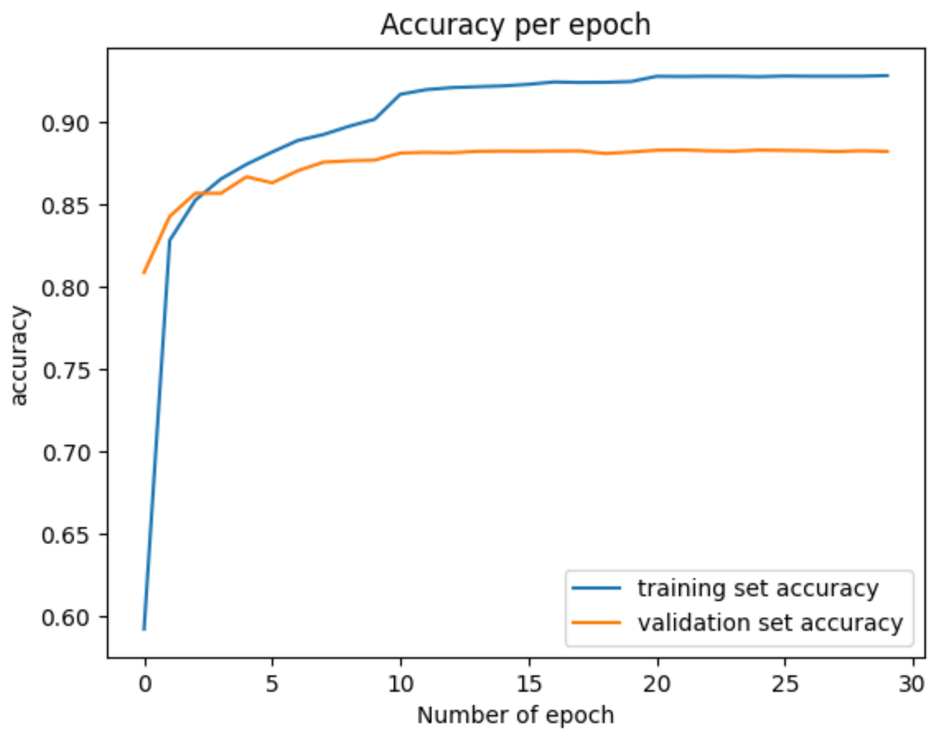




In [33]:
```
# Training the CNN (with kernel size 3)
print()
print("Loss and accuracy for the CNN:")
convNet = call_train_model(convNet, './convNet.pth', False)
```

```
Loss and accuracy for the CNN:
```





In [ ]:

```
# *** Computer Exercise 4 *** #
```

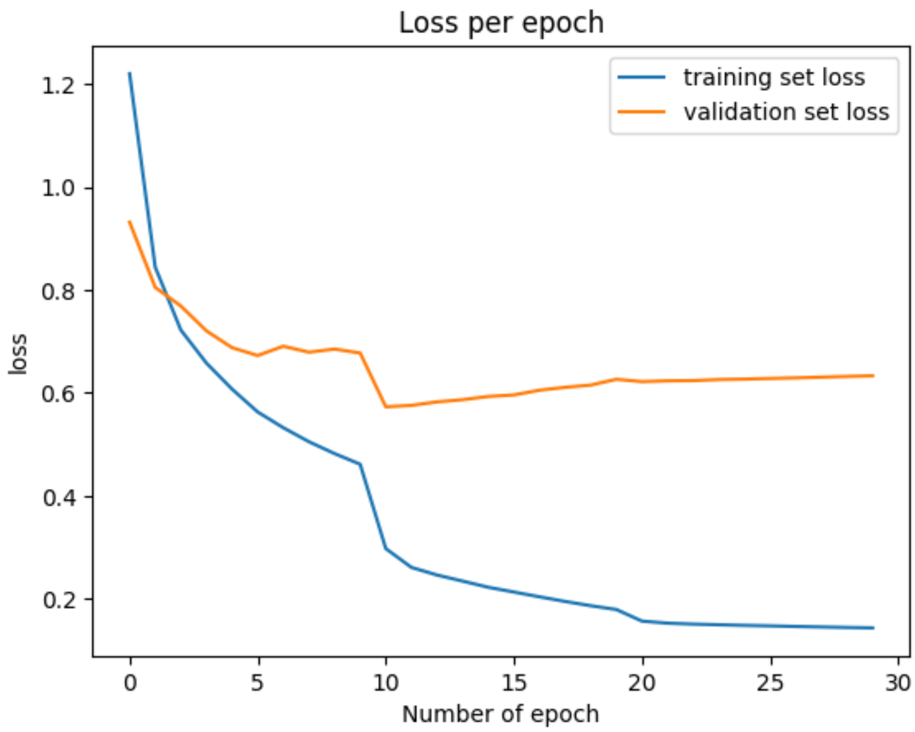
In [34]:

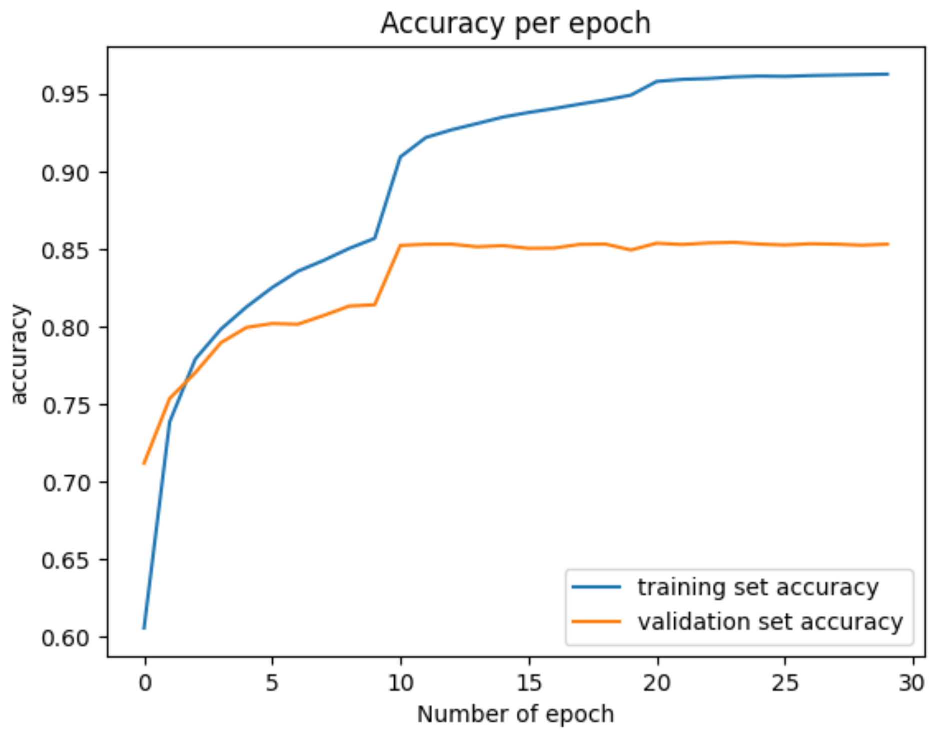
```
# Defining the second and third FC networks
# fcNet2 will be the name of the FC network with hidden layers of sizes 512 and 256
hidden_sizes2 = [512, 256]
fcNet2 = Net(hidden_sizes2[0], hidden_sizes2[1]).to(device)

# fcNet3 will be the name of the FC network with hidden layers of sizes 64 and 32
hidden_sizes3 = [64, 32]
fcNet3 = Net(hidden_sizes3[0], hidden_sizes3[1]).to(device)

# Defining the second and third CNNs
#convNet2 will be the name of the CNN with kernel size 10 for the convolutional layers
convNet2 = ConvNet(kernel_size=10).to(device)

#convNet3 will be the name of the CNN with kernel size 1 for the convolutional layers
convNet3 = ConvNet(kernel_size=1).to(device)
```

In [35]:

```
# Training the second FC network (with hidden layers of size 512 and 256)
print("Loss and accuracy for the second FC network:")
fcNet2 = call_train_model(fcNet2, './fcNet2.pth', True)
```

```
Loss and accuracy for the second FC network:
```

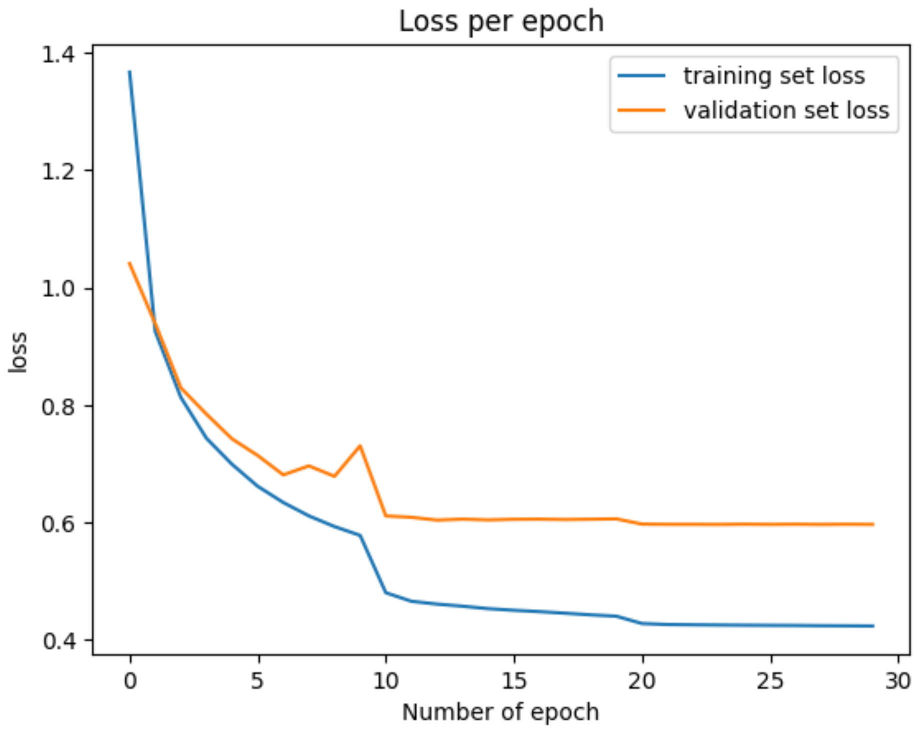


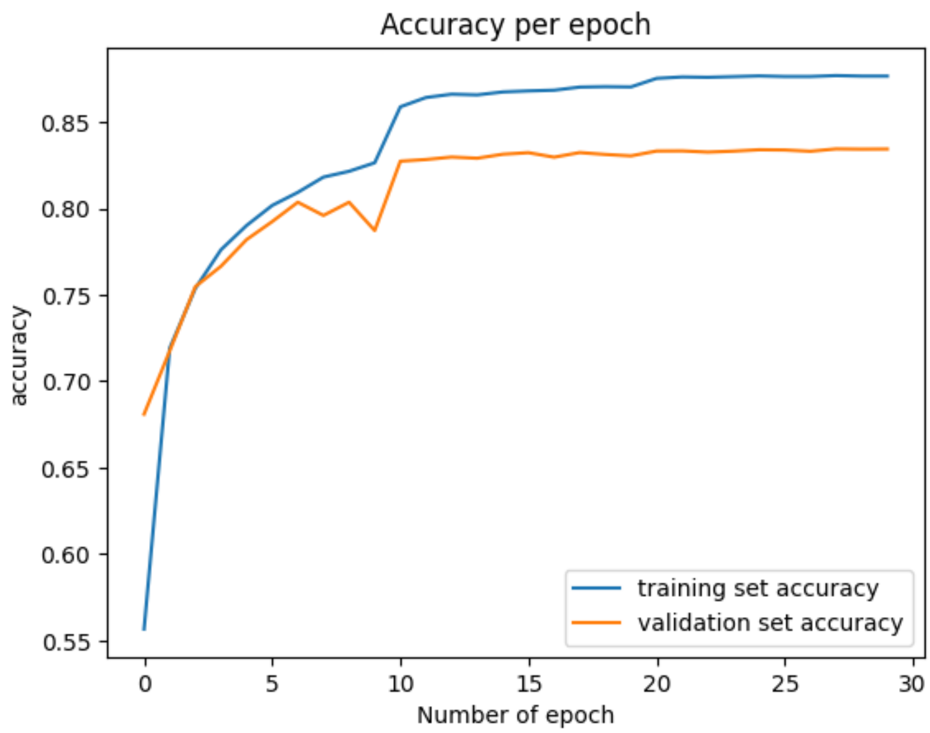


In [36]:

```
# Training the third FC network (with hidden layers of size 64 and 32)
print("Loss and accuracy for the third FC network:")
fcNet3 = call_train_model(fcNet3, './fcNet3.pth', True)
```

```
Loss and accuracy for the third FC network:
```

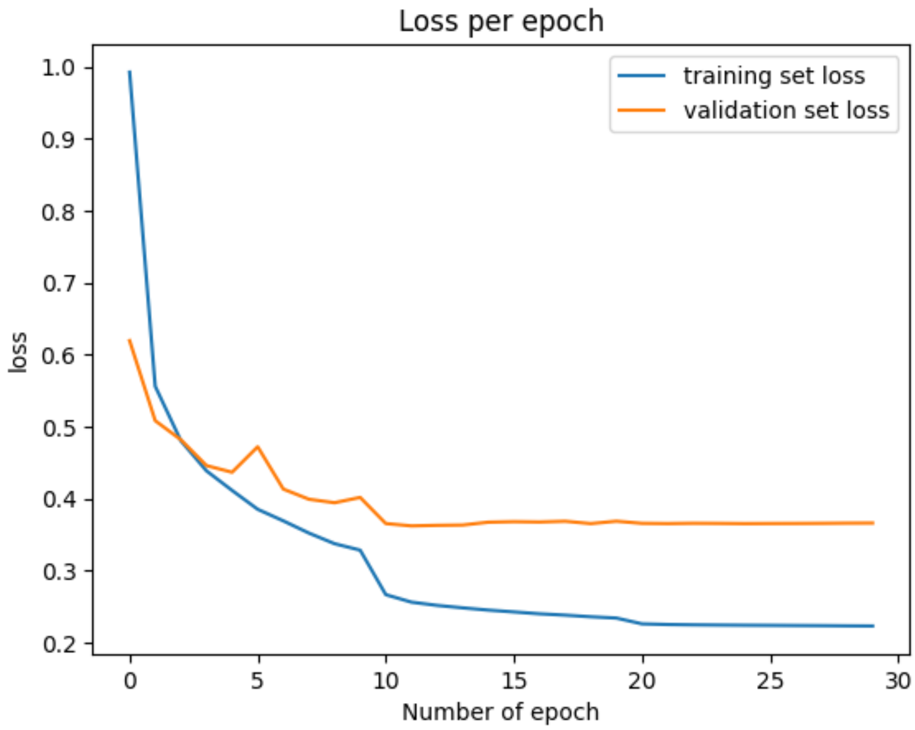


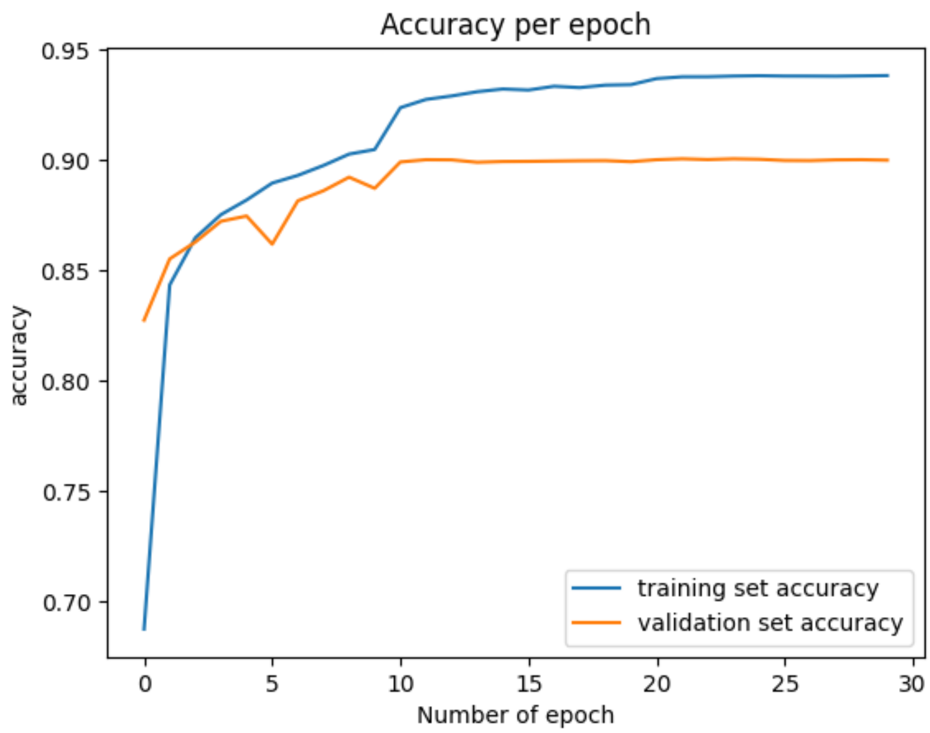


In [ ]:

```
# Training the second CNN (with kernel size 10)
print("Loss and accuracy for the second CNN:")
convNet2 = call_train_model(convNet2, './convNet2.pth', False)
```

```
Loss and accuracy for the second CNN:
```

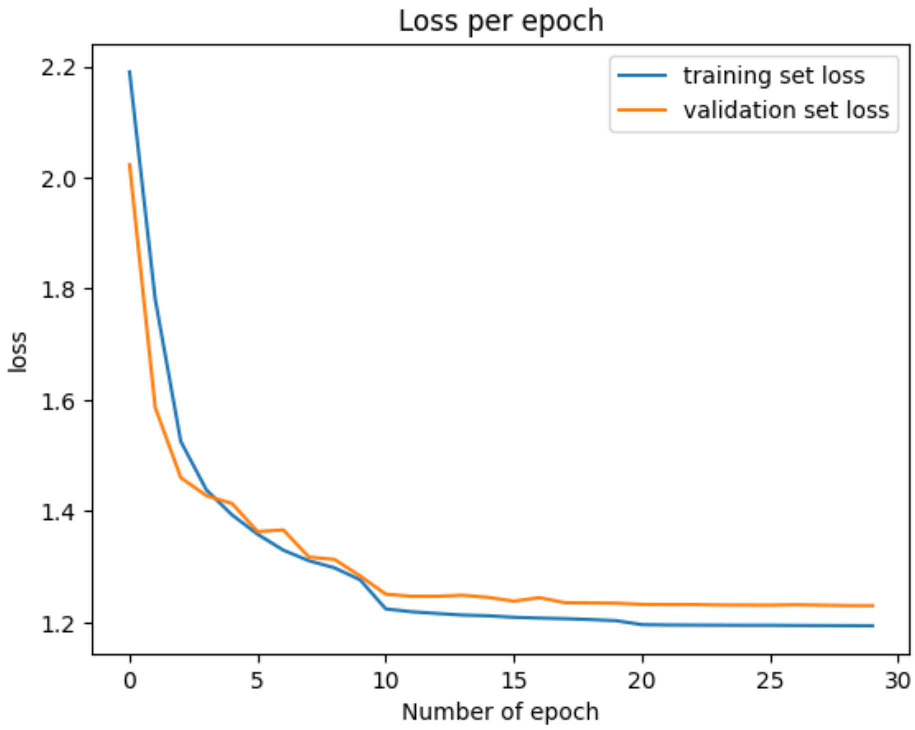


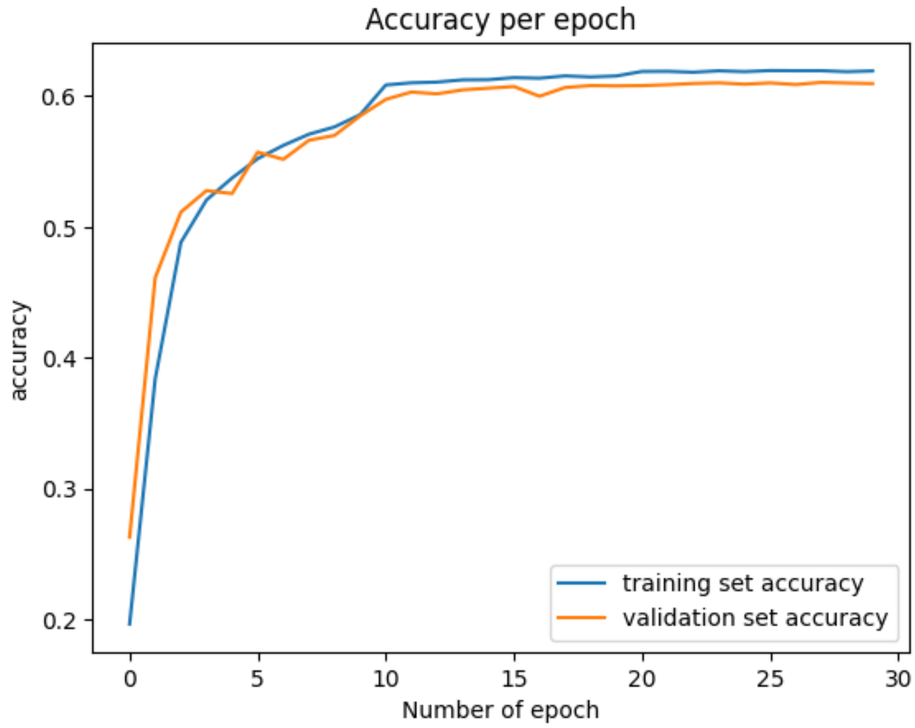


In [ ]:

```
# Training the third CNN (with kernel size 1)
print("Loss and accuracy for the third CNN:")
convNet3 = call_train_model(convNet3, './convNet3.pth', False)
```

```
Loss and accuracy for the third CNN:
```





In [ ]:
```
# *** Computer Exercise 5 *** #
```

In [12]:
```
def eval_model(model, dataloader, is_fc_net):
  correct_count, all_count = 0, 0
  model.eval()
  with torch.no_grad():
    for inputs, labels in dataloader:
        if(is_fc_net):
          inputs = inputs.view(inputs.shape[0], -1)
        inputs = inputs.to(device)
        labels = labels.to(device)
        outputs = model(inputs)
        _, preds = torch.max(outputs, 1)
        correct_pred = torch.eq(labels, preds).cpu()
        correct_count += correct_pred.numpy().sum()
        all_count += len(labels)

  print("Number Of Images Tested =", all_count)
  print("\nModel Accuracy =", (correct_count/all_count))
```

In [20]:
```
convNet2.load_state_dict(torch.load('./convNet2.pth'))
# Testing accuracy of convNet2, whose accuracy was the highest, on the test set
```

```
print("Accuracy of the second CNN on the test set:")
eval_model(convNet2, testloader, False)
```

```
Accuracy of the second CNN on the test set:
Number Of Images Tested = 26032

Model Accuracy = 0.8863322065150584
```

In [ ]:

```
# *** Computer Exercise 6 *** #
```

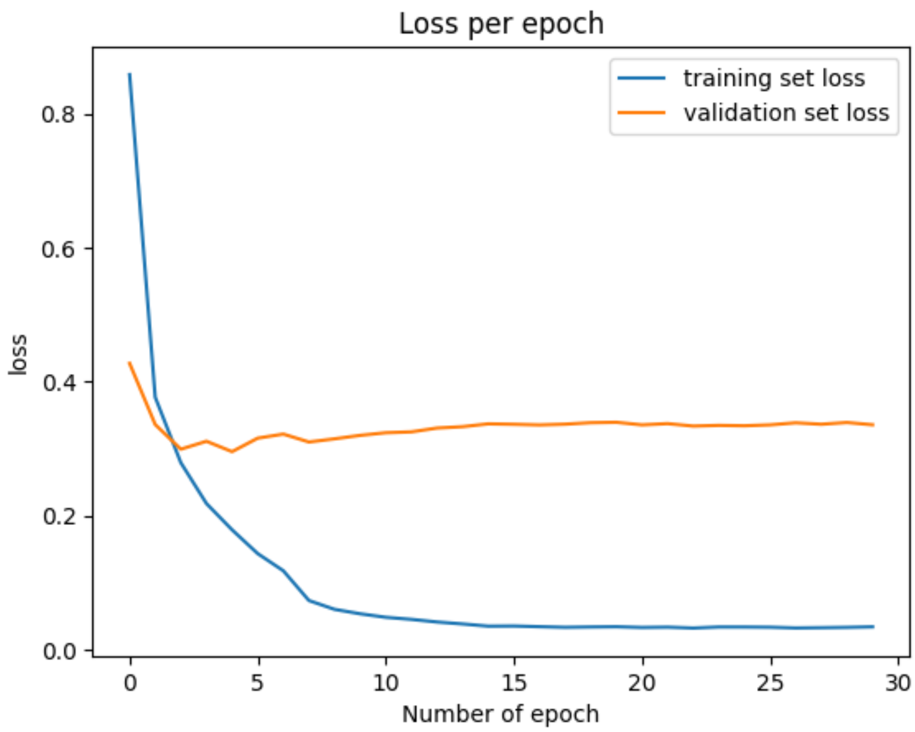
In [22]:

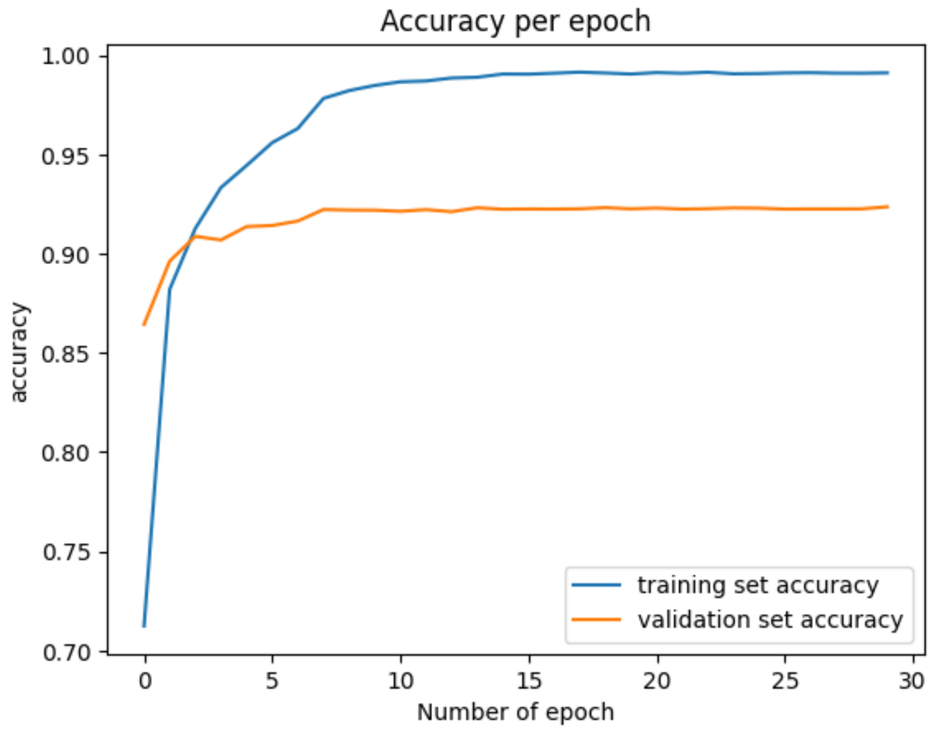
```
# Fine tuning ResNet

model_ft = torchvision.models.resnet18(pretrained=True)
num_ftrs = model_ft.fc.in_features
model_ft.fc = nn.Linear(num_ftrs, 10)
model_ft = model_ft.to(device)
```

In [23]:

```
optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.001, momentum=0.9)
# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7, gamma=0.1)

print("Loss and accuracy for ResNet:")
model_ft = train_model(model_ft, criterion, optimizer_ft, exp_lr_scheduler, False, 30)
PATH = 'model_resnet_ft.pth'
torch.save(model_ft.state_dict(), PATH)
```

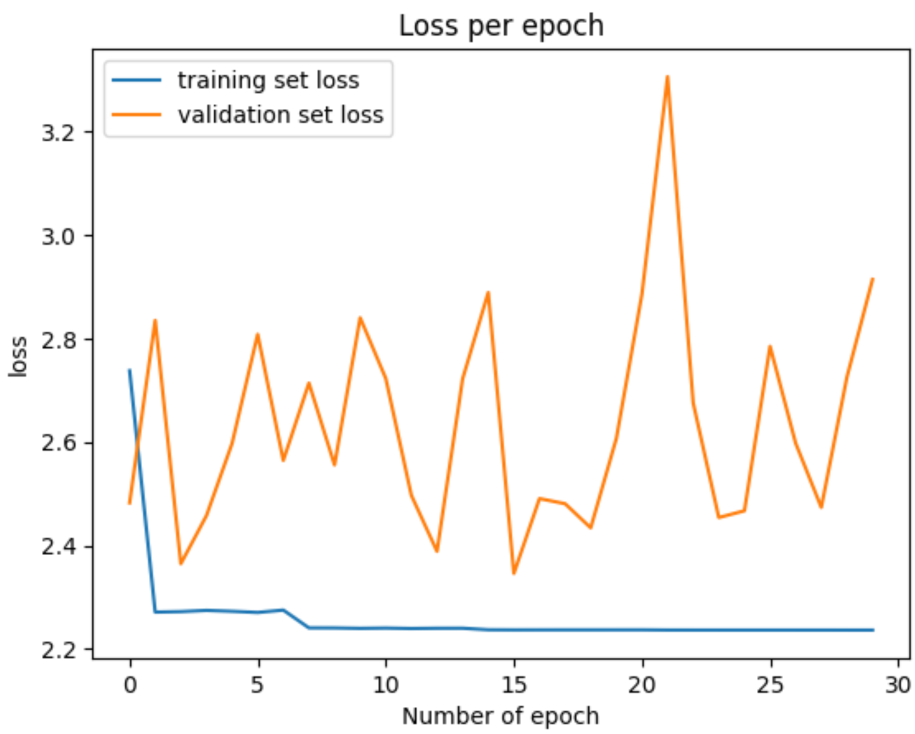
```
Loss and accuracy for ResNet:
```

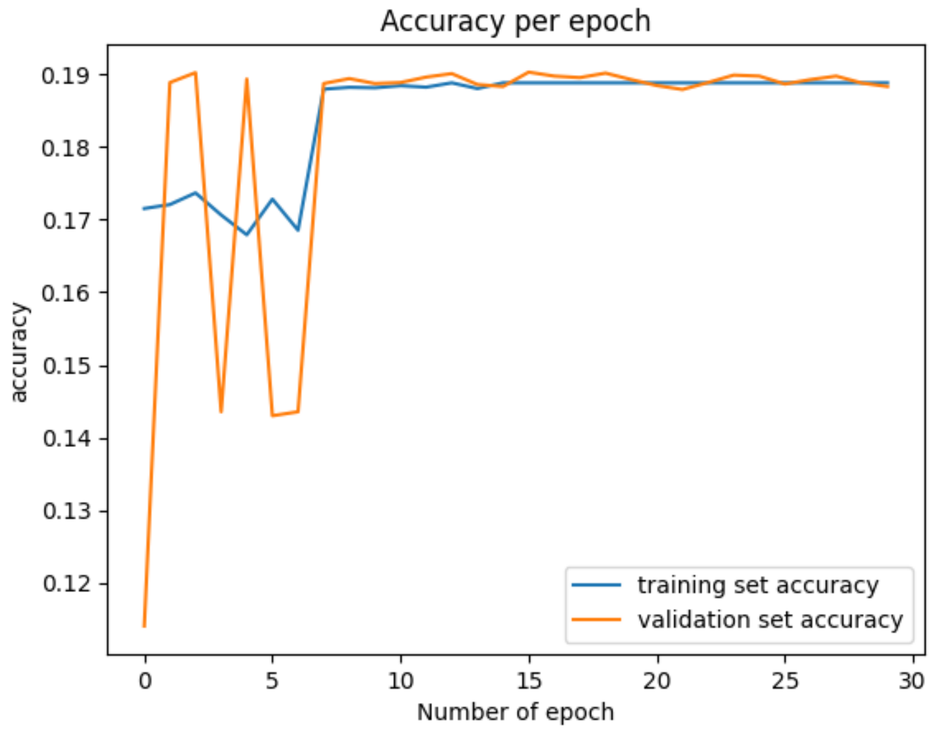




In [24]:
```python
# Learning rate 1
optimizer_ft = optim.SGD(model_ft.parameters(), lr=1, momentum=0.9)
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7, gamma=0.1)

print("Loss and accuracy for ResNet with leaning rate 1:")
model_ft = train_model(model_ft, criterion, optimizer_ft, exp_lr_scheduler, False, 30)
PATH = 'model_resnet_ft_lr1.pth'
torch.save(model_ft.state_dict(), PATH)
```

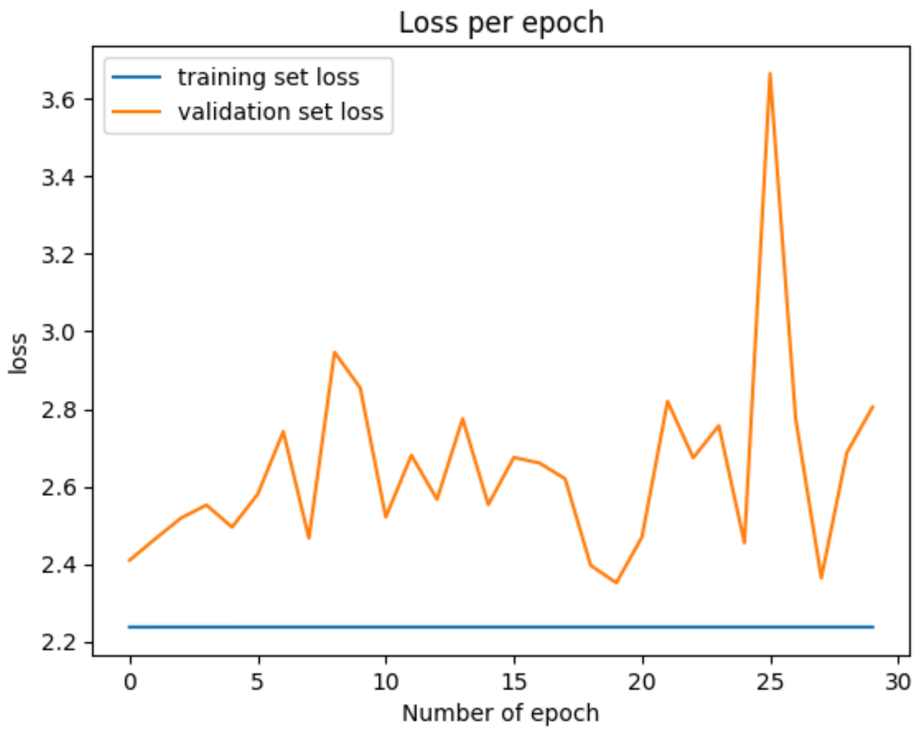
```
Loss and accuracy for ResNet with leaning rate 1:
```

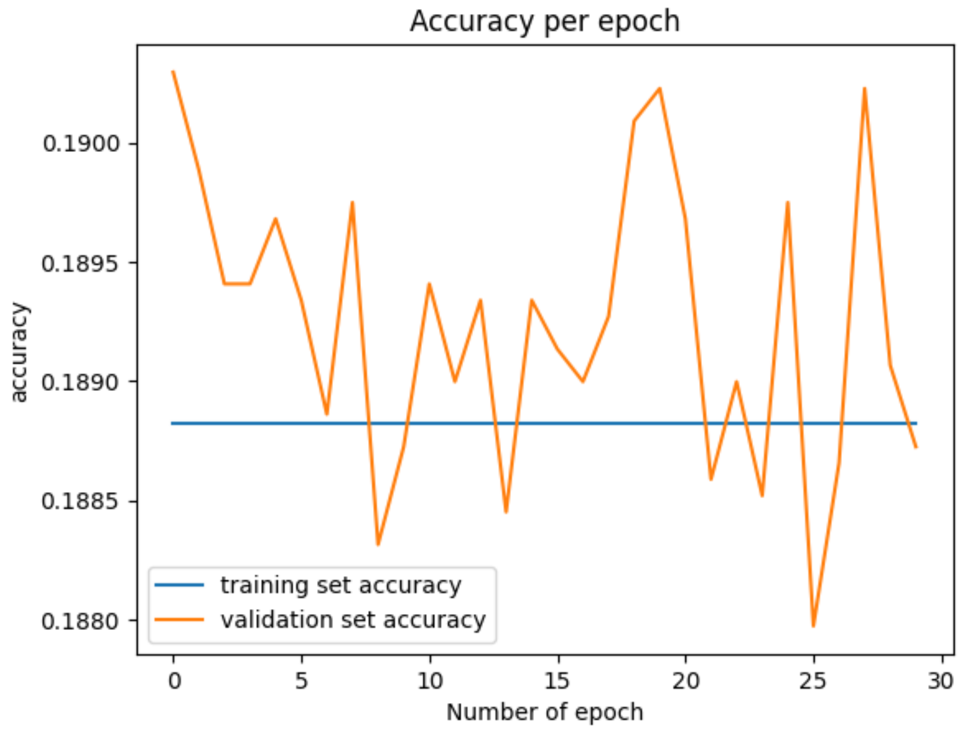




In [25]:
```
# Learning rate 0.00001
optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.00001, momentum=0.9)
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7, gamma=0.1)

print("Loss and accuracy for ResNet with leaning rate 0.00001:")
model_ft = train_model(model_ft, criterion, optimizer_ft, exp_lr_scheduler, False, 30)
PATH = 'model_resnet_ft_lr000001.pth'
torch.save(model_ft.state_dict(), PATH)
```

```
Loss and accuracy for ResNet with leaning rate 0.00001:
```





Accuracy per epoch
0.1900
0.1895
0.1890
0.1885
0.1880
accuracy
0
5
10
15
20
25
30
Number of epoch
training set accuracy
validation set accuracy